Title
Efficient, Scalable and High-Throughput Runtime Reconfigurable Arrays for Accelerator as a Service

Author
Nagi, Sumeet Singh

Publication Date
2022

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Efficient, Scalable and High-Throughput Runtime Reconfigurable Arrays for Accelerator as a Service

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical and Computer Engineering

by

Sumeet Singh Nagi

2022

ABSTRACT OF THE DISSERTATION

Efficient, Scalable and High-Throughput Runtime Reconfigurable Arrays
for Accelerator as a Service

by

Sumeet Singh Nagi
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2022
Professor Dejan Marković, Chair

Advancements in silicon processing are responsible for the exponential growth in computing performance and algorithmic development. With the end of Dennard scaling, conventional computing architectures, like CPU, are unable to keep up with the increasing computation requirements of modern algorithms. Hardware accelerators are designed for each such computation-heavy algorithm and incorporated into the system; a modern System-On-Chip (SoC) for phones can have up to 30 different accelerators. Modern high-compute applications such as 5G, machine learning, and autonomous driving vehicles require accelerators to keep up with their rapidly evolving standards and computation needs. However, with the rising design costs at newer technology nodes, the iterative development of inflexible accelerators becomes prohibitively expensive. Reconfigurable architectures, with their ability to adapt to rapidly-evolving standards as well as their ability to accommodate several such high-performance applications in the system, provide an ideal solution. The motivation of this dissertation is to develop such a Coarse Grain Reconfigurable Architecture called Universal Digital Signal Processor (UDSP) which could replace accelerator blocks in an SoC, and develop a hardware management system to enable concurrent multiprogram functionalities in the reconfigurable architectures. UDSP consists of 196 Compute Elements (CEs)

and a statistics-based scalable, delayless, high speed routing network. It is developed using an algorithm-driven framework to allow for faster development of each successive revision of the design. The tileable and scalable nature of UDSP allowed us to put together 4 UDSP dies on a $10\mu m$ fine-pitch interposer Silicon Interconnect Fabric, as a 2×2 UDSP Multi-Chip Module (MCM), quadrupling the number of compute resources. The UDSP 2×2 assembly has a peak throughput of 3,450 Giga-Operations per second (GOPs) or 1,725 Giga-Multiply Accumulates per second (GMACs) at 1.1GHz clock frequency while consuming 6W power including 0.38pJ/bit to transfer data across dies in TSMC 16nm. It achieves a peak efficiency of 785GMACs/J (0.42V, 315MHz). UDSP lies within 4.2× energy efficiency and 6.4× area efficiency gap relative to ASICs at nominal operation conditions (0.8V, 1.1GHz).

Multiprogram tenancy on conventional reconfigurable arrays requires high manual effort from the programmer to foresee and account for runtime program dynamics during compilation. The inability to predict runtime and multiprogram dynamics places the recompilation time of programs in the critical timing path, leading to long reconfiguration times, poor active resource utilization, and low acceleration performance. We developed an active hardware resource management system for reconfigurable arrays that automatically accounts for multi-program dynamics at runtime, eases the workload of the programmer, and improves the array's performance. These hardware management techniques enable dynamic runtime relocation of programs on the Runtime Reconfigurable Array (RTRA) with minimal reconfiguration latency overhead, which allows the array to offer Accelerator as a Service (ACAS). The ACAS architecture virtualizes the array by spatially and temporally scheduling multiple programs on its available resources, thus achieving higher active utilization for the mapped programs on the array. ACAS allows developers to compile programs for acceleration on reconfigurable array without requiring additional manual steps for runtime resource planning at compile time. Provided with high program pressure, ACAS exceeds 90% active utilization of arrays. For signal processing workloads, our simulated 9×12 RTRA uses a 3× smaller area and delivers $3.2-4.3\times$ more throughput than a 18×18 UDSP and the 18×18 RTRA delivers $8 - 14\times$ more throughput as compared to its equivalent-sized 18×18 UDSP counterpart.

The dissertation of Sumeet Singh Nagi is approved.

Anthony John Nowatzki

Chih-Kong Ken Yang

Gregory J. Pottie

Subramanian Srikantes Iyer

Dejan Marković, Committee Chair

University of California, Los Angeles

2022

To my parents and siblings

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

It's been a long road as a graduate student at UCLA for the past 7 years. As this chapter of my life comes to a close, I wish to thank the people who have helped me on this journey. I thank my advisor, Professor Dejan Marković, for guiding me in my research. He gave me the freedom and resources to explore research topics that piqued my interest. This was a once in a lifetime opportunity, for which I will always be very grateful.

I am also very grateful to Professor Tony Nowatzki and Professor Subramanian Iyer. Over the years, I have learnt a lot from the graduate and undergraduate courses that they teach at UCLA and from many lengthy discussions I have had with them at various stages of my research. Their support and encouragement has been invaluable and vital for my work, and I wholeheartedly thank them.

I thank Professors Ken Yang, Greg Pottie, Yuval Tamir, Puneet Gupta, Danijela Cabric, Hooman Darabi, and Achuta Kadambi for their wonderful graduate and undergraduate courses. Attending their courses at UCLA has really enhanced my understanding of circuits, communication systems, and signal and image processing. I was introduced to digital circuits during my undergraduate studies by Professor Amit Sethi and Gaurav Trivedi at IIT-Guwahati, India. Their enthusiasm for the subject and the courses inspired me to continue my research in circuits. I thank them for setting me on this journey.

At UCLA, I have had the privilege to be in the company of brilliant friends and colleagues. I have had numerous discussions with Uneeb Rathore, Steven Moran and Trevor Black regarding a wide range of topics, and my gratitude for them cannot be overstated. I am also grateful to the senior students from my lab Sina Basir-Kazeruni, Hariprasad Chandrakumar and Dejan Rozgic for their guidance and my fellow batch mates Wojciech Romaszkan, Saptadeep Pal, Sida Li, Wenhao Yu, Tim Ling, Siva Chandra Jangam, Krutikesh Sahoo, Sihao Liu and many others, for making this a rewarding journey.

The administrative staff at UCLA have been very supportive, and they have made the life of graduate students much easier. I thank Kyle Jung, Katie Christensen, Deeona Columbia,

Toan Nguyen, Mandy Smith, late Ryo Arreola, Julio Romero, Dayna Bilderback and Ylena Requena for all their hard work. I would also like to thank DARPA DRBE, CHIPS and CRAFT programs for funding my research, and UCLA graduate division for dissertation year fellowship during my dissertation year. I would also like to thank Boeing for partnership during the CRAFT program and people at Intel, Dr. Farhana Sheikh, Sergey Shumarayev and David Kehlet for a very educational internship.

I would like to thank my great set of friends that surrounded me and made my time in Los Angeles enjoyable. I would especially like to thank Dr. Hannah Carlan for being my biggest support, friend, and family in LA the past few years, for always inspiring and motivating me, and providing valuable feedback, and for proofreading all my papers.

Finally and most importantly, I thank my parents Daljit Kaur Nagi and Ishvinder Singh Nagi, and sisters Amanpreet Kaur Assal and Jasneet Kaur Nagi for their numerous sacrifices, unconditional love, unwavering support and encouragement. They have patiently stood by my side during this long journey, with all its ups and downs. This would not have been possible without them.

To quote Dr. Sheldon Cooper, a character from TV series The Big Bang Theory, "I was under a misapprehension that my accomplishments were mine alone. Nothing could be further from the truth. I have been encouraged, sustained, inspired, and tolerated not only by my family, but by the greatest group of friends anyone ever had."

Parts of the Chapter 2 were presented at the 2021 International Workshop on Signal Processing Systems (SiPS), titled "A Multi-Domain Architectural Efficiency Metric" [33]. Chapter 3 contains materials that were published and presented at 2022 IEEE International Solid-State Circuits Conference (ISSCC), titled "A 16nm 785GMACs/J 784-Core Digital Signal Processor Array with a Multilayer Switch Box Interconnect, Assembled as a 2 $\times$ 2 Dielet with 10μm-Pitch Inter-Dielet I/O for Runtime Multi-Program Reconfiguration" [40] and 2020 IEEE Electronic Components and Technology Conference (ECTC), titled "Demonstration of a Low Latency ($< 20ps$) Fine-Pitch ($\leq 10\mu m$) Assembly on the Silicon Interconnect Fabric" [21].

VITA

2014	Interim Engineering Intern, Standard Cell Library Group Qualcomm Inc., Bangalore, India
2015	B.Tech., Electronics and Communication Engineering Indian Institute of Technology Guwahati, India
2017	M.S., Electrical Engineering University of California, Los Angeles
2018-2021	Teaching Assistant, Electrical and Computer Engineering Department University of California, Los Angeles
2019	Analog Devices Outstanding Student Designer Award Analog Devices Inc.
2019	Hardware Design Graduate Intern, Intel CTO Office Intel, Santa Clara, CA
2021	Teaching Assistant, Computer Science Department University of California, Los Angeles
2021-2022	UCLA Dissertation Year Fellowship
2015-2022	Graduate Student Researcher, Electrical and Computer Engineering University of California, Los Angeles

PUBLICATIONS

U. Rathore*, S. S. Nagi*, S. Iyer, D. Marković, "A 16nm 785GMACs/J 784-Core Digital Signal Processor Array with a Multilayer Switch Box Interconnect, Assembled as a 2×2 Dielet

with $10\mu m$-Pitch Inter-Dielet I/O for Runtime Multi-Program Reconfiguration," Proceedings ISSCC '22 - International Solid-State Circuits Conference, 2022, pp. 52-53.

S. S. Nagi and D. Marković, "A Multi-Domain Architectural Efficiency Metric," 2021 IEEE Workshop on Signal Processing Systems (SiPS), 2021, pp. 265-270.

S. S. Nagi, D. Marković, "Run-Time Reconfigurable Architecture", US Provisional Patent Granted 63/223,787, August 2021.

S. Jangam, U. Rathore, S. Nagi, D. Marković and S. S. Iyer, "Demonstration of a Low Latency ($< 20ps$) Fine-pitch ($\leq 10\mu m$) Assembly on the Silicon Interconnect Fabric," 2020 IEEE 70th Electronic Components and Technology Conference (ECTC), 2020, pp. 1801-1805.

S. S. Nagi, U. Rathore, F. Sheikh, S. Weber, "Reconfigurable Digital Signal Processing (DSP) Vector Engine", US Patent Granted US20200225947 2020.

U. Rathore, S. Nagi, D. Marković, "Peta Scale Compute Fabric," UCLA CHIPS 2019 Poster.

U. Rathore, S. Nagi, J. Zhang, S. Basir, D. Marković, "Next Generation Dynamically Reconfigurable Universal Digital Signal Processor in 16nmFF," UCLA ARR 2017 Poster.

CHAPTER 1

Introduction

For decades, industry has relied upon exponential transistor scaling for performance improvements. This scaling has enabled the development of high-performance and energy-efficient applications. However, with the end of Dennard scaling [10] and decreasing returns from transistor scaling [12], [48], system designers are now incorporating specialized and domain-specific architectures [18] to allow for increasingly higher-performance algorithms as well as energy-efficient edge devices. An increasing number of hardware accelerator blocks are added to the SoCs along with multi-core CPUs/GPUs to balance the performance and energy efficiency requirements [43]. Although the performance gains from hardware accelerators are highly desirable, their inflexible, non-programmable nature makes them impractical for rapidly evolving applications such as 5G and Machine Learning (ML).

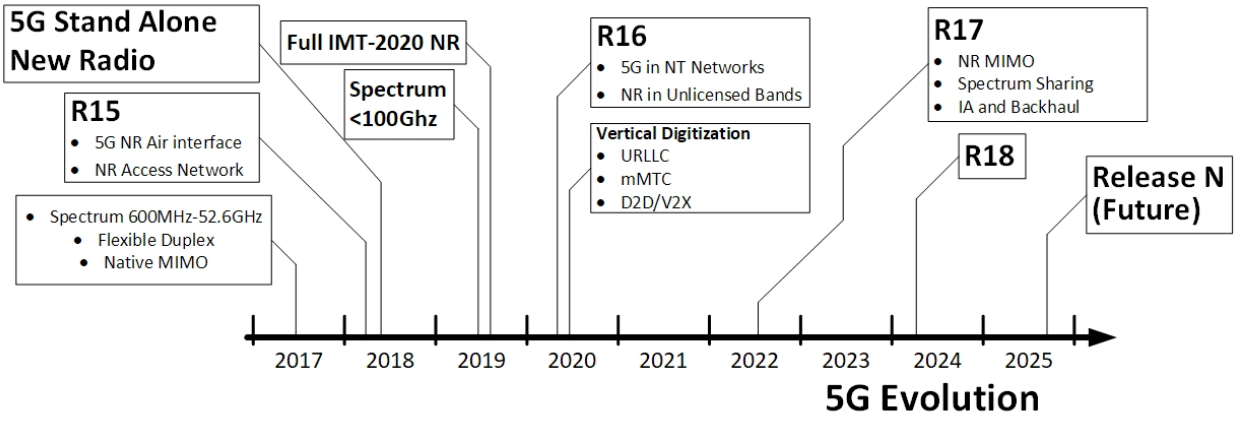


Figure 1.1: Advanced plan of 5G evolution by 3GPP [1].

An accelerator designed for 5G network infrastructure cannot follow the same strategy as 4G networks, as the 4G network architecture and interface was very well-defined. Due to

the diverse use cases, the complexity of networks, and rapidly-evolving standards as shown in Figure 1.1 [1], tackling the demands of 5G infrastructure with a hardware accelerator or a dedicated ASIC would require multiple design iterations [55]. In the field of ML, the model sizes and complexity of different algorithms nearly doubles every 3-4 months [44], thus requiring a new iterative hardware design to accommodate the algorithmic changes and increasing memory requirements.

However, the development costs of a new design in the advanced technology nodes is increasing exponentially as shown in Figure 1.2. This prohibits the iterative development of hardware accelerators for low volume requirements. With the rising design costs in smaller technology nodes, the volume requirements for the break-even cost of hardware accelerators as compared to reconfigurable/programmable solutions would keep increasing. Hence, there is a need to develop a cost-effective, energy and area-efficient reconfigurable solution which can provide the optimal flexibility required to avoid the iterative process of hardware accelerator development and which can keep pace with the evolving standards and complexities.

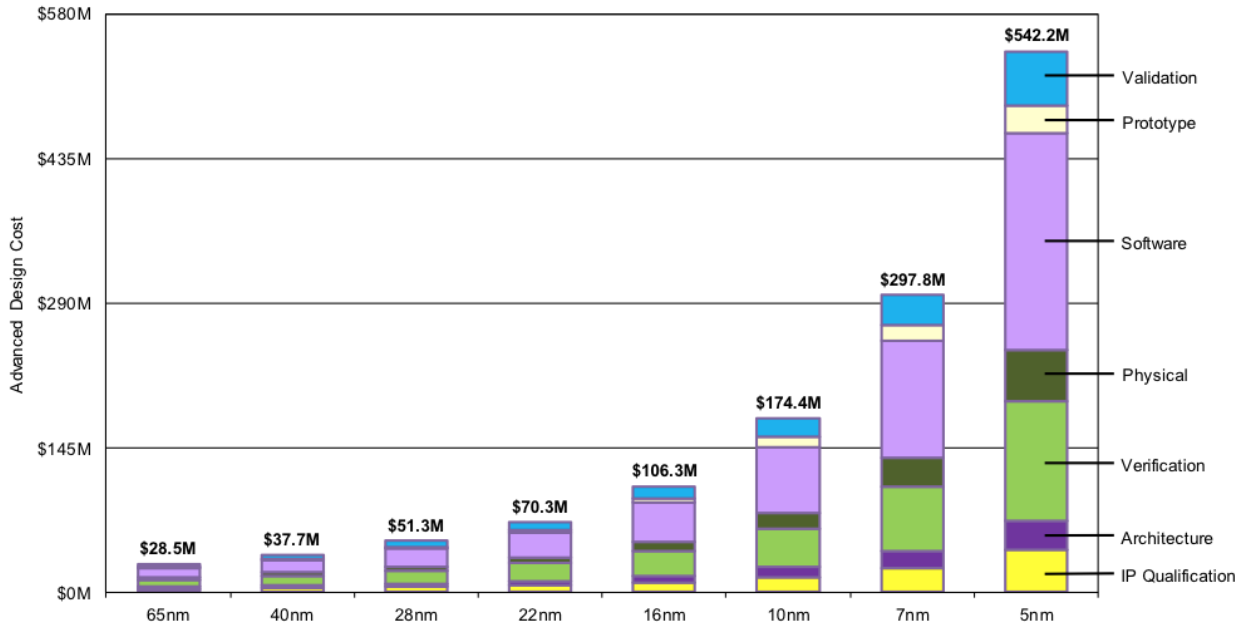


Figure 1.2: Design costs at advanced nodes [20].

Reconfigurable solutions offer several advantages [37] over hardware accelerators:

- **Reduced time-to-market:** Time-to-market for the reconfigurable solution is lower as the hardware can be reused across multiple platforms.

- **Higher chip volume:** Since a reconfigurable solution can be adapted to support multiple standards and applications without requiring hardware changes, the chip volumes of such solutions are higher relative to application-specific accelerators.

- **Prototyping:** It is easier and faster to fix any issues during the ongoing adoption and evolution of products and specifications on a reconfigurable solution with a simple software revision or a kernel update even after the chipset has been deployed into the systems.

- **Multi-mode operation:** A reconfigurable solution can be used to accelerate multiple applications from the broader domain, unlike hardware accelerators which are highly application-specific and can only accelerate a small subset of applications. An SoC block might require multiple hardware accelerators to achieve acceleration over the domain of desired applications.

For a single application system with evolving needs, the designer has to consider the total cost of ownership for the solution, whether it is an ASIC or a reconfigurable solution like Field Programmable Gate Arrays (FPGA) or Coarse Grain Reconfigurable Architectures (CGRA). Figure 1.3 shows the study of total cost of ownership of an ASIC, FPGA, and CGRA over the course of the product cycle and volume for 5G applications.

To address the challenges of constantly evolving application standards and increasing computation requirements, we developed a reconfigurable solution called Universal Digital Signal Processor (UDSP). Our design exploits the spatial correlation between the functional units (multipliers, adders, logic) and the interconnect network requirements in the domain of applications to design a 196 core UDSP which is 4.2× as energy efficient and 6.4× as area efficient as an ASIC, as shown in Figure 1.4. We then demonstrate the scalability and tileability of our architecture using fine-pitch interposer interconnect Si-IF [21].

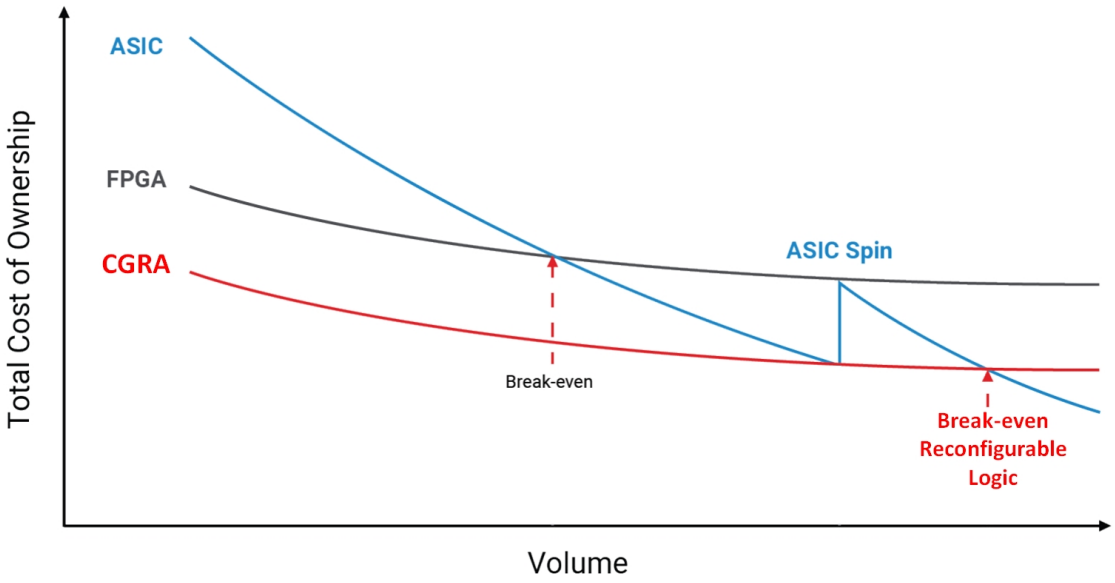


Figure 1.3: Total cost of ownership curve for ASIC, FPGA and CGRA. Adapted from Xilinx RFSoC [55].

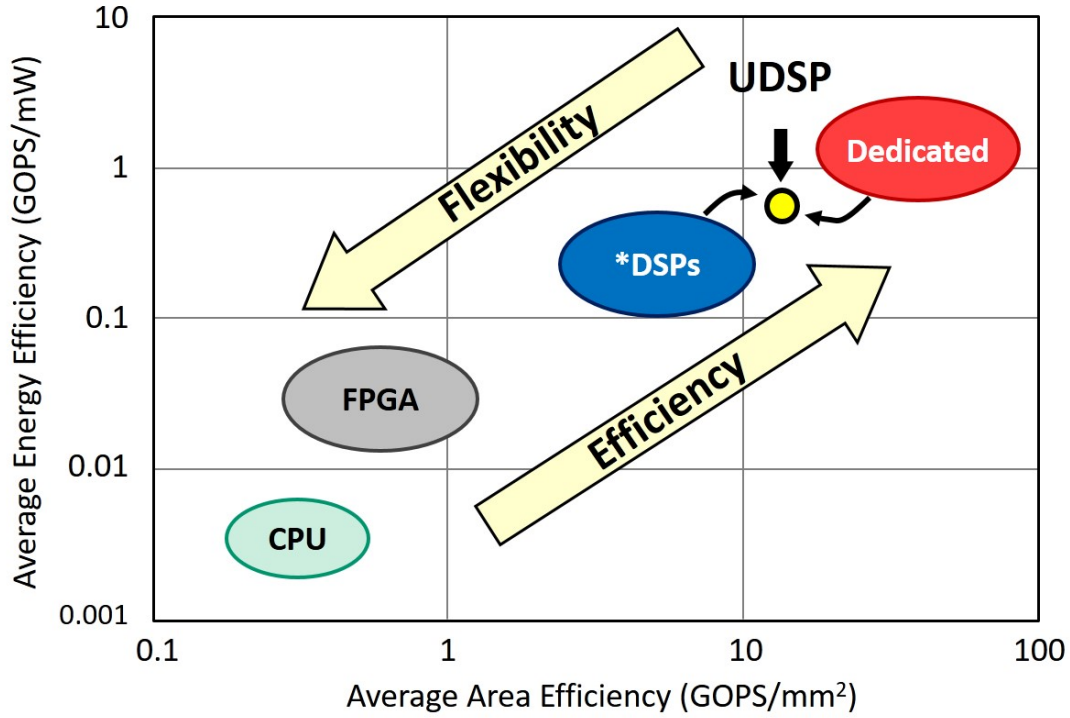


Figure 1.4: Energy and area efficiency of UDSP relative to other architectures. Adapted from [50], [57], [30].

We also develop a methodology for active hardware resource management of reconfigurable arrays to enable the use of such highly scalable and high-performance architectures in other emerging fields such as self-driving cars, cognitive radios, cloud computing accelerators, cloud inferencing, etc. Such domains require a high number of hardware accelerators and could benefit from the use of reconfigurable arrays for multimode operation as well as continuously-evolving algorithms. The hardware and throughput requirements of programs may vary based on their execution stages, environment, and runtime multiprogram dynamics. Overprovisioning the hardware for different runtime program traces, which may or may not exist simultaneously, leads to poor active utilization of the array. In the self-driving car example, several Neural Network (NN) models exist, but very few would be actively used per video feed from different cameras. In the cognitive radios example, the classification algorithms depend on the incoming signal, but only a few of the algorithms would be actively used for each detected signal. Programmers have to optimize the runtime hardware resource use by predicting and planning for the runtime program dynamics during compilation. Such compilation procedures are time-intensive and require high manual effort from the programmer. The problem is further exacerbated when multiple programs from multiple developer teams have to run simultaneously on the array. The developers have to predict multiple runtime program traces and efficiently map hardware resources among their programs at the time of compilation. An automated runtime management system eliminates such extensive manual effort and accelerates the overall compilation process.

We call our active hardware resource management feature ACAS for Accelerator as a Service. The key innovative feature of ACAS is runtime program relocation, enabled by symmetries in the array's interconnect network. The reconfigurable array used for demonstration is UDSP, which uses a mesh-based interconnect network that is translation- and rotation-symmetric. The symmetries allow us to translate, rotate, and mirror an incoming program on the array with minimal modifications to the program binary while preserving the internal configuration of the program. This feature allows ACAS to accommodate the incoming programs on the available resources without a complete recompile. Our hardware-

based solution completes the program relocation procedure in under 20 clock cycles. The ACAS architecture virtualizes the accelerator array by spatially and temporally scheduling multiple programs on its available resources, thus achieving higher active utilization for mapped programs on the array. ACAS allows developers to compile programs for hardware acceleration on the array without requiring additional manual steps for runtime resource planning at compile time.

By dynamically programming the array for various program requests, the ACAS feature along with our Runtime Reconfigurable Array is $8-14\times$ faster in signal processing workloads for the blind signal classification model when using an array size of 18×18, as compared to a statically-configured CGRA such as UDSP. For workloads comprised of Basic Linear Algebra Subroutines (BLAS), fully-connected Neural Network (NN), Convolutional Neural Network (CNN), Image Processing (IP), Machine Learning (ML), and media encoding/decoding, we observe an average throughput gain of up to $5.14\times$ over statically-configured arrays of similar array size, while consuming similar energy and occupying marginally higher ($< 5\%$) area.

1.1 Dissertation Outline

The organization of this dissertation is as follows:

- Chapter 2: Architectural Efficiency Study: Design and Efficiency Insights
  In this chapter, we explore the program execution flow in various compute architectures. Based on the analysis, we develop insights about the underlying microarchitectures and Instruction Set Architecture (ISA) and the enabling features, which allow for a wide variability of throughput, efficiency, and flexibility. We use the exploration and analysis to devise a multi-domain Architectural Efficiency (AE) metric, which quantifies the performance of various microarchitectures and ISAs. We use the AE metric to measure the performance of our UDSP architecture and the insights will allow us to design better future architectures.

- Chapter 3: Hardware Architecture of UDSP

In this chapter, we elaborate the hardware architecture of UDSP and its various hardware components. We describe the domain exploration of the field of digital signal processing, and explain the decision-making process and analysis of various existing architectures to select an appropriate interconnect network for the array and its constituent switchboxes. We elaborate the design of SNR-10 interconnect channel for fine-pitch interconnect fabric. Finally, we present the results obtained from the single UDSP and the Multi-Chip Module (MCM) of 2×2 UDSP on the fine-pitch interposer Si-IF.

- Chapter 4: Software Compiler for UDSP and RTRA

  In this chapter, we describe the design of a compiler for the software-friendly reconfigurable architecture, UDSP. We go over various software components required to compile the user input data flow graph to the binary bits required to configure UDSP. Then we describe the software compiler changes required to accommodate the multiprogram tenancy and hardware virtualization features, such as multi-size compile, multi-step compile, and program abstraction as polygons, enabled by Accelerator as a Service (ACAS) on Runtime Reconfigurable Array (RTRA).

- Chapter 5: RTRA for Accelerator as a Service (ACAS)

  In this chapter, we describe the hardware components required to enable runtime reconfiguration on the array. We present the active runtime hardware management features on the array which enable array virtualization using ACAS, including the hardware scheduler, hardware compiler, multi-bank memory and a sparsely-connected IO network.

- Chapter 6: Conclusion

  In this chapter, we conclude the dissertation and describe the significance and contributions of this work and the future use case scenarios of the RTRA-ACAS design, as well as describe various avenues of exploration and optimization for future researchers.

CHAPTER 2

Architectural Efficiency Study: Design and Efficiency Insights

While designing a compute solution for any given application or application domain, the system architects and designers have to analyze its computation requirements and decide upon a system architecture from a wide variety of available architectures such as CPU, GPU, FPGA, DSP, hardware accelerators etc. They have to perform trade-off study between efficiency, flexibility, generalizability, domain-specificity and programmability of various architectures. Architectures such as CPU and GPU are highly generalizable and can provide high degree of flexibilty however at the cost of poor efficiency relative to other architectures, on the other end of flexiblity spectrum we have dedicated accelerators like ASICs, which are highly area and energy efficient but have a very limited flexibility. In the middle of the spectrum of flexibility and efficiencies there are reconfigurable architectures such as Digital Signal Processors (DSP), FPGA and CGRA. Over the years the designers have come up with many novel reconfigurable architectures with varying degree of compute specialization and interconnect networks for various different application domains. These architectures employ several optimization techniques such as frequency-voltage tuning, multicore operation, deeper compute pipelines, innovative device physics, higher memory bandwidths etc to achieve higher throughput as well as higher efficiencies. The performance of such architectures is usually measured with metrics such as operations per second, operations per watt, throughput, latency etc. However, many of the optimization techniques employed to achieve higher performance and efficiencies can be implemented in other architectures as well and can be used to improve upon them. As an example, multicore operation and parallelism op-

timizations are algorithm dependent, and architectures that can optimally implement SIMD style execution can leverage the algorithmic structures to improve upon their efficiency, additionally, the frequency-voltage tuning can be applied to every architecture to select an optimal system operating point to improve upon their energy efficiency and most architectures would see an improved throughput with increase in memory bandwidth. Moreover, the energy- and area-dependent metrics such as $GOPS/mW$ and $GOPS/mm^2$ are highly technology dependent, which implies that an architecture would score higher in those metrics with a redesign using a smaller technology node even with no changes in the underlying microarchitecture.

We develop a multi-domain Architectural Efficiency (AE) metric which can be used to quantify the throughput and efficiency gains of a microarchitecture independent of the hardware and physical implementation variables. A metric that can be used to optimize and quantify changes to the the Instruction Set Architecture (ISA) at the conceptual or pre-RTL stage of the microarchitecture space exploration. AE metric can quantify performance and efficiency gains of different types of architectures and helps analyze the specialization decisions, compute, interconnect network, and memory hierarchy that drive those gains, as well as provides a method of analyzing the architecture at the ISA stage prior to RTL/hardware implementation. The AE metric provides the designer with an insight into the architecture. Based on the architectural decisions made during ISA design and level of parallelism and concurrency, the AE metric can also be used to estimate throughput of the final design.

2.1 Architectural Efficiency Metric

Energy efficiency and area efficiency of an architecture depend on several factors but one of the important ones that can get overlooked is the ISA. We analyzed the execution flows of a few commonly used program kernels on widely used architecture such as CPU, GPU, reconfigurable architectures such as UDSP and FPGA; and hardware accelerators such as Google's Tensor Processing Unit (TPU) [23], we will go over the analysis in upcoming sec-

tions. Using the analysis we came up with the Architectural Efficiency (AE) metric as given in Equation 2.1.

$$AE = \frac{\sum_{computations}(compute\ \ bitwidth)}{\sum_{instructions}(instruction\ \ bitwidth)} \tag{2.1}$$

AE metric is defined as the ratio of the sum of bits required for computation and sum of bits required for instructions or in the case of reconfigurable architectures the size of reconfiguration bits required to perform the computation. The metric essentially quantifies the amount/number of instructions required to get the architecture to perform the desired computation. We consider any arithmetic or logical operation performed on the data as a computation. Any other computation performed for the control mechanism, address generation or loops, that is any computation not performed on the data is not counted. We sum compute bitwidth over all the computations to accommodate architectures with increasing or variable wordsizes, and sum the instruction bitwidth to accommodate ISAs with variable instruction widths, and reconfigurable architectures like CGRA and FPGA where the reconfiguration bits can be several kilobits in length. Compute bitwidth and instruction bitwidth also enable users to estimate the requirements for instruction and data bandwidth for the particular computation and architecture.

The optimal ISA for the algorithm under test would have smaller size of reconfiguration bits and least number of instructions for most computations, and thus would score a higher AE metric compared to other architectures. As an extension, an architecture designed for a specific program or domain of programs would score highly in that particular domain since its memory hierarchy, interconnect network and compute would be specialized to perform the tasks of that domain with least number of instructions bits, however it might score low on other programs not belonging to the original set. It may implement the other program in a suboptimal way, thus incurring penalty in instruction count for the same computation. The efficiency of an architecture varies from program to program as well as the size of computation required for the program. AE metric is presented as a graph where the AE score is plotted against the size of computation, as we will observe, many architectures that

are highly efficient for larger programs may not be so for smaller size computations of the same program. Additionally, provided the width of instruction decoders, Instructions Per Cycle (IPC), compute latency, concurrency (hardware parallelization), and clock frequency of the particular architecture, the AE metric can also be used to predict throughput of that architecture. This throughput estimate allows the designer to get a rough estimate of the throughput of their architecture at pre-RTL or conceptual stage and theoretically analyze the concurrency, compute, or network for data reuse specializations and other optimizations that they can employ to design an optimal architecture.

2.1.1 ISA vs Reconfiguration Bits

While defining the AE metric, we form an analogy between the instructions in instruction-driven architectures like CPU or GPU with the reconfiguration bits in reconfigurable architectures like FPGA or DSP; here we provide an intuitive reasoning behind the analogy. Every algorithm or program can be abstracted as a data flow graph, where the nodes represent compute, decision, logic elements, memory, and registers, and the edges represent movement of data or control bits. In an instruction-driven architecture, the instructions are applied temporally, and each instruction facilitates data or control movement along the edges of the graph. In a reconfigurable architecture, the configuration bits would perform a similar data flow graph, but spread out spatially by programming the interconnect network, selecting the compute elements and data path registers.

```
static void simple_gemm(int n1 , int n2 , const float *A,
const float *B, float *C){
    int i,j,k;
    for (i=0; i<n1;++i) {
        for (j=0;j<n1;++j){
            float prod =0;
            for (k=0;k<n2;++k){
                prod  += A[k*n2+i] * B[j*n1+k];
            }
            C[i*n1 + j] = prod;
        }
    }
}
```

Figure 2.1: C code snippet for matrix multiplication.

Figure 2.1 shows a C Code snippet for the kernel inside matrix multiplication program. Once compiled the assembly output of the same kernel is shown in Figure 2.2. Each of the instructions in the assembly code is decoded and executed consecutively in the CPU hardware. The instructions effectively create a data flow graph in temporal domain as shown in Figure 2.3. In a typical CPU program, there could be additional graphs for instructions to update frame pointers, program counters, and memory address generation connected to the data flow graph, not shown in the Figure 2.3.

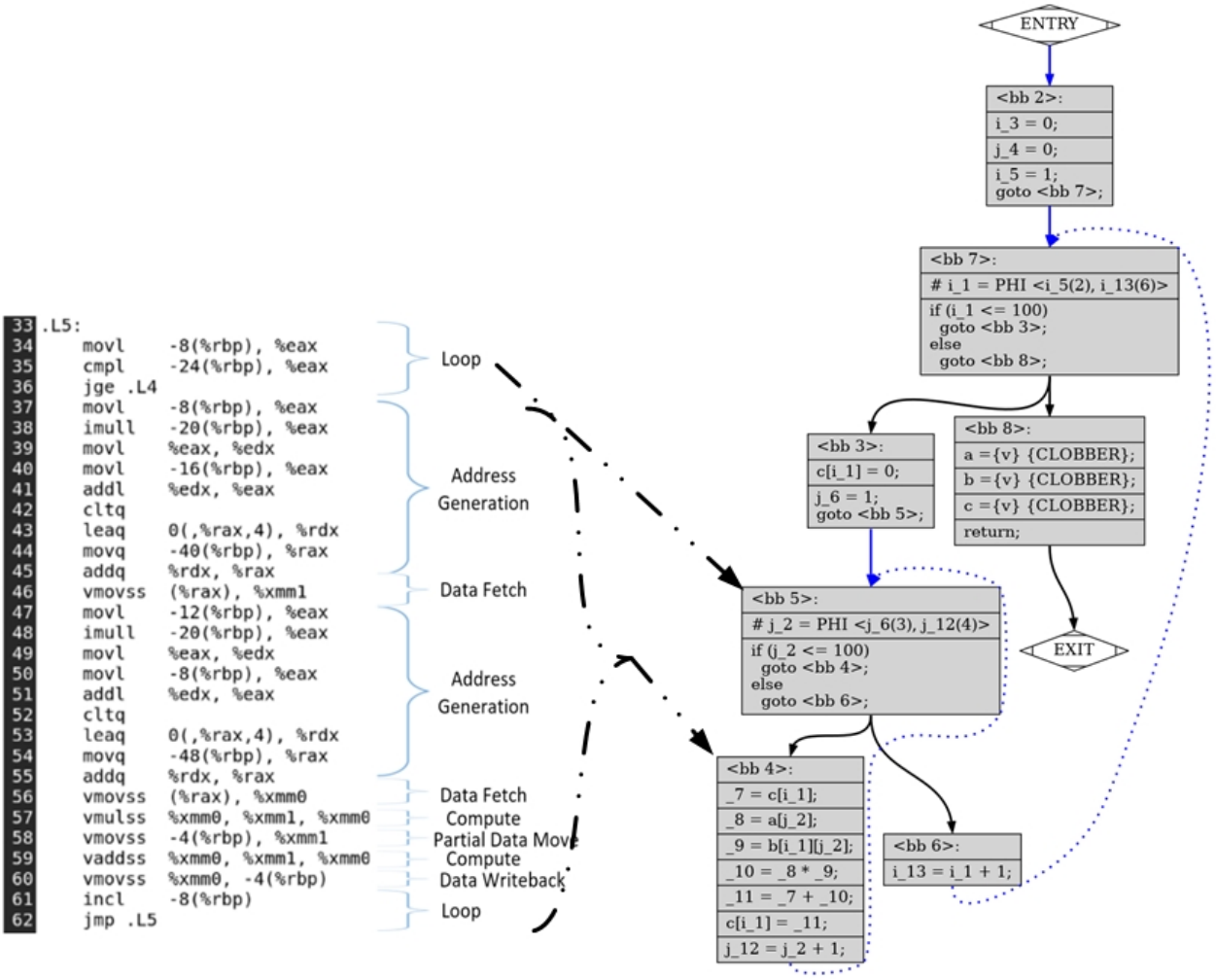


Figure 2.2: Assembly code output of the C code kernel from Figure 2.1.

Similarly when we program a reconfigurable hardware such as FPGA using the reconfiguration bits for the MAC program, the program modifies the data paths inside the routing

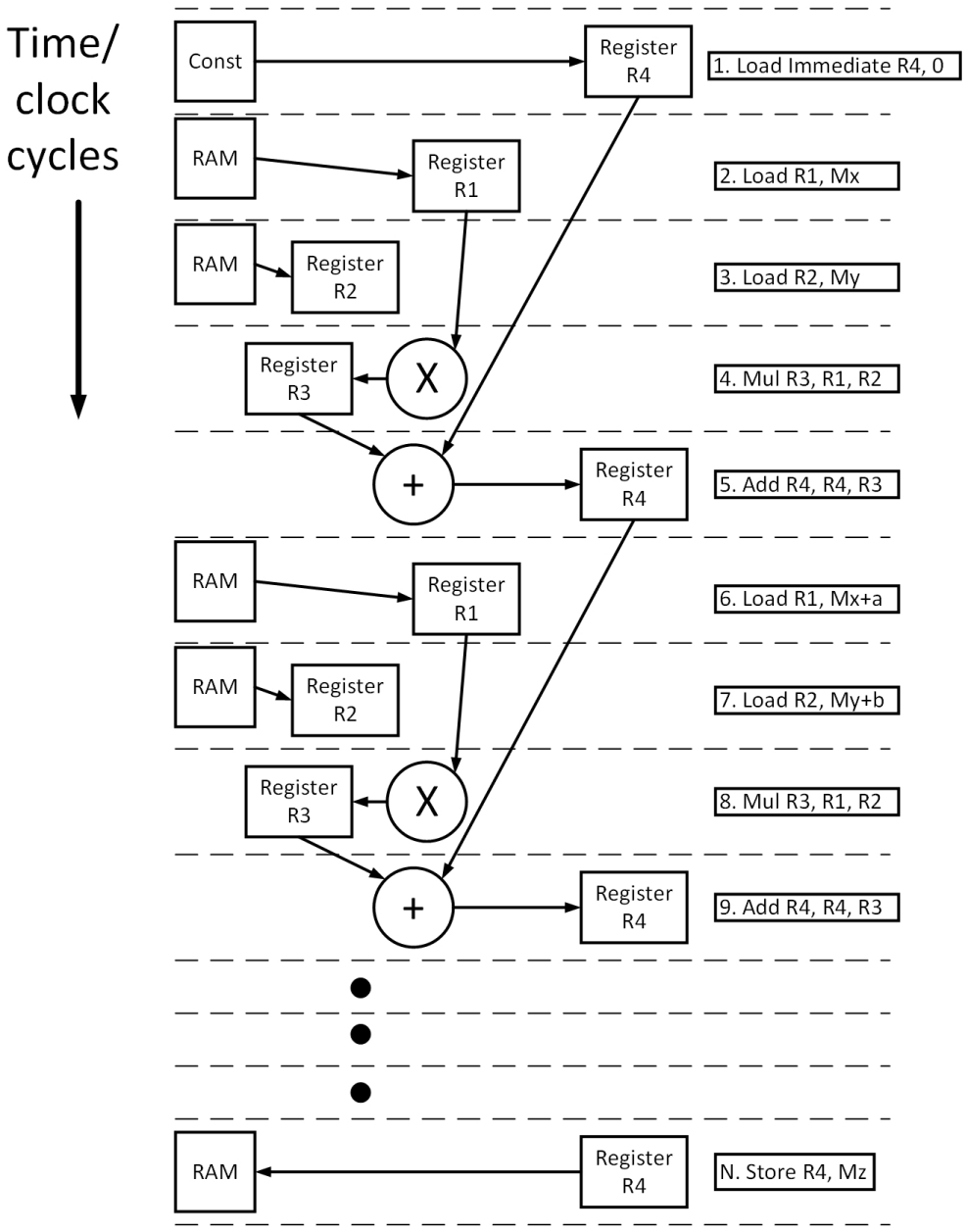


Figure 2.3: Example temporal data flow graph of the MAC operation executing on CPU.

network inside the FPGA to perform the MAC operation and the Look Up Table (LUT) or a similar processing element inside the reconfigurable array is also modified to perform compute required for the MAC program. An example of such spatial construction of MAC data flow graph on FPGA is shown in Figure 2.4.

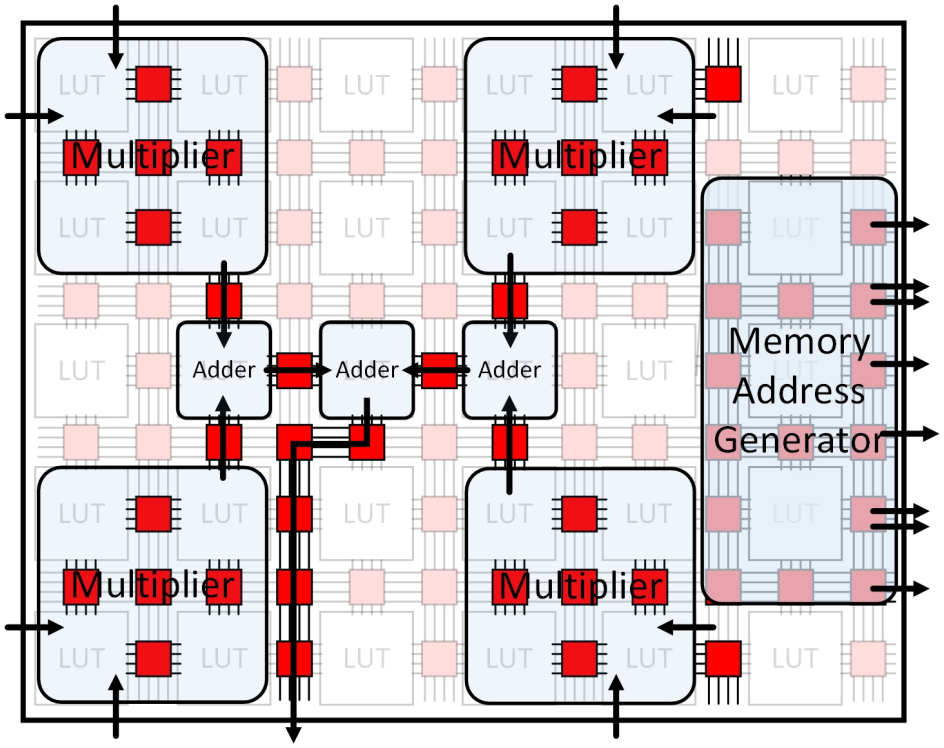


Figure 2.4: Example spatial data flow graph of the MAC operation mapped out on FPGA.

Thus, instructions for CPU and reconfiguration bits for FPGA perform the same functionality in a different temporal or spatial fashion, and various architectures lie along this temporal or spatial spread. For architectures like FPGA, CGRA, and ASICs, the sum of instruction bits includes the initial reconfiguration bits, instructions executing on the main thread for the program, and any supplemental instructions issued to the architecture at runtime. For instance, if we simulate a CPU on an FPGA, then the instructions for the program running on the simulated CPU are also counted towards instructions bits, along with reconfiguration bits required to map the CPU program to FPGA. Additionally, an application specific hardware accelerator or a domain-specific hardware accelerator optimized for a particular algorithm/domain would have its network and compute elements optimized

for that algorithm and thus require a lesser number of instructions or configuration bits, and thus score higher on the AE metric.

2.2 Program Flow Analysis and AE Scores of Common Architectures

In this section, we analyze the program flow and calculate the AE metric using some of the widely used architectures such as CPU, GPU, and FPGA, along with novel architectures such as Tensor Processing Unit. Noting an increase in Machine Learning and Neural Network workload, we wrote programs for General Matrix Multiplication (GEMM) on the different architectures and compared the results of the AE metric. Executing the AE metric on a generalized architecture like a CPU helps understand the specializations and instruction reduction techniques that form the underlying principles for many other architectures. Specialized architectures like CGRA, DSP, and Near-Memory Compute (NMC) have features like interconnect networks, memory hierarchies, and multicompute pipelines, which, compared to the execution style of a CPU, can be conceptualized as instruction reduction techniques like vector processing, eliminating load/store operations, data path optimizations, compute specialization, etc.

2.2.1 CPU

Decades of research has led to many different CPU architectures. Although the ISAs for CPUs are different but the instructions and functions involved are similar. Most of the ISAs comprise of common instructions such as load/store, compare, jump, branch, and arithmetic, multiply, add etc., some ISAs may even have some other specialized instructions like AVX-512, SSE, AVX-2 [13], [46], [11]. The variation in performance of the architectures arises from the difference in clock and memory speed, memory hierarchy, power budget, and other factors which are largely independent of the ISA. The instructions in the CPU ISAs each perform a relatively small operation on a single data element; therefore, to perform a single

Table 2.1: Number of instructions required to run matrix multiplication program kernel on a matrix of size 1×1, i.e. a scalar element, using 2 threads on different CPU architectures.

Architecture (ISA)	Main	Program Kernel Loop	Multithreading
ARM A53 (ARMv8-a)	58	30	9717
Ryzen 3950x(x86-64)[a]	59	29	9013
Xeon E5-2640v4(x86-64)	62	24	6922
i7-2670qm (x86-64)	58	27	8444

[a]VirtualBox Instance using SVM

computation, many supplementary instructions are executed to read/write data, generate addresses, and branch operations. Moreover, CPUs can only execute very few instructions per clock cycle depending on the width of instruction decoder, so the extra instructions used are an overhead in an IPC limited CPU architecture.

The same C code was compiled with GCC with native architecture optimization flags "-march" and "-mtune" enabled on each architecture, the c code is shown in Figure 2.1. Analyzing the assembly code for different architectures tested, the code can be divided into three main categories 1) the main function, 2) program kernel loop, 3) multithreading overhead. In a given architecture the size of main function and the GEMM program kernel loop remains constant, and multithreading overhead scales with the number of threads enabled. The number of instructions required in the nested C loop within the GEMM kernel for different CPU architectures is provided in Table 2.1.

For different CPU architectures, the assembly code largely consists of opcodes which perform similar operations, differing mainly in their order of execution, and certain architecture specific optimized instructions. For a large matrix of size $N \times N$, the initial overhead and setup instructions get amortized, while the nested GEMM loop which is executed N^3 times becomes the limiting factor for the program execution. Thus, by observing the assembly

output, shown in Figure 2.2, we can calculate the number of instructions required for the nested GEMM loop which has 2 useful computations (1 fp32 multiplication, 1 fp32 addition). For x86-64(i7), the loop has 27 instructions, so its AE metric should converge to 0.037 for large matrices. Enabling multithreading did not affect AE score for large matrices. In a multithreaded program, the total computation required for the program is spread over multiple threads and multiple cores but each thread is running an independent subprogram, repeating a similar set of instructions, hence, the total number of instructions required to perform the compute in all of the threads remains largely unchanged. However, enabling multithreading requires additional setup instructions, which get added to the total number of instructions. These added instructions are independent of the size of computation, however, their contribution to the total number of instructions reduces the AE metric score for small matrix sizes as shown in Figure 2.5, while for larger matrices the added multithreading instructions get amortized.

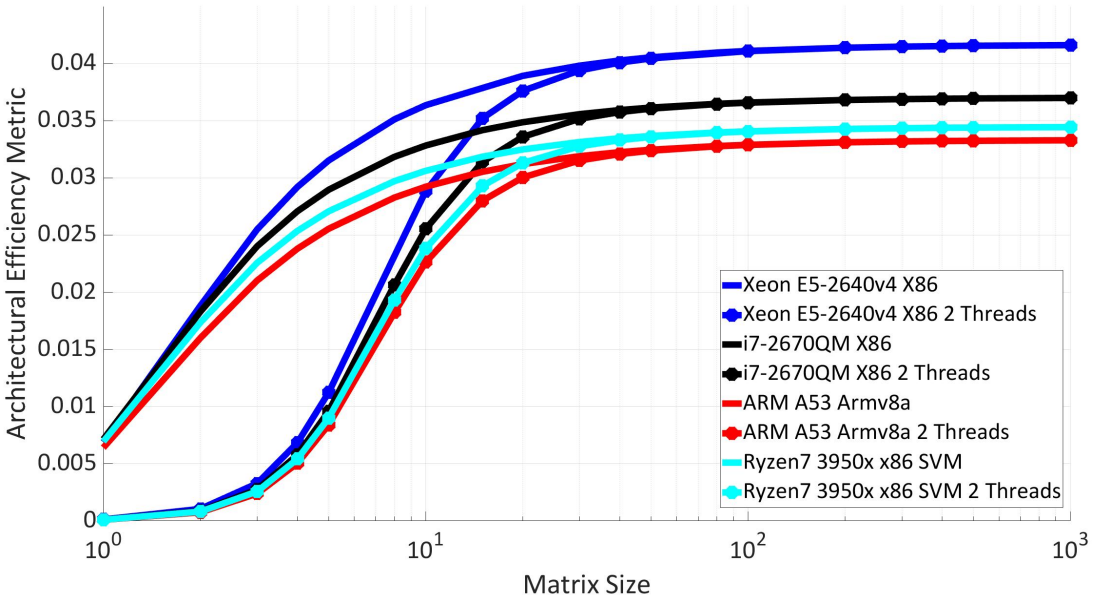


Figure 2.5: AE metric for different CPU architectures, running singlethreaded and multithreaded GEMM workload with varying matrix sizes.

We can also observe in Figure 2.5, that the CPU ISAs which require the lower number

of instructions in the nested loop score higher on the AE metric score, as they require lower number of instructions to perform the same amount of compute. However, this may not relate to actual throughput gains of the CPU since the scores are so close to each other, the secondary effects of microarchitecture such as memory bandwidth, branch prediction, physical implementation and most importantly clock frequency may shuffle around the measured throughput of the architectures. An argument can be made, that a designer could design the secondary structures with same performance and operate the designs at same clock frequency across the CPU architectures, and the AE metric could tell us which architecture would fare better in that case. This is essentially the RISC vs CISC comparison, although CISC architectures are much better for compute the physical implementation of these architectures can be inefficient, and on other hand RISC ISA may not have a very efficient ISA however the simpler instructions and decodes allow them to be much more efficient. Overall, as of now the CISC architecture provide a better single thread performance albeit at the cost of higher energy and power than RISC architectures, as can be observed in AE metric.

2.2.2 GPU

GPU architectures are optimized to perform matrix operations. For our testing we used Nvidia GPUs, with CUDA libraries and NSight Compute Profiler. We performed our testing by running the Matrix Multiplication kernel on CUDA cores in Nvidia 1650 Super (Turing Architecture), Nvidia 3080 (Ampere Architecture) and Nvidia 1050ti (mobile Pascal Architecture) [35], [36], [34]. The CUDA core architecture in the three generations is largely the same. The GPU is divided into Streaming Multiprocessor (SM) units each comprising 4 Instruction Dispatch Units (IDUs), each issuing 32 threads every clock cycle, one each for the attached 32 CUDA cores as shown in Figure 2.6 . A single instruction in the GPU assembly code dispatched by IDU is executed as threads on the 32 CUDA cores. Hence, GPU architecture exploits the parallelism and concurrency of the algorithm, and this transformation allows GPU to reduce instructions by 32× to score 32 times higher as compared to CPU on AE metric.

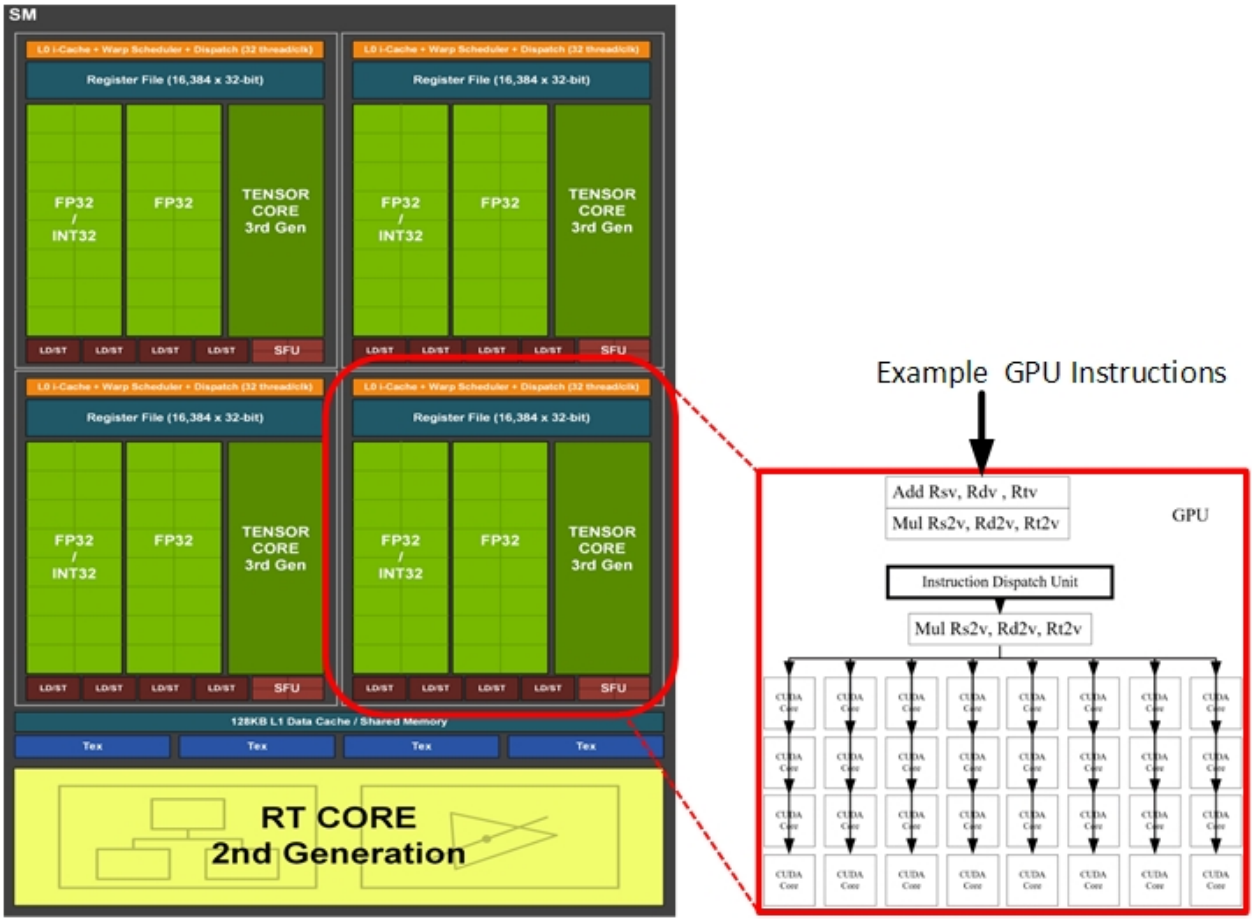


Figure 2.6: Architecture of a Streaming Multiprocessor (SM) unit inside Nvidia GPU [36], and an example warp dispatch.

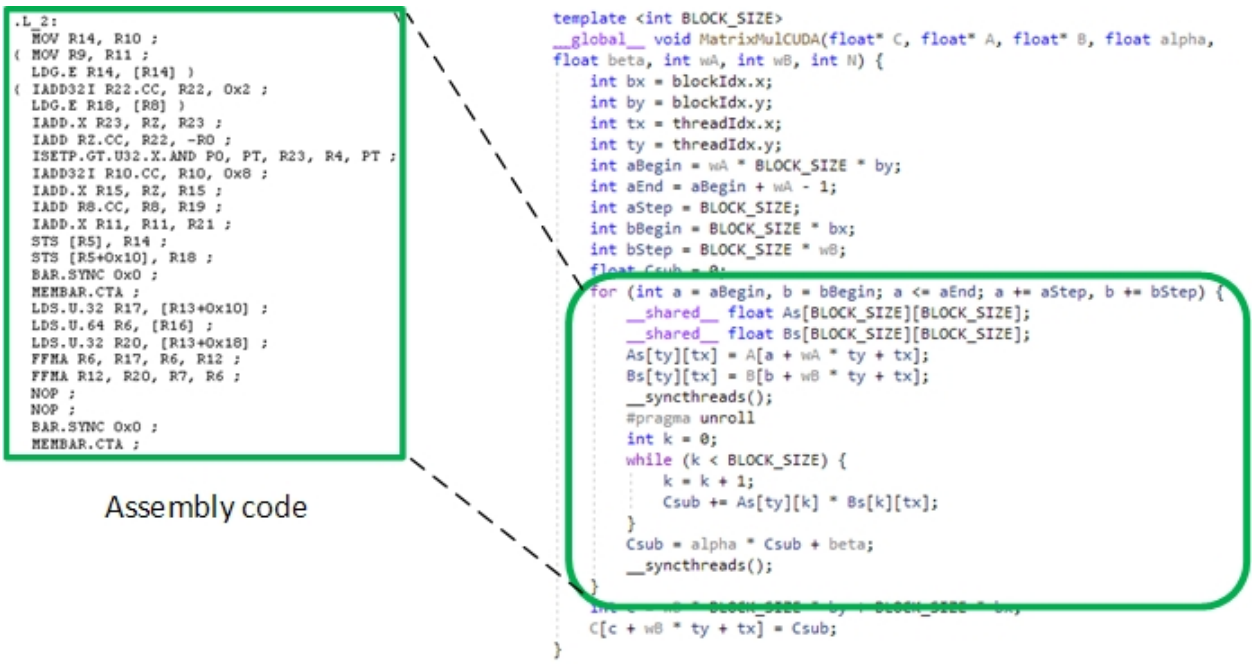


Figure 2.7: GPU assembly code (left) for the matrix multiply kernel.

The GEMM loop in GPU code for a single CUDA warp is shown in Figure 2.7. It is similar to the CPU style of execution but the GPU has a fused instruction for floating point multiply and add, and a few extra instructions for threadID and blockID to allocate threads to their respective CUDA cores. Matrix multiplication in GPU was segmented into smaller 3×32 matrices, and the loop has been unrolled 32 times to speed up program execution. These transformations, fused instructions, and loop unrolling allow the GPU to gain another $9\times$ reduction in the number of instructions as compared to a CPU for the same program, thus a $\sim 288\times$ total gain in the AE metric. For Pascal, Ampere, and Turing architectures, the unrolled loop has 94, 89, and 89 instructions respectively, and each loop performs 64 computations and each instruction/warp is dispatched to 32 CUDA cores, thus, theoretically, AE metric should converge to 11.5 for large matrices for Ampere and Turing and to 10.9 for Pascal, as can be observed in Figure 2.8.

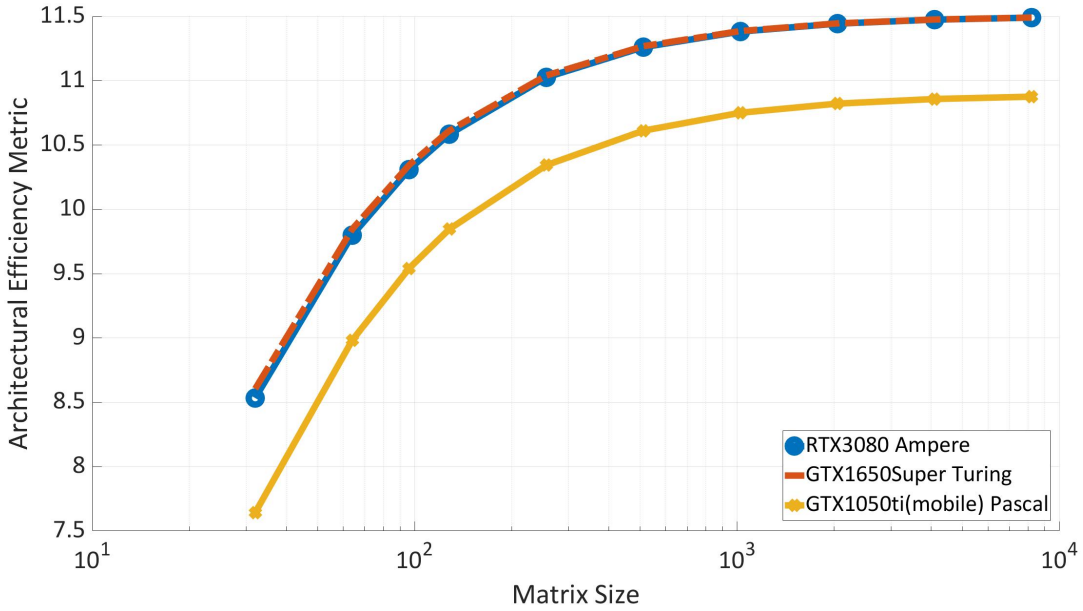


Figure 2.8: AE metric for different GPU architectures for GEMM workload with varying matrix sizes.

2.2.3 FPGA

FPGA architecture is flexible, which allows the user to exploit the parallelism of the algorithm, increase data reuse (interconnect network), and compute specialization (multicompute pipeline). We mapped our GEMM kernel as a data flow graph and a memory address generation block, as shown in Figure 2.4 on Altera MAX10 FPGA and Intel Stratix10 FPGA. The concurrency of program implementation is limited by hardware factors like memory bandwidth or the number of logic blocks. In our implementation, the concurrency was limited by the width of read/write ports on the onboard RAM, which allowed for 18 input vector MAC (Multiply Accumulate) on MAX10 and 512 on Stratix10. Once an FPGA is configured, no further instructions are required to execute the GEMM program; however, the limited size of RAM on the FPGA boards limits the maximum size of matrices. Hence for larger matrices, the FPGAs ran out of memory, so we had to subdivide the matrices and perform multiple multiplications to accommodate the memory limitation. Compared to a CPU, an FPGA is able to eliminate instructions for partial data movements, multiple computes, loops, and

memory address generation by using its dedicated interconnect, implementing multicompute elements or MAC units, implementing unrolled, vectorized MAC units, and implementing special compute elements for generating addresses for memory accesses. Thus, FPGA eliminates the requirement for multiple instructions to perform computation, and requires just one time instruction or configuration bits. The size of instruction or configuration bits required for FPGA is 190kb for Stratix10 and 3kb for MAX10 FPGA. The large instruction size is detrimental to the AE metric score for FPGA in the beginning but the absence of regular incremental instructions allows its AE metric score to improve with growing matrix sizes, as in Figure 2.9.

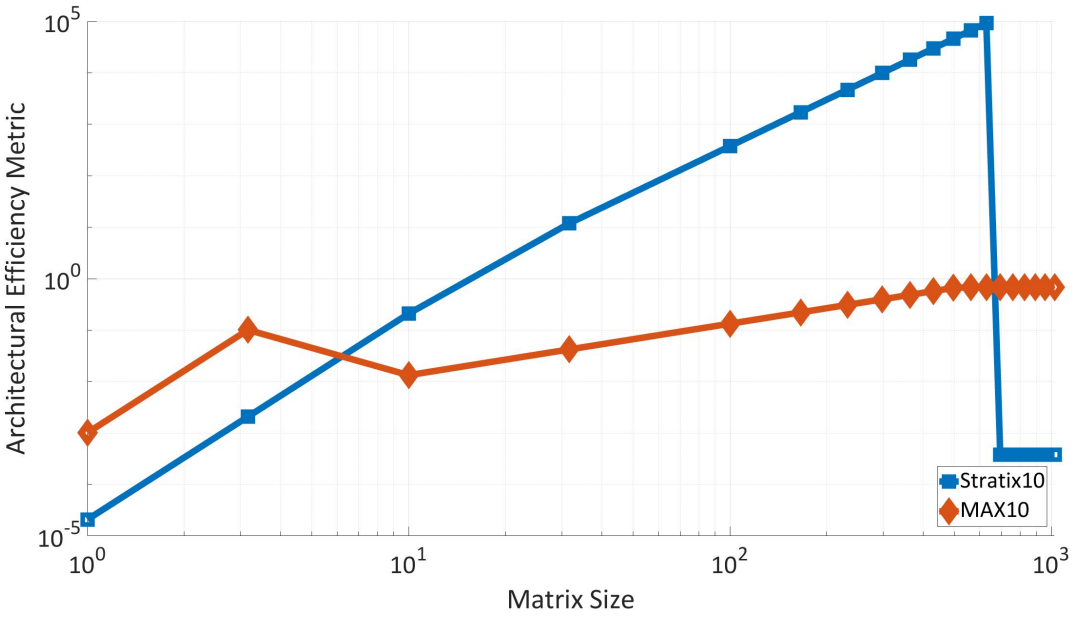


Figure 2.9: AE metric for different FPGA architectures for GEMM workload with varying matrix sizes.

2.2.4 Hardware Accelerators

Hardware accelerators or ASIC can be categorized into two broad categories, instruction driven ASICs and statically programmed ASICs [30]. An example for an instruction driven ASIC would be Google's TPU [23], which is a domain-specific architecture designed specif-

ically for matrix-vector multiplication, whereas an example of statically programmed ASIC would be [60]. In this architecture, the program is statically mapped onto the CGRA and the data is streamed through it. We performed Matrix Multiplication on our own static CGRA, UDSP (Universal Digital Signal Processor) and mapped the data flow graph.

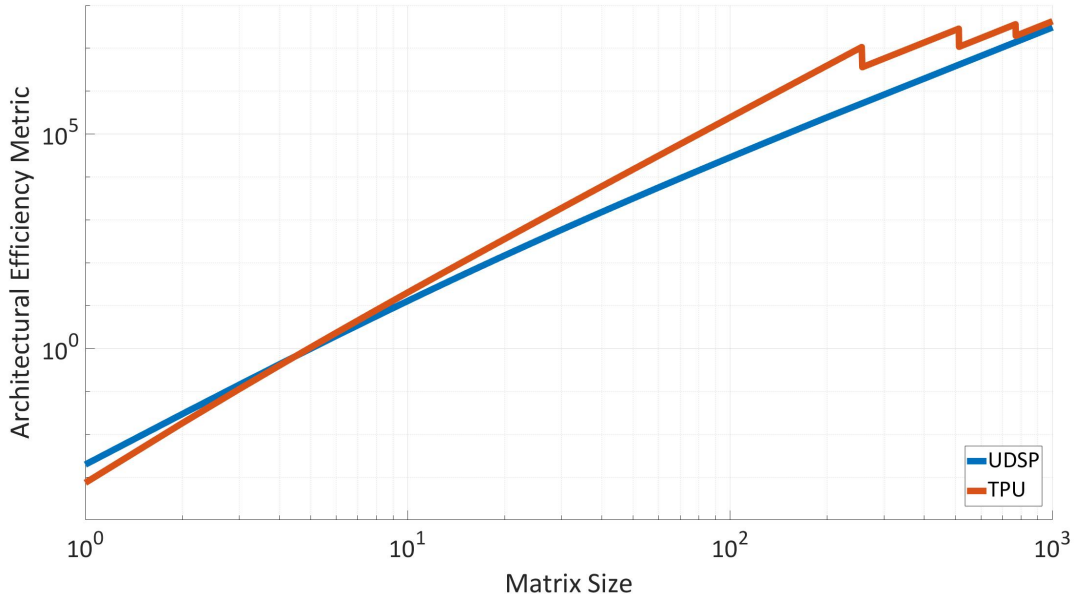


Figure 2.10: AE metric for TPU and UDSP architectures for GEMM workload with varying matrix sizes.

We will go over the internal architecture of UDSP in detail in Chapter 3, however for a brief description, UDSP is a 196 core, 16-bit fixed point processor and each UDSP core can be configured to perform a size 2 vector MAC. UDSP can perform 128-input vector MAC and memory address generation when provided with sufficient I/O bandwidth from the paired memory. Compared to an FPGA-based implementation, UDSP requires fewer configuration bits to implement the same input width MAC data flow graphs, hence, it outperforms FPGA in the AE metric, as in Figure 2.10.

Hardware accelerator TPU has very few instructions in its ISA, and requires 4 TPU instructions for Read_Host_Memory, Read Weights, MatrixMultiply, and Write_Host_Memory [23] to perform matrix vector multiplication. To perform a matrix multiplication, we need to

perform the matrix-vector operation multiple times, once for each column, which reduces the AE for the architecture. Assuming that instruction bitwidth for TPU is 64 bits (configuration + memory address) the AE metric for UDSP and TPU is as shown in Figure 2.10. TPU combines the multiple instructions for word level memory read/write into a single vector/matrix level instruction. The compute instructions for individual matrix elements are also reduced to a single instruction for large matrix level computation and the instructions for reusing the partial computes in matrix multiplication are absent since the interconnect/datapaths between the compute elements in TPU are dedicated to performing just that operation. It is optimized to perform matrix-vector multiplications of size [256x256][256x1] and larger matrices need to be subdivided to perform multiplication.

2.3 Discussion

We observe orders of magnitude difference in AE metric score of CPU, GPU, FPGA, UDSP and TPU as shown in Figure 2.11. We can observe that for extremely small matrix sizes it is much more efficient to perform the computation within the CPU core rather than initiating any of the acceleration devices. We can observe that AE metric score for UDSP and FPGA grow with similar slope for small matrix sizes since they both can implement the same data flow graph, however the orders of magnitude difference in the AE score originates from the different number of instruction bits required to configure the two architectures. Since FPGA requires much larger size of instruction bits to configure its interconnect network and also the internal Look Up Tables (LUTs), the instruction cost is much higher than a CGRA like UDSP, where arithmetic functional unit level granularity and domain-specific interconnect help cut down a lot on instruction bits.

Instruction driven architectures like CPU, GPU and TPU all converge to a single AE metric score for large matrices. The convergence value is driven by the number of instructions in the nested loop within the assembly programs of such architectures. Since CPU performs a single operation at a time with single instruction, the convergence AE score is much lower

as compared to other architectures. A SIMD architecture like GPU can perform upto 32 operations in parallel using a single instruction hence score an order of magnitude higher than CPU. The TPU can perform much larger computation, that is $[256 \times 256] \times [256 \times 1]$ matrix vector computation with just 4 instructions, hence is one of the best architectures to perform the task. However, TPU can perform just that one single operation, hence the enhanced AE score comes at the cost of flexibility.

We can also observe that the relation between throughput and energy efficiency numbers provided for various architectures can be derived from the AE score. We can observe a strong correlation in those reported metrics and the AE metric score at large matrix sizes. Hence, we might be able to incur that the traditional metrics of evaluation might not be considering the overhead and the inefficiencies of the program to fully utilize the hardware. The traditional metrics are only valid if the hardware resources are maximally utilized, which would amortize overhead of using the said hardware.

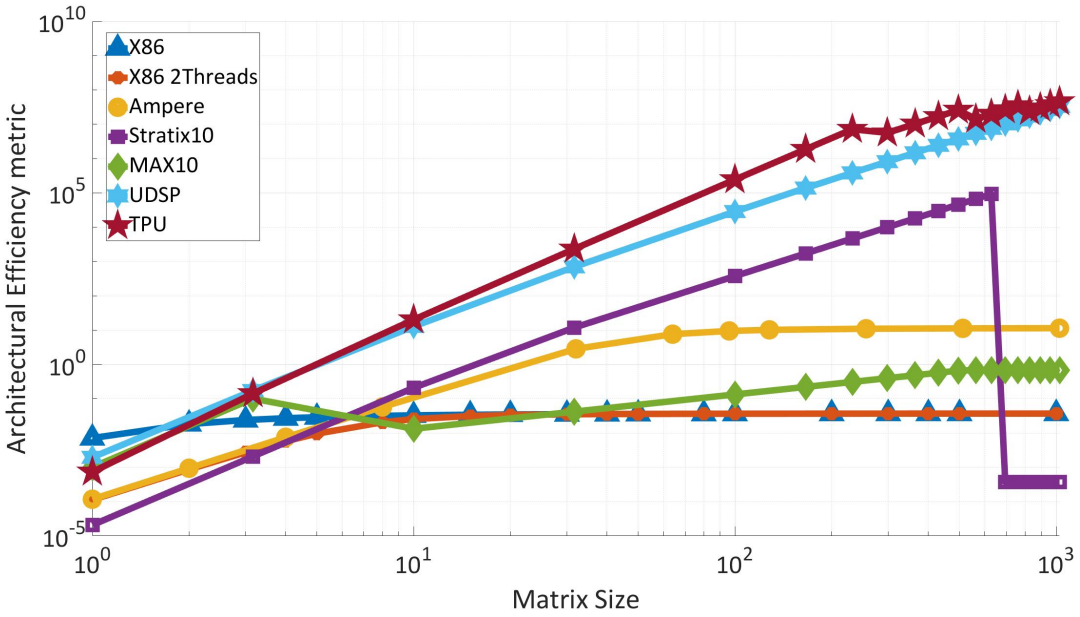


Figure 2.11: AE metric plot of various different architectures for GEMM workload with varying matrix sizes for comparison.

2.3.1 Throughput Estimation

For IPC bound architectures like CPU or GPU, We can estimate throughput of the architecture using its AE metric and its hardware implementation features such as instruction decoder width, multithreading and maximum operating clock frequency. For static architectures with pipelined compute data paths, like FPGA or UDSP, we can estimate the throughput by multiplying the AE metric with their clock frequency and concurrency/vectorization. For compute latency bound architectures like TPU, or dynamic CGRAs, estimating throughput requires information of compute latency. For instance, i7-2670qm Sandy Bridge architecture has an instruction decode width of 4 [13], and max clock frequency of 3GHz. RTX 3080 was operating at 0.7-0.9 GHz, has 68 SM modules, and each SM can schedule 4 warps per clock cycle. Figure 2.12 and Figure 2.13 show the measured throughput and estimated throughput for CPU and GPU architectures using AE model in 1D-convolution, Fast Fourier Transform (FFT) and GEMM workloads.

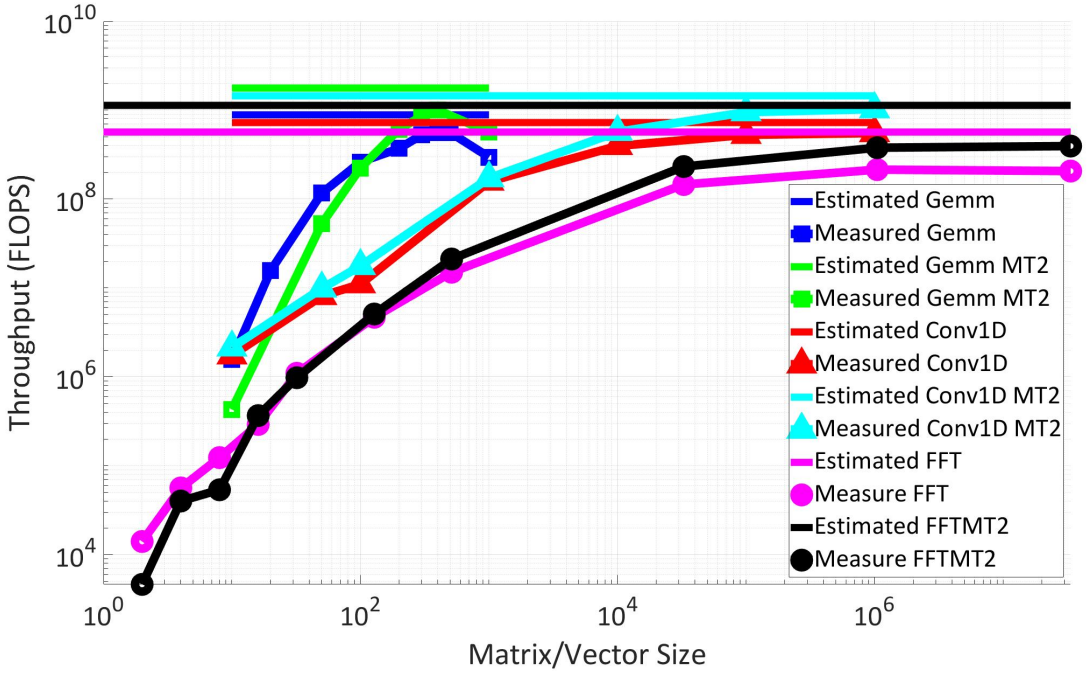


Figure 2.12: Estimated vs measured throughput of various programs on Intel i7 CPU.

The variance in the AE score could be used to infer flexibility of different architectures.

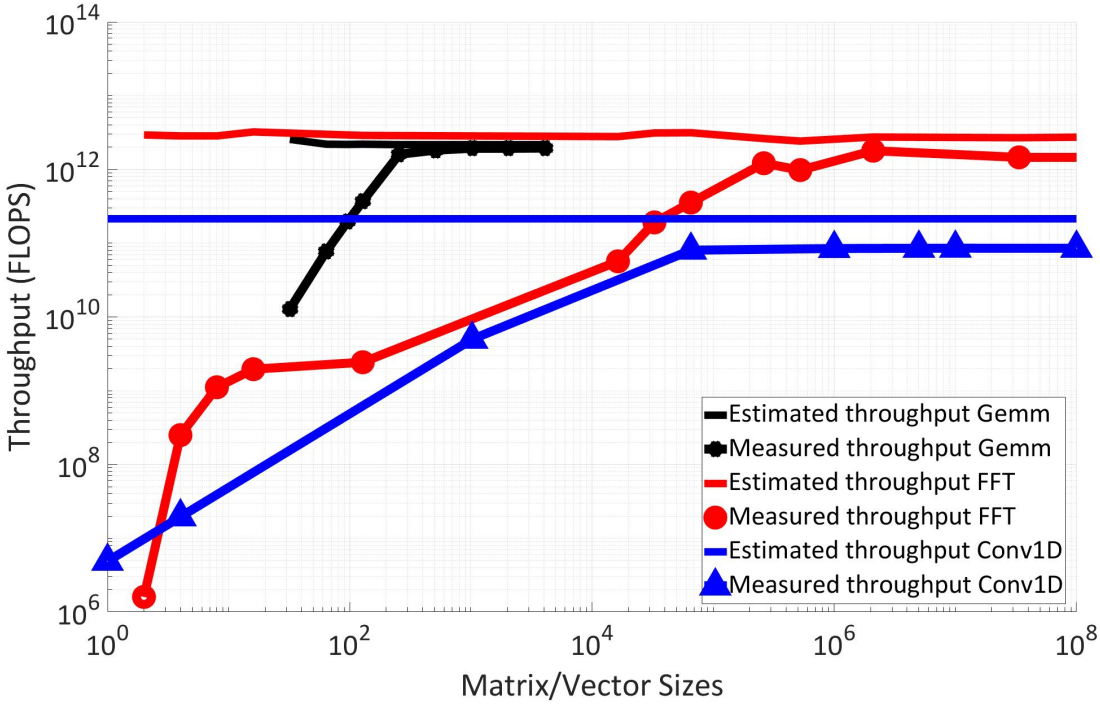


Figure 2.13: Estimated vs measured throughput of various programs on Nvidia 3080 GPU.

We observe that highly flexible architectures such as CPU tend to have a very low deviation in AE score across various computation requirements across different programs. Since a CPU can always decode, dispatch and execute a fixed number of instructions every clock cycles, depending on the program, it can very well perform a certain number of computations per instructions. However, architectures such as FPGA or CGRA, may require large number of configuration bits to configure certain programs, may have some inefficiencies in spatially mapping the program, which can cause a wider variation in their AE scores. The above is an observation and would require further testing and experimentation.

2.3.2 Domain Specificity

One of the major takeaways from the AE metric analysis would be the domain specificity. Spatially reconfigurable architectures such as FPGA and UDSP can be benefit a great deal from domain and requirement specificity. We can observe in Figure 2.11 that a smaller FPGA

MAX10 can outperform the larger FPGA Stratix10 for small matrix sizes, the reason being that for smaller matrix sizes, the extra hardware on the larger FPGA is an overhead and slows down the programming and reconfiguration and negatively effects the performance. Similarly, UDSP can benefit from microarchitecture optimizations such as compute specialization, domain-specific interconnect and fixed data widths, which would help cut down on the number of instructions bits required to perform the computation and would also benefit the utilization of the array. We observe these benefits in the results section of UDSP Chapter 3, and later we can also observe the benefit of reducing the instruction size in the programming and control bandwidth section of RTRA Chapter 5.

CHAPTER 3

Hardware Architecture of UDSP

Universal Digital Signal Processor or UDSP belongs to a broad class of architectures called Coarse Grain Reconfigurable Array (CGRA). A CGRA typically consists of an array of compute nodes, each executing a word level operation, communicating to each other across an interconnect network. Most of the CGRA designs can be differentiated and characterized based on their size, node functionality, interconnect and control mechanism. The size of CGRA refers to the number of processing elements or cores in the array, and the node functionality describes the function that each processing element performs [37]. Node functionality can be as simple as a Look-Up Table (LUT), arithmetic and/or logical operands or a relatively small general purpose core. If a CGRA has a LUT as its Functional Units (FUs) then it may very closely represent an FPGA, and a CGRA with a general purpose core would be resemble a multi-core CPU, in these two scenarios the implementation scheme of a program running on the CGRA would better distinguish a CGRA from the other two architectures. The CGRA designer can choose to keep the functional units homogeneous, having all the FUs identical or heterogeneous, in which some of the FUs could be functionally different than others, operate on different data types or could be memory storage elements like SRAM, DRAM or register file. There are variety of implementations for the interconnect network as well, while some CGRA architectures employ simple nearest neighbour architectures, others may have hierarchical or bus-based interconnects, a few of them implement a hybrid interconnect network which could be a mix of other networks at different level of granularity.

Figure 3.1 shows an example CGRA architecture with 16 Processing Elements (PEs) or

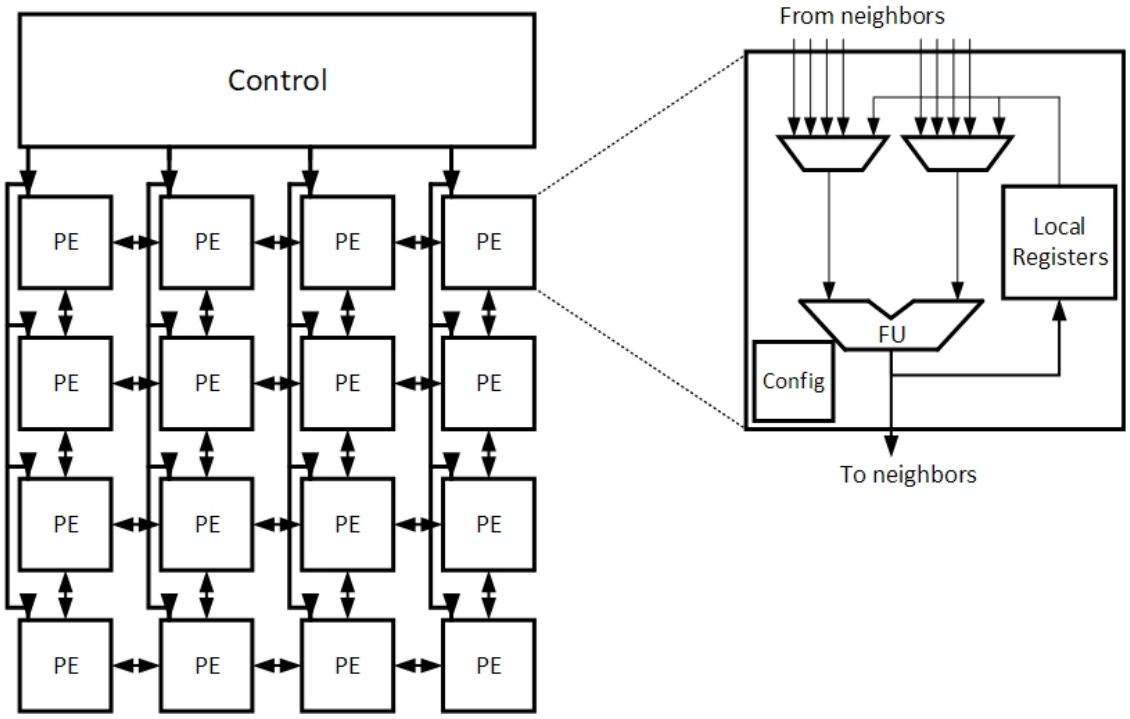


Figure 3.1: An example CGRA architecture.

cores arranged in a 4×4 2D grid. The interconnect network allows each PE to communicate to its four nearest neighbour PEs in a '+' fashion. Each PE contains a functional unit which can perform simple arithmetic operations, some muxes to allow inputs from neighbors, a configuration memory to store the configuration of the PE and a register file to store partially computed data. This architecture is just a simple example of vast design choices possible in designing a CGRA.

In this chapter we go over the hardware architecture of a reconfigurable array architecture that we designed called Universal Digital Signal Processor (UDSP).

3.1 UDSP Architecture

Figure 3.2 provides an overview of UDSP architecture. There are various components to the design of UDSP and we go over them in detail in the upcoming sections.

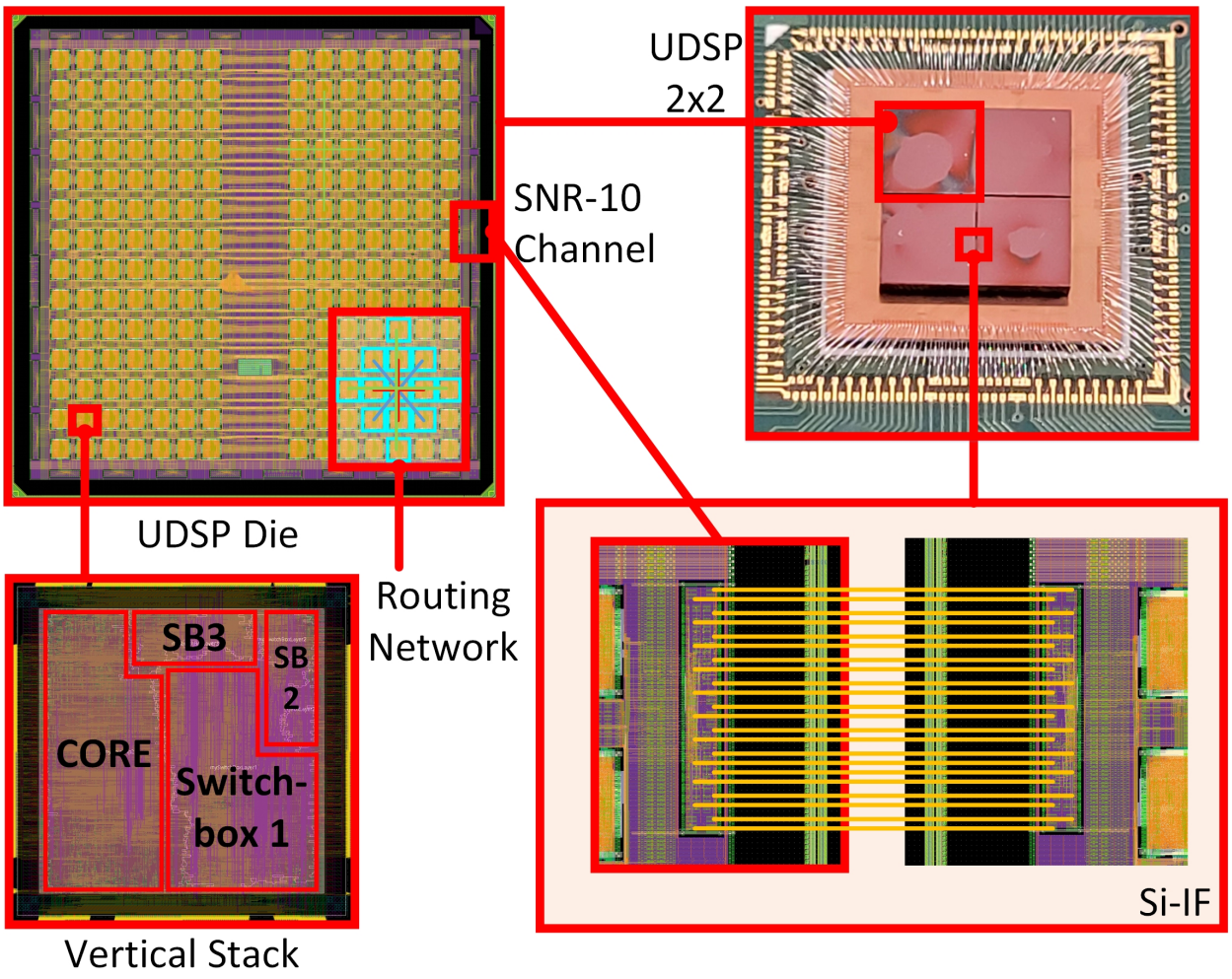


Figure 3.2: UDSP 2×2 architecture overview.

3.1.1 Core

Core or Processing Element (PE) forms the compute unit of any CGRA. The core consists of functional units like arithmetic operations, logical operations, bitwise manipulation etc. Our goal for UDSP is to design an energy and area efficient flexible architecture for the Digital Signal Processing domain with sufficient flexibility to accommodate the algorithm space while maintaining ASIC like performance and efficiency. The cores contribute the largest to the overall energy and area consumption of these architectures since they are replicated multiple times throughout the array. In order to design an efficient and optimal core architecture we need to examine the requirements of the algorithm space of the domain as well as prior work and architectures in the field.

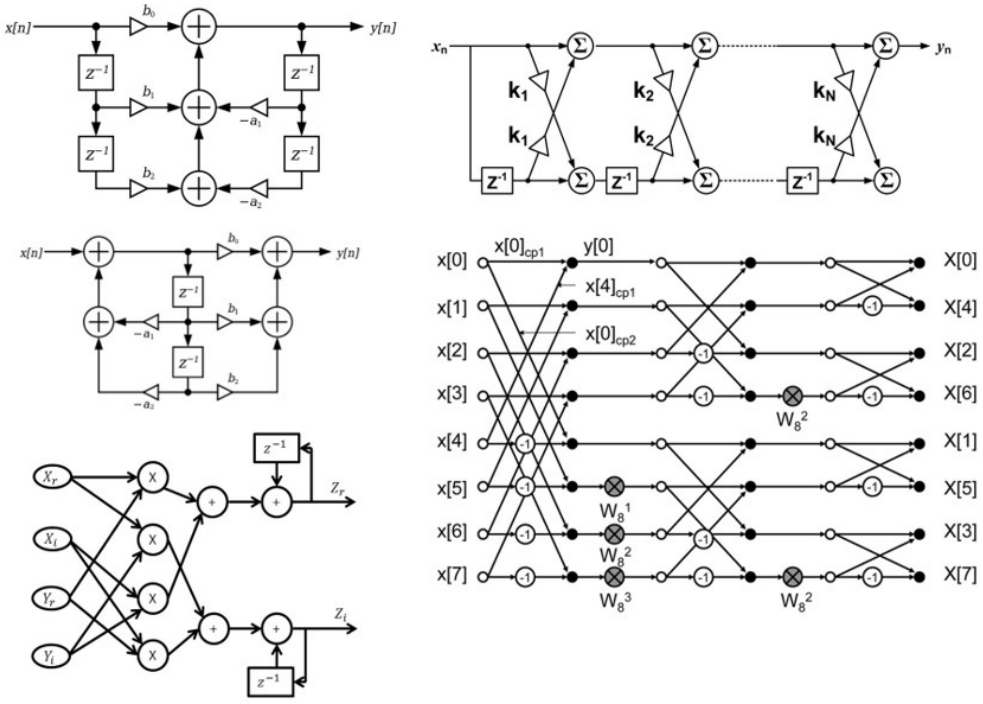


Figure 3.3: Repeating kernels for IIR filter, FIR filter, lattice filter, complex MAC, Cooley–Tukey 8-pt FFT.

To understand the requirements of the domain better we study an extensive list of DSP algorithms such as single cycle MAC, complex arithmetic, FIR/IIR/adaptive filters, blind equalization, power spectral density, sphere decoding, QR decomposition, SDF FFT/butterfly,

Convolutional Neural Networks (CNN), matrix multiplication etc. We break down the DSP algorithms into their constituent arithmetic operations and data movements. With arithmetic operations in the algorithms represented as nodes and the data movements represented as edges, we develop data flow graphs of each algorithm and identify the repeating kernels. Some of the kernels and data flow graphs are shown in Figure 3.3.

The existing and some of the most popular CGRA architectures [37], [31], [45], [16], [38], [17] use CPU style Functional Units (FU) or Arithmetic Logic Units (ALUs) inside the core as shown in Figure 3.1. If we try to map the algorithm kernels as shown in Figure 3.3 on the FU or ALU style cores, we observe high mapping inefficiencies. The arithmetic units inside the ALU or FU are connected in an either-or fashion, meaning that only one of the arithmetic units multiplier or adder can be used in a single pass through the functional unit. This means that when a large program is spatially mapped on to the CGRA at least half of the arithmetic units remain unutilized because of the architectural design choices and the FU structure of the core.

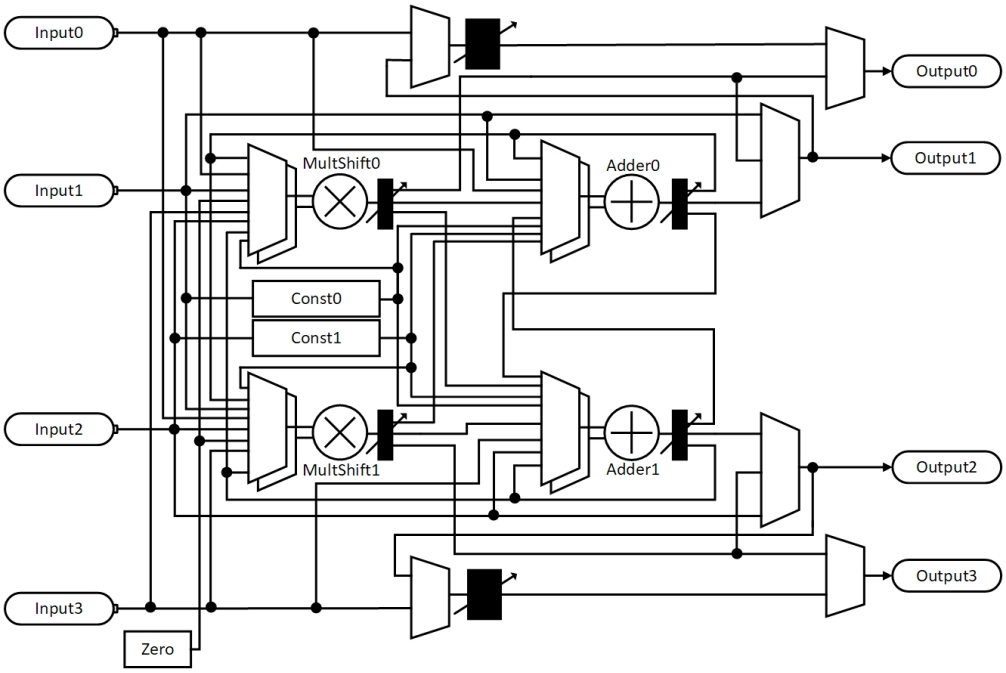


Figure 3.4: Core architecture of UDSP.

Instead of going with the traditional ALUs, we design our core with the target program

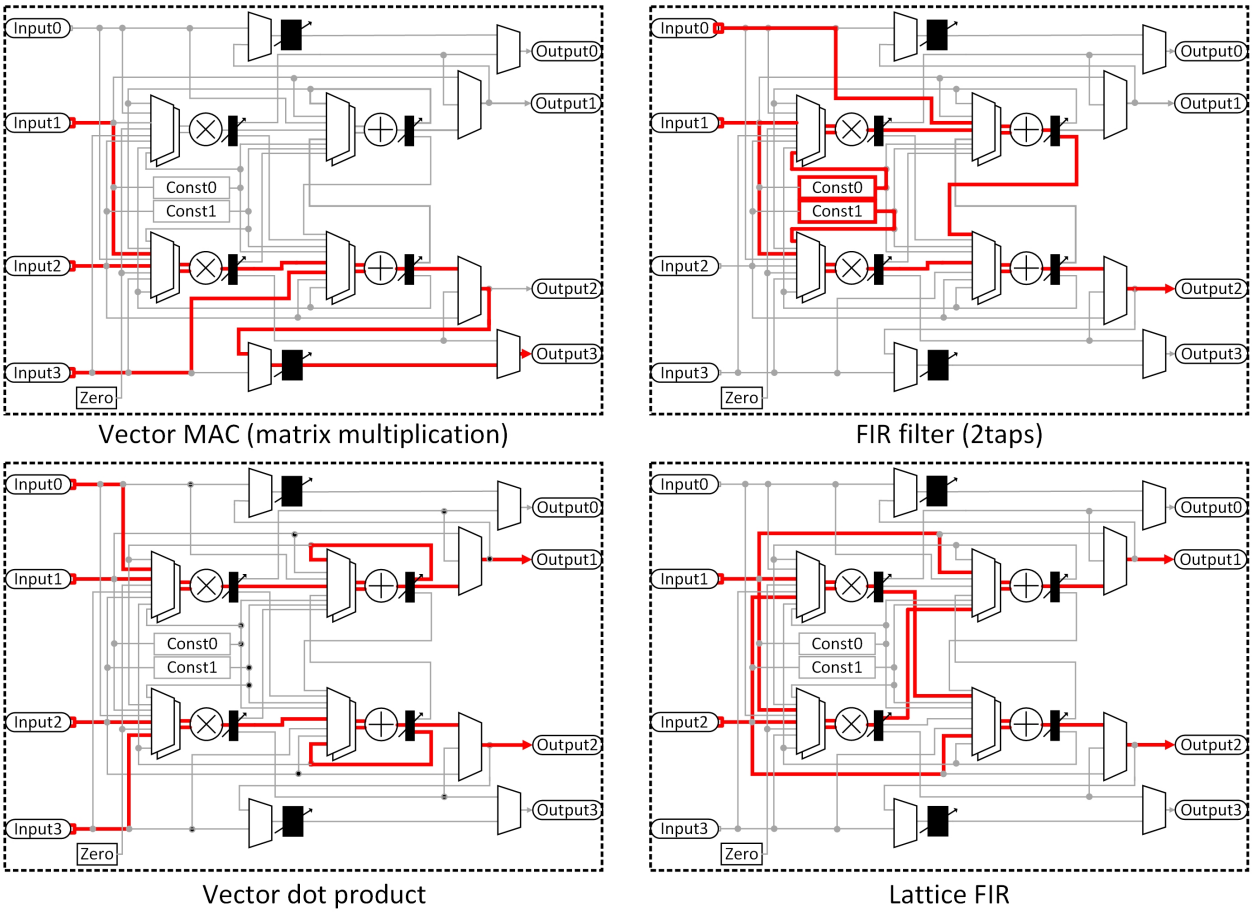


Figure 3.5: Some example kernels mapped onto UDSP core.

kernels and domain in mind. We identify and reuse the common elements in the program flow graphs and repeating kernels. The program kernels belonging to the DSP domain have a multiplier-to-adder ratio of nearly 1, which means that for every multiplier used there is a high probability that an adder would be mapped next. Additionally some of the kernels with complex arithmetic also require heavy use of multipliers and adders with a high degree of communication or routing density between them, and the same is also true for filters like adaptive filters and parallel multi-rate FIR filters. So, the final core design contains 2 multipliers and 2 adders to ensure performance and sufficient mappability across the domain. However, the interconnect internal to the core between the multipliers and adders was an iterative process. We design a core iteration and try to map the target application kernels onto the core and over the course of various iterations the data paths were added and removed between the arithmetic units, and the final core architecture is shown in Figure 3.4. Some of the example mappings of program kernels mapped onto the UDSP core are shown in Figure 3.5 and additional mappings are shown in the results section in Figure 3.34 and Figure 3.35.

3.1.2 Routing Network

The routing network is used for communication between the PEs in a reconfigurable architecture. It drives the physical organization and topology of the architecture. The choice of routing network has implications on the performance, energy consumption, area, scalability, reliability and compilability of the architecture. Overdesigned routing network for a domain-specific architecture would mean inefficiencies in area and power, and an underdesigned/underprovisioned network would mean lack of flexibility, increased compilation times, mapping inefficiencies, complete lack of support for certain algorithms and network topologies and therefore large throughput penalties.

There are several classifications of on-chip network architectures based on their topology and interconnect policy [6]:

- Shared-medium networks: The link is shared amongst multiple nodes, and the nodes take turns to transmit data one node at a time as shown in Figure 3.6. An example for such networks is bus-based networks such as AMBA by ARM [4], or coreConnect by IBM [15].

- Direct networks: As the name suggests in this network topology the nodes which require communication are directly connected to other nodes and routers via point-to-point links, as shown in Figure 3.7. Examples of this topology are mesh-based networks and crossbar networks.

- Indirect networks: In this network topology each node is connected to a switch and switches have point-to-point connections to other switches as shown in Figure 3.8.

- Hybrid networks: This is a mixture of a combination of other network approaches. FPGA architectures use such a hybrid network between direct and indirect networks as shown in Figure 3.10.

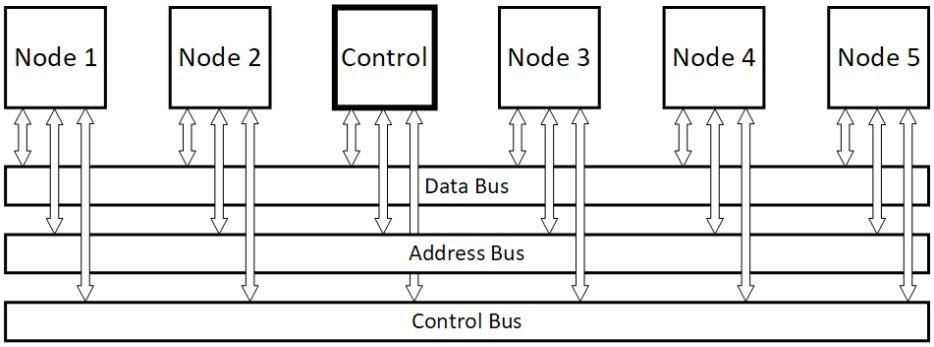


Figure 3.6: Shared-medium bus network.

Shared-medium bus networks requires an additional layer of protocol for arbitration strategies and bus moderators to control the use of bus by the connected nodes, additionally the cores need to be modified to pause the program execution to accommodate for network congestion and conflict issues on the bus. However our requirements for the architecture are streaming interconnects where the connections are fixed based on the program and do not

deviate once a program is spatially mapped on the array. Thus, based on the requirements we can rule out the use of shared medium networks.

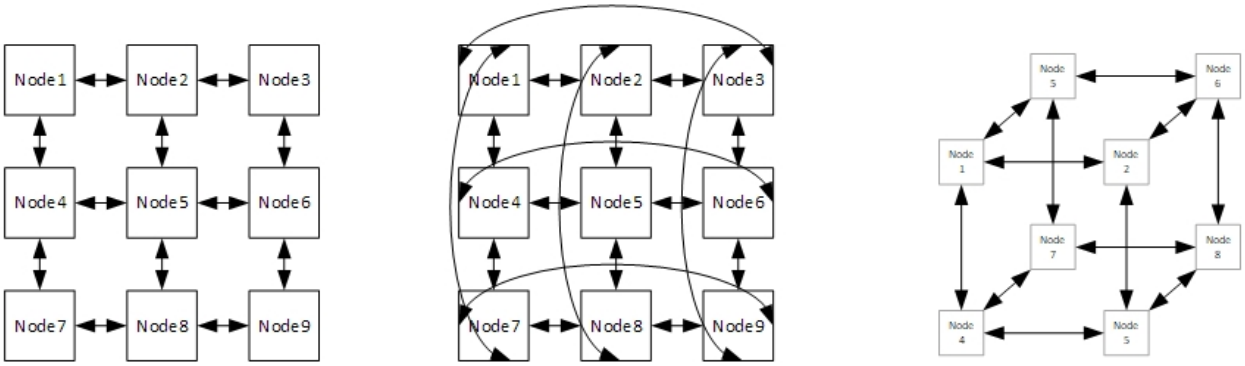


Figure 3.7: Examples of direct network, a 2-dimensional mesh, a 3-ary 2-cube torus, and a 3-dimensional hypercube.

Direct and indirect networks support the streaming data connectivity as required by our architecture. Direct networks as shown in Figure 3.7 have a distinct advantage as they overcome the scalability problems of shared medium networks. In this network topology each node is directly connected to a subset of neighbouring nodes in the network. Each node contains its functional units, or memory and in addition it contains the network interface block called router or a switch. The router or switch can be programmed to enable direct connectivity to the neighbours through the wires. Properties of direct networks have been extensively studied using graph modelling techniques. These networks are described using properties like node degree, network diameter, bisection bandwidth, path diversity, symmetricity and regularity.

Indirect networks can provide another scalable alternative to direct networks. The properties and graph modelling techniques used to describe direct networks can be used for indirect networks as well. Over the decades many topologies have been proposed for this network types, one of the simplest topology is a crossbar-based indirect network. Crossbar allows any connected node to be connected to any other connected node. Since crossbar is a fully connected network internally it scales with $order(N^2)$ with the number of nodes in the system.

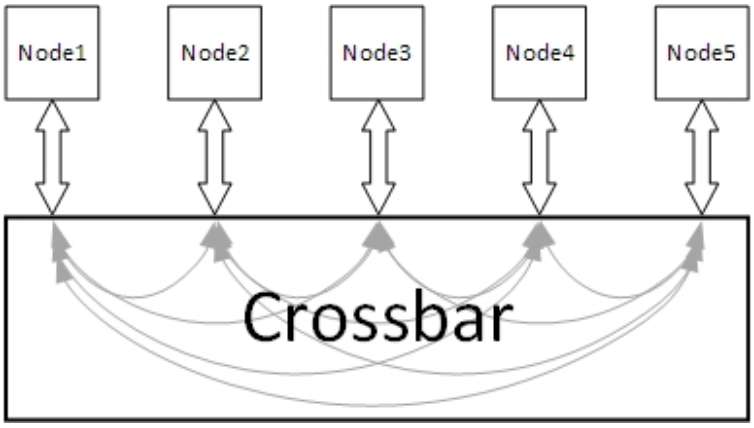


Figure 3.8: Examples of indirect network, with a fully-connected crossbar.

Several solutions have been proposed to improve the scalability of the crossbar. Benes network introduces a multi layer crossbar which improves the scalability from $order(N^2)$ to $order(Nlog_2(N))$, as seen in Figure 3.9 [49].

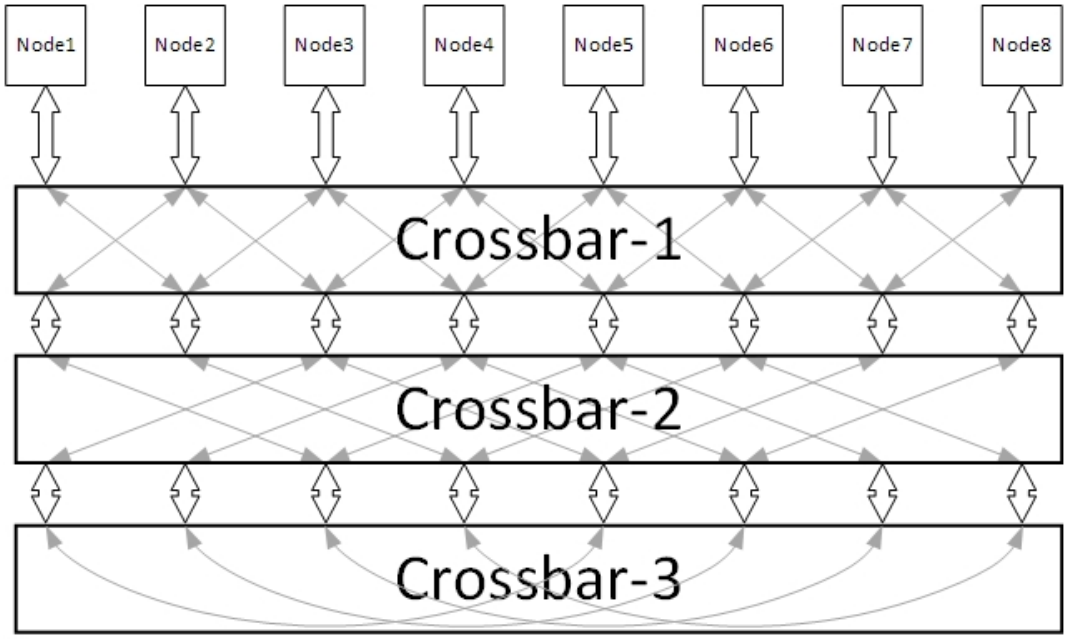


Figure 3.9: Examples of indirect network, with a fully-connected crossbar.

Some of the commonly used FPGA architectures employ indirect network using multiple crossbars with crossbar-crossbar connections as shown in Figure 3.10. Such an architecture solves the scalability problem at the larger scale, since the mesh is scalable with array size

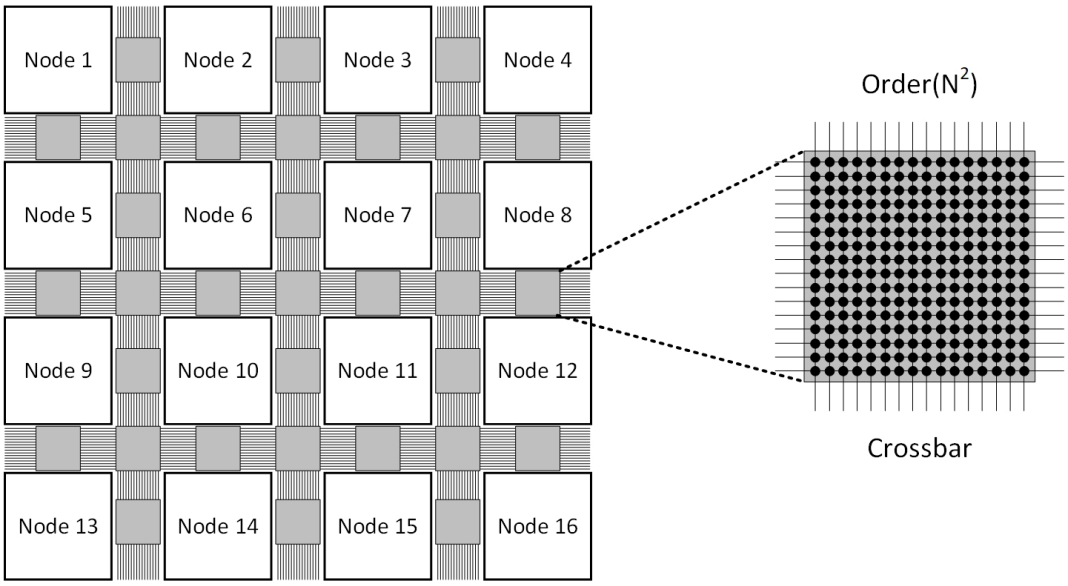


Figure 3.10: Examples of FPGA network, with a mesh interconnect and crossbar-crossbar connections.

but the crossbar is still not scalable. These crossbar-based multi-stage networks have several other problems. The crossbars have a fixed bandwidth among them which can result in blocking and network congestions. The crossbar and switches once used to connect an input-output node pair cannot be used for any other connection, thus the connection is blocked. The bigger problem, however is regarding the compiler friendliness. The number and the selection of crossbars a connection has to pass through to achieve a successful connectivity requirement is non-deterministic. A crossbar network provides a large number of solutions to connect an input on any one of the nodes to outputs on any other node. In a reconfigurable array using such a network architecture, upon placement of the program, the compiler has to perform a high effort routing, the solution to which is not guaranteed because of blocking, and the path taken is non-deterministic because of high number of possible solutions. This means that FPGA compilers using such a interconnect network require multiple high effort iterations of placement, mapping, routing and timing checks to map the input program to the array, and the timing requested by the user may still not be guaranteed.

Based on the above analysis of the domain of available network architectures and their

benefits and shortcomings, we chose to use direct network as our network of choice for UDSP and now we explore and adjust parameters for the direct network that would be optimal for our architecture.

We explore the domain of digital signal processing and its requirements by developing a graph theory driven methodology to understand the requirements of the domain. We extract the graph characteristics of kernels belonging to the domain by mapping the kernels to a connected graph as shown in Figure 3.3 and then calculating the average node degree, and its standard deviation. We generate several random graphs based on the extracted characteristics. The degree of distribution of node degrees of any randomly generated graph follows a Poisson distribution. Figure 3.11a shows an example of such randomly generated graphs, the graph shows the degree distribution of a randomly generated DSP-like graph with 500 nodes.

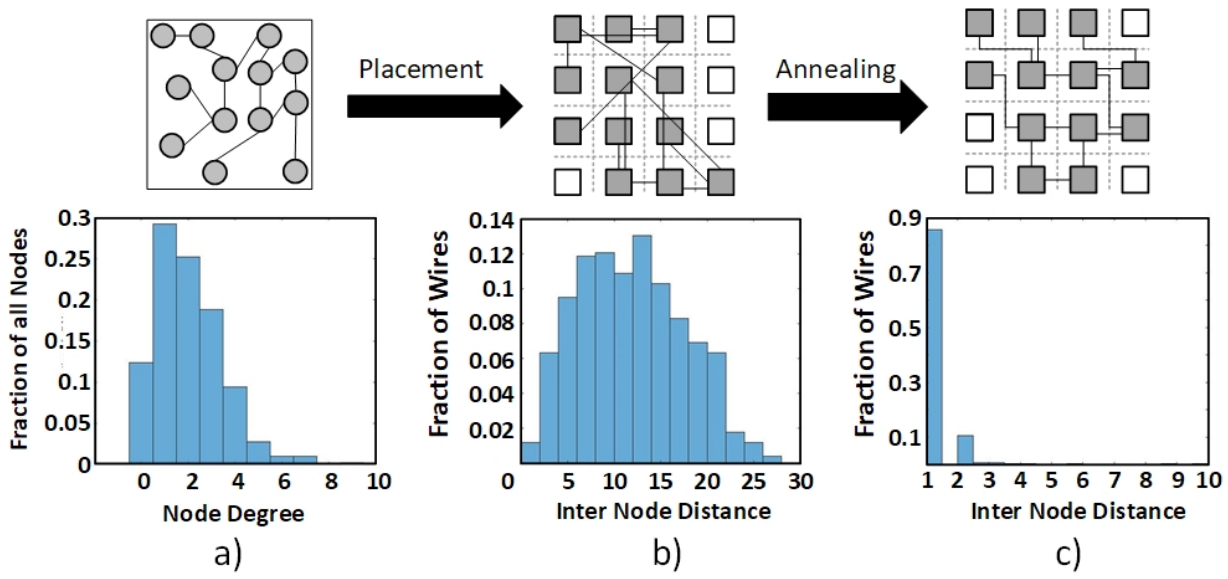


Figure 3.11: Network requirement analysis of the DSP domain.

Hundreds of graphs generated this way were mapped to the 2D array of UDSP and the inter node distances of the connected nodes were calculated and plotted against the total

number of wires as a fraction as shown in Figure 3.11b. Since this is a brute force mapping without optimizations there is a large fraction of wires with high inter node distances. At this stage we apply the method of simulated annealing that we developed in parallel for our software compiler. In this annealing approach we place the highly connected cores closer to each other based on a cost function given by the Equation 3.1. The final result after performing the annealing operation on sub-optimally placed nodes on a 2D array looks like Figure 3.11c. The plot is an example for just one 500 node DSP-like graph, here we can observe that most of the wires in the mapping have an inter-node distance 1, and nearly 90% of all the wires lie within the inter-node distance of 3 as shown in Figure 3.12.

$$Wiring\ Cost = \sum_{n=1}^{N_{wires}} \left| n_{xlen}^2 + n_{ylen}^2 \right|^2 \tag{3.1}$$

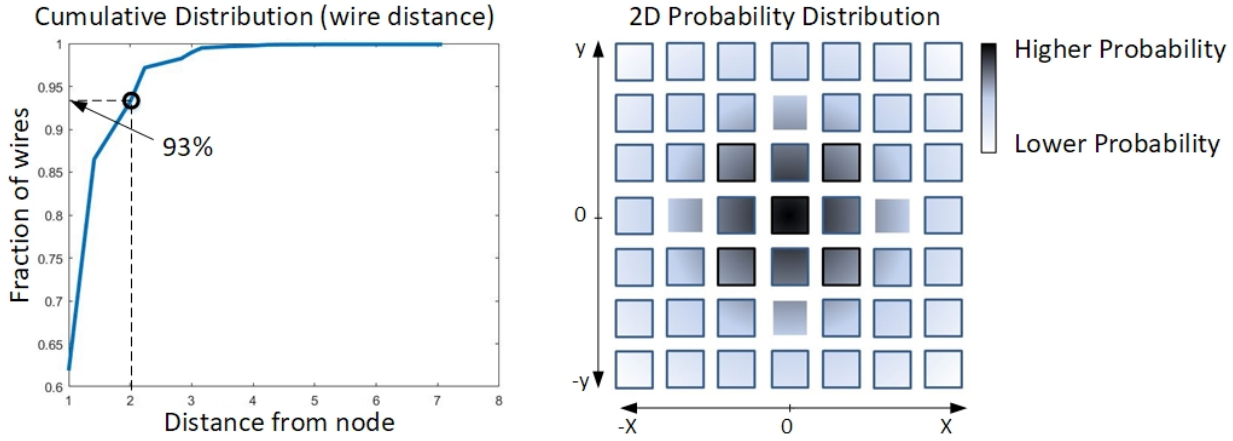


Figure 3.12: Cumulative Distribution Function (CDF) of the fraction of wires less than a wire distance, and a 2D Probability Distribution Function (PDF) of the fraction of wires for a wire distance.

There are several optimization tools in the bag of tricks of the compiler, and the placement optimization and simulated annealing is just one of many. Although the distance 2 hop network can accommodate over 93% of inter node connections after just annealing operation, the compiler can perform reclustering, remapping and retiming of the underlying algorithm to accommodate any outlier programs. Hence we make a design choice to implement a direct

network of upto distance 2 hop delay-less deterministic connections, and as a precaution add a very sparsely connected buffered, registered connections to accommodate any unpredictable and unroutable connections.

In addition to the reclustering, remapping and retiming tricks, we could also tune the order parameter of the wiring cost 'p', as shown in Equation 3.2. Since wiring cost is a non linear function, a higher value of 'p' would penalize the longer connections or the connected cores placed further apart much more than the connected cores placed closer together. The increased value of 'p' would result in shorter connections and the connected cores being placed closer together, however, it would come at the cost of longer convergence times as the compiler has to exert higher effort to satisfy and minimize the cost function, which is a reasonable trade off since compilation is a one time operation for multiple executions of the program.

$$Wiring\ Cost = \sum_{n=1}^{N_{wires}} \left| n_{xlen}^2 + n_{ylen}^2 \right|^p \tag{3.2}$$

3.1.3 Switchbox

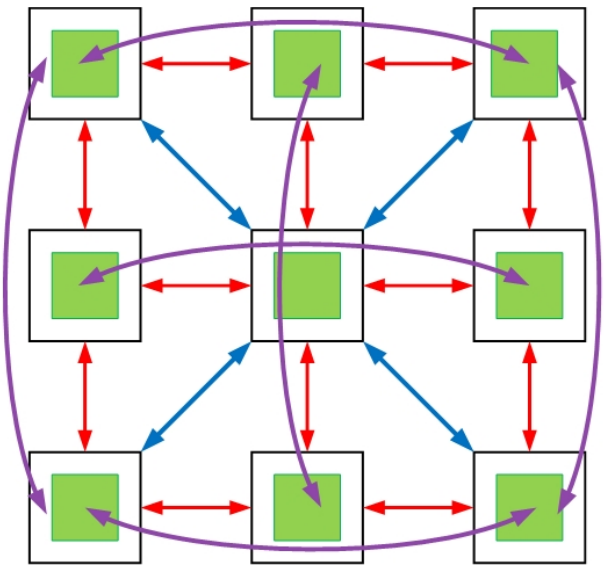


Figure 3.13: Target connectivity required from interconnect network.

Now that we have explored the domain and selected the type of interconnect network required, we can dig deeper into the switchboxes required to implement that direct network. We have calculated that the desired level of connectivity required from the interconnect network is an upto distance 2 mesh network as shown in Figure 3.13. In this section, we look into designing an optimal switchbox with minimal area and energy overhead to the area, while providing optimal connectivity.

As an abstract, to make the task of designing switchboxes simpler we divide the problem of switchbox design into 3 layers. The layer-1 switchboxes provides connectivity of nearest neighbor nodes upto distance 1 away in a '+' fashion, layer-2 switchboxes provide connectivity of nearest neighbour nodes upto distance $\sqrt{2}$ away in a '×' fashion and layer 3 switchboxes provide connectivity of nearest neighbour nodes upto distance 2 away in a '+' fashion. The first three layers of switchboxes are delayless and the fourth layer with registered connectivity is added to accommodate the outlier connectivity cases that cannot be fulfilled using the first three layers as shown in Figure 3.14.

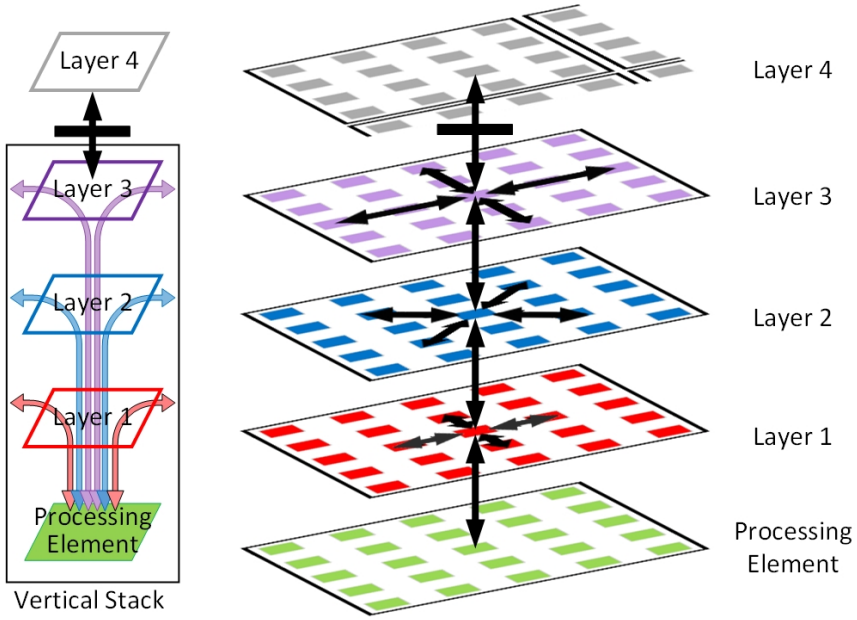


Figure 3.14: The full stack of connectivity switchbox layers in the vertical stack of UDSP .

A switchbox in Layer 4 spans 4×4 vertical stack grid, one of the ports in each VS in

the grid connects to layer 4 switchbox. Layer 4 switchbox is fully connected internally and connects to its neighboring layer 4 switchboxes in a '+' fashion. The connections in layer 4 switchbox are all single delay registered to allow the synthesis and layout tools to use smaller size gates and thus save area and power.

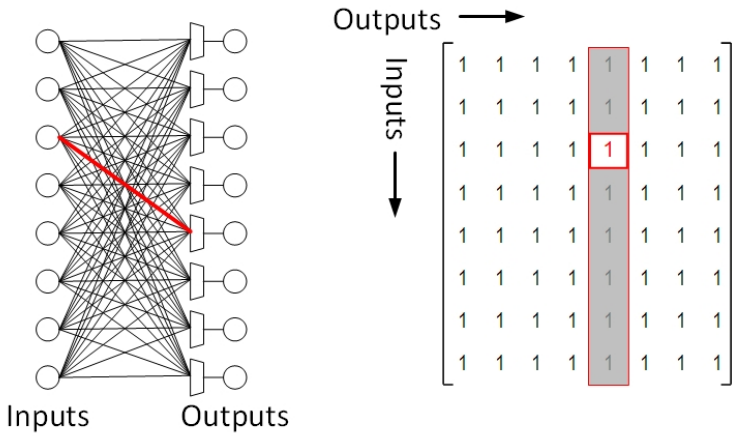


Figure 3.15: A fully-connected switchbox and its connectivity matrix.

Once the topology of routing network and inter switchbox connectivity is decided we can focus on designing the optimal switchbox designs. Traditional switchbox designs use fully connected networks inside the switchbox and have an $order(N^2)$ hardware, where N is the number of inputs/outputs and internal network complexity as shown in Figure 3.15. As an example shown in Figure 3.15, to route a connection from input 3 to output 5, the highlighted wire is chosen which is also represented by a '1' in the $(5, 3)$ location in the connectivity matrix. Similarly, hierarchical switchboxes have been devised which can provide the least redundancy for the hardware complexity of $order(Nlog_2(N))$ as shown in Figure 3.16. As an example, we show the same connection from input 3 to output 5 but routed via multiple layers of the hierarchical network.

These two architectures form the extremes of architectural design choices. While the fully connected network provides the best connectivity it comes at the cost of extra hardware and the hierarchical network provides the least redundancy at very low hardware cost, it does not provide great connectivity and suffers from blocking issue. As seen in the previous

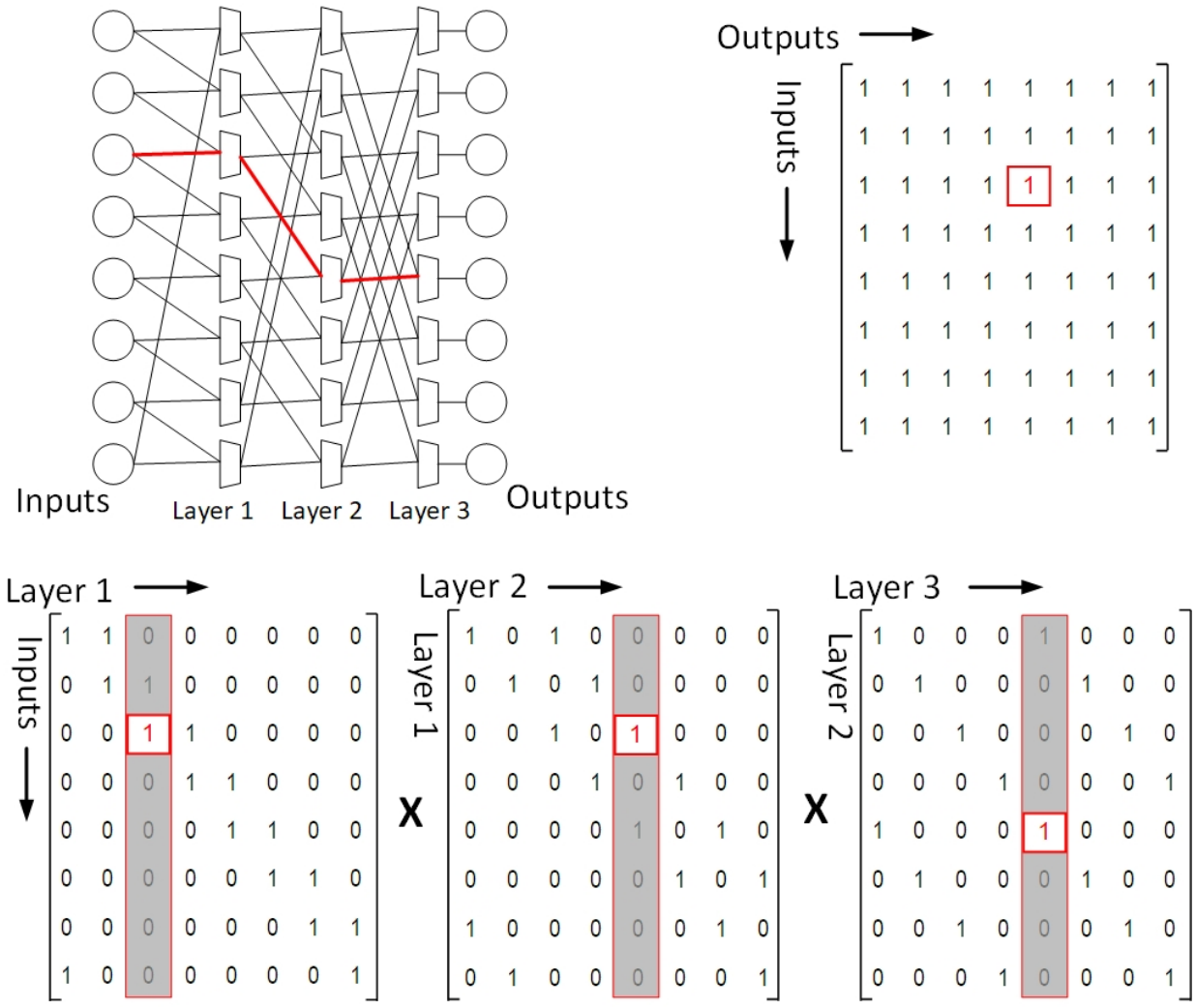


Figure 3.16: A hierarchical switchbox and connectivity matrices of its various intermediate layers and final input-output connectivity matrix.

connectivity example, once we have routed the path from input 3 to output 5 on the hierarchical network, the 3^{rd} mux in layer 1 and 5^{th} mux in layer 2 and layer 5 are completely occupied, and thus any attempt at routing any other connections would fail 100% of the time. For example, $3 \rightarrow 5$ connection and $2 \rightarrow 3$ connection cannot be made simultaneously in the hierarchical switchbox shown in Figure 3.16. Additionally these switchbox designs assume that all inputs would need to be mapped to outputs, however as per our network design exploration study, we need a limited connectivity bandwidth from these switchboxes. Although the switchbox in layer 1 has 22 inputs coming into it only 8 of them would be actively routed at any moment, as the core only has 4 inputs and 4 outputs.

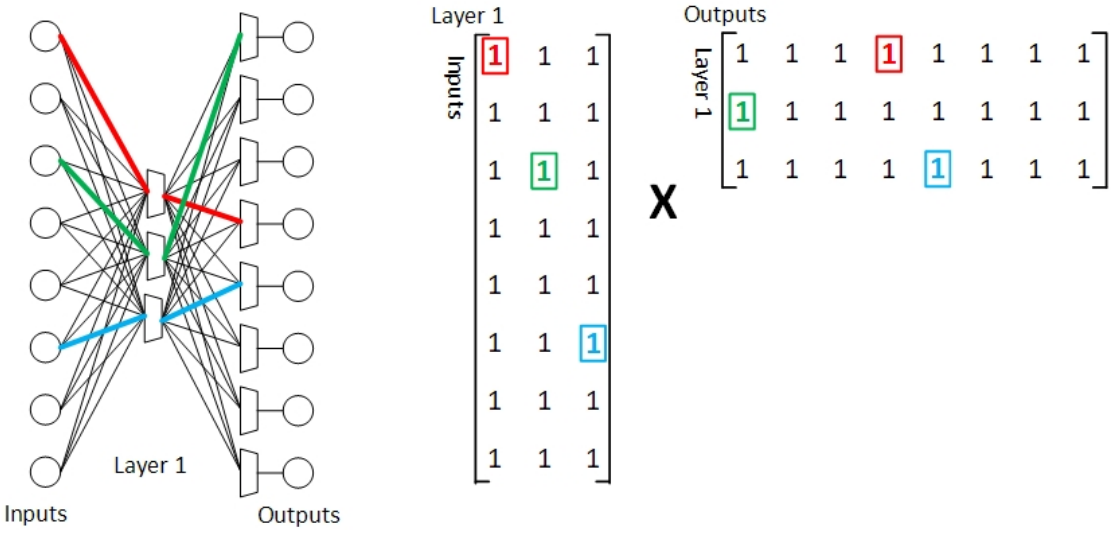


Figure 3.17: A throughput-constricted fully-connected switchbox using 1 middle layer to restrict throughput.

We design a throughput-constricted multilayer switchbox using fully connected network, where the middle layer is used to constrict the throughput of a fully connected network as shown in Figure 3.17. The hardware complexity of a fully connected switchbox can be calculated using the Equation 3.3. The equation essentially calculates the total numbers of connections in the switchbox. For the throughput-constricted fully connected switchbox as described in Figure 3.17 the hardware cost is $2 \times N \times b$, where N is the number of inputs/outputs and b is the number of muxes/tokens in the middle layer. Hence, in the

condition that the number of simultaneous active IO connections required from the switchbox is $b \leq N/2$, it is beneficial to go with a multilayer switchbox.

$$Hardware\ Cost = \sum_{all\ layers} \sum_{i}^{row} \sum_{j}^{column} C_{i,j} \tag{3.3}$$

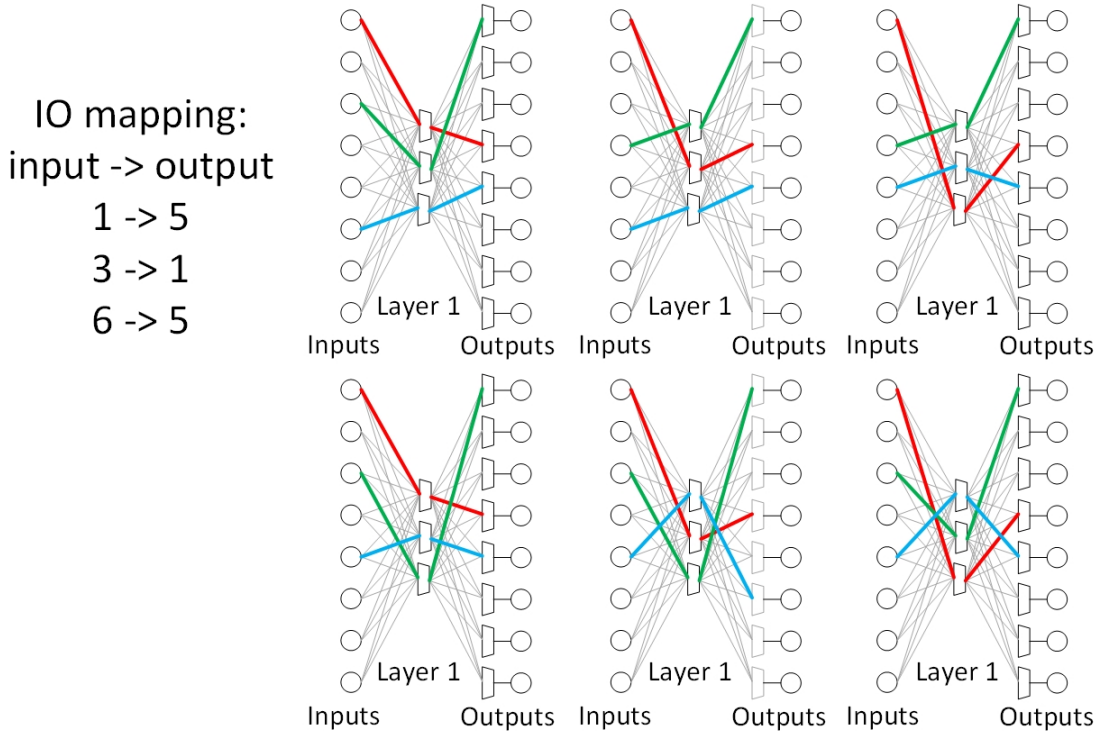


Figure 3.18: 6 different redundant ways to achieve the same IO mapping in a multilayer fully connected switchbox.

However, we can improve upon the hardware cost of this switchbox even further. Fully connected networks are ideal for on-the-fly connectivity with requirements for incremental connections, which means that in the scenario where a few of the connections through the switchbox are already connected then routing one additional connection through the switchbox is very easy, since all the middle layer muxes have identical and full coverage to the outputs. Another way to describe the problem would be that in multilayer fully connected switchbox there exists multiple paths and routes from the inputs to outputs to accommodate the same connectivity and routing requirements as shown in Figure 3.18. But, our design is

a precompiled solution, and the configuration for switchbox is predetermined, and no incremental connections are required. So we have room to introduce sparsity to our multilayer fully connected switchbox, to reduce some of the redundant connections.

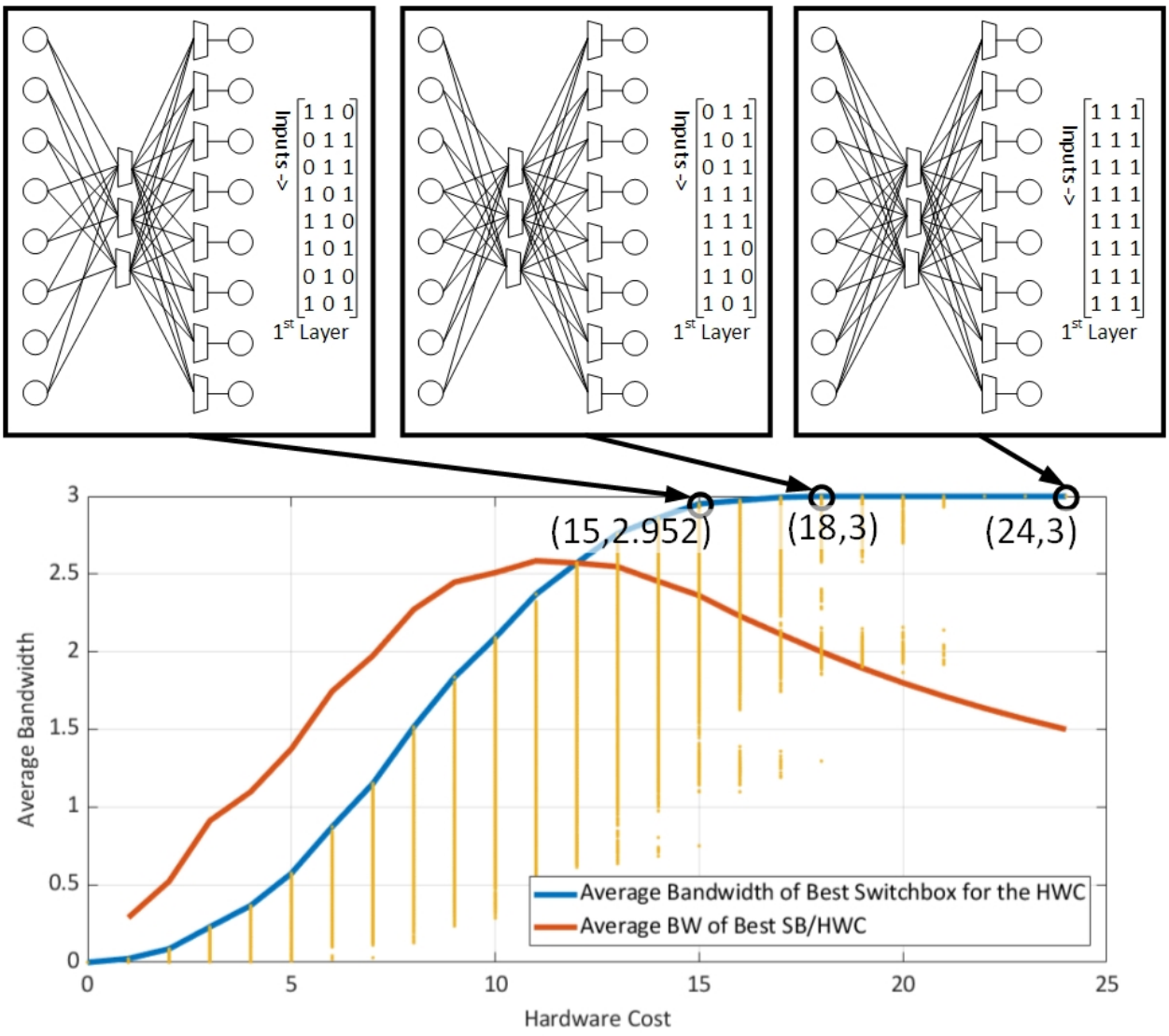


Figure 3.19: Sparse switchboxes with 8 inputs and 3 muxes, and their average bandwidth vs hardware cost for 1^{st} layer.

We introduce sparsity into our switchbox designs by randomly removing wiring connections from the switchbox and measuring its average bandwidth, which in this case is equivalent to the number of simultaneously active IO connections. As we can observe in Fig-

ure 3.19, going from a switchbox with first layer hardware cost 24 to another with hardware cost 18 has no effect in the average bandwidth, beyond which we observe some detriment in the average bandwidth as can be seen with example switchbox with hardware cost 15 and average bandwidth 2.952 IOs. The analysis of switchbox designs does not end here, as we can very well put 4 muxes in the middle layer, and analyze the system for bandwidth requirement of 3 IOs. As can be observed in Figure 3.20, there exists a switchbox with 4 muxes in the token layer which has a lower hardware cost of 12, it is very poor in serving bandwidth requirement of 4 IOs but does a great job to provide bandwidth 3 IOs.

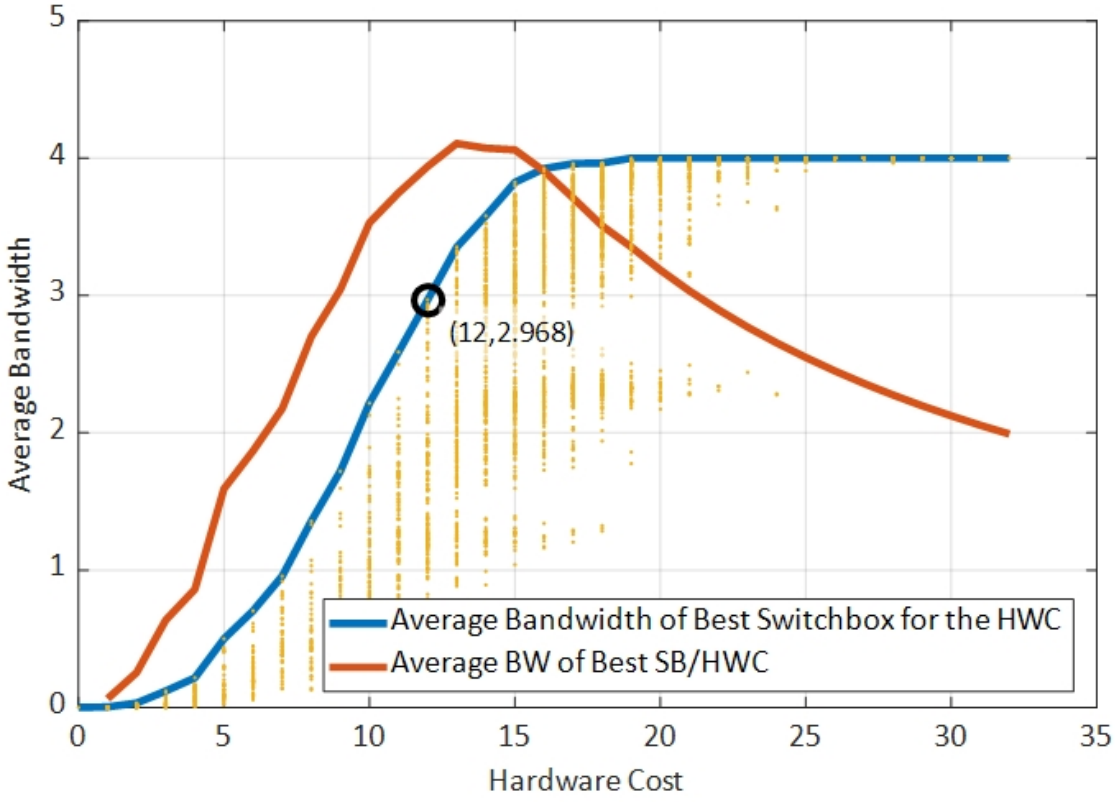


Figure 3.20: Sparse switchboxes with 8 inputs and 4 muxes, and their average bandwidth vs hardware cost of 1^{st} layer, to be used for network bandwidth requirement of 3 IOs.

The switchbox designed for switchbox layer 1 was analyzed extensively using this average bandwidth metric. Millions of switchboxes were randomly designed with varying connectivities in layer 1 and layer 2, and plotted in a 3D space with x and y axis representing their

layer 1 and layer 2 densities, and the z-axis representing the average bandwidth as seen in Figure 3.21. The saddle of the curve given by the MCBF plot is shown in Figure 3.22.

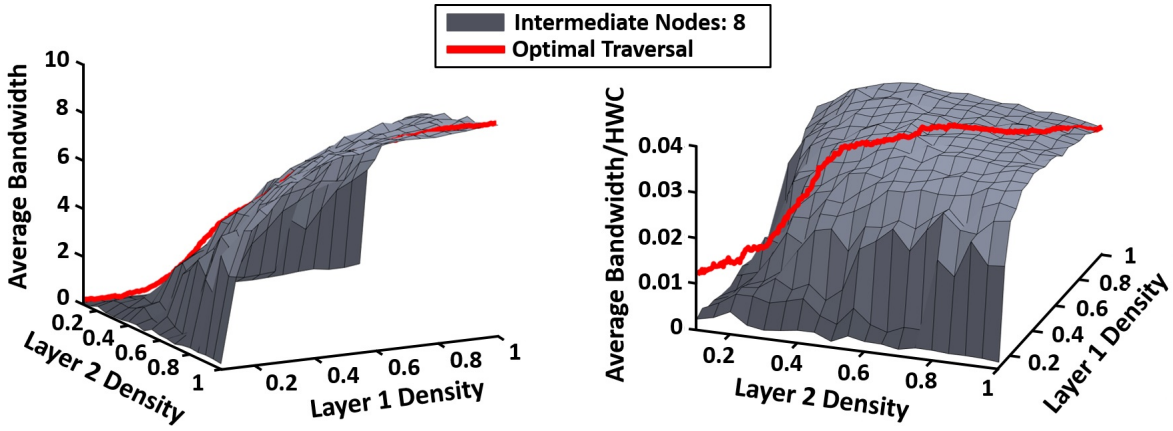


Figure 3.21: 3D representation of average bandwidth vs the connection density in the two layers of a multilayer switchbox with 22 inputs, 22 outputs and 8 muxes in intermediate layer.

3.1.4 Switchbox Design Methodology

To make the job of compiler easier after the hardware design phase, a design choice was made to keep the layer 2 of the multilayer switchbox fully connected and vary the connection density and hardware cost only in layer 1. The actual compilation process would be discussed in depth in the compiler chapter, here we go over the metrics used to design the switchbox. To maintain the symmetry and reliability of the switchbox connectivity over all the inputs, the number of inputs mapped to each multiplexer were kept constant at 8, and one of the inputs was hard wired to a zero, chosen as the default input, to ensure absence of loops and unnecessary dynamic power loss in the interconnect network at system startup . The number of inputs were so selected to keep the configuration bits required to program the switchbox minimum while maximizing the number of configurations, while also minimizing the hardware cost. The randomly generated connectivity matrices were optimized using the cost functions given by Equations 3.4, subject to constraints given by the Equations 3.5.

$$Minimize\,(\#shared\ muxes\ per\ input) = \sum_{i=1}^{\#rows} \sum_{j=1}^{\#rows(j\neq i)} row_i.row_j \tag{3.4a}$$

$$Minimize\,(\#shared\ inputs\ per\ muxes) = \sum_{i=1}^{\#columns} \sum_{j=1}^{\#columns(j\neq i)} column_i.column_j \tag{3.4b}$$

$$Number\ of\ muxes\ per\ input = \sum_{j=1}^{\#columns} A_{i,j} \tag{3.5a}$$

$$Number\ of\ inputs\ of each\ mux = \sum_{i=1}^{\#rows} A_{i,j} \tag{3.5b}$$

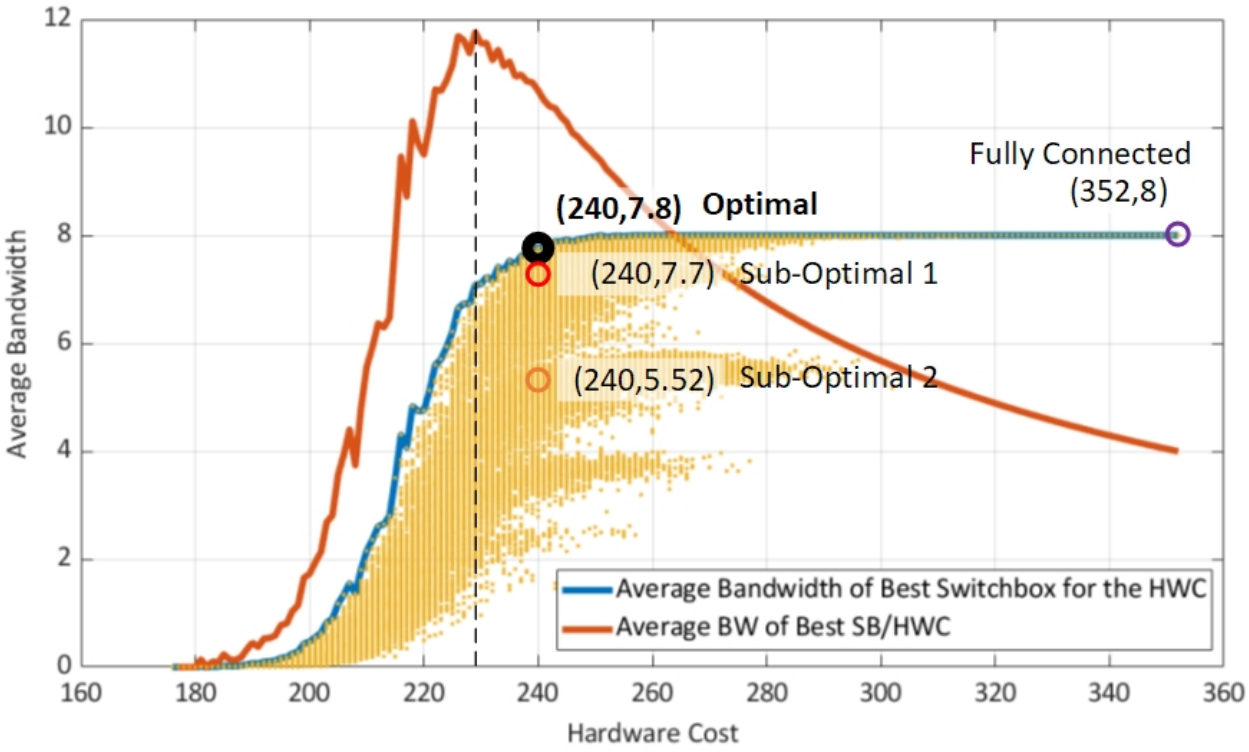


Figure 3.22: Average bandwidth vs hardware cost along the saddle curve of the 3D plot of various switchbox architectures with 22 inputs, 22 outputs and 8 muxes in intermediate layer.

The average bandwidth of millions of switchboxes generated this way were analyzed and plotted as shown in Figure 3.22. A few of the switchboxes and their bandwidth plots are

shown in the Figure as well, the switchbox that was selected is highlighted in the Figure 3.23 and its connectivity matrix is as shown in the Figure 3.24.

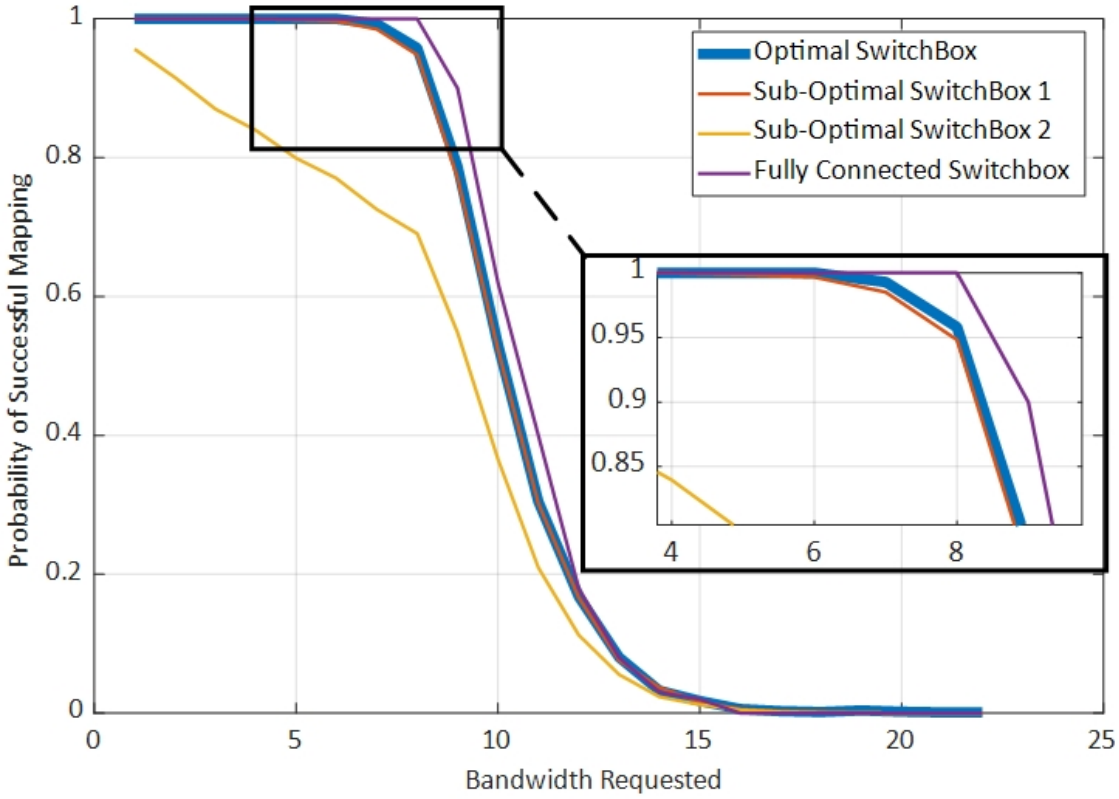


Figure 3.23: Probability that the bandwidth requested by the compiler would be satisfied by the selected switchbox design.

3.1.5 SNR-10 Channel

The Silicon Interconnect Fabric (Si-IF) is a high density fine-pitch interposer technology which can be used to provide high speed, high density interconnects between heterogenous Multi-Chip Modules (MCM). Using innovative solderless thermal compression bonding technique [5], Si-IF allows fine-pitch die-to-die interconnect with pad size of just $7\mu m \times 7\mu m$ and pad pitch of $10\mu m$. Some MCM designs have utilized traditional coarse-pitch based technologies that are quite mature and have a pad pitch ranging from $140\mu m$ to $160\mu m$ [52]. There are quite a few fine-pitch based technologies in development such as $55\mu m$ based EMIB [28],

```
SwitchBoxMatrix1 =

  23×8 cell array

    {'xxx'}    {'010'}    {'100'}    {'xxx'}    {'010'}    {'xxx'}    {'xxx'}    {'xxx'}
    {'xxx'}    {'011'}    {'xxx'}    {'xxx'}    {'011'}    {'xxx'}    {'xxx'}    {'011'}
    {'010'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'001'}    {'011'}    {'xxx'}
    {'xxx'}    {'100'}    {'xxx'}    {'xxx'}    {'xxx'}    {'010'}    {'100'}    {'xxx'}
    {'100'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'100'}    {'xxx'}    {'xxx'}
    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'110'}    {'101'}    {'xxx'}    {'xxx'}
    {'xxx'}    {'xxx'}    {'xxx'}    {'110'}    {'111'}    {'xxx'}    {'xxx'}    {'xxx'}
    {'101'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'110'}
    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'111'}    {'111'}
    {'xxx'}    {'xxx'}    {'101'}    {'111'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}
    {'110'}    {'xxx'}    {'110'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}
    {'xxx'}    {'110'}    {'xxx'}    {'xxx'}    {'xxx'}    {'110'}    {'xxx'}    {'xxx'}
    {'xxx'}    {'001'}    {'001'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'001'}
    {'xxx'}    {'xxx'}    {'010'}    {'xxx'}    {'xxx'}    {'xxx'}    {'001'}    {'010'}
    {'xxx'}    {'xxx'}    {'011'}    {'001'}    {'001'}    {'xxx'}    {'xxx'}    {'xxx'}
    {'001'}    {'xxx'}    {'xxx'}    {'010'}    {'xxx'}    {'xxx'}    {'010'}    {'xxx'}
    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'100'}    {'xxx'}    {'101'}    {'100'}
    {'xxx'}    {'xxx'}    {'xxx'}    {'011'}    {'101'}    {'xxx'}    {'110'}    {'xxx'}
    {'011'}    {'xxx'}    {'xxx'}    {'100'}    {'xxx'}    {'011'}    {'xxx'}    {'xxx'}
    {'xxx'}    {'101'}    {'xxx'}    {'101'}    {'xxx'}    {'xxx'}    {'xxx'}    {'101'}
    {'111'}    {'111'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}    {'xxx'}
    {'xxx'}    {'xxx'}    {'111'}    {'xxx'}    {'xxx'}    {'111'}    {'xxx'}    {'xxx'}
    {'000'}    {'000'}    {'000'}    {'000'}    {'000'}    {'000'}    {'000'}    {'000'}
```

Figure 3.24: Connectivity matrix of the layer-1 switchbox implemented in UDSP.

and $40\mu m$ based CoWoS [27].

As the physical dimensions of the pad shrink, the overhead of the circuitry to facilitate the communications through the smaller pads increases. Hence, the major problem with these fine-pitch interconnect technologies becomes reducing the size of the supporting circuits that enable the die-to-die communications on MCM with reduced overhead. Figure 3.25 shows a comparison between the area available per IO in different high bandwidth IO technologies. The circuit supporting the communication for Si-IF has to fit into an area 74.2× smaller than traditional IOs to eliminate the overhead and ensure high area efficiency and throughput gains from using such a fine-pitch interconnect.

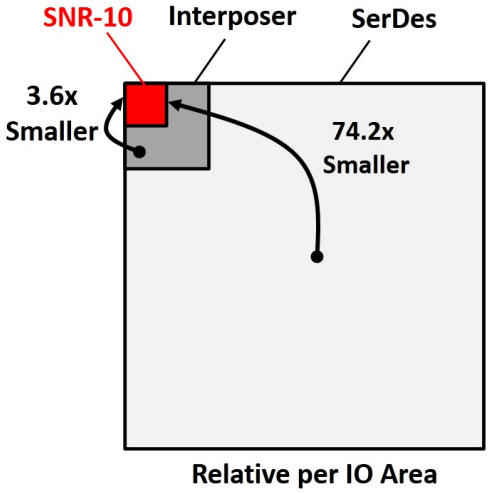


Figure 3.25: Area available per IO for Si-IF compared to SerDes [27], and interposer [52].

The SNR-10 channel is implemented in UDSP 14×14 as two flavours, a standard pad layout and uniform pad layout as shown in Figure 3.26. For each pad the internal circuit is as shown in Figure 3.27. The uniform pad layout has uniform pads each of size $7\mu m \times 7\mu m$, and it completely relies upon the active redundancy circuit built into the SNR-10 channel for any data link mishaps and errors, however it is not immune to any mishaps in the control critical and clocking link errors. The standard pad layout has two distinct pad sizes a normal $7\mu m \times 7\mu m$ for data links and larger $14\mu m \times 20\mu m$ pads for control links. The larger pads ensure that at least two copper pillars from Si-IF connect to the control pads and thus

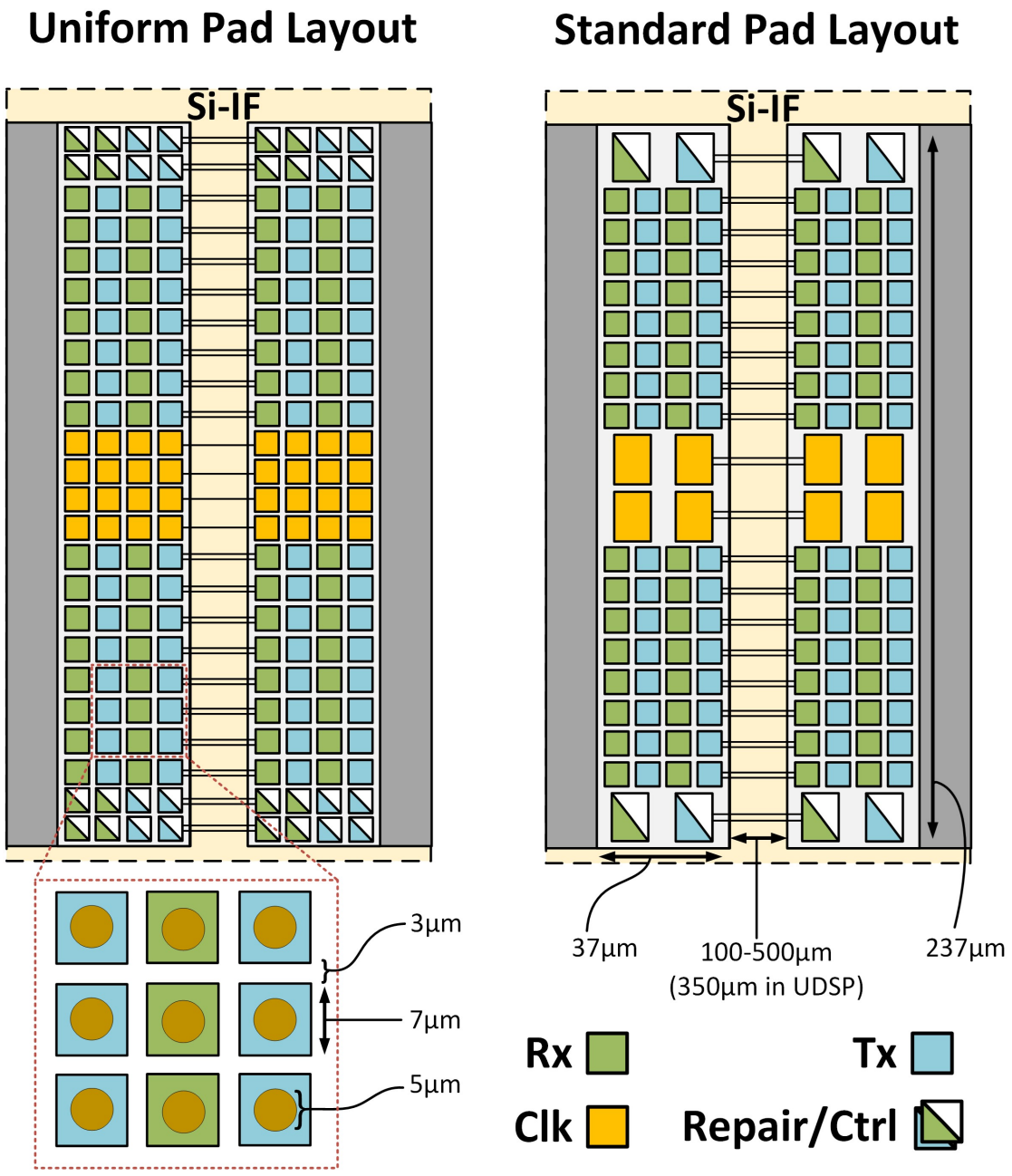


Figure 3.26: Uniform (horizontal channel on UDSP) and standard (vertical channel on UDSP) pad layouts of SNR channel for Si-IF interposer.

provide a physical redundancy to the critical paths.

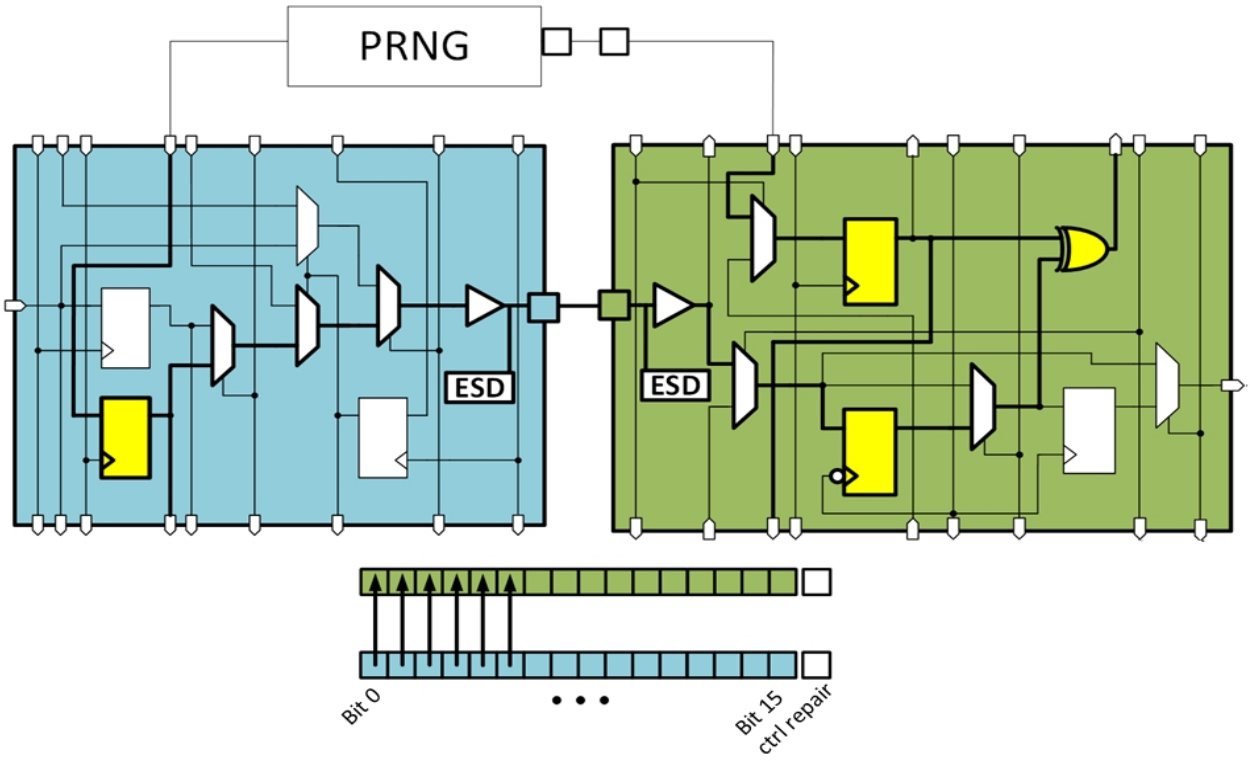


Figure 3.27: Internal circuit and mechanism per pad for SNR-10 channel.

The SNR-10 channel has a Pseudo Random Number Generator (PRNG) to perform the power-on self-test, to detect any faulty links and then it uses the redundant pads in the channel to route around and bypass the faulty Si-IF link. One of the benefits of having a smaller pad interfacing the real world is that the antenna effect of the pad is reduced, since the area of the pad is much smaller. This allows us to use smaller ESD diodes to protect the chip against any ESD event. The smaller diodes have a smaller load capacitance which helps reduce the energy consumption per bit transfered across the Si-IF channel. To test for channel performance across different modes of operations, the horizontal channel in the UDSP was hardwired to operate in asynchronous mode of operation, while the vertical channel operated as synchronous.

3.2 UDSP Chip Prototypes

The design of UDSP is modular and very repetitive. The vertical stack includes the compute core and the first 3 routing layers. The Verilog RTL for UDSP is generated automatically using a MATLAB script which stitches the Verilog vertical stack modules together and then connects them to layer 4 and control module. Since the process is automated and the scripts have been tested extensively, the RTL writing process of UDSP is greatly simplified. The automated scripts require the number of vertical stacks in x and y dimensions and outputs the toplevel Verilog for UDSP which can be directly passed through the synthesis and layout scripts.

The constraint input files for synthesis require several blocking and timing constraints. Automated Tcl scripts were written which could generate the Standard Delay Format (SDF) and Synopsys Design Constraint (SDC) file for any given number of input cores and designs. The automated process is quick and efficient, and took less than a month to go from conception to layout to verification to tapeout for UDSP 14×14 and just 2 weeks for Global Foundry 22nm 2×2.

The scalable and technology portable UDSP array design has been taped out as 9×9, 14×14, 15×15 arrays in TSMC 16nm FinFET and as a 2×2 array in Global Foundries 22nm FDSOI technology as seen in Figure 3.28. UDSP implementation on TSMC 16nm FinFET is shown in Figure 3.29. The design consists of 14×14 vertical stacks with 28 SNR-10 channels operating at 1.1GHz frequency at 0.8V nominal operating voltage. 14 vertical SNR-10 channels are configured to operate in a synchronous mode, while the 14 horizontal channels were configured as asynchronous channels. The 14×14 vertical stack implementation was then assembled on an Si-IF interposer in a single die configuration as shown in Figure 3.30. The same die was also assembled in a 2×2 die configuration on Si-IF interposer using the SNR channels to allow communication between the dies for a seamless interconnect network spanning 4 dies, thus providing a 28×28 vertical stack configuration as shown in Figure 3.31. The dimensions of UDSP on TSMC 16nm is $2.5mm \times 2.5mm$, it

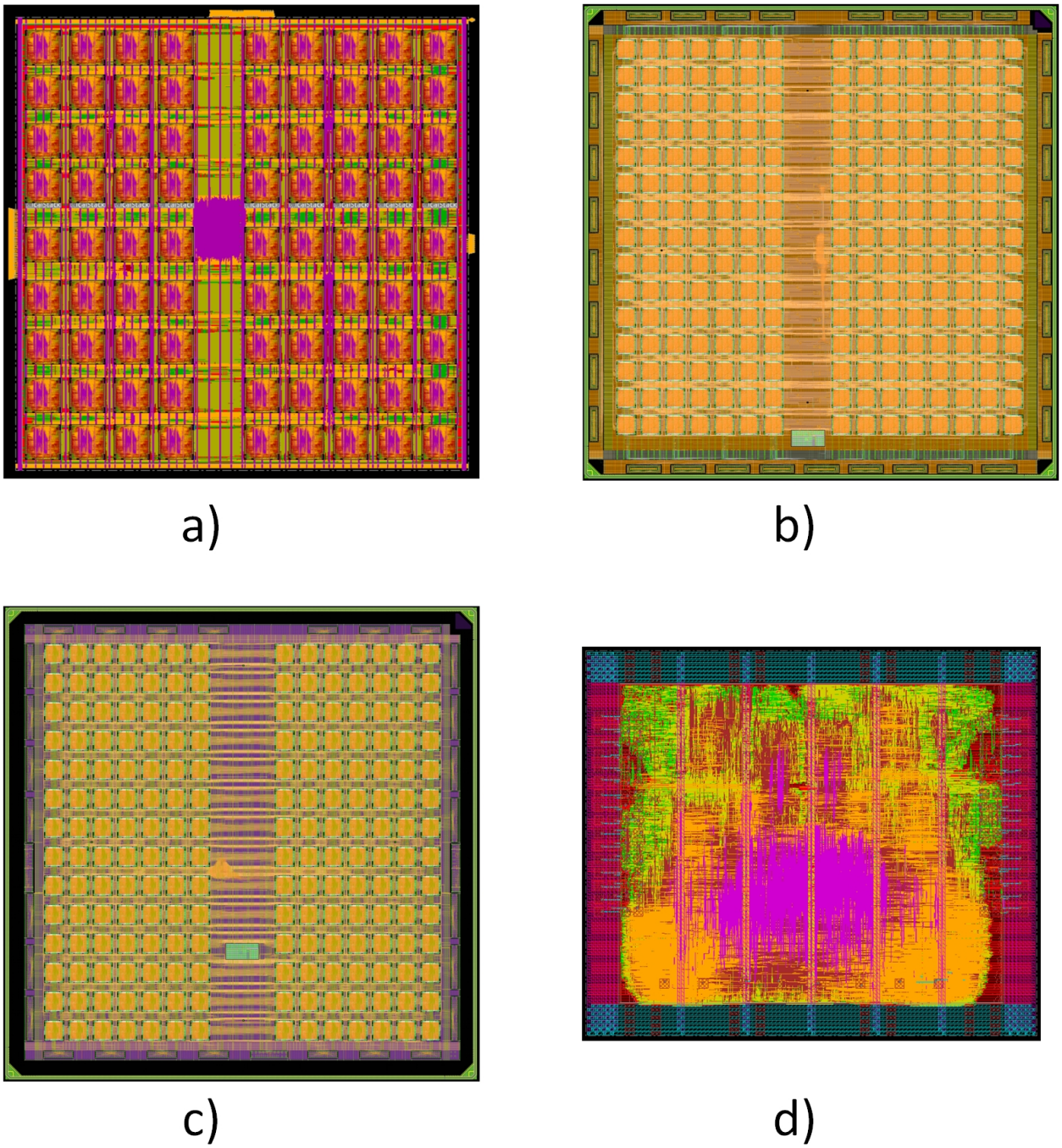


Figure 3.28: Various implementations of UDSP design a) TSMC 16nm UDSP 9×9. b) TSMC 16nm UDSP 15×15. c) TSMC 16nm UDSP 14×14. d) Global Foundry 22nm UDSP 2×2.

operates at 1.1GHz and consumes 1.5W power.

The UDSP design was also ported to Global Foundry 22nm FDXSOI technology as a 2×2 vertical stack design to test the core count scalability, portability and design compatibility across technologies, as shown in Figure 3.32. The process of generating a smaller core count design was automated using MATLAB scripts which stitches the vertical stacks modules, and connects their switchbox layered interconnects and wraps them with a toplevel wrapper and generates a synthesizable Verilog. The constraints required for synthesis and physical implementation were scripted as well, the entire process of generating RTL and porting the design from TSMC 16nm to Global Foundry 22nm process required 2 weeks. The dimensions of UDSP on Global Foundry 22nm is $345\mu m \times 295\mu m$, it operates at 500Mhz and consumes 16mW power at 0.8V operating voltage.

3.3 UDSP Results

The testing for UDSP functionality and mapping were performed on the TSMC 16nm UDSP 14×14 vertical stack dies assembled as 2×2 on the Si-IF interposer as shown in Figure 3.31.

3.3.1 Mapping Efficiency

We define the mapping efficiency of our vertical stack architecture as the number of unused functional units in the utilized cores. The metric quantifies the routing performance of interconnect network, as well as quantifies the core architecture inside the vertical stack over different algorithms and programs. Mapping programs with small kernels like Multiply Accumulate (MAC), lattice structures, and FIR filters can achieve near 100% mapping efficiencies as shown in Figure 3.34, while radix-2 kernel for Fast Fourier Transform (FFT) has a lower mapping efficiency of 42%, as shown in Figure 3.35. The constant multiply and pass through configuration of the vertical stack can be used to execute larger programs like fully connected neural networks and convolutional neural networks as shown in Figure 3.34.

Mapping efficiency of a program depends on the architecture of the hardware as well as

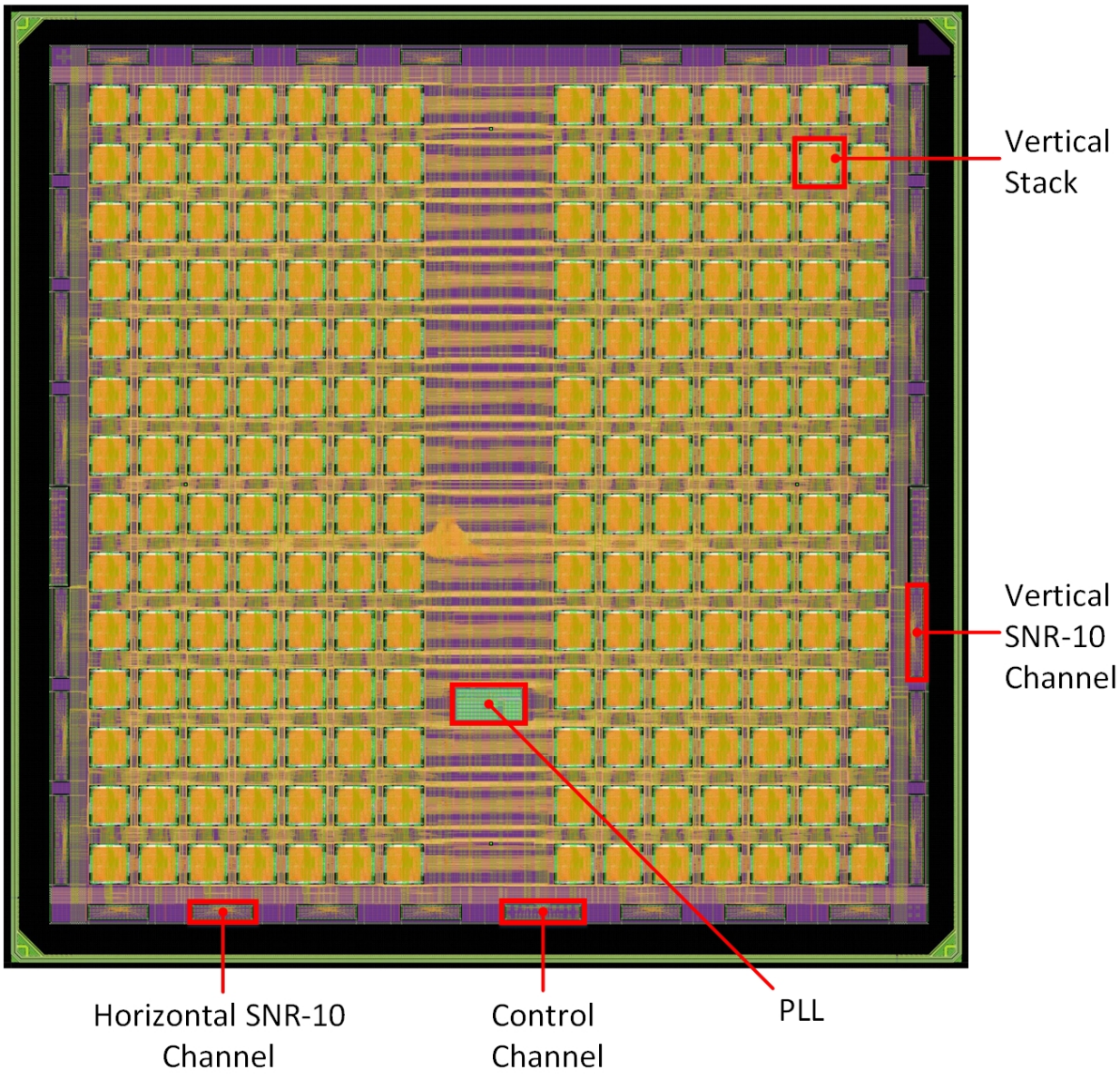


Figure 3.29: Physical layout of 14×14 vertical stack UDSP design in TSMC 16nm.

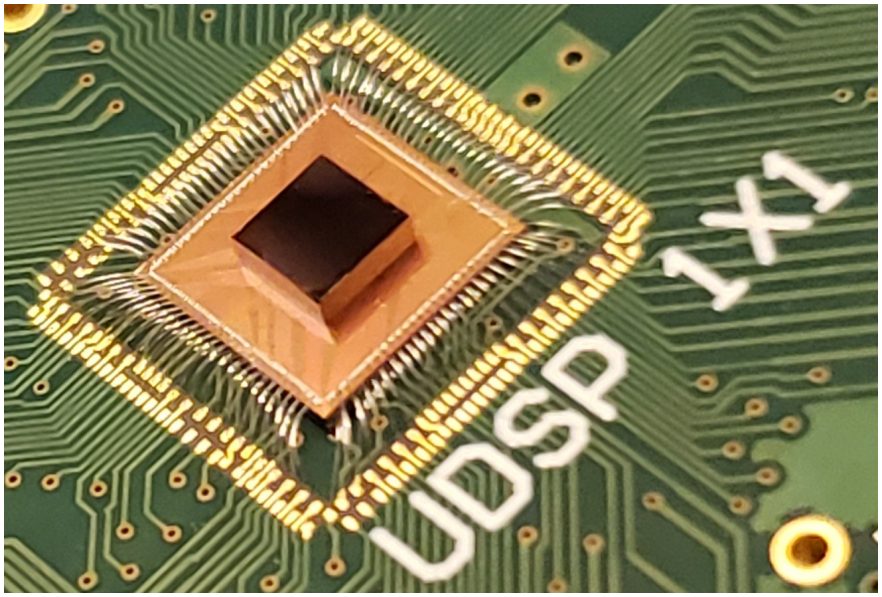


Figure 3.30: Die shot of single TSMC 16nm UDSP assembled on Si-IF interposer.

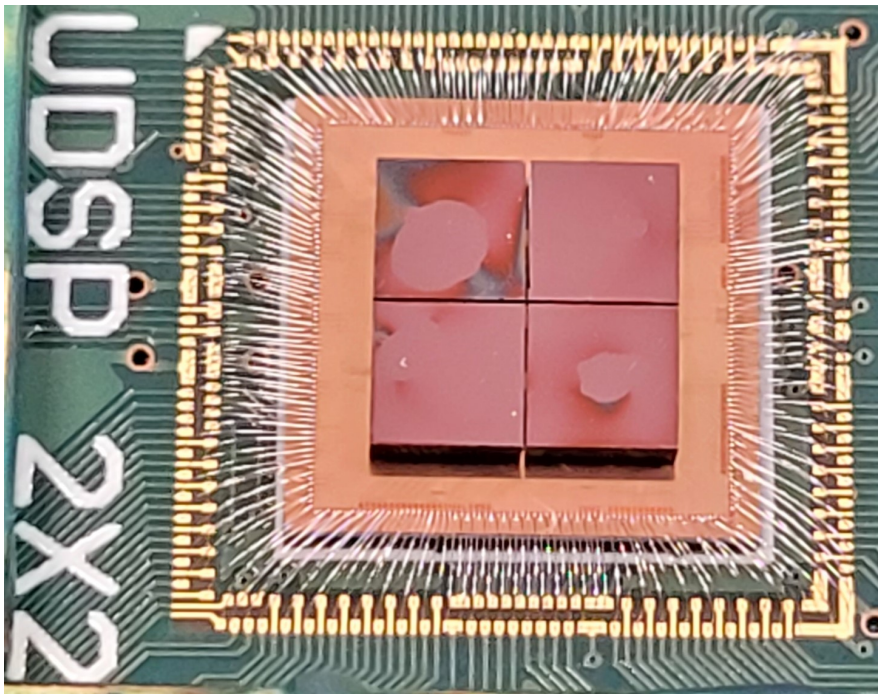


Figure 3.31: Die shot of TSMC 16nm UDSP assembled as 2×2 on Si-IF interposer.

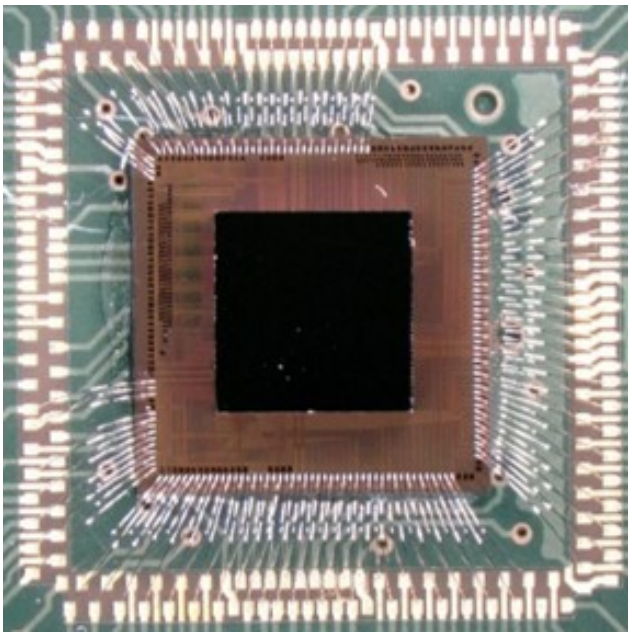


Figure 3.32: Die shot of Global Foundry 22nm UDSP assembled on Si-IF interposer.

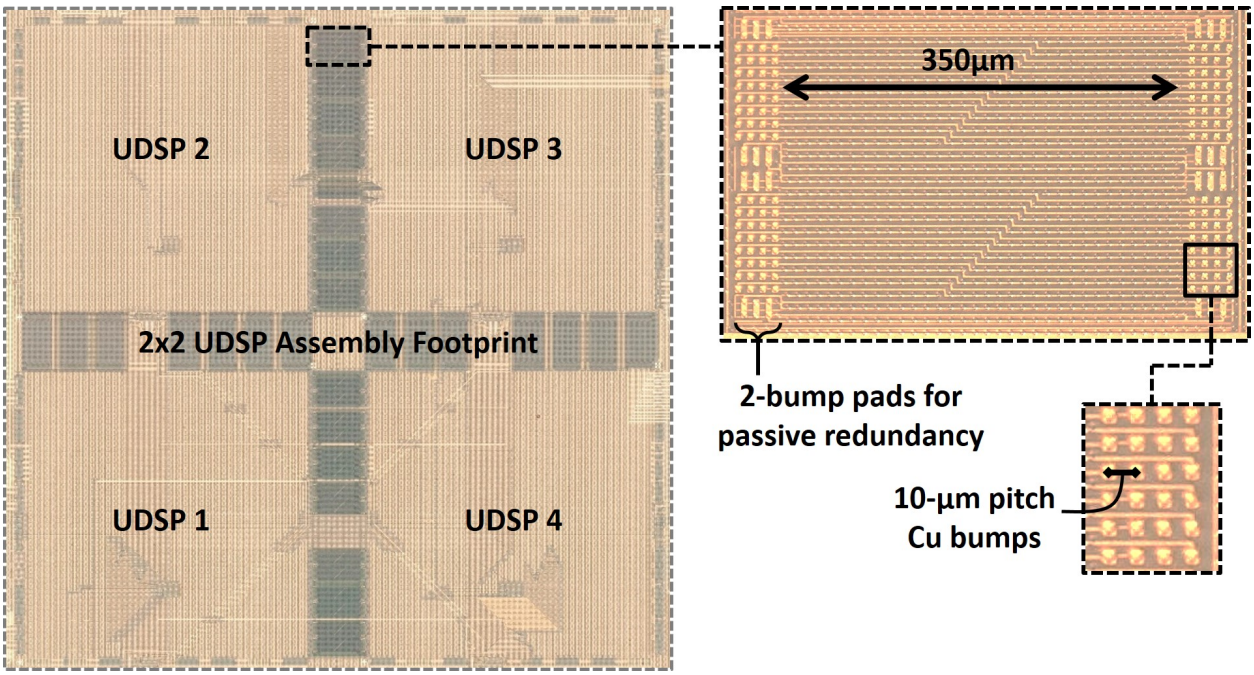


Figure 3.33: Si-IF interposer for 2×2 UDSP.

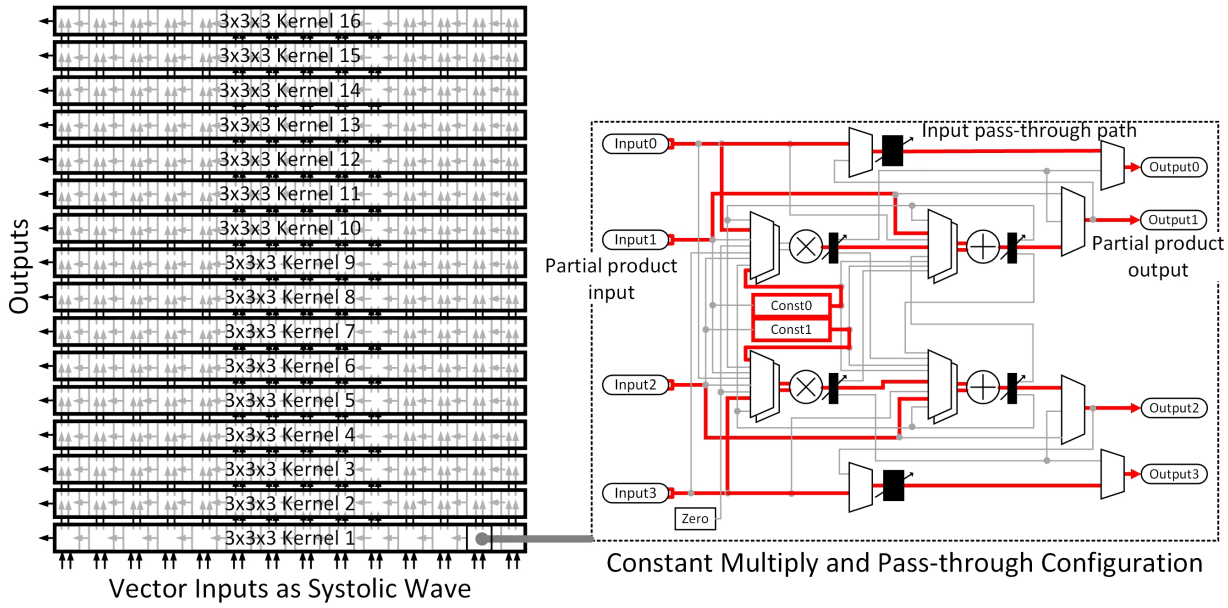


Figure 3.34: UDSP program mapping for 16 out of 32 parallel $3 \times 3 \times 3$ kernels required for first convolutional layer for MobileNet [19]. In the mapping all of the functional units inside the vertical stack are used for the program thus resulting in a 100% mapping efficiency.

on the capabilities of the compiler. Further work needs to be done to improve the mapping efficiency of completely automated compilers to manual levels.

3.3.2 Energy and Area Efficiency

Figure 3.36 shows the power and frequency scaling with supply voltage for UDSP. The maximum frequency of 1.1GHz is reached at 0.8V while consuming 6W of power, and the peak energy efficiency of 785GMACs/J is achieved at 0.42V while operating at 315MHz. The maximum frequency at each voltage was tested by checking the stability of vertical stack across the SNR boundary. VSs were configured to generate sawtooth waveform and pass the waveform to the neighbouring core across dielet boundary, the VSs subtract their internal waveform from the received waveform. This new subtracted waveform is then read out and checked for stability. Under stable operating conditions the waveform should read out a constant '0'. Under conditions of instability, with increasing frequency at each voltage step,

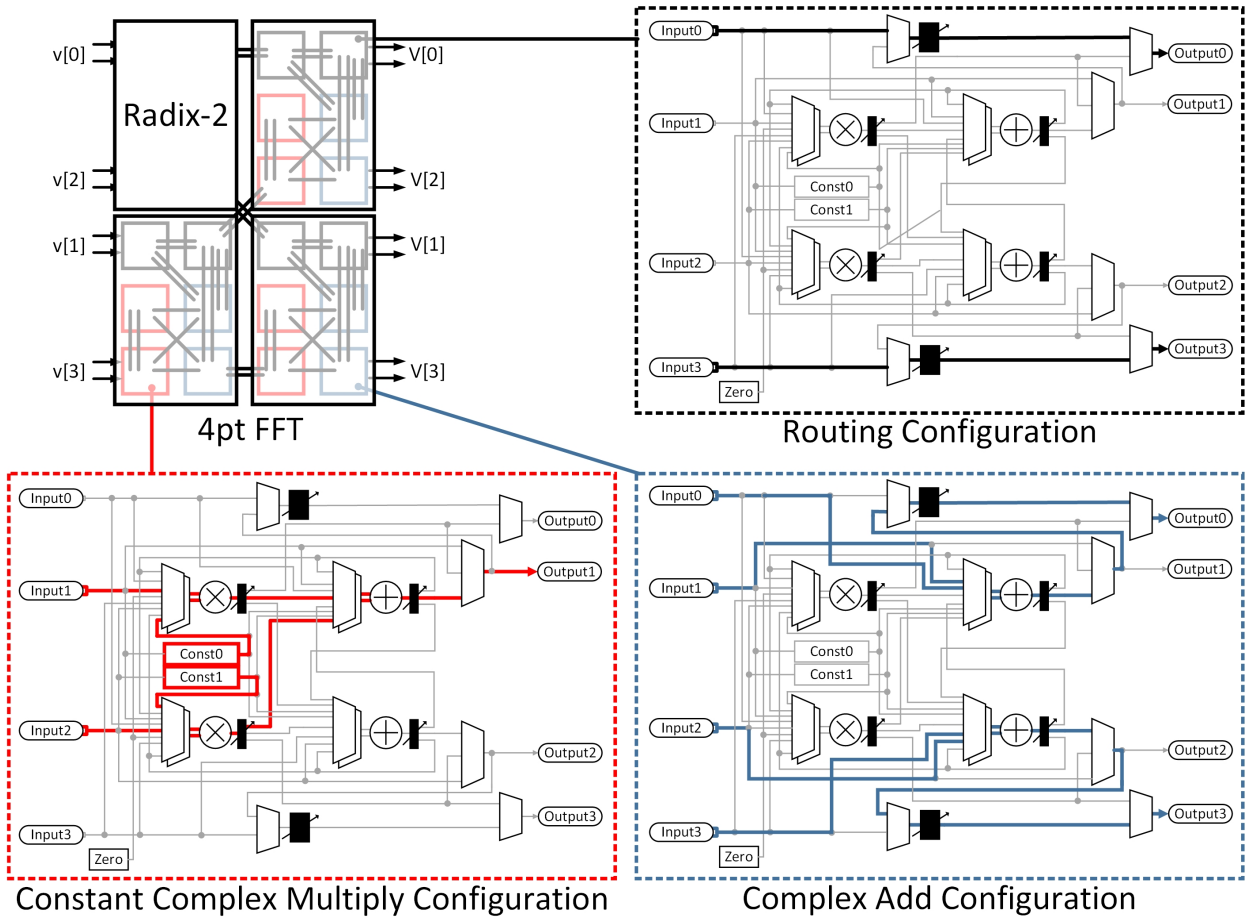


Figure 3.35: UDSP program mapping for 4pt FFT using radix-2. Insets show various different configurations used for radix-2. In the mapping many of the functional units inside the vertical stack are unused, while in some configurations the vertical stack is only being used for routing, thus resulting in a 42% overall mapping efficiency.

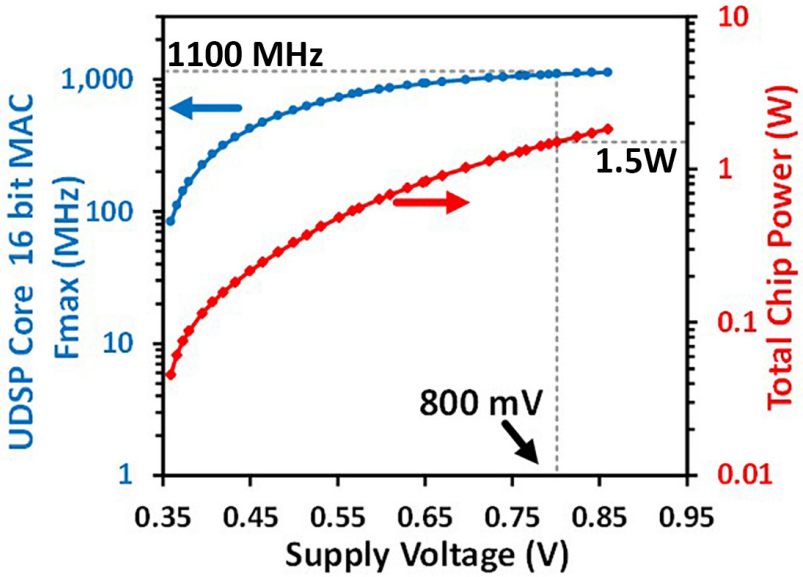


Figure 3.36: UDSP frequency and power scaling with supply voltage.

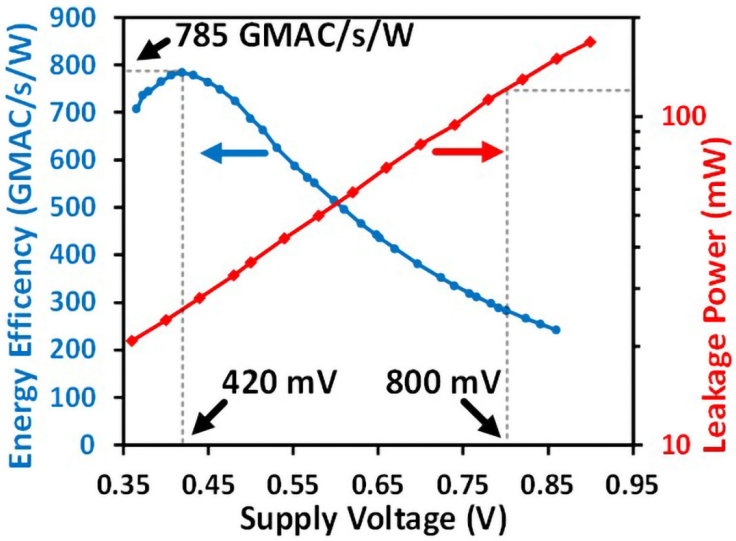


Figure 3.37: Energy efficiency of UDSP and leakage power scaling with supply voltage.

some of the bits might get flipped or some data-path inside the core might get corrupted leading to a non zero value at the output. These results were then measured and tabulated to generate the frequency, power scaling with supply voltage graph.

We mapped some commonly used program kernels and DSP operations for beamforming, FIR filter, real matrix multiply and FFT to measure algorithm specific performance as shown in Table 3.1.

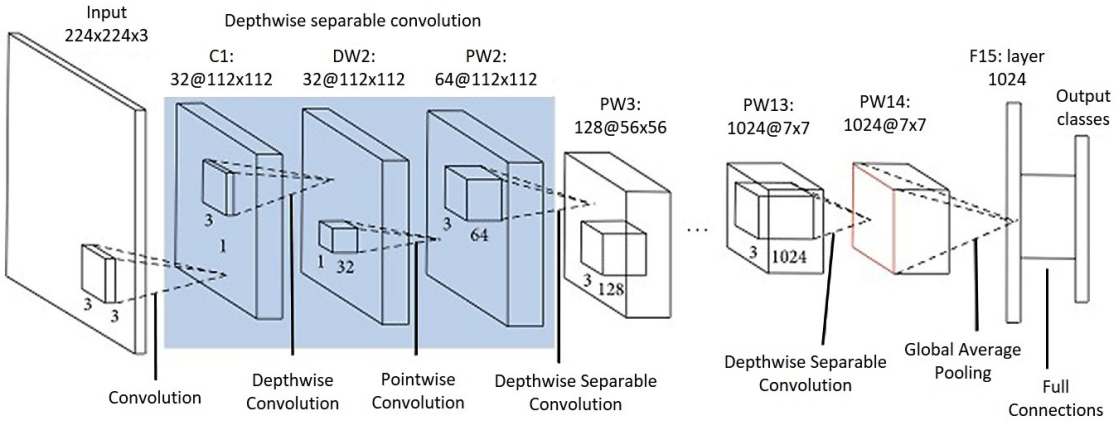
Table 3.1: UDSP throughput and energy measurements for various algorithms.

Operating Condition →	0.8V, 40°C			0.42V,40°C
↓ Algorithm	Throughput (GS/S)	$Energy_{total}(pJ)$	Cores Utilized	$Energy_{total}(pJ)$
16-tap FIR	1.1 (Real)	3.54 /Tap	8	1.29 /Tap
16-pt FFT	17.6 (Complex)	11.2 /Radix2	86	4.09 /Radix2
4x4 beam-forming	4.4 (Complex)	14.2 /Complex-MAC	34	5.15 /Complex-MAC
8x8 matrix multiply	8.8 (Real)	3.5 /MAC	32	1.27 /MAC

3.3.3 Application Results on UDSP

We test the application performance of UDSP by mapping MobileNet CNN [19] on to the array. We selected MobileNet as an application of choice as Neural Network inferencing is a very current and relevant topic. MobileNet is an object recognition Convolutional Neural Network for mobile and embedded vision applications. MobileNet is one the very few CNN algorithms which uses averaging pooling operation. Most of the CNN kernels require the use of non-linear pooling and activation techniques such as max pooling, mixed pooling and L_p pooling etc. As UDSP was primarily designed for DSP applications, such non-linear operations lie beyond the domain of original intended applications for UDSP. However, these non-linear pooling kernels can still be mapped onto the UDSP array using Taylor series or Maclaurin series or CORDIC approximations but the mapping efficiency of such algorithms

tends to be lower and thus the overall throughput of UDSP for such approximations can be low.



Filter Shape	3x3x3x32	3x3x32	1x1x32 x64	3x3x64	1x1x64 x128	...	3x3x1024	1x1x1024 x1024	1024x 1000
Stride	2	1	1	2	1		1	1	1

Figure 3.38: Layers and filter sizes of MobileNet CNN.

MobileNet forms an ideal candidate CNN for UDSP. MobileNet requires multiple kernels and layers of filters as shown in Figure 3.38, the dense and low-latency interconnect network of UDSP allows us to map multiple filters in the same CNN layer simultaneously onto the array and maximize the data reuse and computation while minimizing the IO bandwidth requirements as shown in Figure 3.39. Each layer of MobileNet application requires a separate programming step for UDSP, thus computation stages for multiple filters of the CNN are staggered with long times of UDSP reconfiguration. Although UDSP has a reconfiguration bandwidth of 1Gbps, we also simulate the program mapping and results for other higher reconfiguration bandwidths of the array at 20Gbps and 200Gbps. The results of MobileNet inferencing on UDSP are shown in Figure 3.40. We plot our results in the Figure for varying batch sizes and programming bandwidths. In the batch operation, the system collects multiple images that require CNN inferencing and passes them through the hardware array together. Using higher batch size, allows us to amortize the time required to program UDSP for different layers over multiple images, by essentially programming the kernel once

and passing multiple images through the hardware. As we can observe in the Figure 3.40, the throughput performance for UDSP increases with increasing batch sizes with the upper limit at 2,000 frames per second (fps), which is bound by the time required to perform the computation for inferencing per image. At point 1 as shown in the Figure 3.40, UDSP while operating with 1Gbps programming bandwidth can achieve 17fps inference rate for batch size of 1, this operating point would be ideal for mobile applications such as landmark recognition for navigation. The operating point 2 represents an inferencing throughput of 300fps with a batch size of 20, this result would be ideal for autonomous-driving vehicles as the vehicle can have multiple video feeds coming from multiple cameras, which would require a response time of $< 100ms$ for object detection and reaction. At operating point 3, we provide a simulated result for UDSP with 20Gbps programming bandwidth. The increase in programming bandwidth allows UDSP to switch between multiple kernels of the CNN much faster and provide better throughput result of 1,560fps at batch size of 20. We discuss the effects of increasing programming bandwidth further and in detail in the RTRA Chapter 5.

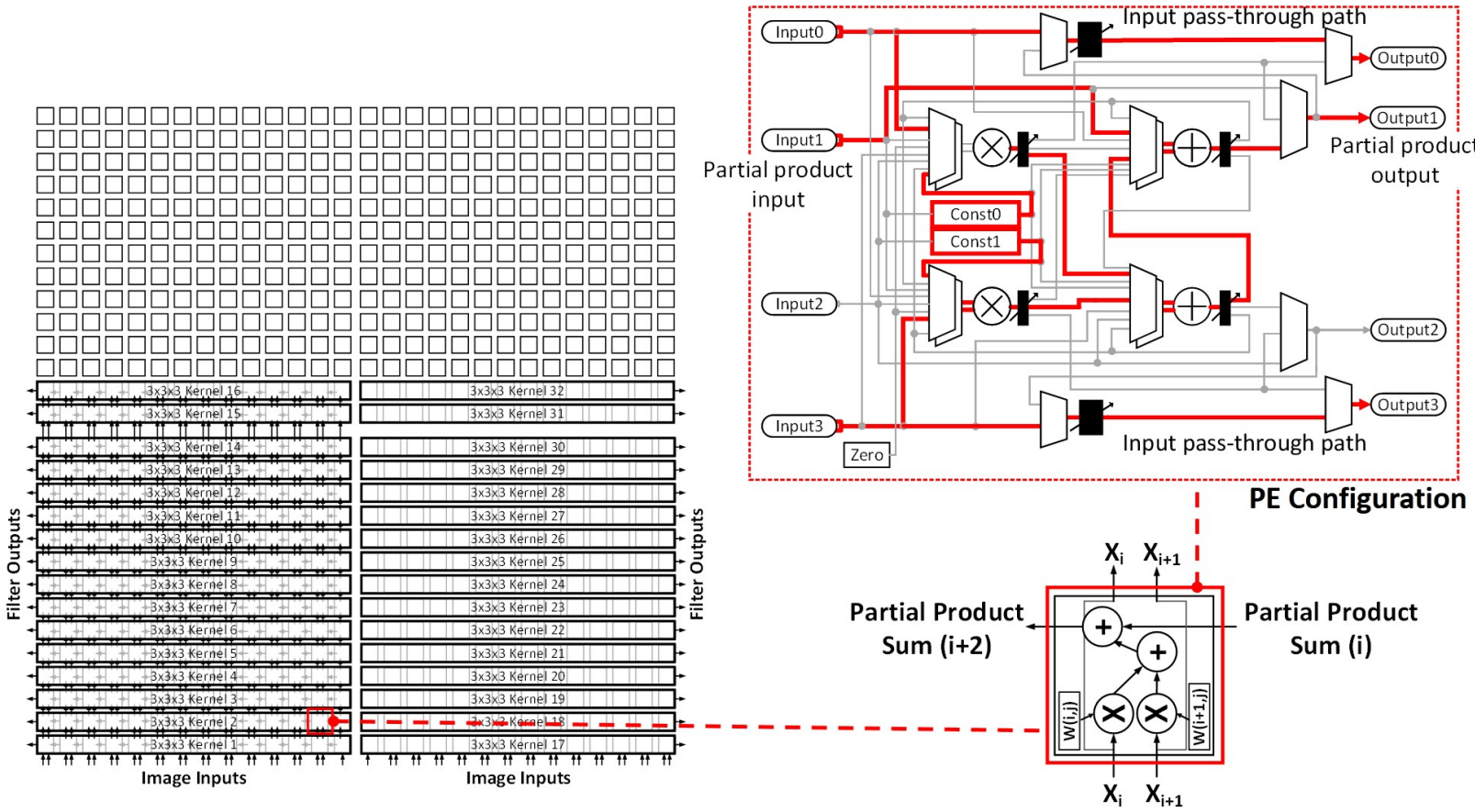


Figure 3.39: Multiple kernels from first layer of MobileNet CNN mapped spatially onto the UDSP array, and an inset showing the internal configuration of each Processing Element (PE).

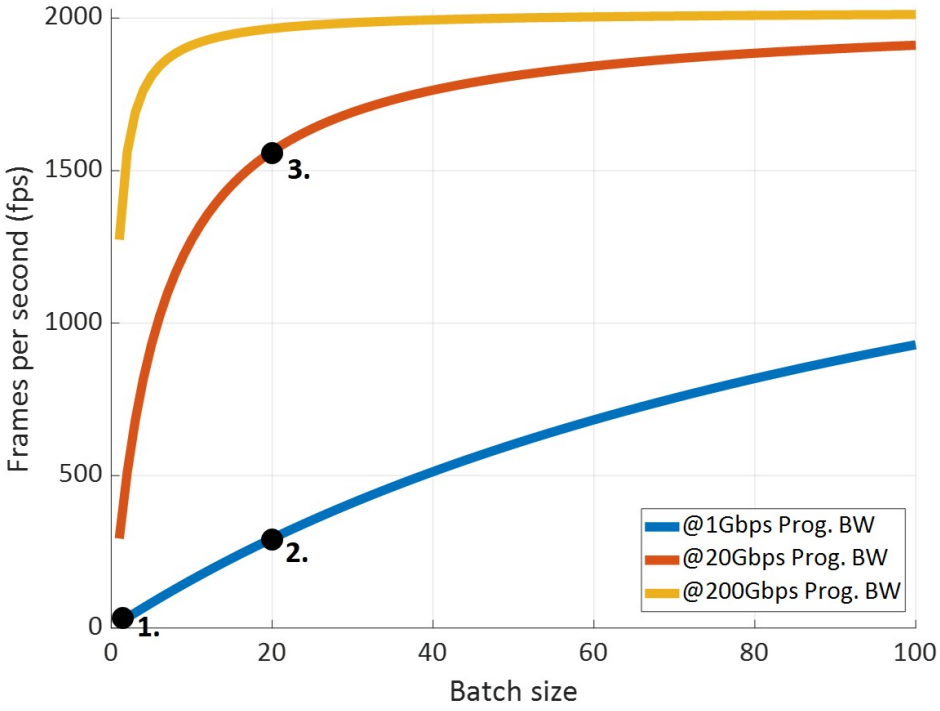


Figure 3.40: MobileNet throughput expressed in frames per second (fps) for varying programming bandwidths. Operating points 1, 2, and 3 represent a throughput of 17fps, 300fps and 1560fps for batch sizes of 1, 20 and 20 images respectively.

For lower batch sizes the energy consumption per image is dominated by the leakage energy. The array consumes leakage energy continuously, however during the programming of the array, the dynamic energy is negligible and leakage is the major contributing factors. So, as the batch size is increased, the leakage energy during the programming of the array is amortized over multiple images, hence the total energy consumption per image decreases. The power consumption of the array follows a different trend, while the energy consumption per image decreases with higher batch sizes, however the throughput and fps of the array increases, which increases the power consumption of the array. Leakage power consumption of UDSP 2×2 is roughly 100mW which is less than 2% of the maximum dynamic power i.e. 6W of the array. For lower batch sizes, as array spends most of the time in programming, so the leakage power lowers the average power of the array, however for higher batch sizes as the activity of the array increases, the dynamic power becomes a larger contributing factor

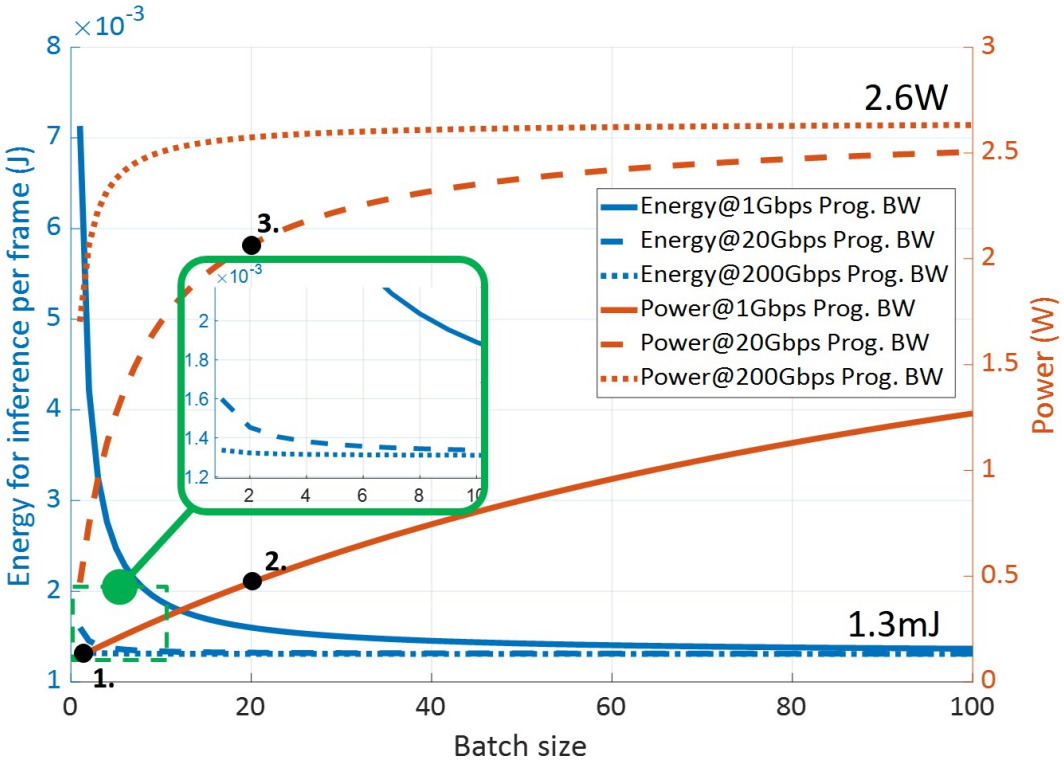


Figure 3.41: Energy and power consumption estimates for UDSP for varying programming bandwidths. Points 1, 2, and 3 represent power consumption of 121mW, 470mW, and 2W for batch sizes of 1, 20, and 20 images respectively.

to the power consumption of the array.

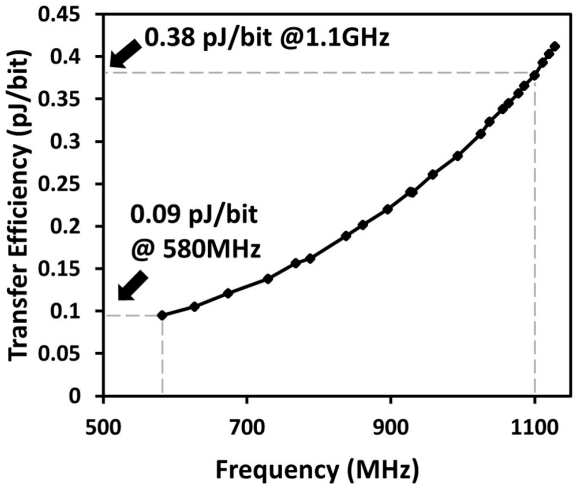


Figure 3.42: Bit transfer efficiency and frequency scaling.

3.4 SNR-10 Measurements

We tested the SNR-10 channel designed for TSMC 16nm UDSP assembled as 2×2. The SNR-10 channel has 64IOs and has a physical footprint of $8769\mu m^2$ resulting in an area IO density of $137\mu m^2$ per bit. The power measurements for SNR-10 channel were calculated by sending a toggling stream of $1 \rightarrow 0 \rightarrow 1$ across the UDSP dielet boundary using the SNR-10 channel at 1.1GHz clock rate with 0.8V operating voltage. The Si-IF interposer has 2 routing layers to carry data between UDSP dies across a wire reach of $350\mu m$ as shown in Figure 3.33. The channel requires 0.38pJ/bit at 1.1GHz, 0.8V which drops down to 0.9pJ/bit at 580MHz, 0.5V. The bit transfer efficiency and frequency scaling of SNR-10 channel is shown in Figure 3.42. Table 3.2 summarizes the metrics for SNR-10 channel and provides a comparison with some of the state-of-the-art implementations.

Table 3.2: Metrics for SNR-10 channel and comparison with state-of-the-art high density multi-chip packaging technologies.

Metric	SNR-10	AIB [28]	LIPINCON [27]	GRS [52]
Technology	16nm	16nm	7nm	16nm
Package Substrate	2-Layer Si-IF	4-Layer EMIB	15-Layer CoWoS	MCM /PCB
Bump Pitch (μm)	10	55	40	150
Wire Reach (μm)	350	3,000	500	80,000
Data Rate (Gbps/pin)	1.1	2	8	25
Voltage (v)	0.8	0.9	0.3	0.95
Energy Efficiency (pJ/bit)	0.38	0.83	.056	1.17
IO Area Density (μm^2/bit)	137	203	500	10,175
Peak Shoreline BW Density (Gbps/mm)	297	256	1,600	292
Layer BW Density (Gbps/mm/layer)	149	64	107	25
Delay (ns)	2.8	4	N/A	N/A

CHAPTER 4

Software Compiler for UDSP and RTRA

In this chapter we go over the design of software compiler for our UDSP and RTRA architecture. The compilation procedure for UDSP and RTRA are very similar. Compilation for RTRA differs from that of UDSP in the last few steps where the compiler adds some additional metadata information pertaining to the physical attributes of the compiled program to help the performance and capability of hardware scheduler and hardware compiler. We will go over the compilation procedure for UDSP architecture and then the modifications made to the compilation pipeline to accommodate RTRA.

4.1 Overview

UDSP hardware and software compiler were co-designed to achieve a trade-off between minimizing the hardware cost of UDSP while also simplifying the compilation process for the same. The simplifaction of compilation procedure or its programmability is a hard metric to quantify as different architectures can have very different methods of compilation. For example, compiling a program for general purpose processor like a CPU would take short time, since the arithmetic operations in the CPU are atomic, the architecture is highly flexible and the compilers are typically not constrained by program execution timings and performance. On the other hand, compiling a program for spatial architecture like FPGA takes a long time, as the process requires several iterations of mapping functionality, placement, routing and checking timing constraints. The non-deterministic nature of routing and varied timing between the nodes means that the compilation for FPGA may require large number of iterations; running into minutes worth of compile time for relatively simple programs.

UDSP belongs to this category of spatial architectures, and also requires an iterative compilation process. One of the goals while designing the UDSP architecture was to minimize the number of iterations required for clustering, mapping, routing and timing checks and hence simplify the compilation procedure. During the design of UDSP routing network, special considerations were taken to make the routing network deterministic and have a fixed timing latency between nodes. In addition, multiple programmable delays were added to the compute core, to achieve a low effort retiming in case of a routing failure. These special considerations, high routability and low cost retiming reduce the number of iterations required to achieve a successful program compilation.

Due to the high degree of reconfigurability offered by UDSP, and many architectural differences as compared to other prevalent architectures we designed our own custom compiler/mapper that we call Multicore Mapper. It takes the user defined software algorithms as data flow graphs and generate hardware programming bits necessary for configuring UDSP array. Figure 4.1 provides an overview of compiler's flow.

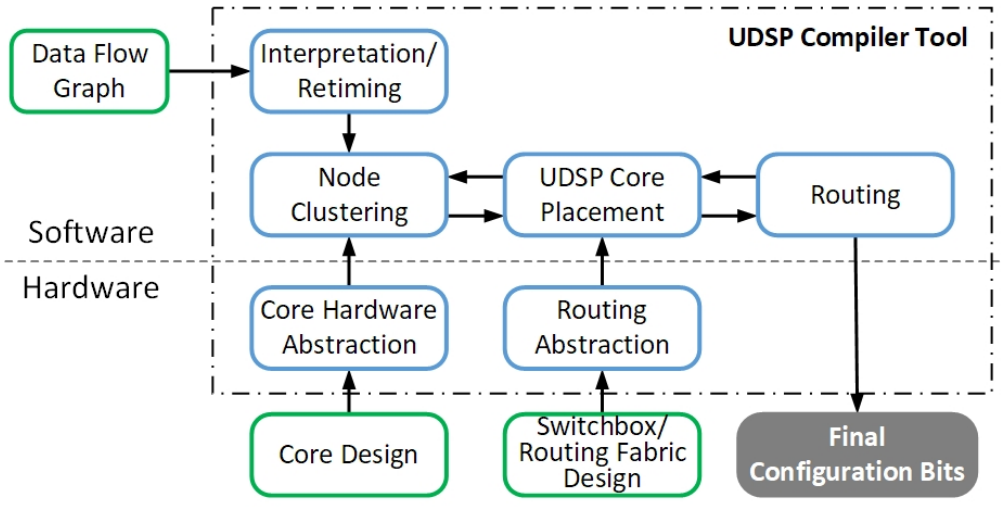


Figure 4.1: Compiler toolflow overview.

4.2 Compiler Inputs

We designed our compiler to be generalizable, so that future revisions of UDSP architecture can also be accommodated. To achieve this goal the hardware characteristics of UDSP, i.e. the connectivity matrix of the processing elements and the switchboxes are provided as an input to the compiler. This hardware configuration input can be stored inside the compiler program for the used hardware version. Thus, inputs given to the compiler can be divided into program specific inputs and hardware specific inputs.

4.2.1 Program Inputs

The compilation process starts with the data flow graph of the programs as an input. The data flow graph can be input as a MATLAB model and it should comprise of the functional units that are supported by the UDSP core architecture like multiplier, shifter, adder/subtractor, constant banks and delay elements. The delay elements present in the graph should satisfy the functional requirements of the program and are not required to meet the retiming effort, the compiler can automatically adjust for those retiming delays. In the current iteration of compiler, associativity and commutativity of arithmetic operations are partially supported, so if required, the user would have to manually modify the input data flow graph of the program to meet their hardware utilization goals. Some of the example input data flow graphs are shown in Figure 4.2 and Figure 4.3.

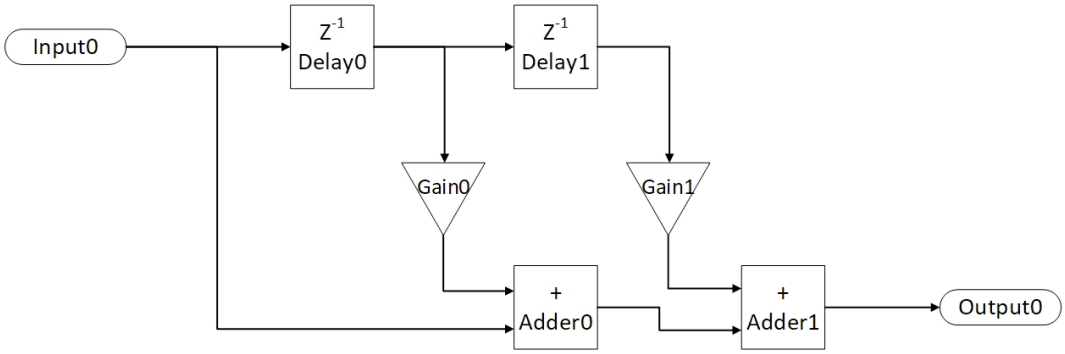


Figure 4.2: Dataflow graph for a 2-tap FIR filter.

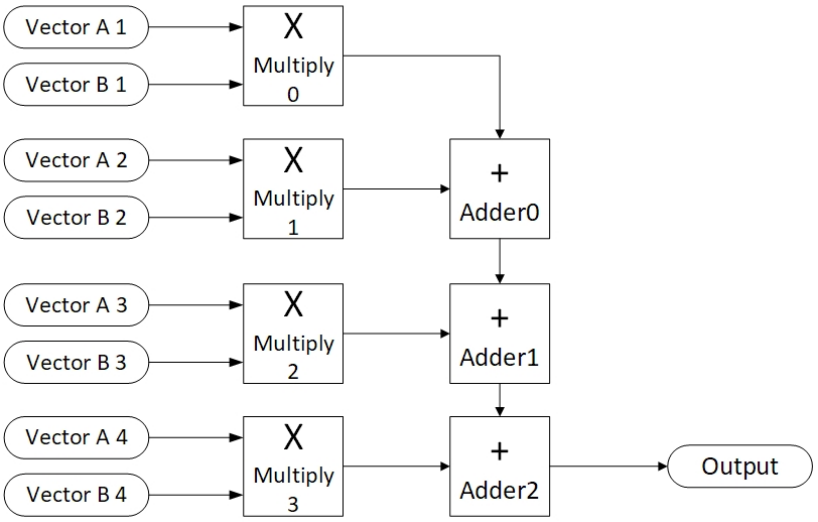


Figure 4.3: Data flow graph for a vector dot product operation using Multiply Accumulate (MAC) operation with an adder tree.

4.2.2 UDSP Hardware Abstraction Inputs

UDSP compiler maps the retimed and modified program data flow graph onto the functional units and routing of the UDSP architecture. To maintain the generality of the compiler over various revisions of the UDSP architecture, we provide the compiler with hardware parameters which it uses to generate mappings. The hardware parameters are divided into two abstract layers, i.e. the core and routing layer, routing layer is then further abstracted into its composing three switchbox layers. The description of the core is represented as a matrix which helps the compiler to understand various data paths and connections between the functional units inside the core and the routing layer description helps the compiler to route the data between the inputs/outputs of the compute core and the inputs/outputs of the vertical stack.

The core configuration represents the connectivity amongst its different sources and sinks as well the length of variable delay between the pair. For easier description the core configuration is further broken down into two configuration matrices, one connectivity matrix to describe connectivity for various elements within the core, and another delay matrix to

tabulate the number of programmable delays within each of the connections as described in connectivity matrix.

Table 4.1 shows the connectivity matrix of the core configuration. The entries in connectivity matrix describe the connections between the source, or signal generating nodes and the signal sink nodes. The entry in the i^{th} row and the j^{th} column represents the port of the sink node to which the source node is connected, if a particular sink node is not connected to the source node, then that is represented with a '0' in the connectivity matrix. As an example, the source Input 0 connects to MultShift0 at the port 2 of MultShift0 and the output of MultShift0, which is now the source connects to Adder0 at the port 1 of Adder0. The design philosophy behind coming up with this connectivity matrix is described in detail in the hardware description of UDSP in Chapter 3.

Table 4.2 shows the delay matrix of various elements within the core. The entries in the delay matrix represent a list of programmable delay entries that can be selected by the compiler between the source and the sink. As an example the compiler can select either a single delay or two clock cycle delay while traversing between $In0 \rightarrow MultShift0$. The programmable delays help ease the routing and placement pressure on the compiler in later steps of compilation and also eases the retiming effort, since small number of delays can be added or removed from the mapped program without requiring high effort and iterative re-placement and reclustering operations. Some entries have a longer length of variable delays, e.g. $In3 \rightarrow Out3$ has 1-8 programmable delays, compiler can use these data paths to configure the vertical stack as a hop path to connect two distant connected cores which could not be routed otherwise.

Table 4.1: Connectivity matrix for core configuration.

Sink → ↓ Source	MultShift0	MultShift1	Adder0	Adder1	Const0	Const1	Out0	Out1	Out2	Out3
In0	[2]	[2]	[1]	0	0	0	[1]	0	0	0
In1	[1]	[2]	0	[2]	[1]	0	0	[1]	0	0
In2	[2]	[1]	[2]	0	0	[1]	0	0	[1]	0
In3	[2]	[2]	0	[1]	0	0	0	0	0	[1]
MultShift0	0	0	[1]	[2]	0	0	[1]	[1]	0	0
MultShift1	0	0	[2]	[1]	0	0	0	0	[1]	[1]
Adder0	[1]	[1]	[1]	[2]	0	0	0	[1]	[1]	0
Adder1	[1]	[1]	[2]	[1]	0	0	0	[1]	[1]	0
Const0	[2]	0	[2]	[1]	0	0	0	0	0	0
Const1	0	[2]	[1]	[2]	0	0	0	0	0	0
Zero	[1]	[1]	0	0	0	0	0	0	0	0

Table 4.2: Delay matrix for core configuration.

Sink → ↓ Source	MultShift0	MultShift1	Adder0	Adder1	Const0	Const1	Out0	Out1	Out2	Out3
In0	[1 2]	[1 2]	[1 2]	-	-	-	[2-9]	-	-	-
In1	[1 2]	[1 2]	-	[1 2]	[1]	-	-	[2 3]	-	-
In2	[1 2]	[1 2]	[1 2]	-	-	[1]	-	-	[2 3]	-
In3	[1 2]	[1 2]	-	[1 2]	-	-	-	-	-	[2-9]
MultShift0	-	-	[1]	[1]	-	-	[1 2]	[1 2]	-	-
MultShift1	-	-	[1]	[1]	-	-	-	-	[1 2]	[1 2]
Adder0	[1 2]	[1 2]	[1]	[1 2]	-	-	-	[1 2]	[1 2]	-
Adder1	[1 2]	[1 2]	[1 2]	[1]	-	-	-	[1 2]	[1 2]	-
Const0	[0]	-	[0]	[0]	-	-	-	-	-	-
Const1	-	[0]	[0]	[0]	-	-	-	-	-	-
Zero	[0]	[0]	-	-	-	-	-	-	-	-

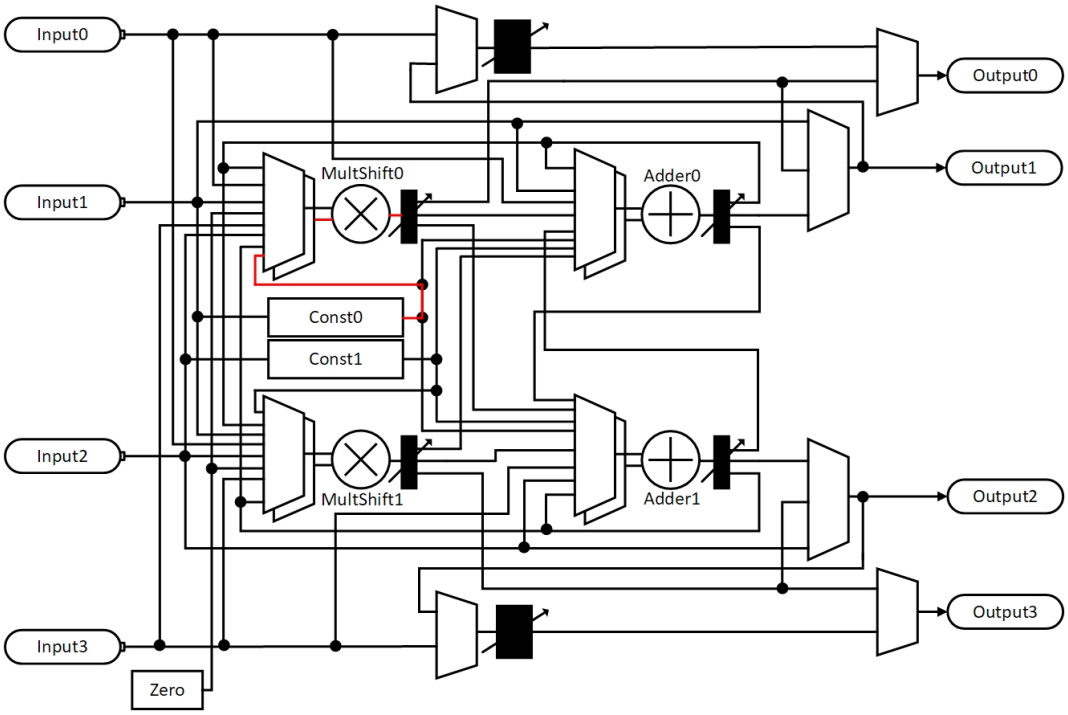


Figure 4.4: Connectivity inside UDSP core, delay elements are not represented in this figure. The highlighted path represents one of the longest delay paths.

The delays and connections are very carefully chosen so that for any configuration of the compute core, 1GHz operation is always guaranteed. One of the longest non-registered path in the delay matrix would be the path initiating at the $Const0 \rightarrow MultShift0 \rightarrow Adder1$, as shown in Figure 4.4. As can be observed in the delay matrix, Table 4.2, the path is registered at its starting point which is the output of Const0 or the input of MultShift0 and at its destination point which is the input of Adder1 or output of MultShift0. All the various combinations of paths from various sinks and sources were very carefully tested to satisfy the 1GHz timing, and ensure successful operation of all the configurations at that timing constraint.

4.3 Interpretation

The compiler processes the input program data flow graphs into an intermediate representation which the compiler uses to generate the final binary program output. The intermediate representation is a transformation of the input flow graph into multiple scheduled levels, where each level represents the rank of the node. The ranks are assigned in a way so that each node with its outputs connected to the inputs of the current node has a rank lower than the current node, an example for rank assignment and levelling is shown in Figure 4.5.

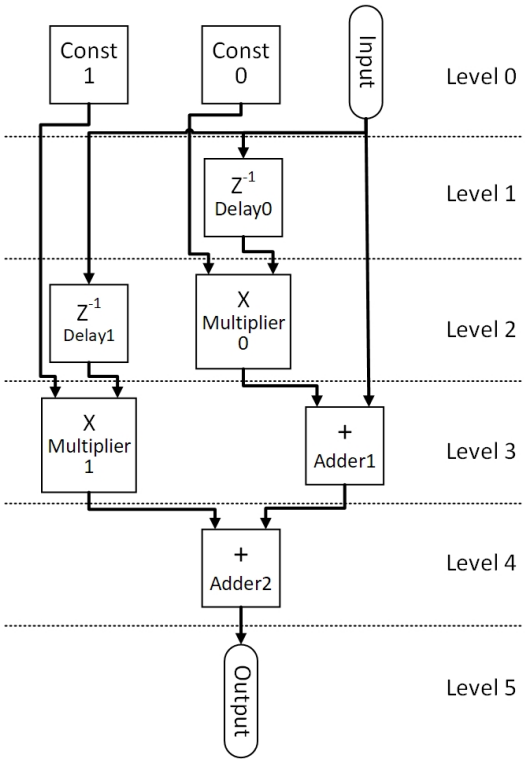


Figure 4.5: Levelled intermediate representation DFG of FIR filter from Figure 4.2.

The process of level generation and intermediate representation has to account for all different types of input programs. So, we broadly classify the input program data flow graphs into two main categories for the purposes of understanding the intermediate representation of our compiler. First, a feed-forward only data flow graph, in such a graph the connections between nodes do not form a loop/cycle, for example, FIR filter as shown in Figure 4.2, and vector dot product operation using MAC with an adder tree as shown in Figure 4.3.

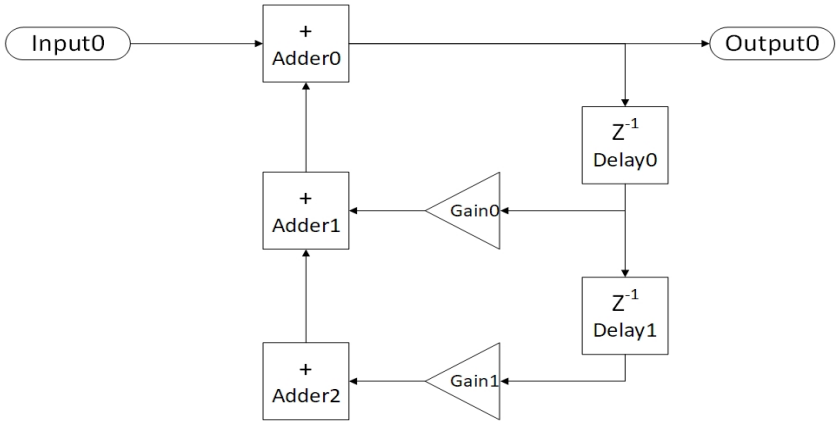


Figure 4.6: DFG for a 2-tap IIR filter.

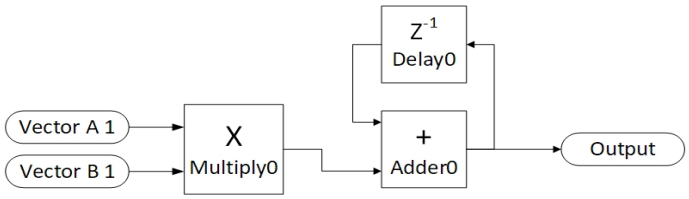


Figure 4.7: DFG for a multiply accumulator operation on vector inputs.

The second category of graphs have a feed-back path in the graph, in such programs the connections between its various nodes form at least one loop or a cycle, an example of this category would be IIR filter, as shown in Figure 4.6 and temporal vector dot product operation using multiplication and an accumulator, as shown in Figure 4.7.

The levels can be thought of as the flow of computation and dependency of each computation on its previous steps. The process of levelization for programs belonging to the first category of feedforward only data flow graphs is quite trivial, however the second category of graphs needs special handling. If a loop is detected in an input data flow graph, i.e. if a node is repeated inside the data flow graph, the graph needs to be broken down at an appropriate datapath to make sure the nodes are properly assigned ranks, and the nodes with loops are assigned a preferential status to be mapped separately. Once the nodes are ranked, the tool then starts the retiming process.

4.3.1 Retiming

Retiming process adds delays to the data flow graphs at appropriate locations to make the modified DFG mappable to the UDSP core architecture. The core connectivity can be observed in the Table 4.1 and Figure 4.4, there exist delay elements at the input and output of multiplier/shifter and adder/subtractor, additionally there exists no data path between the processing elements with higher than 3 delay elements. Based on these constraints the compiler retimes the input DFG, it adds delays through the input of output of the DFG and pushes them to appropriate locations as well as retimes the data paths with higher than 3 delay elements. As an example the DFG of FIR filter from Figure 4.5 after retiming looks like Figure 4.8. The special nodes belonging to the looped DFG of second category are left untouched in this operation are passed along as is to the next compiler operation. Once the delay constraints are met the tool starts the clustering operation.

4.3.2 Clustering

In the clustering operation, the compiler groups together the arithmetic operations so that the grouped operations can be mapped together into a single UDSP core. The clustering algorithm uses a greedy algorithm, such that it tries to map as many operations into a single cluster, and starts a new cluster when it fails, the operation which gets added into a cluster is randomized in case multiple operations are available to be added into a cluster. Clustering starts at the input nodes, and adds operations into the initial nodes, initiating a new cluster as needed based on the algorithm. The initial and final steps of clustering operation on the retimed DFG of FIR filter are as shown in Figure 4.9.

4.4 Placement

The clustering operation generates clusters of programs which can be mapped into a single UDSP core. In this placement operation, we place those clusters onto the array of UDSP cores such that routing distance between the cores is minimized and satisfies to the routing

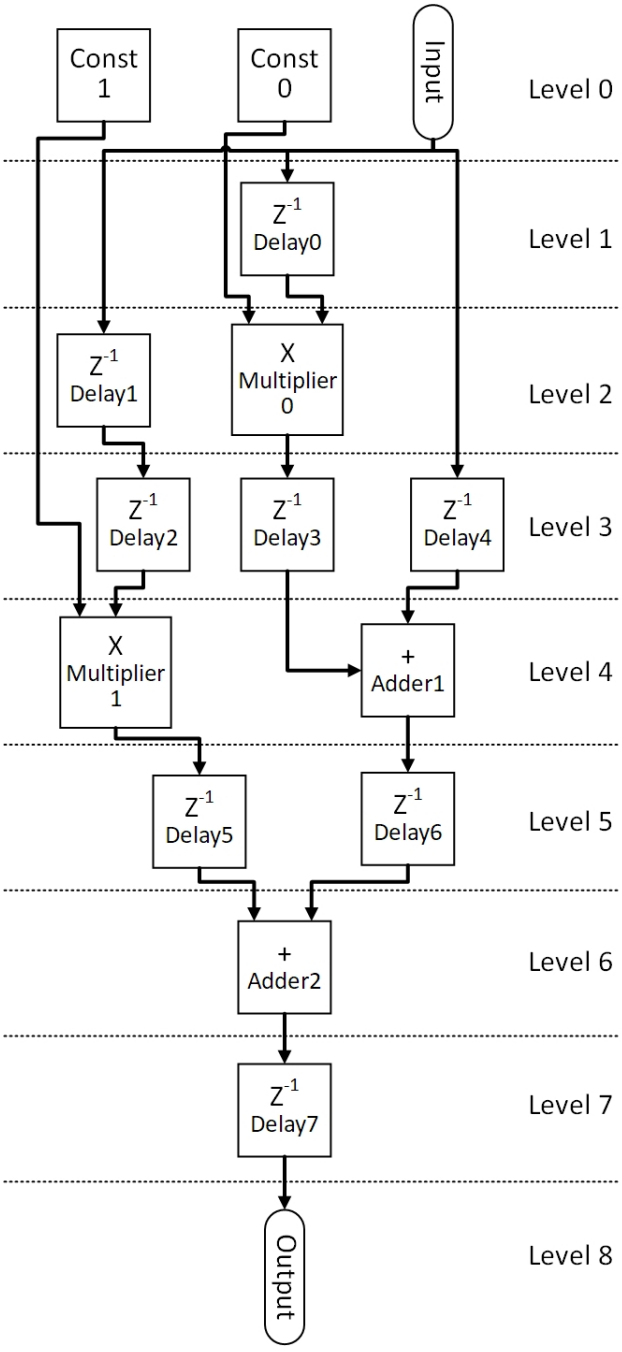


Figure 4.8: Retimed DFG of the levelled FIR filter.

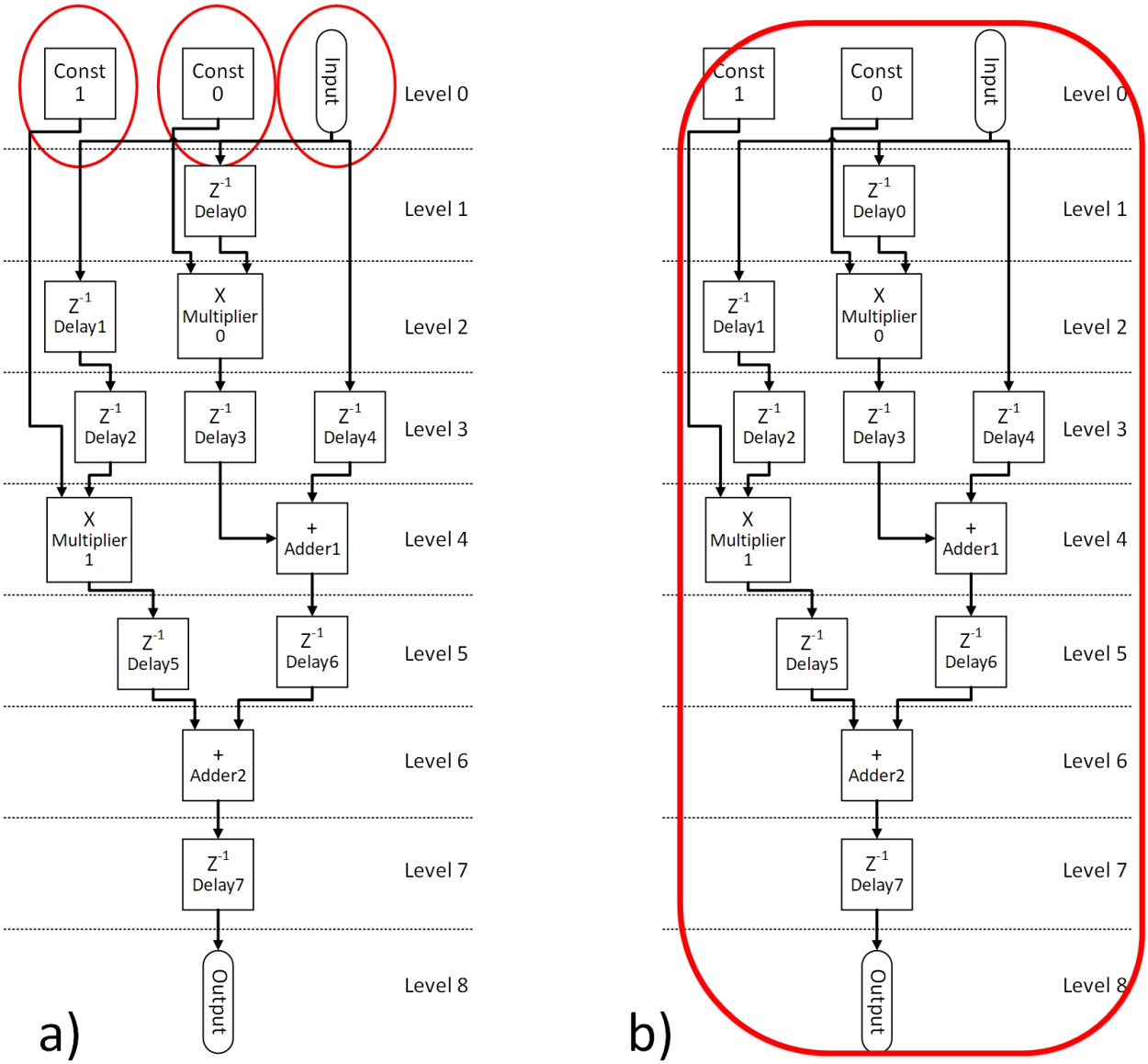


Figure 4.9: a) The initialization of clustering operation. b) The final resulting output of clustering operation for retimed DFG of FIR filter.

constraints as dictated by the interconnect network.

The problem of minimizing the routing distance that we are trying to solve is very similar to that of an FPGA compiler but FPGA architectures have flexibility in timing constraints. An FPGA compiler optimizes for the minimum routing distance, and can output the resulting setup time for the longest path as its operating frequency/clock period. We subject our architecture to a 1GHz clock frequency or 1ns delay timing constraint, and upto 1 hop distance 2 routing as supported by the three layers of switchboxes. From our initial study of algorithms belonging to the Digital Signal Processing (DSP) domain, the likelihood of having a route of distance greater than 2 is quite low but its still possible. Such routes are handled by iterative placements, retiming and revised clustering operations. In retiming approach, the programmable delay paths inside the core can be used to delay the datapaths, to accommodate an additional hop in the violating critical path.

The method chosen to reduce the wiring routing distance between the core is analogous to simulated annealing. The clusters are randomly placed initially over the UDSP cores and the algorithm tries to minimize the cost function as described by the Equation 4.1. We added the degree p of the wiring cost function to vary the penalty for longer connections, a connection of length 2 would be penalized highly with $p = 4$, as compared to $p = 1$. The variablity of p was added to discourage the placement of cores with large connection lengths which are unsupported by the hardware. The simulated annealing algorithm works by randomly selecting the two cores and swaps their placement if it reduces the wiring cost function. A probability function was added to the annealing algorithm to randomly accept some core swap operations which might increase the cost function, this effectively helps the placement operation get out of local minimas in the optimizing operation. The addition of probability acceptance would allow the simulated annealing operation to reach global minimas given enough time, but the compiler can be designed to quit out of the placement operation if the values seem to converge, and not improve upon successive iterations. An example of

simulated annealing operation is shown in Figure 4.10.

$$Wiring\ Cost = \sum_{n=1}^{N_{wires}} \left| n_{xlen}^2 + n_{ylen}^2 \right|^p \tag{4.1}$$

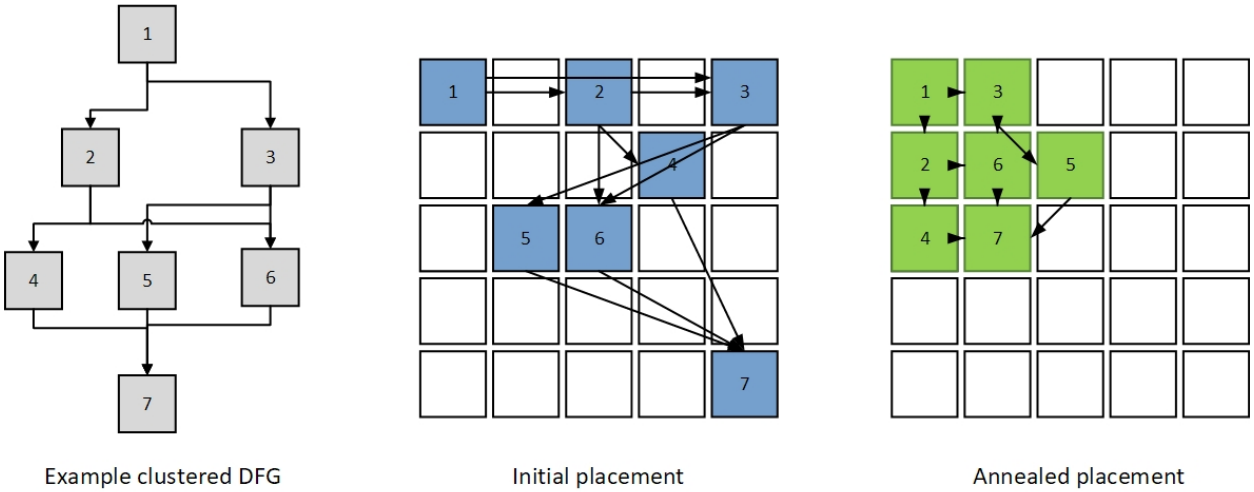


Figure 4.10: Various steps in the placement operation, starting from the cluster DFG to initial placement and then final simulated annealed placement.

4.5 Routing

Routing process configures the switchboxes inside the vertical stack for the appropriate network configuration. The procedure of routing is greatly simplified in our UDSP architecture since the network guarantees 1GHz operation and 1ns delay network, so the iterative process of optimizing routing and timing delay of the network is eliminated. The output of the placement operation takes care of routing distances, and the paths which exceed the distance 2 communications are handled either using retiming or revised clustering. So, the task of routing is greatly simplified as the violating paths and critical paths have largely been taken care of by this stage. Any remaining violating paths are handled using a registered layer 4 network, which allows for long distance communications and thus retiming the critical data paths might be required at the end.

As mentioned in the hardware description of UDSP in Chapter 3, the switchbox routing

layers almost guarantee the routability of any algorithm as shown in Figure 4.11.

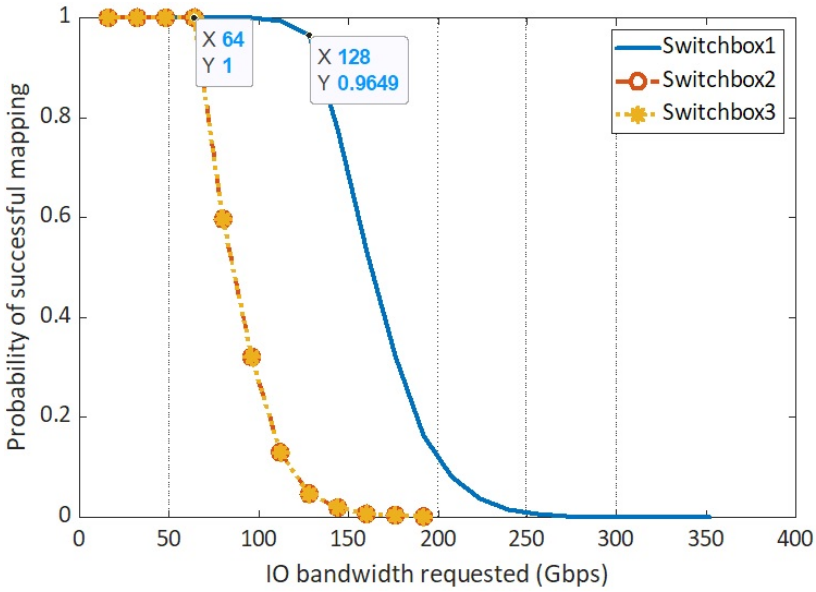


Figure 4.11: Switchbox layer 1 in each of the vertical stacks can support a 128Gbps routing connection request with a probability of 96.49%, while the switchboxes in layers 2 and 3 can support 100% of 64Gbps routing connection requests. The bandwidth requirements of 128Gbps and 64Gbps are driven by domain-specific hardware requirements.

The routing tool extracts the input/output requirements from a single cluster mapped to a core. The extracted IO data is then translated as input-output pairs which the router can map easily. The tool performs a depth first search on each input-output pair iteratively, until a successful routing is achieved. If the routing fails then the compiler repeats the placement procedure and/or revised clustering operation. We go over the switchbox configuration in detail in the next section.

4.6 Switchbox Compilation

The routing step from the previous section, generates the routing requirements of the program, and provides the input-output mappings required from every switchbox. The generated mapping for one switchbox is a list of input-output pairs that the switchbox needs to

connect. For example, if the routing requires the switchbox layer 1 at array location (4,5) the input 1 needs to connect to output 3, input 2 to output 4 and input 3 to output 7, the IO mapping list for the switchbox would be [(1,3),(2,4),(3,7)]. The switchbox compiler uses the mapping list and selects a configuration for the switchbox that can achieve that connectivity. We devised to approaches for the switchbox compiler to achieve that, 1) A randomized Depth First Search (DFS) method or 2) Breadth First Search (BFS) method. While the randomized-DFS method is faster at reaching the outcome for large switchboxes, there is a possibility it may not find a solution for the desired switchbox configuration even though the switchbox hardware permits it. On the other hand, BFS approach is slower in finding the desired solution, however the BFS approach is guaranteed to find a solution if the switchbox hardware permits the configuration.

4.6.1 Random Depth-First Search

The randomized DFS search process for switchbox compilation can be broken down in to following steps:

1. Randomize the order of IO mappings in the IO mapping list.
2. Randomly assign a token mux from the middle layer to the first IO in the mapping list based on the switchbox connectivity matrix Figure 3.24.
3. Randomly assign a token mux from the middle layer to the next IO in the mapping list based on the switchbox connectivity matrix but without the already selected token mux in the previous step.
4. Repeat the step 3 for the next listing in the IO mapping list until finished and thus the compiler achieves a successful IO mapping.
5. If the IO mapping list is not finished, however the selected IO in the list cannot be mapped onto the switchbox, then the compilation process is started again from step 1.

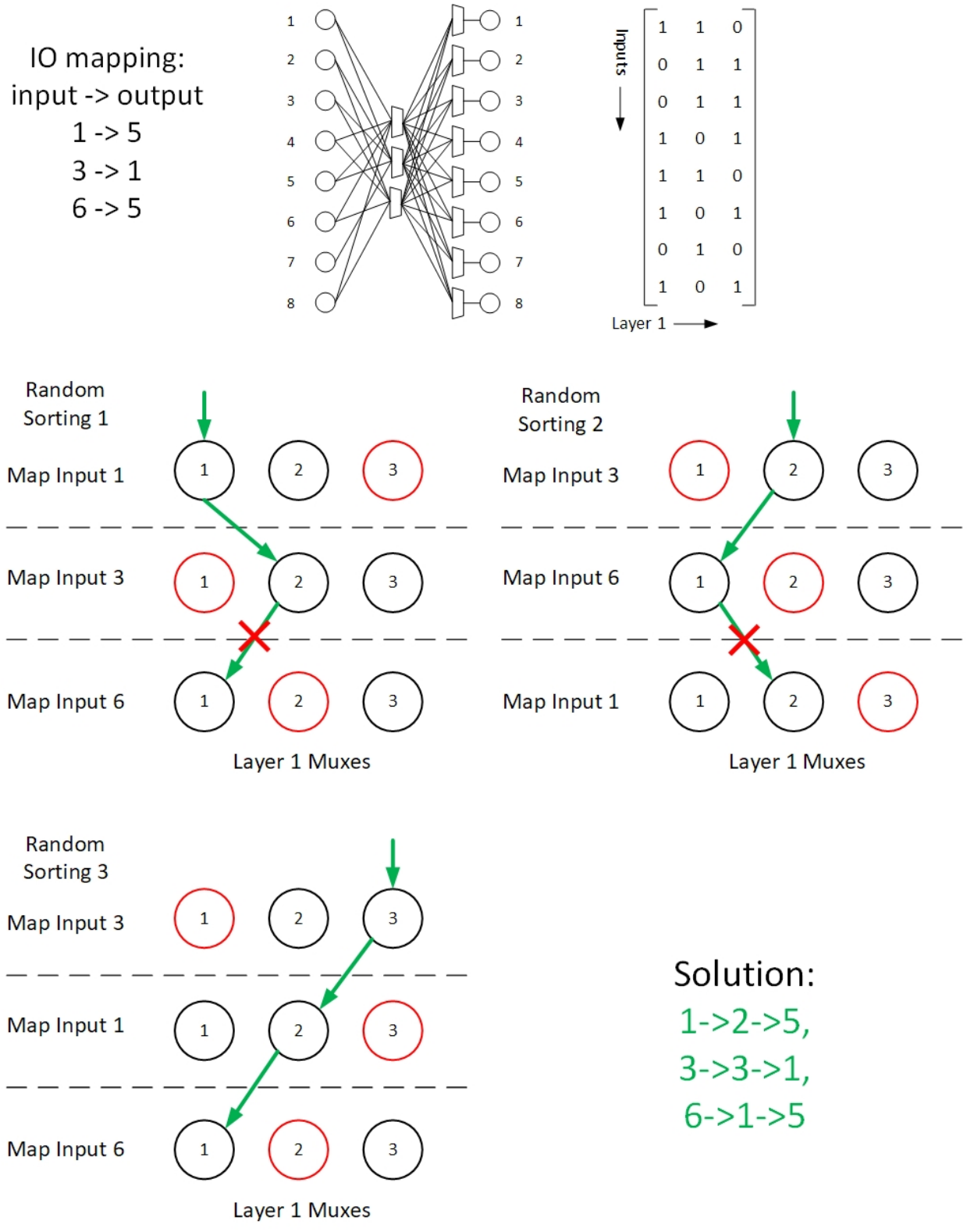


Figure 4.12: Random-DFS approach to switchbox compilations.

The randomized DFS search, repeats the steps 1-5 a fixed number of iteration, we chose 100 for our compiler. If the compiler is not able to achieve a successful mapping within the fixed number of iterations then we declare the selected routing cannot be mapped with the switchbox, and requires rerouting, retiming, reclustering or any other optimization technique available to the compiler. An example random-DFS flow for a 8-3-8 input-token-output switchbox configuration is shown in Figure 4.12.

4.6.2 Breadth-First Search

The BFS is compute and memory intensive approach to switchbox compiler operation however is guaranteed to return a solution if it exists. The BFS approach is not a randomized approach so it considers and tries all possible switchbox token layer combinations to achieve the required IO mapping. The procedure for BFS is very simple:

1. Start multiple directed graphs where the initial nodes are all possible token layer muxes which can map the first input in the IO mapping list based on the switchbox connectivity matrix Figure 3.24.

2. Propagate each of the directed graphs, in each of the graphs assign multiple nodes which can map the next input in the IO mapping list without the tokens already selected in the respective directed graphs. If for any of the directed graphs, the next input cannot be mapped to any of the muxes in the token layer, then graph is taken out of consideration and the process is continued for the remaining IO listings with the remaining directed graphs.

Since the directed graph technique exhaustively checks each possible combination of token muxes and switchbox configurations, it can guarantee a solution if it exists. An example BFS compilation flow for a 8-3-8 input-token-output switchbox configuration is shown in Figure 4.13. The dotted line and bold solid line show two possible solution for the requested IO mapping. The input 1 is allocated token 1 in directed graph-1 and token 2 in directed graph-2. From here, the input 3 can now be allocated either token 2 or 3 in directed graph-1,

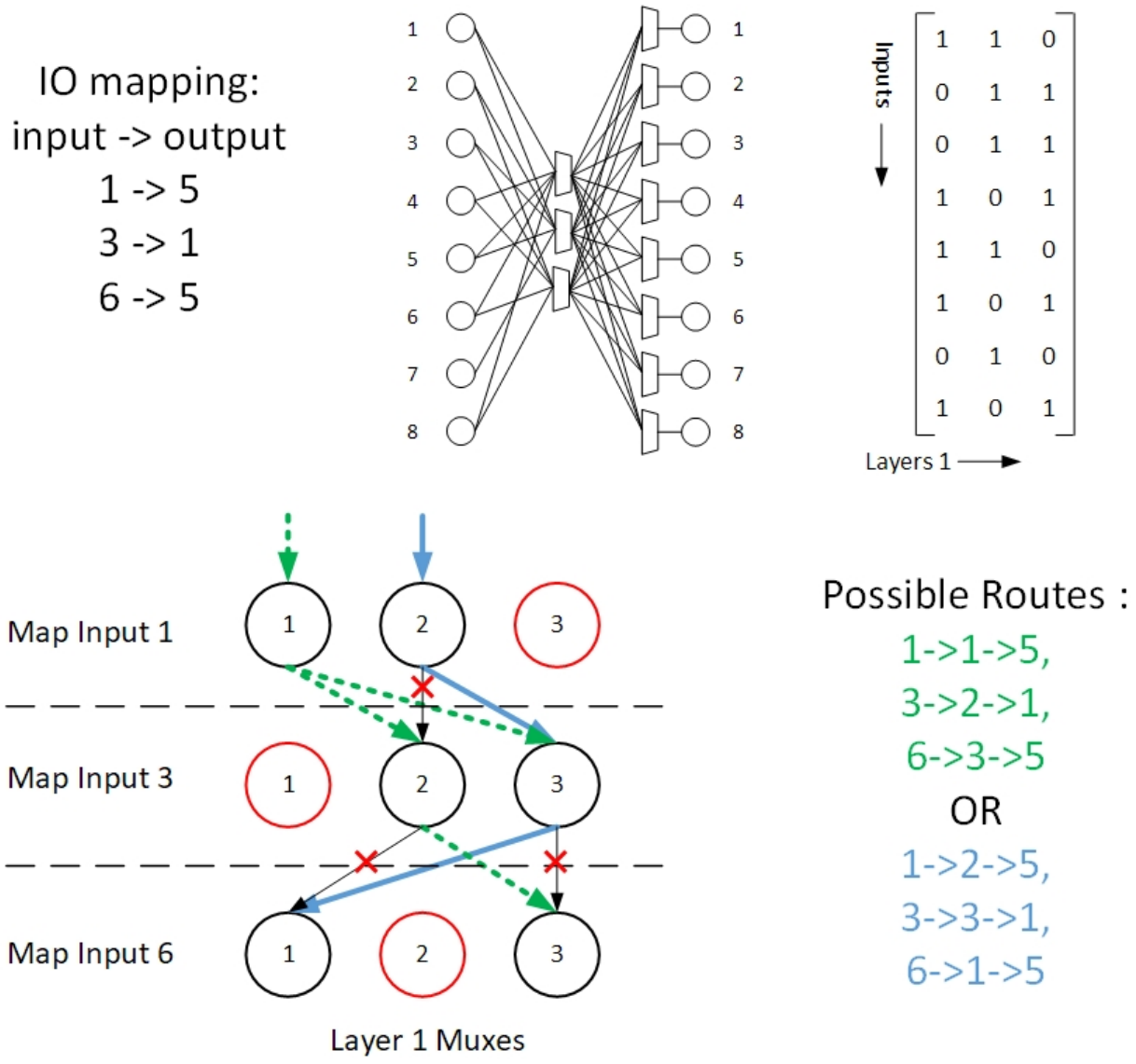


Figure 4.13: BFS approach to switchbox compilations.

thus splitting the graph into directed graph-1,1 and graph-1,2, or the in the directed graph-2, input 3 can be assigned token 3, thus graph-2 stays graph-2. Next, the last IO listing, input 6 can be mapped using token 3 in the directed graph-1,1, however no such possible token mapping exists for input 6 in the graph-1,2, thus it is removed from consideration, while the input 6 can be mapped using token 1 in the directed graph-2. Hence, there are two possible solutions for the IO mapping requirement, that is directed graph-1,1 ($1 \rightarrow 1 \rightarrow 5, 3 \rightarrow 2 \rightarrow 1, 6 \rightarrow 3 \rightarrow 5$) or directed graph-2 ($1 \rightarrow 2 \rightarrow 5, 3 \rightarrow 3 \rightarrow 1, 6 \rightarrow 1 \rightarrow 5$).

Both the solutions are equally fast when the sparsity of connections is increased. If we have more tokens in the middle layer than the probability of routing conflict when using either of the two techniques random-DFS or BFS is decreased, thus leading to similar complexity and time to solutions. We use this approach to design an optimal switchbox solver for RTRA's IO network, which is discussed in the later RTRA Chapter 5.

Upon successful compilation process, the UDSP compiler generates configuration bits for UDSP. These configuration bits are structured into frames, each of those frames is applicable to one vertical stack location. The frame is subdivided into the location information of the respective vertical stack, the switchbox1 configuration bits, switchbox2 configuration bits, switchbox3 configuration bits and the processing element configuration as shown in Figure 4.14.

4.7 RTRA

In this section we look into the compiler design for Runtime Reconfigurable Array (RTRA) and its virtualized program execution flow. The architecture of RTRA supports spatially co-mapped algorithms which can be temporally multiplexed using runtime reconfigurability and scheduling. The spatial mapping inside a single program are handled by the UDSP compiler, it translates the input DFG into the configuration bits that can be directly sent to the UDSP array. We modify the UDSP compiler for RTRA to enable the features that allow for runtime relocation of program resources onto available resources on the RTRA array. The runtime

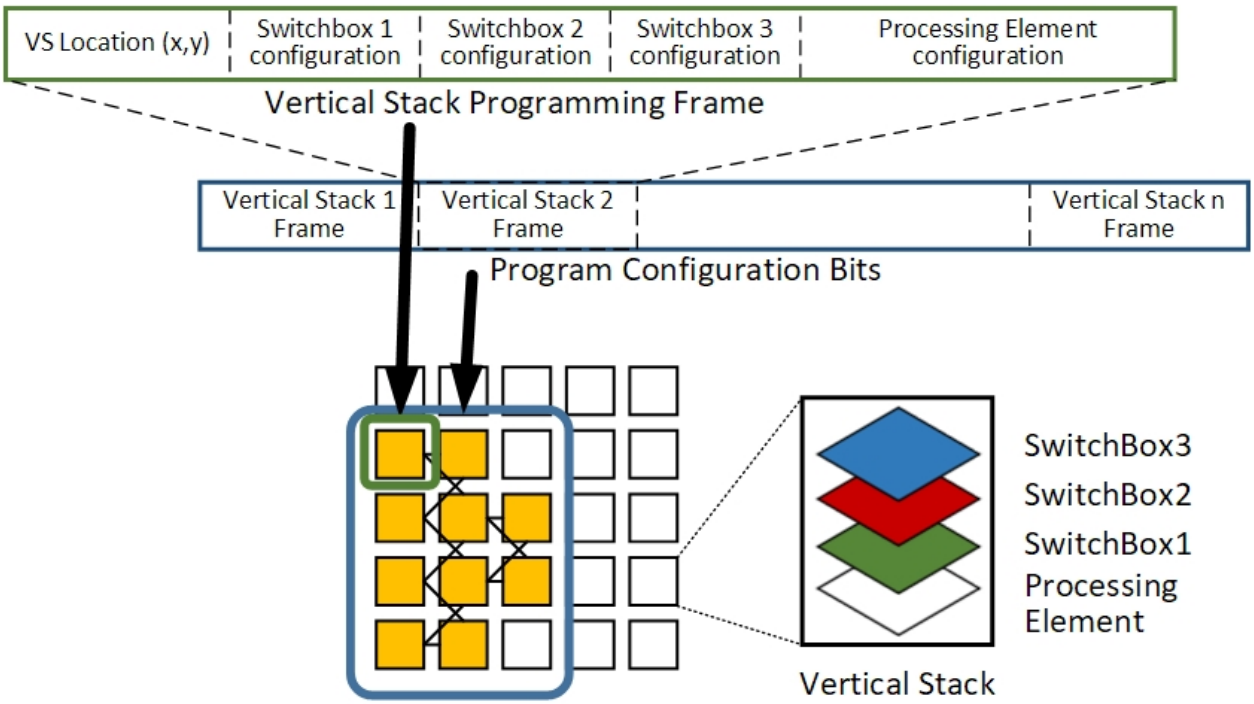


Figure 4.14: The structure of configuration bits of a UDSP program, and its internal vertical stack frames.

program relocation is facilitated by two new features, which are multi-size compilation and multi-step compilation.

4.7.1 Multi-Size Compilation

Software compiler usually generate a single stream of configuration bits or a single compiler program which can be executed on the system. The size of the program is decided by the programmer, and the compiler translates the requirements input by the user/programmer into a program binary. In multi-size compile paradigm, we seek to optimize the physical footprint of program on the array relative to the amount of computations required by the program, while also ensuring maximum utilization of the array in the presence of multiple program requests and high programming pressure.

In the spatially multiplexed RTRA, multiple programs could be occupying and actively executing on the array, so generating a single size compiled binary at the software compiler stage might mean that the requested program size may not be available on the array at runtime. We solve this problem by generating multiple compiled binaries of the program of multiple physical footprint sizes on the array. Therefore, at the time of mapping an incoming program the different program sizes would be available to the scheduler and it can then arbitrate between them to select the most suitable size possible for the available resources on the array. In this approach we basically trade-off the memory storage required to store the multiple sized binaries of the program with the waiting time for execution and thus the time required to finish executing the program. The generation of multiple sizes of the program binary by the compiler falls under two main categories: 1) when the DFG of the input program is large relative to the array size, 2) when the DFG of the input program is small relative to the array size. In the first scenario when the DFG of the input program is large relative to the array size, the compiler breaks down the larger DFG into smaller set of programs as shown in Figure 4.15 and Figure 4.17. The cut set of the original larger DFG can be decided based on smallest bisection bandwidth between the smaller graphs or based on control decision points. The Figure 4.15 shows an example where the cutset is decided

based on bisection bandwidth, the Figure 4.16 represents an example program of blind signal classification [56] where the broader algorithm flow graph is cut based on decision and control decision points as well as bisection bandwidth. For the second scenario, where the DFG of input program is small relative to the size of the array, that is the program could fit into a very small number of array resources, the compiler uses various concurrency-based techniques to increase its array use and generate multiple sizes of the program. The compiler performs loop unrolling, scalar expansion, subgraph matching techniques to expand and restructure the data flow graph, these techniques follow the description in Dyser [17].

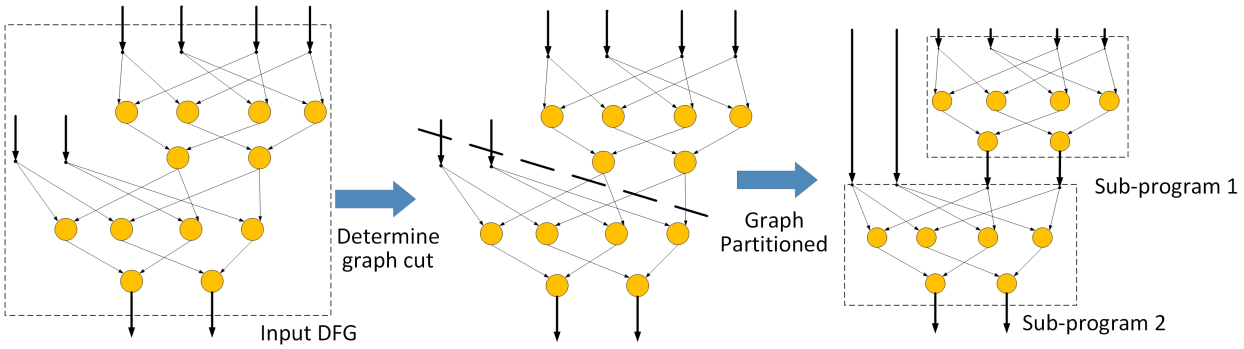


Figure 4.15: Example of large DFG program being broken into sub-programs with a graph cut based on minimum bisection bandwidth.

In the first program DFG size scenario, the graph cut effectively increases the execution time of the program by executing the program on the array in two distinct subgraphs, whereas in the second program DFG scenario, the compiler effectively decreases the execution time of the program by allocating more resources than requested by the user/programmer, by executing the program effectively in parallel. Both of these methods help in optimal utilization of resources of the array while balancing the execution time of the program, however the compiler has to calculate limit to the multi-size compile. Some of the "embarassingly" parallel algorithms belonging to the second program category can achieve an execution time speed up till all of its computations are performed in parallel, so we need to study the optimal sizing of the program by understanding the benefits in reduction of execution time versus the number of resources allocated to that program.

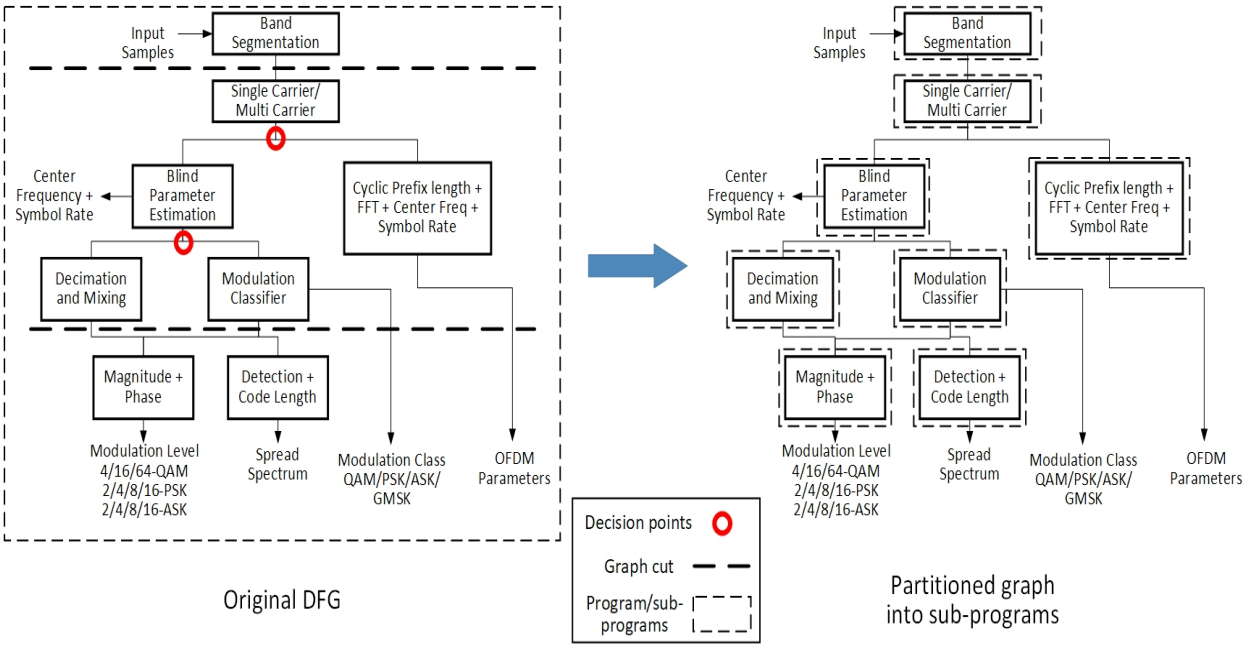


Figure 4.16: Example of large DFG program for blind signal classification [56] broken into subgraphs with graph cuts based on decision points or control flow points and minimum bisection bandwidth.

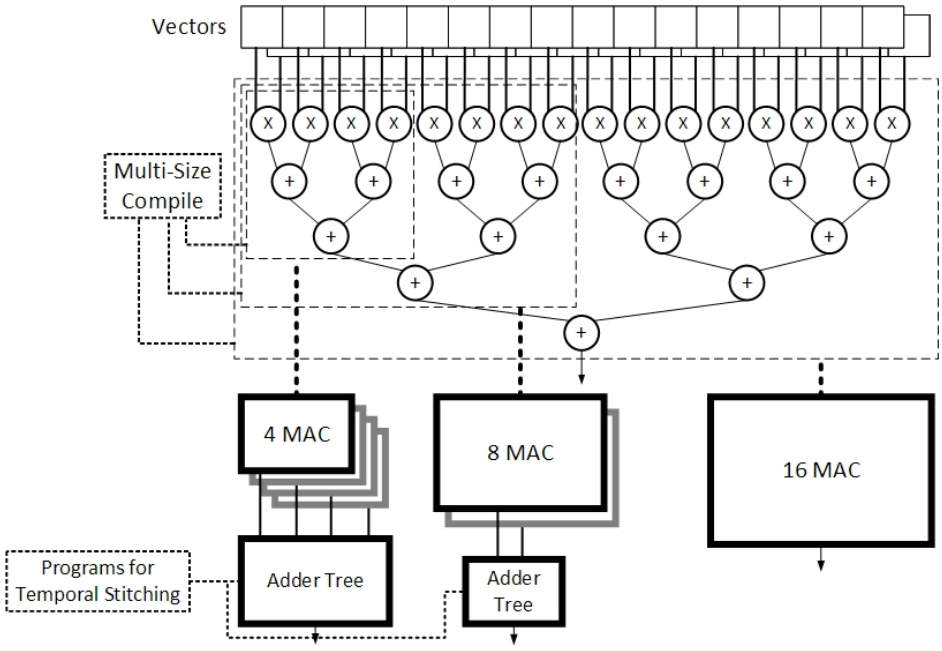


Figure 4.17: A large vector input MAC tree can be broken into smaller MAC programs with temporal stitching programs to finish computation from smaller DFG MACs.

4.7.2 Optimally Sized Programs

One of the factors that can usually get ignored in the compilation and design process of a reconfigurable array is its programming time. The architecture of UDSP allows us to only configure the cores that are needed for the program, which is different than some of the commonly available reconfigurable architectures like FPGA where the entire array has to be configured for every program, even for small programs. Since in traditional hardware arrays the entire array is configured for every new program the programming time is a constant, and thus the programmers has to maximize the resource use of the array to minimize the execution time. In our study we find that, the above architectural model is sub-optimal, and since UDSP allows for programming only the required processing elements for a given program, instead of maximizing the resource usage for each program we should optimize the size of program, so that we can balance the programming time of the processing elements required for the program and the execution time of the program.

There are two main factors affecting the total time for a single program, when the number of processing elements allocated for a program is increased, i.e the size of program is increased, the programming time required to configure all of those processing elements also increases, and at the same time, as more resources are allocated to the execution of the program, the execution time required by the program decreases. Figure 4.19, shows an example of sizing of the program with its execution, programming and total time. In the example we have a $[100 \times 100] \times [100 \times 1]$ matrix-vector multiplication, where we implement a MAC with an adder tree with varying vector input sizes. As we vary the size of input vector for the MAC tree, different amount of computation cycles are needed to finish the required computation by the program. In the Figure 4.19, we can observe that as we implement a wider vector input MAC tree, the number of resources or processing elements required also increases, so the programming time increases while the execution time is decreasing.

The multi-size compile paradigm allows the compiler to traverse the graph as shown in the Figure 4.19 and generate multiple programs of various sizes. We go over the detailed description and performance numbers of each program classification and sizes in the results

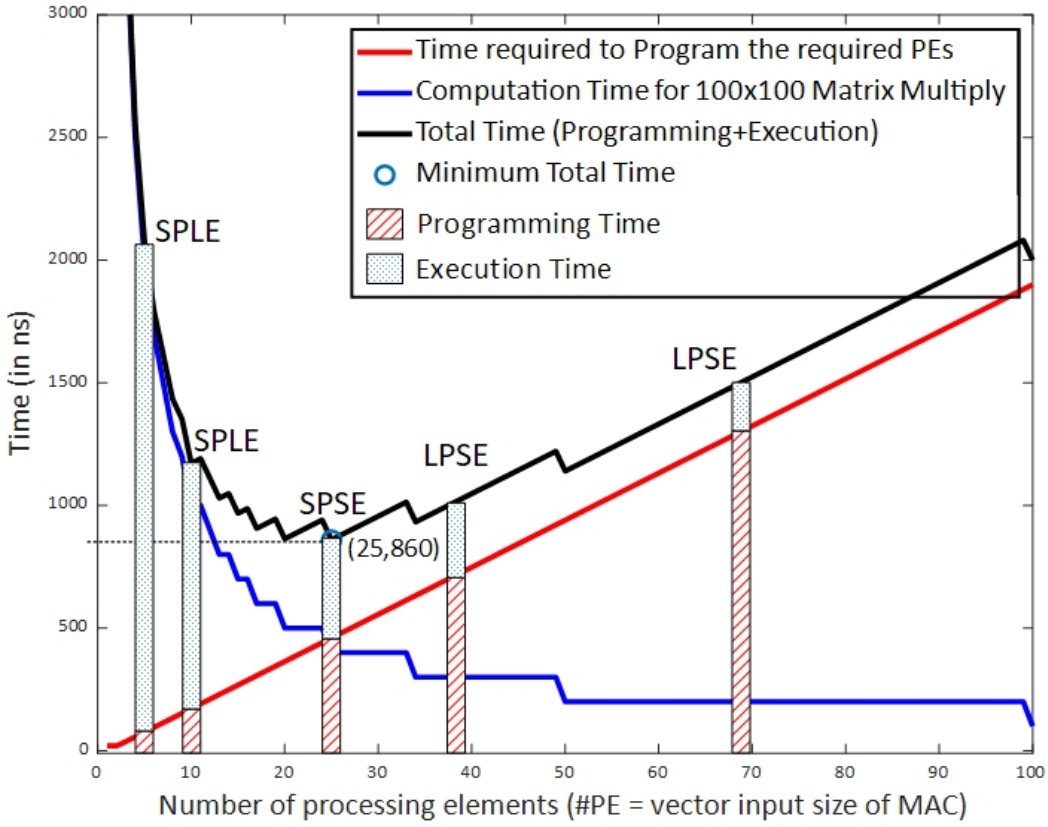


Figure 4.18: Multi-size compiler traversing multiple program classification types of the MAC program.

section of RTRA Chapter 5. Here we provide a brief overview of various types of classifications of a program based on their program time and execution time duration:

1. SPSE: Small Program Short Execution.
2. SPLE: Small Program Long Execution.
3. LPSE: Large Program Short Execution.
4. LPLE: Large Program Long Execution.

The compiler traverses the different program types based on the parallelization, folding, unfolding and partitioning of the program as seen in Figure 4.18.

The optimal size and optimal resources allocated for this single program would be the point where the total time, given by Equation 4.2, for the program is minimum, which occurs at the number of PEs which satisifes the Equation 4.5, when the derivative of total time with respect to number of PEs is zero. Thus, if the $[100 \times 100] \times [100 \times 1]$ matrix-vector multiplication is implemented as a 25 input vector MAC tree, it would require the least amount of total time to execute that program. The programming time and thus the total time of operation depends on the programming bandwidth of the array as well, a higher programming bandwidth means that array can be programmed in shorter duration and vice versa, in the Figure 4.19 the programming bandwidth for the array is 20Gbps.

$$Total\ time = Programming\ Time\ +\ Execution\ Time \tag{4.2}$$

$$\frac{\delta\left(Total\ time\right)}{\delta PE} = \frac{\delta\left(Programming\ time\right)}{\delta PE} + \frac{\delta\left(Execution\ time\right)}{\delta PE} \tag{4.3}$$

$$\frac{\delta\left(Programming\ time\right)}{\delta PE} = -\frac{\delta\left(Execution\ time\right)}{\delta PE} \tag{4.4}$$

The study for least total time for single program is only applicable for single program, and does extend very well for a multiprogram scenario. The programming bandwidth of

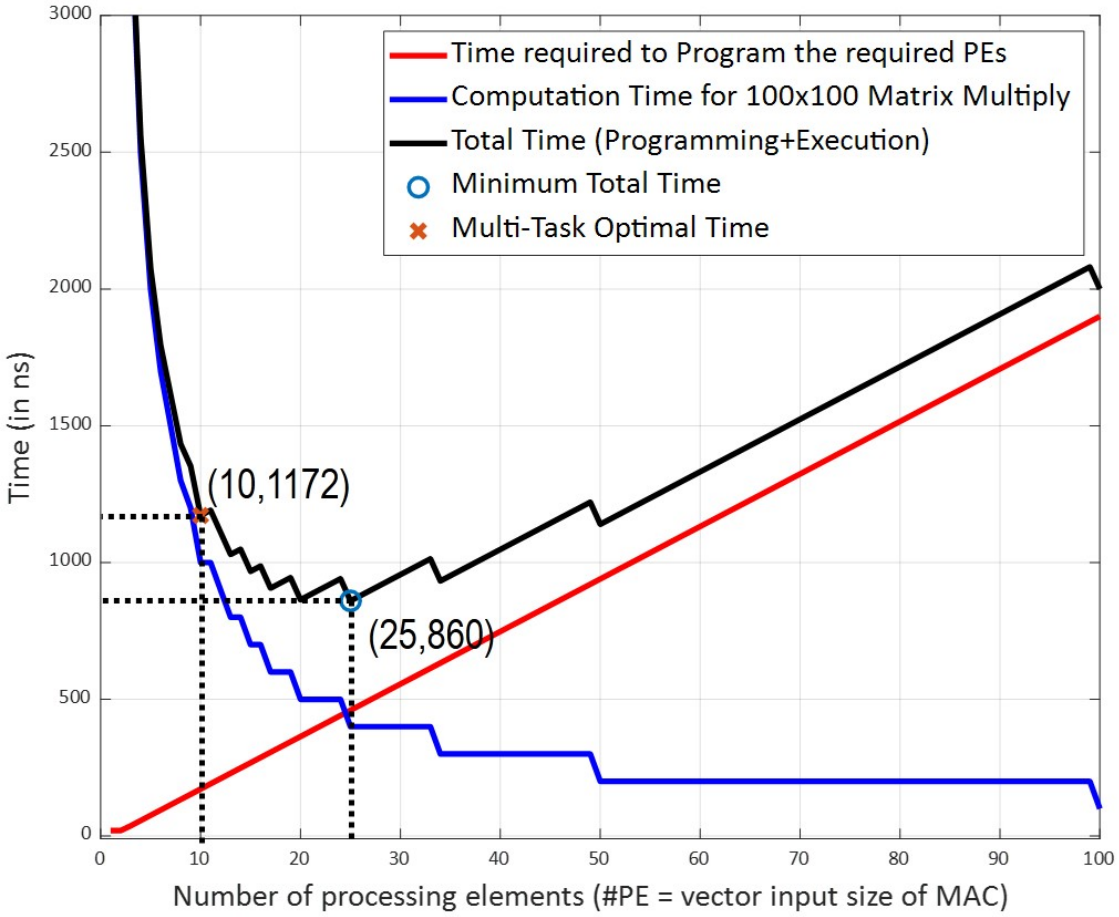


Figure 4.19: Multi-size compile for a $[100 \times 100] \times [100 \times 1]$ matrix-vector multiplication, representing here the programming time, execution time and total time to perform the operation with varying sizes of vector inputs on the multiply accumulate unit. Programming bandwidth of the reconfigurable array used to generate the graph is 20Gbps.

the array is fixed at 20Gbps, and only a single program can be programmed at any given instant of time while utilizing the full programming bandwidth. So, if the compiler were to optimize each program for its minimal total time, the computation flow would follow the Figure 4.20a. For simplicity to represent the problem we assume that we are executing 4 equally sized programs and each program is similar to the size of the array. Its easy to observe that the active utilization of the array in this scenario is quite low. The minimal total time point happens at *programming time* = *execution time*, which means that the array is programming program 2 while program 1 is executing, and thus array utilization can never go higher than that of a single program. With some heuristic driven optimization as well as the knowledge of program statistics, the better operating point for such a multiprogram operation would occur if each program was sized to occupy only the $1/4^{th}$ of the array, since we are trying to execute 4 programs on the array, as shown in Figure 4.20b.

The above condition assumes that the statistics of program execution at runtime is known in advance and additionally it all of the programs are assumed to have identical resources utilization as well as identical execution times, theses assumptions might not be true in a realistic scenario. So, we come up with a generalized heuristic, which the compiler can follow at the compile time, much before the actual execution of the program and optimally size the programs for multiprogram execution scenario without any prior knowledge of runtime program statistics. The heuristic developed is given by the Equation 4.5. The heuristic was designed while keeping in mind that an incremental increase in the number of PEs for a single program gives a decreasing rate of return in total time, while if the same number of PEs are dedicated for a different newer incoming program, the benefits in total time are far greater. As an example, assume that their are 2 programs, program 1 has an execution time of 1,000 clock cycles for a single PE, and program 2 has an execution time of 1,000 clock cycles for a single PE as well. At this stage let's say we allocate 4 PE for program 1 thus bringing down its execution time to 250 clock cycles, however if we were to allocate one another PE to the program 1, its execution time would decrease from 250 $\rightarrow$ 200. If instead of programming an additional PE for program 1, we had spent that programming

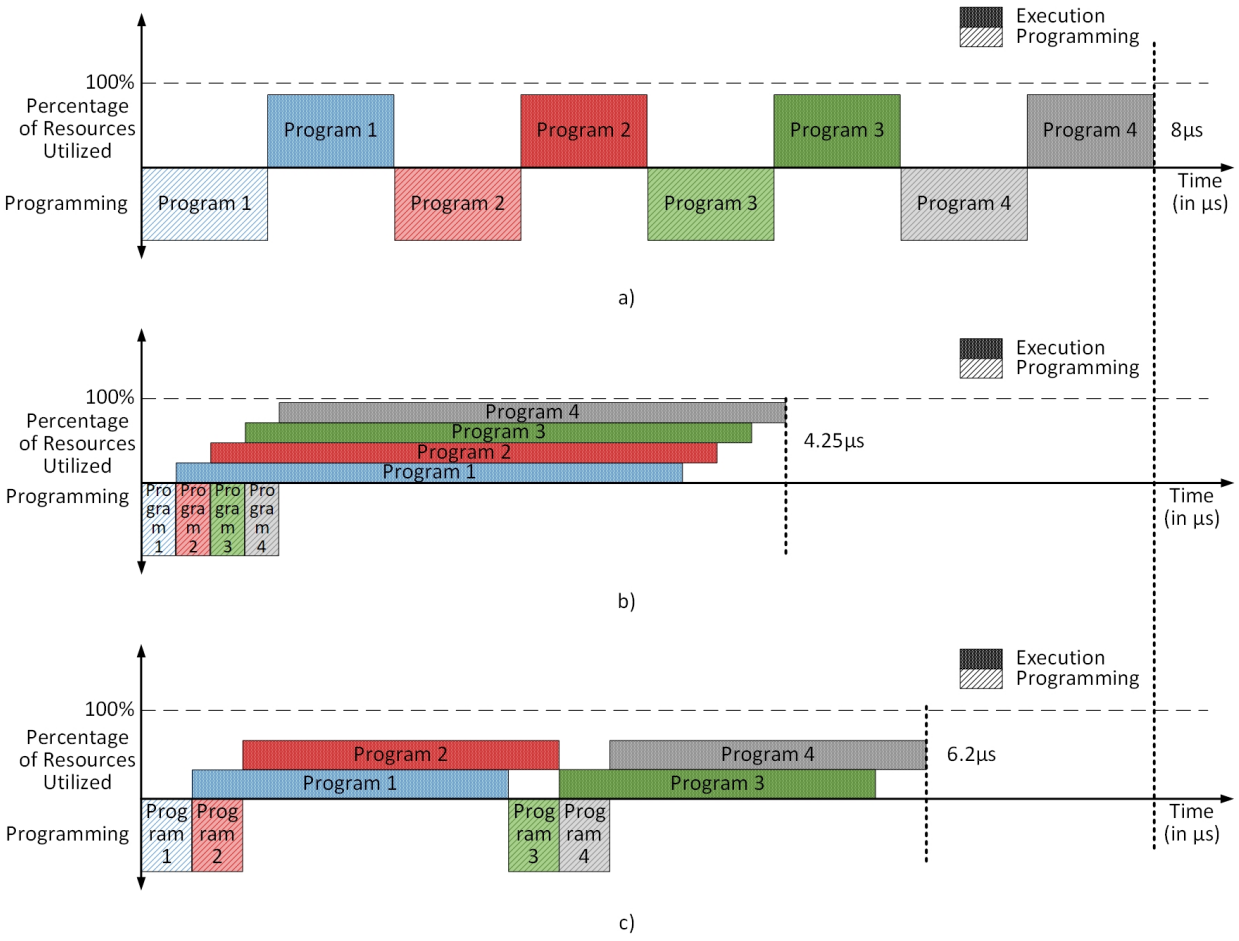


Figure 4.20: Total time for the $[100 \times 100] \times [100 \times 1]$ matrix-vector multiplication program under various different multi-size compilation policies. a) The compilation policy where each program is compiled to run at minimum total time. b) The compilation policy where the program runtime statistics are known in advance and the compiler accounts for the statistics during the compilation sizing. c) The compilation policy where compiler follows the heuristics driven multi-sizing optimized for unknown runtime multiprogram statistics.

time to program an additional PE for program 2, its execution time would have decreased from $1,000 \rightarrow 500$, thus giving a greater rate of return.

$$\frac{\delta\left(Programming\ time\right)}{\delta\left(PE\right)} = -\frac{\delta\left(Total\ time\right)}{\delta\left(PE\right)} \tag{4.5}$$

The optimal size heuristic effectively allocates the number of PEs to each program where the rate of decrease in total time per additional PE is lesser than the rate of increase in programming time per additional PE, given by the Equation 4.5. The program execution flow of the same vector matrix multiplication program with optimal sizing as driven by heuristics without prior knowledge of runtime program statistics is shown in Figure 4.20c.

4.7.3 Multi-Step Compilation

In RTRA architecture, we rely upon multi-step compilation to achieve a true multiprogram array use virtualization. The multi-step compilation feature is enabled by the ease of compile for UDSP architecture coupled with its translation, mirror and rotational symmetries. A program compiled for UDSP resources at a particular location can be relocated to any other location with minimal changes to the program configuration bits. This operation is achieved by the hardware compiler on-chip besides the array, however to assist and speed up the hardware runtime dynamic compile operation some modification are required in the software compilation process. We effectively break down the work of compilation into two parts, software compiler and hardware compiler. Most of the iterative and time consuming steps required for program compilation such as resource clustering, placement, and inter-PE routing are taken care of before runtime inside the software compiler as we discusses in previous sections. However, instead of generating a hard mapped fixed program binary, the software compiler generates a soft-mapped program binary with some additional metadata to assist the runtime hardware compiler to easily relocate and hard map the program to the available resources on the array as shown in Figure 4.21.

The metadata included in each soft-mapped program binary consists of:

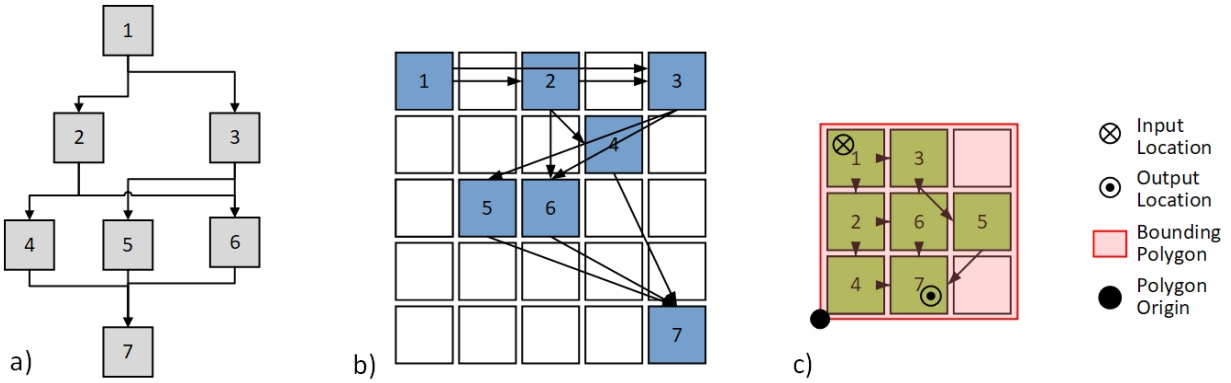


Figure 4.21: a) Example DFG user input. b) Randomized initial placement of the DFG. c) The polygon, origin and other metadata information added to the soft-mapped program.

- A bounding polygon and origin; the bounding polygon defines the physical array resource boundaries that are required by the program. The vertices of the polygon are defined with respect to an origin node.

- Input/output locations; a list of locations of all the input and output ports to the soft-mapped program is added to the program metadata such that the hardware compiler can route the program IO ports to the physical IO ports of the array. This information is added to account for program relocation, which would also relocate the location of IO relative to the physical array. The origin node of the polygon is defined to be the vertex which is adjacent to the edge with high number of IO ports, this makes the process of placing the program as well as routing the IO ports easier and faster.

- Timing information; the program execution time and tentative program start time is added to the polygon of each multi-sized program. Depending on the extent of parallelism during multi-size compile, each of the compiled binaries of varying sizes would have a different execution length. This timing information helps the on-chip hardware scheduler keep a track of the programs in the waiting list which need to be mapped as well as vacating programs from the array which have finished their respective execution.

CHAPTER 5

RTRA for Accelerator as a Service (ACAS)

Computation heavy programs require large number of resources, thus necessitate the development of large arrays with high number of processing elements. However, within the domain of applications the compute requirement of different programs may vary. Programs requiring large number of resources could potentially achieve higher active utilization of the array while the smaller programs requiring fewer resources would not be able to actively utilize all of the resources available on a large array, and thus lead to lower active utilization. This inability of actively using the resource leads to inefficiency in program execution and a lower aggregate throughput of the array. We develop a runtime active management system for the array, using a hardware scheduler and an onboard hardware compiler, which allows multiple programs to be simultaneously mapped onto the array, enabling multiprogram tenancy. This technique virtualizes the array resources over multiple programs and allows for a higher active hardware utilization using multiple programs as compared to a single program. In this chapter, we go deeper into the hardware resources and blocks which enable such virtualization.

5.1 Introduction

While adding a reconfigurable array to the system, designers may decide upon the size and number of processing elements on the array based on several different factors, a few of them could be:

- Size of the largest program which could be required to be accelerated by the recon-

figurable array. Figure 5.1 shows an example program for MobileNet Convolutional Neural Network (CNN) [19] where multiple filters in convolutional layer are acting on the same input matrix.

- The size of the array could also be decided based on the throughput requirements of program/programs. To match the throughput requirement the program can be unfolded and parallelized which increases the number of resources required on the array.

- Some set of programs or program kernel which require simultaneous execution, an example could be implementing RF digital front end on the reconfigurable array, which could consist of digital up converter, digital down converter, digital predistortion filters, and other parallel multirate FIR filters, as shown in Figure 5.2.

While the size of the array could be decided based on the algorithms and programs which stretch the limits of throughput of the array, its possible that the array is not always executing such large programs. A reconfigurable array attached to the system can also be used to accelerate other simple algorithms like image encoding/decoding, vector matrix multiplications, loop acceleration, K-means clustering which require fewer resources on the array as shown in Figure 5.3. Since in a typical reconfigurable array, only a single program or a precompiled set of programs can occupy the array at any given time, this leads to an inefficient use of the array and poor array utilization.

This premise motivates the development of actively managed Runtime Reconfigurable Array (RTRA). An array-based accelerator where the configuration, location and programming bits of any incoming program can be dynamically composed within the array hardware, thus enable Accelerator As a Service (ACAS) paradigm. We develop a hardware scheduler, hardware compiler and supporting hardware resources and software paradigm which help actively manage hardware array resources among multiple incoming programs. The management system calculates the available vacant resources on the array, and feasible resources to map incoming programs and then allocates and maps those programs to the resources,

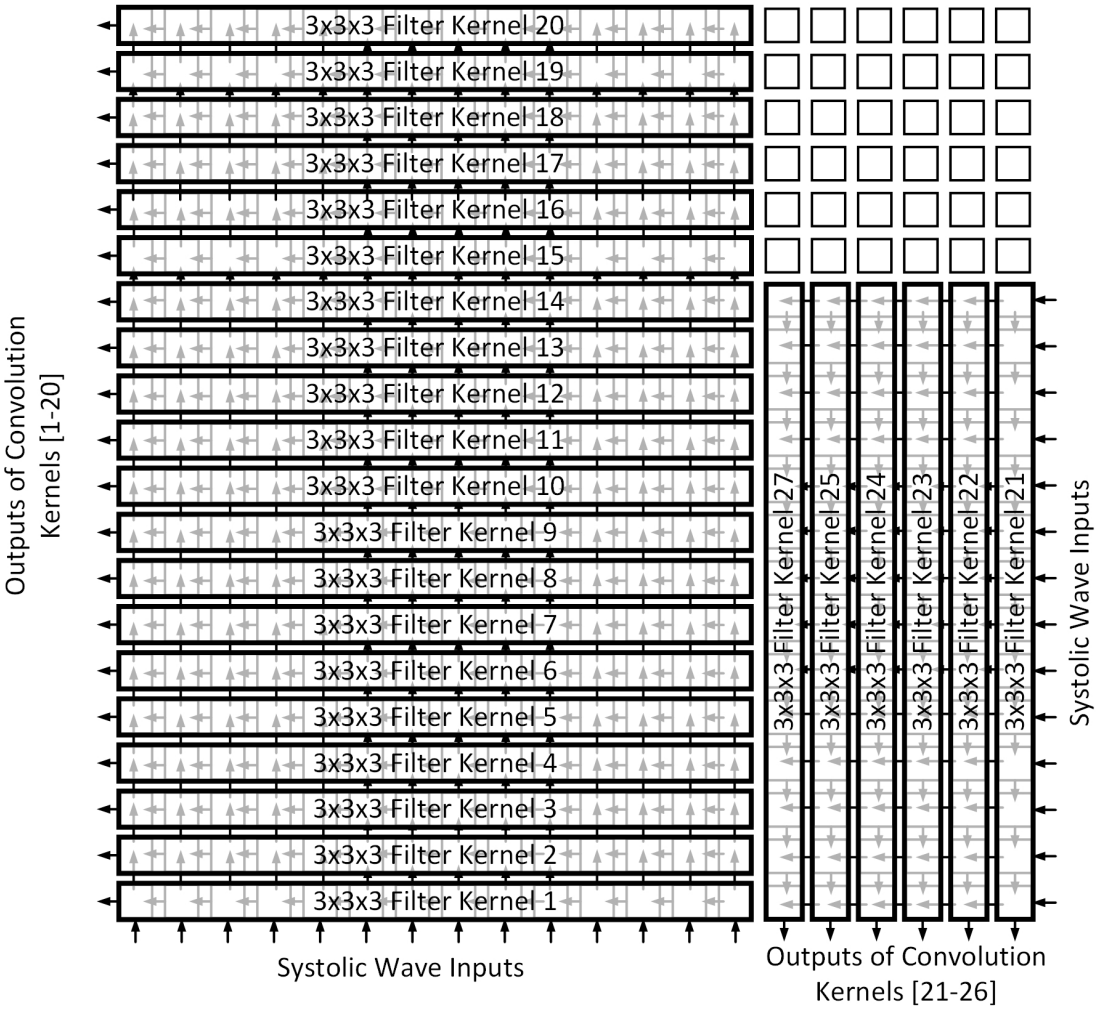


Figure 5.1: An example mapping showing 27 mapped kernels from the first layer of $3 \times 3 \times 3 \times 32$ convolution kernels of MobileNet CNN [19].

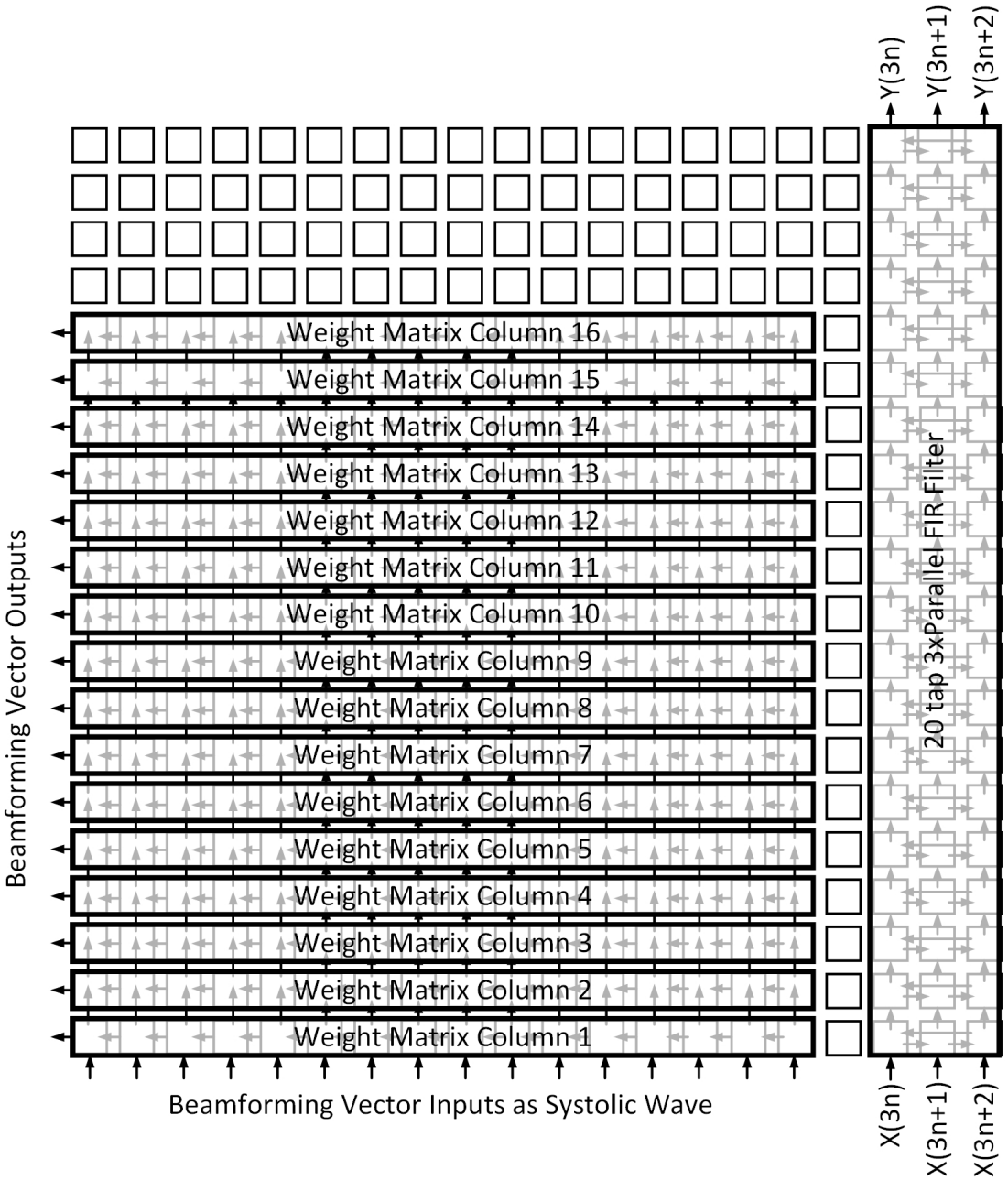


Figure 5.2: An example mapping of line-rate 8×16 beamforming MIMO along with a 20-tap $3\times$ parallel FIR filter.

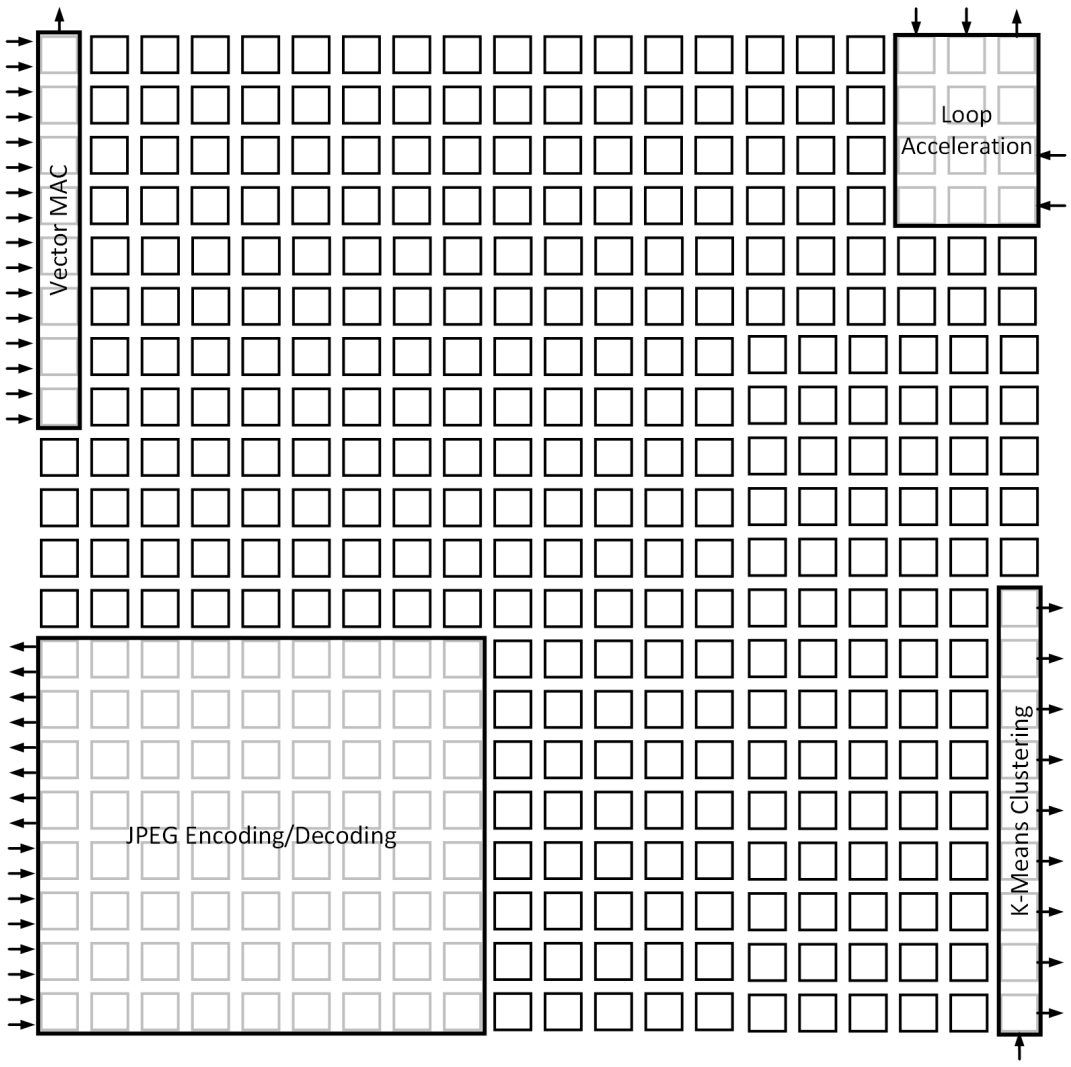


Figure 5.3: An example mapping of some small physical footprint programs. Although the programs are shown simultaneously mapped on the array, the active array utilization is still low, this would have been even lower is the programs were mapped to the array independently and one at a time.

discussed in detail in the next section.

5.2 Runtime Reconfigurable Array

The overview of the architecture is shown in Figure 5.4 and Figure 5.5. The architecture can be connected to the host system and assembled on an SoC or it can be connected to multiple clients and hosts over a network. In this section we go over the overall working of the Accelerator as a Service (ACAS) on RTRA briefly and then we will cover the architectural components that enable the active management of resources on the array. The working process of RTRA is as follows:

- The host/client sends the polygon information and program metadata over to the array.
- Scheduler receives the information and compares the execution start time of the program with the current time, and decides whether to map the program onto the array.
- When the incoming program needs to be mapped onto the array, the scheduler uses the polygon-based overlap detection technique at each of the anchor points and calculates appropriate location to map the incoming program.
- Once the vacant resources have been calculated for the incoming program, the scheduler sends the program translation functions to the hardware compiler.
- The hardware compiler modifies the program binary of the incoming program to translate it to the allocated resources on the array.
- Then the hardware compiler configures the IO network and memory controller to provide the appropriate streams of data to the appropriate program ports along the physical ports of the IO network.
- Once the program has been successfully mapped, the hardware compiler idles and the scheduler starts on its bookkeeping mechanism.

- In the bookkeeping mechanism the scheduler keeps track of valid and executing programs on the array and generates valid anchor points so that the new incoming programs can be mapped in least amount of time. It also vacates the programs which have finished execution and invalidates any of the anchor points which were contributed by any of the retired programs.

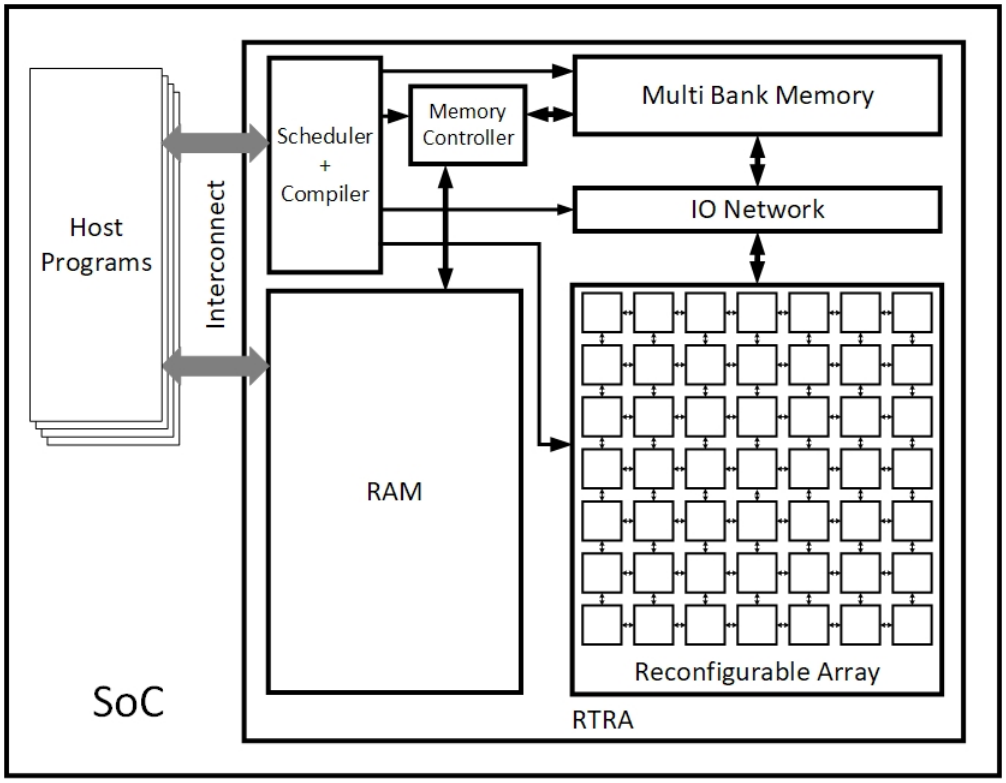


Figure 5.4: Architectural overview for RTRA connected to a host system on SoC. Multiple programs on the host can interact with the RTRA virtualized accelerator.

5.2.1 Software Compiler

In this section we provide a brief overview of the software compiler required to understand the functioning of other hardware components and would be discussed in detail in the Compiler Chapter 4. The software compiler generates reconfiguration bits of a soft-mapped program for the reconfigurable array using the data flow graph of the input program. The

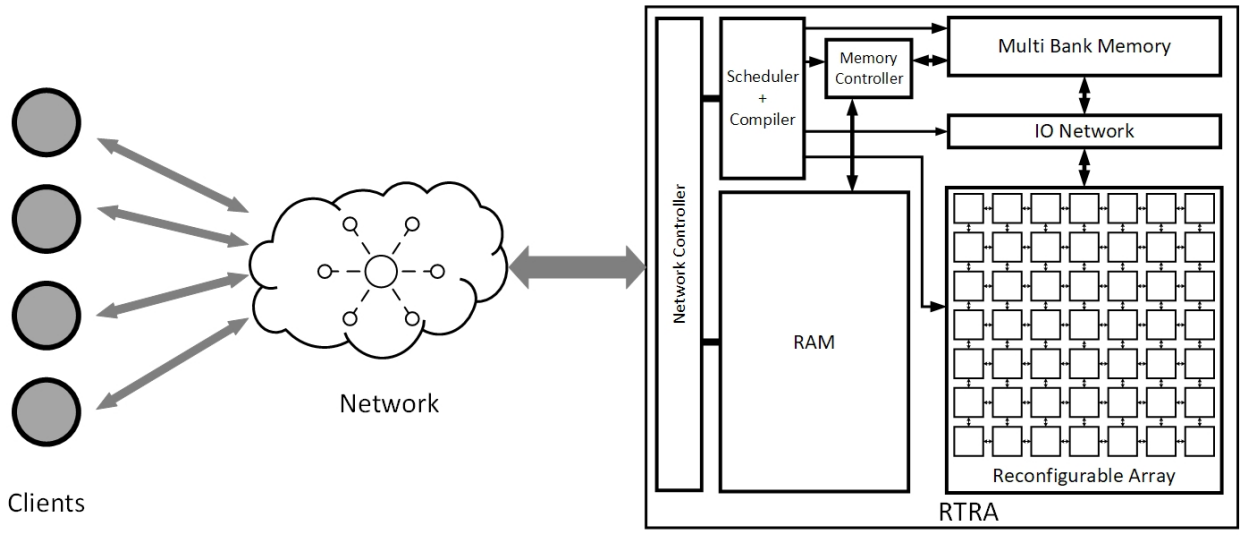


Figure 5.5: Architectural overview for RTRA connected to the network and multiple clients and programs interacting with it over the network.

soft-mapped program consists of a configuration of processing elements and the required interconnect between them bound by a polygon shape and some additional metadata as shown in Figure 5.6.

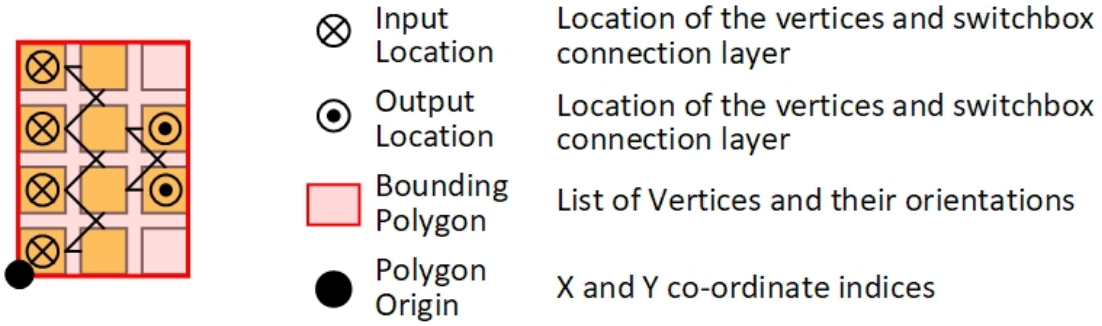


Figure 5.6: An example of a soft-mapped program with the additional metadata as output by the software compiler.

The scheduler is able to map and translate the polygon on various locations on the array by exploiting various symmetries of the array. The bounding polygon surrounding the soft-mapped program allows the scheduler to easily check for program overlaps with the existing programs on the array, and quickly translate and relocate the polygon and thus

program to available resources on the array, while preserving the internal connections and configurations of the processing elements inside the polygon. The polygon-based translation and relocation helps avoid computationally complex software recompilation. Along with the polygon metadata, software compiler also includes information for the program's execution time, as well as its execution length, location of polygons origin and the location of IO ports on the polygon with respect to the origin.

5.2.2 Scheduler

The scheduler performs the task of spatial and temporal array resource allocation for incoming programs. It keeps track of programs actively executing on the array and their respective locations and uses that information to map any incoming programs to the available resources on the array without overlapping the hardware resources of the array required by the incoming program with the existing programs. If the scheduler is unable to spatial co-map the incoming program to the array along with other existing programs then it adds the incoming program into a temporal list of program which can be mapped onto the array when the active programs on the array retire or finish execution. To perform this task of spatial and temporal multiplexing and scheduling, the scheduler uses the metadata information provided by the software compiler. The metadata information consists of the bounding polygon, origin, location of input/outputs ports of the program, program's requested start time, and program's execution time period, as mentioned in the previous section. The bookkeeping mechanism and placement mechanism work together to ensure proper and quick functioning of the scheduler, in the next section we look deeper into those mechanisms.

5.2.2.1 Bookkeeping Mechanism

The bookkeeping mechanism keeps track of polygons mapped on to the array. When a polygon gets mapped onto the array it is added into a list of programs, called bookkeeping list. This bookkeeping list keeps track of currently executing programs on the array as well as the programs which were requested but unable to be mapped currently to the array. The

main function of bookkeeping mechanism is to generate a list of anchor points based on the the currently active and executing programs, this list of anchor points is used by the placement mechanism to figure out the relocation of new incoming programs or the temporal list of programs. The dimensions of the array are saved as program 1 in the bookkeeping list with the infinite execution time length, and the highest program priority, this allows the mechanisms to treat the physical dimensions of the array as an inverted polygon with the 4 corners of the array as anchor points, which get automatically generated when the array is started and/or reset, or when the array finished executing all of the programs in its to-do list. In case of any hard faults on the array due to manufacturing defect or any other mechanical/thermal reasons, we can modify the program 1 which is the array dimensions or add another program 2 with indefinite execution time, thus instructing the hardware scheduler to place and map the other incoming programs around the hard fault.

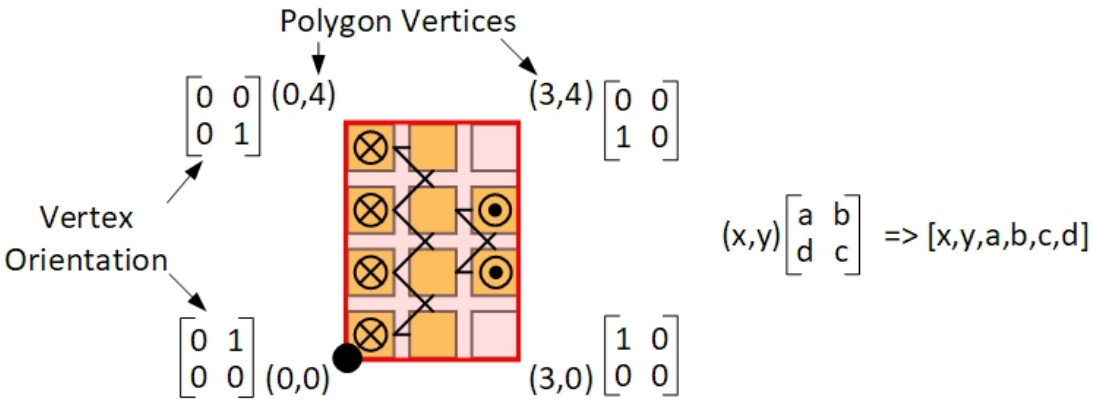


Figure 5.7: An example of a program with its bounding polygon. The bounding polygon is defined by its vertices and their respective orientations.

The polygon metadata of the program consists of the vertices of the polygon and their orientations as shown in Figure 5.7, among other things. While generating and updating the list of anchor points the scheduler goes through each valid and active polygon in the bookkeeping list. It goes through each polygon one by one, and performs these steps:

- Check if the polygon is active. To perform this check the bookkeeping mechanism

compares the actual start time of the program on the array with the current time of the array. Since each program executes for a fixed length of specified time, the mechanism can thus calculate the active, retired as well as programs waiting for execution in the bookkeeping list. If no program has changed its status since the last check, that means the anchor points list generated in the last cycle is up to date and does not require any updates. In this scenario, the bookkeeping mechanism skips the remaining steps in the list, and repeats this step until any change is detected in any program's status.

- Once the mechanism verifies the activity of the program. It uses the orientation information of each of its vertices to create a list of tentative anchor points, as shown in Figure 5.8. It then checks for overlaps of the tentative anchor points with other existing valid polygons on the array. If the anchor point does not intersect/overlap with any of the existing polygons on the array it is added to the list of anchor points.

- Upon updating the list of anchor points, the mechanism then checks for any programs which finished execution and retired since the last check. If so, the mechanism goes through the programs which are awaiting execution in the bookkeeping list, and tries to map them to the newly freed resources using the placement mechanism.

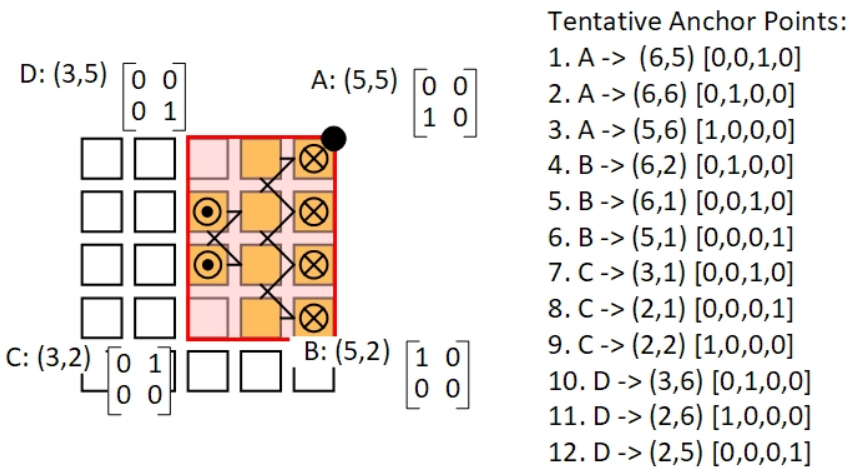


Figure 5.8: A tentative list of anchor points generated from a placed program polygon in the bookkeeping list.

The bookkeeping mechanism runs continuously in the background of the scheduler, while its idling. The scheduler is idling during the time between program requests, statistically between any two program requests there would be a few hundred cycles where scheduler is not performing any placement operation, and it performs bookkeeping mechanism during those times. This background process of bookkeeping provides scheduler with an updated list of anchor points and the pre-processed information, which speeds up placement of any incoming program when they do.

5.2.2.2 Placement Mechanism

Scheduler uses the placement mechanism to exploit the symmetries of the reconfigurable array and place polygons of the incoming programs onto the available resources on the array. If the array is translation symmetric, that would allow the scheduler to translate any incoming program to different resources on the array while maintaining the functionality of the program, similarly rotational symmetry allows program to be rotated and mirror symmetry along any axis would allow any incoming program to be flipped about that axis while maintaining functionality and without requiring lengthy software recompilation procedure. In our design the array is translation, rotation as well as mirror symmetric which allows us to translate, rotate and flip any incoming program to map onto the available resources on the array without affecting the functionality of the program.

The placement mechanism places the polygon of the incoming program on each anchor point in the anchor point list while aligning the orientation of the vertex of the incoming polygon with the orientation of the anchor point. It is a two step procedure, where the mechanism first translates the selected vertex of the incoming polygon, typically origin of the polygon, to the location of the anchor point and then rotates the polygon around that vertex to match the new orientation of rotated incoming polygon to that of the anchor point as shown in Figure 5.9. In the Figure 5.9b, rotation Program 2b would be tried for placement at the selected anchor point, since the orientation parameter of the anchor point i.e [0,0,1,0] would match that of the vertex of the rotated polygon i.e. [0,0,1,0]. The mechanism then

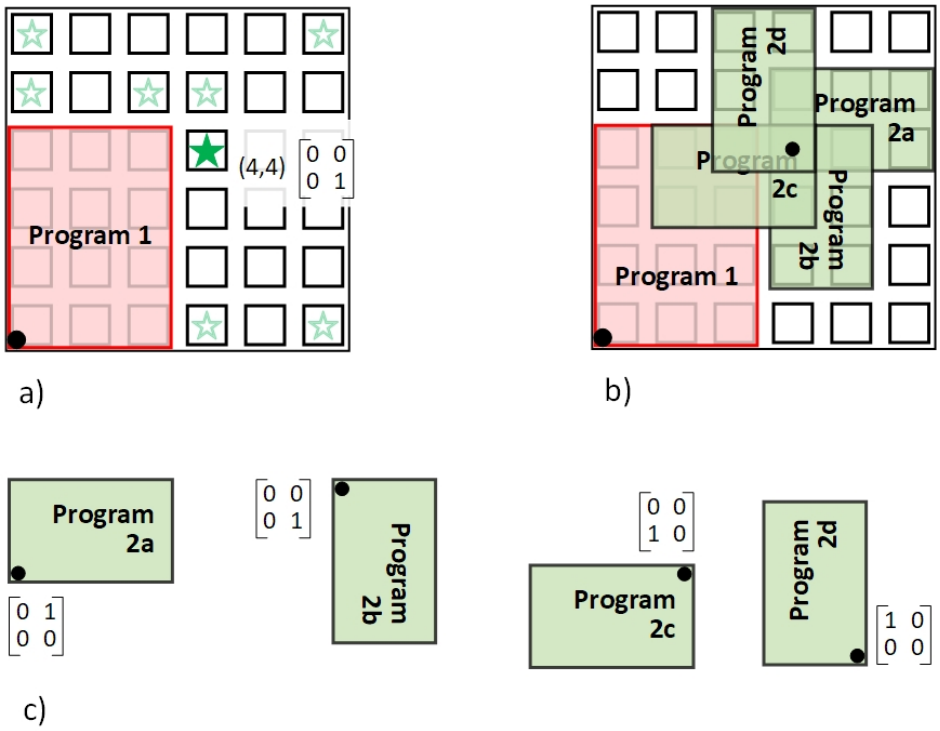


Figure 5.9: Figure demonstrate the use of orientation information of the anchor point and vertex of the polygon. a) The orientation parameters of the selected anchor point. b) The possible rotations of the polygon about its origin or chosen vertex when translated to the selected anchor point. c) The changes in orientation parameters of a vertex with the rotation of the polygon.

repeats this procedure for each of the remaining anchor points in the list, until successful. If the mechanism is not successful in placing the polygon in the first iteration, it tries to flip the polygon along its x-axis and repeats the placement procedure, and if still unsuccessful it can repeat the procedure with a different vertex of the incoming polygon.

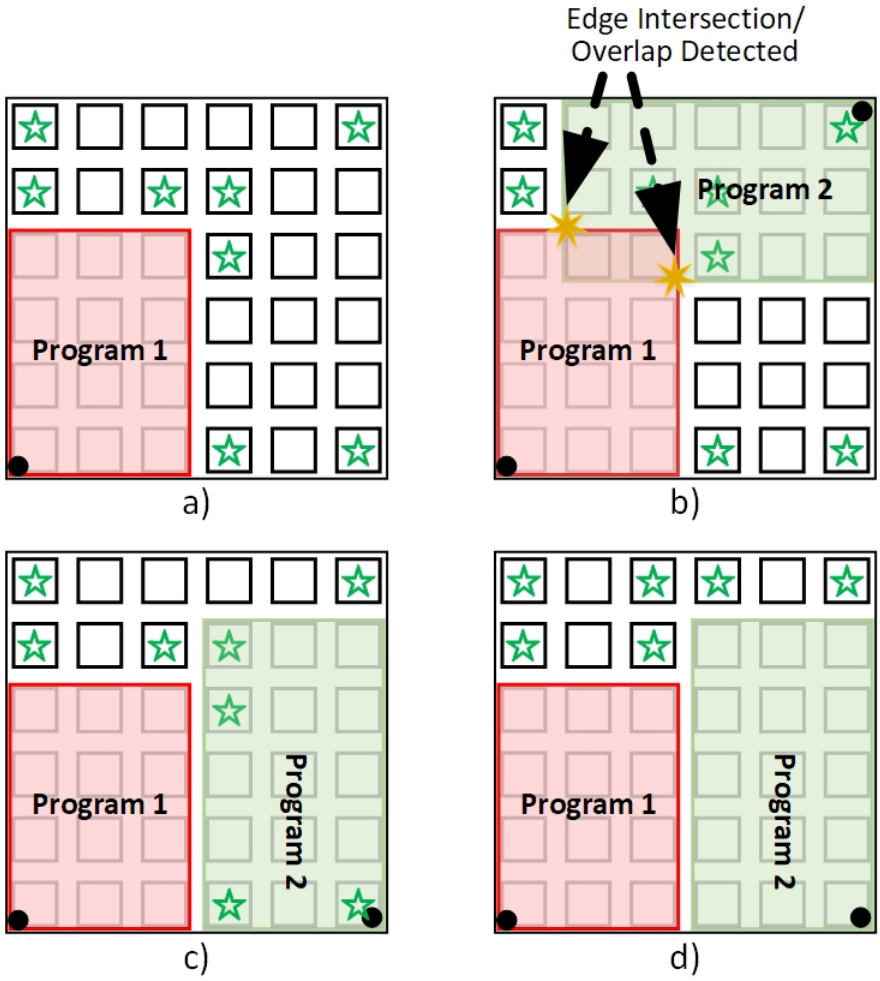


Figure 5.10: A tentative list of anchor points generated from a placed program polygon in the bookkeeping list.

The check for program overlap at each anchor point is greatly simplified due to the bounding polygons. Instead of checking overlap of processing element used by the existing programs on the array and incoming programs, the mechanism simply checks for intersection of the polygon edges of the programs. The mechanism to check for program edge intersection

is independent of the size of polygon/program and only depends on the number of edges of the polygon. In our implemented design we chose a bounding polygon with four edges, which meant that we had to check for intersection of 16 different pairs of edges while checking overlap of any two programs, as shown in Figure 5.10. This is a greedy procedure, since the mechanism stops at the very first successfully placement, various other mechanisms can be developed to ensure that the occupied regions, or placed programs span the least amount of area on the array, and leave large contiguous resources vacant for incoming programs in future.

Once the placement mechanism is able to successfully place the incoming polygon onto available resources on the array, it relays the information regarding the program translation, rotation and flip to the hardware compiler, which uses the information to modify and hard place the incoming program to the selected location.

5.2.3 Hardware Compiler

The hardware compiler uses the transform information provided by the placement mechanism of scheduler and uses it to transform and map the incoming program to the vacant location as calculated. The task of hardware compiler is greatly simplified because of deterministic and constant routing delays of the interconnect network. Since the interconnect network requires 1ns to perform any 1 hop connection, so rotating and translating and program mapping across the array preserves the timing constraints internal to the program polygon.

The location bits of the programming frame are very clearly demarcated in the programming frame as shown in Figure 5.11. The hardware compiler has to modify the location indices in the soft-mapped program configuration bits to the location indices corresponding to the new location and send the programming bits to the reconfigurable array to achieve desired program rotations and translations following Equations 5.1 - 5.6 as shown in Figure 5.12.

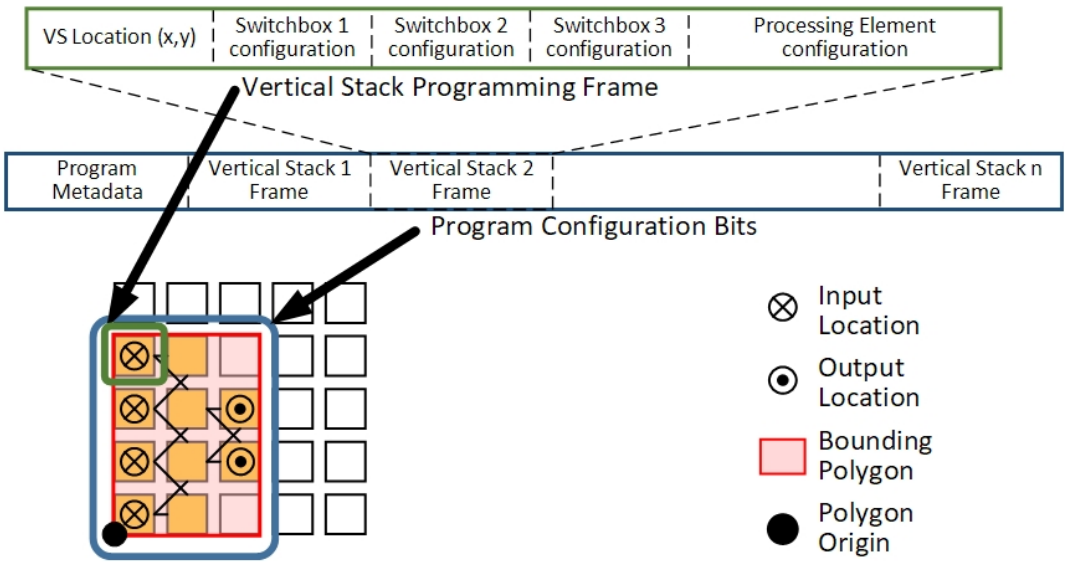


Figure 5.11: The structure of configuration bits of the program, divided into multiple programming frame applicable to its respective vertical stack.

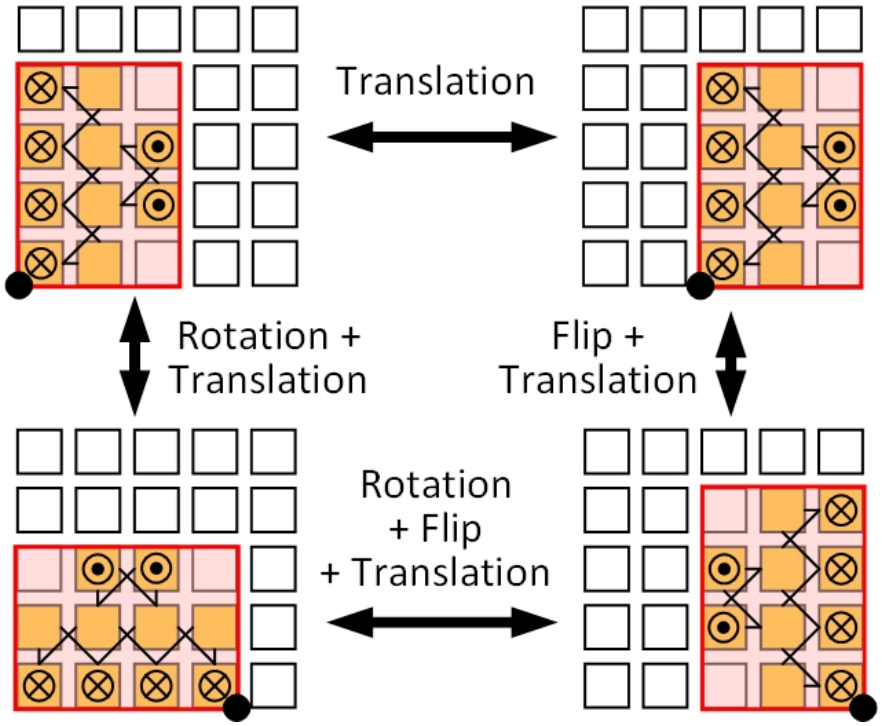


Figure 5.12: The structure of configuration bits of the program, divided into multiple programming frame applicable to its respective vertical stack.

$$Translate\ program\ by\ a\ in\ x\ direction\ and\ b\ in\ y : x, y- > a + x, b + y \tag{5.1}$$

$$Rotate\ program\ clockwise\ by\ 90^\circ : x, y- > y, m - x \tag{5.2}$$

$$Rotate\ program\ clockwise\ by\ 180^\circ : x, y- > m - x, n - y \tag{5.3}$$

$$Rotate\ program\ clockwise\ by\ 270^\circ : x, y- > n - y, x \tag{5.4}$$

$$Mirror\ program\ along\ y\ axis : x, y- > m - x, y \tag{5.5}$$

$$Mirror\ program\ along\ x\ axis : x, y- > x, n - y \tag{5.6}$$

where m and n are dimensions of the program on the array.

5.2.4 Memory Subsystem

The memory subsystem consisting of a large RAM block to store data and a cache was designed to provide streaming data to the array, and accommodate the array virtualization and program relocation strategies. The memory structure for the array is composed of multiple parallel memory banks, memory controller and RAM. The host(s) or client(s) requesting a program stores the data required by the program in the memory. The memory controller performs a scatter-gather based memory transfers. It requests large contiguous words of data from the RAM and stores those words into a respective parallel memory bank address. In our case, each input to the reconfigurable array is a 16-bit word. The output from each individual memory bank in the parallel banks is controlled by an independent sequence generator which is programmed by the compiler. A large parallel data word is constructed from those parallel banks and sent to the ports of IO network towards the reconfigurable array shown in Figure 5.13. Any output from the reconfigurable array is also read out in a similar fashion. The number of parallel memory banks is decided by the network throughput required by the IO network to and from the array. The number of entries or elements in each of those parallel memory banks indirectly depends on the network IO throughput. Since memory controller can only read or write contiguous memory from RAM, the word has to be sufficiently large to accommodate the random nature of RAM, and hide RAM latency

and keep the parallel memory banks filled to provide the streaming data to/from the array in orderly fashion. Each of the parallel banks have identical connectivity to the IO network and array.

During the virtualization process, the hardware compiler uses the memory structure to provide data streams to the virtualized programs. When the hardware compiler is done placing programs and configuring the IO network, it sends the connectivity of the virtual IO ports of the placed program to the physical IO ports of the array to the memory controller. Memory controller uses this connectivity information to move the appropriate word from RAM to its appropriate parallel bank. Since any word from RAM can be placed and accessed through any of the memory banks, this simplifies the design of the IO network.

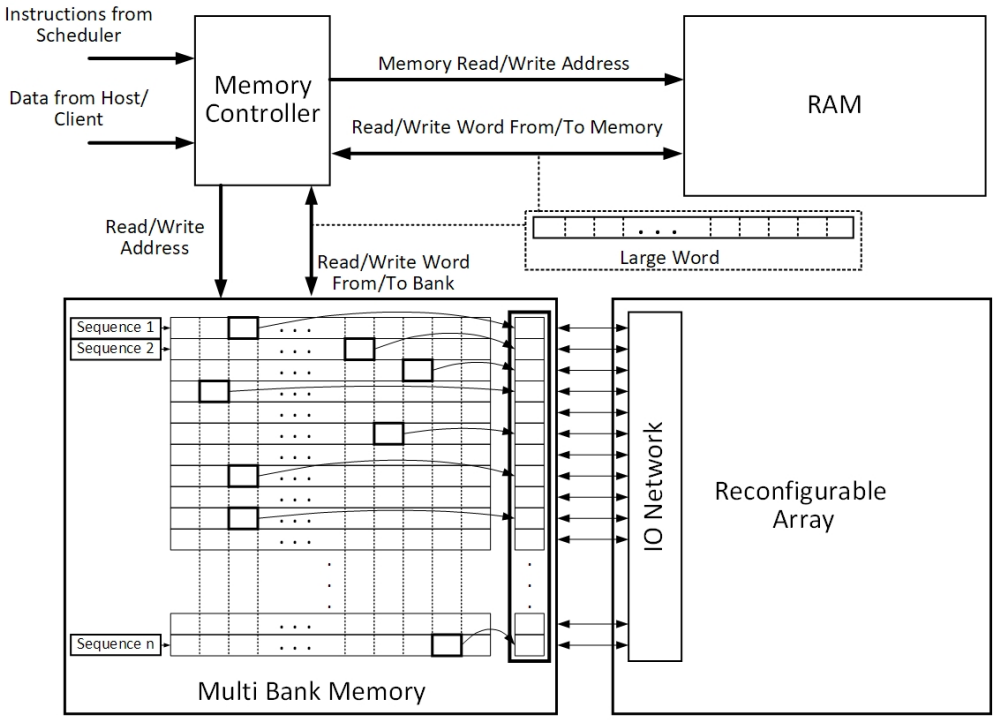


Figure 5.13: Memory structure for RTRA.

5.2.5 IO Network

The ACAS architecture allows for program allocation over the area of the array at runtime, which requires that the program Inputs and Outputs (IOs) need to be connected with the physical IOs of the array using an IO network. The internal connectivity of the program, which is mesh-based, still remains intact as the program is relocated, however the IO ports of the program need to be accommodated during relocation as shown in Figure 5.12. Although a symmetric mesh-based interconnect network is ideal for domain-specific inter-PE connectivity required by programs, it is not ideal for an IO network which connects program IOs on PEs to physical IOs on the boundary of the array. As shown in Figure 5.14 and Figure 5.15, a mesh network when used as an IO network provides limited and non-deterministic routing. Each switchbox node in the mesh network is directly connected to a processing element. So if the node is used by the network for some routing, then its connected PE can be blocked from utilizing its network resources. Mesh networks also create a large number of possible routes, and solving for the least-delay route while accounting for blockages at runtime requires a high time complexity. The latency to map any physical IO to a program IO to any array location is also not uniform, as can be seen in Figure 5.15, in the example mesh of 5×5 array size, the array location $[1, 1]$ is directly connected to physical IO and hence can be reached within one clock cycles, however the array location $[3, 3]$ can only be reached after three hops in the temporal layers going from various different neighbouring mesh node switchboxes. The other solution for IO network could be a high-radix, $n \times n$ crossbar, where n is the number of processing elements on the array. This is a hardware costly solution and thus not feasible for larger arrays.

The IO network designed as shown in Figure 5.16 can be summarized as optimally sparse multi-layer network. The optimal sparsity ensures that there is minimal overlap between the input nodes and output nodes in each layer while providing sufficient connectivity from each PE to the physical IO, meaning no two output nodes in a layer share more than one common input node. The number of nodes in each spatial layer is proportional to the number of processing elements (equal in this example) and each node in the spatial layer has a fixed

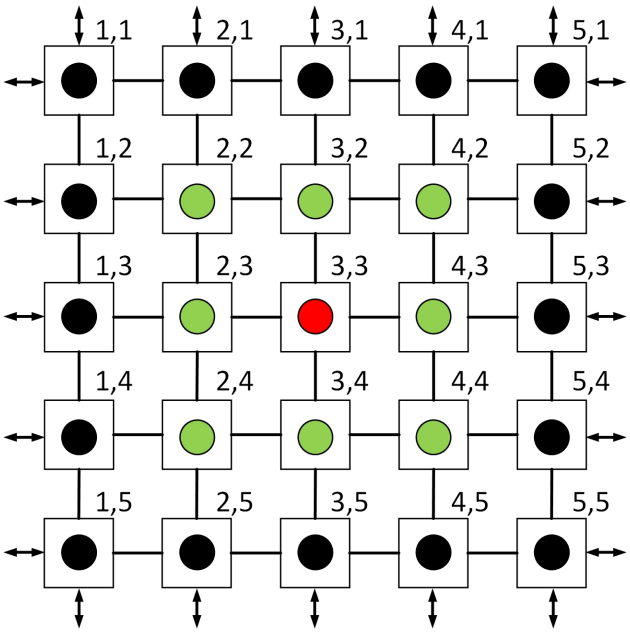


Figure 5.14: 2D spatial representation of the mesh-based IO network for RTRA.

node degree. The sparsity ensures that a disjoint path exists for each PE to the physical IO at a minimum hardware cost.

When compiling the routing and configuration for this network at runtime, each program IO in a PE is back-propagated towards the physical IO. This operation outputs possible routing paths as a tree for each program IO, with the root of the tree being the Program IO on the PE, the intermediate nodes being the spatial layer nodes, and leaf nodes being the physical IOs. The disjoint paths between those trees provide a valid configuration for the IO network to connect each of the PEs to a physical IO. We apply a depth-first search to look for those solutions within a fixed number of iterations. If the hardware scheduler fails to find a feasible solution, the IO routing is treated as failed and the program placement is tried again at a different location. The network is statistical, it does not guarantee a solution like a fully-connected network but the probability of finding a solution is engineered to be high by adjusting the number of nodes in intermediate layers and their degree of connectivity for a given array size. For incoming programs with high compute-to-IO ratio, the program and the array would be compute limited, which eases the routing pressure on

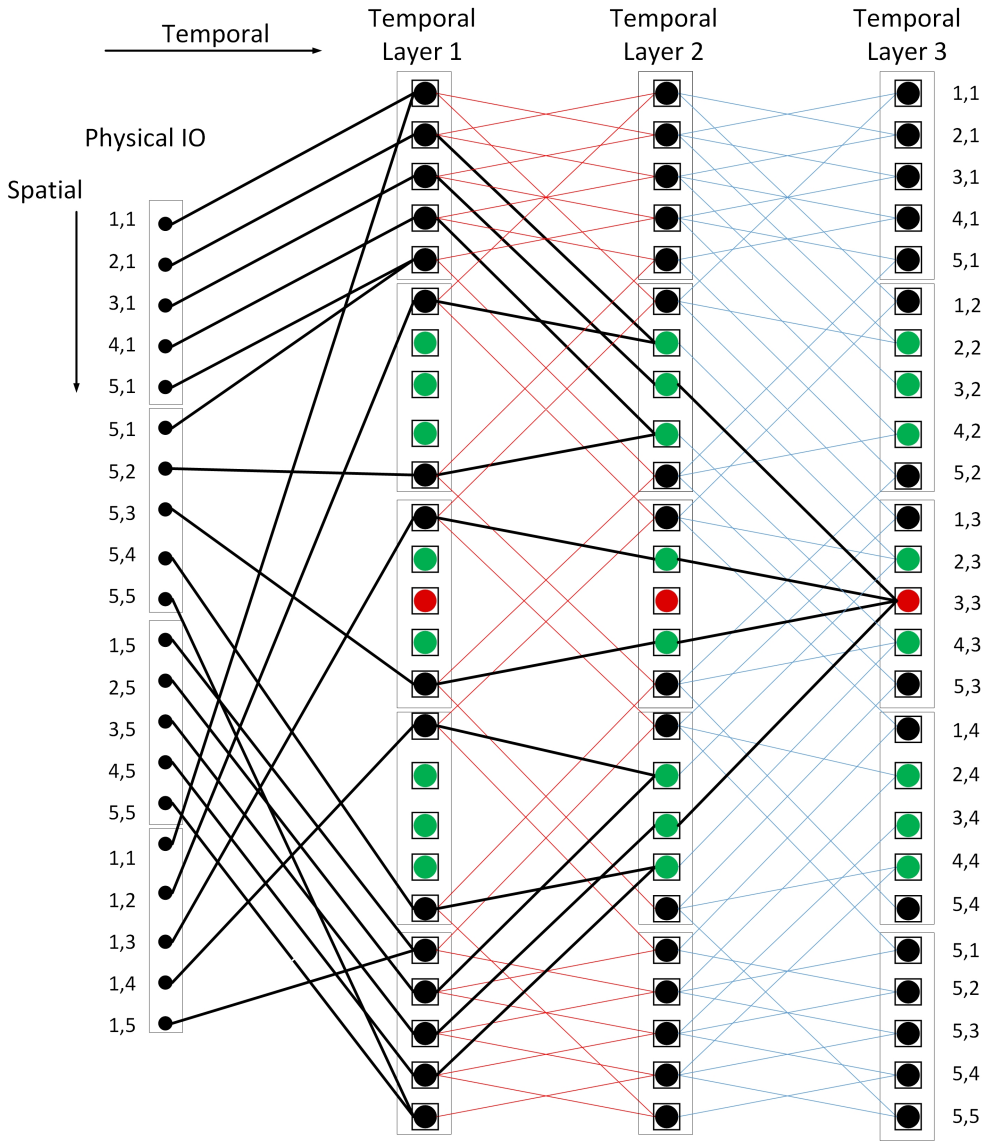


Figure 5.15: 1D spatial and temporal representation of the mesh-based IO network for RTRA.

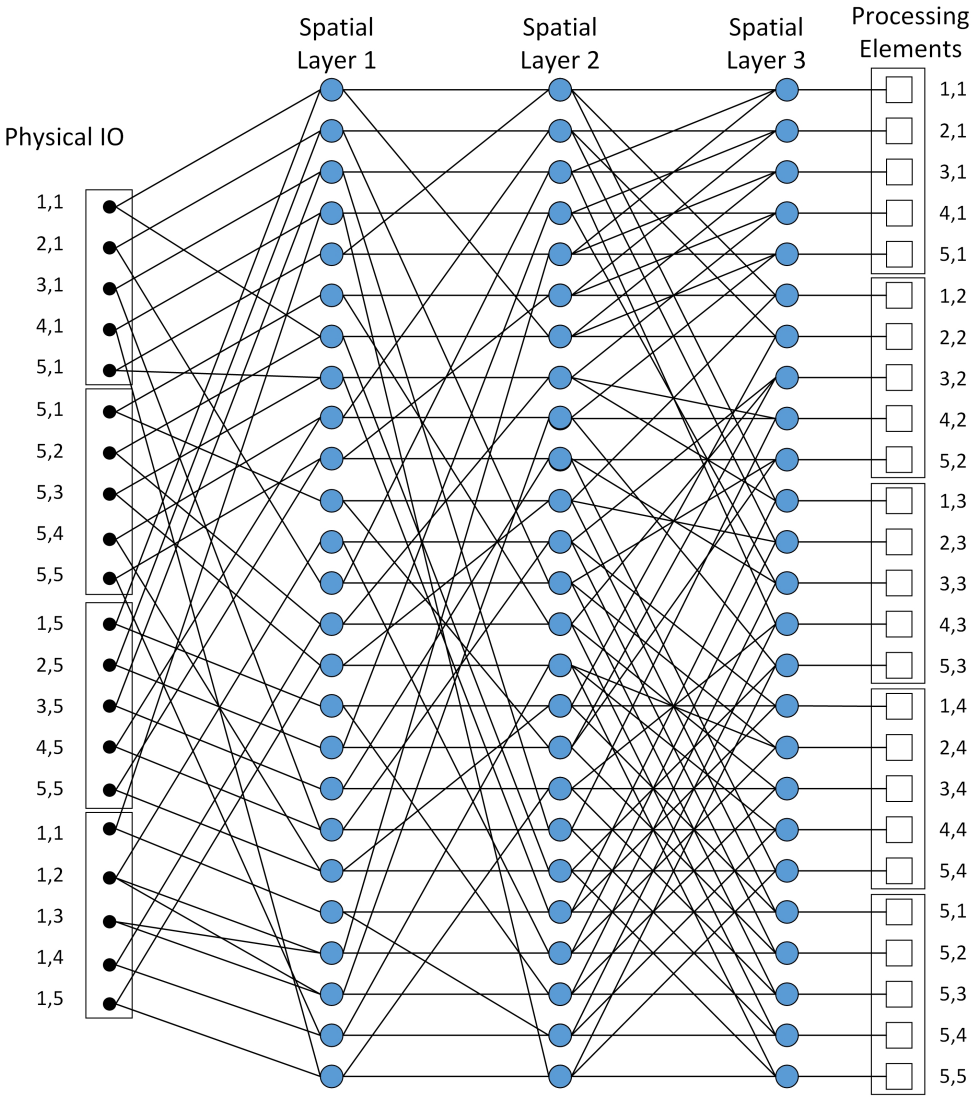


Figure 5.16: Sparse-switchbox based IO network for RTRA.

the network. Poorly optimized programs can have low compute-to-IO ratio which makes the array network limited [25], [51], since the peripheral IO of an array scales linearly with the dimensions of the array while compute scales as its square. The onus is on the compiler to maintain high compute-to-IO ratio for each compiled program for larger arrays to ensure optimal and balanced resource utilization.

5.3 Prior Architectures

Researchers have explored various different software, hardware and domain-specific techniques and policies for array hardware resource management, and improve throughput. There are methods for runtime resource management during the compilation stage, where the user can plan ahead for runtime program dynamics and accommodate them into the program at the time of compilation. Dynamic Partial Reconfiguration (DPR) for FPGA falls under the aforementioned category. DPR can potentially address the problem of dynamic resource allocation [53], by allowing the hardware array to be partitioned and used by different programs or different sections of the same program. Using DPR the user can identify various different small programs which can be compiled together to execute simultaneously or, the user could partition larger program into smaller sections which can get mapped onto the selected array resources, thus reducing the hardware requirements and allowing a larger program to be mapped onto a smaller array. The process of designing and floorplanning programs using DPR is very cumbersome and labor intensive [54], it relies upon the expertise of the user to plan ahead to manage runtime resources at the compile time. The tools for automated floorplanning and DPR have been developed, but the solutions only cover a few steps of the whole process and are limited by the intricacies of the underlying micro-architecture [42], [7], [24]. In HPC cloud environment, software frameworks to virtualize and manage the FPGA resources have been proposed [58], [59]. However, they do not support partial reconfiguration and are not ideal for workloads involving fine-gain runtime dynamics.

There are implementations such as PRR-PRR [9] which limit the number of possible

resource relocations and eliminate the task of floorplanning by the user. PRR-PRR hardware partitions a large FPGA into four identical FPGAs, and every compiled program is designed to reside in one of the smaller FPGAs. At runtime the incoming program can be assigned to any of the available smaller FPGAs. Therefore, the FPGA can be virtualized to support more than one program at any given time on a single device. However the time required to reconfigure an FPGA for virtualizing operations makes them impractical for real applications [39].

Domain-specific accelerators like TPUv4i [22], TSP [2] and Planaria [14] enable multiprogram tenancy and scheduling. However, these architectures are designed for AI/ML applications and rely on domain-specific architecture and description for resource management, which limits their scope for general purpose use.

For general purpose dataflow, architectures such as TRIPS [41], WaveScalar [47] and RAW [3]. These architectures allow for dynamic reconfiguration of array resources based on program phase, in a systolic fashion. The flexibility in these architectures comes at the cost of efficiency. WaveScalar [47] and RAW [3] architectures schedule their instructions dynamically and perform tag matching and operand queue scheduling. Other architectures like Tartan [32] uses a more efficient statically scheduled array, but prohibits reconfiguration during program execution. Some architectures such as VEAL [8] enable a software/driver level virtualization to enable program binary compatibility across different architecture implementations. However the software level support limits the accelerator resource sharing to just one program at a time, and multiple programs cannot simultaneously co-execute on the hardware.

Polymorphic Pipeline Array (PPA) [37], is a reconfigurable architecture that is similar to our approach of solving hardware virtualization. It allows a program to be accommodated and relocated to the resources available at runtime. The program accommodation is achieved by modifying the program binary, folding or unfolding it at runtime in a "virtualized modulo scheduling" fashion. This techniques relies on fixed-latency memory access at every position on the array, which is an unfair assumption for larger array sizes, thus making scaling for PPA

for larger arrays impractical. Additionally the presence of dynamic instruction scheduling at finer, processing element level means that the modulo scheduling requires increasingly higher time to perform program modification for larger program sizes, and might be impractical for larger programs.

5.4 Evaluation Methodology

The Runtime Reconfigurable Architecture (RTRA) has several features that help it improve throughput and performance relative to other architectures. Features such as runtime hardware compiler and hardware scheduler for program relocation, multiprogram tenancy, high programming bandwidth and dynamic program execution. In order to highlight each of the features independently and also highlight their contributions to the overall throughput gains of RTRA we designed a few example architectures each with different subset of RTRA's features, and then compare their throughput and workings. We compare the following architectures to evaluate RTRA :

- Statically configured array: We design and simulate a 18×18 array, it is a typical reconfigurable array architecture, where the array is statically configured to perform accelerator operations. This architecture fully relies on the knowledge of runtime program execution statics at the compile time. As shown in Figure 5.17, if at compile time the user has information pertaining to simultaneously executing programs, then they can co-compile and place all the subprograms together on the array to generate a single program binary for their execution. Since all of the programs share the resources of the array, each program is mapped sub-optimally as a compromise to map them all onto array's resources.

- Dynamically configured array: We design and simulate a 18×18 dynamic array architecture that leverages the high programming bandwidth and dynamic program execution to achieve a higher throughput and active utilization as compared to the statically configured array. The high programming bandwidth and dynamic program execution

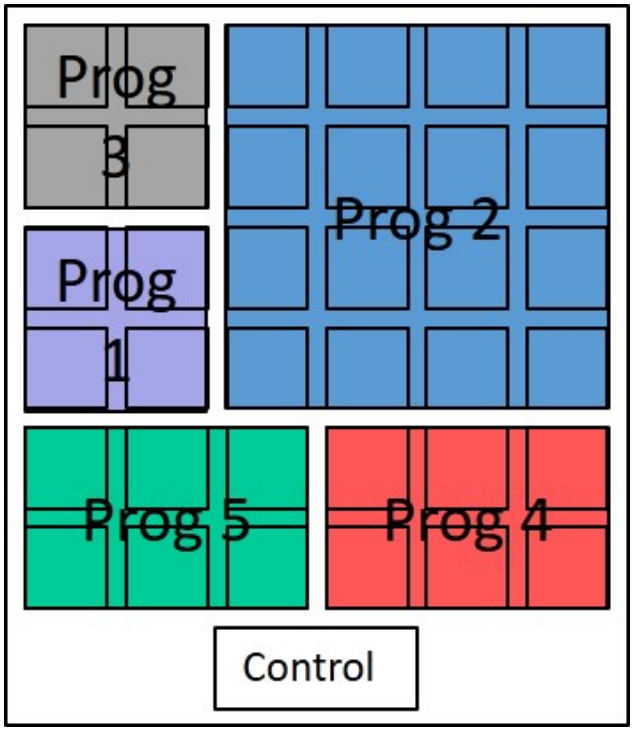


Figure 5.17: Statically configured array programmed to execute programs 1-5 simultaneously.

allows the dynamic array to allocate maximum resources to a single program and map the largest feasible program size to allow for lowest program execution times and thus higher throughput. The array is programmed to perform a single program in succession, thus the programs execute one a time over time, and new program is loaded onto the array once the previous program has finished execution as shown in Figure 5.18.

- Hard-partitioned array: This architecture model is created following the PRR-PRR [9] implementation. We design and simulate a 18×18 array that is hard partitioned as 4 subarrays of size 9×9 each, and the programs are compiled to execute on one of the subarrays. The controller/mapper manages the resources on the array and maps any incoming program request to one of the available subarrays as shown in Figure 5.19. This architecture employs the high programming bandwidth, multiprogram tenancy, dynamic program execution but the scope of hardware scheduler is limited.

Additionally, we devise a new metric to explore the utilization of processing elements,

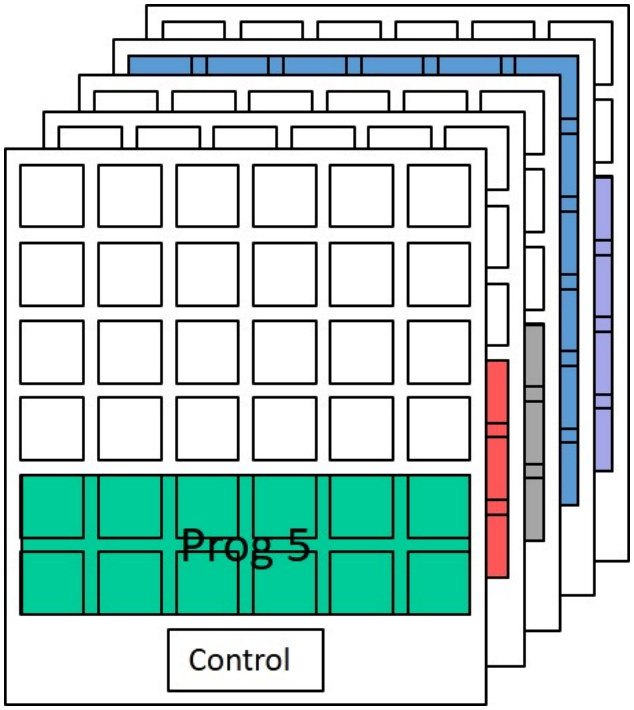


Figure 5.18: Dynamically configured array which reprograms for programs 1-5 over time. Each program is mapped onto the array once the previous program has finished execution.

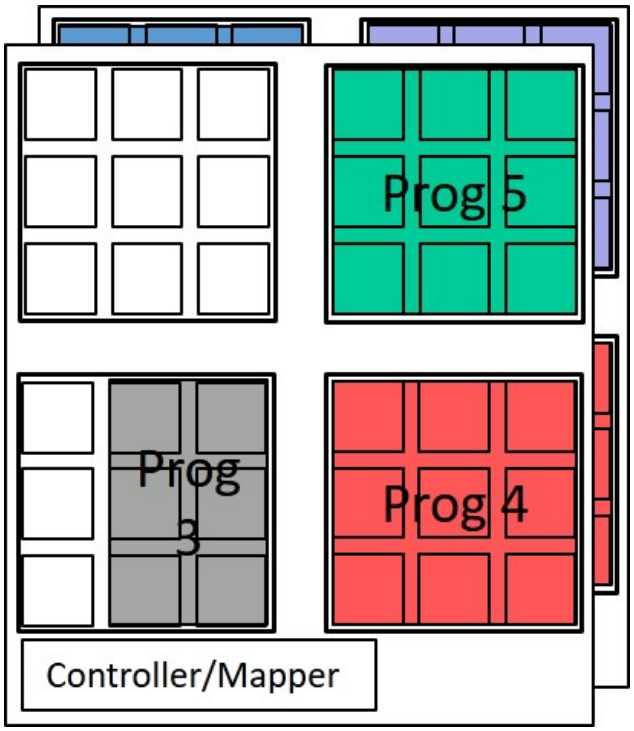


Figure 5.19: A hard-partitioned array which programs the programs 1-5 in its subarrays in the program sequence.

Table 5.1: Example characteristics for subprograms or kernels for a large program.

Kernel/subprogram	Size on array (#PEs)	Execution time(ns)
Prog 1	2×2	1,000
Prog 2	4×4	10,000
Prog 3	2×2	20,000
Prog 4	2×3	20,000
Prog 5	2×3	5,000

called active utilization. The metric can be represented as graph of active array utilization over time or as an average active array utilization over the duration of program. Typically the reconfigurable array compilers report an array resource utilization number, which is the physical footprint of the program over the resources of the array. However the program may not use all of those physically allocated resources fully throughout the duration of program execution. Thus, we devise active utilization as a metric to keep track of array resources being actively utilized and not just physically mapped to and/or allocated. As an example to demonstrate the usage of active utilization, we create a large dummy program which comprises of various small kernels and subprograms. Each program kernel has a different physical footprint, and number of Processing Elements (PEs) requirement and requires a different execution time to perform its task, as shown in Table 5.1. If such a program were compiled to be executed on a statically configured array, most of the kernels would lie idle while a few of them are working on their process. In this example scenario although the utilization/occupation of the array is 100%, however its active utilization of resources is quite low, as shown in Figure 5.20. Most of the array even though it is programmed and occupied is not performing any useful calculation. We can minimize such inactivity and resource idling by using a dynamically configured array and only placing the blocks that actively require processing.

Another way to represent and think about the speed up benefits of using RTRA is as shown in Figure 5.21. The program pipeline can only one of the paths based on the

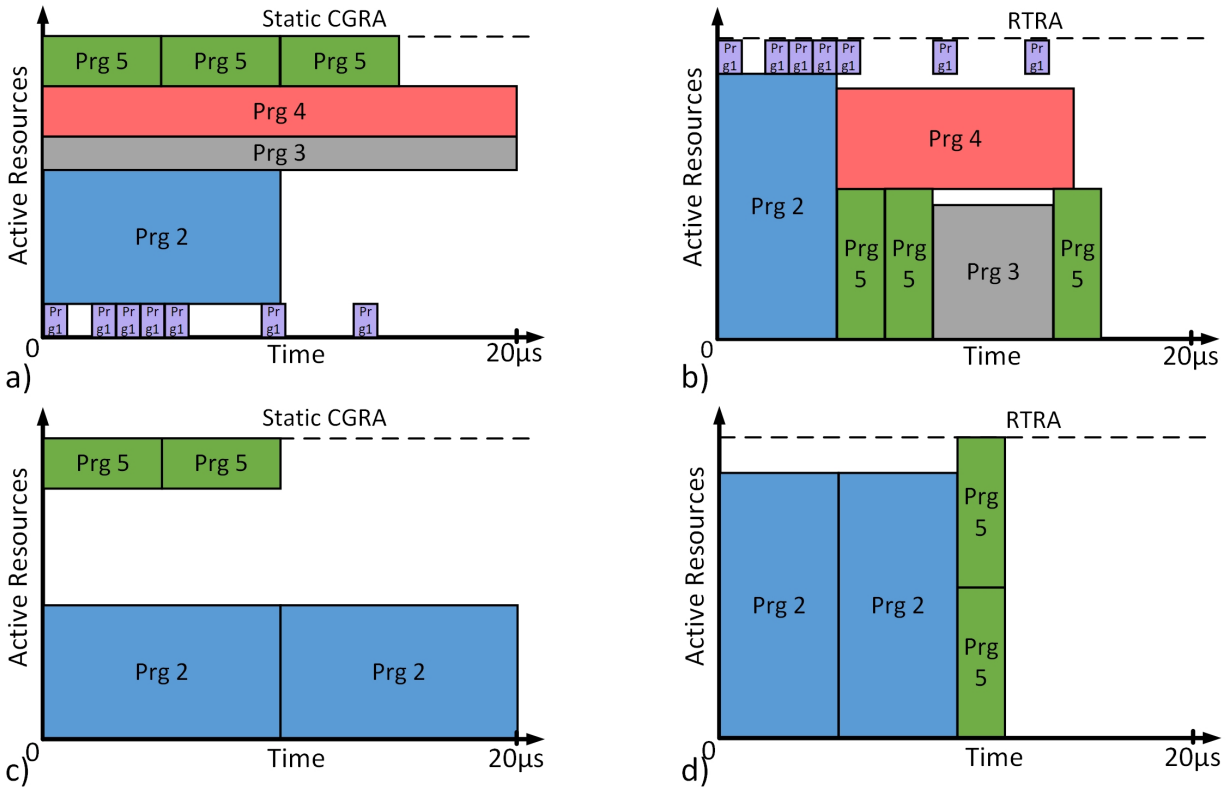


Figure 5.20: a) An example of program requests for multiple programs over time for a static CGRA and its array resource activity. b) The same program requests are mapped onto an RTRA, by using multi-size compile and active resource allcocation and program relocation, RTRA is able to efficiently map the programs on the array leading to higher utilization and lower runtime. c) Example workload representing a scenario where only two program blocks are being requested by the host, a static CGRA has a large inefficiency as the array resources mapped out for other programs cannot be re-purposed for the requested programs. d) The RTRA can handle the same requests from (c) with much higher resource utilization and lower runtime.

decision tree. While the minimum size of a statically configured CGRA is dicatated by the size of the whole program. However, for a static compiled CGRA all of the paths, blocks and kernels need to be statically mapped to accomodate any one of the outcomes of the decisions. Whereas the minimum sized RTRA only needs to handle one of the blocks at a time. Additionally to design an RTRA with equivalent performance to static CGRA, RTRA needs to accommodate only one of the program flow pipelines at a time.

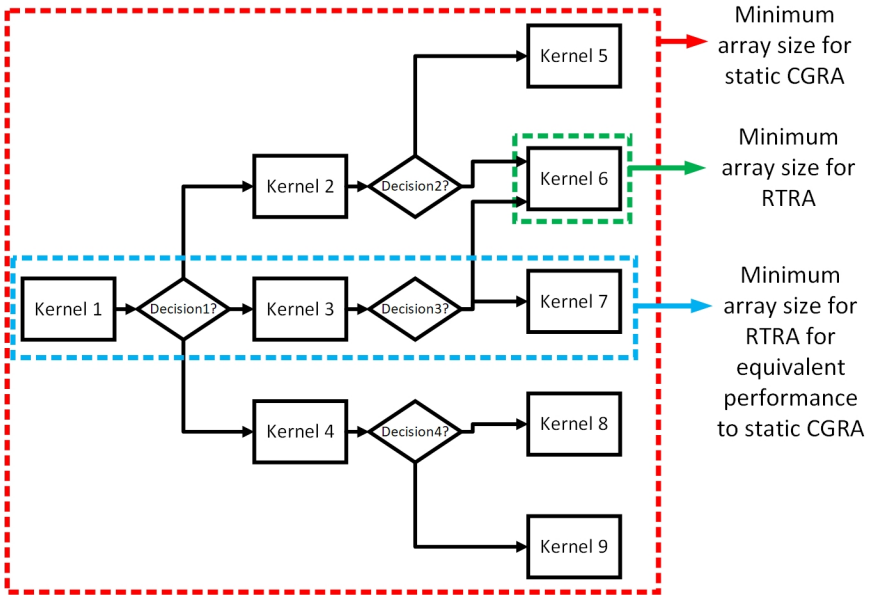


Figure 5.21: An example program flow with multiple kernels or subprograms with various decision points and the size requirements for different implementations of reconfigurable arrays.

5.4.1 Workload Selection

The performance benefits of an actively managed runtime reconfigurable array would be highlighted when executing a workload comprising of multiple independent programs as shown in Figure 5.20 or when executing a large program with multiple decision pipelines and subprograms/kernels as shown in Figure 5.21. In the aforementioned scenario, each independent programs requests the array for a program accelerator and the hardware scheduler arbitrates and services each of those program accelerator requests. Provided with a sufficient program

request pressure, the reconfigurable array would be continuously busy programming and executing. The total time required to fully service a program is a sum of programming and execution time of the program. As we discussed in the compiler Chapter 4, the programming time required for a program depends on the programming bandwidth of the array, which should be constant for any given hardware and the size of array resources required by the program, and the execution time depends on the type of program and inversely proportional to the size of array resources required by the program.

A program workload meant to be run on reconfigurable array can be categorized into 4 broad categories based on their size and execution times as shown in Figure 5.22.

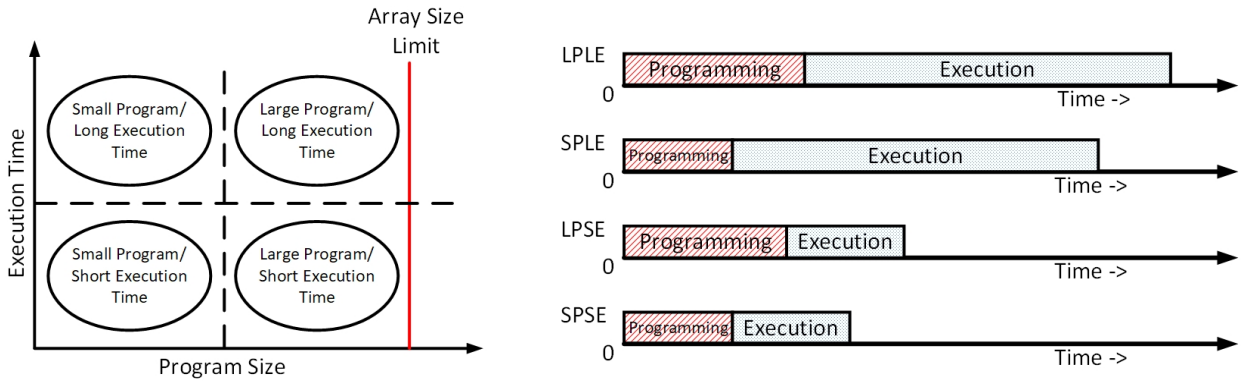


Figure 5.22: Classification of programs into 4 broad categories based on their program size and execution times.

- Large program size, long execution time (LPLE) : This category of programs are large in size and thus require large number of array resources. The size of these programs is comparable to the size of reconfigurable array, and thus the programming time is bound by the upper limit i.e. the time required to program the entire reconfigurable array. While there is no upper bound on the execution times and thus the execution time can be much larger than the programming time of the programs belonging to this category. Since the size of these programs is large, and comparable to the size of array, these programs might prevent other programs from being mapped onto hardware resources and thus hinder multiprogram tenancy.

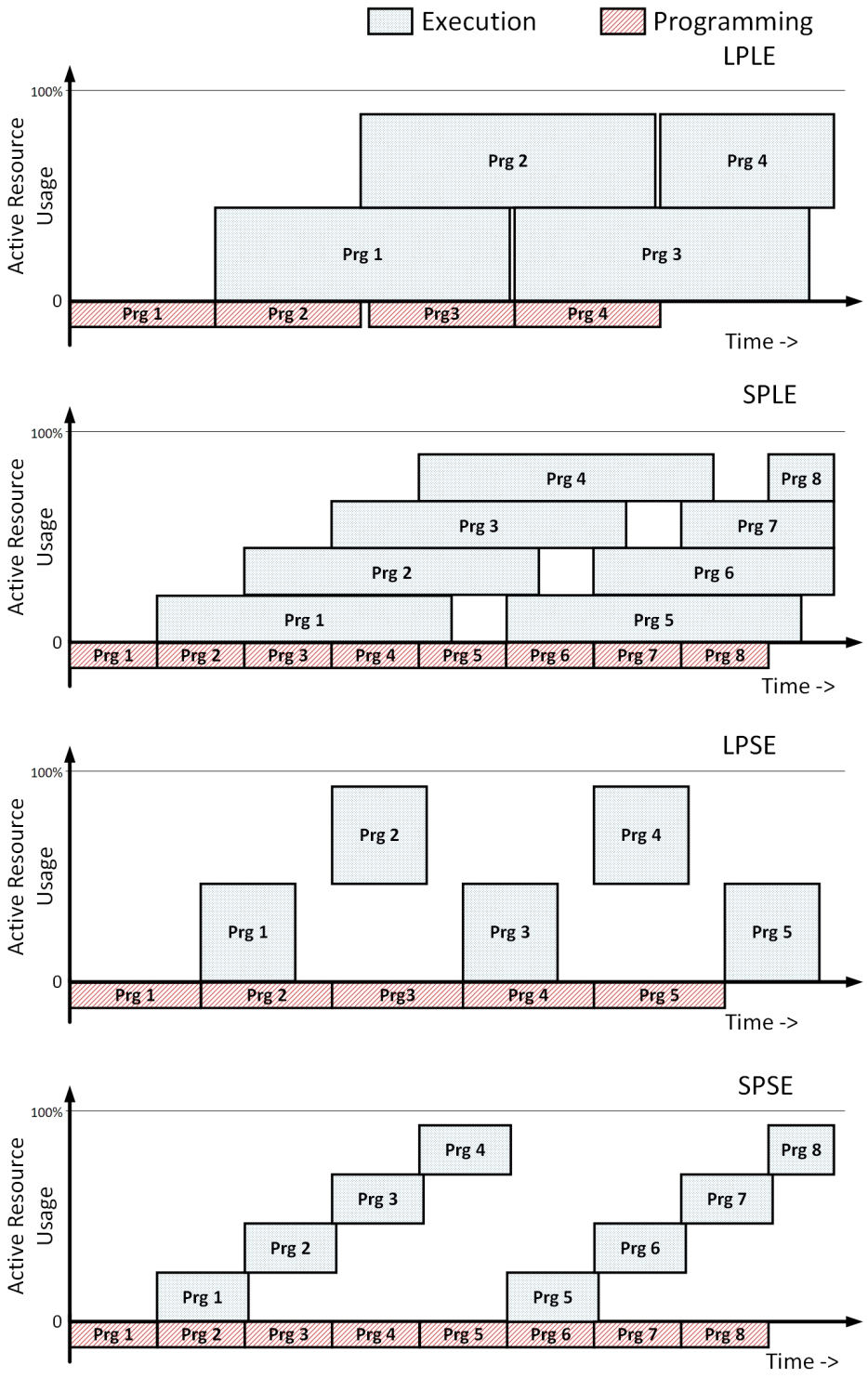


Figure 5.23: Multiprogram tenancy for compiled programs belonging to different categories.

- Small program size, long execution time (SPLE): This category of programs require a few array resources and are thus quick to program but require long execution time. Some unparallelized program algorithms may fall into this category such as Cipher Block Chaining (CBC) based AES encryption [29], or it could also be a sub-optimal implementation of some program. Since the programs are small in size, they require short programming times, and leave aside large vacant array resources this category of programs is ideal for RTRA, as programming time is relatively shorter compared to execution times, so the active utilization of array resources under high programming pressure is also quite high. This topic is discussed in detail in multi-size compile section of Chapter 4.

- Large program size, short execution time (LPSE): This category of programs consists of a few outliers where the program size is large but the execution time of the program is quite low. The programs belonging to this category could one of the few programs which cannot be modified, loop rolled or serialized to execute on smaller array resources. One of the examples could be a 16-pt FFT based on Cooley-Tukey radices, the implementation would require a high number of resources on our array architecture but would only require a few clock cycles to execute. However, a single-path delay feedback FFT could reduce the number of resources required for such an FFT by huge margins, and thus our perception of this category being an outlier. Since the size of these programs is large, and comparable to the size of array, these programs might prevent other programs from being mapped onto hardware resources and thus hinder multiprogram tenancy. However since the execution times are small, the program vacates the hardware resources quickly, thus the wait times for other programs is also shorter.

- Small program size, short execution time (SPSE): This last category of programs are simply small programs which require short execution times. An example could be matrix multiplication operation for smaller matrices. These programs require comparable times to program the reconfigurable array and program execution. Since the scheduler

can program the array for an incoming program while existing programs are executing, having programming size comparable to execution times means that scheduler is unable to map more than just a few active programs on the array. The existing programs would finish their execution while scheduler is programming the incoming program. These programs would benefit a great deal from increased programming bandwidth.

The major point to realise in the program categories is that in all of the above categories the program sizes are relative to the array, which means that a program which is large for a smaller array might become a smaller program for a larger array since these two conditions are completely independent of each other. A corollary to the aforementioned statement is that for a reticle limited extremely large array sizes, any sub-optimized program would fall under the small program, short execution times category which would definitely limit the maximum active utilization of the array, and thus the programming bandwidth for large array sizes should also be proportionally increased to maintain optimal high resource utilization. Figure 5.23 shows the active array resource utilization for programs belonging to different categories, in real world scenario, a workload would consist of programs from multiple categories requesting accelerator access. The figure demonstrates how the program characteristics facilitate or impede active array utilization.

In the scenario where RTRA is connected over the network as shown in Figure 5.5, which can have a lower programming bandwidth as compared to on SoC interconnect network. The programs should be designed and compiled to exclusively fall under the SPLE category, so as to overcome the bottlenecks faced by lower programming bandwidth and ensure high active utilization. As we shall observe in the results section, regardless of the program sizes and program categories, RTRA would perform equivalent to the existing array architectures in extremely sub-optimal program conditions, if not better.

We evaluate the performance of RTRA against the other architectures by designing a workload comprising of :

- BLAS (Basic Linear Algebra Subroutines): We estimate the performance of our ar-

chitecture for BLAS programs by running a workload of assortment of small matrix multiplications and vector dot products. This program set belongs to SPSE category.

- Multi-Layer Perceptron (MLP): We test our architecture for MLP neural networks by estimating its throughput for larger matrix multiplications. This program set belongs to LPLE category.

- Image Processing (IP): We tested our architecture for image processing workloads comprising of applying 2D filters and 2D convolution operations for CNN. This program set belongs to LPLE category.

- Machine Learning (ML): For ML workload we tested the performance of our architecture against an assortment of K-means clustering programs with a varying number of data points and centroids. This program set belongs to SPLE category.

- Media encode/decode: For media decode/encode applications we implemented Discrete Cosine Transform (DCT) and its inverse and tested the performance of our architecture. This program set belongs to LPLE category.

We also map a blind signal classification application [56] on the RTRA, as shown in Figure 4.16 and compare it relative to statically mapped array implementation. The application consists of various programs and kernels which have been compiled and optimized to belong to SPLE and SPSE program categories. Each of the blocks in the blind signal classification DFG requires different execution times and has a different physical footprint and minimum processing elements size requirements, the requirements are tabulated in Table 5.2. The program is able to detect and classify up to 32 concurrent non-overlapping signals up to 125MHz bandwidth in 500MHz signal bandwidth. The number of signals present in the spectrum bandwidth snapshot cannot be predicted in advance at compile time and is a runtime variable. the total number of possible combinations and configurations required to accommodate the pre-planned configurations would render the pre-planned solution impractical to implement. Further the user might have to limit the hardware performance or the number of detectable signals, to design a practical pre-planned solution.

Since preplanning was not feasible, we compared the performance of our RTRA architecture against a statically configured UDSP. The statically configured UDSP has been preprogrammed with all the necessary components and program kernels in advance as shown in Figure 5.24. The host or main thread parses the data from ADCs through the necessary required blocks based on the programs data flow graph. On the other hand the example program execution flow for two single-carrier signals on RTRA is shown in Figure 5.25. In RTRA style of execution the host or main thread places only the required subprogram or kernels on the array with maximum parallelization and array utilization. As observed in Figure 5.25, for two signals detected in the spectrum after the band segmentation block, the two independent blocks of single-carrier classifying kernels are placed and executed on the RTRA, and the scheduler manages the array allocation based on the host programming request and results of the previous blocks. The size of the array 18×18 is derived from the minimum size of array required for the UDSP to accommodate all of the blocks of the blind signal classification pipeline. In addition to the 18×18 RTRA for ACAS, we also tested our program execution flow on a smallest possible RTRA array of size 9×12 which could accommodate the blind signal classification pipeline.

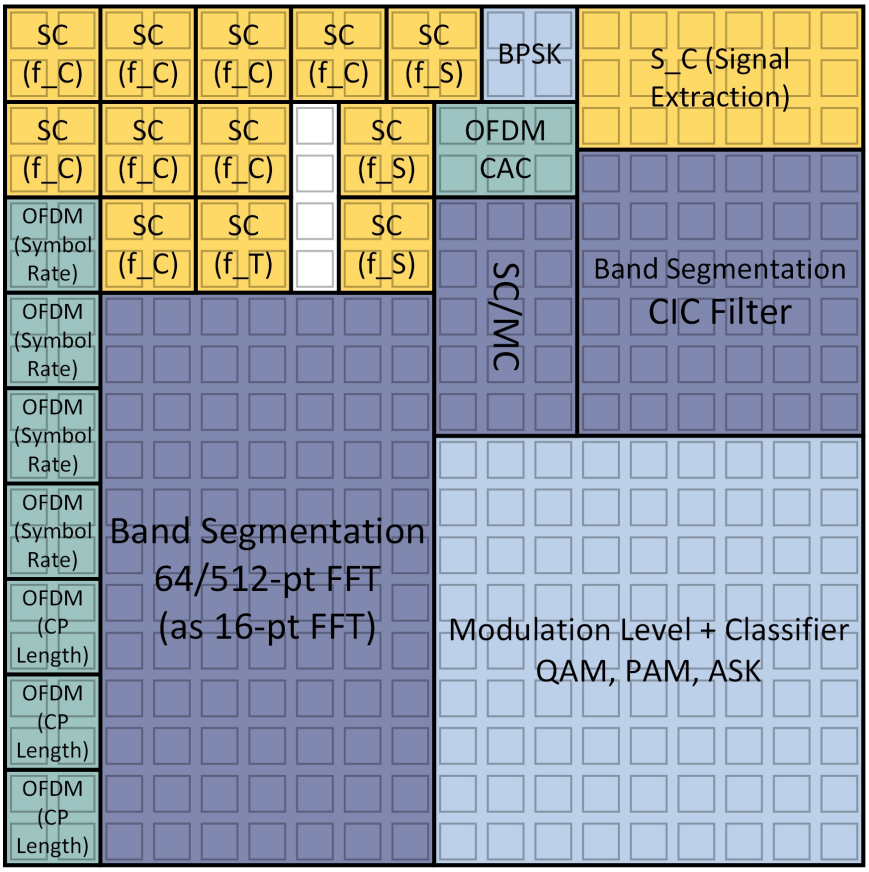


Figure 5.24: Blind signal classification pipeline statically configured onto 18×18 UDSP array.

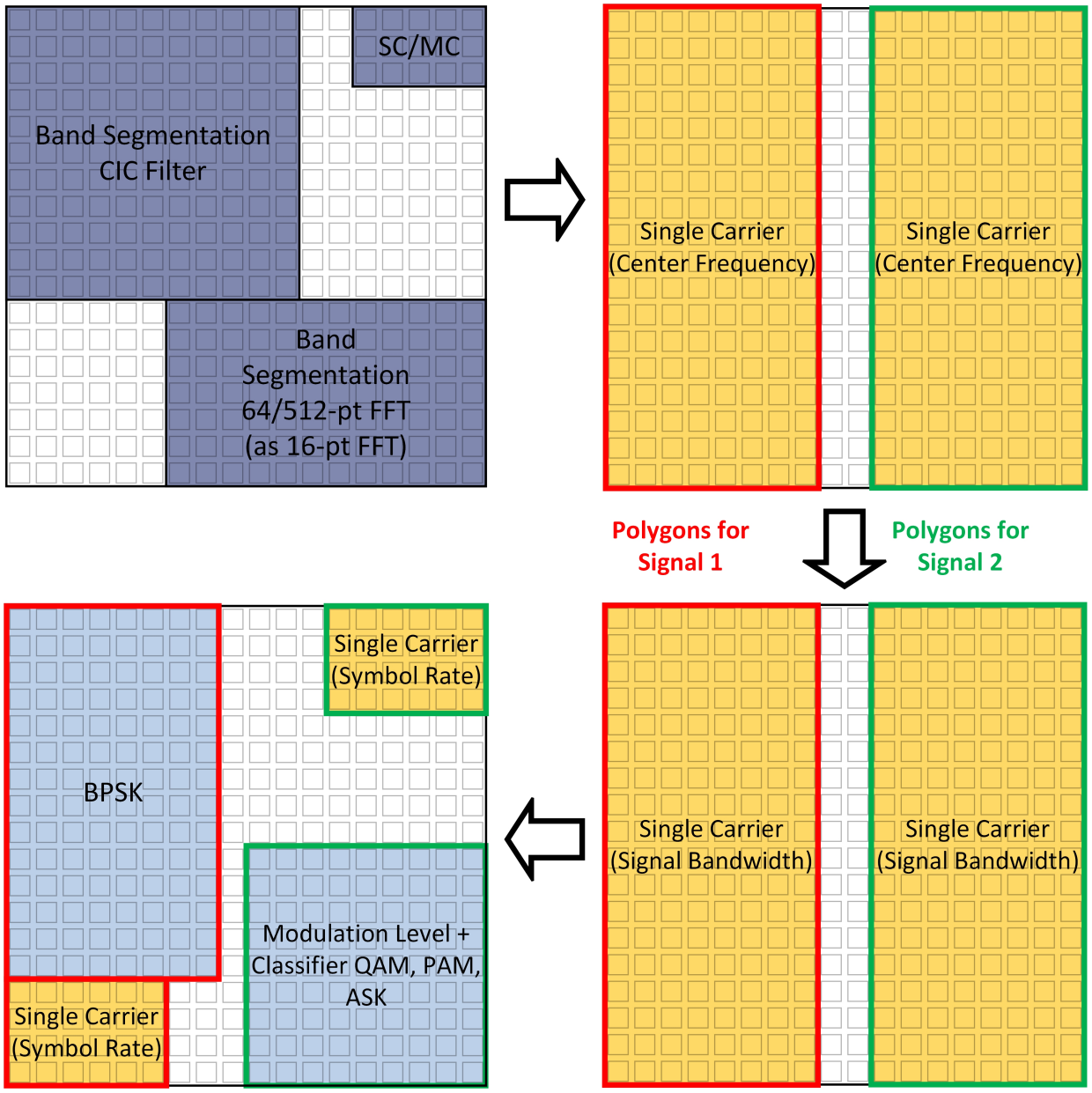


Figure 5.25: Blind signal classification program execution flow for 18×18 RTRA array. The example program flow here shows 2 single-carrier signals being detected in the spectrum snapshot and the subsequent programmings and program blocks on the RTRA.

Table 5.2: Size on reconfigurable array, computation time (@1GHz) for various functions in a blind signal classifier for 0dB SNR.

Function	**Minimum Program Size (PE)**	**Execution Time (ns)**	**Multi-Size Enabled**
BSG CIC filter	6×6	1.2×10^5	Yes
BSG 64pt FFT	7×12	94	Yes
BSG 512pt FFT	7×12	228	Yes
Single carrier/ multi carrier	3×5	100	Yes
OFDM (symbol rate)	2×2	1.6×10^6	Yes
OFDM (cyclic-prefix length)	2×2	7.2×10^5	Yes
OFDM cyclic auto-correlation	2×3	3×10^4	Yes
Single carrier (center frequency)	2×2	1.3×10^8	Yes
Single carrier (signal bandwidth)	2×2	4.3×10^7	Yes
Single carrier (symbol rate)	2×2	5.8×10^4	Yes (upto 6x)
Single carrier (signal extraction)	3×6	20	Yes
Modulation level + classifier (BPSK)	2×2	1.8×10^5	Yes
Modulation level + classifier (QAM/PAM/ASK)	9×9	1,000	No

5.5 Results and Discussion

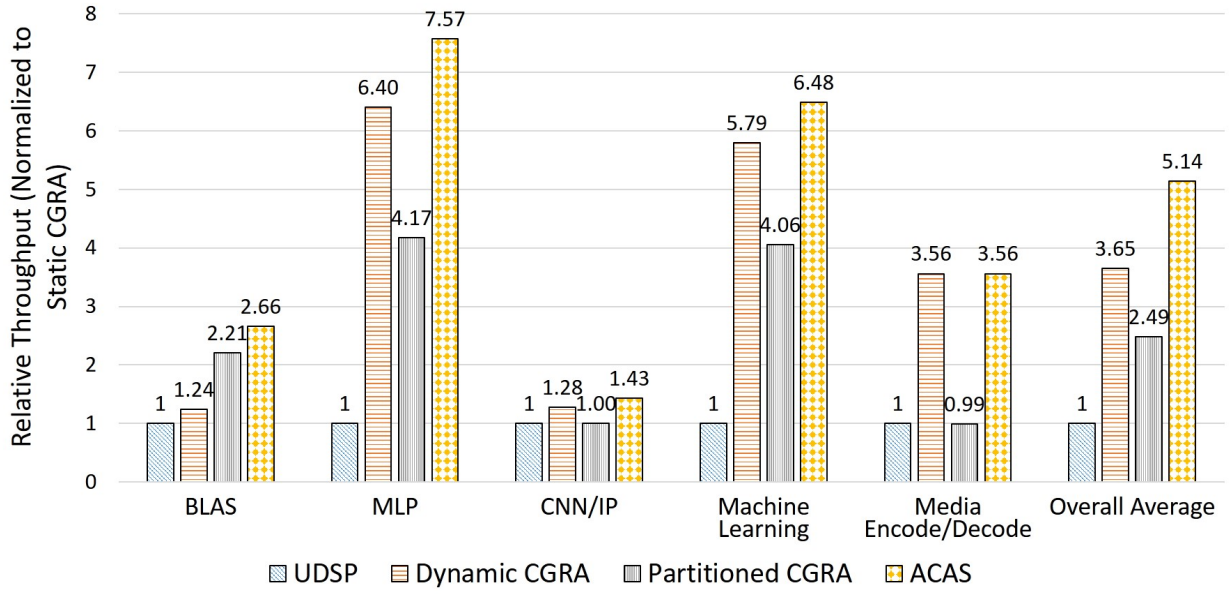


Figure 5.26: The relative throughput of static CGRA, dynamic CGRA, hard-partitioned CGRA and RTRA normalized to static CGRA.

The ACAS model helps increase the throughput of an architecture by increasing the active utilization of the array. The workloads mentioned in previous workload choice section were simulated and tested on the static CGRA, dynamic CGRA, hard-partitioned CGRA and RTRA architectures. The results of the assortment of workloads is as shown in Figure 5.26. As can be observed overall on average the RTRA architecture performs over 5 times better than the baseline statically programmed CGRA architecture and the various features of RTRA combined helps it perform the best overall. The performance benefits of the RTRA vary widely as compared to the other architectures. The cause behind the varying performance is dependent on the sizes of the original programs and the total active usage of the resources. As can be observed in the Figure 5.27, the RTRA architecture performance is much better compared to the dynamic CGRA for program workloads with low active utilization. Workloads such as BLAS, and multi-layer perceptron have a very low active utilization for dynamic CGRA, while the RTRA with its dynamic program composition and relocation

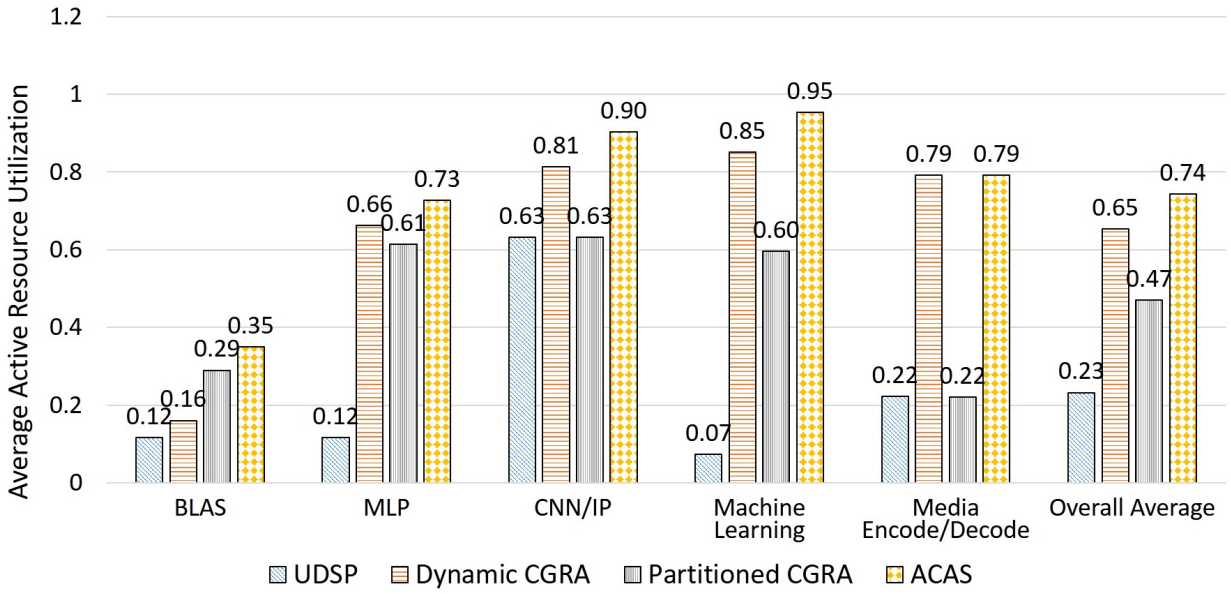


Figure 5.27: Analyzing the average active resource utilization of static CGRA, dynamic CGRA, hard-partitioned CGRA and RTRA for different workloads.

is able to achieve a much higher active utilization and thus higher performance. Where as for workloads such as video encode/decode, the program has a much higher actively utilization, additionally the program kernel is quite large, which limits the maximum parallelization that can be achieved using the array, and leaves a substantial 20% array resources un-utilized. In these array limited scenarios, with maximum array utilization, RTRA is still able to match the performance of dynamic CGRA. The hard-partitioned CGRA performs quite well for small BLAS applications, the partitions allow it to accomodate multiple program kernels simultaneously, but its upper bound by the number of hard-partitions while RTRA does not have any such limitation, thus performs better. For larger workloads the hard-partitions interfere with the amount of parallelization that can be achieved and can lead to under-utilized array, and thus poor throughput for hard-partitioned CGRA.

RTRA was able to exploit the multi-size compile and multi-step compilation to achieve higher throughput and higher active utilization for blind signal classification workloads. As shown in Figure 5.28, RTRA ACAS system is able to complete the signal detection pipeline for a variety of detected signal scenarios much faster than the statically configured UDSP.

As shown in Figure 5.29, the RTRA of size 18×18 as well as 9×12 outperform the statically configured 18×18 UDSP array. The high parallelization and active utilization of the array as shown in Figure 5.30 of the ACAS model on RTRA as compared to UDSP helps it achieve a much higher throughput. Intuitively we can also observe that the larger resource blocks from Table 5.2, achieve a higher active utilization on UDSP and the blocks with high resource utilization and short execution (LPSE) category are not parallelized in RTRA architecture as they limit the active utilization and hurt the overall performance.

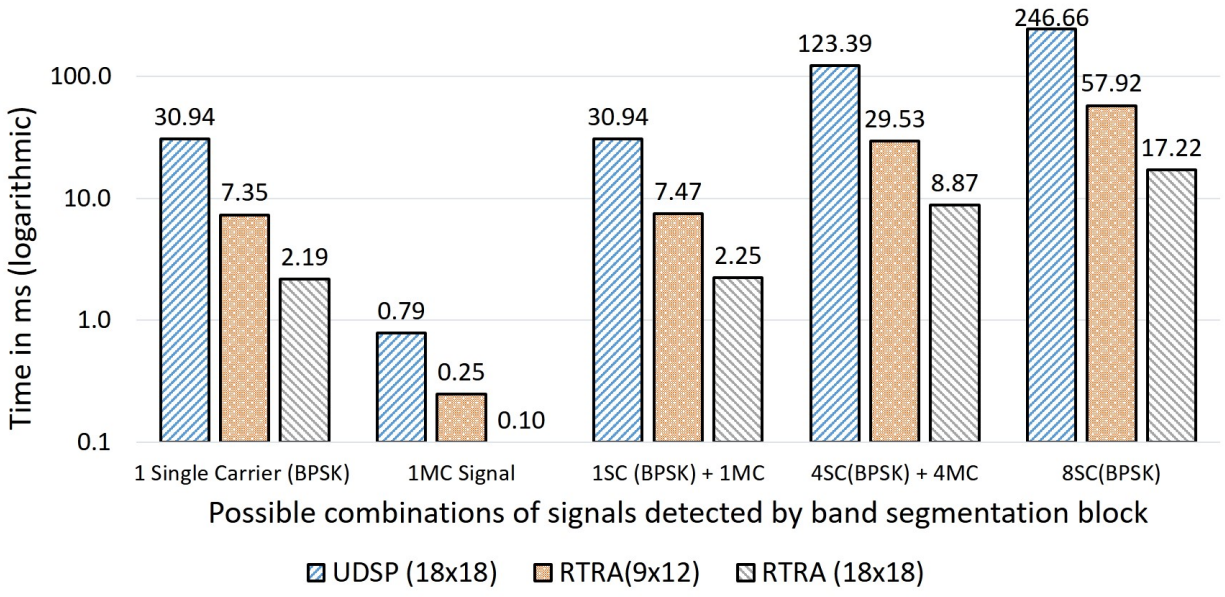


Figure 5.28: Signal classification time required by the statically configured UDSP architecture, in comparison to the 9×12 RTRA and 18×18 RTRA for various combinations of input signals detected in the incoming spectrum snapshot.

Figure 5.31 shows the number of clock cycles (@1GHz) required by runtime hardware scheduler and hardware compiler to find appropriate vacant resources on the array to map the incoming program for bookkeeping memory for 10 active programs and 32 anchor points. The polygon-based edge intersection method to detect program overlap along with the anchor point bookkeeping approach to keep track of vacant resources on the array, helps scheduler achieve a low time-to-map latency. The special programming frame and symmetric mesh-based interconnect help speed up the process of mapping incoming programs to available

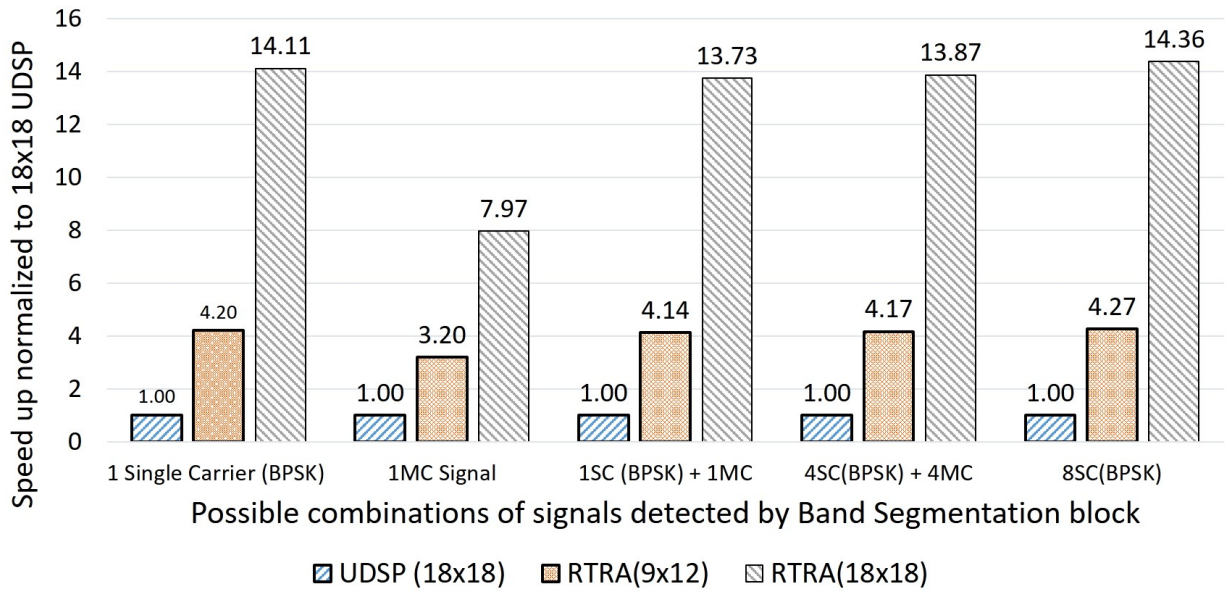


Figure 5.29: Speed up of blind signal classifier workloads on a 9×12 and 18×18 RTRA, normalized with respect to 18×18 UDSP.

resources.

5.6 Array Size Scaling

In this section we explore the effect of array size and programming bandwidth on the performance benefit of dynamic actively managed architecture RTRA vs statically configured UDSP. As we observed in the results section, the major performance benefit of RTRA comes from the opportunistic parallelism that it can exploit by only placing the active section/kernel of the program on to the array resources, whereas UDSP has to content with the array resource statically divided amongst various program kernels, even the ones which are inactive due to the selected pipeline, or inactive because of unavailability of data to process. Thus, intuitively increasing the size of array should improve the performance of RTRA but the larger array should improve UDSP as well. The extent of benefits of two architecture might be different and that is what we seek to explore in this section.

Increasing the size of array improves the execution time required for the program, however

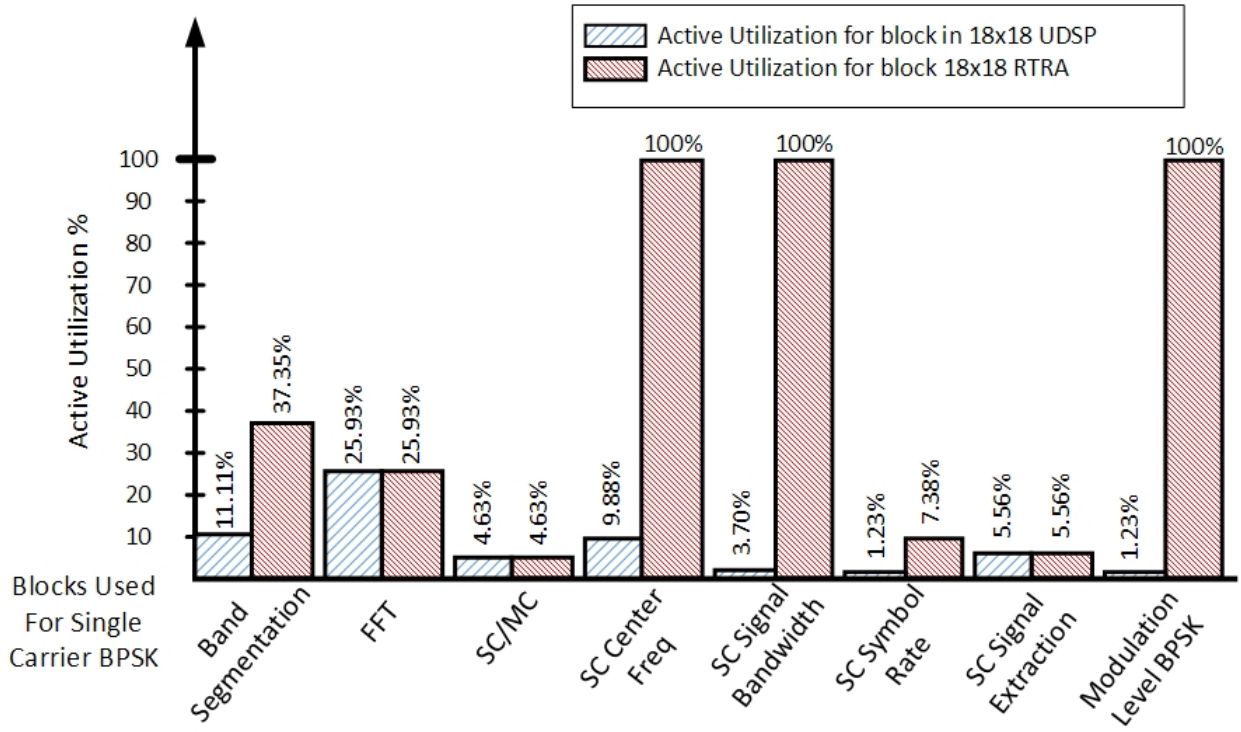


Figure 5.30: Active utilization for UDSP and RTRA at different stages of execution for one single-carrier BPSK signal classification pipeline.

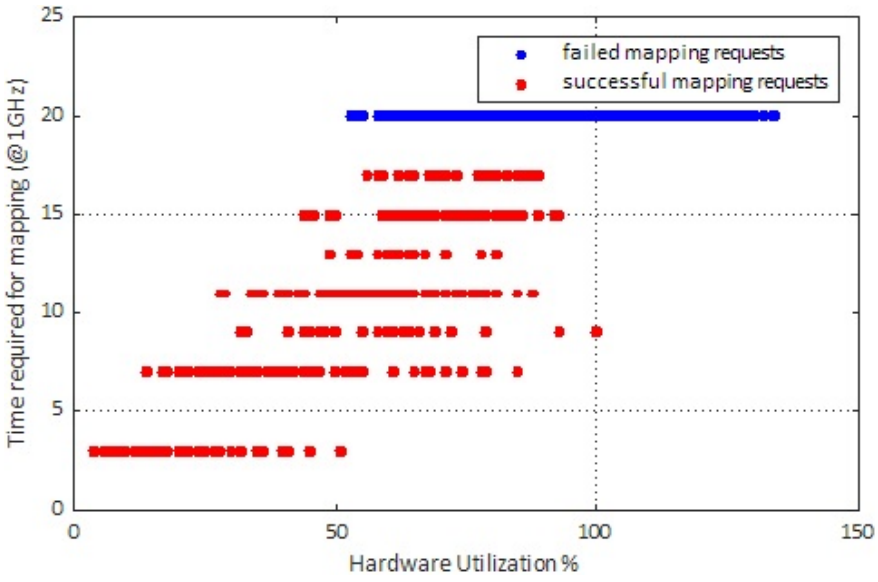


Figure 5.31: Time required (@1GHz clock) to find appropriate vacant resources on the array and map the incoming program varies based on hardware utilization with the number of actively executing programs on the array.

for RTRA the programming time required increases with the size of program as well, so there is a point of diminishing return and negative return beyond which increasing the resource allocation for the program negatively impacts RTRA performance as seen in Figure 4.19, however UDSP does not suffer with such negative performance issues. Additionally, we observe that different algorithm kernels within the same program could have different optimal parallelization points. A program section with large execution times, could optimally utilize the larger array resources, however a kernel with shorter execution time would have it optimal parallel operating point at much smaller array resources, thus would inefficiently occupy the array resources.

As an example exercise we calculated the optimal maximum parallel points of various kernels in the blind signal classification kernel and calculated the time required to compute each of the kernels independently for various different programming bandwidths, as shown in Table 5.3 for 1Gbps programming bandwidth, Table 5.4 for 20Gbps programming bandwidth and Table 5.5 for 200Gbps programming bandwidth. As we can notice that for single-carrier

Table 5.3: Maximum optimal parallelization for various functions in a blind signal classifier for 0dB SNR at 1Gbps programming bandwidth.

Function	**Minimum Program Size**	**Execution Time (ns)**	**Max Parallel (Size)**	**Max Parallel Total Time (ns)**
BSG CIC filter	6×6	1.2×10^5	$3(9 \times 12)$	8.1×10^4
BSG 64pt FFT	7×12	94	$1(7 \times 12)$	3.2×10^4
BSG 512pt FFT	7×12	228	$1(7 \times 12)$	3.2×10^4
Single carrier/ multi carrier	3×5	100	$1(3 \times 5)$	5,860
OFDM (symbol rate)	2×2	1.6×10^6	$32(8 \times 16)$	9.9×10^4
OFDM (cyclic-prefix length)	2×2	7.2×10^5	$22(10 \times 10)$	6.6×10^4
OFDM cyclic auto-correlation	2×3	3×10^4	$4(4 \times 6)$	1.67×10^4
Single carrier (center frequency)	2×2	1.3×10^8	$291(34 \times 34)$	8.93×10^5
Single carrier (signal bandwidth)	2×2	4.3×10^7	$166(26 \times 26)$	5.1×10^5
Single carrier (symbol rate)	2×2	5.8×10^4	$5(4 \times 6)$ (upto 6x)	1.92×10^4
Single carrier (signal extraction)	3×6	20	$1(3 \times 6)$	6,932
Modulation level + classifier (BPSK)	2×2	1.8×10^5	$11(6 \times 8)$	3.3×10^4
Modulation level + classifier (QAM/PAM/ASK)	9×9	1,000	$1(9 \times 9)$	3.2×10^4

Table 5.4: Maximum optimal parallelization for various functions in a blind signal classifier for 0dB SNR at 20Gbps programming bandwidth.

Function	**Minimum Program Size**	**Execution Time (ns)**	**Max Parallel (Size)**	**Max Parallel Total Time (ns)**
BSG CIC filter	6×6	1.2×10^5	$12(24 \times 18)$	1.8×10^4
BSG 64pt FFT	7×12	94	$1(7 \times 12)$	1,774
BSG 512pt FFT	7×12	228	$1(7 \times 12)$	1,908
Single carrier/ multi carrier	3×5	100	$1(3 \times 5)$	388
OFDM (symbol rate)	2×2	1.6×10^6	$144(24 \times 24)$	2.2×10^4
OFDM (cyclic-prefix length)	2×2	7.2×10^5	$98(14 \times 28)$	1.48×10^4
OFDM cyclic auto-correlation	2×3	3×10^4	$16(8 \times 12)$	3,700
Single carrier (center frequency)	2×2	1.3×10^8	$1296(72 \times 72)$	2×10^5
Single carrier (signal bandwidth)	2×2	4.3×10^7	$756(54 \times 56)$	1.1×10^5
Single carrier (symbol rate)	2×2	5.8×10^4	$6(4 \times 6)$ (upto 6x)	1×10^4
Single carrier (signal extraction)	3×6	20	$1(3 \times 6)$	365
Modulation level + classifier (BPSK)	2×2	1.8×10^5	$48(8 \times 24)$	7,436
Modulation level + classifier (QAM/PAM/ASK)	9×9	1,000	$1(9 \times 9)$	2,555

Table 5.5: Maximum optimal parallelizatoin for various functions in a blind signal classifier for 0dB SNR at 200Gbps programming bandwidth.

Function	**Minimum Program Size**	**Execution Time (ns)**	**Max Parallel (Size)**	**Max Parallel Total Time (ns)**
BSG CIC filter	6×6	1.2×10^5	$42(36 \times 72)$	5,670
BSG 64pt FFT	7×12	94	$1(7 \times 12)$	262
BSG 512pt FFT	7×12	228	$1(7 \times 12)$	396
Single carrier/ multi carrier	3×5	100	$2(6 \times 5)$	110
OFDM (symbol rate)	2×2	1.6×10^6	$442(42 \times 42)$	7,100
OFDM (cyclic-prefix length)	2×2	7.2×10^5	$300(30 \times 40)$	4,800
OFDM cyclic auto-correlation	2×3	3×10^4	$50(15 \times 20)$	1,200
Single carrier (center frequency)	2×2	1.3×10^8	4010 (127×127)	6.4×10^4
Single carrier (signal bandwidth)	2×2	4.3×10^7	$2290(96 \times 96)$	3.7×10^4
Single carrier (symbol rate)	2×2	5.8×10^4	$6(4 \times 6)$ (upto 6x)	1×10^4
Single carrier (signal extraction)	3×6	20	$1(3 \times 6)$	56
Modulation level + classifier (BPSK)	2×2	1.8×10^5	$150(20 \times 30)$	2,400
Modulation level + classifier (QAM/PAM/ASK)	9×9	1,000	$3(9 \times 27)$	820

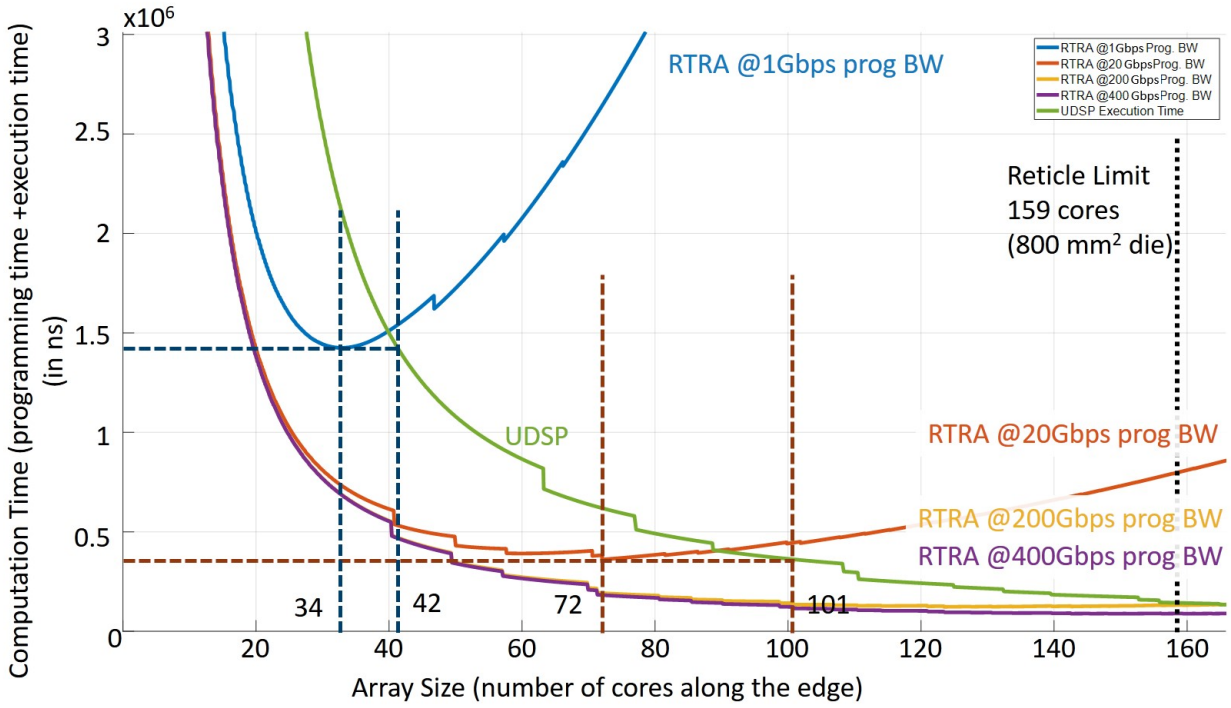


Figure 5.32: Single-carrier blind signal classification computation time for RTRA (various bandwidths) and UDSP for varying array sizes.

classification pipeline, the Single-Carrier (Center Frequency) kernel is the most computation intensive section of the whole classification pipeline, it occupies the most array resources as well as requires the most total time. So, for increasing array sizes the SC(FC) kernel would be able to occupy all of the array resources and ensure maximum active utilization, however kernels such as Single-Carrier (Symbol Rate) with small array utilization of just 4×6 would undermine the active utilization of array. These effects cause the RTRA vs UDSP throughput gains to diminish for large array sizes, however these effects can be counteracted by increase in programming bandwidth. Figure 5.32, shows the effect of array size scaling on RTRA various bandwidths and UDSP. We can observe that for the same throughput at the best performance points of RTRA we require larger array size for UDSP as compared to RTRA. However when the array size is increased beyond the optimal operating point of RTRA for a given program, then the programming time for RTRA is detriment to its performance and leads to sub-optimal operation, hence RTRA programs should never be executed and compiled beyond the optimal operating points. However, if needed the prorgamming bandwidth of the array should be scaled in tandem with the scaling array sizes, as we can observe RTRA with programming bandwidth 200Gbps stays competitive with UDSP up to the reticle limit of $800mm^2$ which happens at array size 159×159.

In the next subsection we take a look at a simple method to increase the programming bandwidth of the RTRA.

5.6.1 Programming Bandwidth

We observed that increasing programming bandwidth has a great effect on the throughput benefits of RTRA for larger array sizes. Increasing the array size, should increase the number of IO pins coming in to and out of the dielet, so the total aggregate bandwidth can be scaled very easily with scaling array sizes. However, the problem should arise at distributing the increased programming bandwidth effectively and with minimal wiring cost to the processing elements on the array.

Increasing the programming bandwidth of the array requires some special accommoda-

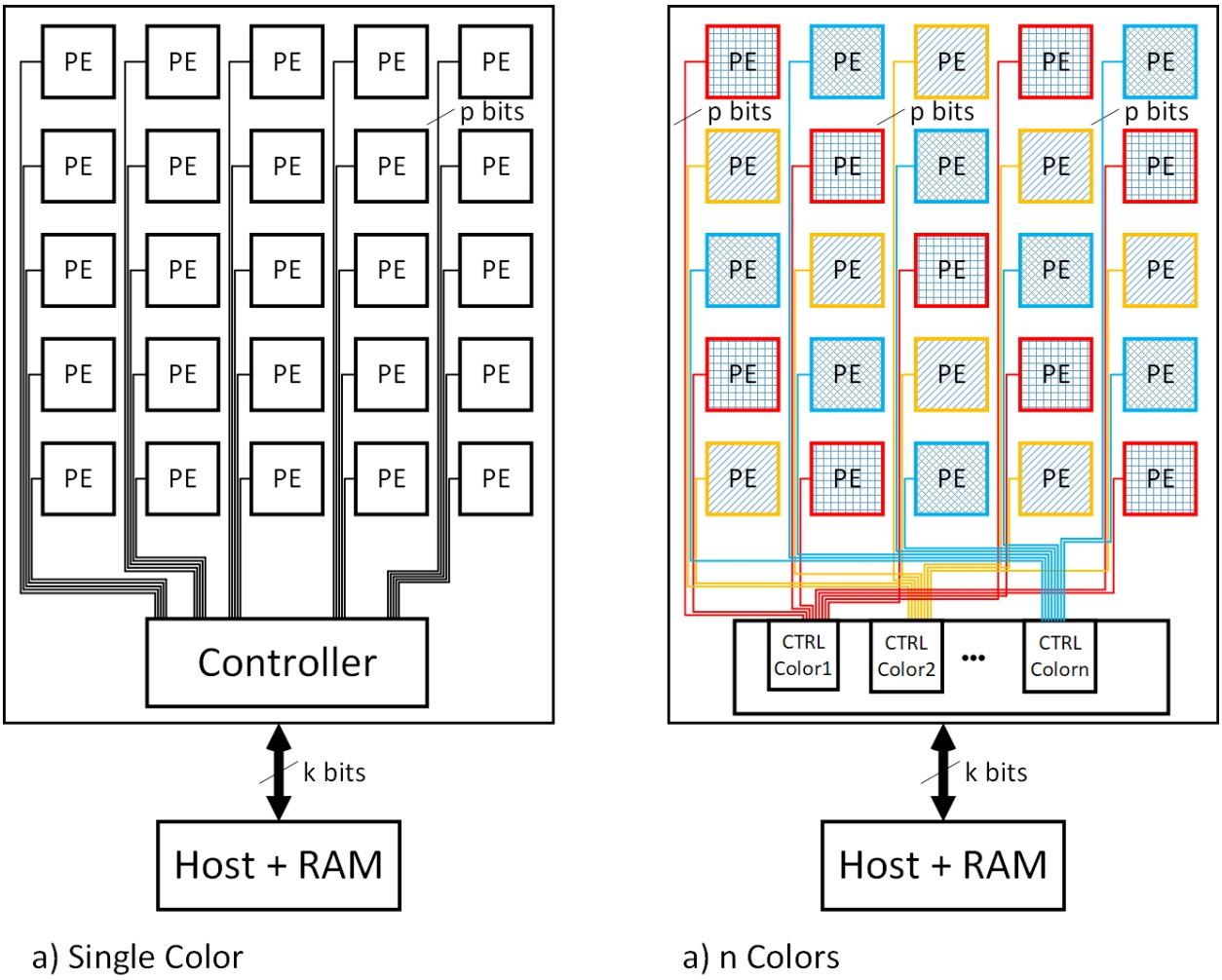


Figure 5.33: A low area- and energy-overhead coloring-based high-bandwidth programming interface.

tions to the control mechanism. In a typical programming scenario with a control module for programming, a p-bit bus at high clock speed, let's assume 'X' GHz runs from the controller to each of the Processing Elements. Since its a point to point connection, for n number of PEs, there would be n connections each 'p' bits wide originating from the control module to each of the PEs. However since there is just one controller/decoder/control module, it can only program one PE at a time thus giving a maximum of '$p \times X$' Gbps programming bandwidth.

We use a graph coloring [26] technique to color the PEs such that no two adjacent PEs have the same color. This method divides the array and the PEs into distinct segments of 'n' colors, and by implementing 'n' distinct programming controllers for each of the colors, we can increase the programming bandwidth while maintaining same number of wire cost for programming but increased controller complexity as shown in Figure 5.33. In this scenario a PE belonging to each of the 'n' colors can be programmed in parallel, and thus the architecture could potentially achieve '$n \times p \times X$' Gbps programming bandwidth. The graph coloring technique allows the color controllers to work in parallel. If we draw any random polygon on the array on top of the PEs, there would be almost same number of PEs of each color, this ensures that independent of the size and shape of the polygon maximum programming bandwidth can be achieved in parallel with all programming controllers active.

Increasing the programming bandwidth helps reduce the time a scheduler has to spend in programming the array resources for the program, which helps reduce the total time for program. However the increased programming bandwidth can be used to readjust the optimal operating point or parallelization of multi-size compile as shown in Figure 5.34. Additionally we show a plot for the condition that the designer can optimize the architecture further and reduce the instruction bits required to configure the processing elements and the reconfigurable array in half, which is an extension of our initial discussion about architecture efficiency metric from Chapter 2, and we can observe the speed up benefits, as well as better resource utilization.

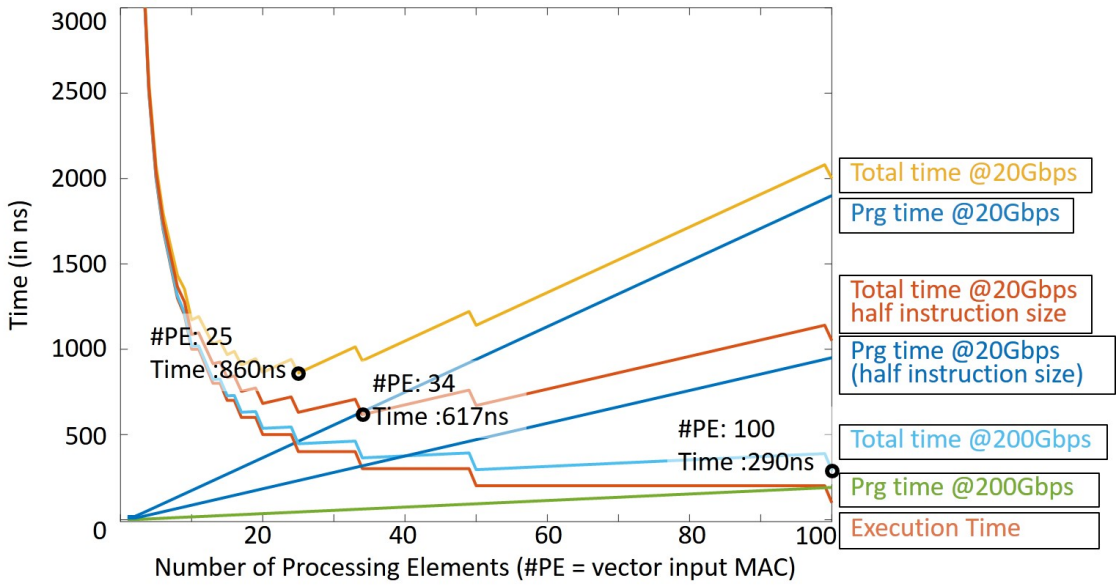


Figure 5.34: Scaling of programming time and total time to compute for the 100×100 matrix multiplication program with changing programming bandwidth and adjusting instruction size.

5.7 Discussion

The actively managed RTRA allows the array to achieve its peak performance and throughput by ensuring high active utilization of the array. The scheduler, hardware compiler consume $60mW$ power at 1GHz and occupy $0.15mm^2$ area, based on post synthesis numbers from TSMC16nm FFC technology. These RTRA array overhead of scheduler and compiler contribute to less than 3% area and 6% power to the baseline hardware implemented 14×14 UDSP die. Additionally, the size of scheduler and hardware compiler is independent of the size of hardware array, so it doesn't scale up with increase in array size. Thus, the impact of RTRA on the peak power and area of the reconfigurable array is minimal, however the increased active utilization helps achieve a higher throughput in the same peak power bracket of the chip. Needless to say, this also means that if an array is used without the active hardware management system its actual power consumption is much lower than it is peak power, since it is underutilized.

The speed up benefits of using RTRA over other architectures can be better understood by observing the Figure 5.20 and Figure 5.21. The synthetic program workloads such as BLAS, multi-layer perceptron (neural networks), image processing, video encode/decode etc. belong to the first category of independently executing programs as shown in Figure 5.20. Each kernel or subprogram for the synthetic workload requests an array access to the hardware scheduler and when granted executes its routines independently with the largest feasible array resources allocated to it by the scheduler. The blind signal classification program falls in the other category of programs as shown in Figure 5.21. There are multiple decision points in the blind signal classification workload while one of the paths is required for any one signal. As we observed even the ACAS 9×12 array with 3 times fewer resources than the reference 18×18 statically configured UDSP array is able to perform $3.2 \times -4\times$ times better. Due to the dynamic programming, active program composition and efficient allocation of resources the ACAS 9×12 model outperforms a larger static configured CGRA. The benefits of RTRA architecture and ACAS model can be used to either reduce the hardware requirements of the reconfigurable array, e.g. 9×12 RTRA vs 18×18 UDSP, or increase the performance or throughput of the reconfigurable array at the same area and resource cost e.g. 18×18 RTRA vs 18×18 UDSP. The 18×18 RTRA is able to perform at least $8\times$ and upto $14\times$ better as compared to the equivalent size 18×18 statically configured UDSP.

CHAPTER 6

Conclusion

This dissertation work aimed to develop an area- and energy-efficient and scalable Runtime Reconfigurable Array solution, which could provide hardware accelerator-like performance and efficiency for domain-specific applications. We began with an analysis of the existing program execution flow techniques of various spatial and temporal architectures such as CPU, GPU, FPGA, and hardware accelerators. Based on the findings, we developed a multi-domain Architectural Efficiency (AE) metric that quantifies the influence of micro-architecture, instruction set, and reconfiguration bits on the throughput, area, and energy efficiency of the final architecture. We ran the AE metric on different aforementioned architectures and gained insights into the importance of domain-specificity, optimal interconnect network, compute specialization, and compute pipeline depth in the design of an optimal low area and energy overhead, yet flexible and high-throughput architecture.

We then developed a coarse-grain reconfigurable array, which we call Universal Digital Signal Processor (UDSP), using the software-hardware co-design features, statistics-based optimization techniques, and sparse switchbox interconnect network to achieve a scalable, high-throughput, area- and energy-efficient design. We explored the computation and connectivity requirements of algorithms belonging to the Digital Signal Processing (DSP) domain, and developed connected-graph models for the domain. We used the derived DSP graph models and our hardware-software co-design technique to develop an interconnect network of optimal connectivity, which meets the requirements of high-throughput applications, while also ensuring high compilability of the reconfigurable architecture. We then developed a method of designing multi-layer sparse switchboxes for the interconnect network

by optimally pruning the connections inside the switchboxes, while retaining a near-perfect average bandwidth and connectivity, and while minimizing the hardware cost. The approach helps us design a switchbox that can achieve 96.5% average bandwidth of an equivalent fully-connected switchbox with 31.8% lower hardware cost. The multi-layer sparse switchbox and the optimal connectivity interconnect network are linearly scalable with the number of processing elements on the array and contribute roughly 55% to the area and 15% to the energy consumption of the array. The UDSP architecture achieves energy and area efficiencies within 4.2× and 6.4× to that of a hardware accelerator (ASIC) with equivalent throughput.

We developed a Runtime Reconfigurable Array (RTRA), which is a spatial and temporal fast multiplexing architecture that improves the capabilities of a reconfigurable array by increasing its active utilization. We developed a method of increasing active utilization of the RTRA by allowing multiple programs awaiting execution to be simultaneously spatially mapped onto the available array resources, effectively virtualizing the array. The fast program relocation and hardware compilation enabled by various components of active hardware management system in RTRA allow hardware resource virtualization over multiple programs. The virtualization mechanism, which we call Accelerator as a Service (ACAS), allows the system to offer hardware acceleration features to multiple programs, threads, or hosts simultaneously, where each such hardware acceleration request from the program is handled independently by the active hardware management system. RTRA significantly enhances the performance and throughput of a reconfigurable array by increasing its active utilization while adding minimal overhead to area and energy consumption of the array. We observed throughput gains varying between $8 - 14\times$ while using RTRA as compared to a statically-configured UDSP array of the same size for signal processing workload of blind signal classification, and $1.4 - 8\times$ improvement in throughput for miscellaneous workloads comprising of linear algebra, machine learning, neural networks and media encoding.

6.1 Research Contributions

6.1.1 Multi-Domain Architectural Efficiency Metric

- The architectural efficiency metric can be used to compare and analyze multiple architectures using instruction-based analysis. Designers can gauge the performance benefits and efficiency of a new architecture or design at the pre-RTL stage and compare it against other existing architectures. The insights provided can be used to optimize the instruction set or reconfiguration bits of the design. The metric provides a methodology to analyze multiple architectures solely on the basis of their micro-architecture, e.g. RISC vs CISC, independent of hardware or physical optimizations. With additional insights about the clock frequency and parallelism, the metric can be used to provide a tighter estimate of throughput of the architecture for a given program.

6.1.2 Universal Digital Signal Processor (UDSP)

- The method of analyzing and developing interconnect networks allows the designer to apply a graph-based approach to analyzing the connectivity requirements of a domain of algorithms and apply simple, typical compiler optimization techniques on the graphs to understand and develop the interconnect network of optimal bandwidth, network diameter, and connectivity. This approach cuts down the overhead of interconnect network of a reconfigurable hardware by helping the designer better understand the requirements of domain and reduce the over-provisioning of the network.

- The method to develop optimally connected sparse switchboxes helps cut down the redundant connectivity of a fully-connected switchbox by using a multi-layer approach. The connectivity between the layers and hardware cost of the switchbox can be optimized by randomly removing connections between elements of different layers while keeping track of loss in average bandwidth of the switchbox. Any unacceptable random connection removal that causes massive drop in average bandwidth of the switchbox

is back-tracked, and the process is repeated multiple times until the lowest hardware cost is achieved for the required average bandwidth.

- The optimizations enable UDSP to achieve a peak energy efficiency of $785GMAC/s/W$ at 420mV and 315MHz clock frequency and $284GMAC/s/W$ energy efficiency and $34.5GMAC/mm^2$ area efficiency at nominal operating voltage of 800mV and 1.1GHz clock frequency. The UDSP is within 4.2× and 6.4× energy and area efficiency of an ASIC at nominal operating conditions.

6.1.3 Streaming Near Range $10\mu m$ (SNR-10) Channel

- SNR-10 is a light-weight communication channel with a simple redundancy and repair scheme over unidirectional IOs, which enables a tight, low-overhead integration of multiple IO pins required for fine-pitch $10\mu m$ Silicon Interconnect Fabric (Si-IF) interposer.

- The first $10\mu m$ pitch protocol on the Si-IF fine-pitch interposer has the lowest area per IO of $137\mu m^2$ on TSMC 16nm and Global Foundry 22nm technologies.

- The SNR-10 channel is a fully synthesizable, 0.38pJ/bit, 3 clock cycle latency, 297Gbps/mm implementation in TSMC 16nm. Using the SNR-10 channel, a 2×2 UDSP MCM can achieve a cross-sectional inter-chiplet bandwidth of 493Gbps using only two layers on the Si-IF interposer.

6.1.4 Runtime Reconfigurable Array for Accelerator as a Service

- The multi-step compile paradigm breaks up the program compilation for reconfigurable arrays into two steps of software and hardware compilation. The software compiler performs the time intensive steps of program compilation and provides a soft-mapped pre-compiled binary to the hardware compiler, which finishes the compilation procedure by hard mapping the binary to an appropriate vacant resource on the array.

- The multi-size compile allows the software compiler to generate program binaries of multiple sizes. The hardware scheduler and active hardware management system of the reconfigurable array can then proactively choose the appropriately-sized program binary that can be mapped onto the available resources on the array. The multi-size compile solution is a trade-off between the amount of storage required to store the multiple program binaries and the runtime resource allocation and execution of the program.

- The polygon abstraction approach for reconfigurable arrays allows for a low complexity check of overlap between the incoming program to the array and the existing programs on the array. The polygon abstraction also enables a tunable fine-grain and coarse-grain representation of the program. A fine-grain higher order polygon may have higher active utilization of processing elements within the polygon but the complex, irregular shape would require more calculations to be mapped on to the array and vice versa. We chose polygons of degree 4 to demonstrate the polygon abstraction technique.

- The anchor points-based approach speeds up the mapping process of the incoming program to the array. The anchor point-based mapping approach allows the active hardware management system to calculate feasible locations on the array that have a high likelihood of successful mapping of any incoming program. The active hardware management system can perform this task of anchor point generation in the background during idle time between two successive incoming programs, thus reducing the time-to-map of further incoming programs.

- The polygon abstraction, multi-step compile, multi-size compile, anchor points allows us to devise methods for array virtualization. Array virtualization allows for any number of programs to be simultaneously spatially or consecutively temporally mapped on to the array. The array is effectively virtualized over multiple programs of multiple sizes as permitted by the size of array.

6.2 Future Directions

This research offers multiple directions for future exploration. First, there is potential room for improvements in spatial mappings of programs onto the array. The scheduler that we developed for the array has a greedy scheduling mechanism, as it places the incoming program onto the first available feasible anchor point. This greedy scheduling may cause higher fragmentation of the array, thus there is potentially room to develop advanced scheduling mechanisms which can reduce fragmentation. Second, the active hardware management system devised in this work requires that no program mapped onto the array be directly in communication with another program spatially mapped onto the array. This requirement ensures that a malicious program cannot spy on other programs, which forces the array to always bring the IO data of the programs from the memory. However, if two consecutive kernels of the same program are mapped spatially simultaneously onto the array, the array should be able to provide direct communication between the two without requiring memory read/writes of IOs, thus saving memory bandwidths as well as data moving time and energy. Additionally, some of the most frequently used program binaries can be stored in a local program memory on the active hardware management system, which could help increase the programming bandwidth of the array by providing for a simple instruction compression technique.

The current software compiler requires the user input of the program in the form of a data flow graph. Further work is required to implement translation of some commonly used programming languages such as C/C++ to such data flow graphs, which could greatly improve the use of such reconfigurable hardware. Further, a library of configurations of some commonly-used program kernels such as FFT, MAC, etc. can be generated to speed up the compilation process. The user could call the library function in their program and the compiler can easily recognize the function and replace it with an optimal binary.

I envision that RTRA can be integrated into the SoC for servers, self-driving vehicles and mobile phones, where it can help accelerate general-purpose compute and domain spe-

cific programs. A 'CPU + RTRA' server SoC can achieve a higher throughput at greater efficiencies as compared to other such hybrid architectures. This hybrid system can accelerate many commonly used algorithms as we demonstrated, and the reconfigurable array can be modified and made heterogeneous to accommodate a wider domain of applications to accelerate. An RTRA based solution can be used in self-driving vehicles to perform object recognition and perform guidance control operations. UDSP array is efficient and provides high throughput for inferencing operations, addition of active management system and high-bandwidth programming would allow the array (RTRA) to adapt to fast changing environment and emergency situations that might emerge during self-driving operations of a car or an aerial vehicle. For hardware-accelerator dominant systems such as mobile phone SoCs, RTRA provides a unique solution as a replacement for the hardware accelerators. The increasing count of hardware accelerators can be replaced with a single large RTRA array, and the algorithms/applications that require the acceleration can be spatially co-mapped onto the RTRA. This approach would allow the designers to reduce the number of hardware accelerator blocks on the SoC and also reduce the amount of inactive silicon on the SoC, while also ensuring a longer system relevance due to the flexible, programmable nature of reconfigurable arrays instead of hard-wired and inflexible hardware accelerators.

REFERENCES

[1] 3GPP, "Advanced plans for 5g," 2021. [Online]. Available: https://www.3gpp.org/news-events/2210-advanced_5g

[2] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins *et al.*, "Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2020, pp. 145–158.

[3] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar *et al.*, "The raw compiler project," in *Proceedings of the Second SUIF Compiler Workshop*. Citeseer, 1997, pp. 21–23.

[4] ARM, "Amba 5 chi architecture specification," ARM, Tech. Rep., 2021.

[5] A. A. Bajwa, S. Jangam, S. Pal, N. Marathe, T. Bai, T. Fukushima *et al.*, "Heterogeneous integration at fine pitch (10 um) using thermal compression bonding," in *Proceedings of the 67th Electronic Components and Technology Conference (ECTC)*. IEEE, 2017, pp. 1276–1284.

[6] L. Benini and G. De Micheli, *Network On Chips*. Morgan Kaufmann, 2006.

[7] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable fpgas," in *Proceedings of the 37th Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 1–12.

[8] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*. ACM, 2008, pp. 389–400.

[9] A. Dasu, A. Sudarsanam, and R. Kallam, "Prr-prr dynamic relocation," *IEEE Computer Architecture Letters*, vol. 02, pp. 44–47, jul 2009.

[10] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[11] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport *et al.*, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[12] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. IEEE/ACM, 2011, pp. 365–376.

[13] A. Fog, "The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers," 2012. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[14] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla *et al.*, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *Proceedings of the 53rd International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2020, pp. 681–697.

[15] A. Goel and W. R. Lee, "Formal verification of an ibm coreconnect processor local bus arbiter core," in *Proceedings of the 37th Design Automation Conference (DAC)*. ACM, 2000, p. 196–200.

[16] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[17] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam *et al.*, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[18] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, p. 48–60, jan 2019.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[20] IBS and M. Lapedus, "Big trouble at 3nm." [Online]. Available: https://semiengineering.com/big-trouble-at-3nm/

[21] S. Jangam, U. Rathore, S. Nagi, D. Markovic, and S. S. Iyer, "Demonstration of a low latency (20 ps) fine-pitch (10 um) assembly on the silicon interconnect fabric," in *Proceedings of the 70th Electronic Components and Technology Conference (ECTC)*. IEEE, 2020, pp. 1801–1805.

[22] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian *et al.*, "Ten lessons from three generations shaped google's tpuv4i : Industrial product," in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2021, pp. 1–14.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa *et al.*, "In-datacenter performance analysis of a tensor processing unit," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, p. 1–12, jun 2017.

[24] M. Kirchhoff, P. Kerling, D. Streitferdt, W. Fengler, and J. Kalomiros, "A real-time capable dynamic partial reconfiguration system for an application-specific soft-core processor," *Hindawi Limited International Journal of Reconfigurable Computing*, vol. 2019, Jan. 2019.

[25] M. Y. Lanzerotti, G. Fiorenza, and R. A. Rand, "Microminiature packaging and integrated circuitry: The work of e. f. rent, with an application to on-chip interconnection requirements," *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 777–803, 2005.

[26] R. R. Lewis, *A Guide to Graph Colouring: Algorithms and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[27] M.-S. Lin, T.-C. Huang, C.-C. Tsai, K.-H. Tam, K. C.-H. Hsieh, C.-F. Chen *et al.*, "A 7-nm 4-ghz arm[1]-core-based cowos[1] chiplet design for high-performance computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 4, pp. 956–966, 2020.

[28] C. Liu, J. Botimer, and Z. Zhang, "A 256gb/s/mm-shoreline aib-compatible 16nm finfet cmos chiplet for 2.5d integration with stratix 10 fpga on emib and tiling on silicon interposer," in *Proceedings of the International Custom Integrated Circuits Conference (CICC)*. IEEE, 2021, pp. 1–2.

[29] D. Liu, "Chapter 3 - an introduction to cryptography," in *Next Generation SSH2 Implementation*, D. Liu, M. Caceres, T. Robichaux, D. V. Forte, E. S. Seagren, D. L. Ganger *et al.*, Eds. Burlington: Syngress, 2009, pp. 41–64.

[30] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han *et al.*, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys*, vol. 52, no. 6, oct 2019.

[31] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '04, vol. 2. IEEE Computer Society, 2004, p. 21224.

[32] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: Evaluating spatial computation for whole program execution," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, p. 163–174, oct 2006.

[33] S. S. Nagi and D. Markovic, "A multi-domain architectural efficiency metric," in *Proceedings of the International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2021, pp. 265–270.

[34] Nvidia, "Nvidia tesla p100 - the most advanced datacenter accelerator ever built - featuring pascal gp100, the world's fastest gpu," 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[35] Nvidia, "Nvidia turing gpu architecture," 2018. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/geforce-rtx-20-series-turing-architecture-whitepaper/

[36] Nvidia, "Nvidia ampere ga102 gpu architecture," 2020. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/rtx-30-series-ampere-architecture-whitepaper-download/

[37] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2009, pp. 370–380.

[38] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis *et al.*, "Plasticine: A reconfigurable architecture for parallel patterns," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. IEEE/ACM, 2017, pp. 389–402.

[39] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the 41st International Symposium on Computer Architecuture (ISCA)*. IEEE, June 2014, pp. 13–24, selected as an IEEE Micro TopPick.

[40] U. Rathore, S. Nagi, S. Iyer, and D. Markovic, "2.6 a 16nm 785gmacs/j 784-core digital signal processor array with a multilayer switch box interconnect, assembled as a 2×2 dielet with 10µm-pitch inter-dielet i/o for runtime multi-program reconfiguration," in *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2022, pp. 52–53.

[41] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan *et al.*, "Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 1, p. 62–93, mar 2004.

[42] B. Seyoum, M. Pagani, A. Biondi, and G. Buttazzo, "Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in fpga soc," in *Proceedings of the 36th ACM Symposium on Applied Computing (SAC)*. ACM, 2021, p. 481–490.

[43] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "The aladdin approach to accelerator design and modeling," *IEEE Micro*, vol. 35, no. 3, pp. 58–70, 2015.

[44] J. Simon, "Large language models: A new moore's law?" [Online]. Available: https://huggingface.co/blog/large-language-models

[45] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, pp. 465–481, 2000.

[46] D. Suggs, M. Subramony, and D. Bouvier, "The amd "zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.

[47] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson *et al.*, "The wavescalar architecture," *ACM Transactions on Computer Systems*, vol. 25, no. 2, may 2007.

[48] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez *et al.*, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2010, p. 205–218.

[49] C. C. Wang, F.-L. Yuan, T.-H. Yu, and D. Markovic, "27.5 a multi-granularity fpga with hierarchical interconnects for efficient and flexible mobile computing," in *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 460–461.

[50] C. Wang, "Building efficient, reconfigurable hardware using hierarchical interconnects," Ph.D. dissertation, University of California, Los Angeles, 2013. [Online]. Available: https://escholarship.org/uc/item/2vt0b5cb

[51] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, p. 65–76, apr 2009.

[52] J. M. Wilson, W. J. Turner, J. W. Poulton, B. Zimmer, X. Chen, S. S. Kudva *et al.*, "A 1.17pj/b 25gb/s/pin ground-referenced single-ended serial link for off- and on-package communication in 16nm cmos using a process- and temperature-adaptive voltage regulator," in *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2018, pp. 276–278.

[53] Xilinx, "Benefits of partial reconfiguration," *Xcell Journal*, vol. 55, pp. 65–67, 2005.

[54] Xilinx, "Planahead software as a platform for partial reconfiguration," *Xcell Journal*, vol. 55, pp. 68–71, 2005.

[55] Xilinx, "Zynq rfsoc dfe backgrounder," 2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/rfsoc/zynq-ultrascale-plus-rfsoc-dfe.html

[56] T.-H. Yu, O. Sekkat, S. Rodriguez-Parera, D. Markovic, and D. Cabric, "A wideband spectrum-sensing processor with adaptive detection threshold and sensing time," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 11, pp. 2765–2775, 2011.

[57] F.-L. Yuan, "Energy-efficient vlsi architectures for next-generation software-defined and cognitive radios," Ph.D. dissertation, University of California, Los Angeles, 2014. [Online]. Available: https://escholarship.org/uc/item/5vw9x3p9

[58] Y. Zha and J. Li, "Virtualizing fpgas in the cloud," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020, p. 845–858.

[59] Y. Zha and J. Li, "Hetero-vital: A virtualization stack for heterogeneous fpga clusters," in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2021, pp. 470–483.

[60] Z. Zhao, W. Sheng, W. He, Z. Mao, and Z. Li, "A static-placement, dynamic-issue framework for cgra loop accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 1348–1353.