EPHEMERAL VECTOR ENGINES

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
Khalid Musa Jaffar Al-Hawaj
December 2022

© 2022 Khalid Musa Jaffar Al-Hawaj
ALL RIGHTS RESERVED

EPHEMERAL VECTOR ENGINES

Khalid Musa Jaffar Al-Hawaj, Ph.D.

Cornell University 2022

With the recent slowdown of Moore's Law and the end of Dennard's Scaling, computer architects have turned to specialization to retain the regular improvements in performance and efficiency conventionally obtained through process advancements. Although traditional fixed-function acceleration is able to achieve high performance and efficiency by leveraging specialization, it struggles with low programmability and lack of flexibility. Previous work has shown the ability of next-generation vector abstraction to balance programmability and specialization. The recent rise in popularity of next-generation vector architectures highlights the inherent tension between its two traditional micro-architectures: integrated vector unit and dedicated vector engine. While integrated vector unit achieves modest performance with low area-overhead, dedicated vector engine achieves higher performance at the expense of higher area-overhead. This thesis leverages recent advancements in in-situ compute-in-memory to address this tension. The culmination of this thesis, ephemeral vector engines (EVE), aims at solving this tension. EVE is a novel next-generation vector micro-architecture leveraging SRAM-based compute-in-memory (S-CIM) circuits to reconfigure private L2 caches on-the-fly to support next-generation vector execution. While previous work on S-CIM has explored bit-serial execution, this thesis further explores bit-parallel execution with the following conclusion: bit-serial achieves high-throughput but high-latency, while bit-parallel lowers the latency greatly at the expense of lower throughput. This thesis considers a bit-hybrid approach instead to balance throughput and latency. To evaluate the area and cycle-time of EVE, this thesis presents a detailed circuit template that enables bit-hybrid S-CIM with varying parallelization factor. To evaluate the performance of EVE, this thesis leverages high-fidelity cycle-approximate models for an integrated vector unit, a decoupled vector engine, and EVE. By leveraging S-CIM, EVE increases performance by 4.59 $\times$ over an integrated vector unit, thus matching the performance of a decoupled vector engine while incurring a tenth of its area-overhead. In summary, EVE leverages S-CIM to achieve a performance comparable to the decoupled vector engine, while incurring an area-overhead equivalent to that of the integrated vector unit.

BIOGRAPHICAL SKETCH

Khalid Al-Hawaj was born on the 15th of Rabi' Al-Thani 1407 AH in Al-Qatif, Saudi Arabia to Huda Al-Sha'aban and Musa Al-Hawaj. He grew up in Al-Qalah to a loving family of two brothers and three sisters. Ever-since he was a little boy, Khalid showed interest in science and technology. During middle school, he taught himself BASIC and C programming. To his mother's dismay, in high school, he spent his time working on science, electronics, and programming projects.

Khalid was accepted to King Fahd University of Petroleum and Minerals (KFUPM) in Al-Dhahran, Saudi Arabia. Due to his immense interest in hardware and software, he pursued a Bachelor's degree in computer engineering, during which he realized that computer architecture would be his passion. After graduating from KFUPM, Khalid joined Saudi Arabia Basic Industry Company (SABIC), where he worked as a metal production consultant. After 1three years at SABIC, Khalid decided to quit and join Harvard University to pursue Masters degree with focus on computer architecture and advised by Professor David Brooks.

Khalid was admitted to the Ph.D. program in the School of Electrical and Computer Engineering at Cornell University in 2015, where he was mentored by Professor Christopher Batten. He initially worked on the loop-task accelerators project led by Dr. Ji Kim. Later, he was fortunate to join Cornell University's effort on the Celerity chip, which is a part of the DARPA CRAFT (Chip Realization at Faster Timescale) project. Khalid led the effort on designing and testing AgileBNN. After the conclusion of the CRAFT project, Khalid started and the led the EVE project within the Batten Research Group (BRG). Throughout his Ph.D. journey and under the mentorship of Professor Christopher Batten, Khalid dived into compute-in-memory and gained invaluable knowledge and experience about emerging applications, computer architecture, circuits, and VLSI.

Aside from computer architecture, Khalid has a fascination for old technology and philosophy. Despite his ruptured ACL, Khalid is an avid Squash player and have recently picked up Badminton. As he missed home during his Ph.D. journey, and with the help of his mother, Khalid learned to cook his favorite childhood comfort food.

Khalid is very thankful for the opportunity given to him at Cornell University. During his Ph.D. experience, due to the excellent mentorship from his advisor and the irreplaceable companionship of his friends, Khalid grew as a person and a researcher. Khalid is thankful for his Ph.D. experience–the ups and the downs, the happy time and the gloomy time; for without failure, one cannot recognize nor appreciate the rich taste of success.

In the Name of Allah, the Most Beneficent, the Most Merciful.

Alhamdulillah

This document is dedicated to my mother "*Umm Faisal*", my father "*Abu Faisal*", my siblings, my nephews, and my friends ...

ACKNOWLEDGEMENTS

First and foremost, I am thankful for Allah for all His blessing all His guidance. I am also thankful for my family–my source for motivation and inspiration. My mother, Huda Al-Sha'aban, was my first mentor and has taught me at a tender age my science foundation. She showed me at a younger age the value of resilience–persevering through my inevitable failures to reach my eventual success. My father, Musa Al-Hawaj, was my role model and has inspired me to delve into the science behind technology and electronics. I am thankful for my eldest brother Faisal Al-Hawaj for including me in his morning study sessions for medical school. Talking to my cousin Mohammed Al-Hawaj every morning provided me with joyful company. I am thankful for my young siblings (Mariam, Hajir, Sarah, and Ahmad) and my nephews (Abdullah, Ali, and Abdulaziz) as they provided me with the support and motivation to continue through the hard times during my Ph.D. journey.

This dissertation would not have been possible without the guidance and insight of my graduate committee: Professor Christopher Batten, Professor José Martínez, and Professor Christoph Studer. I would especially like to thank my mentor and advisor, Professor Christopher Batten, who provided me with an environment that fostered my curiosity and allowed me to grow as a researcher in computer architecture. When I came to Professor Christopher Batten with a crazy idea to perform in-situ computations in a 6T-SRAM, he provided useful and positive criticism and allowed the idea to flourish, thus becoming the main theme of this dissertation. I would like to acknowledge the rest of my committee: Professor José Martínez, I am thankful for the different perspective I gained from interacting and working along your side in our collaborative projects; and Professor Christoph Studer, the VLSI and circuits themed chats were a well of knowledge that benefited me throughout my journey.

Before I have started my Ph.D. journey, I was at Harvard University working towards my Master's degree under the mentorship of Professor David Brooks. I would like to acknowledge Professor David Brooks for accepting me into VLSIArch and mentoring me. I owe a lot of my foundation in computer architecture to the time I spent as a member of VLSIArch. I would like to acknowledge the colleagues from VLSIArch: Brandon Reagen, Simone Campanoni, Svilen Kanev, Kevin Brownell, Yakun Sophia Shao, Bob Adolf, Tao Tong, Mario Lok, Sam Xi, and Emma Wang. I would like to especially acknowledge Glenn Holloway for all the invaluable knowledge

and wisdom he ever-so-generously bestowed upon me. I would like to acknowledge my companion throughout my stay in Cambridge: Shuyi James Chen, you made my stay more enjoyable.

I would like to acknowledge my brilliant colleagues at the Batten Research Group (BRG) for their help and support: Ji Kim whose project was my introduction to BRG; I appreciate your help, guidance, and your words of wisdom (which stuck with me throughout my Ph.D. journey). Shreesha Srinath, thanks for being available to discuss ideas and argue about the intellectual merits. Berkin Ilbeyi, you taught me a lot about JIT, Python, and compilers. Christopher Torng, you were a great mentor and has taught me everything you knew about chip-design and productivity. Moyang Wang, maybe one day, you will retire by buying a six-wheeler and drive around the USA endlessly. Shunning Jiang, I genuinely enjoyed our discussion about research in my first office I shared with you at Wards hall. Tuan Ta, thank you for waking me up for the paper submission. I believe every successful Ph.D. experience requires a bouncer, where one can bounces ideas and get positive critique and feedback, and Tuan Ta was my bouncer. Lin Chang, one day I will find these fries on the pier. Shady Agwa, thank you for being my circuit-buddy at the BRG. Peitian Pan, your sharp answers and keen critical thinking amazes me. Yanghui Ou, remember: the tape-out deadline is never-ending. Nick Cebry, thanks for all your contribution and discussions while developing the BRG-DV model. Preslav Ivanov, I will always remember fondly sharing herbal Bulgarian drink at the CSL. I would like to acknowledge the newest additions of BRG: Austin Rovinski, working with you as a Ph.D. student at the University of Michigan before you have joined BRG as a post-doctorate is quiet the twist in my Ph.D. journey.

Maybe the Ph.D. was the friends we made along the way; I would like to acknowledge the close friends that helped me and supported me throughout my Ph.D. journey: Zainulabideen Khalifa, I learned a lot from you and I will, insha'Allah, keep learning for years to come; Omar Al-Thuhaiby, I do not believe there is an end to your surprises. Charles Jeon, I will never forget our late night chats in the patio of CTB till the break of dawn. Ryan O'Hern, I am thankful for all the enlightening philosophical discussions. Ritchie Zhao, sharing an office with you made the journey much more fun. Tuan Ta, thank you for introducing me to Badminton and maybe one day I will beat you in a singles match. Seyed Hadi Mirfarshbafan, I will always include you in my Dua'a. Philip Bedoukian, may we have our Halo sessions for many years to come. Haron Abdel-Raziq, shouting at each other in our football matches was the highlight of my week. Nikita Lazarev, one day I will bike up the hill as I promised. Steven Ceron, I appreciate all the knowledge I learned from

you about soft robots. The Maplewood crew (Jordan Dotzel, Jamie Ye, Shimin Huang, Thomas Conroy), I am thankful for your generous hosts every Friday night. The Squash crew (Celine Lee, Anshuman Mohan, Mauricio Pezo), thank you for all the challenging games; I would like to especially acknowledge Celine Lee for generously accepting Mister dead Lobstah. The 471F crew (Steve Dai, Gai Liu, Ryan O'Hern, Shreesha Srinath, Berkin Ilbyei, Ritchie Zhao, and Ecenur Üstün), thanks for the laughs and the fun times. The rest of the CSL community (Sachille Atapattu, Helena Caminal, Neeraj Kulkarni, Nitish Srivastava, Skand Hurkat, Shuang Chen, and Mark Buckler) for creating an awesome environment with diverse research. I also would like to acknowledge Collegetown Bagels (CTB) for all the late-night double-shot espressos.

My childhood friends, who could not have accompanied me physically during my Ph.D. journey in Ithaca, are a big source of support. I would like to acknowledge every friend who kept in-contact with me sending words of motivation all the way from my home city, Al-Qatif. I would like to apologize to them as some of their messages sometimes went unanswered due to deadlines, but an answer eventually is provided. I would like to acknowledge Aman Al-Askri, Ali Al-Numer, Ali Al-Turaish, Ahmad Al-Saif, Sayed Mohammed Al-Muri'e, Sa'eed Aal-Shaikh, Ahmad Al-Doshaishi, and many more that I cannot fit in this paragraph. I would like to especially acknowledge my late friends, whose memory I hold dearly till the day I join them.

The work in this thesis was supported in part by: King Abdullah's Scholarship Program (KASP), King Salman's Scholarship Program (KSSP), NSF E2CDA Award #1740136, NSF SHF Award #2008471, NSF PPoSS Award #2118709, DARPA SDH Award #FA8650-18-2-7863, DARPA CRAFT Award #HR0011-16-C-0037, the Semiconductor Research Corporation (SRC) as nCORE task 2758.002 and 2758.004, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, as well as equipment, tool, and/or physical IP donations from Intel, Synopsys, Cadence, and ARM.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CMP	chip multiprocessor
SIMD	single-instruction multiple-data
SIMT	single-instruction multiple-thread
GPP	general-purpose processor
GPGPU	general-purpose graphical processing unit
AVX	(Intel) Advanced Vector Extensions
SVE	(Arm) Scalable Vector Extension
RVV	RISC-V Vector Extension
ISA	instruction set architecture
RTL	register-transfer level
VLSI	very-large-scale integration
ASIC	application-specific integrated circuit
RF	register file
VRF	vector register file
ALU	arithmetic and logical unit
VALU	vector arithmetic and logical unit
VCU	vector control unit
VSU	vector sequencing unit
VXU	vector execution unit
VRU	vector reduction unit
DTU	data transpose unit
BL	bitline wire
BLB	bitline-bar wire
SA	sense-amplifier
MOSFET	metal-oxide field-effect transistor
RS	reservation station
SRAM	static random access memory
DRAM	dynamic random access memory
RRAM	resistive random access memory
PIM	processing-in-memory
CIM	compute-in-memory
S-CIM	SRAM-based compute-in-memory
D-CIM	DRAM-based compute-in-memory
R-CIM	RRAM-based compute-in-memory
I$	level-1 instruction cache
D$	level-1 data cache
L2	level-2 cache
LLC	last-level cache
LDQ	load queue
STQ	store queue
LSQ	load-store queue
REQ	request
RESP	response

CHAPTER 1
INTRODUCTION

Emerging data-parallel applications (e.g., image processing, computer vision, and machine learning) continue to demand increases in both performance and efficiency; however, the slowdown of Moore's Law and the end of Dennard Scaling have forced computer architects to turn to specialization to meet these demands. More recently, there has been a resurgence of interest in vector architectures as demonstrated by the recent introduction of the RISC-V Vector extension (RVV), ARM's Scalable Vector Extension (SVE), and Intel's Advanced Vector eXtension (AVX-512). Vector architectures exploit the regularity inherent in data-parallel workloads to achieve high performance and efficiency despite the slowdown of Moore's Law and the end of Dennard's Scaling. There are two possible implementations of these vector architectures: (1) decoupled vector engines, and (2) integrated vector units. There is a fundamental tension between these vector micro-architectures: decoupled vector engines achieve high performance at high area overhead, while integrated vector units achieve more modest perforamnce at lower area overhead. Prior work on SRAM-based compute-in-memory have shown promise in mitigating the area overhead associated with vector execution, thus addressing the aforementioned tension between vector micro-architectures; at the same time, this prior work have also shown how vector abstraction is a fitting solution to compute-in-memory's own programmability challenge.

In this thesis, I propose ephemeral vector engines (EVE)—a novel next-generation vector architecture that mitigates the area-overhead traditionally associated with vector execution through silicon reconfigurability. EVE leverages novel SRAM-based compute-in-memory techniques to reconfigure on-the-fly private L2 caches of a core to act as execution units. The main insight in my thesis revolves around exploring and proposing different execution paradigms between bit-serial and bit-parallel in the quest to balance throughput and latency. In my thesis, I make the observation that the design point with the optimal parallelization factor for an SRAM-based compute-in-memory vector engine lies in-between bit-serial and bit-parallel. Through a circuits-to-architecture evaluation methodology, I argue that a bit-hybrid execution paradigm is able to balance throughput and latency allowing EVE to achieve performance equivalent to an aggressive decoupled vector engine while incurring an area overhead comparable to an integrated vector unit.

Attribute	Packed SIMD	Long Vector	Next Generation
Length	fixed, short	scalable, long	scalable
Element Width	variable	fixed	variable
Predication	limited	full	full
Cross-Element Ops	full	limited	full
Memory Gather/Scatter	limited	full	full
Integration	integrated	decoupled	either
Speculative Execution	yes	no	either
Compute Pipeline	integrated	decoupled	either
Memory Bandwidth	modest	large	either
Memory Latency	low	high	either

Table 1.1: A Summary of Vector Architectures.

1.1 Vector Architectures

Table 1.1 shows an overview of the various vector architectures and their properties. Traditionally, vector architectures has been implemented as aggressive long-vector machines to accelerate scientific computations conducted in super-computers [Rus78, DVWW05, KTHK03, TNH+06]. Since vector machines lose their performance and efficiency advantage when executing non-regular code, mainstream compute systems conventionally utilize packed-SIMD units to perform simplified vector execution. Recently, there has been an emerging trend towards next-generation vector architectures that provide unified abstraction across the wide compute systems spectrum. This section serves as an introduction for the different vector architectures.

1.1.1 Long-Vector Architectures

Early implementations of vector machines (e.g., Texas Instruments ASC [Wat72], CDC STAR 100 [Sch87], Illiac IV [BDM+72], CRAY-1 [Rus78]) were predominantly long vector architectures. As their name implies, long-vector architectures adopt a scalable hardware vector length abstraction. The long hardware vector length often implements chiming, where the long active vector is broken down into element groups; these element groups are executed in a tight-coupled fashion. To avoid redundant computations, long vector architectures employ masking to specify which elements in the active vector to be executed and which to be skipped. To bring data into

vector registers, the vector memory operations often support expressive and complex addressing (e.g., constant stride addressing, and indexed addressing).

More recent implementations of long-vector architectures span two micro-architectures: vector processors [ABS+07, DVWW05, Sat20, KTHK03, TNH+06], and vector engines [EAE+02, CXL+20, CSZ+19]. Vector processors are specialized processors that are designed to optimally execute vector code with less emphasis on their scalar performance. Whereas vector engines are decoupled units attached to a scalar control processor responsible for dispatching vector instructions to these engines. Vector processors approach sacrifice scalar performance to achieve lower area overhead, whereas vector engines approach incur higher area overhead to retain higher scalar performance. Nevertheless, both micro-architectures achieve high performance in vector execution at the expense of high area requirement.

1.1.2 Packed-Single-Instruction-Multiple-Data Architectures

With higher demands for accelerating regular data-parallel execution, several packed-single-instruction-multiple-data (Packed-SIMD) architectures [PW96, int07, int12, arm09, JKK11] were introduced as a compromise between area and performance for vector execution. Packed-SIMD architectures are executed through packed-SIMD units, which are tightly integrated into general-purpose cores. By re-using silicon from the general-purpose core, these packed-SIMD architectures are able to achieve modest vector performance while retaining minimum area overhead. Unlike long-vector architectures, packed-SIMD architectures restrict their vector support to minimize their area overhead. The hardware vector length for packed-SIMD is often short to be able to re-use the register file from the general-purpose core; as a result, chiming is often unsupported by these architectures. Packed-SIMD architectures supports unit-stride vector memory operations that can be easily supported by the general-purpose core's load-store unit without any modifications. Masking support is often restricted in packed-SIMD architectures to avoid changes to the datapath of the general-purpose core. Packed-SIMD architectures support wide variety of cross-element vector operations; these operations are often easier to implement for these architectures due to their restricted hardware vector lengths.

1.1.3 Next-Generation Vector Architectures

Between long-vector and packed-SIMD architectures, there lies a spectrum of compute systems for vector support. Recently, there has been an emerging trend towards next-generation vector architectures that provide unified abstraction across the wide compute systems spectrum. There are two possible design points for these next-generation vector micro-architectures: either an aggressive decoupled vector engine (similar to conventional long-vector machines), or a light-weight integrated vector unit (similar to conventional packed-SIMD units). There is a fundamental tension between performance and area among these next-generation vector micro-architectures: (1) integrated vector units achieve modest performance in accelerating regular data-parallel workloads while costing modest area overhead; (2) decoupled vector engines have significantly better performance at significantly higher area overhead. Here lies an interesting research question: Is it possible to achieve performance comparable to decoupled vector engines while requiring equivalent area overhead to integrated vector units?

1.2 Compute-In-Memory

Compute-in-memory (CIM) is a novel approach to reducing the area overhead associated with accelerating data-parallel kernels, offering a promising path to solving the tension between next-generation vector micro-architectures and addressing the research question that concluded Section 1.1: Is it possible to achieve performance comparable to decoupled vector engines while requiring equivalent area overhead to integrated vector units? Compute-in-memory has the potential to lower the area overhead in two ways: (1) fusing the compute elements with the storage elements and thus alleviating the need for highly multi-ported vector register files; and (2) the reconfigurability of the different components used in these proposals so their silicon can be utilized for different operations if vector is not needed.

CIM landscape, shown in Table 1.2, can be outlined in a taxonomy based on two dimensions: (1) the mechanism of in-situ computations between the operands, and (2) the technology of the underlying memory bit-cell. Previous work on CIM mainly explored three mechanisms of in-situ computations (i.e., boolean, multiply-and-accumulate, and associative) implemented on three memory technologies (i.e., SRAM, DRAM, and emerging non-volatile RAM). In boolean CIM, the

computation between the operands is some form of bit-wise logical boolean operation. Whereas in multiply-and-accumulate CIM, the inputs are multiplied with some weight and accumulated to compute the output. In associative CIM, on the other hand, the computations are achieved through a series of searches and multi-writes. For example, in addition, a series of searches and multi-writes emulate carry propagation and sum computation for each bit of the input.

1.2.1 Traditional Processing-Near-Memory

The ever-growing speed-mismatch between compute and memory in von Neumann machines causes the infamous *memory wall* [WM95, Wil95, McK04]. To overcome this memory wall, modern compute systems leverage a complex sub-system of on-chip hierarchical caches to lower the memory latency and increase the bandwidth. First wave of processing-near-memory (PNM) proposed placing compute closer to the data, thus overcoming the memory wall. By placing compute near the data, these proposals were able to achieve two important goals: (1) saving the energy required to move the data back-and-forth across the main memory bus to the compute units; (2) exploiting the abundant bandwidth available before the narrow main memory bus to increase parallelism. Intelligent RAM (IRAM) [PAC+97a, PAC+97c, PAB+97] was one of the early proposals to propose adding logic to DRAM in an effort to achieve the goals aforementioned. Furthermore, IRAM shows that vector abstraction constitutes a natural fit for PNM architectures. VIRAM1 [GWPK04] has further explored vector abstraction on a PNM architecture with embedded DRAM instead. DIVA [HKK+99] explored leveraging single-program multiple-data (SPMD) abstraction for PNM. While FlexRAM [KHY+99, FRF+03] explored SPMD abstraction for a PNM architecture with embedded DRAM.

1.2.2 Boolean Compute-in-Memory

Jeloka et. al. [JASB16, JASB15] introduced the bit-line compute technique in SRAMs, which performs digital bit-wise logical operation between rows. Bit-line compute constitutes an important precursor to subsequent SRAM-based compute-in-memory work. Compute caches [AJS+17] uses bit-line compute to transform the caches in a chip multi-processor (CMP) into bit-wise logical compute engines. Further work based on bit-line compute explored extending its functionality by utilizing a bit-serial approach to complex integer and floating-point operations. This work has

Computational Method	Memory Technology		
	SRAM	DRAM	eNVM
Boolean	Carefully tuned sense-amplifiers used to binarize a boolean operation performed on the bit-lines by accessing multiple rows. Bit-periphery compute is often used to expand the supported functionality beyond simple boolean functions.	Using a controlled hidden row, charge-sharing is altered to implement different majority boolean operations. Much like reads in DRAM, these operations are destructive to all accessed rows. Due to leakage, this approach is often very susceptible to noise and errors.	NVME can be used to implement this approach. The only difference with traditional SRAM and DRAM would be the design of the sense-amplifiers that binarize the boolean operations.
MAC	Multiply-and-accumulate can be implemented by accessing multiple rows and then tuning the sense-amplifiers to sense accumulation of accessed weighted values. Due to extremely high off-resistance of SRAM, often MACs are achieved through charge-based sensing.	Charge-sharing in DRAM reads is a natural fit for implementing charge-based multiply-and-accumulate.	Due to the ability of most NVME bit-cells to have variable resistivity, MACs can be implemented nicely as the value of the weight is encoded in the bit-cell resistivity and then the current induced by each accessed bit-cell can be summed for the final value.
Associative	Associative compute is achieved through searches and multi-writes. SRAM-based CAMs and TCAMs are used to provide the primitives necessary for associative compute. Multi-writes are often hard to implement and require some circuit work.	Performing searches in DRAM is challenging due to the destructive nature of the DRAM reads. However, theoretically, it is possible to implement associative compute on DRAMs.	Most NVME bit-cells constitute perfect match to implement CAMs due the lack of leakage and CAM-density. However, multi-write is more challenging in NVME because of its dependency and sensitivity to heat, current, spin-orbit direction, or charges.

Table 1.2: Compute-in-Memory Landscape – SRAM = synchronus random access memory, DRAM = dyanmic random access memory, eNVM = emerging non-volatile memory.

demonstrated the ability to use bit-line compute to transform caches in a CMP into fixed-function accelerators for neural networks [EWW+18] and single-instruction-multiple-threads (SIMT) engine [FMD19]. To mitigate transposing data, Wang et. al. [WWE+19] explored adding bit-line compute to an 8T-SRAM allowing the computation to be performed horizontally in the compute-bitline (CBL), while conventional data read and writes are performed on the vertical bitlines. The 8T bitcell hinders its use in traditional caches due to low density. VRAM [AHAA+20] explored utilizing bit-serial and bit-parallel execution paradigms to extend the functionality of bit-line compute.

Rowclone [SKF+13] is among the earliest work to explore transforming a DRAM into a DRAM-based compute-in-memory engine capable of in-DRAM row cloning. Ambit [SLM+17] proposed transforming the DRAM into an accelerator capable of massive bit-wise logical operations between multiple rows. DRISA [LNM+17] also explores transforming bit-lines in a DRAM memory into bit-wise logical units. ComputeDRAM [GTW19] explores implementing these techniques with entirely stock DRAM chips.

1.2.3 Multiply-and-Accumulate Compute-in-Memory

There are two methods to performing in-situ multiply-and-accumulate (MAC) computation: current-based, or charge-based. The design of the sense-amplifiers and the analog-digital converters (ADCs) mainly depend on which method is chosen. For charge-based in-situ MACs, a shared capacitor is pumped with charges controlled by gated switches. Then, the ADCs are configured to measure the voltage on the shared capacitor to calculate the output of the in-situ compute operation. Most work on charge-based in-situ MACs [BC18, LHS+21, SEN+21, ZWV17, LXK21, ZLW+22] use SRAM-based technology due to their bit-cell on/off impedance high ratio. For current-based in-situ MACs, multiple cells are accessed at the same time with each cell inducing a current on the common bit-line. The ADCs compute the output of the in-situ compute operation by measuring the current passing through the bit-lines. To implement current-based in-situ MACs, the on/off resistance of the bit-cell has to yield current values that can be sensed. Therefore, most work on current-based in-situ MACs [CLL+18, KCL+18] uses emerging non-volatile memory to implement bit-cells. This approach to compute-in-memory suffers from sensitivity to temperature and process variations. Some work proposed temperature compensation by adding a bias to offset the

temperature effect. Other work embrace the variability in the compute and implement approximate algorithms that can tolerate such variability.

1.2.4 Associative Compute-in-Memory

Associative compute was first proposed as a way to achieve compute-in-memory back in the mid-seventies [Fos76, Say76]. By using associative primitives (i.e., search, and multi-write), any boolean operation can be executed using a set of multi-writes to matched entries. The matches are set through bit-masked searches. Recent work [YMG15, CYS+21, ZL20, YEK18] has re-explored the premise with fresh perspective and using newer technology. This recent work propose using a content addressable memory (CAM) to perform a series of searches/multi-writes to execute boolean operations (e.g., addition, and multiplication). Conventionally, SRAM-based technology is used to implement CAMs. Whereas [YMG15] uses a traditional 12T-SRAM CAM, CAPE [CYS+21] leverages the techniques proposed by Jeloka et. al. [JASB16, JASB15] to enable CAM functionalities (i.e., search) thus mitigating the area overhead associated with traditional CAMs. Non-volatile emerging memory technology was shown to reduce the area overhead and energy of memory bit-cells thus yielding efficient CAMs (e.g., Hyper-AP [ZL20] shows a design using RRAM technology, and Yantır et. al. [YEK18] proposes using the memristor). In-situ associative CIM operations are executed by implementing their truth table through searches and multi-writes for each bit of inputs and outputs. As a result, associative CIM suffers from extremely long latencies.

1.3 Thesis Overview

This thesis explores the aforementioned tension between integrated vector units and decoupled vector engines for next-generation vector architectures, thus addressing the research question at the end of Section 1.1. The thesis leverages compute-in-memory to mitigate the inherent area overhead associated with vector machines in two ways: (1) the need for heavily multi-ported vector register file can be alleviated by fusing compute with storage; and (2) the re-configurability of the various components lowers the area overhead further. I focus on SRAM-based CIM due to its readily feasibility, as well as the prevalence and abundance of SRAM arrays in today's chips.

Chapter 2 explores extracting performance and efficiency in a post Dennard's Scaling era: by designing a fixed function accelerator. This chapter details the design and implementation of AgileBNN as part of the Celerity chip. AgileBNN is a fixed-function machine-learning accelerator for binarized neural networks designed using high-level synthesis (HLS) rather than the traditional register-level transfer (RTL) abstraction. Celerity is a 16nm FinFET chip that implements a tiered accelerator fabric (TAF) architecture, which has three tiers: general-purpose tier, specialization tier, and the massively parallel tier; AgileBNN represents the specialization tier in the TAF architecture. In this chapter, I detail my proposed high-level synthesis (HLS) flow for designing and implementing AgileBNN. Also, I propose the design of a memory unit completely in HLS to handle fetching and unpacking weights and outputs. AgileBNN has a sneakpath mode where the scratchpads in the manycore are reconfigured to store the weights for the machine-learning model and streamed to the AgileBNN through the sneakpath unit. I argue that AgileBNN demonstrates a conventional approach to extracting performance and efficiency in a post Dennard's Scaling era. Although AgileBNN was able to achieve respectful speedups and decent efficiency, my work finds that, by the time the chip was finished, the implemented algorithm was already outdated and the fixed-function aspect of AgileBNN was limiting its performance and efficiency. Nevertheless, the reconfigurability aspect of the manycore's scratchpads was very promising in lowering the area-overhead while increasing performance and efficiency and further motivates the work explored later in the thesis.

In order to study alternative designs for next-generation vector architectures, there is a need for baselines representing the conventional approach detailed in Section 1.1. As there were no publicly available cycle-approximate models for next-generation vector architectures when I started working on my thesis, I designed, implemented, and evaluated cycle-approximate models for an integrated vector unit and a decoupled vector engine. Chapter 3 discusses the design of these cycle-approximate integrated vector unit and decoupled vector engine models for next-generation vector architecture implemented in gem5 [BBB+11, LPAA+20, TCB18a]. The integrated vector unit design is loosely based on Arm's SVE units [SBB+17a, RBGZ19]. The decoupled vector engine, on the other hand, is loosely based on DEC's Tarantula [EAE+02]. In this chapter, I further study the different design points through the models, and I motivate the design-decisions through multiple studies. The studies were conducted by running multiple hand-vectorized applications and sweeping the various parameters for these models.

Chapter 4 explores the design and implementation of EVEv1, which is a vector execution unit that leverages SRAM-based boolean CIM techniques. In the chapter, I detail two design points for EVEv1: bit-serial EVEv1 (BS-EVEv1), and bit-parallel EVEv1 (BP-EVEv1). BS-EVEv1, as the name implies, adopts a bit-serial execution paradigm. Whereas BP-EVEv1, adopts a bit-parallel execution paradigm. There were multiple previous work that explored SRAM-based boolean CIM with bit-serial execution paradigm; however, BP-EVEv1 is, to my knowledge, the first work of its kind to explore bit-parallel execution paradigm for SRAM-based boolean CIM. In this chapter, I argue that bit-serial execution paradigm for SRAM-based boolean CIM yields high throughput, while bit-parallel execution paradigm achieves lower-latency for operations despite the cycle-time impact it incurs.

Chapter 5 details the design of ephemeral vector engines (EVE), which is the title of the thesis. EVE leverages the work detailed in Chapter 4 and other SRAM-based boolean CIM techniques to transform a private level-2 cache of a core into a decoupled engine for next-generation vector architecture. EVE implements the vector extension of RISC-V (RVV) [RISC-V Foundation19]. Due to the conclusion of EVEv1, to strike a balance throughput (bit-serial execution) and latency (bit-parallel execution), I propose a novel bit-hybrid execution that targets any parallelization factor (i.e., the number of bits processed in-parallel by the execution hardware) between bit-serial and bit-parallel. For this novel approach, I detail the new circuits proposal for EVE. Using a cycle-approximate model in gem5 [BBB$^+$11, LPAA$^+$20, TCB18a], I evaluate EVE with various parallelization factors on multiple hand-vectorized applications. In this chapter, I argue that the ideal parallelization factor for SRAM-based boolean CIM is somewhere in-between bit-serial and bit-parallel and strikes a balance between throughput and latency.

The primary contributions of this thesis are:

- I present highly configurable cycle-approximate models for a decoupled vector engine and an integrated vector unit, which serve as a tool to enable future research on next-generation vector architectures.
- I propose a novel template for ephemeral vector engines (EVE) circuits that can, at design time, be configured to target: bit-serial execution, bit-parallel execution, and any parallelization factor in-between.

- I propose ephemeral vector engines (EVE), a novel next-generation vector architecture that leverages SRAM-based compute-in-memory circuits to transform L2 caches on-the-fly into vector execution engines. I also present a holistic exploration from circuits to architecture of the EVE designs with different parallelization factors.
- To my knowledge, I am the first to make the case for bit-hybrid execution paradigm in SRAM-based compute-in-memory to balance throughput and latency.

1.4 Collaboration, Previous Publications, and Funding

I led the implementation of AgileBNN, presented in Chapter 2. To this end, I also developed an in-house HLS flow using Cadence's Stratus HLS and PyMTLv2. The System-C implementation of AgileBNN required for Cadence Stratus HLS was initially developed by Steve Dai, which in-turn was based on the C implementation contributed by Ritchie Zhao from his earlier work [ZSZ+17] that targets Xilinx Vivado HLS. The initial implementation of AgileBNN lacked any System-C structure, thus causing two major issues: (1) exorbitantly long synthesis time; (2) triggering multiple bugs in Stratus HLS causing it to generate incorrect RTL. I re-structured the System-C implementation to speed-up the synthesis. With guidance from Ritchie Zhao and insights about the BNN algorithm, I also re-wrote the System-C implementation to avoid Stratus HLS bugs, thus generating correct RTL. I performed co-simulation testing in Stratus HLS of AgileBNN verifying the RTL accuracy.

AgileBNN is part of the Celerity chip and constitutes its specialization tier. The Celerity chip was the fruit of the circuit realization at a faster time-scale (CRAFT) project. The CRAFT project was realized through the combined efforts of five different universities, and over twenty Ph.D. students spanning multiple geographical locations. The integration of AgileBNN into the Celerity chip was led by my colleague Christopher Torng. I helped significantly in the integration process by writing the software drivers for AgileBNN as well as testing the integration and verifying the RTL accuracy. With help from Christopher Torng, Austin Rovinski, and Professor Michael B. Taylor, I debugged the synthesized and the placed-and-routed top-level netlist of Celerity. The sneakpath switch unit was implemented by Christopher Torng in PyMTLv2 and I integrated it in my proposed flow effortlessly and was able to perform functional RTL testing, synthesized netlist testing, and placed-and-routed netlist testing. The software instrumenting the manycore to stream

the weights of the model to the AgileBNN through the sneakpath switch unit was developed by Scott Davidson, Shaolin Xie, and Chun Zhao from the Bespoke Silicon Group (BSG). Dustin Richmond, and Paul Gao led the silicon bring-up for the Celerity chip. I helped with the silicon bring up for AgileBNN by providing software to instrument the hardware to execute different models to explore the performance and accuracy trade-offs for AgileBNN. The work on AgileBNN has been published at the 38th volume of the IEEE Micro magazine [DXT+18]. I co-presented the work with my colleagues Scott Davidson and Austin Rovinski at the 29th IEEE Hot Chips Symposium, 2017 (HotChips-29) [AAHA+17]. The energy and performance characterization has been published at 2nd volume IEEE Solid-State Circuits Letters (SCCL) [ea19c].

I collaborated with Helena Caminal, Professor Christopher Batten, and Professor José Martínez on the content addressable processing engine (CAPE) project. I helped in the front-end implementation of the vector instructions in gem5 [BBB+11, LPAA+20, TCB18b]. I led the initial effort in implementing the cycle-approximate model of CAPE in gem5 connected to the MinorCPU, which is an in-order core model. For the connection between the core and the model, I developed my first implementation of the Rocket chip co-processor (RoCC) interface in gem5. I also led the implementation effort on a Python-based emulation infrastructure for CAPE's compute primitives. When the team began working on the CAPE project, there was no compiler support for RISC-V Vector extension (RVV); to that end, I worked on modifying the GNU compiler collection (GCC) and the GNU binary utilities (binutils) to support RISC-V vector inlined instructions. The work has been published and presented by the lead of the project, Helena Caminal, at the 27th IEEE International Symposium on High-Performance Computer Architecture, 2021 (HPCA-27) [CYS+21].

I led the effort in the Batten Research Group (BRG) to implement cycle-approximate models for a decouple vector engine and integrated vector unit. This effort would not have been possible with the hard work of: Tuan Ta, Nick Cebry, and Yanghui Ou. Tuan Ta helped with guidance and tips for implementing a specialized port in L2 caches to connect the decoupled vector engine in the traditional memory model of gem5 as well as conventional Ruby memory model. When the project switched to Arm's CHI for the Ruby protocol, Tuan contributed the code to implement the specialized port in L2. Nick Cebry implemented stats in the gem5's front-end to collect statistics about vectorized applications execution. He also worked on implementing the assembly testing suite for vector instructions used to verify the functionality and execution of the models for the

decoupled vector engine and the integrated vector unit. Both Nick Cebry and Yanghui Ou worked on implementing enhanced modeling of functional unit in the decoupled vector engine model.

I was fortunate enough to lead BRG's effort on the EVE project. The EVE project contains two parts: (1) EVEv1, which is a predominantly circuit effort; and (2) EVEv2, which is a circuit-to-architecture exploration of the project incorporating the findings from EVEv1. This project with both its parts would not have been possible without the hard work of my collaborators: Tuan Ta, Nick Cebry, Olalekan Afuye, Dr. Shady Agwa, Eric Hall, Courtney Golden, Professor Al Molnar, Professor Alyssa Apsel, and Professor Christopher Batten.

Olalekan Afuye, Dr. Shady Agwa, Professor Al Molnar, and Professor Alyssa Apsel primiarly contributed to the first part of the EVE project (i.e., EVEv1). Olalekan Afuye ported OpenRAM to TSMC's 28nm. He also helped with power and energy analysis. Shady Agwa laid out the netlists of multiple stages. Shady also helped in the discussion about circuit design and gave important feedback. Professor Al Molnar provided helpful guidance and feedback for designing and implementing the reconfigurable sense-amplifiers. Finally, both Olalekan Afuye and Shady Agwa helped in creating an OpenRAM module for EVEv1. I presented this work at the 2020 IEEE International Symposium on Circuits and Systems (ISCAS) [AHAA$^+$20].

Tuan Ta, Nick Cebry, Eric Hall, Courtney Golden, Olalekan Afuye, and Professor Alyssa Apsel contributed to the second part of the EVE project (i.e., EVEv2). Tuan Ta had the aforementioned contribution in the decoupled vector engine model and the integrated vector unit baselines for EVE. His work on Arm's CHI modification for the decoupled vector engine model also contributed to EVE's memory model. Nick Cebry work on vector assembly testing for the decoupled vector engine and the integrated vector unit models was used to further test EVE's cycle-approximate model. Tuan Ta, Nick Cebry, Eric Hall, and Courtney Golden hand-vectorized applications from RiVEC and Rodinia application suite that helped benchmark EVE and further understands its performance. Olalekan Afuye helped in running the experiments to estimate the cycle-time for the different bit-hybrid circuits.

I collaborated with Tuan Ta who was the lead of the big.VLITTLE effort in BRG. I contributed the decouple vector engine and integrated vector unit baselines. I also helped in debugging and fixing the big.VLITTLE model. Moreover, I contributed in the discussion of several intellectual aspects of big.VILITTLE. The work will be published and presented at the 55th IEEE/ACM International Symposium on Microarchitecture, 2022 (MICRO-55) [TAHC$^+$22].

The work in this thesis was supported in part by: King Abdullah's Scholarship Program (KASP), King Salman's Scholarship Program (KSSP), NSF E2CDA Award #1740136, NSF SHF Award #2008471, NSF PPoSS Award #2118709, DARPA SDH Award #FA8650-18-2-7863, DARPA CRAFT Award #HR0011-16-C-0037, the Semiconductor Research Corporation (SRC) as nCORE task 2758.002 and 2758.004, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, as well as equipment, tool, and/or physical IP donations from Intel, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. 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
AGILEBNN: TRADITIONAL APPROACH TO ACCELERATOR DESIGN

This chapter describes my early work at achieving performance and efficiency by designing AgileBNN, which is a fixed function accelerator for binarized neural networks. AgileBNN was integrated into the Celerity tiered accelerator fabric (TAF). Tiered accelerator fabrics seek to address the tension between less-efficient general-purpose architectures and more-efficient specialized architectures by tightly integrating a heterogeneous mix of: (1) general-purpose tier (GPT)–a set large general-purpose cores, (2) massively parallel tier (MPT)–an array of small lightweight cores, and (3) specialization tier (SpT)–a set of specialized fixed-function accelerator. A workload can be efficiently accelerated using a tiered accelerator fabric by mapping it to the three tiers appropriately. AgileBNN was taped-out as the specialization tier in the Celerity TAF on a Taiwan Semiconductor Manufacturing Company (TSMC) 16 -nm technology node. Celerity TAF also includes five RISC-V cores capable of booting an operating system for the general-purpose tier, and 496 lightweight RISC-V cores for the massively parallel tier.

In this chapter, I detail the novel high-level synthesis (HLS) flow used to design, synthesize, and test the AgileBNN. I also include seven lessons-learned that can provide useful guidance for future work on tiered accelerator fabrics as well as using HLS. Moreover, I present post-silicon measurement results of AgileBNN's accuracy, area, throughput, and energy. AgileBNN reduces area by over $2\times$ and improves area normalized throughput by almost $2\times$ compared to three other state-of-the-art BNN accelerators. While AgileBNN is able to improve area normalized throughput, the rigidity of its fixed-function aspect meant that further modifications to the executed BNN model were heavily restricted. AgileBNN inspired and motivated several aspects of my later work on EVE. AgileBNN rigidity motivated the pursuit of a more programmable specialization, while the ability to re-use silicon blocks for several functionalities to reduce effective area overhead inspired the pursuit of ephemerality.

2.1 Introduction

Machine-learning (ML) algorithms are rapidly evolving, which complicates efforts at implementing specialized accelerators for such workloads. Developing an accelerator ASIC for a spe-

cific ML algorithm can take years, by which time the ML algorithm has likely evolved to such an extent that it can no longer take advantage of the accelerator. Developing a highly programmable accelerator means the long design-time can be amortized over a larger domain of ML algorithms, but such accelerators sacrifice energy efficiency and/or throughput. In our previous work, we argued for a new architectural design paradigm to address these challenges based on *tiered accelerator fabrics* [DXT+18]. A tiered accelerator fabric includes three heterogeneous tiers: the *general-purpose tier* with a set of cores capable of running an operating system to manage the system-on-chip (SoC); the *massively parallel tier* with an array of small, lightweight, tightly coupled programmable cores tuned for efficient throughput computing; and the *specialization tier* with high-performance, ultra-energy-efficient fixed-function accelerators.

Figure 2.2 shows the Celerity architecture, an instantiation of the tiered accelerator fabric paradigm [DXT+18]. Celerity's general-purpose tier includes five Linux-capable RISC-V RV64GC rocket cores [ea19d]. Celerity's massively parallel tier includes 496 Vanilla cores arranged in a 16×31 two-dimensional array. Vanilla cores are lightweight RISC-V RV32IM cores designed in-house and connected together using a high-throughput network-on-chip. Each core contains scratchpad memories for data and instruction storage. For the specialization tier, we chose to focus on accelerating image classification using emerging ML algorithmic research on binarized neural networks. Figure 2.1 shows the Celerity SoC which implements the Celerity architecture in a 5×5 mm 385M-transistor chip fabricated on an aggressive 16 nm FinFET technology node. Our prior work included preliminary simulation results for Celerity [DXT+18] and post-silicon measurement results for Celerity's massively parallel tier [ea19c, ea19b].

This chapter is the first to focus exclusively on Celerity's specialization tier, which we call AgileBNN. AgileBNN is a configurable accelerator for binarized neural networks implemented using an agile SystemC-based high-level-synthesis (HLS) methodology. In this chapter, we describe the AgileBNN design and integration in detail, and we include seven lessons learned that can provide useful guidance for future work on tiered accelerator fabrics: (1) SystemC-based HLS enabled a modular design methodology; (2) HLS enabled early and rapid algorithmic design-space exploration; (3) medium-grain integration enabled balancing integration complexity and overhead; (4) hardware/software co-design enabled a more configurable and complexity-effective accelerator; (5) latency-insensitive accelerator interfaces simplified integration with variable-latency cache-

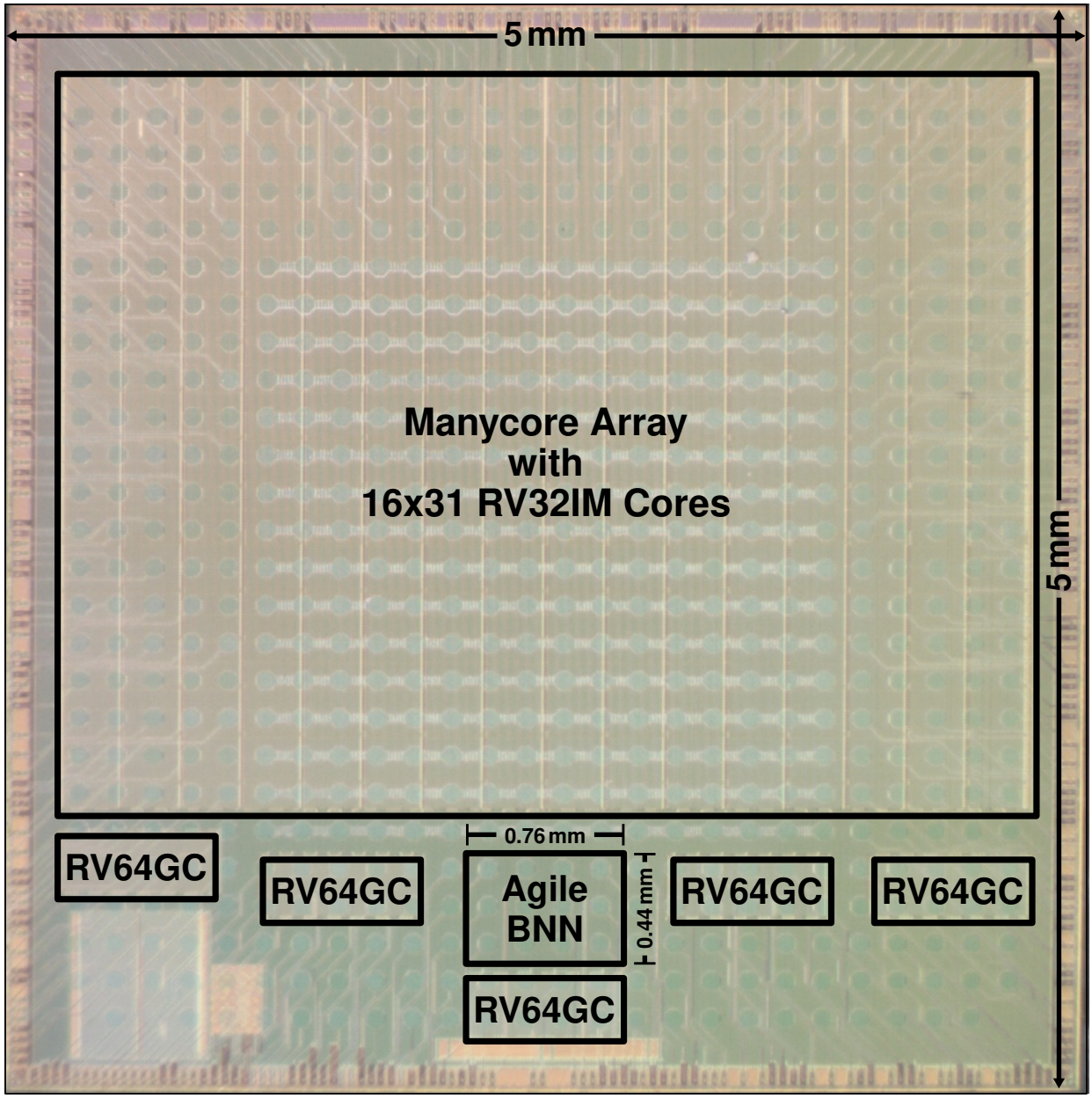


Figure 2.1: Celerity System-on-Chip Die Photo.

based memory systems; (6) cross-tier optimization enabled multiplicative benefits; and (7) HLS enabled rapid timing closure late in the design cycle.

This chapter presents new post-silicon measurement results of AgileBNN's accuracy, area, throughput, and energy. AgileBNN achieves a classification throughput of 375 inf/s yet requires only 0.33 mm^2 of additional area while maintaining an accuracy of 88.4% on CIFAR-10. At its most efficient operating point, AgileBNN consumes 141 µJ/inf with a throughput of 17 inf/s. Our design and integration strategy enables AgileBNN to reduce area by over 2× and improve area normalized throughput by almost 2× compared to three state-of-the-art BNN accelerators that required either manual RTL design or full-custom mixed-signal design. While there has been significant academic work on evaluating HLS-based accelerators implemented in an FPGA, AgileBNN is one of just a few examples of an academic HLS-based accelerator implemented in an ASIC on an aggressive technology node.

2.2 AgileBNN Design

There is a growing interest in reduced precision neural network models with recent work demonstrating neural networks with binarized weights and activations achieving reasonable accuracies [CHS+16, ea19a]. This motivated our own interest in exploring an accelerator for image classification using binarized neural networks (BNN). Specifically, we trained a BNN model based on Courbariaux et al. [CHS+16]. By binarizing the weights and activations, convolution becomes a simple exclusive-negated-OR (XNOR) followed by a pop-count rather than multiplications followed by a reduction summation. Not only is a BNN model hardware friendly in-terms of operations, it has the potential to reduce the size of the weights thus reducing storage overhead. The BNN model constitutes of: a 20-bit fixed-point convolution layer, six binarized convolution layers, and three binarized fully-connected (dense) layers. Our trained model is able to achieve 89.8% accuracy on the CIFAR-10 dataset [DXT+18, ZSZ+17].

AgileBNN includes the control unit, the compute unit, and the memory unit (see Figure 2.2). The *control unit* is responsible for maintaining the configuration registers that hold values required to execute a given model. Once the configuration values are set, the control unit orchestrates the execution inside the accelerator. The *compute unit* includes the functional units for: the fixed-point convolution layers, the binary convolution layers, and the binary fully-connected layers. The

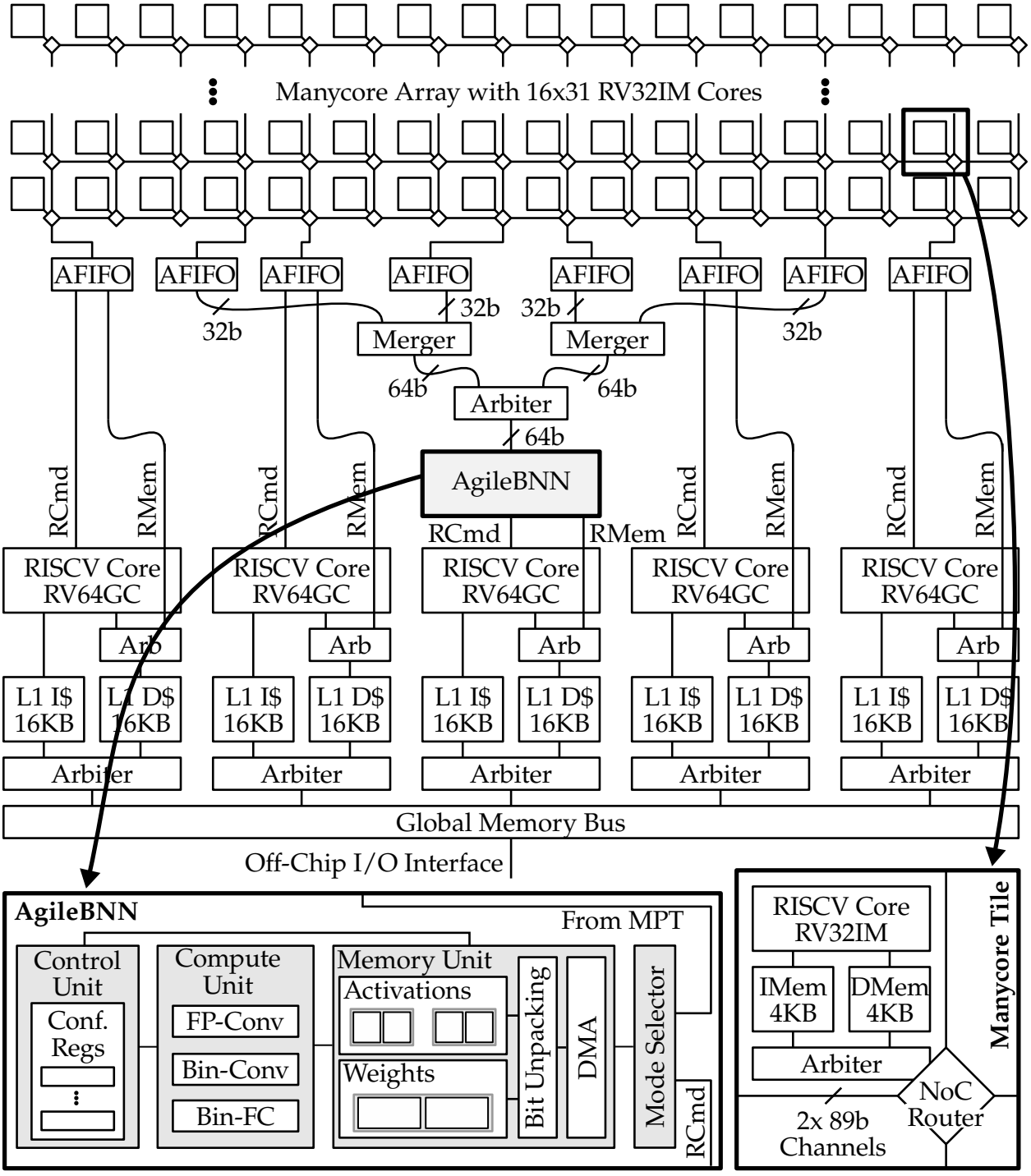


Figure 2.2: Celerity Architecture – AFIFO = asynchronous FIFO; RCmd = RoCC command interface; RMem = RoCC memory interface; NoC = network-on-chip; L1 I$/D$ = instruction/data cache; IMem/DMem = instruction/data memory; MPT = massively parallel tier.

compute unit can execute any of these layer types with configurable input sizes and output sizes. The *memory unit* fetches weights and input activations from main memory and streams them to the compute unit in the correct order. A direct memory access (DMA) subunit was built into the memory unit to handle fetching blocks of data. Each 64-bit incoming chunk of data is unpacked into individual elements (i.e., bits for binarized layers, 20-bit for the fixed-point layers) and placed into an SRAM bank for later use. To supply data to two convolvers, the memory unit divides its weight and data storage across two banks. The two banks are either decoupled to supply two binary convolvers or are coupled as one wide bank to supply one fixed-point convolver. There are two sets of banks for activations (for double buffering) and one set of banks for weights.

To implement AgileBNN, we started from unstructured C++ code that implemented the BNN algorithm. We experimented with using HLS to directly synthesize this unstructured code, yet the corresponding register-transfer-level (RTL) implementation produced disappointing area and timing results which were difficult to reason about and optimize. We then transformed our original C++ implementation into a more structured SystemC-based implementation with key hardware modules implemented using SystemC modules (e.g., memory unit, compute unit). The communication and synchronization between these different modules are implemented with the help of SystemC's hierarchical channels. The generated RTL implementation of AgileBNN is further wrapped using an in-house PyMTL [LZB14] wrapper to interface with the rest of the SoC. This SystemC-based implementation and testing required a two-person team working for 12 weeks. About six weeks were spent debugging and working around a subtle bug related to applying HLS to complex multi-dimensional variable array indexing. As the HLS tools continue to improve, we expect designers to spend more time on design-space exploration and less time working around various HLS tool issues.

Lessons learned from using an HLS flow to design AgileBNN:

- Using a SystemC-based HLS flow to implement AgileBNN introduced structure into the implementation, which drastically increased the ability to reason about design decisions. Compared to our original unstructured C++ code, SystemC quickly proved to be a natural candidate for synthesizing accelerators.

- HLS enabled us to quickly iterate over the possible design space comparing different ideas and quantitatively evaluating design decisions. HLS enables designers to focus on the algo-

rithm and high-level architecture of the design rather than being distracted with the implementation details.

2.3 AgileBNN Integration

There are multiple approaches to integrating a specialized accelerator into an SoC spanning fine-, medium-, and coarse-grain integration. In *fine-grain integration*, the accelerator is brought closer to the control processor, potentially even coupling it tightly as a functional unit. In *medium-grain integration*, the accelerator is placed further from the processor yet still shares a private data cache and likely uses a decoupled request/response command interface. In *coarse-grain integration*, the accelerator is completely separate from the processor and has its own interface to main memory for communication and task offloading. On one end of the spectrum, fine-grain integration yields minimum overhead in task offloading at the expense of being highly intrusive to the processor pipeline. On the other end of the spectrum, coarse-grain integration incurs high overhead in task offloading with the benefit of being largely isolated from the processor. For AgileBNN, we chose medium-grain integration to strike a balance between the overhead of task offloading and intrusiveness to the processor pipeline. By leveraging the Rocket custom co-processor (RoCC) interface, AgileBNN can be integrated in a medium-grain fashion without spending any effort modifying the Rocket processor pipeline. AgileBNN can also benefit from low overhead task offloading to utilize the Rocket core as a configuration control processor.

AgileBNN is integrated into the Celerity SoC through one of the five Rocket cores available in the general-purpose tier. For a given neural network model, software is developed to calculate the appropriate values for the various configuration registers in AgileBNN. A RISC-V custom instruction provides read/write access to these configuration registers through the RoCC command interface.

The configuration software running on the Rocket core prepares the input weights and activations for each layer. Then, the memory addresses for all input data are placed in their designated configuration registers. As AgileBNN starts execution of the specified layer, the DMA in the memory unit starts sending read requests through the latency-insensitive RoCC memory interface. The read requests are serviced through the L1 cache of the Rocket core and once ready the responses are sent back out-of-order. After re-ordering the responses, the memory unit unpacks the incoming

data into their corresponding internal storage. Sharing the L1 data cache significantly simplifies task offloading.

Loading weights from main memory is expensive due to off-chip accesses. AgileBNN addresses this challenge by providing a collaborative mode that leverages the scratchpads in the idling massively parallel tier to store BNN weights. A light-weight manycore program is written to run on the massively parallel tier to send the weights in-order to AgileBNN through a hardware FIFO. The wrappers for AgileBNN are designed to start dequeuing from this FIFO whenever the requested address is within a pre-mapped address space. This execution mode allows the Celerity SoC to load the weights once from the main memory and reuse them multiple times for every image classification.

After integrating AgileBNN into the Celerity SoC and testing its functionality in simulation, the last step is to analyze timing in order to meet the SoC timing requirements. The HLS-generated AgileBNN along with its wrappers were pushed through synthesis without the rest of the SoC to better understand and resolve the critical path. Meeting a specific timing constraint was easier than conventional RTL flows; the HLS tool handled automatic pipelining to meet the desired timing constraint. Note that SRAM floorplanning was also an important step in meeting more aggressive timing constraints. The use of latency-insensitive interfaces helped in dealing with meeting timing constraints at the input and output ports. Input and output paths from the accelerator that violate timing were pipelined late in the design cycle without impacting functional correctness. A two-person team spent roughly four weeks integrating AgileBNN into the SoC performing debugging, software development, and meeting design constraints.

Lessons learned from integrating HLS-generated AgileBNN into the Celerity SoC:

- Medium-grain integration allowed us to strike a balance between the task offloading overhead and the effort required for the integration. By leveraging the RoCC interface, medium-grain integration required relatively modest effort.

- Hardware/software co-design in implementing the execution of BNN models helped reduce AgileBNN complexity. The configuring software can utilize the general-purpose processor to prepare input data, calculate complicated model parameters, and synchronize various operations between layer execution.

- Latency-insensitive interfaces in AgileBNN helped simplify interacting with variable latencies in the Rocket cache-based memory system. They also helped in meeting timing constraints by enabling additional pipelining late in the design cycle.

- Cross-tier optimization in the Celerity SoC enabled us to extract throughput beyond the allocated area for AgileBNN by leveraging idle storage in the massively parallel tier. Collaborative execution between tiers often yields multiplicative improvements.

- HLS enabled us to meet timing constraints effectively late in the design cycle. HLS is able to leverage the high-level information represented in the SystemC code to generate higher quality-of-result RTL.

2.4 AgileBNN Evaluation

This section discusses the methodology we used in evaluating AgileBNN, characterizes the AgileBNN chip's different performance and efficiency operating points, and compares AgileBNN with previous work in key metrics.

2.4.1 Methodology

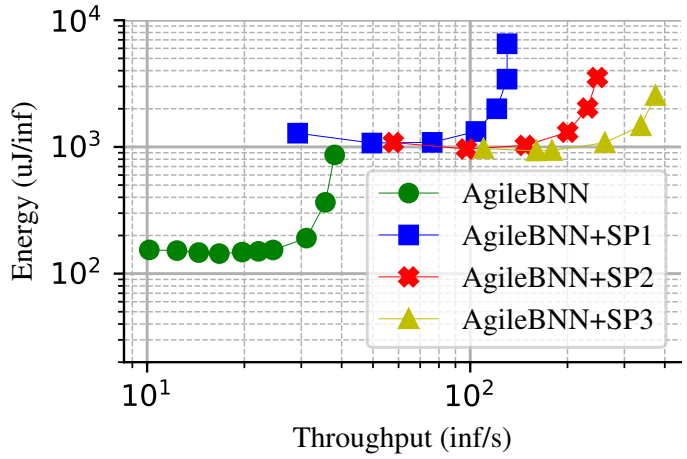
The methodology used to evaluate AgileBNN spans multiple important aspects for any machine-learning hardware including accuracy, area, throughput, and energy.

Accuracy: We trained a BNN model based on previous work achieving a training accuracy of 89.8% on CIFAR-10 dataset. A standard testing dataset was used to measure the classification accuracy of AgileBNN running on the chip.

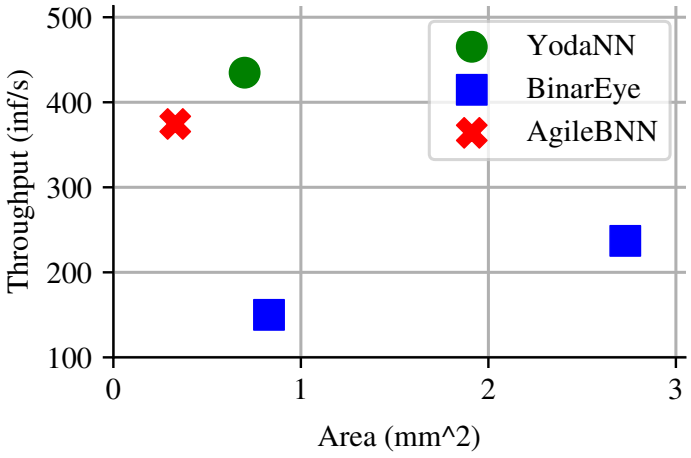
Area: The area of AgileBNN, including the wrappers, was measured from the sign-off layout.

Throughput: To measure the throughput of AgileBNN, we utilize cycle count performance counters on the Rocket core. The throughput is simply the number of classifications performed per second and can be calculated given the size of the dataset and the total time.

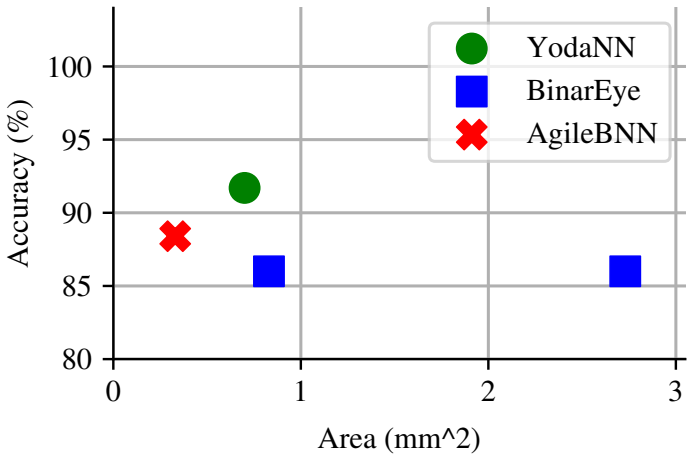
Energy: To measure energy, we constructed a power model that excludes leakage induced by other components on the Celerity SoC. We estimated the leakage for the SoC by halting the clock tree globally and measuring the power consumed by the chip using a Keithley source measurement



(a) Chip Voltage/Freq Characterization.



(b) Throughput vs. Area.



(c) Accuracy vs. Area.

Figure 2.3: Silicon characterization of energy, throughput, area, and accurcy.

unit. We estimated the leakage for AgileBNN by considering its relative area compared to the entire SoC. To calculate total energy, we crafted a software test program that executes classifications for a guaranteed 10 seconds to obtain average power. Energy per classification is calculated from the throughput and the average power consumed.

2.4.2 Chip Characterization

As a classification utilizes multiple components in its execution, there is a space defined by the different operating points for the SoC, where every point exhibits a different efficiency/throughput trade-off. Figure 2.3a shows a sweep to capture these trade-offs. The AgileBNN curve shows the throughput vs. energy of classification as the voltage is varied from 0.55–0.82 V and frequency from 31.25–845 MHz. AgileBNN+SP curves show the throughput vs. energy sweep for collaborative execution (i.e., storing BNN weights in the massively parallel tier). By fixing the AgileBNN operating point, each curve is calculated by sweeping the massively parallel tier frequency from 15.625–1000 MHz. The three AgileBNN+SP curves correspond to the following AgileBNN frequencies: 250 MHz, 500 MHz, and 845 MHz.

By excluding off chip energy, Figure 2.3a shows that AgileBNN alone is more efficient and achieves lower throughput when compared to the collaborative execution (AgileBNN+SP) mode. However, it is worth noting that as energies for off chip accesses are accounted for, collaborative execution would be more efficient as it requires accessing weights off chip once. We can also notice that the benefits of increasing the frequency of AgileBNN stops at 125 MHz, after which memory latencies become the bottleneck. As for the collaborative execution (AgileBNN+SP) mode, the plot shows that clocking the massively parallel tier higher benefits AgileBNN by reducing latencies for loading weights. Once the frequency of the massively parallel tier reaches half the frequency of AgileBNN, the computations start becoming the bottleneck and throughput saturates. Since the massively parallel tier has two 64-bit asynchronous FIFOs providing the weights (meaning two streams of weights), half the frequency is sufficient for the massively parallel tier to saturate the compute throughput of AgileBNN.

Paper	AgileBNN **This Work**	YodaNN TCAD'17 [ea18]	BinarEye-D CICC'18 [MBY+18]	BinarEye-MS JSSC'19 [BYM+19]
Design Methodology	HLS/stdcell	RTL/stdcell	RTL/stdcell	Mixed-Signal
Technology	16nm FinFET	65nm CMOS	28nm CMOS	28nm CMOS
Supply Voltage (V)	0.54 - 0.83	0.6 - 1.2	0.66 - 0.9	0.6, 0.8
Area (mm^2)	0.33	2.2	1.40	4.6
Norm. Area (mm^2)	0.33	0.7	0.83	2.73
Frequency (MHz)	$13-845$	$27-480$	$1.5-48$	10
Weight Precision (bits)	1b	$1b \times 12b$	1b	1b
On-Chip Memory	71kB SRAM	9.2kB SCM	328kB SRAM	328kB SRAM
Supported Networks				
Layer Count	Any	Any	$1-16$	$1-16$
Filter Count	Any	Any	64, 128, 256	256
Filter Sizes	3×3	$< 7 \times 7$	2×2	2×2
Accuracy	88.40%	**91.70%**	86.00%	86.05%
Max. Throughput (inf/s)	375	**435**	150	237
Energy Efficiency (uJ/inf)	141.21	21	13.82	**3.79**
Area Norm. Throughput (inf/s/mm^2)	**1,136.36**	621.43	180.72	86.81

Table 2.1: Comparison to Previous Work – Normalized area scaling factor calculated by squaring the contacted gate pitch (CPP) of equivalent node. All BinarEye designs can keep the weights and activations on-chip with no need for off-chip accesses; SCM: latch-based standard cell memory.

2.4.3 Comparison to Previous Work

We evaluated AgileBNN against three other state-of-the-art works that accelerate binarized neural networks (see Table 2.1). AgileBNN has the highest area efficiency, 1,136 inf/s/mm^2, as demonstrated in Figure 2.3b. By leveraging the massively parallel tier for weight storage, AgileBNN is able to increase its throughput while still maintaining a small footprint. AgileBNN achieves pareto-optimal accuracy and area, as well, (shown in Figure 2.3c).

YodaNN [ea18] is an accelerator for neural networks that uses binarized weights and 12-bit fixed-point activations. Due to higher precision activations, YodaNN achieves 3.3% higher accuracy on CIFAR-10 classification compared to AgileBNN. YodaNN exploits aggressive parallelism (employing 32 compute units) yet only achieves 16% higher throughput compared to AgileBNN at the expense of increasing area by $2\times$.

Digital BinarEye [MBY+18] (BinarEye-D) is an accelerator that implements a binarized neural network model inspired by Courbariaux et al [CHS+16]. Although the BNN model used is similar to AgileBNN, BinarEye-D has lower accuracy as it substitutes two of the fully-connected layers with binary convolution layers. Such change allows the design to achieve lower energy consumption and lower storage overhead, but at the expense of lower accuracy. AgileBNN achieves 2.5× higher throughput. BinarEye-D is the closest accelerator to AgileBNN, as it uses an RTL/standard-cell design methodology and a similar BNN model.

Mixed-signal BinarEye [BYM+19] (BinarEye-MS) is a mixed-signal implementation of BinarEye-D. Instead of performing convolution digitally, the result of binary multiplication is stored as a charge on a set of capacitors. Then, the wide 1-bit vector summation with added bias is estimated using a capacitive digital-to-analog converter (CDAC). While BinarEye-MS is able to increase throughput by almost 60% compared to BinarEye-D, AgileBNN is still able to achieve 1.6× higher throughput. BinarEye-MS retains the same accuracy as BinarEye-D, with the caveat that temperature and process variation can now affect the classification accuracy. The use of CDACs and capacitors adds an extra area overhead compared to BinarEye-D. Also, implementation effort for mixed-signal designs can be significantly higher that manual RTL design.

As for energy, AgileBNN suffers from overheads due to memory accesses going through the SoC memory subsystem. There is some energy overhead due to the use of HLS design methodology instead of traditional RTL, which we believe is overshadowed by the overhead of the SoC memory subsystem. Moreover, the energy consumed by AgileBNN is dwarfed by the energies consumed by the full SoC. Both flavors of BinaryEye are the most efficient with the mixed-signal design consuming 5× lower energy per inference compared to other work.

2.5 Conclusion

In this chapter, we have presented silicon results and characterization for AgileBNN, which is an HLS-generated binarized neural network accelerator integrated into the Celerity SoC and fabricated on a 16 nm FinFET technology node. AgileBNN reduces area by over 2× and increases area normalized throughput by almost 2× when compared to the state-of-the-art BNN accelerators. As transistor scaling slows down, accelerators are becoming a crucial part of SoC design to achieve throughput and energy efficiency targets. HLS-generated accelerators can help SoC

designers achieve better composition by enabling rapid design space exploration through reduced implementation effort. Despite the demonstrated improvements in area-normalized throughput for AgileBNN, its rigid specialization restricted any modifications to the BNN model that would improve throughput and efficiency. This limitation motivated the choice for a more programmable vector abstraction for EVE. Nevertheless, the ability for the sneakpath mode in AgileBNN to reconfigure several components to increase throughput and performance thus retaining lower area overhead motivated the ephemerality aspect of EVE.

CHAPTER 3
MODELING NEXT-GENERATION VECTOR MICRO-ARCHITECTURES

With their rise in popularity, emerging next-generation vector architectures form a fertile ground for research. The vector abstraction supported by these architectures balances programmability and specialization, which alleviates the rigidity observed in AgileBNN detailed in Chapter 2. Vector engines exploit the regularity inherent in vector abstraction to increase their performance and efficiency while executing data-parallel workloads. Identifying and modeling baselines constitute a fundamental aspect of architectural research. This chapter details my work on designing, implementing, and evaluating two baselines for next-generation vector micro-architectures: a decoupled next-generation vector engine, and an integrated next-generation vector unit. These models have been essential in guiding and evaluating my later work on EVE.

3.1 Introduction

Early implementations of vector machines [Wat72, Sch87, BDM+72, Rus78] demonstrated their ability to increase performance and efficiency while executing data-parallel workloads by exploiting the regularity in the vector abstraction. There has been a resurgence of interest in vector abstraction demonstrated by recent vector extensions to mainstream ISA's (e.g., ARM SVE [Ste16, SBB+17b], RISC-V RVV [RISC-V Foundation19], Intel AVX-512 [int12]). Traditionally, there has been two micro-architectures to support vector execution: aggressive long-vector engines [DVWW05, KTHK03, TNH+06, EAE+02], and light-weight sub-word packed single-instruction-multiple-data (SIMD) units [SBB+17a, RBGZ19, PW96, int07, JKK11]. Long-vector engines achieve high performance at the expense of area overhead. Since these vector engines lose their performance and efficiency edge while executing irregular code, packed-SIMD units has been proposed to retain moderate vector performance and efficiency while incurring minimum area overhead. Packed-SIMD units are able to lower their area requirements through: (1) repurposing hardware from the datapath of a general-purpose core to support vector execution, (2) supporting short hardware vector lengths, and (3) supporting limited vector execution. While both long-vector engines and packed-SIMD units are micro-architectures to support vector execution, they represent two extreme ends.

There has been a recent effort towards next-generation vector architectures [Ste16, SBB+17b, RISC-V Foundation19] to unify both extremes (i.e., long-vector, and packed-SIMD). There are two possible next-generation vector micro-architectures: an integrated vector unit, or a decoupled vector engine. Integrated vector units follow the same design philosophy as sub-word packed-SIMD units, while decoupled vector engines follow the same design principles as long-vector engines. By repurposing hardware from the general-purpose core's datapath, integrated vector units achieve modest performance while incurring minimum area overhead. On the other hand, decoupled vector engine achieve significantly better performance at the cost of significantly high area overhead. Both integrated vector units and decoupled vector engines form a natural baselines for any research in next-generation vector architectures.

When I started working on my thesis, there was no publicly available models for next-generation vector micro-architectures. Recently, there has been a multiple works that target implementing next-generation vector micro-architectures with publicly available models. There models are register-transfer-level and often the work

3.2 RISC-V Vector Extension (RVV)

The RISC-V Vector extension (RVV) went through multiple revisions. This section provides an overview of the first public release of RVV (i.e., revision 1.0). The instructions of the RVV extension can be divided into four categories: vector configuration, vector arithmetic, vector memory, and vector cross-element. RVV allows for these instructions to be predicated through masking support.

3.2.1 Vector Configuration Instructions

RVV provides multiple instructions that can be used to configure the hardware used for vector execution. One aspect is setting the active vector length through the `vsetvl` instruction. The semantic of the `vsetvl` instruction is as follows: given the requested vector length as an input, the new active vector length is the maximum between the hardware vector length and the requested vector length. Another aspect that can be configured is the active element width for the vector hardware. The number of available vector registers can also be configured dynamically through

the length multiplier (LMUL). By increasing the length multiplier by one, the number of vector registers is halved, but the hardware vector length supported by the remaining vector registers is doubled.

3.2.2 Vector Arithmetic Instructions

The instructions in this category perform a simple dot-operation between two source operand. The resulting values of the dot-operation is stored into a destination operand. Source operands can be of three possible types: vector, scalar, and immediate. Vector operands represent a vector register; scalar operands represent a scalar register; immediate operands are encoded as constants in the instruction. For instructions with two source operands, there are three possible source operand configuration: vector-vector, vector-scalar, and vector-immediate. To distinguish each configuration, a suffix is added to the instruction name. Vector-vector (suffix .vv) arithmetic instructions operate on two operands of type vector. Here is the pseudo-code semantics of a vector-vector arithmetic operation (vop.vv):

```
1    def vop.vv(vd, vs2, vs1):
2
3      vlen = active_vector_length
4
5      for (i = 0; i < vlen; i++) {
6        vd[i] = op(vs2[i], vs1[i])
7      }
```

Vector-scalar (suffix (.vx) arithmetic instructions operate on two operands: a vector operand and a scalar operand. Here is the pseudo-code semantics of a vector-scalar arithmetic operation (vop.vx):

```
1    def vop.vx(vd, vs2, rs1):
2
3      vlen = active_vector_length
4
5      for (i = 0; i < vlen; i++) {
6        vd[i] = op(vs2[i], rs1)
7      }
```

Vector-immediate (suffix `.vi`)arithmetic instructions operate on two operands: a vector operand and an immediate operand. Here is the pseudo-code semantics of a vector-immediate arithmetic operation (`vop.vi`):

```
def vop.vi(vd, vs2, imm):

  vlen = active_vector_length

  for (i = 0; i < vlen; i++) {
    vd[i] = op(vs2[i], imm)
  }
```

3.2.3 Vector Memory Instructions

The instructions in this category perform either a load from memory into a vector register, or a store to memory from a vector register. Memory vector load instructions read elements from the memory and write to the elements of the destination vector register. Memory vector store instruction store elements from the source vector register into elements in the memory. The size of the elements being loaded and stored is specified and encoded into the instruction. RVV supports element sizes of 8, 16, 32, and 64 bits. There are three possible memory addressing modes for these instructions: unit-stride, constant-stride, and indexed-stride.

Unit-stride is the simplest of all modes and assumes a contiguous block of data in the memory. Here is the pseudo-code of the 32-bit unit-stride vector memory load instruction (`vle32.v`):

```
def vle32.v(vd, base_address):

  vlen = active_vector_length
  addr = base_address
  elem_size = 4

  for (i = 0; i < vlen; i++) {
    vd[i] = mem[addr + (i * elem_size)]
  }
```

Here is the pseudo-code of the 32-bit unit-stride vector memory store instruction (`vse32.v`):

```
def vse32.v(base_address, vs3):
```

```
2
3       vlen = active_vector_length
4       addr = base_address
5       elem_size = 4
6
7       for (i = 0; i < vlen; i++) {
8         mem[addr + (i * elem_size)] = vs3[i]
9       }
```

Constant-stride assumes a stride that is provided through a scalar register. Here is the pseudo-code of the 32-bit constant-stride vector memory load instruction (`vlse32.v`):

```
1     def vlse32.v(vd, stride_value, base_address):
2
3       vlen = active_vector_length
4       stride = stride_value
5       addr = base_address
6
7       for (i = 0; i < vlen; i++) {
8         vd[i] = mem[addr + (i * stride)]
9       }
```

Here is the pseudo-code of the 32-bit constant-stride vector memory store instruction (`vsse32.v`):

```
1     def vsse32.v(stride_value, base_address, vs3):
2
3       vlen = active_vector_length
4       stride = stride_value
5       addr = base_address
6
7       for (i = 0; i < vlen; i++) {
8         mem[addr + (i * stride)] = vs3[i]
9       }
```

Indexed-stride is the most complex of all modes and uses a vector register to specify the stride for each element. There two types of indexed-stride vector memory instructions: Ordered and unordered. Ordered indexed-stride vector memory instructions guarantee a strict order of execution between the elements within a vector by the hardware, whereas unordered relaxes the

ordering to potentially enable faster execution. Here is the pseudo-code of the 32-bit indexed-stride vector memory load instruction (`vlxei32.v`):

```
def vluxei32.v(vd, base_address, vs2):

  vlen = active_vector_length
  addr = base_address

  for (i = 0; i < vlen; i++) {
    vd[i] = mem[addr + vs2[i]]
  }
```

Here is the pseudo-code of the 32-bit indexed-stride vector memory store instruction (`vsxe32.v`):

```
def vsuxe32.v(base_address, vs2, vs3):

  vlen = active_vector_length
  addr = base_address

  for (i = 0; i < vlen; i++) {
    mem[addr + vs2[i]] = vs3[i]
  }
```

Since the different between ordered and unordered is hardware support, the pseudo-code shown above applies to both ordered (i.e., `vluxei32.v`, `vsuxei32.v`) and unordered (i.e., `vloxei32.v`, `vsoxei32.v`).

3.2.4 Vector Cross-Elements Instructions

Vector cross-element instructions perform an operation that requires cross-element communication. With the exception of `vpopc.m`, the outcome of the operation is stored into a destination vector register. There are three categories of vector cross-elements instruction: reduction, cross-element move, and register gather instructions.

Vector Reduction Instructions – these instructions perform an operation between all elements of a vector register, thus reducing it into a single value. The value is stored into the first element of the destination vector register. The pseudo-code for a vector `op` reduction instruction is as follows:

```
def vredop(vd, vs2):
```

```
2
3       vlen = active_vector_length
4
5       for (i = 0; i < vlen; i++) {
6         vd[0] = op(vd[0], vs2[i])
7       }
```

Vector Cross-Element Move Instructions – these instructions perform a move of elements within a vector register, thus changing the order of these elements. RVV supports slide instructions, which move the elements up/down the vector register. The slide can be performed in either direction with a constant slide amount provided through either a scalar register or as an immediate encoded into the instruction. The pseudo-code for a slide right (`vlsideup.v`) instruction is as follows:

```
1     def vslideup.v(vd, vs2, slide_amount):
2
3       vlen = active_vector_length
4       s_amnt = slide_amount
5
6       for (i = 0; i < vlen; i++) {
7         if (i < s_amnt) {
8           vd[i] = 0
9         } else {
10          vd[i] = vs2[i - s_amnt]
11        }
12      }
```

The pseudo-code for a slide left (`vlsidedown.v`) instruction is as follows:

```
1     def vslidedown.v(vd, vs2, slide_amount):
2
3       vlen = active_vector_length
4       s_amnt = slide_amount
5
6       for (i = 0; i < vlen; i++) {
7         if (i + s_amnt < vlen) {
8           vd[i] = vs2[i + s_amnt]
9         } else {
10          vd[i] = 0
```

```
    }
  }
```

Vector Gather Instructions – these instructions shuffle the elements of a source vector register based on the indices specified by either a second source vector register, a scalar value, or an immediate value. The pseudo-code for the vector-vector gather (`vrgather.vv`) instruction is as follows:

```
def vrgather.vv(vd, vs2, vs1):

  vlen = active_vector_length

  for (i = 0; i < vlen; i++) {
    if (vs1[i] < vlen) {
      vd[i] = vs2[vs1[i]]
    } else {
      vd[i] = 0
    }
  }
```

Whereas the pseudo-code for the vector-scalar (`vrgather.vx`) and vector-immediate (`vrgather.vi`) gather instruction is as follows:

```
def vrgather.vx(vd, vs2, idx):

  vlen = active_vector_length

  for (i = 0; i < vlen; i++) {
    if (idx < vlen) {
      vd[i] = vs2[idx]
    } else {
      vd[i] = 0
    }
  }
```

Since vector-immediate (`vrgather.vi`) gather instruction achieves the same functionality but takes an immediate instead of scalar, the pseudo-code for both vector-immediate (`vrgather.vi`) and vector-immediate (`vrgather.vi`) is the same.

3.2.5 Masking Support

RVV masking allows for various vector instructions to be predicated. By setting a masking bit encoded in an instruction, each element can be predicated conditionally based on a vector mask. If the vector mask is set, then the element is predicated, otherwise the element is not predicated. RVV allows for two possible configurations regarding the behavior for elements that are not predicated: either the element is preserved, or the element is not preserved. RVV provides a single vector mask register, which is the first vector register (i.e., `v0`). Each mask for an element is a single-bit. These bits are stored consecutively in the vector mask register.

3.3 Background: gem5 Simulator

This section provides some background information on the gem5 simulator, which has been chosen due to: its prevalence in computer architecture research, its pre-existing high-fidelity models of cores and caches, its extensibility, and being community-driven and open-source. Section 3.3.1 begins with a quick overview explaining gem5 execution. Section refchap-ngva-sec-gem5-models given a brief description of provided models in gem5 that would be leveraged later by this work.

3.3.1 Overview

The gem5 simulator is an open-source cycle-level event-driven simulator. A simulation in gem5 can be configured dynamically through Python with no need for re-compilation. The configured environment includes multiple `SimObjects`. Each `SimObject` is simulated at different ticks (i.e., specific simulated wall-clock time) through events managed by the `SimObject` itself. These events, when created, specify a function to call in the `SimObject` and a specific tick to trigger the event. The tick is specified with respect to a clock-domain assigned to the created `SimObject`. To compose multiple `SimObject` together, one can instantiate these `SimObject` in the Python-based configuration dynamically with no need to re-compile. The configuration also specifies the clock-domain for each `SimObject`. The different `SimObjects` can communicate together through `ports`–a standardized channels for communication specifying functions to be implemented and

handled by the SimObjects. These communication channels are connected together through the dynamic configuration.

3.3.2 Provided Models

One of the main reasons for choosing the gem5 simulator is the plethora of provided open-source high-fidelity models. Here is a list of models that are provided by gem5 that are used in work:

The O3 Model – a pipelined out-of-order super-scalar core SimObject capable of modeling different ISAs. The core model has 7 main stages: fetch, decode, dispatch, issue, execute, write-back, and commit. Each stage might be further broken into a deeper pipeline. The model is very extensible and allows to be configured with different branch prediction schemes, and different functional units arrangements. The model is also very configurable and allows for the following to be configured dynamically: fetch block size, decode width, dispatch width, issue width, writeback width, commit width, latencies for different functional unit, execution style of the different functional units, and functionality supported by each functional unit. The model is very elaborate and is composed of more than 25,000 lines of code.

The Ruby Cache Model – a high-fidelity model for a sub-system of caches provided with support for more than 12 different cache coherency protocols implemented using the SLICC domain-specific language. Ruby cache model can also be dynamically configured to compose a sub-system of caches with different hierarchies and connections. These cache sub-systems also use the provided Garnet network model to connect the caches with different network-on-chip (NoC) micro-architectures. The transactions and actions defining cache coherency protocol in these caches are specified in SLICC and synthesized to C++ at compile time. For a SimObject to connect to any of the caches specified, a Sequencer is instantiated to interface between the SimObject and the ruby model. These Sequencers help in implementing coalescing support and re-ordering.

```
decode QUADRANT default Unknown::unknown() {
  0x0: decode COPCODE {
    0x0c: decode FUNCT3 {
      format ROp {
        0x0: decode FUNCT7 {
          0x0: add({{
            Rd = Rs1_sd + Rs2_sd;
          }});
          0x1: mul({{
            Rd = Rs1_sd * Rs2_sd;
          }}, IntMultOp);
          0x20: sub({{
            Rd = Rs1_sd - Rs2_sd;
          }});
        }
      }
    }
  }
}
```

Figure 3.1: Snippet of Instructions Defined in The Decoder – This snippet is showing `add`, `mul`, and `sub` instructions. These instructions are assigned to the format `ROp` and pass arguments to functionally define the instruction.

3.4 Modeling Functional Next-Generation Vector Execution

This section details a functional model built in the gem5 simulator to execute RISC-V Vector (RVV) code and verify its correctness. Section 3.4.1 begins with general overview of the model. Section 3.4.2 details the front-end implementation to enable the model

3.4.1 Overview

The gem5 simulator implements a domain-specific parser for defining ISAs. To define a new instruction, the decoder in gem5 is modified with the correct bit fields to correctly insert the new instruction (example implementation is show in Figure 3.1. These instructions are assigned to defined formats. These formats help the parser by defining the general structure of the new instructions as well as defining common code-base. The domain-specific parser in gem5, then, proceeds to generate all required code to be compiled along gem5. To separate core model development from ISA development, gem5 implements a timing/execution separation, thus execution semantics is defined by the implementation of the instruction while timing is defined by implementation of the core. gem5 has an agnostic `StaticInst` class that constitute the parent class for all generated

instructions. This class has a well-defined programming interface that enables ISA developers to convey information to help core model developers in instrumenting the timing of execution correctly. Some examples of these information includes number of source registers, source registers architectural names, required functionality, functions to interact with other units in the core.

3.4.2 Front-End Implementation

To facilitate vector implementation in the gem5 simulator, we introduce several new formats. Figure 3.2 shows the implementation of `VOPIVV`, which is one of the newly added formats and used for instructions such as `vadd.vv`. The format generates four code blocks to define and implement the `StaticInst` class that represents the instruction: `header_output`, `decoder_output`, `decode_block`, and `exec_output`. The `header_output` defines the declaration of the class. The `decoder_output` defines the constructor of the class. The `decode_block` defines the proper instantiation of a static instruction by the decoder. The `exec_output` defines all functions specified in the `header_output` and implements them correctly (Figure 3.3 shows one of the execution templates). To support all of RVV specifications, we added 21 different formats. The implementation of the `vadd.vv` instruction in the decoder is shown in Figurechap-ngva-fig-decoder-vvadd.

3.5 Modeling Decoupled Next-Generation Vector Engine

This section describes the decoupled next-generation vector engine model and implementation.

3.5.1 Decoupled Vector Engine Model

The model of the decoupled vector engine is heavily inspired by Tarantula [EAE+02]. Figure 3.5 shows the details of the decoupled vector engines model and its integration aspect with the core. The model includes multiple stages and units. The main execution datapath includes five stages: decode, issue, execution, writeback, and commit. The model also includes a vector memory unit (VMU) to handle vector memory operations execution. The engine is connected to an out-of-order core (O3) in a decoupled fashion, since the engine is non-speculative. The O3 core is responsible for fetching instructions and resolving control hazards. Once the O3 core encounters a vector instruction, the core forwards the instruction to the vector engine when it reaches the top

```
def format VOPIVV(pred_tcode,
                  pred_fcode={{    }},
                  tail_code ={{    }},
                  pred_if   ={{true}},
                  loop_limit={{VLEN}},
                  rs1_x     ={{ 0  }},
                  rs1_data  ={{ 0  }},
                  rs2_x     ={{ 0  }},
                  rs2_data  ={{ 0  }},
                  rs3_data  ={{ 0  }},
                  rd_x      ={{ 0  }}, funct=0, flags=[]) {{
  flags+=['IsVector']
  regs = ['_destRegIdx[0]','_srcRegIdx[0]', '_srcRegIdx[1]']
  iop = InstObjParams(name, Name, 'VOp',
                        {'pred_tcode':pred_tcode,
                         'pred_fcode':pred_fcode,
                         'tail_code':tail_code,
                         'pred_if':pred_if,
                         'loop_limit':loop_limit,
                         'rs1_x':rs1_x,
                         'rs1_data':rs1_data,
                         'rs2_x':rs2_x,
                         'rs2_data':rs2_data,
                         'rs3_data':rs3_data,
                         'rd_x':rd_x,
                         'funct':funct,
                         'regs':','.join(regs)},
                       flags)
  header_output = VectorDeclare.subst(iop)
  decoder_output = VectorConstructor.subst(iop)
  decode_block = VectorDecode.subst(iop)
  exec_output = (BasicVectorExecute.subst(iop) +
                 VDisassemblyIVV.subst(iop))
}};
```

Figure 3.2: The VOPIVV New Vector Arithmetic Format for RVV – The format generates the following code blocks: header_output, decoder_output, decode_block, and exec_output. These code blocks are produced using common-code templates with placeholders that are defined in the `iop` dictionary.

```
def template BasicVectorExecute {{
  Fault %(class_name)s::execute(ExecContext *xc, Trace::InstRecord *traceData) const
  {
    Fault fault = NoFault;

    %(op_decl)s;
    %(op_rd)s;

    // Get maximum hardware vector length
    auto hardware_vlen = xc->maxVectorLength();

    if (fault == NoFault) {
      for (int i = 0; i < %(loop_limit)s; i++) {
        if (%(pred_if)s) {
          %(pred_tcode)s;
        } else {
          %(pred_fcode)s;
        }
      }
      for (int i = %(loop_limit)s; i < hardware_vlen; i++) {
        %(tail_code)s;
      }
      if (fault == NoFault) {
        %(op_wb)s;
      }
    }

    return fault;
  }
}};
```

Figure 3.3: Common-Code Vector Execution Template – This template `BasicVectorExecute` is used by multiple formats and multiple instructions. To use this template, the placeholders in the template looks as follows: "`%(...)s`". The values to replace these placeholders are defined by the format.

of the re-order buffer (ROB) and about to commit, thus the instruction is no longer speculative. If the vector instruction writes its results to a scalar register, then the core stalls the committing of vector instruction until a reply is received from the vector engine.

In the vector engine, received vector instructions are enqueued to be decoded as a vector execution command. These vector command are sent to the issue stage. The issue stage handles hazards by waiting for all dependencies to be satisfied before the vector command is sent to the execution stage. The execution stage is composed of multiple execution pipes, which include a set of functional units and is allocated their own read and write vector register file ports. Each execution pipe also includes a reservation station that handles generating chimes for each vector command to be

```
decode QUADRANT default Unknown::unknown() {
  0x0: decode COPCODE {
    // +--> VINT
    0x15: decode FUNCT3 {
      // VOPIVV
      0x0: decode FUNCT6 {
        // vadd
        0x0: decode VM {
          // vadd_vv
          0x1: VOPIVV::vadd_vv({{
            Vd_ud[i] = Vs2_ud[i] + Vs1_ud[i];
          }}, funct=0, flags=['IsVectorAluInst', 'SimdAddOp']);

          // vadd_vvm
          0x0: VOPIVV::vadd_vvm({{
            Vd_ud[i] = Vs2_ud[i] + Vs1_ud[i];
          }}, pred_fcode={{
            Vd_ud[i] = Vs3_ud[i];
          }}, pred_if={{Vm_ud[i] & 0x1llu}}, funct=0,
          flags=['IsVectorAluInst', 'IsVectorMaskedInst',
                 'SimdAddOp']);
        }
      }
    }
  }
}
```

Figure 3.4: Implementation of `vadd.vv` in The Decoder – This snippet is showing the implementation of `vadd.vv` instruction. The instruction is assigned to format `VOPIVV`. Different instruction-specific code is passed as arguments to the format.

executed by the functional units. Executed chimed vector commands from the different execution pipes are sent to the writeback stage where the result is written to the vector register file using the each execution pipe's write port. If the chimed vector command is flagged as being the last chime, the vector instruction is sent to the commit stage. Committed instructions always send a response back to the O3 core to confirm execution.

For memory, the model employs memory request-writeback decoupling. Decoded vector memory instructions generate two vector commands: execution vector command (handled as aforementioned above), and a memory vector command. Vector memory commands are sent to the VMU and the VMU handles its execution. The generated execution vector command depends on the type of the vector memory instruction; for unit- and constant-stride vector load instructions, the execution vector command is simply a write from the VMU into the vector register file; for unit- and constant-stride vector store instructions, the execution vector command is a read from the vec-

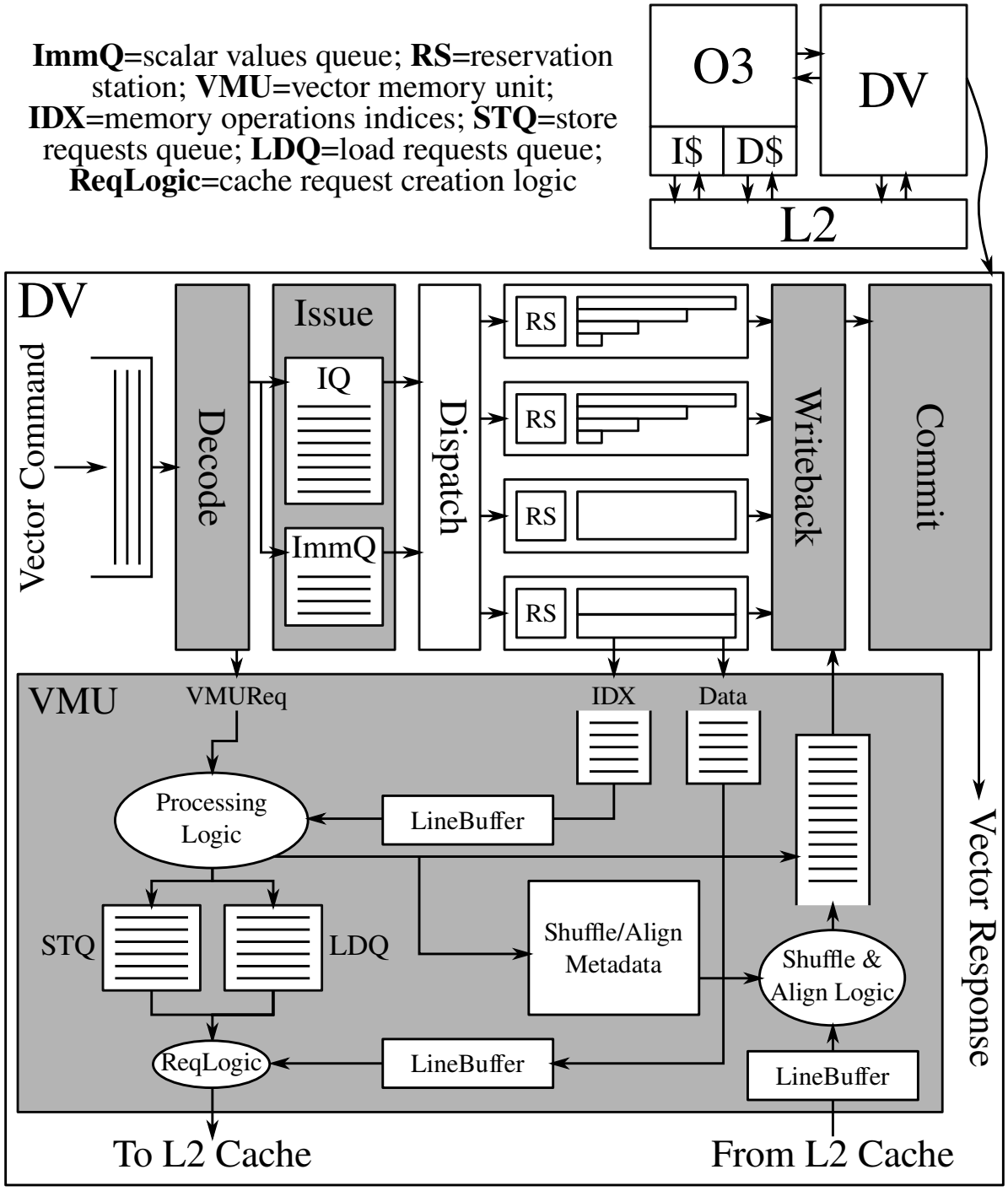


Figure 3.5: Overview of Decoupled Vector Engine (O3+DV).

tor register file into the VMU. As for indexed vector memory instructions, the generated execution vector command will additionally read the indices into the VMU.

The detailed of the vector memory unit (VMU) is shown in Figure 3.5. Incoming memory vector commands into the VMU is queued awaiting processing. The processing logic will handle one memory vector command at a time. While the processing logic is generating cache-line requests, it will not accept any other memory vector commands. The processing logic takes a cycle for each required cache-line; it also includes a lightweight coalescing logic where consecutive elements are considered for coalescing. The coalescing is canceled with the first element that cannot be coalesced. This enables the processing logic to efficiently identify situations where a constant-stride vector memory operation has a constant set to one. The single-cycle also accounts for address translation of the generated cache-line request. If a data is to be expected with a cache-line request, the processing logic reserves a place for the data in the cache-line queue. The processing logic also adds shuffle and alignment information to an internal storage that can be used to ready data before it is sent back to the datapath. If indices are expected for the cache-line requests, the processing logic relies on the indices queue (i.e., `IDX` queue) to provide them. Generated cache-line requests are enqueued into the load queue (`LDQ`) and store queue (`STQ`). While inserting these cache-line requests, the VMU depends on a content-addressable memory (CAM) to identify cache-line-level memory dependencies between older vector stores and younger vector loads. If a dependency is identified, the younger vector load is marked as not ready. Once the older vector store is sent to the cache, the younger dependent vector loads are marked as ready.

To synchronize the execution between the O3 core and the decoupled vector engine, we have introduced multiple vector fences (`vfence`) to specify type of synchronization. Naturally, a `vmv.x.s` can be used to wait for the datapath of the vector engine to finish executing; the `vmv` will stall at the top of the ROB waiting for the vector response, which will be sent in-order. Since no other vector commands can be sent due to the commit stage stalling, a synchronization can be achieved. However, this way of synchronization will not guarantee waiting for all vector memory operations to finish execution. A new `vfence.m` is introduced that form a vector memory fence. For inter-scalar-vector memory dependency, the newly introduced `vfence.m` is needed since the hardware cannot check for memory dependency. This decision has been made to increase the performance of the VMU. Mandating all requests between scalar and vector to be checked for dependency would have introduced hardware bottlenecks for the VMU and severely complicate and restrict imple-

mentations for the hardware. To executed these fences, the datapath waits for the VMU to signal that all in-flight stores have been executed (i.e., a response has been received by the VMU).

3.5.2 Decoupled Vector Engine Implementation

The decoupled vector engine requires three key modifications to the gem5 simulator:

Decoupled Vector Engine SimObject – The decoupled vector engine `SimObject` was implemented in the gem5 simulator following the model shown by Figure 3.5. The `SimObject` has parameters to control through configuration the sizes of queues between the different stages. The `SimObject` is ticked every cycle and subsequently, the `SimObject` will tick all stages start from the commit stage to the decode stage. After all stages are ticked, the VMU is ticked as well.

Rocket Chip Coprocessor (RoCC) Interface – To integrate the decoupled vector engine `SimObject` with the O3 core model, we have implemented the Rocket coprocessor chip (RoCC) interface in the gem5 simulator. Starting from the gem5 O3 core model, we have added a RoCC interface unit. Then, we modified the vector formats introduced in Section 3.4.2 to include functions to communicate correctly with the RoCC interface. After that, the O3 core model is modified to invoke these functions to instrument the execution of a RoCC-capable dynamic instruction.

The changes to the O3 core model are as follows: when a vector instruction is decoded and the O3 core model is connected to a decoupled vector engine `SimObject`, then the dynamic vector instruction is flagged as RoCC-capable. Upon receiving a RoCC-capable instruction, the dispatch ensures the RoCC interface has a space for the incoming request. Then, the function `initiateReq` of the dynamic instruction is called to create and enqueue a RoCC request in the interface unit. Then, the dynamic instruction waits for all dependencies to be ready before it can be issued (e.g., dependency on scalar registers). When all dependency are ready, the dynamic instruction can issue and is sent to commit stage awaiting completion. When the dynamic instruction is ready to commit, the member function `executeReq` of the dynamic instruction is called to update the RoCC request in the interface with renamed registers identifiers and the values needed from scalar registers. The function `executeReq` will also flag the RoCC request as ready to be sent to the vector engine. Finally, if the dynamic instruction does not write values back to the scalar registers, then it is committed and removed from the ROB. However, if the dynamic instruction writes values, then the commit stage stalls waiting for a response from the vector engine. When the response is received, the RoCC interface invokes the member function `completeReq` with the response to be

```
def format VOPIVV(pred_tcode,
                  pred_fcode={{    }},
                  tail_code ={{    }},
                  pred_if   ={{true}},
                  loop_limit={{VLEN}},
                  rs1_x     ={{ 0  }},
                  rs1_data  ={{ 0  }},
                  rs2_x     ={{ 0  }},
                  rs2_data  ={{ 0  }},
                  rs3_data  ={{ 0  }},
                  rd_x      ={{ 0  }}, funct=0, flags=[]) {{
  flags+=['IsVector']
  regs = ['_destRegIdx[0]','_srcRegIdx[0]', '_srcRegIdx[1]']
  iop = InstObjParams(name, Name, 'VOp',
                        {'pred_tcode':pred_tcode,
                         'pred_fcode':pred_fcode,
                         'tail_code':tail_code,
                         'pred_if':pred_if,
                         'loop_limit':loop_limit,
                         'rs1_x':rs1_x,
                         'rs1_data':rs1_data,
                         'rs2_x':rs2_x,
                         'rs2_data':rs2_data,
                         'rs3_data':rs3_data,
                         'rd_x':rd_x,
                         'funct':funct,
                         'regs':','.join(regs)},
                      flags)
  header_output = VectorDeclare.subst(iop)
  decoder_output = VectorConstructor.subst(iop)
  decode_block = VectorDecode.subst(iop)
  exec_output = (BasicVectorExecute.subst(iop) +
                 VectorInitiateReq.subst(iop) +
                 VectorExecuteReq.subst(iop) +
                 VectorCompleteReq.subst(iop) +
                 VDisassemblyIVV.subst(iop))
}};
```

Figure 3.6: The VOPIVV Format with RoCC Support – The format now includes three new templates in the `exec_output`: `VectorInitiateReq`, `VectorExecuteReq`, and `VectorCompleteReq`.

```
def template VectorInitiateReq {{
  Fault
  %(class_name)s::initiateReq(ExecContext *xc,
    Trace::InstRecord *traceData) const
  {
    Fault fault = NoFault;

    %(op_src_decl)s;
    %(op_rd)s;

    if (fault == NoFault) {
      fault = initiateRoccReq(xc, traceData);
    }

    return fault;
  }
}};
```

Figure 3.7: `VectorInitiateReq` **Template –** This template implements the member function `initiateReq` of the vector instruction class.

parsed by the dynamic instruction. Then, the commit stage can finally complete the instruction and remove it from the ROB.

If this dynamic instruction is squashed, then the interface unit will handle the squashing correctly. There are three possible places where squashing is possible: before the dynamic instruction creates a RoCC request in the interface, after the dynamic instruction creates a RoCC request but it is not sent to the vector engine, and after the dynamic instruction creates a RoCC request and it is already sent to the vector engine. For the first scenario, the dynamic instruction is squashed and turned into a `NOP` and thus it would not create a request and become a bubble in the pipeline. The second scenario, the RoCC interface unit is notified with the squash signal and the corresponding RoCC request will be removed without any side-effects. As for the third scenario, the RoCC interface unit is not designed to handle speculative RoCC request and, thus, this scenario should not occur. The implementation includes checks and will throw and exception in case of an invalid scenario.

Cache Sub-System Model – A new protocol has been introduced to the Ruby cache model to handle an extra port in the level-2 cache to connect the vector engine, as shown in Figure 3.5. The protocol is configured to give priority to connection between level-1 and level-2 caches, while the connection with the vector engine is given a second priority. We also modified the Arm CHI proto-

```
// RoCC execute request template
def template VectorExecuteReq {{
  Fault
  %(class_name)s::executeReq(bool _xd,  uint8_t _rd,
    bool _xs1, uint8_t _rs1,
    bool _xs2, uint8_t _rs2,
    bool _xm,  uint8_t _rm,
    ExecContext *xc,
    Trace::InstRecord *traceData) const
  {
    Fault fault = NoFault;

    %(op_src_decl)s;
    %(op_rd)s;

    .
    .
    .
    // Determine all required values:
    // opcode
    // n_funct
    // rd_x, rd_id, rd_data,
    // rs1_x, rs1_id, rs1_data
    // rs2_x, rs2_id, rs2_data
    .
    .
    .

    if (fault == NoFault) {
      fault = executeRoccReq(xc, traceData,
                             opcode, n_funct,
                             rd_x,  rd_id,  rd_data,
                             rs1_x, rs1_id, rs1_data,
                             rs2_x, rs2_id, rs2_data);
    }

    return fault;
  }
}};
```

Figure 3.8: `VectorExecuteReq` **Template –** This template implements the member function `executeReq` of the vector instruction class.

col to support a second connection to the vector engine in all of its levels. These implementations has been tested using Ruby's pre-existing testing infrastructure.

```
def template VectorCompleteReq {{
  Fault
  %(class_name)s::completeReq(RoccPacketPtr pkt, ExecContext *xc,
    Trace::InstRecord *traceData) const
  {
    Fault fault = NoFault;

    %(op_decl)s;
    %(op_rd)s;

    // Output
    bool     rd_x;
    uint64_t rd_id;
    uint64_t rd_data;

    // Parse incoming data
    parseRoccResponse(xc, traceData, pkt,
                      rd_x, rd_id, rd_data);

    if (fault == NoFault) {
      %(op_wb)s;
    }

    return fault;
  }
}};
```

Figure 3.9: `VectorCompleteReq` **Template –** This template implements the member function `completeReq` of the vector instruction class.

3.6 Modeling Integrated Next-Generation Vector Unit

This section describes the integrated next-generation vector unit model and implementation.

3.6.1 Integrated Vector Unit Model

The integrated vector unit model is loosely based on recent work and publications on the Arm Scalable Vector Extension (SVE) [SBB+17a, RBGZ19]. To reduce the area overhead of the unit, most of the components needed for execution is re-purposed from pre-existing components in the O3 datapath. The vector registers share the register file with the floating point registers. The renaming stage is modified to support renaming both vector registers and floating-point registers to map to the same physical registers space. The floating-point ALUs are modified and enhanced to support vector operations along side the scalar floating-point operations. For vector memory operations, the scalar load-store queue (LSQ) is modified to support unit-stride vector memory

operations. For constant-stride and indexed vector memory operations, the decoder in the O3 core will breakdown vector memory operations into several micro-operations; each micro-operation will handle one element of the vector. As a result, these vector memory operations do not need any special support from the scalar LSQ, as they are treated the same as scalar memory operations.

The integrated vector unit is tightly integrated into the O3 core datapath. As a result, the execution of the integrated vector unit follows the O3 core execution; thus, the integrated vector unit execution is speculative. Vector registers are renamed to eliminate false hazards. Vector instructions are issued out-of-order to the vector ALUs whenever they are ready. Moreover, vector instructions can be squashed due to control hazard or an earlier exception. Vector memory operations need to perform memory address disambiguation. In case the scalar LSQ perform speculative memory address dependency, the vector memory instructions need to support the same. Finally, vector store instructions are executed out-of-order, but they are placed in a queue along with scalar store instructions waiting for their corresponding instructions to commit before they are sent to the cache.

3.6.2 Integrated Vector Unit Implementation

To implement the integrated vector unit model, there are four required changes and contributions to vanilla gem5: front-end modifications, macro-/micro-operation decomposition and support, and LSQ changes to support vector memory operation. Recently, Arm has mainstreamed multiple changes to enable support for Arm SVE in gem5. We leveraged some of these changes in our implementation of an integrated vector unit. The LSQ changes to support vector memory operation has already been contributed by Arm and only needed to be adapted to RVV.

The front-end modifications were detailed in Section 3.4. Since there is a separation between execution and timing in gem5, the execution specified by these front-end implementation applies to the integrated vector unit. As for macro-/micro-operation decomposition and support, this required some changes to renaming logic in the O3 core model. In situations where some elements are masked-off, the masked-off elements are expected to be unchanged. As a result, vector instructions read the previously mapped destination register in masked instructions to copy masked-off elements to the newly allocated vector physical register. As a result, when renaming micro-operations that belong to the same macro-operation, the renaming for the old physical register assigned to the destination is preserved and unchanged until the last micro-operation is renamed.

Name	Suite	Input	Type	Scalar	
				DIns	IOc
vvadd	k	8.388M	int	75.5M	205M
mmult	k	1024	int	8.60B	48.5B
saxpy	k	512×512	fp	1.08B	3.18B
k-means	ro	10Kx34	fp	4.65B	13.6B
jacobi-2d	rv	2Kx10	fp	1.59B	9.64B
lavamd	rv	4x4x4	fp	1.05B	3.58B
backprop	ro	524K	fp	1.17B	16.4B
particlefilter	rv	6K	fp	1.24B	3.52B
blackscholes	rv	2.5M	fp	726M	2.75B
k-means-int	ro	10Kx34	int	2.11B	3.59B
jacobi-2d-int	rv	2Kx10	int	1.59B	8.50B
lavamd-int	rv	4x4x4	int	617M	1.04B
backprop-int	ro	524K	int	1.17B	15.7B
pathfinder	ro	5Mx10	int	1.08B	2.43B
sw	g	2048	int	1.38B	1.69B

Table 3.1: Benchmark Applications List – k = kernel, ro = rodinia, rv = RiVEC, g = genomics, int = integer type benchmark, fp = floating-point type benchmark, DIns = number of dynamic instructions in ROI, IOc = number of cycles to run on an in-order core.

A micro-operation asking for the mapping of `vd` as a destination will be given the new physical mapping; however, the micro-operation asking for the mapping of `vd` as a source will be given the old physical mapping.

3.7 Evaluation

This section details the evaluation methodology (Section 3.7.1) and results (Section 3.7.2).

3.7.1 Methodology

To quantify our models and their performance, we leveraged the gem5 [BBB+11, LPAA+20, TCB18a] simulator detailed in Section 3.3. For the baselines, we built BRGIO, which is an in-house developed-from-scratch in-order core model. The other baseline is an altered version of the O3 core model (BRGO3), which is an out-of-order core model. The differences between gem5's O3 core model and BRGO3 includes the implementation of the RoCC interface (detailed in Sec-

	Vector (VL=64)															
Name	**DIns**	**VI%**	**ctrl**	**ialu**	**imul**	**fpu**	**xe**	**us**	**st**	**idx**	**prd**	**DOp**	**VO%**	**VPar**	**WInf**	**ArInt**
vvadd	1.6M	42%	20	20	0	0	0	7	0	0	0	35.5M	96%	22.6	0.47	0.33
mmult	151M	44%	25	25	25	0	0	25	0	0	0	3.35B	97%	22.2	0.39	2.00
saxpy	18.9M	44%	25	25	25	0	0	25	0	0	0	416M	97%	22.0	0.39	2.00
k-means	67.9M	46%	1	13	∼0	57	∼0	∼0	21	7	1	2.01B	98%	29.6	0.43	2.41
jacobi-2d	35.4M	44%	8	17	0	42	17	17	∼0	0	0	942M	98%	26.6	0.59	4.50
lavamd	25.0M	79%	7	11	0	57	0	5	20	0	5	966M	99%	38.7	0.92	2.72
backprop	21.5M	39%	13	∼0	0	44	0	12	31	0	0	484M	97%	22.5	0.41	1.00
particlefilter	43.1M	51%	1	72	1	26	0	∼0	∼0	∼0	∼0	1.41B	99%	32.8	1.14	200.9
blackscholes	10.6M	90%	4	13	0	72	0	3	8	0	5	567M	100%	53.5	0.78	7.48
k-means-int	51.5M	46%	1	52	18	0	∼0	∼0	10	7	1	1.53B	98%	29.8	0.72	2.44
jacobi-2d-int	35.4M	44%	8	50	8	0	17	7	0	0	0	940M	98%	26.6	0.59	4.50
lavamd-int	26.6M	61%	23	24	14	0	20	2	9	1	7	726M	97%	27.3	1.18	2.98
backprop-int	21.5M	39%	13	19	25	0	∼0	5	12	0	0	488M	96%	22.7	0.42	1.00
pathfinder	22.5M	50%	31	37	0	0	0	16	0	0	25	513M	97%	22.8	0.48	1.20
sw	20.4M	39%	10	55	0	0	11	10	14	0	10	433M	97%	21.2	0.31	2.75

Table 3.2: Benchmark Applications Characterization – DIns = number of dynamic instructions in ROI, IOc = number of cycles to run on an in-order core, VI% = 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 = vector unit stride memory instructions, st = vector constant stride memory instructions, idx = vector indexed memory instructions, prd = predicated vector instructions, DOps = total number of operations (scalar instructions + vector instructions $\times$ active vector length), VO% = percent of operations performed by vector unit, VPar = logical parallelism (total ops / dynamic instructions in vectorized program), WInf = work inflation (total ops in vectorized program / dynamic instructions in scalar program), ArInt = arithmetic intensity (mathematical operations / memory operations) for vector unit.

tion 3.5.2), and the modification to the renaming logic accommodating RVV special needs (detailed in Section 3.6.2). As for the memory model, a new protocol was modified to support an extra connection to the level-2 cache specifically for the decoupled vector engine. Arm CHI [gem21] Ruby cache model was modified to support an extra port in the level-2 cache specifically for the decoupled vector engine. These changes has been tested and verified for functional and timing correctness.

We have built two new next-generation vector models: the integrated vector unit (detailed in Section 3.6) and the decoupled vector engine (detailed in Section 3.5) in the gem5 simulator. We then verified the correctness of these models using an infrastructure of tests and micro-benchmarks.

IO	Single-issue in-order RV64GC core: • L1I: 1-cycle-hit 4-way 32KB • L1D: 2-cycle-hit 4-way 32KB • L2: 8-way 8-bank 8-cycle-hit 512KB
O3	Out-of-order 8-way RV64GC core: • Same L1I, L1D, and L2 as *IO*
O3+IV	Small vector unit integrated into O3: • Same L1I, L1D, and L2 as *O3* • IV: 4-element VL, out-of-order issue, 3 exec pipes
O3+DV	Decoupled vector engine connected to O3: • Same L1I, L1D, and L2 as *O3* • DV: 64-element VL (chime of 4, element group of 16), in-order issue, 4 exec pipes
LLC	Same for all systems: 16-way, 12-cycle-hit & 2MB
Memory	Same for all systems: single-channel DDR4-2400
Compiler	LLVM-13 • *Flags*: `-O3 -menable-experimental-extensions --target=riscv64-unknown-linux -march=rv64gcv0p10`

Table 3.3: Simulated Systems.

The tests were developed to verify the functional correctness; while, the micro-benchmarks verify the performance and timing correctness. The configuration used for evaluation is detailed in Table 3.3. The sizes of the queues used and different model parameters (e.g., chime size, and number of chimes) were motivated either form sweep experiments or from the models on which they are based.

To understand the performance of the models, we leveraged multiple applications with varying properties and computational patterns. We have chosen applications from the Rodinia [CBM+09] and RiVEC [RHP+20] benchmark suites. In addition, we have developed our own implementation of the Smith-Waterman (sw) benchmark from computational biology field. On top of these applications, we have added multiple kernels as micro-benchmarks to emphasize different aspects of the vector models (e.g., bandwidth, and latency of memory and computation). The list of benchmarks and micro-benchmarks are detailed in Table 3.1. We also developed a functional model in the gem5 simulator to characterize the scalar implementation (data shown in Table 3.1) and the vectorized implementation (data shown in Table 3.2)

Name	Type	Speedup vs. IO			Speedup vs. O3	
		O3	O3+IV	O3+DV	O3+IV	O3+DV
vvadd	int	2.21	4.62	16.82	2.09	7.63
mmult	int	6.89	20.01	88.50	2.91	12.85
saxpy	fp	2.47	6.51	15.06	2.64	6.10
k-means	fp	3.32	8.02	17.69	2.42	5.33
jacobi-2d	fp	5.23	6.24	36.50	1.19	6.99
lavamd	fp	2.25	4.74	12.38	2.10	5.49
backprop	fp	4.03	7.78	16.51	1.93	4.10
particlefilter	fp	5.53	2.97	25.30	0.54	4.58
blackscholes	fp	2.49	5.12	31.76	2.06	12.77
k-means-int	int	3.55	5.19	11.81	1.46	3.33
jacobi-2d-int	int	6.19	5.17	32.92	0.84	5.32
lavamd-int	int	2.51	2.33	4.37	0.93	1.74
backprop-int	int	4.04	7.65	16.36	1.89	4.05
pathfinder	int	2.05	4.99	40.45	2.43	19.73
sw	int	2.06	8.98	17.22	4.35	8.35
geomean		3.35	5.39	18.88	1.61	5.63

Table 3.4: Benchmark Applications Results – geomean = calculated for: {pathfinder, sw, k-means, jacobi-2d, lavamd, backprop, k-means-int, jacobi-2d-int, lavamd-int, backprop-int}.

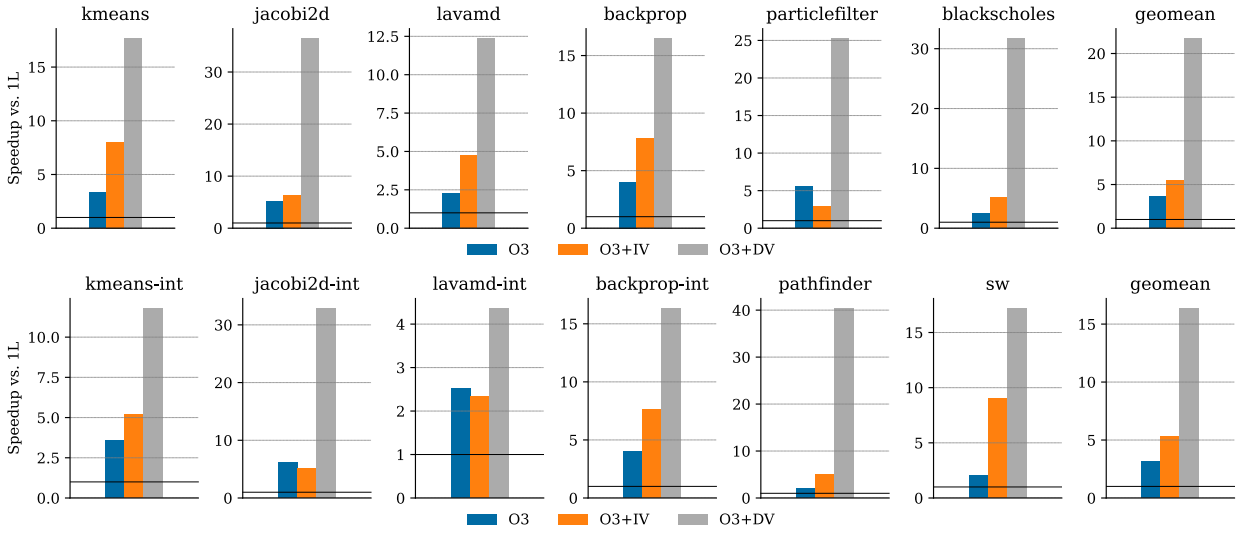


Figure 3.10: Performance Evaluation – performance of the out-of-order model and the two vector models: integrated vector unit (O3+IV) and decoupled vector engine (O3+DV). The performance results are normalized to the performance of an in-order core model executing the scalar version of the application.

3.7.2 Results

The characterization of every application is shown in Table 3.2. The results of the vector models evaluation is shown in Figure 3.3 and detailed in Table 3.4. As the results show, all applications are properly vectorized with the percentage of vector operations are 97 % and above. The integrated vector unit is able to increase performance when compared to an O3 core by a 1.61 $\times$. When compared to IO core, the integrated vector unit improve performance by 5.39 $\times$; this figures aligns with expected value of 4.0 $\times$ since the integrated vector unit has 4.0 $\times$ advantage in-terms of functional units. The integrated vector unit is able to achieve performance higher than expected due to efficiency in executing vector memory operations. The decoupled vector engine is able to increase performance by up to 5.63 $\times$ compared to the O3 core executing scalar version of the benchmarks. When compared to IO scalar core, the decoupled vector engine is able to increase performance by up to 18.88 $\times$. The decoupled vector engine has 16 $\times$ more functional units than the IO core. Much like the integrated vector unit, the decouple vector engine is able to achieve higher performance due to its efficiency in handling vector memory instructions and operations.

3.8 Conclusion and Discussion

A fundamental aspect of every architectural research is baseline modeling. This chapter has introduced the functional vector model, the integrated vector unit model, and the decoupled vector engine model for the RISC-V Vector extension (RVV). These models have contributed to evaluating and quantifying the performance of EVE. While these models have the sufficient fidelity required, further future work on these models can explore modifying the RoCC interface implementation to support out-of-order issue and execution. Such future work would enable a line of research into out-of-order decoupled vector engines. To enable out-of-order issue and execution in RoCC interface requires conveying squashing and committing through the interface to any connected model.

CHAPTER 4
EVEV1: TOWARDS BIT-SERIAL/BIT-PARALLEL SRAM-BASED COMPUTE-IN-MEMORY

Recent work has demonstrated how SRAM-based compute-in-memory (S-CIM) can reduce the area overhead of vector arithmetic-logical units (ALUs). Most work has predominantly explored utilizing bit-serial execution paradigm. In this chapter, I present EVEv1–my early work to explore bit-serial (BS-EVEv1) and bit-parallel (BP-EVEv1) ephemeral vector ALUs by leveraging SRAM-based compute-in-memory (S-CIM). To the best of my knowledge, BP-EVEv1 is the first to propose bit-parallel execution paradigm for S-CIM vector ALUs. I also present a rigorous evaluation of bit-serial vs. bit-parallel approaches. The results show that both approaches have comparable area overhead. The two approaches have very different latency vs. throughput tradeoffs. BS-EVEv1 requires more cycles per operation, but is able to execute thousands of operations in parallel, while BP-EVEv1 requires fewer cycles per operation, but can only execute hundreds of operations in parallel. For 32-bit arithmetic operations, BS-EVEv1 improves throughput by 1.3–5.0 $\times$ compared to BP-EVEv1, while BP-EVEv1 improves latency by 3.0–23.0 $\times$ compared to BS-EVEv1. The findings of EVEv1 further motivates my subsequent work on bit-hybrid approach rather than bit-serial or bit-parallel. As bit-serial approach favors throughput and bit-parallel approach favors latency, the goal of bit-hybrid approach is strike a balance between throughput and latency.

4.1 Introduction

Vector accelerators are seeing a resurgence in both general purpose and domain-specific processing [Ste16, SBB+17b, RISC-V Foundation19]. These accelerators can achieve high performance on well-structured workloads by using a complex vector ALU and register file. To keep the vector ALU busy, vector register files are usually highly multi-ported which incurs significant area and energy overheads. Recent work on in-situ processing-in-SRAM attempts to reduce these overheads by fusing the vector ALU and register file. In-situ processing-in-SRAM uses bit-line computation to perform basic bit-wise logical operations in a single read of a traditional SRAM [JASB16, JASB15]. Each SRAM column can be transformed into a bit-serial ALU by adding extra logic, multiplexing, and state elements in the peripheral circuitry. Alternatively, a

set of SRAM columns can be grouped into a bit-parallel ALU by adding bit-parallel logic in the peripheral circuitry instead.

In this chapter, we provide a detailed implementation for bit-serial EVEv1 (BS-EVEv1) and bit-parallel EVEv1 (BP-EVEv1) as two representative design points for processing-in-SRAM. These two flavors of EVEv1 support a variety of micro-operations for implementing macro-operations. Starting with an implementation of a traditional 28 nm 6T-SRAM in OpenRAM [GSA+16], we designed and laid out the additional peripheral circuitry required to implement BS-EVEv1 and BP-EVEv1. Although surprisingly both designs have comparable area overhead, they have very different performance characteristics. For a 32-bit MAC operation, BS-EVEv1 has 5× higher throughput (6.4 GOPS) than BP-EVEv1 (1.3 GOPS), while BP-EVEv1 can achieve 6.5× lower latency (197.9 ns) when compared to BS-EVEv1 (1281 ns). BS-EVEv1 consumes lower energy per operation, but we discuss possible techniques to help close this gap. As the two designs require similar peripheral circuitry, this work can be seen as a step towards building a reconfigurable bit-serial/bit-parallel vector accelerator that is able to achieve either high-throughput or low-latency depending on the application requirements.

The main contributions of this chapter are: (1) a detailed circuit-level design of BS-/BP-EVEv1 in 28nm technology; (2) implementation of 17 macro-operations in BS-/BP-EVEv1 using micro-operations; (3) a detailed study of the trade-offs in area, cycle time, latency, throughput, and energy for BS-EVEv1 vs. BP-EVEv1. To our knowledge, this is the first work to rigorously explore the trade-offs between a bit-serial vs. bit-parallel approach to in-situ processing-in-SRAM.

4.2 EVEv1 Circuits

BS-EVEv1 and BP-EVEv1 start with a basic 6T SRAM with support for bit-line computation (i.e., extra decoder and reconfigurable single-ended/differential sense amplifiers as in [JASB16, JASB15]). Bit-line computation simplifies the required peripheral logic to implement BS-EVEv1 and BP-EVEv1. Figure 4.1 shows the additional peripheral circuitry required to enable BS-EVEv1 and BP-EVEv1 beyond what is necessary for bit-line computation.

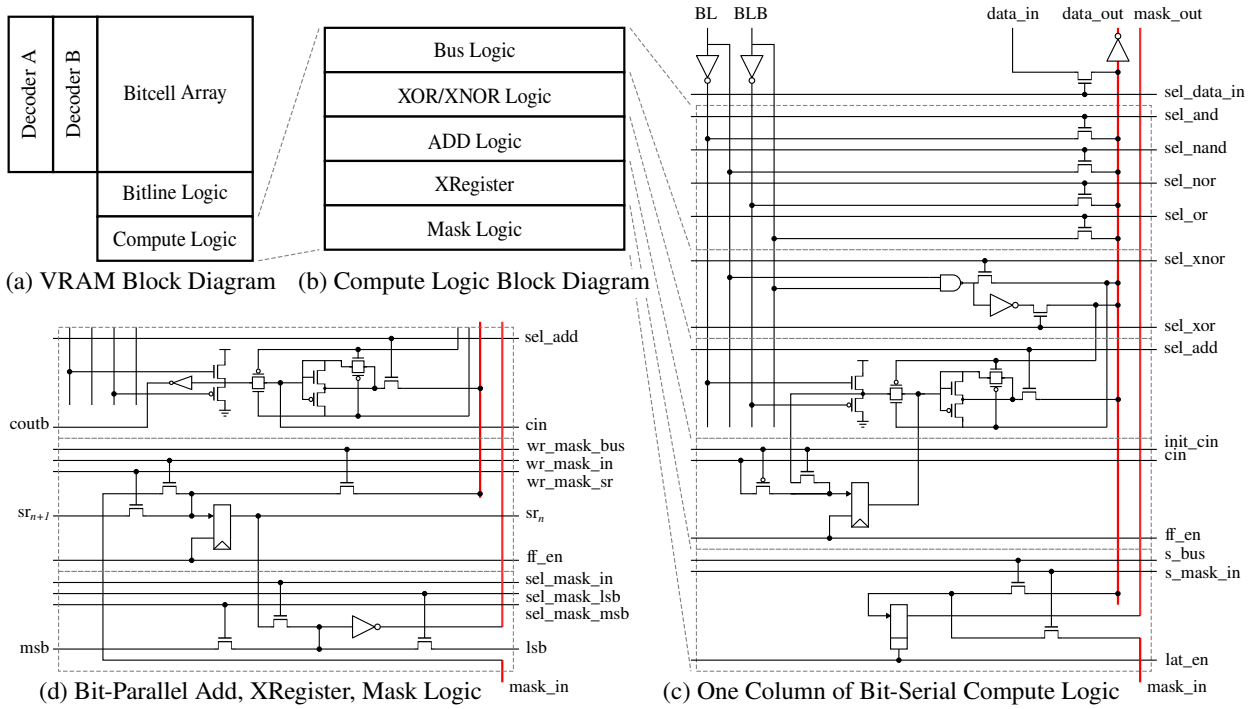


Figure 4.1: VRAM Circuits – High-level block diagram of VRAM microarchitecture showing a column slice of circuit-level details of all the different logic blocks of both VRAM flavors. BP-VRAM shares many logic blocks implementation with BS-VRAM, shown in (c), and only differ in three blocks that are shown in (d). The boxed logic blocks are explained in section 4.2 from top to bottom. The blocks are: bus logic, XOR/XNOR logic, ADD logic, XRegister, and mask logic.

4.2.1 Bit-Serial Compute Logic (BSCL)

The inputs to the BSCL are the output of each sense-amplifier and its complement. Bit-line computation provides bit-wise logical AND, NAND, OR, and NOR on these inputs.

The BSCL is composed of the following blocks. *Bus Logic:* BSCL uses a distributed bus with NMOS pass-transistors to choose between basic bit-wise logical operations (i.e., AND, NAND, OR, NOR). *XOR/XNOR Logic:* Computing XOR and NOR in BSCL requires an additional NAND gate and inverter. *ADD Logic:* BSCL uses a modified serial Manchester carry chain (MCC). As addition is computed bit serially, the carry out is stored for the subsequent cycle while previous carry is used as carry in. The carry in is XOR'ed with the bitwise logical XOR to compute the sum. *XRegister:* XRegister's input is multiplexed to choose either the carry out from the ADD logic or an input carry, which enables initializing the carry to zero for an addition or one for a subtraction. The output of the XRegister is the input to the ADD logic. *Mask Logic:* The mask logic in BSCL

Energy (pJ)	rd	wr	blc	*cond*.wb.*src*	wr_mask.*src*	srl
BS-VRAM	34.7	11.7	33.4	17.3	13.6	-
BP-VRAM	45.3	13.6	43.0	16.8	16.5	6.4

Table 4.1: Micro-operation Energy Characterization – Energy for the supported micro-operations characterized by simulating the extracted layout of a complete sub-array for both BS-VRAM and BP-VRAM; *cond* = {msb, lsb, always}. *src* = {and, nand, nor, or, xor, xnor, add, data_in}.

is a latch with a multiplexer to choose either the unbuffered bus or an input mask. The output of the latch is the mask used by the SRAM for writes. In a conventional write, the input mask will be chosen to be stored in the latch.

4.2.2 Bit-Parallel Compute Logic (BPCL)

Bit-parallel compute logic reuses all but the last three blocks from BSCL. *ADD Logic:* Bit-parallel addition requires a carry propagation for which BPCL uses an MCC. The addition result is computed by XOR'ing the carry-in of every column with the bitwise logical XOR value. For better performance, lightweight buffering along the carry chain is inserted by using the inversion property of the adder. *XRegister:* The XRegister can act as shift register for the multiplier, controlling whether a µop is conditionally executed. The input of every flip-flop is multiplexed choosing either the input mask, the output of the compute logic on the bus, or the output of the XRegister of the column to the left (thus shifting right). *Mask Logic:* To generate the appropriate mask of every column, a multiplexer is added to select between the XRegister output of the current column, the XRegister output of the first column of the element (i.e., least-significant bit "LSB"), or the XRegister output of the last column of the element (i.e., most-significant bit "MSB"). For masking, multiplication uses the LSB while comparators use the MSB.

4.3 EVEv1 Micro-Programming

BS-EVEv1 and BP-EVEv1 implement single-cycle primitives µoperations (µops). This section describes the supported µops and illustrates how a sequence of µops can be used to implement a complex macro-operation.

4.3.1 Micro-Operations

Normal SRAM read and write in a EVEv1 use the `rd` and `wr` μops. The remaining μops are as follows:

Bit-line Compute (`blc`): This μop performs a read and uses the compute logic to generate AND, NAND, OR, NOR, and ADD. The sense-amps latch their outputs, meaning these logic value can be reused until the next `blc` or `rd`.

Writeback (`cond.wb.src`): After executing a `blc`, a writeback μop writes the selected source back to the SRAM. The writeback μop includes two parameters: a condition and a source selection. The condition (supported only by BP-EVEv1) specifies whether the mask bit of a column would be set to: the corresponding `mask_in` bit, the element's LSB, or its MSB. BS-EVEv1 and BP-EVEv1 utilize the already existing write mask (native to SRAMs) to conditionally writeback.

Write to Mask (`wr_mask.src`): Instead of writing to the SRAM, BS-EVEv1 and BP-EVEv1 allow writing to the mask state element (which is the latch in BS-EVEv1 and the XRegister for BP-EVEv1). An algorithm can generate the mask dynamically for subsequent conditional writes.

Shift Right Logical (`srl`): This μop (supported only by BP-EVEv1) shifts the content of the XRegister to the right by one bit. Multiplication uses the LSB of the XRegister as a mask for conditional addition. By shifting, the algorithm can step through the bits of the multiplier from the LSB to the MSB.

Jump if not done (`j_n_done_{0,1}`): There are two counters for control flow, and each counter can be initialized to the desired bitwidth (e.g., 8, 32). This control μop decrements one of these counters (indicated with the suffix). If the counter is zero, the counter is reset and execution falls through to the next μop. If the counter is not zero, execution jumps to the label.

4.3.2 Macro-Operations

Both flavors of EVEv1 utilize arithmetic and control μops to implement multi-cycle macro-operations (see Table 4.2). In BS-EVEv1, single-loop control is used to implement simple macro-operations (e.g., bit-wise logic and addition) while nested-loop control is required for complex macro-operations (e.g., multiplication and division). In BP-EVEv1, bit-parallel hardware is used for simple macro-operations while single-loop control is only required to implement complex macro-operations using a mixed bit-serial/bit-parallel approach. Figure 4.2 shows the implementa-

Macro-Operation	Cycle Count		# Temporary Rows	
	BS-VRAM	BP-VRAM	BS-VRAM	BP-VRAM
`add`	64	2	0	0
`sub`	128	4	0	0
`and,nand,or nor,xor,xnor`	64	2	0	0
`mul`	1185	133	0	1
`mac`	1153	132	0	1
`udiv`	1712	519	5	1
`rem`	1680	390	4	2
`slt,sle,sgt,sge`	162	6	1	0
`seq`	96	11	1	1

Table 4.2: Supported Macro-operations.

tion of addition and multiplication in both BS-/BP-EVEv1. Addition in BS-EVEv1 is a loop where each iteration uses a bit-line compute µop (`blc`) followed by writing back the result of the ADD logic (`wb.add`). Whereas, addition in BP-EVEv1 only requires one bit-line compute µop (`blc`) and one write back (`wb.add`).

4.4 Evaluation

This section discusses the evaluation methodology as well as the area, performance, and energy results of BS-/BP-EVEv1.

4.4.1 Methodology

OpenRAM [GSA+16] was adapted to produce a layout for a traditional 6T-SRAM targeting a 28nm technology node. OpenRAM was also extended to produce a layout for bit-line compute-capable SRAM (BC-SRAM) by adding an extra decoder and reconfigurable single-ended/differential sense-amplifiers. Finally, OpenRAM was used to produce the layout for BS-EVEv1 and BP-EVEv1 based on BC-SRAM by adding the layout for the compute logic.

Figure 4.3 shows the layout of BS-EVEv1. This layout is used to create a detailed extracted netlist for cycle time and energy analysis. The energy of each µop is estimated by averaging the energy consumed for 10 random inputs. Macro-operation energies are estimated by accumulating

Design	Area (μm^2)
SRAM	22,868 (1.00×)
BL-SRAM	23,722 (1.04×)
BS-VRAM	24,940 (1.09×)
BP-VRAM	24,786 (1.08×)

Table 4.3: Sub-Array Area Comparsion – SRAM = Traditional 6T SRAM, BL-SRAM = bit-line compute capable SRAM, BS-VRAM = bit-serial vector RAM, BP-VRAM = bit-parallel vector RAM.

the energy of all executed μops. The cycle time is estimated by simulating the worst-case inputs on the extracted netlist.

4.4.2 Results

Since OpenRAM's SRAM implementation does not use pushed design rules [IKT+98,JSPR08], bitcells are roughly 80% larger compared to an equivalent SRAM generated by a commercial memory compiler. Due to larger bitcells, OpenRAM's SRAM consumes around 1.5× write energy and 3× read energy. The read energy difference is attributed to less optimized sense-amplifiers that require a larger voltage drop on the bitlines. OpenRAM's SRAM achieves an operating frequency of 1.1 GHz compared to commercial SRAM compiler, which can achieve up to 2 GHz. While using OpenRAM without pushed design rules obviously incurs significant overhead compared to a commercial memory compiler, OpenRAM also enables detailed layout design of BS-/BP-EVEv1 and rigorous comparative analysis. Our techniques will also apply to pushed design rule SRAMs.

Table 4.3 shows that BS-/BP-EVEv1 incur marginal area overhead compared to a traditional 6T-SRAM ($\leq$10%). BS-EVEv1 operates at slightly lower clock frequency (900 MHz) compared to the SRAM (1.1 GHz) mainly due to the switch from a differential sense-amp to a reconfigurable sense-amp. The MCC is on the critical path in BP-EVEv1 resulting in an operating frequency of 645 MHz. Table 4.4 shows BP-EVEv1, despite its lower frequency, can achieve 23× lower latency compared to BS-EVEv1. However, BS-EVEv1 is able to achieve 5× higher throughput. Even with lower latency, BP-EVEv1 struggles to compensate for fewer ALUs compared to BS-EVEv1. Per sub-array, BS-EVEv1 has 32× more ALUs, while BP-EVEv1 reduces latency by only 23×.

Table 4.4 shows the energy comparison between BS-EVEv1 and BP-EVEv1 for `add` and `mul` operating on 8-bit and 32-bit data. BS-EVEv1 can reduce the number of executed μops for 8-bit

		32-bit		8-bit	
		BS	BP	BS	BP
Latency (ns)	`add`	71.1	**3.1** (23.0×)	17.8	**3.1** (5.8×)
	`mul`	1316.7	**206.2** (6.4×)	116.7	**57.4** (2.0×)
Throughput (GOPS)	`add`	**3.6** (1.4×)	2.6	**14.4** (5.5×)	2.6
	`mul`	**0.2** (5.0×)	0.04	**2.2** (15.7×)	0.14
Energy (pJ/Op)	`add`	4.7	4.8	1.2	4.8
	`mul`	112.5	221.3	9.0	58.1

Table 4.4: Detailed Comparsion Table Between BS-VRAM and BP-VRAM.

data (and thus the energy), while BP-EVEv1 essentially always operates on 32-bit data. For example, BS-EVEv1 requires only 1.2 pJ for an 8-bit addition, while BP-EVEv1 requires 4.8 pJ. BP-EVEv1's energy efficiency could be improved by using transmission gates to segment the MCC. For 32-bit data, BP-EVEv1's `add` energy is similar to BS-EVEv1, but bit-parallel `mul` consumes more energy due the multiplicand shifts required to generate partial-products (i.e., more reads and writes).

4.5 Comparison to Prior Work

Processing-in-memory has shown promise in increasing performance and energy efficiency by moving the computation closer to the memory [AYMC15, BTKK04, KKC+16, OCS98, PAC+97b, PJZ+14, SKF+13, ZJL+14, ZAS+14]. Specifically, recent work on processing-in-SRAM uses bit-line compute to push logic into the SRAM with minimal area-overhead. Prior work demonstrates the potential for bit-line compute by transforming the cache subsystem of a chip multi-processor into different engines: a bit-parallel bit-wise logic engine [AJS+17]; a fixed-function accelerator for neural networks [EWW+18]; and a SIMT accelerator [FMD19].

Jeloka et al. were the earliest to introduce the concept of bit-line compute, where computations are performed digitally inside the SRAM [JASB16, JASB15]. Wang et al. propose CRAM which extends bit-line compute to an 8T SRAM and include support for integer arithmetic [WWE+19]. Instead of performing the computation vertically, the 8T bitcell allows the computations to be performed horizontally in the compute bitlines. Additional functionality is implemented using bit-

serial logic in the periphery with appropriate multiplexing. Sub-array banking helps mitigate area overhead by sharing column decoders and compute logic with neighboring sub-arrays.

Table 4.5 shows a comparison of BS-VRAM and BP-VRAM against prior work. Despite CRAM's single-cycle read-write, BS-VRAM is has higher throughput because: all sub-arrays in BS-VRAM can be active at the same time while CRAM can only use half of the subarrays due to banking; per sub-array, BS-VRAM has twice computing bit-lines resulting in twice the ALUs compared to CRAM; and BS-VRAM operates at a higher frequency. The use of a 6T bitcells help BS-VRAM and BP-VRAM in occupying only half the area of CRAM. Energy efficiency of CRAM is roughly 3× higher than BS-VRAM because CRAM uses lower wordline voltage while performing a bit-line computation as well as utilizing a more optimized sense-amp that reduces read and bit-line computation energy.

Although CRAM achieves 2–3× higher energy efficiency in 8-bit and 32-bit `mac`, the gap between BS-VRAM and CRAM can easily be closed by scaling the BS-VRAM supply voltage. Considering BS-VRAM achieves around 16–18× higher throughput, by scaling the voltage to 0.6 V, BS-VRAM can achieve similar energy efficiency to CRAM while maintaining higher throughput.

4.6 Conclusion

Leveraging in-situ processing-in-SRAM opens a rich design space for vector accelerators by reducing area and energy costs. Considering BS-EVEv1 and BP-EVEv1 as representative design points for bit-serial and bit-parallel approaches, our exploration shows that bit-serial achieves higher throughput compared to bit-parallel, while bit-parallel has lower latency. Both approaches incur comparable area overhead. Although the bit-serial has lower energy, we believe adding better μop support for some macro-operation can bridge the gap. Finally, both designs share a significant amount of circuitry. A reconfigurable design is possible by adding multiplexing to break the addition chain into individual adders, thus transforming bit-parallel into bit-serial and vice-versa.

Paper	VRAM		ISSCC'19	JSSC'16	VLSI'17
	BS	BP	[WWE+19]	[JASB16]	[ZXY+17]
Technology	28nm	28nm	28nm	28nm	40nm
Voltage	0.9V	0.9V	0.9V	0.9V	0.9V
SRAM Capacity	128kB	128kB	128kB	128kB	128kB
SRAM Macro	4kB	4kB	16kB	0.5kB	8kB
SRAM Bitcell	6T	6T	8T	6T	10T
Precision	Arb.	32b	Arb.	Arb.	Arb.
Freq (MHz)	900	645	225	594	90
Area (mm^2)*	1.1	1.1	2.7	0.7	1.28
Logic Ops	✓	✓	✓	✓(a)	✓(b)
Basic Int Ops	✓	✓	✓		
Cmplx Int Ops	✓	✓	✓(c)		
Cmp Ops	✓	✓	✓(d)		
Search			✓	✓	
FX Ops	✓	✓			
FP Ops			✓		
8b MAC GOPS	76.0	4.5	4.2	n/a	n/a
8b MAC GOPS/W	115.5	17.2	245.5	n/a	n/a
32b MAC GOPS	6.4	1.2	0.4	n/a	n/a
32b MAC GOPS/W	9.0	4.5	22.5	n/a	n/a

Table 4.5: Comparsion to Prior Work – Area is extrapolated to a full chip with 128kB total capacity considering 80% density; BS-VRAM and BP-VRAM consider 10% overhead for a controller; Cmplx = Complex, Cmps = comparators, FX = fixed-point, FP = floating-point. Logic ops: `and`, `nand`, `or`, `nor`, `xor`, `xnor`. Basic int ops: `add`, `sub`. Cmplex int ops: `mul`, `udiv`, `rem`. Cmps ops: `slt`, `sle`, `sgt`, `sge`, `seq`. FX Ops: `addfx`, `subfx`, `mulfx`, `udivfx`. FP Ops: `addfp`, `subfp`, `mulfp`, `udivfp`. (a) limited to `and`, `nor`. (b) limited to `and`, `nor`, `xor`. (c) limited to `mul`, `udiv` (d) limited to `slt`, `sgt`, `seq`. n/a = the corresponding work does not support this functionality.

```
  loop:
1   blc           addr_a, addr_b
2   wb.add        addr_c
    ; j_n_done_0 loop
```

(a) `add` in BS-VRAM

```
1   blc           addr_a, addr_b
2   wb.add        addr_c
```

(b) `add` in BP-VRAM

```
1   set_cin       1
2   wb_mask       <(1)
  init:
3   wr            addr_C <(0)
    ; j_n_done_0 init
  iter:
4   rd            addr_B
5   wr_mask.and
  iter_add:
6   blc           addr_C, addr_A
7   wb.add        addr_C
    ; j_n_done_1 iter_add
8   j_n_done_0    iter
```

(c) `mul` in BS-VRAM

```
1   wr            addr_c <(0)
2   rd            addr_a
3   wb.and t0
4   rd            addr_b
5   wr_mask.and
  iter:
6   blc           addr_c, t0
7   msb.wr.add    addr_c
    ; srl
8   rd            t0
9   wb.add        t0
    ; j_n_done_0 iter
```

(d) `mul` in BP-VRAM

Figure 4.2: `add` **and** `mul` **Macro-Operations –** Each macro-op is used as `op c, a, b`, where `a` and `b` are inputs and the resulting value would be stored in `c`. `addr_X` is the row address of variable `X`. Labels are at beginning of line with colon. Control μops use labels as destination to redirect execution flow. `<(X)`: indicates setting the data_in port of the VRAM to the specified value `X`. Semicolon (`;`) indicates a mini-op composed of two μops.

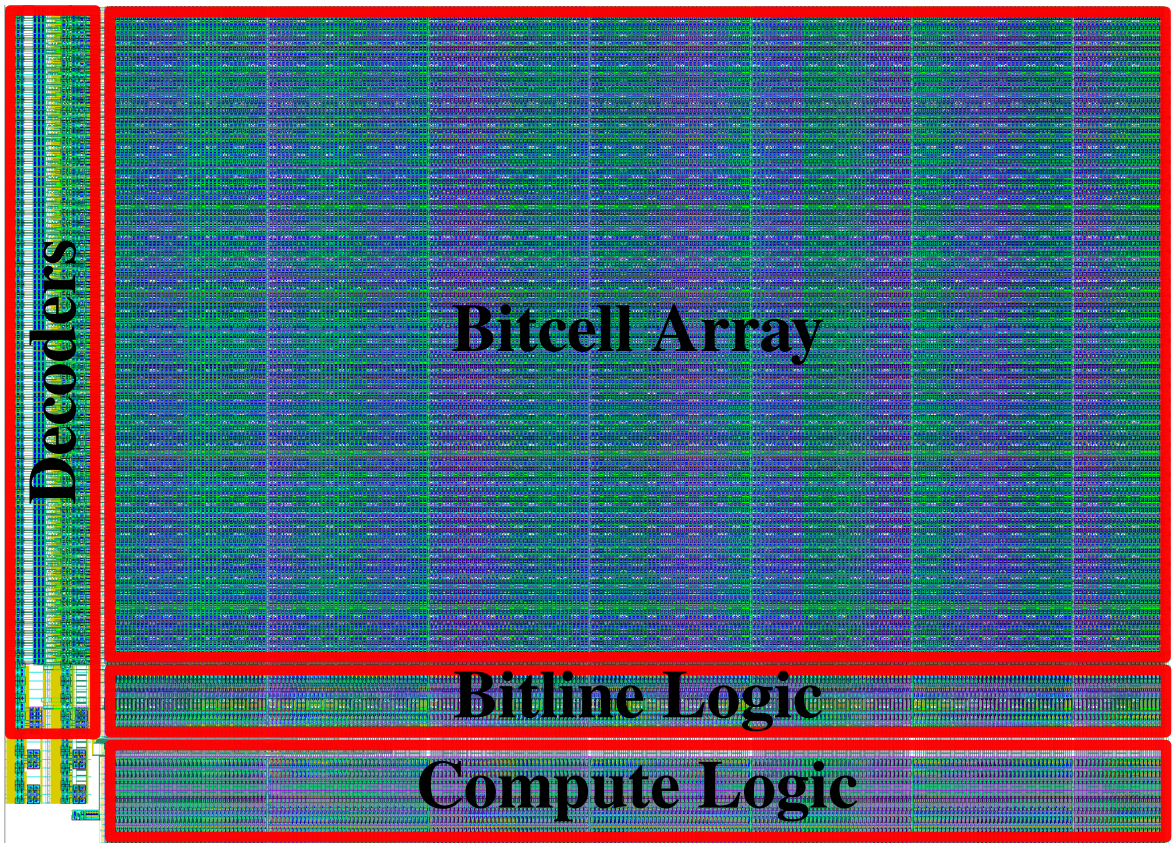


Figure 4.3: BS-VRAM Layout – BP-VRAM have identical floorplan and similar sizing.

CHAPTER 5
EVEV2: CIRCUITS-TO-ARCHITECTURE EXPLORATION OF EPHEMERAL VECTOR ENGINES

Chapter 4 showed that bit-serial approach for S-CIM vector ALUs achieves high throughput while bit-parallel approach achieves low latency. These findings motivates further research into a new approach in-between bit-serial and bit-parallel. In this chapter, I present bit-hybrid–a novel approach to S-CIM execution balancing throughput and latency. In bit-hybrid execution, elements are broken into `n`-bits segments. The hardware of bit-hybrid execution, then, processes each segment in-parallel while it processes segments serially. By choosing the size of a segment, bit-hybrid is able to optimize for the amount of cycles needed per operation (latency) and for the number of elements being processed in-parallel (throughput). Moreover, the cycle-time penalty incurred by bit-parallel approach is mitigated in bit-hybrid approach by choosing segments with smaller size. To evaluate the architectural aspect of EVE, I build a cycle-level model and collect run-time statics of hand-vectorized applications being executed on EVE. To quantify the performance of EVE, I use the baseline models detailed in Chapter 3. The findings in this chapter shows that EVE is able to balance throughput with latency achieving performance comparable to the performance of an aggressive decoupled vector engine while incurring area overhead equivalent to that of an integrated vector unit.

5.1 Introduction

As technology scaling fails to provide regular improvements in transistor performance and efficiency [EBA+11, CDGT17], there is a resurgence of interest in vector architectures demonstrated by ARM SVE [Ste16, SBB+17b] and RISC-V RVV [RISC-V Foundation19]. Traditionally, vector execution has been achieved either through simplified subword packed SIMD units or through aggressive long-vector engines [Rus78, DVWW05, KTHK03, TNH+06]. There is an emerging trend towards next-generation vector architectures, which provide unified abstractions suitable for a variety of different micro-architectures and implementations (see Table 5.1). To implement these next-generation vector architectures, one can either use an integrated vector unit (IV) or a decoupled vector engine (DV). Integrated vector units are typically tightly coupled into the pipeline of the control processor. These units incur lower area overhead as they reuse the control processor

Attribute	Packed SIMD	Long Vector	Next Generation
Length	fixed, short	scalable, long	scalable
Element Width	variable	fixed	variable
Predication	limited	full	full
Cross-Element Ops	full	limited	full
Memory Gather/Scatter	limited	full	full
Integration	integrated	decoupled	either
Speculative Execution	yes	no	either
Compute Pipeline	integrated	decoupled	either
Memory Bandwidth	modest	large	either
Memory Latency	low	high	either

Table 5.1: A Summary of Vector Architectures.

execution hardware and often support short hardware vector lengths. Decoupled vector engines, on the other hand, are often loosely coupled with the control processor and incur higher area overhead as they use aggressive execution hardware and support long hardware vector lengths. There is a fundamental tension between performance and area among these next-generation vector micro-architectures: (1) integrated vector units achieve modest performance in accelerating regular data-parallel workloads while costing modest area overhead; (2) decoupled vector engines have significantly better performance at significantly higher area overhead. This chapter seeks to address this tension: **Is it possible to achieve the performance of decoupled vector engines with the area overhead of integrated vector units?**

Compute-in-memory (CIM) is a novel approach to reduce the area overhead associated with accelerating data-parallel kernels, offering a promising path to solving this tension. There are multiple flavors of CIM targeting different technologies: DRAM-based compute-in-memory (D-CIM) [LDL+18,LNM+17,SLM+17,IGKR19], RRAM-based compute-in-memory (R-CIM) [CLX+16, SQLC17,SZQ+18,QJZ+20,CRS+20,LXZ+20,CXZ+17], and SRAM-based compute-in-memory (S-CIM) [STH+21,SQR+20,AJR+19,AJLR18,JASB15,WWE+19,AJS+17,EWW+18,FMD19]. This work focuses on S-CIM since it can be readily implemented in current state-of-the-art processes and enables closer integration with general-purpose processors. Prior work on S-CIM leverages bit-line computation [JASB15] to perform simple bit-wise logical operations through a single read of a traditional SRAM. Complex integer, fixed-point, and floating-point operations can be executed in-situ by additional peripheral hardware. There are two key challenges when leveraging

S-CIM to accelerate data-parallel workloads: (1) **S-CIM Programming:** offering a compelling abstraction enabling wider applicability and flexible programmability, and (2) **S-CIM Serialization Latency:** mitigating the latency overhead incurred by the serialized nature of S-CIM in both compute and memory operations.

Duality cache [FMD19] is a recent work that leverages S-CIM techniques to accelerate data-parallel workloads in a coarse-grain fashion. Duality cache addresses the S-CIM programming challenge by adopting a SIMT abstraction. The shared last-level cache (LLC) in a chip multiprocessor (CMP) can be reconfigured to create a large SIMT-style execution engine on demand. Leveraging S-CIM techniques, memory arrays in the LLC are used to create a large number of bit-serial arithmetic and logical units (ALUs). Due to the use of bit-serial execution, duality cache suffers from high latencies (i.e., thousands of cycles) in arithmetic operations. Incoming data need to be transposed to fit the required layout for bit-serial execution, incurring area overhead, additional latency, and reduced memory bandwidth. The fully-transposed data along with the requirement for all registers of a given thread to exist in the same column forces duality cache to allocate architectural registers to neighboring SRAM arrays. As a result, extra move operations are required to execute instructions between registers in different SRAM arrays. Duality cache requires extreme levels of parallelism to achieve compelling speed-ups; thus requiring most of the last-level cache for execution. As a result, these limitations constrain the use of duality cache to a coarse grain offloading model.

Though most of the prior work on S-CIM uses bit-serial execution, VRAM [AHAA$^+$20] proposes bit-parallel as well as bit-serial execution. VRAM shows that bit-parallel execution incurs similar area overhead to that of bit-serial. VRAM attempts to address the S-CIM serialization latency challenge by using bit-parallel execution. Whereas a bit-serial approach targets higher throughput, a bit-parallel approach can achieve lower latencies in compute and memory operations. CRAM [WWE$^+$19] is another work on S-CIM that addresses the S-CIM serialization latency challenge in bit-serial execution by utilizing 8T-SRAM bit cells; thus, lowering the serialization overhead. The compute operations in CRAM still require high latencies due to their bit-serial nature. Due to the use of 8T-SRAM bit cells, CRAM also incurs a high area overhead.

In this chapter, we propose ephemeral vector engines (EVE). EVE leverages prior work on S-CIM to build efficient next-generation vector accelerators with support for all 32-bit integer instructions in the RISC-V vector extension [RISC-V Foundation19]. By way-partitioning a pri-

vate L2 cache, each core in a CMP can dynamically create an ephemeral private vector engine to execute data-parallel workloads efficiently. While state-of-the-art S-CIM leverages bit-serial execution, EVE opts for a novel bit-hybrid approach that balances the throughput and latency of different vector instructions. Elements are broken down into *n*-bit segments that are computed in bit-parallel fashion, while the segments themselves are computed serially. SRAM arrays in the partitioned cache ways are replaced with EVE SRAM, which is a 6T-SRAM capable of bit-line computation with novel bit-peripheral circuits added to enable complex bit-hybrid operations. Additional units are added to: control instruction execution, handle memory instructions, and perform reduction/cross-element instructions.

We use a vertically integrated research methodology to evaluate EVE on two important metrics: area and performance. We generated a layout of a simplified EVE SRAM through a modified version of OpenRAM [GSA+16], which is a Python-based open-source SRAM generator. The layout is composed of a simplified version of the EVE circuits added to a bit-line compute capable 6T-SRAM. We used this layout to estimate the area overhead of EVE circuits as well as the cycle-time penalty over a traditional SRAM array. For performance evaluation, we built a cycle-approximate model of EVE in gem5 [BBB+11] and simulated its performance on vectorized implementations of applications from the Rodinia [CBM+09] and RiVEC [RHP+20] benchmark suites. To quantify EVE's performance, we built a cycle-approximate model of: (1) a high-performance decoupled vector engine loosely based on Tarantula [EAE+02], and (2) an integrated vector unit with hardware vector length matching conventional SIMD width [SBB+17a, RBGZ19]. EVE is able to achieve comparable speedups to a decoupled vector engine, and a 3.5 $\times$ speedup over an integrated vector unit with as little as 5% area overhead.

Our main contributions are: (1) a novel bit-hybrid execution approach for SRAM-based compute-in-memory (S-CIM); to our knowledge, this is the first work that proposes using bit-hybrid approach for S-CIM to balance throughput and latency; (2) a template for EVE circuits that enables building EVE-*n* SRAM capable of executing vector operations targeting *n*-bit-hybrid execution; (3) an exploration of the cycle-level impact and trade-offs of targeting different *n*-bit-hybrid execution configurations (i.e., n = 1, 2, 4, 8, 16, 32); (4) the novel EVE micro-architecture that enables transforming private L2 cache ways into ephemeral vector engines; (5) a detailed evaluation of different EVE design points exploring the trade-offs and impact of various design parameters.

5.2 Taxonomy of Vector S-CIM

Previous work has examined two different design points for a S-CIM vector engine: bit-serial [AHAA+20, WWE+19, EWW+18] and bit-parallel [AHAA+20]. In bit-serial execution, the S-CIM vector engine breaks down each 32-bit element into one-bit segments (i.e., each element consists of 32 segments). The engine processes one segment of a given element in a cycle, and processes the rest of the segments serially. In bit-parallel execution, the S-CIM vector engine processes each 32-bit element as one segment, where the segment size is 32 bits. We define the *parallelization factor* of a S-CIM vector engine as the width (in bits) of a segment from an element that the engine can process in parallel. Using this generalization, bit-serial designs are expressed as S-CIM vector engines with a parallelization factor of one. Bit-parallel designs are considered to be S-CIM vector engines with a parallelization factor of n, where n is the element size supported.

By varying the parallelization factor, there is a spectrum that describes different S-CIM vector engines. On one end of the spectrum, S-CIM vector engines with a parallelization factor of one (i.e., bit-serial) achieve higher throughput at the cost of higher latency. On the other end of the spectrum, S-CIM vector engines with a parallelization factor of 32 (i.e., bit-parallel, assuming 32-bit elements) achieve lower latency but lower throughput as well. To explore this spectrum, we construct an analytical model that calculates the throughput and latency for a vector addition and multiplication. As one of the principles of S-CIM is to have all input elements for an operation in the same column, the model requires all vector registers, with all their elements, to be stored in the same SRAM array.

Element Layout & Available In-Situ ALUs – Figure 5.1 shows the layout of vector registers holding 8-bit elements in a 16×16 SRAM while varying the parallelization factor. Considering an ISA that supports one vector register, with parallelization factor of one, each element occupies a single column in the SRAM. As a result, half the SRAM is occupied providing storage for 16 elements. By increasing the parallelization factor, the segment size increases while the number of segments decreases. As each segment occupies a row, the number of elements for the vector register decreases; thus, lowering the number of available in-situ ALUs.

Starting with a parallelization factor of one, as the number of supported vector registers increases, the utilization of the SRAM grows as well. Each column in the SRAM can be used as an ALU performing in-situ computation of corresponding elements from different vector registers.

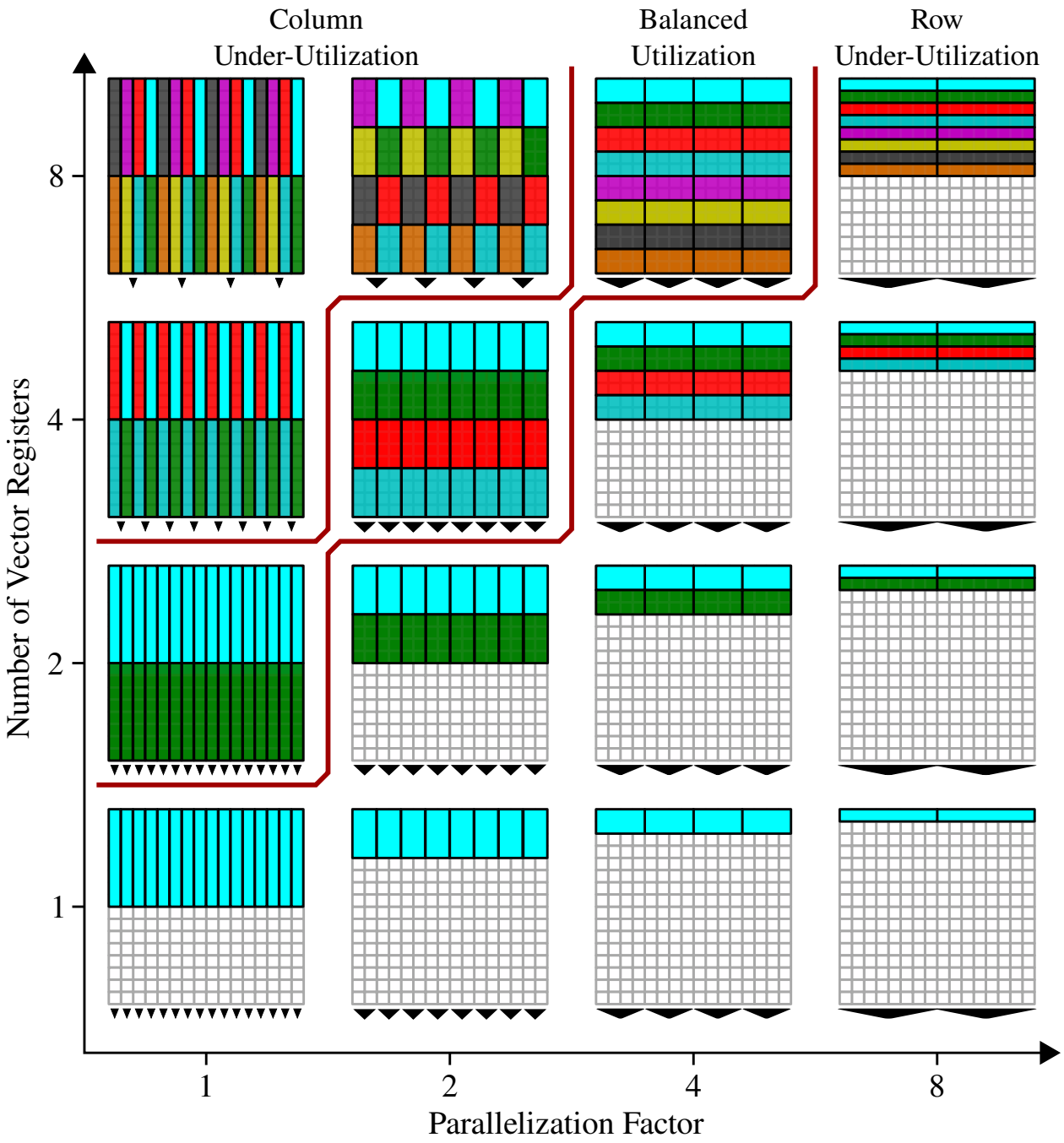


Figure 5.1: Data Organization in S-CIM SRAM Array – varying number of supported vector registers and the parallelization factor. Grey boxes represent bit-cells. Each vector register is assigned a unique color. Elements belonging to the same vector register are shown as boxes with the same color. Triangles at the periphery represent in-situ ALUs.

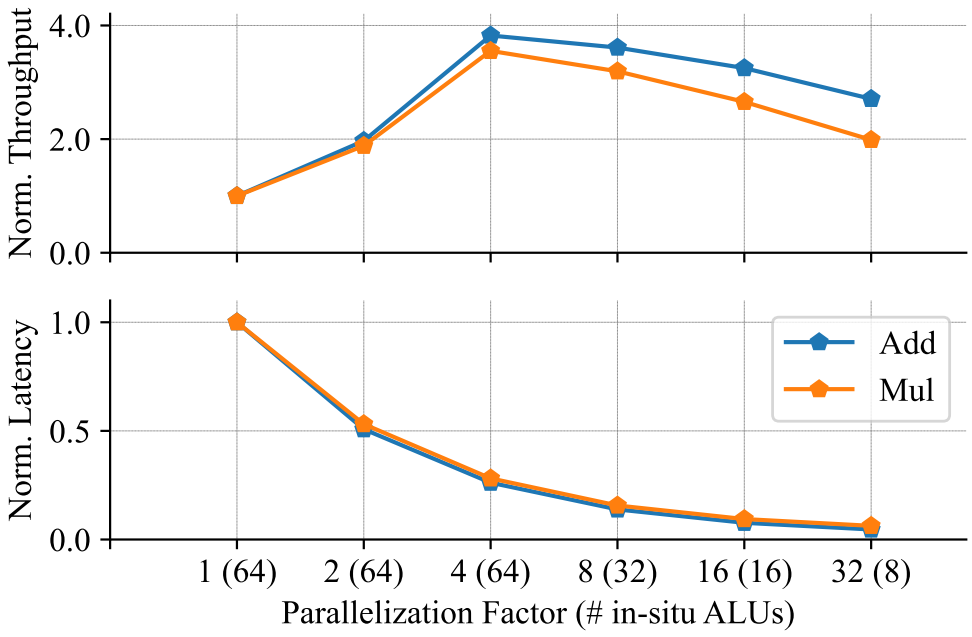


Figure 5.2: Latency and Throughput of Add/Logic and Multiply vs. Parallelization Factor – achieved by a 256 × 256 S-CIM SRAM assuming 32 vector register support normalized to latency and throughput of parallelization factor of 1. The number of in-situ ALUs for each parallelization factor is shown between parentheses in the X-axis.

As Figure 5.1 shows, the SRAM reaches balanced utilization with two vector registers. However, to support more vector registers, some of the columns are repurposed to hold the additional registers, reducing the number of in-situ ALUs. With higher parallelization factor, the SRAM can support more vector registers without reaching column under-utilization, as each vector register is composed of fewer segments. But, high parallelization factors struggle to increase the row utilization of the SRAM unless provided with more vector registers. Figure 5.2 shows the latency and throughput for vector addition and multiplication as the parallelization factor increases for a S-CIM vector engine implemented using a 256 × 256 SRAM with 32 vector register support.

Latency – Figure 5.2 shows the latency for a vector addition and multiplication normalized to that of parallelization factor of one. As the parallelization factor increases, the number of segments decreases accordingly; because the number of cycles required to perform a vector addition and multiplication correlates to the number of segments, the latency of these operations decreases as well. However, one can observe from Figure 5.2 that the latency is not linearly correlated with

the number of segments. The insight behind this is the control overhead induced by initializing counters and control branching to process segments serially.

Throughput – Figure 5.2 shows the throughput for a vector addition and multiplication. Starting with a parallelization factor of one, the S-CIM vector engine experiences column under-utilization (as previously exemplified in Figure 5.1). As the parallelization factor increases, the elements are composed of fewer segments; as a result, the column under-utilization is lessened and the throughput increases despite having the same number of in-situ ALUs (since the latency of the in-stu ALUs decreases). The throughput peaks when the parallelization factor reaches four, achieving balanced utilization for the S-CIM SRAM. Beyond balanced utilization, the S-CIM vector engine experiences row under-utilization and the number of in-situ ALUs is decreased causing a drop in the throughput. Although, as discussed previously, the latency will decrease further as the parallelization factor increases, the decrease is not enough to compensate for the reduction in the number of in-situ ALUs.

Key Insights – This section shows that both bit-serial and bit-parallel are sub-optimal and induce problems such as row and columnunder-utilization; moreover, the taxonomy shows that targeting bit-hybrid, which is neither bit-serial nor bit-parallel is the optimal design decision. *To our knowledge, EVE is the first to make this observation and explore bit-hybrid design space.* Previous work has predominantly explored either bit-serial or bit-parallel. Duality cache, which leverages bit-serial execution, tried to mitigate column under-utilization by dividing vector registers into four banks and introducing explicit move instructions inserted automatically through compiler analysis. As a result, duality cache binaries are not optimal nor portable across designs with different sub-array sizes. EVE's elegant bit-hybrid approach alleviates column under-utilization without the need for compiler solutions, making its binaries optimal and portable regardless of underlying hardware micro-architecture.

5.3 EVE Circuits

EVE transforms traditional 6T-SRAM into a vector execution unit with additional peripheral hardware. EVE leverages the circuit design from prior work on VRAM [AHAA+20] as the peripheral hardware with additional modifications to expand its functionality, as well as proposing a new circuit design for supporting bit-hybrid execution. Considering a 32-bit precision, EVE-1 utilizes a

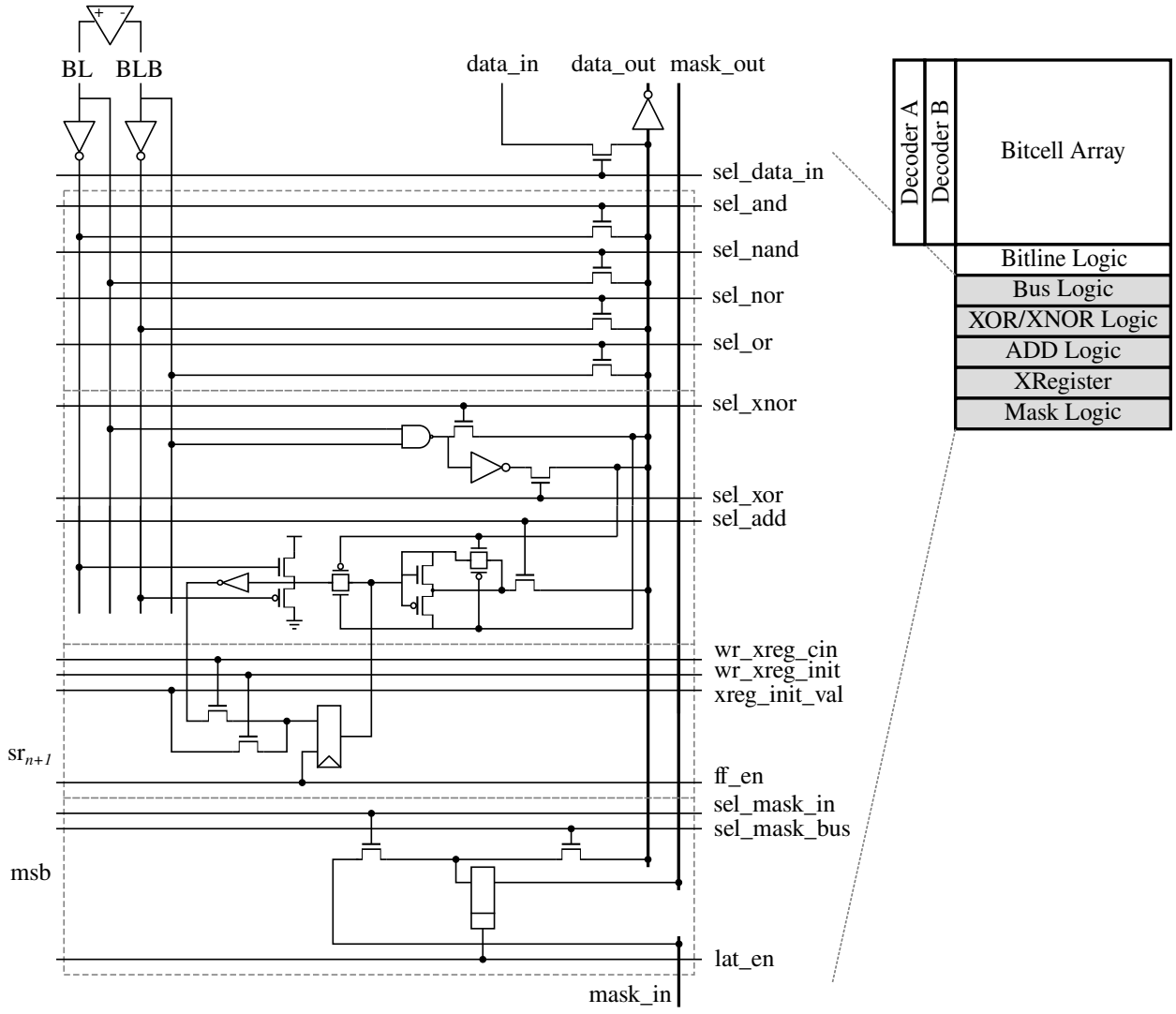


Figure 5.3: EVE-1 Bit-Serial Circuits – Composed of five different stacks: bus logic, XOR/XNOR logic, add logic, XRegister, and mask logic.

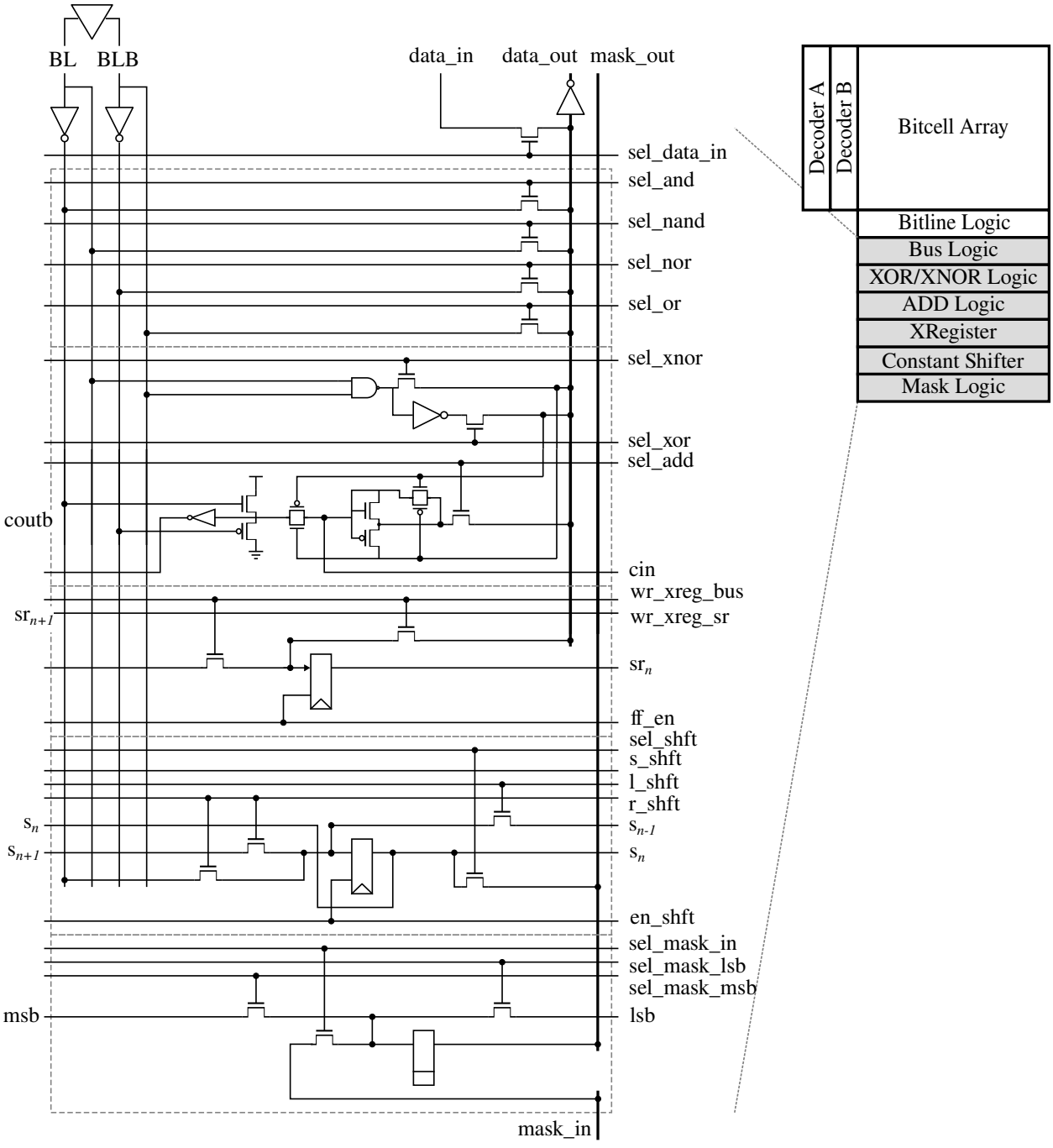


Figure 5.4: EVE-32 Bit-Parallel Circuits – Composed of six different stacks: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifters, and mask logic.

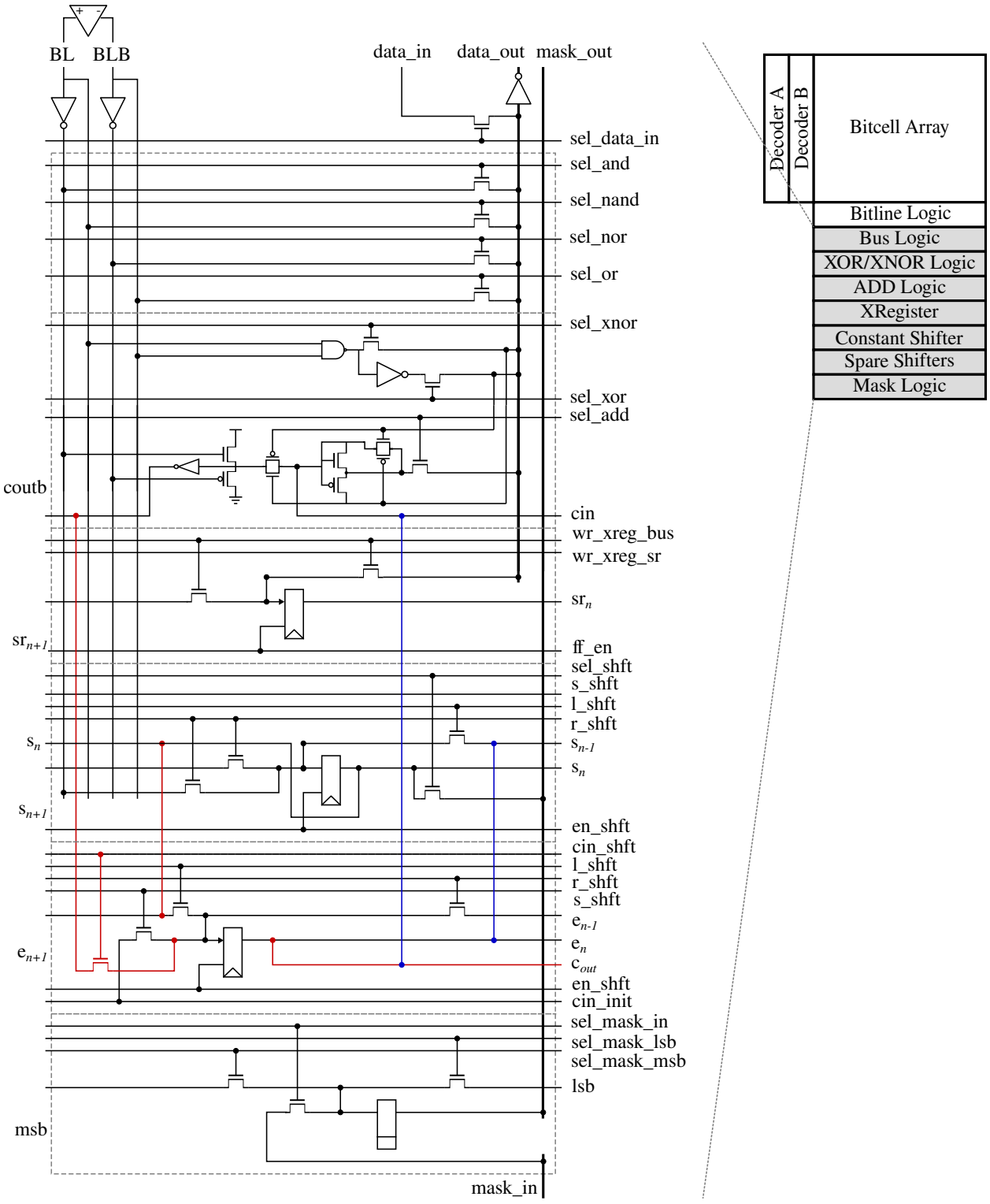


Figure 5.5: EVE-1 Bit-Serial Circuits – Composed of seven different stacks: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifter, spare shifter, and mask logic; Blue lines indicate connections and components existing only in the LSB column of the segment. Red lines indicate connections and components existing only in the MSB column of the segment.

bit-serial execution approach, and EVE-32 utilizes a bit-parallel execution approach, while EVE-*n* targets an *n*-bit-hybrid execution approach.

The EVE circuits are composed of different stacks of logic and take as an input the outcome of a bit-line compute operation. To perform bit-line compute, the differential sense-amplifiers in the traditional 6T-SRAM are modified to support reconfigurable differential and single-ended modes. An extra address decoder is added to allow the selection of two wordlines simultaneously. When performing bit-line compute operation, the two operands (i.e., wordlines) to the operation are selected simultaneously and the sense-amplifiers are set in the single-ended mode. As a result, the sense-amplifiers compute four bit-wise logical operations: `and`, `nand`, `or`, and `nor`. The circuit for an EVE design is a stack composed of a mixture from seven different layers of logic: bus logic, XOR/XNOR logic, add logic, XRegister, mask logic, constant shifter, and spare shifter. Due to the bit-wise nature of the bus logic and XOR/XNOR logic, these layers are the same for the different EVE designs. The bus logic amplifies and selects one of the values computed by the circuit for it to be written back to the SRAM. Meanwhile, the XOR/XNOR logic uses the `nand` and `or` values to compute the `xor` and `xnor` of the operands.

The remainder of the section discusses the design of the rest of the stack for each EVE design. The first, second, and third subsections discuss EVE-1 circuit, EVE-32 circuit, and EVE-*n* circuit, respectively.

5.3.1 EVE-1 Bit-Serial Circuit

For bit-serial execution, EVE adopts the same circuit from BS-VRAM [AHAA+20]. Although BS-VRAM does not support variable shift/rotation, the functionality can be supported through programming the circuits to perform multiple reads/writes to shift the values down the rows without any changes to the circuit. The circuit, shown in Figure 5.3, is composed of five different layers of logic: bus logic, XOR/XNOR logic, add logic, XRegister, and mask logic. Each column of the circuit operates independently to process 32-bit values over the span of multiple cycles. The add logic in each column employs a single block of a Manchester carry chain to perform a one-bit full addition and uses the `xor` and `xnor` of the operands along with the carry-in to calculate the sum and the carry-out of the addition. The XRegister stores the carry bit from the add logic facilitating a bit-serial addition. The mask logic contains a single latch to store the masking value for each

column. The input to the latch can either be one of the values computed by the circuit or an input mask provided to the SRAM.

5.3.2 EVE-32 Bit-Parallel Circuit

The bit-parallel circuit in EVE heavily leverages the BP-VRAM [AHAA+20]. However, as BP-VRAM does not support variable shifts and rotations, extra layers have been added to the stack to facilitate shift/rotation support. The EVE-32 bit-parallel circuit, Figure 5.4, is composed of six layers: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifter, and mask logic. To support bit-parallel execution, sets of 32 columns are grouped together to process 32-bit values in parallel. For the add logic, a Manchester carry chain is used to propagate the carry of the addition and calculate the sum value. Each column contains a block of the Manchester carry chain. The XRegister is configured as a shift-right register spanning the 32 columns. This shift register aids in executing complex operations such as multiplication, division, and shift. The mask logic contains a single latch to store the masking value. As with bit-serial circuit, the input to the latch can either be the values computed by the circuit or an input mask provided to the SRAM through port.

In contrast to BP-VRAM, the EVE-32 bit-parallel circuit contains constant shifter logic, which can be used to execute variable shifts and rotations. After the 32-bit value is loaded, the shifter supports conditional one-bit shift-left and shift-right. The condition for the constant shifter is set to the mask of the column. By leveraging conditional one-bit shifts, a variable shift larger than one can be computed through binary decomposition of the shift amount. For each bit of the shift amount at index i, the constant shifter can perform multiple one-bit shifts adding up to 2^i. By iterating through the shift amount, the conditional one-bit shifts will eventually add-up to the shift amount over time.

5.3.3 EVE-*n* Bit-Hybrid Circuit

The circuit for EVE-n is set at design-time to target a fixed parallelization factor of n (where n is between 2 and 16). The EVE-n bit-hybrid circuit, shown in Figure 5.5, is composed of seven layers: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifter, spare shifter, and mask logic. The circuit processes one n-bit segment from an element in parallel and processes each of the n-bit segments serially. Every n columns of the SRAM are grouped together to form an n-bit-

hybrid execution hardware. The mask logic contains a single latch used to store a mask for a given column. The mask can be set to the XRegister value of either the most-significant column or the least-significant column of the segment. The mask latch can also be loaded with either a value computed by the circuit or an input mask.

For the add logic, an n-bit Manchester carry chain is used to perform full n-bit addition between two segments. The carry is then stored in one of the unused flip-flops in the spare shifter. Then, the flip-flop is wired to provide the stored carry as the carry-in for the Manchester carry chain when adding subsequent segments. Unlike the bit-serial circuit, the XRegister is no longer used to store the carry. As a result, the XRegister can be used as a shift-right register, similar to the XRegister in the bit-parallel circuit, to implement complex operations such as multiplication, division, and shift.

For shifts, the bit-hybrid circuit employs a constant shifter, similar to the bit-parallel circuit. However, the constant shifter processes one n-bit segment. To be able to perform shifts in a bit-hybrid fashion across multiple segments, the bit-hybrid circuit utilizes a newly introduced layer called the spare shifter. As the constant shifter performs either a left or right shift, the spare shifter performs either a right or left shift, respectively. As a result, the spare shifter can store the bits being shifted across different segments. A binary decomposition approach similar to that of the bit-parallel circuit can be used to perform the shifts efficiently. However, indices in the shift amount greater than or equal to $log_2(n)$ implies shifts that are integer multiples of the segment size. These shifts can be performed much more efficiently in the bit-hybrid approach than the bit-parallel approach by conditionally shifting a whole n-bit segment rather than performing conditional one-bit shifts.

5.4 EVE Micro-Programming

To control the circuits detailed in Section 5.3, EVE employs a micro-operation (μop) abstraction that enables writing micro-programs to achieve elaborate and more complex operations. Each μop takes a single cycle to execute and has well-defined inputs, outputs, and side-effects. This section discusses the micro-programming aspect of EVE. The remainder of the section is organized as follows: the first subsection details the different μops implemented by EVE; the second subsection explains how different macro-operations are implemented by sequencing μops.

μOperation	Syntax	Description
`read`	`rd a, src`	read `a` into `src`
`write`	`wr d, src`	write `src` into `d`
`blc`	`blc a, b`	Bit-line compute of `a` and `b`
`lshift`	`lshft`	1-bit shift left
`rshift`	`rshft`	1-bit shift right
`lrotate`	`lrot`	1-bit rotate left
`rrotate`	`rrot`	1-bit rotate right
`mask shft`	`m_shft`	1-bit shift right the XRegister
`cnt_init`	`init cnt, val`	Initialize `cnt` to `val`
`cnt_decr`	`decr cnt`	Decrement `cnt` by one
`bnz`	`bnz cnt, l`	Branch to `l` if `cnt` is not zero
`bnd`	`bnd cnt, l`	Branch to `l` if `cnt` is a decade
`ret`	`ret`	Conclude execution

Table 5.2: Supported EVE Micro-Operations – *m* = {`msb`, `lsb`, `none`}. *src* = {`(n)and`, `(n)or`, `x(n)or`, `add`, `shift`, `data_in`}.

5.4.1 Micro-Operations

Table 5.2 lists all the μops supported by the different EVE-*n* types. There are three types of μops: arithmetic, control, and counter. Arithmetic μops are executed by the circuits detailed in Section 5.3. The other two categories are executed by the control logic. The arithmetic μops are as follows:

Read/Write (`rd/wr a, [dst/src], mask`) – These two μops represent the native SRAM read and write. The μops take as an operand a destination to store resulting read; or a source to the data for the write. Both operations include a mask as an operand to specify which columns in the SRAM are active or inactive to during the read/write execution.

Bit-Line Compute (`blc a, b`) – This μop performs a bit-line compute operation between wordline `a` and wordline `b`. It starts by enabling the two decoders in the EVE SRAM at the same time selecting wordline `a` and wordline `b`. Then, it reconfigures the sense-amplifiers in the single-ended mode. The sense-amplifiers will compute the bit-wise logical operations and feed these values to the rest of the EVE circuitry (as detailed in Section 5.3).

Shift Left/Right (`{l,r}shift`) – This μop shifts left or right the content of the constant shifter by one bit. For EVE-32, only the constant shifter is shifted. However, for EVE-*n*, a spare shifter

is shifted along side the constant shifter; whereas constant shifter is shifted left or right, the spare shifter is shifted in the opposite direction (i.e., right or left, respectively).

Writeback (`wb d, src, mask`) – After executing a `blc`, this μop reads the value computed by `src` and writes it back to the EVE SRAM at wordline `d`. The destination of the write back can also be specified as the mask registers (XRegister in bit-parallel and bit-hybrid; mask logic in bit-serial).

Mask Shift (`mask_shift`) – Conditionally executing μops is an essential functionality to implement more complex operations. To this end, for bit-hybrid and bit-parallel, the `mask_shift` μop allows manipulating a masking value through left shifts without the need for additional SRAM reads. The mask registers can be loaded through the writeback μop.

EVE also includes 12 counters shared by all EVE SRAMs. These 12 counters are grouped in three groups: segment counters (`seg_cnt[0-3]`), bit counters (`bit_cnt[0-3]`), and array counters (`arr_cnt[0-3]`). Each group is initialized to a specific value, which is inferred from the EVE configuration and execution state: segment counters are initialized to number of segments; bit counters are initialized to the size of a segment; and array counters are initialized to the number of active arrays in EVE. When the value in any counter is decremented to zero, the counter is reset to its initial value. There are two sets of flags that track the state of each counter: zero flags, which track whether each counter has reached zero and was reset to its initial value; and binary decade flags, which record whether a counter has reached a binary decade. Counter μops are executed by a unified control logic to manipulate the counters. The first counter μop is: `incr/decr cnt`, which increments or decrements the counter specified by `cnt`. The second μop is: `init_cnt, val`, which forces an initialization of the counter `cnt` to the value `val`—a specified value inferred from the EVE configuration and execution state (e.g., number of segments, segment size, or active SRAM arrays).

To control which arithmetic μop is executed, a unified control logic in EVE executes control μops to manipulate a micro-program counter (μpc). There are two classes of control μops: conditional and non-conditional branches. Conditional branches inspect the corresponding flags for a specified counter and redirect the μpc depending on the condition (i.e., whether the counter has reached zero or a binary decade). The inspected flag for the conditional branch is reset when the branch is taken. Non-conditional branches change the μpc unconditionally. Control μops include a

Each μop is used as `op c, a, b`: `a` and `b` are inputs and result is stored in `c`. `addr_X` is the row address of `X`. Labels are at beginning of line with colon. Control μops use labels as destination to redirect execution flow. `<(X)`: indicates setting the data_in port to `X`. Semicolon (`;`) indicates a mini-op composed of two to three μops.

```
  loop:
1  blc      addr_a, addr_b
   ; decr   seg_cnt[0]
2  wb       addr_c, add
   ; bnz    seg_cnt[0], loop
```

(a) `add`

```
1  rd       addr_b, mask
  iter:
2  blc      addr_c, addr_a
   ; decr   seg_cnt[0]
3  wb       addr_c, add, mask
4  rd       addr_c
5  wb       addr_c, add
   ; bnz    seg_cnt[0], iter
6  mask_shift
   ; decr   bit_cnt[0]
   ; bnz    bit_cnt[0], iter
7  rd       addr_b, mask
   ; decr   seg_cnt[1]
   ; bnz.r  seg_cnt[1], iter
```

(b) `mul`

Figure 5.6: `add` and `mul` Macro-Operations.

`ret` flag, which indicates whether the current micro-program is terminating execution and yielding to the next micro-program.

5.4.2 Macro-Operations

Incoming vector instructions are broken down into one or multiple macro-operations. Any of these macro-operations to be executed on the EVE SRAM are implemented as a micro-program (μprogram), which is a sequence of micro-operation tuples with a micro-program counter (μpc). Each tuple is composed of three μops, one from each of the three categories outlined in Section 5.4.1. These tuples are executed in the following order: counter μop, arithmetic μop, then control μop.

Figure 5.6 shows the bit-hybrid implementation of integer addition and multiplication. Integer addition, shown in Figure 5.6(a), is executed by performing bit-line computation between corresponding segments of the two input elements and then writing the results back into the destination. The carry between the segments is propagated through the XRegister in the circuits. The control required for the addition is straight-forward requiring a simple count-down loop that iterates over the segments. Integer multiplication, shown in Figure 5.6(b), requires more elaborate control with nested loops and different bounds for each loop. An outer loop (`iter`) handles predicated

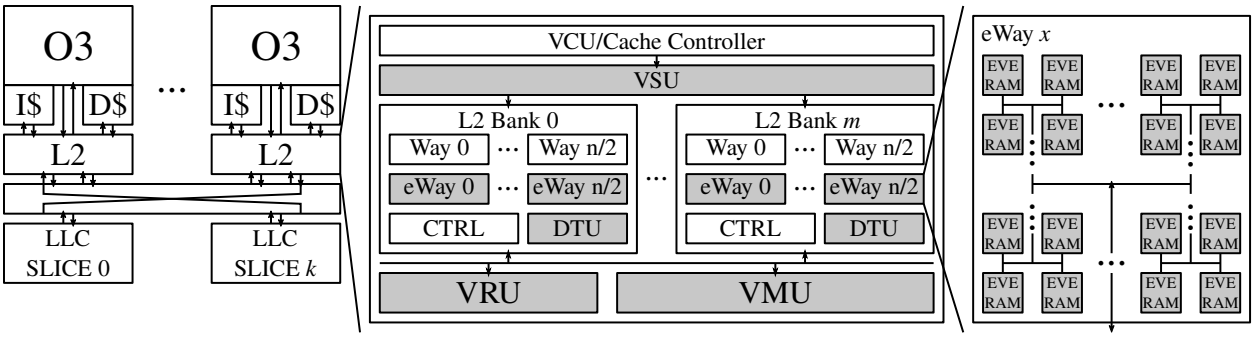


Figure 5.7: EVE Micro-Architecture.

summation for calculating the multiplication (executed for N iterations, where N is the number of segments). The inner loop (i.e., `iter_add`) will perform a single addition (executed for n iterations).

5.5 EVE Micro-Architecture

This section introduces the micro-architecture of EVE, shown in Figure 5.7. To facilitate vector execution in EVE SRAM, a vector control unit (VCU) converts incoming vector instructions into one or multiple macro-operations. Depending on the type of the vector instruction, these macro-operations are broken down into a sequence of micro-operations by the help of either the vector sequencing unit (VSU), vector memory unit (VMU), and/or vector reduction unit (VRU). The stream of micro-operations are executed by EVE SRAMs. The VCU coordinates execution and communitcation between these units and the control processor.

5.5.1 Vector Control Unit (VCU)

When the control processor encounters a vector instruction, it is inserted in a special queue waiting to be sent to EVE. As EVE does not support precise exceptions, the vector instructions are *only* sent to EVE when they are ready to be committed. Once sent, the vector instructions are committed to unblock the commit logic. Then, these vector instructions are sent to the retire stage awaiting a response confirming their execution from EVE. If a writeback is to be expected from a vector instruction (e.g., `vmv.x.s`), the instruction instead stalls the commit logic awaiting a response from EVE with the value to be written back. To manage scalar-vector memory de-

pendency and synchronization, a new vector memory fence instruction (`vmfence`) is introduced. The `vmfence` stalls the load-store-queue (LSQ) in the control processor from executing subsequent memory instructions until the vector fence is committed. Once at the commit stage and all pending scalar stores are performed, the `vmfence` is sent to EVE and stalls awaiting a response.

Once a vector instruction is received, depending on its type, the VCU creates one or more macro-operations. For non-memory and non-cross-element vector instructions, the VCU creates one macro-operation that is executed by the VSU. Once the macro-operation is sent to the VSU, the vector instruction is marked as performed and a response is sent back to the control processor. For memory and reduction/cross-element vector instructions, two macro-operations are created: one macro-operation is sent to either the VMU to initialize the memory operation or the VRU to initialize the reduction operation; the second macro-operation is sent to the VSU to perform the necessary reads and writes from and to EVE SRAMs. For vector memory fences, the VCU stalls waiting for the VMU to drain all pending loads and stores; then, the VCU executes the fence by simply sending a response to the control processor.

5.5.2 Vector Sequencing Unit (VSU)

The vector sequencing unit (VSU) decodes macro-operations into a sequence of μops primitives as explained in Section 5.4. The arithmetic μops are sent to the EVE SRAMs for execution as detailed by Section 5.3, while the control μops are executed by the VSU. The VSU consists of a micro-program counter (μPC), a ROM containing the implementation of various macro-operations, and a set of counters. The μPC points to the current μop tuple being executed in the ROM. The set of counters help in executing the various control μops (e.g., `bnz` and `bnd`). The VSU adapts a VLIW-style encoding for the μops. Each cycle the VSU fetches a tuple consisting of three different μops: arithmetic μop, counters μop, and control μop. These micro-operations can be executed simultaneously. Executing control μops with the `ret` flag set will cause the VSU to stop executing the current vector instruction and return control back to the VCU.

5.5.3 Vector Memory Unit (VMU)

Executing memory vector instructions requires upwards of three macro-operations. One macro-operation is for the vector memory unit (VMU), shown in Figure 5.8, to generate the appropriate

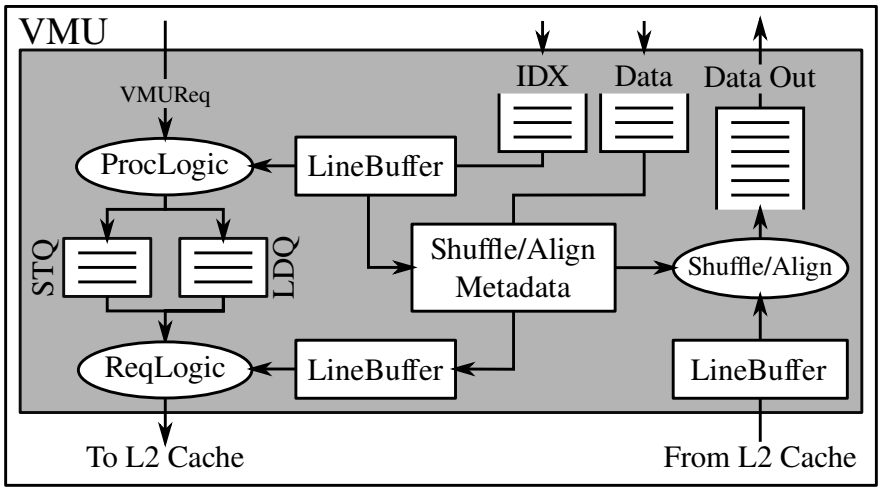


Figure 5.8: VMU Micro-Architecture.

requests to the memory sub-system. Another one or two macro-operations are for the VSU to read indices required for the memory requests and to perform a read/write for the data from/to the EVE SRAMs. The VMU guarantees cache-line alignment for the generated requests. As for memory gather instructions (i.e., indexed loads), the VMU generates a request for each element and then performs the required gather operation to form a single line to be written back into the EVE SRAMs. An extra port in the TLB of the control processor is dedicated for the VMU to translate virtual addresses for incoming requests.

5.5.4 Vector Reduction Unit (VRU)

The vector reduction unit handles reduction instructions (e.g., `vredsum.vs`) as well as vector cross-element instructions (e.g., `vrgather`). Assuming `B` bits as the read/write bandwidth of the EVE SRAM, the VSU is able to stream $\frac{B}{n}$ elements (i.e., `E`) into the VRU for a given n-bit-hybrid configuration. The elements will be streamed one segment at a time over the span of $\frac{32}{n}$ for 32-bit element precision. The VRU contains `E` ports with each port having a fast lightweight detranspose logic. After all the segments are received, the VRU begins performing a dot-operation between the received `E` elements and the previously reduced `E` elements. Finally, once all elements have been streamed and the dot-operation between the different element groups has finished, the VRU starts a linear reduction operation. Then, the VSU issues a read μop to read the calculated reduction and either write it to a destination vector register or back to the control processor.

5.5.5 Reconfigurability

To support EVE, half of the private L2 cache ways are designed with EVE SRAMs instead of vanilla SRAMs. These ways operate as normal SRAMs performing reads and writes for L2. To spawn EVE, the associativity for the private L2 cache is reduced by half and the L2 cache is way-partitioned into: cache ways, and EVE ways. The cache ways constitute the new L2 and continue to service the core with no impact, aside from the halved associativity. The overhead of setting up EVE consists of reconfiguring the remaining EVE ways through invalidating cache lines residing in these ways. For dirty cache lines, the invalidation causes a writeback to the LLC. For clean cache lines, the invalidation causes the number of sharers to be decremented in the LLC, depending on the cache protocol. Since the cache hierarchy is inclusive, the invalidation and write-back to LLC should scale linearly with the number of cache lines (i.e., each cache line should incur constant number of cycles to invalidate in L1 and write-back to the LLC). To achieve this, a simple FSM machine and a counter can iterate through the reconfigured EVE ways causing the L2 to stall. However, the core can continue to be serviced from L1 as long as no misses are encountered. To reconfigure EVE ways back to L2 cache, there is no overhead and simply the cache associativity can be increased with all the cache lines returned to the L2 cache initialized as invalid.

5.6 Circuits Evaluation

This section discusses the evaluation methodology and results of EVE's circuits.

5.6.1 Methodology

To evaluate the circuit of the different EVE designs with varying parallelization factors, we leveraged OpenRAM [GSA+16], which is an open-source Python-based memory generator. OpenRAM was modified to generate layout of an SRAM memory capable of bit-line computation on a 28nm technology node. Bit-line computation required an additional decoder to the SRAM as well as modifying the sense-amplifiers to support reconfigurable modes: differential and single-ended. A simplified version of the EVE-1 and EVE-32 circuits were implemented and used as an estimate for the proposed circuits. The simplified version lacked support for constant shifting; but, otherwise, matched the rest of the proposed circuits. The simplified circuits were implemented

IO	Single-issue in-order RV64GC core: • L1I: 1-cycle-hit 4-way 32KB • L1D: 2-cycle-hit 4-way 32KB • L2: 8-way 8-bank 8-cycle-hit 512KB
O3	Out-of-order 8-way RV64GC core: • Same L1I, L1D, and L2 as *IO*
O3+IV	Small vector unit integrated into O3: • Same L1I, L1D, and L2 as *O3* • IV: 4-element VL, out-of-order issue, 3 exec pipes
O3+DV	Decoupled vector engine connected to O3: • Same L1I, L1D, and L2 as *O3* • DV: 64-element VL, in-order issue, 4 exec pipes
O3+EVE	EVE engine connected to O3: • Same L1I and L1D as *O3* • L2: 4-way 8-bank 8-cycle-hit 256KB in vector mode • EVE-*x*: in-order issue, 1 exec pipe • VL (elements): EVE-{1,2,4}=2048, EVE-8=1024, EVE-16=512, EVE-32=256
LLC	Same for all systems: 16-way, 12-cycle-hit & 2MB
Memory	Same for all systems: single-channel DDR4-2400

Table 5.3: Simulated Systems.

and laid out on the aforementioned 28nm technology node and they passed DRC and LVS checks. After the simplified circuits were verified through SPICE simulations, they were added as modules to OpenRAM. Then, OpenRAM was modified to include these circuits as part of the peripheral hardware for EVE SRAM. By leveraging the flexibility of OpenRAM, once OpenRAM was capable of generating EVE-1 and EVE-32 SRAMs, multiple designs of bit-parallel EVE SRAM (i.e., EVE-32) with varying bit-precision (b) were generated effortlessly (i.e., b = {2, 4, 8, 16, 32}). Each b bit-parallel EVE SRAM was used as proxy to analyze b-bit-hybrid EVE SRAMs as both designs have the same critical combinational path (i.e., carry propagation). Cycle-time for the EVE-n SRAM was estimated through the extracted netlist for the SRAM generated by OpenRAM. Area overhead was estimated by inspecting the layout of the simplified circuits adding estimated area from the missing stacks.

5.6.2 Results

OpenRAM was used to generate the layout of a 256×128 simplified EVE SRAM, which includes the simplified EVE circuit in its peripheral hardware. The layout passed the DRC and LVS checks for the 28nm technology node. Schematic circuits for the full bit-serial, bit-parallel, and bit-hybrid EVE configuration was built and used to verify correct functionality.

According to the layout, the 256×128 simplified EVE SRAM incurs 8.2% area overhead compared to a vanilla 28nm SRAM generated by OpenRAM. By estimating the additional stacks for the different designs, EVE-1 incurs 9.0% area overhead, EVE-x (bit-hybrid) incurs 15.6% area overhead, and EVE-32 incurs 12.6% area overhead. An EVE SRAM is composed of two banked 256×128 sub-arrays, further reducing the area overheads by half: EVE-1 (4.5%), EVE-x (7.8%), and EVE-32 (6.3%). As for the cycle-time, the baseline vanilla SRAM has a cycle-time of 1.025ns with the read logic being the critical timing path. By estimating n-bit-hybrid circuit through bit-parallel circuit with n-bit precision, our analysis shows that n-bit-hybrid (with $n <= 8$) has a cycle-time equivalent to that of the baseline with no penalty. However, targeting 16-bit-hybrid incurs a cycle-time penalty of about 15% (cycle time of 1.175ns). 32-bit-hybrid configuration further increases the overhead to 51% (cycle-time of 1.55ns).

Vanilla SRAM supports two basic operations: read, and write. EVE SRAM supports read and write basic operations as well as extra operations (shown in Table 5.2). According to power analysis conducted on the extracted netlist from the layout, EVE SRAM energy for the basic operations (i.e., read and write) are similar to the vanilla SRAM. Other than bit-line compute operation (blc), the extra operations supported by EVE SRAMs incur much lower energy since no sense-amplifiers nor bit-line pre-charging is involved. Bit-line compute (blc) in EVE SRAM incur around 20% higher energy when compared to a read (the most energy-expensive operation in the vanilla SRAM). Despite the energy increase, EVE is energy-efficient since no energy-expensive data movements are needed through the H-tree.

5.7 Architecture Evaluation

This section discusses the performance methodology and evaluation of the different EVE designs.

			Scalar	
Name	**Suite**	**Input**	**DIns**	**IOc**
vvadd	k	8.388M	75.5M	205M
mmult	k	1024	8.60B	48.5B
k-means	ro	10Kx34	2.11B	3.59B
pathfinder	ro	5Mx10	1.08B	2.43B
jacobi-2d	rv	2Kx10	1.59B	8.50B
backprop	ro	524K	1.17B	15.7B
sw	g	2070	1.38B	1.69B

Table 5.4: Benchmark Applications List – k = kernel, ro = rodinia, rv = RiVEC, g = genomics, DIns = number of dynamic instructions in ROI, IOc = number of cycles to run on an in-order core.

5.7.1 Methodology

To model the performance of the different EVE designs, we leveraged gem5 [BBB+11,LPAA+20, TCB18a], which is a cycle-approximate simulator. We built cycle-approximate models for EVE-*n* and other baseline systems in gem5 and verified their correctness. In our modeling, we used two core types as the scalar baselines: out-of-order (O3) and in-order (IO). For the O3 core, we leveraged gem5's out-of-order core. As for IO core, we developed our own single-issue, in-order core. For the memory sub-system, we used ARM's CHI cache coherency model [gem21]. We modified ARM CHI to include special ports to connect vector units to either the L2 cache or the LLC, while modeling arbitration between the L1 and the vector unit.

To quantify the performance of EVE, we leveraged two scalar baselines and two vector baselines, detailed in Table 5.3. The two scalar baselines are in-order core (IO) and out-of-order core (O3). Since an EVE design incurs about 15% area overhead in the L2 cache, we evaluated the performance of an O3 system with 15% bigger L2 cache. An O3 system with 15% bigger L2 cache performs equivalently to an O3 without the increase in the L2 cache. The two vector baselines are: integrated vector unit added to an out-of-order core (O3+IV) (loosely based on [SBB+17a,RBGZ19]) and decoupled vector unit added to an out-of-order core (O3+DV) (loosely based on [EAE+02]). The development and design for the vector baseline models are discussed in Chapter 3. IV shares three execution pipes with the control core (two floating-point/SIMD pipes, and memory execution pipe) and shares the load-store queue. Constant strides and indexed memory operations are decomposed to micro-operations and handled as scalar load-

s/stores by the load-store queue. As for DV, it has four execution pipes: simple integer, pipelined complex integer, iterative complex integer/cross-element, and memory execution pipe. DV includes a detailed model of a vector memory unit (VMU) capable of performing unit-stride, strided, and indexed memory operations. The VMU uses its TLB port to translate addresses for each generated cacheline memory request. Our model accounts for the request generation and address translation with one cycle and it assumes translated addresses always hit in the TLB.

We built a flexible cycle-approximate model of EVE in gem5 to calculate the performance of the different EVE-*n* designs. The model contains four units: vector control unit (VCU), vector sequencing unit (VSU), vector reduction unit (VRU), and vector memory unit (VMU). The control processor sends a vector request to EVE upon encountering a vector instruction in commit stage. The VCU handles the incoming vector requests as well as the outgoing vector responses. The VCU forwards the requests in the order received to the VSU and the VMU. The VSU performs a micro-decoding and starts executing the micro-program for the vector request. Each generated µops from the VSU is sent to the EVE SRAMs in one cycle. The VSU stalls special read/write µops from/to the DTU until the data is ready. Our model performs a separation between execution and timing by using the µops to estimate timing while execution for the vector instructions happens functionally.

We evaluated the performance of the different EVE designs on applications form Rodinia [CBM+09] and RiVEC [RHP+20] benchmark suites (shown in Table 5.4). The applications were vectorized manually using vector intrinsics in LLVM 13. The characterization of the scalar and vector versions of the applications is detailed in Table 5.5.

5.7.2 Results

Figure 5.9 shows the simulated performance (normalized to IO) of the different EVE designs against other baselines. Table 5.6 details the performance of the different EVE designs normalized to different baselines. O3+DV achieves the best performance on the chosen data-parallel workloads, as expected. On average, O3+DV achieves 21.58× speedup over the in-order scalar baseline (IO). O3+IV achieves a speedup of 5.58× over IO and 1.74× speedup over out-of-order scalar baseline (O3). Among all the different EVE designs, EVE-8 achieves the best performance of 25.60× over the IO baseline, which is comparable to O3+DV performance. EVE-16 achieves the best next performance among EVE designs; however, EVE-16 suffers from a cycle-time penalty

	Vector (VL=64)														
Name	**DIns**	**VI%**	**ctrl**	**ialu**	**imul**	**xe**	**us**	**st**	**idx**	**prd**	**DOp**	**VO%**	**VPar**	**WInf**	**ArInt**
vvadd	1.6M	42%	20	20	0	0	7	0	0	0	35.5M	96%	22.6	0.47	0.33
mmult	151M	44%	25	25	25	0	25	0	0	0	3.35B	97%	22.2	0.39	2.00
k-means	51.5M	46%	1	52	18	∼0	∼0	10	7	1	1.53B	98%	29.8	0.72	2.44
pathfinder	22.5M	50%	31	37	0	0	16	0	0	25	513M	97%	22.8	0.48	1.20
jacobi-2d	35.4M	44%	8	50	8	17	7	0	0	0	940M	98%	26.6	0.59	4.50
backprop	21.5M	39%	13	19	25	∼0	5	12	0	0	488M	96%	22.7	0.42	1.00
sw	20.4M	39%	10	55	0	11	10	14	0	10	433M	97%	21.2	0.31	2.75

Table 5.5: Benchmark Applications Characterization – DIns = number of dynamic instructions in ROI, IOc = number of cycles to run on an in-order core, VI% = 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, xe = vector cross-element instructions, us = vector unit stride memory instructions, st = vector constant stride memory instructions, idx = vector indexed memory instructions, prd = predicated vector instructions, DOps = total number of operations (scalar instructions + vector instructions $\times$ active vector length), VO% = percent of operations performed by vector unit, VPar = logical parallelism (total ops / dynamic instructions in vectorized program), WInf = work inflation (total ops in vectorized program / dynamic instructions in scalar program), ArInt = arithmetic intensity (mathematical operations / memory operations) for vector unit.

that affects its scalar performance. As a result, EVE-8 is a very compelling design point incurring modest overhead, yet achieving performance comparable to a decoupled vector unit.

EVE incurs a modest area overhead making it area-comparable to O3+IV. The VMU employed by EVE is similar and sized equivalently to O3+IV's VMU. EVE requires only eight data transpose units (DTUs). Each DTU is equivalent in size to half a sub-array. The ROM required for to implement the macro-operations is equivalent to one sub-array. In total, EVE incurs a 7.8% increase in the number of sub-arrays in the L2 cache (which contains 64 sub-arrays). Considering EVE-8, the circuits incur 7.8% area overhead, but since only half the SRAMs are EVE SRAMs, the circuit overhead is 3.9%. Overall, EVE-8 incurs a total area overhead of 11.7%. EVE in its best configuration can achieve 4.59$\times$ performance speedups over the O3+IV. By achieving higher performance at a similar area overhead, EVE is more area-efficient than O3+IV.

Compute Throughput – Figure 5.10 shows the execution breakdown of different EVE designs running several applications. The execution breakdown, normalized to EVE-1, clearly shows the column under-utilization effect in S-CIM: *the percentage of time where EVE is busy executing useful work goes down as the parallelization factor increases until balanced utilization (i.e., EVE-4),*

	Speedup vs. O3+IV							E-8 vs.	
Name	**DV**	**E-1**	**E-2**	**E-4**	**E-8**	**E-16**	**E-32**	**E-1**	**E-32**
vvadd	3.64	3.19	3.23	3.24	3.28	3.23	3.38	1.03	0.97
mmult	4.42	0.93	1.84	3.55	5.34	5.29	4.60	5.71	1.16
k-means	2.28	1.22	1.32	1.35	1.86	1.82	1.51	1.53	1.24
pathfinder	8.11	5.37	6.27	6.33	6.30	6.30	6.20	1.17	1.02
jacobi-2d	6.36	6.18	9.50	11.30	13.49	13.50	12.69	2.18	1.06
backprop	2.14	2.01	2.05	2.07	2.07	2.06	2.06	1.03	1.01
sw	3.44	2.43	3.97	5.32	6.21	6.14	5.08	2.55	1.22
geomean	3.87	2.88	3.64	4.03	4.59	4.55	4.16	1.59	1.10

Table 5.6: Benchmark Applications Results – DV = O3+DV, E-x = EVE-x, geomean = calculated for: {k-means, pathfinder, jacobi-2d, backprop, sw}.

then it starts to increase due to row under-utilization and slower clock. The reason behind this behavior is: despite the hardware vector length being constant, the latency of the in-situ ALUs decreases going from EVE-1 to EVE-4 (with EVE-4 being balanced utilization), giving EVE higher compute throughput and thus requiring less time to perform the same amount of work; beyond EVE-4, row under-utilization coupled with slower clock (for EVE-16, and EVE-32 only) causes lower compute throughput and thus EVE requires more time to perform the same amount of work.

For applications that are memory-bound (i.e., *vvadd*, *pathfinder*, *backprop-int*), the execution time for the different parallelization factors is dominated by memory stalls. Therefore, compute throughput have a negligible impact on the overall performance. *vvadd* is inherently memory bound, while the others (as will be shown later) are faced with hardware limitations from the memory sub-system.

Transpose/Detranspose Overhead – Although EVE employs a very conservative transpose units, the transpose/detranspose overhead is minimal in most applications, as shown in Figure 5.10. One of the applications that experience measurable overhead from transpose/detranspose is *vvadd*, which is heavily memory bound. In *vvadd*, the bandwidth of transpose/detranspose in EVE fails to match the bandwidth of the data streaming from the cache sub-system. As a result, the transpose units start to induce negligible stalls. *pathfinder* is a the only application that show significant transpose/detranspose overheads. EVE struggles to overlap compute with transpose/detranspose and as a result, the execution stalls waiting for the data.

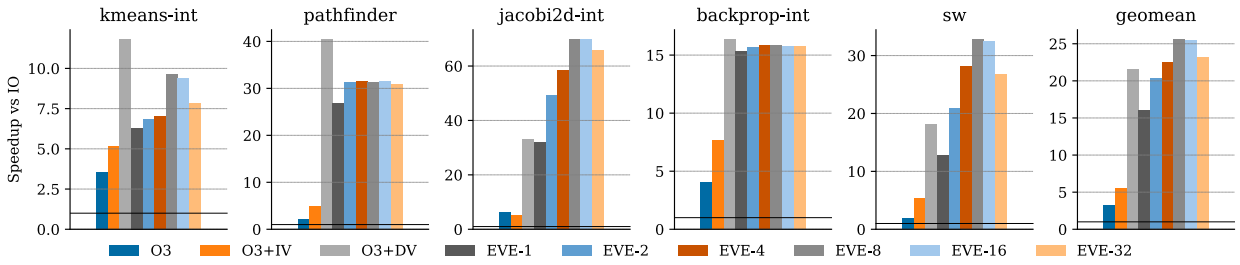


Figure 5.9: Performance Evaluation – performance of the different EVE configuration on applications from Rodinia and RiVEC benchmark suite normalized to the performance of an in-order core.

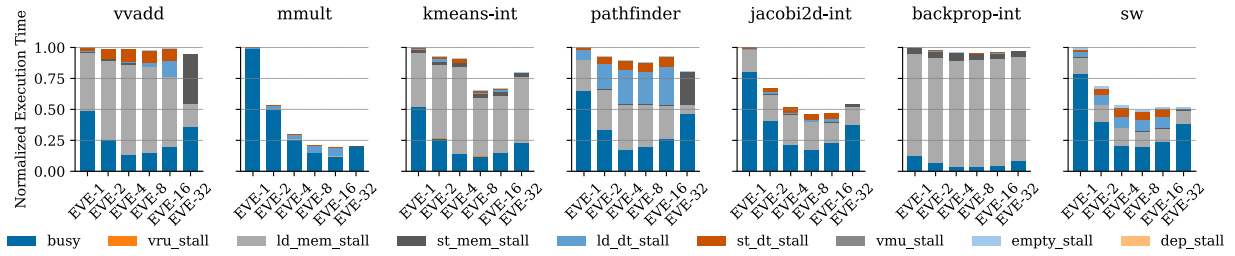


Figure 5.10: Execution Breakdown – profiling execution breakdown between different categories (normalized to EVE-1 execution time); busy=executing useful work; vru_stall=VRU structural hazard stall; ld_mem_stall=load memory stall; st_mem_stall=store memory stall; ld_dt_stall=load transposing stall; st_dt_stall=store detransposing stall; vmu_stall=VMU structural hazard stall; empty_stall=empty cycle stall; dep_stall=register dependency stall.

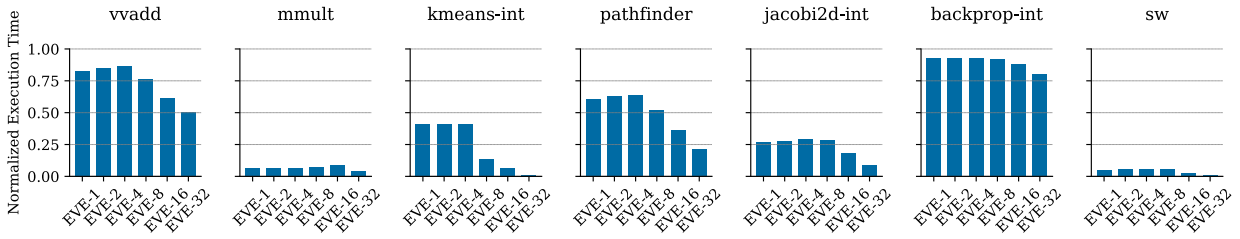


Figure 5.11: Cache-Induced Stalls in the VMU – showing the percentage of time the VMU is faced with a stall when sending a request to the LLC (normalized to EVE-1 execution time). These stalls do not necessarily cause a bubble/stall in execution as they can be hidden by overlapping outstanding compute in EVE.

In EVE-1, the transpose/detranspose overhead is insignificant regardless of the application. Due to its low compute throughput, EVE-1 is able to overlap execution with the transpose/detranspose operation. This effect is more prominent in *mmult*, which is compute-bound. As the compute throughput increases starting from EVE-1 till EVE-4, the transpose/detranspose overhead increases while memory stalls are non-existent; EVE-32, however, requires no transpose/detranspose and, therefore, has higher performance despite the increase in its compute time. For *pathfinder* (which has a significant transpose/detranspose overhead), EVE-32 achieves higher performance despite its lowered compute throughput, also. In *sw*, the lack of transpose/detranspose operation for EVE-32 allows it to match EVE-16 performance while still having lower compute throughput and frequency.

Limited MSHR Effect – For applications dominated by memory stalls, as explained before, some are inherently memory-bound, while others are limited by the number of available MSHRs in the LLC. Figure 5.11 shows the percentage of execution time in which the VMU experiences a stall in issuing a cache request. Execution breakdown for *kmeans-int* indicates a significant amount of memory stalls; these stalls are explained by Figure 5.11 as it shows that the VMU experiences cache-induced stalls for almost half of the execution time between EVE-1 and EVE-4. In EVE-8, however, due to row under-utilization, the hardware vector length is halved and, thus, the required number of MSHRs is also halved. Indeed Figure 5.11 shows that the cache-induced stalls are reduced by more than half. As the hardware vector length is halved further for EVE-16 and EVE-32, the cache request stalls in the VMU keep on decreasing further. Although similar behavior is observed in *pathfinder* enabling EVE-8 in achieving the highest performance, the transpose/detranspose overhead is a bottleneck in achieving higher performance. Finally, *backprop-int* performs strided-memory operations with a very large stride. As a result, no two elements in these operations would reside in the same cacheline, and thus this application requires significantly more MSHRs than available. This is further shown in Figure 5.11 where the VMU experiences cache-induced stalls for more than 90% of the time while executing *backprop-int*. While these stalls are lowered as the parallelization factor increases and the hardware vector length is halved, the decrease in these stalls is very minimal and indicates a significant limitation on the number of available MSHRs.

Energy/Power Analysis – According our evaluation of EVE circuits, the peak power consumption of the SRAM arrays is expected to increase by 20% . Although this figure might look

concerning, performing vector operations in EVE requires a mixture of multiple μops, and as such the expected power consumption overhead is lower than 20% . Previous work [FMD19,EWW+18, AJS+17] have shown the energy efficiency of S-CIM execution, and more recent work [AHAA+20] have shown that different execution paradigms have comparable energy efficiency. EVE leverages these energy efficient S-CIM execution techniques with bit-hybrid execution. Moreover, vector execution in EVE is energy-efficient as it eliminates the need for highly multi-ported vector register files that incur high access energy. Also, since the compute is fused with storage, there is no need for expensive redundant data movement from private L2 caches (through the costly H-tree) to these highly multi-ported vector register files, only to be read again to the functional units.

Area Efficiency Analysis – The best design point of EVE achieves comparable performance to O3+DV while incurring an area overhead equivalent to O3+IV. Compared to O3 core, the O3+IV baseline area is estimated to be 1.10× , while the O3+DV baseline area is estimated to be 2.00× . As for EVE design points, EVE-1 has an area estimate of 1.10× , EVE-2 through EVE-16 has an area overhead of 1.12× , and EVE-32 has an area overhead of 1.11× . The best EVE best design point, EVE-8, is able to increase performance by 4.59× at comparable area overhead to the O3+IV. EVE-8 also achieves over twice the area-normalized performance compared to O3+DV. EVE-8 is able to attain higher area-normalized performance than O3+DV by achieving comparable performance at a much lower area-overhead.

5.8 Related Work

Conventional vector processing architectures use dedicated long-vector engines to accelerate data-parallel computation [DVWW05, EAE+02, ABS+07, TNH+06]. These machines achieve incredibly high performance on data parallel workloads at the expense of a significant cost in area. Subword packed SIMD is an attempt to reap some of the same benefits as conventional vector processing without paying the same area overhead. These designs re-use many of the scalar components of a processor to perform vector operations [PW96,int07,SGC+16]. However, their performance is limited by the lack of available hardware units and limited instruction sets. Recently, next-generation scalable vector length ISA extensions which are flexible enough to support both of the traditional hardware paradigms have been developed [CSZ+19, CXL+20, Sat20, SBB+17a, RISC-

V Foundation19]. These vector extensions also have the flexibility to support new developments in the vector processing space, like EVE.

Traditional processing-in-memory (PIM) techniques have primarily involved implementing computation logic in DRAM chips [PAC+97b, ESS+99, OCS98]. The goal of these implementations is to address the memory bandwidth wall by reducing the amount of data transfer across the system memory bus. Some recent variations have leveraged 3D manufacturing techniques to reduce the amount of change required to the physical design of the memory circuits [FFAMK15, AHY+15]. These designs aim to address the fact that modifying DRAM chips is a difficult proposition due to the incredibly high circuit density and desire for wide compatibility by implementing the compute circuitry on separate chips that can be assembled with 3D manufacturing techniques.

While traditional processing-in-memory (PIM) focuses on bringing compute closer to data, in-situ processing proposes to reconfigure the memory into compute engines achieving higher efficiency and performance. Rowclone [SKF+13] is among the earliest work to explore transforming DRAM into a compute engine capable of in-DRAM row cloning. Ambit [SLM+17] and DRISA [LNM+17] proposed transforming the DRAM into an accelerator capable of massive bit-wise logical operations between multiple rows. ComputeDRAM [GTW19] explores implementing this technique with entirely stock DRAM chips.

While the work on in-situ processing-in-DRAM seems promising and can be explored beyond rudimentary bit-wise logical operations, special technology considerations and reduced margins-of-error make DRAM a difficult choice. Instead, SRAM is a more promising venue to explore in-situ processing. Jeloka et. al. [JASB16, JASB15] introduces the bit-line compute technique, which performs digital bit-wise logical operations between SRAM rows. Bit-line compute constitutes an important precursor to subsequent processing-in-SRAM work. Compute caches [AJS+17] uses bit-line compute to transform the caches in a chip multi-processor into bit-wise logical compute engines.

Further work based on bit-line compute explored extending its functionality by utilizing a bit-serial approach to complex integer and floating-point operations. This work has demonstrated the ability to use bit-line compute to transform caches in a CMP into fixed-function accelerators for neural networks [EWW+18] and single-instruction-multiple-threads (SIMT) engine [FMD19]. To mitigate transposing data, Wang et. al. [WWE+19] explored adding bit-line compute to an 8T-SRAM allowing the computation to be performed horizontally in the compute-bitline (CBL),

while data read and writes are performed on the vertical bitlines. The 8T bitcell hinders its use in traditional caches due to low density. EVE leverages traditional 6T-SRAM, thus retaining high density and can be used in traditional caches. VRAM [AHAA+20] explored utilizing bit-serial and bit-parallel execution paradigms to extend the functionality of bit-line compute. As bit-serial execution achieves high throughput but high latency and bit-parallel execution achieves lower latency at the expense of lower throughput [AHAA+20], EVE proposes bit-hybrid execution to balance throughput and latency.

Another line of work [GGP+13,MYKG16,YMG15,ZL20,YEK18,CYS+21] on in-situ compute-in-memory explores leveraging associative compute abstraction [Fos76,Say76,Say94]. Some work in this space proposes utilizing analog compute on emerging resistive technology [GGP+13,ZL20, YEK18] to perform the computations. However, analog compute accuracy suffers due to process variation. Other work utilizes CAM search/multi-write [CYS+21, MYKG16, YMG15] to perform the computations, incurring extremely long latencies for the operations.

5.9 Conclusion

This chapter have demonstrated the ability of EVE to resolve the tension between performance and area in next-generation vector architectures. By employing S-CIM techniques, EVE is able to reconfigure private L2 caches into ephemeral vector engines with comparable performance to decoupled vector engines. EVE addresses the S-CIM programming challenge by adopting next-generation vector architecture. For the S-CIM serialization latency challenge, EVE mitigates the serialization overhead by proposing a novel bit-hybrid execution mechanism facilitating lower compute and memory latency. EVE serves as a motivation for future research in leveraging S-CIM techniques to alleviate the area-overhead associated with next-generation vector architectures.

Future research can explore techniques to further increase the utilization of the S-CIM arrays (e.g., dynamic micro-operation scheduling) with the help of an out-of-order core. Since EVE is an example of a new breed of very long next-generation vector machines, another line of research is to address the limited MSHRs efficiently to enable EVE to utilize memory bandwidth more effectively. EVE have shown the ability of bit-hybrid execution paradigm in balancing latency and throughput with focus on integer operations, future research can explore using bit-hybrid execution to balance latency and throughput for floating-point operations.

CHAPTER 6
CONCLUSION

As specialization is becoming a driving factor for regular improvements and efficiency, there has been a recent resurgence of next-generation vector architectures, such as ARM Scalable Vector Extension (SVE) [Ste16,SBB+17b], RISC-V Vector extension (RVV) [RISC-V Foundation19], and Intel Advanced Vector Extension (AVX-512) [int12]). There are two conventional implementations for next-generation vector micro-architectures: an integrated vector unit, and decoupled vector engine. There is a fundamental tension between these two micro-architectures: An integrated vector unit achieves modest performance while having low area overhead; however, a decoupled vector engine achieves higher performance at the cost of significantly higher area overhead. Recently, SRAM-based compute-in-memory (S-CIM) have shown a lot of promise in addressing the tension between these two micro-architectures. Previous work on S-CIM have either: (1) exclusively explored bit-serial or bit-parallel execution paradigm; and/or (2) lacked a fitting programming abstraction. This thesis shows that bit-serial execution achieves high throughput at the expense of high latency, while bit-parallel execution achieves lower latency at the cost of lower throughput. The thesis proposes the novel approach of bit-hybrid execution to balance throughput and latency. Moreover, the thesis leverages a circuits-to-architecture exploration to evaluate bit-hybrid execution thoroughly. The thesis presents a circuit template that can target any bit-hybrid design point. This thesis makes a case for bit-hybrid execution paradigm for S-CIM to balance latency and throughput. The rigorous analysis and results in this thesis shows the potential and promise of bit-hybrid execution paradigm for further research. This thesis, also, introduces a new breed of vector micro-architectures capable of supporting hardware vector length of thousands of elements, thus presenting a new challenge for vector abstraction. The rest of this chapter summarizes primary contribution of this thesis and discusses future research directions.

6.1 Thesis Summary and Contribution

This thesis began by motivating the need to address the tension between two next-generation vector micro-architectures: integrated vector units and decoupled vector engines. With the slowdown of Moore's Law and the end of Dennard's Scaling, there is a need for specialization to retain regular improvements in performance and efficiency. In this thesis, I present a background on

vector architectures and highlight the rise of next-generation vector architecture. Then, I explain the tension between the two extreme points of next-generation vector micro-architectures: while integrated vector unit achieves modest performance while incurring low area overhead, decoupled vector engine achieves higher performance at significantly higher area overhead. Then, I present a background on compute-in-memory and highlight it as means to address the previously mentioned tension.

The thesis then discusses AgileBNN–an early attempt to address the need for specialization in a post Moore's Law and post Dennard's Scaling era. AgileBNN is a fixed-function accelerator for binaried neural network (BNN) developed and synthesized using a novel high-level synthesis (HLS) flow. AgileBNN was part of the Celerity SoC, a 5×5 mm chip implementing the tiered accelerator fabric (TAF) architecture. A tiered accelerator fabric includes three tiers: (1) general-purpose tier, (2) massively-parallel tier, and (3) specialization tier. AgileBNN formed the specialization tier in the Celerity SoC. With co-operative execution, AgileBNN was able to increase its area normalized performance and achieving $2\times$ higher area normalized when compared to state-of-the-area accelerators. AgileBNN inspired two main themes in thesis: the rigidity of AgileBNN's fixed-function aspect inspired the adoption of the more programmable specialized vector abstraction; and (2) the ability of AgileBNN to reconfigure other components to increase its performance and reduce its effective area foot-print inspired the pursuit of ephemerality.

The core of the thesis is a circuits-to-architecture exploration of the ephemeral vector engines (EVE). Starting with the observation that previous work on S-CIM explored building vector execution units leveraging bit-serial execution, the exploration of EVE starts with a circuits investigation to build a S-CIM vector execution unit using bit-parallel execution paradigm. To the best of my knowledge, this was the first work to explore an execution paradigm other than bit-serial. There were two main outcomes of this investigation: (1) bit-serial and bit-parallel execution incur similar and comparable area overhead; and (2) bit-serial execution achieves high throughput at the expense of high latency, while bit-parallel execution achieve low latency at the cost of low throughout. This work motivated the later pursuit of a more balanced execution paradigm in-between bit-serial and bit-parallel. To address this motivation, this thesis proposes bit-hybrid execution paradigm to balance latency and throughput. By breaking elements into segments, the circuit of bit-hybrid-capable S-CIM vector unit operates on the bits of a segment in parallel, while the segments themselves are processed serially. To the best of my knowledge, this is the first work that motivates and proposes

bit-hybrid execution to balance latency and throughput. To perform an architectural exploration, this thesis discusses modeling two next-generation vector micro-architectures: integrated vector unit and decoupled vector engine. Using a circuits-to-architecture evaluation methodology, this thesis shows that EVE is able to leverage a bit-hybrid execution paradigm to reconfigure on-the-fly private level-2 caches of a core to construct a vector execution engine. EVE is able to balance latency and throughput to achieve vector performance comparable to an aggressive decoupled vector engine while incurring an area overhead equivalent to that of integrated vector unit.

To reiterate the major contribution of this thesis:

- I presented highly configurable cycle-approximate models for a decoupled vector engine and an integrated vector unit, which serve as a tool to enable future research on next-generation vector architectures.
- I proposed a novel template for ephemeral vector engines (EVE) circuits that can, at design time, be configured to target: bit-serial execution, bit-parallel execution, and any parallelization factor in-between.
- I proposed ephemeral vector engines (EVE), a novel next-generation vector architecture that leverages SRAM-based compute-in-memory circuits to transform L2 caches on-the-fly into vector execution engines. I also presented a holistic exploration from circuits to architecture of the EVE designs with different parallelization factors.
- To my knowledge, I am the first to make the case for bit-hybrid execution paradigm in SRAM-based compute-in-memory to balance throughput and latency.

6.2 Future Work

This thesis has showcased how EVE leverages bit-hybrid execution to balance latency and throughput, thus achieving performance comparable to an aggressive decoupled vector engine while incurring an equivalent area overhead to a lightweight integrated vector unit. This section discusses some promising research directions for future work.

Floating-Point Support – The thesis focused on integer support for EVE showing promising results. The thesis was able to explore the impact of bit-hybrid execution on integer arithmetic as well as memory operations. Previous work on S-CIM have already explored floating-point support

for bit-serial execution [FMD19]. Future work can explore floating-point support for bit-hybrid execution. As EVE have shown that there is some trade-offs between segment size and throughput/latency for integer and memory operations, future work can explore the impact of segment size on floating-point operations.

MSHR Bottleneck in Super-Long-Vector Engines – This thesis introduces EVE as a new breed of super-long-vector machines with hardware vector length of thousands of elements. In the architectural evaluation of EVE, it was shown that MSHRs are a bottleneck for super-long-vector machines due to their limited amount. When a miss occurs, the cache controller sends a memory request for a cacheline and records all needed meta-data in a data structure called miss status handling register (MSHR). This meta-data is required to track on-going misses as well as handling responses from memory that arrives out-of-order. These MSHRs are designed to handle memory requests on cacheline granularity. While fitting for scalar compute, multiple MSHRs are required for a single vector memory operation. This problem is further exacerbated super-long-vector machines, as shown in this thesis. Future work can explore mitigating the limitation of MSHRs for super-long-vector machines. The idea of creating *super MSHR* can alleviate this bottleneck by enabling a single MSHR to represent more than one cacheline. For a secondary-miss search, these *super MSHR* need to be treated differently than regular MSHRs. Addressing this bottleneck can further increase the performance of EVE unlocking its potential.

Sparsity in EVE – While this thesis have shown a lot of promise in EVE, S-CIM vector unit struggle with sparsity. Most S-CIM vector units leverage their super-long hardware vector lengths to mitigate their inherent longer latencies, thus the break-even point for these units is high. As a result, executing sparse workloads on these S-CIM vector units is quite challenging. Although EVE is able to balance latency and throughput, EVE still has a higher break-even point when compared to a traditional aggressive decoupled vector unit. Future work on circuits-to-architecture exploration can address these challenges. Future work can inspect additional hardware to increase EVE performance with sparsity. In addition, future work can explore additional units and different micro-architectural solutions to increase performance for sparse workloads and sparse datasets. Moreover, future work can explore additional instructions and suggestions to enhance vector abstraction for sparse workloads.

BIBLIOGRAPHY

[AAHA+17] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Lotfi, J. Puscar, A. Rao, A. Rovinski, L. Salem, N. Sun, C. Torng, L. Vega, B. Veluri, X. Wang, S. Xie, C. Zhao, R. Zhao, C. Batten, R. G. Dreslinski, I. Galton, R. K. Gupta, P. P. Mercier, M. Srivastava, M. B. Taylor, and Z. Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. *Symp. on High Performance Chips (Hot Chips)*, Aug 2017.

[ABS+07] D. Abts, A. Bataineh, S. Scott, G. Faanes, J. Schwarzmeier, E. Lundberg, T. Johnson, M. Bye, and G. Schwoerer. The Cray BlackWidow: A Highly Scalable Vector Multiprocessor. *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Nov 2007.

[AHAA+20] K. Al-Hawaj, O. Afuye, S. Agwa, A. Apsel, and C. Batten. Towards a Reconfigurable Bit-Serial/Bit-Parallel Vector Accelerator using In-Situ Processing-In-SRAM. *Int'l Conf. on Circuits and Systems (ISCAS)*, pages 1–5, Oct 2020.

[AHY+15] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[AJLR18] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy. X-SRAM: Enabling In-Memory Boolean Computations in CMOS Static Random Access Memories. *IEEE Trans. on Circuits and Systems I*, Jul 2018.

[AJR+19] A. Agrawal, A. Jaiswal, D. Roy, B. Han, G. Srinivasan, A. Ankit, and K. Roy. Xcel-RAM: Accelerating Binary Neural Networks in High-Throughput SRAM Compute Arrays. *IEEE Trans. on Circuits and Systems I*, Apr 2019.

[AJS+17] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute Caches. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2017.

[arm09] Introducing NEON: Development Article. Online Webpage, 2009 (accessed Jun, 2009). `https://documentation-service.arm.com/static/5f1071760daa596235e8320a`.

[AYMC15] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[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.

[BC18] A. Biswas and A. P. Chandrakasan. Conv-RAM: An Energy-Efficient SRAM with Embedded Convolution Computation for Low-Power CNN-Based Machine Learning Applications. *Int'l Solid-State Circuits Conf. (ISSCC)*, pages 488–490, 2018.

[BDM+72] W. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. The Illiac IV System. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[BTKK04] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge. A Low Cost, Multithreaded Processing-in-Memory System. *Proc. of Workshop on Memory Performance Issues*, Jun 2004.

[BYM+19] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann. An Always-On 3.8μJ/86% CIFAR-10 Mixed-Signal Binary CNN Processor With All Memory on Chip in 28-nm CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 54(1):158–172, Jan 2019.

[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.

[CDGT17] T. M. Conte, E. P. DeBenedictis, P. A. Garhini, and E. Track. Rebooting Computing: The Road Ahead. *IEEE Computer*, 50(1):20–29, Jan 2017.

[CHS+16] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv*, arXiv:1602.02830, Feb 2016.

[CLL+18] W.-H. Chen, K.-X. Li, W.-Y. Lin, K.-H. Hsu, P.-Y. Li, C.-H. Yang, C.-X. Xue, E.-Y. Yang, Y.-K. Chen, Y.-S. Chang, T.-H. Hsu, Y.-C. King, C.-J. Lin, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, and M.-F. Chang. A 65nm 1Mb Nonvolatile Computing-in-Memory ReRAM Macro with Sub-16ns Multiply-and-Accumulate for Binary DNN AI Edge Processors. *Int'l Solid-State Circuits Conf. (ISSCC)*, pages 494–496, 2018.

[CLX+16] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[CRS+20] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan. GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, May/Jun 2020.

[CSZ+19] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating-Point Support in 22nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[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)*, 2020.

[CXZ+17] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang. TIME: A training-in-memory architecture for memristor-based deep neural networks. *Design Automation Conf. (DAC)*, Jun 2017.

[CYS+21] H. Caminal, K. Yang, S. Srinivasa, A. K. Ramanathan, K. Al-Hawaj, T. Wu, V. Narayanan, C. Batten, and J. F. Martínez. CAPE: A Content-Addressable Processing Engine. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 557–569, 2021.

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

[ea18] R. A. et al. YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 48–60, Jan 2018.

[ea19a] M. R. el al. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. Online Webpage, 2016 (accessed Oct 20, 2019).

[ea19b] A. R. et al. A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS. *Symp. on Very Large-Scale Integration Circuits (VLSIC)*, pages 30–31, Dec 2019.

[ea19c] A. R. et al. Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL. *IEEE Solid-State Circuits Letters (SSCL)*, pages 289–292, Dec 2019.

[ea19d] K. A. et al. The Rocket Chip Generator. Online Webpage, 2016 (accessed Oct 20, 2019).

[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.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[ESS+99] D. Elliot, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie. Computational RAM: Implementing Processors in Memory. *IEEE Design and Test of Computers*, Jan-Mar 1999.

[EWW+18] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer†, D. Sylvester, D. Blaauw, and R. Das. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. *Int'l Symp. on Computer Architecture (ISCA)*, Jul 2018.

[FFAMK15] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2015.

[FMD19] D. Fujiki, S. Mahlke, and R. Das. Duality Cache for Data Parallel Acceleration. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2019.

[Fos76] C. C. Foster, editor. *Content Addressable Parallel Processors*. Van Nostrand Reinhold, 1976.

[FRF+03] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the FlexRAM Parallel Intelligent Memory System. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Jun 2003.

[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/.

[GGP+13] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman. AC-DIMM: Associative Computing with STT-MRAM. *Int'l Symp. on Computer Architecture (ISCA)*, 41(3):189–200, 2013.

[GSA+16] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar. OpenRAM: An Open-Source Memory Compiler. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Jan 2016.

[GTW19] F. Gao, G. Tziantzioulis, and D. Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Sdelf DRAMs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2019.

[GWPK04] J. Gebis, S. Williams, D. Patterson, and C. Kozyrakis. VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM. *Design Automation Conf. (DAC)*, Jun 2004.

[HKK+99] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture. *Int'l Symp. on Supercomputing (ICS)*, pages 57–75, 1999.

[IGKR19] M. Imani, S. Gupta, Y. Kim, and T. Rosing. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2019.

[IKT+98] M. Ishida, T. Kawakami, A. Tsuji, N. Kawamoto, M. Motoyoshi, and N. Ouchi. A Novel 6T-SRAM Cell Technology Designed with Rectangular Patterns Scalable Beyond 0.18/spl mu/m Generation and Desirable for Ultra High Speed Operation. *International Electron Devices Meeting*, Dec 1998.

[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.

[JASB15] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. A Configurable TCAM/BCAM/SRAM Using 28nm Push-Rule 6T Bit Cell. *Symp. on Very Large-Scale Integration Circuits (VLSIC)*, Jun 2015.

[JASB16] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit-Cell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits (JSSC)*, Apr 2016.

[JKK11] M. Jang, K. Kim, and K. Kim. The Performance Analysis of ARM NEON Technology for Mobile Platforms. *Symp. on Research in Applied Computation*, pages 104–106, 2011.

[JSPR08] T. Jhaveri, A. Strojwas, L. Pileggi, and V. Rovner. Enabling Technology Scaling with "In Production" Lithography Processes. *Optical Microlithography XXI*, Feb 2008.

[KCL+18] W.-S. Khwa, J.-J. Chen, J.-F. Li, X. Si, E.-Y. Yang, X. Sun, R. Liu, P.-Y. Chen, Q. Li, S. Yu, and M.-F. Chang. A 65nm 4Kb Algorithm-Dependent Computing-in-Memory SRAM Unit-Macro with 2.3ns and 55.8TOPS/W Fully Parallel Product-Sum Operation for Binary DNN Edge Processors. *Int'l Solid-State Circuits Conf. (ISSCC)*, pages 496–498, 2018.

[KHY+99] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. *Int'l Conf. on Computer Design (ICCD)*, pages 192–201, 1999.

[KKC+16] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. *Int'l Symp. on Computer Architecture (ISCA)*, Aug 2016.

[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.

[LDL+18] G. Li, G. Dai, S. Li, Y. Wang, and Y. Xie. GraphIA: an In-Situ Accelerator for Large-Scale Graph Processing. *Int'l Symp. on Memory Systems*, Oct 2018.

[LHS+21] E. Lee, T. Han, D. Seo, G. Shin, J. Kim, S. Kim, S. Jeong, J. Rhe, J. Park, J. H. Ko, and Y. Lee. A Charge-Domain Scalable-Weight In-Memory Computing Macro With Dual-SRAM Architecture for Precision-Scalable DNN Accelerators. *Int'l Conf. on Circuits and Systems (ISCAS)*, 68(8):3305–3316, 2021.

[LNM+17] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *2017 IEEE/ACM 50th International Symposium on Microarchitecture MICRO, Boston, MA, USA*, pages 288–301, Oct 2017.

[LPAA+20] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, et al. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[LXK21] C. Liu, Z. Xuan, and Y. Kang. A 40-nm 202.3nJ/Classification Neuromorphic Architecture Employing In-SRAM Charge-Domain Compute. *Int'l Conf. on ASIC*, pages 1–4, 2021.

[LXZ+20] W. Li, P. Xu, y. Zhao, H. Li, Y. Xie, and Y. Lin. Timely: Pushing Data Movements And Interfaces In Pim Accelerators Towards Local And In Time Domain. *Int'l Symp. on Computer Architecture (ISCA)*, May/Jun 2020.

[LZB14] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[MBY+18] B. Moons, D. Bankman, L. Yang, B. Murmann, and M. Verhelst. BinarEye: An always-on energy-accuracy-scalable binary CNN processor with all memory on chip in 28nm CMOS. *Custom Integrated Circuits Conf. (CICC)*, pages 1–4, Dec 2018.

[McK04] S. A. McKee. Reflections on the Memory Wall. *Association for Computing Machinery*, page 162, Apr 2004.

[MYKG16] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar. Resistive GP-SIMD Processing-In-Memory. *ACM Trans. on Architecture and Code Optimization (TACO)*, 12(4):1–22, 2016.

[OCS98] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 1998.

[PAB+97] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick. Intelligent RAM (IRAM): The Industrial Setting, Applications, and Architectures. *Int'l Conf. on Computer Design (ICCD)*, pages 2–7, 1997.

[PAC+97a] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, pages 224–225, 1997.

[PAC+97b] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, Mar 1997.

[PAC+97c] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM (IRAM): Chips That Remember and Compute. *Int'l Solid-State Circuits Conf. (ISSCC)*, pages 224–225, 1997.

[PJZ+14] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Jun 2014.

[PW96] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *Int'l Symp. on Microarchitecture (MICRO)*, Aug 1996.

[QJZ+20] K. Qiu, N. Jao, M. Zhao, C. S. Mishra, G. Gudukbay, S. Jose, J. Sampson, M. T. Kandemir, and V. Narayanan. ResiRCA: A Resilient Energy Harvesting ReRAM Crossbar-Based Accelerator for Intelligent Embedded Processors. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2020.

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

[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):30, Nov 2020.

[RISC-V Foundation19] RISC-V Foundation. RISC-V "V" Vector Extension. *https://github.com/riscv/riscv-v-spec/releases/download/0.7.1/riscv-v-spec-0.7.1.pdf*, Jun 2019.

[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. *19th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020.

[Say76] G. E. Sayre. Staran: An Associative Approach to Multiprocessor Architecture. *Computer Architecture*, 4(2):199–221, 1976.

[Say94] G. E. Sayre. ASC: An Associative-Computing Paradigm. *Computer*, 27(11):19–25, 1994.

[SBB+17a] 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.

[SBB+17b] 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*, Mar 2017.

[Sch87] P. B. Schneck. *The CDC STAR-100*. Springer US, 1987.

[SEN+21] M. E. Sinangil, B. Erbagci, R. Naous, K. Akarvardar, D. Sun, W.-S. Khwa, H.-J. Liao, Y. Wang, and J. Chang. A Charge-Domain Scalable-Weight In-Memory Computing Macro With Dual-SRAM Architecture for Precision-Scalable DNN Accelerators. *IEEE Journal of Solid-State Circuits (JSSC)*, 56(1):188–198, 2021.

[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.

[SKF+13] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2013.

[SLM+17] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *2017 IEEE/ACM 50th International Symposium on Microarchitecture MICRO, Boston, MA, USA*, pages 273–287, Oct 2017.

[SQLC17] L. Song, X. Qian, H. Li, and Y. Chen. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2017.

[SQR+20] W. A. Simon, Y. M. Qureshi, M. Rios, A. Levisse, M. Zapater, and D. Atienza. BLADE: An in-Cache Computing Architecture for Edge Devices. *IEEE Trans. on Computers (TOC)*, Feb 2020.

[Ste16] N. Stephens. ARMv8-A Next-Generation Vector Architecture for HPC. *Symp. on High Performance Chips (Hot Chips)*, Aug 2016.

[STH+21] X. Si, Y.-N. Tu, W.-H. Huang, J.-W. Su, P.-J. Lu, J.-H. Wang, T.-W. Liu, S.-Y. Wu, R. Liu, Y.-C. Chou, Y.-L. Chung, W. Shih, C.-C. Lo, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, N.-C. Lien, W.-C. Shih, Y. He, Q. Li, and M.-F. Chang. A Local Computing Cell and 6T SRAM-Based Computing-in-Memory Macro With 8-b MAC Operation for Edge AI Chips. *IEEE Journal of Solid-State Circuits (JSSC)*, Apr 2021.

[SZQ+18] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. GraphR: Accelerating Graph Processing Using ReRAM. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2018.

[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)*, 2022.

[TCB18a] T. Ta, L. Cheng, and C. Batten. Simulating Multi-Core RISC-V Systems in gem5. *Workshop on Computer Architecture Research with RISC-V*, 2018.

[TCB18b] T. Ta, L. Cheng, and C. Batten. Simulating Multi-Core RISC-V Systems in gem5. *Workshop on Computer Architecture Research with RISC-V (CARRV)*, Jun 2018.

[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.

[Wat72] W. J. Watson. The TI ASC: A Highly Modular and Flexible Super Computer Architecture. *AFIPS: Fall Joint Computer Conference*, pages 221–228, 1972.

[Wil95] M. V. Wilkes. The Memory Wall and the CMOS End-Point. *Association for Computing Machinery*, 23(4):4–6, Sep 1995.

[WM95] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Association for Computing Machinery*, 23(1):20–24, Mar 1995.

[WWE+19] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester. A Compute SRAM with Bit-Serial Integer/Floating-Point Operations for Programmable In-Memory Vector Acceleration. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2019.

[YEK18] H. E. Yantır, A. M. Eltawil, and F. J. Kurdahi. Low-Power Resistive Associative Processor Implementation Through the Multi-Compare. *Int'l Conf. on Electronics, Circuits, and Systems (ICECS)*, pages 165–168, 2018.

[YMG15] L. Yavits, A. Morad, and R. Ginosar. Computer Architecture with Associative Processor Replacing Last-Level Cache and SIMD Accelerator. *IEEE Trans. on Computers (TOC)*, 64(2):368–381, 2015.

[ZAS+14] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti. A 3D-Stacked Logic-in-Memory Accelerator for Application-Specific Data Intensive Computing. *Int'l 3D Systems Integration Conf.*, Jan 2014.

[ZJL+14] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. *Proc. of Int'l Symp. on High-Performance Parallel and Distributed Computing*, Jun 2014.

[ZL20] Y. Zha and J. Li. Hyper-Ap: Enhancing Associative Processing Through A Full-Stack Optimization. *Int'l Symp. on Computer Architecture (ISCA)*, pages 846–859, 2020.

[ZLW+22] J. Zhang, Z. Lin, X. Wu, Z. Tong, C. Peng, W. Lu, Q. Zhao, H. Wu, and J. Chen. In-Memory Multibit Multiplication Based on Bitline Shifting. *IEEE Trans. on Circuits and Systems II*, 69(2):354–358, 2022.

[ZSZ+17] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2017.

[ZWV17] J. Zhang, Z. Wang, and N. Verma. In-Memory Computation of a Machine-Learning Classifier in a Standard 6T SRAM Array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):915–924, 2017.

[ZXY+17] Y. Zhang, L. Xu, K. Yang, Q. Dong, S. Jeloka, D. Blaauw, and D. Sylveste. Recryptor: A Reconfigurable In-Memory Cryptographic Cortex-M0 Processor for IoT. *Symp. on Very Large-Scale Integration Circuits (VLSIC)*, Jun 2017.