LOGICAL ACCELERATORS ON MANYCORE PROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Philip Bedoukian

August 2023

LOGICAL ACCELERATORS ON MANYCORE PROCESSORS

Philip Bedoukian, Ph.D.

Cornell University 2023

Traditional general-purpose processors emphasize finding parallelism using hardware control logic, but this logic is energy inefficient and will limit future performance improvements. Manycore processors offer a contrasting approach for general-purpose computation: find parallelism in software and remove energy inefficient hardware control logic. Manycores offer up to thousands of programmable threads that can efficiently target applications with thread-level parallelism (TLP). However, manycores have not yet gained market adoption due to the difficulty of programming and lack of performance on applications containing data-level parallelism (DLP) and instruction-level parallelism (ILP).

This thesis proposes to use software reconfiguration to improve manycore performance on various classes of applications with little hardware overhead. We propose two designs where software can group together existing hardware resources of adjacent cores to form logical accelerators. Rockcress adds lightweight hardware to enable the formation of a logical vector unit to accelerate applications with DLP. Watercress adds lightweight hardware to enable the formation of a logical CGRA to accelerate applications with ILP. These techniques will expand the set of applications that can enjoy the performance and computational density of manycore architectures.

BIOGRAPHICAL SKETCH

Philip Bedoukian was born to Gail and Robert Bedoukian in Danbury, Connecticut (1995). His friends often joked that Phil should know random scientific facts because "Phil's Dad is a chemist". This teasing inspired him to actually learn the facts and pursue a degree in science and engineering.

Phil earned a bachelors of science in Physics at Lafayette College in Easton, Pennslyvania. While at Lafayette, he took a computer architecture class because there was an open spot in the Physics curriculum. It was an incredibly exciting course. Phil decided to pursue computer architecture after realizing the limited career prospects in Physics.

Phil did undergraduate research with Professor Matt Watkins on integrated CPU-GPU systems. Professor Watkins, a Cornell alumni himself, inspired Phil to pursue higher education and apply to Cornell. While working with Professor Watkins, Phil also worked on a computational Physics thesis using GPUs to simulate crystal growth with Professor Andrew Dougherty. These projects juxtaposed gave him insights on advanced computer systems from both the software and hardware perspectives.

Phil joined Cornell University in 2017 and was advised by Professor Adrian Sampson. Upon joining, he assisted Dr. Mark Buckler and Professor Adrian Sampson on a computer vision accelerator project. Phil switch focused from computer vision to manycore processors in his second year. He initially joined the HammerBlade research colloboration between Cornell and University of Washington. With insights from HammerBlade, he led development on new manycore architectures.

This document is dedicated to my family and friends.

ACKNOWLEDGEMENTS

I am grateful to all of my family and friends, and even to anyone who just said "Hi" to me during my graduate study. I am also thankful to the feedback my committee provided. Professor Adrian Sampson, thank you for being (for lack of a better word) chill. You made my transition to graduate school very easy. I fondly remember our first research meeting where you kept nodding profusely whenever I said something. Professor Chris Batten, I will always appreciate your critical feedback that encouraged me to be a better researcher. There we many times where I felt like an honorary member of your group. Professor Zhiru Zhang, thank you for your guidance on research and teaching undergraduate digital logic.

I am grateful to my undergraduate research advisors who prepared me for graduate school. Thank you to Professor Matt Watkins who inspired me to get into computer engineering. Thank you to Professor Andrew Dougherty who I did my first major research project within Physics. Thank you to my undergraduate friends Blake, Kevin and roommates Joe, Connor, and Andy.

I want to particularly thank Khalid Al-Hawaj, Tuan Ta, Neil Adit, Edwin Peguero, and Mark Buckler for both their academic collaboration and personal friendship. My Ph.D. would have taken at least one or two more years without your suggestions and technical help. Khalid thank you for all your life lessons. I'm not sure all of them were correct... but many changed my views on life. Tuan you were the first person I met in Ithaca and have enjoyed our continued friendship throughout the years. Thanks for all the car rides and don't be greedy with parking! Mark thank you for teaching me how to be a graduate student. Neil I enjoyed collaborating with you and always enjoyed your sense of humor. Edwin it was nice running into you randomly in Boston.

To the CAPRA research group thank you for collaboration and feedback. I

generally looked forward to our weekly meetings and status updates. Mark, Sachille, and Edwin it was fun brewing beer together.

Tristan Wang, thank you for being a great landlord, roommate, and friend. I am so glad I walked up to a random guy walking to the graduate student orientation. We went through a lot together, from being locked in together during the pandemic to Indian Meal Moth infestations to besting you by one second in a half marathon.

I am thankful for everyone who provided comradery during board and video game sessions especially during the COVID-19 pandemic. League of Legends: Jamie, Kai, Khash, Vlad, Blake. Magic: the Gathering: Tristan, Ritchie, Mark, Becky. Call of Duty: Steven, Jamie, Eric, Ken, Khalid. Halo: Jordan, Khalid. Overwatch: Lin, Derin, Zhuangzhuang, Peitian. Starcraft: Chenhui, Sungbo. Steven I enjoyed seeing movies with you and thank you for breaking me out of my shell post-pandemic.

Here are some miscellaneous acknowledgements. Yichi your A-Exam was an inspiration to me. Mulong you had the best catchphrase in CSL. Haron it was fun playing soccer with you for pickup and in the CSL intramural soccer team. Oscar, Yanqi, and Yu thanks for joining me for lunch and chatting early on in my Ph.D.. Mingyu, Zichao, Jonathan and Derin it was fun chatting with you and going out to lunch towards the end of my Ph.D..

There are a few places that had a profound impact on my graduate study and I will miss. Asia Cuisine you were the best restaurant I have ever eaten at in my life. The sweaty walk up to school up on the Buffalo Street hill. The beautiful corridor along Cascadilla Creek on my walk home in the evening. The Black Diamond Trail of which I have run more than 500 times.

Finally, I would like to thank my family who supported me on my path to and through graduate school. To my parents, thank you for the sacrifices you have made

for me from therapy to driving me to school and soccer practice. Mom thank you for teaching me how to stand up for myself. I'll never admit it to you in person, but I enjoy spending time with you over yard work. Dad thanks for being a role model for me in science and engineering. I loved watching Star Trek: Next Generation and football with you. Dave and Matt I always felt so lucky to have two cool older brothers. I think you have gotten slightly less cool after having children, but I'm glad your kids get to have cool Dads. Dave thank you for playing games with me and telling me jokes when I was younger; I still use your jokes on my friends to this day. Matt thank you for inspiring me to keep fit. I enjoy running with you and hope one day I can beat you in a 5K. To my nephews, thank you for reminding me of the pure joy that can be found in life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The demand for massively multithreaded computation is ever-increasing. Recent large-language models (LLMs) require around 1 million GPU hours to train [50] and LLM-based web search requires 10× more computation than conventional search [48]. Scientific computing demands higher fidelity and more computationally expensive simulations to improve our understanding of nuclear fusion devices, material science, and cosmology [70].

Petascale and Exacale computing have emerged to satisfy the computational demands of these applications. Frontier [71], the first exascale supercomputer, can process at a staggering rate of 1.1 exa-FLOPs with 46,400 processor nodes (4 GPUs per 1 CPU) and 21MW power consumption. The design of the many individual processor nodes is crucial to continue scaling performance (throughput) and reduce the operation costs. However, designers cannot just add more performance per processor without considering energy efficiency. Modern CPUs and GPUs already employ a variety of energy inefficient hardware structures that have become a energy bottleneck.

Reducing hardware complexity has re-emerged as a potential solution to improve energy efficiency while maintaining performance [99]. The key idea is to shift more responsibility of performance from expensive hardware structures onto the software stack. Manycore processors are a viable candidate for this shift. Manycores comprise 100s or 1000s of low throughput, but area and energy efficient cores. To achieve high performance, the programmer and software must split the application into many threads and map these threads onto the abundance of cores.

Table 1.1: Where is Parallelism Discovered.

Location	**Performance**	**Energy Efficiency**	**Drawbacks**
Design	High	High	Inflexible
Software	Low-Medium	Medium	Programming-Effort
Hardware	Medium	Low	N/A

While manycores can effectively execute programs with thread-level parallelism (TLP), they are less effective at executing programs containing data-level parallelism (DLP) and instruction-level parallelism (ILP). This thesis explores software reconfiguration of existing manycore hardware structures to accelerate programs containing DLP and ILP with little area and energy overhead. We propose to form logical DLP and ILP accelerators using the pre-existing functional units and local memories within groups of adjacent cores. When there is DLP or ILP available to exploit in a program, logical accelerators can be formed to improve performance and energy efficiency. When no DLP or ILP is available, the proposed system acts as a standard manycore.

1.1 Finding Parallelism

A computer system improves performance by extracting and exploiting parallelism in the program being executed. There are three places to extract parallelism: in the design process, software, or hardware. Each method has benefits and drawbacks, but the choice is key to energy efficiency. Table 1.1 broadly enumerates the properties of the methods.

Hardware-extracted parallelism uses control logic to discover ways to schedule operations at runtime. However, this hardware control logic consumes a significant portion of the total system energy for both CPUs [42, 73] and GPUs [58, 64].

Hardware-extracted parallelism is considered energy inefficient because only a small portion of energy goes to actual computation.

Design-extracted parallelism requires chip designers to bake-in knowledge of an application (i.e., datapath) into hardware to create application-specific accelerators (ASICs). This approach eliminates energy inefficient runtime control logic at the cost of flexibility. ASICs are not a viable option for supercomputers as a wide variety of fields will want to use them. Only one of the top 10 strongest supercomputers uses an ASIC that is not a CPU or GPU [102].

Software-extracted parallelism uses a combination of programming languages and the compiler to extract parallelism in programs. Software-extracted parallelism does not suffer from the inflexibility of ASICs and is more energy efficient than hardware control logic: a multiplexer is cheaper than a complicated instruction scheduler. However, this approach places a greater requirement on the programmer to come up with performant code and creates adoption challenges.

1.2 Types of Parallelism

Applications contain different forms of parallelism that can be extracted to improve performance. Thread-level parallelism (TLP) finds groups of instructions that can execute independently across threads. Data-level parallelism (DLP) finds instructions that independently execute the same operation. Instruction-level parallelism (ILP) finds instructions that can execute independently within same thread. Pipeline parallelism (PLP) finds operations that can occur in sequence, but are independent from previous iterations of the loop. This dissertation will mainly focus on architectures with TLP, DLP, and ILP.

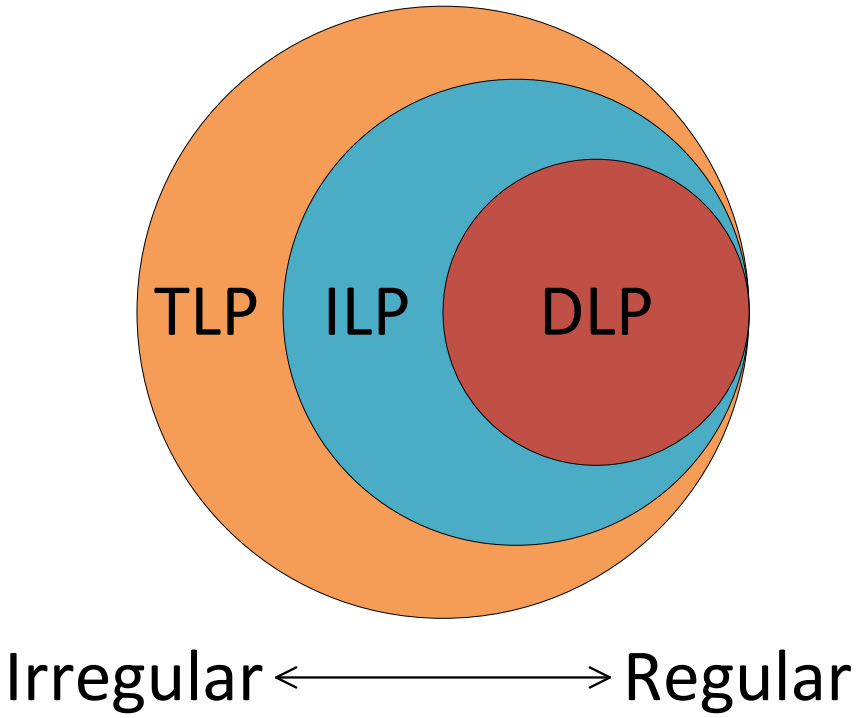


Figure 1.1: Types of workloads that TLP, ILP, and DLP can effectively target.

Multithreaded workloads can contain TLP, ILP, and/or DLP. DLP exists in a subset of workloads that contain ILP and ILP exists in a subset of workloads that contain TLP. Figure 1.1 gives an abstract representation of the types of parallelism. TLP is the most general purpose and can target all multithreaded workloads. Some multithreaded workloads are very irregular and have very little ILP and DLP where few operations can occur independently within a thread. Other workloads have irregularity like data-dependent memory access and control flow that limit DLP, but can still benefit from ILP. This dissertation will focus on improving the performance of regular workloads where TLP, DLP, and ILP intersect. While regular workloads can benefit somewhat from each type of parallelism, there will be performance and efficiency differences.

Table 1.2 enumerates where various prior works extract these different types of parallelism. Most architectures include multiple forms of parallelism and realize them using a combination of design, software, and hardware optimizations. Multicores and manycores typically use a combination of techniques whereas CGRA-style and vector architectures rely mostly on software. Software-extracted parallelism has a large presence in academic architectures, but plays less of a role in commercial architectures where multicores and GPUs dominate.

1.3 Modern Software-Extracted Parallelism: Languages, Co-design, and Necessity

Traditionally, computer systems relying on software-extracted parallelism failed to achieve wide-spread adoption [78, 89, 104]. However, we believe there are three major factors that will lead future computers systems towards software: Languages, Co-design, and Necessity.

1.3.1 Necessity

Figure 1.2 shows performance and energy trends as the transistors per chip increases due to smaller transistor sizes. Single-threaded performance has stagnated while multi-threaded performance continues to improve. Multi-threaded performance improvement is driven by multicore CPUs (at least for embarrassingly parallel benchmarks like SpecRate) and GPUs [79] (performance not shown in plot). However, multi-threaded performance scaling is not expected to continue due to reduced transistor efficiency and lack of exploitable TLP in real programs [29].

Table 1.2: Taxonomy of Parallelism and Reconfigurability. Thread-Level (TLP), Instruction-Level (ILP), Data-Level (DLP), Pipeline (PLP) parallelism extracted in Design (D), Software (S), or Hardware (H). Architectures may have multiple ways to extract parallelism, but the most significant one is shown. * means significant hardware is required on top of the base architecture to realize the parallelism. We distinguish ILP from PLP as PLP does not improve peak ILP, but rather can spread computation over multiple cores if no TLP is available.

Hardware	**Base Architecture**	**TLP**	**ILP**	**DLP**	**PLP**
Single core	Single core	-	H	S	-
Multicore	Multicore	S	H	S	-
Core Fusion [45]	Multicore	S	S	-	-
DySER [35]	Multicore	S	S	-	-
big.VLITTLE [95]	Multicore	S	H	S	-
MAVEN [62]	Multicore	S	-	S	-
Itanium [104]	Multicore	S	S	-	-
Swarm [47]	Multicore	H	-	-	-
Celerity [25]	Manycore	S	D	-	S
Kilocore [13]	Manycore	S	-	-	-
TileGx100 [78]	Manycore	S	S	-	S
RAW [100]	Manycore	S	-	-	S
Loki [5]	Manycore	S	-	S	S
Versa [57]	Manycore	S	-	-	S
Xeon Phi [89]	Manycore	S	H	S*	-
Stitch [96]	Manycore	S	D	-	S
Transmuter [74]	Manycore	S	-	-	S
GPU	Manycore	S	-	H	-
Variable Warp Size GPU [81]	Manycore	S	-	H	-
GPU w/ integrated CPU	Manycore	S	H*	H	S*
Rockcress (Chapter 2)	Manycore	S	-	S	S
Watercress (Chapter 3)	Manycore	S	S	-	S
TPU [51]	CGRA	-	D	-	-
MorphoSys [87]	CGRA	-	S	-	-
ADRES [66]	CGRA	-	S	-	-
TRIPS [84]	CGRA	S	S	S	-
Plasticine [76]	CGRA	-	S	S	-
MT-CGRA [106]	CGRA	S	S	-	-
WaveScalar [94]	CGRA	S	S	-	-
FPGA	LUTs	S	S	S	S
Blackwidow [1]/Tarantula [30]	Vector Machine	-	-	S	-
Cray X1 [28]	Vector Machine	-	-	S	-
Libra [75]	Vector Machine	-	S	S	-

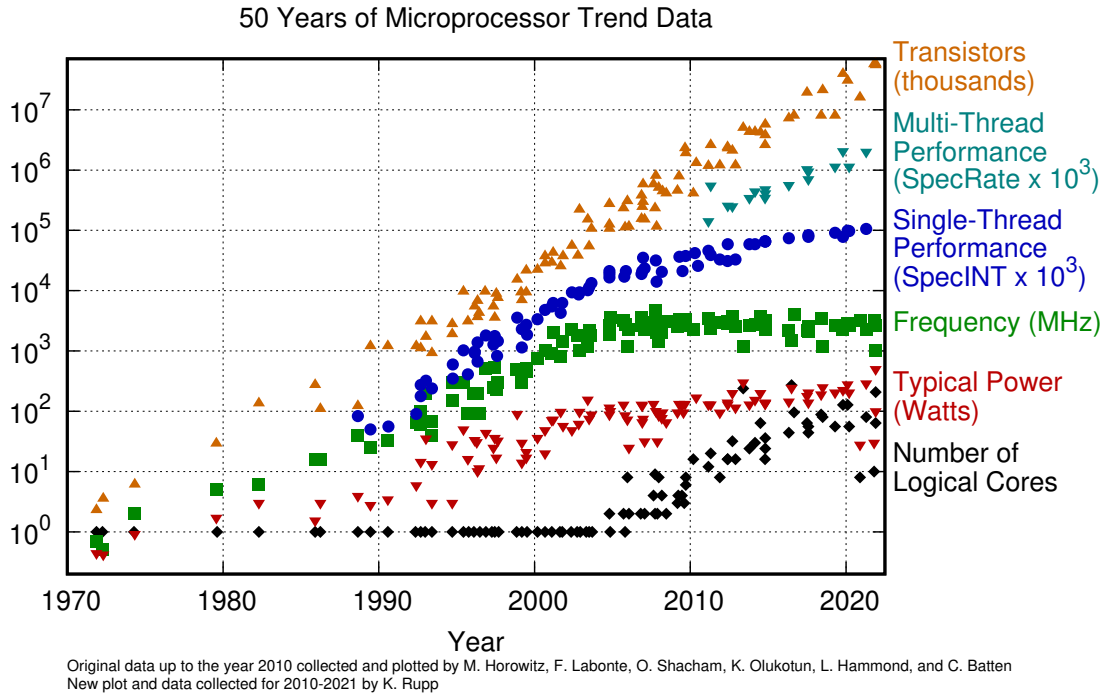


Figure 1.2: Processor Trends [53]. SpecRate data collected by the thesis author using a subset of the original processors.

The success of multicore CPUs and GPUs is an early success story of software-extracted parallelism. Programmers now have meaningful ways to directly exploit parallelism in the form of TLP rather than just rely on hardware-extracted parallelism for ILP. However, this approach has also come with challenges: parallel programming is infamously hard.

In a naive system, there is an inverse relationship between programmer effort and hardware complexity to maintain high performance. Simpler and more energy efficient hardware requires more programmer effort, while complex hardware requires little programmer effort but suffers from inefficient hardware operations. Unfortunately, hardware must become more efficient to sustain the computational demand through and beyond the exascale era. This means software stacks must become more complex and potentially lead to increased programmer effort. However, unique solutions such as domain-specific languages (DSLs) and software-hardware co-design have emerged to reduce programmer effort while producing complex software to manage simpler hardware.

1.3.2 Languages

Compiler techniques such as auto-parallelization have not improved enough to eliminate the need for programmer managed parallelization [68]. However, new languages have emerged to reduce programming effort.

Domain-specific languages (DSLs) apply broad application knowledge into the programming language design to improve programming productivity. Halide [77] decouples the algorithm from optimization techniques for image computations. GraphIt [16] enables efficient specification of parallel graph algorithms. T2S [91]

provides languages features to improve productivity in dataflow computing for tensor applications.

Other languages seek to improve the productivity for specific forms of parallelism. Go Language [80] provides high-level concurrency features such as thread communication interfaces. OpenCL [54] enables efficient management of heterogeneous architectures.

1.3.3 Co-Design

Hardware-Software Co-Design seeks to further improve programmer productivity by widening the hardware-software interface and giving languages and compilers more features to exploit. CUDA [69] targets GPUs and is constantly evolving to support new hardware features. Swarm [47] leverages thread-level speculation (TLS) to have the hardware find a large chunk of TLP rather than the programmer. Elastic CGRAs [43] reduce compiler scheduling constraints by making PE to PE communication latency insensitive. Performance counter driven compilation [21] can leverage hardware run-time information to dynamically improve software compilation. Task-based hardware [24] allows hardware management for load balancing of small threads.

1.4 Manycore Processors

The potential of software-extracted parallelism has reignited interest in low complexity hardware architectures [99]. We believe manycore processors are the most viable low complexity architecture due to their programmability. This section

explores the design and challenges of manycores.

Manycore processors seek to maximize the number of threads exposed to software. High-performance manycores contain 100s [20, 25, 78] or 1000s [13, 27, 72, 112] of small in-order cores. These cores are smaller and have less throughput than the cores comprising a multicore CPU. The total system throughput is made up for by having lots of these low throughput cores: TLP is maximized even if ILP must be sacrificed. Manycores must be designed around scalability due to the number of cores. For example, maintaining cache coherence is challenging with 100s to 1000s of cores [108]. The on-chip data network is typically a mesh network so the wire latency does not increase with the number of nodes in the network.

Manycore processors are tiled to improve physical and logical design. Each tile contains one or more cores (typically no more than 8), an I-Cache, and a local data memory. This data memory is usually a scratchpad [13, 20, 25, 27, 72, 112], which requires programmer-managed data movement and caching.

The core pipeline design in a manycore is typically a single-issue in-order pipeline [13, 20, 25, 112]. Superscalar cores are antithetical to manycore design principles due to their area and energy overhead. Additional per-core area and energy will lower the number of cores that can be fit and operated on the manycore, which will reduce TLP. Superscalar pipelines require additional functional units, control logic, and local memory ports which are expected to have linear area overhead. In addition, superscalar pipelines often require out-of-order (O3) scheduling and multithreading to effectively utilize the additional issue slots, which consumes additional energy and area [73].

Manycores have excellent general-purpose programmability. Each core can

manage their own control flow and can tolerate variable latency operations. While manycores can support most of a RISC ISA, they are intended to be a parallel accelerator rather than a processor that runs an operating system (similar to a GPU).

Manycores seem like they should be a viable commercial architecture due to their programmability, high performance, and energy efficiency. However, high performance manycore architectures have failed to find commercial success. The Intel Xeon Phi [89] and startup Tilera [10, 78] never found market success.

There are two main reasons prior manycore processors have not yet succeeded. First, there are additional programming challenges that do not exist or are less severe in conventional CPUs and GPUs. The second reason is the low performance on applications containing regular parallelism (predictable data accesses and control patterns) compared to other throughput-oriented processors like GPUs. Manycores naturally target irregular parallelism with little locality and redundancy in the data and instruction streams.

The programming challenge has been improved by prior work, but remains a challenge. However, this dissertation focuses on improving the performance of applications with regular parallelism.

1.4.1 Programming Challenges

Compiler and programmer effort is important for instruction scheduling because the cores are in-order. The lack of local caches and low issue width reduces the number of dynamic stalls and allows for more accurate software instruction scheduling. Software-based DMA [23] can reduce the load-use latency to global memory in

manycores. Scratchpad management can also be alleviated somewhat with compiler techniques [65].

The main manycore programming challenge is managing the massive number of threads and effectively mapping them to cores. Single-Program Multi-data (SPMD) programming models are an effective approach to manage this feature [23]. Each core or thread runs the exact same program but distinguishes themselves with a unique thread id. This model is effective when each core will do about the same operations with few thread-specific control paths (otherwise there will be code size bloat). It is unlikely any application would require vastly different operations on each of the 100s or 1000s of cores.

While SPMD can increase productivity in embarrassingly parallel applications, applications with fine-grained synchronization and inherent lack of TLP create challenges. This dissertation does not focus on this challenge.

1.4.2 Performance Challenges

While manycore programming is similar in complexity to GPU programming, there remain issues with the types of applications that a manycore can accelerate. Manycores target workloads with irregular parallelism i.e., have little memory or control regularity. This means that cores are fetching data from different places and executing different parts of the program. For these workloads, it makes sense that each set of functional units should have its own control logic and memory. However, this is an expensive ratio and limits the maximum speedup that can be achieved per area. GPUs, on the other hand, focus on exploiting regular parallelism that exists in the form of DLP or ILP (predictable control flow, memory access,

and instruction schedules). GPUs exploit the fact that each thread will run similar operations and can reduce the control logic to functional unit ratio. GPUs can also coalesce memory requests to similar addresses together. While these features are inefficient for irregular parallelism, it leads to massive efficiency on regular parallelism.

Manycores lack fundamental ways to accelerate ILP and DLP workloads. An obvious solution is to add ILP and DLP accelerators to each manycore tile. For example, a designer could add out-of-order superscalar pipeline structures and SIMD units. However, this breaks the main tenet of manycore design: maximize the number of threads. While transistors and area are cheap in modern chips, per-tile area in a manycore is precious. It is important to minimize wire delay in a tile [100] and larger tiles reduce the total number per chip.

1.5 Reconfigurable Hardware

While software-extracted parallelism can alleviate hardware overhead, conventional hardware limits how the extracted parallelism can be *expressed* to hardware. For example, manycore processors cannot exploit DLP and ILP natively due to hardware constraints. Even if software knows there is DLP or ILP to be exploited it cannot express this to the manycore hardware. Therefore, successful software-driven computer systems must both extract parallelism in software and be able to effectively express it to simple-yet-flexible hardware. Software reconfiguration of manycore hardware can allow DLP and ILP tailored execution with little area overhead. The idea is to use intelligently placed and managed multiplexers rather than large hardware structures to target different forms of parallelism.

Previous work has explored capturing additional forms of parallelism with reconfiguration. TRIPS [84] enables TLP, ILP, DLP on the same CGRA fabric. Core Fusion [45] enables TLP and ILP reconfiguration by merging or splitting out-of-order CPU pipelines. big.VLITTLE [95] enables TLP and DLP reconfiguration by having one big core execute a vector instruction on multiple little cores.

While these architectures can exploit various forms of parallelism on the same fabric, the performance on these forms depends on the baseline architecture (see Table 1.2). Forms of parallelism that require little reconfiguration will have the highest performance per area, while forms that require considerable reconfiguration will have lower performance per area. For example, a manycore is better suited to TLP execution than to ILP execution. CGRA-based architectures like TRIPS are best suited to ILP execution and are limited in the amount of TLP that can be expressed. Vector Machines are best suited for DLP execution.

TLP is the most important target for a baseline architecture because it requires significant hardware support to be exploited well by software. TLP needs many instruction fetch resources, random memory access efficiency, and efficient synchronization mechanisms to be performant. In contrast, ILP and DLP are more regular forms of parallelism and require minimal hardware for software to effectively exploit them (i.e., CGRAs or vector machines). We posit that a manycore is the best reconfiguration target because the architecture has the most ability to exploit TLP.

1.6 Thesis Overview

Software-extracted parallelism is the future of computer systems. However, software must not only extract parallelism, but also be able to express how to exploit

parallelism to the underlying hardware. Mainstream parallel processors cannot support this feature due to hardware inflexibility and must be redesigned. Manycores are a viable candidate for this transition, but lack efficient methods to improve performance on programs with regular ILP and DLP parallelism.

This thesis proposes two reconfigurable designs that build upon a conventional RISC manycore processor to improve ILP and DLP capabilities with little area and energy overhead. The key idea in both proposals is to share resources (mainly functional units and local memories) between multiple cores in the manycore to realize logical accelerators for different types of code. We add direct links between each neighboring core in the X and Y dimensions to facilitate communication between the per-core resources. Chapter 2 describes Rockcress, a manycore that can exploit DLP using vector execution. We allow adjacent cores to forward instructions between one another and fetch data together to reduce redundant control and memory operations. Chapter 3 describes Watercress, a manycore that can exploit ILP using CGRA-style execution. We allow the functional units in each core to act as a local CGRA and to group together with functional units in adjacent cores to form large logical CGRAs.

Chapters 2 and 3 detail the hardware modifications implemented to enable reconfiguration as well as the software stacks to manage the new hardware. The proposed architectures are modeled using the gem5 cycle-level simulator [12]. We evaluate the models using C applications with various software reconfiguration techniques.

1.7 Collaboration, Other Work, and Funding

The work presented in Chapters 2 and 3 was led by myself with various collaborators. Neil Adit, Edwin Peguero, Tuan Ta, Khalid Al-Hawaj contributed to Chapter 2. Neil contributed to kernel and compiler development. Edwin led compiler development. Tuan provided a baseline manycore cycle-level model including a custom in-order core and scratchpad. Khalid provided RISC-V vector extensions for the evaluated model. I designed the architecture and contributed to kernel development. This work was presented at MICRO 2021 [9].

Chapter 3 builds upon insights, benchmarks, and models developed in collaboration with the people mentioned above. I designed the architecture, developed the kernels, and modified an open-source CGRA compiler. This work is currently in submission.

I previously worked on EVA^2 [18] led by Dr. Mark Buckler. The work used hardware-software co-design to create an improved computer vision accelerator using temporal redundancy in consecutive video frames. I led RTL hardware development on the temporal redundancy components of the proposed accelerator.

This thesis describes research supported by NSF awards #1845952 and #1723715. The research is also partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7863. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the

authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This research was also supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

CHAPTER 2

SOFTWARE-DEFINED VECTOR PROCESSING ON MANYCORE FABRICS

This chapter explores adding DLP execution capabilities to a manycore processor. We map techniques found in traditional vector architectures onto the execution resources of multiple adjacent cores in a manycore. A logical vector unit will be formed from this mapping.

2.1 Background: Vector Architectures

Vector machines [1,30,62,63,83] exploit SIMD parallelism (DLP) to amortize control and data overheads in data-parallel workloads. There are multiple styles to realize vector (SIMD) execution including traditional SIMD, SIMT, and vector-thread. However, all of these styles implement some form of control amortization, wide memory access, and bulk memory latency hiding.

Traditional SIMD is the most efficient vector style. A single controller issues the same commands to multiple functional units to execute on independent pieces of data. The compiler encodes parallelism within a single instruction and removes this responsibility from the hardware. Traditional SIMD execution can be integrated with the core [44] or decoupled [1, 30].

Single-instruction multiple-thread (SIMT) encodes parallelism in instructions, but only as a hint to the hardware. The hardware can still split up the individual elements if there is control flow divergence. This requires less hardware than a CPU, but more than a traditional SIMD engine.

Vector-thread architectures [59, 62] allow a single core to direct the control flow of other fully-fledged cores. The cores can be instructed to run the same operations or direct their own control flow. This approach does not save area like a traditional SIMD engine would because the control logic still exists. However, it does save a similar amount of energy because this control logic is not used when the leader core can coalesce redundant control flow.

2.2 Introduction

Manycore processors target workloads containing irregular MIMD parallelism (TLP) while vector machines target workloads containing regular SIMD parallelism (DLP). However, workloads are neither perfectly regular nor entirely irregular—computations exist on a spectrum of regularity. Successful parallel machines therefore tend to combine aspects of both SIMD and MIMD parallelism. GPGPUs augment vector engines with sophisticated runtime monitoring to cope with irregularity, and Larrabee and its descendants [85, 107] augment manycore tiles with small SIMD functional units. These standard approaches, however, "bake in" a specific point in the trade-off space between regular and irregular parallelism and suffer low utilization for other points in the spectrum.

We propose a manycore architecture that can flexibly adopt the efficiency advantages of a vector machine. The architecture supports run-time configuration to group together small, independent computation tiles to form large, SIMD-optimized vector engines. We introduce *software-defined vectors:* an architectural abstraction that lets a single manycore fabric operate as completely independent processors, large aggregate vector engines, and as flexible combinations of these

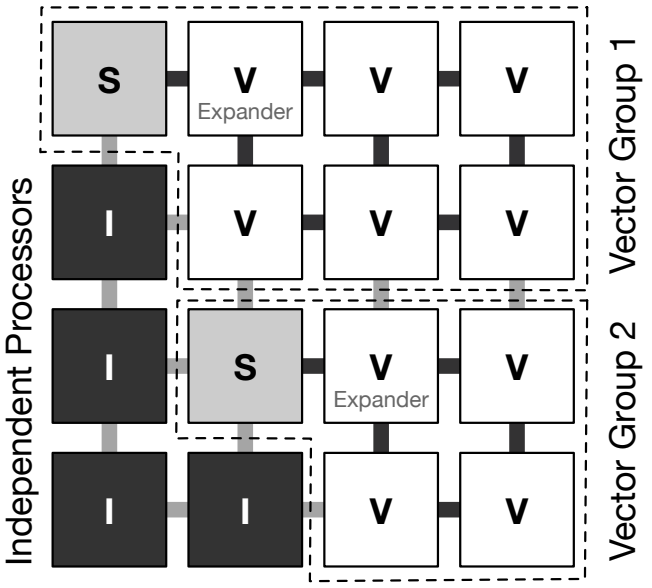


Figure 2.1: A manycore fabric with two software-defined vector groups. Cores can be in one of three modes: *scalar* cores lead vector groups, *vector* cores follow scalar cores, and *independent* cores operate in a plain manycore style. Cores can change which mode they are in during run time.

two extremes.

Figure 2.1 gives the software view of a 4×4 tiled machine with software-defined vectors. The software can coalesce tiles into *vector groups* that execute in lockstep and disable the processors' front-ends, including their instruction caches. A vector group consists of one *scalar core*, which runs scalar computations and issues memory accesses for the group, and several *vector cores* that share a single SIMD instruction stream. A vector group emulates a classic vector-thread architecture [59]: the scalar core controls execution by launching data-parallel microthreads that run on the vector cores. Tiles that are not part of a vector group execute in an *independent* mode where they continue to behave as in a typical manycore.

Vector machines offer three main efficiency advantages: compute density, coalesced wide memory accesses, and control cost amortization. Compared to standard per-core hardware SIMD units, software-defined vector groups sacrifice sheer com-

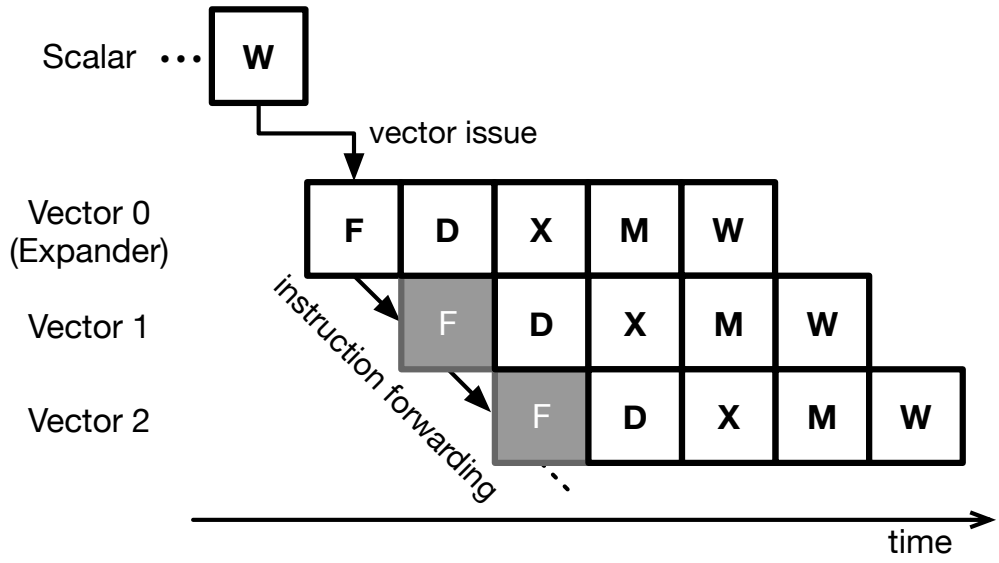


Figure 2.2: Execution in a Rockcress vector group using a classic RISC pipeline. The scalar core starts a microthread with a *vector issue* instruction, which tells the first vector core (the *expander*) to begin fetching instructions. The expander core fetches an instruction from its instruction cache and sends it to the second vector core over the *instruction forwarding network* (inet). Subsequent vector cores disable their fetch logic and accept instructions from the inet instead.

pute density but more flexibly amortize memory and control overheads. The key research challenge in this chapter is realizing these benefits while reusing the distributed computational resources in the manycore fabric. The control coupling within a vector group allows the scalar core to centralize regular, wide memory requests on the vector cores' behalf. Software-defined vectors also naturally support a configurable vector length: applications can choose an ideal hardware vector length according to their regularity. They naturally compose with per-core SIMD by grouping small, fixed-size vector units into larger logical vector engines.

This chapter instantiates the software-defined vector abstraction in Rockcress, a processor consisting of minimal modifications on top of an existing RISC-V manycore processor. The key addition in Rockcress is an *instruction forwarding network* (inet): vector cores pass each instruction they execute directly to their

neighbors in the vector group. One core fetches an instruction and sends it to its immediate neighbors, those cores then begin executing the instruction and forward it to their neighbors, and so on. Figure 2.2 shows how Rockcress uses the inet to forward instructions from the first vector core in a group, called the *expander* core, to the other vector cores. Only the expander needs a full fetch stage: the remaining cores disable their instruction cache and fetch logic, instead accepting instructions from the inet.

To keep vector groups busy, Rockcress relies on the independently controlled scalar core to perform shared computations and issue memory requests on behalf of the group. We augment the manycore's scratchpads with cheap bookkeeping to support a decoupled access/execute [88] arrangement.

Contributions The contributions in this chapter encompass the design of a hardware–software abstraction for software-defined vectors (Section 2.3) and a hardware implementation that augments a simple manycore processor with instruction forwarding to support the abstraction (Section 2.4). We demonstrate a C-based programming model and compiler that targets the augmented manycore machine to support both MIMD and SIMD parallelism (Section 2.5).

Relative to manycore execution, Rockcress vector groups offer these efficiency advantages: (1) Performance and energy savings by aggregating word-sized loads into wide vector loads. (2) Energy savings from disabling vector cores' frontends and instruction caches. (3) Performance improvement from overlapping scalar computation and data accesses with vector computation in a decoupled access/execute arrangement. The scheme's overheads consist of the cost to set up vector groups and invoke microthreads. Our evaluation (Section 2.7) uses cycle-level modeling to

investigate whether the benefits consistently outweigh the costs.

On the 15 benchmarks in the PolyBench/GPU suite [36], we find that software-defined vectors improve performance by an average of 1.7× over optimized manycore baselines that exploit memory-level parallelism (MLP). The scheme also reduces the total dynamic energy by 22%. We also compare against a GPU model configured to match the memory and execution resources of the manycore. Rockcress outperforms the GPU by an average of 1.9×.

2.3 Software-Defined Vectors

We describe the software-defined vector ISA from the programmer's perspective. The goal is to let software dynamically form flexibly-sized vector units by reserving portions of a standard manycore machine. The ISA abstracts the concrete hardware implementation in Section 2.4. There are three main components: forming vector groups (Section 2.3.1), using them to run microthreads of vectorized computation (2.3.2), and feeding them with vectorized memory accesses (2.3.3). We describe the ISA as an extension to RISC-V [109], but the principle applies to any RISC-like machine.

2.3.1 Vector Groups

The first component of the software-defined vector ISA is the instructions for creating *vector groups*. A vector group is a contiguous region of tiles in the manycore fabric that are logically coupled so that they can run SIMD computations.

Forming a vector group The programmer can create a vector group from a rectangular region of tiles by instructing each constituent core to enter vector mode. Cores compute a bitmask that describes the vector group configuration, the instruction forwarding path, frontend configuration (e.g., whether the I-cache is enabled), and the group coordinate and size. The scalar core and the first vector core in the group, called the *expander* core, leave their frontends enabled; the other vector cores disable their I-cache. To enter vector mode, cores write the bitmask to a special control/status register (CSR), `vconfig`.

When adjacent cores write to the `vconfig` CSR, a vector group is formed. Each core determines a thread id to distinguish itself from the other cores in the group. The cores then wait for a signal on the inet indicating that their neighboring cores have also entered vector mode and then begin forwarding instructions. The latency to form a group is similar to that of a software barrier; all of the cores in the future group must reach the same configuration instruction.

Disbanding a vector group Cores need to exit vector mode to return to MIMD execution or to participate in global synchronization. Since cores in vector mode do not maintain a program counter, they need to receive an up-to-date PC and control signal to resume normal execution. The scalar core indicates that the vector group should disband by sending a special `devec` instruction along with a valid PC. When a core receives this signal, the core forwards it to adjacent cores, resets the `vconfig` CSR, and begins normal execution from the given PC.

2.3.2 Vectorized Computation

Vector groups support a vector-thread execution model [59]. The scalar core uses the `vissue` instruction to launch microthreads on the vector cores:

`vissue` *imm*

The `vissue` instruction includes an instruction pointer indicating the beginning of the microthread. On commit, the scalar core sends the starting PC to the vector group via the inet. Vector threads are *asynchronous:* the `vissue` instruction retires immediately, and the scalar core executes concurrently while the microthread runs.

The expander core is responsible for fetching the instructions in the microthread. It fetches and executes instructions itself, but it also forwards them via the inet to the other cores in the vector group. During microthread execution, all cores in the vector group share a single instruction stream, so control divergence is not possible. Only jumps to consistent addresses (i.e., function calls) are allowed within microthreads. If the application needs divergent control flow, it must disband the vector group and return to MIMD execution.

The microthread ends when the expander core encounters a thread termination instruction `vend`.

Instruction forwarding not only amortizes frontend energy, but also enables efficient synchronization across cores in a vector group (Section 2.5.2).

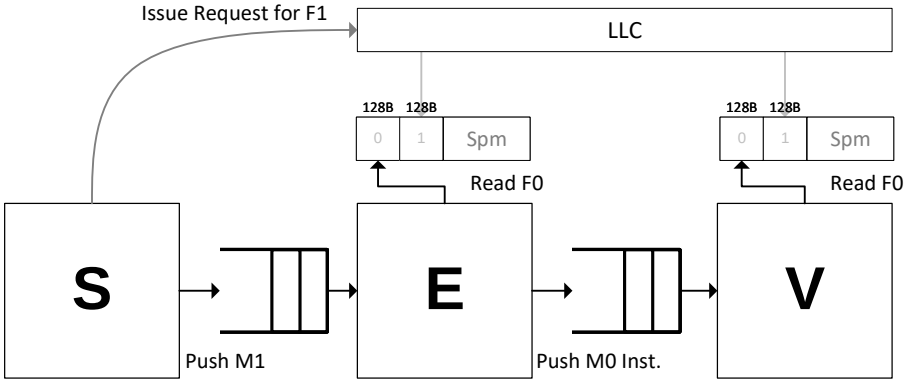


Figure 2.3: A logical view of frames and microthreads. The vector cores read frame 0 (F0) to execute microthread 0 (M0). Concurrently, the scalar core issues memory requests to fill the next frame (F1) and initiates the corresponding microthread (M1) that will consume this frame.

2.3.3 Vectorized Memory Access

A key advantage of vector architectures is their ability to exploit regular memory access patterns. Vector groups need a way to aggregate many small memory demands from all the cores and emit a single, wide request. Our ISA relies on the scalar core to centralize vector memory requests, which entails two main mechanisms: (1) a decoupled access/execute (DAE) scheme to let the scalar core run ahead and feed data to the vector cores, and (2) support for wide load instructions that run on the scalar core.

Decoupled Access/Execute for Vector Groups

Vector groups centralize their regular memory requests on the scalar core to amortize their overhead. To make this work, we adopt a lightweight decoupled access/execute (DAE) [88] scheme that lets the scalar core run ahead and load data to be delivered to the vector cores.

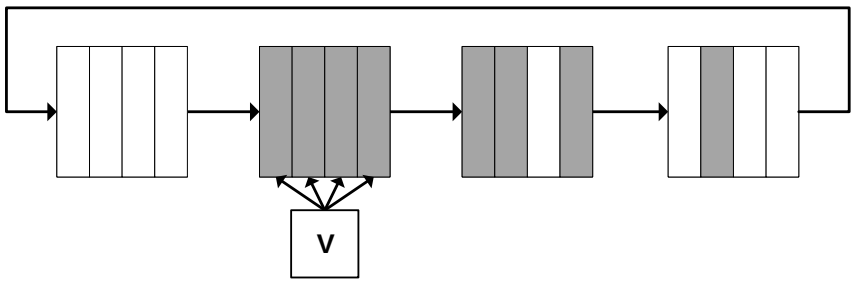


Figure 2.4: An example state of the frame queue with four frames each containing four words. Older frames are farther left. The first frame is freed and awaiting memory. The vector core is currently accessing the second, full frame. The third and fourth frames are partially full (data is still arriving) and the vector core cannot read them yet.

Our scheme reuses the vector cores' local scratchpads as a queue for the incoming data. To simplify the queue's bookkeeping, the architecture organizes the loaded data into chunks of memory called *frames*. Each frame contains the data that a single microthread needs to consume.

Figure 2.3 shows a logical view of the frame queue. The scalar core issues loads to fill frames, and vector cores consume frames in creation order—but the order that data arrives *within* a given frame does not matter. Vector cores maintain a circular buffer of frame-sized regions in their scratchpads. Figure 2.4 illustrates an example state of this circular buffer: the core can read from the head of the queue while the memory system concurrently fills future frames.

Before forming a vector group, all cores configure their frame size and total number of frames by writing to a CSR. This event allocates a fixed amount of space on the scratchpad to be used as a logical frame queue. The rest of the scratchpad is free to be used for programmer-allocated data and the stack.

Within microthread code, the vector cores consume frames from their scratch-

pads. The instruction `frame_start` stalls the current microthread until the frame at the head of the queue is ready, i.e., all its data has arrived. Each vector core executes `frame_start` to receive a base offset to the current frame on writeback. The vector core can then freely read and write the frame's region of memory in the scratchpad. When it finishes using the memory, the vector core uses another instruction, `remem`, to free the current frame.

At the C level, using frames on vector cores looks like this:

```
int frame_ptr = FRAME_START();
int a = spad[frame_ptr + 0];
int b = spad[frame_ptr + 1];
int c = a+b;
REMEM();
```

Our architecture's DAE mechanism is only for the load path—stores do not use it. The primary reason is that loads are on the critical path for vector computations, but stores typically are not, so it is possible to use simple non-blocking store operations that hide memory latency without a DAE setup. Standard non-blocking stores also avoid the cost for execution units to synchronize and forward output data for coalescing on a central access unit.

Wide Vector Loads

The reason for centralizing a vector group's loads is to coalesce them into *wide accesses* for contiguous chunks of data. We augment our architecture with a simple way to issue wide loads that reuses the existing memory network on the manycore. The key component is a new *vector load* instruction that the scalar core can run on behalf of its vector group. A vector load requests an entire cache line from the LLC

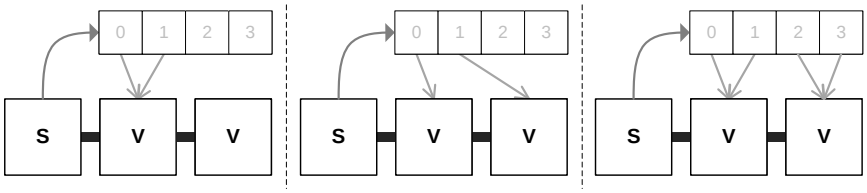


Figure 2.5: A scalar core sends a request to a cache line and generates multiple responses to the vector cores. (Left) Single load of fetch width 2. (Middle) Group load of fetch width 1. (Right) Group load of fetch width 2.

and distributes the results to the vector cores in the group, one chunk at a time. We limit the length of a vector load to the cache line size: aligned loads (beginning at offset 0 in the line) request data from a single cache line and unaligned loads request data from *at most* two cache lines.

We add a new instruction for non-blocking vector accesses:

```
vload sp, addr, off, width, var
```

The operands are: (1) the offset in the receiving scratchpads, (2) the source memory address, (3) the core offset in the vector group to receive the first response, (4) the access width per vector core, and (5) the variant (see below). Operands (1), (2), and (3) allow data to be loaded from main memory directly into a scratchpad memory. Operands (4) and (5) configure the style of vector access. We pack these operands into two registers and an immediate in order to fit within a standard RISC-V instruction format.

While `vload` by default only supports aligned cache lines, we also support unaligned loads using pairs of instructions. Two additional variants load a *suffix* of one line and a *prefix* of a second line that combine to form a full line-sized block. The program issues both instructions with the same arguments to achieve an unaligned load.

Vector loads support three variants which instruct the LLC where to send each part of the accessed cache line:

- Single: Sends part of the line to a single vector core.
- Group: Sends consecutive parts of the line to each core in the vector group.
- Self: Sends all data back to the core that sent the request.

Figure 2.5 illustrates the variants' request and response paths.

We find that the supported variants can map effectively to many patterns, and code that does not fit them can fall back to single-word loads. The work division strategy determines what type of load can be used. Consider this sum example:

```
for (int i = tid; i < N; i += VLEN)
    for (int j = 0; j < M; j += FLEN)
        c[i] += a[i*M+j];
```

Each vector core performs a separate sum across the `j` dimension. The scalar core must issue a separate vector load of size `FLEN` for each vector core.

```
for (int core = 0; core < VLEN; core++)
    vload sp, a[(i+core)*N+j], core, FLEN, SINGLE;
```

However, if the work division strategy was changed, then a more efficient variant could be employed.

```
for (int i = 0; i < N; i++)
    for (int j = tid*FLEN; j < M; j += FLEN*VLEN)
        c_partial[i] += a[i*M+j];
```

Here, the vector cores work on the same sum across `j`. The scalar core can fetch consecutive chunks of `j` to each core to generate a partial sum. In this case, a group

load can be used to fetch `FLEN*VLEN` worth of data:

```
vload sp, a[i*N+j], 0, FLEN, GROUP;
```

The added cost is the need for a reduction of the partial sums, but this can be done somewhat cheaply.

In some cases such as a single-level loop nest, any variant can suffice—the choice comes down to performance rather than correctness. Groups loads can be more efficient because they require fewer scalar instructions and memory requests.

2.3.4 Vector Odds and Ends

Vector groups emulate classical vector machines and can support traditional vector operations, including prediction, gather/scatter, and shuffles.

Predication lets vector machines implement conditional execution without resorting to divergent control flow. Our microthreads support predication using a single mask that consists of a 1-bit flag on each vector core. Two predication instructions set the per-core flag based on a comparison between two registers:

```
pred_eq rs1, rs2
pred_neq rs1, rs2
```

When the flag is 0, a vector core executes all instructions as nops until a subsequent predication instruction sets the flag back to 1. At the C level, a simple condition:

```
if (cond == 1) { c = a + b; }
```

Can be implemented this way:

```
PRED_EQ(cond, 1);
c = a + b;
```

```
PRED_EQ(0, 0);
```

By using a single, implicit mask, our ISA's predication feature avoids the need for special predicated instructions, as in other vector ISAs, that would make the core more complex even in MIMD mode. The scheme adds more instruction overhead than a typical predication scheme, but the worst cases are for nested conditionals where the kernel is better suited to standard manycore mode anyway.

Vector groups perform scatters/gathers by performing word-sized memory operations on the vector cores (not the scalar core). The accesses are non-blocking to avoid stalling the entire vector group to wait for each component memory access.

To perform a shuffle, vector cores execute remote stores to other cores' scratchpads (a common feature in manycore architectures). To ensure consistent access to shuffled data, the vector group must synchronize (Section 2.5.2).

2.4 The Rockcress Architecture

This section describes a design implementing the software-defined vector ISA from Section 2.3. The key architectural mechanism is *instruction forwarding:* vector cores share a single instruction stream by passing each instruction directly to one another, hop by hop.

2.4.1 A Manycore Baseline

Our extensions start with a minimal set of assumptions about a standard manycore processor. We base our assumptions on a mature open-source design [11], which

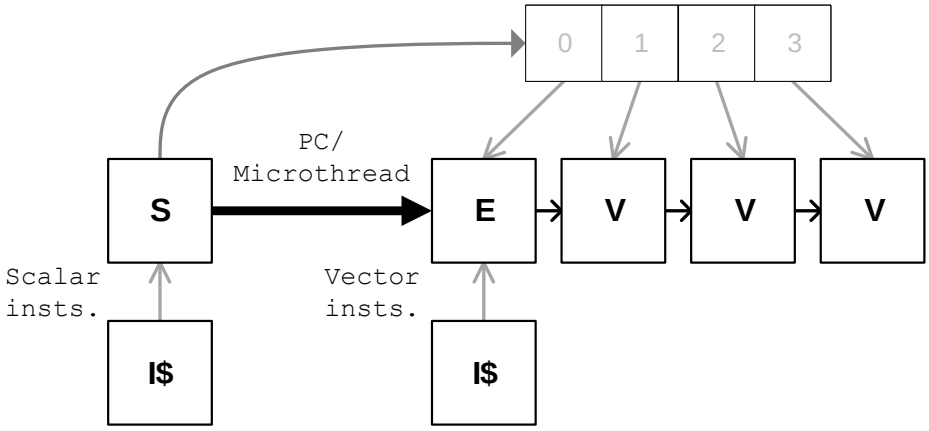


Figure 2.6: The scalar (S), expander (E), and vector (V) roles in a vector group of length four. The expander is a vector core but also fetches instructions.

consists of a tiled grid of simple in-order RISC-V CPUs. Each CPU has a local instruction cache but an explicitly managed scratchpad for data—we do not assume cache coherence. We assume a NoC connecting the cores and the global memory system. The array shares a set of LLCs in front of off-chip DRAM; these caches partition the global address space by striping.

2.4.2 Instruction Forwarding

We add a systolic instruction forwarding network (inet) that is separate from the manycore's existing data network. Whereas the data network may be a dynamic, packet-switched network, the inet is a simple static network of direct 1-cycle connections between neighboring tiles. Forwarding an instruction consists of a 32-bit register read and write, which consumes significantly less energy than an I-cache hit.

A vector group consists of one scalar core, one expander core, and many vector

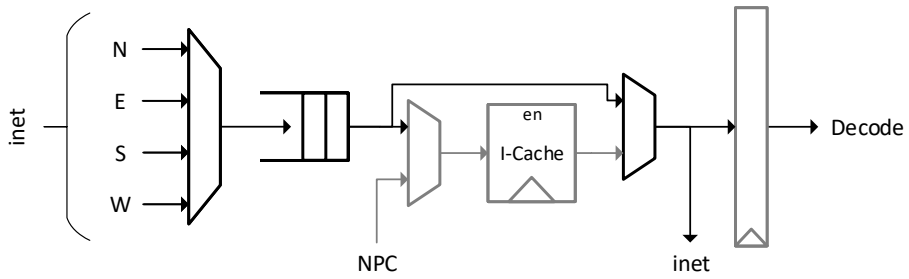


Figure 2.7: Modifications to the fetch stage to interface with the inet and form vector groups. The black components are additions; gray indicates parts of an ordinary fetch stage.

cores. Figure 2.6 illustrates an example vector group of length four. Only the scalar core and the expander core need active processor frontends and I-caches; the remaining non-expander vector cores receive instructions from the inet. Energy efficiency scales with vector length as more cores become vector cores.

Figure 2.7 shows how we modify cores' fetch stage to take advantage of the inet. A vector core receives instructions in its fetch stage via a single inet queue. The queue is driven by a multiplexer that selects between the output of each of the four adjacent cores. A vector core bypasses the I-cache directly to the decode stage, while the expander core uses PCs received from the scalar core to fetch instructions from the I-cache. The instructions sent to the decode stage are also output on the inet to be used by adjacent cores.

The rest of this section describes how instruction forwarding works from the perspective of each of the three roles.

Scalar core Scalar cores act independently and fetch from the scalar instruction stream. Its instructions include both normal scalar computations and `vissue` instructions, which launch microthreads on the vector cores (Section 2.3.2). It is

not possible to cancel a speculatively launched microthread, so the scalar core sends the launch message after the `vissue` commits. The scalar core can execute all RV-G instructions; it is typically responsible for integer address calculations and memory requests.

Expander core The expander core can run most RV-G instructions, including direct jumps and function calls. It may also execute conditional branches, but the program must ensure that the branch outcome is the same for the entire vector group. The expander core pauses instruction fetch when it encounters a branch to avoid sending the wrong instructions to other vector cores and resumes fetching when the branch path is resolved later in the pipeline. The expander core does not forward control flow instructions because other vector cores cannot diverge. The microthread finishes when the expander core encounters a `vend` instruction.

The main purpose of the expander core is to enable asynchronous computation. The scalar core need not facilitate instruction fetch for the vector cores, so it can freely run ahead.

Vector core Vector cores act as vector computation lanes. Their task is to do arithmetic work with low control overhead. They automatically forward every instruction they receive to their downstream vector cores—they never squash instructions. Arithmetic, memory, and predication instructions are allowed in a vector core's instruction stream: control flow is not allowed.

2.4.3 Scratchpad-Based Decoupled Access

We augment the cores' scratchpads to support the logical DAE queue described in Section 2.3.3. The key hardware support is metadata that tracks when data is available and ready for consumption. A straightforward but costly approach would be to add per-word full/empty bits to the entire scratchpad. Rockcress opts for a lower-complexity solution that tracks readiness at a frame granularity.

The hardware maintains a set of counters that track the number of words that have arrived in a given frame. Whenever a word arrives to the scratchpad from the data network, the core increments the counter for the frame containing the destination address. When the counter's value equals the configured frame size, the entire frame is ready and the core can begin using the data. When the core frees the frame, the values of each counter are shifted to the left by one and the rightmost count is set to zero. This counter-based mechanism allows data to arrive out of order within a frame while still enforcing in-order consumption of frames themselves.

The number of frame counters in the hardware limits the number of frames that can be *open* (i.e., receiving data) simultaneously. More counters let the DAE scheme run farther ahead and tolerate more variability in memory latency. Our Rockcress implementation has five 10-bit frame counters.

2.4.4 Architecture Support for Wide Accesses

Our baseline processor has a collection of shared LLCs that sit between the manycore array and main memory. We modify these caches to support the wide

access instructions described in Section 2.3.3. The caches need to send multiple chunked responses for a single line-sized request.

We add a counter to each cache, which it uses to serially generate responses for consecutive words in a line. When a wide access hits in the cache, it initializes this counter. Each cycle, the cache generates a response based on the base address plus the current count and then increments the counter. Accesses generate serial responses to a given base core and base scratchpad offset ($Core$, $Offset$) as:

$$(Addr + Cnt) \rightarrow (BC + Cnt/RPC, BO + Cnt\%RPC)$$

where Addr is the memory address, Cnt is the current response count, BC is the base core to respond to, RPC is the responses per core, and BO is the base scratchpad offset.

Caches can generate responses to any rectangular vector group layout, but this layout must be provided by a wide access packet. The scalar core's memory unit generates a wide access packet using the vector group's configuration (`vconfig`) and the in-flight `vload` instruction. `vconfig` provides the base core of the vector group, which is the top-left vector core, and the group's dimensions. This is combined with the number of responses and the offset (from the base core) specified in the `vload`. The unit also determines the proper access widths and offset in the case of unaligned loads.

2.5 Programming with Software-Defined Vectors

This section covers two programming concerns: compilation and synchronization within vector groups.

```
for (int i = 0; i < 128; ++i) {
  process(i, a[i]);
}
```

(a) Original DOALL loop.

```
VECTORIZE(get_vector_config(...));
VECTOR_ISSUE({
  int vec_i = 0;
  float a_spad;
});
for (int i = 0; i < 128; i += VECTOR_LENGTH) {
  VECTOR_LOAD(a_spad, &(a[i]), GROUP);
  VECTOR_ISSUE({
    process(vec_i + thread_id, a_spad);
    vec_i += VECTOR_LENGTH;
  });
}
DEVECTORIZE();
```

(b) Simplified Rockcress vectorized code.

Figure 2.8: Example code for a simple DOALL loop.

2.5.1 Compiling Vector Microthreads

We use C preprocessor macros and a custom assembly-manipulation pass to provide a programming model for implementing code for Rockcress. The workflow first compiles application C code to RISC-V assembly using stock GCC, and then our custom pass runs on the assembly to produce executable Rockcress code.

Figure 2.8 shows a simple DOALL loop in sequential C and its vectorization for Rockcress. The vectorized code strip-mines the loop to move by `VECTOR_LENGTH` steps. C macros `VECTORIZE` and `DEVECTORIZE` wrap inline assembly to manipulate the `vconfig` CSR (Section 2.3.1) to create and destroy a vector group. We provide a `VECTOR_LOAD` macro that emits a wide load instruction to copy data from main memory to the vector cores' scratchpads.

Next, the code uses `VECTOR_ISSUE` to delimit microthread code. The compiler splits the program into code that runs on the scalar core and the microthread

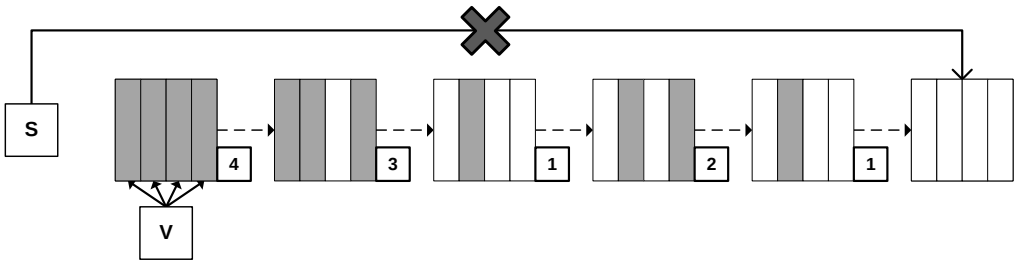


Figure 2.9: The scalar core cannot fetch words for frames that are ahead of the active frame by more than the number of frame counters.

bodies that run on the vector cores. Critically, the two different contexts maintain separate state. For example, the loop induction variable, `i`, resides on the scalar core, so the vector code maintains its own copy, declared as `vec_i`. The code uses one microthread to initialize the vector state and a second microthread for the loop body, including the increment to `vec_i`.

The Rockcress compiler must generate assembly for individual microthreads while preserving their semantics in the context of the entire program. For example, to generate correct and efficient assembly for the statement `vec_i += VSIZE`, the compiler needs to allocate a register for `vec_i` that persists across invocations of the microthread. It does not suffice to compile each microthread in isolation, such as in a separate function—the compiler must be aware that the body microthread is invoked in a loop, and that the loop follows the setup microthread. Instead, the Rockcress compiler preprocesses the program to extract the microthread bodies into a separate C source file, compiles it separately, and then uses an assembly post-processing pass to merge the microthreads back into the main scalar code.

2.5.2 Intra-Group Implicit Synchronization

The cores within a vector group occasionally need to synchronize. Most prominently, scalar cores performing decoupled accesses (Section 2.3.3) need to ensure that they do not run too far ahead of vector core execution; as Figure 2.9 illustrates, the scalar core could overrun the number of hardware counters allocated to the frames. Shuffles must also synchronize to avoid freeing frames that other cores need.

We implement a compiler-driven scheme that *implicitly* synchronizes vector groups by waiting for instructions to propagate through the inet. The key insight is that the inet forms a bounded queue: it is impossible to send instructions into it indefinitely, so slow cores later in the group will cause earlier cores to block. Therefore, a core can only stay a bounded number of instructions "behind" any other core in the vector group. To synchronize, each core can wait until it has seen enough instructions execute such that all cores in the group must have passed a given barrier point.

To perform implicit synchronization, we first need a bound on this *instruction delay:* a value n such that any two instructions in the pipelines of any cores in the vector group may be separated by at most n dynamic instructions. To find this bound, we identify the longest path from any core to any other core in a $m \times m$ vector group and include every source of queueing along the path, including the inet queue itself and the stages of the CPUs' pipelines. We derive this bound:

$$n = (2m - 2) \cdot q_{inet} + \sum_{i}^{stages} (buf_i) + ROB$$

Here, $2m - 2$ is the longest instruction forwarding path in the vector group, q_{inet} is the size of the inet queues, buf_i is the length of pipeline buffers in the decode, rename, issue and commit stages, and ROB is the number of entries in each core's

reorder buffer. This n bounds the number of instructions that can be accommodated in the buffers before stalling the inet.

Our compiler-driven implicit synchronization scheme implements a vector-group barrier by ensuring that code before and after the barrier is separated by at least n microthread instructions. To prevent excessive scalar-core runahead, for example, we need to ensure that the scalar core does not request too many data frames without issuing microthreads to consume them. We first compute the maximum number of in-flight frames:

$$num\_active\_frames = \left\lceil \frac{n}{instructions\_per\_frame} \right\rceil$$

where $instructions\_per\_frame$ is the length of a `vissue` microthread. Using this value, the compiler can determine how many frames the scalar core can safely run ahead:

$$ahead\_offset = max\_frames - (num\_active\_frames + q_{inet})$$

where max_frames is the number of frame counters and q_{inet} accounts for the maximum number of queued microthreads between the scalar and expander cores. The compiler uses this bound to ensure that the scalar core does not exhaust the frame counters in the vector cores.

2.6 Experimental Setup

We model a baseline manycore machine, the software-defined vector extensions, and a competitive GPU using the gem5 cycle-level simulation infrastructure [12].

Table 2.1: Microarchitectural parameters for the models.

(a) Manycore.

Component	Setting
Cores	64
ALU Latency	1
Multiply Latency	2
Divide Latency	20
FP ALU Latency	3
FP MUL Latency	3
SIMD Width	4 words
SIMD ALU Latency	3
Load Queue Entries	2
inet Queue Entries	2
Cache line Size	64 bytes
I-Cache Capacity	4kB
I-Cache Hit Latency	1 Cycle
I-Cache Ways	2
Spm Capacity	4kB
Spm Hit Latency	1 Cycle
Router Hop Latency	1
On-Chip Net Width	4 words
LLC Capacity	256kB
LLC Banks	16
LLC Hit Latency	1 Cycle
LLC Ways	4
DRAM Latency	60ns
DRAM Bandwidth	16GB/s

(b) APU.

Component	Setting
Compute Units (CUs)	4
Lanes per vALU	16
vALUs per CU	4
vALU Latency	4
Wavefront Size	64
Wavefronts per CU	4
Registers per CU	8192
Cacheline Size	64 bytes
TCP Capacity	16kB
TCP Hit Latency	1 Cycle
TCP Ways	16
TCC Capacity	256kB
TCC Hit Latency	2 Cycles
LLC Capacity	4MB
LLC Hit Latency	2 Cycles
LLC Ways	16
CPU L2 Capacity	512kB
CPU L2 Hit Latency	2 Cycles
CPU L2 Ways	8
CPU L1D Capacity	64kB
CPU L1I Capacity	32kB
DRAM Latency	60ns
DRAM Bandwidth	16GB/s

2.6.1 Manycore & Rockcress

For the cycle-level model of Rockcress, we start with a baseline manycore that reflects the assumptions in Section 2.4.1: the Celerity open-source RISC-V manycore [11,25]. Our gem5 model uses the Ruby memory system and the Garnet2.0 mesh NoC. Each tile has an I-cache, a scratchpad, and an 8-stage CPU with in-order issue, out-of-order writeback, and in-order commit. At the top and bottom of each mesh column, there is a shared LLC. DRAM is connected to each LLC slice and uses a fixed-latency, fixed-bandwidth model. Table 2.1a lists the microarchitectural parameters. The SRAM latencies are estimated using CACTI 6.5 [67] assuming a 32 nm process at 1 GHz. We augment the baseline model to support software-defined vectors by modifying the CPUs, scratchpads, and LLCs as described in

Section 2.4.

The LLCs represent disjoint address spaces and thus do not require cache coherence. They are write-back, pseudo-LRU replacement caches with 64-byte lines, which limits wide accesses to 16 words. We also experiment with longer lines to increase the amount of coalescing that can occur.

A vector request can only generate one word response per cycle per port (CPU-side or memory-side). We experiment with various on-chip network widths to increase the number of words that can be sent per cycle to a single core.

We also consider manycore configurations with standard fixed-length SIMD units in each core using the RISC-V vector extension [2]. These configurations, unlike Rockcress, optimize for compute density. Rockcress's extensions can also apply to this baseline, aggregating short per-core SIMD units into wider vector groups.

2.6.2 Energy Model

We develop a first-order energy model to complement our cycle-level simulation. The model assigns energy costs to simulation statistics, such as memory accesses and instruction executions, and computes a total dynamic energy for a benchmark execution. When a core is in vector mode, it omits the energy costs for fetch and I-cache accesses.

We use CACTI 6.5 [67] to model the access costs of the I-caches, scratchpads, and LLCs. A 4-wide vector load consumes as much energy in the LLC as 4 scalar loads in our model. We model a single I-cache fetch per instruction.

For CPU energy, we use a published breakdown for Ariane [111]. Ariane is a single-issue RISC-V core with in-order issue, out-of-order writeback, and in-order commit, like our gem5 model. It has been used in a manycore [4]. Zaruba and Benini [111] break down Ariane's energy per component, per instruction, per cycle. We use the energy costs as follows:

- The I-cache and D-cache costs are substituted for our modeled I-cache and scratchpad respectively.
- We omit virtual memory (VM) and performance counter (CTS) costs because our architecture lacks them.
- Four different instructions costs (integer ALU, integer MUL, integer DIV, and load/store) are mapped to corresponding statistics from the gem5 simulation.
- Floating-point operations map to costs for integer operations (including FMA, which counts as multiply).
- For MUL and DIV, we scale the multiplier energy cost to the maximum number of cycles the operation takes (2 cycles for multiply and up to 64 cycles for divide).
- Vector instruction costs are estimated by multiplying the functional unit and writeback costs by the vector length. The rest of the instruction cost is left unchanged.

2.6.3 GPU

We use an existing gem5 APU model [40] to model a GPU. The APU consists of CPUs coupled to a GPU via a shared LLC. The GPU is divided into compute units

Table 2.2: PolyBench/GPU applications used in the evaluation.

Name	Input	Description
2dconv	2048×2048 image	3x3 filter applied to an image
2mm	256×256 matrix	Two matrix multiplies
3dconv	256×256×256 volume	3x3 filter applied to a volume
3mm	256×256 matrix	Three matrix multiplies
atax	2048×2048 mat, 2048 vec	Mat-transpose vec ($y = A^T Ax$)
bicg	2048×2048 mat, 2048 vec	Biconjugate Gradient Method
corr	512×512 matrix	Matrix correlation
covar	512×512 matrix	Matrix covariance
fdtd-2d	512×512 mat, 30 tmax	Finite-difference Time-domain
gemm	256×256 matrix	Matrix mul. ($C = \alpha AB + \beta C$)
gesummv	4096×4096 mat, 4096 vec	Matrix vector ($y = \alpha Ax + \beta Bx$)
gramschm	320 vectors of length 320	Gram–Schmidt decomposition
mvt	4096×4096 mat, 4096 vec	Mat-vec(Ax_1), transpose($A^T x_2$)
syr2k	256×256 matrix	Symmetric Rank-2K Update
syrk	256×256 matrix	Symmetric Rank-K Update

(CUs) with four vector ALUs (vALUs) each. Each vALU has 16 lanes and executes a 64-thread wavefront every four cycles.

We tune the microarchitectural parameters to make a rough comparison with the manycore. Table 2.1b lists the model's parameters. The DRAM model is identical to the one in the manycore model. The L2s (called LLC in the manycore and TCC in the GPU) have the same capacity. The L1s (scratchpad in manycore, TCP in GPU) have different sizes but remain that way due to architectural differences. The GPU also has an additional L3 (GPU LLC) that is shared with two CPUs in the system. We model hit latencies using CACTI, assuming a 1 GHz frequency and 32 nm process.

The number of CUs in the GPU is determined by arithmetic intensity per area, which will be higher than a manycore. In Ariane, only 15% of the core area is devoted to the integer arithmetic unit. We roughly estimate that there should

Table 2.3: PolyBench/GPU software optimizations.

Name	Algorithm opt.	Mem opt.	Kernels
2dconv			1
2mm	Tiled Outer Product.	Transpose	2
3dconv			1
3mm	Tiled Outer product	Transpose	3
atax	Loop reordering		2
bicg			2
corr	Kernel fusion	Transpose	2
covar	Kernel fusion	Transpose	2
fdtd-2d			3
gemm	Tiled Outer product	Transpose	1
gesummv			1
gramschm			3
mvt			1
syr2k			1
syrk			1

be 4× more ALU lanes in the GPU configuration than there are ALUs in the manycore.

Each GPU CU has four wavefronts of 64 threads each. Larger GPU designs typically have more wavefronts to hide memory latency, however, these designs require significant resources and would be unfairly provisioned compared to the manycore.

2.7 Evaluation

This section uses our cycle-level models to compare manycore, GPU, and software-defined vector execution efficiency.

Table 2.4: Benchmark configurations evaluated.

Config. Name	Group Size	SIMD Words	Wide Access	DAE	Long Lines
NV	1	1			
NV_PF	1	1	×		
PCV_PF	1	4	×		
V4	4	1	×	×	
V16	16	1	×	×	
V4_PCV	4	4	×	×	
V16_PCV	16	4	×	×	
V4_LL_PCV	4	4	×	×	×
V16_LL	16	1	×	×	×
V16_LL_PCV	16	4	×	×	×
BEST_V	4 or 16	1	×	×	?
BEST_V_PCV	4 or 16	4	×	×	?
GPU	1	16			

2.7.1 Benchmarks

We evaluate Rockcress on all 15 benchmarks in the PolyBench/GPU suite [36] (Tables 2.2 and 2.3). We compile the C code using GCC 10.1.0 with `-O3` optimization, targeting the uncompressed RV-G ISA and vector extensions. We optimize the benchmarks for each target by unrolling loops using the canonical GCC pragma. We translate the GPU benchmarks from CUDA to HIP and compile using HIPCC.

On the software-defined vector architecture, we identify kernels, form vector groups at the beginning of each kernel, and disband them at the end. The cores use a global barrier between kernels. The typical mapping strategy is to partition a kernel's outer loops among vector groups and inner loops among the cores within the groups. In general, microthreads consist of multiple inner loop iterations to reduce the communication cost between vector and scalar cores. We compare 4- and 16-wide vector groups and create the maximum number of vector groups that fit within 64 cores. We strip-mine the loops in the kernels according to the configured

vector length.

We check correctness using a serial version of each kernel. Floating point errors never exceed the thresholds specified in the PolyBench/GPU implementations.

2.7.2 Configurations

We compare multiple versions of the benchmarks running on the manycore and a separate GPU version (Section 2.6.3). Table 2.4 enumerates the naming convention and corresponding features. We consider a basic MIMD baseline (NV), a competitive baseline optimized with non-blocking wide accesses for MLP (NV_PF), and a baseline with narrow per-core SIMD units (PCV_PF). The MLP optimized baselines use the *vload* instruction to fetch full cache lines into their private scratchpads. NV_PF approximates the Celerity manycore [11, 25] which supports non-blocking scalar loads. Vector group lengths can be configured at compile time, so we compare the baselines with the fastest Rockcress configuration (BEST_V). BEST_V includes both V4 and V16 and the ability to choose a larger cache line size. All MLP optimized benchmarks were able to use SIMD extensions with the exception of gramschm. We choose the closest valid configuration in place of them for completeness (PCV without MLP for PCV_PF, V4 for V4_PCV, and V16 for V16_PCV).

In the vector configurations, the vector groups leave some unused cores. In V16, for example, we can create 3 groups of 17 tiles, so we use only 80% of the tiles in total. V4 uses 94% while NV and NV_PF use 100%. While it is possible to use the remaining cores in independent mode or a smaller vector group, our evaluation leaves them idle for simplicity.

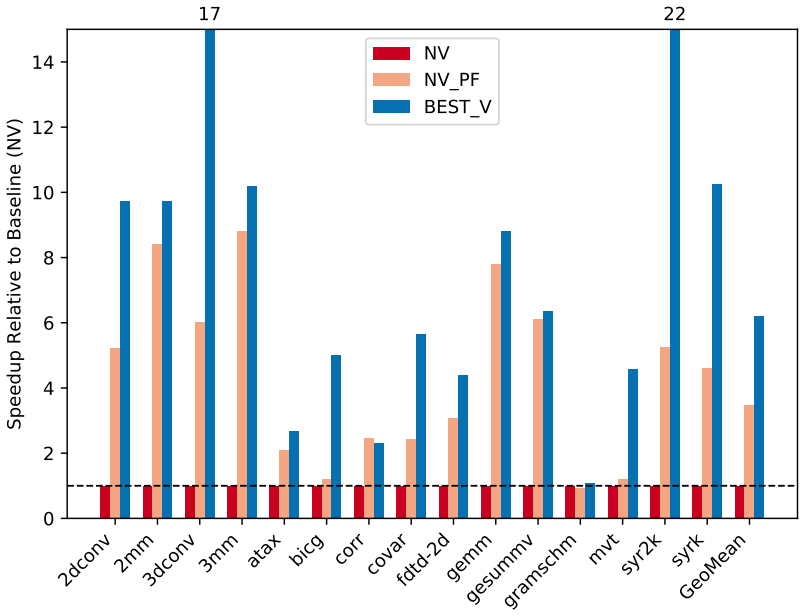


Figure 2.10: Speedup relative to NV baseline.

2.7.3 Performance

Figure 2.10 shows the speedup over the basic manycore baseline. Both NV_PF and the vector configurations outperform the NV baseline by exploiting MLP. However, the vector configurations are the fastest and outperform the manycore baseline optimized with non-blocking wide loads (NV_PF) by 1.7× on average.

The benefit of software-defined vectors varies by application. For example, 3dconv using 16-wide vector groups outperforms the NV_PF baseline by 2.0×, while 2mm only approximately matches the performance of NV_PF. Along with 3dconv, bicg and mvt perform exceptionally well in a vector configuration, with 4.1× and 3.8× speedup over NV_PF respectively. The only benchmark that did not improve due to decoupled access was gramschm. This benchmark is not able to take advantage of vector loads due to its access pattern and must resort to scalar loads.

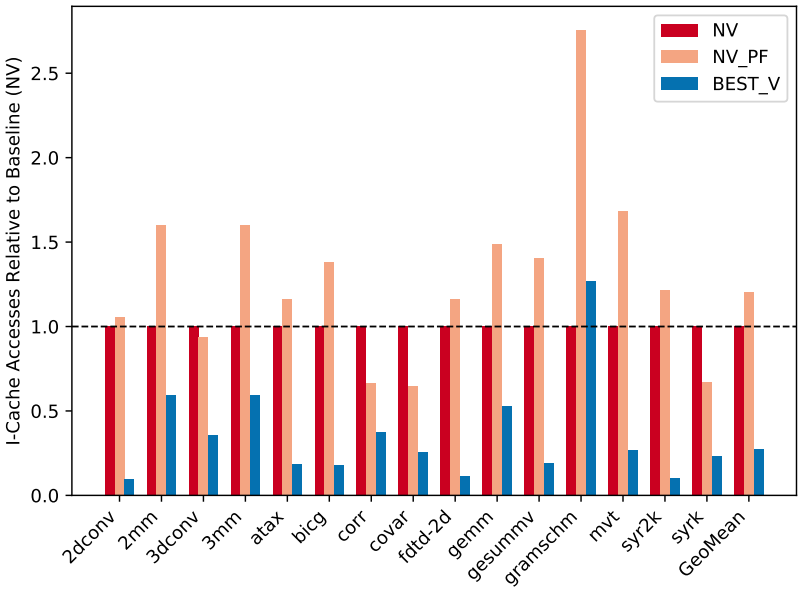


Figure 2.11: I-cache accesses.

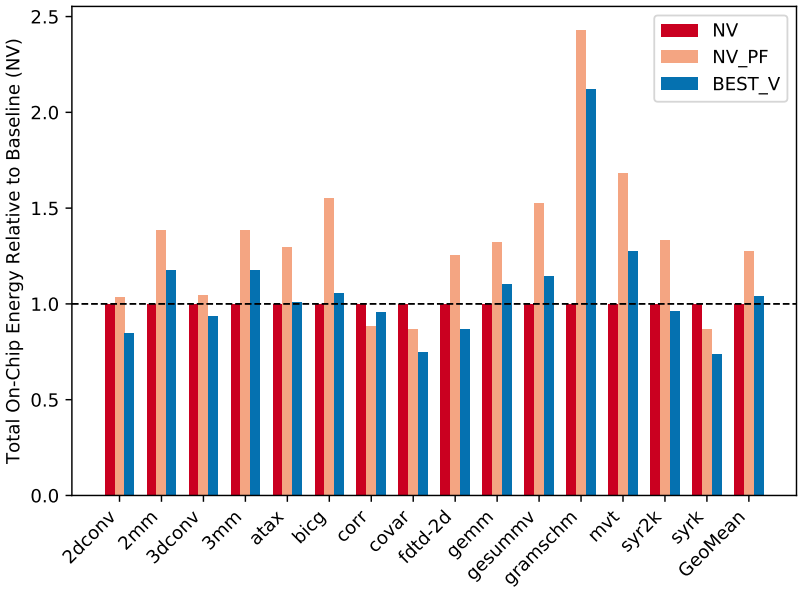


Figure 2.12: Total on-chip energy.

2.7.4 Energy

Rockcress saves energy by reducing the total cost of fetching instructions. Figure 2.11 shows the number of I-cache accesses in each configuration. V4 and V16 reduce I-cache accesses compared to NV by 2.2× and 4.7× respectively. NV_PF increases I-cache accesses over the baseline because it adds instructions to stage data to the scratchpad before moving it to a register. The vector configurations incur the same cost but amortize it over fewer frontends. V4 and V16 reduce I-cache accesses over NV_PF by 2.7× and 5.6×.

Figure 2.12 shows the resulting changes in total energy according to our energy model. The reductions in fetch costs translate to a 22% reduction in energy versus the NV_PF baseline, approximately matching the NV baseline. Vector configurations are not more energy efficient than NV because the energy the inet saves is offset by the additional energy required by the scratchpad to exploit MLP.

2.7.5 Scalability

This section evaluates the viability of the NV_PF baseline and highlights its main bottlenecks.

Baseline Scalability We evaluate the scalability of our baseline manycore system (NV_PF). Figure 2.13 shows the speedup for an increasing number of cores over a single core (with the same memory system capacity and bandwidth). The scability of 2mm, 3mm, and gemm increases linearly, while the majority of benchmarks are sub-linear after 16 cores.

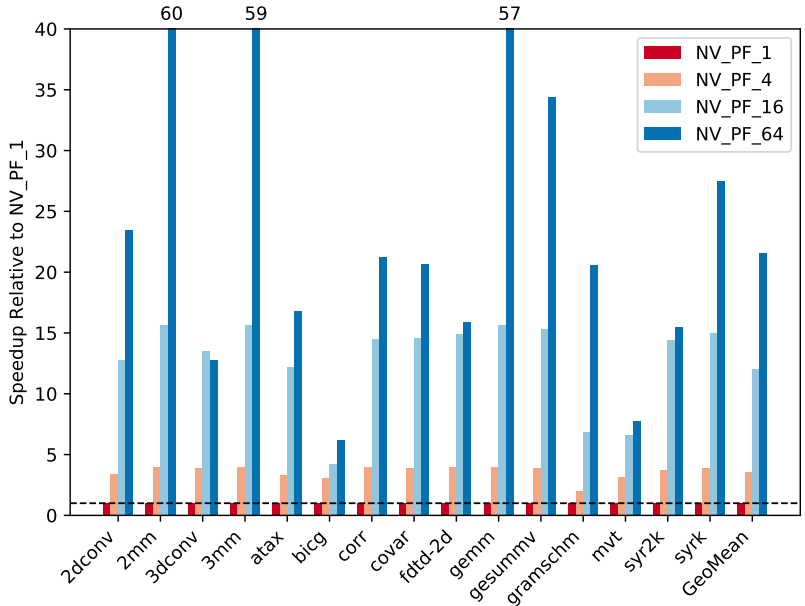


Figure 2.13: The relative speedup for an increasing number of cores compared to a single core processor.

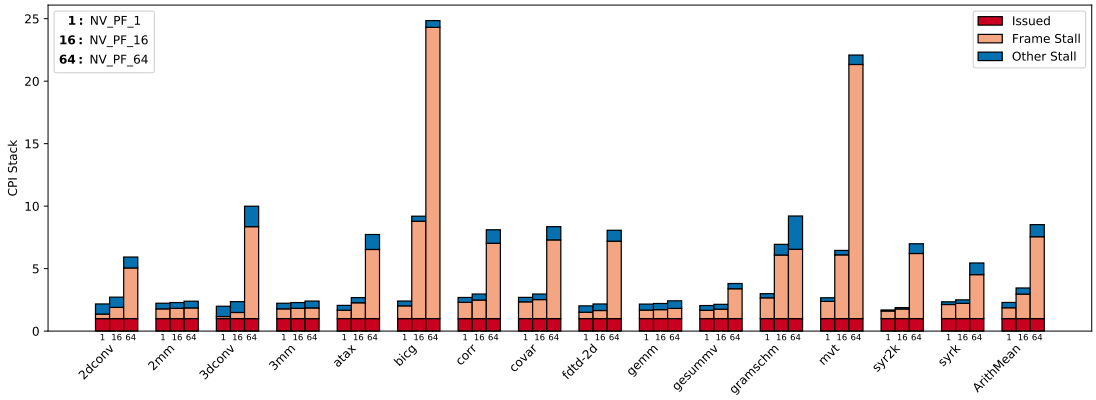


Figure 2.14: A CPI stack showing the core pipeline stalls for various manycore sizes and benchmarks. As the number of cores increases, most benchmarks are dominated by frame/memory stalls.

Figure 2.14 details the core performance for different manycore sizes using a CPI stack.[1] Benchmarks that do not scale well see a significant increase in stalls due to memory past 16 cores. At larger core counts, the majority of stalls are waiting on loads (labeled *frame stall* in the figure).

[1]Each stacked bar in the CPI stack shows the relative number of cycles where an event occurs in a core's issue stage (normalized to cycles where an instruction was issued). Each total stacked bar height indicates the actual CPI of the core.

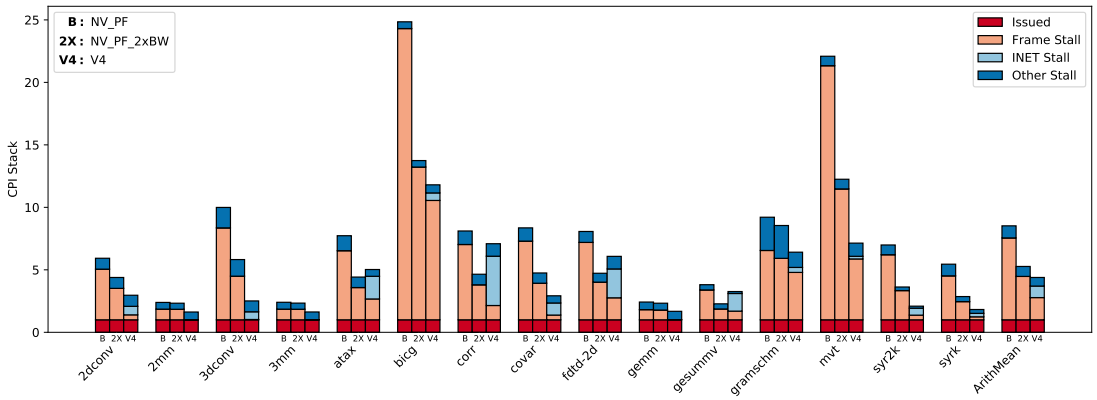


Figure 2.15: A CPI stack showing the core pipeline stalls for the baseline (NV_PF), the baseline with 2× the DRAM bandwidth (NV_PF_2xBW), and vector groups of size four (V4). V4 with 16GB/s of DRAM bandwidth outperforms the baseline with 32GB/s of DRAM bandwidth due to better utilization of the existing memory bandwidth. The vector configurations only average the events from the expander cores because the root cause of a stall is not apparent in a non-expander vector core.

DRAM bandwidth DRAM bandwidth is the main bottleneck for larger core counts. The effective DRAM bandwidth per core decreases from 16 GB/s/core at one core to 0.25 GB/s/core at 64 cores. While this is enough bandwidth for benchmarks with high reuse, such as 2mm, 3mm, and gemm, it bottlenecks others.

A recent iteration of Celerity [11] dedicates two HBM2 channels per 128 cores which yields 0.5 GB/s/core (double our system's bandwidth). We experiment with increasing the memory bandwidth for the 64-core baseline and compare it to using a vector configuration. Figure 2.15 shows the effects on memory performance. The additional memory bandwidth provides a linear performance improvement for most benchmarks, indicating that they are bottlenecked by DRAM bandwidth. However, we find that using a vector configuration can improve memory performance with no additional DRAM bandwidth. Benchmarks such as 2dconv, 3dconv, covar, syr2k, and syrk see better overall performance with a vector configuration than with additional DRAM bandwidth. Other benchmarks such as atax, corr, fdtd-2d, and

gesummv see a reduction in memory stalls, but the overhead from using the inet overshadows this improvement overall. Section 2.7.6 shows how our DAE system and group wide accesses improve memory performance.

Software-defined vectors provide a way for DRAM-bottlenecked manycore architectures to make better use of memory bandwidth. For applications with regular parallelism, then, the technique facilitates scalability to larger core counts.

2.7.6 Characterization

Wide access performance Wide accesses can improve the cache hit rate and better utilize DRAM bandwidth. Figure 2.21 shows the cache miss rate for various configurations. Vector groups have a better hit rate than NV_PF for two reasons. First, loads can be coalesced across a vector group. Three benchmarks, atax, bicg, and mvt, use group loads where NV_PF cannot. The second reason is that fewer requests are sent per cycle in V4 because there are fewer scalar cores than independent cores in NV_PF.

Both bicg and mvt see the largest reductions in LLC misses. These benchmarks access the critical matrix column-wise which results in poor cache line utilization. Grouped loads are able to extract spatial locality across cores for these types of loads.

DAE performance DAE reduces the effective memory latency for vector cores, but we find that it does not completely eliminate memory stalls. Figure 2.19c shows the number of stalls in vector cores due to waiting for a frame to fill for V4 and NV_PF. Ideally, every vector core would never stall as is the case for 2mm, 3mm,

and gemm. V4 reduces frame stalls by about 2× over NV_PF. This plot does not show the absolute stall reduction, but instead is normalized to each configuration's run time. For example, bicg V4 is 2.3× faster than NV_PF and has about the same ratio in the plot. However, V4 actually has about 2.7× fewer absolute stalls.

Per-core hardware prefetchers could achieve similar speedups to our DAE techniques. However, software DAE offers flexibility: fixed hardware may be over- or under-utilized for a given kernel. Our DAE system's overhead is also amortized by having only one scalar core (access core) for multiple vector cores (execute cores).

Hardware vector units and GPU We compare Rockcress to traditional, fixed-length SIMD units in each core. Figure 2.16 shows the performance impact of adding SIMD units to both the baseline (PCV_PF) and Rockcress (BEST_V_PCV), along with the GPU.

Versus a GPU, Rockcress achieves 1.9× speedup on average. Benchmarks with high arithmetic intensity perform well on the GPU, as expected. These include 2mm, 3mm, gemm, 2dconv, and 3dconv. However, most benchmarks are memory-bound and are slower on the GPU. GPUs rely on massive multithreading and register files to hide memory latency; a larger GPU design would perform better on memory-bound benchmarks. We compare to an under-provisioned GPU design to highlight the area efficiency of Rockcress's DAE mechanism. Software-based DAE is more efficient at hiding memory latency because it only requires one additional run-ahead thread.

Narrow SIMD units do not improve performance in most cases. The problem is that manycores already have high compute density, and the bottleneck is memory

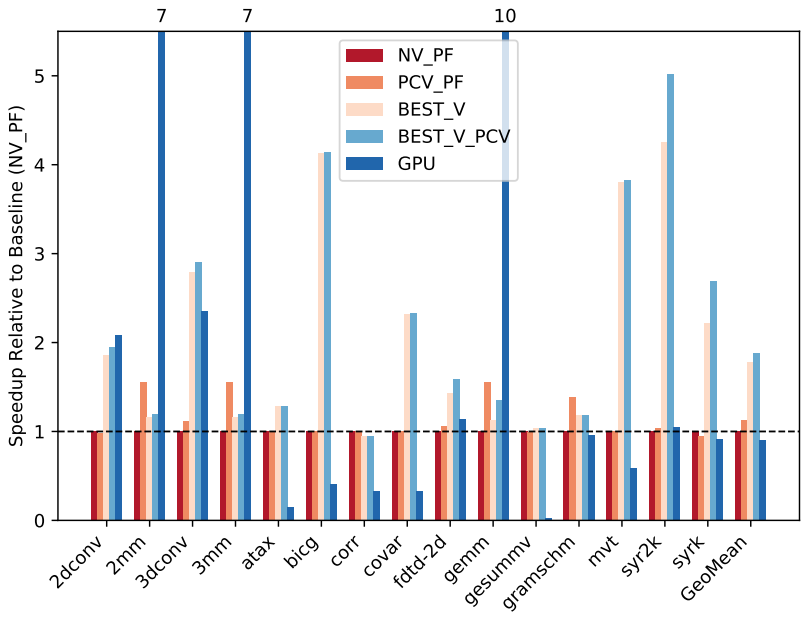


Figure 2.16: SIMD Units: Speedup relative to NV baseline.

bandwidth—so adding SIMD units does not help. As with the GPU, compute-bound kernels such as 2mm, 3mm, and gemm see some benefit. However, most kernels are memory bound and SIMD units exacerbate the issue.

We also experiment with adding SIMD to Rockcress vector groups, so that RISC-V vector instructions are forwarded on the inet. This design point adjusts the vector length by a coarser amount per added core. SIMD units composed in vector groups also have a negligible impact on performance.

Figures 2.17 and 2.18 compare the I-cache accesses and energy consumption of SIMD units and vector groups. All configurations significantly reduce the number of I-cache accesses, but the energy reduction per vector length is superior for SIMD units. Vector groups only amortize fetch energy whereas SIMD instructions amortize all of the pipeline energy besides the functional units and register file. SIMD units composed within vector groups do not significantly impact the total energy consumption because it is already well amortized.

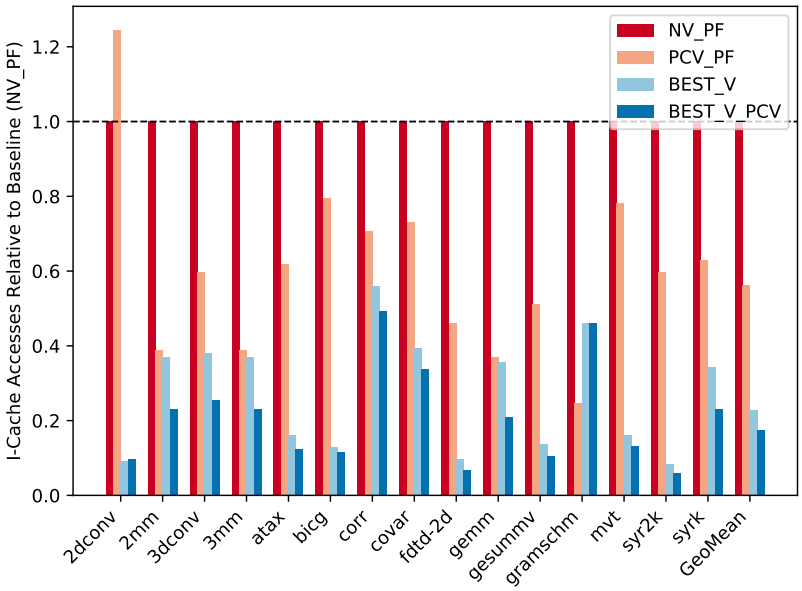


Figure 2.17: SIMD Units: I-cache accesses.

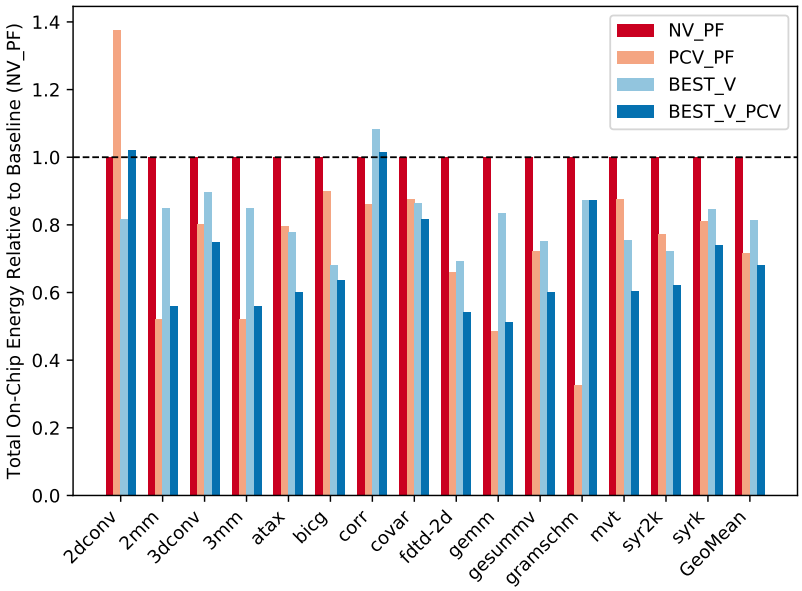
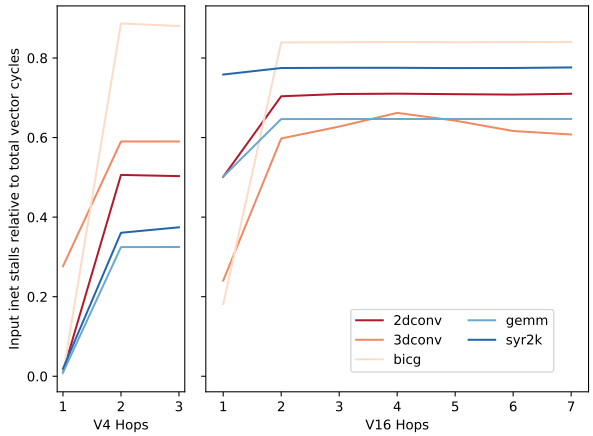
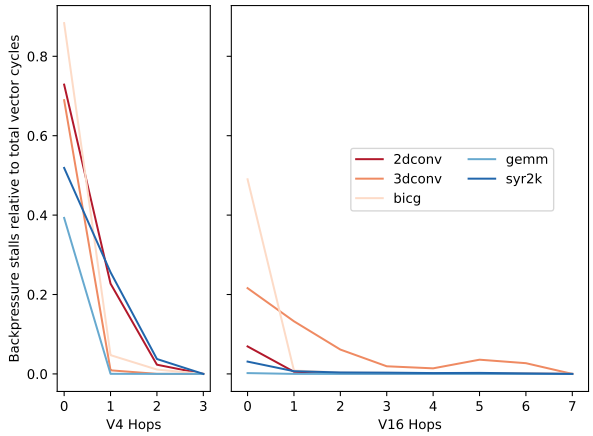


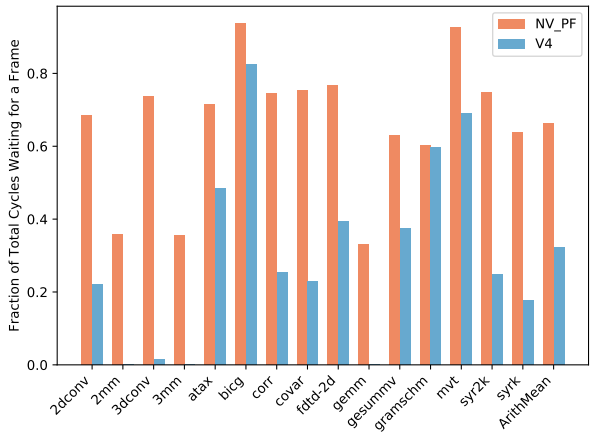
Figure 2.18: SIMD Units: Total on-chip energy.



(a) Input inet stalls.



(b) Backpressure stalls.



(c) Cycles waiting for frames.

Figure 2.19: Characterization of vector groups. Hops are the traveled inet distance from the scalar core (hop 0 is the scalar core).

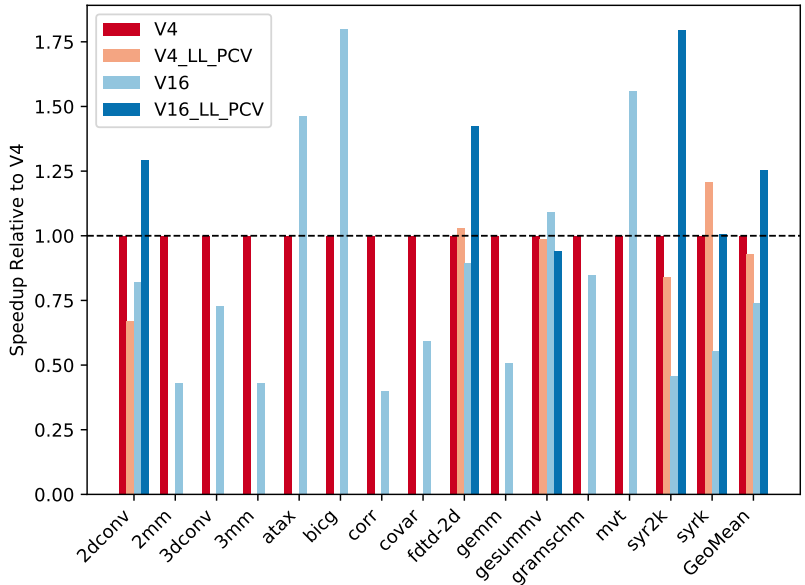


Figure 2.20: Speedup for various vector group configurations.

Vector length flexibility Rockcress can adapt to different optimal vector widths for different applications. The average speedup is 1.1× for V16 alone and 1.5× for V4, but by allowing benchmarks to choose the best among the two options, the mean rises to 1.7×.

Increasing the vector length can help amortize frontend costs (cf Figure 2.11), but it can also make performance worse. There are three main reasons: (1) fewer cores are active in our V16 configuration (see Section 2.7.1), (2) the scalar core falls behind and becomes a bottleneck, (3) there is a longer forwarding network, which could create more opportunities for inet stalling and backpressure. To investigate these possibilities, Figure 2.19a compares the number of cycles the inet input buffer is empty between V4 and V16. The scalar bottleneck is worse in V16, as indicated by the number of inet stalls at hop 1 (the expander core). However, the trend plateaus after two hops, which suggests that all of the inet stalls are generated by the expander core pipeline and these persist through the entire forwarding network. Thus, the scalar bottleneck is the problem rather than the longer forwarding

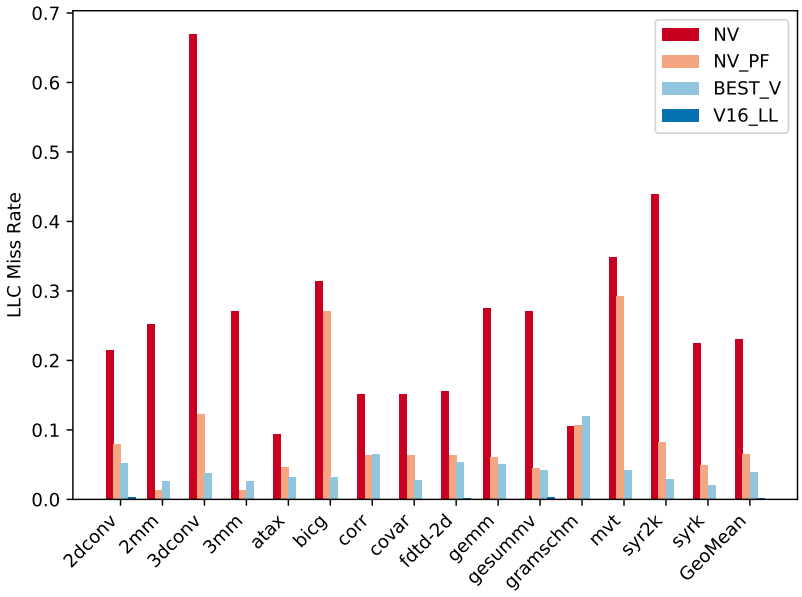


Figure 2.21: LLC stalls.

network: The scalar core has to fetch the data to more cores and cannot keep up with the vector core demand.

Figure 2.19b compares the number of cycles forwarding cores are stalled due to backpressure on the inet. V4 configurations have more backpressure than V16 because microthreads are launched at a slower rate in V16 (scalar bottleneck) and V16 has more buffer space within the group (Section 2.5.2).

The best vector configuration is V16 for atax, bicg, and mvt; it outperforms V4 by 1.5×, 1.8×, and 1.6× respectively. They perform better at V16 because they use group loads. The number of group loads does not increase as vector groups grow (unlike single core loads), so amortization is free.

Long cache lines We experiment with increasing the cache line size to allow for longer vector loads and more request amortization. Without any changes to the existing algorithms, the performance would decrease due to cache thrashing.

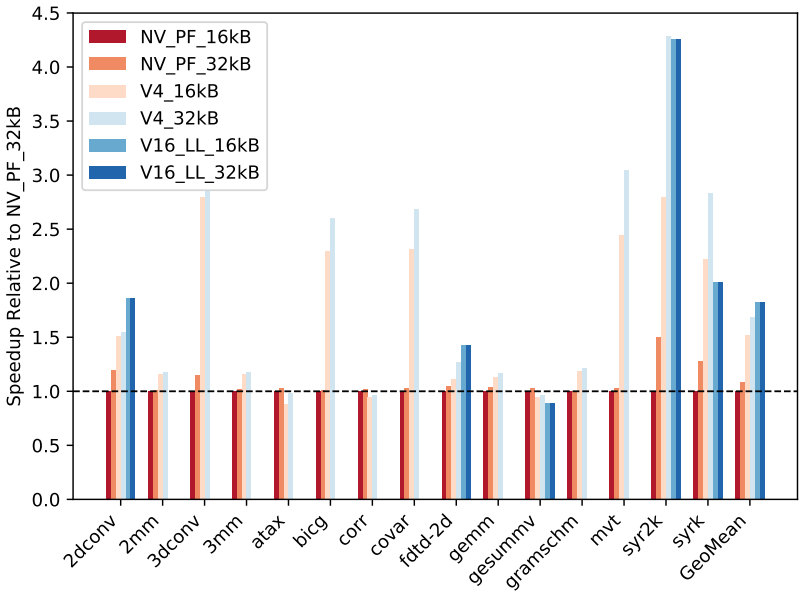


Figure 2.22: Speedup: LLC capacity.

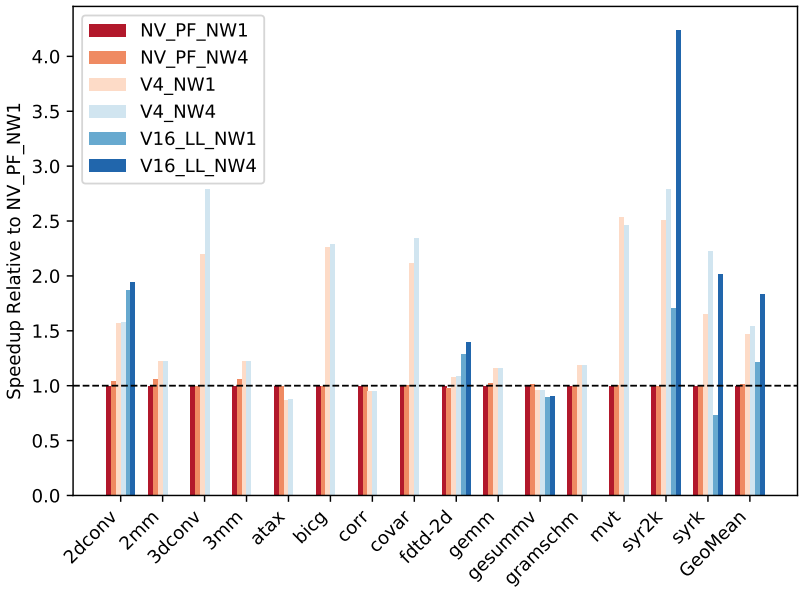


Figure 2.23: Speedup: On-chip network width.

We modify five benchmarks: 2dconv, fdtd-2d, gesummv, syr2k, and syrk with wider loads to take advantage of the longer lines. Group vector loads must be used because larger lines will not fit into a single core's scratchpad. Thus, longer lines are exclusive to vector groups and not as practical for independent cores. We envision a system with reconfigurable cache line sizes [105] to realize long lines for amenable kernels.

Figure 2.20 shows the performance of long lines using a cache line size of 1024 bytes. Long lines can minimize the scalar bottleneck because fewer load instructions are needed per microthread. Long lines also improves the cache hit rate as shown in Figure 2.21. However, the hit rate was already high with the baseline configurations, so the benefits may be limited in this setup. The benchmarks see modest overall improvement over the non-long-line configurations.

Memory system We evaluate our system's sensitivity to the on-chip network width and LLC capacity. Figures 2.22 and 2.23 shows the impact of varying the cache size and network width for various software configurations. Certain benchmarks like syr2k and syrk are very sensitive to the cache size and network width, while most others show little performance improvement. Increasing the cache size for syr2k has a similar effect to using long lines due to the reduction in miss rate. In general, the on-chip network width is not critical to the performance of vector loads and Rockcress could feasibly be designed with a single word-wide network.

Irregular algorithms As an example of an irregular application that would waste a standard vector machine, we measure bfs: a breadth-first graph search. A pure manycore (NV) version of bfs is 2.9× faster than either vector version

(V4, V16). In Rockcress, a single machine can efficiently execute both regular, vector workloads and irregular, graph-like workloads such as `bfs` via run-time configuration.

2.8 Related Work

Two recent vector ISAs, the RISC-V vector extension [2] and ARM's scalable vector extension (SVE) [92], let programs stay agnostic to the hardware's vector length. However, this flexibility is only in the abstraction: the hardware does not change the compute resources it dedicates to a vector engine. We see it as future work to support such a traditional vector ISA using Rockcress's instruction forwarding approach.

Other architectures can dynamically compose multiple small compute engines into one larger machine. Several efforts reconfigure a multicore processor to trade-off single-thread performance with thread-level parallelism (TLP) [38, 46, 56, 98, 113]. These designs do not target SIMD parallelism, and they require tight coupling between the coalesced components. Software-defined vectors need less invasive modifications to cores. The closest work to ours tightly couples a small group of cores to allow a master core to multicast instructions and scalar work [5]. They do no exploit MLP like in our vector DAE scheme and the tight coupling limits the scalability of large vector lengths. The Cray X1 [28] consists of two-wide vector units that can be optionally coalesced in groups of four to form a single eight-wide vector engine. Rockcress achieves a similar effect but permits a more flexible range of vector widths while also offering a completely independent manycore mode. Libra [75] reconfigures a group of PEs to execute SIMD or VLIW instructions

but does not support TLP. Flexible vector lengths have also been proposed for GPUs [49, 60, 81]. The amount of flexibility between MIMD and SIMD is limited due to the centralized controllers and data paths inherent in GPU architectures.

Vector-thread architectures [59, 62] enable a somewhat flexible vector length where each thread can operate independently in MIMD mode or in lockstep SIMD mode to amortize control overhead. Rockcress realizes classic vector-thread architectures as a flexible overlay over a standard manycore machine and provides additional vector length flexibility.

Rockcress's memory system adapts ideas from decoupled access-execute architectures [6, 88] and runahead schemes that use regular cores to implement DAE [17, 93] and applies them to the software-defined vector setting.

2.9 Conclusion

Software-defined vector architectures aim to compete with GPUs for general-purpose parallel programming. By combining a simple MIMD mode and a flexible SIMD mode on the same silicon substrate, architectures can avoid complex hardware thread schedulers and rely on software to choose its own parallelism strategy.

2.10 Future Work

Rockcress was often limited by the scalar core performance. The scalar core could not fetch enough memory fast enough for many vector cores. Solving this bottleneck may improve performance.

Other control signals could be forwarded instead of just the instruction to save more of the pipeline energy. We also did not consider our model using out-of-order core control forwarding. This potentially offers more energy savings, but requires more complex interactions between cores.

Vector length selection was not fully explored in this work. The evaluation only looked at two static vector lengths. The optimal length is not trivial to determine at compile time and many change based on program phase. A dynamic compiler that can monitor and reconfigure the vector length at runtime could find the optimal code and configuration.

Another area of interest was the lack of performance improvement on compute bound kernels. Rockcress does not add any additional functional units or issue slots so no additional performance is expected on these programs. The next chapter explores using reconfiguration to accelerate compute bound programs containing ILP.

CHAPTER 3

REPURPOSING MANYCORE EXECUTE STAGES AS COARSE-GRAINED RECONFIGURABLE ARRAYS

This chapter explores adding ILP execution capabilities to a manycore processor. We map techniques found in traditional CGRAs onto the execution resources of multiple adjacent cores in a manycore. A logical CGRA will be formed from this mapping.

3.1 Background: Coarse-Grained Reconfigurable Arrays

Coarse-Grained Reconfigurable Arrays (CGRAs) are a dataflow architecture that encodes ILP entirely within the instruction schedule [66, 87]. The hardware makes no attempts to schedule instructions like in SIMD or MIMD; it is fully up to software. The hardware consists of functional units connected to neighboring functional units via peer-to-peer/systolic links. The idea is that each functional unit can be working at the same time without control logic interference. When an application can be mapped onto these architectures the energy efficiency and performance is greater than a manycore.

While the potential efficiency of a CGRA is great, it is challenging to map general-purpose applications onto them. These architectures lack native control flow mechanisms and must add complexity to hardware and software to achieve control flow [41]. Additionally, variable latency operations are challenging to handle because timing is built into the instruction schedule. There have been works that try to add hardware to address this [43]. However, CGRAs are still limited in application mapping compared to a manycore, CPU, or GPU.

3.2 Introduction

Exploiting ILP is an important feature in high performance architectures. Many-cores lack a lightweight general-purpose mechanism to accelerate workloads with ILP. The single-issue in-order core pipelines commonly found in manycores [13,20,25,112] have low area footprint but result in low per-core throughput. Out-of-order issue and superscalar pipelines can increase ILP, but these incur a high area and energy overhead due to extra control and datapath logic. VLIW and per-core dataflow execution have less overhead, but either have limited mapping capabilities [15] or require many additional per-core functional units to be effective [35]. The RAW manycore [100] realizes a form of dataflow execution by forwarding values to other cores via logical bypass paths, but this achieves pipeline parallelism and does not increase the peak ILP of a core. In this work, we use the term "dataflow" to refer to CGRA-style execution that aims to increase the peak ILP of the system with little control overhead. Other work has explored stitching together small application specific accelerators across manycore tiles to increase ILP [96], but lacks general purpose programmability.

We propose a manycore architecture that can repurpose its existing functional units and local memories to target workloads with abundant ILP. Concretely, we add the ability for functional units in each core's execute stage to send and receive data between each other (like in DySER [35]), but also to functional units of neighboring cores. This allows the formation of a logical CGRA, which we refer to as a Many eXecute stage Reconfigurable Array (MXRA). An MXRA can map more operations across multiple cores than could be mapped in a single core using either VLIW or a dataflow mini-graph [15]. Our technique also avoids the overhead of adding more functional units necessary in DySER-style dataflow or superscalar processing. We

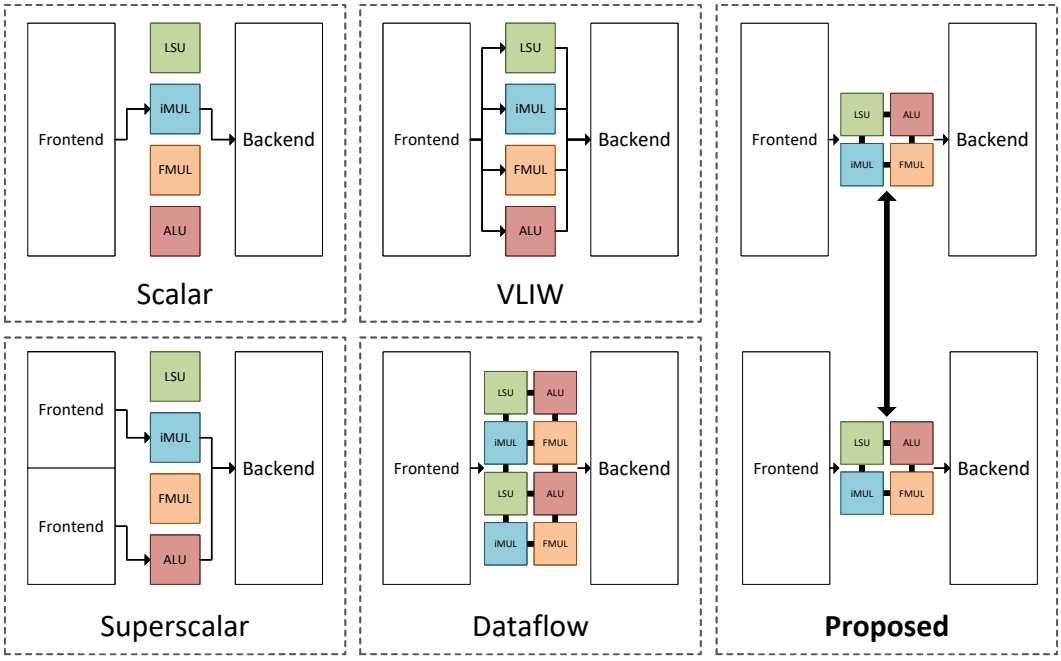


Figure 3.1: A selection of core architectures with varying ILP and control overhead. Our proposed architecture increases ILP over a scalar core without 1) the control overhead of superscalar, 2) the low utilization of VLIW, 3) and the additional functional units of dataflow (i.e., DySER).

compare our approach to a selection of prior approaches in Figure 3.1.

MXRA execution can in theory increase per-core instruction throughput even though it does not add additional functional units. Scalar cores can only issue one operation to a functional unit per cycle even though there are multiple functional units. For example, a core might have four pipelined functional units (IntAlu, IntMul, FpAlu, FpMul), of which three will go underutilized per cycle. MXRA can potentially execute an instruction on each functional unit every cycle without any scheduling overhead. Thus an MXRA can ideally achieve a 4× speedup over a scalar core with the same hardware resources.

MXRA execution is software driven. Multiple instructions are compiled into a dataflow schedule to run on the MXRA and can be invoked by a core at runtime.

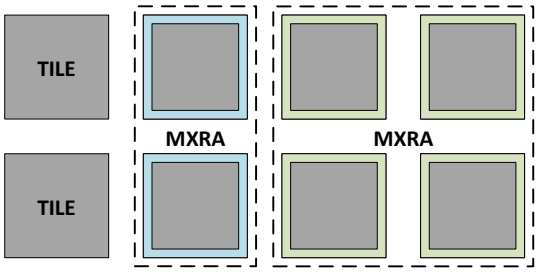


Figure 3.2: A section of a manycore fabric with various configurations. Each core can operate independently or contribute its resources to a MXRA of various sizes.

Once invoked, the MXRA execution is not managed by the core. Each core can still use its functional units that compose the MXRA even while the MXRA is executing, but the core must stall if the MXRA schedule will use that functional unit in the current cycle. This feature allows cores to fetch memory while the MXRA is executing and increase the functional unit utilization even further.

Dataflow architectures need extensive local memory bandwidth to sustain the arithmetic. We do not want to add additional memory ports to the tile's local memory as this would increase area and energy consumption. An MXRA does not need additional memory ports to sustain high ILP because it extends over multiple tiles and can leverage each tile's local memory, logically forming a multi-banked memory.

The number of cores whose functional units compose an MXRA is decided by the programmer at compile time. If the programmer chooses not to form a MXRA, they can use the cores as normal for TLP workloads with little degradation to performance and energy efficiency. Figure 3.2 shows a high level example of different run-time configurations.

Our goal is to preserve the area-normalized performance of the original manycore on TLP workloads while providing speedups to ILP programs that may be ill-suited to the massive level of threads on the manycore. This work resembles previous work like Core Fusion [45], TRIPS [84] and their descendants. However, these works target out-of-order multicores or an array of functional units and are not applicable to manycore processors. Our methods differ in that they aim to augment a standard RISC ISA manycore with minimal hardware changes.

We make the following contributions:

- A hardware mechanism to repurpose manycore functional units and local memories as a logical CGRA.
- A design that overcomes many of the overheads introduced by sharing across cores.
- A dataflow compiler that creates efficient schedules across cores.

3.3 Many Execute Stage Reconfigurable Arrays

Many eXecute stage Reconfigurable Arrays (MXRAs) are an abstraction to repurpose manycore resources to form a logical CGRA. A programmer can specify a group of tiles to form an MXRA, ranging from two tiles to the entire chip. Many MXRAs can exist across the manycore. A single MXRA consists of the functional units and local data memories of the tiles in the defined group. Systolic links are added between each manycore tile to enable functional units to send data to each other. These links are separate from the existing manycore data network. Functional units are also linked together within a core.

While an MXRA can work on various core microarchitectures, we make a few assumptions to give a concrete description of the technique. We envision a tiled manycore architecture where each tile contains a single core and a programmer managed scratchpad. We assume each tile has the following properties.

- Scratchpads support remote stores and remote loads over a mesh data network.
- Each core contains multiple pipelined functional units that support different arithmetic operations.
- MXRAs are latency sensitive and cannot tolerate variable latency operations.

Figure 3.3 shows an MXRA formed from a group of 2×2 tiles. We refer to each tile in the MXRA as a "section".

3.3.1 Formation and Disbandment

Software controls the formation and disbandment of MXRA groups. Each core updates their local CSR (the MXRA CSR) to set group metadata such as size and origin tile for the group. The MXRA can be accessed by performing a special remote store to the scratchpad of the origin tile.

A core can configure the MXRA operation by sending a remote store to the origin tile with a configuration id. Each tile has a small memory containing the MXRA instructions for the given configuration id. These instructions are loaded into a configuration register. The origin tile will also propagate the configuration request to neighboring tiles in the MXRA over a systolic link. Then the neighboring tiles will configure their local section of the MXRA and pass on the request if needed.

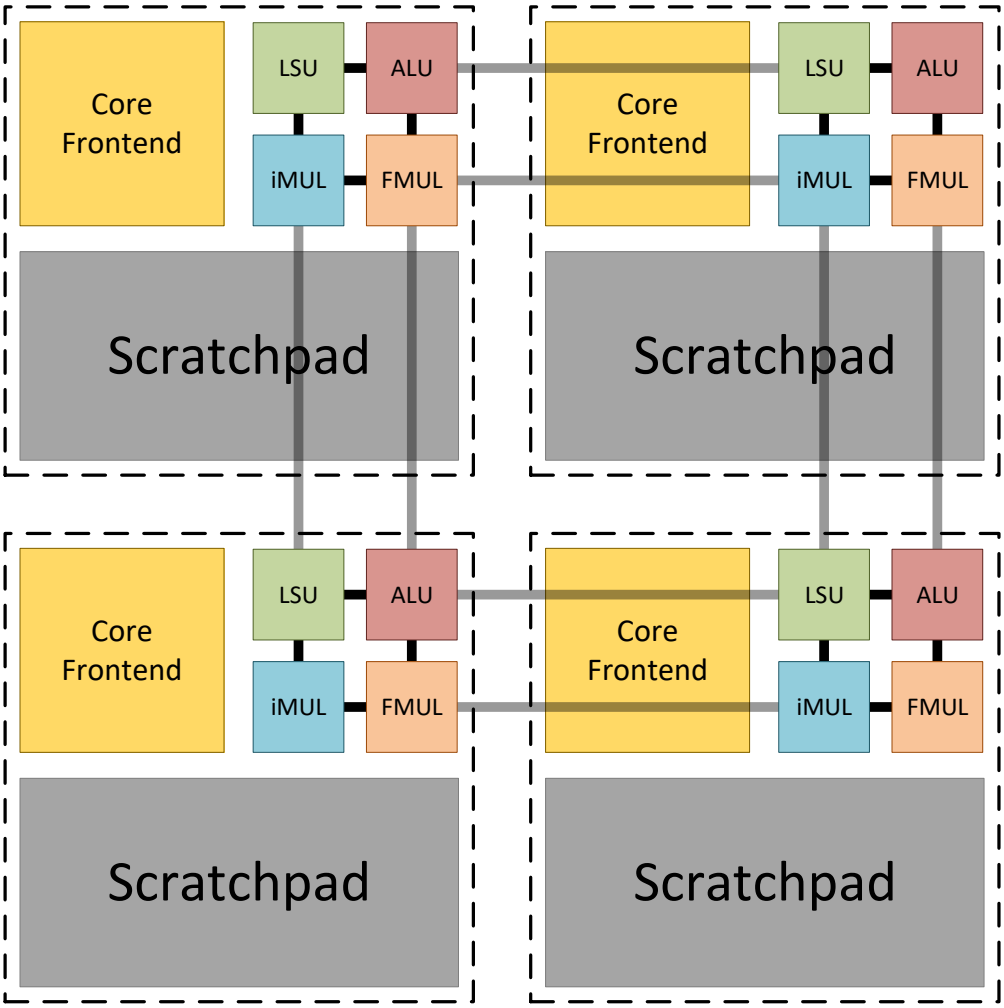


Figure 3.3: A 2×2 MXRA composed of four tiles. The functional units of the core can forward data to adjacent cores in the MXRA group.

The MXRA can be disbanded by updating each local MXRA CSR and no longer sending requests.

3.3.2 Computation

Once a group has been configured, an MXRA can perform computation using the shared resources. Any core can invoke MXRA computation using a special instruction: `cgracomm`. This instruction issues a remote store to the MXRA origin tile and contains pointers to local memory to operate on. The work request will propagate to each neighboring core in the MXRA group. The MXRA will operate on inputs contained in the local scratchpads and write the results back to a local scratchpad.

Cores have no fine-grained control over an active MXRA computation. Upon completion, the MXRA will send a completion message to the core that sent the `cgracomm` request. The core can then retrieve the results from an appropriate scratchpad.

3.3.3 Sharing Conflicts

A single core or multiple cores within an MXRA group can send work to the MXRA. No arbitration is needed if only a single core uses the MXRA, but then the other cores in the group may go underutilized. If multiple cores would like to use the MXRA, they must take turns. We use a simple round-robin approach where a core can only submit work to the MXRA on their turn. The intuition is that each core should not have significant performance skew in the steady state due to lack of

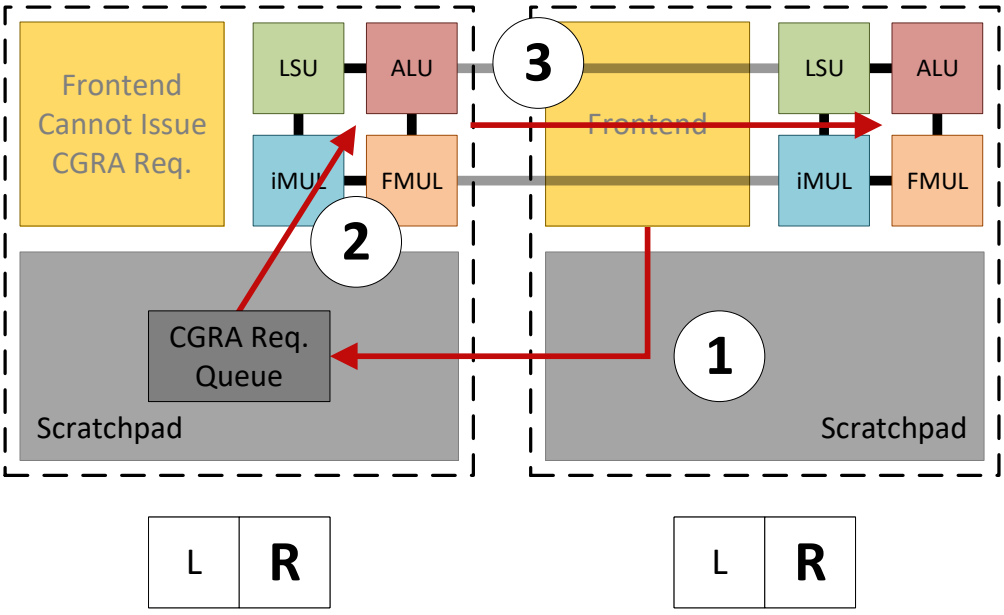


Figure 3.4: The process of MXRA arbitration. An internal counter keeps track of which core can access the MXRA (the right core in this example). 1) The right core submits a request to the origin tile (the left tile) in the MXRA group. 2) The request arrives and is dequeued to initiate one or more MXRA iterations. 3) The work request is propagated to the next section of the MXRA.

local caching and small speculation windows.

A counter in each section of the MXRA tracks which core can access the MXRA. A core will read their local counter to determine whether they can send a `cgracomm` instruction. This counter is updated when the active core sends a remote store to inform the MXRA that it is done. Cores can also opt to remove themselves from the round-robin pool entirely if they no longer have any work to do.

Figure 3.4 shows an example process of cores sharing an MXRA.

3.3.4 Multiple Functional Units Per Core

Scalar core pipelines typically contain multiple independent functional units: ALU, memory units, and floating-point units. Functional units are physically distinct due to differences in datapath and latency. An MXRA can leverage all of these units for computation. We link together all local functional units and add systolic links between the functional units of neighboring cores.

The peak performance improvement of an MXRA is equal to the number of independent functional units per core. Each scalar core can only use one of their functional units per cycle, but an MXRA can in theory use all of them in parallel. In practice, dataflow compilers rarely can use all of the available execution resources due to spatial mapping complexities. A single core MXRA is especially challenging to map onto due to limited resources, which motivates the need to aggregate execution resources across groups of neighboring cores.

3.3.5 Functional Unit Conflicts

Both the core frontend and the MXRA want to use the functional units within the core's pipeline. However, a functional unit cannot be used by both the core frontend and MXRA in the same cycle. The MXRA gets priority because its schedule is latency sensitive. The core frontend will stall if the MXRA is using the same functional unit. Certain instructions like `cgracomm` and global memory requests have a conflict free path to allow for better throughput.

3.3.6 Local Memory

The MXRA can only read from and write to a local scratchpad to avoid variable latency memory operations over the mesh data network. *If an MXRA memory access occurs in a specific tile, it can only access the scratchpad of that tile.* An MXRA can access more local scratchpads if it contains more tiles. Repurposing local ports towards a single computation is a key feature of MXRA execution. The alternative is to add additional physical memory ports, but this is expensive in terms of area. We use CACTI 6.5 [67] to estimate the area of a 4kB scratchpad at 32nm. We find that doubling the number of memory ports more than doubles the area of the scratchpad. This is significant considering the scratchpad can be up to 28% of a manycore tile's area [82].

While an MXRA can utilize the scratchpad bandwidth and capacity of multiple tiles during execution, these scratchpads are physically disjoint. Memory for the computation must be distributed among the cores comprising the MXRA according to the compiled dataflow schedule. A programmer can distribute memory with either remote stores or direct memory fetch into a remote scratchpad (see Section 3.4.6).

3.3.7 Group Sizing

The programmer can determine the optimal size of an MXRA group. Typically, larger sizes result in larger overheads from sharing conflicts and distribution of memory for the computation. However, smaller sizes are often unable to map a large enough sub-computation of the program.

3.4 The Watercress Architecture

The Watercress architecture implements the MXRA abstraction on a manycore architecture based on Celerity [25]. Watercress consists of a mesh of tiles each containing a scalar in-order core with integer and floating point support, a local scratchpad, and a local I-Cache. We describe how Watercress implements MXRAs using these standard manycore components. We also explore how to mitigate overheads required for MXRA execution.

3.4.1 MXRA design

There is no physical CGRA in our design; rather it is embedded across multiple tiles. However, extra configuration and data registers are required to enable efficient execution. We add this extra hardware to each tile with direct access to the core functional units. This section describes how the CGRA is designed as if it was standalone for simplicity.

The CGRA is latency sensitive and cannot tolerate variable latency operations. Memory accesses can only go to the local fixed latency scratchpad.

The CGRA is heterogeneous. Each node in the array is a different arithmetic unit (e.g., integer ALU, floating point multiplier). Unlike homogenous CGRAs, where every node is capable of any operation, the compiler is constrained to map every operation type onto its corresponding CGRA node type.

Each node in the CGRA consists of a functional unit, two or three input registers, four bypass registers, four configuration registers, and link multiplexers [97]. The input registers allow data to be buffered at a node for more than one cycle. The

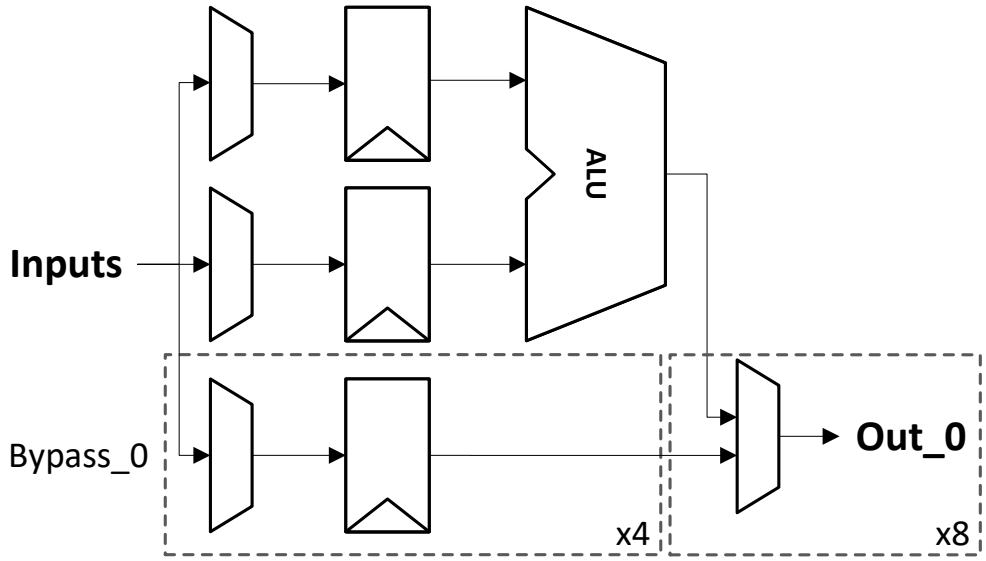


Figure 3.5: Node microarchitecture. A node can process input data in its arithmetic unit and bypass input data. There are four bypass paths and eight output directions to send data to.

bypass registers allow data operands to go around a functional unit without using it. The configuration registers allow context switching between up to four operations in order. We also allow nodes to send and receive data from one or two nodes away in each cardinal direction. There is a total of eight input directions and eight output directions. Figure 3.5 shows what a standalone node looks like.

We integrate this design into a scalar core pipeline mainly by adding the registers and muxes as outlined above to each functional unit. In addition, we also add the following components to each tile.

- A context memory for the local functional units and a counter to cycle through them.
- Conflict detection logic in the issue stage to prevent the core frontend from using a functional unit while the MXRA is active.

- A counter to keep track of which core can use the MXRA.
- A queue to contain incoming work for the CGRA.
- Four registers containing the base memory pointers (Section 3.4.3).

3.4.2 Area and Energy Overhead

The input multiplexers and registers added to each functional unit are the main source of area and energy overhead in MXRA execution. A two-input functional unit includes 6 input registers, 6 8-to-1 input muxes, and 8 5-to-1 output muxes. Bypass hardware comes with an added hardware cost, but helps to ease the scheduling constraints of the dataflow compiler (Section 3.5). Torng et al. [103] provide an area and energy breakdown for a comparable CGRA design. Their node design has eight input registers (in the form of queues), five input and output directions, and two bypass paths. They find these features have similar area to an integer ALU and twice the energy consumption.

We estimate the per-core area overhead is 5-10%. The area overhead is estimated to double the per-core functional unit area. The functional units in Celerity take up 4% of the per-core area [82] with the majority of the area dedicated to instruction and data memories. Celerity does not contain an FPU like in Watercress, so Watercress will have higher functional unit area.

We expect MXRA execution to be more energy efficient than using a scalar core pipeline. MXRA execution does consume additional multiplexer energy to use the functional units, but eliminates frontend energy including I-Cache accesses and issue logic. We estimate the I-Cache and issue energy relative to functional unit energy using measurements from the scalar in-order Ariane core [111]. The I-Cache

is four times as big as in Watercress so we reduce the I-Cache energy by half. We find that the I-Cache and issue logic consume 27× and 4.5× the functional unit energy for an ALU and multiply instruction respectively.

3.4.3 Core-MXRA Interface

A core requests MXRA work using a special instruction: `cgracomm`. `cgracomm` contains four pointers to local scratchpad memory (combined into two registers) to reuse the same schedule with different data. We optimize `cgracomm` to reduce the overhead on the requesting core. First, `cgracomm` is non-blocking, so that the core can issue multiple requests without stalling. A special purpose fence is used to check when the work is complete. Second, we allow `cgracomm` to execute multiple iterations per request according to a previously set CSR value. The memory pointers are updated within the dataflow schedule and stored in a special purpose register. Finally, we allow predication for each memory pointer. There are eight total predication bits: four for loads and four for stores. Each bit is associated with a memory pointer. If a load predication bit is set then all loads using that memory pointer will fetch zero, and the store predication bit skips the execution of stores. This reduces communication overhead to reset data in a remote scratchpad.

Figure 3.6 illustrates how a core can control a local MXRA.

3.4.4 Functional Units

Core pipelines contain multiple functional units that are designed to efficiently perform different operations. We incorporate five units into the MXRA: an integer

```
// setup 2x2 group to use schedule 0
SET_CGRA_GROUP(2, 2);
SET_CGRA_CONFIG(0);

// execute N iterations
SET_CGRA_BATCH_CNT(N);
CGRA_COMM(a_ptr, b_ptr, c_ptr, d_ptr);

// wait
CGRA_FENCE();
```

Figure 3.6: The programming interface to use a MXRA.

Table 3.1: MXRA node properties.

Coordinate	**Functional Units**	**Latency**	**Core Shares**
0,0	Load/Store	1	
0,1	IntAlu	1	×
1,0	FpMul	3	×
1,1	IntMul, FpAdd	2	×

ALU, integer multiplier, floating point ALU, floating point multiplier, and load-store unit. The functional units have different latencies and the dataflow scheduler must consider this (Section 3.5.2). We do not make a MXRA node for each functional unit; some functional units are grouped together to share input and output registers and muxes. We create a 2×2 CGRA out of the five functional units. Table 3.1 enumerates the properties of each node. Only functional units with the same latency are grouped together to avoid potential output collisions.

As noted in Section 3.3, the core frontends are still active during MXRA execution. They must stall if they attempt to use a functional unit that the MXRA schedule is using that cycle. However, the core can still issue `cgracomm` and global memory loads freely during MXRA execution, so we create separate load-store unit datapaths for the core and the MXRA usage. The scratchpad data port is still conflicted though and the MXRA will get priority.

3.4.5 Sharing Across Tiles

There is a latency penalty to sending data across cores due to the physical design of the wires [52]. We add a one cycle penalty to send data across tile boundaries into another core. This will encourage the dataflow compiler to do as many local operations as possible before traveling to another node.

The other complication of this latency is starting iterations. The core will send a `cgracomm` request to the origin MXRA tile, but this request does not reach the other MXRA tiles in the same cycle. We modify the compiler (Section 3.5.2) to account for this startup delay in the schedule.

3.4.6 Memory Distribution

Data must be in local scratchpads before the MXRA execution begins. A core could fetch memory into its own local scratchpad and have the MXRA read from it. However, this creates challenges because each MXRA section can only read from the scratchpad in that section; if the data is only in one tile, other tile memories would remain unused. There is both more memory bandwidth and capacity available if all of the scratchpads are integrated into the MXRA schedule. A possible method to utilize these remote scratchpads is to copy memory from a core's local scratchpad into a remote one, but this can create significant instruction overhead and latency.

Instead, we use a slightly modified version of frames (Section 2.3.3) that allows for asynchronous memory fetches into other cores scratchpads directly from memory (fine-grained DMA). We first describe the single core version of frames and will show how the concept can be expanded to enable efficient MXRA execution. Frames

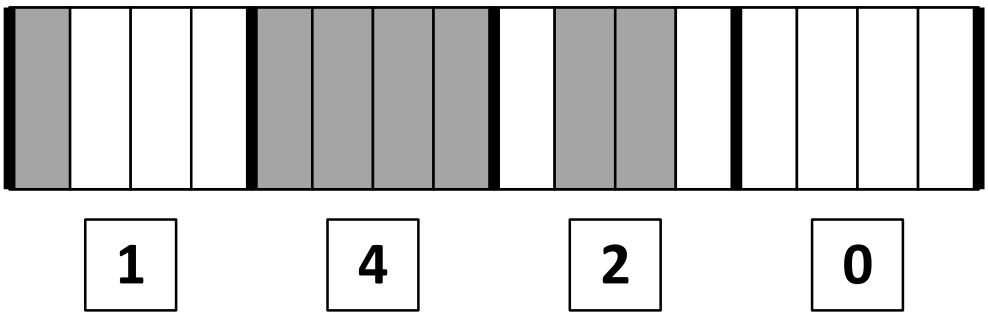


Figure 3.7: An example state of a scratchpad using frames. The scratchpad section shown contains four frames of four words each. Data written into a range bound by a frame is not tracked on a per-word granularity, but rather a per-frame granularity with a small counter. When the counter of a frame reaches four, the core pipeline can proceed.

provide a low-cost way to wait for a pre-configured amount of data to arrive in a local scratchpad. First, we configure the number of words in a frame. Then we fetch that many words from global memory (these requests do not need to be tracked). The frame has a register that will count how many responses have arrived in the section of the scratchpad. We can use an instruction to query whether a frame has received the set number of words. When used to track local memory fetch, frames can improve memory-level parallelism (MLP). Figure 3.7 shows an example of a scratchpad using frames.

Frames can also assist in MXRA execution with little modification. The frame counter will increase regardless of which core initiated the request for memory. We can have one core fetch data and direct it to go into a frame of a remote scratchpad. When the frame counter of the remote scratchpad reaches a configured value, we can have it send a message to the equivalent frame in the sending core. This message will update the frame count in the sending core's local scratchpad, which will inform it that the memory arrived in the remote scratchpad. Figure 3.8

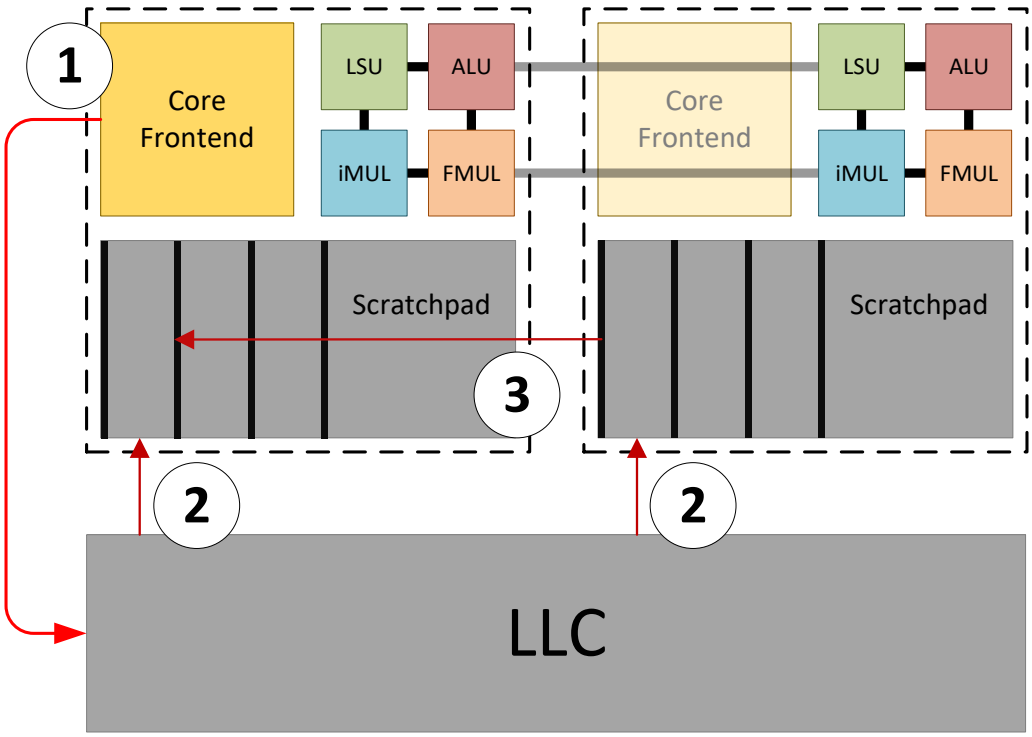


Figure 3.8: The process of remote frame access. 1) The left core requests data for its next MXRA compuation. 2) The LLC responds with data to both the left core and the right core (the left core specified this in the request). 3) When the right core's frame is filled it automatically sends a message to the frame in the left core.

illustrates the process.

Figure 3.9 shows how the programmer can use frames to exploit MLP and fetch data into remote scratchpads. Each core in the example executes the same program, but with a different frame offset to hold its own working set across both memories.

3.4.7 Memory Collection

The MXRA must write output data to a scratchpad to be collected by a core. We consider having the MXRA remote store data back to the scratchpad of the core

```
// N elements to core 0 at frame F
FETCH_DATA(a_ptr, F, N, 0);
// N elements to core 1 at frame F
FETCH_DATA(b_ptr, F, N, 1);

// wait for frame F to receive N + 1 messages
WAIT_FRAME(F, N + 1);

// execute the schedule using data in frame F
CGRA_COMM(F, F, 0, 0);
```

Figure 3.9: Marshalling data in a two core MXRA.

who sent the work request. However, the MXRA cannot always send a remote store as the local network may be congested at the time and might need to stall (even with buffering). This creates issues for the simple MXRA control logic. Instead, the MXRA locally stores data back to an arbitrary scratchpad. Cores will remote load the computed data from this scratchpad after the MXRA computation is complete.

3.5 MXRA Compilation

A dataflow compiler is used to generate schedules for the MXRA at compile time.

3.5.1 Writing a Kernel

We write MXRA kernels as regular C functions that are compiled separately from the core code. Each memory pointer can hold additional metadata to optimize the kernel. The metadata includes:

- A tag to identify which memory addresses to associate with the metadata.

- The scratchpad(s) to map which memory addresses to. This is mainly used to spread out data and improve local memory bandwidth.
- A flag to compact the addresses. If there are four memory addresses spread across four scratchpads, the local address to each scratchpad will be the same if compacted.
- The stride of the memory pointer on each successive iteration of the schedule (Section 3.4.3).

Correct usage of this metadata is critical to the performance of the MXRA. For example, distributing loads and stores to different scratchpads within the MXRA is critical for fast schedules. If data is only mapped to a single scratchpad, then all memory operations will have to be serialized and it may not be possible to fit the kernel within four contexts.

The stride metadata will generate operations to compute the next memory offset based on the last pointer for the given tag. Address calculations occur at the beginning of the dataflow schedule. However, stores happen at the end of the schedule, and it is hard to buffer the address in the MXRA. We resolve this by appending buffer operations, and is currently implemented as an fadd with a constant of zero.

We do not consider kernels that have loops– all of the kernel is unrolled. Data is not passed between successive iterations of the kernel. Reductions are done in the core after the MXRA kernel finishes.

3.5.2 Compiler

We modify an open-source CGRA compiler [97] to target Watercress. The compiler maps a dataflow graph (DFG) of operations onto MXRA nodes and does routing between dependencies. A greedy algorithm places each operation onto the best currently available node based on a cost model. We add randomness to the compiler to perturb the cost model and generate different mappings. In total, we evaluate 65 seeds and choose the best generated mapping.

The compiler must consider the following structural contraints.

- Each section of the MXRA does up to one memory access per cycle.
- Only a single bypass connection or functional unit can write to an output link.
- Only four bypass connections are allowed per cycle.
- Certain nodes can only support specific operations.
- Output latency of different functional units.

The compiler treats the entire MXRA as a monolithic CGRA and initially generates a single schedule spanning multiple MXRA sections. The specific schedules for each MXRA section are cut out of the monolithic schedule in a post processing phase. To achieve this simplification, the compiler must account for the latency to move data between neighboring MXRA sections. There is a one cycle delay for both operand transmission and starting MXRA execution in a section.

We model operand transmission latency with imaginary MXRA nodes between each physical section of the MXRA. These imaginary nodes only support bypass

operations and force the compiler to adhere to a one cycle latency to transmit data. Imaginary nodes are discarded when the MXRA section schedule is cut out of the full schedule.

MXRA execution starts on a different cycle depending on the section. The compiler can "bake" in this startup delay and avoid hardware mechanisms needed to synchronize the schedule start. We must add an additional constraint that all root operations in the DFG (no dependencies) cannot be scheduled within the startup latency. For example, tile 3 is 2 hops away from the origin, tile 0. Tile 3 cannot have any root operations begin in cycle 0 or 1. Non-root operations (have nonzero dependencies) have no restriction on scheduling because this latency is modeled by the imaginary nodes discussed above. When the MXRA section schedule is cut out from the full schedule, the operation times are shifted forward by the section's startup latency.

Finally, the compiler determines what is the last operation to run on each section of the MXRA to determine when the execution has completed.

3.6 Experimental Setup

We model a baseline RISC-V manycore machine, the MXRA extensions, and private per-core CGRAs using the gem5 cycle-level simulation infrastructure [12].

Table 3.2: Microarchitectural parameters for the model.

Component	**Setting**
Cores	64
Int. ALU Latency	1
Int. MUL Latency	2
Divide Latency	20
FP ALU Latency	2
FP MUL Latency	3
Load Queue Entries	8
Share Latency	1 Cycle
MXRA Contexts	4
MXRA Section	2x2 FUs
MXRA I/O Directions	8
MXRA Bypass Paths	4
I-Cache Capacity	4kB
I-Cache Hit Latency	1 Cycle
I-Cache Ways	2
Spm Ports	1RW
Spm Capacity	4kB
Spm Hit Latency	1 Cycle
Router Hop Latency	1
On-Chip Net Width	2 words
Cache line Size	64 bytes
LLC Capacity	256kB
LLC Banks	16
LLC Hit Latency	1 Cycle
LLC Ways	4
DRAM Latency	60ns
DRAM Bandwidth	64GB/s

3.6.1 Manycore & Watercress

We model a baseline manycore resembling Celerity [25] similar to the baseline in Chapter 2. We construct a mesh of tiles each containing a scratchpad, I-Cache, and 8-stage core with in-order issue and out-of-order writeback [95]. The tiles are connected using the Garnet2.0 mesh NOC. At the top and bottom of each column are LLC slices modeled using the Ruby memory system. The LLC slices are connected to a fixed-latency and fixed-bandwidth DRAM model. Table 3.2 enumerates the modeled system parameters. We assume a 1GHz clock frequency for the cycle latencies.

Each LLC slice can contain a disjoint set of addresses and thus does not require cache coherence. The LLC is write-back, pseudo-LRU replacement with 64-byte lines. A core can request a full cache line (16 words) with a single request. This request will stream responses back serially over multiple cycles. The modeled mesh network is two words wide, so a hit of a wide request will take eight cycles to process.

We assume each core contains an integer ALU, integer multiplier, floating point ALU, floating point multiplier, and a load store unit. We assume a 1RW port local scratchpad. Multiple sources may try to access the scratchpad in the same cycle. We prioritize the requests in the following order: 1) Local MXRA requests, 2) Remote requests, and 3) Local core requests.

We model the MXRA as a standalone CGRA in each tile. The compiler enforces the heterogeneity of each CGRA node according to Table 3.1. Each local CGRA can send and receive data from the local scratchpad and from CGRAs in neighboring tiles. The core pipeline is modified to check for functional unit conflicts when the

local CGRA is running. There are four contexts available for each MXRA node, which limits the II to four.

We compare MXRA to private per-core CGRAs. These CGRAs are designed to have aggressive throughput with no regard for area or energy efficiency. This design is meant to be a limit study on the benefit of additional ILP mechanisms in a manycore. It is not meant to be a realistic design that we should hope to outperform.

Each node in the per-core CGRA has a similar architecture to an MXRA node (Section 3.4.1). The CGRA does not reuse the functional units in the core and each functional unit has a one cycle latency. Each CGRA is 4×4 nodes and each node supports any operation. We also provide 3 additional memory ports dedicated to the CGRA and leave 1 memory port dedicated to the local core (for a total of 4RW ports). Additional memory ports alone will have a significant area overhead on the overall tile as discussed in Section 3.3.6. However, the CGRA would hardly be able to map any kernels with a single memory port. We also add an extra piece of logic to calculate the base memory addresses for each iteration (rather than map these operations on the CGRA).

3.6.2 Benchmarks

We select an assortment of benchmarks from Polybench [36] and Embench [19] that contain a high degree of ILP. We do not consider any kernels with fine-grain control flow. Loop management is done on the RISC-V cores. Table 3.3 gives descriptions of each studied benchmark and Table 3.4 describes their optimizations and kernel mappings. cholesky, gemm, and syrk use the same dataflow kernel, but the code

Table 3.3: Applications used in the evaluation.

Name	Input	Description
conv2d	2048×2048 image	3×3 filter applied to an image
cholesky	256×256 matrix	Cholesky decomposition
fir	400K. 64 Filter Dim	Finite Impulse Response
gemm	256×256 matrix	Matrix mul. ($C = \alpha AB + \beta C$)
gesummv	4096×4096 mat, 4096 vec	Matrix vector ($y = \alpha Ax + \beta Bx$)
syr2k	256×256 matrix	Symmetric Rank-2K Update
syrk	256×256 matrix	Symmetric Rank-K Update

Table 3.4: Application software optimizations.

Name	Algorithm opt.	MXRA kernel size	MXRA cores
conv2d	Column and row reuse	3 elements	2×2
cholesky	Tiled Outer product	2×2 tile	2×2
fir	Reuse	2 elements	2×1
gemm	Tiled Outer product	2×2 tile	2×2
gesummv	Tiled matrix	1×1 tile	2×1
syr2k	Tiled Outer product	1×1 tile	2×2
syrk	Tiled Outer product	2×2 tile	2×2

surrounding these kernels are distinct for each benchmark. The size of the MXRA groups were chosen based on the minimum size required to map a well-defined piece of the benchmark and heuristics.

Memory optimizations are the same between all versions, except in uncommon cases as listed.

- The baseline in cholesky is faster without using an outer product matrix multiply.
- The tile size in gemm and syrk varies between 8×8 and 16×16 depending on the best instruction to memory bandwidth tradeoff.
- The tile size in gesummv is 4×16 for MXRA and 8×16 in the baseline because the MXRA is constrained by memory capacity for multiple users.

Table 3.5: Benchmark configurations.

Configuration	CGRA type	Cores Active
MANYCORE	N/A	64
SHARE	MXRA	64
REDUCED SHARE	MXRA	16-32
PRIVATE	4×4 CGRA	64

We compile the C code using GCC 10.1.0 with `-O3` optimization and target the uncompressed RV-G ISA. The CGRA compiler is a custom LLVM 11.1.0 optimization pass. We check correctness using a serial version of each kernel.

3.6.3 Configurations

We run each benchmark under four different configurations. Table 3.5 enumerates the properties of each. MANYCORE uses standard manycore execution. Two different MXRA configurations are studied. SHARE keeps all of the cores in the MXRA active when possible, while REDUCED SHARE only has one core in the MXRA group active. PRIVATE uses aggressive and intentionally unrealistic per-core CGRAs.

3.7 Evaluation

This section uses our cycle-level models to compare manycore, MXRA, and private per-core CGRA execution efficiency.

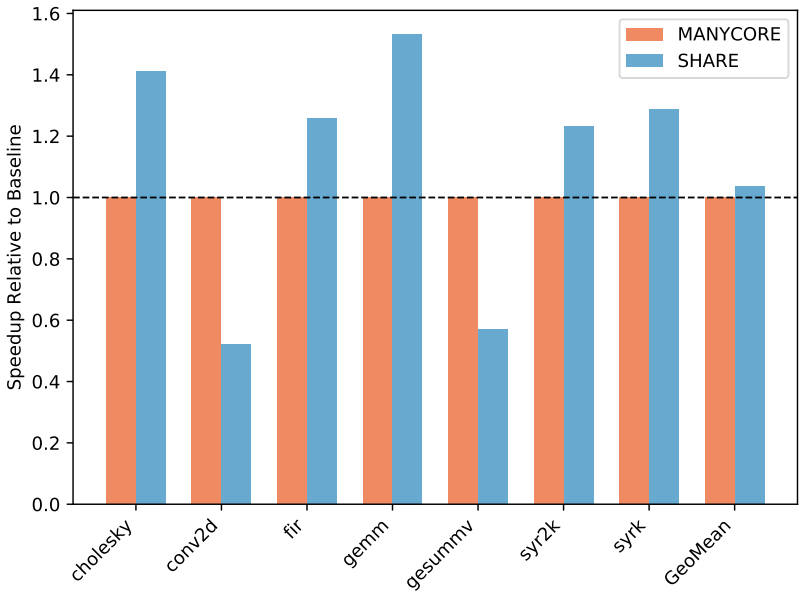


Figure 3.10: Speedup relative to the manycore baseline.

3.7.1 Performance

Figure 3.10 shows the speedup of MXRA over the baseline manycore. Five benchmarks, cholesky, fir, gemm, syr2k, and syrk achieve an average speedup of 1.3× over the baseline with gemm achieving a 1.5× speedup. These speedups exceed the estimated 1.05-1.1× area overhead estimated in Section 3.4.2. Two benchmarks, conv2d and gesummv, are slower than the baseline. Section 3.7.4 explores the characteristics of these benchmarks that explain the low performance. For these benchmarks, the programmer can opt to use the manycore normally rather than a MXRA configuration.

3.7.2 Superscalar

In-order superscalar and VLIW are alternative methods to efficiently improve per-core throughput. Figure 3.11 shows dual-issue and quad-issue in-order super-

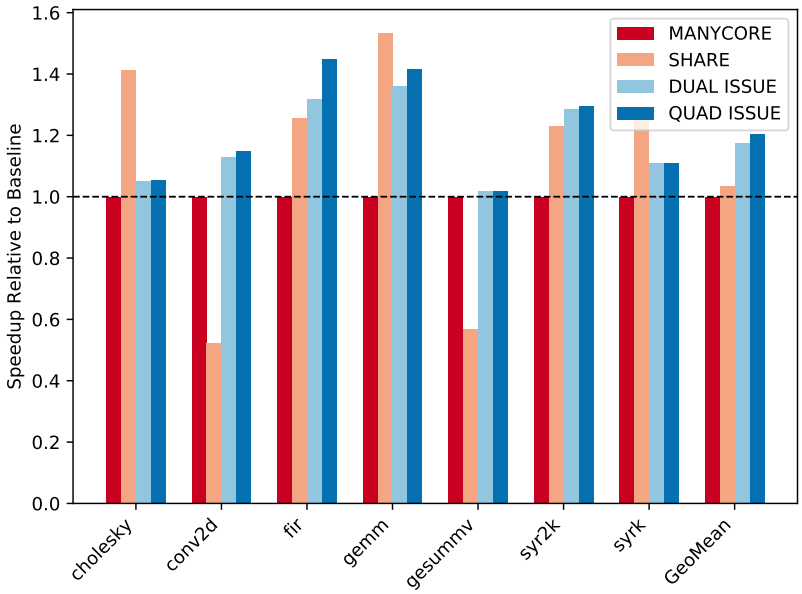


Figure 3.11: Superscalar speedup relative to the scalar manycore baseline.

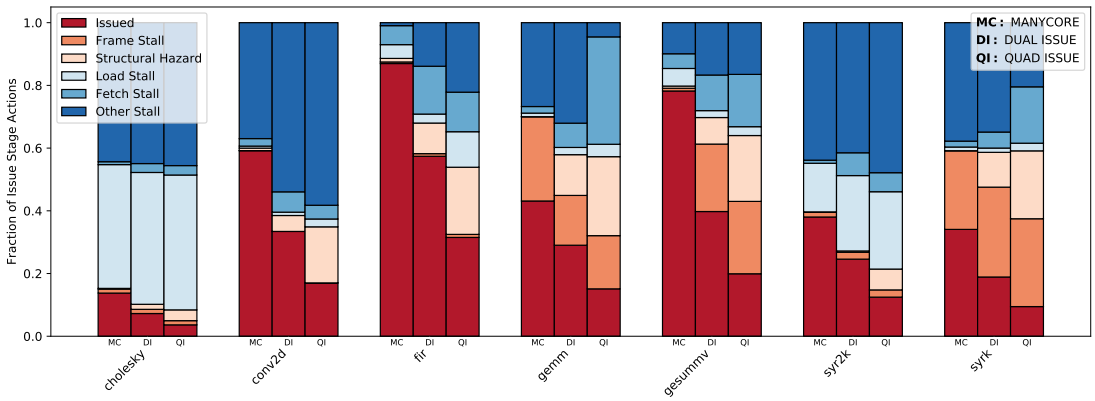


Figure 3.12: Issue stage event breakdown for various core widths.

scalar with the same execution resources as the single-issue baseline. We do not consider VLIW because the performance will be similar to superscalar in the regular benchmarks considered. However, we do use the GCC `-mtune=sifive-7-series` flag to improve instruction scheduling for superscalar cores. Superscalar execution of fir, syr2k, gesummv, and gesummv is slightly faster than the MXRA and scalar baseline. gemm, cholesky, and syrk are slower using superscalar execution than MXRA execution. Quad-issue does not increase performance significantly over dual-issue.

The limited superscalar performance is due to a variety of factors including limited issue windows (i.e., not out-of-order scheduling) and limited functional units. Figure 3.12 shows a breakdown of the event that occurs when the issue stage tries to schedule an instruction. conv2d, fir, gemm, gesummv, and syrk see an increase in structural hazards due to limited functional units and memory ports. cholesky and syr2k see an increase in memory stalls due to increased memory pressure.

While superscalar closes the performance gap between MXRAs and standard core pipelines, MXRA has energy advantages. As noted in Section 3.4.2, even scalar in-order pipelines have significant frontend energy and superscalar will have even more. MXRA execution can eliminate most of this frontend energy. Figure 3.16 illustrates the reduction in I-Cache access due to MXRA execution.

3.7.3 Sharing

There are two ways to use an MXRA: only use one of the cores in the group (REDUCED SHARE) or use all of the cores (SHARE). SHARE requires arbitration and more group memory capacity to hold the working set of each core, but REDUCED

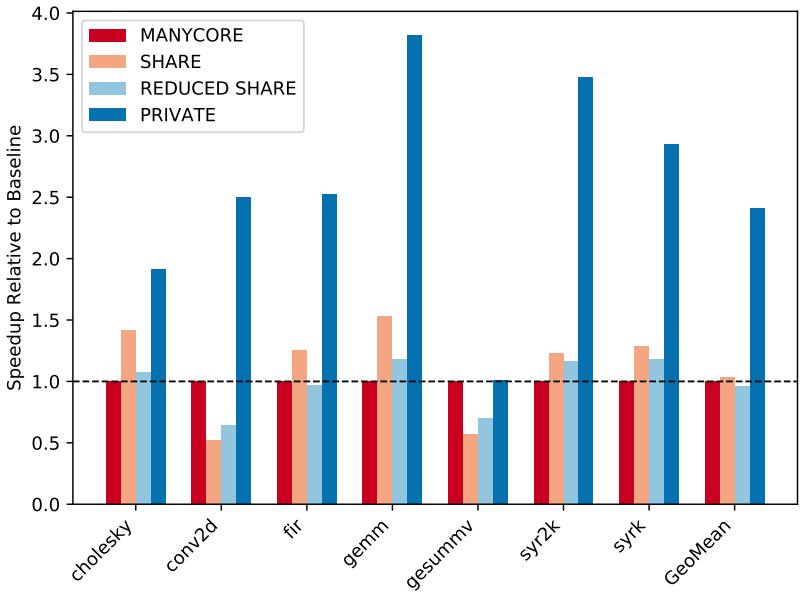


Figure 3.13: Speedup relative to the manycore baseline of every simulated configuration.

SHARE leaves cores underutilized. Figure 3.13 compares SHARE and REDUCED SHARE execution times. SHARE almost always outperforms REDUCED SHARE except in conv2d and gesummv where both MXRA configurations are already inferior to the baseline. For gemm, SHARE is 1.3× faster than REDUCED SHARE. REDUCED SHARE has fewer stalls waiting on the MXRA, but the reduction in cores and TLP is not a favorable tradeoff.

3.7.4 Compute Scalability

This section explores whether the selected benchmarks are well-suited to CGRA execution. Figure 3.13 includes the performance of a large private per-core CGRA with the previous performance results. On average, per-core CGRAs achieve a 2.4× speedup over the manycore baseline. Only gesummv did not perform better with the per-core CGRAs.

Figure 3.14 explores the CPI breakdown of the baseline, SHARE, and the private CGRA.The CPI (the total height of the bars) is higher in the MXRA and CGRA cases, which means the core pipeline performance is worse than in the baseline. However, this is expected because many instructions are offloaded onto the MXRA and CGRA (see Section 3.7.5).

"CGRA Fence" represents the stalls due to the core waiting for CGRA requests to finish. In the CGRA configurations, cores spend most of their time waiting for the CGRA to finish rather than doing work themselves. The CGRA performance is primarily determined by the II of the mapped schedule. The schedule II is typically three or four in the considered benchmarks. There are few functional unit sharing conflicts between the core pipeline and MXRA. This is shown by the "FU Conflict" bar. The core will not utilize functional units if it is mostly waiting on the CGRA to finish.

The CGRA configurations typically result in more stalls due to memory ("Frame Stall"). More compute ability results in a higher demand on memory. Benchmarks like gemm have enough spare memory bandwidth to sustain the additional compute. However, other benchmarks like gesummv become dominated by memory stalls. Figure 3.15 shows the DRAM bandwidth utilized by each benchmark for different configurations. gesummv has almost full DRAM bandwidth saturation and is thus memory bound and not suitable for CGRA acceleration.

conv2d has a high CPI due to waiting on the MXRA to finish ("CGRA Fence") and due to algorithmic mapping (Other stalls). A 3×3 filter cannot be mapped effectively onto a reasonably sized MXRA. Row reuse prevents input rows from being spread out to different scratchpads; they must all exist in the same scratchpad. This makes the dataflow schedule more constrained. The kernel must be split into

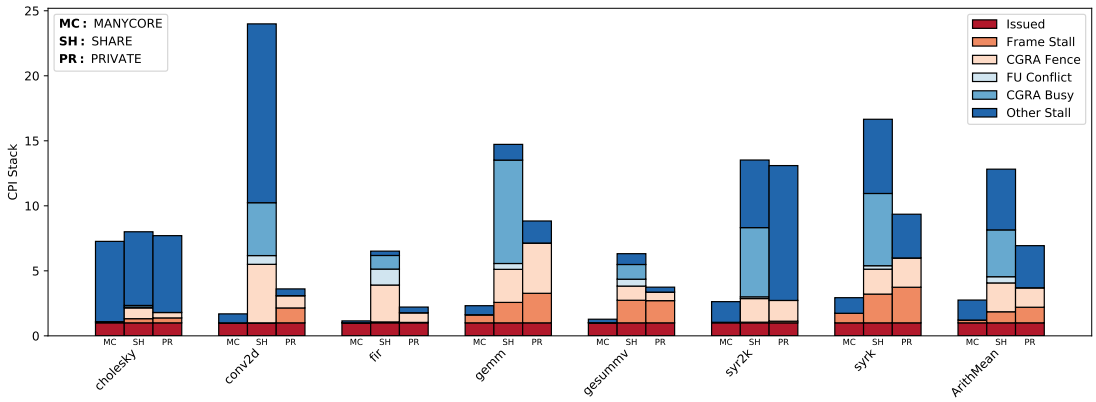


Figure 3.14: A CPI stack showing core pipeline stalls for different configurations. A stacked bar in the CPI stack shows the relative number of cycles where an event occurs in a core's issue stage (normalized to cycles where an instruction was issued). The total stacked bar height indicates the CPI of the core. The MXRA and CGRA configurations have a worse total CPI because much of the computation is offloaded off the core.

three seperate schedules and executed sequentially (each does one row of the filter). Performance is lost due to reconfiguring the MXRA and waiting for pipeline cold start and draining. We also find that a single scratchpad can only hold enough data for two active cores out of the four core MXRA. Core inactivity is represented as "Other Stall" in the CPI stack.

The CPI stack shows the MXRA performance for SHARE which does arbitration between multiple cores using the MXRA. A large portion of cycles are spent waiting for MXRA access due to another core using it ("CGRA Busy"). Even with a large amount of arbitration stalls, using multiple cores in the group is still superior to only using one core pipeline in the MXRA group. cholesky has relatively few stalls due to sharing contention. MXRA access in cholesky is more frequent, but for a shorter amount of time than other benchmarks like gemm.

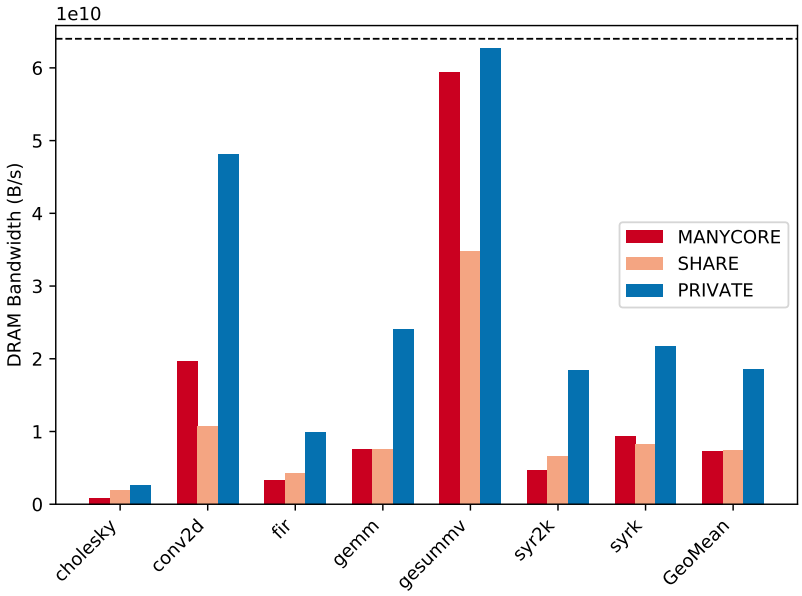


Figure 3.15: DRAM Bandwidth Usage.

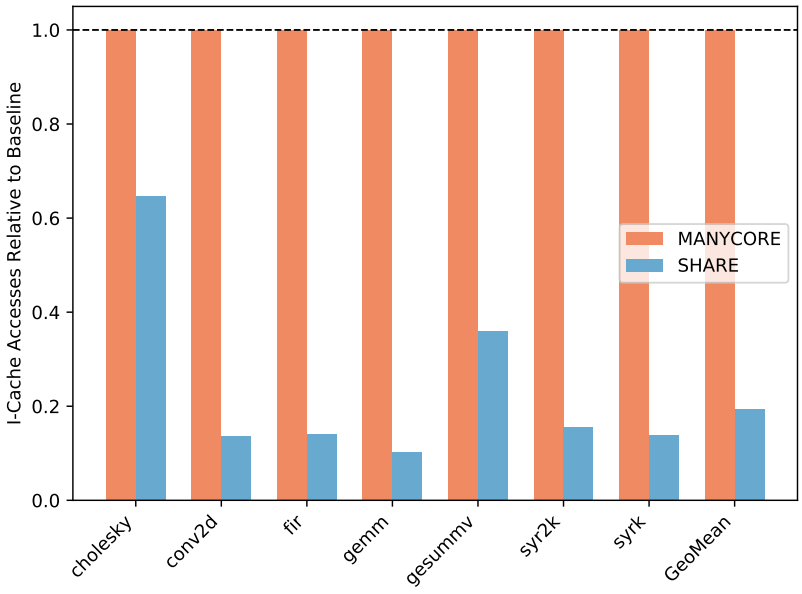


Figure 3.16: I-Cache accesses.

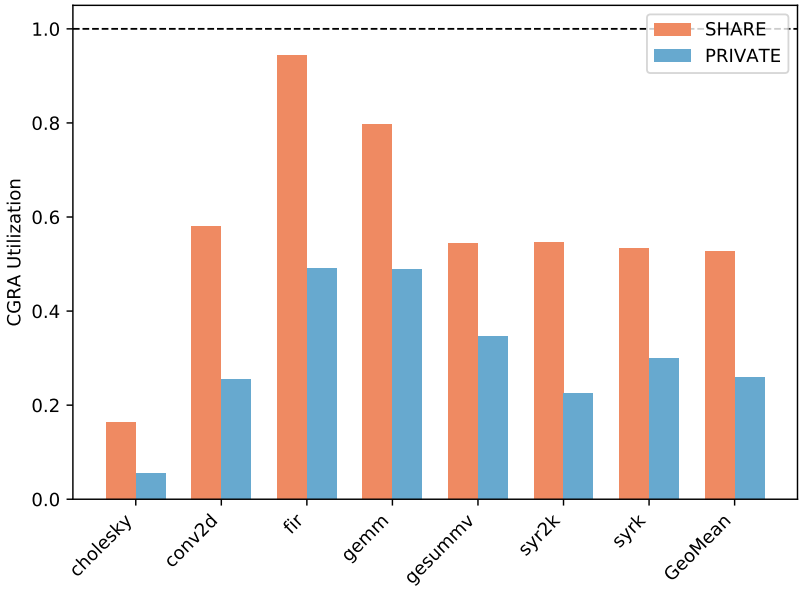


Figure 3.17: CGRA Utilization.

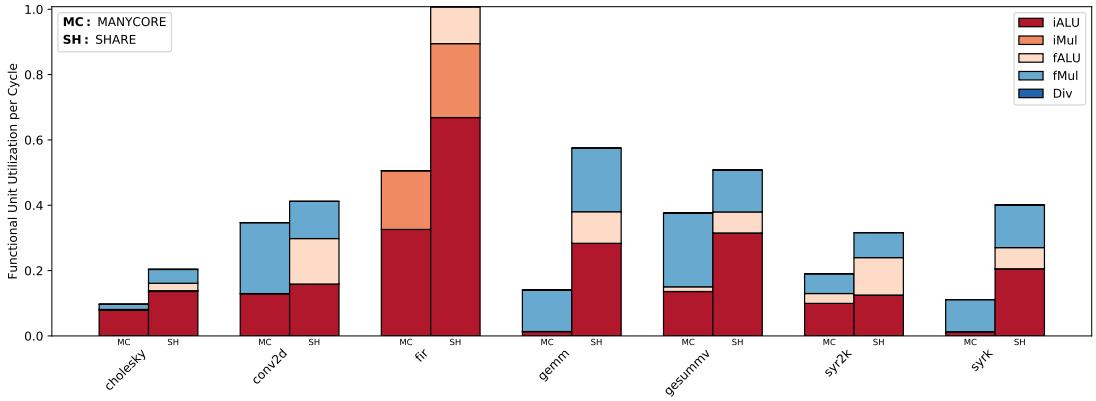


Figure 3.18: Functional unit usage for the baseline manycore and MXRA configuration. Each bar represents the usage of a different functional unit per cycle. The total stacked bar height of the baseline manycore is the IPC minus memory instructions.

3.7.5 MXRA Utilization

Each MXRA must be well utilized to achieve speedup (Amdahl's Law). Figure 3.16 compares the I-Cache usage between the baseline manycore and a MXRA configuration. MXRA configurations reduce the I-Cache usage by 5.3× over the baseline.

The operations encoded by the removed instructions are executed on the MXRA rather than the core. cholesky has the least reduction in I-Cache usage. This is because the baseline version uses a different algorithm with less instruction overhead, but worse memory utilization. The baseline version is faster under this tradeoff, but the MXRA is slower so it uses another algorithm.

Figure 3.17 shows the utilization of the MXRA and private CGRA in their respective configurations. The MXRA is used for 0.53% of the total execution time on average. fir is able to almost fully utilize the MXRA. The utilization of the private CGRA is low (0.26%) which suggests that it is overprovisioned for the system. Even if a program can benefit from a CGRA, the CGRA will be under-utilized and does not justify the performance per area tradeoff. MXRAs seek to solve this under-utilization problem by limiting the additional hardware, but still improving performance through dataflow acceleration.

The main source of speedup from an MXRA is the potential to use more functional units per cycle than a scalar core could. Figure 3.18 compares the functional unit utilization of the baseline manycore and the MXRA. Only the arithmetic units are considered; the load-store unit and scratchpad memory port usage are not shown. Higher local memory utilization will result in higher arithmetic unit utilization, so this plot does represent the excluded units. The MXRA configuration increases the functional unit utilization as expected. The total number of operations executed on each is not always the same. For example, the core instruction stream might use a fmadd whereas the MXRA splits it up into a fmul and fadd. Some of the benchmarks use fmadd when able: cholesky, gemm, syrk, gesummv, but others do not due to mapping challenges. Another artificial difference is the additional fALU usage is partially from using it as a data buffering

operation.

3.7.6 Group Sizing

We study gemm to see how changing the MXRA size affects performance. The reported results used a kernel tile size of 2×2 and map it onto a group of 2×2 cores. A smaller 2×1 tile can be mapped onto a group of 2×1 cores. This results in a slight degradation to performance. A 1×1 tile cannot successfully map onto a single core.

MXRA mapping efficiency is superlinear up to four cores. Address computations can be reused for multiple memory operations in larger schedules. A single core MXRA typically has one address computation per memory operation whereas a larger mapping on a larger MXRA would have a better ratio. Another reason gemm is superlinear is that the same load can be reused for multiple operations in the larger mapped tile. MXRAs larger than four cores have compilation challenges and are not explored further.

3.8 Related Work

DySER [35], BERET [39], and Dataflow Mini-graphs [15] seek to reduce the overhead of superscalar processors by mapping small kernels onto a dataflow functional unit. These works do not explore sharing functional units between cores to reduce hardware overhead and improve mapping efficiency.

Stitch [96] adds small accelerators to each tile in a manycore and allows systolic

arrays to be formed between the accelerators of neighboring tiles. It does not explore general-purpose computation and allows only a single core to use the shared accelerators at a time. Previous work has explored tile based general-purpose reconfigurable fabrics with no cores [26, 76]. While MXRA is also a tile based reconfigurable fabric, a key contribution is the interface with the local cores.

TRIPS [84] reconfigures a sea of processing elements to produce dataflow kernels. TRIPS supports ILP/TLP reconfiguration by forming groups of independent processing elements. However, TRIPS lacks hardened cores and is therefore fundamentally limited in the amount TLP it can express as compared to a manycore. Our work takes the opposite approach: we prioritize TLP by building lightweight dataflow extensions on top of an existing manycore. Pure manycore execution also benefits from using a standard RISC ISA as opposed to the specialized ISA of TRIPS.

Core Fusion [45] and related work [38, 55, 56, 98, 113] tradeoff ILP for TLP by dynamically grouping or splitting core pipeline elements. These works do not explore dataflow execution and target multicore systems rather than tiled manycores.

The RAW manycore [100, 101] allows data operands to be transmitted between cores to improve pipeline parallelism, but does not support CGRA-style dataflow execution. The key difference is that the RAW core frontends are still active and must issue instructions to the functional units. While RAW utilizes functional units across multiple tiles, it cannot increase functional unit utilization past what a single core could achieve. RAW also employs a software-managed scalar operand network [7, 8] that is seperate from the dynamic data network similar to this work. The RAW compiler [61] and StreamIt [33, 34] target the scalar operand network and must distribute operations across tiles. Our compiler does similar tasks, but

has more scheduling constraints due to the latency-sensitivity of a control-minimal CGRA.

Loki [5] can emulate CGRA-style dataflow execution by forwarding operands between cores and repeating the same instruction schedule in a core. Versa [57] uses similar operand forwarding as RAW, but using a systolic network rather than the programmable network in RAW. Similar to RAW, the core frontend still must issue instructions to the functional units and thus cannot increase functional unit utilization. Other work takes a coarser grain approach where each core runs a program on input data and then stores data to a nearby core in a systolic fashion [23, 74]. Rockcress [9] forwards instructions between groups of manycore tiles to exploit DLP, but leaves ILP acceleration unexplored.

Previous work on multicore processors explores sharing a monolithic reconfigurable accelerator over time [86, 90] or spatially [22, 32, 37, 110]. These architectures do not target manycores and do not use the core functional units.

Functional unit sharing between a fixed set of cores is well explored in commercial architectures like AMD Bulldozer [3]. Other work explores issuing remote functional unit usage in 3D stacked cores [14]. These prior works do not explore dataflow execution and flexible sharing between multiple cores.

3.9 Conclusion

By sharing execution resources across manycore tiles, MXRA execution can increase performance for compute bound workloads on manycores. Speedups do not depend on costly control, local memory ports, or additional functional units that are

intractable in a manycore tile.

3.10 Future Work

A major challenge in this work was getting the dataflow compiler to find valid mappings. An improved compiler that uses a more advanced algorithm than greedy search may alleviate this issue. In addition, adopting a more flexible CGRA design like Elastic CGRAs [43] may also improve the compiler success rate.

CHAPTER 4

CONCLUSION

Manycore processors have the potential to change the landscape of high throughput computing. By relying on software to drive performance, manycores avoid the fundamental hardware inefficiencies of modern multicore CPUs and GPUs.

This thesis proposed Rockcress and Watercress, manycore processors that use software reconfiguration to support DLP and ILP respectively. Lightweight hardware extensions enable groups of adjacent tiles to reconfigure into logical accelerators to target various forms of parallelism. Prior manycores with DLP or ILP capabilities either sacrificed TLP by having larger cores or relied mainly on design-extracted parallelism.

Rockcress and Watercress broaden the number of applications that manycores can efficiently execute. These improvements will make manycores more commercially viable and noteworthy to supercomputer designers.

4.1 Vector Engines, Logical Vector Engines, CGRAs, or Logical CGRAs?

This thesis explored accelerating regular workloads which can be accelerated by vector engines, CGRAs, logical vector engines, and logical CGRAs. This section comments on which of these accelerators should be deployed in future manycores.

The logical accelerators were shown to be more area efficient than their physical counterparts and can still achieve speedup over the baseline manycore. As noted in the introduction, per-tile area overhead is expensive in a manycore because it reduces

the number of tiles that can exist on chip. For this reason, logical accelerators are preferred even though the potential speedup of physical accelerators may be higher.

While the prior chapters looked at logical vector units and logical CGRAs seperately, these accelerators can be realized on the same reconfigurable resources. Should both types of acceleration be applied to regular workloads? Logical vector engines improve memory bandwidth utilization with DAE and spatial access locality across cores while logical CGRAs improve functional unit utilization. Both techniques reduce core frontend usage and will improve energy efficiency. It likely does not make sense to compose the two forms of reconfiguration, i.e., have a logical vector of logical CGRA execution because the instruction overhead is amortized enough with just one technique. However, a system could support both forms of reconfiguration and choose between the best one to use at compile time based on the application.

We compare logical vector units and logical CGRAs using the same gem5 model with the same parameters and baseline benchmark implementations as Chapter 3. The baseline manycore is compared to V4 (software defined-vector group size of 4 from Chapter 2) and SHARE which is the MXRA configurations from Chapter 3. There have been algorithmic (i.e., more memory reuse) and model improvements (i.e., more memory bandwidth) to the baseline since the original software-defined vector evaluations. For this reason, the vector configurations are relatively worse compared to the baseline than in Chapter 2.

Figure 4.1 compares conv2d, gemm, and gesummv using software-defined vectors and MXRA execution. Logical CGRAs improve performance in gemm while logical vector units improve performance in conv2d, and neither improve performance in gesummv. gemm is compute-bound so it benefits from more compute provided by a

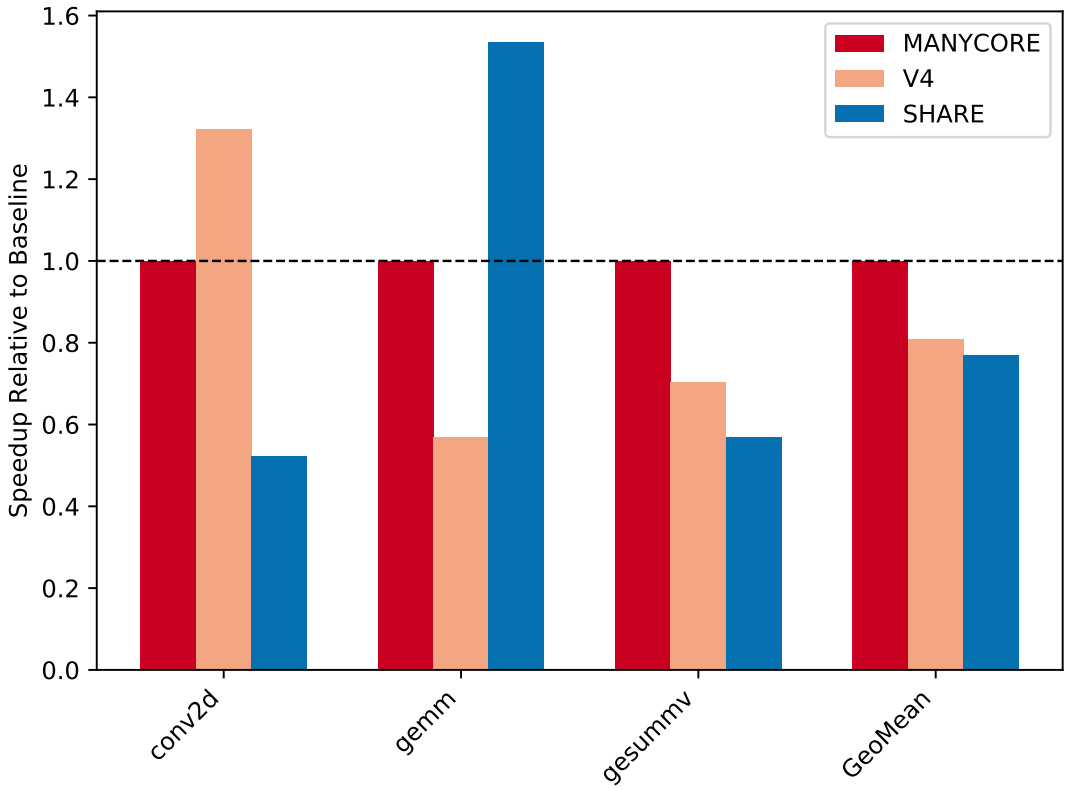


Figure 4.1: Speedup of logical vector units and logical CGRAs against a baseline manycore.

logical CGRA and conv2d is more memory-bound and can benefit from DAE used in a logical vector unit. While all these benchmarks contain regular parallelism the best approach to accelerate them varies.

Figure 4.2 shows the I-Cache access reduction in the two approaches. Both logical vector units and logical CGRAs can significantly reduce I-Cache accesses and improve energy efficiency over a standard manycore.

In summary, both logical vector units and logical CGRAs can accelerate and improve energy efficiency in workloads with regular parallelism, but the best technique to use is workload dependent. Logical vector units have a lower per-core area overhead compared to logical CGRAs, which makes them a safer addition to a

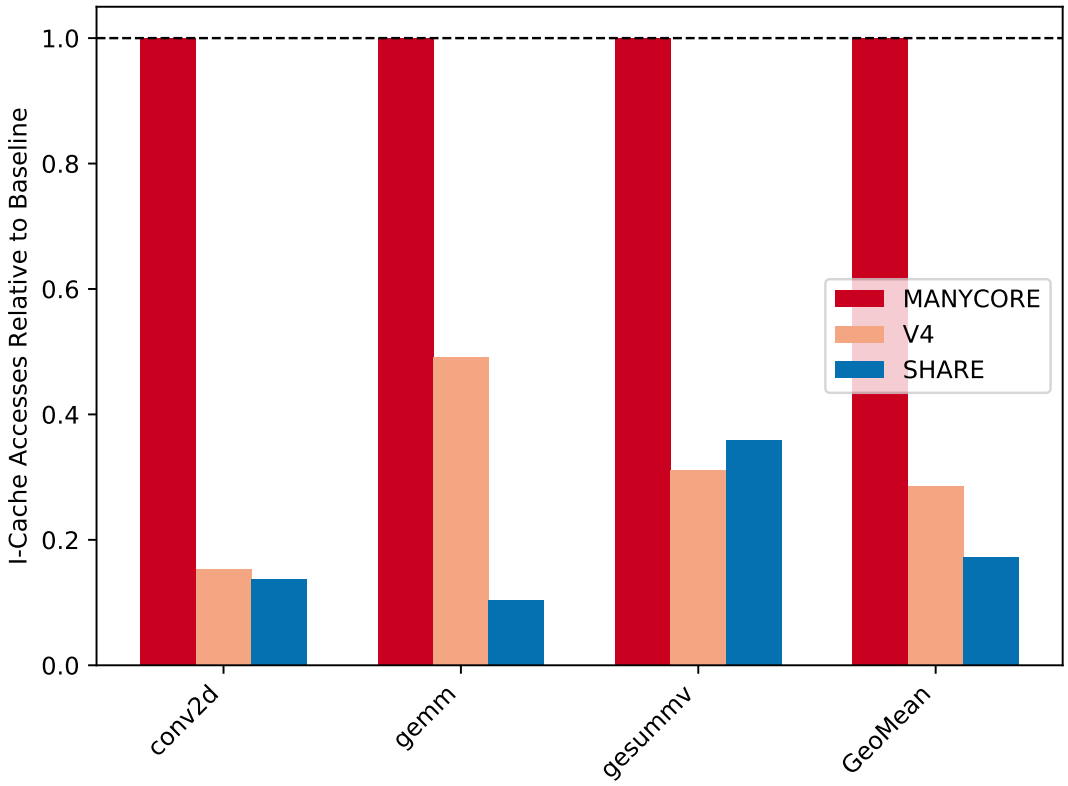


Figure 4.2: I-Cache access reduction of logical vector units and logical CGRAs against a baseline manycore.

general-purpose manycore.

4.2 Manycores and VLIW: A Shared Fate?

There are similarities between the design principles between manycores and VLIW architectures. Both rely on software to extract parallelism: VLIW for ILP and manycore for TLP. This is concerning because VLIW architectures have not achieved major commercial adoption after decades of research and engineering. High performance CPUs continue to employ expensive superscalar pipelines often with O3 scheduling.

Like superscalar dominating the CPU landscape over VLIW, GPUs may continue to dominate high throughput computing even though manycores are more efficient in principle. The software challenges may be too hard to overcome. However, there are key differences between the two scenarios.

First, VLIW mainly relied on compiler techniques rather than improvements to programming languages. Compiler techniques require less programming effort and are easier to adopt, but are limited because they neither have runtime information that hardware has nor the high-level program knowledge that the programmer has. In contrast, manycore performance mainly comes from the programming language and programmer effort rather than compiler-based optimization.

The second challenge was the fundamental limit to parallelism that can be extracted in software. VLIW is limited in ILP due to control flow which requires runtime information to schedule effectively. Techniques have been proposed to limit the challenges with control flow [31], but limited software-exposed ILP remains a

fundamental problem. Manycores also suffer from the lack of TLP in applications due to unavoidable fine-grained synchronization. The techniques proposed in this thesis may mitigate this problem by trading off unused TLP for either DLP or ILP using reconfiguration. In addition, manycores have ways to mitigate expensive synchronization overhead using cheap core-to-core synchronization like token queues [25].

4.3 Programming Challenges

This thesis did not consider improving the programming experience in manycores. Parallel programming of massively multithreaded architectures is a challenge and must be solved to improve manycore viability. Even the embarrassingly parallel applications considered in this thesis were often hard to achieve high performance on. More complex applications with fine-grained synchronization will be more difficult.

Memory management was difficult with the scratchpad, especially dealing with the stack. The stack must be placed in the scratchpad for performance, but is controlled by the compiler rather than the programmer. This creates awkward corruption errors or forces the programmer to be conservative with memory allocation. Manycores should consider small local caches for compiler-managed data to prevent these conflicts.

Techniques to improve MLP were incredibly important for performance in the manycore architectures explored. Every benchmark was either DRAM latency constrained or DRAM bandwidth constrained when using the baseline lightweight cores. To improve MLP, our systems used inexpensive software DMA with wide

memory accesses and frames to keep track of the responses. However, this technique was a challenge to program especially with tracking when asynchronous data requests had completed. Most of the bugs encountered in the evaluations of Rockcress and Watercress were due to functional errors with frame management.

Software reconfiguration also requires programmer effort to utilize. Compiler assistance can help reduce the boilerplate code and unify the programming style to use logical accelerators.

4.4 Future Work

We believe there is a rich design space of manycore resource sharing. Unlike multicores, manycore functional units and memories are physically close, which enables efficient sharing like proposed in this thesis. Section 2.10 mentioned how Rockcress may benefit from forwarding post-decode control signals rather than just instructions. Another potential option is increasing ILP using VLIW issue across cores rather than CGRA-style dataflow.

Using frames in both Rockcress and Watercress enabled efficient memory access in sharing. This mechanism is very important for data distribution in logical accelerators, but also enabled cheap MLP for the lightweight cores and is useful even in the baseline manycores. Polishing and deploying this feature in ordinary manycores may be worthwhile.

BIBLIOGRAPHY

[1] Dennis Abts, Abdulla Bataineh, Steve Scott, Greg Faanes, Jim Schwarzmeier, Eric Lundberg, Tim Johnson, Mike Bye, and Gerald Schwoerer. The cray blackwidow: a highly scalable vector multiprocessor. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[2] Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Aliaksei Chapyzhenka, Silviu Chiricescu, Ken Dockser, Bob Dreyer, Roger Espasa, Sean Halle, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Constantine Korikov, Ben Korpan, Hanna Kruppe, Yunsup Lee, Guy Lemieux, Filip Moc, Rich Newell, Albert Ou, David Patterson, Colin Schmidt, Alex Solomatnikov, Steve Wallach, Andrew Waterman, and Jim Wilson. RISC-V "V" vector extension, version 0.9, May 2020. `https://github.com/riscv/riscv-v-spec`.

[3] Anand Lal Shimpi. The Bulldozer Review: AMD FX-8150 Tested. `https://www.anandtech.com/show/4955/the-bulldozer-review-amd-fx8150-tested`.

[4] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. BYOC: A "bring your own core" framework for heterogeneous-ISA research. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[5] D. Bates, A. Bradbury, A. Koltes, and R. Mullins. Exploiting tightly-coupled cores. In *Journal of Signal Processing Systems*, volume 80, pages 103–120, 2015.

[6] Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanović. Cache refill/access decoupling for vector machines. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004.

[7] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar operand networks: on-chip interconnect for ilp in partitioned architectures. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 341–353, 2003.

[8] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal.

Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, 2005.

[9] Philip Bedoukian, Neil Adit, Edwin Peguero, and Adrian Sampson. Software-defined vector processing on manycore fabrics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, page 392–406, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, 2008.

[11] Bespoke Silicon Group. HammerBlade. `https://github.com/bespoke-silicon-group/bsg_bladerunner`.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2), May 2011.

[13] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[14] Demid Borodin, Winston Siauw, and Sorin Dan Cotofana. Functional unit sharing between stacked processors in 3d integrated systems. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 311–317, 2011.

[15] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 18–29, 2004.

[16] Ajay Brahmakshatriya, Emily Furst, Victor A. Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael B. Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman

Amarasinghe. Taming the zoo: The unified graphit compiler framework for novel architectures. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 429–442, 2021.

[17] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. Speculative precomputation on chip multiprocessors. In *In Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.

[18] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. EVA^2: Exploiting temporal redundancy in live computer vision. In *International Symposium on Computer Architecture (ISCA)*, 2018.

[19] Andrew Burgess, Ashley Whetter, George Field, Graham Markall, Hendrik Oosenbrug, James Pallister, Jeremy Bennett, Neville Grech, Pierre Langlois, and Simon Cook. Embench: Open benchmarks for embedded platforms, 2019. URL: `https://github.com/embecosm/embench-beebs`.

[20] Matheus Cavalcante, Samuel Riedel, Antonio Pullini, and Luca Benini. Mempool: A shared-l1 memory many-core cluster with a low-latency interconnect. In *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 701–706, 2021.

[21] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197, 2007.

[22] Liang Chen and Tulika Mitra. Shared reconfigurable fabric for multi-core customization. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 830–835, 2011.

[23] Lin Cheng, Peitian Pan, Zhongyuan Zhao, Krithik Ranjan, Jack Weber, Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Preslav Ivanov, Dustin Richmond, Michael B. Taylor, Zhiru Zhang, and Christopher Batten. A tensor processing framework for cpu-manycore heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1620–1635, 2022.

[24] Lin Cheng, Max Ruttenberg, Dai Cheol Jung, Dustin Richmond, Michael Taylor, Mark Oskin, and Christopher Batten. Beyond static parallel loops: Supporting dynamic task parallelism on manycore architectures with software-managed scratchpad memories. In *Proceedings of the 28th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, page 46–58, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald G. Dreslinski, Christopher Batten, and Michael Bedford Taylor. The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, 2018.

[26] Cláudio Machado Diniz, Muhammad Shafique, Sergio Bampi, and Jörg Henkel. Run-time accelerator binding for tile-based mixed-grained reconfigurable architectures. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2014.

[27] David R. Ditzel and the Esperanto team. Accelerating ml recommendation with over 1,000 risc-v/tensor processors on esperanto's et-soc-1 chip. *IEEE Micro*, 42(3):31–38, 2022.

[28] Thomas H. Dunigan, Jeffrey S. Vetter, James B. White, and Patrick H. Worley. Performance evaluation of the Cray X1 distributed shared-memory architecture. *IEEE Micro*, 25(1):30–40, 2005.

[29] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[30] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the alpha architecture. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 281–292, 2002.

[31] Joseph Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[32] Philip Garcia and Katherine Compton. Kernel sharing on reconfigurable multiprocessor systems. In *2008 International Conference on Field-Programmable Technology*, pages 225–232, 2008.

[33] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 151–162, New York, NY, USA, 2006. Association for Computing Machinery.

[34] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 291–303, New York, NY, USA, 2002. Association for Computing Machinery.

[35] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2011.

[36] Scott Grauer-Gray and Louis-Noël Pouchet. PolyBench/GPU: Implementation of PolyBench codes for GPU processing, 2012. URL: `http://www.cs.ucla.edu/pouchet/software/polybench`.

[37] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. Corefab: Concurrent reconfigurable fabric utilization in heterogeneous multi-core systems. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, New York, NY, USA, 2014. Association for Computing Machinery.

[38] S. Gupta, S. Feng, A. Ansari, and S. Mahlke. Erasing core boundaries for robust and configurable performance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[39] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 12–23, 2011.

[40] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level. In *2018 IEEE In-*

ternational Symposium on High Performance Computer Architecture (HPCA), pages 608–619, 2018.

[41] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Branch-aware loop mapping on cgras. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

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

[43] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. Elastic cgras. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, page 171–180, New York, NY, USA, 2013. Association for Computing Machinery.

[44] Intel. AVX-512. `https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html`.

[45] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[46] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[47] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 228–241, New York, NY, USA, 2015. Association for Computing Machinery.

[48] Jeffrey Dastin and Stephen Nellis. For tech giants, AI like Bing and Bard poses billion-dollar search problem. `https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/`.

[49] Xingxing Jin, Brian Daku, and Seok-Bum Ko. Improved GPU SIMD control flow efficiency via hybrid warp size mechanism. *Microprocessors and Microsystems*, 38:717–729, 2014.

[50] Jonathan Vanian and Kif Leswing. ChatGPT and generative

AI are booming, but the costs can be extraordinary. https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html.

[51] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[52] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. Ruche networks: Wire-maximal, no-fuss nocs : Special session paper. In *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2020.

[53] Karl Rupp. Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data.

[54] Khronos Group. OpenCL. https://www.khronos.org/opencl/.

[55] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 305–316, 2012.

[56] Changkyu Kim, Simha Sethumadhavan, Madhu Saravana Sibi Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[57] Sung Kim, Morteza Fayazi, Alhad Daftardar, Kuan-Yu Chen, Jielun Tan, Subhankar Pal, Tutu Ajayi, Yan Xiong, Trevor Mudge, Chaitali Chakrabarti, David Blaauw, Ronald Dreslinski, and Hun-Seok Kim. Versa: A 36-core systolic multiprocessor with dynamically reconfigurable interconnect and memory. *IEEE Journal of Solid-State Circuits*, 57(4):986–998, 2022.

[58] John Kloosterman, Jonathan Beaumont, D. Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. Regless: Just-in-time operand staging for gpus. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 151–164, 2017.

[59] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. The vector-thread architecture. In *International Symposium on Computer Architecture (ISCA)*, 2004.

[60] Ahmad Lashgar, Amirali Baniasadi, and Ahmad Khonsari. Dynamic warp resizing: Analysis and benefits in high-performance SIMT. In *IEEE International Conference on Computer Design (ICCD)*, 2012.

[61] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 46–57, New York, NY, USA, 1998. Association for Computing Machinery.

[62] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[63] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. The Hwacha vector-fetch architecture manual version 3.8.1. Technical Report UCB/EECS-2015-262, EECS Department, University of California, Berkeley, December 2015.

[64] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 487–498, New York, NY, USA, 2013. Association for Computing Machinery.

[65] Lian Li, Lin Gao, and Jingling Xue. Memory coloring: a compiler approach

for scratchpad memory management. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 329–338, 2005.

[66] B. Mei, M. Berekovic, and J-Y. Mignolet. *ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors*, pages 255–297. Springer Netherlands, Dordrecht, 2007.

[67] Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 6.5. `https://github.com/HewlettPackard/cacti`.

[68] Dheya Mustafa. A survey of performance tuning techniques and tools for parallel applications. *IEEE Access*, 10:15036–15055, 2022.

[69] NVIDIA. CUDA C++ Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[70] Oak Ridge National Laboratory. Frontier. `https://www.olcf.ornl.gov/frontier/`.

[71] Oak Ridge National Laboratory. Frontier supercomputer debuts as world's fastest, breaking exascale barrier. `https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier`.

[72] Andreas Olofsson. Epiphany-v: A 1024 processor 64-bit risc system-on-chip, 2016. `https://arxiv.org/abs/1610.01832`.

[73] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Mirage cores: The illusion of many out-of-order cores using in-order hardware. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 745–758, 2017.

[74] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, Kuba Kaszyk, John Magnus Morton, Jiawen Sun, Michael O'Boyle, Murray Cole, Chaitali Chakrabarti, David Blaauw, Hun-Seok Kim, Trevor Mudge, and Ronald Dreslinski. Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 175–190, New York, NY, USA, 2020. Association for Computing Machinery.

[75] Y. Park, J. J. K. Park, H. Park, and S. Mahlke. Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 84–95, 2012.

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

[77] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery.

[78] Carl Ramey. Tile-gx100 manycore processor: Acceleration interfaces and architecture. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–21, 2011.

[79] Reddit. Efficiency AMD GPUs and Nvidia GPUs have gained in 10 years. `https://www.reddit.com/r/hardware/comments/o87h5z/i_made_charts_showing_how_much_efficiency_amd/`.

[80] Robert Griesemer and Rob Pike and Ken Thompson. Go Language. `go.dev`.

[81] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. A variable warp size architecture. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[82] Austin Rovinski, Chun Zhao, Khalid Al-Hawaj, Paul Gao, Shaolin Xie, Christopher Torng, Scott Davidson, Aporva Amarnath, Luis Vega, Bandhav Veluri, Anuj Rao, Tutu Ajayi, Julian Puscar, Steve Dai, Ritchie Zhao, Dustin Richmond, Zhiru Zhang, Ian Galton, Christopher Batten, Michael B. Taylor, and Ronald G. Dreslinski. Evaluating celerity: A 16-nm 695 giga-risc-v instructions/s manycore processor with synthesizable pll. *IEEE Solid-State Circuits Letters*, 2(12):289–292, 2019.

[83] Richard M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[84] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture (ISCA)*, 2003.

[85] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, August 2008.

[86] Raul Silva, Guilherme Korol, Michael Guilherme Jordan, Marcelo Brandalero, Michael Hübner, Monica Pereira, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck. A management technique for concurrent access to a reconfigurable accelerator. In *2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6, 2020.

[87] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[88] James E. Smith. Decoupled access/execute computer architectures. In *International Symposium on Computer Architecture (ISCA)*, 1982.

[89] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[90] Mostafa I. Soliman and Ghada Y. Abozaid. Shared cryptography accelerator for multicores to maximize resource utilization. In *The 2011 International Conference on Computer Engineering Systems*, pages 33–38, 2011.

[91] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, Geoff Lowney, Adam Herr, Christopher Hughes, Timothy Mattson, and Pradeep Dubey. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189, 2019.

[92] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez,

Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, March 2017.

[93] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 2000.

[94] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 291–302, 2003.

[95] Tuan Ta, Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, Courtney Golden, and Christopher Batten. big.vlittle: On-demand data-parallel acceleration for mobile systems on chip. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 181–198, 2022.

[96] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *International Symposium on Computer Architecture (ISCA)*, 2018.

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

[98] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Design Automation Conference (DAC)*, 2008.

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

[100] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jae Won Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew I. Frank, Saman P. Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, 2002.

[101] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota,

Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, page 2, USA, 2004. IEEE Computer Society.

[102] TOP500. TOP500 November 2022 List. `https://www.top500.org/lists/top500/2022/11/`.

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

[104] Steve Undy. Poulson: An 8 core 32 nm next generation intel® itanium® processor. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–22, 2011.

[105] Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, page 145–154, New York, NY, USA, 1999. Association for Computing Machinery.

[106] Dani Voitsechov and Yoav Etsion. Single-graph multiple flows: Energy efficient design alternative for gpgpus. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 205–216, 2014.

[107] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. *High-Performance Computing on the Intel Xeon Phi: How to Fully Exploit MIC Architectures*. Springer, 2014.

[108] Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten. Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 173–186, 2020.

[109] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, volume I: Base user-level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[110] Matthew A. Watkins and David H. Albonesi. Remap: A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 2010 43rd Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO '43, page 497–508, USA, 2010. IEEE Computer Society.

[111] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[112] Florian Zaruba, Fabian Schuiki, and Luca Benini. Manticore: A 4096-core risc-v chiplet architecture for ultraefficient floating-point computing. *IEEE Micro*, 41(2):36–42, 2021.

[113] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.