Task Parallel Programming on the HammerBlade Manycore

Max Ruttenberg

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington
2025

Reading Committee:
Mark Oskin, Chair
Michael Taylor
Zachary Tatlock

Program Authorized to Offer Degree:
Computer Science and Engineering

© Copyright 2025

Max Ruttenberg

University of Washington

Abstract

Task Parallel Programming on the HammerBlade Manycore

Max Ruttenberg

Chair of the Supervisory Committee:

Mark Oskin
Paul G. Allen School of Computer Science and Engineering

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). However, 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.

Single program multiple data the de-facto standard parallel programming paradigm for manycore processors, not because the programming model is simple, but because its overheads are low. By contrast, the dynamic task parallel programming model has enjoyed considerable success in addressing the programmability challenge of multi-core processors with tens of complex cores and robust and coherent cache memory hierarchy.

In this thesis, I focus on the HammerBlade manycore, and demonstrate that a work-stealing runtime is not just feasible on manycore architectures with SPMs, but such a runtime can also significantly improve the performance of irregular workloads when executing on these architectures. I also explore optimizations to leverage unused SPM space. This runtime framework achieves as much as 1.2–$28.5\times$ speedup on select workloads, and only induces minimal overheads. I show this runtime remains scalable up to a thousand-core system. Loss of locality can be mitigated by embedding locality-aware semantics to the scheduler scheduling while adding a minimum burden on the programmer.

Acknowledgements

Pursuing a PhD can be a long, lonely process. The pandemic started in my second year of seven in this program, making it even lonelier, if not longer. I probably would not have started this degree without the mentorship and inspiration from a few special people. I am certain that I would have failed to complete it without the support of my family, friends, and advisors.

First, and most importantly, I must thank my wife, Katie. You moved across the country with me so that I could attend UW before we were even married. You gave me bottomless love and encouragement for all seven years. You did all of this while starting a family with me and raising our two girls, Kelly and Sage.

I must also thank my daughters. I may have taught you both to walk and talk, but you taught me to be a father. You inspired me to be a better man. Finally, you taught me that, while research and school are important, time spent with family is as rewarding as life can be.

Thank you to my father and mother for pushing me to finish. It was tough love, but I needed to hear it.

Thank you to my advisors and mentors, Mark Oskin and Michael Taylor. I learned so much from both of you. You both worked so hard to provide financial support to me and my labmates and ensured that we could do the most impactful research that we could. You did this all during a period of unprecedented challenges brought upon by the pandemic and societal unrest. Thank you both.

I must also thank Richard Lethin, Marty Deneroff, and Preston Briggs. You three were mentors to me before I started my PhD in the first place. You all took a chance on me when I was young and did not even know how little I knew. Then, you encouraged me to take a chance on myself and pursue this degree. I am not sure what I would be doing now without you, but it certainly would not be as interesting.

Thank you to the folks at AMD Research, including my mentors Srikant Bharadwaj and Yasuko Eckert, who guided me through the second chapter of this thesis, and collaborators Valerie Chen, Robin Knauerhase,

and Ganesh Dasika.

Thank you to all my BSG labmates. Special thanks to Paul Gao, without whom the final chapter of this thesis would not have been possible, and Tommy Jung, the chief architect of HammerBlade. Many thanks to Dustin Richmond, who practically served as my third advisor.

I would never have finished without the support of my friends. Thank you to Dan Petrisko, Jacob Van Geffen, Pratyush Patel, Ellis Michael, and Gus Smith for all the days spent playing board games, playing pickle ball, enjoying good beer, digging for shellfish, going on long runs, and skiing down the streets of Capitol Hill. Thank you to Tal August and Kim Dacarogna, who Katie and I befriended our first week in Seattle and who became parents just a month after us. Thank you to the folks in RCR who helped me rediscover the joy of running with friends in the first two years of my PhD. Thank you to everyone who played department Mafia, in particular Erin Wilson and Matt Johnson, who introduced me that scene.

Lastly, I would like to thank an absent friend, Connor Tait. You passed away while I was finishing this dissertation. You became my friend when I needed one more than ever. I will miss hearing you perform your music, talking to you about movies only you and I have ever watched, and making edgy jokes we would never make in front of anyone else. Wherever you are, I hope there is a room with an open mic and a crowd with a sense of humor.

DEDICATION

To Katie.

For the nights you did not see me, because I was under a deadline.

For the times you pushed me to seek victory, when I would have declared defeat.

This dissertation is written with your endless love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Writing efficient software for manycores is difficult. One major reason why is fundamental. The human brain comes naturally to doing tasks one or, perhaps, two at a time, but not hundreds at once. Doing so is the essence of writing parallel software for manycores. As if this hurdle was not big enough, the challenges of building such hardware in a scalable manner compel architects to forgo optimizations that have made single-core and multi-core systems highly performant, even with relatively little effort from software engineers.

The most impactful difference is in the memory system. Cache coherence protocols are notoriously challenging to scale. But without these coherence protocals, any distribute cache system renders the memory model almost impossible to reason about. By contrast, Scratchpad Memories (SPMs) provide key advantages in parallel systems. Manycore architectures that adopt software-manage SPMs over a traditional cache hierarchy have been proposed and fabricated by both academia and industry Davidson et al. [2018]; Ajayi et al. [2017b]; Bohnenstiehl et al. [2017]; Olofsson [2016]; Brahmakshatriya et al. [2021]. They improve the efficiency and scaling of the memory system by removing the need for a coherence protocol and associated network traffic. When used effectively, SPMs can yield critical performance and energy savings by reducing data movement, improving synchronization times, and eliminating overheads that can arise from false sharing. However, SPMs are often hard to use effectively.

Replacing the traditional L1 caches in favor of SPMs comes at a cost to software productivity. Manycore architectures that rely on SPMs are notoriously challenging to program. Such systems usually require programmers to write applications in low-level C environments and/or directly in assembly. This places the

burden on the programmer to explicitly manage data coherence among private memories and adopt a more restricted programming model (e.g., explicit task partitioning Kelm et al. [2009], message passing Olofsson [2016], and remote store programming Davidson et al. [2018]). 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 with software-managed SPMs.

In my thesis work, I focus on HammerBlade, a manycore architecture developed at the University of Washington. As is the case with similar architectures, programming HammerBlade without loss of domain generality requires using a low-level C runtime environment. Doing so demands that the programmer have both an extensive domain knowledge for their application and for the underlying hardware. Concerns such as data placement, synchronization, and load-balancing are left entirely to the programmer. Having to use a low-level C runtime environment prevents easily reusing existing code written for multi-cores and requires most applications to be completely rewritten.

A common approach to facilitate programming on manycores is by utilizing domain-specific frameworks. This approach has had success in application spaces such as graph processing Brahmakshatriya et al. [2021] and machine learning Cheng et al. [2022]. These frameworks express domain-specific workloads effectively and achieve high performance. However, not every domain is covered. Extending and re-purposing these frameworks for other domains requires non-trivial effort by programmers. General-purpose parallel programming frameworks provide more flexibility than domain-specific ones. However, most such frameworks (e.g., OpenCL ope [2011]) usually adopt a single program multiple data (SPMD) programming model, in which native support for dynamic work scheduling and load balancing is highly limited, if provided at all.

My thesis work approaches this problem by taking inspiration from the success of the dynamic task parallel programming model in the multi-core era, and attempts to address the programmability challenge of manycores with software-managed SPMs by offering a dynamic task parallel programming library that is similar to those that are common on multi-core systems (e.g., Intel Cilk Plus int [2012], Intel Threading Building Blocks (TBB) int [2019], and OpenMP Ayguadé et al. [2009]; ope [2013]). 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 and provide automatic load balancing.

I begin in Chapter 2 with a detailed study into High Bandwidth Memory in its second generation. This study was conducted in the summer of 2020, but the HammerBlade team planned to use HBM2 as its main-memory technology. HammerBlade would not exist as a realized system for a few years more years. As such, this study focused on the performance and power consumption of HBM2 for an AMD Radeon VII graphics processor. I investigate workload sensitivity to HBM's clock and its standard DRAM timing parameters RCD and RP. Using real-system data I proposed a memory clocking and power policy based up on application specific characteristics such as bandwidth consumption and row buffer hit rate.

Chapter 3 presents an overview of the HammerBlade architecture. I desribe HammerBlade's Vanilla Core, the basic compute fabric of HammerBlade. I also describe HammerBlade's multi-tier memory system including its scratchpad memories, its shared tile-memory, and its cached main-memory. The on-chip network is also covered in brief. I explain the SPMD programming model that HammerBlade adopts as its default programming environment. Lastly, I describe how HammerBlade is scaled up using its replicated pod architecture.

Chapter 4 is drawn from work published in ASPLOS 2023. This work explored implementing a dynamic task-parallel runtime system that used a work-stealing scheduler for load-balancing. It is the first work, to my knowledge, to explore this on a system with no coherent L1 or L2 caches, only scratchpad memories. It also proposes optimizations to best leverage the scratchpads to reduce runtime overhead. We evaluated this runtime library on a 128-core HammerBlade system and compared its performance to one that uses a static scheduler.

Chapter 5 extends this published work to run on a multi-thousand-core HammerBlade chip that was taped-out in the Spring of 2021. This chip, christened as BigBlade, was made available in our lab after many years of hard work from me and my fellow labmates who worked on HammerBlade. I detail the software extensions that were required to adapt the runtime library to BigBlade and I propose more optimizations to improve performance, preserve locality, and reduce overhead. I compare the performance of the parallel runtime library to the SPMD model using benchmarks that were also used to evaluate HammerBlade holistically in an ISCA 2024 publication Jung et al. [2024].

I conclude in chapter 6 with brief thoughts on HammerBlade and specifically the parallel programming models explored in this thesis. I propose some ideas for future research directions for HammerBlade software.

Chapter 2

An Experimental Study of HBM2

I begin with a study of an HBM2 memory system on a Radeon VII GPU. This study was conducted early in my PhD when this type of memory system was state-of-the-art. HammerBlade was still a young project at that time, but we planned to adopt HBM2 as its memory system once we did have it in silicon. We would model HammerBlade's performance and energy efficiency using HBM2 for all publications relating to that project during my PhD. At the time of this study, a GPU was the most practical platform from which I could study HBM2 on real hardware.

I learned a lot from this study. First, I learned the impacts of memory frequency and timing parameters in practice. Second, I learned a good deal about where energy is spent in the memory system. These lessons would serve us well in designing HammerBlade's on-chip memory hierarchy.

2.1 Introduction

Memory performance is critical to the overall performance of parallel computation Wulf and McKee [1995]. A key technology that has emerged to meet the memory demands of parallel compute fabrics is die-stacked memory Li et al. [2018b]; O'Connor et al. [2017]. The most commercially successful of these technologies is High Bandwidth Memory (HBM) which is currently in its second generation (HBM2) jed [2015, 2018, 2020]. HBM2 is commercially available on several parallel compute fabrics. Its primary advantage over other DRAM substrates is its ability to deliver cutting edge memory bandwidth at significantly lower power than other high bandwidth memories Villa et al. [2014]; Li et al. [2018b]; Balasubramonian [2019].

Nevertheless, energy consumed by such memory structures remains a significant fraction of the total energy consumption of systems Kanev et al. [2015]; Chatterjee et al. [2017]. Consequently, the energy efficiency of such systems, or performance-per-Watt, has become an important metric for measuring the efficiency of commercial products AMD. Dynamic voltage and frequency scaling (DVFS) is one of the major techniques used in devices to improve the energy efficiency of systems at a system level, and has been studied in detail over the several decades Keramidas et al. [2010]; Miftakhutdinov et al. [2012]; Eyerman and Eeckhout [2010]; Rountree et al. [2011]; Curtis-Maury et al. [2006]; Su et al. [2014]; Kaxiras and Martonosi [2008]; Själander et al. [2014]. DVFS has also been applied to memory devices David et al. [2011]; Deng et al. [2011, 2012]. In addition to frequency and voltage, recent work showed that increasing the latency of certain DRAM operations can help in reducing the required operating voltage, and thus power, of memory systems Chang et al. [2017].

With the increasing usage of HBM in commercial devices such as GPUs AMD [2019]; NVIDIA [2017, 2016], it has become important to study the power characteristics of this type of memory device for improving the energy efficiency Villa et al. [2014]; Diniz et al. [2007]; Kanev et al. [2015]; Li et al. [2018b]; O'Connor et al. [2017]. Consequently, the bandwidth and power characteristics of HBM2 has received a lot of research attention O'Connor et al. [2017]; Li et al. [2020]. However, the performance and power effects of the HBM voltage and frequency and the associated memory characteristics of GPU applications have not been studied in detail. Further, considering that many GPU applications are memory intensive, the latencies of DRAM operations heavily impact the overall performance of the system Chatterjee et al. [2014]. To our knowledge, no previously published work has characterized GPU applications or attempted to modify HBM parameters dynamically to suit the performance and power needs.

In this chapter, our goal is to (a) examine the memory access characteristics of GPU compute applications and their behavior on HBM, (b) develop analytical models that can project the power and performance of applications at different operating states of HBM including frequency, voltage, activation latency, and precharge latency, and then (c) utilize the insights from these models to develop a dynamic HBM tuning policy (RAMP) to improve the energy efficiency and/or performance of the overall system. For this objective, we leverage performance counters and power profiling tools on commercially available hardware to profile GPU compute applications. We explore the performance impact of HBM scaling and characterize applications

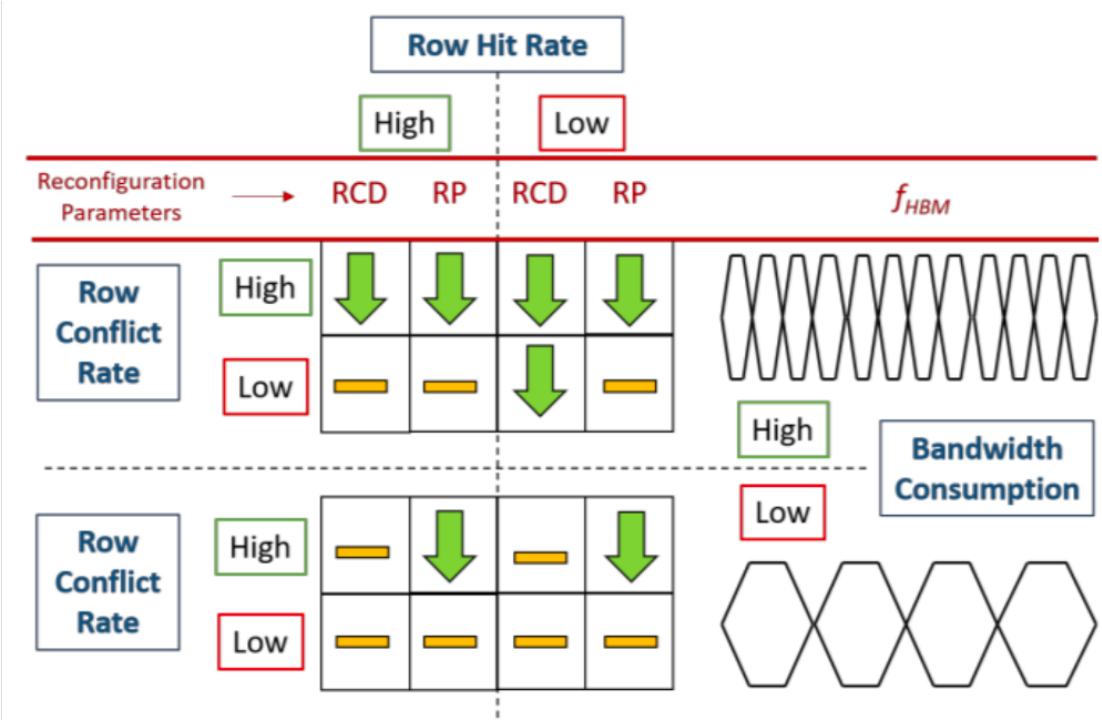

Figure 2.1: RAMP involves reconfiguring HBM to improve energy efficiency. Application characteristics such as bandwidth consumption, row conflict rate, and row hit rate are taken into account to tune HBM parameters such as activation latency (RCD), precharge latency (RP), and operating frequency (f_{HBM}). This reconfiguration is done while changing the supply voltage (V) for improving the overall energy efficiency.

based on memory demands and access behaviors.

We observe that applications show varying sensitivity to memory parameters depending on their access volume and pattern. First, we find that applications with high bandwidth demands benefit most from scaling the operating frequency of HBM. Second, we find that memory access behaviors of applications have a significant effect on the extent to which DRAM timing parameters impact the overall application performance. Applications with high bandwidth consumption typically benefit from reducing the activation latency, or RAS to CAS time (RCD). In addition, among applications that are high bandwidth consumers, we observe that a higher row buffer miss rate increases the sensitivity to RCD. Applications that are low bandwidth consumers show little sensitivity to varying RCD. Finally, we find that performance is sensitive only to row precharge latency (RP) if an application has a high row buffer conflict rate.

Based on our hardware measurements, we develop an analytical model to project the performance and power consumption of applications as we scale HBM's frequency, voltage, activation, and precharge latencies.

We then use the model to project the potential performance improvements from a dynamic overclocking, overvolting, and overtiming technique based on the memory access patterns of the application. We build our performance model by applying linear regressions on frequency and latency sweeps for each kernel in our target workload set. For power, we utilize linear regression to forecast power as a function of bandwidth and instruction throughput as we scale voltage and frequency.

Using these model projections we propose an Application aware dynamic Reconfiguration of Memory for improving Power efficiency (RAMP) of the overall system. RAMP exploits application-specific behavior to adjust HBM parameters dynamically. Using profiling data we assign a set of memory parameters for each GPU kernel. High-bandwidth consuming kernels which have high row conflict rate are allocated higher operating frequency, in addition to lower activation and precharge latencies. Low-bandwidth consuming applications with low row hit rate are allocated lower frequency and low precharge latency only if the row conflict rate is high. The overall policy is shown in Figure 2.1.

Our hardware-based evaluations show that RAMP, when applied at a GPU kernel level, can improve the energy efficiency of the system by 4.3%. For memory-intensive applications, the performance-per-Watt offered by the system improves by as much as 16%. Our per-kernel RAMP policy also improves the performance of applications by up to 27%.

2.2 Background

In this section, we briefly summarize the necessary DRAM, HBM, and DVFS background with respect to how it is used to control memory system power consumption.

2.2.1 DRAM Operation and AC Timings

The most basic building block of any DRAM memory chip is the single transistor-capacity bit cell. Cells are organized into subarrays and addressed a full row at a time. Operations for opening and writing back a row require a delay time to which the on-chip memory controller must adhere to ensure data integrity.

Activation and Precharge. Figure 2.2 (a) shows an example of this timing. An ACTIVATE (ACT) command is issued with the bank address (BA) and row address (RA) over the command interface to open a specific row and bank. The memory controller must then wait a set amount of time before sending a CAS

request to read or write a set of columns (CO) from that row. The wait time required after row activation reflects the delay of the local wordlines and the wire capacitance of the bitlines Keeth et al. [2007]; Lee et al. [2013]. This wait time is commonly denoted as the row to column (RCD) delay. Additionally, before activating a new row in a bank the bitline must be precharged as shown in Figure 2.2 (b). Precharge can be initiated by the memory controller sending a PRECHARGE (PRE) command with the bank address (BA) and row address (RA). Although for certain high-speed DRAMs, including HBM, precharge can be implied with activation by setting a control bit in a CAS request. The associated delay of this process is denoted as row precharge (RP).

Row Hits, Conflicts, and Misses. Critically, for a given bank only one row can be opened at a time. The best-case scenario, with respect to latency, is when a memory request can be serviced by accessing an already opened row. This event is referred to as a row hit. By contrast, a row miss occurs when a request maps to an unopened row thereby requiring an activation before reading or writing. Row misses incur the latency penalty associated with RCD. The worst case scenario, from a latency perspective, is a row conflict. Row conflicts occur when a memory request maps to a row in a bank with another row already opened. Completing this requests requires the bank to be precharged first, then activated to open the necessary row. This causes the request to incur both latencies associated with RCD and RP.

2.2.2 High Bandwidth Memory

High Bandwidth Memory is a die-stacked multi-channel DRAM technology for systems with high parallelism and memory bandwidth demands. Figure 2.3 shows a high-level diagram of an HBM stack. Each die is divided into two channels. The through-silicon vias (TSV) are centered around the middle of each die and route the interface for each channel to its target die. Banks are groups of subarrays each with its own local wordlines and sense amplifiers. Multiple banks share a channel which is used to transfer data and issue commands such as ACT and PRE.

In GPU system we study, HBM is integrated into the same package and connects to the compute fabric by use of a silicon interposer. The interface is multi-channel and each channel operates independently. Each channel's interface is a wide 128-bit data bus. Each HBM stack has eight channels which brings the total width of HBM's interface to 1024-bits per stack jed [2020, 2015]. HBM2 introduced pseudo-channel

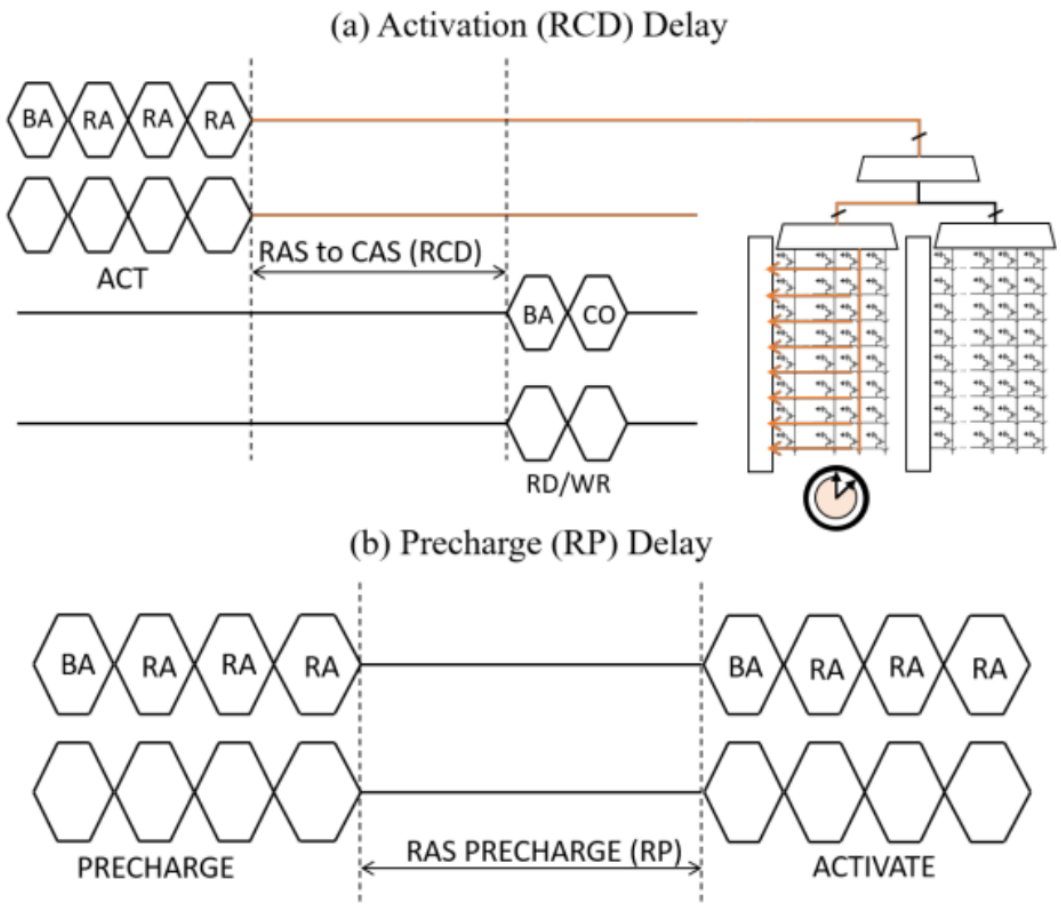

Figure 2.2: (a) Bank activation(ACT) and RCD time (b) Bank precharge and subsequent activation: The memory control issues a PRE with the bank address (BA) and row address (RA). Before the same bank can be targeted with an ACT, time RP must elapse while the bitlines charge to half the supply voltage.

mode jed [2015, 2018, 2020]. Pseudo-channels split the physical channels into two semi-independent virtual ones. The data-lines of each channel are bifurcated making each pseudo-channel's data interface 64-bits wide. The command lines are shared, hence "pseudo"-channel. The burst length in pseudo-channel mode is doubled which means the number of bits per memory request remains the same as in legacy mode. HBM2E increases the peak interface frequency of HBM from 1GHz to 1.2GHz. As with DDR, all generations of HBM to date run data-lines at double-data rate so that effective bandwidth per data-line is 2x the frequency.

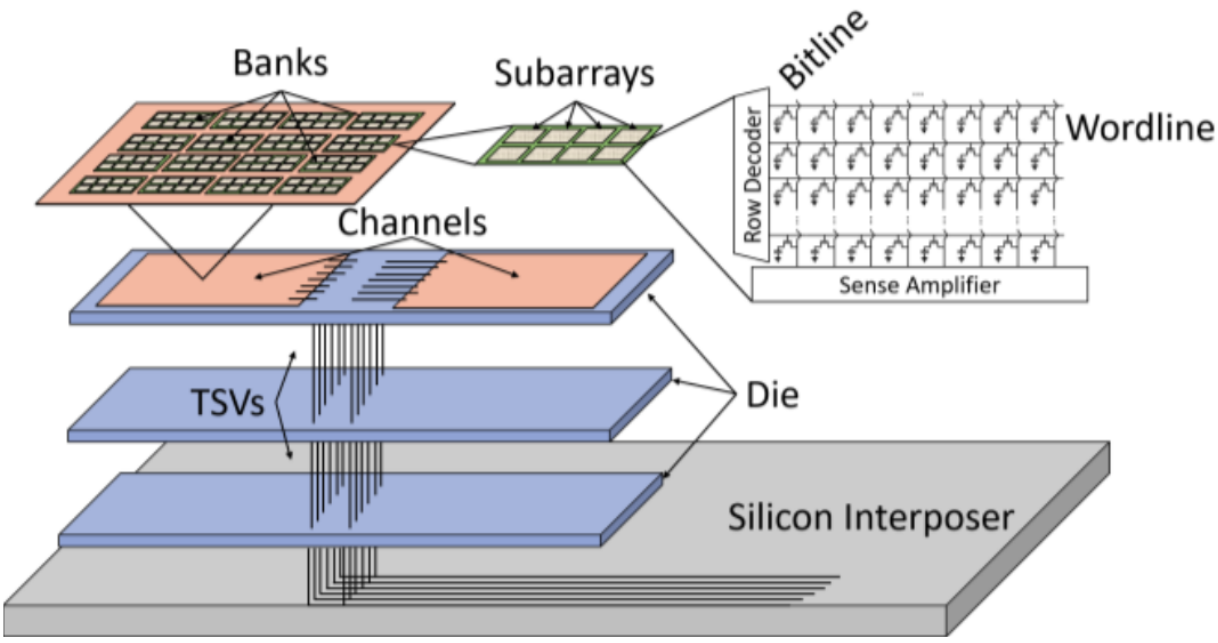

Figure 2.3: High level block diagram of HBM. Each die contains two legacy mode channels. TSVs route per-channel interfaces to their respective die. Each channel is composed of banks which are built from subarrays.

2.2.3 Dynamic Voltage and Frequency Scaling in Memory

Prior work has found significant energy savings from applying DVFS in the memory system David et al. [2011]; Deng et al. [2011]. When applied to DRAM there is an added impact of increased voltage/frequency scaling, whereby refresh rates must be increased due to increased bit-cell capacitance leakage. Few works have studied the impact of DRAM operational latencies to the overall performance of applications.

Voltron. A recent work introduced Voltron Chang et al. [2017, 2018], a policy for a CPU memory system in which the supply voltage for the internal DRAM array voltage is decoupled from the supply voltage of the DRAM chip as a whole. They leverage this decoupling to increase the latencies of RCD and RP, without reducing the memory frequency, in order to reduce the subarray voltage thereby conserving memory system power. Voltron estimates expected performance loss by decreasing voltage, thereby increasing DRAM operation latencies. As we will see later in this work, CPU traffic based policies, such as Voltron, cannot be trivially adopted for GPU based applications accessing HBM.

A key insight in this chapter is that HBM operating frequency has varying performance impact depending on the memory access patterns exhibited by applications. Further, GPU applications show distinct sensitivity

to DRAM operation latencies. In the next section, we show experimental measurements that illustrate this phenomenon.

2.3 Memory Parameter Sensitivity

In this section, we report our observations from a comprehensive study of the impact of memory frequency and timing parameters on GPU applications. We collect this data on an AMD Radeon VII with 16GB of HBM2. To measure performance on real hardware, we utilize work from the overclocking community which has made tools available to adjust component frequencies and memory timings Eliovp. The two memory timing parameters we focus on are RCD and RP as these are the ones we found to have the greatest impact on performance. Overall, our results show that the performance sensitivity of GPU applications towards memory parameters is heavily dependent on their memory access patterns.

2.3.1 Memory Clock Frequency Scaling

Figure 2.4 shows the overall application performance sensitivity as HBM operating frequency is scaled. Stream and Lulesh scale best with increased memory frequency. RNN, CNN, and Stride also see significant, if lesser, performance variation. SGEMM scales to a small degree up until the interface speed reaches 2x its minimum speed but then plateaus. The rest of the applications hardly scale with the interface speed at all.

Performance scaling with HBM frequency closely trends with an application's bandwidth needs. Figure 2.5 shows the relationship between frequency scaling and consumed memory bandwidth at a GPU kernel granularity. Each data point represents a unique GPU kernel from an application. Kernels that consume more than 60% of the peak memory bandwidth see performance scales directly with the interface speed. Kernels that consume 40-60% of the peak bandwidth scale reasonably well although there is a steep drop-off as bandwidth demands lower. Finally, kernels that consume less than 40% of the peak bandwidth see little to no correlation to memory frequency. The RNN and CNN kernels are apparent outliers from the overall trend, as their memory demands occur in bursts. This lowers their average memory bandwidth, yet these kernels are bandwidth sensitive. A notable point on this chart is our pointer-chasing Stride benchmark which scales at 40% with the HBM frequency despite consuming the smallest fraction of the memory bandwidth of any benchmark. This is a purely latency bound benchmark and its speedup relative to memory speed is due to

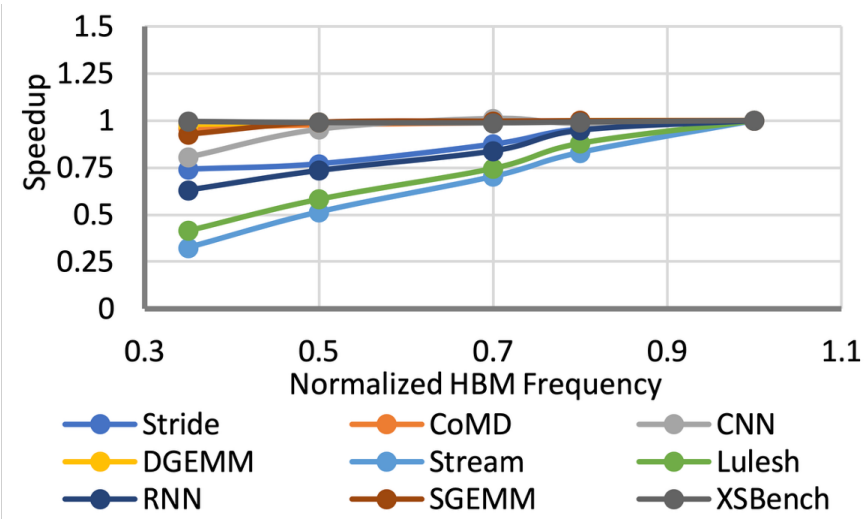

Figure 2.4: GPU Application speedup as HBM operating frequency is scaled. Stream and Lulesh scale at nearly 1-1 ratio with frequency. RNN and Stride scale as well, but to a lesser extent. The other application do not scale much with frequency.

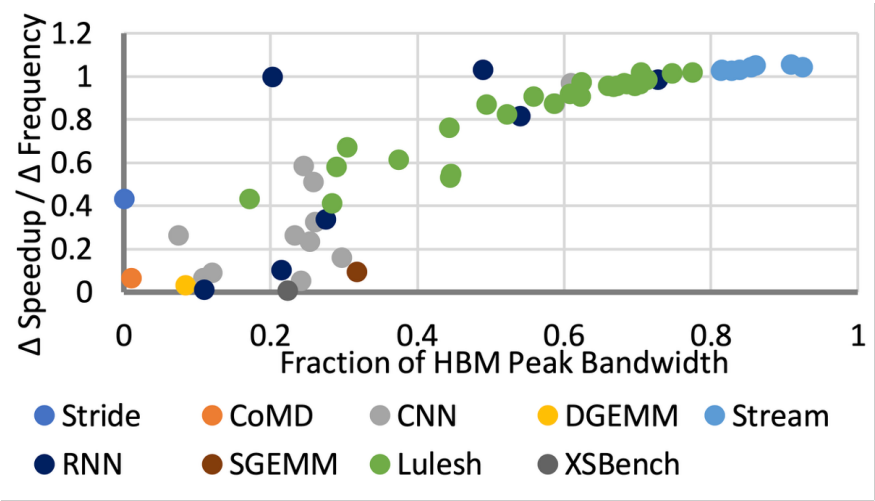

Figure 2.5: Performance sensitivity to HBM speed in relation to the consumed bandwidth of each GPU kernel. There is a strong correlation between consumed kernel bandwidth and its sensitivity to the HBM frequency.

lower access latency and not increased bandwidth.

Overall this shows that high bandwidth-consuming applications deliver improved performance at higher frequencies. This performance sensitivity towards HBM operating frequency can thus be used to reconfigure memory depending on the application traffic.

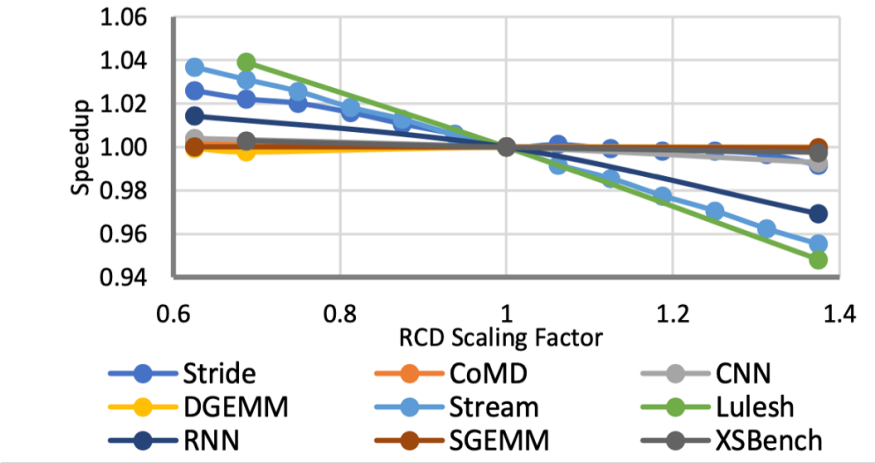

Figure 2.6: Application speedup as RCD is reduced. The x-axis is the scaling factor by which the RCD delay is reduced. Reducing RCD improves performance because it is directly reducing HBM access latency.

2.3.2 Performance Sensitivity to RCD

As described earlier, RCD is a latency parameter so reducing its value should ideally benefit performance. Figure 2.6 shows performance variation as RCD is changed. RCD latency values have been normalized to the default value set by vendors. Lulesh and Stream benefit the most from reducing RCD due to their high memory demands. Stride also benefits from RCD scaling due, but not due to bandwidth demands. Stride's performance is completely bound by the latency of a single memory request at a time and its page hit rate is zero. As discussed in Section II, this means that RCD falls on the critical path of every memory request made in Stride and thus directly impact Stride's performance.

Effect of Bandwidth. Sensitivity is measured as the change in performance divided by the change in RCD normalized to its default value. A higher sensitivity means that reducing RCD will have a higher positive impact on performance. Figure 2.7 shows the relationship between RCD sensitivity and bandwidth. The kernels with higher bandwidth requirements are the most sensitive to RCD changes. For example, Stream benefits significantly as RCD varies for this reason.

Row Buffer Hit Rate. To understand the effect of application characteristics on the impact delivered by memory parameters, we analyze the row buffer hit rate of kernels. Figure 2.8 charts the impact that the row buffer hit rate has on the application's sensitivity to RCD scaling for high-bandwidth kernels. Row buffer hits do not incur the latency penalty of needing to activate a bank. Lulesh consumes less of the peak system

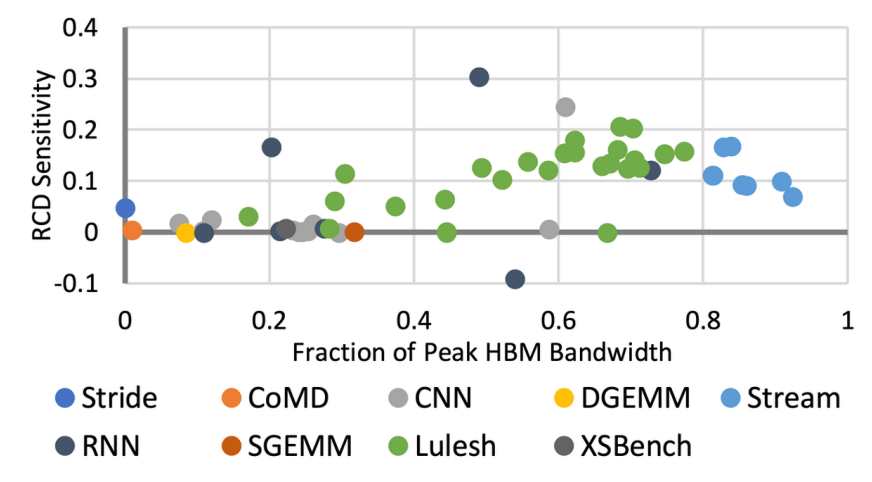

Figure 2.7: Kernel sensitivity to RCD scaling with bandwidth consumption. High bandwidth kernels are most sensitive to changing RCD.

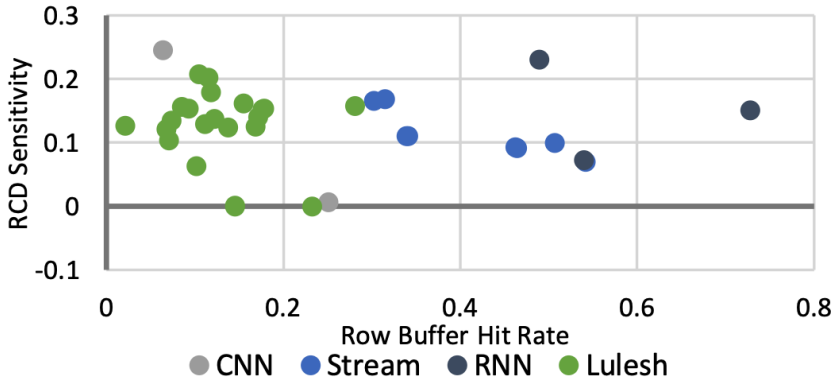

Figure 2.8: High-bandwidth kernel sensitivity to RCD with the row buffer hit rate. Kernels with high bandwidth consumption and low buffer hit rates are most sensitive to RCD.

bandwidth than Stream but it has a significantly lower row buffer hit rate. Therefore, RCD latency is critical to performance for kernels with high memory bandwidth and high row buffer conflict rates.

In summary, the data shows reduced RCD increases performance for applications with high memory bandwidth utilization but poor row buffer locality.

2.3.3 Performance sensitivity to RP

Sensitivity to RP is measured as the performance change divided by the change in RP normalized to its default value. If a workload has a high sensitivity to RP, this means that reducing RP impacts its performance more

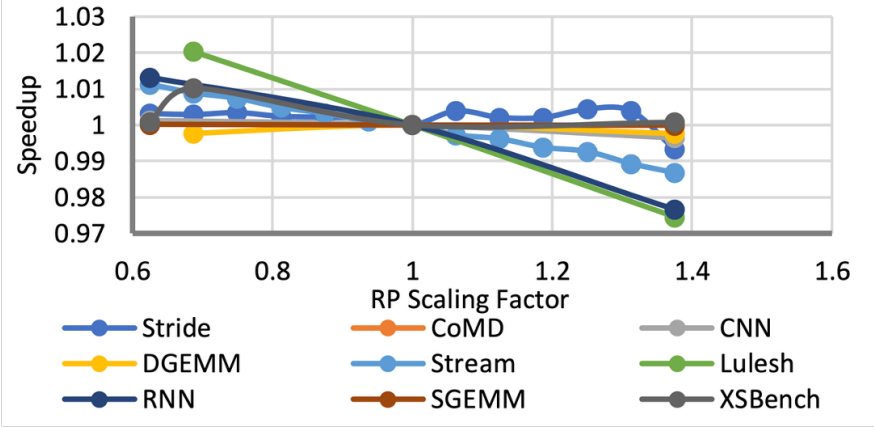

Figure 2.9: Application speedup as RP is reduced. The X-axis is the RP scaling factor. Reducing improves latency in the case of bank conflicts.

positively. Like RCD, RP is a latency parameter; reducing its value should positively impact performance. We plot the performance variation of each application with changing values of RP in Figure 2.9. Just as with RCD, the latencies plotted on the x-axis have been normalized to RP's default value. RP has the greatest impact on Lulesh RNN, and Stream.

Row Buffer Conflict Rate. Row buffer conflicts place row precharge directly on the critical path to fulfilling a memory request. For this reason, we would expect that changing RP would have the greatest performance impact on workloads with more memory requests resulting row buffer conflicts. Figure 2.10 plots the sensitivity to RP of each kernel compared to the row conflict rate of each benchmark. As expected, kernels with higher row buffer conflicts are most sensitive to changing the row precharge delay time. Lulesh benefits the most from scaling RP and it also has the highest rate of row conflicts, followed by RNN. Stream sees more benefit than the rest of the benchmarks due to its high memory demands in general.

2.3.4 Power Trade-offs in Reconfiguring Memory Parameters

HBM frequency is the most significant factor driving application performance of the three parameters explored, but raising the frequency comes at a significant power cost. Raising the frequency requires raising the supply voltage as otherwise critical circuits would fail because of inability of meeting timing. There is a large body of prior work that explores the balance of scaling frequency and voltage Deng et al. [2012]; Keramidas et al. [2010]; Miftakhutdinov et al. [2012], including in the memory system David et al. [2011];

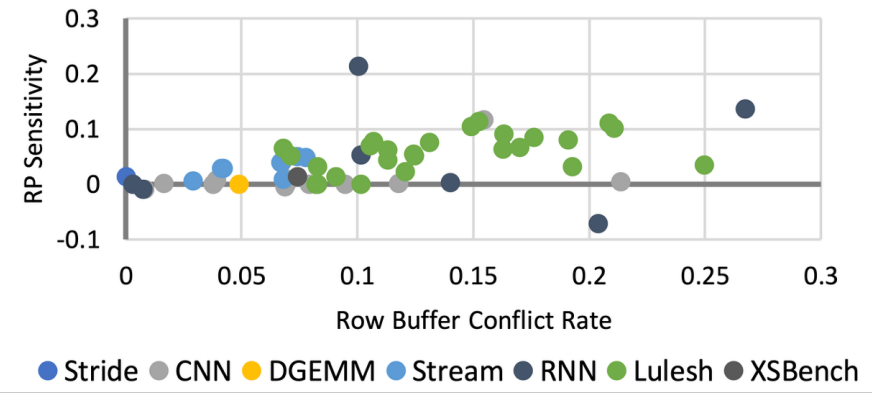

Figure 2.10: Sensitivity to RP scaling with the HBM row buffer conflict rate. Kernels with higher rates of row buffer conflicts are most sensitive to reducing RP.

Deng et al. [2011].

Voltage, and consequently power, is also a key consideration when scaling HBM latency. Voltage controls the floor latency for RCD and RP; reducing the timing parameters below this floor will cause data errors Chang et al. [2017]; Lee et al. [2015]. There can be an incentive to increase latency, at the cost of performance, for the purpose of lowering voltage and reducing power Chang et al. [2018]. Conversely, for workloads that are more sensitive to these timing parameters, raising voltage for the purpose of reducing latency can significantly boost performance. Configuring memory for an ideal voltage and latency setting on a per-application basis improves the performance and power-efficiency of the system.

In summary, the key insights we learned from this experimental study are: (1) interface frequency is by far the most critical parameter impacting HBM performance, (2) RCD and RP for HBM can be tuned depending on the workload memory access pattern to trade-off performance for power consumption, and (3) applications with high bandwidth needs and low row buffer hit rates are most sensitive to RCD while those with high rate of row buffer conflicts are most sensitive to RP.

2.4 Analytical Model for Performance and Power

We develop an analytical model to project the performance and power of workloads using a comprehensive set of results obtained from hardware experiments. As described earlier, our objective is to develop a memory system tuning policy that can dynamically balance performance and power by varying HBM frequency,

voltage, RCD, and RP. First, this policy needs to have a method by which it can project performance change for any given workload leveraging available hardware counters. Second, we would like to evaluate this policy on a system capable of frequency and parameter scaling beyond what is supported on currently available GPUs. To do so, we need a performance model to project a workload's sensitivity to these key HBM parameters and we also need a power model to evaluate how a workload's power consumption will change at varying HBM configurations.

2.4.1 Performance Model

Utilizing the exhaustive results of our performance study on hardware, we build the analytical model to project kernel speedup from HBM tuning based upon its bandwidth consumption and row buffer locality. We apply a linear regression to find the marginal performance change when the interface speed is increased. We use the R^2 value, a commonly used metric, to measure how well the linear regression approximates our observed results. We achieve a linear fit with $R^2 = 0.79$. This lets us measure the consumed bandwidth of a kernel, which we do using publicly available GPU profiling tools ROCMm-Developer-Tools, and model a performance curve as the HBM interface is scaled.

2.4.2 Power Model

In addition to the performance model, we model the HBM power consumption of the kernel based on its consumed memory bandwidth and interface frequency. The major components to dynamic HBM power consumption are the clock frequency and data line activity. For each target HBM frequency, we apply a linear regression to model power consumption as a function of clock frequency and kernel consumed bandwidth. We use publicly available commercial GPU profiling tools to measure power consumption and application memory bandwidth. Using the memory bandwidth data as a proxy for data-line switching activity, clock frequency and voltage we derive an average HBM dynamic power on a per-kernel basis. We also model the static power consumption of the system using the voltage data to obtain the total memory power. The R^2 value for this modeling approach was found to be 0.93.

We also wish to model performance gains and power trade-offs for scaling HBM timing parameters. Previous work has modeled the activation and precharge times for contemporary DRAM technologies as a

function of the supply voltage Chang et al. [2017]. We leverage these estimates and apply linear regression models on our performance data from our study to model performance and power changes as these parameters are scaled.

In addition, we model full GPU power by running linear regressions against performance and power data from executing our benchmarks with a sweep of clock frequencies. We use instructions per second as a measurement of GPU activity. Our linear fit for the power of our GPU computational units achieves an R^2 value of 1, implying this very simple technique can estimate the power consumption of the commercial GPU device with high accuracy for our objective.

2.5 RAMP: Application-Aware Dynamic Reconfiguration of Memory

Using the insights from Section III we build RAMP, a policy for reconfiguring GPU HBM memory timing parameters depending on the memory access pattern of the application. Memory parameters such as frequency, voltage, RCD, RP are tuned according to the sensitivity of the workload. RAMP utilizes memory system hardware counters to select a configuration best balanced for performance and power delivering optimal performance-per-Watt.

RAMP uses application memory access characteristics such as bandwidth, row buffer hits, and row buffer conflicts to select from a settings matrix of HBM frequency, RCD, and RP. RAMP's overall tuning strategy is illustrated in Figure 2.1. After selecting a setting, RAMP scales HBM voltage to the minimum level required to support both the target frequency and latency, as both are tied to the same voltage source. This, in turn, determines the memory power consumed by the application.

Frequency Configuration. RAMP uses bandwidth demand to assign the HBM frequency. The higher the bandwidth demand of the application, the higher the HBM frequency RAMP chooses. This is motivated by our insight from Section III that applications which consume a higher fraction of the peak bandwidth have the highest sensitivity to the HBM frequency. Such applications receive the most performance benefit from running HBM at a higher clock rate. RAMP breaks bandwidth demand into tiers based on the fraction of peak bandwidth consumed as shown in Table 2.1. These bandwidth tiers are low, midling, high, and extra-high. RAMP then assigns a frequency based on which tier an application falls into. This policy helps improve the performance of applications that are starved for memory bandwidth. On the other hand, it also conserves

Table 2.1: RAMP's classification of a workload's bandwidth demands. RAMP uses these classes to select and HBM frequency.

Bandwidth	Classification
$< 20\%$	Low
20% - 40%	Mid
40% - 60%	High
$> 60\%$	Extra-High

power while running compute-intensive workloads by running the memory system at a lower frequency and voltage.

Activation Latency (RCD) Configuration. RAMP's configuration for RCD is driven by both an application's bandwidth demand and its row buffer hit rate. Our performance study shows that RCD is most critical in workloads which consume higher fractions of the peak bandwidth. It follows that applications which fall into the high and extra-high tiers of bandwidth consumption benefit most from reducing latency, while applications with lower bandwidth demands do not benefit much at all. Further, among these high-bandwidth workloads, ones with fewer row buffer hits especially benefit from a lower RCD latency. Reducing RCD latency requires a higher voltage thereby causing the memory system to consume more power. As a result of this power cost, RAMP prioritizes reducing RCD only for workloads it identifies as both consuming a high fraction of the peak bandwidth and having a low row buffer hit rate. This avoids overvolting HBM, and thus drawing the additional power, for applications that will not benefit from the lower latency while allowing for applications that incur penalties from the RCD delay most frequently to receive an extra performance boost.

Precharge Latency (RP) Configuration. An application's row buffer conflict rate drives how RAMP prioritizes reducing RP latency. Our analysis found that applications with more row buffer conflicts were the ones that benefited most from scaling RP. Unlike with RCD, for which workload sensitivity also depended on bandwidth consumption, sensitivity to RP had no correlation with bandwidth needs. However, as is the case with RCD, reducing RP requires raising the supply voltage. Hence, RAMP will prioritize reducing RP for applications with high row buffer conflict rates. This allows applications that benefit most from a low RP latency, even those with low bandwidth demands, to tap into this performance gain.

Voltage Configuration. Because HBM voltage determines the minimum time for both latencies, RAMP scales RCD and RP together. As far as we know, there is no performance or power benefit to only scaling one

Table 2.2: Summary of the RAMP policy used for evaluations. Voltage, frequency, and latencies are scaled based upon the memory access patterns of the running application. The scaling factors are applied to the each parameter's default settings.

Bandwidth	**Conflict Rate**	**Hit Rate**	**HBM Clock Scaling**	**RCD Scaling**	**RP Scaling**	**HBM** V_{dd}
$< 20\%$	$< 25\%$	-	0.35	1.00	1.00	0.9
$< 20\%$	$> 25\%$	-	0.35	0.44	0.32	1.5
20% - 40%	$< 25\%$	-	0.50	1.00	1.00	1.1
20% - 40%	$> 25\%$	-	0.50	0.44	0.32	1.5
40% - 60%	$> 25\%$	-	1.00	0.44	0.32	1.5
40% - 60%	$< 25\%$	$< 25\%$	1.00	0.44	0.32	1.5
40% - 60%	$< 25\%$	$> 25\%$	1.00	0.69	0.69	1.2
$> 60\%$	-	-	1.20	0.44	0.32	1.5

but not the other. RAMP considers RCD and RP as a single parameter and scales them both based on the combined criteria of bandwidth, row buffer conflict rate, and row buffer hit rate. Thus, if RAMP decides that an application is will benefit greatly from reducing either RCD or RP, RAMP will elect to raise the HBM voltage and reduce both parameters accordingly.

Finally, once RAMP has determined the required HBM voltage, it will reduce RCD and RP to their minimum values possible given that voltage. An example of this situation can occur when RAMP raises the HBM frequency to improve the performance of a bandwidth-starved workload. In such a situation, RAMP must also increase voltage. As a convenient side effect, both timing parameters can then be reduced. In situations where RAMP must already raise the HBM voltage for the sake of increasing the frequency, we have found no extra power cost to reducing RCD and RP. For this reason, RAMP will reduce both latencies appropriately for workload's it has already decided should run with a higher HBM frequency.

In the rest of this section, we provide details on how RAMP profiles and classifies workloads to inform its parameter configuration.

2.5.1 Application Profiling

As we saw, RAMP involves reconfiguring the memory depending on the profile of the workload. Note that RAMP can be implemented at both software and hardware levels depending on the granularity of the reconfiguration. We evaluate RAMP at a GPU-kernel level granularity where the reconfiguration of memory happens for each kernel launch. For such a granularity, a software defined framework is sufficient to manage the overall technique. RAMP first profiles the application to retrieve bandwidth and row buffer contention data. This data is stored within the framework. When the application is run (e.g., in production), the GPU's HBM memory system is tuned for optimal performance and/or efficient power-performance.

2.5.2 Memory System Reconfiguration

RAMP reconfigures the memory system at the launch of each kernel based upon its profiled memory access behavior. RAMP's goal is to achieve the best performance improvement-per-Watt consumed. Our insights from Section III inform us that performance can be improved by raising the HBM frequency and reducing RCD and RP latency. However, the extent to which doing so improves the workload's performance depends on both its bandwidth demands and the row buffer contention of its memory traffic. This means that the additional power consumption required to apply these high performance settings is not always justified.

To simplify RAMP's policy, memory traffic is mapped to a discrete classification space for bandwidth, row buffer conflict rate, and row buffer hit rate. As discussed previously, bandwidth demands are broken up into tiers the criteria for which are shown in Table 2.1. RAMP uses a similar approach to classify a kernel's row buffer conflict and hit rates. A kernel is considered to have a low row buffer hit rate if less than 25% of its memory requests result in a hit, otherwise it is considered to have a high hit rate. Similarly, a kernel is classified as having a low conflict rate if less than 25% of its memory requests result in a conflict.

Upon selecting a voltage, clock, and parameter setting for the launched kernel, RAMP configures the HBM memory system on the GPU and HBM timings Eliovp. The application then proceeds with kernel invocation.

Table 2.3: GPU workloads used as benchmarks in this paper.

Workload	Description	Area
SGEMM	Single precision Matrix Multiply	ML
DGEMM	Double precision Matrix Multiply	ML
RNN	Recurrent Neural Network	ML
CNN	Convolutional Neural Network	ML
CoMD	Molecular Dynamics Modeling	HPC
XSBench	Monte Carlo Neutron Transport	HPC
Lulesh	Hydrodynamics Modeling	HPC
Stream	Streaming Benchmark	μBMK
Stride	Single-Threaded Pointer Chasing	μBMK

2.6 Methodology

We evaluate our performance and power results using frequency and latency scaling results on a AMD Radeon VII with 16GB of of HBM2. Performance and power results for parameter and frequency scaling are acquired using our data collected from hardware and leveraging our linear regression models described in Section IV. HBM core voltage levels required for different latency settings are modeled using measurement taken in prior work Chang et al. [2017].

2.6.1 Applications

Table 2.3 shows the benchmarks used for this study Richards et al. [2020]; Karlin et al. [2013]; Tramm et al. [2014]; Deakin et al. [2017]; Villa et al. [2014]; baidu-research. We use a selection of HPC proxy applications, machine learning workloads, and microbenchmarks. In all experiments, input data is sized to ensure significant (several seconds to minutes) runtime. For all benchmarks except Stride, which is designed to be single-threaded, all GPU compute units are utilized.

2.6.2 Baseline Policy

As a baseline we use a configuration which does not perform any tuning of the HBM frequency or its timing parameters. The HBM frequency is left at its default value of 1000 MHz jed [2020]. RCD and RP remain at their default settings. Voltage is left at its default level of 1.2V.

2.6.3 Voltron for GPUs

Voltron Chang et al. [2018] attempts to minimize DRAM power subject to a performance constraint. To estimate the expected performance loss by increasing DRAM latency, Voltron applies a linear model based on (1) the LLC misses per kilo-instruction (MPKI) of the application and (2) the fraction of program execution in which the CPU is stalled waiting on memory requests. Voltron then selects the minimum voltage, and hence maximum latency, for which the expected performance does not rise above an input performance threshold. In this way, Voltron seeks to control power consumption and leaves it to the system administrator to dictate the performance demands.

As a comparison to RAMP, we modify Voltron for use in GPU. Voltron in its original implementation cannot be directly applied to a GPU memory system because of several reasons. First, the key metrics that drive Voltron's policy do not translate well to a GPU. For example, MPKI for a GPU may be a significantly inflated number from its CPU counterpart. The number of cache misses generated by a single instruction is heavily dependent on the thread divergence, address coalescing, and locality within the threads of wavefront (or warp). For this reason, the number of concurrently outstanding memory requests a GPU workload can generate without stalling is much higher and is not comparable across workloads. Second, Voltron uses processor time spent stalling on memory requests as a program characteristic. This is a hard metric to define in GPUs given the massive levels of parallelism. Wavefronts within a Compute Unit(CU) execute in a lock-step manner, and defining stall time at a CU-level is not trivial. Nonetheless, we estimate these application characteristics with existing GPU hardware counters such as cache misses, vector instructions, and the utilization of various functional units. We utilize an aggregating method to determine the stall time spent by applications. Third, Voltron assumes that the voltage sourced by internal DRAM cell arrays is decoupled from the IO. This would allow it to vary latency and voltage without changing the frequency of the memory interface. This assumption does not completely hold for HBM in GPU systems due to the tighter integration between the processor and memory. We allow Voltron to reduce the voltage as much as the hardware allows at the specific settings.

Unlike the policy introduced in this chapter, Voltron does not consider memory frequency or bandwidth demands in its cost model. This is a key difference between our policy and Voltron's as we have identified the HBM frequency as a crucial memory system configuration driving the performance of GPU workloads.

We refer to our ported Voltron policy as Voltron-GPU from this point forward. We use a performance loss threshold of 25% for defining this policy letting us obtain the maximum performance-per-Watt.

2.6.4 Oracle Policy

For our evaluation, we define performance as the speedup of the application from its runtime on the baseline system described. Performance-per-Watt is measured as the overall speedup of the GPU kernels over the baseline divided by the total GPU power (Memory power + GPU core power). We compare against an oracle policy that always selects a configuration for HBM frequency, latency, and voltage resulting in the best performance-per-Watt. This policy should be interpreted as a ceiling on the effectiveness of our design space with respect to tuning for performance-per-Watt.

2.7 Evaluation

We compare the overall performance, power, and efficiency of RAMP against the three memory system tuning policies described in Section VI.

2.7.1 Performance-per-Watt

Figure 2.11 shows the improvement in performance-per-Watt for different kernels of the GPU applications. RAMP achieves the best performance-power balance for high-bandwidth kernels. Figure 2.12 shows RAMP's performance-per-Watt improvement across kernels with varying bandwidth consumption. RAMP's performance-per-Watt improvement begins to break the 5% threshold for kernels that consume more than 40%. This is due to RAMP finding more opportunities to boost performance from scaling latency and frequency for the more memory-intensive workloads. Overall, RAMP achieves an impressive 4.3% performance-per-Watt improvement over the baseline which is within 3% of the oracle. Note that the performance-per-Watt calculated here is for the total system, whereas RAMP is only being applied on the HBM. RAMP outperforms Voltron-GPU for high-bandwidth applications. It achieves an average performance-per-Watt improvement of 6.2% for Lulesh, 5.7% improvement for RNN, and 2% for Stream over baseline compared to a (-)1.7% degradation by Voltron-GPU.

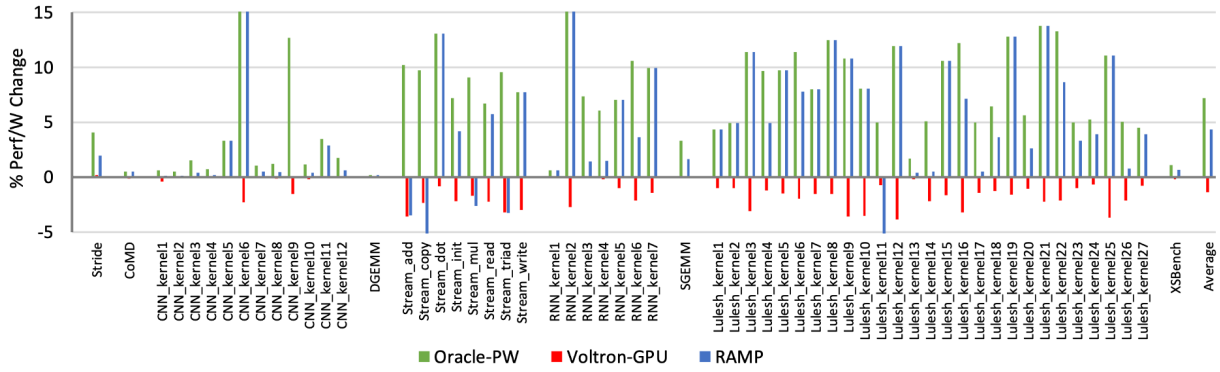

Figure 2.11: Performance per Watt improvement by kernel over a baseline system with no parameter tuning or frequency scaling. High bandwidth kernels can have a performance-per-Watt improvement of over 16% and a mean improvement of 4.3%.

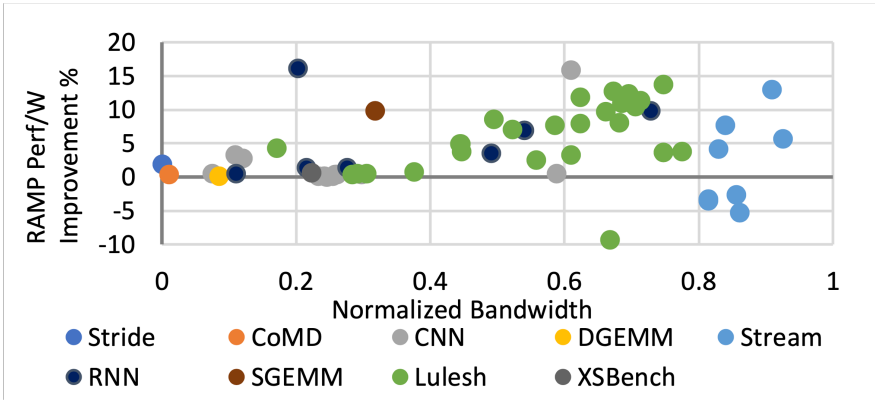

Figure 2.12: Performance-per-Watt improvement across kernels with varying bandwidth demands. RAMP finds the best balance between performance and power for high-bandwidth kernels.

Nonetheless, the oracle achieves a higher performance-per-Watt than RAMP on certain kernels. The most notable application for which RAMP and the oracle deviate is Stream, which has the highest bandwidth consumption of all of our benchmarks. Each Stream kernel consumes more than 80% of the peak HBM bandwidth which meets RAMP's criteria for overclocking. However, when performance and power are of equal concern, the oracle finds that a low-power configuration balances out the performance loss for these kernels. This is in contrast to Lulesh and RNN, the other high-bandwidth applications, for which the oracle and RAMP align more closely.

A critical distinction between these high-bandwidth applications is the extent to which they benefit from reducing RCD and RP. As we show in Figure 2.8, Lulesh and RNN kernels have a sensitivity to RCD as much as twice that of Stream kernels. Additionally, Figure 2.10 shows that RNN and Lulesh kernels can

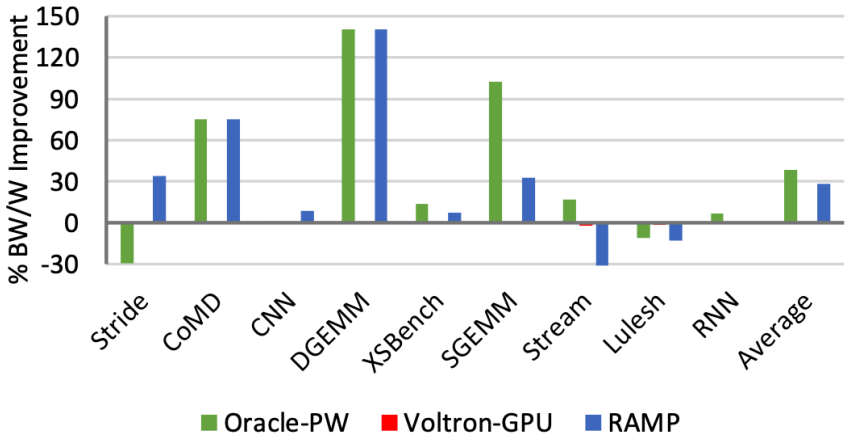

Figure 2.13: Bandwidth-per-Watt improvement over the baseline system.

benefit more from reducing RP due to their high row buffer conflict rates. This additional performance benefit helps balance the increased power costs from the high voltage needed to aggressively clock HBM. However, as is previously noted, Stream does not benefit from reduced latency to the same extent and the performance improvement from raising the frequency alone does not make up for the increased power.

Voltron-GPU does not improve performance-per-Watt for any kernel and results in a mean degradation of 1.3%. Voltron-GPU does not adjust the frequency as part of its policy. Without frequency scaling, the lack of voltage source decoupling makes Voltron-GPU's power reduction approach ineffective. For this reason, Voltron is an impractical solution for performance and power tuning in GPU's with HBM.

2.7.2 Bandwidth-per-Watt

In addition to evaluating the performance-per-Watt improvement, we also examine the improvement in bandwidth-per-Watt which takes into account just the memory system performance and power. These results are shown in Figure 2.13. RAMP improves the bandwidth-per-Watt by an average of 31% across all applications, falling within 7% of the oracle. For the reasons previously described, Voltron has hardly any impact on bandwidth-per-Watt, degrading it by less than 1%.

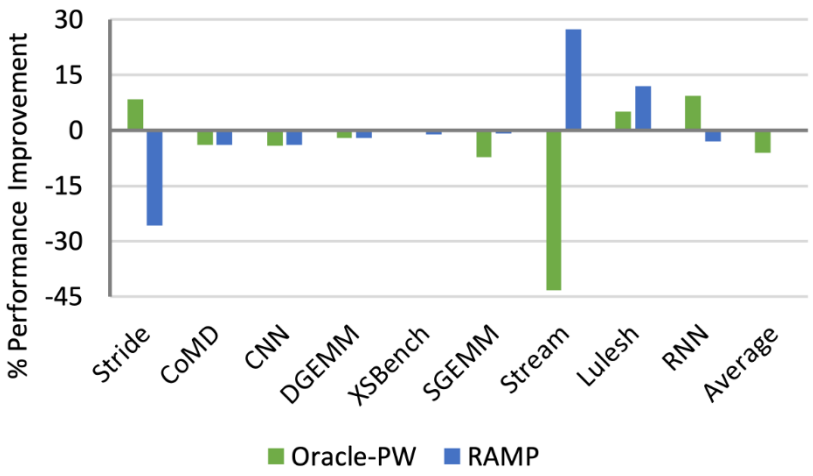

Figure 2.14: Performance improvement by application over a baseline system.

2.7.3 Performance

The performance results for each application are shown in Figure 2.14. On average, RAMP only degrades application performance by 1%. Among the three memory-intensive applications (Stream, Lulesh, and RNN) RAMP improves performance by 12% on average and by as much as 27%. By contrast, the oracle policy degrades performance by 5.9% and among the three memory-intensive workloads it degrades performance by 9.6%. This suggests that RAMP strikes a nice balance between performance and power-efficiency.

RAMP's improvement for Stream and Lulesh is attributable to HBM overclocking and reduced timing latency for the highest bandwidth-consuming applications.

2.7.4 Power

The power results are shown in Figure 2.15. RAMP reduces overall system power by 2.4% and by as high as 27% from underclocking and undervolting low-bandwidth applications. RAMP increases power consumption for the bandwidth-heavy workloads by 7.4% from overclocking HBM to improve performance.

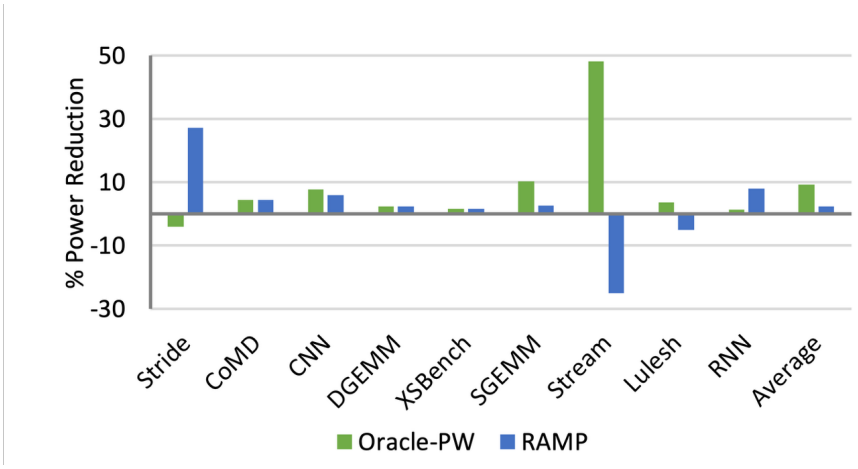

Figure 2.15: Power reduction by kernel over a baseline system. RAMP applies a high power setting for memory-intensive workloads to improve performance.

2.8 Related Work

2.8.1 DVFS for Memory Systems

Several prior works have proposed DVFS for memory system and power management David et al. [2011]; Deng et al. [2011, 2012]. However, they mainly focus on CPU systems with DDR3 memory. Some work also base their policies on the bandwidth demands of the running application David et al. [2011]. Recent studies also proposes DVFS and they use the row buffer hit rate as a key metric to model mean memory latency, which informs their policy Deng et al. [2011, 2012]. This work differs from theirs mainly because we focus on GPU systems with HBM DRAM. However, their approach focuses on scaling frequency and they do not discuss reducing timing parameters based on row buffer contention. In addition to bandwidth-based frequency scaling, we propose scaling voltage and timing parameters based on row buffer hit rate and contention rate.

2.8.2 DRAM Operation Latency Tuning

The overclocking community has examined the performance benefits and power costs of lowering DRAM latencies, including for HBM Burke; Eliovp; Liu. As far as we are aware, this community has primarily focused on statically setting frequency, voltage, and latency. By contrast, we focus on how a workload's dynamic memory access behavior drives the power and performance tradeoffs of reconfiguring HBM.

Previous work shows that the true latency of accessing closed DRAM rows is in fact lower than what

is stated on their data sheets when operating at lower temperatures Lee et al. [2015]. They exploit this by tuning RCD and RP based on ambient temperatures but do not consider tuning them based on memory traffic characterization. Some works observe that the true latency of row activation varies depending on how recently the cells have been refreshed Hassan et al. [2016]; Shin et al. [2014]; Hassan et al. [2019]; Wang et al. [2018]. They propose techniques to reduce DRAM latency including assigning priorities to memory requests based on how recently their target cells have been written back.

There are other bodies of work that propose microarchitectural changes to reduce DRAM latency Keeth et al. [2007]; Seongil et al. [2014]. While these policies might change the tuning parameters, RAMP's core idea of reconfiguration will still be applicable on such architectures.

2.8.3 Memory Systems for GPUs

Another body of work addresses the power management in GPU memory systems by proposing changes to die-stacked DRAM microarchitectures to reduce the energy overhead of data movement within memory chips and to reduce the activation energy O'Connor et al. [2017]; Chatterjee et al. [2017]. Such optimizations are orthogonal to the idea presented by RAMP, where the activation latency is tuned according to the application requirements.

Another previous work observes that memory latency divergence can be a major bottleneck in GPUs Chatterjee et al. [2014]. They explore the benefits of changing the memory scheduling policy from one which optimizes for row-buffer locality to a policy in which requests from the same SIMT group are scheduled together. The policies in this work could synergize with their approach by improving tolerance to request streams with high row-buffer contention.

2.9 Summary

In this chapter, we performed a thorough memory-side performance study with HBM on a set of GPU workloads from the HPC and ML application domains. We found that workloads which consume more than 60% of the peak bandwidth benefit enormously from frequency scaling. Further, we found that key DRAM timing parameters can heavily impact performance for applications with irregular DRAM row buffer access patterns. Our insights into memory behavior and performance led us to develop RAMP, a memory

system reconfiguration policy for HBM power, frequency, and latency. We developed an analytical model for GPU-HBM performance and power based on measurements collected from real hardware to evaluate this policy. Our evaluations show that RAMP can provide a mean performance improvement of 12% for memory-intensive workloads and an overall performance-per-Watt improvement of 4.3% from our policy over a baseline HBM system outperforming state-of-art techniques. These results strongly show that application aware reconfiguration of HBM is a promising technique for building power efficient GPU systems.

Chapter 3

The HammerBlade Manycore

HammerBlade is a tiled compute fabric in the mold of predecessors such as RAW Taylor et al. [2004]; Gordon et al. [2006], Tilera Wentzlaff et al. [2007]; Bell et al. [2008]; Ramey [2011], and Celerity Ajayi et al. [2017b]; Rovinski et al. [2019b,a]; Ajayi et al. [2017a]; Davidson et al. [2018]. It is composed of processing elements, or tiles, that operate in parallel and can communicate over an on-chip network using a shared memory space. Having been taped-out and silicon-verified in a 14 nm process node, HammerBlade surpasses these predecessors in scale reaching a threshold of more than 2000 tiles on a single die.

HammerBlade's mission as a fabric is to be (1) scalable, (2) build-able, and (3) programmable. Scalable in that it can achieve linear performance gains as core counts expand. Build-able as in the design is silicon-verified and the system as a whole could be made into a physical chip on which parallel programs can be run "as is." Finally, programmable as in HammerBlade is designed to accelerate a broad range of application domains using established programming languages and frameworks, making it a highly flexible substrate.

At a high level, the HammerBlade architecture is a configurable-sized array of scalar RISC-V cores supporting the floating-point and AMO extensions. Each core owns a 4 KB region of low-latency scratchpad. Cores communicate with a load/store interface over a 2-D mesh-with-ruching on chip network (OCN) Jung et al. [2020]; Ou et al. [2020]. Figure 3.1 presents an architectural diagram of a small-scale (i.e., 128-core) HammerBlade system. There are three levels of the memory hierarchy: a core-local scratchpad; inter-core scratchpad(s); and DRAM which is backed with a banked last-level cache (LLC). The core-local scratchpad, remote scratchpads, caches, and other network locations are mapped to non-intersecting regions

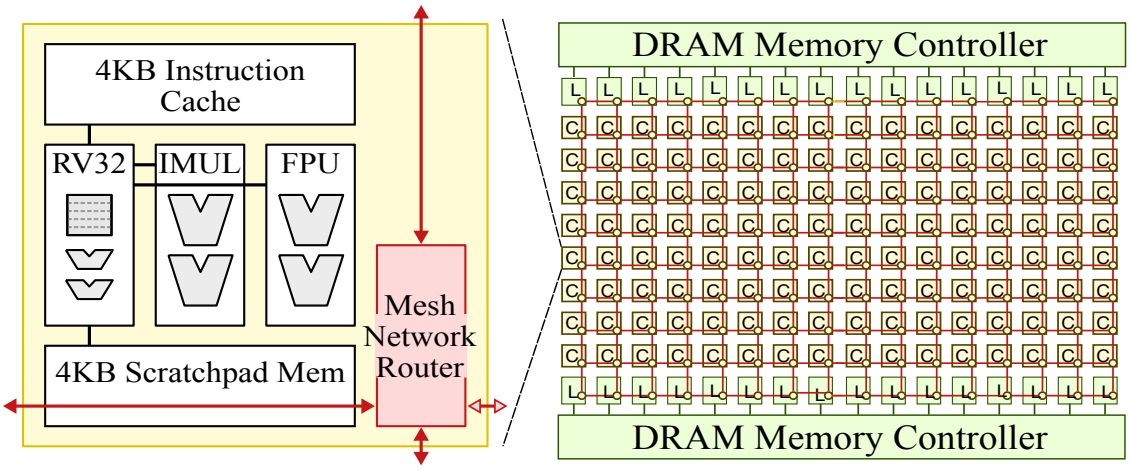

Figure 3.1: A HammerBlade pod with 128 vanilla tiles (C) and 32 last-level cache (L) banks interconnected via mesh-based on-chip network.

of a core's address space. Consequently, the architecture exposes a partitioned global address space (PGAS) programming model.

I will spend the rest of this chapter diving deeper on the specifics of HammerBlade's cores, memory system, and network. I will also describe how the system scales to chip-size using its pod architecture. I will conclude this chapter with description of the CUDA-Lite programming framework as well as the simulation and testing infrastructure we have used to develop HammerBlade over the years.

3.1 Vanilla Tile

At the heart of HammerBlade lies the vanilla tile. A tile is composed of a core, a 4 KB data memory, a 4 KB instruction-cache, and a network router that serves as a gateway between the tile and the rest of the system. The Vanilla tile's core implements the open-source RISC-V Instruction Set Architecture (ISA). It implements the 32-bit instruction set and includes the multiply (M) and floating-point (F) extensions as well as a non-standard subset of the atomic memory operations (A).

The core, and the entire tile, is designed to maximize system-wide thread-level parallelism. The vanilla tile is designed to be area-efficient so that more of them can fit on one chip. Thus the core forgoes many optimizations in its pipeline that would be preferred for single-threaded performance such as multiple-issue, out-of-order execution, or advanced branch prediction. The instruction-cache is compact; It has a capacity for

just one thousand instructions and it is direct mapped. There is no data cache on the tile at all, only the 4 KB data memory.

The vanilla core features select optimizations that improve overall throughput by a margin that justifies the area. It features a 5-stage pipeline that balances performance with area efficiency. It includes an additional 5-stage FPU. It also has a scoreboard to support long-latency instructions such as integer division. Critically, this scoreboard enables non-blocking remote memory operations with which programs can efficiently utilize off-tile memory such as with remote store programming Hoffmann et al. [2010].

3.2 Multi-Tiered Memory System

HammerBlade implements a partitioned global address space (PGAS) to provide a shared memory interface. This address space scheme exposes the multi-tiered memory system to software. The first tier is each tile's local scratchpad memory, the address subspace for which is replicated, meaning that each core can access its 4 KB of fast data storage with a 12-bit offset from zero. The next tier is tile shared-memory, or rather the scratchpad memory of other tiles. Through this tier, tiles can read and write each other's local data storage directly with scalar loads and stores. Addresses in this tier encode the target tile's network location (its X and Y coordinates on the mesh) into the address. The final tier is DRAM memory which uses a shared LLC to reduce latency and exploit locality when fetching. There is also a special tier for IO which is reserved for communication with a host co-processor. HammerBlade uses a 2-bit encoding to distinguish between these tiers.

3.2.1 Scratchpad Memory

The fastest data store for the HammerBlade tile is its 4 KB scratchpad memory. The latency for accessing the scratchpad memory is 2 cycles, which is the minimum memory latency on HammerBlade. This tiny memory is ideal for data that will be accessed often and will be reused frequently. It is also ideal for data that only needs to be accessed by one core. This reduces pollution in the on-chip network and the rest of the shared-memory system. From a scalability perspective, this tile-local scratchpad is the only memory type on HammerBlade whose capacity and bandwidth scales with the number of cores.

The most common use case for the scratchpad is for stack memory. This type of data fits into the second

category mentioned above; Stack memory is generally only accessed by the core who owns the stack. Placing every single core's stack in shared memory places pressure on the shared network and cache. Thus, for performance purposes, the scratchpad is often the best place for it. It must be said that 4 KB of memory is not very much. A stack placed in scratchpad memory is easily overflowed making deep call stacks perilous. This limitation is a topic of discussion in chapters 4 and 5.

A second common use case for scratchpad memory is as a software managed data cache. An application might perform many operations on a small working set before committing it back to main memory. Workloads that can leverage scratchpad for this purpose often enjoy sizable speedups over reading it from the shared memory system over and over. This data-management pattern is used in several applications implemented to evaluate HammerBlade in Jung et al. [2024]. Example applications written for HammerBlade that do this include matrix multiplication, 2D-FFT, Smith-Waterman DNA alignment, and AES encryption.

The third common use is for statically allocated core-local storage. Because HammerBlade's hardware implements a PGAS that replicates the 12-bit address space for scratchpad memory, it is possible to access statically allocated scratchpad data with no overhead. A 12-bit literal offset from zero can be accessed with a single instruction using an offset-load/store and the **zero** register. A consequence of this is that a copy of these static variables exist for every single core. Often this third use is leveraged to implement the second mentioned above. It can be used for other purposes such as replicating frequently accessed read-only data such as small lookup tables.

The adoption of software-managed scratchpad memories is a practical necessity for a scalable shared-memory architecture like HammerBlade. An alternative would be to implement a standard cache hierarchy with a scalable coherence protocol. Constructing such systems scalably is a well-explored subject of previous study Fu and Wentzlaff [2015]; Singh et al. [2013]; Power et al. [2013]. What can be said for certain on this subject is that hardware implementations of coherent cache hierarchies quickly become complex and expensive at scale. At the same time, the common case, or at least the best one, for parallel applications is that threads synchronize rarely. Thus, implementing a cache-coherence system entirely in hardware would amount to a large investment in a feature that, in the ideal case, will not be heavily used. By eschewing a coherent cache system, HammerBlade leaves it to software to manage data placement and memory consistency during synchronization. This buys area at the cost of higher software complexity.

3.2.2 Tile Shared Memory

Tiles can read and write each other's scratchpad memories. The latency associated with doing so depends on the sender and receivers distance relative to the on-chip network. This second tier of the memory system is similar to the first in that it refers to the same physical scratchpad memories. However, the manner in which it is used differs significantly.

This memory tier is most commonly used for scalable synchronization and polling. HammerBlade's shared memory can be a poor choice for a polling location, particularly if a significant number cores are involved. Even if each core polls its own location, as is the case for scalable mutual exclusion locks or barriers, the memory traffic pollutes the network and LLC, and deteriorates performance over all. Using tile shared memory, each core can poll its own scratchpad memory until some other core writes to it. This contains memory traffic caused by polling to within the poller's tile and avoids network and cache pollution.

Another use of this tier is for data sharing at scales larger than what will fit in a single 4 KB bank. The best example of this can be seen in an implementation of Jacobi stencil written for HammerBlade in Jung et al. [2024] Stencils are particularly advantaged from this use case since they have a high degree of spatial locality that HammerBlade's tile-grid structure can exploit. In this case, HammerBlade tiles access data stored in their direct neighbors' scratchpad memories, incurring a minimal network latency.

There are other known uses for tile-shared memory. Chapters 4 and 5 explore using this memory tier to implement shared work queues and a work-stealing scheduler. This memory tier is versatile and can provide a scalable means for inter-core communication without placing pressure on the shared off-chip memory.

3.2.3 Shared DRAM Memory

The final tier of HammerBlade's memory system is shared DRAM memory. It is the highest capacity tier with a 2 GB address range per HammerBlade pod. This makes this tier ideal for large data sets or any other data structure whose spatial complexity is unbounded by the number of cores. It is also the last-level storage for instruction memory.

It is the slowest tier of the memory hierarchy. DRAM fetches are time-intensive and can range from tens to hundreds of cycles. The data path to DRAM is by way of a multi-banked last-level cache. These caches are illustrated in Figure 3.1 with each bank denoted with an **L**. While the cache amortizes latency associated

with DRAM fetches, the individual bank can service just one scalar memory request per cycle. This can drive latency higher since the cache banks are outnumbered by the vanilla tiles. [1]. A final factor that can increase latency is that, as of the time of this writing, the cache banks block on a miss and cannot service hits. This means that a single miss can significantly increase the latency to access even cache-resident data. These caches are used for both data and instructions, meaning that L1 instruction cache misses can put additional pressure on this memory tier.

The work in this Thesis studies HammerBlade instances with an HBM2 memory system. The system is configured to map each HammerBlade pod's DRAM address space to a single pseudo-channel. In order to mitigate the restriction that each HammerBlade cache bank blocks on a miss, banks are assigned to non-overlapping regions of main memory, each of which maps to a distinct DRAM bank. This means that, even though only one outstanding fetch per cache bank is allowed, DRAM bank-level parallelism can still be saturated.

3.2.4 Atomic Memory Operations

HammerBlade implements a subset of the RISC-V extension for atomic memory operations. These operations are implemented in the LLC. As a result, atomic memory operations are only supported on memory addresses mapping to shared DRAM. This makes DRAM the only option for many lock-free data structures.

An important exception is single-bit atomic read-modify-writes. **amoswap** operations targeting tile scratchpad memories are remapped to a single one-bit register, of which each tile has one. This is sufficient to implement correct spin-lock semantics for a lock whose value is either zero or one. It is limiting, however, in that all data structures using scratchpad memories to hold a lock share this register. This operation is best thought of as a lock on the entire tile.

3.3 On-Chip Network

As shown in Figure 3.1, the tiles and caches in a HammerBlade pod are connected in a mesh network topology. The network is single-flit and each packet contains 32-bit scalar payloads. Dimension order routing is used as

[1] Note that the ratio of vanilla tiles to cache-memory banks is not a fixed value in HammerBlade's architecture, but that ratio is 4:1 for all research in this Thesis. All data presented in chapters 4 and chapters 5 are collected from HammerBlade pods configured as shown in Figure 3.1

it is simple, low cost, and it ensures the network is deadlock free. HammerBlade's network is in fact two distinct mesh networks, one for request packets and another for responses. Using separate networks for the two packet types also prevents deadlock. Additional flow control is enforced at the endpoints using credit counters. Endpoint routers restrict its tile's total number of outstanding requests. This functionally caps the total number of packets in the network and mitigates starvation and congestion. This also the means by which software enforces memory ordering semantics. Waiting for all memory requests to complete can be accomplished at the endpoints by waiting until its credit count returns to its full value. For the vanilla tiles, this is accomplished with a **fetch** instruction.

HammerBlade's network on-chip has been a major subject of research in the group. Work has been done to extend the network design to have "rouche links" to increase the bisection bandwidth and reduce hop count at low cost in terms of the extra wiring. I believe that I have given a description of the network here that is necessary and sufficient context for this thesis. If the reader is interested in learning more, I encourage them to read any of Jung et al. [2020, 2024].

3.4 Pod Architecture

The HammerBlade systems is organized into pods. Each pod is a submesh of vanilla tiles and its own cache array at the north and south end. Figure 3.1 illustrates a HammerBlade pod with 128 tiles, denoted with **C**, and 32 LLC banks marked with **L**. A pod's address space is a 32-bit PGAS described above in Section 3.2. Multiple pods connect together over a unified mesh network. This preserves bisection bandwidth across the chip and preserves the simplicity of the mesh design.

HammerBlade's pods can address each other's memory using a 4-bit extension to the address space by indicating a pod coordinate to whom memory operations should be rerouted. This 4-bit extension is implemented with a control and status register (CSR) instantiated on each core. When the core dispatches a memory operation to a remote endpoint, not to its own local memory, the 4-bit register is checked and the pod coordinates that it encodes are used to format the network packet. This enables parallel software targeting one large data set to which all pods can collectively share access.

3.5 Programming Model

HammerBlade adheres to the single program multiple data (SPMD) model for parallel programs. The basic model of SPMD is that control is transferred to the programmer by way of some entry point or kernel, and that entry point is instantiated once per thread. This means that all threads are running the *same program*, but can use identifiers or other control variables to operate on *separate data*. Synchronization between threads in the SPMD model are, in the ideal case, extremely rare. Typically, the key synchronization primitive in a SPMD program are barriers in which a large set of threads participate.

SPMD has advantages as a parallel programming paradigm. It assumes that every processing element, be it a thread, core, or processor, exists as an isolated instance and can potentially run its program from start to finish without any communication with its co-processing elements whatsoever. In other words, the SPMD model assumes that the program is parallel by default, and the programmer must dictate when serialization and synchronization must occur. The result is that a SPMD program that *is* embarrassingly parallel can run with hardly any overhead at all. Thus, a SPMD runtime can be implemented minimally and allow highly parallel programs to scale up to the limits of the hardware without being weighed down by communication bottlenecks. This is a key reason why SPMD has been adopted by commonly used parallel software frameworks such as cuda and mpi.

We call HammerBlade's primary programming framework CUDA-Lite, named as such since it is meant to mirror CUDA's API and programming model. CUDA-Lite's mission is to provide a SPMD programming environment for HammerBlade at minimal overhead. Additionally, on the host side it provides key services that are necessary to run HammerBlade kernels. These services include program loading, address translation, ELF symbol table decoding, shared buffer allocation, and kernel invocation and scheduling. Thus, there are two core parts of a CUDA-Lite program: the host software making API calls and managing kernel invocations, and the SPMD style parallel code that is meant to run on the accelerator itself.

3.5.1 Related Manycore Architectures

Early manycore research prototypes integrated 16–110 cores on a single die Taylor et al. [2003]; McKeown et al. [2017]; Howard et al. [2010]; Hoskote et al. [2007]; Lis et al. [2013]; Vivet et al. [2020]; Tan et al. [2008]. The industry has adopted the manycore approach as well and products available typically include

64–256 cores Bell et al. [2008]; Ramey [2011]; Kanter [2015]; Wheeler [2020]; Halfhill [2020]; Wentzlaff et al. [2007]; Li et al. [2018a]; Kalray. Recent research prototypes have scaled core counts by an order-of-magnitude to over a thousand cores (e.g., 1000-core KiloCore Bohnenstiehl et al. [2017], 1024-core Epiphany-V Olofsson [2016], and 4096-core Manticore Zaruba et al. [2021])

Raw implements a 16-core, general-purpose, 32-bit manycore architecture with a RISC ISA Taylor et al. [2004]. Raw was one of the first manycore architecures. Although Raw supported a global address space, it did not support load and store instructions that could access other core's memory spaces; instead explicit dynamic messages had to be sent in software, and the receiving core either serviced this request by triggering an interrupt or by explicitly receiving the memory request.

Tilera's TILE64 Bell et al. [2008] is a commercial, 64-core Linux-capable manycore that evolved from Taylor et al. [2004]. Several features in HammerBlade are inspired by TILE64, including its PGAS memory space accessible via light-weight remote load and stores, and efficient support for the remote store programming model Hoffmann et al. [2010]. Unlike TILE64, HammerBlade does not run an operating system on each core. TILE64 also lacked a floating point unit and its synchronization primitives relied on static and dynamic networks. By contrast, HammerBlade relies mostly on its PGAS and atomic memory operations for synchronization.

Adapteva Epiphany V Olofsson [2016] is a manycore that supports PGAS and lightweight loads and stores. Epiphany lacked an L2 and external memory system and focused on a generalized systolic-array style communication, where computation is explicitly placed and communication only happens between nodes and I/O. Epiphany's strong support for remote stores made it suitable for streaming or remote store programming Hoffmann et al. [2010].

Celerity is a direct predecessor of HammerBlade , designed to run streaming and remote store programming applications Rovinski et al. [2019a]. Like in HammerBlade , all scratchpads in the Celerity architecture are globally addressable on the 2-D mesh, providing a PGAS for communication; however Celerity only supports remote stores and not remote loads.

Chapter 4

A Dynamic Task Parallel Library

In this chapter, I explore the implementation of a task-parallel runtime for HammerBlade. This runtime supports dynamic fork-join parallelism, allowing for greater flexibility than single program multiple data (SPMD). Additionally, the runtime presented in this chapter implements a work-stealing scheduling policy and discusses optimizations that leverage HammerBlade's scratchpad memories to reduce the overhead. This work was published in ASPLOS and the original paper can be found here: `https://dl.acm.org/doi/10.1145/3582016.3582020`.

4.1 Introduction

Scratchpad memories (SPMs) provide key advantages in single-chip parallel architectures. Most crucially, they improve the efficiency and scaling of the memory system by removing the need for a coherence protocol and associated network traffic. When used effectively, SPMs can yield critical performance and energy savings by reducing data movement, improving synchronization times, and eliminating overheads that can arise from false sharing. As a result, academic and industry chip-makers have increasingly favored these software-managed fast memories over L1 caches as core counts scale from the tens to hundreds and thousands Davidson et al. [2018]; Ajayi et al. [2017b]; Bohnenstiehl et al. [2017]; Olofsson [2016]; Brahmakshatriya et al. [2021], a trend illustrated in Figure 4.1.

Replacing the traditional L1 caches in favor of SPMs comes at a cost to software productivity. Manycore architectures (i.e., those with more than a hundred cores) that rely heavily on SPMs are notoriously challenging

to program. Such systems usually require programmers to write applications in low-level C environments and/or directly in assembly. This places the burden on the programmer to explicitly manage data coherence among private memories and adopt a more restricted programming model (e.g., explicit task partitioning Kelm et al. [2009], message passing Olofsson [2016], and remote store programming Davidson et al. [2018]). 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 with software-managed SPMs.

One common method to facilitate programming on such architectures is by providing domain-specific frameworks. This approach has had success in application spaces such as graph processing Brahmakshatriya et al. [2021] and deep learning Cheng et al. [2022]. These frameworks express domain-specific workloads effectively and achieve high performance. However, not every domain is covered. Extending and re-purposing these frameworks for another domain requires non-trivial effort by programmers. General-purpose parallel programming frameworks provide more flexibility than domain-specific ones. However, most such frameworks (e.g., OpenCL ope [2011]) usually adopt a single-program-multiple-data (SPMD) programming model, in which native support for dynamic work scheduling and load balancing is highly limited, if provided at all.

In this work, 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 manycore architectures with software-managed SPMs by offering a dynamic task parallel programming framework that is similar to those that are common on multi-core systems (e.g., Intel Cilk Plus int [2012], Intel Threading Building Blocks (TBB) int [2019], and OpenMP Ayguadé et al. [2009]; ope [2013]). 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 McCool et al. [2012].

We demonstrate our ideas by implementing the proposed dynamic task parallel programming framework on an open source manycore. Our approach allows dynamic task parallel applications written for traditional hardware-based cache coherence multi-cores to work on manycore architectures with only minimal changes to the software. In Section 4.2, we provide a general background on our target open-source manycore archi-

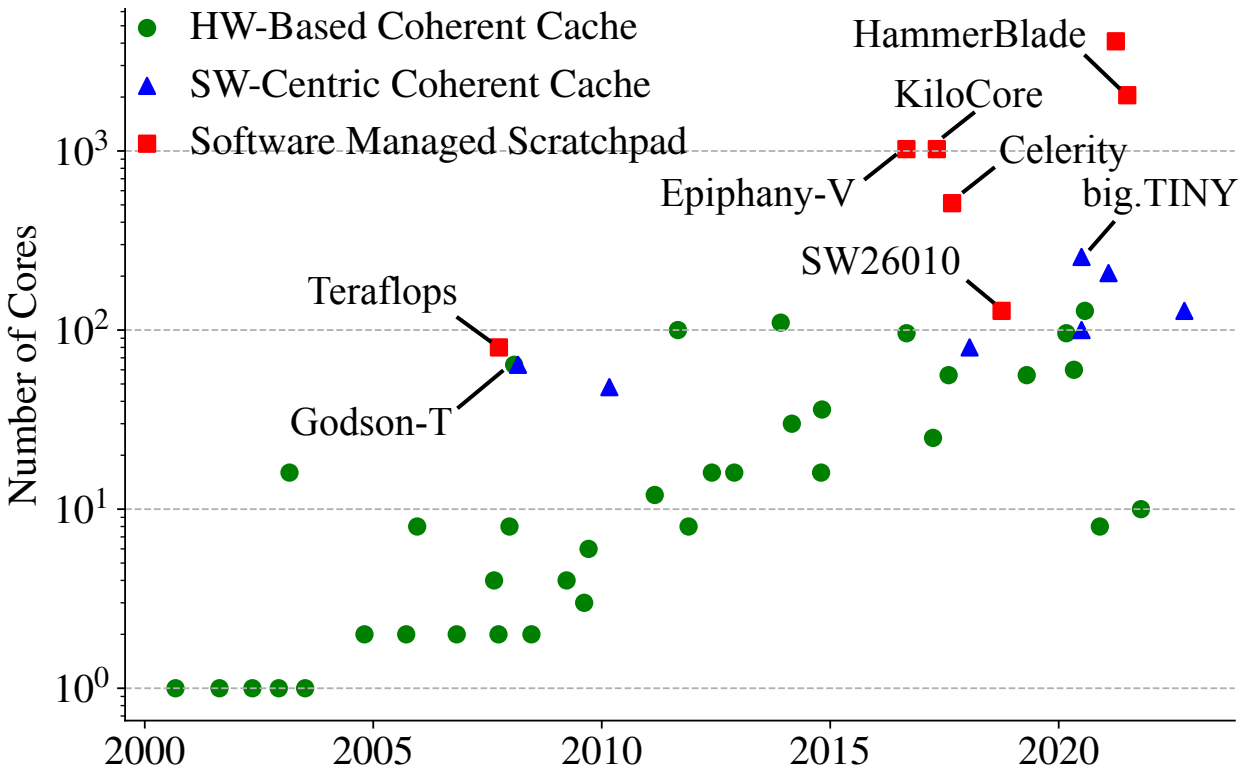

Figure 4.1: On Chip Memory Hierarchy in Manycore Architectures – SPM is needed for manycore architectures to reach very high core counts. Filled marker = real chip; unfilled marker = proposal/simulator only. Data is in part from CPU DB Danowitz et al. [2012].

tecture, work-stealing runtimes, and the manycore architecture programmability challenge. In Section 4.3, we describe in detail how to implement a work-stealing runtime, which is the core component of dynamic task parallel frameworks, on manycore architectures with software-managed SPMs. In Section 4.3.3, we discuss three optimizations for enabling the runtime to leverage SPMs and achieve high performance. In Section 4.4 and Section 4.5, we 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*. While conventional wisdom believes implementing a work-stealing runtime is either not viable or not beneficial on systems that do not have caches Zakkak and Pratikakis [2016]; Wang et al. [2020], our evaluation demonstrates that our proposed task parallel programming framework can achieve $1.2\times$–$28.5\times$ speedup for workloads that benefit from our techniques, and only induce minimal overhead for workloads that do not.

The contributions of this work are: (1) we provide, to the best of our knowledge, the first work that describes the implementation of a work-stealing runtime on manycore architectures with software-managed SPMs; (2) we summarize three optimizations which enable the runtime to leverage scratchpad memories to achieve high performance; and (3) we provide a detailed cycle-accurate evaluation using a silicon-validated RTL design of an open source manycore architecture.

4.2 Background

In this section, we give a brief introduction on dynamic task parallelism and the programmability challenge of manycore architectures. Note that HammerBlade , the specific manycore architecture targeted by this wor, is described in Chapter 3.

4.2.1 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 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. The most common computation model for dynamic task parallelism is the *fork-join* model. It was popularized by MIT Cilk Blumofe et al. [1995] and then adopted by various parallel programming frameworks Leiserson [2009]; int [2012]; Reinders [2007]; int [2019]; Charles et al. [2005]; Schardl et al. [2017]. 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 the *join* (also commonly referred to as *wait*) primitive is called. The parent task blocks until all of its child tasks have finished. 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); and (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 Blumofe et al. [1996b]; Frigo et al. [1998].

Work-stealing is likely the most widely-adopted scheduling algorithm for task parallel programming frameworks Blumofe and Leiserson [1999]. 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* a task from its own task queue from the tail (i.e., in last-in-first-out (LIFO) order). 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 is located higher in the task graph, which typically contains more work. The stealing mechanism automatically balances the workload across threads, leads to better locality, and helps establish time and space bounds Blumofe and Leiserson [1999]; Frigo et al. [1998].

4.2.2 Manycore Architecture Programmability Challenge

Manycore architectures that have high core counts (i.e., more than a hundred cores) and adopt software-manage scratchpad memories have been proposed and fabricated by both academia and industry Davidson et al. [2018]; Ajayi et al. [2017b]; Bohnenstiehl et al. [2017]; Olofsson [2016]; Brahmakshatriya et al. [2021]. While the hardware has gained most of the attention, the software stack of such architectures is less explored. As is the case with similar architectures, programming HammerBlade without loss of domain generality requires using a low-level C runtime environment. This demands that the programmer have both an extensive domain knowledge for their application and for the underlying hardware. Concerns such as data placement, synchronization, and load-balancing are left entirely to the programmer. Having to use a low-level C runtime environment prevents easily reusing existing code written for multi-cores and requires most applications to be completely rewritten for such manycore architectures.

Prior works propose leveraging a domain-specific framework approach to address the manycore programmability challenge (e.g., machine learning frameworks based on adapting PyTorch Cheng et al. [2022] and graph processing frameworks based on porting GraphIt Brahmakshatriya et al. [2021]). A domain-specific framework approach has three main drawbacks: (1) programmers need to rewrite their applications to use the constructs provided by the framework; (2) the framework is designed for a specific domain, meaning it is difficult to express computation from other domains; and (3) there is no easy way for a programmer who has little knowledge about the underlying manycore hardware to extend the framework.

4.3 Supporting Dynamic Task Parallelism on Manycore Architectures

In this work, we propose resolving 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,

```
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 4.2: Task-Based Parallel Programs – Examples for calculating the Fibonacci number using (a) a low-level API with explicit calls to **spawn()** and **wait()** , the implementations of which are shown in Figure 4.3; and (c) a high-level API with templated **parallel_invoke()** pattern. (b) shows the **Task** based class from which the **FibTask** class inherits in (a). (d) and (e) show alternative templated patterns **parallel_for()** and **parallel_reduce()** respectively.

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 through domain-specific frameworks, our framework is general-purpose. Furthermore, it provides an interface with which programmers that have used Cilk/TBB or OpenMP are familiar, making it possible to port 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 Blumofe et al. [1995], software-centric coherence Long et al. [2008]; Wang et al. [2020]; Tagliavini et al. [2018], and distributed memory Dinan et al. [2009]; Pezzi et al. [2007]; Saraswat et al. [2011] 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 Zakkak and Pratikakis [2016]; Wang et al. [2020].

In this section, we first demonstrate our programming model using running examples. We give details on how we implement a low-level API for spawning and synchronizing with new tasks. We also give a description of a higher-level API for expressing common parallel programming patterns. Lastly, We describe a naive implementation of a work-stealing runtime on the HammerBlade manycore, before discussing key optimizations in Section 4.3.3.

4.3.1 Running Example

We use an application programming interface (API) similar to Intel TBB to illustrate our programming model (see Figure 4.2). Each task is described by a C++ class derived from the **Task** base class (Figure 4.2 (b)) which contains an **execute()** method and a metadata variable **ready_count** , also known as the *reference counter*. This metadata tracks a task's unfinished children. After a task finishes 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). This mechanism enforces the ordering between parent and children: a task cannot complete until all of its children complete and join it (see Section 4.2.1). Programmers override the virtual **execute()** function to hold the logic of the concrete task. In this example (Figure 4.2 (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 the low-level APIs, our framework also provides templated functions that support various parallel patterns. This includes **parallel_invoke()** for divide-and-conquer (Figure 4.2 (c)), **parallel_for()** for parallel loops (Figure 4.2 (d)), and **parallel_reduce()** for parallel reduction (Figure 4.2 (e)).

4.3.2 A Naive 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 critical to both correctness and performance. 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 causes longer memory latency when accessing this shared data, allocating it in SPM would require software to keep it coherent, introducing significant software complexity. By allocating all data in the shared memory space, we can easily implement a naive work-stealing runtime that runs on the HammerBlade manycore architecture. Namely, the runtime does not utilize the scratchpads at all: all data lives 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 4.3 (a) shows an implementation of the **spawn()** and **wait()** functions for this naive 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 checks 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 resumes 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 from the queue of another thread, a victim. Tasks are stolen from the victim's 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 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 decrement have release semantics associated. This is to ensure that writes by child tasks complete before the parent task can exit from the scheduling loop (i.e., reference counter reaches 0).

4.3.3 Scratchpad Enhanced Runtime

Prior work has shown that leveraging the scratchpad memory is critical to achieving peak performance on manycore architectures Cheng et al. [2022]. However, SPMs 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 naive work-stealing runtime we introduced in Section 4.3 allocates all data, including both the stack and runtime data structures, such as the task queues, in DRAM. While this naive implementation yields a functionally correct work-stealing runtime, it is likely to have sub-optimal 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 SPMs 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 runtime that automatically utilizes SPMs on manycore architectures.

Scratchpad-Allocated Stack

Allocating the stack in SPM has been mentioned and explored by various prior work in the literature Cheng et al. [2022]. 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 when tasks are stolen; 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).

```
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 4.3: 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.

Data in the user code (e.g., **y** in line 14 of Figure 4.2 (a)) includes potential shared variables that 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 (see Chapter 3), and it enables us to keep unique copies of shared data in a core's SPM. 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 while 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 4.2.1) guarantees that reads and writes by `core_0` and `core_1` to `y` will not result in any data-race.

Manycore architectures like HammerBlade usually have limited per core scratchpad space (e.g., each core in HammerBlade has a 4 KB SPM). Applications running recursive algorithms (e.g., divide-and-conquer) can easily create deep call stacks, which cannot fit in the SPM. 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 work, 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 light-weight 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 the 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 allocate 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. The runtime calculates available stack space using the information given by the programmer at compile-time. It then allocate 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.

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 naive runtime implementation, all runtime data structures, including the core local task queues, are allocated in DRAM. Applications with 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 manipulate the local task queue efficiently is key to achieving high performance. The local scratchpad has a 2-cycle access latency where the DRAM has an access latency of tens to hundreds of cycles. Therefore, instead of going to DRAM for runtime data, we would like to keep them in the SPMs for faster accesses.

Similar to what we have mentioned in Section 4.3.3, 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 SPM. However, unlike the user data, to which a pointer is passed around dynamically, a core must know before run-time where other cores' task queues are located in order to conduct stealing without first accessing a DRAM allocated centralized data structure, such as the array of pointers to task queues (i.e., **tg[]** in Figure 4.3 (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 core. Figure 4.3 (b) shows an implementation of **spawn()** and **wait()** for our runtime which has both the stack and runtime data structures in the SPMs. The first noticeable difference is instead of loading the victim's queue from an array (line 18 in Figure 4.3 (a)), we calculate the address of victim's queue using the address of the

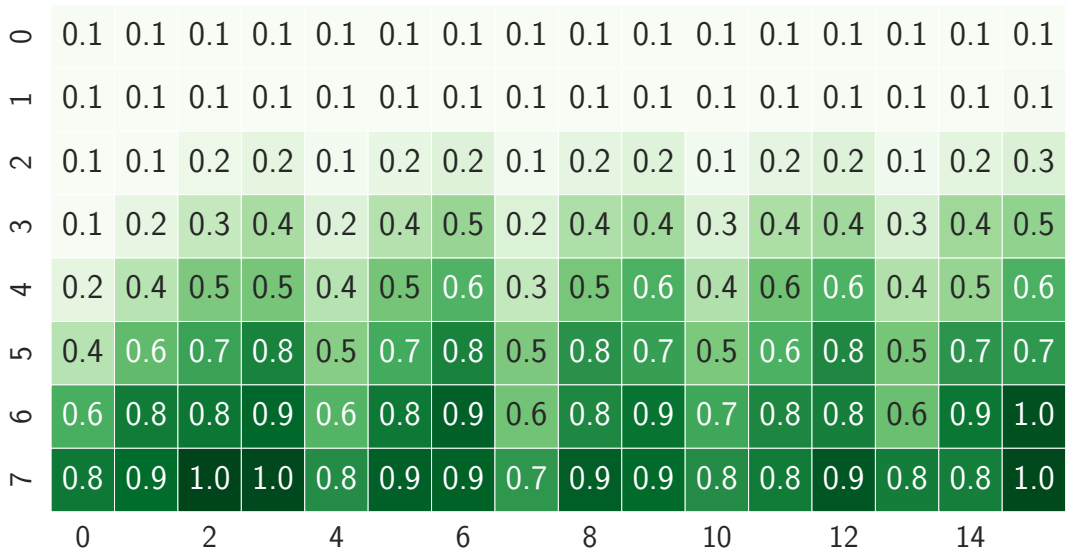

Figure 4.4: Normalized Remote Scratchpad Load Latency – Remote scratchpad load latency of 128 cores arranged in 16 rows and 8 columns, normalized to the core which has the highest latency.

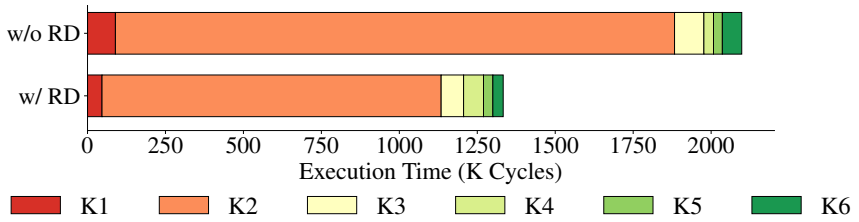

Figure 4.5: Performance Impact of Read-Only Data Duplication – Execution time of six parallel kernels (K1 to K6) in one iteration of PageRank with and without read-only data duplication optimization.

local queue (lines 18–19 in Figure 4.3 (b)). We also separate the spin lock protecting the task queue from the queue itself (lines 2–4 in Figure 4.3 (b)). Doing so allows us to directly calculate the address of the remote spin lock (lines 20–21 in Figure 4.3 (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.

Read-Only Data Duplication

After implementing the two optimizations described above, profiling data collected from the one of the apps (i.e., *PageRank*) shows an unexpected pattern. Figure 4.4 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 are located 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 are accessing `core_0`, the bandwidth in the 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 Jung et al. [2020]; Ou et al. [2020]. Our work-stealing runtime selects victims randomly, and thus we expect cores read and write their peers' 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, such as **parallel_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 reference capture (**&**), including for read-only values (e.g., pointers **dst** in line 5 of Figure 4.2 (d)). On systems with hardware-base or software-centric coherence, this read-only data can be cached and reused. However, in our case, these values are all allocated on the scratchpad of `core_0`, and thus other cores repeatedly load from `core_0`. This traffic to `core_0` causes congestion in the OCN. 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 4.2 (d) by value). We show the performance impact of the read-only data duplication optimization on *PageRank* in Figure 4.5. Each iteration of *PageRank* is composed by six parallel kernels. The proposed optimization is able to reduce execution time of all but one kernel, and achieve an overall speedup of $1.57\times$. 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.

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 4.2 (c) shows its implementation, and Section 4.4.1 provides details on the simulated hardware. *Fib* is known for generating many tasks each of which only contains a 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*

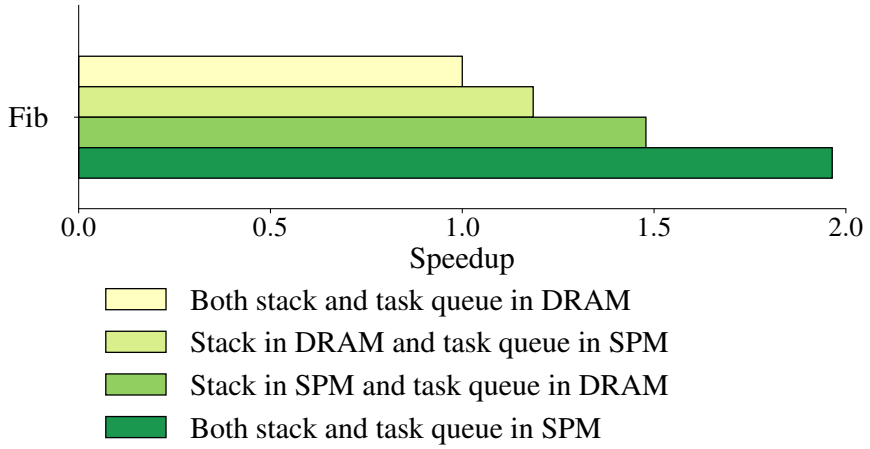

Figure 4.6: Speedup from Optimizing Data-Placement with SPM in Work-Stealing Runtime – Fib = measured speedups with the proposed SW/HW co-design scheme.

on four variants of the runtime: both stack and task queue in DRAM which is the naive implementation we introduced in Section 4.3, stack in DRAM and task queue in scratchpad, stack in scratchpad and task queue in DRAM, and both stack and task queue in scratchpad. Results are summarized in Figure 4.6. From the plot we can observe that, as we expected, the naive 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).

4.4 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.

Table 4.1: 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.

				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	pf	bundle1	42	50850	42	50718	183	12877	281	13409	189	12911	279	12992
	Transpose		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	ss	512	–	–	–	–	3	496	3	502	3	416	3	421
	Transpose		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

4.4.1 Simulated Hardware

We model the HammerBlade manycore architecture using cycle-accurate RTL simulation. We leverage an RTL simulator 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 Li et al. [2020]. 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. We model one HBM2 channel because, through experimentation, we found that 128 cores is required to saturate a single channel's bandwidth. 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.

4.4.2 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 4.3.3.

4.4.3 Workloads

We use a group of nine workloads to evaluate our proposed parallel programming framework, and the applications are summarized in Table 4.1. 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 three 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 the Ligra graph processing framework Shun and Blelloch [2013]. Both mainly use a pair of nested parallel loops: The outer loop iterates 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

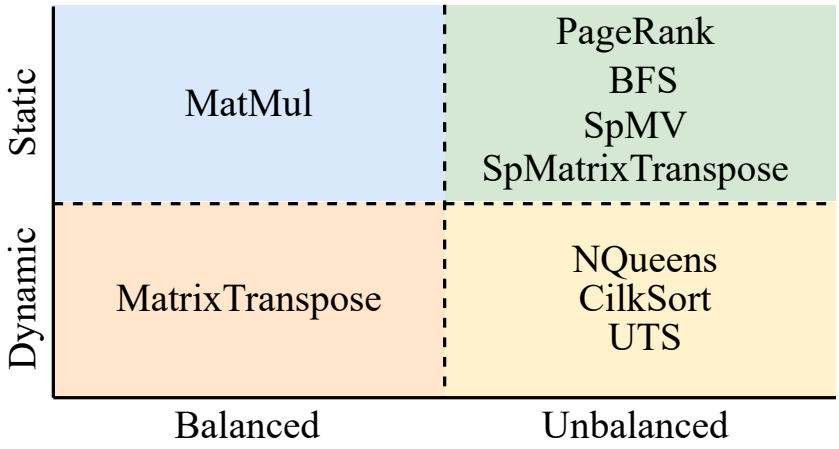

Figure 4.7: Anatomy of Workloads – we categorize workloads into four categories based on if he workload leverages dynamic parallelism and if the tasks have load imbalance

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 bactracking 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. Olivier et al. [2006], which contains recursive parallel loops to enumerate an unbalanced tree. 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 4.7).

4.5 Results

Table 4.1 summarizes the cycles and dynamic instruction counts of simulated configurations. Figure 4.8 shows speedup of workloads over a static runtime with stack in SPM. We plot *MatrixTranspose* and *CilkSort* separately in Figure 4.9, as they do not have static baselines. Comparing the left-most two bars in Figure 4.8, 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 operations when running with the static runtime, and thus leaving the

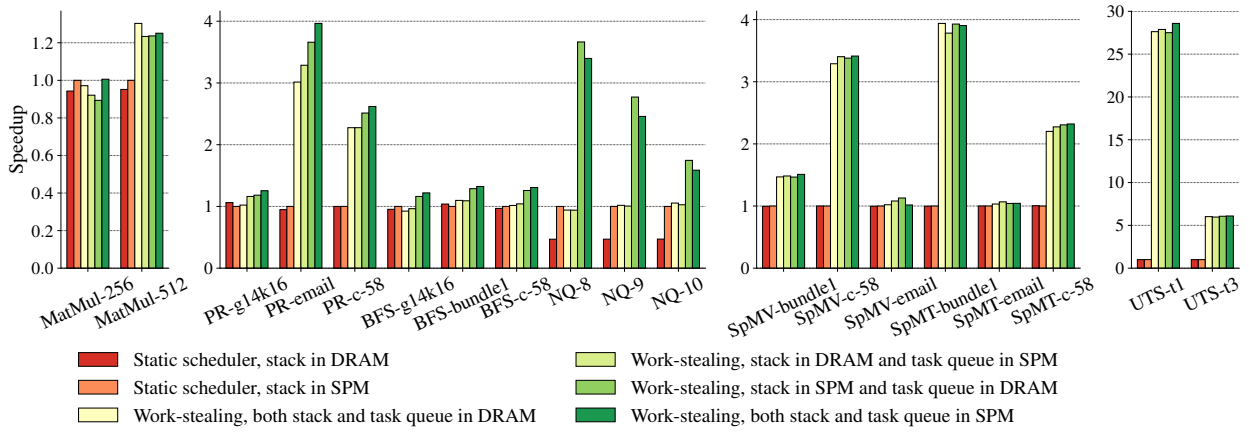

Figure 4.8: Work-stealing runtime provides a speedup between 1.2 - 28× and a slowdown of no more than 10% – PR = PageRank, NQ = NQueens, SpMT = SpMatrixTranspose. Applying data-placement optimizations to leverage the SPM provides an additional benefit of as much as 25% and compensates for any slowdown observed from work-stealing overhead.

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 the static scheduler that places stack in SPM to our baseline work-stealing runtime that 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× and 25× 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 achieve different speedups on different inputs (e.g., *PageRank* shows only moderate speedup on the synthetic graph *g14k16*, but achieves 3× 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 helps mitigate this difference by scheduling more compute to cores with lower memory latency.

Different workloads show varied benefit from our optimization techniques that leverage the SPM space not claimed by the programmer. *PageRank* is able to benefit from both optimizations and achieves best

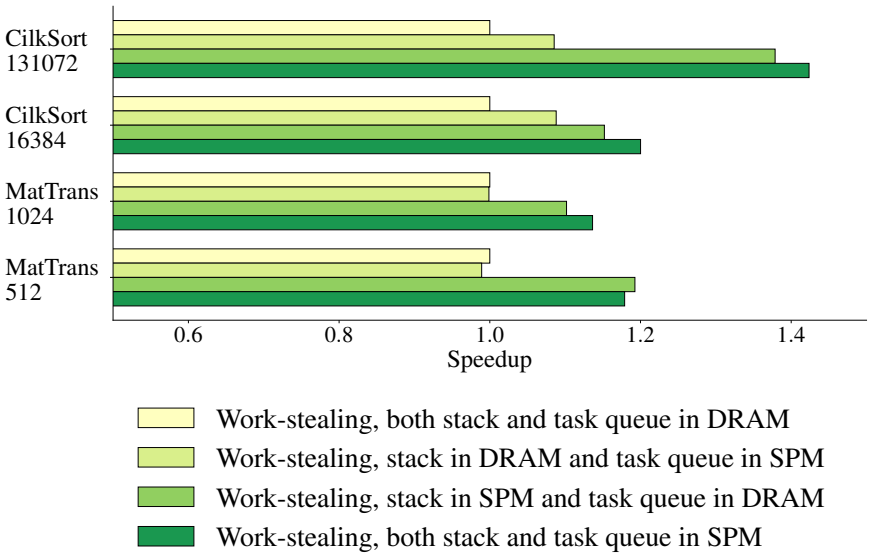

Figure 4.9: Performance of CilkSort and MatrixTranspose – normalized to having both stack and task queue in SPM; MatTrans = MatrixTranspose. Note that the X-axis starts at 0.5.

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. 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 stacks and thus more stack frame overflows to DRAM, *NQueens* becomes more DRAM bandwidth bound. *MatrixTranspose* and *CilkSort* are also able to benefit from having the stack in SPM (see Figure 4.9). *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 on these three workloads.

Across all workloads, we observe an increase in the number of dynamic instructions on work-stealing runtimes vs. on static runtimes (see Table 4.1). This is expected as it is well-known that work-stealing runtimes add overheads from various sources (e.g., task creation and scheduling), especially when working with very fine-grained tasks. We also observe an increase in the number of dynamic instructions when the

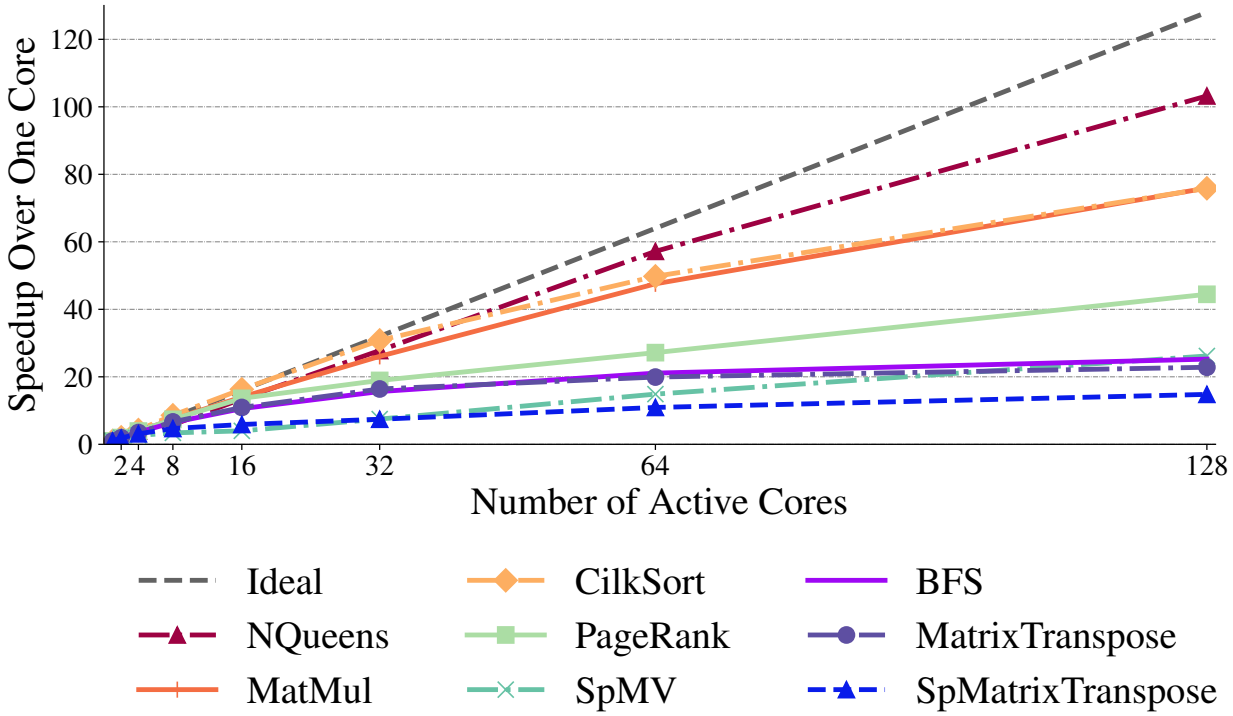

Figure 4.10: Workload Scaling – inputs: MatMul = 256; PageRank = g14k16; MatrixTranspose = 512; NQueens = 8; BFS = g18k8; CilkSort = 131072; SpMV = u16k32; SpMatrixTranspose = c-58. Data collected on work-stealing runtime with both task and task queue in SPM.

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 all workloads except *UTS*. We did not include *UTS* due to its extensively long simulation time. Results are shown in Figure 4.10. *NQueens* scales the best since, with more cores, more stack allocated data can be kept in SPM. *CilkSort*, as the name suggests, is an algorithm well suited to a dynamic task parallel runtime and is also well balanced, minimizing the overhead from stealing. *MatMul* is another balanced workload that scales well; it has high arithmetic intensity and loads from DRAM infrequently. *MatrixTranspose* is memory intensive and its scalability is limited by memory bandwidth. *BFS*, *PageRank*, *SpMV*, and *SpMatrixTranspose* are similarly bounded by memory, and in addition they can suffer from severe imbalance. While our runtime is a major boon to these workloads (static scheduling fairs much worse), task stealing becomes more frequent on unbalanced inputs as the core count increases.

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 dependent 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.

4.6 Related Work

Early manycore research prototypes integrated 16–110 cores on a single die Taylor et al. [2003]; McKeown et al. [2017]; Howard et al. [2010]; Hoskote et al. [2007]; Lis et al. [2013]; Vivet et al. [2020]; Tan et al. [2008]. The industry has adopted the manycore approach as well and products available typically include 64–256 cores Bell et al. [2008]; Ramey [2011]; Kanter [2015]; Wheeler [2020]; Halfhill [2020]; Wentzlaff et al. [2007]; Li et al. [2018a]; Kalray. Recent research prototypes have scaled core counts by an order-of-magnitude to over a thousand cores (e.g., 1000-core KiloCore Bohnenstiehl et al. [2017], 1024-core Epiphany-V Olofsson [2016], and 4096-core Manticore Zaruba et al. [2021])

A number of prior works explored work-stealing runtimes on manycore architectures that provide software-centric cache coherence. Long et al. Long et al. [2008] 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 stores all runtime-related shared data (e.g., task queues) into the Coherence Vector. For user data with parent-child dependency, they exploit the DAG-consistency Blumofe et al. [1996a] and insert L1 invalidate and write-back instructions in the runtime. Similarly, Wang et al. Wang et al. [2020] worked on a similar system (i.e., big.TINY) and also proposed inserting L1 cache invalidation and write-back instructions at proper locations in their Cilk-like runtime. Unlike Long et al. who identified runtime shared data as non-cachable 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. 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 send 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. Tagliavini et al. [2018] implemented an OpenMP runtime on a manycore architecture that has non-coherent private L1 caches. Similar to both works 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 works studied manycore architectures with software-centric cache coherence, while our work targets architectures that have only software-managed scratchpads. Orr et al. Orr et al. [2014] implemented a Cilk-like work-stealing runtime on GPGPUs with software-centric caches.

Although not a manycore, the Cray T3D/E architectures cra [1993]; Anderson et al. [1997] bear similarities to HammerBlade . Both are global shared memory architectures capable of parallel work-sharing programming models. A notable difference is that the Cray machines' notion of local memories pertains to abundant-but-slow DRAM, as opposed to HammerBlade 's local memories being fast-but-scarce SRAM. Nonetheless, we believe that techniques from this work could be applied to these Cray machines.

Zakkak et al. Zakkak and Pratikakis [2016] proposed an implementation of the Java virtual machine on a SPM manycore and adopted work-dealing instead of work-stealing. 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 SPM. Alvarez et al. Alvarez et al. [2015] described a task-based parallel runtime which can transparently use the SPM 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 Dinan et al. [2009]; Pezzi et al. [2007]; Saraswat et al. [2011]. Li et al. Li et al. [2010] studied efficient implementations of conditional division on manycore architectures. Their work focused on improving the work scheduling efficiency on top of an existing work-stealing runtime and is orthogonal to ours. Chen et al. Chen et al. [2018] and Margerm et al. Margerm et al. [2018] explored generating task parallel accelerators with coherent caches. Our work can be applied to support accelerators with SPMs.

4.7 Summary

We demonstrate that, in contrast to conventional wisdom, a work-stealing runtime is viable and beneficial on manycore architectures with only software-managed scratchpad memories. This work provides programmers a familiar programming model for efficient software development on manycore architectures like HammerBlade , and achieves significant performance improvements over traditional programming models such as statically scheduled parallel loops (i.e., up to $3.94\times$ speedup for workloads that can be statically scheduled and up to $28.5\times$ speedup for workloads with dynamic parallelism). This work is a small yet important step towards solving the manycore architecture programmability challenge. While we evaluated our work-stealing runtime on HammerBlade, our techniques are applicable to other PGAS manycore architectures that have software-managed scratchpads memories.

Chapter 5

Work-Stealing on One Thousand Cores

In the last chapter, I demonstrated that a dynamic task-parallel software runtime with a work-stealing scheduler is feasible on HammerBlade. I also showed that, on the selection of applications I examined, the runtime provides good performance scaling. However, this runtime was only evaluated on a single HammerBlade pod consisting of 128 cores. HammerBlade, however, is specialized to be a massively scalable fabric; a 128-core pod is just a fraction of a real-world HammerBlade system.

Fortunately, a new avenue for evaluation has become available. Our research group taped-out a 2048-core HammerBlade chip, named BigBlade, in 2021. That processor now, in 2025, is in our lab at the University of Washington and ready to run HammerBlade software. BigBlade presents an opportunity to evaluate how scalable and practical the runtime work is at system-scale.

In this chapter, as my final thesis work, I extend my research on the dynamic task-parallel system to run on real HammerBlade hardware, and on a full HammerBlade system. This is a progression from the 128-core system on which I evaluated my runtime work previously.

Scaling any system up by a factor of eight requires engineers to rethink how they have implemented their design. For example, HammerBlade is a non-uniform memory access (NUMA) architecture and spatial locality can significantly impact performance. A major downside to using a work-stealing scheduler, such as the one implemented in Chapter 4, is it renders exploitation of this locality significantly more challenging. Preserving the programmer's ability to leverage locality while maintaining the strengths of work-stealing is critical to successfully scaling up the runtime.

HammerBlade's native address space posed another challenge. It is a 32-bit architecture, with an address space too small to hold large data sets. Enabling a larger address space, without sacrificing the area savings from a 32-bit architecture, required creative hardware solutions with a non-standard hardware-software interface. Improper use of this interface can result in strange bugs that are very difficult to track down. The challenge posed by extending the address space required thoughtful and careful software engineer to solve.

Conducting research on real silicon presents new difficulties as well. Profiling and debugging on BigBlade are different beasts from doing so in simulation. While simulation has its challenges, namely the prohibitive wall-clock time, it can also have its benefits. In simulation, profiling counters and detailed instruction traces are available. Over the years, the HammerBlade team has built tools and infrastructure to give us insight into how our parallel programs behave. This can be as detailed as cycle breakdowns by program counter, or how many requests each of the cache banks receive. We are even able to see breakdowns on stall causes, such as how many cycles were spent waiting on long latency loads to complete as opposed to waiting for network availability. None of this infrastructure is available on the real hardware. I must therefore rely on micro-benchmarking, experimentation, and simulation of a smaller system to profile and debug software.

Another challenge is that my previous research relied on "soft" hardware design, meaning that, because I was working in simulation, I was free to change the hardware as needed. This enabled me to fix bugs and add features. For example, HammerBlade's RTL had a bug when reading or writing tile-group addresses that happened to point to the local core's scratchpad. The LSU did not decode this address and determine that it was local. Rather, it forwarded this to the tile's network router and the request eventually routed back to the scratchpad. While this did not cause a correctness issue, it was a performance problem, especially when setting the stack pointer to a tile-group address as we did (see Chapter 3 for details on this type of addressing). We were free to fix this issue, but this occurred well after the BigBlade chip was taped-out and sent to the foundry for fabrication.

Along the same lines, we were able to use stack space liberally thanks to a change we made to the RTL to decode addresses overflowing the scratchpads to point to DRAM. This enabled us to write our runtime in a manner that did not need to pay too much attention to scratchpad use. On BigBlade, however, I cannot use hardware hacks to surmount such issues. Rather, I need to seek out solutions in the software itself, often times imperfect ones.

A final challenge is that the scale of BigBlade is not its only difference from HammerBlade in simulation. The memory system is fundamentally different. The latency and bandwidth are significantly more challenging than the system I modeled previously. This significantly impacts performance and dramatically alters the trade-offs for using off-chip memory.

In this Chapter, I propose and implement software solutions to address the challenges described above. I demonstrate how the unrestricted work-stealing scheduler in the previous chapter provides poor performance on the larger HammerBlade system, and I propose a restricted work-stealing scheduler to replace it. I also detail a library-based solution to the extended address space of a multi-pod HammerBlade system. Furthermore, some of the lessons learned from my single-pod research are taken to adapt to larger systems to improve reliability.

5.1 Bigblade's Memory System

BigBlade's memory system is important context for understanding the design decisions I make in this chapter. It is also crucial to interpreting the results I present in Section 5.4.

BigBlade's memory system has some stark differences from what we on the HammerBlade team often model in simulation. There are good reasons for this. When we work in simulation for research purposes, we have the luxury of imagining that we have financial and manufacturing resources that, in the real world, we do not. Additionally, in simulation, we can reconfigure the design to optimize for a specific purpose or application. In silicon, we are committed to optimizing for the one purpose we chose during tape-out.

HammerBlade was designed to be a high-throughput fabric with the intention that it be equipped with a state-of-the-art high-bandwidth memory system. This would make HammerBlade's memory system competitive with similar systems and architectures built for a like purpose. Our simulation models assumed that HammerBlade's main memory system would be at least as state-of-the-art as HBM2 with a direct connection between the on-chip cache hierarchy and the memory substrate. This made sense for comparing HammerBlade to similar architectures. If we were trying to evaluate HammerBlade as a compute fabric, why would we constrain it to a slower memory system than its viable alternatives? However, direct integration with HBM is no small thing financially or logistically. Large semiconductor companies like NVIDIA or AMD have the financial resources and market scale to partner with Samsung, SKHynix, or Micron on direct

Parameter	BigBlade	Simulation	Simulation : BigBlade
Miss Latency	430 ns	55 ns	8
Revised Miss Latency	366 ns	55 ns	6.5
Bandwidth Per Pod	550 MB/s	14.5 GB/s	$1/26$
Revised Bandwidth Per Pod	2.5 GB/s	14.5 GB/s	$1/6$
Line size	32 bytes	64 bytes	2
Associativity	4 ways	8 ways	2
Pod cache size	256 KB	2 MB	8
Total cache size	2 MB	Pods $\times$ 2 MB	Pods $\times 8$

Table 5.1: BigBlade and Simulation Cache Configuration Parameters

integration with their chips. A small research group such as ours does not.

HammerBlade is a fabric designed for massively parallel software substrates. But that can mean almost anything in 2025. Although the architecture is general purpose, individual instances of it must be specialized. Do we want to accelerate memory intensive applications? What about applications with challenging spatial locality, such as sparse linear algebra kernels? If so, spending more area on a robust cache hierarchy would be to its advantage. Or, perhaps, we mean to accelerate workloads that are heavily floating point intensive, in which case it behooves us to fit as many fast FPUs on the chip as possible. This last case is the one for which the BigBlade instance of HammerBlade is most closely optimized.

As a result the cache hierarchy is slimmed down to accommodate more area for compute. Table 5.1 shows the cache parameters of BigBlade relative to the system modeled in Chapter 4. The total cache size per pod has $8\times$ less capacity. The associativity is 4-way down from 8-way, increasing the likelihood of misses due to conflicts. Furthermore, the cache line itself is half the size. Since BigBlade's caches are blocking and have no prefetcher, this would double the latency incurred per byte fetched from memory, assuming that the latency of a memory fetch on BigBlade matches that in simulation.

This assumption does not hold true, however. BigBlade is mounted on a PCB that routes off-chip memory traffic to HBM by way of an FPGA (the model is a Xilinx Virtex Ultrascale+ VU47P). Moreover, the interconnect between BigBlade's caches differs from simulation in that 4 pods will share a request pipeline to the off-chip IO, which is then routed through a cross-bar on the FPGA. In simulation, by contrast, each pod has a dedicated and direct pipeline to a single HBM channel that it owns. This results in dramatic discrepancies in memory bandwidth and latency between BigBlade and the system we typically model.

One major performance difference is the latency of a cache miss. I use a microbenchmark called **stride**,

```
struct node {
    node *next;
};

static int kernel_dram()
{
    node *p = &dram_node[0];
    int i = N;
    for (; i != 0;) {
        p = p->next;
        asm volatile ("addi %0, %1, -1"
                      : "=r"(i)
                      :  "r"(i)
                      : "memory");
    }
    result_node = p;
    return 0;
}
```

```
DRAM(int) *A; // set by host
int fetch()
{
    int *p = &A[(__bsg_x + bsg_tiles_X*(__bsg_y == bsg_tiles_Y
        -1)) * CACHE_LINE];
    bsg_barrier_tile_group_sync();
    // only the north and south rows participate
    if (__bsg_y == 0 || __bsg_y == bsg_tiles_Y-1) {
        // parameter N is total lines to fetch
        for (int i = 0; i < N/(bsg_tiles_X*2); i++) {
            asm volatile ("lw x0, %0"
                :: "m"(*p) : "memory");
            p += bsg_tiles_X * 2 * CACHE_LINE;
        }
    }
    bsg_barrier_tile_group_sync();
    return 0;
}
```

Figure 5.1: Shown at left is the **stride** benchmark used to measure memory latency. A single core executes a tight pointer-chasing loop. Each iteration of the loop has a read dependency on the previous one. On BigBlade, I run this benchmark many times and alter the source core and number of loop iterations. At right is the **fetch** benchmark used to measure achievable memory bandwidth. This benchmark uses a single core per bank in the cache array. A single word per cache line is fetched. This keeps the pod's cache array busy servicing misses at a near constant rate.

shown at left in Figure 5.1, to measure this latency consisting of a single core performing loads at a stride equal to a cache line. Every load will result in a compulsory cache miss. **stride** uses dependent loads to prevent any latency hiding from HammerBlade's non-blocking memory operations. Figure 5.2 shows a plot of the total runtime as I scale up the number of loads. I use a linear regression to estimate the latency of each load, shown as the slope, and the constant initialization overhead. Note that I repeat the experiment for a random selection of cores and take a mean. The result of this experiment shows that the expected latency of a cache miss on BigBlade is 430 nanoseconds[1]. In contrast, running this experiment on our simulated models shows a 55 nanoseconds latency, which is nearly an $8\times$ difference.

A second big difference is the memory bandwidth. Our simulated models assign each pod ownership of an HBM2 memory channel clocked at 1 GHz. Each of these channels has a peak memory bandwidth of 16 GB/s. I use another microbenchmark called **fetch**, shown at right in Figure 5.1, to measure how much of this bandwidth a HammerBlade pod can saturate. In **fetch**, the cores issue a single load to each cache line in a large vector. This is the minimum number of loads to prompt the cache memory system to fetch the entire vector from DRAM. Running this experiment in our simulation model, a pod's cache array saturates the memory system, achieving a rate of 14.5 GB/s which is 90% of a channel's line rate. Running the same

[1]This latency is revised to 366 nanoseconds after memory system improvements

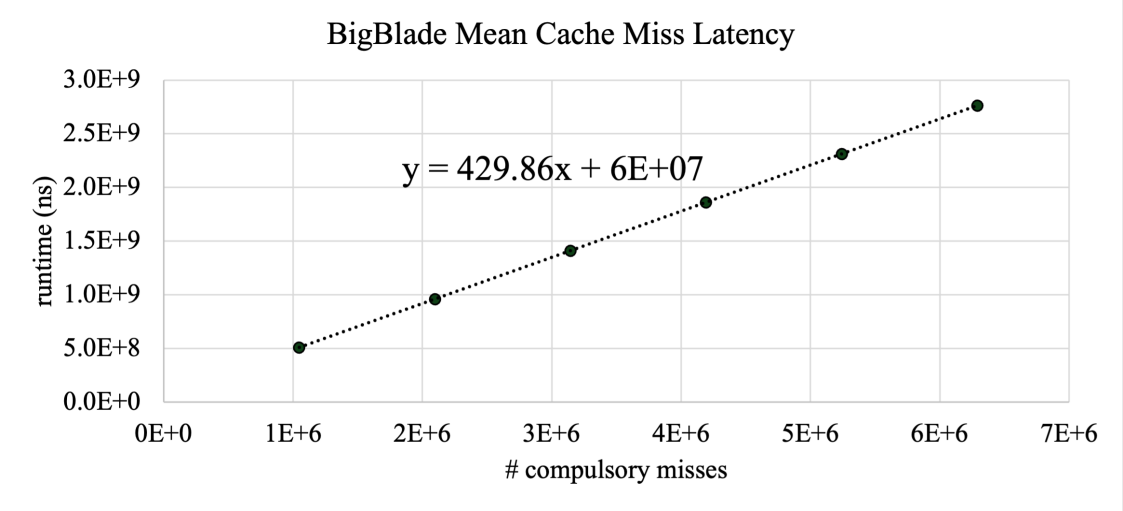

Figure 5.2: Results from a microbenchmark on BigBlade designed to measure memory latency. Each load results in a compulsory cache miss. I run the benchmark with a single core, varying the number of memory fetches and the source core. I then take a linear fit to account for initialization time. The slope of about 430 ns estimates the average latency of a cache miss.

benchmark on a BigBlade pod, I saw an effective bandwidth of 550 MB/s[2], which is roughly a $26\times$ reduction.

Revision: When I originally collected the data presented in this chapter, BigBlade's memory system contained bugs that were resolvable in the field. They have since been addressed. Table 5.1 shows revised bandwidth and cache-miss latency numbers with the improved memory system. All data presented in this chapter was collected before these issues were resolved.

5.2 Addressing the Extended Address Space

HammerBlade is a 32-bit system for a reason. Area is a constraint for a scalable fabric, especially when the aim is to maximize compute density and energy efficiency. BigBlade fits more than two thousand independent cores on a 99 mm^2 chip. That density would be much harder to achieve if they all required 64-bit register files and integer multipliers. BigBlade adopts a solution to this problem used by previously proposed manycore systems Karaoui et al. [2016] in which the physical address space is replicated across pods. In this way, each pod "owns" a 32-bit space and thus can operate within that space efficiently. This works well when we run

[2]This bandwidth is revised to 2.5 GB/s after memory system improvements

the same kernel on distinct data sets or use each pod to run distinct kernels that do not need to share any data at all.

Suppose, however, that we want to use the entire BigBlade chip to run a graphical analysis on a large network. Maybe that network does not fit in the 32-bit address space of a single pod. BigBlade's answer to this problem is an address translation control register that software can read and write at its discretion. This control register, implemented as a RISC-V control status register (CSR), serves as a 4-bit extension to the address space indicating the pod whose address space should be targeted by memory operations. When the core dispatches a memory operation to a remote endpoint, not to its own local memory, the 4-bit register is checked and the pod coordinates that it encodes are used to format the network packet. This 4-bit extended address control register provides a hardware mechanism to access the entire address space provided by BigBlade. This enables us to write parallel software on BigBlade to target one large data set that cores can access dynamically based on runtime dependencies.

There remains challenges for software to address to effectively use the extended address space. The control register is effectively a global control on where memory operations target. This introduces complexity when trying to use this register. Reliable software design favors orthogonality, meaning that side-effects from function calls or object construction are eliminated or restricted to the scope of those operators. Global control variables are antithetical to maintaining orthogonality in software because they introduce, almost by definition, a side-effect. For example, the physical memory location that a native 32-bit address references changes when the control register is set. This can have a major impact on software reliability in higher-level programming languages since not all memory accesses are explicit. One such common example of implicit memory accesses are stack spills and callee-saves. Another common example in object-oriented languages such as C++ is accessing class data members.

As part of my effort to extend the task-parallel runtime to run on the full BigBlade chip, I chose a library approach to provide reliable and effective support for software use of the extended address space register. The first two primitives I introduced are the **pod_address** and the **pod_address_guard** , shown in Figure 5.3. The **pod_address** is a wrapper around a 4-bit integer that is initialized by reading the vanilla core's CSR. It has *x* and *y* bitfields to identify the physical pod address on HammerBlade's network. The **pod_address_guard** class assists software by providing an API for creating a live-range in which a particular pod address is active.

```
class pod_address
{
public:
    // constructors, constants, etc.
    static pod_address readPodAddrCSR() {
        unsigned raw;
        asm volatile
        ("csrr %0, 0x360"
         : "=r"(raw)
         :: "memory");
        return pod_address{raw};
    }
    static void writePodAddrCSR
    (pod_address addr) {
        unsigned raw = addr.raw;
        asm volatile
        ("csrw 0x360, %0"
         :: "r"(raw) : "memory");
        return;
    }
    union {
        unsigned short raw_;
        struct {
            unsigned short px_ : pod_x_width;
            unsigned short py_ : pod_y_width;
        };
    };
};
```

```
class pod_address_guard
{
public:
     // save the old pod address
     //set the new pod address
    pod_address_guard(pod_address set) {
        set_pod_addr(set);
    }
    // restore the old pod address
    ~pod_address_guard() {
        set_pod_addr(save_);
    }
    // gets the pod address in the CSR
    pod_address get_pod_addr() {
        return pod_address::readPodAddrCSR();
    }
    // sets the pod address in the CSR
    void set_pod_addr(pod_address addr) {
        return pod_address::writePodAddrCSR(addr);
    }
    pod_address save_;
};
```

Figure 5.3: C++ classes to encapsulate the pod address and the scoping of the address.

The live-range is active as long as the guard object remains in scope. This helps restrict the side-effects of setting the global extended address space register by leaving it to the compiler to resolve automatic cleanup code in functions with complex control flow or multiple exit points.

pod_address and **pod_address_guard** are sufficient for many applications to use BigBlade's full address space. If the application has statically allocated data, or data that is allocated at the same 32-bit address on every pod, then these two software constructs are all that is needed. However, some applications are not able to determine the 32-bit address of the data needed at compile-time. These workloads need a fat-pointer, or an abstraction that encapsulates the full 36 bits of an address and facilitates memory operations to it.

My implementation of this fat-pointer is shown in Figure 5.4. The **address** class contains the 32-bit native pointer and the 4-bit extended address as data members. It provides the **read** and **write** member functions which use **pod_address_guard** and **pod_address** to set the control register and redirect the loads and stores to an absolute 36-bit address. The **reference** class provides a syntactic overlay to **address** by overloading the assignment and cast operators to invoke **write** and **read** respectively. **reference** is meant to mirror the syntax of a native C++ reference, or **lvalue** , object. A class with the syntax of a C++ pointer is easily derived from **reference** . The de-reference operator returns a copy of the **reference** and the arrow operator returns a pointer it.

```
class address
{
public:
    // updates the value pointed to
    template <typename T>
    void write(const T& other) {
        register T wv = other;
        register T* ptr = (T*)raw;
        {
            pod_address_guard grd(ext.pod_addr);
            *ptr = wv;
        }
    }
    // reads the value pointed to
    template <typename T>
    T read() const {
        register T rv;
        register T* ptr = (T*)raw;
        {
            pod_address_guard grd(ext.pod_addr);
            rv = *ptr;
        }
        return rv;
    }
    address_ext ext; //!< pod address etc.
    uintptr raw; //!< the raw pointer
};
```

```
template <typename T>
class reference
{
public:
    // copy assignment (read => write)
    reference& operator=(const reference& other) {
        *this = (T)other;
        return *this;
    }
    // assignment
    reference& operator=(const T& other) {
        write(other);
        return *this;
    }
    // cast
    operator T() const {
        return read();
    }
    void write(const T& other) {
        addr.write(other);
    }
    T read() const {
        return addr.read<T>();
    }
    address addr; //!< the address information
};
```

```
template <typename T>
class pointer
{
public:
    // dereference operator
    reference<T> operator*() {
        return ref_;
    }
    // arrow operator
    reference<T>* operator->() {
        return &ref_;
    }
    // indexing operator
    template <typename I>
    reference<T> operator[](I i) {
        return reference<T>
        (ref_.addr() + (i * sizeof(T)));
    }
    reference<T> ref;
};
```

Figure 5.4: The primitives to encapsulate a fat-pointer interface in C++. The **address** class implements **read** and **write** explicitly. **reference** provides a syntactic layer mirroring C++ reference objects. A **pointer** class is trivially derived from **reference** and overloads the dereferencing operators.

5.2.1 Discussion

Was a library approach the best solution to the extended address space problem? This is sufficient for prototyping a research project because it provides reliability and ease of programming. However, like any library approach, it is language-specific. My primitives rely on overloading C++ operators allowed by the language specification. Furthermore, from a performance perspective, it certainly has detractors. A compiler or language approach, which would be the most viable alternative, would enable optimizations such as removing redundant accesses of the control register or auspicious reordering of memory operations to 36-bit addresses. Because my approach relies on inserting inline assembly, around which the compiler must not reorder memory operations, no such reordering is possible. Furthermore, the compiler has no understanding of the control register's semantics and thus redundant accesses to it cannot be eliminated. I leave the merits of compiler support for this control register to future work.

It is also reasonable to wonder if the control register was the right hardware approach to extend the address space. Certainly, from a software perspective, 64-bit pointer support would have made using the extended address space simpler. Hardly any work at all would have been needed to port existing applications and libraries. But, as touched on earlier, this would doubtlessly have come at a high area cost and resulted in fewer cores.

Implicit memory operations, a feature of most high-level programming languages, is another challenge of the control register approach. On BigBlade, local scratchpad memory addresses are not impacted by this register. This turns out to be critical to making this approach practical for languages like C that rely on a memory stack. Without there being at least one constant address space in which stack memory can be placed, regions in which the control register's contents are modified would likely need to be written in assembly to ensure safe execution. On the other hand, it is certainly limiting that we are restricted to placing stack memory in precious scratchpad; Not all applications would prefer this.

Overhead from setting and restoring the control register's contents is another issue. On a system like BigBlade, the extraordinarily high memory latency dwarfs the several cycles of overhead from manipulating this register. If this were not the case, however, that latency would doubtlessly be a nuisance.

A more salient overhead on BigBlade is instruction memory footprint. BigBlade's instruction cache has capacity for only one thousand instructions and it is direct-mapped. There is thus a strong incentive to keep instruction footprint low. Excessive manipulations of the control register are a permanent instruction footprint overhead on the program. This is certainly a hazard of this approach for BigBlade software.

5.3 Runtime Library Design and Extensions

I carried over many of the design principles from previous work to BigBlade. Nonetheless, due to factors related to scaling, back-porting, and the extended address space there was a need to adapt certain elements of the library. I discuss these adaptations and extensions in this section.

5.3.1 Inclusive Linked List for the Task Queue vs a Ring Buffer

My previous work used a ring buffer instead of linked list structures to queue program tasks. Extending this project to run on BigBlade motivated me to explore more scalable solutions. The ring buffer has performance

```
class delegate_queue
{
public:
    // push task to target core's delegates
    void delegater_push(task * t) {
        task_list().push_back(&t->queued_);
    }
    // pop from my core's delegates
    task *owner_pop() {
        if (task_list().empty()) {
            return nullptr;
        }
        list_item*l = task_list().pop_front();
        return container_of(l, task, queued_);
    }

    bool empty() const {
        return task_list().empty();
    }

    list task_list;
};
```

```
template <>
class reference<lockable<delegate_queue>>
{
    void delegater_push(const pointer<task> &tp)
    {
        // cast this reference to 32-bit pointer
        auto *queue = to_local();
        task *ntv_ptr = tp.to_local();
        if (tp.pod_x() == pod_x()
            && tp.pod_y() == pod_y()) {
            {
                pod_address_guard gd(pod_addr());
                queue->delegater_push(ntv_ptr);
            }
        } else {
            register size_t size = tp->size();
            register task* dst_task;
            {
                pod_address_guard gd(pod_addr());
                dst_task = (task*)allocate(size);
                copy(dst_task, tp, size);
                queue->delegater_push(dst_task);
            }
            delete ntv_ptr;
        }
    }
};
```

```
void schedule()
{
    task *t = nullptr;
    // 1. check delegates
    t = my_delegates_ptr->owner_pop();
    if (t) {
        t->execute();
        delete t;
        return;
    }
    // 2. check local tasks
    t = my_tasks_ptr->owner_pop();
    if (t) {
        t->execute();
        return;
    }
    // 3. steal work
    int victim_id = fast_random() % num_tiles();
    if (victim_id == my::tile_id())
        return;

    auto victim_tasks = tasks_of(victim_id);
    auto stolen = victim_tasks->thief_pop();
    if (!is_null(stolen)) {
        execute_task(victim_id, stolen);
    }
}
```

Figure 5.5: Delegate operations are instances of **task** but they are dealt from one core to another. Note that unlike the default task queue, the **delegate_queue** is FIFO. When using a specialization **reference** for a **delegate_queue** , if pushing work onto a remote pod, a clone local to the target pod is made and the pusher cleanups the original copy. Cores prioritize delegates as shown in the updated **schedule** function.

advantages over a linked-list since it has minimal data-dependent loads and it has spatial locality when adding and removing items to the queue. However, ring-buffers have a fixed capacity which limits the number of tasks that can be spawned. This problem becomes more critical as the system and the inputs increase in size. This was not an issue in the previous work since it focused on a system with only 128 cores and the inputs were small relative to the ones I explore here.

Linked-lists have few space limitations. It is relatively cheap to place the head node of a linked-list in local memory since it is only a two pointer structure, and the tasks themselves can live in DRAM. Furthermore, pushing new work in LIFO order onto a linked-list should in theory work well with BigBlade's caches which have an LRU ejection policy. Since work is often pushed onto the queue in bursts, the list implementation of the queues should benefit from temporal locality. In order to avoid excessive dependent loads and suffering

extra latency penalties, the list itself is intrusive, meaning that the list nodes are embedded into the task structures.

5.3.2 Spawning and Stealing Work

Spawning and stealing functions similarly to how it does in my previous work. First, tasks are allocated and initialized. Next, the spawner appends the task to the back of its work queue. The spawning will pop from the front of its task queue in LIFO order. Thieves steal from the back of the queue in FIFO order. There are two differences in the implementation details from the previous work. First, I implement the task queues as lists instead of as circular buffers, as described above. Second, the tasks are allocated in DRAM by default.

Why allocate the tasks in DRAM? In Section 5.1 I showed that BigBlade's memory system poses significant challenges to off-chip memory performance. Given that, a reasonable engineer would think that storing the tasks in local scratchpad would be as beneficial as ever.

Practical limitations of the silicon-committed hardware prevents me from doing so. The previous work was conducted in simulation. This had its own challenges related to scale and simulation times, but a major benefit was that the hardware design was malleable. We took advantage of this fact to make the address space for scratchpad memory overflow to DRAM. This enabled placement of the stack in scratchpad without concern for the system crashing from overflowing the scratchpad memory. BigBlade lacks those hardware design changes.

In the previous work, we discussed a possible compiler change that could address this issue and make it feasible to implement the DRAM overflow mechanism in software. This would help with reliability. Nonetheless, we would still end up allocating application data in DRAM since it would not actually prevent scratchpad overflow.

I do believe there are opportunities in the compiler to improve performance and reliability for this runtime library on HammerBlade. However, time constraints force me to leave that research for future work.

5.3.3 Delegation

Work-stealing is known to be effective on multi-core machines especially with conventional memory systems. It is an open question how effective it is on NUMA architectures, particularly if processing elements can steal

work from across NUMA domains. An example of this on HammerBlade would be a core stealing a task from a victim on a different pod.

One can certainly imagine why this type of scheduling might not be ideal for BigBlade. Locality is often critical to efficiency on any system, but it is particularly impactful on an architecture with no private caching and only a very large scalar network for data movement. Although work-stealing is effective in placing the overhead of scheduling work on idle cores, it can also come at a sacrifice to coveted locality.

Previous work has recognized this problem and, in-fact, frequently taken it as a given that an unrestricted work-stealing scheduler would not be ideal Guo et al. [2010]; Acar et al. [2000]; Torng et al. [2016]; Shiina and Taura [2019]; Farooqui et al. [2016]. One solution is to accept an element of SPMD-ness to the programming model. This is often done in distributed systems. A second solution is active-messaging or delegation in which a processor can forward requests to a remote core to execute code on its behalf Nelson et al. [2015]. A third solution, particularly on single-node systems, is to allow work-stealing universally but to form a hierarchy in which stealing from other works tied to the same socket is prioritized.

I extended the task-parallel library by adding delegates as a feature. Figure 5.5 shows the implementation of the **delegate_queue** class, of which each core maintains one on its local scratchpad. Delegate operations are instances of **task** and are dealt from one core to another. Unlike the default task queue, delegate queues are FIFO. I also show a specialization of a **reference** for a **delegate_queue.** When pushing work onto a remote pod, a clone local to the target pod is made and the pusher deallocates the original. When a core is looking for work to schedule, it prioritizes checking its own **delegate_queue** for work before checking its default work queue.

It is reasonable to wonder why delegation would be a preferred solution to accepting a degree of SPMD-ness to the programming model. In Chapter 3, I give some perspective on the inherit benefits of SPMD. SPMD assumes parallelism and leaves it to the programmer to define the critical sections that must be serialized. The model explored by this runtime, by contrast, assumes that the program is serial unless the programmer specifies that it can be run in parallel. This gives the SPMD model an inherent advantage towards reducing overhead, since synchronization is assumed to be unnecessary by default, but disadvantages in terms of usability. I describe the limitations of SPMD and its less desirable programmability aspects in Chapter 4. A goal of this research is to investigate improvements to the programming model as it relates to how easily a

human can write a parallel program for HammerBlade. For this reason, I chose to explore a solution that did not require SPMD programming but rather maintained the task-parallel model in full.

5.3.4 Locking

HammerBlade's atomic memory instructions enable locking primitives. Specifically, the **amoswap** family of instructions makes it possible to implement spin-locks. The semantics of **amoswap** is a memory address is read and written atomically and the old value is returned and written back to a destination register. Additionally, the **amoswap** instruction comes with two flags **aq** and rl that control the memory consistency semantics. These flags are used to enforce memory ordering surrounding the acquisition and release of the lock.

On HammerBlade, there is an important semantic difference between invoking **amoswap** on a location in off-chip memory as opposed to one mapping to scratchpad memories. Invoking **amoswap** on a location in off-chip memory is unrestricted and any value can read or written using this instruction. The semantics of **amoswap** on scratchpad memory, however, is non-standard. In order to conserve area and mitigate occupancy hazards on the HammerBlade tiles, **amoswap** operations targeting scratchpad are remapped to a single one-bit register per tile. This is sufficient to implement correct spin-lock semantics for a lock whose value is either zero or one. It is limiting, however, in that all data structures using scratchpad memories to hold a lock share this register. Notably, this is relevant to this project in that each tile maintains two queues that require mutual exclusion support: one for spawned tasks and one for delegates from other cores.

Endpoint bottlenecks are a major concern on BigBlade. With more than a thousand cores in the system, if any one lock becomes a hotspot the impact on performance is widespread. Traffic trying to access a single endpoint will need to spill back into the network and this can have a cascading effect as a traffic jam extends along the network, delaying even memory traffic routed to a different endpoint by router and FIFO occupancy.

To mitigate the impact of hotspots and excessive locking traffic, I attempt two optimizations. First, I add exponential backoff to all spin-locks, a strategy known to significantly reduce memory traffic around spin-locks. Exponential backoff is especially potent when many cores are trying to acquire the same lock. This can certainly be the case in this system since fetching tasks from memory can result in a long latency cache miss. The second optimization is to provide a **try_lock** routine in which a core only attempts to acquire

a lock and stops trying if it fails to do so. This is a valid operation for cores that are idle and seeking work to steal from others. There is no correctness requirement or performance guarantee that this specific thief must steal from that specific victim. Furthermore, if a thief fails to acquire a lock it means either that the victim is working on its own critical path, in which case it should have priority, or some other thief is in the process of stealing whatever work there is. [3]

5.3.5 Removing Tail Recursion from Parallel Foreach

The method for implementing **parallel_for** in my previous work leveraged recursive task spawning to create a tree of sub-tasks. The motivation for doing this is to make the tasks that spawn more work stealable by idle cores. This strategy remains sensible on BigBlade, especially given that the time to spawn a task has only increased with a slower memory system. On the other hand, using recursion directly has major downsides that were surmountable when prototyping in simulation, but are rendered untenable once developing for hard silicon.

The salient problem is stack space utilization. My previous work showed that using the scratchpad for stack space had a sizable positive impact on performance. Although we did not show an energy impact, it is easy to imagine the high cost if we placed the stack in off-chip memory for all cores. However, scratchpad is an extremely scarce resource, and thus a stack placed in it easily overflows.

Some of the unnecessary recursion in **parallel_for** used in the previous work can be refactored into a loop instead. The number of sub-tasks that the current call to **parallel_for** needs to spawn directly is easily computable. It is the log of the number of "leaf" tasks. This avoids stack overheads associated with function calls such as spilling callee-saves. Furthermore, software can coalesce the allocation of multiple task's data into one buffer, the pointer to which needs only one register to be saved by the caller. This further reduces register allocation pressure and stack footprint.

[3] During my thesis proposal a committee member asked me if HammerBlade's memory consistency semantics ever impacted software. In fact, I encountered an interesting need for using release semantics, i.e. **amoswap.rl** while implementing tile lock software for task queues. We call HammerBlade "endpoint consistent" meaning that point-to-point memory traffic completes in sequentially consistent order. Thus, although HammerBlade nominally has a relaxed consistency model, many of its memory operations complete as if it were a sequentially consistent machine. However, the locks are not always co-located with the data they are guarding. Thus, it is possible to issue the memory operations on data and release the lock, but have the lock release before the data modifications have been committed.

```
// elements are distributed across pods
// alternating every STRIDE elements
vector<partial_table, STRIDE> C_product;
csr<float, int, ROWW_MAJOR> A;
csr<float, int, ROWW_MAJOR> B;
// calls the body over all elements in parallel
C_product.foreach([](int i, partial_table & result) {
    partial_table accum;
    // native pointers to A[i;*]
    auto [A_idx_start, A_idx_end, A_val_start, A_val_end] = A.row_lcl(i);
    value_type *A_val_p = A_val_start;
    for (index_type *A_idx_p = A_idx_start;
         A_idx_p != A_idx_end;
         A_idx_p++, A_val_p++) {
        index_type A_idx = *A_idx_p;
        value_type A_val = *A_val_p;
        // fat pointers to B[A_idx;*]
        auto [B_idx_start, B_idx_end, B_val_start, B_val_end] = B.row(A_idx);
        auto B_val_p = B_val_start;
        for (auto B_idx_p = B_idx_start;
             B_idx_p != B_idx_end;
             B_idx_p++, B_val_p++) {
            index_type B_idx = *B_idx_p;
            value_type B_val = *B_val_p;
            value_type C_val = A_val * B_val;
            auto C_entry = accum.find(B_idx);
            if (C_entry == nullptr) {
                accum.insert(B_idx, C_val);
            } else {
                value_type C_val = C_entry->val;
                C_entry->val = fma(A_val, B_val, C_val);
            }
        }
    }
    // write back
    result = accum;
});
```

Figure 5.6: Implementation of SpGEMM using both the **vector** and **csr** primitives. **vector** distributes elements across pods, pod indexes using the **STRIDE** parameter. It provides a **foreach** method that iterates over all elements in parallel and provides a native reference to the application-provided closure. **csr** distributes the row offsets and non-zeros across pods similarly. It provides iterator methods mapping to native and **pointer** primitives to row data.

5.3.6 Vector and Sparse Matrix Abstractions

There were two key challenges porting the six applications, discussed in 5.4, to the runtime library targeting BigBlade. First, these applications needed to be re-written from their SPMD style of parallelism to use the task-parallel library. Second, they needed to be adapted to distribute data across the address spaces of all participating pods. This last challenge was novel with respect to HammerBlade since, to my knowledge, this was the first time anyone had written a parallel program to execute with more than one pod.

Auspiciously, the data distribution challenge can be reduced to implementing a common data structure. To do this, I implemented a **vector** primitive that is responsible for (i) distributing the data across the address spaces of BigBlade's pods, (ii) providing an random-access interface for its elements, and (iii) providing a **foreach** method that maps a loop body to a parallel execution schedule. This schedule can use the **delegate** operations to place the loop body iterations on pods where the data that will be accessed resides. This common design pattern was sufficient to port most of the SPMD applications.

The exception was sparse matrix multiplication. Although the **vector** abstraction was helpful to port **SpGEMM**, some additional abstractions proved necessary. Leveraging **vector** , I also implemented a distributed compressed sparse row (CSR) sparse matrix format. The key features provided by this class are

initialization, data layout, and accessor methods for iterating over row data that may be local or remote to the executing core. Figure 5.6 shows in part the implementation of SpGEMM using **vector** and **csr** . Line 7 shows the invocation of the **foreach()** member that executes the loop body, provided as a closure argument, in parallel across all pods. Line 11 invokes **row_lcl()** which may be called if the row data is known to be resident on the calling core's pod; It returns native pointers to the column indices and non-zero values. This is in contrast to **row()** , invoked on line 18, to acquire extended address space fat-pointers to the row data that may be resident on any pod.

5.4 Evaluation

In this chapter I have introduced BigBlade, a 2048-core HammerBlade ASIC, described its memory system, and have introduced software extensions to my task-parallel library to run on it. In this section I evaluate those extensions. First, I attempt to quantify the cost of spawn and delegate operations using a microbenchmarking approach.

Next, I evaluate the effectiveness of the optimizations described in Section 5.3. I do so by porting six applications from SPMD-style implementations the performance results from which were showcased in Jung et al. [2024]. I also give a performance comparison between the ported applications and their SPMD-style counterparts. Finally, I conclude with a scaling study to understand how the runtime as a whole performs on a kilo-core scale system.

5.4.1 Application Suite

The applications used in this evaluation are shown in Table 5.2, along with their inputs. They were used in Jung et al. [2024] to evaluate HammerBlade for both its programmability and scalability. They were deemed suitable for evaluating HammerBlade's programmability because they were each examples of different parallel programming motifs, or "dwarfs" defined by Asanovic et al. [2006].

Here it is worth explaining the rationale behind the inputs I selected. Four of these six benchmarks are implementable as a large parallel loop over a large vector. Indeed, I use the **vector** primitive described in Section 5.3 to handle data placement and scheduling for all four of them. These four benchmarks are AES, Barnes-Hut, Black-Scholes, and Smith-Waterman. I would expect to observe a runtime overhead over the

Benchmarks (Abbrev.)	*Dwarfs*	Input Data
AES (AES)	Combinational Logic	64K and 1K ($\times$1KB messages)
Barnes-Hut (BH)	N-Body	1K and 64K bodies
Black-Scholes (BS)	MapReduce	1M and 4M options
2-D FFT (FFT)	Spectral Method	256 $\times$ 256 points
Smith-Waterman (SW)	Dynamic Programming	8K and 1M sequences
Sparse Matrix Multiplication (SpGEMM)	Sparse LA	See Table 5.2b

(a) List of benchmarks and their corresponding *Dwarfs* from Asanovic et al. [2006].

Name (Abbrev.)	Type	Edges	Vertices
wiki-Vote (WV)	Social	103689	8297
email-Enron (En)	Social	367662	36692
roadNet-CA (CA)	Road	5533214	1971281

(b) List of sparse matrices used from Davis and Hu [2011]

Table 5.2: Six parallel benchmarks ported from work presented in Jung et al. [2024].

SPMD-style baselines. I would also expect that overhead to be most severe on small inputs but for that overhead to amortize on larger ones. I use both a small input and a large input for each of these applications to confirm that this amortization occurs.

5.4.2 Overheads of Spawning and Delegating Tasks

Given the differences in BigBlade's memory system, as overviewed in Section 5.1, coupled with the changes to runtime implementation, it is prudent to quantify the cost of spawning and delegating tasks. I attempt to do so once again using microbenchmarks as I did for evaluating BigBlade's memory system.

Spawning: I use a benchmark called **spawn** to measure the runtime overhead of spawning a new task pushing it to the front of a core's task queue. **spawn** is a tight loop in which a core allocates memory for a task, initializes it, and invokes the runtime's subroutine for enqueueing the new task. I run **spawn** many times, varying the total number of spawned tasks to estimate the cost with a linear regression, shown in Figure 5.7. The total cost of spawning is around 630 ns.[4] This extraordinarily expensive operation is dominated by allocating and initializing the task in memory. To show this, I run the same experiment but compiled out the region of code in which the task is pushed onto the work queue. The same analysis as before shows that initializing the tasks accounts for 85% of the runtime.

Why is spawning so expensive? In Table 5.1, I showed the latency of a DRAM access on BigBlade, which accounts for most of this cost. A design choice I made for BigBlade was to allocate task memory in off-chip

[4] A memory system improvement that has since been applied to BigBlade would likely improve the cost of spawning stated here.

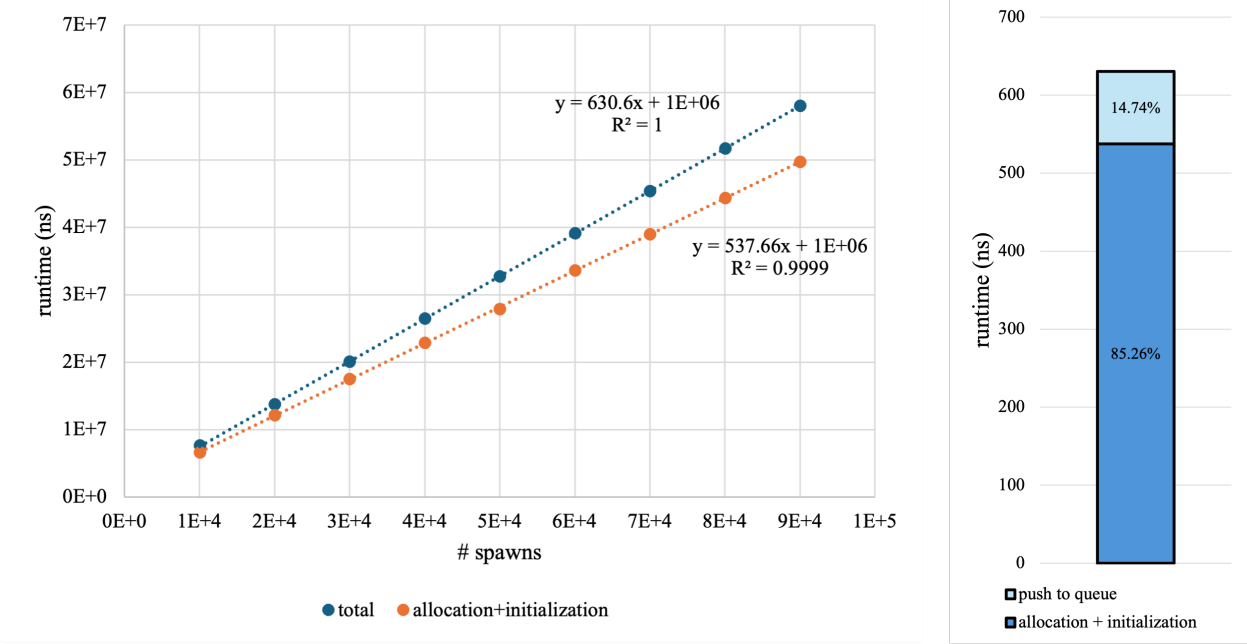

Figure 5.7: The costs of spawn collected using a benchmark that spawns a task in a tight loop over and over. Running it many times while varying the number of spawns enabled me to estimate the cost to **630 ns**. Additionally, I ran the benchmark without pushing the tasks to the work queue and found the allocation and initialization of the task accounts for about **85**% of the time.

memory rather than in core local scratchpad. I made this choice to conserve stack scratchpad memory for use by the application and to improve system reliability to avoid stack overflows. I expand on this decision in Section 5.3.2. Nonetheless, this design choice comes at an overhead cost for spawns. Allocating task memory often means writing to a buffer that has not been accessed recently, resulting in a DRAM fetch. [5] As a point of comparison, I ran the **spawn** benchmark using the scratchpad for task storage. Using a scratchpad reduced the cost of spawning to around 55 ns, which is more than an 11× speedup. However, the unreliability of storing tasks in such a limited stack makes doing so impractical.

Delegation: I use a benchmark called **delegate** to measure the cost of delegating a task to a core on a remote pod. **delegate** cycles through all pods round-robin to issue delegation requests. When spawning a delegate to a pod, a core on that pod is selected at random as the target. This mirrors how delegates are used in the data structures I describe in Section 5.3.6.

Figure 5.8 shows the results and an analysis of running this benchmark. I plot the run time as I vary the

[5] It is worth noting that these fetches would not be needed if the BigBlade caches had write-validation, a feature that was added in a later design.

number of delegate tasks spawned and I do so for four different task sizes. Recall that the size of task is related to its closure size, such as values captured in a C++ lambda for example. The 20-byte task payload is effectively an empty task containing only meta-data such as its virtual table pointer. Before optimizations, the expected cost of delegating a task to a remote pod is over 5.4 microseconds for the larger delegates and around 2.7 microseconds for the empty task. Since a cache line on BigBlade is 32-bytes, we can surmise that the roughly $2\times$ difference in overhead is related to requiring at least two lines for storage.

Over 5 microseconds is a hard cost to accept for such a critical operation. Examining the assembly from this benchmark, a key region of code stood out as a bottleneck. The subroutine that transfers data from the local pod to the remote one leverages the **pointer** primitive for transferring data. The baseline code transfers a single byte at a time. This is ripe for optimization. First, byte transfers can be coalesced into words to reduce the total number of packets sent over the network. Second, the loop itself can be unrolled to take advantage of HammerBlade's non-blocking memory operations. This is particularly advantageous in this instance when software is sending several packets across the entire chip. Shown at left in Figure 5.8 is the results from **delegate** after applying these optimizations. It is evident that the cost is far less sensitive to the size of the task itself; All four task sizes have similar time costs. Additionally, the cost of a delegation is reduced from 5.4 microseconds to roughly 1.2 microseconds. This is a greater than $4\times$ improvement. Nonetheless the cost of spawning a delegate remains high. Thus, it is prudent to avoid delegation in the inner loops of application code.[6]

5.4.3 Optimizations

Here, I evaluate the optimizations and design decisions I made while extending this library to BigBlade. Additionally, I explore two additional design parameters. First, I explore an alternative random number generator for victim selection. The original work uses a linear congruential generator. I found that using an xor-shift generator often improved performance particularly on smaller inputs. Second, I refactored the library's organization to help avoid caller and callee program text from conflicting with respect to the instruction cache. Instruction cache misses can be particularly pernicious on BigBlade due to its high-latency and low-bandwidth memory system.

[6] A memory system improvement that has since been applied to BigBlade would likely improve the cost of delegation stated here.

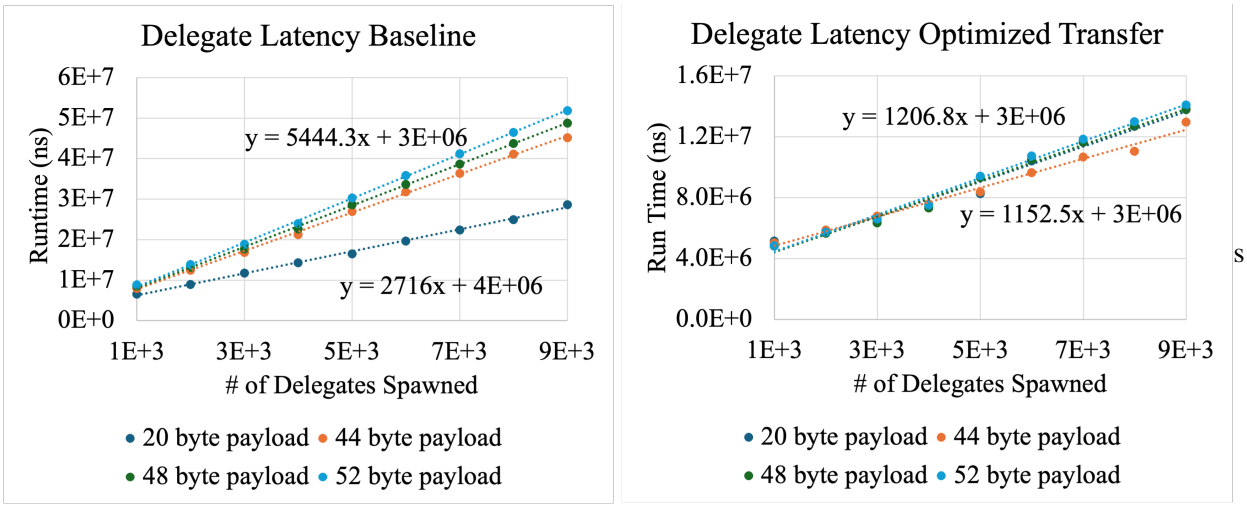

Figure 5.8: Delegation cost as measured using the **delegate** benchmark. I run the benchmark varying the size of the delegate closure and the number of delegates issued. The 20-byte payload is all meta-data i.e. an empty task. I selected the other delegate sizes based on what I observed from applications I ported. Above, I show the benchmark before any optimizations. The 20-byte payload takes half as the other because it only spans a single 32-byte cache line. Below, I show the results after optimizing the routine that transfers the delegate data to the remote pod using the **pointer** primitive.

In Section 5.3.3 I explained my design choice to implement delegation to address the friction between a work-stealing scheduler and preserving spatial locality. Here I evaluate this decision by implementing an unrestricted work-stealing scheduler. In this version of the scheduler, any core on the chip can steal work from any other regardless of their pods.

A comparison between unrestricted work-stealing and restricting it to a pod can be seen for four applications in Figure 5.9.[7] Speedups are normalized to the runtime that restricts work-stealing to within a pod. Perhaps surprisingly, the only application that seems particularly sensitive to the scheduler policy is AES. AES operates on 1 KB buffers, so the cost of a locality mismatch between execute and compute is high. FFT has a similar program property. However, FFT performs more work on the data once it is in scratchpad, which better amortizes the cost of data movement.

Another surprising result is that exponential back-off and the use of **try_lock** degrades performance. The **Simpl. Lock** actually shows improvement from removing the locking optimizations in preference for a simple spin-lock. Exponential back-off is useful when one spin-lock becomes a bottleneck, but can be detrimental in other cases. If it is not ameliorating contention, then exponential back-off is bloat at best. The

[7]This data was collected before memory system issues were resolved, as discussed in Section 5.1.

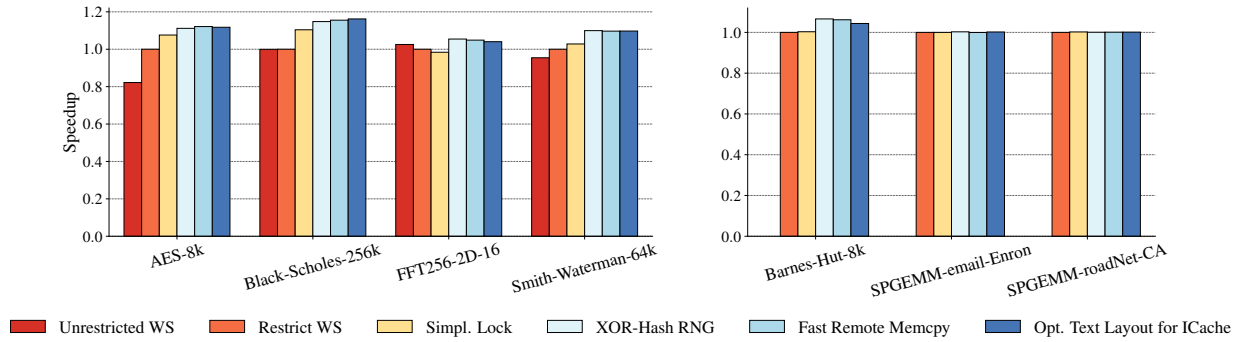

Figure 5.9: Cumulative impacts of different runtime design optimizations. Speedups are normalized to a design that restricts work-stealing, and uses locking optimizations and a linear congruential number generator for victim selection.

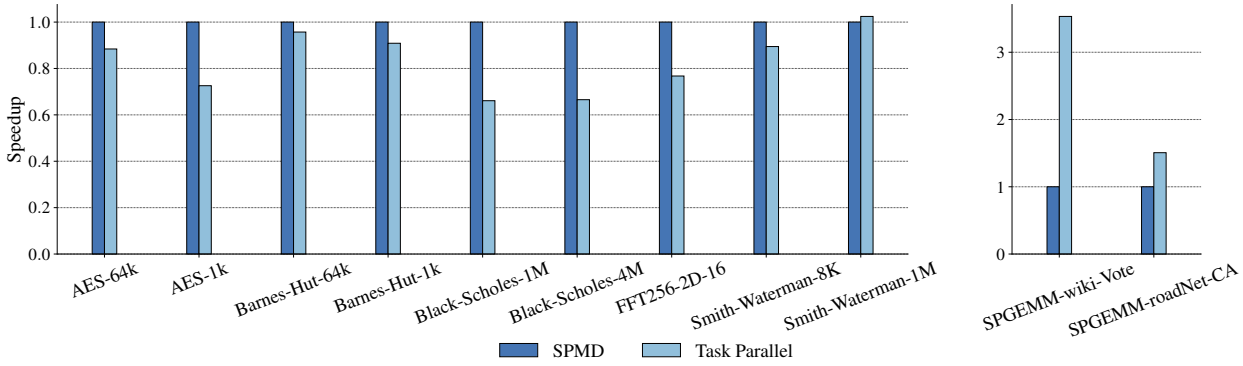

Figure 5.10: Comparison of six applications written in the SPMD model to task-parallel counterparts.

same can be said for **try_lock** . In addition, thief cores use **try_lock** to "give up" and look elsewhere for work to steal if they cannot acquire the lock. On the other hand, if the victim's lock is held by another thief in the process of stealing work, that may indicate the victim has even more work to steal.

Generally, the impact of these optimizations is less than 20%. One would hope this indicates that the majority of the runtime is driven by the application code, and that the dynamic task-parallel library introduces minimal overhead. If that were the case, any optimization made to the library would have minimal impact on the total runtime.

5.4.4 Comparison to Single Program Multiple Data

I evaluate the overheads of my library on the six applications shown in Table 5.2 and find that the overheads are highest when the amount of work per spawn is low. These results are shown in Figure 5.10.[8] The running times are normalized to those of the six applications in the SPMD style. These applications are run as they were written for Jung et al. [2024] without modifications. All results are run on a single pod, since the baseline sources are not written for multi-pod execution. AES, Barnes-Hut, Black-Scholes, and Smith-Waterman are all semantically implemented as do-all loops. Thus I would expect that when the loop is small, the overhead to be higher but to see that overhead amortize on a larger loop. That is what happens for AES, Barnes-Hut and Smith-Waterman; the large inputs have overheads of less than 10% and in the case of Smith-Waterman the overhead is completely amortized.

Black-Scholes does not see any overhead amortization as the input increases. Rather, the performance consistently degrades 30% even when the work-to-spawn ratio increases by 4×. This suggests that some overhead has leaked into the work loop of this application. Profiling in simulation on a smaller input showed that using the library introduced 9.8 instruction-cache misses per kilo-instruction. The baseline version fits entirely in the tile's 4 KB instruction cache. Instruction-cache misses on HammerBlade can be particularly detrimental to performance if all tiles are missing on the same instructions. This is because HammerBlade's secondary instruction storage is the pod's shared cache array, which are blocking. On BigBlade, this is compounded by the DRAM latency discussed in Section 5.1. Furthermore, misses on the same instruction will map to the same cache bank and can cause hotspots and traffic jams in the network. Indeed, HammerBlade's instruction-cache poses a significant challenge to implementing any library with sizable instruction footprint.

Sparse matrix multiplication is another anomaly. One might be tempted to believe that the library has improved the performance with load-balancing. This is unlikely. The original SPMD-style source load-balances by leveraging atomic add operators. However, the implementations have a salient difference in their work loops. The data structure used to store partial results is a key element of the algorithm. The original source uses a linked-list structure. As many as possible of these list nodes are stored in scratchpad to maximize reuse. Unfortunately, I was unable to provide a working implementation using the original list implementation along with my library. This might be due to stack overflows, since the task-parallel library

[8] This data was collected before memory system issues were resolved, as discussed in Section 5.1.

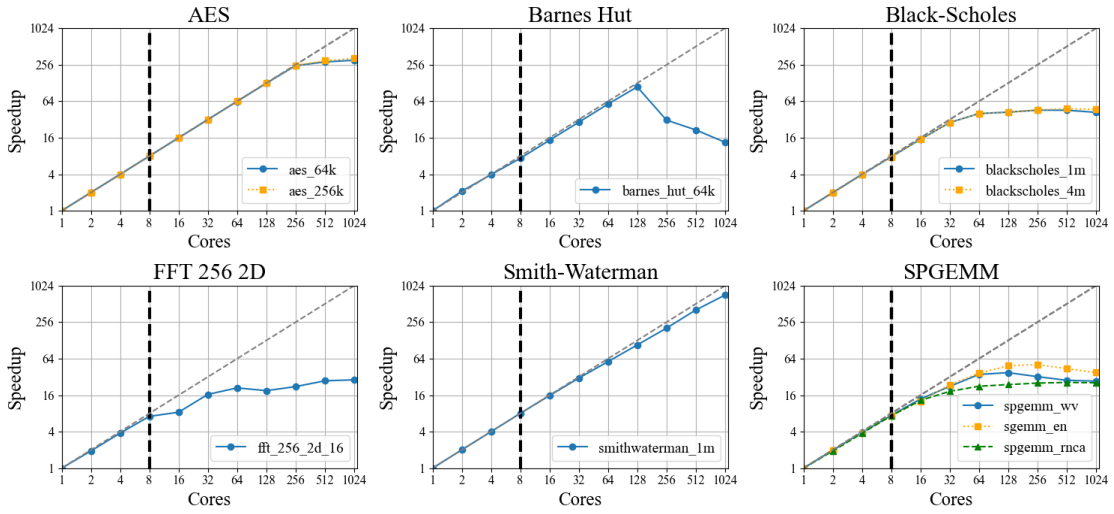

Figure 5.11: Scaling trends for six applications written using the dynamic task library. The vertical lines mark where 8 pods are in use and we start scaling up more cores per pod. Both axes are in log scale.

consumes stack aggressively, or it could be due to a bug in the original source. Ultimately, the version I implemented here uses a balanced binary tree to store partial sums instead. This has an algorithmic complexity improvement over a list for merging results, and this is most likely the reason for a $3\times$ improvement on wiki-Vote and $1.5\times$ on roadNet-CA.

5.4.5 Scaling

Figure 5.11 shows the scaling of the six applications from a single core to system scale.[9] The vertical line on each plot marks the core-count at which the applications are running with all pods and subsequent data points are increasing the number of cores used per pod. All workloads scale close to ideally up to this point. This is expected since increasing the number pods increases the number of cores, shared cache banks, and memory channels. Smith-Waterman and AES scale best, with Smith-Waterman almost reaching ideal scaling all the way to 1024 cores. Both of these applications benefit from requiring no synchronization, other than that imposed by the runtime as overhead, high operational intensity, and balanced workloads.

Black-Scholes shares these characteristics, and yet we see a performance plateau at 32 cores. As seen in Figure 5.10, using the task-parallel library appears to have introduced an inefficiency in Black-Scholes. I conjectured above that this may be due to instruction-cache misses. This would certainly pose a scaling

[9]This data was collected before memory system issues were resolved, as discussed in Section 5.1.

challenge since the secondary storage for instruction memory is the shared cache, of which there are $4\times$ fewer banks than there are cores. Furthermore, if the misses are triggered on the same instructions, all cores will need to fetch instructions from the same cache bank making it a hotspot.

Sparse matrix multiplication has a high memory intensity and some synchronization in between bulk synchronous parallel phases. Furthermore, the memory accesses are random and this can impact load-balancing on the cache banks. FFT also has frequent synchronization relative to some of the other applications which is likely contributing to its poor scaling.

The most stark anomaly in the scaling data is Barnes-Hut. It scales well until running with 256 cores, at which point performance sharply declines. Barnes-Hut has a couple characteristics that would impact its ability to scale. It is memory intensive and the accesses are random across the memory space of the chip. Furthermore, the outer loop over which the work is parallelized is imbalanced since each body requires varying depths of traversal in a tree. Neither the work-imbalance nor the memory intensity fully explains why Barnes-Hut performance sharply declines as more cores are used. A more likely culprit is memory system imbalance. For each body, the kernel traverses a tree depth-first from its root. Nodes shallower in three are visited most frequently. Thus, cache banks in which these nodes are resident will become hotspots. If the hotspots are severe enough, then the memory traffic can backup the on-chip network and impact other ambient endpoints.

5.5 Related Work

Guo et al. [2010] introduces a SLAW, Locality-Aware Work-Stealing scheduler that relies on programmer-provided hints on what "place" tasks should execute. Workers are restricted to stealing work from their own place which this work defines as workers who share a common LLC. They will first attempt to execute from their own queues, then they will attempt to steal from other workers in the same place, and lastly they will check a mailbox that is shared by all workers in the same place. This work was conducted on a multi-core systems with private L1 and L2 caches, unlike HammerBlade. Nonetheless, my approach is similar if you consider a HammerBlade pod to be analogous to a place in their work. The key difference is, rather than having a shared queue per pod, a delegation target is selected at random among a pod's cores. This avoids contention on any work single queue. Min et al. [2011] propose an extension to SLAW, enabling the

programmer to define place hierarchies. Workers prioritize stealing from victims according to this hierarchy, starting with threads closer in the place hierarchy. Both works focus on multi-core systems with traditional memory hierarchies including private caches with coherency.

Shiina and Taura [2019] propose Almost Deterministic Work-Stealing in which tasks are deterministically allocated to workers, but hierarchical work-stealing is allowed. This is, in essence, a hybrid approach of work-stealing and work-dealing. What makes their approach novel is that they embed the hierarchy into the tasks. When a worker becomes idle, it searches for work among its current task's worker pool. If it cannot find work, it searches in the range of that task's parent's worker pool. This approach embeds a locality-bias into the distribution of work to which idle cores will adhere. It requires hints from the programmer of the workload distribution among tasks. Similarly to Guo et al. [2010]; Min et al. [2011], this work was evaluated on a server class multi-core CPU with a robust and coherent cache hierarchy, although this approach could certainly be explored on HammerBlade by extending the scheduler proposed in this thesis.

Torng et al. [2016] explores the design of a task-parallel work-stealing system across a manycore architecture with asymmetric compute resources. They propose a scheduling scheme that involves idle fast cores "mugging" slow cores when the idle core is unable to find work. This "mugging" feature leverages an interrupt to steal the smaller cores context, including register state, when the small core is in the process of executing a task. Mugging ensures that the fast, more powerful core is running useful work as frequently as possible. They focus their work on a system more closely related to HammerBlade then the other related work mentioned, but the problem they are addressing is more concerned with efficient of utilization asymmetric compute resources rather than locality.

Nelson et al. [2015] implement a task-parallel software framework for a large distributed shared-memory machine using commodity hardware. They also have a notion of delegates and active messaging to assign small tasks on remote processors to improve data locality. They assume delegates are small messages that are forbidden from spawning more work. I relax this restriction in part because I use these delegates at to assign work at a coarser granularity and expect them to be used at a significantly lower frequency. The system that they focused on has mcuh greater latency implication impacts depending on locality than BigBlade. While BigBlade has extraordinarily high memory latency, the contribution to this from the cores' on-chip network locality is dwarfed by that of the off-chip memory latency which is uniform.

Chapter 6

Conclusion

This thesis explored the task-parallel programming model for the HammerBlade manycore. In Chapter 2, I performed a study of the performance and power characteristics of High Bandwidth Memory on an AMD Radeon VII. HBM was the memory system we ultimately planned to use for HammerBlade which, at the time that study was conducted, did not yet exist in silicon form. I provided an overview of HammerBlade in Chapter 3. I then introduced a library to support the task-parallel model in the form of fork-join parallelism in Chapter 4. This work was informative as it revealed that such a model could be implemented with low-overhead on HammerBlade aided by a work-stealing task scheduler. In Chapter 5, I was finally able to run applications for HammerBlade on the real silicon that the group taped-out in 2021. This final work revealed that applications using the dynamic task-parallel runtime can scale to full system size and that the overheads are small at large inputs relative to pure SPMD implementations. Additionally, by restricting the work-stealing schedulers to consider victims only within their pods, performance can improve by as much as 25%.

6.1 Future Research Directions

Work-Stealing with Asymmetric Computation: In this thesis, a worker is defined to be a single core. This does not need to be the case. A worker could be defined as a group of cores in close proximity on the network, each of which executes distinct regions of program text that collectively execute a task. For example, one core in the worker group could be in charge of scheduling, another in charge of memory allocation and

prefetching, and yet another in charge of executing the core application logic. Some cores might not execute at all, and allow others to claim use of their tile's scratchpad memory. One could imagine a few reasons why this might be a good idea. First, as I touched on in Section 5.4, HammerBlade's tile instruction-cache is built for area efficiency but presents serious performance challenges when program text outgrows its 4 KB capacity. Dividing the regions of program text between several different cores instead of having all of them execute the text uniformly mitigates instruction redundancy in the system and enables each core to specialize its tile's memory resources. Second, not all applications are bounded by compute resources. This is true regardless of the architecture, and Figure 5.11 shows that HammerBlade and BigBlade is no exception. Therefore, rather than allocating more cores towards fruitlessly executing memory-intensive application code, redirecting them towards other purposes such as reducing scheduling overheads or increasing on-chip memory utilization is more likely to improve performance and efficiency.

Opportunities for Compiler Support: A key concern in Chapter 5 is reliability stemming from limited stack space in scratchpad memory. The work in Chapter 4 found that placing the stack in scratchpad was often a boon to performance, and the challenging memory system on BigBlade makes that optimization as important as ever. At the same time, 4 KB of stack is very little, especially for a work-stealing runtime that relies on a recursive divide-and-conquer scheme such as the one studied in this thesis. Furthermore, stack memory allocated in the root task is utilized much less than that allocated in the leaves. Cactus stacks Yang and Mellor-Crummey [2016], which have been explored for work-stealing frameworks in the past, are a particularly promising direction for HammerBlade. The compiler could amend the ABI so that the memory location of individual stack frames can be selected on a procedure-by-procedure basis.

Chapter 5 also discusses HammerBlade's mechanism for extending the native 32-bit address space using control registers. I introduce software primitives in C++ for simplified, if not seamless, software adoption of this control register. A limitation of that approach is inefficiency. Because the compiler does not know that control register's semantics, it often redundantly sets it to the same value to which it was already programmed. Sometimes this is unavoidable, even if the compiler has semantic understanding, such as setting and restoring its value across procedure calls (although this too could be mitigated with callee-save conventions). Often times, however, it is avoidable. Empowering the compiler with some semantic understanding of the register would improve work-efficiency and also reduce instruction footprint.

Targeting Different Memory Systems: In Chapter 5, I conduct a brief study of BigBlade's memory performance to ground our understanding of system behavior. I showed that BigBlade's memory latency and bandwidth pose serious performance challenges to all but the most compute-intensive applications. This characteristic makes off-chip memory the most significant constraint when designing any software for BigBlade. A faster memory system would have a dramatic impact on software design, and that dynamic would certainly be worth exploring in future HammerBlade systems.

Evaluating Programmability of Parallel Models: At the time of this writing, those of us interested in researching programming models, programmability, and other software-engineering problems are due for reflection. The ubiquity of artificial intelligence has expanded rapidly in recent years, and the pace of this expansion will likely accelerate. When I started my graduate student career in 2018, my first grant called for a machine that could be programmed with Python. Python, rightly or not, was widely considered the gold standard with respect to "programmability" as far as programming languages were concerned. I use the word "was" here because, seven years later, industry leaders are suggesting that the future gold standard might not be any programming language at all, but rather natural language conveyed to an AI agent to write the software in our stead. The degree to which this prediction will come to pass remains uncertain. Nonetheless, it is worth pondering how we should define programmability in a future where machines, rather than humans, are write the code.

That aforementioned grant called for a "programmable" architecture. However, there was some mystery as to how such a goal should be evaluated. In many of our deliverables, we cited the number of lines of code as a proxy. This metric is imperfect for many reasons that are, hopefully, obvious. Initially, the grant sponsors attempted to evaluate programmability by enlisting a large team of undergraduates as test subjects. This approach ran into logistical problems and was, without a doubt, expensive to an extent that it would have been out of reach for most researchers to reproduce. In 2025, results for programmability using a suite of accepted AI models would be significantly easier to reproduce and also more accessible for researchers with fewer financial resources.

Bibliography

1993. *CRAY T3D System Architecture Overview*. Cray Research, Inc.

2011. *OpenCL Specification, v1.2*. Khronos Working Group.

2012. *Intel Cilk Plus Language Extension Specification*. Intel Corporation.

2013. *OpenMP Application Program Interface, Version 4.0*. OpenMP Architecture Review Board.

2015. Jedec standard jesd235a: High bandwidth memory (hbm) dram.

2018. Jedec standard jesd235b: High bandwidth memory (hbm) dram.

2019. *Intel Threading Building Blocks*. Intel Corporation.

2020. Jedec standard jesd235c: High bandwidth memory (hbm) dram.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, page 1–12, New York, NY, USA. Association for Computing Machinery.

Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael B. Taylor, and Zhiru Zhang. 2017a. Celerity: An open-source risc-v tiered accelerator fabric. *Symp. on High Performance Chips (Hot Chips)*.

Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Anuj Rao, Austin Rovinski, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Rajesh K. Gupta, Michael B. Taylor, and Zhiru Zhang. 2017b. Experiences using the risc-v ecosystem to design an accelerator-centric soc in tsmc 16nm. *Workshop on Computer Architecture Research with RISC-V (CARRV)*.

Lluc Alvarez, Miquel Moretó, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. 2015. Runtime-guided management of scratchpad memories in multicore architectures. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*.

AMD. Rdna architecture whitepaper. `https://www.amd.com/system/files/documents/rdna-whitepaper.pdf`. [Online].

AMD. 2019. Amd radeon vii graphics card. `https://www.amd.com/en/products/graphics/amd-radeon-vii`. [Online].

E. Anderson, J. Brooks, C. Grassl, and S. Scott. 1997. Performance of the cray t3e multiprocessor. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, pages 39–39.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. The landscape of parallel computing research: A view from berkeley.

Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The design of openmp tasks. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 20(3):404–418.

baidu-research. Deepbench. `https://github.com/baidu-research/DeepBench`. [Online].

R. Balasubramonian. 2019. *Innovations in the Memory System*, volume 14 of *Synthesis Lectures on Computer Architecture*.

Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, Dave Wentzlaff, Walker

Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. 2008. Tile64 processor: A 64-core soc with mesh interconnect. *Int'l Solid-State Circuits Conf. (ISSCC)*.

Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. 1996a. An analysis of dag-consistent distributed shared-memory algorithms. *Symp. on Parallel Algorithms and Architectures (SPAA)*.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996b. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69.

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748.

Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. 2017. KiloCore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902.

Ajay Brahmakshatriya, Emily Furst, Victor Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. 2021. Taming the zoo: The unified graphit compiler framework for novel architectures. *Int'l Symp. on Computer Architecture (ISCA)*.

S. Burke. Gamersnexus. `https://www.gamersnexus.net/hwreviews/3020-amd-rx-vega-56-review-undervoltage-hbm-vs-core`. [Online].

K. K. Chang, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu. 2017. Understanding reduced-voltage operation in modern dram devices: Experimental characterization, analysis, and mechanisms. In *SIGMETRICS*, New York, NY, USA.

Kevin K. Chang, Abdullah Giray Yaglikçi, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Onur Mutlu. 2018. Voltron: Understanding and exploiting the voltage-latency-reliability trade-offs in modern DRAM chips to improve energy efficiency. *CoRR*, abs/1805.03175.

P. Charles, C. Grothoff, V. Sarkar, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*.

N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. 2017. Architecting an energy-efficient dram system for gpus. In *HPCA*.

N. Chatterjee, M. O'Connor, G. Loh, N. Jayasena, and R. Balasubramonian. 2014. Managing dram latency divergence in irregular gpgpu applications. In *SC'14*, pages 128–139.

Tao Chen, Shreesha Srinath, Christopher Batten, and Edward Suh. 2018. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. *Int'l Symp. on Microarchitecture (MICRO)*.

Lin Cheng, Peitian Pan, Zhongyuan Zhao, Krithik Ranjan, Jack Weber, Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Preslav Ivanov, Dustin Richmond, Michael B. Taylor, Zhiru Zhang, and Christopher Batten. 2022. A tensor processing framework for cpu-manycore heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1620–1635.

cuda. 2013 (accessed Nov 2013). Cuda. Online Webpage. `http://www.nvidia.com/object/cuda_home_new.html`.

Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, page 157–166, New York, NY, USA. Association for Computing Machinery.

Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. 2012. CPU DB: Recording microprocessor history. *ACM Queue*, page 10–27.

H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. 2011. Memory power management via dynamic voltage/frequency scaling. In *ICAC*, New York, NY, USA.

Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald G. Dreslinski, Christopher Batten, and Michael B. Taylor. 2018. 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.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1).

T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. 2017. Evaluating attainable memory bandwidth of parallel programming models via babelstream. *IJCSE*, 17:247–262.

Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. 2012. Coscale: Coordinating cpu and memory system dvfs in server systems. In *MICRO*.

Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. 2011. Memscale: Active low-power modes for main memory. In *ASPLOS*, New York, NY, USA.

James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*.

B. Diniz, D. Guedes, W. Meira, and R. Bianchini. 2007. Limiting the power consumption of main memory. In *ISCA*, New York, NY, USA.

Eliovp. Amd memory tweak. `https://github.com/Eliovp/amdmemorytweak`. [Online].

S. Eyerman and L. Eeckhout. 2010. A counter architecture for online dvfs profitability estimation. *IEEE Transactions on Computers*, 59(11):1576–1583.

Naila Farooqui, Rajkishore Barik, Brian T. Lewis, Tatiana Shpeisman, and Karsten Schwan. 2016. Affinity-aware work-stealing for integrated cpu-gpu processors. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the cilk-5 multithreaded language. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

Yaosheng Fu and David Wentzlaff. 2015. Coherence domain restriction on large scale systems. *Int'l Symp. on Microarchitecture (MICRO)*.

Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Yi Guo, Jisheng Zhao, V. Cave, and V. Sarkar. 2010. Slaw: A scalable locality-aware adaptive work-stealing scheduler. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*.

Tom R. Halfhill. 2020. Thunderx3's cloudburst of threads: Marvell previews 96-core 384-thread arm server processor. *Microprocessor Report, The Linley Group*.

Hasan Hassan, Minesh Patel, Jeremie S. Kim, A. Giray Yaglikci, Nandita Vijaykumar, Nika Mansouri Ghiasi, Saugata Ghose, and Onur Mutlu. 2019. Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 129–142.

Hasan Hassan, Gennady Pekhimenko, Nandita Vijaykumar, Vivek Seshadri, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. 2016. Chargecache: Reducing dram latency by exploiting row access locality. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 581–593.

Henry Hoffmann, David Wentzlaff, and Anant Agarwal. 2010. Remote store programming. *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*.

Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. 2007. A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61.

Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada,

Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. 2010. A 48-core ia-32 message-passing processor with DVFS in 45nm CMOS. *Int'l Solid-State Circuits Conf. (ISSCC)*.

Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. 2020. Ruche networks: Wire-maximal, no-fuss nocs : Special session paper. *Int'l Symp. on Networks-on-Chip (NOCS)*.

Dai Cheol Jung, Max Ruttenberg, Paul Gao, Scott Davidson, Daniel Petrisko, Kangli Li, Aditya K Kamath, Lin Cheng, Shaolin Xie, Peitian Pan, Zhongyuan Zhao, Zichao Yue, Bandhav Veluri, Sripathi Muralitharan, Adrian Sampson, Andrew Lumsdaine, Zhiru Zhang, Christopher Batten, Mark Oskin, Dustin Richmond, and Michael Bedford Taylor. 2024. Scalable, programmable and dense: The hammerblade open-source risc-v manycore. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 770–784.

Kalray. 2022 (accessed Aug 2022). Kalray mppa products. Online Webpage. `https://www.kalrayinc.com/products/mppa-technology/`.

S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. 2015. Profiling a warehouse-scale computer. In *ISCA*, New York, NY, USA.

David Kanter. 2015. Knights Landing reshapes HPC.

Mohamed Lamine Karaoui, Pierre-Yves Péneau, Quentin L. Meunier, Franck Wajsbürt, and Alain Greiner. 2016. Exploiting large memory using 32-bit energy-efficient manycore architectures. *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 61–68.

I. Karlin, J. Keasler, and R. Neely. 2013. Lulesh 2.0 updates and changes. Technical report, Livermore.

S. Kaxiras and M. Martonosi. 2008. *Computer Architecture Techniques for Power-Efficiency*, volume 3 of *Synthesis Lectures on Computer Architecture*.

B. Keeth, R. J. Baker, B. Johnson, and F. Lin. 2007. *DRAM Circuit Design: Fundamental and High-Speed Topics*, 2nd edition. Wiley-IEEE Press.

John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. 2009. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. *Int'l Symp. on Computer Architecture (ISCA)*.

Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2010. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, page 287–296, New York, NY, USA. Association for Computing Machinery.

D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. K.-W. Chang, and O. Mutlu. 2015. Adaptive-latency dram: Optimizing dram timing for the common-case. In *HPCA*, Burlingame, CA.

D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. 2013. Tiered-latency dram: A low latency and low cost dram architecture. In *HPCA*.

Charles E. Leiserson. 2009. The cilk++ concurrency platform. *Design Automation Conf. (DAC)*.

L. Li, J. Fang, H. Fu, J. Jiang, W. Zhao, C. He, X. You, and G. Yang. 2018a. swcaffe: A parallel framework for accelerating deep learning applications on sunway taihulight. *Int'l Conf. on Cluster Computing*.

S. Li, D. Reddy, and B. Jacob. 2018b. A performance & power comparison of modern high-speed dram architectures. In *MEMSYS*, New York, NY, USA.

S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob. 2020. DRAMsim3: A cycle-accurate, thermal-capable dram simulator. *Computer Architecture Letters (CAL)*.

Zheng Li, Jose Duato, Olivier Certner, and Olivier Temam. 2010. Scalable hardware support for conditional parallelization. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*.

Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Ilia Lebedev, and Srinivas Devadas. 2013. Hardware-level thread migration in a 110-core shared-memory multiprocessor. Technical Report 512, MIT CSAIL CSG.

Z. Liu. Tom's hardware. `https://www.tomshardware.com/news/amd-memory-tweak-tool,39407.html`. [Online].

Guo-Ping Long, Jun-Chao Zhang, and Dong-Rui Fan. 2008. Architectural support and evaluation of cilk language on many-core architectures. *Chinese Journal of Computers*, 31(11):1975–1985.

Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. Tapas: Generating parallel accelerators from parallel programs. *Int'l Symp. on Microarchitecture (MICRO)*.

Michael McCool, Arch D. Robinson, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.

Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Jonathan Balkind, Alexey Lavrov, Mohammad Shahrad, Samuel Payne, and David Wentzlaff. 2017. Piton: A manycore processor for multitenant clouds. *IEEE Micro*, 37(2):70–80.

R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt. 2012. Predicting performance impact of dvfs for realistic memory systems. *IEEE Micro*, pages 155–165.

Seung-Jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, volume 625.

mpi. 2013 (accessed Nov 2013). Message passing interface (mpi) standard. Online Webpage. `http://www.mcs.anl.gov/research/projects/mpi/standard.html`.

Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 291–305, USA. USENIX Association.

NVIDIA. 2016. Nvidia tesla p100. `https://www.nvidia.com/en-us/data-center/tesla-p100/`. [Online].

NVIDIA. 2017. Nvidia titan v. `https://www.nvidia.com/en-us/titan/titan-v/`. [Online].

M. O'Connor, N. Chatterjee, D. Lee, J. M. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally. 2017. Fine-grained dram: energy-efficient dram for extreme bandwidth systems. *IEEE Micro*, pages 41–54.

Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2006. Uts: An unbalanced tree search benchmark. *Int'l Workshop on Lanaguages and Compilers for Parallel Computing (LCPC)*.

Andreas Olofsson. 2016. Epiphany-V: A 1024-processor 64-bit RISC system-on-chip. *Computing Research Repository (CoRR)*, arXiv:abs/1610.01832.

Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-grain task aggregation and coordination on gpus. *Int'l Symp. on Computer Architecture (ISCA)*.

Yanghui Ou, Shady Agwa, and Christopher Batten. 2020. Implementing low-diameter on-chip networks for manycore processors using a tiled physical design methodology. *Int'l Symp. on Networks-on-Chip (NOCS)*.

Guilherme P. Pezzi, Marcia C. Cera, Elton Mathias, Nicolas Maillard, and Philippe O. A. Navaux. 2007. Online scheduling of mpi-2 programs with hierarchical work stealing. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*.

J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. 2013. Heterogeneous system coherence for integrated cpu-gpu systems. *Int'l Symp. on Microarchitecture (MICRO)*.

Carl Ramey. 2011. TILE-Gx100 manycore processor: Acceleration interfaces and architecture. *Symp. on High Performance Chips (Hot Chips)*.

James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly.

D. Richards, Y. Alexeev, X. Andrade, R. Balakrishnan, H. Finkel, G. Fletcher, C. Ibrahim, W. Jiang, C. Junghans, J. Logan, A. Lund, D. Lykov, R. Pavel, and V. Ramakrishnaiah. 2020. Fy20 proxy app suite release. Technical report, Livermore.

ROCMm-Developer-Tools. rocprofiler. `https://github.com/ROCm-Developer-Tools/rocprofiler`. [Online].

Barry Rountree, David K. Lowenthal, Martin Schulz, and Bronis R. de Supinski. 2011. Practical performance prediction under dynamic voltage frequency scaling. In *2011 International Green Computing Conference and Workshops*, pages 1–8.

Austin Rovinski, Chun Zhao, Khalid Al-Hawaj, Paul Gao, Shaolin Xie, Christopher Torng, Scott Davidson, Aporva Amarnath, Luis Vega, Bandhav Veluri, Anuj Rao, Tutu Ajayi, Julian Puscar, Steve Dai, Ritchie Zhao, Dustin Richmond, Zhiru Zhang, Ian Galton, Christopher Batten, Michael B. Taylor, and Ron G. Dreslinski. 2019a. 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)*.

Austin Rovinski, Chun Zhao, Khalid Al-Hawaj, Paul Gao, Shaolin Xie, Christopher Torng, Scott Davidson, Aporva Amarnath, Luis Vega, Bandhav Veluri, Anuj Rao, Tutu Ajayi, Julian Puscar, Steve Dai, Ritchie Zhao, Dustin Richmond, Zhiru Zhang, Ian Galton, Christopher Batten, Michael B. Taylor, and Ron G. Dreslinski. 2019b. 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.

Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. 2011. Lifeline-based global load balancing. *SIGPLAN Not.*, page 201–212.

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding fork-join parallelism into llvm's interemdiate representation. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*.

O. Seongil, Young Hoon Son, Nam Sung Kim, and Jung Ho Ahn. 2014. Row-buffer decoupling: A case for low-latency dram microarchitecture. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 337–348.

Shumpei Shiina and Kenjiro Taura. 2019. Almost deterministic work stealing. In *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16.

Wongyu Shin, Jeongmin Yang, Jungwhan Choi, and Lee-Sup Kim. 2014. Nuat: A non-uniform access time memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 464–475.

Julian Shun and Guy Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*.

I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for gpu architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*.

M. Själander, M. Martonosi, and S. Kaxiras. 2014. *Power-Efficient Computer Architectures: Recent Advances*, volume 9 of *Synthesis Lectures on Computer Architecture*.

B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang. 2014. Ppep: Online performance, power, and energy prediction framework and dvfs space exploration. *IEEE Micro*, pages 445–457.

Giuseppe Tagliavini, Daniele Cesarini, and Andrea Marongiu. 2018. Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight openmp tasking. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2150–2163.

Guangming Tan, Dongrui Fan, Junchao Zhang, Andrew Russo, and Guang R. Gao. 2008. Experience on optimizing irregular computation for memory hierarchy in manycore architecture. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*.

Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. 2003. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. *Int'l Solid-State Circuits Conf. (ISSCC)*.

Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. 2004. Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. *Int'l Symp. on Computer Architecture (ISCA)*.

Christopher Torng, Moyang Wang, and Christopher Batten. 2016. Asymmetry-aware work-stealing schedulers. *Int'l Symp. on Computer Architecture (ISCA)*.

J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. 2014. Xsbench - the development and verification of a performance abstraction for monte carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally. 2014. Scaling the power wall: A path to exascale. In *SC'14*.

Pascal Vivet, Eric Guthmuller, Yvain Thonnart, Gael Pillonnet, Guillaume Moritz, Ivan Miro-Panadès, Cesar Fuguet, Jean Durupt, Christian Bernard, Didier Varreau, Julian Pontes, Sebastien Thuries, David Coriat, Michel Harrand, Denis Dutoit, Didier Lattard, Lucile Arnaud, Jean Charbonnier, Perceval Coudrain, Arnaud Garnier, Frederic Berger, Alain Gueugnot, Alain Greiner, Quentin Meunier, Alexis Farcy, Alexandre Arriordaz, Severine Cheramy, and Fabien Clermidy. 2020. 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. *Int'l Solid-State Circuits Conf. (ISSCC)*.

Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten. 2020. Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems. *Int'l Symp. on Computer Architecture (ISCA)*.

Yaohua Wang, Arash Tavakkol, Lois Orosa, Saugata Ghose, Nika Mansouri Ghiasi, Minesh Patel, Jeremie S. Kim, Hasan Hassan, Mohammad Sadrosadati, and Onur Mutlu. 2018. Reducing dram latency via charge-level-aware look-ahead partial restoration. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 298–311.

David Wentzlaff, Patrick Griffin, Henry Hoffman, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31.

Bob Wheeler. 2020. Ampere maxes out at 128 cores. *Microprocessor Report, The Linley Group*.

W. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall. *ACM SIGARCH Computer Architecture News*.

Chaoran Yang and John Mellor-Crummey. 2016. A practical solution to the cactus stack problem. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 61–70, New York, NY, USA. Association for Computing Machinery.

Foivos S. Zakkak and Polyvios Pratikakis. 2016. 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)*.

Florian Zaruba, Fabian Schuiki, and Luca Benini. 2021. Manticore: A 4096-core RISC-V chiplet architecture for ultraefficient floating-point computing. *IEEE Micro*.