Exploiting Reconfiguration and Co-Design for Domain-Agnostic Hardware Acceleration

by

Sung Kim

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical and Computer Engineering)
in the University of Michigan
2023

Doctoral Committee:

Associate Professor Ronald Dreslinski, Chair
Professor David Blaauw
Assistant Professor Jean-Baptiste Jeannin
Associate Professor Hun-Seok Kim

Sung Kim

sungmk@umich.edu

ORCID iD: 0000-0002-9010-4617

© Sung Kim 2023

Acknowledgments

First, I'd like to thank Ron Dreslinski for his mentorship during my Ph.D. Ron's approach to research fosters curiosity and open-mindedness, that lead naturally to driven and productive researchers! Additionally, I'd like to thank Hun-Seok Kim for his trust and confidence as a co-advisor. Among other things, Hun-Seok enthusiastically supported a leading role in the DARPA SDH project when I first joined the University of Michigan. Working on SDH was one of the best professional and personal experiences I've had to date. I'd also like to thank David Blaauw, Trevor Mudge, and Chaitali Chakrabarti, who were invaluable sources of advice, technical input, and lively discussion. In particular, I'm grateful for David's steady support during SDH chip development.

I've been fortunate to have great informal teachers, colleagues, and friends along the way. Special thanks goes to Morteza Fayazi, Tutu Ajayi, Subhankar Pal, Siying Feng, Dong-Hyeon Park, Kuan-Yu Chen, Alhad Daftardar, Jielun Tan, Chi-Sheng Yang, and all the members of the CE Lab I had the pleasure of interacting with. At UW, I enjoyed my time with (and learned a great deal from) Patrick Howe, Fahim ur Rahman, Xun Sun, Anthony Smith, Rajesh Pamula, Thierry Moreau, Armin Alaghi, Amrita Mazumdar, and the rest of the Sampa crew.

None of my work would be possible without Visvesh Sathe and Luis Ceze. Besides the irreplaceable and unique technical training, Visvesh's mentorship from my time at UW is still a frequent source of wisdom and inspiration.

Finally, thank you to my family, for the love and support over the years.

Table of Contents

Acknowledgments ii

List of Figures v

List of Tables ix

Abstract x

Chapter

I. Introduction 1

1.1 Thesis Overview 3

II. Reconfigurable Hardware-Software 7

2.1 Introduction 7
2.2 Motivation 11
2.3 Hardware Design 12
2.3.1 Reconfigurable Modes 13
2.3.2 Reconfigurable Data Cache 15
2.3.3 Reconfigurable Crossbar 16
2.3.4 Processor Cores 17
2.3.5 Host-Device Relationship 17
2.4 Software Interfaces 18
2.5 Test Kernels 19
2.6 FFT Kernel Generation 21
2.6.1 Model-Specific Mappings 21
2.6.2 Limitations of Explicit Programming 23
2.6.3 MIMD Kernel Generation 24
2.7 Modeling and Experimental Methodology 24
2.8 Results 27
2.9 Related Work 34
2.10 Conclusion 36

III. A Reconfigurable Systolic Multiprocessor 37

3.1 Introduction 37
3.1.1 High-Level Objectives 38
3.2 Hardware Overview 39
3.2.1 Compute Tiles 39
3.2.2 Memory Hierarchy and Tile-External Support 41
3.3 Algorithm-Driven Reconfigurability 43
3.3.1 Reconfigurable Crossbar 43
3.3.2 Reconfigurable On-Chip Memory 46
3.3.3 Systolic Execution 48
3.3.4 Hierarchical Thread Primitives 51
3.4 Chip Implementation 54
3.5 Baselines and Test Methodology 56
3.6 Measurements and Analysis 58
3.6.1 Nominal Energy-Efficiency and Performance 60
3.6.2 Voltage-Scaled Characteristics 62
3.7 Related Work 62
3.8 Conclusion 65

IV. Compiler Support for Versatile Hardware 66

4.1 Introduction 66
4.2 Motivation 68
4.3 Polca: Polyhedral Compilation for Versatile Hardware 69
4.4 Machine Model and IR 70
4.4.1 Accelerator Architecture 70
4.4.2 Polca Dialect 71
4.5 Exploiting Versatile Parallelism 72
4.5.1 Frontend 72
4.5.2 Polyhedral Loop Restructuring. 73
4.5.3 SPMD Parallelism. 73
4.5.4 Systolic Reduction Parallelism. 74
4.5.5 MIMD Conversion. 76
4.5.6 Final Lowering 78
4.6 Evaluation 78
4.7 Related Work 84
4.8 Conclusion 86

V. Conclusion 88

Bibliography 90

List of Figures

Figure

1.1 The demand for computation has outpaced the capabilities of conventional processors, such as CPUs. This is largely due to the departure from Dennard Scaling in the early 2000s, marked by stringent physical constraints in power density and clock frequency. It is unclear whether ASICs, in addition to repurposed legacy architectures, will be sufficient to support the growing range of new applications and requirements. . . . 2

1.2 Conventional general-purpose architectures and ASICs reside at extremes of a performance, energy-efficiency, and programmability spectrum. In contrast, accelerators that exploit algorithm characteristics without sacrificing software-programmability target a knee point in the performance-efficiency pareto curve. Hardware-algorithm co-design techniques that apply to broad classes of computations are a promising way to improve the pareto frontier, enabling better support for emerging applications that continually evolve. 3

2.1 High-level analysis of workloads and key constituent kernels from the DARPA software-defined hardware (SDH) program. **(top)** breakdown of execution time, and **(bottom)** summary characterization. Runtime of SDH workloads is dominated by diverse linear algebra and signal processing kernels, with disparate levels of arithmetic intensity, data reuse, and control/memory divergence. 9

2.2 Performance disparity on synthetic workloads for simulated multi-core systems with different on-chip memory type, and different memory scopes. **(left)** Speedup of scratchpad memory (SPM) over iso-capacity cache in a 1-core system, under contiguous and random access patterns; SPM and cache are provided the same stream of load/store addresses for a given access pattern. **(right)** Speedup of 1 shared memory over 8 private memories in an 8-core system, under data access patterns with different levels of overlap. 11

2.3 Overview of the Transmuter hardware design. **(a)** Top-level view of a Transmuter system including host and DRAM interfaces. **(b)** A single cluster containing 4 tiles with associated L2 R-DCaches and L2 R-Xbars. Crossbar interfaces to HBM are non-reconfigurable. **(c)** Contents of a single tile, containing GPE cores, the LCP core, LCP-GPE work/status queues, L1 R-DCaches, and L1 R-XBars. Arbiters and instruction caches are omitted. **(d)** Microarchitecture of an R-XBar; circled numbers indicate the mode of operation (①: Arbitrate with LRG, ②: Transparent, ③: Rotate). 13
2.4 Overview of the three Transmuter configurations evaluated in this work. **(a)** Trans-SC: L1 shared cache with L2 shared cache. **(b)** Trans-PS: L1 private SPM with L2 private cache. **(c, d)** Trans-SA: L1 systolic array with L2 private cache. 14
2.5 Illustration of crossbar ROTATE functionality: **(a)** Physical connections between cores and R-DCache banks are time-multiplexed and "rotate" every cycle. **(b)** Alternating port connections provide functional behavior equivalent to spatial dataflow. 15
2.6 Summary of radix-2 FFT implementation: **(top)** Algorithmic partitioning of vertical FFT stages onto processors. **(bottom)** Resulting pipeline of butterfly computations across processors. 22
2.7 Overview of FFT kernel generation. 24
2.8 Performance comparison across kernels and input sizes for Trans-SC, Trans-PS, and Trans-SA Transmuter configurations. 29
2.9 Performance attribution: (left axis) attribution of cycles for different architectural events, and (right axis) work imbalance across cores. 30
2.10 Average performance (throughput) and energy-efficiency, normalized to the CPU baseline. Data is averaged across all test inputs for each kernel. GeoMean columns indicate the aggregated geometric mean for compute-bound and memory-bound kernels. 32
3.1 Versa exploits reconfiguration of on-chip memory and compute to provide algorithm-optimized hardware characteristics. Based on learnings from Transmuter, Versa evolves the programming model, provides enhanced reconfigurability, and resolves key scalability limitations. 39
3.2 Composition of a Versa compute tile. 40
3.3 Overview of the Versa prototype and memory hierarchy. 43
3.4 The reconfigurable crossbar (RXB) and overview of sub-modes. Diagrams in Private and Queue illustrate a sub-section of the crosspoint matrix. 45
3.5 Sub-mode logic and SRAM components in the reconfigurable on-chip memory. 46
3.6 Comparison of banking schemes (wide v.s. sub-banked): **(a)** data utilization per access, and **(b)** design costs. 47
3.7 Conventional instruction sequence: obligatory memory access overheads are interleaved with computation. 48

3.8 Example computation with R2R: **(a)** instruction sequence, **(b)** logical (pipeline) view, and **(c)** physical view of read/write conventions. R2R-writes bypass the local RF, while R2R-reads utilize local registers. Cardinal directions are inverted for writes to maintain spatial symmetry. . . . 49
3.9 R2R microarchitectural implementation: **(a)** integration with the ARM Cortex-M4F pipeline, **(b)** interception of operands in the R2R Shim and 2-bit link-state control. 50
3.10 Accelerated thread-synchronization barriers: **(a)** Distributed scratchpads decentralize atomic updates to barrier data. Cores first synchronize within tiles over the T-SPM, followed by a global synchronization phase involving only managers over the G-SPM. After managers pass the global synchronization phase, each manager releases the barrier (and the workers) within their respective tiles. **(b)** Serialization is reduced by partitioning synchronization into parallel sections. In contrast to a centralized barrier with $O(N)$ scaling with respect to core count, the distributed hardware-software barrier implementation scales with $O(W+T)$, where W and T are the number of workers per tile and number of tiles, respectively. . . . 52
3.11 Comparison of thread synchronization latency with alternative barrier implementations. `pthreads` represents the off-the-shelf thread barrier implementation from the popular `pthreads` library, measured on a quadcore Arm A57 with 4 spawned threads. `centralized SP` is a barrier implemented on Versa using single centralized scratchpad memory. `tree-based SP` is the Versa thread barrier implementation that leverages distributed scratchpads. 52
3.12 **(L)** Die photo and **(R)** summary characteristics. 55
3.13 Setups for chip testing: **(L)** Linux single-board-computer debug host, primarily for bringup and data collection, and **(R)** FPGA interface for emulated MPSoC integration. 55
3.14 Evaluation kernels from MachSuite: kernels are selected to capture a range of characteristics. 56
3.15 Energy-efficiency improvement over CPU and GPU baselines: **(a)** trends across dataset footprints, and **(b)** summary boxplots. Boxplots are labeled on min, median, and max values. 59
3.16 Chip characteristics under voltage scaling. **(L)** System frequency and power, **(M)** energy-per-cycle, and **(R)** the performance-efficiency Pareto curve. 62
4.1 Development of accelerator kernels and associated libraries is a challenging task that prioritizes hardware performance. Because kernel development is already complex and effort-intensive, *e.g.* requiring extensive tuning and detailed hardware knowledge, the rapid evolution of both hardware and algorithms presents a non-trivial challenge. 67
4.2 Overview of the proposed compiler and IR lowering pipeline. 69
4.3 Overview of Polca target accelerator. 70
4.4 Key mid-level analysis and transformation passes. 72

4.5 Overview of Polca spatial reduction parallelization. **(a)** Operations associated with a reduction loop are tabulated into a preamble, body, and postamble. **(b)** Unparallelized reductions are restricted to parallelism in outer dimensions, limiting data locality. Polca leverages spatial links to parallelize inner reduction loops with low performance overhead, yielding net speedup. **(c)** Pseudocode for functional partitioning of work. . . . 75
4.6 Physical and temporal views of the optimized reduction strategy. **(a)** Outer-parallel iterations are mapped onto hardware stripes (vertical stripes shown), with one stripe executing an entire parallel reduction. **(b)** Temporal view of execution for one stripe. Intra-stripe reduction in core postambles is used to aggregate per-core (parallel) local reductions. 76
4.7 Excerpt of transformed IR for reduction, nested in a tiled parallel loop. Identifiers edited for clarity. Affine predication on symbolic spatial indices (`tid, sid, eid`) models differentiated execution. Symbols are later substituted for constants during MIMD conversion; downstream simplification passes produce predication-free IR for each core. 77
4.8 Speedups normalized to baseline `sequential` execution. `tiled` shows sequential execution with tiling and basic restructuring optimizations enabled, while `tiled-parallel` enables both tiling and base parallelization optimizations. 81
4.9 Speedups for kernels amenable to systolic reduction parallelization. `r2r-reduce` enables all optimizations for tiling, base parallelization, and reduction parallelization. `est-atomic-<20,100,200>` labels denote estimated speedups when R2R-based spatial data transfers are substituted for 20-200ns atomic RMWs. **(a)** Cumulative speedup over baseline `sequential` execution, and **(b)** relative speedup over `tiled-parallel`. . . . 82

List of Tables

Table

2.1 Key characteristics of Transmuter and comparable alternatives. 10
2.2 Microarchitectural parameters for gem5 simulations. 25
2.3 Hardware and software comparison baselines. 27
2.4 Breakdown of power and area for one 64× 64 Transmuter cluster. 28
2.5 Estimated speedup for end-to-end applications. 32
3.1 Composition of logical memory modes. R2R functionality is orthogonal and compatible with all. 44
3.2 Hardware baselines. 56
3.3 Efficiency and performance improvement from dynamic reconfiguration between modes. Stars indicate whether the mode-advantage is consistent (★) or mixed (☆). 58
3.4 Median improvements across all kernels. 58
4.1 Summary of average (geomean) speedup across kernels. 'Rel.' denotes relative speedup obtained by stacking optimizations, while 'Total' denotes total (cumulative) speedup over sequential. 84

Abstract

Hardware accelerators have become permanent features in the post-Dennard computing landscape, displacing conventional processors for a variety of applications. Not only have semiconductor power and performance limitations become more stringent, but the demand for computing power has accelerated at an unprecedented pace. Data and compute-intensive application domains – such as machine learning, vision, and bioinformatics – require processing power orders of magnitude greater than what general-purpose processors can provide. The requirements of emerging applications, in conjunction with the limitations associated with conventional processors, have resulted in industry-wide efforts to develop new application-specific integrated circuit (ASIC) designs. Nevertheless, conventional ASIC accelerators sacrifice programmability for the sake of performance and energy-efficiency – a non-ideal state of affairs.

To address the problems above, this thesis introduces an end-to-end hardware-software concept for a semi-specialized accelerator that *retains ASIC-like characteristics without sacrificing software programmability*. In particular, we propose hardware-software co-design techniques to (1) exploit workload characteristics in programmable accelerators via rapid hardware reconfiguration, and (2) develop a compiler stack that generates optimized, auto-parallelized application kernels.

Chapter I discusses why hardware acceleration is needed, the current landscape of ASIC and general-purpose processor hardware, and identifies challenges associated with building accelerators that are both programmable and efficient. Chapter II introduces an initial design concept for a rapidly-reconfigurable programmable accelerator, and discusses challenges associated with the paradigm. Based on learnings from Chapter II, Chapter III

proposes key enhancements to improve performance and resolve key hardware bottlenecks, and presents results from a fabricated prototype chip. Chapter IV discusses software development challenges inherent with our hardware approach, and introduces an end-to-end optimizing compiler to automatically generate kernels that exploit the proposed accelerator architecture.

Chapter I

Introduction

Hardware-software co-design is more important than ever. Improvements in computing hardware translate to faster software, more efficient resource usage, and can enable entirely new application domains such as computational biology, machine learning (ML), and AR/VR. In conjunction, better, more detailed information about how programs behave can be structurally exploited in hardware to improve application support. Nevertheless, while computing devices are directly constrained by physics, software is not – the breadth, diversity, and processing requirements of software continue to grow at staggering rates. Coupled with underlying semiconductor technologies that are no longer improving at historical rates (thereby yielding robust, predictable improvements in power and performance), fundamental changes elsewhere in hardware-software stack are required to sustain future computing demand.

Benefits from traditional semiconductor process scaling – the physical shrinking of integrated circuits and associated benefits from better chip fabrication technology – are nearly exhausted. For several decades, computer architects and system designers could rely on semiconductor technologies with key metrics that improved predictably over short periods of time. Rate of improvement in transistor density (Moore's Law [1]), and scaling of integrated circuit electrical characteristics (Dennard Scaling [2])) are notable examples. Nevertheless, the escalating complexity of advanced semiconductor manufacturing, in ad-

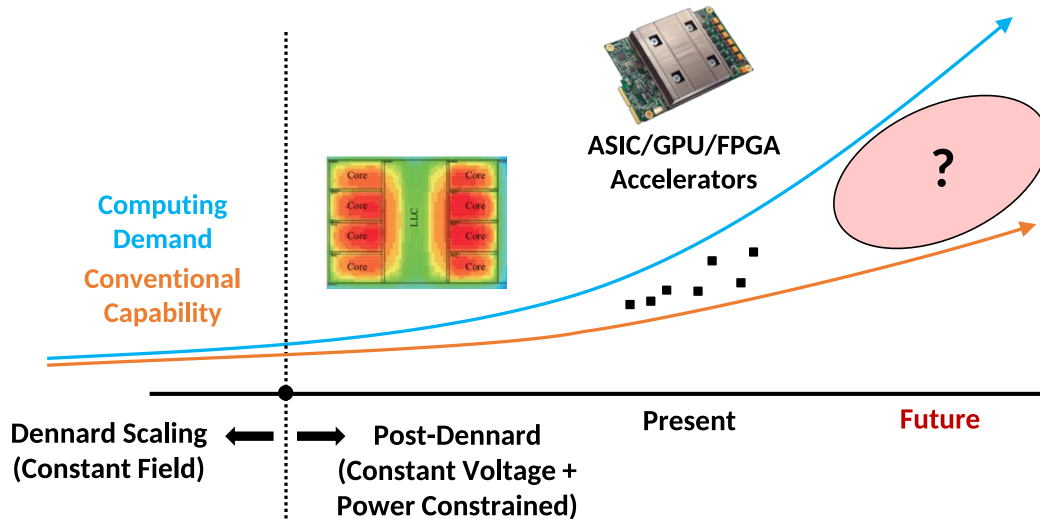


Figure 1.1: The demand for computation has outpaced the capabilities of conventional processors, such as CPUs. This is largely due to the departure from Dennard Scaling in the early 2000s, marked by stringent physical constraints in power density and clock frequency. It is unclear whether ASICs, in addition to repurposed legacy architectures, will be sufficient to support the growing range of new applications and requirements.

dition to physical constraints such as power density [3], have resulted in a marked departure from historical process scaling [4, 5]. As a result, domain experts, system architects, and engineers are no longer able to rely solely on general-purpose hardware to satisfy application requirements [6, 7]. This shift away from homogeneous CPU-based platforms puts more burden on complementary solutions, creating new opportunities for novel design (Figure 1.1).

Orthogonal to process scaling and conventional general-purpose approaches, hardware can be designed to take advantage of context-specific algorithm characteristics; this entails a pareto frontier of performance and energy-efficiency versus flexibility (Figure 1.2). For example, in contrast to general-purpose CPUs that are fully-programmable, an application-specific integrated circuit (ASIC) is designed to support a small, restricted set of computations. By reorganizing on-chip resources and stripping away unnecessary hardware units, an ASIC can yield energy-efficiency and performance orders of magnitude better than a general-purpose counterpart. The inherent tradeoff with ASIC-like specialization is a loss

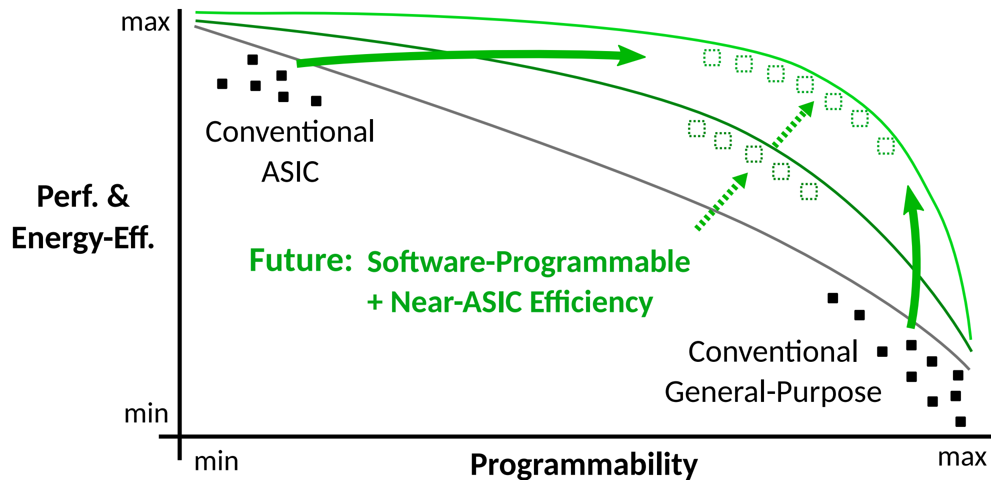


Figure 1.2: Conventional general-purpose architectures and ASICs reside at extremes of a performance, energy-efficiency, and programmability spectrum. In contrast, accelerators that exploit algorithm characteristics without sacrificing software-programmability target a knee point in the performance-efficiency pareto curve. Hardware-algorithm co-design techniques that apply to broad classes of computations are a promising way to improve the pareto frontier, enabling better support for emerging applications that continually evolve.

of flexibility – fixed-function designs are typically limited to one use-case and carry greater risk of near-term obsolescence. Given current engineering realities (e.g., resources and time required to develop and deploy new silicon), fixed-function ASICs are likely sub-optimal for applications that are still changing rapidly. Thus, while ASICs may be a good fit for mature computations[1], there is an abundant opportunity for semi-specialized designs that improve energy-efficiency with little to no impact on programmability.

1.1 Thesis Overview

This thesis investigates how architects and hardware designers can make better use of the specialization-programmability middle-ground illustrated in Figure 1.2. In particular, we show that algorithm-specific behavior can be exploited in hardware without sacrificing programmability, and that recent compilation techniques can be leveraged to rapidly construct performant, end-to-end compilers for general-purpose accelerators. We show

[1]For example, the use of ASICs and fixed-function IP in media codec.

that energy-efficient, tiled programmable architectures can be constructed from commodity off-the-shelf cores. Furthermore, we show that when such architectures are augmented with a targeted form of memory reconfiguration and core-to-core dataflow, they exhibit better performance and energy-efficiency due to hardware that can adapt to algorithm needs. To demonstrate how a reconfigurable parallel architecture can be exploited from high-level software, we develop an end-to-end optimizing compiler that builds on progressive lowering and polyhedral compilation techniques.

The following chapters of this dissertation are organized as follows.

Recofigurable Hardware-Software

Homogeneous CPU-based machines have been unable satisfy the compute and memory requirements of critical algorithms, such as machine learning, leading to the rapid growth and deployment of ASICs and graphics processing units (GPUs). Nevertheless, existing ASIC and GPU designs exhibit highly-suboptimal performance for specific classes of emergent computations. In particular, existing ASIC and GPU platforms lack sufficient support for sparsity, which is a key component of many emerging workloads. Chapter II introduces Transmuter, a tiled reconfigurable architecture and design-space exploration for software-defined reconfigurable hardware. Cycle-level simulation experiments with Transmuter illustrate the value of targeted memory reconfiguration for both sparse and dense kernels, and also clarify architectural trade-offs for the design in terms of compute and memory hierarchy design. This chapter also illustrates the importance of software support; while optimized kernels for multi-core accelerators can be generated with adhoc infrastructure, a holistic optimizing compiler is crucial for productivity and performance.

A Reconfigurable Systolic Multiprocessor

Inter-core data transfer and thread synchronization operations are critical performance and scalability bottlenecks for software-defined hardware designs. While processing ele-

ments in ASICs often achieve efficient data transfer over simple registers (*e.g.*, in systolic arrays), existing programmable processors lack an equivalent mechanism; this not only results in performance and energy-efficiency penalties, but also impacts the design and mapping of application kernels.

Chapter III introduces Versa, a programmable multi-core accelerator designed to exploit diverse workloads with high scalability and energy-efficiency. Based on lessons from Transmuter, Versa supports sparse and irregular workloads with a tiled SPMD/MIMD architecture composed of lightweight in-order cores. To exploit data-reuse characteristics on a per-algorithm basis, the Versa design introduces a reconfigurable hierarchy of interconnects and on-chip memory. Inspired by ASIC designs, Versa introduces a register-to-register (R2R) data transfer mechanism that fuses instruction-level computation with core-to-core communication. R2R obviates the need for expensive SRAM and cache-based communication, and enables systolic array-type algorithm mappings onto software-programmable hardware. To alleviate crucial parallelization overheads, Versa also incorporates hardware support for a hierarchical tree-based thread synchronization scheme. To evaluate our hypothesis and experimental hardware features, a prototype of Versa is fabricated in 28nm CMOS, along with a custom C++ software stack. We develop hand-tuned implementations for several kernels exhibiting diverse control and compute characteristics, and benchmark the prototype against comparable 22nm CPU and GPU designs.

Compiler Support for Versatile Hardware

Developing high-performance kernel code for accelerators in a scalable manner is challenging for several reasons. Because accelerator kernels are typically in the critical path of end-to-end execution time, developers strive for accelerator programs that extract maximum hardware performance. However, accelerator kernel development assumes deep knowledge of parallel software optimization, and also of underlying microarchitectural details – a non-ideal state of affairs. Even with specialized hardware and software knowl-

edge, delivering high-performance accelerator code is highly time and labor intensive, typically requiring invasive code optimizations. Such optimizations involve non-trivial code restructuring that not only obfuscates core algorithms, but also prevents accelerator software portability. The explosion and rapid evolution of accelerator variants only exacerbates the problems above.

Resolving critical software barriers – to yield accelerator code that is performant, portable, and extensible – is a problem that requires new compiler infrastructure. Chapter IV introduces Polca, an end-to-end optimizing compiler for parallel SPMD/MIMD architectures with reconfigurable memory and fine-grained core-to-core data transfer. Based on experiences developing accelerator kernel code for Transmuter and Versa, we identify key optimizations, such as auto-parallelization, memory tiling, and fast parallel reduction operations, that are non-trivial to develop by hand but crucial for performance. Through a combination of modular compiler infrastructure reuse, polyhedral techniques, and lightweight intermediate representation (IR) extensions, Polca supports the full compilation flow, from unoptimized single-threaded C++ to object code, for Versa and Transmuter accelerator targets. We validate the end-to-end compilation stack on a set of unoptimized benchmark kernels, demonstrating the feasibility of Polca and future accelerator compilers.

Chapter II

Recofigurable Hardware-Software

2.1 Introduction

The end of Dennard scaling and the slowing of Moore's law has driven both industry and research efforts to develop accelerator-centric solutions. GPPs, ASICs, and reconfigurable accelerators have been historically bound by three conflicting constraints: (1) software programmability, (2) algorithm-specificity, (3) and overhead from low-level configurability. While existing General-Purpose Processors (GPPs) are designed for high programmability, ASIC accelerators maximize performance and energy-efficiency, albeit for a single algorithm or at best a narrow set of kernels. For instance, hardware mechanisms such as caching and out-of-order execution improve performance for the general case, but carry significant energy and area overhead compared to ASIC designs. ASICs achieve high performance and efficiency by stripping away extraneous hardware (*e.g.*, control overhead), and exploit algorithm-specific data access patterns. At the same time, due to high non-recurring expenses, building ASIC accelerators is typically infeasible except for a small number of application-critical workloads. Furthermore, fixed-function ASICs for fast-moving application domains, such as machine learning, carry high risk for near-term obsolescence.

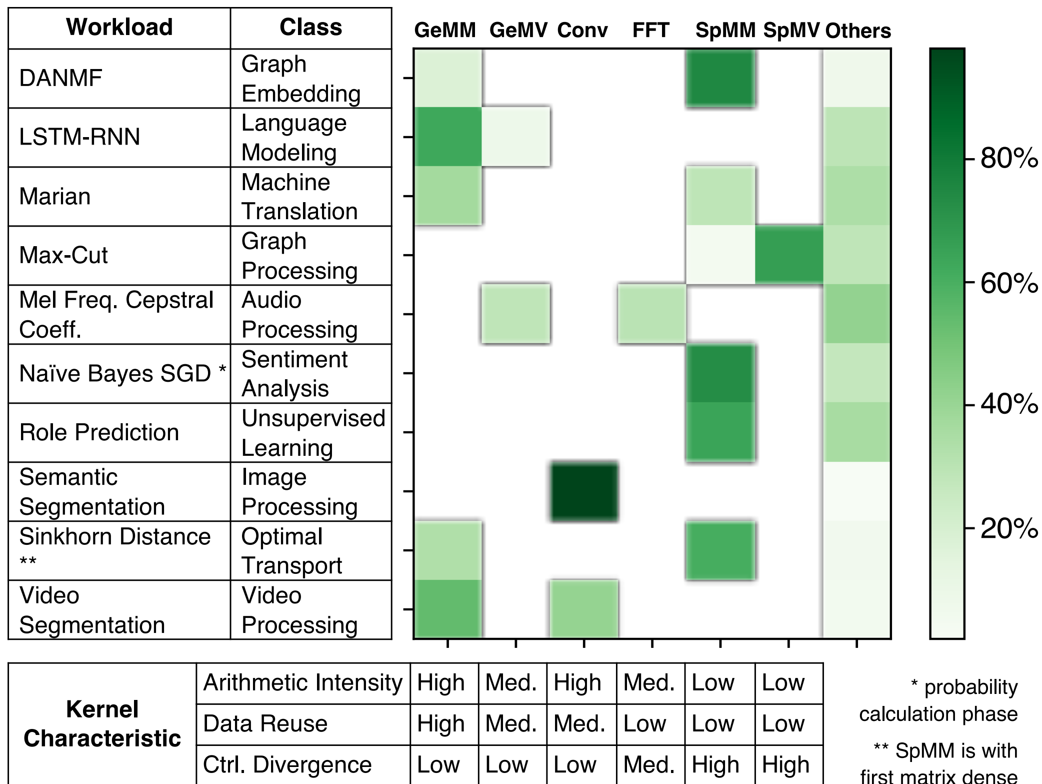
In an effort to accelerate emerging workloads without the drawbacks of an ASIC design, a variety of solutions that leverage reconfigurable hardware have been developed. For

instance, promising solutions that incorporate reconfigurability have developed for web search [8], machine learning [9], graph processing [10], sparse linear algebra [11], and general data-parallel workloads [12]. Nevertheless, compared to GPPs and ASICs, reconfigurable accelerators such as FPGAs have historically been limited to a narrow range of use-cases. FPGAs are effective tools for logic emulation, but gate- and bit-level reconfigurability constitute significant overhead – the efficiency of an FPGA accelerator is typically orders of magnitude lower than a corresponding ASIC design. FPGA reconfiguration at kernel granularity is also too slow to be used at runtime, on the order of microseconds to milliseconds despite partial reconfiguration [13, 14]. Furthermore, the programming model for FPGAs involves low-level hardware description languages that impact productivity. [1]

A key challenge for programmable accelerators is the need to handle emerging workloads with contrasting compute and memory characteristics. For instance, Figure 2.1 illustrates that real-world applications exhibit diverse characteristics not only across domains, but also within a single application. Handling both the inter- and intra-application diversity efficiently – in a single processor architecture – calls for an architecture without a conventional, static hierarchy of compute and memory. A programmable accelerator for emerging applications must be capable of tailoring itself at runtime, according to the needs of different applications and diverse underlying kernels.

To overcome the challenges above we present a flexible accelerator called Transmuter [15]. Transmuter is a hardware design that bridges the gap between GPPs and ASICs via low-latency, runtime reconfiguration of on-chip resources. In contrast to FPGAs and ASICs, Transmuter retains software-programmability, but exploits lightweight reconfiguration to achieve greater performance and efficiency compared to GPPs. In particular, reconfiguration of on-chip memory type, resource sharing, and dataflow are used to address the requirements of contrasting application kernels.

[1] While the FPGA programming model is currently a non-trivial barrier to the deployment of FPGA accelerators, compiler and high-level synthesis (HLS) solutions are improving rapidly. The FPGA programming model will likely improve. On the other hand, energy-efficiency overheads in FPGAs appear to be irreconcilable, and constrained by logic emulation use-cases.



Workload	Class
DANMF	Graph Embedding
LSTM-RNN	Language Modeling
Marian	Machine Translation
Max-Cut	Graph Processing
Mel Freq. Cepstral Coeff.	Audio Processing
Naïve Bayes SGD *	Sentiment Analysis
Role Prediction	Unsupervised Learning
Semantic Segmentation	Image Processing
Sinkhorn Distance **	Optimal Transport
Video Segmentation	Video Processing

Kernel Characteristic		GeMM	GeMV	Conv	FFT	SpMM	SpMV
	Arithmetic Intensity	High	Med.	High	Med.	Low	Low
	Data Reuse	High	Med.	Med.	Low	Low	Low
	Ctrl. Divergence	Low	Low	Low	Med.	High	High

Figure 2.1: High-level analysis of workloads and key constituent kernels from the DARPA software-defined hardware (SDH) program. **(top)** breakdown of execution time, and **(bottom)** summary characterization. Runtime of SDH workloads is dominated by diverse linear algebra and signal processing kernels, with disparate levels of arithmetic intensity, data reuse, and control/memory divergence.

Hardware	Program-mability	Compiler Support	Reconfig. Time	Relative Efficiency
ASIC	N.A.	Custom	N.A.	Very High
CGRA	Partial	Custom	$O(\mu s)$-$O(ns)$	High
FPGA	High	COTS	$O(ms)$-$O(\mu s)$	Medium
ASIP/GPP	Very High	COTS	N.A.	Low-Medium
Transmuter	Very High	COTS	$<$10 cycles	High

Table 2.1: Key characteristics of Transmuter and comparable alternatives.

Transmuter differs from existing solutions that use gate-level reconfigurability (FPGAs), in addition to pipeline-level reconfigurability used in most coarse-grained reconfigurable arrays (CGRAs). Notably, Transmuter selectively reconfigures the on-chip memory type, resource sharing patterns, and dataflow, at a coarser granularity than contemporary CGRAs. From a hardware and energy-efficiency perspective, targeted reconfiguration limits its logic overhead, and also eases the development of downstream software toolchains. Downstream software benefits similarly from the use of general-purpose cores with a standard ISA as compute units. As a result, Transmuter 's reconfigurable hardware enables runtime reconfiguration within 10s of nanoseconds, faster than existing CGRA and FPGA designs. A qualitative comparison of key characteristics for Transmuter and existing architectures are listed in Table 2.1.

From a software perspective, Transmuter provides application and developer interfaces at higher levels of abstraction (*e.g.*, compared to hardware description language). The software stack exposes two layers: (i) a C++ intrinsics layer that compiles directly for the hardware using a commercial off-the shelf (COTS) compiler, and (ii) a drop-in replacement for existing high-level libraries in Python, called TransPy, that exposes optimized Transmuter kernel implementations to an end-user. Transmuter incorporates an end-to-end software solution, providing both low-level and high-level APIs for kernel and application development, demonstrating the feasibility of the approach.

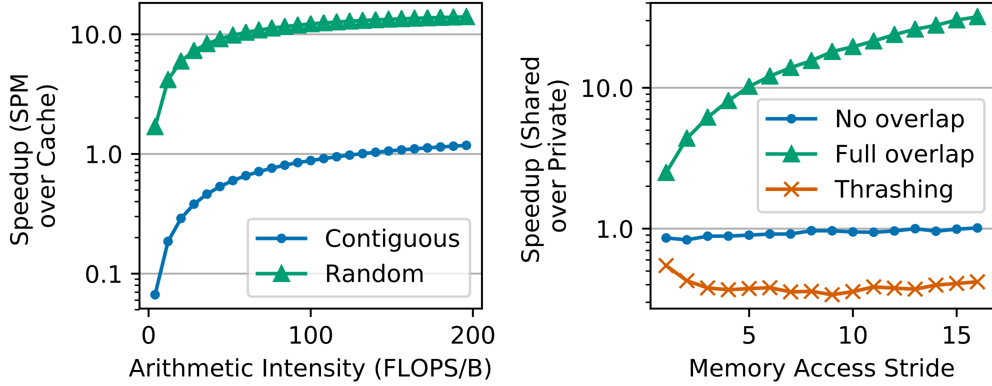


Figure 2.2: Performance disparity on synthetic workloads for simulated multi-core systems with different on-chip memory type, and different memory scopes. **(left)** Speedup of scratchpad memory (SPM) over iso-capacity cache in a 1-core system, under contiguous and random access patterns; SPM and cache are provided the same stream of load/store addresses for a given access pattern. **(right)** Speedup of 1 shared memory over 8 private memories in an 8-core system, under data access patterns with different levels of overlap.

2.2 Motivation

Emerging workloads are often heterogeneous with respect to arithmetic and data access characteristics. For instance, neural network models may entail a mix of dense and sparse linear algebra, while graph problems heavily depend on irregular index structure traversals. These differences are crucial, determining when programs become compute- or memory-bound, and how kernels interact with hardware structures (such as on-chip memory banks). For conventional architectures that incorporate static hardware, runtime variation in compute and access patterns represent missed optimization opportunities.

To understand the interactions above, we designed synthetic kernels with varying levels of arithmetic intensity, access stride, and data overlap, and measured their performance against potential on-chip memory configurations (Figure 2.2). In particular, cache and scratchpad are the predominant types of on-chip memory used in programmable architectures, but offer distinct tradeoffs [16, 17]. The partitioning of on-chip memories – *e.g.*, into groups that are shared between cores or private to a single core – also has significant performance implications.

Figure 2.2 (left) illustrates that performance disparity between scratchpad memory (SPM) and cache can vary by up to ~10×, depending on the access pattern and arithmetic intensity. Regardless of access pattern, SPM speedup over cache is drastically reduced if arithmetic intensity is low, largely due to SPM buffering overhead. SPM buffering overhead is amortized as arithmetic intensity increases, implying that SPM is more useful if algorithmic data reuse can be found. Under random (irregular) accesses, iso-capacity cache incurs significant penalties compared to SPM due to cache granularity, imperfect caching, and resulting evictions. On the other hand, SPM offers little to no advantage under contiguous (regular) accesses since cache miss penalties are largely eliminated; if accesses are random *and* arithmetic intensity is low, SPM incurs significant slowdown.

Figure 2.2 (right) shows the speedup of shared cache over private cache, for different access patterns and access strides. If loads/stores from different cores do not overlap and access distinct cache lines, shared and private caching offer similar performance; accesses are distributed across memories in either case. However, in the case of full overlap, shared caching offers significant benefits up to 1-2 orders of magnitude, since data is reused across cores. The 'Thrashing' access pattern illustrates the pitfall of shared caching – shared caching is vulnerable to cache conflicts across cores (*i.e.*, cache pollution), while private caching is free of this problem by design.

The experiments above highlight the potential advantages (and pitfalls) that any single, static memory configuration provides. We claim that it is advantageous to eschew static hardware assumptions, and instead provide hardware suited to varying application needs via reconfiguration.

2.3 Hardware Design

Transmuter (Figure 2.3) is a massively parallel tiled architecture, composed of energy-efficient in-order cores and a two-level memory hierarchy. A single Transmuter chip may contain one or more clusters that interface to HBM stack(s) in a 2.5D configuration, en-

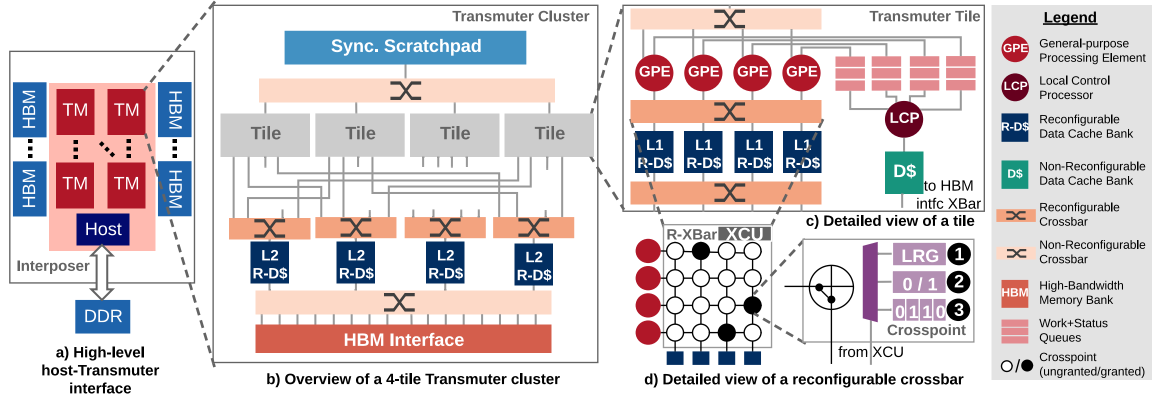


Figure 2.3: Overview of the Transmuter hardware design. **(a)** Top-level view of a Transmuter system including host and DRAM interfaces. **(b)** A single cluster containing 4 tiles with associated L2 R-DCaches and L2 R-Xbars. Crossbar interfaces to HBM are non-reconfigurable. **(c)** Contents of a single tile, containing GPE cores, the LCP core, LCP-GPE work/status queues, L1 R-DCaches, and L1 R-XBars. Arbiters and instruction caches are omitted. **(d)** Microarchitecture of an R-XBar; circled numbers indicate the mode of operation (①: Arbitrate with LRG, ②: Transparent, ③: Rotate).

abling scalability up to GPU-like system sizes. Tiles within clusters connect to main memory through two levels of reconfigurable caches (R-DCaches, Section 2.3.2) and reconfigurable crossbars (R-XBars, Section 2.3.3). The network of R-DCaches and R-XBars provide fast reconfiguration of the on-chip memory type, resource sharing pattern, and dataflow (Section 2.3.1). As discussed in Section 2.1, restricted reconfigurability in the R-DCaches and R-XBars is a key aspect of the design that enables workload adaptation with low overhead. Similarly, the abundance of in-order cores as general-purpose processing elements (GPEs, Section 2.3.4) maximally exploit workload parallelism, with minimal overhead compared to conventional processors.

2.3.1 Reconfigurable Modes

Transmuter contains coarse-grained reconfigurable resources that enable emulation of multiple architectural configurations, which may cater to different workloads. Specifically, reconfiguration in the L1 and L2 – detailed later in Section 2.3.2 and Section 2.3.3 – allow for different on-chip memory type (cache, scratchpad, or FIFO), resource sharing (shared

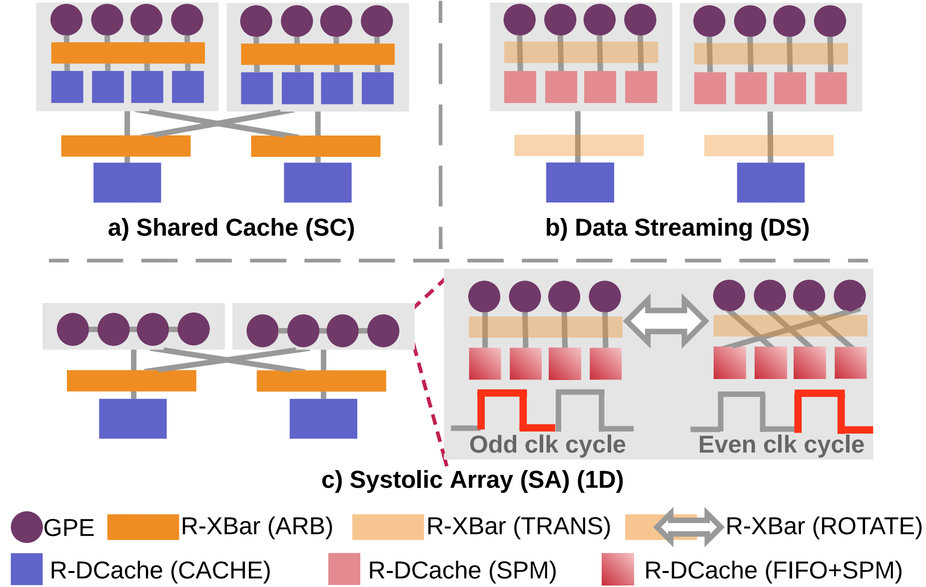


Figure 2.4: Overview of the three Transmuter configurations evaluated in this work. **(a)** Trans-SC: L1 shared cache with L2 shared cache. **(b)** Trans-PS: L1 private SPM with L2 private cache. **(c, d)** Trans-SA: L1 systolic array with L2 private cache.

or private), and dataflow (demand-driven or spatial). While reconfiguration of the L1 and L2 allow for 8 modes each and 64 modes in total, we identify and evaluate 3 distinct configurations (Figure 2.4); these configurations are described as follows.

Shared Cache (Trans-SC): L1 and L2 as shared cache. Trans-SC is most similar to the cache hierarchy of existing multicore CPUs. R-XBars in the L1 and L2 levels arbitrate accesses to tile-internal and tile-external R-DCache banks, respectively. This configuration maximizes cache hit rates under regular data access patterns.

Private Scratchpad (Trans-PS): L1 as private scratchpad and L2 as shared cache. Trans-PS reconfigures L1 cache banks in each tile to be core-private, while the L2 is shared cache (*i.e.*, the same L2 configuration as Trans-SC). This configuration is suited for data access patterns that would otherwise induce cache thrashing, for instance when data structures are partitioned across cores without overlap. Private L2 banks in Trans-PS promote occasional reuse of shared secondary data structures, or victim data that may be spilled.

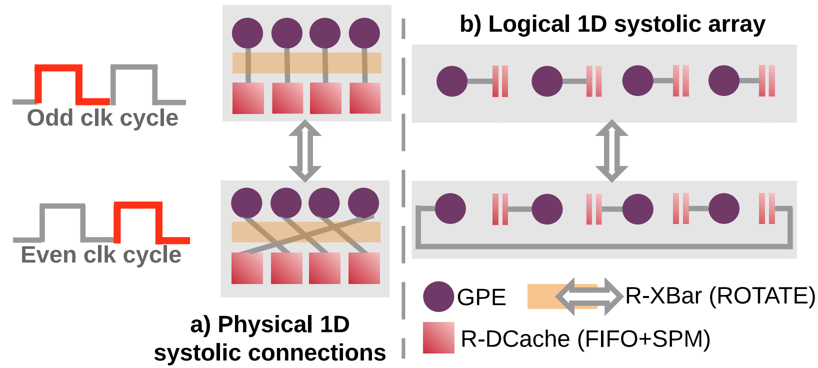


Figure 2.5: Illustration of crossbar ROTATE functionality: **(a)** Physical connections between cores and R-DCache banks are time-multiplexed and "rotate" every cycle. **(b)** Alternating port connections provide functional behavior equivalent to spatial dataflow.

Emulated Systolic Array (Trans-SA): L1 as FIFO buffers and L2 as private cache. Trans-SA provides a view of FIFO-buffered ports in the R-XBars and R-DCaches to emulate either a 1D or 2D systolic array. Finite-state-machines in the R-XBars time-multiplex connections between cores and R-DCache FIFOs on a per-cycle basis, thereby providing spatial array functionality. This configuration is suited for applications that exhibit minimal workload imbalance, and are amenable to mapping onto spatial hardware.

The three configurations above are chosen to be well-suited to the SDH kernels presented in Figure 2.1. For example, while L1 as shared cache and L2 as private cache is functionally valid, the configuration is largely redundant (and often sub-optimal) compared to Trans-SC. Similarly, we find little benefit to using the L2 as scratchpad in conjunction with either L1 scratchpad or cache, largely due to greater L2 access latencies. Thus, we find that the three modes above provide sufficient functional coverage over useful behavior without exposing all possible modes to software.

2.3.2 Reconfigurable Data Cache

Transmuter has two layers of multi-banked memories called reconfigurable data caches, *i.e.* R-DCaches [Figure 2.3(b, c)]. Each R-DCache bank supports cache functionality, in addition to enhancements to support the following modes of operation:

- **CACHE**: Each bank is accessed as a non-blocking, write-back, write-no-allocate cache with a least-recently used (LRU) replacement policy. The banks are interleaved at set-granularity, and a cache line physically resides in one bank. Additionally, this mode uses a simple stride prefetcher to boost performance for regular kernels.

- **SPM**: The tag array, set-index logic, prefetcher and MSHRs are power gated and the bank is accessed as a scratchpad.

- **FIFO + SPM**: A partition of the bank is configured as SPM, while remaining partitions are accessed as FIFO queues using a set of head/tail pointers.[2] This mode is used to implement spatial dataflow in Trans-SA (Figure 2.4).

2.3.3 Reconfigurable Crossbar

A fully-digital multicasting $N \times N$ crossbar creates one-to-one or one-to-many connections between N source and N destination ports. We augment the crossbar design with a Crosspoint Control Unit (XCU) that enables reconfiguration by programming individual crosspoints.

A block diagram of a reconfigurable crossbar (R-XBar) is illustrated in Figure 2.3(d). R-XBars support the following modes of operation:

- **ARBITRATE**: This R-XBar mode is used in Trans-SC. Any source port can access any destination port, and contended accesses to the same port are serialized. Arbitration completes in a single cycle using a least-recently granted (LRG) policy [18], while the serialization latency varies from 0 to $(N-1)$ cycles.

- **TRANSPARENT**: Trans-PS (in L1 and L2) and Trans-SA (in L2) employ R-XBars in TRANSPARENT mode. Crosspoints within the crossbar are fixed to 0 or 1 [Figure 2.3(d)], *i.e.*, a requester can only access its corresponding private resource. Thus, the R-XBar is "transparent" and incurs no arbitration or serialization in this mode.

[2] The queue depth is configurable over memory-mapped registers.

- **ROTATE**: Trans-SA relies on the ROTATE mode, where the R-XBar cycles through a set of pre-programmed 1-to-1 port connections. Similar to the TRANSPARENT mode, ROTATE incurs no arbitration cost. Figure 2.5 illustrates how time-multiplexing of ports is used to achieve 1D spatial dataflow in Trans-SA.

There are two L1 R-XBars within a tile [Figure 2.3(c)]. The upper R-XBar enables GPEs to access the L1 R-DCache, while the lower R-XBar (interfacing with the tile-external crossbar) provides alternate routes to reduce L1-L2 congestion under load.

2.3.4 Processor Cores

A general-purpose processing element (GPE) is a standard ISA scalar processor with a single floating point (FP) unit and single load/store (LS) unit. Its small footprint enables Transmuter to incorporate hundreds to a few thousand GPEs within current reticle sizes. The large number of GPEs, coupled with MSHRs in the cache hierarchy, allows Transmuter to maximally exploit memory-level parallelism across cores. GPEs operate in a MIMD/SPMD fashion, and have per-core (private) instruction caches (I-caches).

GPEs are grouped into tiles and are coordinated by a local control processor (LCP) via work-status queues. Each LCP has private D- and I-caches that connect to the HBM interface. The LCP is primarily responsible for distributing work across GPEs, using either static or dynamic scheduling, thus trading-off code complexity for work-balance.

2.3.5 Host-Device Relationship

A Transmuter accelerator is treated as a slave device, and is hosted by a separate application-class processor. Device and host main-memory (HBM and DDR4, respectively) are distinct, and transfers between the two are orchestrated with hardware direct memory access (DMA). The host is responsible for executing serial/latency-critical work, while parallelized kernels are dispatched to Transmuter.

2.4 Software Interfaces

We implement a software stack for Transmuter in order to support flexible programming and ease-of-adoption of our solution. The software stack has two main components: a high-level Python application programming interface (API), and lower-level C++ APIs for the host, LCPs, and GPEs. The highest-level API, called TransPy [19], is a drop-in replacement for NumPy [20], the well-known high-performance Python library. To swap in Transmuter-accelerated function calls, developers can simply change Python import statements from `import numpy as np` to `import transpy.numpy as np`. TransPy also contains drop-in replacements for SciPy [21], PyTorch [22], NetworkX [23], and other common libraries used in scientific and numerical applications. TransPy binds to underlying accelerated kernels with pybind11 [24], effectively hiding software integration complexity, while individual kernels are implemented with the C++ API layer.

Transmuter kernels are developed by library writers with the aid of the C++ API layer. In order to develop an SPMD kernel (MIMD is discussed in Section 2.6), three programs are required: one each for the host, LCP, and GPE. Host code primarily orchestrates kernel launches and is written in the style of OpenCL [25]. LCP and GPE code comprise the bulk of kernel, and are written in SPMD-style parallel code that utilizes C++ intrinsics for core/thread identification. Other notable C++ API methods used in GPE and LCP code include those for SPM and FIFO access, cache control, and memory reconfiguration.

The Transmuter software stack facilitates the use of accelerator hardware by end-users without exposing hardware details of reconfiguration. At the same time, the C++ layer provides expert programmers with the freedom to develop custom kernel implementations if needed. In the future, library developers may automatically generate kernel code from architecture-specific intermediate languages (Chapter IV), or leverage recent program synthesis and rewriting techniques [26–28].

2.5 Test Kernels

Transmuter was evaluated using constituent kernels from the workloads characterized in Section 2.1. Evaluation kernels were implemented in C++ (using the C++ API layer discussed in Section 2.4), hand-optimized for each Transmuter configuration, and compiled at the `-O2` optimization level using the Arm GNU compiler. Data transfer and initialization times are excluded for all platforms. All experiments used single-precision floating point. Throughput is reported in FLOPS/s, and arithmetic op counts are derived analytically to account for useful (algorithmic) work.

The following describes the kernels and their mappings onto Transmuter hardware.

GeMM and GeMV. GeMM and GeMV are both dense linear-algebra kernels with high regularity. Both kernels exhibit contiguous data, and computation that can be evenly partitioned across cores. GeMM implementations apply standard on-chip memory blocking optimizations [29, 30], while GeMV does not apply blocking due to lower reuse. The Trans-SC kernel implementation blocks data in the cache, while the Trans-PS implementation blocks partial results in private L1 SPMs. For Trans-SA the GPEs stream rows of the first operand matrix through the L2 cache, while columns of the second operand matrix are loaded from the L1 SPM.

Conv. Conv cross-correlates OC number of $F \times F \times IC$ filters over an $N \times N \times IC$ image with stride S, where IC and OC are the number of input and output channels, respectively. The filter is reused while computing one output channel, and across multiple images. For Trans-SC, we assign each GPE to compute the output of multiple rows to maximize filter reuse. For Trans-PS and Trans-SA, we partition each image into $B \times B \times IC$ sub-blocks, such that the input block and filter fit in the private L1 SPM; each block is then mapped to a different GPE. Trans-SA uses a row-stationary approach similar to Chen et al. [31], where blocks are mapped to a set of F adjacent processors.

SpMM. SpMM is a memory-bound kernel with low FLOPS that decrease with increasing sparsity.[3] Sparse storage formats lead to indirection and thus irregular memory accesses [11, 32]. We implement SpMM in Trans-SC using a prior outer product approach [11, 33]. In the multiply phase of the algorithm, the GPEs multiply a column of A with the corresponding row of B, such that the row elements are reused in the L1 cache. In the merge phase, a GPE merges all the partial products corresponding to one row of C. Each GPE maintains a private list of sorted partial results and fills it with data fetched from off-chip. Trans-PS operates similarly, but with the sorting list placed in private L1 SPM, given that SPMs are a better fit for operations on disjoint memory chunks. Lastly, SpMM in Trans-SA is implemented following a recent work that uses sparse packing [34].

SpMV. SpMV, similar to SpMM, is bandwidth-bound and produces low FLOPS.[4] We exploit the low memory traffic in the outer product algorithm for sparse vectors, mapping it to Trans-SC and Trans-PS. The GPEs and LCPs collaborate to merge the partial product columns in a tree fashion, with LCP 0 writing out the final elements to the HBM. SpMV on 1D Trans-SA is implemented using inner product on a packed sparse matrix as described in Kung et al. [35]. The packing algorithm packs 64 rows as a slice and assigns one slice to each 1×4 sub-tile within a tile. Each GPE loads the input vector elements into SPM, fetches the matrix element and performs MAC operations, with the partial results being streamed to its neighbor within the sub-tile.

FFT. FFT in 1D computes an N-point discrete Fourier transform in $log(N)$ sequential stages. Each vertical stage of the FFT graph consists of $N/2$ butterfly operations. FFT applications often operate on streaming input samples, and are highly-amenable to spatial dataflow architectures [36, 37]. While butterfly computations are deterministic, dependencies between FFT stages are challenging to implement efficiently with the SPMD model.

[3] $2N^3r^2$ for uniform-random $N \times N$ matrices with density r.

[4] $\sim 2N^2rr_v$ for a uniform-random $N \times N$ matrix with density r, and vector with density r_v.

In particular, frequent thread synchronization and irregular control-flow introduce significant overhead. Thus, in addition to Trans-SC and Trans-PS SPMD implementations, we develop a control-free Trans-SA MIMD FFT implementation that leverages code generation (Section 2.6).

2.6 FFT Kernel Generation

In addition to direct implementation of kernels in high-level languages, deeper optimizations can be exploited with kernel-specific code generation. With the generator-based approach, kernel writers develop a *meta-programming* system to control the emission of optimized kernel code, instead of writing it by hand. For Transmuter we leverage a simple template-based meta-programming flow to generate control-free, statically optimized FFT kernels. By simplifying control-flow and statically specializing kernel code a per-core basis, generated kernels yield up to 20$\times$ speedup over comparable implementations.

2.6.1 Model-Specific Mappings

Before describing the details of the kernel generator, we first give an overview of how FFT is mapped using Transmuter parallel programming models. The following describes the mappings onto Transmuter under the SPMD model and Trans-SA MIMD model.

SPMD Mappings. The kernel mappings for Trans-SC and Trans-PS resemble a GPU SIMT implementation, where control-flow and data partitioning are determined by thread indices. The sequence of vertical stages is computed sequentially (i.e., with thread-synchronization between each stage), while individual stages are parallelized. Butterflies within each stage are statically partitioned evenly among GPEs; the LCPs assign inputs and collect outputs. Trans-PS uses the L1 SPM to store partial results but the Trans-SC and Trans-PS kernels are otherwise the same, with identical scheduling and work partitioning schemes. Look-up-tables are used to store twiddle coefficients when the transform size is small enough

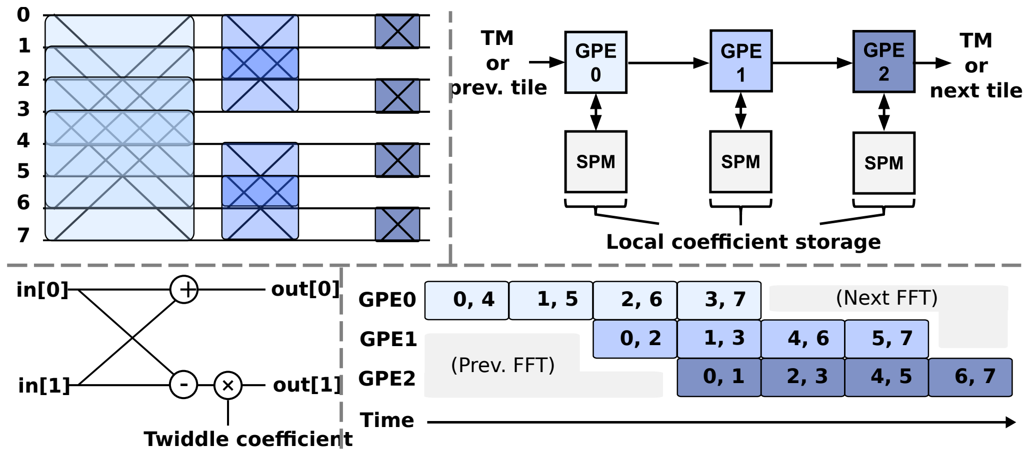


Figure 2.6: Summary of radix-2 FFT implementation: **(top)** Algorithmic partitioning of vertical FFT stages onto processors. **(bottom)** Resulting pipeline of butterfly computations across processors.

to fit in on-chip memory. When the transform size becomes large, runtime twiddle coefficient generation is used to reduce memory-related overheads at the expense of added computation.

MIMD-Optimized Trans-SA Mapping. The scheduling and dataflow for the Trans-SA mapping is similar to the radix-2 multipath-delay-commutator (R2-MDC) FFT algorithm [38]. With the R2-MDC dataflow, an entire vertical stage is assigned to a single GPE, and every GPE immediately transmits butterfly operands using L1 FIFO buffers. Like the SPMD mappings, the Trans-SA mapping also leverages runtime twiddle coefficient generation. In contrast to the SPMD mappings – where computations from different vertical stages do not overlap – the MIMD kernel uses a fully-pipelined schedule (Figure 2.6). Pipelining the schedule of computations ensures high utilization, since butterfly operations can proceed greedily as soon as operands are available. The MIMD-optimized mapping also does not rely on thread indices in the kernel implementation to partition work, since cores execute independent programs.

2.6.2 Limitations of Explicit Programming

While most Transmuter kernels are implemented by hand with the SPMD programming model, the FFT Trans-SA MIMD kernel leverages meta-programming infrastructure. The need for meta-programming (*i.e.*, code generation) in this case is largely driven by performance and pragmatism. For instance, compared to the SPMD model where a single program is sufficient to specify program behavior, the MIMD model entails writing N variants of the same kernel, where N can range up to the number of cores in the system.[5] The following summarizes additional key factors, based on experience with Transmuter kernel development and tuning:

(i) SPMD inefficiency: explicitly specializing the program behavior of cores or threads at fine grain to improve performance is counter-productive in SPMD and SIMT programs. For instance, avoiding redundant computations across cores by incorporating fine-grained core index-based control appears desirable, since the scheme skips computation. However, in our experience such core-/thread-specific runtime specialization under SPMD/SIMT often introduces overheads (*e.g.*, conditional branching and predication) that outweigh purported benefits.

(ii) MIMD verbosity: canonical MIMD programs are permissive and enable differentiated behavior across threads or cores, but expressing differentiated behavior tends to require more code. This contrasts with canonical SPMD and SIMT programs, where the behavior of parallel computation is expressed within a single unit of code. Verbosity makes it harder to create, optimize, and maintain sophisticated kernels.

(iii) Conservative compilation: general-purpose compilers apply code transformations and optimizations conservatively to guarantee correctness. Unless implementations are carefully written and tuned to produce efficient machine code (often leading to greater verbosity), important transformations that rely on static analysis may not be fully exploited.

[5] The number of "kernel variants" required under a MIMD model is architecture and implementation dependent, but roughly proportional to the number of divergent control-flow paths in an SPMD implementation that uses the same algorithm mapping.

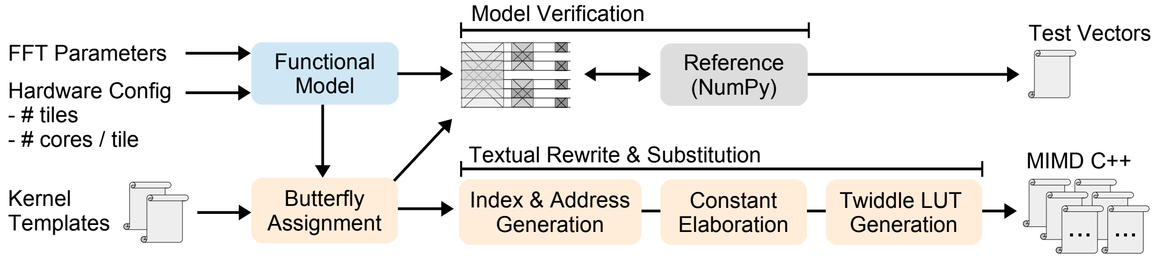


Figure 2.7: Overview of FFT kernel generation.

2.6.3 MIMD Kernel Generation

Compared to standalone hand-written kernel implementations, kernel generation improves abstraction, maintainability, implementation leverage, and enables a variety of critical optimizations (Figure 2.7). In particular, generator automation facilitates the emission of MIMD code that is specialized per-core, and allows for statically multi-versioned kernels. Kernel multi-versioning is critical for downstream compiler optimizations related to constant propagation, dead-code elimination, and loop unrolling. MIMD specialization and kernel multi-versioning both contribute to reduced code size, reduced branch density, and overall improvement in compute throughput.

2.7 Modeling and Experimental Methodology

This section describes the methodology to derive Transmuter performance, power and area, the experimental setup, and baseline comparisons.

Performance Models

We used the gem5 cycle-level simulator [39, 40] to model Transmuter performance. Simulation parameters for key modules are listed in Table 2.2. The GPE and LCP cores use a parameterized gem5 `MinorCPU` pipeline, with parameters set to resemble a typical in-order scalar core with basic DSP support. Cache and crossbar latencies were based on prior prototype designs [18, 41, 42].

Table 2.2: Microarchitectural parameters for gem5 simulations.

Module	Microarchitectural Parameters
GPE/LCP core	1-issue, 4-stage, in-order core @ 1.0 GHz, 3-cycle pipelined integer & floating-point FUs, 9-cycle divide
Work/Status Queue	4 B, 4-entry FIFO buffer between each GPE and LCP within a tile
R-DCache (per bank)	CACHE: 4 kB, 64 B block size, 4-way set-associative, 1-ported, non-coherent, 8 MSHRs, stride prefetcher with degree 2 and 1-cycle latency SPM: 4 kB, 1-ported, physically-addressed, word-granular FIFO+SPM: 4 kB, 1-ported, physically-addressed, 32-bit head/tail pointer registers
R-XBar	Non-coherent N-by-N crossbar with 1-cycle response ARBITRATE: 1-cycle arbitration latency, serialized at conflicts TRANSPARENT: no arbitration, direct access ROTATE: alternate port connections at programmable intervals L1 R-XBar width: 32 address + 32 data bits L2 R-XBar width: 32 address + 128 data bits
GPE/TM I-cache	4 kB, 64 B block size, 4-way set-associative, 1-ported, non-coherent, 8 MSHRs
Sync. SPM	4 kB, 1-ported, physically-addressed scratchpad
Memory	1 HBM2 stack, 16 64-bit pseudo-channels @ 8000 MB/s, 80-150 ns average access latency

The resource requirement for running detailed timing simulations with the standalone gem5 model is only tractable for Transmuter systems up to 8× 16. For larger systems, we substitute the gem5 cores with trace replay engines while retaining the gem5 model for the rest of the system. Offline traces are generated on a native machine and replayed with trace-based timing; this allows scaling of Transmuter size up to one 64× 64 cluster in simulation. On average, across the evaluated kernels, the trace-driven model is pessimistic to 4.5% of the execution-driven (detailed) model. To model multi-cluster system sizes, we use analytical models from gem5-derived bandwidth and throughput scaling data.

Power and Area Models

We designed RTL models for Transmuter hardware blocks and synthesized them. For the R-XBar, we use a synthesizable version of the design proposed by Satpathy et al. [18], augmented with an XCU. The R-XBar power model is taken from synthesis reports, and calibrated against the data reported by Satpathy et al. [18, 43]. The R-DCaches are cache modules enhanced with SPM and FIFO control logic. For the caches and sync SPM, we used CACTI 7.0 [44] to estimate the dynamic energy and leakage power. Power and area for GPEs and LCPs are based on the Arm Cortex-M4/M4F specification document, while area for other blocks are derived from synthesis estimates. Finally, we cross-referenced our power estimate for SpMM on Transmuter against a prior SpMM ASIC prototype [45] and obtained a pessimistic deviation of 17% after accounting for architectural differences.

Baselines and Comparisons

We compared Transmuter with a high-end Intel Core i7 CPU and an NVIDIA Tesla V100 GPU, both using software kernels from optimized commercial libraries (Table 2.3). For comparable evaluations we collected results for two different Transmuter designs, `TransX1` and `TransX8`, which are similar in terms of area to the CPU (~120mm^2) and GPU (~800mm^2), respectively. `TransX1` has a single 64× 64 Transmuter cluster while `TransX8`

Table 2.3: Hardware and software comparison baselines.

Platform	Hardware Detail	Kernel Libraries
CPU	Intel i7-6700K (14nm), 4 cores/8 threads @ 4.2 GHz, 16 GB DDR3 @ 34.1 GB/s, AVX2, SSE4.2	MKL 2018.3.222 (GeMM, GeMV, SpMM, SpMV), DNNL 1.1.0 (Conv), FFTW 3.0 (FFT)
GPU	NVIDIA Tesla V100 (12nm), 5120 CUDA cores @ 1.25 GHz, 16 GB HBM2 at 900 GB/s	cuBLAS v10 (GeMM, GeMV), CUSP v0.5.1 (SpMM), cuSPARSE v8.0 (SpMV), cuDNN v7.6.5 (Conv), cuFFT v10.0 (FFT)

employs eight 64× 64 clusters with one HBM2 stack per cluster in both designs.

We used standard profiling tools to measure GPU and CPU power, namely `nvprof` and `RAPL`. For fairer comparison with the GPU, we estimated HBM access energy [46], measured memory bandwidth, and subtracted HBM power from the value reported by `nvprof`. Power for baselines is scaled for iso-technology comparison using quadratic scaling.

2.8 Results

This section presents experimental results for Transmuter. Using the modeling methodology described in Section 2.7, we evaluate Transmuter configurations (Trans-SC, Trans-PS, and Trans-SA) across different system sizes. We analyze the performance and energy impact of architectural choices, and interpret comparisons against the CPU and GPU baselines. This section also discusses the impact of kernel generation, using observations from the MIMD FFT kernel.

Power and Area

Table 2.4 lists the power and area for one Transmuter cluster in 14 nm, resulting from the models described in Section 2.7. Static (leakage) power constitutes the majority of total power at 8 W (60% of total), largely due to instruction and data caches (~32 MiB SRAM in total for I-/-DCaches). Meanwhile, dynamic power is dominated by the GPE cores and

Table 2.4: Breakdown of power and area for one 64× 64 Transmuter cluster.

	Power (mW)			**Area**
	Static	**Dynamic**	**Total**	**(mm^2)**
GPE Cores	361.3	2380.5	2741.7	28.9
LCP Cores	5.6	22.5	28.1	0.4
Sync. SPM	0.6	0.1	0.6	0.1
ICaches	2566.6	373.6	2940.1	25.7
LCP DCaches	39.5	0.9	40.4	0.5
L1 R-DCaches	2527.1	204.0	2731.0	30.7
L2 R-DCaches	37.4	18.3	55.7	0.5
L1 R-XBars	1757.8	2149.3	3907.1	30.3
L2 R-Xbars	36.9	14.8	51.7	0.8
Muxes/Arbiters	581.9	87.6	669.5	0.7
Memory Ctrls.	47.5	129.0	176.4	5.5
Total	**8.0 W**	**5.4 W**	**13.3 W**	**124.1**

crossbars, which reflects the high switching activity that is anticipated for those modules. The estimated total power for a single TransX1 cluster in 14 nm CMOS is 13.3 W, with an active area of 124.1 mm^2; this makes the die size of TransX1 and TransX8 roughly comparable to the CPU and GPU baselines (Table 2.3), respectively. The worst-case reconfiguration energy is less than 266 nJ (10 cycles) – an order of magnitude lower than the energy (and latency) required by existing reconfigurable architectures.

Impact of Reconfiguration

Figure 2.8 shows the performance across kernels and kernel-specific input sizes for the three evaluated Transmuter modes, using a small 2× 8 system. Notably, the results show that the best performing mode is both kernel-dependent and input-dependent. This is a critical result that illustrates how static hardware is insufficient to obtain optimal performance, and why dynamic reconfiguration is beneficial.

We now analyze the performance of individual kernels and their interaction with different Transmuter configurations.

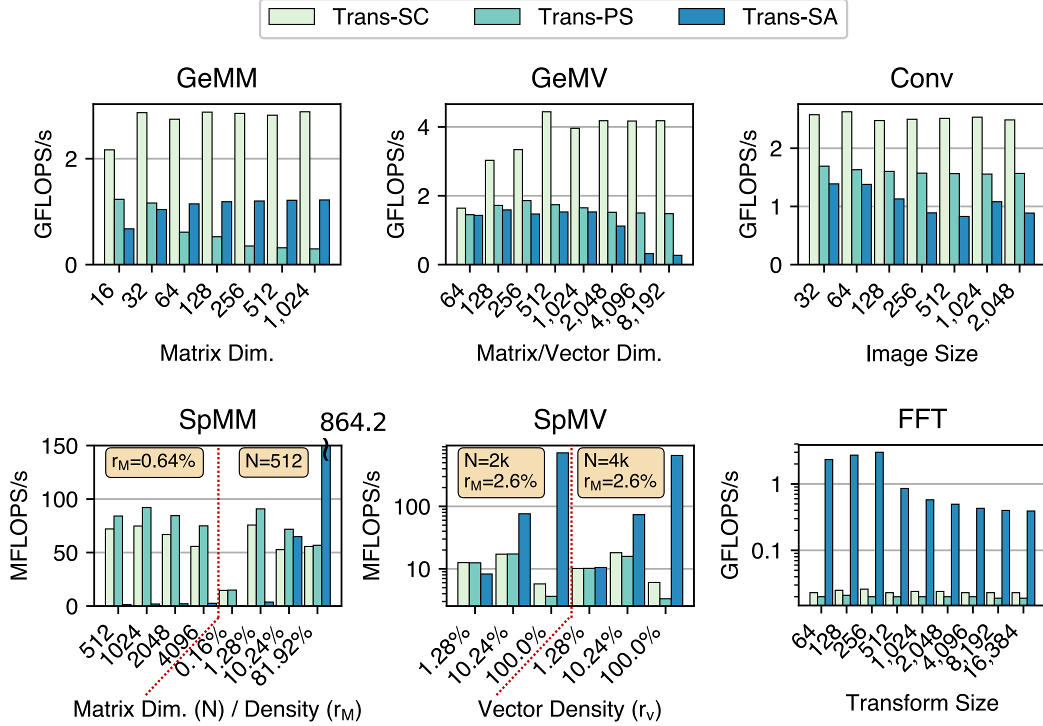


Figure 2.8: Performance comparison across kernels and input sizes for Trans-SC, Trans-PS, and Trans-SA Transmuter configurations.

Dense kernels. For GeMM and GeMV, Trans-SC achieves high L1 hit rates (>99%), with efficient blocking and shared memory that leads to high data reuse. In contrast, Trans-PS exhibits a large fraction of L2 cache capacity misses that impact performance. Trans-SC also performs consistently better than Trans-SA, as it does not incur the overhead of explicit L1 SPM buffering (*i.e.*, scratchpad copy loops). For Conv, as with GeMM/GeMV, Trans-SC performs the best due to a regular access pattern with sufficient filter and input reuse. Across the dense kernels, stride prefetching in Trans-SC is sufficient to capture regular access patterns.

Sparse kernels. For SpMM, the multiply phase of outer product is better suited to Trans-SC as rows in the second operand matrix are shared. The merge phase is amenable to Trans-PS, since private SPMs are not susceptible to cache trashing that otherwise occurs when merging disjoint element lists. Trans-SA obtains the best performance for matrix densities greater than ~11%, but performs poorly in comparison to outer product for highly-

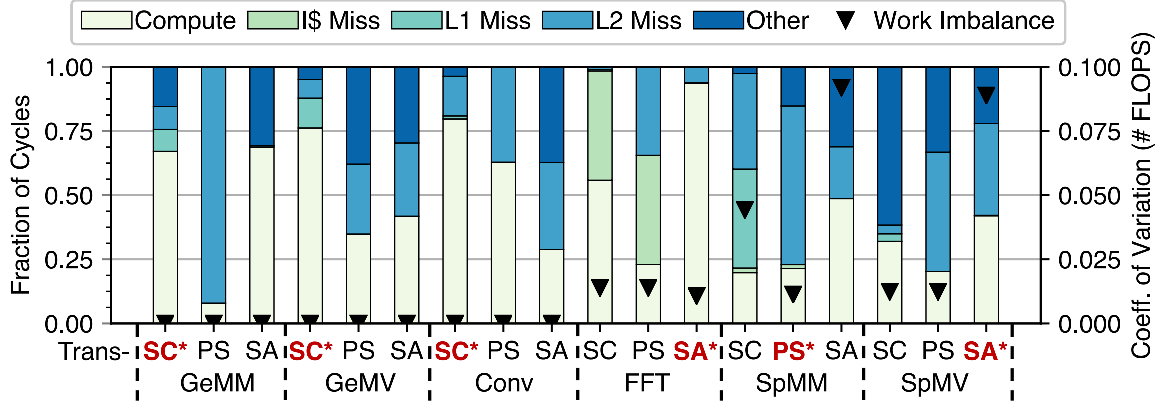


Figure 2.9: Performance attribution: (left axis) attribution of cycles for different architectural events, and (right axis) work imbalance across cores.

sparse matrices. For SpMV, performance depends on the input matrix size, dimensions, as well as the vector density. SpMV with Trans-SA benefits from spatial dataflow, due to dense vectors that can be streamed regularly to GPEs, while SpMM does not. As vector sparsity increases, the number of zero elements fetched by the Trans-SA sparse-packing algorithm increases, and outer product on Trans-SC/Trans-PS outperforms Trans-SA (as with SpMM). For Trans-SA with highly-sparse inputs, the work imbalance (Figure 2.9) exhibited by SpMM and SpMV suggests that the sparse-packing algorithm is unable to keep GPEs utilized, resulting in lower throughput.

FFT. For FFT, inter-GPE synchronization and coherence handling in the SPMD kernel implementations limit the performance achievable by Trans-SC and Trans-PS. In contrast, the MIMD Trans-SA kernel achieves significantly higher throughput compared to Trans-SC and Trans-PS SPMD kernels, and exhibits ~10× lower memory bandwidth usage. The large performance disparity above is achievable due to the FFT generator infrastructure (Section 2.6), as we describe in the following section.

Impact of FFT Kernel Generation

Generated MIMD FFT kernels outperform other hand-written Transmuter SPMD kernels for several key reasons. First, the generator allows for specialized code emission that

either eliminates or optimizes thread synchronization and data communication costs. For instance, bulk-synchronous thread synchronization is eliminated in the generated Trans-SA code since producer-consumer GPE cores communicate directly over FIFO channels; these fine-grained, core-specific communication patterns are emitted by the generator such that nearly zero runtime control overhead is required. The generator also makes it trivial to statically parameterize the MIMD code with optimizations that are input-size dependent. At smaller FFT sizes, MIMD kernels are generated with small, per-core twiddle coefficient look-up-tables. At larger sizes where look-up-table size is prohibitive, the generator instead enables runtime twiddle computation. As a result of efficient generator optimizations, MIMD FFT kernels outperform hand-optimized SPMD implementations by 1-2 orders of magnitude (Figure 2.8). In comparison, the performance range across different Transmuter modes (including Trans-SA) is typically within 1.5-4×, as illustrated by GeMM/GeMV, Conv, and SpMM/SpMV.

Second, MIMD code generation produces smaller per-core binaries and eliminates control indirection that otherwise occurs in SPMD code. As described in Section 2.6.3, generated kernels are statically simplified and individually pre-processed prior to downstream compilation. Downstream compilation for generated kernels is able to produce machine code that is compact, minimal in stack usage, minimal in control and branch density, and highly-dense in terms of compute instructions. As a result, the generated Trans-SA kernels obtains extremely high compute efficiency, while the non-systolic SPMD code is impacted by branchy control flow and expensive cache misses (Figure 2.9). Notably, the compute efficiency exhibited by generated MIMD FFT code (~90%) is greater than all other evaluated kernels. In comparison, the Trans-SC and Trans-PS SPMD implementations are limited to 50% and 25% compute efficiency.

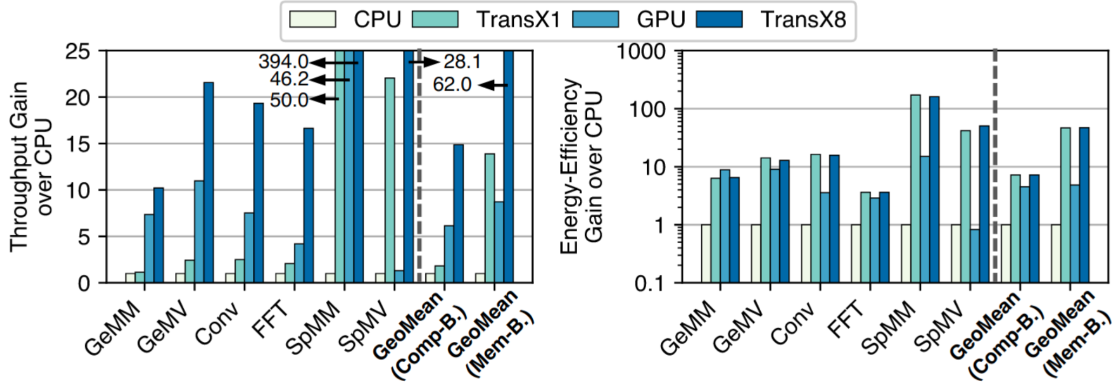


Figure 2.10: Average performance (throughput) and energy-efficiency, normalized to the CPU baseline. Data is averaged across all test inputs for each kernel. GeoMean columns indicate the aggregated geometric mean for compute-bound and memory-bound kernels.

Table 2.5: Estimated speedup for end-to-end applications.

Speedup	**DANMF**	**LSTM**	**Marian**	**MaxCut**	**MFCC**
TransX1 v.s. CPU	4.1×	1.1×	2.2×	6.2×	1.7×
TransX8 v.s. GPU	3.5×	3.8×	2.1×	7.2×	1.6×
	NBSGD	**RolePred**	**SemSeg**	**Sinkhorn**	**VidSeg**
TransX1 v.s. CPU	3.5×	2.7×	2.4×	3.1×	2.2×
TransX8 v.s. GPU	2.8×	2.3×	2.5×	3.0×	2.8×

Comparison with CPU and GPU Baselines

This section presents performance and energy-efficiency comparisons against the baselines introduced in Section 2.7. The TransX1 and TransX8 system sizes are used as comparison points for the CPU and GPU, respectively. Figure 2.10 shows the average performance and energy-efficiency for evaluated kernels, normalized to the CPU baseline. Data from individual kernels is back-annotated into runtimes for the end-to-end SDH applications introduced in Section 2.1, culminating in estimated end-to-end speedups in the range of 1.1-7.2× (Table 2.5). The following provides deeper analysis for individual kernels.

Compute Bound Kernels: GeMM, Conv, and FFT. Compared to the CPU, TransX1 achieves performance improvements of 1.2-2.5× across the group of compute bound ker-

nels. Profiling suggests that the majority of performance improvement with Transmuter results from greater hardware parallelism. Similarly, energy-efficiency improvements range from 3.6-16.3×, in large part due to Transmuter's use of simple in-order cores and non-coherent cache. Compared to the GPU, TransX8 obtains performance improvements of 1.3-2.6×, and energy-efficiency improvements of 0.8-4.4×. The ~20% energy-efficiency loss corresponds to the GeMM kernel, where the GPU is known to be highly optimized. The result with GeMM is within reason, since independent scalar cores, in theory, should incur greater control overhead relative to an iso-compute SIMD counterpart.[6] On FFT, Transmuter obtains significant performance improvement over both the CPU and GPU; interestingly, the 4× performance gain over GPU does not translate proportionally into the energy metric due to efficient FFT batching provided by the `cuFFT` GPU kernel.

Memory Bound Kernels: GeMV, SpMM and SpMV. On GeMV, TransX1 achieves 2.4× greater throughput relative to the CPU; notably, the CPU becomes bandwidth bound for input dimensions beyond 1024 compared to both the GPU and Transmuter. The 14.2× energy-efficiency gain obtained by TransX1 stems from reducing the number of active GPEs to mitigate bandwidth-starvation, thereby saving power. On SpMM and SpMV, performance improvement is highly dependent on the parameters of the inputs (*e.g.*, sparsity), with improvements over the CPU and GPU in the ranges of 4.4-110.8× and 5.9-37.7×, respectively. While Transmuter is memory-bottlenecked for SpMM, SpMV is bounded by the scheduling granularity of packing algorithm deployed on Trans-SA. In comparison, the GPU has 7.2× greater available memory bandwidth but is bottlenecked by thread divergence and synchronization stalls (*i.e.*, due to SIMT side-effects); the GPU achieves just 0.6% and 0.002% of its peak performance for SpMM and SpMV, respectively. Relative to the GPU, Transmuter demonstrates ASIC-level performance gains of 5.9-11.6× by reducing off-chip traffic in the outer product algorithm, and by employing independent scalar cores that are tolerant to divergence.

[6] We don't believe this is a hard and fast rule, since realized overheads are implementation-dependent.

2.9 Related Work

There is a vast body of work that explores the use of reconfigurability in programmable hardware. Although the compute, on-chip memory, and programming model of Transmuter and CGRAs differ significantly, aspects of Transmuter interconnects are similar to those used in CGRAs. Several prior works augment more conventional processor architectures to exploit reconfiguration within core pipelines, caches, or other functional units, in a fashion similar to the reconfiguration in Transmuter.

Reconfigurable Core Pipelines and CMPs. Several prior works have used reconfiguration to augment multi-core processor designs, or emulate flexible versions of them. Smart Memories [47] was among the first to consider reconfigurable processor components. In addition to fine-grained logic reconfiguration within the processor datapath, the on-chip memory design in Smart Memories supported cache, streaming accesses (*i.e.*, FIFO), and scratchpad. Although the Smart Memories hardware resembled a CGRA, Mai et al. [47] demonstrated the potential of reconfiguration by emulating two dissimilar parallel architectures.[7] Compared to Smart Memories, Transmuter employs off-the-shelf cores to maintain software compatibility, and restricts the scope of reconfiguration to limit hardware overheads.

CoreFusion [50], MorphCore [51], and Composite Cores [52] explored reconfiguration within processor datapaths. For example, CoreFusion augmented a conventional OoO processor design to support operation as a single large OoO core, or as multiple smaller dual-issue cores. Similarly, MorphCore and Composite Cores enhance processor datapaths to operate either as big OoO cores or little in-order cores, supported by efficient thread migration. These works are complementary to Transmuter (which does not consider reconfiguration within the processor datapath) but share a common overall goal, *i.e.*, to optimize across software phases by dynamically switching between architectural configurations.

[7] The Imagine stream processor [48], and the Hydra CMP [49].

Coarse-Grained Reconfigurable Arrays. CGRA architectures [10, 53] employ reconfiguration of interconnects and simple functional units (FUs) to support variants of explicit dataflow graph (DFG) execution. Piperench [54] and RSVP [55] are examples of early CGRA designs that use reconfigurable interconnect to form dynamic chains of pipelined FUs. Early CGRA designs did not support indirect or irregular computations, however, and required static DFGs that could be unrolled (and then mapped) onto hardware. Subsequent efforts such as TRIPS [56, 57], Wavescalar [58], and Dyser [12] extended dataflow ideas to general-purpose CGRA-like hardware, with the goal of increasing ILP in superscalar execution.

Recent work such as Plasticine [59, 60], Stream Dataflow [61], and SPU [62], adapt CGRA hardware and programming models to address a broader set of accelerator kernels. Compared to early CGRA designs, these efforts place more emphasis on reconfiguration of control logic within CGRA memories in order to support a variety of memory access patterns, and also target both dense and sparse computations.

Compared to CGRAs and dataflow execution-based architectures, Transmuter uses similar low-level mechanisms for logic and interconnect reconfiguration, but applies them in the context of adaptive many-core acceleration. CGRAs use fine-grained interconnect reconfiguration to form physical pipelines between ALUs and FUs. In contrast, Transmuter interconnect reconfiguration is used in the SPMD/MIMD model to provide different logical views of memory, and to improve interconnect performance. Similarly, Plasticine, Stream Dataflow, and SPU incorporate reconfigurable control logic to support different scratchpad addressing modes, while Transmuter employs reconfiguration of memory control logic to expose multiple types of memory (beyond scratchpad) to software. Finally, reconfiguration in the dataflow and CGRA designs above is strictly necessary to support targeted applications (*i.e.*, a functional requirement); Transmuter does not require reconfiguration, but incorporates broadly-applicable reconfiguration mechanisms to enhance many-core performance and energy efficiency.

2.10 Conclusion

The end of traditional semiconductor scaling is driving the need for alternative compute substrates, but balancing flexibility, performance, and energy-efficiency is a challenge. In this chapter we introduced Transmuter, an energy-efficient many-core architecture designed to accelerate diverse application kernels. Transmuter consists of simple in-order cores connected to a network of reconfigurable caches and crossbars, supporting fast reconfiguration of memory type, resource sharing pattern, and dataflow. Thus, Transmuter is amenable to different workloads with varying resource requirements and memory access patterns. We also showed how Transmuter can be easily used and integrated by application developers, with software APIs that provide entry points at different levels of the library stack. On a benchmark suite with a mix of dense and sparse kernels, Transmuter obtains significant improvements over a baseline CPU/GPU; $46.8\times/9.8\times$ considering the memory-bound kernels, and $7.2\times/1.6\times$ for the compute-bound kernels. Transmuter thereby demonstrates how lightweight interconnect and memory reconfiguration may be used to address the flexibility-efficiency gap in future programmable accelerators.

Chapter III

A Reconfigurable Systolic Multiprocessor

3.1 Introduction

Kernels with diverse computation and data-transfer are ubiquitous in emerging applications, but are challenging to support in general-purpose processors. For instance, sparsity techniques [63, 64] continue to gain adoption for machine learning but rely on complex, irregular data structures. Similarly, irregular algorithms that operate on index structures (such as trees and adjacency formats) are ubiquitous in graph analytics [65, 66] and genomics pipelines [67, 68]. Nevertheless, existing programmable designs lack first-class support for the types of workloads above and fail to exploit their properties.

Multi-core CPUs are the gold-standard in programmability but incur significant overhead for features like speculative out-of-order (OoO) execution and hardware-managed cache coherence. These features are crucial for fully-general software and single-thread workloads, but have limited utility in acceleration contexts. On the other hand, despite being programmed with the SIMT model, GPUs rely on backend SIMD units that amortize control-related overheads. GPU programmers benefit from a SIMT programming model (that lends the illusion of scalar execution), but the underlying SIMD hardware is vulnerable to thread divergence; under irregular workloads in particular, GPUs suffer from underutilization and degraded performance [69]. FPGAs are highly configurable but incur non-trivial hardware overheads in exchange for gate-level reconfigurability. In addition,

FPGAs exhibit long reconfiguration times at microsecond scales [70].

This work describes Versa [71] (Figure 3.1): a general-purpose accelerator that exploits *microarchitectural flexibility* to support diverse algorithms. In contrast to existing designs that incorporate fixed compute, interconnect, and on-chip memory, Versa provides optimized modes for each of the previous that are reconfigurable at *nanosecond* scales. This enables kernel implementations and hardware that are co-optimized for per-algorithm characteristics and dynamic application needs.

This section discusses high-level design choices and introduces the Versa architecture.

3.1.1 High-Level Objectives

An accelerator architecture that addresses the limitations from Section 3.1 should consider the following criteria:

1. Programmability using productive, high-level languages
2. Tolerance to irregular compute and data-access patterns (i.e., no thread divergence)
3. Maximal support for contrasting (potentially unknown) workloads

This work addresses the above with a combination of existing techniques and novel contributions:

1. ARM-based cores, coupled with a robust ecosystem of toolchain and compiler infrastructure
2. Multi-threaded execution backed by scalar core clusters, intrinsically tolerant to divergence
3. Reconfiguration and lightweight pipeline augmentations to support distinct, workload-optimizing hardware modes

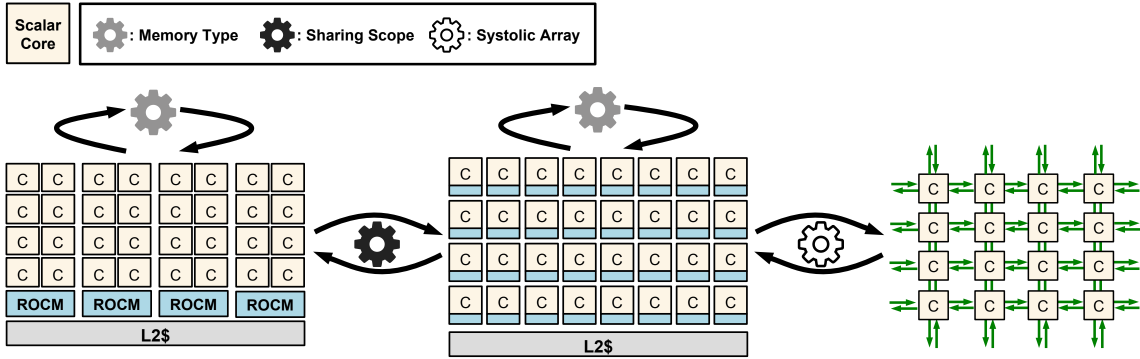


Figure 3.1: Versa exploits reconfiguration of on-chip memory and compute to provide algorithm-optimized hardware characteristics. Based on learnings from Transmuter, Versa evolves the programming model, provides enhanced reconfigurability, and resolves key scalability limitations.

While not listed above, the need for energy-efficiency and performance dictates additional design traits. For instance, while the possible scopes and granularities of hardware reconfiguration are virtually unbounded, reconfiguration that is finer-grained generally incurs greater overhead. The energy-efficiency and performance penalties of fine-grained reconfigurability are illustrated by FPGA designs, as discussed in Section 3.1. Thus, Versa incorporates reconfigurability that is limited to critical functional units, and designed to either reuse existing logic or incur minimal added overhead. Detailed aspects of the hardware design are discussed in the following section.

3.2 Hardware Overview

3.2.1 Compute Tiles

The majority of Versa's energy-efficiency, performance uplift, and research novelty are attributed to features in its compute tiles (Figure 3.2). A tile contains eight ARM Cortex-M4F 'worker' cores that are responsible for the bulk of an algorithmic computation. Workers are single-issue cores equipped with an IEEE 754-compliant single-precision (i.e., FP32) scalar floating-point unit (FPU). The inclusion of the FPU extends the base

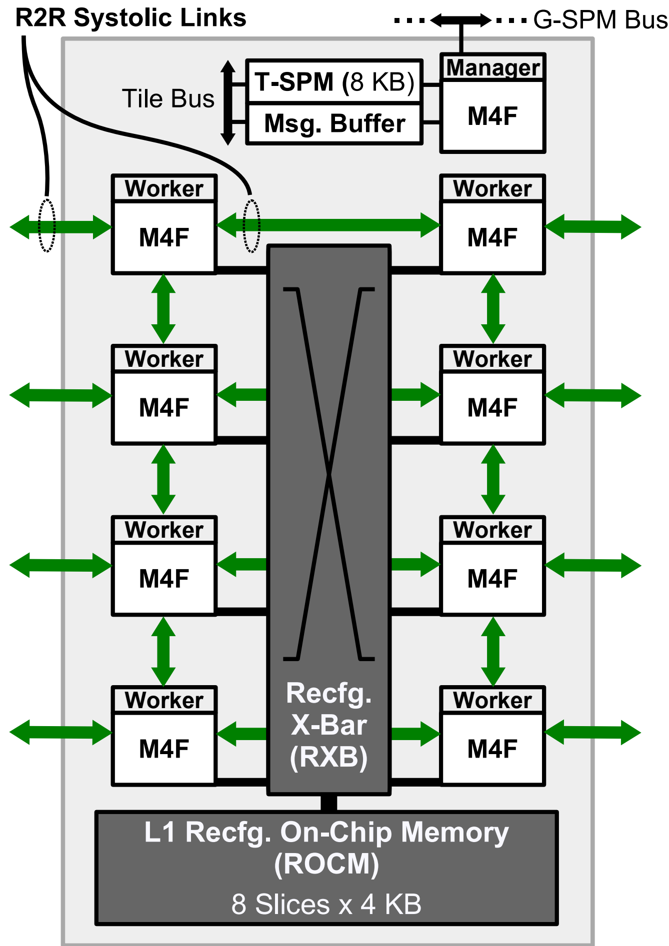


Figure 3.2: Composition of a Versa compute tile.

ARMv7-M ISA with DSP-oriented floating-point instructions, and adds 32 additional operands (`s0-s31`) to the M4F register set. Bare-metal ARM binaries are loaded into a 16 KB instruction memory that resides in each M4F core. A tile also includes a Cortex-M4F 'manager' core to handle supervisory tasks, including reconfiguration.

Versa workers have reduced functional coupling to the manager, unlike prior processors that explicitly partition management and computation. For instance, "Synergistic Processing Elements" (SPEs) in the Cell processor [72] could not directly access memory, instead requiring manager-coordinated DMA transfer between memory and SPE scratchpads. In contrast, Versa workers can access system memory without manager intervention, thereby reducing software complexity.

Each tile contains reconfigurable resources to support multiple hardware modes - namely the reconfigurable on-chip memory (ROCM), reconfigurable crossbar (RXB), and register-to-register (R2R) links. The ROCM and RXB reconfigure in a pair, and compose to provide a multi-modal L1 memory with optimized characteristics. 32 KB of L1 memory is partitioned into 8 slices (4 KB each), where the functionality of each slice is determined by RXB and ROCM sub-modes. Although it is possible to add reconfigurability in other levels of the memory hierarchy, early simulation experiments suggest diminishing returns. This is largely due to the intensity of data-transfer at the L1 level, and the Cortex-M4F's relative sensitivity to load/store latency [1]. In addition to the ROCM and RXB, R2R links (illustrated by green arrows) augment the M4F processors to enable systolic computations in arbitrary 2D-spatial groupings. Further detail on the ROCM, RXB, and R2R-related enhancements are provided in Section 3.3.

Large data structures that require cross-mode persistence are placed in a dedicated tile-level scratchpad memory (T-SPM). Notably, the T-SPM and a global-level scratchpad (G-SPM) external to the tiles facilitate accelerated thread-synchronization operations, which is discussed further in Section 3.3.4. A small point-to-point message buffer between the manager and each worker facilitates low-latency distribution of small runtime variables (e.g., parameters for load balancing).

3.2.2 Memory Hierarchy and Tile-External Support

The four compute tiles in the Versa prototype are supported by two additional levels of cache (Figure 3.3). Overall, the L1-L3 have capacities that form an 'hourglass' or inverted shape, typical for processors with high core counts. In addition, cache coherence and invalidation is explicitly software-managed; this mechanism significantly reduces hardware overhead, and is also the mechanism used in current GPU products.

[1] The M4F cores are augmented with a simple form of software-controlled prefetching to facilitate memory latency-hiding. This prefetch mechanism leverages 'store inversion', where cores may prefetch a cache line by storing its address to a special memory section. Effectively leveraging this mechanism involves compiler integration, which we leave for future work.

In terms of cache policies, Versa utilizes both read-allocate write-through, and read-allocate write-back caches. This is due to relative advantages between cache policies that depend on the level of temporal locality. For instance, a write-through policy prevents the problem of false sharing [73], while write-back typically incurs less spurious traffic under high temporal locality.

The L2 is a 16 KB fixed-function shared cache (implemented with four 4 KB slices) that utilizes read-allocate write-through. Slices are statically cacheline-interleaved [2]. In contrast to a small L2 write-back cache that would retain 'victim' lines evicted from the L1 [74], the write-through L2 effectively serves as a staging area for data shared across tiles. For instance, while cache lines never experience write-back eviction to the L2, updates to cross-tile shared data still update valid cachelines in the L2, obviating the need for L3 access.

The last level of the cache hierarchy contains a 1 KB 'L2.5' cache and 512 KB L3 cache. As discussed above, write-through stores necessarily propagate through the hierarchy. However, it is desirable to minimize the number of SRAM accesses in the L3. Thus, in contrast to the L2, the L2.5 is a fully-associative write-back cache that supports sub-block valid tracking [73]. Due to the small size of the L2.5, only 256 bits are required for word-granularity sub-block valid state.

The last notable component outside the tiles is an 8 KB global scratchpad memory (G-SPM) that is accessible by managers. Similar to the T-SPM, the G-SPM facilitates low-latency transfer of data related to control and supervisory tasks, in particular for the thread-synchronization optimizations discussed in Section 3.3.4.

[2] The interleaving function is $s = \texttt{index} \bmod_4$, where s is the slice number and `index` is the cacheline index bit-selected from an incoming address.

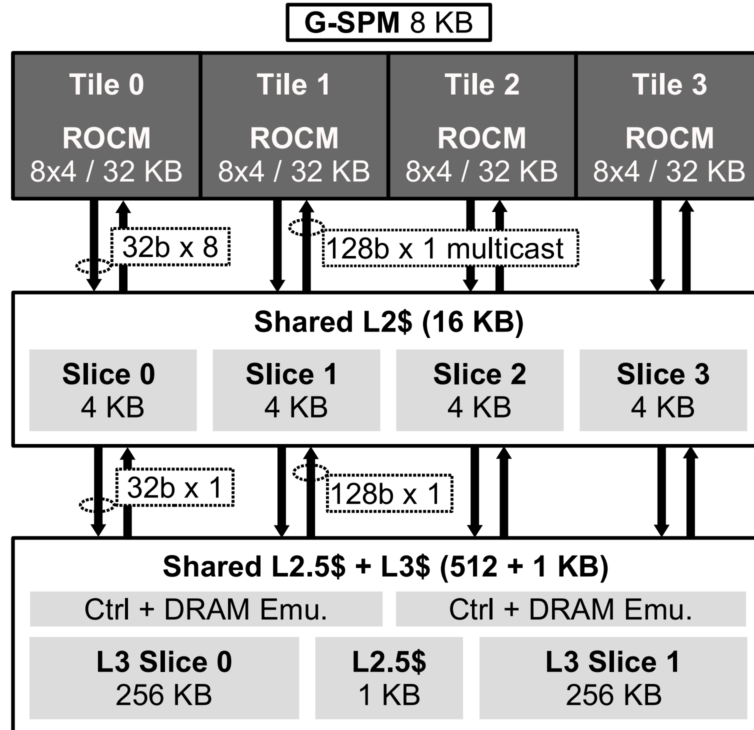


Figure 3.3: Overview of the Versa prototype and memory hierarchy.

3.3 Algorithm-Driven Reconfigurability

Versa supports 5 composite modes (Table 3.1) that can be dynamically configured at runtime, in addition to systolic R2R. The ROCM implements functionality that governs the memory type, namely for ROCM-cache, ROCM-scratchpad, and ROCM-queue. The RXB implements functionality for the memory scope, namely RXB-private, RXB-shared, and RXB-queue - a variation of the private scope that allows a pair of cores to access the same slice simultaneously. The microarchitecture of the RXB and ROCM and functionality of their sub-modes is described below.

3.3.1 Reconfigurable Crossbar

The reconfigurable crossbar (RXB) (Figure 3.4, top-left) contains 8 bidirectional ports, equal to the number of worker core and ROCM slice pairs. Each port is implemented with a pair of arbiters, one per upstream and downstream direction. Arbiters operate on

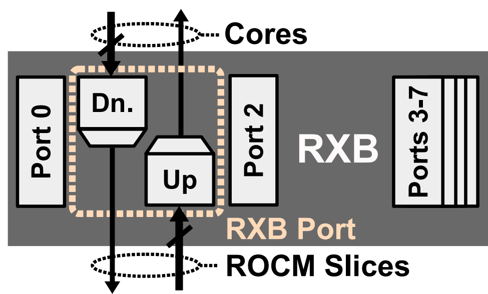
Table 3.1: Composition of logical memory modes. R2R functionality is orthogonal and compatible with all.

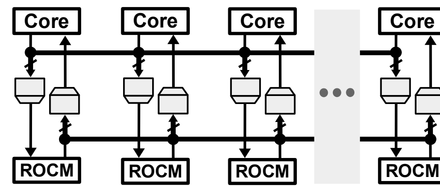
		ROCM Sub-mode		
		Cache	**Scratchpad**	**Queue**
RXB Sub-mode	**Private**	✓	✓	-
	Shared	✓	✓	-
	Queue	-	-	✓

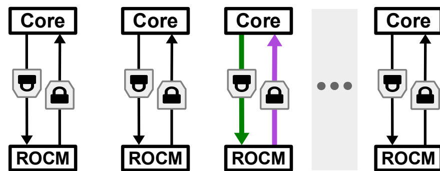
a per-direction basis - rather than per-signal basis - to amortize crossbar overhead (e.g., muxes and arbitration logic). The pair of arbiters is reconfigurable to one of 3 RXB sub-modes (Figure 3.4): 1) *RXB-shared*, 2) *RXB-private*, or 3) *RXB-queue* (access from a pair of cores). These sub-modes function as follows:

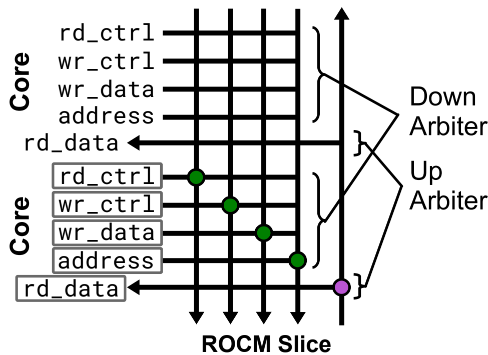
- *RXB-shared*: The crossbar provides all-to-all 'winner-take-all' connectivity between workers and ROCM slices, with least-recently-granted (LRG) arbitration [43].
- *RXB-private*: The arbiter crosspoints are statically fixed in both directions with 'winner-take-all', locking connections vertically between worker and ROCM pairs.
- *RXB-queue*: Upstream arbiters use 'winner-take-all' as in RXB-private, while downstream arbiters are statically 'split' between a pair of cores.

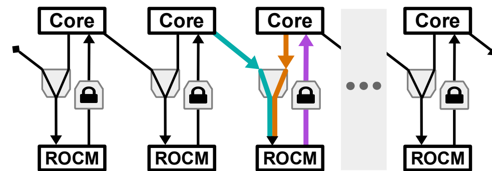
In RXB-shared, the ROCM slices appear as a single shared memory that is accessible by all workers in the tile. This is often beneficial for workloads with data structures that are accessed (and reused) across multiple cores. In addition, the 8× increase in capacity provided by RXB-shared reduces spills and re-fetches from subsequent memory levels for larger data footprints. In RXB-private, the locking of crosspoints between worker and ROCM slices not only eliminates bank contention, but also obviates arbitration. This results in up to 10.6× improvement in bandwidth and latency relative to shared mode (i.e., contention elimination + arbitration skipping), with a lower bound of 1.33× improvement due to arbitration skipping alone. RXB-queue supports common streaming DSP and filtering











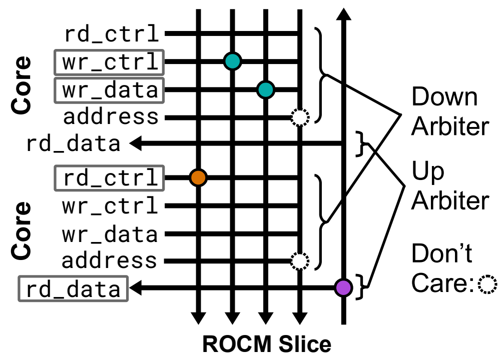


Figure 3.4: The reconfigurable crossbar (RXB) and overview of sub-modes. Diagrams in Private and Queue illustrate a sub-section of the crosspoint matrix.

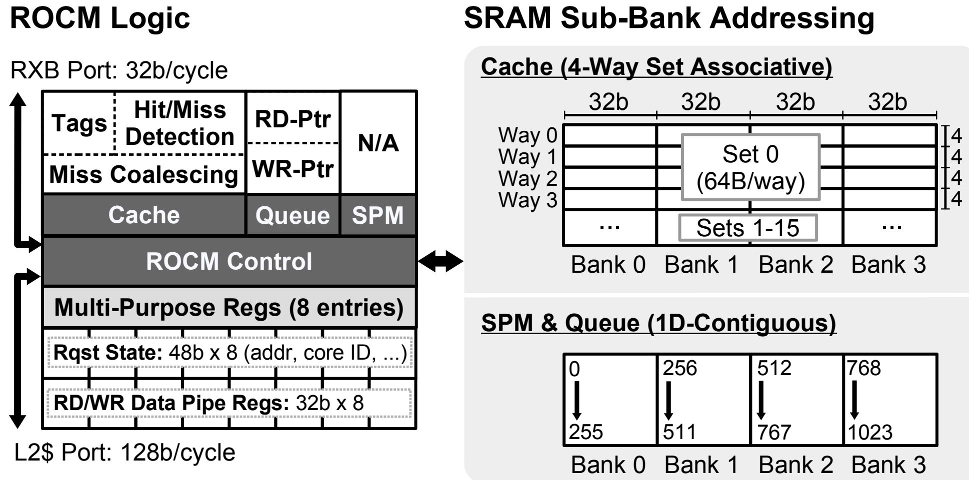


Figure 3.5: Sub-mode logic and SRAM components in the reconfigurable on-chip memory.

kernels. In contrast to the 'winner-take-all' pattern used in RXB-private and RXB-shared, RXB-queue resolves structural contention by sub-partitioning ownership of signals inside an RXB port. This 'port-splitting' enables simultaneous reader-writer access to ROCM slices, and effectively doubles bandwidth over the same port. Notably, FIFO semantics in queue mode enables full reuse of existing crossbar signals, without signal duplication or widening of buses.

3.3.2 Reconfigurable On-Chip Memory

Similar to the RXB, the L1-ROCM (Figure 3.5) contains logic to support multiple sub-modes as follows:

- *ROCM-cache*: a 4-way set-associative read-allocate write-through cache. Coherence is software-managed.
- *ROCM-scratchpad*: an explicitly-managed scratchpad memory. Main-memory accesses bypass the L1 and are forwarded directly to the L2.
- *ROCM-queue*: an explicitly-managed FIFO queue between core pairs. Main-memory accesses are forwarded to the L2 (as with ROCM-scratchpad).

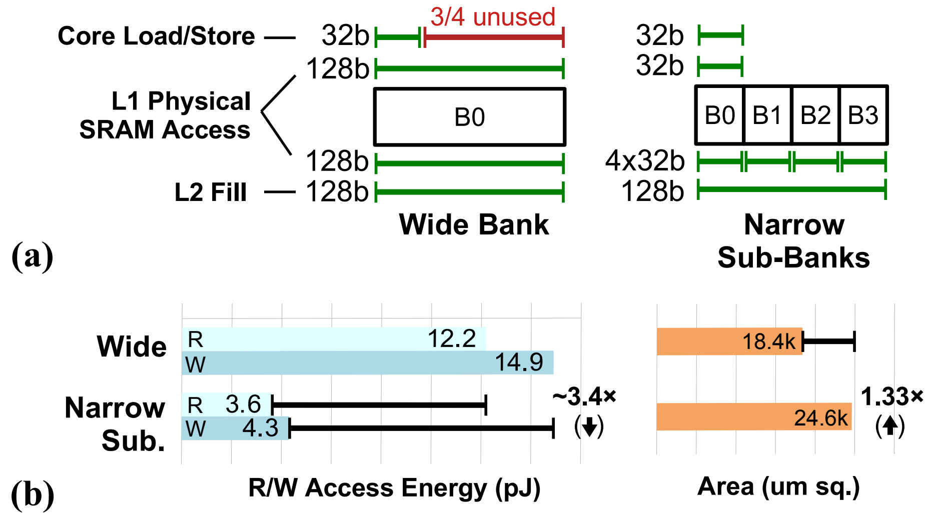


Figure 3.6: Comparison of banking schemes (wide v.s. sub-banked): **(a)** data utilization per access, and **(b)** design costs.

The cache sub-mode logic is primarily composed of cache tags, support for non-blocking cache requests (i.e., multiple outstanding requests), hit/miss detection, and coalescing logic. Coalescing refers to the case when an existing (in-flight) request with a cacheline match is merged with an incoming request, such that dispatch of a new L2 request is unnecessary (saving both bandwidth and request slots).

Logic to support ROCM-scratchpad and ROCM-queue is minimal. Aside from interface logic to forward requests that have addresses pointing to main-memory, a ROCM slice configured as scratchpad is functionally equivalent to a simple SRAM. With a 4 KB physical capacity (1024 32-bit words), additional state for FIFO operation in ROCM-queue consists of two 10-bit read/write pointers.

SRAM is reused across modes, with interface overhead limited to combinational logic that maps logical addresses to physical sub-banks (Figure 3.5-right). The choice of 32-bit sub-banks is driven by the native load/store width of the cores (Figure 3.6). While a single 128-bit bank amortizes SRAM periphery more effectively, 75% of the physical SRAM read-width would be unutilized for 32-bit scalar accesses (Figure 3.6a). Thus, sub-banking reduces common-case access energy by 3.4× in exchange for 33% more area (Figure 3.6b).

```
vldr   s4, [*]          Memory loads
vldr   s5, [*]
v[op]  s6, s5, s4       Useful work (mul, fma, ...)
vstr   [*], s6          Memory store
```

Figure 3.7: Conventional instruction sequence: obligatory memory access overheads are interleaved with computation.

3.3.3 Systolic Execution

Systolic arrays have been recently applied in ASIC designs [31, 75–77] but lack a general-purpose, programmable counterpart. While systolic arrays leverage registers for data movement, inter-core data transfer in load-store architectures necessitates cache or main-memory access. This reliance on explicit load/store instructions for data transfer has two impacts:

1. Performance costs (Figure 3.7): explicit load/store instructions consume pipeline cycles, stalling pipeline ALUs even if they are physically available for use
2. Energy costs: load/store instructions entail underlying operations that are significantly more dissipative than register access, e.g., SRAM access and cache traversal

Versa performs systolic array computation with *register-to-register (R2R) tunneling*. R2R directly connects adjacent cores to enable programmable systolic computation and enhanced spatial data reuse. Compared to existing multi-threaded programming models, R2R confers the following benefits:

1. Implicit data movement that increases utilization of pipeline ALUs, up to the physical limit
2. Inter-core data transfer that is register-to-register only, eliminating the energy cost of cache and SRAM access

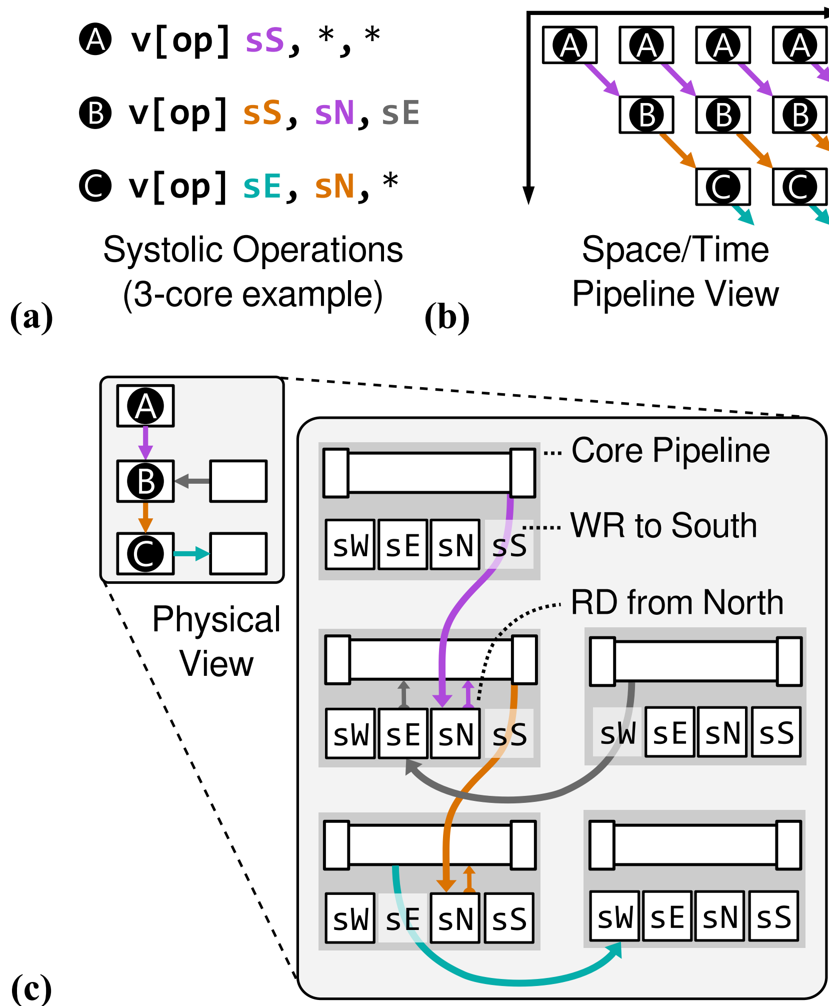
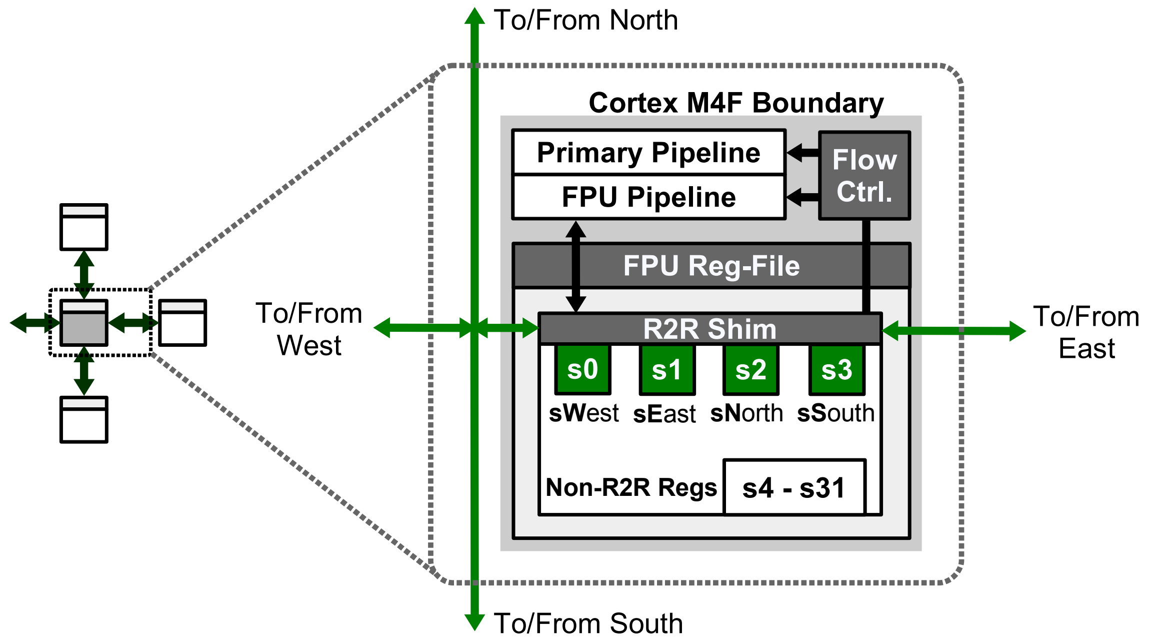


Figure 3.8: Example computation with R2R: **(a)** instruction sequence, **(b)** logical (pipeline) view, and **(c)** physical view of read/write conventions. R2R-writes bypass the local RF, while R2R-reads utilize local registers. Cardinal directions are inverted for writes to maintain spatial symmetry.



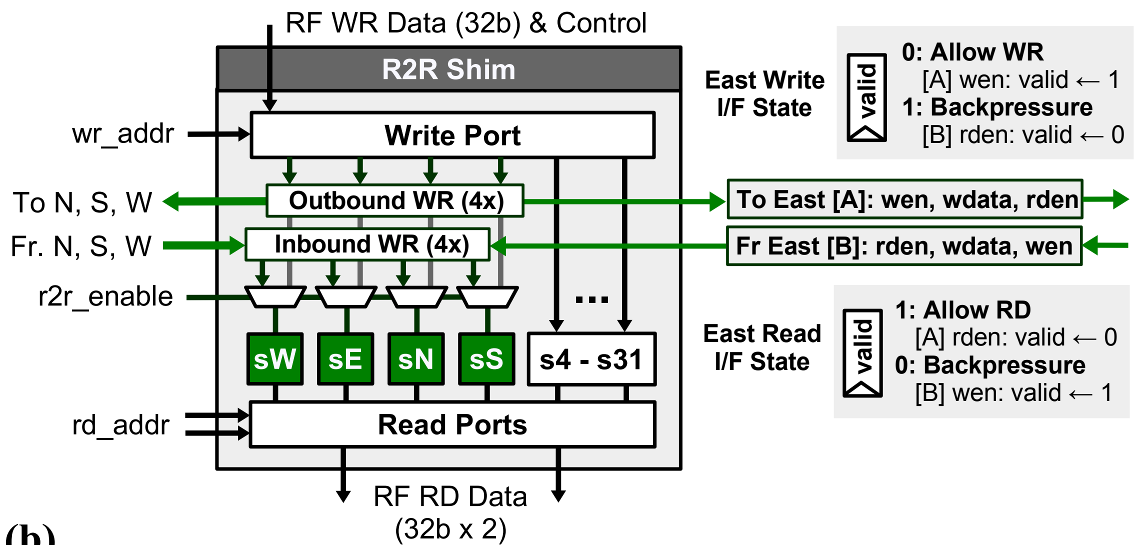


Figure 3.9: R2R microarchitectural implementation: **(a)** integration with the ARM Cortex-M4F pipeline, **(b)** interception of operands in the R2R Shim and 2-bit link-state control.

Figure 3.8a shows an example R2R instruction sequence across three cores. In place of explicit loads and stores, data movement with R2R is *implicitly controlled* by the inclusion of spatial registers as source or destination operands. This enables fine-grained, multi-step systolic computations (Figure 3.8b) with energy and performance benefits similar to an ASIC equivalent. Figure 3.8c illustrates the physical view of data movement between spatial registers. We note that the semantics of R2R requires a convention for the physical location of a shared spatial register; in Versa, the write destination always resides in the remote core. E.g., “Write to South” and “Read from North” both physically refer to the reader’s ‘North’ register.

R2R integrates seamlessly in the Cortex-M4F pipeline with minimal overhead. If enabled at runtime (i.e., in software), the FPU registers `s0-s3` are aliased to scalar data links in the `<W, E, N, S>` directions, respectively (Figure 3.9a). Link state (i.e., data valid tracking) requires 2 bits per bidirectional link. When an instruction writes to an R2R register, data from register write-back - normally directed to the local register file (RF) - is instead intercepted by the ‘R2R Shim’ (Figure 3.9b) and forwarded to an adjacent core. An R2R-write updates the link state and allows a matching R2R-read to proceed at the neighbor. In contrast, R2R-reads proceed if the link state is valid but utilize the local RF. Symmetry in read/write logic across cores minimizes timing impact and prevents the creation of new critical paths. Finally, flow control that prevents stale reads and destructive writes is implemented by tie-in with existing pipeline stall mechanisms. While this work augments the FPU register file to demonstrate the R2R concept with floating point workloads, the integer pipeline may be similarly extended (if desired) with no loss of generality.

3.3.4 Hierarchical Thread Primitives

Thread-synchronization barriers are fundamental operations that are notorious performance bottlenecks in parallel programs [78, 79]. For instance, thread barriers may consume up to 60% of execution time in complex parallel workloads [80]. Because barriers

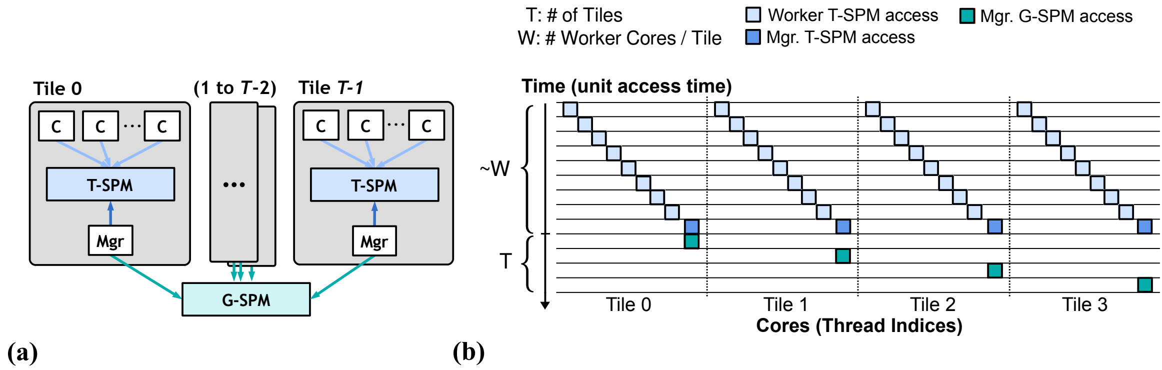


Figure 3.10: Accelerated thread-synchronization barriers: **(a)** Distributed scratchpads decentralize atomic updates to barrier data. Cores first synchronize within tiles over the T-SPM, followed by a global synchronization phase involving only managers over the G-SPM. After managers pass the global synchronization phase, each manager releases the barrier (and the workers) within their respective tiles. **(b)** Serialization is reduced by partitioning synchronization into parallel sections. In contrast to a centralized barrier with $O(N)$ scaling with respect to core count, the distributed hardware-software barrier implementation scales with $O(W+T)$, where W and T are the number of workers per tile and number of tiles, respectively.

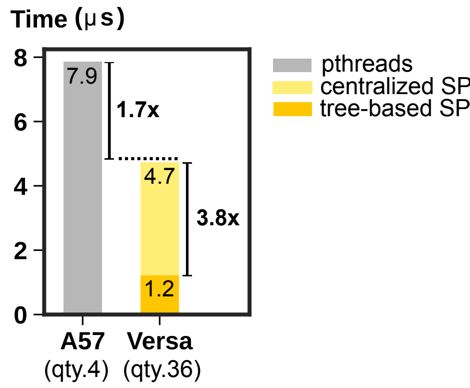


Figure 3.11: Comparison of thread synchronization latency with alternative barrier implementations. `pthreads` represents the off-the-shelf thread barrier implementation from the popular `pthreads` library, measured on a quadcore Arm A57 with 4 spawned threads. `centralized SP` is a barrier implemented on Versa using single centralized scratchpad memory. `tree-based SP` is the Versa thread barrier implementation that leverages distributed scratchpads.

constitute a non-trivial fraction of parallelization overhead, application developers go to significant lengths to both minimize their usage and develop low-level optimizations. Prior efforts include entirely new algorithm implementations [81, 82], in addition to a significant body of work on underlying synchronization-related primitives [83]. Nevertheless, thread barriers can be minimized but typically not eliminated entirely.

Thread synchronization entails serialized, atomic updates to shared variables that track whether threads can safely enter or exit a region of parallel execution. In conventional CPUs each atomic update can require 100's to 1000's of cycles (i.e., microseconds), largely due to deep coherent caches that centralize barrier data. In addition, barrier operations have poor scalability in manycore designs, and in the worst case exhibit $O(N)$ scaling with respect to core count.

Versa abolishes cache altogether and instead utilizes scratchpads distributed at the tile (T-SPM) and global (G-SPM) levels for lock and barrier operations. In addition to controlled latency and full support for Cortex-M exclusive access extensions, the T-SPM and G-SPM enable decentralized updates in a tree-based strategy (Figure 3.10a). The hierarchical (tree-based) strategy partitions atomic updates across tiles into sections that run in parallel (Figure 3.10b), improving scalability from $O(WT)$ to $O(W+T)$, where W and T are the number of workers per tile and number of tiles, respectively. Although we focus on thread barriers for this work, the flexibility of the T/G-SPMs facilitate the full range of thread synchronization primitives built on atomic memory access (*e.g.*, semaphores, locks, and general multi-threaded data structures).

The tree-based scratchpad barrier is evaluated (Figure 3.11c) against two baselines: a centralized scratchpad barrier that is measured with Versa in RTL simulation, and an off-the-shelf barrier implementation from the popular `pthreads` library, measured on a quadcore ARM A57 CPU. The centralized scratchpad-based approach alone achieves a 1.7× speedup compared to the cache-based barrier from `pthreads` on the CPU. Adding the tree-based strategy yields an additional 3.8× speedup - or 6.5× total - despite a 9× higher

threadcount. Thus, Versa accelerates barrier operations such that thread synchronization contributes marginally to parallelization overheads; this is critical for design scalability, and to prevent performance bottlenecks at higher core counts.

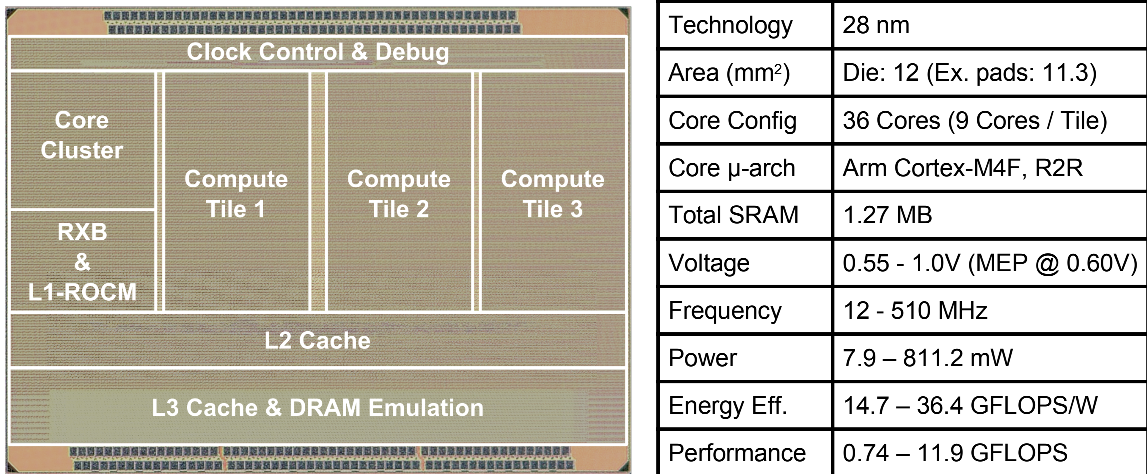
3.3.4.1 Reconfiguration Control

Typical reconfigurable systems (e.g., FPGAs) have combinatorially-large configurations that necessitate large bitstreams and significant on-chip storage. In contrast, the finite set of Versa modes are governed by a single memory-mapped register (MMR) in each tile, accessible by the manager core. 4 bits control RXB and ROCM sub-modes (2 bits each). Thus, memory mode transitions involve a simple MMR write and complete in 2 cycles. Versa software libraries currently support (optional) memory reconfiguration by managers during thread synchronization, which guarantees that reconfigurable resources are not accessed during the 2-cycle reconfiguration window. In contrast, the enable/disable for systolic R2R is tied to 1 bit in an MMR local to each worker core, and R2R registers are guarded from automatic compiler usage in regions of code where R2R is enabled (i.e., correctness for R2R is compiler-enforced).

3.4 Chip Implementation

Versa is fabricated in a 28 nm CMOS process and occupies 12 mm^2 die area (Figure 3.12). The design is implemented hierarchically with tiles and the L3 as hard partitions. The L3 also includes logic for DRAM timing emulation, which is disabled for this work and left for future evaluation with Versa's prefetching features. A single unstructured clock tree is sufficient to meet the 2 ns T_{clk} target, with 520 ps mean insertion delay and 70 ps mean global skew (3.5% of T_{clk}). At the nominal voltage (1.0V) the system operates at 510 MHz, corresponding to 811.2 mW and 11.9 GFLOPS power and performance, respectively.

The test chip incorporates parallel boundary scan interfaces for test and debug, in addition to a VITA 57.1 FMC interface. Automated infrastructure for data collection is de-



Technology	28 nm
Area (mm²)	Die: 12 (Ex. pads: 11.3)
Core Config	36 Cores (9 Cores / Tile)
Core μ-arch	Arm Cortex-M4F, R2R
Total SRAM	1.27 MB
Voltage	0.55 - 1.0V (MEP @ 0.60V)
Frequency	12 - 510 MHz
Power	7.9 – 811.2 mW
Energy Eff.	14.7 – 36.4 GFLOPS/W
Performance	0.74 – 11.9 GFLOPS

Figure 3.12: **(L)** Die photo and **(R)** summary characteristics.

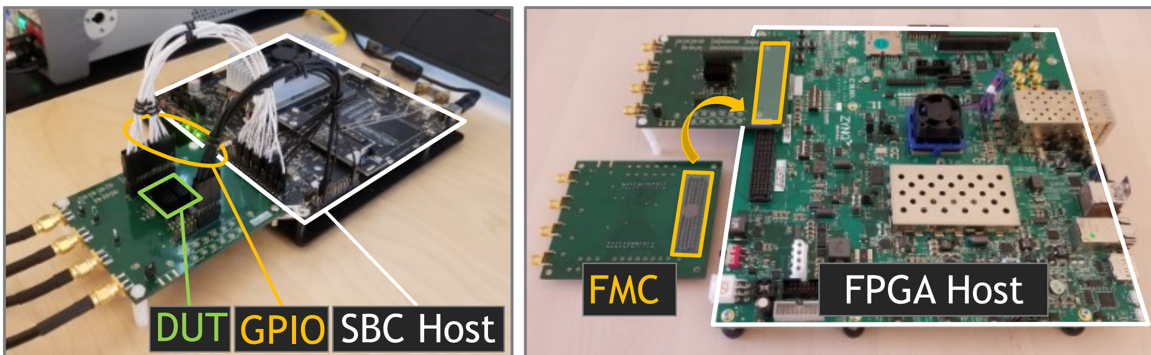


Figure 3.13: Setups for chip testing: **(L)** Linux single-board-computer debug host, primarily for bringup and data collection, and **(R)** FPGA interface for emulated MPSoC integration.

Table 3.2: Hardware baselines.

	Cores	Clock	Architecture	Feature Note
ARM A57	4	1.8 GHz	ARMv8-A	2-way superscalar, OoO
NVIDIA TX1	256	1.0 GHz	Maxwell Gen.2	Eight 32-lane SIMD units

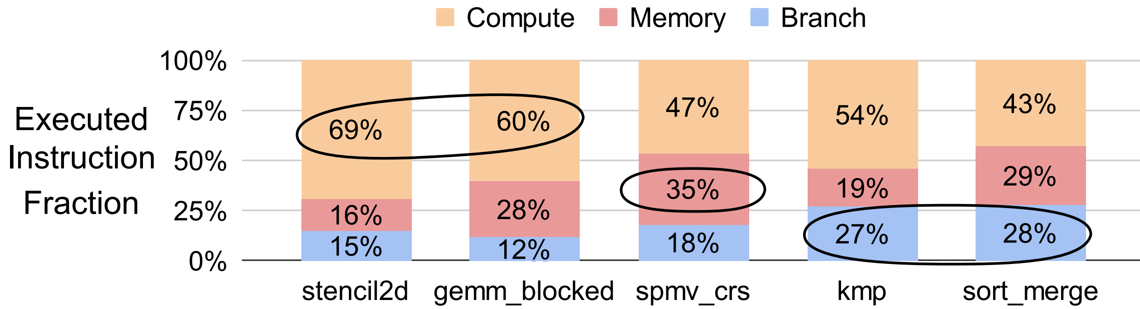


Figure 3.14: Evaluation kernels from MachSuite: kernels are selected to capture a range of characteristics.

veloped in python and C++ and runs on a Linux single-board computer (Figure 3.13-left). The FMC interface (Figure 3.13-right) enables emulated integration with an FPGA MPSoC device, in particular with hard application-class host cores.

3.5 Baselines and Test Methodology

Versa is evaluated against two energy-efficient hardware baselines. The baselines - a quad-core ARM A57 CPU and 256-core Maxwell Gen.2 GPU (both integrated in NVIDIA's Tegra X1 SoC) - are mobile-class processors fabricated in a comparable 20 nm CMOS process. All measurements are performed with uncapped power states, and averaged across the latter half of 1000 iterations with the first half discarded for cache warmup. GPU measurements utilize custom timers implemented in PTX assembly to obtain nanosecond-resolution timings free of host-side overheads. CPU measurements use nanosecond-resolution timers from `std::chrono`.

Power dissipation for the Versa chip is measured from a benchtop power-supply, while

the CPU and GPU measurements leverage on-board I2C-addressable current monitors on the Jetson TX1 platform. All major portions of the test chip (core logic, test and debug circuitry, SRAM) are included in measurements, and recorded as a lump number from a single supply. Digital I/O and a tunable clock generator reside on separate voltage domains and are excluded, but contribute marginally to power dissipation. The current monitors on the TX1 enable independent measurement of CPU and GPU power, but to the best of our knowledge do not compensate for DC-DC converter losses or DRAM power [3].

Five kernels from MachSuite [84] are selected based on the mix of instruction types, capturing representative diversity (Figure 3.14). Stencil2D (2D convolution) and GeMM (matrix mult.) exhibit regular data accesses to dense data, while KMP (string search) and SpMV (sparse matrix-vector mult.) have data-dependent variation in access patterns. Mergesort is a branch and synchronization-heavy comparison-based sort. We select 2 Versa modes per kernel based on analysis of the MachSuite reference kernels, and modulate data sizes up to 512 KB. The CPU is evaluated with the reference kernels, while the GPU kernels use hand-optimized CUDA. CUDA kernels were developed with an effort-level of 2-4 weeks per kernel, guided by `nvprof` profiling and analysis with hardware performance counters. Best-effort optimizations were utilized wherever possible, including memory coalescing, data tiling, scratchpad ('CUDA shared mem.') usage, divergence optimization, and sweeps of thread-block and grid sizes. Kernels for all platforms use FP32 as the primary workload datatype, which is predominant in ML research, sparse HPC, graph analytics, and genomics. We note that an evaluation with integer datatypes is largely redundant, since benefits from memory reconfiguration are orthogonal, and performance of the Cortex-M4F FPU pipeline is a lower bound relative to integer performance [85].

[3] The TX1 uses two Samsung 16Gbit LPDDR4 chips (part no. K4F6E3S4HM) soldered on-board. According to manufacturer datasheets the two DRAM chips dissipate between 150-400 mW. This would translate to less than 10% of either CPU or GPU power.

Table 3.3: Efficiency and performance improvement from dynamic reconfiguration between modes. Stars indicate whether the mode-advantage is consistent (★) or mixed (☆).

Kernel		Mode A		Mode B	Relative Gain
STENCIL2D	☆	P. Cache	☆	P. SPM+R2R	1.26×
KMP		P. SPM	★	P. Cache	2.62×
GEMM	☆	S. Cache	☆	S. SPM	1.73×
MERGESORT		S. Cache	★	S. SPM	1.22×
SPMV		P. Cache	★	S. Cache	1.57×
All Kernels					**1.53**×

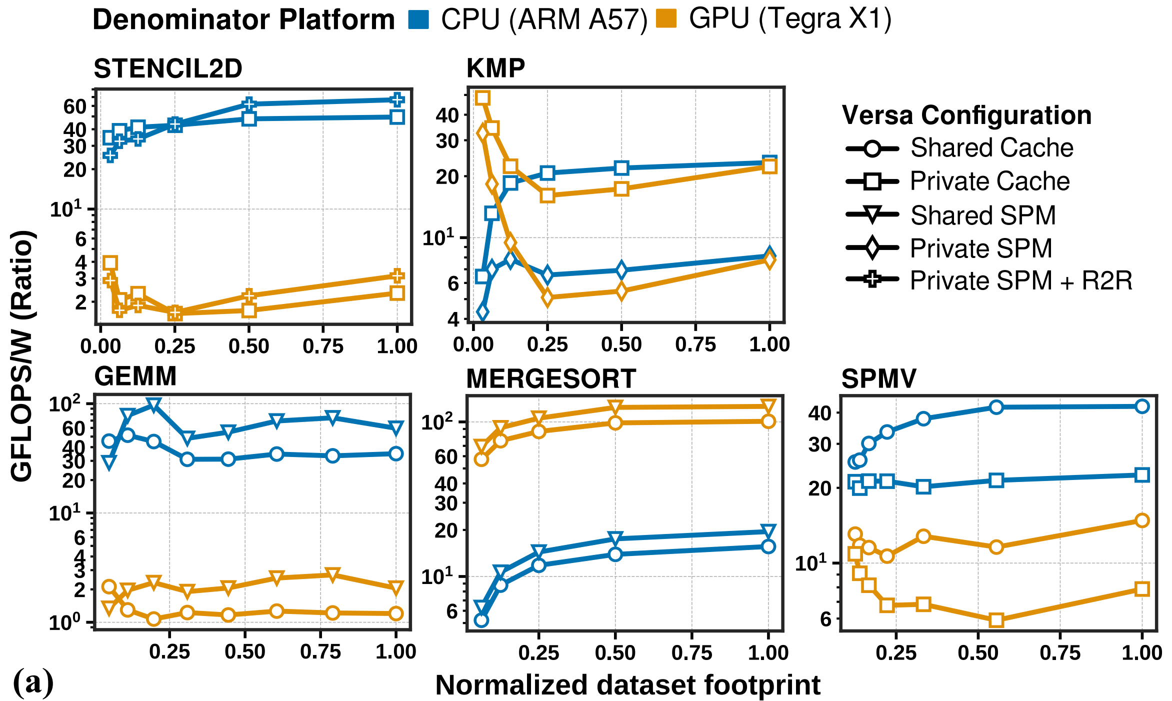
Table 3.4: Median improvements across all kernels.

	GFLOPS/W		GFLOPS	
Kernel	**CPU**	**GPU**	**CPU**	**GPU**
STENCIL2D	42.2×	2.25×	6.92×	0.16×
KMP	19.6×	22.3×	3.18×	1.51×
GEMM	64.5×	2.18×	10.5×	0.15×
MERGESORT	14.4×	105×	2.33×	71.6×
SPMV	33.5×	11.8×	5.42×	1.80×
All Kernels	**37.2**×	**11.6**×	**5.86**×	**0.78**×

3.6 Measurements and Analysis

The following section presents measured results for MachSuite test kernels at nominal voltage, followed by voltage-scaling measurements.

We note that because hardware reconfiguration time is marginal, measured results for kernel sequences (with reconfiguration interleaved) is virtually identical to the sum of runtimes for kernels executed in isolation. This property also holds for multi-modal kernels that are decomposed into distinct phases.



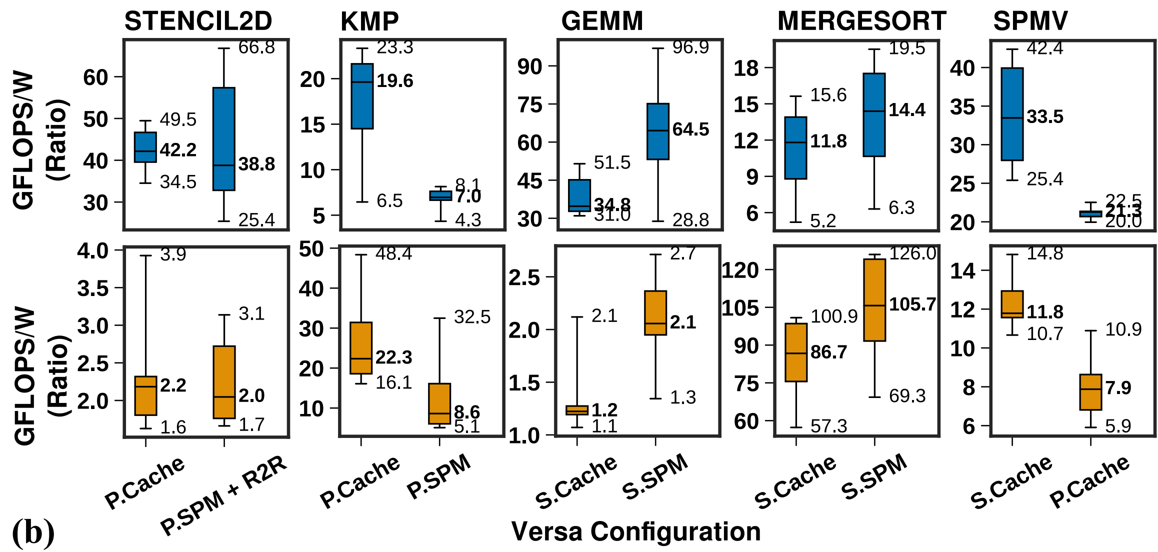


Figure 3.15: Energy-efficiency improvement over CPU and GPU baselines: **(a)** trends across dataset footprints, and **(b)** summary boxplots. Boxplots are labeled on min, median, and max values.

3.6.1 Nominal Energy-Efficiency and Performance

The benefit of Versa reconfiguration is illustrated by trends in energy-efficiency across dataset footprints[4] (Figure 3.15a) and median overall improvements (Figure 3.15b). We find that the best mode varies not only across the testsuite, but also within individual kernels. Energy-efficiency improvements between Versa modes extend up to 3.17×, with 1.53× disparity on average (Table 3.3). This is a key result that indicates the lack of *any mode* that could serve as a static replacement for reconfigurability. For instance, Stencil2D with Versa private cache (P.Cache) yields 1.37× higher GFLOPS/W relative to private SPM+R2R at small data sizes, but the advantage between modes is inverted at larger sizes. This result is due to the use of R2R to share and reuse overlapped input patches across cores, and cache pressure as dataset footprint increases. Overall, Versa achieves 42.2×/2.2× median improvement over the CPU/GPU for Stencil2D.

For KMP, P.Cache consistently outperforms private SPM (P.SPM) despite $Q-1$ reuses of the query per search iteration, where Q is the query length. Since scratchpads require explicit buffering, the number of loads/stores per iteration is nearly doubled relative to cache mode. Given a short query string (tests use $Q=4$), reuse is low and SPM buffering dominates. This results in 2.62× disparity between modes, with 19.6×/22.3× improvement over the CPU/GPU with P.SPM.

GeMM utilizes data tiling in shared cache (S.Cache) and shared scratchpad (S.SPM) modes, and has a reuse/buffering tradeoff similar to KMP. However, the benefit of quadratic data reuses outweighs scratchpad buffering cost, resulting in 1.73× median mode-advantage for S.SPM and 64.5×/2.1× improvement over the CPU/GPU.

On Mergesort, Versa attains 2.33× and 71.6× speedups over the CPU and GPU, respectively, translating to 14.4× and 105× energy-efficiency improvements. GPU profiling indicates bottlenecks in parallel synchronization and branch-heavy comparison operations. Results from Mergesort suggest that Versa's independent scalar cores and tree-based

[4] Dataset footprint is the size in bytes for a kernel's input and output arguments.

scratchpad barriers are highly effective in-practice.

For SpMV, the Versa kernels allocate 1 or more sparse dot product operations per worker core, implying that accesses between cores are non-overlapping. However, since MachSuite provides matrices in CSR format, sparse values are packed contiguously in memory such that multiple sparse rows frequently reside in the same cache lines. This produces implicit cache prefetching of sparse matrix values and column pointers into the shared cache. Synergistic prefetching does not occur in P.Cache, resulting in a mode-advantage for S.Cache up to 1.9×, and 33.5×/11.8× improvement against the CPU/GPU.

Across all kernels, energy-efficiency improvements extend up to 64.5×/105× against the CPU/GPU, with median improvements of 37.2× and 11.6× overall (Table 3.4-left).

In terms of performance, Versa consistently outperforms the CPU but results are mixed against the GPU (Table 3.4-right). 5.86× median improvement in performance is obtained relative to the CPU; we estimate that hardware parallelism accounts for roughly half of the disparity, with the other half attributed to reconfiguration benefits. GPU performance ratios range from 0.15× to 71.6× on GeMM and MergeSort, respectively. While GPUs are known to excel on linear algebra computations (which are prevalent in graphics applications), the difference in core counts is a plausible explanation for the performance shortfall. For instance, multiplying Versa's performance by a factor of 8 to normalize core count results in performance improvements of 1.28× and 1.2× for Stencil2D and GeMM, respectively. However, it is unlikely that Versa's power dissipation would scale linearly, in addition to production-related overheads and design margins that the Versa prototype lacks. Nevertheless, these observations are promising and suggest that Versa is comparable on dense kernels, and within the margin of error created by differences in implementation methodology.

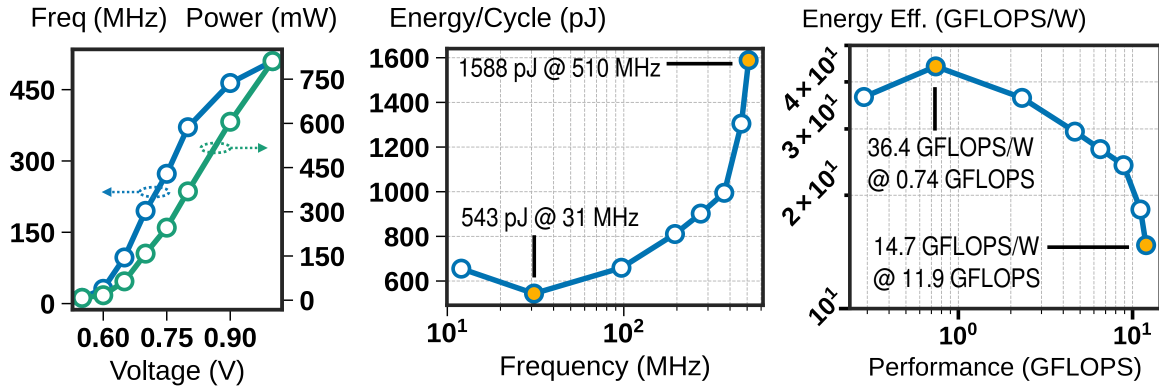


Figure 3.16: Chip characteristics under voltage scaling. **(L)** System frequency and power, **(M)** energy-per-cycle, and **(R)** the performance-efficiency Pareto curve.

3.6.2 Voltage-Scaled Characteristics

We conclude the measurement results with chip characteristics under voltage-scaling, examined on a non-terminating 11-tap FIR filter. This gives a reasonable measure of 'peak-practical' performance and efficiency, since the FIR kernel is compute-oriented but retains a realistic number of memory accesses. At the far ends of the 0.55-1.0V operating range, the chip dissipates 7.9 mW (at 31 MHz) and 811 mW (at 510 MHz) (Figure 3.16-left). Energy-per-cycle (Figure 3.16-mid) varies from 543-1588 pJ/cycle. The minimum-energy-point (MEP) is observed at 0.6V, resulting in 2.47× improvement over the nominal voltage. Thus, Pareto-optimal operation (Figure 3.16-right) corresponds to 11.9 GFLOPS performance at 1.0V, or 36.4 GFLOPS/W energy-efficiency at 0.6V.

3.7 Related Work

Prior designs most similar to Versa include the Raw processor [86], Manticore [87], and the ET-SoC-1 from Esperanto Technologies [88] (the latter two developed concurrently with this work). Raw incorporates packet-switched routers that are instruction-addressable for stream-based dataflow, similar to Versa's instruction-level systolic computation. However, Versa's systolic mechanism is comparatively lightweight (*e.g.*, without packet routers), does not require per-packet initialization or link setup, and is physically-

constructed to mimic an ASIC design. Manticore incorporates hardware for DMA operations into core registers. This enables removal of explicit load/store instructions similar to Versa, but still incurs DMA setup overhead, and the mechanism does not facilitate cross-thread spatial data reuse. ET-SoC-1 is a flexible accelerator targeted for machine learning that incorporates lightweight general-purpose cores. To the best of our knowledge, ET-SoC-1 is the most recent industry accelerator design to incorporate memory that is dynamically reconfigurable as cache or scratchpad.[5] We speculate that the ET-SoC-1's reconfigurable memory confers benefits similar to the Versa design, however Esperanto has not disclosed details or performance metrics related to reconfigurability.

In addition to the works above, aspects of Versa can be compared to recent FPGA systems, reconfigurable ASICs, and general-purpose processors with programmable interconnect. We also summarize relevant work on application phase-based dynamic reconfiguration and optimization. The authors of [89] and [90] integrate embedded FPGAs and fixed-function accelerators in heterogeneous SoCs. In this setup, kernels are accelerated by offloading to different, distinct subsystems or cores. In contrast, Versa is a standalone accelerator architecture that reconfigures to support different algorithm needs. The ASICs from [45, 91, 92] leverage programmable interconnect or packet routers in order to support multiple applications. For instance, the systolic array in [91] uses reconfigurable routers to support both dense and sparse linear-algebra operations. The ASIC design from [45] reconfigures between phases of a single sparse matrix-multiplication algorithm, and [92] use programmable routers to support multiple image processing kernels. The efforts above employ reconfigurability to broaden the capabilities of a fixed-function ASIC design. In contrast, Versa is the first to demonstrate how reconfigurability - in a general-purpose processor - can enhance programmability *and* performance.

The designs presented in [86, 87, 93–95] are fabricated multi-core processors capable of spatial data-transfer that resembles that of a systolic array. Versa also leverages pro-

[5]Recent Nvidia GPUs have configurability along these lines, but it is rather limited; CUDA shared memory size can be configured via the host GPU driver API, but not at fine temporal grain.

grammable interconnect, but for the purpose of reconfiguration in-conjunction with ROCM memory modes. Furthermore, systolic dataflow in Versa is not emulated but instead employs direct R2R links, with performance and energy-efficiency characteristics closer to those of an ASIC design.

Versa can be compared to recent coarse-grained reconfigurable array (CGRA) designs with dataflow execution [12, 56–58, 61, 96]. Dataflow machines [96] depart from program counter (PC)-based execution and instead traverse dataflow graphs explicitly. Data values and associated data-tags flow through distributed on-chip memories and compute resources according to graph dependencies ('firing rules'). In comparison, our work retains PC-based execution and reconfigures at higher levels of abstraction, but follows dataflow-like firing rules for systolic R2R. In this sense, Versa is closer to a "hybrid dataflow machine," or "pseudo-systolic processor" [97] that incorporates von Neumann processing elements.

Phase-based optimization [98] for adaptive hardware is a well-established research area, with prior work that spans the hardware stack [15, 99–104]. [99] and [100] examine program features and learning-based approaches to optimize main-memory allocation in real server workloads. [101] optimizes last-level cache capacity in an off-the-shelf Xeon processor, using L1 and L2 cache performance counters to build a phase-based heuristic controller. [15, 102–104] are architectural (cycle-level simulation) studies that optimize on-chip memory types, cache sizes, and pipeline resources. For instance, [15] and [104] leverage explicit workload phases and varying data sparsity to reconfigure on-chip resources according to power-performance tradeoffs. Similarly, [102] and [103] adapt microarchitectural parameters, but use learning-based models to detect and exploit implicit workload phases. While we leave an in-depth study of phase-based optimizations on Versa for future work, our hardware is designed to be compatible with both explicit and implicit phase-based methods, as described above.

3.8 Conclusion

This chapter introduced Versa, a flexible multi-core accelerator that optimizes for diversity in computation and data-access patterns. The premise is that given dynamic compute and data-access characteristics, any static hardware design will exhibit sub-optimal energy-efficiency and performance. Versa addresses this issue through fast, nanosecond-scale reconfiguration between distinct hardware modes. Reconfigurable functional units exploit mode-dependent operating guarantees (such as privatized data) to optimize microarchitectural characteristics. In addition, Versa's scalar cores are augmented to support a new class of programmable, register-to-register systolic array computation. To minimize and prevent performance bottlenecks on synchronization-heavy workloads, Versa employs distributed scratchpads that facilitate a tree-based thread-synchronization algorithm. The techniques above are demonstrated in a 28 nm prototype chip that incorporates industry-grade IP cores and system components. Against comparable CPU and GPU baselines, the prototype achieves 37.2× and 11.6× median improvements in energy-efficiency respectively on a set of diverse kernels. Overall, this work indicates that compute and memory reconfiguration are highly-effective, and may be broadly applicable to future accelerator designs.

Chapter IV

Compiler Support for Versatile Hardware

4.1 Introduction

Specialized software-programmable accelerators were introduced in Chapter I as a promising avenue to obtain better compute performance and energy-efficiency. However, while the shape and quantity of accelerator hardware resources set theoretical limits on performance and efficiency, *accelerator software* dictates whether those resources can be utilized in practice. In particular, accelerator programs are typically partitioned into computationally-intensive subunits called *kernels*. Kernels are not only ubiquitous building blocks that constitute larger programs, but kernels also typically dominate end-to-end workload execution time. Thus, developing high-performance kernel code in a productive, scalable way is crucial to harness the potential of future accelerator designs.

Developing high-performance kernel code is notoriously challenging, and still an active area of research for several key reasons. In particular 1) the need for complex, expertise-driven optimizations over a large space of possible code variants, and 2) the software churn and effort-intensive development required to support a diverse, evolving hardware landscape (Figure 4.1). These reasons make accelerator software development (and kernel engineering) a non-trivial, effort-intensive affair. For instance, effective loop tiling must take the structure of multi-level memory hierarchies into account, and parallelization strategies will depend on the structure of compute units, available memory bandwidth, and latency

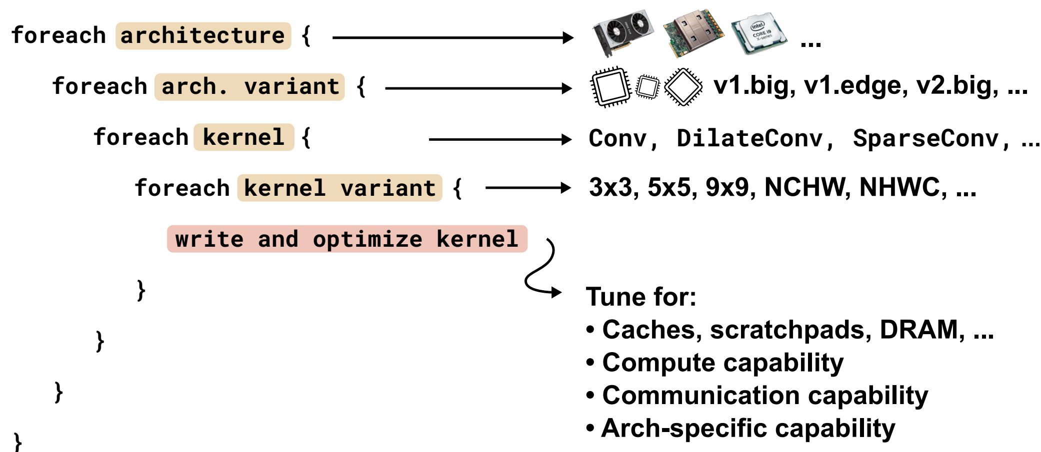


Figure 4.1: Development of accelerator kernels and associated libraries is a challenging task that prioritizes hardware performance. Because kernel development is already complex and effort-intensive, *e.g.* requiring extensive tuning and detailed hardware knowledge, the rapid evolution of both hardware and algorithms presents a non-trivial challenge.

hiding capabilities. Not only are impactful code optimizations typically invasive (*i.e.*, requiring major source modification), but tuning them is also hard. Exploring over the space of possible code optimizations is a combinatorially hard problem, and different programs often exhibit varying behavior under the same optimization. A single mathematical operation may be implemented as several independent kernels that are tuned for different input/output parameters, or to exploit different hardware functionality. Hardware that exposes a variety of architectural features exacerbates the combinatorial explosion of kernel and optimization variants, and breaking software abstractions impacts the portability of accelerated code. In summary, development of conventional kernel libraries is highly architecture-dependent, labor intensive, and intractable for programmable accelerators that will continuously evolve to support emerging domain-specific computations.

As a result of the challenges above, compute kernel developers must typically choose between tradeoffs in performance, portability, and productivity. For instance, hand-tuning and adhoc heuristics are often used in-place of comprehensive design-space-exploration, resulting in performance opportunities that are overlooked. On the other hand, the need to develop and optimize a large set of kernels for a given target architecture is a major

contributor to maintenance issues and large, esoteric kernel libraries [105]. Regardless of the tradeoffs that engineers select, the development methodology suffers due to the lack of appropriate tools and abstractions.

4.2 Motivation

Exploiting the proliferation of accelerator hardware in the post-Moore era is a difficult software problem. We argue that better compilation is the key to enable emerging programmable accelerators, and the hardware-software co-design that is required.

One critical barrier is the lack of compatible tools and compiler infrastructure. Because most existing compilers are designed primarily to support mainstream CPU architectures, existing internal representations (IRs), optimization methods, and compiler utilities are ill-suited to accelerators that employ a drastically different hardware organization.

The lack of first-class compiler support also holds for the Transmuter and Versa accelerators presented in Chapters II-III. In particular, while we programmed both Transmuter and Versa in C++ (supported by a GNU-based toolchain), the language and tooling limitations described in Section 4.1 reduced the scope of feasible application development. For instance Transmuter required adhoc code generator infrastructure to be effective (Section 2.6), and kernels using Versa reconfiguration and register-to-register features (Section 3.3) were challenging to develop by hand. For both Transmuter and Versa, weeks to months of development were required to obtain a relatively small set of hardware-optimized kernels; In our experience, this appears to be the status quo for most programmable accelerators (*e.g.*, GPUs) rather than the exception.

Polyhedral methods offer a promising framework to compile and optimize high-level accelerator programs. By representing program components as affine geometric objects – specifically polyhedra – the polyhedral framework enables a wide array of crucial analysis and transformations. Polyhedral tools are especially amenable to compute kernels and multi-dimensional loop nests, enabling composable transformations that are well-suited for

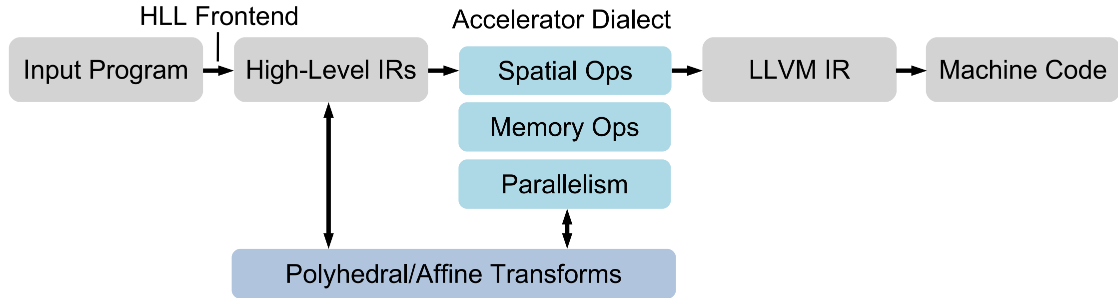


Figure 4.2: Overview of the proposed compiler and IR lowering pipeline.

tiled accelerator architectures. In contrast to conventional low-level IRs that lose critical dependency information, polyhedral methods enable exact memory dependency analysis; this is crucial for automatic parallelization, compute partitioning, and transformations involving program dataflow.

4.3 Polca: Polyhedral Compilation for Versatile Hardware

We propose the use of a progressively lowered compiler infrastructure augmented with polyhedral methods to automatically generate performant kernels, and exploit emerging accelerator hardware. Our approach (Figure 4.2) will initially target the Versa and Transmuter architectures proposed in Chapters II-III, and can be summarized as follows: (i) An end-to-end toolchain to compile high-level input programs into optimized, executable MIMD binaries. The toolchain leverages MLIR [106, 107] – a recent compiler infrastructure with strong, modular support for progressive lowering – to mitigate the loss of semantics that are required for invasive (but impactful) transformations. Segregating functionality in the form of different internal IRs maximizes reuse of upper-level compiler passes, improves modularity, and reduces the engineering overhead required to support new hardware.

(ii) Custom language constructs formalized as an accelerator IR dialect to support architecture-specific features and operations. In particular, constructs to support partitioned reconfigurable memory and reconfigurable buffer allocation, parallelization and hardware mapping, and systolic dataflow (i.e., inter-core register-to-register operations).

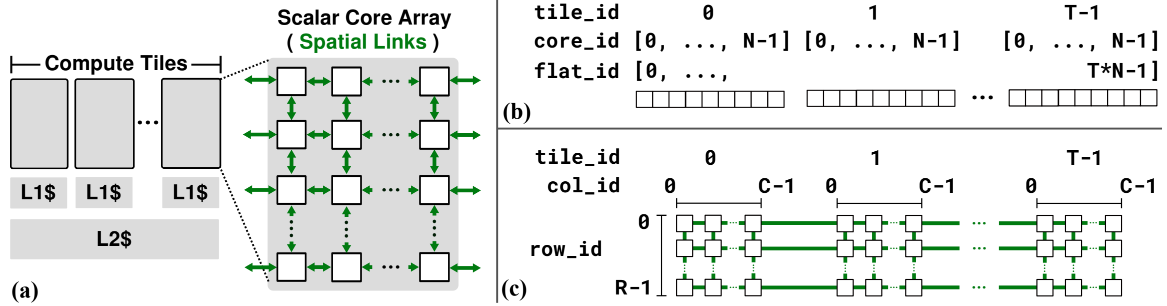


Figure 4.3: Overview of Polca target accelerator.

(iii) Polyhedral transformations and program analysis that exploit light-weight analytical cost models. Analytical cost models significantly reduce compilation time compared to methods that require performance feedback (i.e., direct execution on target hardware). The reduction in compilation time still applies if cost models are substituted for performance feedback in program autotuners and search-based techniques. Finally, compiler cost models can guide future hardware design decisions, while performance-directed feedback cannot.

4.4 Machine Model and IR

This section describes the abstract hardware architecture that Polca targets, and the associated IR constructs that facilitate compiler optimization passes.

4.4.1 Accelerator Architecture

Figure 4.3 illustrates the parallel accelerator architecture supported by Polca. The base abstract architecture – modeled on the Versa [108] and Transmuter [15] designs – is a hierarchical processor composed of tiled scalar cores and software managed memory (either explicitly invalidated cache, or scratchpad). The base architecture assumes an SPMD programming model where cores and tile identifiers are exposed to software as integer indices. Groups of cores participate in data-parallel computations with relaxed memory consistency, where parallel compute is interleaved with barriers that enforce memory ordering at syn-

chronization boundaries. Synchronization is supported at both tile and global scope. Tile scope synchronization appears in two forms: a stronger form that provides ordering over both scratchpad and shared cache (if any), and a weaker form that provides ordering over only intra-tile scratchpad.

Extended functionality and optimization is enabled if the machine supports point-to-point communication between cores, and exposes associated spatial information. In particular, the extended architecture assumes spatial scalar (word-sized) links that enable low-latency data transfer at instruction granularity. For example, the design from Kim et al. [108] implements such links with hardware extensions that allow instruction operands to access the physical register files of adjacent neighbors. We note that the hardware from Kim et al. [108] supports spatial register operands with arbitrary instructions, such that compute and spatial data transfer are fused in a single instruction. For simplicity, Polca's backend only utilizes `mov` instructions for spatial data transfer.

While the abstract machine is designed to capture the traits of specific targets, the compiler is not tied to a specific target except where necessary (*i.e.*, when IR is finally lowered to machine code). Polca optimization passes are ISA-agnostic, accept hardware size/shape information as parameters, and are generally applicable to an arbitrary hardware target, provided the target can satisfy the basic assumptions of the machine model.

4.4.2 Polca Dialect

IR constructs to support Polca functionality are embedded in the MLIR infrastructure as a lightweight collection of IR operations and attributes. These Polca-specific constructs form an MLIR *dialect*, which defines the language-level interfaces between distinct groups of IR and constituent operations.

The Polca dialect is minimal, since the compilation pipeline reuses IR from other dialects in the MLIR ecosystem. *I.e.,* the Polca dialect adds the minimum set of constructs to support custom hardware features and pass optimizations, but does not need to represent

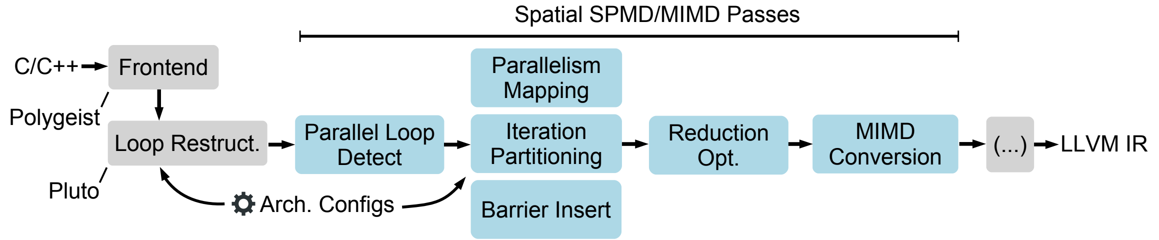


Figure 4.4: Key mid-level analysis and transformation passes.

arithmetic operations, basic loop structures, functions, and so on.

IR operations for work identification and partitioning includes logical core, logical tile, and physical/spatial indices; these are symbolic identifiers that Polca uses to implement SPMD data parallel computations, or to model MIMD control flow in a single unit of IR. The Polca dialect also includes GPU-like memory synchronization barriers (*i.e.*, `polca.mem_barrier`), and associated attributes to specify synchronization scopes. Finally, the Polca dialect adds operations to express spatial data movement, namely two-operand `polca.spatial_<push|pop>` operations that accept a cardinal direction and scalar value. In addition to explicit IR operations, other accelerator-specific details (such as core array geometries) are either passed at compile-time as arguments, or attached to relevant IR in the form of attribute tags as needed.

4.5 Exploiting Versatile Parallelism

This section presents the key components and optimizations in the Polca compilation flow (Figure 4.4).

4.5.1 Frontend

The earliest stages of our compiler re-uses existing IR and transform infrastructure from the MLIR framework. This enables inter-operation and targeting from high-level MLIR dialects such as `Linalg` and `Tensor`, and reuse of frontends for mainstream languages such as C/C++. We use the `Affine` dialect as the main entry-point to our custom infrastructure,

as it is still abstract enough for many important transformations – such as loop tiling – but low enough in the compiler stack to benefit from optimizations shared between mid-level IRs.

4.5.2 Polyhedral Loop Restructuring.

Many real-world programs appear in forms that are not trivially parallelizable, despite parallelism that is intrinsically present in the algorithm. Optimizing for data locality is also critical for performance, but difficult and time consuming to perform manually. Loop restructuring techniques address the problems above by transforming loop nests to expose parallelism and improve locality.

The polyhedral model [] provides a powerful framework for optimizations that operate on loops. In contrast to conventional methods, polyhedral techniques express programs in terms of abstract integer sets and affine transforms, enabling precise analysis of memory dependencies, succinct specification of desired transformations, and global optimization under mathematical objectives. Polca uses Pluto [] as a generic polyhedral optimization backend to perform automatic loop restructuring. Our integration is enabled by recent work improves MLIR interoperability with polyhedral scheduling tools [109]. While Pluto [**pluto**] was originally developed to generate OpenMP for CPUs, we find that modest configuration is sufficient to obtain good schedules for the SPMD architecture described in Section 4.4.1.

4.5.3 SPMD Parallelism.

This set of passes converts abstract loop constructs into concrete IR, and materializes other key operations from IR attributes. In particular, this pass lowers unordered parallel loops into SPMD-like loops, with the iteration space mapped onto thread indices. As an initial implementation heuristic, the iteration partitioning pass maps outermost loops onto space (*i.e.* hardware), and inner loops onto time. Buffer allocation is also required if a mem-

ory reference is marked with IR attributes indicating SPM usage; this consists of an SPM buffer declaration outside the nest, and a new loop inserted within the nest at the appropriate level to materialize SPM initialization. Finally, synchronization operations are inserted within the nest to enforce parallel load-store ordering, for instance, to prevent races on data that is cooperatively buffered and subsequently reused across threads. Synchronization insertion can be implemented in several ways, in particular by analyzing affine dependence vectors and their intersection points (e.g., Bondhugula et al. [110]).

4.5.4 Systolic Reduction Parallelism.

Polca leverages the systolic array-like hardware capabilities described in Section 4.4.1 to efficiently exploit *reduction parallelism*. A reduction is a function $g(f(\cdot), V)$ that applies a reduce operation $f(\cdot)$ to every element in a set of values V, producing up to one result. For example, accumulation over elements of an array and histogram-style counting are two frequently-occurring reduction patterns. In terms of IR, reductions appear in the form of a looped `load` $\rightarrow$ `compute` $\rightarrow$ `store` pattern, where the load-store pair operates on the same memory reference with an access index that is loop-invariant. If a reduction pattern is detected, the reduction load-store pairs are hoisted out of the loop, and the memory reference is converted into a local accumulator value (thereby eliminating intermediate memory operations); in the IR, this value is expressed as a secondary induction variable. In addition to reducing memory pressure, if the reduce operation is associative and there are no aliasing interleaved stores, then the entire abstract reduction function can be parallelized [109]. Crucially, the parallel reduction optimization exposes fine-grained parallelism within inner loops where potential for data reuse is high.

We observe that IR regions containing reductions can be organized into four pieces: 1) an outer (enclosing) parallel loop, 2) a preamble containing hoisted loads, 3) the inner reduction loop, and 4) a postamble containing hoisted stores [Figure 4.5(a)]. Polca maps the pieces above onto hardware *stripes* that correspond to a contiguous 1-D chains of cores.

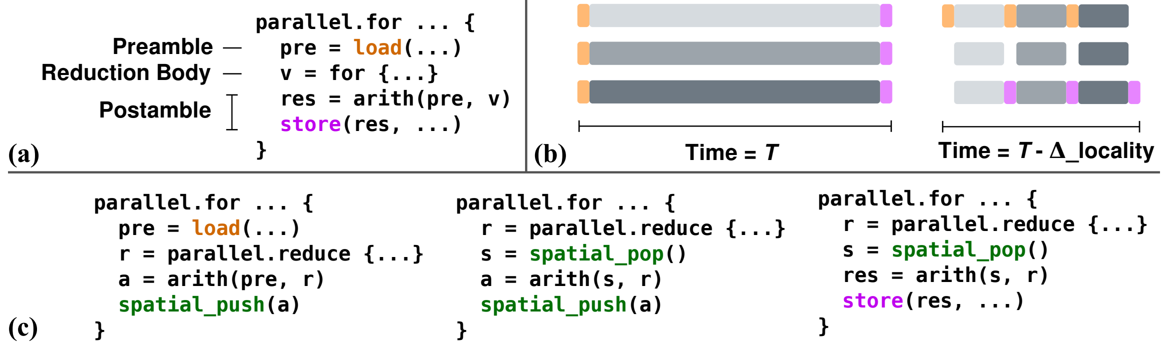


Figure 4.5: Overview of Polca spatial reduction parallelization. **(a)** Operations associated with a reduction loop are tabulated into a preamble, body, and postamble. **(b)** Unparallelized reductions are restricted to parallelism in outer dimensions, limiting data locality. Polca leverages spatial links to parallelize inner reduction loops with low performance overhead, yielding net speedup. **(c)** Pseudocode for functional partitioning of work.

Figure 4.6 illustrates the lowering and mapping strategy. Outer parallel iterations that correspond to independent, full reductions are mapped element-wise onto stripes (cyclic, modulo the number of stripes). Inner iterations of the reduction loop are mapped across cores within the stripe element-wise (cyclic, modulo the stripe length), such that each core performs a partial reduction. Within each stripe, two cores at the spatial edges are designated as 'head' and 'tail' cores (*e.g.*, `row_id`'s 0 and `R-1`, respectively, for vertical stripes). In addition to inner reduction iterations, head and tail cores also execute the initial loads and concluding stores, respectively. Thus, a complete inner reduction after transformation corresponds to initial loads by the leader, parallel partial reductions, intra-stripe aggregation, and concluding stores in the tail's postamble.

Notably, deferring the intra-stripe reduction reduces data movement substantially, compared to a naïve scheme that reduces across cores with every inner iteration. In addition, because spatial data transfers lower to inter-core register-register instructions [108], Polca's reduction scheme is more performant than conventional reduction strategies that require atomics (*e.g.*, on CPUs).

Differentiated execution (*e.g.*, between head and tail cores) in transformed IR is expressed with affine predication on `row_id` and `col_id` symbolic indices. That is, core-

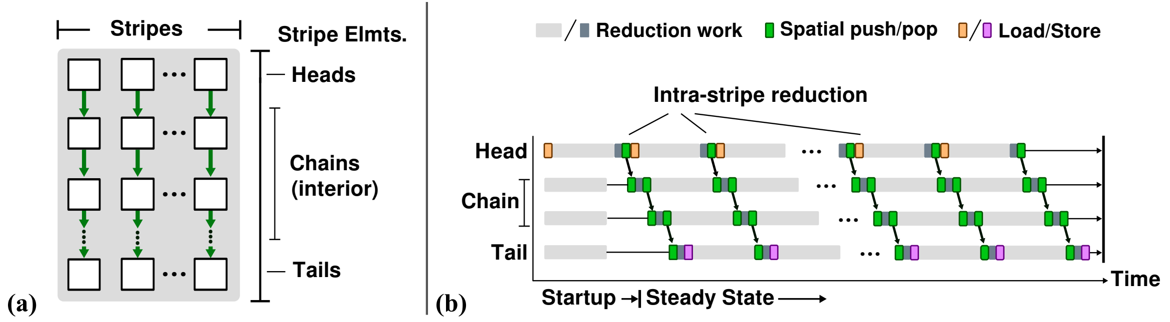


Figure 4.6: Physical and temporal views of the optimized reduction strategy. **(a)** Outer-parallel iterations are mapped onto hardware stripes (vertical stripes shown), with one stripe executing an entire parallel reduction. **(b)** Temporal view of execution for one stripe. Intra-stripe reduction in core postambles is used to aggregate per-core (parallel) local reductions.

specific operations are guarded in an `affine.if` region with an affine constraint (`affine_set` in MLIR). Language-level predication at this point does not correspond to executed code, however. Subsequent MIMD conversion simplifies the IR for each core, resulting in code that is free of index-based predication.

4.5.5 MIMD Conversion.

The MIMD conversion pass lowers the SPMD-like IR into independent units of scalar code. This pass operates in two steps, followed by downstream passes that simplify and DCE unnecessary operations. The IR is first replicated to produce an independent unit of kernel code for each core, with all per-core IR units encapsulated in a parent module. Per-core IR is tagged with identification attributes in the process, followed by rewrites on symbolic indices. All symbolic indices, such Polca `core_id`'s, `tile_id`'s, etc., are substituted for constant values (*i.e.*, `arith.constant` operations). With runtime core identification removed, regions predicated on core indices and other index-based computations are simplified or DCE'd entirely in downstream lowering.

```
// I, J, K              := Loop bounds
// N_TILES, N_STRIPES := Number of hardware tiles, stripes per tile
// LEN_STRIPE         := Hardware stripe length

#is_head = affine_set<()[s0] : (s0 == 0)>
#is_tail = affine_set<()[s0] : (s0 - LEN_STRIPE + 1 == 0)>

func @kernel_..._wrap(...) {
  %tid = polca.tile_id
  %sid = polca.stripe_id
  %eid = polca.stripe_elem_id
  (...)
  affine.parallel (%i) = (symbol(%tid)) to (I) step (N_TILES) {
    affine.parallel (%j) = (symbol(%sid)) to (min(%i + 1, J)) step (N_STRIPES) {
      // Preamble (head only)
      %10 = affine.if #is_head()[%eid] -> f32 {
        %13 = affine.load %arg4[%i, %j] : memref<IxJxf32>
        affine.yield %13 : f32
      } else { // this else-branch is for IR legality only
        %cst_0 = arith.constant 0.000000e+00 : f32
        affine.yield %cst_0 : f32
      }
      // Reduction body (all cores)
      %11 = affine.parallel (%k) = ↩
          (symbol(%eid)) to (K) step (LEN_STRIPE) reduce ("addf") -> (f32) {
        %13 = affine.load %arg5[%j, %k] : memref<JxKxf32>
        %14 = arith.mulf %13, %cst : f32
        %15 = affine.load %arg6[%i, %k] : memref<IxKxf32>
        %16 = arith.mulf %14, %15 : f32
        %17 = affine.load %arg6[%j, %k] : memref<JxKxf32>
        %18 = arith.mulf %17, %cst : f32
        %19 = affine.load %arg5[%i, %k] : memref<IxKxf32>
        %20 = arith.mulf %18, %19 : f32
        %21 = arith.addf %16, %20 : f32
        affine.yield %21 : f32
      }
      // Postamble
      // (head)
      %12 = affine.if #is_head()[%eid] -> f32 {
        %13 = arith.addf %10, %11 : f32
        affine.yield %13 : f32
      // (chain and tail)
      } else {
        %13 = polca.spatial_pop N : f32
        %14 = arith.addf %13, %11 : f32
        affine.yield %13 : f32
      }
      // (tail)
      affine.if #is_tail()[%eid] {
        affine.store %12, %arg4[%i, %j] : memref<IxJxf32>
      // (head and chain)
      } else {
        polca.spatial_push S %12 : f32
      }
    }
  }
  polca.mem_barrier all
  (...)
  return
}
```

Figure 4.7: Excerpt of transformed IR for reduction, nested in a tiled parallel loop. Identifiers edited for clarity. Affine predication on symbolic spatial indices (`tid, sid, eid`) models differentiated execution. Symbols are later substituted for constants during MIMD conversion; downstream simplification passes produce predication-free IR for each core.

4.5.6 Final Lowering

Polca largely maintains the loop-based structure of the IR until final lowering to LLVM IR. At this stage, a few operations use custom lowering to LLVM IR with a `polca-to-llvm` conversion pass to maintain ease of integration with accelerator C++ support libraries. `memref.alloc` memory allocations are lowered to `llvm.call`'s that invoke accelerator malloc routines. Similarly, other coarse-grained Polca dialect operations are converted into `llvm.call` operations that invoke accelerator library counterparts. Any library-side calls that return/accept bare pointers have additional handling that statically pack/unpack `memrefs` for type-compatibility. For `memrefs` in function signatures, we emit LLVM IR with the bare pointer calling convention so that generated kernels are callable from C++ without special handling. Spatial data transfer operations (discussed further in Section 4.6) correspond 1:1 with instructions, and are lowered to LLVM IR intrinsics.

4.6 Evaluation

This section presents the experimental methodology, benchmarks, and evaluation results.

Benchmarks. We evaluate Polca on PolyBench/C [111], a well-known suite of computational kernels from linear algebra, signal processing, and optimization problems. PolyBench kernels are naïve, single-threaded C programs containing static control parts (SCoPs) []. The Polygeist frontend natively generates IR that strips `memref` datatypes of shape information; this removes information present in the original C function signatures, and impedes downstream affine analysis. We patch the frontend to preserve `memref` shape information, and invoke kernels from C++ with the bare pointer calling convention as described in Section 4.5.6. Kernels are compiled separately from wrapping C++ testbench code (*i.e.*, without inlining or DCE that crosses function call boundaries). Using the `mini` dataset size, wall time to execute a set of kernels in RTL simulation is bounded to ~10-12 hours. All but

7 kernels compile successfully, and 7 of the 23 evaluated kernels contain reduction loops amenable to systolic reduce optimizations.

Hardware. Polca uses Versa [108], a prototype accelerator design previously validated in silicon, as the specific hardware target for experiments **??**. The programmable SPMD/MIMD core arrays are implemented in Versa with modified Arm Cortex M4F cores[1] All kernels are evaluated with single-precision floating point, supported in Versa cores by hardware floating point units. Machine code generation for Versa entails translation of LLVM IR to ARMv7 assembly, followed by assembly and linking using standard GNU tools (configured for Versa's bare-metal software environment). Unless noted otherwise, performance for all experiments is measured in cycle-exact RTL simulation with Versa's development infrastructure. Performance measurements count only the time required for the kernel function call, including final barrier synchronization.

The fine-grained spatial communication required for reduction acceleration is supported with Versa's prototype *register-to-register (R2R)* links. R2R links form connections between the floating point (FP) register files of adjacent cores, and are addressable as instruction operands that alias to four standard FP registers (one register per cardinal direction). Because R2R operates on physical machine registers, which are not exposed at the IR level, the final lowering stage in Polca translates spatial data transfers to LLVM IR intrinsics. Corresponding LLVM IR intrinsics are subsequently custom lowered in LLVM `llc` during instruction selection, using a modified version of `llc`'s ARM backend. Similarly, we modify routines relevant to ARMv7 register allocation to prevent clobbering and auto-allocation of R2R registers.

Compilation Variants. We evaluate Polca with several compilation flows that successively enable proposed optimizations. All variants share the same frontend pre-processing,

[1] The Arm Cortex M4F is a in-order single-issue scalar core; this makes the M4F core very efficient in terms of silicon area and power.

and differentiation is restricted to points after initial lowering from C++ to MLIR. Unless noted otherwise, all compilation variants share the same options for lowering, and the same support utility passes (*e.g.*, DCE, CSE, constant propagation) with a common pass ordering. Evaluated compilation flows include:

- `sequential`: Sequential execution without any restructuring optimizations.
- `tiled`: Sequential execution with polyhedral tiling optimizations only, and all parallelization passes disabled.
- `tiled-parallel`: MIMD parallel execution with polyhedral tiling and loop restructuring for parallelism.
- `r2r-reduce`: MIMD parallel execution with tiling, restructuring, and R2R-based systolic reduction parallelization.

`tiled` aims to distinguish the impact of data locality optimizations alone. Thus, `tiled` enables polyhedral tiling but disables all parallelization optimizations, and downstream IR is lowered for sequential execution on a single Versa core. Relative to `tiled`, `tiled-parallel` enables parallelization options in the polyhedral backend (tiling, in addition to transforms that expose parallelism such as skewing, fission, etc.), in addition to parallelization passes for MIMD execution. The `r2r-reduce` pass consumes the same IR as `tiled-parallel` at the SPMD stage, but inner reduction loops are restructured as described in Section 4.5.4 prior to MIMD conversion.

Performance Breakdown. Tiling alone (Figure 4.8: `tiled`) improves performance by up to 6.3× (down to 0.3×) compared to unoptimized sequential kernels. This is entirely due to changes in spatial-temporal data locality across multiple loop nests, where restructuring is beneficial for the majority of kernels. 5 kernels do not exhibit speedups with tiling, however, largely due to the addition of loop control overheads that outweigh small improvements in locality. Small data footprints that limit potential benefits from tiling also

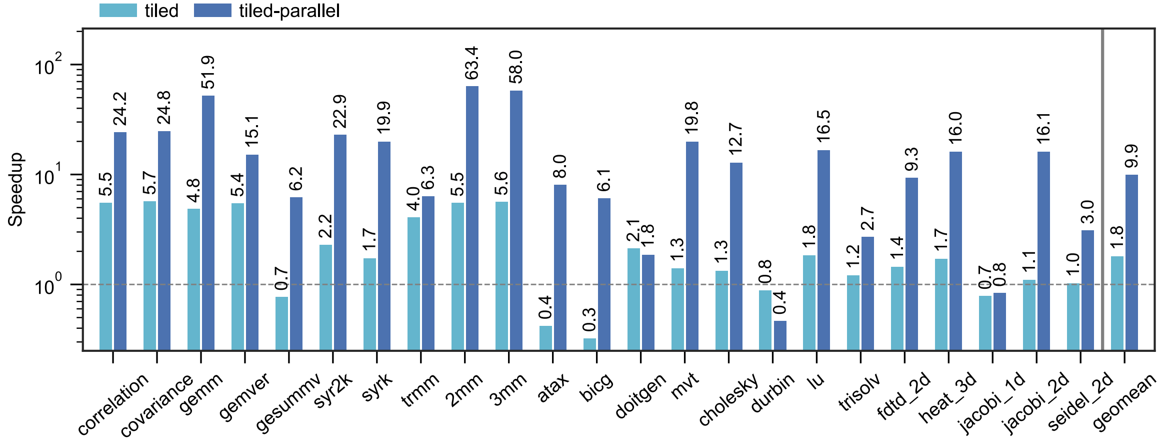


Figure 4.8: Speedups normalized to baseline `sequential` execution. `tiled` shows sequential execution with tiling and basic restructuring optimizations enabled, while `tiled-parallel` enables both tiling and base parallelization optimizations.

likely contribute to slowdowns in `durbin` and `bicg`. Overall, tiling provides 1.8× geomean speedup compared to the sequential baseline.

Parallelization with tiling (Figure 4.8: `tiled-parallel`) results in up to 63.4× speedup (down to 0.4×) over baseline sequential execution. Under this compilation variant, polyhedral outer-tiling typically produces tiled loop nests with two or more parallelizable bands. Parallel bands typically reside on outer and inner edges of the nest, with non-parallel intermediate bands that capture dependencies between data tiles. Any non-parallel inner bands that exist are executed serially per core (with parallel execution along outer bands). Thus, outer-parallel loop nests map naturally onto Versa hardware, which is structurally organized into hardware tiles and core arrays. One notable exception is the `durbin` kernel, where the polyhedral backend finds parallel inner loops but no outer-parallelism; sequential work and synchronization overheads dominate in this case. Otherwise, most kernels exhibit parallelization efficiency of ~20-50%, translating to 9.9× geomean speedup overall.

Reduction operations that can be meaningfully accelerated are detected in 7 kernels (Figure 4.9: `r2r-reduce`). A reduction is also detected in `durbin`, but the loop is not a valid candidate for mapping due to the lack of nested parallelism.[2] Compared to the

[2] The non-nested reduction on `durbin` could hypothetically be parallelized, but resulting speedup would likely be small regardless (due to the absence of outer-parallel loops).

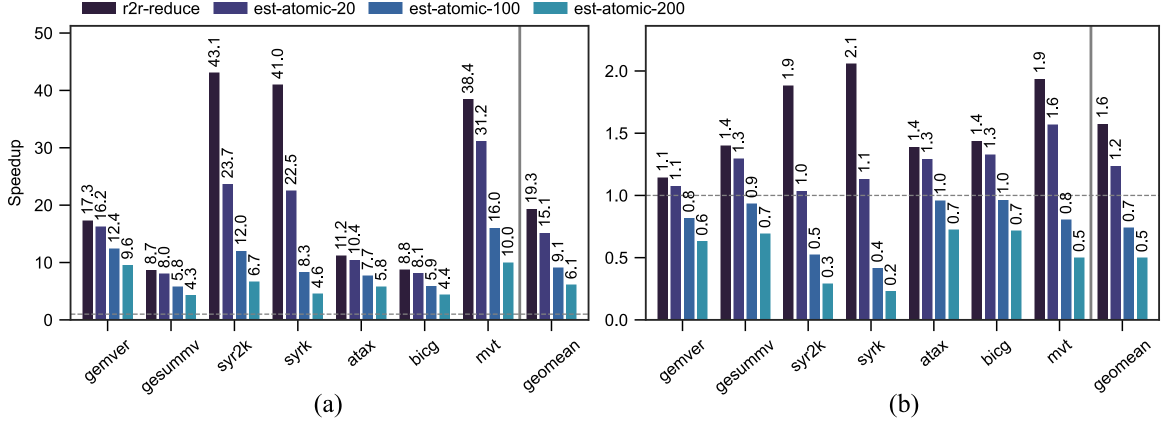


Figure 4.9: Speedups for kernels amenable to systolic reduction parallelization. `r2r-reduce` enables all optimizations for tiling, base parallelization, and reduction parallelization. `est-atomic-<20,100,200>` labels denote estimated speedups when R2R-based spatial data transfers are substituted for 20-200ns atomic RMWs. **(a)** Cumulative speedup over baseline `sequential` execution, and **(b)** relative speedup over `tiled-parallel`.

sequential baseline [Figure 4.9(a)], `r2r-reduce` results in up to 43.1× speedup (down to 8.8×). However, we are more interested with how optimized reductions perform relative to the 'baseline' parallel execution of `tiled-parallel`; Figure 4.9(b) illustrates up to 2.1× speedup (down to 1.1×) in this case. For the `r2r-reduce` kernel subset, enabled by R2R-based spatial links, reduction optimizations result in 19.3× and 1.6× geomean speedups over the sequential and parallel baselines, respectively.

Substituting Spatial Data Transfer. Reductions accelerated using R2R spatial links can also be compared to parallelized reductions that use an alternative hardware mechanism. Because the exact number of spatial data transfers along the loop critical path is known, in addition to R2R latencies, we can estimate the impact of alternative link implementations. The closest alternative to the R2R-based reduction is likely an implementation that uses the same mapping, but substitutes R2R with atomic read-modify-write (RMW) operations.

Since the latency of atomic RMWs varies, depending on the underlying hardware and runtime contention levels, we evaluate three abstract atomics that exhibit latencies of 20ns, 100ns, and 200ns per update (*e.g.*, latency per `atomic_add()`). Changes in kernel execution times are calculated by replicating affine loop nests for the 7 applicable kernels in python,

and counting the number of updates (N) on reduction accumulator values along the parallelized critical path. The new runtime is then estimated as $t' = t - N(L_{\texttt{r2r}} - L_{\texttt{atomic}})$, where t is the runtime from measurement, and $L_{\texttt{r2r}}$ / $L_{\texttt{atomic}}$ is the latency per R2R/atomic operation.

`est-atomic` labels in Figure 4.9 illustrate the resulting speedups when R2R, with 4ns per transfer (two cycles), is substituted for variable-latency atomic RMWs. The first observation is that changes in performance are not linear with respect to latency; this is expected since reduction loops constitute a fraction (< 1) of execution time. In addition, some kernels are impacted more than others; for instance, `syr2k` and `syrk` exhibit steep drops in performance, while `gemver` and `gesummv` show smaller changes. In this case we observe that `syr2k` and `syrk` both contain a 2D→3D loop sequence, where the innermost band of the 3D loop constitutes the reduction. In contrast, `gemver` contains a 4D→2D→3D loop sequence, with reduction in the final 3D loop.

The most important consideration is: at what point does atomic latency negate the benefits of parallelizing reductions entirely? Figure 4.9(b) shows that most kernels start to exhibit slowdowns under a 20ns atomic RMW. At 100ns latency, all kernels exhibit slowdowns and the geomean speedup over `tiled-parallel` is 0.7×, indicating that parallelized reductions offer no benefit. Especially at higher core counts, atomics are subject to greater contention and non-deterministic latencies grow proportionally [83, 112]. Thus, the results above demonstrate the value of hardware support for systolic-inspired spatial data transfer, and how it can be exploited by compilers.

Ablation Summary. Table 4.1 summarizes the performance of kernels compiled with different Polca optimizations. Considering all kernels [Table 4.1(a)], base parallelization provides 5.53× relative speedup (9.88× total) over tiling, with an additional 1.16× (11.46× total) provided by reduction optimizations. A more precise ablation is obtained if we consider the 7 kernels where reduction optimizations apply [Table 4.1(b)]. In this case,

Table 4.1: Summary of average (geomean) speedup across kernels. 'Rel.' denotes relative speedup obtained by stacking optimizations, while 'Total' denotes total (cumulative) speedup over sequential.

(a) Speedup over all kernels.

	+tiled	**+parallel**	**+r2r-reduce**
Rel.	-	5.53×	1.16×
Total	1.79×	9.88×	11.46×

(b) Speedup on `r2r-reduce` subset.

	+tiled	**+parallel**	**+r2r-reduce**
Rel.	-	10.40×	1.57×
Total	1.18×	12.27×	19.31×

the relative speedup contributions for base parallelization and reduction optimization both grow to 10.40× and 1.57×, respectively, translating to 12.27× and 19.31× total speedup. While reduction optimizations are not applicable to all kernels, these results are encouraging and suggest Polca's optimizations are effective. The results also suggest future directions – for instance the detection of additional parallel patterns (beyond reductions) that might be exploited with compiler-hardware co-design.

4.7 Related Work

This section discusses previous work on polyhedral methods, compilation for accelerators, and related techniques for systolic array-like hardware.

While polyhedral compilation is not new,[3] recent advances have made polyhedral methods more practical and accessible for a variety of platforms. Bondhugula et al. [110] (Pluto) and Baskaran et al. [114] were among the first to demonstrate end-to-end polyhedral optimization of parallelism and locality for CPUs and GPUs, respectively. In particular, the optimization strategies from Pluto are general enough such that many subsequent works have been able to reuse or improve on variants of the Pluto algorithm (*e.g.*, PPCG [115] and our work).

There is also a large body of historical work on constructing systolic array hardware and appropriate algorithm mappings [116–118], in addition to subsequent efforts that em-

[3] Cedric Bastoul's thesis [113] contains a good overview of techniques, historical references, and recent tools.

ploy generalized polyhedral methods [119–123]. These efforts use the same underlying mathematical machinery of polyhedral and affine analysis as our work, but towards a different end; *i.e.*, the works above use polyhedral methods to synthesize different systolic array designs from algorithm kernels, while Polca compiles a variety of algorithm kernels to a common spatial architecture.

Building on traditional polyhedral methods, several efforts have developed compilers and domain-specific languages (DSLs) focused on accelerator programs, in particular for image processing [124, 125] and machine learning [126–130]. PolyMage [124] introduces a DSL and auto-tuning framework for image processing pipelines, leveraging model-driven search and polyhedral techniques that match or outperform comparable Halide pipelines. Pencil [125] is a C99-based DSL and polyhedral compiler that supports multiple CPU and GPU backends starting from a single platform-agnostic kernel implementation; affine and non-affine kernels are supported through OpenMP-like directives. Tensor Comprehensions [126] (TC) develops a DSL to express end-to-end DAGs of ML operators, and a polyhedral JIT compiler with auto-tuning functionality. While the DSLs above target either CPU or GPU platforms, Verdoolaege et al. [128] and Zhao et al. [130] develop compilers and employ polyhedral techniques for specific ML accelerators; the former for generation of SIMD instructions on the Cerebras CS-1, and the latter for loop transformations and memory management on the Huawei Ascend 910.

Polca and the works above all address different problems associated with the generation of optimized kernel libraries, but Polca places more emphasis on optimizations that exploit the Versa spatial SPMD/MIMD accelerator prototype. In addition, PolyMage and TC are motivated by context-dependent optimization opportunities that arise in sequences of kernels (*e.g.*, in ML operator graphs and image processing pipelines); PolyMage and TC focus on operator fusion in particular. In this sense, Polca is most similar to Verdoolaege et al. [128] and Zhao et al. [130], which target novel accelerator cores, and focus on transformations within isolated kernels or operator sub-graphs.

Polca bears some similarity to previous works that develop systolic array-like programming models for mainstream processor architectures [131–134]. Suriana et al. [131] introduces a scheduling primitive called `rfactor` to exploit reduction parallelism in Halide programs. In contrast to Polca, where reductions are automatically exploited using pattern-based detection, Halide developers use `rfactor` directives explicitly in the schedule specification, obviating the need for modifications to the corresponding algorithm code. Chen et al. [132], Rong et al. [133], and Thuerck et al. [134] develop programming abstractions and compilation methods to semi-automatically exploit systolic array-like data movement over register caches. Shuffle-based data movement between threads in the GPU register cache idiom[4] is similar to the fine-grained inter-core data movement exploited by Polca. The methods above involve DSLs and GPU-specific abstractions, while Polca hides spatial data transfers from the programmer, and targets an accelerator with native support for systolic data movement between scalar cores.

4.8 Conclusion

This chapter presented Polca, an end-to-end compiler for versatile SPMD/MIMD parallel hardware. Polca incorporates polyhedral optimization to enable parallelization and critical loop transformations, and exploits progressive lowering from high-level C programs. Progressive lowering within the MLIR infrastructure enables rapid construction of an entire toolchain that runs on prototype silicon. Polca adds an accelerator IR extension to exploit core-to-core communication features, enabling an efficient form of reduction parallelism. Evaluated on 23 kernels from the Polybench suite, Polca optimizations result in 11.5× average speedup (up to 63.4×). For kernels where reduction parallelism is exposed, Polca's R2R-based systolic reduction results in 19.3× average speedup (up to 43.1×).

There are a number of avenues for future work. For example, Polca uses spatial data

[4] Originating from Ben-Sasson et al. [135], the register cache idiom is now a fairly well-known GPU optimization technique that enhances intra-warp data reuse over registers.

transfer in a restricted manner while the hardware allows for arbitrary 2-D communication patterns; this flexibility is likely amenable to a variety of parallel patterns, provided they can be detected reliably. Polca also does not exploit spatial instructions that fuse arithmetic and data movement, which will yield additional performance in tight inner loops. Finally, hardware techniques such as coarse-grained memory reconfiguration [103, 108] create additional opportunities for automatic optimization.

Chapter V

Conclusion

This dissertation is motivated by the end of classical scaling in the performance and energy-efficiency of hardware-software systems. With the departure from historical semiconductor scaling trends, domain experts, system architects, and engineers are no longer able to rely solely on general-purpose hardware to satisfy application requirements. Thus, greater emphasis is placed on new paradigms in hardware and software, including co-designed solutions that leverage components from both. While fixed-function ASICs can provide performance and energy-efficiency orders of magnitude better than general-purpose counterparts, they come with steep tradeoffs in flexibility and are only practical for narrow sub-domains. This leaves ample room for flexible accelerators that strike a different balance between specialization and efficiency, and that leverage opportunities for co-design with accelerator programming models.

To address the challenges associated with flexible accelerator design, this dissertation explored the idea of flexibility in software-programmable manycore hardware; such an accelerator architecture is not only semi-specialized with ideas from ASIC design, such as systolic dataflow, but also exploits algorithm-specific opportunities that arise at runtime through fast reconfiguration. In particular, we explored the problem and developed solutions in three pieces of research: 1) Transmuter (Chapter II), a comprehensive, simulation-based study of architectural requirements; 2) Versa (Chapter III), an architectural refine-

ment and concrete silicon prototype; and 3) Polca (Chapter IV), an end-to-end compiler enabling automatic optimization over multiple levels of parallelism and locality.

In conclusion, prospects for semiconductor scaling are dim; in order to improve the pareto frontier of performance and energy efficiency, future accelerators must blur the lines between fixed-function and general-purpose designs without sacrificing flexibility. As this dissertation demonstrates, such accelerator designs obtain such gains by exploiting context- and algorithm-specific opportunities in both hardware and software, thereby enabling a variety of future applications.

Bibliography

[1] Robert R. Schaller. "Moore's Law: Past, Present and Future." In: *IEEE Spectrum*. Vol. 34. 6. 1997, pp. 52–59.

[2] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions." In: *IEEE Journal of Solid-State Circuits (JSSC)*. Vol. 9. 5. 1974, pp. 256–268.

[3] Mark Horowitz, Elad Alon, Dinesh Patil, Samuel Naffziger, Rajesh Kumar, and Kerry Bernstein. "Scaling, Power, and the Future of CMOS." In: *IEEE International Electron Devices Meeting (IEDM)*. 2005, pp. 7–15.

[4] Karl Rupp. *50 Years of Microprocessor Trend Data*. 2022. URL: https://github.com/karlrupp/microprocessor-trend-data.

[5] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. "CPU DB: Recording Microprocessor History." In: *Communications of the ACM*. Vol. 55. 4. 2012, pp. 55–63.

[6] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376.

[7] Michael B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse." In: *ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2012, pp. 1131–1136.

[8] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2014, pp. 13–24.

[9] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. "A Configurable Cloud-Scale DNN Processor for Real-Time AI." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2018, pp. 1–14.

[10] Mingyu Gao and Christos Kozyrakis. "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 126–137.

[11] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 724–736.

[12] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing." In: *IEEE Micro*. Vol. 32. 5. 2012, pp. 38–51.

[13] Kizheppatt Vipin and Suhaib A. Fahmy. "FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications." In: *ACM Computing Surveys*. Vol. 51. 4. 2018, pp. 1–39.

[14] Xilinx Inc. *Vivado Design Suite User Guide: Partial Reconfiguration (UG909 v2018.1)*. 2018. URL: https://docs.xilinx.com/v/u/2018.1-English/ug909-vivado-partial-reconfiguration.

[15] 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: *ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2020, pp. 175–190.

[16] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. "Whirlpool: Improving Dynamic Cache Management with Static Data Classification." In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2016, pp. 113–127.

[17] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. "Stash: Have Your Scratchpad and Cache It Too." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2015, pp. 707–719.

[18] Sudhir Satpathy, Reetuparna Das, Ronald Dreslinski, Trevor Mudge, Dennis Sylvester, and David Blaauw. "High Radix Self-Arbitrating Switch Fabric With Multiple Arbitration Schemes and Quality of Service." In: *ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2012, pp. 406–411.

[19] Transmuter Software Demo (DARPA SDH). *TransPy Documentation*. 2019. URL: https://transpy.readthedocs.io.

[20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array programming with NumPy." In: *Nature*. Vol. 585. 7825. Sept. 2020, pp. 357–362.

[21] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods*. Vol. 17. 2020, pp. 261–272.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019, pp. 8024–8035.

[23] Aric Hagberg, Pieter Swart, and Daniel S. Chult. *Exploring Network Structure, Dynamics, and Function Using NetworkX*. Tech. rep. Los Alamos National Lab (LANL), Los Alamos, NM (United States), 2008.

[24] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 — Seamless operability between C++11 and Python*. 2016. URL: https://github.com/pybind/pybind11.

[25] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems." In: *Computing in Science Engineering*. Vol. 12. 3. 2010, pp. 66–73.

[26] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation." In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 74–85.

[27] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels." In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019, pp. 65–78.

[28] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F. P. O'Boyle. "Bind the Gap: Compiling Real Software to Hardware FFT Accelerators." In: *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2022, pp. 687–702.

[29] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication." In: *ACM Transactions on Mathematical Software (TOMS)*. Vol. 34. 3. 2008, pp. 1–25.

[30] Clint R. Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated empirical optimizations of software and the ATLAS project." In: *Parallel Computing*. Vol. 27. 1-2. 2001, pp. 3–35.

[31] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks." In: *IEEE Journal of Solid-State Circuits (JSSC)*. Vol. 52. 1. 2016, pp. 127–138.

[32] Benjamin C. Lee, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. "Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-Vector Multiply." In: *International Conference on Parallel Processing (ICPP)*. 2004, pp. 169–176.

[33] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms." In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1991, pp. 63–74.

[34] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. "Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices." In: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 2020, pp. 1–12.

[35] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. "Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization." In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019, pp. 821–834.

[36] Preston A. Jackson, Cy P. Chan, Jonathan E. Scalera, Charles M. Rader, and M. Michael Vai. "A Systolic FFT Architecture for Real Time FPGA Systems." In: *IEEE High Performance Extreme Computing Conference (HPEC)*. 2004.

[37] Earl E. Jr. Swartzlander. "Systolic FFT Processors: Past, Present and Future." In: *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2006, pp. 153–158.

[38] Mario Garrido, Jesús Grajal, MA Sanchez, and Oscar Gustafsson. "Pipelined Radix-2^k Feedforward FFT Architectures." In: *IEEE Transactions on VLSI Systems (TVLSI)*. Vol. 21. 1. 2011, pp. 23–32.

[39] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. "The M5 Simulator: Modeling Networked Systems." In: *IEEE Micro*. 4. 2006, pp. 52–60.

[40] 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." In: *SIGARCH Computer Architecture News*. Vol. 39. 2. 2011, pp. 1–7.

[41] Korey Sewell, Ronald G. Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F. Wenisch, Dennis Sylvester, David Blaauw, and Trevor Mudge. "Swizzle-Switch Networks for Many-Core Systems." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*. Vol. 2. 2. 2012, pp. 278–294.

[42] Supreet Jeloka, Reetuparna Das, Ronald G. Dreslinski, Trevor Mudge, and David Blaauw. "Hi-Rise: A High-Radix Switch for 3D Integration With Single-Cycle Arbitration." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2014, pp. 471–483.

[43] Sudhir Satpathy, Korey Sewell, Thomas Manville, Yen-Po Chen, Ronald Dreslinski, Dennis Sylvester, Trevor Mudge, and David Blaauw. "A 4.5 Tb/s 3.4 Tb/s/W 64×64 Switch Fabric with Self-Updating Least-Recently-Granted Priority and Quality-Of-Service Arbitration in 45nm CMOS." In: *IEEE International Solid-State Circuits Conference (ISSCC)*. 2012, pp. 478–480.

[44] Norman P. Jouppi, Andrew B. Kahng, Naveen Muralimanohar, and Vaishnav Srinivas. "CACTI-IO: CACTI With Off-Chip Power-Area-Timing Models." In: *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2012, pp. 294–301.

[45] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, Jonathan Beaumont, Kuan-Yu Chen, Chaitali Chakrabarti, Michael Bedford Taylor, Trevor Mudge, David Blaauw, Hun-Seok Kim, and Ronald G. Dreslinski. "A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix-Matrix Multiplication Accelerator." In: *IEEE Journal of Solid-State Circuits (JSSC)*. Vol. 55. 4. 2020, pp. 933–944.

[46] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2017, pp. 41–54.

[47] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. "Smart Memories: A Modular Reconfigurable Architecture." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2000, pp. 161–171.

[48] Brucek Khailany, William J Dally, Ujval J Kapasi, Peter Mattson, Jinyung Namkoong, John D Owens, Brian Towles, Andrew Chang, and Scott Rixner. "Imagine: Media Processing with Streams." In: *IEEE Micro*. Vol. 21. 2. 2001, pp. 35–46.

[49] Lance Hammond, Benedict A Hubbert, Michael Siu, Manohar K Prabhu, Michael Chen, and Kunle Olukolun. "The Stanford Hydra CMP." In: *IEEE Micro*. Vol. 20. 2. 2000, pp. 71–84.

[50] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2007, pp. 186–197.

[51] 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: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2012, pp. 305–316.

[52] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald G. Dreslinski, Thomas F. Wenisch, and Scott Mahlke. "Exploring Fine-Grained Heterogeneity with Composite Cores." In: *IEEE Transactions on Computers (TOC)*. Vol. 65. 2. 2016, pp. 535–547.

[53] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. "Coarse Grained Reconfigurable Architectures in the Past 25 Years: Overview and Classification." In: *IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 2016, pp. 235–244.

[54] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R. Reed Taylor. "PipeRench: A Reconfigurable Architecture and Compiler." In: *IEEE Computer*. Vol. 33. 4. 2000, pp. 70–77.

[55] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. "The Reconfigurable Streaming Vector Processor (RSVP)." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2003, pp. 141–150.

[56] K. Sankaralingam, R. Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, D. Burger, S.W. Keckler, and C.R. Moore. "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2003, pp. 422–433.

[57] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2006, pp. 480–491.

[58] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. "The WaveScalar Architecture." In: *ACM Transactions on Mathematical Software (TOMS)*. Vol. 25. 2. 2007, pp. 1–54.

[59] 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: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2017, pp. 389–402.

[60] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. "Spatial: A Language and Compiler for Application Accelerators." In: *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2018, pp. 296–311.

[61] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. "Stream-Dataflow Acceleration." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2017, pp. 416–429.

[62] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. "Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2019, pp. 924–939.

[63] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." In: *International Conference on Learning Representations (ICLR)*. 2015.

[64] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. "Deep Gradient Compression: Reducing the Communication Bandwidth For Distributed Training." In: *International Conference on Learning Representations (ICLR)*. 2017.

[65] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Tech. rep. Stanford InfoLab, 1999.

[66] Jaewon Yang and Jure Leskovec. "Defining and Evaluating Network Communities Based on Ground-Truth." In: *Knowledge and Information Systems*. Springer, 2015, pp. 181–213.

[67] Ben Langmead and Steven L Salzberg. "Fast Gapped-Read Alignment with Bowtie 2." In: *Nature Methods*. Vol. 9. 4. 2012, pp. 357–359.

[68] Heng Li and Nils Homer. "A Survey of Sequence Alignment Algorithms for Next-Generation Sequencing." In: *Briefings in Bioinformatics*. Vol. 11. 5. 2010, pp. 473–483.

[69] Ping Xiang, Yi Yang, and Huiyang Zhou. "Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 284–295.

[70] Dirk Koch. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer Science & Business Media, 2012.

[71] 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 Dataflow-Centric Multiprocessor with 36 Systolic ARM Cortex-M4F Cores and a Reconfigurable Crossbar-Memory Hierarchy in 28nm." In: *IEEE Symposium on VLSI Circuits (VLSI)*. 2021, pp. 1–2.

[72] Jim Kahle. "The Cell Processor Architecture." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2005, pp. 3–3.

[73] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. "Cache Coherence for GPU Architectures." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 578–590.

[74] Norman P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 1990, pp. 364–373.

[75] 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, Daniel Killebrew, 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: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2017, pp. 1–12.

[76] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*. Vol. 9. 2. 2019, pp. 292–308.

[77] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration." In: *ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2021.

[78] Mark Roth, Micah J. Best, Craig Mustard, and Alexandra Fedorova. "Deconstructing the Overhead in Parallel Applications." In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2012, pp. 59–68.

[79] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. "Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing." In: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 2013, pp. 1–11.

[80] Shucai Xiao and Wu chun Feng. "Inter-block GPU Communication Via Fast Barrier Synchronization." In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2010, pp. 1–12.

[81] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. "NOMAD: Non-locking, Stochastic Multi-machine Algorithm for Asynchronous and Decentralized Matrix Completion." In: *arXiv*. 2013.

[82] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization." In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2015, pp. 2737–2745.

[83] Michael L. Scott. "Shared-memory Synchronization." In: *Synthesis Lectures on Computer Architecture*. Vol. 8. 2. Morgan & Claypool Publishers, 2013.

[84] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-yeon Wei, and David Brooks. "MachSuite: Benchmarks for Accelerator Design and Customized Architectures." In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2014, pp. 110–119.

[85] *Arm Cortex-M4 Processor Technical Reference Manual*. 100166_0001_04_en. Rev. r0p1. Arm Ltd. 2020.

[86] 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: *ACM SIGARCH Computer Architecture News*. Vol. 32. 2. 2004, p. 2.

[87] Florian Zaruba, Fabian Schuiki, and Luca Benini. "Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultra-Efficient Floating-Point Computing." In: *IEEE Micro*. Vol. 41. 2. 2020, pp. 36–42.

[88] Ditzel Dave, Espasa Roger, Aymerich Nivardl, Baum Allen, Berg Tom, Burr Jim, Hao Eric, Iyer Jayesh, Izquierdo Miquel, Jayaratnam Shankar, Jones Darren, Klingner Chris, Kim Jin, Lee Stephen, Lupon Marc, Magklis Grigorios, Maric Bojan, Nath Rajib, Neilly Mike, Northcutt Duane, Orner Bill, Renau Jose, Reves Gerard, Reves Xavier, Riordan Tom, Sanchez Pedro, Samudrala Sri, Sole Guillem, Tang Raymond, Thorn Tommy, Torres Francisco, Tortella Sebastia, and Yau Daniel. "Accelerating ML Recommendation with over a Thousand RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip." In: *IEEE Hot Chips Symposium (HCS)*. 2021.

[89] Paul N. Whatmough, Sae Kyu Lee, Marco Donato, Hsea-Ching Hsueh, Sam Likun Xi, Udit Gupta, Lillian Pentecost, Glenn G. Ko, David M. Brooks, and Gu-Yeon Wei. "A 16nm 25mm2 SoC With a 54.5x Flexibility-Efficiency Range From Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators." In: *IEEE Symposium on VLSI Circuits (VLSI)*. 2019, pp. C34–C35.

[90] Pasquale Davide Schiavone, Davide Rossi, Alfio Di Mauro, Frank K Gürkaynak, Timothy Saxe, Mao Wang, Ket Chong Yap, and Luca Benini. "Arnold: An eFPGA-Augmented RISC-V SoC for Flexible and Low-Power IoT End Nodes." In: *IEEE Transactions on VLSI Systems (TVLSI)*. Vol. 29. 4. 2021, pp. 677–690.

[91] Mark A. Anders, Himanshu Kaul, Sanu K. Mathew, Vikram B. Suresh, Sudhir K. Satpathy, Amit Agarwal, Steven K. Hsu, and Ram Kumar Krishnamurthy. "2.9TOP-S/W Reconfigurable Dense/Sparse Matrix-Multiply Accelerator With Unified INT8 / INTI6 / FP16 Datapath in 14NM Tri-Gate CMOS." In: *IEEE Symposium on VLSI Circuits (VLSI)*. 2018, pp. 39–40.

[92] Sander Smets, Toon Goedemé, Anurag Mittal, and Marian Verhelst. "A 978GOP-S/W Flexible Streaming Processor for Real-Time Image Processing Applications in 22nm FDSOI." In: *IEEE International Solid-State Circuits Conference (ISSCC)*. 2019, pp. 44–46.

[93] Anthony Mark Jones and Mike Butts. "TeraOPS Hardware: A New Massively-Parallel MIMD Computing Fabric IC." In: *IEEE Hot Chips Symposium (HCS)*. 2006.

[94] 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." In: *IEEE Journal of Solid-State Circuits (JSSC)*. Vol. 52. 4. 2017, pp. 891–902.

[95] Joao P. Cerqueira, Thomas J. Repetti, Yu Pu, Shivam Priyadarshi, Martha A. Kim, and Mingoo Seok. "Catena: A Near-Threshold, Sub-0.4-mW, 16-Core Programmable Spatial Array Accelerator for the Ultralow-Power Mobile and Embedded Internet of Things." In: *IEEE Journal of Solid-State Circuits (JSSC)*. Vol. 55. 8. 2020, pp. 2270–2284.

[96] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. "Hybrid Dataflow/von-Neumann Architectures." In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. Vol. 25. 6. 2014, pp. 1489–1509.

[97] Kurtis T. Johnson, Ali R Hurson, and Behrooz Shirazi. "General-Purpose Systolic Arrays." In: *IEEE Computer*. Vol. 26. 11. 1993, pp. 20–31.

[98] Keeley Criswell and Tosiron Adegbija. "A Survey of Phase Classification Techniques for Characterizing Variable Application Behavior." In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. Vol. 31. 1. 2019, pp. 224–236.

[99] Chad T. Effler, Brandon Kammerdiener, Michael R. Jantz, Saikat Sengupta, Prasad A. Kulkarni, Kshitij A. Doshi, and Terry Jones. "Evaluating the Effectiveness of Program Data Features for Guiding Memory Management." In: *ACM International Symposium on Memory Systems (MEMSYS)*. 2019, pp. 383–395.

[100] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. "Learning-based Memory Allocation for C++ Server Workloads." In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020, pp. 541–556.

[101] Ziqiang Huang, José A. Joao, Alejandro Rico, Andrew D. Hilton, and Benjamin C. Lee. "DynaSprint: Microarchitectural Sprints with Dynamic Utility and Thermal Management." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2019, pp. 426–439.

[102] Christophe Dubach, Timothy M. Jones, and Edwin V. Bonilla. "Dynamic Microarchitectural Adaptation Using Machine Learning." In: *ACM Transactions on Architecture and Code Optimization (TACO)*. Vol. 10. 4. 2013.

[103] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. "SparseAdapt: Runtime Control for Sparse Linear Algebra on a Reconfigurable Accelerator." In: *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 2021, pp. 1005–1021.

[104] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O'Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. "CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics." In: *ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2021.

[105] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "A Hardware–Software Blueprint for Flexible Deep Learning Specialization." In: *IEEE Micro*. Vol. 39. 5. 2019, pp. 8–16.

[106] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A Compiler Infrastructure for the End of Moore's Law." In: *arXiv*. 2020.

[107] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14.

[108] 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." In: *IEEE Journal of Solid-State Circuits (JSSC)*. 2022, pp. 1–1.

[109] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. “Polygeist: Raising C to Polyhedral MLIR.” In: *ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2021.

[110] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. “A Practical Automatic Polyhedral Parallelizer and Locality Optimizer.” In: *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2008, pp. 101–113.

[111] Louis-Noel Pouchet and Tomofumi Yuki. *PolyBench/C 4.2.1*. URL: https://polybench.sourceforge.net.

[112] Travis Downs. *A Concurrency Cost Hierarchy*. 2020.

[113] Cédric Bastoul. “Contributions to High-Level Program Optimization.” PhD thesis. Habilitation Thesis. Paris-Sud University, France, 2012.

[114] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. “A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs.” In: *ACM International Conference on Supercomputing (ICS)*. 2008, 225–234.

[115] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. “Polyhedral Parallel Code Generation for CUDA.” In: *ACM Transactions on Architecture and Code Optimization (TACO)*. Vol. 9. 4. 2013, pp. 1–23.

[116] H. T. Kung. “Algorithms for VLSI Processor Arrays.” In: *Introduction to VLSI Systems* (1980), pp. 271–292.

[117] Patrice Quinton. “Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations.” In: vol. 12. 3. 1984, pp. 208–214.

[118] Pei Zong Lee and Zvi M. Kedem. “Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays.” In: vol. 1. 1. 1990, pp. 64–76.

[119] Jason Cong and Jie Wang. “PolySA: Polyhedral-Based Systolic Array Auto-Compilation.” In: *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2018, pp. 1–8.

[120] 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: *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 181–189.

[121] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, and Pradeep Dubey. “SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs.” In: *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–9.

[122] Jie Wang, Licheng Guo, and Jason Cong. "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA." In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2021, pp. 93–104.

[123] Ruizhe Zhao and Jianyi Cheng. "Phism: Polyhedral High-Level Synthesis in MLIR." In: *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*. 2021. URL: https://arxiv.org/abs/2103.15103.

[124] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines." In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015, pp. 429–443.

[125] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. "Pencil: A Platform-Neutral Compute Intermediate Language for Accelerator Programming." In: *ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2015.

[126] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions." In: *arXiv*. 2018.

[127] Tim Zerrell and Jeremy Bruestle. "Stripe: Tensor Compilation via the Nested Polyhedral Model." In: *arXiv*. 2019.

[128] Sven Verdoolaege, Manjunath Kudlur, Rob Schreiber, and Harinath Kamepalli. "Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques." In: *International Workshop on Polyhedral Compilation Techniques (IMPACT)*. 2020.

[129] Uday Bondhugula. "High Performance Code Generation in MLIR: An Early Case Study with GEMM." In: *arXiv*. 2020.

[130] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations." In: *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 2021, pp. 1233–1248.

[131] Patricia Suriana, Andrew Adams, and Shoaib Kamil. "Parallel Associative Reductions in Halide." In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 281–291.

[132] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. "A Versatile Software Systolic Execution Model for GPU Memory-Bound Kernels." In: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 2019.

[133] Hongbo Rong, Xiaochen Hao, Yun Liang, Lidong Xu, Hong H Jiang, and Pradeep Dubey. "Systolic Computing on GPUs for Productive Performance." In: *arXiv*. 2020.

[134] Daniel Thuerck, Nicolas Weber, and Roberto Bifulco. "Flynn's Reconciliation: Automating the Register Cache Idiom for Cross-accelerator Programming." In: *ACM Transactions on Architecture and Code Optimization (TACO)*. Vol. 18. 3. 2021, pp. 1–26.

[135] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. "Fast Multiplication in Binary Fields on GPUs via Register Cache." In: *ACM International Conference on Supercomputing (ICS)*. 2016, pp. 1–12.