Programmable Hardware Acceleration

by

Vinay Gangadhar

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 11/16/2017

To Amma, Appa and my dear sister Vindhya
To my fiancée, Chandini

Acknowledgments

It has been a great doctorate journey so far, and this would not have been possible without the support and encouragement of so many people. I want to sincerely thank all of them involved to make this dissertation happen.

First is, of course, my advisor and mentor, Karu Sankaralingam. From the point of taking the first research project with him in my Masters, to the final defense of my Ph.D. dissertation, Karu believed in me and has always been supportive by providing valuable advice and suggestions. He took time out of his protected academic time to help pursue my goals and provide extensive personal and professional guidance. He has taught me a great deal about computer architecture research and also to lead a happy and positive life in general. I am most thankful to him for instilling the confidence in me that I can do impactful research and deliver great talks. I will always remember those endless brainstorming sessions we had on how I should focus on bigger picture research problems and its impact on the computer hardware industry. Karu, thanks for making me the researcher I am today.

My dissertation committee was very kind in always providing me insightful comments and feedback on how I should proceed with my research. Special thanks to Mark Hill (who I always look up to as an architect), for appreciating my dissertation findings and his advice on keeping things simple and yet be most impactful. Despite his busy travel schedule this year, he was always available to read my dissertation and ask interesting questions. I will never forget the meeting I had in your office where you gave great advice on after-PhD life. Thank you, David, for teaching me about parallel programming skills and forcing me to think critically about memory subsystem problems. I always enjoyed your real-life analogies linked to some of the architecture concepts. Thanks Mikko for teaching me the two important architecture classes, and be always available for interesting technical discussions. Thank you Dimitris for your abundant knowledge transfer on machine learning and to apply them to my research. I would also take this opportunity to thank other professors in UW-Madison – Nam Sung Kim from whom I learned about processor micro-architecture, Eric Hoffman for teaching me the Verilog and digital design skills and Dan Negrut who passionately taught about GPGPUs and programming them.

I would like to thank all my fellow students in the architecture community. First is the Vertical research group. Special thanks to my officemates Tony Nowatzki and Vijay Thiruvengadam for supporting me when I was going through a turbulence in my personal life. Tony, I have been inspired by your research so much and have imbibed the architecture modeling skills from our collaborations. You taught me how to stay calm, composed and easily attack complex problems and it would not be wrong at all if I have to call you as my "second advisor". Cheers to you and looking forward to more such collaborations. To the awesome MIAOW GPGPU group – Raghu, Mario, Ziliang, Cherin, Robin, Sharath, Pradip, and Jai. This is a special project in my life which made me a sound micro-architect I am today and also get known as the "Open-source GPGPU guy" in so many conferences. This project would not have been famous as it is today without the effort from each one of you. Thanks to my other work buddies – Newsha for the machine learning discussions we had; Preyas (the best hardware engineer I know of) and Jiwei for making my research realize in hardware; Will, Simon and Alex for being the next door office buddies.

I want to thank my alumni involving amazing architects for their advice, feedback and time for discussing the potential research path I can take in my initial years of Ph.D. – Jason, Marc, Nilay, Jayneel, Shoaib, Joel, Lena, Hongil, Dibakar, and Gagan. You guys introduced me to so many aspects of architecture research and hope to meet you guys more often in near future. Architecture lunch and reading group – Swapnil, Gokul, Rohit, Aditya, Mitchell, Carly, Michael, David, and Kyle. It was pleasure interacting with you guys and I am sure all those insightful and heated arguments we had on each topic will come into use for my future conference talks.

I now want to take the opportunity to thank people from the new hardware startup I am part of – SimpleMachines, Inc. While I was at the last phase of my Ph.D. the experience I got from working in this company gave me the confidence to defend my dissertation with an industry perspective. Thanks to my advisor Karu for being the perfect CEO and also trusting my research findings and making it realize in a product. Thanks to our President Jeff for his constant support and push for me to finish Ph.D. so that I can be part of this new family. Thanks Zev for the systems wizard you are; Zach for all the hard work and more importantly the fun and humor you bring to our company; Ziliang for the FPGA expert you are; Simon and Preyas for the expert micro-architecture knowledge you bring to the company; Rahul for putting my research into test with your test-framework; Rongyi for the hard work you put and for those life-zen lessons; William for the optimization expert you are; Vijay for your architecture knowledge and your immense modesty, simplicity with which you pull off the work; Gabe, Karan, Anshu, Suhas, and Yunfeng for the AI experts you are. Guys, this has been a wonderful start I had exactly dreamt of and let's make our hard work pay off.

How can I not mention my friends far apart in India as well as here in US, who have been virtually always present with me in my last leg of the Ph.D., providing constant support in difficult phases – Thank you Prathik, Sid, Abhishek, Anil, Sharmila, Sanjay, Amogha, Puneeth, Shrestha, Naresh, Anusha, Shivapramod, Rakesh, Harini, Abhilash, and Spandan. I owe you guys a lot for filling so much joy in my life and make me accept things with at most positivity.

And finally thanks to my lovely family – Nobody has been more important to me in this pursuit than my family. I would like to thank my mom for the constant unconditional love and support she has given me even when I am 8000 miles far. Thank you, dad, for trusting me that I can pull off a Ph.D. even without knowing what exactly I was doing all these years. You both were always there to motivate me, stand beside me when I was going through situations of fear, uncertainty, and confusion. Mom and Dad, you are my ultimate role-models and thank you for everything. Thank you, my dear sister, Vindhya for taking care of mom and dad while I am away and being the strong piller of emotional support for our family.

To the love of my life and my dear fiancée – Chandini. I know this is not the path we had chosen for us, but thank you for understanding my passion and respecting my decision to do a Ph.D. It has been extremely difficult at times, but we have successfully made it till here and many more adventures to come. Thank you for all that support, care and love you gave me when I needed it the most and for being a proud partner when I got the doctorate. Love you.

Contents

List of Tables

List of Figures

Abstract

The rising dark silicon problem and the waning benefits of device scaling has caused a push towards specialization and hardware acceleration in last few years. While existing programmable and general purpose data-parallel architectures like SIMD engines and GPGPUs are orders of magnitude more efficient than their scalar counterparts, these benefits are still insufficient for the emerging applications. Recently, computer architects both in industry and academia have followed the trend of building custom high performance hardware engines for individual application domains, generally called as Domain-Specific Accelerators (DSAs). DSAs have been shown to achieve 10 to 1,000 times performance and energy efficiency improvements over general-purpose and data-parallel architectures for various application domains like machine learning, computer vision, databases and others. While providing these huge benefits, DSAs sacrifice programmability for efficiency and are prone to obsoletion due to domain volatility. The stark trade-offs between efficiency and generality at these two extremes poses an interesting question: Is it possible to have an architecture which has the best of both – programmability and efficiency, and how close can we get to such a design?

This dissertation explores how far the efficiency of a programmable architecture can be pushed, and whether it can come close to the performance, energy, and area efficiency of a domain-specific based approach. We specifically propose a type of hardware acceleration called "Programmable Hardware Acceleration", with the design, implementation and evaluation of a hardware accelerator which is programmable using an efficient hardware-software interface and yet achieve efficiency close to DSAs. This work has several observations and key findings. First, we rely on the insight that 'acceleratable' algorithms have common specialization principles and most of the DSAs employ these. Second, these specialization principles can be exploited in a hardware architecture with a right composure of programmable and configurable micro-architectural mechanisms. Third, these mechanisms can be used as hardware primitives in building a generic programmable hardware accelerator. Fourth, the same primitives can also be exposed to the programmers as a hardware-software interface to take benefit of the programmable hardware acceleration. Finally, our evaluation and analysis suggest that a programmable hardware accelerator can achieve performance as close as DSAs with only 2x overheads in area and power. Along with these findings, we also believe that this programmable hardware accelerator can serve as a meaningful baseline to measure the true benefits of specialization and also enable future accelerator innovation by acting as a platform to discover more programmable specialization mechanisms. In summary, this work shows a principled approach in building hardware accelerators by pushing the limits of their efficiency while still retaining the programmability.

1 | Introduction

Data processing continues to dominate the global economy – from the scale of web services and warehouse computing, to networked Internet of things and personal mobile devices [1, 2]. Especially in recent times, as the requirements and applications in these areas have evolved, general purpose processors have become increasingly less effective, and have fallen out of focus, because of the energy and performance overheads of traditional VonNeumann architectures [3, 4].

For many years, general-purpose processor (GPP) architectures and micro-architectures have taken advantage of Dennard scaling and Moore's Law, exploiting increased circuit integration to deliver higher performance and lower energy computation. As has been noted [5, 6, 7], those physical trends are slowing, and thus there has been a recent surge of interest in more narrowly-applicable architectures in the hope of continuing system improvements in at least some significant application domains. This is far from an academic movement and this trend has been observed even in industries. For large scale computing, Microsoft has deployed the Catapult FPGA (Field-Programmable Gate Array) accelerator [8] in its datacenters, and likewise Google's Tensor Processing Unit (TPU) for distributed machine learning [9, 10]. For scalable and confiruable acceleration of the deep learning inference, NVIDIA has released a custom hardware solution along with a system architecture [11]. Internet of things devices and modern mobile systems on chip (SOCs) are already laden with custom hardware, and innovation continues in this space with companies like Movidius developing specialized processors for computer vision [12]. These examples also point out how General Purpose Programmable Graphics Processing Units (GPGPUs) and many-core SIMD[1] engines are becoming insufficient for power-optimized data processing tasks.

In the aspect of building such narrowly-applicable architectures, a popular approach so far has been – *Domain Specific Accelerators* (DSAs); hardware engines capable of performing computations

[1]Some have argued that GPGPUs and multi-core + SIMD are essentially the same [13, 14], simply trading off caches for register files.

of one application domain with high performance and energy efficiency. Acceleration with DSAs is generally called as *Domain Specific Acceleration* or *Specialization*. DSAs have been developed for machine learning [15, 16, 17], cryptography [18], XML processing [18], regular expression matching [19, 20, 21], H.264 decoding [3], databases [22, 23], speech recognition [24] and many others. Together, these works have demonstrated that, for many important workloads, accelerators can achieve between 10$\times$ to 1,000$\times$ performance and energy benefits over high performance, power hungry general purpose processors. For all of their efficiency benefits, DSAs sacrifice programmability and generality and are prone to obsoletion. Critically, the alternative to domain specific accelerators are not necessarily standard general-purpose processors, but rather *programmable* and *re-configurable* architectures which employ similar micro-architectural mechanisms for specialization.

This dissertation proposes a new paradigm in hardware acceleration called the "**Programmable Hardware Acceleration**", which enables us to have a programmable architecture composed of specialization elements. This specialized architecture called as *Programmable Hardware Accelerator* will be able to achieve efficiency as close as DSAs while still retaining the programmability. With regards to that, we first make a fundamental observation that DSAs, though differing greatly in their design choices, all employ a similar set of specialization[2] principles. Based on this important observation, the primary insight of the dissertation is that these common specialization principles can be exploited in a general way by composing *known* programmable and configurable micro-architectural mechanisms and arrive at a programmable architecture with efficiency close to DSA. We examine the specialization benefits of those mechanisms by first building a Generic Programmable Hardware Accelerator model called *GenAccel* which acts as a proof-of-concept model for evaluating the performance, area and power efficiency of programmable hardware acceleration.

Based on the model, we then consider a specific instance of programmable hardware acceleration called *Stream-Dataflow Acceleration* aimed at accelerating typical data-streaming applications. We propose a detailed architecture with programming abstractions, execution model, an ISA interface

[2]We broadly make use the terms specialization and acceleration interchangeably, as acceleration (both hardware and software) is achieved by specializing certain aspects of a general purpose architecture execution.

and a low-level micro-architecture implementation for the stream-dataflow accelerator. In summary, this dissertation explores a principled approach of modeling and designing a programmable hardware accelerator along with its architecture, micro-architecture, execution model and compilation techniques by adapting a hardware-software co-design methodology.

The rest of the chapter briefly describes hardware acceleration and domain-specific acceleration in general and then delves into the dissertation's main theme *Programmable Hardware Acceleration* by introducing two different specialization paradigms. We finally end the chapter with the contributions of the thesis and dissertation organization.

1.1 Hardware Acceleration

Hardware Accelerators are units of hardware logic which *specialize* certain aspects of a general purpose processor's execution to improve performance and/or energy efficiency. Since, the application phases amenable to acceleration are offloaded to the hardware accelerators from a host general purpose processor, these accelerators are also called as *co-processors*. These accelerators are composed of custom hardware logic specifically designed to execute important program phases of an application which would otherwise incur lot of power and performance overheads in general purpose architectures. Especially, with the recently explored dark silicon problem [25, 26, 27], its important to utilize the available transistors to do meaningful computation in an efficient way, thus achieving better performance and higher energy efficiency. Even from an industry perspective, hardware accelerators are very important for reducing the overall total cost of ownership (TCO), by utilizing the chip area efficiently and also reducing the power consumption for doing the same amount of computation.

In hardware accelerators, this process of giving up generality for efficiency on a single workload, workload domain or workload type is called *architectural specialization*. They firstly do this by exploiting particular aspects of a target application domain and building the specialized custom logic to execute those application phases. The specialized hardware logic might include customized functional units (FUs), memory interfaces, datapaths and interconnect architectures. Second, hardware accelerators sometimes may expose a minimal software interface with computation abstractions to be tuned by the programmer. A popular example is SIMD, which specializes for data-parallel computation, and another is Coarse Grain Reconfigurable Architectures (CGRAs), which specialize for computation pattern re-use. In the last decade, General Purpose Computing on Graphical Processing Units (GPGPUs) have gained traction and are prevalent for executing both data-parallel and thread-parallel applications [28, 14]. The way that accelerators rely on applications for certain properties is what we call *accelerator-application* interaction. In addition to exploiting properties of applications, accelerators also use specific hardware techniques to reduce the overheads of a general purpose processor. SIMD processors have parallel datapaths, and reduce instruction count, CGRAs employ reconfigurable datapaths and GPGPUs have thousands of tiny-cores to exploit thread-level parallelism. The strategies by which accelerator's specialize the general purpose core is what we call the *accelerator-core* interaction.

There are associated challenges with hardware acceleration in general. *Data transfer* and *integration* with host processor are two main challenges and has tradeoffs with respect to power, area and performance. In case of closely-coupled hardware accelerators, these two are related to each other as the accelerator needs to be tightly integrated to processor pipeline in order for a faster data transfer and avoid the switching overhead. Hardware-software co-design is another challenge in accelerators where right hardware abstractions need to be exposed to the programmers in-order to take advantage of the specialized hardware. Designing the accelerator with set of micro-architectural mechanisms supporting the hardware-software co-design is very important affecting the factors like chip cost, efficiency and speedup. And finally modifying the application to take advantage of the hardware acceleration and compiling the application to produce the right set of instructions to reduce runtime overheads also needs to be addressed when designing the hardware accelerator. We address all these challenges in detail for a programmable hardware accelerator in Chapter 2.

1.2 Domain-Specific Acceleration

As performance improvements from general-purpose processors have proved elusive in recent years, it has lead to a surge of interest in narrowly-applicable architectures in the hope of continuing system improvements in some emerging application domains like – Machine Learning, database streaming, image processing, speech recognition, neural networks etc. Since, these architectures specialize/accelerate a particular application domain by giving up generality, the acceleration is called as Domain-Specific Acceleration. When generality is no longer a requirement, a range of engineering approaches opens up from fully custom digital systems design, through hardware-software co-design. In many cases, such as in systems-on-chips (SoCs), the system as a whole may remain general purpose (or multi-purpose), while multiple domains are candidates for specialization within it. Domain-Specific accelerators have become pervasive in recent heterogeneous systems where a 'sea of diverse customized accelerators' [29, 30] are deployed on a SoC to take advantage of orders of magnitude improvements in energy efficiency.

There is another subclass of accelerators where a purely fixed-function digital approach is used without providing any flexibility and is generally fabricated as a stand-alone hardware chip. These are called as fixed-function accelerators or more commonly as ASICs. It has its own discipline and history and is very popular in multi-media, embedded systems and recently in Internet of Things (IOT) domain. This dissertation does not consider such accelerators and instead restricts discussion to domain-specific hardware that exists alongside or integrated with general-purpose processors.

1.3 Programmable Hardware Acceleration

The main principle behind programmable hardware acceleration is that common accelerator application behaviors can be exploited using a generic hardware accelerator design with an efficient hardware-software interface exposed to the programmers.

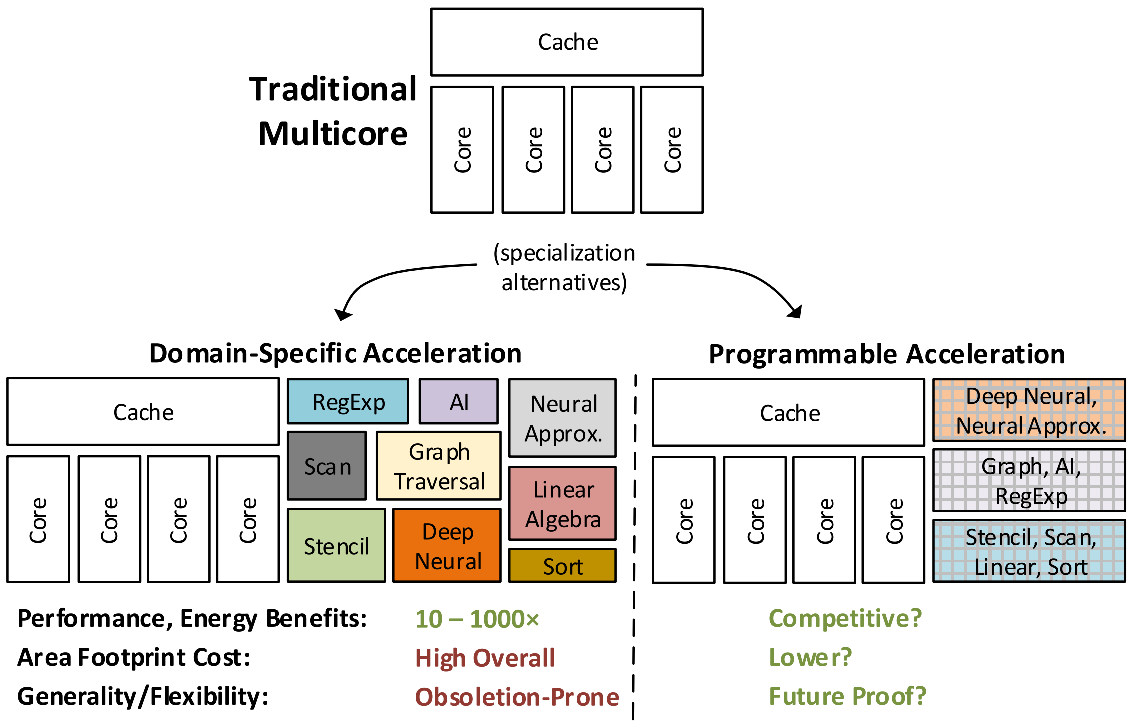


Figure 1.1: Specialization Paradigms and Tradeoffs

Specialization Paradigms Figure 1.1 depicts the two specialization paradigms at a high level, which also leads to the central question of this work: *how far can the efficiency of programmable architectures be pushed, and can they be competitive with domain-specific designs?* As shown in the figure, the goal is to move towards programmable acceleration paradigm by understanding the caveats of domain-specific acceleration and be able to map multiple accelerator applications on to a programmable fabric which is competitive with the DSA designs.

In terms of achieving programmable hardware acceleration, ideally what we require is a hardware that is capable of executing these data-intensive algorithms at high performance with much lower power overheads than existing programmable architectures, while remaining broadly applicable and adaptable. The promise of such an architecture is high efficiency and the ability to be flexible across multiple application domains. Given that requirement, following are the five hypotheses that must hold true for programmable acceleration to be effective:

1. *Specialization Principles* – Most of the DSAs executing the accelerator applications have commonality in the way they specialize these algorithms and these common properties can be categorized as "Principles of Architectural Specialization".

2. *Mechanisms* – A programmable hardware accelerator design must exploit the common specialization principles of the accelerator applications, to be competitive with DSAs. Such a design can be implemented by known programmable and reconfigurable micro-architecture mechanisms associated with each of the specialization principle.

3. *Programming Interface* – The programming interface must express the accelerator's hardware mechanisms as software primitives to take advantage of the specialization. The efficiency of the programmable accelerator derives from this hardware-software co-design.

4. *Application Adaptability* – Any new algorithm or application in general amenable to hardware acceleration is easy to adapt with the programming interface and get the benefits of programmable hardware acceleration.

This dissertation with two main pieces of work –

i) Building a generic programmable accelerator model called *GenAccel* to explore the viability and efficiency of programmable hardware acceleration;

ii) A detailed architecture with programming abstractions, execution model, accelerator ISA interface for a programmable hardware acceleration instance called *Stream-Dataflow Acceleration* aimed at accelerating data-streaming applications ;

shows that the four hypotheses mentioned above are true and it is possible to have a programmable accelerator with efficiency close to DSAs and still retain programmability. We discuss the need for such a programmable accelerator fabric, the primitives needed to build such fabric along with the challenges in Chapter 2.

1.4 Thesis Statement

A programmable hardware accelerator nearing the efficiency of a domain-specific accelerator is feasible to build by: i) identifying the common principles of architectural specialization, ii) applying general set of micro-architectural mechanisms for the identified principles and iii) having an efficient hardware-software interface to be able to express any typical accelerator application.

1.5 Contributions

This dissertation proposes a specialization paradigm called programmable hardware acceleration along with its high-level principles and mechanisms needed to achieve it. It focuses on a hardware-software co-design approach by describing the ISA interface needed for such programmable architecture alongside micro-architecture details for the implementation of the same. It discusses the potential impact of having such a programmable accelerator able to achieve efficiency close to domain-specific hardware for future applications without having the overheads of existing data-parallel architectures. We describe the detailed contributions below:

Accelerator Modeling to Explore Principles of Architectural Specialization

In this line of research we first explore, identify and propose principles of architectural specialization common to many domain-specific accelerators (DSAs). We then validate these principles in DSAs by building simple analytical architectural models based on instruction counts, trace-based simulation of the DSA workloads and show the application of the identified principles to the DSA models. These accelerator models are used to identify the sources of efficiency in each workload domain and also relate to the specialization principle defined.

Mechanisms to Exploit Specialization Principles

Once we have identified the common specialization principles, we identify the fundamental programmable and configurable micro-architectural mechanisms for each of the principles in order to achieve programmable acceleration. We build a model of the generic programmable accelerator fabric/layout called *GenAccel*, composed of the identified mechanisms. We explore different phases of GenAccel design, its usage including the programming phase and runtime configuration. This work finally describes a systematic design point selection of GenAccel model by provisioning the resources to map various applications from a single or multiple domains.

Quantitative Evaluation and Analysis of a Generic Programmable Accelerator Model

In order to understand the efficiency of a generic programmable accelerator design, we quantitatively evaluate GenAccel model with DSAs of four prominent application domains (Neural Approximation, Image processing, Deep Neural Networks and Database Queries) for performance, area and power and analyze its potential sources of efficiency. We also explore the cost of programmability added with regards to area and power for two designs of GenAccel model – a performance matched single domain-provisioned GenAccel and a multi-domain provisioned balanced GenAccel. We show that a generic programmable accelerator like GenAccel can match DSAs' performance, limiting the overheads of programmability to only 2× to 4×, as opposed to the 100× to 1000× inefficiency of large Out-Of-Order (OOO) processor cores.

System-Level Trade-offs with Programmable Hardware Accelerator and DSA

It becomes important to understand the significance of the area and power overheads of a programmable accelerator compared to DSAs, when considering system-level energy benefits and economic costs, rather than just evaluating the stand-alone accelerator. This part of work analyzes such broader system-level understanding of having a programmable accelerator (with limited area/power overheads) in a chip over DSA, provided both of them have same speedups. We also provide an analytical reasoning for the design decision question – In what scenarios would one employ a DSA over programmable accelerator in a System-On-Chip setting? We show that when programmable accelerator like GenAccel and DSAs have equivalent performance, the potential further system-level energy savings by using a DSA over GenAccel in a SoC is marginal.

Stream-Dataflow Acceleration – Architecture, Programming Interface and Execution Model

Based on the analysis of the generic programmable accelerator model (GenAccel), we define a detailed architecture for an instance of programmable hardware acceleration called *Stream-Dataflow Acceleration*. In specific, we define an execution model and an ISA interface for stream-dataflow acceleration aimed at accelerating regular data-streaming applications from many domains. We also define a detailed architecture for stream-dataflow accelerator by taking advantage of the opportunities available, after studying the overheads and inefficiencies present in existing data-parallel architectures. We illustrate the programmability of the stream-dataflow accelerator with a detailed workload example. We show that these programmer abstractions are simple enough, are effective for mapping diverse set of applications and also can efficiently and easily express any acceleratable workload.

Stream-Dataflow Accelerator Micro-Architecture Design – Softbrain

In order to have lower-overheads compared to exiting data-parallel architectures and be as energy efficient as application specific designs, efficient micro-architecture is needed in programmable accelerators. Stream-dataflow accelerator also has such requirements along with the need to have support for highly concurrent execution for both computation and memory access phases of the program. We describe a detailed, low-level micro-architecture design for stream-dataflow accelerator called *Softbrain* enabling pipelined concurrent execution with a reconfigurable computation fabric, data-reuse with a programmable scratchpad, a memory interface handling low-overhead address generation and a simple controller core for co-ordinating the data streams.

Stream-Dataflow Accelerator (Softbrain) Implementation

To evaluate stream-dataflow architecture for performance, area, power with domain-specific implementations and to prove its broader applicability to different application domains we need the hardware implementation of the accelerator alongside the support of a complete software stack comprising of compiler, simulator etc., We describe the software and hardware infrastructure support for end-to-end application execution of stream-dataflow programs on a stream-dataflow accelerator hardware. We explain the details of software stack comprised of a dataflow-graph compiler, a modified RISCV assembler (part of RISCV GNU compiler toolchain) for compiling stream-dataflow programs and a C++ based cycle-level architectural simulator for modeling stream-dataflow performance evaluation.

We also describe a detailed chisel based hardware prototype implementation of stream-dataflow accelerator (softbrain) with details of its interfaces and pipeline implementation.

Quantitative Evaluation of Softbrain

We evaluate our implementation of stream-dataflow accelerator – softbrain for performance, power, area and energy efficiency for two different classes of workloads. We first show the domain-specific application adaptability of stream-dataflow programmable accelerator by comparing domain-provisioned softbrain's performance with DSAs designed for that application domain. These applications include standard state-of-the-art deep neural network (DNN) kernels. Softbrain when rightly provisioned can match the performance of the DSAs corresponding to these application domains with around 2x average overheads in area and power. Next, we also show broader applicability of stream-dataflow architecture on a more challenging accelerator application suite called Machsuite, which has both irregular memory accesses and computation patterns. Broadly provisioned softbrain can again match the performance of application-specific design points (ASICs) for each application with around 2x average overheads in power and energy. This quantitative analysis proves that softbrain can be used as a generic programmable accelerator design to accelerate wide variety of accelerator workloads. We also do a detailed area estimation of softbrain's RTL synthesized and compare it to both domain-specific hardware and application specific ASICs.

In summary, the contributions of the dissertation lies in identifying the general specialization principles, accelerator modeling to analyze different mechanisms, system-level trade-offs of having such a programmable accelerator. We also describe an efficient programming interface with application compilation techniques, execution model, architecture details and a detailed micro-architecture implementation to evaluate with state-of-the-art domain-specific hardware designs.

1.6 Dissertation Organization

Chapter	**Topic**
2	A Case for Programmable Acceleration
3	Principles of Architectural Specialization
4	Mechanisms for Programmable Acceleration
5	Evaluation of GenAccel Model
6	Stream-Dataflow Acceleration
7	Micro-Architecture of Softbrain
8	Evaluation of Softbrain
9	Related Work
10	Conclusion

Table 1.1: Dissertation Organization
Related Publications: HPCA 2016 [31], IEEE Micro Top Picks 2017 [32], ISCA 2017 [33]

The disseration organization is outlined in Table 1.1. We first establish a case for programmable hardware acceleration and provide the motivation for this disseration in Chapter 2. We then describe the common principles of architectural specialization in Chapter 3. Chapter 4 details the mechanisms for the common specialization principles. We also explain the modeling and designing of the generic programmable hardware accelerator (GenAccel) in this chapter. Chapter 5 describes the evaluation of the GenAccel model with four prominent DSAs. Next, Chapter 6 describes the steam-dataflow acceleration and an execution model, programming interface and the architecture for the same. Chapter 7 details the low-level micro-architecture details and implementation of the stream-dataflow accelerator called *Softbrain*. Chapter 8 describes the evaluation of implemented Softbrain for prominent and emerging applications. Finally, related work is explained in Chapter 9 and Chapter 10 summarizes the disseration and its future implications.

2 | A Case for Programmable Acceleration

In this chapter, we first describe the specialization spectrum and show how existing architectures align with regards to generality and efficiency. We then motivate the need for programmable hardware acceleration by discussing the caveats of domain-specific acceleration and the primitives needed for designing and building a programmable hardware accelerator. We then discuss the challenges associated with modeling, design, implementation and compiling applications for a programmable hardware accelerator. We end the chapter by briefly explaining the evaluation approach used in this dissertation.

2.1 Specialization Spectrum

This section provides an overview of how different architectures line up in the specialization spectrum with generality and efficiency. We first briefly clarify the definitions of generality and efficiency from this dissertation's perspective below.

Generality is the feature of hardware architecture to be able to be programmable for any algorithm and also be able to execute any computation even after chip fabrication and remain future proof with a considerable performance goal. These hardware architectures generally conserve the area by reusing and time multiplexing the expensive active circuitry for different functions of the application. Such architectures are called as "general-purpose" or application-indifferent as any application can be programmed using a high-level language and executed on the hardware by compiling it to the architecture's ISA. One such popular general architecture is the general-purpose processor (GPP) or simply put micro-processor. Although processors have different micro-architectural implementations to achieve different performance and frequency ratings across embedded, desktop and server market, they still remain general enough to map any software and present a system-

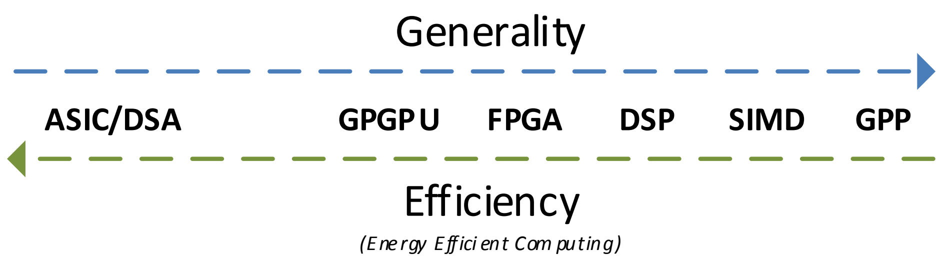


Figure 2.1: Generality and Efficiency Spectrum of Different Architectures

(ASIC: Application Specific Integrated Circuit, DSA: Domain Specific Accelerator, GPGPU: General Purpose Computing using Graphics Processing Unit, FPGA: Field Programmable Gate Arrays, DSP: Digital Signal Processors, SIMD: Single Instruction Multiple Data engines, GPP: General Purpose Processor)

independent programming model to a large base of applications. In recent years, with specialization efforts to achieve higher performance even processors specialize the floating point execution with floating point units (FPU), vector execution with MMX, SSE etc.

Efficiency is generally associated with power, area, energy and cost efficiency of any architecture for executing different applications. Although, a more pedantic definition of efficiency is associated with "performance per watt" or energy efficiency of a hardware architecture, different metrics are used for embedded, IOT, desktop and server space designs. Specialized architectures generally have higher energy efficiency as they consume less power to execute certain portions of the algorithm with high performance, as they possess custom hardware circuitry to perform the computations when compared to a general purpose design. Power efficiency is associated with how much less power a hardware unit spends in order to achieve the same performance. Area and cost efficiency are generally associated with "performance per mm^2" and "performance per dollar" metrics. For the purpose of discussion, this dissertation considers power, area and energy efficiency w.r.t hardware accelerators and other general purpose architectures.

Figure 2.1 shows the spectrum of generality and efficiency across existing general purpose processors, data-parallel architectures and the custom application-specific hardware. Towards the extreme right end of the generality spectrum are general purpose processors, which are highly-inefficient in order to be too general to support the mapping of all the applications or algorithms using a software-only approach. And towards extreme left end of the efficiency spectrum are the application specific hardware (ASICs) and domain-specific hardware or DSAs) which sacrifices generality to be very efficient. SIMD engines and DSPs even though have better efficiency for vectorized and Very-Long Instruction Word (VLIW) code execution compared to GPPs, they are very restrictive designs and are power hungry because of large register file accesses and data-alignment constraints. GPGPUs even though are generic enough to be programmable and are current goto solutions for data-parallel algorithms, are power hungry for typical accelerator applications with all the unnecessary overhead control structures and redundant mechanisms supported. On the other hand, FPGAs are generic enough to map any workload but lack in efficiency (frequency and power) because of fine-grained programmability and global routing mechanisms. FPGAs are also hard to program as algorithms need to be expressed in low-level hardware specific languages. Though Coarse-Grained Reconfigurable Architectures (CGRAs) with dynamic routing mechanism are good candidates for efficiency, are not general enough to map any algorithm and would be power inefficient if lot of coarse-grained functional units (FUs) are used to map all the workloads. A careful analysis of algorithms and a design-space exploration of FUs could make CGRA better from generality perspective. Future SOCs with special purpose hardware engines need a good balance between programmability and efficiency in order to push high performance computing to a new horizon.

2.2 Why Programmable Hardware Acceleration?

In this section, we establish a case for the need of a programmable hardware accelerator in the current era of specialization to accelerate emerging applications. We first describe the caveats of domain-specific hardware and then delve into the motivation for programmable acceleration.

2.2.1 Caveats of Domain-Specific Acceleration

We briefly introduced domain-specific acceleration in Section 1.2 and stressed the importance of such architectures in the current era of specialization to achieve orders of magnitude efficiency improvements over general purpose architectures.

For all of their *efficiency* benefits, DSAs have their own shortcomings and pose many challenges. Below we list some major caveats of domain-specific acceleration:

- **Hardware Re-design to Support Evolving Algorithms:** DSAs in order to achieve high efficiency give up on *programmability* - a high price to pay. This makes DSAs prone to obsoletion - the application domains which needs to be specialized, as well as the best algorithms to use, are constantly evolving at an alarming rate with scientific progress and changing user needs. As algorithms rapidly change, hardware must be re-architected, re-designed and re-verified which is burdensome in terms of development cost and time-to-market in new application areas. This is arguably true today for Augmented Reality and other emerging technologies. Moreover, this domain-specific approach burdens developers with problem-specific APIs and system stack modifications. Even the relevant application domains change between device types like server, mobile, wearables and hence the accelerator must be architected with new performance, power and area goals which restricts the architecture to be flexible.

- **Domain-Specific Hampering Algorithm Innovation:** As a corollary to the above caveat, innovation in algorithms becomes more difficult without access to flexible hardware. Even from the academic research viewpoint, it is much more difficult to formalize and apply improvements to a domain-specific hardware for new algorithms and in general to the broader field of computer architecture, thus limiting the scientific impact of such work.

- **Domain-Specific Hardware Cost:** While programmable hardware can be time-shared across multiple applications, that is not possible in DSAs, making it more expensive in terms of silicon. More subtly, most devices run several different important workloads (e.g. mobile chips), and therefore multiple DSAs will be required – this may mean that though each DSA is area-efficient, a combination of multiple DSAs in a System-On-Chip (SOC) may not be.

2.2.2 Motivation for Programmable Hardware Acceleration

In addition to the above mentioned disadvantages of DSAs, the cost per transistor in leading-edge process technologies is now increasing with each generation [34]. Thus, just when the pressure towards specialization is increasing inorder to continue delivering performance and functionality improvements, the cost of specialization is increasing too. It is therefore insufficient to consider each accelerator proposal in isolation. From programmability purposes and for overall better system-level optimization it is important that accelerators are considered in a common context to identify duplication and resource-sharing possibilities.

The academic literature, however, has tended to treat accelerator proposals in a standalone manner, as point solutions for a particular problem domain. Typically, each study demonstrates a worthwhile performance and power advantage relative to a general-purpose (out-of-order, superscalar) processor core – and, typically, even that baseline is different for each study. With no common frame of reference, it can be hard to abstract the real insight and contribution of each design, and to relate alternative partially-overlapping proposals to each other. Furthermore, its difficult to understand whether these point solutions have contributions that are valuable beyond their niche, if they are merely applications of well-understood engineering principles.

The alternative to domain specific acceleration is not necessarily general purpose cores, but a spectrum of more or less narrow *programmable architectures* – those which are applicable across several but perhaps not all workload domains. If a programmable architecture could be designed to capture most of the benefits of a DSA, then it could alleviate their design time, programming, and area burdens, and serve as a common baseline and framework to understand other accelerator proposals. The problem then becomes how to design a programmable yet specialized alternative for a variety of commonly accelerated workloads. As described earlier, such programmable accelerator

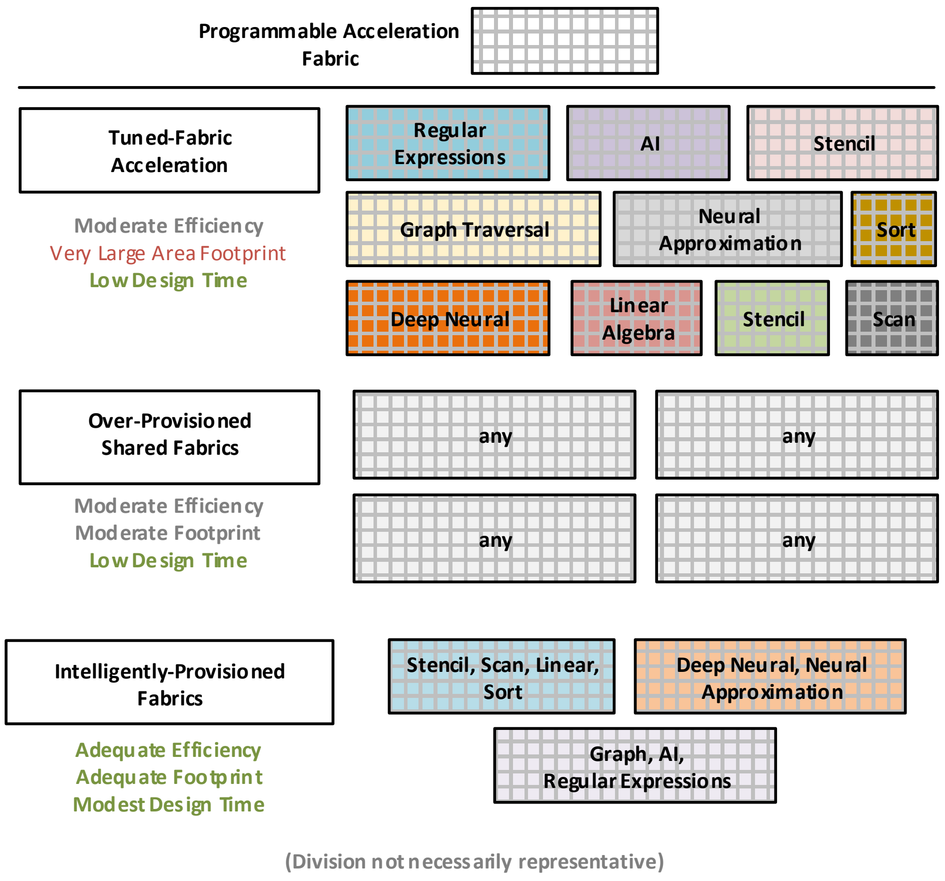

Programmable Acceleration Fabric
Tuned-Fabric Acceleration
Moderate Efficiency
Very Large Area Footprint
Low Design Time
Regular Expressions
AI
Stencil
Graph Traversal
Neural Approximation
Sort
Deep Neural
Linear Algebra
Stencil
Scan
Over-Provisioned Shared Fabrics
Moderate Efficiency
Moderate Footprint
Low Design Time
any
any
any
any
Intelligently-Provisioned Fabrics
Adequate Efficiency
Adequate Footprint
Modest Design Time
Stencil, Scan, Linear, Sort
Deep Neural, Neural Approximation
Graph, AI, Regular Expressions
(Division not necessarily representative)

Figure 2.2: Provisioning Techniques for Programmable Accelerator Fabric

architectures need to be composed of simple domain-agnostic micro-architectural mechanisms exploiting the common specialization principles of DSAs. We abstract this hardware architecture with the mechanisms as *Programmable Acceleration Fabric* for the purpose of discussion. Such fabrics can achieve programmable acceleration either by mapping one or many application domains on to the fabric's hardware substrate.

The acceleration fabric can be provisioned in multiple ways to achieve programmable acceleration, but it needs to have minimum overheads pertaining to programmability and must reach reasonable efficiency targets. Figure 2.2 shows different techniques for provisioning such a programmable accelerator fabric. The acceleration fabric is useful in several specialized-design paradigms. First, it can simply be a parameterizable fabric to produce DSAs, where its purpose is to reduce design time.

This is called as *Tuned Accelerator Fabric* and has overheads in terms of area and design cost but requires very less design time as it involves straight forward application of a domain to the hardware substrate. However, this is not a reasonable design choice when multiple application domains need to be mapped as it almost has same caveats as domain-specific hardware except the parameterization advantage. Alternatively, to achieve highly programmable and general acceleration, the fabric can be time-shared to execute all the application domains in the same hardware substrate. The fabric can synthesize the desired applications into a smaller set of homogeneous units. This is called as an *Over-Provisioned Shared Accelerator Fabric* and has moderate design time and higher generality compared to the tuned fabric. This can improve the area efficiency if workloads can time-share the fabric. However, if multiple applications need to be executed then the fabric units must be duplicated which could add to the area of the chip.

Finally, its possible to intelligently-provision a small number of programmable accelerator fabrics, each synthesized for a subset of workload domains. This paradigm costs modest design time and area and provides significant specialization benefits. Though we focus on synthesizing a single domain at a time in this work, we see this as the logical endpoint for a programmable accelerator architecture.

Besides facilitating implementations, this fabric also serves as a reference point along a spectrum of specialization techniques – a programmable architecture well suited for many commonly-specialized domains. We argue this enables it to serve as a standard baseline for the exposition and evaluation of different accelerator proposals, or even for side-by-side comparisons of DSAs from different domains. Given a problem-specific set of algorithmic insights, a workload synthesized into the fabric can be compared against a newly proposed DSA. The contributions of the DSA proposal then becomes much clear – i.e, any additional benefit can be attributed either to its ability to exploit algorithmic insights that did not synthesize well into standard fabric, or to more efficient underlying hardware mechanisms. These problem-specific insights and mechanisms could eventually be integrated into the programmable accelerator model. Its also conceivable to modify the programmable accelerator fabric over time by generalizing domain-specific insights and integrating new principles and hardware mechanisms.

Overall, on a general note a common framework for presenting and evaluating accelerator proposals is urgently needed on all fronts. Also, when you specialize the architecture, the relevant and most insightful comparison point is not the general purpose core, but a spectrum of more or less narrow accelerators that would also apply to the problem domain. For example, while a domain specific accelerator achieving 100$\times$ performance over an out-of-order does not tell us much, achieving 5$\times$ over a programmable accelerator engine tells us a lot – that specific micro-architectural innovations have bought 5$\times$ over the reasonable alternative of building something more generally useful for a variety of tasks.

Thus, this dissertation based on these insights and findings propose a programmable accelerator fabric using a principled approach of applying fundamental primitives involving common specialization principles of DSAs, mechanisms to exploit those principles and software interface to express these primitives. For the scope of the evaluation, we consider only tuned-fabric and the over-provisioned shared fabrics, as designing an intelligently provisioned accelerator fabric needs a detailed design-space exploration and is out of the scope for this dissertation. We discuss the primitives needed for building a programmable hardware accelerator in the upcoming section.

2.3 Primitives for Building a Programmable Hardware Accelerator

In this section, we discuss the fundamental requirements and primitives needed for a hardware architecture to achieve programmable acceleration. Figure 2.3 extends the specialization spectrum and implies the primitives that can be used to arrive at a programmable accelerator architecture. We explain each of them below:

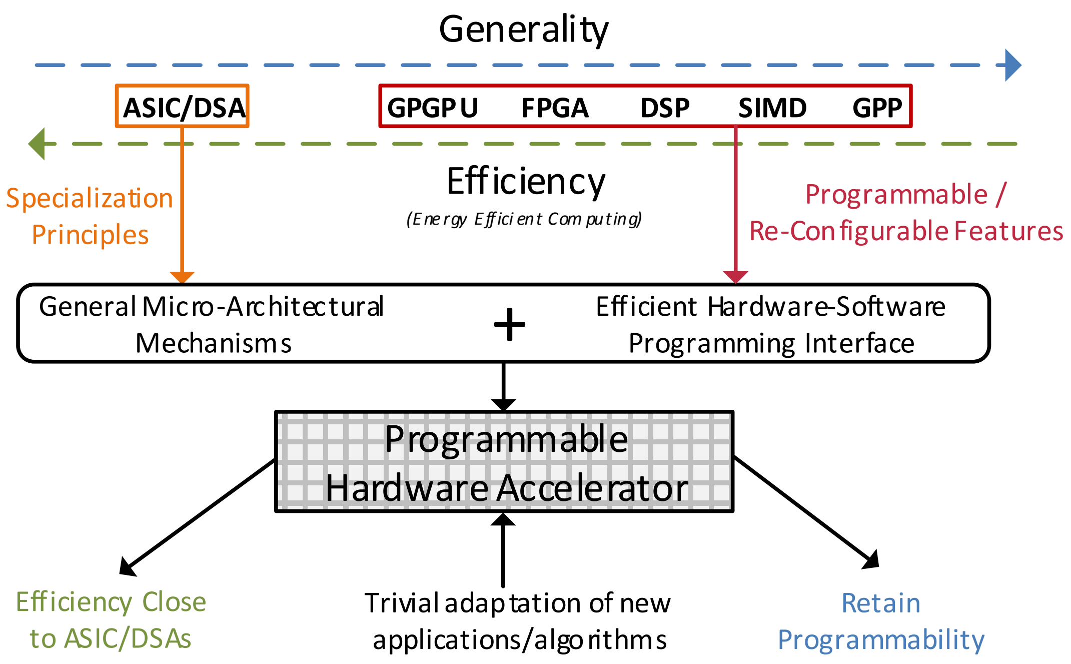

Generality
ASIC/DSA
GPGPU
FPGA
DSP
SIMD
GPP
Efficiency
(Energy Efficient Computing)
Specialization Principles
Programmable / Re-Configurable Features
General Micro-Architectural Mechanisms
+
Efficient Hardware-Software Programming Interface
Programmable Hardware Accelerator
Efficiency Close to ASIC/DSAs
Trivial adaptation of new applications/algorithms
Retain Programmability

Figure 2.3: Primitives for Programmable Hardware Acceleration

2.3.1 Specialization Principles

As alluded in Section 1.3, in order to achieve the benefits of specialization as domain-specific hardware, the programmable accelerator needs to derive inspiration from DSAs and employ the common specialization principles. And based on our primary observation, we find that most of the DSAs have similarity in the way they specialize the algorithmic properties of the application domain. So, the first primitive needed for building a programmable accelerator fabric is to employ the same common specialization principles as DSAs or ASICs in order for the programmable hardware accelerator to have its efficiency close to them. We discuss these principles in detail in Chapter 3.

2.3.2 Micro-Architecture Mechanisms

To exploit the common specialization principles in hardware, the programmable accelerator needs to embed general set of mechanisms in its micro-architecture implementation. These micro-architecture mechanisms need to be general enough so as to make the programmable accelerator fabric parameterizable for different application domains, but not have the power and area overheads of general purpose hardware mechanisms. For example, having expensive CAM like structures, large storage and control overhead structures gives generality but effects the efficiency of the accelerator design. So, the second fundamental primitive for an efficient programmable hardware accelerator design is to have simple configurable and programmable micro-architecture mechanisms exploiting the specialization principles either in decoupled/stand-alone fashion or all-together as integrated hardware components. Another important point to consider is that the accelerator design must also take into account of how it gets integrated into the host system or what is the mechanism for data transfer when accelerator applications are offloaded. A general design principle for accelerators has been that, it must be as non-intrusive as possible without disturbing the hardware interface of the host system too much or incurring overheads on the hardware-software co-design side to support the accelerator invocation from the host.

The abstract generic accelerator model capturing the general mechanisms needed to exploit the specialization principles is discussed in Chapter 4. We discuss the detailed micro-architecture implementation for stream-dataflow programmable accelerator employing these mechanisms along with support of ISA interface in Chapter 7.

2.3.3 Hardware-Software Interface

Finally, the third and important primitive for programmable accelerator architecture is to have an efficient programming interface by having synergy between the software and hardware primitives. A new ISA interface is needed for programmable accelerator for efficiently mapping the computation and memory access patterns to the hardware. This interface must encode the hardware primitives used for specialization and must express it to the software layer. By using this hardware-software co-design approach, we can get the efficiency of specialized architectures by retaining the programmability or reconfigurability features of general programmable architectures. Figure 2.4 shows a typical arrangement of software stack for three different types of architectures. Domain-specific

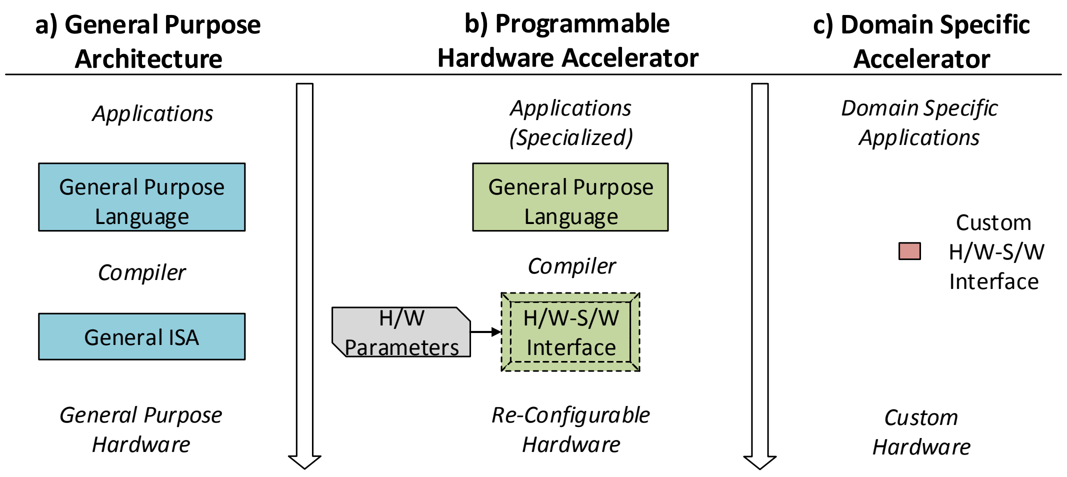


Figure 2.4: Efficient Hardware-Software Interface for Programmable Acceleration

acceleration typically has a very ad-hoc interface and is customized only for that application domain it specializes. Hence, it achieves 10× to 1000× efficiency benefits compared to the most generic interface of general purpose architectures. What we need is an efficient hardware-software or an ISA interface for programmable acceleration which can express common accelerator principles and embodies the execution model by representing the programs to hardware. This also enables the hardware to have much simpler structures without having it to artificially extract the computation or memory access patterns which would be power and area inefficient. This interface must encode multiple operations in fewer instructions in order to achieve high computation ratio for the amount of data it operates on. This is possible only by expressing the hardware primitives to the software interface and mapping the application characteristics directly to hardware mechanisms. We discuss such programming abstractions needed for the stream-dataflow programmable accelerator along with a new accelerator ISA interface in Chapter 6.

2.4 Challenges

In this section, we briefly discuss some key challenges in realizing the programmable hardware acceleration from accelerator modeling, compiler and architecture design perspectives.

2.4.1 Accelerator Modeling

Before realizing the architecture and programming interface for programmable hardware acceleration, it is important to evaluate the primitives discussed in Section 2.3 for typical accelerator applications by constructing simple yet effective analytical and simulation based accelerator models. But this is not trivial as there are no sophisticated accelerator models developed in research community for programmable hardware accelerators. There are modeling tools developed to study the efficiency benefits and perform design space exploration of fixed-function or domain-specific accelerators [29, 35, 36]. But, there exists very limited modeling techniques for exploring programmable accelerator designs. So, we in our research use a framework combined of trace-based simulation [37, 38] and a python based domain-specific modeling approach to evaluate domain-specific designs and extend that to involve the hardware primitives needed for programmable hardware accelerator design. We also rely on the accelerator specific graph-based transformation tool from Nowatzki et. al, [39], to extract out the acceleration phases from the applications and analyze its interaction with host system, memory access patterns and instruction scheduling. The computation phases and memory access patterns of accelerator workloads are generally straight-forward to analyze and we model the specialization hardware primitives and its impact on performance, area and power for each of these phases. We also consider different optimization techniques like loop-unrolling, software-pipelining, vectorization and parallel phases in our modeling framework. Overall, we address the challenge of programmable accelerator modeling by using a simple framework able to capture the impact of specialization primitives on the accelerator application phases and evaluate the efficiency of such designs. We explain the strategy used to model the generic programmable accelerator (GenAccel) in Section 4.3 and detail our evaluation methodology for other domain-specific accelerator models in Section 5.1.

2.4.2 Compilation Techniques

Compiling the application code for a programmable accelerator is quite challenging, as it needs to map the computation and memory access phases from the accelerator workload to the hardware substrate. To achieve that, the compiler needs to be aware of the accelerator's hardware capability and also to find the right mapping of the computation subgraph to the accelerator fabric, it needs extensive modifications. The compiler is also responsible for inserting the communication instructions before and after the accelerator invocation code. We address these compiler challenges by using the traditional decoupled access-execute technique. First, for mapping the computation subgraph to the specialized accelerator hardware we use Integer Linear Programming (ILP) based scheduling technique [40]; of-course this is dependent on what micro-architecture mechanisms are chosen for the computation elements in the accelerator hardware; Second, we rely on compiler intrinsics to generate the communication code to coordinate with the accelerator hardware by also exposing the special ISA interface of the programmable accelerator to the compiler. Thus, any general purpose program can still be mapped to the accelerator with minimum modifications to the compiler. We explain the typical programming interface and compilation technique for a programmable hardware accelerator in Section 4.2.2.

2.4.3 Application Adaptability

As the programmable hardware accelerator exposes a programming interface to the application-level, the workloads need to be modified to suit the new programming abstractions and interface. By doing that, we need to be careful to not modify the accelerator algorithm thus losing the performance benefit we get. Also, we need to analyze what kind of applications suit the accelerator hardware and whether the workload can take advantage of the specialization principles and accelerator hardware mechanisms. Applications can have lot of control-flow, data-parallelism, thread-level parallelism and streaming behaviors and each of these characteristics must be exploited in order to accelerate such workloads. Generally, the specialized architectures find it hard to accelerate irregular memory access workloads, and hence the mechanisms for programmable accelerator need to take care of such applications for them to be adaptable to the new architecture. We discuss the application adaptability and the ease of using the programming interface developed for our programmable accelerator in Section 6.5 and Section 6.6.

2.4.4 Micro-Architecture Design

Once we have identified the micro-architecture mechanisms for specialization principles it is important to consider the design complexity and the overheads it introduces. Extra care need to be taken with how the accelerator substrate is integrated to the host system as modifying the host processor pipeline or cache subsystem increases the design complexity and also makes the verification hard. The functional units also needs special attention in the way they specialize floating point and wide-vector SIMD operations as they consume lot of area and power. Large storage structures or control-heavy structures also introduce power and area inefficiency and micro-architecture design should consider these design decisions into consideration. Chapter 7 has a detailed micro-architecture implementation for the stream-dataflow programmable accelerator and considers all these critical design decisions made to have a highly efficient programmable accelerator hardware.

2.5 Evaluation Approach

To summarize, we propose that programmable hardware acceleration is an essential type of specialization to explore and it is important to evaluate such programmable designs both from the accelerator modeling perspective and also by a more detailed architectural realization. In the upcoming chapters, we provide more details about the primitives we discussed for building a programmable hardware accelerator along with a detailed accelerator architecture for stream-dataflow paradigm. Below we briefly discuss the two evaluation approaches we follow for this dissertation:

2.5.1 Generic Programmable Accelerator (GenAccel) Modeling

We first evaluate the mechanisms implementing the specialization principles by building a generic programmable accelerator model called GenAccel. We detail our GenAccel modeling approach in Section 4.3, but at a high-level it models the hardware primitives chosen to exploit each of the accelerator specialization principle. We use workloads from four application domains which are all single-threaded and have domain-specific implementations. We study the trade-offs of programmable hardware acceleration by comparing our generic programmable accelerator model GenAccel against DSAs from these four diverse application domains and conclude that programmable accelerator architectures like GenAccel can be within 2× - 4× power and area efficiencies of specialized hardware (ASICs/DSAs), while being able to still program/re-configure and not using any domain specific architectural abstractions.

2.5.2 Architectural Realization of Stream-Dataflow Programmable Accelerator

Based on the findings from modeling results and the GenAccel model evaluation, we believe that it is important to validate those results by realizing a more detailed architecture alongside the hardware-software interface. As introduced in Chapter 1, we propose a detailed architecture for a programmable hardware accelerator by studying an instance of programmable acceleration called stream-dataflow acceleration. The stream-dataflow paradigm consists of an architecture with a set of programming abstractions, an execution model, ISA interface and a detailed micro-architecture design of a programmable hardware accelerator. We evaluate this architecture by using a Chisel [41] based hardware prototype implementation of the stream-dataflow accelerator along with a complete software stack implementation comprising computation subgraph scheduler, cycle-level performance simulator and compiler toolchain. We consider a class of applications involving regular streaming memory access patterns with computationally intensive dataflow execution phases for domain-specific evaluation of the stream-dataflow accelerator. To distill the limitations of the stream-dataflow programmable accelerator design and to perform application-specific evaluation, we also consider a broader set of typically-accelerated workloads consisting of irregular memory accesses and complex computation patterns. We evaluate the performance of stream-dataflow accelerator using a RISCV based cycle-level performance simulator compared to state-of-the art domain-specific and application-specific accelerator designs. and evaluate the power, area efficiency using the synthesized RTL design. We provide more details of this architecture implementation and evaluation in Chapter 8.

3 | Principles of Architectural Specialization

Domain specific accelerator (DSAs) achieve efficiency through the employment of five *common* specialization principles. In this chapter, we first discuss those common principles of specialization in detail (Section 3.1) and then contrast the application of these principles with the mechanisms and algorithms used in four prominent and diverse DSAs (Section 3.2). We have chosen these four DSAs as we believe they are representative of the application domains they are targeting and all these domains are very important and relevant for current technology trend. Broadly, we see this as a counterpart to the insights from Hameed et. al [3]. While they describe the sources of inefficiency in a general purpose processor, we explain the sources of potential efficiency from specialization.

3.1 Defining the Principles of Specialization

In this section, we now define the five common principles of architectural specialization found in most of the DSAs based on the accelerator application analysis. Figure 3.1 shows the five principles and we discuss the potential area, power, and performance trade-offs of targeting each in a programmable accelerator design.

3.1.1 Concurrency Specialization

The concurrency of a workload is the degree to which its operations can be performed in parallel and simultaneously. Specializing for a high degree of concurrency means organizing the hardware to perform work in parallel by favoring lower overhead hardware structures. Examples of specialization strategies include employing many independent processing elements with their own controllers or using a wide vector model with a single controller.

Applying hardware concurrency increases the performance and efficiency of parallel workloads, while increasing power, at a close-to-linear ratio while parallelism exists.

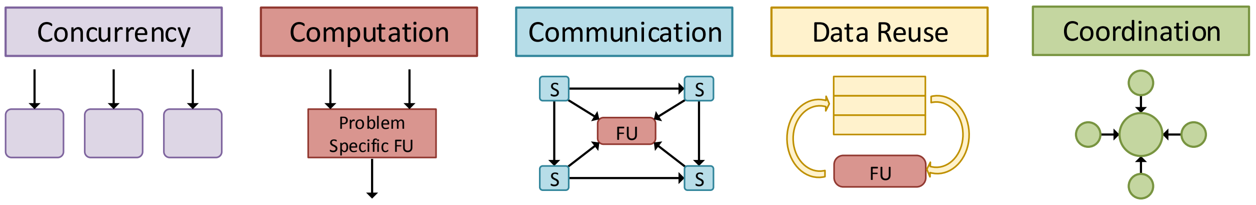


Figure 3.1: Common Principles of Architectural Specialization

3.1.2 Computation Specialization

Computations are individual units of work in a workload algorithm. The hardware which performs this work are functional units (FUs) – which typically take few inputs and produce a single output. Specializing *computation* means creating problem-specific FUs. For instance, a custom `sin` FU would much more efficiently compute the sine function than iterative methods on a general purpose processor. It is arguable whether certain FUs can be considered *specialized*. There is no set answer here and in this work we rely on the intuition that FUs that are not typically found in a general purpose processor should be considered specialized. Specializing computation improves performance and power by reducing the total work. We note some computation specialization can be problem-specific. However, commonality between and inside domains is almost guaranteed.

3.1.3 Communication Specialization

Communication is the means of transmission of data values and control information between and among storage and functional units. Specialized communication is simply the instantiation of communication channels, and potentially buffering the data elements between the hardware units to ultimately facilitate a faster operand throughput to the FUs. This reduces power by lessening access to intermediate storage like register-files or local memory, and potentially area if the alternative is a general communication network. One example is a broadcast network for efficiently sending immediately consumable data to many compute units or a shuffling network to exploit more complex mappings of producers to consumers. This type of explicit communication is also very efficient if the workloads have lot of producer-consumer relationship.

3.1.4 Data-Reuse Specialization

Data-reuse is an algorithmic property where intermediate values are consumed multiple times. The specialization of data reuse means using custom storage structures for these temporaries. To be more precise, algorithmic reuse can be exploited by direct communication when values can be directly communicated from producer to consumer. Specializing reuse benefits performance and power by avoiding the more expensive access to a larger global memory or register files. A classic example of reuse specialization is caches. In the context of accelerators, access patterns are often known a priori, meaning that low-ported, wide scratchpads are most effective as caches spend energy in banking, tag lookup and wires. Accelerators also specialize the reuse of constant values, either through read-only memories or FU-specific storage, if the constants are specific to one static operation. Streaming buffers used in accelerators is one more example for efficiently storing memory which is accessed through DMA. Indeed, the relationship between communication, computation, and reuse specialization is non-trivial. The presence of better or more computational resources can potentially obviate the need for reuse specialization, and the properties of specialized communication channels (e.g. width) must be often co-designed with memory structures for reuse.

3.1.5 Coordination Specialization

Hardware coordination is the management of hardware units and their timing to perform the overall work. Instruction sequencing, command dispatch, control flow, signal decoding and address generation are all examples of coordination tasks. Specializing it usually involves the creation of small finite state machines to perform each task, rather than relying on a brawny general purpose processor and out-of-order instruction scheduling. Performing more coordination specialization typically means less area and power compared to something more programmable, at the price of generality.

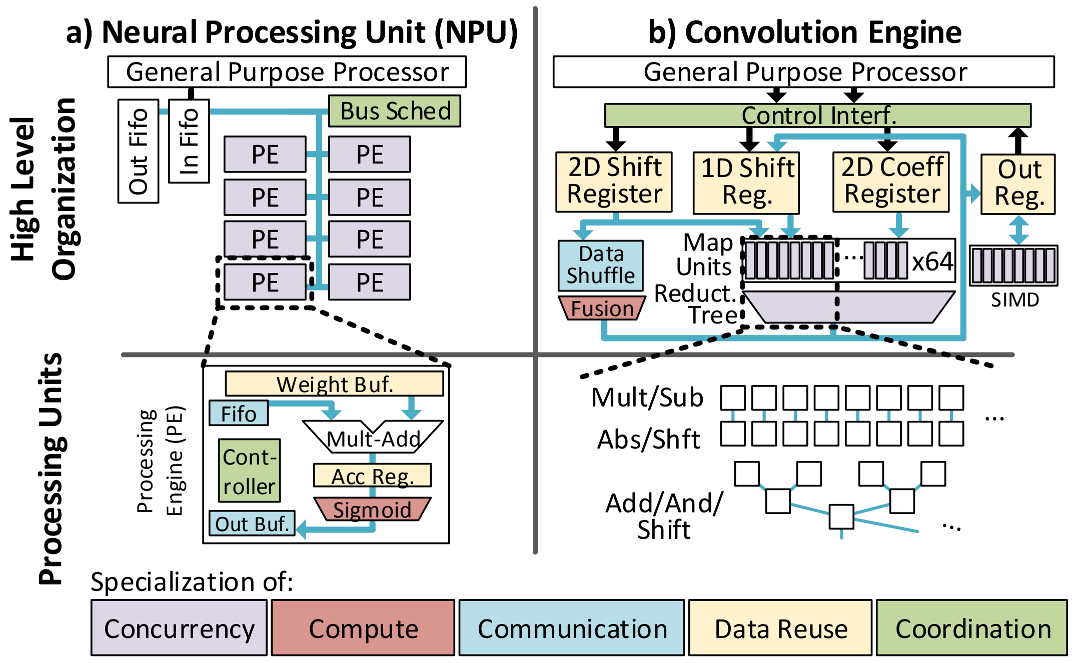


Figure 3.2: Application of Specialization Principles in NPU and Convolution Engine DSAs

3.2 Relating Specialization Principles to Accelerator Mechanisms

In this section, we contrast the specialization principles discussed with four prominent and diverse domain specific accelerators. Appendix A describes all the four DSAs and their architecture in more detail. Figure 3.2 and 3.3 depicts the block-diagrams of the four DSAs belonging to four different application domains that we study. In the figure, the colors indicate the types of specialization for each component. The specialization mechanisms and algorithmic properties exploited are summarized in Table 3.1, and are explained below.

3.2.1 Neural Processing Unit (NPU) for Neural Approximation

NPU [42] shown in Figure 3.2(a) is a DSA for approximate computing using the neural network algorithm, integrated to the host core through a FIFO interface. NPU is designed to exploit the concurrency of each level of a neural network, using parallel processing entities (PEs) to pipeline the computations of eight neurons simultaneously. NPU specializes data-reuse with accumulation registers and per-PE weight buffers. For communication, NPU employs a broadcast network specializing the large fan-outs of the neural network, and specializes computation with sigmoid FUs for activation of output neurons. A bus scheduler and PE controller are used to specialize the hardware coordination.

3.2.2 Convolution Engine for Image Processing

Convolution Engine [43] shown in Figure 3.2(b) accelerates stencil-like computations for image processing applications. The host core uses special instructions to coordinate control of the hardware through a control interface. It exploits concurrency through both wide-vector and pipeline parallelism and relies heavily on reuse specialization by using custom memories/registers for storing pixels and coefficients. In addition, column and row interfaces provide shifted versions of intermediate pixel values. These, along with other wide buses, provide communication specialization. Finally, it specializes reduction computation using a specialized graph-fusion unit.

3.2.3 Q100 for Database Streaming

Q100 [44] shown in Figure 3.3(a) is a DSA for streaming database queries, which exploits the pipeline concurrency of database operators and intermediate outputs. To support a streaming model, it uses the stream buffers as staging area to prefetch the required database columns. Q100 specializes the communication by providing dynamically routed channels between FUs to prevent memory spills. Finally, Q100 relies on custom database FUs like Join, Sort, and Partition. Reuse specialization happens inside these FUs when storing constants (ALUs, filters, partitioners) and reused intermediates (aggregates, joins). The communication network configuration and stream buffers are coordinated using an temporal instruction sequencer.

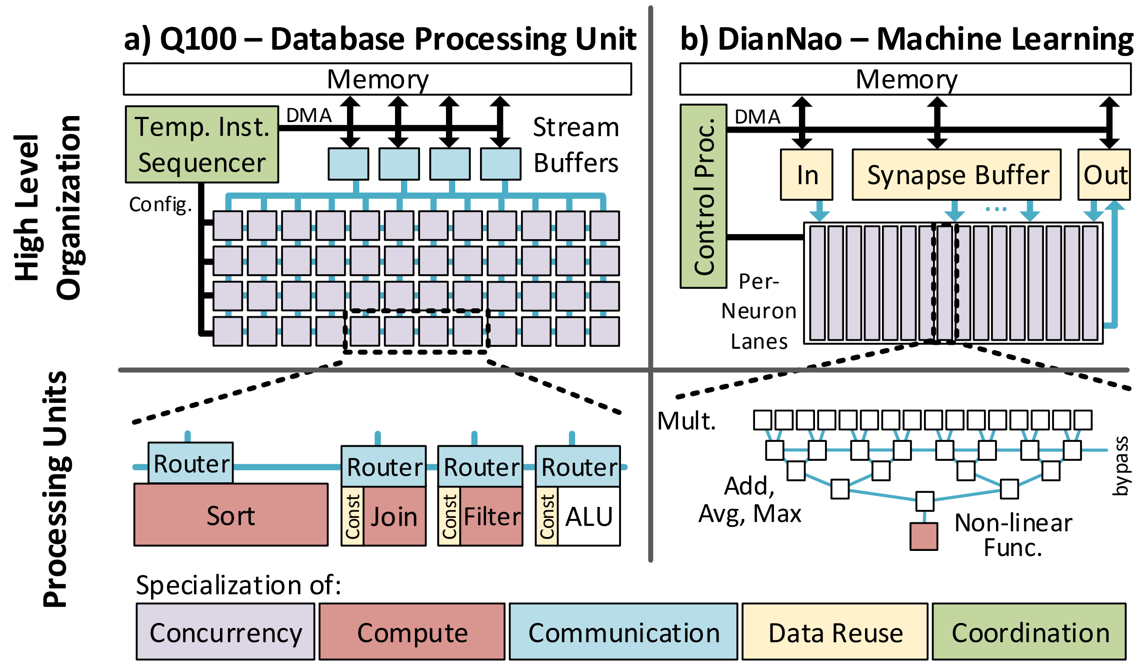


Figure 3.3: Application of Specialization Principles in Q100 and DianNao DSAs

3.2.4 DianNao for Deep Neural Networks

Diannao [15] shown in Figure 3.3(b) is a DSA for deep-learning neural networks. It achieves concurrency by applying a very wide vector computation model (multiply-accumulate), and uses wide memory structures (4096-bit wide SRAMs) for reuse specialization of neurons, accumulated values and synapses/weights. DianNao also relies on specialized sigmoid FUs for activation of output neurons. Point-to-point links between FUs, with little bypassing, specializes the communication. A specialized control processor is used for coordination.

We believe that these principles are not an exhaustive list to capture the specialization mechanisms of all DSAs, but based on our analysis and observation, these principles are common in many more such DSAs belonging to the same application domains.

Principle		**NPU** [42]	**Convolution Engine** [43]	**Q100** [44]	**DianNao** [15]
Computation	Alg.	Sigmoid computation	Graph fusion	Sort, Partition, Join...	Transfer functions
	HW	Problem specific special function units			
Communication	Alg.	All-to-all neuron communication between layers	Wide-vector + shuffling + reduction	Streaming accesses between operators	Wide-vector + reduction
	HW	Broadcast network between processing entities	Wide buses + shuffle and reduction network	Buffered inter-operator routers	Point-to-point links + reduction network
Data Reuse	Alg.	Neural weights	Reuse of shifted values	Constants	Synapse accumulation
	HW	Per-PE weight buffers	1D/2D Shift Registers	FU Constants Storage	Synapse scratchpads
Coordination	Alg.	Sequencing of algorithmic steps			
	HW	Bus Scheduling/PE Control	Host + Control Interface	Instruction Sequencer	Control Processor
Concurrency	Alg.	Intra-layer neuron independence	Independence of pixels + stencil computations	Streaming queries + operator independence	Independent neurons inside + across layers
	HW	Per-neuron independent PEs	Wide-vector + Pipelining	FU Array + Pipelining	Wide-vector + Pipelining

Table 3.1: Specialization Principles and Mechanisms
(Alg: Algorithmic insight, HW: Hardware implementation)

3.3 Chapter Summary

In this chapter, we described the five common principles of specialization found in most of the DSAs. We then contrasted each of the principle and their application in the four DSAs we studied. For the programmable hardware accelerator to achieve the benefits of specialization as DSAs, it needs to employ these principles.

4 | Mechanisms for Programmable Acceleration

Our primary insight as described in Chapter 2 is that well-understood mechanisms can be composed to target the same specialization principles that domain specific accelerators (DSAs) use, but in a programmable fashion. In this chapter, we first explain the general set of mechanisms in Section 4.1.2 that can be used to exploit the specialization principles detailed in Chapter 3. We also describe the design of a generic programmable accelerator (GenAccel), and highlight how it performs programmable acceleration. We then briefly describe how to use such a design in practice over the phases of its life-cycle in Section 4.2. In Section 4.3, we then explain the modeling of the GenAccel design using the basic building blocks of implementation along with the strategy used for performance, power and area models. In Section 4.4, we describe two important design points of correctly provisioned GenAccel model for four application domains, which will be used in the evaluation. Finally, we end the chapter by discussing the other alternative choices that can be used for programmable acceleration mechanisms.

4.1 Design of a Generic Programmable Accelerator

Before describing the mechanisms in detail, we briefly discuss the requirements for the design of a programmable accelerator below. We then delve into the mechanisms based on the requirements and also describe the *GenAccel* design which is an abstract design capturing all the specialization principles mechanisms.

4.1.1 Design Requirements

At a high-level, we outline three intuitive requirements for a programmable accelerator design:

1. **Is programmable:** While some DSAs provide limited programmability, this is usually appli-

cable only for a specific domain they are specialized to. A generic programmable accelerator architecture provides domain-agnostic programmability and quick trivial adaption to any new domain being mapped.

2. **Applies specialization principles:** For achieving energy and performance efficiency competitive with DSAs, they must apply the specialization principles described in Chapter 3 in a general and domain-agnostic way.

3. **Design is parameterizable:** At design/synthesis time, we must provide a parameterizable interface to allow customizing the aspects of the design and to meet the combined requirements of the targeted domains.

4.1.2 Mechanisms for *GenAccel* Design

As we will describe, it is possible to construct an accelerator design which embodies the specialization principles by selecting a set of mechanisms from well known techniques. This is not the only possible set of mechanisms, but they are a simple and effective set commonly found in many architectures.

The most critical principle is exploiting *concurrency*, of which there is typically an abundant amount when considering acceleratable workloads. The requirement of high concurrency pushes the design towards simplicity, and the requirement of programmability implies the use of some sort of programmable core with a small cache. The natural way to fill these combined requirements is to use an array of tiny low-power cores, which communicate through memory. Also in terms of supporting virtual memory support TLB's are a source of energy inefficiency, which DSAs typically sidestep. We imagine that programmable accelerator architectures will support limited virtual memory through large pages and small TLBs or using simple base-bound based translation. This is a sensible trade-off as commonly-specialized workloads exhibit little communication and synchronization between the coarse-grained parallel units. The main benefit also here is natural programmability, using standard programs, with an added benefit that known irregular memory accesses can benefit from using the cache. We call each of the concurrent structures with the low-power core and all other components (described later) as one "GenAccel" Unit. Also, using many units having this core, as opposed to a wide-vector model with a single controller, has a flexibility advantage. When memory locality is not possible, parallelism can still be achieved through multiple execution threads. The remainder of the design is a straight-forward application of the remaining specialization principles for each of the units.

Achieving *communication* specialization of intermediate and computed transient values requires an efficient distribution mechanism for operands that avoids expensive intermediate storage like multi-ported register files. Arguably, the best known approach is an explicit routing network, which is exposed to the ISA to eliminate the hardware burden of dynamic routing. This property is what defines spatial architectures (as well as the early explicit-dataflow machines that inspired them [45]), and therefore we add a spatial fabric/architecture as our first mechanism. This serves three additional purposes. First, it is an appropriate place to instantiate problem-specific custom functional units to achieve *computation* specialization. Second, it accommodates the *data-reuse* of constant values associated with specific computations by instantiating simple flip-flops at each FU. Third, when coarse-grained parallelism of concurrent thread executions with multiple GenAccel units is not possible, independent instruction-level computations in spatial fabric of each GenAccel unit benefits *concurrency* at instruction-level.

To achieve communication specialization with the global memory, a natural solution is to add a Direct-Memory Access (DMA) engine and a configurable local-storage like scratchpad, with a wide-vector interface to the spatial architecture. The scratchpad, configured as a DMA buffer, enables the efficient streaming of data values from memory by decoupling memory access phases from the execute/computation phases of spatial architecture. This mechanism of decouple access-execute has in-turn been explored in traditional architectures [46] and recently in some of the spatial-dataflow architectures [47]. When configured differently, the scratchpad can act as a reuse buffer to achieve

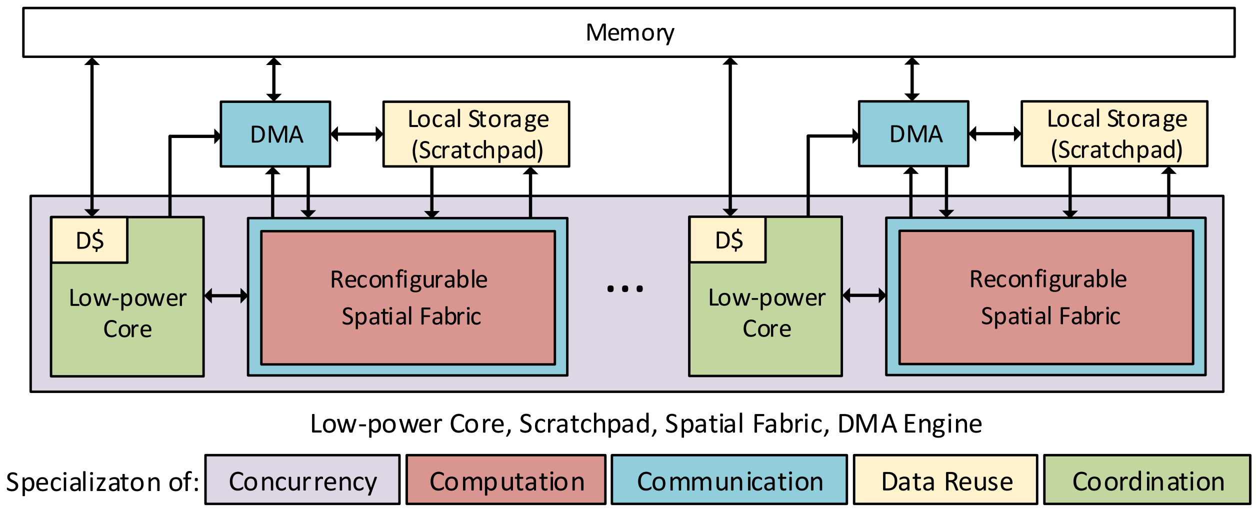

Memory
DMA
Local Storage (Scratchpad)
D$
Low-power Core
Reconfigurable Spatial Fabric
...
DMA
Local Storage (Scratchpad)
D$
Low-power Core
Reconfigurable Spatial Fabric
Low-power Core, Scratchpad, Spatial Fabric, DMA Engine
Specializaton of:
Concurrency
Computation
Communication
Data Reuse
Coordination

Figure 4.1: Mechanisms Used in Modeling the Generic Programmable Accelerator (GenAccel) Design

data-reuse specialization. In either context, a single-ported SRAM-based scratchpad is enough, as access patterns are usually simple and known ahead of time in accelerator workloads. Spatial fabric also has an input-output interface to support the wide-vector interface to scratchpad and DMA.

Even with the above mechanisms and four specialization principles satisfied, we are still not adequately addressing the generality and efficiency requirements from two perspectives. A spatial architecture which is designed to be general needs to hold significant state and perform all possible functionality and is unattractive because this generality adds too much complexity in terms of area and power. Secondly, the lack of a traditional cache has severe efficiency consequences for workloads with any kind of irregular memory access. We address these shortcomings by explaining the last and important mechanism to exploit the co-ordination specialization principle. In a programmable hardware accelerator, we require an efficient mechanism for **coordinating** the above hardware units like for example configuring the spatial architecture or synchronizing DMA with the computation in each GenAccel unit. Again, here we propose relying on the simple core, as this brings a huge programmability and generality benefit. Furthermore, the cost is low – if the core is low-power enough, and the spatial architecture is large enough, the overheads of co-ordination can be kept low.

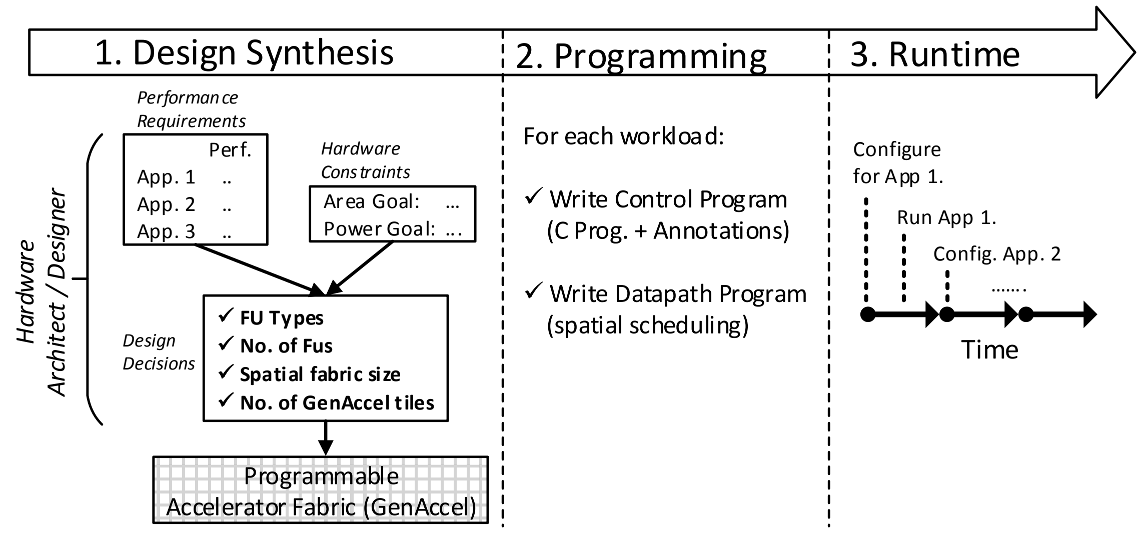


Figure 4.2: Phases of GenAccel's Design and Use

Generally speaking, this low-power core takes the place of the state machines in a DSA, serving as a memory fetching engine and as a fall-back for irregular computations which are not suited for the spatial architecture.

To summarize the generic programmable accelerator (**GenAccel**) design, each unit contains a – *Low-power core*, a flexible and reconfigurable *Spatial architecture*, a programmable local-storage like *Scratchpad* and a *DMA engine* as shown in Figure 4.1. This design satisfies the aforementioned requirements: programmability, efficiency through the application of specialization principles, and simple parameterizability. We believe that this design is simple enough through the integration of a small number of architectural building blocks. Note that, we do believe that there are other different baseline architectures that can be used to evolve into a programmable hardware accelerator by applying the same principles and is discussed in the Section 4.5.

4.2 GenAccel Usage

We now explain the typical usage of the GenAccel design and the advantages of a parameterizable accelerator enabling to perform accelerator design space exploration in large. Preparing the GenAccel fabric for use occurs in three phases, as shown in the Figure 4.2.

1. Choose the most general single-output problem-specific FUs which suits the algorithm.
2. Determine the total number of FU resources required to attain the desired throughput.
3. Divide FUs into groups, where each group accesses a limited amount of data per cycle, excluding per-instruction constants. We use 64 bytes maximum per-cycle (cache line size), resulting in a reasonable scratchpad line-size usable by many algorithms. The number of these groups is the number of GenAccel units.
4. If the algorithm has reuse within a small working set, then size the number of SRAM entries to match that.

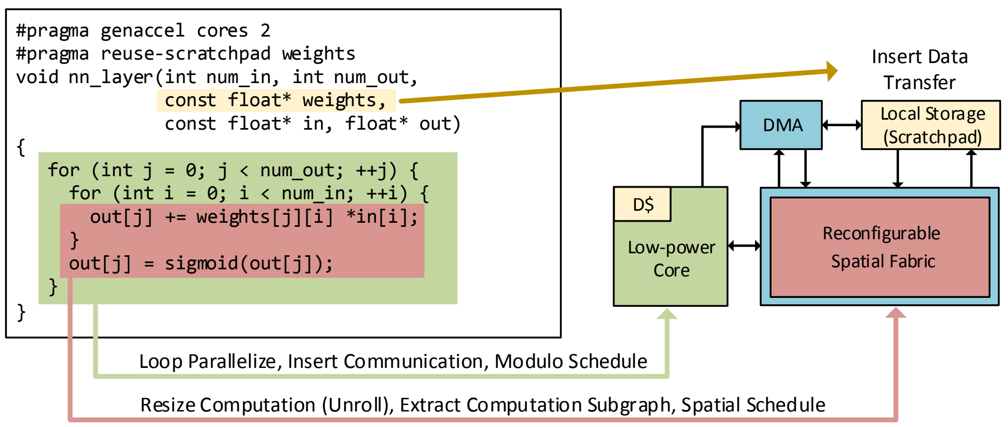
Figure 4.3: Procedure for provisioning each GenAccel fabric and number of GenAccel units

4.2.1 Design Synthesis

For specialized architectures, *design synthesis* is the process of provisioning an architecture for given performance, area and power goals. It involves examining one or more workloads or workload domains, and choosing the appropriate functional units, the datapath size, the scratchpad sizes and widths, and the degree of concurrency exploited through multiple instantiations of the GenAccel fabric. In GenAccel, this phase involves the selection of its hardware parameters to suit the chosen workload domain(s) performance, area and power constraints [1]. This is usually done by the hardware architects or designers, before the design closure of the GenAccel fabric which is provisioned and synthesized on chip to execute the chosen application domain(s).

Though many optimization strategies are possible, in this work we consider the primary constraint of the programmable architecture to be "performance" – i.e. there exists some throughput target that must be met by the design, and the overall power and area should be minimized, while still retaining some degree of generality and programmability. Figure 4.3 outlines a simple strategy used to provision GenAccel design for an application or domain(s). GenAccel fabric has a small number of tunable/configurable parameters – total number of units, width and size of the scratchpad, FU mix, layout and the size the spatial architecture. In the context of a real design, GenAccel's

[1]Our treatment of synthesis-time configurability is reminiscent of architectures like Smart Memories [48] and Custom Fit Processors [49], and we revisit these in the related work.



```
#pragma genaccel cores 2
#pragma reuse-scratchpad weights
void nn_layer(int num_in, int num_out,
              const float* weights,
              const float* in, float* out)
{
  for (int j = 0; j < num_out; ++j) {
    for (int i = 0; i < num_in; ++i) {
      out[j] += weights[j][i] *in[i];
    }
    out[j] = sigmoid(out[j]);
  }
}
```

Insert Data Transfer
DMA
Local Storage (Scratchpad)
D$
Low-power Core
Reconfigurable Spatial Fabric
Loop Parallelize, Insert Communication, Modulo Schedule
Resize Computation (Unroll), Extract Computation Subgraph, Spatial Schedule

Figure 4.4: Example GenAccel Program and Compiler Passes
(Arrows labeled with required compilation passes)

synthesis-time parameters would be chosen to maximize performance of several workloads under area and power constraints. Of course, if multiple application domains has to be targeted, then the superset of the above hardware features should be chosen. Section 4.4 discusses the design points chosen to match the workload domains studied in this work.

4.2.2 Programmability of GenAccel

This phase involves programming GenAccel accelerator fabric using a typical general purpose language programming interface. We model the programmability of GenAccel by generating the control program and the spatial datapath configuration to carry out the computation of the algorithm. In order to model the hardware-software interface for programmability of GenAccel, it must involve two major components – i) creation of the coordination or control code for the low-power core to perform unrolling, parallelizing and coordinating the accelerator components; and ii) generation of the configuration data for the spatial datapath to execute the computation subgraph and to match the available hardware resources. Programming for GenAccel in intrinsic assembly may be reasonable because of the simplicity of both the control and data portions. In practice, using standard languages like OpenACC [50] or C++AMP with `#pragma` annotations, or even languages like OpenCL [51] kernel style programming would likely be effective.

Though we do not explain the compiler implementation and programming model details in this initial work of accelerator modeling (See Section 5.1 for our modeling approach), for illustrative purposes, Figure 4.4 shows an example annotated code for computing a simple neural network layer, along with a provisioned GenAccel. This code computes one neural approximation layer for NPU. The figure also shows the compilation steps to map each portion of the code to the accelerator fabric. At a high level, the compiler will use annotations to identify arrays for use in the scratchpad SRAM (either as a stream buffer or scratchpad) and also number of GenAccel units needed for the concurrent execution. In the example, the weights can be loaded into the scratchpad, and reused across neural network invocations. Subsequently, the compiler will unroll computations and create a large-enough datapath to match the resources present in the spatial fabric, which could be spatially scheduled using known techniques [40]. Communication instructions would be inserted into the main coordination code, as well as instructions to control the DMA access for streaming inputs into the scratchpad or spatial fabric input interface, and also to drain out the data out of spatial fabric to memory or scratchpad. Finally, the coordination loop would be modulo scheduled to effectively pipeline the spatial architecture.

4.2.3 Runtime Configuration

Finally, the GenAccel need to be configured at runtime between the execution of multiple applications or domains. The configuration is switched for execution of the computation of each application or accelerator kernel. Table 4.1 highlights the distinction of synthesis-time and run-time configuration parameters for the GenAccel fabric.

4.2.4 GenAccel Integration to Host System

Before describing the modeling strategy and design point selection of GenAccel for chosen example application domains, we briefly discuss about GenAccel's interface to the host system and benefits of coarse-grained versus fine-grained integration. At the time of design-synthesis, there are funda-

	Synthesis-Time	Run-Time
Concurrency	Number of instantiated GenAccel units to map multiple application(s) and domain(s); Size of spatial datapath; SIMD width of functional units	Power-gating unused GenAccel units during execution
Computation	Spatial architecture functional unit (FU) mix	Scheduling of spatial architecture and low-power core
Communication	Spatial datapath and SRAM interface widths	Configuration of spatial datapath and input/output ports
Data Reuse	Scratchpad (SRAM) size and line-size	Use of Scratchpad as DMA or reuse buffer

Table 4.1: Configurable Parameters for GenAccel Fabric

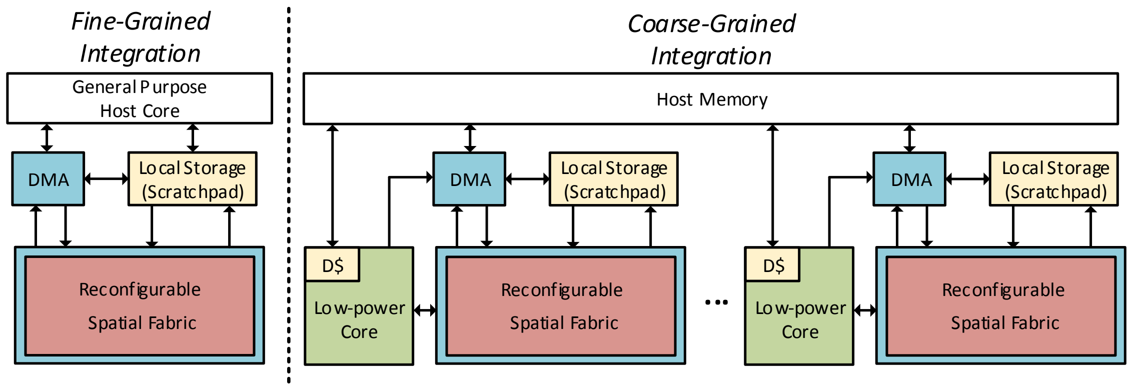

Fine-Grained Integration
General Purpose Host Core
DMA
Local Storage (Scratchpad)
Reconfigurable Spatial Fabric
Coarse-Grained Integration
Host Memory
D$
Low-power Core
DMA
Local Storage (Scratchpad)
Reconfigurable Spatial Fabric
...
D$
Low-power Core
DMA
Local Storage (Scratchpad)
Reconfigurable Spatial Fabric

Figure 4.5: Integration of GenAccel to Host System

mentally two different ways to integrate the specialization fabric, and GenAccel is suitable for both (with some caveats) as shown in Figure 4.5. In coarse-grained integration, each of many GenAccel units along with low-power core has access to the host's memory system, and does not interact with host core itself. Host core can be power-gated after offloading GenAccel application kernels.

Contrastingly, acceleration is often profitable at a finer granularity, and it becomes beneficial to explore integration with the host processor's caches. For example, if acceleration happens at a granularity of tens of cycles, then the overhead of memory transfer every time is prohibitive. To minimize these overhead in fine-grained integration, we let the host core serve the role of low-power core and drop the superfluous low-power core.

4.3 Modeling the GenAccel Design

In this section, we describe the strategy used to model the generic programmable accelerator design (GenAccel). We first delve into the basic building blocks used for GenAccel model implementation and then discuss performance, power and area modeling of GenAccel.

4.3.1 Building Blocks of GenAccel Implementation

We use several existing components, both from the literature and from available designs, as building blocks for the GenAccel fabric. The spatial architecture we leverage is a modified version of DySER coarse-grained reconfigurable architecture (CGRA) [52], which is a lightweight statically-routed mesh of FUs. To avoid the power overheads of credit-based packet switched routers of DySER, we instead consider the circuit-switched statically routed switches. We believe this does not affect the performance of our spatial fabric as the static spatial schedule generated for the CGRA considers the amount of data needed for each instance of the computation and the resources available. Also, this statically scheduled CGRA provides a throughput oriented dataflow computation without the power overheads and design complexity of a traditional dynamically routed CGRA like DySER. Note that we will use CGRA and "spatial architecture" interchangeably henceforth. The spatial fabric belongs to a family of similar architectures like CCA [53], BERET [54], SGMF [55] and others in how they specialize the communication, concurrency and data-reuse. This choice is a good trade-off between efficiency and configuration time. Because of the CGRA's network organization, we use only FUs which have single-outputs and low-area footprints. It also supports a vector interface with configurable mapping between offset of vector elements and spatial architecture ports, which enables more efficient irregular memory access patterns.

The processor core we leverage is a Tensilica LX3 [56], which is a simple VLIW based core-design featuring a low-frequency (1GHz) with 7-stage deep pipeline. This core is chosen because of its very low area and power footprint (0.044 mm^2 and 14mW at 1GHz, 45nm), and is capable of running irregular code. An equally viable candidate is the similarly smaller core like Cortex M3 [57].

4.3.2 Modeling GenAccel Performance

The insight we use for estimating performance is that GenAccel design is quite simple and the targeted workloads have straightforward memory access and computation patterns, making execution time straightforward to capture. Our strategy uses a combined PIN [58] based trace-simulator and application-specific modeling framework to capture the role of the compiler and the GenAccel program execution. The framework is parameterizable for different FU types, concurrency parameters (SIMD width and GenAccel unit counts), reuse and communication structures.

Our framework considers execution stages, each corresponding to a particular computation or kernel. For each workload, we customize the general model to account for the specific data amount, types of computation, available parallelism, working-set sizes and global memory accesses required by each phase. Kernels in NPU are individual neural network layers, while Convolution Engine has scratchpad-load and map/reduce computation phases. Each Q100 stage is a compound database operation like the "temporal instruction" as they call, and DianNao has convolution, classifier and pooling phases. All of these have significant parallelism, are generally long running, non-overlapping, and contain many pipelined computations. The latency of each phase is either bound by the memory bandwidth, instruction execution rate (considering FU contention and issue-width), cache or scratchpad ports, cache or scratchpad fill rate and the critical path latency.

4.3.3 Modeling GenAccel Power and Area

For the LX3 core parameters, we rely on published datasheets [56]. For the integer FUs, we use the estimates from the DianNao work [15], and for floating point FUs we use DySER implementation's [59, 52] synthesized estimates. To estimate CGRA-network power, we synthesized RTL of a router using a 55nm ARM library and did a sanity check against the Orion [60] network model. We obtained the vector interface estimates from DySER work again. SRAMs for data-reuse and buffering were estimated using CACTI 6.5 [61] and McPAT [62]. Specialized FU properties (Sigmoid, Paritioner, Sort etc.) were taken from the original works.

4.4 Design Point Selection and Provisioning GenAccel

We now briefly describe the GenAccel design points that we study in this work for evaluation, with their rationale, and how workloads map to them.

4.4.1 Synthesized GenAccel Designs and Program Mapping

For our evaluation, we consider synthesized GenAccel designs for four application domains that we target – Neural Approximation, Deep Neural Networks, Image Processing and Database Querying. We provision GenAccel similar to the corresponding application domain's DSA in order to achieve the same throughput with performance being the primary optimization target. For the purposes of evaluation in Section 5.2, we consider two different design scenarios:

- **Domain-provisioned GenAccel** – programmable accelerator's resources target a *single* application domain;
- **Balanced GenAccel** – programmable accelerator resources target many different application domains.

Both sets of design points are attained using the procedure explained in Figure 4.3 and are characterized in Table 4.2. We now discuss in detail how each of the design point is selected, synthesized and provisioned for different application/workload or domain chosen.

GA$_N$ for NPU's Neural Approximation Domain

We provision GA$_N$ to meet NPU's performance by including the same datapath size to perform 8 neuron computations in parallel and hence 8 32-bit multipliers, adders, and sigmoid units. As alluded to in the neural network layer example (Figure 4.4), the read-only weights are stored in scratchpad, and since we need to read a maximum of 8 32-bit words per cycle (256-bit total), the SRAM line size becomes 256-bit. We use a 2K entry SRAM for the weight re-use and the size is based on the average re-use distance between multiple invocations of neural network layer. For the

Name	Equivalent DSA	Concurrency	Computation	Communication	Data-Reuse
GA$_N$	NPU	24-Tile CGRA (8 Add, 8 Mul & 8 Sigmoid); Single GenAccel Unit	2048 x 32-bit Sigmoid Lookup Table	32-bit CGRA; 256-bit SRAM interface	2048 x 32-bit Weight Buffer
GA$_C$	Convolution Engine	64-Tile CGRA (32 Mul/Shf, 32 Add/Logic); Single GenAccel Unit	Standard 16-bit units	16-bit CGRA; 512-bit SRAM interface	512 x 16-bit SRAM for image inputs
GA$_Q$	Q100	32-Tile CGRA (16 ALU, 4 Aggregate, 4 Join); 4 GenAccel Units	Aggregate Unit + Filter Unit	64-bit CGRA; 256-bit SRAM interface;	SRAMs for buffering
GA$_D$	DianNao	64-Tile CGRA (32 Mul, 30 Add, 2 Sigmoid); 8 GenAccel Units	Piecewise linear sigmoid FU	16-bit x 2 CGRA; 512-bit SRAM interface;	2048 x 16-bit Weight Buffer
GA$_B$	Balanced	64-Tile CGRA (combination of above); 8 GenAccel Units	Combination of above FUs	64-bit CGRA; 512-bit SRAM interface;	(2048 entries x 16-bit) 4KB SRAM

Table 4.2: Configuration of GenAccel (GA) Design-Points Provisioned for each Application Domain (Add: Adder, Mul: Multiplier, Shf: Shifter)

co-ordination program, we consider different implementations of the neural network layer, which vectorize across either input or output neurons, depending on which is larger.

GA$_C$ for Convolution Engine's Domain

The primary operation in convolution engine is a map operation (on the image inputs) followed by reduce, with shifted version of input pixels. We assume 32 16-bit multipliers/subtractors for the map stage, and 32 16-bit ALUs for the reduction (Although Convolution engine uses a more custom 10-bit datapath, we use 16-bit as it is general.) The input shift can be performed internally by the CGRA network, which allows input splitting. To feed these FUs, we only need 512-bits of data per cycle, meaning we can use one 512-bit wide SRAM and a single GenAccel unit.

GA$_Q$ for Q100's Streaming Database Queries

To target Q100's streaming query workloads, we start with some of the same database query-oriented functional units, like `Filter` and `Aggregate`, for which we include the same number as the original.

However, there are several functional units which do not fit well into the type of CGRA we propose using, which we outline next.

First, Q100's Join unit requires data-dependent data-consumption, and adding this capability is non-trivial as it complicates the CGRA itself as well as its communication with the DMA engine (increasing the CGRA's area). Also, Q100's Partitioner and Sorter, used in tandem for fast sorting[2], do not fit naturally into the paradigm of our spatial fabric. The Q100-Partitioner produces output-pairs (destination-bucket,value) and is quite large ($0.94mm^2$ at 32nm [44]). The Q100-Sorter sorts 1024 elements simultaneously and also has a large area footprint ($0.19mm^2$). Therefore, we do not include the above Q100 units, and instead employ the low-power core for executing the same algorithm. Unlike for the other domains, the mismatch in FUs means that we had to size the number of cores and GenAccel units empirically to match Q100, which turned out to be 4.

Programming the Q100 unit in practice would likely be different than the others as well, as it would need to be integrated into some database management system (DBMS). The DBMS will need to generate a query plan which is composed of subgraphs containing streaming operators (or temporal instructions in Q100's terminology). Each one of these operator subgraphs can be converted to an GenAccel program by applying standard code-generation techniques.

GA$_D$ for DianNao's Deep Learning Domain

DianNao's fundamental operation is a parallel multiply-reduce followed by an optional non-linear function. To match DianNao's performance, GA$_D$ includes 256 16-bit multipliers, 240 adders, and 16 non-linear sigmoid units, which are split up among 8 cores or units. It includes a 512-bit SRAM interface to feed the 32 multiplier per unit. Each unit includes a 2K entry 16-bit SRAM, primarily used for streaming in neural weights (16-bit each), and the neurons are stored in the core's cache.

[2]We assume Q100 uses a "sample sort", which first range-partitions into 1024 size buckets, then sorts each bucket.

GA$_B$ Balanced Design

To create a more balanced programmable accelerator design with generality across the above four domains, we consider a design point which can achieve the same performance goals as each DSA. Essentially, we create this design point by taking the largest design (GA$_D$ in this case) (or generically a superset if more domains considered), and distributing the necessary FUs and storage elements among the remaining units, creating a homogeneous GenAccel. This design point enables the study of how increased programmability affects the area and power of balanced GenAccel design. Note, that a balanced design need not include all the application domains in a single programmable architecture, but an intelligent division of multiple domains into many balanced design points creating heterogeneous GenAccel designs. We have not considered such design for our current work as evaluate only four application domains, but we believe the results still hold good for heterogeneous GenAccel design.

4.5 Evolution Towards Programmable Acceleration

The approach we took in designing GenAccel in Section 4.1 was to start with a concurrent architecture and evolve it by applying specialization principles with the micro-architectural mechanisms. Here we explain how perhaps equally effective and plausibly similar designs could be arrived at by starting with different baseline architectures and applying the same principles.

Large Cores: One possible path forward is to begin with traditional, large out-of-order cores, and add mechanisms to achieve more concurrency and specialized execution. One example is MorphCore [63], which specializes for concurrency by adding an in-order SMT mode to an OOO core. WiDGET [64] is similar in that as it decouples the execution resources from threads, allowing it to specialize the concurrency and coordination of different threads. Another example is specializing for communication by adding reduction instructions to SIMD. While a valuable direction, this style of approach is arguably more difficult as it necessarily retains some of the power overheads of the large core.

General Purpose Computing using Graphics Processing Units (GPGPUs): GPGPUs are interesting as they already apply many specialization principles – concurrency through many independent SIMD/SIMT units, computation through transcendental functions, and data-reuse through programmable scratchpads. That said, they do not specialize for operand communication, instead relying on large register files and shared memory. Some research architectures have even explored such specialization through the addition of spatial fabrics, like SGMF [55].

Making a GPGPU more suitable for programmable acceleration would also mean removing or scaling back features geared towards more general computation, which are simply not needed if only targeting towards commonly accelerated workloads. Some candidates for simplification may be the hardware for coalescing memory, as access patterns are usually known a priori (specializing for communication), or the hardware that handles diverging control flow (specializing for coordination). We suspect that a straightforward application of specialization principles to the GPGPU may lead to a similar design as GenAccel.

Field Programmable Gate Arrays (FPGAs): FPGAs are an interesting starting point as they already apply all specialization principles – DSP slices for computation, configurable networks for communication, block RAMs for data-reuse, configurable logic for coordination, and ample LUT resources for concurrency. They are also programmable, with a spectrum of language choices spanning HDL design, HLS and even OpenCL. Furthermore, they allow further specialization of the coordination using efficient finite state machines rather than a general purpose core.

Given the simple control-flow nature of the accelerator workloads, they easily fit inside an FPGA's resources. Table 4.3 lists the number of control states for the largest kernels from three application domains (belonging to the DSAs explained in Section 3.2) synthesized on an Altera Cyclone-V FPGA using LegUp [65].

Application-Domain Kernel	**Number of States in FSM**
NPU	17
Convolution Engine	14
DianNao	86

Table 4.3: Number of FSM states on Cyclone-V FPGA for Three DSA Applications

However, FPGAs today lack in efficiency (both in frequency and power) because of the mechanisms they rely on – fine-grained programmability, bit-level datapath configurability and global routing. Also, the workloads we specialize require large scratchpad/reuse buffers and the small size of block RAMs (2 to 5 KB each) in FPGAs mean that many must be chained or composed to support the needed access patterns, leading to area and power overheads. Furthermore, when considering the fact that many operations can occur in parallel, effectively a multi-ported register file must be constructed to pass intermediates between the DSP slices. Indeed these challenges can all be addressed by constraining global routing by performing physical layout to achieve the benefits of tiled architectures, and allowing customization of DSP slices to emerging workload needs. These innovations are achieved in recent Altera's 1GHz Stratix 10 [66] FPGAs. Exploring these customized FPGA solutions is an alternative avenue for extending the generality and efficiency of accelerators. Recent FPGA designs are beginning to overcome these burdens by relaxing fully global overheads, and using more efficient routing. System-On-Chip FPGAs (SOC-FPGAs) are also getting faster by using high frequency general purpose processors like ARM Cortex A53 on chip [67]. In the aspect of programming FPGAs, OpenCL and OpenCV based solutions are gaining lot of traction in both industry and academic research [68, 69]. Recent software releases of Intel and Altera's SDK [70, 71] are proofs that FPGAs are evolving to become programmable architectures for masses. If we allow the tailoring of DSP slices to the workloads in question (eg. adding a sigmoid unit), FPGAs with improved frequency and wider SRAMS may eventually be viable architectures for programmable acceleration.

In summary, the stronger argument against using any particular off-the-shelf FPGA as our programmable acceleration design point, or in fact any other existing off-the-shelf product is to do with a lack of *parameterizability*. If we do not allow the parameterization of the number and type of functional units (DSP slices), scratchpad (block RAMs) line sizes, or total design size (and other factors) then these will become the primary reasons for the differences between a programmable architecture and a DSA. This would not allow us to achieve one of the dissertation's goal, which is to discover how low can the cost of programmability be? What we need is an architectural specialization "fabric" like GenAccel which can be customized to a set of chosen workload domains.

4.6 Chapter Summary

This chapter described the general set of mechanisms needed to exploit the specialization principles for programmable acceleration in a domain-agnostic way. It proposed an abstract design for a generic programmable accelerator called *GenAccel* capturing all the mechanisms discussed. It then briefly introduced how GenAccel could be used with design-synthesis, programming and runtime phases. We discussed our strategy in modeling GenAccel design with building blocks of implementation along with the performance, area and power modeling. We explained the provisioning technique for GenAccel model by discussing four domain specific design-points to execute four different application domains and also a more generic and balanced design-point to execute all four application domains in the same architecture. We use these design-points in for our model evaluation in the upcoming chapter. We finally ended the chapter by discussing the possible alternatives to arrive at a design like GenAccel.

5 | Evaluation of GenAccel Model

In this chapter we evaluate our generic programmable accelerator (GenAccel) model's efficiency (performance, power and area) by comparing the designs provisioned for four application domains with corresponding DSAs of those domains. We also evaluate the balanced GenAccel design's area and power efficiency in Section 5.2.3, when provisioned to match the performance/throughput to execute all four application domains. In Section 5.3, we discuss the impact of power and area overheads of a programmable accelerator like GenAccel from an overall system's perspective and critique the usage of GenAccel vs. DSA in a SoC environment. Finally, we conclude the chapter by discussing benefits of programmable acceleration along with some limitations of this model.

We first explain the evaluation methodology we use, which includes workloads evaluated, DSA models and baseline used for evaluation, power and area numbers of the DSAs in the following section and then delve into the detailed evaluation.

5.1 Methodology

At a high level, our methodology attempts to fairly assess GenAccel model's trade-offs across workloads from four different accelerators by obtaining data from the literature, past works and the original authors. We perform GenAccel's evaluation by applying performance modeling techniques, using sanity checks against real systems and using standard area/power models as discussed in Section 4.3. Wherever assumptions are necessary, we made those that favored the benefits of the DSA. The remainder of this section provides details.

5.1.1 Workloads

For fair comparisons, we use the same algorithms where possible, and use the workloads from the original works. For NPU we use the neural network topologies from the original work [42]. Our

DSA	Workload Characteristics
NPU	Layer Topology – Number of input and output neurons
Convolution Engine	Image block and Stencil Size; map-reduce functions
Q100	Query plans; database column format and sizes
DianNao	Feature map and kernel sizes; tiling parameters

Table 5.1: Workload Inputs to Modeling Framework

results include the approximate regions *only* [1]. For Convolution Engine, we used the four basic convolution kernels [43]. As these kernels do not use the DSA's graph fusion unit, we do not include it in its area calculation. We use all DianNao topologies [15] for comparison. Table 5.1 gives an overview of workload characteristics used for comparison of GenAccel with four DSAs.

For Q100 we use 11 TPCH-queries [72] with the same TPCH scale factor (0.01) as the original Q100 paper [44]. We use the same query plan for each, though the GenAccel version sometimes has more phases as its FUs differ (see Section 4.4.1).

5.1.2 Evaluation Strategy

To compare the synthesized domain-provisioned and balanced GenAccel model, we consider the domain-specific models for each application domain. Below we discuss the comparison points and the DSA models built for each each application domain. We also briefly discuss the how we validate our model with regards to DSA specific data points and the comparison strategy.

DSA and Baseline Characteristics

As explained earlier we use four DSAs for comparison against GenAccel and hence need DSA's estimates. Each DSAs' performance, area and power were obtained as shown in Table 5.2 below.

[1]If we used GenAccel on the non-approximate regions as well, GenAccel's performance would surpass NPU on several kernels.

DSA	Execution Time	Power	Area
NPU	Authors Provided	MCPAT-based estimation	
Convolution Engine	Authors Provided	MCPAT-based estimation	
Q100	Optimistic Model	In Original Paper	
DianNao	Optimistic Model	In Original Paper	

Table 5.2: Sources for Domain-Specific Accelerator Characteristics

For NPU and Convolution Engine execution times, we directly use the baseline numbers and DSA performance numbers provided by the authors. For the performance of the Q100 and DianNao DSAs, we constructed a model consistent with the GenAccel framework, which is generally optimistic for the DSA. For Q100, we built a model which takes query-plans as inputs in our own python-based domain-specific language. We validated this DSA model against execution times provided by the authors in their paper, and our model is always optimistic, at most by a factor of $2\times$. We use our model for Q100 because it allowed us to make the same assumptions about the query plans. For DianNao, we used an optimistic performance model, based on its maximum computation throughput and memory bandwidth.

For NPU power and area, we used CACTI for each internal storage structures, since this was similar to the strategy used in the original work itself. Though we did receive the power for Convolution Engine from the authors, we used our own estimates, as these were more consistent with our results [2]. For Q100 and DianNao we used the power and area estimates from the paper. All area and power estimates are scaled and normalized to 28nm.

Model Validation

Table 5.3 shows the techniques used for validating the core component of the accelerator models inorder to evaluate DSAs. The approach we took for validating these modeling techniques was

[2]Using convolution engine power, area estimate would have been favorable to GenAccel.

	GenAccel Modeling Strategy	Model Inputs	Baseline Validation
Common	Analytical cycle-count models for impact of each specialization principle	#Cores, #FUs/types, memory bandwidth, SRAM size and width, CGRA size with SIMD widths	Against modified gem5 [37] or instruction counts from PIN
NPU	Model for neural-network computation based on topology	Layer Topology (Number of Input and Output Neurons)	Developed code generation for neural-network approximation; Used gem5
Convolution Engine	Model for map and reduce stages of convolution kernels	Image block size, map/reduce functions, stencil sizes	Implemented convolution kernels; Used gem5
Q100	Streaming database query model, queries partitioned into compound operations	Query plans written in domain-specific language, Database-FU latency	Implemented single-core query; PIN instruction counts
DianNao	Convolutional/Pooling/Classifier neural networks topology model	Input/output feature maps and its size, kernel size, and tiling parameters	Used published DianNao kernels [15]; Used gem5

Table 5.3: GenAccel Modeling Techniques, Model Inputs and Baseline Validation

to cross-check the model's cycle count prediction against a simulated version where possible, or if the execution time was too large (as for Q100), instead use a PIN model for instruction count, and approximate the execution time. In all the cases, the developed model is within 30% of the predicted execution cycles.

Integration of GenAccel and Number of Units

GA$_N$ and GA$_C$ only required a single unit of GenAccel for achieving the DSA's performance, meaning that these architectures could be directly integrated with a host core. Therefore we do not include the LX3 core's area in the estimation for these design points. Whereas, GA$_D$ and GA$_Q$ needed 8 and 4 GenAccel units respectively, and hence includes the LX3 core area, power.

Comparison to Baseline General Purpose Processor

Much of our results use an Out-Of-Order (OOO) core as a baseline and intuitive reference point. We use the Intel 3770K (IvyBridge) processor, and estimate the power and area based on datasheets and die-photos [73]. All the performance, power and area results are normalized based on this baseline. We scale its frequency to 2GHz, as this is similar to the baseline used in the other DSA works [42, 15]. An exception to this is the Q100 results, where we use their estimated performance of MonetDB (on a 2.2GHz Intel Xeon E5620), as this proved more consistent than the version of MonetDB we had available.

5.2 Evaluation

We now explain the detailed evaluation of our GenAccel model and is mainly organized around three main questions:

Q1. Can GenAccel match the performance of DSAs, and what are the sources of its performance?

Q2. What is the cost of general programmability in terms of area and power overheads?

Q3. If multiple workloads are required on-chip, can GenAccel ever surpass the area or power efficiency?

We answer all the three questions in following subsections. and our primary result is that GenAccel is a viable and programmable accelerator alternative for DSAs, matching their performance with only modest (max $2\times$ to $4\times$) power and area overheads.

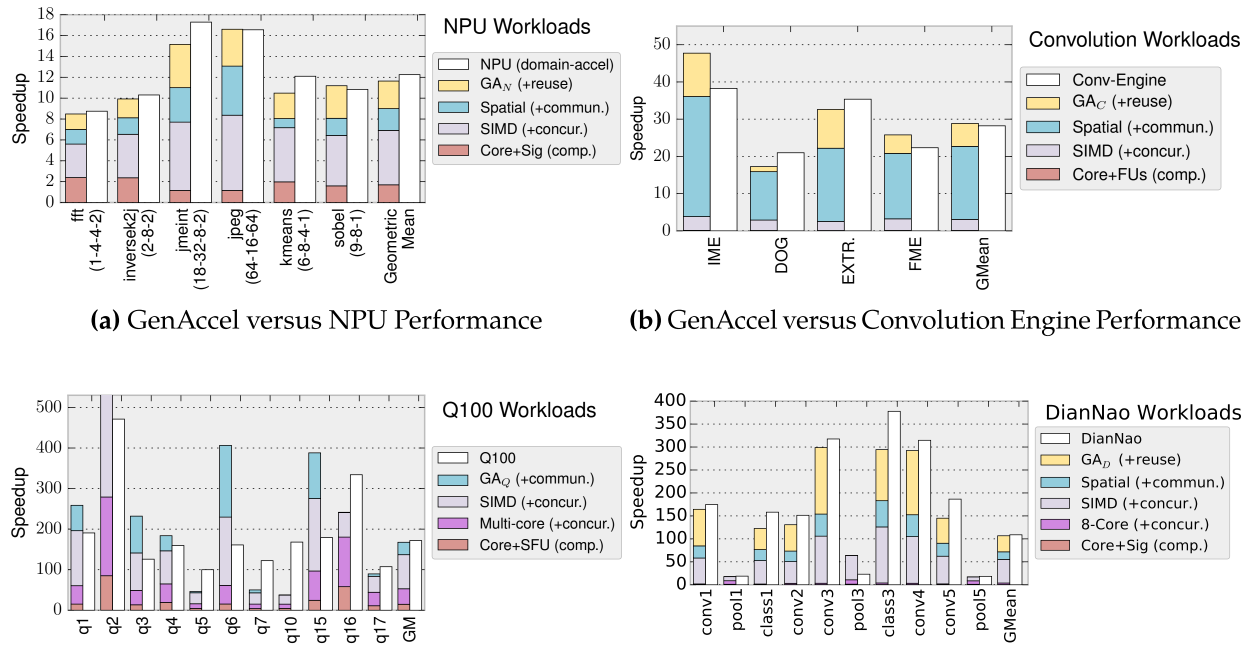
5.2.1 GenAccel Performance Analysis (Q1)

In this section, we compare the performance of the DSAs and domain-provisioned GenAccel designs normalized to our baseline, 4-wide OOO core. To elucidate the sources of benefits of each specialization principle in GenAccel, we also include five additional sub-design points (four in case of single GenAccel unit based designs), where each builds on the capabilities of the previous. Note that these intermediate design points are not area or power-normalized, their purpose is to demonstrate the sources of performance. Figure 5.1 shows the performance comparison of GenAccel with four DSAs; the colored bars with legend also indicate five other sub-design points which is explained in detail below:

1. **Core + SFU** – The LX3 core with added problem-specific functional units (computation specialization).
2. **Multicore** – LX3 multi-core system (+concurrency) (Applicable for multiple GenAccel units designs – Q100 and DianNao).
3. **SIMD** – An LX3 core with SIMD capability, its width corresponding to GenAccel's memory or SRAM interface (+concurrency).
4. **Spatial** – An LX3 core where, the spatial architecture (with SIMD capability) replaces the SIMD units (+communication).
5. **GenAccel** – Previous design points with scratchpad (+reuse)

Across workload domains, GenAccel matches the performance of the DSAs, seeing performance improvements over a modern OOO core between $10\times$ and $150\times$. We argue this is the most insightful, as specialized FUs are generally required to achieve even reasonable performance, and it therefore makes apparent the contributions of the other specialization principles. These DSA works have already shown large speedups over OOO cores, so we do not attempt to reproduce that baseline result here.

NPU versus GA$_N$: Figure 5.1(a) shows GenAccel versus NPU performance, with the neural network topology and size of network listed in parenthesis of each workload. NPU's and GenAccel's organizations differ greatly – independent processing elements in NPU versus the core + CGRA hardware organization of GenAccel. However, their performance is nearly the same, as replicating the neuron values across processing entities is essentially *as good* as storing these together and reading through vectorized loads. In terms of speedup sources, concurrency (SIMD) provides the most benefit – around $4\times$ speedup. Communication specialization (Spatial) and reuse specialization (GenAccel) together provide only $1.7\times$ speedup.



(a) GenAccel versus NPU Performance

(b) GenAccel versus Convolution Engine Performance

(c) GenAccel versus Q100 Performance

(d) GenAccel versus DianNao Performance

Figure 5.1: GenAccel versus DSA Performance Across Four Domains

Convolution Engine versus GA$_C$: Figure 5.1(b) shows GenAccel versus Convolution Engine performance for four important image processing and motion estimation kernels. Convolution engine is essentially a wide vector mapping and reduction accelerator engine that allows fast accesses to shifted image rows/columns and coefficients. Though GA$_C$ performs similarly to the DSA, it has 0.84× clock-to-clock performance. This is because the DSA is running at a slower 800MHz frequency. For the DOG kernel, GenAccel loses performance as the scratchpad size could not fit all of the image sizes needed for the computation. Overall, the major reason for performance difference is also because of a lack of a custom shift-register interface. Currently, the CGRA's vector interface needs to reorganize the data to produce shifted versions of inputs. The major performance contribution among specialization principles for GA$_C$ design is from concurrency – around 31× from SIMD. Note that the LX3 Core + FUs bar is below 1, as it is slower than the baseline. CGRA's spatial communication and reuse play a lesser role, with benefits around 7.3× and 1.3× respectively.

Q100 versus GA$_Q$: Figure 5.1(c) shows GenAccel versus Q100 performance for 11 database-streaming kernels using MonetDB. Though the 4-unit GA$_Q$ performs similarly with Q100 overall (only 2% difference), performance varies across individual queries. Specifically, queries 5, 7 and 10 are sort-heavy (and 16 and 17 to a lesser extent), and Q100 benefits from the specialized Sort and Partition FUs. We also emphasize here that the benefits of both GenAccel and Q100 over MonetDB reduce by around a factor of 10 on larger databases (e.g. TPCH scale factor = 1), but we report this scale factor of 0.1, as it is what Q100 was optimized for. The major source of performance is concurrency – around 3.60$\times$ from multi-core or units, 2.58$\times$ from SIMD. Some workloads benefit significantly from the CGRA, which is helpful when the bandwidth is not saturated and the workload is not dominated by sorts or joins. Note that to be consistent with the Q100 work, the baseline here is an OOO core running MonetDB, while Q100 and GenAccel model manually-optimized queries. This explains why even the LX3 core with special FUs can outperform the baseline. As an example, we ran an optimized q1 (matching the GenAccel/Q100 algorithm) on the OOO core, and it was 35x faster than the MonetDB version.

DianNao versus GA$_D$: Figure 5.1(d) shows GenAccel versus DianNao DSA performance for the classifier, convolution and pooling kernels of a typical deep-neural network application. Performance is similar on most workloads. GA$_D$ has minor instruction overhead in managing the DMA engine to stream the neurons and weights. But it gains back some ground on one of the pooling workloads, because the decoupled design of GenAccel allows it to load neurons at a higher bandwidth and hence is not memory saturated. Again, concurrency was most important for performance – around 8$\times$ from multi-core and 14.4$\times$ from SIMD. The CGRA provides an additional small benefit by reducing instruction management. Also, adding a reuse buffer reduces cache contention. The combined speedup from CGRA and reuse buffers is 1.9$\times$.

Takeaway: *GenAccel designs have competitive performance with DSAs. The performance benefits come mostly from* concurrency *than any other specialization technique.*

5.2.2 GenAccel Area and Power Overheads (Q2)

As we have shown so far, GenAccel is generally able to compete with the more specialized designs in terms of performance – this makes sense as each of those domain-provisioned GenAccel models were designed for equivalent throughput. The main costs are in the in the area and power overheads which in-turn reflect the programmability overheads, and the breakdown of area and power for each GenAccel design-point is shown in Table 5.4 respectively.

	Area (mm^2)	**GA$_N$**	**GA$_C$**	**GA$_Q$**	**GA$_D$**	**Power (mW)**	**GA$_N$**	**GA$_C$**	**GA$_Q$**	**GA$_D$**
Breakdown	Core + Cache			0.09	0.09	Core + Cache	41	41	41	41
	SRAM	0.04	0.02	0.04	0.04	SRAM	9	5	9	5
	Functional Unit	0.24	0.02	0.09	0.02	Functional Unit	65	7	33	7
	CGRA Network	0.09	0.11	0.22	0.11	CGRA Network	34	56	46	56
	Unit Total	0.37	0.15	0.44	0.26	Unit Total	149	108	130	108
	GA Total Area	0.37	0.15	1.78	2.11	GA Total Power	149	108	519	867
	DSA Total Area	0.30	0.08	3.69	0.56	DSA Total Power	74	30	870	213
	GA/DSA Overhead	1.23	1.74	0.48	3.76	**GA/DSA Overhead**	2.02	3.57	0.60	4.06

Table 5.4: GenAccel (GA) Power and Area Breakdown/Comparison (normalized to 28nm)

To elucidate the costs of programmability, Figure 5.2 shows the power and area efficiency for four performance-equivalent designs, using a single OOO core as the baseline. We also show an intermediate design-point GA$_{simd-only}$, because it is a more standard reference point. It does not employ scratchpads or a CGRA network but does have specialized FUs with SIMD capability. We do not show the single-core or multi-core only points, as scaling up their cores to meet the performance target does not result in practical designs.

Overall, GenAccel has some, but not excessive overheads – up to 3.8$\times$ area and 4.1$\times$ power compared to the DSAs. Even so, the GenAccel designs are between 6$\times$ to 90$\times$ area-efficient (less-area) than a single core, and between 5$\times$ and 40$\times$ power-efficient (less power consumed). Also, the GenAccel designs are generally better than the performance-normalized SIMD design point, implying that the spatial fabric and reuse buffers are effective for reducing the overheads, compare

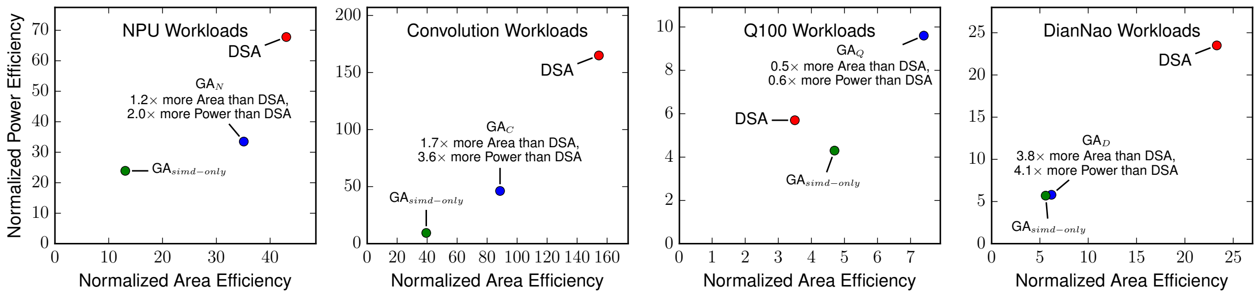


Figure 5.2: Area and Power Trade-offs Using Performance Equivalent GenAccel Designs (Baseline: Core + L1 + L2 from I3770K processor; higher and to-the-right is better

to SIMD vector lanes.

GA$_D$ has the worst case area and power overheads of 3.8× and 4.1× respectively compared to DianNao. The CGRA network dominates the area and power, because it supports relatively tiny 16-bit FUs and routing of 16-bit values adds lot of power overhead. This is a reasonable trade-off given DianNao uses a mostly fixed datapath, while GenAccel supports a highly configurable mesh.

GA$_C$ also has overheads of 1.7× area and 3.6× power. Besides the CGRA network overhead, Convolution Engine optimizes for a non-standard datapath width (10-bit versus 16-bit in GenAccel), and runs at a lower frequency of 800MHz.

For the NPU workloads, the GA$_N$ is similar area and has a 2× power overhead. The reason for these relatively low overheads in this case is the high contribution of floating point and sigmoid FUs, amortizing the overhead of the LX3 core and CGRA network in GenAccel. Overall for area, this is sensible because the specialization of computation (sigmoid FU, implemented with a large lookup table) is the largest contributor to area, and is identical across designs. For power, the most significant source of overhead is the core itself, but NPU also has power overhead in its per processing entity controllers and bus scheduler. Compared to a simpler SIMD-based design, GenAccel is nearly 2× more power and energy efficient, due to the large benefits of the scratchpad storing the read-only neural weights.

Surprisingly, GA_Q has $0.5\times$ the area and $0.6\times$ the power of Q100. One reason for this is that GenAccel does not embed the expensive Sort and Partition units, which did lead to performance loss on several queries, but was arguably a reasonable trade-off overall. GenAccel also uses a simple circuit-switched CGRA, whereas Q100 uses highly-buffered routers based on the Intel Teraflops chip [74].

Overall, the programmable core and the CGRA were the largest contributors for power and area. The scratchpads for for reuse specialization played a small role in the overhead – this is natural because similar sized SRAMs can be used in both the DSA and GenAccel designs. CGRA network costs overheads, but including it enables the specialization of communication, which is important for performance. More importantly, it enables run-time reconfigurability, and therefore generality. Even with significant overheads versus the less general DSA designs, the GenAccel design points still achieved high area and power efficiency advantages over a single OOO core.

Takeaway: *With suitable engineering, the overheads of programmability can be reduced to small factors of $2\times$ to $4\times$, as opposed to the $100\times$ to $1000\times$ inefficiency of large OOO cores.*

Evaluation of Area-Power Normalized GenAccel Design's Performance with DSAs

Up to this point, we have considered GenAccel designs targeted to each DSA's performance, then measured the power and area overheads. We now consider the opposite – restricting the design to target either the DSA power or area (within 15%), and observe the other metrics. Table 5.5 shows the results.

When area is constrained, the GA_C requires the most cuts to resources, at about 1/2 of the CGRA size. The area constrained GenAccel designs are $1.2\times$ slower than the DSAs in average and have about almost equivalent power. When power is constrained, DianNao is the most affected, requiring us to cut 6/8 GenAccel units. In this scenario, the geometric mean performance is $1.37\times$ slower and the area is $0.76\times$ that of the DSAs.

Area < 1.15× DSA	**NPU**	**Convolution Engine**	**Q100**	**DianNao**
Execution Time Ratio	1.19	2.27	0.53	2.78
Area Ratio	1.1	1.1	0.23	1.14
Power Ratio	1.1	0.75	0.55	1.74
GenAccel Changes	3/4 size	1/2 size	None	3/8 units

Power < 1.15× DSA	**NPU**	**Convolution Engine**	**Q100**	**DianNao**
Execution Time Ratio	1.19	1.33	0.53	4.17
Area Ratio	1.1	1.72	0.23	0.76
Power Ratio	1.1	1.09	0.55	1.15
GenAccel Changes	3/4 size	7/8 size	None	2/8 units

Table 5.5: Area limited (top) and power limited (bottom) GenAccel Characteristics (lower is better)

5.2.3 Supporting Multiple Application Domains (Q3)

Until now, we evaluated only the domain provisioned GenAccel designs and estimated the cost of added programmability. We now in this subsection mainly try to answer the third question asked in Section 5.2 – If multiple workloads are required on a single-chip, can GenAccel ever surpass the area or power efficiency compared to having multiple-DSAs? In order to answer this question, we evaluate the most general *Balanced* GenAccel design described in Section 4.4.1 below:

If multiple workload domains require programmable acceleration on the same chip or SOC, but do not need to be run simultaneously, it is possible that GenAccel can be more area efficient than a Multi-DSA design. Figure 5.3 shows the geometric-mean area and power trade-offs for two different workload domain sets, comparing the Multi-DSA chip to the balanced GA_B design.

The domain set [NPU/Convultion-Engine/DianNao] excludes our best result (Q100 workloads). In this case, GA_B still has 2.7× area and 2.4× power overhead. However, with Q100 added (All four application domains), GA_B is only 0.6× the area of all DSAs combined. And we believe, with more domains targeted for GenAccel, the balanced GenAccel design will be more area and power efficient than all DSAs area and power combined.

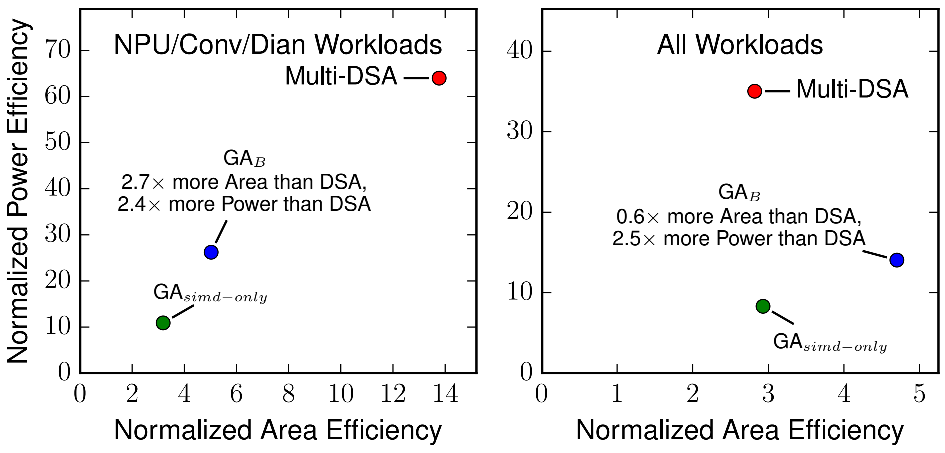


Figure 5.3: Area and Power Comparison of Multi-DSA vs Balanced GenAccel Design (GA$_B$)
(Baseline: Core + L1 + L2 from I3770K processor; higher and to-the-right is better)

Takeaway: *If only one domain needs to be supported at a time in Multi-DSA chip, GenAccel can become more area efficient than using multiple DSAs.*

5.3 System Level Trade-offs with Programmable Accelerator and DSA

With a detailed evaluation of the GenAccel model, we have explored the relative power, area, and performance trade-offs of using a programmable accelerator over a DSA, but it is important to understand how this affects the overall decision of which architecture to employ in a SOC. Specifically, while the speedups of DSAs and a Programmable Accelerator (GenAccel Model) can be similar, an important question to consider is – by how much does the power and area overheads affect the energy benefits and economic costs, when accelerating a general purpose chip. We use simple analytical reasoning in this section to explore the trade-offs.

5.3.1 Energy Efficiency Trade-offs

In this subsection, we analytically bound the possible energy efficiency improvement of a general purpose system accelerated with a DSA versus a GenAccel design, by considering a zero-power DSA – meaning the DSA is most power efficient with consuming almost zero power.

We first define the overall relative energy, E, for an accelerated system in terms of:

- S – the accelerator's speedup.
- U – the accelerator utilization as a fraction of the original execution time.
- P_{core} – general purpose core power.
- P_{sys} – overall system power.
- P_{acc} – accelerator power.

The core power includes components which are generally *not* used when the computation is offloaded by invoking the accelerator. he system power includes chip components that are active while accelerating, which could include higher level caches and DRAM, or other SOC components, for example. The total energy then becomes:

$$E = P_{acc}(U/S) + P_{\text{sys}}(1-U + U/S) + P_{\text{core}}(1-U) \tag{5.1}$$

Based on the above equation, the energy efficiency improvement of a DSA versus GenAccel (GA) system ($\text{Eff}_{\text{dsa/ga}}$), given that their speedups are held equivalent, becomes:

$$\text{Eff}_{\text{dsa/ga}} = \frac{P_{\text{ga}}(U/S) + P_{\text{sys}}(1-U + U/S) + P_{\text{core}}(1-U)}{P_{\text{dsa}}(U/S) + P_{\text{sys}}(1-U + U/S) + P_{\text{core}}(1-U)} \tag{5.2}$$

Based on our earlier assumption, by setting the DSA power to zero and rearranging, the equation now becomes:

$$\text{Eff}_{\text{dsa/ga}} < \frac{P_{\text{ga}}(U/S)}{P_{\text{sys}}(1-U + U/S) + P_{\text{core}}(1-U)} + 1 \tag{5.3}$$

The best case for the DSA would be 100% utilization, and in that case we get the intuitive result that maximum energy efficiency improvement is: $\text{Eff}_{\text{dsa/ga}} < P_{\text{ga}}/P_{\text{sys}} + 1$. Since GenAccel's power

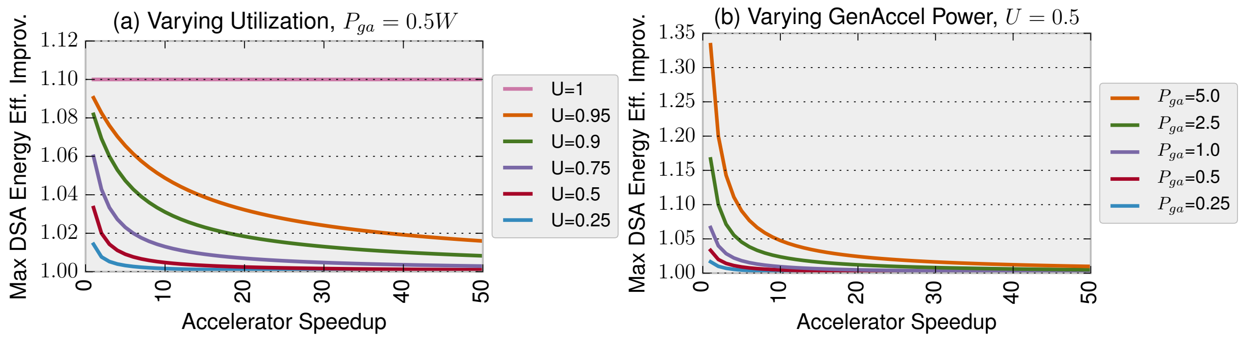


Figure 5.4: Energy Efficiency Benefits of a System with Zero-Power DSA ($P_{\text{core}} = 5W$,$P_{\text{sys}} = 5W$)

is usually very low, perhaps a factor of 10 less than system power, the maximum system energy efficiency savings is less than 10%, even with a perfect DSA.

More interesting scenario to consider is a plausible but still optimistic accelerator utilization like 50%. Under these settings, the maximum efficiency becomes:

$$\text{Eff}_{\text{dsa/ga}}|(U = 0.5) < 0.5 \times \frac{P_{\text{ga}}/S}{P_{\text{sys}}/S + P_{\text{sys}} + P_{\text{core}}} + 1 \tag{5.4}$$

Since we are assuming high-performance accelerators, a reasonable value for S (speedup) is $10\times$, and we could again assume that GenAccel's power is $10\times$ less than the core and system power. In this setting the *maximum* possible energy efficiency gain from a DSA is less than 0.5%.

We characterize these trade-offs across different accelerator utilizations and speedups in Figure 5.4, using 5W as the core and system power. Figure 5.4(a) shows that the maximum benefits from a DSA reduce both as the utilization goes down (stressing core power), and when accelerator speedup increases (stressing both core and system power). For a reasonable utilization of U=0.5 and speedup of S=10, the maximum energy efficiency gain from a DSA is less than 0.5%. Figure 5.4(b) shows a similar graph where GenAccel's power is varied, while utilization is fixed at U=.5. Even considering an GenAccel with power equivalent to the core, when GenAccel has a speedup of $10\times$, there is only

5% potential energy savings remaining for a DSA to optimize.

Takeaway: *Putting the above together, we claim that when an GenAccel can match the performance of a DSA, the further potential energy benefits of a DSA are usually very small. In other words, a 4× power overhead for GenAccel versus a DSA is generally inconsequential.*

5.3.2 Economic Trade-offs

DSAs are a reality today and without a significant reason to invest in the programmable accelerator based SOC, its possible that GenAccel would not ever become economically viable. That said, we argue that there are tangible reasons why an GenAccel-based approach could be economically advantageous.

First, the fixed costs of targeting some domain may indeed be much lower for GenAccel. Once the GenAccel hardware is designed, targeting a domain is a matter of hardware parameterization and relatively straightforward software development. In some cases, GenAccel may be able to target new domains without any hardware changes or parameterization. It is also true that for a given application domain, GenAccel's recurring area costs are factors higher. However, GenAccel is relatively small compared to a modern general purpose multi-core, and when targeting multiple domains, GenAccel's area overheads are amortized.

5.4 Discussion

In this section, we briefly discuss some primary limitations of GenAccel model with regards to the workload generality and also the advantages of it compared to DSAs.

5.4.1 Limitations

Accelerator Workload Generality: For our evaluation, we made several assumptions about the type of workloads targeted which are generally computationally intensive, have lots of parallelism with simple control flow and regular memory access patterns. The implication is that there are workload varieties that GenAccel design will not be suitable for, as well as at least several existing accelerators which GenAccel will not be able to match. One example would be the packet classification, which would require too much irregular memory accesses, but Spitznagel et al. [75] is able to create effective specialized hardware with extended TCAMs. Another example is the unsuitable bit-level optimizations that Fowers et al. uses to design an accelerator for lossless compression on FPGAs [76].

It is possible that, when considering a broader or different set of workload properties, the specialization principles and their mechanisms identified here may not be the most operative. Even so, a GenAccel reference point could help identify what should be these new mechanisms, or perhaps even help to expose additional specialization principles.

5.4.2 Advantages of GenAccel Model

Characterizing Generality: Contrasting to the above mentioned limitations, GenAccel also provides a certain degree of flexibility. For example, considering the NPU approximate computing accelerator, GenAccel can be used for non-approximate code as well. Another example is in the case of the TPCH queries that Q100 supports where, the query plans including non-streaming operators (e.g. hash-join) are not supported by the DSA – but this is natural for GenAccel as it has fall-back mechanism in the form of low-power core. For DianNao, a recent extension added is the architecture support for more machine learning algorithms [16]. A GenAccel design could achieve the same through synthesis-time reconfiguration rather than re-inventing a new architecture. Quantifying and characterizing the achievable generality more rigorously is future work. In some cases (e.g. `kmeans`) non-approximate acceleration is significantly more effective [52, 42].

5.5 Chapter Summary

In this chapter, we studied a detailed evaluation of domain-provisioned GenAccel designs against four different application domains and their DSAs. When provisioned to match the throughput of the target applications, GenAccel design can easily match the performance of DSAs with modest overheads in area and power. We also explore the advantages of a more generic balanced GenAccel design which can map all four application domains onto a single programmable accelerator fabric with better area efficiency and minimal power overheads. An analytical model is developed exploring the system-level energy efficiency trade-offs of having a DSA versus a programmable accelerator like GenAccel. We conclude that when the system power dominates and speedups of applications accelerated are less with lower accelerator utilization, then it is economically and logically not viable to build DSA over a programmable accelerator.

6 | Stream-Dataflow Acceleration

Using the generic programmable accelerator model GenAccel we showed that programmable accelerators can be designed with performance matched to domain-specific hardware with minimum overheads in area and power. In this chapter, we introduce a particular programmable acceleration paradigm called *Stream-Dataflow Acceleration* to accelerate data-streaming applications and realize it with a detailed architecture, execution model, ISA interface and programming abstractions.

We first provide an overview of the stream-dataflow paradigm in the following section. We then motivate the need for a stream-dataflow accelerator in Section 6.2 by contrasting it with existing data-parallel architectures and discuss available opportunities for stream-dataflow in Section 6.3. Then in Section 6.4, we delve into the architecture details by explaining the programming abstractions and the stream-dataflow execution model. Finally, we end the chapter by discussing the new ISA interface proposed for programmable accelerators and illustrate a simple stream-dataflow application written using that interface.

6.1 Introduction to Stream-Dataflow Acceleration

An important observation, as alluded to in the literature [31, 3] and in Section 3.1, is that typically-accelerated workloads have common characteristics:

1. High computational intensity with long phases.
2. Small instruction footprints with simple control flow.
3. Straightforward memory access and re-use patterns.

The reason for this is simple – these properties lend themselves to very efficient hardware implementations through exploitation of concurrency. Furthermore, data processing algorithms either naturally have these properties, or new algorithms are crafted with these properties in mind, to enable high performance on data-parallel hardware architectures. Existing data-parallel hardware solutions perform well on these workloads, but in their attempt to be far more general, sacrifice too much efficiency to supplant domain-specific hardware. This also sheds light on why traditional data parallel architectures have overheads: As an example, short-vector SIMD relies on inefficient general pipelines for control and address generation, but accelerated codes typically do not have complex control and memory access. GPGPUs hide memory latency using hardware for massive multi-threading, but accelerated codes' memory access patterns can usually be trivially decoupled without multi-threading.

To take advantage of this opportunity, we propose an accelerator architecture and execution model for acceleratable workloads, whose hardware implementation can approach the power and area efficiency of specialized designs, while remaining flexible across application domains. Because of its fundamental specialization components, it is called as *Stream-Dataflow* accelerator. The architecture explores – how to express the common algorithmic properties of regular streaming applications in a general way by instantiating the hardware primitives exploiting the common specialization principles. In the stream-dataflow interface, the *dataflow* component enables dataflow computations with high concurrency, and the *stream* component enables communication and coordination of data-streams at very low power and area overhead. This part of the dissertation explores the hardware and software implications of the stream-dataflow interface and describes a detailed architecture and micro-architecture of the specialization mechanisms and hardware primitives discussed before in Chapter 4.

Stream-dataflow acceleration exposes the high-performance hardware substrate with the following three basic abstractions:

- A dataflow graph (DFG) abstracting the repeated, pipelined execution of computations from typical accelerator applications.
- Stream-based commands for facilitating efficient data-movement and communication across

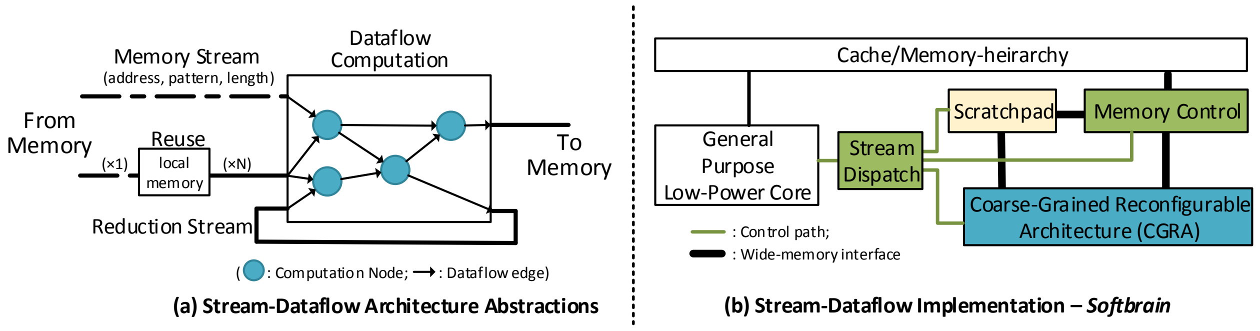

Memory Stream
(address, pattern, length)
Dataflow Computation
From Memory
Reuse
local memory
(×1)
(×N)
To Memory
Reduction Stream
(●: Computation Node; ⟶: Dataflow edge)
(a) Stream-Dataflow Architecture Abstractions
Cache/Memory-heirarchy
General Purpose Low-Power Core
Stream Dispatch
Scratchpad
Memory Control
Coarse-Grained Reconfigurable Architecture (CGRA)
: Control path;
: Wide-memory interface
(b) Stream-Dataflow Implementation – *Softbrain*

Figure 6.1: Stream-Dataflow Abstractions and Implementation

the hardware components and memory; The source and destination architectural locations include the memory address space, DFG vector ports, and the scratchpad address space used for efficient data-reuse.

- Barrier commands to facilitate coordination between different parallel phases of the algorithm.

Figure 6.1(a) depicts the programmer view of stream-dataflow, consisting of the dataflow graph (DFG) itself, and explicit stream communication support for memory access, data read reuse and recurrence. The intuition behind this architecture is that the dataflow based execution will be similar to the datapath of an ASIC, and the specialized stream based communication will have low overheads, similar to that of an ASIC. The abstractions lead to an intuitive hardware implementation; Figure 6.1(b) shows our high-level design of the stream-dataflow accelerator implementation called *Softbrain*.

It consists of a coarse-grained reconfigurable architecture, a locally addressed scratchpad and a memory interface control connected with wide buses to memory. All the components are controlled from the simple controller core, which sends stream commands to be executed concurrently by the memory interface controller, CGRA and scratchpad controller. Infact, all these mechanisms are inspired from the findings and discussions in Section 4.1. The architecture for stream-dataflow is derived from straight-forward application of those mechanisms along with the software programming interface enabled hardware primitives. This coarse-grained nature of the stream-based interface enables the core to be incredibly simple, while still enabling highly concurrent execution in CGRA. Also, to keep the datapath modifications of the low-power core as minimal as possible and have non-intrusive integration, we have a special dispatcher called stream-dispatcher which is responsible for taking the commands from low-power core essentially decoupling the core off the critical path to continue execution while co-ordinating the other components. The stream access patterns and restricted memory semantics enable efficient address generation and co-ordination hardware compared to traditional architectures.

Relative to a domain specific architecture, a stream-dataflow processor can reconfigure its datapath and memory streams, so it is far more general and adaptable. Moreover, because it explicitly separates the implementation of data movement from computation, the architecture becomes trivially parameterizable for various sets of domains. Relative to existing solutions like GPGPUs or short-vector SIMD, the power and area overheads are significantly less on amenable workloads. An implementation can also be deployed flexibly in a variety of settings, either as a standalone chip or as a block on a SoC. It could be integrated with virtual memory, and either use caches or directly access memory. By restricting the algorithms that can run on the architecture, the hardware can perform each primitive task (memory access, computation etc,) at a much coarser grain – effectively eliminating the per-instruction overheads that plague traditional architectures.

Over the next few sections, we first define stream-dataflow, describe its execution model, and explain why it provides specialization benefits over existing architectures. We then discuss the ISA and programmability before describing an example program for stream-dataflow. To demonstrate the generality of this architecture and this implementation's capabilities, we compare the stream-dataflow implementation against a state-of-the-art machine learning accelerator, as well as fixed function accelerators for MachSuite workloads. Our evaluation shows it can achieve equivalent performance to the accelerators, with orders-of-magnitude area and power efficiency improvement over CPUs. Compared to the machine learning accelerator, we average only $2\times$ power and area overhead. On the broader set of MachSuite workloads, compared to custom ASICs, the average overhead was $2\times$ power and $8\times$ area. We discuss our hardware implementation – Softbrain and the evaluation in detail in Chapter 7 and 8.

Our specific contributions for stream-dataflow acceleration part of the dissertation are:

- We describe the inefficiencies common to many existing data-parallel hardware architectures, and opportunities to reduce them through specialization (Section 6.2 and 6.3).
- We define a stream-dataflow architecture for accelerating workloads (Section 6.4), with a general set of abstractions that can efficiently express acceleratable workloads. We believe these abstractions can serve as a natural program representation for compiling to different programmable architectures can be directly executed on an architecture which exposes the hardware primitives of streams and independent computation blocks. The implementation of such an architecture is competitive with custom solutions, and is suitable for trivial domain-adaptation.
- We present a detailed, low-level micro-architecture details of the stream-dataflow architecture and elucidate its generality, performance, area and power trade-offs (Chapter 7).
- We further implement a chisel-based prototype of stream-dataflow architecture called *Softbrain* combined with an open-source RISCV core, with simple programming interface, and we evaluate its performance, power and area characteristics respective to state-of-the-art general purpose processors and accelerators (Chapter 8).

6.2 Motivation

For a broad class of data-processing algorithms, the mere existence of domain-specialized hardware, and the typical orders of magnitude performance and/or energy benefits they provide over existing general purpose and data-parallel hardware techniques indicates that the overheads in current existing architectures are significant. By definition, the strategy that domain-specific accelerators employ is to limit the programming interface to support a much narrower set of functionality suitable for the domain, and in doing so simplify the hardware design and improve efficiency. We further hypothesize that the efficiency gap between domain-specific and general purpose architectures is fundamental to the way general purpose programs are expressed at an instruction-level, rather than a facet of the micro-architectural mechanisms employed.

So far, existing programmable architectures like SIMD, SIMT, Spatial-Dataflow have shown some promise, but have only had limited success in providing a hardware-software interface that enables the same specialized micro-architecture techniques that more customized designs have employed. Therefore, our motivation is to discover what are the architectural abstractions that would enable micro-architectures with the execution style and efficiency of a customized design, at least for a broader and important class of applications that have long phases of data-processing and streaming memory behavior. To get insights into the limitations of current architectures and opportunities, this section examines the inefficiencies in existing programmable hardware paradigms. We then discuss how their limitations can inspire a new set of architecture abstractions for stream-dataflow paradigm in the following section. Overall, we believe that the stream-dataflow abstractions we propose could serve as the basis for future programmable accelerator innovation.

6.2.1 Inefficiencies in Existing Approaches

An application or domain-specific accelerator achieves performance by feeding a highly concurrent datapath with streaming data from memory and efficiently reusing data through custom scratchpad memories. It does this using control logic specific to the application, avoiding the power and area overheads of general purpose architectures. Several fundamentally different architecture paradigms have been explored in an attempt to enable programmable hardware acceleration and the prominent classes are shown in Figure 6.2. [1].

Generally speaking, relative to DSA or ASIC, these architectures have three broad sources of overheads as listed below:

1. Address generation and communicating with the caches or memory subsystem.

[1]We discuss here the core micro-architecture. Integration into multi-processor systems is mostly orthogonal to this discussion.

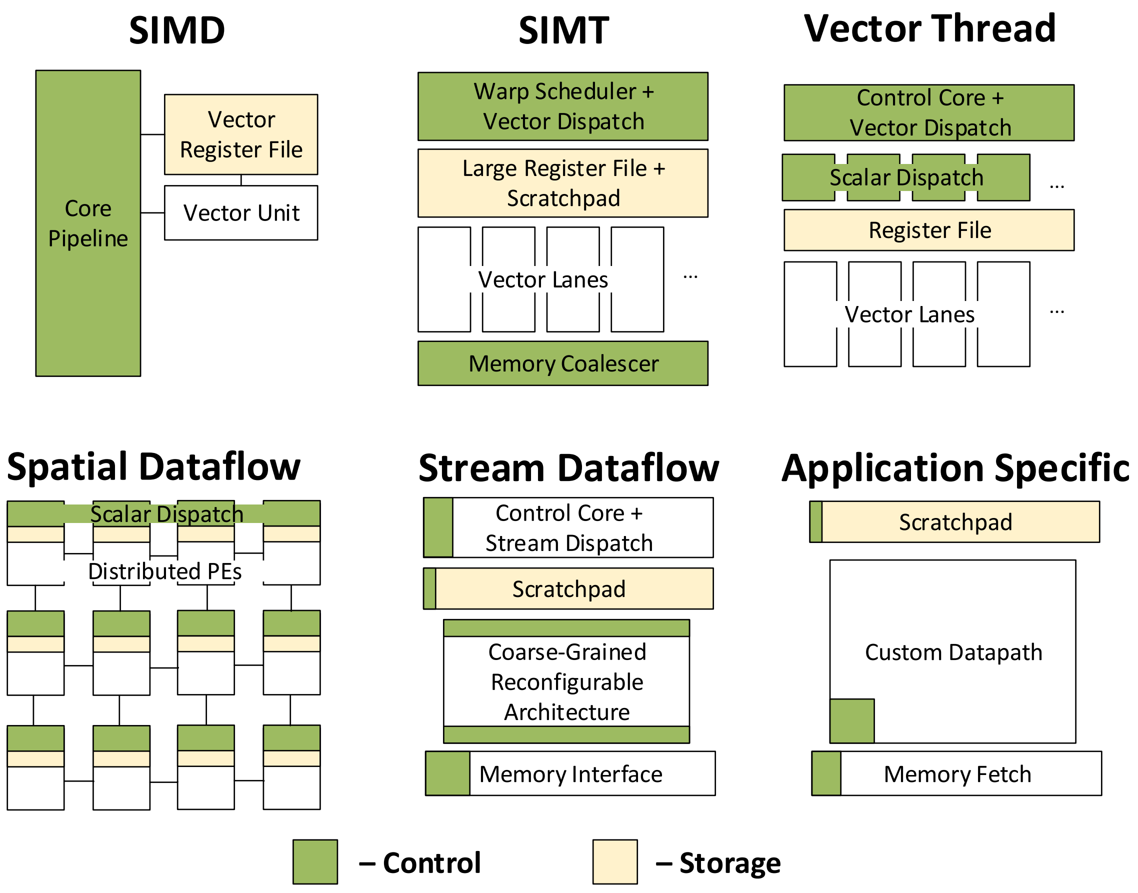


Figure 6.2: High-Level Organization of Data-Parallel Architecture

2. Mechanisms for attaining high utilization of execution resources through concurrency.

3. Non-decoupling of access-execute phases in the program. They also have overheads in how they handle irregular or otherwise unsuitable code.

Table 6.1 summarizes these overheads for the above three categories, and we discuss them in detail below.

SIMD and SIMT

Both SIMD and SIMT hardware provide fixed-length vector abstractions in their ISA, which enables micro-architectures that amortize instruction dispatch and enable fewer, wider memory accesses.

Feature	**SIMD (Short-Vector)**	**SIMT (GPGPU)**	**Vector Threads**	**Spatial Dataflow**	**Stream-Dataflow**
Addressing and Communication	• Unaligned address • Complex scatter-gather instructions • Mask and merge instructions	• Redundant address generation for unit stride • Address coalescing across threads	• Redundant address generation units (area only)	• Redundant address generation units for unit stride • Inefficient bandwidth use for local access	*Opportunity: Efficient hardware for stream patterns*
Utilization and Latency Hiding	• General core issue width • General core instructions reordering	• Thread scheduling • Very large register files • Cache-pressure from many threads	• Redundant dispatch units (area only) • Control core issue width and reordering	• Redundant dispatch units	*Opportunity: Distribute control without replication*
Irregular Execution	• Inefficient general pipeline	• Hardware supporting branch divergence	-	-	*Opportunity: Provide simplest possible hardware for uncommon case*

Table 6.1: Overheads in Existing Architectures and Opportunities

The specification of computation at the instruction-level, however, means that neither can avoid instruction communication through large register files. In addition, both architectures do not provide inexpensive support of high hardware utilization. Because the vector length is fixed and relatively short, short-vector SIMD processors constantly rely on the general purpose core for dynamically scheduling the parallel instructions. Scaling issue width, reordering logic and register file ports is expensive both in area and power. Also, fixed-length vector execution causes unaligned accesses thus increasing the bandwidth pressure. For complex access patterns, SIMD requires either masking and merging instructions, or complex scatter-gather hardware. SIMD extensions also lack programmable scratchpads for efficient data reuse.

SIMT hardware like GPGPUs expose massive multi-threading capability to enable high hardware utilization, by allowing concurrent execution of many warps, which are groups of threads that issue together. This requires large multi-ported register files to hold live state, complex warp-scheduling hardware, and incurs cache pressure from many independent warps. SIMT improves handling of irregular accesses, but then require redundant address generation units across many threads for spatially local accesses and also need additional expensive logic for address coalescing hardware. They also improve SIMD by using a simple frontend pipeline to dispatch each instruction to multiple vector execution lanes.

Vector Threads ([77, 78])

A vector-thread data-parallel architecture is similar to SIMT, but exposes the programming ability to specify both SIMD-style and scalar execution of the computation lanes. They make use of their vector and scalar dispatch units for flexibly executing such non data-parallel codes. While this does eliminate the control divergence penalty, it faces many of the same limitations as SIMD. Specifically, it cannot avoid register file access due to the instruction-level specification of computation, and the limited vector-length means it must rely on control core's pipeline to achieve high utilization. It also introduces redundant hardware for address generation and in dispatching instructions.

Spatial Dataflow ([79, 80, 81, 82, 83])

Spatial dataflow architectures further distribute the control and expose the routing and communication channels of an underlying computation fabric through their hardware-software interface. This enables a distributed instruction dispatch, and eliminates the need for register file accesses between live instructions. The dispatch overheads can be somewhat amortized using a configuration step. The distributed nature of this abstraction also enables high utilization without the need for multi-threading or multi-issue logic.

However, these architectures are unable to specialize for the memory accesses to the same degree. Micro-architectural implementations typically perform redundant address generation and issue more and smaller cache accesses for spatially local accesses. This is because the spatially distributed memory address generation and accesses are more difficult to coalesce into vector-memory operations and can incur inefficient bandwidth utilization.

Application Specific Hardware

Application-specific or Domain-specific accelerators achieve performance by providing highly concurrent pipelined execution, and high energy efficiency by minimizing the overheads of communication and control. At a high level, these architectures generally consist of a pipelined datapath supporting very high utilization, ad-hoc address generation, custom buses to access a memory space, and custom memories like scratchpads to enable efficient data-reuse. To get these specialization benefits with minimum overheads programmable accelerators must keep the same essential execution model, but add programmable abstractions.

Summary and Observations

First, being able to specify vectorized memory accesses is extremely important, not just for parallelism and reducing memory accesses, but also for reducing address generation overheads. On the other hand, though vectorized instructions do reduce instruction dispatch overhead, the separation of the work into fixed-length instructions requires inefficient operand communication through register files and requires high-power mechanisms to attain higher utilization. Exposing a spatial dataflow substrate to software solves the above problems, but complicates and disrupts the ability to specify and take advantage of vectorized memory access.

6.3 Overview

Based on our study and analysis of inefficiencies in different existing architectures explained in the previous section, we take away three important attributes for stream-dataflow model to explore:

- Eliminate redundant address generation logic and inefficient communication, through efficient hardware support for common data stream patterns.
- Distribute the control logic in the architecture and thus decentralizing the control of concurrency without duplicating control logic for the same task.
- Provide the simplest possible hardware for the uncommon case of executing irregular or non-concurrent code regions.

At the heart of the previous observations of inefficiencies also lies the fundamental trade-off between vector and spatial architectures – *vector architectures expose a far more efficient parallel memory interface, while spatial architectures expose a far more efficient parallel computation interface*. For it to be conceivable that a programmable accelerator architecture can be competitive with an application-specific or domain-specific hardware, it must expose both efficient interfaces.

While achieving the benefits of spatial and vector architectures in the general case is perhaps impossible, we argue that it *is* possible in a restricted but important workload setting. In particular, many data-processing algorithms exhibit the property where their computation and memory access components can be specified independently. Following the principles of decoupled access-execute [84] mechanism, we propose an architecture combining stream and dataflow abstractions –

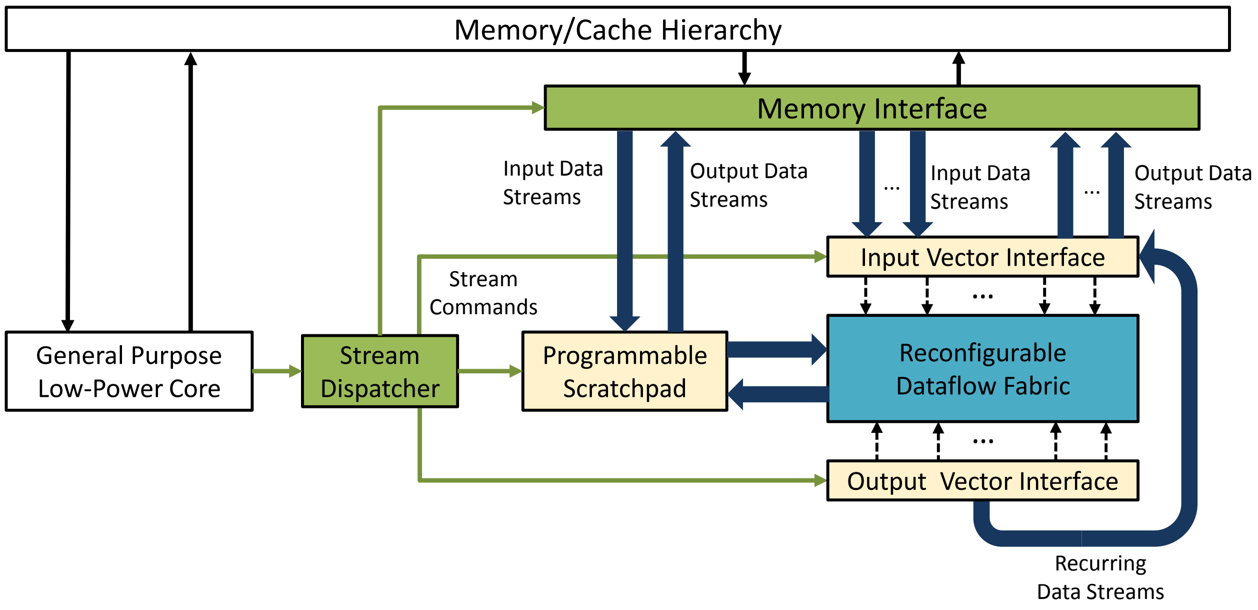


Figure 6.3: Stream-Dataflow Programmable Accelerator

stream-dataflow. The stream component exposes a vector-like memory interface, and the dataflow component exposes a spatial specification of computation.

The stream-dataflow paradigm is naturally derived from these insights and observations. By restricting the scope of data-parallel programs, a stream-dataflow model can obviate the need for many of the overhead structures in data-parallel hardware. To explain, we briefly describe the stream-dataflow accelerator below and then explain the opportunities.

6.3.1 Stream-Dataflow Programmable Accelerator

Inspired from the specialization principles and mechanisms needed for a programmable accelerator discussed in Chapter 4, at a high-level *stream-dataflow* accelerator is also designed with a programmable scratchpad memory for data-reuse, a coarse-grained reconfigurable dataflow fabric (CGRA) for computation and communication specialization, and a memory interface facilitating easy communication with memory hierarchy, as shown in the Figure 6.3. A special vector interface is embedded on input and output side of the CGRA to facilitate vector-wide execution. The parallel

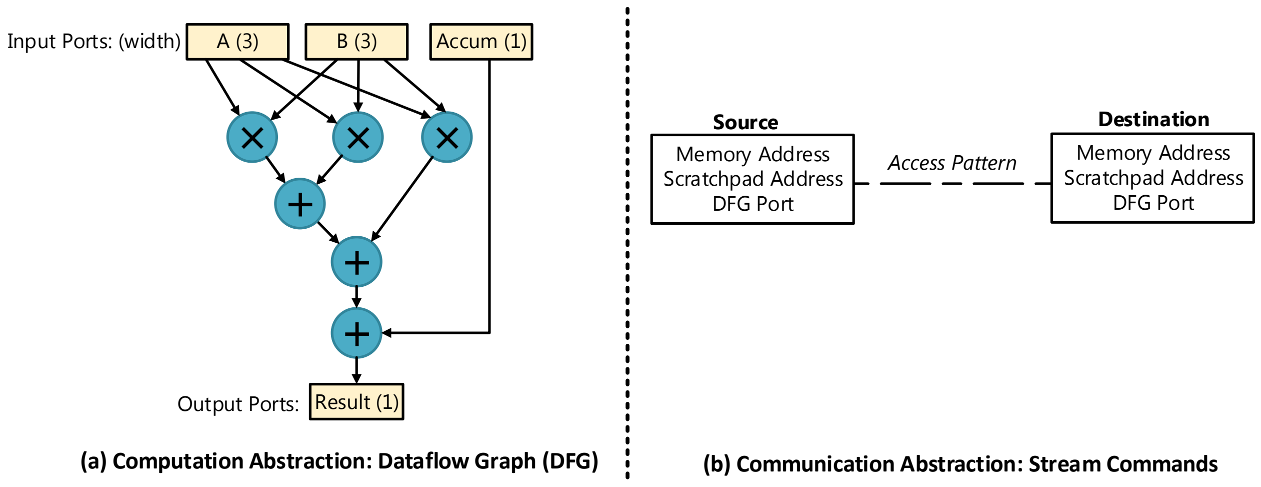


Figure 6.4: Stream-Dataflow Abstractions

communication between these hardware units is controlled by arbitrary-length stream commands issued from simple control core and so the control becomes distributed among concurrent units of the architecture. A stream dispatcher enforces architectural dependences between data streams. The control core can execute arbitrary programs, but is programmed to offload as much work as possible to the stream-dataflow hardware.

Abstractions: The following is a brief overview of the abstractions, as shown in Figure 6.4. The *stream* interface provides support for an ordered set of stream commands, which is embedded within an existing Von Neumann ISA. Stream commands specify long and concurrent patterns of memory accesses and general expressible access patterns include contiguous, strided, and indirect. We add a separate "scratchpad" address space, which can be used to efficiently collect and access re-used data. *Data Streams* are defined by a source and a destination pair along with an access pattern. The dependences between architecture locations are enforced dynamically. Finally, the *dataflow* interface exposes computation instructions and their dependences through a dataflow graph (DFG). The input and output interfaces of the DFG are named ports with configurable vector-width, that are part of the sources and destinations for stream commands.

6.3.2 Opportunities

Stream-dataflow implementations enable coalesced memory accesses and because stream-dataflow model has special support for streaming the data with variable access patterns (discussed in detail in Section 6.5), it can significantly reduce the address generation overheads. Unaligned accesses only becomes burdensome at the beginning and end of stream accesses and rest of the data fits nicely into cache-line based data requests. Strided access patterns are simple to support without additional mask and merge instructions on the critical path unlike vector-SIMD units. There is no redundant address generation or need for memory coalescing, as the memory interface generates access requests at the width of the entire cache-line (64B) or required memory interface. This also enables a cleaner decoupling of access and execute phases of the program without explicitly depending on the memory interface to get all the data to a storage, before starting the computation. The computation phase can overlap with the access phase effectively achieving higher degree of concurrency.

In terms of gaining high resource utilization, stream-dataflow model does require programs to be pipelinable with low initiation intervals, but does not require massive multi-threading support or multi-issue pipeline logic to hide the latency. High utilization is provided by using a dataflow computation substrate, which also avoids large register file access to communicate values. Even the caches do not have to be designed around supporting many independent contexts which in-turn can cause a large working set. Also, because the control is distributed across architectural components, it becomes easier to scale up the hardware concurrency, without needing to rely on general purpose cores which must enforce instruction ordering at a more finer granularity. Flexible-length (typically long) stream commands mean the control core can be very simple, as it is needed only to generate streams and not to manage their execution.

A final opportunity is in regards to stream-dataflow's ability to be easily customizable to a certain application domain. Traditional SIMD-based architectures use wide vector lanes of homogeneous functional units. This makes it more difficult to scale the number of execution resources to the proportional demand inside a particular domain or domains to be targeted. A reconfigurable fabric like CGRA can easily support arbitrary proportions of functional units without causing pipeline inefficiencies. A disadvantage is that fine-grain control relies on predication (power overhead), and coarse-grain control requires reconfiguration (performance overhead). But generally both are somewhat rare in typically accelerated workloads.

6.4 Architecture for Stream-Dataflow Acceleration

In this section, we describe the stream-dataflow architecture through its programming abstractions and an execution model.

6.4.1 Architectural Abstractions

Stream-dataflow architecture abstracts computation as a dataflow graph, and communication as streams and barriers, as shown in Figure 6.4. We describe these abstractions, the associated commands, and any dependencies which must be enforced below.

Dataflow Graph (DFG)

The DFG abstraction is an acyclic graph (as shown in Figure 6.4(a)) containing computation instructions and dependences to be mapped to the CGRA substrate. Note that we do support cycles for direct accumulation, where an instruction produces a value accumulated by a later instance of itself. More general cyclic dependences *are* supported through "recurrence streams", which we describe later.

Inputs and outputs are named as *ports* with explicit vector widths. These ports are used to facilitate vector-wide communication to CGRA substrate. For every set of inputs that arrive at the input ports one set of outputs are generated. The execution is very much similar to the traditional dataflow firing where in this case, the ports are monitored for the arrival of data (input ports) and draining of data (output ports). This one execution iteration of the entire dataflow graph through the input ports, computation units and output ports is called as *computation-instance* in stream-dataflow model. Dataflow graphs can be switched through a configuration command, which must logically happen after any outstanding computations, initiated by streams.

Streams

Data-Streams are defined by a source architectural location, a destination and an access pattern as shown in Figure 6.4(b). Since a private scratchpad address space is exposed, locations are either a memory address or programmable scratchpad address, or a named DFG port. Ports either represent communication channels to the inputs or outputs of the DFG, or they can be indirect ports which are used to facilitate indirect memory accesses. Streams from DFG outputs to inputs support recurrence. Access patterns for DFG ports are first-in-first-out (FIFO) only, while access patterns for scratchpad and memory can be more complex (linear, strided, repeating, indirect, scatter-gather etc.), but a restricted subset of patterns may be chosen at the expense of generality.

Streams generally execute concurrently, but streams with the same DFG port must logically execute in program order, and streams that write from output DFG ports wait until that data is available. Also, streams from DFG outputs to DFG inputs can be used to represent inter-iteration dependences which we call as recurrence.

Barriers and Concurrency

Barrier instructions serialize the execution of certain types of commands. They include a location, either scratchpad or memory, and a direction (read or write). The semantics is that any following stream command must logically enforce the "happens-before" relationship between itself and any prior commands described by the barrier. For example, a scratch read barrier would enforce that a scratch write which accesses address A must come after a scratch read of A issued before the barrier. Barriers can also optionally serialize the control core, to coordinate when data is available to the host.

Note that in the absence of barriers, all streams are allowed to execute concurrently. Therefore, if two stream-dataflow commands read and write the same scratchpad or memory address, with no barrier in-between them, the semantics of that operation are undefined. It could effectively end up reading or overwriting the corrupted data in the source-destination locations respectively. The same is true between the low-power core memory instructions and the stream-dataflow commands. In general, the principle is that the programmer or compiler is responsible for enforcing memory dependencies.

6.4.2 Programming and Execution Model

A stream-dataflow program consists of an ordered set of configuration, data-stream, and barrier commands, that interact with and are ordered with respect to the instructions of a general program. These are generated from a general purpose low-power core, and their ordering can be considered to be a part of the total program order. Programs generally use the stream-dataflow accelerator in phases, which are initiated by reading memory and end at a final barrier, after which the core can initiate its memory instructions. Phases can last anywhere from one to millions of computation instances.

Figure 6.5(a) shows a vector dot-product code region before and after transformation to the stream-dataflow architecture. The memory streams for the accesses of `a`, `b` and `r` are now explicitly represented, and the computation has been completely removed (it is represented in the DFG in Figure 6.4). Also note that the loop is completely removed as well as the loop control is now implicitly coupled with the stream length. This also indicates the vast reduction in the total number of dynamic instructions required on the control core.

Execution Model

Stream-dataflow programs execute in *phases*, each starting with a stream command to initiate data movement and ending at a final barrier which synchronizes the control core. Phases have arbitrary length consisting of many computation instances. A simple example in Figure 6.5(b) demonstrates

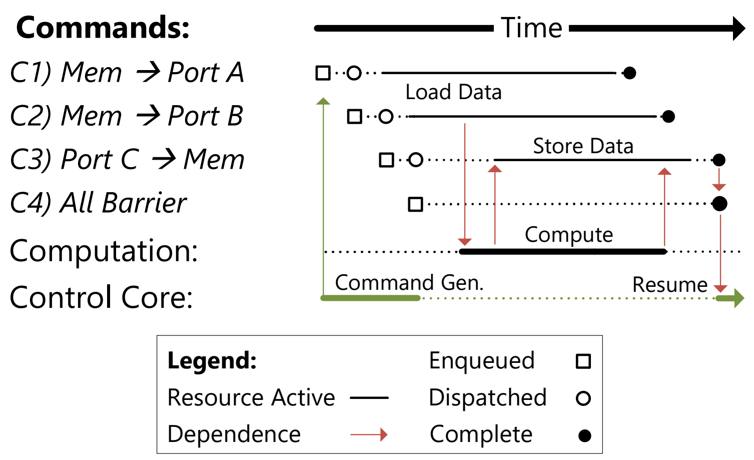
Original Program

```
for(i=0; i<n; ++i) {
  r[i]=a[i].x * b[i].x+
       a[i].y * b[i].y+
       a[i].z * b[i].z;
}
```

Stream-Dataflow Program

```
Load a[0:n]   → Port_A
Load b[0:n]   → Port_B
Store Port_C → r[0:n]
Barrier_All
```

(a) Stream-Dataflow Program Transformation



(b) Stream-Dataflow Execution

Figure 6.5: Stream-Dataflow Program Transformation and Execution

how the execution model exposes concurrency. The state of the stream commands, CGRA, and control core is shown over time. For each stream we note where it was enqueued from the control core, dispatched to execute in parallel, and completed, and we mark the duration in which it has ownership of the source resource for data transfers. The red arrows show dependences between events.

To explain, the first two commands are generated on the control core. They are enqueued for execution and dispatched in sequence, as there are no resource dependences between them. Both streams share the memory fetch bandwidth. As soon as 3 items worth of data are available on ports A and B (3 being the port vector-width), the computation begins. Meanwhile the last two commands are generated and enqueued. As soon as one instance of the computation is complete, the computed data starts streaming to memory. When all data is released into the memory system, the barrier command's condition is met, and the control core resumes.

Performance

The abstractions and execution model lead to intuitive implications for achieving higher-performance. First, the DFG size should be as large as possible to maximize instruction parallelism. Second, streams should be as "long" as possible to avoid instruction overheads on the control core. Third, reused data should be pushed to the scratchpad to reduce bandwidth to memory.

Appendix B explains a more detailed stream-dataflow execution model example including the scratchpad streams and usage.

6.5 Stream Dataflow ISA Interface

In this section, we describe the ISA interface exposed inorder for the low-power core to communicate to the stream-dataflow accelerator. We first describe the access patterns it supports and then present the details of the ISA. At the end of the section, we also illustrate an example of stream-dataflow program.

6.5.1 Access Patterns

We mainly focus on common memory address streams, for which we can build simple hardware to perform address generation for the specified access patterns. The first type of stream we support are – two dimensional affine streams (we refer to these as just *affine* hereafter). They are defined by an *access size* (size of lowest level access), a *stride* (size between consecutive accesses) and *number of strides*. This abstraction along with different example patterns it can generate is shown in Figure 6.6. We emphasize here that only the lowest level of a data-structure in the program should have some stride or contiguous access patterns for this abstraction to be effective, as the low-power core is free to generate arbitrary addresses as the starting locations of streams.

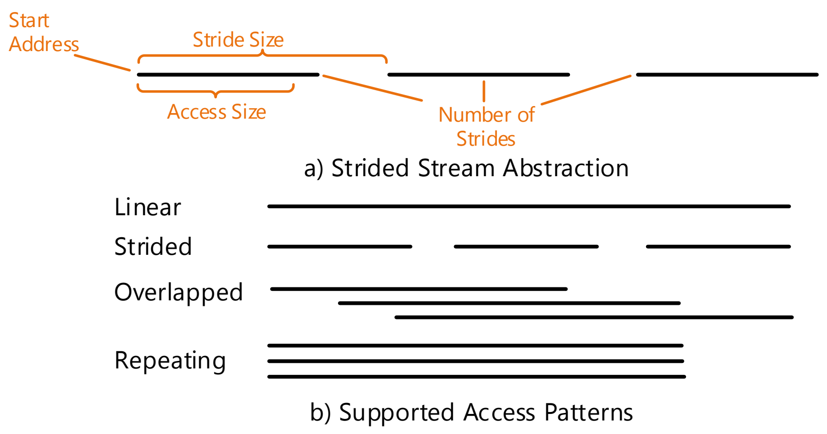


Figure 6.6: 2D Affine Access Patterns

The other type of stream we support are *indirect* streams. Indirect memory operations essentially take another data-stream as input (either an affine stream, or another indirect stream), and uses those values to generate irregular memory addresses. Indirect streams can be chained to create multi-indirect access patterns (eg. `a[b[c[i]]]`). Note, currently the indirect stream abstraction can only support simple irregular memory access patterns with one induction variable. It become complex and induces overheads to support address streams with combination of induction variables like for eg. i + j, i * j and we believe its very rare to see such patterns in acceleratable streaming applications.

6.5.2 Stream Dataflow ISA

Table 6.2 lists the ISA instructions for the stream-dataflow accelerator's programming interface. Note that although, this ISA interface is not as low-level as SIMD SSE intrinsic, on contrary its neither as high-level as an OpenCL API. We believe this is a starting step towards achieving a cleaner API, and at this stage this can act more like a hardware-programming interface (HPI).

Command Name	Parameters	Description
SD_Config	Address, Size	Stream CGRA configuration from given address
SD_Mem_Scratch	Source Mem Address, Stride, Access Size, Num Strides, Dest. Scratch Address	Read from memory with pattern to scratchpad
SD_Scratch_Port	Source Scratch Address, Stride, Access Size, Strides, Input Port #	Read from scratchpad with pattern to input port
SD_Mem_Port	Source Mem Address, Stride, Access Size, Num Strides, Input Port #	Read from memory with pattern to input port
SD_Const_Port	Constant Value, Num Elements, Input Port #	Send constant value to input port
SD_Clean_Port	Num Elements, Output Port #	Throw away some elements from output port
SD_Port_Port	Output Port #, Num Elements, Input Port #	Issue recurrence between input-output port pairs
SD_Port_Scratch	Output Port #, Num Elements, Scratch Address	Write from port to scratchpad
SD_Port_Mem	Output Port #, Stride, Access Size, Num Strides, Dest. Mem Address	Write from port to memory with pattern
SD_Mem_IndPort	Source Mem Address, Stride, Access Size, Num Strides, Indirect Port #	Read the addresses from memory with pattern to indirect port
SD_IndPort_Port	Indirect Port #, Offset Address, Input Port #	Indirect load from addresses present in indirect port
SD_IndPort_Mem	Indirect Port #, Output Port #, Dest. Offset Address	Indirect store to addresses present in indirect port
SD_Barrier_Scratch_Rd	-	Barrier for scratchpad reads
SD_Barrier_Scratch_Wr	-	Barrier for scratchpad writes
SD_Barrier_All	-	Barrier to wait for all commands completion

Table 6.2: Stream-Dataflow ISA (Mem: Memory, Scratch: Scratchpad, IndPort: Indirect Port)

DFG Specification

The first command is a configuration instruction for the computation substrate (CGRA), which takes an address and size of configuration array stored in memory. Configuration data mainly specifies widths of the each vector port used, DFG computation operations and dependences. We leave the specific encoding format of the configuration out of the ISA, as the computation substrate may differ between implementations (CGRA or FPGA), and efficiency is gained only by exposing interface at this level.

Stream Specification

The next set of instructions are for affine stream patterns. One instruction is listed for every combination of source and destination location, and their parameters follows the access pattern convention described earlier, except when the source or destination is a port. In the case of ports, we simply use an element count for the length parameter. We also add a `SD_Const_Port` command to support sending in a constant value several times (rather than loading from memory), and an `SD_Clean_Port` command to dump unneeded values from an output port (rather than reading them to some arbitrary location and then discarding). Both commands are useful in software pipelining the CGRA.

The next set of commands are for the indirect load and store streams. The first instruction in that class is to load the addresses present in memory with the specified pattern to an indirect port. Following two commands take an indirect port as input, and use it to load/store the data from memory/output port to an input port/memory pair respectively. They also take an offset address parameter, which is added to addresses before accessing memory. This is useful for accessing streams where the values are indices rather than direct addresses.

Barrier Specification

The last three instructions are the barrier instructions for scratchpad read/writes and for synchronizing with the low-power core and memory system. Stream commands are embedded into the low-power core ISA to enable fast command communication between the core and stream-dataflow processor. Each command can be embedded as 1-3 instructions in a fixed-width core ISA.

Additional ISA Exposed Elements

In addition to the stream-dataflow commands, there are four other ISA-exposed elements:

1. Functional unit limitations of the CGRA i.e, max total instructions of each type.
2. Maximum width of a DFG port – number of items which can be consumed per stream per cycle. The vector width of a computation.
3. Scratchpad size and the line-width.
4. The longest recurrence allowable.

Recurrence length must be exposed because there is limited buffering in any implementation, and completely filling the input buffer before the associated dependent output buffer can drain may cause deadlock (depending on other dependences). Reasonable values for maximum dependence length is 16-64 for scalar values; note that the scratchpad itself can be used for longer dependence chains (and of course memory) if proper barriers are employed.

These exposed elements have intuitive implications for achieving high-performance. The most important are – i) the DFG size should be as large and as pipelined as possible to maximize instruction parallelism; ii) streams should be as "long" as possible to avoid the overhead of coordinating streams and per-instruction decode overheads; iii) reused data should be pushed to the scratchpad to reduce bandwidth to memory.

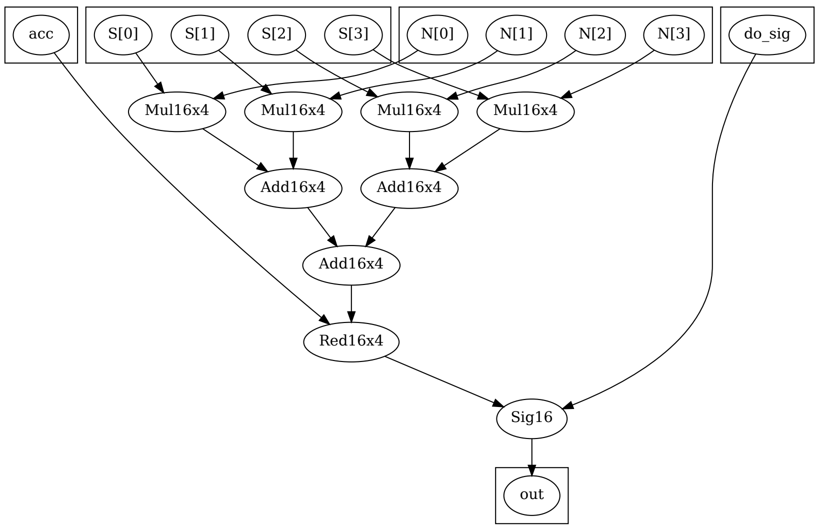


Figure 6.7: Dataflow Graph (DFG) for DNN classifier
(S: Synapse Port, N: Input Neuron Port, out: Output Neuron Port, acc: Accumulator Port, do_sig: Sigmoid Predicate Port; Mul16x4: 4 x 16-bit Multiplier, Add16x4: 4 x 16-bit Adder, Red16x4: 4 x 16-bit Reducer, Sig16: 16-bit Sigmoid)

6.6 Example Stream-Dataflow Program

We now illustrate an example of stream-dataflow program and show the program transformation. Figure 6.8 is an example classifier neural-network layer written in C language, showing the original code and the stream-dataflow program, whose ISA is exposed through the interface detailed in Section 6.5. Figure 6.7 shows the dataflow graph (DFG) mapped to CGRA for the computation. The original program is essentially a dense matrix-vector multiply of synapses and input neurons.

The transformation to a stream-dataflow version pulls the entire loading of neurons and synapses out of the inner-loop with long stream commands (lines 5-8). The input neurons are loaded into scratchpad while simultaneously reading the synapses into the synapse (*S*) port. Inside the single loop (lines 11-17) are streams which co-ordinate the accumulation, activation with sigmoid and then send out the final value for each layer to memory. Note, that the number of instructions executed by the core is dramatically reduced, by roughly a factor of `Ni`, which can be 10-1000$\times$ in practice.

Original Code:

```
// Neural Network Size
#define Ni 784
#define Nn 10

// Synapse and input/output neurons
uint16_t synapse[Nn][Ni];
uint16_t neuron_i[Ni];
uint16_t neuron_n[Nn];

// iterate over each output neuron
for (n = 0; n < Nn; n++) {
  sum = 0;

  // iterate over each input neuron
  for (i = 0; i < Ni; i++){
    sum += synapse[n][i] * neuron_i[i];
  }
  neuron_n[n] = sigmoid(sum);
}
```

Stream-Dataflow Program:

```
// Streaming CGRA configuration
SD_Config(classifier_config)

// Scratchpad load and memory to port loading
SD_Mem_Port(synapse, Ni * 2, Ni * 2, Nn, Port_S)
SD_Mem_Scratch(neuron_i, Ni * 2, Ni * 2, 1, 0)
SD_Barrier_Scratch_Wr()
SD_Scratch_Port(0, Ni * 2, Ni * 2, 1, Port_N)

// iterate over each output neuron
for (n = 0; n < Nn; n++){
  SD_Const_Port(0, 1, Port_acc)
  SD_Const_Port(0, Ni/4 - 1, Port_do_sig)
  SD_Const_Port(1, 1, Port_do_sig)
  SD_Port_Port(Port_out, Ni/4-1, Port_acc)
  SD_Port_Mem(Port_out, 1, &neuron_n[i])
}

SD_Barrier_All();
```

Figure 6.8: Stream-Dataflow Program Example for DNN Classifier

7 | Micro-Architecture of Softbrain

Our goal in constructing a micro-architecture for the stream-dataflow ISA and architecture is to have lower overheads compared to data-parallel architectures and efficiency as close as to an application or domain-specific design. Therefore, we adopt three fundamental design principles in implementing the stream-dataflow ISA:

1. First, we should not introduce any control-heavy or large power hungry structures like multi-ported memories or Content-Addressable-Memories (CAMs).
2. Second, we must exploit the full degree of concurrency that the ISA exposes. Also, the design decisions must support efficient execution of concurrent stream commands with simple resource dependency tracking.
3. Third, we should not significantly hamper any programmability features through architectural decisions.

In the following sections, we describe our stream-dataflow detailed micro-architecture implementation called *Softbrain* and how it accomplishes the above goals.

7.1 Overview

Figure 7.1 shows the high-level overview of the stream-dataflow micro-architecture. There are five core components, which we summarize below, and describe them in detail in the following sections:

- **Reconfigurable Fabric – CGRA:** The coarse-grained reconfigurable architecture enables pipelined computation of dataflow graphs. The spatial nature of the CGRA avoids the overheads of accessing register files or memories for live values and facilitates direct communication of intermediate values produced by the functional units (FUs).

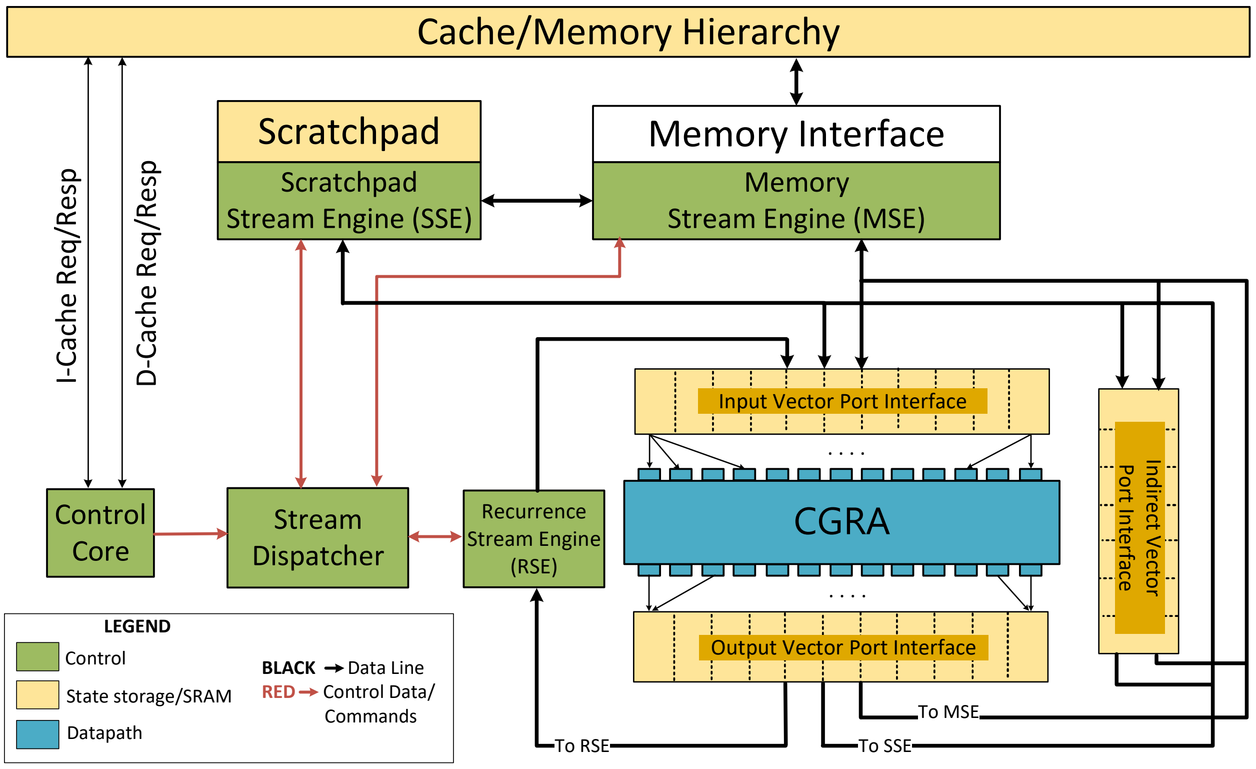

Cache/Memory Hierarchy
I-Cache Req/Resp
D-Cache Req/Resp
Scratchpad
Scratchpad Stream Engine (SSE)
Memory Interface
Memory Stream Engine (MSE)
Input Vector Port Interface
Indirect Vector Port Interface
CGRA
Output Vector Port Interface
Control Core
Stream Dispatcher
Recurrence Stream Engine (RSE)
To RSE
To SSE
To MSE
LEGEND
Control
State storage/SRAM
Datapath
BLACK → Data Line
RED → Control Data/ Commands

Figure 7.1: Stream-Dataflow Micro-Architecture Implementation – Softbrain

- **Vector Ports:** The vector ports are the staging interface elements between the computation performed in the CGRA and the incoming and/or outgoing data streams. They are also responsible for storing the data-elements in vector width specified in the DFG. In addition, a specific type of vector ports called *Indirect vector Ports* not connected to the CGRA are used for storing the address streams of indirect loads/stores.

- **Stream Engines:** Concurrent data-stream communication is carried out through three special control engines called *stream engines* – one for memory facilitating wide memory accesses, one for scratchpad for efficient data-reuse, and one for DFG recurrences enabling immediate re-use of reduced data without memory storage or registers.

- **Stream Dispatcher:** The stream dispatcher manages the concurrent execution of the stream engines by tracking resource dependencies among streams and issued commands to stream

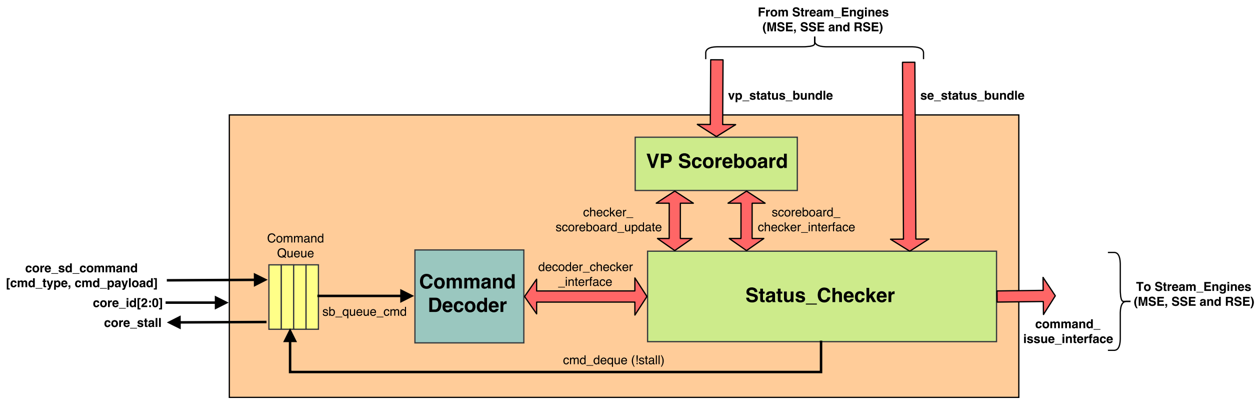


Figure 7.2: Stream Dispatcher Micro-Architecture

engines. It is also the only component which is interfaced to the low-power core which receives the stream-dataflow ISA instructions.

- **Low-Power Control Core:** A tiny, low-power single-issue in-order core which generates ISA instructions for the stream dispatcher. It facilitates programmability without introducing much power or area overhead.

7.2 Stream Dispatcher and Low-Power Core Integration

The role of the stream dispatcher is to issue the stream commands to the requested stream-engines as soon as there is storage available to take new stream commands, and also if there are no concurrently executing streams with the same resource dependencies. It also enables synchronization with the core. Figure 7.2 shows the micro-architecture details of the stream-dispatcher.

7.2.1 Resource Management

Subsequent streams that have the same source or destination port must be issued in program order, i.e., the dynamic order of the streams on the control core. The stream dispatch unit is responsible for maintaining this order, and does so by tracking vector port and stream engine status in a scoreboard. Before issuing a stream, it checks the state of these scoreboards.

Memory based dependencies are not implicitly enforced, but dependencies through the vector ports are, and these are tracked through the vector port status scoreboard (VP scoreboard in Figure 7.2). The state of a vector port is either *taken*, *free*, or *all-requests-in-flight*. The final state indicates that all the requests for a memory stream are completely sent to the memory system, but the response have not yet arrived or still in streaming stage. This state exists as an optimization to enable two memory streams using the same vector port to have their requests in-flight to memory system at the same time enabling highly-concurrent execution of data-streams.

7.2.2 Barriers

Recall that barriers can enforce ordering for either memory or scratchpad accesses as mentioned in Section 6.5. The approach we take for implementing these is simple – we issue the barrier commands to respective stream-engine (scratchpad or memory) and the stream-engines are responsible for not processing any new stream command until the on-going barrier's condition is met. For example, no outstanding scratchpad writes, while the scratchpad read command has been issued. This works efficiently because the other active streams can continue to perform useful work while the stream-engine waits for the completion of enforced barrier. The *SD_BARRIER_ALL* is a special synchronous barrier instruction which allows the control core to stall to not process any new stream-dataflow instructions to respect the memory consistency model. This restriction can be relaxed and programmer has the flexibility to make sure the data operated by control core gets visible to the Softbrain accelerator.

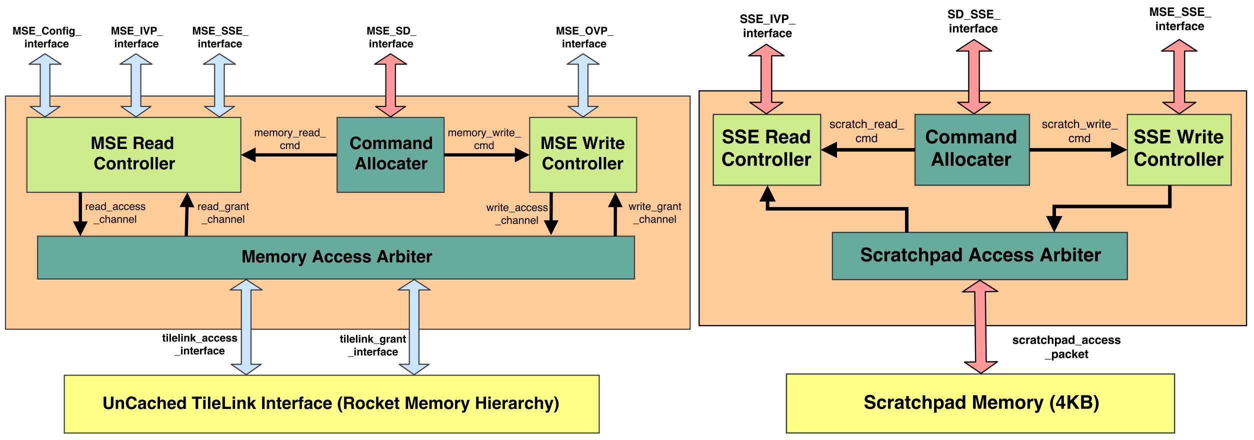
7.2.3 Interface to the Low-Power Core

The basic interface to the core is straight-forward with the core sending the encoded stream-dataflow ISA instructions to stream-dispatcher's command queue. The dispatcher also has a stall interface back to core as a back-pressure signal. Stalls occur when either the stream dispatcher cannot accept any more commands, or when a `SD_Barrier_All` command is in the command queue. The command queue entries is configurable and can be parameterized based on the application domain you want to target. For our implementation of the control core, we rely on the readily available open-source based RISCV Rocket Core [85]. Its a simple 5-stage pipelined single issue in-order processor core which executes RISCV64G instruction set. To interface into our accelerator, we only need to modify a special accelerator interface specific decode stage and hazard generation logic of the Rocket core. This special interface for accelerator integration in rocket core is called as Rocket Custom Co-Processor (RoCC) and enables easy and non-intrusive integration of loosely-coupled accelerators. We do not discuss the core's datapath here and interested readers can look at the detailed Rocket's micro-architecture here in Appendix C.

7.3 Stream Engines

We now describe the critical component responsible for address generation for the data-streams based on the access patterns. Stream engines manage concurrent access to various resources (memory interface, scratchpad, output vector port) by many active streams. They are critical to achieving high parallelism with low power overhead, by fully utilizing the associated resources through arbitrating stream accesses. There are total three stream engines – for memory, scratchpad and recurrence control. Figure 7.3 shows the micro-architecture of two stream-engines – memory stream-engine (MSE) and scratchpad stream-engine (SSE).

All stream engines receive commands from the stream dispatcher, coordinate data transfer, and send notification signals back to the dispatcher when the corresponding vector ports are freed or when the streams complete after stream-command execution. The stream engines take data-inputs from memory or scratchpad and deliver it to the vector ports or memory, and each has their own dedicated wide bus interface (512 bits in our implementation) for both reads and writes. The stream dispatcher ensures that each stream engine has dedicated write access to the vector ports, which are used by its active data-streams. Indirect access is facilitated by streams sent to indirect ports which are not connected to the CGRA, but instead can be the source for indirect streams issued to



(a) Memory Stream Engine (MSE) **(b)** Scratchpad Stream Engine

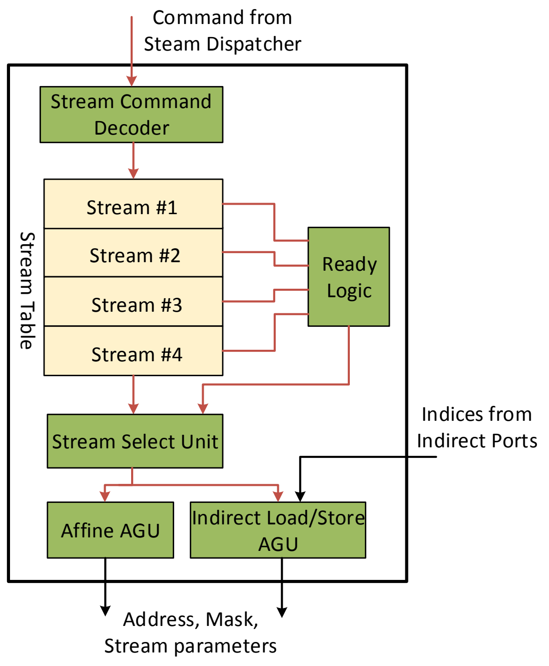
Figure 7.3: Micro-Architecture of Two Stream Engines

either scratchpad or memory.

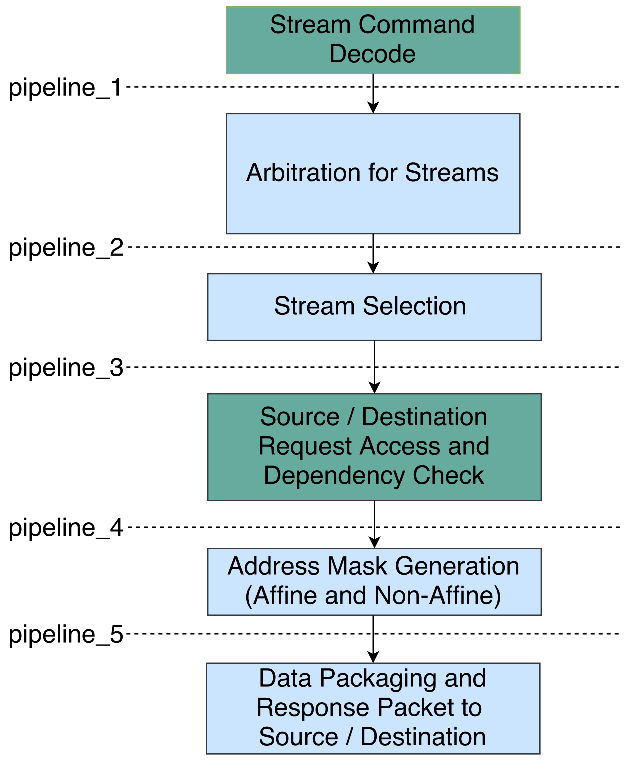
The rest of this subsection first describes the central control unit in each stream engine – stream controller along with its stream request pipeline and then describes specific design aspects of each stream engine.

7.3.1 Stream Controller

The primary role of this unit is to select multiple active stream requests inorder to facilitate sending or receiving of the associated data. Figure 7.4(a) shows the template for a stream engine controller along with its request pipeline stages shown in detail in Figure 7.4(b) which is tailored slightly for each type of stream engine. First, the incoming stream command is decoded for its functionality and is inserted into a stream table which maintains a set of active streams with each entry containing the associated data for a stream request. A selector/arbiter uses this to determine a ready stream for issue based on either a round-robin fashion or a more balanced arbitration based on the data-holding capacity and data currently available in vector ports. This is the combined stage of stream selection and dependency checking in the pipeline diagram. A stream is ready if its destination is not full and its source has data (if it has one).



(a) Stream Engine Controller



Stream Command Decode
pipeline_1
Arbitration for Streams
pipeline_2
Stream Selection
pipeline_3
Source / Destination Request Access and Dependency Check
pipeline_4
Address Mask Generation (Affine and Non-Affine)
pipeline_5
Data Packaging and Response Packet to Source / Destination

(b) Stream Engine Controller Request Pipeline Stages

Figure 7.4: Details of Stream-Engine Controller

If the ready stream is for an affine access, the state is sent to an address generation unit (affine AGU), which computes the next 64B aligned address. It also generates a mask to indicate which 8B words of the mask are relevant, based on the access size, stride size and number of strides parameters. For indirect accesses, an indirect address generation unit will perform the same function, except that it takes address values from an indirect port. This unit will attempt to combine up to eight word addresses, if they are increasing and in the same cache line.

7.3.2 Memory Stream Engine (MSE)

A memory stream engine delivers data from or to the memory system, which in case of Softbrain could be a wide-interface L2 cache. Figure 7.3(a) shows the internal details of MSE. The read and write engines are separated, and have their own independent stream request pipelines. The memory *read* engine has buffering for outstanding requests, and uses a balance arbitration unit (discussed in Section 7.5 for stream priority selection. The back-pressure signal for memory reads to the CGRA is the number of entries free in the buffers associated with the vector ports. For handling back-pressure on scratchpad writes, a buffer sits between the MSE and SSE. This buffer is allocated on a request to memory to ensure space exists. The memory *write* engine uses the data available signals from vector ports for priority selection.

7.3.3 Scratchpad Stream Engine (SSE)

This unit is similar to the above, except that it manages a scratchpad memory. A single-read, single-write ported SRAM is sufficient, and its width is sized proportional to the maximum data consumption rate of the CGRA. Similar to the memory stream engine, the back-pressure signal is the number of free buffer entries on the vector ports. If there are no entries available i.e, if there is back-pressure, then the corresponding stream will not be selected for loading data. Multiple concurrent streams reading and writing into scratchpad can be achieved with a more complex address checking unit to check overlapping of read-write addresses along with a 2-port scratchpad memory. Figure 7.3(b) shows the details of SSE.

7.3.4 Reduction/Recurrence Stream Engine (RSE)

A reduction or recurrence stream engine delivers data from the output to input vector ports for efficiently handling dependences without writes to any memories or temporary storage. It also is used for sending in "constants" from the stream-commands. It does not require the address generation unit in its stream request pipeline.

7.4 Computation and Data Firing

The vector ports and CGRA form a deeply pipelined concurrent execution substrate. Figure 7.5 gives a high-level overview of CGRA substrate interface with the input and output vector ports.

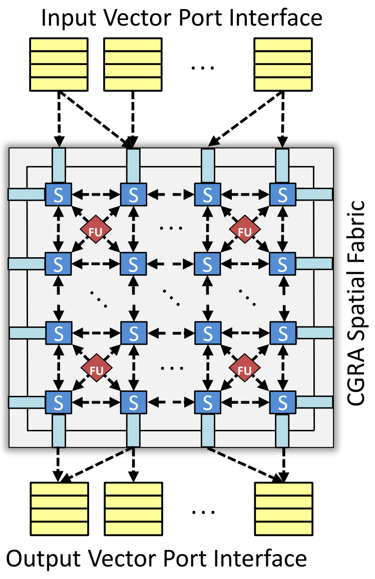


Figure 7.5: CGRA Spatial Fabric and Vector Port Interface

7.4.1 CGRA

Our CGRA acting as the deeply pipelined spatial architecture computation substrate is similar to prior proposals [86, 52]. Specifically, it is a circuit-switched mesh of processing elements, with each tile of mesh containing a set of pipelined functional units. It differs from the referenced designs in that there is no flow-control inside the mesh and is completely statically scheduled (reduces the area and power of the CGRA). This is enabled by the synchronized dataflow firing of input vectors. The lack of flow-control also requires the DFG compiler to ensure delay-matching along all computation paths, including to the output vectors ports. The CGRA's datapath is 64-bit in our implementation, and functional units can perform multiple sub-word operations including 32-bit and 16-bit. Dataflow firing occurs in a coarse-grained fashion, when one instance worth of data (equal to the vector-port width) is available on all relevant vector ports.

The CGRA's configuration is initialized by the `SD_Config` command, which is sent to the memory stream engine (MSE) to fetch the data configuration, which then is streamed to CGRA in less than 10 cycles if the configuration bytes hit in the cache.

7.4.2 Vector Ports

As alluded previously, vector ports are the interface between the CGRA and stream engines, and are essentially 512-bit wide FIFOs that hold values waiting to be consumed by the CGRA. Each vector port can accept or send a variable number of words per cycle, up to 8 64-bit words depending on the request size. On the CGRA side, vector ports attach to a heterogeneous set of CGRA ports, which are selected to spread incoming/outgoing values around the CGRA to minimize contention. This mapping is fed to the DFG scheduler to map ports of the program DFG to hardware vector ports. Dataflow firing occurs in a coarse-grained fashion, when one instance worth of data is available on all relevant vector ports, all of the relevant data is simultaneously released into the CGRA.

7.5 Cross-cutting design issues

We end this section by describing four important issues regarding the overall design.

7.5.1 Buffering and Deadlocks

The Softbrain unit must avoid deadlock by balancing requests to different vector ports. This balancing only needs to happen with a stream engine, as each stream engine owns a single resource, and can operate independently. Deadlock can occur, for example, when many long-latency operations for a single port fill the request pipeline to memory, but data is needed on another port. This can happen when one stream is strided, but the other is linear, so the effective bandwidth coming into the port is much higher.

We solve this issue by adding a balance unit to the memory load stream engine. It tracks the amount-of-unbalance for each active vector port, and heavily unbalanced ports are de-prioritized for stream access.

7.5.2 Memory Coherence

Because Softbrain's memory interface directly accesses the L2 cache, there is the possibility of incoherence between the control core's L1 and the L2. To avoid incoherent reads to the Softbrain, the control core's L1 is write-through. To avoid incoherent reads on the core, the Softbrain sends tag invalidation to the control core whenever a stream crosses a page boundary.

7.5.3 Role of the Compiler and Programmer

In this work, we express programs directly in terms of intrinsics for the stream-dataflow commands (see Figure 6.8 in Section 6.6). So the primary compiler task is to generate an appropriate CGRA configuration for the DFG and vector port mapping. The DFGs are specified in a simple graph language, and we extend an integer linear optimization based scheduling approach from prior work [40].

Though programming is low-level, the primitives are more flexible than their SIMD counterparts. Compiling to a stream-dataflow ISA from a higher level language (OpenCL/OpenMP/OpenAcc) seems practical and quite useful, especially to scale the design to more complex workloads. This is future work.

Integration Softbrain can be integrated into a broader system in a number of ways, including as a unit in an SoC, through unified virtual memory, or as a standalone chip. In this work we assume a standalone device for evaluation purposes. It is possible to support integration to a unified virtual memory with coherent caches. Address translation could be supported at the L2 level (making L1 and L2 virtual) if dedicated accelerator access is assumed, or by integrating TLBs in the memory stream engine.

There are other design aspects critical to certain settings and systems, like support for precise exceptions, backwards compatibility, security and virtualization. These are deferred for future work.

8 | Evaluation of Softbrain

In this chapter, we discuss the detailed implementation of Stream-Dataflow accelerator (Softbrain), the experimental methodology and evaluated results of two differently provisioned Softbrain's compared to state-of-the art domain-specific accelerators. We also correlate our initial results from the generic programmable accelerator model GenAccel which was evaluated for four application domains.

8.1 Implementation

Figure 8.1 shows an overview of our implementation and evaluation methodlogy. This is described in terms of hardware, software stack, and simulation below.

8.1.1 Hardware

We implemented the design from Chapter 7 in Chisel [41]. The design is parameterizable where CGRA size, FU types, vector-port widths, scratchpad size and line-width can be modified using an architecture description model file written in a domain-specific language. This model file is shared with the software stack and the performance simulator.

8.1.2 Software Stack

For the programming interface implementation, we create a simple wrapper API that is mapped down into the RISCV-encoding of the stream-dataflow ISA. We modified a GCC cross compiler for RISCV with stream-dataflow ISA extensions, and implemented our own DFG compiler based on the integer-linear-programming based scheduler from the related work [40]. Though programming in stream-dataflow is low-level, the primitives are much more flexible than vector versions. Compiling

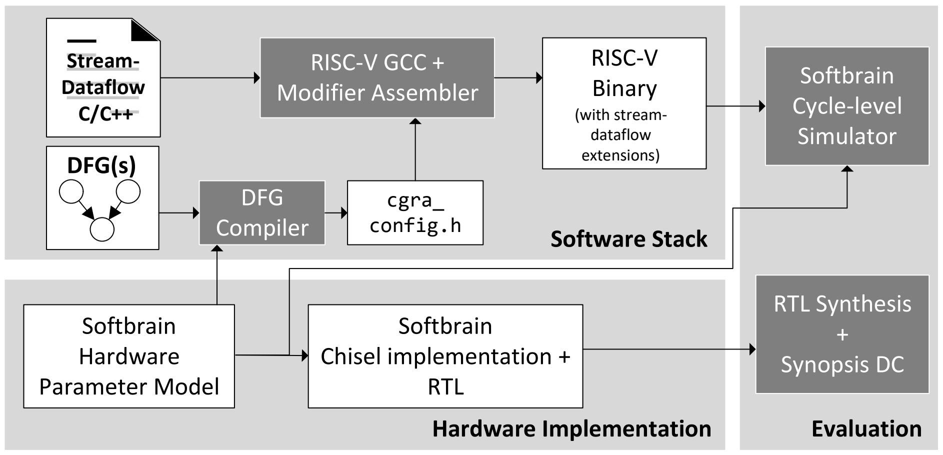

Stream-Dataflow C/C++
DFG(s)
RISC-V GCC + Modifier Assembler
RISC-V Binary (with stream-dataflow extensions)
DFG Compiler
cgra_ config.h
Software Stack
Softbrain Cycle-level Simulator
Softbrain Hardware Parameter Model
Softbrain Chisel implementation + RTL
Hardware Implementation
RTL Synthesis + Synopsis DC
Evaluation

Figure 8.1: Software and Hardware Evaluation Flow of Softbrain

to a stream-dataflow ISA from a higher level language (OpenCL/OpenMP/OpenAcc) is possible but is out of the scope for this dissertation.

8.1.3 Simulator

We implement a cycle-level RISCV based simulator for the core and the Softbrain hardware. The Softbrain simulator is a simple module, which takes stream-dataflow commands and integrates with the core's cache interface for load/store requests and this module can be integrated easily to other core simulators also. The core we use in this simulator is an inorder core. The simulator is regression checked against our RTL modules.

8.1.4 Hardware-Software Workflow

In practice, the hardware would be provisioned once per chip family. For instance, if it is known ahead of time that all data types for a particular application market were a maximum of 16-bit (eg. for machine learning), this could be incorporated into the functional unit composition of the CGRA. Here, an architect either uses existing knowledge or profiles applications from the domain(s) in question. Then they would adjust *only* the FU mix and scratchpad size, recording this in the hardware parameter model file. Even in this case, no Chisel or hardware interfaces need modification.

For each application, the developer constructs DFGs of the computation component, and writes the stream coordination program (stream commands embedded into a C program). It is future work to develop compiler techniques to automatically extract/construct DFGs from unmodified programs, and produce stream commands.

8.2 Experiment Methodology

8.2.1 Workloads

To compare against domain specific accelerators, we use the standard deep neural network (DNN) workloads from the DianNao accelerator work [15], including classifier, convolutional and pooling layers. They have high data-parallelism and memory regularity, and vary in their re-use behavior (and thus memory-bandwidth). Our implementations are tiled to capture locality and concurrency.

We also implemented two end-to-end neural network applications based on the DNN kernels from DianNao. These applications include popular image classification convolutional neural network suites – VGGNet and ResNet50. But, the analysis of these applications is out of the scope for this dissertation.

To capture a broader understanding of the efficiency and generality trade offs, we consider MachSuite [87], a set of typically-accelerated workloads. Unlike the DNN workloads, these capture wider program behaviors like regular and irregular memory access patterns, data-dependent control, and varying computation intensity. We compare these designs against application specific versions.

8.2.2 Power and Area

For the power and area of our Softbrain implementation, we synthesize the Chisel-generated verilog with Synopsis DC and the ARM 55nm technology library, which meets timing at 1GHz. We use Cacti [61] for SRAM and caches estimates.

8.2.3 Comparison Methodology

For the DNN workloads we compare against the DianNao accelerator using a simple performance model. This model optimistically assumes perfect hardware pipelining and scratchpad reuse. It is only bound by parallelism in the neural network topology and by memory bandwidth. We take the power/area numbers from the relevant publication [15]. For comparison points, we consider single-threaded CPU implementations, running on a i7 2600K Sandy Bridge machine. We also compare against GPGPU implementations of these workloads written in CUDA, running on a Kepler-based GTX 750. It has 4 SMs and 512 total CUDA cores.

For comparing to MachSuite accelerators, we use Aladdin [29], a pre-RTL accelerator modeling tool. Aladdin determines the fixed-function accelerator performance, power, and area given a set of prescribed hardware transformations (eg. loop unrolling, loop flattening, memory array partitioning and software pipelining, which impact the datapath and scratchpad/caches sizes). We describe a detailed design-exploration technique we used to evaluate the area-power of ASIC generated from Aladdin with 1-tile of Softbrain unit running each of the Machsuite kernel.

8.3 Evaluation

In this section, we address five important questions for evaluating Softbrain, and we list them here with brief answers for each:

1. What are the sources of its power and area overhead? – Predominantly CGRA network and control core with caches.

2. Can it match the speedup of a domain specialized accelerator? – Yes, when rightly provisioned with computation resources, softbrain can match the performance with limited power and area overheads.

3. Is the stream-dataflow paradigm general? – Yes, All DNN and most MachSuite are implementable using the stream-dataflow abstractions with efficiency close to their domain/application specific implementations.

4. What are its limitations in terms of generality? – Algorithmic properties that are not suitable for softbrain include arbitrary memory-indirection and aliasing, control-dependent loads, and bit-level manipulations.

5. How does stream-dataflow compare to application-specific hardware? – Only 2× power and 8× area overhead with performance same as the custom ASIC hardware for each of the application.

8.3.1 Domain-Specific Accelerator Comparison

Here we explore the power and area of Softbrain compared to a domain-specific accelerator for deep neural networks, DianNao. Our approach is to compare designs with equivalent performance, and examine the area and power overheads of Softbrain.

Area and Power Comparison

To make an intuitive comparison, we configure the Softbrain's functional units to meet the needs of the DNN workloads. Here, we need four-way 16-bit subword-SIMD multipliers and ALUs, and a 16-bit sigmoid. We also include 8 total tiles (Softbrain units), which enables Softbrain to reach the same number of functional units as DianNao.

Table 8.1 shows the breakdowns of area and power. All the analysis is normalized to 55nm process technology. For the power calculations here, we use the maximum activity factors across

		Area(mm^2)	Power (mw)
Control Core + 16kB I & D$		0.16	39.1
CGRA	Network	0.12	31.2
	FUs (4×5)	0.04	24.4
	Total CGRA	0.16	55.6
5×Stream Engines		0.02	18.3
Scratchpad (4KB)		0.1	2.6
Vector Ports (Input & Output)		0.03	3.6
1 Softbrain Total		**0.47**	119.3
8 Softbrain Units		3.76	954.4
DianNao		2.16	418.3
Softbrain / DianNao Overhead		1.74	2.28

Table 8.1: Area and Power Breakdown / Comparison of Domain-Provisioned Softbrain
(All numbers normalized to 55nm process technology)

the DNN workloads. The majority of the area comes from the CGRA network, consuming about one-fourth of the total area and power. The other large factor is the control core, which consumes a third of the power and the area. Compared to the DianNao accelerator, Softbrain is only about 1.75× more power and a little over twice as large.

Performance Figure 8.2 shows the speedups of the Kepler GPU, DianNao and Softbrain for the three classes of DNN workloads. Overall, the GPU is able to obtain up to 20× performance improvement, while DianNao and Softbrain achieve similar performance, around 100× or more on some workloads. The reason is intuitive: both architectures use the same basic algorithm, and they are able to keep 100s of FUs active in every cycle by decoupling memory access from deep pipelined computation. Softbrain does see some advantage in pooling workloads, as its more flexible network

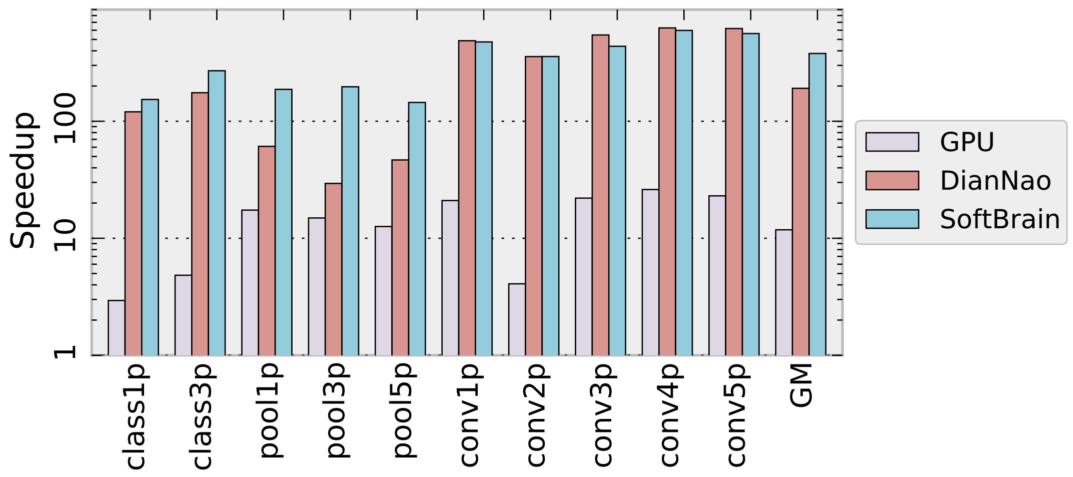


Figure 8.2: Performance of Softbrain on DNN Workloads.

allows it to reuse many of the subsequent partial sums in neighboring computations, rather than re-fetching them from memory. This allows it to reduce bandwidth and improve speedup.

8.3.2 Stream-Dataflow Generality

Here we attempt to distill the limitations of the stream-dataflow accelerator in terms of its generality. To this end, we study a broader set of typically-accelerated workloads from MachSuite, and provision a single design for them. We first characterize our implementations of these workloads and the limitations we discovered.

Softbrain Provisioning

To provision Softbrain's FU resources, we implemented stream-dataflow versions of the MachSuite workloads targeting a maximum of 20 DFG instructions (the same size we used for the DianNao comparison). We then provisioned Softbrain's FU mix to the maximum needed across workloads. Note that for consistency we used 64-bit integer/fixed-point datatypes. Note that using floating point would have decreased the relative overheads of Softbrain versus an ASIC, but also decreased the area/power benefits of acceleration slightly. Hereafter, we refer to this as the broadly provisioned Softbrain.

Implemented Codes	Stream Patterns	Datapath
`bfs`	Indirect Loads/Stores, Recurrence	Compare/Increment
`gemm`	Affine, Recurrence	8-Way Multiply-Accumulate
`md-knn`	Indirect Loads, Recurrence	Large Irregular Datapath
`spmv-crs`	Indirect, Linear	Single Multiply-Accumulate
`spmv-ellpack`	Indirect, Linear, Recurrence	4-Way Multiply-Accumulate
`stencil2d`	Affine, Recurrence	8-Way Multiply-Accumulate
`stencil3d`	Affine	6-1 Reduce and Multiplier Tree
`viterbi`	Recurrence, Linear	4-Way Add-Minimize Tree
Unsuitable Codes	**Reason**	
`aes`	Byte-level data manipulation	
`kmp`	Multi-level indirect pointer access	
`merge-sort`	Fine-grain data-dependent loads/control	
`radix-sort`	Concurrent reads/writes to same address	

Table 8.2: Workload Characterization

Softbrain Generality

Table 8.2 summarizes the architectural abstractions used in the stream-dataflow program implementations. It describes the streaming patterns and datapath structure. There were 4 additional workloads which we have not yet implemented, but do fit into the stream-dataflow paradigm – `fft`, `md`(gridding version), `nw` and `backprop`.

We found that each architectural feature was useful across several workloads. Affine accesses were used in `gemm` and `stencil` codes to reduce access penalties. Indirect loads or stores were required in four workloads (`bfs` and `knn`, and `spmv` versions). Recurrence patterns were useful across most workloads, mainly for reduction variables. The size and configuration of the datapath varies greatly, from reduction trees, SIMD-style datapaths, and more irregular datapaths, suggesting that the flexible CGRA was useful.

There were four workloads that we found could not be efficiently implemented in stream-dataflow. The `aes` encryption workload required too much byte-level manipulation (access words less than 16-bit) that made it difficult to justify offloading onto a coarse-grained fabric. The `kmp` string matching code requires arbitrary-way indirect loads, and the architecture can only support a finite amount of indirection in an efficient way. The `merge-sort` code contains fine-grain data-dependent loads and control instructions, used to decide which list to read from next. The `radix-sort` workload has several phases where reads or writes during that phase could be to the same address (and we don't provide hardware support for implicit store-load forwarding).

Overall, Softbrain is quite general and applicable across many data-processing tasks, even some with a significant degree of irregularity. Its limitations are potentially addressable in future work.

8.3.3 Application-Specific Comparison

In this section we compare the broadly provisioned Softbrain from the previous section to customized ASICs generated for each application, in terms of their power, performance, energy and area.

ASIC Design Point Selection

For comparing the broadly provisioned Softbrain with a workload-specific ASIC, we chose to do an iso-performance analysis, while secondarily minimizing ASIC area and power. To explain in detail - for each workload we explore a large ASIC design space by modifying hardware optimization parameters, and find the set of ASIC designs within a certain performance threshold of Softbrain (within 10% where possible). Within these points, we chose a Pareto-optimal ASIC design across power, area, and execution time, where power is given priority over area. Appendix D explains the ASIC design point selection with an example workload and its design space exploration.

Performance

For performance evaluation of Softbrain to ASIC, the execution cycles obtained from our Softbrain RISC-V based simulator is compared to the execution cycles of the benchmark-specific custom accelerator generated from Aladdin. For a fair comparison, we provision the CGRA of Softbrain to have the maximum functional units capability as the ASIC generated for each workload. Figure 8.3

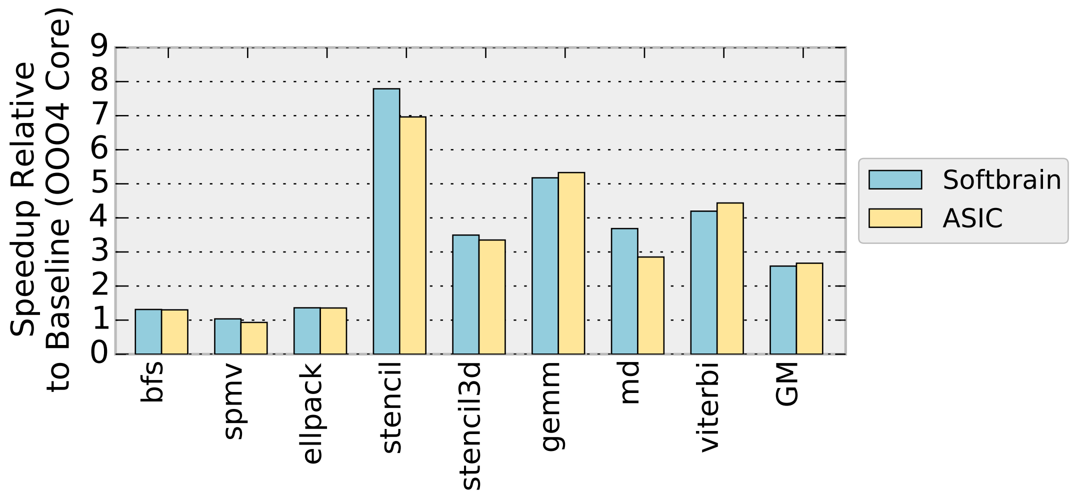


Figure 8.3: Softbrain Performance Comparison to ASIC

shows the performance of Softbrain compared to benchmark specific Pareto optimal ASICs. For both ASIC and Softbrain, the execution cycles are normalized to a SandyBridge OOO4(4-wide) core, with both performing achieving 1-7x speedup. In most cases we found an ASIC design with similar performance to Softbrain[1].

Power, Area and Energy vs. ASIC Designs

As explained above, we choose the iso-performance design points for power, energy and area comparison of Softbrain to ASIC.

For power analysis, we consider that only benchmark-specific FUs are active during execution in Softbrain's CGRA along with other support structures, including the control core, stream engines, scratchpad and vector ports. The static power and area for Softbrain are obtained from the synthesized design, and the dynamic power estimates reported are based on the activity factor of each module. Energy estimates are straightforward to get from execution cycles and dynamic power estimates. ASIC area and power are obtained from Aladdin, using 40nm technology, and are normalized to 55nm.

[1]Note that for some workloads (eg. stencil, md) there is an ASIC deign point with better performance than Softbrain, but falls outside the performance threshold.

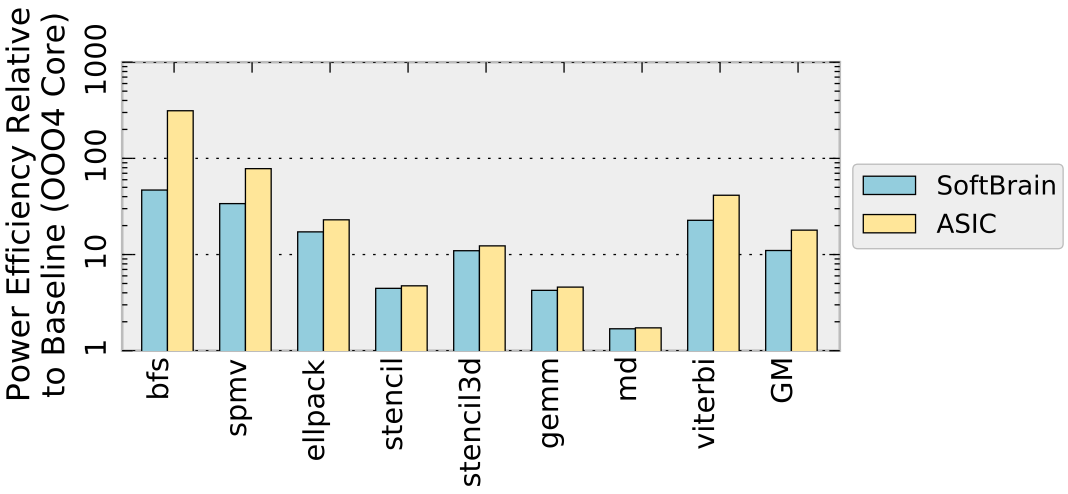


Figure 8.4: Softbrain Power Comparison to ASIC

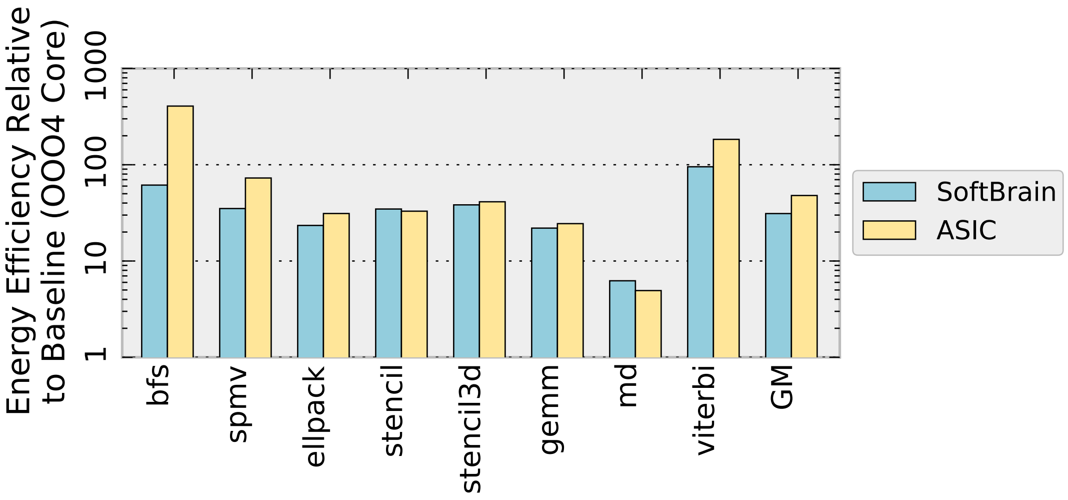


Figure 8.5: Softbrain Energy Comparison to ASIC

Figure 8.4 shows the power savings (efficiency) over a Sandybridge OOO4 core[2] as the baseline. Both ASIC and Softbrain have a large power savings of up to 300x compared to the OOO4 core, which is expected because of the power and area which the OOO4 core spends on supporting generality. ASICs have better power efficiency than Softbrain overall, but only by 2× across all benchmarks. With some workloads, ASIC has almost the same power as Softbrain, and this is

[2]We consider the dynamic power of 1 core at 32nm and scale it to 55nm.

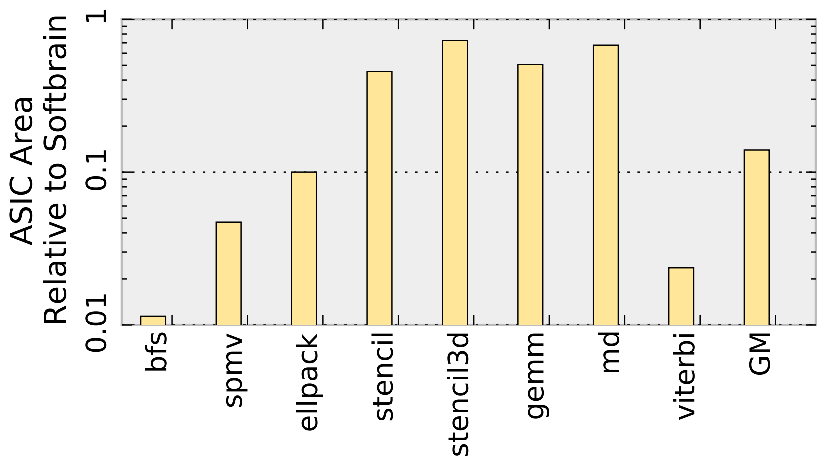


Figure 8.6: Softbrain Area Comparison to ASIC

due to the fact that Aladdin instantiates larger memory structures (scratchpads, buffers etc.) for loop-unrolling, essentially flattening the array data-structures in order for the design space to have performance points close to Softbrain. Note that we include the local memory structures of the ASICs in their power estimation as Softbrain also has a programmable scratchpad. Most of the additional power consumption in Softbrain is because of the generality supporting structures, such as the CGRA network, which is capable of mapping a wide variety of possible DFGs.

Figure 8.5 shows the energy efficiency comparison of Softbrain and the ASICs, showing a high efficiency advantage for both compared to the baseline OOO4 core. The energy consumption of Softbrain is within 2x of ASIC, and this is mainly due to the difference in power consumption.

Figure 8.6 shows the area comparison. As Softbrain's area is fixed across benchmarks, the results show ASIC area relative to Softbrain's area. We do not include the ASIC designs' memory structures area in their estimates, as most of the time these workloads have streaming behavior and ASICs can achieve the same performance with more parallel FUs, rather than larger storage structures[3]. The mean Softbrain area is 8× that of the ASIC, which is expected as Softbrain is programmable and must run all workloads. From another perspective, Softbrain is area efficient, as including all eight MachSuite accelerators would have required 2.54× as much area as only including Softbrain.

[3]Including the memory structure area for the ASIC estimates would make Softbrain look better in comparison.

8.4 Chapter Summary

Overall, Softbrain is competitive with ASICs in terms of performance, power, energy and area, even with the hardware necessary to support significant programmability. This demonstrates that there is scope to develop programmable architectures by tapping the right synergy between the algorithm properties of typically accelerated workloads and the micro-architectural mechanisms supporting stream-dataflow execution.

9 | Related Work

In this chapter, we discuss the related work to this dissertation for both the generic programmable accelerator model and the architectural realization of stream-dataflow accelerator – Softbrain.

9.1 Programmable Specialization Architectures

The literature contains many examples of flexible and efficient specialization architectures. Smart Memories [48], which when configured acts like either a streaming engine or a speculative multiprocessor. One of its primary innovations is mechanisms allowing SRAMs to act as either scratchpads or caches for reuse. Smart Memories is both more complex and more general than GenAccel, though likely less efficient on the regular workloads we target.

Another example is Charm [88]: composable heterogeneous accelerator-rich microprocessor, which integrates coarse-grain configurable FU blocks and scratchpads for reuse specialization. A fundamental difference is in the decoupling of the compute units, reuse structures, and host cores, allowing concurrent programs to share blocks in complex ways. Camel [89] augments this with a fine-grained configurable fabric. These architectures provide more choice in mapping computation but are more complex. The Vector-Thread architecture [78] supports unified vector and multi-threading execution, providing flexibility across data-parallel and irregularly-parallel workloads.

The most similar design in terms of micro-architecture to GenAccel is MorphoSys [86]. It also embeds a low power TinyRisc core, integrated with a CGRA, DMA engine and frame buffer. Here, the frame buffer is not used for data reuse, and the CGRA is more loosely coupled with the host core. Still, we consider MorphoSys to be an instance of an GenAccel. To be clear, the goal of our work is in showing the value of GenAccel model for architectural specialization, not necessarily in significant micro-architectural innovation.

There are also a number of related models for exploring energy trade-offs in heterogeneous environments [90, 91].

9.2 Alternate Approaches

An alternate approach to enable further specialization when area is constrained is to reduce the footprint of DSAs themselves. Lyons et al. explore sharing SRAMs across accelerators [92], and their later work explores virtualizing computation components in FGPAs [93].

Our work leverages the notion of synthesis-time reconfigurability, where architectures can be tuned easily for particular workloads. A prior example of such an approach is Custom Fit Processors [49], which tunes the VLIW instruction organization to a workload set.

9.3 Principles of Specialization

As mentioned earlier, the work by Hameed et al. [3] studies the principles of specialization from the opposite perspective: in identifying the sources of *inefficiency* in a general purpose processor. While both our work and their work argue that the best way forward appears to be augmenting general purpose systems with specialization techniques, their proposed methods differ significantly. In contrast to their work, we argue that large (100-operation) fixed-function units are *not* necessary to bridge the performance gap between general purpose and ASICs, and that a specialized programmable architecture can come close.

9.4 Streaming in Data Parallel ISAs

The concept of exposing streams in a core's ISA to communicate to reconfigurable hardware was proposed in the Reconfigurable Streaming Vector Processor (RSVP) [94]. RSVP uses similar descriptions of affine streams and dataflow graphs, but have several fundamental limitations. RSVP's communication patterns for any given computation cannot change during the streaming phase of the execution. This reduces the flexibility of the types of patterns that can be expressed (eg. ports that are sometimes used for reduction and sometimes written to memory). It also hampers the ability to prefetch data across different phases – one phase must complete before the data for another phase can begin, which eliminates performance benefits when phases are not long. Next, the inter-iteration dependence distance in RSVP can only be 1, which limits programmability. Finally, the address space of RSVP's "scratchpad memory" is not exposed to the programmer, and stream operations cannot address the scratchpad, as they only map linear portions of the address space to the scratch. Allowing general stream patterns to the scratchpad increases generality and also potentially performance if data can be compacted into the scratchpad and read multiple times in a concise format.

An early foundational work in this area is Imagine [95, 96], which is a scalable data parallel architecture for media processing. Imagine uses concepts of streams for explicit communication between memory and a so-called stream register file which acts as a scratchpad for communicating between memory and execution units, as well as between subsequent kernels. Streams here are restricted to being linear and have a maximum size. Streams are also not exposed in the lower level interface for controlling the execution resources: a cluster of VLIW pipelines which are all activated in SIMD fashion by single a micro-controller. A stream based ISA in this context could reduce the complexity of the controlling VLIW core. From a high-level view, Imagine can be viewed as stream-dataflow processors which reads all memory through the scratchpad, and where the reconfigurable fabric is replaced by more rigid SIMD+VLIW execution units.

The problem of efficiently interfacing with reconfigurable hardware also occurs in an FPGA computation offloading environment. CoRAM++ [97] enables data-structure specific API interfaces for transferring data to FPGA-synthesized datapaths, which is implemented with specialized soft-logic for each supported data-structure. This interface is primarily based on streams. DHDL is framework for producing fixed-function accelerators mapped to FPGAs [98]. Other such FPGA efforts include [99, 100, 101].

Finally, while stream-dataflow architectures are micro-architecturally quite different to those of classic dataflow machines [102, 103], an instance of a stream-dataflow computation can be viewed in dataflow terms, where dataflow operators are replaced by an instruction DAG, and their operands are replaced with streams. Dataflow extensions to the GPU model include SGMF [55]. An event-triggered execution model that is orthogonal to dataflow and VonNeumann is Triggered Instructions [104] also co-opted in MAD [105].

9.5 Removing Data-Parallel Architecture Inefficiencies

A number of works attempt to remove the inefficiencies of existing data parallel architectures. One example for SIMT is exploiting value structure to eliminate redundant affine address and value computations [106]. Extensions to classical data-parallel include XLOOPS [107] and Maven-VT [77].

9.6 Heterogeneous Cores

There is a large body of work on combining general purpose cores and reconfigurable or otherwise specialized engines. Those designed for irregular workloads (eg. Composite Cores [108]) are orthogonal to the stream-dataflow ISA. Programmable accelerators targeting data parallelism (eg. DySER [52] or Libra [109]) could benefit from such an architectural interface, and it would be interesting if more general purpose workloads and compilers could target such architectures.

A highly related work is that of Memory Access Dataflow [110], which is another access-execute style architecture. It consists of a general purpose core, some sort of compute fabric, and a reconfigurable fabric to perform address computation and memory access. We considered using a reconfigurable fabric in this work, but found that for the address patterns we needed to support, the overheads would have been in the order of multiple factors in area because of the large CGRA required.

One recurring problem in relying on resource-exposed architectures is binary compatibility. The VEAL work uses a dynamic compilation system to translate the baseline instruction into a template loop accelerator consisting of address generators for memory streams and a modulo scheduled programmable hardware engine [111]. Similar techniques have been proposed to dynamically compile for dataflow-based CGRAs [112]. Such dynamic compilation techniques can be trivially applied to stream-dataflow.

9.7 Streaming in Domain Specific Accelerators

Many domain-specific accelerators use streaming and dataflow abstractions. Eyeriss is a domain-specific accelerator for convolutional neural networks, using streaming access to bring in data, as well as a dataflow substrate for computation [113]. A recent work, Cambricon [114], proposes SIMD instruction set extensions which can perform the stream-like access patterns found in DNNs. Outside the domain of machine learning, Q100 [44] is an accelerator for performing streaming database queries. It uses a stream-based abstraction for accessing database columns, and a dataflow abstraction for performing computations.

10 | Conclusion

This dissertation proposes a new paradigm in hardware acceleration called the "**Programmable Hardware Acceleration**", which enables us to have a programmable architecture composed of specialization elements. It focuses on a hardware-software co-design approach by describing the ISA interface needed for such programmable architectures alongside micro-architecture details for the implementation of the same. A generic programmable accelerator model called GenAccel is developed with simple micro-architectural mechanisms exploiting common specialization principles that pushes the limits of efficiency while retaining generality. GenAccel can achieve efficiency close to domain-specific implementations of the acceleratable applications with only $2\times$ to $4\times$ overheads in area and power. This work also evaluates a particular instance of programmable hardware acceleration called "**Stream-Dataflow Acceleration**" by realizing a detailed architecture with an execution model, an accelerator ISA interface, programming abstractions and the micro-architecture. We implement the stream-dataflow accelerator's micro-architecture in hardware called *Softbrain* and evaluate it with domain-specific and application-specific designs for a wide variety of applications. From our evaluation, we found out that softbrain can match the performance of these custom hardware solutions with minimum overheads in power and energy efficiency and better area efficiency when considered in a SoC environment.

In additions to these results, the dissertation has the following key findings: First, most of the domain-specific hardware accelerators specialize the applications in a common way and can be categorized as architectural principles of specialization comprising of – *Concurrency, Computation, Communication, Data-Reuse and Coordination*. Second, known general micro-architectural mechanisms can exploit the above principles and a generic programmable hardware accelerator can be designed around those. Third, with an efficient ISA interface and programming abstractions exposing the hardware specialization primitives of the accelerator, we can achieve efficiency close to domain-specific hardware and also be future proof to trivially adapt any new accelerator applications out in the market. Fourth, accelerator models and architectures like GenAccel and Softbrain can actually be used as a meaningful baseline for future accelerator research as well as exploring new paradigms with specialization principles and mechanisms.

Finally, we envision that this programmable acceleration paradigm can have a radical simplifying effect on future chips by reducing the number of specialization blocks. An accelerator fabric like Softbrain can sit alongside CPU and GPU processors, with functionality synthesized on the fly as programs encounter suitable phases for efficient offloading. This not only reduces the area and complexity of having vast arrays of specialized accelerators, it also can mitigate growing design and verification costs. In such a broad setting, it will be critical to develop effective compilation tools that can balance the complex trade-offs between parallelism and data reuse that these architectures provide. Overall, we believe that programmable hardware accelerators have a large role to play going forward.

We conclude by discussing the implications of this work and future research directions.

10.1 Implications

The broad intellectual impact of this work is to create a canonical accelerator architecture driving future investigations. As an analogy, the canonical five-stage pipelined processor was simple and effective enough to serve as a framework for almost three decades of big ideas, policies, and micro-architecture mechanisms that drove the general-purpose processor era. GenAccel fabric similarly simple and effective enough – having been shown to be competitive with four award-winning accelerators from different domains.

Up to now, architects have not focused on an equivalent framework for accelerators. Most accelerator proposals are presented as a novel design with a unique composition of mechanisms. Comparing one to another is purportedly meaningless since they target different domains. However, from an intellectual standpoint, our work shows these accelerators are more similar than dissimilar;

they exploit the same essential principles with differences in their implementation. This is why we believe an architecture designed around these principles can serve as a standard framework. Of course, the development of domain-specific accelerators will continue to be critical for architecture research, both to enable the exploration of the limits of acceleration, and as a means to extract new acceleration principles.

To our knowledge, GenAccel model is the first work to generalize accelerators into a common framework. In this role, our architecture can serve as a baseline for comparison and a framework for exploring new policies for future accelerators. And stream-dataflow interface has demonstrated that carefully choosing a set of rich ISA abstractions can simultaneously enable the construction of extremely efficient and lean hardware, while also retaining significant generality and programmability. The Softbrain accelerator, which is competitive with ASICs and domain specific accelerators, is evidence that its ISA principles can be exploited by hardware in practice. The overall significance of the having such programmable accelerators is three-fold – a novel accelerator ISA paradigm with unique benefits can lead to an architecture that can be put it in the same class as Vector, SIMT, and VLIW; it has immediate practical value in easing programmable accelerator development, and it enables specialization beyond just the computational substrate.

A New Baseline to Measure True Benefit of Specialization

In the literature today, DSAs are proposed and compared to conventional high-performance processors and typically yield several orders of magnitude better measurements on various metrics of interest. GenAccel model can serve as a better baseline for future DSAs to compare to, and thus allow measuring the true benefit of specialization. Until the definition of GenAccel and Softbrain, there has been no framework for authors to compare to, besides the generic OOO processor or a GPU - but neither is a good target for distilling out the benefits of specialization. For the four DSAs we have looked at, this true benefit of specialization is only 2$\times$ to 4$\times$ in area and power and basically no advantage in performance (when area & power are provisioned to match performance of the DSA). Using GenAccel as a baseline will reveal more and deeper insights on what techniques are truly needed for a particular problem or domain, as opposed to merely removing the inefficiency of a general-purpose OOO processor using already known techniques applied in a straightforward manner to a new domain. Overall, our work can help decouple accelerator research from workload domains, which we believe can help foster more shared innovation in this space.

Discovering new principles

Orthogonally to using GenAccel as a baseline, it can also serve as a guideline for discovering big ideas for specialization. Undoubtedly, there are additions necessary to the five principles, alternative formulation, and microarchitecture extensions. The definition of the principles and demonstrated coverage on several classes of problems makes specific what types of program behavior are left untouched, which can be covered with new principles. These include ideas which have been demonstrated already in an accelerator's specific context, somewhat straight-forward general principles, and principles not discovered yet.

Considering some accelerator-specific "extensions" that have already been proposed, it is clear how GenAccel can serve as a framework for generalizing their mechanisms and conclusions. As an example, we consider two works from other groups recently - namely Proteus (ICS-2016) and Cnvlutin (ISCA-16). Both have been proposed and studied as mechanisms meant for one existing accelerator (DianNao). The GenAccel frameworks allows those principles to be easily generalized and evaluated across multiple workloads and domains. In essence, the idea of bit-serial multiplication (Proteus) and eliminating zero-computing (Cnvlutin) are equally useful in the database processing and image processing examples our GenAccel framework has considered. We expect future such styles of work can be done on GenAccel, freeing them from being restricted to one accelerator. The framework itself is reproducible completely from the model description provided in the paper and is being made available to researchers on request.

In terms of general principles that can be integrated into GenAccel, it is clear that programs with highly irregular memory accesses but yet exhibiting concurrency are poorly served by GenAccel. DeSC (MICRO-15) has revisited this principle in the general purpose context recently by employing the Decoupled Access/Execute principle. *The GenAccel framework allows the consideration of such ideas in the accelerator context in a non-domain specialized way.*

Finally, our definition of principles makes clear what workload behaviors are currently uncovered and need discovery of new principles to match existing accelerators. This direction leads to the more open question of whether the number of principles are eventually too numerous to be practical to put in a single substrate, whether efficient mechanisms can be discovered to target many principles with a single substrate (memory blocks with a TCAM and SRAM mode for example), or whether they are few in number that one can build a single universal framework. *Enabling and starting this discussion is a key long term impact.*

Embedding principles into existing designs

In designing GenAccel, we took a clean slate approach; we started with a simple concurrent architecture and evolved it by applying specialization principles. However, because these principles are architecture-independent, one point of long-term impact and significance is likely going to be in the adoption of individual principles/mechanisms in existing large core CPUs, GPUs, and FPGAs. Our paper discusses practical strategies to accomplish this by adding hardware mechanisms or restricting the program/hardware scope. Ultimately, these modifications would help to close the general purpose to domain-specific accelerator gap.

An ISA Paradigm for the Specialization Era

Very rarely is a new ISA principle introduced which fundamentally changes architectural tradeoffs, and there are only a few long-lasting paradigms: *VLIW* expresses the independence of instructions, *Vector* expresses an operation over multiple data items, *SIMT*, which encodes multiple threads and their relationship to access locality. We believe that *stream-dataflow* belongs in this category; it expresses arbitrarily long patterns of memory access in a single instruction (and the same for computation in a handful of instructions). In fact, stream-dataflow can be seen as an evolution of the principle of encoding many independent operations with a single instruction. Where as SIMD and and VLIW can reduce the instruction overhead by constant factors, a stream-dataflow ISA can reduce the number of instructions by multiple *dimensions* (eg. a nested loop in SIMD can become a constant number of stream-dataflow commands).

Beyond reducing instruction overheads, stream-dataflow confers other benefits over existing ISA classes. This includes the encoding of the alias free nature of memory instructions, meaning that concurrent memory access can be achieved without dynamic alias analysis. The encoding of memory patterns enables specialized address generation hardware for those patterns, which reduces the hardware overhead over general purpose hardware, reduces the number of cache requests (implicit coalescing), and removes the penalty of unaliased access. Dependences are explicitly encoded, either as instruction dependences (inside iteration) or recurrences (across iterations), and are enforced without consulting memory.

Of course, employing this ISA imposes software challenges. Programming directly is challenging, and automatic compilers need to support for detecting address patterns, loop-interchange/flattening, explicit dependence insertion, memory tiling, and much more – certainly beyond what is required for traditional SIMD. That said, because of where the community is headed with specialization and programmable accelerators, we see stream-dataflow as one of the best choices going forward, with a high potential and (intellectually stimulating) challenges that can be overcome.

Practical Value

Building an extremely low-overhead programmable accelerator with high computational-density is straight-forward with stream-dataflow, which can be immediately useful for industry and researcher. The main principles here are that any complex infrequent program behavior can be relegated to the simple core (for which open source implementations like RISCV Rocket already exist), and the accelerator implementation is made simple because the concurrency and dependences of the program are encoded in the ISA directly, rather than needing to be dynamically discovered with complex hardware. We demonstrated this through the implementation of Softbrain, which took only a modest effort to prototype.

Moreover, Softbrain is merely one example from an enormous design space admitted by the ISA; this space includes local/global memory interface, computational substrate, supported access/control patterns, etc. For example, the design of the computational substrate (the CGRA in Softbrain) is decoupled from the stream-based memory access interface, and can be trivially replaced with any number of engines (SIMD, triggered instructions, or a set of ASIC datapaths). As such, the range of applicable market/application settings for this architecture is quite large – from general purpose big-data processing engines, to smaller and more fixed-function accelerators in mobile or wearable devices and IoT.

Beyond Computational Acceleration

The vast majority of the research effort behind accelerators and specialization has been geared towards making efficient and/or programmable computational pipelines. In any hardware setting that has a general address space and memory, this leaves an enormous amount of energy inefficiency in communication and storage in the cache and memory hierarchy. What would be desirable would be to apply the same principles from computational specialization to the memory system – somehow exposing more of the inherent nature of the program through to the cache and memory hierarchy.

Before stream-dataflow, it was hard to imagine how exactly specialization could be employed in the memory hierarchy. After all, typical memory hierarchies are oblivious to the program which generates requests, and useful information (eg. for prefetching) must be recovered through costly dynamic analysis. However, stream-dataflow encodes coarse grain patterns of access, raising the question of whether this can be used to optimize the memory system, if it was somehow had access to this information. Everything from cache replacement policies, network arbitration, and off-chip data access seems like it could benefit highly by knowing with certainty what the future what and where the "future" program accesses will be from.

10.2 Concluding Thoughts

This dissertation with its proposal of the programmable hardware acceleration paradigm has broken the limits of accelerators, freeing them from being domain-specific ad-hoc engines. It has identified the foundational principles of accelerators, creating a general programmable accelerator fabric which serves as a standard point of reference in exploring new accelerator mechanisms and innovations. It also defines a new accelerator ISA interface that would shape programmable accelerator research during the specialization era. With its detailed architecture, micro-architecture implementation, evaluation and analysis, not only has it opened many difficult questions in programmable accelerator micro-architectures and compilers, but also has inspired and enabled a unification of specialization and memory system research.

A | Architecture Details of the DSAs Studied

In this chapter, we describe the domain-specific accelerators (DSAs) we shave studied and evaluated, for readers to understand about their architecture in detail. Most of the details can be found in the original papers, but we summarize them here for the ease of readers.

A.1 Neural Processing Unit (NPU) for Neural Approximation Acceleration

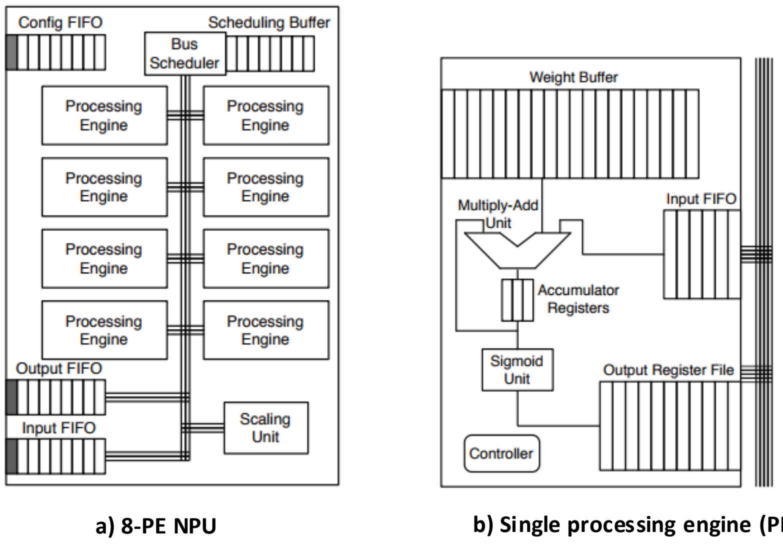


Figure A.1: Architecture Details of Neural Processing Unit (NPU) DSA
(Figure reused from the original NPU [42] paper)

NPU is a DSA for approximate computing using the neural network algorithm, integrated to the host core through a FIFO interface. It exploits the approximation in accelerator code for better performance and energy efficiency. The key idea is to learn how an original region of approximable code behaves and replace the original code with an efficient computation of the learned neural-network model. This work proposes a technique for harnessing the domain-specific hardware for neural networks in general purpose computations. The authors using their evaluation and workflow show that the regions of imperative code can be replaced with neural networks for variety of applications and achieve around 2.x$\times$ speedup, with 3$\times$ energy savings and still have average accuracy of 90% in all cases.

For NPU acceleration, the original approximable code needs to be annotated using some known approaches. The code region also has a requirement to have well-defined inputs and outputs. Once the acceleratable code region has been identified, the compilation workflow implements a transformation called *Parrot transformation* in three steps: observation, training and instrumented binary generation. Observation phase mainly involves the compiler to train the neural network on a realistic data set. It profiles the normal execution of the code with the input-output set and produces a training data set for the next stage – training. In the training phase, the compiler uses the training data from the previous phase and produces a neural network with a topology which replaces the original function. The compiler here uses the back-propagation technique to train the neural networks which are multilayer perceptrons in the NPU case. The final phase of workflow involves the code generation itself where an instrumented binary running on the host core invokes the NPU accelerator when the replaced NPU neural network code is encountered. The program configures the NPU when it is first loaded by sending the topology parameters and synaptic weights to the NPU via a configuration interface. The compiler replaces the calls to the original function with special NPU instructions that sends the inputs to the NPU and collect the outputs from it. These configuration, invocation and collection of outputs is achieved through ISA extensions.

Figure A.1(a) shows the NPU architecture with eight identical processing engines (PEs) along with a bus scheduler and a scaling unit. The scaling unit scales the neural network's inputs and outputs if necessary using scaling factors defined in the NPU configuration process. The PEs in the NPU are statically scheduled. The scheduling information is part of the configuration information for the NPU, which is based on the neural network topology derived during the training process and is populated in the configuration FIFO. In the NPU's static schedule, each neuron in the neural network is assigned to one of the eight PEs. The global bus scheduling information is stored in its circular scheduling buffer and co-ordinates the values being sent between the PEs. Figure A.1(b) shows the internal structure of a single PE. Each PE performs the computation for all of its assigned neurons. NPU implements a sigmoid-activation multilayer perceptron, each neuron computes its output as:

$$y = sigmoid(\sum(x_{\mathrm{i}} \times w_{\mathrm{i}})) \tag{A.1}$$

where x_i is an input to the neuron and w_i is its corresponding weight. More detailed evaluation and workloads used can be found in the paper [42].

A.2 Convolution Engine for Image Processing Acceleration

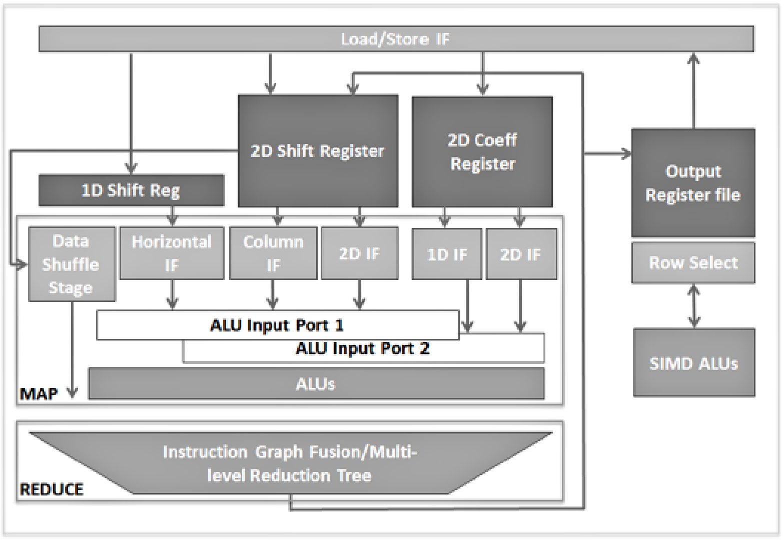


Figure A.2: Architecture Details of Convolution Engine DSA
(Figure reused from the original Convolution Engine [43] paper)

Convolution Engine (CE) is a DSA which accelerates stencil or convolution-like dataflow computations for image processing applications. CE achieves energy efficiency by capturing the data-reuse patterns, eliminating the data transfer overheads and enabling a large number of operations per memory access. Based on the evaluation, authors claim that for most of the image processing applications they are within a factor of 2×-3× the energy and area efficiency of a custom ASIC like hardware built for each kernel. Figure A.2 shows the convolution-engine architecture. The host core uses special instructions to coordinate control of the hardware through a control interface or a load/store interface. It exploits concurrency through both wide-vector and pipeline parallelism and relies heavily on custom memories/registers for storing pixels and coefficients. The interface units (IF) (both row and column) connect the register files to the functional units and provide shifted broadcast to facilitate convolution. Data shuffle (DS) stage combined with Instruction Graph Fusion (IGF) stage form the Complex Graph Fusion Unit. IGF is integrated into the reduction stage for greater flexibility. CE abstracts the convolution operation as a *map* step that transforms each input pixel into an output pixel. In their implementation, the interface units and ALUs work together to implement the map operation; the interface units arrange the data as needed for the particular map pattern and the functional units perform the arithmetic. The *reduce* functionality is provided by the complex graph fusion unit.

CE reduces most of the register file overheads with the help of custom shift register file or a FIFO like storage structure. When such storage structures are augmented with an ability to generate multiple shifted versions of the input data, it can not only facilitate execution of multiple simultaneous stencils, but can also eliminate most of the shortcomings of traditional vector register files. Aided by the ability to broadcast data, these multiple shifted versions can fill sixty four ALUs from just a small 256-bit register file saving valuable register file access energy as well as area. CE facilitates further reductions in energy overheads by supporting more complex operation in the reduction tree, allowing multiple "instructions" to be fused together. This fusion also offers the added benefit of eliminating temporary storage of intermediate data in big register files saving valuable register file energy. Furthermore, by changing the shift register to have two dimensions, and by allowing column accesses and two dimensional shifts, these shift registers possess the potential to extensively improve the energy efficiency of vertical and two dimensional filtering.

As shown in the figure, these 1D and 2D shift registers are the main components facilitating the acceleration. Details about the programming convolution engine, a 2D filter example written for it and the evaluation against SIMD units and custom hardware is explained in their paper [43].

A.3 Q100 DSA for Database-Streaming Acceleration

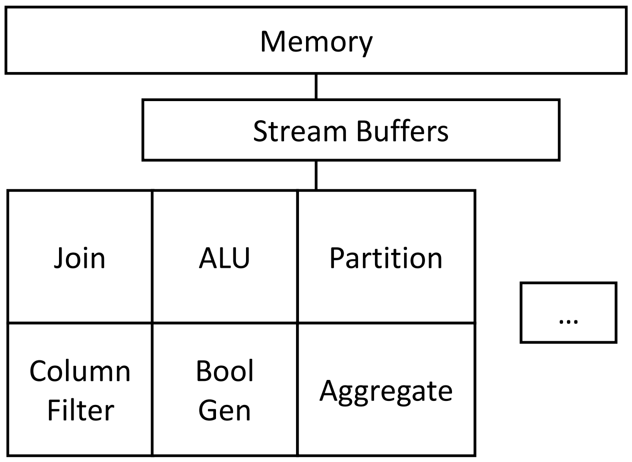


Figure A.3: Architecture Details of Q100 DSA

Q100 data processing unit (DPU) is a DSA for accelerating streaming database queries, which exploits the pipeline concurrency of database operators and intermediate outputs. Q100 contains a heterogeneous collection of fixed function application-specific integrated circuit (ASIC) tiles, each of which implements a well-known database relational operator, such as a join, sort or partition. The Q100 tiles operate on streams of data corresponding to tables and columns, over which the micro-architecture aggressively exploits pipeline and data parallelism. Figure A.3 shows the high-level architecture of Q100 DSA which consists of custom ASIC tiles for the common relational operators. It consists of stream buffers acting as a staging area to prefetch the required database columns.

Q100 specializes the communication by providing dynamically routed channels between FUs to prevent memory spills.

The Q100 ISA implements standard relational operators that manipulate database primitives such as columns, tables, and constants using their instructions. These coarse grained instructions manipulate streams of data, thereby maximizing pipeline and data parallelism, and minimizing the need to time multiplex the accelerator tiles and spill intermediate results to memory. The producer and consumer relationships between operators are captured with dependencies specified by the instruction set architecture. Queries are represented as graphs of these instructions, with the edges representing data dependencies between instructions. For execution, a query is mapped onto a spatial array of specialized processing tiles, each of which carries out one of the primitive functions. The communication between these functional units or tiles are carried out by dynamically routed channels, thus avoiding register file accesses or memory spills. In situations where a query does not fit on the array of available Q100 of tiles, it must be split into multiple temporal stages. These temporal stages are called *temporal instructions* and are executed in order. Each temporal instruction contains a set of spatial instructions, pulling input data from the memory subsystem and pushing completed partial query results back to the memory subsystem The communication network configuration and stream buffers are coordinated using a temporal instruction sequencer.

Q100 contains 11 types of hardware tiles corresponding to 11 operators in ISA. Their work also explores a Q100 design space of 150 configurations, selecting three for further analysis: a small, power-conscious implementation, a high performance implementation, and a balanced design that maximizes performance per Watt. The authors also demonstrate that the power-conscious Q100 handles the TPC-H queries with three orders of magnitude less energy than a state of the art software DBMS, while the performance-oriented design outperforms the same DBMS by 70$\times$. More details on their analysis and query plans used for evaluation can be found in the paper [44].

A.4 DianNao DSA for Deep-Neural Network (DNN) Acceleration

DiannNao is a DSA for accelerating convolutional neural networks (CNN) and deep-learning neural networks (DNN), which are typical machine learning building blocks used for image classification. In this work, the authors have designed an accelerator for large-scale CNNs and DNNs, with a

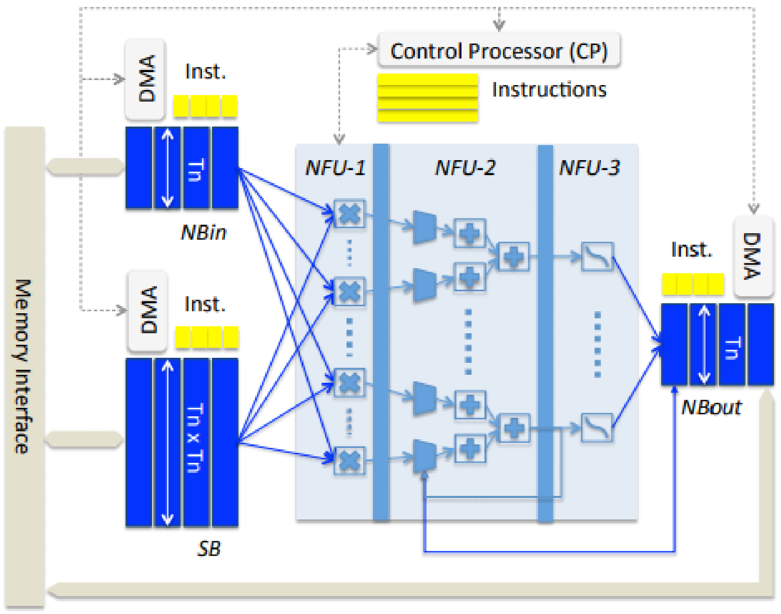


Figure A.4: Architecture Details of DianNao DSA
(Figure reused from the original DianNao [15] paper)

special emphasis on the impact of memory on accelerator design, performance and energy. They show in their evaluation that it is possible to design an accelerator with a high throughput, capable of performing 452 Giga-Operations-Per-Second (GOP/s) for key neural network (NN) operations such as synaptic weight multiplications and neurons outputs additions in a small footprint of 3.02mm^2 area and 485 mW power chip. Compared to a 128-bit 2GHz SIMD processor, the DianNao accelerator is 117.87$\times$ faster, and it can reduce the total energy by 21.08$\times$.

The authors first determine the memory bandwidth bottleneck of typical classifier, convolutional and pooling layers of DNN kernels in the processor based implementation of neural networks. In order to maximize the computation throughput for available memory bandwidth, they design a hardware accelerator composed of Neural Function Unit (NFU) aimed at performing neural computations using hardware neurons. Figure A.4 shows the architecture of DianNao accelerator with multiple units of NFUs. DianNao performs the traditional matrix-multiply and accumulate operations of each neural network layer decomposed into multiple stages with each stage mapped to

NFU. It uses a staggered pipeline to perform the operations. The first or first two stages (respectively for pooling, and for classifier and convolutional layers) operate as normal pipeline stages, but the third stage is only active after all additions have been performed (for classifier and convolutional layers; for pooling layers, there is no operation in the third stage). The accelerator also has split the total storage into three structures to exploit the locality: an input buffer (NBin), an output buffer (NBout) and a synapse buffer (SB). The split buffers are streamed with data values from the memory interface using a DMA engine with explicit DMA instructions. It also makes use of the control instructions issued from the control processor (CP) to coordinate the neural network layer execution. Each layer execution is broken down into a set of instructions corresponding to the loop execution of the layers. Overall, DianNao is used as a state-of-the-art accelerator for DNN applications and also a well known comparison point for other NN based accelerators. More details on the architecture, instruction set and their comparison with SIMD processors is in their paper [15].

B | Detailed Example for Stream-Dataflow Model

In Section 6.4.2, we had provided an overview of the programming and execution model of the stream-dataflow accelerator. Here, we provide a detailed execution model example with a more complex case of scratchpad streams being considered.

To give intuition into the nature of the concurrent execution potential of a stream-dataflow accelerator, we give a simple abstract example in Figure B.1, which shows an example of DFG, and the state of the stream commands, CGRA and the general core over time. The red arrows between the events show those that are plausibly on the critical path.

For explanatory purposes, we show streams in one of the four states:

1. *Enqueued* – stream-command is generated from the core and is enqueued;
2. *Dispatched* – stream is dispatched and allowed to execute in parallel;
3. *Source Resource in Use* – stream is actively using the source resource;
4. *Complete* – all stream's data has arrived at its destination (for data transfer commands);

To explain the example, the DFG is a simple two input multiply, with A and B as input ports and C as output port. The first stream-command (C1) reads some memory data into the scratchpad for later reuse. C2 is a barrier to prevent C3 from reading from the scratchpad until the data is written. Meanwhile C4 can proceed in parallel because it reads to port B from memory, but C6 (another memory read) has to wait until C4 is done reading from memory, because it also reads to port B. Once the C2 barrier has been reached, meaning scratchpad is done writing, C3 can be issued to read the data from scratchpad to port A. Note that, to show the significance of the data-reuse and repeated access pattern, we use two stream-iterations worth of data in the example, by streaming the stored data in scratchpad again to port A second time. Once some data arrives at both ports

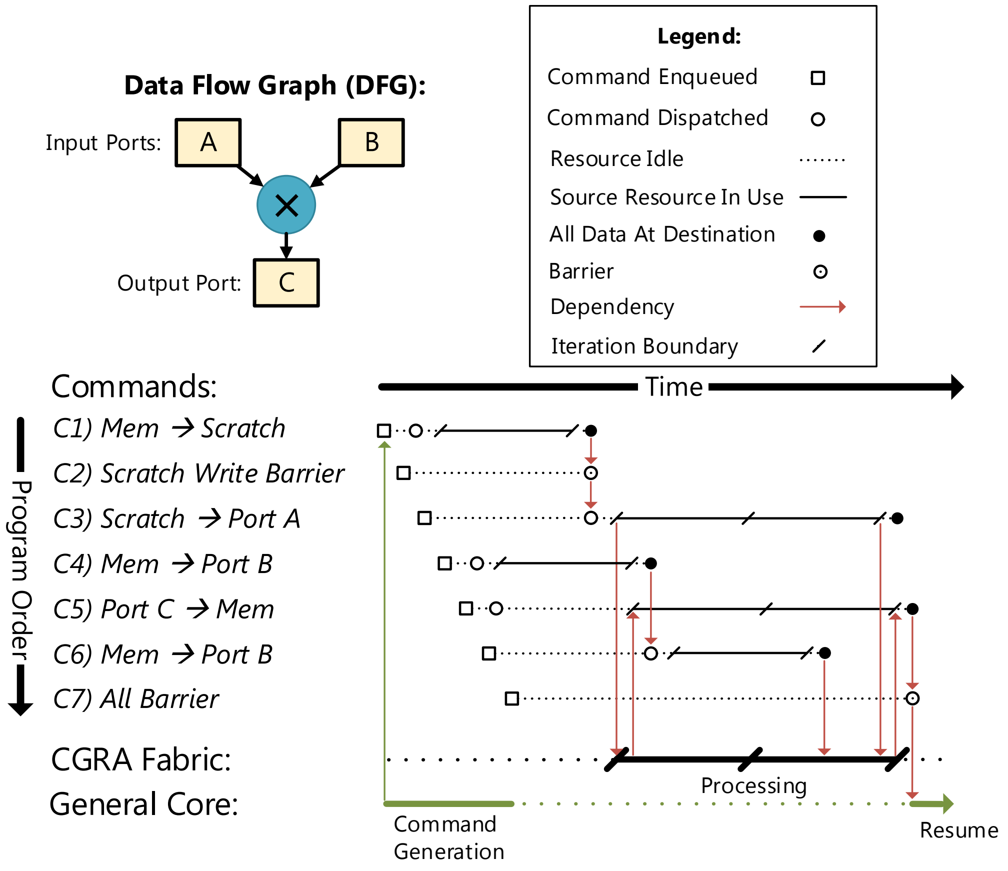


Figure B.1: Detailed Stream-Dataflow Execution Model Example

(C3, C4 in use), the CGRA can start computing the issued instances of the computation data. When the CGRA finishes some instances of the computation, the memory write command (C5) can begin sending write requests to port C. Because of two iterations of scratchpad stream and memory to port B stream, the data produced at port C is equal to total data drained from port A or port B. Finally, when all write requests are complete (C5), the barrier command (C7) can be completed, and the general core can resume generating next set of stream commands. Note that the scratchpad read, memory read, CGRA execution, and memory write can all proceed in parallel. Intuitively, as the execution phase is lengthened with a larger data-stream, or there is more overlap between commands, there will be less instruction/control overheads and more concurrent execution.

C | Rocket Core Micro-Architecture

Figure C.1 shows the detailed 5-stage datapath of the Rocket core [85] which implements the RV64G instruction set. Rocket core is a simple in-order five stage pipeline processor core which implements the RISCV RV64G ISA and is aimed at low-power embedded and IOT processing. We use rocket core's Rocket Custom Co-Processor (RoCC) accelerator interface to attach the stream-dataflow accelerator with minimal modifications. RoCC block which gets enabled at the write-back stage feeds the stream-dataflow accelerator with the commands and also has a stall interface when the accelerator cannot take in more commands.

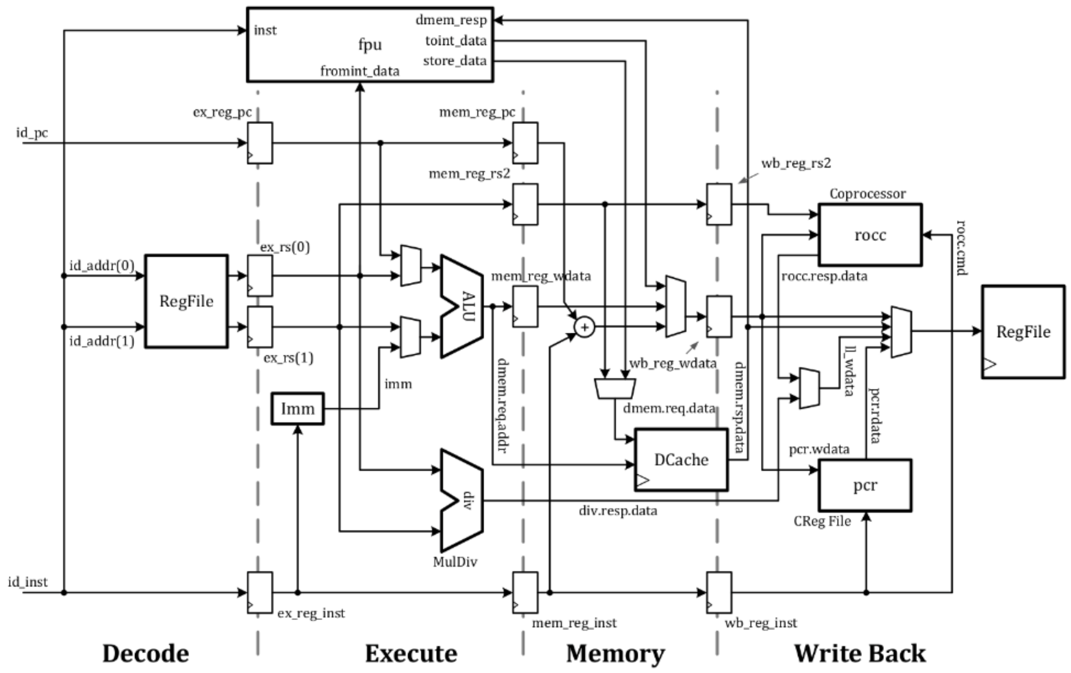
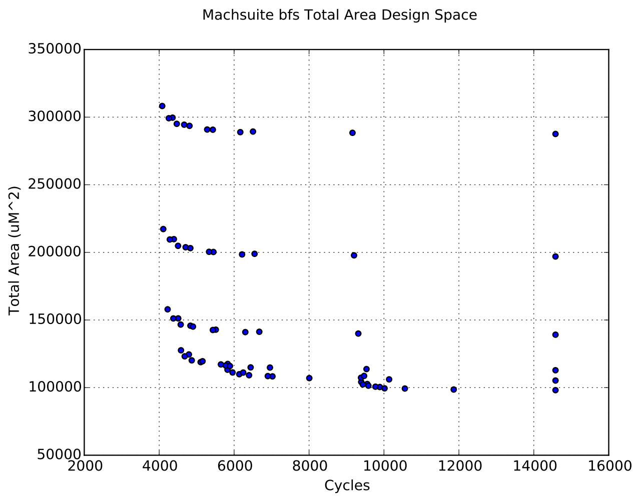


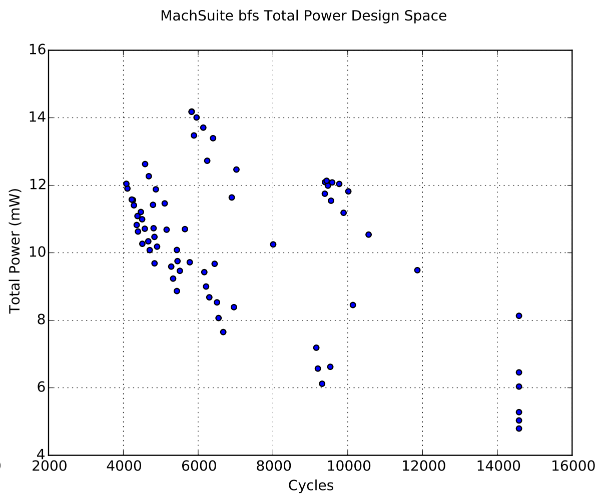
Figure C.1: Low-Power Core – RV64G ISA based 5-stage pipelined Rocket Core
(Figure reused from the lowRISC Rocket Core implementation [115])

D | Pareto-Optimal ASIC Design Point Selection

In Section 8.3.3, we had provided a brief overview of how we perform the ASIC design point selection for performance, power and area based application-specific comparison of Softbrain. Here, we provide a detailed example of ASIC design point selection with regards to *bfs* workload from the Machsuite application suite. As mentioned earlier, we perform the iso-performance analysis of the ASIC design versus application-provisioned Softbrain. We use the fixed-function accelerator specific design space exploration tool Aladdin [29] to generate the ASIC design points for each of the Machsuite applications. In this example, we chose *bfs* workload to show how we select the pareto-optimal design point for comparison.



(a) Total Cycles of bfs ASIC for Various Area Optimized Design Points



(b) Total Cycles of bfs ASIC for Various Power Optimized Design Points

Figure D.1: Design Space Exploration of ASIC Design for *bfs* Application

Figure D.1 shows the area and power optimal ASIC design points generated for bfs workload. In order to perform iso-performance analysis with Softbrain, we need to choose a design-point with execution cycles close to Softbrain, and for our evaluation we restricted this to be within 15% of softbrain cycles for a fair comparison. Now, after this decision, there could be ASIC design points both with regards to area and power which are way out of the range of softbrain's area and power for each of the kernel. This is because Aladdin tool does a sweep of the entire accelerator design space for various application parameters like loop unrolling/flattening, software pipelining and memory partitions. This generates design points with much larger area and power in order to reach the execution cycle count close to softbrain. This is seen in Figure D.1(a) and Figure D.1(b), where there are some design points exceeding 3mm^2 area and 14mW power for bfs ASIC. This leads to an unfair comparison putting softbrain in a better position compared to ASIC design points. So, for our evaluation, we restrict the area and power of the ASIC design points to not exceed softbrain's area and power, as custom hardware or ASIC's area and power cannot be larger than a much general design like softbrain. This sometimes results in less instantiations of problem specific FUs and memory partitions in ASICs, as there is a restriction on area and power of the design. With the limited power budget, the loop unrolling, loop flattening parameters cannot be enabled to have more FUs instantiated (and thus burn power) and hence the performance drop in case of fixed-function ASICs. Another way of performing the evaluation is to do an iso-area or iso-power comparison by restricting the ASIC's area and power close to softbrain and comparing the execution cycles.

Overall, we believe that the iso-performance analysis allows us to provision softbrain to match the performance of custom-hardware solutions and then analyze the overheads of power and area which can be attributed to generality.

Bibliography

[1] R. Miller, "The billion dollar datacenter." http://www.datacenterknowledge.com/archives/2013/04/29/the-billion-dollar-data-centers/, 2014, Online;.

[2] C. Metz, "Facebook catapults $1.5 billion datacenter in iowa." https://www.wired.com/2013/04/facebook-iowa-data-center/, 2013, Online;.

[3] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ISCA*, 2010.

[4] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 298–310, ACM, 2015.

[5] R. Colwell, "The chip design game at the end of Moore's law," Hot Chips, 2013.

[6] B. Sutherland, "No Moore? a golden rule of microchips appears to be coming to an end," The Economist, 2013. http://www.economist.com/news/21589080-golden-rule-microchips-appears-be-coming-end-no-moore.

[7] R. Courtland, "The end of the shrink," *Spectrum, IEEE*, vol. 50, pp. 26–29, November 2013.

[8] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 13–24, June 2014.

[9] N. Jouppi, "Google supercharges machine learning tasks with tpu custom chip," *Google Cloud Platform Blog*, 2016. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[11] N. Corp, "Accelerating deep learning inference using nvdla," 2017. http://nvdla.org/primer.html.

[12] M. H. Ionica and D. Gregg, "The movidius myriad architecture's potential for scientific computing," *IEEE Micro*, vol. 35, no. 1, pp. 6–14, 2015.

[13] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Computer Architecture Letters*, vol. 8, pp. 25–28, 2009.

[14] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ISCA '10*, pp. 451–460.

[15] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[16] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: A polyvalent machine learning accelerator," in *ASPLOS*, 2015.

[17] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[18] J. Brown, S. Woodward, B. Bass, and C. Johnson, "IBM Power Edge of Network Processor: A wire-speed system on a chip," *IEEE Micro*, 2011.

[19] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA*, 2006.

[20] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a programmable wire-speed regular-expression matching accelerator," in *MICRO*, 2012.

[21] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[22] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *MICRO*, 2013.

[23] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," in *ISCA*, 2013.

[24] R. Yazdani, A. Segura, J.-M. Arnau, and A. Gonzalez, "An ultra low-power hardware accelerator for automatic speech recognition," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[25] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon & the end of multicore scaling," ISCA, 2011.

[26] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[27] C. Edwards in *Scary dark silicon is here today*, 2009. http://blog.shrinkingviolence.com/2009/10/scary-dark-silicon-is-here-tod.html.

[28] D. Luebke and M. Harris, “General-purpose computation on graphics hardware,” in *Workshop, SIGGRAPH*, 2004.

[29] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, “Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures,” in *International Symposium on Computer Architecture (ISCA)*, IEEE, 2014.

[30] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, “Accelerator-rich architectures: Opportunities and progresses,” in *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6, ACM, 2014.

[31] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, “Pushing the limits of accelerator efficiency while retaining programmability,” in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 27–39, IEEE, 2016.

[32] T. Nowatzki, V. Gangadhar, G. Wright, and K. Sankaralingam, “Domain specialization is generally unnecessary for accelerators,” *IEEE Micro (Top Picks Issue)*, vol. 37, 2017.

[33] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *Proceedings of the 44th International Symposium on Computer Architecture ”'(ISCA)”'*, 2017.

[34] H. Bauer, J. Veira, and F. Weig, “Moore’s law: Repeal or renewal,” *New York: McKinsey & Company*, 2013.

[35] J. Cong, Z. Fang, M. Gill, and G. Reinman, “Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration,” in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pp. 380–387, IEEE, 2015.

[36] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, “Co-designing accelerators and soc interfaces using gem5-aladdin,” in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[37] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[38] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05*, pp. 190–200.

[39] T. Nowatzki, V. Govindaraju, and K. Sankaralingam, "A graph-based program representation for analyzing hardware specialization approaches," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 94–98, 2015.

[40] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *PLDI*, 2013.

[41] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 1216–1225, ACM, 2012.

[42] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[43] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *ISCA*, 2013.

[44] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *ASPLOS*, 2014.

[45] Arvind and R. S. Nikhil, "Executing a program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Comput.*, 1990.

[46] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 289–308, 1984.

[47] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg," in *PACT*, pp. 341–351, 2013.

[48] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A modular reconfigurable architecture.," in *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 161–171, June 2000.

[49] J. A. Fisher, P. Faraboschi, and G. Desoli, "Custom-fit processors: Letting applications define architectures," in *MICRO*, 1996.

[50] A. Hart, "The openacc programming model," *Cray Exascale Research Initiative Europe, Tech. Rep*, 2012.

[51] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[52] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying functionality and parallelism specialization for energy efficient computing," *IEEE Micro*, 2012.

[53] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *MICRO*, 2004.

[54] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[55] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for GPGPUs," in *ISCA*, 2014.

[56] "Xtensa LX3 customizable DPU, high performance with lexible I/Os," 2010. http://ip.cadence.com/uploads/pdf/LX3.pdf.

[57] J. Yiu, *The definitive guide to the ARM Cortex-M3*. Newnes, 2009.

[58] "Chi-Keung Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, PLDI, 2005, pp. 190-200.."

[59] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, 2011.

[60] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *Proceedings of the conference on Design, Automation and Test in Europe*, pp. 423–428, European Design and Automation Association, 2009.

[61] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[62] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO '09*.

[63] K. Khubaib, M. Suleman, M. Hashemi, C. Wilkerson, and Y. Patt, "Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP," in *MICRO*, 2012.

[64] Y. Watanabe, J. D. Davis, and D. A. Wood, "Widget: Wisconsin decoupled grid execution tiles," in *ISCA*, 2010.

[65] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, 2011.

[66] "Stratix 10 soc highest performance and most power efficient processing," 2015. https://www.altera.com/products/soc/portfolio/stratix-10-soc/overview.html.

[67] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "Xilinx 16nm ultrascale + mpsoc and fpga families," Hot Chips, 2015. http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.40-FPGAs-Epub/HC27.25.411-Zyng-Ahmad-Xilinx-v10.pdf.

[68] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 531–534, IEEE, 2012.

[69] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[70] F. Intel, "Sdk for opencl," *Programming Guide. UG-OCL002*, vol. 31, 2016.

[71] D. Singh, "Implementing fpga design with the opencl standard," *Altera whitepaper*, 2011.

[72] Transaction Processing Performance Council, "TPC-H benchmark specification," *Published at http://www. tcp. org/hspec. html*, 2008.

[73] Intel, "Core i7-3770k processor." http://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz.

[74] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS," in *ISSCC*, 2007.

[75] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended tcams," in *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, 2003.

[76] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on fpgas,"

[77] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *ACM SIGARCH Computer Architecture News*, 2011.

[78] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture.," in *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 52–63, June 2004.

[79] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO 36*.

[80] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a RAW machine," in *ASPLOS VIII*.

[81] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *ASPLOS*, 2006.

[82] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and t. T. Team, "Scaling to the end of silicon with edge architectures," *Computer*, vol. 37, pp. 44–55, July 2004.

[83] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 142–153, ACM, 2013.

[84] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, (Los Alamitos, CA, USA), pp. 112–119, IEEE Computer Society Press, 1982.

[85] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[86] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, 2000.

[87] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pp. 110–119, IEEE, 2014.

[88] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *ISPLED*, 2012.

[89] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *ISLPED*, 2013.

[90] D. H. Woo and H.-H. S. Lee, "Extending Amdahl's law for energy-efficient computing in the many-core era," *Computer*, 2008.

[91] A. Morad, T. Morad, Y. Leonid, R. Ginosar, and U. Weiser, "Generalized MultiAmdahl: Optimization of heterogeneous multi-accelerator soc," *Computer Architecture Letters*, 2014.

[92] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The accelerator store: A shared memory framework for accelerator-based systems," *ACM Trans. Archit. Code Optim.*, 2012.

[93] M. Lyons, G.-Y. Wei, and D. Brooks, "Shrink-fit: A framework for flexible accelerator sizing," *IEEE Comput. Archit. Lett.*, 2013.

[94] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvp trade;)," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 141–150, Dec 2003.

[95] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pp. 3–13, December 1998.

[96] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang, "Imagine: Media processing with streams," *IEEE Micro*, vol. 21, pp. 35–46, March/April 2001.

[97] G. Weisz and J. C. Hoe, "Coram++: Supporting data-structure-specific memory interfaces for fpga computing," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015.

[98] D. Koeplinger, C. Delimitrou, R. Prabhakar, C. Kozyrakis, Y. Zhang, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 115–127, IEEE Press, 2016.

[99] E. S. Chung, M. K. Papamichael, G. Weisz, and J. C. Hoe, "Cross-platform fpga accelerator development using coram and connect," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, (New York, NY, USA), pp. 3–4, ACM, 2013.

[100] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, (Washington, DC, USA), pp. 25–28, IEEE Computer Society, 2014.

[101] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 14–26, IEEE, 2016.

[102] Arvind and R. S. Nikhil, "Executing a program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, 1990.

[103] G. M. Papadopoulos, "Monsoon: an explicit token-store architecture," in *ISCA*, 1990.

[104] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, *et al.*, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 142–153, ACM, 2013.

[105] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 118–130, ACM, 2015.

[106] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural mechanisms to exploit value structure in simt architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[107] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, , and C. Batten, "Architectural specialization for inter-iteration loop dependence patterns," in *MICRO*, 2014.

[108] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *MICRO*, 2012.

[109] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability," in *MICRO*, 2012.

[110] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *ISCA*, 2015.

[111] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *ISCA '08*.

[112] M. A. Watkins, T. Nowatzki, and A. Carno, “Software transparent dynamic binary translation for coarse-grain reconfigurable architectures,” in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 138–150, IEEE, 2016.

[113] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA ’16, (Piscataway, NJ, USA), pp. 367–379, IEEE Press, 2016.

[114] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, “Cambricon: An instruction set architecture for neural networks,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405, June 2016.

[115] lowRISC Project, “Rocket core overview,” 2017. http://www.lowrisc.org/docs/untether-v0.2/rocket-core/.