PROGRAMMING FRAMEWORKS FOR IMPROVING THE PRODUCTIVITY AND PERFORMANCE OF MANYCORE ARCHITECTURES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by
Lin Cheng
December 2022

PROGRAMMING FRAMEWORKS FOR IMPROVING THE PRODUCTIVITY AND PERFORMANCE OF MANYCORE ARCHITECTURES

Lin Cheng, Ph.D.

Cornell University 2022

Manycore architectures integrate hundreds of cores on a single chip by using simple cores and simple memory systems usually based on software-managed scratchpad memories (SPMs). Such architectures are notoriously challenging to program, since the programmers need to manually manage all aspects of data movement and synchronization for both correctness and performance. This manycore programmability challenge is one of the key barriers to achieving the promise of manycore architectures. This thesis presents both domain-specific (`HB-PyTorch` and `HB-Arc`) and general-purpose (`HB-Rubick`) programming frameworks to address the SPM manycore architecture programmability challenge and/or improve performance. `HB-PyTorch` enables domain experts to easily accelerate off-the-shelf tensor workloads. Evaluation on three real-world dense and sparse tensor workloads suggests these workloads can achieve approximately 2–6× performance improvement when scaled to a future 2,000-core manycore system compared to an 18-core out-of-order CPU baseline, while potentially achieving higher area-normalized throughput and improved energy-efficiency compared to GPGPUs. `HB-Arc` explores the potential of decoupled access/execute (DAE) mechanisms, and proposes two software-only techniques, naïve-software DAE and systolic-software DAE, along with a lightweight hardware access accelerator for further performance benefit. However, being domain-specific limits their scope. General purpose dynamic task parallel programming frameworks offer many advantages over domain-specific frameworks, including more flexibility and better load-balancing. Conventional wisdom suggests a work-stealing runtime, which forms the core of most dynamic task parallel programming models, is ill-suited for manycore architectures. However, `HB-Rubick` demonstrates that such a runtime is not just feasible on manycore architectures with SPMs, but it can also significantly improve the performance of irregular workloads when executing on these architectures. The proposed dynamic task parallel programming framework enhanced with three optimizations for leveraging unused SPM space achieves 1.2–28.5× speedup on workloads that benefit from our techniques, and only induces min-

imal overhead for workloads that do not. This thesis provides a small yet important step towards closing the performance and productivity gap of SPM manycore architectures.

BIOGRAPHICAL SKETCH

Lin Cheng was born on Aug 20, 1993 to Mei Li and Zhenxue Cheng in Weifang, Shandong, China. At a young age, he went through a few ups and downs. He found himself interested in computers while in elementary school, and started learning BASIC by himself. However, he did struggle to understand and remember the DOS commands taught in his 1st grade computer course. He was also annoyed by C++ templates as a 6th grade student. During his time in Weifang Guangwen Middle School, he started to dream about turning his interest into his career choice and being a software developer after college. Although this idea received strong objection from his mother, he did not listen and kept his dream alive.

Lin was accepted to University of Illinois at Urbana-Champaign as an undergraduate student. However, he was not accepted to major in computer science as he expected, largely due to his not-sufficiently-high SAT score. Instead, he was in the department of general studies, and till now he still believes that "undecided is a major choice". Thanks to course registration restrictions on non-CS students, Lin took a brand new undergraduate computer architecture course as a freshman and it was the first semester they offer this course. To his surprise, he found himself good at these topics. The instructor stated that, "A+'s are like unicorns. They exist but no one knows why." Top 0.5% of the class got A+ and Lin was one of them. He transferred to the department of computer science a year later, and decided that computer architecture is what he wants to study. Later Lin was admitted to the 5th-year MS program, and was advised by Professor Sarita Adve during his master's study.

Lin was accepted to the PhD program in Cornell Ann S. Bowers College of Computing and Information Science major in computer science in 2017. Throughout the next five and a half years, he was fortunate to be advised by Professor Chris Batten, and had Professor Adrian Sampson, Professor Zhiru Zhang, and Professor Jose Martinez on his committee. On his journey at Batten Research Group and away from Gates Hall, he was involved in multiple projects, spanning from just-in-time compilers for dynamic programming languages to gate-level simulations. During his time at the Computer Systems Lab, he was also lucky enough to be accompanied by his friends from Rhodes Hall 471B, from other uncharted offices in CSL, and from everywhere else at Cornell.

Lin is grateful for his time at Cornell and in Ithaca. While being stuck in the middle of nowhere in upstate NY, he was able to acquire many useful skills, including doing good research, professional writing, debugging, and most importantly cooking.

This document is dedicated to Bip, my wife, DanDan, my cat who helped by being cute, and everyone I've met along the way...

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support, encouragement, and advice from many people. First and foremost I'd like to thank my advisor, Christopher Batten, who was tremendously influential not only for my research direction, but also for the development of my professional skills. He has been trying to teach me how to be a good researcher and reminding me that we are more than engineers. When I told Chris I'd like to work on a topic since it has not been done yet, he guided me to be a scientist and find its research value.

I am deeply thankful to my colleagues in the Batten Research Group for all of their support through the years. I still remember walking down the hill to his car at night with Tuan Ta while we were arguing about details of the MESI coherence protocol. I can also vividly recall when Shreesha Srinath told me to make an accelerator for Halide. Of course I won't forget chatting with Berkin Ilbeyi during my campus visit. Even though I did not have the chance to collaborate directly with Shunning Jiang and Christopher Torng, the night we were all working in Chris' hotel room before the day of our PyMTL3 tutorial at FCRC'19 feels like yesterday. I think I can still hear Khalid Al-Hawaj yelling my name and asking for a squash game, Peitian Pan calling "flash out" in de_dust2, and Yanghui telling me he was so sleepy at 3 AM. That bug I fixed after eyeballing an 8 GB memory access trace for two weeks for Moyang Wang is still my favorite bug. Nick Cebry and Preslav Ivanov, though we did not have much interaction largely because you joined the group during the pandemic, I really hope you are and will continue enjoying your time here in CSL.

I am also deeply thankful to my friends within the CSL community, across the baseball field in Gates Hall, and on that tiny island in NYC. I want to give a special callout to Yi Jiang, who helped build the "culture of 471B". I am glad that you were around to chat about anything and everything in the office, in the kitchen, and over the air throughout the years, and I'm so glad you were around to hear all my complaints about grad school and to help me survive it. I would like to give special thanks to Danna Ma and Shaojie Xiang for dragging me into various strange activities around Ithaca. I also want to thank Zhen Sun for being my lunch buddy during my first year, and Philip Bedoukian for yelling my name every morning when he walks in.

Special thanks to the steering committee of the International Symposium on Gossips of Computer Systems Lab (ISGCSL) and all the contributors. I could not have stayed in the program and been able to write this thesis without participating in this conference. Although I cannot disclose their names, I'd like to thank all of them for convincing me that "I have no idea".

I would like to thank Carl Friedrich Bolz-Tereick. He has a wealth of knowledge for everything PyPy, dynamic languages, and JITs in general. More importantly, he has been a great collaborator and mentor. I'm really thankful to his guidance and advice, both inside and out of work. I would like to thank Prof. Adrian Sampson and Prof. Zhiru Zhang for being on my committee and for their insightful feedback and suggestions on this thesis. Their courses were fun to take as well. I also would like to thank the Bespoke Silicon Group (BSG) at the University of Washington, including Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Dustin Richmond, Mark Oskin, and Michael B. Taylor. This thesis is only possible because BSG has open sourced the HammerBlade manycore.

I also would like to thank all my friends on Steam, especially `AmoDemo`, `Ring-of-Aquila`, `ting`, `hai`, `wuyuebugui`, `copy`, `s1mple`, `haooline`, and `CacheMiss`. Even though I have never had the honor to meet most of them in person, and I haven't even seen their pictures, we spent 7610.5 hours together playing DotA2, 965.1 hours playing CS:GO, and countless hours just chatting in our Discord channels. I could not have survived Ithaca without these friends. However, occasionally I do think that if we never met, I would be able to graduate much faster.

Finally, I want to thank my wife Bip (i.e., Mengqiao Han) for *everything*. I know Ithaca is the last place you want to spend your time and it wasn't easy for you. Thank you so much for all the daily support, countless late night food dishes, and periodically asking me "when will you graduate" throughout the years. I also want to thank our cat, DanDan, who did not help at all.

This work was supported in part by NSF SHF Award #1527065, NSF CRI Award #1512937, NSF PPoSS Award #2118709, DARPA SDH Award under Grant FA8650-18-2-7863, 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 theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

HB	HammerBlade
MC	manycore
GPGPU	general-purpose graphics processing unit
SIMD	single-instruction multiple-data
SPMD	single-program multiple-data
TBB	(Intel) threading building blocks
PGAS	partitioned global address space
DAE	decoupled access/execute
DMA	direct memory access
CUDA	(NVIDIA) compute unified device architecture
MPI	message passing interface
API	application programming interface
SDK	software development kit
ISA	instruction set architecture
CSR	compressed sparse row
CBSR	cyclic bank sparse row
CNN	convolutional neural networks
DSL	domain-specific language
RTL	register-transfer level
VLSI	very-large-scale integration
ASIC	application-specific integrated circuit
FPGA	field-programmable gate array
ALU	arithmetic logic unit
FPU	floating point unit
OCN	on-chip network
AX	access accelerator
SoC	system-on-chip
SRAM	static random access memory
DRAM	dynamic random access memory
HBM	high bandwidth memory
I$	instruction cache
D$	data cache
LLC	last-level cache
SPM	scratchpad memory

CHAPTER 1
INTRODUCTION

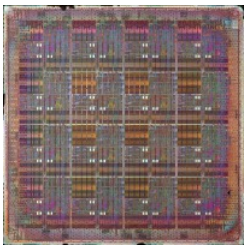
Parallelism and specialization have been the two main techniques for turning the ever increasing number of transistors provided by Moore's Law into performance. A simple way to exploit more parallelism is to have more cores and create *manycore architectures* [TKM+03, MFN+17, HDH+10,HVS+07,LSC+13,VGT+20,TFZ+08,BEA+08,Ram11,Kan15,Whe20,Hal20,WGH+07, LFF+18, kal22, BSP+17, Olo16, ZSB21, DXT+18]. Examples include data-parallel manycore architectures such as general-purpose graphics processing unit (GPGPU) and thread-parallel manycores such as Tile64 [BEA+08] and Celerity [DXT+18]. The *manycore* approach trades a few complex big cores for a large number of simple cores integrated within a single die using a tiled physical design methodology. Compared to general-purpose multi-cores, the manycore approach can improve energy efficiency and throughput per unit area on highly parallel workloads. Compared to specialized hardware (i.e., domain- and application-specific accelerators), the manycore approach is more flexible and can be tailored to accelerate a wider range of applications. However, the flexibility offered by manycore architectures means programmers must navigate a broad software design and optimization space. This is compounded by the fact that manycore processors rely on simple hardware that requires programmers to manage many concerns such as data coherence among private memories manually in software, write applications in low-level C environments and/or directly in assembly, and adopt a more restricted programming model. The cumbersome programming environment coupled with the need for software optimizations to realize the performance promised by hardware is a critical barrier to widespread adoption of most manycore architectures, especially those with software-managed scratchpad memories (SPMs).

In this thesis, I propose both domain-specific and general-purpose programming frameworks to improve the programmability and/or the performance of thread-parallel manycore architectures with software-managed scratchpad memories (i.e., SPM manycore architectures). I first present a brief introduction on the target SPM manycore architecture and discuss manual performance tuning on the target system. I then introduce two domain-specific frameworks for tensor computation and decoupled access/execute programming on SPM manycore architectures. Lastly, I present a general-purpose dynamic task parallel programming framework and evaluate three optimizations that enable the framework to leverage unused scratchpad space for further performance improvement.

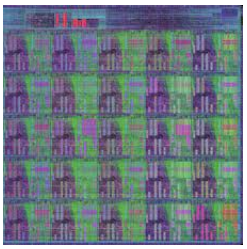
1.1 The Manycore Architecture Era

Manycore processors date back to the early 2000s, when a few research prototypes were made to demonstrate the potential of the manycore approach in executing thread-parallel workloads. Early thread-parallel manycore research prototypes integrated 16–110 cores on a single die. The MIT RAW processor [TKM+03] integrated 16 simple in-order cores with a 4×4 2-D mesh on-chip network (OCN). The Intel Teraflops research chip [HVS+07] contained 80 tiles arranged as a 10×8 2-D mesh OCN of floating-point cores and routers. The Godson-T processor [TFZ+08] from the Institute of Computing Technology (ICT) at the Chinese Academy of Sciences (CAS) had 64 cores organized into an 8×8 2-D mesh OCN. The Intel Single-Chip Cloud Computer (SCC) [HDH+10] was a manycore processor with 48 Pentium cores connected by a 4×6 2-D mesh OCN. The 110-core Execution Migration Machine (EM^2) [LSC+13] is a directory-less shared-memory manycore based on hardware-level thread migration which had a 10×11 layout. The industry has adopted the manycore approach as well and products available typically include 64–256 cores. Examples include the 64-core Tile64 [BEA+08], the 72-core Knights Landing [SGC+16], the 100-core Tile GX100 [Ram11], the 128-core Ampere Altra Max [Whe20], the 128-core Sunway SW26010 [LFF+18], and the 256-core Kalray MPPA-256 [kal22]. Recent research prototypes have scaled core counts by an order-of-magnitude to over a thousand cores, including the 1000-core KiloCore [BSP+17], the 1024-core Epiphany-V [Olo16], and the 4096-core Manticore [ZSB21]. GPGPUs are the most widely adopted type of manycore processor. Unlike the thread-parallel manycore architectures mentioned above, GPGPUs are data-parallel and usually adopt a single instruction multiple data (SIMD) architecture. While they usually have about a hundred cores (referred to as stream multiprocessors for NVIDIA GPGPUs and compute units for AMD GPGPUs), they support thousands of concurrent hardware threads by having one frontend driving multiple scalar pipelines. For instance, The NVIDIA A100 GPGPU has 6912 CUDA cores (i.e., scalar pipelines) in 108 SMs [nvi20].

Hardware designers have long realized that it is more difficult to efficiently implement existing hardware-based cache coherence protocols designed for multi-core processors (e.g., directory-based MESI and its variants) on manycore architectures as their core count keeps increasing. Designing a performance-, complexity-, and area-scalable hardware-based cache coherence protocol has been and remains an active area of research [ZSD10, CLS05, ZSM07, Mos05, MHS12,



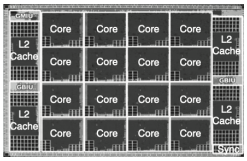
(a) The 16-core MIT RAW Processor [TKM+03]



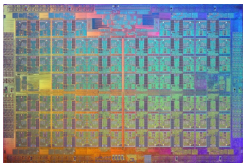
(b) The 25-core Piton [MFN+17]

(c) The 48-core Intel Single-Chip Cloud Computer [HDH+10]

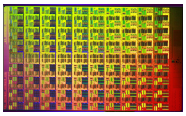
(d) The 60-core Marvell ThunderX3 [Hal20]



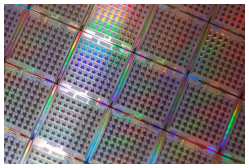
(e) The 64-core ICT Godson-T [TFZ+08]



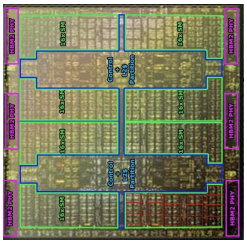
(f) The 72-core Intel Knights Landing [SGC+16]



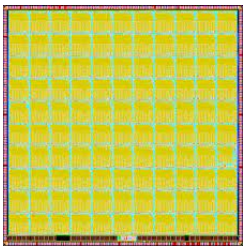
(g) The 80-core Intel Teraflops Research Chip [HVS+07]



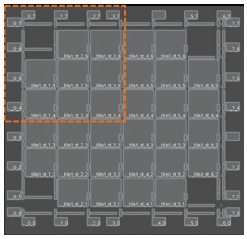
(h) The 100-core TILE-GX100 [Ram11]



(i) The 108-core NVIDIA A100 [nvi20]

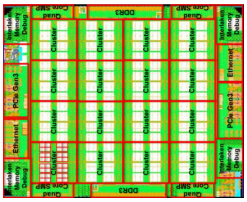


(j) The 110-core Execution Migration Machine (EM^2) [LSC+13]

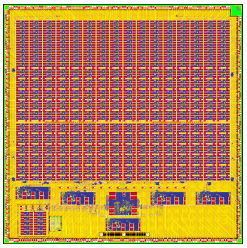


(k) The 128-core Ampere Altra Max [Whe20]

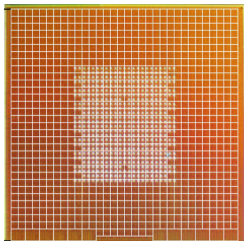
(l) The 128-core Sunway SW26010 [LFF+18]



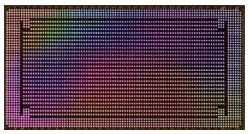
(m) The 256-core Kalray MPPA-256 [kal22]



(n) The 511-core Celerity Research Chip [DXT+18]



(o) The 1024-core KiloCore [BSP+17]



(p) The 1024-core Adapteva Epiphany-V [Olo16]

Figure 1.1: Examples of Manycore Processors – Chip plots or die photos of selected manycore processors mentioned in this thesis.

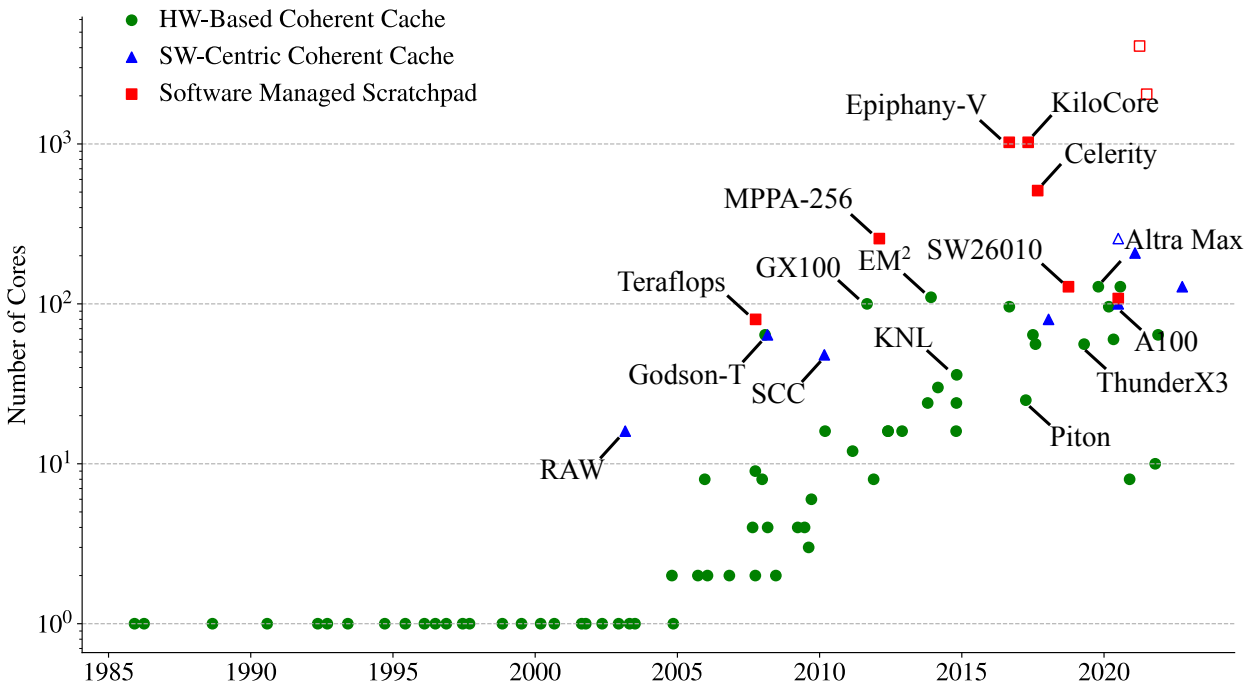


Figure 1.2: Trend of Processor Core Count and on Chip Memory Hierarchy – This figure shows the number of cores and their on chip memory hierarchy in selected processors from 2000 to 2022. Filled marker = real chip; unfilled marker = proposal/simulator only. The data is in part from CPU DB [DKM+12]. See Figure 1.1 for citations.

FLKBF11, BS13, FW15]. When hardware designers scale the number of cores on a single chip from tens to around a hundred cores, both academia and industry have started moving away from hardware-based cache coherence and adopting software-centric cache coherence, which requires programmers to explicitly conduct cache invaldiation and/or dirty data writeback (e.g., Temporal-Coherence [SSF+13]). Examples include Godson-T [TFZ+08], Teraflops [HVS+07], and GPGPUs [nvi20]. As the core count continues scaling into over a thousand cores, software-managed scratchpad memory (SPM) [BSL+02,KSA+15,DXT+18] has become the common choice [BSP+17, Olo16, LFF+18, DXT+18]. The trend is illustrated in Figure 1.2. Manycore architectures, especially these that adopt SPMs are notoriously challenging to program because of their high demand on programmers. This manycore programmability challenge is one of the key barriers to achieving the promise of manycore architectures.

```python
# import library
import numpy as np

# create a sequence of float numbers
# from 0 to 5 and arrange them as
# a 2 by 3 matrix
x = np.arrange(6).reshape(2, 3)
x = x.astype('f')

# take sum along the y-axis
res = x.sum(axis=1)
```

(a) Legacy Code with NumPy.

```python
# import library
import cupy as cp

# create a sequence of float numbers
# from 0 to 5 and arrange them as
# a 2 by 3 matrix
x = cp.arrange(6).reshape(2, 3)
x = x.astype('f')

# take sum along the y-axis
res = x.sum(axis=1)
```

(b) Ported Code with CuPy.

Figure 1.3: CuPy Example – CuPy is designed to be a drop-in replacement of NumPy and SciPy. Porting a piece of legacy code to run on GPGPUs is as simple as replacing `numpy` with `cupy` in lines 2 and 7.

1.2 Domain-Specific Frameworks

One approach to resolve this programmability challenge of manycore architectures is through specialized or domain-specific frameworks that provide either ready-to-use hand-optimized operators embedded within a high-level language or carefully designed domain-specific languages (DSLs). Such domain-specific frameworks played an important role in the adoption of GPGPUs by simplifying both writing new software and reusing existing software.

CuPy [OUN+17] is an open-source array library for GPGPU-accelerated computing with Python. CuPy's programming interface is crafted to be highly compatible with widely used array libraries on traditional multi-core CPUs like NumPy and SciPy [Oli07]. In most cases it can be used as a drop-in replacement: simply replace `numpy` and `scipy` with `cupy` and `cupyx.scipy` in the existing Python code. See Figure 1.3 for an example. Under the hood, CuPy is built on top of the low-level CUDA Toolkit [nvi22] framework, which includes cuBLAS for linear algebra, cuRAND for random number generation, cuSOLVER for solving dense and sparse linear systems, cuSPARSE for sparse linear algebra, cuFFT for fast Fourier transformation, cuDNN for deep neural networks and NCCL for multi-GPU communication to make full use of the GPGPU architecture.

PyTorch [PGM+19] is an open-source deep learning framework that supports various compute platforms, including traditional multi-core processors and GPGPUs. The programming interface of PyTorch is designed to be platform agnostic and contains only abstract operators. At runtime, a dispatching mechanism automatically picks the appropriate platform specific implementations. This extra layer of abstraction provides high code portability. The same code base can run on

```
class Autoencoder(nn.Module):
  def __init__(self):
    self.encoder = nn.Sequential(
      nn.ReLu(),
      nn.BatchNorm1d(800),
      nn.Dropout(0.5)
    )

    self.bneck = nn.Linear(800, 400)

    self.decoder = nn.Sequential(
      nn.ReLu(),
      nn.BatchNorm1d(400),
      nn.Dropout(0.5)
    )

  def forward(self, x):
    x = self.emb(x).sum(dim=1)
    x = self.encoder(x)
    x = self.bneck(x)
    x = self.decoder(x)
    x = self.output(x)
```

(a) Autoencoder model definition.

```
def train(dataloader_train):
  model = Autoencoder()

  for x, y in dataloader_train:
    out = model(x)
    loss = F.MSELoss(out, y)

    opt.zero_grad()
    loss.backward()
    opt.step()
```

(b) Training script on multi-core CPUs.

```
def train(dataloader_train):
  model = Autoencoder().cuda()

  for x, y in dataloader_train:
    x = x.cuda()
    y = y.cuda()
    out = model(x)
    loss = F.MSELoss(out, y)

    opt.zero_grad()
    loss.backward()
    opt.step()
```

(c) Training script on GPGPUs.

Figure 1.4: PyTorch Example – Only three lines of code (i.e., lines 2, 5, 6 of (c)) are needed for porting a deep learning model to run on GPGPUs.

various platforms with only minimal changes. Figure 1.4 shows an example of a deep learning model written with PyTorch. Only three lines of code are needed for porting the model which originally trains on multi-core processors (i.e., Figure 1.4 (b)) to leverage GPGPUs (Figure 1.4 (c)), and none of the code that defines the model (i.e., Figure 1.4 (a)) is changed as they are all platform agnostic. The abstraction layer also enables constructing new abstract operators with a sequence of existing operators which further improves encapsulation and programmability.

Other domain-specific framework examples include cuGraph [rap20] and Gunrock [WDP+16] for graph analytics, TensorFlow [ABC+16] for machine learning, CUVIlib [cuv22] for image processing, and Triton Ocean SDK [Sun22] for water simulation. While these frameworks express domain-specific workloads effectively and achieve high performance, they do not cover all domains. Extending and repurposing them for another domain requires non-trivial effort by programmers.

1.3 General-Purpose Frameworks

Unlike domain-specific frameworks that have a narrow focus, general-purpose programming frameworks provide more flexibility. In the multi-core era, general-purpose parallel programming frameworks with programmer friendly programming models, especially ones that support dynamic task parallelism, played a key role in exploiting/expressing parallelism and achieving high performance. Task parallelism is a style of parallel programming where the workload is divided into *tasks* (i.e., units of computation that can execute in parallel). Dynamic task parallelism is a subset of task parallelism in which tasks and dependencies among tasks are generated at runtime. Dynamically generated tasks are assigned to available worker threads based on a certain scheduling algorithm. They can express a wide range of parallel patterns, provide automatic load balancing, and improve portability for legacy code [MRR12]. Examples include Intel Cilk Plus [int13], Intel Threading Building Blocks (TBB) [int19], and OpenMP [ACD+09, ope13]. Figure 1.5 shows an example of calculating the Fibonacci sequence with the Intel Cilk Plus framework, which adopts the *fork-join* computation model. In such a model, the process in which a task forks two or more parallel tasks is also referred to as *spawning* tasks. The newly created tasks are called the *child* tasks and the original task is called the *parent* task. The parent task can continue until it reaches the point where the *join* (also commonly referred to as *wait* or *sync*) primitive is called. It is then blocked until all of its child tasks have finished. In this example, the parent task (i.e., `fib(n)` *spawns* two tasks, `fib(n - 1)` and `fib(n - 2)`. The parent task is suspended with `cilk_sync` until both child tasks are finished. It then calculate the result of `fib(n)` by adding the return values of both child tasks.

Adopting the dynamic task parallel programming model on manycore architecture can potentially help with resolving the programmability challenge of manycore architectures by both enabling efficient development of new software and easy porting of existing software. Such a framework can also yield better performance by providing better load-balancing. However, conventional wisdom suggests a work-stealing runtime, which forms the core of most dynamic task parallel programming models, is ill-suited for manycore architectures due to the lack of hardware coherent caches [ZP16, WTCB20].

```
1  uint32_t fib(uint32_t n) {
2    uint32_t x, y;
3
4    if (n < 2) {
5      return n;
6    }
7
8    x = cilk_spawn fib(n - 1);
9    y = cilk_spawn fib(n - 2);
10   cilk_sync;
11
12   return (x + y);
13 }
```

Figure 1.5: Intel Cilk Plus Example – The nth Fibonacci number is calculated by spawning two child tasks, one for calculating the (n-1)th Fibonacci number and one for calculating the (n-2)th Fibonacci number.

1.4 Thesis Overview

This thesis presents both domain-specific and general-purpose approaches to address the many-core architecture programmability challenge. I will limit the discussion in this thesis to thread-parallel manycore architectures with software-managed scratchpad memories (SPM manycore architectures). Compared to data-parallel manycore architectures (e.g., GPGPUs), the software stack of thread-parallel manycore architectures is less explored. How to efficiently program such systems with thousands of cores remains an open research question. An overview of this thesis is illustrated in Figure 1.6.

Chapter 2 gives a brief introduction on an open source SPM manycore architecture, HammerBlade (HB), which captures the common features of modern manycore systems. Examples of these common features include relatively simple cores, software-managed memory systems, mesh-based on-chip networks, and simple low-level programming interfaces. In Chapter 2.2, I provide an introduction on the low-level C runtime, CUDA-lite, of the HammerBlade manycore. CUDA-lite adopts a single-program-multiple-data (SPMD) programming model, and statically scheduled parallel loops are the only supported parallel pattern. In Chapter 2.4, I discuss the details of hand tuning and optimizing kernels on the HammerBlade manycore architecture using matrix multiplication as an example. Hand tuning requires programmers to have a deep understanding of both the kernel to be optimized and the underlying manycore hardware. Hand tuning a kernel also often involves manual instruction scheduling and writing assembly code directly, which further reduces the programmability of such systems.

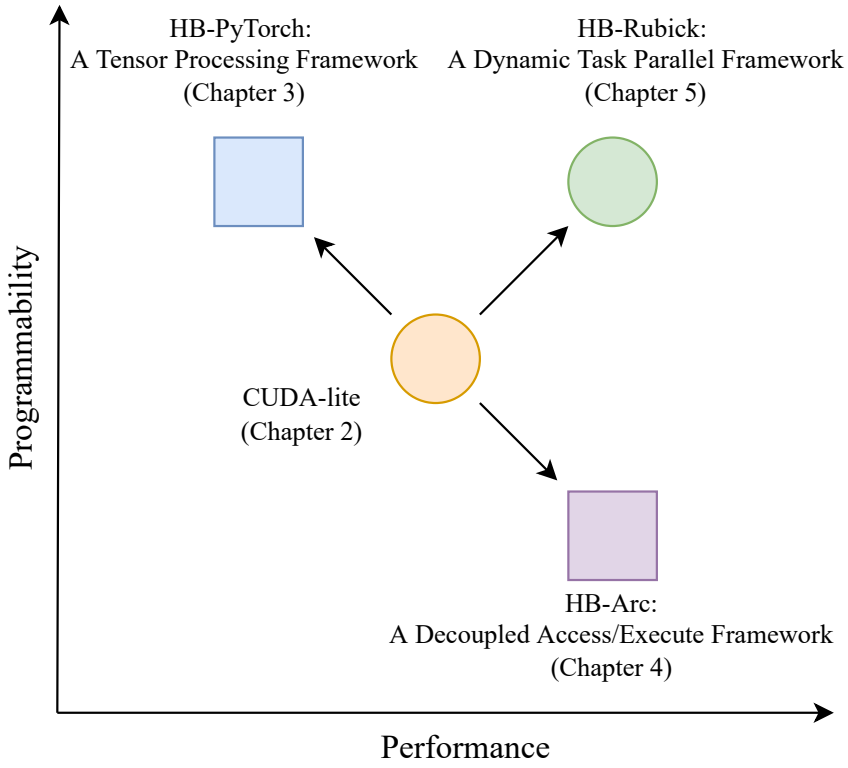


Figure 1.6: Thesis Overview – This thesis explores implementing the software stack of manycore architectures. Chapter 2 provides an introduction on the low-level C runtime of the HammerBlade manycore architecture and gives a case study of hand tuning kernels on HammerBlade by using matrix multiplication as an example; Chapter 3 implements a tensor processing framework on the HammerBlade manycore; Chapter 4 explores decoupled access/execute and systolic execution; Chapter 5 describes, to the best of our knownledge, the first implementation of a work-stealing runtime on manycore architectures with SPMs. Squares = domain-specific frameworks; circles = general-purpose frameworks.

Chapter 3 presents a domain-specific framework, `HB-PyTorch`, for tensor processing on the HammerBlade manycore architecture. In this chapter, I attempt to resolve the manycore architecture programmability challenge by extending PyTorch, a widely adopted tensor processing framework, with a manycore backend. The proposed framework allows deep learning developers to take their existing deep learning models and run them on the HammerBlade manycore by modifying a few lines of code. Compared to writing hand optimized kernels from scratch, our framework significantly improves the programmability of manycore architectures by providing ready-to-use operators that are embedded in a high-level language. However, compared to implementing a workload natively with CUDA-lite, the same workload written with `HB-PyTorch` usually achieves

lower performance because of the overhead of HB-PyTorch's Python frontend and more frequent interaction between the HammerBlade manycore and the host CPU.

Chapter 4 explores software and hardware solutions to enable decoupled access/execute (DAE) and systolic execution on manycore architectures. I demonstrate that DAE and systolic execution are feasible solutions to cope with the ever decreasing per core memory bandwidth as the number of cores keeps increasing in manycore architectures. DAE and systolic execution improves performance but requires programmers to manually reformat and hand tune their applications.

Chapter 5 discusses a general-purpose framework, HB-Rubick, which supports dynamic task parallelism on the HammerBlade manycore architecture. While work-stealing runtimes, which forms the core of most dynamic task parallel programming models, is considered ill-suited for manycore architectures as they usually lack hardware-based coherent caches, I demonstrate that such runtimes are more than just feasible on manycore architectures. A work-stealing runtime can also significantly improve the performance of irregular workloads on such systems with SPMs. I also explore three optimization techniques to enable the work-stealing runtime to leverage unused scratchpad space for further performance benefit. Compared to CUDA-lite, the proposed framework supports parallel patterns beyond static parallel loops, provides dynamic load-balancing, and allows them to be arbitrarily nested. It enables programmers to efficiently express a wider range of algorithms and achieve high performance on irregular workloads. Moreover, the more familiar programming model and interface make it much easier to port legacy code written for traditional multi-core processors to manycore architectures.

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

- an open-source tensor processing framework, which achieves high performance on SPM manycore architectures;
- a novel framework, which enables decoupled access/execute (DAE) and systolic execution on SPM manycore architectures;
- an open-source dynamic task parallel programming framework, which supports arbitrarily nested parallel patterns and dynamic load balancing on SPM manycore architectures; and
- software and hardware optimizations to enable a work-stealing runtime to leverage unused scratchpad space and achieve higher performance on SPM manycore architectures.

1.5 Collaboration, Previous Publications, and Funding

This thesis would not have been possible without the support from my advisor, Christopher Batten, and contributions from my colleagues at Cornell University and outside collaborators. My advisor Christopher Batten was a primary source of inspiration and guidance, and he was integral in all aspects of my research projects.

The HammerBlade manycore architecture described in Chapter 2 is developed by the Bespoke Silicon Group (BSG) at the University of Washington. This thesis is only possible because BSG has open sourced HammerBlade. Particularly, Shaolin Xie and Dai Cheol Jung led the RTL development of HammerBlade; Max Ruttenberg and Dustin Richmond created CUDA-lite and the co-simulation infrastructure; and Bandhav Veluri worked on the compiler/linker for HammerBlade. I collaborated with many colleagues from BSG within the DARPA SDH project. This includes Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Dustin Richmond, Mark Oskin, and Michael B. Taylor.

I led the tensor processing framework and DAE schemes for CPU-manycore heterogeneous systems work presented in Chapter 3 and Chapter 4, but the project would not have been possible without significant contributions from my colleagues Peitian Pan and Zhongyuan Zhao. Peitian led the design and development of the Access Accelerator (AX) and Zhongyuan led the development and implementation of sparse operators in the proposed tensor processing framework. The project would also not have been possible without inputs from various operator writers including Krithik Ranjan, Jack Weber, Kexin Zhang, Janice Wei, Angela Zou, Yuwei Hu, and Adrian Sampson. Bandhav Veluri helped with laying the groundwork of the framework. Zichao Yue contributed to this project by developing the CBSR sparse matrix format. Dustin Richmond provided invaluable input and feedback to this project. This work is published in IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD) in 2022 [CPZ+22].

I led the dynamic task parallel framework work together with Max Ruttenberg at the University of Washington. We contributed equally to this project. The work-stealing runtime described in Chapter 5 is based on the work of Moyang Wang. This work is submitted to the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) and is currently under review.

I collaborated with Berkin Ilbeyi on the software/hardware co-design to exploit object dereference locality work. I implemented a functional model of the Bloom filter. Later I led the work of exploiting attribute type monomorphism in tracing JIT compilers. Both projects received tremendous input from Carl Friedrich Bolz-Tereick from the University of Düsseldorf, Germany. Both projects are possible because of Carl's help and feedback. While I did not include this work in this thesis, it is published in the ACM/IEEE International Symposium on Code Generation and Optimization (CGO) in 2020 [CIBTB20].

This work was supported in part by NSF SHF Award #1527065, NSF CRI Award #1512937, NSF PPoSS Award #2118709, DARPA SDH Award under Grant FA8650-18-2-7863, 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 theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

CHAPTER 2
A PROGRAMMER'S VIEW OF THE HAMMERBLADE MANYCORE ARCHITECTURE

Manycore architectures provide thread-level parallelism and flexibility with hundreds to thousands of general purpose cores, which are typically arranged in two-dimensional arrays and interconnected with packet-based mesh-style OCN for communication. This network of cores is usually surrounded by multiple channels of memory. Cores within the architecture communicate explicitly through memory [DXT+18] or message passing [Gwe11], implicitly through coherence protocols [Ram11], or both using inter-core result networks [TLM+04]. Abundant general purpose cores and diverse communication patterns make manycore architectures flexible enough to be tailored to fit a wide range of parallel applications. Although the manycore software and hardware design space is broad, there are several common features including relatively simple cores, mesh-based OCNs, software-managed memory systems, and simple low-level programming interfaces.

In this chapter, I present a brief introduction on an early version of the HammerBlade (HB) architecture [BFY+21], an open-source manycore which captures these commonly found features. The HammerBlade manycore is designed and implemented by the Bespoke Silicon Group at the University of Washington. While the techniques and mechanisms proposed in this dissertation are implemented and evaluated on the HammerBlade architecture, they are generally applicable to other manycore architectures as well. Section 2.1 gives an overview of the HammerBlade architecture hardware. Section 2.2 describes the software stack and the programming interfaces of the HammerBlade manycore. Section 2.3 discusses our RTL simulation and energy modeling methodologies. Section 2.4 presents a detailed example of hand optimizing a widely used kernel, `MatMul`, on HammerBlade.

2.1 HammerBlade Manycore Hardware

The full HammerBlade CPU-manycore heterogeneous system includes a traditional multicore CPU and an HammerBlade co-processor each with its own dedicated DRAM memory; the multicore CPU uses DDR4 DRAM for high capacity, while the manycore co-processor uses die-stacked HBM2 DRAM for higher bandwidth. The HammerBlade manycore co-processor includes 2000 simple cores interconnected to the on-chip HBM2 memory controllers through a global net-

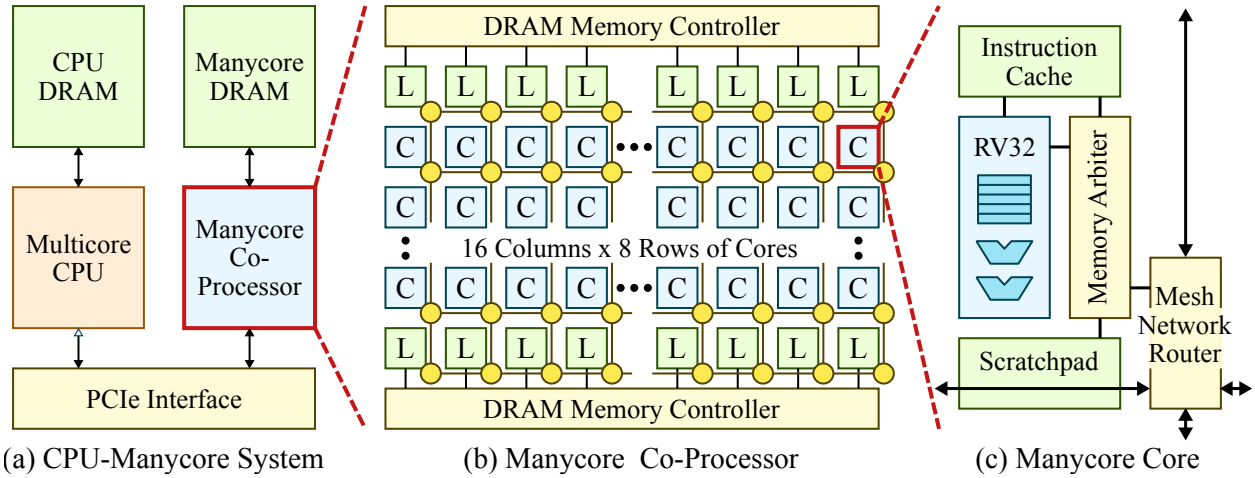


Figure 2.1: HammerBlade CPU-Manycore Heterogeneous System Hardware – (a) target system includes a CPU with its own attached DRAM and a manycore co-processor also with its own attached DRAM; (b) manycore co-processor includes 16×8 simple cores (C) and 32 last-level cache (L) banks interconnected via mesh-based on-chip network; (c) each core is a RISC-V RV32IMAF processor (RV32) with 4 KB instruction cache and 4 KB scratchpad memory.

work. In this thesis, we study an early version of the HammerBlade mancyore which includes 128 cores arranged into an 16×8 grid interconnected via an on-chip mesh network, illustrated in Figure 2.1. This 128-core HammerBlade co-processor includes a last-level cache (LLC) which is shared among all cores. The LLC is divided into 32 address-interleaved banks located at the top and bottom of the mesh. Each core is a simple, ultra-efficient RISC-V core supporting the RV32IMAF instruction set including basic arithmetic operations, atomic memory operations (handled in the LLC banks), and single-precision floating point operations. Each core also includes a 4-KB instruction cache and 4-KB software-managed scratchpad memory (SPM).

2.1.1 Core Microarchitecture

The core uses a single-issue, in-order, five-stage integer pipeline with additional long-latency functional units including a two-cycle pipelined integer multiplier, a three-cycle pipelined floating-point unit, and a 32-cycle iterative divider. The core implementation has been carefully optimized to ensure it can achieve the highest performance in the least amount of area and energy. A critical feature of the core microarchitecture is support for *non-speculative runahead execution*, also called *stall-on-use*, in the spirit of prior work [DM97, CHA+15]. After a remote load is injected into the on-chip network, the core will continue executing subsequent independent instructions. The

core will not wait for the load data to return until it reaches a dependent instruction. Note that this mechanism is completely non-speculative and does not include any form of rollback and re-execution. Careful instruction scheduling can enable many remote loads to be in flight at once.

2.1.2 Memory System

The memory system for the manycore co-processor has four hierarchical levels: HBM2 DRAM, LLC, core-remote SPM, and core-local SPM. Each level is designed to exploit memory parallelism and exposes a trade-off between latency and capacity. Core-local SPM has the lowest latency and smallest capacity, and HBM2 has the highest latency and largest capacity. HBM2 provides two sources of memory-level parallelism. First, HBM2 provides channel-level parallelism with eight independent physical channel interfaces per package with a maximum data transfer rate of 32 GB/s per channel. Second, HBM2 provides bank-level parallelism through pipelined commands. Commands for opening/closing banks and reading data can overlap to hide the latency of long-running commands. Compared to traditional DDR4 DRAM, HBM2 provides more bandwidth and parallelism per-package, but overall system performance depends on carefully exploiting channel- and bank-level parallelism. To this end, the banked LLCs are designed to exploit bank-level parallelism within a channel. As mentioned previously, each bank of the LLC is connected to a column in the manycore architecture. The top LLC banks share one HBM2 channel, and the bottom LLC banks share a second HBM2 channel. Each LLC bank is mapped to a unique address range, and each port is mapped to an exclusive set of HBM2 banks within the HBM2 channel. The core-local SPMs eliminate coherence overhead and false sharing, and enable software to keep stack-allocated data local and stage remote data for reuse. Critically, every core can also directly access any SPM in the system by using regular load and store instructions creating a partitioned global address space (PGAS) and enabling new optimizations and programming models.

2.1.3 On-Chip Network

The cores and memory system are all interconnected through a highly optimized 2D-mesh-with-ruching on-chip network (OCN) based on an earlier silicon-validated design [RZAH$^+$19a, RZAH$^+$19b, JDZ$^+$20, OAB20]. The network uses dimension-ordered routing, single-flit packets, and includes two physical networks to avoid protocol-level deadlock. The OCN preserves ordering

between endpoints. Every word in the SPMs, all configuration registers, and the HBM2 DRAM are mapped into a single unified physical address space, and all packets are single-word memory request/response packets using this unified address space.

2.1.4 Area and Timing

The small-scale 128-core HammerBlade manycore has been implemented in RTL and validated in silicon to enable accurate cycle-level simulation and performance analysis, and a state-of-the-art commercial standard-cell-based toolflow was used to characterize area and timing in an advanced GF CMOS 14 nm technology node. Preliminary area analysis suggests a single core requires approximately 30,000 μm^2, meaning a 128-core HammerBlade manycore (including 32 LLC banks) is approximately 5 mm^2 and the future full 2000-core manycore co-processor in the target system is only 80 mm^2. Timing analysis suggests the manycore co-processor can easily run at 1 GHz and could reach 2 GHz with sufficient physical design optimization. The 128-core HammerBlade manycore running at 1–2 GHz is able to achieve 256–512 GFLOP/s with fused multiply-add operations. This means the total peak throughput of the full 2000-core HammerBlade is 4–8 TFLOP/s with an impressive area normalized throughput of 50–100 GFLOP/s/mm^2. Scaling the target manycore architecture to 10,000+ cores is certainly feasible, although studying the performance implications of such scale-up manycore architectures is left for future work.

2.2 HammerBlade Manycore Software

As is the case with similar architectures, programming HammerBlade without loss of domain generality requires use of a low-level C runtime environment. Concerns such as data placement, synchronization, and load-balancing are left entirely to the programmers, and this demands both an extensive domain knowledge for their application and for the underlying hardware from them.

2.2.1 CUDA-lite

The HammerBlade manycore low-level C runtime, CUDA-lite, adopts a data-parallel programming model similar to CUDA with support for *thread groups* analogous to a thread block in the CUDA programming model. Like CUDA, CUDA-lite focuses on static parallel loops and assumes

an offloading execution model, in which the host CPU configurates the device, allocates device memory, copies input data from host to device, launches the kernel, and copies results back from device to host once the device finishes execution. Once a device kernel is launched, execution on the host CPU is blocked until the kernel returns. Unlike CUDA, CUDA-lite does not support context-switching and keeps a one-to-one mapping between threads and cores.

A critical difference between manycore thread groups and CUDA thread blocks is how they are mapped onto hardware. Thread groups are defined as a rectangle with dimensions specified by the programmer. Unlike CUDA thread blocks, whose multidimensionality is only a software abstraction, manycore thread group dimensions map to a set of cores that are physically arranged in the specified geometry with respect to the OCN. The target manycore runtime thereby exposes physical locality of compute resources in its programming model. Cores can communicate through the use of direct remote scratchpad access for fine-grained synchronization and sharing. This allows software programmers to arrange thread groups in a manner that is most advantageous to the memory access and communication patterns of the workload.

Writing applications for the HammerBlade manycore architecture can be challenging. Programmers who are new to the platform often struggle with both its unfamiliar programming/memory models and the hardware details they need to be aware of. One example would be the way HammerBlade and CUDA-lite utilizes the core-local SPM. By default, HammerBlade/CUDA-lite allocates `.sdata`, `.sbss`, and the stack in SPM. Doing so creates two aspects that a programmer should be aware of: (1) variables in application code that are declared as global (i.e., located in either `.sdata` or `.sbss`) are, actually, not global but thread local variables. Each core will get their unique copy in their SPM, unless the variable is explicitly marked as DRAM allocated with `__attribute__ ((section (".dram")))`. This is a common pitfall as many programmers implement inter-core communication with global variables. And (2) it is difficult to determine the stack space size ahead-of-time, and it is easy to run into stack overflow. As we have mentioned in Section 2.1, each core has a 4KB SPM, which is small by the standard of modern applications. When the stack and user defined buffers are sharing the 4KB SPM with `.sdata` and `.sbss`, programmers must minimize their stack usage to prevent potential stack overflow. To make things even worse, programmers are usually unable to determine the available stack space ahead-of-time without looking at the disassembly of their compiled programs.

2.2.2 Running Example

Figure 2.2 illustrates the target CPU-manycore heterogeneous system software in more detail. The host function (Figure 2.2 (a)) configures the manycore co-processor (lines 6–9), allocates memory on the device (lines 11–17), copies data from the host to device (lines 20), launches execution on the manycore co-processor (lines 22–27), and copies data back to the host (line 36–38) when the co-processor has finished execution. In the device function (Figure 2.2 (b)), each core accumulates non-overlapping ranges of the input array into partial sums (lines 12–17) and stores the partial sums into `core_0`'s scratchpad through direct remote scratchpad access (lines 18–22). Then `core_0` further accumulates these partial sums to yield the final result (lines 27–32). Variables with a `__` prefix (i.e., `__group_x`, `__group_y`, and `__core_id`) are defined by the manycore software runtime.

Lines 18–22 in Figure 2.2 (b) demonstrate the common way to conduct remote scratchpad access. To access a peer's scratchpad memory, we need to have a pointer to the data we would like to access. In this parallel reduce example, all cores write their partial results into the `buf` allocated in `core_0`'s SPM, which means every core needs to have a pointer to it. One approach is to have `core_0` communicate this pointer to other cores through DRAM. Another approach is to have every core calculate this pointer through a local pointer (i.e., a pointer to a variable on the core's own scratchpad). This is the approach we take in this example. The idea is to let every core have exactly the same memory layout in their SPMs. This is why even though only the `buf` in `core_0`'s SPM is used, it is allocated on all 128 cores (lines 8 in Figure 2.2 (b)). Then we can use the X and Y coordinates of the remote core (i.e., `(0,0)` in this case for `core_0`) and the corresponding local pointer (i.e., `buf` in line 19), to calculate a pointer to the `buf` in `core_0`'s SPM (i.e., `remote_buf`).

2.3 Evaluation Methodology

In this section, I briefly introduce the HammerBlade manycore architecture RTL simulation and energy modeling methodologies.

```c
float acc_host(float* A, uint32_t N) {
  // configurate thread group
  mc_dim_t tg_dim   = {.x = 1, .y = 1};
  mc_dim_t grid_dim = {.x = 16, .y = 8};

  // configurate HB device
  mc_dev_t dev;
  mc_dev_init(&dev);
  mc_dev_program_init(&dev);

  // allocate memory on device
  uint32_t wordsz = sizeof(float);
  uint32_t nbytes = N * wordsz;
  eva_t A_dev
  eva_t B_dev;
  mc_dev_malloc(&dev, nbytes, &A_dev);
  mc_dev_malloc(&dev, wordsz, &B_dev);

  // copy data from host to device
  mc_dev_memcpy(&dev, A_dev, A, nbytes);

  // launch kernel on device
  uint32_t mc_argv[3] = {A_dev, B_dev, N};
  mc_kernel_enqueue(&dev, grid_dim,
                    tg_dim, "acc_dev",
                    mc_argv, 3);
  mc_dev_tile_groups_execute(&dev);

  // copy data from device to host
  float B;
  mc_dev_memcpy(&dev,
      &B, B_dev, wordsz);

  return B;
}
```

(a) Host Code

```c
int acc_dev(float* A, float* B, uint32_t N) {
  // index calculation
  uint32_t ncores = (__group_x) * (__group_y);
  uint32_t M  = N / ncores;
  uint32_t s  = __core_id * M;

  // buffer for final reduction
  float buf[ncores];

  // local partial reduction
  float partial = 0.0f;
  for (uint32_t i = s; i < s+M; i++) {
    if (i < N) {
      partial += A[i];
    }
  }

  // get remote pointer of buf on core_0
  float* remote_buf = mc_remote_ptr(0, 0, buf);

  // remote scratchpad access
  remote_buf[__core_id] = partial;

  // synchronizaiton
  mc_barrier();

  // final reduction by core_0
  float acc = 0.0f;
  if (__core_id == 0) {
    for (uint32_t i = 0; i < ncores; i++)
      acc += buf[i];
  }
  *B = acc;

  // end of kernel synchronization
  mc_barrier();

  return 0;
}
```

(b) Device Code

Figure 2.2: CUDA-lite Parallel Reduce Example – This example demonstrates in-strachpad parallel accumulation: each core computes a partial sum of input array and stores partial sum into core_0's scratchpad. Then core_0 further accumulates partial sums to produce final result.

2.3.1 RTL Simulation

The performance results in this thesis are produced using cycle-accurate and RTL simulation that is co-simulated with native execution of applications. We directly simulate the 128-core HammerBlade system using tapeout-verified RTL that is co-simulated with the application software running on the host. RTL simulation is controlled by the CUDA-lite runtime library through the SystemVerilog Direct Programming Interface (DPI). This interface can emulate a tightly coupled system-on-chip host, like a BlackParrot [PGW+20] RISC-V SoC, or an inter-system connection like PCIe. This framework executes the host code natively while the CUDA-lite device code is executed in RTL.

The HammerBlade system can be simulated using commercial simulators like Synopsys VCS, or Verilator, an open-source simulator. The two simulators provide equivalent features and similar execution speeds. We used detailed statistics from these frameworks to analyze performance, and CAD tools to measure the impact of new features on performance, energy, and area. Since the HammerBlade manycore architecture is entirely open source and does not depend on any closed-source or licensed IP, this means that the entire system can be simulated by any interested users.

HammerBlade co-simulation uses DRAMSim3 [LYR+20, RCBJ11], a timing accurate simulator for modeling DRAM to model the performance of the memory system. Cycle-accurate RTL simulations of DRAM slow simulation speed down by orders of magnitude and therefore drastically increase iteration time. DRAMSim3 is an empirically validated [RCBJ11], academically accepted, open-source, C++ simulator for DRAM. By using C++, DRAMSim3 avoids the issues caused by direct RTL simulation of DRAM while providing more flexibility and introspection. In addition to modeling command timing, DRAMSim3 also measures the dynamic and static power of DRAM chips. This information is used to optimize the efficiency of applications on the HammerBlade manycore architecture.

RTL simulation has multiple methods for instrumentation: non-invasive profiling, tracing (debugging), and switching activity logging. The non-invasive profiling uses non-invasive SystemVerilog bind modules to instantiate non-synthesizable code without modifying the tapeout-ready RTL. These modules count events that occur in different parts of the architecture, for example, stalls and instructions executed in the HammerBlade manycore, congestion and backpressure in the network, hits and misses in the cache, and commands to the memory system. After an RTL simulation com-

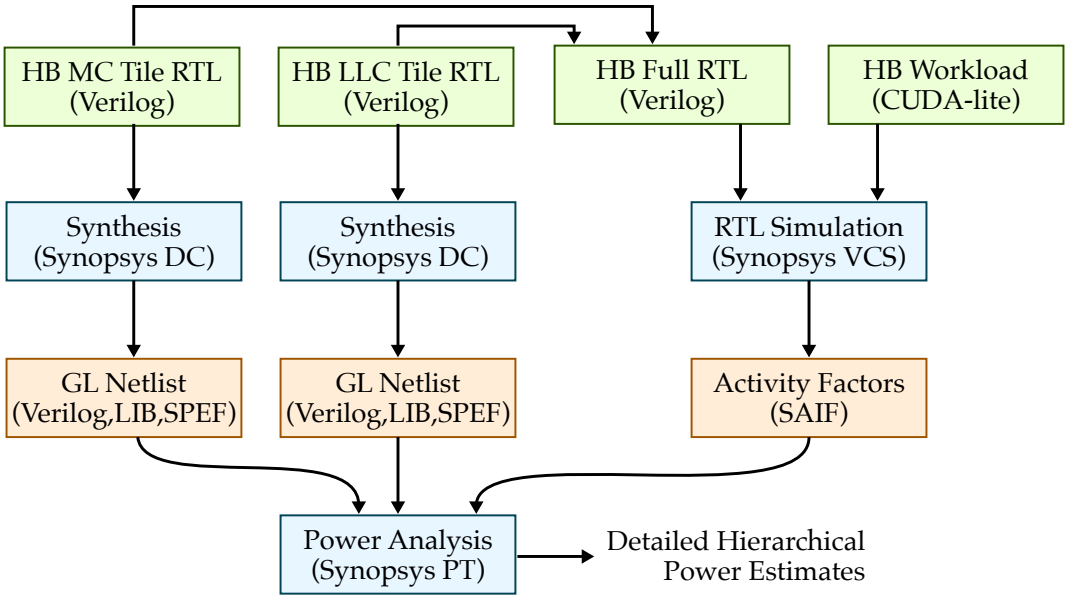


Figure 2.3: HammerBlade Energy Modeling Methodology – Workloads are run on RTL simulation of a small scale 128-core HammerBlade manycore using Synopsys VCS to produce activity factors. Individual cores and LLC banks are pushed through a standard-cell synthesis flow using Synopsys Design Compiler (DC) to generate gate-level (GL) netlists including Verilog, liberty files, and SPEF files to capture interconnect capactitance. The GL netlist and activity factors are input into Synopsys PrimeTime (PT) for power analysis to generate detailed hierarchical power estimates.

pletes these modules produce a table that is parsed and analyzed to produce statistics about regions of interest within the code.

2.3.2 Energy Modeling

We improve our RTL simulation flow to generate activity factors in the industry-standard SAIF format. These activity factors capture the number of toggles on every net in the entire RTL model for each kernel in a specific workload. We then take the various blocks in the HammerBlade system and push them individually through synthesis using Synopsys Design Compiler (DC). For example, we push the RTL for a single HammerBlade manycore core and a single LLC bank through the synthesis flow. This produces detailed gate-level netlists for each of these components. A gate-level netlist actually includes many different views including: a Verilog representation of the gate-level connectivity; the Liberty files which capture the input gate capacitance, internal dynamic power, and leakage power for each standard cell; and a SPEF file that captures the interconnect capacitance throughout the design. We can then combine these gate-level netlists and the activity factors for power analysis using Synopsys PrimeTime (PT) to generate detailed hierarchical power

estimates. Figure 2.3 illustrates the proposed energy modeling methodology, which is using the GF 12/14 nm technology node, a commercial standard cell library, and a commercial SRAM memory compiler.

This kind of detailed energy analysis for full workloads across an entire 128-core HammerBlade chip can be extremely time consuming, so we use a combination of temporal and spatial sampling. For kernels that have extremely long runtimes, we use a portion of the execution trace to generate a truncated SAIF file which can then be used to estimate the energy of the entire execution. For almost all kernels, we use the same gate-level netlist of the HammerBlade manycore core and LLC bank to do energy analysis for many different *instances* of that component throughout the design. For example, we spatially sample about 12% of the cores and LLC banks to determine the energy and then project the energy of the entire HammerBlade manycore. We have conducted time consuming full system analysis to help validate that this approach produces little error.

Our energy modeling methodology accurately captures leakage power by leveraging the standard cell and SRAM Liberty models. Our energy modeling methodology also includes clock power, and we have carefully calibrated the tools to ensure that our clock power estimates are reasonable without the need for clock-tree synthesis during place-and-route. We assume 1 pJ/b for the energy of the on-chip memory controller. For the host power, we assume our HammerBlade system will include BlackParrot RISC-V cores to execute the host code. RISC-V systems have been shown to have very high performance with good energy efficiency on this kind of host code. We accurately measure the host time during co-simulation and then assume a constant 1W of power consumed by the host processor.

We use a a set of *energy microbenchmarks* to characterize the energy required for a variety of RISC-V instructions on a HammerBlade manycore core. Each energy microbenchmark consists of 100 instructions carefully crafted in assembly along with special instructions to precisely start and stop performance and energy profiling at the beginning and end of the 100-instruction sequence (see Figure 2.4). We can then use the energy modeling methodology described above and divide by 100 to estimate the energy required for each type of instruction. We can also explore how much energy in the HammerBlade manycore core is consumed by the register file, instruction cache (I-Cache), scratchpad data memory (SPM), integer arithmetic logic unit (ALU), floating point unit (FPU), mesh network, clock tree, L2 cache, and the HBM memory controller. The other category includes pipeline registers, control logic, and other miscellaneous logic. We can see that

```
extern "C" __attribute__ ((noinline))
int kernel_energy_fadd() {
  //------------------------------------
  // Calling Convention Prologue
  //------------------------------------
  __asm__ __volatile__ (
    "addi sp, sp, -48;"
    "sw   s0, 44(sp);"
    ...
    "sw   s11, 0(sp);"
  );
  __asm__ __volatile__ (
    "lui t0,0xe39c;"
    ...
    "lui t6,0x7c8e3;"
    "fcvt.s.w ft0,t0;"
    ...
    "fcvt.s.w fs10,t6;"
  );
  //------------------------------------
  mc_saif_start();
  //------------------------------------
  // 100 back to back adds
  __asm__ __volatile__ (
    "fadd.s ft0, ft1, ft2;"
    "fadd.s ft1, ft2, ft3;"
    "fadd.s ft2, ft3, ft4;"
    "fadd.s ft3, ft4, ft5;"
    ...
    "fadd.s fa6, fa7, fs2;"
    "fadd.s fa7, fs2, fs3;"
    "fadd.s fs2, fs3, fs4;"
    "fadd.s fs3, fs4, fs5;"
  );
  //------------------------------------
  mc_saif_end();
  //------------------------------------
  // Calling Convention Epilogue
  //------------------------------------
  __asm__ __volatile__ (
    "lw   s0, 44(sp);"
    ...
    "lw   s11, 0(sp);"
    "addi sp, sp, 48;"
  );

  mc_barrier();
  return 0;
}
```

(a) fadd energy microbenchmark.

```
extern "C" __attribute__ ((noinline))
int kernel_energy_branch() {
  //------------------------------------
  // Calling Convention Prologue
  //------------------------------------
  __asm__ __volatile__ (
    "addi sp, sp, -48;"
    "sw   s0, 44(sp);"
    ...
    "sw   s11, 0(sp);"
  );
  __asm__ __volatile__ (
    "li t0, 0xb08aa953;"
    ...
    "li t6, 0x0198e2f3;"
  );
  //------------------------------------
  mc_saif_start();
  //------------------------------------
  // 100 back to back adds
  __asm__ __volatile__ (
    "m0: beq t0, t0, m99;"
    "m1: beq t1, t1, m98;"
    ...
    "m49: beq t6, t6, m50;"
    "m50: beq t0, t0, ms;"
    "m51: beq t1, t1, m49;"
    ...
    "m98: beq t5, t5, m2;"
    "m99: beq t6, t6, m1;"
    "ms:  nop;"
  );
  //------------------------------------
  mc_saif_end();
  //------------------------------------
  // Calling Convention Epilogue
  //------------------------------------
  __asm__ __volatile__ (
    "lw   s0, 44(sp);"
    ...
    "lw   s11, 0(sp);"
    "addi sp, sp, 48;"
  );

  mc_barrier();
  return 0;
}
```

(b) Taken branch energy microbenchmark.

Figure 2.4: Energy Microbenchmark Examples Each energy microbenchmark is written directly in inline assembly and consists of 100 instructions carefully crafted in assembly along with special instructions to precisely start and stop performance and energy profiling.

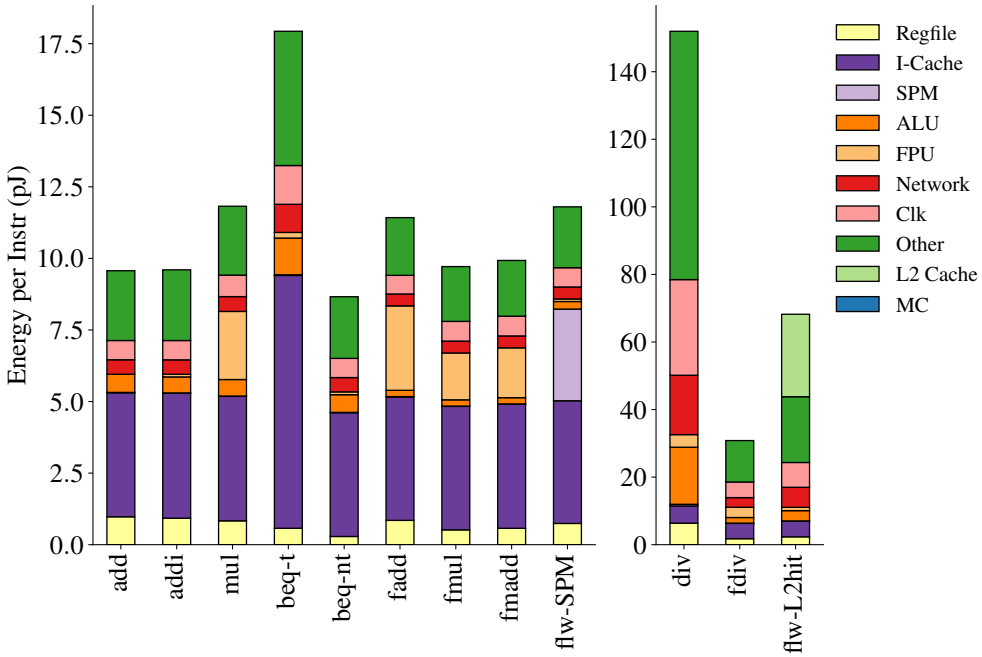


Figure 2.5: HammerBlade Per-Instruction Energy Breakdown – microbenchmarks were run on HammerBlade manycore cores to explore the energy breakdown across various components: register file (Regfile), instruction cache (I-Cache), scratchpad memory (SPM), integer arithmetic logic unit (ALU), floating-point unit (FPU), mesh network, clock tree, L2 cache, and memory controllers (MC). The other category includes pipeline registers, control logic, and other miscellaneous logic. All energy results include leakage and clock power.

on average, arithmetic instructions consume less than 10 pJ. Floating-point operations consume more energy in the FPU but less energy in the control logic of the processor pipeline. Instructions consume significant energy in the instruction cache owing to the standard von Neumann paradigm. Results are summarized in Figure 2.5. One important takeaway from this plot is that, loading data from higher level of the memory hierarchy consumes much more energy than loading data from SPM (see bars *flw-SPM* and *flw-L2hit* in Figure 2.5). This implies that keeping data in SPM is critical to achieve not only high performance but also high energy efficiency.

2.4 Case Study: Optimizing the Matrix Multiplication Kernel

In this section, we use matrix multiplication as a case study to demonstrate the process of hand optimizing a kernel on the HammerBlade manycore architecture. Matrix multiplication (MatMul) is a key kernel in the center of applications from various domains, such as image processing and deep learning. As we have mentioned in Section 2.2, hand tuning a kernel puts high demands on

```
1  void MatMul(float* A, float* B, float* C) {
2    for (uint32_t i = 0; i < N; i++) {
3      for (uint32_t j = 0; j < N; i++) {
4        for (uint32_t k = 0; k < N; i++) {
5          C[i][j] = C[i][j] + A[i][k] * B[k][j];
6        }
7      }
8    }
9  }
```

Figure 2.6: Three Nested For-Loops of MatMul – The computation of MatMul can be represented as three nested for-loops.

the programmers, who must have a deep understanding of both the kernel they are optimizing and the underlying hardware they are targeting. Optimizing the MatMul kernel involves data movement management, SPM space management, DRAM access pattern improvement, unrolling, and instruction arrangement.

2.4.1 Naïve MatMul

Figure 2.6 illustrates the well-known definition of MatMul, which can be represented as three nested for-loops. Without loss of generality, we assume both input matrices are square matrices that have the same size. The computation is straightforward: take one row `i` from matrix `A` and one column `j` from matrix `B`. Then the element `(i,j)` in the output matrix `C` is yielded by accumulating the products of an element-wise multiplication of row `i` and column `j`. Having a simple kernel like MatMul run functionally correct on the HammerBlade manycore is relatively simple, as long as we do not explicitly utilize the scratchpad memories. We can implement a naïve MatMul kernel on HammerBlade by making a few minor tweaks to the three nested for-loops shown in Figure 2.6. Figure 2.7 and Figure 2.8 show the host and device code, respectively. While this naïve version is functionally correct, it does not achieve high performance. In this section, we will add optimizations to this navie MatMul implementation and by the end of the section, arrive at a highly optimized MatMul kernel. Note that the host code stays the same as we add optimizations to the device code.

There are generally two ways to improve the performance of the MatMul kernel: improving arithmetic intensity and reducing control flow overhead. Both are important on HammerBlade. Memory bandwidth, especially the LLC bandwidth, is a key limiting factor for applications running

```
void matmul_host(float* A, float* B, float* C, uint32_t N) {
  // configurate thread group
  mc_dim_t tg_dim   = {.x = 1,  .y = 1};
  mc_dim_t grid_dim = {.x = 16, .y = 8};

  // configurate HB device
  mc_dev_t dev;
  mc_dev_init(&dev);
  mc_dev_program_init(&dev);

  // allocate memory on device
  uint32_t nbytes = N * N * sizeof(float);
  eva_t A_dev
  eva_t B_dev;
  eva_t C_dev;
  mc_dev_malloc(&dev, nbytes, &A_dev);
  mc_dev_malloc(&dev, nbytes, &B_dev);
  mc_dev_malloc(&dev, nbytes, &C_dev);

  // copy data from host to device
  mc_dev_memcpy(&dev, A_dev, A, nbytes);
  mc_dev_memcpy(&dev, B_dev, B, nbytes);

  // launch kernel on device
  uint32_t mc_argv[4] = {A_dev, B_dev, C_dev, N};
  mc_kernel_enqueue(&dev, grid_dim, tg_dim,
                    "acc_dev", mc_argv, 4);
  mc_dev_tile_groups_execute(&dev);

  // copy data from device to host
  mc_dev_memcpy(&dev, &C, C_dev, nbytes);

  return;
}
```

Figure 2.7: MatMul Kernel Host Code – the host code is the same for all device versions.

on HammerBlade. Recall that the HammerBlade manycore has 128 cores arranged into a 16×8 grid, and 32 LLC banks located at the top and bottom of the grid. In each cycle, one LLC bank can fulfill at most one request and respond with one word (4B), and the total LLC bandwidth is 128B per cycle, or 1B per core. Loading from DRAM incurs long latency even if the data is cached by the LLC due to contention in the LLC and/or congestion in the OCN. Thus, improving data reuse (i.e., reducing loads from DRAM and increasing arithmetic intensity) is key to achieve high performance. Reducing control flow overhead is also important. Each core in HammerBlade is a single-issue in-order processor, and thus every instruction that is not a fused-multiply-add (FMA) instruction limits peak performance. Besides these two general approaches, we can also exploit

```
1  int matmul_naive(float* A, float* B, float* C, uint32_t N) {
2    // parallelize outer loop
3    // core interleaved
4    for (uint32_t i = __core_id; i < N; i += ((__group_x) * (__group_y))) {
5      for (uint32_t j = 0; j < N; i++) {
6        float res = 0.0f;
7        for (uint32_t k = 0; k < N; i++) {
8          res += A[i * N + k] * B[k * N + j];
9        }
10       C[i * N + j] = res;
11     }
12   }
13
14   // end of kernel synchronization
15   mc_barrier();
16
17   return 0;
18 }
```

Figure 2.8: Naive MatMul Device Code

the non-speculative runahead execution to hide memory latency by having multiple DRAM loads in flight at the same time on the HammerBlade manycore.

2.4.2 Optimization 1: Tiling into SPM

The first optimization we can add is tiling. Unlike the naïve implementation which parallelizes the outer loop, and moves to another output element only after the current output element is fully computed, the tiled MatMul works on a block (i.e., multiple output elements in a square), and loads input matrices in blocks as well. A single input element can be used to compute multiple output elements in the block. By doing so, we can increase arithmetic intensity as we reduce the number of DRAM accesses. The tiled MatMul helper functions and device code are illustrated in Figure 2.9 and Figure 2.10, respectively. Without loss of generality, we assume the input matrices have input size `N` as a multiple of `BLOCK_DIM`. This a reasonable assumption as padding is a well known and widely adopted technique. Besides tiling, one should also note that we also adopt a 2-D spatial block distribution (see lines 13–14 in Figure 2.10) to have a more LLC friendly DRAM access pattern. Imagine the case where we have two sufficiently large input matrices, in which `num_blk` is larger than 128. In this case, a 1-D distribution scheme will touch 128 unique blocks of matrix `A` and 1 block of matrix `B`. However, the 2-D scheme shown in Figure 2.10 will have the HammerBlade grid access 16 blocks of matrix `A` and 8 blocks of matrix `B`. Thus, the 2-D

```
#define BLOCK_DIM 16

void dram_to_spm_naive(float* dst, float* src, uint32_t r_idx,
                       uint32_t c_idx, uint32_t N) {
  float* src_base = src + r_idx * BLOCK_DIM * N + c_idx * BLOCK_DIM;
  for (uint32_t i = 0; i < BLOCK_DIM; i++) {
    for (uint32_t j = 0; j < BLOCK_DIM; j++) {
      dst[i * BLOCK_DIM + j] = src_base[j]
    }
    src_base += N;
  }
}

void compute_naive(float* result, float* sp_mat1, float* sp_mat2) {
  for (uint32_t iii = 0; iii < BLOCK_DIM; iii++) {
    for (uint32_t jjj = 0; jjj < BLOCK_DIM; jjj++) {
      float tmp = result[iii * BLOCK_DIM + jjj];
      for (uint32_t kkk = 0; kkk < BLOCK_DIM; kkk++) {
        tmp += sp_mat1[iii * BLOCK_DIM + kkk]
               * sp_mat2[kkk * BLOCK_DIM + jjj];
      }
      result[iii * BLOCK_DIM + jjj] = tmp;
    }
  }
}
```

Figure 2.9: Tiled MatMul Helper Functions – only showes `dram_to_spm_naive`; `spm_to_dram` is implemented similarly.

scheme both increases data reuse in the LLC by accessing fewer unique blocks (i.e., 24 v.s. 129), and eliminates LLC hot spots by avoiding all cores accessing the same block of matrix `B`. We choose to use a 16×16 block size based on SPM capacity. The three SPM buffers consume in total 3KB SPM space leaving 1KB for global variables and the stack.

2.4.3 Optimization 2: Tiling into Registers

The tiled MatMul we just introduced reduces DRAM accesses. Similarly, we would like to reduce the number of accesses to the SPM buffer. While the SPM has a much lower latency, recall that ideally we would like to (though will never be able to) eliminate every instruction that is not an FMA. The second optimization we add is called tiling into registers: we keep as much data as possible in the registers to eliminate SPM loads/stores. Figure 2.11 illustrates the new `compute` helper function. The tiling into registers scheme uses a 4×4 sub-block size and fully unrolls the inner compute loop to both allow intermediate values to be kept in the register file and reduce

```
1  #define BLOCK_DIM 16
2
3  int matmul_tiled(float* A, float* B, float* C, uint32_t N) {
4
5    // calculate number of blocks
6    uint32_t num_blk = (N + BLOCK_DIM - 1) / BLOCK_DIM;
7
8    // create buffer in SPM
9    float sp_mat1[BLOCK_DIM * BLOCK_DIM];
10   float sp_mat2[BLOCK_DIM * BLOCK_DIM];
11   float sp_result[BLOCK_DIM * BLOCK_DIM];
12
13   for (uint32_t rr = __core_y; rr < num_blk; rr += __group_y) {
14     for (uint32_t rc = __core_x; rc < num_blk; rc += __group_x) {
15
16       // initialize scratchpad result (init to 0's)
17       reset_sp(sp_result);
18
19       // process blocks
20       for (uint32_t mid = 0; mid < num_blk; mid++) {
21         dram_to_spm_naive(sp_mat1, A, rr, mid, N);
22         dram_to_spm_naive(sp_mat2, B, mid, rc, N);
23         compute_naive(sp_result, sp_mat1, sp_mat2);
24       }
25
26       // copy this block back into DRAM
27       spm_to_dram(C, sp_result, rr, rc);
28     }
29   }
30
31   // end of kernel synchronization
32   mc_barrier();
33
34   return 0;
35 }
```

Figure 2.10: Tiled MatMul Device Code

control flow overhead. Note that lines 6–11, lines 15–24, and lines 43–47 are crafted carefully so that the compiler will generate load and store instructions using register offset addressing with the same base address register, which yields fewer dynamic instructions than having load instructions that each read a different base address register.

2.4.4 Optimization 3: Copying with Non-Spectualive Runahead Execution

The last optimization we applied to the MatMul kernel is to unroll the copying-to-SPM helper function and rearrange instructions to leverage non-speculative runahead execution. The new

```
#define BLOCK_DIM 16

void compute(float* result, float* sp_mat1, float* sp_mat2) {
  for (int iii = 0; iii < BLOCK_DIM; iii += 4) {
    for(int jjj = 0; jjj < BLOCK_DIM; jjj += 4) {
      int dest_base = iii * BLOCK_DIM + jjj;
      register float res00 = dest[dest_base + 0             + 0];
      register float res03 = dest[dest_base + 0             + 3];
      ...
      register float res32 = dest[dest_base + 3 * BLOCK_DIM + 2];
      register float res33 = dest[dest_base + 3 * BLOCK_DIM + 3];
      for(int kkk = 0; kkk < BLOCK_DIM; kkk++) {
        // for iiii in 0...4
        //   for jjjj in 0...4
        int mat1_base = kkk + iii * BLOCK_DIM;
        register float mat1_0 = sp_mat1[mat1_base + 0];
        register float mat1_1 = sp_mat1[mat1_base + BLOCK_DIM];
        register float mat1_2 = sp_mat1[mat1_base + 2 * BLOCK_DIM];
        register float mat1_3 = sp_mat1[mat1_base + 3 * BLOCK_DIM];
        int mat2_base = kkk * BLOCK_DIM + jjj;
        register float mat2_0 = sp_mat2[mat2_base + 0];
        register float mat2_1 = sp_mat2[mat2_base + 1];
        register float mat2_2 = sp_mat2[mat2_base + 2];
        register float mat2_3 = sp_mat2[mat2_base + 3];
        // compute
        res00 += mat1_0 * mat2_0;
        res01 += mat1_0 * mat2_1;
        res02 += mat1_0 * mat2_2;
        res03 += mat1_0 * mat2_3;
        res10 += mat1_1 * mat2_0;
        res11 += mat1_1 * mat2_1;
        res12 += mat1_1 * mat2_2;
        res13 += mat1_1 * mat2_3;
        res20 += mat1_2 * mat2_0;
        res21 += mat1_2 * mat2_1;
        res22 += mat1_2 * mat2_2;
        res23 += mat1_2 * mat2_3;
        res30 += mat1_3 * mat2_0;
        res31 += mat1_3 * mat2_1;
        res32 += mat1_3 * mat2_2;
        res33 += mat1_3 * mat2_3;
      }
      dest[dest_base + 0             + 0] = res00;
      dest[dest_base + 0             + 1] = res01;
      ...
      dest[dest_base + 3 * BLOCK_DIM + 2] = res32;
      dest[dest_base + 3 * BLOCK_DIM + 3] = res33;
    }
  }
}
```

Figure 2.11: Tiling into Register – Similar to the idea of tiling MatMul, we reduce SPM accesses by keeping input values and partial results in the register file.

```
#define BLOCK_DIM 16

void dram_to_spm(float* dst, float* src, uint32_t r_idx,
                 uint32_t c_idx, uint32_t N) {
  float* src_base = src + r_idx * BLOCK_DIM * N + c_idx * BLOCK_DIM;
  for (uint32_t i = 0; i < BLOCK_DIM; i++) {
    // fully unroll for (uint32_t j = 0; j < BLOCK_DIM; j++)
    register float tmp0  = src_base[0];
    register float tmp1  = src_base[1];
    register float tmp2  = src_base[2];
    register float tmp3  = src_base[3];
    register float tmp4  = src_base[4];
    register float tmp5  = src_base[5];
    register float tmp6  = src_base[6];
    register float tmp7  = src_base[7];
    register float tmp10 = src_base[8];
    register float tmp11 = src_base[9];
    register float tmp12 = src_base[10];
    register float tmp13 = src_base[11];
    register float tmp14 = src_base[12];
    register float tmp15 = src_base[13];
    register float tmp16 = src_base[14];
    register float tmp17 = src_base[15];
    // prevent compiler from rearranging loads and stores
    asm volatile("": : :"memory");
    dst[0]  = tmp0;
    dst[1]  = tmp1;
    dst[2]  = tmp2;
    dst[3]  = tmp3;
    dst[4]  = tmp4;
    dst[5]  = tmp5;
    dst[6]  = tmp6;
    dst[7]  = tmp7;
    dst[8]  = tmp10;
    dst[9]  = tmp11;
    dst[10] = tmp12;
    dst[11] = tmp13;
    dst[12] = tmp14;
    dst[13] = tmp15;
    dst[14] = tmp16;
    dst[15] = tmp17;
    // bump src pointer to next row
    src_base += N;
  }
}
```

Figure 2.12: Leveraging Non-Speculative Runahead Execution – We unroll the copy to SPM helper function for both overhead reduction and leveraging non-speculative runhead execution and control flow.

helper function `dram_to_spm` is illustrated in Figure 2.12. We first fully unrolled the inner loop (i.e., `unroll for (int j = 0; j < BLOCK_DIM; j++)`) and then micro-managed the instructions. Namely, we separated DRAM loads and SPM writes with a compiler fence (i.e., `asm volatile("": : :"memory");`) which prevents the compiler from rearranging instructions before and after this fence. The goal is to exploit the non-speculative runahead execution mechanism. Recall that this feature allows a core to continue executing subsequent independent instructions even if there are pending loads. By placing load instructions before any of the store instructions, we allow the core to issue all 16 loads before it is stalled for load-use dependencies (i.e., stores in this case). Unrolling the inner loop helps with both memory latency hiding (i.e., through having multiple concurrent inflight DRAM loads) and control flow overhead reduction.

2.4.5 Performance Evaluation

We conducted micro-benchmarking of the MatMul kernel with 256×256 input matrices. We ran four versions of the MatMul kernel: (1) *Naive MatMul* (see Figure 2.8), (2) *Tiled MatMul* (see Figure 2.10 and Figure 2.9), (3) *Tiled MatMul* with tiling into registers (see Figure 2.11), and (4) *Tiled MatMul* with both tiling into registers and copying-to-SPM with non-speculative runahead execution (see Figure 2.12). Their execution time breakdown is illustrated in Figure 2.13. From the figure we can observe that applying tiling (Section 2.4.2) is able to improve the performance of MatMul kernel by $3\times$. The benefit comes from replacing most of the DRAM load instructions (i.e., light yellow in Figure 2.13) with SPM loads (i.e., purple in the figure), in which doing so significantly reduces the number of stall cycles due to dependent DRAM load (i.e., yellow in the figure). However, the breakdown reveals that tiling introduces additional instructions and we observe an increase in total number of dynamic instructions when comparing *Tiled MatMul* to *Naïve MatMul*. Tiling into registers helps with reducing both SPM accesses and other instructions as it exploits reuse of input data in the register file. *Tiled MatMul* with both tiling into registers and copying with non-speculative runahead execution yields the best performance. The final hand optimized MatMul kernel can achieve $9\times$ speedup compared to *Naïve MatMul*. Non-speculative runahead execution can significantly reduce the number of stall cycles due to DRAM dependency as it allows multiple concurrent DRAM loads inflight. However, a new category of stall, *Stall_OCN* (green in the figure), appears when we apply this technique. This indicates OCN has become congested when cores are issuing multiple back to back DRAM requests.

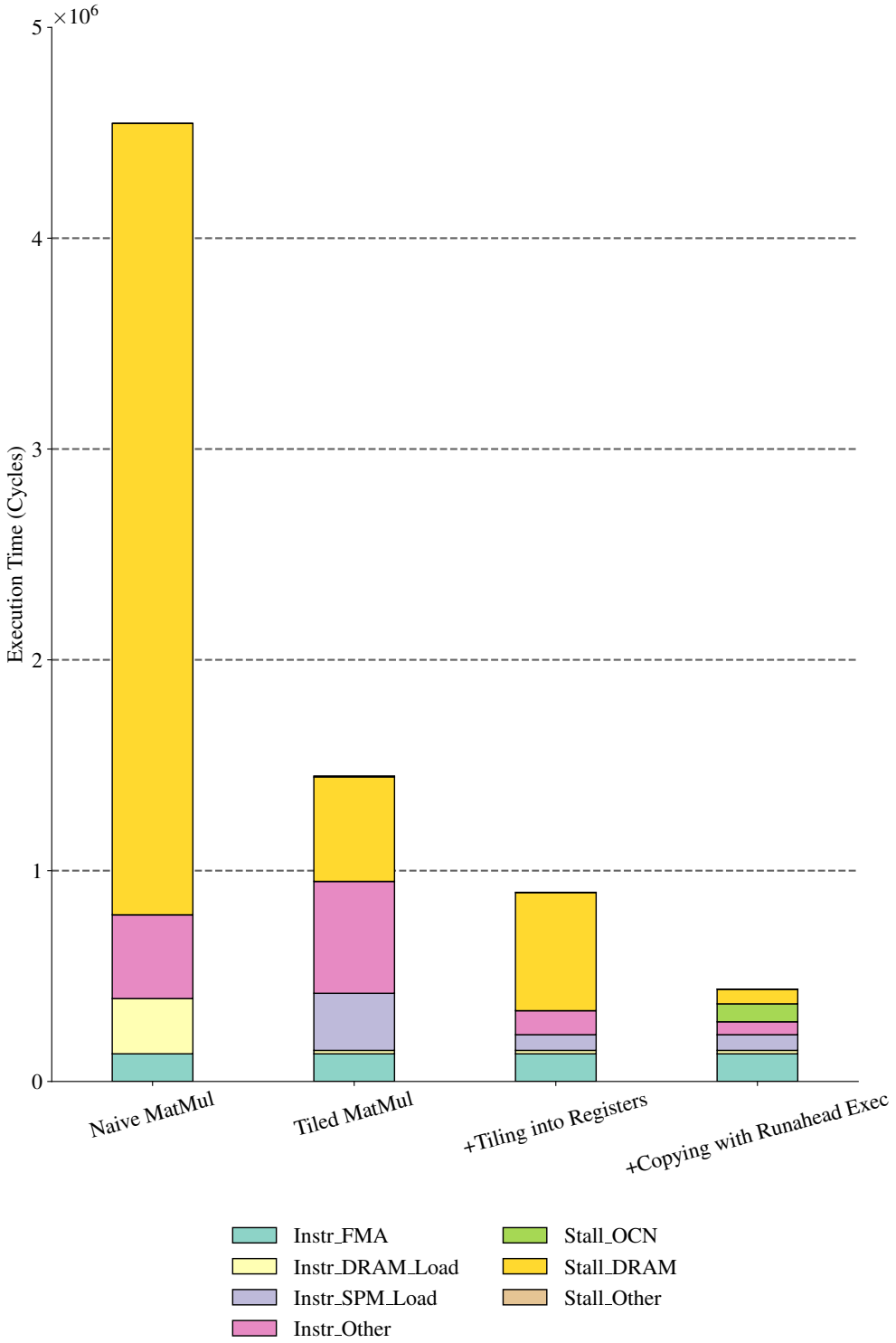


Figure 2.13: MatMul Execution Time Breakdown – Execution time breakdown of the four versions of MatMul we introduced in this section. Ran with 256×256 input matrices.

CHAPTER 3
HB-PYTORCH: A TENSOR PROCESSING FRAMEWORK FOR SPM MANYCORE ARCHITECTURES

As seen in Chapter 2, manycore architectures with software-managed scratchpad memories usually require programmers to express their applications in low-level C environments and/or directly in assembly, rely on them to explicitly manage data coherence among private caches/memories, and adopt a more restricted programming model. The unfamiliar programming model and the broad software design and optimization space are the key reasons why such architectures have not yet been widely accepted. One approach to resolve this programmability challenge of manycore architectures is to provide specialized or domain-specific frameworks. Such frameworks that provide ready-to-use hand-optimized operators embedded within a high-level language played an essential role in the adoption of GPGPUs.

In this chapter, I demonstrate the potential for a domain-specific programming framework approach to address the manycore programmability challenge by extending the PyTorch framework for both dense and sparse tensor processing on the HammerBlade manycore. Our extended PyTorch framework currently provides over 100 hand-optimized operators. Section 3.1 provides a detailed description of the tensor processing library which supports both dense and sparse tensor operations, and Section 3.2 evaluates three real-world workloads using the extended PyTorch tensor processing framework including: a dense residual neural network for computer vision, a dense deep-learning autoencoder-based recommender system for movie recommendations, and a sparse local graph clustering system based on an iterative shrinkage-thresholding algorithm for personalized page ranking.

3.1 A Tensor Processing Framework

PyTorch [PGM+19] is a widely adopted open-source tensor processing framework that provides an easy to use Python frontend for highly optimized tensor operators implemented in a low-level C++ ATen library [zde20]. In this section, we first present our tensor processing framework for CPU-manycore heterogeneous systems developed from PyTorch. We then evaluate and analyze a set of representative operators with micro-benchmarks on the target system to identify performance bottlenecks.

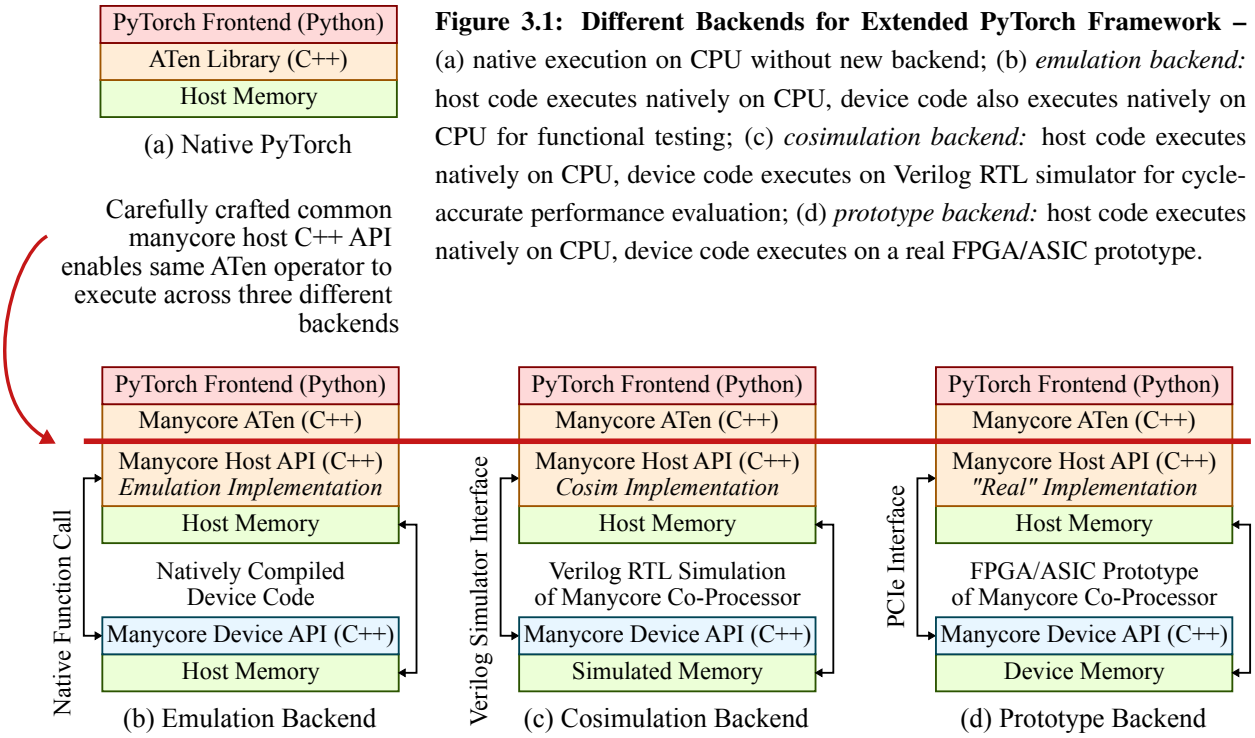


Figure 3.1: Different Backends for Extended PyTorch Framework – (a) native execution on CPU without new backend; (b) *emulation backend:* host code executes natively on CPU, device code also executes natively on CPU for functional testing; (c) *cosimulation backend:* host code executes natively on CPU, device code executes on Verilog RTL simulator for cycle-accurate performance evaluation; (d) *prototype backend:* host code executes natively on CPU, device code executes on a real FPGA/ASIC prototype.

3.1.1 PyTorch on CPU-Manycore Heterogeneous Systems

We extend PyTorch and build an open-source tensor processing framework for CPU-manycore heterogeneous systems to address the manycore programmability challenge. PyTorch's Python-level operators are platform agnostic; a dynamic dispatcher in ATen chooses the appropriate implementation for execution at runtime. The actual ATen operators can be either platform agnostic or platform specific. Platform specific implementations are grouped into *backends* (e.g., a CPU backend or a GPGPU backend). Platform agnostic operators are part of the CPU backend as well. New platforms can be easily supported by plugging new backends into ATen's dynamic dispatcher. We extend PyTorch with a new ATen backend to support both dense and sparse tensor processing on the target manycore co-processor. With our framework, tensor workloads can run exclusively on the CPU of the target heterogeneous system without any changes to the code. In this scenario, the CPU backend supports the framework's Python APIs and data is stored in CPU host memory (see Figure 3.1(a)). One can also choose to accelerate tensor workloads on the manycore co-processor with minimal changes to the existing code (see Figure 3.2(a)). Only changing three lines is necessary: one for migrating the neural network model to the manycore co-processor and two for migrating the input data and expected labels. PyTorch operators that are platform specific

```
class Autoencoder(nn.Module):
 def __init__(self):
  ...
  self.encoder = nn.Sequential(
    nn.ReLu(),
    nn.BatchNorm1d(800),
    nn.Dropout(0.5)
  )

  self.bneck = nn.Linear(800, 400)

  self.decoder = nn.Sequential(
    nn.ReLu(),
    nn.BatchNorm1d(400),
    nn.Dropout(0.5)
  )
  ...

 def forward(self, x):
  x = self.emb(x).sum(dim=1)
  x = self.encoder(x)
  x = self.bneck(x)
  x = self.decoder(x)
  x = self.output(x)

model = Autoencoder().manycore()
...
for x, y in dataloader_train:
 x = x.manycore()
 y = y.manycore()

 out = model(x)
 loss = F.MSELoss(out, y)

 opt.zero_grad()
 loss.backward()
 opt.step()
```

(a) Python Frontend

```
Tensor relu(const Tensor self) {
  Tensor res = opt_result.value_or(Tensor());
  auto iter =
    TensorIterator::binary_op(res, self, self);
  return at::threshold(iter,0,0);
}
```

(b) Platform Agnostic ATen Operator

```
void threshold_kernel_mc(TensorIterator& iter,
                         Scalar t, Scalar v) {
  AT_DISPATCH_FLOAT_TYPE_ONLY(iter.dtype(),
    "threshold_mc",
    [&]() {
      offload_op_binary(iter, t.to<scalar_t>(),
                        v.to<sclar_t>(),
                        "tensorlib_threshold");
    });
}
```

(c) Manycore Backend CPU Host Function

```
int tensorlib_threshold(mc_tensor_t* res_p,
                        mc_tensor_t* self_p,
                        float* threshold_p,
                        float* value_p) {
  MCTensor<float> res(res_p);
  MCTensor<float> self(self_p);
  float threshold = *threshold_p;
  float value     = *value_p;

  mc_tiled_foreach(res, self, [&] (float self_v) {
    return (self_v <= threshold) ? value : self_v;
  });

  mc_barrier ();
  return 0;
}
```

(d) Manycore Backend Device Function

Figure 3.2: Extended PyTorch Framework for CPU-Manycore Heterogeneous Systems – Blue lines 26, 29–30 in (a) are the only changes required to port an existing workload (e.g., training a deep neural network) written with PyTorch to run on the target CPU-manycore heterogeneous system. Red lines show the (simplified) dispatch chain for the PyTorch `ReLu` operator: Python frontend (a) dispatches to platform agnostic ATen operator (b), which dispatches to manycore backend CPU host function (c), which finally launches the manycore device function (d).

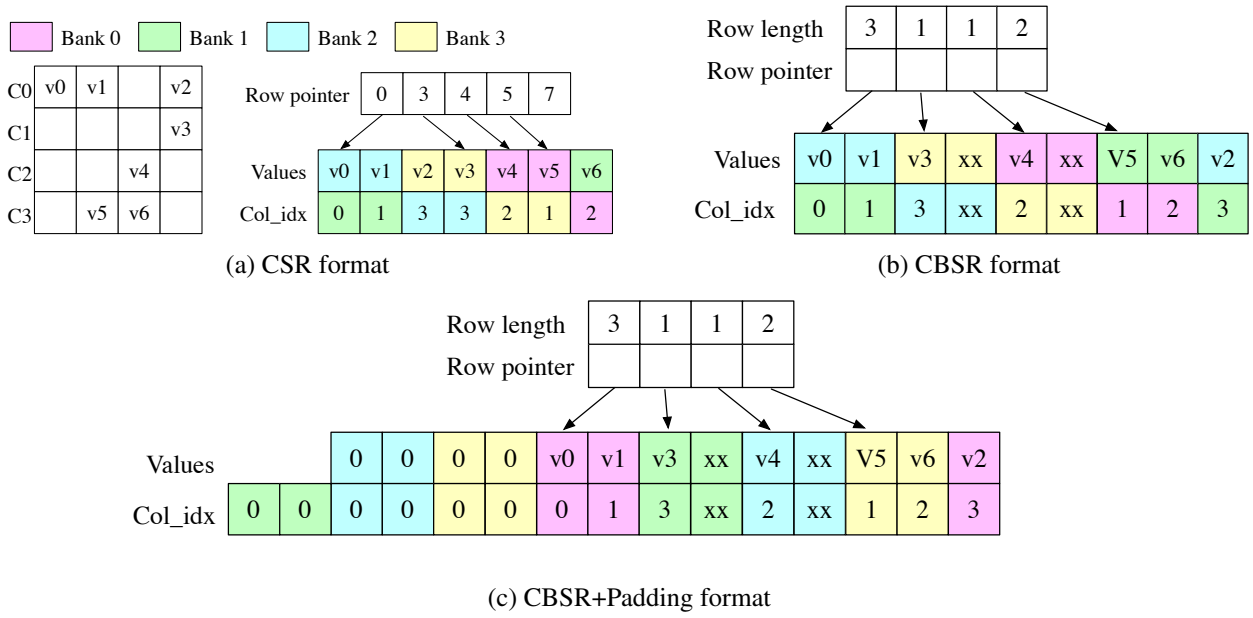


Figure 3.3: CSR and CBSR Sparse Tensor Formats

ATen Operator	**Description**	**PyTorch Operator**	**AI**	**Input**
MatMul	Matrix Multiplication	`mm`	High	$256 \times 256 \times 256$
Conv2D	2D Convolution	`convolution`	Medium	32×32 input w/ 16 channels, 16 3×3 Filters, 32 Images Batch
AddMV	Matrix-Vector Mult	`addmv`	Low	1024×128
SpMV	Sparse Matrix-Vector Mult	`mv`	Low	FB-Johns55, 5157×5157 sparse matrix, density 1.4%
Sum	Reduction	`sum`	Low	One Tensor w/ $192,000$ Elements
EmbBack	Backprop of Embedding	`Embedding`	Low	600×100 Embedding Table, 256 Records Batch, 50 Entries per Record
Add	Element-Wise Add	`torch.add`	Low	Two Tensors w/ $131,072$ Elements Each

Table 3.1: Operator Micro-Benchmarking – Inputs used in the operator micro-benchmarking. See Figure 3.4. AI = arithmetic intensity.

will be dispatched to the manycore backend, and data will be automatically migrated as needed (see Figure 3.1(d)).

An example workload using the proposed framework is shown in Figure 3.2. When PyTorch operator `nn.ReLu()` is used in Python code, its ATen counterpart `relu()` is called. In this case, `relu()` is platform agnostic (i.e., runs on the CPU), and is implemented by reusing a platform-specific ATen operator (i.e., `threshold()`). Since `model` in line 26 of Figure 3.2(a) is on the manycore co-processor, the call to `threshold()` in line 5 of Figure 3.2(b) is dispatched to the manycore implementation (Figure 3.2(c)), and compute is then offloaded to the manycore co-processor (Figure 3.2(d)).

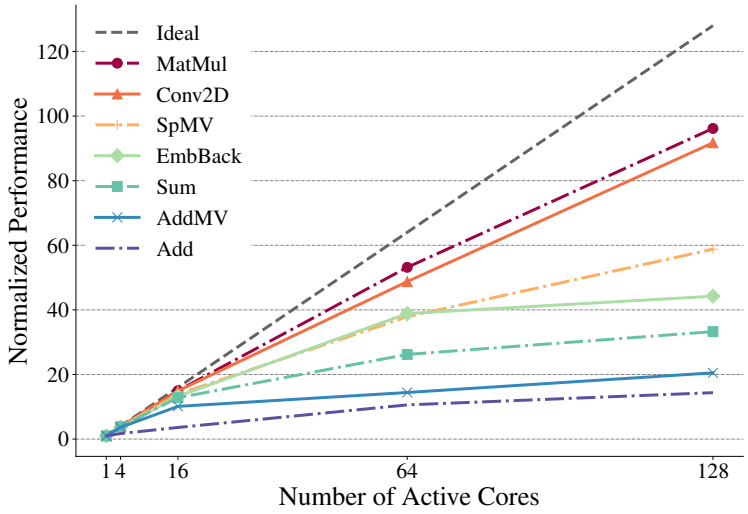


Figure 3.4: ATen Operator Micro-Benchmarking – Scalability of a representative set of ATen operators. See Table 3.1 for operator description and input sizes. Normalized to single core performance.

We have ported over 100 tensor operators including matrix multiplication, 2D convolution, most element-wise operators (e.g., add, subtract), reductions (e.g., sum, mean), and sparse operators (e.g., sparse matrix-vector multiplication). All operators are hand-tuned and aggressively optimized: scratchpad memory is utilized to enable data reuse and increase arithmetic intensity; stall-on-use is leveraged to exploit pipeline parallelism and hide memory latency; unrolling is used to balance instruction cache performance and loop overhead.

For sparse operators, prior work has shown that the layout of sparse tensors can significantly impact performance [FOS+14, SJS+20, SJL+20]. In our framework, we implement a novel *cyclic bank sparse row* (CBSR) tensor layout. CBSR is designed to reduce LLC bank conflicts and network congestion by ensuring cores only access LLC banks located in the same column. Figure 3.3 shows an example using traditional compressed sparse row (CSR), CBSR and CBSR+Padding formats for a 4×4 sparse matrix. In this simplified example, our architecture has one DRAM channel with four LLC banks. Each core only accesses one row of the sparse matrix. The data block size within each bank is two data elements and follows the cyclic memory partitioning scheme of [WLZ+13]. In CSR, the indices of non-zero values of different rows may fall into the same bank, which leads to memory bank conflicts when different cores access either column indices or values (i.e., C0 accesses v2 and C1 accesses v3). Using CBSR can eliminate the memory bank conflict between cores when accessing either indices or values, but memory conflicts still remain when one core is accessing the indices and the other core is accessing the values (i.e., C0 is ac-

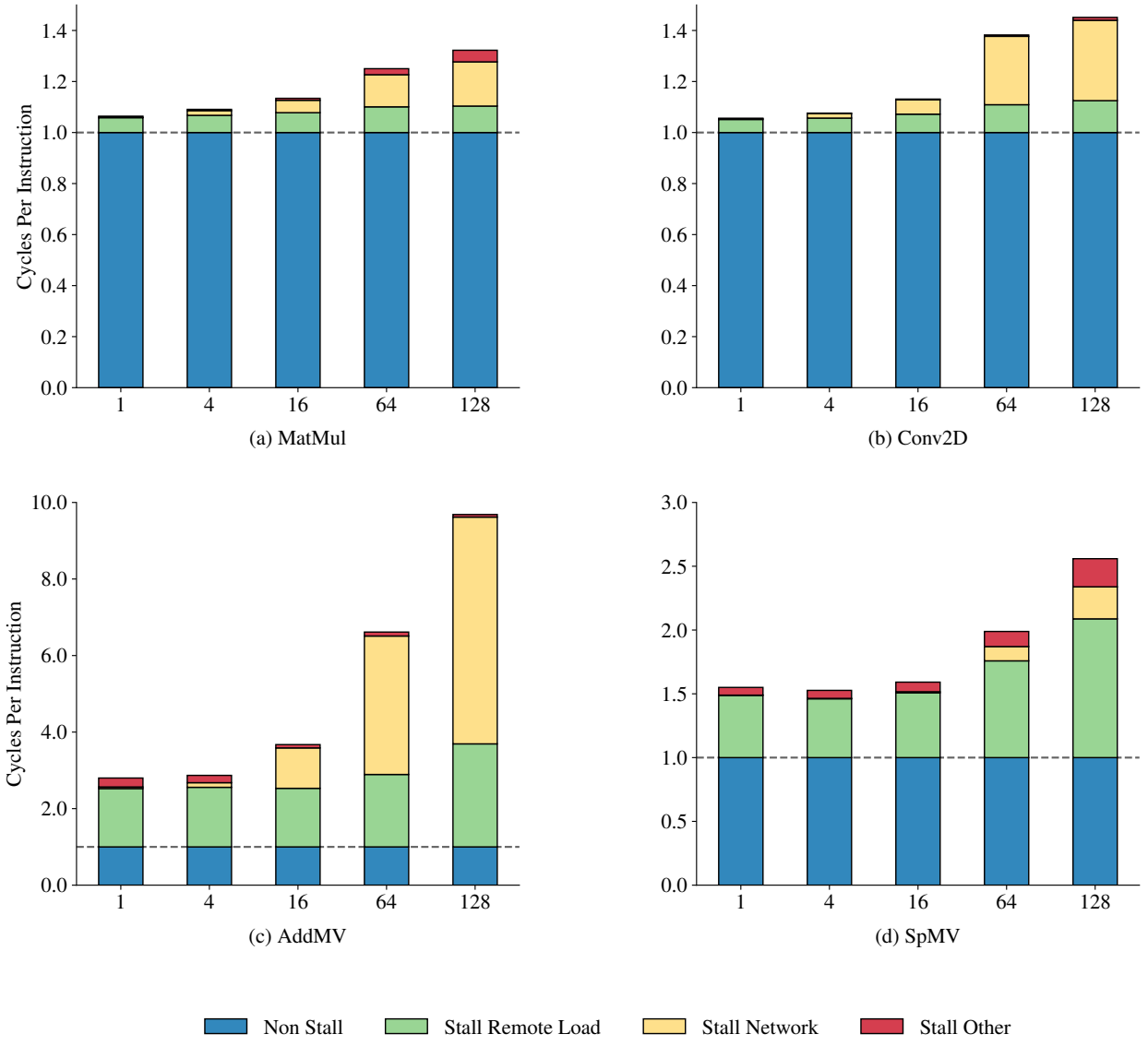


Figure 3.5: Per Core Cycles Per Instruction – Cycles per instruction continues to increase with the number of active cores. Memory latency dominates execution time in all four operators when using 128 cores. Stall-on-Network = load request cannot be sent due to OCN contention; Stall-on-Use = load request has been sent but response haven't received; memory latency = Stall-on-Network + Stall-on-Use.

cessing v0 and C1 is accessing column indices of v3). CBSR+Padding makes indices and values aligned to the same LLC bank, and memory bank conflicts can be completely eliminated.

Our tensor processing framework and the emulation infrastructure are open-source[1]. We use state-of-the-art test-driven design based on `pytest`[2], Hypothesis [MHDmoc19][3], and continuous integration[4]. Operator development proceeds through three levels of emulation, simulation, and finally hardware execution:

Emulation Backend We first develop both the CPU and manycore functions of PyTorch operators using the emulation backend (Figure 3.1(b)). Emulation provides the same APIs as the actual manycore co-processor runtime. It enables functional verification, fast turnaround time, and standard debugging tools (e.g, gdb) on manycore device functions. When building with the emulation backend, offloading uses native function calls, data migration uses regular memory copy, and device functions will be executed natively on the host.

Cosimulation Backend After functional verification, we move to cycle-accurate RTL simulation (Figure 3.1(c)). In this environment, we again verify correctness, and iterate to optimize performance with architectural counters. The cosimulation backend leverages an RTL simulator (e.g., Verilator[5]) to model a small-scale version of the HammerBlade system running at 1GHz with 16 columns and 8 rows. To model DRAM timing we use the open-source DRAMSim3 library [LYR+20], a timing accurate simulator. Architectural performance counters are inserted using non-synthesizable SystemVerilog `bind` statements for no-cost performance analysis of kernels. The RTL for this design has been validated in silicon. Host code executes natively on an Intel Xeon E7-8867v4 CPU. See Section 2.3 for more details.

Prototype Backend Eventually, we plan to support moving to a real FPGA/ASIC prototype (Figure 3.1(d)). Preliminary work has demonstrated the feasibility of using an FPGA prototype to study larger workloads than possible in simulation.

[1] https://github.com/cornell-brg/hb-pytorch
[2] https://pytest.org
[3] https://github.com/HypothesisWorks/hypothesis
[4] https://travis-ci.com/github/cornell-brg/hb-pytorch
[5] https://github.com/verilator/verilator

3.1.2 Micro-Benchmarking

We conduct a scalability study on a set of representative PyTorch operators shown in Table 3.1. These operators vary in arithmetic intensity and enable understanding the performance of our framework on the target CPU-manycore heterogeneous system. Figure 3.4 shows that arithmetic-intensive operators, such as MatMul and Conv2D, scale well and achieve a sustained throughput of 78.5 GFLOP/s and 68.0 GFLOP/s, respectively. Memory-intensive dense operators, such as AddMV, Sum, and Add, show only moderate scalability, as they can easily saturate the many-core co-processor's memory bandwidth. EmbBack is implemented with fine-grained locking, in which each embedding entry is associated with a spin-lock to resolve update conflicts and scales well up to 64 active cores. However, increased memory latency, instead of lock contention, is the primary reason EmbBack scales poorly to 128 active cores. SpMV scales better than other memory-intensive operators because of the CBSR tensor layout, which is specifically designed to avoid LLC bank conflicts on the target manycore co-processor.

We study four operators that are critical to many real-world tensor workloads in more detail: MatMul, Conv2D, AddMV, and SpMV. Figure 3.5 shows that the cycles per instruction (CPI) increases with the number of active cores. For arithmetic-intensive operators such as MatMul and Conv2D, the number of stall-on-network cycles (i.e., load/store requests to LLC cannot be sent due to network congestion) reduces overall performance after reaching 64 active cores (see Figure 3.5 (a–b)). Even with only one active core, MatMul and Conv2D cannot hide enough memory latency to avoid stall-on-use (i.e., true data dependency). Both MatMul and Conv2D can use tiling. Larger tiling blocks increase data reuse resulting in higher arithmetic intensity and thus better performance. However, the necessity of moving large data blocks to the scratchpads with in-order scalar cores introduces phased behavior into these arithmetic-intensive operators. A data-loading phase moves a large block of data into the scratchpad, followed by an execute phase to consume the data block. To move data to the scratchpads, we use a pair of regular load and store instructions. A core first loads a word into one of its registers and then explicitly stores the data into its core-local scratchpad. We can hide memory latency by unrolling the loop so that the instruction stream has a long sequence of loads followed by a long sequence of stores. With stall-on-use, we are able to have many memory requests in-flight which amortizes the memory latency. However, even after applying these optimizations, memory latency still contributes significantly to the overall execution time.

For memory-intensive operators, such as AddMV and SpMV, the number of stall cycles increases quickly beyond 16 active cores (see Figure 3.5 (c–d)). This is likely due to a limited number of LLC banks. With more active cores than available LLC banks, even if memory accesses from cores can be evenly distributed, LLC contention remains. Figure 3.5 shows that unlike AddMV, SpMV execution time is dominated by stall-on-use cycles instead of stall-on-network cycles. This indicates the CSBR tensor layout is able to significantly reduce network congestion.

3.2 First-Order Analysis of SW/HW Scalability

In this section, we conduct first-order end-to-end evaluation on three tensor workloads to evaluate our framework's ability to enable optimized dense and sparse tensor processing on CPU-manycore heterogeneous systems with minimal modifications to existing workloads. We first introduce the workloads and then describe our evaluation methodology. We finish by estimating the performance of the these workloads when scaled to a future 2,000-core CPU-manycore heterogeneous system against an aggressive multicore CPU.

3.2.1 Emerging Tensor Workloads

Residual Neural Network (ResNet) – Residual neural networks are one form of convolutional neural networks (CNNs) for image classification, which won the 2015 ImageNet Large Scale Visual Recognition Challenge by allowing the network's accuracy to scale with its depth [HZRS15]. ResNet introduces *residual blocks*, which are shortcut connections between non-neighboring layers, to overcome a number of training difficulties (e.g., vanishing gradient problem) faced by conventional CNN models. In this work, we build and train a 9-layer ResNet model (i.e., ResNet-9) on the CIFAR-10 dataset. See Figure 3.6.

Recommender System (RecSys) – The input to a recommender system is a list of items a user has previously "liked", and the output is a list of items with scores predicting how much the user might like an unseen item. An autoencoder is a specific kind of unsupervised artificial neural network that learns to copy its input to its output through an intermediate "bottleneck" layer for dimensionality reduction. In this work, we build and train this recommender system on the MovieLens 10M dataset. The implementation of RecSys is illustrated in Figure 3.2 (a).

Local Graph Clustering (LGC-ISTA) – Local graph clustering is an approximate variant of the personalized PageRank algorithm [PBMW99]. Its goal is to find a cluster of nodes that are neighbors of a given seed node. We implement iterative shrinkage-thresholding, which minimizes the loss function of a graph signal vector such that all nodes in the neighborhood of the seed node are associated with high scores, while other nodes receive low scores. The algorithm uses the input adjacency matrix and degree matrix to generate a sparse matrix. It then iteratively updates the gradient, vector, and loss function using SpMV, element-wise multiply, add, and subtraction operations. We run 50 iterations for each seed node on the FB-Johns55 dataset. See Figure 3.7.

3.2.2 Methodology

A common practice to evaluate full-size workloads on simulators is to extract each occurrence of the kernels, and evaluate them individually with either random data or reconstructed data outside of PyTorch. However, this approach leads to inaccuracies since random or reconstructed data may not represent the actual data layout during execution. To address this challenge, we have developed a *re-dispatching* approach that automates the evaluation process and preserves runtime data layout. We first determine which operators in a workload we would like to evaluate, flag them, and then start running the workload on the CPU. When a call-site is reached the execution is forked into a CPU instance (running natively), and a manycore instance (running on an RTL simulator). After both runs return, manycore results are validated against CPU results. With re-dispatching, workload evaluation can be easily parallelized by launching many copies of the workload; one copy for each kernel of interest.

Since it is not feasible to simulate a 2,000-core manycore architecture at reasonable simulation speed, we simulate a smaller 128-core heterogeneous system running 1/16 of the work using the co-simulation infrastructure described in Section 3.1. We then scale the performance of the manycore co-processor to a full 2,000-core system assuming weak scaling. We compare the scaled performance against the performance of running the full workload on the host multicore CPU, which is an aggressive 18-core out-of-order superscalar running at 2.4GHz (Intel Xeon E7-8867v4).

```
class Block(nn.Module):
    def __init__(self, in_channels, out_channels, residual=False):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1,
                      padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
        )
        self.skip = None
        if residual:
            self.skip = nn.Conv2d(in_channels, out_channels, kernel_size=1,
                                  stride=1, padding=0, bias=False)

    def forward(self, xin):
        x = self.layers(xin)
        if self.skip:
            x = x + self.skip(xin)
        return x

class ResNet(nn.Module):
    def __init__(self):
        super(ResNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1,
                      padding=1, bias=False),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            Block(16, 32, True),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            Block(32, 64, True),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            Block(64, 128, True),
            nn.MaxPool2d(kernel_size=(8, 8)),  # global pooling
        )

        self.fc = nn.Linear(128, 2)

        self.logsoftmax = nn.LogSoftmax(dim=-1)

    def forward(self, data):
        x = self.conv(data)
        x = x.view(x.shape[0], -1)
        x = self.fc(x)
        x = x * 0.125  # scale layer
        x = self.logsoftmax(x)
        return x
```

Figure 3.6: ResNet Model

```
# ISTA algorithm

def ista(seeds, adj, alpha, rho, iters):
    out = []
    # Compute degree vectors/matrices
    d       = np.asarray(adj.sum(axis=-1)).squeeze()
    d_sqrt  = np.sqrt(d)
    dn_sqrt = 1 / d_sqrt

    D       = sparse.diags(d)
    Dn_sqrt = sparse.diags(dn_sqrt)
    # Normalized adjacency matrix
    Q = D - ((1 - alpha) / 2) * (D + adj)
    Q = Dn_sqrt @ Q @ Dn_sqrt

    for seed in tqdm(seeds):
        # Make personalized distribution
        s = np.zeros(adj.shape[0])
        s[seed] = 1
        # Initialize
        q = np.zeros(adj.shape[0], dtype=np.float64)
        rad   = rho * alpha * d_sqrt
        grad0 = -alpha * dn_sqrt * s
        grad  = grad0
        # Run
        for _ in range(iters):
            q    = q - grad - rad
            q    = np.maximum(q, 0)
            tmp = Q @ q
            grad = grad0 + tmp

        out.append(q * d_sqrt)

    return np.column_stack(out)
```

Figure 3.7: ISTA algorithm

3.2.3 Results

By leveraging 2D convolution operators with SAD implementations in ResNet, we estimate ResNet can achieve 2× better performance on the target manycore system than on the aggressive multicore CPU (see Table 3.2). 2D convolution operators run much faster on the manycore system by exploiting massive parallelism, but batch normalization and its backward pass (i.e., BatchNorm and BatchNormBack) perform worse on the manycore system compared to the CPU. This is because frequent synchronization is needed in batch normalization operators, and synchronizing the manycore system currently involves higher overhead than synchronizing a multicore CPU. Com-

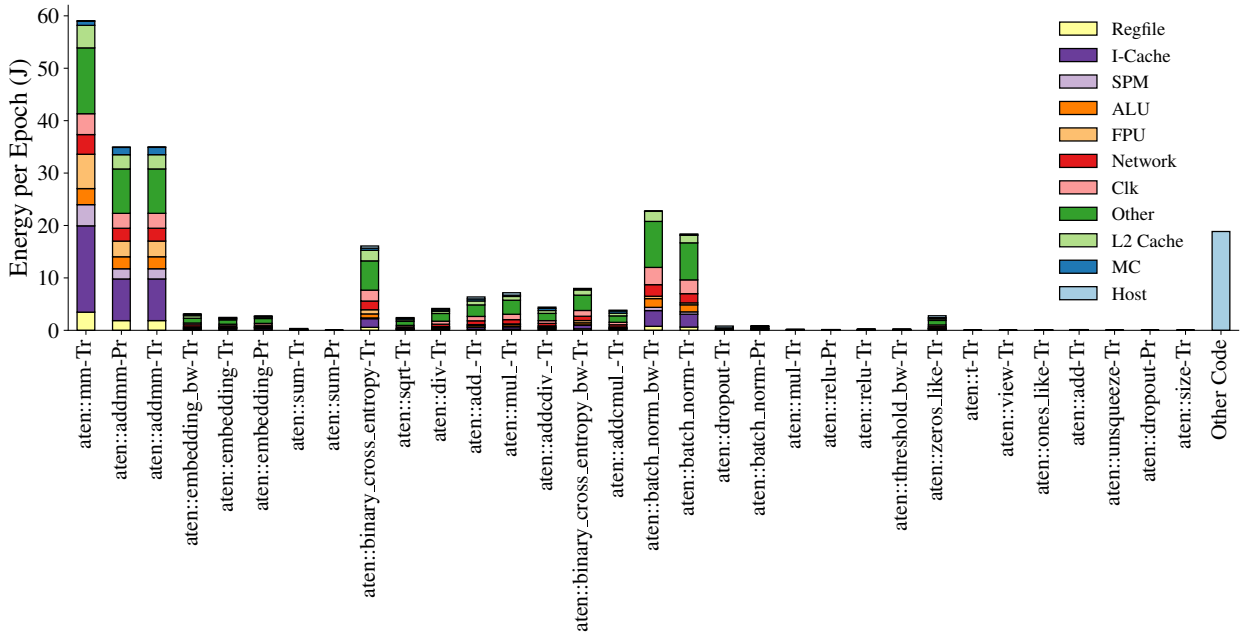


Figure 3.8: RecSys Kernel-Level Energy Breakdown – Energy for each kernel executing on the HammerBlade manycore is shown broken down into various components: register file (Regfile), instruction cache (I-Cache), scratchpad memory (SPM), integer arithmetic logic unit (ALU), floating-point unit (FPU), mesh network, clock tree, LLC, memory controllers (MC), and the host energy. The other category includes pipeline registers, control logic, and other miscellaneous logic. All energy results include leakage. Kernels sorted based on required work per kernel.

pared to having 2D convolution operators implemented with a traditional data-parallel approach, we are able to train ResNet-9 13% faster with systolic-accelerated DAE. Specifically, we observed that Conv2D-fB with systolic-accelerated DAE achieves 2.1× better performance than its data-parallel counterpart, which is higher than we have observed in microbenchmarks (see Table 4.1). Further inspection reveals that unlike the microbenchmarks we used in prior sections, inputs to convolution layers in ResNet do not fit in the LLC. Unstructured memory accesses in the data-parallel implementation lead to significantly more LLC misses.

We estimate RecSys can achieve 5.9× better performance on the target manycore system than on the multicore CPU. Compute intensive operators, such as AddMM and AddMMBack, generally have better performance on the target system because the manycore can better exploit the parallelism in these operators. We also observe that the largest performance improvement comes from embedding (Emb), EmbBack, and Sum. This improvement can be traced to two causes: (1) these operators are memory intensive, and compared to a multicore CPU, the manycore co-processor has a much higher total memory bandwidth (1TB/s); and (2) we apply optimization techniques that are not available by default in the CPU ATen backend such as kernel fusion and intermedi-

ATen Operator	Baseline Time (ms)	MC Total Time (ms)	MC Host Time (ms)	MC Device Time (ms)
Conv2DBack	169.9	45.2	0.9	44.3
Conv2D	77.1	21.9	1.3	20.6
BatchNormBack	18.8	38.2	0.5	37.7
BatchNorm	17.8	36.9	1.9	35.0
Relu	8.5	2.2	0.5	1.7
ThresholdBack	6.3	3.1	0.4	2.7
MaxPool2DBack	6.2	1.2	0.5	0.7
MaxPool2D	5.6	1.1	0.7	0.4
Sqrt	4.3	1.8	0.9	0.9
ZerosLike	3.8	3.0	1.6	1.4
Add	3.3	6.2	2.6	3.6
AddCDiv	3.1	2.2	0.9	1.3
Div	3.1	3.0	1.3	1.7
Other	58.4	32.7	27.6	5.1
Data Transfer	0.0	0.03	0.03	0.0
Total (1 Epoch)	611.2 (s)	310.5 (s)	65.0 (s)	245.5 (s)

Table 3.2: ResNet Execution Breakdown –One training epoch; 1563 batches per epoch; 32 images per batch. MC = target CPU-manycore system. MC total = MC host + MC device.

ate value removal. On the manycore co-processor, we are able to fuse Emb and Sum together to eliminate intermediate value reads and writes. We also explored leveraging systolic-accelerated DAE MatMul in RecSys. However, the dimensions of MatMul instances in RecSys generally lead to severe internal fragmentation [WWB19], and thus worse than baseline performance due to wasted computation. TPUv1 faced a similar issue. Unlike specialized hardware accelerators, we have the flexibility of falling back to a data-parallel implementation with a manycore architecture. We believe other workloads which have more systolic DAE friendly MatMul dimensions will see significant benefits.

We estimate LGC-ISTA can achieve 5.7× better performance on the target manycore system than on the multicore CPU. We observe that unlike RecSys, clustering spends more time on the CPU host than on the co-processor. This is because the input graph has high sparsity, and thus manycore device functions for those operations will not run for long enough time to cover the offloading overhead.

In summary, we estimate all three workloads will be able to achieve much higher (i.e., up to 5.9×) performance on the target CPU-manycore heterogeneous system compared to an aggressive multicore CPU baseline. Note that the weak scaling approach we adopt is optimistic and meant for

ATen Operator	Baseline Time (ms)	MC Total Time (ms)	MC Host Time (ms)	MC Device Time (ms)
EmbBack	427.8	8.2	1.2	6.0
Emb	94.8	1.4	0.5	0.9
Sum	35.7	0.0	0.0	0.0
AddmmBack	23.3	16.4	2.4	14.0
ZerosLike	15.1	4.9	3.9	1.0
CrossEntropyLoss	14.4	10.6	2.7	8.9
Addmm	11.1	7.7	0.5	7.2
BatchNorm	10.1	11.6	1.6	10.0
Addcdiv	8.3	5.4	2.2	3.2
Sqrt	8.3	8.5	1.9	6.6
Div	8.1	7.8	3.4	4.4
BatchNormBack	8.0	8.6	0.6	8.0
Add	7.9	8.9	5.1	3.8
Mul	7.4	11.6	6.6	5.0
Dropout	6.9	6.1	1.4	4.7
Other	17.9	12.4	5.4	7.0
Data Transfer	0.0	3.5	3.5	0.0
Total (1 Epoch)	185.5 (s)	31.5 (s)	11.2 (s)	20.3 (s)

Table 3.3: Recsys Execution Breakdown – One training epoch; 273 batches per epoch; 256 users per batch. MC = target CPU-manycore system. MC total = MC host + MC device.

demonstrating the potential of a future full manycore system, rather than as a rigorous comparison. While computing 1/16 of the output on a 128-core system demonstrates that we have enough software parallelism to fully utilize the 2,000-core system, various architectural challenges (e.g., LLC coherence, DRAM channel scaling, and cross channel data movement) must be solved with minimal performance penalty to realize the estimated performance. This work provides a software stack that lays the groundwork for researchers to explore solutions to these challenges in future work. To help estimate how a future 2,000-core system might compare to a GPGPU, we can consider a previously proposed manycore architecture with 496 RISC-V cores [RZAH+19b, RZAH+19a]. This prior work has shown the ability to achieve 93.04 Giga RISC-V instructions/s per watt and 45.57 GRVIS/ mmsq. Given these prior results, the target CPU-manycore heterogeneous system can potentially achieve significantly higher area-normalized throughput and energy efficiency compared to GPGPUs. Again, this work provides a software stack that can enable more detailed comparative analysis of manycore architectures versus GPGPUs and other programmable accelerators.

ATen Operator	Baseline Time (ms)	MC Total Time (ms)	MC Host Time (ms)	MC Device Time (ms)
SpMV	23960.0	2267.4	1776.0	491.4
Sub	365.9	1120.0	1024.0	96.0
Add	368.8	544.0	496.0	48.0
Max	759.5	480.0	432.0	48.0
Mul	31.1	65.9	56.3	9.6
Clone	0.2	9.6	9.0	0.6
Data Transfer	0.0	2.3	2.3	0.0
Total	25.5(s)	4.5(s)	3.8(s)	0.7(s)

Table 3.4: Local Graph Clustering Execution Breakdown –Personalized PageRank for 500 seed nodes; 50 iterations per seed node. MC = target CPU-manycore system. MC total = MC host + MC device.

3.2.4 Energy Estimation on RecSys

We also conduct an energy analysis with RecSys using the energy modeling methodology discussed in Section 2.3.2. Figure 3.8 plots the energy breakdown for every kernel across various components within the HammerBlade manycore tiles and in the network, clock tree, LLC, memory controllers, and host processor. The dense `mm/addmm` kernels consume about 50% of the overall energy. Other important kernels include the complex `binary_cross_entropy` and `batch_norm` kernels which simply require a significant number of instructions.

3.3 Related Work

Domain-specific programming frameworks express their targeting applications effectively and achieve high performance, and played an important rule in the adoption of GPGPUs and other domain-specific and application-specific accelerators. Examples include CuPy [OUN+17] for array computation, cuGraph [rap20] and Gunrock [WDP+16] for graph analytics, CUVIlib [cuv22] for image processing, and Triton Ocean SDK [Sun22] for water simulation. PyTorch [PGM+19] is an open-source deep earning framework.

Various prior work extended existing programming frameworks to emerging compute platforms such as manycore architectures that do not have hardware coherent caches. For example, Lee et al. [LKK+11] extended OpenCL [ope11] to support Intel SCC [HDH+10]. Marker

et al. [MCP+12] ported a dense matrix library, Elemental [PMVdG+13], to SCC. Our work extended PyTorch to the HammerBlade manycore architecture which captures the common features of SPM manycore architectures. Domain-specific programming frameworks also contributed to the adoption of new compute platforms by offering across platform code portability. For example, TVM [CMJ+18] supports CPUs, GPUs, and also the VTA [MCV+19] architecture. TensorFlow [ABC+16] has backends for CPUs, GPUs, as well as the Google TPU [J+17]. Our work adds another backend to these state-of-the-art software stacks.

3.4 Conclusion

Domain-specific frameworks that provide ready-to-use hand-optimized operators embedded within a high-level language played an essential role in the adoption of GPGPUs. In this chapter, we address the programmability challenge with a tensor processing framework that abstracts hand-optimized operators for dense and sparse workloads. Through end-to-end evaluation of dense and sparse tensor workloads, we show that the proposed framework can potentially achieve up to 5.9× better performance on a 2,000-core CPU-manycore heterogeneous system compared to an aggressive multicore CPU.

CHAPTER 4
HB-ARC: A DECOUPLED ACCESS/EXECUTE FRAMEWORK FOR SPM MANYCORE ARCHITECTURES

Memory latency hiding is now at the center of modern microarchitecture design as the performance gap between compute and memory continues to increase. Multicore CPUs rely on complex out-of-order execution to hide memory latency, while GPGPUs rely on extreme temporal multithreading with fine-grain context switching. Both of these techniques require extensive hardware resources and are not applicable to the simple cores used in manycore architectures. *Non-speculative runahead execution* (i.e., *stall-on-use*), which allows independent instructions to be issued while a long-latency memory instruction is still pending [DM97, MSWP03, MKP05], is a lightweight mechanism to enable memory latency hiding in simple in-order cores. However, Section 3.1 shows this technique alone cannot fully resolve the memory latency issue, and it still dominates the execution time of the HammerBlade manycore for many critical PyTorch operators (e.g., matrix multiplication, 2D convolution, sparse matrix-vector multiplication, and matrix-vector multiplication). Moreover, as manycore architectures generally adopt a mesh-like on-chip network topology, both network bisection bandwidth and the bandwidth to higher levels of the memory hierarchy become scarcer when scaled to future manycore architectures with thousands of cores, leading to increased network congestion and memory access latencies.

Decoupled access/execute (DAE) architectures have been proposed in the literature to aid memory latency hiding by splitting one program into two instruction streams, an access stream and an execute stream [Smi84, HAM15, TJC+18]. The *access stream* contains all instructions related to accessing memory, and the *execute stream* contains the remaining instructions for computation. If the access stream can run sufficiently far ahead, the execute stream will no longer stall due to load-use dependencies. In this chapter, I present HB-Arc[1] which explores DAE in the context of the HammerBlade manycore architecture. Section 4.1 introduces software only techniques to enable decoupled access/execute and systolic execution on SPM manycore architectures. In Section 4.2, we propose combining lightweight *access accelerators* with our software techniques to further improve area normalized throughput.

[1] HB-Arc is named after Arc Warden the character in *Dota 2*, who "creates a copy of himself to split push".

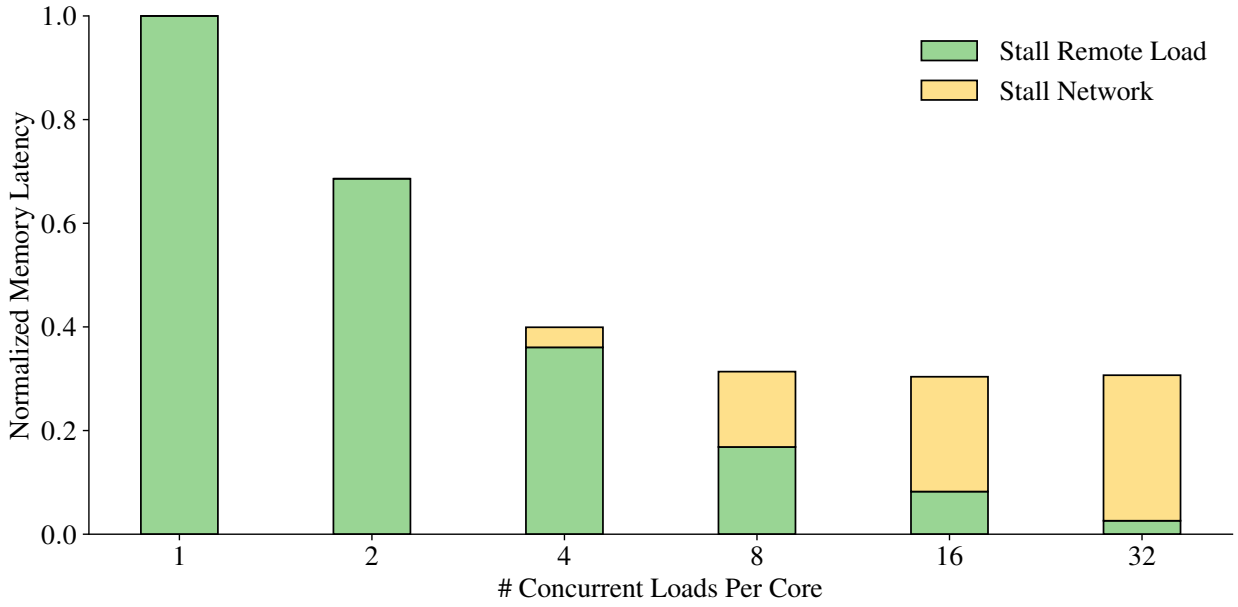


Figure 4.1: Moving Data Blocks with Non-Speculative Runahead Execution – Stall Network = load request cannot be sent due to OCN contention; Stall Remote Load = load request has been sent but response haven't received; memory latency = Stall Network + Stall Remote Load. Normalized to memory latency of 1 concurrent load request per core.

4.1 Software-Enabled DAE

We expect memory latency to become an even more significant issue in future CPU-manycore heterogeneous systems with thousands of cores and 2D mesh on-chip networks, as bisection bandwidth and bandwidth going off the mesh to higher levels of the memory hierarchy scale linearly while the number of cores scales quadratically. As demonstrated for conventional processors in prior work [Smi84, HAM15, TJC+18], decoupled access/execute can reduce or eliminate memory latency and improve performance. In this section, we leverage software-based decoupled access/execute to realize both latency hiding and data movement reduction in the context of a manycore architecture. We propose naïve-software DAE and systolic-software DAE, and we then evaluate their performance against optimized data-parallel baseline implementations.

4.1.1 Naïve-Software DAE

We first explore decoupled access/execute using pairs of cores: one as the access core and one as the execute core. In a typical DAE architecture, access and execute are connected by hardware queues for communication. In the context of a PGAS manycore, we leverage remote store

programming and create software queues in the execute core's scratchpad for the same purpose. We refer to this software decoupled access/execute scheme as *naïve-software DAE*.

In naïve-software DAE, the access core sends requests to higher levels of the memory hierarchy to load data into its registers. Unlike the data-movement scheme described in Section 3.1, the access core stores the loaded value into its peer's scratchpad (i.e., the software queue). When data becomes available, the execute core reads the data block, performs computation, yields the queue space, and writes back the results (if necessary). In many DAE architectures, writing back the results is also done by the access core. However, our early analysis suggested writing results from an execute core to an access core, and then to higher levels of memory hierarchy provided no benefit. Thus, in naïve-software DAE, execute cores write results directly back to DRAM. Since the block currently being processed stays in the software queue (i.e., the execute core pops the entry only after finishing computation), at least two entries in each software queue are necessary to enable access/execute decoupling. This puts increased demand on the scratchpad resulting in smaller tile sizes compared to a data-parallel baseline.

We implement six operators with naïve-software DAE: MatMul, Conv2D, Conv2D-iB (i.e., Conv2D backward w.r.t. input images), Conv2D-fB (i.e., Conv2D backward w.r.t. filters), AddMV, and SpMV. The baselines are hand-tuned data-parallel implementations. We add a second baseline for each operator, in which we only activate 50% of the cores in the manycore co-processor using the data parallel implementation. We refer to this second baseline as 50%-idle. We include this baseline to understand if the benefit of naïve-software DAE comes from fewer cores making memory requests. Since the target manycore is built with scalar cores, each core can inject at most one memory request every cycle. With only 50% cores active, the maximum possible new requests per cycle is halved. This may relieve network congestion and improve operator performance.

Results are summarized in Figure 4.2 and Table 4.1. Compared to the baseline, 50%-idle generally achieves much lower overall throughput, as expected with half of the cores active. However, we also observe an increase in per-core throughput, especially in the cases of AddMV and SpMV. This improvement matches our observation in Section 3.1, that increasing the number of active cores can *reduce* performance due to network congestion. We also observe that for these two operators, naïve-software DAE only provides marginal improvement, or hurts performance because low arithmetic intensity means there is not enough time for the access core to load a block before the execute core needs to consume this block. However, for arithmetic-intensive operators

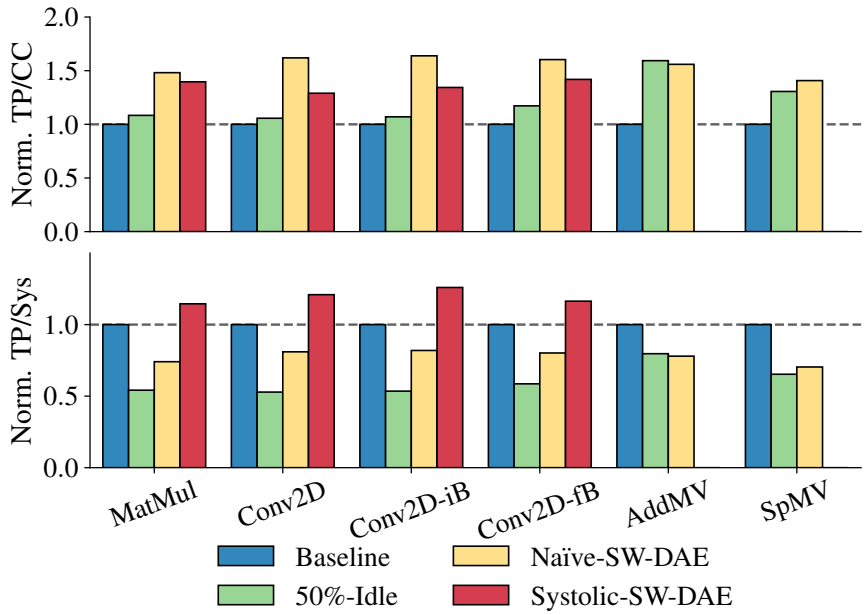


Figure 4.2: Naïve and Systolic Software DAE – TP/CC = throughput per compute core; TP/Sys = overall throughput per system; MatMul showing $768 \times 768 \times 768$; Conv2D, Conv2D-iB, and Conv2D-fB showing 32 images batch; AddMV showing 768×768; SpMV showing FB-Johns55. See Table 4.1 for detailed input specification.

(i.e., MatMul, Conv2D, Conv2D-iB. and Conv2D-fB), naïve-software DAE significantly improves the per-compute-core throughput. Compared to the baseline, naïve-software DAE is able to improve per-compute-core throughput by 1.5–1.9×. Compared to 50%-idle, naïve-software DAE is able to improve per-compute-core throughput by 1.3–1.5×, despite using smaller tiling block sizes than both the baseline and 50%-idle. While this improvement over 50%-idle partially comes from having 2× the resources and offloading load and address generation instructions to access cores, the main source of performance benefit comes from memory-latency hiding. In Conv2D, 13% of the dynamic instructions are related to load and address generation, and these instructions are offloaded to access cores. However, we observe 53% performance improvement over 50%-idle.

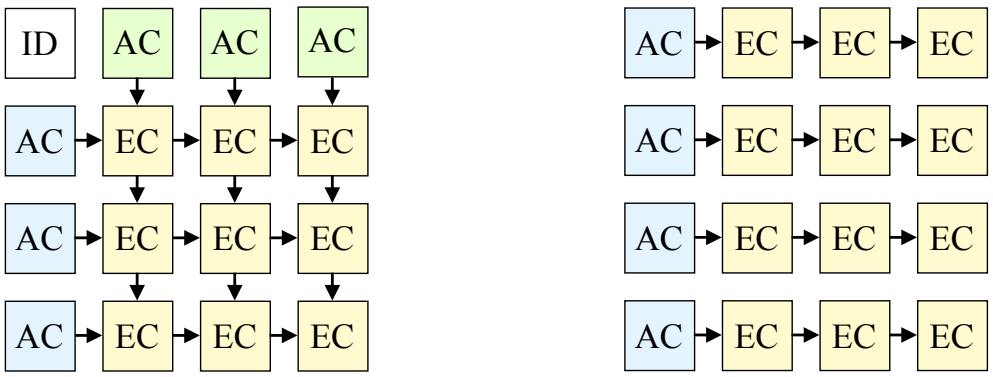
4.1.2 Systolic-Software DAE

While naïve-software DAE implementations show significant per-compute-core improvement, the overall performance decreases because the per-compute-core improvement does not outweigh the reduced number of compute cores performing useful work. To translate the high per-compute-core throughput to an overall performance improvement, we must change the ratio of access to execute cores. However, having one access core serve two or more execute cores can also degrade

performance when the execute cores finish faster than the access core can supply data. For example, in MatMul an access core cannot finish loading data for two execute cores before its execute counterparts finish consuming their current blocks, and thus the execute cores will need to stall. Alternatively, multiple access cores could fetch data for a single execute core. Unfortunately, an asymmetric ratio of access and execute cores results in access cores writing data to execute cores located multiple hops away, which can increase network congestion and further slow down data transfers. Instead of having an access core load independent data blocks for each execute core it serves, we can exploit the fact that the same data is needed by multiple execute cores by intelligently placing the compute and having execute cores pass data blocks in a systolic fashion (i.e., in-compute array reuse). We call this scheme *systolic-software DAE*. Since systolic-software DAE is only feasible for operators with significant data reuse, we focus on the arithmetic-intensive operators (i.e., MatMul, Conv2D, Conv2D-iB, and Conv2D-fB) in the following sections.

The systolic-software DAE implementation of MatMul uses a similar approach as output-stationary systolic hardware accelerators for MatMul, although the systolic-software DAE implementation operates at block granularity instead of scalar value granularity. In systolic-software DAE, blocks of input data are loaded by access cores on the West and North edges of the many-core array, and these blocks are passed along either horizontally or vertically (see Figure 4.3(a)). The systolic-software DAE implementation of Conv2D is implemented in a 1D systolic manner with replication. An input block is passed along a chain of execute cores, in which each execute core applies a different filter to the block (see Figure 4.3(b)). MatMul and Conv2D implemented with systolic-software DAE on a 128-core device has 64% or 88% more respectively execute cores compared to naïve-software DAE.

We implement the four arithmetic-intensive operators (i.e., MatMul, Conv2D, Conv2D-iB, and Conv2D-fB) with systolic-software DAE. Results are summarized in Figure 4.2 and the systolic-software DAE columns of Table 4.1. Conv2D-iB and Conv2D-fB can be implemented in ways that are similar to Conv2D and MatMul, respectively. Across all four operators, systolic-software DAE has a per-compute-core throughput that is lower than naïve-software DAE, but still up to 1.5× higher than the data-parallel baseline. This is because execute cores in systolic-software DAE need to pass data blocks to their neighboring execute cores in addition to performing the actual computation. Additional instructions for data movement lead to lower throughput. However, systolic-software DAE benefits from the additional execute cores, and achieves up to 1.25×



(a) SSD MatMul (b) SSD Conv2D

Figure 4.3: Systolic Mapping – SSD = systolic-software DAE; ID = idle core; AC = access core; EC = execute core. In (a) data is loaded by access cores, and is passed along by execute cores to the South and to the East, while in (b) data is passed in one direction only.

increased system throughput. Note that systolic-software DAE also has fewer compute cores than the baseline. There are three cases (i.e., Conv2D with a batch size of 2 and Conv2D-fB with a batch size of 2 and 4) where systolic-software DAE performs worse than the baseline. This is because in systolic-software DAE data blocks need to be passed from execute core to execute core. Thus, there is a much longer warmup phase for systolic-software DAE, and this results in worse performance when the batch size is small.

4.2 Hardware-Accelerated DAE

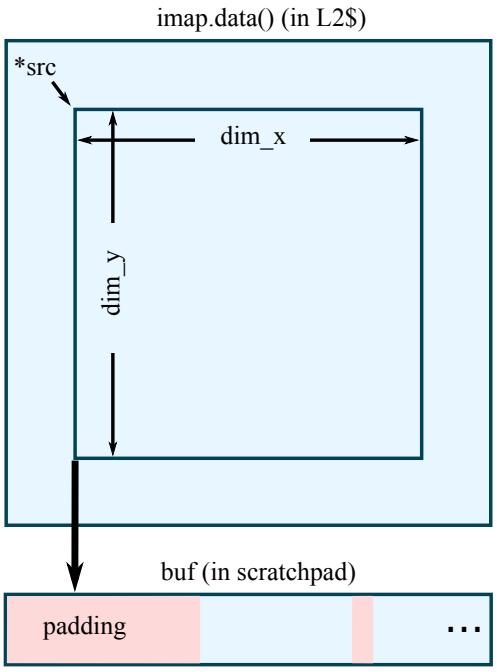
Naïve-software DAE and systolic-software DAE leverage existing hardware mechanisms in the CPU-manycore heterogeneous system and demonstrate both per-compute-core and per-system throughput improvements. However, software-only approaches have two disadvantages. First, general-purpose cores are area-inefficient for data access tasks. Most access tasks only require basic integer arithmetic and simple control flow for 1D and 2D array accesses, but cores in the manycore co-processor are equipped with instruction caches, data scratchpads, and floating point units. Second, dedicating general-purpose cores to data access tasks reduces the peak throughput of the manycore co-processor. While systolic-software DAE can help mitigate this issue by reducing the number of access cores, most operators still require the first column and/or the first row of cores in the manycore co-processor to load data.

		Baseline		50%-Idle		NSD		SSD		NAD		SAD	
Operator	**Input**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**
MatMul	$768 \times 48 \times 768$	0.53	67.8	0.60	38.3	0.89	57.2	0.67	70.4	0.86	81.5	0.64	79.6
	$768 \times 96 \times 768$	0.56	71.6	0.63	40.1	0.92	58.9	0.74	77.9	0.92	87.9	0.71	88.1
	$768 \times 192 \times 768$	0.60	77.3	0.66	42.5	0.95	60.9	0.78	81.7	0.95	90.6	0.75	93.3
	$768 \times 384 \times 768$	0.60	76.5	0.66	42.5	0.96	61.4	0.80	83.7	0.97	92.1	0.77	95.9
	$768 \times 768 \times 768$	0.57	73.6	0.62	39.9	0.85	54.5	0.80	84.3	0.97	92.4	0.78	96.4
Conv2D	Batch Size 2	0.46	58.7	0.52	33.3	0.79	50.4	0.48	57.5	0.74	70.2	0.46	57.5
	Batch Size 4	0.50	63.5	0.55	35.3	0.82	52.6	0.58	69.4	0.78	74.0	0.57	71.0
	Batch Size 8	0.52	66.2	0.56	35.6	0.84	54.1	0.64	76.2	0.80	75.9	0.63	78.9
	Batch Size 16	0.52	67.2	0.56	35.8	0.86	54.7	0.67	80.2	0.81	76.9	0.66	82.0
	Batch Size 32	0.53	68.0	0.56	35.9	0.86	55.0	0.68	82.0	0.81	77.4	0.67	83.6
	Batch Size 64	0.53	68.2	0.56	35.9	0.86	55.2	0.69	82.5	0.82	77.8	0.68	84.3
Conv2D-iB	Batch Size 2	0.46	59.2	0.54	34.4	0.78	50.0	0.49	59.2	0.73	69.7	0.46	56.9
	Batch Size 4	0.50	63.7	0.55	35.3	0.82	52.5	0.59	70.8	0.77	73.7	0.57	70.7
	Batch Size 8	0.52	66.1	0.56	35.6	0.84	53.6	0.65	77.6	0.80	75.9	0.64	79.7
	Batch Size 16	0.52	67.0	0.56	35.7	0.85	54.4	0.68	82.1	0.81	77.2	0.66	81.9
	Batch Size 32	0.52	66.9	0.56	35.8	0.86	54.8	0.70	84.0	0.82	77.7	0.67	83.2
	Batch Size 64	0.53	68.2	0.56	35.9	0.86	55.0	0.71	85.6	0.82	78.0	0.67	83.6
Conv2D-fB	Batch Size 2	0.32	41.3	0.49	31.2	0.64	41.2	0.34	35.4	0.64	60.9	0.28	34.5
	Batch Size 4	0.39	49.5	0.53	33.9	0.76	48.5	0.46	48.0	0.72	68.6	0.40	49.4
	Batch Size 8	0.44	55.9	0.55	35.0	0.76	48.5	0.56	59.2	0.77	73.2	0.51	64.0
	Batch Size 16	0.46	58.3	0.56	35.5	0.75	48.0	0.64	66.7	0.79	75.2	0.58	72.0
	Batch Size 32	0.47	60.6	0.56	35.5	0.76	48.6	0.67	70.6	0.80	76.4	0.61	76.2
	Batch Size 64	0.47	60.0	0.56	35.9	0.76	48.6	0.69	72.9	0.80	75.9	0.63	78.9
AddMV	256×256	0.02	3.0	0.04	2.5	0.04	2.4	–	–	–	–	–	–
	512×512	0.02	3.1	0.04	2.5	0.04	2.7	–	–	–	–	–	–
	768×768	0.03	4.4	0.05	3.5	0.05	3.4	–	–	–	–	–	–
	1024×1024	0.03	3.7	0.04	2.9	0.05	3.1	–	–	–	–	–	–
SpMV	FB-Johns55	0.04	4.9	0.05	3.2	0.05	3.5	–	–	–	–	–	–
	Facebook	0.02	2.9	0.03	2.2	0.04	2.5	–	–	–	–	–	–
	Cora	0.01	1.0	0.01	0.8	0.02	1.0	–	–	–	–	–	–
	CiteSeer	0.01	0.9	0.01	0.7	0.01	0.9	–	–	–	–	–	–

Table 4.1: Operator Throughput –MatMul = matrix multiplication; Conv2D = 2D convolution; Conv2D-iB = 2D convolution backward w.r.t. input image; Conv2D-fB = 2D convolution backward w.r.t. filters; AddMV = general matrix-vector multiplication; SpMV = sparse matrix-vector multiplication; TP/C = throughput per compute core; TP/S = overall throughput per system; NSD = naïve-software DAE; SSD = systolic-software DAE; NAD = naïve-accelerated DAE; SAD = systolic-accelerated DAE. The target system has 128 cores. Conv2D, Conv2D-iB, Conv2D-fB are run with 16-channel 32×32 images with 32 3×3 filters. FB-Johns55 has sparsity of 1.4×10^{-2}; Facebook has sparsity of 5.4×10^{-3}; Cora has sparsity of 1.4×10^{-3}; CiteSeer has sparsity of 8.3×10^{-4}. Per system throughput in naïve-accelerated DAE and systolic-accelerated DAE are area-normalized to baseline manycore. All numbers are in GFLOP/s.

```
1  float* src = imap.data()
2    + img_id  * strides[0]
3    + chnl_id * strides[1]
4    + blk_y   * strides[2]
5    + blk_x   * strides[3];
6
7  if ( pad_first_col )
8    pad_col(buf, 0);
9  if ( pad_last_col )
10   pad_col(buf, dim_x-1);
11 if ( pad_first_row )
12   pad_row(buf, 0);
13 if ( pad_last_row )
14   pad_row(buf, dim_y-1);
15
16 for r in range(0, dim_y) {
17   for c in range(0, dim_x)
18           buf[offset+c] = *(src+c);
19   src += strides[2];
20   offset += dim_x;
21 }
```

(a) Data Access Pseudo Code



(b) Data Access Illustration

Figure 4.4: Conv2D Forward Data Access – In the Conv2D forward kernel, the access cores run program in (a) and load input feature map blocks into the target data scratchpad as shown in (b). Note the access cores calculate `src` and pad zeros (in red) to the imap buffer.

We adopt a software/hardware co-design approach to address these challenges. We design and implement an *access accelerator (AX)*, a configurable hardware unit that streams data from the LLC to the scratchpad of a target execute core. Compared to general-purpose cores, an access accelerator is significantly more area efficient, yet still provides the benefits of decoupled access/execute. This light-weight access accelerator also achieves the same peak computation throughput as the baseline manycore with very low area overhead. While having hardware engines that are dedicated for moving data (e.g., DMA engines) is not a new idea, the proposed access accelerator is unique in its ability to act as a first-class citizen in both the mesh-based on-chip network and the remote store programming model.

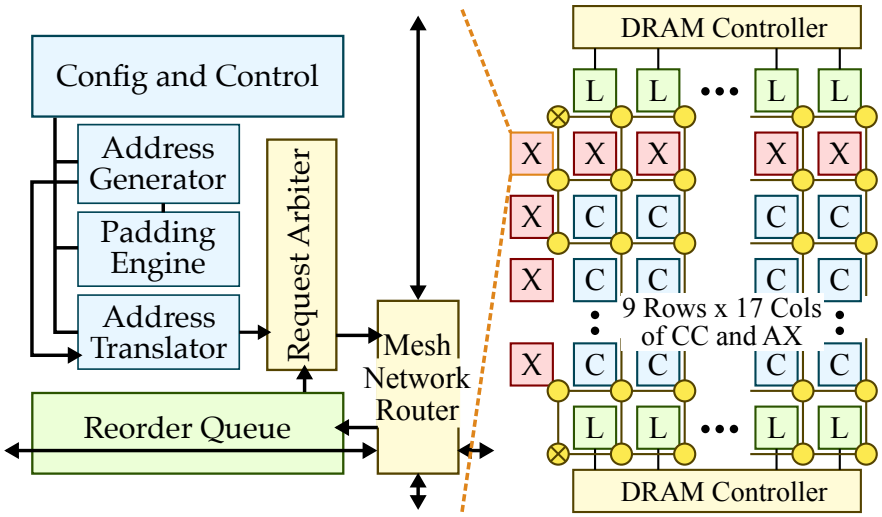
4.2.1 Access Accelerator Design

Data Access Tasks – Figure 4.4 shows the data access pseudocode of the Conv2D kernel and illustrates how the access cores load data from the LLC and pad zeros to the input feature map block. While we explored several operators with software-only DAE schemes, their data access

patterns are all similar. In general, data access tasks involve two nested `for` loops that load a matrix of size `dim_x` by `dim_y` into the scratchpad of the target execute core and an optional padding process that pads zeros around the matrix. This generic data access pattern can be efficiently implemented as an access accelerator that correctly performs common data access tasks given the metadata about the accesses (i.e., the source address, dimensions, strides, padding information, and the destination address).

Accelerator Design – Figure 4.5(a) shows the architecture of the access accelerator and how it is connected to a mesh network router. At the core of the access accelerator is a configurable address generator and a padding engine. These two modules generate a stream of memory requests. Since the mesh network in the target manycore system is only point-to-point ordered, the access accelerator also includes a reorder queue to reorder the memory responses from different LLC banks. The request arbiter arbitrates between memory read requests to the LLC and remote store requests to the target scratchpad because there is only one master interface exposed by the mesh network router. Finally, an address translator is required because the execute cores configure access accelerators using virtual addresses.

Accelerator Integration – Figure 4.5(b) illustrates how access accelerators are integrated into the target manycore array. In the baseline manycore, each mesh network router is connected to a RISC-V core. To integrate the access accelerators, we extend the mesh network and instantiate access accelerators at the top row and the left-most column. This composition works particularly well with systolic-software DAE implementations where most on-chip network traffic is between neighboring cores or accelerators. This composition also ensures a fair comparison with the baseline manycore system for two reasons. First, the access accelerator manycore (AX manycore) has the same number of LLC banks and the same DRAM bandwidth as the baseline manycore. Second, the AX manycore has the same effective mesh network bandwidth as the baseline. The AX manycore mesh network does have larger bisection bandwidth than in the baseline manycore. However, this additional bandwidth does not translate into improved throughput because the extra network links and routers are mostly used to provide access to LLC banks to the access accelerators. The AX is a first-class citizen in the remote store programming model: execute cores control a neighbor AX by performing remote stores into the AX's memory-mapped control registers, and the AX performs remote stores into its neighboring execute core's scratchpad upon receiving data from the LLC.



(a) Access Accelerator (b) AX in Target Manycore

Figure 4.5: Access Accelerator Architecture and Integration – (a) architecture of the access accelerator and how it connects to a mesh network router; (b) access accelerators integrated in the first row and first column of the target manycore. X = access accelerator (AX), L = LLC bank, C = compute core (CC).

4.2.2 Access Accelerator Evaluation

Area – Figure 4.6 compares the post-place-and-route area of an access accelerator in a CMOS 14/16 nm technology and a general-purpose core from prior work in a similar process [RZAH+19a]. We can see from the figure that the access accelerator is highly area-efficient. The network router and endpoint consumes about 40% and the accelerator data path consumes about 30% of the access accelerator area. The transmit adapter (TX) includes a 32-element FIFO to buffer responses from the LLC, and consumes around 30% of the accelerator area. Overall, the access accelerator is 5× smaller than the general-purpose core, making it an area-efficient choice for data access tasks. The AX manycore (with an extra AX row and AX column as shown in Figure 4.5(b)) only increases the overall area by 2.9% compared to the baseline manycore.

Naïve-Accelerated DAE – Similar to the naïve-software DAE evaluation (NSD, see Section 4.1.1), we evaluate the area efficiency of the access accelerators using a naïve-accelerated DAE (NAD) composition. In NAD, each execute core is paired with an access accelerator that replaces the access core. Figure 4.7(a) and the NAD column of Table 4.1 shows the per-compute-core throughput and the area-normalized per-system throughput of different operators under NAD. We can see that compared to NSD, NAD has similar per-compute-core throughput since both access cores and access accelerators are able to decouple data access from the computation on

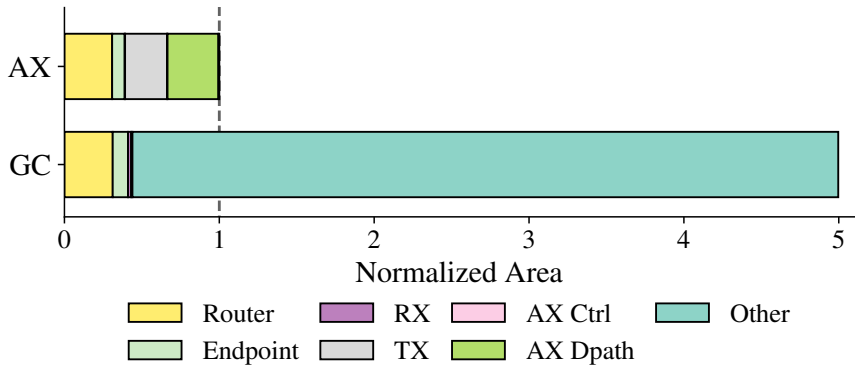
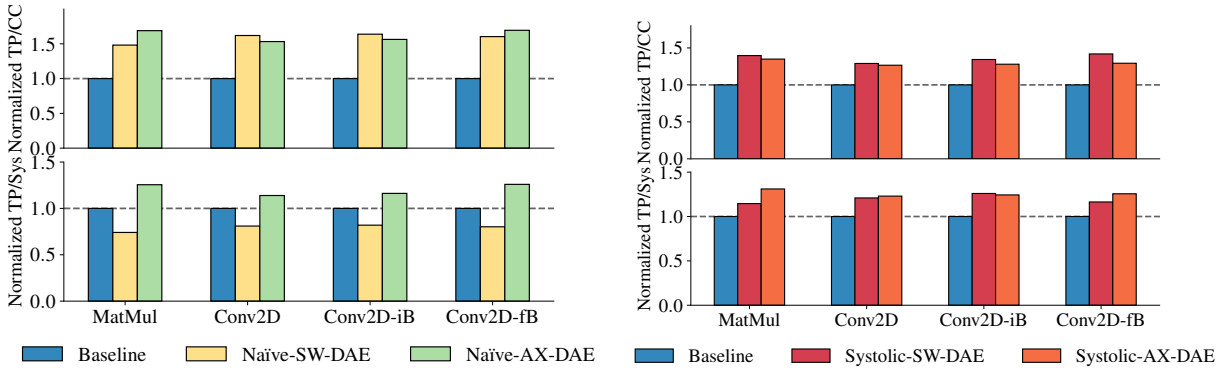


Figure 4.6: Access Accelerator (AX) and General-Purpose Core (GC) Normalized Area – AX eliminates instruction cache, data scratchpad, FPU, etc. and is 5× smaller than a GC in a similar CMOS technology. RX/TX = RX/TX adapter, Ctrl = control logic, Dpath = data path.



(a) Naïve Accelerated DAE

(b) Systolic Accelerated DAE

Figure 4.7: Naïve and Systolic Accelerated DAE – TP/CC = throughput per compute core; TP/Sys = overall throughput per system; MatMul showing $768 \times 768 \times 768$; Conv2D, Conv2D-iB, and Conv2D-fB showing 32 images batch; AddMV showing 768×768; SpMV showing FB-Johns55. See Table 4.1 for detailed input specification.

execute cores. However, NAD has significantly higher area-normalized per-system throughput (46% on average) than NSD. This difference is the largest on the matrix multiplication (MatMul) operator, where NAD achieves 52% higher area-normalized per-system throughput. The superior area-normalized per-system throughput of NAD over NSD confirms that our access accelerator is significantly more area-efficient on data access tasks than general-purpose cores, and still provides the same throughput benefits of DAE. We did not implement and evaluate NAD versions of memory-intensive operators (i.e., AddMV and SpMV). NAD cannot address the fact that these operators are largely limited by memory bandwidth. Prior evaluation has shown that a data-parallel scheme is more effective (see Section 4.1.1).

Systolic-Accelerated DAE – As discussed earlier, systolic-software DAE dedicates multiple general-purpose cores to load data at the cost of manycore compute resources. Based on the systolic-software DAE (SSD, see Section 4.1.2), we create the systolic-accelerated DAE composition (SAD), which uses the access accelerator manycore described in Section 4.2.1 to run systolic-software DAE implementations. Figure 4.7(b) and the SAD column of Table 4.1 shows the per-compute-core throughput and area-normalized per-system throughput of different operators under SAD. We can see that compared to SSD, SAD has similar per-compute-core throughput since both designs are able to achieve decoupled access/execute. In terms of overall area-normalized per-system throughput, SAD has an average of 4.8% better throughput than SSD. On MatMul, SAD is able to achieve 13.9% better average throughput than SSD. On the target 16×8 manycore array, the SSD approach uses eight (Conv2D and Conv2D-iB) or 23 (MatMul and Conv2D-fB) general-purpose cores for data accesses. Therefore, the maximum overall per system throughput improvement of SAD on the same manycore is 6% or 18% (depending on the kernel). In addition, the execute cores in SAD need to perform remote memory stores to configure the access accelerators for every input feature map block, which occupies computation cycles. Despite having more moderate throughput improvements over the highly optimized SSD design, SAD still achieves the highest area-normalized throughput on the four evaluated kernels among all six designs (baseline, 50%-idle, NSD, SSD, NAD, SAD). Compared to the baseline, the AX manycore introduces one extra cycle to the memory latency when accessing LLC banks in the north. However, this should have negligible performance impact on operators that cannot leverage SAD, as our prior results in Section 3.1.2 have shown that network congestion is the main source of stalls for operators implemented with a data-parallel scheme.

4.3 Related Work

A wide variety of coarse-grain parallel architectures have been developed over the past decade to exploit pipeline parallelism. Architectures like Eyeriss [CKES16] and DianNao [CDS+14] are domain-specific accelerators for convolutional neural networks. Later versions support operations on sparse tensors. These proposals demonstrate similar parallel dataflow patterns. The TPU [J+17] and VTA [MCJ+18] architectures integrate systolic matrix-multiply and vector processing units to accelerate more general machine learning computations. More general purpose architectures also

exist: RAW [TLM+04] uses an inter-processor scalar operand network to forward results between processors. Plasticine [PZK+17] contains a mesh of general-purpose compute units for processing workloads from machine learning, data, and graph analytics. These architectures exploit pipeline parallelism by composing coarse grain functional units, similar to our work.

Many architectural solutions have been proposed to decouple memory and compute operations [Smi84]. Decoupled Supply Compute (DeSC) [HAM15] is an automatic extension of DAE for general-purpose CMPs that uses a "Supplier Device" and a "Compute Device", similar to our naïve-software DAE approach. The Load Slice Core [CHA+15] is a form of restricted out-of-order machine. With an additional pipeline, load and address generation slices can be issued out-of-order and speculatively with respect to compute slices, while remaining in-order within a slice. Slice formation is handled by hardware. Tran et al. [TJC+18] proposed a SW/HW co-design method. Instructions are grouped into access and execute phases at compile time. Access phases can run and commit out-of-order with respect to execute phases at runtime. Both techniques rely on hardware that is more complex than the target manycore architecture provides (e.g., superscalar cores). Manticore [ZSB21] introduces custom ISA extensions to leverage DAE and improve FPU utilization. Techniques proposed in this work aim to enable DAE in the context of a manycore with thousands of simple stall-on-use in-order scalar cores, and with existing programming model and core microarchitecture. The Cell processor [GHF+06] includes per-core DMA engines to overlap computation with data transfer. The Epiphany processor [Olo16] also includes a DMA engine. This prior work explores pairs of memory and compute engines, while our approach extends this idea with AX's along the periphery of the target architecture. Our approach is more similar to CoRAM [CHM11], where a control thread can manage multiple scratchpads on an FPGA device. Recent work has shown the potential of using a chiplet-based approach to scale the target manycore architecture to thousands of cores [VGT+20, ZSB21].

Several high-level languages have been created to express complex pipeline parallelism in programming. StreamIt [GTA06] exposed pipeline parallelism for the RAW architecture. More recent work has enabled pipeline parallelism for general-purpose machines. Interstellar [YGL+20] is an extension to Halide's scheduling with pipeline parallelism expressions. Spatial [KFP+18] is a general-purpose DSL for expressing pipelines and can target Plasticine [PZK+17]. These languages are higher-level than our own development language and can be used in the future to ease programmer expression of pipeline parallelism on manycore architectures.

4.4 Conclusion

In this chapter, I identify memory latency as the key limiting factor for performance on SPM manycore architectures and refer to it as the manycore memory latency challenge. I address this challenge by exploring both software and hardware-accelerated decoupled access/execute schemes on the manycore co-processor. Operators implemented with the proposed techniques achieve up to 1.32× throughput improvement, compared to an aggressive data-parallel baseline.

CHAPTER 5
HB-RUBICK: A DYNAMIC TASK PARALLEL FRAMEWORK FOR SPM MANYCORE ARCHITECTURES

In Chapter 3 and Chapter 4, I introduced two domain-specific frameworks, one for dense and sparse tensor processing and one for enabling decoupled access/execute on SPM manycore architectures. While HB-PyTorch can significantly improve the programmability of HammerBlade, it suffers from relatively low performance compared to implementing the same workload directly with CUDA-lite. While HB-Arc provides considerable performance gain over CUDA-lite, it requires programmers to manually reformat and hand tune their applications. More importantly, both frameworks are domain-specific frameworks. Even though such frameworks express domain-specific workloads effectively and/or achieve high performance, not every domain is covered. Extending and re-purposing these frameworks for another domain requires non-trivial efforts by programmers.

In this chapter, we take inspiration from the success of the dynamic task parallel programming model in the multi-core era, and attempt to address the programmability challenge of SPM manycore architectures as well improve performance by offering a dynamic task parallel programming framework, HB-Rubick[1] , that is similar to those that thrived on multi-core systems (e.g., Intel Cilk Plus [int13], Intel Threading Building Blocks (TBB) [int19], and OpenMP [ACD+09, ope13]). These programming frameworks allow parallel tasks to be generated and mapped to hardware dynamically through a software runtime. They can express a wide range of parallel patterns, provide automatic load balancing, and improve portability for legacy code [MRR12]. Our approach allows dynamic task parallel applications written for traditional hardware-based cache coherent multi-cores to work on manycore architectures with only minimal changes to the software. While conventional wisdom believes implementing a work-stealing runtime is either not viable or not beneficial on systems that do not have caches [ZP16, WTCB20], our evaluation demonstrates that our proposed task parallel programming framework can achieve 1.2×–28.5× speedup for workloads that benefit from our techniques, and only induce minimal overhead for workloads that do not.

In Section 5.1, I provide a general background on work-stealing runtimes. In Section 5.2, I describe in detail how to implement a work-stealing runtime, which is the core component of

[1] HB-Rubick is named after *Rubick* the character in *Dota 2*, who "always seeks a new spell to steal".

dynamic task parallel frameworks, on manycore architectures with software-managed SPMs. In Section 5.3, I discuss three optimizations for enabling the runtime to leverage SPMs and achieve high performance. In Section 5.4 and Section 5.5, I use a cycle-accurate RTL evaluation methodology to demonstrate the potential of our approach with four categories of workloads: *static-balanced*, *static-unbalanced*, *dynamic-balanced*, and *dynamic-unbalanced*. Section 5.6 discusses related work of this chapter.

5.1 Programming Models for Dynamic Task Parallelism

Task parallelism is a style of parallel programming where the workload is divided into *tasks* (i.e., units of computation that can execute in parallel). Dynamic task parallelism is a kind of task parallelism in which tasks and dependencies among tasks are generated at runtime. Dynamically generated tasks are assigned to available worker threads based on a certain scheduling algorithm. The most adopted computation model for dynamic task parallelism is the *fork-join* model. It was first introduced by MIT Cilk [BJK+95] and then adopted by various parallel programming frameworks [Lei09,int13,Rei07,int19,CGS+05,SML17]. Fork-join parallelism naturally describes the parallel execution of a program: the program starts running serially and only one control flow exists. At the beginning of the parallel region, the serial control flow *forks* into two or more independent control flows that can be executed in parallel. At the end of the parallel region, these independent control flows *join* into a single control flow and serial execution resumes. In a task parallel programming framework that adopts the fork-join model, the process in which a task forks two or more parallel tasks is also referred to as *spawning* tasks. The newly created tasks are called the *child* tasks and the original task is called the *parent*. The parent task can continue until it reaches the point where *join* (also commonly referred to as *wait*) primitive is called. The parent task blocks until all of its child tasks have finished. This model serves as a basis to express many complex parallel patterns, including divide-and-conquer, parallel loop, reduction, and nesting [Rei07]. The fork-join model has the following properties: (1) a task can only wait for its children to join (e.g. no waiting on locks); (2) a task cannot complete until all of its children complete and join it. This set of properties is called *fully-strict* in Cilk literature [BJK+96, FLR98].

Work-stealing is likely the most widely-adopted scheduling algorithm for task parallel programming frameworks [BL99]. In a typical work-stealing runtime, each thread is associated with a *task queue* to store tasks that are ready for execution. The task queue is usually implemented

with a double-ended queue (*deque*). When a task spawns a child task, it *enqueues* the child on to the task queue of the executing thread. When a thread becomes idle, either because a parent task is waiting for its child tasks to return or the thread has no active task running, it attempts to *dequeue* from its own task queue from the end (i.e., in last-in-first-out (LIFO) order). If the task queue is not empty and a task is popped, the thread starts executing this task. If the task queue is empty, the thread then attempts to *steal* a task from the head of the task queue of another thread (i.e., in first-in-first-out (FIFO) order). The stealing thread becomes a *thief*, and the thread whose tasks are stolen becomes a *victim*. Stealing in FIFO order allows the thief to steal a task that locates higher in the task graph, which typically contains more work. The stealing mechanism automatically balances the workload across threads. It leads to better locality and helps establish time and space bounds [BL99, FLR98].

5.2 Supporting Dynamic Task Parallelism on SPM Manycore

In this chapter, we propose to resolve the manycore architecture programmability challenge by implementing a TBB/Cilk-like dynamic task parallel programming framework on such systems. Compared to the typical low-level C runtimes provided by these architectures which usually adopt the SPMD programming model, the proposed framework supports parallel patterns beyond simple static parallel loops, allows parallel patterns to be arbitrarily nested, and provides dynamic load balancing. Compared to prior work on resolving the programmability challenge, which takes a domain-specific approach, our parallel programming framework is general-purposed and provides a programming interface that programmers who have used either Cilk/TBB or OpenMP are familiar with. This enables simple porting of legacy code to manycore architectures.

The core component of the proposed TBB/Cilk-like dynamic task parallel programming framework is a work-stealing runtime. While how to implement work-stealing runtimes on systems with hardware-based coherence [BJK+95], software-centric coherence [LZF08, WTCB20, TCM18], and distributed memory [DLS+09, PCM+07, SKK+11] has been studied extensively in the literature, conventional wisdom claims that implementing such a runtime is either not viable or not beneficial on systems with software-managed scratchpad memories. In this section, we first demonstrate our programming model using running examples. We then describe a naïve implementation of a work-stealing runtime on the HammerBlade manycore.

```
template <typename Func>
class FibTask : public Task {
public:
  FibTask( int n_, int* sum_, Task* parent_) :
    n( n_ ), sum( sum_ ), parent( parent_ );

  Task* execute() {
    if ( n < 2 ) {
      *sum = n;
      return;
    }

    int x, y;
    FibTask a( n - 1, &x, this );
    FibTask b( n - 2, &y, this );
    this->set_ready_count( 1 );

    task::spawn(b);
    a.execute();

    task::wait();
    *sum = x + y;
    return nullptr;
  }
private:
  int n;
  int* sum;
  Task* parent;
};
```

(a) `fib` using `spawn` and `wait`

```
class Task {
public:
  Task();
  virtual Task* execute();
  void set_ready_count(
    int ready_count );
private:
  int  ready_count;
};
```

(b) `Task` base class

```
int fib( int n ) {
  if ( n < 2 ) {
    return n;
  }
  int x, y;
  parallel_invoke(
    [&]{ x = fib( n - 1 ); },
    [&]{ y = fib( n - 2 ); }
  );
  return x + y;
}
```

(c) `fib` using `parallel_invoke`

```
void vvadd( int a[], int b[],
            int dst[], int n ) {
  parallel_for( 0, n,
    [&]( int i ) {
      dst[i] = a[i] + b[i];
    });
}
```

(d) `vvadd` using `parallel_for`

```
void sum( int a[], int n ) {
  int ident = 0;
  parallel_reduce(0, n, ident,
    [&](int i) {
      return a[i];
    },
    [](int x, int y) {
      return x + y;
    });
}
```

(e) `sum` using `parallel_reduce`

Figure 5.1: Task-Based Parallel Programs – Examples for calculating the Fibonacci number using (a) a low-level API with explicit calls to `spawn` and `wait`; and (c) a high-level API with templated `parallel_invoke` pattern. (b) shows the `Task` based class in which the `FibTask` class inherits from in (a). (d) and (e) show alternative templated patterns `parallel_for` and `parallel_reduce` respectively.

```
template <typename RangeT, typename BodyT>
class ParallelForTask : public Task {
public:
  ParallelForTask( const RangeT& range, const BodyT& body )
      : m_range( range ), m_body( body )
  {
  }

  Task* execute()
  {
    if ( m_range.divisible() ) {
      RangeT  new_range = m_range.split();
      Task    join_point( 2 );

      ParallelForTask<RangeT, BodyT> right_half( new_range, m_body );
      right_half.set_successor( &join_point );

      // spawn the right half
      spawn( &right_half );

      // execute the left half directly
      execute();

      wait( &join_point );
    }
    else {
      m_body( m_range );
    }
    return nullptr;
  }

private:
  RangeT m_range;
  BodyT  m_body;
};
```

Figure 5.2: Implementation of `parallel_for`

5.2.1 Running Example

We use an application programming interface (API) similar to Intel TBB to illustrate our programming model (see Figure 5.1). Each task is described by a C++ class derived from the `Task` base class (Figure 5.1 (b)) which contains a `execute()` method and a metadata variable `ready_count`, also known as the *reference counter*. This metadata tracks a task's unfinished child tasks. After a task finished execution, it checks if it has a parent task. If so, the child will decrement the `ready_count` variable of its parent task to signal its completion. A task in `wait` will be blocked until its `ready_count` reaches 0 (i.e., all children have completed their execution).

```
template <typename RangeT, typename BodyT>
class ParallelReduceTask : public Task {
public:
  ParallelReduceTask( const RangeT& range, const BodyT& body )
      : m_range( range ), m_body( body )
  {
  }

  ParallelReduceTask<RangeT, BodyT> split()
  {
    return ParallelReduceTask<RangeT, BodyT>( m_range.split(),
                                              m_body.split() );
  }

  Task* execute()
  {
    if ( m_range.divisible() ) {
      Task join_point( 2 );
      auto right_half = this->split();

      right_half.set_successor( &join_point );

      // spawn the right half
      spawn( &right_half );

      // execute the left half directly
      execute();

      wait( &join_point );

      // reduce
      m_body.reduce( right_half.m_body );
    }
    else {
      m_body( m_range );
    }
    return nullptr;
  }

  BodyT get_body() const { return m_body; }

private:
  RangeT      m_range;
  BodyT       m_body;
};
```

Figure 5.3: Implementation of `parallel_reduce`

This mechanism enforces the ordering between parent and child tasks: a task can not complete until all of its children complete and join it (see Section 5.1). Programmers overwrite the virtual `execute()` function to hold the logic of the concrete task. In this example (Figure 5.1 (a)), after creating two child tasks `a` and `b`, one for `fib(n-1)` and one for `fib(n-2)`, the parent task (i.e., `fib(n)`) puts `fib(n-2)` onto the task queue and executes `fib(n-1)` locally, before calling `wait()`, which blocks its execution until task `fib(n-2)` returns. The parent task then calculates `fib(n)` by adding the partial results from both tasks and returns. Besides low-level APIs we have shown above, our framework also provides templated functions that support various parallel patterns. This includes `parallel_invoke` for divide-and-conquer (Figure 5.1 (c)), `parallel_for` for parallel loops (Figure 5.1 (d)), and `parallel_reduce` for parallel reduction (Figure 5.1 (e)). Figure 5.2 and Figure 5.3 illustrates how these high level templated functions are implemented with `spawn` and `wait`.

5.2.2 A Naïve Work-Stealing Runtime

The key challenge of implementing a work-stealing runtime on a system like HammerBlade is to cope with the lack of data coherence mechanisms. Typical work-stealing runtimes are built upon various shared data structures (e.g., task queues and reference counters). Where to allocate them and how to keep them coherent is the key question to ask. While possible if carefully implemented, programmers usually avoid keeping copies of shared data in software-managed scratchpads. Instead, they tend to allocate them in the last shared level of the memory hierarchy. While doing so yields longer memory latency for accessing these shared data, keeping multiple copies of scratchpad allocated data coherent is an even worse nightmare that only a few, if not none, highly experienced programmers are willing to face. By allocating all data in the shared memory space, we can easily implement a naïve work-stealing runtime that runs on the HammerBlade manycore architecture. Namely, the runtime does not utilize the scratchpads at all: all data live in the DRAM address space (recall that HammerBlade adopts a PGAS memory model, and DRAM has an address space that is separated from the scratchpads).

Figure 5.4 (a) shows an implementation of the `spawn` and `wait` functions for this naïve work-stealing runtime. `spawn` enqueues a task pointer onto the current thread's task queue, and `wait` puts the current thread into a *scheduling loop*. Within the scheduling loop, a thread first check if all of its child tasks have returned (i.e., `ready_count` has a non-zero value). If so, the thread

exits from the scheduling loop and resume the execution of the parent task (line 8). Otherwise, the thread first attempts to pop a task from the end of its own task queue (i.e., LIFO order, lines 9–15). If there is no task left in the local queue, the current thread becomes a thief and attempts to steal tasks queue of another thread, the victim, from the head (i.e., FIFO order, lines 17–24). The victim is selected randomly (line 17). When a task is executed, its parent's reference counter is atomically decremented (lines 14 and 23). Readers familiar with Intel TBB-like work-stealing runtimes may notice that this implementation is very similar to the implementation on traditional hardware coherent multi-cores. On hardware coherent multi-cores, hardware cache coherence protocols keep multiple copies of shared data coherent. On HammerBlade, as all data is allocated in DRAM, there is exactly one copy of every shared data. All cores access the same copy. Note that the atomics used for reference counter decrements have release semantics associated. This is necessary to ensure that writes by child tasks are performed before the parent task can exit from the scheduling loop (i.e., reference counter reaches 0).

5.3 Scratchpad Enhanced Runtime

Prior work has shown that leveraging the scratchpad memory is critical to achieving peak performance on manycore architectures [CPZ+22]. However, scratchpad memories are often underutilized due to the high demand they put on programmers, in addition to the fact that not every workload is able to benefit from leveraging them (e.g., streaming workloads that do not have any reuse of input data). The naïve work-stealing runtime we introduced in Section 5.2 allocates all data, including both the stack and runtime data structures such as the task queues, in DRAM. While this naïve implementation yields a functionally correct work-stealing runtime, it is likely to have suboptimal performance due to high memory latency and contention at the LLC for applications that have frequent stack operations, task queue operations, or both. Instead of leaving the scratchpad memories unused, we introduce three optimizations which enable work-stealing runtimes to efficiently leverage scratchpads if they are not claimed by programmers. To the best of our knowledge, this is the first work that describes the implementation of a work-stealing framework which automatically utilizes scratchpad memories on manycore architectures.

Before the runtime can safely claim scratchpad space for its own, it has to know how much scratchpad space is reserved by programmers for user code. Reserving scratchpad space on Ham-

merBlade is realized through two APIs: (1) `spm_reserve()` and (2) `spm_malloc()`. `spm_reserve()` sets the maximum amount of scratchpad memory a core will use throughout execution. Programmers cannot reserve more space than what is available in the hardware (i.e., 4 KB). `spm_malloc()` returns a pointer to a chunk of memory allocated in the scratchpad. If the total amount of memory allocated/requested through `spm_malloc()` is larger than the amount set by `spm_reserve()`, it reports a failure by returning a null pointer. Our work-stealing runtime allocates a buffer at the top of the scratchpad as requested by the user, and automatically uses the scratchpad space that is not claimed by the user for both the stack and the task queue. By default, our runtime split the available space by reserving 512 B space for the task queue and the rest of space available to it for the task. However, we also provide APIs to allow experienced programmers to fine tune the runtime usage of the scratchpad. For example, the programmer can instruct the runtime to only scratchpad allocate the stack but not the task queue.

5.3.1 Scratchpad-Allocated Stack

Allocating stack in scratchpad memories has been mentioned and explored by various prior works in the literature [CPZ+22]. However, there are two main concerns on doing the same in the context of a work-stealing runtime: (1) user data can become shared variables; and (2) the stack can easily overflow the size of the scratchpad (e.g., recursively called runtime functions such as `wait()` and divide-and-conquer algorithms with deep recursion depth).

Data in the user code (e.g., `y` in line 14 of Figure 5.1 (a)) are potential shared variables and can be accessed by more than one core if the corresponding task `b` in line 16 is stolen. However, this is not an issue for manycore architectures which adopt the PGAS memory model (e.g., HammerBlade). The PGAS memory model allows every core to read and write any other core's scratchpad, and it enables us to keep a unique copy of data in a core's scratchpad. For example, assume `y` mentioned above is allocated in `core_0`'s scratchpad, and the corresponding task (i.e., `b`) is stolen by `core_1`. When `core_1` accesses `y` through the address taken at line 16 when creating the task, it performs a direct remote scratchpad access. The `y` in the scratchpad of the parent task's core remains as the only copy of `y`. The fully-strict properties of dynamic task parallelism (see Section 5.1) guarantees that reads and writes by `core_0` and `core_1` to `y` will not result in any data-race.

```
1  void task::spawn( task* t ) {
2    tq[tid].lock_aq()
3    tq[tid].enq(t)
4    tq[tid].lock_rl()
5  }
6
7  void task::wait( task* p ) {
8    while ( p->rc > 0 ) {
9      tq[tid].lock_aq()
10     task* t = tq[tid].deq()
11     tq[tid].lock_rl()
12     if (t) {
13       t->execute()
14       amo_sub_lr( t->p->rc, 1 )
15     }
16     else {
17       int vid = choose_victim()
18       tq[vid].lock_aq()
19       t = tq[vid].steal()
20       tq[vid].lock_rl()
21       if (t) {
22         t->execute()
23         amo_sub_lr( t->p->rc, 1 )
24       }
25     }
26   }
27 }
```

(a) Runtime Data in DRAM

```
1  void task::spawn( task* t ) {
2    spm_lock.lock_aq()
3    spm_tq.enq(t)
4    spm_lock.lock_rl()
5  }
6
7  void task::wait( task* p ) {
8    while ( p->rc > 0 ) {
9      spm_lock.lock_aq()
10     task* t = spm_tq.deq()
11     spm_lock.lock_rl()
12     if (t) {
13       t->execute()
14       amo_sub_lr( t->p->rc, 1 )
15     }
16     else {
17       int vid = choose_victim()
18       TaskQ* remote_tq =
19         get_remote_ptr(vid, &spm_tq)
20       QLock* remote_lock =
21         get_remote_ptr(vid, &spm_lock)
22       remote_lock->lock_aq()
23       t = remote_tq->steal()
24       remote_lock->lock_rl()
25       if (t) {
26         t->execute()
27         amo_sub_lr( t->p->rc, 1 )
28       }
29     }
30   }
31 }
```

(B) Runtime Data in Scratchpad

Figure 5.4: Work-Stealing Runtime Implementations – Pseudo-code of `spawn` and `wait` functions for: (a) having runtime data in DRAM; and (b) having runtime data in scratchpads. `tq` = array of task queues; `tid` = thread id; `lock_aq` = acquire lock; `lock_lr` = release lock; `rc` = ready count; `deq` = dequeue from the tail of the task queue; `enq` = enqueue to the tail of the task queue; `steal` = dequeue from the head of the task queue; `choose_victim` = random victim selection; `amo_sub_lr` atomic fetch-and-sub with release semantics; `spm_lock` = task queue lock allocated in scratchpad; `spm_tq` = task queue allocated in scratchpad; `get_remote_pointer` = calculate the address of a piece of data in another core's scratchpad.

(a) naïve runtime

(b) stack in SPM

(c) task queues in SPM

(d) queues and stack in SPM

Figure 5.5: Four implementations work stealing – (a) shows a naïve implementation in which stack and tasks queues are allocated in DRAM. (b) shows optimized stack placement relocated to SPM. (c) places the task queues in SPM while leaving the stack in DRAM. (d) applies both optimzations.

Manycore architectures like HammerBlade usually have limited per core scratchpad space (e.g., each core in HammerBlade has a 4KB scratchpad memory). Applications running recursive algorithms (e.g., divide-and-conquer) can easily create deep call stacks, which cannot fit in the scratchpad memory. When the stack does not fit, ideally we would like to keep the active and more recent frames (i.e., top frames) in scratchpad memory, since these frames are more likely to be accessed than older ones. To achieve this, one can either put the base of the stack in DRAM, and only start allocating in the scratchpad when the stack reaches a certain depth, or one can spill the older stack frames to DRAM when the scratchpad becomes full. However, both approaches have their caveats: starting in DRAM requires determining an ideal switching depth which can vary from workload to workload, while stack spilling cannot be realized without implementing complex hardware/software mechanisms. In this chapter, we opt for a simpler but less ideal solution: rather than keeping the top frames in scratchpads, we keep the bottom frames. When the stack overflows available SPM space, it automatically goes to DRAM, and we refer to this as *overflowing to DRAM*. While overflowing does happen, it only happens in applications with deep recursion depth. We optimize for the common case in which the stack can fit in scratchpads.

We leveraged a software/hardware co-design approach and extended each core with a lightweight hardware extension that snoops on the stack pointer register. We added two new control and status registers (CSRs): one for storing the DRAM overflow threshold (i.e., lowest address of the stack space in scratchpad), and the other for storing the pointer to DRAM overflow buffer. When a new frame is pushed onto the stack and the stack pointer is modified, we check if the stack is overflowed (i.e., new stack pointer has become smaller than the DRAM overflow threshold). If so, we replace the stack pointer with the pointer to the core's DRAM overflow buffer and allocating the new frame in DRAM. Similar checks and replacements are performed when a frame is popped off the stack. By default, the runtime allocates a 256 KB stack space for each core to enable deep recursion calls that can produce many stack frames. As we have mentioned before, the runtime calculates available stack space using the info given by programmers through `spm_reserve()`. It then allocates a buffer with proper size for each core in DRAM, and writes both the pointer of the DRAM allocated buffer and overflow threshold address to corresponding CSRs.

Although we chose to implement overflowing to DRAM on HammerBlade with a software/hardware co-design approach, the same functionality can also be easily implemented in software with modifications to the compiler on other manycore architectures where making hardware changes is

not feasible. Namely, we can change the compiler to insert instructions to check if the newly created stack fits in the scratchpad. If not, a pointer rewrite scheme can redirect the new stack pointer to the overflow buffer in DRAM. While the software solution involves adding extra instructions comparing to our software/hardware co-design approach, this check is light-weighted and the fast path (i.e., frames other than the frame that crosses the boundary) contains only two instructions: a load instruction for loading the overflow threshold address and a conditional jump which compares the stack pointer with the threshold address. The threshold address can and should be allocated in the scratchpad for low overhead access.

5.3.2 Scratchpad-Allocated Task Queue

A common goal of various parallel programming frameworks is to reduce the overhead of their runtimes. Our framework is not an exception. In the naïve runtime implementation, all runtime data structures, including the core local task queues, are allocated in the DRAM. Applications that have fine-grained tasks tend to induce frequent task queue operations as they generate more tasks than coarse-grained ones. For these applications, being able to operate the core local task queue efficiently is key to achieve high performance. The core local scratchpad has a 2-cycle access latency where the DRAM has an access latency of tens of cycles. Therefore, instead of going to DRAM for runtime data, we would like to keep them in the scratchpad memories for faster accesses.

Similar to what we have mentioned in Section 5.3.1, data coherence is not an issue as we keep only one copy of data and perform remote scratchpad accesses if the data is located in another core's scratchpad memory. However, unlike the user data which a pointer to it is passed around, a core must know the exact location of another core's task queue to conduct stealing without first accessing a DRAM allocated centralized data structures, such as the array of pointers to task queues (i.e., `tg[]` in Figure 5.4 (a)). Having such a DRAM allocated data structure diminishes the benefit of keeping stealing traffic away from DRAM. To achieve this, we reserve, by default, the top 512 B of the scratchpad for the core local task queue. The task queue is allocated at a fixed offset from the scratchpad base pointer across all cores. Therefore, if we have a pointer to the local task queue, we can easily calculate the pointer of the task queue of any other cores. Figure 5.4 (b) shows an implementation of `spawn` and `wait` for our runtime which has both the stack and runtime data structures in the scratchpad memories. The first noticeable difference is instead of loading the

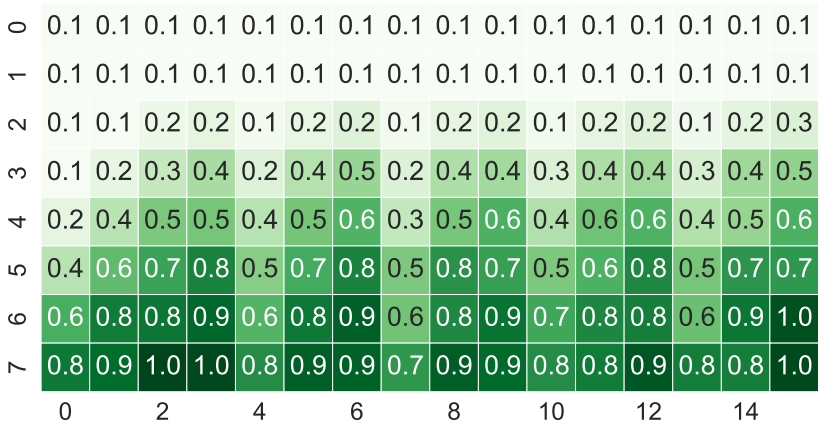


Figure 5.6: Normalized Remote Scratchpad Load Latency – Remote scratchpad load latency of 128 cores arranged in 16 rows and 8 columns, normalized to the core which has the highest latency.

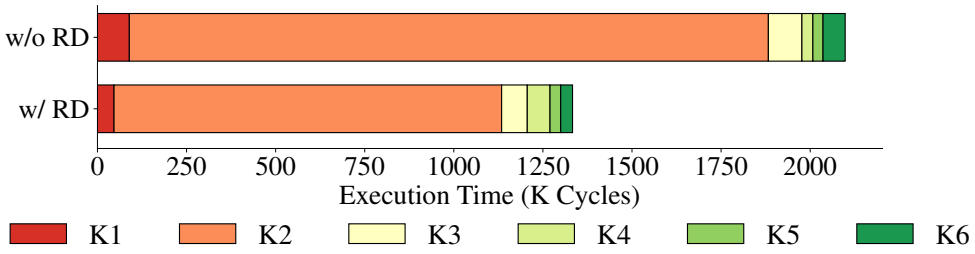


Figure 5.7: Performance Impact of Read-Only Data Duplication – Execution time of six parallel kernels (K1 to K6) in one iteration of PageRank with and without read-only data duplication optimization.

victim's queue from an array (line 18 in Figure 5.4 (a)), we calculate the address of victim's queue using the address of the local queue (lines 18–19 in Figure 5.4 (b)). We also separate the spin lock protecting the task queue from the queue itself (lines 2–4 in Figure 5.4 (b)). Doing so allows us to directly calculate the address of the remote spin lock (lines 20–21 in Figure 5.4 (b)): we do not need the remote scratchpad access for loading the pointer of the lock as in the case where the lock is a member of the task queue.

5.3.3 Read-Only Data Duplication

After implementing the two optimizations described above, profiling data collected from the one of the apps (i.e., Ligra-PR) shows an unexpected pattern. Figure 5.6 shows a heat map of normalized remote scratchpad access latency measured on each core in the 16×8 mesh. From the plot we can observe a clear pattern: cores that locate farther away from `core_0` (upper left corner) generally have longer remote scratchpad access latency. Note that, the distance in Y-direction has

a more significant impact than the distance in X-direction. This is because HammerBlade adopts X-Y routing and when all other cores trying accessing `core_0`, the bandwidth in Y-direction is much scarcer. The difference of latency within the same row is caused by the network topology of the 2-D mesh-with-ruching OCN [JDZ+20]. Our work-stealing runtime selects victims randomly, and thus we expect cores read and write their peer core's scratchpads uniformly and there should not be any hot spots.

A closer look at the profiling data revealed the causes: (1) when we implement the high-level templated functions (e.g., `paralell_for`), we keep a pointer to the user defined lambda function in the customized task class; and (2) in the user code, we write the lambda functions using `&` to capture values, including read-only values (e.g., pointers `dst` in line 5 of Figure 5.1 (d)), by references. On systems with hardware-base or software-centric coherence, these read-only data can be cached and reused. However, in our case, these values are all allocated on the scratchpad of `core_0`, and other cores repeatedly load from `core_0`. This traffic to `core_0` causes congestion in the OCN and long access latency. We resolve this issue by changing both the runtime and user code to duplicate read-only data that is allocated in the scratchpad (e.g., capture `dst` in Figure 5.1 (d) by value). We show the performance impact of the read-only data duplication optimization on *PageRank* in Figure 5.7. Each iteration of *PageRank* is composed of six parallel kernels. The proposed optimization is able to reduce execution time of all but one kernel, and achieve an overall speedup of 1.57×. Read-only data duplication applies to the case where the stack is DRAM allocated as well. It helps eliminate the hot spot in LLC in a similar manner as it eliminates the hot spot in `core_0`'s SPM. We enable this optimization for all work-stealing runtime configurations.

5.3.4 Micro-Benchmarking

We use *Fib*, a widely adopted micro-benchmark for demonstrating work-stealing runtimes in the literature, to illustrate the benefits of having the runtime leveraging the scratchpads. Figure 5.8 shows its implementation, and Section 5.4.1 provides details on the simulated hardware. *Fib* is known for generating significant amount of tasks that contain only minimal amount of compute. It yields both frequent stack operations (both runtime function calls and user-defined functor calls) and frequent task queue operations. We evaluate *Fib* on four variants of the runtime: both stack and task queue in DRAM which is the naïve implementation we introduced in Section 5.2, stack in DRAM and task queue in scratchpad, stack in scratchpad and task queue in DRAM, and both stack

```
int32_t fib_base(int32_t n) {
  if (n < 2)
    return n;
  else
    return fib_base(n-1) + fib_base(n-2);
}

int32_t fib(int32_t n, int32_t gsize = 2) {
  if (n <= gsize) {
    return fib_base(n);
  }

  int32_t x, y;

  parallel_invoke(
      [&] { x = fib(n-1, gsize); },
      [&] { y = fib(n-2, gsize); }
      );

  return x + y;
}

extern "C" __attribute__ ((noinline))
int kernel_fib(int* results, int n, int grain_size) {

  // output
  int32_t result     = -1;

  // --------------------- kernel ------------------------
  runtime_init();

  sync();

  if (__core_id == 0) {
    result = fib(n, grain_size);
    results[0] = result;
  } else {
    worker_thread_init();
  }
  runtime_end();
  // --------------------- end of kernel -----------------

  sync();
  return 0;
}
```

Figure 5.8: Fib Micro-Benchmark

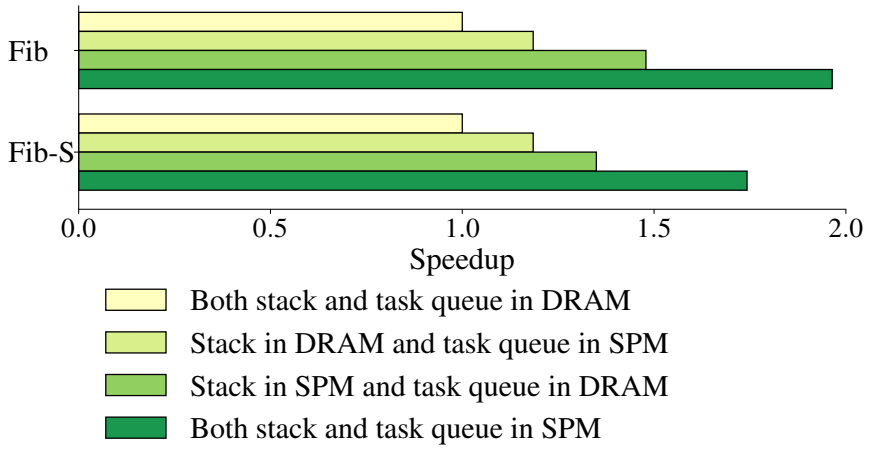


Figure 5.9: Speedup from Optimizing Data-Placement with SPM in Work-Stealing Runtime – Fib = measured speedups with the proposed SW/HW co-design scheme; Fib-S = estimated speedups with the 2-instruction SW-only scheme.

and task queue in scratchpad. Figure 5.5 illustrates the four variants and results are summarized in Figure 5.9. From the plot we can observe that, as we expected, the naïve runtime implementation has the worst performance. As we add optimizations and migrate either the stack or the task queue to scratchpad memories, we observe improved performance due to reduced access latency. Compared with task queue in SPM, stack in SPM shows better performance and it illustrates that having low latency access to the stack is more important for *Fib*. This is caused by: (1) the task queue is protected by a spin lock and the time spent on getting the lock, instead of accessing the task queue itself, dominates the execution time of pushing/popping task queues; and (2) stack operations (e.g., register spilling and saving/restoring saved registers) generate more traffic than task queue operations. Best performance is achieved when both optimizations are applied (i.e., both task queue and stack in SPM).

We also provide a first-order estimation on the impact of implementing the stack overflowing technique with the 2-instruction scheme in software (Section 5.3.1) by adding an additional 2-cycle delay to each `jal` and `ret` instructions. Results are illustrated in Figure 5.9 as *Fib-S*. We can observe that both configurations which have stack in SPM achieve less performance improvement for *Fib-S* than for *Fib* due to the overhead added by the extra instructions. However, both variants still perform significantly better than the naïve implementation. Note that *Fib* is close to the worst case for the potential software overflowing scheme, as it yields extensive amount of tasks and does little compute and thus frequent stack frame pushing/popping with short-living task body. In more

				Static Runtime				Work-Stealing Runtime							
				DRAM Stack		SPM Stack		DRAM Stack DRAM Queue		DRAM Stack SPM Queue		SPM Stack DRAM Queue		SPM Stack SPM Queue	
Cat	**Name**	**PM**	**Input**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**	**DI(M)**	**C(K)**
SB	MatMul	pf	256	37	543	37	512	38	527	39	556	39	573	38	509
			512	289	6914	289	6579	293	5049	295	5333	294	5321	297	5260
SU	PageRank	npf	g14k16	11	1586	11	1685	23	1649	24	1451	23	1425	25	1343
			email	11	5679	11	5384	27	1786	29	1638	24	1471	28	1358
			c-58	15	5136	15	5136	32	2257	40	2257	33	2044	38	1961
SU	BFS	npf	g14k16	3	1114	3	1062	22	1149	27	1102	21	914	26	871
			bundle1	6	1988	6	2065	30	1881	40	1892	29	1604	39	1561
			c-58	7	1943	7	1881	27	1852	35	1806	26	1495	33	1440
SU	SpMV	pf	bundle1	4	1483	4	1476	6	1005	7	995	6	1007	8	978
			email	2	4144	2	4129	95	4046	132	3820	87	3657	142	4060
			c-58	3	3442	3	3444	10	1047	14	1012	11	1019	15	1009
SU	SpMatrix Transpose	pf	bundle1	42	50850	42	50718	183	12877	281	13409	189	12911	279	12992
			email	22	47310	22	47343	1112	45864	1569	44351	1112	45456	1622	45391
			c-58	24	16570	24	16655	91	7568	123	7325	89	7222	129	7177
DB	Matrix Transpose	ss	512	–	–	–	–	3	496	3	502	3	416	3	421
			1024	–	–	–	–	8	2238	9	2240	8	2031	8	1969
DU	CilkSort	ss	16384	–	–	–	–	7	304	9	279	6	264	8	253
			131072	–	–	–	–	30	1799	31	1658	29	1305	32	1264
DU	NQueens	npf	8	4	1094	4	513	8	545	9	546	8	140	8	151
			9	19	5371	19	2522	36	2478	37	2508	37	910	37	1026
			10	100	24820	100	11691	177	11089	182	11381	181	6695	181	7367
DU	UTS	npf	small-t1	11	90684	11	90228	53	3266	71	3236	55	3280	71	3156
			small-t3	13	127199	13	126594	468	21028	663	21209	480	20878	680	20770

Table 5.1: Simulated Workloads – Cat = workload category; SB = static-balanced; SU = static-unbalanced; DB = dynamic-balanced; DU = dynamic-unbalanced; PM = parallelization methods; pf = `parallel_for`, npf = nested or recursive `parallel_for` and ss = recursive spawn and sync; Input = input dataset; DI = dynamic instruction count in millions; C = simulated cycles in thousands;

realistic workloads, we expect the overhead of the potential software overflowing scheme to be much less significant.

5.4 Evaluation Methodology

In this section, we describe our RTL-level cycle-accurate performance modeling methodology. We used this to quantitatively evaluate the proposed work-stealing runtime. We also give a brief introduction on the workloads we used in the evaluation.

5.4.1 Simulated Hardware

We model the HammerBlade manycore architecture using cycle-accurate RTL simulation. We leverage an RTL simulator (e.g., Verilator[2]) to model a silicon-validated small-scale early version of the HammerBlade manycore system running at 1.5 GHz with 16 columns and 8 rows (i.e., 128-cores in total). The RTL of this design has been validated in silicon. The DRAM timing is modeled with the timing-accurate open-source DRAMSim3 simulator [LYR+20]. We model a single 1.0 GHz HBM2 channel with a bus width of 64 and a burst length of 4, yielding a theoretical peak bandwidth of 16 GB/s. Performance counters are implemented with nonsynthesizable SystemVerilog `bind` statements. This allows us to conduct performance analysis without introducing any overhead to the workloads or modifying the digital logic design. See Section 2.3 for more details. We also made small changes to the load-store unit (LSU) of HammerBlade cores. The upstream LSU sends a request to the OCN when accessing a remote pointer that actually points to a piece of data on the core's own scratchpad and incurs a longer-than-necessary access latency in this case. We changed the LSU so that such accesses are handled directly by the core's scratchpad.

5.4.2 Runtimes

We conduct evaluation on both a traditional static runtime which supports only statically scheduled parallel loops and the proposed work-stealing runtime. We implement two variants of the static runtime, one variant has stacks allocated in DRAM and the other has stacks allocated in the SPM. We evaluate all four variants of the work-stealing runtime as in Section 5.3.4.

5.4.3 Workloads

We use a group of nine workloads to evaluate our proposed parallel programming framework, and the applications are summarized in Table 5.1. We select workloads with varied parallelization methods. *MatMul*, *SpMV*, and *SpMatrixTranspose* are dense matrix multiplication, sparse matrix dense vector multiplication, and sparse matrix transpose, respectively. All there workloads are implemented in-house and leverage a single parallel loop. *PageRank* and *BFS* implement pull-based PageRank and pull/push hybrid breadth-first search with the Ligra graph processing framework [SB13]. Both mainly use a pair of nested parallel loops: The outer loop iterates

[2]https://www.veripool.org/verilator/

```
void uts_v3_kernel( Node* parent, bool init = false )
{
  // Calculate how many children this node should have

  int numChildren, childType;

  numChildren = uts_numChildren( parent );
  childType   = uts_childType( parent );

  // Record number of children in parent
  parent->numChildren = numChildren;

  // Construct children and push them onto stack
  int parentHeight = parent->height;

  if ( numChildren > 0 ) {

    // Give a SHA-1 hash to each child

    // Define all nodes, args, tasks on the stack to avoid dynamic
    // memory management complexity. Need to put all definitions up here
    // so that they stay in scope. This function is parallelized with
    // run() and run_and_wait(). After run_and_wait() finishes, these
    // definitions go out of scope and are automatically cleaned up.

    // Node creation
    for ( int i = 0; i < numChildren; i++ ) {
      bsg_print_int(i);
      Node* child = (Node*)malloc( sizeof(Node) );
      initNode( child );
      child->height = parentHeight + 1;
      child->type   = childType;

      for ( int j = 0; j < computeGranularity; j++ ) {
        // computeGranularity controls number of rng_spawn calls per node
        rng_spawn( parent->state.state, child->state.state, i );
      }
    }

    // Process all children
    parallel_for( 0, numChildren, [&]( int i ) {
        uts_v3_kernel( &all_children[i] );
      } );
  }

  // No children
  return;
}
```

Figure 5.10: UTS Benchmark

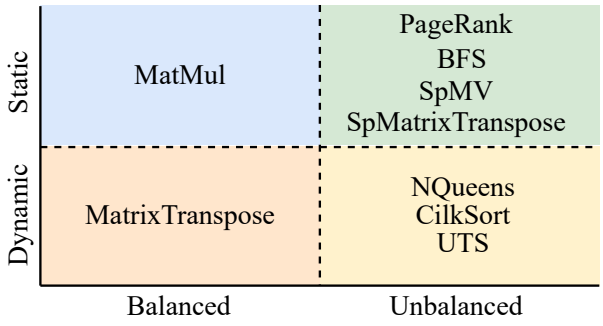


Figure 5.11: Anatomy of Workloads – We categorize workloads into four categories based on if he workload leverages dynamic parallelism and if the tasks have load imbalance

over vertices in the active vertex set while the inner loop iterates over a particular vertex's neighbors. Both *MatrixTranspose* and *CilkSort* mainly use recursive spawn-and-sync parallelization (i.e, `parallel_invoke`). *MatrixTranspose* is dense matrix transpose and *CilkSort* performs parallel mergesort. Both do not have static baseline implementations as spawn-and-sync parallelization starts with a single task. Without a dynamic runtime, their execution is serialized on a single core. *NQueens* uses backtracking to solve the N-queens problem. It is parallelized over the potential positions of the next queen to be placed on the board and contains recursive parallel loops. *UTS* is the Unbalanced Tree Search benchmark introduced by Olivier et al. [OHL+06], which contains recursive parallel loops to enumerate an unbalanced tree (see Figure 5.10. Among these nine workloads, only *MatMul*, which allocates a 3 KB buffer, utilizes SPM in user code. We characterize these nine workloads into four categories (i.e., *static-balanced*, *static-unbalanced*, *dynamic-balanced*, and *dynamic-unbalanced*) by two metrics: (1) if the workload leverages dynamic parallelism; and (2) if the tasks have load imbalance (see Figure 5.11).

5.5 Results

Table 5.1 summarizes the cycles and dynamic instruction counts of simulated configurations. Figure 5.12 shows speedup of workloads over a static runtime with stack in SPM. We plot *MatrixTranspose* and *CilkSort* separately in Figure 5.13, as they do not have static baselines. Comparing the left-most two bars in Figure 5.12, we can see that in the context of the static runtime, allocating the stack in SPM does not provide significant improvement over allocating the stack in DRAM, except in the case of *NQueens*. Workloads other than *NQueens* do not have frequent stack oper-

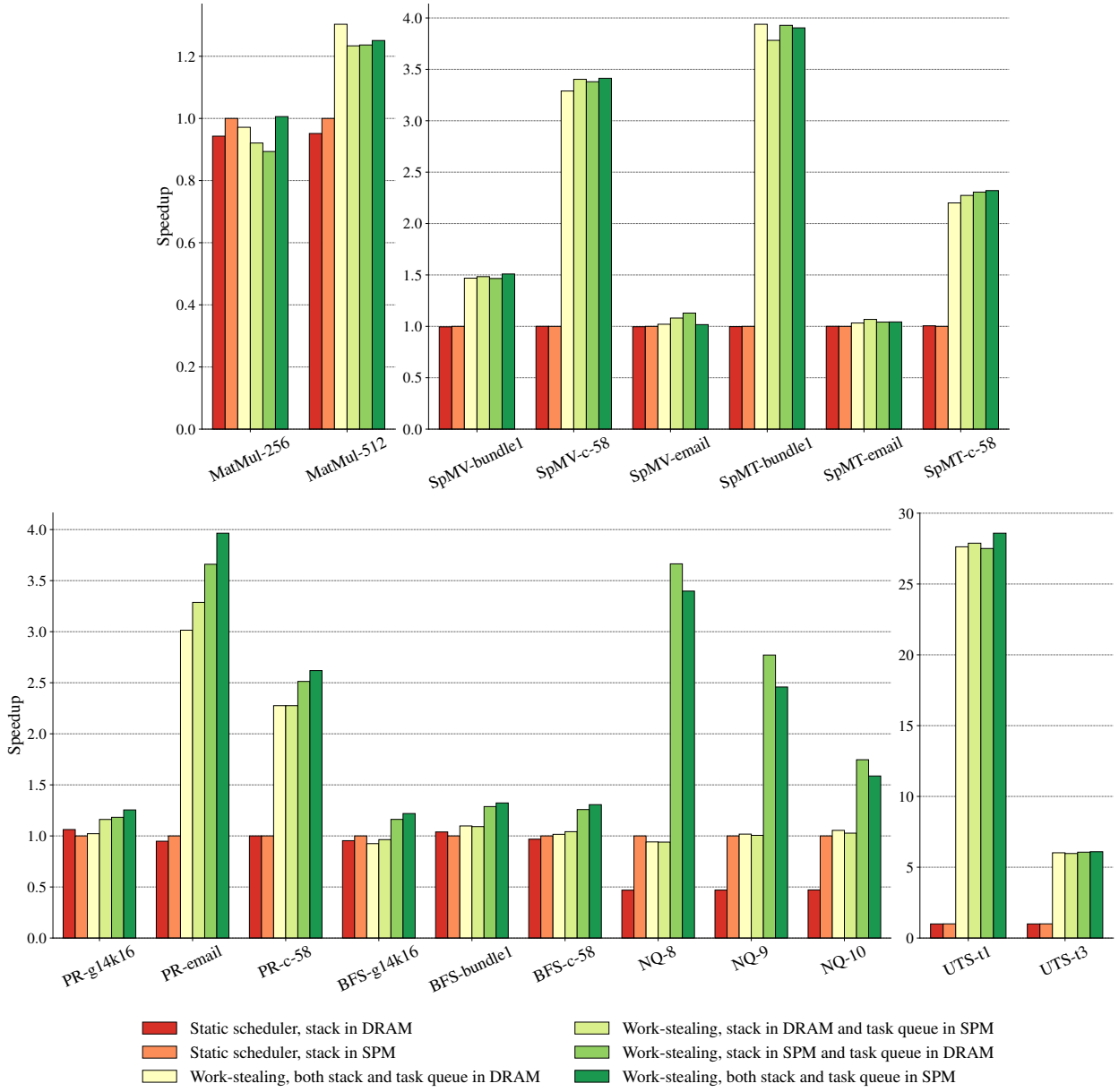


Figure 5.12: Speedup over Static Baseline with Stack in SPM – PR = PageRank, NQ = NQueens, SpMT = SpMatrixTranspose. The work-stealing runtime provides a speedup between 1.2 - 28× and a slowdown of no more than 10%. Applying data-placement optimizations to leverage the SPM provides an additional benefit of as much as 25% and compensates for any slowdown observed from work-stealing overhead.

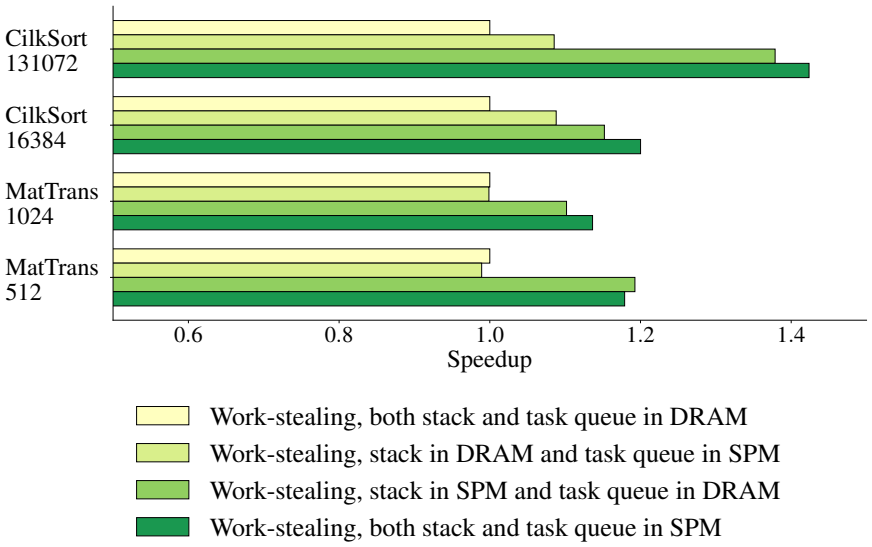


Figure 5.13: Performance of CilkSort and MatrixTranspose – Normalized to having both stack and task queue in SPM; MatTrans = MatrixTranspose. Note that the X-axis starts at 0.5.

ations when running with the static runtime, and thus leaving the stack in DRAM does not incur significant overheads. *NQueens* has heavy reads and writes to the stack as it frequently copies stack allocated arrays. Allocating the stack in DRAM leads to severe performance degradation.

Comparing static scheduler with stack in SPM with our baseline work-stealing runtime which has both the stack and the task queue in DRAM, we can observe that we either only incur minimal overheads over a traditional static runtime (e.g., in the cases of *MatMul-256* and *NQueens-8*) or achieve non-trivial performance improvement (e.g., *PR-email* and *UTS-t1* are able to achieve $3\times$ and $25\times$ better performance, respectively). This demonstrates the benefit of running irregular workloads with a work-stealing runtime on manycores. As expected, *PageRank*, *SpMV*, and *SpMatrixTranspose* show input dependent behavior and achieves different speedup on different inputs (e.g., *PageRank* shows only moderate speedup on the synthetic graph *g14k16*, but achieves $3\times$ speedup on real-world graph *email*). *MatMul* with 512×512 input matrices shows an unexpected 25% performance improvement over the static baseline. This is because while there is no inherent load imbalance in our tiled implementation, cores experience non-uniform memory latency due to their locations in the 2-D mesh OCN. Dynamic load-balancing helped mitigate this difference and scheduled more compute to cores with lower memory latency.

Different workloads benefit differently from our optimization techniques which leverages the SPM space not claimed by the programmer. *PageRank* is able to benefit from both optimizations

```
1  int ok(int n, char *a) {
2    int i, j;
3    int p, q;
4    for (i = 0; i < n; i++) {
5      p = a[i];
6      for (j = i + 1; j < n; j++) {
7        q = a[j];
8        if (q == p || q == p - (j - i) || q == p + (j - i))
9          return 0;
10     }
11   }
12   return 1;
13 }
14
15 void nqueens(int n, int j, char *a) {
16
17   if (n == j)
18     return;
19
20   /* try each possible position for queen <j> */
21   parallel_for( 0, n, 1, [&]( int i ) {
22       char b[j+1];
23       for ( int k = 0; k < j; k++ ) {
24         b[k] = a[k];
25       }
26       b[j] = i;
27       if ( ok(j + 1, b) ) {
28         nqueens(n, j + 1, b);
29       }
30     } );
31 }
32
33 extern "C" __attribute__ ((noinline))
34 int kernel_nqueens(int n) {
35   // --------------------- kernel ------------------------
36   runtime_init(dram_buffer);
37   sync();
38
39   if (__bsg_id == 0) {
40     char a[n];
41     nqueens(n, 0, a);
42   } else {
43     worker_thread_init();
44   }
45   runtime_end();
46   // --------------------- end of kernel -----------------
47
48   sync();
49   return 0;
50 }
```

Figure 5.14: NQueens Benchmark – NQueens involves frequent stack reads and writes as it allocates temporary buffers in the stack (i.e., lines 22–25)

and achieves best performance when both the stack and the task queue are in SPM. *BFS* can only outperform the static baseline with optimizations enabled, and SPM-allocated stack has a higher impact on *BFS* than SPM-allocated task queue. *NQueens* utilizes the stack heavily and achieves the best performance when the SPM is reserved solely for the stack (see Figure 5.14). In this configuration, fewer stack frames are overflowed to DRAM. We also observe that as the input size increases from 8 to 10, more moderate speedup is achieved by our work-stealing runtime compared to the static baseline. This is because larger inputs incur deeper stack and with more stack frames overflow to DRAM, *NQueens* becomes more DRAM bandwidth bound. *MatrixTranspose* and *CilkSort* are also able to benefit from having the stack in SPM (see Figure 5.13). *SpMV*, *SpMatrixTranspose*, and *UTS* do not have either frequent stack or frequent task queue operations. Moreover, both *SpMV* and *SpMatrixTranspose* are already DRAM bandwidth bounded. Extra traffic to DRAM incurred by allocating both stack and task queue in DRAM has only insignificant impact. As a result, our optimizations do not yield better performance.

Across all workloads, we observe an increment in the number of dynamic instructions on work-stealing runtimes v.s. on static runtimes (see Table 5.1). This is expected as it is well-known that the work-stealing runtime adds overheads from various sources (e.g., task creation and scheduling), especially when with very fine-grained tasks. We also observe an increment in the number of dynamic instructions when SPM-allocated task queue optimization is enabled. This is because with reduced task queue access latency, cores can perform stealing attempts faster and fail more when there is no task to steal. These instructions are executed by idle cores that cannot find ready tasks and they are not part of the critical path.

We also conduct a scalability study with five workloads: one workload from each workload category and the *Fib* micro-benchmark we used in Section 5.3.4. Results are illustrated in Figure 5.15. While being a micro-benchmark, *Fib* shows that our work-stealing runtime scales well and does not incur significant overhead when scaled to 128 cores. *MatMul* also shows good scalability it has high arithmetic intensity and only load from DRAM infrequently. *PageRank* and *MatrixTranspose* are both memory intensive and their scalability is highly limited by memory bandwidth. *NQueens* scales the best as with more cores, more stack allocated data can be kept in SPM.

To summarize, the proposed work-stealing runtime: (1) either improves performance of static-balanced workloads by migrating tasks away from cores that have long memory latency or induces only minimal overheads; (2) improves performance of irregular workloads which show input de-

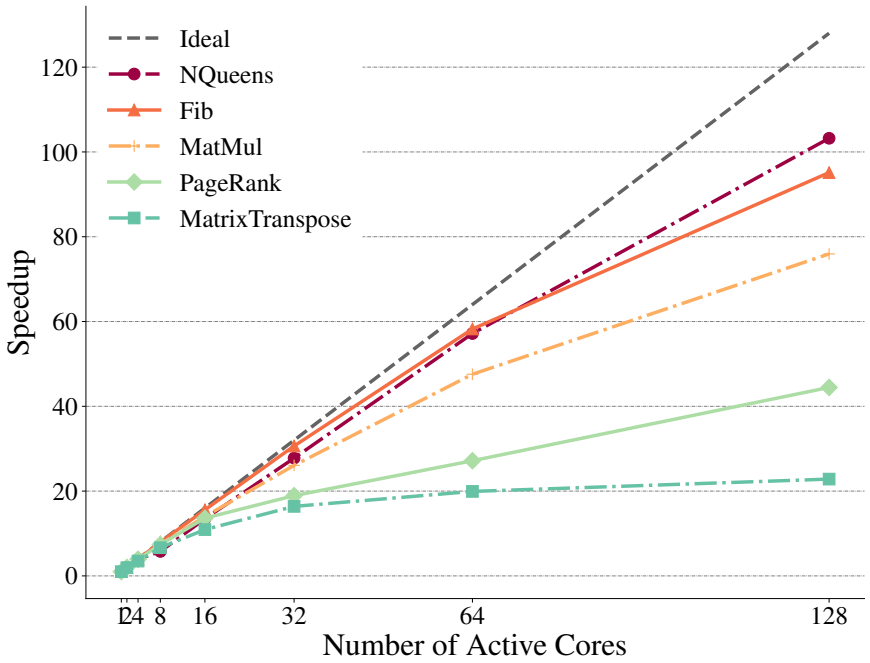


Figure 5.15: Scalability of Workloads – Inputs: Fib = 24; MatMul = 256; PageRank = g14k16; MatrixTranspose = 512; NQueens = 8. Data collected on work-stealing runtime with both task and task queue in SPM.

pendent behavior when there is input induced load imbalance; (3) efficiently supports dynamic-balanced and dynamic-unbalanced workloads to achieve high performance, and (4) provides high scalability. Our proposed optimization techniques which automatically leverage SPM are able to improve performance of applications that have frequent stack and/or frequent task queue operations (i.e., *NQueens*, *MatrixTranspose*, *PageRank*, and *BFS*) and incur only minimal overheads on workloads that cannot benefit from them.

5.6 Related Work

A number of prior work explored implementing work-stealing runtimes on manycore architectures that provide software-centric cache coherence. Long et al. [LZF08] implemented a Cilk-like runtime on a 64-core manycore architecture with a shared L2 cache and non-coherent private L1 caches. They attacked the shared data coherence issue by leveraging a bloom filter based hardware mechanism, Coherence Vector, to identify memory locations that should not be cached in non-coherent private L1 caches. The proposed runtime register all runtime-related shared data (e.g., task queues) into the Coherence Vector. For user data that may have parent-child dependency, they

exploit the DAG-consistency [BFJ+96] and insert L1 invalidate and L1 write-back instructions in the runtime. Similarly, Wang et al. [WTCB20] worked on a similar system (i.e., big.TINY) also proposed to insert L1 cache invalidation and write-back instructions at proper locations in their Cilk-like runtime. Unlike Long et al. who identifies runtime shared data as non-cacheable locations, Wang et al. proposed to leverage the same self-invalidation and self-flush mechanism for keeping runtime shared data coherent. For example, after locking a task queue, a core performs a L1 cache invalidation to avoid reading stale data when accessing the task queue. After push/popping the task queue, a core write dirty data in its L1 cache before releasing the lock on the corresponding task queue. Doing so ensures the core's changes to the task queue is visible to other cores. To mitigate the frequent L1 cache invalidation and write-back induced by task queue operations, Wang et al. proposed a hardware-based mechanism, direct task stealing, which makes task queue a private data structure. Stealing is made possible by having the thief sending a user-level interrupt to the victim. The victim then pops a task from its task queue on behalf of the thief. Tagliavini et al. [TCM18] implemented an OpenMP runtime on a manycore architecture that has non-coherent private L1 caches. Similar to both work mentioned above, the private L1 caches need to be self-invalidated and self-flushed at proper time to maintain coherence. Unlike the two Cilk-like runtimes that have per thread task queues, their proposal leverages a centralized task queue. All three work studies manycore architectures with software-centric cache coherence, while our work targets architectures that have only software-managed scratchpads.

Zakkak et al. [ZP16] proposed an implementation of Java virtual machine for manycore architectures with only software-managed scratchpads. However, their main focus is on managed memory and synchronization primitives. For thread scheduling, they proposed to use work-dealing instead of work-stealing among non-coherent components. Our work, to the best of our knowledge, describes the first implementation of a Cilk-like work-stealing runtime for manycore architectures with only software-managed scratchpads. Alvarez et al. [AMC+15] described a task-based parallel runtime which can transparently leverage the scratchpad memories for holding input and output data in a hybrid memory hierarchy. Prior work also studied work-stealing runtimes on PGAS or distributed memory clusters, including [DLS+09, PCM+07, SKK+11]. Li et al. [LDCT10] studies efficient implementation of conditional division on manycore architectures. Unlike traditional Cilk-like runtimes which splits a task until a predetermined task granularity is reached, conditional division [PLT06] splits a task only if there is an idle core to accept the newly created child

tasks. Their work focuses on improving the work scheduling efficiency on top of an existing work-stealing runtime and is orthogonal to ours. Conditional division can be applied on top of our proposed work-stealing runtime for improved scheduling efficiency and performance. Chen et al. [CSBS18] and Margerm et al. [MSG+18] explored generating task parallel accelerators while assuming coherent caches. Our work can be applied to support accelerators with SPMs.

5.7 Conclusion

This chapter demonstrated that, in opposite to conventional wisdom, a work-stealing runtime is both viable and beneficial on manycore architectures with only software-managed scratchpad memories. This chapter provides programmers a familiar programming model and interface for efficiently developing new software and porting existing software on manycore architectures like HammerBlade, and achieves significant performance improvements over traditional programming models such as statically scheduled parallel loops (i.e., up to 3.94× speedup for workloads that can be statically scheduled and up tp 28.5× speedup for workloads that leverage dynamic parallelism). This work is a small yet important step towards solving the manycore architecture programmability challenge. While the work-stealing runtime is evaluated on the HammerBlade manycore architecture, the general idea is applicable to other PGAS manycore architectures that have software-managed scratchpad memories.

CHAPTER 6
CONCLUSION

Technology constraints continue to drive computer architects to increase parallelism. Many-core processors have been increasingly popular in modern computing platforms. However, while thread-parallel focused manycore architectures have been proposed and fabricated both in academia and in industry since early 2000s, their audience is still limited to a small group of experts who have deep understanding of both the workload and the underlying manycore hardware after nearly two decades of research and development. This is mainly due to their low-level programming interfaces, their unfamiliar programming model, and their need for software optimization to realize the promised high performance. A key research challenge remains: how to facilitate programming on such architectures. As the manycore architectures keep scaling up their core counts and start adopting software-managed scratchpad memories, this manycore architecture programmability challenge has also become more important and more challenging to solve. This thesis took inspiration from the success of domain-specific frameworks on data-parallel manycore architectures (e.g., GPGPUs) and the success of general-purpose dynamic task parallel programming frameworks on multi-core processors, and propose both kind of frameworks to address the SPM manycore architecture programmability challenge. This thesis illusrates that future manycore architecture can safely opt for SPMs without worrying about losing programmability, if the correct programming model and framework are adopted. The rest of this chapter summarizes primary contributions of this thesis and discusses future research directions.

6.1 Thesis Summary and Contributions

This thesis began by discussing the adoption of the manycore approach. I presented a brief survey of both thread-parallel focused and data-parallel focused manycore architectures. The survey showed a trend of abandoning hardware-based coherent caches for software-centric coherent caches and software-managed scratchpad memories. Manycore architectures with hardware-based coherent caches typically have at most a hundred cores. Adopting software-centric coherent caches also cannot push the core count beyond a few hundreds. To reach over a thousand cores in a single chip, hardware designers from both academia and industry converged on software-managed

scratchpad memories. However, abandoning coherent caches for software-managed scratchpads prevents programmers from using traditional multi-core shared-memory programming models, making both reusing existing software and writing new software significantly more challenging on SPM manycore architectures. I then discussed examples of the domain-specific approach, which helped facilitate the wide adoption of data-parallel focused manycore architectures, especially GPGPUs. These frameworks provide hand-optimized domain-specific operators and leverage domain-specific knowledge to enable more optimizations and realize high performance. I also discussed the dynamic task parallel programming frameworks that thrived in the multi-core era. These frameworks usually adopt the fork-join computation model which naturally describes task parallelism. This chapter motivated the necessity of resolving the SPM manycore programmability challenge and pointed to two approaches that have had success on attacking similar challenges on other compute platforms.

This thesis then provided a brief but thorough discussion on the HammerBlade manycore architecture developed and implemented in RTL by BSG at the University of Washington. The thesis provided details on both the hardware and the software of the HammerBlade manycore. Programming HammerBlade is done through its low-level C runtime named CUDA-lite, which adopts a SPMD execution model. I used a parallel reduction kernel as an example to demonstrate CUDA-lite. I then introduced our RTL cycle-level evaluation methodology. Lastly I conducted a case study with the widely used matrix multiplication kernel to illustrate the complexity of hand tuning a kernel on the HammerBlade manycore with CUDA-lite, which involves unrolling, tiling, manual instruction scheduling, and manual register allocation.

`HB-PyTorch` is a domain-specific framework. This thesis extended `PyTorch`, a widely used open-source tensor processing framework, with a HammerBlade backend. We implemented and hand-optimized both dense and sparse tensor processing operators that are essential for running a wide range of existing deep learning workloads on the HammerBlade manycore without modifying the model code. Beside enabling easy reuse of existing deep learning models, the tensor processing framework proposed in the thesis can be used to express other workloads as well. Programmers can rewrite their workloads with the provided operators and achieve high performance on the HammerBlade manycore without knowing anything about HammerBlade itself. First-order estimation of three dense and sparse tensor workloads showed that we are able to achieve much

higher performance on the full scale 2000-core HammerBlade architecture than on an aggressive multi-core CPU baseline.

We used selected `HB-PyTorch` operators as microbenchmarks and identified that memory latency is the key limiting factor even for operators that have high arithmetic intensity. This finding motivated us to explore software-enabled hardware-accelerated decoupled access/execute and systolic execution on SPM manycore architectures with other domain-specific frameworks `HB-Arc`.

`HB-Rubick` is a general-purpose dynamic task parallel programming framework. This thesis provided, to the best of my knowledge, the first detailed description of how to extend a work-stealing runtime to run on SPM manycore architectures. In contrast to conventional wisdom, in this chapter we demonstrated that it is not only viable but also beneficial to implement a work-stealing runtime on SPM manycore architectures. The proposed work-stealing runtime improved performance of irregular workloads, and enabled programmers to express algorithms that leverage dynamic parallelism. Moreover, `HB-Rubick` provides programmers a familiar programming interface which eases the reuse of existing code that are written for traditional multi-core processors as well as the development of new software. `HB-Rubick` also included three optimizations that can leverage the unused scratchpad memories for better performance.

The primary contributions of this thesis are reiterated below:

- an open-source tensor processing framework, which achieves high performance on SPM manycore architectures;
- a novel framework, which enables decoupled access/execute (DAE) and systolic execution on SPM manycore architectures;
- an open-source dynamic task parallel programming framework, which supports arbitrarily nested parallel patterns and dynamic load balancing on SPM manycore architectures; and
- software and hardware optimizations to enable a work-stealing runtime to leverage unused scratchpad space and achieve higher performance on SPM manycore architectures.

6.2 Future Work

The techniques presented in this thesis are first steps towards closing the programmability gap of SPM manycore architectures. There are many opportunities to build on the ideas presented in this thesis. This section discusses some promising research directions for future work.

6.2.1 Improving SPM Utilization

Motivation – Accessing any data that is not allocated in a core's SPM incurs traffic to the LLC. As the number of cores in a single chip keeps increasing, each core receives less and less per-core LLC bandwidth. Thus, keeping as much data as possible in the core local SPM is critical to avoid hotspots in LLC, avoid congestion in the on-chip network, and achieve high performance in SPM manycore architectures that have hundreds to thousands of cores.

Research – This research topic has two directions. One direction is to extend the optimizations that allowed `HB-Rubick` to automatically leverage unused SPM space for storing stack and task queues. As I have discussed in Chapter 5, newer stack frames overflowing to DRAM is the main reason why we observe less speedup as we increase the input of *NQueens*. One way to improve the performance of workloads like *NQueens* that have heavy loads on the stack is to keep newer stack frames, instead of older ones, in the SPM. Like I have mentioned earlier in this thesis, it is possible to create software/hardware co-design mechanisms which automatically spills old stack frames to DRAM. While this approach seems promising, a few open research questions remain: (1) what should be the spilling granularity? One could create a mechanism which works like a two-entry software cache in the SPM. When the stack space in SPM fills up, one can spill the older half to DRAM. One can also create a mechanism which implements a sliding window, which only spills enough data to DRAM to make space for the newly created stack frame. The tradeoff here is the complexity of this spilling mechanism and the overhead of spilling. A simple scheme that spills in large chunks may be also simple to realize with simple hardware or can be implemented easily in software only, but it incurs longer pauses when copying data to DRAM. A complex scheme can avoid these long pauses but may be trickier to implement. And (2) will this spilling mechanism create deadlocks? When a spilling is in progress, all remote SPM accesses to the core must be buffered. In the case where the buffer in the core is full and creates back pressure to the on-chip

network, the spilling from this core can also be stalled, causing a potential deadlock: incoming remote SPM access requests cannot be fulfilled until spilling is done, but these requests congest the OCN and prevents spilling from finishing. Another direction is to explore ways to either make leveraging SPM simpler in user code or automatically detect data in user code that can be SPM allocated.

6.2.2 Scaling to Full-Scale SPM Manycore Architectures

Motivation – While `HB-PyTorch`, `HB-Arc`, and `HB-Rubick` are all proposed to solve the programmability challenge of SPM manycore architectures, we were unable to evaluate these proposals on a full-scale SPM manycore architecture due to simulation speed. With current RTL simulation tools, it is not feasible to simulate the full-scale HammerBlade manycore architecture which has over 2000 cores. If the frameworks I presented in this thesis can achieve good performance on the full-scale HammerBlade system remains unclear. It is very likely that additional optimizations are necessary.

Research – A number of research directions can be explored, including simulators that can facilitate the cycle-level modeling of the full-scale HammerBlade system, memory models of such system, and new optimizations in `HB-PyTorch`, `HB-Arc`, and `HB-Rubick` that help achieving high performance. Key concerns when applying `HB-Rubick` to a 2000-core system are determining optimal task granularity and mitigating work discovery overhead. As the number of cores increases, one can easily run into the case where the workload cannot provide enough parallelism to leverage all the available cores. Then one open research question is: how do we determine the best task granularity? Having each core running only one iteration of the parallel loop may not yield the best performance as runtime overhead can easily dominate the execution time. It is possible that in these cases only cores that are closer to LLC banks should be active. Even in the case where there is enough parallelism, work discovery overhead becomes a problem when scheduling with hundreds to thousands of cores. All parallel patterns in `HB-Rubick` are invoked from a single core (i.e., `core_0` in examples presented in Chapter 5). Having hundreds, even thousands, of idle worker cores trying random stealing from all other 2000 cores is not likely to provide the best performance. One can imagine that we would need either a jump start mechanism which distributes the newly created tasks to idle cores or hierarchical stealing schemes to reduce cross-chip remote

accesses, or both. A hardware-based task distribution network could be a feasible option. Another aspect one might want to explore is the placement of the master core. In this thesis `core_0`, which is located at the top left corner of the 128-core chip, acts as the master core. In the full-scale system, one may want to place the master core in the middle of the grid for better work distribution.

6.2.3 Cooperative Execution with Cache Coherent Multi-Core CPUs

Motivation – Both frameworks I presented in this thesis assumed an offloading execution model, in which a host CPU launches the computation on the HammerBlade manycore. Execution on the host CPU is blocked until the kernel finishes on HammerBlade. This offloading model forces programmers to reason about which device is more suitable to run a certain workload, and it results in underutilized computing resources since a workload cannot leverage both the host CPU and the HammerBlade device at the same time.

Research – Enabling cooperative execution in `HB-PyTorch` is not simple. `PyTorch`'s dynamic dispatching mechanism, which we briefly introduced in Chapter 3, relies on knowing where a piece of data is allocated. If a tensor is allocated on the HammerBlade manycore, it calls the implementation in the HammerBlade backend, and if a tensor is allocated on the host CPU, native CPU backend is called. To implement cooperative execution, one would need to either let the host CPU and the HammerBlade device to have unified memory, or implement transparent data movement schemes to copy data from host to HammerBlade or vice versa on demand. Cooperative execution on `HB-Rubick` is more interesting and also more challenging. The key research question is how do we allow the host CPU, which has traditional hardware-based coherent cache, and the HammerBlade manycore, which has software-managed SPMs, to steal from each other while achieving high performance? One simple idea is to mark cachelines that will be shared with HammerBlade manycore as non-cacheable. However, the concern here is that the host CPU could be running at a much higher frequency and is able to make remote SPM access requests much faster than a HammerBlade core can. The HammerBlade manycore could be hammered by these requests and have suboptimal performance. Also, off the shelf multi-core processors usually do not support marking non-cacheable cachelines, which implies that hardware changes are necessary. Another challenge is to construct a simulation infrastructure that allows us to conduct such cooperative execution experiments. The current co-simulation infrastructure I introduced in Chapter 2 uses

the native x86 CPU as the host CPU. To simulate cooperative execution, one would need to also simulate a multi-core processor, either also in RTL or co-simulated in a cycle-level simulator such as gem5 [BBB+11].

6.2.4 Cooperative Execution with Accelerators

Motivation – Parallelism and specialization have been the two main techniques for turning the ever increasing number of transistors provided by Moore's Law into performance. This thesis went down the path of parallelism. However, the techniques I presented in `HB-Rubick` can potentially be applied to the specialization approach as well. Specialized hardware, for instance application specific integrated circuit (ASIC) and domain-specific accelerators, usually adopts scratchpad memories and relies on direct memory access (DMA) engines to move data in and out local SPMs. As a result, most of such specialized hardware adopts the offloading execution model and leaves the host multi-core CPU underutilized.

Research – There are two directions in this research topic. One is to build on top of the potential research I discussed in Section 6.2.3. If we can implement a technique which allows cooperative execution of SPM manycore architectures and cache coherent multi-core CPUs, we can extend the same technique to work with ASICs and domain-specific accelerators that leverage SPMs. Another direction is to eliminate the necessity of hardware coherent caches in task parallel accelerators. Such accelerators have been proposed by Chen et al. [CSBS18] and Margerm et al. [MSG+18]. However, both of them require hardware coherent caches to facilitate work-stealing on their generated accelerators. The work I presented in Chapter 5 demonstrated that such work-stealing runtimes can be efficiently implemented on SPM manycore architectures. It is very likely that work-stealing through direct remote SPM access is also feasible and beneficial on task parallel accelerators that adopt SPMs. Being able to remove hardware coherent cache from such accelerators can significantly simplify the memory system design of both the overall system and accelerator itself.

6.2.5 Supporting Dynamic Languages on SPM Manycore Architectures

Motivation – Throughout the decades, as processor performance increases, programmability as well as productivity have become increasingly important when choosing programming languages for projects of all sizes. Dynamic programming languages, such as Python, JavaScript, Ruby, Smalltalk, and PHP, have been among the most popular programming languages for the past years because of their flexibility and ease of use [Cas18, Fog18]. However, how to efficiently support them on emerging compute platforms such as SPM manycore architectures remains mostly unexplored.

Research – One possible way to allow dynamic language programmers to leverage SPM manycore architectures is through just-in-time (JIT) compilation [Bol12]. We could potentially extend the widely adopted JIT compiler for GPGPUs, Numba [num19], to include a `HB-Rubick` backend. Programmers can write their applications in Python and have Numba compile the Python code to native code which leverages `HB-Rubick`. However, in this case the compiler must be able to statically type check the Python source code and this limitation forces programmers to use only a restricted subset of Python to express their applications. Dynamic features that made Python productive and flexible are generally not allowed and how to support them remains an open research question.

BIBLIOGRAPHY

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale machine Learning. *Symp. on Operating System Design and Implementation (OSDI)*, Nov 2016.

[ACD+09] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 20(3):404–418, Mar 2009.

[AMC+15] L. Alvarez, M. Moretó, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguadé, and M. Valero. Runtime-Guided Management of Scratchpad Memories in Multicore Architectures. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2015.

[BBB+11] N. Binkert, B. 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.

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

[BFJ+96] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 1996.

[BFY+21] A. Brahmakshatriya, E. Furst, V. Ying, C. Hsu, C. Hong, M. Ruttenberg, Y. Zhang, D. C. Jung, D. Richmond, M. Taylor, J. Shun, M. Oskin, D. Sanchez, and S. Amarasinghe. Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2021.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Jul 1995.

[BJK+96] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug 1996.

[BL99] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, Sep 1999.

[Bol12] C. F. Bolz. *Meta-Tracing Just-In-Time Compilation for RPython*. Ph.D. Thesis, Mathematisch-Naturwissenschaftliche Fakultät, Heinrich Heine Universität Düsseldorf, 2012.

[BS13] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-Defined Caches. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2013.

[BSL+02] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, May 2002.

[BSP+17] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, Apr 2017.

[Cas18] S. Cass. The 2018 Top Programming Languages. *IEEE Spectrum*, Jul 2018.

[CDS+14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2014.

[CGS+05] P. Charles, C. Grothoff, V. Sarkar, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 2005.

[CHA+15] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. The Load Slice Core Microarchitecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[CHM11] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An in-Fabric Memory Architecture for FPGA-Based Computing. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2011.

[CIBTB20] L. Cheng, B. Ilbeyi, C. F. Bolz-Tereick, and C. Batten. Type Freezing: Exploiting Attribute Type Monomorphism in Tracing JIT Compilers. *Int'l Symp. on Code Generation and Optimization (CGO)*, Feb 2020.

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

[CLS05] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2005.

[CMJ+18] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. *Computing Research Repository (CoRR)*, arXiv:abs/1802.04799, Aug 2018.

[CPZ+22] L. Cheng, P. Pan, Z. Zhao, K. Ranjan, J. Weber, B. Veluri, S. B. Ehsani, M. Ruttenberg, D. C. Jung, P. Ivanov, D. Richmond, M. B. Taylor, Z. Zhang, and C. Batten. A Tensor Processing Framework for CPU-Manycore Heterogeneous Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1620–1635, 2022.

[CSBS18] T. Chen, S. Srinath, C. Batten, and E. Suh. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2018.

[cuv22] CUDA Vision & Imaging Library. Online Webpage, 2022 (accessed Sept 27, 2022). `https://cuvilib.com`.

[DKM+12] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: Recording Microprocessor History. *ACM Queue*, page 10–27, Apr 2012.

[DLS+09] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2009.

[DM97] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. *Int'l Symp. on Supercomputing (ICS)*, Jul 1997.

[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.

[FLKBF11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-Core Systems. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

[FLR98] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1998.

[Fog18] A. Fog. Instruction Tables. Online Webpage, Sep 2018. `http://www.agner.org`.

[FOS+14] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. *Symp. on FPGAs for Custom Computing Machines (FCCM)*, May 2014.

[FW15] Y. Fu and D. Wentzlaff. Coherence Domain Restriction on Large Scale Systems. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2015.

[GHF+06] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, Mar 2006.

[GTA06] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2006.

[Gwe11] L. Gwennap. Adapteva: More Flops, less Watts. *Microprocessor Report, The Linley Group*, 6(13):11–02, 2011.

[Hal20] T. R. Halfhill. ThunderX3's Cloudburst of Threads: Marvell Previews 96-core 384-thread Arm Server Processor. *Microprocessor Report, The Linley Group*, Apr 2020.

[HAM15] T. J. Ham, J. L. Aragón, and M. Martonosi. DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2015.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2010.

[HVS+07] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor. *IEEE Micro*, 27(5):51–61, Sep/Oct 2007.

[HZRS15] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *Computing Research Repository (CoRR)*, arXiv:abs/1512.03385, Dec 2015.

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

[int19] Intel Threading Building Blocks. Online Webpage, 2015 (accessed Nov 2019). `https://software.intel.com/en-us/intel-tbb`.

[J+17] N. P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[JDZ+20] D. C. Jung, S. Davidson, C. Zhao, D. Richmond, and M. B. Taylor. Ruche Networks: Wire-Maximal, No-Fuss NoCs : Special Session Paper. *Int'l Symp. on Networks-on-Chip (NOCS)*, Apr 2020.

[kal22] Kalray MPPA Products. Online Webpage, 2022 (accessed Aug 2022). `https://www.kalrayinc.com/products/mppa-technology/`.

[Kan15] D. Kanter. Knights Landing Reshapes HPC, Sep 2015.

[KFP+18] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun. Spatial: A Language and Compiler for Application Accelerators. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2018.

[KSA+15] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have Your Scratchpad and Cache It Too. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[LDCT10] Z. Li, J. Duato, O. Certner, and O. Temam. Scalable hardware support for conditional parallelization. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.

[Lei09] C. E. Leiserson. The Cilk++ Concurrency Platform. *Design Automation Conf. (DAC)*, Jul 2009.

[LFF+18] L. Li, J. Fang, H. Fu, J. Jiang, W. Zhao, C. He, X. You, and G. Yang. swCaffe: A Parallel Framework for Accelerating Deep Learning Applications on Sunway TaihuLight. *Int'l Conf. on Cluster Computing*, Sep 2018.

[LKK+11] J. Lee, J. Kim, J. Kim, S. Seo, and J. Lee. An OpenCL Framework for Homogeneous Manycores with No Hardware Cache Coherence. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2011.

[LSC+13] M. Lis, K. S. Shim, M. H. Cho, I. Lebedev, and S. Devadas. Hardware-Level Thread Migration in a 110-Core Shared-Memory Multiprocessor. Technical Report 512, MIT CSAIL CSG, Nov 2013.

[LYR+20] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *Computer Architecture Letters (CAL)*, Jul 2020.

[LZF08] G.-P. Long, J.-C. Zhang, and D.-R. Fan. Architectural support and evaluation of Cilk language on many-core architectures. *Chinese Journal of Computers*, 31(11):1975–1985, 2008.

[MCJ+18] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. VTA: An Open Hardware-Software Stack for Deep Learning. *Computing Research Repository (CoRR)*, arXiv:abs/1802.04799, Aug 2018.

[MCP+12] B. Marker, E. Chan, J. Poulson, R. van de Geijn, R. F. Van der Wijngaart, T. G. Mattson, and T. E. Kubaska. Programming many-core architectures-a case study: dense matrix computations on the Intel single-chip cloud computer processor. *Concurrency and Computation: Practice and Experience*, 24(12):1317–1333, 2012.

[MCV+19] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro*, Sep 2019.

[MFN+17] M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, J. Balkind, A. Lavrov, M. Shahrad, S. Payne, and D. Wentzlaff. Piton: A Manycore Processor for Multitenant Clouds. *IEEE Micro*, 37(2):70–80, Mar/Apr 2017.

[MHDmoc19] D. R. MacIver, Z. Hatfield-Dodds, and many other contributors. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open-Source Software (JOSS)*, 4(43), Nov 2019.

[MHS12] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Communications of the ACM*, Jul 2012.

[MKP05] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for Efficient Processing in Runahead Execution Engines. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2005.

[Mos05] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2005.

[MRR12] M. McCool, A. D. Robinson, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[MSG+18] S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam. TAPAS: Generating Parallel Accelerators from Parallel Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2018.

[MSWP03] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2003.

[num19] Numba. Online Webpage, accessed Nov 2019. `http://numba.pydata.org`.

[nvi20] NVIDIA A100 Tensor Core GPU Architecture. NVIDIA White Paper, 2020. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`.

[nvi22] CUDA Toolkit. Online Webpage, 2022 (accessed Sept 23, 2022). https://developer.nvidia.com/cuda-toolkit.

[OAB20] Y. Ou, S. Agwa, and C. Batten. Implementing Low-Diameter On-Chip Networks for Manycore Processors Using a Tiled Physical Design Methodology. *Int'l Symp. on Networks-on-Chip (NOCS)*, Sep 2020.

[OHL+06] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. *Int'l Workshop on Lanaguages and Compilers for Parallel Computing (LCPC)*, Nov 2006.

[Oli07] T. E. Oliphant. Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20, 2007.

[Olo16] A. Olofsson. Epiphany-V: A 1024-processor 64-bit RISC System-On-Chip. *Computing Research Repository (CoRR)*, arXiv:abs/1610.01832, Aug 2016.

[ope11] OpenCL Specification, v1.2. Khronos Working Group, 2011. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[ope13] OpenMP Application Program Interface, Version 4.0. OpenMP Architecture Review Board, Jul 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[OUN+17] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. *Conf. on Neural Information Processing Systems (NeurIPS)*, Dec 2017.

[PBMW99] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[PCM+07] G. P. Pezzi, M. C. Cera, E. Mathias, N. Maillard, and P. O. A. Navaux. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2007.

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Conf. on Neural Information Processing Systems (NeurIPS)*, Dec 2019.

[PGW+20] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, et al. Blackparrot: An agile open-source risc-v multicore for accelerator socs. *IEEE Micro*, 40(4):93–102, 2020.

[PLT06] P. Palatin, Y. Lhuillier, and O. Temam. CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs. *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2006.

[PMVdG+13] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):1–24, 2013.

[PZK+17] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A Reconfigurable Architecture For Parallel Paterns. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[Ram11] C. Ramey. TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture. *Symp. on High Performance Chips (Hot Chips)*, Aug 2011.

[rap20] cuGraph - GPU Graph Analytics. Online Webpage, 2020 (accessed Nov 22, 2020). `https://github.com/rapidsai/cugraph`.

[RCBJ11] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters (CAL)*, 10(1):16–19, 2011.

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

[RZAH+19a] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. 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 VLSI Technology and Circuits (VLSI)*, Jun 2019.

[RZAH+19b] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. Evaluating Celerity: A 16nm 695 Giga-RISC-V Instructions/s Manycore Processor with Synthesizable PLL. *IEEE Solid-State Circuits Letters (SSCL)*, 2(12):289–292, Dec 2019.

[SB13] J. Shun and G. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2013.

[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*, 36:34–46, Mar/Apr 2016.

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

[SJS+20] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2020.

[SKK+11] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-Based Global Load Balancing. *SIGPLAN Not.*, page 201–212, feb 2011.

[Smi84] J. Smith. Decoupled Access/Execute Computer Architectures. *ACM Trans. on Computer Systems (TOCS)*, 2(4):289–308, Nov 1984.

[SML17] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Interemdiate Representation. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2017.

[SSF+13] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[Sun22] The Triton Ocean SDK. Online Webpage, 2022 (accessed Sept 27, 2022). `https://sundog-soft.com/portfolio-items/the-triton-ocean-sdk/`.

[TCM18] G. Tagliavini, D. Cesarini, and A. Marongiu. Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2150–2163, 2018.

[TFZ+08] G. Tan, D. Fan, J. Zhang, A. Russo, and G. R. Gao. Experience on Optimizing Irregular Computation for Memory Hierarchy in Manycore Architecture. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Feb 2008.

[TJC+18] K.-A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras. SWOOP: Software-Hardware Co-Design for Non-Speculative, Execute-Ahead, in-Order Cores. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2018.

[TKM+03] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal. A 16-Issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2003.

[TLM+04] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank,

S. Amarasinghe, and A. Agarwal. Evaluation of the RAW Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[VGT+20] P. Vivet, E. Guthmuller, Y. Thonnart, G. Pillonnet, G. Moritz, I. Miro-Panadès, C. Fuguet, J. Durupt, C. Bernard, D. Varreau, J. Pontes, S. Thuries, D. Coriat, M. Harrand, D. Dutoit, D. Lattard, L. Arnaud, J. Charbonnier, P. Coudrain, A. Garnier, F. Berger, A. Gueugnot, A. Greiner, Q. Meunier, A. Farcy, A. Arriordaz, S. Cheramy, and F. Clermidy. A 220GOPS 96-Core Processor with 6 Chiplets 3D-Stacked on an Active Interposer Offering 0.6ns/mm Latency, 3Tb/s/mm2 Inter-Chiplet Interconnects and 156mW/mm2@ 82%-Peak-Efficiency DC-DC Converters. *ISSCC*, Feb 2020.

[WDP+16] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Mar 2016.

[WGH+07] D. Wentzlaff, P. Griffin, H. Hoffman, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27:15–31, Sep/Oct 2007.

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

[WLZ+13] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory Partitioning for Multidimensional Arrays in High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2013.

[WTCB20] M. Wang, T. Ta, L. Cheng, and C. Batten. Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2020.

[WWB19] Y. E. Wang, G.-Y. Wei, and D. Brooks. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. *Computing Research Repository (CoRR)*, arXiv:abs/1907.10701, Jul 2019.

[YGL+20] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2020.

[zde20] ATen: A TENsor library for C++11. Online Webpage, 2020 (accessed Nov 22, 2020). `https://github.com/zdevito/ATen`.

[ZP16] F. S. Zakkak and P. Pratikakis. Building a Java™ Virtual Machine for Non-Cache-Coherent Many-Core Architectures. *Int'l Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, Aug 2016.

[ZSB21] F. Zaruba, F. Schuiki, and L. Benini. Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing. *IEEE Micro*, Mar/Apr 2021.

[ZSD10] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-Based Directory Coherence for Multicore Scalability. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.

[ZSM07] J. Zebchuk, E. Safi, and A. Moshovos. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2007.