SOFTWARE/HARDWARE CO-DESIGN TO IMPROVE PRODUCTIVITY, PORTABILITY, AND PERFORMANCE OF LOOP-TASK PARALLEL APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ji Yun Kim

January 2017

SOFTWARE/HARDWARE CO-DESIGN TO IMPROVE
PRODUCTIVITY, PORTABILITY, AND PERFORMANCE OF LOOP-TASK PARALLEL APPLICATIONS

Ji Yun Kim, Ph.D.
Cornell University 2017

Computer architects are increasingly turning to programmable accelerators tailored for narrower classes of applications in order to achieve high performance and energy efficiency. A continuing challenge with accelerators is enabling the programmer to easily extract maximum performance without intimate knowledge of the underlying microarchitecture. It is important to consider productivity and portability, in addition to performance, as first-class metrics when developing and evaluating modern computing platforms. Software-centric approaches to achieving 3P computing platforms are compelling, but sacrifice efficiency and flexibility by hiding parallel abstractions from hardware and limiting the scope of the application domain. This thesis proposes a new software/hardware co-design approach to achieving 3P platforms, called the loop-task accelerator (LTA) platform, that provides high productivity and portability without sacrificing performance or efficiency across a wide range of applications. The LTA platform addresses the weaknesses of existing approaches that are identified through detailed experimentation with and analysis of modern application development. Discussion of an early attempt at a hardware-centric approach to achieving 3P platforms provides insight into area-efficient accelerator designs and highlights the need for innovations in both software and hardware. The LTA platform focuses on exploiting loop-task parallelism by exposing *loop-tasks* as a common parallel abstraction at the programming API, runtime, ISA, and microarchitectural levels. The LTA programming API uses the `parallel_for` construct to express loop-tasks that can be exploited both across cores and within a core, the LTA runtime distributes loop-tasks across cores, and a new `xpfor` instruction explicitly encodes loop-tasks as functions applied to a range of loop iterations. This thesis introduces a novel task-coupling taxonomy that captures how tasks can be coupled in both space and time. The LTA engine template can be configured at design time with variable spatial and temporal task coupling to accelerate the execution of both regular and irregular loop-tasks within a core. The LTA platform is evaluated with

respect to the 3P's using a vertically integrated research methodology. Compared to an in-order multi-core baseline, the LTA platform yields average improvements of 5.5× in raw performance, 2.5× in performance per area, and 1.2× in energy efficiency, while offering high productivity and portability.

BIOGRAPHICAL SKETCH

Ji Yun Kim was born on July 3, 1987 in Seoul, South Korea. He is the eldest son of Dae Kun Kim and Kang In Kim, and has one younger brother, Chang Yun Kim. When he was 6 years old, Ji and his family immigrated to the United States to pursue better education and a brighter future. As a youth, Ji became fascinated with the electronics that his father would bring back from his business trips in Japan. This sparked his interest in electrical and computer engineering and he began learning how to program and building personal computers. Ji attended the University of Pennsylvania for his undergraduate studies where he majored in electrical engineering. Although his background was in digital circuits and his initial intent was to continue his graduate studies researching radios and fiber optic communication systems, two serendipitous events pushed him towards computer architecture. First, a senior-level class in computer organization broadened his perspective on how the circuits he was designing could be utilized in a complex system. Second, his capstone project (enhancing the resiliency of VoIP) was heavily focused on software, which sparked an interest in application development. These events in combination fostered a deeper interest in the software/hardware interface of computer architecture. At Cornell University, Ji began his PhD career with his advisor, Professor Christopher Batten. Under Professor Batten's tutelage, Ji learned the fundamentals of computer architecture from the ground up, eventually gaining invaluable experience in application development, software runtimes, compilers, as well as microarchitecture and ASIC design.

In his free time Ji enjoys board games, reading, cooking, guitar, biking, and experiencing different cultures abroad. His favorite authors include Haruki Murakami, Kurt Vonnegut, and Paul Auster. His favorite drinks include single-malt whiskey and beers as dark as the night. One day Ji hopes to write a novel of his own and retire to a wood cabin in the wilderness of Maine where he hopes to spend the rest of his days building high-end wooden furniture. Ji is thankful for his PhD experience–through the successes and the failures, the joys and the pains, sometimes it takes seeing your best and your worst to know where to go next.

This document is dedicated to my family, the source of inspiration and motivation in my life.

ACKNOWLEDGEMENTS

It goes without saying that a constant source of inspiration and motivation in my PhD career was my graduate committee consisting of three of the most brilliant people I have ever met: Prof. Christopher Batten, Prof. Jose Martinez, and Prof. Rajit Manohar. This thesis would not be what it is without the invaluable feedback and guidance from these advisors. I would especially like to thank my mentor, Christopher Batten, who not only gave me an opportunity to mature as a researcher, but as a person; for pushing me to be ceaseless in the pursuit of truly great research, and for helping me get through some of the darkest times in my life. One day I hope to be half the researcher he is.

The work in this thesis was built on the foundations laid by my amazing colleagues in the Batten Research Group and I would like to extend a sincere thanks to students both past and present. The technical contributions of each member in the group along with the detailed acknowledgements of other technical collaborators (e.g., Yunsup Lee, Martin Burtscher, Rupesh Nasre, Sripathi Pai, David Bindel) are in Section 1.3, so I would like to use this section for personal thanks. Derek Lockhart guided me as a more experienced graduate student in my early years and I'm thankful for the times that we shared. Shreesha Srinath is a dear friend who has been with me through thick and thin–I would not be here without his council and advice. His passion for research and breadth of knowledge never ceases to impress me. Christopher Torng is a fantastic researcher and is surely destined for success in academia. It was because of Chris I pushed myself to be a better mentor. Berkin Ilbeyi is one of the most gentle and polite human beings I have the pleasure of knowing (he is also a Python wizard). Moyang Wang is a dedicated and outstanding colleague who is already better at C++ than I am at the end of my PhD career. The two newest additions to BRG, Shunning Jiang and Khalid Al-Hawaj, are enthusiastic and talented individuals that make me excited to see the future of BRG. I would also like to acknowledge the tremendous work of the fantastic MEng students I have worked with: Scott McKenzie, Alvin Wijaya, Jason Setter, and Wei Geng.

Aside from my own research group, I would like to thank my friends in the Computer Systems Laboratory at Cornell. KK Yu has been a cherished mentor in my life since I met him, academically and spiritually. KK was always there to listen to me complain, and without him there to give me perspective I would probably have lost my way. Daniel Lo is another great friend who is a joy to be around. His hundreds of hobbies and interests reflect his joy for life and gives me something to strive toward. Ritchie Zhao is my one and only true heir to the throne and, especially in my later

years, was my greatest source of companionship. I will treasure my memories with him and my only regret is not being able to spend more time together. There are many more important people that I will not forget: Rob Karmazin, Saugata Ghose, Jon Tse, Kyle Wecker, Maya Kelner, Janani Mukundan, and the list goes on. A special acknowledgement to the Ward 101 crew of Charles Jeon and Ryan Lee, it was fun while it lasted and thanks for being there during the worst of times.

I would further like to extend my thanks to my undergraduate research advisors, Prof. Jan Van der Spiegel and Prof. Andre DeHon at the University of Pennsylvania, who were willing to take in a inexperienced student into their groups. Without the time I spent in that research environment, I would not have decided to pursue a PhD. In addition, I would like to thank my friend and confidant, Jefferson Wen, who I have known since freshmen year at the University of Pennsylvania, for his support and unwaivering kindness over these many years.

Last but absolutely not least, I would like to thank my family for being loving, supportive, and patient. From a young age, my parents, Dae and Kang Kim, instilled in me the importance of education and pursuing my passion to help others. I hope that what I have learned and experienced during my PhD will help me to succeed in touching people's lives no matter where I end up. My younger brother and best friend, Chang Kim, inspired me to believe in the best of myself and always manages to cheer me up when I need it. Finally, I would like to extend a special thanks to my grandfather, Hui Sok Kim, who has been my role model since I was a child and supported me in uncountable ways my entire life. His guidance and his belief in me has shaped me into the man I am today.

In terms of funding, this thesis was supported in part by an NDSEG Fellowship, NSF CAREER Award #1149464, NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, AFOSR YIP Award #FA9550-15-1-0194, and donations from Intel Corporation, NVIDIA Corporation, and Synopsys, Inc.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CMP	chip multiprocessor
MIC	many integrated core
GPGPU	general-purpose graphics processing unit
SIMD	single-instruction multiple-data
SIMT	single-instruction multiple-thread
RISC	reduced instruction set computer
GPP	general-purpose processor
LTA	loop-task accelerator
TBB	(Intel) threading building blocks
AVX	(Intel) advanced vector extensions
CUDA	(NVIDIA) compute unified device architecture
API	application programming interface
DSL	domain-specific language
IR	intermediate representation
ISA	instruction set architecture
CAD	computer aided design
RTL	register-transfer level
VLSI	very-large-scale integration
ASIC	application-specific integrated circuit
TMU	task management unit
IMU	instruction management unit
PIB	pending instruction buffer
DMU	data management unit
PDB	pending data buffer
FU	fetch unit
PC	program counter
PFB	pending fragment buffer
DU	decode/dispatch unit
RT	rename table
IU	issue unit
IQ	issue queue
RF	register file
SLFU	short-latency functional unit
LLFU	long-latency functional unit
LSU	load/store unit
WCU	writeback/commit unit
WQ	writeback queue
SRAM	static random access memory
DRAM	dynamic random access memory
I$	instruction cache
D$	data cache

CHAPTER 1
INTRODUCTION

With Moore's Law approaching its end [Dub05], architects are increasingly turning to programmable accelerators that can achieve high performance and energy efficiency by specializing for narrower classes of applications. Examples include mainstays like traditional vector processors [EVS98, EV96, Oya99] and packed single-instruction multiple-data (SIMD) engines [Hug15, SS00], but more recently include Intel's many integrated core architectures [Kan16] (MIC), NVIDIA/AMD's general-purpose graphics processing units [nvi16, amd12a] (GPGPU), Movidius' image/vision accelerators [Dem14], and Qualcomm's mobile-focused accelerators [Gwe14]. Unfortunately, the continuing challenge is how much performance and energy efficiency the programmer can *actually* extract from the hardware. Extracting the highest levels of performance and energy efficiency often requires a significant amount of optimizations based on a deep understanding of the underlying microarchitecture [AJ88, DKH11, JR13, LZH+13, BNP12, HKOO11] and optimized implementations of the same application can be vastly different depending on the target architecture [HVS+13, LZH+13, LWH10, MLBP12]. Furthermore, current accelerators primarily address applications with very regular control-flow and memory-access patterns. In this emerging era of computer architecture, it is clear that performance cannot be the sole metric for developing and evaluating computing platforms composed of software frameworks and hardware architectures.

1.1 The 3P's: Productivity, Portability, and Performance

The challenges described above suggest that computer architects should focus on "the 3P's" of **productivity, portability, and performance** as the primary metrics in developing and evaluating computing platforms. In fact, recent trends indicate that architects are already placing more value on productivity and portability.

The demand for productivity is evidenced by the increasing popularity of task-based parallel programming frameworks such as Intel's C++ Threading Building Blocks (TBB) [Rei07, int15], Intel's Cilk Plus [Lei09, int13], Microsoft's .NET Task Parallel Library [LSB09, CJMT10], Java's Fork/Join Framework [Lea00, jav15], and OpenMP [ACD+09, ope13]. This is in contrast to more traditional thread-based frameworks like POSIX threads, C++11 threads, and MPI [mpi13]. Thread-based programming models require explicit thread management and static work distribu-

tion with no dynamic load balancing. On the other hand, task-based programming models implicitly manage an optimal number of threads for the system in a *thread pool* and dynamically schedule fine-grain tasks across any available threads. This frees the programmer to think at a higher level of abstraction and leaves the lower-level details to a software runtime.

Although tasks can express a wide range of parallel programming patterns, loops remain one of the most popular targets of task-based parallelization. This thesis focuses on a subset of task parallelism called *loop-task parallelism* that is captured by the ubiquitous `parallel_for` construct. Loop-task parallelism can be seen as a more general form of data parallelism, where the operator applied to a range of loop iterations can have enough variability due to data-dependent conditionals that loop iterations are closer to exhibiting task parallelism. Loop-task parallelism can be regular or irregular depending on whether there are structured control-flow and memory-access patterns across loop iterations. Note that data parallelism as exploited by traditional vector processors generally refers to *regular* data parallelism; loop-task parallelism captures regular data parallelism as well as more irregular *task* parallelism across loop iterations. For the sake of brevity, applications with regular or irregular loop-task parallelism may be referred to as regular or irregular applications throughout this thesis.

The need for portability is indicated by the profusion of research on efficiently mapping both regular and irregular applications to accelerators like Intel's MICs [PHSJ13, HVS+13, LZH+13, PBV+13] and NVIDIA/AMD's GPGPUs [NBP13b, BNP12, HKOO11, LWH10]. As this trend continues, it would be desirable to have seamless portability between a more conventional chip multiprocessor (CMP) implementation and MIC/GPGPU implementations. This is especially true for applications with high performance targets since it is not always possible to predict which architecture would yield the best performance and with what cost in productivity [LKC+10].

Although software-centric approaches to achieving 3P platforms exist, they are not without their own tradeoffs. Virtual ISAs, like in AMD's heterogeneous system architecture [amd12b] (HSA), leverage software frameworks like OpenCL [ope11] and C++AMP [Som11] as a common framework that are lowered to an intermediate representation that can be further lowered to multiple hardware architectures. Domain-specific languages (DSL), like Halide [RKBA+13] and DeLite [BSL+11], focus on expressing highly specialized abstractions for a narrow application domain that the compiler can map to multiple hardware architectures. Unfortunately, in their current incarnations, both of these approaches fall short of their proposed productivity and portability

goals. For example, a non-trivial amount of architecture-specific optimizations are still required to yield the highest performance, and optimized implementations of the same application differ substantially depending on the target architecture. In addition, the parallel abstractions used by software-centric approaches are never directly exposed to the hardware (i.e., only pushed down to the compiler level), meaning much of the information about available parallelism is never explicitly communicated to the hardware. If a hardware accelerator were to be fundamentally designed to understand the same abstractions in software, there would be more opportunities to efficiently exploit the available parallelism. It is also important to note that both of these approaches do not support fine-grain inter-core load balancing.

1.2 Thesis Overview

This thesis proposes the *loop-task accelerator* (LTA) platform, a software/hardware co-design approach for a 3P platform that seeks to maintain high productivity and high portability without sacrificing performance or efficiency across a wide range of applications. The LTA platform efficiently exploits loop-task parallelism by exposing *loop-tasks* as the common parallel abstraction at the programming API, runtime, ISA, and microarchitectural levels. Figure 1.1 illustrates the overall vision for the LTA platform. Loop-tasks can be exploited *across* cores in software via a task-based work-stealing runtime and *within* a core in hardware via a special accelerator, called an LTA engine, attached to a general-purpose processor (GPP). LTA engines can be configured at design time to be tailored for diverse loop-task parallelism and can be homogeneous or heterogeneous.

Chapter 2 details my personal experience developing, porting, and optimizing application kernels with diverse loop-task parallelism to several platforms consisting of different software frameworks (e.g., TBB, CUDA) and hardware architectures (e.g., CMP, MIC, GPGPU). My findings suggest that no single platform excels at the 3P's for both regular and irregular loop-task parallel applications. One of the key observations from this study is that a software/hardware co-design approach that innovates across the computing stack may offer the best chance at achieving a true 3P platform.

Chapter 3 describes an early attempt at a hardware-centric approach to achieving a 3P platform called *fine-grain SIMT* (FG-SIMT). FG-SIMT is a SIMT-like programmable accelerator that is

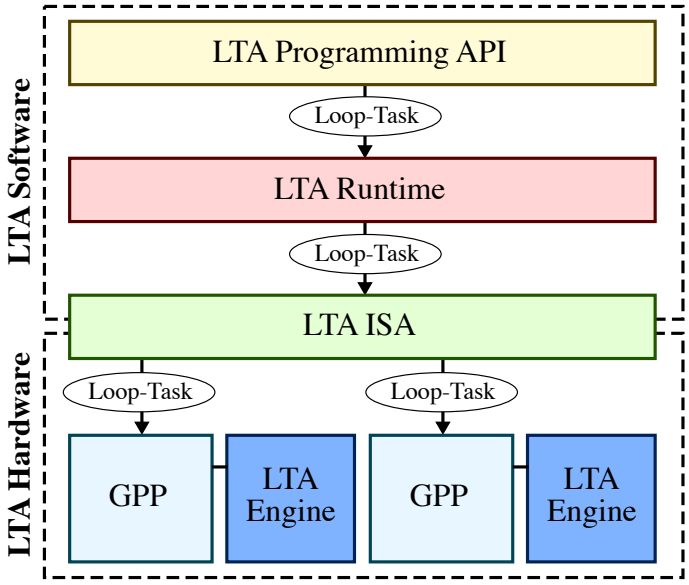


Figure 1.1: Vision for Loop-Task Accelerator (LTA) Platform – The goal of the LTA platform is to efficiently exploit loop-task parallelism both across cores and within a core by exposing *loop-tasks* as the common abstraction throughout the computing stack. The LTA programming API exposes loop tasks using the ubiquitous `parallel_for` construct. The LTA runtime generates, partitions, and distributes loop tasks across cores. The LTA ISA is used to explicitly encode loop tasks. The LTA engines accelerate regular or irregular loop tasks within a core.

designed for fine-grain integration into a MIC-like architecture. FG-SIMT addresses productivity by using threads as a common parallel abstraction (similar to CUDA) which are grouped into basic scheduling units called *warps*, addresses portability by using a single ISA on both the GPP and the accelerator, and addresses performance by focusing on exploiting intra-warp parallelism in lieu of inter-warp parallelism like in GPGPUs as well as enabling fine-grain work offloading. Although FG-SIMT is able to achieve impressive performance and energy efficiency on regular applications, it struggles to do so on more irregular applications. In addition, FG-SIMT is only examined in a single-core context and does not provide any major innovations in software. The conclusions from the FG-SIMT study further serve to motivate the need for innovation in both software and hardware, and inspires aspects of the LTA microarchitecture.

Chapters 4 and 5 detail the software and hardware components of the proposed LTA platform that seeks to achieve the 3P's for both regular and irregular loop-task parallel applications. The key idea is to narrow the scope of the software, broaden the scope of the hardware, and marry these layers by exposing loop-tasks as a common parallel abstraction across the computing stack. The LTA programming API uses the familiar `parallel_for` construct to express loop-tasks that can be exploited *both* across cores and within a core. The LTA task-based work-stealing runtime, inspired by TBB, distributes tasks across cores in software. The LTA platform extends a RISC-V-like ISA with a new `xpfor` instruction that explicitly encodes loop-tasks as a function applied over a range of loop iterations. This instruction can be executed on either traditional GPPs or on LTA engines that are designed to accelerate loop-task execution. I introduce a task-coupling taxonomy that

elegantly captures the spectrum of spatial and temporal task coupling within an LTA engine, and describe the impacts of task coupling on performance, area, and energy efficiency when executing regular and irregular loop-task parallel applications. Furthermore, the LTA engine template can be configured at design time with different levels of spatial and temporal task coupling, enabling a deep design-space exploration of this taxonomy.

Chapter 6 outlines the vertically integrated research methodology used to evaluate the LTA platform. My colleague and I have ported 16 C++ application kernels which include a diverse mix of regular (e.g., image/matrix processing), irregular (e.g., string processing, graph analysis), and mixed (e.g., sorting) loop-task parallelism. The LTA runtime is validated against other popular task-based work-stealing runtimes. For performance analysis, I use the gem5 cycle-level simulator [BBB+11] and PyMTL, a Python-based hardware modeling framework [LZB14]. For area and energy analysis, I use component/event-based modeling based on VLSI results of similar accelerator designs [SIT+14, KLST13] obtained using a commercial ASIC CAD toolflow with a TSMC 40nm standard cell library.

Chapter 7 presents the cycle-level performance, area-normalized performance, and energy efficiency of the LTA platform compared to single-core and multi-core baseline architectures. The results indicate that a moderate amount of both spatial and temporal task coupling offers the best performance across the widest range of applications, and the resulting multi-core LTA platform yields average improvements of 5.5× in raw performance, 2.5× in performance per area, and 1.2× in energy efficiency compared to a in-order multi-core baseline. Compared to a more aggressive out-of-order multi-core baseline, the LTA platform is able to achieve 3.0× improvement in raw performance, 1.7× in performance per area, and 2.5× in energy efficiency. Both the baseline and LTA platforms use the same LTA programming model and runtime. Productivity and portability are improved since a single implementation of any given application can take full advantage of the LTA engines on systems with (or without) any combination of LTA engines.

Chapter 8 catalogues the large body of related work and crystallizes the novelty of the LTA platform. Chapter 9 summarizes the contributions of this thesis and highlights interesting directions for extending this thesis.

The primary contributions of this thesis are condensed below:

- Detailed analysis of the **3P challenges in modern application development flows** based on years of computer architecture experience.

- The **FG-SIMT architecture**, an area-efficient accelerator for regular loop-task parallelism with microarchitectural mechanisms for exploiting value structure, and a detailed evaluation of this architecture with respect to performance, area, and energy based on RTL and gate-level models.
- Software components of the LTA platform including a productive TBB-like **LTA programming API**, a task-based work-stealing **LTA-aware runtime** and lightweight **LTA ISA** extensions including a new `xpfor` instruction that explicitly encodes loop-tasks as a common parallel abstraction.
- A **task-coupling taxonomy** that describes the spectrum of how tasks can be coupled in space and time, with appropriate terminology.
- Hardware components of the LTA platform including an elegant microarchitectural template for the **LTA engine** that can be configured at design time with variable spatial and temporal task coupling to target diverse loop-task parallelism.
- Deep design space exploration of the impacts of task coupling in the LTA engine on performance, area, and energy, as well as a 3P evaluation of the LTA platform using a vertically integrated research methodology.

1.3 Collaboration, Previous Publications, and Funding

This thesis would not have been possible without the immense effort and dedication of all of the members of the Batten Research Group. My advisor Christopher Batten was the key collaborator in iteratively evolving the concepts behind the LTA platform, such as exploring task coupling in *time* as well as space, and helping to refine the narrative to how it is presented in this thesis. Derek Lockhart is the lead developer of PyMTL, the hardware modeling framework used to model the LTA engines, and was integral in establishing the ASIC CAD toolflow used to generate VLSI results for area/energy modeling. Shreesha Srinath developed the gem5-PyMTL co-simulation framework which allowed me to leverage more sophisticated memory system and network models, and ported many application kernels to the LTA platform. Berkin Ilbeyi significantly improved the performance of the aforementioned co-simulator, making the experiments in this thesis viable, and also helped in the development of a couple key components of the LTA engine model. Christopher

Torng and Moyang Wang developed the task-based work-stealing runtime that eventually became the LTA runtime. Chris also authored the event-based energy modeling used in this thesis and was a tremendous resource for VLSI knowledge. Moyang also validated the LTA runtime and helped conduct native experiments. Khalid Al-Hawaj and Shunning Jiang characterized and ported many application kernels used in this thesis. I would also like to acknowledge Scott McKenzie and Alvin Wijaya for their fantastic work and insights on experimenting with early versions of loose spatial task-coupling, as well as Jason Setter and Wei Geng for their equally impressive work on modeling a new scalar processor based on lessons learned from developing the LTA engine.

Collaborators from other research groups are also responsible for making this thesis possible. Yunsup Lee of SiFive was a great resource during the development of early fine-grain SIMT models, especially with establishing the ASIC CAD toolflow and providing me with reference RTL. The entire LonestarGPU benchmark suite team, especially Prof. Martin Burtscher, Prof. Rupesh Nasre, and Sripathi Pai, generously provided additional code and insights into mapping irregular applications to GPGPUs. In addition, Prof. David Bindel at Cornell University graciously granted me access to several Intel Xeon Phi 5110P co-processors which were used in the experiments described in Chapter 2.

Two previous publications have contributed significantly to this thesis. My work on fine-grain SIMT [KLST13] (FG-SIMT) processors was the basis for the lane groups within the LTA engine. The LTA engine implementations of chime sequencing, vector bypassing, density-time, and two-stack reconvergence, amongst many other features, are based off of the RTL model I wrote for FG-SIMT. This RTL model is also the basis for the VLSI results used to build the area/energy modeling in this thesis. My work on integrating hardware worklists [KB14] (HWWL) into GPGPUs helped me to realize the importance of the 3P's as well as pushing software abstractions down to the physical ISA layer. Results from the HWWL paper also revealed that work is rarely transferred across cores (compared to work generated within a core), which was part of the motivation for using a software inter-core work distribution mechanism in the LTA platform.

In terms of funding, this thesis was supported in part by an NDSEG Fellowship, NSF CAREER Award #1149464, NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, AFOSR YIP Award #FA9550-15-1-0194, and donations from Intel Corporation, NVIDIA Corporation, and Synopsys, Inc.

CHAPTER 2
SOFTWARE/HARDWARE PLATFORMS FOR EXPLOITING LOOP-TASK PARALLELISM

This chapter provides a detailed account of developing, porting, and optimizing seven application kernels with diverse loop-task parallelism across several existing platforms. These platforms are qualitatively evaluated with respect to productivity and portability, and quantitatively evaluated with respect to performance. The goal here is not to exhaustively survey all existing platforms, but rather to identify generalizable weaknesses that prevent existing platforms from achieving all 3P's, thereby motivating the need for a new approach for a true 3P platform.

Figure 2.1 illustrates an example development flow for high-performance applications using platforms with various software frameworks and hardware architectures. Specifically, this study examines AVX on CMPs, TBB on CMPs, TBB+AVX on CMPs, TBB/OpenMP+AVX on MICs, and CUDA on GPGPUs. In general, application development begins with a single-threaded scalar implementation on the CMP to verify functionality. At this point, the programmer can extract more performance on a CMP by vectorizing the code on a single thread using packed-SIMD extensions like SSE/AVX, or by parallelizing the code across multiple threads using a parallel programming framework like TBB. In some cases, combining multi-threading across cores and vectorization within a core can achieve multiplicative effects in performance on the CMP. However, as will be seen later in this section, combining these techniques is not as trivial as it may seem and, in certain cases, may actually *worsen* performance. If the performance target is still not met, the application can be written for accelerators such as the MIC or GPGPU. This may require using a different

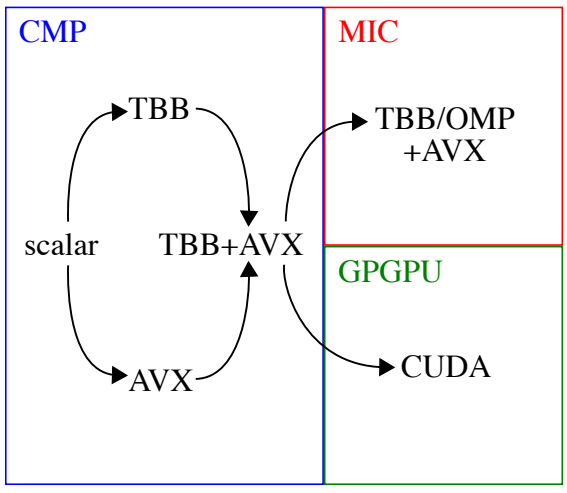


Figure 2.1: Example Development Flow for High-Performance Applications – Application development begins with a single-threaded scalar implementation on a CMP, before proceeding to a single-threaded vectorized implementation using AVX or a multi-threaded non-vectorized implementation. Combining multi-threading with vectorization can sometimes improve performance. Further performance improvements can be achieved by porting the application to accelerators like MICs or GPGPUs, which may require a different software framework.

software framework such as OpenMP or CUDA. Even if a common software framework is used across all architectures, target-specific optimizations and re-tuning are almost always necessary. Both MICs and GPGPUs use multi-threading across cores, but use different mechanisms for accelerating execution within a core (i.e., packed-SIMD units for MICs, SIMT engines for GPGPUs). It is also worth noting that due to the fundamental differences in programming models between CMPs and MICs/GPGPUs, the programmer may choose to jump directly from a scalar implementation on CMPs to the MIC/GPGPU implementation. The caveat is that a careful analysis of the type of parallelism and amount of parallelism available is necessary to ensure the application is suitable for these accelerators.

Figure 2.2 shows the performance of five platforms described in the development flow above on the seven application kernels listed in Table 2.1. CMP experiments were conducted on Intel Xeon E5-2620 v3 (12 cores, 2.40GHz, AVX2, 16MB L3), MIC experiments were conducted on Intel Xeon Phi 5110P (60 cores, 1.05GHz, AVX-512, 30MB L2), and GPGPU experiments were conducted on NVIDIA Tesla C2075 (14 cores, 575MHz, 6GB DRAM). Application kernels are compiled using the Intel compiler (ICC v.14.0.2) as it has been proven to have the greatest success in auto-vectorizing code [MGG+11]. All results are normalized to an optimized scalar implementation of the application kernel on a CMP.

The following includes brief descriptions of the application kernels examined in this study. More details can be found in Section 6.1. *sgemm* is a regular kernel that performs a single-precision matrix multiplication for square matrices using a standard blocking algorithm; computation is parallelized across blocks. *dct8x8m* is a regular kernel that calculates the 8x8 discrete cosine transform on an image; computation is parallelized across 8x8 blocks. *mriq* is a regular kernel that performs image reconstruction for MRI scanning; computation is parallelized across the output magnetic field gradient vector. *rgb2cmyk* is a regular kernel that performs color space conversion on an image and computation is parallelized across the rows. *bfs-nd* is an irregular kernel that generates a breadth-first-search tree from a directed cyclic graph; computation is parallelized across the frontier. *maxmatch* is an irregular kernel that identifies a maximal matching on an undirected graph; computation is parallelized across edges. *strsearch* is an irregular kernel that implements the Knuth-Morris-Pratt algorithm to search for a set of substrings in byte streams; computation is parallelized across different streams.

Name	Suite	Input
sgemm	Custom, CUDA SDK	2K×2K float matrix
dct8x8m	Custom, CUDA SDK	518K 8×8 blocks
mriq	Custom	262K-space 2K points
rgb2cmyk	Custom	7680×4320 image
bfs-nd	PBBS [SBF+12], LSG2.0 [BNP12]	rMatG_J5_10M
maxmatch	PBBS [SBF+12], Custom	randLocG_J5_10M
strsearch	Custom	512 strs, 512 docs

Table 2.1: Application Kernels for Native Experiments – Benchmarks suites used as the basis for the CMP/MIC/GPGPU implementations may be necessarily different depending on availability. Every effort was made to use the same parallel algorithm across the various implementations unless otherwise stated in the text. Datasets used for native experiments are larger than those used for simulator experiments to saturate hardware resources. Average execution time for the scalar implementations is ∼60 seconds.

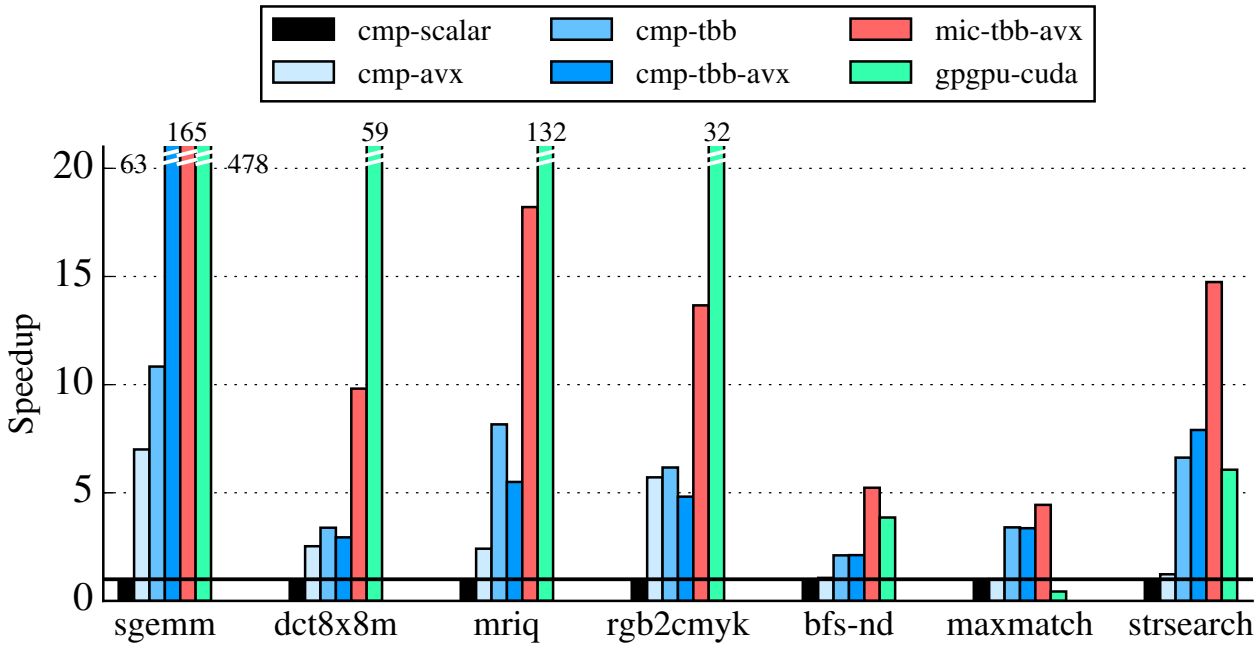


Figure 2.2: Performance Comparison of Selected SW/HW Platforms – Normalized to *cmp-scalar*, the non-vectorized single-threaded implementation. *cmp* = Intel Xeon E5-2620 v3 (12 cores, AVX2/256b); *mic* = Intel Xeon Phi 5110P (60 cores, AVX-512); *gpgpu* = NVIDIA Tesla C2075 (14 SMs); *avx* = ICC v15.0.3 with auto-vectorization [MGG+11]; *tbb* = TBB v4.3.3; *cuda* = CUDA v7.5.17.

2.1 CMP Platforms

The CMP platforms examined in this study use AVX and/or TBB as software frameworks and CMPs with packed-SIMD units as hardware architectures. To clarify, AVX as a software framework is referring to the optimizations for successfully auto-vectorizing code in conjunction with ICC-supported compiler hints, rather than the vector extensions themselves. One key difference between AVX and TBB as software frameworks is the parallel abstraction: AVX operates on lower-level packed data within a core, whereas TBB operates on higher-level tasks across cores. Therefore, one could make a reasonable argument that combining these disjoint abstractions actually changes the programming model enough that TBB+AVX should be treated as a separate software framework.

2.1.1 Optimizations for Successful Auto-Vectorization

The *cmp-avx* results in Figure 2.2 show the performance of the single-threaded vectorized implementation using AVX on a CMP. Vectorization is the process of generating special vector instructions that use packed-SIMD units to operate on packed data (e.g., eight words per vector register in AVX2). Since packed-SIMD units require SIMD groups of packed data to be processed in lock-step, it cannot efficiently handle control-flow or memory-access divergence. As such, application kernels with regular loop-task parallelism (i.e., *sgemm*, *dct8x8m*, *mriq*) see a speedup of at least 2.5×, whereas application kernels with irregular loop-task parallelism (i.e., *bfs-nd*, *max-match*, *strsearch*) see negligible benefits.

Since a 256b-wide packed-SIMD unit increases floating-point operation throughput by a factor of eight (not accounting for fused multiply-adds), the ideal speedup for a perfectly compute-bound application would be 8×. Of course, no application is perfectly compute-bound, so it is very difficult to reach this upper bound. However, it is clear that more compute-bound application kernels achieve higher speedups compared to more memory-bound application kernels. For instance, the more compute-bound *sgemm* yields a speedup of 7×, whereas the more memory-bound *dct8x8m* has a less pronounced speedup of 2.5×. Although both application kernels use blocking algorithms to improve cache locality and help alleviate memory bottlenecks, *dct8x8m* has a lower arithmetic intensity [WWP09] due to operating on smaller 8x8 blocks, which makes it more memory-bound than *sgemm*.

```
void rgb2cmyk_scalar(
 RgbPixel** src, CmykPixel** dst, int size)
{
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {

      // Calculate intermediate values
      int tmp_c = 255 - src[i][j].r;
      int tmp_m = 255 - src[i][j].g;
      int tmp_y = 255 - src[i][j].b;
      int tmp_k = std::min(
        tmp_c, std::min(tmp_m, tmp_y));

      // Store results into output array
      dst[i][j].k = tmp_k;
      dst[i][j].c = tmp_c - tmp_k;
      dst[i][j].m = tmp_m - tmp_k;
      dst[i][j].y = tmp_y - tmp_k;
    }
  }
}
```

(a) Single-Threaded Implementation

```
void rgb2cmyk_tbb(
 RgbPixel** src, CmykPixel** dst, int size)
{
  // Flatten matrices
  RgbPixel* flat_src, flat_dst;
  flatten(flat_src, src, size);
  ...

  // Parallelize computation across rows
  parallel_for(
    blocked_range(0, size*size, TASK_SIZE),
    [&] (blocked_range r) {
    for (int i = r.begin();
            i != r.end(); i++) {

      // Calculate intermediate values
      int tmp_c = 255 - flat_src[i].r;
      ...

      // Store results into output array
      flat_dst[i].k = tmp_k;
      ...
  }});
  ...
}
```

(c) Multi-Threaded Implementation with TBB

```
void rgb2cmyk_avx(
  RgbPixel** src, CmykPixel** dst, int size)
{
  // Restructure data into structs of arrays.
  // Copy data to aligned arrays.

  int nvecs_in_row = ceil(size / SIMD_WIDTH);
  int aligned_size = nvecs_in_row * SIMD_WIDTH;

  byte* __restrict__ aligned_r =
    (byte*) memalign(
      aligned_size * sizeof(byte), SIMD_WIDTH);
  ...

  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      RgbPixel pixel = src[i][j];
      aligned_r[i*aligned_size+j] = pixel.r;
      ...
  }}

  // Vectorize computation across rows
  for (int i = 0; i < size; i++) {
    #pragma simd vectorlength(SIMD_WIDTH)
    for (int j = 0; j < aligned_size; j++) {
      // Calculate index
      int idx = i * aligned_size + j;

      // Calculate intermediate values
      int tmp_c = 255 - aligned_r[idx];
      ...
      int tmp_k = min_nobranch(
        tmp_c, tmp_m, tmp_y);

      // Store results into aligned output
      aligned_k[idx] = tmp_k;
      ...
  }}

  // Copy data to original output

  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      CmykPixel* pixel = &dst[i][j];
      pixel->c = aligned_c[i*aligned_size+j];
      ...
  }}
}
```

(b) Single-Threaded Implementation with AVX

Figure 2.3: Example Development Flow for *rgb2cmyk* Kernel – Various implementations of color-space conversion on sample image for CMP/MIC.

The preferred method of vectorization is auto-vectorization by the compiler, but this term is a bit of a misnomer since naive attempts at auto-vectorization by only annotating loops with `#pragma simd` or `#pragma ivdep` rarely result in any significant speedups. For example, this naive approach yielded speedups of less than 1.10× across all seven application kernels. With current compiler technology, maximally utilizing the packed-SIMD units requires numerous manual optimizations [AJ88, DKH11], sometimes referred to as explicit vectorization. This is true even for embarrassingly regular loop-task parallel application kernels like *sgemm*.

The key objectives of these *vector-optimizations* are to eliminate control flow and enable vector memory accesses. Figure 2.3(a) and Figure 2.3(b) show the single-threaded scalar and vectorized implementations, respectively, of the *rgb2cmyk* application kernel, which highlights the complexities of several vector-optimizations.

Eliminating control flow requires converting branches into arithmetic, often by using bit-level masking and shifting. For example, the expression `y = (x < 0) ? -1 : 1` is equivalent to `y = 1 - (x >> 31) * 2`. The tradeoff here is that this can reduce work efficiency by forcing the same operation to be applied to all data elements regardless of whether or not useful work is actually required. When branches are unavoidable, a less compute-efficient algorithm that is more amenable for vectorization can be used, but there is no guarantee the benefits of vectorization will outweigh the loss in compute efficiency. This is the case in *dct8x8m*, where the more compute-efficient Loeffler-Ligtenberg-Moschytz [LLM89] algorithm had unavoidable branches, thus a more naive algorithm was required for vectorization.

Enabling vector memory accesses means the programmer must ensure that memory accesses will be at SIMD-width-aligned boundaries with unit-stride access patterns. This requires explicitly allocating aligned arrays from/to which data needs to be copied before/after computation, and converting arrays of structs into structs of arrays. Lines 4–21 in Figure 2.3(b) showcases these vector-optimizations. Multi-dimensional arrays must be padded with ghost cells such that each row/column, depending on which dimension computation is vectorized across, is a multiple of the SIMD-width in order to ensure alignment persists across rows/columns. Aligning arrays can be complicated by the use of C++ data structures like `std::vector`, which require custom allocators to ensure alignment of the underlying array in memory. The tradeoff here is that allocating and transferring aligned data can add a non-trivial performance overhead. This overhead is better amortized as the time spent on the computation relative to the transfer increases.

```
void bfs_nd_tbb(Node* G, int nnodes, int nedges) {
  vector<int> front  (nedges, 0);
  vector<int> counts (nnodes, 0);
  ...

  int front_sz = 1;
  visited[0] = 1;

  // Process nodes in frontier until frontier is empty
  while (front_sz > 0) {

    // Initialize number of neighbors for nodes in frontier
    parallel_for(blocked_range(0, front_sz), [&] (blocked_range r) {
      for (int i = r.begin(); i != r.end(); i++)
        counts[i] = G[front[i]].degree;
    });

    // Parallelize computation across nodes in frontier
    int nremain = get_remaining_nodes(counts, front_sz);
    parallel_for(blocked_range(0, front_sz), [&] (blocked_range r) {
      for (int i = r.begin(); i != r.end(); i++) {
        int k = 0;
        int v = front[ri];
        int c = counts[ri];

        // Iterate across neighbors and update if lock obtained
        for (int j = 0; j < G[v].degree; ++j) {
          int ngh = G[v].neighbors[j];
          if (visited[ngh] == 0 && CAS(&visited[ngh],0,1))
            front_next[c+j] = G[v].neighbors[k++] = ngh;
          else
            front_next[c+j] = -1;
        }

        G[v].degree = k;
      }
    });

    // Determine next frontier
    front_sz = filter_frontier(front, front_next, nremain);
  }
}
```

Figure 2.4: Using TBB in *bfs-nd* Kernel – Multi-threaded implementation using TBB of non-deterministic breadth-first search. This kernel is not suitable for vectorization due to unavoidable control flow and memory-access divergence, as well as the atomic compare-and-swap.

Other vector-optimizations include annotating non-overlapping arrays with the `__restrict__` keyword, using `__assume_aligned` hints when passing aligned array pointers to helper functions, and the aforementioned annotation of vectorizable loops with `#pragma simd/ivdep`.

Note that the vector-optimizations described above only apply to regular loop-task parallel applications. More irregular loop-task parallel applications may simply have too many unavoidable branches and unpredictable memory-access patterns (i.e., gathers, scatters) that prevent vectorization. Figure 2.4 shows an example of this case, *bfs-nd*, where vectorizing across nodes in the frontier fails due to the unavoidable control-flow divergence between nodes with different numbers of neighbors as well as the memory-access divergence when accessing nodes in `G`. Vectorizing across the neighbors for a given node also fails due to the data-dependent branch on `visited` and the atomic compare-and-swap for which there is no vector equivalent.

In summary, although AVX on CMPs can improve performance for *regular* loop-task parallel applications, the required vector-optimizations greatly reduce productivity as apparent from the difference between Figure 2.3(a) and Figure 2.3(b). Implementing, testing, and tuning these seven application kernels for AVX on CMPs took an experienced programmer several programmer-weeks, and no amount of manual optimization significantly improved performance for *irregular* loop-task parallel applications.

2.1.2 Multi-Threading with Task-Based Runtime

The *cmp-tbb* results in Figure 2.2 show the performance of the multi-threaded non-vectorized implementation using TBB with 12 threads on a CMP. The CMP in this study has 12 four-way superscalar out-of-order cores, each capable of executing two virtual threads, but in this study, one thread is assigned per core, so that the impact of scaling can be isolated from the impact of virtualization. Unlike vectorization, multi-threading can improve performance for both regular and irregular loop-task parallel application kernels, as seen by the 2–11$\times$ speedup across all application kernels. However, like vectorization, the benefits of multi-threading can be limited by memory bottlenecks. As such, the most memory-bound application kernels (i.e., *dct8x8m*, *rgb2cmyk*, *bfs-nd*, and *mm*) see a lower average speedup of 3$\times$. Profiling the multi-threaded implementation of *dct8x8m* reveals that it is stalled on memory accesses 71% of the execution time, whereas *mriq* is only stalled on memory accesses 14% of the execution time. The memory bottlenecks of these application kernels are partially due to the size of the datasets as well. For example, the datasets

required to saturate hardware resources on the MIC/GPGPU for *dct8x8m* and *rgb2cmyk* are the largest of all the application kernels in this study. Reducing the dataset size by a factor of four yields an average speedup of 10× with TBB. In the case of *bfs-nd* and *mm*, the memory bottleneck is further exacerbated by their reliance on fine-grain locking via atomic memory operations.

TBB is used due to its productive task-based programming model, library-based implementation, and work-stealing runtime for fine-grain dynamic load balancing. Although there are other task-based parallel programming frameworks, the library-based approach is preferred to improve portability as opposed to a compiler-based approach like Cilk or OpenMP. However, preliminary experiments showed that performance trends described below for TBB extended to OpenMP as well.

Figure 2.3(c) shows what the multi-threaded implementation of *rgb2cmyk* looks like using TBB. One strategy is to parallelize computation across the rows of the image. Although in many cases this is sufficient, depending on the dimensions of the image, such coarse-grain parallelization may limit load balancing, especially across a large number of cores. Instead, it is often worthwhile to flatten any multi-dimensional arrays into a single-dimensional array to generate finer-grain tasks that improve load balancing. For example, in *rgb2cmyk*, flattening the input matrix improved performance by 14%. Note the `parallel_for` construct used to implement this strategy. The `blocked_range` represents the range of loop iterations to which the function should be applied. The `TASK_SIZE` macro represents the TBB *grain size* which determines the minimum number of loop iterations in the range of executed tasks. This is an important parameter that needs to be changed for different application kernels. Smaller tasks improve load balancing, whereas bigger tasks introduce more runtime/loop overhead. If the cache footprint of each task is very small, bigger tasks can improve cache locality by ensuring consecutive blocks in memory are processed by the same core. In this study, grain sizes of 2–32 resulted in the highest performance. In general, relatively minimal changes are required to parallelize the single-threaded implementation shown in Figure 2.3(a).

In summary, TBB was the most productive framework in this study. Approximately one programmer-week was required to develop relatively high-performance parallel implementations of all the application kernels. Unfortunately, TBB is limited to exploiting loop-task parallelism across cores and is not able to leverage packed-SIMD extensions to exploit parallelism within a core without manual intervention.

```
void rgb2cmyk_tbb_avx(RgbPixel** src, CmykPixel** dst, int size)
{
  // Restructure data into structs of arrays.
  // Copy data to aligned arrays.

  ...

  // Parallelize computation across aligned chunks of rows

  parallel_for(blocked_range(0, size * CHUNKS_IN_ROW, TASK_SIZE),
    [&] (blocked_range r) {
    for (int ri = r.begin(); ri != r.end(); ri++) {
      int i        = ri / CHUNKS_IN_ROW;
      int i_offset = ri % CHUNKS_IN_ROW;
      int j_start  = i_offset * aligned_size / CHUNKS_IN_ROW;
      int j_end    = j_start + aligned_size / CHUNKS_IN_ROW;

      // Vectorize computation within a chunk
      #pragma simd vectorlength(SIMD_WIDTH)
      for (int j = j_start; j < j_end; j++) {
        // Calculate index
        int idx = i * size + j;

        // Calculate intermediate values
        int tmp_c = 255 - aligned_r[idx];
        ...

        // Store results into aligned output
        aligned_k[idx] = tmp_k;
        ...
      }
    }
  });

  // Copy data to original output

  ...
}
```

Figure 2.5: Combining TBB and AVX in *rgb2cmyk* Kernel – Code that is restructured to increase the number of tasks for better load balancing without hindering auto-vectorization.

2.1.3 Combining Multi-Threading with Auto-Vectorization

The *cmp-tbb-avx* results in Figure 2.2 show the performance of a multi-threaded vectorized implementation using TBB with 12 threads and AVX on a CMP. Ideally, one would expect to see a multiplicative effect in performance on *regular* loop-task parallel application kernels by combining multi-threading across cores and vectorization within a core. Irregular loop-task parallel application kernels would not be expected to yield any improvements in performance from vectorization as discussed above. Although the 63× speedup on `sgemm` does approach the ideal multiplicative speedup of 77× (7× from TBB, 11× from AVX), both `dct8x8m` and `mriq` surprisingly result in a *slowdown*. There are two key reasons why combining these techniques might not always yield a multiplicative effect.

First, task partitioning with TBB can limit auto-vectorization with AVX. Recall that vector memory accesses in AVX can only be generated when the compiler can guarantee memory accesses are SIMD-width-aligned. As such, the grain size in TBB should presumably be set to a multiple of the SIMD-width so that every task accesses SIMD-width-aligned memory. Unfortunately, TBB cannot guarantee exact task sizes at compile time due to the recursive decomposition algorithm the runtime uses to partition tasks. Since the compiler cannot guarantee tasks will access SIMD-width-aligned memory, it cannot safely vectorize a loop which might have been vectorizable *without* using TBB.

Second, vector-optimizations to enable AVX can limit load balancing with TBB. Specifically, eliminating control-flow divergence can also eliminate opportunities for load balancing by superficially equalizing the work across the SIMD group. Considering `cmp-tbb`, which has no vectorization, threads operating on tasks with less work due to control-flow divergence can more quickly steal another task than threads operating on tasks with more work. However, with vectorization, the entire SIMD group always does the maximum amount of work if all branches have been converted into arithmetic.

One easy way to address how task partitioning interferes with auto-vectorization is to parallelize computation across rows/columns (for multi-dimensional data structures) and ensuring SIMD-width-alignment of each row/column. However, as mentioned above, such coarse-grain parallelization can limit load balancing. Figure 2.5 highlights an optimization that can help improve load balancing when combining TBB and AVX. The key here is to split rows into SIMD-width-aligned chunks, then parallelize computation across these chunks. The chunk index is used

to calculate the row and column indices into `src` and `dst`. The compiler can safely vectorize the annotate loop because `j_start` is guaranteed to be a multiple of the SIMD width as long as the chunk size is also a multiple of the SIMD width. Unfortunately, comparing this code to Figure 2.3(c) shows that combining TBB and AVX further increases the code complexity beyond a pure AVX implementation.

In summary, combining TBB with AVX does not always guarantee improvements in performance and negates the productivity of TBB. It took an experienced programmer another several programmer-weeks of manual optimizations to develop the multi-threaded vectorized implementations of the application kernels.

2.2 MIC Platforms

The *mic-tbb-avx* results in Figure 2.2 show the performance of the multi-threaded vectorized implementation using TBB with 60–240 threads and AVX on a MIC. The MIC in this study has 60 single-issue in-order cores, each capable of multi-threading four physical threads. MICs have longer cache-to-cache latencies than CMPs and do not have a shared L3 cache [Rei12, Bol12]. MICs are designed to accelerate applications with immense regular loop-task parallelism. This is evident from the many lightweight cores with very wide packed-SIMD units (512b wide). Therefore performance on the MIC largely depends on maximally utilizing these packed-SIMD units. The results also validate this claim as application kernels that can be successfully auto-vectorized (i.e., shows performance improves with *cmp-avx*), exhibit the highest speedups on the MIC. Due to the less aggressive memory system, some regular loop-task parallel application kernels that are memory-bound, like *dct8x8m* and *rgb2cmyk*, may struggle to achieve *resource-proportional speedups*. Resource-proportional speedups refers to performance that is reasonable for the amount of available hardware resources (e.g., closer to achieving maximum instructions per cycle, floating-point operations per second, memory throughput, etc.). Even irregular loop-task parallel application kernels can achieve high performance on MICs by leveraging the significantly increased core count, *as long as they are not memory-bound*. For instance, *strsearch* is able to achieve an impressive speedup of 15× even without any vectorization.

MICs support two programming models: a native model and an offload model. The *native model* is used to compile code to run directly on the MIC. This approach is generally more pro-

ductive but is limited by the memory available on the MIC (8GB DRAM). Since the code executes natively on the MIC, any dependent libraries must also be cross-compiled for the MIC. Although Intel provides MIC-optimized versions of many of their libraries, like TBB, any custom libraries may need to be separately optimized for the MIC. The *offload model* is a hybrid approach that is used to compile code to run on both the host and the MIC, where the former acts as the driver that spawns off coarse-grain kernels to the latter. This approach is much less productive, as it requires many manual adjustments common to all offload programming models. The challenges involved with such models, like CUDA, will be discussed in the next section. However, there are intricacies that are specific to the MIC offload model that further decrease productivity. A key example is the need to manually specify which data structures should be allocated, reused, or deleted between kernel launches. This kind of explicit memory management is essential for extracting the highest performance out of the MIC. In this study, all MIC implementations of the application kernels employ the native model.

Even though *mic-tbb-avx* uses the same software framework as *cmp-tbb-avx* in this study, there is still a non-trivial amount of re-tuning necessary to yield highest performance on the MIC. For example, due to the much higher core count, load balancing becomes even more important on the MIC. The optimal grain size on the MIC is usually smaller than on the CMP, and optimizations as previously shown in Figure 2.5 become even more important on the MIC. Furthermore, the optimal number of threads is different across application kernels. In fact, specifying the maximum number of threads per core is not always the best option, as there is a delicate balance between cache locality and hiding microarchitectural latencies. Finally, the lighter-weight cores also mean system calls are much more expensive on the MIC. In some cases it may be necessary to restructure code to eliminate dynamic memory allocation and TBB thread spawns to reduce performance overheads. As a reminder, the challenges with achieving a multiplicative effect in performance by combining TBB and AVX applies to the MIC as well.

In summary, porting and optimizing CMP implementations for the MIC is a non-trivial process, even when using the same software framework, and MICs may not always achieve resource-proportional performance for both regular and irregular loop-task parallel applications due to memory bottlenecks and/or lack of vectorization.

2.3 GPGPU Platforms

The *gpgpu-cuda* results in Figure 2.2 show the performance of a multi-threaded implementation using CUDA with 448 threads on a GPGPU. GPGPUs have higher computational throughput than CMPs/MICs, as evident by the 60–165× speedups on *regular* loop-task parallel application kernels. Irregular loop-task parallel application kernels struggle to achieve resource-proportional performance due to serialized execution and inefficient gathers/scatters. Application kernels that require inter-thread communication, such as *bfs-nd* and *maxmatch*, experience even worse performance degradation due to atomic memory operations and synchronization barriers.

In terms of productivity, CUDA offers a unified approach to exploiting loop-task parallelism across and within cores, but its offload programming model requires substantial changes [nvi15] which include: explicit allocation/copying of device memory, manual work partitioning into blocks/grids (due to hardware scheduler), effective utilization of texture/shared memory, limitations on kernel arguments (e.g., complex structs with pointers to pointers), and restrictions on standard C++ containers like `std::vector` within a kernel. The last point is addressed in part by the external libraries like Thrust [BH12]. In addition, inter-thread communication is especially difficult to express efficiently, which may require barriers (i.e., `__syncthreads`). Barriers in CUDA should only be used when *all* threads are guaranteed to reach the barrier, otherwise deadlock is possible. If this condition cannot be guaranteed, it might be necessary to change the algorithm itself to avoid this barrier, for example by breaking the kernel into multiple phases as in `maxmatch`.

The challenges of mapping irregular loop-task parallel applications to GPGPUs has been intensively examined in my previous publication [KB14] as well as many other studies [BNP12, NBP13a, NBP13b, HN07, LWH10, MLNP+10, MLBP12]. Many irregular loop-task parallel applications iteratively apply an operator to nodes in a graph which can modify the graph and generate more work. When mapping such applications to GPGPUs, two common approaches are taken: topology-driven and data-driven. The CUDA pseudo-code for the topology- and data-driven implementations of *bfs-nd* are shown in Figure 2.6. The algorithm iteratively applies a compute operator (i.e., `compute()`) on a subset of nodes at which useful work is required, in an undirected, weighted graph. Such nodes are referred to as *active nodes*, whereas nodes at which no useful work will be done are referred to as *inactive nodes*. The node ID of an active node is called a *work ID*. A check operator (i.e., `check()`) determines whether a node is active or inactive. The compute

```
__global__ void
bfs_nd_topo(Node* nodes, bool* done_ptr) {
  int tidx =
    blockIdx.x * blockDim.x + threadIdx.x;
  Node my_node = nodes[tidx];
  if (check(my_node)) {
    compute(my_node)
    *done_ptr = false;
  }
}

int main() {
  bool done = false;
  while (!done) {
    done = true;
    topo_driven<<<N>>>(nodes, &done);
  }
}
```

(a) Topology-Driven Approach

```
__global__ void
bfs_nd_data(Node* nodes, WL* wl) {
  while (widx = wl->pull()) {
    Node my_node = nodes[widx];
    compute(my_node);
    for (int i = 0;
         i < my_node.nneighbors; i++) {
      int neighbor = my_node.neighbor(i);
      if (check(nodes[neighbor]))
        wl->push(neighbor);
    }
  }
}

int main() {
  init_wl<<<N>>>(nodes, wl);
  data_driven<<<M>>>(nodes, wl);
}
```

(b) Data-Driven Approach

Figure 2.6: Topology- vs. Data-driven Implementations of *bfs-nd* – Pseudo-code showing two common approaches to mapping irregular loop-task parallel applications to GPGPUs. The check operator determines if nodes are active and the compute operator performs work at these nodes. This can activate previously inactive nodes in the next super-step. Execution completes when all nodes are inactive. `N` = number of nodes, `M` = max number of hardware threads, `WL` = software worklist class. In the topology-driven example, there is an acceptable race condition when updating `done_ptr`. In the data-driven implementation, `M` threads are spawned with the kernel since every thread will stay in the loop until no more work is in the worklist. An initialization kernel populates the worklist with active nodes before the main kernel is called. A pull returns zero when the worklist is empty.

operator often accesses neighboring nodes and can activate inactive nodes to be processed later. Execution completes when all nodes are inactive and will not be activated again. Note that this kernel is roughly analogous to the multi-threaded TBB implementation of *bfs-nd* in Figure 2.4, except that instead using a `parallel_for` in software to determine node IDs, the CUDA implementation relies on hardware-assigned thread/block indices to determine the node IDs.

Figure 2.6(a) shows the pseudo-code for the topology-driven implementation of *bfs-nd*. During each super-step, every thread checks to see if the node corresponding to its thread index is active. The compute operator is applied to active nodes and threads with inactive nodes exit without doing any useful work. Every kernel call represents a super-step and the kernel is invoked as long as there are active nodes in the next super-step. All nodes, both active and inactive, are visited every super-step and the number of threads is the number of nodes.

Figure 2.6(b) shows the pseudo-code for the data-driven implementation of *bfs-nd*. The multi-threaded TBB implementation of *bfs-nd* in Figure 2.4 is also data-driven. In this case, an initialization kernel pre-checks all the nodes and populates the worklist with only active nodes. Although checking inactive nodes in the initialization is not useful, this overhead is only incurred once, instead of at every super-step like in the topology-driven implementation. Every thread in the main kernel pulls a work ID from the worklist with an atomic memory operation and applies the compute operator to the corresponding active node. Newly activated nodes are pushed onto the worklist with atomic memory operations, so that only active nodes are ever visited. Even though the check for determining new nodes to be activated is shown separate from the compute in the example, the check is usually combined with useful work in the compute phase. Although this can be done on the topology-driven implementation as well, the difference is that threads operating on inactive nodes in topology-driven implementations must pay this overhead again at the beginning of every super-step. Notice that the super-step loop seen in the topology-driven implementation is moved inside of the kernel in the data-driven implementation so that operations at nodes across multiple super-steps are overlapped. It is sufficient to only spawn a number of threads equal to the maximum number of hardware threads on the GPGPU since every thread stays in the loop until there are no more active nodes. Special consideration must be taken to prevent threads from prematurely dropping out of the computation loop. To this end, the `pull()` function on the worklist is implemented to return a special wait token if there are any threads processing an active node, *even if there are no more work IDs in the worklist*. More details about this mechanism can be found in [KB14].

The data-driven implementation exposes more parallelism by overlapping operations across multiple super-steps and increases work efficiency by avoiding useless work. However, the data-driven implementation is not without its weaknesses, including: high memory contention from accessing a shared worklist, and high memory-access irregularity from dynamic load balancing. These weaknesses can be addressed to some extent using optimizations that focus on: (1) reducing memory contention on pulls, (2) reducing memory contention on pushes, (3) improving load balancing, and (4) further increasing work efficiency. Examples of these optimizations include: worklist double-buffering, work chunking, work donating, variable kernel configuration, and hierarchical worklists. With aggressive software optimizations, data-driven implementations are gen-

erally able to outperform topology-driven implementations, but this does not mean the former is optimal for every application.

In summary, porting CMP implementations to the GPGPU was a heavily involved process requiring several programmer-weeks for all the application kernels. Optimizations for irregular loop-task parallel applications are particularly difficult, and even extensive optimizations do not guarantee improvements in performance for irregular loop-task parallel applications. Furthermore, there are multiple approaches to mapping irregular loop-task parallel applications to GPGPUs and it is not always clear which approach is optimal.

2.4 Summary and Discussion

The insights from this study can be condensed into four key observations. First, software frameworks across all examined platforms have their own weaknesses with respect to productivity, with the possible exception of TBB which requires relatively modest changes from the scalar implementation. Although TBB is highly productive, it is clear that combining TBB with AVX can negate this productivity. Second, it can be difficult to easily port applications across different hardware architectures, even when using the same software framework. Not only do the optimizations change with architectures, but the algorithm itself can change as well, making it necessary to maintain multiple architecture-specific implementations of an application. Third, exploiting loop-task parallelism across cores (e.g., multi-threading) and within a core (e.g., SIMD) do not always yield multiplicative effects in performance. While the software frameworks for exploiting loop-task parallelism across and within cores are composable, they are not necessarily aware of each other, which can lead to performance-degrading interference between these frameworks. Fourth, it is very difficult to achieve high performance on *irregular* loop-task parallel applications proportional to the resources available on the hardware architecture, especially on accelerators like MICs or GPGPUs. This is a consequence of these accelerators fundamentally being designed for *regular* loop-task parallelism (e.g., packed-SIMD units, SIMT engines).

Although Figure 2.1 shows one example of a development flow, there are other flows that attempt to address the 3P's using a purely software solution. For example, even though adoption of OpenCL has been limited, its original goal was to enable portable implementations across CMPs and accelerators [ope11]. OpenCL uses an offload programming model even for CMPs and MICs

in native mode. As such, most of the benefits and challenges associated with a CUDA-like model would apply to OpenCL on CMPs/MICs. For example, OpenCL provides better support for combining multi-threading and vectorization on CMPs/MICs by offering explicit vector data types that are automatically SIMD-width-aligned (albeit only for flattened arrays) and an implicit vectorization compiler pass that allows vectorization within work-groups as long as the work-group size is a multiple of 8 (or left to the compiler). However, this does not obviate the need for additional vector-optimizations such as eliminating control flow or dealing with an offload model's restrictions on arrays of complex structs and manually flattening multi-dimensional data structures. In addition, optimizations and tuning parameters across CMPs, MICs, and GPGPUs will still be different, requiring separate architecture-specific implementations. It should be acknowledged that OpenCL does make it easier to port a CMP implementation to the GPGPU. Finally, as with CUDA, the lack of a work-stealing runtime can also impact the performance of irregular loop-task parallel applications with high work variance across tasks. In summary, OpenCL has its own set of tradeoffs but ultimately it still has issues with productivity and portability, and as a purely software solution, OpenCL does not address the fact that the accelerators discussed in this study are designed for regular loop-task parallelism. For more details, please see Chapter 8.

In conclusion, an ideal 3P computing platform would have the productivity of TBB, portability that allows a single implementation to be efficiently mapped across multiple architectures, and a better guarantee of high performance on both regular and irregular loop-task parallel applications.

CHAPTER 3
FINE-GRAIN SIMT

This chapter describes an early attempt at a hardware-centric approach to achieving a 3P platform. The *fine-grain SIMT* (FG-SIMT) microarchitecture is a programmable accelerator designed to exploit regular loop-task parallelism [KLST13]. The goal of FG-SIMT is to combine the benefits of CMPs and GPGPUs by tightly integrating SIMT engines to GPPs and enabling fine-grain kernel launches between them. FG-SIMT seeks to improve productivity by using threads as a common parallel abstraction, similar to how CUDA threads are used in GPGPUs. FG-SIMT also served as a starting point for experimenting with using a single ISA on the GPP and a tightly integrated accelerator. Although this improves portability to a certain extent, only preliminary studies for enabling execution of the same binary on either the GPP or FG-SIMT were performed. FG-SIMT focuses on improving area-normalized performance in order to allow tight integration with GPPs, thus utilizes orders of magnitude less threads per core than a conventional GPGPU. The FG-SIMT microarchitecture inspired several aspects of the LTA engine, specifically mechanisms for efficiently exploiting regular loop-task parallelism. This early work further highlighted the need for innovation in both software *and* hardware to achieve a true 3P platform.

3.1 FG-SIMT Overview

General-purpose graphics-processing units (GPGPUs) are growing in popularity across the computing spectrum [YKM+11, nvi09, amd12a]. Most GPGPUs use a single-program multiple-data (SPMD) model of computation where a large number of independent threads all execute the same application kernel [NBGS08, ope11, Mic09]. Although these SPMD programs can be mapped to the scalar portion of general-purpose multicore processors (an active area of research [SGM+10, DKYC10]), architects can improve performance and efficiency by using specialized data-parallel execution engines to exploit the structure inherent in SPMD programs. Examples include compiling SPMD programs to packed-SIMD units [int12, KDY12, Col11a], to more traditional vector-SIMD units [int11b, amd12a], or to single-instruction multiple-thread (SIMT) units [amd11, int11a, nvi09, LNOM08].

SPMD programs can be mapped to SIMT microarchitectures using an explicit representation of the SPMD model: a SIMT kernel of scalar instructions is launched onto many *data-parallel*

threads. Threads use their thread index to work on disjoint data or to enable different execution paths. Threads are mapped to an architecturally transparent number of hardware thread contexts to enable scalable execution of many kernel instances in parallel. A SIMT microarchitecture usually includes several *SIMT engines*. Each engine is responsible for managing a subset of the threads, including its own inner levels of the memory hierarchy, and is relatively decoupled from the other engines.

SIMT microarchitectures exploit *control structure* and *memory-access structure* in SPMD programs to improve performance and area/energy efficiency. Control structure characterizes how often multiple threads execute the same instruction in the kernel, while memory-access structure characterizes the regularity of inter-thread addresses for the same load/store instruction. To exploit control structure, a SIMT engine executes a set of consecutively indexed threads (usually called a *warp* or *wavefront*) in lock-step on the SIMT engine, amortizing control overheads (e.g., instruction fetch, decode, interlocking) and hiding execution latencies. Performance and efficiency are maximized when all threads take the same path through the kernel. SIMT engines also include mechanisms for managing control divergence (i.e., threads within the same warp take different paths) and for facilitating reconvergence. To exploit memory-access structure, a memory coalescing unit dynamically compares memory requests made by threads and merges multiple scalar memory requests into one wide memory request, amortizing control overheads in the memory system (e.g., arbitration, tag check, miss management) and reducing bank conflicts. Performance and efficiency are maximized when all threads access the same or consecutive addresses.

Fine-grain SIMT (FG-SIMT) is a variant of SIMT architectures more appropriate for compute-focused data-parallel accelerators. The goal of FG-SIMT is to combine the benefits of CMPs and GPGPUs. This is achieved by tightly integrating SIMT engines to GPPs and enabling fine-grain kernel launches between them. To this end, it is necessary for FG-SIMT to focus on area efficiency by executing one warp at a time, using highly-ported register files to better exploit instruction-level parallelism, and relying on software-managed warp scheduling. FG-SIMT has a simple programming model (e.g., handling complex control flow or modular function calls from within data-parallel threads) yet still achieves much of the benefit of more traditional vector-SIMD execution.

One of the novel features of FG-SIMT is its ability to exploit *value structure*, which refers to situations where threads execute the same instruction operating on data that can be described in

a simple, compact form. For example, all threads might operate on the same value or on values that can be represented with a function of the thread index. Value structure in CUDA programs was characterized using static and dynamic analysis in previous works [CDZ09, CK11, Col11a], and techniques to track and exploit value structure in hardware have been explored to a limited extent [CDZ09, GKS13, CK11]. FG-SIMT has mechanisms to exploit value structure using compact affine execution of arithmetic, branch, and memory instructions.

The vision for a large-scale FG-SIMT processor includes a multi-core system with an on-chip network and sophisticated memory system, where each GPP has a dedicated SIMT engine. However, in this early study, the scope is limited to a detailed implementation of a single GPP with a SIMT engine. In retrospect, broadening the scope of FG-SIMT to a multi-core system revealed many unforeseen challenges with respect to the 3P's that motivated significant changes to both software and hardware which are reflected in the proposed LTA platform.

3.2 FG-SIMT Value Structure

Figure 3.1 shows an example application kernel written in C for FG-SIMT as well as the corresponding assembly sequence generated by the compiler. This example will be used throughout this section to illustrate how value structure occurs and highlight the opportunities for exploiting value structure.

Value structure occurs when values used by the same operation across threads can be represented in a simple, compact form. For example, lines 2–4 in Figure 3.1(b) load the thread index into a register, shift it by a constant, and then add the result to the array base pointer. These instructions operate on *affine values* that can be compactly represented in the following form:

$$V(i) = b + i \times s$$

where i is the thread index, b is the base, and s is the stride. Affine values with a stride of zero are a special case in which all the numbers in the sequence are identical, and are called *uniform values*. This kind of value structure can be exploited by compactly encoding affine values as a base/stride pair.

```
1  void ex_fgsimt_kernel(int y[], int a)
2  {
3    int idx = fgsimt::init_kernel(y, a);
4    y[idx] = a * y[idx];
5    if (y[idx] > THRESHOLD)
6      y[idx] = Y_MAX_VALUE;
7  }
8
9  void ex_fgsimt(int y[], int a, int n) {
10   fgsimt::launch_kernel(
11     n, &ex_fgsimt_kernel, y, a);
12 }
```

(a) FG-SIMT Kernel in C

```
1  u  lw    y_ptr, y_base_addr
2  a  tidx  i
3  a  sll   i, i, 0x2
4  a  addu  y_ptr, y_ptr, i
5  a  lw    y, 0(y_ptr)
6  u  lw    a, a_addr
7  g  mul   y, y, a
8  g  slti  t, y, THRESHOLD
9  a  sw    y, 0(y_ptr)
10 g  bnez  t, done
11 u  li    y, Y_MAX_VALUE
12 a  sw    y, 0(y_ptr)
13   done:
14 u  exit
```

(b) FG-SIMT Kernel in Assembly

Figure 3.1: Example of Value Structure in SIMT Code – Kernel multiplies input array by constant and then saturates to threshold. Explicit bounds check elminated for simplicity. Assembly instructions are manually labeled according to the ability for that dynamic instruction to ideally exploit value structure: `g`: generic, cannot exploit value structure; `u`: uniform, affine stride is zero; `a`: affine with non-zero stride.

Certain arithmetic operations can be performed directly on this compact affine representation to produce an affine result. This is called *affine arithmetic*. For example, an affine addition can be computed as follows:

$$V_0(i) = b_0 + i \times s_0 \qquad V_1(i) = b_1 + i \times s_1$$
$$V_0(i) + V_1(i) = (b_0 + b_1) + i \times (s_0 + s_1)$$

As another example, affine multiplication is possible if at least one of the operands is uniform:

$$V_0(i) = b_0 + i \times s_0 \qquad V_1(i) = b_1$$
$$V_0(i) \times V_1(i) = (b_0 \times b_1) + i \times (s_0 \times b_1)$$

It is straightforward to develop a system of rules for affine addition, subtraction, shifts, and multiplication (note that [CDZ09] does not consider affine multiplication). Using these rules, the instructions in Figure 3.1 that are able to operate directly on compactly encoded affine values have been labeled. A significant fraction of the instructions are affine in this assembly sequence. This is a fundamental consequence of SIMT-based microarchitectures; threads likely work on structured values to calculate the location of their input/output values (e.g., lines 2–4 in Figure 3.1) and to compute similar intermediate values (e.g., line 11 in Figure 3.1). In addition to affine arithmetic, value structure can also have a direct impact on control structure and memory-access structure. For

example, if a kernel has an inner loop where the number of iterations is the same across all threads, then the corresponding backwards branch exhibits uniform value structure. Furthermore, the load on line 5 in Figure 3.1 exhibits affine value structure in its source address.

3.3 FG-SIMT Microarchitecture

FG-SIMT is a SIMT variant suitable for compute-focused data-parallel accelerators with an emphasis on area efficiency. Compared to GPGPUs, each FG-SIMT engine only executes a single warp at a time, uses highly ported register files, includes a software programmable control processor, and lacks specialized hardware for graphics rendering. FG-SIMT still includes the ability to exploit control and memory-access structure to efficiently execute warps of data-parallel threads. The primary motivation for FG-SIMT was to design an area-efficient SIMT microarchitecture that would exploit intra-warp instruction-level and data-level parallelism to hide various latencies instead of relying on extreme, inter-warp temporal multithreading. Banked, shared L1 data caches are used to increase address bandwidth for scatters and gathers and multi-ported register files are used to better exploit instruction-level parallelism. FG-SIMT engines are also tightly integrated with a simple control processor which grants more explicit control over the SIMT issue logic, allowing fine-grain execution of general-purpose and data-parallel workloads. Figure 3.2 shows the microarchitecture for a FG-SIMT engine with an L1 memory system. The FG-SIMT execution engine includes the control processor (CP), FG-SIMT issue unit (SIU), eight SIMT lanes, and the FG-SIMT memory unit (SMU).

FG-SIMT Kernel Launch – The CP is a simple RISC processor with its own program counter (PC), register file, and integer arithmetic unit. The CP uses the following instruction to launch a FG-SIMT kernel:

```
launch_kernel r_n, kernel_addr
```

where `r_n` is a CP register containing the total number of threads to execute. The instruction saves the kernel start address and the return address in microarchitectural registers and initializes a warp counter to n/m where n is the value in `r_n` and m is the number of threads per warp. The instruction then initializes the active warp fragment register (AWFR) with a new *warp fragment*. A warp fragment is simply a PC and an active thread mask indicating which threads are currently

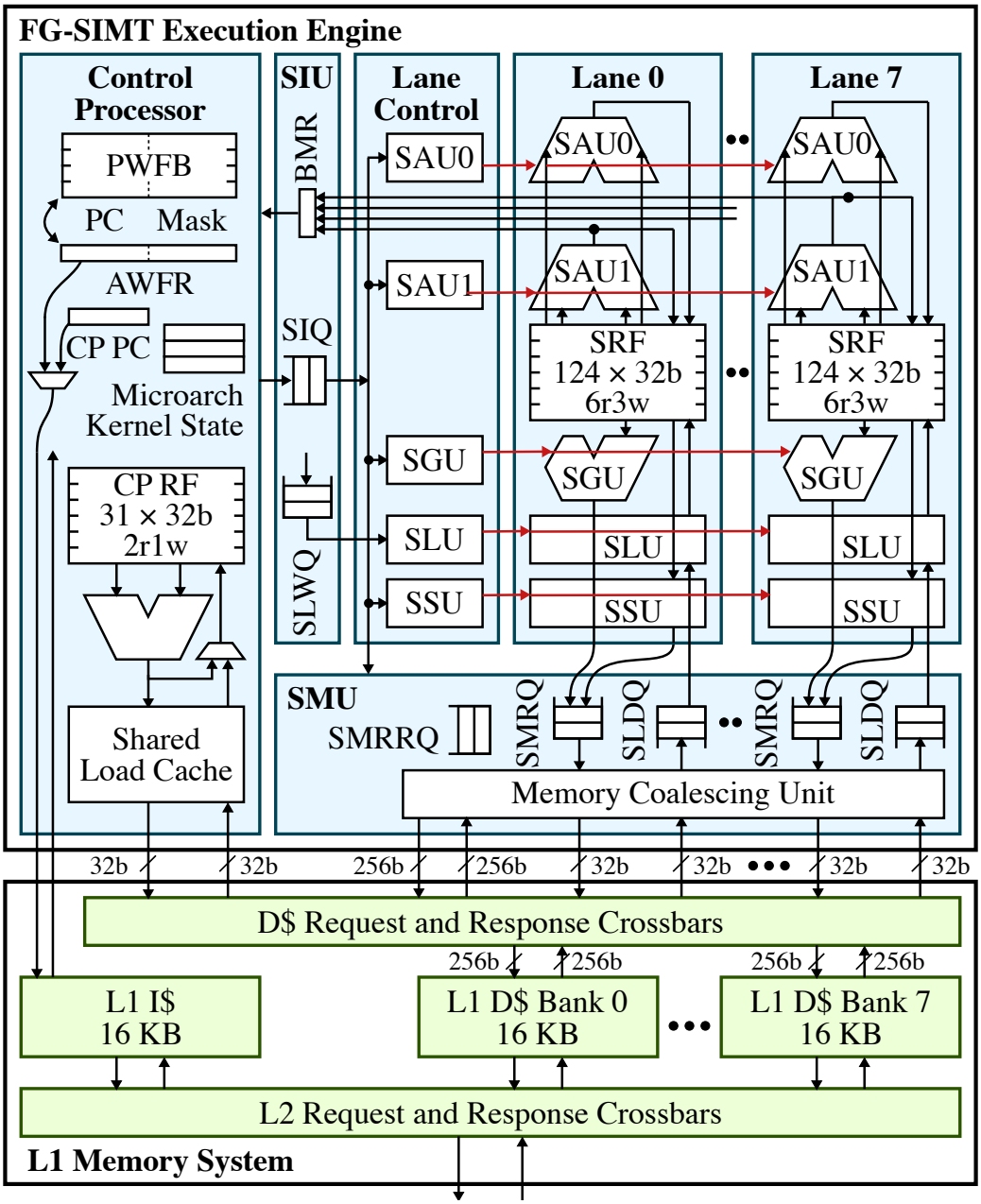


Figure 3.2: Detailed FG-SIMT Microarchitecture – Eight-lane FG-SIMT execution engine with L1 instruction cache and L1 eight-bank data cache. Only one branch resolution path shown for simplicity. PWFB = pending warp fragment buffer; AWFR = active warp fragment reg; CP = control proc; µArch Kernel State = kernel address reg, return address, warp counter reg; RF = regfile; SIQ = SIMT issue queue; SLWQ = SIMT load-write-back queue; SRF = SIMT regfile; SAU0/1 = SIMT arithmetic units; SGU = SIMT addr gen unit; SLU = SIMT load unit; SSU = SIMT store unit; SMRQ/SLDQ/SMRRQ = SIMT mem-req/load-data/resp-reorder queues; BMR = branch resolution mask register

executing that PC (i.e., initially the AWFR will contain the kernel address and a mask of all ones if $n >= m$).

FG-SIMT Control Processor – After launching a kernel, the CP fetches scalar instructions at the PC in the AWFR and sends them to the SIU, acting as the FG-SIMT engine's fetch unit. Unconditional jumps can be handled immediately in the CP. Conditional branch instructions are sent to the SIU, but the CP must wait for the branch resolution to determine the new active thread mask. If all threads take or do not take the branch, then the CP proceeds along the appropriate control flow path. If the threads diverge, then the CP creates a new warp fragment with the branch target and a mask indicating which threads have taken the branch. The new warp fragment is pushed onto the pending warp fragment buffer (PWFB) before continuing execution along the not-taken path. There are a variety of techniques in the literature for managing thread divergence and reconvergence [DAM+11, Col11b, CL08, int11a, amd11], but FG-SIMT uses a two-stack PC-ordered scheme which separates taken forward branches from taken backward branches in the PWFB to more effectively handle complicated control flow caused by intra-kernel loops [LAB+11]. The first warp brings kernel parameters into a shared load cache, reducing access latency for later warps. New warps are not scheduled until all fragments for the current warp execute to completion. Once all warps have been processed, the CP continues executing the instructions after the `launch_kernel` instruction.

FG-SIMT Issue Unit – The SIU is responsible for issuing instructions in-order to the SIMT lanes. Sophisticated scoreboarding is implemented to track various structural and data hazards and enable aggressive forwarding of data values between functional units. Load instructions are split into two micro-ops: address generation and SRF writeback. The former is issued to the lanes to generate the memory request, whereas the latter is enqueued into a SIMT load writeback queue (SLWQ). To facilitate aggressive forwarding from load instructions, writebacks are only issued to the SLU when all load responses have been received. Even though the SIU is single-issue, it is still possible (and very common) to keep multiple functional units busy, since each warp occupies a functional unit for four cycles (warp size of 32, eight lanes, four threads per lane).

In order to detect data hazards, in-flight writebacks to the SRF are tracked in two scoreboards: one for the VAUs and one for the VLU. RAW data hazards are detected by checking whether or not any of the read operands of the current instruction depend on an in-flight writeback. WAW data hazards are detected by checking whether or not the current instruction's writeback will happen

before any existing writebacks in the scoreboard. This means that the scoreboard must know the microarchitectural latencies of each arithmetic unit in the SAU. Structural hazards are detected with busy counters for each of the SIMT functional units. If the counter is non-zero, the SIU cannot issue to the corresponding functional unit due to a conflict on the SRF write port. For memory instructions, the SIU sends the number of load responses to wait for to the SMU and only issues the writeback to the SLU when all the load responses have returned from the memory system.

FG-SIMT Lanes – Each FG-SIMT lane is composed of five SIMT functional units: two arithmetic units (SAU), an address generation unit (SGU), a store unit (SSU), and a load unit (SLU). The lane control unit manages sequencing the functional units through the threads within a warp. Both SAUs can execute simple integer operations including branch resolution. Long-latency operations are fully pipelined and distributed across the two SAU units. These long-latency functional units include: integer multiply and divide, and IEEE single-precision floating-point add, subtract, convert, multiply, and divide. Addresses and store data of memory requests are generated by the SGU and SSU, respectively. The SLU manages writeback of load data and the SLWQ in the SIU enables overlapped memory requests. Each FG-SIMT lane also includes a bank of the SIMT register file (SRF). The SRF is a large, multi-ported register file that supports a maximum of 32 threads per warp. FG-SIMT supports density-time execution, which allows the lane control unit to skip sequencing chimes for which there are no active threads due to control-flow divergence. Density-time execution improves performance by effectively reducing the number of chimes for a given instruction when there is high control-flow irregularity. Note that this does complicate issue/bypass logic since instructions with different active chimes must stall before values can be bypassed.

FG-SIMT Memory Unit – The SMU includes several queues for storing memory requests and responses as well as logic for memory coalescing. Each lane has a memory request queue (SMRQ) and a load data queue (SLDQ). The SMU has eight 32b ports to the memory system, one for each lane, in addition to a single 256b port used for coalesced requests and responses. Requests to the same address or consecutive addresses across lanes can be coalesced into a single wide access. Coalescing is only possible when the requests are cache line aligned.

Every time a group of loads are sent from the SMU, a corresponding entry is pushed into the SIMT memory response order queue (SMROQ). The SMROQ ensures that memory responses will be written back in the order they were requested. This is necessary since although the memory

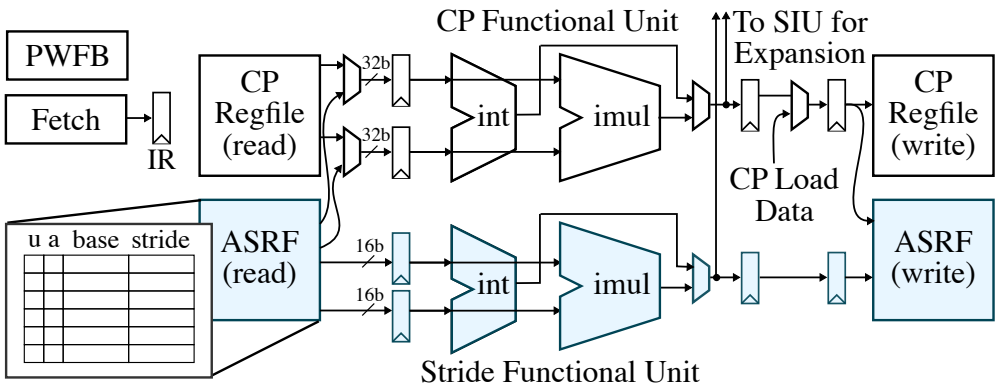


Figure 3.3: FG-SIMT CP with Compact Affine Execution – The ASRF and stride functional units are added to the baseline CP.

system guarantees that memory requests sent on a single port will be serviced in-order, there is no such guarantee *across ports*. The SMROQ also tells the SMU how and from where to write responses back. In other words, the type of request (i.e., coalesced or scalar) of the top entry of the SMROQ determines whether the next response to write back to the SLDQs should be from the wide port or the eight narrow ports. Entries from the SMROQ are removed when all responses corresponding to the entry have been written to the SMLQs. It is important to note that stores are not pushed into the SMROQ. This is because it is not necessary to wait for all store acknowledgements to return as they do not write back to the SRF.

The SMU also includes an in-flight counter of all memory requests, both loads and stores, which is used to implement a memory fence. An implicit memory fence is enforced after every `launch_kernel` instruction so that all pending loads and stores sent by the SMU are committed before the CP can issue any memory requests. The CP sends the implicit memory fence to the SIU and waits for a signal from the SIU that the in-flight counter is zero.

FG-SIMT L1 Memory System – The L1 memory system includes a single 16KB instruction cache along with an eight-bank 128KB data cache. Both caches are blocking, direct mapped, and use 32B cache lines. Queued memory request/response crossbars use round-robin arbitration and handle per-port response reordering internally. The data cache request crossbar supports nine narrow request ports with a data size of 32b and one wide request port with a data size of 256b. Each data cache bank is capable of servicing one 32b or one 256b read/write per cycle.

FG-SIMT Compact Affine Execution – Figure 3.3 illustrates the required modifications to support compact affine execution in FG-SIMT. A per-warp affine SIMT register file (ASRF) is used to track value structure. The ASRF compactly stores affine values as a 32b base and 16b

stride pair. The ASRF has one entry for each architectural register along with a tag indicating whether that register is uniform, affine, or generic. Explicit uniform tags enables more efficiently handling affine arithmetic that is only valid for such values. The ASRF can store both integer and floating-point values, but the latter is limited to uniform values as expanding a floating-point affine value would result in an unacceptable loss of precision. There are three ways in which a register can be tagged as uniform or affine: (1) destination of shared loads (e.g., loads with gp as base address), (2) destination of instructions reading the thread index (e.g., `tidx` instruction), or (3) destination of affine arithmetic.

Affine arithmetic is valid for select instructions with two uniform/affine operands. Affine arithmetic executes by reading base/stride operands from the ASRF, running the computation on the affine functional units, and (potentially) writing the result back into the ASRF. The CP's standard functional unit is used for base computations, and an extra function unit is added for stride computations. Affine arithmetic can improve both performance and energy efficiency by reducing the chime to one and avoiding redundant computations.

Affine branches are branches where the operands used in the branch comparison are affine. When the branch operand is uniform, a single comparison using the affine functional unit in the CP sufficiently applies to all threads in a warp. Uniform branches decrease the branch resolution latency by eliminating communication with the SIMT lanes. Energy efficiency is improved by avoiding reading the SRF and redundant comparisons across all threads in a warp.

Affine memory operations are load instructions with an affine base address. Affine memory operations are still sent to the SIU but are allowed to skip address generation on the SIMT lanes. Instead, affine memory operations are issued to the SMU immediately, since the access pattern is known from the affine base address. Like coalesced memory requests, affine memory operations use a wide request port. A full-warp affine memory operation is broken down into multiple cache-line width memory requests. Affine memory operations improve performance by enabling coalescing across the entire warp. More importantly, affine memory operations avoid the energy required to redundantly compute 32 effective addresses and then dynamically detect that these are consecutive addresses.

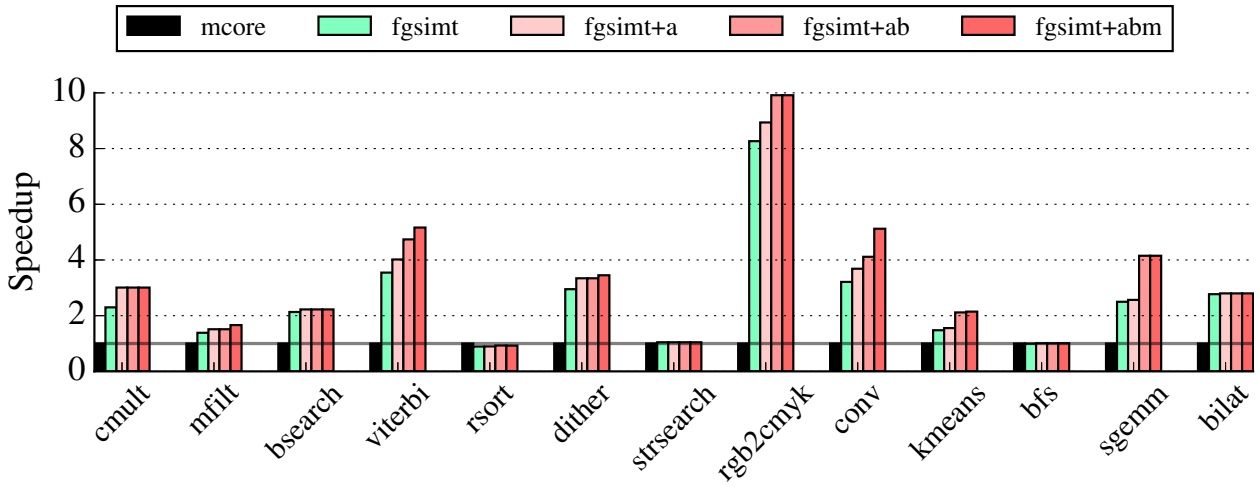


Figure 3.4: FG-SIMT Cycle-Level Performance – Speedup of FG-SIMT with various levels of compact affine execution normalized to a baseline 8-core system.

3.4 FG-SIMT Evaluation

The evaluation examines the performance of the following FG-SIMT designs at the RTL level: baseline FG-SIMT (*fgsimt*), FG-SIMT with a single-ASRF divergence scheme and affine arithmetic (*fgsimt+a*), affine arithmetic and branches (*fgsimt+ab*), and affine arithmetic, branches, and memory operations (*fgsimt+abm*). The results are normalized to the performance of a baseline eight-core system with in-order, single-issue cores, private L1 instruction caches, and a shared eight-bank L1 data cache (*mcore*). Cycle time, area, and energy estimates were obtained by pushing the RTL implementations of the baseline and the FG-SIMT microarchitecture through a commercial ASIC CAD toolflow composed of Synopsys DesignCompiler, IC Compiler, and PrimeTime PX using a TSMC 40 nm standard cell library. A total of 13 application kernels from an in-house benchmark suite were ported to FG-SIMT.

3.4.1 Performance Analysis

Figure 3.4 shows the performance of FG-SIMT with various forms of compact affine execution enabled (i.e., *fgsimt*, *fgsimt+a*, *fgsimt+ab*, *fgsimt+abm*) compared to the baseline eight-core system (i.e., *mcore*). In general, FG-SIMT is able to achieve 2–8× speedup on application kernels with regular loop-task parallelism. Compact affine execution further improves these speedups in several cases. Affine arithmetic benefits application kernels that are more compute-bound with a high density of affine instructions, which increases the amount of work that is amortized across an entire

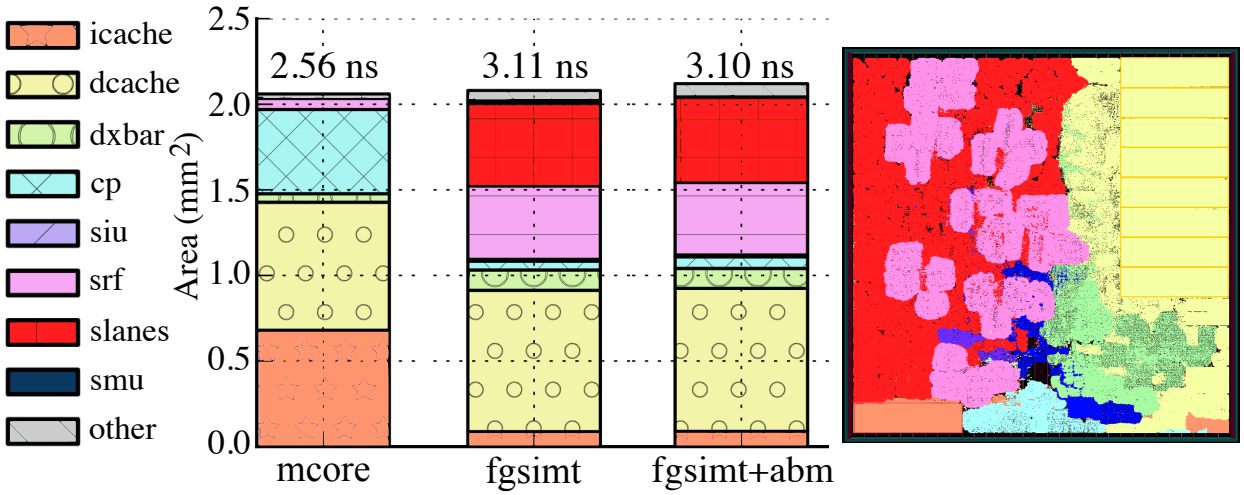


Figure 3.5: FG-SIMT Area Breakdown – (left) absolute area breakdowns of *mcore*, *fgsimt*, and *fgsimt+abm*. Bars are annotated with cycle times; (right) layout of placed-and-routed *fgsimt+abm* implementation in a TSMC 40nm process.

warp by executing an instruction affinely on the CP. Energy efficiency improves since operands are read once from the ASRF instead of having every active thread read from the SRF. Affine branches avoid the expensive branch resolution latency of the SIMT lanes which FG-SIMT cannot hide with multithreading, attributing to the more pronounced performance increase of *fgsimt+ab*. Affine memory operations reduce the total number of memory requests and reduce bank conflicts. The results suggest that affine memory operations can further improve the efficiency gained from coalescing. Overall, exploiting value structure using compact affine execution in *fgsimt+abm* improves performance over *fgsimt* for a majority of the application kernels, with speedups ranging from 20–65% for those with significant improvement. Unfortunately, all FG-SIMT configurations yield minimal or no improvement compared to the baseline for application kernels with more irregular loop-task parallelism (e.g., average number of active threads 13.2 and 17.1, for *strsearch* and *bfs*, respectively). This is because warp fragments created due to control-flow divergence must be serialized, negating the benefits of lock-step execution of threads on the FG-SIMT engine.

3.4.2 Area and Cycle Time Analysis

Figure 3.5 shows the area breakdown of the placed-and-routed baseline and FG-SIMT designs. Overall, the total areas of all designs only vary slightly, with 2.06 mm^2, 2.08 mm^2, and 2.12 mm^2 for *mcore*, *fgsimt*, and *fgsimt+abm*, respectively. The area due to the instruction cache is dramatically

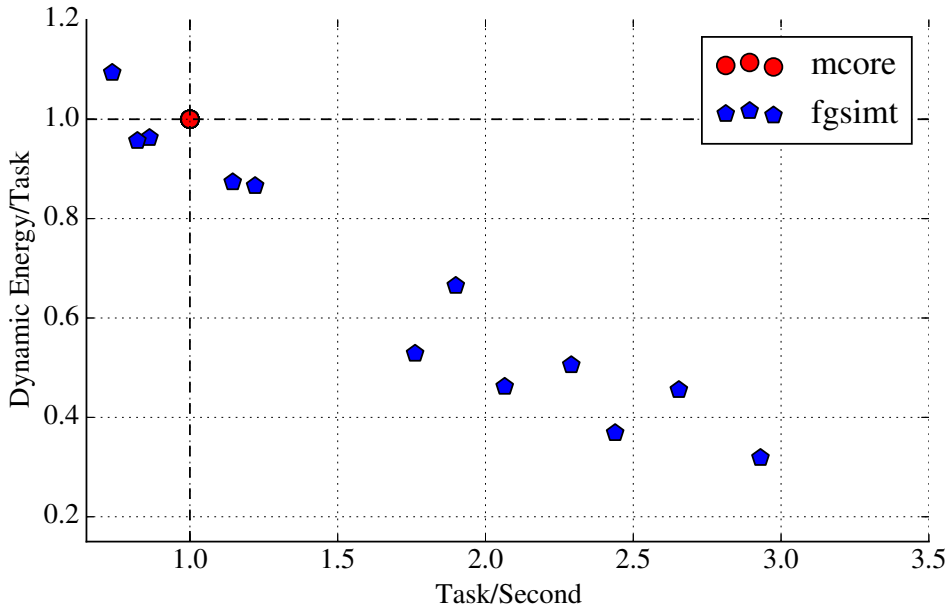


Figure 3.6: FG-SIMT Energy vs. Performance – Dynamic energy and performance of *fgsimt* normalized to *mcore*, which is annotated on (1,1). Each point represents the relative energy and performance of a different application kernel on FG-SIMT architectures.

reduced since *fgsimt* only has one core as opposed to the 8 cores in *mcore*. However, *fgsimt* pays a significant overhead for the larger, highly ported SRFs. The additional wide port for memory coalescing and affine memory operations also adds area overhead to the crossbar. The ASRF, stride functional unit, and expansion logic of *fgsimt+abm* slightly increases the area of the CP and SIMT lanes.

The critical path of *mcore* is through the memory system. The critical path of both the FG-SIMT designs are through the SRF read and address generation. The marginal difference in cycle time of *fgsimt+abm* and *fgsimt* is within the variability margin of the ASIC CAD tools.

3.4.3 Energy Analysis

Figure 3.6 shows the dynamic energy and performance of *fgsimt* relative to *mcore* for all application kernels in this study. *fgsimt* yields significant reductions in energy on regular application kernels, with maximum reductions of up to 70%. This is primarily due to the amortization of instruction fetch and frontend accesses across multiple threads. Although not shown, compact affine execution further reduces energy by eliminating redundant work to complete the same task.

On the other hand, *fgsimt* is only able to achieve energy reductions of up to 15% on irregular application kernels, and in rare cases, may actually increase energy consumed per task. This is because control-flow irregularity limits the number of threads across which instruction fetch and frontend accesses can be amortized. If the average number of active threads is very low, then the increased energy from more expensive SRF accesses and mechanisms for SIMT execution will outweigh the benefits of this amortization.

3.5 FG-SIMT Summary

Although FG-SIMT succeeds in addressing the 3P's to a limited extent, the minimal innovation in software and strong focus on exploiting regular loop-task parallelism prevents it from being a true 3P platform. FG-SIMT addresses productivity by using threads as both the inter-core and intra-core abstraction and requires relatively sparse knowledge of the microarchitecture to yield high performance compared to GPGPUs (e.g., no texture/shared memory, global barriers). However, the programming model is still *thread-based* and does not profer the benefits of more modern *task-based* programming models. In addition, it is still difficult to efficiently map irregular applications to FG-SIMT. Furthermore, a primary weakness of FG-SIMT is that it is designed and evaluated in a intra-core context and does not utilize runtimes for work distribution across multiple cores. As such, it is unclear if threads are the best parallel abstraction for exploiting loop-task parallelism. The use of a single ISA for both the GPP and SIMT engine served as a promising step towards addressing portability, but the ability to execute the same binary on either architecture was not completely fleshed out. In retrospect, the challenges of portability were only fully revealed in the context of a multi-core system running a software runtime. The results certainly provide evidence for the strengths of FG-SIMT as an accelerator for regular loop-task parallelism, but it is clear that it is inherently ill-suited for more irregular loop-task parallelism. The novel mechanisms for exploiting value structure are also very effective at improving performance and energy efficiency in FG-SIMT for regular applications, but quickly lose their effectiveness in the presence of irregular control-flow and memory-access patterns.

Overall, FG-SIMT highlights the need for innovations in both software and hardware. Examining multi-core systems with work-stealing runtimes provides a different context in which to address the 3P's and motivates the use of tasks, or loop-tasks, as a more natural parallel abstraction

for efficiently exploiting loop-task parallelism across cores and within a core. It is also clear that the hardware needs to be fundamentally re-designed to enable acceleration of both regular and irregular applications. Regardless of its flaws, FG-SIMT is capable of impressive performance and energy efficiency for reasonable area overheads for regular applications, and the insights gained from this work are invaluable in guiding the software/hardware co-design approach of the LTA platform that follows in the next chapter.

CHAPTER 4
LOOP-TASK ACCELERATOR SOFTWARE

The observations in Chapters 2 and 3 suggest that a software/hardware co-design may offer the best chance at achieving a true 3P platform. To this end, the scope of the software is narrowed, the scope of the hardware is broadened, and these layers are married by exposing *loop-tasks* as a common parallel abstraction across the computing stack.

One observation from other approaches to 3P platforms as well as my own previous work is that narrowing the scope of the parallel abstraction in software enables a more efficient encoding of exploitable parallelism to the hardware. The LTA platform focuses on exploiting *loop-task parallelism*, a popular form of task parallelism that is exemplified by the `parallel_for` construct. Loop-task parallelism includes traditional data parallelism with regular control-flow and memory-access patterns across loop iterations, but also includes loop iterations that may have significant variability to data-dependent control flow and irregular memory accesses.

This chapter describes the software component of the LTA platform, which is composed of a productive programming API for exposing loop-tasks, a task-based work-stealing runtime for executing loop-tasks across cores, and ISA extensions for explicitly encoding loop-tasks so that they can be seamlessly executed on either GPPs or LTA engines. Figure 4.1 sketches the LTA application development flow. Development would still begin with the single-threaded scalar implementation of the application on the CMP. The LTA software framework can be used to implement the multi-threaded implementation on the CMP, and this same implementation can be used without modification on a variety of LTA engines, while still providing high performance on both regular and irregular loop-task parallel applications.

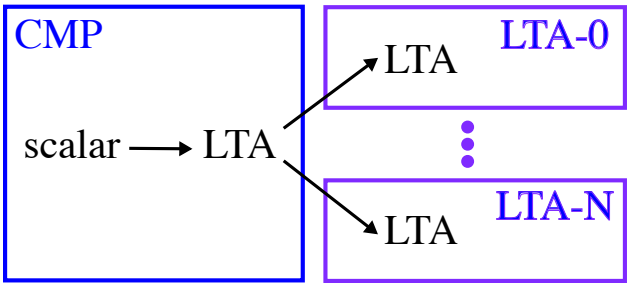


Figure 4.1: LTA Application Development Flow – Application development still begins with a single-threaded scalar implementation on the CMP, but the same multi-threaded implementation using the LTA software framework can be mapped to the CMP or a variety of LTA engines.

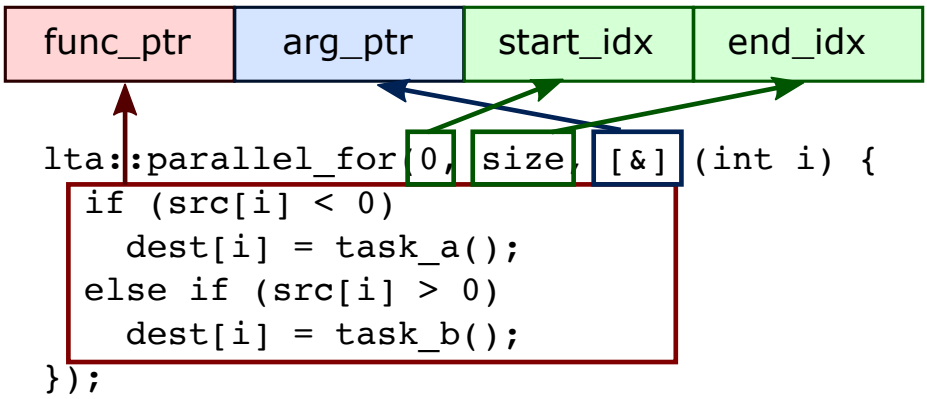


Figure 4.2: Anatomy of a Loop-Task – A loop-task is a four-tuple composed of the pointer to the loop function, the pointer to the arguments of the function, and the start/end indices of the loop iterations on which to apply the function. The example `parallel_for` is encoded into a loop-task as shown. The argument pointer references a container of the arguments captured by the lambda closure.

4.1 Programming API

In order to replicate the productivity of TBB that was observed in Chapter 2, the `parallel_for` construct is used to express loop-tasks that can be exploited both across cores *and* within a core, without the need for inter-core-specific optimizations. Figure 4.2 illustrates the anatomy of a loop-task, which can be defined as a function that is applied to a range of loop iterations. More specifically, a loop-task is a four-tuple of a function pointer, an argument pointer, and the start/end indices of the range.

Figure 4.3 shows different variants of the `parallel_for` in the LTA programming API. Figure 4.3(a) shows the lambda-based variant and Figure 4.3(b) show the macro-based variant. Notice how the loop function has data-dependent branches that can fundamentally change the nature of the computation. Currently only the macro-based `parallel_for` is available because the LTA cross-compiler does not yet support C++11 lambdas. Figure 4.3(c) shows the special *loop-task function* that is generated by the macro. All loop-task functions have a specific signature that corresponds to the fields of the four-tuple described above. There is also an additional argument called the *range step value* (i.e., `step`) that is hidden from the application-level programmer. The range step value can be configured to change the stride of the loop iterations that are traversed for a given loop-task (see Chapter 5 for details).

```
void kernel(int* dest, int* src, int size)
{
  lta::parallel_for(0, size, [&] (int i) {
    if (src[i] < 0)
      dest[i] = task_a();
    else if (src[i] > 0)
      dest[i] = task_b();
  });
}
```

(a) Lambda-Based `parallel_for`

```
void kernel(int* dest, int* src, int size)
{
  LTA_PARALLEL_FOR(0, size, (dest,src), ({
    if (src[i] < 0)
      dest[i] = task_a();
    else if (src[i] > 0)
      dest[i] = task_b();
  }));
}
```

(b) Macro-Based `parallel_for`

```
void lta_function(void* a, int start, int end, int step=1)
{
  args_t* args = static_cast<args_t*>(a);
  int* dest = args->dest;
  int* src0 = args->src0;
  int* src1 = args->src1;
  for (int i = start; i < end; i += step) {
    if (src[i] < 0)
      dest[i] = task_a();
    else if (src[i] > 0)
      dest[i] = task_b();
  }
}
```

(c) Loop-Task Function Generated by Macro-Based Variant

```
void lta_function(void* a, int start, int end, int step=1)
{
  LambdaFunctor& func = static_cast<LambdaFunctor&>(*a);
  for (int i = start; i < end; i += step)
    func(i);
}
```

(d) Loop-Task Function Generated by Lambda-Based Variant

Figure 4.3: LTA Programming API – A `parallel_for` construct is used to express loop-tasks that can be exploited across cores and within a core. Both lambda-based and macro-based variants are shown, but currently only the latter is available since the LTA cross-compiler does not yet support C++11 lambdas. The macro uses preprocessor logic to generate a special loop-task function. All loop-task functions have a specific signature containing a pointer to the arguments to the loop function, the start and end indices of the loop iterations to which to apply the function, as well as a range step value that is hidden to the application-level programmer.

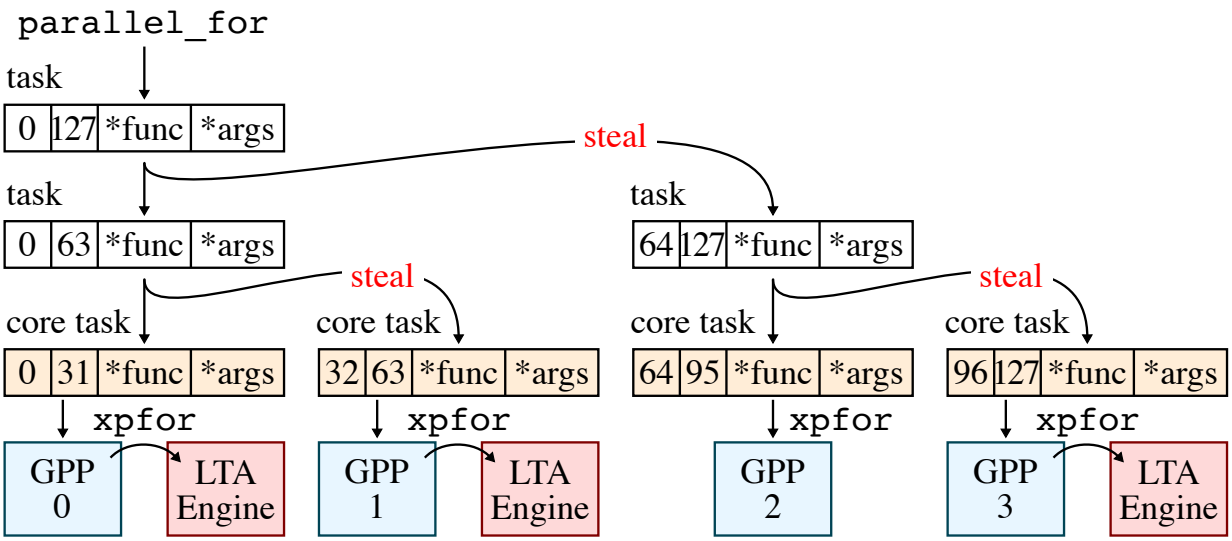


Figure 4.4: Example LTA Runtime Task Partitioning – LTA runtime partitions tasks into *core tasks* which are distributed across cores to exploit loop-task parallelism. Core tasks are executed using the `xpfor` instruction that acts as an indirect function call to a loop-task function. If available, an LTA engine can further exploit loop-task parallelism within a core.

Unlike the macro-based variant of the `parallel_for`, the lambda-based variant will require additional modifications to the cross-compiler, aside from the minor changes to the assembler required for both variants (see Section 4.3). Future work includes exploring the compiler changes required to implement the lambda-based variant.

4.2 Task-Based Runtime

The LTA runtime is a task-based work-stealing runtime inspired by TBB and is responsible for generating, partitioning, and distributing loop-tasks exposed by the LTA programming API across the cores available on the system. It employs child-stealing, Chase-Lev task queues [CL05], and occupancy-based victim selection [CM08]. Figure 4.4 illustrates how a work-stealing runtime recursively partitions loop-tasks into subtasks to facilitate load balancing.

The master core, GPP 0, begins executing the application binary while all other cores spin in the runtime's work-stealing routine. When GPP 0 encounters a `parallel_for`, it generates the initial loop-task by instantiating a loop-task object with the corresponding function pointer, argument pointer, and start/end indices of the range $\langle 0, 127 \rangle$. Then GPP 0 partitions this loop-task by splitting the range in half, generating two loop-tasks with smaller ranges: $\langle 0, 63 \rangle$ and

$\langle 64, 127 \rangle$. GPP 0 pushes $\langle 64, 127 \rangle$ onto its task queue in memory, and continues to recursively partition $\langle 0, 63 \rangle$ into $\langle 0, 31 \rangle$ and $\langle 32, 63 \rangle$, and then executes the former while pushing the latter onto its task queue. Meanwhile, the work-stealing routine on GPP 2 sees the tasks in GPP 0's task queue and steals $\langle 64, 127 \rangle$. GPP 2 then partitions this task into $\langle 64, 95 \rangle$ and $\langle 96, 127 \rangle$, executing the former. GPP 1 steals/executes $\langle 32, 63 \rangle$ and GPP 3 steals/executes $\langle 96, 127 \rangle$. As in traditional work-stealing runtimes, tasks are always stolen in FIFO-fashion (i.e., from the top of the partition tree) to improve locality within a given core [FLR98].

Tasks are partitioned until the range is less than or equal to a configurable *core task size* at which point the loop-task is called a *core task*, which acts as the smallest unit of load balancing across cores. The LTA runtime uses a default core task size of $N/(k \times P)$, where N is the size of the initial range, k is a scaling factor, and P is the number of cores. Increasing k generates more core tasks with smaller ranges (better load balancing, higher overhead), whereas decreasing k generates less core tasks with larger ranges (worse load balancing, lower overhead). Sensitivity studies on the LTA engine configurations and application kernels explored in this thesis indicate that $k = 4$ is a reasonable design point for avoiding starvation due to a lack of core tasks.

One of the key differences between a traditional work-stealing runtime and the LTA runtime is how the runtimes actually execute core tasks. A traditional runtime simply uses an indirect function call (i.e., `jalr`) on the core task's function pointer with the given argument pointer and start/end indices. However, the LTA runtime uses a special instruction that allows a core task to be executed on an LTA engine, if one is available. At a high level, LTA engines are able to accelerate loop-task execution by further partitioning core tasks into *micro-tasks* (μtasks) and mapping these μtasks onto *micro-threads* (μthreads).

The details of the ISA extensions and the mapping of μtasks to μthreads are discussed in the next section, but the important point with respect to the LTA runtime is that an *LTA-aware* task partitioning scheme allows for more efficient execution of loop-tasks. Figure 4.5 shows the difference between a default task partitioning scheme and an LTA-aware task partitioning scheme. Each number in the abstract task partitioning tree indicates the size of the range of the loop-task at that point in the tree. For example, the initial range in the example contains 42 loop iterations. Assuming that the core task size is four, the default scheme recursively splits the range in half until there is a total of 16 core tasks. Further assuming that there are four cores in the system, each with a four-μthread LTA engine, it will take four steps to execute all core tasks. The execution time of

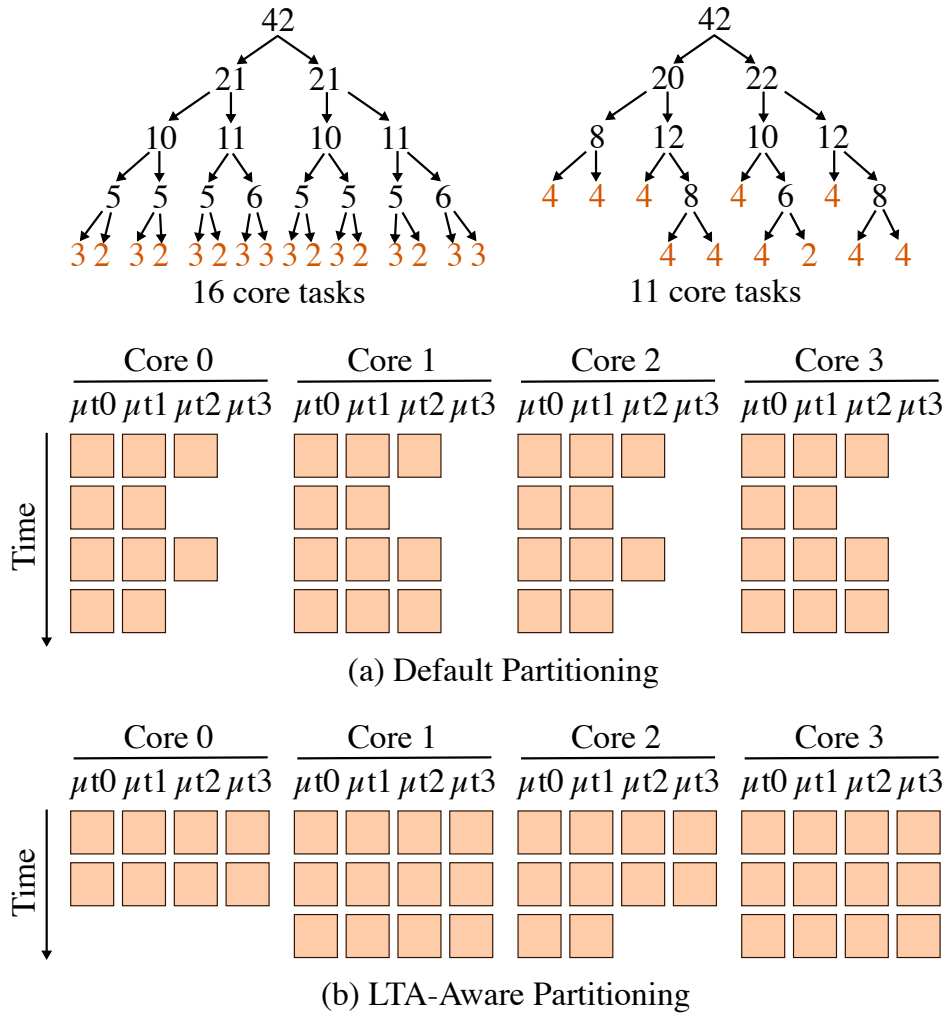


Figure 4.5: LTA-Aware Task Partitioning – The abstract task partitioning tree and the corresponding execution diagram for the default scheme and the LTA-aware scheme are shown. Each number in the tree indicates the size of the range of the loop-task at that point in the tree. The initial range in both cases has 42 loop iterations. The default scheme partitions loop-tasks by splitting the range in half, whereas the LTA-aware scheme partitions loop-tasks by ensuring at least one sub-range is a multiple of the number of μthreads available on the LTA engine. The execution diagrams show how the generated core tasks are executed on a four-core system with four-μthread LTA engines. Each core task is executed across all available μthreads. A block represents one loop-iteration-worth of work and, in this example, takes roughly the same amount of time to execute.

all core tasks is roughly the same because all μthreads must execute in lock-step in this example. Note that because core tasks have ranges that are less than the total number of μthreads available on the LTA engine, some μthreads will remain idle during core task execution. On the other hand, the LTA-aware scheme splits the range such that at least one of the resulting sub-ranges is a *multiple of*

Instruction	Description
`xpfor r_s`	Indirect function call that acts as a hint to accelerate core task execution on LTA engine. Same semantics as `jalr`.
`mtuts r_d, r_s`	Moves values from GPP register to LTA engine register for all μthreads.
`xplock r_d, r_s, r_t`	Attempts to obtain binary lock with special success/failure tokens. Same semantics as `amo.xchg`.
`xpsync`	μthread barrier; forces synchronization of μthreads.

Table 4.1: LTA Instruction-Set Architecture Extensions

the number of μthreads available in a single LTA engine. Specifically, the LTA-aware scheme sets the core task size to $\lceil (N \times t)/(k \times P) \rceil \times 2t$, where t is the total number of μthreads available on the LTA engine. This partitioning generates a total of 11 core tasks, most of which have a range of four loop iterations. Executing all of these 11 core tasks will only take three steps. Note that while LTA-aware task partitioning can increase the maximum difference in size between any two core tasks compared to traditional task partitioning, LTA-aware task partitioning generally improves performance on systems with LTA engines by increasing μthread utilization. The runtime determines the total number of μthreads available on the LTA engine by reading a special coprocessor register on the GPP.

4.3 Instruction-Set Architecture

Table 4.1 outlines the ISA extensions required by the LTA platform. The LTA platform extends a RISC-V-like [WLPA16] ISA with a new `xpfor` instruction that enables exploiting loop-task parallelism within a core if there is an LTA engine attached to the GPP. The `xpfor` instruction is used to execute core tasks and acts as an indirect function call with the same semantics as a `jalr`, except that an `xpfor` can only be used to call a loop-task function pointer with the loop-task function signature in Figure 4.3(c). The calling convention of `xpfor` is crafted to mirror the arguments of the loop-task function: argument pointer in `a0`, range start index in `a1`, and range end index in `a2`. More importantly, an `xpfor` acts as a hint to the hardware that it can potentially accelerate core task execution on an LTA engine.

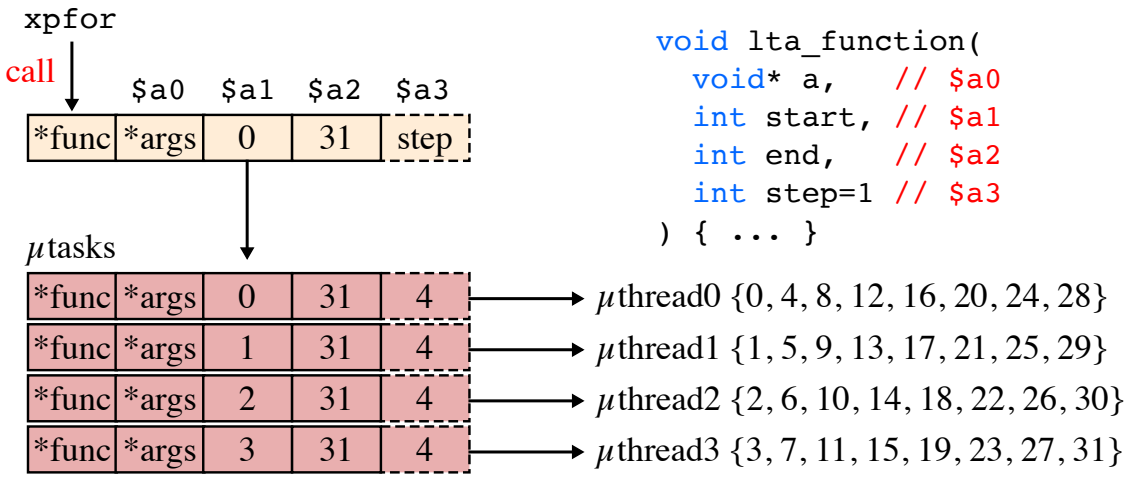


Figure 4.6: Example μtask to μthread Mapping – The `xpfor` instruction acts as a hint to the hardware that the core task can be accelerated if an LTA engine is available. LTA engines accelerate core task execution by further partitioning core tasks into μtasks, then mapping these μtasks onto μthreads. Each μthread is annotated with the loop iterations in the range of its assigned μtask. The hardware is responsible for initializing the architectural state with the arguments of the loop-task function. The range step value is used to configure the stride of the loop iterations that are traversed within a μtask.

If an LTA engine is not available, the `xpfor` can be treated as a standard `jalr`. Serial execution of core tasks are always valid. If an LTA engine *is* available, the `xpfor` sends the core task from the GPP to the LTA engine to be accelerated. The LTA engine accelerates core task execution by further partitioning the core task into μtasks which can be executed concurrently in any order by the LTA using μthreads. The runtime can execute core tasks the same way regardless of LTA engine availability. A single implementation of an application can be used on any architecture that implements `xpfor`, greatly improving portability.

Figure 4.6 illustrates an example of how a core task is partitioned into μtasks and how these μtasks are mapped onto μthreads. The core task containing the range $\langle 0, 31 \rangle$ is partitioned into 4 μtasks containing subsets of this range. The start indices and the range step values are configured by the hardware such that consecutive loop iterations are mapped across consecutive μthreads. Note that the partitioning and mapping of μtasks can change depending on the type of LTA engine available.

Another extension to the LTA ISA is the `mtuts` instruction that allows transferring of data in architectural registers between the GPP and the LTA engine. For instance, this is how the runtime initializes the global pointer (`gp`) and stack pointer (`sp`) on the LTA engine. With the exception

of this initialization, all communication between the GPP and the LTA engine occurs through memory. `xpfor` semantics dictate arguments are caller-saved and only callee-saved architectural registers are guaranteed to be valid after a call.

The LTA ISA also includes two other instructions that are meant to be used inside of a core task: `xplock` and `xpsync`. `xplock` is an alternative to the `amo.xchg` instruction that is used for obtaining a binary lock with special tokens for success and failure. This instruction helps the LTA engine to avoid deadlocks due to collisions between multiple μthreads on a given LTA engine causing continued failures. `xpsync` is a barrier that forces synchronization of μthreads. This instruction is used to prevent further control divergence by halting forward progress until all μthreads have reached the same instruction. Although not necessarily for high performance except in rare cases, `xpsync` can be manually inserted by the application-level programmer to explicitly guide μthread behavior. The vision is to eventually have the compiler generate these instructions automatically.

CHAPTER 5
LOOP-TASK ACCELERATOR HARDWARE

The observations in Chapter 2 suggest that existing accelerator-based 3P platforms mainly focus on exploiting regular loop-task parallelism (e.g., MICs, GPGPUs). Although achieving high performance on irregular applications using such accelerators is certainly possible, it often requires extensive manual, target-specific optimizations that degrade productivity and portability, and the resulting performance may not always be proportional to the resources available in hardware. As such, the LTA platform pursues a clean-slate design of a hardware accelerator capable of exploiting a broader range of loop-task parallelism, without sacrificing productivity and portability. By pushing the loop-task abstraction down to hardware, the same core tasks can be used to efficiently exploit loop-task parallelism across cores and within a core, without limiting the multiplicative effect in performance.

This chapter describes the hardware component of the LTA platform: the microarchitectural template for the LTA engine. The LTA engine is designed to accelerate the loop tasks encoded by the `xpfor` instruction within a core. A task-coupling taxonomy that describes how tasks can be coupled in both space and time is also detailed in this chapter. LTA engines can be configured at design time to have variable spatial and temporal task coupling to target both regular and irregular loop tasks.

5.1 Task-Coupling Taxonomy

As mentioned in Section 4.3, an LTA engine accelerates loop-task execution by further partitioning core tasks into µtasks which are then mapped to µthreads. There is a large design space in how these µthreads are executed.

Figure 5.1(a) illustrates one approach for an eight-µthread LTA engine that **tightly couples µthread execution in both space and time**. The task-management unit (TMU) acts as the primary interface between the GPP and the LTA engine. When the TMU receives information about a core task from the GPP, it partitions this core task into eight µtasks and assigns one µtask per µthread. Each µthread has its own register file context which is initialized with the arguments corresponding to the loop-task function in Figure 4.3(c) before execution. All µthreads share a common frontend (F), and instruction/data memory accesses are managed by the instruction management unit (IMU)

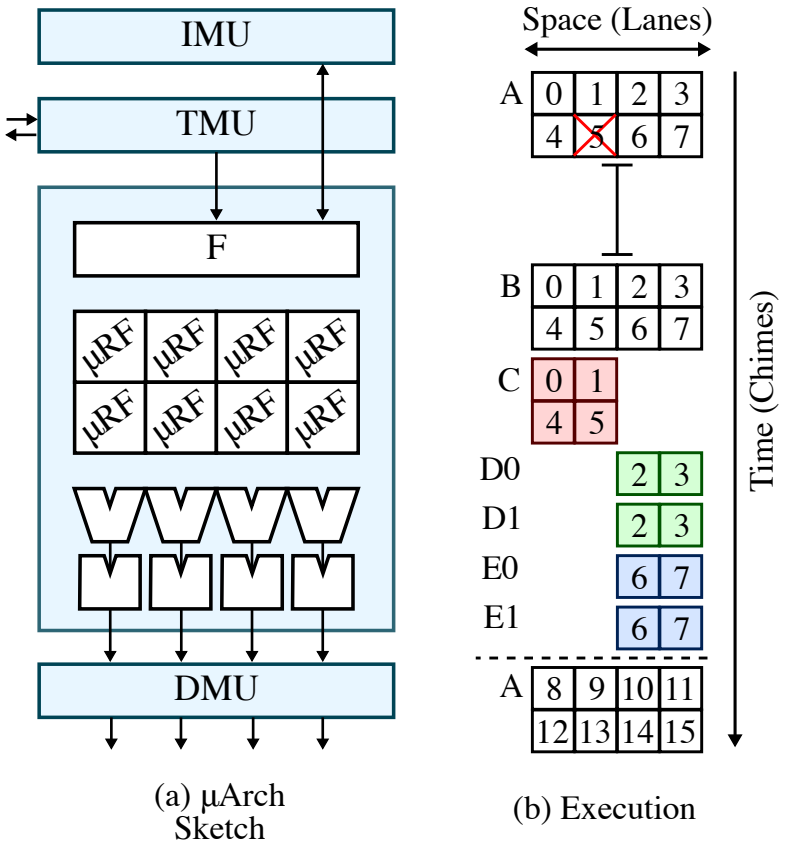


Figure 5.1: LTA Engines with Tight Task Coupling – 8-μthread LTA engine with 4 lanes and 2 chimes. μthreads are tightly coupled in both space and time, meaning they must execute in lock-step. IMU = instr mgmt unit; TMU = task mgmt unit; DMU = data mgmt unit; F = fetch/dispatch unit; μRF = μthread regfile. The execution diagram of an example kernel is shown on the right. Each box represents a unique μthread which is annotated with the loop iteration being executed. The letters indicate the instruction being executed at each timestep. μthreads that share the same control flow on divergence are shown with the same color. Stalls due to microarchitectural latencies are marked by a red *X*.

and the data management unit (DMU). Using terminology from traditional vector processors, the μthreads in this example are organized into four *lanes* and two *chimes*. Each lane has a dedicated set of short-latency and long-latency functional units.

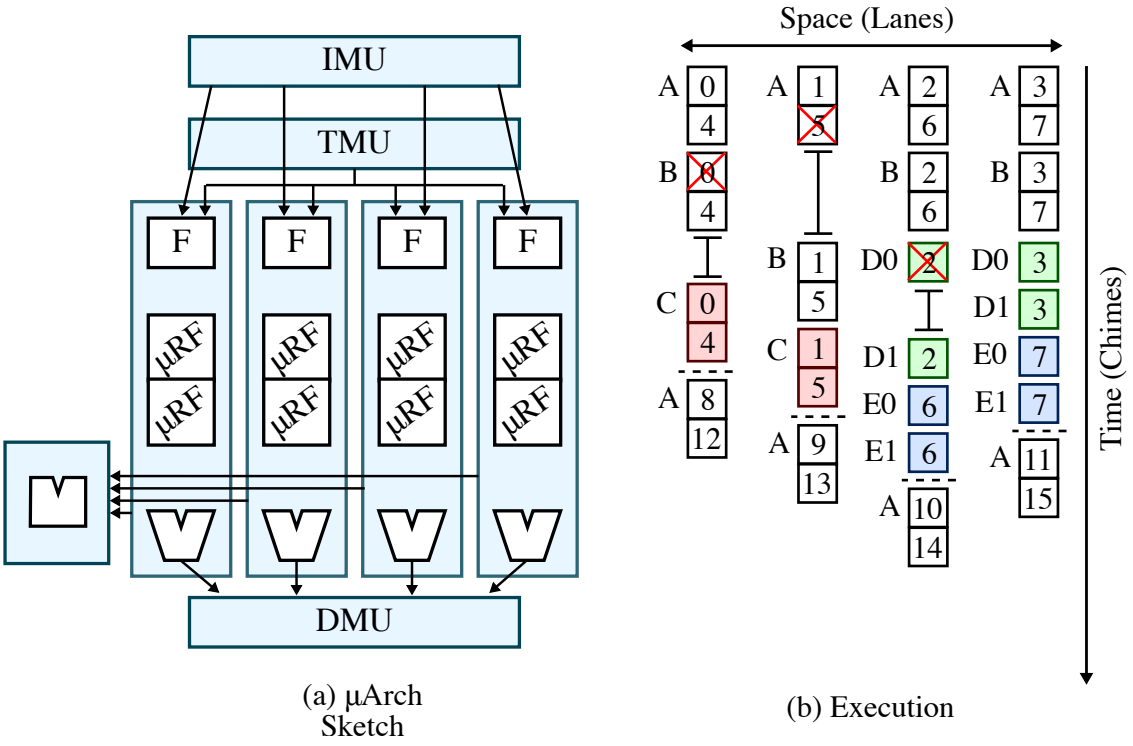
Figure 5.1(b) details how the core task mapped to GPP 0 in Figure 4.4 might execute on this tightly coupled LTA engine. All eight μthreads execute in lock-step in space across the four lanes and in time across the two chimes. Notice that μthreads on neighboring lanes execute consecutive iterations. To enable this, the LTA engine initializes each μthread's range step value in the `a3` register to eight so that μT 0 executes iterations 0, 8, 16, and 24. Tightly coupled execution enables the LTA engine to exploit control and memory structure by amortizing frontend and memory accesses

across multiple μthreads. The frontend can fetch, decode, dispatch, and issue a single instruction before executing it across all eight μthreads, and memory addresses that are consecutive across the μthreads can be coalesced into a single wide access. Unfortunately, tightly coupled execution means if one μthread stalls due to a RAW dependency or cache miss then all μthreads must stall. Furthermore, divergent control flow due to conditional branches requires serialized execution of μthreads. In this example, the red μthreads executing loop iterations 0, 1, 4, and 5 are executed first, then the green μthreads executing loop iterations 2 and 3, followed by the blue μthreads executing loop iterations 6 and 7. Due to this, tightly coupled execution is best suited for regular loop-task parallelism.

Figure 5.2(a) illustrates a different approach for an eight-μthread LTA engine that **loosely couples μthread execution in space while keeping μthread execution in time tightly coupled**. There are still four lanes, but now each lane is *decoupled* in space with per-lane frontends, so that they are able to execute independent instruction streams. In addition, expensive resources such as the long-latency functional units and memory ports are shared between the lanes. As such, accessing these shared resources requires arbitration.

Figure 5.2(b) details how the same core task from the previous example might execute on an LTA engine with loosely coupled execution in space and tightly coupled execution in time. Loosely coupled execution enables the LTA engine to better tolerate irregular control-flow and memory-access patterns at the cost of less amortization of frontend and memory access across μthreads. In this example, μthreads within a lane can independently fetch, decode, dispatch, and issue instructions with respect to μthreads in other lanes. Notice how the stall when executing loop iteration 5 is now isolated to a single lane while the other lanes can continue execution. Similarly, μthreads that are diverged due to irregular control flow can be executed concurrently. The tradeoff with loosely coupled execution in space is that conflicts at the shared resources may cause additional stalls within a lane. μthreads that are tightly coupled in time within a lane must still execute in lock-step, meaning that a stall on one μthread can cause other μthreads mapped to the same lane to stall as well, as seen when executing loop iteration 0 on the first lane.

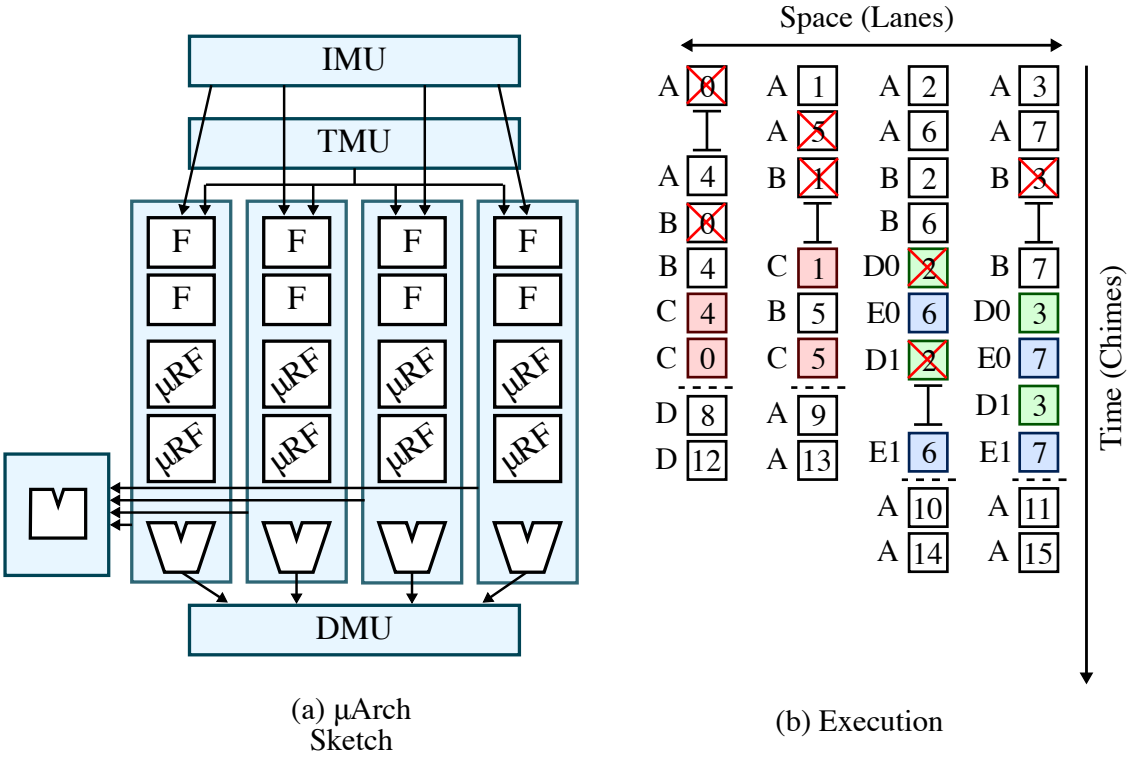
Figure 5.3(a) illustrates the next logical step of an eight-μthread LTA engine that **loosely couples μthread execution in both space and time**. In this example, not only are the lanes decoupled in space, but the chimes are decoupled in time as well. This means that each μthread within a lane also has its own frontend capable of sustaining an independent instruction stream.



(a) µArch Sketch

(b) Execution

Figure 5.2: LTA Engine with Loose Task Coupling in Space and Tight Task Coupling in Time – See Figure 5.1 for details.



(a) µArch Sketch

(b) Execution

Figure 5.3: LTA Engine with Loose Task Coupling in Space and Time – See Figure 5.1 for details.

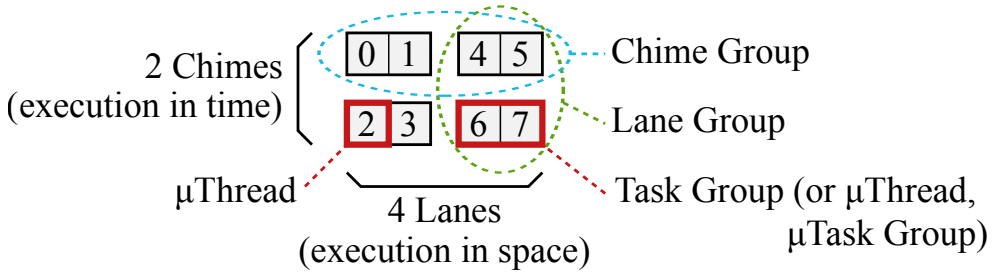


Figure 5.4: Terminology for Task-Coupling Taxonomy – Example of 8-µthread LTA engine with 4 lanes (l) and 2 chimes (c). 8 µthreads are divided into 4 task groups (g_t), also called µthread groups or µtask groups, which execute in lock-step in both space and time. The four lanes are partioned into two lane groups (g_l); the two chimes are partitioned into two chime groups (g_c). Configurations named as l/g_l x c/g_c, so this configuration is 4/2x2/2.

Figure 5.3(b) details how the core task might execute on an LTA engine with loosely coupled execution in space and time. Loosely coupled execution in time further improves the LTA engine's ability to tolerate irregular control-flow and memory-access patterns by allowing µthreads within the same lane to hide microarchitectural latencies with fine-grain vertical multithreading. For instance, the stall when executing loop iteration 0 on the first lane from the previous example is hidden by executing loop iteration 4 on the same lane. The tradeoff with loosely coupled execution in time is an increased pressure on the shared instruction memory port. This is a consequence of each instruction fetch keeping functional units busy for less cycles. These additional conflicts can introduce new stalls across the µthreads. Since all µthreads are loosely coupled, there is no amortization of frontend and memory accesses, which can negatively impact both area and energy efficiency.

The microarchitectures in Figure 5.1 and Figure 5.3 represent opposite extremes of how µthreads, or the µtasks that are mapped to µthreads, can be coupled in both space and time. However, there is a wide spectrum of *spatial and temporal task coupling* that can vary from loose to moderate to tight coupling. Figure 5.4 uses a more abstract diagram to illustrate an intermediate design point where the eight µthreads are divided into four *task groups* that execute in lock-step in both space and time. The task-coupling terminology defines groups of lanes that are tightly coupled as *lane groups*, and groups of chimes that are tightly coupled as *chime groups*. Lane groups are loosely coupled with respect to other lane groups, and chime groups are loosely coupled with respect to other chime groups. This example has two lane groups and two chime groups. A specific task-coupling configuration can then be defined as l/g_l x c/g_c, where l is the total number of lanes,

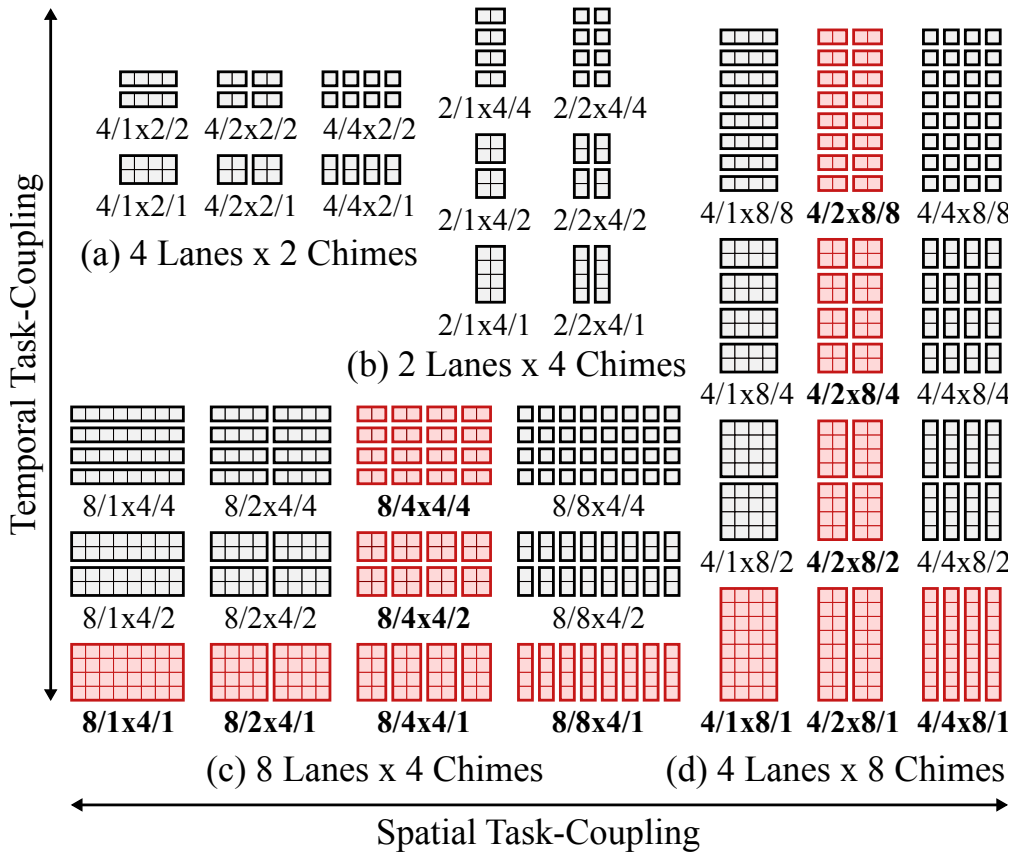


Figure 5.5: Task-Coupling Taxonomy – All possible spatial and temporal task-coupling configurations for: (a) four lanes, two chimes; (b) two lanes; four chimes; (c) eight lanes, four chimes; (d) four lanes, eight chimes. For given subfigure, most-coupled configuration is bottom left and least-coupled configuration is top right. Configurations used in Chapter 7 are highlighted.

g_l is the number of lane groups, c is the total number of chimes, and g_c is the number of chime groups. Furthermore, l x c yields the total number of µthreads, and g_l x g_c yields the number of task groups. The task-coupling configuration in this example can thus be denoted as 4/2x2/2.

Given this terminology, it is possible to describe all spatial and temporal task-coupling configurations for a given number of lanes and chimes using a *task-coupling taxonomy*. Figure 5.5(a) shows the six configurations for the 8-µthread LTA engine discussed above, and Figure 5.5(b) shows an alternative eight-µthread LTA engine with two lanes and four chimes. The task-coupling taxonomy can easily be extended to 32-µthread LTA engines with four lanes and eight lanes. Although LTA engines with 8–128 µthreads were explored, the primary evaluation of this thesis will focus on the 12 configurations of the 32-µthread LTA engines highlighted in Figures 5.5(c) and (d).

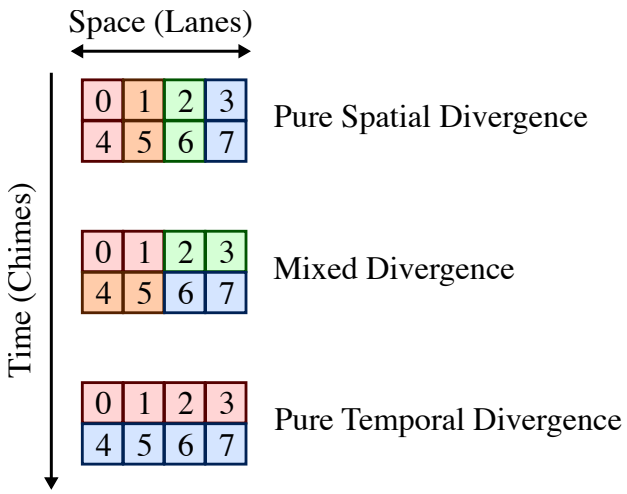


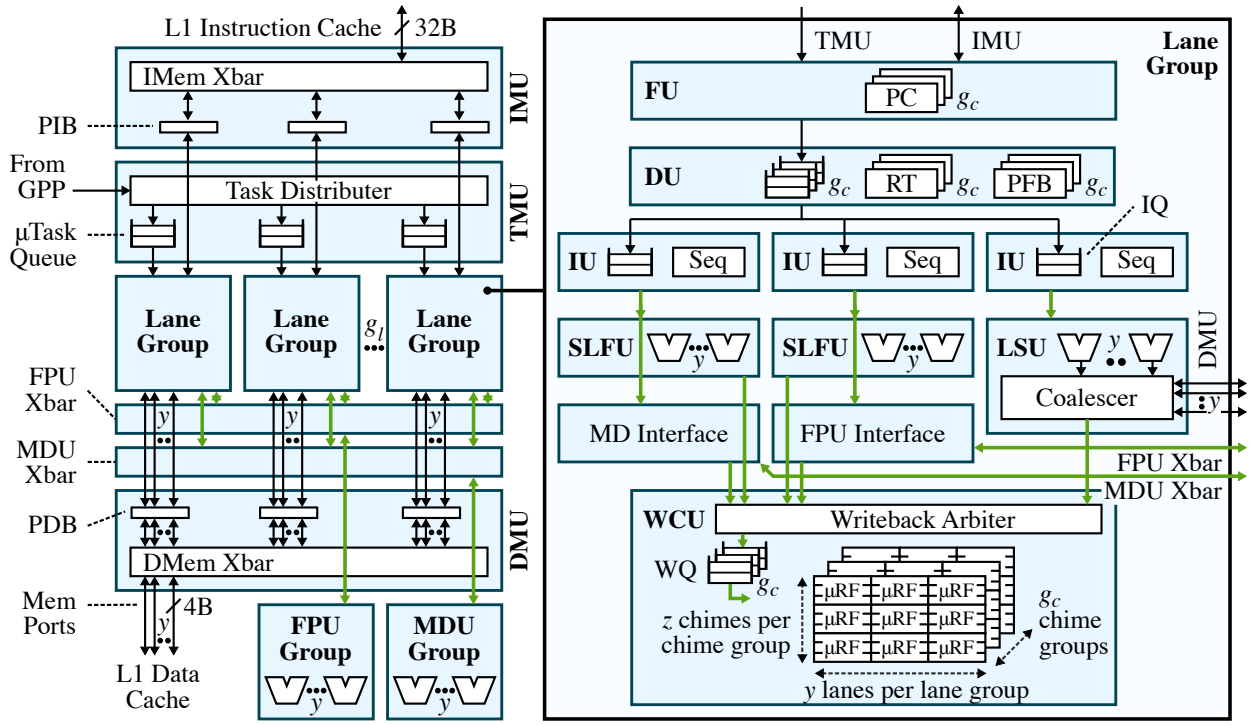
Figure 5.6: Types of Control Divergence – µthreads within a task groups can be diverged in space and/or time. Examples of spatial, temporal, and mixed divergence for 8 µthreads on a 4-lane 2-chime LTA engine are shown. Pure spatial divergence means all µthreads mapped to a given lane share the same control flow, whereas pure temporal divergence means all µthreads in the same chime share the same control flow.

Now that the task-coupling terminology and taxonomy has been established, the different types of control divergence that can occur within a task group can be described in more detail. Figure 5.6 shows how eight µthreads can be diverged in space and/or time. µthreads that share the same control flow are called *task group fragments* and are shown with the same color. When there is pure spatial divergence, all µthreads mapped to the same lane are in the same task group fragment. Loose spatial task coupling specifically helps tolerate spatial control divergence. When there is pure temporal divergence, all µthreads within a chime are in the same task group fragment. Loose temporal task coupling specifically helps tolerate temporal control divergence. Note that density-time, discussed in Section 3.3, is another technique that helps tolerate temporal control divergence by eliminating chimes with no active µthreads. Note that it is more common to have a mix of both spatial and temporal divergence.

5.2 Loop-Task Accelerator Engine Template

Figure 5.7 shows the detailed microarchitectural diagram of the LTA engine template, an elegant microarchitectural template designed to enable flexible design-space explorations of spatial and temporal task coupling. This template can be configured at design time with any number of µthreads, lanes, lane groups, chimes, and chime groups.

The LTA engine template has a configurable number of lanes that are aggregated into *lane groups*. A lane group is a collection of lanes with a common frontend that is capable of sustaining independent instruction streams. A lane group is decoupled in space with respect to other lane



(a) Top-Level LTA Engine Microarchitecture (b) Detail of Lane Group

Figure 5.7: LTA Engine Template – IMU = instr mgmt unit; TMU = task mgmt unit; DMU = data mgmt unit; PIB = pending instr buffer; FPU = floating-point unit; MDU = integer mult/div unit; PDB = pending data buffer; FU = fetch unit; DU = dispatch unit; IU = issue unit; Seq = chime sequencer; SLFU = short-latency integer functional unit; LSU = load-store unit; WCU = writeback/ commit unit; PC = program counter; RT = rename table; PFB = pending fragment buffer; IQ = issue queue; WBQ = writeback queue; µRF = µthread regfile. l = tot num lanes; g_l = num lane groups; y = num lanes per lane group (l/g_l); c = tot num chimes; g_c = num chime groups; z = num chimes per chime group (c/g_c). Thick green arrows indicate channels that can transfer y worth of data in a single cycle.

groups. Each lane group manages a subset of the µthreads available in the LTA engine. These µthreads are mapped across the lanes within a lane group and multiple µthreads can be mapped to a single lane, but each µthread always has its own architectural register file. Furthermore, µthreads within a lane group are always tightly coupled in space, but can have variable coupling in time. A lane group is composed of a fetch unit (FU), a decode/dispatch unit (DU), issue units (IUs), short-latency functional units (SLFU), a load-store unit (LSU), and a writeback-commit unit (WCU). These units are connected by latency-insensitive interfaces, enabling a highly elastic pipeline that helps tolerate microarchitectural latencies.

Expensive resources are shared across lane groups so lane groups must dynamically arbitrate for access to these resources. Shared resources include the instruction memory port, which is managed by the *instruction management unit* (IMU), the long-latency functional units (LLFUs) like the floating-point unit (FPU) and the integer multiply-divide unit (MDU), which are organized into a FPU group and an MDU group, and the data memory ports, which are managed by the *data management unit* (DMU). There is always a single instruction memory port per LTA engine, but the number of FPUs, MDUs, and data memory ports are equal to the number of lanes per lane group. IMUs consist of per-task-group pending instruction buffers (PIBs) that amplify the instruction memory bandwidth by storing a cache-line-worth (32B) of instructions that can be amortized over multiple accesses, as well as a crossbar with round-robin arbitration. The instruction memory bandwidth is 32B per cycle. DMUs consist of per-task-group pending data buffers (PDBs) that can facilitate access/execute decoupling by storing a task group's worth of 4B words loaded from memory, as well as a crossbar with round-robin arbitration. The data memory bandwidth is 4B per cycle per port, but the bandwidth across all ports in the DMU can be combined to enable one wide coalesced access per cycle. FPU/MDU groups have grant-and-hold round-robin arbitration that prioritizes one lane-group until an entire task-group is processed.

While resource sharing is orthogonal to spatial task coupling, the intuition here is that increasing the number of lane groups improves the performance on irregular applications which tend to have low LLFU instruction density (based on the application kernels examined in this thesis). This observation, along with the fact that the lane group's elastic pipeline helps tolerate latencies due to conflicts at shared resources, motivates coupling resource sharing and spatial task coupling together to improve area efficiency.

The *task management unit* (TMU) is the primary interface between the GPP and LTA engine. The TMU is responsible for dividing the core tasks sent by the GPP into µtasks, then dynamically scheduling these µtasks across lane groups by injecting them into per-lane-group µtask queues. Load balancing across lane groups occurs naturally as lane groups that finish µtasks faster will obtain more µtasks from the TMU. The range of the µtasks is set to be a multiple of the number of µthreads in a lane group. The reason for this is the same as why ranges of core tasks are sized to be a multiple of the total number of µthreads in the entire LTA engine. Empirical data suggests that using a multiple of two or three achieves a good balance between load balancing and loop overhead. Upon receiving a new core task, the TMU initializes a pending µtask counter with the number of

generated μtasks. Lane groups assert a completion bit when they finish executing a μtask. The TMU aggregates the completion bits and decrements the pending μtask counter accordingly. The TMU acknowledges the completion of a core task once the counter is zero by sending a completion message to the GPP. Currently, the GPP stalls until it receives this completion message.

When a lane group dequeues a μtask from its μtask queue, it first initializes the architectural state of all of the μthreads that it manages so that the argument registers match the loop-task function arguments: argument pointer in `a0`, start index in `a1`, end index in `a2`, and the range step value in `a3` (always set to be the number of μthreads in a lane group). It does this by injecting four micro-ops that encode the fields of the μtask as an immediate value and write this value to the proper register. The start index must be initialized with a special micro-op that offsets the start index with the μthread index, which effectively partitions the μtask even further and maps the resulting μtasks onto μthreads.

A lane group begins executing μtasks using its μthreads once this initialization is complete. The FU initializes the per-chime-group program counters (PC) to the function pointer of the μtask and begins fetching instructions from this address. Fixed-priority arbitration is used to determine which of the chime groups can fetch an instruction every cycle. Fetched instructions are forwarded to the DU via per-chime-group dispatch queues.

The DU is able to decode and dispatch one instruction to each of the three IUs available in the LTA engine every cycle. Instructions to be dispatched are selected from the dispatch queues based on round-robin arbitration. Only instructions ready to execute are eligible for dispatch. Dependency checking is performed on a per-chime-group basis. Note that the entire chime group must stall on conditional branches until all μthreads in the chime group have resolved the branch. Instructions are dispatched in-order with respect to its chime group to an IU that manages the functional unit required by the instruction type. Simple register renaming via the rename table (RT) and writeback queues (WQ) is used to allow out-of-order writeback. Entries in the WQ are reserved by the DU at chime-group granularities. The DU also tags dispatched instructions with the total number of valid chimes (i.e., chimes with at least one active μthread) corresponding to each instruction.

Divergent branch resolutions within a chime group are handled by executing the not-taken μthreads (active) first and pushing a task group fragment representing the taken μthreads (inactive) into a *pending fragment buffer* (PFB) to be executed later. Fragments in the PFB can reconverge

with other fragments (including the active fragment) with matching PCs. A two-stack PFB is implemented as described in [LAB+11, KLST13], which prioritizes fragments in current loop iterations.

Dispatched instructions wait in the IU's in-order issue queue (IQ) until its operands are ready to be bypassed from the functional units or the WQ, or read from the register file. Operands are read for the entire chime group from 6r3w register file banks. The relatively high number of ports is necessary to allow all three IUs to read two operands and write one result per cycle. Each IU manages a separate set of functional units. By default, two IUs manage SLFUs and one IU manages LSUs. The IU sequences the chimes of instructions that have read all of their operands. The instruction is not dequeued from the IQ until all chimes in the corresponding chime group have been issued to the appropriate functional units. Lane groups also support density-time execution [SFS00], which allows the sequencer to skip scheduling chimes that have no active μthreads.

Since μthreads within a task group execute in lock-step, the frontend operations discussed thus far are amortized across all μthreads in the chime group. However, chime groups are loosely coupled in time with respect to each other, meaning instructions from a given chime group are fetched, decoded, and dispatched independent of other chime groups within the same lane group. This allows instructions from different chime groups to be multiplexed in time to hide microarchitectural latencies.

Sequenced chimes of an instruction are issued to a group of functional units organized into lanes. The μthreads of a given chime are executed in parallel across the number of lanes in the lane group. SLFUs handle integer operations and branches, while LSUs handle memory operations. The LSU can generate one memory request per lane per cycle, and supports coalescing of memory requests across μthreads within the same chime. As mentioned before, LLFUs are shared across lane groups so chimes of an LLFU instruction must arbitrate for access to the shared LLFUs with other lane groups.

The WCU arbitrates writes from functional units to the WQ at chime granularities. The write-back arbiter uses grant-and-hold round-robin arbitration that prioritizes one chime group until an entire chime group has written back to the WQ. Each entry in the WQ can store a chime group's worth of data and has a counter that is incremented every time a chime for the corresponding instruction writes back to the WQ. The WCU knows that the entire chime group's worth of data

is ready to be written back to the register files once this counter is equal to the number of valid chimes for the instruction. Entries in the WQ that are ready to be written back are committed to the register files in program order one chime at a time.

The lane group knows that all μtasks have been executed to completion when all μthreads have returned from the loop-task function. Explicitly, this is known when the `jr` instruction is decoded and there are no more fragments in the PFB. At this point, the lane group asserts a completion bit to the TMU and is ready to dequeue another μtask from its μtask queue.

Note that the LTA engine supports function calls within a μtask. In order to differentiate between such a function call and the end of the μtask, the return address register (`ra`) is initialized to a special token at the beginning of every μtask execution. The LTA engine knows that it has completed the μtask when the return address of a `jr` instruction is this special token, otherwise it will treat it as a normal function call return. The return address with the special token is saved on the stack before a function call and restored upon return according to standard calling convention.

In terms of exploring the task-coupling taxonomy, the level of spatial task coupling can be configured by varying the number of lane groups, while the level of temporal task coupling can be configured by varying the number of chime groups that can be multiplexed within a lane group.

5.2.1 Exception Handling

The LTA engine supports precise exceptions within a μthread but not between μthreads. When a μthread raises an exception, an exception task fragment is pushed onto the PFB with the PC of a special LTA exception handler (different from the OS-level exception handler) and a mask specifying the μthread which caused the exception. This exception task fragment is only allowed to reconverge with other exception fragments and will remain in the PFB until all other μthreads have run to completion. When the exception task fragment finally executes, only the μthreads that raised an exception will begin executing the LTA exception handling routine. The LTA exception handling routine saves the architectural state into the per-μthread stack and sends a pointer to the saved state to the TMU using the coprocessor interface. The TMU sends an exception handling request to the GPP with this saved state pointer, processing one μthread at a time across all lane groups. When the GPP receives an exception handling request, it initializes its architectural state with the data referenced by the saved state pointer, then jump to the OS-level exception handler. Once the exception has been handled, the GPP sends a confirmation to the LTA engine. In addition,

the GPP streams any updated architectural state back to the LTA engine for the current μthread. When all the μthreads that raised an exception within a given lane group have been processed, the TMU signals that the lane group can continue executing the exception fragment from the loop iteration index that raised the exception. At this point, the lane group continues processing the rest of the μtask to completion. All the argument registers stay valid on each μthread's register file context throughout this entire process.

5.2.2 Deadlock Detection and Resolution

The LTA engine also has a mechanism to dynamically detect and resolve deadlocks on irregular applications with significant inter-thread synchronization. To understand when this may occur, consider a graph algorithm that needs to obtain the lock on all of its neighbors before being able to apply an operator to its neighborhood. It is possible for the graph to have a cycle between several nodes and for all of the nodes in the cycle to be assigned to μthreads in the same task group. Now, assume that each μthread checks its left neighbor first then its right neighbor. In this case, all μthreads will obtain the first lock and succeed. However, all μthreads will *fail* to obtain the second lock, thereby sending the μthreads back to the top of the loop to try again and again, resulting in a deadlock. This happens because μthreads in the same task group must execute in lock-step. If a subset of the μthreads were allowed to run to completion before the others, the cycle would be broken and the deadlock avoided.

The LTA engine alleviates this problem by dynamically detecting this deadlock scenario and forcing control divergence. To support this mechanism, the LTA ISA must be further extended with a new `xplock` instruction that has the same semantics as an `amo.xchg` except that it is only used for obtaining a binary lock with special tokens for success and failure. If an `xplock` instruction returns a failure token for all active μthreads, the LTA increments a deadlock counter in the DU. When the next `xplock` instruction is decoded, the DU forces divergence by creating a task group fragment with half of the active μthreads and pushing this fragment into the PFB, thereby reducing the number of active μthreads that can trigger the deadlock. Task group fragments are pushed into the PFB in this manner every time the deadlock counter is incremented, until there is only one active μthread. The counter is only reset when at least one μthread returns a success token on a `xplock` instruction, signaling forward progress. To prevent reconvergence, task group fragments generated in this way are tagged with a special no-reconverge bit. Although this mechanism is not

guaranteed to solve every deadlock scenario, it was effective in preventing deadlock in a couple of the application kernels examined in the evaluation.

CHAPTER 6
LOOP-TASK ACCELERATOR EVALUATION METHODOLOGY

This chapter describes the vertically integrated research methodology used to evaluate the LTA platform, which spans applications, runtime, architecture, and VLSI. Applications kernels from diverse application domains are selected to represent both regular and irregular loop-task parallelism. To build confidence in the performance of the LTA runtime, a native port of the LTA runtime is validated against other popular task-based work-stealing runtimes. Details of the cycle-level performance, area, and energy modeling of the LTA platform is also provided in this chapter.

6.1 Application Kernels

The 16 C++ application kernels used to evaluate the LTA platform were ported from the problem-based benchmark suite [SBF+12] (PBBS) and an in-house benchmark suite. Application kernels are selected from a diverse range of application domains including those with more regular loop-task parallelism like scientific computing (e.g., N-body simulation, MRI gridding, dense matrix multiplication) and image processing (e.g., bilateral filter, color space conversion, discrete cosine transform), as well as application domains with more irregular loop-task parallelism like graph algorithms (e.g., breadth-first search, maximal matching, maximal independent set) and search/sort algorithms (e.g., radix sort, substring matching, dictionary). Table 6.1 catalogues the datasets used with each of the application kernels in the evaluation, and Table 7.1 provides useful simulation statistics for select LTA engines running these application kernels. Application kernels are cross-compiled using GCC-4.4.1, Newlib-1.17.0, and the GNU standard C++ library with a MIPS-derived backend. Note that because this version of GCC does not support C++11 lambdas, the macro-based `parallel_for` is used in all of the ported application kernels. As such, only the assembler needs to be extended to support the LTA ISA, and no further modifications to the cross-compiler are necessary. All application kernels are parallelized using the LTA software framework and minimal LTA-specific optimizations are applied.

Brief descriptions of the application kernels ported from PBBS follow, but the reader should refer to [SBF+12] for more detailed descriptions. *bfs-d* and *bfs-nd* generate a breadth-first-search tree from a directed cyclic graph. Computation is parallelized across nodes in the frontier. Con-

			DynInst (M)			Avg Size		Intensity		
Name	**Suite**	**Input**	**S**	**P**	**T%**	**Task**	**Iter**	**slfu**	**llfu**	**mem**
nbody	pbbs	3DinCube_1000	92	93	99%	1000	31K	18%	43%	33%
bilateral	custom	256×256 image	26	27	99%	66K	409	25%	51%	16%
mriq	custom	100-space, 256 points	11	11	99%	256	23K	53%	20%	21%
sgemm	custom	256×256 FP matrix	75	76	99%	576	131K	47%	19%	21%
rgb2cmyk	custom	1380×1080 image	43	43	99%	1380	31K	47%	0%	39%
dct8x8m	custom	782 8x8 blocks	55	55	99%	50K	1096	4%	64%	30%
knn	pbbs	2DinCube_10K	35	43	33%	9867	716	17%	32%	37%
bfs-nd	pbbs	randLocalGraph_J_5_150K	23	55	81%	36K	99	56%	0%	26%
radix-2	pbbs	exptSeq_500K_int	57	69	81%	46	92K	59%	0%	33%
radix-1	pbbs	randomSeq_1M_int	93	104	94%	229	74K	57%	0%	33%
rdups	pbbs	trigramSeq_300K_int	36	56	99%	508K	23	56%	0%	21%
sarray	pbbs	trigramString_200K	68	75	86%	76K	50	56%	0%	29%
strsearch	custom	210 strings, 210 docs	20	20	99%	210	49K	57%	0%	19%
bfs-d	pbbs	randLocalGraph_J_5_150K	23	35	95%	50K	75	56%	0%	26%
dict	pbbs	exptSeq_1M_int	39	51	99%	451K	25	66%	0%	19%
mis	pbbs	randLocalGraph_J_5_400K	14	32	99%	400K	27	52%	0%	25%
maxmatch	pbbs	randLocalGraph_E_5_400K	23	49	94%	1.7M	19	58%	0%	19%

Table 6.1: Application Kernel Characterization for Simulator Experiments – DynInsts = dynamic instruction count in millions (S for serial impl, P for parallel impl); T% = percent of total dyn. insts in tasks; Avg Task Size = average number of dyn. insts per task; Avg Iter Size = average number of dyn. insts per iteration; { slfu, llfu, mem } Intensity = percent of total dyn. insts that are {short-latency arithmetic, long-latency arithmetic, memory operation}.

flicts are resolved either deterministically with priority writes (by ID) or non-deterministically using compare-and-swap operations. *bfs-d* and *bfs-nd* use deterministic and non-deterministic algorithms respectively, and capture the impact of deterministic execution for the same problem. Determinism is a desirable trait that can mitigate the difficulties of reasoning about both correctness and performance in complex systems [SBF+12]. *dict* measures performance of batch operations on a dictionary data structure. Computation is parallelized across batch inserts, deletes, and searches of sequences of values and accommodates repeated keys. *radix* executes a stable sort of fixed-length unsigned integer keys in ascending order. Two datasets for *radix* are examined (i.e., `radix-1`, `radix-2`), since this application kernel exhibits strong data-dependent variability. *knn* finds the nearest neighbor to each point in a 2D space. A quad-tree is built to speed up neighbor lookups, and computation is then parallelized across the nodes to find the nearest neighbors. *mis* finds the maximal independent set of an undirected graph. Computation is parallelized across

nodes and conflicts are resolved using atomic compare-and-swap operations. *maxmatch* finds a maximal matching on an undirected graph. Computation is parallelized across edges. Endpoints are claimed using atomic compare-and-swap operations. *nbody* computes the net 3D force vector on each particle when subject to forces arising from other particles. The Barnes-Hut (BH) approximation is used with a traditional divide-and-conquer approach. An oct-tree is used to organize particles and approximate the center of mass of particle groups. *rdups* uses a parallel loop to insert elements into an internally deterministic hash table. *sarray* is a parallel variant of the Karkkainen and Sanders algorithm that generates the suffix array of a sequence of strings.

Descriptions for the application kernels ported from the in-house benchmark suite follow. *bilateral* performs a bilateral image filter with a lookup table for the distance function and an optimized Taylor series expansion for calculating the intensity weight. Computation is parallelized across output pixels. *dct8x8m* calculates the 8x8 discrete cosine transform on an image. Computation is parallelized across 8x8 blocks. *mriq* is an image reconstruction algorithm for MRI scanning. Computation is parallelized across the output magnetic field gradient vector. *rgb2cmyk* performs color space conversion on an image and computation is parallelized across the rows. *strsearch* implements the Knuth-Morris-Pratt algorithm with a deterministic finite automata to search a collection of byte streams for a set of substrings. Computation is parallelized across different streams. *sgemm* performs a single-precision matrix multiplication for square matrices using a standard blocking algorithm. Computation is parallelized across blocks.

6.2 Loop-Task Accelerator Runtime

The goal of this section is to validate that the LTA runtime described in detail in Section 4.2 is competitive with other popular task-based work-stealing runtimes when all of these runtimes (including the LTA runtime) are executed on traditional CMPs. Figure 6.1 compares the performance of the x86 port of the LTA runtime (using a standard indirect function call instead of `xpfor`), Intel Cilk++, and Intel TBB running six application kernels using the same experimental setup as in Chapter 2. Results are normalized against the performance of the optimized single-threaded implementation of each application kernel. The results show that the LTA runtime indeed has comparable performance to Intel TBB and Cilk++. In fact, the LTA runtime is slightly faster in some cases, like *mriq* and *maxmatch*, because it does not support C++ exceptions nor task cancel-

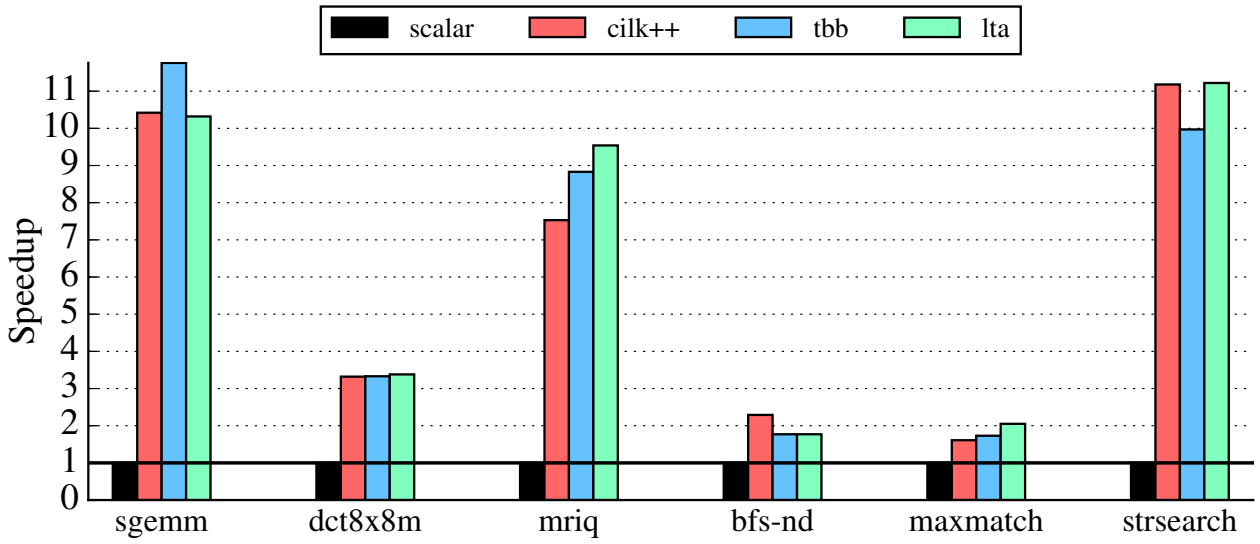


Figure 6.1: Performance Comparison of Various Runtimes on x86 – Speedup over optimized single-thread implementation of various work-stealing runtimes using 12 threads on a Linux server with two Intel Xeon E5-2620 v3 processors. Cilk++ = uses Cilk++'s `cilk_for`; TBB = uses TBB's `parallel_for`; LTA = uses x86 port of the LTA runtime. All apps are compiled with Intel C++ Compiler 15.0.3.

lations, and is more specialized for a narrower subset of task parallelism, making the LTA runtime relatively lighter weight compared to the alternatives.

6.3 Performance Modeling

For cycle-level performance modeling, a co-simulation framework with gem5 [BBB+11] and PyMTL [LZB14] is utilized. The cycle-level models of the GPPs, system interconnect, and memory system were adopted from gem5, a popular C++-based computer architecture simulator. Table 6.2 lists the specific gem5 configurations used for the experiments in the evaluation. Changes to the gem5 framework include: modifications to the in-order processor model to support co-simulation with the PyMTL-based LTA engine models, and an implementation of the `xpfor` instruction for sending core tasks to the LTA engine. The cycle-level models of the LTA engines were developed using PyMTL, a productive Python-based hardware modeling framework. This co-simulation framework leverages the C-embedding features of PyPy [pyp14], which allow a PyPy interpreter to be dynamically linked against a C/C++ program providing very fast communi-

Technology	TSMC 40nm LP, 500MHz nominal frequency
ALU	4/10-cycle Int Mul/Div, 6/6-cycle FP Mul/Div, 4/4-cycle FP Add/Sub
IO	1-way, 5-stage in-order, 32 Phys Regs
O3	4-way, out-of-order, 128 Phys Regs, 32 Entries IQ and LSQ, 96 Entries ROB, Tournament Branch Pred
Caches	1-cycle, 2-way, 32KB L1I, 1-cycle 4-way 32KB L1D per core with 16-entry MSHRs; 20-cycle, 8-way, shared 1MB L2; MESI protocol
OCN	Crossbar topology, 2-cycle fixed latency
DRAM	200ns fixed latency, 12.8GB/s bandwidth SimpleMemory model

Table 6.2: Cycle-Level Simulator System Configuration

cation between the two languages. gem5 is used as the main driver to start the simulation, load the program, and tick the PyMTL simulation.

The evaluation uses two baseline architectures in gem5 to compare the performance, area, and energy of the LTA platform. *IO* denotes the baseline scalar five-stage in-order processor, and *O3* describes the baseline four-way superscalar out-of-order processor. *IO* is also used as the GPP to which LTA engines are attached. Multi-core experiments use a four-core system with private L1 caches and a shared L2 cache. *MC-IO* denotes four *IO* cores and *MC-O3* denotes four *O3* cores. A bare-metal system is simulated with syscall emulation in gem5.

As mentioned before, the evaluation focuses on the 12 LTA engines highlighted in Figure 5.5. LTA engines are denoted according to the task-coupling terminology defined in Section 5.1 with the *LTA-* prefix. For example, a 32-μthread LTA engine with eight lanes, four lane groups, four chimes, and two chime groups is denoted as *LTA-8/4x4/2*.

6.4 Area Modeling

Area and energy are estimated using component/event-based modeling backed by VLSI results obtained from synthesizing comparable accelerator designs. Specifically, the RTL implementations of the FG-SIMT microarchitecture [KLST13] (see Chapter 3) and the XLOOPS microarchitecture [SIT+14] were synthesized and placed-and-routed using Synopsys DesignCompiler and IC

Compiler with a TSMC 40 nm LP standard-cell library characterized at 1 V. SRAMs were modeled using CACTI [MBJ09] since a memory compiler was not available for the target process.

Component-based area modeling estimates area by using an area dictionary containing the area of various hardware components and a list of comparable components in the target microarchitecture. In order to reduce complexity, only the dominant contributors to area are defined in the area dictionary. For example, the dominant contributors to area in the FG-SIMT and XLOOPS models are the L1 caches (33%), regfiles (26%), LLFUs (20%), SLFUs (10%), and assorted queues (7%). Components from the FG-SIMT model are used to estimate the area of the lane-group, DMU, and D$ crossbar network. The area of each lane group is based on the FG-SIMT vector unit with a scaled number of μthreads and lanes, which affects the number of regfiles and functional units. The bit-width of the PDBs and PFBs were scaled down accordingly from the similar VLU/VSU queues and PVFBs in FG-SIMT. Because the D$ crossbar in FG-SIMT has a fixed number of eight ports, the area of crossbars with less ports is estimated by applying a scaling factor that assumes the area increases roughly quadratically with the number of ports. Components from the XLOOPS model are used to estimate the area of the IMU and TMU. The area of the IMU and TMU were based on the XLOOPS area corresponding to the L0 instruction buffers (except modeled as flop-based registers for PIBs) and the lane-management unit.

The area of the dominant contributors above were broken down to μthread, lane, and chime granularities (e.g., area of regfile context per μthread) to better match the configurability of LTA engines. Components in the LTA engines that had no direct analog in either the FG-SIMT or XLOOPS models were estimated using a reasonable combination of integer ALUs, muxes, and 1r1w registers. For example, the area of the dynamic arbiter in the WCU is estimated as a number of 3-to-1 muxes equal to the number of writeback ports in the WQ, with a bitwidth equal to the bitwidth of an entire chime, along with an ALU and a register for each output port to model the grant-and-hold round-robin arbitration logic. Other components that have no equivalent include the RT and the WQ, which were simpler to estimate since they are essentially regfiles.

Since no RTL implementations of microarchitectures comparable to *O3* were available, a reasonable scaling factor of $\approx 3\times$, based on rough McPAT area estimates [LAS+09], was applied to the area of *IO* to estimate the area of *O3*.

6.5 Energy Modeling

Event-based energy modeling estimates energy by using an energy dictionary containing the energy of various microarchitectural events (e.g., regfile read/write) and a list of events with the corresponding number of occurrences generated from cycle-level simulations of the target microarchitecture. In order to reduce complexity, only the dominant contributors to energy are defined in the energy dictionary. The dominant contributors to energy in the FG-SIMT and XLOOPS models are the same as the dominant contributors to area (e.g., caches, regfiles, SLFUs/LLFUs).

Gate-level simulations with energy microbenchmarks were used to extract per-access energies of the dominant contributors. A suite of 70+ energy microbenchmarks were developed to measure per-event energy in the in-order core and the FG-SIMT/XLOOPS microarchitectures. For example, the `addiu` energy microbenchmark warms up the instruction cache and then executes 100 `addiu` instructions in sequence. These microbenchmarks are used to generate bit-accurate activity factors that are then combined with post-place-and-route layout using Synopsys PrimeTime PX for total power estimates and breakdowns. In conjunction with the cycle time, it is possible to calculate per-component energy for each event stressed in the microbenchmark.

Again, due to a lack of access to an *O3*-like RTL implementation, McPAT's component-level models are used to derive energy scaling factors for events in *O3* relative to a comparable component in *IO* and McPAT (e.g., scalar register file and integer ALU). Events in the LTA engine with no analog in FG-SIMT/XLOOPS are also estimated using McPAT's component-level models. Note that the energy microbenchmarks running on *IO* gate-level models, *IO* cycle-level simulation, and *O3* cycle-level simulation were carefully analyzed to ensure that the McPAT component models matched general intuition and correlated well with the other VLSI-based component models. In some cases, additional RTL was written and gate-level simulations performed to obtain a more accurate energy estimate. For instance, different configurations of the D$ crossbar was synthesized to determine a roughly linear increase in energy per crossbar access access with varying number of ports.

CHAPTER 7
LOOP-TASK ACCELERATOR EVALUATION RESULTS

This chapter begins by exploring the design space for a single LTA engine in order to identify a set of suitable configurations that can efficiently accelerate both regular and irregular loop-task parallelism. A 32-μthread LTA engine is the basis for the evaluation, and both four-lane and eight-lane configurations are analyzed. The design space is narrowed by first examining the impact of spatial task coupling on performance, area, and energy efficiency, then doing the same for temporal task coupling. Based on these studies, the LTA engines with the most promising spatial and temporal task coupling configurations are identified. Next, the most promising LTA engines are used to evaluate the multiplicative effect of exploiting loop-task parallelism across cores using the LTA runtime and within a core using these LTA engines. To determine the success of the LTA platform in achieving the ultimate goal of the 3P's, a qualitative evaluation of the productivity and portability is provided. Finally, the end of this chapter contains a set of case studies with deeper examinations of the impact of the LTA engine's other microarchitectural parameters.

7.1 Loop-Task Accelerator Engine: Spatial Task-Coupling

Figure 7.1 shows the single-core performance of LTA engines with a single chime group and varying spatial task coupling. On average, the LTA engines achieve speedups of 4–5× compared to *IO* and 2–2.5× compared to *O3*. Table 7.1 contains many useful statistics from running the application kernels used in this study on various LTA engine configurations. The discussions that follow will reference numbers from this table to help understand the observed trends.

In general, **tighter spatial task coupling improves the performance on regular application kernels with high LLFU instruction density.** Examples include *nbody*, *bilateral*, *mriq*, *sgemm*, and *dct8x8m*. This is due to the higher LLFU bandwidth available when there are more lanes per lane group. Because the LLFUs are pipelined, a higher bandwidth is only useful for bursty access patterns, which are more likely to occur with regular control flow and high LLFU instruction density. Regular control flow means more μthreads are likely to access the LLFU on a given LLFU instruction, and high LLFU instruction density means the distance between LLFU instructions is likely to be shorter. For example, increasing spatial task coupling from *LTA-8/4x4/1* to *LTA-8/1x4/1* reduces the percent of time stalled waiting on LLFUs from 27% to 19% on *nbody*, 26% to 17%

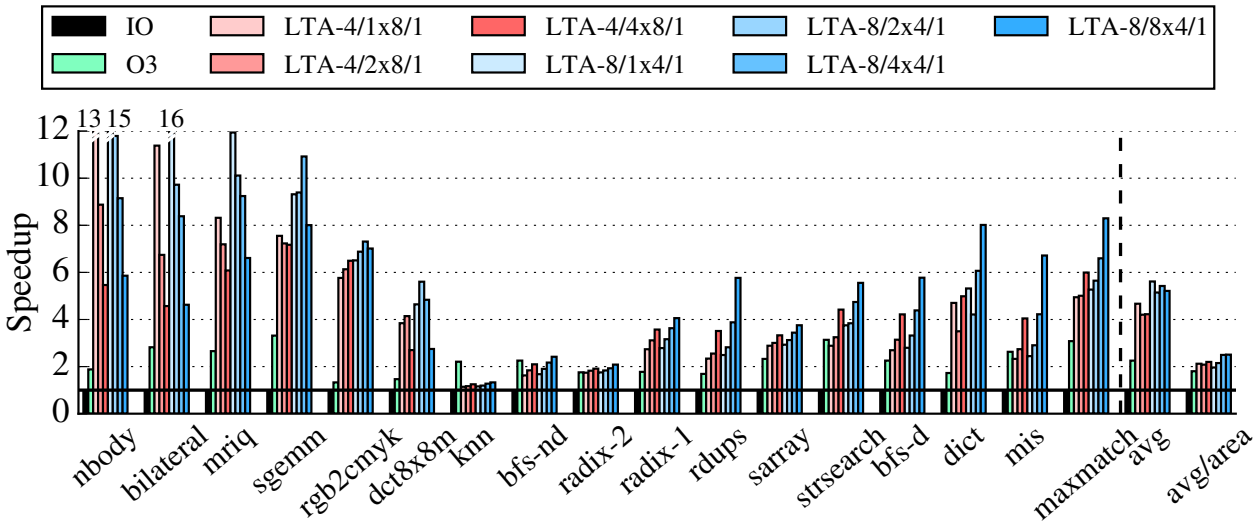


Figure 7.1: Performance of Single-Core LTA Engines with Variable Spatial Task-Coupling – Speedups of various 32-µthread LTA engines with variable spatial task coupling are normalized against the baseline scalar in-order core for each application kernel. Both 4-lane and 8-lane configurations are shown, as well as the speedup of the baseline four-way superscalar out-of-order core. Application kernels are ordered such that those with regular loop-task parallelism are on the left and those with irregular loop-task parallelism are on the right.

on *bilateral*, and 19% to 12% on *mriq*. Increasing the number of lanes will improve performance because of the higher LLFU bandwidth with tighter spatial task coupling, but also because of the increased SLFU and LSU bandwidth.

Tighter spatial task coupling does not improve the performance on application kernels that have high LLFU instruction density but irregular control flow, like *knn* which has 32% LLFU instruction density but an average of only five active µthreads per cycle on *LTA-8/1x4/1*, nor on application kernels that have regular control flow but low LLFU instruction density, like *rgb2cmyk* which has an average of 29 active µthreads per cycle on *LTA-8/1x4/1* but no LLFU instructions.

However, **tighter spatial task coupling can increase stalling due to cache misses.** This is a consequence of having more µthreads in a task group. Since task groups execute in lock-step, the entire task group must wait for all of its loads to be ready before a dependent instruction can be issued. The likelihood of at least one µthread in a task group missing in the cache increases with the number of µthreads in a task group. Furthermore, if multiple µthreads miss in the cache and the corresponding refills are serialized, the entire task group must wait for the last refill to be handled. The degree of this behavior's impact on performance depends on several factors such as the memory bottleneck and variability in memory-access patterns of the application, as well

Name	IO	MC -IO	MC -O3	LTA-8/1x4/1				LTA-8/4x4x1				LTA-8/4x4/2			
				I	A	S%	M	I	A	S%	M	I	A	S%	M
nbody	239.7	3.6	3.6	0.04	30	19.0	0.4	0.14	31	26.5	0.2	0.32	25	18.5	1.0
bilateral	61.9	4.3	3.8	0.03	31	17.0	2.1	0.14	31	25.5	1.2	0.26	31	14.5	1.2
mriq	13.9	4.0	4.1	0.04	32	12.0	0.0	0.14	32	19.2	0.0	0.27	32	7.5	0.0
sgemm	113.3	3.8	3.9	0.03	32	5.0	3.4	0.13	32	16.0	0.9	0.26	32	7.0	0.8
rgb2cmyk	57.1	3.3	2.5	0.04	29	1.0	9.7	0.15	30	1.0	4.4	0.27	31	0.0	7.1
dct8x8m	81.4	4.0	3.9	0.03	31	0.0	24.6	0.13	32	0.0	18.4	0.25	32	0.0	18.2
knn	57.3	1.5	1.3	0.21	5	35.0	10.7	0.41	11	40.0	10.6	0.57	15	22.0	10.6
bfs-nd	88.6	1.5	1.1	0.07	17	0.0	23.5	0.22	21	0.0	16.4	0.36	24	0.0	16.1
radix-2	102.3	1.1	1.1	0.04	30	0.0	49.5	0.14	31	0.0	48.9	0.27	30	0.0	48.7
radix-1	175.1	3.4	3.6	0.05	26	2.0	35.9	0.18	28	1.0	34.5	0.30	29	1.0	34.7
rdups	87.1	3.0	2.0	0.12	11	0.0	30.5	0.28	18	0.0	22.5	0.41	22	0.0	22.5
sarray	203.9	2.5	1.7	0.10	11	5.0	60.5	0.22	20	2.0	41.1	0.35	24	1.0	42.2
strsearch	30.0	3.3	3.7	0.20	6	3.0	0.5	0.51	10	1.0	0.6	0.49	18	0.0	0.6
bfs-d	88.6	2.4	1.5	0.11	10	1.0	13.7	0.29	16	1.0	10.0	0.41	21	0.0	9.8
dict	82.1	3.4	1.7	0.06	20	0.0	23.4	0.17	27	0.0	13.1	0.29	29	0.0	13.0
mis	79.4	3.4	1.6	0.25	5	0.0	20.1	0.49	10	0.0	13.9	0.61	15	0.0	14.2
maxmatch	154.3	3.4	1.3	0.05	25	0.0	16.3	0.17	28	0.0	12.2	0.30	29	0.0	12.3

Table 7.1: Application Kernel Statistics for Simulator Experiments – IO = number of cycles (in millions) of optimized single-threaded implementation on in-order core; MC-IO = speedup of multi-threaded implementation on 4 in-order cores over single in-order core; MC-O3 = speedup of multi-threaded implementation on 4 out-of-order cores over single out-of-order core; I = ratio of total inst fetches to total dyn. insts; A = average active µthreads in LTA engine per dyn. inst; S% = percent of execution time stalled due to non-memory microarchitectural latencies; M = misses in L1 D$ per thousand dyn. insts.

as the µtask to µthread mapping. For example, *sgemm* and *dct8x8m* see noticeable performance degradations with tighter spatial task coupling even though they both have relatively high LLFU instruction densities. Comparing *LTA-8/4x4/1* to *LTA-8/1x4/1*, the percent of time when stalled waiting on D$ misses increases from 26% to 52% on *sgemm* and 35% to 60% on *dct8x8m*.

On the other hand, **looser spatial task coupling improves the performance on irregular application kernels with spatial control divergence.** Examples include *rdups*, *sarray*, *strsearch*, *bfs-d*, and *mis*. Spatial control divergence causes lanes executing inactive µthreads to be idle until reconvergence or the completion of the task group fragment. By increasing the number of lane groups, more (smaller) task groups can be executed concurrently, which in turn can increase the average number of active µthreads per cycle. To better understand why this happens, consider the

perfect spatial control divergence case in Figure 5.6. Using one lane group with four lanes would result in only one of the fragments being executed concurrently, for instance the red fragment. Then imagine spatially splitting the task group in half and mapping these smaller task groups onto two lane groups with two lanes each: now two of the fragments can be executed concurrently, for instance the red and the green fragments. Splitting the task groups even further and mapping them onto four lane groups with one lane each will allow all of the fragments to be executed concurrently. Decreasing spatial task coupling from *LTA-8/1x4/1* to *LTA-8/4x4/1* increases the the average number of active µthreads per cycle across all lanes from 11 to 18 on *rdups*, 11 to 20 on *sarray*, 6 to 10 on *strsearch*, 10 to 16 on *bfs-d*, and 5 to 10 on *mis*.

Note that this does not address temporal control divergence since density-time already compresses chimes that have no active µthreads. However, this effect can still improve performance even when there is mixed control divergence (which is the more realistic scenario) precisely because density-time is able to compress inactive chimes. For example, Figure 5.6 with one lane group would yield only one active µthread per cycle, but four lane groups would yield four active µthreads per cycle because three of the four chimes would be inactive on every lane group.

One tradeoff of increasing the number of lane groups is an increase in I$ conflicts, which can partially offset the benefit of better tolerating control divergence. Fortunately, this rarely impacts performance since instruction fetches are amortized across the entire task group. For example, the only case of a tangible slowdown due to I$ conflicts is in *rgb2cmyk*, where decreasing spatial task coupling from *LTA-8/4x4/1* to *LTA-8/8x4/1* increases the percent of time when stalled due to I$ misses/conflicts from 7% to 38%. However, the I$ port may become the bottleneck as the number of µthreads in a task group decreases.

The estimated area of the LTA engines examined in this chapter are shown in Table 7.2 and the area breakdowns by component are shown in Figure 7.2. Overall, LTA engines consume a factor of 2–3× more area than *IO*. Figure 7.1 also shows the average area-normalized performance of LTA engines with varying spatial task coupling. Table 7.2 shows that a four-lane LTA engine is roughly twice the area of *IO* (including the I$ and D$), and a eight-lane LTA engine is roughly triple the area of *IO*. As such, the average raw speedups of 4× for four-lane LTA engines and 5× for eight-lane LTA engines are adjusted to average area-normalized speedups of 2× compared to *IO*. Although the area-normalized speedup of the LTA engines are only marginally higher than the area-normalized speedup of *O3* in a *single-core* context, the true area efficiency of the LTA

	Area	LTA	Area	LTA	Area	LTA	Area	LTA	Area
IO	0.61	*4/1x8/1*	1.34	*4/2x8/2*	1.23	*8/1x4/1*	1.74	*8/8x4/1*	1.27
O3	0.76	*4/2x8/1*	1.23	*4/2x8/4*	1.24	*8/2x4/1*	1.46	*8/4x4/2*	1.33
		4/4x8/1	1.17	*4/2x8/8*	1.24	*8/4x4/1*	1.32	*8/4x4/4*	1.34

Table 7.2: LTA Engine Area Estimates – Area is shown in mm^2 for single-core configurations and includes the L1 I$ and D$. Area is estimated using the component-based model described in Section 6.4.

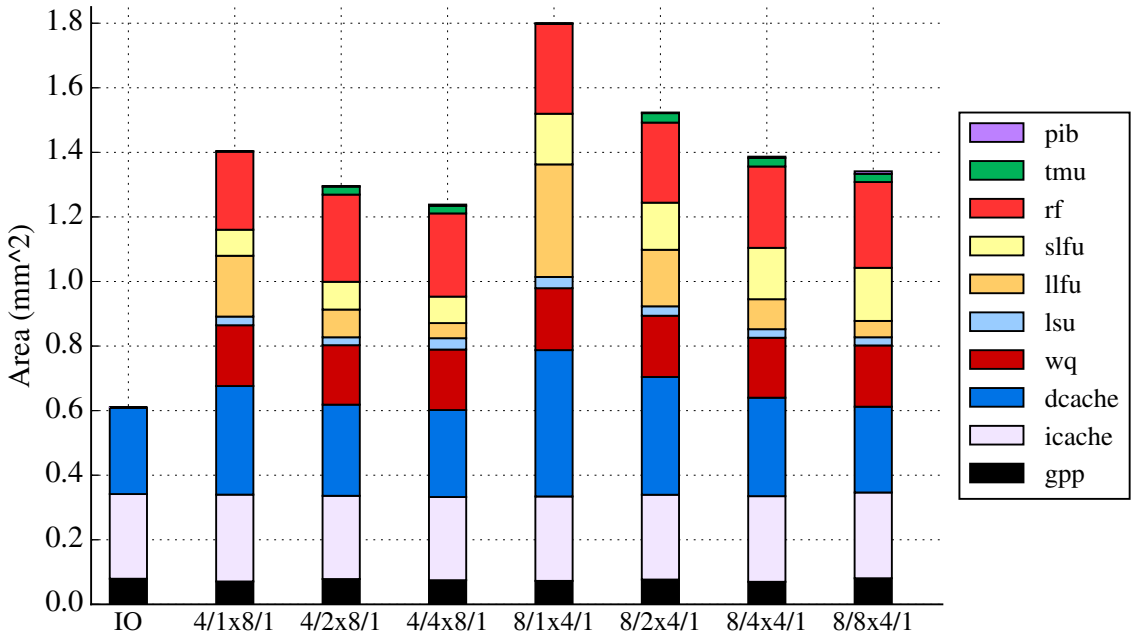


Figure 7.2: Area Breakdown of Single-Core LTA Engines with Variable Spatial Task-Coupling – Absolute area breakdowns of various 32-µthread LTA engines with variable spatial task coupling are shown. Both four-lane and eight-lane configurations are shown. In general, the area of LTA engines are 2–3× larger than the baseline in-order core.

platform will be made clear in a multi-core context later in this chapter. In addition, **looser spatial task coupling improves area-normalized performance as sharing resources reduces absolute area while not severely impacting average performance.** Shared resources include the LLFUs and D$ memory ports, the latter of which impacts how the cache is banked and ported, as well as the crossbar network used to access cache banks. Figure 7.2 shows that for the largest LTA engine in this study, *LTA-8/1x4/1*, the area of the LLFUs and the D$ crossbar network is 30% of the total area, which decreases as more resources are shared. However, this is partially offset by an increase in area from per-lane-group PIBs and frontend units (i.e., FU, DU, IU).

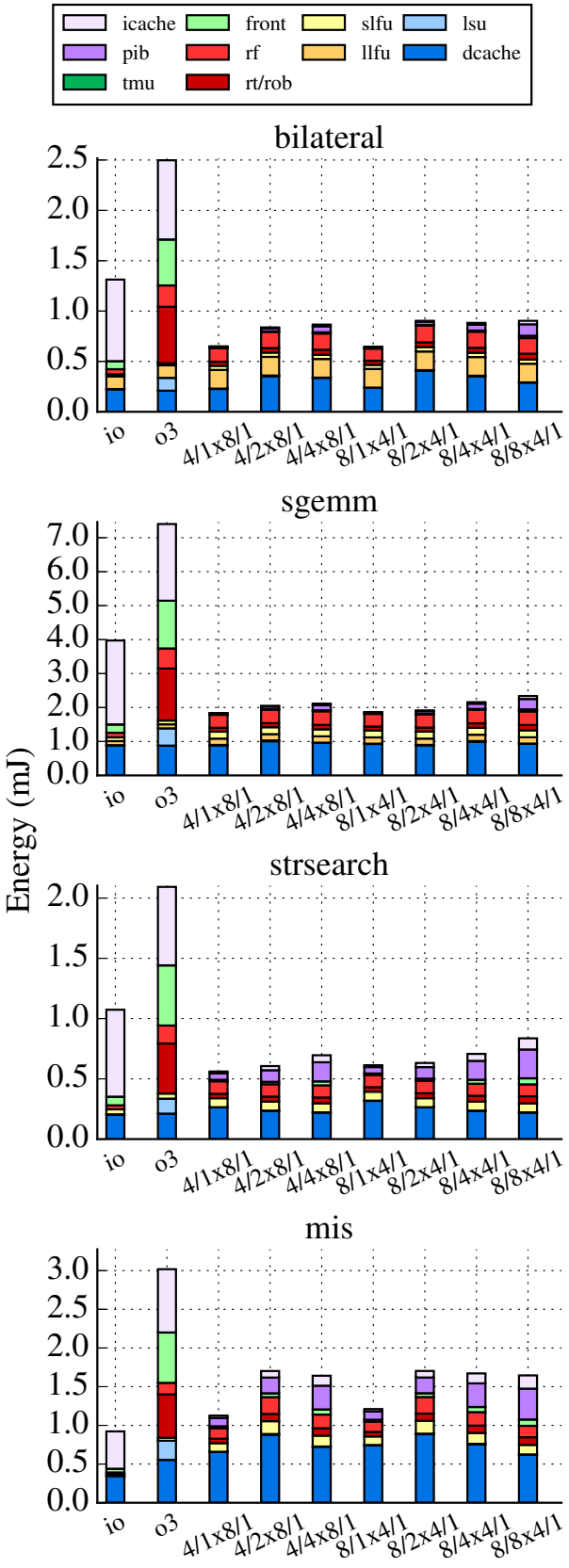


Figure 7.3: Spatial Task-Coupling Energy Breakdown

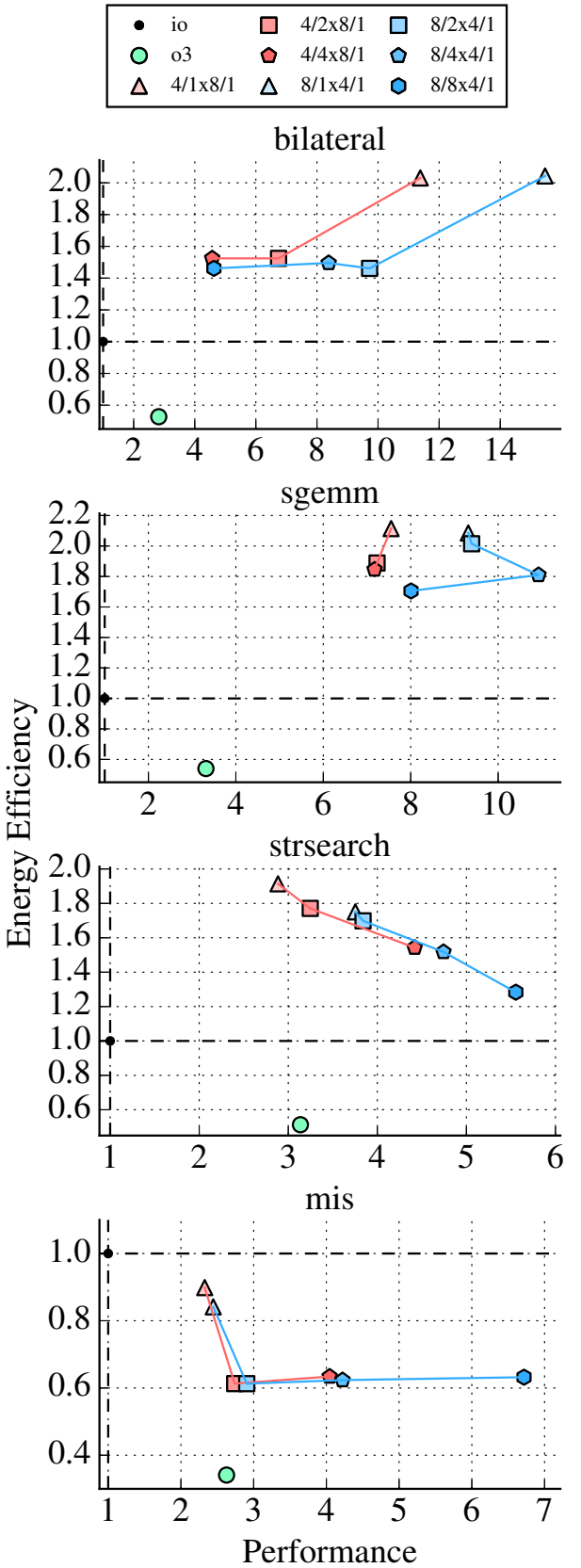


Figure 7.4: Spatial Task-Coupling Energy Efficiency vs. Performance

Figure 7.3 shows the absolute energy breakdowns of LTA engines with varying spatial task coupling for two regular application kernels, *bilateral* and *rgb2cmyk*, and two irregular application kernels, *strsearch* and *mis*. In general, **tighter spatial task coupling improves energy efficiency by amortizing the memory-access and frontend energy across more µthreads.** This amortization is limited to *active* µthreads, thus the energy reduction is more significant on regular application kernels. Because I$ and D$ access energy are the dominant contributors to total energy in the LTA engines, the primary energy savings are derived from amortizing memory accesses. Specifically, the instruction fetch amortization offers the biggest energy savings. For example, comparing *LTA-8/4x4/1* to *LTA-8/1x4/1*, the ratio of instructions fetched to total dynamic instructions decreases from 0.14 to 0.03 on *bilateral*, 0.13 to 0.03 on *sgemm*, 0.51 to 0.20 on *strsearch*, and 0.49 to 0.25 for *mis*. The PIBs further reduce instruction fetch energy by providing less expensive accesses to a given cache line. For all four application kernels, the combined I$ and PIB energy consistently decreases with tighter spatial task coupling. Increasing the number of lane groups increases energy due to redundant accesses for the same instruction, for irregular application kernels as well as regular application kernels with low spatial control divergence. Tighter spatial task coupling also facilitates memory coalescing across more lanes, but this is offset by the increased energy from higher-ported D$ crossbar networks. This is why some regular application kernels like *sgemm* only see marginal D$ energy savings in *LTA-8/1x4/1* compared to *LTA-8/8x4/4* even though 35% of all loads are coalesced. The frontend energy is less dominant, but the savings are tangible in more irregular application kernels. In regular application kernels, amortization across as few as four µthreads makes the frontend energy negligible.

Figure 7.4 shows the energy efficiency vs. performance of LTA engines with varying spatial task coupling for the same subset of application kernels in the energy breakdown. Each point represents the energy efficiency and performance of a different configuration in the LTA task-coupling taxonomy normalized to the baseline scalar in-order core. On average, the LTA engines achieve improvements in energy efficiency of 1.2–1.5× compared to *IO* and 2.6–3.1× compared to *O3*. One important point here is that regular application kernels that suffer from the aforementioned increase in D$-related stalls, like *sgemm*, still have higher energy efficiency with tighter spatial task coupling even though performance may decrease. All regular application kernels in this study have higher performance and energy efficiency than both *IO* and *O3*. Also, most irregular application kernels in this study have higher performance than both *IO* and *O3*. However, although all

irregular application kernels have higher energy efficiency than *O3*, not all of them have higher energy efficiency than *IO*. For example, in application kernels like *mis*, increasing the number of concurrent μthreads can exacerbate aborts from failing to obtain fine-grain locks as well as conflict misses in the cache. The resulting increase in D$ energy outweighs the reduction in I$ and frontend energy. Comparing *IO* to *LTA-8/1x4/1*, the number of total D$ accesses increases by 59% and the miss rate increases from 23% to 54%.

Although the average performance in Figure 7.1 does not significantly change with varying spatial task coupling, the key point here is that the extreme levels of spatial task coupling are specialized for either regular *or* irregular loop-task parallelism. For a true 3P platform, reducing the performance variance across applications with diverse loop-task parallelism is just as important as high average performance. These results suggest that **moderate spatial task coupling is ideal for achieving a reasonable compromise across both regular *and* irregular loop-task parallel applications in terms of performance, area, and energy.** As such, *LTA-4/2x8/1* and *LTA-8/4x4/1* can be identified as the most promising configurations for spatial task coupling in LTA engines.

7.2 Loop-Task Accelerator Engine: Temporal Task-Coupling

Figure 7.5 shows the single-core performance of LTA engines with varying temporal task coupling. **Moderate temporal task coupling can improve performance on some regular application kernels by reducing stalling due to cache misses.** Examples include *bilateral*, *mriq*, and *sgemm*. The reason for this is that increasing the number of chime groups creates smaller task groups, which as discussed before, means less μthreads will stall on a cache miss. For example, decreasing temporal task coupling from *LTA-8/4x4/1* to *LTA-8/4x4/2* reduces the percent of time when stalled waiting on D$ misses from 10% to 4% on *bilateral*, 20% to 11% on *mriq*, and 26% to 9% on *sgemm*.

However, **as the number of chime groups increases, this benefit can quickly become outweighed by increased I$ conflicts and decreased IU utilization.** Recall that I$ conflicts caused by increasing the number of lane groups, as discussed in the previous section, was rarely problematic because instruction fetches could be amortized across a relatively large task-group within each lane group. Smaller chime groups limit this amortization and further inflate the number of redundant accesses for the same instruction, thereby putting more pressure on the shared I$ port.

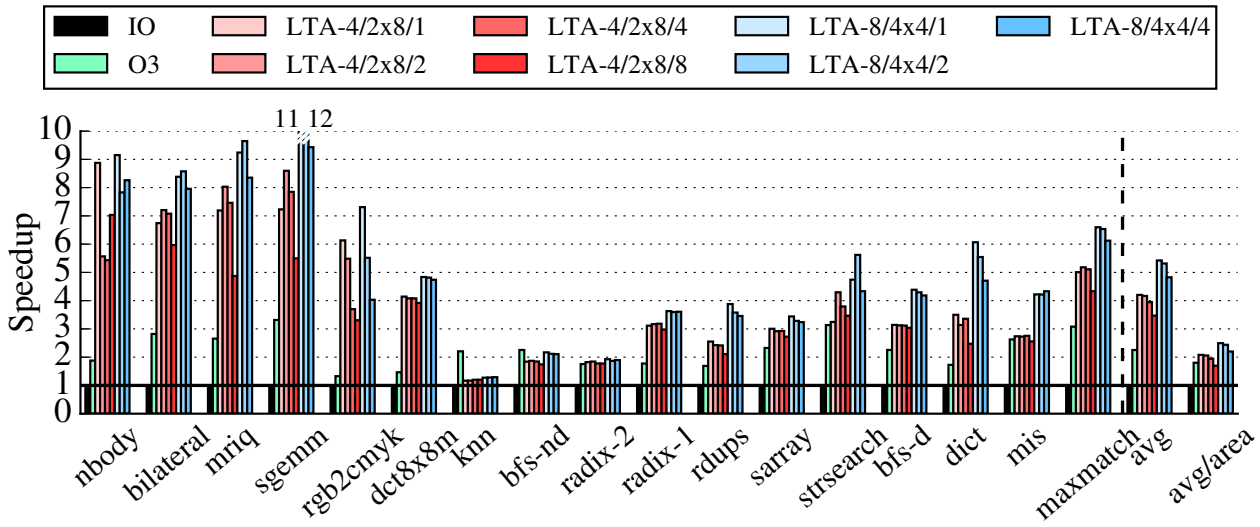


Figure 7.5: Performance of Single-Core LTA Engine with Variable Temporal Task-Coupling – Speedups of various 32-µthread LTA engines with variable temporal task coupling are normalized against the baseline scalar in-order core for each application kernel. 4-lane and 8-lane configurations with moderate spatial task coupling selected based on the study in Section 7.1 are used as the starting point for exploring the impact of temporal task coupling. As before, the speedup of the baseline four-way superscalar out-of-order core is also shown. Application kernels are ordered such that those with regular loop-task parallelism are on the left and those with irregular loop-task parallelism are on the right.

Furthermore, if the number of chimes per chime group is less than the number of IUs in a lane group, the DU will not be able to keep all of the IUs busy unless there is back-pressure. Even though the DU is capable of dispatching an instruction to each IU per cycle, the number of instructions it can dispatch is limited by the instruction fetch. For example, comparing *LTA-8/4x4/1* to *LTA-8/4x4/4*, the average IU utilization decreases by 35% on *mriq*. Another subtle tradeoff of increasing the number of chime groups is that the average number of active µthreads per cycle may decrease. This is because chime groups operate on separate PFBs and reconvergence across chime groups are not supported, thus some control-flow patterns can actually cause µthreads that would stay converged with one chime group to become diverged with more chime groups. *nbody* exhibits this behavior as evident from the decrease in the average number of active µthreads per cycle from 31 to 25 when comparing *LTA-8/4x4/1* and *LTA-8/4x4/2*.

Similarly, **moderate temporal task coupling can improve performance on irregular application kernels with temporal control divergence.** The only example where this can be observed is *strsearch*. Density-time helps tolerate temporal control divergence by compressing chimes with no active µthreads. This improves lane utilization, but it can also expose microarchitectural laten-

cies if the distance between dependent instructions is relatively short. To better understand why this happens, consider the perfect temporal control divergence case in Figure 5.6. Using one chime group would result in only one of the fragments being executed until completion or reconvergence. A back-to-back dependency on an LLFU instruction that takes four cycles would cause a stall of four cycles. Using two chime groups would allow one fragment from each chime group to be multiplexed in time, reducing the stall to two cycles on the same back-to-back dependency. Using four chime groups would completely hide this microarchitectural latency. In addition, fewer chimes per chime group reduce the branch resolution latency since LTA engines wait for all μthreads to resolve the branch before proceeding. Decreasing temporal task coupling from *LTA-8/4x4/1* to *LTA-8/4x4/2* decreases the percent of time when stalled waiting on all branches/LLFUs/D$ from 38% to 13% on *strsearch*.

The same issues with increasing the number of chime groups apply to irregular application kernels as well. For example, comparing *LTA-8/4x4/1* to *LTA-8/4x4/4* on *strsearch*, the percent of time when stalled waiting on I$ misses/conflicts increases from 37% to 55%, and the average IU utilization decreases by 60%.

Figure 7.5 also shows the average area-normalized performance of LTA engines with varying temporal task coupling. Table 7.2 shows that increasing the number of chime groups in both the four-lane and eight-lane LTA engines adds a negligible area overhead of <1%. The only contributors to this area overhead are the additional per-chime-group PIBs as well as the more sophisticated logic for multiple dispatch in the DU. Moderate temporal task coupling only slightly decreases average raw speedups, so the area-normalized speedup remains around 2–2.5× compared to *IO*. However, **the performance degradation of using extremely loose temporal task coupling more tangibly degrades area-normalized performance as well.**

Figure 7.6 shows the absolute energy breakdowns of LTA engines with varying temporal task coupling for two regular application kernels, *bilateral* and *rgb2cmyk*, and two irregular application kernels, *strsearch* and *mis*. The results indicate that **looser temporal task coupling diminishes energy efficiency by inflating redundant instruction fetches and frontend processing, while only marginally improving performance**. This is true for both regular and irregular application kernels. For example, comparing *LTA-8/4x4/1* to *LTA-8/4x4/4*, the ratio of instructions fetched to total dynamic instructions increases from 0.14 to 0.51 on *bilateral*, 0.13 to 0.50 on *sgemm*, 0.51 to 0.67 on *strsearch*, and 0.49 to 0.55 for *mis*.

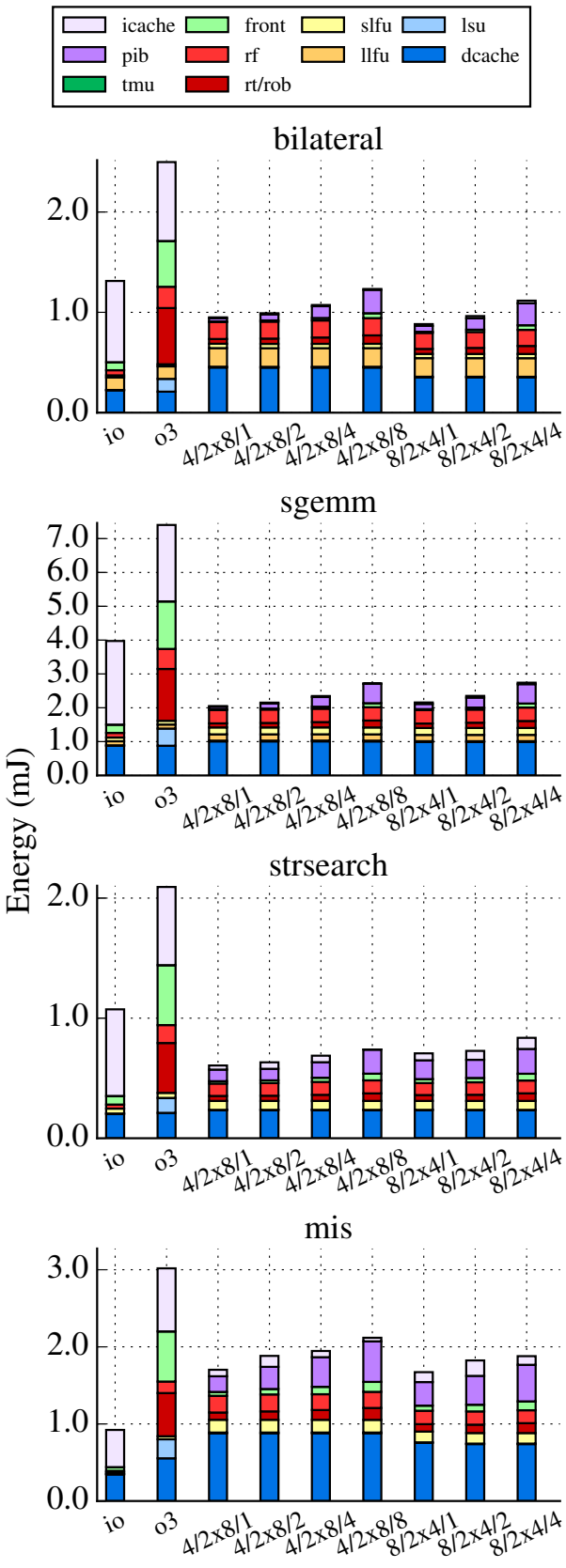


Figure 7.6: Temporal Task-Coupling Energy Breakdown

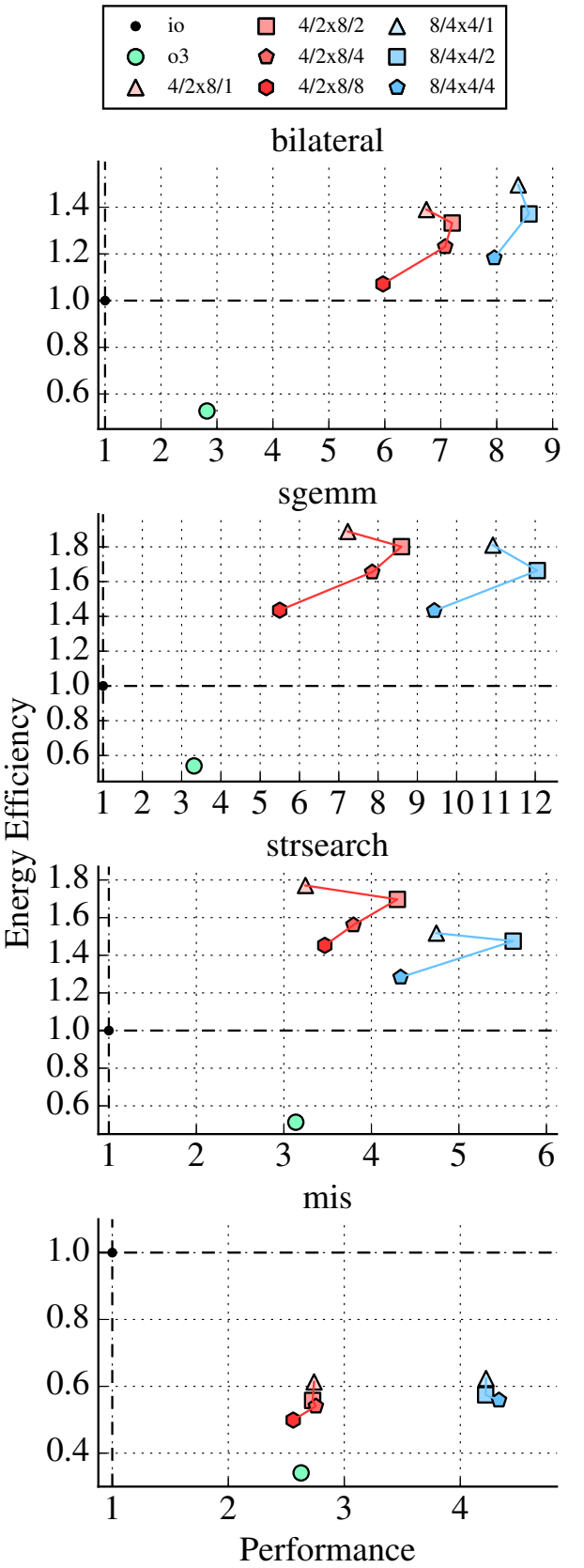


Figure 7.7: Temporal Task-Coupling Energy Efficiency vs. Performance

Figure 7.7 shows the energy efficiency vs. performance of LTA engines with varying temporal task coupling for the same subset of application kernels in the energy breakdown. Each point represents the energy efficiency and performance of a different configuration in the LTA task-coupling taxonomy normalized to the baseline scalar in-order core. The average improvement in energy efficiency for LTA engines with moderate temporal task coupling is 1.1–1.3×, whereas the improvement for LTA engines with extremely loose temporal task coupling is only 1.03–1.04×. The key point here is that, in a few cases, moderate temporal task coupling can non-trivially improve performance in both regular and irregular application kernels (e.g., *bilateral*, *sgemm*, *strsearch*) with a minimal energy overhead. Unfortunately, application kernels with inter-thread communication and fine-grain locking still exhibit lower energy efficiency regardless of the level of temporal task coupling.

These results seem to suggest that using a single chime group might be the most promising approach, but these benchmarks mostly fit within the L2 cache. Looser temporal task coupling will likely result in improved performance with irregular memory accesses that miss in the L2 cache or with longer L1/L2 cache miss penalties. Furthermore, these experiments were based on a 32-µthread LTA engine. Increasing the total number of µthreads might motivate looser temporal task coupling. The case studies in Section 7.4.2 and 7.4.3 validate these hypotheses. Overall, the conclusion of this study is that **moderate spatial and temporal task coupling can achieve high performance on both regular and irregular benchmarks with relatively high area and energy efficiency.** As such, *LTA-4/2x8/2* and *LTA-8/4x4/2* can be identified as the most promising configurations for both spatial and temporal task coupling in LTA engines.

7.3 Loop-Task Accelerator Platform

Based on the detailed design-space exploration of spatial and temporal task coupling thus far, a promising LTA engine configuration for accelerating both regular and irregular applications was identified in a single-core context. This section provides a more holistic evaluation of the LTA platform with respect to the 3P's. To this end, the LTA engine is examined in a multi-core context to ascertain if exploiting loop-task parallelism both across cores and within a core yields multiplicative effects in performance. In addition, a qualitative analysis of the productivity and portability of the LTA platform is offered.

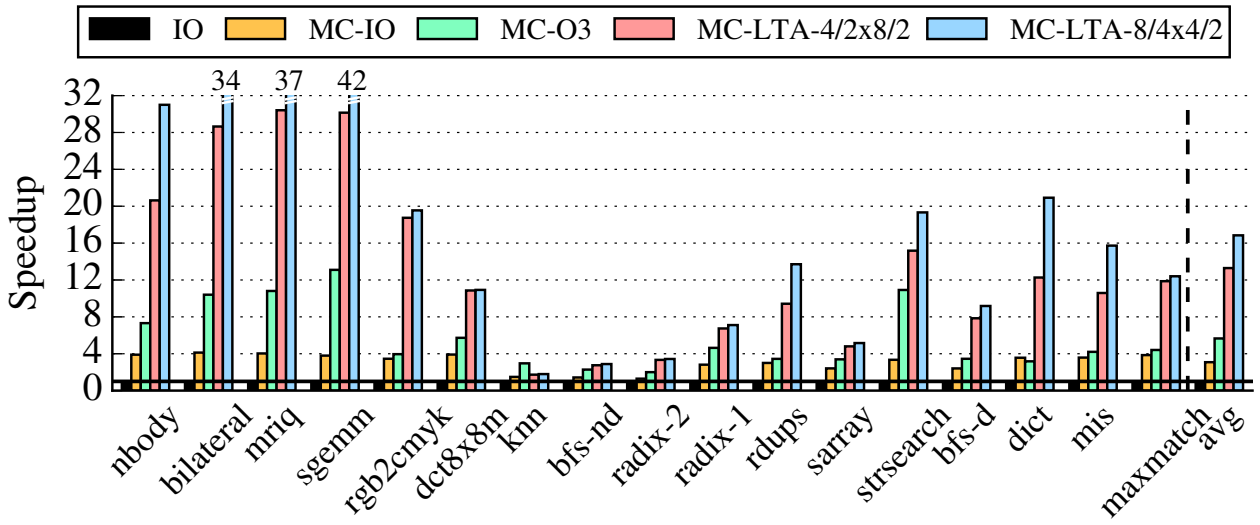


Figure 7.8: Performance of LTA Platform on Multi-Core System – Speedups of the most promising 4-lane and 8-lane 32-µthread LTA engines in a 4-core system normalized against a single in-order core for each application kernel. The speedups of the in-order and out-of-order baseline cores in a 4-core system are also shown for reference.

7.3.1 Combining Inter-Core and Intra-Core Mechanisms

Figure 7.8 shows the performance of a quad-core system with either an *LTA-4/2x8/2* or *LTA-8/4x4/2* LTA engine per core. The results confirm that the **LTA platform is able to achieve multiplicative effects from exploiting loop-task parallelism both across cores and within a core on both regular and irregular loop-task parallel applications.** Referring back to Figure 7.5, the average speedup of the most promising LTA engines is 4.2–5.3× over *IO*, and using the LTA runtime on *MC-IO* yields an average speedup of 3.1× (see Table 7.1). Both *MC-LTA-4/2x8/2* and *MC-LTA-8/4x4/2* are able to achieve ideal multiplicative speedups of 13–16×. Compared to *MC-IO*, the LTA platform is able to achieve average improvements of 5.5× in raw performance, 2.5× area-normalized performance, and 1.2× in energy efficiency. Even compared to a more aggressive *MC-O3*, the LTA platform improves raw performance by 3.0×, performance per area by 1.7×, and energy efficiency by 2.5×.

Recall that in Section 7.1, the single-core LTA engines were only marginally more area efficient than *O3*. This was because compared to *IO*, *O3* only increases area by 1.25× while achieving an average speedup of 2×, whereas the LTA engines increase area by 2–3× while achieving average speedups of 4–5×. However, the LTA engines in a multi-core system are noticeably more area efficient than *O3*. Although the area scales at the same rate for both baselines and the LTA engines,

the performance of the LTA engines scales at a significantly higher rate than either *IO* or *O3* because of the aforementioned multiplicative effect.

The results thus far suggest that **the eight-lane configuration, *LTA-8/4x4/2*, is a better final LTA engine candidate than the four-lane configuration, *LTA-4/2x8/2***. Although *LTA-8/4x4/2* has 8% more area and is 4% less energy efficient than *LTA-4/2x8/2*, the increase in average speedup from 13× to 17× is well worth the relatively minor overheads. In fact, the area-normalized performance of *LTA-8/4x4/2* is 18% higher than *LTA-4/2x8/2*.

7.3.2 Productivity and Portability

In terms of productivity, the application kernels used in the evaluation were ported from the corresponding TBB implementations *with minimal LTA-specific optimizations*. Since the LTA platform uses loop-tasks as the common abstraction across software and hardware, porting the benchmarks was a simple matter of replacing TBB's `parallel_for` constructs with LTA-specific own macros and linking in the proper libraries.

In terms of portability, a single implementation of the benchmark can be **written and compiled once, then executed using the LTA platform on a system with any combination of GPPs and homogeneous or heterogeneous LTA engines**. Of course, a limitation is that existing architectures would have to decode the `xpfor` instruction as a conventional `jalr` instruction to be compatible with the LTA platform.

Let us revisit the challenges with productivity and portability in the application development flow discussed in Chapter 2. AVX was dependent on eliminating control flow and aligning data structures to enable coalesced memory accesses. The LTA engine is designed to handle control divergence within a task group and memory accesses can be coalesced regardless of alignment. TBB required manually setting a grain size to determine an optimal task size. The LTA runtime and TMU in the LTA engine work together to dynamically determine the optimal task size for the available LTA engine. Combining TBB with AVX resulted in interference between the disjoint abstractions for exploiting loop-task parallelism across and within cores. The LTA platform uses loop-tasks as the common abstraction. MIC had different tuning parameters from TBB even when using the same software framework and struggled to achieve resource-proportional performance for both regular and irregular applications. The LTA platform can use the same implementation for GPPs or any LTA engine, and is capable of achieving resource-proportional speedups for diverse

loop-task parallelism. GPGPU had a vastly different offload programming model and required extensive manual optimizations for irregular applications. The LTA platform uses a native programming model, loop-task functions have no limitations on argument types (e.g., array of structs are supported), and does not require different optimizations for irregular applications to achieve high performance. Although further LTA-specific optimizations might improve performance, **the key point is that they are not necessary to extract high performance from the LTA platform for both regular and irregular loop-task parallel applications**.

7.4 Loop-Task Accelerator Case Studies

The following case studies are meant to provide deeper insights into the impact of several configurable parameters in the LTA engine template. To this end, the results presented here will focus on smaller subsets of LTA engine configurations and application kernels that serve to highlight these insights.

7.4.1 Impact of Shared LLFUs on Spatial Task-Coupling

The task coupling taxonomy presented in this thesis assumed that looser spatial task coupling would always be accompanied by more sharing of LLFUs across lane groups, but resource sharing is in fact a dimension that is orthogonal to task coupling.

Figure 7.9 shows the performance of eight-lane 32-µthread LTA engines in a single-core system with variable spatial task coupling *without* any sharing of LLFUs across lane groups normalized to the performance of the baseline IO core for each application kernel. Note that the memory port sharing is unchanged from before (i.e., single I$ port, scaling D$ ports). The results show that **looser spatial task coupling generally yields better performance on both regular *and* irregular application kernels when each lane always has a dedicated set of LLFUs**. This is not surprising, since the primary performance benefit of tighter spatial task coupling seen in Section 7.1 was the increased LLFU bandwidth. Although looser spatial task coupling without LLFU sharing certainly limits the corresponding savings in area, the performance benefits are great enough that the average area-normalized performance is still slightly higher than tighter spatial task coupling.

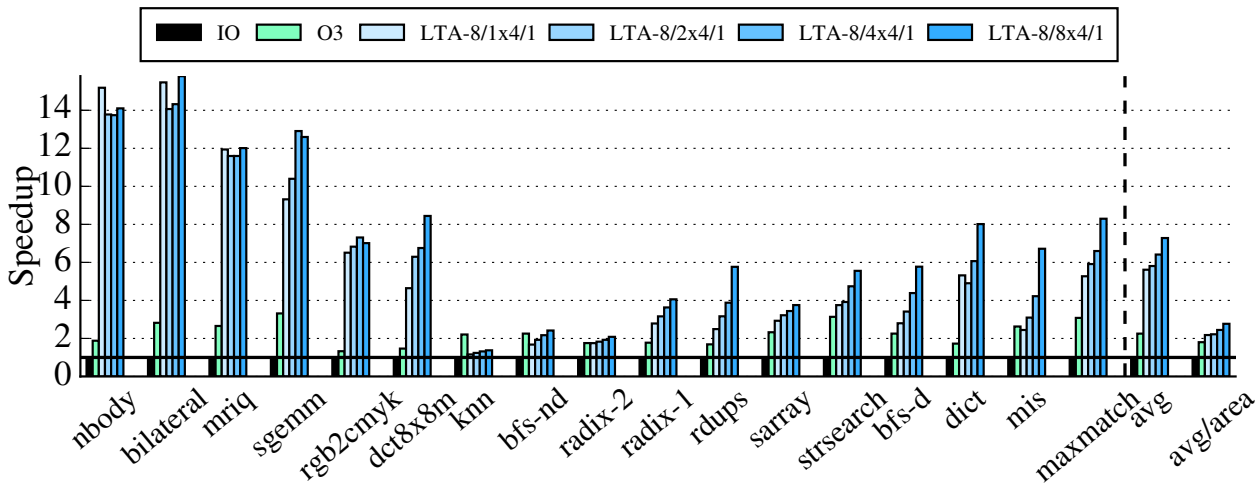


Figure 7.9: Variable Spatial Task-Coupling with No LLFU Sharing – Speedups of 8-lane 32-µthread LTA engines with variable temporal task coupling and no LLFU sharing normalized against the baseline scalar in-order core for each application kernel. This means that there is always a dedicated LLFU for each lane regardless of the level of spatial task coupling.

For instance, not sharing LLFUs increases the area of *LTA-8/8x4/1* by 23% but also increases the average raw performance from 5× to 7×.

This may seem to suggest that looser spatial task coupling without LLFU sharing is a more promising direction for LTA engines, but energy efficiency remains as the key weakness of looser spatial task coupling. Not sharing LLFUs does not change the fact that looser spatial task coupling reduces the number of µthreads across which the instruction fetch and frontend access are amortized. **Even without LLFU sharing, tighter spatial task coupling is able to achieve the same performance as looser spatial task coupling with substantially higher energy efficiency on regular applications.** Furthermore, looser spatial task coupling in conjunction with looser temporal task coupling is still more susceptible to bottlenecks at the shared I$ port. Separating I$ port sharing from task coupling by provisioning per-lane-group I$ ports can address this issue at the cost of higher area and energy due to the more complex I$ crossbar networks.

A separate experiment with no D$ port sharing, but standard I$ port and LLFU sharing was also conducted. In this case, the performance benefits of using a dedicated D$ port per lane regardless of spatial task coupling was minimal. Specifically, using dedicated D$ ports yielded a maximum improvement of 5% compared to sharing D$ ports, which was not enough to offset the increase in area and energy due to the more complex D$ crossbar network.

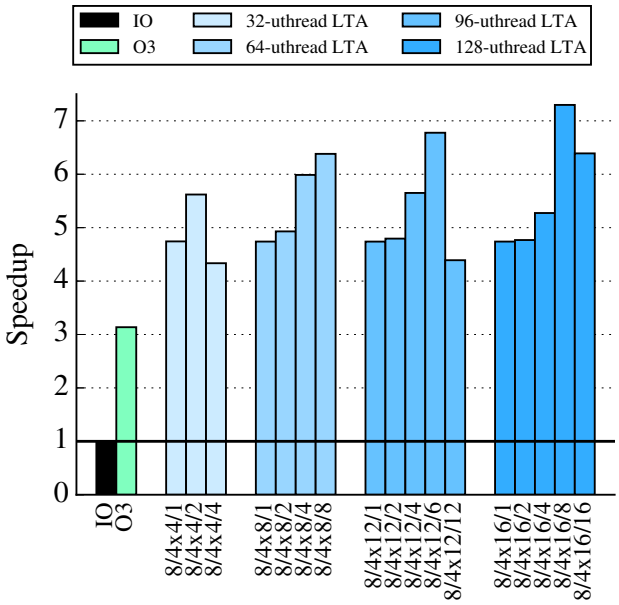


Figure 7.10: Variable µthread Count and Temporal Task-Coupling – Speedups of 8-lane LTA engines with 32, 64, 96, and 128 total µthreads with variable temporal task coupling normalized against the baseline scalar in-order core for *strsearch*.

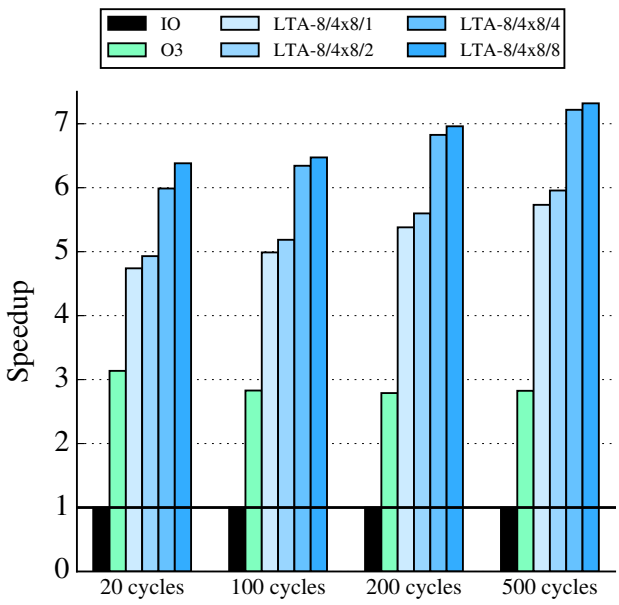


Figure 7.11: Variable Memory Latency and Temporal Task-Coupling – Speedups of 8-lane 64-µthread LTA engines with variable temporal task coupling normalized against the baseline scalar in-order core for *strsearch* using L1 miss latencies ranging from 20–500 cycles.

7.4.2 Impact of µthread Count on Temporal Task-Coupling

The conclusions form the design space exploration in Section 7.2 was that moderate temporal task coupling was desirable to balance the benefits of hiding more microarchitectural latencies with the overheads of I$ conflicts and decreased IU utilization. However, only a few application kernels exhibited any benefit from temporal task coupling and the resulting speedups were relatively minor. The goal of the next two case studies is to examine reasonable scenarios in which these overheads can be minimized in order to ascertain the true potential of looser temporal task coupling.

One way to address the overhead of I$ conflicts is to increase the total number of µthreads in an LTA engine. The performance impact of I$ conflicts was not readily apparent when only exploring looser spatial task coupling. The bottleneck only manifested when looser temporal task coupling was also introduced. This was a consequence of having many, smaller task groups. Assuming the same number of lane groups, having many chime groups is theoretically preferable. It provides more opportunities to hide microarchitectural latencies. The problem is that when the number of

chimes in each chime group is too small, redundant instruction fetches both across and within a lane group increases the pressure on the shared I$ port.

Figure 7.10 shows the performance of single-core LTA engines with 32, 64, 96, and 128 total µthreads, each with varying temporal task coupling, on *strsearch*. In the spirit of a deeper dive, *strsearch* is used to understand the trends across a much larger number of configurations since it yielded the most benefit from looser temporal task coupling in the main evaluation. All LTA engines have eight lanes and four lane groups to reflect the same spatial task coupling configuration selected in Section 7.1. The results show that **increasing the total number of µthreads allows for a degree of temporal task coupling that achieves a more optimal balance between the number of task groups and the size of the task group**. Comparing the best temporal task coupling configuration within each group of same-µthread-count LTA engines, from *LTA-8/4x4/2* to *LTA-8/4x8/4* to *LTA-8/4x12/6* to *LTA-8/4x16/8*, the percent of execution time stalled due to I$ misses/conflicts decreases from 49% to 40% to 30% to 22%, respectively. Compared to more tightly coupled configurations, these configurations are also able to reduce the percent of execution time stalled due to LLFU or D$ misses/conflicts from 15% to 3%. Note that in most cases, extreme loose temporal task coupling is not desirable, as a single-chime chime group makes it very difficult to keep all IUs busy unless there is significant back-pressure on the frontend. Of course, increasing the total µthread count comes at the cost of a notable increase in area from adding more register file contexts to the LTA engine.

7.4.3 Impact of Memory Latency on Temporal Task-Coupling

Even though LTA engines are able to dispatch an instruction to each of the three IUs within a lane group every cycle, it can be difficult to saturate the dispatch bandwidth since LTA engines can only fetch one instruction per cycle. Tighter temporal task coupling can more easily keep all of the IUs busy in spite of this because each dispatched instruction keeps a single IU busy for multiple cycles. However, looser temporal task coupling may not be able to keep all the IUs busy unless there is significant back-pressure on the frontend that causes instructions to be queued up for simultaneous dispatch at a later time. Because the datasets used in the evaluation are small enough that they do not cause many misses in the L2 cache, thus reducing opportunities for instructions to be queued up due to back-pressure, the corresponding results may not be capturing the full

potential of looser temporal task coupling. As such, the goal of this case study is to examine the impact of longer memory latencies on the efficacy of looser temporal task coupling.

Figure 7.11 shows the performance of the single-core eight-lane 64-µthread LTA engines with variable temporal task coupling examined in the previous case study normalized to the performance of the baseline in-order core on *strsearch*, using an L1 D$ with increasing miss penalties ranging from 20 (nominal) to 500 cycles. Obviously this is not realistic, but the hope is to emulate the impact of using larger datasets. The results do not seem to indicate that increasing the memory latency improves the efficacy of looser temporal task coupling, at least on the surface. For instance, the relative speedup of *LTA-8/4x8/1* compared to *LTA-8/4x8/4* is roughly the same regardless of the memory latency. However, a deeper comparison of *LTA-8/4x-8/1* to *LTA-8/4x8/4* reveals that the percent of execution time stalled due to D$ misses/conflicts decreases from 10% to 2% for a memory latency of 20 cycles, 12% to 3% for 100 cycles, 15% to 2% for 200 cycles, and 25% to 4% for 500 cycles. These numbers seem to suggest that looser temporal task coupling is indeed able to reduce the sensitivity of the LTA engine to longer memory latencies. Despite these benefits, the performance is ultimately bottlenecked by the instruction fetch which is exacerbated by looser temporal task coupling. For the same comparison, the percent of execution time stalled due to I$ misses/conflicts increases from 34% to 40% for a memory latency of 20 cycles, 33% to 39% for 100 cycles, 33% to 39% for 200 cycles, and 26% to 34% for 500 cycles. It is also interesting to note that compared to the baselines, *IO* and *O3*, LTA engines are generally less sensitive to longer memory latencies due to multiple chimes hiding microarchitectural latencies for both tight and loose temporal task coupling.

Given these insights, for systems more prone to D$ bottlenecks, it may be preferable to have slightly tighter spatial task coupling (but still moderate) to offset the instruction fetch bottleneck while still being able to reap the benefits of moderate temporal task coupling. Another possibility is to increase the instruction fetch bandwidth at the cost of increased area and energy due to the I$ crossbar network. In fact, these case studies shed some light on why GPGPUs have very large µthread counts (e.g., over a thousand CUDA threads per SIMT engine vs. tens of µthreads per LTA engine), wider instruction fetch, and wider dispatch. This is especially important on GPGPUs, where the extreme temporal warp multithreading tends to thrash the cache and generate inefficient DRAM accesses, both of which create significant data memory bottlenecks.

CHAPTER 8
RELATED WORKS

This chapter describes a subset of the large body of related work on building 3P platforms. As mentioned before, the proposed LTA platform is certainly not the first attempt at addressing the 3P's, although it may be the first software/hardware co-design approach to address the 3P's by focusing on exploiting loop-task parallelism. Existing works have mainly focused on *software-centric* or *hardware-centric* to addressing the 3P's.

8.1 Software-Centric Approaches

Software-centric approaches innovate at the runtime, language, and compiler levels to enable some or all of the 3P's across multiple architectures, but are designed to be layered on top of existing hardware architectures. Such approaches do not directly advocate for any changes to the underlying hardware, thus they may struggle with performance on both regular and irregular applications, or lose efficiency due to a reliance on software translation to map the parallel abstraction to hardware.

Two popular examples include **OpenCL** [ope11] and **C++ AMP** [Som11], which expose *work-groups/tiles* as common abstractions at the programming API and runtime levels. These frameworks use offload programming models that introduce many of the challenges associated with CUDA described in Chapter 2. For instance, both C++ AMP and OpenCL operate at relative coarse granularities of parallelism (i.e., kernel launches) for GPGPUs as well as CMPs/MICs. However, C++ AMP makes kernel argument transfers easier (e.g., implicit data transfer using `array` classes) and OpenCL provides better support for combining multi-threading and vectorization on CMPs/MICs (e.g., automatically SIMD-aligned flattened vector classes, implicit vectorization compiler pass for eligible work-group sizes) [Rot11]. Unfortunately, successful auto-vectorization on both C++ AMP and OpenCL still requires dealing with eliminating control flow and restrictions on data transfer types. Although these frameworks make it easier to port between CMPs and GPGPUs compared to TBB, other architecture-specific optimizations are necessary to yield the highest performance. It is also worth noting that neither OpenCL nor C++ AMP have built-in support for dynamic work-stealing which can limit load balancing compared to alternatives such as TBB or Cilk++.

AMD's **Heterogeneous System Architecture** (HSA) [amd12b] builds off of OpenCL and C++ AMP to push the common abstraction of work-groups down to a virtual ISA layer called the HSA intermediate language (HSAIL). Unifying the abstraction at higher levels in the computing stack trades off efficiency for flexibility. In addition to the translation overhead, HSA still relies on multiple physical ISAs that map to different architectures, which can further hinder fine-grain dynamic work-stealing across heterogeneous resources beyond the coarse-grain parallelism exposed by OpenCL and C++ AMP. Although the HSA vision allows room to incorporate diverse accelerator architectures (like LTA engines), it does not specify a standard microarchitectural template that natively interprets the common abstraction, but rather requires external modifications to the compiler backend to translate this abstraction to a given architecture.

Domain-specific languages (DSLs), such as Halide [RKBA+13], OptiML [SLB+11], and Delite [BSL+11], represent another class of software-centric 3P platforms. DSLs use higher levels of abstraction specific to an application domain (e.g., image processing for Halide, machine learning for OptiML) to achieve high productivity and performance, at the cost of generality at the application level. Internal DSLs expose these abstractions at the programming API and runtime levels by using libraries to wrap domain-specific abstractions. They are generally easier to implement and do not require modifications to the compiler, but limit static analysis and optimizations on internal representations (IR) of applications. External DSLs further push these abstractions down to the compiler level by essentially implementing a truly separate programming language for the domain with its own compiler. They are much more difficult to implement but significantly improve performance. A hybrid approach uses language virtualization to generate a wide range of internal DSLs [LBS+11, CDM+10] that leverage meta-programming to perform static analysis and optimizations on IR to achieve high performance on multiple hardware architectures without requiring per-DSL compiler modifications. It should be noted that even DSLs sometimes require target-specific implementations that reduce portability. For example, Halide code for GPUs is often quite different than Halide code for CMPs.

External CUDA libraries, such as Theano [AAAea16] and Thrust [BH12], address the challenge of productivity on specifically on GPGPUs. Theano is a Python library to enable efficient, fast mathematical operations on matrices, that generates CUDA code. Theano is popular for mapping deep neural networks to GPGPUs and is a part of NVIDIA's cuDNN [cud16]. Thrust is essentially a port of the C++ standard template library for CUDA, and provides a useful selection

of parallel algorithms and data structures compatible with CUDA. Although both libraries indeed improve productivity, they do not address portability and primarily focus on mapping regular applications to GPGPUs. The LonestarGPU benchmark suite [BNP12] includes a shared worklist library that helps mapping of irregular applications such as graph algorithms to GPGPUs, but it still requires many programmer-months of manual optimizations to achieve the highest performance on a handful of applications.

As discussed earlier, **work-stealing runtimes** [BL99, BJK+96, FLR98, ACR13, CM08] are software-centric approaches to exploiting inter-core task parallelism, but do not extend the same abstraction to the inter-core level. Work-stealing runtimes can be language-based or library-based, and utilize continuation stealing or child stealing. Cilk++ is an example of a language-based continuation-stealing runtime, and TBB is an example of a library-based child-stealing runtime. Existing work-stealing runtimes are generally unaware of mechanisms for exploiting parallelism within a core, like vectorization on packed-SIMD units, which can often cause interference when combining both techniques. However, continuing work on vectorizing compilers [MGG+11] could help mitigate the challenges with successful auto-vectorization.

8.2 Hardware-Centric Approaches

Hardware-centric approaches innovate at the ISA and microarchitectural levels to enable high performance for diverse loop-task parallelism, but generally rely on existing software frameworks to expose parallel abstractions. Such approaches do not significantly change the overarching software, which can lead to low productivity and portability.

The LTA platform was inspired by many existing accelerators for exploiting regular loop-task parallelism. **MICs** [Kan15, Kan16] (which rely on packed-SIMD units [Hug15]) and **GPGPUs** [nvi16], as well as their challenges with respect to the 3P's, were discussed at length in Chapter 2. Other SIMT-like architectures like FG-SIMT [KLST13] and vector-threading [KBH+04] use tighter integration of GPPs and SIMT engines to enable finer-grain parallelism to be exploited, as well as focus on exploiting intra-warp parallelism rather than inter-warp parallelism (as in GPGPUs). The LTA engine is indeed a SIMT architecture like GPGPUs but have many key differences, including: configurable task-coupling in both time and space, significantly less threads per core (32 vs. 1536), elastic pipeline with out-of-order writeback, density-time execution, program-

counter-based reconvergence, tight integration with GPP, hosted task-based programming model (as opposed to offload), and software runtime. Of course, inspiration was taken from traditional vector-SIMD processors like the Cray [Rus78, cra93] and Tarantula [EAE+02] as well, which tend to be more efficient but at substantial cost to productivity and portability due to the use of explicit vector instructions.

Other hardware-centric approaches focus on mechanisms or architectures to better tolerate irregular loop-task parallelism. The XLOOPS accelerator [SIT+14] uses lanes that are loosely coupled in space with no coupling in time (only one thread per lane), and are tightly integrated with the GPP similar to FG-SIMT. Conservation cores [VSG+10] focus on improving energy efficiency rather than performance by attaching application-specific circuits specialized for executing common "hot spots" to GPPs in a multi-core system. There are a plethora of microarchitectures for SIMT architectures to more efficiently handle control-flow and memory-access divergence by dynamically changing the size of the warps [MTS10, FSYA09, RE12, FA11], or improving reconvergence of diverged warp fragments [DAM+11, Col11b]. Unfortunately, none of these approaches directly address productivity and portability concerns.

There are also hardware-centric approaches that are capable of adapting to different forms of loop-task parallelism by dynamically changing the degree of task coupling. For example, **variable warp sizing** [RJOK15] on GPGPUs leverages a similar intuition about task coupling in space to dynamically gang smaller warps into a bigger warp via the hardware scheduler to efficiently adapt to different forms of loop-task parallelism. Similarly, **temporal SIMT** [KDK+11] is a GPGPU-inspired architecture with eight latency-optimized cores (i.e., out-of-order superscalar GPPs) and dozens of throughput-optimized cores (i.e., SIMT engines), where each throughput-optimized core can dynamically configure the spatial task coupling of its eight lanes to adapt to control and memory-access divergence. In loosely coupled mode, each lane is capable of executing 64 threads that are tightly coupled in time, whereas in tightly coupled mode these threads are aggregated to one large task group. Another example of an architecture that can dynamically reconfigure spatial task coupling is AMD's Bulldozer [BBSG11]. Although capable of impressive speedups on a wide range of loop-task parallel applications, these architectures do not explore task coupling in time and do not address the productivity and portability concerns associated with CUDA/OpenCL. Another key challenge with such architectures is the dynamic reconfiguration overhead in performance, area, and energy.

Architectural support for fine-grain work distribution is another topic that pushes software abstractions down to hardware to specifically accelerate the distribution of tasks more than the execution of tasks. Carbon [KHN07] and asynchronous direct messages [SYK10] (ADM) provide special instructions to generate and steal tasks in a CMP. Both focus on more efficiently exploiting inter-core parallelism without addressing intra-core parallelism. ADM improves upon Carbon by allowing for software-programmable work distribution algorithms which improves productivity, but neither address the challenges of portability across different architectures. Hardware worklists in GPGPUs [KB14] exposes a shared worklist abstraction often used in graph algorithms to hardware, allowing for more efficient distribution of work both across cores and within a core. However, this abstraction is specific to irregular applications, and although productivity is better than default software worklists, this thesis does not address challenges with portability either.

CHAPTER 9
CONCLUSIONS

This thesis proposed the LTA platform, a software/hardware co-design approach to addressing the challenge of improving productivity, portability, and performance in modern application development. Although some may argue that performance is what ultimately determines the success of a platform, an impressive potential performance may not be as attractive if extracting this performance on a single application requires many programmer-months of optimizations by an experienced programmer. Focusing solely on performance is further complicated by the fact that it is often difficult to predict which architecture will yield the best performance on a given application. Existing software-centric approaches that specifically seek to address the 3P's are promising, but their reliance on additional layers of software translation to map parallel abstractions to hardware leave room for improvement in terms of efficiency. Some of the information about the parallelism exposed in software will be lost during translation. As such, the LTA platform presented in this thesis sought to provide the 3P's without sacrificing efficiency by pushing a common parallel abstraction from software down the computing stack to the hardware.

The rest of this chapter summarizes the key points and primary contributions of this thesis, as well as discuss avenues for future work on this topic.

9.1 Thesis Summary and Contributions

This thesis began by describing an in-depth account of my personal experience developing, porting, and optimizing application kernels for an example application development flow spanning multiple software frameworks and hardware architectures. This case study suggested that although some existing platforms may provide one or two of the 3P's, only a few of them provide all of the 3P's, especially for both regular and irregular loop-task parallel applications. Furthermore, attempting to extract the highest performance out of a hardware substrate by exploiting parallelism across cores and within a core can unintuitively hurt the 3P's. Even platforms that specifically seek to address the 3P's unfortunately sacrifice efficiency by relying on software translation to map the abstractions exposed in software to hardware.

The thesis then discussed an early attempt at a hardware-centric approach to a 3P platform, the FG-SIMT architecture. FG-SIMT addresses productivity and portability by using data-parallel

threads as the common parallel abstraction in software and hardware, and a single ISA that can be executed on GPPs or FG-SIMT engines. Unfortunately, although FG-SIMT is able to achieve high performance and efficiency on regular applications, it struggled to achieve resource-proportional performance on more irregular applications. In addition, FG-SIMT was only explored in a single-core context and does not use a software runtime, so it is unclear how FG-SIMT would fare in a multi-core context with respect to the 3P's.

The core of the thesis detailed the software and hardware components of the LTA platform. In order to make it possible to efficiently encode the parallel abstraction in software and hardware, the LTA platform exploits a narrower form of task parallelism, called loop-task parallelism, and uses loop-tasks as the common parallel abstraction. Loop-tasks are exposed at the programming API, runtime, ISA, and microarchitectural levels. The LTA programming API and runtime are meant to replicate the productivity of TBB, while using loop-tasks to exploit parallelism both across cores and within a core. The LTA ISA allows the same implementation of an application to be mapped to either a GPP or any LTA engine to improve portability, without requiring major modifications to the compiler. The task-coupling terminology and taxonomy were introduced, which describes the spectrum of spatial and temporal task coupling. The LTA engine template is a clean-slate microarchitectural design that can be configured at design-time with variable spatial and temporal task coupling to target a diverse range of loop-task parallelism. LTA engines can accelerate loop-task execution by leveraging information about available loop-task parallelism explicitly encoded into the `xpfor` instruction.

A detailed evaluation of the LTA platform with respect to the 3P's was provided, which included a deep exploration of the impacts of spatial and temporal task coupling in the LTA engine on performance, area, and energy. The results indicated that moderate task coupling in both space and time is the most promising for guaranteeing high performance on both regular and irregular applications. Overall, the LTA platform in a multi-core system is able to achieve average improvements of 5.5× in raw performance, 2.5× in performance per area, and 1.2× in energy efficiency compared to a in-order multi-core baseline, and equally impressive improvements compared to a more aggressive out-of-order multi-core baseline. The LTA platform is able to achieve high performance and efficiency without sacrificing productivity and portability. For example, the same multi-threaded implementation for the CMP using the LTA software framework can be used without target-specific optimizations on systems with LTA engines.

The primary contributions of this thesis are reiterated below:

- Detailed analysis of the **3P challenges in modern application development flows** based on years of computer architecture experience.
- The **FG-SIMT** architecture, an area-efficient accelerator for regular loop-task parallelism with microarchitectural mechanisms for exploiting value structure, and a detailed evaluation of this architecture with respect to performance, area, and energy based on RTL and gate-level models.
- Software components of the LTA platform including a productive TBB-like **LTA programming API**, a task-based work-stealing **LTA-aware runtime** and lightweight **LTA ISA** extensions including a new `xpfor` instruction that explicitly encodes loop-tasks as a common parallel abstraction.
- A **task-coupling taxonomy** that describes the spectrum of how tasks can be coupled in space and time, with appropriate terminology.
- Hardware components of the LTA platform including an elegant microarchitectural template for the **LTA engine** that can be configured at design time with variable spatial and temporal task coupling to target diverse loop-task parallelism.
- Deep design space exploration of the impacts of task coupling in the LTA engine on performance, area, and energy, as well as a 3P evaluation of the LTA platform using a vertically integrated research methodology.

This thesis explores a subset of the larger, more difficult question of how computer architects can build and program parallel heterogeneous systems. On one hand, the application landscape is rapidly evolving and becoming more complex, necessitating more productive, general software programming frameworks. On the other hand, the architecture trends are pointing to hardware specialization for mitigating power limits at the physical level. What is not clear is these seemingly disparate trends can be reconciled. Regardless of what the answer might be, the best chance of success will require innovations across the entire computing stack, driving down from applications, to runtimes, to compilers, to ISAs, to microarchitectures, and to circuits.

9.2 Future Work

The LTA platform is a promising first step towards a true 3P platform, but as always, there is room for improvement. This section discusses several of the most interesting avenues for future work on the LTA platform.

Extending the parallel abstraction – Currently the LTA platform only exploits loop-task parallelism, but it may be possible to broaden the scope of the parallel abstraction without losing too much efficiency. For example, nested parallelism could be supported by allowing loop-tasks to dynamically generate more loop-tasks (i.e., `parallel_for` inside `parallel_for`). A naive way to achieve this might be to have each μthread execute the task-stealing routine in the runtime but the increase in concurrent task queue accesses may cause unnecessary memory contention as seen in [KB14]. Alternatively, the LTA engine could be modified to allow lane groups to inject newly generated loop-tasks into separate μtask queues that the TMU could access to partition and distribute more μtasks across the lane groups. The challenge here is in maintaining portability, as the `parallel_for` inside of the loop-task function needs to be compiled such that it can still be executed serially on the GPP. Another example would be supporting concurrent execution of loop-tasks for different parallel regions (i.e., "true" task parallelism). Of course this would require the GPP to continue execution after an `xpfor` instruction instead of stalling, and only `parallel_for`'s with no dependencies would be eligible for concurrent execution.

Heterogeneous LTA engines – One of the conclusions of this thesis is that LTA engines with moderate spatial and temporal coupling are the most promising for guaranteeing high performance on a diverse range of loop-task parallelism. Based on this insight, a homogeneous set of LTA engines with moderate task coupling were used to evaluate the LTA platform in a multi-core context. However, assuming that the LTA platform is extended to support a broader range of task parallelism as described above, more complex applications with multiple kernels that exhibit different forms of loop-task parallelism may benefit from a heterogeneous mix of LTA engines in a multi-core context. The idea here is that the LTA runtime would be able to *adaptively* schedule loop-tasks to the LTA engines for which they are most suited based on heuristics collected from the hardware. For instance, hardware counters in the LTA engines could keep track of the average number of active μthreads per cycle when executing loop-tasks, then this information could be aggregated by the runtime and used to schedule loop-tasks with a history of high control-flow irregularity to LTA

engines with more loose task coupling, and those with a history of low control-flow irregularity to LTA engines with more tight task coupling. In the ideal case, the LTA platform as a whole would be able to achieve close to the maximum speedups seen across all LTA engines in the evaluation for applications with sufficient diversity of loop-task parallelism.

Reconfigurability of the LTA Engine – Another interesting direction for future work is exploring the viability of dynamically reconfiguring LTA engines at boot time or run time, as opposed to a static configuration at design time. Dynamic reconfiguration is certainly an attractive idea that would ideally allow the LTA platform to achieve the maximum speedup possible on an LTA engine by configuring the spatial and temporal task coupling to be best suited for a given application. Reconfiguring the total number of µthreads would be more difficult since each µthread requires its own register file. However, there are several tradeoffs with this approach including resource over-provisioning and reconfiguration overheads. To enable reconfiguration at either boot time or run time, the LTA engine would need to be provisioned to support the maximum number of lane groups, the maximum number of chime groups, the maximum number of shared LLFUs, and a D$ crossbar that supports the maximum number of memory ports. This means that even if the LTA engine was configured with looser task coupling, it would not reap the benefits in area-normalized performance that was seen in the evaluation. The additional reconfiguration logic (e.g., crossbar to allow a single frontend to manage multiple lanes/chimes, RT/WQ/PFB with dynamic partitioning, reconfigurable bitwidth of bypass network, etc.) would not only further increase the area overhead of LTA engines, but could also increase the cycle time due to the increased hardware complexity. Depending on how fast reconfiguration can occur (e.g., 10s of cycles, 100s of cycles, 1000s of cycles) and how often reconfiguration is triggered (e.g., between loop-tasks, between parallel regions, between application kernels), reconfiguration may further offset the maximum speedup of a statically configured LTA engine as well. In order to prevent impacts on productivity by forcing reconfiguration decisions at the programmer level, dynamic reconfiguration will likely require an LTA runtime that can schedule loop-tasks based on heuristics similar to the one that would be used with a heterogeneous mix of statically configured LTA engines.

Multiple LTA Engines in a Single-Core System – An interesting design point in terms of area efficiency is a single GPP connected to multiple LTA engines. In this case, the GPP would act as a dedicated core task generator that offloads core tasks to available LTA engines for acceleration. This is a compelling alternative since it would ideally yield similar speedups of a comparable

multi-core system with LTA engines without needing the overhead of extra GPPs, system interconnect, and private L1 caches. However, the key tradeoff here is precisely the lack of cache capacity: multiple LTA engines sharing the same L1 cache system will likely cause significant thrashing. This could be addressed by allocating per-LTA-engine L1 caches (perhaps only necessary for the D$, assuming core tasks are from the same parallel region with the same form of loop-task parallelism), but this would greatly offset the area reductions since a majority of the area in a multi-core system is due to the caches. There is also no reason multiple LTA engines could not be connected a GPPs within a multi-core context as well. For instance, a two-core system with two LTA engines per core might be competitive with an four-core system with one LTA engine per core.

BIBLIOGRAPHY

[AAAea16] R. Al-Rfou, G. Alain, A. Almahairi, and et al. Theano: A Python framework for fast computation of mathematical expressions. *CORR*, abs/1605.02688, 2016.

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

[ACR13] U. A. Acar, A. Chargeéraud, and M. Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2013.

[AJ88] R. Allen and S. Johnson. Compiling C for Vectorization, Parallelization, and Inline Expansion. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 1988.

[amd11] HD 6900 Series Instruction Set Architecture, Rev 1.1. AMD Reference Guide, Nov 2011.

[amd12a] AMD Graphics Cores Next Architecture. AMD White Paper, 2012. `http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf`.

[amd12b] Heterogeneous System Architecture: A Technical Review. AMD White Paper, 2012. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf`.

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

[BBSG11] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas. Bulldozer: An Approach to Multithreaded Compute Performance. *IEEE Micro*, 31(2):6–15, Mar/Apr 2011.

[BH12] N. Bell and J. Hoberock. *Thrust: A Productivity-Oriented Library for CUDA*. Morgan Kaufmann, 2012.

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

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

[BNP12] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2012.

[Bol12] J. Bolaria. Xeon Phi Targets Supercomputers. *Microprocessor Report*, Sep 2012.

[BSL+11] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2011.

[CDM+10] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 2010.

[CDZ09] S. Collange, D. Defour, and Y. Zhang. Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations. *Workshop on Highly Parallel Processing on a Chip (HPPC)*, Aug 2009.

[CJMT10] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures (Patterns & Practices)*. Microsoft Press, 2010.

[CK11] S. Collange and A. Kouyoumdjian. Affine Vector Cache for Memory Bandwidth Savings. Technical Report ENSL-00622654, ENSL, Dec 2011.

[CL05] D. Chase and Y. Lev. Dynamic Circular Work-stealing Deque. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2005.

[CL08] B. W. Coon and J. E. Lindholm. System and Method for Managing Divergent Threads in a SIMD Architecture. US Patent 7353369, Apr 2008.

[CM08] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. *Int'l Symp. on Workload Characterization (IISWC)*, Sep 2008.

[Col11a] S. Collange. Identifying Scalar Behavior in CUDA Kernels. Technical Report HAL-00622654, ARENAIRE, Jan 2011.

[Col11b] S. Collange. Stack-less SIMT Reconvergence at Low Cost. Technical Report HAL-00622654, ARENAIRE, Sep 2011.

[cra93] CRAY T3D System Architecture Overview. Cray Research Inc. Referece Manual, 1993.

[cud16] NVIDIA cuDNN–GPU Accelerated Deep Learning. Online Webpage, 2016 (accessed Sep, 2016). `https://developer.nvidia.com/cudnn`.

[DAM+11] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Keer, H. Wu, and S. Yalamanchili. SIMD Re-Convergence at Thread Frontiers. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2011.

[Dem14] M. Demler. Movidius Eyes Computational Vision. Microprocessor Report, The Linley Group, Sep 2014. http://www.linleygroup.com/mpr/article.php?id=11279.

[DKH11] N. Dickson, K. Karimi, and F. Hamze. Importance of Explicit Vectorization for CPU and GPU Software Performance. *Journal of Computational Physics (JCP)*, 230:5383–5398, Jun 2011.

[DKYC10] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogenous Systems. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.

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

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

[EV96] R. Espasa and M. Valero. Decoupled Vector Architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 1996.

[EVS98] R. Espasa, M. Valero, and J. E. Smith. Vector Architectures: Past, Present, and Future. *Int'l Symp. on Supercomputing (ICS)*, Jul 1998.

[FA11] W. W. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

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

[FSYA09] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. on Architecture and Code Optimization (TACO)*, 6(2):1–35, Jun 2009.

[GKS13] S. Z. Gilani, N. S. Kim, and M. Schulte. Power-Efficient Computing for Compute-Intensive GPGPU Applications. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2013.

[Gwe14] L. Gwennap. Qualcomm Tips Cortex-A57 Plans: Snapdragon 810 Combines Eight 64-Bit CPUs, LTE Baseband. *Microprocessor Report*, Apr 2014.

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

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

[Hug15] C. J. Hughes. Single-Instruction Multiple-Data Execution. *Synthesis Lectures on Computer Architecture*, 2015.

[HVS+13] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. Shet, G. Chrysos, and P. Dubey. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on IntelÂő Xeon Phi Coprocessor. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2013.

[int11a] Intel OpenSource HD Graphics Programmer's Reference Manual, Vol 4, Part 2, Rev 1.0. Intel Reference Manual, May 2011.

[int11b] Introducing Intel Many Integrated Core Architecture. Intel Press Release, 2011. `http://www.intel.com/technology/architecture-silicon/mic`.

[int12] Intel SDK for OpenCL Applications: Optimization Guide. Intel Reference Manual, 2012. `http://software.intel.com/sites/landingpage/opencl/optimization-guide`.

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

[int15] Intel Threading Building Blocks. Online Webpage, 2015 (accessed Aug 2015). `https://software.intel.com/en-us/intel-tbb`.

[jav15] Java API: ForkJoinPool. Online API Documentation, 2015 (accessed Aug 2015). `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html`.

[JR13] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

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

[Kan16] D. Kanter. Xeon Phi 7200 Boots Up for HPC. Microprocessor Report, The Linley Group, Jul 2016. `http://www.linleygroup.com/mpr/article.php?id=11641`.

[KB14] J. Kim and C. Batten. Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

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

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

[KDY12] A. Kerr, G. Diamos, and S. Yalamanchili. Dynamic Compilation of Data-Parallel Kernels for Vector Processors. *Int'l Symp. on Code Generation and Optimization (CGO)*, Apr 2012.

[KHN07] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2007.

[KLST13] J. Kim, D. Lockhart, S. Srinath, and C. Torng. Microarchitectural Mechanisms to Exploit Value Structure in Fine-Grain SIMT Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

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

[LAS+09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2009.

[LBS+11] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31(5):42–53, Sep/Oct 2011.

[Lea00] D. Lea. A Java Fork/Join Framework. *Java Grade Conference*, Jun 2000.

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

[LKC+10] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2010.

[LLM89] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. Practical Fast 1-D DCT Algorithms with 11 Multiplications. *Int'l Conf. on Acoustics Speech and Signal Processing*, May 1989.

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

[LSB09] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Oct 2009.

[LWH10] L. Luo, M. Wong, and W. Hwu. An Effective GPU Implementation of Breadth-First Search. *Design Automation Conf. (DAC)*, Jun 2010.

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

[LZH+13] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh. Optimizing the MapReduce framework on Intel Xeon Phi coprocessor. *IEEE Int'l Conf. on Big Data (BIGDATA)*, Oct 2013.

[MBJ09] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches, 2009.

[MGG+11] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua. An Evaluation of Vectorizing Compilers. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2011.

[Mic09] Graphics Guide for Windows 7: A Guide for Hardware and System Manufacturers. Microsoft White Paper, 2009. `http://www.microsoft.com/whdc/device/display/graphicsguidewin7.mspx`.

[MLBP12] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A GPU Implementation of Inclusion-Based Points-to Analysis. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2012.

[MLNP+10] M. Mendez-Loj, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-Driven Optimizations for Amorphous Data-Parallel Programs. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2010.

[mpi13] Message Passing Interface (MPI) Standard. Online Webpage, 2013 (accessed Nov 17, 2013). `http://www.mcs.anl.gov/research/projects/mpi/standard.html`.

[MTS10] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2010.

[NBGS08] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar/Apr 2008.

[NBP13a] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus Topology-driven Irregular Computations on GPUs. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr 2013.

[NBP13b] R. Nasre, M. Burtscher, and K. Pingali. Morph Algorithms on GPUs. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2013.

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

[nvi15] CUDA C Best Practices Guide. NVIDIA White Paper, 2015. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide`.

[nvi16] NVIDIA Tesla P100. NVIDIA White Paper, 2016. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

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

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

[Oya99] Y. Oyanagi. Development of Supercomputers in Japan: Hardware and Software. *Parallel Computing*, 25(13–14):1545–1567, Dec 1999.

[PBV+13] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim. Tera-scale 1D FFT with low-communication algorithm and Intel Xeon Phi coprocessors. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2013.

[PHSJ13] S. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis. Exploring SIMD for Molecular Dynamics Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2013.

[pyp14] PyPy. Online Webpage, 2014 (accessed Sep 26, 2014). `http://www.pypy.org`.

[RE12] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2012.

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

[Rei12] J. Reinders. An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessors. Intel White Paper, 2012. `https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors.pdf`.

[RJOK15] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler. A Variable Warp Size Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, 2015.

[RKBA+13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2013.

[Rot11] N. Rotem. Intel OpenCL Implicit Vectorization Module. Presentation Slides, 2011. `http://llvm.org/devmtg/2011-11/Rotem_IntelOpenCLSDKVectorizer.pdf`.

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

[SBF+12] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, and H. V. Simhadri. Brief Announcement: The Problem Based Benchmark Suite. *Symp. on Parallel Algorithms and Architectures (SPAA)*, Jun 2012.

[SFS00] J. E. Smith, G. Faanes, and R. Sugumar. Vector Instruction Set Support for Conditional Operations. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2000.

[SGM+10] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, Z. Hu, and W.-M. W. Hwu. Efficient Compilation of Fine-Grained SPMD-Threaded Programs for Multicore CPUs. *Int'l Symp. on Code Generation and Optimization (CGO)*, Apr 2010.

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

[SLB+11] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. *Int'l Conf. on Machine Learning (ICML)*, Jun 2011.

[Som11] S. Somasegar. Targeting Heterogeneity with C++ AMP and PPL. MSDN Blog, Jun 2011. `http://blogs.msdn.com/b/somasegar/archive/2011/06/15/targeting-heterogeneity-with- c-amp-and-ppl.aspx`.

[SS00] N. Slingerland and A. J. Smith. Multimedia Instruction Sets for General Purpose Microprocessors: A Survey. Technical report, EECS Department, University of California, Berkeley, Dec 2000.

[SYK10] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2010.

[VSG+10] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2010.

[WLPA16] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2.1. UCB EECS Technical Report, 2016. `https://riscv.org/specifications/`.

[WWP09] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, Apr 2009.

[YKM+11] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A Fully Integrated Multi-CPU, GPU, and Memory Controller 32 nm Processor. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2011.