Towards Closing the Programmability-Efficiency Gap using Software-Defined Hardware

by

Subhankar Pal

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

নিজের মধ্যে লুকিয়ে থাকা তারার কাছে পৌঁছাবার চেষ্টা করো,
গভীর স্বপ্ন দেখো আর প্রতি স্বপ্নে লক্ষ্য কে আর একটু এগিয়ে রাখো।

– রবীন্দ্রনাথ ঠাকুর

Reach high, for stars lie hidden in you.
Dream deep, for every dream precedes the goal.

– Rabindranath Tagore

Subhankar Pal
subh@umich.edu
ORCID iD: 0000-0002-1564-7443

Dedicated to মা *(Mā),* বাবা *(Bābā) and my partner.*

ACKNOWLEDGMENTS

A good advisor ensures adequate support for the success of their student, whether it be tangible resources in the form of laboratory equipment, or additional time to finish a piece of work. A good advisor sets the right balance between allowing the freedom to explore the student's own research interests and providing necessary course-correction by setting expectations and technical help. A good advisor effectively shields the student from the uncertainties of research funding, and thus enables them to focus on research. I have been privileged to experience all of this, and more, under the guidance of my advisor, Professor Ronald Dreslinski. I thus thank Ron for his guidance in honing my research skills during the early stages of my studies and entrusting me with the independence to explore my ideas and lead research projects.

I also express my profound gratitude to Professors Chaitali Chakrabarti, Trevor Mudge, David Blaauw, Hun-Seok Kim, Christophe Dubach and Michael O'Boyle, who have been invaluable to my success during graduate studies. Chaitali taught me to work with extensive attention to detail, in addition to being of immense help in understanding concepts beyond my subject area. Trev has on several occasions stirred me to think about the "big picture" with my research, which has given me the sense of broader implications when I started to get tunnel vision. David and Hun-Seok provided immense hands-on guidance, without which I would not have been able to lead a chip tape-out. Christophe and Mike were invaluable in providing high-level suggestions and helping me rule out impractical ideas at an early stage.

I gratefully acknowledge the funding received towards my PhD from Rackham Graduate School and the Computer Science and Engineering (CSE) department at Michigan, as well as Defense Advanced Research Projects Agency (DARPA).

I have innumerable number of people outside of Michigan to credit for helping me succeed. I am grateful to Professors Chetan Kumar from BITS-Pilani, Hyderabad, India and Alberto García-Ortiz from University of Bremen, Germany, for mentoring me and motivating me to pursue graduate studies. I greatly appreciate the mentorship I received during internship stints, from John Kalamatianos and Adithya Yalavarti at AMD Research, and Swagath Venkataramani, Jintao Zhang and Viji Srinivasan at

IBM Research. They have been immensely patient with me as I navigated the waters of conducting research in domains beyond my primary research area.

I would like to thank the people who often went above and beyond their obligations to provide support with licensing and infrastructural issues, Reid Pinkham, Kyojin Choo, Hyochan An, Joel VanLaven, Donald Winsor, and Laura Fink. I am also grateful to Karen Liska, Ashley Andrea, Sylvia Galaty, Stephen Reger and Jamie Goldsmith from the CSE department for always being available to help with navigating the non-technical aspects of my PhD journey.

It goes without saying that one's fellow group mates are the best support system through their tenure as a PhD student and I am grateful to have received a ton of support from my lab-mates, Siying Feng, Tutu Ajayi, Nishil Talati, Javad Bagherzadeh, Heewoo Kim, Austin Rovinski, Jielun Tan, Sung Kim, and Yichen Yang. I additionally thank my collaborators and friends from Trev's group, Dong-hyeon Park and Jonathan Beaumont, whom I have had the pleasure to both learn from and share memorable moments with.

I am extremely grateful to have in my life some awesome people and very close friends, Subhayan Roy Moulick, Sarthak Ghosh, Jaideep Singh, Chakradhar Nemani, Sai Haritha Valluri, Sai Mahathi Padegal, Anirban Nag, and Tannishtha Som. A phone call or a text message from them would always lighten up my mood, especially during the stressful times in my PhD journey.

Finally, and most importantly, I dedicate everything to my family. My father, Bijay Pal, is my ultimate source of perseverance and strength. Everything I have ever achieved is because I had him on my side. I am grateful to him for countless things, from the time he taught me finger counting on the *left* hand so that I could count while writing, to the time he supported my decision to go out of state for my undergraduate education. Words cannot describe my gratitude to my partner in work and in life, my fiancé, Aporva Amarnath. She is a person who proactively spent time with selfless considerations just to see me happy, taught me how to keep calm in the face of anxiety, and kept me going through this marathon over the past five years. Last, but not the least, I am thankful to my never-ending source of love, who brought me up overcoming many hardships and yet could not live to experience pride with this milestone, my mother, Tapati Pal. Her eternal love and blessings from afterlife will always keep my spirit uplifted.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction-set Processor

AVX Advanced Vector eXtensions

BLAS Basic Linear Algebra Subsystems

CAM Content-Addressable Memory

CC Compressed Column

CCP Central Control Processor

CGRA Coarse-Grained Reconfigurable Array

CMOS Complimentary Metal-Oxide Semiconductor

COTS Commercial Off-The Shelf

CPU Central Processing Unit

CR Compressed Row

CSC Compressed Sparse Column

CSR Compressed Sparse Row

CU Control Unit

DDR Dual Data-Rate

DFG Data-Flow Graph

DLP Data-Level Parallelism

DMA Direct Memory Access

DMSpM	Dense Matrix - Sparse Matrix multiplication
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSL	Domain-Specific Language
DSP	Digital Signal Processing
DVFS	Dynamic Voltage Frequency Scaling
EDP	Energy-Delay Product
FCFS	First-Come, First Serve
FFT	Fast Fourier Transform
FIFO	First-In, First Out
FLOPS	Floating-point operations
FPGA	Field Programmable Gate Array
FP	Floating-Point
FU	Functional Unit
FSB	Front-Side Bus
GPE	General-purpose Processing Element
GPGPU	General-Purpose Graphics Processing Unit
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
GeMM	General (dense) Matrix - Matrix multiplication
GeMV	General (dense) Matrix - Vector multiplication
HBM	High-Bandwidth Memory
HLL	High-Level Language
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LCP	Local Control Processor

LFSR Linear Feedback Shift Register

LRG Least-Recently Granted

LRU Least-Recently Used

M-GeMM Masked General Matrix - Matrix multiplication

ML Machine Learning

MAC Multiply-And-Accumulate

MIMD Multiple Instruction, Multiple Data

MKL Math Kernel Library

MLP Memory-Level Parallelism

MSHR Miss Status Holding Register

NNZ Number of Non-Zeros

NRE Non-Recurring Engineering

NZE Non-Zero Element

OoO Out-of-Order

PE Processing Element

PPM Partial Product Matrix

RTL Register Transfer Level

SDH Software-Defined Hardware

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SPMD Single Program, Multiple Data

SPM Scratch-Pad Memory

SRAM Static Random Access Memory

SSN Swizzle-Switch Network

SoC System on-Chip

SpMM Sparse Matrix-Matrix multiplication

SpMV Sparse Matrix-Vector multiplication

VLIW Very Long Instruction Word

XCU Crosspoint Control Unit

ABSTRACT

The past decade has seen the breakdown of two important trends in the computing industry: Moore's law, an observation that the number of transistors in a chip roughly doubles every eighteen months, and Dennard scaling, that enabled the use of these transistors within a constant power budget. This has caused a surge in domain-specific accelerators, i.e. specialized hardware that deliver significantly better energy efficiency than general-purpose processors, such as CPUs. While the performance and efficiency of such accelerators are highly desirable, the fast pace of algorithmic innovation and non-recurring engineering costs have deterred their widespread use, since they are only programmable across a narrow set of applications. This has engendered a programmability-efficiency gap across contemporary platforms.

A practical solution that can close this gap is thus lucrative and is likely to engender broad impact in both academic research and the industry. This dissertation proposes such a solution with a reconfigurable Software-Defined Hardware (SDH) system that morphs parts of the hardware on-the-fly to tailor to the requirements of each application phase. This system is designed to deliver near-accelerator-level efficiency across a broad set of applications, while retaining CPU-like programmability.

The dissertation first presents a fixed-function solution to accelerate sparse matrix multiplication, which forms the basis of many applications in graph analytics and scientific computing. The solution consists of a tiled hardware architecture, co-designed with the outer product algorithm for Sparse Matrix-Matrix multiplication (SpMM), that uses on-chip memory reconfiguration to accelerate each phase of the algorithm. A proof-of-concept is then presented in the form of a prototyped 40 nm Complimentary Metal-Oxide Semiconductor (CMOS) chip that demonstrates energy efficiency and performance per die area improvements of 12.6× and 17.1× over a high-end CPU, and serves as a stepping stone towards a full SDH system.

The next piece of the dissertation enhances the proposed hardware with reconfigurability of the dataflow and resource sharing modes, in order to extend acceleration support to a set of common parallelizable workloads. This reconfigurability lends the system the ability to cater to discrete data access and compute patterns, such as workloads with extensive data sharing and reuse, workloads with limited reuse and streaming access patterns, among others. Moreover, this system incorporates commercial cores and a prototyped software stack for CPU-level programmability. The proposed system is evaluated on a diverse set of compute-bound and memory-bound kernels that compose applications in the domains of graph analytics, machine learning, image and language processing. The evaluation shows average performance and energy-efficiency gains of 5.0× and 18.4× over the CPU.

The final part of the dissertation proposes a runtime control framework that uses low-cost monitoring of hardware performance counters to predict the next best configuration and reconfigure the hardware, upon detecting a change in phase or nature of data within the application. In comparison to prior work, this contribution targets multicore Coarse-Grained Reconfigurable Arrays (CGRAs), uses low-overhead decision tree based predictive models, and incorporates reconfiguration cost-awareness into its policies. Compared to the best-average static (non-reconfiguring) configuration, the dynamically reconfigurable system achieves a 1.6× improvement in performance-per-Watt in the Energy-Efficient mode of operation, or the same performance with 23% lower energy in the Power-Performance mode, for SpMM across a suite of real-world inputs. The proposed reconfiguration mechanism itself outperforms the state-of-the-art approach for dynamic runtime control by up to 2.9× in terms of energy-efficiency.

CHAPTER I

Introduction

Moore's observation that the number of transistors on an integrated circuit chip roughly doubles every eighteen months has continued to hold for more than four decades [141]. However, this trend has visibly slowed down as we approach transistors sizes consisting of channels that are just a few atoms thick. The inevitable demise of Moore scaling, coupled with the end of Dennard scaling, a rule that states that the power density of a chip remains constant as the feature size scales down [60], has spawned the rise of heterogeneous computing systems. Typical heterogeneous systems consist of a multi-core CPU paired with a GPU and/or other fixed-function accelerators. CGRAs and Field Programmable Gate Arrays (FPGAs) have also gained traction as programmable hardware solutions, designed for near-Application-Specific Integrated Circuit (ASIC) performance and rapid hardware prototyping. The primary distinction between these devices is defined by the trade-offs between the following competing requirements:

- **Performance and Efficiency**. The performance of a system is generally measured either by the time taken to complete an operation (latency) or the number of useful operations executed in a given time (throughput). Throughput is commonly reported in FLOPS per second. Efficiency is measured as the performance delivered per Watt of power consumed by the system (energy efficiency), or per unit area of the chip (area efficiency).

- **Programmability and Flexibility**. Programmability of a system is a qualitative metric that captures the ease of writing efficient applications on the system. Flexibility is a measure of how versatile the system is, e.g. how many different applications can be efficiently executed on the system.

Multiple prior systems have been developed that deliver high performance and efficiency while retaining some programmability, but typically these are built for specific application domains that expose the hardware to similar compute and data access patterns. For instance, Google's TPU [96] and IBM's RaPiD [205] are efficient hardware engines for Deep Neural Network (DNN) acceleration that can be programmed to execute a multitude of neural network architectures. The challenge arises when the goal involves the acceleration of a broad set of application domains, such as scientific computing, graph analytics, vision and speech processing, etc., that exhibit varying compute and data characteristics.

This dissertation proposes a system that delivers energy efficiency nearing those of ASICs for a given application, while retaining full programmability to support a wide range of applications. Such a system is termed as SDH in this work. The target applications for this system are those that consist of a sequence of kernels that exhibit differing characteristics, e.g. a streaming Fast Fourier Transform (FFT) application followed by SpMV of the output stream with a sparse weight matrix. Clearly, a naïve approach to achieve maximum performance for such multi-kernel applications is to design a System on-Chip (SoC) that is comprised of multiple accelerators, one for each underlying kernel. However, there are two drawbacks to this approach.

- First, this system would be tied to a particular algorithm, and thus would be incompatible with improved algorithms engineered in the future.
- Second, the hardware would be area (and thus cost) inefficient due to under-utilization while executing dependent kernels, i.e. where the new kernel uses data produced from the previous kernel.

In contrast, the proposed SDH system accelerates the different kernels in each application by reconfiguring the same underlying hardware substrate to a configuration tailored to the characteristics of the kernel. The reconfiguration is of the on-chip memory type, resource sharing mode, and the dataflow, guided by a compile-time routine for inter-kernel reconfiguration and a runtime framework for intra-kernel reconfiguration. The runtime system snoops low-cost performance counters built into the SDH hardware in order to detect changes in code phases and/or nature of data (e.g. sparse or dense). This tight feedback loop between the hardware and runtime system allows for fast reconfiguration at a tight granularity, such as hundreds or thousands of executed FLOPS. Furthermore, in order to ensure ease of programming and flexibility across multiple applications, the proposed system has an integrated software stack. High-Level Language (HLL) intrinsics at the lowest level are aimed at expert programmers and library developers who are responsible for writing efficient code to run on the system. The user-facing layer is much more abstracted, with methods similar to contemporary data science libraries on HLLs, e.g. NumPy and SciPy libraries used with Python.

The rest of the dissertation is organized as follows. Chapter II explores contemporary architectural paradigms and motivates the need for and utility of an SDH architecture based on studies concerning implications of kernel characteristics on different hardware. Chapter III presents the architecture of a sparse matrix multiplication accelerator called OuterSPACE that uses memory reconfiguration to accelerate an outer product based algorithm. It then delves into the design of a proof-of-concept of this architecture in the form of a prototyped 40 nm CMOS chip. Chapter IV presents details of the Transmuter architecture, which enhances OuterSPACE to support both memory and dataflow reconfiguration along with full programmability and evaluates it on a set of diverse kernels. Chapter V describes a runtime control framework called SparseAdapt that uses offline training to learn the mapping of program behavior to the best hardware configuration and applies it during execution time to enhance the performance and/or energy-efficiency of operation of a workload on Transmuter. Chapter VI summarizes and concludes the dissertation.

This dissertation presents work done as part of a large research program that involved contributions from colleagues across multiple institutions. For clarity, my individual contributions are highlighted below.

- **OuterSPACE (Chapter III).** My contributions on the architecture front were in conceptualizing the hardware design with my collaborators. I developed the trace-based simulation infrastructure, the CPU implementation of the outer product algorithm, and the multiply phase implementation for OuterSPACE. On the chip tape-out front, I led the frontend development from architecture to Register Transfer Level (RTL). I designed the RTL for the multiply phase compute substrate and the on-chip memory subsystem, in addition to the top-level integration, and was responsible for parts of the pre-silicon verification and chip testing process.

- **Transmuter (Chapter IV).** I was responsible for devising parts of the hardware architecture, and for developing performance and functional models, including a cycle-level simulation model, a trace-driven model (published at IISWC 2020 [156]), and a functional simulator. I was also responsible for mapping and implementing the algorithms for a significant fraction of the evaluated kernel-configuration pairs.

- **SparseAdapt (Chapter V).** I conceptualized the idea and was responsible for implementing, training and tuning the ML models, as well as for developing the framework to simulate a dynamically reconfiguring system. I also implemented the non-standard performance counters and each of the SparseAdapt policies. Finally, I developed the infrastructure to automate training data collection and parallel simulation with hardware configuration sampling.

CHAPTER II

Kernel Characteristics and Reconfigurability

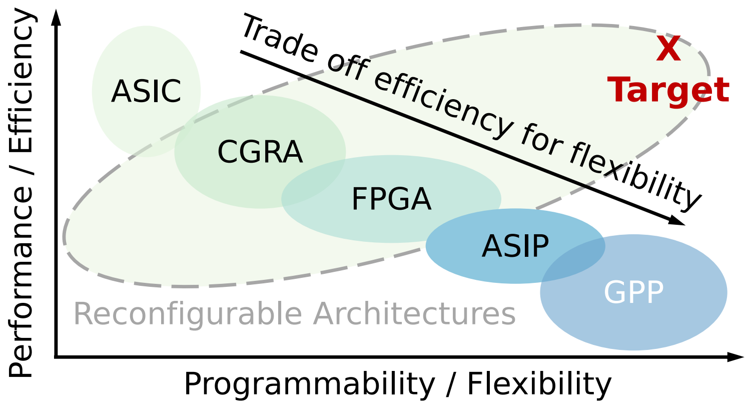
This chapter introduces contemporary computing platforms and discusses the advantages and drawbacks associated with each design. It then provides findings from a characterization study of the predominant kernels across a set of real-world applications spanning the domains of graph analytics, scientific computing, image and text processing, etc. The chapter concludes with a set of fundamental hardware choices that motivate the design of a reconfigurable SDH system.

2.1 Contemporary Architectures

Figure 2.1 shows a summary of prior studies that show trade-offs between performance or energy efficiency, and flexibility or programmability, across these architectural paradigms. CGRAs, FPGAs and Application-Specific Instruction-set Processors (ASIPs) lie in the middle of the spectrum with General-Purpose Processors (GPPs) and ASICs appearing at opposite ends.

CPUs are optimized for serial, latency-critical kernels. They commonly exploit maximum Instruction-Level Parallelism (ILP) in programs by executing instructions Out-of-Order (OoO), reducing control overhead by predicting across branches (branch prediction) and executing multiple instructions of the same kind (superscalar execution). Modern CPUs also support Single Instruction, Multiple Data (SIMD) extensions to harness Data-Level Parallelism (DLP) to a relatively small extent. However, CPUs perform sub-par for massively data-parallel applications.

GPUs, in particular General-Purpose Graphics Processing Units (GPGPUs), are designed to harness maximize data/thread-level parallelism. However, their effectiveness remains limited to regular workloads, i.e. data-parallel workloads that exhibit low thread-divergence [154]. Tuning the GPU in order to accelerate irregular work-



Performance / Efficiency
Programmability / Flexibility
Trade off efficiency for flexibility
X
Target
ASIC
CGRA
FPGA
ASIP
GPP
Reconfigurable Architectures

Figure 2.1: Trade-offs between programmability and efficiency in prominent computer architectures [145] showing the scope of reconfigurable architectures.

loads, such as those that operate on sparse data, remains a hot topic of research today [150, 28, 49].

Research on ASIC-based hardware accelerators has boomed in recent years [183]. ASICs that are custom-designed for a particular application or algorithm generally offer superior efficiency that surpasses any other contemporary architecture running the same application. ASIC designs, however, suffer from a few drawbacks.

- **High Non-Recurring Engineering (NRE) Costs.** Figure 2.2, taken from a recent keynote by Olofsson [152], illustrates that while Moore's law has been giving us an exponentially increasing transistors within a given die area, the design and verification costs associated with chip fabrication has also been growing at an exponential rate. This is an artifact of the complexities (e.g. leakage and tunneling) that are associated with designing a reliable chip with transistor sizes on the order of atomic distances.

- **Long Time-to-Market.** The end-to-end process leading to an ASIC prototype spans design, tape-out, fabrication, testing, and production. This leads to a turnaround time of at least a few months [223]. Despite efforts on accelerating the design flow [47], by the time an ASIC is fabricated and tested, improved algorithms may appear which would necessitate a new ASIC design [32, 83].

- **Applicability to Few Kernels.** An ASIC is generally built to accelerate a single kernel, or at best, a narrow set. Complex real-world problems seldom consist of just one kernel and thus require multiple ASICs for end-to-end acceleration. The main challenge lies in accelerating multi-kernel mixed data based workloads that exhibit amenability to different architectures.

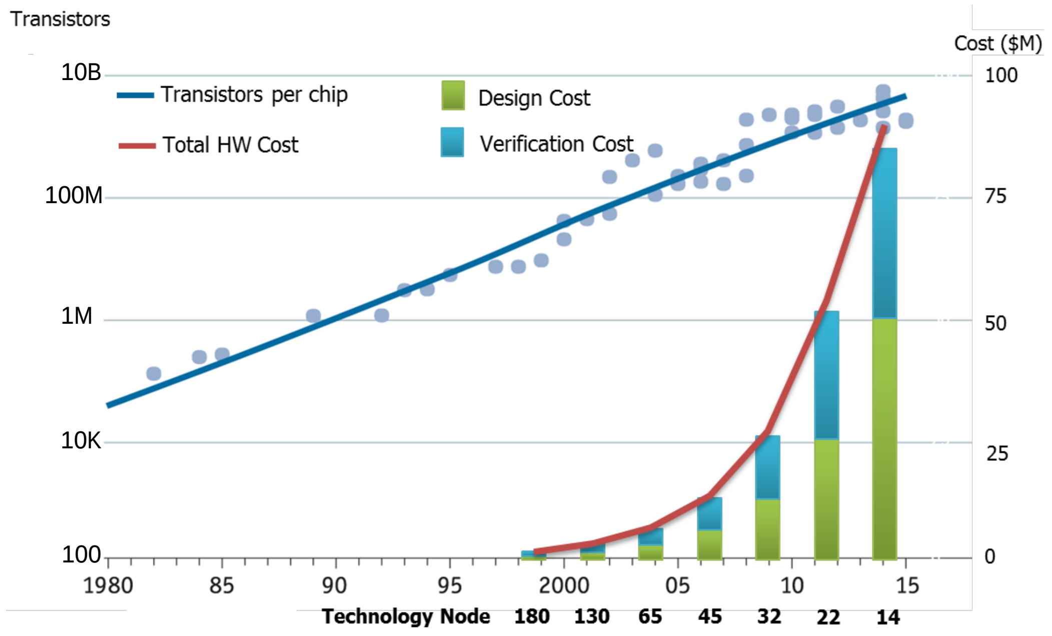
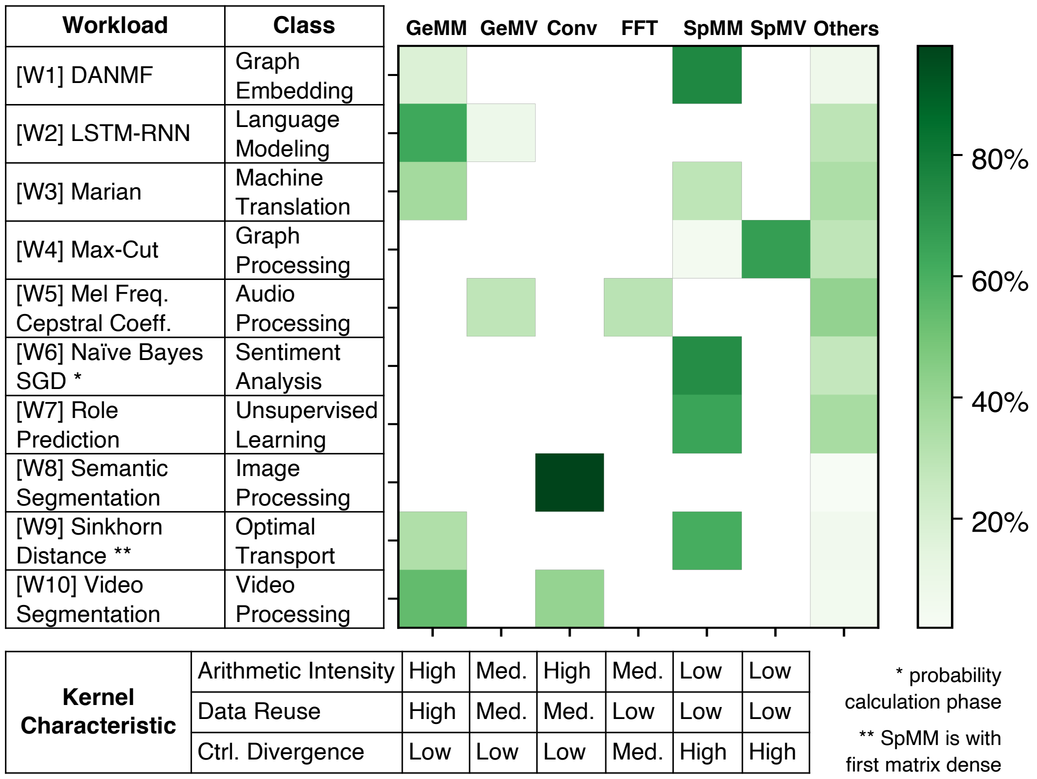


Figure 2.2: Scaling of number of transistors in a chip and design/verification costs between 1980 and 2015. Figure adopted from [152].

FPGAs have been successful for fast prototyping and deployment by eliminating non-recurring costs through programmable blocks and routing fabric. Moreover, high-level synthesis tools have reduced the low-level programmability challenges associated with deploying efficient FPGA-based designs [113, 112, 16]. Despite that, power and cost overheads prohibit FPGAs from adaptation in scenarios that demand the acceleration of a diverse set of kernels [171, 30, 169]. Besides, reconfiguration overheads of FPGAs are in the ms-µs range, even for partial reconfiguration [207, 159, 160], thus impeding fast run-time reconfiguration across kernel boundaries.

CGRAs overcome some of the energy and performance inefficiencies of FPGAs by reconfiguring at a coarser granularity. However, CGRA reconfiguration usually happens at compile-time, and the few that support run-time reconfiguration only support compute datapath reconfiguration [120], with overheads ranging from a few µs to 100s of ns [65, 68, 127]. Furthermore, many CGRAs require customized software stacks but have inadequate tool support, since they typically involve Domain-Specific Languages (DSLs) and custom Instruction Set Architectures (ISAs) [211].

While CPUs and GPUs carry significant energy and area overheads compared to lean ASIC designs, they are the *de facto* choice for programmers as they provide high flexibility and abstracted programming semantics such as OpenCL and CUDA [25].



Workload	Class
[W1] DANMF	Graph Embedding
[W2] LSTM-RNN	Language Modeling
[W3] Marian	Machine Translation
[W4] Max-Cut	Graph Processing
[W5] Mel Freq. Cepstral Coeff.	Audio Processing
[W6] Naïve Bayes SGD *	Sentiment Analysis
[W7] Role Prediction	Unsupervised Learning
[W8] Semantic Segmentation	Image Processing
[W9] Sinkhorn Distance **	Optimal Transport
[W10] Video Segmentation	Video Processing

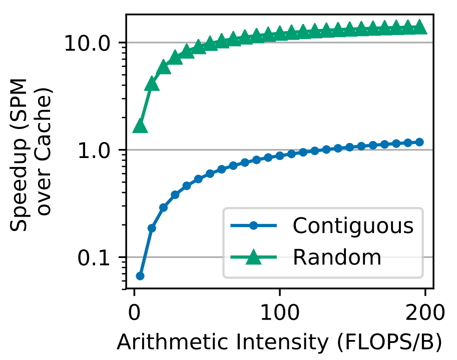
Kernel Characteristic		GeMM	GeMV	Conv	FFT	SpMM	SpMV
	Arithmetic Intensity	High	Med.	High	Med.	Low	Low
	Data Reuse	High	Med.	Med.	Low	Low	Low
	Ctrl. Divergence	Low	Low	Low	Med.	High	High

Figure 2.3: Fraction of execution time of kernels in applications spanning the domains of ML, signal processing, and graph analytics [217, 137, 97, 50, 131, 119, 51, 132, 42, 29] on a heterogeneous CPU-GPU platform. Some key characteristics, namely arithmetic intensity, data reuse and divergence, of each kernel are also listed.

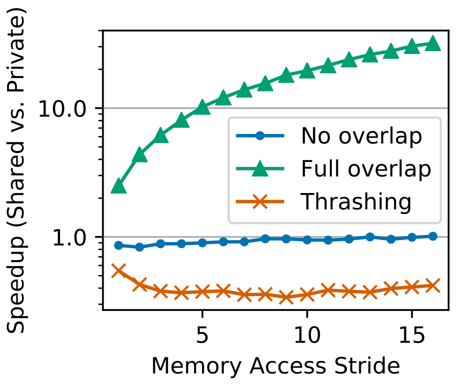
2.2 Hardware Support for Disparate Patterns

Many real-world workloads consist of multiple kernels that exhibit differing data access patterns and computational (arithmetic) intensities. In Figure 2.3, we show the percentage execution times of key kernels that compose a set of ten workloads in the domains of ML, graph analytics, and image and video processing. We characterize the kernels in terms of their arithmetic intensity (measured as the ratio of the amount of useful compute instructions performed to the amount of data fetched from off-chip memory), data reuse (amount of computation performed on a given piece of data that is fetched from off-chip memory), and control divergence (proportional to the number of divergent paths taken by multiple threads in a unit – threads within a warp in GPUs, and SIMD lanes in a CPU). The underlying kernels exhibit a wide range of arithmetic intensities, from $\frac{1}{1000}$ths to 100s FLOPS per byte, i.e. FLOPS/B (Figure 2.1), as well as significant variance in reuse and divergence.

On-Chip Memory Type: Cache vs. SPM. Cache and SPM are two well-



(a) Speedup of SPM over cache (1-core system) for contiguous and random memory access patterns with a sweep of arithmetic intensity.



(b) Speedup of shared over private caching (8-core system) for working sets with different overlap across cores. *Thrashing* is where all cores perform accesses mapping to the same bank in shared mode.

Figure 2.4: Performance comparison between different hardware on simple kernels exhibiting fundamentally different characteristics.

known and extensively researched types of on-chip memory [109, 17, 206]. To explore their trade-offs, we performed experiments on a single-core system that employs these memories. From our experiments in Figure 2.4(a), we observe the following.

- Workloads with low arithmetic intensity (i.e. are memory-intensive) but high spatial locality (contiguous accesses) perform better on a cache-based system.
- Workloads that are compute-intensive and have high traffic to disjoint memory locations favor an SPM if those addresses are known *a priori*. In this case, an SPM outperforms a cache because the software-managed SPM replacement policy supersedes any standard cache replacement policy.

Intuition thus dictates that the diverse characteristics of kernels would demand an equivalent diversity in hardware. We study the implications of some key hardware choices in the remainder of this section.

Thus, caching is useful for kernels that exhibit high spatial locality and low-to-moderate FLOPS/byte, whereas SPMs are more efficient when the data is prone to thrashing but is predictable and has sufficient reuse.

On-Chip Memory Sharing: Private vs. Shared. The performance of shared versus private on-chip resources is dependent on the size of the working-set and the degree of working-set overlap across cores, i.e. inter-core data reuse. We note from our studies in Figure 2.4(b) that:

- When there is significant overlap between the threads' working sets, sharing leads to speedups exceeding 10× over privatization. This is owed to memory access coalescing and deduplication of data in the shared mode.

- When cores work on disjoint data, i.e. chunks of data that are spread across a large memory address range, there is insignificant difference in performance with sharing over no-sharing, if the union of the threads' working sets fit on-chip.

- Regular kernels may exhibit strided accesses that can be hazardous for a shared multi-banked cache, due to conflicting accesses at the same bank. In this case, a private configuration delivers better performance.

Dataflow Type: Demand-Driven vs. Spatial. In this work, we refer to demand-driven dataflow as the dataflow used by GPPs, wherein cores use on-demand loads and stores to read and write data, and communicate via shared memory. In contrast, spatial dataflow architectures (e.g. systolic arrays) are data-parallel designs consisting of multiple Processing Elements (PEs) with direct PE-to-PE channels. Each PE receives data from its neighbor(s), performs an operation, and passes the result to its next neighbor(s) [111]. If pipelined correctly, this form of data orchestration harnesses the largest degree of parallelism. However, it is harder to map and write efficient software for certain applications on spatial architectures [93].

2.3 Key Takeaways and Path to Solution

The analysis presented in this chapter identifies the on-chip memory type, resource sharing and dataflow as three key hardware design choices that are each amenable to a different workload characteristic. This motivates the intuition that an architecture that reconfigures between these designs can accelerate diverse workloads that exhibit a spectrum of characteristics.

This dissertation first proposes a fixed-function accelerator called OuterSPACE that employs on-chip memory type reconfiguration to accelerate an increasingly important class of irregular workloads, namely sparse matrix multiplication. It then presents an enhanced version of this architecture called Transmuter that, in addition, incorporates resource sharing and dataflow reconfiguration along with programmable cores to accelerate kernels that compose broad application domains, such as the ones discussed in this chapter. The full SDH system proposal is culminated with the discussion of a software framework for dynamic reconfiguration.

CHAPTER III

Accelerating Sparse Matrix Multiplication using Memory Reconfiguration

Sparse matrices are widely used in graph and data analytics, machine learning, engineering and scientific applications. This chapter describes and analyzes OuterSPACE, an accelerator that uses memory reconfiguration coupled with an outer product based matrix multiplication algorithm to accelerate applications that involve large sparse matrices. OuterSPACE is a highly-scalable, energy-efficient, reconfigurable design, consisting of massively parallel Single Program, Multiple Data (SPMD)-style processing units, distributed memories, high-speed crossbars and HBM.

We identify redundant memory accesses to non-zeros as a key bottleneck in traditional sparse matrix-matrix multiplication algorithms. To ameliorate this, we implement an outer product based matrix multiplication technique that eliminates redundant accesses by decoupling multiplication from accumulation. We demonstrate that traditional architectures, due to limitations in their memory hierarchies and ability to harness parallelism in the algorithm, are unable to take advantage of this reduction without incurring significant overheads. OuterSPACE is designed to specifically overcome these challenges.

We simulate the key components of our architecture using gem5 on a diverse set of matrices from the University of Florida's SuiteSparse collection and the Stanford Network Analysis Project and show a mean speedup of 7.9× over Intel Math Kernel Library (MKL) on a Xeon CPU, 13.0× against cuSPARSE and 14.0× against CUSP when run on an NVIDIA K40 GPU, while achieving an average throughput of 2.9 GFLOPS/s within a 24 W power budget in an area of 87 mm^2.

We prototyped a scaled-down version of the OuterSPACE with 48 heterogeneous cores and a reconfigurable memory hierarchy in 40 nm CMOS technology. On-chip memories are reconfigured as SPMs or caches and interconnected with synthesizable coalescing crossbars for efficient memory access in each phase of the algorithm. The 2.0 mm×2.6 mm chip exhibits 12.6× (8.4×) energy efficiency gain, 11.7× (77.6×) off-chip bandwidth efficiency gain and 17.1× (36.9×) compute density gain against a high-end CPU (GPU) across a diverse set of synthetic and real-world power-law graph based sparse matrices.

The work presented in this chapter was published in the form of an architecture paper at HPCA 2018 [154], a circuits paper at VLSI 2019 [157] and an extended journal paper at JSSC 2020 [158].

3.1 Introduction

SpMM and SpMV are two key kernels of complex operations in domains such as graph analytics, machine learning, and scientific computation, as we elaborate in Section 3.2. The percentage of non-zero elements in the matrices involved can be very small. For example, the number of active daily Facebook users is currently 1.08 billion and the average number of friends per user is 338 [186]. A graph representing Facebook users as vertices and "friendships" between users as edges results in an adjacency matrix of dimension 1.08 billion with a density of just 0.0003%.

Sparse matrix-based computations are becoming an increasingly important problem. These applications are typically bottlenecked by memory rather than computation, primarily due to irregular, data-dependent memory accesses in existing matrix libraries, leading to poor throughput and performance [74, 135].

The demise of Moore's Law has led to renewed interest in accelerators to improve performance and reduce energy and cost. In order to address these issues for applications involving sparse matrix computations, we propose a custom accelerator, OuterSPACE, that consists of asynchronous SPMD-style processing units with dynamically-reconfigurable non-coherent caches and crossbars. OuterSPACE is designed to work with the unconventional outer product based matrix multiplication approach [26, 201], which involves multiplying the i^{th} column of the first matrix ($\mathbf{A}$) with the i^{th} row of the second matrix ($\mathbf{B}$), for all i. Each multiplication generates a partial product matrix and all the generated matrices are then accumulated element-wise to form the final result. A seemingly obvious drawback of this approach is the maintenance and storage of these partial product matrices. In the case of sparse matrices, however, this is much less of a concern and other considerations dominate. We identify contrasting data-sharing patterns in the two distinct, highly parallel compute phases: multiply and merge. The multiply phase involves data-sharing across parallel computation streams, while the merge phase involves strictly independent processing with little-to-no communication or synchronization between streams. This discrepancy leads to sub-optimal execution of the outer product method on mainstream architectures, namely GPUs and multi-core/many-core CPUs (Section 3.4.4).

The reconfigurability of OuterSPACE enables us to meet the contrasting computational needs of the outer product method's two compute phases. Employing asynchronous SPMD-style processing elements allows for control-divergent code to operate fully in parallel, as opposed to SIMD architectures, which need to serialize it at least partially. Software-controlled SPMs, coupled with hardware-controlled caches, prevent wasted data accesses to the main memory. Further, allowing non-coherence relieves pressure on the memory system associated with excess broadcasts and writebacks (which can contribute up to 30-70% of the total write traffic [121]), providing a fraction of the performance benefits.

While the main focus of our work is the acceleration of sparse matrix-matrix multiplication, we also present results of sparse matrix-vector multiplication and describe how element-wise operations can be performed on the OuterSPACE system.

We use a server-class multi-core CPU and GPU as baselines to compare our architecture against. For sparse matrix multiplication on the CPU, we use state-of-the-art sparse Basic Linear Algebra Subsystems (BLAS) functions from the Intel MKL. MKL provides math routines for applications that solve large computational problems and are extensively parallelized by OpenMP threading while using vector operations provided by the Advanced Vector eXtensions (AVX) instruction set. For the GPU, we compare our architecture against the cuSPARSE and CUSP libraries. cuSPARSE [41] applies row-by-row parallelism and uses a hash table to merge partial products for each row of the output matrix. CUSP [18, 44] presents fined-grained parallelism by accessing the input matrices row-by-row and storing the partial result with possible duplicates into an intermediate coordinate format. The intermediate structure is then sorted and compressed into the output matrix.

The rest of this chapter is organized as follows. Section 3.2 discusses a wide spectrum of applications that utilize sparse matrix operation kernels. Section 3.3 provides a brief background on sparse matrix multiplication algorithms and storage formats, in addition to prior work in this area. Section 3.4 discusses our outer product implementations for sparse matrix-matrix multiplication and evaluates their performance on the CPU and GPU. Section 3.5.1 provides details of the OuterSPACE architecture and how the outer product algorithm efficiently maps to it. Section 3.5.2 presents our experimental setup and Section 3.5.3 presents results and insights drawn from them.

Section 3.6.1 delves into detailed design of the prototype chip, with the mapping of the outer product algorithm on it described in Section 3.6.2. Section 3.6.3 contains the measurement setup used for evaluation and Section 3.6.4 reports the measured results along with comparison against baseline platforms. Section 3.7 concludes with a summary of our analyses and key takeaways.

As of this writing, our 40 nm OuterSPACE chip prototype is the first custom SpMM accelerator that addresses the off-chip memory access bottleneck for real-world sized matrices, evaluating densities $\geq$0.002% and dimensions $\leq$120k.

3.2 Motivation

Sparse matrices are ubiquitous in most modern applications that operate on big data. It is the general consensus that a selection of linear algebra routines optimized over years of research can be used to accelerate a wide range of graph and scientific algorithms [57].

SpMM, in particular, is a significant building block of multiple algorithms prevalent in graph analytics, such as breadth-first search [71, 72], matching [172], graph contraction [27], peer pressure clustering [184], cycle detection [219], Markov clustering [202], and triangle counting [15]. It is also a key kernel in many scientific-computing applications. For example, fast sparse matrix-matrix multiplication is a performance bottleneck in the hybrid linear solver applying the Schur complement method [213] and algebraic multigrid methods [18]. Other computing applications, such as color intersection searching [98], context-free grammar parsing [164], finite element simulations based on domain decomposition [79], molecular dynamics [89], and interior point methods [100] also rely heavily on sparse matrix-matrix multiplication.

SpMV is also predominant across diverse applications, such as PageRank [24], minimal spanning tree, single-source shortest path and vertex/edge-betweenness centrality calculations. It serves as a dominant compute primitive in ML algorithms such as support vector machine [147] and ML-based text analytics applications [139].

While GPUs demonstrate satisfactory compute efficiency on sparse matrix-vector multiplication and sufficiently dense matrix-matrix multiplication [19], we show that compute units are significantly underutilized when the density drops below 0.1%, often achieving fewer than 1 GFLOPS/s, despite a peak theoretical throughput of over 4 TFLOPS/s [199]. This is further supported by the fact that rankings, such as the Green Graph 500 list [181], are dominated by CPU-based systems.

For large dense matrices, inner product multiplication, block partitioning and tiling techniques are used to take advantage of data locality. However, when the density of the input matrices is decreased, the run-time is dominated by irregular memory accesses and index-matching in order to perform the inner product operations. Moreover, while tiling techniques can reduce redundant reads to main memory in the short-term, the on-chip storage constraints still necessitate that many data elements be redundantly fetched multiple times across tiles [94]. The stark inefficiencies on both the hardware and algorithmic fronts motivate our work to formulate a new approach for sparse matrix multiplication acceleration.

3.3 Background and Related Work

This section outlines a few fundamental concepts behind matrix-matrix multiplication and storage formats for representation of sparse matrices in memory.

3.3.1 Matrix Multiplication

3.3.1.1 The Inner Product Approach

Traditionally, General (dense) Matrix - Matrix multiplication (GeMM) is performed using the inner product approach. This is computed using a series of dot product operations between rows of the first matrix (**A**) and columns of the second matrix (**B**) and results in elements of the final product (**C**):

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j}$$

Here, N is the number of columns in **A** (or rows in **B**), while i and j are the row and column indices, respectively, of an element in the final matrix. Thus, each element of the final matrix is computed through a series of Multiply-And-Accumulate (MAC) operations. This is generally optimized using block partitioning and tiling techniques [209].

3.3.1.2 The Outer Product Approach

The outer product method [26, 201] multiples two matrices **A** and **B** by decomposing the operation into outer product multiplications of pairs of columns-of-**A** and rows-of-**B**, as illustrated in Figure 3.1. Mathematically,

$$\mathbf{C} = \sum_{i=0}^{N-1} \mathbf{C}_i = \sum_{i=0}^{N-1} \mathbf{a}_i \mathbf{b}_i$$

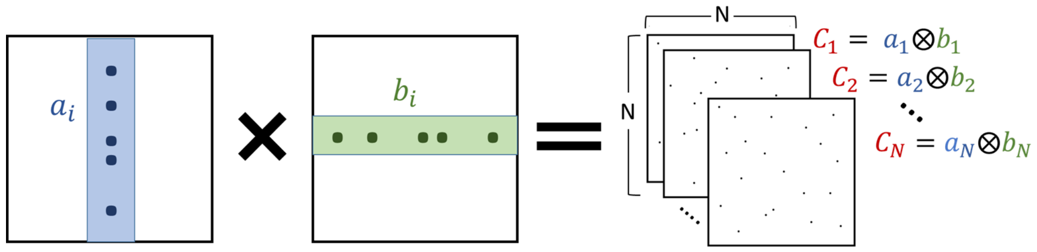


Figure 3.1: Outer product multiplication of matrices **A** and **B**. Each column-of-**A** and the corresponding row-of-**B** are multiplied with each other to produce N partial product matrices, $\mathbf{C_i}$. These are then summed together to produce the final result matrix **C**.

where $\mathbf{a}_i$ is the i^{th} column-of-**A**, $\mathbf{b}_i$ is the i^{th} row-of-**B** and $\mathbf{C}_i$ is a partial product matrix. Thus, the computation is divided into two sets: *multiply* operations to generate the partial products, followed by *merge* operations to accumulate the partial products into the final result. In Section 3.4, we propose an outer product based sparse matrix multiplication paradigm based on this.

3.3.2 Compressed Storage Formats

An $\mathbf{M}\times\mathbf{N}$ matrix is often represented in the dense format as a 2D array laid out in the memory as an $\mathbf{M}\times\mathbf{N}$ contiguous block. For sparse matrices, however, most of the elements are zeros, and hence, there is little merit in storing such matrices in the dense format.

The Compressed Sparse Row (CSR) format represents a matrix in a compressed manner using three arrays. The *vals* array consists of the non-zero elements of the matrix in row-major order, the *cols* array contains the column indices of the elements in *vals*, the *row-ptrs* array contains pointers to the start of each row of the matrix in the *cols* and *vals* arrays. The dual of the CSR format is the Compressed Sparse Column (CSC) format, which is comprised of the *vals*, *rows* and *col-ptrs* arrays.

In our implementation, while not being restrictive, we employ a similar storage scheme, consisting of a contiguous block of row pointers each pointing to contiguous arrays of column index-value pairs. We henceforth refer to this as the Compressed Row (CR) format. The complementary format, Compressed Column (CC) format, consists of column pointers pointing to arrays of row index-value pairs.

3.3.3 Related Work

With the prevalence of sparse matrix operation kernels in big data applications and their rising significance in a multitude of areas, there has been abundant work on accelerating sparse matrix-dense vector/matrix multiplication [81, 135, 2, 20]. However, there has been relatively less work done to accelerate sparse matrix-sparse vector/matrix multiplication and more on creating software frameworks for existing state-of-the-art architectures like multi-core and many-core CPUs [193, 179, 7], GPUs [45, 76, 135, 130] and heterogeneous (CPU-GPU) systems [129, 130, 135]. On the contrary, our work demonstrates an efficient co-design of outer product based sparse matrix multiplication with our custom, scalable, reconfigurable architecture, achieving significant speedups over state-of-the-art CPU and GPU libraries.

There has been some work on enhancing the underlying hardware for sparse matrix-matrix multiplication. Lin et al. [126] propose an FPGA-based architecture for sparse matrix-matrix multiplication that uses on-chip dedicated Digital Signal Processing (DSP) blocks and reconfigurable logic as PEs. However, the design is significantly limited by scarce on-chip FPGA resources, including the number of PEs and the size of the on-chip memory. Yavits and Ginosar [216] explore a juxtaposed Resistive Content-Addressable Memory (CAM) and RAM based sparse matrix-matrix and sparse matrix-vector accelerator, applying a row-by-row algorithm to efficiently match the indices of the multiplier and multiplicand and select the ReRAM row, where the corresponding non-zero element of the sparse multiplicand matrix/vector is stored. Zhu et al. [222] introduce a 3D-stacked logic-in-memory system by placing logic layers between Dynamic Random Access Memory (DRAM) dies to accelerate a 3D-DRAM system for sparse data access and build a custom CAM architecture to speed-up the index-alignment process of column-by-column matrix multiplication by taking advantage of its parallel matching characteristics. A prior fabricated design by Anders et al. [11] has demonstrated relatively high-density ($\geq$3%) matrix-matrix multiplication with small dimensions ($\leq$256) on a variable-precision systolic array and does not consider an off-chip memory interface.

However, the aforementioned hardware solutions accelerate SpMM algorithms with large amount of redundant memory accesses, which we identify to be a key performance bottleneck in sparse matrix-matrix multiplication. With our custom architecture, OuterSPACE, tailored for the outer product algorithm which eliminates most of these redundant accesses, we achieve significant speedups as well as high energy efficiency (Section 3.5.3). Moreover, the prior fabricated work does not address the off-chip memory bottleneck, which is an important source of inefficiency for a memory-bound problem such as SpMM.

There also exists work for accelerating sparse matrix-vector multiplication. Mishra et al. [139] add blocking to the baseline software and design fine-grained accelerators that augment each core in sparse matrix-vector multiplication. Nurvitadhi et al. [147] propose a SpMSpV algorithm using column-based SpMV, row blocking, column skipping, unique value compression (UVC), and bit-level packing of matrix data and a hardware accelerator for it, composed of a data management unit and PEs. Dorrance and Marković [53] propose an embedded SparseBLAS DSP processor that uses intelligent data reordering using a CSC-aware memory controller to accelerate SpMV. In our work, we address both SpMM and SpMV.

3.4 Outer Product Implementation

This section details the highly parallel outer product algorithm alluded to in Section 3.3.1.2, which maximizes memory reuse and avoids redundant reads to Non-Zero Elements (NZEs).

While the inner product approach (i.e. row-of-**A** $\times$ column-of-**B**) works efficiently for dense matrices, beyond a certain sparsity threshold, significant time is spent on matching indices of the two operands to find NZEs with the same row/column indices. This results in low Number of Non-Zeros (NNZ) per byte fetched from off-chip, leading to unproductive loads. Limited on-chip storage further forces repetitive fetching of the same data, worsening the memory bottleneck. The outer product technique circumvents these problems through:

- **Elimination of Index-Matching.** Each pair of non-zero elements from column-of-**A** and the corresponding row-of-**B** produce meaningful outputs. This is in contrast to inner product like algorithms, where the indices need to be matched before multiplication, leading to inefficient utilization of memory bandwidth to fetch elements redundantly.

- **Maximized NZE Reuse.** All elements in a row-of-**B** are shared for all elements in a column-of-**A** within an outer product. This maximizes the amount of reuse within a particular outer product calculation to its theoretical maximum, as we illustrate later in Figure 3.2.

- **Minimized Column and Row Loads.** As a result of maximized data reuse within an outer product calculation, we have no available data reuse across different outer products. Thus, once the computation between a column-of-**A** and the corresponding row-of-**B** is completed, they are never used again and can be evicted from local memory.

The number of loads and stores involved in generating one output NZE using the outer product method is listed in Table 3.1. Each output NZE requires as few as 3 loads and 2 stores from off-chip memory when there are no overlapping elements with the same indices during merge, which is the case for highly sparse matrices.

In the rest of the section, we present details about the two phases of outer product multiplication: *multiply* and *merge*. Since our algorithm requires that **A** be in CC format and **B** be in CR, we describe in Section 3.4.3 how we convert a matrix into its complementary format.

3.4.1 Multiply Phase

Figure 3.2 shows an example multiplication of two 4×4 sparse matrices, given three parallel processing units in the system. For clarity of understanding, the dense representations of matrices **A** and **B** are shown on the top-left of the figure. These matrices are decomposed into pairs of columns-of-**A** and rows-of-**B**. An outer product operation between each pair generates a full 4×4 compressed matrix and each of the generated matrices are summed together to produce the final result. In the CR mode of operation, each processing unit multiplies one non-zero element from a column-of-**A** with all the non-zeros in the corresponding row-of-**B**. The processing units are greedily scheduled in this example.

In our implementation, we store the intermediate partial products as a set of linked lists corresponding to each row (pointed to by R_i), where each node of the list contains a contiguous set of values representing a partial row of an outer product (Figure 3.2). The CC mode of operation is analogous to the CR mode, where we re-program the processing units to multiply an element of a row-of-**B** with all the non-zeros in the corresponding column of **A**. The row pointers, R_i are replaced by column pointers, C_i. This is illustrated in the bottom-right part of Figure 3.2.

3.4.2 Merge Phase

The outer products pointed to by a row/column pointer need to be merged to form the final result. We assign the processing units to walk through the linked list pointed to by R_i/C_i and merge them to form a complete final row/column. In the event that multiple data values from different outer products correspond to the same

Table 3.1: Memory access breakdown for the generation of one output NZE in the outer product approach.

Operation	**Cost**
Loading NZEs from each column of **A**	1 load per NZE
Loading NZEs from each row of **B**	1 load per NZE
Storing partial product result of one multiply	1 store per multiply
Loading an element of a partial product matrix	1 load per NZE
Storing an element of the result matrix	1 store per NZE result
Total	**3 loads + 2 stores per result NZE**

index, they must be summed together. However, this gets increasingly rare with sparser matrices.

In Section 3.5.1.4, we elaborate the merging scheme that maps efficiently to the architecture of OuterSPACE. The hardware can be programmed to produce the resultant matrix in either the CR or the CC format. For brevity, we assume CR mode operation in the rest of this chapter.

3.4.3 Matrix Format Conversion

When matrices **A** and **B** are not available in the CC and CR formats, respectively, either one or both will have to be converted to the complementary format. This is a one-time requirement for chained multiplication operations of the type $\mathbf{A}\times\mathbf{B}\times\mathbf{C}$..., since OuterSPACE can output the result in either CR or CC formats. However, computations such as $\mathbf{A}^{\mathrm{N}}$ can be decomposed into a logarithmic number of operations ($\mathbf{A}^2=\mathbf{A}\times\mathbf{A}$, $\mathbf{A}^4=\mathbf{A}^2\times\mathbf{A}^2$ and so on), where each operation would consist of conversion followed by actual computation. The requirement of conversion is obviated for *symmetric* matrices, since the CR and CC forms are equivalent.

In our evaluations, we assume that both the inputs, **A** and **B**, are available in the CR format, such that **A** must be converted. We partition the conversion operation into *conversion-load* and *conversion-merge* phases, analogous to the *multiply* and *merge* phases. The processing elements stream through **A** and store it into the intermediate data structure (Figure 3.2) in parallel. Conceptually, this is similar to multiplying Matrix **A** with an *Identity Matrix* of the same dimension:

$$\mathbf{I_{CC}}\times\mathbf{A_{CR}} \rightarrow \mathbf{A_{CC}} \text{ (CC mode)}$$

where, $\mathbf{I_{CC}}$ is the *Identity Matrix* and the subscripts represent the respective storage formats.

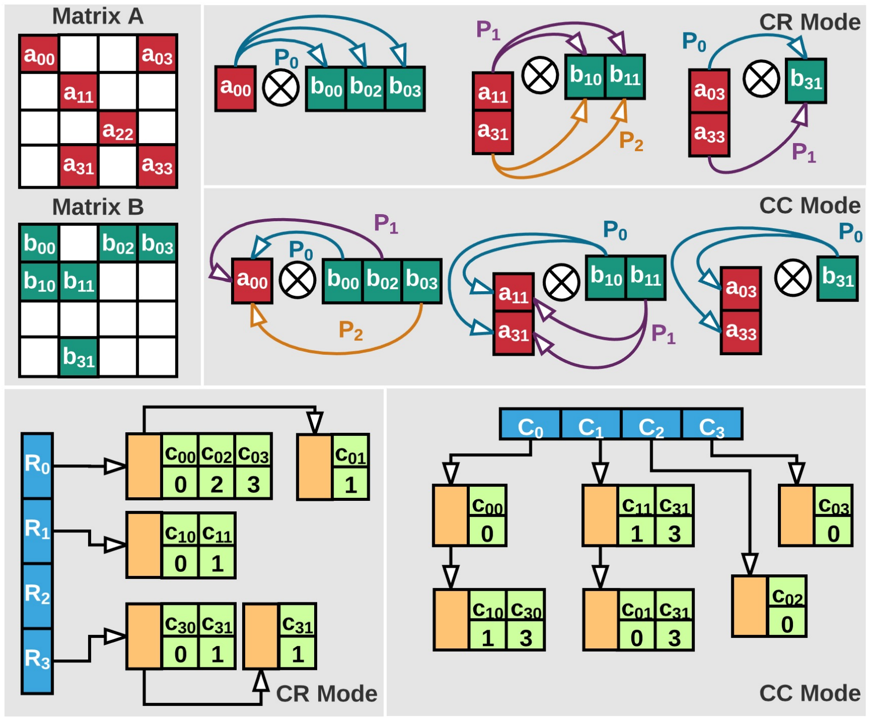


Figure 3.2: Outer product multiplication of matrices **A** (in CC) and **B** (in CR), illustrated in dense format, using three processing elements, and the layout of the partial products in memory. Both the CR and the CC modes of operation are shown here. Note that the third row of **B** is empty and hence no corresponding outer product is formed. The blue blocks represent the row/column pointers and the orange + green blocks are the partial product rows/columns containing index-value pairs.

3.4.4 Performance on Traditional Hardware

Outer product multiplication is a well-established linear algebra routine. Yet, it is not widely implemented in mainstream hardware, as these designs are not well-suited for such algorithms. We expose the inefficiencies of this paradigm on traditional hardware by quantitatively comparing our outer product implementations against state-of-the art libraries, due to the absence of readily available libraries based on outer product multiplication.

3.4.4.1 Multi-Core CPU (Intel Xeon)

Figure 3.3 compares the execution times of our CPU implementation of the outer product algorithm, using the POSIX threads library, against the Intel MKL SpMM library, on an Intel Xeon processor with 6 threads.

While the execution time of MKL drops exponentially with decreasing matrix density, the outer product algorithm must overcome two overheads with increasing

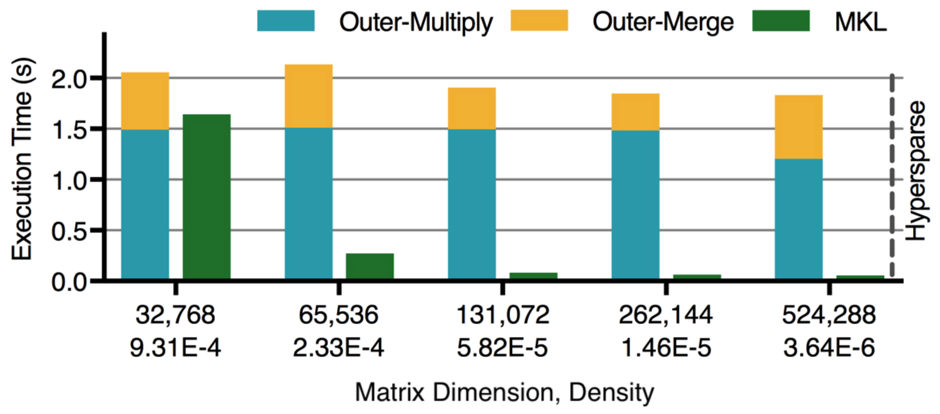


Figure 3.3: Comparison of our outer product implementation against Intel MKL on a Xeon multi-core CPU. The matrices are uniformly random with increasing dimension and decreasing density, keeping the number of non-zeros constant at 1 million. Format conversion and memory allocation times are not considered.

matrix dimension (N): the decreasing number of useful operations performed at each matrix datum and the increasing number of book-keeping operations due to the growing size of the data structure in Figure 3.2. Thus, the price of no index-matching and minimized redundant reads of non-zeros in the outer product technique is paid for by additional pressure on the memory system, as N $N \times N$ partial product matrices are streamed out during the *multiply* phase and back in during the *merge* phase.

This necessitates larger memory bandwidth and more cores to churn through the data streams than available on our 6-core CPU. It is further exacerbated by the absence of software-controlled SPMs and the ineffectiveness of CPU caching for the *merge* phase, which does not exhibit any data sharing within caches and thus leads to thrashing. This is substantiated by our studies of cache performance of the matrices in Figure 3.3, which show mean L2 hit rates of 0.14 and 0.12 during the *multiply* and *merge* phases, respectively. In comparison, MKL SpMM routines are vectorized and heavily optimized for the multi-core architecture.

Table 3.2 presents data generated using Intel VTune Amplifier for a Core i7 CPU running the MKL on the same matrices as in Figure 3.3. The under-utilization of bandwidth (average of 62%) suggests that bandwidth is not the primary bottleneck for the MKL and increasing it will likely provide only sub-linear speedups.

3.4.4.2 GPU (NVIDIA Tesla)

NVIDIA's implementations for matrix multiplication provided in their CUSP and cuSPARSE libraries perform very poorly at density levels below 0.01%. Running

Table 3.2: Bandwidth utilization of MKL sparse SpMM on an Intel Core i7 running 4 threads. Each matrix has a uniform random distribution of 10 million non-zeros.

Matrix Dimension	**Peak Bandwidth Utilization (%)**	**Avg. Bandwidth Utilization (%)**
1,048,576	62.5	44.2
2,097,152	67.5	58.4
4,194,304	67.5	62.0
8,388,608	85.0	62.4

synthetic workloads at this level of sparsity achieves fewer than 1 GFLOPS/s. L1 cache hit rate and utilized memory bandwidth drop to under 40% and 10 GB/s, from 86% and 30 GB/s at 10% density. We compare the performance of our custom outer product implementation, written in CUDA, against CUSP. Our implementation makes use of the GPU's available SPM storage. Figure 3.4 compares the execution times when run on an NVIDIA K40 GPU.

A comparison of results from Figure 3.3 and Figure 3.4 show that the GPU makes better use of available processing power and bandwidth than the CPU. The multiplication phase streams and processes the data much faster than the CPU implementation, scaling roughly linearly with decreasing density.

However, latency is quickly dominated by the merge phase. Despite both phases achieving similarly high L1 hit rates ($>\sim$80%) and low data dependency stalls ($<\sim$5%), the merge phase suffers from a much lower total throughput. This is a result of numerous conditional branches within the code to handle different relative column indices as they are read in and sorted. Because there is little correlation between adjacent threads as they process these branches, many threads within a given warp diverge and must be executed serially. Thus, while the high degree of parallelism available is attractive, the SIMD nature of the GPU's processing elements prevent an overall win of the algorithm over traditional libraries.

3.4.4.3 Many-Core CPU (Intel Xeon Phi)

Our experiments with the CPU outer product code on an Intel Xeon Phi Knights Corner system show an average slowdown of 14.7× compared to the CPU, for uniformly random matrices of dimensions varying from 32K to 524K with the number of non-zeros fixed at 1 million. We also note that denser matrices incur a significant amount of memory allocation overhead, which worsens the overall execution time. Although the outer product approach has high degrees of parallelism, it lacks an equivalent vectorizability. Moreover, Akbudak and Aykanat show in [7] that the

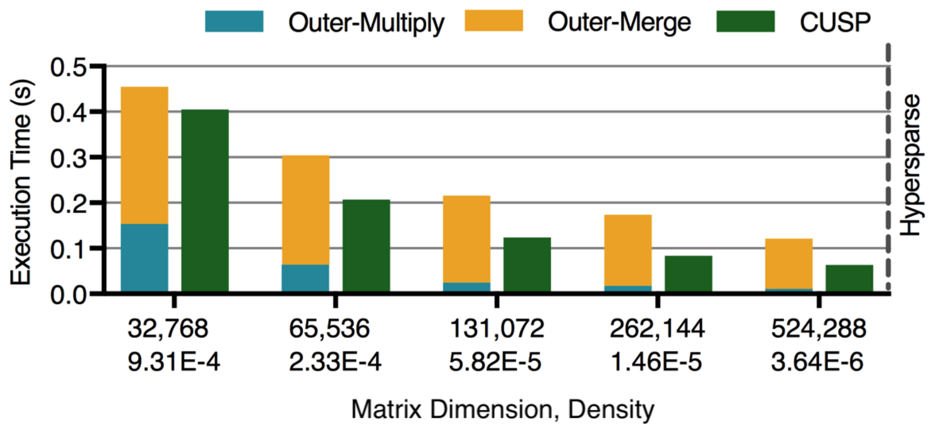


Figure 3.4: Comparison of a GPU outer product implementation against CUSP. The matrices are uniform random with increasing size while density is decreased, keeping the number of non-zeros constant at 1 million.

memory latency, rather than bandwidth, is the performance bottleneck for the many-core Xeon Phi system.

Intel MKL's SpMM function also shows 1.1× to 8.9× increase in execution time with respect to that on the Xeon CPU with decreasing density. The large caches of CPUs result in significantly better sparse matrix-matrix multiplication performance of CPUs as compared to the Xeon Phi, because repeatedly accessed rows in **B** may already be available in cache. The throughput-oriented Xeon Phi architecture has much smaller caches, resulting in many inefficient reloads of data from global memory. This trend is similar to what is observed in [174] for both the CPU and the Xeon Phi.

To combat the inefficiencies of existing architectures, we design a many-core architecture using an SPMD paradigm to exploit the massive parallelism inherent in the algorithm. We allow simple, dynamic resource allocation using asynchronous tiles and a non-coherent memory system. The SPMD paradigm allows computation to drift among the cores when work is imbalanced, leading to better compute utilization than the Single Instruction, Multiple Threads (SIMT) programming model in GPUs. Furthermore, to address the performance discrepancy between the multiply and merge phases due to different memory access patterns, we employ a reconfigurable cache hierarchy to allow data-sharing across multiple cores when needed, and segmenting storage to isolated units when it is not.

3.5 The OuterSPACE Architecture

This section of the chapter describes the OuterSPACE architecture and evaluates it on synthetic and real-world world sparse matrices using the gem5 [22] simulator.

3.5.1 Architectural Description

Qualitative analysis and results from the previous section reveal two key reasons why outer product multiplication does not perform well on conventional hardware:

- Outside of a particular outer product calculation during the *multiply* phase, there is no reuse of elements, as explained in Section 3.4. During this phase, the corresponding columns-of-**A** and rows-of-**B** can be shared across processing elements, whereas there is no data sharing between the processing units during the *merge* phase. Traditional hardware lacks the support to optimize for both of these phases, while dealing with variable memory allocation.

- While the *multiply* phase has consistent control flow between adjacent threads, dynamic execution paths in the *merge* phase necessitate fine-grained asynchrony across processing elements to fully utilize the available parallelism. An ideal architecture will allow for fully decoupled processing without sacrificing the ability to effectively share data.

To harness the massive parallelism and data reuse through fine-grained control over what data is fetched from memory, we propose our custom architecture, OuterSPACE. Figure 3.5 shows the microarchitecture of the OuterSPACE system. Our architecture is a system-on-chip consisting of SPMD-style parallel processing elements arranged as tiles, with two levels of shared, reconfigurable caches, and a set of control processors for scheduling and coordination, all connected to an HBM. In this work, we implement simple cache-to-SPM reconfiguration by switching-off tag arrays, although recent work such as coherent scratchpads [9] and Stash [109] have the potential to make OuterSPACE more general.

Following is a summary of the key elements of our proposed architecture:

- **Processing Element (PE).** A custom data-streaming and compute engine comprised of an ALU with a floating point unit, SPM, a control unit, a work queue, and an outstanding request queue (PEs are grouped into processing tiles).

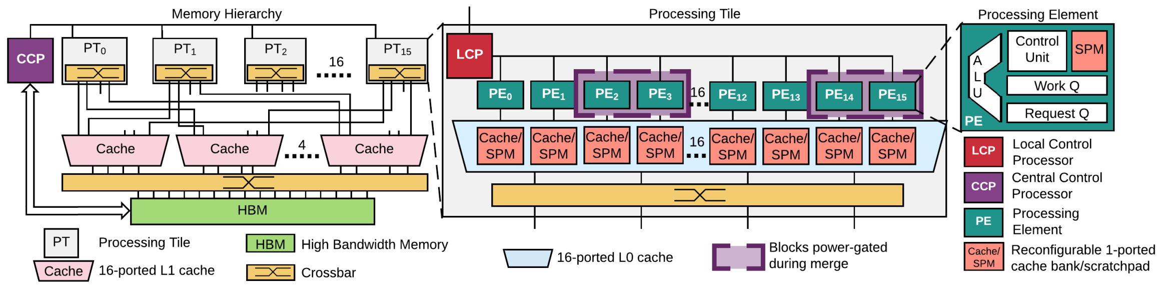


Figure 3.5: The memory hierarchy (left) and the architectures of the Processing Tile (center) and the Processing Element (right). The solid dark lines represent 64-bit bidirectional links.

- **Local Control Processor (LCP).** A small in-order core that coordinates the PEs within a tile, streaming instructions for the PEs to execute.
- **Central Control Processor (CCP).** A power-efficient core that is responsible for scheduling work and allocating memory for intermediate data structures.
- **High-speed crossbars** and **coalescing caches** that can be reconfigured into Scratch-Pad Memories (SPMs).
- **High-Bandwidth Memory (HBM).** that stores the input matrices and the intermediate partial products.

In the following sections, we present a detailed description of each element of the OuterSPACE system. We model 16 PEs per tile and 16 tiles based on scalability studies [182] to ensure that crossbar sizes do not bottleneck our architecture.

3.5.1.1 Processing Element

A block diagram of the PE is shown on the right side of Figure 3.5. At the core of the PE is a floating-point capable ALU for multiplication and summation of the matrix elements. These elements are streamed-in from the memory hierarchy (Section 3.5.1.3), orchestrated by the Control Unit, which generates loads and stores to memory. An outstanding request queue keeps track of the loads and stores that are in-flight. Lastly, there is a small private SPM and a FIFO work queue for decoded instructions and bookkeeping data, which are supplied to the PE by the LCP.

3.5.1.2 Processing Tile

A processing tile in our design consists of 16 PEs, an LCP, and a reconfigurable cache with 16 processor-side read/write ports and 4 memory-side ports. These caches internally consist of 16 single-ported cache banks and a controller (not shown) that interfaces with the LCP to reconfigure the cache into a SPM. As mentioned in Section 3.4, this is the key structure that reconfigures our system from a shared memory architecture into a non-shared one. The PEs within a tile communicate with the lower memory levels through a 4-ported crossbar.

3.5.1.3 Memory Hierarchy

Figure 3.5 shows 16 processing tiles that interface with the main memory through 4 L1 caches. These caches act like victim caches [95] and thus are smaller than their L0 counterparts, in order to minimize undesired eviction of data that is going to be reused. They cache-in elements that are evicted from the L0 caches when some PEs start drifting away in their execution flow from others within a tile, which occurs when the PEs are operating on multiple rows simultaneously. The L0 caches also contain a 16$\times$16 crossbar (not shown).

Our baseline main memory features an HBM 2.0 x64 interface [185], with 16 memory channels and a total memory bandwidth of 128 GB/s. With a PE clocked at 1.5 GHz, the total bandwidth for all the PEs is 9.2 TB/s (256 PEs $\times$ 1.5 giga-operations per second $\times$ 12 B per access for double-precision value and index pair $\times$ read + write channels). We overcome this bandwidth gap with a multi-level cache-crossbar hierarchy connecting the PEs to the memory. This hierarchy provides extensive reuse of temporally correlated data within an outer product.

The PEs in our system execute in an SPMD-fashion, often drifting apart from each other and only synchronizing at the end of the *multiply* and *merge* phases, as outlined in Section 3.4. This contrasts with the GPU, which traditionally employs a SIMT execution model, where compute units operate in lockstep over a dataset.

This also opens up avenues for various circuit-level techniques to improve energy efficiency, such as voltage throttling and dynamic bandwidth allocation. Furthermore, the PEs only share read-only data, which allow for the crossbars to be non-coherent structures without breaking correctness. Incorporating techniques such as SARC [101], VIPS [102] or DeNovo [37] would help expand OuterSPACE to algorithms demanding coherence, which we defer to a future work.

3.5.1.4 Mapping the Outer Product Algorithm

This section illustrates how the outer product algorithm described in Section 3.4 maps to OuterSPACE.

Multiply Phase. As mentioned in Section 3.4.1 and illustrated in Figure 3.2, processing units (PEs in OuterSPACE) multiply an element of a column-of-**A** with the entire corresponding row-of-**B**. The L0 caches retain the rows-of-**B** until all the PEs within a tile are finished with this set of multiplication. The PEs store the multiplied results in contiguous memory chunks, which are part of the linked list corresponding to a row-pointer ($\mathrm{R_i}$), using a write-no-allocate policy to avoid results evicting elements of **B**. The memory layout described in Figure 3.2, where chunks of memory in each node of the list are discontiguous, allows each PE to work independently without any synchronization. Thus, the only data that the PEs share throughout the *multiply* phase is read-only data (values and column-indices within rows-of-**B**).

Merge Phase. A subset of the PEs within each tile is assigned to merge all the partial products corresponding to a single row of the resultant final matrix at the start of the *merge* phase. The values to be merged are distributed across the partial products, as indicated by $\mathrm{R_i}$ in Figure 3.2.

For minimum computational complexity, a parallel merge-sort algorithm would be optimal for merging rows across partial products in $rN\log(rN)$ time, where rN is the total number of elements in the row. However, this will result in multiple re-fetches of the same data when the entire row cannot be contained within the upper memory hierarchy, which will dominate the execution time. Instead, we focus on minimizing memory traffic. Our algorithm operates as follows (assuming the number of rows to merge is rN, each of which contains rN elements, where r and N are the density and dimension of the matrix, respectively):

1. Fetch the head of each row and sort by column index into a linked list ($\mathcal{O}(r^2N^2)$ operations)

2. Store the smallest-indexed element from the list into the final location, load the next element from the corresponding row and sort it into the list ($\mathcal{O}(rN)$ operations)

3. Repeat 2 until all elements of each row have been sorted and shipped to memory (r^2N^2 iterations)

The overall complexity is $\mathcal{O}(r^3N^3)$. While less efficient algorithmically, number of elements stored in local memory is only on the order of rN. A local buffer of the next elements to sort can help hide the latency of inserting elements into the list under the latency of grabbing a new element from main memory. Given our target workloads and system specifications, the time to sort the values is expected to be on the order of one-tenth to one-millionth of the time for memory to supply the values, with sparser matrices having a smaller discrepancy.

Because this phase requires no data sharing across tiles (each set of rows that is being merged reads independent data), the shared cache can be reconfigured into private SPMs. This minimizes memory transactions by eliminating conflict cache misses within a processing tile and saves energy by eliminating tag bank lookups. If a SPM bank is too small to buffer the entire merge operation of each row, we recursively merge a subset of the rows into a single row until the number of rows is sufficiently small.

Figure 3.5 illustrates the reconfigurability of the tiles to handle the different phases of computation. Specifically:

- A batch of the PEs within a tile are disabled to throttle bandwidth to each PE and conserve power. Half of the PEs load the row buffers, while the remainder sort the incoming values and store them to the resultant matrix.

- A subset of the cache banks within a tile are reconfigured as private SPMs for the PEs (Figure 3.5). The reconfiguration can be achieved simply by power-gating the tag array of the cache. The address range of each bank is remapped by software, with assistance from the LCP. Thus, the reconfigured private cache-SPM structure maximizes Memory-Level Parallelism (MLP), while minimizing stalls due to computation. This technique has been employed on a smaller scale in GPUs [148].

3.5.1.5 Memory Management

Precise memory pre-allocation for the intermediate partial products is impossible, as the sizes of the outer products are dependent on the specific row/column sizes [130]. However, due to the predictable nature of the two phases, we can greatly reduce the overhead of dynamic memory allocation over general schemes.

For the multiply phase, we statically assign each partial product enough storage to handle the average case (the average number of non-zero elements can be quickly calculated from the compressed format before computation begins), as well as a large spillover stack to be used dynamically for larger products. As a statically assigned PE (one per row/column pair) begins computation of a given product, it can evaluate from the row-pointers exactly how much spillover space is needed. The PE sends a single atomic instruction to increment a global stack pointer by the appropriate amount, and writes the current value location visible to the other PEs. As long as the PEs do not consume the entirety of their static allocation before the atomic load returns, the latency of memory allocation can be entirely hidden.

In the merge phase, we perform a single memory allocation before computation by maintaining a set of counters for the size of each row in the multiply phase. Data is then streamed from two contiguous memory segments for each partial product row: the static partition from the multiply phase and, if used, a portion of the spillover space. The data is merged and streamed-out to the separately-allocated merge phase storage. As matrices get sparser, the amount of space wasted due to merge collisions in this space becomes negligibly small. We discuss further about allocation overheads in Section 3.5.3.3.

The memory footprint of the outer product approach can be represented as $(\alpha \cdot N + \beta \cdot N^2 \cdot r + \gamma \cdot N^3 \cdot r^2)$, where α, β and γ are small, implementation-dependent constants, for uniformly random sparse matrices with dimension N and density r. For non-uniform matrices, this metric is not easily quantifiable, as the sparsity patterns of the two matrices heavily influence the number of collisions between non-zeros.

3.5.1.6 Other Matrix Operations

We evaluate a sparse-matrix sparse-vector multiplication algorithm similar to our matrix-matrix implementation, with a few simplifications. In particular, the amount of work assigned to each PE is reduced and no SPM is needed in the merge phase, as partial products do not need to be sorted.

Element-wise matrix operations follow a similar procedure as the *merge* phase of the matrix-matrix multiplication algorithm described in Section 3.4.2. Given N matrices $\mathbf{A}_1$, $\mathbf{A}_2$, ..., $\mathbf{A}_N$ with the same dimensions, the data can be reorganized into a data structure similar to the one illustrated in Figure 3.2 and element-wise operations (+, -, $\times$, /, ==) can be performed on it.

There is close to a one-to-one correspondence between data operations in each of the typical element-wise matrix routines (addition, subtraction, multiplication, and comparison) and the *merge* phase of outer product sparse matrix-matrix multiplication. Thus, they are expected to have similar complexities and we do not evaluate them separately in this work.

3.5.2 Experimental Setup

To evaluate the performance of the outer product algorithm on OuterSPACE, we modeled the key elements, namely, the PEs, the cache-crossbar hierarchy and the

Table 3.3: Simulation parameters of OuterSPACE.

Processing Element	1.5 GHz clock, 64-entry outstanding requests queue, 1 kB SPM *Multiply* phase: All 16 PEs per tile active *Merge* phase: 8 PEs per tile active, rest disabled
L0 cache/ SPM	*Multiply* phase: 16 kB, 4-way set-associative, 16-ported, shared, non-coherent cache with 32 MSHRs and 64 B block size per tile *Merge* phase: 2 kB, 4-way set-associative, single-ported, private cache with 8 Miss Status Holding Registers (MSHRs) and 64 B block size + 2 kB SPM per active PE-pair
L1 cache	4 kB, 2-way set-associative, 16-ported, shared, non-coherent with 32 MSHRs and 64 B blocks
Crossbar	16×16 & 4×4 non-coherent, swizzle-switch based
Main Memory	HBM 2.0 with 16 64-bit pseudo-channels each @ 8000 MB/s with 80-150 ns average access latency

HBM, using the gem5 simulator [22]. We created two separate models pertaining to the two phases. We ignored the start-up time since it can be easily hidden by the latency of a few memory accesses, and scheduling delays. We also assumed that the PEs are greedily scheduled for the *multiply* phase. We modeled an outstanding request queue with 64 entries, for each PE, which is at par with the number of load-store units in modern GPUs [149] and CPUs [86], in order to hide latencies of memory accesses. The simulation parameters used are shown in Table 3.3.

We chose our parameters to optimize for performance and power efficiency. For the *merge* phase, we enabled only 8 of the 16 PEs per tile and reconfigure a proportional number of cache banks into private SPMs. We, in fact, observed that enabling a greater number of PEs results in slight performance degradation due to thrashing in the L1 cache. The SPM size was chosen to be large enough to hide the load latency during sort operations.

CACTI 6.5 [143] was used for modeling cache latency, area, and power values. For power dissipated by the core, we used static and dynamic power consumption values for an Arm Cortex-A5 with VFPv4 in 32 nm from [182]. We pessimistically used the same aggressive core model for the PEs in addition to the LCPs and CCP. Dynamic power consumption was calculated by capturing activity factors of the cache and cores from simulation. The HBM power was derived from the JEDEC specification document [185] and [8]. The parameters for modeling the crossbars were obtained from [182].

We built an instruction trace generator for the PEs and ran the generated traces through our gem5 model in order to process large matrices. Due to the practical limitations of this approach, we did not model the dynamic memory allocation overhead in OuterSPACE, and thus do not consider this overhead across any other platform in our evaluation. However, in Section 3.5.3.3, we provide an analysis of this overhead by quantifying the number of dynamic allocation requests using the allocation approach in Section 3.5.1.5.

3.5.3 Results and Evaluation

We evaluate the OuterSPACE architecture by comparing against state-of-the-art library packages on commercial systems, namely, Intel MKL (Version 2017 Initial Release) on the CPU, NVIDIA cuSPARSE (Version 8.0) and CUSP (Version 0.5.1) on the GPU. The specifications of these hardware are summarized in Table 3.4. We show the performance of OuterSPACE on two important classes of matrix operations, sparse matrix-matrix and sparse matrix-vector multiplication.

We report the simulation times obtained from our gem5 models for the *multiply* and *merge* models running instruction traces that exclude memory-allocation code. In order to provide fair basis for comparison against the CPU and the GPU, we discard memory allocation time and only consider the execution time of computation functions for the MKL, cuSPARSE and CUSP implementations (memory-allocation and computation are discrete functions for these libraries). While reporting throughput, we only consider operations associated with multiplication and accumulation in order to maintain consistency across algorithms and to avoid artificially inflating performance by accounting for additional bookkeeping. We s that the raw GFLOPS/s reported by our GPU performance counters on the Florida benchmark suite (Section 3.5.3.1) are similar to those reported by Liu and Vinter [130], but omit the results for brevity.

3.5.3.1 Sparse Matrix-Matrix Multiplication

Without loss of generality, we evaluate the performance of sparse matrix-matrix multiplication on our platform by multiplying a sparse matrix with itself ($\mathbf{C} = \mathbf{A}\times\mathbf{A}$; $\mathbf{A}$ in CR format to begin with, generating $\mathbf{C}$ in CR), in order to closely mimic the multiplication of two matrices of similar sizes/densities. We use two different sets of matrices, *synthetic* and *real-world*, as benchmarks to evaluate OuterSPACE. We account for format conversion overheads for non-symmetric matrices (Section 3.4.3) while reporting performance results for OuterSPACE, in order to model the worst-case scenario.

Synthetic Matrices. In Figure 3.6, we compare the performance-scaling of the

Table 3.4: Baseline CPU and GPU configurations.

CPU	3.6 GHz Intel Xeon E5-1650V4, 6 cores/12 threads 128 GB RAM, solid state drives
GPU	NVIDIA Tesla K40, 2880 CUDA cores @ 745 MHz, 12 GB GDDR5 at 288 GB/s

outer product algorithm on OuterSPACE against the CPU and GPU libraries, using a set of synthetic datasets obtained from the Graph500 R-MAT data generator [144]. The R-MAT parameters were set to their default values (A=0.57, B=C=0.19) used for Graph500 to generate undirected power-law graphs, which is also employed in recent work in graph analytics [178] [194]. We present results for medium-sized matrices corresponding to `nEdges` equal to 100,000 with `nVertices` swept between 5,000 and 80,000. In order to illustrate the impact of sparsity pattern on performance, we also provide comparisons against uniformly random matrices of same dimensions and densities.

OuterSPACE performs consistently well with respect to other platforms. It outperforms MKL and CUSP with a greater margin for the R-MATs than for the uniformly random matrices, with the execution time changing only slightly across matrix densities. cuSPARSE, on the other hand, performs better with increasing density. OuterSPACE exhibits slight performance degradation with increasing density for uniformly random matrices, but gets starker speedups over the GPU for power-law graphs. This data also substantiates that the outer product algorithm is much less sensitive to a change in size for a given number of non-zeros than the MKL, which correlates with our observations on the CPU (Figure 3.3).

Real-World Matrices. The real-world matrices we evaluate are derived from the University of Florida SuiteSparse Matrix Collection [48] and the Stanford Network Analysis Project (SNAP) [122], containing a wide spectrum of real-world sparse matrices from diverse domains such as structural engineering, computational fluid dynamics, model reduction, social networks, web graphs, citation networks, etc. These matrices have been widely used for performance evaluation in prior work in this area [129, 179, 76, 45]. We choose matrices from this collection that have both *regular* and *irregular* structures. Table 3.5 presents a summary of the structure and properties of these matrices, which have dimensions varying from 4,096 to 3,774,768 and densities ranging between 0.001% and 1.082%.

In Figure 3.7, we present the speedups of OuterSPACE over the MKL, cuSPARSE and CUSP. OuterSPACE steadily achieves speedups across all the matrices identified

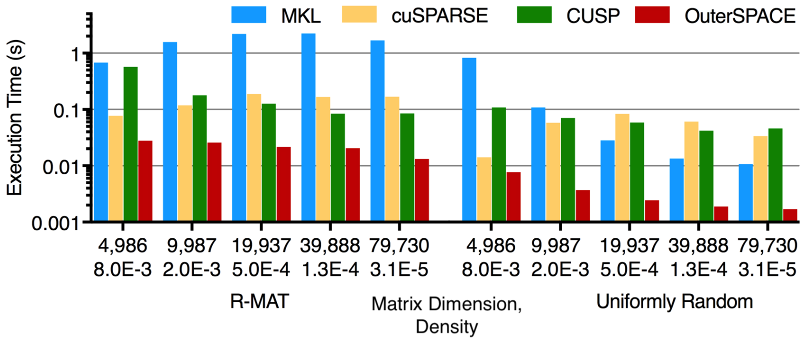


Figure 3.6: Performance-scaling comparison of OuterSPACE with change in matrix dimension and density. The set of data on the left is for R-MATs with parameters (A=0.57, B=C=0.19) for undirected graphs. The set on the right is for uniformly random matrices of the same size and density as the R-MATs.

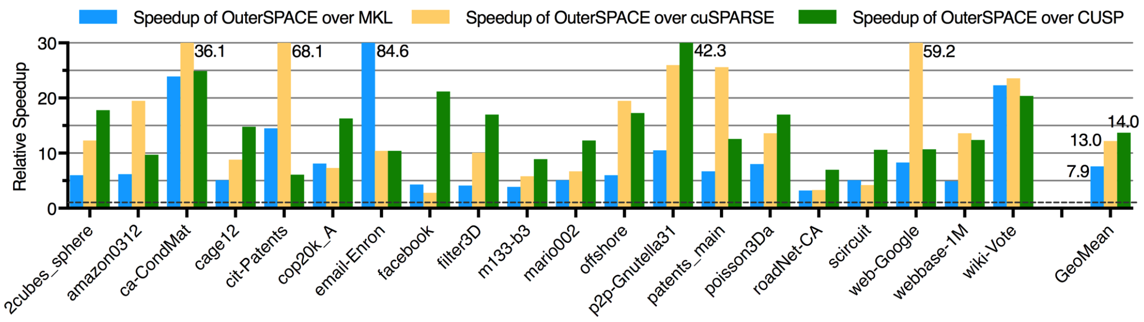
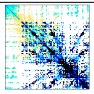
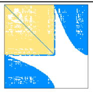
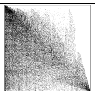
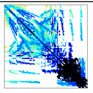
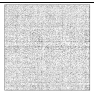
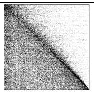
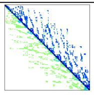
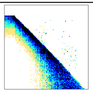
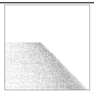
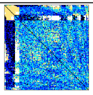
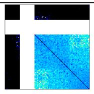
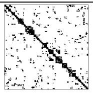
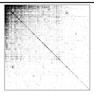
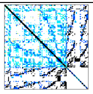
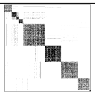
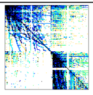
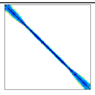
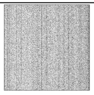
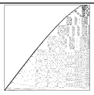
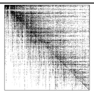


Figure 3.7: Speedups of OuterSPACE over the CPU running Intel MKL and the GPU running cuSPARSE and CUSP.

in Table 3.5, with an average of 7.9× over the Intel MKL, 13.0× over cuSPARSE and 14.0× over CUSP.

Some of the matrices that show relatively lesser speed-ups over the MKL and cuSPARSE are *filter3D* and *roadNet-CA*. These matrices are regular (*i.e.* have most of their non-zeros along their diagonals) and work better with the multiplication algorithms used by the libraries, because they incur fewer comparisons while multiplying two regular matrices. OuterSPACE performs only 3.9× better than MKL for the *m133-b3* matrix due to the uneven distribution of non-zeros along the columns, which leads to load imbalances during the *merge* phase and uneven data sharing patterns during the multiply phase. MKL performs particularly bad on *email-Enron*, a real-world email dataset with the characteristics of a power-law graph [33], substantiating the observation made in Section 3.5.3.1. OuterSPACE also achieves the highest speedups over cuSPARSE for matrices that have a more smeared (irregular) NZE distribution, such as *ca-CondMat*, *cit-Patents*, *p2p-Gnutella31* and *web-Google*

Table 3.5: Matrices from University of Florida SuiteSparse (SS) [48] and Stanford Network Analysis Project (SNAP) [122] with their plots, dimensions, number of non-zeros (nnz), average number of non-zeros per row/column (nnz_{av}) and problem domain.

Matrix	**Plot**	**Dim.** nnz nnz_{av}	**Kind**	**Matrix**	**Plot**	**Dim.** nnz nnz_{av}	**Kind**
2cubes_-sphere		101K 1.6M 16.2	EM problem	mario-002		390K 2.1M 5.4	2D/3D problem
amazon-0312		401K 3.2M 8.0	Co-purchase network	offshore		260K 4.2M 16.3	EM problem
ca-Cond-Mat		23K 187K 8.1	Condensed matter	p2p-Gnutella-31		63K 148K 2.4	p2p network
cage12		130K 2.0M 15.6	Directed weighted graph	patents_-main		241K 561K 2.3	Directed weighted graph
cit-Patents		3.8M 16.5M 4.4	Patent citation network	poisson-3Da		14K 353K 26.1	Fluid Dynamics
cop20k-_A		121K 2.6M 21.7	Accelerator design	roadNet-CA		2.0M 5.5M 2.8	Road network
email-Enron		36.7K 368K 10.0	Enron email network	scircuit		171K 959K 5.6	Circuit simulation
facebook		4K 176K 43.7	Friendship network	webbase-1M		1M 3.1M 3.1	Directed weighted graph
filter3D		106K 2.7M 25.4	Reduction problem	web-Google		916K 5.1M 5.6	Google web graph
m133-b3		200K 801K 4.0	Combi-natorial problem	wiki-Vote		8.3K 104K 12.5	Wikipedia network

(Table 3.5).

OuterSPACE running the outer product algorithm over this suite of matrices achieves an average throughput of 2.9 GFLOPS/s, accounting only for useful operations (multiplications and summations). We also observe a memory bandwidth utilization of 59.5-68.9% for the multiply phase and a slightly lower 46.5-64.8% for the merge phase. This is due to fewer active PEs and greater use of local memory during the merge phase. This can be further improved if the matrices are strategically laid out in memory such that there is fairer access to every memory channel, but this would require compiler support and is beyond the scope of our work.

3.5.3.2 Sparse Matrix-Vector Multiplication

In this section, we evaluate the performance of sparse matrix-vector multiplication on OuterSPACE. Table 3.6 shows the speedups of OuterSPACE over the CPU running MKL and GPU running cuSPARSE with vector densities ranging from 0.01 to 1.0 (fully dense). The performance of MKL is constant across different vector densities for a given matrix dimension and density, because MKL's sparse matrix-vector method performs the best when the vector is treated as a dense vector regardless of the number of zeros in the vector. cuSPARSE, however, scales with change in vector density.

Across all matrix sizes, the speedup of OuterSPACE scales linearly with vector density, with a 10× reduction in density resulting in approximately a 10× gain in speedup. Such performance scaling is possible because the outer product algorithm only accesses specific columns in the input matrix that match the indices of the non-zero elements of the vector. This eliminates all redundant accesses to the matrix that are present in a conventional inner product algorithm. Thus, the number of memory accesses to the sparse matrix is directly proportional to the number of non-zeros in the vector.

We identify two striking differences between the outer product algorithm on OuterSPACE and existing matrix-vector multiplication on CPUs and GPUs. First, unlike conventional algorithms where the performance depends heavily on the dimensions of the matrix regardless of the density, the performance of the outer product algorithm scales with the number of non-zero elements in the matrix, while remaining independent of the matrix dimension, for uniformly random matrices. Second, the performance of the sparse matrix-vector algorithm on OuterSPACE also scales linearly with the density of the vector, which allows OuterSPACE to outperform traditional algorithms for sparse vectors. Even while working on small, dense vectors, OuterSPACE achieves within 80% of the MKL's performance, as reflected in the fourth column of Table 3.6.

3.5.3.3 Dynamic Memory Allocation

As detailed in Section 3.5.1.5, the latency of dynamic allocation requests by the PEs can typically be hidden by sending the atomic request to increment the spill-over pointer before the PE begins multiplication. Increasing the amount of space statically assigned for partial products lowers the execution time by decreasing the number of accesses for dynamic allocation, at the expense of wasted storage, illustrating a performance-storage trade-off.

Table 3.6: Speedups of OuterSPACE over CPU (MKL) and GPU (cuSPARSE) for sparse matrix-vector multiplication. The vector density ($\boldsymbol{r}$) is varied from 0.01 to 1.0. The sparse matrices contain uniformly random distribution of one million non-zeros.

	Speedup over CPU			**Speedup over GPU**		
Matrix Dimension	$\boldsymbol{r}$= **0.01**	$\boldsymbol{r}$= **0.1**	$\boldsymbol{r}$= **1.0**	$\boldsymbol{r}$= **0.01**	$\boldsymbol{r}$= **0.1**	$\boldsymbol{r}$= **1.0**
65,536	93.2	8.7	0.8	92.5	11.2	3.8
131,072	107.5	9.9	1.0	98.2	11.2	2.8
262,144	152.1	12.6	1.2	126.0	12.5	2.3
524,287	196.3	17.2	1.7	154.4	17.4	2.2

We assume $\alpha \cdot \frac{nnz^2}{N}$ elements are allocated statically, where $\frac{nnz^2}{N}$ is the amount of storage needed for and average row and α is a parameter. Our analysis of the total number of dynamic requests to increment the spill-over pointer, while sweeping (α), shows that the count of these requests drops to less than 10,000 for $\alpha \geq 2$ for almost all the matrices in Table 3.5. *m133-b3* is an outlier, with zero dynamic requests, as it has exactly 4 non-zeros per row, which fits within the statically allocated space even for $\alpha = 1$. Our strategy works best for matrices that are more uniformly distributed, since a suitable value of α can eliminate most of the dynamic allocation requests. However, this overhead for real-world matrices is largely dependent on the sparsity patterns of the matrices.

3.5.3.4 Power and Area Analysis

Table 3.7 presents the area and power estimates for OuterSPACE in 32 nm. The total chip area, excluding the HBM controllers, is calculated to be 87 mm^2. The power consumption of the system is calculated to be 24 W, using the parameters presented in Section 3.5.2. This yields on average 0.12 GFLOPS/s/W (GFLOPS/J) for OuterSPACE.

For comparison, the mean measured power consumption of the K40 GPU while running the workloads was 85 W. With the GPU achieving only 0.067 GFLOPS/s on an average, this yields 0.8 MFLOPS/J. The OuterSPACE system is, thus, approximately 150× better than the GPU on the performance/power metric.

3.5.3.5 OuterSPACE Scaling

Our current architecture is still well below current reticle sizes and power limitations. In order to handle matrix sizes larger than a few million, a silicon-interposed system with 4 HBMs and 4× the PEs on-chip could be realized. This extended con-

Table 3.7: Power and area estimates for OuterSPACE (Figure 3.5).

Component	**Area (mm^2)**	**Power (W)**
All PEs, LCPs, CCP	49.14	7.98
All L0 caches/SPMs	34.40	0.82
All L1 caches	3.13	0.06
All crossbars	0.07	0.53
Main memory	N/A	14.60
Total	**86.74**	**23.99**

figuration would support matrices containing tens to hundreds of millions of non-zero elements, limited by the capacity of the HBM. In order to process larger matrices, we conceive equipping our architecture with node-to-node serializer-deserializer (SerDes) channels to allow multiple OuterSPACE nodes connected in a torus topology, thus minimizing system latency, and maximizing throughput. Such a system would be able to process matrices with billions of non-zeros. To scale to problems involving matrices with trillions of non-zeros, we envision interconnecting many such 16 OuterSPACE-node clusters.

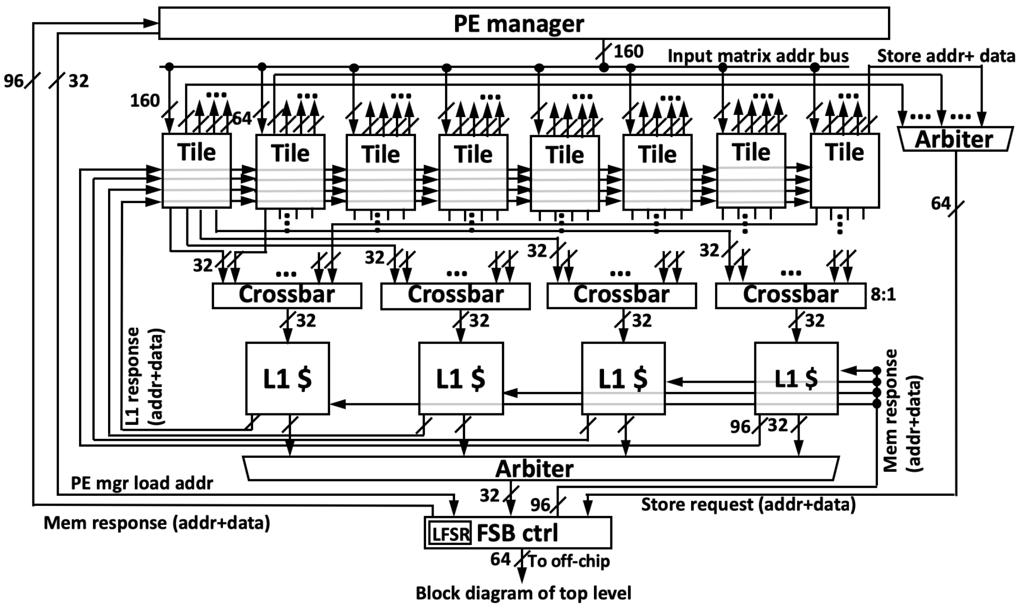
3.6 Chip Prototype

This section of the chapter details the circuit implementation and outer product mapping on a scaled-down OuterSPACE prototype that was fabricated in 40 nm CMOS. While the multiply phase compute fabric consist of dedicated state machines with Floating-Point (FP) multiplication units, the merge phase is computed using general-purpose Arm Cortex-M0 [39] and Cortex-M4 [40] cores. The off-chip interface used is a Front-Side Bus (FSB) that connects the chip to a Dual Data-Rate (DDR) memory on an external FPGA.

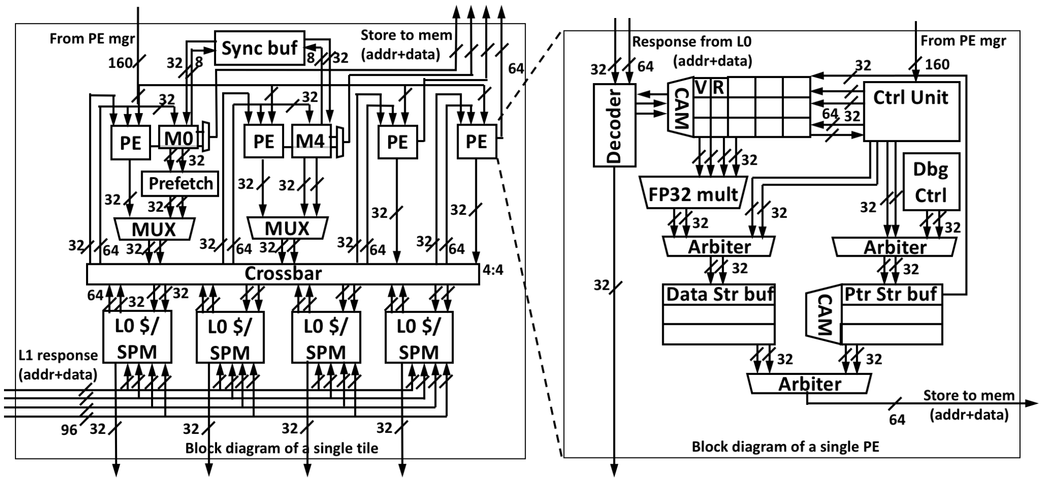
We focus on three key metrics while presenting results from the chip evaluation, namely number of output NNZs produced per second (throughput), NNZ/s per Watt (energy efficiency), and NNZ/s per GB/s (bandwidth efficiency).

3.6.1 Circuit Implementation

Our chip consists of two compute substrates, as shown in Figure 3.8. The first, composed of 32 PEs (4 PEs/tile), computes the *multiply* phase. Each PE has a 32-bit FP multiplier and supports out-of-order loads/stores. The second substrate consists of eight Arm Cortex M0+M4 pairs (1 pair/tile) for the *merge* phase.



(a) Top-level view of the chip, showing the second cache layer (L1) connecting the tiles to the FSB controller through crossbars. A central PE manager distributes work to all the PEs.



(b) Left. View of a tile showing the Arm cores and PEs connected to the first cache layer (L0) through a crossbar. Two PEs share ports with the cores. The store path bypasses the two cache layers. Right. View of a PE showing the control unit, FP multiplier, request queue, split store buffers and arbiters.

Figure 3.8: Top level diagram of the chip, a tile, and a PE. The chip contains a total of 8 tiles, with each tile consisting of 4 PEs and a pair of Cortex-M0 and M4 cores.

All the compute elements are connected through a reconfigurable network. The network consists of a fully-synthesizable Swizzle-Switch Network (SSN) crossbar based on [182], with the original pull-down networks replaced by OR trees (Figure 3.9). The synthesizable SSN still uses the same priority algorithm, but can also be easily ported to different process technologies since it does not require a custom layout. The crossbars support request coalescing, multicasting (Figure 3.11) and Least-Recently Granted (LRG) arbitration (Figure 3.10).

3.6.1.1 Compute Substrate

The PEs are custom Finite State Machine-based elements that perform the *multiply phase* of the outer product algorithm. At the core of the PE is a Control Unit (CU) that walks through the algorithm state machine. The CU initiates loads of elements of columns of Matrix A and rows of Matrix B, tracking requests in a *request queue*. The *request queue* is a structure that allows *out-of-order* loads to the elements of the input matrices. Load responses satisfy an entry in the queue by associatively searching the address field of each *request queue* entry. Each PE also houses a single-cycle, single-precision floating point multiplier that multiplies elements of A and B as soon as they are available in the *request queue*. The calculated partial product elements are stored into a “data” store buffer. This is a simple FIFO queue of (address, data, valid) tuples. There exists a separate buffer to store pointers, which is associatively-searchable, unlike the data buffer. Through this split store buffer design, we are able to reduce the energy consumed by limiting expensive associative searches to fewer registers. Finally, a debug block is used to relay important messages at programmable intervals to the off-chip interface, such as state of each PE, number of multiplications committed, etc.

The general purpose cores, Arm Cortex-M0 and Cortex-M4 cores, handle the computation in the *merge* phase. They are both low-power, in-order cores designed for high energy efficiency. The M4 performs the bulk of the computation including the floating-point operations. The M0 acts as a programmable prefetcher for loading data into the SPM independent of the M4’s operation.

The M0 and M4 cores communicate through the use of local SPM for shared data, and hardware mutex locks to streamline synchronization. The mutex locks come in two types: First-Come, First Serve (FCFS) mutex and sleep mutex. The FCFS mutex is a simple synchronization lock where the core that acquires the lock first prevents the other core from acquiring the lock, until the first one releases it. When querying the lock for acquisition, the cores have the option to stall until the lock is freed. The sleep mutex is a unidirectional lock with a predetermined owner. Sleep mutex begins with its lock pre-acquired by its designated core, and the non-designated core stalls whenever it accesses a locked mutex. During the *merge* phase, the sleep mutex is used by the M4 core to prevent M0 core from starting the prefetch before M4 has finished initiating the metadata.

3.6.1.2 Coalescing Crossbar

The crossbar takes one cycle to arbitrate, based on an LRG scheme, and another cycle to transmit data. As shown in Figure 3.10, each requester sends its priority bits to be bitwise OR'd. The corresponding bit of the result vector, based on the index of the requesters, is sent back to the requesters and the one with a 0 on its granted bitline wins. Next cycle, the winner clears its priority bits and other requesters set the priority bit corresponding to the winner to 1, granting them higher priority than the winner. In any particular cycle, one column will always be zero among all requesters, since there will always be one with the highest priority. If any channel is not actively requesting, it will assert all 0s instead of its actual priority bits to put it on the lowest priority possible. For example in Figure 3.11, in Cycle 0, only requesters 1 and 2 request the channel, and therefore only these two assert their priority bits while 0 and 3 assert all zeroes. The result of the bitwise OR would be 1100, and then each requester checks their corresponding bit, in which case requester 2 wins. Since requesters 0 and 3 did not request, it ignores the result of the bitwise OR. The winner, in this case requester 2, then clears its priority bits. Once granted, the requester can hold on to the channel until it chooses to free the channel. Requests can be coalesced in the crossbar, shown in Figure 3.11. Since the channel can observe all the requesters and their requesting addresses, it can simply compare them with the winner's address and grant to any matching requesters. Coalescence does not affect the priority status, since it happens after arbitration.

3.6.1.3 Reconfigurable Cache

The downstream L0 crossbar connects to the reconfigurable L0 cache consisting of four logical Static Random Access Memory (SRAM) banks, each of which consists of four physical SRAM banks. The L0 cache provides second-level coalescing by comparing the new requests with existing pending requests stored in the MSHRs. Along with tracking missed requests, the MSHRs also act as a request queue that takes in the inbound requests, a fill buffer that temporarily holds the returned data before storing to SRAM and a response queue that sends the read data back to the PEs. For coalescence, each MSHR entry stores a bit vector of all requesters and adds additional requesters, should any coalesce in the process. The upstream crossbar then multi-casts the read data back to the PEs based on the requester bit vector.

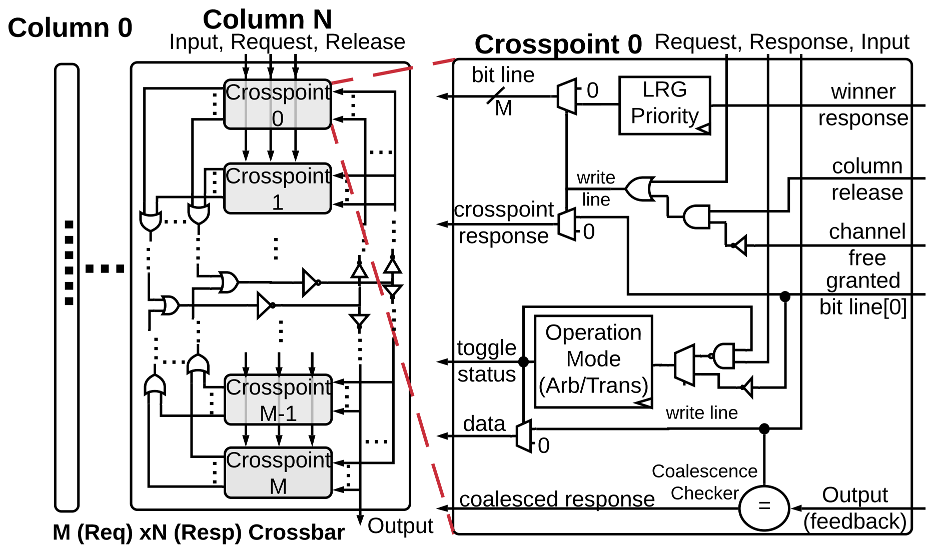


Figure 3.9: OR trees of the SSN crossbar. Each crosspoint is one requester and the bitwise OR'd results are sent back to each crosspoint.

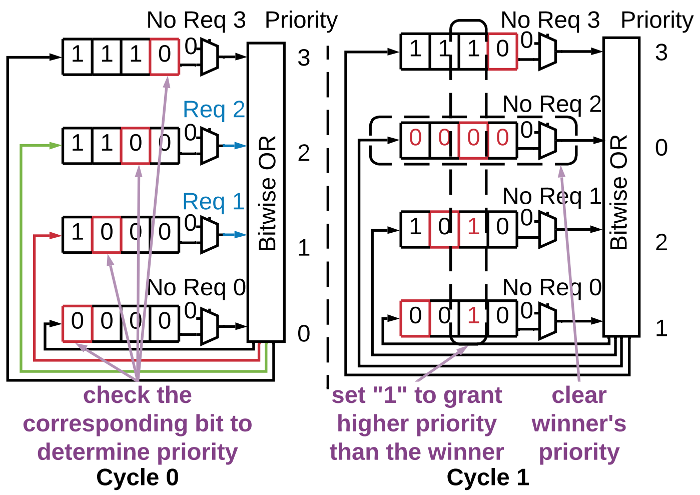


Figure 3.10: LRG scheme. The requesting nodes assert their priority bits, and the winner is determined based on which requester receives a 0 in the corresponding response bit. The winner's priority bits are then cleared.

For the *multiply* phase, the L0 is a multi-banked set-associative cache, allowing NZEs of B to be shared. For *merge*, it is reconfigured into a multi-banked SPM by disabling the tag array and the Least-Recently Used (LRU) counter and is private to each M0-M4 pair. Through another set of coalescing crossbars, the L0 cache in each tile connects to the L1 layer, which interfaces to the FSB.

Only minor modifications were made to the cache controller to enable reconfiguration into SPM mode. In the SPM mode, the tag arrays and the set index bits are disabled, and the controller addresses directly into each SRAM bank.

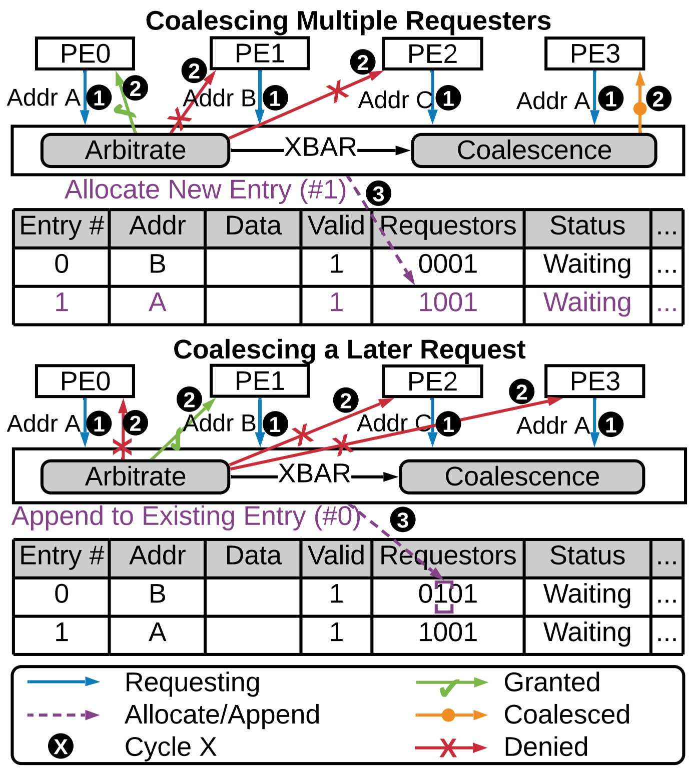

Coalescing Multiple Requesters
PE0
PE1
PE2
PE3
Addr A
Addr B
Addr C
Addr A
Arbitrate
XBAR
Coalescence
Allocate New Entry (#1)
Entry # | Addr | Data | Valid | Requestors | Status | ...
0 | B | | 1 | 0001 | Waiting | ...
1 | A | | 1 | 1001 | Waiting | ...
Coalescing a Later Request
Append to Existing Entry (#0)
Entry # | Addr | Data | Valid | Requestors | Status | ...
0 | B | | 1 | 0101 | Waiting | ...
1 | A | | 1 | 1001 | Waiting | ...
Requesting
Allocate/Append
Cycle X
Granted
Coalesced
Denied

Figure 3.11: Crossbar and cache coalescence. The crossbar coalesces identical requests by marking the requesters in a bit vector, which is then stored in the cache controller. While it is in the cache controller, more requests can be coalesced along the way should there be any requesters asking for the same address.

3.6.2 Outer Product Mapping

The implementation details of the outer product algorithm on our chip are presented in this section. In the *multiply* phase, each PE multiplies an element of a column of the first operand (**A**) with a row of the second operand (**B**). For the *merge* phase, the chip reconfigures to enable the Arm core substrate and SPM in the L0. The two cores act as a single unit to stream in the results of *multiply*, perform *merge-sort*, and store the final results to the off-chip DRAM. Our studies reveal that a SPM leads to better performance than a cache for this phase, due to the irregular nature of data accesses (Figure 3.16).

In the *multiply* phase, each PE multiplies a non-zero element of column i of A with all non-zero elements of row i of B to produce one Partial Product Matrix (PPM) row. Each NZE is fetched only once. The PPMs are stored as a set of linked lists of pointers to "chunks" in the DRAM, as shown in Figure 3.2. The *multiply* phase computes multiplications of *all* combinations of fetched elements, resulting in maximum reuse of inputs *without* any index matching, thus circumventing the problem of unproductive loads. Since each PE traverses through the non-zero elements of a row in Matrix B, the memory access during this phase is sequential and predictable. In addition, multiple PEs operate on the same row for each column element that corresponds to the row, resulting in high data reuse across the PEs.

In the *merge* phase, each M4 core is assigned a pointer array of chunks that correspond to a single row of the result matrix C, as shown in Figure 3.2. When merging the different chunks, the M4 core needs to ensure that all the elements in the final row are *ordered by their column index*. To ensure this ordering, each M4 core maintains a sorting list. The sorting list only needs to be big enough to hold one element from every chunk that is being merged by this core. This is because all the chunks are ordered by their column indices when they are produced in the *multiply* phase. Once the first element of every chunk is inserted into the sorting list, the steady state involves writing out the smallest element to DRAM and fetching one element to be sorted.

The *merge* phase, as shown in Figure 3.12, is broken down into three steps: initialization, sorting list construction, and on-demand sorting.

Step 1. Initialization. Each chunk is augmented with metadata that is used to keep track of the number of elements that have been fetched by the core. During initialization, the metadata of each chunk assigned to the core is written into SPM.

Step 2. Sorting List Construction. The M4 core begins constructing the sorting list by inserting the head of every chunk into the list (Step 2a). The list is sorted again each time an element is pushed into the list, based on the column index. The core iterates over all of its assigned chunks, and so the list starts with first element of every chunk. As the M4 core inserts elements from the SPM into the sorting list, the M0 core fetches the next elements of the chunk into the new empty blocks (Step 2b).

Step 3. On-Demand Sorting. The M4 core *pops* the smallest element of the list to be placed in the output buffer (Step 3a). The M4 core checks the chunk that this popped element originated from, fetches the next element in the chunk, and *pushes* it into the sorting list (Step 3b). The popped element is compared against the element that is currently in the output buffer. If the indices of the two elements match, the values of the two elements are summed. If the indices do not match, the element in the output buffer is written to memory as the first element of one row in the result Matrix C. The popped element then becomes the new element in the output buffer, and the next element is fetched from the chunk of the last popped element. This process is repeated until all the assigned chunks have been processed. As the M4 core

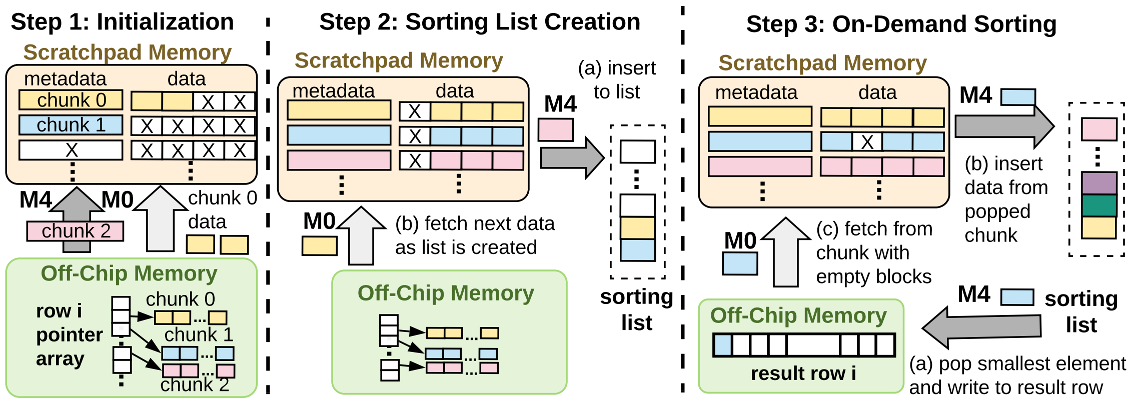


Figure 3.12: Breakdown of the three steps of merge phase: initialization, sorting list construction, and on-demand sorting. The M4 performs sorting on the data that has been loaded into the SPM by the M0.

consumes data from the SPM, the M0 core independently fetches the data of each chunk onto the emptied blocks (Step 3c).

Unlike the *multiply* phase where most of the memory accesses are sequential and there is plenty of data sharing between different PEs, the data accesses of the merge phase are mostly irregular, with no shared data across the Arm core pairs. Each core is assigned a disjoint pool of chunks, so that each Arm core pair operates on independent memory space. The location of each memory load is determined by the element that was popped from the sorting list. Therefore, the memory access is highly irregular and difficult to predict. Because the two phases have such drastically different access patterns, we implemented a reconfigurable architecture that can tune its memory hierarchy based on the needs of each phase.

In this implementation, we use a linear sorting algorithm, where the element to be inserted into list is compared one-by-one down the list, until a smaller element is found. While a priority queue-based sorting algorithm has better scalability as the length of the sorting list increases, the high overhead of managing the binary tree favors linear sorting when it is small.

3.6.2.1 Scratchpad Prefetching

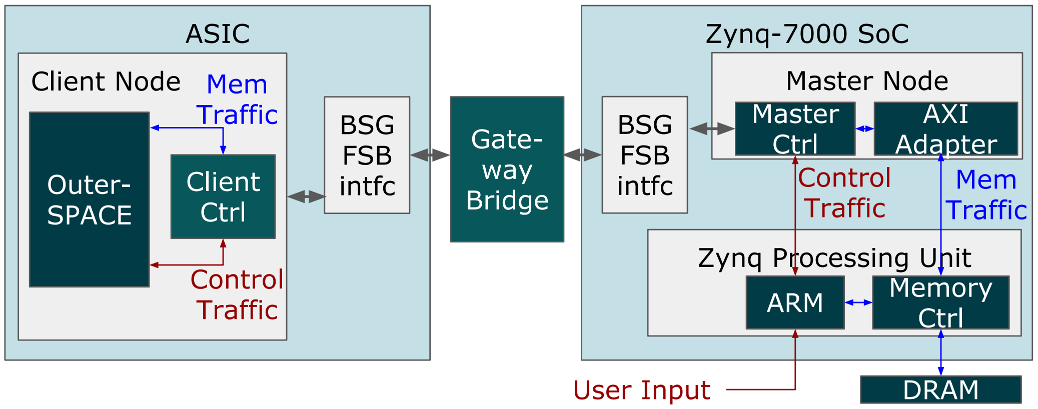
While all the core computation of the *merge* phase is handled by the M4 cores, each M4 is paired with an M0 core (Figure 3.8(b)), which acts as a programmable prefetcher. The primary purpose of the M0 core is to fill the private SPM with the elements of the PPM rows, so that the M4 core can grab its data from the SPM instead of the memory.

The M0 starts fetching the head elements of the chunks at the initialization step. As shown in Figure 3.12, the M0 begins fetching data once the metadata of a chunk has been registered into the SPM. This allows the M4 core to immediately proceed to the construction of its sorting list, without waiting on the memory. As the M4 core pushes a new element from the SPM into the sorting list, the M0 core loads the next element of the chunk into the evicted space, until all the elements have been consumed.

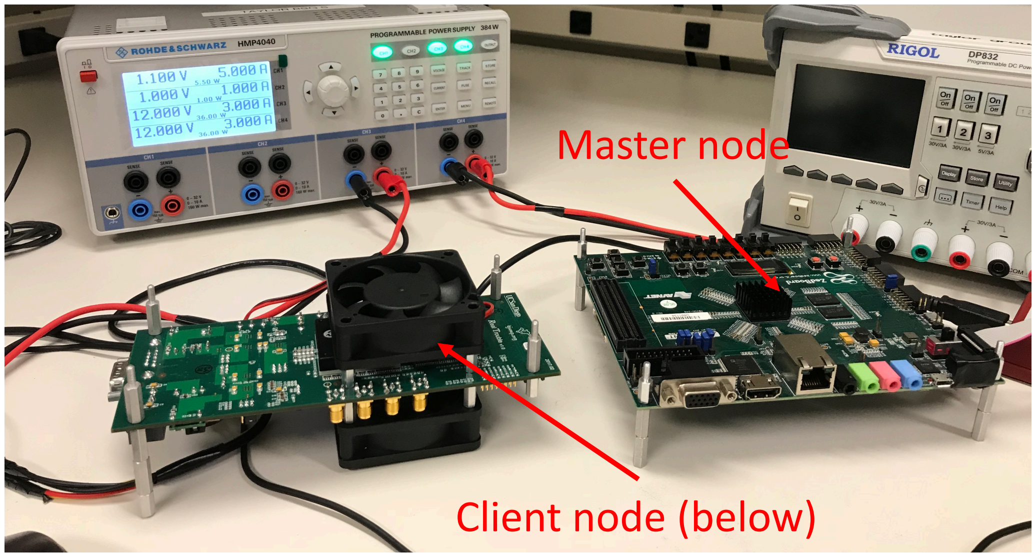
Due to the size of the local SPM, there is a limit to the number of chunks that can be held in the SPM. This also limits the length of the sorting list maintained by the M4, since the length of the sorting list is equal to the number of individual chunks being merged. When the total number of chunks assigned to an Arm core pair exceeds the maximum length of the sorting list (L), the PPM rows are divided into subgroups of L PPM rows. The merge phase is then performed in multiple passes, each one generating an intermediate result of L merged chunks. During each pass, the intermediate results are written out to a temporary space in memory. Once there is enough capacity to merge the remaining chunks as well as the intermediate results, the final merge pass produces a single, fully merged row of result matrix C. These intermediate passes are expensive because the data needs to be stored in external memory, and read again during the final merge pass. To minimize the number of passes, L needs to be as high as possible. However, for the M0's prefetching to be effective, each chunk needs to have sufficient number of elements that have been loaded ahead in the SPM. Therefore, there exists a trade-off between the number of PPM rows that is tracked during the merge phase, and the number of elements that can be prefetched into the SPM for each PPM row.

3.6.3 Experimental Setup

We use the BaseJump [4] infrastructure for testing our chip, a schematic and photograph of which are shown in Figure 3.13. The chip is placed on a RealTrouble board and connects through an FMC interface to a Zynq-7000 FPGA. The FSB controller in OuterSPACE connects through asynchronous FIFOs to the client control block, which is responsible for merging memory/control traffic and sending them to the FSB interface block. This block communicates with the FPGA via a source-synchronous DDR interface. The master node decodes and manages the control/memory FSB traffic and streams instructions to the Arm cores on OuterSPACE. The memory traffic goes through an AXI-adapter that converts the FSB packets to AXI packets, which then reach the Zynq memory controller. The FPGA Arm core sends out control pack-



(a) End-to-end testing infrastructure showing the master-client interface.



(b) Photograph of the test setup showing the RealTrouble board (left) and Zynq-7000 FPGA (right).

Figure 3.13: Schematic of the test setup and photograph of the testbed.

ets to the master node, initiated by user input, and accesses DRAM via the memory controller.

3.6.4 Measured Results and Evaluation

The performance of our 2.0 mm×2.6 mm accelerator for Sparse Matrix-Matrix multiplication, with the chip layout shown in Figure 3.14, was evaluated through matrix squaring on synthetic matrices, as well as power-law graphs that are representative of real-world sparse matrices [33], [178]. The measured characteristics of the chip are summarized in Table 3.8.

At the optimal frequency and voltage points, the accelerator achieves an energy

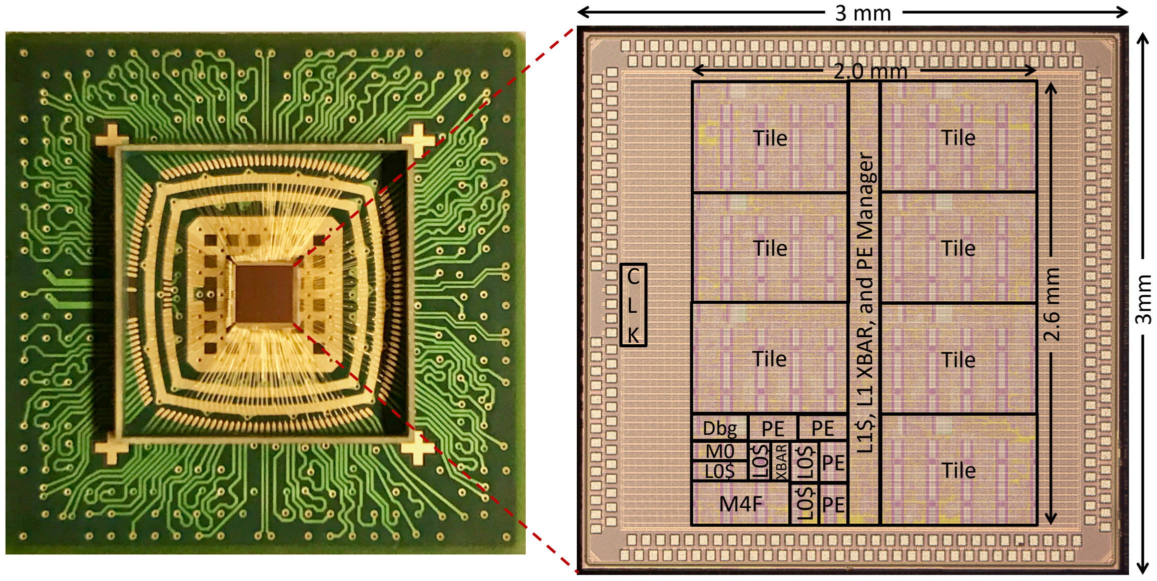


Figure 3.14: Annotated die photo of the fabricated 3.0 mm × 3.0 mm chip with GDS overlay. There are eight tiles per chip, each tile containing an Arm Cortex-M0 core, a Cortex-M4F core, and four PEs.

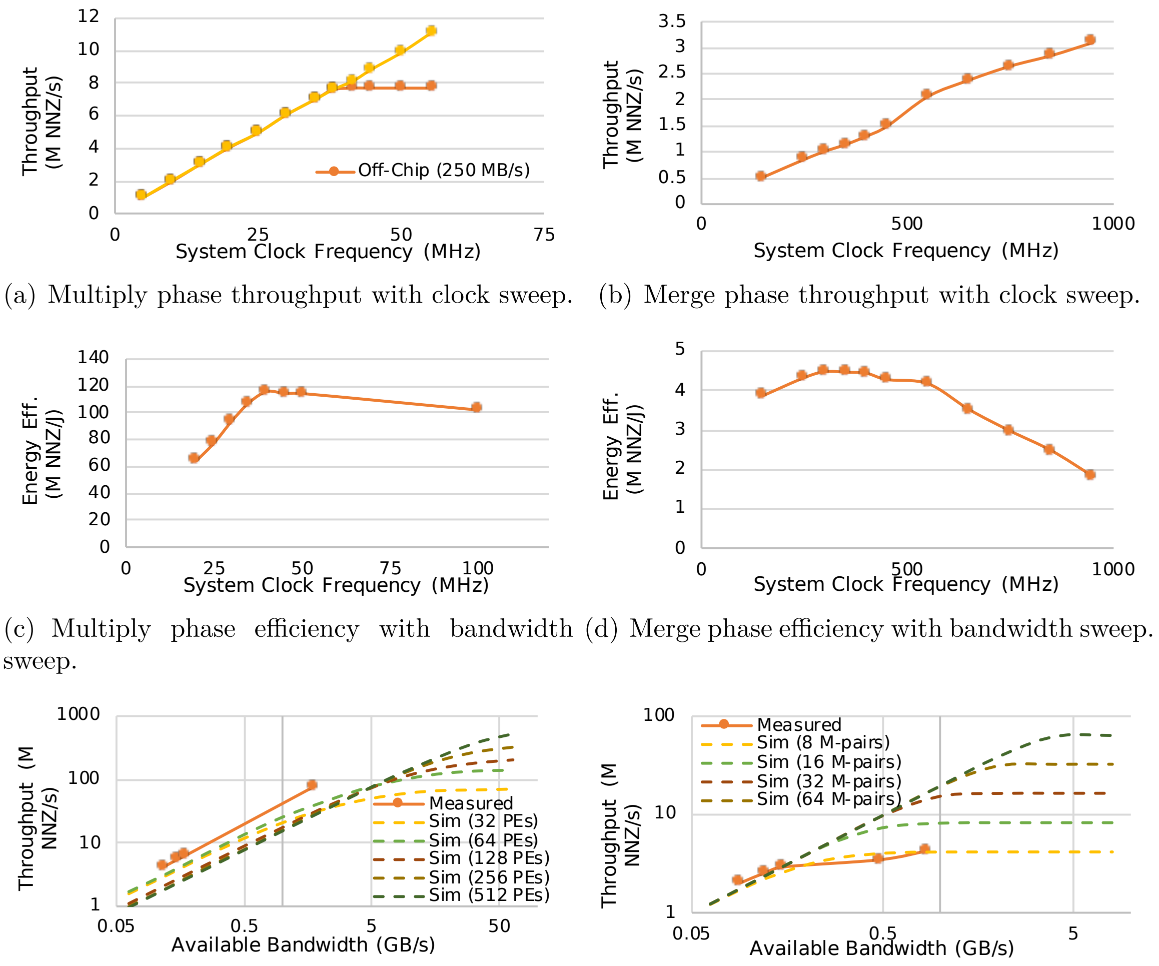
Table 3.8: Characterization summary of the chip.

Technology	40 nm CMOS
Die Size	3.0 mm × 3.0 mm
Block Size	2.0 mm × 2.6 mm
# Transistors	25,134,927
Total SRAM	112 KB
Data Precision	Single-Precision Floating Point
Nominal Frequency (Minimum Energy)	41.7 MHz 0.860 V (Multiply) 352.0 MHz 0.864 V (Merge)
Maximum Frequency	950.0 MHz 1.27 V
Nominal Power Consumption	66.6 mW (Multiply) 226.0 mW (Merge)

efficiency of 6.1-8.4 M NNZ/J and bandwidth efficiency of 6.4-15.5 M NNZ/GB. The SSN crossbar gives the chip a 24.9% performance gain at 86.3% the energy and 1.3% more area over a MUX crossbar based design.

3.6.4.1 Frequency and Bandwidth Sweep

Figure 3.15 shows the clock and bandwidth sweeps for a square sparse matrix of dimension 100,000 and density of 0.0008%. The multiply and the merge phases were evaluated separately in order to determine the optimal parameters for each phase. Clock sweeps show that while multiply performance hits a roofline, merge



(a) Multiply phase throughput with clock sweep.

(b) Merge phase throughput with clock sweep.

(c) Multiply phase efficiency with bandwidth sweep.

(d) Merge phase efficiency with bandwidth sweep.

(e) Multiply phase throughput with bandwidth sweep.

(f) Merge phase throughput with bandwidth sweep.

Figure 3.15: Clock and bandwidth sweeps for SpMM between matrices with dimension 100,000 and density of 0.0008%. For measurements with increased bandwidth, an on-chip LFSR is used for the multiply phase and the M0 is used for the merge phase.

performance saturates slowly, as merge is more compute-heavy due to the overhead of maintaining the sorting list. We observe the frequency and voltage level in which the chip achieves optimal energy efficiency to be at 41.7 MHz and 0.860 V for the multiply phase, and 352.0 MHz and 0.864 V for the merge phase.

For the bandwidth sweeps, simulation results are appended to measured results to illustrate the impact of higher bandwidth and more compute units. The performance of multiply phase continues to increase with higher bandwidth, while the merge phase reaches saturation early, at less than 1 GB/s. The points at which the throughput saturates for each of the phases shows that the multiply phase is ∼30× more sensitive to external bandwidth than the merge phase.

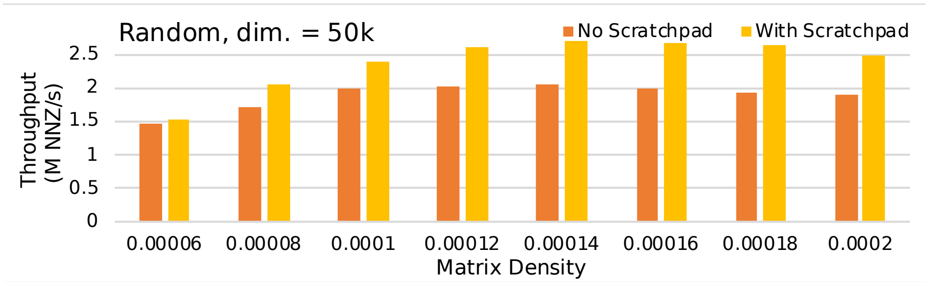


Figure 3.16: Measured merge phase performance with and without scratchpad memory in the L0 layer. Overall performance benefit of scratchpad is 25.7%.

Based on the frequency and bandwidth scaling of the chip, scaling out our current chip to 16× the current configuration would meet the CPU's performance at 9.5× less bandwidth, 16.7× lower power and 0.08× the area. At this configuration, the chip will be able to make optimal use of available bandwidth by minimizing off-chip traffic.

3.6.4.2 Benefits of Reconfigurable Memory

One of the key design choices of the chip is the use of reconfigurable memory that transitions between cache and SPM based on the demands of the algorithm. For workloads with well-defined data access and reuse patterns, the SPM improves performance over the cache by preventing any data that will be reused by the program from getting evicted out to memory during intermediate computation, ensuring each critical data to only be fetched once. Figure 3.16 shows the benefit of using the SPM during the merge phase at varying matrix densities, but with the matrix dimension fixed. We observe an average performance benefit of 25.7% across the different matrices, with higher benefits for denser matrices.

3.6.4.3 Comparison with State-of-the-Art Approaches

Figure 3.17 compares the energy and bandwidth efficiency of the chip executing sparse matrix-matrix multiplication against the highly-optimized, commercial software libraries on a high-end CPU (Intel Core i7) and GPU (Tesla V100). The matrix dimension, density, and pattern of non-zeros were varied to observe how different platforms react to each matrix parameter. For matrices with a uniformly-random distribution of non-zeros, the chip exhibits greater bandwidth efficiency for larger and denser matrices for both the CPU and GPU. In contrast, the improvement in energy efficiency over the CPU is more prominent when the matrix is small and

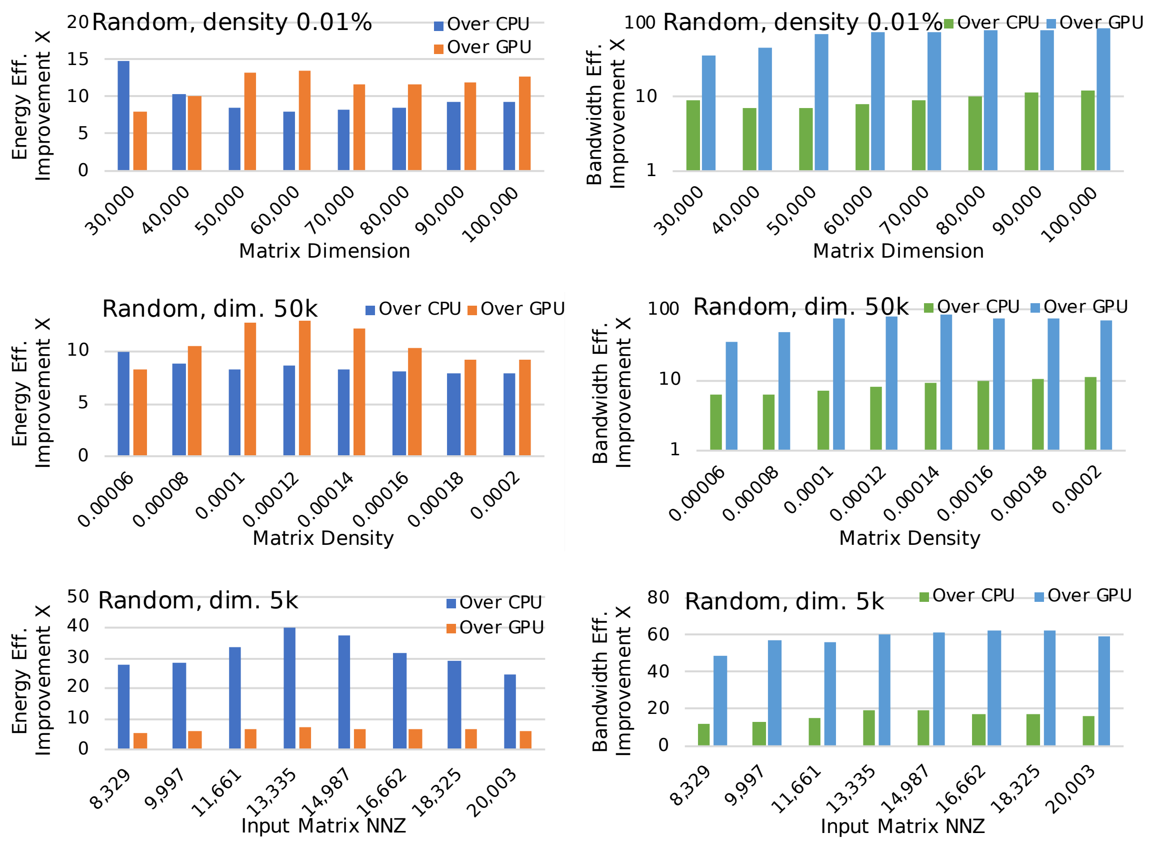


Figure 3.17: Measured results over different matrices showing energy and bandwidth efficiency of the proposed chip on uniform random matrices, normalized to a Core i7 CPU and V100 GPU running SpMM packages.

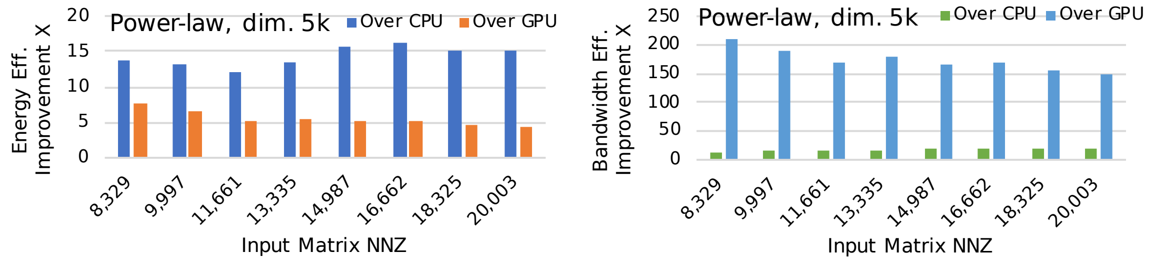


Figure 3.18: Energy and bandwidth efficiency of the proposed chip on power-law graphs with matrix dimension of 5,000, normalized to a Core i7 CPU and V100 GPU running SpMM packages.

sparse, but relatively constant against the GPU at any matrix size or density. This is because the performance of CPU degrades more prominently as density is lowered as amount of extraneous computation increases. On the other hand, GPU performance is relatively consistent because work is scheduled in large batches, and thus

Table 3.9: Key metrics and comparison vs. CPU/GPU and prior work.

Platform / **Feature**	**Core i7-6700K (MKL)**	**Tesla V100 (CUSP)**	**DSP** [53] *	**ASIC** [11] *	**This work**
Kernel	SpMM	SpMM	SpMV†	DMM‡	SpMM
Max. Matrix Dim.	120,000	120,000	217,918	256	120,000
Min. Matrix Density	0.002%	0.002%	0.003%	3%	0.002%
Reconfigurability	×	×	×	✓	✓
Process (nm)	14	12	40	14	40
Core Count	8	5120	4	16	48
Total Core Area (mm^2)	122.00	815.00	0.93	0.02	5.20
Frequency (MHz)	4000	1250	515	800	744^
Off-Chip Memory Bandwidth (GB/s)	34.10	900.00	3.20–8.53	N/A (on-chip only)	0.24
Power (W)	58.84	123.95	0.06	0.04	0.25^
Compute Density (NNZ/s/mm^2) [$\times 10^6$] **	0.0279	0.0129	9.0183	N/A	0.4775
Energy Efficiency (NNZ/J) [$\times 10^6$]	0.58††	0.87‡‡	129.95††	N/A	7.28††
Bandwidth Efficiency (NNZ/GB) [$\times 10^6$]	1.00	0.15	0.98–2.61	N/A	11.73^

* Not directly comparable to this work
† Sparse Matrix-Vector Multiplication ‡ Dense Matrix-Matrix Multiplication
** Area normalized to 40 nm technology ** NNZ/s is used in place of FLOPS/s to count only the operations that produce meaningful results.
^ 744 MHz for bandwidth efficiency. Multiply phase at 41.7 MHz and merge phase at 352 MHz were used for energy efficiency and power. †† Only on-chip energy ‡‡ Combines on-chip and off-chip energy. The GPU cores account for ∼75% of the total energy [125]

less sensitive to changes in data.

In the case of power-law graphs, as shown in Figure 3.18, the improvement in bandwidth efficiency exhibits a slight decrease with increasing NNZ for the GPU. The power-law graphs were synthetically generated using the Graph500 R-MAT data generator [144] to emulate the characteristics of real-world graph datasets.

Table 3.9 summarizes the key metrics of this work compared with that of the CPU, the GPU, a DSP [53], and an ASIC [11]. The DSP is designed specifically for sparse matrix-vector multiplication, and the ASIC focuses on multiplication between matrices with relatively higher density (≥3%) using only on-chip storage. Therefore, these two works cannot be directly compared to our work. Our work built the first chip that aims at accelerating sparse matrix-matrix multiplication for real-world sized sparse matrices and addresses the off-chip memory bottleneck. The chip consumes 0.25 W on average when operating at its optimal energy efficiency point of 41.7 MHz for multiply and 352 MHz for merge. In general, our chip achieves an average energy efficiency gain of 12.6× against the CPU and 8.4× against the GPU. The compute density of the chip, which is throughput (NNZ/s) per area, is 17.1× that of the CPU, and 37.1× that of the GPU. The bandwidth efficiency is a key metric measuring the number of non-zero elements in the result matrix computed per bandwidth used; it shows how well the accelerator can make use of the available bandwidth. This work is able to achieve 11.7× and 77.6× improvements in terms of bandwidth efficiency compared to the CPU and the GPU, respectively.

3.7 Conclusion

The Intel MKL and NVIDIA CUSP and cuSPARSE libraries are successful state-of-the-art libraries for sparse linear algebra. However, our experiments and analyses show that MKL and cuSPARSE work optimally only for regular sparse matrices. While CUSP is insensitive to the irregularity of sparse matrices, it introduces extra memory overheads for the intermediate storage [130].

In this chapter, we discussed an outer product based matrix multiplication approach and evaluated its performance on traditional CPUs and GPUs. We discovered inefficiencies in these architectures, which lead to sub-optimal performance of the outer product algorithm, and resolved them by building a new custom reconfigurable hardware. Our novelty lies in the efficient co-design of the algorithm implementation and the hardware. We demonstrate OuterSPACE's efficiency for two key kernels, sparse matrix-matrix multiplication and sparse matrix-vector multiplication, which form the building blocks of many major applications.

The reconfigurable memory hierarchy of OuterSPACE, which adapts to the contrary data-sharing patterns of the outer product algorithm, aids in reducing the number of off-chip accesses. This, coupled with the increased flexibility across PEs through SPMD-style processing (due to lack of synchronization and coherence overheads), enables OuterSPACE to achieve good throughput and high speedups over traditional hardware. Energy savings are attributed to the bare-bone PE and energy-efficient swizzle-switch crossbar designs [182].

In essence, our work demonstrated that a massively-parallel architecture consisting of asynchronous worker cores, coupled with memory hierarchies that are tailored to retain reusable data, uncovers an enormous potential to accelerate kernels that are heavily memory-bound. Our simulated 256-core OuterSPACE design achieves speedups of 7.9×, 13.0× and 14.0× over the MKL, cuSPARSE and CUSP libraries, respectively. Moreover, our fabricated 40 nm chip is the first custom SpMM accelerator that addresses the off-chip memory access bottleneck for real-world sized matrices, evaluating densities $\geq$0.002% and dimensions $\leq$120k. The chip delivers an energy efficiency of 7.3 M output NNZ/J, which is 12.6× and 8.4× higher than that achieved by state-of-the-art software libraries on the CPU and GPU, respectively. The ability to switch from cache to SPM in different phases of the workload resulted in speedups of up to 27.3%. Lastly, our solution achieves improvements of 11.7× and 77.6× compared to the CPU and GPU, in terms of bandwidth efficiency, which is the key figure-of-merit for memory-bound workloads such as SpMM.

CHAPTER IV

Accelerating Mixed-Data Applications using Memory and Dataflow Reconfiguration

This chapter presents a novel effort to bridge the gap between General-Purpose Processors (GPPs) and Application-Specific Integrated Circuits (ASICs) with a flexible accelerator called Transmuter. Transmuter adapts to changing kernel characteristics, such as data reuse and control divergence, through the ability to reconfigure the on-chip memory type, resource sharing and dataflow at run-time within a short latency. This is facilitated by a fabric of light-weight cores connected to a network of reconfigurable caches and crossbars. Transmuter addresses a rapidly growing set of algorithms exhibiting dynamic data movement patterns, irregularity, and sparsity, while delivering GPU-like efficiencies for traditional dense applications. Finally, in order to support programmability and ease-of-adoption, we prototype a software stack composed of low-level runtime routines, and a high-level language library called TransPy, that cater to expert programmers and end-users, respectively.

The evaluations with Transmuter show average throughput (energy-efficiency) improvements of 5.0× (18.4×) and 4.2× (4.0×) over a high-end CPU and GPU, respectively, across a diverse set of kernels predominant in graph analytics, scientific computing, and machine learning. Transmuter achieves energy-efficiency gains averaging 3.4× and 2.0× over prior FPGA and CGRA implementations of the same kernels, while remaining on average within 9.3× of state-of-the-art ASICs.

The work presented in this chapter was published in PACT 2020 [155]. Detailed evaluation of deep neural network inference and dense linear algebra kernels for a subset of the proposed architecture appeared in ISCAS 2020 [212] and ICASSP 2020 [187], respectively. Finally, a hardware-software co-reconfigurable framework for SpMV-based graph analytics on Transmuter appears in DAC 2021 [63].

4.1 Introduction

Section 2.1 discussed the contemporary architectural paradigms, namely ASICs, CGRAs, FPGAs and GPPs, and their strengths and weaknesses in terms of the flexibility-efficiency trade-off that plagues modern computer architecture. In the remainder of this chapter, we describe a novel architecture called Transmuter that builds on top of OuterSPACE and takes a step toward bridging the flexibility-efficiency gap.

Transmuter, a reconfigurable accelerator that adapts to the nature of the kernel through a flexible fabric of light-weight cores, and reconfigurable memory and interconnect. Worker cores are grouped into tiles that are each orchestrated by a control core. All cores support a standard ISA, thus allowing the hardware to be fully kernel-agnostic. Transmuter overcomes inefficiencies in vector processors such as GPUs for irregular applications [154] by employing a Multiple Instruction, Multiple Data (MIMD) / SPMD paradigm. On-chip buffers and SPMs are used for low-cost scheduling, synchronization, and fast core-to-core data transfers. The cores interface to an HBM through a two-level hierarchy of reconfigurable caches and crossbars.

Our approach fundamentally differs from existing solutions that employ gate-level reconfigurability (FPGAs) and core/pipeline-level reconfigurability (most CGRAs); we reconfigure the on-chip memory type, resource sharing, and dataflow, at a coarser granularity than contemporary CGRAs, while employing general-purpose cores as the compute units. Moreover, Transmuter's reconfigurable hardware enables run-time reconfiguration within 10s of nanoseconds, faster than existing CGRA and FPGA solutions (Section 2.1).

We further integrate a prototype software stack to abstract the reconfigurable Transmuter hardware and support ease-of-adoption. The 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 HLL libraries in Python, called TransPy, that exposes optimized Transmuter kernel implementations to an end-user. Libraries are written by experts using the C++ intrinsics to access reconfigurable hardware elements. These libraries are then packaged and linked to existing HLL libraries, e.g. NumPy, SciPy, etc.

In summary, this chapter presents the following contributions.

- **Proposes a general-purpose, reconfigurable accelerator design** composed of a sea of parallel cores interweaved with a flexible cache-crossbar hierarchy that supports fast run-time reconfiguration of the memory type, resource sharing and dataflow.

Table 4.1: Characteristics of Transmuter vs. the architectures in Figure 2.1.

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

- **Demonstrates the flexibility of Transmuter** by mapping and analyzing six fundamental compute- and memory-bound kernels, that appear in multiple HPC and datacenter applications, onto three distinct Transmuter configurations.

- **Illustrates the significance of fast reconfiguration** by evaluating Transmuter on ten end-to-end applications (one in detail) spanning the domains of ML and graph, signal and image processing, that involve reconfiguration at kernel boundaries.

- **Proposes a prototyped compiler runtime and HLL library called TransPy** that expose the Transmuter hardware to end-users through drop-in replacements for existing HLL libraries. The stack also comprises of C++ intrinsics, which foster expert programmers to efficiently co-design new algorithms.

- **Evaluates the Transmuter hardware against existing platforms** with two proposed variants, namely TransX1 and TransX8, that are each comparable in area to a high-end CPU and GPU.

In summary, Transmuter demonstrates average energy-efficiency gains of 18.4×, 4.0×, 3.4× and 2.0×, over a CPU, GPU, FPGAs and CGRAs respectively, and remains within 3.0×-32.1× of state-of-the-art ASICs. Table 4.1 presents a qualitative summary of key differences of Transmuter in comparison to these architectures.

4.2 Motivation

In Section 2.2, we studied the kernels that compose a set of real-world applications from the domains of ML, graph analytics, and image and video processing. These underlying kernels show significant diversity in terms of arithmetic intensity, data reuse, and control divergence. Transmuter is primarily evaluated with these kernels, and we briefly introduce them here. General (dense) matrix-matrix multiplication (GeMM) and matrix-vector multiplication (General (dense) Matrix - Vector multiplication (GeMV)) are regular kernels in ML, data analytics and graphics [62, 67]. Convolution is a critical component in image processing [3] and convolutional neural networks [110]. Fast Fourier Transform (FFT) is widely used in speech and image processing for signal transformation [140, 14]. Sparse matrix-matrix multiplication (SpMM) is an important irregular kernel in graph analytics (part of GraphBLAS [104]), scientific computation [57, 18, 214], and problems involving big data with sparse connections [165, 88]. Another common sparse operation is sparse matrix-vector multiplication (SpMV), which is predominant in graph algorithms such as PageRank and Breadth-First Search [136], as well as ML-driven text analytics [10].

Takeaways. Figure 2.3 illustrates that real-world applications exhibit diverse characteristics not only across domains, but also within an application. Thus, taming both the inter- and intra-application diversity efficiently in a single piece of hardware calls for an architecture capable of tailoring itself to the characteristics of each composing kernel.

4.3 Related Work

A plethora of prior work has gone into building programmable and reconfigurable systems in attempts to bridge the flexibility-efficiency gap. A qualitative comparison of our work over related designs is shown in Table 4.2. Transmuter differentiates by supporting two different dataflows, reconfiguring faster at a coarser granularity, and supporting a COTS ISA/compiler.

4.3.1 Reconfigurability

A few prior work reconfigure at the sub-core level [134, 87, 105, 43, 171] and the network-level [75, 106, 198, 146]. In contrast, Transmuter uses native in-order cores and the reconfigurability lies in the memory and interconnect. Some recent work propose reconfiguration at a coarser granularity [124, 6, 171, 43]. PipeRench [73] builds an efficient reconfigurable fabric and uses a custom compiler to map a large logic configuration on a small piece of hardware. HRL [68] is an architecture for near-data processing, which combines coarse- and fine-grained reconfigurable blocks into a compute fabric. The Raw microprocessor [198] implements a tiled architecture focusing on developing an efficient, distributed interconnect. Stream Dataflow [146] and SPU [43] reconfigure at runtime, albeit with non-trivial overheads to initialize the Data-Flow Graph (DFG) configuration. Transmuter, on the other hand, relies on flexible memories and interconnect that enable fast on-the-fly reconfiguration, thus catering to the nature of the application.

4.3.2 Flexibility

Prior work has also delved into efficient execution across a wide range of applications. Plasticine [171] is a reconfigurable accelerator for parallel patterns, consisting of a network of Pattern Compute/Memory Units (custom SIMD Functional Units (FUs)/single-level SPM) that can be reconfigured at compile-time. Stream Dataflow [146] is a new computing model that efficiently executes algorithms expressible as DFGs, with inputs/outputs specified as streams. The design comprises a control core with stream scheduler and engines, interfaced around a custom, pipelined FU-based CGRA. SPU [43] targets data-dependence using a stream dataflow model on a reconfigurable fabric composed of decomposable switches and PEs that split networks into finer sub-networks. The flexibility of Transmuter stems from the use of general-purpose cores and the reconfigurable memory subsystem that morphs the dataflow and on-chip memory, thus catering to both inter- and intra-workload diversity.

4.3.3 Programmability

There have been proposals for programmable CGRAs that abstract the low-level hardware. Some work develop custom programming models, such as Rigel [103] and MaPU [210]. Others extend an existing ISA to support their architecture, such as Stitch [196] and LACore [190]. Plasticine [171] uses a custom DSL called Spatial [108]. Ambric [78] is a commercial system composed of asynchronous cores with a software stack that automatically maps Java code onto the processor-array. Transmuter distinguishes itself by using a standard ISA supported by a simple library of high-level language intrinsics and a COTS compiler, thus alleviating the need for ISA extensions or a DSL.

4.4 High-Level Architecture

The takeaways from the previous section are the fundamental design principles behind our proposed architecture, Transmuter. Transmuter is a tiled architecture composed of a massively parallel fabric of simple cores. It has a two-level hierarchy

Table 4.2: Qualitative comparison with prior work [171, 146, 43, 78, 198].

Architecture	**PE Compute Paradigm**	**Dataflow**	**Compiler Support**	**Reconfig. Granularity**	**On-chip Memory**
Plasticine	SIMD	Spatial	DSL	Pipeline-level, compile-time	SPM
Stream Dataflow	SIMD	Stream	ISA extn.	Network-level, run-time	SPM+FIFO
SPU	SIMD	Stream	ISA extn.	Network-/ Sub-PE-level, run-time	Compute-enabled SPM+FIFO
Ambric	MIMD/ SPMD	Demand-driven	Custom	Network-level, run-time	SPM+FIFO
RAW	MIMD/ SPMD	Demand-driven	Modified COTS	Network-level, run-time	Cache
Transmuter [this work]	**MIMD/ SPMD**	**Demand-driven/ Spatial**	**COTS**	**Network-/ On-chip-memory-level, run-time**	**Reconfig. Cache/SPM/ SPM+FIFO**

of crossbars and on-chip memories that allows for fast reconfiguration of the on-chip memory type (cache/scratchpad/FIFO), resource sharing (shared/private) and dataflow (demand-driven/spatial). The various modes of operation are listed in Table 4.3. The two levels of memory hierarchy, i.e. L1 and L2, supports 8 modes each. Furthermore, each Transmuter tile can be configured independently, however these tile-heterogeneous configurations are not evaluated in this work.

In this work, we identify three distinct Transmuter configurations to be well-suited for the evaluated kernels based on characterization studies on existing platforms (Section 4.2). These configurations are shown in Figure 4.1 and discussed here.

- **Shared Cache (Trans-SC).** Trans-SC uses shared caches in the L1 and L2. The crossbars connect the cores to the L1 memory banks and the tiles to the L2 banks, respectively. This resembles a manycore system but with a larger compute-to-cache ratio, and is efficient for regular accesses with high inter-core reuse.

- **Private Scratchpad (Trans-PS).** Trans-PS reconfigures the L1 cache banks into SPMs, while retaining the L2 as cache. The crossbars reconfigure to privatize the L1 (L2) SPMs to their corresponding cores (tiles). This configuration is suited for workloads with high intra-core but low inter-core reuse of data that is prone to cache-thrashing. The private L2 banks enable caching of secondary data, such as spill/fill variables.

- **Systolic Array (Trans-SA).** Trans-SA employs systolic connections between

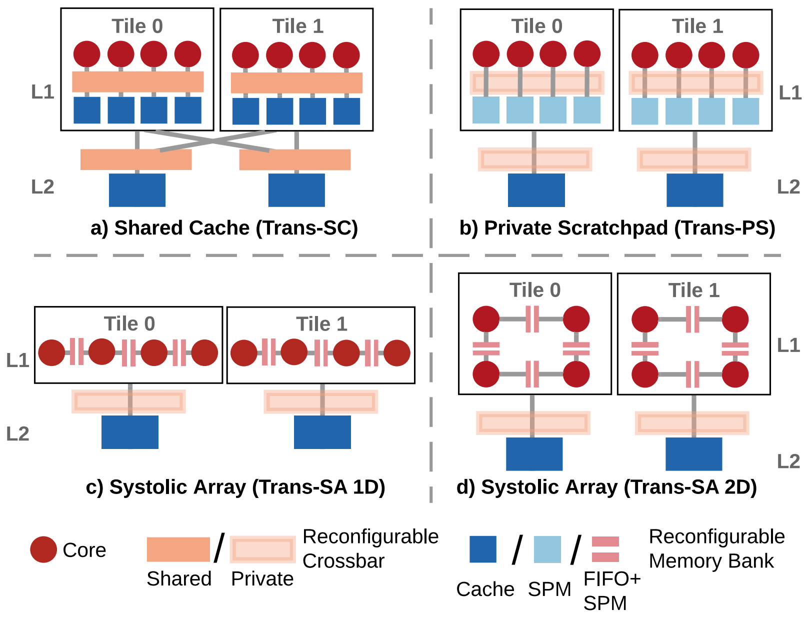

Tile 0
Tile 1
L1
L2
a) Shared Cache (Trans-SC)
b) Private Scratchpad (Trans-PS)
c) Systolic Array (Trans-SA 1D)
d) Systolic Array (Trans-SA 2D)
Core
Shared
Private
Reconfigurable Crossbar
Cache
SPM
FIFO+ SPM
Reconfigurable Memory Bank

Figure 4.1: High-level Transmuter architecture showing the evaluated configurations, namely a) Trans-SC (L1: shared cache, L2: shared cache), b) Trans-PS (L1: private SPM, L2: private cache), and c, d) Trans-SA (L1: systolic array, L2: private cache).

Table 4.3: Reconfigurable features at each level in Transmuter. In the "hybrid" memory mode, banks are split between caches and SPMs.

Dataflow	**On-Chip Memory**	**Resource Sharing**	**# Modes**
Demand-driven	Cache / SPM / Hybrid	Private / Shared	6
Spatial	FIFO + SPM	1D / 2D Systolic Sharing	2

the cores within each tile and is suited for highly data parallel applications where the work is relatively balanced between the cores. Transmuter supports both 1D and 2D systolic configurations. Note that the L2 is configured as a cache for the same reason as with Trans-PS.

We omit an exhaustive evaluation of all possible Transmuter configurations, given the vastness of the design space for algorithm mapping and hardware configuration choices. In the rest of this thesis, we use the notation of $N_T \times N_G$ Transmuter to describe a system with N_T tiles and N_G worker cores per tile.

4.5 Hardware Design

A full Transmuter system is shown in Figure 4.2-a. A Transmuter chip consists of one or more Transmuter (TM) clusters interfaced to HBM stack(s) in a 2.5D configuration, similar to modern GPUs [118]. A small host processor sits within

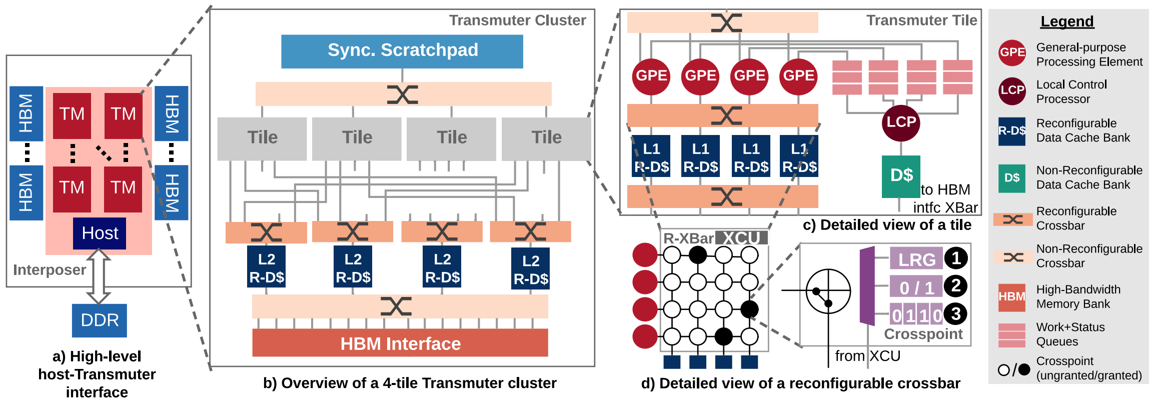

HBM
TM
Host
Interposer
DDR
a) High-level host-Transmuter interface
Transmuter Cluster
Sync. Scratchpad
Tile
L2 R-D$
HBM Interface
b) Overview of a 4-tile Transmuter cluster
Transmuter Tile
GPE
LCP
L1 R-D$
D$
to HBM intfc XBar
c) Detailed view of a tile
R-XBar
XCU
LRG
0 / 1
0110
Crosspoint
from XCU
d) Detailed view of a reconfigurable crossbar
Legend
General-purpose Processing Element
Local Control Processor
Reconfigurable Data Cache Bank
Non-Reconfigurable Data Cache Bank
Reconfigurable Crossbar
Non-Reconfigurable Crossbar
High-Bandwidth Memory Bank
Work+Status Queues
Crosspoint (ungranted/granted)

Figure 4.2: a) High-level overview of a host-Transmuter system. b) Transmuter architecture showing 4 tiles and 4 L2 R-DCache banks, along with L2 R-XBars, the synchronization SPM and interface to off-chip memory. Some L2 R-XBar input connections are omitted for clarity. c) View of a single tile, showing 4 GPEs and the work/status queues interface. Arbiters, instruction paths and ICaches are not shown. d) Microarchitecture of an R-XBar, with the circled numbers indicating the mode of operation: ①: ARBITRATE, ②: TRANSPARENT, ③: ROTATE.

the chip to enable low-latency reconfiguration. It is interfaced to a separate DRAM module and data transfer is orchestrated through Direct Memory Access (DMA) controllers (not shown) [66]. The host is responsible for executing serial/latency-critical kernels, while parallelizable kernels are dispatched to Transmuter.

4.5.1 General-Purpose Processing Element and local control processor

A GPE is a small processor with FP and load/store (LS) units that uses a standard ISA. Its small footprint enables Transmuter to incorporate many such GPEs within standard reticle sizes. The large number of GPEs coupled with MSHRs in the cache hierarchy allows Transmuter to exploit MLP across the sea of cores. The GPEs operate in a MIMD/SPMD fashion, and thus have private instruction (I-) caches.

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

4.5.2 Work and Status Queues

The LCP distributes work to the GPEs through private FIFO work queues. A GPE similarly publishes its status via private status queues that interface to the LCP

(Figure 4.2-c). The queues block when there are structural hazards, i.e. if a queue is empty and a consumer attempts a `POP`, the consumer is idled until a producer `PUSH`es to the queue, thus preventing wasted energy due to busy-waiting. This strategy is also used for systolic accesses, discussed next.

4.5.3 Reconfigurable Data Cache

Transmuter has two layers of multi-banked memories, called reconfigurable data caches, i.e. R-DCaches (Figure 4.2 – b, c). Each R-DCache bank is a standard cache module with 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 replacement policy. The banks are interleaved at a set-granularity, and a cacheline 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 powered off and the bank is accessed as a scratchpad.

- **FIFO+SPM.** A partition of the bank is configured as SPM, while the remainder are accessed as FIFO queues (Figure 4.3 – left), using a set of head/tail pointers. The queue depth can be reconfigured using memory-mapped registers. The low-level abstractions for accessing the FIFOs are shown in Figure 4.3 (right). This mode is used to implement spatial dataflow in Trans-SA (Figure 4.1).

4.5.4 Reconfigurable Crossbar

A multicasting $N_{\text{src}} \times N_{\text{dst}}$ crossbar creates one-to-one or one-to-many connections between N_{src} source and N_{dst} destination ports. Transmuter employs SSN-based crossbars that support multicasting [182, 92]. These and other work [1] have shown that crossbars designs can scale better, up to radix-64, compared to other on-chip networks. We augment the crossbar design with a Crosspoint Control Unit (XCU) that enables reconfiguration by programming the crosspoints. A block diagram of a reconfigurable crossbar (R-XBar) is shown in Figure 4.2-d. The R-XBars support the following modes of operation:

- **ARBITRATE.** Any source port can access any destination port and contended accesses to the same port get serialized. Arbitration is done in a single cycle using a least-recently granted policy [182], while the serialization latency varies between 0 and ($N_{\text{src}} - 1$) cycles. This mode is used in Trans-SC.

- **TRANSPARENT.** A requester can only access its corresponding resource, i.e. the crosspoints within the crossbar are set to 0 or 1 (Figure 4.2-d). Thus, the R-XBar is transparent and incurs *no* arbitration or serialization delay in this mode. Trans-PS (in L1 and L2) and Trans-SA (in L2) employ TRANSPARENT R-XBars.

- **ROTATE.** The R-XBar cycles through a set of one-to-one port connections programmed into the crosspoints. This mode also has no crossbar arbitration cost. Figure 4.4 illustrates how port multiplexing is used to emulate spatial dataflow in a 1D systolic array configuration (Trans-SA).

There are two L1 R-XBars within a tile (Figure 4.2-c). The upper R-XBar enables GPEs to access the L1 R-DCache, and the lower R-XBar amplifies on-chip bandwidth between the L1 and L2.

4.5.5 Synchronization

Transmuter implements synchronization and enforces happens-before ordering using two approaches. The first is *implicit*, in the form of work/status/R-DCache queue accesses that block when the queue is empty or full. Second, it also supports *explicit* synchronization through a global synchronization SPM for programs that require mutexes, condition variables, barriers, and semaphores. For instance, say that GPEs `0` and `1` are to execute a critical section (CS) in a program. With explicit synchronization, the programmer can instantiate a mutex in the synchronization SPM and protect the CS with it. The same can also be achieved through implicit synchronization, with the following sequence of events: ① both GPEs $\leftarrow$ LCP, ② LCP $\rightarrow$ GPE`0`, ③ GPE`0` executes the CS, ④ GPE`0` $\rightarrow$ LCP, ⑤ LCP $\rightarrow$ GPE`1`, ⑥ GPE`1` executes the CS, ⑦ GPE`1` $\rightarrow$ LCP, where $\leftarrow$ denotes `POP`-from and $\rightarrow$ is a `PUSH`-to the work or status queue.

Compared to traditional hardware coherence, these techniques reduce power through lower on-chip traffic [103, 154]. The synchronization SPM is interfaced to the LCPs and GPEs through a low-throughput two-level arbiter tree, as accesses to this SPM were not bottleneck for any of the evaluated workloads.

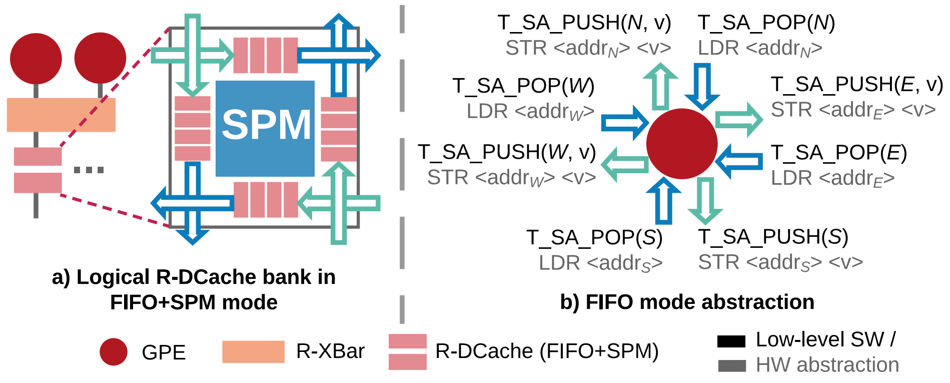


Figure 4.3: a) Logical view of an R-DCache bank in FIFO+SPM mode, showing 4 FIFO partitions, one for each 2D direction. b) Loads and stores to special addresses corresponding to each direction are mapped to POP and PUSH calls, respectively, into the FIFOs.

4.5.6 Miscellaneous Reconfiguration Support

The GPE LS unit is augmented with logic to route packets to the work/status queue, synchronization SPM, and the L1 or L2 R-DCache, based on a set of base/bound registers. Reconfiguration changes the active base/bound registers, without external memory traffic. LCPs include similar logic but do not have access to the L1 or L2. Lastly, the system enables power-gating individual blocks, i.e. cores, R-XBars, R-DCaches, based on reconfiguration messages. This is used to boost energy-efficiency for memory-bound kernels.

4.5.7 Reconfiguration Overhead

Transmuter can self-reconfigure at run-time (initiated by an LCP) if the target configuration is known *a priori*. Reconfiguration can also be initiated by the host using a command packet with relevant metadata. The programming interface used to initiate this is discussed in Section 4.6. Each step of the hardware reconfiguration happens in parallel and is outlined below.

- **GPE.** Upon receiving the command, GPEs switch the base/bound registers that their LS units are connected to (Section 4.5.6) in a single cycle.
- **R-XBar.** ARBITRATE $\leftrightarrow$ TRANSPARENT reconfiguration entails a 1-cycle latency, as it only switches MUXes in the R-XBar (Figure 4.2-d). The ROTATE mode uses set/unset patterns, which requires a serial transfer of bit vectors from on-chip registers (e.g. a 64×64 design incurs a 6-cycle latency[1]).

[1]Latency (in cycles) = ceil($N_{\text{rotate\_patterns}} \times N_{\text{dst}} \times \log_2(N_{\text{src}})$ / xfer_width)

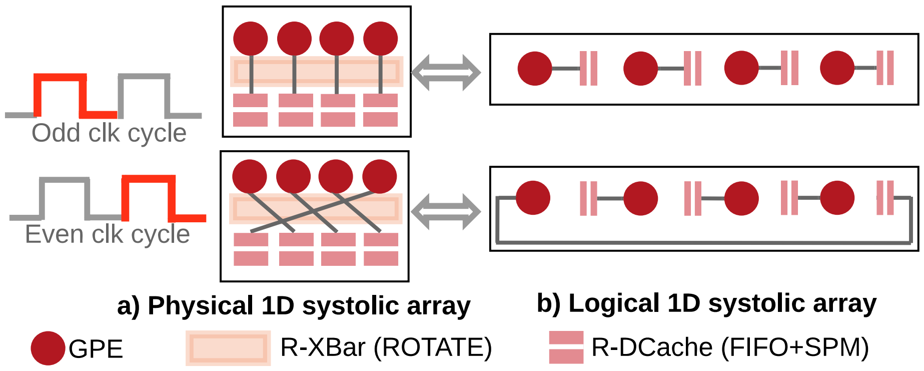


Figure 4.4: a) Physical and b) logical views of 1D systolic array connections within a Transmuter tile. Spatial dataflow is achieved by the R-XBar rotating between the two port-connection patterns.

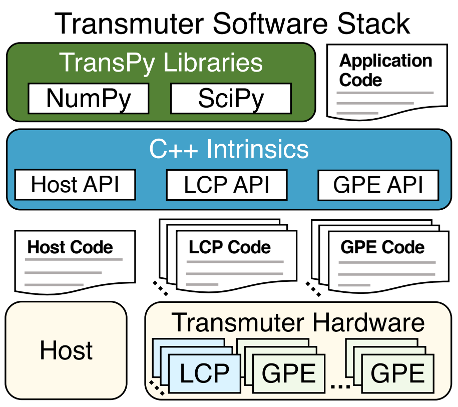
- **R-DCache.** Switching from CACHE to SPM mode involves a 1-cycle toggle of the scratchpad controller. The FIFO+SPM mode involves programming the head and tail pointer for each logical FIFO queue, which are transferred from control registers (4 cycles for 4 FIFO partitions).

Thus, the net reconfiguration time, accounting for buffering delays, amounts to *~10 cycles*, which is faster than FPGAs and many CGRAs (Section 2.1). For host-initiated reconfiguration, overheads associated with host-to-Transmuter communication leads to a net reconfiguration time of few 10s of cycles. We limit our discussions to self-reconfiguration in this work. Since Transmuter does not implement hardware coherence, switching between certain Transmuter configurations entails cache flushes from L1 to L2, from L2 to HBM, or both. The levels that use the SPM or FIFO+SPM mode do not need flushing. Furthermore, our write-no-allocate caches circumvent flushing for streaming workloads that write output data only once. Even when cache flushes are inevitable, the overhead is small (<1% of execution time) for the evaluated kernels in Section 4.9.

4.6 Prototype Software Stack

We implement a software stack for Transmuter in order to support good programmability and ease-of-adoption of our solution. The software stack has several components: a high-level Python Application Programming Interface (API), and lower-level C++ APIs for the host, LCPs and GPEs. An outline of the software stack and a working Transmuter code example are shown in Figure 4.5.

The highest level API, called TransPy, is a drop-in replacement for the well-known



App Code
```
1: import transpy.numpy as np
2: //Init input x and filter f
3: x=np.arange(...)
4: f=np.arange(...)
5: y=np.correlate(f, x, mode="full")
6: [...]
```

LCP Code
```
1: //Init input x & filter f
2: float x[x_N]={...};
3: float f[f_N]={...};
4: [...]
5: for(int i=0; i<f_N; ++i) {
6:   //Stream x to all GPEs
7:   T_WORKQ_PUSH_BCAST(x[i]);
8: }
9: [...]
```

GPE Code
```
 1: [...]
 2: //Iterate over size of array x
 3: for(int i=0; i<x_N; ++i) {
 4:   float x = T_WORKQ_POP();
 5:   float psum = 0;
 6: #if (GPE_ID != (N_TILES-1))
 7:   psum = T_SA_POP(Dir::East);
 8: #endif
 9:   psum += x * f; //MAC operation
10: #if (GPE_ID != 0)
11:   T_SA_PUSH(Dir::West, psum);
12: #else
13:   y[i] = psum;
14: #endif
15: }
16: [...]
```

Figure 4.5: Transmuter software stack. Application code is written using Python and invokes library code for the host, LCPs and GPEs. The implementations are written by experts using our C++ intrinsics library. Also shown is an example of a correlation kernel on Trans-SA (host library code not shown). The end-user writes standard NumPy code and changes only the import package to `transpy.numpy` (App:L1). Upon encountering the library call (App:L5), the host performs data transfers and starts execution on Transmuter. The LCP broadcasts the vector `x` to all GPEs (LCP:L7). Each GPE pops the value (GPE:L4), performs a MAC using its filter value (`f`) and east neighbor's partial sum (GPE:L7), and sends its partial sum westward (GPE:L11). The last GPE stores the result into HBM. The host returns control to the application after copying back the result `y`.

high-performance Python library NumPy, i.e. the TransPy API *exactly* mirrors that of NumPy. In the code example in Figure 4.5, note that only one change is needed to convert the NumPy program to TransPy. The `np.correlate` function is trapped in TransPy, dispatched to the Transmuter host layer, and a pre-compiled kernel library is invoked. We use `pybind11` [91] as the abstraction layer between Python and C++. TransPy also contains drop-in replacements for SciPy, PyTorch, NetworkX, and other libraries used in scientific computing, ML, graph analytics, etc.

TransPy invokes kernels that are implemented by library writers and expert programmers, with the aid of the C++ intrinsics layer. A Transmuter SPMD kernel implementation consists of three programs, one each for the host, LCP and GPE. The host code is written in the style of OpenCL [192], handling data transfers to and from Transmuter, launching computation, initializing reconfigurable parameters (e.g. R-DCache FIFO depth), and triggering reconfiguration if needed. On the Transmuter-side, notable API methods include those associated with the queue interface, for accessing SPMs and FIFOs, triggering cache flushes, and reconfiguration. Synchronization is handled using intrinsics that wrap around POSIX threads functions [142]. These calls allow for synchronization at different granularities, such as globally, within tiles, and across LCPs.

Thus, the Transmuter software stack is designed to enable efficient use of the Transmuter hardware by end-users, *without* the burden of reconfiguration and other architectural considerations. At the same time, the C++ layer allows for expert programmers to write their own implementations, such as sophisticated heterogeneous implementations that partition the work between the host CPU and Transmuter. As an alternative to writing hand-tuned kernels for Transmuter, we are actively working on prototyping a compiler to automatically generate optimized C++-level library code for Transmuter based on the LIFT data-parallel language [191], the details of which are left for a future work.

4.7 Kernel Mapping

Transmuter is built using COTS cores that lend the architecture to be kernel-agnostic. Here, we present our mappings of the fundamental kernels in Section 4.2 on the selected Transmuter configurations. Additional kernels in the domain of linear algebra have been mapped and evaluated on a preliminary version of Transmuter for different resource sharing configurations [187, 212].

We note that while executing memory-bound kernels, Transmuter powers-down resources within a tile to conserve energy.

4.7.1 Dense Matrix Multiplication and Convolution

GeMM. GeMM is a regular kernel that produces $O(N^3)$ FLOPS for $O(N^2)$ fetches and exhibits very high reuse [77]. It also presents contiguous accesses, thus showing amenability to a shared memory based architecture. Our implementation of GeMM on Trans-SC uses a common blocking optimization [123]. We similarly implement GeMM on Trans-PS but with the blocked partial results stored in the private L1 SPMs. Naturally, Trans-PS misses the opportunity for data sharing. For Trans-SA, the GPEs execute GeMM in a systolic fashion with the rows of A streamed through the L2 cache, and the columns of B loaded from the L1 SPM.

GeMV. GeMV is a memory-bound kernel that involves lower FLOPS/B than GeMM, i.e. $O(N^2)$ FLOPS for $O(N^2)$ fetches, but still involves contiguous memory accesses [64]. The Trans-SC and Trans-PS implementations are similar to those for GeMM, but blocking is not implemented due to lower data reuse. On Trans-SA, the vector is streamed into each GPE through the L2 cache, while the matrix elements are fetched from the L1 SPM. Each GPE performs a MAC, and passes the partial sum and input matrix values to its neighbors. We avoid network deadlock in our GeMM and GeMV Trans-SA implementations by reconfiguring the FIFO depth of the L1 R-DCache (Section 4.5.3) to allow for sufficient buffering.

Conv. Conv in 2D produces $(2 \cdot F^2 \cdot N^2 \cdot IC \cdot OC)/S$ FLOPS, for an $F{\times}F$ filter convolving with stride S over an $N{\times}N$ image, with IC input and OC output channels. The filter is reused while computing one output channel, and across multiple images. Input reuse is limited to $O(F \cdot OC)$, for $S{<}F$. On Trans-SC, we assign each GPE to compute the output of multiple rows, to maximize the filter reuse across GPEs. For Trans-PS and Trans-SA, we statically 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 GPE for Trans-PS, and to a set of F adjacent GPEs of a 1D systolic array for Trans-SA using a row stationary approach similar to [34].

4.7.2 Fast Fourier Transform

FFT. FFT in 1D computes an N-point discrete Fourier transform in $\log(N)$ sequential *stages*. Each stage consists of $N/2$ *butterfly* operations. FFT applications often operate on streaming input samples, and thus are amenable to spatial dataflow architectures [90, 58]. Our Trans-SA mapping is similar to pipelined systolic ASICs; each stage is assigned to a single GPE, and each GPE immediately pushes its outputs to its neighbor. The butterflies in each stage are computed greedily. To reduce storage and increase parallelism, Trans-SA uses run-time twiddle coefficient generation when the transform size is too large for on-chip memory, e.g. $>$256 for 2$\times$8, with the trade-off of making the problem compute-bound. On Trans-SC, the butterfly operations are distributed evenly among GPEs to compute a stage in parallel. LCPs assign inputs and collect outputs from GPEs. All cores synchronize after each stage. For Trans-PS, the same scheduling is used, and partial results are stored in the L1 SPM.

4.7.3 Sparse Matrix Multiplication

SpMM. SpMM is a memory-bound kernel with low FLOPS that decrease with increasing sparsity, e.g. $\sim 2N^3r_M^2$, for uniform-random $N\times N$ matrices with density r_M. Furthermore, sparse storage formats lead to indirection and thus irregular memory accesses [117, 154]. We implement SpMM in Trans-SC using a prior outer product approach [154]. 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 [80]. Both the columns of A and rows of B are packed in memory. The computation is equally split across the tiles.

SpMV. SpMV, similar to SpMM, is bandwidth-bound and produces low FLOPS ($\sim 2N^2r_Mr_v$ for a uniformly random $N\times N$ matrix with density r_M, and vector with density r_v). 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 [80]. 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.

Finally, for both SpMM and SpMV, we use dynamic scheduling for work distribution to the GPEs (Section 4.5.1), in order to exploit the amenability of sparse workloads to SPMD architectures [154].

4.8 Experimental Methodology

This section describes the methodology used to derive performance, power and area estimates for Transmuter. Table 4.4 shows the parameters used for modeling Transmuter. We compare Transmuter with a high-end Intel Core i7 CPU and NVIDIA Tesla V100 GPU running optimized commercial libraries. The baseline specifications and libraries are listed in Table 4.5. For fair comparisons, we evaluate two different Transmuter designs, namely **TransX1** and **TransX8**, that are each comparable in area to the CPU and GPU, respectively. TransX1 has a single 64×64 Transmuter cluster and TransX8 employs 8 such clusters. Both designs have one HBM2 stack/cluster to provide sufficient bandwidth and saturate all GPEs in the cluster.

4.8.1 Performance Models

We used the gem5 simulator [22, 21] to model the Transmuter hardware. We modeled the timing for GPEs and LCPs after an in-order Arm Cortex-M4F, and cache and crossbar latencies based on a prior chip prototype that uses SSN crossbars [157, 158]. Data transfer/set-up times are excluded for all platforms. Throughput is reported in FLOPS/s and only accounts for useful (algorithmic) FLOPS.

The resource requirement for running simulations using this detailed 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 streamed through these engines. This allows us to simulate systems up to one 64×64 cluster. On average, across the evaluated kernels, the trace-driven model is pessimistic to 4.5% of the execution-driven model. For a multi-cluster system, we use analytical models from gem5-derived bandwidth and throughput scaling data (Section 4.9.2).

We implemented each kernel in C++ and hand-optimized it for each Transmuter configuration using the intrinsics discussed in Section 4.6. Compilation was done using an Arm GNU compiler with the -O2 flag. All experiments used single-precision FP arithmetic.

4.8.2 Power and Area Models

We designed RTL models for Transmuter hardware blocks and synthesized them. The GPEs and LCPs are modeled as Arm Cortex-M4F cores. For the R-XBar, we use the SSN design proposed in [182], augmented with an XCU. The R-DCaches are cache modules enhanced with SPM and FIFO control logic.

The crossbar and core power models are based on RTL synthesis reports and the Arm Cortex-M4F specification document. The R-XBar power model is calibrated against the data reported in [182]. For the caches and synchronization SPM, we used CACTI 7.0 [143] to estimate the dynamic energy and leakage power. We further

Table 4.4: Microarchitectural parameters of Transmuter gem5 model.

Module	**Microarchitectural Parameters**
GPE/LCP	1-issue, 4-stage, in-order (MinorCPU) core @ 1.0 GHz, tournament branch predictor, FUs: 2 integer (3 cycles), 1 integer multiply (3 cycles), 1 integer divide (9 cycles, non-pipelined), 1 FP (3 cycles), 1 LS (1 cycle)
Work/Status Queue	4 B, 4-entry FIFO buffer between each GPE and LCP within a tile, blocks loads if empty and stores if full
R-DCache (per bank)	CACHE: 4 kB, 4-way set-associative, 1-ported, non-coherent cache with 8 MSHRs and 64 B block size, stride prefetcher of degree 2, word-granular (L1) / cacheline-granular (L2) SPM: 4 kB, 1-ported, physically-addressed, word-granular FIFO+SPM: 4 kB, 1-ported, physically-addressed, 32-bit head and tail pointer registers
R-XBar	$N_{\text{src}} \times N_{\text{dst}}$ non-coherent crossbar with 1-cycle response ARBITRATE: 1-cycle arbitration latency, 0 to $(N_{\text{src}} - 1)$ serialization latency depending upon number of conflicts TRANSPARENT: no arbitration, direct access ROTATE: switch port config. at programmable intervals Width: 32 address + 32 (L1) / 128 (L2) data bits
GPE/LCP ICache	4 kB, 4-way set-associative, 1-ported, non-coherent cache with 8 MSHRs and 64 B block size
Sync. SPM	4 kB, 1-ported, physically-addressed scratchpad
Main Memory	1 HBM2 stack: 16 64-bit pseudo-channels, each @ 8000 MB/s, 80-150 ns average access latency

verified our power estimate for SpMM on Transmuter against a prior SpMM ASIC prototype [157] and obtained a pessimistic deviation of 17% after accounting for the architectural differences. Finally, the area model uses estimates from synthesized Transmuter blocks.

We note that this work considers only the chip power on all platforms, for fair comparisons. We used standard profiling tools for the CPU and GPU, namely nvprof and RAPL. For the GPU, we estimated the HBM power based on per-access energy [151] and measured memory bandwidth, and subtracted it out. The power is scaled for iso-technology comparisons using quadratic scaling.

4.9 Evaluation

We evaluate the Trans-SC, Trans-PS and Trans-SA configurations on the kernels in Section 4.7. We then compare the best-performing Transmuter to the CPU and GPU, and deep-dive into the evaluation of an application that exercises rapid

Table 4.5: Specifications of baseline platforms and libraries evaluated.

Platform	Specifications	Library Name and Version
CPU	Intel i7-6700K, 4 cores/8 threads at 4.0-4.2 GHz, 16 GB DDR3 memory @ 34.1 GB/s, AVX2, SSE4.2, 122 mm^2 (14 nm)	MKL 2018.3.222 (GeMM/GeMV/SpMM/SpMV), DNNL 1.1.0 (Conv), FFTW 3.0 (FFT)
GPU	NVIDIA Tesla V100, 5120 CUDA cores at 1.25 GHz, 16 GB HBM2 memory at 900 GB/s, 815 mm^2 (12 nm)	cuBLAS v10 (GeMM/GeMV), cuDNN v7.6.5 (Conv), cuFFT v10.0 (FFT), CUSP v0.5.1 (SpMM), cuSPARSE v8.0 (SpMV)

reconfiguration. Lastly, we show comparisons with prior platforms and power/area analysis.

4.9.1 Performance with Different Configurations

Figure 4.6 presents relative comparisons between Trans-SC, Trans-PS and Trans-SA in terms of performance. This analysis was done on a small 2×8 system to stress the hardware. The results show that the best performing Transmuter configuration is kernel-dependent, and in certain cases also input-dependent. Figure 4.7 shows the cycle breakdowns and the work imbalance across GPEs.

For GeMM, Trans-SC achieves high L1 hit rates (>99%), as efficient blocking leads to good data reuse. Trans-PS suffers from capacity misses due to lack of sharing, noted from the large fraction of L2 misses. Further, Trans-SC performs consistently better than Trans-SA, as it does not incur the overhead of manually fetching data into the L1 SPM. For *GeMV*, Trans-SC and Trans-PS behave the same as GeMM. However, Trans-SA experiences thrashing (increasing with matrix size) in the private L2. For *Conv*, as with GeMM/GeMV, Trans-SC performs the best due to a regular access pattern with sufficient filter and input reuse. Across these kernels, stride prefetching in Trans-SC is sufficient to capture the regular access patterns.

For FFT, Trans-SA achieves significantly higher throughput because it benefits from the streaming inputs and exploits better data reuse, evidenced by ∼10× less memory bandwidth usage compared to Trans-SC/Trans-PS. Inter-GPE synchronization and coherence handling at the end of each stage limit the performance for Trans-SC/Trans-PS. In addition, the control flow in the non-systolic code is branchy and contributes to expensive ICache misses. Trans-SA performs better for sizes <512 compared to other sizes, as the twiddle coefficients are loaded from on-chip rather than being computed.

For SpMM, the multiply phase of outer product is better suited to Trans-SC as

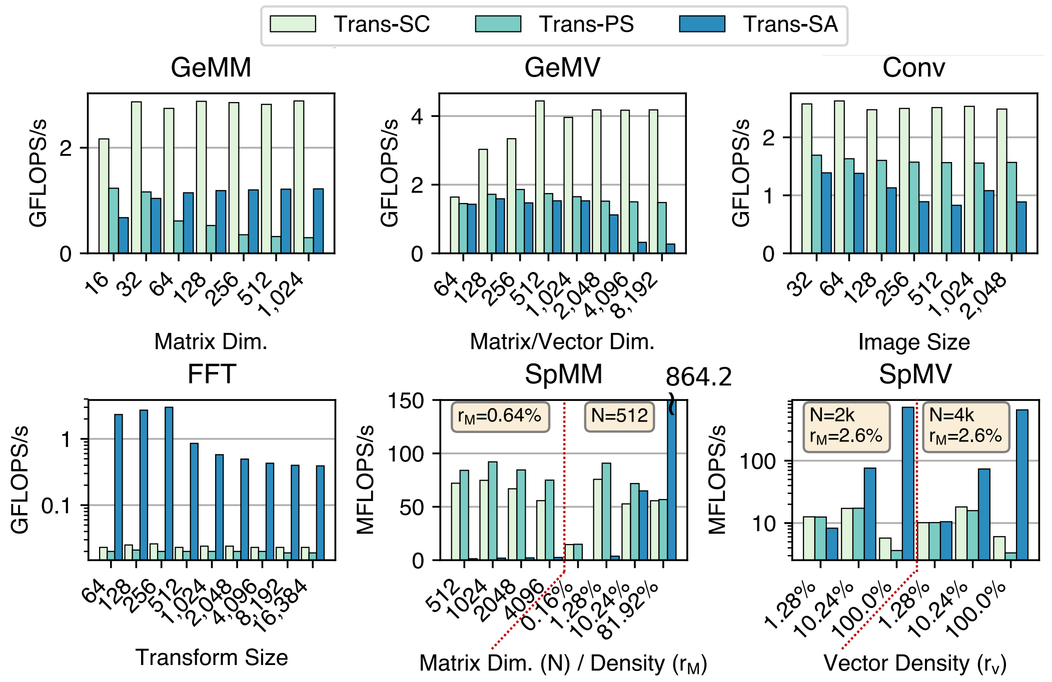


Figure 4.6: Performance of 2×8 Trans-SC, Trans-PS and Trans-SA configurations across different inputs for the kernels in Section 4.7. All matrix operations are performed on square matrices without loss of generality. Conv uses 3×3 filters, 2 input/output channels, and a batch size of 2.

the second input matrix rows are shared. The merge phase is amenable to Trans-PS since the private SPMs overcome the high thrashing that Trans-SC experiences while merging multiple disjoint lists. Trans-SA dominates for densities >∼11%, however it performs poorly in comparison to outer product for highly-sparse matrices. Although ~50% of the time is spent on compute operations (Figure 4.7), most of these are wasted on fetched data that are discarded after failed index matches. For SpMV, performance depends on the input matrix size, dimensions, as well as the vector density. Notably, Trans-SA benefits through the spatial dataflow for SpMV but not for SpMM, because the SpMV implementation treats the vector as dense, and thus can stream-in the vector elements efficiently into the GPE arrays. At sufficiently high vector sparsities, outer product on Trans-SC/Trans-PS outperforms Trans-SA by avoiding fetches of zero-elements. For higher densities, they suffer from the overhead of performing mergesort that involves frequent GPE-LCP synchronization, and serialization at LCP `0`.

Takeaways. Demand-driven dataflow with shared caching outperforms other configurations for GeMM, GeMV and Conv due to sufficient data sharing and reuse. Streaming kernels such as FFT and SpMV (with dense vectors) are amenable to spa-

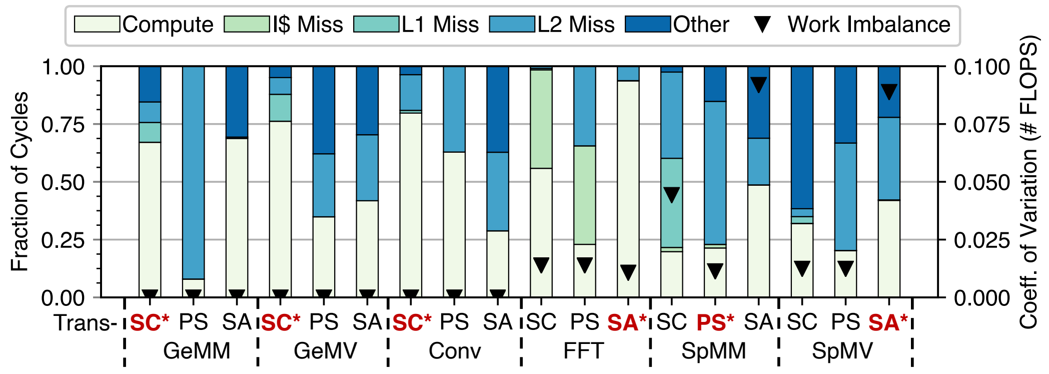


Figure 4.7: Cycle breakdown for the kernels in Section. 4.7. * (red) indicates the best-performing configuration. "Other" comprises of stalls due to synchronization and bank conflicts. ▼: work imbalance across GPEs (σ/μ of # FLOPS). Inputs are: 1k (GeMM), 8k (GeMV), 2k (Conv), 16k (FFT), 4096, 0.64% (SpMM), 4k, 2.6%, dense vector (SpMV).

tial dataflow. SpMM and high-sparsity SpMV show amenability to private scratchpad or shared cache depending on the input size and sparsity, with the systolic mode out-performing only for very high densities.

4.9.2 Throughput and Bandwidth Analysis

We investigate here the impact of scaling the number of tiles (N_T) and GPEs per tile (N_G) for an $N_T \times N_G$ Transmuter. Figure 4.8 illustrates the scaling of Transmuter for GeMM, GeMV and SpMM. GeMM shows near-linear performance scaling with the GPE-count. The bandwidth utilization, however, does not follow the same trend as it is dependent on the data access pattern at the shared L2 R-DCache that influences the L2 hit-rate. GeMV exhibits increased bank conflicts in the L1 shared cache upon scaling up N_G, e.g. from 32×32 to 32×64. Thus, the performance scaling shows diminishing returns with increasing N_G, but scales well with increasing N_T. SpMM performance scales well until the bandwidth utilization is close to peak, at which point bank conflicts at the HBM controllers restrict further gains. SpMV follows the trend of GeMV, while FFT and Conv, show near-linear scaling with increasing system size (not shown).

We also discuss some takeaways from our cache bandwidth analysis for the best-performing Transmuter configuration. GeMM exhibits a high L1 utilization (20.4%) but low L2 utilization (2.7%), as most of the accesses are filtered by the L1. In contrast, SpMM and SpMV in Trans-PS and Trans-SA modes, respectively, have higher L2 utilizations of 68.5-90.5%. The linear algebra kernels show a relatively balanced utilization across the banks, with the coefficient of variation ranging from 0-10.1%. In contrast, both FFT and Conv have a skewed utilization, due to the layout of twiddle coefficients in the SPM banks for FFT, and the small filter size for Conv.

4.9.3 Design Space Exploration

We performed a design space exploration with the mapped kernels to select R-DCache sizes for Transmuter. Sizes of 4 kB per bank for both L1 and L2 show the best energy-efficiency for all kernels except SpMV. SpMV in Trans-SA benefits from a larger L2 private cache that lowers the number of evictions from fetching discrete packed matrix rows (recall that in Trans-SA, all GPEs in a tile access the same L2 bank). Other kernels achieve slim speedups with larger cache capacities. The dense kernels already exhibit good hit rates due to blocking and prefetching in Trans-SC. SpMM is bottlenecked by cold misses due to low reuse. FFT has a 3.0× speedup with 64 kB L1/L2, compared to 4 kB L1/L2, as the number of coefficients stored on-chip scales with L1 size. But, this is outweighed by a 6.4× increase in power. Other parameters such as work and status queue depths were chosen to be sufficiently large such that the GPEs are never idled waiting on the LCP.

4.9.4 Performance with Varying Control Divergence and Data Reuse

In Section 4.2, we characterized some fundamental kernels based on their *control divergence*, *data reuse* and *arithmetic intensity*. We now build an intuition around the architectural advantages of Transmuter over a GPU for applications with notable contrast in these characteristics. We implement a parallel microbenchmark on Transmuter and the GPU that allows independent tuning of the divergence and reuse. Figure 4.9 (left) illustrates this application. The reuse (R) is controlled by the size of the coefficient array, while divergence (D) scales with the number of bins, since threads processing each input element apply functions unique to a bin.

While this is a synthetic application, it is representative of real-world algorithms that perform image compression using quantization. We execute this microbenchmark with a batch of 1,000 32×32 images on a 4×16 Transmuter design, and compare it with the GPU running 64 threads (2 warps, inputs in shared memory) to ensure fairness. Figure. 4.9 (right) presents two key observations:

- The speedup of Transmuter roughly doubles as the number of divergent paths double. This is because threads executing different basic blocks get serialized

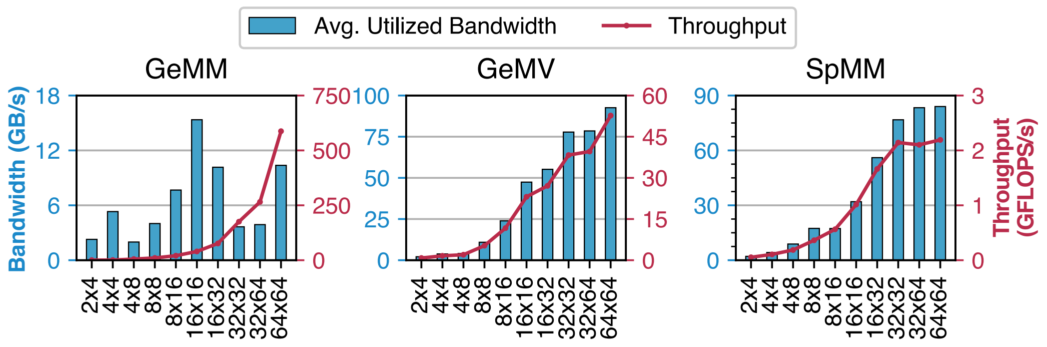


Figure 4.8: Effect of scaling tiles and GPEs per tile on performance and memory bandwidth for GeMM (Trans-SC), GeMV (Trans-SC) and SpMM (Trans-PS). Inputs are: 1k (GeMM), 8k (GeMV), 4096, 0.64% (SpMM).

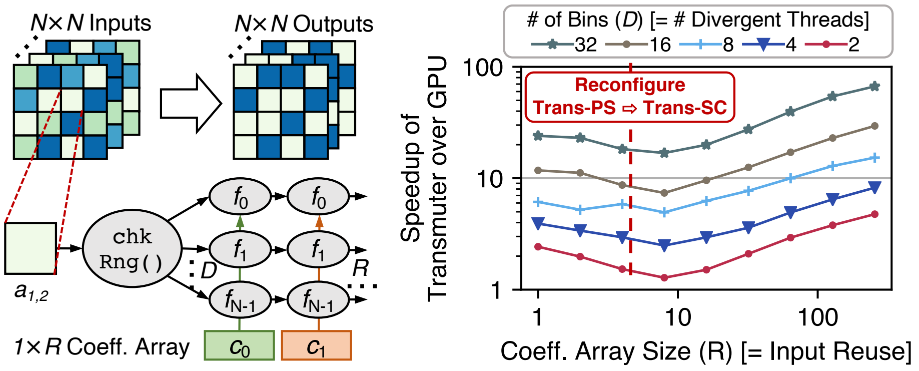


Figure 4.9: *Left:* A synthetic parallel application that launches threads to process $N \times N$ matrices. Each thread (i) reads the input value and bins it into one of D bins, (ii) applies R instances of function f_d unique to bin d and writes the result. Each element of a *coefficient* array feeds into f_d. Thus, the input is reused R times and the degree of divergence scales with D. *Right:* Speedup of Transmuter with a uniform-random matrix (# GPEs = # GPU threads = 64). Transmuter reconfigures from Trans-PS to Trans-SC beyond $R = 4$.

in the SIMT model (as they are in the same warp), whereas they can execute parallel in SPMD.

- Transmuter has the inherent flexibility to reconfigure based on the input size. In this example, Trans-PS is the best-performing until $R = 4$. Beyond that, switching to Trans-SC enables better performance – up to 7.4× over Trans-PS – as the benefit of sharing the coefficient array elements across the GPEs in Trans-SC outweighs its higher cache access latency.

Takeaways. The SPMD paradigm in Transmuter naturally lends itself well to kernels exhibiting large control divergence, and its ability to reconfigure dynamically allows it to perform well for very low- and high-reuse, and by extension mixed-reuse, workloads.

4.9.5 Comparison with the CPU and GPU

We now compare the best-performing Transmuter configuration with the CPU and GPU running optimized commercial libraries (Table 4.5). The throughput and energy-efficiency gains of Transmuter for each kernel in Section 4.7 are presented in Figure 4.10. We compare TransX1 to the CPU and TransX8 to the GPU, as discussed in Section 4.8.

Compute-Bound Kernels (GeMM, Conv, FFT). TransX1 harnesses high data-level parallelism, and thus achieves performance improvements of 1.2-2.5× over the CPU, despite clocking at $\frac{1}{4}$th the speed of the deeply-pipelined CPU cores. The true benefit of Transmuter's simple cores and efficient crossbars appear in the form of energy-efficiency gains, ranging from 3.6-16.3×, which is owed largely to the high power consumption of the bulky out-of-order CPU cores. Over the GPU, TransX8 gets performance gains of 1.3-2.6× and efficiency improvements of 0.8-4.4× with an efficient implementation on Trans-SC for GeMM and Conv. The ∼20% energy-efficiency loss for GeMM is explained by the amenability of GeMM to a SIMT paradigm; although the performance is similar between SIMT and SPMD, SPMD incurs slightly larger energy costs associated with higher control overhead over SIMT. On FFT, Transmuter sustains consistent performance scaling using the spatial dataflow of Trans-SA, with each tile operating on an independent input stream, thus leading to minimum conflicts. The gap between throughput gain (4.0×) and energy-efficiency gain (1.3×) over the GPU is explained by the cuFFT algorithm that is more efficient for batched FFTs.

Memory-Bound Kernels (GeMV, SpMM, SpMV). TransX1 on GeMV achieves 2.4× better throughput over the CPU, with the CPU becoming severely DRAM-bound (>98% bandwidth utilization) for input dimensions beyond 1,024. The 14.2× energy-efficiency gain of TransX1 stems from tuning down the number of active GPEs to curtail bandwidth-starvation, thus saving power.

On SpMM and SpMV, the performance of Transmuter is highly sensitive to the densities and sizes of the inputs, with improvements ranging from 4.4-110.8× over the CPU and 5.9-37.7× over the GPU. With SpMM, execution in Trans-PS enables overcomes the CPU's limitation of an inflexible cache hierarchy, as well as harnesses high MLP across the sea of GPEs. While Transmuter is memory-bottlenecked for SpMM, SpMV is bounded by the scheduling granularity of packing algorithm deployed on

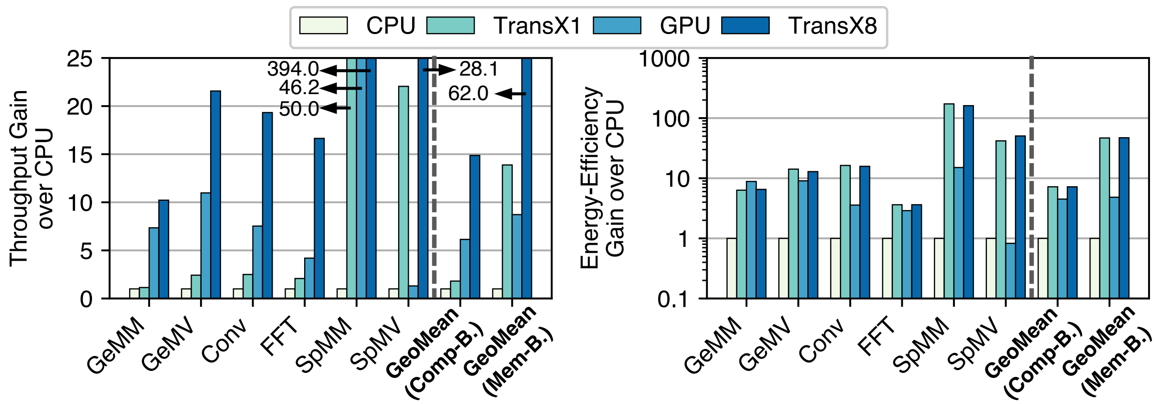


Figure 4.10: Throughput (left) and energy-efficiency (right) improvements of Transmuter over the CPU and GPU. Data is averaged across the inputs: 256-1k (GeMM), 2k-8k (GeMV), 512-2k (Conv), 4k-16k (FFT), 1k-4k, 0.64% (SpMM), and 2k-4k, 2.6% (r_M), 10.2%-100% (r_v) (SpMV). Geometric mean improvements for the compute-bound and memory-bound kernels are shown separately.

Trans-SA. Despite that, for SpMV, TransX1 outperforms both the CPU as well as the GPU that has 7.2× greater available bandwidth. In case of the GPU, while there are sufficient threads to saturate the SMs, the thread divergence in the SIMT model is the bottleneck. The GPU achieves just 0.6% and 0.002% of its peak performance, respectively for SpMM and SpMV, impaired by memory and synchronization stalls. In comparison, SPMD on Transmuter reduces synchronization, resulting in 21-42% time spent on useful computation (Figure 4.7). For SpMM, the outer product implementation demonstrates ASIC-level performance gains of 5.9-11.6× [154] over the GPU, by minimizing off-chip traffic and exploiting the asynchronicity between GPEs. As with GeMV, disabling bandwidth-starved resources contributes to the energy-efficiency gains.

Effect of Iso-CPU Bandwidth. TransX1 uses one HBM stack that provides 125 GB/s peak bandwidth, about 3.6× greater than the DDR3 bandwidth to the CPU. If given the bandwidth of the DDR3 memory, TransX1 still achieves performance gains averaging 17.4× and 6.3× for SpMM and SpMV, respectively. For GeMV, TransX1 remains within a modest 6-8% of the CPU with this low bandwidth.

4.9.6 End-to-End Workload Analysis

We estimate speedups of Transmuter over the CPU and GPU for the end-to-end workloads in Figure 2.3, and report the results in Table 4.6. File I/O and cross-platform data transfer (e.g. `memcpy` from CPU to GPU) times are excluded for all platforms. Overall, Transmuter achieves speedups averaging 3.1× over the CPU and 3.2× over the GPU.

Table 4.6: Estimated speedups for the end-to-end workloads in Figures. 2.3.

Speedup	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
TransX1 vs. CPU	4.1×	1.1×	2.2×	6.2×	1.7×	3.5×	2.7×	2.4×	3.1×	2.2×
TransX8 vs. GPU	3.5×	3.8×	2.1×	7.2×	1.6×	2.8×	2.3×	2.5×	3.0×	2.8×

Next, we elucidate how rapid reconfiguration enables efficient execution of workloads that involve mixed sparse-dense computation in an inner loop. We make a case study on a representative mixed-data application, namelySinkhorn, that performs iterative computation to determine the similarity between documents [114, 175]. Sinkhorn computation typically involves large, sparse matrices in conjunction with dense matrices. We implement the algorithm described in [42].The inner loop has two major kernels: a GeMM operation masked by a sparse weight matrix, i.e. M-GeMM, and a DMSpM.

The mapping on Transmuter is shown in Figure 4.11. M-GeMM uses a variation of blocked-GeMM, wherein only rows/columns of the dense matrices that generate an element with indices corresponding to non-zeros in the weight matrix are fetched and multiplied. DMSpM uses a simplified outer product algorithm similar to SpMM (Section 4.7.3) that splits the kernel into DMSpM-Multiply and DMSpM-Merge.

We show the analysis of Sinkhorn on different Transmuter sizes in Figure 4.12. As observed, M-GeMM and DMSpM-Multiply exhibit the best performance in Trans-SC configuration, due to good data reuse across GPEs. In contrast, DMSpM-Merge has optimal performance on Trans-PS, exhibiting a 84.9-98.3% speedup (not shown in figure) over Trans-SC. Therefore, the optimal Sinkhorn mapping involves two reconfigurations per iteration: Trans-SC $\rightarrow$ Trans-PS before the start of DMSpM-Merge, and Trans-PS $\rightarrow$ Trans-SC at the end of it, for the next M-GeMM iteration. Recall from Section 4.5.7 that the reconfiguration time is $\sim$10 cycles, and hence does not perceptibly impact the performance or energy. Cache flushing (net 0.2% of the total execution time) is required for M-GeMM but not DMSpM, as DMSpM uses a streaming algorithm. Overall, dynamic reconfiguration results in 47.2% and 96.1% better performance and energy-delay product (EDP), respectively, over Trans-SC-only for the 4×16 Transmuter. A heterogeneous solution is also compared against, where M-GeMM is done on the CPU and DMSpM on the GPU, but this implementation is bottlenecked by CPU $\rightarrow$ GPU data transfers. As derived from Figure 4.12, the 4×16 Transmuter achieves 38.8× and 144.4× lower EDP than the GPU and heterogeneous solutions, respectively.

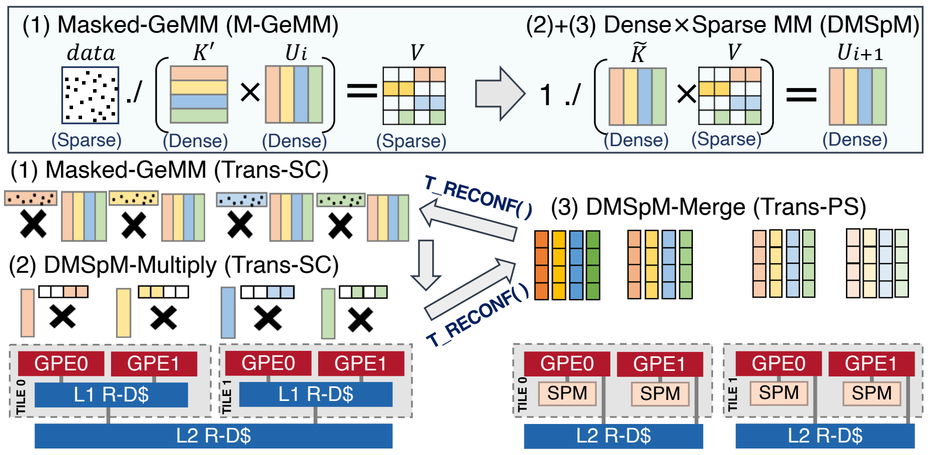


Figure 4.11: Mapping of a multi-kernel, mixed data application, Sinkhorn, on Transmuter. Computation iterates between M-GeMM and DMSpM, with Trans-SC ↔ Trans-PS reconfiguration before and after DMSpM-Merge. DMSpM-Merge benefits from the private SPMs in Trans-PS, since each GPE works on multiple disjoint lists.

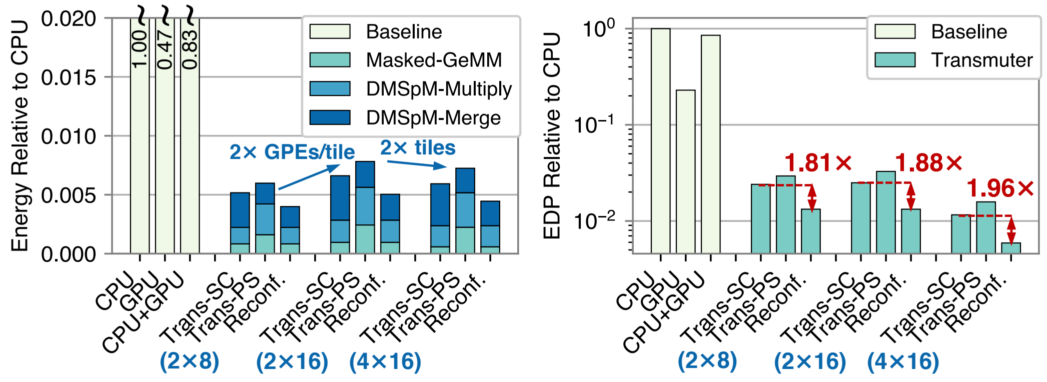


Figure 4.12: Per inner-loop iteration energy (left) and EDP (right) comparing Trans-SC, Trans-PS and Reconf. (Trans-SC ↔ Trans-PS) for Sinkhorn normalized to CPU. Input matrix dimensions and densities are — *query*: (8k×1), 1%, *data*: (8k×1k), 1%, *M*: (8k×8k), 99%.

4.9.7 Comparison with Other Platforms

Table 4.7 shows the estimated energy-efficiency improvements of Transmuter over recent FPGA, CGRA, and ASIC implementations. The efficiencies reported in prior work are scaled quadratically for iso-technology comparisons with Transmuter. Overall, Transmuter achieves average efficiency gains of 3.4× and 2.0× over FPGAs and CGRAs, respectively, and is within 9.3× (maximum 32.1×) of state-of-the-art ASICs for the evaluated kernels.

Table 4.7: Energy-efficiency improvements (black) and deteriorations (red) of Transmuter over prior FPGAs, CGRAs and ASICs.

Platform	GeMM	GeMV	Conv	FFT	SpMM	SpMV
FPGA	2.7× [69]	8.1× [115][2]	2.7× [220]	2.2× [69]	3.6× [70]	3.0× [54]
CGRA	2.2× [171]	3.0× [43]	1.2× [43]	1.0× [99][3]	1.9× [43]	2.9× [43]
ASIC	(32.1×) [161]	(10.5×) [180]	(13.8×) [204] (7.6×) [180]	(18.1×) [162] (17.0×) [61]	(3.0×) [157] (4.1×) [222]	(3.9×) [154]

[2]Performance/bandwidth used as power is N/A. [3]Estimated for floating-point based on [200].

Table 4.8: Power and area of a 64×64 Transmuter cluster in 14 nm.

Module	**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
All 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 mm^2**

4.9.8 Power and Area

Table 4.8 details the power consumption and area footprint of a 64×64 Transmuter cluster in 14 nm. Most of power is consumed by the network and memory, i.e. L1 R-XBars, R-DCaches and ICaches, while the cores only consume 20.8%. This is consistent with a growing awareness that the cost of computing has become cheaper than the cost to move data, even on-chip [82]. GPEs and L1 R-XBars, the most frequently switched modules, consume 84.2% of the total dynamic power. The estimated power for a single Transmuter cluster is 13.3 W in 14 nm with an area footprint within 1.7% of the CPU's area. The estimated worst-case reconfiguration overhead is 74.9 nJ.

4.10 Conclusion

This work tackled the important challenge of bridging the flexibility-efficiency gap with Transmuter. Transmuter consists of simple processors connected to a network of reconfigurable caches and crossbars. This fabric supports fast reconfiguration of the memory type, resource sharing and dataflow, thus tailoring Transmuter to the nature of the workload. We also presented a software stack comprised of drop-in replacements for standard Python libraries. We demonstrated Transmuter's performance and efficiency on a suite of fundamental kernels, as well as mixed data-based multi-kernel applications. Our evaluation showed average energy-efficiency improvements of 46.8× (9.8×) over the CPU (GPU) for memory-bound kernels and 7.2× (1.6×) for compute-bound kernels. In comparison to state-of-the-art ASICs that implement the same kernels, Transmuter achieves average energy-efficiencies within 9.3×.

CHAPTER V

Runtime Control for Accelerated Sparse Linear Algebra on Reconfigurable Hardware

Dynamic adaptation is a post-silicon optimization technique that adapts the hardware to workload phases. However, current adaptive approaches are oblivious to implicit phases that arise from operating on irregular data, such as sparse linear algebra operations. Implicit phases are short-lived and do not exhibit consistent behavior throughout execution. This calls for a high-accuracy, low overhead runtime mechanism for adaptation at a fine granularity. Moreover, adopting such techniques for reconfigurable manycore hardware, such as coarse-grained reconfigurable architectures (CGRAs), adds complexity due to synchronization and resource contention.

This chapter describes the proposal of a lightweight machine learning-based adaptive framework called SparseAdapt. It enables low-overhead control of configuration parameters to tailor the hardware to both implicit (data-driven) and explicit (code-driven) phase changes. SparseAdapt is implemented within the runtime of a recently-proposed CGRA called Transmuter, which has been shown to deliver high performance for irregular sparse operations. SparseAdapt can adapt configuration parameters such as resource sharing, cache capacities, prefetcher aggressiveness, and Dynamic Voltage Frequency Scaling (DVFS). Moreover, it can operate under the constraints of either (i) high energy-efficiency, i.e. maximal GFLOPS/s/W, or (ii) high power-performance, i.e. maximal (GFLOPS/s)^3/W.

We evaluate SparseAdapt with sparse matrix-matrix and matrix-vector multiplication (SpMM and SpMV) routines across a suite of uniform random, power-law and real-world matrices. SparseAdapt achieves similar performance on SpMM as the largest static configuration, with 5.3× better energy-efficiency. Furthermore, on both performance and efficiency, SparseAdapt is at most within 13% of an Oracle that adapts the configuration of each phase with global knowledge of the entire program execution. Finally, SparseAdapt is able to outperform the state-of-the-art approach for runtime reconfiguration by up to 2.9× in terms of energy-efficiency.

The work presented in this chapter is accepted to appear at MICRO 2021 [153].

5.1 Introduction

Sparse linear algebra operations are key components of a plethora of modern applications, from graph analytics to scientific computing and machine learning [27, 71, 172, 24, 184, 219, 202, 15, 18, 213, 218, 215, 203]. Many graph analytics algorithms, such as breadth-first search, shortest path, etc. can be represented as sparse linear algebra operations, as encapsulated by the GraphBLAS framework [104]. Two important kernels belonging to BLAS levels 2 and 3, respectively, are sparse matrix – sparse matrix multiplication (SpMM) and sparse matrix – sparse vector multiplication (SpMV). These are highly inefficient on traditional computing platforms such as CPUs and GPUs, as they are heavily memory-bounded and thus bottlenecked on data movement rather than compute [154].

Recent work have led to a myriad of proposals on optimizing sparse computation through fixed-function accelerator designs [221, 188, 13, 189, 154]. While these demonstrate energy-efficiency improvements of the order of 100 of times over a GPU, there is an important trade-off in terms of loss of flexibility, i.e. such designs are only applicable to a few kernels. One class of solutions that propose to close this gap between flexibility and efficiency are CGRAs [155, 43, 171, 99, 176, 198, 146, 78]. CGRAs incorporate word-granular operations to overcome the energy inefficiency of FPGAs, while retaining programmability. They allow for hardware reconfiguration at the granularity of the PE array, network fabric, or the memory subsystem. A key challenge lies with efficient mapping of sparse kernels onto CGRA hardware. Typically, the CGRA configuration is determined at compile-time by extracting a DFG from the application code, mapping it onto the PEs, and rewiring the on-chip memory and interconnect. There are two drawbacks of compile-time mapping.

- Real-world sparse datasets are seldom uniform, e.g. in graph analytics workloads, the input graphs are often clustered and follow power-law distributions [46, 116]. Thus, determining the best configuration requires intimate knowledge of the input, beyond its shape and NNZs, which is not feasible at compile-time.

- Compile-time optimizations fail if the dataset evolves over time, since no accurate estimations can be made for the distribution of non-zeros even with data pre-processing. This is common in the world of social networks, for instance, where connections between users form and break in real-time, which translates to dynamic changes in the underlying data structures [38, 59].

In order to tackle these challenges, we propose an adaptive runtime framework, SparseAdapt, that reconfigures a CGRA to adapt to evolving phases in sparse computation kernels. SparseAdapt establishes a feedback loop between the software (running on a host) and the CGRA hardware, that enables *fine-grained* introspection, i.e. the hardware exposes performance counters to the runtime software, which periodically collects this data to determine *when* and *how* to reconfigure the hardware, thus catering to transitions in both implicit (data-driven) and explicit (code-driven) phase changes. SparseAdapt is oblivious to the underlying program binary, and thus requires no programmer intervention. We implement SparseAdapt as an extension to the runtime of the reconfigurable hardware proposed in Chapter IV called Transmuter, however it is applicable to any CGRA system that exposes similar hooks from the hardware to the runtime. SparseAdapt is designed to be deployed on both cloud and edge scenarios, which have drastically different power-performance requirements. As such, SparseAdapt can operate in one of two optimization modes, an Energy-Efficient mode that optimizes for giga-FLOPS executed per second per Watt (GFLOPS/s/W), and a Power-Performance mode that optimizes for $(\text{GFLOPS/s})^3/\text{W}$.

In summary, this work makes the following contributions:

- Identifies the existence of hardware reconfiguration opportunities associated with phase transitions during execution for sparse algebra routines, both due to (i) change in code, i.e. explicit phases, and (ii) evolving sparsity patterns in the dataset, i.e. implicit phases.

- Proposes a framework called SparseAdapt that uses low-cost hardware performance monitoring to adapt to phase changes and reconfigure the underlying hardware configuration. SparseAdapt is integrated with the runtime of the Transmuter [155] hardware.

- Proposes a predictive model based on an ensemble of decision trees, and a heuristic-based, reconfiguration cost-aware policy that allows for hardware reconfiguration at fine granularities (~ 500–$5k$ floating-point instructions), while reducing the frequency of reconfiguration penalties.

- Demonstrates improvements in energy-efficiency and performance metrics on SpMM and SpMV routines across a suite of random, power-law and real-world matrices, and compares it to various oracle adaptive control mechanisms.

In terms of evaluation, SparseAdapt achieves similar performance as the largest static configuration with 5.3× better energy-efficiency on SpMM. When compared to an Oracle scheme that exploits full knowledge of the entire program duration, SparseAdapt achieves within 13% performance and 5% efficiency in the Power-Performance optimization mode. Finally, in comparison to a prior adaptive control scheme [56], SparseAdapt achieves average gains of up to 2.9× and 2.8× in terms of energy-efficiency and performance, respectively.

5.2 Motivation and Related Work

This section discusses the challenges and opportunities associated with dynamic reconfiguration, and prior work.

5.2.1 Existence of Implicit and Explicit Phases

Phases arise naturally during the execution of any non-trivial code. The obvious phase changes occur due to control flow from the program structure. We term these as *explicit* phase transitions. Explicit phase changes are relatively simple to detect, even at compile-time. However, computation involving sparse datasets introduce another type of phase change owing to the irregularity in the data. Such phase changes may be correlated with instantaneous properties of the data, such as spatial locality. We define these phase transitions to be *implicit*. Real-world matrices have sparsity patterns that vary spatially across the matrix, thus introducing implicit phase changes during computation. Thus, a clever adaptation framework needs to incorporate runtime reconfigurability and cater to both implicit and explicit phase changes.

We illustrate the phenomenon of phase changes through the example of outer product (OP) based SpMM [154]. OP-SpMM decomposes the computation into two explicit phases, namely *multiply* and *merge* (Figure 5.1 – left). We demonstrate this on a synthetic matrix generated with dense columns separating eight sparse strips to motivate our work. Similar structures of alternating dense and sparse row/column clusters exist in real-world datasets from graphs, optimization and economics problems, etc. [48]. We simulate this on a tiled manycore architecture (Section 5.3) with two processing tiles. We report the energy-efficiency (GFLOPS/s/W), instantaneous clock frequency and L2 cache bank size with a dynamic reconfiguration scheme versus the best *static* hardware configuration in Figure 5.1. We first note the presence

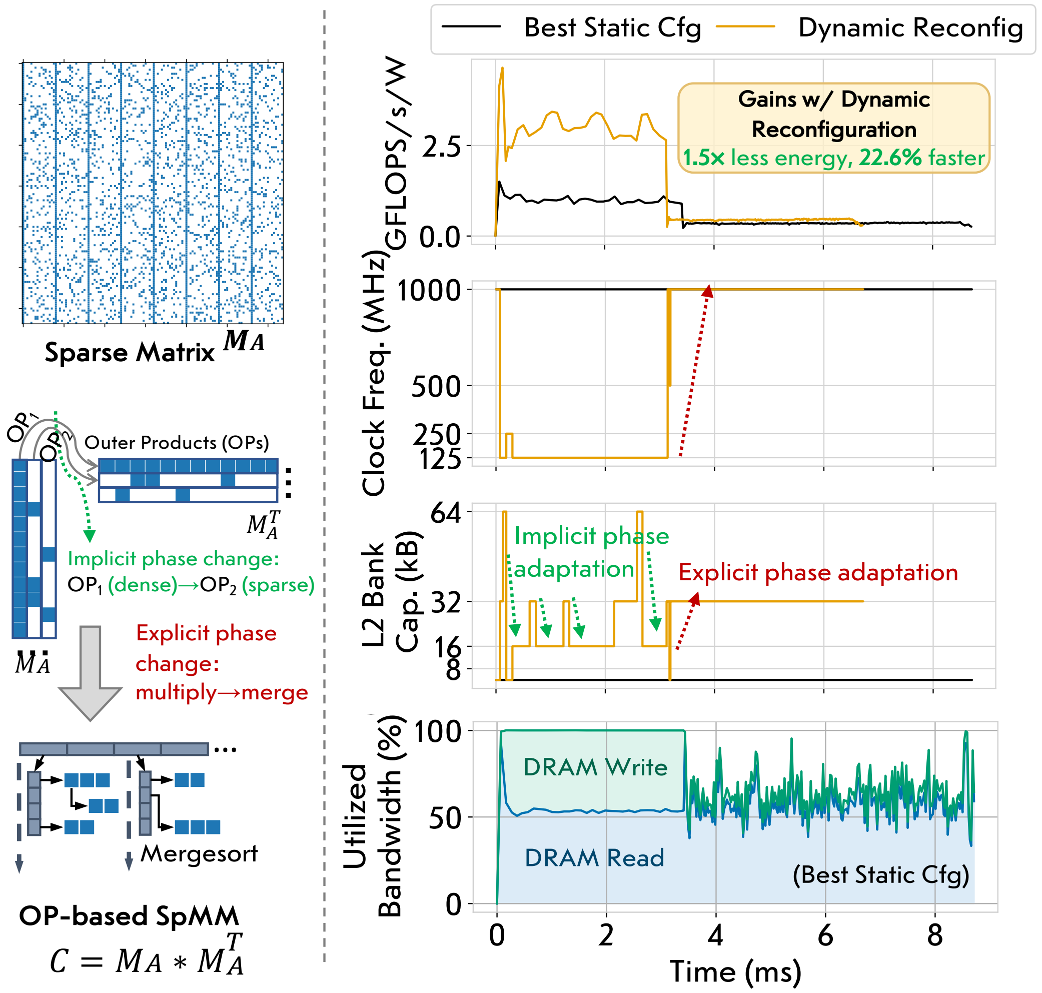


Figure 5.1: *Left.* Illustration of outer product (OP) SpMM showing implicit phase changes due to switch from dense to sparse outer products (column × row). *Right.* Execution timeline of OP-SpMM on a 128×128, 20% dense matrix and its transpose. Shown are the gains achieved using a dynamic reconfiguration scheme that adapts the hardware to these phase transitions.

of the two explicit phases, corresponding to the multiply (ending at ∼ 3.5 ms) and merge phases of OP-SpMM. The dynamic control scheme observes the off-chip bandwidth utilization to be ∼100% during multiply (Figure 5.1 – bottom), and acts by applying DVFS to lower the clock speed and balance the compute-to-memory ratio of the system. This improves the multiply phase energy-efficiency over the baseline by ∼2×.

We further note the presence of implicit phases arising from the computation of dense columns with corresponding dense rows during the first phase (Figure 5.1 – left). These implicit phases are adapted to by adjusting the dynamic L2 cache capacity, based on observation of a combination of various hardware counters (not shown in figure). Thus, exploiting both types of phase changes lead to an overall speedup of 22% and energy savings of 1.5× over the static baseline.

In summary, this simple example illustrates the opportunities associated with dynamic reconfiguration, which we seek to exploit with our proposed framework, SparseAdapt.

5.2.2 Related Work

A few prior work have explored run-time adaptation for improving the efficiency of existing hardware architectures.

5.2.2.1 Adaptation for Traditional Hardware

Dubach et al. [56, 55] propose a maximum likelihood estimation (MLE) predictive model that adapts the sizes of microarchitectural structures in a single-threaded, out-of-order processor. CHARSTAR [173] uses a multilayer perceptron (MLP) model that accounts for the clock hierarchy and topology, and proposes a mechanism that jointly optimizes for both DVFS and clock-aware power gating. Both of these work are proposed only for single-core systems. Tarsa et al. [197] propose an adaptive CPU based on Intel SkyLake that uses random forests to dynamically adjust the issue width of a clustered core. These work explore hardware adaptation for traditional workloads only have explicit phases. SparseAdapt, on the other hand, is designed for manycore hardware, and caters to workloads that have both explicit and implicit phases. Moreover, these prior techniques rely on SimPoint [166] for fast generation of offline profiling data. However, SimPoint fails when workloads have diverse implicit phases, arising from unstructured sparsity, throughout program execution. Our work instead considers end-to-end simulations, as no assumptions can be made about the nature of the input matrices.

Lukefahr et al. [133] introduce Composite Cores, that uses a linear regression based reactive online controller to switch execution between big and little μengines in a heterogeneous system. Flicker [167] is an adaptive multicore architecture designed to adapt to varying allocated power constraints. SOSA [52] is a resource manager that targets manycore systems and dynamic workloads using rule-based reinforcement learning (RL). The heavy weighted RL model require additional acceleration that the authors demonstrate on an FPGA. Sartor et al. [177] propose an MLP model coupled with a polymorphic Very Long Instruction Word (VLIW) processor that is trained at design-time and predicts at runtime. However, this framework predicts a static configuration for a kernel when the kernel is re-executed, whereas SparseAdapt adapts to the dynamic phase changes during run-time. Yukta [170] applies the structured singular value (SSV) control for EDP optimization on a multicore big.LITTLE system. Imes et al. [85, 84] propose a runtime system called POET that uses control theory and optimization techniques to minimize energy consumption while meeting soft real-time constraints, demonstrated on both big.LITTLE systems-on-chips (SoCs) and multicore server-class systems. However, these prior approaches do not explore configuration parameters associated with CGRAs, and do not explore workloads that have varying implicit phases.

5.2.2.2 Adaptation for CGRAs

CGRAs have gained traction recently as they promise to achieve near-ASIC performance with post-fabrication adaptation [128]. Earlier work have considered only compile-time CGRA reconfiguration for simplicity and ease-of-use [171, 5, 68, 36]. More recent work have alluded to the limitations with static adaptation, especially for irregular workloads like sparse linear algebra. Recent techniques have thus adopted dynamic scheduling and dataflow techniques to harness parallelism [146, 35, 208]. However, work on dynamic hardware reconfiguration that caters to both implicit and explicit phases in the workload has been comparatively lacking.

5.2.2.3 Comparison with ProfileAdapt

The technique proposed in this chapter is closely related to the work by Dubach et al. [56] (referred to as ProfileAdapt in this work) in terms of the offline training, hardware telemetry and inference methodologies on the predictive model. However, there are several key differences. ProfileAdapt executes in the following order: detection of a new phase, execution of the new phase in a profiling configuration (where each reconfigurable parameter takes its maximum value) for a certain duration, hardware telemetry, inference on the predictive model, followed by reconfiguration and execution on the predicted configuration.

Since ProfileAdapt reconfigures at coarse phase boundaries, it cannot adapt to short-lived implicit phases due to the reconfiguration cost associated with switching to the profiling configuration. In contrast, our approach removes the need for a profiling configuration and instead directly feeds back the hardware configuration parameters as inputs to the predictive model. This enables us to reconfigure at a finer granularity with low-overhead, which is necessary to adapt to the implicit phases in sparse computation. Additionally the evaluation in [56] relies on an external phase detection mechanism (e.g. SimPoint), whereas we assume that such a mechanism is not feasible for implicit phases. Our approach considers hardware reconfiguration at fixed epochs, guided by our predictive model and reconfiguration-cost aware heuristics.

5.3 Hardware Design

We provide background on the Transmuter architecture [155], followed by the configuration parameters that we explore and implement in this work. We then discuss the performance counters that we implement and the cost of reconfiguration.

5.3.1 Architectural Background

SparseAdapt is integrated with the runtime of the Transmuter [155] hardware, a detailed architectural description of which was presented in Chapter IV. This section delves into the microarchitectural and system-level design changes made to Transmuter to support dynamic reconfiguration, in addition to the configuration parameters considered.

In this work, we enhance Transmuter and implement a feedback loop between a host CPU and the Transmuter device, as shown in Figure 5.2. The host executes Python code and is responsible for offloading parallelizable kernels to Transmuter for accelerated execution. The first step associated with kernel dispatch consists of selecting a version of the code to execute on the target. This algorithmic selection is dependent on the properties of the input data, and the kernel itself. The next steps involve allocation of input and output buffers in the HBM, streaming data out, triggering Transmuter to start, waiting for it to finish *while executing runtime routines*, streaming data back in, and de-allocating memory. In this work, we assume shared physical memory between the host and Transmuter.

During execution, Transmuter sends performance counters to the host at continuous intervals (epochs). Using this information, the host makes a decision about when and how the architecture should be reconfigured. Actual reconfiguration is handled by dedicated blocks incorporated in the design.

5.3.2 Configuration Parameters

In this work, we consider seven hardware configuration parameters, namely, L1 R-DCache type, L1 R-DCache capacity, L2 R-DCache capacity, L1 sharing mode,

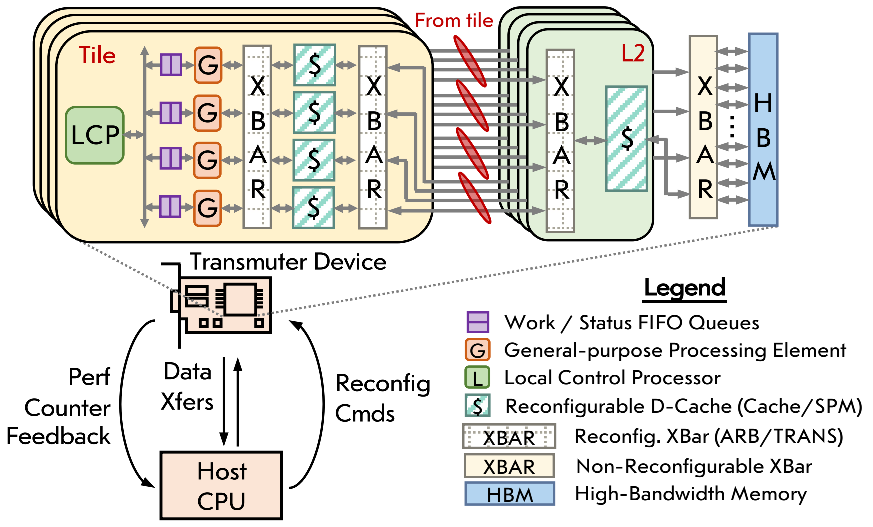


Figure 5.2: Simplified illustration of the Transmuter architecture and feedback loop between the host CPU and Transmuter.

L2 sharing mode, system clock frequency, and prefetcher aggressiveness. The values considered for these parameters are listed in Table 5.1. While much of prior work (Section 5.2.2) focuses on reconfiguration of the core microarchitecture, our work delves into reconfiguration of the interconnect, memory and DVFS.

We next discuss the microarchitectural aspects and overhead of reconfiguring each selected hardware parameter.

5.3.2.1 Dynamic Voltage-Frequency Scaling (DVFS)

We implement a simple clock divider composed of N flip-flops that enables generation of clocks with frequencies of $f/2, f/4, \ldots, f/2^N$, where f is the frequency of the system clock. For circuit-level simplicity, we consider global DVFS, i.e. the same clock feeds into each microarchitectural block, as opposed to per-core or per-block DVFS. The overheads and switching times can be kept small through the use of on-chip regulators [107], or dual-voltage rail designs [168, 138] at the expense of a small area overhead.

In our DVFS model, the target voltage is calculated based on the formula $f \propto (V_{DD} - V_{th})^2/V_{DD}$, where f is the clock frequency, V_{DD} is the supply voltage, and V_{th} is the threshold voltage. Given a target frequency f_{target}, the target supply voltage is calculated based on the following equation:

$$\frac{f}{f_{target}} = \frac{\frac{(V_{DD}-V_t)^2}{V_{DD}}}{\frac{(V_{target}-V_t)^2}{V_{target}}} = \frac{(V_{DD}-V_t)^2}{V_{DD}} \times \frac{V_{target}}{(V_{target}-V_t)^2},$$

Table 5.1: Hardware configuration parameters for the evaluated Transmuter design.

Type	**Parameter**	**Values Assumed**	**Count**
Categorical	L1 R-DCache type[2]	Cache, SPM	2
	L1 sharing mode	Shared, private	2
	L2 sharing mode	Shared, private	2
Ordinal	L1 R-DCache bank capacity[3]	4 kB $\rightarrow$ 64 kB : 2$*$	5
	L2 R-DCache bank capacity	4 kB $\rightarrow$ 64 kB : 2$*$	5
	System clock frequency	31.25 MHz $\rightarrow$ 1 GHz : 2$*$	6
	Prefetcher aggressiveness	0 (off), 4, 8	3
Total Count			$\mathbf{3,600}$

[2]Only this parameter is configured at compile-time. [3]Not varied for SPM mode.

$$V_{target} = \begin{cases} V_{target} & V_{target} \geq 1.3V_t \\ 1.3V_t & \text{otherwise} \end{cases}$$

The nominal supply voltage V_{DD}, nominal frequency f and the threshold voltage V_t are constants and are derived from empirical measurements. The minimal target voltage allowed for correct functionality is set to be 30% higher than V_t. The target voltage V_{target} calculated is then used to scale down the total power of the system by $(\frac{V_{target}}{V_{DD}})^2$.

DVFS allows for lowered dynamic power consumption, at the cost of some performance. However, the performance loss is negligible if the system is memory-bounded, which depends on the amount of compute resources, external memory bandwidth and, crucially, the application and its input data.

5.3.2.2 Cache Capacity

We re-implement each R-DCache bank as a set of sub-banks, i.e. each logical R-DCache bank is composed of a set of physical SRAM and tag arrays. There is a small combinatorial logic overhead to select between different set-index and tag bits, depending on the active cache capacity value. A larger cache allows for fewer misses, at the cost of increased latency and energy per access. Adaptable cache sizing has been explored in the past and shown to improve energy-efficiency in traditional architectures [195].

5.3.2.3 Sharing Mode

Transmuter is composed of swizzle-switch network based crossbars [182] that are augmented with XCUs. The XCU incorporates the ability to reconfigure the L1 and

L2 memory between a shared and a private configuration, across the GPEs within a tile for the L1 and across the tiles for the L2.

The choice of sharing mode can have either a positive or negative effect on performance. On one hand, the shared R-XBar configuration incurs larger access latency, due to the overhead of arbitration between different requesters accessing the same resource(s), but allows for data sharing, which can lead to better hit-rates and improved reuse. On the other hand, the private mode offers a fixed, 1-cycle access latency, but privatizes the memory resource to the requester. Privatization can reduce cache pollution if the system encounters thrashing, but also causes duplication for shared data.

5.3.2.4 On-Chip Memory Type

We consider reconfiguration between two on-chip memory types, cache and scratchpad memory (SPM). SPM consumes lower power than an equivalent cache by power-gating the tag array and other unused logic. SPM is also faster when there is reuse across a set of arbitrary (but known) memory locations. It thereby takes advantage of a tailored replacement policy at the cost of additional instructions to orchestrate data. In contrast, a cache trades-off the flexibility of SPM by implicitly managing data, and thus generally outperforms for regular memory accesses or when there is low reuse.

5.3.2.5 Prefetcher Aggressiveness

The L1 and L2 layers in Transmuter consist of a stride prefetcher based on a PC-based index table. We incorporate the ability to switch between different degrees of prefetching, i.e. the number of cachelines to prefetch ahead. The usefulness of this prefetcher is correlated with the amount of structure in the non-zeros within the input matrices. For highly unstructured data, turning off the prefetcher can lead to minimum performance degradation while saving power.

5.3.3 Performance Counters

We implement low-overhead performance counters in the hardware that are reset after they are queried. These counters are listed in Table 5.2. The telemetry data is averaged both spatially (across all replicated hardware blocks) and temporally (normalized to the elapsed cycle count of the epoch) by the runtime. The runtime routine performs lightweight pre-processing upon receiving the telemetry data, such as normalization and feature set augmentation.

The performance counters are hand-picked based on architectural knowledge and correlation studies, such that we select those that have small cross-correlations and are oblivious to the code being executed. At the same time, they are designed with low hardware overhead considerations. The performance counters for GPE/LCP cores are available off-the-shelf. The remainder are constructed as simple saturating counters that poll on existing wires, with the exception of the crossbar contention-to-access-ratio for which dedicated wiring is used to detect bank contentions. Overall, we estimate the storage overhead to be < 1 kB for the evaluated 2×8 system, and $\sim (42.5 \cdot M \cdot N)$ B for large $M\times N$ systems.

Query Mechanism. The performance counters are memory-mapped and accessed by the host using a PCIe-like protocol. The counter data is queried and streamed out to DRAM, before being loaded by the host. These steps take place in the shadow of the workload executing on the device.

5.3.4 Cost of Telemetry and Reconfiguration

We estimate the decision-making and communication process to cost on the order of 50-100 host clock cycles. The greater runtime reconfiguration cost arises from the overhead to flush caches, as we pessimistically assume that the entire cache hierarchy is dirty prior to reconfiguration. The prediction problem is further complicated, because not all configuration changes incur the same cost, as not all configuration changes require cache flushing, or require local flushes only (i.e. from L1 to L2). As such, we introduce the following taxonomy of our configuration parameters.

- **Coarse-Grained.** Hardware parameters that require substantive change in the code running on the GPE and LCP cores, in addition to cache flushing. Both the memory type and dataflow configuration changes belong to this category.
- **Fine-Grained.** Parameters that require at most a cache flush, but do not impact the code executed on the cores.
- **Super Fine-Grained.** Parameters that incur a small, fixed reconfiguration cost and can be reconfigured without impacting the code or even necessitating a cache flush.

We assume in this work that the L1 memory type parameter is selected by the compiler and use SparseAdapt to predict for the remaining six parameters (Table 5.1).

Table 5.2: List of hardware counters in this work.

Hardware Block	**Performance Counters**
R-DCaches	Cache/SPM access throughput, i.e. number of accesses per unit time, cache occupancy, i.e. fraction of valid tags in the bank, overall miss rate, number of prefetches issued per cache access, current cache capacity.
R-XBars	Contention-to-Access Ratio, i.e. ratio of number of contentions across all output ports to the number of accesses through the crossbar.
LCP/GPE Cores	Floating-point instructions per cycle (including loads and stores), overall instructions per cycle (IPC), clock speed.
Memory Controller	Read and write memory bandwidth utilization, i.e. used bandwidth normalized to available bandwidth.

This is done to avoid the cost of checkpointing and rollback that would be incurred while switching between the cache and SPM modes for L1 memory. For fair comparisons in our evaluation, we compare against distinct static configurations that each perform best for the cache and SPM modes, respectively.

5.4 Predictive Model

We illustrate the functionality of the proposed SparseAdapt framework in Figure 5.3. SparseAdapt partitions the program execution into multiple *epochs*. An epoch finishes when the floating-point instructions executed (inclusive of loads and stores), averaged across spatial hardware instances, exceeds a fixed value. We use this instead of considering *all* operations, in order to discount operations that do not affect program behavior, such as spin-waiting on a lock variable. At the end of each epoch, a sequence of three operations is performed, namely, (i) hardware telemetry, (ii) inference using a predictive model, and (iii) reconfiguration of the hardware.

First, a set of performance counter registers in the Transmuter design are polled and their values are streamed through the off-chip interface to the host processor. Then, the predictive model is invoked on the host, and it is responsible for predicting the parameters of the hardware configuration that it deems to be the best for the next epoch of execution.

In contrast, the state-of-the-art ProfileAdapt, switches to a profiling configuration before the hardware telemetry is performed. It further assumes that the hardware has

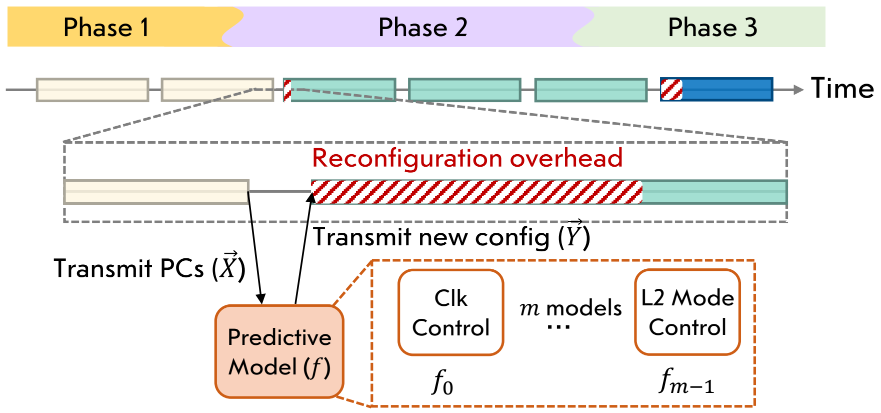


Figure 5.3: Overview of SparseAdapt illustrating how the predictive model is used to read performance counter (PC) feedback and predict the best configuration for the remainder of the program phase.

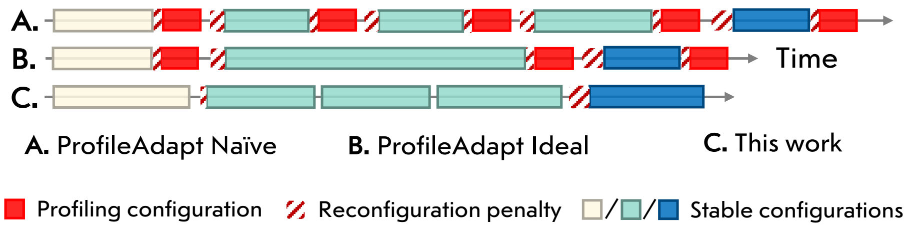


Figure 5.4: Timing of SparseAdapt compared to ProfileAdapt [56] considering **A.** fixed epochs, and **B.** epoch size equal to program phase size. SparseAdapt alleviates the overhead of switching to the profiling configuration.

the ability to dynamically predict phases at runtime. This is a highly non-trivial task, particularly for real-world datasets in sparse computation, that produce implicit phase transitions during execution. We thus illustrate two versions of ProfileAdapt [56] in Figure 5.4. **A.** represents the worst-case scenario that relies on reconfiguration at epoch-granularity when phases cannot be determined through other mechanisms. **B.** shows the execution timeline assuming phase changes could be known in advance, which is idealistic given the irregular nature of the sparse workload.

As we will see in the evaluation (Section 5.6.4), our approach is superior to both **A.** and **B.**, by virtue of eliminating the back-and-forth switch to the profiling configuration. While this may be of limited importance if the program phases are relatively stable and sufficiently long, it helps achieve significant savings for bursty program behavior consisting of unique implicit phases, such as with the SpMM example shown in the motivation (Figure 5.1). The rest of this section is dedicated to discussion on the predictive model.

5.4.1 Model Construction

The predictive model (Figure 5.3) can be formulated as a function $f : \vec{X} \rightarrow \vec{Y}$ where $\vec{X}$ is a tuple of selected performance counter values, and $\vec{Y}$ is the set of configuration parameter values. Similar to the ProfileAdapt approach, we consider each configuration dimension $Y_i \ \forall \ i \ \in \ \vec{Y}$ to be conditionally independent given the values of performance counters $\vec{X}$, in order to simplify the model. The predictive model is thus an ensemble of independent functions $f_i, i \in M$ where M is the space of the configuration parameters (Table 5.1).

We describe the methodology we use to determine the "best" architectural configuration, where "best" refers to the highest GFLOPS/s/W (Energy-Efficient mode) or $(\text{GFLOPS/s})^3/\text{W}$ (Power-Performance mode). This is illustrated in Figure 5.5, assuming three configuration parameters, Y_0, $Y_1 \cdots Y_{m-1}$ with $m = 3$, for ease of illustration. It involves these steps:

1. **Random Sampling.** We first sample K unique configurations from the space of *all* architectural configurations. Then, given a program phase P and input dataset D, we execute the code on each of the K configurations. We record the "best" among these K configurations as $\vec{Y}_{D,P,rand}$.

2. **Neighbor Evaluation.** We next evaluate the configurations in the m-dimensional hyper-sphere surrounding $\vec{Y}_{D,P,rand}$, and record the "best" one as $\vec{Y}_{D,P,neigh}$.

3. **Dimension Sweep.** Finally, starting with $\vec{Y}_{D,P,neigh}$, we sweep each configuration dimension in isolation and record the "best" points as $\vec{y}_0^*$, $\vec{y}_1^*$ and $\vec{y}_2^*$ (orange dots in Figure 5.5). We denote the point $\vec{Y}_{D,P,sweep} = \{\vec{y}_0^*, \vec{y}_1^*, \vec{y}_2^*\}$ (starred in figure) as the best configuration given D and P.

5.4.2 Dataset Construction and Training

Our approach differs from ProfileAdapt [56] in how we construct the training dataset that is used for training our predictive model. *Our key insight is to use the values of configuration parameter of the last epoch as inputs to the predictive model.* This gives us access to vastly more amount of training data, since for each program phase P we can construct K examples containing performance counter inputs that should trigger reconfiguration to the "best" hardware configuration (Figure 5.6). This approach also helps the model generalize and learn to predict from *any configuration* to the best configuration, rather than just from the profiling configuration to the best configuration.

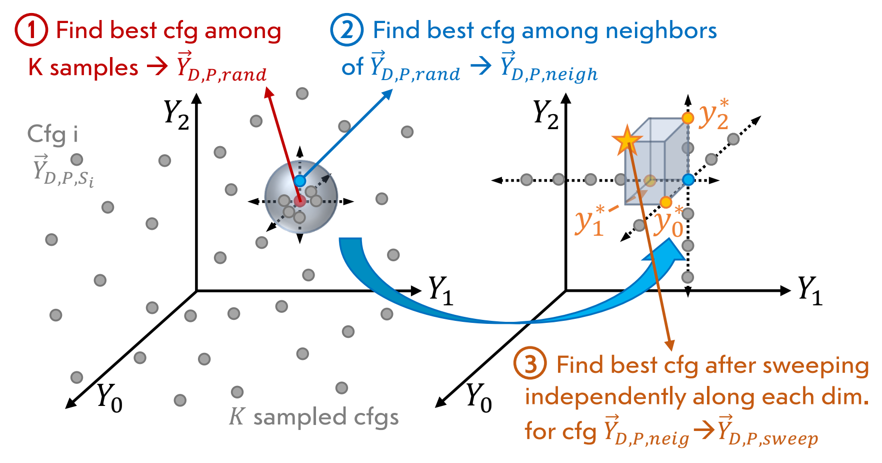

① Find best cfg among K samples → $\vec{Y}_{D,P,rand}$
② Find best cfg among neighbors of $\vec{Y}_{D,P,rand}$ → $\vec{Y}_{D,P,neigh}$
③ Find best cfg after sweeping independently along each dim. for cfg $\vec{Y}_{D,P,neig}$→$\vec{Y}_{D,P,sweep}$
Cfg i $\vec{Y}_{D,P,S_i}$
K sampled cfgs
Y_0
Y_1
Y_2
y_0^*
y_1^*
y_2^*

Figure 5.5: Simulations to find the "best" configuration are performed in three steps: random sampling, neighbor evaluation and dimension sweep.

5.4.3 Choice of Predictive Model

We pose the following requirements from an ideal model.

- **Accuracy.** The model should have high accuracy in predicting the best configuration parameters for the next epoch given the current performance counters of the current epoch.
- **Generalizability.** The model should generalize and predict for counter values that it has not been trained with.
- **Overhead.** The inference overhead should be sufficiently low such that the time cost of executing telemetry, inference, and reconfiguration $\ll$ the epoch size. This is necessary to enable fine-grained reconfiguration, in order to adapt to short-lived implicit phases.

We experimented with four machine learning models, namely decision trees, random forests, linear regression, and logistic regression. We found similar inference accuracies between decision trees and random forests, whereas the linear and logistic regression models gave us poor accuracies. We thus selected decision trees (with pruning) as our predictive model, since they adhere to the aforementioned constraints the best.

5.4.4 Reconfiguration Cost-Aware Prediction

As discussed earlier, some architectural parameters incur a higher reconfiguration cost. While the super fine grained parameters incur relatively lower cost (Sec-

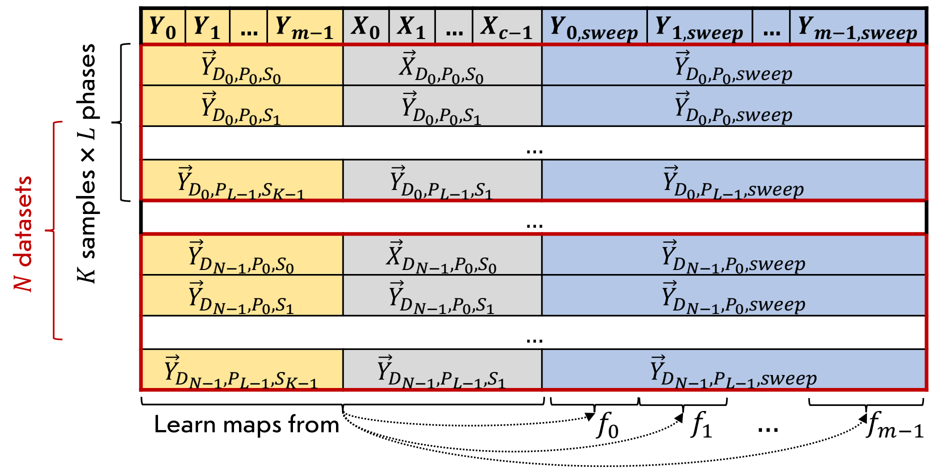

Y_0 Y_1 ... Y_{m-1} X_0 X_1 ... X_{c-1} $Y_{0,sweep}$ $Y_{1,sweep}$ ... $Y_{m-1,sweep}$
$\vec{Y}_{D_0,P_0,S_0}$ $\vec{X}_{D_0,P_0,S_0}$ $\vec{Y}_{D_0,P_0,sweep}$
$\vec{Y}_{D_0,P_0,S_1}$ $\vec{Y}_{D_0,P_0,S_1}$ $\vec{Y}_{D_0,P_0,sweep}$
$\vec{Y}_{D_0,P_{L-1},S_{K-1}}$ $\vec{Y}_{D_0,P_{L-1},S_1}$ $\vec{Y}_{D_0,P_{L-1},sweep}$
$\vec{Y}_{D_{N-1},P_0,S_0}$ $\vec{X}_{D_{N-1},P_0,S_0}$ $\vec{Y}_{D_{N-1},P_0,sweep}$
$\vec{Y}_{D_{N-1},P_0,S_1}$ $\vec{Y}_{D_{N-1},P_0,S_1}$ $\vec{Y}_{D_{N-1},P_0,sweep}$
$\vec{Y}_{D_{N-1},P_{L-1},S_{K-1}}$ $\vec{Y}_{D_{N-1},P_{L-1},S_1}$ $\vec{Y}_{D_{N-1},P_{L-1},sweep}$
N datasets
K samples × L phases
Learn maps from
f_0 f_1 ... f_{m-1}

Figure 5.6: The training dataset uses both the current configuration tuple $\vec{Y}_{D,P,S}$ and the performance counters $\vec{X}_{D,P,S}$, given an input dataset D, program phase P, and sampled configuration S. The predictive model $\{f_0, f_1, \cdots, f_{m-1}\}$ learns the mapping of $\{\vec{Y}_{D,P,S}, \vec{X}_{D,P,S}\} \rightarrow \vec{Y}_{D,P,sweep}$.

tion 5.3.4), frequent reconfiguration for other parameters, such as cache capacity, can lead to performance degradation *even if the prediction is accurate*. In order to prevent the model from switching high-cost parameters too frequently, we add a degree of hysteresis through a set of heuristic-based schemes:

- **Conservative.** The predictor does not perform reconfiguration for a parameter that incurs more than a fixed cost.
- **Aggressive.** A scheme where the predictor always chooses to reconfigure based on its decision, regardless of the cost.
- **Hybrid.** For each configuration parameter, the predictor only reconfigures if the time cost of reconfiguring along that dimensions is within a certain percentage of the previous epoch's elapsed time. We consider this instead of an absolute threshold on the reconfiguration time, so as to penalize frequent bursts of reconfiguration within short periods (shorter epoch times), but allow for them when they occur occasionally (larger epoch times). The percentage threshold is selected empirically based on our experiments.

5.5 Experimental Setup

This section describes our training data collection and system modeling methodologies, followed by descriptions of baselines used for evaluation.

5.5.1 Data Collection and Model Training

The ideal training dataset should produce a high variability in the program behavior in order to stimulate a wide range of performance counter values. For this purpose we select a set of sparse matrices with a broad range of input working set sizes, ranging from 1.5 kB to 67 MB. We further vary the external memory bandwidth so as to stimulate the system with both memory-bound and compute-bound scenarios. Overall, we generate $\sim 360k$ training examples (total for two modes of operation) by sweeping the parameters in Table 5.3 and running them on ~ 285 discrete hardware configurations.

We consider uniform random datasets as our inputs, so that we can safely consider the entire program phase to have uniform behavior. Each training example is generated by running a program phase P until the program behavior stabilizes, terminating it, and sampling the performance counter values. For outer product based SpMM, there are two program phases, multiply and merge, while for SpMV the multiply and merge steps happen in tandem.

Predictive Model Training. We train a decision tree classifier for each of our configuration parameters. While a clear advantage of our choice of decision trees as the predictive model lies in its explainability, decision trees are prone to overfitting [23]. We thus train our decision trees using k-fold cross-validation [12] with $k = 3$, while sweeping the hyperparameters of `criterion`, `max_depth`, and `min_samples_leaf` using Python and Scikit-learn [163].

5.5.2 System Modeling

We used gem5 [22, 21] to model the Transmuter system based on the architectural specification in [155]. A power estimator is constructed using a combination of RTL synthesis reports for crossbars, Arm specification document for the cores, and CACTI [143] for the caches and SPM. The estimates are scaled to 14 nm across the evaluated systems. We made additional modifications to the simulator and power estimator to model the microarchitectural changes from Section 5.3.2.

For workload simulation, we do not use SimPoint as we evaluate for matrices with arbitrary sparsity structures (Section 5.2.2). This limits the maximum sizes of matrices we can simulate within a reasonable time, since we simulate each input-kernel combination with different hardware configurations. Due to these constraints, we consider a relatively small Transmuter system on which we exercise sparse linear algebra kernels with moderate dataset sizes which are much larger than the on-chip

Table 5.3: Parameter sweeps to generate training data.

Kernel Name	Algorithm Variant	Matrix Dimension	Matrix Density	Memory Bandwidth (GB/s)
SpMM	Cache, SPM	$128 \rightarrow 1k : 2*$	$0.2 \rightarrow 13\% : 2*$	$0.01 \rightarrow 100 : 10*$
SpMV	Cache, SPM	$256 \rightarrow 8k : 2*$	$0.2 \rightarrow 13\% : 2*$	$0.01 \rightarrow 100 : 10*$

memory. Specifically, we consider a Transmuter system that has 8 GPEs (8 L1 cache banks) per tile, and 2 tiles (2 L2 cache banks). We also assume a reduced off-chip memory bandwidth of 1 GB/s in order to maintain similar compute-to-memory ratio as the full system in [155].

Reconfiguration Cost. Following the taxonomy in Section 5.3.4, we model variable penalty for reconfiguration of each parameter. Changes to the super fine grained parameters are assigned a fixed cost of 100 cycles. An increase in cache capacity also incurs this fixed cost, as our cache implementation (described in Section 5.3.2.2) allows for it.

For reconfiguration of the fine-grained parameters, we pessimistically assume that all the cache lines are dirty. We thus assign a penalty equal to the time to flush all the R-DCache banks in a layer to the next level of hierarchy, i.e. L1 to L2 (100–961k cycles, up to 157 µJ energy), and L2 to main memory (100–122k cycles, up to 22 µJ energy) with 1 GB/s off-chip memory bandwidth. In reality, these overheads are expected to be lower due to fewer dirty lines, since only the partial products and bookkeeping data structures incur read-modify-write operations whereas the rest can be written directly to main memory. The host selects the optimal clock speed for cache flushing based on the mode of operation, i.e. Energy-Efficient or Power-Performance, based upon a lookup table that is indexed by the operational mode, L1 capacity per bank, and L2 capacity per bank. A significant amount of energy savings is obtained by power-gating the cores, ICaches, work and status queues, and the synchronization SPM while caches are being flushed.

The coarse-grained parameter decision is assumed to be made at compile-time, and so does not impact the runtime.

5.5.3 Comparison Points

We consider several comparison points for evaluating the improvements achieved using SparseAdapt:

- **Baseline.** A non-reconfiguring Transmuter system that achieves the best average metric across a broad set of applications (dense and sparse) evaluated in [155].

- **Best Avg.** A non-reconfiguring system that achieves the best average metric for the SpMM and SpMV kernels on the datasets considered in this work.
- **Max Cfg.** A non-reconfiguring system that has maximum values for each ordinal configuration parameter, and that uses shared caches in both the L1 and L2.
- **Ideal Static.** A non-reconfiguring system (from our sampled space) that achieves the best average metric for the given sparse algebra routine and dataset.
- **Ideal Greedy.** A system that dynamically reconfigures during execution of the routine. It greedily selects the configuration that is optimal *for the next epoch*. For each epoch, we choose the sampled configuration state that achieves the best metric when adjusted for the reconfiguration cost.
- **Oracle.** A system that dynamically reconfigures assuming full knowledge of the entire program execution. It selects the sequence of configuration changes that result in the maximum average metric value. We model this as a dynamic programming problem and solve it using a modified Dijkstra algorithm to derive the globally optimal sequence for the sampled configurations.

We note that Ideal Static, Ideal Greedy and Oracle are configurations that cannot be determined at runtime, and thus are hypothetical systems used for upper-bound studies in this work. All the static configurations are specified in Table 5.4.

5.5.4 Choice of Dataset and Parameters

We evaluate SparseAdapt on SpMM and SpMV kernels using a variety of input datasets. Without loss of generality, we perform our evaluation on square matrices, stored in compressed sparse column (CSC) for Matrix A and compressed sparse row (CSR) for Matrix B (or as an array of index-value tuples for vector B).

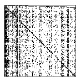
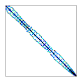
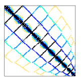
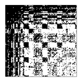
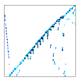
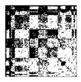
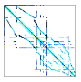
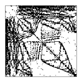
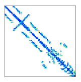
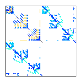
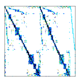
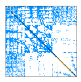
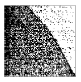
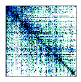
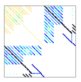
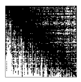
Synthetic Dataset. We generate uniform-random matrices using the SciPy library and for power-law matrices, we use the R-MAT generator [31] with parameters $A = C = 0.1$ and $B = 0.4$. The properties of these matrices are listed in Table 5.5 (top). They are selected to show trends across increasing NNZ count with a fixed dimension (U1 to U3 and P1 to P3).

Table 5.4: Specifications of baseline Transmuter hardware.

Config Name	L1 Bank Size (kB)	L1 Mode	L2 Bank Size (kB)	L2 Mode	Clock (MHz)	Pref. Agg.
Baseline	4	Shared	4	Shared	1000	4
Best Avg (L1: cache)	4	Private	4	Shared	1000	0
Best Avg (L1: SPM)	4	Private	32	Private	500	8
Maximum	64	Shared	64	Shared	1000	8

Table 5.5: Properties of matrices used for evaluation: synthetic (top) and real-world [122, 48] (bottom).

ID	U1	U2	U3	P1	P2	P3
Type	Uniform	Uniform	Uniform	Power-Law	Power-Law	Power-Law
Dim.	8,192	8,192	8,192	8,192	8,192	8,192
NNZ	25,000	50,000	100,000	25,000	50,000	100,000

ID	Matrix Name Matrix Dim., NNZ Application Domain	Plot	ID	Matrix Name Matrix Dim., NNZ Application Domain	Plot
R01	California (9.7K, 16.2K) (Directed Graph)		R09	EX3 (1.8K, 52.7K) (Comp. Fluid Dyn.)	
R02	Si2 (0.8K, 17.8K) (Quant. Chemistry)		R10	Oregon-1 (11.5K,46.8K) (Undirected Graph)	
R03	bayer09 (3.1K, 11.8K) (Chemical Simulation)		R11	as-22july06 (23.0K, 96.9K) (Undirected Graph)	
R04	bcsstk08 (1.1K, 13.0K) (Structural Problem)		R12	crack (10.2K, 60.8K) (2D/3D Problem)	
R05	coater1 (1.3K, 19.5K) (Comp. Fluid Dyn.)		R13	kineticBatchReactor_3 (5.1K, 53.2K) (Optimal Control)	
R06	gemat12 (4.9K, 33.0K) (Power Network)		R14	nopoly (10.8K, 70.8K) (Undirected Graph)	
R07	p2p-Gnutella08 (6.3K, 20.8K) (Directed Graph)		R15	soc-sign-bitcoin-otc (5.9K, 35.6K) (Directed Graph)	
R08	spaceStation_11 (1.4K, 19.0K) (Optimal Control)		R16	wiki-Vote_11 (8.3K, 103.7K) (Directed Graph)	

Real-World Dataset. We derive real-world matrices for our evaluation from two popular sparse matrix collections, namely SuiteSparse [48] and Stanford's SNAP [122].

These matrices are described in Table 5.5 (bottom).

We evaluate SpMM for matrices R01-R08, and SpMV using R09-R16. We select comparatively more modest matrix sizes for SpMM since it is computationally more expensive to simulate than SpMV. Sizes of this order provided us a balanced trade-off with the resources required to perform cycle-accurate simulations.

We use a small epoch size of 500 instructions for SpMV to capture implicit phases at runtime. For the same epoch size, SpMM generates significantly more performance counter data than SpMV. In order to consume tractable amount of data with the given resources, we use a larger epoch size of $5k$ for SpMM. However, we note that SparseAdapt itself is not limited to any specific workload or epoch size, and the benefits reported in this section are representative of larger datasets. Finally, unless otherwise noted, we assign the conservative policy for SpMM and hybrid policy with 40% tolerance for SpMV based on sweep studies on the real-world dataset for each kernel.

5.6 Evaluation

We evaluated our proposed framework first against the Baseline, Max Cfg and Best Avg configurations, followed by comparison against multiple oracle control schemes. We then present insights into our predictive model. This is followed by comparison against ProfileAdapt [56]. We conclude with studies on the impact of SparseAdapt policies and the gains achieved across memory bandwidth sweeps.

5.6.1 Comparison with Standard Configurations

We discuss our analysis of improvements over the Baseline, Max Cfg and Best Avg systems here.

5.6.1.1 Analysis on Synthetic Dataset

We report evaluation of SpMV on our synthetic dataset against a uniform-random vector of density 50%. We omit our analysis of SpMM on the synthetic dataset, for brevity.

Power-Performance Mode. Figure 5.7 shows the performance in GFLOPS/s (left) and energy efficiency in GFLOPS/s/W (middle) of Baseline, Best Avg, Max Cfg and SparseAdapt. In this mode, SparseAdapt delivers average performance gains of 1.8× over Baseline. In this mode, the energy efficiency achieved is 3.5× better than Max

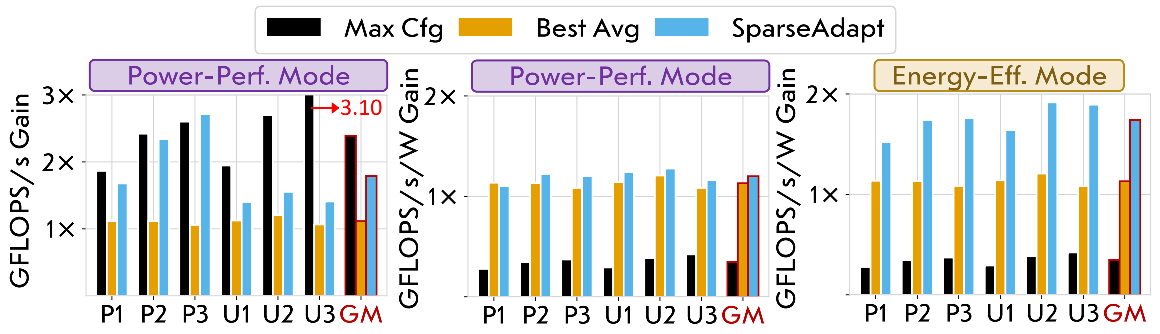


Figure 5.7: Gains over Baseline for SpMV on synthetic matrices for Power-Performance (left, middle) and Energy-Efficient (right) modes for L1 as cache.

Cfg, while the average performance remains within 34%. Although SparseAdapt achieves only 6% better efficiency than Best Avg, it delivers 1.6× better performance.

Energy-Efficient Mode. As can be seen in Figure 5.7 (right), SparseAdapt achieves an average energy-efficiency gain of 1.5–1.9× over the Baseline. The gains for both the power-law and uniform random matrices are observed to roughly saturate with increasing the density, as SparseAdapt follows the Oracle decisions more closely, particularly for the clock frequency and L2 mode. The Max Cfg configuration is 2.9× less energy efficient than Baseline despite being faster. In comparison, the Best Avg achieves 1.1× over the Baseline.

5.6.1.2 Analysis on Real-World Dataset

We present our analysis of SpMM on matrices R01-R08 in Table 5.5. Each matrix is multiplied with its transpose, i.e. $C = A \cdot A^T$, where A is the input and C is the result.

Power-Performance Mode. The gains for this mode are shown in Figure 5.8. The scope of performance improvements over Baseline is smaller than in the case of SpMV (for our synthetic dataset), and SparseAdapt achieves similar performance as Best Avg (within 8% of Max Cfg). However, SparseAdapt delivers this performance at 1.3× less energy over Best Avg , and is 5.3× more efficient than Max Cfg.

Energy-Efficient Mode. Figure 5.8 (right) shows the efficiency gains with SparseAdapt, which are significantly higher (1.8×) than Baseline, and is better than Best Avg by (1.6×).

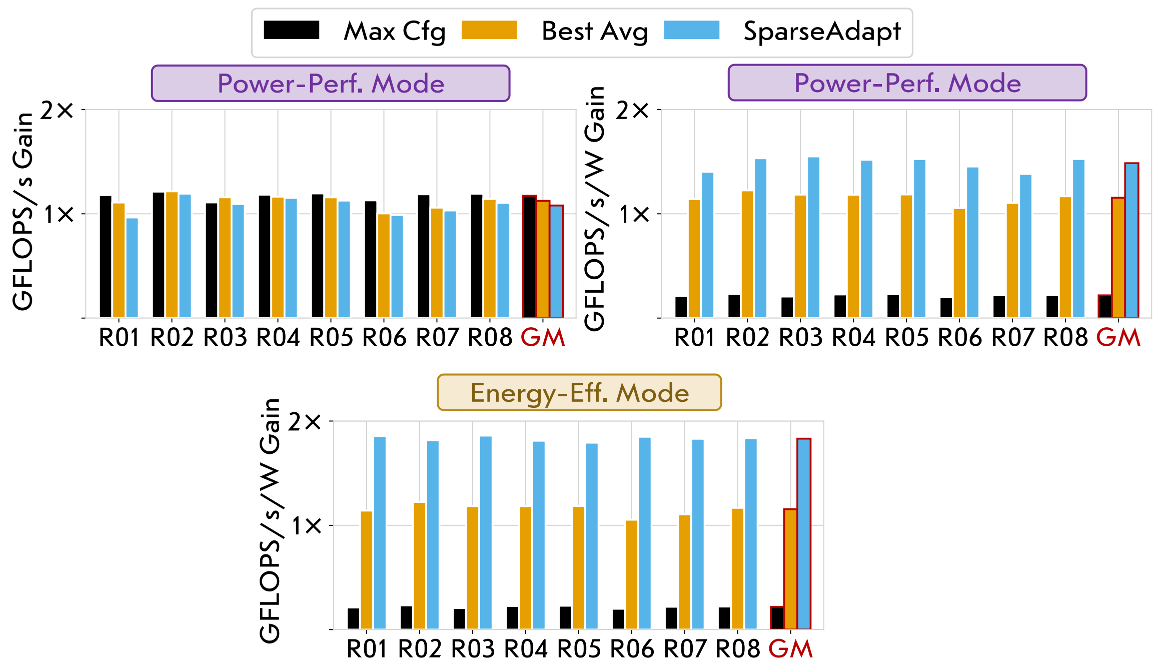


Figure 5.8: Improvements over Baseline for SpMM on real-world matrices in Power-Performance (left, middle) and Energy-Efficient (right) modes for L1 as cache.

Table 5.6: TEPS per Watt gains over Baseline for graph algorithms in Energy-Efficient mode with L1 as cache.

		R09	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**	**R16**	**GM**
BFS	Max Avg	1.26	1.14	1.11	1.07	1.17	1.13	1.17	1.24	**1.16**
	SparseAdapt	1.28	1.32	1.44	1.34	1.15	1.40	1.27	1.27	**1.31**
SSSP	Max Avg	1.21	1.13	1.10	1.03	1.13	1.05	1.15	1.18	**1.12**
	SparseAdapt	1.21	1.43	1.45	1.27	1.17	1.31	1.29	1.21	**1.29**

5.6.1.3 Analysis on Graph Algorithms

We implement two popular graph algorithms: breadth-first search (BFS) and single-source shortest path (SSSP). Our implementations map vertex programs to iterative SpMV operations, similar to GraphMat [194]. The end-to-end improvements in traversed edges per second (TEPS) per Watt are reported in Table 5.6, showing that SparseAdapt delivers gains of up to 1.5× over Baseline (1.3× over Best Avg). The largest gains are observed on graphs with highly power-law behavior, for instance, R10, R11 and R14. In contrast, the scope of improvement over Best Avg is small for R09, as it consists of local connections only and thus the non-zeros are distributed roughly uniformly along the diagonal (Table 5.5).

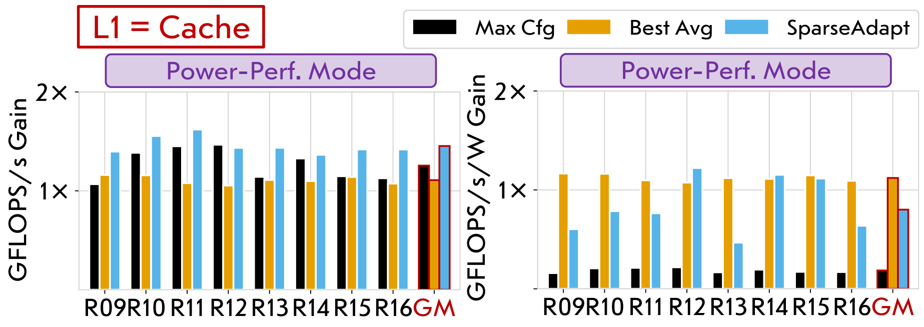


Figure 5.9: Improvements over Baseline for SpMV on real-world matrices in Power-Performance mode with L1 as cache.

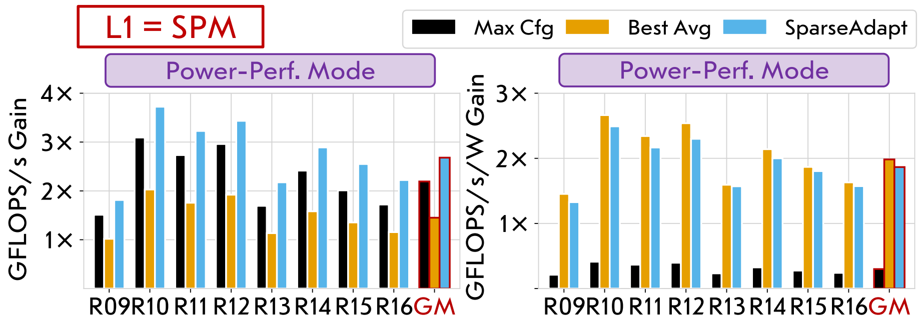


Figure 5.10: Improvements over Baseline for SpMV on real-world matrices in Power-Performance mode with L1 as SPM.

5.6.1.4 Analysis with Cache/Scratchpad L1 Memory

As noted in Section 5.3.4, we assume that the choice of L1 mode (cache or SPM) is made at compile-time. We thus have two configurations for the two L1 modes, listed in Table 5.4. The results for Power-Performance mode on SpMV for our real-world dataset (R09-R16) with L1 as cache and SPM are shown in Figure 5.9 and Figure 5.10, respectively. The performance gains are larger for L1 SPM (1.9× over Best Avg) compared to L1 cache (1.3× over Best Avg). This is 1.2× better than Max Cfg for cache and 1.2× better than Max Cfg for SPM, while being 4.3× and 6.2× more energy-efficient for the two L1 modes, respectively. The cache mode benefits are 1.5× better over Baseline, albeit with 1.2× greater energy, and this is due to the predictor selecting large L2 cache sizes.

5.6.1.5 Insights from Configuration Choices

We observe that the model applies DVFS based on the bandwidth requirement of the explicit phase (Section 5.2.1), and selects faster clocks when there is greater data locality. The L1 size choice is correlated to the cache occupancy, however the number of reconfigurations are curbed based on our hybrid policy for fine-grained knobs (Section 5.4.4). In comparison, the prefetcher aggressiveness and L2 capacity are reconfigured more often, in response to implicit phases.

The L1 mode choice is highly data-dependent for SpMV; for SpMM, we observe the multiply phase to be amenable to shared L1 while merge performs better on private L1, which are consistent with the description in [154]. When the L1 is configured as SPM, the model selects larger L2 sizes to accommodate auxiliary data structures (those not mapped to SPM) and spill variables. Finally, the model shows preference for larger L1 and L2 capacities for the Power-Performance mode, and for smaller sizes in the Energy-Efficient mode.

5.6.2 Comparison against Ideal and Oracle

We report our analysis of upper-bound studies to evaluate SparseAdapt and compare it with Ideal Static, Ideal Greedy and Oracle. Figure 5.11 shows the results for SpMM on real-world dataset R01-R08 with L1 as cache.

The gaps between each of these and SparseAdapt convey different insights; Ideal Static represents the gains achievable if we had an ideal *compile-time* predictor that can select the best average configuration. Ideal Greedy represents our SparseAdapt approach if the predictive model was ideal, and Oracle shows the gains achievable if the predictor had full knowledge of the future. While the performance gains are comparable between Oracle over Ideal Static (Power-Performance mode), we note the high scope of improvements with dynamic reconfiguration (1.3–1.8×) for GFLOPS/s/W. Compared to Oracle, SparseAdapt is within 13% performance for Power-Performance mode, and just 5% efficiency for both the modes. SparseAdapt achieves an energy efficiency within 3% of Ideal Greedy (Energy-Efficient mode), but interestingly the gains are 11% and 6% *better* for performance and efficiency, respectively, in Power-Performance mode. This is made possible by the policies introduced in Section 5.4.4, which generate inertia toward frequent reconfiguration changes along costly dimensions. When disabled, i.e. for the Aggressive scheme (not shown), the gains drop to 9% worse than Ideal Greedy for performance and energy efficiency, respectively.

Finally, for SpMV in the Power-Performance mode, SparseAdapt has 1% better (4% worse) GFLOPS/s and is within 56% (15%) in terms of GFLOPS/s/W compared to Oracle, for L1 as cache (SPM). We omit detailed analyses due to space constraints.

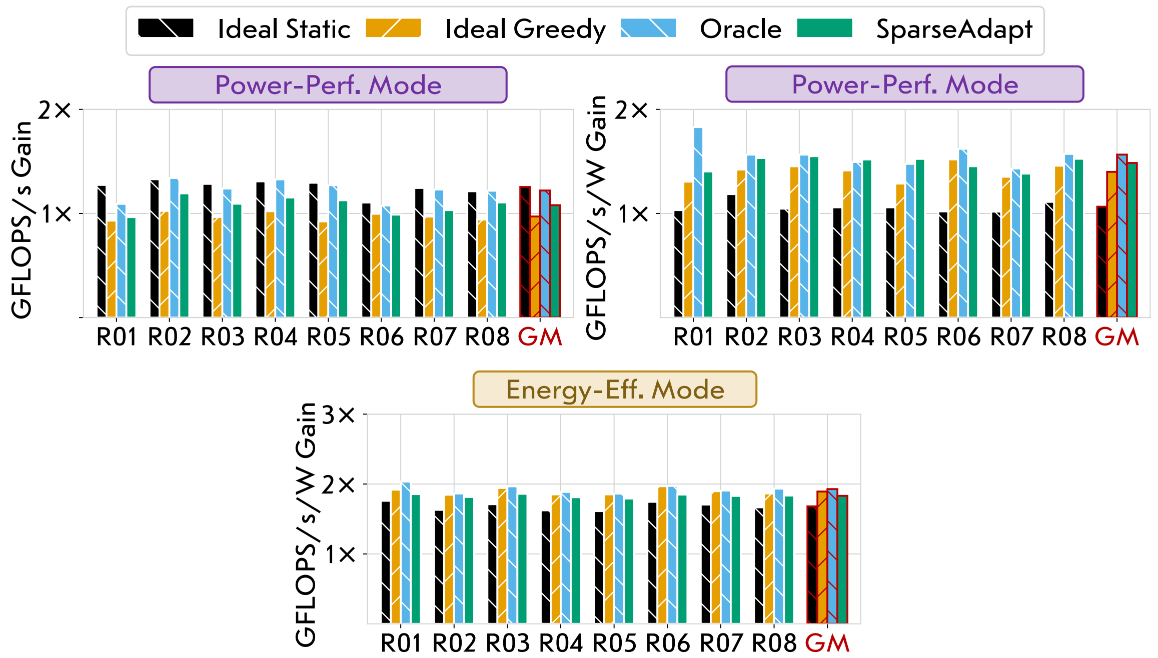


Figure 5.11: Comparison against Ideal Static, Ideal Greedy and Oracle for SpMM on real-world matrices in Power-Performance (left, middle) and Energy-Efficient (right) modes for L1 as cache. Gains are shown compared to Baseline.

5.6.3 Analysis of Model and Features

We discuss insights from our analysis of hyperparameter exploration and feature selection for our predictive model.

5.6.3.1 Effect of Model Complexity

As decision trees tend to overfit, it is important to regularize them at training. We explore this phenomenon by training decision trees with depths ranging from $2 \rightarrow 26 : 4*$. We analyze the gains achieved while independently varying the depth of the tree corresponding to each of the configuration parameters one-at-a-time, while using the original trees for the remaining. Here, original trees refer to the trees trained using the methodology in Section 5.5. The improvements over the Baseline in Power-Performance mode, for the case of SpMV with two matrices, P1 and P3, with a 50% dense vector, are shown in Figure 5.12. Given that our Power-Performance mode gives greater importance to performance optimization, GFLOPS/s is more sensitive to model complexity compared to GFLOPS/s/W. The model for L1 size is relatively unaffected by the model complexity, and this is due to inherent bias in our training data. We observe that most of our training phases have their optimal L1 capacity

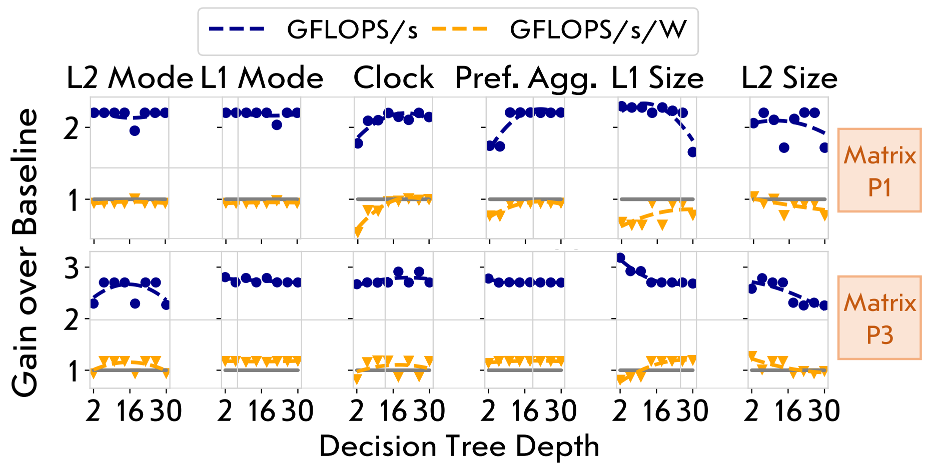


Figure 5.12: Effect of complexity of predictive model on the gains with SparseAdapt in Power-Performance mode for SpMV with L1 as cache.

values at either 4 or 8 kB per bank, which is explained by the diminishing performance returns and higher power with increasing cache capacity.

5.6.3.2 Feature Importance

Scikit-learn computes feature importance as the total reduction of criterion (i.e. function to measure the quality of a split in the tree) brought by that feature, also known as Gini importance. The feature importance values of our performance counters in determining the decisions of the learned predictive model are shown in Figure 5.13. The counters are grouped into categories for ease of illustration. Overall, we observe that counters probing the L1 R-DCache block and the memory controller are the most important across the models for each configuration parameter. Interestingly, the LCP counters (Instructions Per Cycle (IPC) and FP-IPC) are given more importance than GPE counters in the trained models for most cases. This can be attributed to the fact that because LCPs are responsible for scheduling work and load-balancing, they have a "global" view of the activity within each tile, compared to any individual GPE core. Overall, this analysis is useful to reason about which counters are the most critical to have in order to balance prediction accuracy with hardware requirements.

5.6.4 Comparison with ProfileAdapt Scheme

We compare SparseAdapt with a prior runtime reconfiguration scheme, ProfileAdapt [56]. ProfileAdapt triggers a reconfiguration into a "profiling" configuration

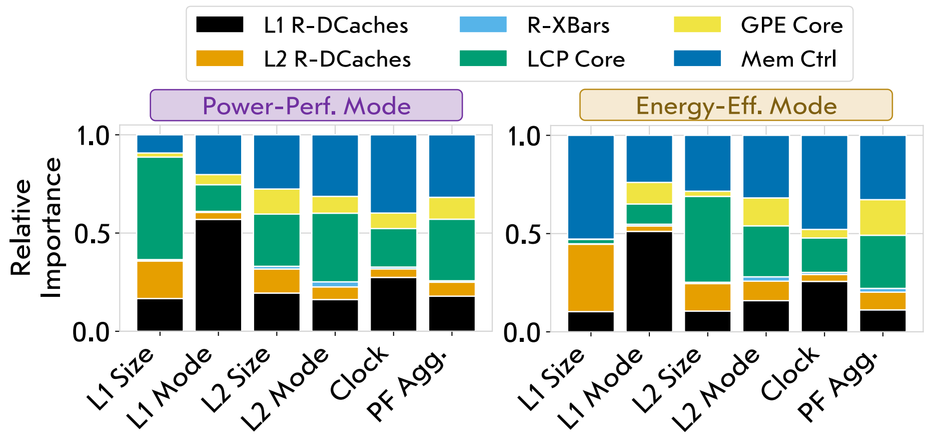


Figure 5.13: Relative importance of each class of performance counters for each trained model (x-axis) in Power-Performance (left) and Energy-Efficient (right) modes with L1 as cache.

at every epoch (naïve) or phase boundary (ideal) as illustrated in Figure 5.4, which can lead to large overheads especially for fine-grained phases. We implement ProfileAdapt by adding the cost to reconfigure to and from the profiling configuration into our Oracle sequence for each epoch (naïve), and for epochs following which there is a change in configuration (ideal). Furthermore, it would be unfair to compare ProfileAdapt and SparseAdapt at the same epoch size, since ProfileAdapt is designed to work with much larger epoch sizes. We, therefore, perform an epoch size sweep and select operational points for ProfileAdapt where the metric-of-interest is maximum, which is $6k$ FLOPS for the Power-Performance mode and $5k$ FLOPS for the Power-Performance mode. This is compared with the selected epoch sizes for SparseAdapt (Section 5.5.4).

We evaluate the SparseAdapt scheme against ProfileAdapt for SpMV in cache mode on the real-world dataset. Compared with the naïve ProfileAdapt, which switches to the profiling configuration at each epoch, SparseAdapt achieves significant improvements of 2.8× and 2.0× in GFLOPS/s and GFLOPS/s/W, respectively for the Power-Performance mode, and 2.9× gain in GFLOPS/s/W for the Energy-Efficient mode.

The ideal ProfileAdapt relies on a SimPoint based external phase detection mechanism which is unrealistic for workloads with implicit phases. Despite this idealistic assumption, SparseAdapt with its ability to reconfigure at implicit phases with a low overhead achieves gains of 1.7× and 1.1× GFLOPS/s/W for the Power-Performance mode, and 2.4× in GFLOPS/s/W for the Energy-Efficient mode.

5.6.5 Effect of Parameter Sweeps

Cost-Aware Reconfiguration Policies. In Section 5.4.4, we introduced our conservative, aggressive, and hybrid prediction policies for reconfiguration-cost aware prediction. We evaluate the efficacy of each of these on SpMV execution (epoch size 500) on representative matrices from the synthetic and real-world datasets, P3 and R12. The results are shown in Figure 5.14 (left). The conservative and hybrid schemes with small tolerance for reconfiguration penalty overly restrict the system from reconfiguring across implicit phases during execution, resulting in limited gains. As the tolerance is increased, the reconfiguration cost starts dominating over the benefit achieved from switching to the new configuration. The ideal tolerance values are observed to be between 10-40% across most of our inputs, given this epoch size and kernel.

Memory Bandwidth. SparseAdapt can be deployed without re-training in scenarios where there is sharing of bandwidth across concurrent kernel executions, or if the device is interfaced to a different type of main memory. We study the impact of these scenarios by sweeping the external memory bandwidth. Figure 5.14 (right) shows the energy-efficiency gains with SparseAdapt (in Energy-Efficient mode) for SpMV. The trend of gains deviates from a smooth curve for smaller bandwidth values, because the reconfiguration cost increases, thus increasing the impact of discrete parameter choices. When the system is memory-bounded, as is generally true for sparse computation, SparseAdapt achieves large energy-efficiency gains of >3× over both Baseline and Best Avg. It selects smaller cache sizes and slower clock speeds to recover both static and dynamic energy, with negligible performance impact. Even at the other end of compute-boundedness, it provides 1.1× gains over Best Avg.

System Size. Figure 5.15 shows our gains over Baseline when scaling (*i*) the number of tiles and L2 banks (M), and (*ii*) the number of GPEs and L1 banks per tile (N), for an $M \times N$ system. We observe mean GFLOPS/s/W gains of 1.7-2.0× across the four systems, obtained using the predictive model trained for a 2×8 system (no re-training) and an epoch size of $5k$. As the system size grows, the benefits with DVFS dominate, particularly for the multiply phase of OP-SpMM. These gains demonstrate the scalability of our framework.

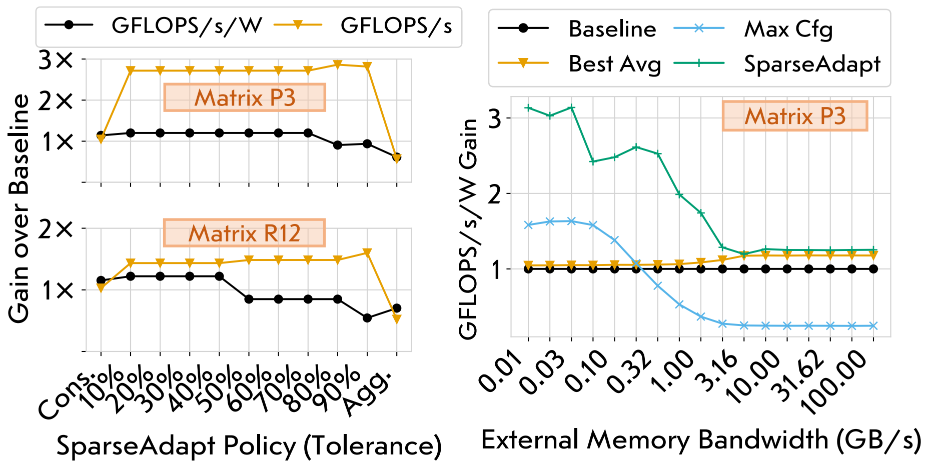


Figure 5.14: Left. Gains with different SparseAdapt policies (Power-Performance mode). Right. Impact of sweeping external memory bandwidth in Energy-Efficient mode. Both are for SpMV with L1 as cache.

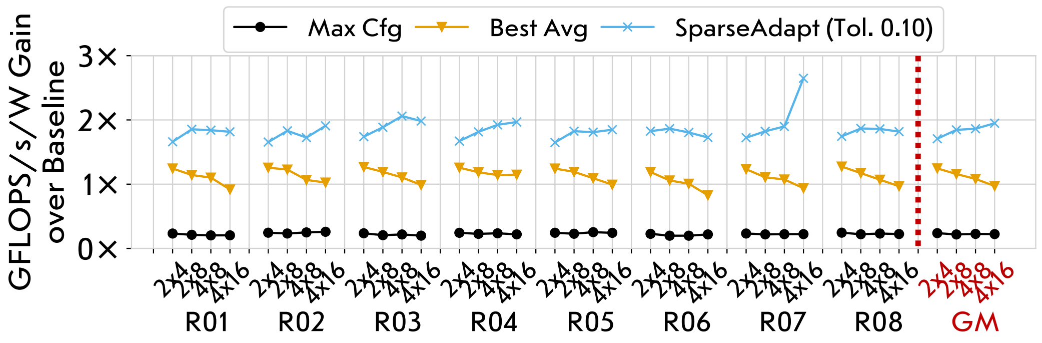


Figure 5.15: GFLOPS/s/W gains in Energy-Efficient mode for SpMM (R01-R08; L1 as cache) while varying the tile and GPEs per tile counts (fixed 1 GB/s bandwidth).

5.7 Discussion

Adaptation to Different Hardware and Algorithms. This work evaluates the SparseAdapt framework for sparse linear algebra operations. However, SparseAdapt is designed for programmable CGRAs (Transmuter in this work) and thus can be extended to additional kernels beyond SparseBLAS. This would simply require collection of additional training data using simulations for the new kernels, and re-training the predictive model. Since our performance counters are workload-agnostic, we expect no need for additional counters.

Bridging the Gap with Oracle. Ideal Greedy is a fundamental upper bound for the SparseAdapt approach in the Aggressive mode of operation. However, there is additional scope for improvement even between Oracle and Ideal Greedy in Section 5.6.2.

An extension to SparseAdapt will explore using telemetry data from multiple past epochs to learn a history-based pattern of program execution, borrowing ideas from branch prediction and prefetching.

Memory Mode Reconfiguration. We assume that the choice of L1 mode (cache or SPM) is made at compile-time, since this determines the version of code executed. This leaves out some scope for optimization when different parts of the program show amenability to a cache or SPM. Dynamic cache-to-SPM reconfiguration can be enabled using existing hardware techniques, such as Stash [109], that map sections of the SPM to global memory.

5.8 Conclusion

This work proposed a dynamic control scheme called SparseAdapt that targets changes in phase due to evolving properties of the data in irregular workloads (implicit), in addition to those due to change in code (explicit).

This requires fine-grained reconfiguration with low overhead, for which SparseAdapt uses runtime telemetry of hardware performance counters. SparseAdapt uses an ensemble of decision trees and a set of reconfiguration cost-aware heuristics to make decisions about configuration parameters of the hardware. We evaluated SparseAdapt on key SparseBLAS kernels (SpMM and SpMV) in two operational modes, namely Energy-Efficient and Power-Performance modes, that predict for the best GFLOPS/s/W and $(\text{GFLOPS/s})^3/\text{W}$, respectively. Our analysis on SpMM with real-world datasets show performance and energy-efficiency improvements of 1.1× and 1.5× over the baseline static configuration (similar performance as Max Cfg with 5.3× better efficiency) in Power-Performance mode, and 1.7× better efficiency in Energy-Efficient mode. Finally, the SparseAdapt technique achieves average gains of 1.7–2.8× in performance and 1.1–2.0× in energy-efficiency over a state-of-the-art control scheme for runtime adaptation.

CHAPTER VI

Conclusion

With transistor size no longer scaling in accordance with Moore's law, and the demise of Dennard scaling for constant transistor power density, the challenge is on hardware and systems researchers to innovate and develop faster and more efficient processors. While Application-Specific Integrated Circuit (ASIC) based accelerators are the predominant hardware paradigm choice for their superior performance, the rapid pace of algorithmic development and emergence of new applications has outpaced the turnaround time for designing new ASICs. General-Purpose Processors (GPPs), on the other hand, have long remained the *de facto* choice for programmers to implement arbitrary applications. Thus, there exists a programmability-efficiency gap between highly-efficient but fixed-function ASIC accelerators, and flexible GPP hardware.

This dissertation proposed a direction towards closing this gap with a reconfigurable Software-Defined Hardware (SDH) solution. The first two chapters introduced the trade-off problem in detail with qualitative studies on the state-of-the-art hardware platforms, in addition to motivating the need for a reconfigurable hardware based solution. The next part of the dissertation explored the use of on-chip memory reconfiguration and hardware-algorithm co-design to build an accelerator called OuterSPACE. OuterSPACE is a dedicated accelerator for sparse matrix multiplication operations, which are a particular class of irregular kernels that appear across domains such as graph analytics, scientific computing, and so on. Architectural simulation studies showed speedups of the order of $10\times$ over commercial libraries over a high end CPU and GPU. The dissertation then delved into the design and characterization of a prototyped 40 nm chip that served as an intermediary proof-of-concept towards a full SDH system. The measurements from the chip show an order of magnitude improvement in energy and bandwidth efficiency over the CPU.

The next part of this dissertation proposed a general-purpose accelerator called Transmuter that additionally incorporates reconfigurability of the on-chip resource sharing mode and the dataflow. The analysis across a set of common regular and irregular kernels showed roughly an order of magnitude gain in energy efficiency for compute-bound workloads, and an average improvement of $\sim 50\times$ for memory-bound workloads. With static (i.e. compile-time) reconfiguration, the Transmuter hardware delivers performance-per-Watt that is on average within an order of magnitude of fixed-function ASICs for the same kernels. The remainder of the chapter described a High-Level Language (HLL) library called TransPy that expands its usability to end users, by abstracting away the details of the reconfigurable hardware while exposing familiar Python-like semantics.

The last piece of work proposed a smart runtime framework called SparseAdapt that dynamically reconfigures the Transmuter system in response to both implicit (data-driven) and explicit (code-driven) phase changes during workload execution. The underlying predictive model uses machine learning to learn the mappings from a set of hardware performance counter values to the best set of hardware configuration parameters, and is deployed for dynamic reconfiguration at fine granularities ranging from hundreds to thousands of executed Floating-point operations (FLOPS) per core in the system. Cycle-level simulations showed $\sim 2\times$ boost in performance-per-Watt in the Energy-Efficient operational mode, and equal performance for ¾$^{\text{th}}$ the energy in the Power-Performance mode, over the static Transmuter configuration that is best (on average) for this problem domain. The methodology used in SparseAdapt itself delivers up to $2.8\times$ and $2.0\times$ gains in performance and energy-efficiency, respectively, over a state-of-the-art dynamic control scheme.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Nilmini Abeyratne, Reetuparna Das, Qingkun Li, Korey Sewell, Bharan Giridhar, Ronald G. Dreslinski, David Blaauw, and Trevor Mudge. "Scaling Towards Kilo-core Processors with Asymmetric High-Radix Topologies". In: *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2013, pp. 496–507. ISBN: 978-1-4673-5585-8.

[2] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. "Improving Performance of Sparse Matrix Dense Matrix Multiplication on Large-Scale Parallel Systems". In: *Parallel Computing* 59 (2016). ISSN: 0167-8191. DOI: `http://dx.doi.org/10.1016/j.parco.2016.10.001`. URL: `http://www.sciencedirect.com/science/article/pii/S0167819116301041`.

[3] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. "Pyramid Methods in Image Processing". In: *RCA Engineer* 29.6 (1984), pp. 33–41.

[4] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, et al. "Celerity: An Open Source RISC-V Tiered Accelerator Fabric". In: *Symp. on High Performance Chips (Hot Chips)*. 2017.

[5] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. "PX-CGRA: Polymorphic Approximate Coarse-Grained Reconfigurable Architecture". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 413–418.

[6] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. "X-CGRA: An Energy-Efficient Approximate Coarse-Grained Reconfigurable architecture". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019).

[7] Kadir Akbudak and Cevdet Aykanat. *Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures*. 2017.

[8] Michael Alfano, Bryan Black, Jeff Rearick, Joseph Siegel, Michael Su, and Julius Din. "Unleashing Fury: A New Paradigm for 3-D Design and Test". In: *IEEE Design & Test* 34.1 (2017), pp. 8–15.

[9] Lluc Alvarez, Llués Vilanova, Miquel Moreto, Marc Casas, Marc Gonzàlez, Xavier Martorell, Nacho Navarro, Eduard Ayguadé, and Mateo Valero. "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures". In: *Proceedings of the 42nd Annual Int'l Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: ACM, 2015, pp. 720–732. ISBN: 978-1-4503-3402-0. DOI: `10.1145/2749469.2750411`.

[10] Anima Anandkumar, Rong Ge, Daniel J. Hsu, Sham M. Kakade, and Matus Telgarsky. "Tensor Decompositions for Learning Latent Variable Models". In: *CoRR* abs/1210.7559 (2012). arXiv: `1210.7559`.

[11] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy. "2.9TOPS/W Reconfigurable Dense/Sparse Matrix-Multiply Accelerator with Unified INT8/INTI6/FP16 Datapath in 14NM Tri-Gate CMOS". In: *2018 IEEE Symposium on VLSI Circuits*. June 2018, pp. 39–40. DOI: `10.1109/VLSIC.2018.8502333`.

[12] Martin Anthony and Sean B Holden. "Cross-Validation for Binary Classification by Real-Valued Functions: Theoretical Analysis". In: *Proceedings of the eleventh annual conference on Computational learning theory*. 1998, pp. 218–229.

[13] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. "Alrescha: A Lightweight Reconfigurable Sparse-Computation Accelerator". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 249–260.

[14] Tuba Ayhan, Wim Dehaene, and Marian Verhelst. "A 128~2048/1536 point FFT Hardware Implementation with Output Pruning". In: *2014 22nd European Signal Processing Conference (EUSIPCO)*. IEEE. 2014, pp. 266–270.

[15] Ariful Azad, Aydin Buluç, and John Gilbert. "Parallel Triangle Counting and Enumeration Using Matrix Algebra". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE. 2015, pp. 804–811.

[16] David F Bacon, Rodric Rabbah, and Sunil Shukla. "FPGA Programming for the Masses". In: *Communications of the ACM* 56.4 (2013), pp. 56–63.

[17] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems". In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002*. IEEE. 2002, pp. 73–78.

[18] Nathan Bell, Steven Dalton, and Luke N Olson. "Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods". In: *SIAM Journal on Scientific Computing* 34.4 (2012), pp. C123–C152.

[19] Nathan Bell and Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, Dec. 2008.

[20] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. "Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model". In: *Theory of Computing Systems* 47.4 (Nov. 2010), pp. 934–962. ISSN: 1433-0490. DOI: `10.1007/s00224-010-9285-4`.

[21] 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* 26.4 (2006), pp. 52–60.

[22] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. "The gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (2011), pp. 1–7.

[23] Max Bramer. "Avoiding Overfitting of Decision Trees". In: *Principles of Data Mining* (2007), pp. 119–134.

[24] S Brin and Page L. "The Anatomy of a Large-Scale Hypertextual Web Search Engine". In: *7th Int'l WWW Conference* (1998).

[25] Ian Buck. "The Evolution of GPUs for General Purpose Computing". In: *Proceedings of the GPU Technology Conference 2010*. 2010, p. 11.

[26] Aydin Buluç and John R Gilbert. "On the Representation and Multiplication of Hypersparse Matrices". In: *IEEE Int'l Symposium on Parallel and Distributed Processing*. IPDPS '08. IEEE. 2008, pp. 1–11.

[27] Aydın Buluç and John R Gilbert. "The Combinatorial BLAS: Design, Implementation, and Applications". In: *The International Journal of High Performance Computing Applications* 25.4 (2011), pp. 496–509.

[28] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. "A Quantitative Study of Irregular Programs on GPUs". In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2012, pp. 141–151.

[29] Sergi Caelles, Kevis-Kokitsi Maninis, Jordi Pont-Tuset, Laura Leal-Taixé, Daniel Cremers, and Luc Van Gool. "One-Shot Video Object Segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 221–230.

[30] Benton Highsmith Calhoun, Joseph F Ryan, Sudhanshu Khanna, Mateja Putic, and John Lach. "Flexible Circuits and Architectures for Ultralow Power". In: *Proceedings of the IEEE* 98.2 (2010), pp. 267–282.

[31] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. “R-MAT: A Recursive Model for Graph Mining”. In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM. 2004, pp. 442–446.

[32] Web Chang. *Embedded Configurable Logic ASIC*. US Patent 6,260,087. July 2001.

[33] Anurat Chapanond, Mukkai S. Krishnamoorthy, and Bülent Yener. “Graph Theoretic and Spectral Analysis of Enron Email Data”. In: *Computational & Mathematical Organization Theory* 11.3 (Oct. 2005), pp. 265–281. ISSN: 1572-9346. DOI: `10.1007/s10588-005-5381-4`.

[34] 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* 52.1 (2016), pp. 127–138.

[35] T. Chen, S. Srinath, C. Batten, and G. E. Suh. “An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 55–67. DOI: `10.1109/MICRO.2018.00014`.

[36] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. “CGRA-ME: A Unified Framework for CGRA Modelling and Exploration”. In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2017, pp. 184–189.

[37] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. “DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 155–166.

[38] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. “Computational Methods for Dynamic Graphs”. In: *Journal of Computational and Graphical Statistics* 12.4 (2003), pp. 950–970.

[39] *Cortex-M0 - Arm Developer*. `https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0`.

[40] *Cortex-M4 - Arm Developer*. `https://developer.arm.com/products/processors/cortex-m/cortex-m4`.

[41] *cuSPARSE Library*. 2014.

[42] Marco Cuturi. “Sinkhorn Distances: Lightspeed Computation of Optimal Transport”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’13. 2013, pp. 2292–2300.

[43] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. “Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms”. In: *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: ACM, 2019, pp. 924–939. ISBN: 978-1-4503-6938-1.

[44] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*. Version 0.5.1. 2015.

[45] Steven Dalton, Luke Olson, and Nathan Bell. “Optimizing Sparse Matrix-Matrix Multiplication for the GPU”. In: *ACM Trans. Math. Softw.* 41.4 (Oct. 2015), 25:1–25:20. ISSN: 0098-3500. DOI: 10.1145/2699470.

[46] Maximilien Danisch, Oana Balalau, and Mauro Sozio. “Listing k-Cliques in Sparse Real-World Graphs”. In: *Proceedings of the 2018 World Wide Web Conference*. 2018, pp. 589–598.

[47] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rakesh K Gupta, Zhiru Zhang, Ronald G Dreslinski, Christopher Batten, and Michael B Taylor. “The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips”. In: *IEEE Micro* 38.2 (2018), pp. 30–41.

[48] Timothy A Davis and Yifan Hu. “The University of Florida sparse matrix collection”. In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[49] Yangdong Deng, Bo David Wang, and Shuai Mu. “Taming Irregular EDA Applications on GPUs”. In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. 2009, pp. 539–546.

[50] Chris HQ Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D Simon. “A Min-Max Cut Algorithm for Graph Partitioning and Data Clustering”. In: *Proceedings 2001 IEEE International Conference on Data Mining*. IEEE. 2001, pp. 107–114.

[51] Claire Donnat, Marinka Zitnik, David Hallac, and Jure Leskovec. “Learning Structural Node Embeddings via Diffusion Wavelets”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2018, pp. 1320–1329.

[52] Bryan Donyanavard, Tiago Mück, Amir M Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. “SOSA: Self-Optimizing learning with Self-Adaptive Control for Hierarchical System-on-Chip Management”. In:

Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 2019, pp. 685–698.

[53] R. Dorrance and D. Markovic. “A 190 GFLOPS/W DSP for Energy-Efficient Sparse-BLAS in Embedded IoT”. In: *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. June 2016, pp. 1–2. DOI: `10.1109/VLSIC.2016.7573527`.

[54] Richard Dorrance, Fengbo Ren, and Dejan Marković. “A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-BLAS on FPGAs”. In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM. 2014, pp. 161–170.

[55] Christophe Dubach, Timothy M Jones, and Edwin V Bonilla. “Dynamic Microarchitectural Adaptation using Machine Learning”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.4 (2013), pp. 1–28.

[56] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, and Michael FP O’Boyle. “A Predictive Model for Dynamic Microarchitectural Adaptivity Control”. In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2010, pp. 485–496.

[57] Iain S Duff, Michael A Heroux, and Roldan Pozo. “An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum”. In: *ACM Transactions on Mathematical Software (TOMS)* 28.2 (2002), pp. 239–267.

[58] E Swartzlander Earl Jr. “Systolic FFT Processors: Past, Present and Future”. In: *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP’06)*. IEEE. 2006, pp. 153–158.

[59] David Ediger, Rob McColl, Jason Riedy, and David A Bader. “Stinger: High Performance Data Structure for Streaming Graphs”. In: *2012 IEEE Conference on High Performance Extreme Computing*. IEEE. 2012, pp. 1–5.

[60] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark Silicon and the End of Multicore Scaling”. In: *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE. 2011, pp. 365–376.

[61] Nasim Farahini, Shuo Li, Muhammad Adeel Tajammul, Muhammad Ali Shami, Guo Chen, Ahmed Hemani, and Wei Ye. “39.9 GOPS/Watt Multi-Mode CGRA Accelerator for a Multi-Standard Basestation”. In: *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*. IEEE. 2013, pp. 1448–1451.

[62] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. “Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication”. In: *Pro-*

ceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. 2004, pp. 133–137.

[63] 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 G Dreslinski. "CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics". In: *Proceedings of the 58th Design Automation Conference (DAC)*. 2021, to appear.

[64] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. "Optimizing CUDA Code by Kernel Fusion: Application on BLAS". In: *The Journal of Supercomputing* 71.10 (2015), pp. 3934–3957.

[65] Florian Fricke, André Werner, Keyvan Shahin, and Michael Hübner. "CGRA Tool Flow for Fast Run-Time Reconfiguration". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2018, pp. 661–672.

[66] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. "Data Transfer Matters for GPU Computing". In: *2013 International Conference on Parallel and Distributed Systems*. IEEE. 2013, pp. 275–282.

[67] Noriyuki Fujimoto. "Dense Matrix-Vector Multiplication on the CUDA Architecture". In: *Parallel Processing Letters* 18.04 (2008), pp. 511–530.

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

[69] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. "Measuring and Modeling the Power Consumption of Energy-Efficient FPGA Coprocessors for GEMM and FFT". In: *Journal of Signal Processing Systems* 85.3 (Dec. 2016), pp. 307–323. ISSN: 1939-8018.

[70] Heiner Giefers, Peter Staar, Costas Bekas, and Christoph Hagleitner. "Analyzing the Energy-Efficiency of Sparse Matrix Multiplication on Heterogeneous Systems: A Comparative Study of GPU, Xeon Phi and FPGA". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2016, pp. 46–56.

[71] John R Gilbert, S. Reinhardt, and V B Shah. "A Unified Framework for Numerical and Combinatorial Computing". In: *Computing in Science & Engineering* 10.2 (), pp. 20–25.

[72] John R Gilbert, S. Reinhardt, and V B Shah. "High-performance Graph Algorithms from Parallel Sparse Matrices". In: *Proc. of the Int'l Workshop on Applied Parallel Computing* (2006).

[73] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. "PipeRench: A Reconfigurable Architecture and Compiler". In: *Computer* 33.4 (2000), pp. 70–77.

[74] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. "Understanding the performance of sparse matrix-vector multiplication". In: *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. PDP '08. IEEE. 2008.

[75] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. "Dynamically Specialized Datapaths for Energy Efficient Computing". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE. 2011, pp. 503–514.

[76] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. "GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging". In: *SIAM Journal on Scientific Computing* 37.1 (2015), pp. C54–C71. eprint: `http://dx.doi.org/10.1137/130948811`.

[77] Azzam Haidar, Mark Gates, Stan Tomov, and Jack Dongarra. "Toward a Scalable Multi-GPU Eigensolver via Compute-Intensive Kernels and Efficient Communication". In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 223–232.

[78] Tom R Halfhill. "Ambric's New Parallel Processor". In: *Microprocessor Report* 20.10 (2006), pp. 19–26.

[79] Václav Hapla, David Horák, and Michal Merta. "Use of Direct Solvers in TFETI Massively Parallel Implementation". In: *Applied Parallel and Scientific Computing: 11th Int'l Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers*. Ed. by Pekka Manninen and Per Öster. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 192–205. ISBN: 978-3-642-36803-5. DOI: `10.1007/978-3-642-36803-5_14`.

[80] 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: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. ACM, 2020.

[81] Alexander Friedrich Heinecke. "Cache Optimised Data Structures and Algorithms for Sparse Matrices". B.S. thesis. Technical University of Munich, Apr. 2008.

[82] Mark Horowitz. "Computing's Energy Problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE. 2014, pp. 10–14.

[83] Randall A Hughes and John D Shott. “The Future of Automation for High-Volume Wafer Fabrication and ASIC Manufacturing”. In: *Proceedings of the IEEE* 74.12 (1986), pp. 1775–1793.

[84] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. “POET: a Portable Approach to Minimizing Energy under Soft Real-Time Constraints”. In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2015, pp. 75–86.

[85] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. “Portable Multicore Resource Management for Applications with Performance Constraints”. In: *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE. 2016, pp. 305–312.

[86] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-025. June 2011.

[87] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. “Core Fusion: Accommodating Software Diversity in Chip Multiprocessors”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA ’07. San Diego, California, USA: ACM, 2007, pp. 186–197. ISBN: 978-1-59593-706-3.

[88] Satoshi Itoh, Pablo Ordejón, and Richard M Martin. “Order-N Tight-Binding Molecular Dynamics on Parallel Computers”. In: *Computer physics communications* 88.2-3 (1995), pp. 173–185.

[89] Satoshi Itoh, Pablo Ordejón, and Richard M. Martin. “Order-N Tight-Binding Molecular Dynamics on Parallel Computers”. In: *Computer Physics Communications* 88.2 (1995), pp. 173–185. ISSN: 0010-4655. DOI: `http://dx.doi.org/10.1016/0010-4655(95)00031-A`.

[90] 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,” High Performance Embedded Computing Conference (HPEC04”. In: *In High Performance Embedded Computing Conference (HPEC04*. 2004.

[91] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11–Seamless Operability between C++ 11 and Python*. 2017. URL: `https://github.com/pybind/pybind11`.

[92] 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: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2014, pp. 471–483.

[93] Kurtis T. Johnson, Ali R Hurson, and Behrooz Shirazi. “General-Purpose Systolic Arrays”. In: *Computer* 26.11 (1993), pp. 20–31.

[94] R. W. Johnson, C. H. Huang, and J. R. Johnson. “Multilinear Algebra and Parallel Programming”. In: *The Journal of Supercomputing* 5.2 (1991), pp. 189–217. ISSN: 1573-0484. DOI: `10.1007/BF00127843`.

[95] Norman P Jouppi. “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”. In: *Proceedings of the 17th Annual Int’l Symposium on Computer Architecture*. ISCA ’90. IEEE. 1990.

[96] 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: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA ’17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: `10.1145/3079856.3080246`. URL: `https://doi.org/10.1145/3079856.3080246`.

[97] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. “Marian: Fast Neural Machine Translation in C++”. In: *arXiv preprint arXiv:1804.00344* (2018).

[98] Haim Kaplan, Micha Sharir, and Elad Verbin. “Colored Intersection Searching via Sparse Rectangular Matrix Multiplication”. In: *Proceedings of the twenty-second annual symposium on Computational geometry*. 2006, pp. 52–60.

[99] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. “HyCUBE: A CGRA with Reconfigurable Single-Cycle Multi-Hop Interconnect”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[100] George Karypis, Anshul Gupta, and Vipin Kumar. “A Parallel Formulation of Interior Point Algorithms”. In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. Supercomputing ’94. Washington, D.C.: IEEE Computer Society Press, 1994, pp. 204–213. ISBN: 0-8186-6605-6.

[101] S. Kaxiras and G. Keramidas. “SARC Coherence: Scaling Directory Cache Coherence in Performance and Power”. In: *IEEE Micro* 30.5 (Sept. 2010), pp. 54–65. ISSN: 0272-1732. DOI: 10.1109/MM.2010.82.

[102] Stefanos Kaxiras and Alberto Ros. “Efficient, Snoopless, System-on-Chip Coherence”. In: *2012 IEEE International SOC Conference*. IEEE. 2012, pp. 230–235.

[103] John Kelm, Daniel Johnson, Matthew Johnson, Neal Crago, William Tuohy, Aqeel Mahesri, Steven Lumetta, Matthew Frank, and Sanjay Patel. “Rigel: An Architecture and Scalable Programming Interface for a 1000-Core Accelerator”. In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 140–151.

[104] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. “Mathematical foundations of the GraphBLAS”. In: *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. IEEE, 2016, pp. 1–9.

[105] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. “MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP”. In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), pp. 305–316.

[106] Martha Mercaldi Kim, John D. Davis, Mark Oskin, and Todd Austin. “Polymorphic On-Chip Networks”. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA ’08. IEEE Computer Society, 2008, pp. 101–112. ISBN: 978-0-7695-3174-8.

[107] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. “System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators”. In: *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE. 2008, pp. 123–134.

[108] 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: *Proceedings of the 39th ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 296–311. ISBN: 978-1-4503-5698-5.

[109] 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 SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 707–719.

[110] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[111] Hsiang-Tsung Kung. "Why Systolic Architectures?" In: *IEEE computer* 15.1 (1982), pp. 37–46.

[112] Ian Kuon and Jonathan Rose. "Measuring the Gap between FPGAs and ASICs". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 26.2 (2007), pp. 203–215.

[113] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA Architecture: Survey and Challenges*. Now Publishers Inc, 2008.

[114] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. "From Word Embeddings to Document Distances". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France, 2015, pp. 957–966.

[115] Georgi Kuzmanov and Mottaqiallah Taouil. "Reconfigurable Sparse/Dense Matrix-Vector Multiplier". In: *2009 International Conference on Field-Programmable Technology*. IEEE. 2009, pp. 483–488.

[116] Matthieu Latapy. "Main-memory triangle computations for very large (sparse (power-law)) graphs". In: *Theoretical computer science* 407.1-3 (2008), pp. 458–473.

[117] 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, 2004. ICPP 2004*. IEEE. 2004, pp. 169–176.

[118] Chang-Chi Lee, CP Hung, Calvin Cheung, Ping-Feng Yang, Chin-Li Kao, Dao-Long Chen, Meng-Kai Shih, Chien-Lin Chang Chien, Yu-Hsiang Hsiao, Li-Chieh Chen, Michael Su, Michael Alfano, Joe Siegel, Julius Din, and Bryan Black. "An Overview of the Development of a GPU with Integrated HBM on Silicon interposer". In: *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*. IEEE. 2016, pp. 1439–1444.

[119] Chang-Hwan Lee. “A Gradient Approach for Value Weighted Classification Learning in Naive Bayes”. In: *Knowledge-Based Systems* 85 (2015), pp. 71–79.

[120] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoung Choi. “FloRA: Coarse-Grained Reconfigurable Architecture with Floating-Point Operation Capability”. In: *2009 International Conference on Field-Programmable Technology*. IEEE. 2009, pp. 376–379.

[121] Eunji Lee, Hyojung Kang, Hyokyung Bahn, and Kang G Shin. “Eliminating Periodic Flush Overhead of File I/O with Non-Volatile Buffer Cache”. In: *IEEE Transactions on Computers* 65.4 (2014), pp. 1145–1157.

[122] Jure Leskovec and Rok Sosič. “SNAP: A General-Purpose Network Analysis and Graph-Mining Library”. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), pp. 1–20.

[123] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. “An Optimized Large-Scale Hybrid DGEMM Design for CPUs and ATI GPUs”. In: *Proceedings of the 26th ACM international conference on Supercomputing*. ACM. 2012, pp. 377–386.

[124] Cao Liang and Xinming Huang. “SmartCell: A Power-Efficient Reconfigurable Architecture for Data Streaming Applications”. In: *2008 IEEE Workshop on Signal Processing Systems*. IEEE. 2008, pp. 257–262.

[125] Jieun Lim, Nagesh B. Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. “Power Modeling for GPU Architectures Using McPAT”. In: *ACM Trans. Des. Autom. Electron. Syst.* 19.3 (June 2014), 26:1–26:24. ISSN: 1084-4309. DOI: `10.1145/2611758`. URL: `http://doi.acm.org/10.1145/2611758`.

[126] Colin Yu Lin, Ngai Wong, and Hayden Kwok-Hay So. “Design Space Exploration for Sparse Matrix-Matrix Multiplication on FPGAs”. In: *International Journal of Circuit Theory and Applications* 41.2 (2013), pp. 205–219.

[127] Leibo Liu, Dong Wang, Min Zhu, Yansheng Wang, Shouyi Yin, Peng Cao, Jun Yang, and Shaojun Wei. “An Energy-Efficient Coarse-Grained Reconfigurable Processing Unit for Multiple-Standard Video Decoding”. In: *IEEE Transactions on Multimedia* 17.10 (2015), pp. 1706–1720.

[128] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. “A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications”. In: *ACM Computing Surveys (CSUR)* 52.6 (2019), pp. 1–39.

[129] Weifeng Liu and Brian Vinter. “A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors”. In: *CoRR* abs/1504.05022 (2015).

[130] Weifeng Liu and Brian Vinter. "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 370–381.

[131] Beth Logan. "Mel Frequency Cepstral Coefficients for Music Modeling". In: *ISMIR*. Vol. 270. 2000, pp. 1–11.

[132] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 3431–3440.

[133] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. "Composite Cores: Pushing Heterogeneity into a Core". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2012, pp. 317–328.

[134] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. "Smart Memories: A Modular Reconfigurable Architecture". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 161–171. ISBN: 1-58113-232-8.

[135] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. "Sparse Matrix-Matrix Multiplication on Modern Architectures". In: *2012 19th International Conference on High Performance Computing*. IEEE. 2012, pp. 1–10.

[136] Tim Mattson, David A. Bader, Jonathan W. Berry, Aydin Buluç, Jack J. Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E. Leiserson, Andrew Lumsdaine, David A. Padua, Stephen Poole, Steven P. Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. "Standards for graph algorithm primitives". In: *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*. IEEE, 2013, pp. 1–2.

[137] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. "An Analysis of Neural Language Modeling at Multiple Scales". In: *arXiv preprint arXiv:1803.08240* (2018).

[138] Timothy N Miller, Xiang Pan, Renji Thomas, Naser Sedaghati, and Radu Teodorescu. "Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips". In: *IEEE International Symposium on High-Performance Comp Architecture*. IEEE. 2012, pp. 1–12.

[139] Asit K Mishra, Eriko Nurvitadhi, Ganesh Venkatesh, Jonathan Pearce, and Debbie Marr. "Fine-Grained Accelerators for Sparse Machine Learning Work-

loads". In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2017, pp. 635–640.

[140] Badri Narayan Mohapatra and Rashmita Kumari Mohapatra. "FFT and Sparse FFT Techniques and Applications". In: *2017 Fourteenth International Conference on Wireless and Optical Communications Networks (WOCN)*. IEEE. 2017, pp. 1–5.

[141] Gordon E Moore et al. *Cramming More Components onto Integrated Circuits*. 1965.

[142] Frank Mueller. "Pthreads Library Interface". In: *Florida State University* (1993).

[143] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. "CACTI 6.0: A Tool to Model Large Caches". In: *HP laboratories* 27 (2009), p. 28.

[144] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. "Introducing the Graph 500". In: (2010).

[145] Geoffrey Ndu. "Boosting Single Thread Performance in Mobile Processors using Reconfigurable Acceleration". PhD thesis. Oct. 2012.

[146] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. "Stream-Dataflow Acceleration". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 416–429. ISBN: 978-1-4503-4892-8.

[147] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. "A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine". In: *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE. 2015, pp. 109–116.

[148] *NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}*. Tech. rep. NVIDIA, 2015. (Visited on 04/04/2017).

[149] *NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Kepler^{TM} GK110*. Tech. rep. NVIDIA, 2012. URL: `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[150] Molly A O'Neil and Martin Burtscher. "Microarchitectural Performance Characterization of Irregular GPU Kernels". In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 130–139.

[151] 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: *2017 50th An-*

nual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE. 2017, pp. 41–54.

[152] Andreas Olofsson. *Silicon Compilers - Version 2.0.* 2018. URL: `http://www.ispd.cc/slides/2018/k2.pdf`.

[153] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald G Dreslinski, and Christophe Dubach. "SparseAdapt: Runtime Control for Sparse Linear Algebra on a Reconfigurable Accelerator". In: *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 2021, to appear.

[154] 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: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE. 2018, pp. 724–736.

[155] 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 G Dreslinski. "Transmuter: Bridging the Efficiency Gap Using Memory and Dataflow Reconfiguration". In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT).* 2020, pp. 175–190.

[156] Subhankar Pal, Kuba Kaszyk, Siying Feng, Björn Franke, Murray Cole, Michael O'Boyle, Trevor N Mudge, and Ronald G Dreslinski. "HetSim: Simulating Large-Scale Heterogeneous Systems using a Trace-driven, Synchronization and Dependency-Aware Framework". In: *2020 IEEE International Symposium on Workload Characterization (IISWC).* IEEE. 2020, pp. 13–24.

[157] Subhankar Pal, Dong-Hyeon Park, 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 N. Mudge, David T. Blaauw, Hun-Seok Kim, and Ronald G. Dreslinski. "A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm". In: *2019 Symposium on VLSI Circuits, Kyoto, Japan, June 9-14, 2019.* IEEE, 2019, p. 150.

[158] 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 N. Mudge, David T. 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: *Journal of Solid-State Circuits* 55.4 (2020), pp. 933–944.

[159] *Partial Reconfiguration User Guide UG702 (v13.3)*. English. Version UG702 (v13.3). Xilinx. Oct. 2011. URL: `%5Curl%7Bhttps://www.xilinx.com/support/documentation/sw%5C_manuals/xilinx13%5C_3/ug702.pdf%7D`.

[160] *Partial Reconfiguration User Guide UG909 (v2018.1)*. English. Version UG909 (v2018.1). Xilinx. Apr. 2018. URL: `%5Curl%7Bhttps://www.xilinx.com/support/documentation/sw%5C_manuals/%20xilinx2018%5C_1/ug909-vivado-partial-reconfiguration.pdf%7D`.

[161] Ardavan Pedram, Andreas Gerstlauer, and Robert A Van De Geijn. "A High-Performance, Low-Power Linear Algebra Core". In: *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE Int'l Conference on*. IEEE. 2011, pp. 35–42.

[162] Ardavan Pedram, John D. McCalpin, and Andreas Gerstlauer. "A Highly Efficient Multicore Floating-Point FFT Architecture Based on Hybrid Linear Algebra/FFT Cores". In: *Journal of Signal Processing Systems* 77.1 (Oct. 2014), pp. 169–190. ISSN: 1939-8115.

[163] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine Learning in Python". In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[164] Gerald Penn. "Efficient Transitive Closure of Sparse Matrices over Closed Semirings". In: *Theoretical Computer Science* 354.1 (2006), pp. 72–81.

[165] Gerald Penn. "Efficient Transitive Closure of Sparse Matrices over Closed Semirings". In: *Theoretical Computer Science* 354.1 (2006), pp. 72–81.

[166] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. "Using SimPoint for Accurate and Efficient Simulation". In: *ACM SIGMETRICS Performance Evaluation Review* 31.1 (2003), pp. 318–319.

[167] Paula Petrica, Adam M Izraelevitz, David H Albonesi, and Christine A Shoemaker. "Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 2013, pp. 13–23.

[168] Nathaniel Pinckney, Matthew Fojtik, Bharan Giridhar, Dennis Sylvester, and David Blaauw. "Shortstop: An On-Chip Fast Supply Boosting Technique". In: *2013 Symposium on VLSI Circuits*. IEEE. 2013, pp. C290–C291.

[169] Kara KW Poon, Steven JE Wilton, and Andy Yan. “A Detailed Power Model for Field-Programmable Gate Arrays”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 10.2 (2005), pp. 279–302.

[170] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros Voulgaris, and Josep Torrellas. “Yukta: Multilayer Resource Controllers to Maximize Efficiency”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 505–518.

[171] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. “Plasticine: A Reconfigurable Architecture for Parallel Patterns”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 389–402.

[172] Michael O Rabin and Vijay V Vazirani. “Maximum Matchings in General Graphs through Randomization”. In: *Journal of algorithms* 10.4 (1989), pp. 557–567.

[173] Gokul Subramanian Ravi and Mikko H Lipasti. “CHARSTAR: Clock Hierarchy aware Resource Scaling in Tiled Architectures”. In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 147–160.

[174] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jüngel, and Siegfried Selberherr. “ViennaCL-Linear Algebra Library for Multi- and Many-Core Architectures”. In: *SIAM Journal on Scientific Computing* 38.5 (2016), S412–S439. DOI: 10.1137/15M1026419.

[175] Tim Salimans, Han Zhang, Alec Radford, and Dimitris Metaxas. “Improving GANs using optimal transport”. In: *arXiv preprint arXiv:1803.05573* (2018).

[176] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. “Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture”. In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* IEEE. 2003, pp. 422–433.

[177] Anderson Luiz Sartor, Pedro Henrique Exenberger Becker, Stephan Wong, Radu Marculescu, and Antonio Carlos Schneider Beck. “Machine Learning-Based Processor Adaptability Targeting Energy, Performance, and Reliability”. In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2019, pp. 158–163.

[178] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. “Navigating the Maze of Graph Analytics Frameworks us-

ing Massive Graph Datasets". In: *Proceedings of the 2014 ACM SIGMOD Int'l conference on Management of data*. ACM. 2014, pp. 979–990.

[179] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi". In: *CoRR* abs/1302.1078 (2013).

[180] Fabian Schuiki, Michael Schaffner, and Luca Benini. "NTX: An Energy-Efficient Streaming Accelerator for Floating-Point Generalized Reduction Workloads in 22 nm FD-SOI". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 662–667.

[181] *Seventh Green Graph 500 List*. 2016. URL: `http://green.graph500.org/lists.php`.

[182] Korey Sewell, Ronald G. Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Ross Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F. Wenisch, Dennis Sylvester, David T. Blaauw, and Trevor N. Mudge. "Swizzle-Switch Networks for Many-Core Systems". In: *IEEE J. Emerg. Sel. Topics Circuits Syst.* 2.2 (2012), pp. 278–294.

[183] Muhammad Shafique and Siddharth Garg. "Computing in the Dark Silicon Era: Current Trends and Research Challenges". In: *IEEE Design & Test* 34.2 (2016), pp. 8–23.

[184] Viral B Shah. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. University of California, Santa Barbara, 2007.

[185] Anton Shilov. *JEDEC Publishes HBM2 Specification*. 2016.

[186] Aaron Smith. *6 New Facts about Facebook*. Feb. 2014. URL: `http://www.pewresearch.org/fact-tank/2014/02/03/6-new-facts-about-facebook/`.

[187] Anuraag Soorishetty, Jian Zhou, Subhankar Pal, David Blaauw, H Kim, Trevor Mudge, Ronald Dreslinski, and Chaitali Chakrabarti. "Accelerating Linear Algebra Kernels on a Massively Parallel Reconfigurable Architecture". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 1558–1562.

[188] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator based on Row-wise Product". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 766–780.

[189] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. "Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense

Tensor Computations”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 689–702.

[190] Samuel Steffl and Sherief Reda. “LACore: A Supercomputing-Like Linear Algebra Accelerator for SoC-Based Designs”. In: *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 137–144.

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

[192] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.

[193] Peter D Sulatycke and Kanad Ghose. “Caching-Efficient Multithreaded Fast Multiplication of Sparse Matrices”. In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. Mar. 1998, pp. 117–123. DOI: 10.1109/IPPS.1998.669899.

[194] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. “GraphMat: High Performance Graph Analytics made Productive”. In: *Proceedings of the VLDB Endowment* 8.11 (2015), pp. 1214–1225.

[195] Karthik T Sundararajan, Timothy M Jones, and Nigel P Topham. “The Smart Cache: An Energy-Efficient Cache Architecture through Dynamic Adaptation”. In: *International Journal of Parallel Programming* 41.2 (2013), pp. 305–330.

[196] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. “Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 575–587.

[197] Stephen J Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya, Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin, Robert Chappell, Ronak Singhal, and Hong Wang. “Post-Silicon CPU Adaptation Made Practical Using Machine Learning”. In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2019, pp. 14–26.

[198] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jae W. Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker

Strumpen, Matthew I. Frank, Saman P. Amarasinghe, and Anant Agarwal. “The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs”. In: *IEEE Micro* 22.2 (2002), pp. 25–35.

[199] Anand Tech. *NVIDIA Launches Tesla K40*. 2013.

[200] Vaishali Tehre, Pankaj Agrawal, and RV Kshrisagar. “Implementation of Fast Fourier Transform Accelerator on Coarse Grain Reconfigurable Architecture”. In: *International Journal of Computer Science and Network (IJCSN)* (2016), pp. 955–959.

[201] Robert A Van De Geijn and Jerrell Watts. “SUMMA: Scalable Universal Matrix Multiplication Algorithm”. In: *Concurrency: Practice and Experience* 9.4 (1997), pp. 255–274.

[202] Stijn Marinus Van Dongen. “Graph Clustering by Flow Simulation”. PhD thesis. 2000.

[203] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S Schreiber. “Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 197–210.

[204] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. “ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks”. In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 13–26.

[205] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, et al. “RaPiD: AI Accelerator for Ultra-low Precision Training and Inference”. In: *Training* 4.8 (), p. 16.

[206] Manish Verma, Lars Wehmeyer, Peter Marwedel, and Peter Marwedel. “Cache-Aware Scratchpad Allocation Algorithm”. In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*. DATE ’04. IEEE Computer Society, 2004, pp. 21264–. ISBN: 0-7695-2085-5.

[207] Kizheppatt Vipin and Suhaib A Fahmy. “FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications”. In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–39.

[208] D. Voitsechov, O. Port, and Y. Etsion. “Inter-Thread Communication in Multithreaded, Reconfigurable Coarse-Grain Arrays”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 42–54. DOI: 10.1109/MICRO.2018.00013.

[209] Vuduc, Richard W and Moon, Hyun Jin. "Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block structure". In: Springer. 2005.

[210] Donglin Wang, Xueliang Du, Leizu Yin, Chen Lin, Hong Ma, Weili Ren, Huijuan Wang, Xingang Wang, Shaolin Xie, Lei Wang, Zijun Liu, Tao Wang, Zhonghua Pu, Guangxin Ding, Mengchen Zhu, Lipeng Yang, Ruoshan Guo, Zhiwei Zhang, Xiao Lin, Jie Hao, Yongyong Yang, Wenqin Sun, Fabiao Zhou, NuoZhou Xiao, Qian Cui, and Xiaoqin Wang. "MaPU: A Novel Mathematical Computing Architecture". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), pp. 457–468.

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

[212] Yan Xiong, Jian Zhou, Subhankar Pal, David Blaauw, Hun-Seok Kim, Trevor Mudge, Ronald Dreslinski, and Chaitali Chakrabarti. "Accelerating Deep Neural Network Computation on a Low Power Reconfigurable Architecture". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5.

[213] Ichitaro Yamazaki and Xiaoye S Li. "On Techniques to Improve Robustness and Scalability of a Parallel Hybrid Linear Solver". In: *International Conference on High Performance Computing for Computational Science*. Springer. 2010, pp. 421–434.

[214] Ichitaro Yamazaki and Xiaoye S Li. "On Techniques to Improve Robustness and Scalability of a Parallel Hybrid Linear Solver". In: *International Conference on High Performance Computing for Computational Science*. Springer. 2010, pp. 421–434.

[215] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. "Balanced Sparsity for Efficient DNN Inference on GPU". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2019, pp. 5676–5683.

[216] Leonid Yavits and Ran Ginosar. "Sparse Matrix Multiplication on CAM Based Accelerator". In: *CoRR* abs/1705.09937 (2017). URL: `http://arxiv.org/abs/1705.09937`.

[217] Fanghua Ye, Chuan Chen, and Zibin Zheng. "Deep Autoencoder-like Nonnegative Matrix Factorization for Community Detection". In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 2018, pp. 1393–1402.

[218] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. "Scalpel: Customizing DNN pruning to the Underlying

Hardware Parallelism". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 548–560.

[219] Raphael Yuster and Uri Zwick. "Detecting Short Directed Cycles using Rectangular Matrix Multiplication and Dynamic Programming." In: *SODA*. Vol. 4. Citeseer. 2004, pp. 254–260.

[220] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2015, pp. 161–170.

[221] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. "SpArch: Efficient Architecture for Sparse Matrix Multiplication". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 261–274.

[222] Qiuling Zhu, Tobias Graf, H Ekin Sumbul, Larry Pileggi, and Franz Franchetti. "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware". In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2013, pp. 1–6.

[223] Paul S Zuchowski, Christopher B Reynolds, Richard J Grupp, Shelly G Davis, Brendan Cremen, and Bill Troxel. "A Hybrid ASIC and FPGA Architecture". In: *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.* IEEE. 2002, pp. 187–194.