EVOLUTIONARY HARDWARE SPECIALIZATION FOR MODERN VECTOR AND MATRIX ARCHITECTURES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by
Tuan Quang Ta
August 2023

EVOLUTIONARY HARDWARE SPECIALIZATION FOR
MODERN VECTOR AND MATRIX ARCHITECTURES

Tuan Quang Ta, Ph.D.
Cornell University 2023

With the slowdown of Moore's Law and the end of Dennard Scaling, computer architects have embraced *specialization* as the main way forward for continuing performance and efficiency growth previously made through traditional technology scaling. Specialization comes in several forms including *application specific specialization*, *domain specific specialization*, and *parallel pattern specific specialization*. This emergence of hardware specialization has pushed the chip industry towards integrating a *sea of heterogeneous specialized hardware units*, each with its own specialized program abstraction, into a single system on chip (SoC). However, given certain area, power, and budget constraints, there is limited room for the number of specialized hardware units possibly integrated into an SoC. One viable solution is to unify multiple kinds of specialization under the same program abstraction and in the same hardware (e.g., GPGPUs). This unifying approach essentially lowers area costs by trading off the optimality of program abstractions and hardware implementations for individual program patterns.

In this thesis, I explore another specialization approach called *evolutionary specialization* that supports multiple types of specialization in the same hardware. The evolutionary specialization refers to starting from an optimal abstraction and micro-architecture for one program pattern and gradually adding a minimal set of hardware changes to the existing micro-architecture to support additional program patterns without changing their optimal abstractions. The thesis makes a case for the evolutionary specialization through two novel architectures: big.VLITTLE and SparseZipper. The big.VLITTLE architecture evolves a multi-little-core system to efficiently support both single-program multiple-data (SPMD) and single-instruction multiple-data (SIMD) program patterns. The SparseZipper architecture minimally extends a modern matrix architecture specialized for a dense general matrix multiplication (GEMM) pattern to support a sparse GEMM pattern.

BIOGRAPHICAL SKETCH

Tuan Ta was born on September 24, 1992 to Ta Dac Ngo and Nguyen Thi Thu Van. He is the youngest child with a sister six years older. During his childhood, Tuan showed his interests in science and technology, especially math and programming. He picked up basic programming skills in Pascal and then C languages in his middle school. He joined Hanoi-Amsterdam High School for the Gifted and took advanced math and programming classes.

Tuan was accepted to the Hanoi University of Science and Technology to study computer and information sciences. During his first year at the university, he became interested in pursuing higher education outside Vietnam to satisfy his desire to explore the world. He started applying for colleges in the US and got accepted to the University of Mississippi with a full-tuition scholarship. After the first year of struggling with drastic cultural differences, he adapted to the new environment and started making good academic progress. In his sophomore year, Tuan joined a research group led by Professor Byunghyun Jang. This undergraduate research experience set him up for his later academic career and a strong interest in computer architecture.

Tuan was admitted to the Ph.D. program in the School of Electrical and Computer Engineering at Cornell University in 2017, and he joined Professor Christopher Batten's group. Throughout the next six years, he was fortunate to collaborate with smart and hardworking students in the group to advance state-of-the-art research in the field of computer architecture. He initially worked on a smart sharing architecture led by Dr. Shreesha Srinath, a heterogeneous cache coherence project led by Dr. Moyang Wang, and then EVE project led by Dr. Khalid Al-Hawaj. Tuan gained first-hand experience of taping out a chip in CIFER project led by a research group at Princeton University. He then took the lead in two projects: big.VLITTLE and SparseZipper that constitute the main portion of this dissertation.

In his free time during the Ph.D. journey, Tuan enjoyed playing badminton. During the tough COVID years, he also learned how to play other racket sports including squash, tennis, pickle ball, and ping pong. He was interested in checking out many beautiful hiking trails and waterfalls that the nature in upstate New York offers.

Tuan is very thankful for his Ph.D. experience despite numerous challenges during the journey. The great mentorship from his advisor, invaluable moments with his friends, and most importantly the endless support from his family helped him persist throughout the six years of his Ph.D. journey and learn how to become a good person and researcher.

This document is dedicated to my parents and my beloved wife Thuy Nguyen.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support of many people in my life. I am thankful for all encouragement, advice, and help that I received during my long Ph.D. journey.

First and foremost, I would like to thank my parents for always being on my side whenever I go through difficulties in life and encouraging me to persist and complete my "decade-long" study. Despite being far away from home, I can always feel your support in our daily phone calls. My parents have taught me the importance of hard work in achieving any meaningful goals in life. I thank my wife, Thuy Nguyen, for being with me through happiest and toughest moments during my entire study in the US. My journey in a foreign country would have not been meaningful without you going with me in the past 11 years. I am thankful for our marriage, and I will continue making you feel happy in the future. I would like to thank my sister, Trang Ta, for understanding and supporting my decisions in life.

This dissertation would not have been possible without the guidance of my committee: Professor Christopher Batten, Professor José Martínez, and Professor Adrian Sampson. My great advisor, Professor Christopher Batten, has taught me the importance of low-level details in doing good research, challenged me to stand on my own feet to argue for my research ideas, put me out of my comfort zone by assigning me to lead a team in a tapeout project, and trusted me to finish the final stretch of my Ph.D. journey. More importantly, his honest and constructive criticisms made me become a better researcher. I would like to acknowledge the rest of my committee: Professor José Martínez and Professor Adrian Sampson for your feedback to improve my research and comments challenging me to think differently about my research.

I am deeply thankful for working with my colleagues in BRG group. Shreeshra Srinath convinced me about the future of RISC-V and always challenged me to look for ambitious research ideas. Christopher Torng taught me how to organize my research and inspired me to make it more productive using automation tools and effective note taking. Moyang Wang taught me about work-stealing runtime and not giving up until the last moment before a paper submission deadline. Shunning Jiang taught me how to use PyMTL, and I am thankful for our discussions about the BRG in-order core design. I appreciate my collaborations with Khalid Al-Hawaj in various projects. He told me about common pitfalls in a Ph.D. journey and helped me through important milestones. He was patient with me explaining ideas and willing to provide constructive criticisms to make them better. I truly enjoyed working with him in the second half of my Ph.D. journey. Lin

Cheng always has his magic in finding out tricky bugs. Peitian Pan, I enjoyed discussing research ideas with you and thank you for helping with using PyMTL. I would like to acknowledge Yanghui Ou for his generosity and kindness in helping me in various projects. I hope the last part of your Ph.D. will go smoothly. Nick Cebry, I enjoyed working with you in the big.VLITTLE project. Hopefully, your collaborations with me and Khalid help you later in your Ph.D. process. I am thankful for working with Dr. Shady Agwa and his deep VLSI knowledge. I would also like to thank Courtney Golden for helping with the big.VLITTLE project. I hope you will do well in your own Ph.D. journey at MIT. I thank Eric Hall, Xiaoyu Yan, and Eric Tang for helping with various projects in this dissertation.

I would like to thank my collaborators. I appreciate Ang Li and August Ning for hosting me at Princeton University to test out the CIFER chip and helping me evaluate it remotely. I enjoyed our conversations at conferences, and hopefully I can see you in another conference in the near future. I thank the rest of CIFER team for making the chip tapeout happen. I would also like to acknowledge my collaborators at Arm Research: Alex Rico for mentoring me during my internship, José Joao for giving me comments on the big.VLITTLE project, Tiago Muck for sharing a pre-published version of CHI implementation in gem5, and Joshua Randall for mentoring and helping me in the SparseZipper project.

I would like to thank my friends at the Computer Systems Laboratory (CSL). CSL introduced me to a wonderful group of people to share ideas, have meals together, and enjoy life after work. Sachille Atapattu and Helena Caminal, I enjoyed our time together hiking, kayaking, cooking, and having fun. Philip Bedoukian, although we could not schedule our B-exam on the same day, I will miss our weekend meals at Asia cuisine. Khalid Al-Hawaj, Yichi Zhang, Chanhui Deng, and Professor Zhiru Zhang are my badminton "buddies" from CSL.

I also would like to thank my friends outside CSL for great moments outside work during my Ph.D. process. Thank Chethani Athukorala and Sunil Atapattu for being my good friends. I thank the badminton team including Akula Sai Pratyush, Chao-Ming Jian, Ray Yu, Kaushalendra Singh, Yun Liu, Jin, and Jun Lin for great badminton games and early morning training sessions. Playing badminton with you helped me remain sane during my Ph.D. journey.

I would like to thank all people that helped me get to where I am today. I thank Professor Byunghyun Jang for getting me started in doing research and strongly supporting me throughout the whole process. My time in the HEROES group at the University of Mississippi was wonderful.

I would like to acknowledge Dr. David Troendle for teaching so many things about hardware, system programming, and operating systems. You are my role model for me to keep learning, working hard, and being creative regardless of age. I would like to thank Dr. Kyoshin Choo for helping me in my first ever research project. I thank my mentors at the National Center for Atmospheric Research for introducing me to scientific programming during my first internship. I thank Bradford Beckmann and Anthony Gutierrez for mentoring me during my time at AMD Research. I would like to thank my English teacher, Nghiem Ngo, for inspiring me to look for opportunities outside Vietnam and guiding me to apply for colleges in the US.

This work was supported in part by NSF PPoSS Award #2118709, NSF SHF Award #2008471, DARPA POSH Award #FA8650-18-2-7852, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CMP	chip multiprocessor
SIMD	single-instruction multiple-data
SPMD	single-program multiple-data
GEMM	general matrix multiplication
SpGEMM	sparse matrix-matrix multiplication
GPP	general-purpose processor
SoC	system on chip
GPGPU	general-purpose graphical processing unit
eFPGA	embedded FPGA
CIFER	coherent interconnect and FPGA enabling reuse
AVX	(Intel) Advanced Vector Extensions
SVE	(Arm) Scalable Vector Extension
RVV	RISC-V Vector Extension
AMX	(Intel) Advanced Matrix Extensions
SME	(Arm) Scalable Matrix Extension
ISA	instruction set architecture
FL	functional level
CL	cycle level
RTL	register-transfer level
VLSI	very-large-scale integration
RF	register file
RAT	register alias table
ROB	re-order buffer
VRF	vector register file
ALU	arithmetic and logical unit
LDQ	load queue
STQ	store queue
LSQ	load-store queue
VCU	vector control unit
VXU	vector cross-element unit
VMU	vector memory unit
VLU	vector load unit
VSU	vector store unit
SRAM	static random access memory
DRAM	dynamic random access memory
I$	level-1 instruction cache
D$	level-1 data cache
L2	level-2 cache
LLC	last-level cache
CSR	compressed sparse row
CSC	compressed sparse column
COO	coordinate format
ESC	expand-sort-compress algorithm

CHAPTER 1
INTRODUCTION

With the slowdown of Moore's Law [Dub05] and the end of Dennard Scaling [DKM+02], computer architects have embraced *specialization* as the main way forward for continuing performance and efficiency growth previously made through traditional technology scaling [HP19, SB22]. Specialization comes in several forms including *application specific specialization* [LLW+06, LL18, MSS+15], *domain specific specialization* [DTH20, JYPP18, CKES16, QHS+13, TBD18], and *parallel pattern specific specialization* [PZK+18, Bat10, CHM08, KJT+17, SIT+14]. This emergence of hardware specialization has pushed the chip industry towards integrating a *sea of heterogeneous specialized hardware units*, each with its own specialized program abstraction, into a single system on chip (SoC) [HR21, SB22]. However, given certain area, power, and budget constraints, there is limited room for the number of specialized hardware units possibly integrated into an SoC. One viable solution is to unify multiple kinds of specialization under the same program abstraction and in the same hardware. For example, a general-purpose graphics processing unit (GPGPU) has evolved from a graphics-only accelerator to decently accelerate other application domains (e.g., machine learning and graph analytics) and parallel patterns (e.g., data-level and thread-level parallelism) [KDK+11, LNOM08]. This unifying approach essentially lowers area costs by trading off the optimality of program abstractions and hardware implementations for individual program patterns.

In this thesis, I explore another specialization approach called *evolutionary specialization* that supports multiple types of specialization in the same hardware. The evolutionary specialization approach starts from an optimal abstraction and micro-architecture for one program pattern and gradually adds a minimal set of hardware changes to the existing micro-architecture to support additional program patterns without impacting their optimal abstractions. This thesis motivates the evolutionary specialization using two novel architectures: big.VLITTLE and SparseZipper. In big.VLITTLE architectures, a multi-little-core system specialized for single-program multiple-data (SPMD) pattern is reconfigured to support single-instruction multiple-data (SIMD) pattern with minimal area overhead to the original hardware. In SparseZipper architectures, a programmable systolic-array-based micro-architecture designed for accelerating dense matrix-matrix multiplication (GEMM) is repurposed so that it can efficiently perform GEMM on sparse matrices as well.

This chapter begins by discussing the SPMD and SIMD patterns and motivating the need for their support in modern systems. Then, I present a recent trend towards specialization for dense and sparse GEMM patterns and how commonly those patterns exist in traditional and emerging workload domains. I finally introduce the concept of evolutionary specialization before outlining the key contributions of this thesis.

1.1 Specialization for SPMD and SIMD Patterns

SPMD and SIMD are by far two of the most commonly used patterns for structuring parallel programs [MSM05]. In an SPMD program, all processing elements (PEs) execute multiple streams of instructions derived from the same program (i.e., single program) in parallel, and each PE independently operates on its own set of data (i.e., multiple data). By sharing the same program across all PEs, the SPMD pattern makes managing PEs and their interactions less complex for programmers compared to executing different programs on multiple PEs (e.g., multiple-program multiple-data or MPMD pattern). The SPMD pattern is well-suited for expressing certain task-level parallelism in which different tasks share similar computations. In contrast, the SIMD pattern expresses all parallelism in terms of the data (i.e., data-level parallelism). In an SIMD program, all PEs execute the same stream of instructions (i.e., single instruction) in parallel on their own sets of data (i.e., multiple data). For regular data-parallel programs, the SIMD pattern simplifies managing parallel executions of all PEs by having only one instruction stream. The SIMD pattern is most efficient for expressing regular data-level parallelism in which computations on multiple subsets of data are the same.

1.1.1 Specialization for SPMD Pattern

An execution model for the SPMD pattern typically maps multiple PEs to logical threads, each running on a virtual processor. A thread holds an execution context of its corresponding PE (e.g., which instruction in a program the PE is executing). A thread scheduler may schedule multiple threads to run in parallel, and a thread may communicate with others via a synchronization mechanism. A multi-thread scalar instruction set architecture (ISA) is a specialized abstraction for the SPMD execution model. In such ISA, each thread has a private set of architectural states (e.g., program counters and registers). Each thread executes a sequence of scalar instructions

(e.g., arithmetic, memory, and control-flow instructions) independently from other threads. Those ISAs typically support inter-thread communications via either a shared memory (e.g., using atomic instructions) or a message passing mechanism (e.g., using message sending and receiving instructions).

A multi-thread scalar ISA is commonly implemented in shared-memory multi-core processors. Each core in such a multi-core processor runs one or multiple hardware threads (e.g., in simultaneous multithreading or SMT processors) that are mapped to logical threads in the SPMD execution model. Each hardware thread has its private program counter tracking its currently executed instruction and a set of registers for storing its private local data. A core can be implemented using a simple in-order or a complex out-of-order pipeline. Different cores in a processor can have the same implementation (i.e., homogeneous multi-core processors such as TILE64 [BEA+08] and Ampere Altra [Whe20]) or different implementations (i.e., heterogeneous multi-core processors such as Samsung Exynox [Gwe14b] and Qualcomm Snapdragon [Gwe14a]). The number of cores in a multi-core processor may vary from a few to hundreds of cores (e.g., the 1024-core Adapteva Epiphany-V [Olo16]). By integrating more but simpler cores into a single system (i.e., many-core processors with energy-efficient little cores), we increase its specialization for the SPMD pattern. For inter-thread communications via a shared memory, a multi-core processor is typically implemented with a cache coherence support for sharing data among threads.

1.1.2 Specialization for SIMD Pattern

The SIMD pattern structures parallel programs in multiple data segments (e.g., parts of a one- or two-dimensional array) and performs the same instruction stream across all segments. A typical SIMD execution model maps data segments to virtual lanes sharing a global control (i.e., each lane is an abstract execution unit). All lanes perform the same instruction stream on their local data segments, so they always execute in lock steps. Vector ISAs (e.g., Intel AVX [int12], Arm SVE [SBB+17], and RISC-V vector extension [RIS21]) are specialized for the SIMD pattern. A vector length (i.e., the number of virtual lanes) can either be fixed (e.g., in packed-SIMD ISA such as Intel AVX 512) or variable (e.g., in Arm SVE and RISC-V vector extension). In such ISA, vector registers are used to store per-lane local data while vector instructions perform operations on the data. Vector memory instructions move data between vector registers and memory. Predication (i.e., execution of some virtual lanes are masked off) is often supported to express control diver-

gence between virtual lanes. Modern vector ISAs typically support cross-lane vector instructions (e.g., vector reduction) for communicating data across virtual lanes.

A vector micro-architecture primarily consists of a physical vector register file, multiple physical lanes of arithmetic execution units, a vector memory unit, and a control core. The number of data elements in a physical vector register is the hardware vector length. The hardware vector length can be equal to or greater than the number of physical lanes (i.e., a lane performing the same vector operation on different data elements over one or multiple cycles). There is a control core managing the vector execution of all physical lanes. Regular vector memory accesses (e.g., unit-stride vector load) are typically issued to memory ahead of time (i.e., decoupled access-execute mechanism) to hide memory latency. A vector ISA can be implemented as either short-vector units tightly integrated into a general-purpose processor (e.g., Intel Knights Landing [SGC+16]) or long-vector engines decoupled from their control processors (e.g., Cray-BlackWidow [Abt07]).

1.1.3 The Need for Both SPMD and SIMD Patterns

The SPMD pattern is highly flexible that it can support various types of parallelism such as loop and fork/join parallelism with either regular or irregular control flows and memory access patterns. This flexibility is realized in multi-thread scalar ISAs and shared-memory multi-core micro-architectures by decoupling the execution of multiple threads. However, this decoupling becomes inefficient when the SPMD pattern is used to support regular data parallelism (i.e., with regular control flows and memory access patterns) since different threads redundantly perform the same control operations and memory accesses. In contrast, the SIMD pattern is relatively rigid as it targets mainly regular data-level parallelism. Vector ISAs and micro-architectures are specialized for exploiting the same control and regular memory accesses across vector lanes to minimize redundancy in control and maximize the utilization of physical lanes in vector hardware. However, the rigidity in both vector abstraction and hardware leads to a poor utilization of hardware resources when the SIMD pattern is applied to highly irregular problems with control-flow divergence and irregular memory access patterns across vector lanes.

Parallel workloads often exhibit different kinds of parallelism (e.g., task-level and data-level parallelism) with various levels of regularity in control flows and memory accesses, which needs both SPMD and SIMD patterns for maximizing the overall efficiency. For example, graph analytics applications typically perform computation on irregular data structures (e.g., trees and

graphs) which results in highly irregular data-dependent control flows and random memory accesses [SB13a, HN07, HKOO11]. Therefore, hardware specialization for the SPMD pattern is in general more suitable for graph analytics workloads. In contrast, scientific computing workloads [SRS+12, Gal96], which typically operate on regular array and matrix data structures, often exhibit ample amount of data-level parallelism. Therefore, it is more efficient to execute such workloads on hardware specialized for the SIMD pattern. Modern hardware systems need to support both kinds of specialization for the SPMD and SIMD patterns to target a variety of workloads having those patterns.

1.2 Specialization for Dense and Sparse GEMM Patterns

General matrix multiply (GEMM) is the key building block in many application domains such as machine learning, graph analytics, and scientific computing. Based on the density of input matrices (i.e., the fraction of non-zero elements in a matrix), there are two common program patterns for GEMM: dense and sparse. Dense GEMM performs a matrix-multiply operation on matrices stored in a two-dimensional array data structure in which all matrix elements including zero values are represented. In contrast, the sparse GEMM pattern refers to multiplying matrices stored in a compact data structure in which only non-zero values are represented. Some widely used compact matrix data structures include compressed sparse row (CSR), compressed sparse column (CSC), and coordinate (COO) formats. Due to the difference in data formats for representing dense and sparse matrices, typical accelerators for dense and sparse GEMM patterns are quite different in both their abstractions and hardware implementations. This section discusses architectural specialization for the two patterns.

1.2.1 Specialization for Dense GEMM Pattern

GEMM is a key building block in traditional and emerging workloads. To improve its performance and efficiency, architects have been designing and integrating accelerators for the dense GEMM pattern in modern systems. For example, Google introduced its Tensor Processing Unit (TPU) as a co-processor next to a general-purpose CPU for accelerating training and inference kernels in machine learning workloads [JYP+17, Tie20, JYK+20, JKL+23a]. At the heart of TPU is a

large matrix-matrix multiply unit that significantly improves the performance and energy efficiency compared to contemporary CPUs and GPGPUs. NVIDIA also integrates tensor cores specialized for multiplying and adding matrices in its recent GPUs [CGG+21a].

The need for accelerating GEMM has pushed specialization for the dense GEMM pattern further into modern general-purpose CPU instruction sets as well. Arm recently released its Scalable Matrix Extension (SME) that introduces a new instruction performing an outer product of two vectors and accumulating its results into a new two-dimensional accumulator register state [arm23]. IBM took a similar approach in its Matrix-Multiple Assist (MMA) extension for the Power ISA [ibm23]. Intel introduced a new Advanced Matrix Extension (AMX) that adds several two-dimensional matrix register states called tile registers and a new matrix-matrix multiply instruction performing a matrix multiplication on two input tile registers [int23b, NMM+22]. The RISC-V community is also working on a matrix extension proposal [ris23] that is similar to Intel AMX's approach.

Regardless of programming abstractions, specialization for the dense GEMM pattern is typically implemented in hardware using a two-dimensional systolic array of multiply-add processing elements (PEs) [JYP+17, JQS+21, NMM+22]. An implementation of a systolic array can be either input-, weight-, or output-stationary, depending on its programming abstraction. The integration of a matrix-multiply unit and a general-purpose CPU can be either coarse-grained (e.g., as a co-processor like TPU), medium-grained (e.g., sharing some levels of caches with the CPU), or fine-grained (e.g., as a functional unit in the CPU's pipeline).

1.2.2 Specialization for Sparse GEMM Pattern

As emerging workloads (e.g., machine learning and graph analytics) are processing increasingly large and sparse datasets [RCK+20, NMS+19, HMD15, JYP+17, WBC+19b, Dav19, HS11, ST15], specialization for the sparse GEMM pattern is becoming critical to maximizing work and storage efficiency by avoiding multiplying and storing zeros. Previous work has proposed multiple accelerators specialized for the sparse GEMM pattern, mainly based on three different dataflows: inner product, outer product, and row-wise product. OuterSPACE implements the outer product dataflow using tiles of processing elements to perform outer products for pairs of sparse vectors, each including a column of the first matrix and a row of the second matrix, and to merge partial output matrices [PBP+18]. MatRaptor implements the row-wise dataflow by multiplying each

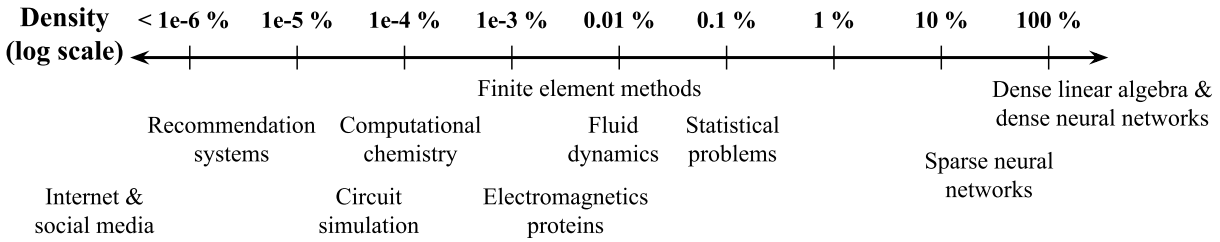


Figure 1.1: Matrix Density Spectrum Across Different Workload Domains – This figure is adopted from [HAMP+19].

non-zero element in the first matrix with a corresponding row in the second matrix [SJL+20]. Each group of processing elements in MatRaptor is mapped to a set of output rows, and they perform multiplications followed by merging partial results. SIGMA is an inner-product-based accelerator for sparse GEMM in deep learning applications [QSK+20]. It implements a flexible array of dot-product engines consisting of multipliers, adders, a distribution network, and a reduction network. There are several other accelerators specialized for the sparse GEMM pattern such as SpArch [ZWHD20], Sextans [SCS+22], Extensor [HAMP+19], and Gamma [ZAES21].

In addition to those decoupled accelerators specialized for the sparse GEMM pattern, previous work has proposed an ISA extension called SparseCore for performing sparse computation [RCYQ22]. SparseCore introduces a set of stream registers to store information about a sparse vector (e.g., stream length and starting addresses of key/value arrays). SparseCore supports the inner-product dataflow via a stream-intersecting instruction, and the outer-product and row-wise product dataflows via a stream-merging instruction. SparseCore also provides a list of stream load and store instructions to move data between stream registers and memory.

Regardless of programming abstractions, hardware specialized for the sparse GEMM pattern typically includes processing units performing an index matching operation for intersecting key/-value streams (i.e., in the inner-product dataflow) and/or a merge operation for merging key/-value streams (i.e., in the outer-product and row-wise product dataflows). In addition, since sparse GEMM computation is typically memory-bound, such accelerators are often implemented with large on-chip scratchpads and/or data buffers to stage intermediate results and minimize memory latency.

1.2.3 The Need for Both Dense and Sparse GEMM Patterns

Hardware specialized for the dense GEMM pattern is efficient for processing highly dense matrices in which most values are non-zeros due to the regularity of its two-dimensional array data structure. This regularity enables efficient blocking/tiling approaches to keep data in fast local storage (e.g., caches and scratchpads) and maximize data reuse. The systolic execution of dense GEMM further increases data reuse via input or output stationary approach.

When processing highly sparse matrices, overheads of storing zeros and performing ineffectual multiplications with zeros outweigh those benefits of dense-GEMM hardware specialization. By storing and computing only non-zero values in highly sparse matrices, the sparse GEMM pattern can offer significantly more efficient execution and storage than using the dense GEMM pattern. However, the sparse GEMM pattern is not well suited for performing dense GEMM due to extra metadata (e.g., row/column indices of non-zeros) for tracking non-zero values and irregular memory accesses to compact sparse matrix storages.

Figure 1.1 shows a variety of both traditional and emerging workload domains in a spectrum of matrix density. The wide range of matrix density levels across those workloads motivates the significance of efficiently supporting specialization for both dense and sparse GEMM patterns in modern systems.

1.3 Approaches to Supporting Multiple Types of Specialization

Figure 1.2 shows a design space of different approaches and their performance trade-offs with respect to two hypothetical program patterns A and B. The most general approach is to use a general-purpose processor (GPP) to execute all kinds of applications with different program patterns including both A and B. Since there is no specialization for either A or B, this approach often produces the lowest performance and efficiency compared to specialized approaches. Singular specialization refers to specializing both abstraction and micro-architecture for a single program pattern. A singularly specialized accelerator typically gives higher performance and efficiency for applications with its target program pattern than a GPP. In the singular specialization, multiple micro-architectures implementing the same specialized abstraction may be possible with different performance-area trade-offs. For example, both integrated short-vector units (e.g., Ocelot [Ten23])

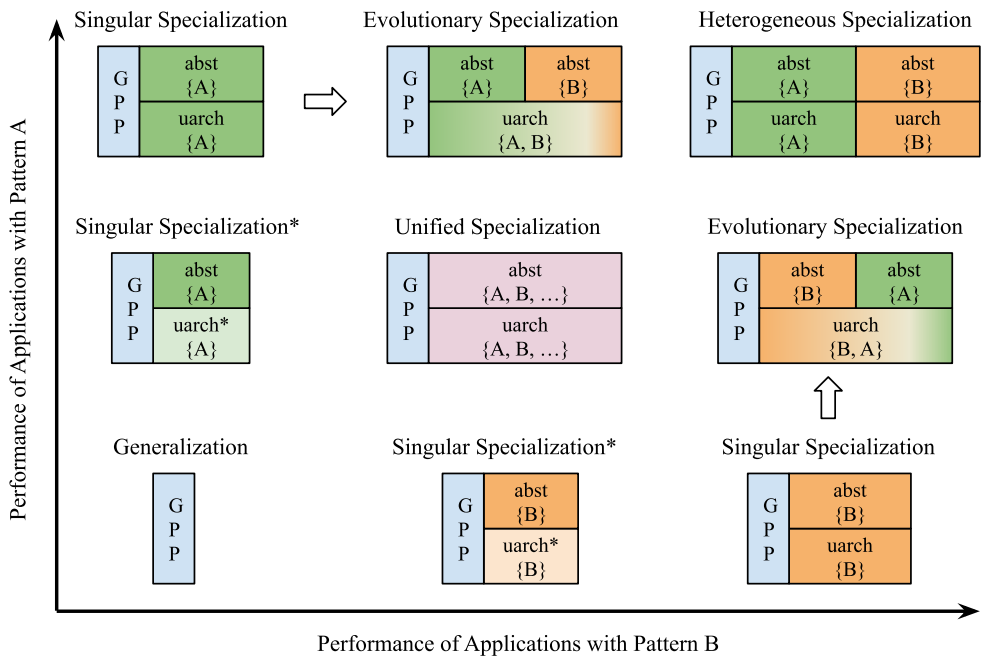


Figure 1.2: Relative Performance of Different Specialization Approaches – GPP = general-purpose processor; abst = program abstraction; uarch = micro-architecture; The size of each box for a design point roughly represents its relative silicon area with respect to other design points. (*) = less optimal but smaller micro-architecture with the singular specialization approach.

and decoupled long-vector engines (e.g., Ara [PCW+22]) can implement the same vector abstraction (e.g., RISC-V vector extension), but they differ significantly in their sizes and performance for workloads with the SIMD pattern.

In general, it is desirable to support multiple kinds of specialization for different program patterns in modern systems due to the co-existence of multiple patterns across workloads and application domains. In this section, I discuss three different approaches to supporting multiple kinds of hardware specialization and their trade-offs: (1) heterogeneous specialization, (2) unified specialization, and (3) evolutionary specialization.

Heterogeneous Specialization – This approach refers to composing multiple singularly specialized accelerators without changing their abstractions and micro-architectures into a single system. The heterogeneous specialization has already been seen in modern processors from mobile systems to servers. For example, modern Qualcomm Snapdragon SoCs include multiple little cores specialized for the SPMD pattern, vector units specialized for the SIMD pattern, tensor accelerators and neural processing units specialized for processing deep neural networks, and image processing units specialized for compute patterns in computer vision [CWS+14, Gwe14a, CHK+21]. Some

examples of the heterogeneous specialization in servers are Intel Knights Landing [SGC+16] and Cray BlackWidow [Abt07]) that consist of scalar and vector cores to support both SPMD and SIMD patterns. Google TPU-v4 includes a SparseCore, which is specialized for sparse embeddings in machine learning inference, next to its systolic array specialized for the dense GEMM pattern [JKL+23b]. By retaining each optimal abstraction and micro-architecture for individual program pattern, this approach can achieve the highest performance for applications with the supported patterns. However, since there is no reuse across specialized hardware components, the heterogeneous specialization is relatively expensive in terms of silicon area. In addition, it is often challenging to determine a "perfect" on-chip area ratio among different hardware components at the design time since the balance of the supported program patterns in workloads may change as software evolves over time.

Unified Specialization – This approach refers to using a single unified abstraction and micro-architecture that are generic enough to support multiple program patterns. For example, vector-thread architectures [KBH+04a] and GPGPUs [nvi09] provide unified abstractions and micro-architectures for supporting both the SPMD and SIMD patterns. Another example are vector architectures and GPGPUs that support generic enough abstractions and micro-architectures to accelerate both the dense GEMM pattern [LLK+19, KTD12] and the sparse GEMM pattern [FC23a, NMAB18, WMZ+19]. Unlike the heterogeneous specialization that retains optimal abstractions and micro-architectures for individual program patterns, the unified specialization trades off the per-pattern optimality for the capability of supporting multiple patterns in the same abstraction and micro-architecture. Having the same abstraction often enables a unified software stack for multiple programs. Sharing the same micro-architecture enables hardware reuse when executing programs with different patterns, which increases hardware utilization. In addition, a unified abstraction and micro-architecture can support more than just a set of program patterns that a heterogeneous system targets. Due to adopting a sub-optimal abstraction for a program pattern, the unified specialization leaves out opportunities for software to convey potential pattern-specific optimizations to a corresponding micro-architecture. For example, in a GPGPU abstraction, threads appear to execute independently with their own control flows, which hinders the abstraction from expressing regular unit-stride memory accesses across multiple threads. In comparison to a vector engine implementing a vector abstraction, this missing information about the inter-thread memory access pattern makes it necessary to include expensive hardware logic to coalesce those unit-stride

memory accesses for performance at the cost of less energy efficiency [nvi09], and further prevents the hardware from issuing those regular memory accesses ahead of time to hide long memory latency [LAB+11].

Evolutionary Specialization – This approach refers to starting from an optimal abstraction and micro-architecture for one program pattern and gradually adding a minimal set of hardware changes to the existing micro-architecture to support additional program patterns without changing their optimal abstractions. Since both abstraction and micro-architecture for the starting program pattern are unchanged, the evolutionary specialization is guaranteed to retain the optimal performance and efficiency of the singular specialization. For the additional program patterns, the evolutionary specialization keeps the same optimal abstractions as supported in their singularly specialized accelerators so that all program properties specific to those patterns can be conveyed to the hardware. In comparison to the heterogeneous specialization that dedicates separate hardware components for accelerating individual program patterns, this evolutionary specialization maximizes hardware reuse across multiple patterns in one single micro-architecture, which enables higher hardware utilization and lower area costs. Since the single micro-architecture is not optimally specialized for the additional program patterns, it may not perform workloads with those patterns as efficiently as a heterogeneous micro-architecture. There are two key design questions in this approach. Firstly, what should be the starting and additional program patterns in this approach? The answer to this question largely depends on specific situations. Generally, one could choose a more commonly used program pattern to start with so that workloads with that pattern can perform the best without any compromise in performance and efficiency. Secondly, what is a set of common micro-architectural features shared between the starting and additional micro-architectures? For example, in both multi-/many-core systems and vector engines, arithmetic execution pipelines are somewhat similar, and a per-lane slice of a vector register file could be viewed as a register file in a scalar core. This approach may require repurposing certain hardware components in the starting micro-architecture and/or rethinking how those components could be used in different ways to implement the additional abstractions. However, the evolutionary specialization may not be appropriate when micro-architectures specialized for the starting and additional program patterns are radically different since the amount of extra hardware needed to support the additional patterns would be significant.

1.4 Thesis Overview

In this thesis, I explore the evolutionary specialization to support multiple types of specialization in a unified hardware implementation by gradually evolving an existing micro-architecture specialized for one programming pattern to support other patterns. This thesis motivates the evolutionary specialization using two novel architectures: big.VLITTLE and SparseZipper. In big.VLITTLE architectures, a multi-little-core system specialized for the SPMD pattern is reconfigured to support the SIMD pattern with minimal area overhead to the original hardware. In SparseZipper architectures, a programmable systolic-array-based micro-architecture designed for accelerating dense GEMM is repurposed so that it can efficiently perform GEMM on sparse matrices as well.

Chapter 2 presents a multi-university chip tapeout project called CIFER that illustrates the heterogeneous specialization approach. The CIFER chip includes heterogeneous hardware components: multiple general-purpose cores capable of running Linux, several tiles of tiny cores designed for exploiting massive task-level parallelism, and an embedded FPGA for application-specific acceleration. This chapter details the design of many-tiny-core tiles and their integration into the CIFER chip, which is the key contribution of a team of post-doc and students led by me at Cornell University. Using CIFER as an example, this chapter also presents opportunities and challenges of the heterogeneous specialization.

Chapter 3 describes my early work in supporting cycle-level modeling of multi-core RISC-V systems in gem5, a popular cycle-level architecture simulator. Compared to the RTL modeling used in the CIFER tapeout, this work provides a fast and flexible methodology for exploring architectural design space and evaluating big.VLITTLE and SparseZipper work in Chapter 4 and 5.

Chapter 4 presents big.VLITTLE architecture that provides on-demand data-parallel acceleration for mobile systems on chip. The big.VLITTLE architecture illustrates the evolutionary specialization approach by gradually evolving (i.e., minimally adding just enough hardware components) to a multi-core micro-architecture designed for the SPMD pattern to support a vector abstraction designed specially for the SIMD pattern.

Chapter 5 presents the SparseZipper architecture that can efficiently accelerate the dense and sparse GEMM patterns. The SparseZipper instruction set provides additional matrix instructions specialized for the sparse GEMM pattern on top of existing matrix instructions specially used for

the dense GEMM pattern. At the micro-architecture level, the SparseZipper minimally extends a systolic array specialized for computing dense GEMM to enable sparse GEMM computation in the same hardware.

Chapter 6 summarizes the contributions of this thesis and discusses several research directions for future work. The primary contributions of this thesis are:

- an exploration of the evolutionary specialization that supports multiple types of hardware specialization by gradually evolving a micro-architecture specialized for one kind of specialization to support other kinds of specialization without impacting their programming abstractions;
- a novel big.VLITTLE architecture, an example of the evolutionary specialization, supporting on-demand data-parallel acceleration via a vector abstraction in a multi-core micro-architecture designed for the SPMD pattern; and
- a novel SparseZipper architecture, another example of the evolutionary specialization, efficiently supporting both dense and sparse GEMM computation by gradually extending an existing matrix ISA and micro-architecture specialized for the dense GEMM pattern to support the sparse GEMM pattern.

1.5 Collaboration and Funding

I collaborated with Moyang Wang who led the heterogeneous cache-coherent systems project. I helped model the baseline cache-coherent systems using the Ruby memory model in gem5. I helped debug cache coherence implementation bugs in gem5. I contributed in the discussion of several intellectual aspects of this project regarding cache coherence protocol and dynamic task scheduling. I contributed to setting up an automated simulation flow to enable more productive and less error-prone research. The work has been published and presented by the lead of the project, Moyang Wang, at the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA) in June 2020 [WTCB20]. This work inspired our software-managed cache coherence implementation in the CIFER chip discussed in Chapter 2.

I led the effort in the Batten Research Group (BRG) to support RISC-V ISA in gem5 with the help of Lin Cheng. The work is discussed in details in Chapter 3. I improved gem5 to sup-

port simulations of a multi-core RISC-V system in its system call emulation mode. I ported an open-source RISC-V test suite from the RISC-V community into gem5 to enable instruction-level assembly testing that was missing in the gem5 simulator. I fixed numerous bugs related to simulating multiple RISC-V threads in the in-order and out-of-order core models in gem5. I contributed all of my development including changes in gem5 and a new set of assembly tests to the open-source gem5 codebase. This work has been published and presented by me at the 2nd Workshop on Computer Architecture Research with RISC-V (CARRV) held in conjunction with ISCA-45 in 2018 [TCB18].

I was part of the multi-university CIFER chip tapeout project presented in Chapter 2. CIFER is a heterogeneous system integrated with open-source Ariane cores, an embedded FPGA contributed by the research group led by Professor David Wentzlaff at Princeton University, and many-tiny-core tiles contributed by BRG group led by Professor Christopher Batten at Cornell University. The Princeton team was the overall lead of this project, and I was the lead of the BRG group in this project. More specially, I was in charge of developing, implementing, and verifying our tiny in-order core RTL model. I helped Moyang Wang, Xiaoyu Yan, and Eric Tang with integrating tiny cores with their software-managed coherent caches. Together with Moyang, we constructed a tile of tiny cores and integrated three tiny-core tiles into the final CIFER chip. With tremendous help from Shady Agwa and Yanghui Ou, I pushed the tiny-core tiles through a standard-cell-based ASIC toolflow for gate-level testing, preliminary timing closure, and early area analysis. I helped Moyang develop a runtime system that enables running task-parallel programs on CIFER. The project would not have been possible without the lead of the Princeton team including Ting-Jung Chang, Ang Li, Fei Gao, Georgios Tziantzioulis, Jinzheng Tu, Kaifeng Xu, Paul Jackson, August Ning, Grigory Chirkov, Marcelo Orenes-Vera, and Jonathan Balkind in top-level integration, final-step design rule checking, and post-silicon testing. With the help of the Princeton team, I was able to verify and evaluate the many-tiny-core tiles after we received CIFER chips back from a fabrication plant. The work has been published and presented by the project's general lead, Ang Li, at the IEEE Custom and Integrated Circuits Conference (CICC) in April, 2023 [CLG+23].

I led the big.VLITTLE project presented in Chapter 4. Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, and Courtney Golden contributed significantly to the project. Khalid developed the decoupled vector engine and the integrated vector unit models used as the baseline in this work. He assisted in developing and debugging various other gem5 components. He also contributed to the

intellectual development of this project. Nick led the development of benchmarks from the RIVEC benchmark suite with the help of Eric and Courtney. Yanghui contributed to the first-order VLSI area model used to evaluate the area overhead of big.VLITTLE. The work has been published and presented by me at the 55th ACM/IEEE International Symposium on Microarchitecture (MICRO) in October 2022 [TAHC+22].

I led the SparseZipper project presented in Chapter 5. Joshua Randall, Krishnendra Nathella, and Jesse Beu contributed to the initial idea that led to SparseZipper while I was doing an internship at Arm Research. Josh continued the collaboration with me after I finished the internship, and he provided important feedback regarding technical aspects in SparseZipper. Yanghui Ou contributed a first-order VLSI component-based area model used to evaluate the area overhead of SparseZipper.

This work was supported in part by NSF PPoSS Award #2118709, NSF SHF Award #2008471, DARPA POSH Award #FA8650-18-2-7852, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

CHAPTER 2
CIFER: A CHIP PROTOTYPE USING HETEROGENEOUS SPECIALIZATION

CIFER[1] is the first academic open-source multicore-eFPGA system on chip (SoC) composed of heterogeneous architectures capable of exploiting both parallelism and specialization by integrating parallel manycore, eFPGA, and Linux-capable multicore into a single SoC. By leveraging five open-source hardware projects [ZB19, BMF+16, TOJ+19, LW21, BLS+20], a team of postdocs, graduate and undergraduate students across Princeton University and Cornell University designed and finished the tapeout within seven months during the pandemic. CIFER has been taped out using GlobalFoundries 12nm FinFET technology node on a $16mm^2$ (4mm×4mm) die and packaged in a 208-pin ceramic quad flat pack. Figure 2.1 shows the CIFER die photo with annotations showing heterogeneous components in the chip.

This chapter first presents the overall architecture of CIFER including the Linux-capable multicore tiles, the TinyCore tiles, the eFPGA, and the heterogeneous cache coherence. Section 2.2 then discusses verification methodology that enabled efficient testing of individual components and the entire CIFER chip. Section 2.3 evaluates performance and energy efficiency of CIFER compared to other state-of-the-art chips targeting edge/IoT workloads. Finally, we conclude with discussions on opportunities and challenges of heterogeneous specialization approach from this CIFER tapeout experience.

2.1 CIFER Architecture

Figure 2.2 details the CIFER micro-architecture including Linux-capable multicore tiles, TinyCore tiles, an eFPGA tile with its controller, a 2×4 on-chip mesh network, private level-one instruction and data caches, and a shared level-two cache distributed across multiple tiles. In this section, we discuss architecture-level details of the multicore tile, TinyCore tile, eFPGA, and heterogeneous cache coherence implemented in CIFER.

[1] CIFER stands for Coherent Interconnect and FPGA Enabling Reuse

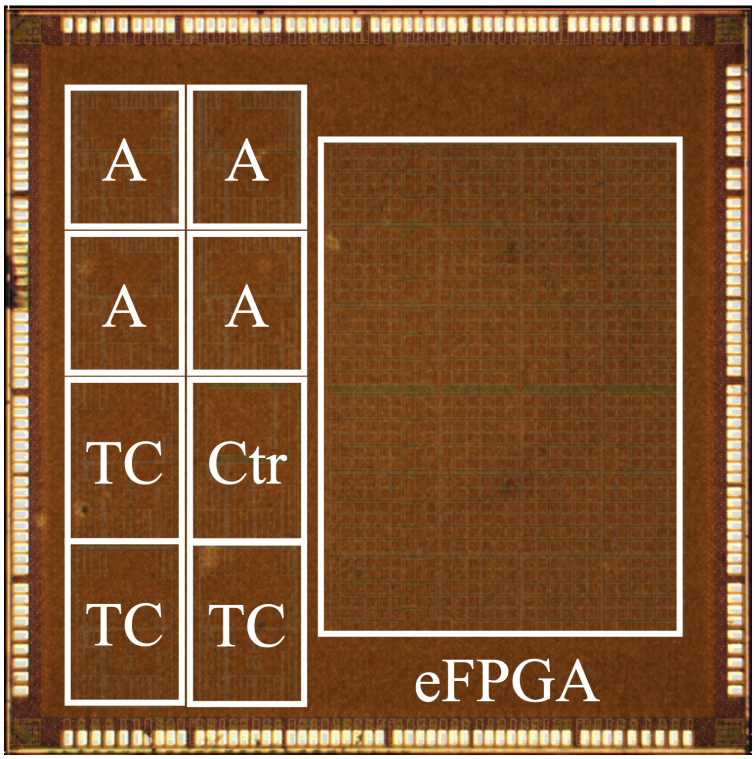


Figure 2.1: CIFER System-on-Chip Die Photo – A = Ariane cores; TC = Tiny-core clusters; Ctr = FPGA controller tile; eFPGA = Embedded FPGA.

2.1.1 Linux-Capable Multicore Tile

There are four Linux-capable multicore tiles in the CIFER chip. Each tile includes one Ariane core that is an open-source Linux-capable processor supporting RV64GC instruction set [ZB19]. The pipeline of Ariane supports in-order issue, out-of-order write-back, and late commit for precise exceptions and interrupts that are essential for running Linux. Ariane also supports double-precision floating-point unit (FPU). Each core has a 16KB L1 instruction cache and an 8KB L1 data cache. Coherence between an Ariane core's private caches and the shared L2 cache is implemented in hardware using the BYOC interface [BLS+20]. These Ariane cores are used for general-purpose compute such as running an operating system, managing threads running on the TinyCore clusters, and offloading compute tasks to the eFPGA. In addition, those Ariane cores can be used to run applications with limited thread-level and/or irregular parallelisms that would not be efficient to run on either the TinyCore clusters (i.e., due to low single-thread performance of a tiny core) or eFPGA (i.e., due to reconfiguration overheads).

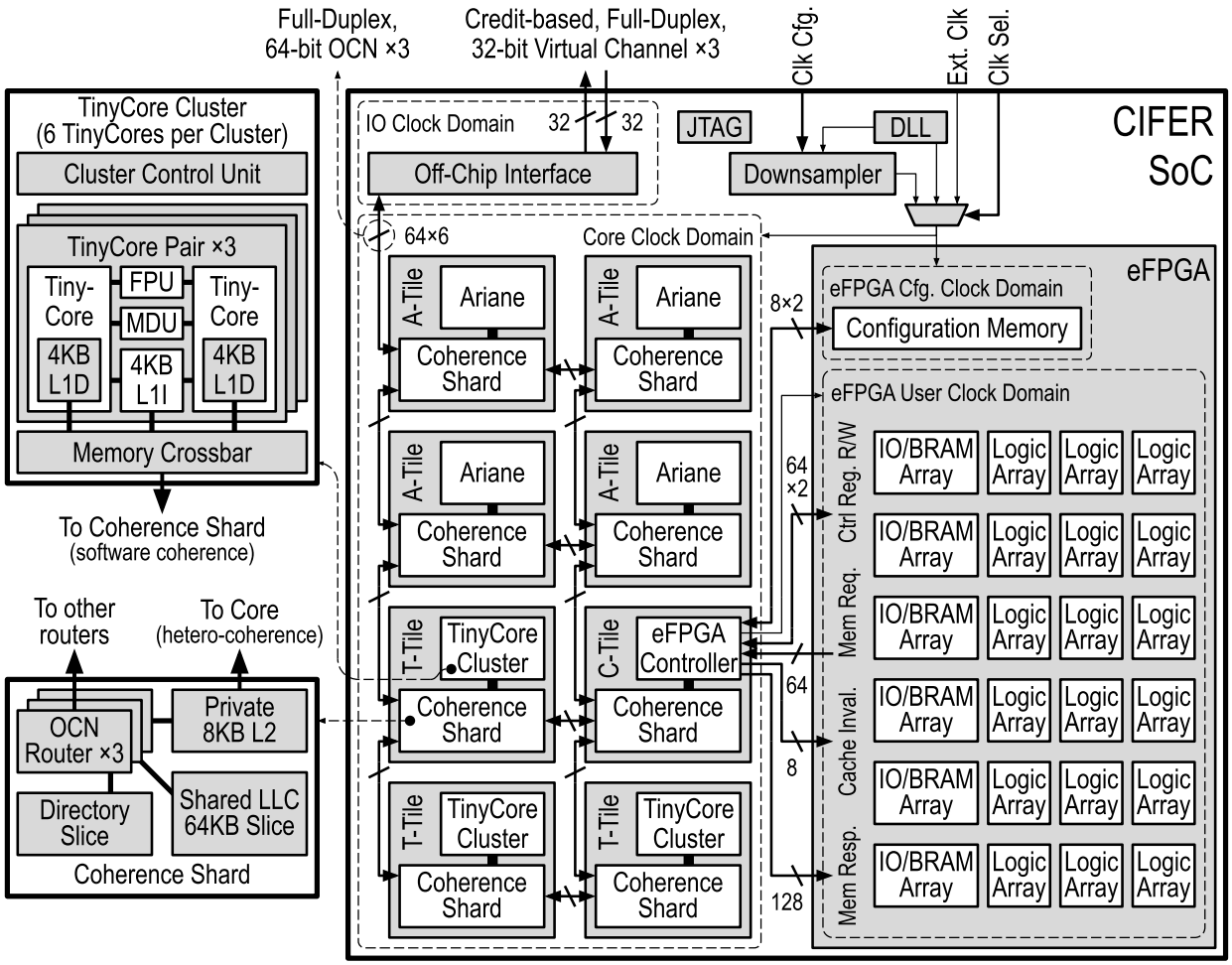


Figure 2.2: CIFER System-on-Chip Architecture – A-Tile = Ariane core tiles; T-Tile = Tiny-core tiles; C-Tile = FPGA controller tile; eFPGA = Embedded FPGA.

2.1.2 TinyCore Tile

To exploit massive thread-level parallelism, CIFER includes three TinyCore tiles, each consisting of six light-weight cores, for a total of 18 tiny cores on chip. Each tiny core is a 32-bit RV32IMAF processor with a six-stage, in-order issue, out-of-order write-back, late commit and scalar pipeline. To address write-after-write and write-after-read hazards during out-of-order execution, each tiny core supports limited register renaming capability with 40 integer and floating-point physical registers. Each core has a private, 4KB L1 data cache. Two adjacent cores form a pair share a 4KB L1 instruction cache, an integer multiply-divide unit (MDU), and a single-precision floating-point unit (FPU). A small L0 instruction buffer is added to the front-end of each

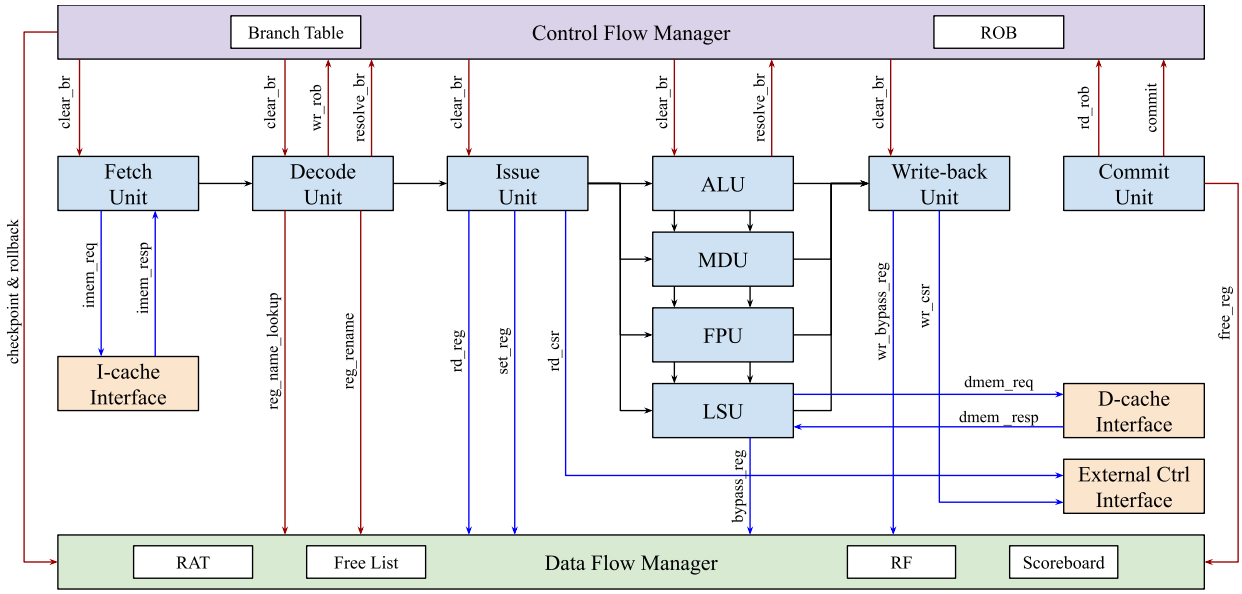


Figure 2.3: Modular Tiny-Core Micro-architecture – ROB = reorder buffer; RAT = register alias table; RF = register file; ALU = integer arithmetic unit; MDU = integer multiply/divide unit; FPU = single-precision floating-point unit; LSU = load/store unit; br = branch; reg = register; rd = read; wr = write.

tiny core to minimize the latency impact of sharing the L1 instruction cache. The sharing of area-expensive execution units maximizes computation density in a tiny core cluster.

The implementation of tiny core is modular so that individual components in its pipeline can be tested as standalone units. Figure 2.3 shows the modular micro-architecture of the tiny core. There are six main pipeline stages: fetch, decode, issue, execute, write-back, and commit implemented as individual units with well-defined interfaces with other stages, global control and data managers, memory interfaces, and external control interface. The fetch unit handles fetching instructions from the L1 instruction cache, predicting branch outcomes (i.e., using a simple always-not-taken branch predictor), and redirecting the instruction stream in case of mispredictions. The decode unit is in charge of instruction decoding and register renaming. The issue unit schedules instructions to the back-end execution pipelines and performs register file read for source operands of issued instructions. The execute stage includes multiple execution units for integer/floating-point arithmetic and load/store operations. The write-back unit handles writing back data for output operands into the register file. Finally, the commit unit monitors the head of a re-order buffer (ROB) and tries to commit at most one instruction per cycle. Communications (e.g., instruction forwarding and pipeline back-pressure) between two adjacent units are performed via a valid/ready interface.

The control flow manager includes a branch table for tracking speculative branch instructions that are waiting for their branch resolutions Once a prediction is made for a branch instruction in the fetch unit, the branch with its prediction is added to the branch table. In addition, the control flow manager directs the data flow manager to take a snapshot of the register alias table (RAT) so that it can be rolled back in case the prediction is wrong. Until the branch's outcome is resolved later in the pipeline (i.e., in the decode stage for unconditional jump instruction and in the execute stage for conditional branch instructions), instructions following the predicted branch are tagged as speculative under the branch. In case of a branch misprediction, all speculative instructions under the corresponding branch need to be squashed. Once a branch's outcome is resolved (e.g., in the execute stage), the control flow manager broadcasts a clear branch (i.e., `clear_br`) signal with the branch tag to related front-end and back-end stages for either killing instructions under the branch in case of a misprediction or clearing the speculative flag in those instructions in case of a correct prediction. If a branch is mispredicted, the control flow manager needs to direct the data flow manager to roll back the RAT snapshot associated with the mispredicted branch instruction. In addition to the branch table, the control flow manager includes a re-order buffer (ROB) for tracking pending instructions waiting to commit. Instructions are inserted into the ROB when they are dispatched in the decode stage. The commit stage monitors the ROB's head for committing completed instructions and potentially handling exceptions.

The data flow manager handles the data flowing across multiple stages in the pipeline. It includes a register alias table (RAT) for tracking the mapping between logical and physical registers, a free list for maintaining a list of physical registers available for new mapping, a register file, and a scoreboard tracking whether values for registers are ready. Register names are looked up and updated in the decode stage. If there is no available physical register, the pipeline simply stalls. Once an instruction is committed, the previous register name of its destination register is returned to the free list. Once an instruction is issued, the issue unit directs the data flow manager to read the register file for source operand values and mark its destination register as pending (i.e., its value is not ready yet). Value bypassing happens at the end of each execution pipeline and the write-back stage via the data flow manager. The write-back stage sends a write-back command to the data flow manager for writing back register values once instructions complete their execution.

The tiny core pipeline also includes interfaces to external memory (caches) and control. The i-cache and d-cache interfaces are connected to the fetch unit and the load/store unit respectively. For

external control communications (e.g., debug and reset signals) with the rest of the chip, we implemented an external control interface in each tiny core. Two special instructions (i.e., `proc2mngr` and `mngr2proc`) are implemented so that the core can send and receive control signals through the external control interface.

Coherence between private L1 caches and the shared L2 cache is managed explicitly in software by inserting special cache flush and invalidation instructions. In particular, a cache flush performs a full cache walk to write back each dirty cache line while a cache invalidation clears the valid bits of clean cache lines. A small per-cluster private cache in the BYOC interface forwards write-back and atomic requests from private L1D caches to the shared L2 cache. Snoop requests from the shared L2 cache are not propagated to L1D caches due to explicit software-managed cache invalidation and flush.

In each tiny core tile, we implemented a per-tile control logic that coordinates the execution of tiny cores within a tile via the external control interface in each core. The control logic is responsible for waking up all cores upon receiving a wakeup signal from CIFER's top-level control and monitoring whether those cores complete their executions. An Ariane core can wake up a tiny core tile through a memory-mapped register interface. Once all cores in a tiny core tile complete their tasks, the per-tile control logic updates a memory-mapped register to signal the tile is available.

2.1.3 Embedded FPGA

The eFPGA designed with PRGA [LW21] has 6720 multi-node, 6-input LUTs and 18 24Kbit, dual-port block RAMs. Hardware designers can use an open-source RTL-to-bitstream toolchain consisting of Yosys [Wol20], VPR [MPZ+20], and PRGA's bitstream assembler to build accelerator designs that are then mapped to the eFPGA. The eFPGA is the first silicon instantiation of Duet [LNW23] that includes two interfaces: (1) the control register interface for a host processor (e.g., Ariane core) to control the eFPGA via a memory-maped I/O interface; and (2) the coherent memory interface supporting non-coherent, IO-coherent, or bidirectionally coherent memory accesses of the eFPGA. Communications between the eFPGA and the rest of the system are performed through the eFPGA controller. To enable fine-grain cooperative execution between the eFPGA and other compute tiles in CIFER, atomic requests to shared memory are supported in the eFGPA.

2.1.4 Heterogeneous Cache Coherence

CIFER unifies heterogeneous cache coherence protocols of different processing tiles within a fully coherent cache system. This unification minimizes the communication overhead across those tiles, which enables fine-grained cooperative execution of heterogeneous processing elements on chip, besides the conventional offloading programming model. Processing tiles with different cache coherence protocols communicate over the BYOC framework in which a small per-cluster cache (i.e., L1.5 cache) handles coherence transactions between the shared L2 cache and private L1 cache(s) in each cluster.

To address programming challenges in using a software-managed coherence protocol implemented in the tiny core tiles, we implemented a task-parallel work-stealing runtime that facilitates parallel thread execution across Ariane and tiny cores by leveraging coherent caches. The runtime is in charge of inserting automatically cache flush and invalidation instructions when necessary to enable the coherence of shared data between tiny cores and the rest of the system. An eFPGA-emulated accelerator can be efficiently invoked by simply passing the memory addresses of the data to be processed. Depending on the computation, the accelerator can either copy a continuous chunk of data into its BRAM scratchpad or read/write memory in a random byte-granular manner.

2.2 Verification Methodology

Integrating heterogeneous hardware IPs into a single SoC poses significant challenges in testing not just individual IPs but also the whole systems with integrated components. For pre-silicon verification, we used a combination of various testing strategies to rigorously stress different parts of CIFER and the entire chip.

Unit & Integration Testing – Since we leveraged open-source hardware IPs (e.g., Ariane core, OpenPiton for top-level cache system, PRGA for the eFPGA, and PyOCN for on-chip network), we rely on previous verifications of those components, which reduced significantly our verification effort. For other components such as the TinyCore tiles, we leveraged the PyMTL3 testing framework to perform unit testing. For example, we developed directed test cases for individual pipeline units and flow managers of the tiny core's pipeline (i.e., discussed in Section 2.1.2) before the integration of those components into a tiny core's pipeline is tested. Such low-level fine-grained

unit testing helped us build confidence in the correctness individual components and manage the complexity of verifying this entire chip. Since unit testing is focused on individual components, integration testing is focused more on verifying connections between components. One of the most challenging verification tasks was to verify the integration of a TinyCore tile with the top-level shared L2 cache. We developed specific test cases for invoking coherence transactions going through the L1.5 cache of the BYOC interface.

Directed & Random Testing – We developed directed test cases to verify specific hardware behaviors and transactions (e.g., cache replacements and squashing behaviors triggered by a branch misprediction in the tiny core's pipeline). In addition to directed test cases, we relied on random testing to cover cases that are more complex. For example, we used PyH2 testing framework [JOP+20] in PyMTL3 to generate random valid RISC-V programs with arbitrarily long sequence of instructions. Those random test cases helped discover bugs only triggered by certain unique sequence of transactions (e.g., nested branches with certain branch outcome patterns).

RTL & Gate-Level Testing – We used the PyMTL3 testing framework for efficiently writing test cases in Python. The framework automatically generated Verilog testbenches from Python test cases for RTL testing. We reused the same testbenches for post-synthesis gate-level testing.

2.3 Evaluation

Figure 2.4 shows the maximum operating frequency of each hardware component across the range of functional supply voltage. The eFPGA's maximum operating frequency depends on the emulated design. Figure 2.4 shows the maximum operating frequency of a 64-bit LFSR. CIFER's maximum frequency is 1195MHz when running at 1.1V. Table 2.1 compares CIFER with other state-of-the-art CPU-FPGA SoCs targeting the edge/IoT computing.

The aggregate peak performance and energy efficiency of CPUs in CIFER are 15.54 GFLOPS at 1.1V and 53.18 GFLOPS/W at 0.7V. For the estimated power dissipation, we excluded the eFPGA's configuration clock power based on our post-layout power analysis. CIFER outperforms the next best SoC by 6.5× and 1.4×. The eFPGA's area efficiency is 1541 LUT6/mm^2 which is 11.2× better than other synthesizable eFPGAs and only 1.3× worse than the best full-custom eFPGA. Regarding performance, the eFPGA achieved the peak of 1.92 MOPS/LUT, 126MHz at 1.1V, 148.1 GOPS/W at 0.7V, and 97% utilization when performing a 64-point FFT. The eF-

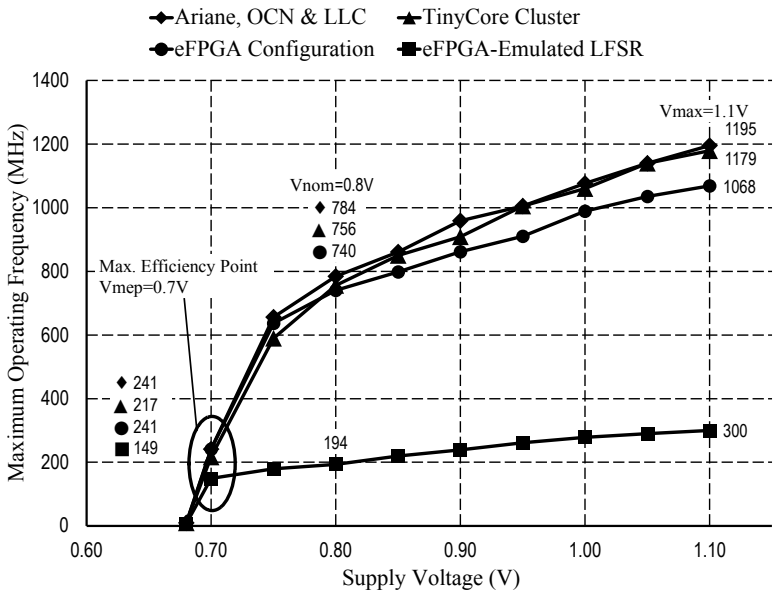


Figure 2.4: Maximum Operating Frequency vs. Supply Voltage.

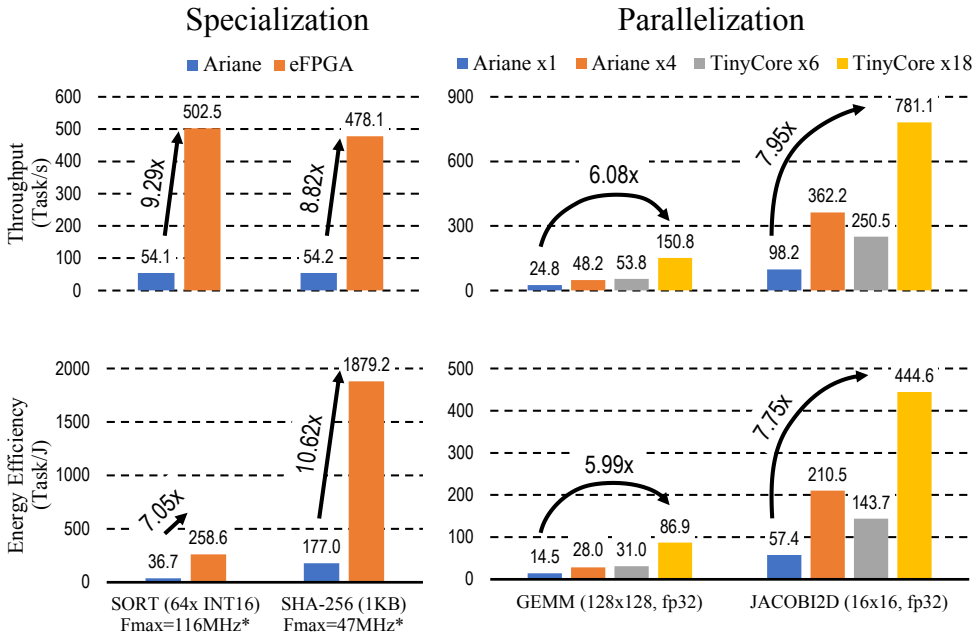


Figure 2.5: Performance and Energy Efficiency Gains – The CPUs, on-chip network, cache system, and the eFPGA controller run at full speed (i.e., 740MHz at 0.8V). Fmax indicates the maximum operating frequency of the eFPGA-emulated design.

				This Work	TCAS'20	ISSCC'19	TVLSI'21	JSSC'22
Chip		Technology		12nm FinFET	90nm BCD	40nm CMOS + 39nm MRAM	22nm FD-SOI	16nm FinFET
		Die Area (mm^2)		16	1.78	22.09	9	25
		V_{nom} (V_{min} - V_{max})		0.8 (0.68 - 1.1)	1.2 (-)	- (1.1 - 1.3)	0.8 (0.5 - 0.8)	0.8 (0.5 - 1.05)
		Active Power (mW)	V_{nom}	1792	1.2	5.34	24.95	918
		F_{max} (MHz)	V_{max}	**1195**	10	200	600	972
CPU	Host	Core Type		4× Ariane	RI5CY	Cortex-M0	RI5CY	2× Cortex-A53
		ISA		RV64GC	RV32I	ARMv6-M	RV32IMFC	ARMv8-A
		CoreMark Score	V_{max}	**7918**	31.9	466	1914	6376
	Other	Core Type		18× TinyCore	N/A	N/A	N/A	Cortex-M0
		ISA		RV32IMAF				ARMv6-M
		Function		Parallel Compute				Monitor
		CoreMark Score	V_{max}	**19198**				2265
	Total	Peak GFLOPS	V_{max}	**15.54**	NO HW FPU	NO HW FPU	NO HW FPU	1.94
		Peak GFLOPS/W	V_{mep}	6.63 **[53.18†]**				38.03

Table 2.1: Comparison to Prior Work – TCAS'20 [RMR+20]; ISSCC'19 [NST+19]; TVLSI'21 [SRDM+21]; JSSC'22 [LSPS10].

PGA achieved lower performance and energy efficiency than the best full-custom eFGPA for three reasons. First, CIFER is synthesized with a standard cell library. Second, there is no hardware multiply-accumulate unit in the eFGPA. Third, we used an open-source RTL-to-bitstream toolchain which is likely to perform worse than a proprietary toolchain.

We measured performance and energy efficiency improvement of running real benchmarks on the TinyCore tiles and eFPGA by offloading SORT and SHA-256 to the eFPGA, and executing GEMM and JACOBI2D on the TinyCore tiles. Those kernels represent the edge/IoT application domain which is the target domain of CIFER. The reported execution time includes all control overhead (e.g., data transfer between tiles) and coherent memory access latency. For a fair energy comparison of different components, we exclude full-chip idle power (i.e., static and clock power). At nominal voltage, the eFPGA is 9.29× better in throughput and 10.62× more energy efficient than the Ariane-only baseline while the TinyCore tiles achieved up to 7.95× speedup and 7.75× improvement in energy efficiency.

2.4 Conclusion

In this chapter, we present CIFER, the first academic open-source multicore-eFPGA SoC composed of multiple Linux-capable cores for running general-purpose workloads, TinyCore tiles for exploiting massive thread-level parallelism, and eFPGA for application-specific acceleration. The chip was fabricated on a GlobalFoundries 12nm FinFET technology node. CIFER features a heterogeneous cache coherence implementation that enables seamless on-chip communications across different compute tiles. For workloads with massive thread-level parallelism, our evaluation results show that the TinyCore clusters improve their performance and energy efficiency by up to 7.95× and 7.75× respectively compared to a single general-purpose Ariane core. For workloads that are well-suited for mapping to the eFPGA, we show up to 9.29× and 10.62× performance and energy efficiency improvement respectively.

CIFER is an example of the heterogeneous specialization by composing heterogeneous hardware components specialized for different compute patterns into an SoC. CIFER demonstrates the key opportunities and challenges of this approach.

Opportunities for exploiting different kinds of specialization in an SoC – CIFER illustrates the potential performance and energy efficiency benefits of integrating hardware components spe-

cialized for different compute patterns into an SoC. Workloads with different compute patterns can be scheduled to run on the most well-suited compute platforms to maximize their performance and energy efficiency.

Opportunities for leveraging open-source projects to build a complex SoC – We leveraged various open-source projects including OpenPiton [BMF+16], PyMTL3 [JIB18], PRGA [LW21], PyOCN [TOJ+19], Ariane core [ZB19], and BYOC framework [BLS+20] to facilitate an efficient and agile hardware development that enabled a team of postdocs and students across two universities to finish the CIFER tapeout with 450+ million transistors in seven months during the pandemic.

Opportunities for fine-grained cooperative execution paradigm – CIFER unifies heterogeneous cache coherence protocols of different processing units within a global, fully-coherent system. Besides the conventional offloading model, this unification enables opportunities for fine-grained cooperative execution across different compute platforms via a shared on-chip memory with heterogeneous cache coherence.

Challenges in maximizing utilization of on-chip resources – The amount of hardware resources dedicated to each kind of specialization is determined at the design time. Therefore, when the balance of compute patterns in workloads is different from the design-time hardware resource allocation, heterogeneous systems such as CIFER are likely to suffer poor hardware resource utilization. More specifically in CIFER, at design time, it was virtually impossible to determine a "perfect" area ratio of Ariane, TinyCore, and eFGPA tiles that could maximize resource utilization when CIFER runs various workloads after the chip comes back.

Challenges in integrating and verifying heterogeneous components – While using open-source hardware IPs lowers the design complexity of individual heterogeneous hardware components, it is challenging to integrate them into a single chip and verify the integration. In CIFER, we did rigorous testing to verify the functionality of whole system after integrating different hardware components and found numerous bugs related to connections between TinyCore tiles and the shared last-level cache at the top level. Besides functional verifications, timing closure is another challenge in CIFER. The whole system was not able to meet the target frequency that each individual tile could achieve in their own timing closure.

CHAPTER 3
CYCLE-LEVEL MODELING OF MULTI-CORE RISC-V SYSTEMS

The RISC-V ecosystem is becoming popular in both industry and academia. The ecosystem provides rich open-source software and hardware tool chains that enable computer architects to quickly leverage RISC-V in their research. While the RISC-V ecosystem includes functional-level, register-transfer-level, and FPGA simulation platforms, there is currently a lack of cycle-level simulation platforms for early design-space exploration. The gem5 simulator is a popular cycle-level simulation platform that provides reasonably flexible, fast, and accurate simulations. Previous work has added single-core RISC-V support to gem5. This chapter presents our early work on simulating multi-core RISC-V systems in gem5. We first describe our approach to functional and timing validation of RISC-V systems in gem5. We then evaluate the performance of the gem5/RISC-V simulator and discuss a design-space-exploration case study using gem5, the open-source RISC-V software tool chain, and two popular task-based parallel programming frameworks. Compared to the register-transfer-level modeling used in Chapter 2, this cycle-level modeling methodology using gem5 enables relatively fast and efficient architecture explorations and evaluations of both big.VLITTLE and SparseZipper architectures presented in Chapter 4 and 5.

3.1 Introduction

RISC-V is an emerging open-source software and hardware ecosystem that has gained in popularity in both industry and academia [ris18, AP14]. At the heart of the ecosystem, the RISC-V ISA is designed to be open, simple, extensible, and free to use. The RISC-V software tool chain includes open-source compilers (e.g., GNU/GCC and LLVM), a full Linux port, a GNU/GDB debugger, verification tools, and simulators. On the hardware side, several RISC-V prototypes (e.g., Celerity [DXT+18]) have been published. The rapid growth of the RISC-V ecosystem enables computer architects to quickly leverage RISC-V in their research.

Hardware modeling and simulation are critical for system design-space explorations. An ideal model is fast to simulate, accurate, and easy to modify. However, achieving all three goals in a single model is difficult (see Table 3.1). The RISC-V ecosystem provides functional-level models (e.g., Spike, QEMU), register-transfer-level (RTL) models (e.g., Rocket, Boom, Ariane), and

	Time to Modify	Time to Simulate	Accuracy
FL	++	++	−−
CL	+	+	−
RTL	−	−−	++
FPGA	−−	++	++

Table 3.1: Different Modeling Levels and Their Trade-Offs – FL = functional-level; CL = cycle-level; RTL = register-transfer-level. Plus and minus symbols show relative comparisons between levels.

FPGA models (e.g., Rocket Zedboard). Functional-level modeling is fast and easy to modify, but it does not capture the timing of the target system. RTL modeling provides cycle-accurate details of the target system at the cost of being slow to simulate and hard to modify. FPGA modeling provides both accurate and fast simulations but is even more challenging to modify owing to lengthy synthesis and place-and-route times. Cycle-level modeling offers a middle ground that is easier to modify than FPGA modeling, faster to simulate than RTL modeling, and more accurate than functional-level modeling. Its flexibility and performance provide a good platform for early system design-space exploration.

gem5 is a popular cycle-level simulator that supports various instruction sets including x86, MIPS, and ARM. The simulator already provides a number of processor, cache, interconnection network, and DRAM models. It also offers advanced simulation features such as fast-forwarding and check-pointing. Previous work has added *single-core* RISC-V support to gem5 [RS17], and our work has focused on adding *multi-core* RISC-V support to gem5.

In section 3.2, we describe our modifications to gem5 to support simulating multi-core RISC-V systems. Sections 3.3 and 3.4 present our functional and timing validation of the implementation. In Section 3.5, we describe the applications used to evaluate our work. Section 3.6 shows the performance of gem5. Section 3.7 presents a small design-space exploration study on a heterogeneous multi-core system with two different task-parallel programming frameworks using the RISC-V implementation in gem5.

3.2 Adding Multi-Core RISC-V Support to gem5

In this section, we describe our modifications to gem5 to support the thread-related system calls (e.g., `clone`, `futex`, and `exit`) and RISC-V synchronization instructions (e.g., atomic memory

operation, load-reserved, and store-conditional instructions) that are required to run multi-threaded applications in the simulator.

3.2.1 Adding Threading System Call Support

gem5 supports two modes of simulation: full-system (FS) and system-call-emulation (SE) [BBB+11]. In FS mode, applications execute using a simulated operating system (OS) exactly as they would on a real system. All system calls are trapped and handled by the simulated OS. In SE mode, system calls are directly emulated within the simulator itself. When an application executes a `write` system call, gem5 simply invokes a corresponding `write` system call using the host machine running the simulator, and no OS code is simulated. In this work, we focus only on SE mode. The implementation of system calls in SE mode is mostly ISA-independent, so much of this code can be directly reused to support RISC-V.

Each CPU in gem5 has a number of hardware (HW) threads. When an application executes, each software (SW) thread is mapped to a particular HW thread. A HW thread maintains its corresponding SW thread's state including its program counter (PC), its register values, and whether the SW thread is active. Thread creation, synchronization, and termination are handled through three system calls: `clone`, `futex`, and `exit`. So to run multi-threaded applications in SE mode, we must focus on supporting these three key system calls. Other thread-related system calls such as `gettid`, `getgid`, and `getpid` are completely ISA-independent and so are implemented in gem5 for RISC-V by default.

Clone System Call – In Linux, an application spawns a new thread by calling the `clone` system call. The new thread can share resources with its parent thread, including the virtual memory space, file descriptors, and other process attributes. The sharing is specified through different flags (e.g., *CLONE_VM*, *CLONE_FILES*, and *CLONE_THREAD*) given to the system call. If the *CLONE_CHILD_ CLEARTID* flag is set, then when a child SW thread terminates it should wake up its parent SW thread.

When executing the `clone` system call in gem5's SE mode, the simulator first finds an available HW thread. Pointers to shared resources (e.g., page table and file descriptor table) are copied from the calling SW thread to the new one. If non-shared resources (e.g., stack space and thread local storage) are pre-allocated, pointers to these resources will be passed into the `clone` system call and then used to initialize the SW thread. Otherwise, gem5 will allocate such resources on its own.

After all necessary attributes and resources are initialized, gem5 activates the HW thread context, and the new SW thread starts executing its first instruction. Most of the existing implementation of the `clone` system call was leveraged to support RISC-V. We implemented some RISC-V specific requirements including a different system call API and register file initialization process.

Futex System Call – Linux supports OS-level thread synchronization through the `futex` system call. The system call supports two operations: *FUTEX_WAIT* and *FUTEX_WAKEUP*. When a SW thread executes the *FUTEX_WAIT* operation, the SW thread checks if the value at a given address still matches a given expected value. If so, the SW thread waits by sleeping. A different SW thread can execute the *FUTEX_WAKEUP* operation to wake up one or more SW threads waiting on a given address. The *FUTEX_WAIT_BITSET* and *FUTEX_WAKE_BITSET* flags enable a SW thread to use a bit map to control which waiting thread(s) to wake up when performing the *FUTEX_WAKEUP* operation. The bit-set flags are commonly used in some parallel programming frameworks (e.g., OpenMP and Cilk).

In gem5's SE mode, each futex address is associated with a list of waiting HW threads. To execute the *FUTEX_WAIT* operation, gem5 puts the calling HW thread into a thread list associated with a given futex address and then suspends the HW thread. The suspended HW thread becomes idle. When a SW thread executes the *FUTEX_WAKEUP* operation on the same address, some HW threads waiting in the thread list are woken up and re-activated. The implementation of the `futex` system call is ISA-independent, so we only needed to modify it to support the bit-set flags to selectively wake up threads. We also needed to fix a more fundamental issue in the thread suspension and activation logic used by all gem5 CPU models. More details on our modifications are described in Section 3.3.

Exit System Call – A SW thread calls the `exit` system call to terminate its execution. If the *CLONE_CHILD_CLEARTID* flag was used to `clone` the child SW thread, then the parent SW thread needs to be woken up.

In gem5's SE mode, when a SW thread executes the `exit` system call, gem5 cleans up all micro-architectural and architectural state belonging to the thread in the CPU pipeline. It then detaches the SW thread from its current HW thread, and the HW thread becomes available for future use. If waking up its parent thread is required, gem5 performs the *FUTEX_WAKEUP* operation on an address given to the `clone` system call that was used to create this SW thread.

3.2.2 Adding Synchronization Instruction Support

The RISC-V "A" standard extension for atomic instructions supports two types of synchronization instructions: atomic memory operations (`AMO`) and load-reserved/store-conditionals (`LR/SC`) [ris18]. RISC-V supports the release consistency model and a memory fence instruction (`FENCE`). These instructions and the memory model are used to synchronize threads through shared variables. Although they have been recently implemented in gem5, their functionality was only validated for single-core simulations [RS17]. In multi-core simulations, we found that some executions using synchronization instructions implemented in the previous work could lead to race conditions and/or thread starvation. We describe our modifications to the implementation to fix these issues.

AMO Instructions – `AMO` instructions (e.g., `amoadd`) perform read-modify-write operations atomically to a given address. They appear to be executed in a single step with respect to all threads.

There are two ways to implement `AMO` instructions: (1) locking a target cache line before performing the operation using the CPU pipeline; and (2) embedding AMO arithmetic logic units (ALU) inside private L1 caches [roc18]. We chose the second approach to implement `AMO` instructions in gem5. We modified its cache model to support executing ALU operations directly in caches. We added a new memory request type called *atomic* in addition to *load* and *store*. *Atomic* requests are treated as if they were normal *store* requests except that no data-forwarding between an *atomic* request and a subsequent *load* request to the same address is allowed. This is due to the fact that *atomic* requests carry no valid data until they are executed in caches. Similar to *store* memory requests, an *atomic* request requires exclusive access to its target cache line through the cache coherence protocol before updating the line in an L1 cache. The cache then executes the request's ALU operation and updates the cache line in one step. The exclusive access and one-step execution inside the cache guarantees the atomicity of the `AMO` instruction. The previous value at the target address is returned to the executing CPU pipeline after the *atomic* memory request is completed in the cache.

LR/SC Instructions – An `LR` instruction reserves exclusive access to a shared address. An `SC` instruction performs an update to the value at the shared address only if there is still a valid reservation on the address. The pair of instructions is commonly used to perform lock-free atomic read-modify-write operations. Using an `LR/SC` instruction pair to synchronize multiple threads is prone to livelock. An `SC` instruction executed by thread A may never succeed if an `LR` instruction

executed by another thread continually invalidates thread A's reservation. RISC-V guarantees an SC instruction will eventually succeed under certain constraints on the number and type of instructions between an LR/SC pair [ris18].

The implementation of LR/SC in gem5 maintains a per-HW-thread list of reserved addresses. When an LR instruction is executed in a HW thread, a snoop request is placed on a cache coherence bus to revoke any reservation of the instruction's target address. Once all reservations in other HW threads are invalidated, the address is pushed into the requesting thread's reservation list. Later, when executing an SC instruction, the HW thread checks if the instruction's target address still exists in the thread's reservation list. If so, the SC instruction succeeds, and the address is popped off the list. Otherwise, the instruction fails. If a HW thread receives a snoop request for an address, it revokes any matched entry in its own list. We made the reservation list structure private for each HW thread to correctly support LR/SC in multi-core simulations. To implement RISC-V's livelock freedom guarantee, we modified the L1 cache to hold off processing LR snoop requests to an address for a bounded period of time if there is an active reservation on the address.

Release Consistency Model – RISC-V supports a release consistency model [GLL+90]. Under the model, memory operations are free to be re-ordered unless there is a memory fence (FENCE) instruction between them. RISC-V also supports two memory ordering flags (*acquire*, *release*) encoded in two corresponding bits (*aq*, *rl*) inside AMO and LR/SC instructions. The *acquire* flag prevents memory operations *after* an AMO or LR/SC instruction from being re-ordered with respect to the instruction. The *release* flag prevents memory operations *before* an AMO or LR/SC instruction from being re-ordered with respect to the instruction.

The RISC-V FENCE instruction is implemented in the current version of gem5. Its implementation prevents memory instructions after the FENCE instruction from being issued until all memory instructions before the FENCE instruction retire. To implement the memory ordering bits embedded in AMO and LR/SC instructions, we used gem5's micro-operation feature that allows breaking an instruction into a sequence of smaller micro-operations to be executed by the CPU pipeline. Depending how *aq* and *rl* are set in an AMO or LR/SC instruction, we inserted a fence micro-operation(s) before and/or after the AMO or LR/SC instruction. Table 3.2 shows all four configurations of *aq* and *rl* bits, their memory ordering semantics, and their corresponding sequences of micro-operations.

aq	*rl*	Ordering Semantics	Micro-op Sequence
0	0	Relaxed	`AMO/LR/SC`
0	1	Releasing	`fence`; `AMO/LR/SC`
1	0	Acquiring	`AMO/LR/SC`; `fence`
1	1	Sequentially consistent	`fence`; `AMO/LR/SC`; `fence`

Table 3.2: Micro-Operation Sequences for `AMO` and `LR/SC` Instructions – Each sequence corresponds to a configuration of *aq* and *rl* bits set in the instructions and a memory ordering rule in the release consistency model.

3.3 Functional Validation

In this section, we describe our functional validation of the RISC-V implementation in gem5. We first show a major challenge with using gem5's current regression tests to validate the implementation. We then explain our approach and describe how we applied it to validate the functionality of thread-related system calls and RISC-V instructions.

Challenge – Although gem5 already has a regression test suite including some C/C++ benchmarks and their reference outputs, using these tests to debug a complex CPU model is challenging. A C/C++ benchmark, even a very simple one, can compile to thousands of instructions. Since a compiler can optimize the benchmark, the generated assembly code is often hard to understand. When the benchmark fails, tracing the problem through the large number of instructions is difficult and time-consuming. Debugging a multi-core CPU model that runs multi-threaded applications is even worse. A problem can appear to happen in a code region that is far from where the actual bug occurs. Therefore, we need a better approach to validate functionality in gem5.

Approach – Instead of using C/C++ benchmarks to validate a model in gem5, we used extensive, well-crafted assembly and low-level C unit tests. Each small test written in assembly code stresses a single instruction or system call without extra complexities coming from any C/C++ library and compiler. We used low-level C unit tests to discover missing functionality that is used in real libraries (e.g., GNU `pthread` library). By thoroughly testing an implementation at a low level, we can be more certain about the correctness of each instruction and system call.

Implementation – We applied the approach to validate functionality of the single-threaded and multi-threaded implementation of RISC-V in gem5.

For the single-threaded implementation, we leveraged an extensive assembly test suite in the open-source RISC-V tool chain[1]. The RISC-V test suite is designed to run on bare metal systems

[1] `https://github.com/riscv/riscv-tests`

without any OS support, and it communicates to a host machine to inform test outputs. However, gem5 simulates systems with OS support, so to integrate the suite into gem5, we added a new testing environment that ignores the initial to-host communication setup in the original suite and calls the `exit` system call with an exit status number denoting which test case fails.

For the multi-threaded implementation, we built our own assembly and low-level C unit tests. Since we do not want to have the complexity of threading libraries (e.g., GNU `pthread` library) in our assembly tests, we wrote a minimal threading library written in assembly code to simplify developing new multi-threaded assembly tests. The library includes minimal functionality to create, synchronize, and terminate threads using the `clone`, `futex`, and `exit` system calls. We first validated the implementation of these system calls. Then we built new tests using the minimal library to validate the implementation of `AMO` and `LR/SC` instructions on a multi-core system. The multi-threaded tests are focused on inducing potential race conditions and other synchronization bugs that are impossible to detect in single-threaded tests. Low-level C unit tests were built to detect missing functionality used in the GNU `pthread` library. Each unit test is focused on a single pthread function (e.g., `pthread_create`, `pthread_join`, and `pthread_mutex_lock`).

Using our approach, we were able to detect and fix numerous bugs in gem5's CPU models efficiently. Some of the bugs are related to incorrect suspension and resumption of HW threads in a CPU pipeline, which would be hard to reveal, trace, and fix using only C/C++ benchmarks. Some other bugs happened in the out-of-order CPU pipeline's memory disambiguation unit and load/store queue. Without the ability to control interactions between memory instructions, it would be challenging to reproduce and trace such memory-related bugs. There were a couple of bugs related to incorrect interpretation of the `clone` system call's API. They were easily detected in our simple assembly tests.

3.4 Timing Validation

Detailed CPU models in gem5 are meant to be used as generic models and are not validated against an actual cycle-accurate micro-architecture [NMHS15]. Users of the models often need to re-configure them and validate their performance against a target micro-architecture [GPD+14]. In this section, we first explain the challenges involved with timing validation for the RISC-V implementation in gem5, before describing a general approach for such timing validation. We then

show an example of how we validated a multiplier unit in gem5's in-order CPU model against the multiplier unit in the Rocket chip.

Challenge – Timing or performance validation of a CPU model in gem5 is often performed using C/C++ benchmarks (e.g., SPEC CPU2006) [GPD+14]. Performance counters (e.g., the total number of cycles and instructions) are used to compare the performance of the simulated system vs. the target system. There are two main problems with this approach. First, it is often challenging to detect a performance bug. Since this approach relies on very general performance statistics of high-level benchmarks, different simulation errors and performance bugs can together skew the overall performance results [NMHS15]. Second, parameters of a model can be tuned to make the model appear to have correct timing behavior only in a small set of benchmarks [NMHS15]. When running the model with a benchmark that heavily uses HW units that are not validated, the model's performance may become incorrect.

Approach – Instead of using C/C++ benchmarks to validate the performance of a whole CPU model in gem5, we argue for an incremental validation approach using assembly micro-benchmarks. Each micro-benchmark is carefully designed to validate a specific HW unit (e.g., branch predictor, multiplier, decoder, and memory load/store queue) using a sequence of instructions that heavily use the target unit. The sequence's performance is measured through HW cycle and instruction counters. Some techniques including cache warm-up and loop unrolling can be applied to minimize interference from other HW units.

Implementation – In this work, we applied the approach to validate the multiplier's performance in gem5's in-order CPU model against the multiplier in a Rocket chip. We configured the Rocket chip generator to generate an in-order CPU model that has an 8-cycle iterative multiplier. We wrote a micro-benchmark that executes 500 `mul` instructions back-to-back with minimal read-after-write dependencies. We did not use branch instructions to loop through the sequence to prevent the branch predictor from affecting the sequence's timing behavior. We also warmed up the instruction cache by pre-executing the sequence to minimize interference from the memory system. We used `rdcycle` and `rdinstret` instructions to count the number of cycles and dynamic instructions in the sequence of interest.

For each `mul` instruction, the Rocket core's iterative multiplier spends one cycle taking input values in, eight cycles doing the multiplication, and another cycle pushing the output value to the next pipeline stage. In total, each `mul` instruction takes 10 cycles to complete in the Rocket core's

Metrics	gem5's In-Order Model	Rocket In-Order Model
DInst	503	503
CPU Cycle	5010	5003
CPI	9.96	9.95

Table 3.3: Timing Validation of the Multiplier Unit in gem5's In-Order CPU Model against the Multiplier in the Rocket Chip – Performance numbers are for the sequence of 500 `mul` instructions. DInst = Dynamic Instruction. CPI = Cycles Per Instruction.

multiplier. The multiplier is not pipelined, so it cannot execute new `mul` instructions until the current one completes. To model this iterative multiplier in gem5, we configured gem5's in-order CPU's multiplier unit to have 10-cycle issue and execution latency. Table 3.3 shows performance numbers for both models after the validation. The multiplier in gem5's in-order model performed close to the multiplier in the Rocket core in terms of the instruction throughput.

This validation of the multiplier unit is a starting point, and future validation of other HW units in gem5's CPU models are necessary. Our work suggests an incremental timing validation approach that can be applied to gem5's CPU models.

3.5 Evaluation Workload

We chose 13 applications from the Ligra benchmark suite [SB13b] as our workload (see Table 3.4). Ligra is a graph processing framework designed for shared-memory systems. It supports multiple threading libraries including Cilk [int13] and OpenMP [ope08]. Ligra provides two lightweight routines called *mapEdge* and *mapVertex* to process subsets of edges and vertices respectively. Multiple subsets can be processed in parallel. Many applications in Ligra show irregular characteristics in their parallelism and memory access patterns.

In terms of input graphs, we picked two real-world graphs called *socfb-American75*, which is from a collection of Facebook networks [RA16], and *Kneser_10_4_1*, which is from a sparse matrix collection [DH11]. *socfb-American75* is a dense graph which has around 6,000 vertices and 440,000 edges. *Kneser_10_4_1* is a sparser graph with around 350,000 vertices and 990,000 edges.

In this work, we used four different versions of each Ligra application: Serial, OpenMP-Static (OMP-S), OpenMP-Guided (OMP-G), and Cilk-Work-Stealing (Cilk-WS). We used the OpenMP

Applications	Input Graphs	DInst (M)			
		Serial	OMP-S	OMP-G	Cilk-WS
BC	kneser	1152	1148	1149	1196
BFS	kneser	502	502	502	523
BFSCC	kneser	607	988	1079	2595
BFS-Bitvector	kneser	1042	1059	1059	1082
Components	kneser	1719	1719	1735	1746
MIS	kneser	646	810	844	835
KCore	socfb	628	641	648	1009
PageRank	socfb	2858	2861	2862	3241
PageRankDelta	socfb	400	401	401	441
Radii	socfb	268	268	268	283
Triangle	socfb	1069	1069	1069	1121
BellmanFord	socfb	138	137	137	147
CF	socfb	2670	2670	2670	2889

Table 3.4: List of Ligra Applications – Serial = single-threaded versions. OMP-S = multi-threaded versions using OpenMP runtime with the static scheduling policy. OMP-G = multi-threaded versions using OpenMP runtime with the guided scheduling policy. Cilk-WS = multi-threaded versions using Cilk runtime with the work-stealing policy. DInst = dynamic instruction count. Multi-threaded versions are simulated on a 4-core in-order CPU in gem5. The reported numbers are only for regions of interest that include only graph computation phases and exclude input graph initialization phases in each application.

and Cilk task parallel runtime libraries shown in Table 3.5. We used the open-source RISC-V tool chain including the GNU compiler and GNU libc to compile the Ligra applications. We only made a minor modification to the compiler to be able to compile the Cilk runtime. For the OpenMP runtime, the tool chain works out-of-the-box.

3.6 Simulator Performance

In this section, we show the performance of gem5 simulating four Ligra applications: *PageRank*, *PageRankDelta*, *KCore*, and *Triangle*. We first show a performance comparison between gem5 and the Chisel C++ RTL simulator. Then we show that using the fast-forwarding feature provided with gem5 increased its simulation speed up to 2× in the set of studied benchmarks. Finally, we present the scalability of gem5's performance in multi-core simulations. We ran our experiments on a multi-core machine with Intel Xeon E5620 CPUs running at 2.40 GHz.

gem5 vs. Chisel C++ RTL Simulator – We chose the Rocket chip generator as our baseline [AAB+16]. The generator is written in Chisel and can be configured to generate an RTL

Runtime	Chunk Size	Task Assignment	Work Stealing
OMP-S	Fixed	Static	No
OMP-G	Adaptive	Dynamic	No
Cilk-WS	Fixed	Dynamic	Yes

Table 3.5: Threading Libraries Used in Our Experiments – Fixed chunk size = the task chunk size is fixed for a particular parallel region. Adaptive chunk size = task chunk size is adjusted dynamically to better handle workload imbalance within a parallel region. Static task assignment = tasks are assigned statically before entering a parallel region. Dynamic task assignment = tasks are assigned on the fly during the execution of a parallel region. If work stealing is available, a thread can steal tasks from other threads.

model in Verilog. Verilator is then used to compile the RTL model into a C++ cycle-accurate model that is significantly faster than the Verilog RTL model. We used a RISC-V proxy kernel to handle system calls executed in the Chisel C++ RTL simulator instead of simulating a full Linux kernel for better simulation performance. Unfortunately, the proxy kernel does not support the `clone` and `futex` system calls. Therefore, we are unable to perform a multi-threaded simulation performance comparison between gem5 and Chisel C++ RTL simulator. For this comparison, we used the gem5 configuration with the validated multiplier and the Rocket core as described in Section 3.4.

We chose the single-threaded version of *KCore* from the Ligra benchmark suite. We measured the end-to-end simulation time of both simulators using the `time` Linux command. We counted the number of cycles and instructions simulated in the simulators using `rdcycle` and `rdinstret` instructions. Table 3.6 shows a performance comparison between the two simulators. The Chisel C++ RTL simulator simulated slightly more instructions than gem5 due to the extra instructions required for executing system calls in the RISC-V proxy kernel. The Chisel C++ RTL simulator simulated roughly 36% more cycles than gem5 did. One possible reason for the difference is that despite using the validated multiplier in gem5, the performance of other HW units (e.g., branch predictor, memory load/store unit, and memory system) has not been validated.

Despite differences in the absolute instruction and CPU cycle counts, the average number of cycles per second and instructions per second provide intuition into the relative performance of both simulators. gem5 is more than an order of magnitude faster compared to the Chisel C++ RTL simulator. This large difference is one of the key benefits of cycle-approximate vs. cycle-accurate simulation. To provide context, simulating 1B instructions would take almost three days when

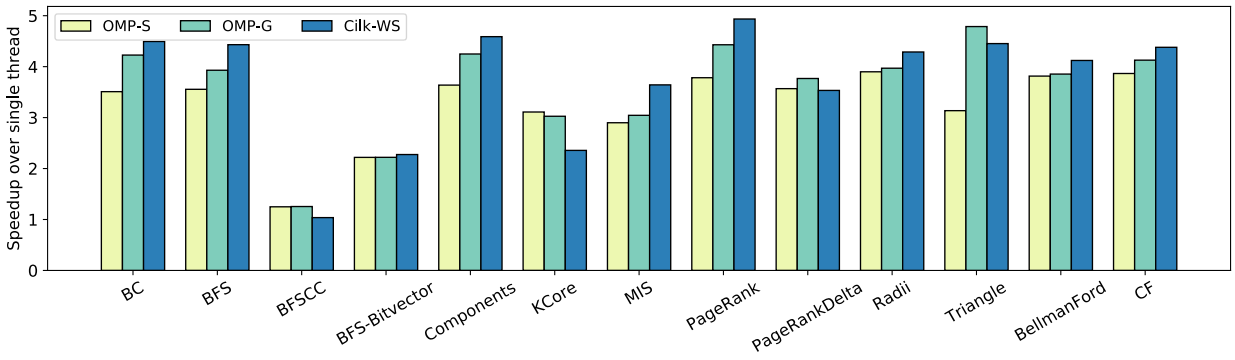


Figure 3.1: Performance of Different Task Scheduling Mechanisms in a Heterogeneous System

Metrics	gem5 Simulator	Chisel C++ RTL Simulator
DInst (M)	1125	1161
CPU Cycle (M)	1440	1956
KCPS	225	7
KIPS	175	4

Table 3.6: Performance Comparison between gem5 and Chisel C++ RTL Simulator – Both simulators run the same single-threaded binary of *KCore*. DInst = dynamic instruction. KCPS = kilo CPU cycles per second. KIPS = kilo instructions per second.

using the Chisel C++ RTL simulator, but this same simulation would take less than two hours when using gem5.

Fast-Forwarding Simulation – gem5 can fast forward a sequence of instructions by simulating them with a simple CPU model that only captures functional behavior and excludes the timing behavior of the CPU pipeline, the memory system, or both. Table 3.7 shows two simple and two detailed CPU models that are available in gem5. Switching between a simple and a detailed model can happen on any given simulation tick. We modified gem5 to support a custom control-status register (CSR) to enable software running on the simulator to indicate when to switch into or out of a detailed CPU model. We used this CSR to fast forward our benchmarks during their input graph initialization phase.

Table 3.8 shows performance improvements of gem5 when using fast-forwarding. All applications studied in this section used the same input graph, so they all had the same number of fast-forwarded instructions. Depending on the length of the initialization phase with respect to the full execution time, fast-forwarding results in a speedup of 1.16–2.01×.

CPU Models	CPU Pipeline	Memory
AtomicSimpleCPU	Simple	Simple
TimingSimpleCPU	Simple	Detailed
MinorCPU	Detailed In-order	Detailed
DerivO3CPU	Detailed Out-of-order	Detailed

Table 3.7: Available CPU Models in gem5. – Simple models only simulates functional behaviors while detailed models capture both functional and timing behaviors.

Benchmarks	DInst-FF (M)	DInst-Detailed (M)	Speedup
PageRank	496	2858	1.16x
PageRankDelta	496	400	2.01x
KCore	496	628	1.67x
Triangle	496	1069	1.42x

Table 3.8: Performance Speedup over Full Simulations without Fast-Forwarding Mode – DInst-FF = number of dynamic instructions that are fast-forwarded. DInst-Detailed = number of dynamic instructions that are simulated in the detailed mode.

Performance Scalability – To understand the performance scalability of gem5 in multi-core simulations, we ran the four benchmarks using the OpenMP runtime and the static scheduling policy on systems with a different number of in-order CPU cores. Figure 3.2 shows the average number of simulated instructions per second. The result shows that gem5's performance scales well with the number of simulated CPU cores. When simulating more CPU cores, gem5's does slows down a little bit since it simulates more thread communication events between cores.

3.7 Design Space Exploration

In this section, we are interested in using the implementation of RISC-V in gem5 to study how irregular applications using different threading libraries and task scheduling policies perform on a heterogeneous multicore system. In particular, we are interested in the relative performance of static, guided, and work-stealing task scheduling policies for graph applications executing on a system with both simple and complex cores [TWB16].

We used gem5 to model a quad-core cache-coherent RISC-V system with two simple in-order and two complex out-of-order cores. Each core has its own private L1 cache. Constructing this model is straight-forward in gem5 due to its modular design and simple Python-based configuration

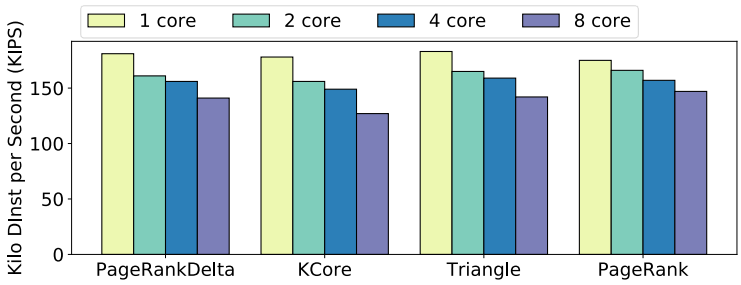


Figure 3.2: Performance of gem5 in Multi-Core Simulations

interface. We only needed to make minor changes in the Python configuration. In contrast, building such a system in RTL would be very challenging.

We ran all 13 Ligra applications with different task scheduling policies. Figure 3.1 shows the speedup of each configuration over the single-threaded version of Ligra applications. The multi-threaded versions of most applications except *BFSCC* performed significantly better than their single-threaded versions. In *BFSCC*, our selected input graph results in a large serial region. On average, the quad-core heterogeneous system achieved a 3.53× speedup over the single-core system.

Dynamic task scheduling policies (i.e., OMP-G and Cilk-WS) generally performed better than the static task scheduling policy. This is due to the workload imbalance in many graph applications and the heterogeneity of the studied system. Complex and simple cores complete tasks at different rates. A dynamic task scheduling mechanism helps balance the workload between cores, which helps increase the overall throughput. In terms of performance, OMP-S is never the best choice for Ligra applications except *KCore* and *BFSCC*. Most of parallel regions in *KCore* are highly regular, and its tasks are light-weight. *BFSCC* has little parallelism, so its multi-threaded versions did not perform much better than its single-threaded version.

3.8 Conclusion

We presented our work on simulating multi-threaded RISC-V systems in gem5. We contributed an implementation of thread-related system calls and synchronization instructions to the existing RISC-V implementation in gem5. We also modified gem5's CPU models to simulate multi-threaded workloads correctly. We showed our validation approach and how we applied the ap-

proach to validate the functional and timing behavior of the implementation. We presented gem5's simulation performance in comparison to the Chisel C++ RTL simulator, the simulation speedup achieved by using the fast-forwarding feature in gem5, and gem5's scalability in multi-core simulations. Our implementation in gem5 can run real-world applications and task parallel runtime libraries including OpenMP and Cilk. We showed a small design space exploration to illustrate how gem5 can help system designers explore different design options quickly.

CHAPTER 4
big.VLITTLE: EVOLUTIONARY SPECIALIZATION FOR MODERN VECTOR ARCHITECTURES

Single-ISA heterogeneous multi-core architectures offer a compelling high-performance and high-efficiency solution to executing task-parallel workloads in mobile systems on chip (SoCs). In addition to task-parallel workloads, many data-parallel applications, such as machine learning, computer vision, and data analytics, increasingly run on mobile SoCs to provide real-time user interactions. Next-generation scalable vector architectures, such as the RISC-V Vector Extension and Arm SVE, have recently emerged as unified vector abstractions for both large- and small-scale systems. In this chapter, we propose novel area-efficient high-performance architectures called big.VLITTLE that support next-generation vector architectures to efficiently accelerate data-parallel workloads in conventional big.LITTLE systems through the evolutionary specialization approach. big.VLITTLE architectures reconfigure multiple little cores on demand to work as a decoupled vector engine when executing data-parallel workloads. Using the cycle-level modeling methodology presented in Chapter 3, we show that a big.VLITTLE system can achieve 1.6× performance speedup over an area-comparable big.LITTLE system equipped with an integrated vector unit across multiple data-parallel applications and 1.7× speedup compared to an aggressive decoupled vector engine for task-parallel workloads.

4.1 Introduction

Modern mobile systems on chip (SoCs) adopt single-ISA heterogeneous multi-core architectures (e.g., Arm big.LITTLE) to offer a compelling high-performance and high-efficiency solution for task-parallel workloads [KTR+04, KFJ+03] in many commercial devices [Gwe14b, Gwe14a, Dem13, Gwe13, app20]. These architectures consist of several high-performance power-hungry out-of-order big cores and multiple high-efficiency low-power in-order little cores. This ISA homogeneity and micro-architecture heterogeneity enable high performance and efficiency by seamlessly distributing high- and low-intensity compute tasks to high-performance and high-efficiency cores respectively [Ran13, ZR13].

In addition to task-parallel workloads, data-parallel applications are emerging in mobile SoCs to fully utilize their increasing compute power and sensing capabilities. Workloads such as aug-

mented and virtual reality (AR/VR) [CDM+18, HDJ+20], natural language processing [BDVJ03, CW08], facial and voice recognition [PVZ15], and image processing [TMB14] increasingly rely on in-device computing power instead of cloud servers to deliver real-time interactions with humans [RSM+11, WBC+19a, ZCL+19, LCS+19, WAZ+19]. These applications often use compute-intensive data-parallel computer vision, machine learning, and data analytic algorithms to process a large amount of data in real time. Since mobile SoCs operate under a tight power and area budget, such increasing computational demand poses a significant challenge to design both **high-performance** and **high-efficiency** mobile architectures to accelerate data-parallel workloads.

The need to efficiently accelerate data-parallel workloads has led to an emergence of next-generation scalable vector architectures exemplified by the RISC-V Vector Extension (RVV) [RIS21] and the Arm Scalable Vector Extension (Arm SVE) [SBB+17]. Traditional vector architectures are typically implemented as either large high-performance variable-length decoupled vector engines [Rus78, DVWW05, KTHK03, TNH+06] in super-computing systems or modest area-efficient fixed-length packed-SIMD integrated vector units (e.g., Intel AVX) in mobile and desktop systems. Next-generation vector architectures strive to provide unified scalable vector abstractions for both large decoupled vector engines that yield superior performance with significant area overheads and small integrated vector units that require modest extra silicon area with modest performance improvement compared to an out-of-order scalar core.

In this chapter, we propose novel area-efficient high-performance architectures called big.VLITTLE that adopt next-generation vector architectures to accelerate data-parallel workloads in widely used big.LITTLE systems. big.VLITTLE architectures achieve both high performance and area efficiency by reconfiguring a cluster of little cores as a decoupled vector engine on demand when executing data-parallel workloads. When a big.VLITTLE system executes in vector mode, its big core fetches, decodes, and sends vector instructions to its associated cluster of little cores, which allows decoupling memory accesses and vector computation. Little cores reconfigure their scalar pipelines into vector execution lanes, leverage their physical register files to store vector register elements, transform their level-one cache subsystem to provide high memory bandwidth, and work together as a decoupled vector engine.

Due to its reconfigurability, big.VLITTLE architectures do not need to add area-expensive components such as wide execution pipelines and vector register files typically required in large decoupled vector engines. Compared to integrated vector units, big.VLITTLE systems can provide

longer vector length and higher memory bandwidth, which results in better performance. When not executing in vector mode, big.VLITTLE systems incur no performance overhead for multi-threaded task-parallel workloads since they operate in the same way as equivalent big.LITTLE systems. Our cycle-level performance evaluation shows that a big.VLITTLE system with one big and four little cores can achieve 1.6× speedup over an area-comparable big.LITTLE system equipped with an integrated vector unit for data-parallel workloads from the Rodinia suite [CBM+09], RiVec suite [RHP+20], and a genomics benchmark suite. For task-parallel applications, the big.VLITTLE system is 1.7× faster than an aggressive decoupled vector engine for applications from the Ligra benchmark suite [SB13a]. Our post-synthesis area evaluation shows the big.VLITTLE system incurs less than 5% overhead compared to a cluster of four little cores and their L1 private caches. Our design space exploration shows the potential of using voltage/frequency scaling to boost the little cores while slowing down the big core to further increase both performance and power efficiency of the big.VLITTLE system.

Our key contributions include: (1) a new reconfigurable little core cluster that leverages its existing scalar execution pipelines and reconfigures its scalar register files to operate as a high-performance decoupled vector engine; (2) a novel reconfigurable L1 cache subsystem that can turn private L1 data caches of little cores into a logically shared multi-bank L1 data cache for vector execution and re-purpose SRAM arrays in L1 instruction caches as data buffers to enable high vector memory bandwidth; (3) a detailed cycle-level performance evaluation of a big.VLITTLE system compared to an area-comparable conventional big.LITTLE system with an integrated vector unit and an aggressive decoupled vector engine, and a VLSI-level area analysis demonstrating the big.VLITTLE system's area efficiency; and (4) a design space exploration showing the potential of voltage/frequency scaling in increasing performance and power efficiency of a big.VLITTLE system.

4.2 The Resurgence of Vector Architectures

Traditional vector architectures can be classified into two classes: long-vector and packed-SIMD architectures. A recent resurgence of interest in adopting vector abstractions for emerging data-parallel workloads has led to next-generation vector architectures that provide unified abstractions for both high-performance and commodity systems. In this section, we discuss a taxonomy

	Features	Long-Vector	Packed-SIMD	Next Generation
ISA	Vector length	scalable, long	fixed, short	scalable
	Element width	fixed	variable	variable
	Predication	full	limited	full
	Cross-element ops	limited	full	full
	Memory gather/scatter	full	limited	full
uArch	Vector register file	decoupled	integrated	either
	Speculative execution	yes	no	either
	Compute pipeline	decoupled	integrated	either
	Memory bandwidth	large	modest	either
	Memory latency	high	low	either

Table 4.1: A Taxonomy of Vector Architectures

of both traditional and next-generation vector architectures as well as their key tradeoffs (see in Table 4.1).

4.2.1 Long-Vector Architectures

Long-vector architectures target large-scale systems, such as supercomputers, to execute highly data-parallel and regular workloads. Most long-vector architectures support a variable vector length that scales with a specific implementation of the architecture [Rus78, DVWW05]. The width of each vector element is typically fixed. Supporting cross-element instructions, such as reducing and shuffling vector elements, requires expensive hardware due to long vector length, so such instructions are usually not fully supported in long-vector architectures. Instead, memory gather and scatter instructions are available to support complex data movements through memory.

Vector units in supercomputing vector machines [Rus78, Abt07, KTHK03, TNH$^+$06, EML88] are typically decoupled from their control cores. Vector units have separate large vector register files and wide execution lanes. To sustain a high compute throughput and fully utilize all execution lanes, long-vector machines require large memory bandwidth, so they are typically connected to highly banked memory systems (e.g., 1024 memory banks [SWL$^+$92]).

4.2.2 Packed-SIMD Architectures

Since packed-SIMD ISAs often target multimedia workloads in commodity hardware, their vector lengths are typically limited and fixed. Early packed-SIMD architectures, such as Intel

MMX [PW96] and SEE [int07], are designed to handle sub-word computations on general-purpose registers. More recent packed-SIMD architectures, such as Intel AVX-128, extend their vector lengths beyond the width of a single word (e.g., 128 bits). To support multiple SIMD operations in a fixed hardware vector length, packed-SIMD ISAs support variable element widths to dynamically change the effective number of elements depending on applications. For example, a 128-bit wide packed SIMD ISA can support two 64-bit and four 32-bit operations. Since packed-SIMD ISAs are designed for commodity hardware, their support for complex vector memory instructions, such as gather-load and scatter-store instructions, is limited.

Packed-SIMD units are often tightly integrated with their control processors. Most of them share the same register files (i.e., typically floating-point register files) with the control processors although some recent short-vector units, such as ones in Intel Knights Landing [SGC+16], may have dedicated SIMD register files. Floating-point and SIMD instructions typically share the same execution pipelines. Since the number of vector elements is small, SIMD units typically share the same memory interface with their control processors to private data caches. Therefore, compared to long-vector machines, short-vector units have relatively modest memory bandwidth.

4.2.3 Next-generation Vector Architectures

Conventional vector architectures target two drastically different domains: high-performance computing in large-scale systems and multimedia workloads in commodity hardware. However, recent interest in data-parallel workloads have driven a trend to converge both conventional design approaches into next-generation vector architectures that are flexible enough to cover a wider spectrum of workloads and hardware implementations [SBB+17].

Modern vector ISAs, such as the Arm Scalable Vector Extension (SVE) [SBB+17] and the RISC-V Vector Extension (RVV) [RIS21], adopt a vector-length-agnostic (VLA) design similar to conventional long-vector architectures. This VLA design enables such ISAs to target a wide range of implementations with different hardware resource constraints. It also allows executing the same vector code on multiple vector machines with different hardware vector lengths without recompiling the code and/or rewriting compiler intrinsics. Cross-element, load-gather, and store-scatter instructions are also supported in these ISAs to increase the overall scope of applications that can be vectorized.

Due to their flexibility, these next-generation vector ISAs can target both large-scale (i.e., decoupled from a control processor) micro-architectural implementations and small-scale (i.e., tightly integrated with a control processor). Example machines include Xuantie-910 [CXL+20] and Ara [CSZ+20] implementing RVV, and Fugaku A64FX [Sat20] implementing Arm SVE.

4.3 big.VLITTLE Architectures

The reconfigurability of big.VLITTLE architectures helps achieve the performance level of decoupled long-vector engines while minimizing area overheads as in integrated vector units. In this section, we first provide an overview of big.VLITTLE architecture and then provide details on how multiple aspects of a next-generation vector architecture are implemented in big.VLITTLE.

4.3.1 Architectural Overview

big.VLITTLE architectures support both scalar and vector execution modes. In the scalar mode, big and little cores execute instructions independently as they do in conventional big.LITTLE systems, and components added to support the vector execution mode are disabled. In the vector mode, the big core becomes a control core, and the little cores work together as a single decoupled vector engine called VLITTLE. The big core executes scalar instructions while vector code is executed in the VLITTLE engine. A vector instruction waits at a vector dispatching unit in the big core until it is at the head of the ROB, and then is dispatched to the VLITTLE engine. If the instruction does not write back to a scalar register, the big core can commit and remove it from the ROB. Otherwise, the big core waits for the VLITTLE engine to respond with a scalar value, writes the value back, wakes up any dependent instruction(s), and finally commits the vector instruction. Although the big.VLITTLE concept is applicable to both Arm SVE and RISC-V RVV, we use RISC-V RVV version 1.0 [RIS21] to explore the big.VLITTLE idea in the context of this chapter.

A big.VLITTLE system includes additional components to facilitate its vector execution on top of an equivalent big.LITTLE system. Figure 4.1 shows a big.VLITTLE instance with one big, four little cores, and additional vector-specific components. First, a *vector control unit* (VCU) controls the global architectural states of a VLITTLE engine (e.g., effective vector length), communications with the big core (e.g., receiving vector instructions), and the execution of all little

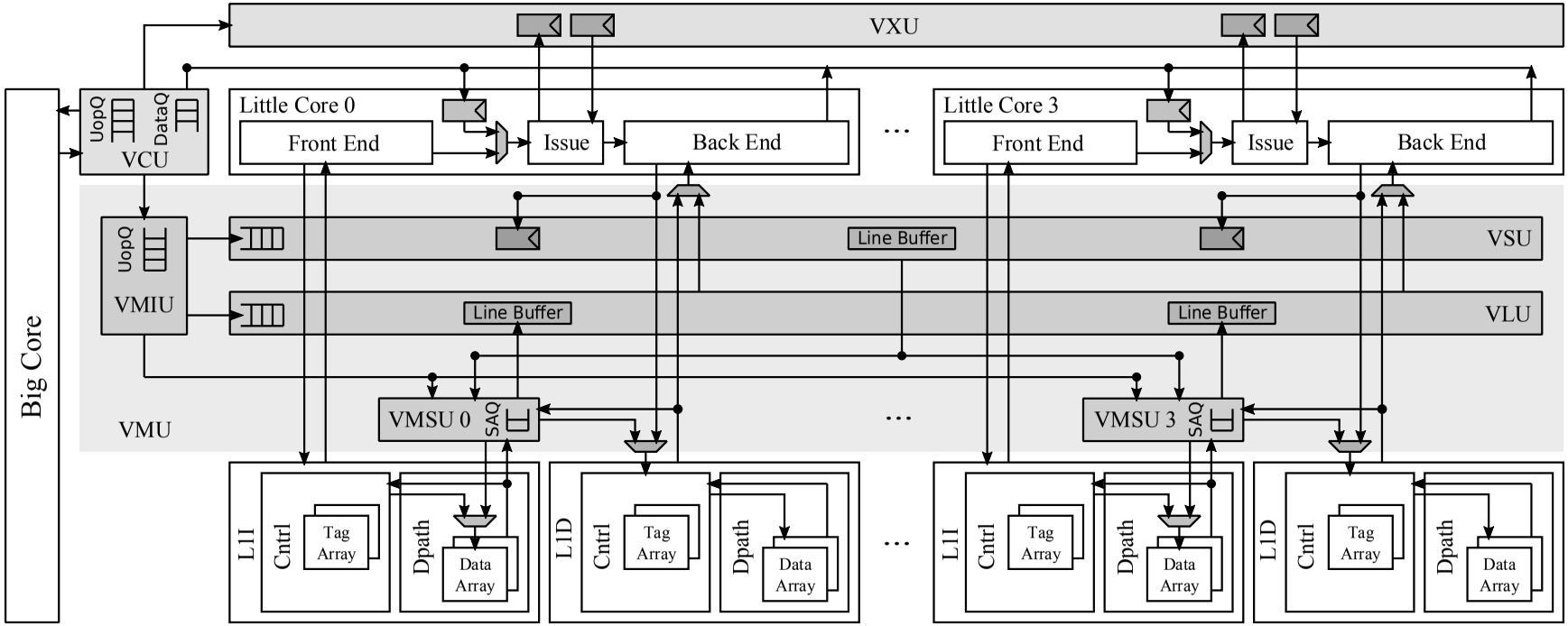


Figure 4.1: A big.VLITTLE system with One Big and Four LITTLE Cores – VCU = vector control unit; VXU = vector cross-element unit; VMU = vector memory unit; VMIU = vector memory issue unit; VMSU = vector memory slice unit; VLU = vector load unit; VSU = vector store unit; UopQ = micro-operation queue; DataQ = scalar data queue; SAQ = store-address queue. Components added to support the vector mode are shaded.

CSR	Description
mstatus	Machine status • *vs*: vector context status (off/clean/dirty)
vl	Effective vector length
vtype	Vector element type • *vsew*: vector element width • *vta*: vector tail agnostic (undisturbed/agnostic) • *vma*: vector mask agnostic (undisturbed/agnostic)

Table 4.2: A subset of vector-controlling CSRs in RVV

cores. Second, a *vector cross-element unit* (VXU) handles inter-core data communications to support cross-element vector instructions including vector permutation and reduction. Lastly, a *vector memory unit* (VMU) manages vector memory instructions by issuing requests to the memory subsystem and delivering data to little cores. Those additional components are pipelined so that they do not affect common critical paths and increase the cycle time of the little core cluster. Therefore, in the scalar mode, big.VLITTLE performs exactly the same as an equivalent big.LITTLE system. Several muxes are added to select the right input signals based on the current execution mode of the little cores. For example, an L1 data cache takes input requests from its little core's back-end in the scalar mode while receiving requests from the VMU in the vector mode.

4.3.2 Vector Control Support

We envision using a simple application interface managing an OS-privilege control status register (CSR) to switch between scalar and vector modes on demand. Applications running on the big core can switch into or out of the vector mode by requesting the OS to change the CSR. When switching into the vector mode, the OS allocates a group of little cores to form a VLITTLE engine, and those cores become unavailable to other processes. If one or multiple little cores are not readily available (e.g., busy with other processes), the OS can decide to either wait, pre-empt processes running on those little cores, or simply allocate a light-weight integrated vector unit in the big core for vector execution. Such OS-level resource scheduling decisions are left for future work. Once a little core is allocated to a VLITTLE cluster, its current thread context is saved to memory, and its pipeline is flushed. The overhead of saving a thread context into memory and flushing an in-order short pipeline is relatively small (e.g., 500+ cycles), especially when the target vectorized region

is large. A control register is then updated to indicate the core is now working in the vector mode. When switching out of the vector mode, the OS returns those cores to its scheduling pool, and they become available independent scalar cores. The switching typically happens at a coarse-grained level (e.g., application and kernel levels) to amortize its overhead.

big.VLITTLE architectures implement a weak memory consistency model which is a work in progress in RISC-V RVV. We introduce a vector memory fence `vmfence` to handle vector/scalar memory dependencies (e.g., unit-stride vector store followed by a scalar load to the same address) in software to avoid complex hardware checking for such dependencies between scalar and vector pipelines. The big core executes `vmfence`, waits for all outstanding scalar loads and stores to retire before sending a memory fence command to the VCU. The VCU blocks subsequent vector memory instructions from being issued to the VMU until all outstanding vector memory instructions to retire. Effectively all scalar and vector memory instructions before `vmfence` in their program order happen before all scalar and vector memory instructions after the `vmfence`. It is important to note that this software-managed vector/scalar memory fence solution is common in most decoupled vector machines [Asa98, Abt07, EAE+02] due to their large vector lengths and decoupled vector execution pipelines. Future auto-vectorization and compiler techniques, which are being actively researched for next-generation vector architectures [SBB+17, AS22], can help insert vector/scalar memory fences to guarantee the correctness of applications. In addition, any efficient hardware-managed solution for other decoupled vector machines would also be applicable to the big.VLITTLE architecture. Vector/vector memory dependencies (e.g., unit-stride vector store followed by an indexed vector load to the same address) are handled in hardware by the VLITTLE engine's VMU.

In a big.VLITTLE system, the VCU executes *vsetvl* that is a control instruction setting the effective vector length and vector element type. For each non-control vector instruction, the VCU issues multiple micro-operations to little cores and the VMU (only for memory instructions). The VCU buffers those micro-operations and their corresponding scalar data (only for vector instructions reading scalar register values) in command and data FIFO queues in order to enable decoupling of vector memory accesses and vector executions by issuing memory micro-operations to the VMU ahead of time. Not all vector instructions need to carry scalar values, so the scalar data queue needs not to be as deep as the command queue to minimize area overheads. In each cycle, the VCU processes the oldest micro-operation from the command queue and broadcasts it and its associated

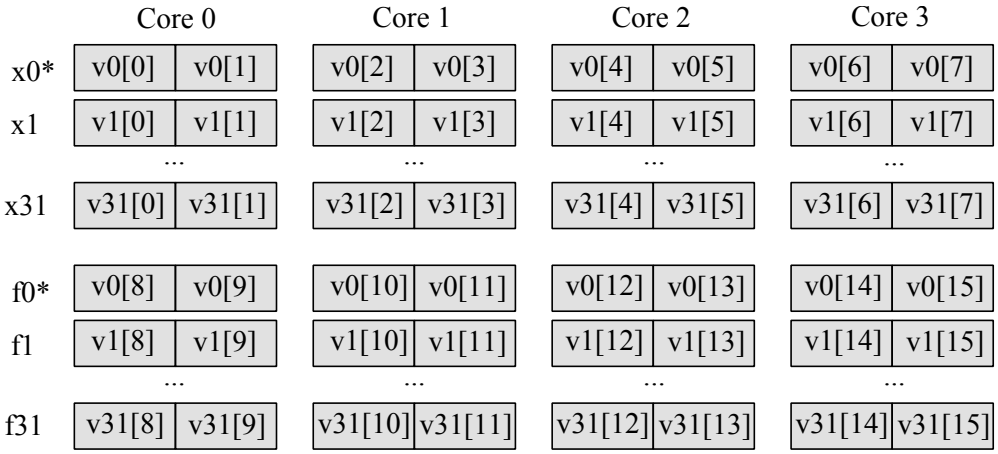


Figure 4.2: Mapping of 32-bit Vector Elements to Scalar 64-bit Registers in Four Little Cores – *xN* = scalar integer registers. *fN* = scalar floating point registers. *vN[m]* = m-th element in a vector register. Elements of the vector register 0 are mapped to newly added physical registers *x0** and *f0** in little cores.

scalar data (if any) to all little cores via a shared bus as shown in Figure 4.1. This command bus is pipelined to account for physical distance between little cores in a cluster so that it does not affect existing critical paths in the little cores.

4.3.3 Reconfigurable Little Cores

In big.VLITTLE architectures, scalar physical registers existing in little cores are re-purposed to implement all general vector registers except *v0*, which makes big.VLITTLE architectures area-efficient by not adding area-expensive vector register files as in conventional long-vector engines. Since *v0* register is used to store mask values according to the RISC-V RVV specification, predicated instructions can read up to three source operands. To avoid adding a read port to existing register files, *v0* is implemented using an extra register(s) added to each little core, which allows predicated instructions to read mask values in parallel with reading other source operands.

To maximize the hardware vector length in big.VLITTLE architectures, both integer and floating-point physical registers in little cores can be effectively used to support multiple vector element groups (chimes). Vector elements of the same group are always executed together in time. The actual number of element groups depends on the number of available physical registers in little cores. Figure 4.2 shows an example of a VLITTLE engine with four little cores, each of which has 32 integer and 32 floating-point physical registers (i.e., *x0-x31* and *f0-f31* respectively). The VLITTLE engine supports two vector element groups. Vector elements in the first and second

groups can be stored in the integer and floating-point registers respectively across all little cores. In addition, multiple consecutive vector elements can be packed into the same physical scalar register if their element width is less than the physical register's width. Figure 4.2 shows a case in which two 32-bit adjacent vector elements are packed into the same 64-bit physical register. With multiple element groups and packed vector elements, the example VLITTLE engine in Figure 4.2 can support a 512-bit hardware vector length by effectively using all physical registers in four little cores. Both optimizations increase the hardware vector length, reduce front-end instruction overheads, and hide long execution latency induced by complex instructions (e.g., multiplication, division, and memory instructions) in big.VLITTLE architectures.

For each vector instruction, the VCU issues multiple per-element-group micro-operations to little cores in order. Little cores receive micro-operations from the VCU at their issue stages. Their fetch and decode stages are not used in vector mode and hence disabled. In a little core, micro-operations are issued to its back-end execution pipelines in order as if they were normal scalar instructions. Except reading mask values from *v0*, no other change is added to a little core's issue stage to handle issuing micro-operations and reading operand values from the core's register file.

Back-end execution pipelines in little cores require minimal changes to support packed vector elements. For simple integer arithmetic micro-operations (e.g., addition), multiple vector elements packed into the same physical register can be processed in parallel with small area overheads to the existing little cores [Lee97]. For more complex integer micro-operations (e.g., division) and floating-point micro-operations, we serialize the execution on different packed vector elements in multiple cycles to avoid adding non-trivial hardware overheads to the existing little cores.

4.3.4 Cross-Element Instruction Support

The RISC-V RVV supports two types of cross-element instructions: vector permutation and vector reduction. Vector permutation instructions (e.g., `vrgather`) read per-element values from a source vector register and write them to different elements of a destination vector register. Vector reduction instructions read per-element values from a source register, perform a reduction operation to a single value, and write it to either the first element of a destination vector register (e.g., `vredsum`) or a scalar register (e.g., `vpopc`).

For each vector permutation instruction, the VCU generates two micro-operations: `vxread` and `vxwrite` per vector element group to little cores. `vxread` micro-operations read values of

their source vector elements and send them to the VXU. `vxwrite` micro-operations wait for the source values at the issue stage of each little core. Once receiving source values from the VXU, `vxwrite` micro-operations write the values to register files in little cores.

For each vector reduction instruction, the VCU first issues per-element-group `vxread` micro-operations to little cores to read values of source vector elements. The VCU then issues `vxreduce` micro-operation only to the first little core to perform a reduction. Once receiving a `vxreduce` micro-operation, the first little core's issue stage receives one value for each source vector element each cycle from the VXU, issues it to an execution pipeline, and waits for all source element values to arrive before completing issuing the micro-operation.

In order to move values across the little cores, we implement a light-weight uni-directional ring network connecting all little cores in the VXU. The ring network is pipelined to avoid affecting the cycle time of existing little core cluster. Other lower-latency network topologies (e.g., crossbar) are also viable although they may potentially incur higher area overhead compared to the uni-directional ring topology. The VXU receives per-element source values from little cores executing `vxread` micro-operations. The VXU receives requests for specific source elements from little cores executing `vxwrite` and `vxreduce` micro-operations. Once receiving all source values, the VXU iteratively shifts all per-element values by one hop each cycle. If a value's source element index matches with a request's source element index, the value is returned to the requesting core. The VXU completes shifting all elements after N cycles where N is the number of source vector elements. To avoid inter-instruction deadlocks and further complexities, the VXU processes at most one cross-element instruction at a time. Subsequent cross-element instructions must wait in the VCU for an outstanding instruction in the VXU to complete.

4.3.5 Reconfigurable Cache Subsystem

In a big.VLITTLE system, the VMU is the interface between its VLITTLE engine and memory subsystem. The VMU consists of a vector memory issue unit (VMIU), multiple vector memory slice units (VMSU), a vector load unit (VLU), and a vector store unit (VSU). Each VMSU corresponds to a private L1 data cache of a little core. In vector mode, private L1 data caches of all little cores work together as a logically shared cache with multiple address-interleaved slices or banks for the entire VLITTLE engine. We adopt an addressing scheme similar to a previous work [IKKM07] to distribute memory accesses across multiple L1 data caches. Given an effective

address, its bank bits are located between the block offset and index bits to minimize bank conflicts in the case of consecutive requests to adjacent cache lines. The VMU uses the bank bits to select an L1 cache for a given address. The remaining bits and the bank bits are used as a tag in L1 caches to disambiguate cache lines properly regardless of which mode the caches operate under and to avoid expensive cache flushes when the system switches between modes. After switching to vector mode, a cache line that is not in the right bank will eventually either be evicted (i.e., if not used) or migrated to the right bank (i.e., if used) by the cache coherence protocol.

For unit- and constant-stride memory instructions, their base virtual addresses are translated in the big core before they are dispatched to the VLITTLE cluster. The big core also checks their access ranges (i.e., spanning across multiple pages) using both base addresses and strides for potential page faults or invalid memory accesses. This early address translation mechanism allows the VMU to decouple unit- and constant-stride memory accesses from vector executions happening in the little cores. However, for indexed vector memory instructions, since their index values are stored in the VLITTLE cluster, per-element address translations happen in the little cores using their existing address translation hardware.

To enable decoupling of memory accesses and vector execution, the VCU sends load and store micro-operations to the VMIU as soon as it receives and processes memory instructions from its associated big core so that load requests can be issued to memory ahead of vector executions using their loaded values. In addition, the VCU also sends per-element-group micro-operations to little cores to write back values from the VLU to register files (`wb_ld`), read and send data to the VSU for store instructions (`rd_data`), and read and send memory indices to the VMIU for indexed memory instructions (`rd_idx`).

Vector memory issue unit (VMIU) – The VMIU processes load and store micro-operations from the VCU in the order they arrive. It breaks down a micro-operation into one or multiple cache-line-sized requests depending on its base address, stride, and indices. For unit-stride and constant-stride micro-operations, the VMIU uses their base addresses and strides to generate one memory request for a cache line per cycle. For indexed memory micro-operations, the VMIU waits for index values sent by `rd_index` micro-operations from little cores before generating memory requests. The VMIU tries to coalesce a small number of consecutive indices (e.g., four) into a single cache-line request in each cycle. Multiple memory requests can be generated for a memory micro-operation if it accesses across different cache lines. Each generated memory request is

tagged with a bit mask indicating active bytes in its cache-line-sized data. Once generated by the VMIU, a memory request is issued to a corresponding VMSU, based on its cache line address via a shared pipelined command bus. A small sequence number per request is sent to the VLU (for load requests) or VSU (for store requests) so that load and store data is processed in the order of their corresponding instructions in those units.

Vector memory slice unit (VMSU) – The VMSUs receive requests from the VMIU and operate at cache-line granularity to communicate with their corresponding L1 data caches. They also check memory dependencies between load and store requests. Each VMSU has a small content-addressable memory (CAM) holding addresses of outstanding store requests not yet issued to memory memory. Every load request arriving at a VMSU is checked against all previous outstanding store requests for potential address overlapping using their cache-line addresses. If a dependency is detected between a load and an outstanding store request, the load request is stalled until the store request is sent to the memory subsystem since the load request may read data written by the store request. Otherwise, the load request can be issued ahead of the store request to the memory.

The VMSUs need to buffer cache-line-sized data of all outstanding load requests (i.e., waiting for their cache responses) and store requests (i.e., waiting for their data from the VSU). To maximize the memory-level parallelism and enable memory accesses to run far ahead of vector execution, the amount of data buffering can be significant for each VMSU in memory-intensive workloads. Therefore, to minimize area overheads, we reconfigure SRAM data arrays in L1 instruction caches, which are unused by little cores in the vector mode, as circular FIFO queues for outstanding load and store requests. We simply use the SRAMs as FIFO queues and do not modify the cache control logic to avoid any timing overhead in the caches. Each VMSU controls head and tail pointers to its data queues and arbitrates between enqueueing and dequeueing operations since there is one read/write port in each SRAM. A VMSU writes response data from its L1 data cache into the load data queue and store data from the VSU into the store data queue. Once data for the oldest load request in a VMSU is ready, the VMSU sends the data to the VLU. Similarly, once data for the oldest store request in a VMSU is received from the VSU, the request is sent to the corresponding L1 data cache.

Vector load unit (VLU) – The VLU receives data responses from multiple VMSUs, breaks them down into per-core responses, and sends them to little cores. There are multiple line buffers,

each corresponding to a VMSU, to hold ready cache-line data responses from the VMSUs. The VLU processes data responses in the order their corresponding requests are generated by the VMIU since little cores expect their data to arrive in order.

For each unit- and constant-stride load response, a small hardware logic uses its first vector element index and stride information to slice its cache-line data into multiple vector-element-width responses, which are then pushed to existing load-store queues inside corresponding little cores. Since the VLU actively pushes data to little cores through `wb_ld` micro-operations, the cores can directly read the data from their internal load queues without sending extra requests to the VLU for the data. This reduces the latency of `wb_ld` micro-operations.

For indexed load micro-operations, actively pushing data to little cores from the VLU would require extra storage for index values and more complex logic to slice, shuffle, and align data elements of a cache-line response for multiple little cores. Therefore, little cores pull data from the VLU by sending per-vector-element requests to the VLU when executing `wb_ld` micro-operations. Each little core handles indexed memory `wb_ld` micro-operations as if they were scalar load instructions by leveraging its existing address calculation logic. Once receiving an indexed load request from a little core, the VLU extracts data from one of its line buffers and returns it to the core.

Vector store unit (VSU) – The VSU receives data elements from little cores and assembles them into cache-line-sized data blocks for store requests. The VSU processes store requests in the order they are issued by the VMIU. In each cycle, the VSU waits for little cores to send data elements for the oldest store request. Each little core executes `rd_data` micro-operations as if they were scalar store instructions. Per-vector-element data requests including both address and data are sent by little cores to the VSU. The VSU takes data elements from the cores and assemble them into a cache-line-sized data block by updating its line buffer. Once data in the line buffer is ready, the VSU sends the data to a VMSU, based on its cache-line address.

4.4 Evaluation Methodology

In this section, we describe a set of simulated systems, our cycle-level modeling methodology, and application benchmarks used to evaluate the performance of our big.VLITTLE architectures.

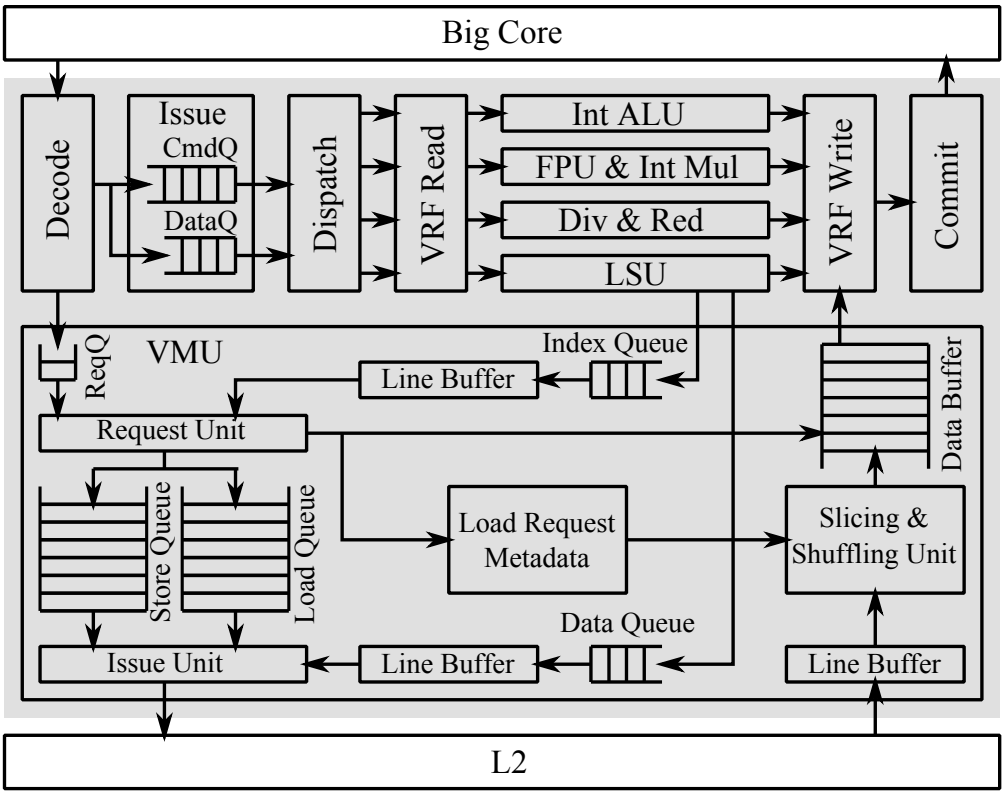


Figure 4.3: A Decoupled Vector Engine – CmdQ = command queue; DataQ = scalar data queue; VRF = vector register file; Int ALU = integer arithmetic & logic unit; FPU = floating-point unit; Int Mul = integer multiplication unit; Div = division unit; Red = reduction unit; LSU = load store unit; ReqQ = request queue.

Little Core (L)	• RISC-V ISA (RV64GC), single-issue, in-order • L1I cache: 1-cycle hit latency, 2-way, 32KB • L1D cache: 2-cycle hit latency, 2-way, 32KB
Big Core (b)	• RISC-V ISA (RV64GC), 8-way-issue out-of-order, 16-entry LSQ, 90 physical integer and 90 physical floating-point registers, 60-entry ROB • L1I cache: 1-cycle hit latency, 4-way, 64KB • L1D cache: 2-cycle hit latency, 4-way, 64KB
L2 Cache	• For the big.LITTLE and big.VLITTLE systems: 4-way, 4-bank, 8-cycle hit latency, 256KB for each big and little core cluster • For the decoupled vector system: 8-way, 8-bank, 8-cycle hit latency, 512KB shared by both the big core and the vector engine
LLC	Shared, 16-way, 12-cycle hit latency, 2MB
Main Memory	DDR4-2400

Table 4.3: Simulator Configuration

4.4.1 Simulated Systems

We use gem5 [BBB+11, LPAA+20, TCB18], a cycle-approximate simulator, to evaluate the performance of different hardware systems studied in this work. We use gem5's out-of-order pro-

1bIV	**One big core with an integrated vector unit** • 128-bit-long vector unit capable of issuing instructions out of order • Two vector arithmetic execution pipelines capable of executing integer and floating point vector instructions • The big core's load/store unit capable of handling 128-bit wide unit-stride memory requests
1b-4L	**Conventional big.LITTLE system** • One big core and a cluster of four little cores • Private L1I and L1D cache per core • Private L2 cache for each core cluster
1bIV-4L	**big.LITTLE system with an integrated vector unit** • One big core and a cluster of four little cores with L1 and L2 caches similar to *1b-4L* • The big core including the same integrated vector unit as in *1bIV*
1bDV	**Long-vector system with a decoupled vector engine** • One big core with an aggressive decoupled vector engine • 2048-bit-long vector engine with four vector element groups, a 8KB vector register file with eight read and four write ports, 64-entry command queue, 16-entry load queue, 16-entry store queue, and 64-entry data buffers • The vector engine connected directly to L2 cache
1b-4VL	**big.VLITTLE system** • One big core and a VLITTLE engine of four little cores with L1 and L2 caches similar to *1b-4L* • 64-entry micro-operation queue and 16-entry scalar data queue in the VLITTLE's VCU • 512-bit hardware vector length with two element groups, 64 entries in load data queues, and 32 entries in store data queues in VMU

Table 4.4: Evaluated Systems

Name	Input	1L		1b-4L		1b-4VL vs
		DIns	Cycles	DIns	DTsk	1bDV
bc	rMat_1M	332M	1.7B	800M	0.5M	1.3x
bf	rMat_1M	782M	4.7B	1.4B	0.8M	2.3x
bfs	rMat_1M	56M	0.3B	203M	0.2M	1.0x
bfsbv	rMat_1M	113M	0.3B	241M	0.2M	1.7x
cc	rMat_1M	480M	1.9B	1.0B	0.4M	1.7x
mis	rMat_1M	337M	1.8B	864M	0.2M	1.1x
tc	rMat_1M	748M	1.7B	1.0B	0.1M	2.4x
prd	rMat_1M	3.9B	23.9B	5.9B	2.8M	3.2x
geomean						1.7x

Table 4.5: Task-Parallel Applications – All task-parallel applications are taken from Ligra suite [SB13a]. DIns = dynamic instruction count in billions; DTsk = dynamic task count in millions; Cycles = cycle count in billions.

			1L		1bDV																1bIV-4L			1b-4VL vs	
Name	**Suite**	**Input**	**DIns**	**Cyc**	**DIns**	**VIns**	**ctrl**	**ialu**	**imul**	**fpu**	**xe**	**us**	**st**	**idx**	**prd**	**DOp**	**VOp**	**VPar**	**WInf**	**ArInt**	**DIns**	**VIns**	**DTsk**	**1bIV**	**1bIV-4L**
vvadd	k	8.4M	75.5M	210M	1.6M	42%	20	20	0	0	0	60	0	0	0	35.1M	97%	22.3	0.47	0.33	52M	10%	4.3K	4.0x	2.6x
saxpy	k	8.4M	75.5M	291M	1.6M	50%	17	0	0	33	0	50	0	0	0	42.9M	98%	27.2	0.57	0.67	50M	16%	4.3K	2.9x	2.2x
mmult	k	512x512	1.08B	3.18B	18.9M	44%	25	25	25	0	0	25	0	0	0	416M	97%	22.0	0.39	2.00	567M	18%	23.4K	2.7x	2.4x
k-means	ro	10Kx34	4.65B	13.6B	67.9M	46%	1	13	∼0	57	∼0	∼0	21	7	1	2.01B	98%	29.6	0.43	2.41	2.8B	25%	11.7K	1.4x	1.2x
pathfinder	ro	5Mx10	1.08B	2.92B	22.5M	50%	31	37	0	0	0	31	0	0	25	510M	98%	22.7	0.47	1.20	764M	16%	36.9K	4.0x	3.0x
jacobi-2d	rv	2Kx10	1.59B	9.64B	35.4M	44%	8	17	0	42	17	17	∼0	0	0	942M	98%	26.6	0.59	4.50	1.1B	16%	640	2.8x	2.4x
blackscholes	rv	2.5M	726M	2.75B	10.6M	90%	4	13	0	72	0	3	8	0	5	567M	100%	53.5	0.78	7.48	467M	26%	2.0K	1.8x	1.1x
lavamd	rv	4x4x4	1.05B	3.58B	25.0M	79%	7	11	0	57	0	5	20	0	5	966M	99%	38.7	0.92	2.72	680M	29%	64	2.0x	1.2x
backprop	ro	524K	1.17B	16.4B	21.5M	39%	13	∼0	0	44	0	12	31	0	0	484M	97%	22.5	0.41	1.00	1.2B	11%	130	1.5x	1.5x
particlefilter	rv	6K	1.24B	3.52B	43.1M	51%	1	72	1	26	0	∼0	∼0	∼0	∼0	1.41B	99%	32.8	1.14	200.9	906M	16%	3.9K	2.8x	1.1x
sw	ge	2048	1.34B	2.09B	132.7M	4%	10	58	2	0	10	10	12	0	9	404M	69%	3.0	0.30	3.20	932M	6%	1.0K	1.7x	1.8x
geomean																								2.1x	1.6x

Table 4.6: Data-Parallel Applications – 1L = one little core; 1bIV, 1bIV-4L, 1bDV and 1b-4VL = see Table 4.4; ro = Rodinia; rv = RiVEC; ge = Genomics; DIns = dynamic instruction count in regions of interest; Cyc = cycle count; VIns = percent of dynamic instructions that are of vector type; ctrl = vector control instructions; ialu = vector integer ALU instructions; imul = vector integer multiplication and division instructions; fpu = vector floating-point instructions; xe = vector cross-element instructions; us = unit-stride memory instructions; st = constant-stride instructions; idx = indexed memory instructions; prd = predicated instructions; DOp = total number of performed operations (i.e., scalar instructions + vector instructions × active vector length); VOp = percent of operations performed by a vector unit; VPar = logical parallelism (i.e., the total number of performed operations divided by the total dynamic instructions in vectorized code); WInf = work inflation (i.e., the total number of performed operations divided by the total dynamic instructions in scalar code); ArInt = arithmetic intensity (i.e., arithmetic operations / memory operations); DTsk = dynamic number of parallel tasks; geomean = calculated for Rodinia, RiVec, and genomics applications.

cessor model for our simulated big core and our in-house model to simulate in-order single-issue little cores. Our simulated cache subsystem is based on an Arm AMBA 5 CHI cache-coherent model provided in gem5 [gem21], and we use its simple network model for our simulated on-chip network. We model one-cycle address translation overhead per memory access (i.e., assuming memory accesses always hit in level-one TLBs) for all evaluated designs. For performance evaluations in Section 4.5, we keep the big core, little cores, and caches running at the same frequency (i.e., 1GHz) to isolate micro-architecture-level behaviors of all designs from potential performance impacts of voltage/frequency scaling. In Section 4.7, we then explore a performance/power design space when considering voltage/frequency scaling of big and little core clusters. Table 4.3 shows the details of our simulated processors and memory subsystem.

Table 4.4 shows a list of evaluated systems and their configurations. *1bIV* supports next-generation vector architectures by integrating a small vector unit into its big core. The integrated vector unit supports 128-bit hardware vector length that is similar to a typical SIMD width in common mobile SoCs (e.g., Samsung M3 [RBGZ19]) implementing traditional packed-SIMD architectures such as Arm NEON. This unit also leverages two of its existing execution pipelines in its associated big core for vector execution and shares the same data cache port with the big core to minimize area overheads. This unit exemplifies future modest integrated vector units implementing next-generation vector architectures [SBB+17]. *1b-4L* is a conventional big.LITTLE system including one big and four little cores without any vector execution support. *1bIV-4L* includes a big core with an integrated vector unit and a cluster of four little cores.

1bDV is a long-vector system with a decoupled vector engine connected to a big core, which is similar to aggressive vector machines such as Tarantula [EAE+02]. Figure 4.3 shows its vector engine's micro-architectural details. *1bDV* includes a large vector register file (i.e., 2048-bit vector length), wide multi-lane execution pipelines (e.g., 16 arithmetic operations can be processed in parallel on 32-bit vector elements), a high-bandwidth connection to an L2 cache that can support more requests in parallel than L1 caches, and deep command and data buffers to aggressively decouple memory accesses from vector computation. Those significant resources enable best-in-class performance for data-parallel workloads at the cost of extra silicon area.

1b-4VL is a big.VLITTLE system that has an equivalent area compared to *1bIV-4L*. To ensure no cycle time penalty to the existing little cores, we conservatively model fully pipelined communication paths between multiple vector-specific components and the cores. For example, it takes

a full cycle to broadcast a command from the VCU to all little cores, to send a request from the VMIU to a VMSU, to send data from a core to the VSU and the VXU, and to forward a data response from the VLU to a core. We added a fixed penalty of 500 cycles to the beginning of each vector region to account for switching overheads (e.g., saving thread contexts and flusing little core pipelines).

4.4.2 Application Benchmarks

We evaluate the systems using eight task-parallel applications from Ligra benchmark suite [SB13a] and eight data-parallel applications from Rodinia suite [CBM+09], RiVec suite [RHP+20], and a genomics benchmark suite. We also study three data-parallel kernels to further understand the performance of the simulated systems. *vvadd* and *mmult* do vector addition and matrix multiplication respectively. *saxpy* performs a single-precision $A \times X + Y$ on two vectors. Table 4.6 and Table 4.5 summarize these applications and kernels. The set of studied applications and kernels represent real-world workloads running in mobile SoCs such as smartphones, drones, and AR/VR systems. For example, *backprop* performs a forward classification on fully connected layers, and *kmeans* clusters items into similar groups. Both algorithms are used in machine learning mobile applications. *particlefilter* is an image processing algorithm tracking an object in each frame of an input video, which can be used to do image processing in smartphones and AR/VR headsets. *blackscholes* and *jacobi2d* are data analytics applications that represent big-data processing workloads such as natural language processing. *sw* (i.e., Smith-Waterman) implements a local genome sequence alignment algorithm that finds regions of similarity between reference and query DNA sequences. Graph analytics are important to perform fast on-device analysis of large datasets in mobile devices without relying on the cloud.

For data-parallel applications and kernels, we manually vectorize them using RISC-V RVV vector intrinsics supported in LLVM 13. We parallelize task-parallel applications using a task runtime system (i.e., similar to Intel TBB [Rei07] and Cilk Plus [int13]) implementing a random work-stealing algorithm that helps distribute tasks dynamically and evenly across heterogeneous cores. Since the *1bIV-4L* system can support both vector execution on its big core and scalar tasks on its little cores, we implement both scalar and vectorized versions of each data-parallel application. The task-parallel runtime system dynamically chooses which version of a task to run depending on which core executes the task.

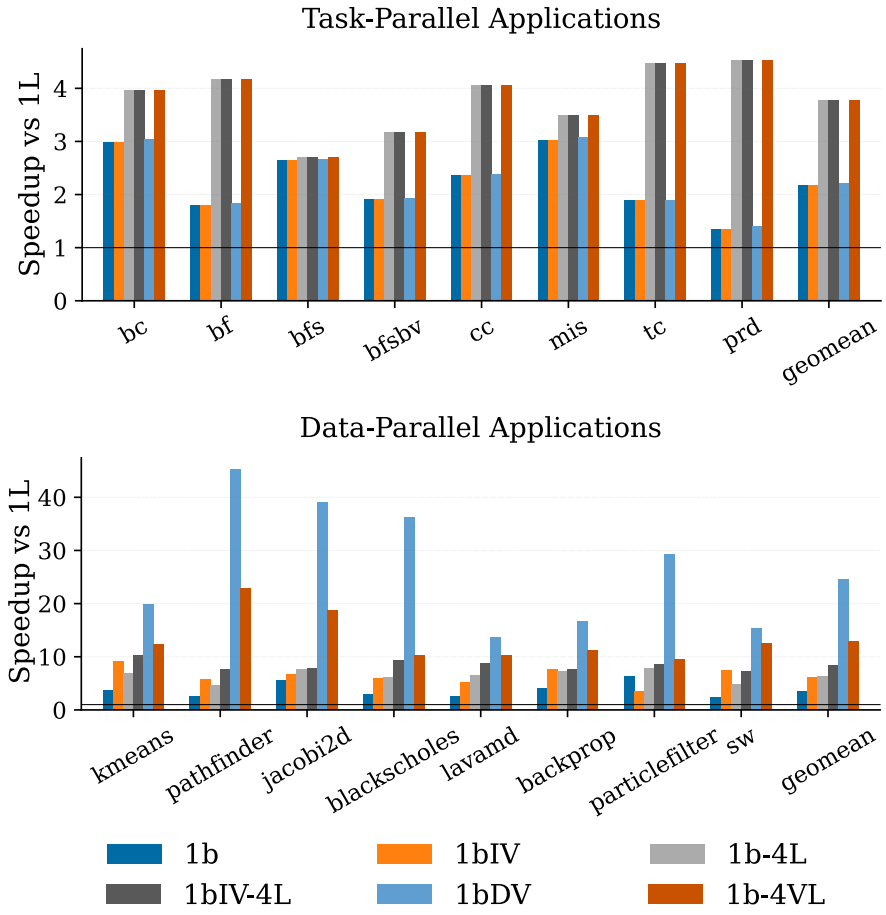


Figure 4.4: Speedup over 1L – *1L* = one little core; *1b* = one big core; *1bIV* = one big core with an integrated vector unit; *1b-4L* = one big & four little cores; *1bIV-4L* = one big core with an integrated vector unit & four little cores; *1bDV* = one big core with a decoupled vector engine; *1b-4VL* = big.VLITTLE system with one big and a VLITTLE engine of four little cores.

4.5 Performance Evaluation

In this section, we describe our cycle-level performance results comparing the *1b-4VL* system to the *1bIV-4L* and *1b-DV* baseline systems for both task-parallel and data-parallel workloads. We then analyze performance impacts of multiple vector element groups, packed vector element support, and reconfigurable cache subsystem on the *1b-4VL* system's performance.

4.5.1 Overall Performance

Figure 4.4 shows the overall performance of all simulated systems normalized to *1L* for both sets of task-parallel and data-parallel applications.

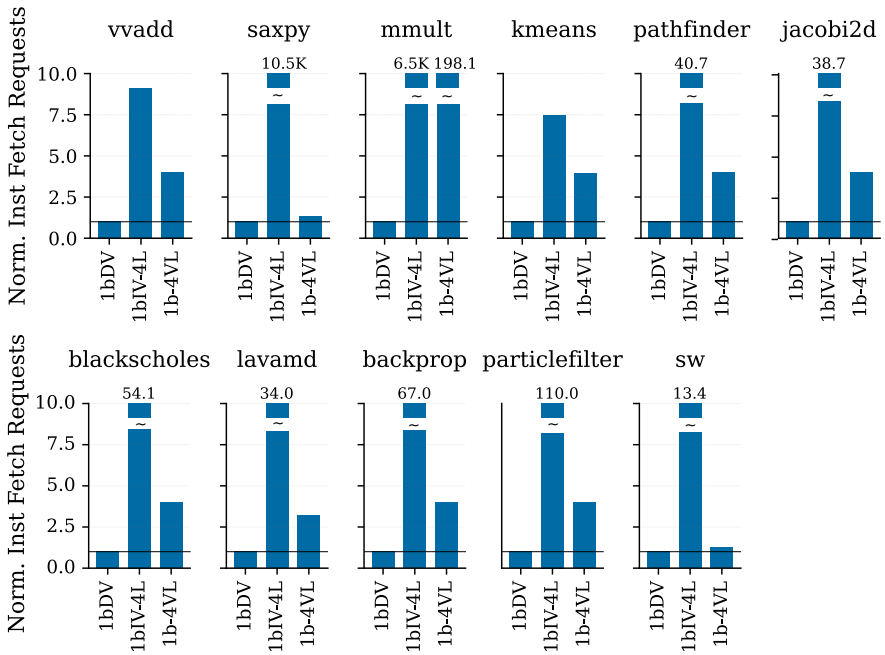


Figure 4.5: Number of Instruction Fetch Requests to Memory – All numbers are normalized to *1bDV*.

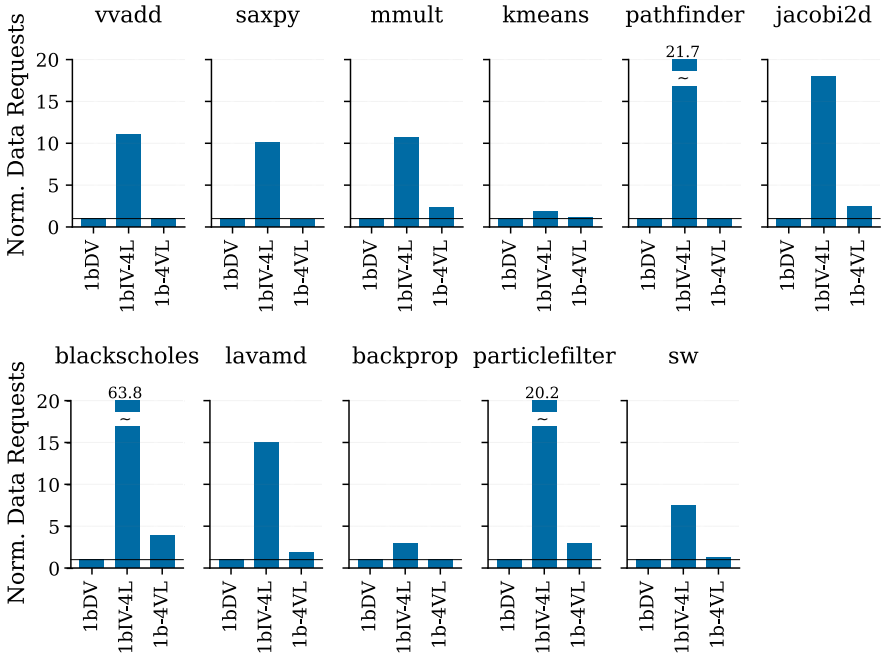


Figure 4.6: Normalized Number of Data Requests to Memory – All numbers are normalized to *1bDV*.

Task-parallel applications – *1bIV-4L* and *1b-4VL* perform the same since they both execute the same scalar code without using their integrated vector unit and VLITTLE vector engine respectively. In scalar mode, *1b-4VL* simply bypasses all additional vector-specific components, which incurs no performance overheads. *1bIV-4L* and *1b-4VL* are able to achieve 1.7× speedup over the *1bDV* system since the *1bDV* system is able to use only its big core to execute scalar code. Since not all workloads can be efficiently vectorized (e.g., irregular graph applications) and task-parallel applications remain an important set of workloads in mobile SoCs, it is hard to justify a large decoupled vector engine in small mobile SoCs to accelerate only data-parallel applications. Both *1bIV-4L* and *1b-4VL* are more efficient than *1bDV* in using their computing resources with the help of the work-stealing runtime system that dynamically distributes tasks to available cores.

Data-parallel applications – The *1b-4VL* system performs 1.6× faster than *1bIV-4L* while being able to achieve roughly half of *1bDV*'s performance. The *1bDV* system supports 2048-bit hardware vector length that is significantly larger than the 128-bit vector length of the integrated vector unit in *1bIV-4L* and the 512-bit vector length of *1b-4VL*. The larger a hardware vector length is, the better a system can amortize its front-end instruction overheads by performing more computation per vector instruction. Figure 4.5 shows that across all vectorized kernels and applications, *1bDV* and *1b-4VL* perform significantly fewer instruction fetch requests than the *1bIV-4L* system does. In addition, the four little cores in *1bIV-4L* independently execute tasks, which results in duplicated instruction fetches among the four little cores and runtime overheads to dynamically distribute tasks across the system.

The *1bDV* supports higher compute throughput using its wide execution pipelines that are capable of performing up to 16 arithmetic operations on 32-bit data elements in parallel. In contrast, *1bIV-4L*'s integrated vector unit is able to perform four operations on 32-bit data elements per cycle, and its four little cores can issue four scalar instructions in total per cycle. Meanwhile, *1b-4VL* can perform eight simple integer arithmetic and multiplication operations, and four complex integer and floating-point operations per cycle on 32-bit vector elements. Moreover, *1b-4VL* and *1bDV* support respectively two and four element groups that can effectively hide the latency of complex instructions (e.g., multiplication and division) in compute-intensive workloads such as *mmult*, *blackscholes*, *jacobi-2d*, and *lavamd* (see Table 4.6).

1b-4VL and *1bDV* systems are also able to fetch data more efficiently from memory than *1bIV-4L* does. Figure 4.6 shows the normalized number of data memory requests in the three

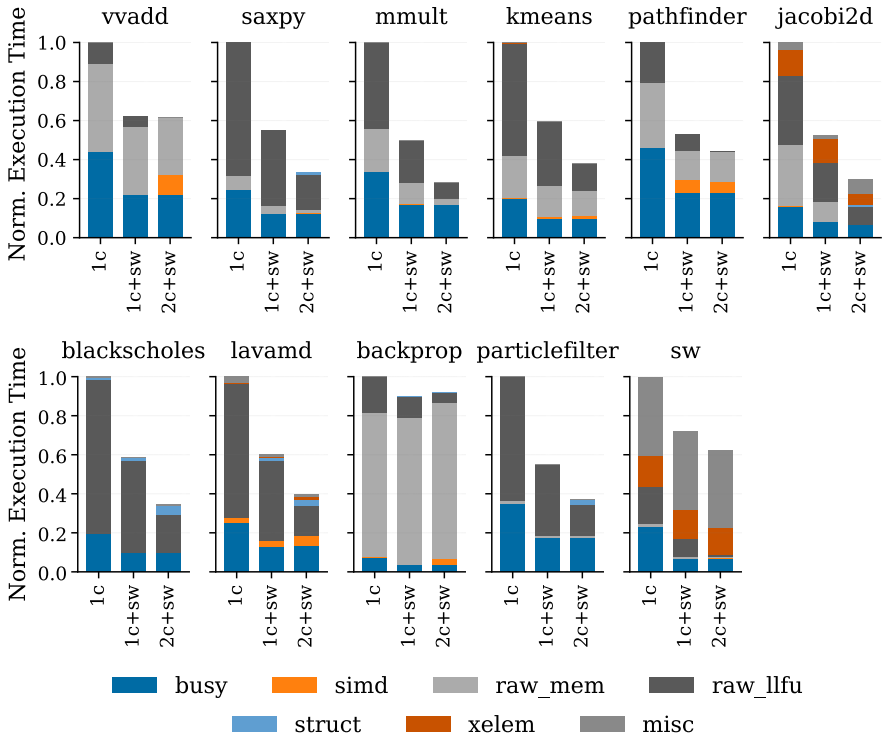


Figure 4.7: Average Execution Time Breakdown of Four Little Cores in *1b-4VL* – *1c* = *1b-4VL* with one chime (vector element group); *1c+sw* = *1b-4VL* with one chime & packed vector elements; *2c+sw* = *1b-4VL* with two chimes & packed vector elements; *busy* = cycles in which little cores are not stalled; *simd* = stalled cycles due to lock-step issuing of micro-ops in the VCU; *raw_mem* = stalled cycles due to waiting for memory; *raw_llfu* = stalled cycles due to little cores waiting for long-latency micro-ops to complete; *struct* = stalled cycles due to structural hazards; *xelem* = stalled cycles due to cross-element micro-ops; *misc* = other stalled cycles (e.g., no micro-op from the VCU).

systems. For workloads with regular memory access patterns (i.e., using unit-stride and constant-stride memory instructions) such as *vvadd*, *saxpy*, *pathfinder*, and *lavamd*, *1b-4VL* and *1bDV* can efficiently fetch multiple per-element pieces of data using a single wide memory request. In contrast, the integrated vector unit's limited hardware vector length, the scalar execution of four little cores, and runtime overheads in the *1bIV-4L* system require significantly more memory requests compared to both *1b-4VL* and *1bDV*.

4.5.2 Reconfigurable Compute Pipeline

To evaluate performance impacts of packed-vector-element support and multiple vector element groups on the performance of *1b-4VL*, we study three configurations: (1) *1c* - one element group and no packed element support, (2) *1c+sw* - one element group with packed element sup-

port, and (3) *2c+sw* - two element groups with packed element support. Figure 4.7 shows their execution time breakdown.

Since all studied data-parallel applications use 32-bit data types, enabling packed-vector-element support effectively doubles the *1b-4VL*'s hardware vector length and increases its compute throughput. This reduces the number of executed instructions, which results in less dependency stalls (e.g., *raw_mem* and *raw_llfu* cycles in *saxpy*, *pathfinder*, and *lavamd*). In addition, the utilization of execution pipelines in little cores is increased since more per-element arithmetic operations can be performed in the same cycle (e.g., integer addition) and back to back in consecutive cycles (e.g., floating-point multiplication).

The *2c+sw* configuration introduces a second element group to *1b-4VL* compared to the *1c+sw* configuration. The second element group helps hide the long latency of complex instructions such as floating-point multiplication by overlapping the execution of the first and second element group, which reduces further read-after-write dependency stalled cycles in compute-intensive applications such as *blackscholes*, *particlefilter*, and *lavamd*. Some of memory latency can be hidden as well in memory-intensive workloads such as *saxpy*, *jacobi-2d*, and *pathfinder* since more memory requests from multiple element groups can be in flight at the same time. In some cases such as *vvadd* and *backprop*, adding the second element group slightly increases *simd* stalled cycles. The little cores run out of sync due to more memory requests conflicting for resources (e.g., accessing the same L1D bank) in the cache subsystem, which eventually stalls the VCU from issuing micro-operations in lock step to all little cores.

4.5.3 Performance Impacts of Data Buffering

We evaluate performance impacts of data buffering in the *1b-4VL* system by varying the size of its VMU's load and store data queues. Figure 4.8 shows that by increasing the amount of data buffering in the VMU, the performance of memory-intensive workloads such as *vvadd*, *saxpy*, *pathfinder*, and *backprop* can be improved significantly. Supporting larger load and store data queues allows more in-flight memory requests to fully take advantage of the available bandwidth provided by the logically shared multi-bank L1 data cache in the VLITTLE engine. This enables more decoupling of memory accesses and arithmetic computation, which can effectively hide long memory latency in memory-sensitive workloads. However, deep data buffering comes at significant area cost. Our technique to re-purpose SRAM data arrays in L1 instruction caches, which

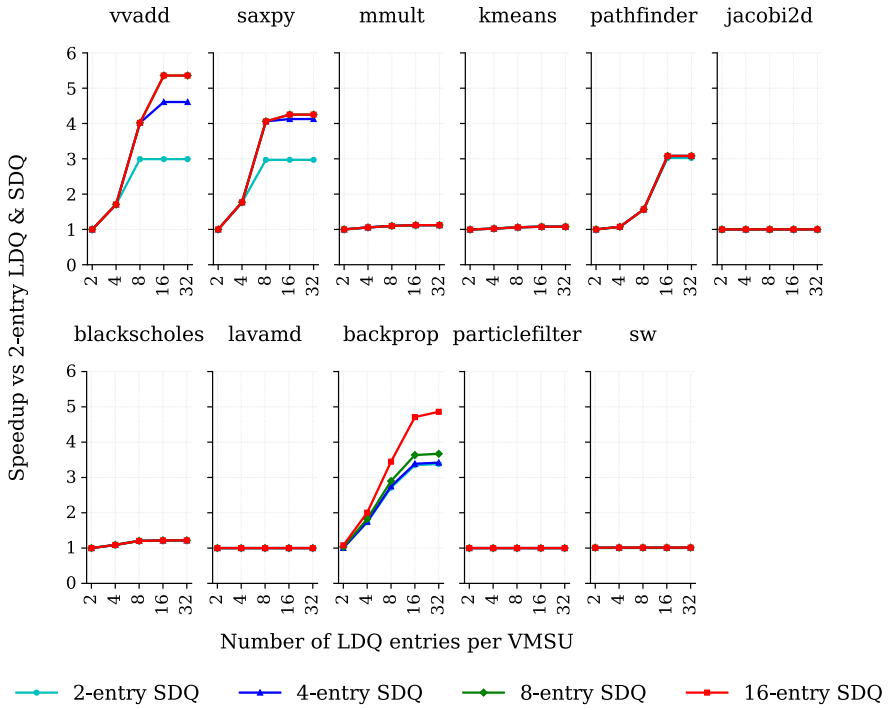


Figure 4.8: Performance Impacts of VMU's Load Data Queue (LDQ) and Store Data Queue (SDQ)

are otherwise unused in the vector mode, as data buffers for load and store requests provides an area-efficient way to unlock more memory-level parallelism and memory-computation decoupling without adding non-trivial area overheads.

4.6 Area Evaluation

In this section, we first evaluate area overheads of additional vector-specific components in the *1b-4VL* system using a post-systhesis area model. We then estimate the area of the *1bDV* system by referencing an open-source RISC-V decoupled vector machine.

Methodology – We use a post-synthesis component-level area modeling methodology to evaluate area overheads of extra hardware added to support a VLITTLE engine composed of four little cores. We implement key components of the VLITTLE engine in RTL. We use two different RTL models for the little core: simple and Ariane [ZB19]. The simple core is our in-house single-issue in-order processor implementing RISC-V RV64IMAF. The Ariane model is an open-source Linux-capable RISC-V RV64G in-order core. For L1 instruction and data caches, we use a 32KB two-way set-associative cache model that is configured to support either 64-bit or 512-bit data

Component	Area	Simple		Ariane	
	(k µm^2)	4L	4VL	4L	4VL
Simple core	26.1	×4	×4		
Ariane core	41.8			×4	×4
32KB L1I with 64b data path	40.3	×4	×4	×4	×4
32KB L1D with 64b data path	40.3	×4		×4	
32KB L1D with 512b data path	41.6		×4		×4
VXU: Ring network	0.3		×1		×1
VMU:	2.9		×1		×1
• Micro-op & command queues	1.7				
• Store-address CAM	0.8				
• Line buffers	0.4				
VCU:	2.0		×1		×1
• Micro-op queue	1.0				
• Data queue	1.0				
Total		427.0	437.4	489.8	500.1
4VL vs. 4L overhead			**2.4%**		**2.1%**

Table 4.7: Post-Synthesis Area Results – 4L = a cluster of four little cores with L1I and L1D caches; 4VL = a VLITTLE engine with four little cores, L1I, and L1D caches.

path. For the VCU and VMU, we model multiple micro-operation, scalar data, command queues, and store address CAM according to the configuration of the *1b-4VL* system shown in Table 4.4. For the VXU, we implement a unidirectional 64-bit-wide ring network. We then use a commercial standard-cell-based toolflow in a 12-nm technology node to generate post-synthesis area results.

Area overheads of big.VLITTLE – Table 4.7 shows the detailed area comparison between a cluster of four little cores and an equivalent VLITTLE engine. The 4VL engine only adds around 2% area overhead (i.e., 2.4% if using the simple little cores and 2.1% if using Ariane little cores) compared to the 4L cluster including their private L1 data and instruction caches. The main area overheads come from the VCU and VMU that includes multiple FIFO queues for micro-operations, scalar data, and VMU commands. For a complete big.LITTLE system including a big core, its private L1 and L2 caches, and interconnect network, we expect the area overhead of big.VLITTLE architectures to be less than 1% of the entire system.

First-order area estimate of 1bDV – We reference Ara [CSZ$^+$20], an open-source decoupled vector machine, to estimate the area of our simulated decoupled vector engine. We use an Ara configuration that includes eight 64-bit compute lanes that are equivalent to the 16x 32-bit lanes in our simulated decoupled vector engine in the *1bDV*, which makes the areas of the two vector

engines comparable. The work reported that the Ara configuration has an area of around 6,000 kilo-gates (kGE) (i.e., 738 kGE per lane) and that an Ariane core without its L1 caches has an area of 524 kGE. Table 4.7 shows that one L1 32KB cache's area is roughly the same as one Ariane core's area without caches. Therefore, a cluster of four Ariane cores with their L1 instruction and data caches is as large as an eight-64-bit-lane Ara vector engine (i.e., roughly 6,000 kGE). Since our VLITTLE cluster incurs less than 3% of area overhead compared to a cluster of four Ariane cores with their L1 caches, a four-core VLITTLE cluster's area is comparable to the simulated decoupled vector engine used in *1bDV*. More detailed area analysis of the *1bDV* system is left for future work.

4.7 Power & Energy Evaluation

In this section, we first qualitatively evaluate power and energy efficiency of big.VLITTLE architectures. We then explore the potential of voltage/frequency scaling to further improve the performance and power efficiency of big.VLITTLE architectures for data-parallel workloads.

4.7.1 Qualitative Power and Energy Efficiency Analysis

In terms of power, a big.VLITTLE system leverages existing little core pipelines (i.e., functional units and register files) for vector execution and the big core for scalar control flow. Extra vector-specific components mainly consist of small FIFO command/data buffers and control logics, and they can be power-gated in the scalar mode to avoid leakage power. In the vector mode, front-end components (e.g., fetch, decode stages, and branch predictor) in little cores and control logic in instruction caches are not used, so they do not contribute to the overall dynamic power consumption. Therefore, we do not anticipate a big.VLITTLE system to draw significantly more power than an equivalent big.LITTLE system.

Regarding energy efficiency, by reconfiguring little cores as a medium-sized decoupled vector engine, big.VLITTLE architectures can reduce significantly the number of instruction and data memory accesses due to less dynamic instructions (Figure 4.5) and wider data memory requests (Figure 4.6). This reduction translates directly to less energy consumed in the memory subsystem for data-parallel workloads compared to an equivalent big.LITTLE system with an integrated

Big core			Little core		
	Frequency (GHz)	**Avg Power** (W)		**Frequency** (GHz)	**Avg Power** (W)
b0	0.8	0.432	l0	0.6	0.043
b1	1.0	0.591	l1	0.8	0.059
b2	1.2	0.841	l2	1.0	0.095
b3	1.4	1.205	l3	1.2	1.450

Table 4.8: Average Power Consumption of a Big and Little Core at Multiple Voltage/Frequency Levels – The average power consumption of a big and little core at different voltage/frequency levels was reported in previous work [VSP+17]. The work used an Odroid XU+E board that includes a Samsung Exynos 5410 SoC and per-cluster voltage/current sensors for the measurement. This SoC consists of four big Arm Cortex-A7 cores and four little Arm Cortex-A15 cores. The power consumption was measured by running 26 benchmarks on all cores at different frequencies (i.e., 500-1200MHz for the little cores and 800-1500MHz for the big cores at corresponding appropriate voltage levels).

vector unit. In addition, higher performance at a similar power consumption yields higher energy efficiency. Previous work [LSFJ06, LAB+11] has also studied and reported the energy efficiency of vector architectures. Future work can explore a more detailed power/energy evaluation of big.VLITTLE.

4.7.2 Voltage/Frequency Scaling Design Space Exploration

Methodology – We assume the voltage/frequency of the big and little core clusters can be scaled independently, which is similar to typical commercial big.LITTLE systems (e.g., Samsung Exynos [KKCL13] and Qualcomm Snapdragon [Gwe14a]). We use previously reported average power consumption of a big and little core at different voltage/frequency levels [VSP+17] to estimate the average power consumption of the big and little core clusters in our simulated big.LITTLE and big.VLITTLE systems. Table 4.8 shows the selected voltage/frequency levels for big and little core clusters and their corresponding average power consumption as reported in the previous work. In this design space exploration study, we assume that both *1bIV-4L* and *1b-4VL* have similar average power consumption compared to *1b-4L*. To estimate the power consumption of *1bDV*, we reference the decoupled vector engine in Tarantula [EAE+02]. The work reported its vector engine consumed roughly 40% more power than its out-of-order super-scalar core. Both the vector engine and the out-of-order core were clocked at the same frequency, which is similar to our simulated *1bDV* system. We assume the same power consumption ratio between the big core and

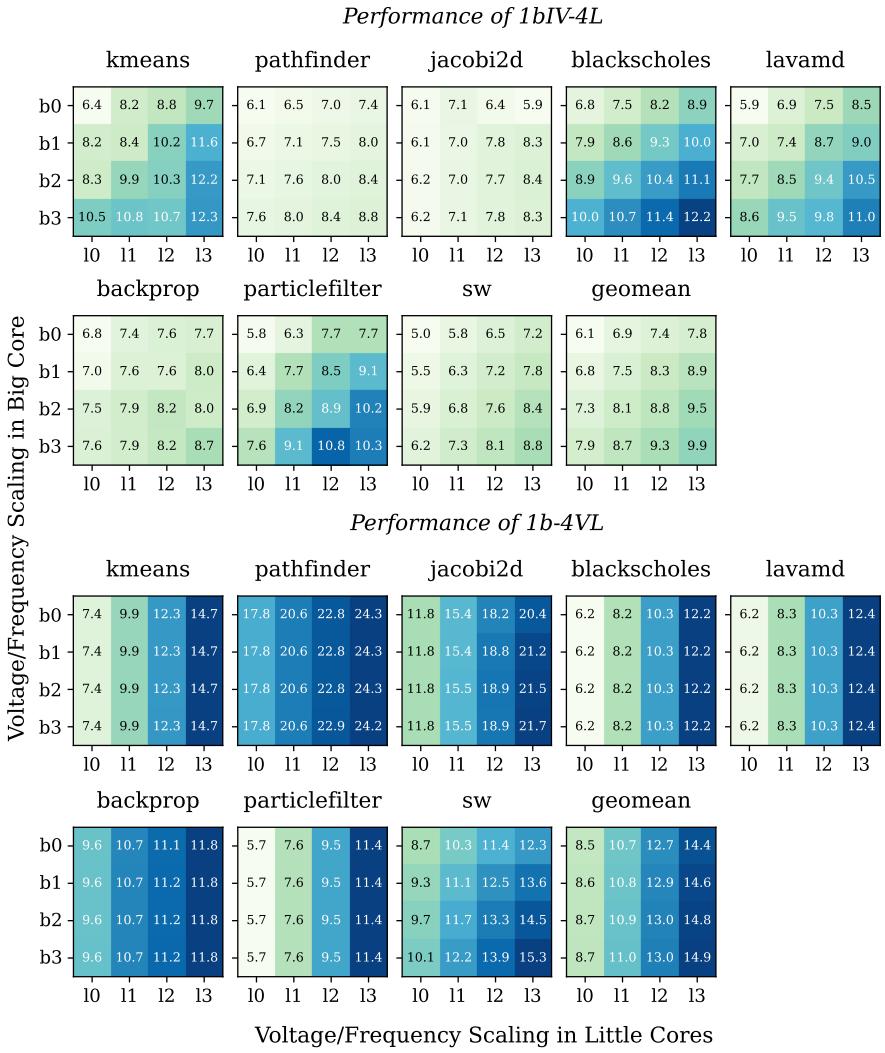


Figure 4.9: Performance of 1bIV-4L and 1b-4VL at Different Voltage/Frequency Scaling Levels for Big and Little Cores – All studied voltage/frequency levels (i.e., $\{b0, b1, b2, b3\}$ and $\{l0, l1, l2, l3\}$) are shown in Table 4.8. The performance numbers show speedup over *1L* system running at 1GHz. The color scaling for each application is the same for both 1bIV-4L and 1b-4VL.

its decoupled vector engine at different voltage/frequency levels. More accurate power models for all designs are left for future work.

Performance impacts of voltage/frequency scaling – Figure 4.9 show the performance of *1bIV-4L* and *1b-4VL* at different combinations of voltage/frequency levels for big and little core clusters. For *1bIV-4L*, whether to boost the big core or the little core cluster for higher performance depends on specific workloads and existing voltage/frequency levels. For example, in *blackscholes*, boosting the big core cluster (e.g., from $(b1, l1)$ to $(b2, l1)$) always yields better performance than

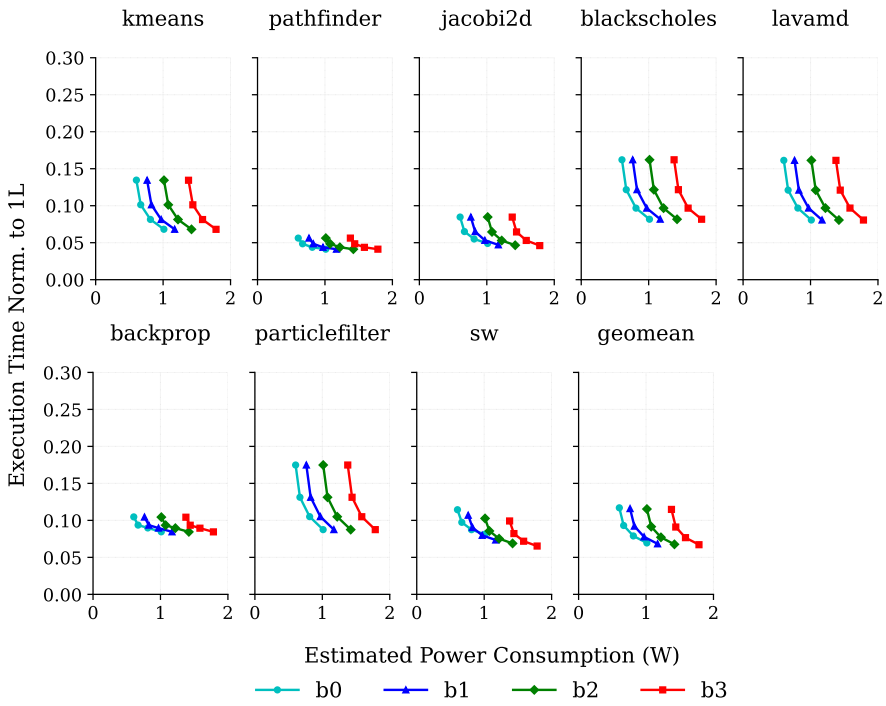


Figure 4.10: Execution Time and Estimated Power Consumption of *1b-4VL* at Different Voltage/Frequency Levels – Each performance-power data point corresponds to a combination of big and little core's voltage/frequency levels shown in Table 4.8.

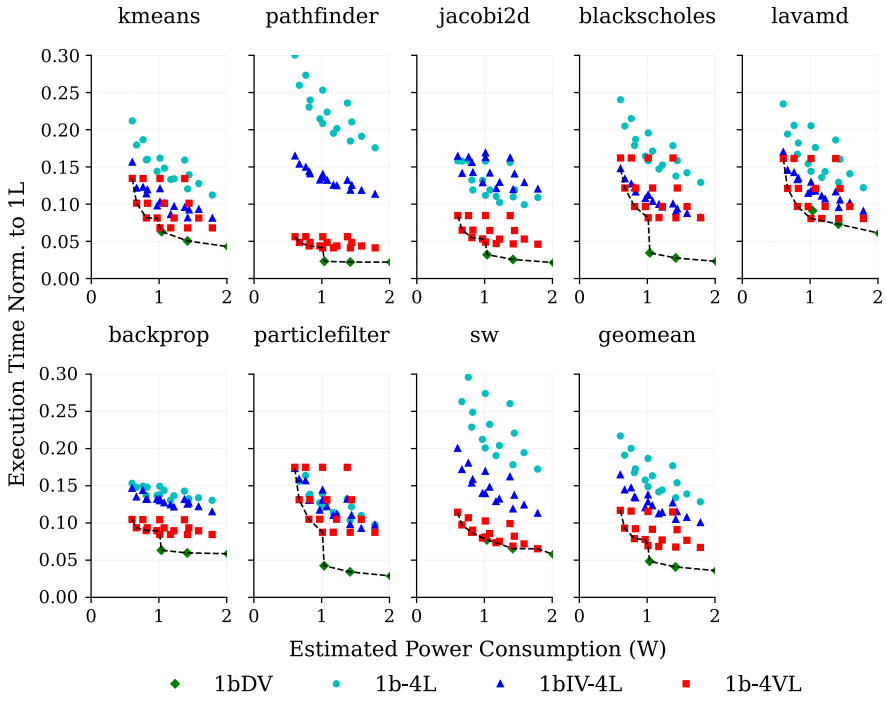


Figure 4.11: Execution Time and Estimated Power Consumption of Multiple Designs at Different Frequencies – The dotted lines show Pareto frontier curves. Each performance-power data point corresponds to a combination of big and little core's voltage/frequency levels shown in Table 4.8.

boosting the little core cluster (e.g., from $(b1, l1)$ to $(b1, l2)$). In contrast, in *sw*, boosting the little core cluster is more beneficial than boosting the big core cluster.

For *1b-4VL*, boosting the big core cluster while keeping the voltage/frequency level of the little core cluster yields insignificant performance benefits across all applications except *sw*. Different from *1bIV-4L* in which both the big and little cores work together on the main computation, in *1b-4VL*, the big core mainly works as a control core for the VLITTLE engine that handles all heavy vector computation. Slowing down the big core to a certain limit does not cause the VLITTLE engine to stall since the engine has a deep command buffer and long vector length. For *sw*, since only 69% of the work is vectorized (see *VOp* in Table 4.6) and the rest is executed by the big core, boosting the big core while keeping the little cores running at the same voltage/frequency level helps increase the overall performance.

Performance and power consumption trade-offs – Figure 4.10 shows the performance and estimated average power consumption of all studied voltage/frequency combinations for *1b-4VL*. Boosting the little core cluster and slowing down the big core help *1b-4VL* achieve the Pareto optimal performance/power curve. Given a certain power budget, the power saved by lowering down the big core's speed can be used to boost the little cores that execute most of the vector computation in data-parallel workloads. This power trading translates to higher performance and efficiency for *1b-4VL*.

Figure 4.11 shows performance/power data points for all designs including *1bDV*, *1b-4L*, *1bIV-4L*, and *1b-4VL*. In the low-power region (i.e., less than 1W), *1b-4VL* stays on the Pareto optimal performance/power curve across all data-parallel applications. For *1bDV*, despite its ability to deliver high performance for data-parallel applications, its power-hungry decoupled vector engine makes it not feasible in the low-power region. In the high-power region (i.e., more than 1W), *1b-4VL* is able to get close to the performance/power efficiency of *1bDV*. It is important to note that unlike *1bDV*, *1b-4VL* does not sacrifice the performance of important task-parallel applications to achieve this power/performance efficiency for data-parallel workloads (see Section 4.5).

4.8 Related Work

Rockcress [BAPS21] extends many-core architectures with vector-like execution support by dynamically grouping multiple small cores together into vector groups executing the same stream

of instructions. Different from big.VLITTLE, Rockcress targets scale-out many-core systems with scratchpads and mesh-based tiled network by loosely coupling multiple cores in a vector group together, which requires frequent intra-vector-group synchronizations, a nontrivial amount of data buffering, and a dedicated instruction-forwarding network in each core's scratchpad as the group size grows. In contrast, big.VLITTLE architectures aim to provide vector execution in small mobile systems, which allows a small number of OS-capable little cores in a VLITTLE engine to execute strictly in lock step, which greatly simplifies its design and implementation. Rockcress adopts a non-standard vector-thread abstraction [KBH+04a] and requires extensive compiler-level support to insert implicit instruction barriers so that its scalar cores do not run out of resources in vector mode. In contrast, big.VLITTLE architectures support next-generation vector architectures and compilers out of the box.

Vector-thread architectures [KBH+04a, KBH+04b, KBA08, BAA08, LAB+11] enable a SIMD-like micro-architecture to execute MIMD code. They propose a non-standard hybrid vector/thread ISA abstraction that would require non-trivial programming model and compiler support. Vector-thread architectures strive to achieve a single abstraction for both task- and data-parallel workloads with certain trade-offs in performance, programmability, and energy efficiency. In contrast, big.VLITTLE provides both multi-thread and vector solutions in a single micro-architecture to provide the best multi-thread support when running task-parallel workloads and the most efficient vector support when running data-parallel applications.

Cray X1 [DVWW05] provides options to group multiple vector engines into a single long-vector machine, which is more applicable to large-scale super-computing systems already equipped with vector engines than to small mobile SoCs. Taking an opposite approach compared to big.VLITTLE architectures, vector lane threading [RSOK06] reconfigures multiple lanes in a vector engine as individual scalar cores that can execute independently from each other. Similarly, Libra [PPPM12] attempts to overcome the inflexibility of SIMD accelerators by allowing different lanes to work in either SIMD or VLIW execution styles.

Some prior work has proposed to gang multiple scalar threads dynamically to amortize their front-end instruction overheads [KJT04,GCC+08,LFB+10,MBW14,KJT+17,KBH+04a,KBH+04b, KBA08]. While preserving the simplicity of multi-thread programming abstractions, those approaches spend extra energy at run time to dynamically align multiple streams of scalar execution. Some other reconfigurable architectures aim to exploit both thread-level and instruction-level par-

allelism such as CoreFusion [IKKM07], MorphCore [KSH+12], and others [GFAM10, KSG+07, TBS08, TCS20]. Unlike big.VLITTLE architectures, they do not explore data-level parallelism.

4.9 Conclusion

This chapter has demonstrated that big.VLITTLE architectures offer a compelling high-performance and area-efficient solution to accelerating data-parallel workloads in heterogeneous multi-core mobile systems. The reconfigurability of big.VLITTLE architectures resolves the fundamental tension between performance and area in implementing next-generation vector architectures, which opens up opportunities to provide the performance level of decoupled vector engines for data-parallel workloads in small mobile systems without sacrificing either valuable silicon area on chips or performance of task-parallel workloads. This work provides a small but important step toward a future era of efficient next-generation vector architecture support in mobile SoCs. Future research can explore the scalability of big.VLITTLE architectures beyond the scope of mobile SoCs.

CHAPTER 5
SparseZipper: EVOLUTIONARY SPECIALIZATION FOR MODERN MATRIX ARCHITECTURES

General matrix multiply (GEMM) is the key building block in many different domains including machine learning, graph analytics, and scientific computing. There have been many solutions in both general-purpose and domain-specific architectures accelerating GEMM with dense input matrices. However, matrices in workloads are not always dense. In fact, many traditional and emerging workload domains such as neural network models, graph analytics, and scientific simulations operate on sparse matrices where the majority of values are zeros. Sparse matrices are typically stored in compact formats with metadata indicating positions of non-zero values, which makes them incompatible with built-in dense matrix engines without first uncompressing the matrices. Previous work has proposed several ISA extensions on CPUs to accelerate sparse computations. In this work, rather than designing a completely new ISA extension for sparse computations on CPUs, we propose SparseZipper that enhances the existing matrix extensions specialized for dense GEMM to accelerate sparse GEMM (SpGEMM) through the evolutionary specialization approach. SparseZipper targets the key bottleneck, which is merging partial sparse vectors, streams of key-value pairs, in a conventional SpGEMM algorithm for data-parallel architectures. At the core of SparseZipper is its ability to efficiently merge such streams in parallel by leveraging in-place matrix registers to store parts of concurrent streams and built-in systolic array to merge those streams together. To facilitate that merge operation, we propose a minimal set of additional architectural states to keep track of active streams and matrix instructions to move streams between matrix registers and memory. Our performance evaluations using the cycle-level modeling methodology presented in Chapter 3 show SparseZipper achieves 5.98× and 2.61× speedup over a scalar hash-based implementation of SpGEMM and a vectorized SpGEMM version respectively.

5.1 Introduction

General matrix multiply (GEMM) is the key building block in many different domains including machine learning, graph analytics, and scientific computing. Therefore, countless solutions in both general-purpose and domain-specific architectures have been proposed to accelerate dense GEMM (i.e., most values in matrices are non-zeros) with various trade-offs in generality,

programmability, compute density, performance, and energy efficiency [JYP+17, Tie20, CYES19, JYK+20, CGG+21b]. CPU vendors have recently introduced matrix extensions such as Intel's Advanced Matrix Extension (AMX) [int23b, NMM+22, JQS+21], Arm's Scalable Matrix Extension (SME) [arm23], and IBM's Matrix-Multiply Assist (MMA) [ibm23] to their ISAs for dense GEMM acceleration. The RISC-V community has also put forward a proposal for a new matrix extension [ris23]. Compared to domain-specific processors (DSPs), those matrix extensions attempt to strike a balance between generality and specialization on CPUs that are still widely deployed for processing dense GEMM in edge devices [WBC+19b] and large-scale servers [HBB+18].

However, matrices in workloads are not always dense. In fact, many recent neural network models [RCK+20, NMS+19, HMD15, JYP+17, WBC+19b], real-world graph analytics [Dav19, HS11, ST15], and scientific simulations [CGM+96, Gal96] operate on sparse matrices where the majority of values are zeros. In addition, matrix densities (i.e., the percentage of non-zero values in a matrix) vary dramatically across domains (e.g., from 10^{-6}% density in matrices representing social graphs to 50% density in matrices used in neural network models [HAMP+19]). Such low levels of matrix density prevent us from computing GEMM for sparse matrices efficiently on CPUs using the recently introduced matrix extensions since most multiply operations are ineffectual (i.e., multiplying with zero). Moreover, sparse matrices are typically stored in compact formats (e.g., compressed sparse row (CSR)) with metadata indicating positions of non-zero values, which makes them incompatible with built-in dense matrix engines without first uncompressing the matrices.

Previous work has proposed several ISA extensions on CPUs to accelerate sparse computations. SparseCore [RCYQ22] is a stream-based ISA extension designed specifically for sparse computations, but it requires substantial architectural changes (e.g., stream registers and stream value processing units) without being able to support dense computations. VEGETA proposes to extend the recent matrix extensions to accelerate sparse/dense matrix multiplication (SpMM) in addition to dense computations [JDB+23]. However, VEGETA is limited to SpMM and DNN-specific sparsity structures, and it is not applicable to SpGEMM and general sparse matrices with vastly different density levels and structures.

In this work, rather than designing a completely new ISA extension for sparse computations on CPUs, we propose SparseZipper that enhances the existing matrix extensions to accelerate the general sparse matrix-matrix multiplication (SpGEMM) with arbitrary sparsity levels and structures. SparseZipper targets the key bottleneck, which is merging partial sparse vectors, in a con-

ventional SpGEMM algorithm for data-parallel architectures [LYA18,LWAQ19,FC23a,WMZ+19, LV14, DBM+15]. Each partial sparse vector is considered as a stream of keys (i.e., representing row/column indices of non-zeros in a matrix) and corresponding non-zero values. At the core of SparseZipper is its ability to efficiently merge such streams in parallel by leveraging in-place matrix registers to store parts of concurrent streams and built-in systolic array to merge those streams together. In order to facilitate that merge operation, we propose a minimal set of additional architectural states to keep track of active streams and matrix instructions to move streams between matrix registers and memory. Our performance evaluations show SparseZipper achieves 5.98× and 2.61× speedup over a scalar hash-based implementation of SpGEMM and a vectorized SpGEMM version respectively.

Contributions – Our key contributions include: (1) an ISA extension called SparseZipper that includes new matrix instructions to efficiently support sorting and merging key-value streams, a main operation in merge-based SpGEMM algorithm on matrices represented in CSR or CSC formats, (2) a merge-based SpGEMM implementation using the proposed matrix instructions, (3) a minimal set of micro-architectural changes to an integrated systolic array to support the new stream sorting and merging instructions, and (4) a detailed cycle-level performance evaluation showing benefits of using the new instructions in accelerating SpGEMM.

5.2 Background: Modern Matrix Architectures

The ever-growing importance of GEMM performance and efficiency in emerging workloads such as machine learning has driven architects to design and integrate accelerators for dense GEMM in modern systems. For example, Google introduced its Tensor Processing Unit (TPU) as a co-processor next to a general-purpose CPU for accelerating training and inference kernels in machine learning workloads [JYP+17, Tie20, JYK+20, JKL+23a]. At the heart of TPU is a large matrix-matrix multiply unit that significantly improves the performance and energy efficiency of multiplying two dense matrices compared to contemporary CPUs and GPGPUs. NVIDIA also integrates tensor cores specialized for multiplying and adding matrices in its recent GPUs [CGG+21a].

The need for accelerating GEMM has pushed specialization for dense GEMM further into modern general-purpose CPU instruction sets as well. Arm recently released its Scalable Matrix

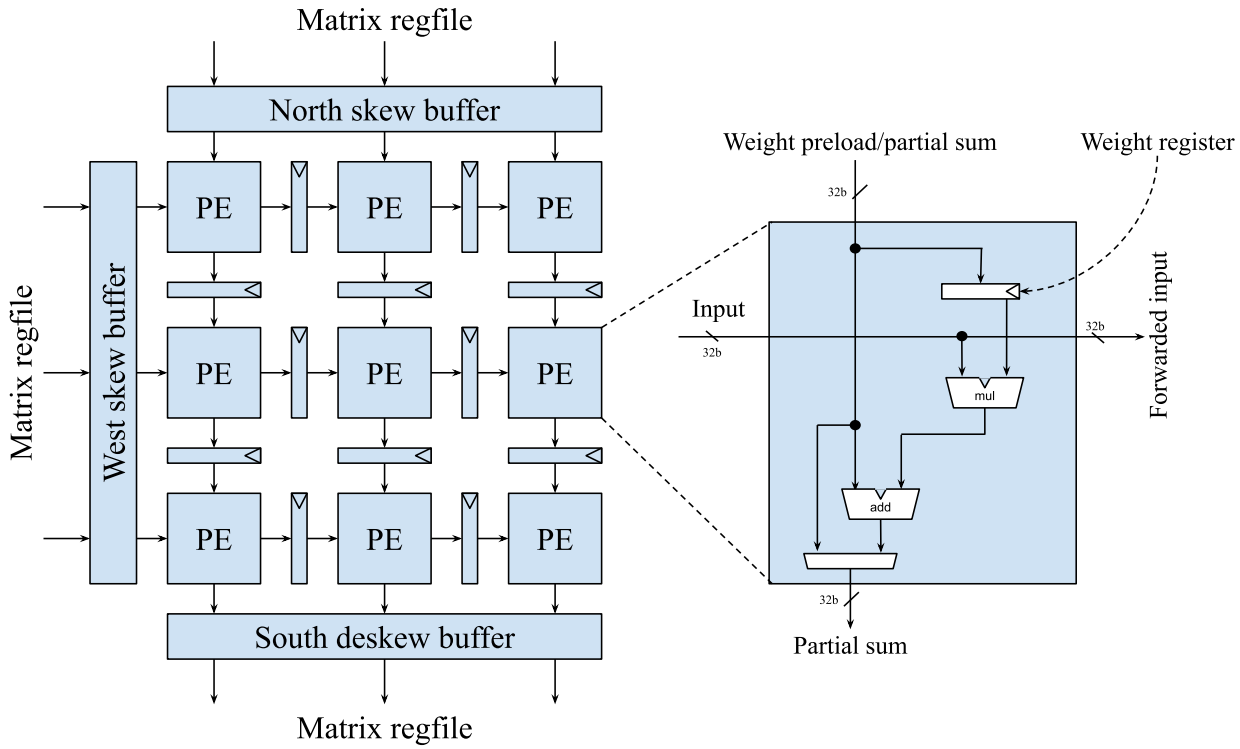


Figure 5.1: A Baseline Systolic Array Micro-architecture for Accelerating Dense GEMM – PE = processing element.

Extension (SME) that introduces a new instruction performing an outer product of two vectors and accumulating its results into a new two-dimensional accumulator register state [arm23]. IBM took a similar approach in its Matrix-Multiple Assist (MMA) extension for the Power ISA [ibm23]. Intel introduced a new Advanced Matrix Extension (AMX) that adds several two-dimensional matrix register states called tile registers and a new matrix-matrix multiply instruction performing a matrix multiplication on two input tile registers [int23b, NMM+22]. The RISC-V community is also working on a matrix extension proposal [ris23] that is similar to Intel AMX's approach.

Regardless of programming abstractions (e.g., accelerator-based interfaces and instruction sets), specialization for dense GEMM is typically implemented in hardware using a two-dimensional systolic array consisting of multiply-add processing elements (PEs) [JYP+17,JQS+21,NMM+22]. An implementation of a systolic array can be either input-, weight-, or output-stationary (i.e., which input/output matrix stays inside the systolic array throughout the computation), depending on its programming abstraction. The integration of a matrix-multiply unit and a general-purpose CPU can be either coarse-grained (e.g., as a co-processor like TPU), medium-grained (e.g., sharing some levels of caches with the CPU), or fine-grained (e.g., as a functional unit in the CPU's pipeline).

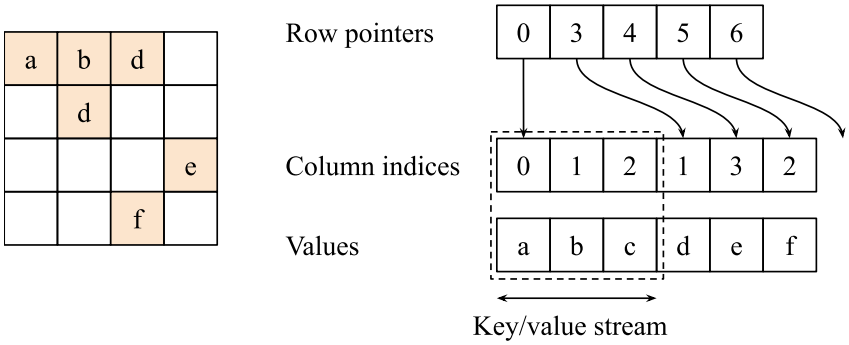


Figure 5.2: Compressed Sparse Row Format

Figure 5.1 shows a weight-stationary systolic array for accelerating dense GEMM. The array consists of multiple PEs connected in a two-dimensional mesh network. Each PE receives input data from its west and north input ports, sends output value (i.e., partial sum of an output matrix element) to its south neighbor PE, and forwards input data to its east neighbor PE. Each PE consists of a weight register for keeping weight values that are multiplied with input values over multiple iterations in a weight-stationary GEMM implementation. In addition, a multiply-accumulate (MAC) unit is used to multiply a weight value (i.e., in the weight register) with an input value (i.e., from the west input port) and accumulate the result into an output value (i.e., from the north input port). There are skew buffers in the west and north input ports so that input data are staggered in time when entering the systolic array.

5.3 Background: Sparse General Matrix Multiplication

Sparse general matrix multiplication (SpGEMM) is a commonly used building block in various application domains including graph analytics [Dav19, HS11, ST15], machine learning [RCK+20, NMS+19, HMD15, JYP+17, WBC+19b], and scientific computing [CGM+96, Gal96]. This section describes widely used data structures for representing sparse matrices, various algorithms for multiplying two sparse matrices, and their trade-offs.

5.3.1 Sparse Matrix Formats

Since most elements in a sparse matrix are zeros, it is efficient to store only non-zero elements to minimize the amount of required storage and also avoid ineffectual computation (i.e., multiply-

ing with zero). The most common data structures for storing a sparse matrix are coordinate format (COO), compressed sparse row (CSR), and compressed sparse column (CSC). They are the default data structures in many widely used linear algebra libraries such as Intel Math Kernel Library (MKL) [WZS+14], CUSPARSE [NCVK10], and Matlab [GMS92].

COO format stores a list of tuples including row index, column index, and value of non-zero elements in a sparse matrix. Typically, the list is kept sorted by either row indices, column indices, or a combination of both row and column indices for more efficient lookups and memory accesses. Further improving the storage efficiency, compared to COO, CSR format groups per-row non-zero elements so that the row indices for elements in the same row can be further compressed. Figure 5.2 shows an example of CSR format. Non-zero elements in each row are represented by a stream of keys (i.e., column indices) and values (i.e., element data). Typically, elements in a stream are sorted by their keys, and streams of adjacent rows are placed consecutively in memory to form the column index and value arrays. The row pointer array consists of pointers indicating where each row starts in the column index and value arrays. CSC format is similar to CSR format except that non-zero elements are compressed by columns.

In addition to those general sparse matrix formats, there are numerous other sparse matrix formats specialized for certain structures of non-zero elements and/or target architectures [BVWH14, HDUZ21, GMFC21, SJL+20, CLY+18]. However, such specialized matrix formats often incur format converting overheads, and their performance is not portable across different matrices and architectures. In this chapter, we target CSR and CSC formats that are storage-efficient and widely used sparse matrix formats.

5.3.2 SpGEMM Dataflows

Figure 5.3 shows common dataflows for SpGEMM: inner-product, outer-product, and row-wise-product (also known as Gustavson's algorithm).

Inner-product dataflow – This dataflow computes each element in the output matrix one at a time by doing a dot product between two vectors (i.e., a row in matrix A and a corresponding column in matrix B), as shown in Figure 5.3. Since there is no data dependency among output elements, the inner-product dataflow is highly parallelizable by computing multiple dot products for different output elements in parallel. In addition, this dataflow also has good data locality for the output matrix but low data reuse for the input matrices. One major downside of this approach

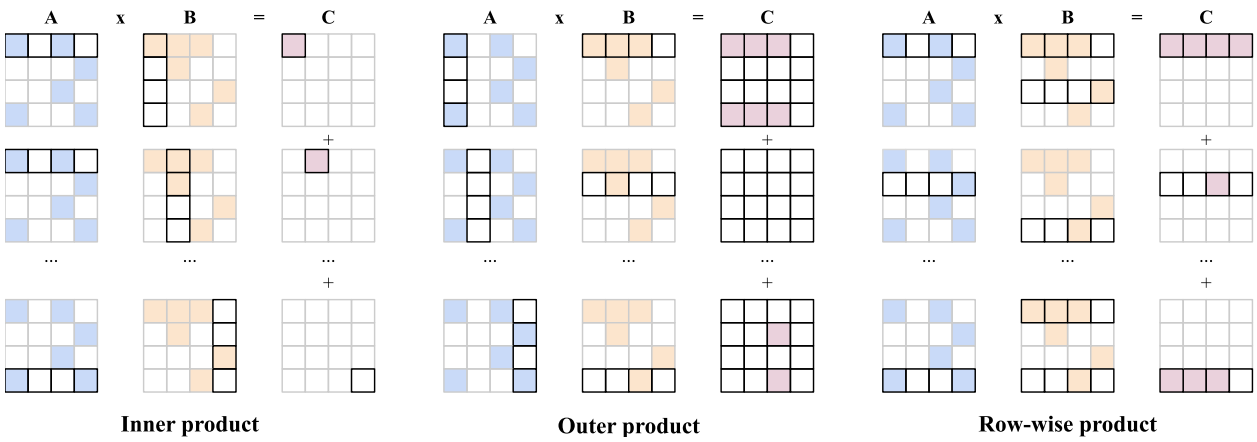


Figure 5.3: Different SpGEMM Dataflows – Non-zero elements in the input matrices (i.e., A and B) and the output matrix (i.e., C) are colored. Zero elements are left blank. In each computation step, involved matrix elements are shown within the dark borders.

is the inefficiency of doing a dot product between two sparse vectors. Since input matrices are sparse, the output matrix is likely to have few non-zero elements. Therefore, most of the dot products result in zeros. Furthermore, the dot product of two sparse vectors requires intersecting two lists of indices, and this index matching is highly inefficient due to the high sparsity of a row and a column in the input matrices.

Outer-product dataflow – Unlike the inner-product dataflow, the outer-product dataflow computes partial results for the entire output matrix at each step by doing an outer product between a column in matrix A and a corresponding row in matrix B, as shown in Figure 5.3. Partial results are combined into a single final output matrix C. All non-zero elements in a column in matrix A and a row in matrix B are used to generate non-zero elements in matrix C, so the outer-product dataflow avoids the problem of ineffectual index matching operations in the inner-product dataflow. Multiple outer-product operations on different pairs of matrix A's columns and matrix B's rows can happen in parallel. However, combining partial outer-product results typically requires complex synchronizations, and this step is often a performance bottleneck in this approach. Despite having good input data reuse, the outer-product dataflow has low output data reuse, and the aggregate memory space of all partial output matrices is often much larger than the final output matrix's size.

Row-wise-product dataflow – The row-wise-product dataflow computes each row of the output matrix at a time by multiplying a row in matrix A (sparse vector) with the entire matrix B, as shown in Figure 5.3. Similar to the outer-product dataflow, this row-wise-product dataflow is work-efficient since it processes only non-zero input elements that contribute to generating non-

Dataflow	Data Access Efficiency			Sparse Format			Algorithmic Efficiency	
	Mtx A	Mtx B	Mtx C	Mtx A	Mtx B	Mtx C	Time	Space
Inner-product	Low	High	High	CSR	CSC	Either	Low	High
Outer-product	High	High	Low	CSC	CSR	CSR	High	Low
Row-wise-product	High	Low	High	CSR	CSR	CSR	High	Medium

Table 5.1: Trade-offs Among Different SpGEM Dataflows

zero output elements. In each step, a multiplication between a sparse vector and a sparse matrix may require merging some partial output vectors into a final row in the output matrix. However, compared to the two-dimensional matrix merging step in the outer-product dataflow, merging partial one-dimensional sparse vectors is much less complex and requiring much less temporary memory space, which may fit in on-chip caches. The row-wise-product dataflow has relatively poor data reuse of matrix B since column indices of non-zero elements in each row in matrix A are used to access corresponding rows in matrix B. Finally, unlike both the inner-product and outer-product approaches, the row-wise-product dataflow does not require input matrices to be stored in two different formats: CSR and CSC. All input and output matrices can be consistently stored in CSR, so there is no need for converting between sparse matrix formats.

Table 5.1 summarizes trade-offs among the three SpGEMM dataflows. In this work, we consider only the row-wise-product dataflow since it is relatively work-efficient by avoiding the extremes in both inner-product and outer-product dataflows.

5.3.3 Row-Wise Product Algorithms

There are three common algorithms to implement the row-wise-product dataflow for SpGEMM: array-based, hash-based, and expand-sort-compress algorithms.

Array-based row-wise SpGEMM – This is also known as sparse accumulator algorithm (SPA) [GMS92]. Figure 5.4 shows an example of how this algorithm works. The algorithm uses a set of three dense arrays that have the same size as the number of columns in the output matrix. The first two arrays store accumulated values and valid flags. When a key-value tuple is inserted, it accesses the two arrays by the key. If the flag for the key is false (i.e., no valid value has been added for this key yet), it overwrites the corresponding entry in the accumulated value array and sets the flag to true. Otherwise, the new value is added to the existing value. The third array stores

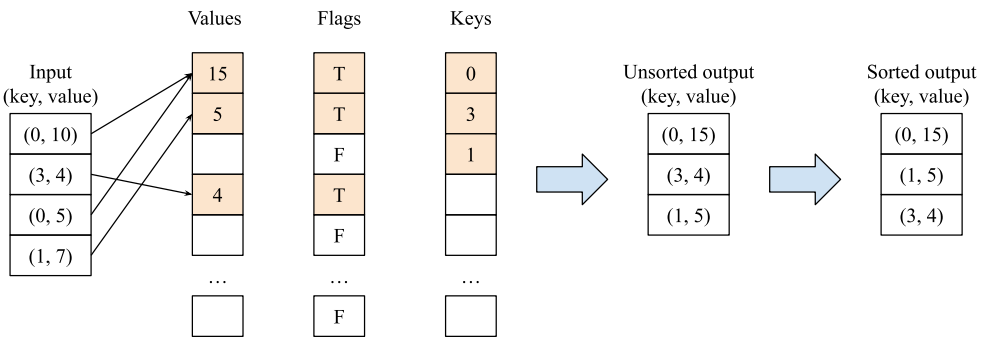


Figure 5.4: Array-Based SpGEMM Algorithm

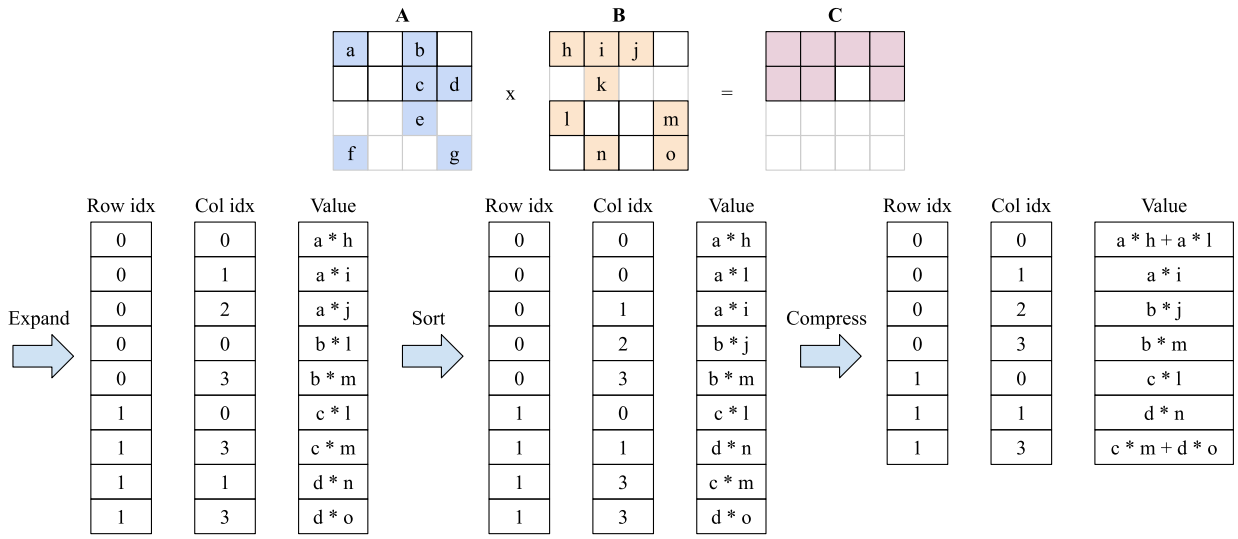


Figure 5.5: Expand-Sort-Compress Algorithm with the Block Size of Two Rows – Non-zero elements are colored in the input and output matrices. Zero elements are left blank. A block of two adjacent rows in the matrix A and their corresponding rows in the matrix B are expanded, sorted, and compressed together to generate output values for the two rows in the output matrix C. All rows involved in the current computation step are shown within the dark borders.

inserted keys in the order they are first added to the list. Those keys are also indices to the value and flag arrays. After all key-value tuples are inserted, we can iterate through the key array to construct an unsorted list of key-value output list. The list is then sorted by keys to produce one row of the output matrix. The flag array needs to be reset before processing the next matrix row.

Hash-based row-wise SpGEMM – Another common approach to accumulate sparse values in row-wise SpGEMM is using a hash table [AFW16, DTR18]. Key-value tuples are inserted into a hash table based on their hashed keys. Similar to the array-based row-wise SpGEMM, key-value tuples in the hash table are not sorted, so a final sorting step is needed to generate an output matrix with per-row non-zero elements sorted by their column indices.

Expand-Sort-Compress (ESC) algorithm – ESC algorithm was initially proposed for performing SpGEMM on GPUs [DOB15, WMZ+19] and later adopted to vector architectures [FC23b, LWAQ19]. Figure 5.5 shows an example of how this ESC algorithm works. Unlike the array-based and hash-based SpGEMM algorithms, more than one row of the input matrix A and output matrix C can be processed together to increase the amount of work that can be parallelized or vectorized. Intermediate results of multiplications are expanded in triples of row index, column index, and value. The list of triples are then sorted by their row and then column indices. Triples with duplicate key (i.e., same row and column indices) are compressed into one entry with the values being accumulated in the final output.

5.4 SparseZipper Instruction Set Extension

In this section, we first describe a merge-based SpGEMM that fuses the sorting and compressing steps of the ESC algorithm into one merging step. The merging step is similar to a typical merge sort algorithm except that two key-value tuples with the same key are combine into one tuple. We then propose and specify an instruction set extension that can accelerate this merging procedure. The extension is built on top of the existing matrix instruction set that is designed for accelerating dense GEMM.

5.4.1 Merge-Based SpGEMM

In the ESC algorithm, one or multiple adjacent rows of an output matrix can be processed at a time. Figure 5.6 shows an example of using the ESC algorithm to produce one row of the output matrix C. Partial results for an *i-th* row in matrix C are generated by multiplying each non-zero element `A[i][j]` in an *i-th* row of matrix A with all non-zero elements `B[j][k]` in a *j-th* row of matrix B. After this expansion phase, for each row in matrix C, we get a list of tuples, each of which is a pair of a column index (or key) and a partial value. Each list of key-value tuples is called a stream. We then combine this stream of partial results into a single sorted stream of unique key-value pairs representing non-zero elements in an *i-th* row of matrix C.

Section 5.3 describes one way of combining those partial key-value pairs by first sorting the list by keys (e.g., using radix sort), then compressing adjacent tuples with duplicate keys, and

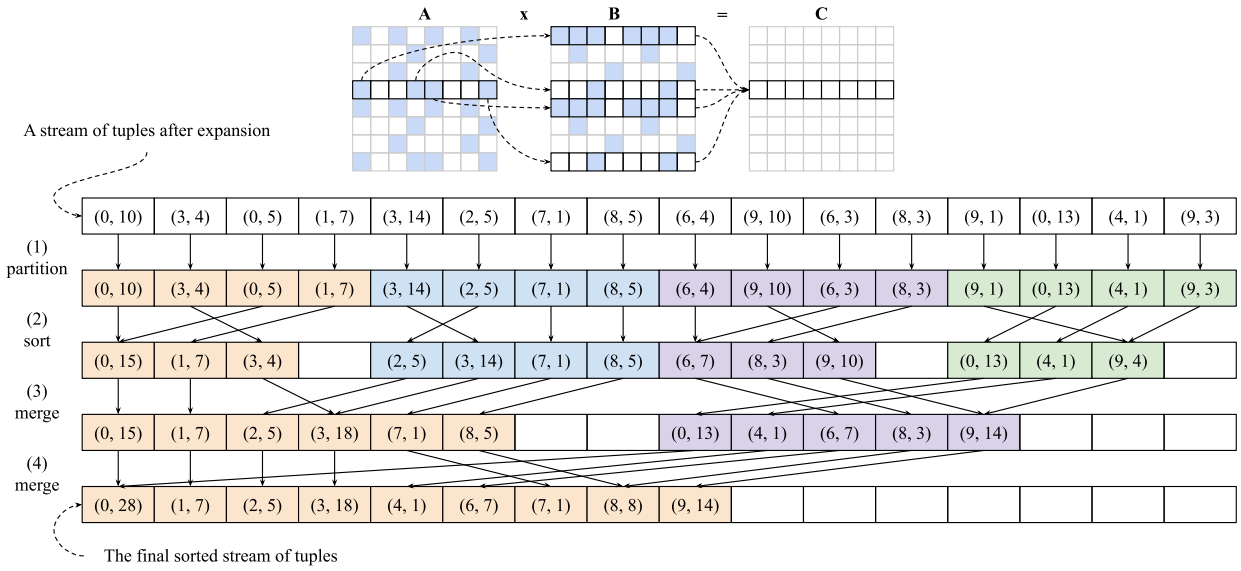


Figure 5.6: Multiple Steps Merge Partial Results for a Row in Output Matrix C – Each tuple is a pair of a column index (i.e., key) and a partial value. A list of tuples, called a stream, represents all partial results after multiplying each non-zero in a row of matrix A (e.g., colored elements in the row with black borders) and all non-zeros in the corresponding rows (e.g., rows with black borders) in matrix B. A stream is split into multiple groups of adjacent tuples (i.e., ones with the same color), and each group is called a partition of the stream.

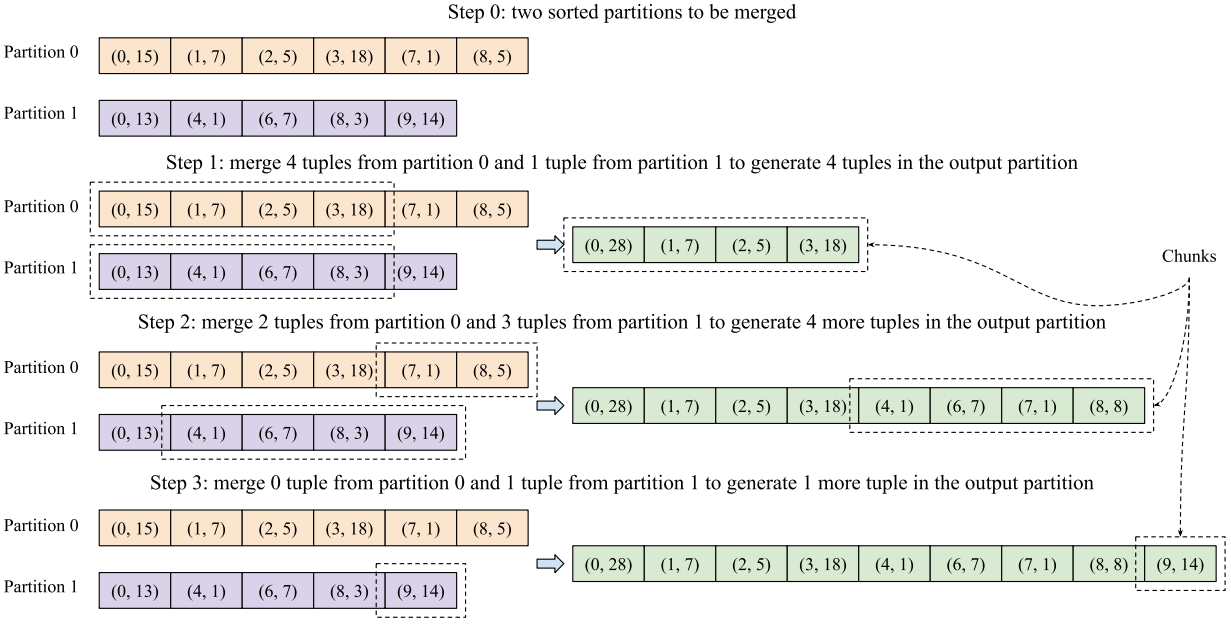


Figure 5.7: Multiple Steps to Merge Two Sorted Partitions of Key/Value Tuples – Each step processes two chunks of at most N elements (e.g., four in this example), one chunk from each input partition, and generates one output chunk.

finally adding up values having the same keys. Figure 5.6 shows another approach that performs a merging operation. First, the list is split into equally sized partitions of tuples. Tuples in each partition are then sorted by their keys. Finally, adjacent partitions are merged together in multiple reduction steps to form a final sorted streams of tuples, as shown in Figure 5.6. Each reduction step reduces the number of partitions by half while increasing the size of each partition. This sorting and merging procedure is similar to a typical merge sort algorithm except that tuples with duplicate keys are combined in each sorting and merging step. Therefore, the final stream of tuples may have fewer elements than the original expanded list of partial results.

In each stream, two sorted partitions of tuples can be merged by repeatedly comparing two tuples with the smallest keys, one from each partition. The tuple with a smaller key is moved from its input partition into the output partition. If the two tuples have the same key, their values are added up, and the tuple of the key and the sum value is added to the output partition. Finally, both tuples with the same key are removed from their input partitions. The process continues until both partitions run out of tuples.

Instead of processing one tuple from each partition at a time, we can load and merge two chunks of N tuples, one chunk from each partition, in one step. Figure 5.7 shows an example of merging two long sorted partitions of tuples by repeatedly merging two N-element chunks, one from each tuple, at a time. It is important to note that we may not be able to move all N tuples from each partition in one step. For example, in step one (in Figure 5.7), three tuples (4, 1), (6, 7), and (8, 3) from partition one cannot be moved to the output partition since their keys are greater than every key from the current chunk in partition zero. Instead, those tuples are merged in a subsequent step. The number of tuples that we can advance at the end of a step in each partition is data-dependent.

In the next sections, we propose a new instruction set extension called SparseZipper for accelerating the two primitive operations in this merge-based approach to SpGEMM: (1) stream sorting (i.e., sorting N tuples of keys and values) and (2) stream merging (i.e., merging two sorted partitions of key-value tuples in a stream).

5.4.2 Architectural Register States

SparseZipper extends a matrix instruction set designed for dense GEMM and a vector instruction set, so it inherits both vector and matrix register states from the base vector and matrix ISAs. Without the loss of generality, in this work, we use the RISC-V vector extension [RIS21] as the

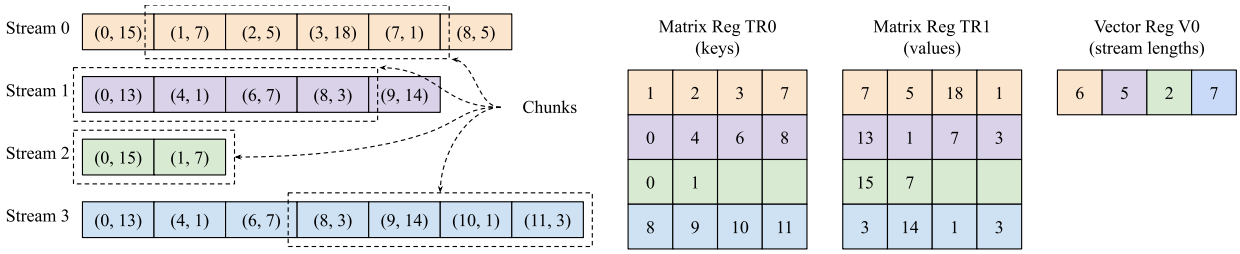


Figure 5.8: Logical Mapping Between Key/Value Streams and Matrix Registers – Only chunks of key/value tuples with dashed borders are held in the two matrix registers. The vector register `V0` stores the total number of elements in the four streams.

base vector ISA and a base matrix instruction set inspired by Intel AMX [int23b] and a proposed RISC-V matrix extension [ris23].

Matrix registers – The base matrix ISA supports eight general-purpose two-dimensional matrix registers (also known as tile registers) named `TR0`, `TR1`, `TR2`, `TR3`, `TR4`, `TR5`, `TR6`, and `TR7`. The length in bits for a row in a matrix register is the same as the number of bits in a single vector register (i.e., `VLEN`) as defined in the RISC-V vector extension. In the scope of this thesis, we limit the size in bits of each element (i.e., `ELEN`) in a vector register to 32 bits to simplify our description of SparseZipper. A complete matrix instruction set may support other element bit widths such as 8, 16, and 64 bits. There are $R = VLEN/ELEN$ elements in a row of a matrix register. We assume that each matrix register has the same number of rows as the number of elements in a row. Therefore, the total number of elements in a matrix register is R^2.

Mapping streams to matrix and vector registers – SparseZipper enables processing multiple streams of key-value tuples in parallel by mapping those streams to different rows of matrix registers. Figure 5.8 shows an example of this stream-register mapping. Matrix register `TR0` stores keys, and matrix register `TR1` stores values. Each row of a matrix register is mapped to a stream. Since matrix registers have a limited size defined by the hardware vector length and a stream can be arbitrarily long, only a chunk of each stream can be held in matrix registers and processed at a time. SparseZipper uses existing vector registers to store other information about those streams (e.g., vector register `V0` in Figure 5.8 stores the number of tuples in each stream).

Special-purpose counter vector registers – In addition to registers in the base vector and matrix ISAs, SparseZipper introduces a set of four light-weight special-purpose input and output counter vector registers: `IC0`, `IC1`, `OC0`, and `OC1`. Their usage is specified in details in the following instruction set specification. Since each counter in a counter vector register counts up to the max

Instructions	Description
Matrix Instructions for Dense GEMM (base matrix ISA)	
`mmult.tt td1, ts2, ts3`	Multiply matrices in `ts2` and `ts3` and accumulate results into `td1`
`mlse.t td1, 0(rs1), rs2, vs3`	Load matrix data into `td1` using a constant stride `rs2`
`msse.t ts1, 0(rs1), rs2, vs3`	Store matrix data from `ts1` using a constant stride `rs2`
Matrix Instructions for Sparse GEMM (SparseZipper ISA extension)	
`mszipk.tt td1, td2, vs1, vs2`	Merge keys in `td1` and `td2`
`mszipv.tt td1, td2, vs1, vs2`	Merge values in `td1` and `td2` based on last key merging results
`mssortk.tt td1, td2, vs1, vs2`	Sort keys in `td1` and `td2`
`mssortv.tt td1, td2, vs1, vs2`	Sort values in `td1` and `td2` base on last key sorting results
`mlxe.t td1, 0(rs1), vs2, vs3`	Load data into `td1` using indices in `vs2`
`msxe.t ts1, 0(rs1), vs2, vs3`	Store data from `ts1` using indices in `vs2`
`mmv.vi vd, cimm`	Move values from an input counter vector `IC[cimm]` to `vd`
`mmv.vo vd, cimm`	Move values from an output counter vector `OC[cimm]` to `vd`

Table 5.2: List of Matrix Instructions – The set of instructions for dense GEMM are inspired by the Intel AMX specification [int23b] and RISC-V matrix extension proposal [ris23]. This work proposes the set of instructions for sparse GEMM.

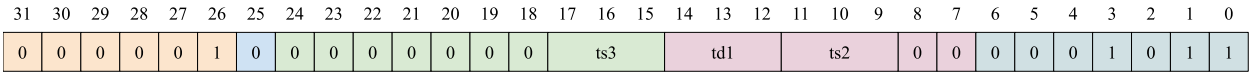
number of elements (i.e., `R`) in a row of a matrix register, each counter is $\log_2 R$-bit wide. Therefore, each counter vector register has $R \times \log_2 R$ bits in total.

5.4.3 Instruction Set Specification

Table 5.2 summarizes a list of matrix instructions for both dense and sparse GEMM. The base matrix ISA includes three matrix instructions: (1) matrix multiply `mmult.tt`, (2) strided matrix load `mlse.t`, and (3) strided matrix store `msse.t` for accelerating dense GEMM. Their syntax, encodings, and semantics are shown in Figure 5.9. SparseZipper extends the base matrix ISA by adding four groups of new matrix instructions: (1) indexed matrix load/store instructions, (2) stream sorting instructions, (3) stream merging/zipping instructions, and (4) counter vector move instructions.

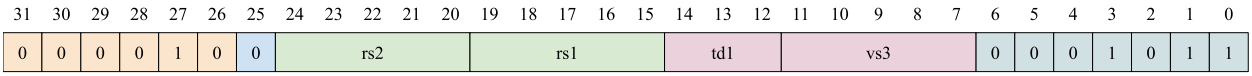
Indexed matrix load/store instructions – SparseZipper introduces two new memory instructions to move data of multiple streams between matrix registers and memory. Figure 5.10 shows the syntax, encodings, and semantics of the indexed matrix load (i.e., `mlxe.t`) and store (i.e., `msxe.t`) instructions. Unlike dense GEMM in which memory accesses to adjacent rows of a dense matrix are regular and distanced by a constant stride, in the merge-based SpGEMM, streams of key-value tuples often have different lengths and are located at arbitrary memory locations. There-

mmult.tt td1, ts2, ts3




```
integer esize = 32                          # element size in bits
integer dim   = MAX_VLEN / esize            # matrix dimension in elements
array<esize>[dim][dim] src_op_0 = TREG[ts2]  # first input matrix
array<esize>[dim][dim] src_op_1 = TREG[ts3]  # second input matrix
array<esize>[dim][dim] dst_op   = TREG[td1]  # output matrix
for i = 0 to dim-1:
  for j = 0 to dim-1:
    float tmp = float(dst_op[i][j])
    for k = 0 to dim-1:
      tmp += float(src_op_0[i][k]) * float(src_op_1[k][j])
    dst_op[i][j] = tmp
```

mlse.t td1, 0(rs1), rs2, vs3



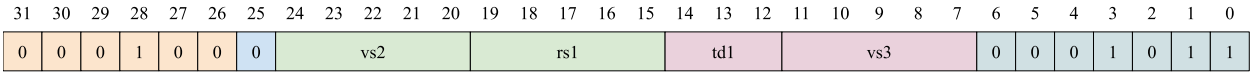

```
integer esize = 32                          # element size in bits
integer dim   = MAX_VLEN / esize            # matrix dimension in elements
integer                src_op_0 = SREG[rs1]  # base address
integer                src_op_1 = SREG[rs2]  # stride
array<esize>[dim]      src_op_2 = VREG[vs3]  # row lengths
array<esize>[dim][dim] dst_op   = TREG[td1]  # output matrix
for row = 0 to dim-1:
  integer addr = src_op_0 + row * src_op_1
  integer rlen = min(dim, src_op_2[row])
  for col = 0 to rlen-1:
    dst_op[row][col] = MEM[addr + col * esize]
```

msse.t ts1, 0(rs1), rs2, vs3


```
integer esize = 32                          # element size in bits
integer dim   = MAX_VLEN / esize            # matrix dimension in elements
integer                src_op_0 = SREG[rs1]  # base address
integer                src_op_1 = SREG[rs2]  # stride
array<esize>[dim]      src_op_2 = VREG[vs3]  # row lengths
array<esize>[dim][dim] src_op_3 = TREG[ts1]  # input matrix
for row = 0 to dim-1:
  integer addr = src_op_0 + row * src_op_1
  integer rlen = min(dim, src_op_2[row])
  for col = 0 to rlen-1:
    MEM[addr + col * esize] = src_op_3[row][col]
```

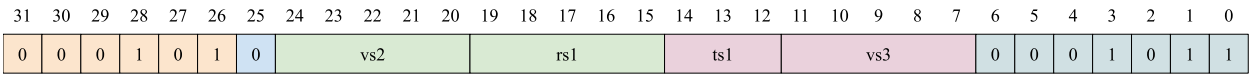
Figure 5.9: Syntax, Encodings, and Semantics of Instructions in the Base Matrix ISA – SREG = scalar registers; VREG = vector registers; TREG = matrix registers; MEM = memory.

mlxe.t td1, 0(rs1), vs2, vs3




```
integer esize = 32                          # element size in bits
integer dim   = MAX_VLEN / esize            # matrix dimension in elements
integer                 src_op_0 = SREG[rs1]  # base address
array<esize>[dim]       src_op_1 = VREG[vs2]  # index
array<esize>[dim]       src_op_2 = VREG[vs3]  # partition lengths
array<esize>[dim][dim] dst_op    = TREG[td1]  # output matrix
for row = 0 to dim-1:
  integer addr = src_op_0 + src_op_1[row]
  integer rlen = min(dim, src_op_2[row])
  for col = 0 to rlen-1:
    dst_op[row][col] = MEM[addr + col * esize]
```

msxe.t ts1, 0(rs1), vs2, vs3



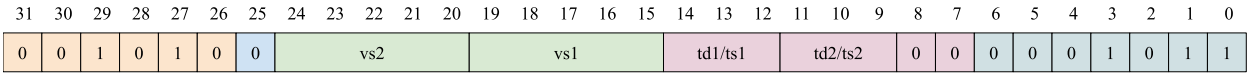

```
integer esize = 32                          # element size in bits
integer dim   = MAX_VLEN / esize            # matrix dimension in elements
integer                 src_op_0 = SREG[rs1]  # base address
array<esize>[dim]       src_op_1 = VREG[vs2]  # index
array<esize>[dim]       src_op_2 = VREG[vs3]  # partition lengths
array<esize>[dim][dim] src_op_3 = TREG[ts1]  # input matrix
for row = 0 to dim-1:
  integer addr = src_op_0 + row * src_op_1
  integer rlen = min(dim, src_op_2[row])
  for col = 0 to rlen-1:
    MEM[addr + col * esize] = src_op_3[row][col]
```

Figure 5.10: Syntax, Encodings, and Semantics of Indexed Load/Store Instructions in SparseZipper – SREG = scalar registers; VREG = vector registers; TREG = matrix registers; MEM = memory.

fore, `mlxe.t` and `msxe.t` take two vector operands: `vs2` specifying memory locations (i.e., byte offsets to a base address) and `vs3` holding stream lengths.

Stream sorting instructions – SparseZipper introduces two new instructions called `mssortk.tt` (i.e., stream sorting for keys) and `mssortv.tt` (i.e., stream sorting for values) to sort multiple chunks of key-value tuples (i.e., up to the maximum number of elements in a row of a matrix register). Figure 5.11 and Figure 5.12 show their syntax, encodings, and semantics. The two instructions work together by first sorting keys (i.e., duplicate keys are combined) and then shuffling values (i.e., values with duplicate keys are added up) based on the key reordering done by `mssortk.tt`. In order to pass the key reordering information between `mssortk.tt` and `mssortv.tt` instructions, SparseZipper adds an abstract special-purpose architectural state that captures how input

mssortk.tt td1, td2, vs1, vs2

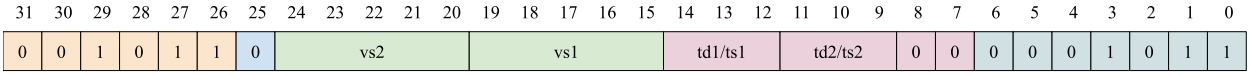



```
integer esize = 32                            # element size in bits
integer dim   = MAX_VLEN / esize              # matrix dimension in elements
array<esize>[dim][dim] src_op_0 = TREG[ts1] # 1st input matrix
array<esize>[dim][dim] src_op_1 = TREG[ts2] # 2nd input matrix
array<esize>[dim]      src_op_2 = VREG[vs1] # lengths of 1st input chunks
array<esize>[dim]      src_op_3 = VREG[vs2] # lengths of 2nd input chunks
array<esize>[dim][dim] dst_op_0 = TREG[td1] # 1st output matrix
array<esize>[dim][dim] dst_op_1 = TREG[td2] # 2nd output matrix
# input-output index maps to be produced
array<map<integer,integer>>[dim] idx_map_0
array<map<integer,integer>>[dim] idx_map_1

for i = 0 to dim-1:
  integer inp_len_0 = min(dim, src_op_2[i])
  integer inp_len_1 = min(dim, src_op_3[i])
  # insert keys into ordered sets
  set<esize> key_set_0
  set<esize> key_set_1
  key_set0.insert(src_op_0[i][0:inp_len_0])
  key_set1.insert(src_op_1[i][0:inp_len_1])
  # update index maps
  for j = 0 to inp_len_0:
    idx_map_0[i].insert(j, key_set_0.index(src_op_0[i][j]))
  for j = 0 to inp_len1:
    idx_map_1[i].insert(j, key_set_1.index(src_op_1[i][j]))
  # update counter vectors
  integer out_len_0 = key_set_0.size()
  integer out_len_1 = key_set_1.size()
  OC0[i] = out_len_0
  OC1[i] = out_len_1
  # update dst_op
  dst_op_0[i][0:out_len_0] = key_set_0[0:out_len_0]
  dst_op_1[i][0:out_len_1] = key_set_1[0:out_len_1]
```

Figure 5.11: Syntax, Encoding, and Semantic of `mssortk.tt` Instruction in SparseZipper – SREG = scalar registers; VREG = vector registers; TREG = matrix registers; MEM = memory; OC* = output counter vector registers; `set` = data structure storing an ordered list of unique values; `map` = key-value map data structure.

mssortv.tt td1, td2, vs1, vs2




```
integer esize = 32                              # element size in bits
integer dim   = MAX_VLEN / esize                # matrix dimension in elements
array<esize>[dim][dim] src_op_0 = TREG[ts1] # 1st input matrix
array<esize>[dim][dim] src_op_1 = TREG[ts2] # 2nd input matrix
array<esize>[dim]      src_op_2 = VREG[vs1] # lengths of 1st input chunks
array<esize>[dim]      src_op_3 = VREG[vs2] # lengths of 2nd input chunks
array<esize>[dim][dim] dst_op_0 = TREG[td1] # 1st output matrix
array<esize>[dim][dim] dst_op_1 = TREG[td2] # 2nd output matrix
# input-output index map produced by last mssortk.tt
array<map<integer,integer>>[dim] idx_map_0
array<map<integer,integer>>[dim] idx_map_1

for i = 0 to dim-1:
  integer out_len_0 = OC0[i] # 1st output chunk's length produced by last mssortk.tt
  integer out_len_1 = OC1[i] # 2nd output chunk's length produced by last mssortk.tt
  # initialize dst_op
  for j = 0 to out_len_0:
    dst_op_0[i][j] = float(0)
  for j = 0 to out_len_1:
    dst_op_1[i][j] = float(0)
  # update dst_op
  integer inp_len_0 = min(dim, src_op_2[i])
  integer inp_len_1 = min(dim, src_op_3[i])
  for j = 0 to inp_len_0:
    out_idx = idx_map_0[i][j]
    dst_op_0[i][out_idx] += float(src_op_0[i][j])
  for j = 0 to inp_len_1:
    out_idx = idx_map_1[i][j]
    dst_op_1[i][out_idx] += float(src_op_1[i][j])
```

Figure 5.12: Syntax, Encoding, and Semantic of `mssortv.tt` Instruction in SparseZipper – SREG = scalar registers; VREG = vector registers; TREG = matrix registers; MEM = memory; OC* = output counter vector registers; `map` = key-value map data structure.

keys are reordered in each chunk of key-value tuples. This state is intentionally left abstract in the SparseZipper ISA as a list of maps of input-output indices (e.g., `idx_map_0` and `idx_map_1` in Figure 5.11 and Figure 5.12), so an implementation of SparseZipper can freely choose how to implement this state. Section 5.5 later discusses one implementation of this state using a systolic array. Instruction `mssortk.tt` also updates the output counter vector registers with the new lengths of sorted output chunks since an output chunk may be shorter than its input chunk (i.e., tuples with duplicate keys are combined).

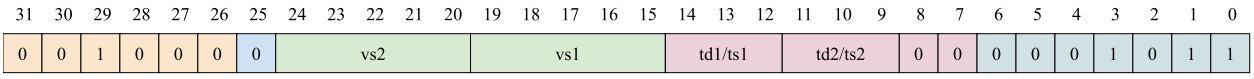
mszipk.tt td1, td2, vs1, vs2


```
integer esize = 32                            # element size in bits
integer dim   = MAX_VLEN / esize              # matrix dimension in elements
array<esize>[dim][dim] src_op_0 = TREG[ts1] # 1st input matrix
array<esize>[dim][dim] src_op_1 = TREG[ts2] # 2nd input matrix
array<esize>[dim]      src_op_2 = VREG[vs1] # lengths of 1st input chunks
array<esize>[dim]      src_op_3 = VREG[vs2] # lengths of 2nd input chunks
array<esize>[dim][dim] dst_op_0 = TREG[td1] # 1st output matrix
array<esize>[dim][dim] dst_op_1 = TREG[td2] # 2nd output matrix
# input-output index maps to be produced
array<map<integer,integer>>[dim] idx_map_0
array<map<integer,integer>>[dim] idx_map_1

for i = 0 to dim-1:
  integer inp_len_0 = min(dim, src_op_2[i])
  integer inp_len_1 = min(dim, src_op_3[i])
  set<esize> key_set  # ordered set of keys from both streams
  IC0[i] = 0          # initialize 1st input counter
  IC1[i] = 0          # initialize 2nd input counter
  # insert keys into the ordered set
  for j = 0 to inp_len_0:
    if (src_op_0[i][j] <= src_op_1[i][inp_len_1]):
      key_set.insert(src_op_0[i][j])
      IC0[i] += 1
  for j = 0 to inp_len_1:
    if (src_op_1[i][j] <= src_op_0[i][inp_len_0]):
      key_set.insert(src_op_1[i][j])
      IC1[i] += 1
  # update index maps
  for j = 0 to inp_len_0:
    idx_map_0[i].insert(j, key_set.index(src_op_0[i][j]))
  for j = 0 to inp_len_1:
    idx_map_1[i].insert(j, key_set.index(src_op_1[i][j]))
  # update dst_op and output counter vectors
  integer out_len_0 = key_set.size() if (key_set.size() <= dim) else dim
  integer out_len_1 = 0 if (key_set.size() <= dim) else (dim - key_set.size())
  for j = 0 to out_len_0:
    dst_op_0[i][j] = key_set[j]
  for j = 0 to out_len_1:
    dst_op_1[i][j] = key_set[out_len_0 + j]
  # update output counter vectors
  OC0[i] = out_len_0
  OC1[i] = out_len_1
```

Figure 5.13: Syntax, Encoding, and Semantic of `mszipk.tt` Instruction in SparseZipper – SREG = scalar registers; VREG = vector registers; TREG = matrix registers; MEM = memory; IC* = input counter vector registers; OC* = output counter vector registers; `set` = data structure storing an ordered list of unique values; `map` = key-value map data structure.

mszipv.tt td1, td2, vs1, vs2

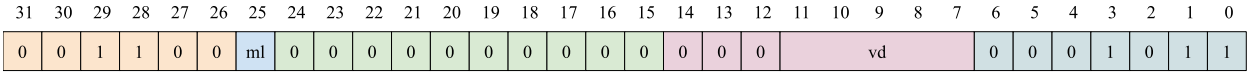



```
integer esize = 32                          # element size in bits
integer dim   = MAX_VLEN / esize            # matrix dimension in elements
array<esize>[dim][dim] src_op_0 = TREG[ts1] # 1st input matrix
array<esize>[dim][dim] src_op_1 = TREG[ts2] # 2nd input matrix
array<esize>[dim]      src_op_2 = VREG[vs1] # lengths of 1st input chunks
array<esize>[dim]      src_op_3 = VREG[vs2] # lengths of 2nd input chunks
array<esize>[dim][dim] dst_op_0 = TREG[td1] # 1st output matrix
array<esize>[dim][dim] dst_op_1 = TREG[td2] # 2nd output matrix
# input-output index map produced by last mszipk.tt
array<map<integer,integer>>[dim] idx_map_0
array<map<integer,integer>>[dim] idx_map_1

for i = 0 to dim-1:
  # initialize dst_op
  integer out_len_0 = OC0[i]
  integer out_len_1 = OC1[i]
  integer out_len   = out_len_0 + out_len_1
  # initialize a temporary array for accumulating values
  array<esize>[out_len] accum_arr
  for j = 0 to out_len - 1:
    accum_arr[j] = float(0)
  # update accum_arr
  for j = 0 to inp_len_0:
    out_idx = idx_map_0[i][j]
    accum_arr[out_idx] += float(src_op_0[i][j])
  for j = 0 to inp_len_1:
    out_idx = idx_map_1[i][j]
    accum_arr[out_idx] += float(src_op_1[i][j])
  # update dst_op
  for j = 0 to out_len_0 - 1:
    dst_op_0[i][j] = accum_arr[j]
  for j = 0 to out_len_1 - 1:
    dst_op_1[i][j] = accum_arr[out_len_0 + j]
```

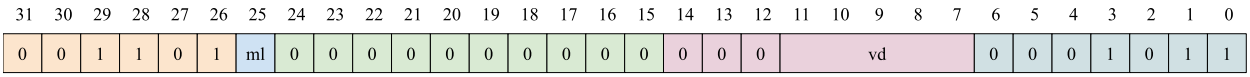
Figure 5.14: Syntax, Encoding, and Semantics of `mszipv.tt` Instruction in SparseZipper – SREG = scalar registers; VREG = vector registers; TREG = matrix registers; MEM = memory; IC* = input counter vector registers; OC* = output counter vector registers; `map` = key-value map data structure.

mmv.vi vd, cimm




```
integer esize = 32                    # element size in bits
integer dim   = MAX_VLEN / esize      # matrix dimension in elements
integer src_imm           = cimm      # counter vector index
array<esize>[dim] dst_op  = VREG[vd]  # output vector
for i = 0 to dim-1:
  dst_op[i] = IC0[i] if src_imm == 0 else IC1[i]  # copy from output counter vector reg
```

mmv.vo vd, cimm




```
integer esize = 32                    # element size in bits
integer dim   = MAX_VLEN / esize      # matrix dimension in elements
integer           src_imm = cimm      # counter vector index
array<esize>[dim] dst_op  = VREG[vd]  # output vector
for i = 0 to dim-1:
  dst_op[i] = OC0[i] if src_imm == 0 else OC1[i]  # copy from output counter vector reg
```

Figure 5.15: Syntax, Encoding, and Semantics of Counter Vector Move Instructions in SparseZipper – VREG = vector registers;

Stream merging/zipping instructions – SparseZipper provides two instructions called `mszipk.tt` (i.e., stream merging for keys) and `mszipv.tt` (i.e., stream merging for values) to merge sorted partitions of key-value tuples in a stream. Figure 5.13 and Figure 5.14 show their syntax, encodings, and semantics. Similar to `mssortk.tt` and `mssortv.tt`, the two zipping instructions work together by first zipping keys (i.e., duplicate keys are combined) and then shuffling values (i.e., values with duplicate keys are added up). The key reordering information is also captured by an abstract special-purpose architectural state (e.g., `idx_map_0` and `idx_map_1` in Figure 5.13 and Figure 5.14) that is produced by `mszipk.tt` and then consumed by `mszipv.tt`. Instruction `mszipk.tt` updates input counter vector registers with the number of tuples that have been merged from each input partition. The output counter vector registers are also updated with the number of elements in each merged output partition.

Counter vector move instructions – In order to extract special-purpose input and output counter vectors (i.e., for updating stream lengths), SparseZipper provides two move instructions that move values from a counter vector register into a general-purpose vector register. Figure 5.15 shows their syntax, encoding, and semantics.

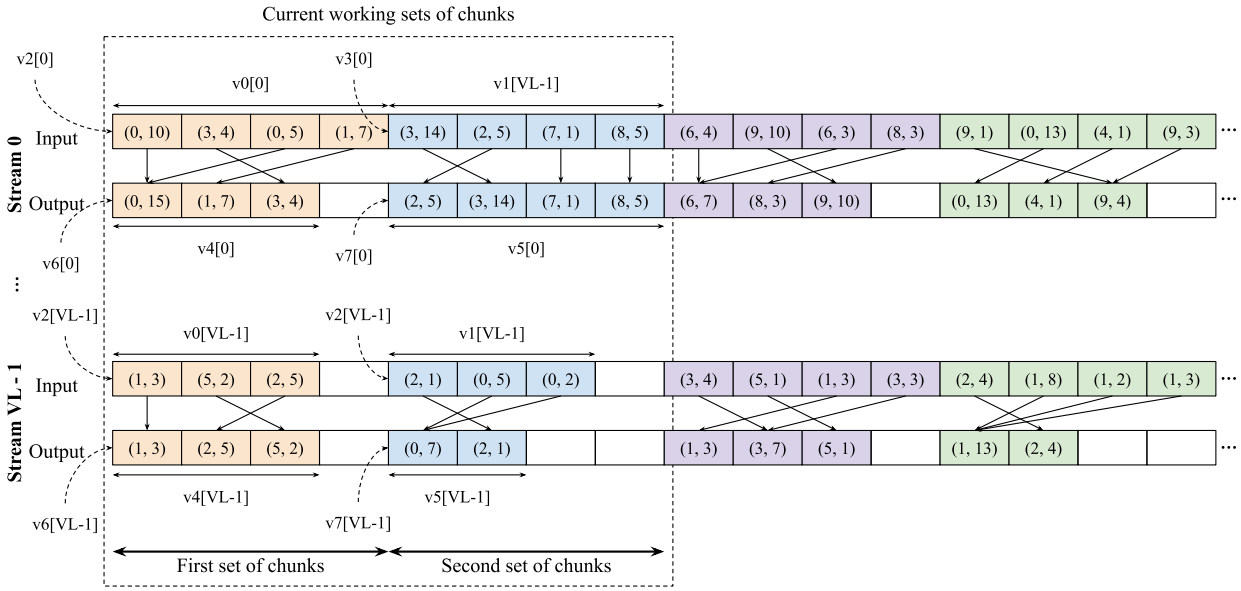
5.4.4 Code Examples

In this section, we show how to use the proposed SparseZipper instruction set extension to sort and merge key-value tuples in multiple streams in parallel.

Sorting key-value tuples – Figure 5.16 shows a RISC-V assembly code snippet of sorting key-value tuples in chunks across multiple streams. Each stream is partitioned into multiple chunks of at most VL key-value tuples. By mapping each stream to a row in matrix registers, we can process a VL number of streams in parallel. Parallelism also happens in each stream by sorting tuples from two adjacent chunks together. Figure 5.16 only shows the most inner loop that processes one set of chunks across VL streams at a time. The most outer loop iterates through groups of VL streams.

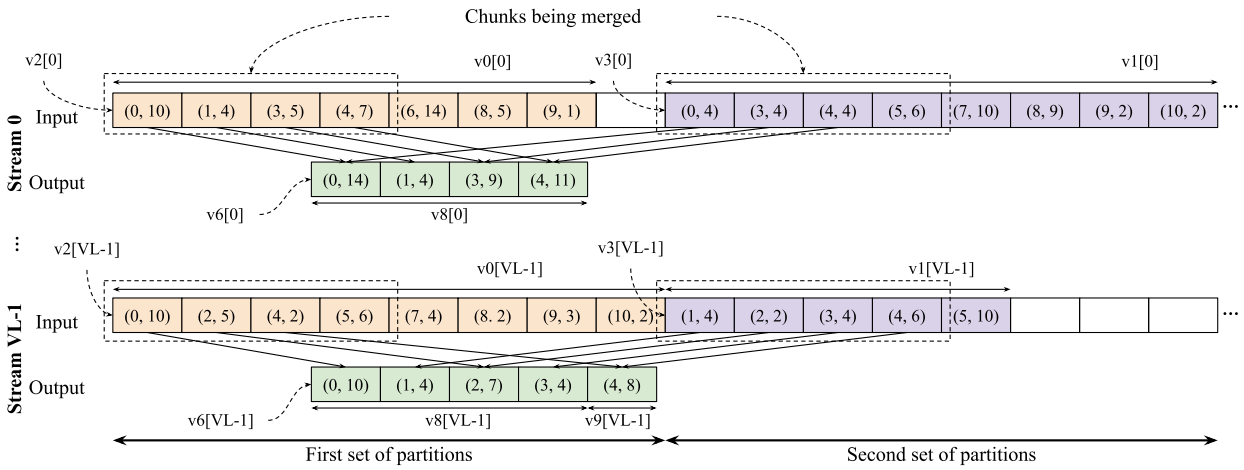
First, keys and values of the current working set of tuples are loaded into matrix registers using `mlxe.tt` instruction (i.e., in lines 10-13). Register `tr0` and `tr2` hold input keys while register `tr1` and `tr3` hold corresponding input values. A set of vector registers are used to track lengths and beginning indices of input and output chunks in the working set. Then instruction `mssortk.tt` sorts two sets of keys (i.e., in `tr0` and `tr2`) and writes the sorted keys into the same matrix registers (i.e., in line 15). Duplicate keys are combined into a single key, so there may be fewer tuples in an output chunk than there are in its input chunk. Instruction `mssortk.tt` writes the number of tuples in each output chunk in the output counter vector registers `OC0` and `OC1`. Instruction `mmv.vo` moves the counter values into general-purpose vector registers (i.e., in lines 17-18). Following instruction `mssortk.tt` that defines the position where each key in an input chunk appears in its output chunk, `mssortv.tt` (i.e., line 16) shuffles values in the input chunk to the correct positions of their corresponding keys in the output chunk. Values having duplicate keys are added up. Finally, keys and values in output chunks are written back to memory using instruction `msxe.t` (i.e., in lines 20-23). The most inner loop then continues processing the next two adjacent chunks across the VL streams.

Merging streams of key-value tuples – Figure 5.17 shows a RISC-V assembly code snippet of the most inner loop merging partitions of key-value tuples across VL number of streams. Each stream consists of multiple partitions, and a pair of adjacent partitions in a stream are merged into one output partition. Since a partition may have an arbitrary length, only a chunk of at most VL tuples from each partition is loaded into matrix registers and processed at each iteration of the most inner loop. The inner loop continues until all tuples in the current working set of input partitions




```
# a0, a1: base address of input key and value arrays
# a2, a3: base address of output key and value arrays
# v0, v1: lengths of the 1st and 2nd input chunks
# v2, v3: pointers to the 1st and 2nd input chunks
# v4, v5: lengths of the 1st and 2nd output chunks
# v6, v7: pointers to the 1st and 2nd output chunks
# v8, v9: index of the current set of chunk pairs and the numbers of chunk pairs
loop:
  # load
  mlxe.t tr0, 0(a0), v2, v0   # load keys of the 1st input chunks
  mlxe.t tr1, 0(a1), v2, v0   # load values of the 1st input chunks
  mlxe.t tr2, 0(a0), v3, v1   # load keys of the 2nd input chunks
  mlxe.t tr3, 0(a1), v3, v1   # load values of the 2nd input chunks
  # sort
  mssortk.tt tr0, tr2, v0, v1 # sort keys of the 1st and 2nd input chunks
  mssortv.tt tr1, tr3, v0, v1 # sort values of the 1st and 2nd input chunks
  mmv.vo v4, 0x0              # get lengths of the 1st output chunks
  mmv.vo v5, 0x1              # get lengths of the 2nd output chunks
  # store
  msxe.t tr0, 0(a2), v6, v4   # store keys of the 1st output chunks
  msxe.t tr1, 0(a3), v6, v4   # store values of the 1st output chunks
  msxe.t tr2, 0(a2), v7, v5   # store keys of the 2nd output chunks
  msxe.t tr3, 0(a3), v7, v5   # store values of the 2nd output chunks
  # check if we finish all pairs
  vadd.vx v8, v8, 0x2
  vmsltu.vv v10, v8, v9
  vpopc.m a4, v10
  bnez a4, loop
```

Figure 5.16: Sorting Chunks of Keys and Values from Multiple Streams in Parallel Using the Proposed Matrix Instructions – This code snippet only shows the most inner loop that iterating through the working set of chunks bordered by the dash lines. VL = vector length; `a{0..4}` = scalar registers; `v{0..9}` = vector registers; `tr{0..3}` = matrix registers.




```
# a0, a1: base address of input key and value arrays
# a2, a3: base address of output key and value arrays
# v0, v1: lengths of the 1st and 2nd input paritions
# v2, v3: pointers to the 1st and 2nd input paritions
# v4: lengths of the output chunks
# v5: pointers to the output chunks
loop:
  # load
  mlxe.t tr0, 0(a0), v2, v0  # load chunks of keys from 1st input partitions
  mlxe.t tr1, 0(a1), v2, v0  # load chunks of values from 1st input partitions
  mlxe.t tr2, 0(a0), v3, v1  # load chunks of keys of 2nd input partitions
  mlxe.t tr3, 0(a1), v3, v1  # load chunks of values of 2nd input partitions
  # zip
  mszipk.tt tr0, tr2, v0, v1 # zip keys of 1st and 2nd input chunks
  mszipv.tt tr1, tr3, v0, v1 # zip values of 1st and 2nd input chunks
  mmv.vi v6, 0x0             # get number of zipped input elements (1st chunks)
  mmv.vi v7, 0x1             # get number of zipped input elements (2nd chunks)
  mmv.vo v8, 0x0             # get number of output elements (1st part)
  mmv.vo v9, 0x1             # get number of output elements (2nd part)
  vsub.vv v0, v0, v6         # update input lengths (1st chunks)
  vsub.vv v1, v1, v7         # update input lengths (2nd chunks)
  vadd.vv v2, v2, v6         # bump input pointers (1st chunks)
  vadd.vv v3, v3, v7         # bump input pointers (2nd chunks)
  # store
  msxe.t tr0, 0(a2), v5, v8  # store keys of 1st output chunks
  msxe.t tr1, 0(a3), v5, v8  # store values of 1st output chunks
  vadd.vv v5, v5, v8         # bump output pointers
  msxe.t tr2, 0(a2), v5, v9  # store keys of 2nd output chunks
  msxe.t tr3, 0(a3), v5, v9  # store values of 2nd output chunks
  vadd.vv v5, v5, v9         # bump output pointers
  ...
  vpopc.m a4, v10   # count the number of active pairs of partitions
  bnez a4, loop
```

Figure 5.17: Zipping Partitions of Keys and Values across Multiple Streams in Parallel Using the Proposed Matrix Instructions – VL = vector length (i.e., four elements in this example); `a{0..4}` = scalar registers; `v{0..10}` = vector registers; `tr{0..3}` = matrix registers.

across all VL streams are merged. An outer loop iterates through pairs of adjacent partitions across VL streams.

In each step of the most inner loop shown in Figure 5.17, keys and values from two chunks of at most VL tuples from two partitions in each stream are loaded into matrix registers using `mlxe.t` instruction (i.e., lines 9-12). Register `tr0` and `tr2` hold input keys while register `tr1` and `tr3` hold corresponding input values. A set of vector registers are used to track lengths and beginning indices of input and output chunks in the current working set. Then instruction `mszipk.tt` merges two sets of sorted keys (i.e., in `tr0` and `tr2`) into one set of sorted keys (i.e., duplicate keys are combined into one key). The output set of keys is stored in the same matrix registers `tr0` and `tr2`. If the output set has more than VL number of keys (i.e., larger than the number of elements a row in a matrix register can hold), `tr0` holds the first VL keys, and `tr2` holds the rest of the keys in the output set. Two output counter vector registers `OC0` and `OC1` are updated with the number of keys each row of `tr0` and `tr2` has after `mszipk.tt` is executed. Line 18-19 uses `mmv.vo` to move values from the output counter vector registers into general-purpose vector registers. As explained previously, not all tuples from an input chunk can be merged in each step. Therefore, `mszipk.tt` also updates two input counter vector registers `IC0` and `IC1` to indicate numbers of tuples that have been merged from input chunks. Following `mszipk.tt` that merges keys from each pair of input chunks into an ordered set of output keys, `mszipv.tt` (in line 15) shuffles values from input chunks of tuples (i.e., in `tr1` and `tr3`), based on the reordering of keys. Values with duplicate keys are added up in the output set of tuples. Finally, keys and values in output chunks are written back to memory using instruction `msxe.t` (i.e., in lines 25-30).

5.5 SparseZipper Micro-Architecture

This section details the SparseZipper micro-architecture that extends the baseline systolic array specialized for dense GEMM (discussed in Section 5.2) to support the set of proposed instructions for accelerating sparse GEMM presented in Section 5.4. We first describe the novel systolic executions of sorting and merging one pair of key-value lists using a systolic array in Section 5.5.1 and 5.5.2. We then discuss how to process multiple pairs of key-value lists stored in multiple rows of matrix registers in Section 5.5.3. Finally, we present a minimal set of micro-architectural changes in the baseline systolic array to realize those systolic executions in Section 5.5.4.

5.5.1 Systolic Execution of Sorting a Pair of Key-Value Lists

Figure 5.18 shows an example of a systolic execution of `mssortk` instruction using a 3×3 systolic array to independently sort two unsorted list of keys in an ascending order cycle by cycle. In cycle 0, the two input lists of keys are located in the west and north sides of the systolic array. Similar to the systolic execution of a dense GEMM, input values are streamed into the systolic array in a skewed pattern. The two sorted output lists of keys come out from the east and south sides of the systolic array. All PEs are connected in the same way as in the baseline systolic array. Each PE receives input data in its west and north input ports, and it sends output data through its east and south output ports.

For `mssortk` instruction, a list of input keys are streamed through the systolic array in two passes: sorting and compressing. In the sorting pass, keys are sorted in an ascending order. Since a list of keys may contain multiple duplicates (e.g., the north-side input list in Figure 5.18), the sorting pass detects and combines those duplicates into one single key. If duplicates exist, the sorted list of output keys will be shorter than the unsorted list of input keys since duplicated keys are excluded and becoming invalid. Those invalid keys may exist in between valid output keys after the first pass. The compressing pass then moves invalid output keys to the end of an output list so that sorted valid keys are placed consecutively without invalid keys in between. For example, in Figure 5.18, after the sorting pass, the north-side input list of {5, 8, 5} comes out of the systolic in the east side as {5, `d`, 8} (i.e., `d` indicates excluded duplicated key(s)). After the compressing pass, the partial output list becomes {5, 8, `d`} in the south side of the systolic array. For both `mssortk` and `mssortv` instructions, north-side inputs come out from east side after the sorting pass. Then they are re-streamed into the systolic array from the west side, and they finally come out from the south side after the compressing pass.

Given a list of keys, `mssortk` generates an ordering in which keys and values should be placed in output lists. Instruction `mssortv` then follows the ordering to sort its list of values. The systolic array uses a set of micro-architectural states in each PE to keep track of the ordering. In each PE, those states encode the direction in which west-side and north-side input data should be routed towards and whether the data should be combined due to duplicates. In each PE, given a pair of incoming inputs, there are four possible states: initial (i.e., no data routing between input and output ports), forwarding, switching, and combining. In the forwarding state, a PE routes west-side and north-side input data to the east and south output ports respectively. In the switching state, a

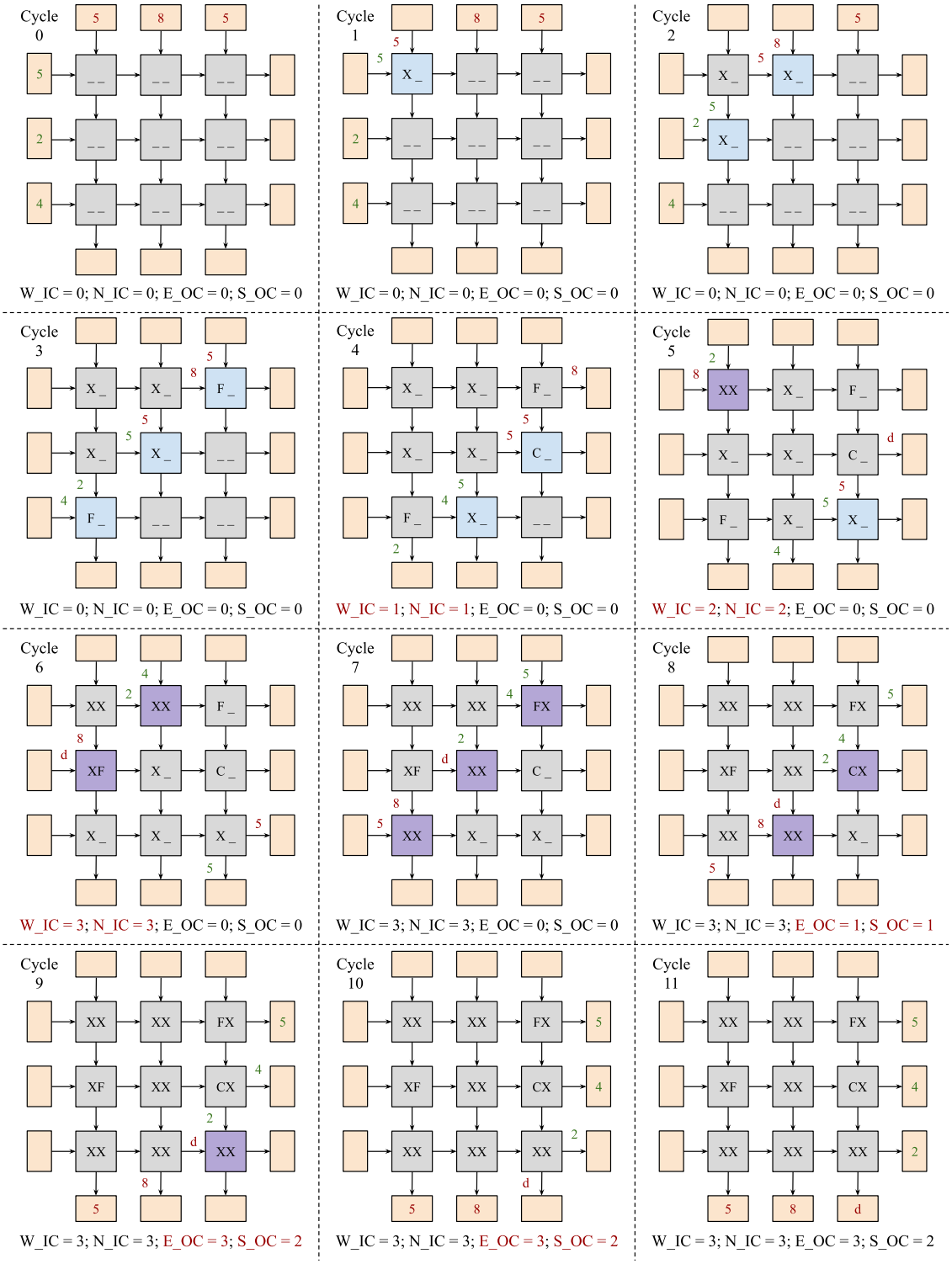


Figure 5.18: Cycle-by-Cycle Systolic Execution of `mssortk` in a 3×3 Systolic Array for Two Unsorted Lists of Keys – PE states: F = forward, X = switch, C = combine; W_IC = west input counter; N_IC = north input counter; E_OC = east output counter; S_OC = south output counter; d = duplicate key that is excluded; Counters in red indicate they are updated in corresponding cycles. PEs in gray are inactive. PEs in blue are zipping keys. PEs in purple are compressing valid output keys. Keys in red come from the north input. Keys in green come from the west input. Keys in west and east sides are ordered from bottom to top. Keys in north and south sides are ordered from left to right.

PE routes west-side and north-side input data to the south and east output ports respectively. In the combining state, a PE combines input data (i.e., producing a single output key for `mssortk`, and adding up two input values for `mssortv`) into a single output. The combined data is routed to the south output port, and the east-side output data is flagged as duplicated. Since each input data is streamed through the systolic array twice, each PE needs to keep track of states for both sorting and compressing passes.

The west- and north-side input lists are sorted independently using either the bottom-left or top-right half of the systolic array. PEs on the main diagonal always switch their input data so that keys and values from the two input lists are not intermixed. In each PE, the sorting and compressing algorithm works by comparing two input keys and routing the larger one to the east output port and the smaller one to the south output port. Overall, larger keys and their values flow to the east side while smaller keys and their values flow to the south side of the systolic array. If two input keys are equal, an output flagged as invalid is sent through the east output port of an PE. In subsequent PEs, an invalid input is considered larger than any valid data and, therefore, always routed to the east output port unless both input data are invalid (i.e., in this case, a PE simply forwards the two invalid inputs to the output ports). Therefore, after the compressing pass, invalid data flow to the end of an output list, and all valid output data stay consecutively in its beginning part.

Instruction `mssortk` also updates a set of counters tracking the number of valid input and output data elements: west-input (`W_IC`), north-input (`N_IC`), east-output (`E_OC`), and south-output (`S_OC`) counters as shown in Figure 5.18. Input counters (i.e., `W_IC` and `N_IC`) are incremented when output elements appear in the east and south output sides of the systolic array after a sorting pass. For example, in Figure 5.18, in cycle 4, 5 and 6, W_IC counts the number of valid elements (i.e., in red) coming from the west side, and N_IC counts the number of valid elements (i.e., in green) coming from the north side. Output counters (i.e., E_OC and S_OC) are updated when output elements come out of the systolic array after a compressing pass (e.g., in cycles 8-10 in Figure 5.18).

Similar to `mssortk`, `mssortv` instruction streams input values through the systolic array twice. However, instead of making comparisons of input values, `mssortv` routes values based on the states previously generated by `mssortk` instruction when sorting corresponding keys. Therefore, values flow through the systolic array in the same pattern as their keys. In a PE, if the state is combining, two input values are added up. The accumulated value is then forwarded to the PE's

south output port while an invalid value is generated for the east output port. Instruction `mssortv` does not update the input and output counters.

5.5.2 Systolic Execution of Merging a Pair of Sorted Key-Value Lists

Figure 5.19 shows an example of a systolic execution of `mszipk` instruction using a 3×3 systolic array to merge two sorted key-value lists into one sorted list. In cycle 0, the two input lists of sorted keys are placed in the west and north sides of the systolic array. Keys in the west-side list are ordered from bottom to top with the smallest key staying at the bottom while keys in the north-side list are ordered from left to right with the smallest key staying on the left. The final output list is stored in two parts. The part with smaller keys (e.g., {2, 3, 5} in Figure 5.19) are located in the east-side output while the other part with larger keys (e.g., {8} in Figure 5.19) are stored in the south-side output.

In this merging operation, input keys flow through the systolic array in two passes: merging and compressing. The merging pass generates a list of sorted keys with possible duplicated keys to be excluded (i.e., invalid output keys) in between valid output keys. The compressing pass moves those invalid keys to the end of the list so that valid output keys stay in consecutive positions. Unlike the sorting pass in the sorting operation, the merging pass intermixes keys from both input key-value lists. PEs on the main diagonal in the merging pass work exactly the same as other PEs in the array instead of always doing north-to-east and west-to-south routing as in the sorting pass of the sorting operation. Similar to the sorting operation, larger keys and their values flow to the east side while smaller keys and their values flow to the south side of the systolic array. The compressing pass of this merging operation works exactly the same as discussed earlier in Section 5.5.1.

Unlike the sorting operation, not all valid input keys become valid output keys, as discussed in Section 5.4.1. Keys from a list that are greater than every key in the other list need to be excluded since we do not know yet their positions after the current merging step. For example, in Figure 5.19, the key 9 from the west-side input list is excluded from the output list (i.e., becoming an `x` in cycle 4) since it is greater than every key from the north-side list. In order to detect such keys to be excluded, each key is tagged with two extra bits to track (1) from which input side the key comes from (i.e., source bit) and (2) whether the key has been compared with another larger or equal key from the other input side yet (i.e., merge bit). The merge bit is initially set to false

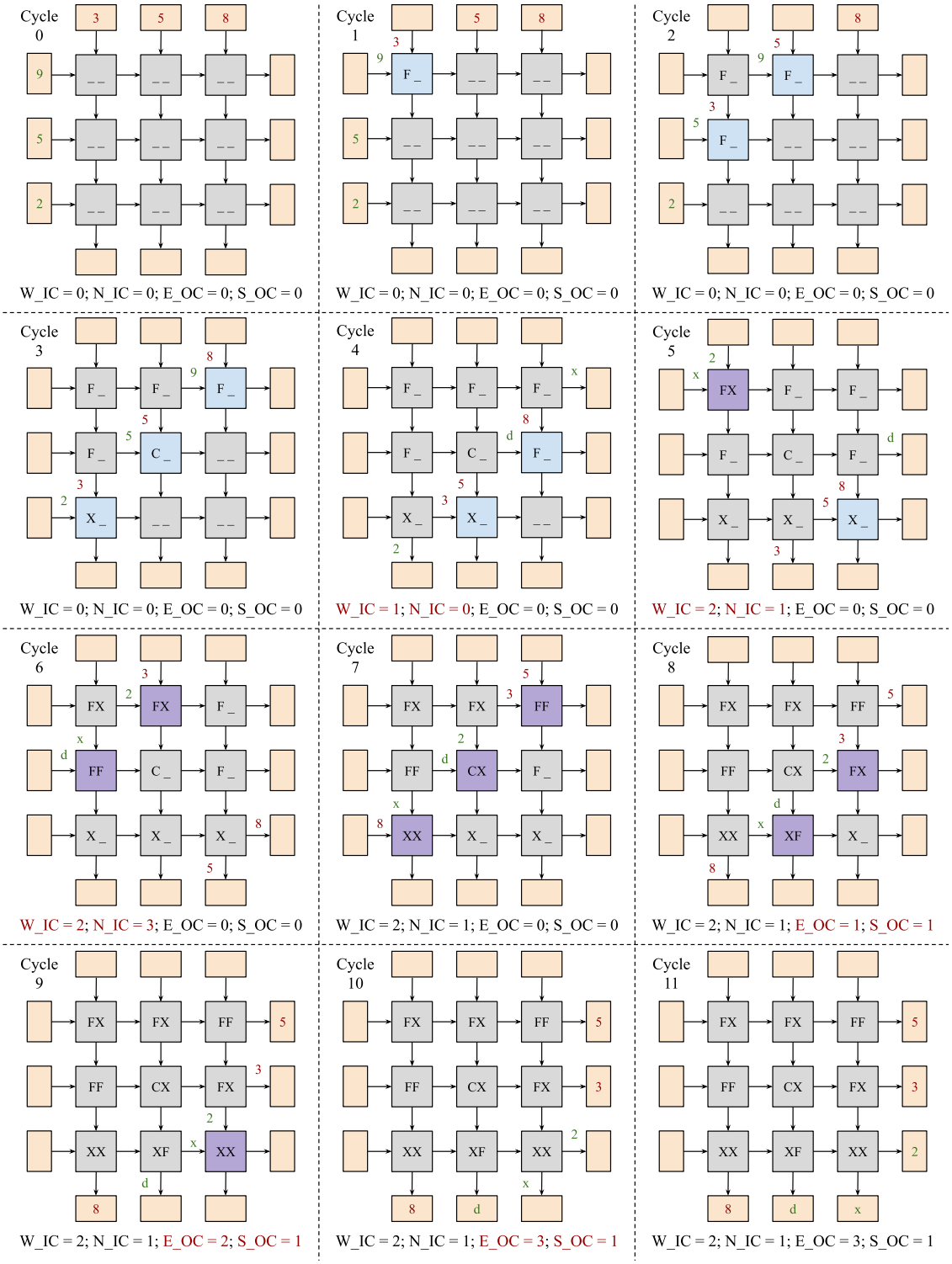
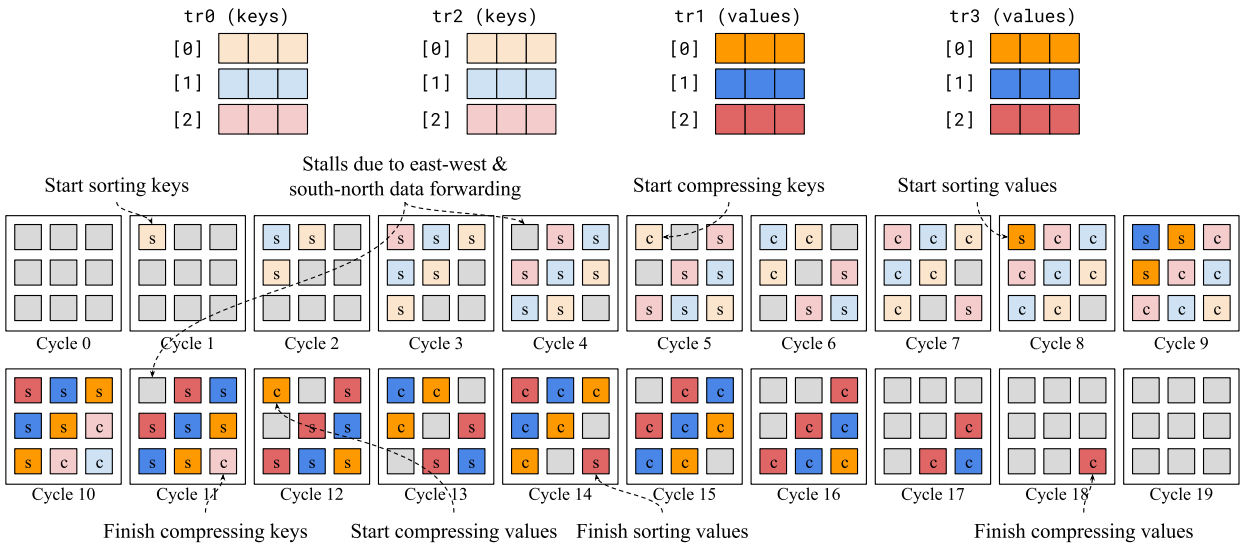


Figure 5.19: Cycle-by-Cycle Systolic Execution of `mszipk` in a 3×3 Systolic Array for Two Sorted Lists of Keys – PE states: F = forward, X = switch, C = combine; W_IC = west input counter; N_IC = north input counter; E_OC = east output counter; S_OC = south output counter; d = duplicate key that is excluded; x = unmergeable key; Counters in red indicate they are updated in corresponding cycles. PEs in gray are inactive. PEs in blue are merging keys. PEs in purple are compressing valid output keys. Keys in red come from the north input. Keys in green come from the west input. Keys in west and east sides are ordered from bottom to top. Keys in north and south sides are ordered from left to right.




```
mssortk.tt tr0, tr2, v0, v1 # sort keys of the 1st and 2nd input chunks
mssortv.tt tr1, tr3, v0, v1 # sort values of the 1st and 2nd input chunks
```

Figure 5.20: Cycle-by-Cycle Systolic Execution of Sorting Multiple Key-Value Lists in a 3×3 Systolic Array – PEs performing key-value sorting are annotated with letter `s`. PEs performing key-value compression are annotated with letter `c`. PEs in gray color are idle. Otherwise, the color of a PE refers to a set of rows in matrix registers that the PE is processing. Time in cycles progresses from left to right and top to bottom. The code snippet is extracted from the sorting code in Figure 5.16.

for each key and flipped to true when a PE detects a larger or equal key from the other input side. After the merging pass, if the merge bit is still false, the key becomes invalid and excluded from the output list.

Input and output counters are updated in the same way as in the sorting operation (e.g., in cycles 4-6 and cycles 8-10 in Figure 5.19). `W_IC` and `N_IC` count the numbers of mergeable keys from the west-side and north-side input lists respectively. `E_OC` and `S_OC` count the numbers of valid output keys in the east-side and south-side output lists respectively.

5.5.3 Merging and Sorting Multiple Pairs of Key-Value Lists

Section 5.5.1 and 5.5.2 detail the systolic executions of sorting and merging a pair of key-value lists. This section discusses how multiple pairs of key-value lists can be processed in parallel using the systolic array.

Figure 5.20 shows the cycle-by-cycle systolic execution of sorting multiple key-value lists. The cycle-by-cycle behaviors of merging multiple key-value lists are similar to the one of the sorting

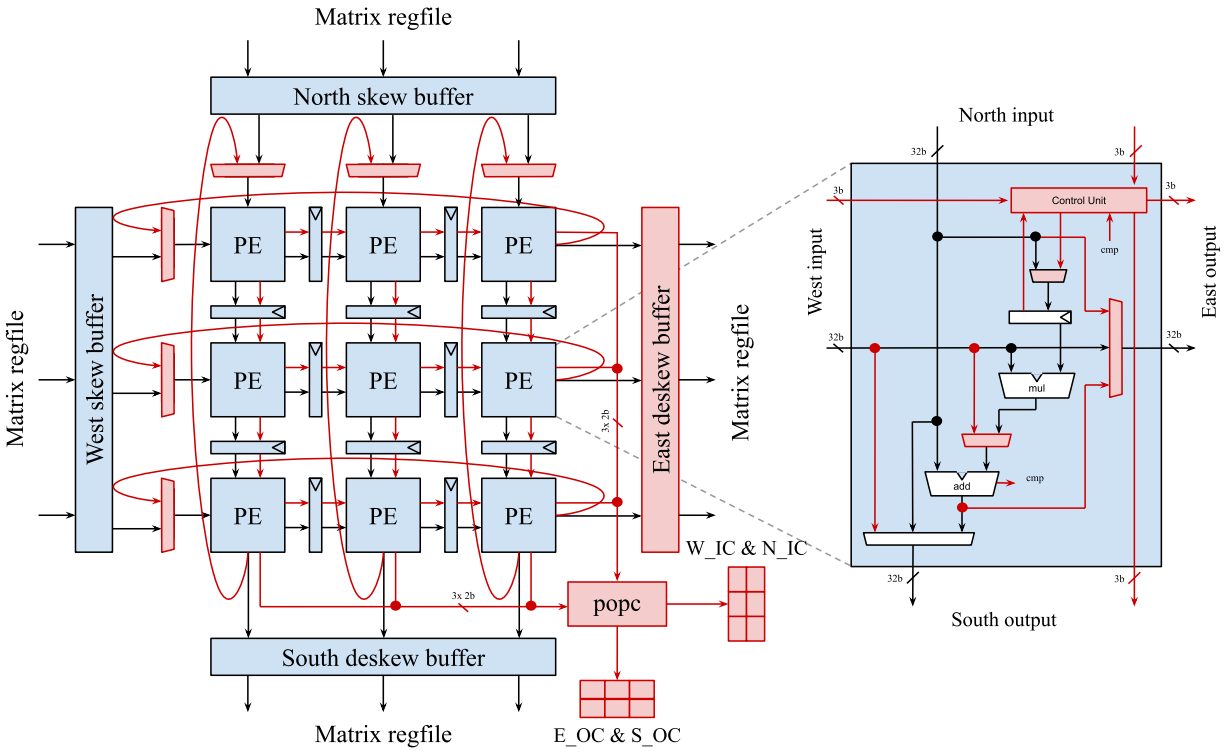


Figure 5.21: SparseZipper Systolic Array Micro-architecture – Components and wires added to support sparse computation are in red; PE = processing element; popc = population counting logic; W_IC = west input counter vector; N_IC = north input counter vector; E_OC = east output counter vector; S_OC = south output counter vector.

operation. As explained in Section 5.4.2, multiple key-value streams are mapped to multiple rows in a matrix register for parallel processing. Each row in a matrix register in Figure 5.20 corresponds to a chunk of a key-value stream. First, keys are sorted (i.e., by `mssortk`), and then values are shuffled (i.e., by `mssortv`) based on the outcome of `mssortk`.

Data elements from adjacent rows enter the systolic array in consecutive cycles to maximize the systolic array's utilization. There are one-cycle stalls in cycle 4 and cycle 11 since it takes one extra cycle to route data from the west/south output sides at the end of a sorting pass to the east/north input sides at the beginning of a compressing pass. Since `mssortk` and `mssortv` are typically executed back to back, the systolic array can schedule to start the following `mssortv` as soon as the top-left-corner PE finishes its last key-compressing operation (e.g., in cycle 8 in Figure 5.20).

5.5.4 Micro-architectural Extension to the Baseline Systolic Array

Figure 5.21 shows micro-architectural changes for sparse computation on top of the baseline systolic array specialized for dense GEMM (shown in Figure 5.1).

We add a second output port to the matrix register file to support the sorting and merging instructions as each of those instructions has two output operands. Since each physical matrix register is quite large (e.g., 1KB for a 16×16 32-bit-element matrix register), the matrix register file may consist of multiple SRAM banks, one for each physical matrix register. Therefore, adding an additional write port to the register file simply requires an extra crossbar instead of adding an extra write port to each SRAM bank which would incur significant area overheads. In order to retrieve the east-side output data from the systolic array, we add a second deskew buffer. For `mszipk` and `mszipv`, data in the west-bound input and east-bound output needs to be reversed. Therefore, simple crossbar networks are added to the west and east sides of the systolic array.

As explained in Section 5.5.2, additional control bits (i.e., source, duplicate and merge bits) are tagged along with each data element for tracking its input source and whether the element is duplicated and mergeable. Therefore, we add a three-bit control path between any two PEs on top of the existing data path. For routing data between a sorting/merging pass and a compressing pass, two loop-back paths are added to connect east and south output sides to the west and north input sides respectively. Each path is pipelined via an extra register to account for its long distance between two sides of the systolic array.

Four input and output vectors of N counters are added to track the number of valid input and output elements for N rows of matrix registers. Each counter counts up to N (i.e., the number of elements in a row), so a vector of N counters is $N \times \log_2 N$ bit wide. The population counting logic uses output control signals from the systolic array and increments corresponding input/output counters.

In each PE, we slightly modify the existing adder to compare north- and west-side input keys. An additional control unit uses the comparison outcome to make a routing decision (i.e., either forwarding, switching, or combining inputs) and route data from input to output ports by controlling the two output multiplexers. The control unit also updates the duplicate and merge bits based on the source bit and the comparison outcome. We use the same adder for adding up values for `mssortv` and `mszipv` instructions in case of combining inputs. We repurpose the weight register in each PE to store routing states (i.e., initial, forwarding, switching, and combining). Each state

CPU	• RISC-V ISA (RV64GC) • 8-way out-of-order issue • 72-entry LQ, 56-entry SQ, 96-entry IQ & 224-entry ROB • 180 physical integer, 168 physical floating-point & 128 physical 512-bit vector registers • Two 512-bit-wide SIMD execution units
Matrix Unit	• A systolic array with 16×16 processing elements (PEs) • Each PE has a single-precision multiply-accumulate (MAC) unit • 16 physical matrix registers
Caches	• L1I cache: 8-way, 32KB & 2-cycle hit latency • L1D cache: 8-way, 32KB & 2-cycle hit latency • L2 cache: 4-way, 4-bank, 256KB & 8-cycle hit latency • LLC: 8-way, 8-bank, 512KB & 8-cycle hit latency
Memory	DDR4-2400

Table 5.3: Baseline System Configuration – LQ = Load queue; SQ = Store queue; IQ = Issue queue; ROB = Reorder buffer; LLC = Last-level cache. The CPU and caches are loosely modeled after Intel Skylake CPU [int23a, Fog12]. The matrix unit is based on previous work on an implementation of Intel AMX [JQS+21] and Intel Sapphire Rapids [NMM+22].

requires two bits to encode. Each pair of rows from two input matrix registers needs to store two states for their sorting/merging and compressing passes. Therefore, for N pairs of rows, we need a total of $N \times 4$ bits for all the routing states (e.g., 64 bits in total if the hardware vector length is 16 elements).

5.6 Evaluation Methodology

In this section, we describe our simulated systems, cycle-level modeling methodology, and a matrix dataset used to evaluate the performance of SparseZipper architecture.

5.6.1 Simulated Systems

We use gem5 [BBB+11, LPAA+20, TCB18], a cycle-approximate simulator, to evaluate the performance of SparseZipper architecture in this work. Table 5.3 details the configuration of our baseline system. We use gem5's out-of-order core and configure it to loosely model Intel Skylake CPU [int23a, Fog12], an aggressive high-performance out-of-order CPU with 512-bit SIMD support for accelerating compute-intensive workloads like GEMM in high-end servers and data

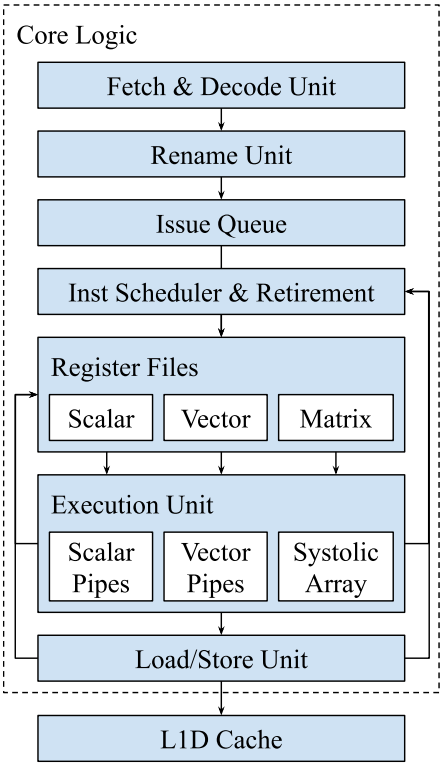


Figure 5.22: Out-Of-Order Core Pipeline with an Integrated Systolic Array for Executing Matrix Instructions

centers. We model two 512-bit-wide SIMD execution units integrated into the CPU pipeline for vector execution. Vector instructions can be speculatively executed out-of-order. The simulated cache subsystem is based on Arm AMBA 5 CHI cache model provided in gem5 [gem21].

Baseline systolic array for dense GEMM – For matrix instruction support, we model a systolic array with 16×16 PEs based on previous work on an implementation of Intel AMX [JQS+21] and Intel Saphhire Rapids [NMM+22]. Each PE consists of a double-precision multiply-accumulate (MAC) unit with a latency of four CPU cycles. There are 16 physical two-dimensional matrix registers, each capable of storing 16×16 32-bit data elements (i.e., for a total of 1KB). The baseline matrix register file supports two read ports and one write port for the `mmult.tt` instruction. Since each matrix register is quite large, a viable physical implementation of the matrix register file can include several one-read one-write SRAM blocks, one for each matrix register, and a crossbar for reading and writing row(s) of matrix registers. Similar to scalar and vector registers, matrix registers can be renamed so that matrix instructions in the base matrix ISA (i.e., `mmult.tt`, `mlse.t`, and `msse.t`) can be executed out-of-order. Constant-stride matrix load and store instructions (i.e., `mlse.t` and `msse.t`) are decomposed into row-wise unit-stride memory micro-ops that are exe-

Matrix	# Rows	NNZs	Density	Per Row					Per 16 Rows	
				Avg Inp NNZ	Inp NNZ Variation	Avg Work	Avg Out NNZ	Work / Out NNZ	Avg Work	Avg Work Variation
p2p	63K	148K	3.78E-05	2.36	1.86	8.60	8.59	1.00	0.14K	2.26
wiki	8K	104K	1.51E-03	12.50	3.16	547.52	220.70	2.48	8.76K	2.06
soc	76K	509K	8.84E-05	6.71	3.88	526.09	271.20	1.94	8.48K	1.43
ca-cm	23K	187K	3.49E-04	8.08	1.32	178.66	101.82	1.75	2.86K	1.35
ndwww	326K	930K	8.76E-06	2.85	1.95	29.42	12.63	2.33	0.78K	1.30
patents	241K	561K	9.69E-06	2.33	1.28	10.83	9.48	1.14	0.20K	1.29
ca-cs	227K	1628K	3.15E-05	7.16	1.48	164.38	72.68	2.26	2.63K	0.98
email	37K	184K	1.37E-04	5.01	1.76	163.04	89.30	1.83	2.64K	0.88
scircuit	171K	959K	3.28E-05	5.61	0.78	50.74	30.54	1.66	0.81K	0.48
bcsstk17	11K	220K	1.83E-03	20.03	0.45	445.71	56.58	7.88	7.13K	0.38
usroads	129K	331K	1.98E-05	2.56	0.31	7.18	5.45	1.32	0.11K	0.31
p3d	14K	353K	1.93E-03	26.10	0.53	870.85	218.85	3.98	13.93K	0.24
cage11	39K	560K	3.66E-04	14.32	0.31	225.13	97.59	2.31	3.60K	0.08
m133-b3	200K	800K	2.00E-05	4.00	0.00	16.00	15.90	1.01	0.26K	0.00

Table 5.4: Evaluated Datasets – p2p = p2p-Gnutella31; wiki = wiki-Vote; soc = soc-Epinions1; ca-cm = ca-CondMat; ndwww = NotreDame_www; patents = patents_main; ca-cs = coAuthorsCiteseer; email = email-Enron; p3d = poisson3Da; Avg = average; Variation = coefficient variation, ratio of the standard deviation to the mean; Inp = input; Out = output; Density = ratio of non-zero values to all values in a matrix; NNZ = number of non-zero values; Avg Out NNZ = average number of non-zero values in an output matrix row; Work = number of multiplications needed to compute one output row or one group of 16 consecutive output rows; In this work, we multiply each square matrix with itself. Table entries are sorted by the avg work variation in a descending order.

cuted in the same way as unit-stride vector memory instructions. Figure 5.22 shows the integration of the matrix register file and systolic array in the O3 CPU pipeline.

Extended systolic array for sparse GEMM – We model a non-speculative execution of stream sorting and merging instructions (e.g., `mssortk.tt` and `mszipk.tt`) to simplify the hardware implementation of special-purpose registers (e.g., input and output counter vectors as discussed in Section 5.4). Those instructions wait until they are at the ROB's head (i.e., no longer speculative) before they are issued to the systolic array for execution. Once issued, those instructions are placed into a retirement queue waiting for their execution (i.e., with no possible exception as defined by the SparseZipper ISA) to finish, and subsequent instructions can continue to commit. We model extending the matrix register file's crossbar to support the second write port. We model a latency of one CPU cycle in each PE to process one pair of input data when the PE executes the sorting and merging instructions since those instructions do not use the PE's long-latency floating-

point multipler. Similar to constant-stride matrix memory instructions, indexed matrix load/store instructions (i.e., `mlxe.t` and `msxe.t`) are broken into row-wise micro-ops that are executed by the core's load/store unit.

5.6.2 SpGEMM Implementations

Scalar SpGEMM – We evaluate two scalar row-wise implementations of SpGEMM: *scl-array* and *scl-hash* using a dense array [GMS92] and a hash table respectively for accumulating intermediate non-zero values in each output row (as discussed in Section 5.3). In *scl-hash*, we use linear probing to solve hash collision (i.e., storing value to the next available position in the hash table in case of collision). For both *scl-array* and *scl-hash*, after all intermediate non-zeros are accumulated for each output row, they are sorted by their column indices using a quick sort routine from the C++ standard library.

Vectorized ESC-based SpGEMM – We ported a vectorized implementation of SpGEMM from prior work [FC23b]. This vectorized SpGEMM implementation called *vec-radix* is based on a radix-sort-based ESC algorithm (i.e., ESC algorithm is described in Section 5.3). The *vec-radix* is vectorized using the RISC-V vector extension. The output matrix is divided into multiple blocks of consecutive rows. In *vec-radix*, there is a preprocessing step that calculates the amount of work (i.e., the number of multiplications) for each output row and groups multiple consecutive rows into one block based on a given configurable target block size. The preprocessing step is also necessary for allocating enough temporary memory space for intermediate non-zeros generated in the expansion step. A too small block size limits the amount of parallelism that can be vectorized while a too large block size can lead to thrashing the caches. In order to pick the best block size, we sweep multiple block sizes for each input matrix and report the one yielding the best performance. In the sorting phase which is typically the most time-consuming step in the ESC algorithm, *vec-radix* uses a vectorized radix sort algorithm [ZB91] to sort intermediate non-zeros by their row and then column indices. Finally, the compressing phase uses constant-stride memory instructions to combine adjacent intermediate non-zeros sharing the same row and column indices into one final non-zero value.

Merge-based SpGEMM using SparseZipper instructions – We implemented two versions of the merge-based SpGEMM (i.e., discussed in Section 5.4) using the proposed SparseZipper instruction set extension: *mtx-merge* and *mtx-merge-rsort*. Similar to *vec-radix*, both calculate the

amount of work for each output row to allocate enough temporary memory space for intermediate results in a preprocessing step. Different from *mtx-merge*, *mtx-merge-rsort* sorts row indices by their amount of work calculated in the preprocessing step so that output rows with similar amount of work can be computed together. Once all output rows are computed, they need to be re-sorted by their row indices so that non-zeros in the final output matrix are ordered by their row indices. The sorting is done using a quick sort routine from the C++ standard library. In both *mtx-merge* and *mtx-merge-rsort*, the expansion phase is vectorized using the RISC-V vector extension while the merge phase is implemented using the proposed SparseZipper instructions.

5.6.3 Matrix Datasets

We evaluate SparseZipper using a set of matrices, shown in Table 5.4, taken from SuiteSparse [DH11] across multiple domains such as road networks, scientific simulations, and social networks. This collection of matrices represents a variety of sparsity levels and patterns (e.g., variations in number of non-zeros across matrix rows). In this work, we multiply each matrix with itself, which is a common evaluation method used in prior work [SJL+20, PBP+18, ZAES21]. Table 5.4 reports the amount of work, the number of multiplications needed, for each output row (i.e., in per-row avg work column) and for each group of 16 output rows (i.e., in per-16-row avg work column). We also report the ratio of work to the number of non-zero values in an output matrix row (i.e., in work / output NNZ column). This ratio shows the degree in which duplicates in a list of intermediate non-zero values are compressed into a final list of unique non-zero values in an output matrix row.

5.7 Evaluation

In this section, we first discuss our cycle-level performance of the *mtx-merge* and *mtx-merge-rsort* in comparison to both scalar (i.e., *scl-array* and *scl-hash*) and vector (i.e., *vec-radix*) baseline implementations. We then qualitatively analyze area overheads of SparseZipper micro-architecture with respect to the baseline hardware designed for accelerating dense GEMM.

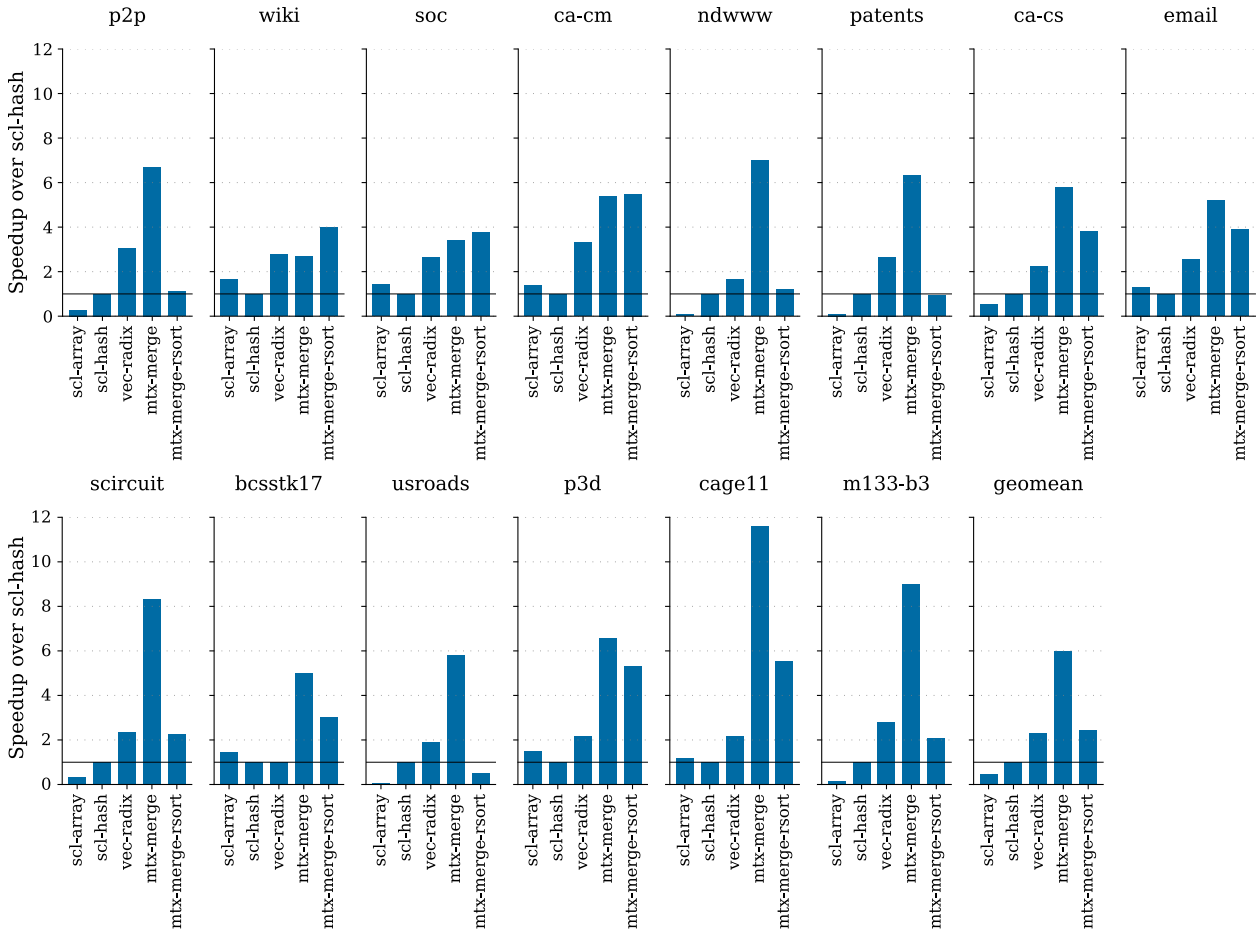


Figure 5.23: Speedup over Scalar Baseline Using Hash Table

5.7.1 Performance Evaluation

Figure 5.23 shows the relative performance of all SpGEMM implementations evaluated in this work. On average, *mtx-merge* achieves 12.13×, 5.98×, 2.61× speedup over the three baseline versions *scl-array*, *scl-hash* and *vec-radix* respectively. Figure 5.24 shows the execution time breakdown in different execution phases of the *vec-radix*, *mtx-merge*, and *mtx-merge-rsort* versions. Figure 5.25 and 5.26 show the number of accesses to L1 data cache and its hit rate respectively. Figure 5.27 shows the normalized number of dynamic `mssortk` and `mssortv` instructions in the *mtx-merge* and *mtx-merge-rsort* implementations.

Performance of the scalar baseline SpGEMM implementations – On average, *scl-hash* is 2.03× faster than *scl-array*. For matrices that have relatively sparse outputs (e.g., *p2p*, *patents*, *usroads*, and *ndwww*), using a hash table (in *scl-hash*) to accumulate sparse non-zero values is significantly more efficient than using a dense array (in *scl-array*) since each output matrix row has

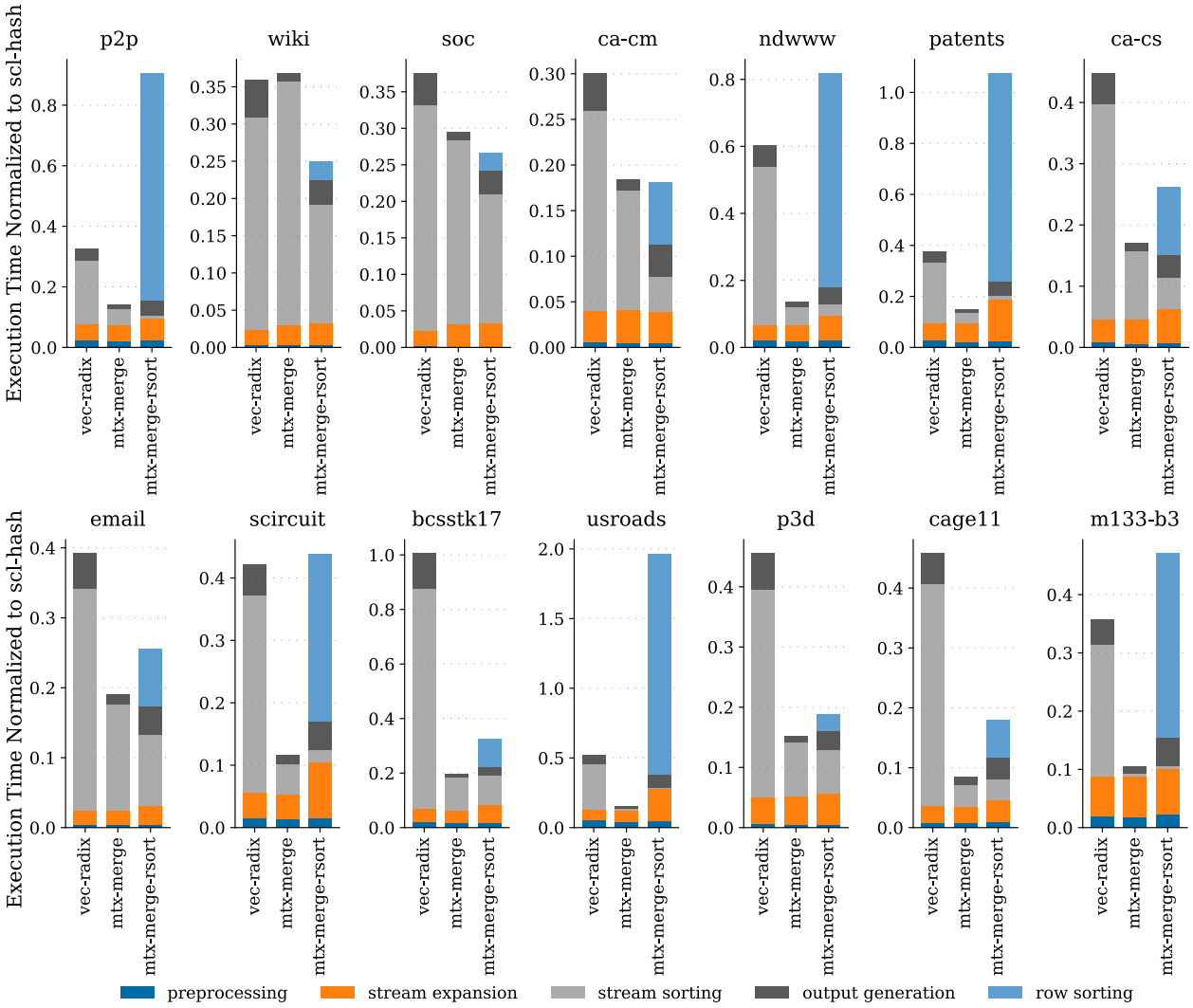


Figure 5.24: Execution Time Breakdown – The *preprocessing* phase quantifies per-output-row amount of work, divides the work across into multiple row blocks (i.e., only in `vec-radix`), and allocates temporary memory space. The *stream expansion* phase performs all multiplications and generates intermediate outputs. The *stream sorting* phase sorts and compresses the intermediate outputs (i.e., only in `mtx-merge*`).

a few non-zero values (i.e., shown in the average output NNZ column in Table 5.4). In *scl-array*, accesses to the dense array are scattered randomly, which leads to low L1 data cache hit rates (e.g., less than 20% for *ndwww*, *patents*, and *usroads* as shown in Figure 5.26). In contrast, for those sparse output matrices, *scl-hash* uses much smaller hash tables that help improve significantly L1 data cache hit rates (e.g., close to 100% for *ndwww*, *patents*, and *usroads*). A hash table's size is based on the amount of work per output matrix row (i.e., shown in the per-row average work in Table 5.4) calculated in a preprocessing step.

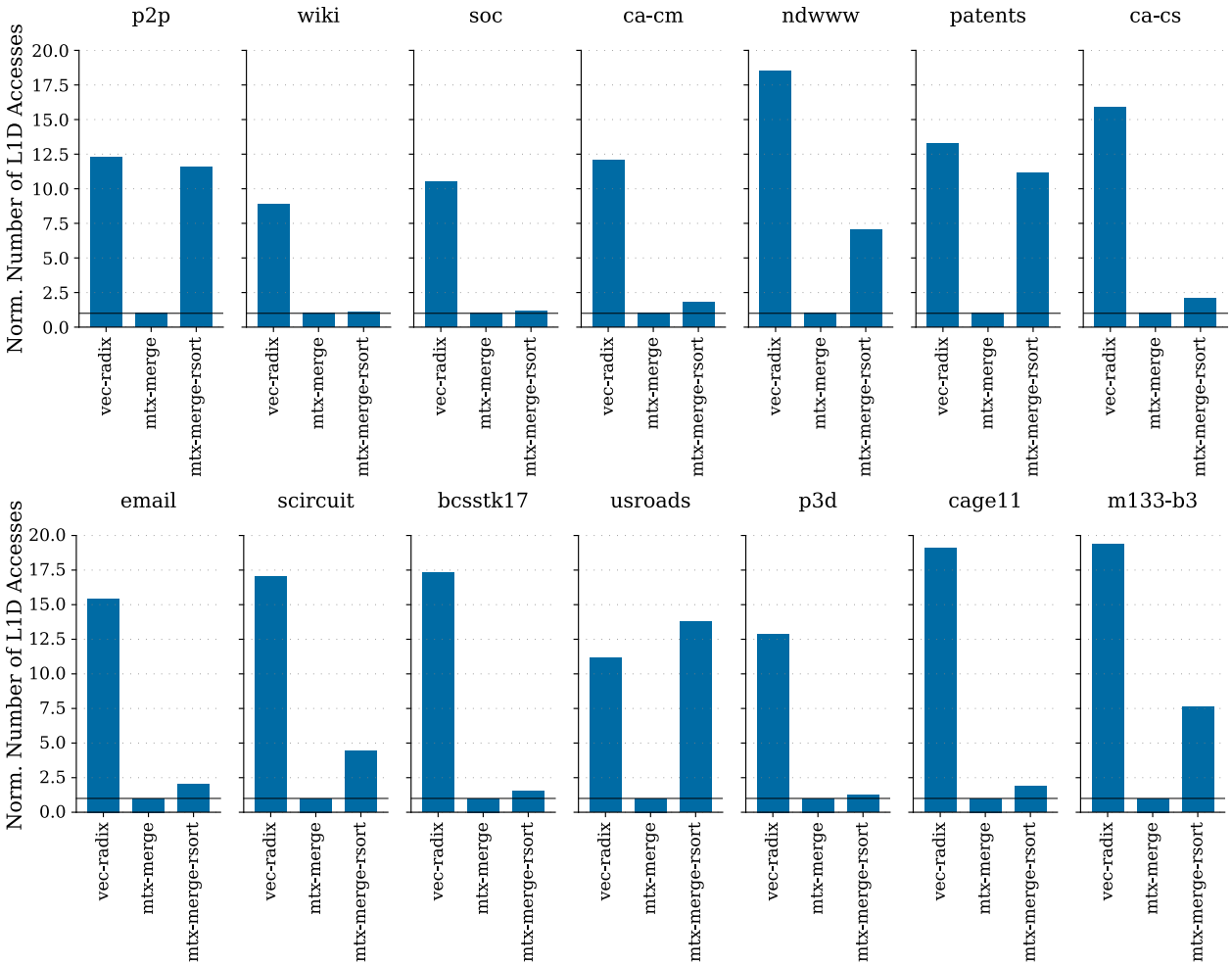


Figure 5.25: Number of L1 Data Cache Accesses

However, for matrices that have relatively dense outputs (e.g., *wiki*, *soc*, *bcsstk17*, and *p3d*), *scl-array* performs better than *scl-hash*. The main reason is that accesses to a hash table for a relatively dense output matrix cause frequent hash collisions that incur extra overheads due to linear probing. In addition, those relatively dense matrices are typically smaller in sizes, which helps improve the L1 cache hit rates.

Performance of the vectorized SpGEMM implementation – On average, *vec-radix* is 4.65× and 2.29× faster than *scl-array* and *scl-hash* respectively. Figure 5.24 shows the execution time breakdown of *vec-radix* in multiple steps. Across all matrices, the combination of stream sorting (i.e., sorting key-value tuples) and output generation (i.e., compressing adjacent tuples with duplicate keys and generating final output matrix rows) dominates the total execution time of *vec-radix*. For *bcsstk17*, *vec-radix* is slightly worse than *scl-hash*. The main reason is that *bcsstk17* has a

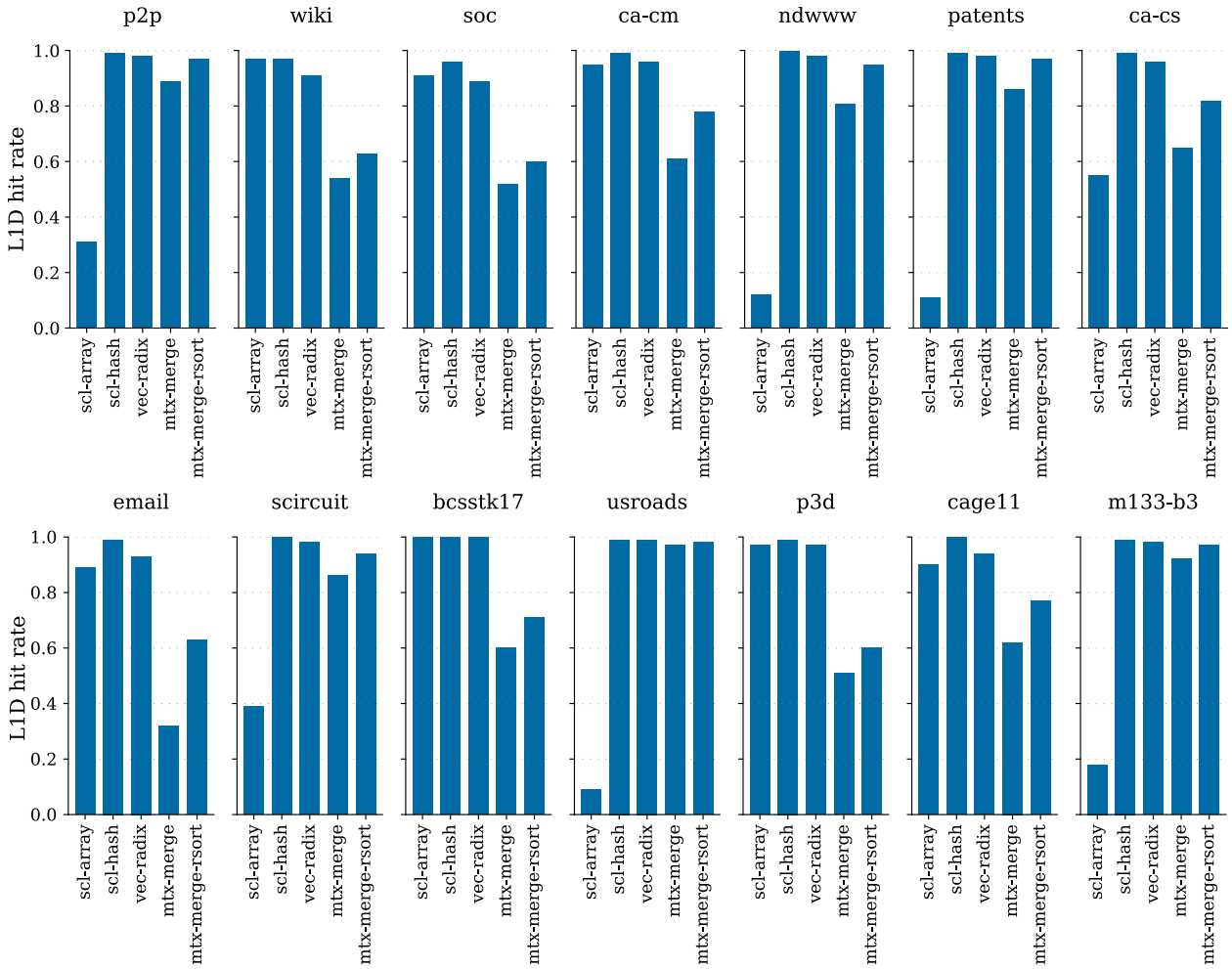


Figure 5.26: L1 Data Cache Hit Rate

high ratio of per-row work to the per-row number of output non-zeros (i.e., 7.88 as shown in Table 5.4), indicating a high number of intermediate results that share the same keys and are finally compressed into a few non-zero values. It is relatively inefficient to sort uncompressed key-value tuples with many duplicate keys in the stream sorting step.

Performance of the merge-based SpGEMM using SparseZipper instructions – The *mtx-merge* version is 2.60× faster than the *vec-radix* implementation. Figure 5.24 shows the execution time breakdown of *mtx-merge* in multiple steps. The preprocessing and stream expansion steps in *mtx-merge* are similar to the ones in *vec-radix*. The use of the newly proposed sorting and merging instructions in SparseZipper is focused on reducing the execution time of the stream sorting step which dominates the execution time of *vec-radix*. This reduction is shown in Figure 5.24 in almost all matrices except *wiki*. It is important to note that the execution time for output generation is

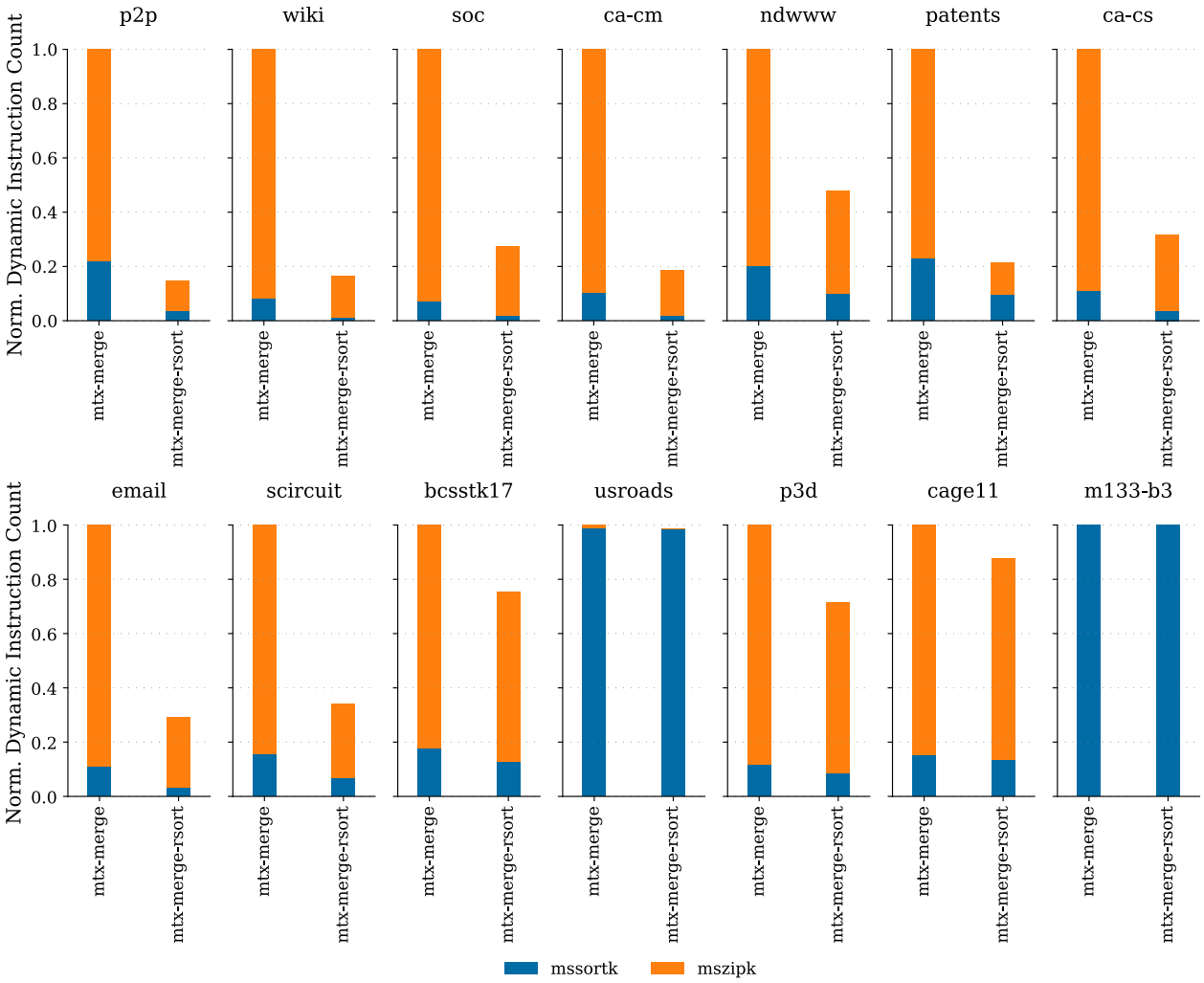


Figure 5.27: Number of Dynamic `mssortk` and `mszipk` Instructions.

decreased as well since *mtx-merge* combines tuples with duplicate keys while performing a merge sort on those tuples. This avoids a separate tuple compression step which is part of the output generation as in *vec-radix*.

One key reason for the better performance of *mtx-merge* is that the merge-based SpGEMM takes advantage of loading and storing chunks of consecutive data using the new indexed matrix load/store instructions (i.e., `mlxe` and `msxe`). Each row of a matrix register is either loaded or stored using a unit-stride vector memory micro-operation that minimizes the number of cache line accesses per chunk of data. In contrast, *vec-radix* uses a vectorized radix sort algorithm that performs both long-stride and indexed vector memory accesses that span across multiple cache lines, which results in multiple cache line accesses per vector memory instruction. Figure 5.25

shows the significant reduction in the number of L1 data cache accesses between *vec-radix* and *mtx-merge* across all evaluated matrices.

Despite efficient unit-stride vector memory accesses, *mtx-merge* has lower L1 data cache hit rates than *vec-radix* as shown in Figure 5.26 for some matrices (e.g., *wiki*, *soc*, *email*, *bcsstk17*, and *p3d*). Those matrices have relatively large numbers of intermediate results to merge per output matrix row, as shown in the per-row average work in Table 5.4. Since *mtx-merge* exploits parallelism across multiple output matrix rows (i.e., up to VLEN number of rows can be processed in parallel), *mtx-merge* has larger aggregate working set (i.e., the average work per 16 rows in Table 5.4) at a time than *vec-radix*, which causes more L1 data cache misses. However, the latency of those misses can be hidden since indexed matrix load instructions can be issued out of order as soon as the next address offsets are produced by either `mssortk` or `mszipk` instruction. For other matrices (e.g., *p2p*, *patents*, *usroads*, and *m133-b3*) that have relatively low amount of work per row, the L1 data cache hit rate in *mtx-merge* is close to the one in *vec-radix*.

Since *mtx-merge* processes multiple key-value streams (i.e., mapped to multiple adjacent matrix rows) in lock steps, any variation in the lengths of those streams could impact its performance which is determined by the processing time of the longest stream in the group. Table 5.4 shows the work variation, a ratio of the work standard deviation to the work mean, within a group of 16 consecutive matrix rows. The higher the work variation is, the more unbalanced the stream lengths of adjacent matrix rows in a group are. The relatively high work variation in *wiki* and *soc* explains the relatively low performance of *mtx-merge* compared to *vec-radix*. Although matrix *p2p* has the highest work variation, *mtx-merge* performs well for this matrix since the average per-row work is low. This low per-row amount of work minimizes the performance impact of high work variation since it takes, on average, one iteration to finish processing one key-value stream in *p2p*.

To further demonstrate the performance impact of high work variation, we sort all matrix rows by the per-row amount of work in *mtx-merge-rsort*. It is important to note that we do not actually shuffle an input matrix's data but simply sort row indices. Rows with similar amount of work are then processed together. At the end, it is necessary to shuffle the output matrix's data based on row indices so that the final output data are sorted by their row indices. Figure 5.24 shows the execution breakdown of *mtx-merge-rsort*. By processing rows with similar amount of work together, the stream sorting time in *mtx-merge-rsort* is significantly reduced for matrices that have high work variation (e.g., *wiki*, *soc*, *ndww*, and *ca-cm*). Figure 5.27 shows the reduction in dy-

Component	Area (k μm^2)	Baseline	SparseZipper
Baseline PE (with a 32-bit MAC unit)	0.45	× 256	
SparseZipper PE (with a 32-bit MAC unit)	0.51		× 256
Skew buffer (16-lane)	3.16	× 2	× 2
Deskew buffer (16-lane)	3.16	× 1	× 2
Matrix register (16 × 512b)	0.96	× 16	× 16
Popc logic	0.45		× 1
Total		140.16	158.00
SparseZipper vs. baseline overhead			**12.72%**

Table 5.5: Post-synthesis Area Estimates of SparseZipper Components – PE = processing element; MAC = multiply-accumulate; The estimated area numbers are for a 16×16 systolic array (256 PEs in total) and 512-bit vector length.

namic instruction counts of `mssortk` and `mszipk` across matrices with high work variation. This reduction correlates to less number of iterations required to sort and merge key-value streams due to more balanced work across rows processed in a group. For *cage11*, *mtx-merge-rsort* results in a minimal reduction in the stream sorting time since it has low work variation. For *usroads* and *m133-b3*, since their average amount of work per row is less than the vector length (i.e., 16), *mtx-merge* and *mtx-merge-rsort* finish sorting each stream in one iteration on average (i.e., only a few dynamic `mszipk` instructions).

The row sorting and output data shuffling cause significant overheads in *mtx-merge-rsort*. First, row indices are sorted by a serial quick-sort routine provided in the standard C++ library, which explains its high execution time. Future work may extend SparseZipper to include instructions that are similar to the stream merging and sorting without combining key-value tuples with duplicate keys for accelerating a standard merge-sort routine that could potentially lower the row sorting overhead. Second, processing rows in an order different from how their data are laid out in memory causes a slight increase in the stream expansion time (e.g., in *patents* and *scircuit*) due to poor spatial locality between rows.

5.7.2 Area Evaluation

Methodology – We use a post-synthesis component-level area modeling methodology to evaluate area overheads of exta hardware added to a baseline 16×16 systolic array to implement SparseZipper. We implement area-significant components of the systolic array in RTL. Each PE

includes a 32-bit single-precision floating-point multiply-accumulate unit. We model extra control logic added to a PE to support for the stream sorting and merging operations. Each skew/deskew buffer is used to stagger input and output data coming in and going out of the systolic array. We model each buffer as an array of 16 shift registers with their sizes ranging from one to 16 entries. For this evaluation, we assume 16 rows, each is 512-bit wide (i.e., 16× 32-bit data elements), in a matrix register and a total of 16 physical matrix registers. Regarding the popc logic, we implemented an array of 16 five-bit counters (i.e., counting up to 16) and a list of counter vector registers (i.e., 16 × 5 bits per register).

Aver overheads of SparseZipper – Table 5.5 shows the detailed area comparison between SparseZipper and the baseline using our first-order component-based area modeling methodology. In overall, a 16×16 SparseZipper implementation adds around 12.72% area overhead compared to the baseline implementation with the same systolic array's dimensions. When considering a complete system including an out-of-order core, its vector engine and vector register file, its caches, and memory, we expect the percentage of extra area added to the baseline systolic array for supporting SparseZipper to be much lower.

5.8 Related Work

Extending systolic arrays to support sparse computation – There have been various proposals to extend a systolic-array-based micro-architecture to support sparse matrix computation. NVIDIA introduces sparse tensor cores that accelerate multiplying a sparse matrix that has a specific sparsity pattern with a dense matrix [PSR21, CGG+21a]. The sparse input matrix must have two zeros out of four contiguous elements (i.e., 50% density) so that non-zeros can be compressed using light-weight metadata tracking their positions in a block of four contiguous elements. VEGETA further extends the support for more flexible sparsity patterns (i.e., N:4 and N:M structures) [JDB+23]. VEGETA stores a compressed input matrix with its metadata in a systolic array and streams the other input matrix stored in a dense format into the array in a skewed manner. Each PE needs to perform index matching operations to either skip or perform a multiplication. The output matrix is stored in a dense format. Similar to SparseZipper, VEGETA targets fine-grain GEMM accelerators using matrix instruction set extensions (e.g., Intel AMX). Unlike NVIDIA's sparse tensor core and VEGETA, SparseZipper targets sparse-matrix and sparse-matrix multipli-

Figure 5.28: Normalized Number of Multiplications Performed in Different Sparsity Compression Levels– All numbers are normalized to the number of multiplications needed when representing both input matrices in a sparse format. In dense-dense, both matrices are stored in a dense format. In sparse(1:4)-dense and sparse(1:16)-dense, one matrix is stored in a sparse format with certain compression levels (i.e., one non-zero in a block of four or 16 contiguous elements), and the other is stored in a dense format.

cation with both matrices being highly sparse (i.e., much less than 1% density) and unstructured (i.e., no specific sparsity structure is assumed in the design of SparseZipper).

Figure 5.28 shows the computation demand (i.e., in the number of multiplications) of multiplying two matrices stored in different sparsity compression levels. SparseZipper stores both input matrices in a sparse format, which is the baseline in the figure. Sparse(1:4)-dense and sparse(1:16)-dense configurations represent two compression levels that can be supported in VEGETA. Dense-dense configuration represents a conventional systolic array that processes matrices in a dense format. Since our target matrix datasets are high sparse, the amount of computation needed to multiply a matrix stored in a sparse format with another matrix stored in a dense format is multiple orders of magnitude higher than the case in which both matrices are stored in a sparse format. SparseZipper complements the NVIDIA's sparse tensor core and VEGETA by expanding the spectrum of sparsity level and structure supports towards highly sparse and unstructured matrices.

Sparse-TPU [HPA+20] proposed an offline column packing algorithm that merges sparse columns to minimize the number of zeros mapped to a systolic array. The proposed systolic array supports conditional execution to skip multiplications for values that do not have matching indices. How-

ever, Sparse-TPU explicitly targets sparse-matrix dense-vector multiplication kernel, not sparse-matrix sparse-matrix multiplication as in SparseZipper. STA [LWM20] proposed a new block-sparse format that targets matrices with an upper limit on the number of non-zeros in a block of elements. In contrast, SparseZipper does not make any assumption on the sparsity structure of input matrices.

Software-only proposal to accelerate sparse GEMM on existing systolic arrays – Guo et al. proposed a software-only tiling optimization for DNN-specific sparse GEMM on existing systolic-array-based dense GEMM accelerators [GHL+20]. Their pruning algorithm enforces a particular tile-wise sparsity pattern on pruned DNN models so that dense tiles can be mapped directly to an underlying systolic array without any hardware support. However, this pruning algorithm is specific to DNN and only works for sparse matrices generated from pruned DNN models. In contrast, SparseZipper targets more general sparse matrices from various domains such as graph analytics that may not have a particular sparsity pattern.

Coarse-grain dense GEMM accelerators – The ever-growing importance of GEMM performance and efficiency in emerging workloads motivates the needs for coarse-grain accelerators for dense GEMM. Google includes a large systolic array of multiply-accumulate units (MAC) in its Tensor Processing Unit (TPU), a co-processor next to a general-purpose CPU, for accelerating training and inference kernels [JYP+17, Tie20, JYK+20, JKL+23a]. Other examples of coarse-grain dense GEMM accelerators include Eyeriss [CKES16] and Amazon AWS neuron core [aws23]. However, those accelerators are highly inefficient in supporting sparse GEMM due to their rigid structure of underlying systolic array, lack of sparse format support, and inability to skip ineffectual multiplications.

Coarse-grain sparse GEMM accelerators – Previous work has proposed various coarse-grain accelerators mainly based on three different dataflows of implementing sparse GEMM: inner product, outer product, and row-wise product. OuterSPACE implements the outer product dataflow using tiles of processing elements to perform outer products for pairs of sparse vectors (i.e., each pair includes a column of the first matrix and a row of the second matrix) and to merge partial output matrices [PBP+18]. MatRaptor implements the row-wise dataflow by multiplying each non-zero element in the first matrix with a corresponding row in the second matrix [SJL+20]. Each group of processing elements in MatRaptor is mapped to a set of output rows, and they perform multiplications followed by merging partial results. SIGMA is an inner-product-based accelerator for sparse

GEMM in deep learning applications [QSK+20]. It implements a flexible array of dot-product engines consisting of multipliers, adders, a distribution network, and a reduction network. There are several other accelerators specialized for SMMP such as SpArch [ZWHD20], Sextans [SCS+22], Extensor [HAMP+19], and Gamma [ZAES21]. Different from those decoupled SpGEMM accelerators, SparseZipper takes a more programmable approach that extends an existing matrix ISA for supporting SpGEMM. In addition, instead of adding dedicated hardware specialized just for SpGEMM, SparseZipper micro-architecture can be built on top of existing systolic arrays specialized for dense GEMM without adding much hardware to support SpGEMM.

Fine-grain dense GEMM accelerators – Arm recently released its Scalable Matrix Extension (SME) that introduces a new instruction performing an outer product of two vectors and accumulating its results into a new two-dimensional accumulator register state [arm23]. IBM took a similar approach in its Matrix-Multiple Assist (MMA) extension for the Power ISA [ibm23]. Intel introduced a new Advanced Matrix Extension (AMX) that adds several two-dimensional matrix register states called tile registers and a new matrix-matrix multiply instruction performing a matrix multiplication on two input tile registers [int23b, NMM+22]. The RISC-V community is also working on a matrix extension proposal [ris23] that is similar to Intel AMX's approach. Intel recently launched its Sapphire Rapids architecture that includes a matrix engine specialized for dense GEMM. RASA is an academic proposal for integrating a systolic array into a general-purpose out-of-order processor for accelerating dense GEMM [JQS+21]. SparseZipper proposes to extend such matrix abstractions and micro-architectures to support sparse GEMM without introducing significant area overheads.

Fine-grain sparse GEMM accelerators – SparseCore is a proposed ISA extension for sparse tensor computation [RCYQ22]. SparseCore introduces new stream registers, stream load/store instructions, and stream merging/intersecting instructions for accelerating sparse tensor computation. The merging instructions in SparseZipper were inspired by SparseCore. Unlike SparseCore, instead of adding large stream registers, SparseZipper leverages existing matrix states (i.e., matrix registers) designed for dense GEMM to store parts of key-value streams. In addition, SparseZipper does not add dedicated sparse processing units for merging and intersecting key-value streams. Instead, SparseZipper minimally modifies an existing systolic array designed for dense GEMM to support merging key-value streams for SpGEMM.

5.9 Conclusion

This chapter has demonstrated performance benefits of minimally extending a matrix ISA and a systolic array micro-architecture originally designed for dense GEMM to support sparse GEMM. The SparseZipper ISA introduces new stream sorting and merging instructions to enable sorting and merging key-value streams representing sparse vectors in the merge-based sparse GEMM algorithm with matrices represented in CSR or CSC formats. SparseZipper leverages existing matrix registers to store parts of key-value streams and minimally extends a systolic array to support the stream sorting and merging instructions. Future research can explore opportunities to add instructions specialized for certain sparsity structures (e.g., structured sparsity in deep learning workloads) and sparse formats other than CSR and CSC (e.g., COO and block-compress-sparse row). Regarding supporting the COO format, one potential extension could be to support sorting and merging tuples with two keys (i.e., row and column indices) and one value in the same systolic array. Tuples are sorted by row indices and then by column indices. Another line of future research is to improve the utilization of a systolic array when it performs the sorting or merging instructions.

CHAPTER 6
CONCLUSION

In this thesis, I discussed and explored the evolutionary specialization that supports multiple types of specialization in the same hardware by gradually evolving an existing micro-architecture specialized for one programming pattern to support other patterns. I, then, motivated the evolutionary specialization through two novel architectures: big.VLITTLE and SparseZipper. In big.VLITTLE architectures, a multi-little-core system specialized for single-program multiple-data (SPMD) pattern is reconfigured to support single-instruction multiple-data (SIMD) pattern with minimal area overhead to the original hardware. In SparseZipper architectures, a programmable systolic-array-based micro-architecture designed for accelerating dense matrix-matrix multiplication (GEMM) is repurposed so that it can efficiently perform GEMM on sparse matrices as well. In this chapter, I summarize my thesis contributions, discuss lessons learned regarding the evolutionary specialization, and present some thoughts on potential future research directions based on the insights in this thesis.

6.1 Thesis Summary and Contributions

The thesis began by discussing the trend towards specialization in response to the slowdown of Moore's Law and the end of Dennard Scaling. There are two conventional approaches to supporting multiple kinds of specialization in an SoC: heterogeneous and unified specialization. First, the heterogeneous specialization refers to composing multiple singularly specialized accelerators without impacting their abstractions and micro-architectures. By retaining the same optimal abstractions and micro-architectures for individual program patterns, this approach can achieve the highest performance for applications with the supported patterns. However, given certain area, power, and budget constraints, there is limited room for the number of specialized hardware units possibly integrated into an SoC. Second, the unified specialization refers to using a single unified abstraction and micro-architecture that are generic enough to support multiple program patterns. Having the same abstraction often enables a unified software stack for multiple programs. Sharing the same micro-architecture enables hardware reuse when executing programs with different patterns, which increases hardware utilization. However, due to adopting a sub-optimal abstraction for a program pattern, the unified specialization leaves out opportunities for software to convey po-

tential pattern-specific optimizations to a corresponding micro-architecture. This thesis explores another approach called evolutionary specialization. This approach refers to starting from an optimal abstraction and micro-architecture for one program pattern and gradually adding a minimal set of hardware changes to an existing micro-architecture to support additional program patterns without impacting their optimal abstractions. Since both abstraction and micro-architecture for the starting program pattern are unchanged, the evolutionary specialization retains the optimal performance and efficiency for that pattern as in the singular specialization. For the additional program patterns, the evolutionary specialization keeps the same optimal abstractions as supported in their singularly specialized accelerators so that all program properties specific to those patterns can be conveyed to and potentially exploited in hardware.

I then presented CIFER, the first academic open-source multicore-eFPGA SoC composed of multiple Linux-capable cores for running general-purpose workloads, TinyCore tiles for exploiting massive thread-level parallelism, and eFPGA for application-specific acceleration. The chip was fabricated on a GlobalFoundries 12nm FinFET technology node. CIFER features a heterogeneous cache coherence implementation that enables seamless on-chip communications across different compute tiles. For workloads with massive thread-level parallelism, our evaluation results show that the TinyCore clusters improve their performance and energy efficiency by up to 7.95× and 7.75× respectively compared to a single general-purpose Ariane core. For workloads that are well-suited for mapping to the eFPGA, we show up to 9.29× and 10.62× performance and energy efficiency improvement respectively. CIFER demonstrates opportunities for exploiting different kinds of specializations in an SoC, leveraging open-source projects and IPs to ease the process of building a complex SoC, and cooperating executions of heterogeneous hardware units seamlessly in a single system. However, CIFER also presents challenges in adopting the heterogeneous specialization regarding maximizing utilization of on-chip resources, integrating heterogeneous components into a single system, and verifying the system as a whole.

Using CIFER as a motivation for the evolutionary specialization, I then presented big.VLITTLE architectures that support both the SPMD and SIMD patterns in the same system. The big.VLITTLE architectures offer a compelling high-performance and area-efficient solution to accelerating data-parallel workloads in heterogeneous multi-core mobile systems. The reconfigurability of big.VLITTLE architectures resolves the fundamental tension between performance and area in implementing next-generation vector architectures, which opens up opportunities to provide the performance

level of decoupled vector engines for data-parallel workloads in small mobile systems without sacrificing either valuable silicon area on chips or performance of task-parallel workloads.

Lastly, I described SparseZipper that extends a matrix ISA and a systolic array micro-architecture originally specialized for computing dense GEMM to support sparse GEMM. The SparseZipper ISA introduces new stream sorting and merging instructions to enable sorting and merging key-value streams representing sparse vectors in sparse GEMM computation. SparseZipper leverages existing matrix registers to store parts of key-value streams and minimally extends a systolic array to support the stream sorting and merging instructions. Our performance evaluations show SparseZipper achieves 5.98× and 2.61× speedup over a scalar hash-based implementation of sparse GEMM and a vectorized sparse GEMM version respectively.

The primary contributions of this thesis are:

- an exploration of the evolutionary specialization that supports multiple types of hardware specialization by gradually evolving a micro-architecture specialized for one kind of specialization to support other kinds of specialization without impacting their programming abstractions;
- a novel big.VLITTLE architecture, an example of the evolutionary specialization, supporting on-demand data-parallel acceleration via a vector abstraction in a multi-core micro-architecture designed for the SPMD pattern; and
- a novel SparseZipper architecture, another example of the evolutionary specialization, efficiently supporting both dense and sparse GEMM computation by gradually extending an existing matrix ISA and micro-architecture specialized for the dense GEMM pattern to support the sparse GEMM pattern.

6.2 Discussions on Evolutionary Specialization

In this section, I discuss several aspects of the evolutionary specialization approach from my experience in designing and implementing big.VLITTLE and SparseZipper architectures.

How to choose a combination of program patterns? This is arguably the most important question when computer architects consider the evolutionary specialization approach. While the

answer to this question largely depends on certain situations, there are aspects in existing pattern-specific singular accelerators that may help us figure out whether a single evolutionary accelerator for those patterns would be beneficial. Firstly, are there enough similarities across abstractions and micro-architectures of those singular accelerators? In big.VLITTLE architectures, a set of scalar registers from multiple threads could be collectively viewed as a vector register file. Each scalar register file from a thread corresponds to a slice of a vector register file per virtual lane. A vector arithmetic instruction could be viewed as a collection of multiple scalar instructions that execute together. In SparseZipper, similarities between a matrix register and a collection of key-value streams are not as obvious and heavily dependent on how the underlying systolic array is repurposed to perform key-value sorting and merging operations. For example, the systolic execution of key merging and sorting operations requires independent streams of keys to be mapped to multiple rows in a matrix register so that those rows can be processed by the systolic array in parallel. This specific execution style inhibits mapping one single stream of keys to an entire matrix register. If abstractions and micro-architectures of singular accelerators are fundamentally different, the evolutionary specialization is likely to be a wrong approach since hardware can hardly be shared between those accelerators. Secondly, program patterns that could benefit from the evolutionary specialization may come from the same application domain. For example, in SparseZipper, both dense and sparse GEMMs perform the same matrix-matrix multiplication operation on two matrices although their algorithms and data structures are drastically different. This similarity in the use of program patterns makes the evolutionary specialization an attractive solution. Depending on certain datasets, the hardware can flexibly support either pattern without adding too much area overhead.

Overheads to the starting micro-architecture – One key design goal of the evolutionary specialization approach is to minimally impact the starting micro-architecture in performance, energy efficiency, and area. For example, in big.VLITTLE architectures, supporting cross-element instructions requires a careful consideration of how underlying little cores could communicate without adding too much overhead. In the end, big.VLITTLE architectures adopted a light-weight uni-directional ring network to minimize area and timing overhead to the existing cluster of little cores. This design decision trades off performance of workloads with the SIMD pattern for the minimal overhead. Another example is the support for compressing duplicate keys and values in SparseZipper. One naive solution is to add extra hardware logics that compress keys and

values based on a valid mask to the east and south bound outputs of the systolic array. However, this solution would require a shuffling network to move data across lanes in each output, which could potentially introduce high area overhead to the baseline systolic array. In SparseZipper, uncompressed keys and values are passed through the systolic array one more time, which incurs higher latency in merging and sorting operations without adding too much area overhead. At one extreme, if enough hardware is added to maximize performance of additional patterns, an evolutionary accelerator could become a heterogeneous system of multiple singular accelerators with their own hardware. A thorough design space exploration is necessary to understand such performance/area/energy trade-offs in the evolutionary specialization approach.

Potentials and limitations of evolutionary specialization – Evolutionary specialization could be a compelling solution when considering patterns sharing similar architectural features in an SoC that has a tight area budget (e.g., big.VLITTLE architectures targeting mobile SoCs). In addition, evolutionary specialization could be an attractive solution to effectively amortize area overhead of a large singular accelerator by maximizing its utilization. For example, a 16×16 array of PEs, each including a floating-point multiply-accumulate unit, is significant in terms of area with respect to its host processor. SparseZipper extends the array's usability for a broader set of matrices, which amortizes this high area cost of the original systolic array. However, in general, evolutionary designs are relatively more complex than simply composing existing singular designs as in the heterogeneous specialization. This complexity may impact the software stack as well. For example, in big.VLITTLE architectures, a reconfiguration of a multi-little-core cluster requires some heavy-weight system-level operations such as making those cores not available for software threads and switching them into the vector mode. In the end, the evolutionary specialization is one tool that computer architects should consider in the era of hardware specialization.

6.3 Future Work

In this section, I discuss some potential future research directions based on insights provided in this thesis.

Decoupling executions of little cores in big.VLITTLE in the vector mode – The work presented in Chapter 4 conservatively couples multiple little cores as vector lanes by letting them execute vector instructions in lock steps. A little core in a group needs to wait for other cores be-

fore it can start executing the next vector micro-operation. This coupling helps reduce the hardware complexity by having less control logics to sync up multiple cores. However, this coupling may leave out opportunities to let some cores run ahead of others in the case of control and memory access divergence. A limited decoupling of little cores could be beneficial to the performance of big.VLITTLE in executing irregular data-parallel workloads.

Exploring performance benefits of decoupling a SparseZipper engine from its host processor – Chapter 5 assumes a systolic array integrated in an out-of-order processor and sharing the same cache hierarchy. Since sparse GEMM is memory bound, matrix load and store instructions are limited by the bandwidth of the core's leve-one cache. Future work may consider a decoupled SparseZipper engine that is attached to level-two or last-level caches for higher memory bandwidth. A decoupled design may come with certain challenges such as supporting exceptions and address translations for matrix load/store instructions.

Sharing a SparseZipper engine between multiple cores – Since the engine is relatively large, sharing it between multiple cores can amortize its area overhead especially when those cores do not perform GEMM computation most of the time. Some interesting research questions regarding this sharing include: (1) at what granularity (e.g., kernel, instruction levels) the engine should be shared between cores, (2) how to support multiple contexts without simply adding more matrix registers which are area-expensive, and (3) whether spatial sharing, in which multiple threads use subsets of PEs at the same time, would be beneficial.

Supporting other sparse matrix formats and high-dimensional tensor operations – The SparseZipper architecture presented in this thesis supports CSR/CSC sparse matrix format. Future work may further extend the baseline systolic array to support other sparse matrix formats such as COO and block compressed sparse row (BCSR) and data flows based on those formats. In addition to matrix computation, the proposed sorting and merging operations in SparseZipper could be extended to support high-dimension tensor operations with more than two dimensions. In high-dimension tensor operations, a key-value tuple may include several keys, one for index in each dimension. The proposed SparseZipper engine could be extended to support sorting and merging key-value streams with more than one key per tuple.

BIBLIOGRAPHY

[AAB+16] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[Abt07] Abts, Dennis and Bataineh, Abdulla and Scott, Steve and Faanes, Greg and Schwarzmeier, Jim and Lundberg, Eric and Johnson, Tim and Bye, Mike and Schwoerer, Gerald. The Cray BlackWidow: A Highly Scalable Vector Multiprocessor. *ACM/IEEE Conference on Supercomputing (SC)*, Nov 2007.

[AFW16] P. N. Q. Anh, R. Fan, and Y. Wen. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. *Int'l Symp. on Supercomputing (ICS)*, Jun 2016.

[AP14] K. Asanovic and D. Patterson. Instruction Sets Should Be Free: The Case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[app20] Apple Unleashes M1. Online Webpage, 2020. `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`.

[arm23] Introducing the Scalable Matrix Extension for the Armv9-A Architecture. Online Webpage, 2021 (accessed Apr 2023). `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/scalable-matrix-extension-armv9-a-architecture`.

[AS22] N. Adit and A. Sampson. Performance Left on the Table: An Evaluation of Compiler Auto-Vectorization for RISC-V. *IEEE Micro*, 2022.

[Asa98] K. Asanović. *Vector Microprocessors*. Ph.D. Thesis, EECS Department, University of California, Berkeley, 1998.

[aws23] NeuronCore-v2 Architecture. AWS Neuron Technical Reference Manual, 2023. `https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/neuron-core-v2.html#neuroncores-v2-arch`.

[BAA08] C. Batten, H. Aoki, and K. Asanović. The Case for Malleable Stream Architectures. *Workshop on Streaming Systems (WSS)*, Nov 2008.

[BAPS21] P. Bedoukian, N. Adit, E. Peguero, and A. Sampson. Software-Defined Vector Processing on Manycore Fabrics. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2021.

[Bat10] C. Batten. *Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators*. Ph.D. Thesis, MIT, 2010.

[BBB+11] N. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[BDVJ03] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research*, 13, 2003.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2008.

[BLS+20] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, et al. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2020.

[BMF+16] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, et al. OpenPiton: An Open Source Manycore Research Framework. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2016.

[BVWH14] U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter. Sparse Matrix Multiplication: The Distributed Block-Compressed Sparse Row Library. *Parallel Computing*, 40(5-6):47–58, 2014.

[CBM+09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, , and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2009.

[CDM+18] H. Chen, Y. Dai, H. Meng, Y. Chen, and T. Li. Understanding the Characteristics of Mobile Augmented Reality Applications. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2018.

[CGG+21a] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky. Nvidia a100 Tensor Core GPU: Performance and Innovation. *IEEE Micro*, 41(2):29–35, 2021.

[CGG+21b] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky. Nvidia A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro*, 41(2):29–35, 2021.

[CGM+96] A. Canning, G. Galli, F. Mauri, A. De Vita, and R. Car. O(N) Tight-binding Molecular Dynamics on Massively Parallel Computers: an Orbital Decomposition Approach. *Computer Physics Communications*, 94(2-3):89–102, 1996.

[CHK+21] A. Cabrera, S. Hitefield, J. Kim, S. Lee, N. R. Miniskar, and J. S. Vetter. Toward Performance Portable Programming for Heterogeneous Systems on a Chip: A Case Study with Qualcomm Snapdragon SoC. *IEEE Conf. on High Performance Extreme Computing (HPEC)*, Sep 2021.

[CHM08] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2008.

[CKES16] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2016.

[CLG+23] T.-J. Chang, A. Li, F. Gao, T. Ta, G. Tziantzioulis, Y. Ou, M. Wang, J. Tu, K. Xu, P. J. Jackson, et al. CIFER: A 12nm, 16mm 2, 22-Core SoC with a 1541 LUT6/mm 2 1.92 MOPS/LUT, Fully Synthesizable, CacheCoherent, Embedded FPGA. *Custom Integrated Circuits Conf. (CICC)*, Apr 2023.

[CLY+18] Y. Chen, K. Li, W. Yang, G. Xiao, X. Xie, and T. Li. Performance-Aware Model for Sparse Matrix-Matrix Multiplication on the Sunway Taihulight Supercomputer. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 30(4):923–938, 2018.

[CSZ+20] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI. *TVLSI*, 28(2):530–543, 2020.

[CW08] R. Collobert and J. Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. *Int'l Conference on Machine Learning (ICML)*, Jul 2008.

[CWS+14] L. Codrescu, A. Willie, S.Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro*, 34(2):34–43, Mar/Apr 2014.

[CXL+20] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi. Xuantie-910: A Commercial Multi-core 12-stage Pipeline Out-of-order 64-bit High Performance RISC-V Processor with Vector Extension: Industrial Product. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2020.

[CYES19] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, 9:292–308, Jun 2019.

[Dav19] T. A. Davis. Algorithm 1000: SuiteSparse: GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Transactions on Mathematical Software*, 45(4), 2019.

[DBM+15] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland. Optimizing Sparse Matrix Operations on GPUs Using Merge Path. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2015.

[Dem13] M. Demler. MediaTek Steps Up to Tablets: MT8135 Brings Heterogeneous Multiprocessing to Big.Little. *Microprocessor Report*, Aug 2013.

[DH11] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec 2011.
URL `http://doi.acm.org/10.1145/2049662.2049663`

[DKM+02] A. Danowitz, K. Kelley, J. Mao, J. Stevenson, M. Horowitz, and C. DB. Recording Microprocessor History. *ACM Queue Magazine*, 10(4), 2002.

[DOB15] S. Dalton, L. Olson, and N. Bell. Optimizing Sparse Matrix—Matrix Multiplication for the GPU. *ACM Trans. on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

[DTH20] W. J. Dally, Y. Turakhia, and S. Han. Domain-Specific Hardware Accelerators. *Communications of the ACM*, 63(7):48–57, 2020.

[DTR18] M. Deveci, C. Trott, and S. Rajamanickam. Multithreaded Sparse Matrix-Matrix Multiplication for Many-Core and GPU Architectures. *Parallel Computing*, 78:33–46, 2018.

[Dub05] M. Dubash. Moore's Law is Dead, Says Gordon Moore. *Techwold*, Apr 2005.

[DVWW05] T. H. Dunigan, J. S. Vetter, J. B. White, and P. H. Worley. Performance Evaluation of the Cray X1 Distributed Shared-Memory Architecture. *IEEE Micro*, 25(1):30–40, 2005.

[DXT+18] S. Davidson, S. Xie, C. Torng, K. Al-Hawaj, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. G. Dreslinski, C. Batten, and M. B. Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, Mar/Apr 2018.

[EAE+02] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2002.

[EML88] C. Eoyang, R. H. Mendez, and O. M. Lubeck. The Birth of the Second Generation: the Hitachi S-820/80. *Conference on Supercomputing*, 1988.

[FC23a] V. L. Fèvre and M. Casas. Optimization of SpGEMM with RISC-V Vector Instructions. *Computing Research Repository (CoRR)*, arXiv:2303.02471, Mar 2023.

[FC23b] V. L. Fèvre and M. Casas. Optimization of SpGEMM with RISCV-V Vector Instructions. *Computing Research Repository (CoRR)*, Mar 2023.

[Fog12] A. Fog. The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. Technical report, College of Engineering, Copenhagen University, 2012.

[Gal96] G. Galli. Linear Scaling Methods for Electronic Structure Calculations and Quantum Molecular Dynamics Simulations. *Current Opinion in Solid State and Materials Science*, 1(6):864–874, 1996.

[GCC+08] J. Gonzalez, Q. Cai, P. Chaparro, G. Magklis, R. N. Rakvic, and A. Gonzalez. Thread fusion. *Int'l Symp. on Low-Power Electronics and Design (ISLPED)*, Aug 2008.

[gem21] Arm's AMBA 5 CHI Ruby Model in gem5. Online Webpage, accessed Nov 20, 2021. `https://www.gem5.org/documentation/general_docs/ruby/CHI/`.

[GFAM10] S. Gupta, S. Feng, A. Ansari, and S. Mahlke. Erasing Core Boundaries for Robust and Configurable Performance. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2010.

[GHL+20] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu. Accelerating Sparse DNN Models Without Hardware-Support via Tile-Wise Sparsity. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2020.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, pages 15–26, 1990.

[GMFC21] C. Gómez, F. Mantovani, E. Focht, and M. Casas. Efficiently Eunning SpMV on Long Vector Architectures. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2021.

[GMS92] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse Matrices in MATLAB: Design and Implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[GPD+14] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, N. Paver, and N. K. Jha. Sources of Error in Full-System Simulation. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2014.

[Gwe13] L. Gwennap. Renesas Mobile Goes Big (and Little). *Microprocessor Report*, Feb 2013.

[Gwe14a] L. Gwennap. Qualcomm Tips Cortex-A57 Plans: Snapdragon 810 Combines Eight 64-Bit CPUs, LTE Baseband. *Microprocessor Report*, Apr 2014.

[Gwe14b] L. Gwennap. Samsung First with 20 nm Processor. *Microprocessor Report*, Sep 2014.

[HAMP+19] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher. Extensor: An accelerator for sparse tensor algebra. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2019.

[HBB+18] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2018.

[HDJ+20] M. Huzaifa, R. Desai, X. Jiang, J. Ravichandran, F. Sinclair, and S. V. Adve. Exploring Extended Reality with ILLIXR: A New Playground for Architecture Research. *arXiv preprint arXiv:2004.04643*, 2020.

[HDUZ21] Y. Hu, Y. Du, E. Ustun, and Z. Zhang. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2021.

[HKOO11] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2011.

[HMD15] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *Computing Research Repository (CoRR)*, Oct 2015.

[HN07] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. *Int'l Conf. on High-Performance Computing (HIPC)*, Dec 2007.

[HP19] J. L. Hennessy and D. A. Patterson. A New Golden Age for Computer Architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[HPA+20] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge. Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices. *Int'l Symp. on Supercomputing (ICS)*, Jun 2020.

[HR21] M. D. Hill and V. J. Reddi. Accelerator-Level Parallelism. *Communications of the ACM*, 64(12):36–38, 2021.

[HS11] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. *Int'l Symp. on Supercomputing (ICS)*, May 2011.

[ibm23] Matrix-Multiply Assist Best Practices Guide. Online Webpage, 2021 (accessed Apr 2023). https://www.redbooks.ibm.com/redpapers/pdfs/redp5612.pdf.

[IKKM07] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2007.

[int07] Intel SSE4 Programming Reference. Intel Reference Manual, 2007. http://software.intel.com/file/18187.

[int12] Intel AVX. Online Webpage, 2012 (accessed March, 2012). http://software.intel.com/en-us/avx.

[int13] Intel Cilk Plus Language Extension Specification, Version 1.2. Intel Reference Manual, Sep 2013. https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[int23a] Intel 64 and IA-32 Architectures Optimization Reference Manual. Intel Reference Manual, 2023. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[int23b] Intel Advanced Matrix Extensions Overview. Online Webpage, (accessed Apr 2023). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html.

[JDB+23] G. Jeong, S. Damani, A. R. Bambhaniya, E. Qin, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna. VEGETA: Vertically-Integrated Extensions for Sparse/-Dense GEMM Tile Acceleration on CPUs. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar 2023.

[JIB18] S. Jiang, B. Ilbeyi, and C. Batten. Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks. *Design Automation Conf. (DAC)*, Jun 2018.

[JKL+23a] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, et al. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2023.

[JKL+23b] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. *Computing Research Repository (CoRR)*, arXiv:2304.01433, Apr 2023.

[JOP+20] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang, and C. Batten. PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies. *IEEE Design and Test of Computers*, 38(2):53–61, 2020.

[JQS+21] G. Jeong, E. Qin, A. Samajdar, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna. RASA: Efficient Register-Aware Systolic Array Matrix Engine for CPU. *Design Automation Conf. (DAC)*, Nov 2021.

[JYK+20] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Communications of the ACM*, 63(7):67–78, Jul 2020.

[JYP+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[JYPP18] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. A Domain-Specific Architecture for Deep Neural Networks. *Communications of the ACM*, 61(9):50–59, 2018.

[KBA08] R. Krashinsky, C. Batten, and K. Asanović. Implementing the Scale Vector-Thread Processor. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 13(3), Jul 2008.

[KBH+04a] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[KBH+04b] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *IEEE Micro*, 24(6):84–90, Nov/Dec 2004.

[KDK+11] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, Sep/Oct 2011.

[KFJ+03] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2003.

[KJT04] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-Core Chip Multiprocessing. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[KJT+17] J. Kim, S. Jiang, C. Torng, M. Wang, S. Srinath, B. Ilbeyi, K. Al-Hawa, and C. Batten. Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2017.

[KKCL13] M. Kim, H. Kim, H. Chung, and K. Lim. Samsung Exynos 5410 Processor-Experience the Ultimate Performance and Versatility. *White Paper*, 2013.

[KSG+07] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2007.

[KSH+12] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2012.

[KTD12] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 23(11):2045–2057, 2012.

[KTHK03] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A Hardware Overview of SX-6 and SX-7 Supercomputer. *NEC Research & Development Journal*, 44(1):2–7, Jan 2003.

[KTR+04] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[LAB+11] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerator Cores. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[LCS+19] Y. Leng, C.-C. Chen, Q. Sun, J. Huang, and Y. Zhu. Energy-efficient Video Processing for Virtual Reality. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2019.

[Lee97] R. Lee. Multimedia Extensions for General-purpose Processors. *IEEE Workshop on Signal Processing Systems (SiPS) Design and Implementation*, 1997.

[LFB+10] G. Long, D. Franklin, S. Biswas, P. Oritz, J. Oberg, D. Fan, and F. T. Chong. Minimal Multi-threading: Finding and Removing Redundant Instructions in Multi-threaded Processors. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2010.

[LL18] S. Liu and D. Liu. A High-Flexible Low-Latency Memory-Based FFT Processor for 4G, WLAN, and Future 5G. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 27(3):511–523, 2018.

[LLK+19] R. Lim, Y. Lee, R. Kim, J. Choi, and M. Lee. Auto-tuning GEMM Kernels on the Intel KNL and Intel Skylake-SP Processors. *Journal of Supercomputing*, 75:7895–7908, 2019.

[LLW+06] T.-M. Liu, T.-A. Lin, S.-Z. Wang, W.-P. Lee, J.-Y. Yang, K.-C. Hou, and C.-Y. Lee. A 125μW, Fully Scalable MPEG-2 and H. 264/AVC Video Decoder for Mobile Applications. *IEEE Journal of Solid-State Circuits (JSSC)*, 42(1):161–169, 2006.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[LNW23] A. Li, A. Ning, and D. Wentzlaff. Duet: Creating Harmony between Processors and Embedded FPGAs. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2023.

[LPAA+20] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[LSFJ06] C. Lemuet, J. Sampson, J. Francios, and N. Jouppi. The Potential Energy Efficiency of Vector Acceleration. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2006.

[LSPS10] S.-K. Lee, Y.-H. Seo, H.-J. Park, and J.-Y. Sim. A 1 GHz ADPLL With a 1.25 ps Minimum-Resolution Sub-Exponent TDC in 0.18*mu*m CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 45(12):2874–2881, 2010.

[LV14] W. Liu and B. Vinter. An Efficient GPU General Sparse Matrix-matrix Multiplication for Irregular Data. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2014.

[LW21] A. Li and D. Wentzlaff. PRGA: An Open-Source FPGA Research and Prototyping Framework. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2021.

[LWAQ19] J. Li, F. Wang, T. Araki, and J. Qiu. Generalized Sparse Matrix-matrix Multiplication for Vector Engines and Graph Applications. *IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, Nov 2019.

[LWM20] Z.-G. Liu, P. N. Whatmough, and M. Mattina. Systolic Tensor Array: An Efficient Structured-Sparse GEMM Accelerator for Mobile CNN Inference. *Computer Architecture Letters (CAL)*, 19(1):34–37, 2020.

[LYA18] H. Li, H. Yokoyama, and T. Araki. Merge-Based Parallel Sparse Matrix-Sparse Vector Multiplication with a Vector Architecture. *IEEE Int'l Conf. on High Performance Computing and Communications; IEEE Int'l Conf. on Smart City; IEEE Int'l Conf. on Data Science and Systems (HPCC/SmartCity/DSS)*, Jun 2018.

[MBW14] M. Mckeown, J. Balkind, and D. Wentzlaff. Execution Drafting: Energy Efficiency Through Computation Deduplication. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MPZ+20] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker, et al. VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(2):1–55, 2020.

[MSM05] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[MSS+15] S. Mathew, S. Satpathy, V. Suresh, M. Anders, H. Kaul, A. Agarwal, S. Hsu, G. Chen, and R. Krishnamurthy. 340 mV–1.1 V, 289 Gbps/W, 2090-gate NanoAES Hardware Accelerator with Area-Optimized Encrypt/Decrypt GF (2 4) 2 Polynomials in 22 nm Tri-gate CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 50(4):1048–1058, 2015.

[NCVK10] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. CUSPARSE Library, 2010.

[NMAB18] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures. *Workshop Proceedings of Int'l Conf. on Parallel Processing (ICPP)*, Aug 2018.

[NMHS15] T. Nowatzki, J. Menon, C. H. Ho, and K. Sankaralingam. Architectural Simulators Considered Harmful. *IEEE Micro*, 35(6):4–12, Nov 2015.

[NMM+22] N. Nassif, A. O. Munch, C. L. Molnar, G. Pasdast, S. V. Lyer, Z. Yang, O. Mendoza, M. Huddart, S. Venkataraman, S. Kandula, et al. Sapphire Rapids: The Next-generation Intel Xeon Scalable Processor. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2022.

[NMS+19] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, et al. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *Computing Research Repository (CoRR)*, May 2019.

[NST+19] M. Natsui, D. Suzuki, A. Tamakoshi, T. Watanabe, H. Honjo, H. Koike, T. Nasuno, Y. Ma, T. Tanigawa, Y. Noguchi, et al. 12.1 An FPGA-Accelerated Fully Nonvolatile Microcontroller Unit for Sensor-Node Applications in 40nm CMOS/MTJ-Hybrid Technology Achieving 47.14 μW Operation at 200MHz. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2019.

[nvi09] NVIDIA's Next Gen CUDA Compute Architecture: Fermi. NVIDIA White Paper, 2009. `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`.

[Olo16] A. Olofsson. Epiphany-V: A 1024-processor 64-bit RISC System-On-Chip. *CoRR arXiv:1610.01832*, Aug 2016.

[ope08] OpenMP Application Program Interface. OpenMP Architecture Review Board, 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[PBP+18] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2018.

[PCW+22] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini. A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design. *Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, Jul 2022.

[PPPM12] Y. Park, J. J. K. Park, H. Park, and S. Mahlke. Libra: Tailoring SIMD Execution Using Heterogeneous Hardware and Dynamic Configurability. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2012.

[PSR21] J. Pool, A. Sawarkar, and J. Rodge. Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT. NVIDIA Technical Report, 2021. `https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt`.

[PVZ15] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep Face Recognition, 2015.

[PW96] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[PZK+18] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A Reconfigurable Accelerator for Parallel Patterns. *IEEE Micro*, 38(3):20–31, 2018.

[QHS+13] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[QSK+20] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2020.

[RA16] R. A. Rossi and N. K. Ahmed. An Interactive Data Repository with Visual Analytics. *SIGKDD Explor.*, 17(2):37–41, 2016.
URL `http://networkrepository.com`

[Ran13] R. Randhawa. Software Techniques for ARM big. LITTLE Systems. Arm Whitepaper, 2013.

[RBGZ19] J. Rupley, B. Burgess, B. Grayson, and G. D. Zuraski. Samsung M3 Processor. *IEEE Micro*, 39(2):37–44, 2019.

[RCK+20] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al. MLPerf Inference Benchmark. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2020.

[RCYQ22] G. Rao, J. Chen, J. Yik, and X. Qian. SparseCore: Stream ISA and Processor Specialization for Sparse Computation. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb 2022.

[Rei07] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[RHP+20] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. *ACM Trans. on Architecture and Code Optimization (TACO)*, 17(4):1–30, 2020.

[ris18] RISC-V Foundation. `http://www.riscv.org`, 2018 (accessed Mar 17, 2018).

[RIS21] RISC-V Vector Extension (Version 0.10). Online Webpage, 2021. `https://github.com/riscv/riscv-v-spec/releases/tag/v1.0`.

[ris23] RISC-V Matrix Extension Specification Proposal. Online Webpage, (accessed Apr 2023). `https://github.com/T-head-Semi/riscv-matrix-extension-spec`.

[RMR+20] F. Renzini, C. Mucci, D. Rossi, E. F. Scarselli, and R. Canegallo. A Fully Programmable eFPGA-Augmented SoC for Smart Power Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(2):489–501, 2020.

[roc18] Rocket Core Overview. `http://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core`, 2018 (accessed Mar 17, 2018).

[RS17] A. Roelke and M. R. Stan. RISC5: Implementing the RISC-V ISA in gem5. *Workshop on Computer Architecture Research with RISC-V (CARRV)*, Oct 2017.

[RSM+11] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. *Int'l Conference on Mobile Systems, Applications, and Services*, Jun 2011.

[RSOK06] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector Lane Threading. *Int'l Conference on Parallel Processing (ICPP)*, Aug 2006.

[Rus78] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[Sat20] M. Sato. The Supercomputer "Fugaku" and Arm-SVE Enabled A64FX Processor for Energy Efficiency and Sustained Application Performance. *Int'l Symp. on Parallel and Distributed Computing (ISPDC)*, 2020.

[SB13a] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2013.

[SB13b] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2013.

[SB22] Y. S. Shao and D. Brooks. *Research Infrastructures for Hardware Accelerators*. Springer Nature, 2022.

[SBB+17] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2), 2017.

[SCS+22] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2022.

[SGC+16] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016.

[SIT+14] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[SJL+20] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2020.

[SRDM+21] P. D. Schiavone, D. Rossi, A. Di Mauro, F. K. Gürkaynak, T. Saxe, M. Wang, K. C. Yap, and L. Benini. Arnold: An eFPGA-Augmented RISC-V SoC for Flexible and Low-Power IoT End Nodes. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 29(4):677–690, 2021.

[SRS+12] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, UIUC, IMPACT-12-01, Mar 2012.

[ST15] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. *Int'l Conf. on Data Engineering (ICDE)*, Apr 2015.

[SWL+92] M. L. Simmons, H. J. Wasserman, O. M. Lubeck, C. Eoyang, R. Mendez, H. Harada, and M. Ishiguro. A Performance Comparison of Four Supercomputers. *Communications of the ACM*, 1992.

[TAHC+22] T. Ta, K. Al-Hawaj, N. Cebry, Y. Ou, E. Hall, C. Golden, and C. Batten. big.VLITTLE: On-Demand Data-Parallel Acceleration for Mobile Systems on Chip. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2022.

[TBD18] Y. Turakhia, G. Bejerano, and W. J. Dally. Darwin: A Genomics Co-processor Provides up to 15,000 x Acceleration on Long Read Assembly. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2018.

[TBS08] D. Tarjan, M. Boyer, and K. Skadron. Federation: Repurposing Scalar Cores for Out-Of-Order Instruction Issue. *Design Automation Conf. (DAC)*, Jun 2008.

[TCB18] T. Ta, L. Cheng, and C. Batten. Simulating Multi-Core RISC-V Systems in gem5. *Workshop on Computer Architecture Research with RISC-V*, 2018.

[TCS20] A. Tino, C. Collange, and A. Seznec. SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores. *ACM Trans. on Architecture and Code Optimization (TACO)*, 17(2), 2020.

[Ten23] Ocelot: The Berkeley Out-of-Order RISC-V Processor with Vector Support, 2023. `https://github.com/tenstorrent/riscv-ocelot`.

[Tie20] P. Tiech. Under the Hood of Google's TPU2 Machine Learning Clusters. Online Article, May 2017 (accessed Feb 2020). `https://www.nextplatform.com/2017/05/22/hood-googles-tpu2-machine-learning-clusters`.

[TMB14] R. Thabet, R. Mahmoudi, and M. H. Bedoui. Image Processing on Mobile Devices: An Overview. *Int'l Image Processing, Applications and Systems Conference (IPAS)*, Nov 2014.

[TNH+06] S. Tagaya, M. Nishida, T. Hagiwara, T. Yanagawa, Y. Yokoya, H. Takahara, J. Stadler, M. Galle, and W. Bez. The NEC SX-8 Vector Supercomputer System. *High Performance Computing on Vector Systems*, May 2006.

[TOJ+19] C. Tan, Y. Ou, S. Jiang, P. Pan, C. Torng, S. Agwa, and C. Batten. PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks. *Int'l Conf. on Computer Design (ICCD)*, Nov 2019.

[TWB16] C. Torng, M. Wang, and C. Batten. Asymmetry-Aware Work-Stealing Runtimes. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[VSP+17] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. G. Katevenis. Modeling Energy-Performance Tradeoffs in ARM big. LITTLE Architectures. *Int'l Symp. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep 2017.

[WAZ+19] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra. High-throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 39(10), 2019.

[WBC+19a] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine Learning at Facebook: Understanding Inference at the Edge. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2019.

[WBC+19b] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine Learning at Facebook: Understanding Inference at the Edge. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar 2019.

[Whe20] B. Wheeler. Ampere Maxes Out at 128 Cores. *Microprocessor Report*, Jul 2020.

[WMZ+19] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. Adaptive Sparse Matrix-matrix Multiplication on the GPU. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2019.

[Wol20] C. Wolf. Yosys Open SYnthesis Suite. Yosys Manual, 2020. `https://yosyshq.net/yosys/`.

[WTCB20] M. Wang, T. Ta, L. Cheng, and C. Batten. Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems. *Int'l Symp. on Computer Architecture (ISCA)*, May 2020.

[WZS+14] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, Y. Wang, E. Wang, Q. Zhang, B. Shen, et al. Intel Math Kernel Library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pages 167–188, 2014.

[ZAES21] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2021.

[ZB91] M. Zagha and G. E. Blelloch. Radix Sort for Vector Multiprocessors. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, 1991.

[ZB19] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 27(11):2629–2640, 2019.

[ZCL+19] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang. Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.

[ZR13] Y. Zhu and V. J. Reddi. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[ZWHD20] Z. Zhang, H. Wang, S. Han, and W. J. Dally. SpArch: Efficient Architecture for Sparse Matrix Multiplication. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2020.