Enhancing Programmability, Portability, and Performance with Rich Cross-Layer Abstractions

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Nandita Vijaykumar

M.S., Electrical and Computer Engineering, Carnegie Mellon University
B.E., Electrical and Electronics Engineering, P.E.S. Institute of Technology

Carnegie Mellon University
Pittsburgh, PA

December 2019

To my loving parents, Porul and Vijay.

Acknowledgments

There have been many many people, more than I can name, who have contributed in very different ways to this thesis and to my academic journey over the last 6 years. This is but a brief reflection on their invaluable contributions.

First and foremost, I would like to thank my advisors, Prof. Onur Mutlu and Prof. Phillip B. Gibbons. Onur generously gave me all the resources, opportunities, guidance, and freedom I needed to grow and succeed. His unrelenting emphasis on clarity of thinking and writing and his indomitable work ethic was a constant source of inspiration and learning. I am immensely grateful for his unwavering support and confidence in me throughout the ups and downs of my PhD. I thank Phil for his tremendous generosity with his time and faith in me. His technical perspectives played a big role in widening the scope of my research and challenging my thinking. I am very grateful for the safe and stimulating environment Phil provided during my PhD journey. My advisors' constant encouragement and nurturing shaped me into the researcher that I am today.

I am grateful to my thesis committee members, Prof. James Hoe, Prof. Mattan Erez, and Prof. Wen-Mei Hwu, for their valuable feedback in making this thesis stronger and their support during my academic job search. I am very grateful to Prof. Hoe for all his advice and for providing the stimulating research environment in CALCM at CMU. I sincerely thank Prof. Todd Mowry and Prof. Chita Das for the research collaborations that strengthened this thesis and their invaluable support during my academic job search.

Graduate school was a long and lonely journey, and I am immensely grateful to the members of the SAFARI research group for all their companionship: I found my family away from home in SAFARI. I will always be grateful to Kevin Hsieh for the many long hours of brainstorming, his endless support and positivity, and our invaluable research synergy. Gennady Pekhimenko was an irreplaceable mentor, friend, and big brother, who stood by me right from the beginning. I am very thankful to Samira Khan for being a great friend, mentor, and confidante. Her unrelenting high standards pushed me to succeed and her support kept me sane during my hardest times. I thank Hongyi Xin for being a wonderful friend and an unending source of laughter and support. I thank Vivek Seshadri for being a great mentor and a role model to aspire to. I thank Lavanya Subramanian for her support, kindness, and warmth throughout my PhD. These lifelong friendships are one of the most valuable outcomes of my PhD and I cannot express in words my gratitude to them.

I am very grateful to all the other members of SAFARI at CMU and ETH for being great friends and colleagues and providing a stimulating research environment: Yoongu Kim for setting high

standards in the group and the many intellectual discussions; Donghyuk Lee for his kindness, technical expertise, and all his support during my Nvidia internship; Saugata Ghose for his support, positivity, encouragement, and all the enjoyable dinners; Rachata Ausavarungnirun for the technical discussions and for being a kind friend; and Juan Gomez Luna for helping me settle into ETH. I also thank Justin Meza, Kevin Chang, Yixin Luo, Amirali Boroumand, Hasan Hassan, Minesh Patel, Damla Senol, Jeremie Kim, Giray Yaglikci, Nika Mansouri, Mohammed Alser, and the other SAFARI members at CMU and ETH.

I thank the members of CALCM and PDL at CMU and the Systems Group at ETH for all the technical feedback and for creating a rich stimulating research environment. I also thank my collaborators at Penn State, Prof. Mahmut Kandemir, Adwait Jog, and Onur Kayiran for the research collaborations.

I am grateful to all the students and mentees who worked closely with me over the years: Abhilasha Jain, Ashish Shrestha, Konstantinos Kanellopoulos, Mehrshad Lotfi, Diptesh Majumdar, Abhishek Bhowmick, Nisa Bostanci, and Ataberk Olgun. This thesis would not be possible without them.

I would like to thank all my internship mentors and managers for their guidance during my many internships: Eiman Ebrahimi, Steve Keckler, Dave Nellans, Chris Wilkerson, Kingsum Chow, Alaa Alameldeen, Zeshan Chisti, Olatunji Ruwase, and Trishul Chilimbi. I sincerely thank Intel, Nvidia, and Microsoft for these opportunities.

I gratefully acknowledge the generous support from Facebook, Google, IBM, Intel, Microsoft, Qualcomm, VMware, Nvidia, Samsung, Alibaba, the Intel Science and Technology Center for Cloud Computing, the Semiconductor Research Corporation and NSF (grants 0953246, 1065112, 1205618, 1212962, 1213052, 1302225, 1302557, 1317560, 1320478, 1320531, 1409095, 1409723, 1423172, 1439021, and 1439057). I am also grateful for the support from the Benjamin Garver Lamme/Westinghouse fellowship at CMU.

I sincerely thank my mentors and colleagues at AMD for introducing me to computer architecture and PhD research: Kanishka Lahiri, Hrishikesh Murukkathampoondi, Anasua Bhowmick, Ramkumar Jayaseelan, and Subhash Kunnoth.

I thank my family and friends for all their love and support during this journey. I am tremendously blessed to have a big and wonderful family and amazing friends. This thesis would in no way be possible without them and I hope I have made them proud. I especially thank Santosh Harish and Manzil Zaheer for their support and companionship. My heartfelt gratitude is to Neha Singh for our decades-long friendship and her constant support and care. I will forever be grateful to my aunt, Akila Suresh, for her unending love and care and for making my travel-filled PhD so much easier. I thank my sister, Dipti, for being a constant source of love, patience, and support throughout graduate school.

Finally, I thank my parents, Porul and Vijay. I will forever be indebted to them for their unconditional love, unwavering support, and the inspiration to pursue my academic dreams. They cannot know the immense role they played over the last few years and words cannot express my gratitude to them. I will forever be grateful for everything they have given me and enabled me to do, and this thesis is dedicated to them.

Abstract

Programmability, performance portability, and resource efficiency have emerged as critical challenges in harnessing complex and diverse architectures today to obtain high performance and energy efficiency. While there is abundant research, and thus significant improvements, at different levels of the stack that address these very challenges, in this thesis, we observe that we are fundamentally limited by the interfaces *and abstractions between the application and the underlying system/hardware—specifically, the hardware-software interface. The existing narrow interfaces poses two critical challenges. First, significant effort and expertise are required to write high-performance code to harness the full potential of today's diverse and sophisticated hardware. Second, as a hardware/system designer, architecting faster and more efficient systems is challenging as the vast majority of the program's semantic content gets lost in translation with today's hardware-software interface. Moving towards the future, these challenges in programmability and efficiency will be even more intractable as we architect increasingly heterogeneous and sophisticated systems.*

This thesis makes the case for rich low-overhead cross-layer abstractions as a highly effective means to address the above challenges. These abstractions are designed to communicate higher-level program information from the application to the underlying system and hardware in a highly efficient manner, requiring only minor additions to the existing interfaces. In doing so, they enable a rich space of hardware-software cooperative mechanisms to optimize for performance. We propose 4 different approaches to designing richer abstractions between the application, system software, and hardware architecture in different contexts to significantly improve programmability, portability, and performance in CPUs and GPUs: (i) *Expressive Memory: A unifying cross-layer abstraction to express and communicate higher-level program semantics from the application to the underlying system/architecture to enhance memory optimization;* (ii) *The Locality Descriptor: A cross-layer abstraction to express and exploit data locality in GPUs;* (iii) *Zorua: A framework to decouple the programming model from management of on-chip resources and parallelism in GPUs;* (iv) *Assist Warps: A helper-thread abstraction to dynamically leverage underutilized compute/memory bandwidth in GPUs to perform useful work. In this thesis, we present each concept and describe how communicating higher-level program information from the application can enable more intelligent resource management by the architecture and system software to significantly improve programmability, portability, and performance in CPUs and GPUs.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficient management of compute and memory resources, today and in the future, is as critical as ever to maximize system performance and energy efficiency. Important goals when it comes to effectively managing system resources include: ***programmability***, to minimize programmer effort in optimizing for performance; ***portability*** of software optimizations across architectures with different resources/characteristics and in the presence of co-running applications that share resources; and ***resource efficiency***, to maximize utilization of all available resources and effectively leverage features of a given architecture.

The importance of managing the diverse compute/memory resources in an easy-to-program, portable, and efficient manner has inspired a large body of research in programming languages, software frameworks, compilers, and architectures. We, however, argue that ever-growing complexity at each level of the stack *cannot*, in isolation, fully achieve the three-fold goal of programmability, portability, and resource efficiency: we are fundamentally constrained by current cross-layer abstractions that are *not* designed to optimize for these goals. The levels of the computing stack—***application*** *(the application/programming model),* ***system*** *(OS/runtime/compiler), and* ***architecture*** *(the hardware architecture)*—still interact with the traditional interfaces and abstractions (e.g., virtual memory, instruction set architecture (ISA)), which were primarily designed to convey *functionality*, rather than for the efficient management of resources which is critical for *performance.*

1.1 Motivation: Narrow Hardware-Software Interfaces Constrain Performance, Programmability, and Portability.

The existing interfaces have two important implications that make achieving programmability, portability, and resource efficiency significantly challenging:

Implication 1: *Diminishing returns from hardware-only approaches.* The existing interfaces between layers of the computing stack, e.g., the instruction-set architecture (ISA) and virtual memory, strip any application down to the basics of what is required to execute code correctly: a sequence of instructions and memory accesses. Higher-level information—even the simple notion of data structures, their access semantics, data types, and properties—are all lost in translation. In other words, there is a large *semantic gap* between the application and the underlying system/hardware. Today, system designers *work around* this gap and the vast majority of optimizations in the hardware architecture (optimizing caches, memory, coherence, computation, and so on) try to predict/infer program behavior. However, these approaches are fundamentally limited by how much application information is visible. Hence, we are seeing diminishing returns from hardware-only approaches in general-purpose computing. There have been *numerous* proposals for cross-layer optimizations and hardware-software cooperative mechanisms, but since they require full-stack changes for a *single* optimization, they are challenging to adopt.

Implication 2: ***Application/system manages low-level hardware resources with limited visibility and access.*** With the existing narrow hardware-software interface, the architecture is heavily constrained in resource management. Thus, we rely on the application software to do much of the heavy lifting in optimizing code to the specifics of each architecture. The application and system software need to be aware of low-level system resources, and manage them appropriately to tune for performance. GPU programming is challenging task today, as many hardware resources that are critical for performance need to be directly allocated and managed by the programmer. This causes challenges in programmability, portability, and resource efficiency. The software may *not* always have visibility into available resources such as available cache space (e.g., in virtualized environments) and even if it does, software has little access to many hardware features that are critical when optimizing for performance (e.g., caching policies, memory mapping). Furthermore, software *cannot* easily adapt to changes in the runtime environment (e.g, co-running applications, input data).

1.2 Our Approach: *Rich Cross-Layer Abstractions to Enable Hardware-Software Cooperative Mechanisms*

In this thesis, we propose *unifying* cross-layer abstractions to bridge the semantic gap between the application and underlying system/hardware. These abstractions directly communicate higher-level program information, such as data structure semantics, parallelism, and data access properties, to the lower levels of the stack: compiler, OS, and hardware. This information is conveyed by the programmer using our programming abstractions or automatically inferred using software tools. These abstractions enable a rich space of hardware-software cooperative mechanisms to improve performance. The abstractions are expressive enough to convey a wide range of program information. At the same time, they are designed to be highly portable and low overhead, requiring only small additions to existing interfaces. This makes them *highly practical* and easy to adopt. We look at 4 different contexts in CPUs and GPUs where new rich cross-layer abstractions enable new hardware-software cooperative mechanisms that address challenges in performance, programmability, and portability.

Cross-layer cooperative mechanisms are highly effective because providing the compiler, OS, and hardware a *global* view of program behavior, ahead of time, enables these components to actively optimize resources accordingly. For example, we demonstrated that knowledge of data structures and their access semantics enables the OS to place data intelligently in memory to maximize locality and parallelism (Chapter 2). Similarly, knowledge of the locality semantics of GPU programs enables the hardware thread scheduler to co-locate threads that share data at the same core to enhance data locality (Chapter 3). These are just two simple examples in a large space of hardware-software codesigns in general-purpose processors, including numerous previously-proposed cross-layer optimizations.

In this thesis, we propose cross-layer abstractions, along with the new hardware-software mechanisms that they enable in different contexts in CPUs and GPUs, including: (1) rich programming abstractions that enable expression of application-level information and programmer intent, completely agnostic to the underlying system and hardware; (2) a cross-layer system that efficiently integrates the OS/runtime system and compiler, enabling these components to flexibly tap into a rich reservoir of application information; and (3) a low-overhead implementation in the hardware architecture.

This thesis, hence, provides evidence for the following thesis statement:

A rich low-overhead cross-layer interface that communicates higher-level application information to hardware enables many hardware-software cooperative mechanisms that significantly improve performance, portability, and programmability.

1.3 Key Benefits

While there is a wide space of research opportunities into what a rich cross-layer abstraction enables, the benefits we demonstrated are detailed below.

(1) Enabling intelligent and application-specific cross-layer optimizations in the system/hardware: In addition to the aforementioned examples, more generally, communicating program information enables more intelligent cross-layer optimizations to manage caches, memory, coherence, computation, and so on. Examples of this information include semantics of how a program accesses its data structures and properties of the data itself. The system/hardware can now effectively *adapt* to the application at runtime to improve overall system performance (Chapter 2, 3): For example, the Locality Descriptor (Chapter 3) leverages knowledge of an application's data access properties to enable coordinated thread scheduling and data placement in NUMA (non-uniform memory access) architectures. A richer cross-layer abstraction also enables *customized optimizations*: Expressive Memory (Chapter 2) in CPUs enables adding specialization in the *memory hierarchy* to accelerate specific applications/code segments. For example, the abstraction enables transparently integrating a customized prefetcher for different types of data structures and their access semantics (e.g, hash table, linked list, tensor). Similarly, with Assist Warps (Chapter 5), we demonstrate customized hardware data compression in GPUs, specific to the data layout of any data structure.

(2) Enabling hardware to do more for productivity and portability: In addition to performance, access to program semantics and programmer intent enables hardware to improve productivity and portability. We demonstrate (Chapter 2-5) that architectural techniques can improve the performance portability of optimized code and reduce the effort required to write performant code when privy to the semantics of software optimizations (e.g., cache tiling, graph locality optimizations). For example, by communicating to hardware the tile size and access pattern when using a cache-blocking optimization (e.g., in stencils and linear algebra), the cache can intelligently coordinate eviction policies and hardware prefetchers to retain as much as possible of the tile in the cache, irrespective of available cache space (Chapter 2). This avoids cache thrashing and the resulting performance cliffs that may occur when the available cache is less than what the program was optimized for—thus improving *performance portability*. Similarly, performance cliffs are rampant in GPU programs, requiring great precision in tuning code to efficiently use resources such as scratchpad memory, registers, and the available parallelism. We demonstrate how enabling more intelligent hardware can significantly alleviate performance cliffs and enhance productivity and portability in GPUs via careful management of resources at runtime (Chapter 4, 5).

1.4 Overview of Research

We propose 4 different approaches to designing richer abstractions between the application, system software, and hardware architecture, which we briefly describe next.

1.4.1 Expressive Memory [336]: A rich and low-overhead cross-layer abstraction in CPUs to enable hardware-software cooperative mechanisms in memory (Chapter 2)

In this work, we proposed a new cross-layer interface in CPUs, Expressive Memory (XMem), to communicate higher-level program semantics from the application to the operating system and architecture. To retain the existing abstraction as is, we instead associate higher-level program semantics with *data* in memory. Examples of this information include: *(i)* data structure semantics and access properties; *(ii)* properties of the data values contained in the data structures, e.g., data sparsity, data type; *(iii)* data locality. The OS and any component in hardware (e.g., cache, memory controller) can simply query the XMem system with any memory address to retrieve the associated program information. XMem was architected such that the entire interface is simply defined by three key operators that are recognized by all levels of the stack—`CREATE`, `MAP`, and `ACTIVATE`. These operators enable rich expressiveness to describe more complex semantics in any programming language. These operators can be demonstrably implemented with low overhead and are portable across different architectures. We demonstrated significant benefits in enabling the system/hardware to do more for performance, productivity, and portability and specializing for different applications. The abstraction was designed to flexibly support many cross-layer optimizations. To demonstrate its utility and generality, we presented 9 different use cases.

1.4.2 The Locality Descriptor [333]: Expressing and leveraging data locality in GPUs with a rich cross-layer abstraction (Chapter 3)

While modern GPU programming models are designed to explicitly express *parallelism*, there is no clear way to express semantics of the program itself: i.e., how the thousands of concurrent threads access its data structures. We designed a rich cross-layer abstraction that describes how the hierarchy of threads in the GPU programming model access each data structure and the access semantics/properties of the data structures themselves. We then leverage this abstraction to significantly improve the efficacy and ease with which we can exploit *data locality* in modern GPUs—both *reuse-based* locality, to make efficient use of the caches, and *NUMA* locality, to place data and computation in near proximity in a non-uniform memory access (NUMA) system.

Exploiting data locality in GPUs today is a challenging but elusive feat both to the programmer and the architect. Software has no access to key components (such as the thread scheduler) and hardware misses key information such as: which threads share data? We designed a powerful abstraction (named by its use case: the Locality Descriptor) that communicates the locality semantics of any application to the hardware. This enables hardware to transparently coordinate many locality optimizations such as co-scheduling threads that share data at the same core and placing data close to the threads that use it. The programming interface is designed to be seamlessly integrated into modern GPU programming models (e.g., CUDA and OpenCL). The abstraction's semantics are defined such that it can be automatically generated via software tools and is highly portable, making no assumptions about the underlying architecture.

1.4.3 Zorua [334, 335]: Decoupling the GPU programming model from hardware resource management (Chapter 4)

In accelerators, such as modern day GPUs, the available parallelism as well as the memory resources need to be *explicitly* managed by the programmer. There exists no powerful abstraction between the architecture and the programming model, and the management of many hardware resources is tied to the programming model itself. This leads to underutilization of resources (and hence, significant loss in performance) when the application is not well tuned for a given GPU. Even when an application is perfectly tuned for one GPU, there can still be a significant degradation in performance when running the same program on a different GPU. Furthermore, it is unclear how to write such architecture-specific programs in virtualized environments, where the same resources are being shared by multiple programs.

To achieve the three-fold goal of enhanced programmability, portability, and resource efficiency, we designed a new hardware-software cooperative framework, ***Zorua***, to enhance the interface between the programming model and architecture for the management of several critical compute and memory resources. Zorua *decouples* the resource management as specified by the programming model and the actual utilization in the system hardware by effectively *virtualizing* each of the resources (register file, scratchpad memory, and thread slots) in hardware. Zorua also communicates fine-grained information regarding the application's future resource requirements at regular intervals to the hardware with a new hardware-software interface. This virtualization, along with the rich interface, enables the hardware to intelligently and dynamically manage these resources depending on the application behavior and makes the performance of any program far less sensitive to the software-provided resource specification. High performance is thus made far easier to attain and performance is portable across generations of the architecture which may have varying amounts of each resource.

1.4.4 Assist Warps [337, 338]: A cross-layer abstraction and hardware-software framework to leverage idle memory and compute resources (Chapter 5)

In modern throughput-oriented processors, *even with highly optimized code*, imbalances between the compute/memory resources requirements of an application and the resources available in hardware can lead to significant idling of compute units and available memory bandwidth. The current programming models and interface to the architecture provide *no simple abstraction* to manage the utilization of critical resources such as memory/compute bandwidth and on-chip memory. To *leverage* this undesirable wastage to perform useful work, we propose a new hardware-software abstraction—*the assist warp*—in the GPU programming model and architecture. Assist warps enable light-weight execution of helper-thread code alongside the primary application to perform optimizations to accelerate the program (such as data compression or prefetching) and perform background tasks, system-level tasks, etc. Assist warps automatically adapt to the availability of resources and unbalances in the primary application's execution to increase overall efficiency and throughput.

1.5 Related Work

In this section, we provide a brief overview of prior work that address similar challenges in enhancing programmability, portability, and resource efficiency and works that propose cross-layer interfaces. We then contrast the *general* approaches taken by these works with the approaches taken in this thesis. We discuss related work specific to each of the proposed works at the end of each chapter.

1.5.1 Expressive Programming Models and Runtime Systems

Numerous software-only approaches tackle the disconnect between an application, the OS, and the underlying resources via programming models and runtime systems that allow explicit expression of data locality and independence [35, 36, 44, 56, 60, 101, 122, 303, 326, 327, 331, 362] in the programming model. This explicit expression enables the programmer and/or runtime system to make effective memory placement decisions in a NUMA system or produce code that is optimized to effectively leverage the cache hierarchy. For example, the Legion programming system [35] provides software abstractions to describe properties of data such as locality and independence. The programmer and runtime system can then explicitly place data in the memory hierarchy to maximize parallelism and memory efficiency. These approaches in general have several shortcomings. First, they are entirely software-based and are hence limited to using the *existing* interfaces to the architectural resources. Second, programming model-based approaches *require* rewriting applications to suit the model. For example, programs need to adapted to the Legion programming model to expose parallelism and locality using the model's abstractions. This requires explicit programmer effort to ensure correctness is retained. Third, these systems are very specific to an application type (e.g., operations on tiles [331], arrays [101]). Only those programs that can be expressed with Legion's task-based programming model can leverage its benefits.

In contrast, in this thesis, we proposed abstractions (Chapter 2-5), that are, first, *cross-layer*, and are hence *not* limited to existing interfaces between hardware and software and enable hardware-software cooperative mechanisms. As we demonstrate, this enables significant performance, programmability, and portability benefits. Second, all the abstractions proposed in this thesis *retain existing programming models and execution paradigms* to minimize programmer and developer effort. Third, each approach taken in this thesis is *general* and is *not* limited to any programming language or application. Programming model/runtime system approaches are orthogonal to our proposed approaches, and can *leverage* the abstractions we propose to enable a wider range of optimizations.

1.5.2 Leveraging Hints, Annotations, and Software Management for Performance Optimization

A large body of prior work aims to leverage the benefits of static program information in the form of hints, annotations, or directives in performance optimization. For example, Cooperative Cache Scrubbing [288] is a hardware-software mechanism that communicates to the cache which data blocks will not be used any longer in the program (dead blocks), so they can be evicted from the cache to make space for more useful data. More generally, these include *(i)* hint-based approaches, such as software prefetch instructions [1] and cache bypass/insertion/eviction hints [41, 42, 50, 120, 141, 216, 253, 254, 272, 288, 289, 329, 347, 363]; *(ii)* hardware-software cooperative prefetch techniques [64, 69, 96, 105, 117, 142, 161, 302, 346, 350] that use compiler analysis or annotations to inform a hardware prefetch engine; and *(iii)* program annotations to place data in heterogeneous memories (e.g., [9, 206, 214]).

The approaches taken in this thesis, XMem (Chapter 2), Locality Descriptor (Chapter 3), Zorua (Chapter 4), and CABA (Chapter 5), differ from these works in several ways. First, many of these prior works seek to inform hardware components with *specific directives* that override dynamic policies by enforcing *static* policies. This loss in dynamism introduces challenges when the workload behavior changes, the underlying architecture changes or is unknown (portability), or in the presence of co-running applications [193, 233, 334]. Our approaches do *not direct* policy at any component but only provide higher-level program semantics. System/hardware components can use this information to *supplement* their dynamic management policies. Second, these prior works are *specific* to an optimization (e.g., prefetching, cache insertion/eviction). For example, Cooperative Cache Scrubbing is only applicable to dead block eviction from the cache. Our approaches, XMem (Chapter 2) and Locality Descriptor (Chapter 3), however, provide a *general* interface to communicate program semantics that can be leveraged by *many* system/architectural components.

1.5.3 Enhancing Programming Ease and Portability in GPUs

There is a large body of work that aims to improve programmability and portability of modern GPU applications using software tools, such as auto-tuners [82, 89, 169, 292, 308], optimizing compilers [62, 130, 158, 208, 365, 366], and high-level programming languages and runtimes [88, 124, 271, 330]. For example, Porple [62] is an optimizing compiler that automatically selects the memory allocation in registers and scratchpad memory for GPU programs. hiCUDA [124] is a directive-based programming language that automatically chooses the lower-level specifications required in GPU programs (i.e., registers, scratchpad, threads per block), based on the directives provided by the programmer. These tools tackle a multitude of optimization challenges, and have been demonstrated to be very effective in generating high-performance portable code.

However, there are several shortcomings in these works in contrast with our works, Locality Descriptor (Chapter 3) and Zorua (Chapter 4). First, these prior works often require profiling runs [62, 82, 292, 308, 365, 366] on the GPU to determine the best performing resource specifications. Porple requires runtime profiling to determine the program's access patterns before selecting how much register space or scratchpad memory should be allocated to the program. These runs have to be repeated for each new input set and GPU generation. Second, software-based approaches still require significant programmer effort to write code in a manner that can be exploited by software to optimize the resource utilization. hiCUDA requires rewriting any program with its directive-based language. Zorua (Chapter 4) is software transparent and the Locality Descriptor (Chapter 3) does not require rewriting the application, but only requires provide hint-based annotations that do not affect program correctness. Third, selecting the best-performing resource specifications statically using software tools is a challenging task in virtualized environments (e.g., clouds), where it is unclear which kernels may be run together on the same Streaming Multiprocessor (SM) or where it is not known, a priori, which GPU generation the application may execute on. Finally, software tools assume a fixed amount of available resources. This leads to *dynamic* underutilization due to static allocation of resources, which cannot be easily addressed by these tools. None of the above tools, including Porple and hiCUDA, can handle dynamic recompilation in the presence of co-running applications or address dynamic underutilization of hardware resources. Furthermore, these prior works are largely orthogonal can be used in conjunction with our proposed approaches to further improve performance.

1.5.4 Tagged Architectures

Prior work proposes to associate software-defined metadata with each memory location in the form of tagged/typed memory [86, 102, 353, 377]. These proposals are typically used for fine-grained memory access protection, debugging, etc., and usually incur non-trivial performance/storage overhead. For example, PUMP [86] associates every word in the memory system with a software-defined metadata tag. These tags are then used to enforce security policies in hardware, e.g., avoiding buffer overflows. In contrast, XMem (Chapter 2) aims to deliver *general* program semantics to many system/hardware components to aid in *performance optimization*. This necessitates a low overhead implementation that is also general enough to enable a wide range of cross-layer optimizations, while not sacrificing programmability. While a fundamental *component* of XMem is the metadata tracking system similar to tagged memories, to achieve the above goal in performance optimization requires several other key components: a hardware-software *translator* that enables a many cross-layer optimizations with a common set of information provided by the programmer and alleviates challenges in portability and programmability; and a full hardware-software system design that partitions work between the compiler, OS, and hardware to minimize system complexity. Furthermore, XMem is designed to enable a number of features and benefits that cannot be obtained from tagged/typed architectures: *(i)* a flexible and extensible abstraction to *dynamically* describe program behavior with `XMemLib`; and *(ii)* low-overhead interfaces to many hardware components to easily access the expressed semantics, including the prefetcher, caches, memory controller, etc. PARD [215] and Labeled RISC-V [372] are tagged architectures that enable labeling memory requests with tags to applications, VMs, etc. These tags are used to convey an application's QoS, security requirements, etc., to hardware. XMem is similar in that it provides an interface to hardware to convey information from software. However, unlike these works [215, 372], we design a new abstraction (the atom) to flexibly express program semantics that can be seamlessly integrated into programming languages, runtime systems, and modern ISAs. The atom lends itself to a low-overhead implementation to convey software semantics to hardware components *dynamically* and at flexible granularities. XMem can potentially *leverage* tagged architectures to communicate atom IDs to different hardware components. Hence, PARD and Labeled RISC-V are complementary to XMem.

Chapter 2

Expressive Memory

This chapter proposes a rich low-overhead interface to enable the operating system and hardware architecture to leverage key higher-level program information regarding how an application accesses its data to optimize memory system performance in CPUs. We first motivate the programmability, portability, and resource efficiency challenges that exist in modern CPUs when optimizing for memory system performance. We then describe our proposed cross-layer abstraction, Expressive Memory, and detail its design and effectiveness in addressing these challenges.

2.1 Overview

As discussed in Section 1, traditionally, the key interfaces between the software stack and the architecture (the ISA and virtual memory) have been primarily designed to convey program *functionality* to ensure the program is executed as required by software. An application is converted into ISA instructions and a series of accesses to virtual memory for execution in hardware. The application is, hence, stripped down to the basics of what is necessary to execute the program correctly, and the *higher-level semantics* of the program are lost. For example, even the simple higher-level notion of different *data structures* in a program is *not* available to the OS or hardware architecture, which deal only with virtual/physical pages and addresses. While the higher-level semantics of data structures may be irrelevant for correct execution, these semantics could prove very useful to the system for *performance optimization.*

There is, in other words, a *disconnect* or *semantic gap* between the levels of the computing stack when it comes to conveying higher-level program semantics from the application to the wide range of system-level and architectural components that aim to improve performance. While the implications of the disconnect are far-reaching, in this work, we narrow the focus to a critical component in determining the overall system efficiency, *the memory subsystem.* Modern systems employ a large variety of components to optimize memory performance (e.g., prefetchers, caches, memory controllers). The semantic gap has two important implications:

Implication 1. The OS and hardware memory subsystem components are forced to *predict* or *infer* program behavior when optimizing for performance. This is challenging because: *(i)* each component (e.g., L1 cache, memory controller) sees only a *localized* view of the data accesses made by the application and misses the bigger picture, *(ii) specialized* hardware may be required for each component optimizing for memory, and *(iii)* the optimizations are typically only *reactive* as the program behavior is *not* known a priori.

Implication 2. Software is forced to optimize code to the specifics of the underlying architecture (e.g., by tuning *tile size* to fit a specific cache size). Memory resource availability, however, can change or be unknown (e.g., in virtualized environments or in the presence of co-running applications). As a result, software optimizations are often unable to make accurate assumptions regarding memory resource availability, leading to significant challenges in *performance portability*.

The challenges of predicting program behavior and hence the benefits of knowledge from software in memory system optimization are well known [9, 42, 50, 64, 69, 92, 117, 120, 141, 206, 216, 233, 253, 254, 272, 288, 289, 329, 346, 347, 350, 363]. There have been numerous hardware-software cooperative techniques proposed in the form of fine-grain hints implemented as new ISA instructions (to aid cache replacement, prefetching, etc.) [42, 50, 120, 141, 216, 253, 254, 272, 288, 289, 329, 347, 363], program annotations or directives to convey program semantics and programmer intent [9, 92, 120, 206, 233], or hardware-software co-designs for specific optimizations [64, 69, 117, 346, 350]. These approaches, however, have two significant shortcomings. First, they are designed for a *specific memory optimization* and are limited in their implementation to address only challenges specific to that optimization. As a result, they require changes across the stack for a *single optimization* (e.g., cache replacement, prefetching, or data placement). Second, they are often very specific directives to instruct a particular component to behave in a certain manner (e.g., instructions to prefetch specific data or prioritize certain cache lines). These specific directives create portability and programmability concerns because these optimizations may *not* apply across different architectures and they require significant effort to understand the hardware architecture to ensure the directives are useful.

Our Goal. In this work, we ask the question: *can we design a unifying general abstraction and a cohesive set of interfaces between the levels of the system stack to communicate key program semantics from the application to all the system-level and architectural components?* In response, we present Expressive Memory (XMem), a rich cross-layer interface that provides a new *view* of the program data to the entire system. Designing XMem in a *low-overhead, extensible, and general* manner requires addressing several non-trivial challenges involving conflicting tradeoffs between generality and overhead, programmability and effectiveness (§2.2.2). In this work, we provide a first attempt at designing a new end-to-end system to achieve our goal while addressing these challenges.

Expressive Memory comprises two key components:

(1) The Atom. We introduce a new hardware-software abstraction, the *atom*, which is a region of virtual memory with a set of well-defined properties (§2.3.1). Each atom maps to data that is *semantically similar*, e.g., a data structure, a *tile* in an array, or any pool of data with similar properties. Programs explicitly specify atoms that are communicated to the OS and hardware. Atoms carry program information such as: *(i)* data properties (e.g., data type, sparsity, approximability), *(ii)* access properties (e.g., access pattern, read-write characteristics), and *(iii)* data locality (e.g., data reuse, working set). The atom can also track properties of data that *change* during program execution.

(2) System and Cross-layer Interfaces. Figure 1 presents an overview of this component: *(i)* The interface to the application enables software to explicitly express atoms via program annotation, static compiler analysis, or dynamic profiling ❶; *(ii)* The XMem system enables summarizing, conveying, and storing the expressed atoms ❷; *(iii)* The interface to the OS and architectural components (e.g., caches, prefetchers) provides key supplemental information to aid optimization ❸. This interface enables any system/architectural component to simply *query* the XMem system for the higher-level semantics attached to a memory address ❹.

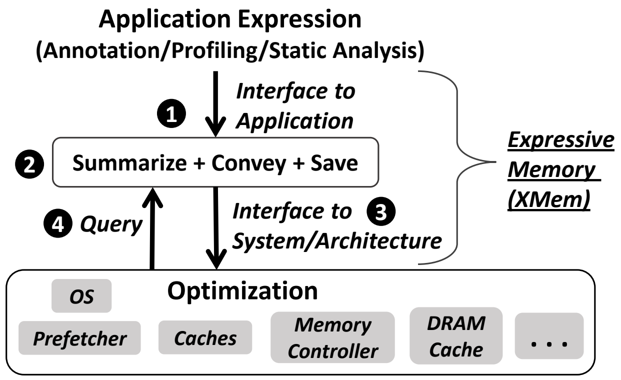


Figure 1: XMem: the system and interfaces.

Use Cases. XMem is designed to be a *general* interface to aid a wide range of memory optimizations. In this work, we first demonstrate the benefits of XMem in enhancing the portability of *software-based cache optimizations* (§3.4.4). The effectiveness of such optimizations (e.g., *cache tiling* [47, 72, 128, 220, 349, 359, 373]) is highly susceptible to changes in cache space availability: If the available cache space at runtime is *less* than what the program was optimized for, *cache thrashing* often ensues. We demonstrate that by leveraging data locality semantics (working set size and data reuse), we can enhance and coordinate the cache management and prefetching policies to avoid cache thrashing and ensure high hit rates are retained, thereby improving the portability of the optimization. We demonstrate that when software optimizations inaccurately assume available cache space, XMem reduces the loss in performance from 55% in the baseline system to 6% on average. Second, we demonstrate the performance benefits of *intelligent OS-based page placement in DRAM* by leveraging knowledge of data structures and their access semantics (§2.6). XMem improves performance by *(i)* isolating regular data structures with high row buffer locality in separate banks and *(ii)* spreading out irregular data structures across many banks/channels to maximize parallelism. Our experimental evaluation demonstrates an 8.5% average performance improvement (up to 31.9%) over state-of-the-art techniques.

Table 1: Summary of the example memory optimizations that XMem aids.

Memory optimization	**Example semantics provided by XMem**	**Example Benefits of XMem**
Cache management	*(i)* Distinguishing between data structures or pools of similar data; *(ii)* Working set size; *(iii)* Data reuse	Enables: *(i)* applying different caching policies to different data structures or pools of data; *(ii)* avoiding cache thrashing by *knowing* the active working set size; *(iii)* bypassing/prioritizing data that has no/high reuse.
Page placement in DRAM e.g., [234, 255]	*(i)* Distinguishing between data structures; *(ii)* Access pattern; *(iii)* Access intensity	Enables page placement at the *data structure* granularity to *(i)* isolate data structures that have high row buffer locality and *(ii)* spread out concurrently-accessed irregular data structures across banks and channels to improve parallelism.
Cache/memory compression e.g., [22, 93, 97, 260, 262–264, 337]	*(i)* Data type: integer, float, char; *(ii)* Data properties: sparse, pointer, data index	Enables using a *different compression algorithm* for each data structure based on data type and data properties, e.g., sparse data encodings, FP-specific compression, delta-based compression for pointers [264].
Data prefetching e.g., [31, 306, 307, 340]	*(i)* Access pattern: strided, irregular, irregular but repeated (e.g., graphs), access stride; *(ii)* Data type: index, pointer	Enables *(i) highly accurate* software-driven prefetching while leveraging the benefits of hardware prefetching (e.g., by being memory bandwidth-aware, avoiding cache thrashing); *(ii)* using different prefetcher *types* for different data structures: e.g., stride [31], tile-based [69], pattern-based [265, 306, 307, 340], data-based for indices/pointers [78, 96], etc.
DRAM cache management e.g., [147, 148, 150, 223, 224, 369, 371]	*(i)* Access intensity; *(ii)* Data reuse; *(iii)* Working set size	*(i)* Helps avoid cache thrashing by knowing working set size [371]; *(ii)* Better DRAM cache management via reuse behavior and access intensity information.
Approximation in memory e.g., [99, 226, 227, 283, 284, 324, 368]	*(i)* Distinguishing between pools of similar data; *(ii)* Data properties: tolerance towards approximation	Enables *(i)* each memory component to track how approximable data is (at a fine granularity) to inform approximation techniques; *(ii)* data placement in heterogeneous reliability memories [214].
Data placement: NUMA systems e.g., [8, 81]	*(i)* Data partitioning across threads (i.e., relating data to threads that access it); *(ii)* Read-Write properties	Reduces the need for profiling or data migration *(i)* to co-locate data with threads that access it and *(ii)* to identify Read-Only data, thereby enabling techniques such as replication.
Data placement: hybrid memories e.g., [92, 198, 343]	*(i)* Read-Write properties (Read-Only/Read-Write); *(ii)* Access intensity; *(iii)* Data structure size; *(iv)* Access pattern	Avoids the need for profiling/migration of data in hybrid memories to *(i)* effectively manage the asymmetric read-write properties in NVM (e.g., placing Read-Only data in the NVM) [92, 343]; *(ii)* make tradeoffs between data structure "hotness" and size to allocate fast/high bandwidth memory [9]; and *(iii)* leverage row-buffer locality in placement based on access pattern [369].
Managing NUCA systems e.g., [125, 233]	*(i)* Distinguishing pools of similar data; *(ii)* Access intensity; *(iii)* Read-Write or Private-Shared properties	*(i)* Enables using different cache policies for different data pools (similar to [233]); *(ii)* Reduces the need for reactive mechanisms that detect sharing and read-write characteristics to inform cache policies.

More generally, Table 1 presents nine example memory optimizations and the benefits XMem can provide over prior works that propose these optimizations in a specialized manner. XMem's benefits arise in three ways. First, it provides a *unifying*, central interface for a wide range of optimizations that use many of the *same* semantics. Second, it partitions data into pools of semantically-similar data. This enables using different policies (e.g., cache policies, compression algorithms) for different pools of data. Third, it enhances optimizations by providing higher-level semantics that *(i)* are unavailable locally to each component at runtime (e.g., distinguishing between data structures, data properties), *(ii)* are challenging to accurately infer (e.g., working set size, data reuse) or *(iii)* require profiling/monitoring to determine (e.g., read-only/read-write, private/shared data characteristics).

This work makes the following **contributions**:

- This work is the first attempt to design a *holistic and general* cross-layer interface to communicate higher-level program semantics to the different system and architectural components in order to enable more effective memory optimizations in modern CPUs.
- To this end, we introduce XMem, which comprises a new software-hardware abstraction—the Atom—and a full end-to-end system design. XMem *(i)* is general and flexible enough to cover a wide range of program semantics and use cases, *(ii)* is completely decoupled from system functionality and affects only performance not correctness, *(iii)* can react to phase changes in data properties during execution, and *(iv)* has a low-overhead implementation.
- We quantitatively demonstrate the benefits of using XMem to *(i)* improve the portability of software-based cache optimizations by leveraging *data locality* semantics and *(ii)* enhance OS-based DRAM placement by leveraging semantics of data structures and their access properties. We highlight seven other use cases (Table 1).

2.2 Goals and Challenges

2.2.1 Key Requirements

There are several key requirements and invariants that drive the design of the proposed system:

***(i)* Supplemental and hint-based.** The new interface should *not* affect functionality or correctness of the program in any way—it provides only *supplemental* information to help improve *performance*. This reduces the necessity of obtaining precise or detailed hints, and system implementation can be *simpler* as information can be conveyed/stored imprecisely.

***(ii)* Architecture agnosticism.** The abstraction for expressing semantics must be based on the *application* characteristics rather than the specifics of the system, e.g., cache size, memory banks available. This means that the programmer/software need *not* be aware of the precise workings of the memory system resources, and it significantly alleviates the portability challenges when the programmer/software optimizes for performance.

***(iii)* Generality and extensibility.** The interface should be general enough to flexibly express a wide range of program semantics that could be used to aid many system-level and architectural (memory) optimizations, and extensible to support more semantics and optimizations.

***(iv)* Low overhead.** The interface must be amenable to an implementation with low storage area and performance overheads, while preserving the semantics of existing interfaces.

2.2.2 Challenges

Current system and architectural components see only a description of the program's data in terms of *virtual/physical addresses.* To provide higher-level program-related semantics, we need to associate each address with much more information than is available to the entire system today, addressing the following three challenges:

Challenge 1: Granularity of expression. The granularity of associating program semantics with program data is challenging because the best granularity for *expressing* program semantics is program dependent. Semantics could be available at the granularity of an entire data structure, or at much smaller granularities, such as a *tile* in an array. We cannot simply map program semantics to *every* individual virtual address as that would incur too much overhead, and the fixed granularity of a *virtual page* may be too coarse-grained, inflexible and challenging for programmers to reason about.

Challenge 2: Generality vs. specialization. Our architecture-agnosticism requirement implies that we express higher-level information from the application's or the programmer's point of view—without any knowledge/assumptions of the memory resources or specific directives to a hardware component. As a consequence, much of the conveyed information may be either irrelevant, too costly to manage effectively, or *too complex* for different hardware components to easily use. For example, hardware components like prefetchers are operated by simple hardware structures and need only know prefetchable access patterns. Hence, the abstraction must be *(i) high-level and architecture-agnostic*, so it can be easily expressed by the programmer and *(ii) general*, in order to express a range of information useful to many components. At the same time, it should be still amenable to translation into simple directives for each component.

Challenge 3: Changing data semantics. As the program executes, the semantics of the data structures and the way they are accessed can change. Hence, we need to be able to express *dynamic* data attributes in *static* code, and these changing attributes need to be conveyed to the running system at the appropriate time. This ensures that the data attributes seen by the memory components are accurate any time during execution. Continual updates to data attributes at runtime can impose significant overhead that must be properly managed to make the approach practical.

2.3 Our Approach: Expressive Memory

We design Expressive Memory (XMem), a new rich cross-layer interface that enables *explicit* expression and availability of key program semantics. XMem comprises two key components: *(i)* a new hardware-software *abstraction* with a well-defined set of properties to convey program semantics and *(ii)* a rich set of *interfaces* to convey and store that information at different system/memory components.

2.3.1 The Atom Abstraction

We define a new hardware-software abstraction, called an *Atom*, that serves as the *basic unit* of expressing and conveying program semantics to the system and architecture. An atom forms both an *abstraction* for information expressed as well as a *handle* for communicating, storing, and retrieving the conveyed information across different levels of the stack. Application programs can dynamically create atoms in the program code, each of which describes a specific range of program data at any given time during execution. The OS and hardware architecture can then interpret atoms specified in the program when the program is executed. There are three key components to an atom: *(i) Attributes:* higher-level data semantics that it conveys; *(ii) Mapping:* the virtual address range that it describes; and *(iii) State:* whether the atom is currently active or inactive.

2.3.2 Semantics and Invariants of an Atom

We define the invariants of the atom abstraction and then describe the operators that realize the semantics of the atom.

- **Homogeneity:** All the data that maps to the same atom has the *same* set of attributes.
- **Many-to-One VA-Atom Mapping:** At any given time, any virtual address (VA) can map to *at most* one atom. Hence, the system/architecture can query for the atom (if any) associated with a VA and thereby obtain any attributes associated with the VA, at any given time.
- **Immutable Attributes:** While atoms are dynamically created, the attributes of an atom *cannot* be changed once created. To express different attributes for the same data, a new atom should be created. Atom attributes can, hence, be specified *statically* in the program code. Because any atom's attributes *cannot* be updated during execution, these attributes can be summarized and conveyed to the system/architecture at compile/load time *before execution*. This minimizes expensive communication at runtime (Challenge 3).
- **Flexible mapping to data:** Any atom can be flexibly and dynamically mapped/unmapped to any set of data of any size. By selectively mapping and/or unmapping data to the same atom, an atom can be mapped to non-contiguous data (Challenge 1).

• **Activation/Deactivation:** While the attributes of an atom are immutable and statically specified, an atom itself can be *dynamically* activated and deactivated in the program. The attributes of an atom are recognized by the system *only* when the atom is currently active. This enables updating the attributes of any data region as the program executes: when the atom no longer accurately describes the data, it is deactivated and a new atom is mapped to the data. This ensures that the system always sees the correct data attributes during runtime, even though the attributes themselves are communicated earlier at load time (Challenge 3).

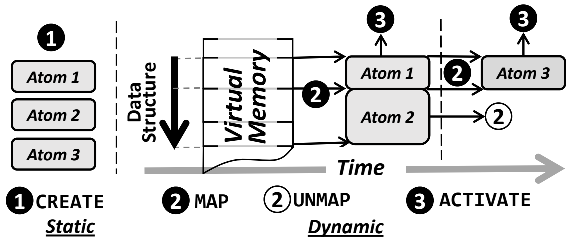


Figure 2: Overview of the three atom operators.

To manipulate the three components (*Attributes*, *Mapping*, and *State*), there are three corresponding operations that can be performed on an atom via a corresponding library call: *(i)* **`CREATE`** an atom, providing it with immutable statically-specified attributes; *(ii)* **`MAP/UNMAP`** an atom to/from a range of data; and *(iii)* **`ACTIVATE/DEACTIVATE`** an atom to dynamically tell the memory system that the attributes of the atom are now (in)valid for the data the atom is mapped to.

Figure 2 depicts an overview of the atom operators. Atoms are first created in the program with statically-specified attributes ❶. During memory allocation (e.g., malloc), data structures are allocated ranges of virtual memory. After allocation, atoms with the appropriate attributes can be flexibly *mapped* to the corresponding VA range that each atom describes ❷. Once the mapped atom is *activated*, all the system components recognize the attributes as valid ❸. Data can be easily remapped to a different atom that describes it better as the program moves into a different phase of execution (using the `MAP` operator ❷), or just unmapped ② when the atom no longer accurately describes the data. The `MAP/UNMAP` operator can be flexibly called to selectively map/unmap multiple data ranges to/from the same atom at any granularity. The `ACTIVATE/DEACTIVATE` operator dynamically validates/invalidates the atom attributes relating to *all* data the atom is mapped to.

2.3.3 Attributes of an Atom and Use Cases

Each atom contains an extensible set of attributes that convey the key program semantics to the rest of the system. Table 1 lists example use cases for these attributes. The three classes of attributes (to date) are:

(1) Data Value Properties: An expression of the attributes of the data values contained in the data pool mapped to an atom. It is implemented as an extensible list using a single bit for each attribute. These attributes include data type (e.g., `INT32`, `FLOAT32`, `CHAR8`) and data properties (e.g., `SPARSE`, `APPROXIMABLE`, `POINTER`, `INDEX`).

(2) Access Properties: This describes three key characteristics of the data the atom is mapped to:

• **`AccessPattern`**: This attribute defines the `PatternType`, currently either `REGULAR` (with a specific stride that is also specified), `IRREGULAR` (when the access pattern is *repeatable* within the data range, but with no repeated stride, e.g., graphs), or `NON_DET` (when there is no repeated pattern).

• **`RWChar`**: This attribute describes the read-write characteristics of data at any given time, currently either `READ_ONLY`, `READ_WRITE`, or `WRITE_ONLY`. It could also be extended to include varying degrees of read-write intensity, and include shared/private information.

• **`AccessIntensity`**: This attribute conveys the access frequency or "hotness" of the data relative to other data at any given time. This attribute can be provided by the programmer, compiler, or profiler. It is represented using an 8-bit integer, with `0` representing the lowest frequency. Higher values imply an increasing amount of intensity *relative* to other data. Hence, this attribute conveys an access intensity ranking between different data mapped to different atoms.

(3) Data Locality: This attribute serves to express software optimizations for cache locality explicitly (e.g., cache tiling, stream buffers, partitions, etc.). The key attributes include *working set size* (which is inferred from the size of data the atom is mapped to) and *reuse*, for which we use a simple 8-bit integer (`0` implying no reuse and higher values implying a higher amount of reuse *relative* to other data).

Note that the atom abstraction and its interface do *not* a priori limit the program attributes that an atom can express. This makes the interface flexible and forward-compatible in terms of extending and changing the expressed program semantics. The above attributes have been selected for their memory optimization benefits (Table 1) and ready translation into simple directives for the OS and hardware components.

2.3.4 The XMem System: Key Design Choices

Before we describe XMem's system implementation, we explain the rationale behind the key design choices.

• **Leveraging Hardware Support:** For the XMem design, we leverage hardware support for two major reasons. First, a key design goal for XMem is to minimize the runtime *overhead* of tracking and retrieving semantics at a fine granularity (even semantics that change as the program executes). We hence leverage support in *hardware* to efficiently perform several key functionalities of the XMem system—mapping/unmapping of semantics to atoms and activating/deactivating atoms at runtime. This avoids the high overhead of frequent system calls, memory updates, etc. Second, we aim to enable the many hardware-managed components in the memory hierarchy to leverage XMem. To this end, we use hardware support to efficiently transmit key semantics to the different hardware components.

- **Software Summarization and Hardware Tracking:** Because the potential atoms and their attributes are known statically (by examining the application program's `CREATE` calls), the compiler can summarize them at compile time, and the OS can load them into kernel memory at load time. The program directly communicates an atom's active status and address mapping(s) at runtime (via `MAP` and `ACTIVATE` calls) with the help of new instructions in the ISA (§2.4) and hardware support. This minimizes expensive software intervention/overhead at runtime. In other words, the static `CREATE` operator is handled in software before program execution and the dynamic `MAP` and `ACTIVATE` operators are handled by hardware at runtime.
- **Centralized Global Tracking and Management:** All the statically-defined atoms in the program are assigned a global *Atom ID* (within a process) that the *entire* system recognizes. Tracking which atoms are active at runtime and the inverse mapping between a VA and atom ID is also centralized at a *global* hardware table, to minimize storage and communication cost (i.e., all architectural components access the same table to identify the active atom for a VA).
- **Private Attributes and Attribute Translation:** The atom attributes provided by the application may be too complex and excessive for easy interpretation by components like the cache or prefetcher. To address this challenge (Challenge 2), when the program is loaded for execution or after a context switch, the OS invokes a *hardware translator* that converts the higher-level attributes to sets of specific primitives relevant to each hardware component and the optimization the component performs. Such specific primitives are then saved *privately* at each component, e.g., the prefetcher saves only the access pattern for each atom.

2.3.5 XMem: Interfaces and Mechanisms

Figure 3 depicts an overview of the system components to implement the semantics of XMem.

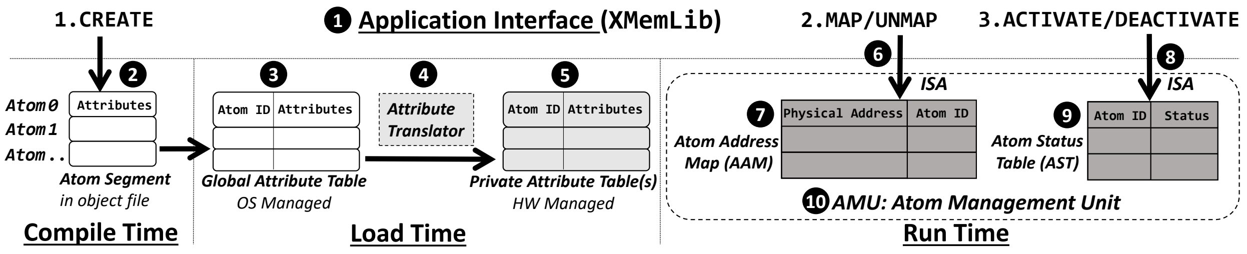


Figure 3: XMem: Overview of the components.

Programmer/Application Interface. The application interface to XMem is via a library,

Table 2: The XMem operators and corresponding XMemLib functions and ISA instructions (sizes/lengths in bytes).

XMem Op	XMemLib Functions (Application Interface)	XMem ISA Insts (Architecture Interface)
CREATE	AtomID CreateAtom(data_prop, access_pattern, reuse, rw_characteristics)	No ISA instruction required
MAP/UNMAP	AtomMap(atom_id, start_addr, size, map_or_unmap) Atom2DMap(atom_id, start_addr, lenX, sizeX, sizeY, map_or_unmap) Atom3DMap(atom_id, start_addr, lenX, lenY, sizeX, sizeY, sizeZ, map_or_unmap)	ATOM_MAP AtomID, Dimensionality ATOM_UNMAP AtomID, Dimensionality *Address ranges specified in AMU-specific registers*
ACTIVATE/ DEACTIVATE	AtomActivate(atom_id) AtomDeactivate(atom_id)	ATOM_ACTIVATE AtomID ATOM_DEACTIVATE AtomID

XMemLib (❶). An atom is defined by a class data structure that defines the attributes of the atom and the operator functions (CREATE, MAP/UNMAP, and ACTIVATE/DEACTIVATE). An atom and its static attributes can be instantiated in the program code (CREATE) by the programmer, autotuner, or compiler.

System/Architecture Interface. XMemLib communicates with the OS and architecture in the following two ways.

First, at compile time, the compiler summarizes all the atoms in the program statically and creates a table for atom attributes, indexed by atom ID. During run time, the same static atom can have many instances (e.g., within a for loop or across multiple function calls). All of the calls to create the same atom will, however, be mapped to the same static atom (and Atom ID). This is possible because atom attributes are immutable. However, the address mapping of each atom is typically not known at compile time because virtual address ranges are only resolved at runtime. The compiler creates a table of all the atoms in the program along with the atom attributes. This table is placed in the *atom segment* of the program object file (❷). When the program is loaded into memory for execution by the OS, the OS also reads the atom segment and saves the attributes for each atom in the GLOBAL ATTRIBUTE TABLE (GAT ❸), which is managed by the OS in kernel space. The OS also invokes a *hardware translator* (❹) that converts the higher-level attributes saved in the GAT to sets of specific hardware primitives relevant to each hardware component, and saves them in a per-component PRIVATE ATTRIBUTE TABLE (PAT ❺), managed in hardware by each component.

Second, at run time, XMem operators, in the form of function calls in XMemLib, are translated into *ISA instructions* that inform the system and architecture of the atoms' activation/deactivation and mapping. Conceptually, the MAP/UNMAP operator (❻) is converted into ISA instructions that update the ATOM ADDRESS MAP (AAM ❼), which enables looking up the atom ID associated with a physical address (PA). We use the PA to index the AAM instead of the VA to simplify the table design

(§2.4.2). The ACTIVATE/DEACTIVATE operator (❽) is converted into ISA instructions that update an atom's active status in the ATOM STATUS TABLE (AST ❾). The AST and AAM are managed by the Atom Management Unit (AMU ❿). Because tables with entries for each PA are infeasible, we use simple mechanisms to avoid them. These mechanisms, and the functioning and implementation of these hardware and software components, are described in §2.4.2.

Flexibility and Extensibility. The system/architecture interface ensures that the ISA and the microarchitecture need only implement the three operators, but does *not* dictate what application attributes can be conveyed. The attributes are stored in the binary as a separate metadata segment with a version identifier to identify the information format. The information format can be enhanced across architecture generations, ensuring flexibility and extensibility, while the version identifier ensures forward/backward compatibility. Any future architecture can interpret the semantics and older XMem architectures can simply ignore unknown formats.

2.4 XMem: Detailed Design

We now detail the design and implementation of the interfaces and components in XMem. We describe the application, OS, and architecture interfaces (§2.4.1), the key components of XMem (§2.4.2), the use of XMem in virtualized environments (§2.4.3), and the overheads of our design (§2.4.4).

2.4.1 The Interfaces

Application Interface. The primary interface between XMem and the application is XMemLib, a library that provides type definitions and function calls for atoms. XMemLib includes an atom class definition with the attributes described in §2.3.3. XMemLib provides three types of XMem operations on atoms in the form of function calls. These operations are the interface to manipulate the attributes, mappings and state of an atom. Table 2 summarizes the definition of all the functions (also discussed below):

(1) CREATE: The function CreateAtom creates an atom with the attributes specified by the input parameters, and returns an *Atom ID*. Multiple invocations of CreateAtom at the same place in the program code always return the same Atom ID (without reinvoking the function).

(2) MAP/UNMAP: These functions take an Atom ID and an address range as parameters, and invoke corresponding ISA instructions to tell the Atom Management Unit (AMU) to update the Atom Address Map (§2.4.2). We create multiple functions so that we can easily map or unmap multi-dimensional data structures (e.g., 2D/3D arrays). For example, Atom2DMap maps/unmaps a 2D block of data of width sizeX and height sizeY, in a 2D data structure that has a row length lenX.

(3) ACTIVATE/DEACTIVATE: The functions AtomActivate and AtomDeactivate serve to (de)activate the specified atom at any given time. They invoke corresponding ISA instructions that update the Atom Status Table (§2.4.2) at run time.

Operating System Interface. XMem interfaces with the OS in two ways. First, the OS manages the `Global Attribute Table (GAT)` (§2.4.2), which holds the attributes of all the atoms in each application. Second, the OS can optionally query for the *static* mapping between VA ranges and atoms through an interface to the memory allocator. This interface ensures that the OS knows the mapping *before* the virtual pages are mapped to physical pages, so that the OS can perform static optimizations, such as memory placement based on program semantics. Specifically, we augment the memory allocation APIs (e.g., `malloc`) to take Atom ID as a parameter. The memory allocator, in turn, passes the Atom ID to the OS via augmented system calls that request virtual pages. The memory allocator maintains the static mapping between atoms and virtual pages by returning virtual pages that match the requesting Atom ID. The compiler converts the pair `A=malloc(size); AtomMap(atomID,A,size);` into this augmented API: `A=malloc(size,atomID); AtomMap(atomID,A,size);`. This interface enables the OS to manipulate the virtual-to-physical address mapping *without* extra system call overheads.

Architecture Interface. We add two new ISA instructions to enable XMem to talk to the hardware at run time: *(i)* `ATOM_MAP/ATOM_UNMAP` tells the `Atom Management Unit (AMU)` to update the address ranges of an atom. When this instruction is executed, the parameters required to convey the address mapping for the different mapping types (Table 2) are implicitly saved in `AMU`-specific registers and accessed by the `AMU`. To map or unmap the address range to/from the specified atom, the `AMU` asks the Memory Management Unit (MMU) to translate the virtual address ranges specified by `ATOM_MAP` to physical address ranges, and updates the `Atom Address Map (AAM)` (§2.4.2). *(ii)* `ATOM_ACTIVATE/ATOM_DEACTIVATE` causes the `AMU` to update the `Atom Status Table (AST)` to activate/deactivate the specified atom.

2.4.2 System Design: Key Components

The system/architecture retrieves the data semantics associated with each memory address in three steps: *(i)* determine to which atom (if any) a given address maps; *(ii)* determine whether the atom is *active*; and *(iii)* retrieve the atom attributes. XMem enables this with four key components:

(1) Atom Address Map (`AAM`): This component determines the latest atom (if any) associated with any PA. Because the storage overhead of maintaining a mapping table between *each address* and Atom ID would be prohibitively large, we employ an *approximate mapping* between atoms and address ranges at a configurable granularity. The system decides the smallest *address range unit* the `AAM` stores for each address-range-to-atom mapping. The default granularity is 8 cache lines (512B), which means each consecutive 512B can map only to one atom. This design significantly reduces the storage overhead as we need only store one Atom ID for each 512B (0.2% storage overhead assuming an 8-bit Atom ID). We can reduce this overhead further by increasing the granularity or limiting the number of atoms in each application. For instance, if we support only 6-bit Atom IDs with a 1KB address range unit, the storage overhead becomes 0.07%. Note that because XMem provides only *hints* to the system, our approximate mapping may cause optimization inaccuracy but it has no impact on functionality and correctness.

To make it easy to look up the Atom ID for each address, the AAM stores the Atom IDs *consecutively* for *all* the physical pages. The index of the table is the physical page index and each entry stores all Atoms IDs in each page. In the default configuration, each of the Atom IDs require 8B of storage per page (8 bits times 8 subpages). With this design, the OS or the hardware architecture can simply use the physical address that is queried as the table index to find the Atom ID.

We use the PA instead of the VA to index this table because *(i)* there are far fewer PAs compared to VAs and *(ii)* this enables the simplified lookup scheme discussed above.

(2) Atom Status Table (AST): We use a bitmap to store the status (active or inactive) of all atoms in each application. Because `CreateAtom` assigns atom IDs consecutively starting at 0, this table is efficiently accessed using the atom ID as index. Assuming up to 256 atoms per application (all benchmarks in our experiments had under 10 atoms, all in performance-critical sections), the AST is only 32B per application. The `Atom Management Unit (AMU)` updates the bitmap when an `ATOM_(DE)ACTIVATE` instruction is executed.

(3) Attribute Tables (GAT and PAT) and the Attribute Translator: As discussed in §2.3.4, we store the attributes of atoms in a `Global Attribute Table (GAT)` and multiple `Private Attribute Tables (PAT)`. `GAT` is managed by the OS in kernel space. Each hardware component that benefits from XMem maintains its own `PAT`, which stores a *translated* version of the attributes (an example of this is in §3.4.4). This translation is done by the *Attribute Translator*, a hardware runtime system that translates attributes for each component at program load time and during a context switch.

(4) Atom Management Unit (AMU): This is a hardware unit that is responsible for *(i)* managing the AAM and AST and *(ii)* looking up the Atom ID given a physical address. When the CPU executes an XMem ISA instruction, the CPU sends the associated command to the AMU to update the AAM (for ATOM_MAP or ATOM_UNMAP) or the AST (for ATOM_ACTIVATE). For higher-dimensional data mappings, the AMU converts the mapping to a linear mapping at the AAM granularity and broadcasts this mapping to all the hardware components that require accurate information of higher-dimensional address mappings (see §3.4.4 for an example).

A hardware component determines the Atom ID of a specific physical address (PA) by sending an ATOM_LOOKUP request to the AMU, which uses the PA as the index into the AAM. To avoid memory accesses for all the ATOM_LOOKUP requests, each AMU has an atom lookaside buffer (ALB), which caches the results of recent ATOM_LOOKUP requests. The functionality of an ALB is similar to a

TLB in an MMU, so the AMU accesses the AAM *only* on ALB misses. The tags for the ALB are the physical page indexes, while the data are the Atom IDs in the physical pages. In our evaluation, we find that a 256-entry ALB can cover 98.9%[1] of the ATOM_LOOKUP requests.

2.4.3 XMem in Virtualized Environments

Virtualized environments employ virtual machines (VMs) or containers that execute applications over layers of operating systems and hypervisors. The existence of multiple address spaces that are seen by the guest and host operating systems, along with more levels of abstraction between the application and the underlying hardware resources, makes the design of hardware-software mechanisms challenging. XMem is, however, designed to seamlessly function in these virtualized environments, as we describe next.

XMem Components. The primary components of XMem include the AAM, AST, the PATs, and the GAT. Each of these components function with no changes in virtualized environments: *(i)* AAM: The hardware-managed AAM, which maps physical addresses to atom IDs, is indexed by the *host* physical address. As a result, this table is *globally shared* across all processes running on the system irrespective of the presence of multiple levels of virtualization. *(ii)* AST and PATs: All atoms are tracked at the *process level* (irrespective of whether the processes belong to the same or different VMs). The per-process hardware-managed tables (AST and PATs) are reloaded during a context switch to contain the state and attributes of the atoms that belong to the currently-executing process. Hence, the functioning of these tables remains the same in the presence of VMs or containers. *(iii)* GAT: The GAT is software-managed and is maintained by each *guest* OS. During context switches, a register is loaded with a host physical address that points to the new process' GAT and AST.

XMem Interfaces. The three major interfaces (CREATE, MAP/UNMAP, and ACTIVATE/DEACTIVATE) require no changes for operation in virtualized environments. The CREATE operator is handled in software at compile time by the guest OS and all created atoms are loaded into the GAT by the guest OS at program load time. The MAP/UNMAP operator communicates directly with the MMU to map the host physical address to the corresponding atom ID using the XMem ISA instructions. The ACTIVATE/DEACTIVATE operator simply updates the AST, which contains the executing process' state.

Optimizations. OS-based software optimizations (e.g., DRAM placement in §2.6) require that the OS have visibility into the available physical resources. The physical resources may however be abstracted away from the guest OS in the presence of virtualization. In this case, the resource allocation and its optimization needs to be handled by the hypervisor or host OS for all the VMs

[1]Does not include the Gramschmidt [267] workload, which requires a more sophisticated caching policy than LRU to handle large strides.

that it is hosting. To enable the hypervisor/host OS to make resource allocation decisions, the guest OS also communicates the attributes of the application's atoms to the hypervisor. For hardware optimizations (e.g., caching policies, data compression), the hardware components (e.g., caches, prefetchers) retrieve the atom attributes for each process using the AAM and PATs. This is the same mechanism irrespective of the presence of virtualization. These components use application/VM IDs to distinguish between addresses from different applications/VMs (similar to prior work [215] or modern commercial virtualization schemes [19, 137]).

2.4.4 Overhead Analysis

The overheads of XMem fall into four categories: memory storage overhead, instruction overhead, hardware area overhead, and context switch overhead, all of which are small:

(1) Memory storage overhead. The storage overhead comes from the tables that maintain the attributes, status, and mappings of atoms (AAM, AST, GAT, and PAT). As §2.4.2 discusses, the AST is very small (32B). The GAT and PAT are also small as the attributes of each atom need 19B, so each GAT needs only 2.8KB assuming 256 atoms per application. AAM is the largest table in XMem, but it is still insignificant as it takes only 0.2% of the physical memory (e.g., 16MB on a 8GB system), and it can be made even smaller by increasing the granularity of the address range unit (§2.4.2).

(2) Instruction overhead. There are instruction overheads when applications invoke the XMemLib functions to create, map/unmap, activate/deactivate atoms, which execute XMem instructions. We find this overhead negligible because: *(i)* XMem does *not* use extra system calls to communicate with the OS, so these operations are very lightweight; *(ii)* the program semantics or data mapping do *not* change very frequently. Among the workloads we evaluate, an additional 0.014% instructions on average (at most, 0.2%) are executed.

(3) Hardware area overhead. XMem introduces two major hardware components, Attribute Translator and AMU. We evaluate the storage overhead of these two components (including the AMU-specific registers) using CACTI 6.5 [322] at 14 *nm* process technology, and find that their area is 0.144 mm^2, or 0.03% of a modern Xeon E5-2698 CPU.

(4) Context switch overhead. XMem introduces one extra register for context switches—it stores the pointer to AST and GAT (stored consecutively for each application) in the AMU. AAM does not need a context-based register because it is a global table. The OS does not save the AMU-specific registers for MAP/UNMAP (Table 2) as the information is saved in the AAM. One more register adds very small overhead (two instructions, $\leq$ 1 ns) to the OS context switch (typically 3-5 μs). Context switches also require flushing the ALBs and PATs. Because these structures are small, the overhead is also commensurately small (~700 ns).

2.5 Use Case 1: Cache Management

Cache management is a well-known complex optimization problem with substantial prior work in both software (e.g., code/compiler optimizations, auto tuners, reuse hints) and hardware (advanced replacement/insertion/partitioning policies). XMem seeks to *supplement* both software and hardware approaches by providing key program semantics that are challenging to infer at run-time. As a concrete end-to-end example, we describe and evaluate how XMem enhances *dynamic* policies to improve the portability and effectiveness of *static software-based* cache optimizations under *varying* cache space availability (as a result of co-running applications or unknown cache size in virtualized environments).

Many software techniques *statically* tune code by sizing the active working set in the application to maximize cache locality and reuse—e.g., hash-join partitioning [46] in databases, cache tiling [47, 72, 128, 220, 349, 359, 373] in linear algebra and stencils, cache-conscious data layout [320] for similarity search [37]. XMem improves the portability and effectiveness of static optimizations when resource availability is unknown by conveying the *optimization intent* to hardware—i.e., XMem conveys which high-reuse working set (e.g., tile) should be kept in the cache. It does *not* dictate exactly what caching policy to use to do this. The *hardware cache* leverages the conveyed information to keep the high-reuse working set of each application in the cache by prioritizing such data over other low-reuse data. In cases where the active working set does *not* fit in the available cache space, the cache *mitigates thrashing* by *pinning* part of the working set and then *prefetches* the rest based on the expressed access pattern.

2.5.1 Evaluation Methodology

We model and evaluate XMem using zsim [285] with a DRAMSim2 [275] DRAM model. We use the Polybench suite [267], a collection of linear algebra, stencil, and data mining kernels. We use PLUTO [47], a polyhedral locality optimizer that uses *cache tiling* to statically optimize the kernels. We evaluate kernels that can be tiled within three dimensions and a wide range of tile sizes (from 64B to 8MB), ensuring the total work is always the same.

2.5.2 Evaluation Results

Overall performance. To understand the cache tiling challenge, in Figure 4, we plot the execution time of 12 kernels, which are statically compiled with different tile sizes. For each workload, we show the results of two systems: *(i)* `Baseline`, the baseline system with a high-performance cache replacement policy (DRRIP [143]) and a multi-stride prefetcher [31] at L3; and *(ii)* `XMem`, the system with the aforementioned cache management and prefetching mechanisms.

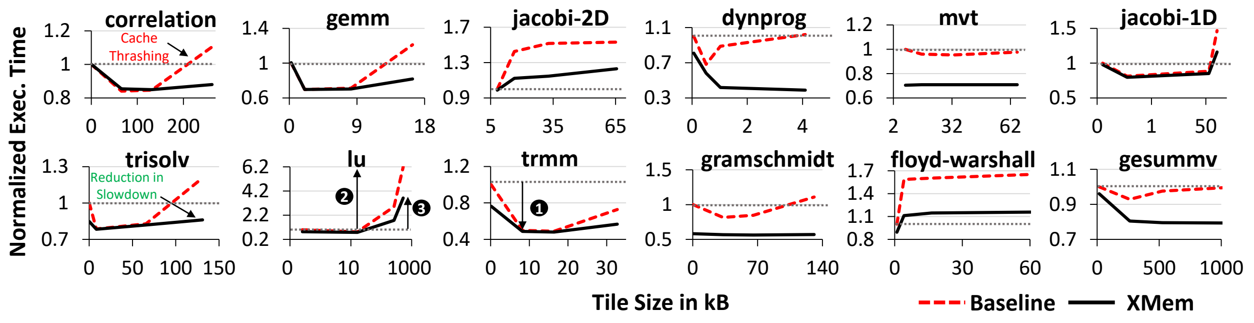


Figure 4: Execution time across different tile sizes (normalized to Baseline with the smallest tile size).

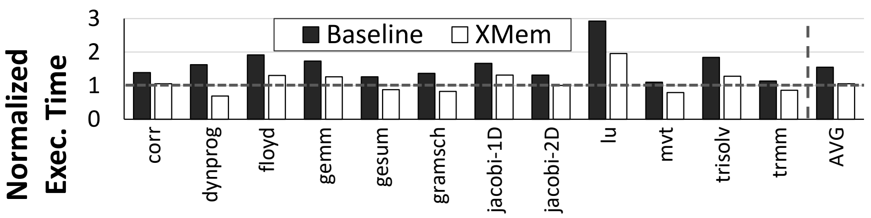


Figure 5: Maximum execution time with different cache sizes when code is optimized for a 2MB cache.

For the `Baseline` system, execution time varies significantly with tile size, making tile size selection a challenging task. Small tiles significantly reduce the reuse in the application and can be on average 28.7% (up to 2× ❶) *slower* than the *best* tile size. Many optimizations, hence, typically size the tile to be as big as what can fit in the available cache space [33, 72]. However, when an optimization makes incorrect assumptions regarding available cache space (e.g., in virtualized environments or due to co-running applications), the tile size may *exceed* the available cache space. We find that this can lead to *cache thrashing* and severe slowdown (64.8% on average, up to 7.6× ❷), compared to the performance with an optimized tile size. `XMem`, however, significantly reduces this slowdown from cache thrashing in the largest tile sizes to 26.9% on average (up to 4.6× ❸). `XMem`'s large improvement comes from accurate *pinning* (that retains part of the high-reuse working set in the cache) and more accurate prefetching (that fetches the remaining working set).

Performance portability. To evaluate portability benefits from the reduced impact of cache thrashing in large tile sizes, we run the following experiment. For each workload, we pick a tile size optimized for a 2MB cache, and evaluate *the same program binary* on a 2MB cache and 2 smaller caches (1MB and 512KB). Figure 5 depicts the *maximum* execution time among these three cache sizes for both `Baseline` and `XMem`, normalized to `Baseline` with a 2MB cache. When executing with less cache space, we find that `XMem` increases the execution time by only 6%, compared to the `Baseline`'s 55%. Hence, we conclude that by leveraging the program semantics, `XMem` greatly enhances the performance portability of applications by reducing the impact of having less cache space than what the program is optimized for.

2.6 Use Case 2: Data Placement in DRAM

Off-chip main memory (DRAM) latency is a major performance bottleneck in modern CPUs [84, 160,235,237,351]. The performance impact of this latency is in large part determined by two factors: *(i) Row Buffer Locality (RBL)* [176, 236]: how often requests access the *same* DRAM row in a bank *consecutively*. Consecutive accesses to the same open row saves the long latency required to *close* the already-open row and *open* the requested row. *(ii) Memory Level Parallelism (MLP)* [114, 237]: the number of concurrent accesses to different memory banks or channels. Serving requests in parallel *overlaps* the latency of the different requests. A key factor that determines the DRAM access pattern—and thus RBL and MLP—is how the program's data is mapped to the DRAM channels, banks, and rows. This data placement is controlled by *(i)* the OS (via the virtual-to-physical address mapping) and *(ii)* the memory controller (via the mapping of physical addresses to DRAM channels/banks/rows).

To improve RBL and MLP, prior works use both the OS (e.g., [43, 87, 146, 202–204, 207, 225, 234, 255, 315, 356, 374]) and the memory controller (e.g., [55, 111, 129, 162, 332, 381, 382]) to introduce *randomness* in how data is mapped to the DRAM channels/banks/rows or *partition* banks/channels between different threads or applications. While effective, these techniques are *unaware* of the different semantics of data structures in an application, and hence suffer from two shortcomings. First, to determine the properties of data, an application needs to be *profiled* before execution or pages need to be migrated *reactively* based on runtime behavior. Second, these techniques apply the *same* mapping for *all* data structures within the application, even if RBL and MLP vary significantly across different data structures.

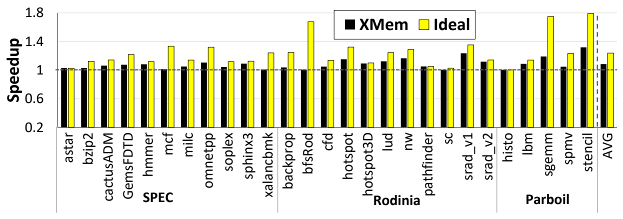
XMem enables distinguishing between data structures and provides key access semantics to the OS. Together with the knowledge of the underlying banks, channels, ranks, etc. and other co-running applications, the OS can create an intelligent mapping at the *data structure* granularity. Based on the data structure access patterns, the OS can *(i)* improve RBL by *isolating* data structures with high RBL from data structures that could cause interference if placed in the same bank and *(ii)* improve MLP by *spreading* out accesses to concurrently-accessed data structures across multiple banks and channels.

2.6.1 Evaluation Methodology

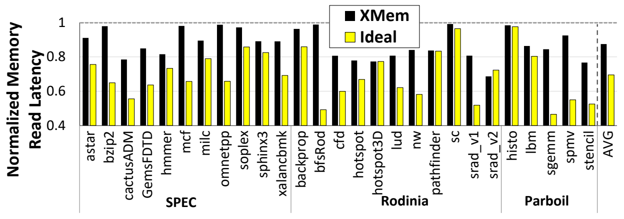
We use zsim [285] and DRAMSim2 [275] for evaluation. We strengthen our baseline system in three ways: *(i)* We use the *best*-performing physical DRAM mapping, among all the seven mapping schemes in DRAMSim2 and the two proposed in [162, 381], as our baseline; *(ii)* We *randomize* virtual-to-physical address mapping, which is shown to perform better than the Buddy algorithm [255]; *(iii)* For each workload, we enable the L3 prefetcher *only if* it improves performance. We evaluate a wide range of workloads from SPEC CPU2006 [309], Rodinia [61], and Parboil [313] and show results for 27 memory intensive workloads (with L3 MPKI > 1).

2.6.2 Evaluation Results

We evaluate three systems: *(i)* `Baseline`, the strengthened baseline system; *(ii)* `XMem`, DRAM placement using XMem; *(iii)* an *ideal* system that has *perfect* RBL, which represents the best performance possible by improving RBL. Figure 6a shows the speedup of the last two systems over `Baseline`. Figure 6b shows the corresponding memory read latency, normalized to `Baseline`. We make two observations.



(a) Speedup w/ XMem-based DRAM placement.



(b) Normalized read latency with XMem-based DRAM placement.

Figure 6: Leveraging XMem for DRAM placement.

First, XMem-based DRAM placement improves performance across a range of workloads: by 8.5% on average over `Baseline`, up to 31.9%. It is a significant improvement as the absolute upper-bound for *any* DRAM row-buffer optimization is 24.4% (`Ideal`). Among the 27 workloads, only 5 workloads do not see much improvement—they either *(i)* have less than 3% headroom to begin with (`sc` and `histo`) or *(ii)* are dominated by random accesses (`mcf`, `xalancbmk`, and `bfsRod`). Second, the performance improvement of XMem-based DRAM placement comes from the significant reduction in average memory latency, especially read latency, which is usually on the critical path. On average, `XMem` reduces read latency by 12.6%, up to 31.4%. Write latency is reduced by 6.2% (not shown).

We conclude that leveraging both the program semantics provided by XMem and knowledge of the underlying DRAM organization enables the OS to create intelligent DRAM mappings at a fine (data structure) granularity, thereby reducing memory latency and improving performance.

2.7 Related Work

To our knowledge, this is the first work to design a holistic and general cross-layer interface to enable the entire system and architecture to be aware of key higher-level program semantics that can be leveraged in memory optimization. We now briefly discuss closely related prior work specific to XMem. A more general comparison of approaches taken in XMem is discussed in Section 1.5.

Expressive programming models and runtime systems. Numerous software-only approaches tackle the disconnect between an application, the OS, and the underlying memory resources via programming models and runtime systems that allow explicit expression of data locality and independence [35, 36, 44, 56, 60, 101, 122, 303, 326, 327, 331, 362] in the programming model. This explicit expression enables the programmer and/or runtime system to make effective memory placement decisions in a NUMA system or produce code that is optimized to effectively leverage the cache hierarchy. These approaches have several shortcomings. First, they are entirely software-based and are hence limited to using the *existing* interfaces to the architectural resources. Second, unlike XMem, which is general and only hint-based, programming model-based approaches *require* rewriting applications to suit the model, while ensuring that program correctness is retained. Third, these systems are specific to an application type (e.g., operations on tiles, arrays). XMem is a general interface that is *not* limited to any programming language, application, or architecture. These approaches are orthogonal to XMem, and XMem can be built into them to enable a wider range of memory.

The Locality Descriptor [333] is a cross-layer abstraction to express data locality in GPUs. This abstraction is similar in spirit to XMem in bridging the semantic gap between hardware and software. However, the Locality Descriptor is primarily designed to convey *locality semantics* to leverage cache and NUMA locality in GPUs. XMem aims to convey *general* program semantics to aid memory optimization. This goal imposes different design challenges, requires describing a different set of semantics, and requires optimizing a different set of architectural techniques, leading to a very different cross-layer design for the abstraction.

Leveraging hints, annotations, and software management for memory optimization. A large body of prior work aims to leverage the benefits of static program information in the form of hints, annotations, or directives in memory optimization. These include *(i)* hint-based approaches, such as software prefetch instructions [1] and cache bypass/insertion/eviction hints [41, 42, 50, 120, 141, 216, 253, 254, 272, 288, 289, 329, 347, 363]; *(ii)* hardware-software cooperative prefetch techniques [64, 69, 96, 105, 117, 142, 161, 302, 346, 350] that use compiler analysis or annotations to inform a hardware prefetch engine; and *(iii)* program annotations to place data in heterogeneous memories (e.g., [9, 206, 214]). XMem differs from these works in several ways. First, many of these approaches seek to inform hardware components with *specific directives* that override dynamic policies by enforcing *static* policies. This loss in dynamism introduces challenges when the workload behavior changes, the underlying architecture changes or is unknown (portability), or in the presence of co-running applications [193, 233, 334]. XMem does *not direct* policy at any component but only provides higher-level program semantics. The memory components can use this information to *supplement* their dynamic management policies. Second, the approaches are *specific* to an optimization (e.g., prefetching, cache insertion/eviction). XMem provides a *general* holistic interface to communicate program semantics that can be leveraged by a wide range of system/architectural components for optimization.

The closest work to ours is Whirlpool [233], which provides a memory allocator to *statically* classify data into pools. Each pool is then managed differently at runtime to place data efficiently in NUCA caches. Whirlpool is similar to XMem in the ability to classify data into similar types and in retaining the benefits of dynamic management. However, XMem is *(i)* more versatile, as it enables *dynamically* classifying/reclassifying data and expressing more powerful program semantics than just static data classification and *(ii)* a general and holistic interface that can be used for a wide range of use cases, including Whirlpool itself. Several prior works [38, 39, 68, 75, 77, 100, 328] use runtime systems or the OS to aid in management of the cache. These approaches are largely orthogonal to XMem and can be used in conjunction with XMem to provide more benefit.

Tagged Architectures. Prior work proposes to associate software-defined metadata with each memory location in the form of tagged/typed memory [86, 102, 353, 377]. These proposals are typically used for fine-grained memory access protection, debugging, etc., and usually incur *non-trivial* performance/storage overhead. In contrast, XMem aims to deliver *general* program semantics to many system/hardware components to aid in performance optimization with *low overhead.* To this end, XMem is designed to enable a number of features and benefits that cannot be obtained from tagged/typed architectures: *(i)* a flexible and extensible abstraction to dynamically describe program behavior with `XMemLib`; and *(ii)* low-overhead interfaces to many hardware components to easily access the expressed semantics. PARD [215] and Labeled RISC-V [372] are tagged architectures that enable labeling memory requests with tags to applications, VMs, etc. These tags are used to convey an application's QoS, security requirements, etc., to hardware. XMem is similar in that it provides an interface to hardware to convey information from software. However, unlike these works [215, 372], we design a new abstraction (the atom) to flexibly express program semantics that can be seamlessly integrated into programming languages, runtime systems, and modern ISAs. The atom lends itself to a low-overhead implementation to convey software semantics to hardware components *dynamically* and at flexible granularities. XMem can potentially *leverage* tagged architectures to communicate atom IDs to different hardware components. Hence, PARD and Labeled RISC-V are complementary to XMem.

2.8 Summary

This work makes the case for richer cross-layer interfaces to bridge the semantic gap between the application and the underlying system and architecture. To this end, we introduce Expressive Memory (XMem), a holistic cross-layer interface that communicates higher-level program semantics from the application to different system-level and architectural components (such as caches, prefetchers, and memory controllers) to aid in memory optimization. XMem improves the performance and portability of a wide range of software and hardware memory optimization techniques by enabling them to leverage key semantic information that is otherwise unavailable. We evaluate and demonstrate XMem's benefits for two use cases: *(i)* static software cache optimization, by leveraging data locality semantics, and *(ii)* OS-based page placement in DRAM, by leveraging the ability to distinguish between data structures and their access patterns. We conclude that XMem provides a versatile, rich, and low overhead interface to bridge the semantic gap in order to enhance memory system optimization. We hope XMem encourages future work to explore re-architecting the traditional interfaces to enable many other benefits that are not possible today.

Chapter 3

The Locality Descriptor

This chapter proposes a rich cross-layer abstraction to address a key challenge in modern GPUs: leveraging data locality. We demonstrate how existing abstractions in the GPU programming model are insufficient to expose the significant data locality exhibited by GPU programs. We then propose a flexible abstraction that enables the application to express data locality concisely and, thus, enables the driver and hardware to leverage any data locality to improve performance.

3.1 Overview

Graphics Processing Units (GPUs) have evolved into powerful programmable machines that deliver high performance and energy efficiency to many important classes of applications today. Efficient use of memory system resources is critical to fully harnessing the massive computational power offered by a GPU. A key contributor to this efficiency is *data locality*—both *(i) reuse* of data within the application in the cache hierarchy (*reuse-based locality*) and *(ii)* placement of data *close* to the computation that uses it in a non-uniform memory access (NUMA) system (*NUMA locality*) [25, 131, 171, 228].

Contemporary GPU programming models (e.g., CUDA [246], OpenCL [20]) are designed to harness the massive computational power of a GPU by enabling explicit expression of *parallelism* and control of *software-managed memories* (scratchpad memory and register file). However, there is no clear explicit way to express and exploit *data locality*—i.e., *data reuse*, to better utilize the hardware-managed cache hierarchy, or *NUMA locality*, to efficiently use a NUMA memory system.

Challenges with Existing Interfaces. Since there is no explicit interface in the programming model to express and exploit data locality, expert programmers use various techniques such as software scheduling [194] and prefetch/bypass hints [243, 357] to carefully manage locality to obtain high performance. However, all such software approaches are significantly limited for three reasons. First, exploiting data locality is a challenging task, requiring a range of hardware mechanisms such as thread scheduling [63, 115, 154, 163, 183, 189, 194, 238, 345, 354], cache bypassing/prioritization [26, 191, 195–197, 199, 229, 325, 357, 358, 383], and prefetching [144, 155, 184, 188, 205, 298], to which software has *no easy access*. Second, GPU programs exhibit many different types of data locality, e.g., inter-CTA (reuse of data across Cooperative Thread Arrays or thread blocks), inter-warp and intra-warp locality. Often, *multiple* different techniques are required to exploit each type of locality, as a single technique in isolation is insufficient [26, 155, 181, 194, 250]. Hence, software-only approaches quickly become *tedious* and difficult programming tasks. Third, any software optimization employing fine-grained ISA instructions to manage caches or manipulating thread indexing to alter CTA scheduling is *not portable* to a different architecture with a different CTA scheduler, different cache sizes, etc [334].

At the same time, software-transparent architectural techniques miss critical program semantics regarding locality inherent in the algorithm. For example, CTA scheduling is used to improve data locality by scheduling CTAs that share data at the same core. This requires knowledge of *which* CTAs share data—knowledge that *cannot* easily be inferred by the architecture [63, 194]. Similarly, NUMA locality is created by placing data close to the threads that use it. This requires a priori knowledge of *which* threads access *what* data to avoid expensive reactive page migration [53, 54]. Furthermore, many architectural techniques, such as prefetching or cache bypassing/prioritization, would benefit from knowledge of the application's access semantics.

A Case Study. As a motivating example, we examine a common locality pattern of CTAs sharing data (*inter-CTA locality*), seen in the `histo` benchmark (Parboil [313]). `histo` has a predominantly accessed data structure (`sm_mappings`). Figure 7 depicts how this data structure ① is accessed by the CTA grid ②. All threads in GPU programs are partitioned into a multidimensional grid of CTAs. CTAs with the same color access the same data range (also colored the same) ③. As depicted, there is plentiful reuse of data between CTAs ④ and the workload has a very deterministic access pattern.

Today, however, exploiting reuse-based locality or NUMA locality for this workload, *at any level of the compute stack*, is a challenging task. The hardware architecture, on the one hand, misses key program information: knowledge of *which CTAs* access the same data ④, so they can be scheduled at the same SM (Streaming Multiprocessor); and knowledge of *which data* is accessed by those CTAs ⑤, so that data can be placed at the same NUMA zone. The programmer/compiler, on the other hand, has *no easy access* to hardware techniques such as CTA scheduling or data placement. Furthermore, optimizing for locality is a tedious task as a *single technique* alone is insufficient to exploit locality (§3.2). For example, to exploit NUMA locality, we need to coordinate *data placement* with CTA scheduling to place data close to the CTAs that access it. *Hence, neither the programmer, the compiler, nor hardware techniques can easily exploit the plentiful data locality in this workload.*

Our Approach. To address these challenges, we introduce the Locality Descriptor: a *cross-layer* abstraction to *express* and *exploit* different forms of data locality that all levels of the compute stack—from application to architecture—recognize. The Locality Descriptor *(i)* introduces a flexible and portable interface that enables the programmer/software to *explicitly* express and optimize for data locality and *(ii)* enables the hardware to *transparently* coordinate a range of architectural techniques (such as CTA scheduling, cache management, and data placement), guided by the

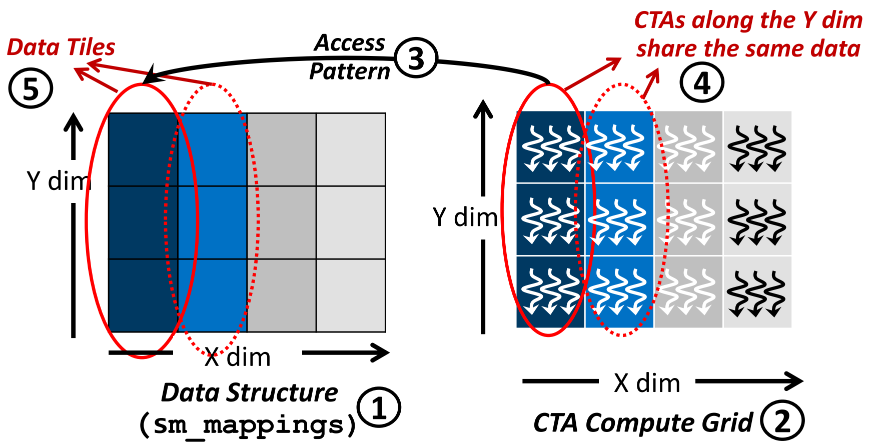
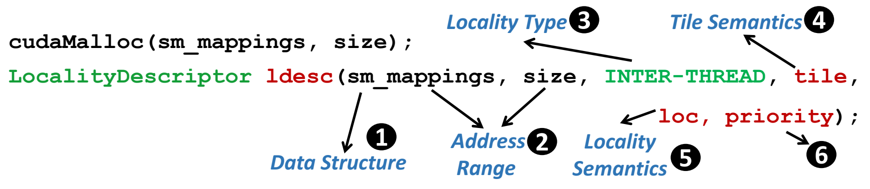

Data Tiles
5
Access Pattern
3
CTAs along the Y dim share the same data
4
Y dim
X dim
Data Structure (sm_mappings)
1
Y dim
X dim
CTA Compute Grid
2

Figure 7: Inter-CTA data locality in `histo` (Parboil).

knowledge of *key program semantics*. Figure 8 shows how the programmer or compiler can use the Locality Descriptor to leverage both reuse-based locality and NUMA locality. We briefly summarize how the Locality Descriptor works here, and provide an end-to-end description in the rest of the chapter.



```
cudaMalloc(sm_mappings, size);
LocalityDescriptor ldesc(sm_mappings, size, INTER-THREAD, tile,
                                                   loc, priority);
```

Figure 8: The Locality Descriptor specification for `histo`.

First, each instance of a Locality Descriptor describes a single *data structure's* locality characteristics (in Figure 8, `sm_mappings` ❶) and conveys the corresponding address range ❷. Second, we define several fundamental *locality types* as a contract between the software and the architecture. The locality type, which can be `INTER-THREAD`, `INTRA-THREAD`, or `NO-REUSE`, drives the underlying optimizations used to exploit it. In `histo`, `INTER-THREAD` ❸ describes inter-CTA locality. The locality type ❸ and the *locality semantics* ❺, such as access pattern, inform the architecture to use CTA scheduling and other techniques that exploit the corresponding locality type (described in §3.3.3). Third, we partition the data structure into *data tiles* that are used to relate data to the threads that access it. In Figure 7, each data range that has the same color (and is hence accessed by the same set of CTAs) forms a data tile. Data tiles and the threads they access are described by the *tile semantics* ❹ (§3.3.3), which informs the architecture *which CTAs* to schedule together and *which data* to place at the same NUMA zone. Fourth, we use a software-provided *priority* ❻ to reconcile optimizations between Locality Descriptors for different data structures in the same program if they require *conflicting* optimizations (e.g., different CTA scheduling strategies).

We evaluate the benefits of using Locality Descriptors to exploit different forms of both *reuse-*

based locality and *NUMA* locality. We demonstrate that Locality Descriptors effectively leverage program semantics to improve performance by 26.6% on average (up to 46.6%) when exploiting reuse-based locality in the cache hierarchy, and by 53.7% (up to 2.8X) when exploiting NUMA locality.

The major **contributions** of this work are:

- This is the first work to propose a holistic cross-layer approach to explicitly *express* and *exploit* data locality in GPUs as a first class entity in both the programming model and the hardware architecture.
- We design the Locality Descriptor, which enables *(i)* the software/programmer to describe data locality in an architecture-agnostic manner and *(ii)* the architecture to leverage key program semantics and coordinate many architectural techniques transparently to the software. We architect an end-to-end extensible design to connect five architectural techniques (CTA scheduling, cache bypassing, cache prioritization, data placement, prefetching) to the Locality Descriptor programming abstraction.
- We comprehensively evaluate the efficacy and versatility of the Locality Descriptor in leveraging different types of reuse-based and NUMA locality, and demonstrate significant performance improvements over state-of-the-art approaches.

3.2 Motivation

We use two case studies to motivate our work: *(i)* Inter-CTA locality, where different CTAs access the same data and *(ii)* NUMA locality in a GPU with a NUMA memory system.

3.2.1 Case Study 1: Inter-CTA Locality

A GPU kernel is formed by a *compute grid*, which is a 3D grid of Cooperative Thread Arrays (CTAs). Each CTA, in turn, comprises a 3D array of threads. Threads are scheduled for execution at each Streaming Multiprocessor (SM) at a CTA granularity. Inter-CTA locality [63, 115, 183, 189, 194, 345, 354] is data reuse that exists when multiple CTAs access the same data. CTA scheduling [63, 115, 183, 189, 194, 345, 354] is a technique that is used to schedule CTAs that share data at the same SM to exploit inter-CTA locality at the per-SM local L1 caches.

To study the impact of CTA scheduling, we evaluate 48 scheduling strategies, each of which groups (i.e., clusters) CTAs differently: either along the grid's X, Y, or Z dimensions, or in different combinations of the three. The goal of CTA scheduling for locality is to maximize sharing between CTAs at each SM and effectively *reduce* the amount of data accessed by each SM. Hence, as a measure of how well CTA scheduling improves locality for each workload, in Figure 9 we plot the *minimum* working set at each SM (normalized to baseline) across all 48 scheduling strategies. We define *working set* as the average number of uniquely accessed cache lines at each SM. A smaller working set implies fewer capacity misses, more sharing, and better locality. Figure 9 also shows the *maximum* performance improvement among all evaluated scheduling strategies for each benchmark.

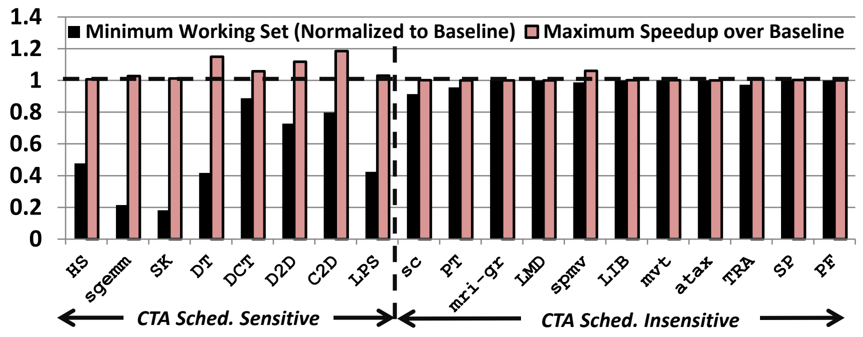


Figure 9: CTA scheduling: performance and working set.

Figure 9 shows that even though CTA scheduling significantly reduces the working set of CTA-scheduling-sensitive applications (on the left) by 54.5%, it has almost no impact on performance (only 3.3% on average across all applications). To understand this minimal impact on performance, in Figure 10 we plot the corresponding increase in L1 hit rate for the specific scheduling strategy that produced the smallest working set (only for the scheduling-sensitive workloads). We also plot the increase in *inflight hit rate*, which we measure as the number of MSHR hits, i.e., another thread already accessed the same cache line, but the line has not yet been retrieved from memory and hits at the MSHRs.

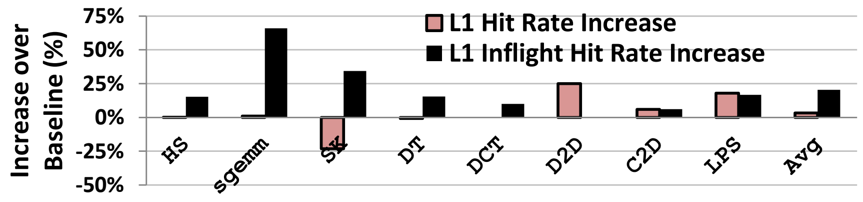


Figure 10: CTA scheduling: L1 hit rate and L1 inflight hit rate.

Figure 10 shows that CTA scheduling has little impact in improving the L1 hit rate (by 3% on average) with the exception of D2D and C2D. This explains the minimal performance impact. CTA scheduling, however, a substantially increases L1 inflight hit rate (by 20% on average). This indicates that even though there is higher data locality due to more threads sharing the same data, these threads wait for the same data at the *same time*. As a result, the increased locality simply causes more threads to *stall*, rather than improving hit rate. Hence, while CTA scheduling is very effective in *exposing* data locality, we still need to address other challenges (e.g., threads stalling together) to obtain performance gains from improved data locality. Furthermore, determining *which* scheduling strategy to use is another challenge, as each application requires a *different* strategy to maximize locality based on the program's sharing pattern.

In summary, to exploit inter-CTA locality *(i)* the hardware-controlled CTA scheduler needs to know *which CTAs access the same data*, to choose an appropriate scheduling strategy (this requires knowledge of program semantics) and *(ii)* a scheduling strategy that *exposes* locality in the cache is *not* necessarily sufficient for translating locality into performance (we need to coordinate other techniques).

3.2.2 Case Study 2: NUMA Locality

For continued scaling, future GPUs are expected to employ non-uniform memory access (NUMA) memory systems. This can be in the form of multiple memory stacks [131, 171], unified virtual address spaces in multi-GPU/heterogeneous systems [8, 53, 54, 174, 187, 281, 386] or multi-chip modules, where SMs and memory modules are partitioned into *NUMA zones* or *multiple GPU modules* [25, 228]. Figure 11 depicts the system evaluated in [25] with four NUMA zones. A request to a remote NUMA zone goes over the lower bandwidth inter-module interconnect, has higher latency, and incurs more traffic compared to *local requests* [25]. To maximize performance and efficiency, we need to control *(i)* how data is placed across NUMA zones and *(ii)* how CTAs are scheduled to maximize local accesses.

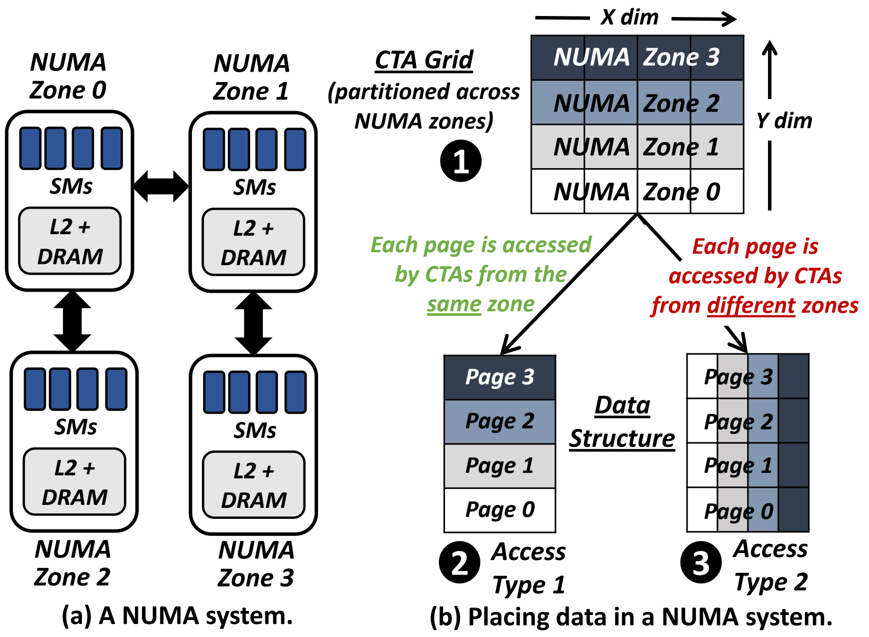


Figure 11: NUMA locality.

To understand why this is a challenging task, let us consider the heuristic-based hardware mechanism proposed in [25], where the CTA grid is partitioned across the 4 NUMA zones such that contiguous CTAs are scheduled at the same SM. Data is placed at the page granularity (64KB) at the NUMA zone where it is *first accessed*, based on the heuristic that consecutive CTAs are likely to share the same page(s). Figure 11 depicts a CTA grid (❶), which is partitioned between

NUMA zones in this manner—consecutive CTAs along the X dimension are scheduled at the same NUMA zone. This heuristic-based mechanism works well for Access Type 1 (❷), where CTAs that are scheduled at the same NUMA zone access the same page(s) (the color scheme depicts which CTAs access what data). However, for Access Type 2 (❷), this policy fails as a single page is shared by CTAs that are scheduled at *different zones*. Two challenges cause this policy to fail. First, suboptimal scheduling: the simple scheduling policy [25] does not *always* co-schedule CTAs that share the same pages at the same zone. This happens when scheduling is *not* coordinated with the application's access pattern. Second, large and fixed page granularity: more CTAs than what can be scheduled at a single zone may access the *same* page. This happens when there are fine-grained accesses by *many* CTAs to each page and when different data structures are accessed by the CTAs in *different* ways. For these reasons (as we evaluate in §3.6.2), a heuristic-based approach is often ineffective at exploiting NUMA locality.

3.2.3 Other Typical Locality Types

We describe other locality types, caused by different access patterns, and require other optimizations for locality next.

Inter-warp Locality. Inter-warp locality is data reuse between warps that belong to the same/different CTAs. This type of locality occurs in stencil programs (workloads such as `hotspot` [61] and `stencil` [313]), where each thread accesses a set of neighboring data elements, leading to data reuse between neighboring warps. Inter-warp locality is also a result of misaligned accesses to cache lines by threads in a warp [180, 181, 188], since data is always fetched at the cache line granularity (e.g., `streamcluster` [61] and `backprop` [61]). Inter-warp locality has short reuse distances [181] as nearby warps are typically scheduled together and caching policies such as LRU can exploit a significant portion of this locality. However, potential for improvement exists using techniques such as inter-warp prefetching [188, 194, 250] or CTA scheduling to co-schedule CTAs that share data [194].

Intra-thread Locality. This is reuse of data by the *same thread* (seen in `LIBOR` [244] and `lavaMD` [61]), where each thread operates on its own working set. Local memory usage in the program is also an example of this type of locality. The key challenge here is *cache thrashing* because *(i)* the overall working set of workloads with this locality type is large due to lack of sharing among threads and *(ii)* the reuse distance per thread is large as hundreds of threads are swapped in and out by the GPU's multithreading before the data is reused by the same thread. Techniques that have been proposed to address cache thrashing include cache bypassing or prioritization (e.g. pinning) of different forms [26, 30, 191, 195–197, 199, 229, 325, 357, 358, 383] and/or warp/CTA throttling [163, 164, 194, 274].

3.2.4 Key Takeaways & Our Goal

In summary, locality in GPUs can be of different forms depending on the GPU program. Each locality type presents *different* challenges that need to be addressed. Tackling each challenge often requires coordination of *multiple* techniques (such as CTA scheduling and cache bypassing), many of which software has no easy access to. Furthermore, to be effective, some of these techniques (e.g., CTA scheduling, memory placement) require knowledge of *program semantics*, which is prohibitively difficult to infer at run time.

Our goal is to design a holistic *cross-layer* abstraction—that all levels of the compute stack recognize—to express and exploit the different forms of data locality. Such an abstraction should enable connecting a range of architectural techniques with the locality properties exhibited by the program. In doing so, the abstraction should *(i)* provide the programmer/software a simple, yet powerful interface to express data locality and *(ii)* enable architectural techniques to leverage key program semantics to optimize for locality.

3.3 Locality Descriptor: Abstraction

Figure 12 depicts an overview of our proposed abstraction. The goal is to connect program semantics and programmer intent (❶) with the underlying architectural mechanisms (❷). By doing so, we enable optimization at different levels of the stack: *(i)* as an additional knob for static code tuning by the programmer, compiler, or autotuner (❸), *(ii)* runtime software optimization (❹), and *(iii)* dynamic architectural optimization (❼) using a combination of architectural techniques. This abstraction interfaces with a parallel GPU programming model like CUDA (❺) and conveys key program semantics to the architecture through low overhead interfaces (❻).

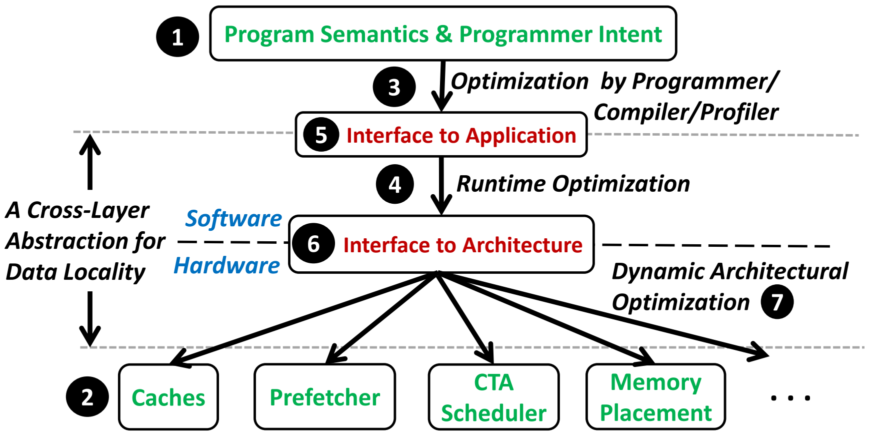

1
Program Semantics & Programmer Intent
3
Optimization by Programmer/ Compiler/Profiler
5 Interface to Application
4
Runtime Optimization
A Cross-Layer Abstraction for Data Locality
Software
Hardware
6 Interface to Architecture
Dynamic Architectural Optimization 7
2
Caches
Prefetcher
CTA Scheduler
Memory Placement
. . .

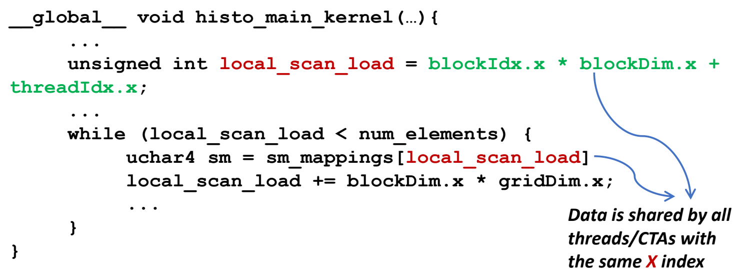
Figure 12: Overview of the proposed abstraction.

3.3.1 Design Goals

We set three goals that drive the design of our proposed abstraction: *(i) Supplemental and hint-based only*: The abstraction should be an optional add-on to optimize for *performance*, requiring no change to the rest of the program, nor should it impact the program's *correctness*. *(ii) Architecture-agnosticism*: The abstraction should abstract away any low-level details of the architecture (e.g., cache size, number of SMs, caching policy). Raising the abstraction level improves portability, reduces programming effort, and enables architects to flexibly design and improve techniques across GPU generations, transparently to the software. *(iii) Generality and flexibility*: The abstraction should flexibly describe a wide range of locality types typically seen in GPU programs. It should be easy to extend what the abstraction can express and the underlying architectural techniques that can benefit from it.

3.3.2 Example Program

We describe the Locality Descriptor with the help of the `histo` (Parboil [313]) workload example described in §1. We begin with an overview of how a Locality Descriptor is specified for `histo` and then describe the key ideas behind the Locality Descriptor's components. Figure 13 depicts a code example from this application. The primary data structure is `sm_mappings`, which is indexed by a function of the thread and block index only along the X dimension. Hence, the threads that have the same index along the X dimension access the same part of this data structure.



```
__global__ void histo_main_kernel(…){
     ...
     unsigned int local_scan_load = blockIdx.x * blockDim.x +
threadIdx.x;
     ...
     while (local_scan_load < num_elements) {
          uchar4 sm = sm_mappings[local_scan_load]
          local_scan_load += blockDim.x * gridDim.x;
          ...
     }
}
```

Figure 13: Code example from `histo` (Parboil).

Figure 14 depicts the data locality in this application in more detail. ① is the CTA grid and ② is the `sm_mappings` data structure. The CTAs that are colored the same access the same data range (also colored the same). As §1 discusses, in order to describe locality with the Locality Descriptor abstraction, we partition each data structure in *data tiles* ③ that group data shared by the same CTAs. In addition, we partition the CTA grid along the X dimension into *compute tiles* ④ to group together CTAs that access the same data tile. We then relate the compute and data tiles with a *compute-data* mapping ⑤ to describe which compute tile accesses which data tile. Figure 15 depicts the code example to express the locality in this example with a Locality Descriptor. As §1 describes, the key components of a Locality Descriptor are: the associated data structure (❶), its locality type (❷), tile semantics (❸), locality semantics (❹), and its priority (❺). We now describe each component in detail.

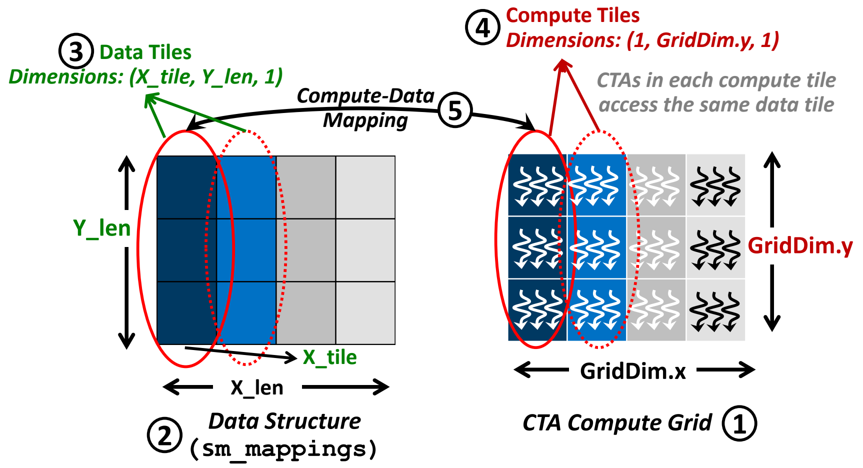
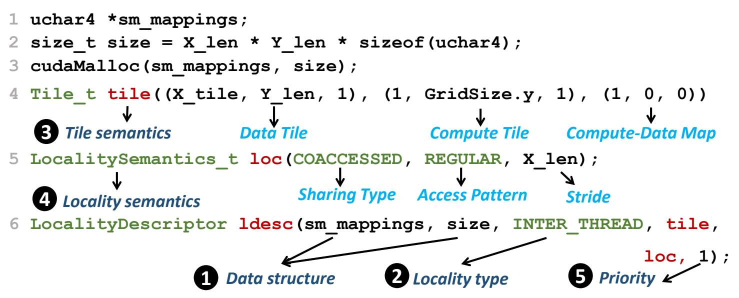

③ Data Tiles
Dimensions: (X_tile, Y_len, 1)
④ Compute Tiles
Dimensions: (1, GridDim.y, 1)
CTAs in each compute tile access the same data tile
Compute-Data Mapping ⑤
Y_len
X_tile
X_len
GridDim.y
GridDim.x
② Data Structure (sm_mappings)
CTA Compute Grid ①

Figure 14: Data locality and compute/data tiles in `histo`.



```
uchar4 *sm_mappings;
size_t size = X_len * Y_len * sizeof(uchar4);
cudaMalloc(sm_mappings, size);
Tile_t tile((X_tile, Y_len, 1), (1, GridSize.y, 1), (1, 0, 0))
LocalitySemantics_t loc(COACCESSED, REGULAR, X_len);
LocalityDescriptor ldesc(sm_mappings, size, INTER_THREAD, tile,
                                                   loc, 1);
```

Figure 15: Locality Descriptor example for `histo`.

3.3.3 An Overview of Key Ideas and Components

Figure 16 shows an overview of the components of the Locality Descriptor. We now describe the five key components and the key insights behind their design.

Data Structure (❶). We build the abstraction around the program's *data structures* (each specified with its base address and size). Each instance of the Locality Descriptor describes the locality characteristics of a single data structure. Designing the Locality Descriptor around the program's data structures is advantageous for two reasons. First, it ensures *architecture-agnosticism* as a data structure is a software-level concept, easy for the programmer to reason about. Second, it is natural to tie locality properties to data structures because in GPU programming models, *all* threads typically access a given data structure in the same way. For example, some data structures are simply streamed through by all threads with no reuse. Others are heavily reused by groups of threads.

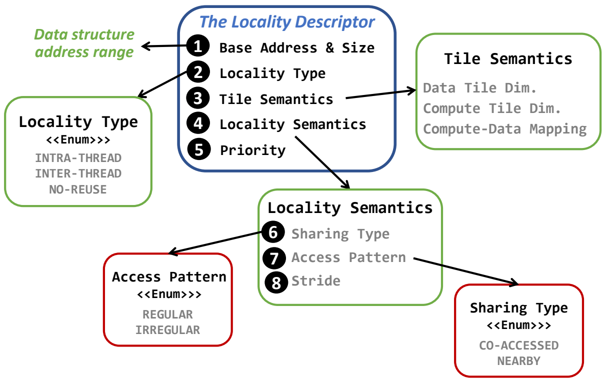


Figure 16: Overview of the Locality Descriptor.

Locality Type (❷). Each instance of the Locality Descriptor has an explicit *locality type*, which forms a contract or *basis of understanding* between software and hardware. This design choice leverages the known observation that locality type often determines the underlying optimization mechanisms (§3.2). Hence, software need only specify locality type and the system/architecture transparently employs a different set of architectural techniques based on the specified type. We provide three fundamental types: *(i)* INTRA-THREAD: when the reuse of data is by the same thread itself, *(ii)* INTER-THREAD: when the reuse of data is due to sharing of data between different threads (e.g., inter-warp or inter-CTA locality), and *(iii)* NO-REUSE: when there is no reuse of data (NUMA locality can still be exploited, as described below). If a data structure has multiple locality types (e.g., if a data structure has both *intra-thread* and *inter-thread* reuse), multiple Locality Descriptors with different types can be specified for that data structure. We discuss how these cases are handled in §3.4.

Tile Semantics (❸). As data locality is essentially the outcome of how computation accesses data, we need to express the *relation between compute and data*. To do this, we first need a *unit* of computation and data as a basis. To this end, we partition the data structure into a number of *data tiles (D-Tiles)* and the compute grid into a number of *compute tiles (C-Tiles)*. Specifically, a D-Tile is a 3D range of data elements in a data structure and a C-Tile is a 3D range of threads or CTAs in the 3D compute grid (e.g., ③ and ④ in Figure 14).

This design provides two major benefits. First, it provides a *flexible and architecture-agnostic* scheme to express locality types. For example, to express INTER-THREAD locality, a D-Tile is the range of data shared by a set of CTAs; and each such set of CTAs forms a C-Tile. To express INTRA-THREAD locality, a C-Tile is just a single thread and the D-Tile is the range of data that is reused by that single thread. Second, such decomposition is *intrinsic* and conceptually similar to the existing hierarchical tile-based GPU programming model. Tile partitioning can hence be done easily by the programmer or the compiler using techniques such as [53, 54]. For irregular data structures (e.g, graphs), which cannot be easily partitioned, the Locality Descriptor can be used to describe the entire data structure. This imposes little limitation as such data structures exhibit an irregular type of locality that cannot be easily described by software.

We further reduce complexity in expression by stipulating only an *imprecise* description of locality. There are two primary instances of this. First, we use a simple 1:1 mapping between C-Tile and D-Tile. This is a non-limiting simplification because data locality is fundamentally about *grouping* threads and data based on sharing. If multiple C-Tiles access the same D-Tile, a bigger C-Tile should simply be specified. In an extreme case, where the entire data structure is shared by all threads, we should only have one C-Tile and one D-Tile. In another case, where there is only intra-thread locality (no sharing among threads), there is a natural 1:1 mapping between each thread and its working set. This simplification would be an approximation in cases with irregular sharing, which is out of the Locality Descriptor's scope because of the high complexity and low potential. Second, C-Tile and D-Tile partitioning implies that the grouping of threads and data needs to be *contiguous*—a C-Tile cannot access a set of data elements that is interleaved with data accessed by a different C-Tile. This is, again, a non-limiting requirement as contiguous data elements are typically accessed by neighboring threads to maximize spatial locality and reduce memory traffic. If there is an interleaved mapping between C-Tiles and the D-Tiles they access, the C-Tiles and D-Tiles can be approximated by merging them into bigger tiles until they are contiguous. This design drastically reduces the expression complexity and covers typical GPU applications.

Specifically, the tile semantics are expressed in three parts: *(i) D-Tile dimensions*: The number of data elements (in each dimension) that form a D-Tile. Depending on the data structure, the unit could be any data type. In the `histo` example (③ in Figure 14), the D-Tile dimensions are `(X_tile, Y_len, 1)`, where `X_tile` is the range accessed by a single C-Tile along the X dimension, `Y_len` is the full length of the data structure (in data elements) along the Y dimension. *(ii) C-Tile dimensions*: The number of CTAs in each dimension that form a C-Tile. The compute tile dimensions in the `histo` example (④ in Figure 14) are `(1, GridDim.y, 1)`: 1 CTA along the X dimension, `GridDim.y` is the length of the whole grid along the Y dimension, and since this is a 2D grid, the Z dimension is one. *(iii) Compute-data map*: We use a simple function to rank which order to traverse C-Tiles first in the 3D compute grid as we traverse the D-Tiles in a data structure in $X \to Y \to Z$ order. For example, the mapping function `(3,1,2)` implies that when D-Tiles are traversed in the $X \to Y \to Z$ order, and the C-Tiles are traversed in the $Y \to Z \to X$ order. In our `histo` example, this mapping (⑤ in Figure 14) is simply `(1,0,0)` as the C-Tiles need only be traversed along the X dimension. This simple function saves runtime overhead, but more complex functions can also be used.

Locality Semantics (❹). This component describes the type of reuse in the data structure as well as the access pattern. §3.4 describes how this information is used for optimization. This component has two parts: Sharing Type (❻) and Access Pattern (❼). There are two options for

Sharing Type (❺) to reflect the typical sharing patterns. `COACCESSED` indicates that the *entire* D-Tile is shared by all the threads in the corresponding C-Tile. `NEARBY` indicates that the sharing is more irregular, with nearby threads in the C-Tile accessing nearby data elements (the form of sharing seen due to misaligned accesses to cache lines or stencil-like access patterns [181, 194]). Sharing type can be extended to include other sharing types between threads (e.g., partial sharing). Access Pattern (❻) is primarily used to inform the prefetcher and includes whether the access pattern is `REGULAR` or `IRREGULAR`, along with a stride (❽) within the D-Tile for a `REGULAR` access pattern.

Priority (❺). Multiple Locality Descriptors may require *conflicting* optimizations (e.g., different CTA scheduling strategies). We ensure that these conflicts are rare by using a conflict resolution mechanism described in §3.4. When a conflict *cannot* be resolved, we use a software-provided *priority* to give precedence to certain Locality Descriptors. This design gives the software more control in optimization, and ensures the key data structure(s) are prioritized. This priority is also used to give precedence to a certain *locality type*, when there are multiple Locality Descriptors with different types for the same data structure.

3.4 Locality Descriptor: Detailed Design

We detail the design of the programmer/compiler interface (❺ in Figure 12), runtime optimizations (❹), and the architectural interface and mechanisms (❷ and ❼)—CTA scheduling, memory placement, cache management, and prefetching.

3.4.1 The Programmer/Compiler Interface

The Locality Descriptor can be specified in the code after the data structure is initialized and copied to global memory. Figure 15 is an example. If the semantics of a data structure change between kernel calls, its Locality Descriptor can be re-specified between kernel invocations.

The information to specify the Locality Descriptor can be extracted in three ways. First, the compiler can use static analysis to determine forms of data locality, *without* programmer intervention, using techniques like [63, 194] for inter-CTA locality. Second, the programmer can annotate the program (as was done in this work), which is particularly useful when the programmer wishes to hand-tune code for performance and to specify the *priority* ordering of data structures when resolving potential optimization conflicts (§3.3.3). Third, software tools such as auto-tuners or profilers [82, 168, 291] can determine data locality and access patterns via dynamic analysis.

During compilation, the compiler extracts the variables that determine the address range of each Locality Descriptor, so the system can resolve the virtual addresses at run time. The compiler then summarizes the Locality Descriptor semantics corresponding to these address ranges and places this information in the object file.

3.4.2 Runtime Optimization

At run time, the GPU driver and runtime system determine how to exploit the locality characteristics expressed in the Locality Descriptors based on the specifics of the underlying architectural components (e.g., number of SMs, NUMA zones). Doing so includes determining the: *(i)* CTA scheduling strategy, *(ii)* caching policy (prioritization and bypassing), *(iii)* data placement strategy across NUMA zones, and *(iv)* prefetching strategy. In this work, we provide an algorithm to coordinate these techniques. Both, the set of techniques used and the algorithm to coordinate them, are extensible. As such, more architectural techniques can be added and the algorithm can be enhanced. We first describe the algorithm that determines *which* architectural techniques are employed for different locality types, and then detail each architectural technique in the following subsections.

Figure 17 depicts the flowchart that determines which optimizations are employed. The algorithm depicted works based on the three *locality types*. First, for `INTER-THREAD`Locality Descriptors, we employ CTA scheduling (§3.4.3) to expose locality. We also use other techniques based on the *access pattern* and the *sharing type*: *(i)* For `COACCESSED` sharing with a `REGULAR` access pattern, we use guided stride prefetching (§3.4.5) to overlap the long latencies when many threads are stalled together waiting on the same data; *(ii)* For `COACCESSED` sharing with a `IRREGULAR` access pattern, we employ cache prioritization using *soft pinning* (§3.4.4) to keep data in the cache long enough to exploit locality; *(iii)* For `NEARBY` sharing, we use simple nextline prefetching tailored to the frequently-occurring access pattern. Second, for an `INTRA-THREAD` Locality Descriptor, we employ a thrash-resistant caching policy, *hard pinning* (§3.4.4), to keep a part of the working set in the cache. Third, for a `NO-REUSE` Locality Descriptor, we use cache bypassing as the data is not reused. In a NUMA system, *irrespective of the locality type*, we employ CTA scheduling and memory placement to minimize accesses to remote NUMA zones. If there are conflicts between different data structures, they are resolved using the priority order, as described in §3.4.3 and §3.4.6.

3.4.3 CTA Scheduling

Figure 18 depicts an example of CTA scheduling for the CTA grid (❶) from our example (`histo`, §3.3.2). The default CTA scheduler (❷) traverses one dimension at a time ($X \rightarrow Y \rightarrow Z$), and schedules CTAs at each SM in a round robin manner, ensuring load balancing across SMs. Since this approach does *not* consider locality, the default scheduler schedules CTAs that access the same data at *different* SMs (❷).

The Locality Descriptor guided CTA scheduling (❸) shows how we expose locality by grouping

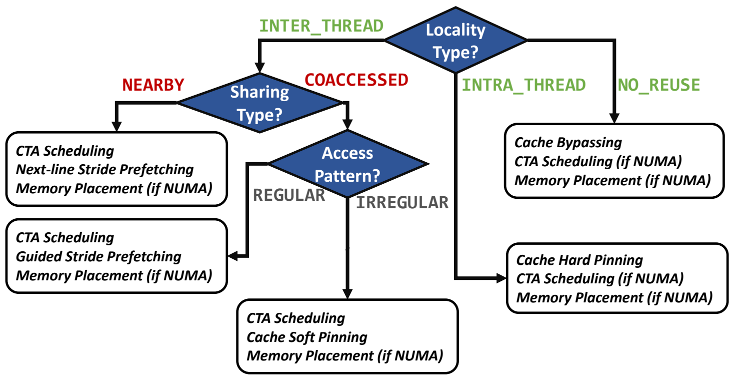


Figure 17: Flowchart of architectural optimizations leveraging Locality Descriptors.

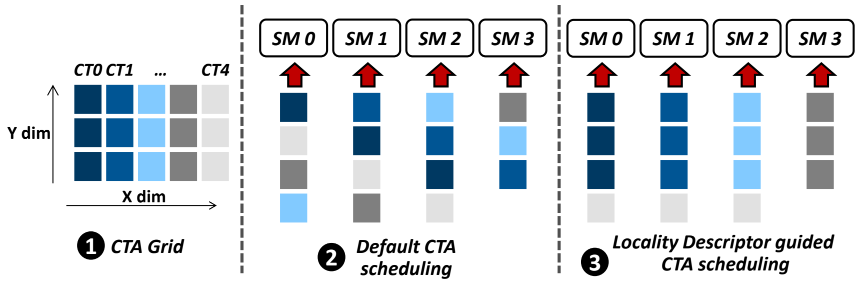


Figure 18: CTA scheduling example.

CTAs in each C-Tile into a cluster. Each cluster is then scheduled at the same SM. In this example, we spread the last C-Tile (CT4) across three SMs to trade off locality for parallelism. To enable such application-specific scheduling, we need an algorithm to use the Locality Descriptors to drive a CTA scheduling policy that *(i)* schedules CTAs from the same C-Tile (that share data) together to expose locality, *(ii)* ensures all SMs are fully occupied, and *(iii)* resolves conflicts between multiple Locality Descriptors. We use Algorithm 1 to form *CTA clusters*, and schedule each formed cluster at the same SM in a non-NUMA system.[2] In a NUMA system, we first *partition* the CTAs across the different NUMA zones (see §3.4.6), and then use Algorithm 1 *within* each NUMA zone.

The algorithm first ensures that each Locality Descriptor has enough C-Tiles for all SMs. If that is not the case, it `splits` C-Tiles (lines 3–7), to ensure we have enough clusters to occupy all SMs. Second, the algorithm uses the C-Tiles of the highest priority Locality Descriptor as the initial CTA clusters (line 8), and then attempts to `merge` the lower-priority Locality Descriptors (lines 9–16).[3] *Merging* tries to find a cluster that also groups CTAs with shared data in other lower-priority Locality Descriptors while keeping the clusters larger than the number of SMs (first step). By scheduling the merged cluster at each SM, the system can expose locality for multiple data structures. The GPU driver runs Algorithm 1 before launching the kernel to determine the

[2]This algorithm optimizes only for the L1 cache, but it can be extended to optimize for the L2 cache as well.

[3]Although the algorithm starts by optimizing the highest priority XMem, it is designed to find a scheduling strategy that is optimized for *all* XMems. Only when no such strategy can be found (i.e., when there are conflicts), is the highest priority XMem prioritized over others.

Algorithm 1 Forming CTA clusters using Locality Descriptors

```
1: Input: $LDesc_{1...N}$: all N Locality Descriptors, sorted by priority (highest first)
2: Output: $CLS = (CLS_X, CLS_Y, CLS_Z)$: the final cluster dimensions
3: for $i = 1$ to $N$ do    ▷ Step 1: Split C-Tiles into 2 to ensure each LDesc has enough C-Tiles for all SMs (to load balance)
4:     while CT_NUM($LDesc_i$) < $SM_{NUM}$ and CT_DIM($LDesc_i$) != (1, 1, 1,) do
5:         Divide the C-Tile of $LDesc_i$ into 2 along the largest dimension
6:     end while
7: end for
8: $CLS \leftarrow$ CT_DIM($LDesc_1$)    ▷ Each cluster is now formed by each of the highest priority LDesc's C-Tiles after splitting
9: for $i = 2$ to $N$ do    ▷ Step 2: Merge the C-Tiles of lower priority LDescs to form larger clusters to also leverage locality from lower priority LDescs
10:     for $d$ in $(X, Y, Z)$ do
11:         $MCLS_d \leftarrow CLS_d \times$ MAX (FLOOR(CT_DIM($LDesc_i$) / $CLS_d$) , 1)    ▷ Merge C-Tiles along each dimension
12:     end for
13:     if CT_NUM($MCLS$) $\geq SM_{NUM}$ then    ▷ Ensure there are enough C-Tiles for all SMs
14:         $CLS \leftarrow MCLS$
15:     end if
16: end for
```

CTA scheduling policy.

3.4.4 Cache Management

The Locality Descriptor enables the cache to distinguish reuse patterns of different data structures and apply policies accordingly. We use two caching mechanisms that can be further extended. First, *cache bypassing* (e.g., [191, 195–197, 199, 229, 325, 357, 358, 383]), which does not insert data that has no reuse (`NO-REUSE` locality type) into the cache. Second, *cache prioritization*, which inserts some data structures into the cache with higher priority than the others. We implement this in two ways: *(i)* hard pinning and *(ii)* soft pinning. *Hard pinning* is a mechanism to prevent cache thrashing due to large working sets by ensuring that part of the working set stays in the cache. We implement hard pinning by inserting all hard-pinned data with the highest priority and evicting a specific cache way (e.g., the 0th way) when *all* cache lines in the same set have the highest priority. Doing so protects the cache lines in other cache ways from being repeatedly evicted. We use a timer to automatically reset all priorities to *unpin* these pinned lines periodically. *Soft pinning*, on the other hand, simply prioritizes one data structure over others without any policy to control thrashing. As §3.4.2 discusses, we use hard pinning for data with `INTRA-THREAD` locality type, which usually has a large working set as there is very limited sharing among threads. We use soft pinning for data with `INTER-THREAD` locality type to ensure that this data is retained in the cache until other threads that share the data access it.

3.4.5 Prefetching

As §3.2 discusses, using CTA scheduling alone to expose locality hardly improves performance, as the increased locality causes more threads to stall, waiting for the *same critical data* at the *same time* (see the L1 inflight hit rate, Figure 9). As a result, the memory latency to this critical data becomes the performance bottleneck, since there are too few threads left to hide the memory latency. We address this problem by employing a hardware prefetcher guided by the Locality Descriptor to prefetch the *critical data* ahead of time. We employ a prefetcher only for `INTER-THREAD` Locality Descriptors because the data structures they describe are shared by multiple threads, and hence, are more critical to avoid stalls. The prefetcher is triggered when an access misses the cache on these data structures. The prefetcher is instructed based on the access pattern and the sharing type. As Figure 17 shows, there are two cases. First, for `NEARBY` sharing, the prefetcher is directed to simply prefetch the next cache line. Second, for `COACCESSED` sharing with a `REGULAR` access pattern, the prefetched address is a function of *(i)* the access stride, *(ii)* the number of bytes that are accessed at the same time (i.e., the width of the data tile), and, *(iii)* the size of the cache, as prefetching too far ahead means more data needs to be retained in the cache. The address to prefetch is calculated as: `current address + (L1_size/(number_of_active_tiles*data_tile_width) * stride)`. The `number_of_active_tiles` is the number of D-Tiles that the prefetcher is actively prefetching. The equation decreases the prefetch distance when there are more active D-Tiles to reduce thrashing. This form of controlled prefetching avoids excessive use of memory bandwidth by only prefetching data that is shared by many threads, and has high accuracy as it is informed by the Locality Descriptor.

3.4.6 Memory Placement

As §3.2.2 discusses, exploiting locality on a NUMA system requires coordination between CTA scheduling and memory placement such that CTAs access local data within each NUMA zone. There are two major challenges (depicted in Figure 11 in §3.2.2): *(i)* how to partition data among NUMA zones at a fine granularity. A paging-based mechanism (e.g., [25]) does not solve this problem as a large fixed page size is typically ineffective (§3.2.2), while small page sizes are prohibitively expensive to manage [29], and *(ii)* how to partition CTAs among NUMA zones to exploit locality among *multiple* data structures that may be accessed differently by the CTAs in the program. To address these two challenges, we use a *flexible* data mapping scheme, which we describe below, and a CTA partitioning algorithm that leverages this scheme.

Flexible Fine-granularity Data Mapping. We enhance the mapping between physical addresses and NUMA zones to enable data partitioning at a flexible granularity, smaller than a page (typically 64KB). Specifically, we use consecutive bits within the physical address itself to index the NUMA zone (similar to [131] in a different context). We allow using a *different* set of bits for different data structures. Thus, each data structure can be partitioned across NUMA zones at a *different* granularity.[4] Figure 19 shows how this is done for the example in §3.2.2. As the figure shows, CTAs in each NUMA zone ❶ access the same page (64KB) for data structure A ❷, but they

[4]We limit the bits that can be chosen to always preserve the minimum DRAM burst size (128B) by always specifying a size between 128B-64KB (bits 7-16). We always use bit 16/17 for granularities larger than 64KB as we can flexibly map virtual pages to the desired NUMA zone using the page table. We enable flexible bit mapping by modifying the hardware address decoder in the memory controller.

only access the same *quarter-page (16KB)* for data structure B ❸. If we partition data across NUMA zones only at the page granularity [25], most accesses to data structure B would access remote NUMA zones. With our mechanism, we can choose bits 16-17 (which interleaves data between NUMA zones at a 64KB granularity) and bits 14-15 (which interleaves data at a 16KB granularity) in the physical address to index the NUMA zone for data structures A and B respectively. Doing so results in all accesses to be in the *local* NUMA zone for *both* data structures.

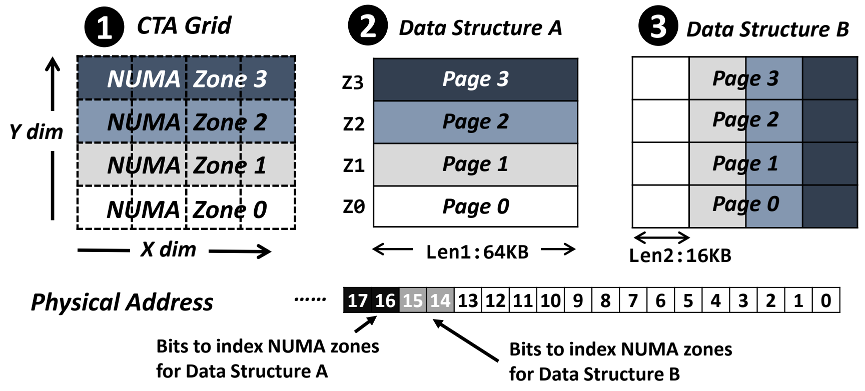


Figure 19: Memory placement with Locality Descriptors.

This design has two constraints. First, we can partition data only at a power-of-two granularity. Our findings, however, show that this is not a limiting constraint because *(i)* regular GPU kernels typically exhibit power-of-two strides across CTAs (consistent with [131]); and *(ii)* even with non-power-of-two strides, this approach is still reasonably effective compared to page-granularity placement (as shown quantitatively in §3.6.2). Second, to avoid cases where a data structure is *not* aligned with the interleaving granularity, we require that the GPU runtime align data structures at the page granularity.

CTA Scheduling Coordinated with Data Placement. To coordinate memory placement with CTA scheduling, we use a simple greedy search algorithm (Algorithm 2) that partitions the CTAs across the NUMA zones and selects the most effective address mapping bits for each data structure. We provide a brief overview here.

The algorithm evaluates the efficacy of all possible address mappings for the data structure described by the highest-priority Locality Descriptor (line 4). This is done by determining which N consecutive bits between bit 7-16 in the physical address are the most effective bits to index NUMA zones for that data structure (where N is the base-2-log of the number of NUMA zones). To determine which mapping is the most effective, the algorithm first determines the corresponding CTA partitioning scheme for that address mapping using the NUMA_PART function (line 5). The NUMA_PART function simply schedules each C-Tile at the NUMA zone where the D-Tile it accesses is placed (based on the address mapping that is being tested). The 1:1 C-Tile/D-Tile compute mapping in the Locality Descriptor gives us the information to easily do this. To evaluate the effectiveness or *utility* of each address mapping and the corresponding CTA partitioning scheme, we use the `COMP_UTIL` function (line 7). This function calculates the ratio of local/remote accesses for each mapping.

Algorithm 2 CTA partitioning and memory placement for NUMA

```
1: Input: $LDesc_{1...N}$: all N Locality Descriptors, sorted by priority (highest first)
2: Output 1: $CTA\_NPART$: the final CTA partitioning for NUMA zones
3: Output 2: $MAP_{1...N}$: the address mapping bits for each $LDesc$
4: for $b\_hi = 7$ to 16 do                                    ▷ Test all possible mappings for the highest-priority LDesc
5:     $CTA\_PART_{b\_hi} \leftarrow$ NUMA_PART($LDesc_1$, $b\_hi$)    ▷ Partition the CTAs based on the address mapping being evaluated
6:     $best\_util\_all \leftarrow 0$                           ▷ best_util_all: the current best utility
7:     $util_{b\_hi} \leftarrow$ COMP_UTIL($N$, $LDesc_1$, $CTA\_PART_{b\_hi}$, $b\_hi$)    ▷ Calculate the utility of the CTA partitioning scheme + address mapping
8:     for $i$ = 2 to $N$ do                                    ▷ Test other LDescs
9:         $TMAP_i \leftarrow 7$                                ▷ TMAP: temporary mapping
10:        $best\_util \leftarrow 0$                            ▷ best_util: the utility with the best mapping
11:        for $b\_lo$ = 7 to 16 do                             ▷ Test all possible address mappings
12:            $util \leftarrow$ COMP_UTIL($N - i + 1$, $LDesc_i$, $CTA\_PART_{b\_hi}$, $b\_lo$)    ▷ Calculate overall best mapping
13:            if $util > best\_util$ then
14:                $TMAP_i \leftarrow b\_lo$; $best\_util \leftarrow util$    ▷ update the best mapping
15:            end if
16:        end for
17:        $util_{b\_hi} \leftarrow util_{b\_hi} + best\_util$    ▷ update the new best utility
18:    end for
19:    if $util_{b\_hi} > best\_util\_all$ then
20:        $MAP \leftarrow TMAP$; $MAP_1 \leftarrow b\_hi$;
21:        $best\_util\_all \leftarrow util_{b\_hi}$; $CTA\_NPART \leftarrow CTA\_PART_{b\_hi}$
22:    end if
23: end for
```

Since we want a CTA partitioning scheme that is effective for *multiple* data structures, we also evaluate how *other data structures* can be mapped, based on each CTA partitioning scheme tested for the high-priority data structure (line 8). Based on which of the tested mappings has the highest overall utility, we finally pick the CTA partitioning scheme and an address mapping scheme for each data structure (line 12).

The GPU driver runs Algorithm 2 when all the dynamic information is available at run time (i.e., number of NUMA zones, CTA size, data structure size, etc.). The overhead is negligible because: *(i)* most GPU kernels have only several data structures (i.e., small N), and *(ii)* the two core functions (`NUMA_PART` and `COMP_UTIL`) are very simple due to the 1:1 C-Tile/D-Tile mapping.

The Locality Descriptor method is more flexible and versatile than a first-touch page migration scheme [25], which *(i)* requires demand paging to be enabled, *(ii)* is limited to a *fixed* page size, *(iii)* always schedules CTA in a fixed manner. With the knowledge of how CTAs access data (i.e., the D-Tile-C-Tile compute mapping) and the ability to control and coordinate both the CTA scheduler and flexibly place data, our approach provides a powerful substrate to leverage NUMA locality.

3.5 Methodology

We model the entire Locality Descriptor framework in GPGPU-Sim 3.2.2 [32]. To isolate the effects of the cache locality versus NUMA locality, we evaluate them separately: we evaluate reuse-based locality using an existing single-chip non-NUMA system configuration (based on Fermi GTX 480); and we use a futuristic NUMA system (based on [25]) to evaluate NUMA-based locality. We use the system parameters in [25], but with all compute and bandwidth parameters (number of SMs, memory bandwidth, inter-chip interconnect bandwidth, L2 cache) scaled by 4 to ensure that the evaluated workloads have sufficient parallelism to saturate the compute units. Table 3 summarizes the major system parameters. We use GPUWattch [192] to model GPU power consumption.

Table 3: Major parameters of the simulated systems.

Shader Core	1.4 GHz; GTO scheduler [274]; 2 schedulers per SM Round-robin CTA scheduler
SM Resources	Registers: 32768; Scratchpad: 48KB, L1: 32KB, 4 ways
Memory Model	FR-FCFS scheduling [273, 388], 16 banks/channel
Single Chip System	15 SMs; 6 memory channels; L2: 768KB, 16 ways

Multi-Chip System	4 GPMs (GPU Modules) or NUMA zones;
	64 SMs (16 per module); 32 memory channels;
	L2: 4MB, 16 ways; Inter-GPM Interconnect: 192 GB/s;
	DRAM Bandwidth: 768 GB/s (192 GB/s per module)

We evaluate workloads from the CUDA SDK [244], Rodinia [61], Parboil [313] and PolybenchGPU [267] benchmark suites. We run each kernel either to completion or up to 1B instructions. Our major performance metric is instruction throughput (IPC). From the workloads in Table 4, we use cache-sensitive workloads (i.e., workloads where increasing the L1 by 4× improves performance more than 10%), to evaluate reuse-based locality. We use memory bandwidth-sensitive workloads (workloads that improve performance by more than 40% with 2× memory bandwidth), to evaluate NUMA locality.

3.6 Evaluation

3.6.1 Reuse-Based (Cache) Locality

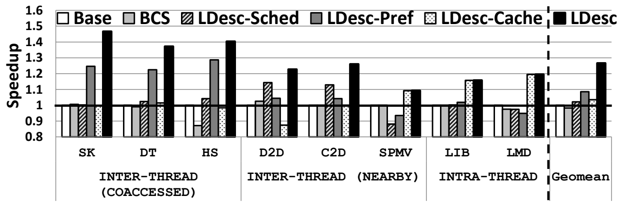
We evaluate six configurations: *(i)* `Baseline`: our baseline system with the default CTA scheduler. *(ii)* `BCS`: a heuristic-based CTA scheduler based on BCS [189], which schedules two consecutive CTAs at the same SM. *(iii)* `LDesc-Sched`: the Locality Descriptor-guided CTA scheduler, which uses the Locality Descriptor semantics and algorithm. Compiler techniques such as [63, 194] can produce the same benefits. *(iv)* `LDesc-Pref`: the Locality Descriptor-guided prefetcher. Sophisticated classification-based prefetchers such as [250], can potentially obtain similar benefits. *(v)* `LDesc-Cache`: the Locality Descriptor-guided cache prioritization and bypassing scheme. *(vi)* `LDesc`: our proposed scheme, which uses the Locality Descriptor to distinguish between the

Table 4: Summary of Applications

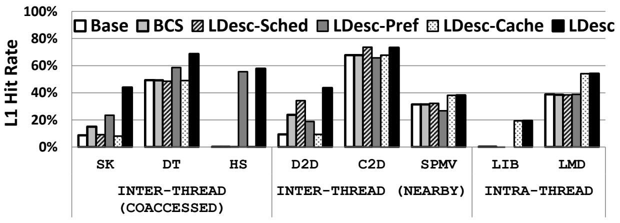
Name (Abbr.)	Locality Descriptor types (§3.3.3)
Syrk (SK) [267]	`INTER-THREAD (COACCESSED, REGULAR), NO-REUSE`
Doitgen (DT) [267]	`INTER-THREAD (COACCESSED, REGULAR), NO-REUSE`
dwt2d (D2D) [61]	`INTER-THREAD (NEARBY, REGULAR)`
Convolution-2D (C2D) [267]	`INTER-THREAD (NEARBY)`
Sparse Matrix Vector Multiply (SPMV) [313]	`INTRA-THREAD,` `INTER-THREAD (COACCESSED, IRREGULAR)`
LIBOR (LIB) [244]	`INTRA-THREAD`
LavaMD (LMD) [61]	`INTRA-THREAD,` `INTER-THREAD (COACCESSED, REGULAR)`
histogram (HS) [313]	`INTER-THREAD (COACCESSED, REGULAR)`
atax (ATX) [267]	`NO-REUSE, INTER-THREAD (COACCESSED, REGULAR)`
mvt (MVT) [267]	`NO-REUSE, INTER-THREAD (COACCESSED, REGULAR)`
particlefilter (PF) [61]	`NO-REUSE`
streamcluster (SC) [61]	`NO-REUSE, INTER-THREAD (NEARBY)`
transpose (TRA) [244]	`NO-REUSE`
Scalar Product (SP) [244]	`NO-REUSE`
Laplace Solver (LPS) [244]	`NO-REUSE, INTRA-THREAD`
pathfinder (PT) [61]	`NO-REUSE`

different locality types and selectively employs different (scheduling, prefetching, caching, and data placement) optimizations.

Figure 20a depicts the speedup over `Baseline` across all configurations. `LDesc` improves performance by 26.6% on average (up to 46.6%) over `Baseline`. `LDesc` always performs either as well as or better than any of the techniques in isolation. Figure 20b shows the L1 hit rate for different configurations. `LDesc`'s performance improvement comes from a 41.1% improvement in average hit rate (up to 57.7%) over `Baseline`. We make three observations that provide insight into `LDesc`'s effectiveness.



(a) Normalized performance with Locality Descriptors.



(b) L1 hit rate with Locality Descriptors.

Figure 20: Leveraging reuse-based (cache) locality.

First, different applications benefit from *different* optimizations. Applications with `INTER-THREAD` type of locality (`SK, DT, HS, D2D, C2D`) benefit from CTA scheduling and/or prefetching. However, LIB, LMD, SPMV do not benefit from CTA scheduling as there is little inter-CTA reuse to be exploited. Similarly, prefetching significantly hurts performance in these workloads (LIB, LMD, SPMV) as the generated prefetch requests exacerbate the memory bandwidth bottleneck. As a result, significant performance degradation occurs when the working set is too large (e.g., when there is no sharing and only INTRA-THREAD reuse, as in LMD and LIB) or where the access patterns in the major data structures are not sufficiently regular (SPMV). Cache prioritization and bypassing is very effective in workloads with INTRA-THREAD reuse (LIB, LMD), but is largely ineffective and can even hurt performance in workloads such as D2D and HS when a non-critical data structure or too many data structures are prioritized in the cache. Since LDesc is able to distinguish between locality types, it is able to select the best combination of optimizations for each application.

Second, a single optimization is very often *insufficient* to exploit locality. For the INTER-THREAD applications (SK, DT, HS, D2D, C2D), LDesc-guided CTA scheduling significantly reduces the L1 working set (by 67.8% on average, not graphed). However, this does *not* translate into significant performance improvement when scheduling is applied by itself (only 2.1% on average). This is because of an 17% average increase in L1 inflight hit rate as a result of more threads accessing the same data. These threads wait on the same shared data at the *same time*, and hence cause increased stalls at the core. The benefit of increased locality is thus lost. Prefetching (LDesc-Pref) is an effective technique to alleviate effect. However, prefetching by itself significantly increases the memory traffic and this hinders its ability to improve performance when applied alone. When combined with scheduling, however, prefetching effectively reduces the long memory latency stalls. Synergistically, CTA scheduling reduces the overall memory traffic by minimizing the working set. For the INTER-THREAD NEARBY workloads (C2D, D2D), CTA scheduling co-schedules CTAs with overlapping working sets. This allows more effective prefetching between the CTAs for the critical high-reuse data. In the cases described above, prefetching and CTA scheduling work better synergistically than in isolation, and LDesc is able to effectively combine and make use of multiple techniques depending on the locality type.

Third, LDesc-guided CTA scheduling is significantly more effective than the heuristic-based approach, BCS. This is because LDesc tailors the CTA scheduling policy for each application by clustering CTAs based on the locality characteristics of each data structure. Similarly, the LDesc prefetcher and replacement policies are highly effective, because they leverage program semantics from the Locality Descriptor.

Conclusions. We make the following conclusions: *(i)* The Locality Descriptor is an effective and versatile mechanism to leverage reuse-based locality to improve GPU performance and energy efficiency; *(ii)* Different locality types require different optimizations—a single mechanism or set of mechanisms do not work for all locality types. We demonstrate that the Locality Descriptor can effectively connect different locality types with the underlying architectural optimizations. *(iii)* The Locality Descriptor enables the hardware architecture to leverage the program's locality semantics to provide significant performance benefits over heuristic-based approaches such as the BCS scheduler.

3.6.2 NUMA Locality

To evaluate the benefits of the Locality Descriptor in exploiting NUMA locality, Figure 21 compares four different mechanisms: *(i)* `Baseline`: The baseline system which uses a static XOR-based address hashing mechanism [381] to randomize data placement across NUMA zones. *(ii)* `FirstTouch-Distrib`: The state-of-the-art mechanism proposed in [25], where each page (64KB) is placed at the NUMA zone where it is first accessed. This scheme also employs a heuristic-based distributed scheduling strategy where the compute grid is partitioned equally across the NUMA zones such that *contiguous* CTAs are placed in the same NUMA zone. *(iii)* `LDesc-Placement`: The memory placement mechanism based on the semantics of the Locality Descriptors, but *without* the accompanying CTA scheduling strategy. *(iv)* `LDesc`: The Locality Descriptor-guided memory placement mechanism with the coordinated CTA scheduling strategy. We draw two conclusions from the figure.

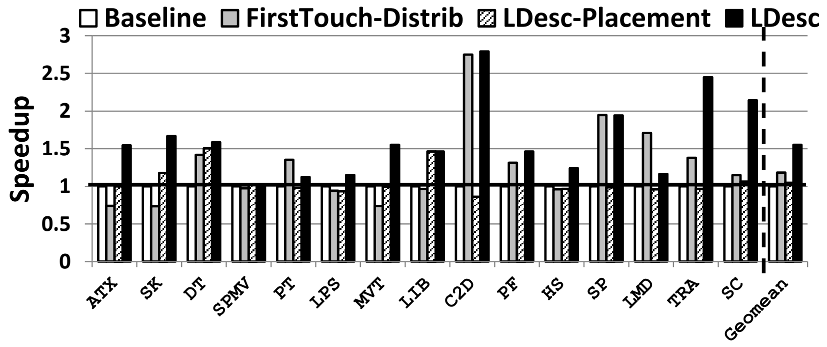


Figure 21: NUMA Locality: Normalized performance.

Conclusion 1. `LDesc` is an effective mechanism in NUMA data placement, outperforming `Baseline` by 53.7% on average (up to 2.8×) and `FirstTouch_Distrib` by 31.2% on average (up to 2.3×). The performance impact of NUMA placement is primarily determined by two factors: *(i) Access efficiency* (plotted in Figure 22), which is defined as the fraction of total memory accesses that are to the local NUMA zone (higher is better). Access efficiency determines the amount of traffic across the interconnect between NUMA zones as well as the latency of memory accesses. *(ii) Access distribution* (plotted in Figure 23) across NUMA zones. Access distribution determines the effective memory bandwidth being utilized by the system—a non-uniform distribution of accesses across NUMA zones may lead to underutilized bandwidth in one or more zones, which can create a new performance bottleneck and degrade performance.

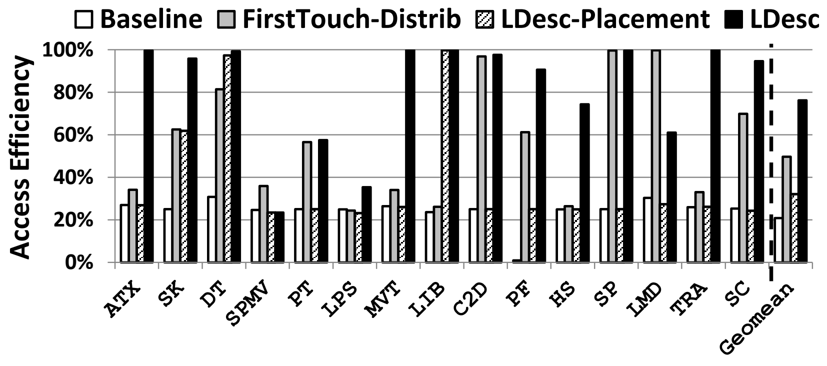


Figure 22: NUMA access efficiency.

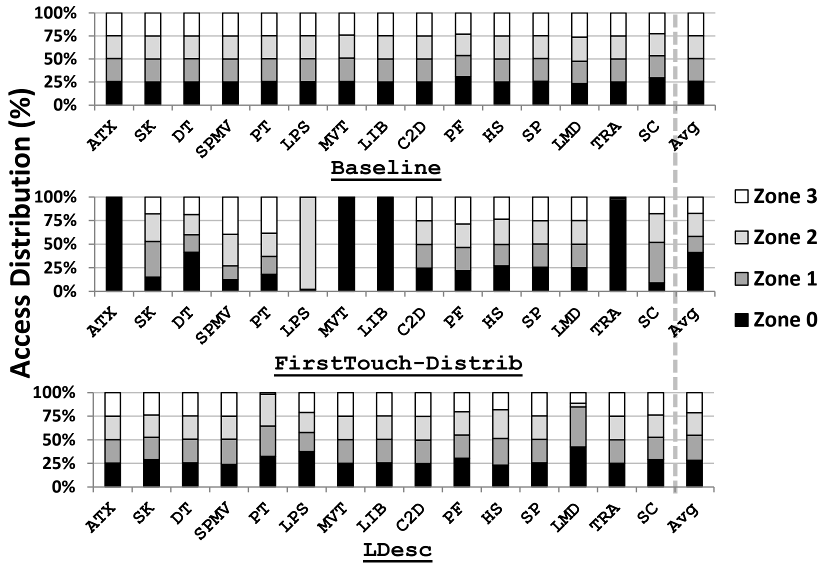


Figure 23: NUMA zone access distribution.

The static randomized mapping in `Baseline` aims to balance access distribution across NUMA zones (with an average distribution of ~25% at each zone), but is *not* optimized to maximize access efficiency (only 22% on average). `FirstTouch-Distrib` on the other hand, has higher access efficiency in some workloads (e.g., `SP, PT`) by ensuring that a page is placed where the CTA that accesses it first is scheduled (49.7% on average). However, `FirstTouch-Distrib` is still ineffective for many workloads for three reasons: *(i)* Large page granularity (64KB) often leads to high skews in access distribution when pages are shared between many CTAs, e.g., `ATX, MVT, LIB` (Figure 23). This is because a majority of pages are placed in the NUMA zone where the CTA that is furthest ahead in execution is scheduled. *(ii)* `FirstTouch-Distrib` has low access efficiency when the heuristic-based scheduler does *not* schedule CTAs that access the same pages at the same NUMA zone (e.g., `DT, HS, SK`). *(iii)* `FirstTouch-Distrib` has low access efficiency when each CTA irregularly accesses a large number of pages because data *cannot* be partitioned between the NUMA zones at a fine granularity (e.g., `SPMV`).

`LDesc` interleaves data at a fine granularity depending on how each data structure is partitioned between CTAs and schedules those CTAs accordingly. If a data structure is shared among more CTAs than what can be scheduled at a single zone, the data structure is partitioned across NUMA zones, as `LDesc` favors parallelism over locality. Hence, `LDesc` tries to improve access efficiency while reducing skew in access distribution in the presence of a large amount of data sharing. As a result, `LDesc` has an average access efficiency of 76% and access distribution close to 25% across the NUMA zones (Figure 23). `LDesc` is less effective in access efficiency in cases where the data structures are irregularly accessed (`SPMV`) or when non-power-of-two data tile sizes lead to imperfect data partitioning (`LPS, PT, LMD, HS`).

Conclusion 2. From Figure 21, we see that `LDesc` is *largely ineffective* without coordinated CTA scheduling. `LDesc-Placement` retains the `LDesc` benefit in reducing the skew in access distribution (not graphed). However, *without* coordinated CTA scheduling access efficiency is very low (32% on average).

We conclude that the Locality Descriptor approach is an effective strategy for data placement in a NUMA environment by *(i)* leveraging locality semantics in intelligent data placement *and* CTA scheduling and *(ii)* orchestrating the two techniques using a single interface.

3.7 Related Work

To our knowledge, this is the first work to propose a cross-layer abstraction that enables the software/programmer to flexibly express and exploit different forms of data locality in GPUs. This enables leveraging program semantics to transparently coordinate architectural techniques that are critical to improving performance and energy efficiency. We now briefly discuss closely related prior work specific to the Locality Descriptor. A more general comparison of the approach taken in this work is discussed in Section 1.5.

Improving Cache Locality in GPUs. There is a large body of research that aims to improve cache locality in GPUs using a range of hardware/software techniques such as CTA scheduling [63, 115,154,163,183,189,194,345,354], prefetching [144,184,188,205,250,298,337], warp scheduling [190, 274,379], cache bypassing [191,195–197,199,229,325,357,358,383], and other cache management schemes [26,67,70,106,149,154,155,166,167,175,177,180,181,231,250,344,380,384]. Some of these works orchestrate multiple techniques [144,154,155,175,181,194,250,384] to leverage synergy between optimizations. However, these prior approaches are either hardware-only, software-only, or focus on optimizing a single technique. Hence, they are limited *(i)* by what is possible with the information that can be solely *inferred* in hardware, *(ii)* by existing software interfaces that limit what optimizations are possible, or *(iii)* in terms of the range of optimizations that can be used. In contrast, the Locality Descriptor provides a new, portable and flexible interface to the software/programmer. This interface allows easy access to hardware techniques in order to leverage data locality. Furthermore, all the above prior approaches are largely orthogonal to the

Locality Descriptor as they can use the Locality Descriptor to enhance their efficacy with the knowledge of program semantics.

The closest work to ours is ACPM [181], an architectural cache management technique that identifies intra-warp/inter-warp/streaming locality and selectively applies cache pinning or bypassing based on the detected locality type. This work is limited to the locality types that can be inferred by hardware, and it does *not* tackle inter-CTA locality or NUMA locality, both of which require a priori knowledge of program semantics and hardware-software codesign.

Improving Locality in NUMA GPU Systems. A range of hardware/software techniques to enhance NUMA locality have been proposed in different contexts in GPUs: multiple GPU modules [25], multiple memory stacks [131], and multi-GPU systems with unified virtual addressing [53, 54, 171, 174, 187, 281]. We already qualitatively and quantitatively compared against `FirstTouch-Distrib` [25] in our evaluation. Our memory placement technique is similar to the approach taken in TOM [131]. In TOM, frequent power-of-two strides seen in GPU kernels are leveraged to use consecutive bits in the address to index a memory stack. TOM, however, *(i)* is the state-of-the-art technique targeted at near-data processing and does *not* require coordination with CTA scheduling, *(ii)* relies on a profiling run to identify the index bits, and *(iii)* does *not* allow using different index bits for different data structures. Techniques to improve locality in multi-GPU systems [53, 54, 171, 174, 187, 281] use profiling and compiler analysis to partition the compute grid and data across multiple GPUs. These works are similar to the Locality Descriptor in terms of the partitioning used for forming data and compute tiles and, hence, can easily leverage Locality Descriptors to further exploit reuse-based locality and NUMA locality in a single GPU.

Expressive Programming Models/Runtime Systems/Interfaces. In the context of multi-core CPUs and distributed/heterogeneous systems, there have been numerous software-only approaches that allow explicit expression of data locality [35, 36, 44, 56, 60, 101, 303, 326, 362], data independence [35, 36, 303, 326] or even tiles [122, 331], to enable the runtime to perform NUMA-aware placement or produce code that is optimized to better exploit the cache hierarchy. These approaches *(i)* are software-only; hence, they do not have access to many architectural techniques that are key to exploiting locality and *(ii)* do not tackle the GPU-specific challenges in exploiting data locality. These works are largely orthogonal to ours and can use Locality Descriptors to leverage hardware techniques to exploit reuse-based locality and NUMA locality in GPUs.

3.8 Summary

This chapter demonstrates the benefits of an *explicit abstraction* for data locality in GPUs that is recognized by all layers of the compute stack, from the programming model to the hardware architecture. We introduce the Locality Descriptor, a rich cross-layer abstraction to explicitly express and effectively leverage data locality in GPUs. The Locality Descriptor *(i)* provides the software/programmer a flexible and portable interface to optimize for data locality without any knowledge of the underlying architecture and *(ii)* enables the architecture to leverage program semantics to optimize and coordinate multiple hardware techniques in a manner that is transparent to the programmer. The key idea is to design the abstraction around the program's data structures and specify locality semantics based on how the program accesses each data structure. We evaluate and demonstrate the performance benefits of Locality Descriptors from effectively leveraging different types of reuse-based locality in the cache hierarchy and NUMA locality in a NUMA memory system. We conclude that by providing a flexible and powerful cross-cutting interface, the Locality Descriptor enables leveraging a critical yet challenging factor in harnessing a GPU's computational power, data locality.

Chapter 4

Zorua

Modern GPU programming models directly manage several on-chip hardware resources in GPU that are critical to performance, e.g., registers, scratchpad memory, and thread slots. This chapter describes how this *tight coupling* between the programming model and hardware resources causes significant challenges in programmability and performance portability, and heavily constrains the hardware's ability to aid in resource management. We propose a new framework, Zorua, that *decouples* the programming model from the management of hardware resources by effectively virtualizing these resources. We demonstrate how this virtualization significantly addresses the programmability, portability, and efficiency challenges in managing these on-chip resources by enabling the hardware to assist in their allocation and resource management.

4.1 Overview

Modern Graphics Processing Units (GPUs) have evolved into powerful programmable machines over the last decade, offering high performance and energy efficiency for many classes of applications by concurrently executing thousands of threads. In order to execute, each thread requires several major on-chip resources: *(i)* registers, *(ii)* scratchpad memory (if used in the program), and *(iii)* a thread slot in the thread scheduler that keeps all the bookkeeping information required for execution.

Today, these hardware resources are *statically* allocated to threads based on several parameters—the number of threads per thread block, register usage per thread, and scratchpad usage per block. We refer to these static application parameters as the *resource specification* of the application. This resource specification forms a critical component of modern GPU programming models (e.g., CUDA [247], OpenCL [7]). The static allocation over a fixed set of hardware resources based on the software-specified resource specification creates a *tight coupling* between the program (and the programming model) and the physical hardware resources. As a result of this tight coupling, for each application, there are only a few optimized resource specifications that maximize resource utilization. Picking a suboptimal specification leads to underutilization of resources and hence, very often, performance degradation. This leads to three key difficulties related to obtaining good performance on modern GPUs: programming ease, portability, and resource inefficiency (performance).

Programming Ease. First, the burden falls upon the programmer to optimize the resource specification. For a naive programmer, this is a very challenging task [82, 218, 251, 277–279, 314]. This is because, in addition to selecting a specification suited to an algorithm, the programmer needs to be aware of the details of the GPU architecture to fit the specification to the underlying hardware resources. This *tuning* is easy to get wrong because there are *many* highly suboptimal performance points in the specification space, and even a minor deviation from an optimized specification can lead to a drastic drop in performance due to lost parallelism. We refer to such drops as *performance cliffs*. We analyze the effect of suboptimal specifications on real systems for 20 workloads (Section 4.2.1), and experimentally demonstrate that changing resource specifications can produce as much as a 5× difference in performance due to the change in parallelism. Even a minimal change in the specification (and hence, the resulting allocation) of one resource can result in a significant performance cliff, degrading performance by as much as 50% (Section 4.2.1).

Portability. Second, different GPUs have varying quantities of each of the resources. Hence, an optimized specification on one GPU may be highly suboptimal on another. In order to determine the extent of this portability problem, we run 20 applications on three generations of NVIDIA GPUs: Fermi, Kepler, and Maxwell (Section 4.2.2). An example result demonstrates that highly-tuned code for Maxwell or Kepler loses as much as 69% of its performance on Fermi. This lack of *portability* necessitates that the programmer *re-tune* the resource specification of the application for *every* new GPU generation. This problem is especially significant in virtualized environments, such as cloud or cluster computing, where the same program may run on a wide range of GPU architectures, depending on data center composition and hardware availability.

Performance. Third, for the programmer who chooses to employ software optimization tools (e.g., auto-tuners) or manually tailor the program to fit the hardware, performance is still constrained by the *fixed, static* resource specification. It is well known [108, 109, 145, 337, 338, 367, 370] that the on-chip resource requirements of a GPU application vary throughout execution. Since the program (even after auto-tuning) has to *statically* specify its *worst-case* resource requirements, severe *dynamic underutilization* of several GPU resources [2, 108, 109, 145, 337, 338] ensues, leading to suboptimal performance (Section 4.2.3).

Our Goal. To address these three challenges at the same time, we propose to *decouple* an application's resource specification from the available hardware resources by *virtualizing* all three major resources in a holistic manner. This virtualization provides the illusion of *more* resources to the GPU programmer and software than physically available, and enables the runtime system and the hardware to *dynamically* manage multiple physical resources in a manner that is transparent to the programmer, thereby alleviating dynamic underutilization.

Virtualization is a concept that has been applied to the management of hardware resources in many contexts (e.g., [21, 40, 79, 85, 121, 140, 249, 342]), providing various benefits. We believe that applying the general principle of virtualization to the management of *multiple* on-chip resources in GPUs offers the opportunity to alleviate several important challenges in modern GPU programming, which are described above. However, at the same time, effectively adding a new level of indirection to the management of multiple latency-critical GPU resources introduces several new challenges (see Section 4.3.1). This necessitates the design of a new mechanism to effectively address the new challenges and enable the benefits of virtualization. In this work, we introduce a new framework, *Zorua*,[5] to decouple the programmer-specified resource specification of an application from its physical on-chip hardware resource allocation by effectively virtualizing the multiple on-chip resources in GPUs.

Key Concepts. The virtualization strategy used by Zorua is built upon two key concepts. First, to mitigate performance cliffs when we do not have enough physical resources, we *oversubscribe* resources by a small amount at runtime, by leveraging their dynamic underutilization and maintaining a swap space (in main memory) for the extra resources required. Second, Zorua improves utilization by determining the runtime resource requirements of an application. It then allocates and deallocates resources dynamically, managing them *(i) independently* of each other to maximize their utilization; and *(ii)* in a *coordinated* manner, to enable efficient execution of each thread with all its required resources available.

Challenges in Virtualization. Unfortunately, oversubscription means that latency-critical resources, such as registers and scratchpad, may be swapped to memory at the time of access, resulting in high overheads in performance and energy. This leads to two critical challenges in designing a framework to enable virtualization. The first challenge is to effectively determine the *extent* of virtualization, i.e., by how much each resource appears to be larger than its physical amount, such that we can minimize oversubscription while still reaping its benefits. This is difficult as the resource requirements continually vary during runtime. The second challenge is to minimize accesses to the swap space. This requires *coordination* in the virtualized management of *multiple resources*, so that enough of each resource is available on-chip when needed.

Zorua. In order to address these challenges, Zorua employs a hardware-software codesign that comprises three components: *(i)* ***the compiler*** annotates the program to specify the resource needs of *each phase* of the application; *(ii)* ***a runtime system***, which we refer to as the coordinator, uses the compiler annotations to dynamically manage the virtualization of the different on-chip resources; and *(iii)* ***the hardware*** employs mapping tables to locate a virtual resource in the physically available resources or in the swap space in main memory. The coordinator plays the key role of scheduling threads *only when* the expected gain in thread-level parallelism outweighs the cost of transferring oversubscribed resources from the swap space in memory, and coordinates the oversubscription and allocation of multiple on-chip resources.

[5]Named after a Pokémon [240] with the power of illusion, able to take different shapes to adapt to different circumstances (not unlike our proposed framework).

Key Results. We evaluate Zorua with many resource specifications for eight applications across three GPU architectures (Section 4.5). Our experimental results show that Zorua *(i)* reduces the range in performance for different resource specifications by 50% on average (up to 69%), by alleviating performance cliffs, and hence eases the burden on the programmer to provide optimized resource specifications, *(ii)* improves performance for code with optimized specification by 13% on average (up to 28%), and *(iii)* enhances portability by reducing the maximum porting performance loss by 55% on average (up to 73%) for three different GPU architectures. We conclude that decoupling the resource specification and resource management via virtualization significantly eases programmer burden, by alleviating the need to provide optimized specifications and enhancing portability, while still improving or retaining performance for programs that already have optimized specifications.

Other Uses. We believe that Zorua offers the opportunity to address several other key challenges in GPUs today, for example: (i) By providing an new level of indirection, Zorua provides a natural way to enable dynamic and fine-grained control over resource partitioning among *multiple GPU kernels and applications.* (ii) Zorua can be utilized for *low-latency preemption* of GPU applications, by leveraging the ability to swap in/out resources from/to memory in a transparent manner. (iv) Zorua provides a simple mechanism to provide dynamic resources to support other programming paradigms such as nested parallelism, helper threads, etc. and even system-level tasks. (v) The dynamic resource management scheme in Zorua improves the energy efficiency and scalability of expensive on-chip resources (Section 4.6).

The main **contributions** of this work are:

- This is the first work that takes a holistic approach to decoupling a GPU application's resource specification from its physical on-chip resource allocation via the use of virtualization. We develop a comprehensive virtualization framework that provides *controlled* and *coordinated* virtualization of *multiple* on-chip GPU resources to maximize the efficacy of virtualization.
- We show how to enable efficient oversubscription of multiple GPU resources with dynamic fine-grained allocation of resources and swapping mechanisms into/out of main memory. We provide a hardware-software cooperative framework that *(i)* controls the extent of oversubscription to make an effective tradeoff between higher thread-level parallelism due to virtualization versus the latency and capacity overheads of swap space usage, and *(ii)* coordinates the virtualization for multiple on-chip resources, transparently to the programmer.
- We demonstrate that by providing the illusion of having more resources than physically available, Zorua *(i)* reduces programmer burden, providing competitive performance for even suboptimal resource specifications, by reducing performance variation across different specifications and by alleviating performance cliffs; *(ii)* reduces performance loss when the program with its resource specification tuned for one GPU platform is ported to a different platform; and *(iii)* retains or

enhances performance for highly-tuned code by improving resource utilization, via dynamic management of resources.

4.2 Motivation: Managing On-Chip Resources and Parallelism in GPUs

The amount of parallelism that the GPU can provide for any application depends on the utilization of on-chip resources by threads within the application. As a result, suboptimal usage of these resources may lead to loss in the parallelism that can be achieved during program execution. This loss in parallelism often leads to significant degradation in performance, as GPUs primarily use fine-grained multi-threading [304, 321] to hide the long latencies during execution.

The granularity of synchronization – i.e., the number of threads in a thread block – and the amount of scratchpad memory used per thread block is determined by the programmer while adapting any algorithm or application for execution on a GPU. This choice involves a complex tradeoff between minimizing data movement, by using *larger* scratchpad memory sizes, and reducing the inefficiency of synchronizing a large number of threads, by using *smaller* scratchpad memory and thread block sizes. A similar tradeoff exists when determining the number of registers used by the application. Using *fewer* registers minimizes hardware register usage and enables higher parallelism during execution, whereas using *more* registers avoids expensive accesses to memory. The resulting application parameters – the number of registers, the amount of scratchpad memory, and the number of threads per thread block – dictate the on-chip resource requirement and hence, determine the parallelism that can be obtained for that application on any GPU.

In this section, we study the performance implications of different choices of resource specifications for GPU applications to demonstrate the key issues we aim to alleviate.

4.2.1 Performance Variation and Cliffs

To understand the impact of resource specifications and the resulting utilization of physical resources on GPU performance, we conduct an experiment on a Maxwell GPU system (GTX 745) with 20 GPGPU workloads from the CUDA SDK [244], Rodinia [61], GPGPU-Sim benchmarks [32], Lonestar [52], Parboil [313], and US DoE application suites [339]. We use the NVIDIA profiling tool (NVProf) [244] to determine the execution time of each application kernel. We sweep the three parameters of the specification—number of threads in a thread block, register usage per thread, and scratchpad memory usage per thread block—for each workload, and measure their impact on execution time.

Figure 24 shows a summary of variation in performance (higher is better), normalized to the slowest specification for each application, across all evaluated specification points for each application in a Tukey box plot [219]. The boxes in the box plot represent the range between the first quartile (25%) and the third quartile (75%). The whiskers extending from the boxes represent the maximum and minimum points of the distribution, or 1.5× the length of the box, whichever is smaller. Any points that lie more than 1.5× the box length beyond the box are considered to be outliers [219], and are plotted as individual points. The line in the middle of the box represents the median, while the "X" represents the average.

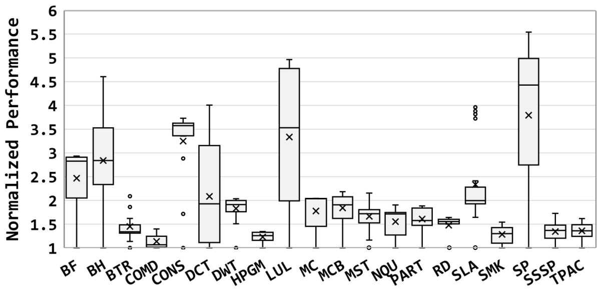
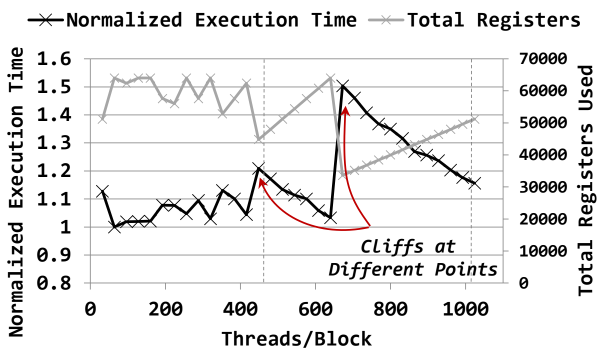


Figure 24: Performance variation across specifications.

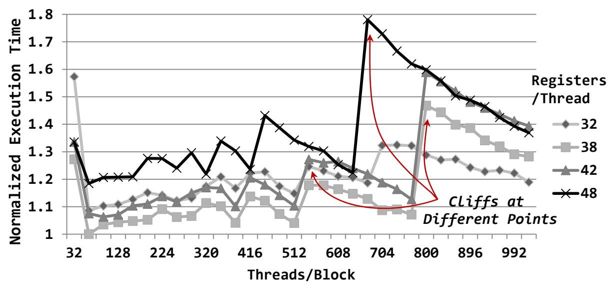
We can see that there is significant variation in performance across different specification points (as much as 5.51× in *SP*), proving the importance of optimized resource specifications. In some applications (e.g., *BTR, SLA*), few points perform well, and these points are significantly better than others, suggesting that it would be challenging for a programmer to locate these high performing specifications and obtain the best performance. Many workloads (e.g., *BH, DCT, MST*) also have higher concentrations of specifications with suboptimal performance in comparison to the best performing point, implying that, without effort, it is likely that the programmer will end up with a resource specification that leads to low performance.

There are several sources for this performance variation. One important source is the loss in thread-level parallelism as a result of a suboptimal resource specification. Suboptimal specifications that are *not* tailored to fit the available physical resources lead to the underutilization of resources. This causes a drop in the number of threads that can be executed concurrently, as there are insufficient resources to support their execution. Hence, better and more balanced utilization of resources enables higher thread-level parallelism. Often, this loss in parallelism from resource underutilization manifests itself in what we refer to as a *performance cliff*, where a small deviation from an optimized specification can lead to significantly worse performance, i.e., there is very high variation in performance between two specification points that are nearby. To demonstrate the existence and analyze the behavior of performance cliffs, we examine two representative workloads more closely.

Figure 25a shows *(i)* how the application execution time changes; and *(ii)* how the corresponding number of registers, statically used, changes when the number of threads per thread block increases from 32 to 1024 threads, for *Minimum Spanning Tree (MST)* [52]. We make two observations.



(a) Threads/block sweep



(b) Threads/block & Registers/thread sweep

Figure 25: Performance cliffs in *Minimum Spanning Tree* (*MST*).

First, let us focus on the execution time between 480 and 1024 threads per block. As we go from 480 to 640 threads per block, execution time gradually decreases. Within this window, the GPU can support two thread blocks running concurrently for *MST*. The execution time falls because the increase in the number of threads per block improves the overall throughput (the number of thread blocks running concurrently remains constant at two, but each thread block does more work in parallel by having more threads per block). However, the corresponding total number of registers used by the blocks also increases. At 640 threads per block, we reach the point where the total number of available registers is not large enough to support two blocks. As a result, the number of blocks executing in parallel drops from two to one, resulting in a significant increase (50%) in execution time, i.e., the *performance cliff*.[6] We see many of these cliffs earlier in the graph as well, albeit not as drastic as the one at 640 threads per block.

Second, Figure 25a shows the existence of performance cliffs when we vary *just one* system parameter—the number of threads per block. To make things more difficult for the programmer, other parameters (i.e., registers per thread or scratchpad memory per thread block) also need to be decided at the same time. Figure 25b demonstrates that performance cliffs also exist when the

[6]Prior work [355] has studied performing resource allocation at the finer warp granularity, as opposed to the coarser granularity of a thread block. As we discuss in Section 1.5 and demonstrate in Section 4.5, this does *not* solve the problem of performance cliffs.

number of registers per thread is varied from 32 to 48.[7] As this figure shows, performance cliffs now occur at *different points* for *different registers/thread curves*, which makes optimizing resource specification, so as to avoid these cliffs, much harder for the programmer.

Barnes-Hut (BH) is another application that exhibits very significant performance cliffs depending on the number of threads per block and registers per thread. Figure 26 plots the variation in performance with the number of threads per block when *BH* is compiled for a range of register sizes (between 24 and 48 registers per thread). We make two observations from the figure. First, similar to *MST*, we observe a significant variation in performance that manifests itself in the form of performance cliffs. Second, we observe that the points at which the performance cliffs occur change greatly depending on the number of registers assigned to each thread during compilation.

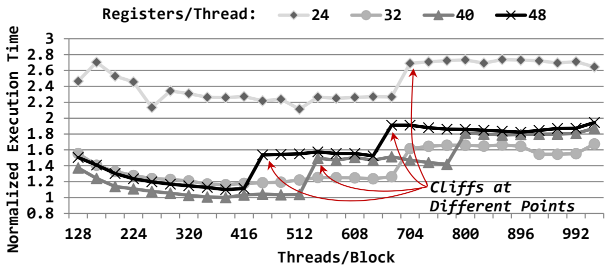


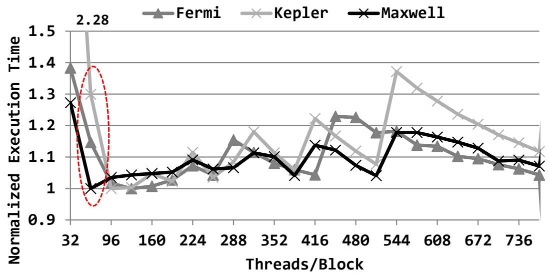
Figure 26: Performance cliffs in *Barnes-Hut* (*BH*).

We conclude that performance cliffs are pervasive across GPU programs, and occur due to fundamental limitations of existing GPU hardware resource managers, where resource management is static, coarse-grained, and tightly coupled to the application resource specification. Avoiding performance cliffs by determining more optimal resource specifications is a challenging task, because the occurrence of these cliffs depends on several factors, including the application characteristics, input data, and the underlying hardware resources.

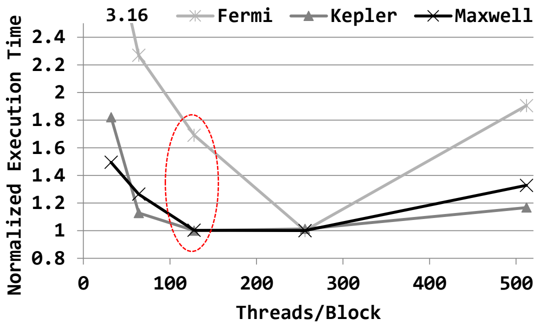
4.2.2 Portability

As we show in Section 4.2.1, tuning GPU applications to achieve good performance on a given GPU is already a challenging task. To make things worse, even after this tuning is done by the programmer for one particular GPU architecture, it has to be *redone* for every new GPU generation (due to changes in the available physical resources across generations) to ensure that good performance is retained. We demonstrate this *portability problem* by running sweeps of the three parameters of the resource specification on various workloads, on three real GPU generations: Fermi (GTX 480), Kepler (GTX 760), and Maxwell (GTX 745).

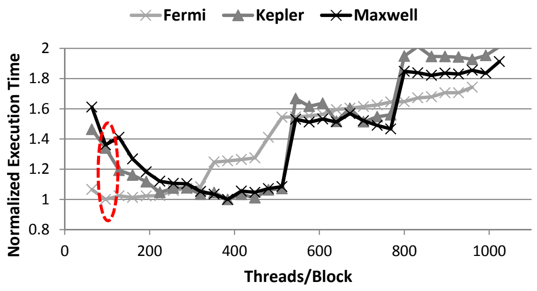
[7] We note that the register usage reported by the compiler may vary from the actual runtime register usage [244], hence slightly altering the points at which cliffs occur.



(a) *MST*



(b) *DCT*



(c) *BH*

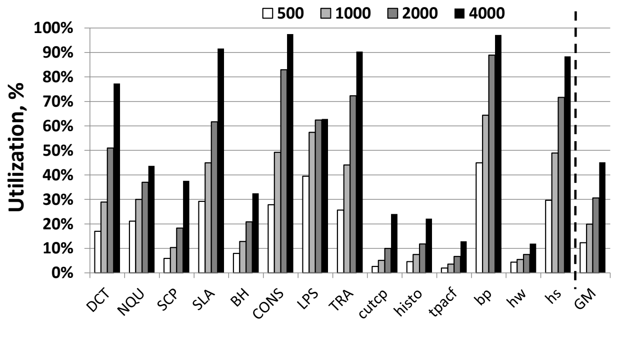
Figure 27: Performance variation across different GPU generations (Fermi, Kepler, and Maxwell) for *MST*, *DCT*, and *BH*.

Figure 27 shows how the optimized performance points change between different GPU generations for two representative applications (*MST* and *DCT*). For every generation, results are normalized to the lowest execution time for that particular generation. As we can see in Figure 27a, the best performing points for different generations occur at *different* specifications because the application behavior changes with the variation in hardware resources. For *MST*, the *Maxwell* architecture performs best at 64 threads per block. However, the same specification point is not efficient for either of the other generations (*Fermi* and *Kepler*), producing 15% and 30% lower performance, respectively, compared to the best specification for each generation. For *DCT* (shown in Figure 27b), both *Kepler* and *Maxwell* perform best at 128 threads per block, but using the same specification for *Fermi* would lead to a 69% performance loss. Similarly, for *BH* (Figure 27c), the optimal point for *Fermi* architecture is at 96 threads per block. However, using the same configuration for the two later GPU architectures – *Kepler* and *Maxwell* could lead to very suboptimal performance results. Using the same configuration results in as much as a 34% performance loss on *Kepler*, and a 36% performance loss on *Maxwell.*

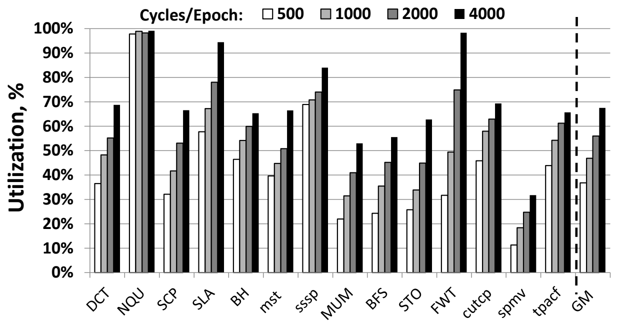
We conclude that the tight coupling between the programming model and the underlying resource management in hardware imposes a significant challenge in performance portability. To avoid suboptimal performance, an application has to be *retuned* by the programmer to find an optimized resource specification for *each* GPU generation.

4.2.3 Dynamic Resource Underutilization

Even when a GPU application is *perfectly* tuned for a particular GPU architecture, the on-chip resources are typically not fully utilized [29, 30, 108–110, 145, 184, 251, 280, 337, 367]. For example, it is well known that while the compiler conservatively allocates registers to hold the *maximum number* of live values throughout the execution, the number of live values at any given time is well below the maximum for large portions of application execution time. To determine the magnitude of this *dynamic underutilization*,[8] we conduct an experiment where we measure the dynamic usage (per epoch) of both scratchpad memory and registers for different applications with *optimized* specifications in our workload pool.



(a) Scratchpad memory



(b) Registers

Figure 28: Dynamic resource utilization for different length epochs.

We vary the length of epochs from 500 to 4000 cycles. Figure 28 shows the results of this experiment for *(i)* scratchpad memory (Figure 28a) and *(ii)* on-chip registers (Figure 28b). We make two major observations from these figures.

First, for relatively small epochs (e.g., 500 cycles), the average utilization of resources is very low (12% for scratchpad memory and 37% for registers). Even for the largest epoch size that we

[8] Underutilization of registers occurs in two major forms—*static*, where registers are unallocated throughout execution [29, 30, 107, 110, 184, 280, 337, 355], and *dynamic*, where utilization of the registers drops during runtime as a result of early completion of warps [355], short register lifetimes [108, 109, 145] and long-latency operations [108, 109]. We do not tackle underutilization from long-latency operations (such as memory accesses) in this work, and leave the exploration of alleviating this type of underutilization to future work.

analyze (4000 cycles), the utilization of scratchpad memory is still less than 50%, and the utilization of registers is less than 70%. This observation clearly suggests that there is an opportunity for a better dynamic allocation of these resources that could allow higher effective GPU parallelism.

Second, there are several noticeable applications, e.g., *cutcp*, *hw*, *tpacf*, where utilization of the scratchpad memory is always lower than 15%. This dramatic underutilization due to static resource allocation can lead to significant loss in potential performance benefits for these applications.

In summary, we conclude that existing static on-chip resource allocation in GPUs can lead to significant resource underutilization that can lead to suboptimal performance and energy waste.

4.2.4 Our Goal

As we see above, the tight coupling between the resource specification and hardware resource allocation, and the resulting heavy dependence of performance on the resource specification, creates a number of challenges. In this work, our goal is to alleviate these challenges by providing a mechanism that can *(i)* ease the burden on the programmer by ensuring reasonable performance, *regardless of the resource specification*, by successfully avoiding performance cliffs, while retaining performance for code with optimized specification; *(ii)* enhance portability by minimizing the variation in performance for optimized specifications across different GPU generations; and *(iii)* maximize dynamic resource utilization even in highly optimized code to further improve performance. We make two key observations from our studies above to help us achieve this goal.

Observation 1: *Bottleneck Resources.* We find that performance cliffs occur when the amount of any resource required by an application exceeds the physically available amount of that resource. This resource becomes a *bottleneck*, and limits the amount of parallelism that the GPU can support. If it were possible to provide the application with a *small additional amount* of the bottleneck resource, the application can see a significant increase in parallelism and thus avoid the performance cliff.

Observation 2: *Underutilized Resources.* As discussed in Section 4.2.3, there is significant underutilization of resources at runtime. These underutilized resources could be employed to support more parallelism at runtime, and thereby alleviate the aforementioned challenges.

We use these two observations to drive our resource virtualization solution, which we describe next.

4.3 Our Approach: Decoupling the Programming Model from Resource Management

In this work, we design Zorua, a framework that provides the illusion of more GPU resources than physically available by decoupling the resource specification from its allocation in the hardware resources. We introduce a new level of indirection by virtualizing the on-chip resources to allow the hardware to manage resources transparently to the programmer.

The virtualization provided by Zorua builds upon two *key concepts* to leverage the aforementioned observations. First, when there are insufficient physical resources, we aim to provide the illusion of the required amount by *oversubscribing* the required resource. We perform this oversubscription by leveraging the dynamic underutilization as much as possible, or by spilling to a swap space in memory. This oversubscription essentially enables the illusion of more resources than what is available (physically and statically), and supports the concurrent execution of more threads. Performance cliffs are mitigated by providing enough additional resources to avoid drastic drops in parallelism. Second, to enable efficient oversubscription by leveraging underutilization, we dynamically allocate and deallocate physical resources depending on the requirements of the application during execution. We manage the virtualization of each resource *independently* of other resources to maximize its runtime utilization.

Figure 29 depicts the high-level overview of the virtualization provided by Zorua. The *virtual space* refers to the *illusion* of the quantity of available resources. The *physical space* refers to the *actual* hardware resources (specific to the GPU architecture), and the *swap space* refers to the resources that do not fit in the physical space and hence are *spilled* to other physical locations. For the register file and scratchpad memory, the swap space is mapped to global memory space in the memory hierarchy. For threads, only those that are mapped to the physical space are available for scheduling and execution at any given time. If a thread is mapped to the swap space, its state (i.e., the PC and the SIMT stack) is saved in memory. Resources in the virtual space can be freely re-mapped between the physical and swap spaces to maintain the illusion of the virtual space resources.

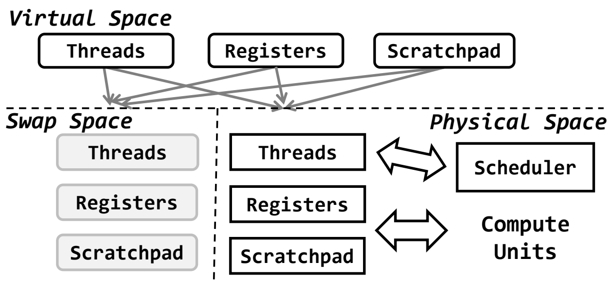


Figure 29: High-level overview of Zorua.

In the baseline architecture, the thread-level parallelism that can be supported, and hence the throughput obtained from the GPU, depends on the quantity of *physical resources*. With the virtualization enabled by Zorua, the parallelism that can be supported now depends on the quantity of *virtual resources* (and how their mapping into the physical and swap spaces is managed). Hence, the size of the virtual space for each resource plays the key role of determining the parallelism that can be exploited. Increasing the virtual space size enables higher parallelism, but leads to higher swap space usage. It is critical to minimize accesses to the swap space to avoid the latency overhead and capacity/bandwidth contention associated with accessing the memory hierarchy.

In light of this, there are two key challenges that need to be addressed to effectively virtualize on-chip resources in GPUs. We now discuss these challenges and provide an overview of how we address them.

4.3.1 Challenges in Virtualization

Challenge 1: *Controlling the Extent of Oversubscription.* A key challenge is to determine the *extent* of oversubscription, or the size of the virtual space for each resource. As discussed above, increasing the size of the virtual space enables more parallelism. Unfortunately, it could also result in more spilling of resources to the swap space. Finding the tradeoff between more parallelism and less overhead is challenging, because the dynamic resource requirements of each thread tend to significantly fluctuate throughout execution. As a result, the size of the virtual space for each resource needs to be *continuously* tuned to allow the virtualization to adapt to the runtime requirements of the program.

Challenge 2: *Control and Coordination of Multiple Resources.* Another critical challenge is to efficiently map the continuously varying virtual resource space to the physical and swap spaces. This is important for two reasons. First, it is critical to minimize accesses to the swap space. Accessing the swap space for the register file or scratchpad involves expensive accesses to global memory, due to the added latency and contention. Also, only those threads that are mapped to the physical space are available to the warp scheduler for selection. Second, each thread requires multiple resources for execution. It is critical to *coordinate* the allocation and mapping of these different resources to ensure that an executing thread has *all* the required resources allocated to it, while minimizing accesses to the swap space. Thus, an effective virtualization framework must coordinate the allocation of *multiple* on-chip resources.

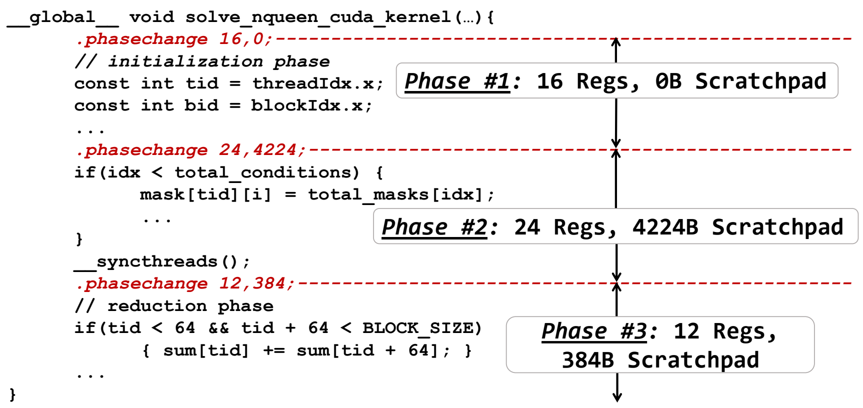
4.3.2 Key Ideas of Our Design

To solve these challenges, Zorua employs two key ideas. First, we leverage the software (the compiler) to provide annotations with information regarding the resource requirements of each *phase* of the application. This information enables the framework to make intelligent dynamic decisions, with respect to both the size of the virtual space and the allocation/deallocation of resources (Section 4.3.2).

Second, we use an adaptive runtime system to control the allocation of resources in the virtual space and their mapping to the physical/swap spaces. This allows us to *(i)* dynamically alter the size of the virtual space to change the extent of oversubscription; and *(ii)* continuously coordinate the allocation of multiple on-chip resources and the mapping between their virtual and physical/swap spaces, depending on the varying runtime requirements of each thread (Section 4.3.2).

Leveraging Software Annotations of Phase Characteristics. We observe that the runtime variation in resource requirements (Section 4.2.3) typically occurs at the granularity of *phases* of a few tens of instructions. This variation occurs because different parts of kernels perform different operations that require different resources. For example, loops that primarily load/store data from/to scratchpad memory tend to be less register heavy. Sections of code that perform specific computations (e.g., matrix transformation, graph manipulation), can either be register heavy or primarily operate out of scratchpad. Often, scratchpad memory is used for only short intervals [367], e.g., when data exchange between threads is required, such as for a reduction operation.

Figure 30 depicts a few example phases from the *NQU* (*N-Queens Solver*) [266] kernel. *NQU* is a scratchpad-heavy application, but it does not use the scratchpad at all during the initial computation phase. During its second phase, it performs its primary computation out of the scratchpad, using as much as 4224B. During its last phase, the scratchpad is used only for reducing results, which requires only 384B. There is also significant variation in the maximum number of live registers in the different phases.



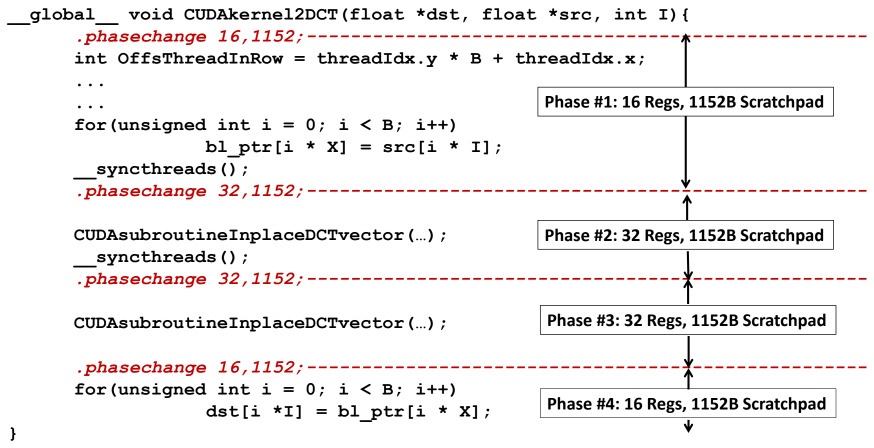
```
__global__ void solve_nqueen_cuda_kernel(…){
      .phasechange 16,0;
      // initialization phase
      const int tid = threadIdx.x;
      const int bid = blockIdx.x;
      ...
      .phasechange 24,4224;
      if(idx < total_conditions) {
            mask[tid][i] = total_masks[idx];
            ...
      }
      __syncthreads();
      .phasechange 12,384;
      // reduction phase
      if(tid < 64 && tid + 64 < BLOCK_SIZE)
            { sum[tid] += sum[tid + 64]; }
      ...
}
```

Phase #1: 16 Regs, 0B Scratchpad
Phase #2: 24 Regs, 4224B Scratchpad
Phase #3: 12 Regs, 384B Scratchpad

Figure 30: Example phases from *NQU*.

Another example of phase variation from the *DCT* (*Discrete Fourier Transform*) kernel is depicted in Figure 31. *DCT* is both register and scratchpad-intensive. The scratchpad memory usage does not vary in this kernel. However, the register usage significantly varies – the register usage increases by 2X in the second and third phase in comparison with the first and fourth phase.

In order to capture both the resource requirements as well as their variation over time, we partition the program into a number of *phases*. A phase is a sequence of instructions with sufficiently different resource requirements than adjacent phases. Barrier or fence operations also indicate a change in requirements for a different reason—threads that are waiting at a barrier do



```
__global__ void CUDAkernel2DCT(float *dst, float *src, int I){
      .phasechange 16,1152;
      int OffsThreadInRow = threadIdx.y * B + threadIdx.x;
      ...
      ...
      for(unsigned int i = 0; i < B; i++)
                  bl_ptr[i * X] = src[i * I];
      __syncthreads();
      .phasechange 32,1152;

      CUDAsubroutineInplaceDCTvector(…);
      __syncthreads();
      .phasechange 32,1152;

      CUDAsubroutineInplaceDCTvector(…);

      .phasechange 16,1152;
      for(unsigned int i = 0; i < B; i++)
                  dst[i *I] = bl_ptr[i * X];
}
```

Phase #1: 16 Regs, 1152B Scratchpad
Phase #2: 32 Regs, 1152B Scratchpad
Phase #3: 32 Regs, 1152B Scratchpad
Phase #4: 16 Regs, 1152B Scratchpad

Figure 31: Example phases from *DCT*

not immediately require the thread slot that they are holding. We interpret barriers and fences as phase boundaries since they potentially alter the utilization of their thread slots. The compiler inserts special instructions called *phase specifiers* to mark the start of a new phase. Each phase specifier contains information regarding the resource requirements of the next phase. Section 4.4.7 provides more detail on the semantics of phases and phase specifiers.

A phase forms the basic unit for resource allocation and de-allocation, as well as for making oversubscription decisions. It offers a finer granularity than an *entire thread* to make such decisions. The phase specifiers provide information on the *future resource usage* of the thread at a phase boundary. This enables *(i)* preemptively controlling the extent of oversubscription at runtime, and *(ii)* dynamically allocating and deallocating resources at phase boundaries to maximize utilization of the physical resources.

Control with an Adaptive Runtime System. Phase specifiers provide information to make oversubscription and allocation/deallocation decisions. However, we still need a way to make decisions on the extent of oversubscription and appropriately allocate resources at runtime. To this end, we use an adaptive runtime system, which we refer to as the *coordinator*. Figure 32 presents an overview of the coordinator.

The virtual space enables the illusion of a larger amount of each of the resources than what is physically available, to adapt to different application requirements. This illusion enables higher thread-level parallelism than what can be achieved with solely the fixed, physically available resources, by allowing more threads to execute concurrently. The size of the virtual space at a given time determines this parallelism, and those threads that are effectively executed in parallel are referred to as *active threads*. All active threads have thread slots allocated to them in the virtual space (and hence can be executed), but some of them may not be mapped to the physical space at a given time. As discussed previously, the resource requirements of each application continuously change during execution. To adapt to these runtime changes, the coordinator leverages information

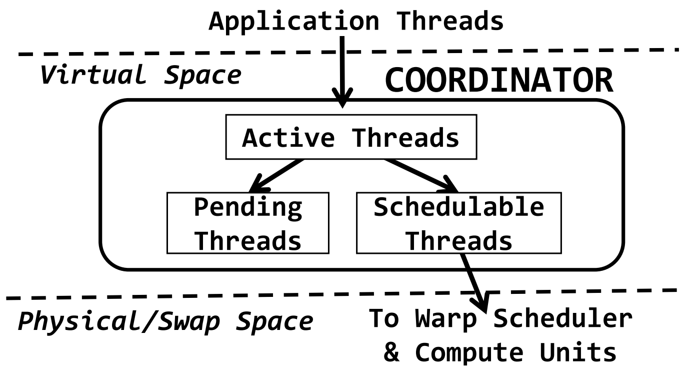


Figure 32: Overview of the coordinator.

from the phase specifiers to make decisions on oversubscription. The coordinator makes these decisions at every phase boundary and thereby controls the size of the virtual space for each resource (see Section 4.4.2).

To enforce the determined extent of oversubscription, the coordinator allocates all the required resources (in the virtual space) for only a *subset* of threads from the active threads. Only these dynamically selected threads, referred to as *schedulable threads*, are available to the warp scheduler and compute units for execution. The coordinator, hence, dynamically partitions the active threads into *schedulable threads* and the *pending threads*. Each thread is swapped between *schedulable* and *pending* states, depending on the availability of resources in the virtual space. Selecting only a subset of threads to execute at any time ensures that the determined size of the virtual space is not exceeded for any resource, and helps coordinate the allocation and mapping of multiple on-chip resources to minimize expensive data transfers between the physical and swap spaces (discussed in Section 4.4).

4.3.3 Overview of Zorua

In summary, to effectively address the challenges in virtualization by leveraging the above ideas in design, Zorua employs a software-hardware codesign that comprises three components: *(i)* ***The compiler*** annotates the program by adding special instructions (*phase specifiers*) to partition it into *phases* and to specify the resource needs of each phase of the application. *(ii)* ***The coordinator***, a hardware-based adaptive runtime system, uses the compiler annotations to dynamically allocate/deallocate resources for each thread at phase boundaries. The coordinator plays the key role of continuously controlling the extent of the oversubscription (and hence the size of the virtual space) at each phase boundary. *(iii)* ***Hardware virtualization support*** includes a mapping table for each resource to locate each virtual resource in either the physical space or the swap space in main memory, and the machinery to swap resources between the physical and swap spaces.

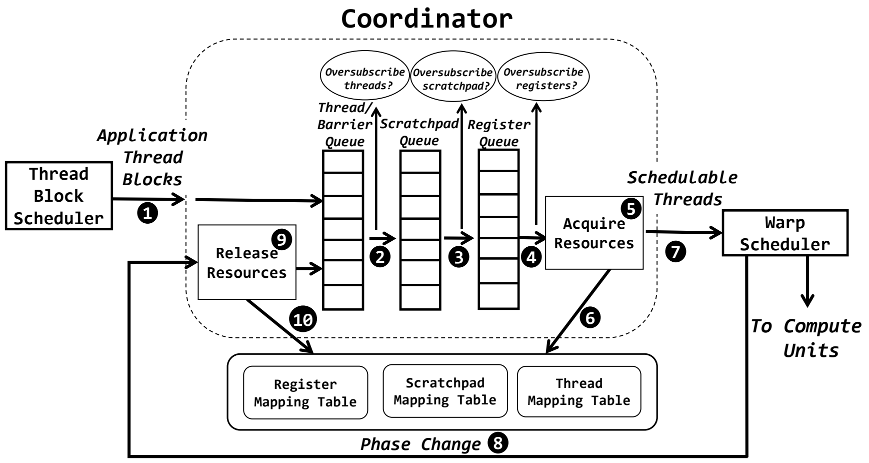


Figure 33: Overview of Zorua in hardware.

4.4 Zorua: Detailed Mechanism

We now detail the operation and implementation of the various components of the Zorua framework.

4.4.1 Key Components in Hardware

Zorua has two key hardware components: *(i)* the *coordinator* that contains queues to buffer the *pending threads* and control logic to make oversubscription and resource management decisions, and *(ii) resource mapping tables* to map each of the resources to their corresponding physical or swap spaces.

Figure 33 presents an overview of the hardware components that are added to each SM. The coordinator interfaces with the thread block scheduler (❶) to schedule new blocks onto an SM. It also interfaces with the warp schedulers by providing a list of *schedulable warps* (❼).[9] The resource mapping tables are accessible by the coordinator and the compute units. We present a detailed walkthrough of the operation of Zorua and then discuss its individual components in more detail.

4.4.2 Detailed Walkthrough

The coordinator is called into action by three events: *(i)* a new thread block is scheduled at the SM for execution, *(ii)* a warp undergoes a phase change, or *(iii)* a warp or a thread block reaches the end of execution. Between these events, the coordinator performs no action and execution proceeds as usual. We now walk through the sequence of actions performed by the coordinator for each type of event.

Thread Block: Execution Start. When a thread block is scheduled onto an SM for execution

[9]We use an additional bit in each warp slots to indicate to the scheduler whether the warp is schedulable.

(❶), the coordinator first buffers it. The primary decision that the coordinator makes is to determine whether or not to make each thread available to the scheduler for execution. The granularity at which the coordinator makes decisions is that of a warp, as threads are scheduled for execution at the granularity of a warp (hence we use *thread slot* and *warp slot* interchangeably). Each warp requires three resources: a thread slot, registers, and potentially scratchpad. The amount of resources required is determined by the phase specifier (Section 4.4.7) at the start of execution, which is placed by the compiler into the code. The coordinator must supply each warp with *all* its required resources in either the physical or swap space before presenting it to the warp scheduler for execution.

To ensure that each warp is furnished with its resources and to coordinate potential oversubscription for each resource, the coordinator has three queues—*thread/barrier, scratchpad, and register queues*. The three queues together essentially house the *pending threads*. Each warp must traverse each queue (❷ ❸ ❹), as described next, before becoming eligible to be scheduled for execution. The coordinator allows a warp to traverse a queue when *(a)* it has enough of the corresponding resource available in the physical space, or *(b)* it has an insufficient resources in the physical space, but has decided to oversubscribe and allocate the resource in the swap space. The total size of the resource allocated in the physical and swap spaces cannot exceed the determined virtual space size. The coordinator determines the availability of resources in the physical space using the mapping tables (see Section 4.4.5). If there is an insufficient amount of a resource in the physical space, the coordinator needs to decide whether or not to increase the virtual space size for that particular resource by oversubscribing and using swap space. We describe the decision algorithm in Section 4.4.4. If the warp cannot traverse *all* queues, it is left waiting in the first (*thread/barrier*) queue until the next coordinator event. Once a warp has traversed *all* the queues, the coordinator acquires all the resources required for the warp's execution (❺). The corresponding mapping tables for each resource is updated (❻) to assign resources to the warp, as described in Section 4.4.5.

Warp: Phase Change. At each phase change (❽), the warp is removed from the list of schedulable warps and is returned to the coordinator to acquire/release its resources. Based on the information in its phase specifier, the coordinator releases the resources that are no longer live and hence are no longer required (❾). The coordinator updates the mapping tables to free these resources (❿). The warp is then placed into a specific queue, depending on which live resources it retained from the previous phase and which new resources it requires. The warp then attempts to traverse the remaining queues (❷ ❸ ❹), as described above. A warp that undergoes a phase change as a result of a barrier instruction is queued in the *thread/barrier queue* (❷) until all warps in the same thread block reach the barrier.

Thread Block/Warp: Execution End. When a warp completes execution, it is returned to the coordinator to release any resources it is holding. Scratchpad is released only when the entire thread block completes execution. When the coordinator has free warp slots for a new thread block, it requests the thread block scheduler (❶) for a new block.

Every Coordinator Event. At any event, the coordinator attempts to find resources for warps waiting at the queues, to enable them to execute. Each warp in each queue (starting from the *register queue*) is checked for the availability of the required resources. If the coordinator is able to allocate resources in the physical or swap space without exceeding the determined size of virtual space, the warp is allowed to traverse the queue.

4.4.3 Benefits of Our Design

Decoupling the Warp Scheduler and Mapping Tables from the Coordinator. Decoupling the warp scheduler from the coordinator enables Zorua to use any scheduling algorithm over the schedulable warps to enhance performance. One case when this is useful is when increasing parallelism degrades performance by increasing cache miss rate or causing memory contention [163, 164, 274]. Our decoupled design allows this challenge to be addressed independently from the coordinator using more intelligent scheduling algorithms [26, 164, 238, 274] and cache management schemes [26, 196, 197, 358]. Furthermore, decoupling the mapping tables from the coordinator allows easy integration of any implementation of the mapping tables that may improve efficiency for each resource.

Coordinating Oversubscription for Multiple Resources. The queues help ensure that a warp is allocated *all* resources in the virtual space before execution. They *(i)* ensure an ordering in resource allocation to avoid deadlocks, and *(ii)* enforce priorities between resources. In our evaluated approach, we use the following order of priorities: threads, scratchpad, and registers. We prioritize scratchpad over registers, as scratchpad is shared by all warps in a block and hence has a higher value by enabling more warps to execute. We prioritize threads over scratchpad, as it is wasteful to allow warps stalled at a barrier to acquire other resources—other warps that are still progressing towards the barrier may be starved of the resource they need. Furthermore, managing each resource independently allows different oversubscription policies for each resource and enables fine-grained control over the size of the virtual space for that resource.

Flexible Oversubscription. Zorua's design can flexibly enable/disable swap space usage, as the dynamic fine-grained management of resources is independent of the swap space. Hence, in cases where the application is well-tuned to utilize the available resources, swap space usage can be disabled or minimized, and Zorua can still improve performance by reducing dynamic underutilization of resources. Furthermore, different oversubscription algorithms can be flexibly employed to manage the size of the virtual space for each resource (independently or cooperatively). These algorithms can be designed for different purposes, e.g., minimizing swap space usage, improving fairness in a multikernel setting, reducing energy, etc. In Section 4.4.4, we describe an example algorithm to improve performance by making a good tradeoff between improving parallelism and reducing swap space usage.

Avoiding Deadlocks. A resource allocation deadlock could happen if resources are distributed among too many threads, such that *no* single thread is able to obtain enough necessary resources for execution. Allocating resources using *multiple* ordered queues helps avoid deadlocks in resource allocation in three ways. First, new resources are allocated to a warp only once the warp has traversed *all* of the queues. This ensures that resources are not wastefully allocated to warps that will be stalled anyway. Second, a warp is allocated resources based on how many resources it already has, i.e. how many queues it has already traversed. Warps that already hold multiple live resources are prioritized in allocating new resources over warps that do *not* hold any resources. Finally, if there are insufficient resources to maintain a minimal level of parallelism (e.g., 20% of SM occupancy in our evaluation), the coordinator handles this rare case by simply oversubscribing resources to ensure that there is no deadlock in allocation.

Managing More Resources. Our design also allows flexibly adding more resources to be managed by the virtualization framework, for example, thread block slots. Virtualizing a new resource with Zorua simply requires adding a new queue to the coordinator and a new mapping table to manage the virtual to physical mapping.

4.4.4 Oversubscription Decisions

Leveraging Phase Specifiers. Zorua leverages the information provided by phase specifiers (Section 4.4.7) to make oversubscription decisions for each phase. For each resource, the coordinator checks whether allocating the requested quantity according to the phase specifier would cause the total swap space to exceed an *oversubscription threshold*, or *o_thresh*. This threshold essentially dynamically sets the size of the virtual space for each resource. The coordinator allows oversubscription for each resource only within its threshold. *o_thresh* is dynamically determined to adapt to the characteristics of the workload, and tp ensure good performance by achieving a good tradeoff between the overhead of oversubscription and the benefits gained from parallelism.

Determining the Oversubscription Threshold. In order to make the above tradeoff, we use two architectural statistics: *(i)* idle time at the cores, *c_idle*, as an indicator for potential performance benefits from parallelism; and *(ii)* memory idle time (the idle cycles when all threads are stalled waiting for data from memory or the memory pipeline), *c_mem*, as an indicator of a saturated memory subsystem that is unlikely to benefit from more parallelism.[10] We use Algorithm 3 to determine *o_thresh* at runtime. Every *epoch*, the change in *c_mem* is compared with

[10]This is similar to the approach taken by prior work [163] to estimate the performance benefits of increasing parallelism.

the change in *c_idle*. If the increase in *c_mem* is greater, this indicates an increase in pressure on the memory subsystem, suggesting both lesser benefit from parallelism and higher overhead from oversubscription. In this case, we reduce *o_thresh*. On the other hand, if the increase in *c_idle* is higher, this is indicative of more idleness in the pipelines, and higher potential performance from parallelism and oversubscription. We increase *o_thresh* in this case, to allow more oversubscription and enable more parallelism. Table 5 describes the variables used in Algorithm 3.

Algorithm 3 Determining the oversubscription threshold

```
1:  o_thresh = o_default                                    ▷ Initialize threshold
2:  for each epoch do
3:      c_idle_delta = (c_idle − c_idle_prev)  ▷ Determine the change in c_idle and c_mem from the previous epoch
4:      c_mem_delta = (c_mem − c_mem_prev)
5:      if (c_idle_delta − c_mem_delta) > c_delta_thresh then   ▷ Indicates more idleness and potential for benefits from parallelism
6:          o_thresh += o_thresh_step
7:      end if
8:      if (c_mem_delta − c_idle_delta) > c_delta_thresh then   ▷ Traffic in memory is likely to outweigh any parallelism benefit
9:          o_thresh −= o_thresh_step
10:     end if
11: end for
```

4.4.5 Virtualizing On-chip Resources

A resource can be in either the physical space, in which case it is mapped to the physical on-chip resource, or the swap space, in which case it can be found in the memory hierarchy. Thus, a resource is effectively virtualized, and we need to track the mapping between the virtual and physical/swap spaces. We use a *mapping table* for each resource to determine *(i)* whether the resource is in the physical or swap space, and *(ii)* the location of the resource within the physical on-chip hardware. The compute units access these mapping tables before accessing the real resources. An access to a resource that is mapped to the swap space is converted to a global memory access that is addressed by the logical resource ID and warp/block ID (and a base register for the swap space of the resource). In addition to the mapping tables, we use two registers per resource to track the amount of the resource that is *(i)* free to be used in physical space, and *(ii)* mapped in swap space. These two counters enable the coordinator to make oversubscription decisions (Section 4.4.4). We now go into more detail on virtualized resources in Zorua.[11]

[11] Our implementation of a virtualized resource aims to minimize complexity. This implementation is largely orthogonal to the framework itself, and one can envision other implementations (e.g., [145, 367, 370]) for different resources.

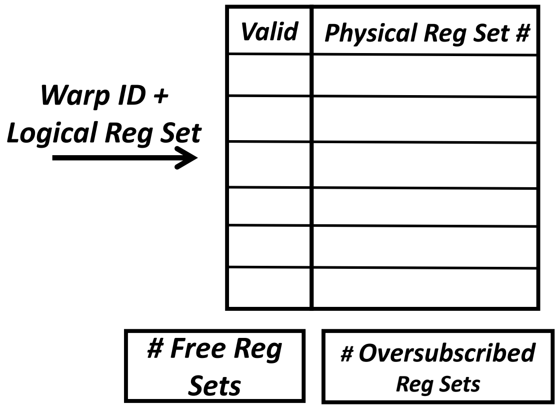
Variable	Description
o_thresh	oversubscription threshold (dynamically determined)
o_default	initial value for *o_thresh* , (experimentally determined to be 10% of total physical resource)
c_idle	core cycles when no threads are issued to the core (but the pipeline is not stalled) [163]
c_mem	core cycles when all warps are waiting for data from memory or stalled at the memory pipeline
**_prev*	the above statistics for the previous epoch
c_delta_thresh	threshold to produce change in *o_thresh* (experimentally determined to be 16)
o_thresh_step	increment/decrement to *o_thresh* , experimentally determined to be 4% of the total physical resource
epoch	interval in core cycles to change *o_thresh* (experimentally determined to be 2048)

Table 5: Variables for oversubscription

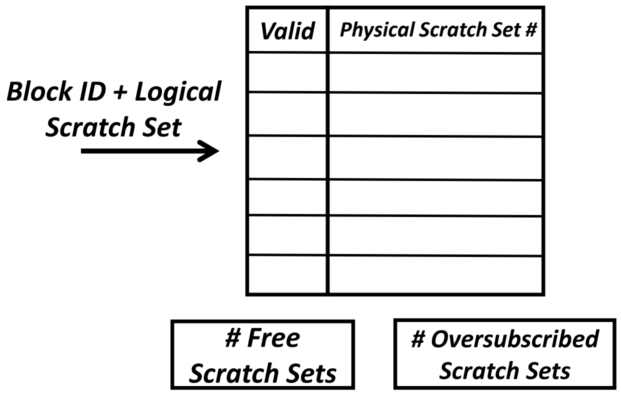
Virtualizing Registers and Scratchpad Memory. In order to minimize the overhead of large mapping tables, we map registers and scratchpad at the granularity of a *set*. The size of a set is configurable by the architect—we use 4**warp_size*[12] for the register mapping table, and 1KB for scratchpad. Figure 34 depicts the tables for the registers and scratchpad. The register mapping table is indexed by the warp ID and the logical register set number (*logical_register_number / register_set_size*). The scratchpad mapping table is indexed by the block ID and the logical scratchpad set number (*logical_scratchpad_address / scratchpad_set_size*). Each entry in the mapping table contains the physical address of the register/scratchpad content in the physical register file or scratchpad. The valid bit indicates whether the logical entry is mapped to the physical space or the swap space. With 64 logical warps and 16 logical thread blocks, the register mapping table takes 1.125 KB ($64 \times 16 \times 9$ bits, or 0.87% of the register file) and the scratchpad mapping table takes 672 B ($16 \times 48 \times 7$ bits, or 1.3% of the scratchpad).

Virtualizing Thread Slots. Each SM is provisioned with a fixed number of *warp slots*, which determine the number of warps that are considered for execution every cycle by the warp scheduler. In order to oversubscribe warp slots, we need to save the state of each warp in memory before remapping the physical slot to another warp. This state includes the bookkeeping required for execution, i.e., the warp's PC (program counter) and the SIMT stack, which holds divergence information for each executing warp. The thread slot mapping table records whether each warp is

[12]We track registers at the granularity of a warp.



(a) Register Mapping Table



(b) Scratchpad Mapping Table

Figure 34: Mapping Tables

mapped to a physical slot or swap space. The table is indexed by the logical warp ID, and stores the address of the physical warp slot that contains the warp. In our baseline design with 64 logical warps, this mapping table takes 56 B (64×7 bits).

4.4.6 Handling Resource Spills

If the coordinator has oversubscribed any resource, it is possible that the resource can be found either *(i)* on-chip (in the physical space) or *(ii)* in the swap space in the memory hierarchy. As described above, the location of any virtual resource is determined by the mapping table for each resource. If the resource is found on-chip, the mapping table provides the physical location in the register file and scratchpad memory. If the resource is in the swap space, the access to that resource is converted to a global memory load that is addressed either by the *(i)* thread block ID and logical register/scratchpad set, in the case of registers or scratchpad memory; or *(ii)* logical warp ID, in the case of warp slots. The oversubscribed resource is typically found in the L1/L2 cache but in the worst case, could be in memory. When the coordinator chooses to oversubscribe any resource beyond what is available on-chip, the least frequently accessed resource set is spilled to the memory hierarchy using a simple store operation.

4.4.7 Supporting Phases and Phase Specifiers

Identifying phases. The compiler partitions each application into phases based on the liveness of registers and scratchpad memory. To avoid changing phases too often, the compiler uses thresholds to determine phase boundaries. In our evaluation, we define a new phase boundary when there is *(i)* a 25% change in the number of live registers or live scratchpad content, and *(ii)* a minimum of 10 instructions since the last phase boundary. To simplify hardware design, the compiler draws phase boundaries only where there is no control divergence.[13]

Once the compiler partitions the application into phases, it inserts instructions—*phase specifiers*—to specify the beginning of each new phase and convey information to the framework on the number of registers and scratchpad memory required for each phase. As described in Section 4.3.2, a barrier or a fence instruction also implies a phase change, but the compiler does not insert a phase specifier for it as the resource requirement does not change.

Phase Specifiers. The phase specifier instruction contains fields to specify *(i)* the number of live registers and *(ii)* the amount of scratchpad memory in bytes, both for the next phase. Figure 35 describes the fields in the phase specifier instruction. The instruction decoder sends this information to the coordinator along with the phase change event. The coordinator keeps this information in the corresponding warp slot.

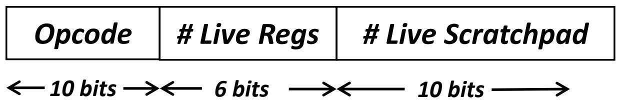


Figure 35: Phase Specifier

4.4.8 Role of the Compiler and Programmer

The compiler plays an important role, annotating the code with phase specifiers to convey information to the coordinator regarding the resource requirements of each phase. The compiler, however, does *not* alter the size of each thread block or the scratchpad memory usage of the program. The resource specification provided by the programmer (either manually or via autotuners) is retained to guarantee correctness. For registers, the compiler follows the default policy or uses directives as specified by the user. One could envision more powerful, efficient resource allocation with a programming model that does *not* require *any* resource specification and/or compiler policies/auto-tuners that are *cognizant* of the virtualized resources.

4.4.9 Implications to the Programming Model and Software Optimization

Zorua offers several new opportunities and implications in enhancing the programming model and software optimizations (via libraries, autotuners, optimizing compilers, etc.) which we briefly describe below. We leave these ideas for exploration in future work.

Flexible programming models for GPUs and heterogeneous systems. State-of-the-art high-level programming languages and models still assume a fixed amount of on-chip resources and hence, with the help of the compiler or the runtime system, are required to find *static* resource

[13]The phase boundaries for the applications in our pool easily fit this restriction, but the framework can be extended to support control divergence if needed.

specifications to fit the application to the desired GPU. Zorua, by itself, also still requires the programmer to specify resource specifications to ensure correctness—albeit they are not required to be highly optimized for a given architecture. However, by providing a flexible but dynamically-controlled view of the on-chip hardware resources, Zorua changes the abstraction of the on-chip resources that is offered to the programmer and software. This offers the opportunity to rethink resource management in GPUs from the ground up. One could envision more powerful resource allocation and better programmability with programming models that do *not* require static resource specification, leaving the compiler/runtime system and the underlying virtualized framework to completely handle *all* forms of on-chip resource allocation, unconstrained by the fixed physical resources in a specific GPU, entirely at runtime. This is especially significant in future systems that are likely to support a wide range of compute engines and accelerators, making it important to be able to write high-level code that can be partitioned easily, efficiently, and at a fine granularity across any accelerator, *without* statically tuning any code segment to run efficiently on the GPU.

Virtualization-aware compilation and autotuning. Zorua changes the contract between the hardware and software to provide a more powerful resource abstraction (in the software) that is *flexible and dynamic*, by pushing some more functionality into the hardware, which can more easily react to the runtime resource requirements of the running program. We can re-imagine compilers and autotuners to be more intelligent, leveraging this new abstraction and, hence the virtualization, to deliver more efficient and high-performing code optimizations that are *not* possible with the fixed and static abstractions of today. They could, for example, *leverage* the oversubscription and dynamic management that Zorua provides to tune the code to more aggressively use resources that are underutilized at runtime. As we demonstrate in this work, static optimizations are limited by the fixed view of the resources that is available to the program today. Compilation frameworks that are cognizant of the *dynamic* allocation/deallocation of resources provided by Zorua could make more efficient use of the available resources.

Reduced optimization space. Programs written for applications in machine learning, computer graphics, computer vision, etc., typically follow the *stream* programming paradigm, where the code is decomposed into many *stages* in an *execution pipeline.* Each stage processes only a part of the input data in a pipelined fashion to make better use of the caches. A key challenge in writing complex pipelined code is finding *execution schedules* (i.e., how the work should be partitioned across stages) and optimizations that perform best for *each* pipeline stage from a prohibitively large space of potential solutions. This requires complex tuning algorithms or profiling runs that are both computationally intensive and time-consuming. The search for optimized specifications has to be done when there is a change in input data or in the underlying architecture. By pushing some of the resource management functionality to the hardware, Zorua reduces this search space for optimized specifications by making it less sensitive to the wide space of resource specifications.

4.5 Evaluation

We evaluate the effectiveness of Zorua by studying three different mechanisms: (i) *Baseline*, the baseline GPU that schedules kernels and manages resources at the thread block level; (ii) *WLM* (Warp Level Management), a state-of-the-art mechanism for GPUs to schedule kernels and manage registers at the warp level [355]; and (iii) *Zorua*. For our evaluations, we run each application on 8–65 (36 on average) different resource specifications (the ranges are in Table 4).

4.5.1 Effect on Performance Variation and Cliffs

We first examine how Zorua alleviates the high variation in performance by reducing the impact of resource specifications on resource utilization. Figure 36 presents a Tukey box plot [219] (see Section 4.2 for a description of the presented box plot), illustrating the performance distribution (higher is better) for each application (for all different application resource specifications we evaluated), normalized to the slowest Baseline operating point *for that application*. We make two major observations.

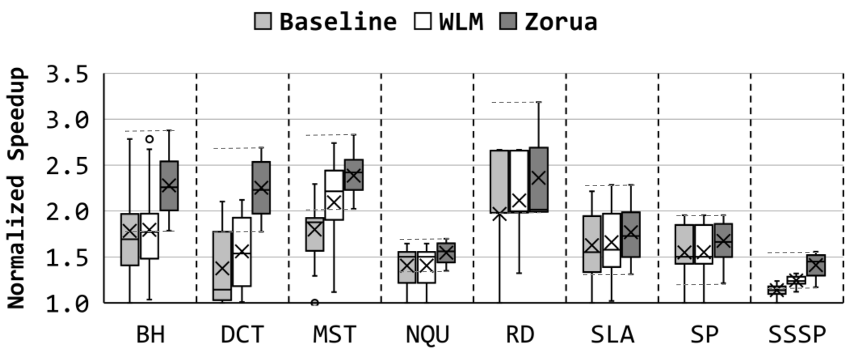
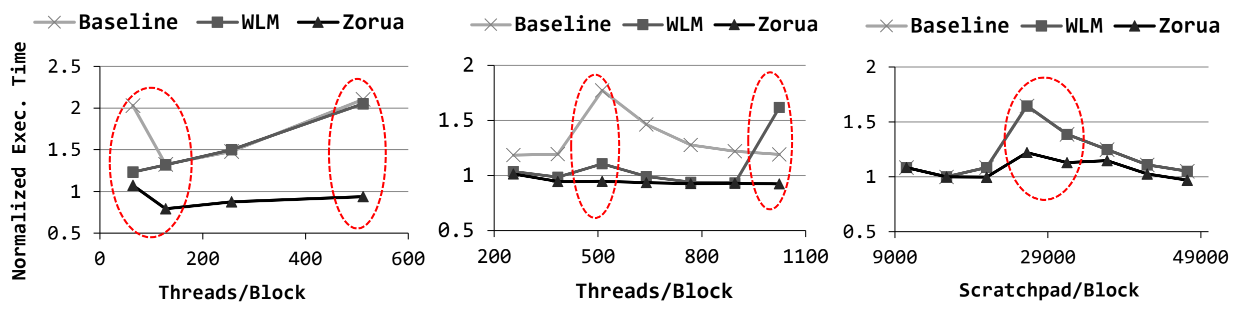


Figure 36: Normalized performance distribution.

First, we find that Zorua significantly reduces the *performance range* across all evaluated resource specifications. Averaged across all of our applications, the worst resource specification for Baseline achieves 96.6% lower performance than the best performing resource specification. For WLM [355], this performance range reduces only slightly, to 88.3%. With Zorua, the performance range drops significantly, to 48.2%. We see drops in the performance range for *all* applications except *SSSP*. With *SSSP*, the range is already small to begin with (23.8% in Baseline), and Zorua exploits the dynamic underutilization, which improves performance but also adds a small amount of variation.

Second, while Zorua reduces the performance range, it also preserves or improves performance of the best performing points. As we examine in more detail in Section 4.5.2, the reduction in performance range occurs as a result of improved performance mainly at the lower end of the distribution.

To gain insight into how Zorua reduces the performance range and improves performance for the worst performing points, we analyze how it reduces performance cliffs. With Zorua, we ideally want to *eliminate* the cliffs we observed in Section 4.2.1. We study the tradeoff between resource specification and execution time for three representative applications: *DCT* (Figure 37a), *MST* (Figure 37b), and *NQU* (Figure 37c). For all three figures, we normalize execution time to the *best* execution time under Baseline. Two observations are in order.



(a) *DCT* (b) *MST* (c) *NQU*

Figure 37: Effect on performance cliffs.

First, Zorua successfully mitigates the performance cliffs that occur in Baseline. For example, *DCT* and *MST* are both sensitive to the thread block size, as shown in Figures 37a and 37b, respectively. We have circled the locations at which cliffs exist in Baseline. Unlike Baseline, Zorua maintains more steady execution times across the number of threads per block, employing oversubscription to overcome the loss in parallelism due to insufficient on-chip resources. We see similar results across all of our applications.

Second, we observe that while WLM [355] can reduce some of the cliffs by mitigating the impact of large block sizes, many cliffs still exist under WLM (e.g., *NQU* in Figure 37c). This cliff in *NQU* occurs as a result of insufficient scratchpad memory, which cannot be handled by warp-level management. Similarly, the cliffs for *MST* (Figure 37b) also persist with WLM because *MST* has a lot of barrier operations, and the additional warps scheduled by WLM ultimately stall, waiting for other warps within the same block to acquire resources. We find that, with oversubscription, Zorua is able to smooth out those cliffs that WLM is unable to eliminate.

Overall, we conclude that Zorua *(i)* reduces the performance variation across resource specification points, so that performance depends less on the specification provided by the programmer; and *(ii)* can alleviate the performance cliffs experienced by GPU applications.

4.5.2 Effect on Performance

As Figure 36 shows, Zorua either retains or improves the best performing point for each application, compared to the Baseline. Zorua improves the best performing point for each application by 12.8% on average, and by as much as 27.8% (for *DCT*). This improvement comes from the improved parallelism obtained by exploiting the dynamic underutilization of resources, which exists *even for optimized specifications*. Applications such as *SP* and *SLA* have little dynamic underutilization, and hence do not show any performance improvement. *NQU does* have significant dynamic underutilization, but Zorua does not improve the best performing point as the overhead of oversubscription outweighs the benefit, and Zorua dynamically chooses not to oversubscribe. We conclude that even for many specifications that are optimized to fit the hardware resources, Zorua is able to further improve performance.

We also note that, in addition to reducing performance variation and improving performance for optimized points, Zorua improves performance by 25.2% on average for all resource specifications across all evaluated applications.

4.5.3 Effect on Portability

As we describe in Section 4.2.2, performance cliffs often behave differently across different GPU architectures, and can significantly shift the best performing resource specification point. We study how Zorua can ease the burden of performance tuning if an application has been already tuned for one GPU model, and is later ported to another GPU. To understand this, we define a new metric, *porting performance loss*, that quantifies the performance impact of porting an application without re-tuning it. To calculate this, we first normalize the execution time of each specification point to the execution time of the best performing specification point. We then pick a source GPU architecture (i.e., the architecture that the GPU was tuned for) and a target GPU architecture (i.e., the architecture that the code will run on), and find the point-to-point drop in performance for all points whose performance on the source GPU comes within 5% of the performance at the best performing specification point.[14]

Figure 38 shows the *maximum* porting performance loss for each application, across any two pairings of our three simulated GPU architectures (Fermi, Kepler, and Maxwell). We find that Zorua greatly reduces the maximum porting performance loss that occurs under both Baseline and WLM for all but one of our applications. On average, the maximum porting performance loss is 52.7% for Baseline, 51.0% for WLM, and only 23.9% for Zorua.

Notably, Zorua delivers significant improvements in portability for applications that previously suffered greatly when ported to another GPU, such as *DCT* and *MST*. For both of these applications, the performance variation differs so much between GPU architectures that, despite tuning the application on the source GPU to be within 5% of the best achievable performance, their performance on the target GPU is often more than twice as slow as the best achievable performance

[14]We include any point within 5% of the best performance as there are often multiple points close to the best point, and the programmer may choose any of them.

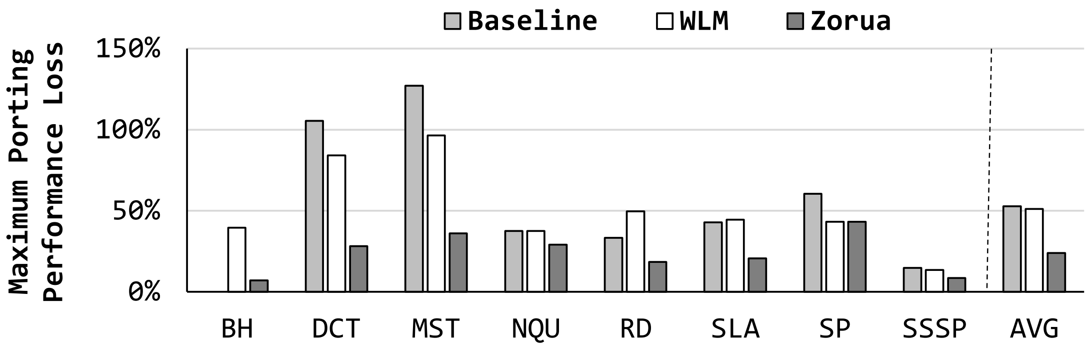


Figure 38: Maximum porting performance loss.

on the target platform. Zorua significantly lowers this porting performance loss down to 28.1% for *DCT* and 36.1% for *MST*. We also observe that for *BH*, Zorua actually increases the porting performance loss slightly with respect to the Baseline. This is because for Baseline, there are only two points that perform within the 5% margin for our metric, whereas with Zorua, we have five points that fall in that range. Despite this, the increase in porting performance loss for *BH* is low, deviating only 7.0% from the best performance.

To take a closer look into the portability benefits of Zorua, we run experiments to obtain the performance sensitivity curves for each application using different GPU architectures. Figures 39 and 40 depict the execution time curves while sweeping a single resource specification for *NQU* and *DCT* for the three evaluated GPU architectures – Fermi, Kepler, and Maxwell. We make two major observations from the figures.

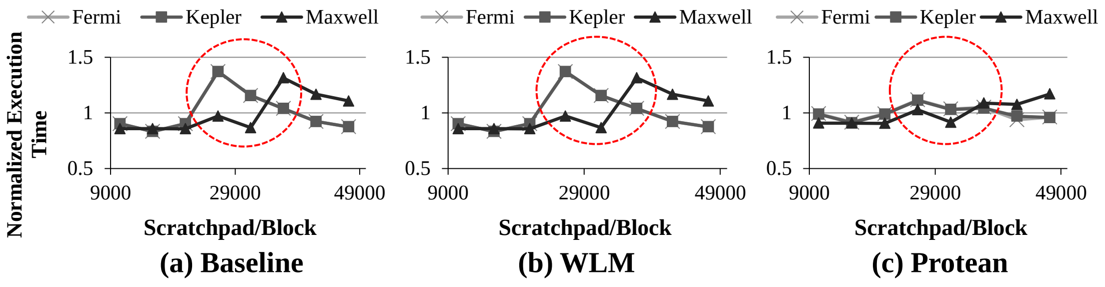


Figure 39: Impact on portability (NQU).

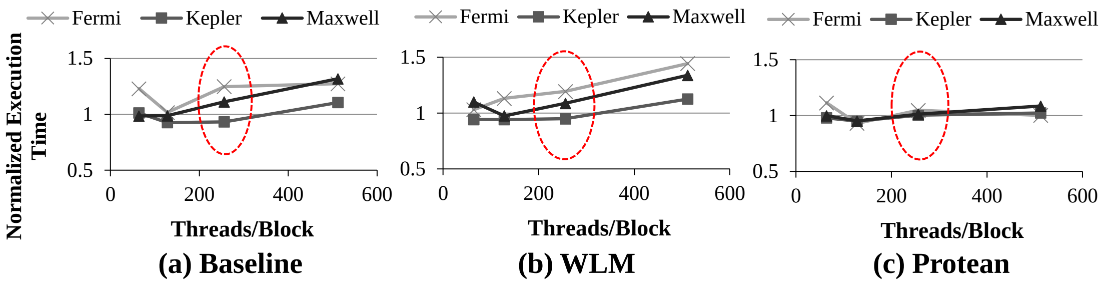


Figure 40: Impact on portability (DCT)

First, Zorua significantly alleviates the presence of performance cliffs and reduces the performance variation across *all* three evaluated architectures, thereby reducing the impact of both resource specification and underlying architecture on the resulting performance curve. In comparison, WLM is unable to make a significant impact on the performance variations and the cliffs remain for all the evaluated architectures.

Second, by reducing the performance variation across all three GPU generations, Zorua significantly reduces the *porting performance loss*, i.e., the loss in performance when code optimized for one GPU generation is run on another (as highlighted within the figures).

We conclude that Zorua enhances portability of applications by reducing the impact of a change in the hardware resources for a given resource specification. For applications that have already been tuned on one platform, Zorua significantly lowers the penalty of not re-tuning for another platform, allowing programmers to save development time.

4.5.4 A Deeper Look: Benefits & Overheads

To take a deeper look into how Zorua is able to provide the above benefits, in Figure 41, we show the number of *schedulable warps* (i.e., warps that are available to be scheduled by the warp scheduler at any given time excluding warps waiting at a barrier), averaged across all of specification points. On average, Zorua increases the number of schedulable warps by 32.8%, significantly more than WLM (8.1%), which is constrained by the fixed amount of available resources. We conclude that by oversubscribing and dynamically managing resources, Zorua is able to improve thread-level parallelism, and hence performance.

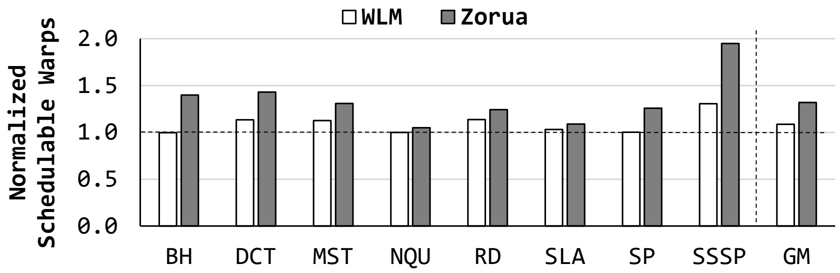


Figure 41: Effect on schedulable warps.

We also find that the overheads due to resource swapping and contention do not significantly impact the performance of Zorua. Figure 42 depicts resource hit rates for each application, i.e., the fraction of all resource accesses that were found on-chip as opposed to making a potentially expensive off-chip access. The oversubscription mechanism (directed by the coordinator) is able to keep resource hit rates very high, with an average hit rate of 98.9% for the register file and 99.6% for scratchpad memory.

Figure 43 shows the average reduction in total system energy consumption of WLM and Zorua

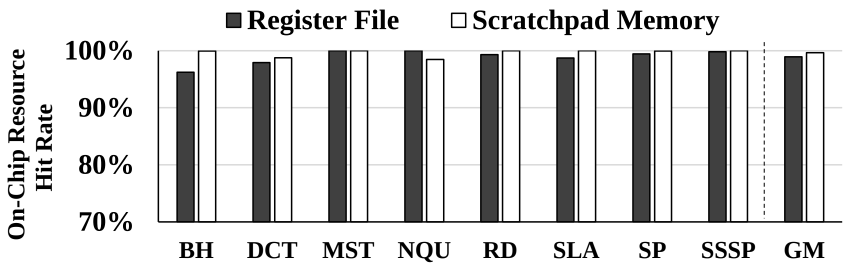


Figure 42: Virtual resource hit rate in Zorua

over Baseline for each application (averaged across the individual energy consumption over Baseline for each evaluated specification point). We observe that Zorua reduces the total energy consumption across all of our applications, except for *NQU* (which has a small increase of 3%). Overall, Zorua provides a mean energy reduction of 7.6%, up to 20.5% for *DCT*.[15] We conclude that Zorua is an energy-efficient virtualization framework for GPUs.

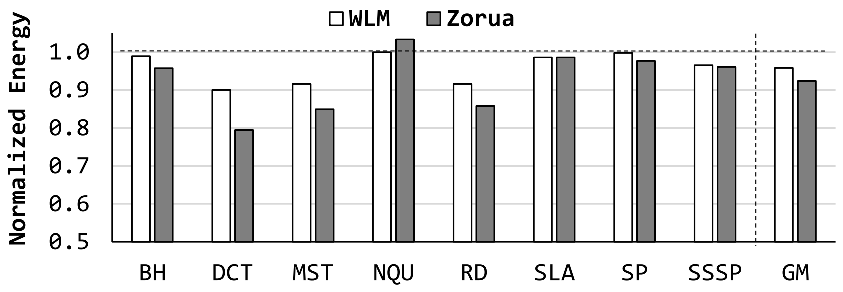


Figure 43: Effect on energy consumption.

We estimate the die area overhead of Zorua with CACTI 6.5 [352], using the same 40nm process node as the GTX 480 , which our system closely models. We include all the overheads from the coordinator and the resource mapping tables (Section 4.4). The total area overhead is 0.735 mm^2 for all 15 SMs, which is only 0.134% of the die area of the GTX 480.

4.6 Other Applications

By providing the illusion of more resources than physically available, Zorua provides the opportunity to help address other important challenges in GPU computing today. We discuss several such opportunities in this section.

4.6.1 Resource Sharing in Multi-Kernel or Multi-Programmed Environments

Executing multiple kernels or applications within the same SM can improve resource utilization and efficiency [27, 29, 30, 119, 153, 251, 348, 385]. Hence, providing support to enable fine-grained

[15]We note that the energy consumption can be reduced further by appropriately optimizing the oversubscription algorithm. We leave this exploration to future work.

sharing and partitioning of resources is critical for future GPU systems. This is especially true in environments where multiple different applications may be consolidated on the same GPU, e.g. in clouds or clusters. By providing a flexible view of each of the resources, Zorua provides a natural way to enable dynamic and fine-grained control over resource partitioning and allocation among multiple kernels. Specifically, Zorua provides several key benefits for enabling better performance and efficiency in multi-kernel/multi-program environments. First, selecting the optimal resource specification for an application is challenging in virtualized environments (e.g., clouds), as it is unclear which other applications may be running alongside it. Zorua can improve efficiency in resource utilization *irrespective* of the application specifications and of other kernels that may be executing on the same SM. Second, Zorua manages the different resources independently and at a fine granularity, using a dynamic runtime system (the coordinator). This enables the maximization of resource utilization, while providing the ability to control the partitioning of resources at runtime to provide QoS, fairness, etc., by leveraging the coordinator. Third, Zorua enables oversubscription of the different resources. This obviates the need to alter the application specifications [251, 385] in order to ensure there are sufficient resources to co-schedule kernels on the same SM, and hence enables concurrent kernel execution transparently to the programmer.

4.6.2 Preemptive Multitasking

A key challenge in enabling true multiprogramming in GPUs is enabling rapid preemption of kernels [256, 319, 348]. Context switching on GPUs incurs a very high latency and overhead, as a result of the large amount of register file and scratchpad state that needs to be saved before a new kernel can be executed. Saving state at a very coarse granularity (e.g., the entire SM state) leads to very high preemption latencies. Prior work proposes context minimization [222, 256] or context switching at the granularity of a thread block [348] to improve response time during preemption. Zorua enables fine-grained management and oversubscription of on-chip resources. It can be naturally extended to enable quick preemption of a task via intelligent management of the swap space and the mapping tables (complementary to approaches taken by prior work [222, 256]).

4.6.3 Support for Other Parallel Programming Paradigms

The fixed static resource allocation for each thread in modern GPU architectures requires statically dictating the resource usage for the program throughout its execution. Other forms of parallel execution that are *dynamic* (e.g., Cilk [45], staged execution [151, 152, 317]) require more flexible allocation of resources at runtime, and are hence more challenging to enable. Examples of this include *nested parallelism* [186], where a kernel can dynamically spawn new kernels or thread blocks, and *helper threads* [337] to utilize idle resource at runtime to perform different optimizations or background tasks in parallel. Zorua makes it easy to enable these paradigms by providing on-demand dynamic allocation of resources. Irrespective of whether threads in the programming model are created statically or dynamically, Zorua allows allocation of the required resources on the fly to support the execution of these threads. The resources are simply deallocated when they are no longer required. Zorua also enables *heterogeneous* allocation of resources – i.e., allocating different amounts of resources to different threads. The current resource allocation model, in line with a GPU's SIMT architecture, treats all threads the same and allocates the same amount of resources. Zorua makes it easier to support execution paradigms where each concurrently-running thread executes different code at the same time, hence requiring different resources. This includes helper threads, multiprogrammed execution, nested parallelism, etc. Hence, with Zorua, applications are no longer limited by a GPU's fixed SIMT model which only supports a fixed, statically-determined number of homogeneous threads as a result of the resource management mechanisms that exist today.

4.6.4 Energy Efficiency and Scalability

To support massive parallelism, on-chip resources are a precious and critical resource. However, these resources *cannot* grow arbitrarily large as GPUs continue to be area-limited and on-chip memory tends to be extremely power hungry and area intensive [6, 107, 109, 145, 280, 370]. Furthermore, complex thread schedulers that can select a thread for execution from an increasingly large thread pool are required in order to support an arbitrarily large number of warp slots. Zorua enables using smaller register files, scratchpad memory and less complex or fewer thread schedulers to save power and area while still retaining or improving parallelism.

4.6.5 Error Tolerance and Reliability

The indirection offered by Zorua, along with the dynamic management of resources, could also enable better reliability and simpler solutions towards error tolerance in the on-chip resources. The virtualization framework trivially allows remapping resources with hard or soft faults such that no virtual resource is mapped to a faulty physical resource. Unlike in the baseline case, faulty resources would not impact the number of the resources seen by the thread scheduler while scheduling threads for execution. A few unavailable faulty registers, warp slots, etc., could significantly reduce the number of the threads that are scheduled concurrently (i.e., the runtime parallelism).

4.6.6 Support for System-Level Tasks on GPUs

As GPUs become increasingly general purpose, a key requirement is better integration with the CPU operating system, and with complex distributed software systems such as those employed for large-scale distributed machine learning [4, 132] or graph processing [10, 210]. If GPUs are architected to be first-class compute engines, rather than the slave devices they are today, they can be programmed and utilized in the same manner as a modern CPU. This integration requires the GPU execution model to support system-level tasks like interrupts, exceptions, etc. and more generally provide support for access to distributed file systems, disk I/O, or network communication. Support for these tasks and execution models require dynamic provisioning of resources for execution of system-level code. Zorua provides a building block to enable this.

4.6.7 Applicability to General Resource Management in Accelerators

Zorua uses a program *phase* as the granularity for managing resources. This allows handling resources across phases *dynamically*, while leveraging *static* information regarding resource requirements from the software by inserting annotations at phase boundaries. Future work could potentially investigate the applicability of the same approach to manage resources and parallelism in *other* accelerators (e.g., processing-in-memory accelerators [10–12, 48, 112, 123, 131, 133, 172, 173, 178, 209, 257–259, 293, 295, 297, 301, 312] or direct-memory access engines [57, 185, 296]) that require efficient dynamic management of large amounts of particular critical resources.

4.7 Related Work

To our knowledge, this is the first work to propose a holistic framework to decouple a GPU application's resource specification from its physical on-chip resource allocation by virtualizing multiple on-chip resources. This enables the illusion of more resources than what physically exists to the programmer, while the hardware resources are managed at runtime by employing a swap space (in main memory), transparently to the programmer.

We briefly discuss prior work related to aspects specific to Zorua (a more general discussion is in Section 1.5): *(i)* virtualization of resources, *(ii)* more efficient management of on-chip resources.

Virtualization of Resources. *Virtualization* [79, 85, 121, 140] is a concept designed to provide the illusion, to the software and programmer, of more resources than what truly exists in physical hardware. It has been applied to the management of hardware resources in many different contexts [21, 40, 79, 85, 121, 140, 249, 342], with virtual memory [85, 140] being one of the oldest forms of virtualization that is commonly used in high-performance processors today. Abstraction of hardware resources and use of a level of indirection in their management leads to many benefits, including improved utilization, programmability, portability, isolation, protection, sharing, and oversubscription.

In this work, we apply the general principle of virtualization to the management of multiple on-chip resources in modern GPUs. Virtualization of on-chip resources offers the opportunity to alleviate many different challenges in modern GPUs. However, in this context, effectively adding a level of indirection introduces new challenges, necessitating the design of a new virtualization strategy. There are two key challenges. First, we need to dynamically determine the *extent* of the virtualization to reach an effective tradeoff between improved parallelism due to oversubscription and the latency/capacity overheads of swap space usage. Second, we need to coordinate the virtualization of *multiple* latency-critical on-chip resources. To our knowledge, this is the first work to propose a holistic software-hardware cooperative approach to virtualizing multiple on-chip resources in a controlled and coordinated manner that addresses these challenges, enabling the different benefits provided by virtualization in modern GPUs.

Prior works propose to virtualize a specific on-chip resource for specific benefits, mostly in the CPU context. For example, in CPUs, the concept of virtualized registers was first used in the IBM 360 [21] and DEC PDP-10 [40] architectures to allow logical registers to be mapped to either fast yet expensive physical registers, or slow and cheap memory. More recent works [249, 360, 361], propose to virtualize registers to increase the effective register file size to much larger register counts. This increases the number of thread contexts that can be supported in a multi-threaded processor [249], or reduces register spills and fills [360, 361]. Other works propose to virtualize on-chip resources in CPUs (e.g., [49, 77, 98, 116, 376]). In GPUs, Jeon et al. [145] propose to virtualize the register file by dynamically allocating and deallocating physical registers to enable more parallelism with smaller, more power-efficient physical register files. Concurrent to this work, Yoon et al. [370] propose an approach to virtualize thread slots to increase thread-level parallelism. These works propose specific virtualization mechanisms for a single resource for specific benefits. None of these works provide a cohesive virtualization mechanism for *multiple* on-chip GPU resources in a controlled and coordinated manner, which forms a key contribution of this work.

Efficient Resource Management. Prior works aim to improve parallelism by increasing resource utilization using hardware-based [27, 28, 110, 145, 154, 155, 189, 355, 367] and software-based [119, 126, 179, 251, 367] approaches. Among these works, the closest to ours are [145, 370] (discussed earlier), [367] and [355]. These approaches propose efficient techniques to dynamically manage a single resource, and can be used along with Zorua to improve resource efficiency further. Yang et al. [367] aim to maximize utilization of the scratchpad with software techniques, and by dynamically allocating/deallocating scratchpad. Xiang et al. [355] propose to improve resource utilization by scheduling threads at the finer granularity of a warp rather than a thread block. This approach can help alleviate performance cliffs, but not in the presence of synchronization or scratchpad memory, nor does it address the dynamic underutilization within a thread during runtime. We quantitatively compare to this approach in the evaluation and demonstrate Zorua's benefits over it.

Other works leverage resource underutilization to improve energy efficiency [6, 107, 109, 145, 286] or perform other useful work [184, 337]. These works are complementary to Zorua.

4.8 Summary

We propose Zorua, a new framework that decouples the application resource specification from the allocation in the physical hardware resources (i.e., registers, scratchpad memory, and thread slots) in GPUs. Zorua encompasses a holistic virtualization strategy to effectively virtualize multiple latency-critical on-chip resources in a controlled and coordinated manner. We demonstrate that by providing the illusion of more resources than physically available, via dynamic management of resources and the judicious use of a swap space in main memory, Zorua enhances *(i) programming ease* (by reducing the performance penalty of suboptimal resource specification), *(ii) portability* (by reducing the impact of different hardware configurations), and *(iii) performance* for code with an optimized resource specification (by leveraging dynamic underutilization of resources). We conclude that Zorua is an effective, holistic virtualization framework for GPUs.

We believe that the indirection provided by Zorua's virtualization mechanism makes it a generic framework that can address other challenges in modern GPUs. For example, Zorua can enable fine-grained resource sharing and partitioning among multiple kernels/applications, as well as low-latency preemption of GPU programs. Section 4.6 details many other applications of the Zorua framework. We hope that future work explores these promising directions, building on the insights and the framework developed in this work.

Chapter 5

Assist Warps

In this chapter, we propose a helper thread abstraction in GPUs to automatically leverage idle compute and memory bandwidth. We demonstrate significant idleness in GPU resources, even when code is highly optimized for any given architecture. We then demonstrate how a rich hardware-software abstraction can enable programmers to leverage idle compute and memory bandwidth to perform light-weight tasks, such as prefetching, data compression, etc.

5.1 Overview

GPUs employ fine-grained multi-threading to hide the high memory access latencies with thousands of concurrently running threads [165]. GPUs are well provisioned with different resources (e.g., SIMD-like computational units, large register files) to support the execution of a large number of these hardware contexts. Ideally, if the demand for all types of resources is properly balanced, all these resources should be fully utilized by the application. Unfortunately, this balance is very difficult to achieve in practice.

As a result, bottlenecks in program execution, e.g., limitations in memory or computational bandwidth, lead to long stalls and idle periods in the shader pipelines of modern GPUs [155, 156, 238, 299]. Alleviating these bottlenecks with optimizations implemented in dedicated hardware requires significant engineering cost and effort. Fortunately, the resulting under-utilization of on-chip computational and memory resources from these imbalances in application requirements, offers some new opportunities. For example, we can use these resources for efficient integration of *hardware-generated threads* that perform useful work to accelerate the execution of the primary threads. Similar *helper threading* ideas have been proposed in the context of general-purpose processors [58, 59, 73, 90, 91, 276, 387] to either extend the pipeline with more contexts or use spare hardware contexts to pre-compute useful information that aids main code execution (e.g., to aid branch prediction, prefetching, etc.).

We believe that the general idea of helper threading can lead to even more powerful optimizations and new opportunities in the context of modern GPUs than in CPUs because (1) the abundance of on-chip resources in a GPU obviates the need for idle hardware contexts [73, 74] or the addition of more storage (registers, rename tables, etc.) and compute units [58, 230] required to handle more contexts and (2) the relative simplicity of the GPU pipeline avoids the complexities of handling register renaming, speculative execution, precise interrupts, etc. [59]. However, GPUs that execute and manage thousands of thread contexts at the same time pose new challenges for employing helper threading, which must be addressed carefully. First, the numerous regular program threads executing in parallel could require an equal or larger number of helper threads that need to be managed at low cost. Second, the compute and memory resources are dynamically partitioned between threads in GPUs, and resource allocation for helper threads should be cognizant of resource interference and overheads. Third, lock-step execution and complex scheduling—which are characteristic of GPU architectures—exacerbate the complexity of fine-grained management of helper threads.

In this work, we describe a new, flexible framework for bottleneck acceleration in GPUs via helper threading (called *Core-Assisted Bottleneck Acceleration* or CABA), which exploits the aforementioned new opportunities while effectively handling the new challenges. CABA performs acceleration by generating special warps—*assist warps*—that can execute code to speed up application execution and system tasks. To simplify the support of the numerous assist threads with CABA, we manage their execution at the granularity of a *warp* and use a centralized mechanism to track the progress of each *assist warp* throughout its execution. To reduce the overhead of providing and managing new contexts for each generated thread, as well as to simplify scheduling and data communication, an assist warp *shares the same context* as the regular warp it assists. Hence, the regular warps are overprovisioned with *available registers* to enable each of them to host its own assist warp.

Use of CABA for compression. We illustrate an important use case for the CABA framework: alleviating the memory bandwidth bottleneck by enabling *flexible data compression* in the memory hierarchy. The basic idea is to have assist warps that (1) compress cache blocks before they are written to memory, and (2) decompress cache blocks before they are placed into the cache.

CABA-based compression/decompression provides several benefits over a purely hardware-based implementation of data compression for memory. First, CABA primarily employs hardware that is already available on-chip but is otherwise underutilized. In contrast, hardware-only compression implementations require *dedicated logic* for specific algorithms. Each new algorithm (or a modification of an existing one) requires engineering effort and incurs hardware cost. Second, different applications tend to have distinct data patterns [264] that are more efficiently compressed with different compression algorithms. CABA offers versatility in algorithm choice as we find that many existing hardware-based compression algorithms (e.g., Base-Delta-Immediate (BDI) compression [264], Frequent Pattern Compression (FPC) [13], and C-Pack [66]) can be implemented using different assist warps with the CABA framework. Third, not all applications benefit from data compression. Some applications are constrained by other bottlenecks (e.g., oversubscription of computational resources), or may operate on data that is not easily compressible. As a result, the benefits of compression may not outweigh the cost in terms of additional latency and energy spent on compressing and decompressing data. In these cases, compression can be easily disabled by CABA, and the CABA framework can be used in other ways to alleviate the current bottleneck.

Other uses of CABA. The generality of CABA enables its use in alleviating other bottlenecks with different optimizations. We discuss two examples: (1) using assist warps to perform *memoization* to eliminate redundant computations that have the same or similar inputs [24, 76, 305], by storing the results of frequently-performed computations in the main memory hierarchy (i.e., by converting the computational problem into a storage problem) and, (2) using the idle memory pipeline to perform opportunistic *prefetching* to better overlap computation with memory access. Assist warps offer a hardware/software interface to implement hybrid prefetching algorithms [96] with varying degrees of complexity. We also briefly discuss other uses of CABA for (1) redundant multithreading, (2) speculative precomputation, (3) handling interrupts, and (4) profiling and instrumentation.

Contributions. In this work, we make the following contributions:

- We introduce the *Core-Assisted Bottleneck Acceleration (CABA) Framework*, which can mitigate different bottlenecks in modern GPUs by using underutilized system resources for *assist warp* execution.
- We provide a detailed description of how our framework can be used to enable effective and flexible data compression in GPU memory hierarchies.
- We comprehensively evaluate the use of CABA for data compression to alleviate the memory bandwidth bottleneck. Our evaluations across a wide variety applications from Mars [127], CUDA [244], Lonestar [52], and Rodinia [61] benchmark suites show that CABA-based compression on average (1) reduces memory bandwidth by 2.1X, (2) improves performance by 41.7%, and (3) reduces overall system energy by 22.2%.
- We discuss at least six other use cases of CABA that can improve application performance and system management, showing that CABA is a primary general framework for taking advantage of underutilized resources in modern GPU engines.

5.2 Motivation: Bottlenecks in Resource Utilization

We observe that different bottlenecks and imbalances during program execution leave resources unutilized within the GPU cores. We motivate our proposal, CABA, by examining these inefficiencies. CABA leverages these inefficiencies as an opportunity to perform useful work.

Unutilized Compute Resources. A GPU core employs fine-grained multithreading [304, 321] of *warps*, i.e., groups of threads executing the same instruction, to hide long memory and ALU operation latencies. If the number of available warps is insufficient to cover these long latencies, the core stalls or becomes idle. To understand the key sources of inefficiency in GPU cores, we conduct an experiment where we show the breakdown of the applications' execution time spent on either useful work (*Active Cycles*) or stalling due to one of four reasons: *Compute, Memory, Data Dependence Stalls* and *Idle Cycles*. We also vary the amount of available off-chip memory bandwidth: (i) half (1/2xBW), (ii) equal to (1xBW), and (iii) double (2xBW) the peak memory bandwidth of our baseline GPU architecture. Section 5.6 details our baseline architecture and methodology.

Figure 44 shows the percentage of total issue cycles, divided into five components (as described above). The first two components—*Memory and Compute Stalls*—are attributed to the main memory and ALU-pipeline structural stalls. These stalls are because of backed-up pipelines due to oversubscribed resources that prevent warps from being issued to the respective pipelines. The third component (*Data Dependence Stalls*) is due to data dependence stalls. These stalls prevent warps from issuing new instruction(s) when the previous instruction(s) from the same warp are stalled on long-latency operations (usually memory load operations). In some applications (e.g., `dmr`), special-function-unit (SFU) ALU operations that may take tens of cycles to finish are also the source of data dependence stalls. The fourth component, *Idle Cycles*, refers to idle cycles when either all the available warps are issued to the pipelines and not ready to execute their next instruction or the instruction buffers are flushed due to a mispredicted branch. All these components are sources of inefficiency that cause the cores to be underutilized. The last component, *Active Cycles*, indicates the fraction of cycles during which at least one warp was successfully issued to the pipelines.

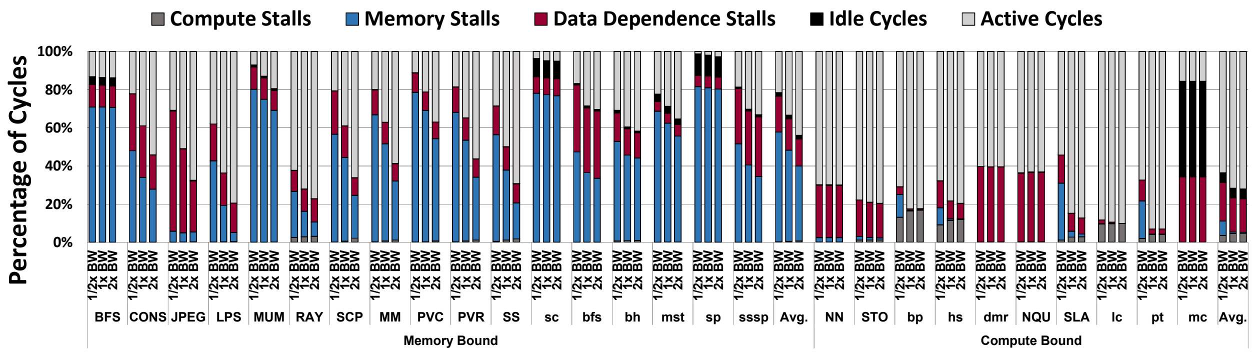


Figure 44: Breakdown of total issue cycles for 27 representative CUDA applications. See Section 5.6 for methodology.

We make two observations from Figure 44. First, *Compute, Memory*, and *Data Dependence Stalls* are the major sources of underutilization in many GPU applications. We distinguish applications based on their primary bottleneck as either *Memory* or *Compute Bound*. We observe that a majority of the applications in our workload pool (17 out of 27 studied) are *Memory Bound*, and bottlenecked by the off-chip memory bandwidth.

Second, for the *Memory Bound* applications, we observe that the *Memory* and *Data Dependence* stalls constitute a significant fraction (61%) of the total issue cycles on our baseline GPU architecture

(1xBW). This fraction goes down to 51% when the peak memory bandwidth is doubled (2xBW), and increases significantly when the peak bandwidth is halved (1/2xBW), indicating that limited off-chip memory bandwidth is a critical performance bottleneck for *Memory Bound* applications. Some applications, e.g., *BFS*, are limited by the interconnect bandwidth. In contrast, the *Compute Bound* applications are primarily bottlenecked by stalls in the ALU pipelines. An increase or decrease in the off-chip bandwidth has little effect on the performance of these applications.

Unutilized On-chip Memory. The *occupancy* of any GPU Streaming Multiprocessor (SM), i.e., the number of threads running concurrently, is limited by a number of factors: (1) the available registers and shared memory, (2) the hard limit on the number of threads and thread blocks per core, (3) the number of thread blocks in the application kernel. The limiting resource from the above, leaves the other resources underutilized. This is because it is challenging, in practice, to achieve a perfect balance in utilization of all of the above factors for different workloads with varying characteristics. Very often, the factor determining the occupancy is the thread or thread block limit imposed by the architecture. In this case, there are many registers that are left unallocated to any thread block. Also, the number of available registers may not be a multiple of those required by each thread block. The remaining registers are not enough to schedule an entire extra thread block, which leaves a significant fraction of the register file and shared memory unallocated and unutilized by the thread blocks. Figure 45 shows the fraction of statically unallocated registers in a 128KB register file (per SM) with a 1536 thread, 8 thread block occupancy limit, for different applications. We observe that on average 24% of the register file remains unallocated. This phenomenon has previously been observed and analyzed in detail in [6, 107, 109, 110, 184]. We observe a similar trend with the usage of shared memory (not graphed).

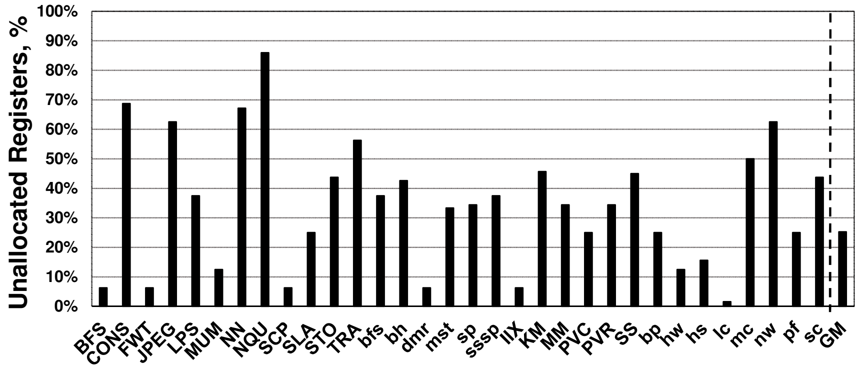


Figure 45: Fraction of statically unallocated registers.

Our Goal. We aim to exploit the underutilization of compute resources, registers and on-chip shared memory as an opportunity to enable different optimizations to accelerate various bottlenecks in GPU program execution. To achieve this goal, we would like to enable efficient helper threading for GPUs to dynamically generate threads in hardware that use the available on-chip resources for various purposes. In the next section, we present the detailed design of our CABA framework that enables the generation and management of these threads.

5.3 The CABA Framework

In order to understand the major design choices behind the CABA framework, we first present our major design goals and describe the key challenges in applying helper threading to GPUs. We then show the detailed design, hardware changes, and operation of CABA. Finally, we briefly describe potential applications of our proposed framework. Section 5.4 goes into a detailed design of one application of the framework.

5.3.1 Goals and Challenges

The purpose of CABA is to leverage underutilized GPU resources for useful computation. To this end, we need to efficiently execute subroutines that perform optimizations to accelerate bottlenecks in application execution. The key difference between CABA's *assisted execution* and regular execution is that CABA must be *low overhead* and, therefore, helper threads need to be treated differently from regular threads. The *low overhead* goal imposes several key requirements in designing a framework to enable helper threading. First, we should be able to easily manage helper threads—to enable, trigger, and kill threads when required. Second, helper threads need to be flexible enough to adapt to the runtime behavior of the regular program. Third, a helper thread needs to be able to communicate with the original thread. Finally, we need a flexible interface to specify new subroutines, with the framework being generic enough to handle various optimizations.

With the above goals in mind, enabling helper threading in GPU architectures introduces several new challenges. First, execution on GPUs involves context switching between hundreds of threads. These threads are handled at different granularities in hardware and software. The programmer reasons about these threads at the granularity of a thread block. However, at any point in time, the hardware executes only a small subset of the thread block, i.e., a set of warps. Therefore, we need to define the *abstraction levels* for reasoning about and managing helper threads from the point of view of the programmer, the hardware as well as the compiler/runtime. In addition, each of the thousands of executing threads could simultaneously invoke an associated helper thread subroutine. To keep the management overhead low, we need an efficient mechanism to handle helper threads at this magnitude.

Second, GPUs use fine-grained multithreading [304, 321] to time multiplex the fixed number of compute units among the hundreds of threads. Similarly, the on-chip memory resources (i.e., the register file and shared memory) are statically partitioned between the different threads at compile time. Helper threads require their own registers and compute cycles to execute. A straightforward approach would be to dedicate few registers and compute units just for helper thread execution, but this option is both expensive and wasteful. In fact, our primary motivation is to utilize *existing idle resources* for helper thread execution. In order to do this, we aim to enable sharing of the existing resources between primary threads and helper threads at low cost, while minimizing the interference to primary thread execution. In the remainder of this section, we describe the design of our low-overhead CABA framework.

5.3.2 Design of the CABA Framework

We choose to implement CABA using a hardware/software co-design, as pure hardware or pure software approaches pose certain challenges that we describe below. There are two alternatives for a fully software-based approach to helper threads. The first alternative, treating each helper thread as independent kernel code, has high overhead, since we are now treating the helper threads as, essentially, regular threads. This would reduce the primary thread occupancy in each SM (there is a hard limit on the number of threads and blocks that an SM can support). It would also complicate the data communication between the primary and helper threads, since no simple interface exists for inter-kernel communication. The second alternative, embedding the helper thread code within the primary thread kernel itself, offers little flexibility in adapting to runtime requirements, since such helper threads cannot be triggered or squashed independently of the primary thread.

On the other hand, a pure hardware solution would make register allocation for the assist warps and the data communication between the helper threads and primary threads more difficult. Registers are allocated to each thread block by the compiler and are then mapped to the sections of the hardware register file at runtime. Mapping registers for helper threads and enabling data communication between those registers and the primary thread registers would be non-trivial. Furthermore, a fully hardware approach would make offering the programmer a flexible interface more challenging.

Hardware support enables simpler fine-grained management of helper threads, aware of micro-architectural events and runtime program behavior. Compiler/runtime support enables simpler context management for helper threads and more flexible programmer interfaces. Thus, to get the best of both worlds, we propose a *hardware/software cooperative approach*, where the hardware manages the scheduling and execution of helper thread subroutines, while the compiler performs the allocation of shared resources (e.g., register file and shared memory) for the helper threads and the programmer or the microarchitect provides the helper threads themselves.

Hardware-based management of threads. To use the available on-chip resources the same way that thread blocks do during program execution, we dynamically insert sequences of instructions into the execution stream. We track and manage these instructions at the granularity of a warp, and refer to them as ***Assist Warps***. An assist warp is a set of instructions issued into the core pipelines. Each instruction is executed in lock-step across all the SIMT lanes, just like any regular instruction, with an active mask to disable lanes as necessary. The assist warp does *not* own a separate context (e.g., registers, local memory), and instead shares both a context and a warp ID with the regular warp that invoked it. In other words, each assist warp is coupled with a *parent warp*. In this sense, it is different from a regular warp and does not reduce the number of threads that can be scheduled on a single SM. Data sharing between the two warps becomes simpler, since the assist warps share the register file with the parent warp. Ideally, an assist warp consumes resources and issue cycles that would otherwise be idle. We describe the structures required to support hardware-based management of assist warps in Section 5.3.3.

Register file/shared memory allocation. Each helper thread subroutine requires a different number of registers depending on the actions it performs. These registers have a short lifetime, with no values being preserved between different invocations of an assist warp. To limit the register requirements for assist warps, we impose the restriction that only one instance of each helper thread routine can be active for each thread. All instances of the same helper thread for each parent thread use the same registers, and the registers are allocated to the helper threads statically by the compiler. One of the factors that determines the runtime SM occupancy is the number of registers required by a thread block (i.e, per-block register requirement). For each helper thread subroutine that is enabled, we add its register requirement to the per-block register requirement, to ensure the availability of registers for both the parent threads as well as every assist warp. The registers that remain unallocated after allocation among the parent thread blocks should suffice to support the assist warps. If not, register-heavy assist warps may limit the parent thread block occupancy in SMs or increase the number of register spills in the parent warps. Shared memory resources are partitioned in a similar manner and allocated to each assist warp as and if needed.

Programmer/developer interface. The assist warp subroutine can be written in two ways. First, it can be supplied and annotated by the programmer/developer using CUDA extensions with PTX instructions and then compiled with regular program code. Second, the assist warp subroutines can be written by the microarchitect in the internal GPU instruction format. These helper thread subroutines can then be enabled or disabled by the application programmer. This approach is similar to that proposed in prior work (e.g., [58]). It offers the advantage of potentially being highly optimized for energy and performance while having flexibility in implementing optimizations that are not trivial to map using existing GPU PTX instructions. The instructions for the helper thread subroutine are stored in an on-chip buffer (described in Section 5.3.3).

Along with the helper thread subroutines, the programmer also provides: (1) the *priority* of the assist warps to enable the warp scheduler to make informed decisions, (2) the trigger conditions for each assist warp, and (3) the live-in and live-out variables for data communication with the parent warps.

Assist warps can be scheduled with different priority levels in relation to parent warps by the warp scheduler. Some assist warps may perform a function that is required for correct execution of the program and are *blocking*. At this end of the spectrum, the *high priority* assist warps are treated by the scheduler as always taking higher precedence over the parent warp execution. Assist warps should be given a high priority only when they are required for correctness. *Low priority* assist warps, on the other hand, are scheduled for execution only when computational resources are available, i.e., during idle cycles. There is no guarantee that these assist warps will execute or complete.

The programmer also provides the conditions or events that need to be satisfied for the deployment of the assist warp. This includes a specific point within the original program and/or a set of other microarchitectural events that could serve as a *trigger* for starting the execution of an assist warp.

5.3.3 Main Hardware Additions

Figure 46 shows a high-level block diagram of the GPU pipeline [118]. To support assist warp execution, we add three new components: (1) an Assist Warp Store to hold the assist warp code, (2) an Assist Warp Controller to perform the deployment, tracking, and management of assist warps, and (3) an Assist Warp Buffer to stage instructions from triggered assist warps for execution.

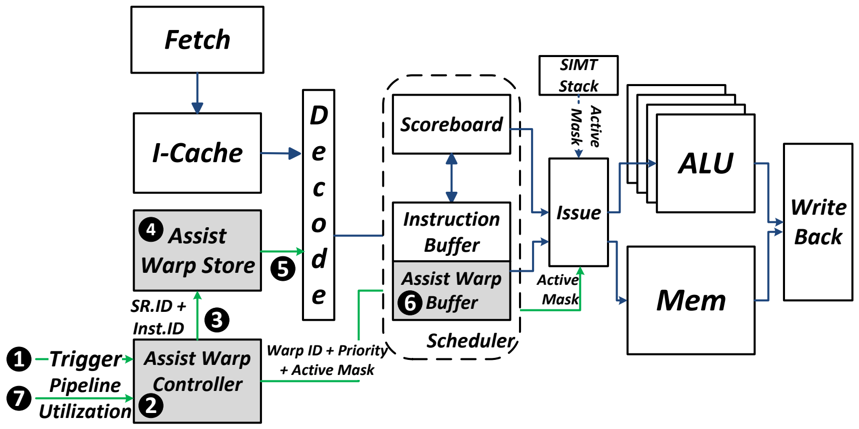


Figure 46: CABA framework flow within a typical GPU pipeline [118]. The shaded blocks are the components introduced for the framework.

Assist Warp Store (AWS). Different assist warp subroutines are possible based on the purpose of the optimization. These code sequences for different types of assist warps need to be stored on-chip. An on-chip storage structure called the Assist Warp Store (❹) is preloaded with these instructions before application execution. It is indexed using the subroutine index (SR.ID) along with the instruction ID (Inst.ID).

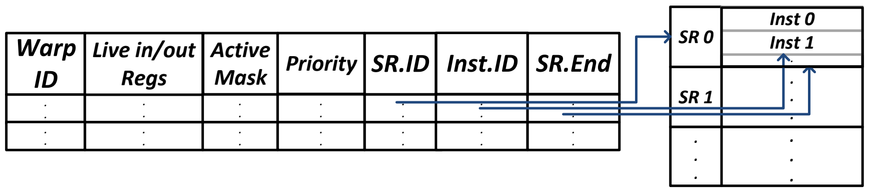


Figure 47: Fetch Logic: Assist Warp Table (contained in the AWC) and the Assist Warp Store (AWS).

Assist Warp Controller (AWC). The AWC (❷) is responsible for the triggering, tracking, and management of assist warp execution. It stores a mapping between trigger events and a subroutine index in the AWS, as specified by the programmer. The AWC monitors for such events, and when they take place, triggers the fetch, decode and execution of instructions from the AWS for the respective assist warp.

Deploying all the instructions within an assist warp, back-to-back, at the trigger point may require increased fetch/decode bandwidth and buffer space after decoding [59]. To avoid this, at each cycle, only a few instructions from an assist warp, at most equal to the available decode/issue bandwidth, are decoded and staged for execution. Within the AWC, we simply track the next instruction that needs to be executed for each assist warp and this is stored in the Assist Warp Table (AWT), as depicted in Figure 47. The AWT also tracks additional metadata required for assist warp management, which is described in more detail in Section 5.3.4.

Assist Warp Buffer (AWB). Fetched and decoded instructions (❷) belonging to the assist warps that have been triggered need to be buffered until the assist warp can be selected for issue by the scheduler. These instructions are then staged in the Assist Warp Buffer (❻) along with their warp IDs. The AWB is contained within the *instruction buffer (IB)*, which holds decoded instructions for the parent warps. The AWB makes use of the existing IB structures. The IB is typically partitioned among different warps executing in the SM. Since each assist warp is associated with a parent warp, the assist warp instructions are directly inserted into the *same partition* within the IB as that of the parent warp. This simplifies warp scheduling, as the assist warp instructions can now be issued as if they were parent warp instructions with the same warp ID. In addition, using the existing partitions avoids the cost of separate dedicated instruction buffering for assist warps. We do, however, provision a small additional partition with two entries within the IB, to hold non-blocking *low priority* assist warps that are scheduled only during idle cycles. This additional partition allows the scheduler to distinguish *low priority* assist warp instructions from the parent warp and *high priority* assist warp instructions, which are given precedence during scheduling, allowing them to make progress.

5.3.4 The Mechanism

Trigger and Deployment. An assist warp is triggered (❶) by the AWC (❷) based on a specific set of architectural events and/or a triggering instruction (e.g., a load instruction). When an assist warp is triggered, its specific instance is placed into the Assist Warp Table (AWT) within the AWC (Figure 47). Every cycle, the AWC selects an assist warp to deploy in a round-robin fashion. The AWS is indexed (❸) based on the subroutine ID (SR.ID)—which selects the instruction sequence to be executed by the assist warp, and the instruction ID (Inst.ID)—which is a pointer to the next instruction to be executed within the subroutine (Figure 47). The selected instruction is entered (❺) into the AWB (❻) and, at this point, the instruction enters the active pool with other active warps for scheduling. The Inst.ID for the assist warp is updated in the AWT to point to the next instruction in the subroutine. When the end of the subroutine is reached, the entry within the AWT is freed.

Execution. Assist warp instructions, when selected for issue by the scheduler, are executed in much the same way as any other instructions. The scoreboard tracks the dependencies between instructions within an assist warp in the same way as any warp, and instructions from different assist warps are interleaved in execution in order to hide latencies. We also provide an active mask (stored as a part of the AWT), which allows for statically disabling/enabling different lanes within a warp. This is useful to provide flexibility in lock-step instruction execution when we do not need all threads within a warp to execute a specific assist warp subroutine.

Dynamic Feedback and Throttling. Assist warps, if not properly controlled, may stall application execution. This can happen due to several reasons. First, assist warps take up issue cycles, and only a limited number of instructions may be issued per clock cycle. Second, assist warps require structural resources: the ALU units and resources in the load-store pipelines (if the assist warps consist of computational and memory instructions, respectively). We may, hence, need to throttle assist warps to ensure that their performance benefits outweigh the overhead. This requires mechanisms to appropriately balance and manage the aggressiveness of assist warps at runtime.

The overheads associated with assist warps can be controlled in different ways. First, the programmer can statically specify the priority of the assist warp. Depending on the criticality of the assist warps in making forward progress, the assist warps can be issued either in idle cycles or with varying levels of priority in relation to the parent warps. For example, warps performing *decompression* are given a high priority whereas warps performing *compression* are given a low priority. Low priority assist warps are inserted into the dedicated partition in the IB, and are scheduled only during idle cycles. This priority is statically defined by the programmer. Second, the AWC can control the number of times the assist warps are deployed into the AWB. The AWC monitors the utilization of the functional units (❼) and idleness of the cores to decide when to throttle assist warp deployment.

Communication and Control. An assist warp may need to communicate data and status with its parent warp. For example, memory addresses from the parent warp need to be communicated to assist warps performing decompression or prefetching. The IDs of the registers containing the live-in data for each assist warp are saved in the AWT when an assist warp is triggered. Similarly, if an assist warp needs to report results to its parent warp (e.g., in the case of memoization), the register IDs are also stored in the AWT. When the assist warps execute, *MOVE* instructions are first executed to copy the live-in data from the parent warp registers to the assist warp registers. Live-out data is communicated to the parent warp in a similar fashion, at the end of assist warp execution.

Assist warps may need to be *killed* when they are not required (e.g., if the data does not require decompression) or when they are no longer beneficial. In this case, the entries in the AWT and AWB are simply flushed for the assist warp.

5.3.5 Applications of the CABA Framework

We envision multiple applications for the CABA framework, e.g., data compression [13, 66, 264, 364], memoization [24, 76, 305], data prefetching [31, 104, 157, 252, 310]. In Section 5.4, we provide a detailed case study of enabling data compression with the framework, discussing various tradeoffs. We believe CABA can be useful for many other optimizations, and we discuss some of them briefly in Section 5.5.

5.4 A Case for CABA: Data Compression

Data compression is a technique that exploits the redundancy in the applications' data to reduce capacity and bandwidth requirements for many modern systems by saving and transmitting data in a more compact form. Hardware-based data compression has been explored in the context of on-chip caches [13, 22, 66, 93, 138, 262, 264, 287, 364], interconnect [80], and main memory [5, 97, 261, 263, 300, 323] as a means to save storage capacity as well as memory bandwidth. In modern GPUs, memory bandwidth is a key limiter to system performance in many workloads (Section 5.2). As such, data compression is a promising technique to help alleviate this bottleneck. Compressing data enables less data to be transferred from/to DRAM and the interconnect.

In bandwidth-constrained workloads, idle compute pipelines offer an opportunity to employ CABA to enable data compression in GPUs. We can use assist warps to (1) decompress data, before loading it into the caches and registers, and (2) compress data, before writing it back to memory. Since assist warps execute instructions, CABA offers some flexibility in the compression algorithms that can be employed. Compression algorithms that can be mapped to the general

GPU execution model can be flexibly implemented with the CABA framework.

5.4.1 Mapping Compression Algorithms into Assist Warps

In order to employ CABA to enable data compression, we need to map compression algorithms into instructions that can be executed within the GPU cores. For a compression algorithm to be amenable for implementation with CABA, it ideally needs to be (1) reasonably parallelizable and (2) simple (for low latency). Decompressing data involves reading the encoding associated with each cache line that defines how to decompress it, and then triggering the corresponding decompression subroutine in CABA. Compressing data, on the other hand, involves testing different encodings and saving data in the compressed format.

We perform compression at the granularity of a cache line. The data needs to be decompressed before it is used by any program thread. In order to utilize the full SIMD width of the GPU pipeline, we would like to decompress/compress all the words in the cache line in parallel. With CABA, helper thread routines are managed at the warp granularity, enabling fine-grained triggering of assist warps to perform compression/decompression when required. However, the SIMT execution model in a GPU imposes some challenges: (1) threads within a warp operate in lock-step, and (2) threads operate as independent entities, i.e., they do not easily communicate with each other.

In this section, we discuss the architectural changes and algorithm adaptations required to address these challenges and provide a detailed implementation and evaluation of *Data Compression* within the CABA framework using the *Base-Delta-Immediate compression* algorithm [264]. Section 5.4.2 discusses implementing other compression algorithms.

Algorithm Overview. Base-Delta-Immediate compression (BDI) is a simple compression algorithm that was originally proposed in the context of caches [264]. It is based on the observation that many cache lines contain data with low dynamic range. BDI exploits this observation to represent a cache line with low dynamic range using a common *base* (or multiple bases) and an array of *deltas* (where a delta is the difference of each value within the cache line and the common base). Since the *deltas* require fewer bytes than the values themselves, the combined size after compression can be much smaller. Figure 48 shows the compression of an example 64-byte cache line from the *PageViewCount (PVC)* application using BDI. As Figure 48 indicates, in this case, the cache line can be represented using two bases (an 8-byte base value, $0x8001D000$, and an implicit zero value base) and an array of eight 1-byte differences from these bases. As a result, the entire cache line data can be represented using 17 bytes instead of 64 bytes (1-byte metadata, 8-byte base, and eight 1-byte deltas), saving 47 bytes of the originally used space.

Our example implementation of the BDI compression algorithm [264] views a cache line as a set of fixed-size values i.e., 8 8-byte, 16 4-byte, or 32 2-byte values for a 64-byte cache line. For the size of the deltas, it considers three options: 1, 2 and 4 bytes. The key characteristic of BDI,

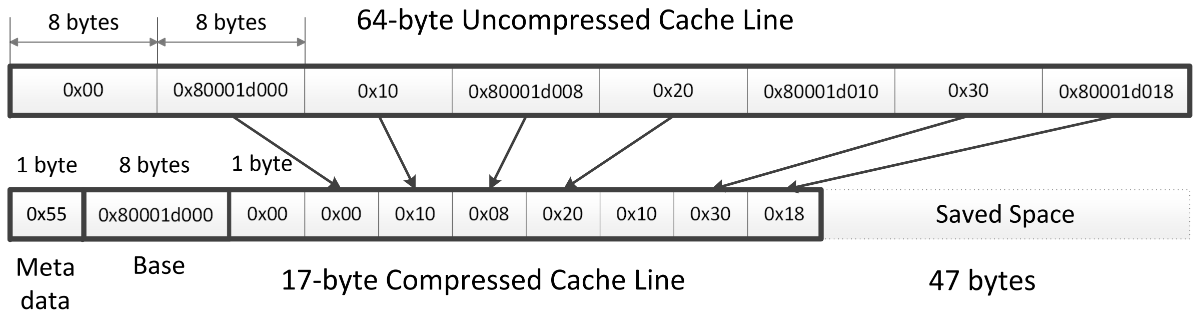


Figure 48: Cache line from *PVC* compressed with BDI.

which makes it a desirable compression algorithm to use with the CABA framework, is its fast parallel decompression that can be efficiently mapped into instructions that can be executed on GPU hardware. Decompression is simply a masked vector addition of the deltas to the appropriate bases [264].

Mapping BDI to CABA. In order to implement BDI with the CABA framework, we need to map the BDI compression/decompression algorithms into GPU instruction subroutines (stored in the AWS and deployed as assist warps).

Decompression. To decompress the data compressed with BDI, we need a simple addition of deltas to the appropriate bases. The CABA decompression subroutine first loads the words within the compressed cache line into assist warp registers, and then performs the base-delta additions in parallel, employing the wide ALU pipeline.[16] The subroutine then writes back the uncompressed cache line to the cache. It skips the addition for the lanes with an implicit base of zero by updating the active lane mask based on the cache line encoding. We store a separate subroutine for each possible BDI encoding that loads the appropriate bytes in the cache line as the base and the deltas. The high-level algorithm for decompression is presented in Algorithm 1.

Algorithm 4 BDI: Decompression

```
1: load $base, deltas$
2: $uncompressed\_data = base + deltas$
3: store $uncompressed\_data$
```

Compression. To compress data, the CABA compression subroutine tests several possible encodings (each representing a different size of base and deltas) in order to achieve a high compression ratio. The first few bytes (2–8 depending on the encoding tested) of the cache line are always used as the base. Each possible encoding is tested to check whether the cache line can be successfully encoded with it. In order to perform compression at a warp granularity, we need to check whether all of the words at every SIMD lane were successfully compressed. In other

[16]Multiple instructions are required if the number of deltas exceeds the width of the ALU pipeline. We use a 32-wide pipeline.

words, if any one word cannot be compressed, that encoding cannot be used across the warp. We can perform this check by adding a global predicate register, which stores the logical AND of the per-lane predicate registers. We observe that applications with homogeneous data structures can typically use the same encoding for most of their cache lines [264]. We use this observation to reduce the number of encodings we test to just one in many cases. All necessary operations are done in parallel using the full width of the GPU SIMD pipeline. The high-level algorithm for compression is presented in Algorithm 2.

Algorithm 5 BDI: Compression

```
1:  for each base_size do
2:      load base, values
3:      for each delta_size do
4:          deltas = abs(values - base)
5:          if size(deltas) <= delta_size then
6:              store base, deltas
7:              exit
8:          end if
9:      end for
10: end for
```

5.4.2 Implementing Other Algorithms.

The BDI compression algorithm is naturally amenable towards implementation using assist warps because of its data-parallel nature and simplicity. The CABA framework can also be used to realize other algorithms. The challenge in implementing algorithms like FPC [14] and C-Pack [66], which have variable-length compressed words, is primarily in the placement of compressed words within the compressed cache lines. In BDI, the compressed words are in *fixed* locations within the cache line and, for each encoding, all the compressed words are of the same size and can, therefore, be processed in parallel. In contrast, C-Pack may employ multiple dictionary values as opposed to just one base in BDI. In order to realize algorithms with *variable length words* and *dictionary values* with assist warps, we leverage the coalescing/address generation logic [241, 248] already available in the GPU cores. We make two minor modifications to these algorithms [14, 66] to adapt them for use with CABA. First, similar to prior works [14, 66, 97], we observe that few encodings are sufficient to capture almost all the data redundancy. In addition, the impact of any loss in compressibility due to fewer encodings is minimal as the benefits of bandwidth compression are only at multiples of a single DRAM burst (e.g., 32B for GDDR5 [135]). We exploit this to reduce the number of supported encodings. Second, we place all the metadata containing the compression encoding at the *head* of the cache line to be able to determine how to decompress the entire line *upfront*. In the case of C-Pack, we place the dictionary entries after the metadata.

We note that it can be challenging to implement complex algorithms efficiently with the simple computational logic available in GPU cores. Fortunately, there are already Special Function Units (SFUs) [83, 200] present in the GPU SMs, used to perform efficient computations of elementary mathematical functions. SFUs could potentially be extended to implement primitives that enable the fast iterative comparisons performed frequently in some compression algorithms. This would enable more efficient execution of the described algorithms, as well as implementation of more complex compression algorithms, using CABA. We leave the exploration of an SFU-based approach to future work.

We now present a detailed overview of mapping the FPC and C-PACK algorithms into assist warps.

Implementing the FPC (Frequent Pattern Compression) Algorithm. For FPC, the cache line is treated as set of fixed-size words and each word within the cache line is compressed into a simple prefix or encoding and a compressed word if it matches a set of frequent patterns, e.g. narrow values, zeros or repeated bytes. The word is left uncompressed if it does not fit any pattern. We refer the reader to the original work [14] for a more detailed description of the original algorithm.

The challenge in mapping assist warps to the FPC decompression algorithm is in the serial sequence in which each word within a cache line is decompressed. This is because in the original proposed version, each compressed word can have a different size. To determine the location of a specific compressed word, it is necessary to have decompressed the previous word. We make some modifications to the algorithm in order to parallelize the decompression across different lanes in the GPU cores. First, we move the word prefixes (metadata) for each word to the front of the cache line, so we know *upfront* how to decompress the rest of the cache line. Unlike with BDI, each word within the cache line has a different encoding and hence a different compressed word length and encoding pattern. This is problematic as statically storing the sequence of decompression instructions for every combination of patterns for all the words in a cache line would require very large instruction storage. In order to mitigate this, we break each cache line into a number of segments. Each segment is compressed independently and all the words within each segment are compressed using the *same encoding* whereas different segments may have different encodings. This creates a trade-off between simplicity/parallelizability versus compressibility. Consistent with previous works [14], we find that this doesn't significantly impact compressibility.

Decompression. The high-level algorithm we use for decompression is presented in Algorithm 3. Each segment within the compressed cache line is loaded in series. Each of the segments is decompressed in parallel—this is possible because all the compressed words within the segment have the same encoding. The decompressed segment is then stored before moving onto the next segment. The location of the next compressed segment is computed based on the size of the previous segment.

Algorithm 6 FPC: Decompression

```
1: for each segment do
2:     load compressed words
3:     pattern specific decompression (sign extension/zero value)
4:     store decompressed words
5:     segment-base-address = segment-base-address + segment-size
6: end for
```

Compression. Similar to the BDI implementation, we loop through and test different encodings for each segment. We also compute the address offset for each segment at each iteration to store the compressed words in the appropriate location in the compressed cache line. Algorithm 4 presents the high-level FPC compression algorithm we use.

Algorithm 7 FPC: Compression

```
1: load words
2: for each segment do
3:     for each encoding do
4:         test encoding
5:         if compressible then
6:             segment-base-address = segment-base-address + segment-size
7:             store compressed words
8:             break
9:         end if
10:    end for
11: end for
```

Implementing the C-Pack Algorithm. C-Pack [66] is a dictionary based compression algorithm where frequent "dictionary" values are saved at the beginning of the cache line. The rest of the cache line contains encodings for each word which may indicate zero values, narrow values, full or partial matches into the dictionary or simply that the word is uncompressible.

In our implementation, we reduce the number of possible encodings to partial matches (only last byte mismatch), full word match, zero value and zero extend (only last byte) and we limit the number of dictionary values to 4. This enables fixed compressed word size within the cache line. A fixed compressed word size enables compression and decompression of different words within the cache line in parallel. If the number of required dictionary values or uncompressed words exceeds 4, the line is left decompressed. This is, as in BDI and FPC, a trade-off between simplicity and compressibility. In our experiments, we find that it does not significantly impact the compression ratio—primarily due the 32B minimum data size and granularity of compression.

Decompression. As described, to enable parallel decompression, we place the encodings and dictionary values at the head of the line. We also limit the number of encodings to enable quick decompression. We implement C-Pack decompression as a series of instructions (one per encoding used) to load all the registers with the appropriate dictionary values. We define the active lane mask based on the encoding (similar to the mechanism used in BDI) for each load instruction to ensure the correct word is loaded into each lane's register. Algorithm 5 provides the high-level algorithm for C-Pack decompression.

Algorithm 8 C-PACK: Decompression

```
1: add base-address + index-into-dictionary
2: load compressed words
3: for each encoding do
4:     pattern specific decompression          ▷ Mismatch byte load for zero extend or partial match
5: end for
6: Store uncompressed words
```

Compression. Compressing data with C-Pack involves determining the dictionary values that will be used to compress the rest of the line. In our implementation, we serially add each word from the beginning of the cache line to be a dictionary value if it was not already covered by a previous dictionary value. For each dictionary value, we test whether the rest of the words within the cache line is compressible. The next dictionary value is determined using the predicate register to determine the next uncompressed word, as in BDI. After four iterations (dictionary values), if all the words within the line are not compressible, the cache line is left uncompressed. Similar to BDI, the global predicate register is used to determine the compressibility of all of the lanes after four or fewer iterations. Algorithm 6 provides the high-level algorithm for C-Pack compression.

Algorithm 9 C-PACK: Compression

```
1: load words
2: for each dictionary value (including zero) do          ▷ To a maximum of four
3:     test match/partial match
4:     if compressible then
5:         Store encoding and mismatching byte
6:         break
7:     end if
8: end for
9: if all lanes are compressible then
10:     Store compressed cache line
11: end if
```

5.4.3 Walkthrough of CABA-based Compression

We show the detailed operation of CABA-based compression and decompression mechanisms in Figure 49. We assume a baseline GPU architecture with three levels in the memory hierarchy – two levels of caches (private L1s and a shared L2) and main memory. Different levels can potentially store compressed data. In this section and in our evaluations, we assume that only the L2 cache and main memory contain compressed data. Note that there is no capacity benefit in the baseline mechanism as compressed cache lines still occupy the full uncompressed slot, i.e., we only evaluate the bandwidth-saving benefits of compression in GPUs.

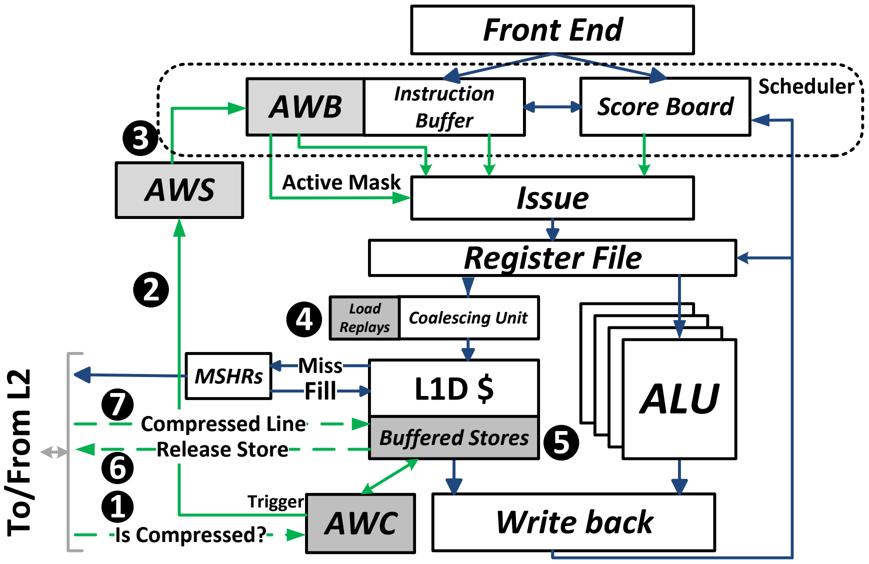


Figure 49: Walkthrough of CABA-based Compression.

The Decompression Mechanism. Load instructions that access global memory data in the compressed form trigger the appropriate assist warp to decompress the data before it is used. The subroutines to decompress data are stored in the *Assist Warp Store (AWS)*. The AWS is indexed by the compression encoding at the head of the cache line and by a bit indicating whether the instruction is a load (decompression is required) or a store (compression is required). Each decompression assist warp is given *high priority* and, hence, stalls the progress of its parent warp until it completes its execution. This ensures that the parent warp correctly gets the decompressed value.

L1 Access. We store data in L1 in the uncompressed form. An L1 hit does not require an assist warp for decompression.

L2/Memory Access. Global memory data cached in L2/DRAM could potentially be compressed. A bit indicating whether the cache line is compressed is returned to the core along with the cache line (❶). If the data is uncompressed, the line is inserted into the L1 cache and the writeback phase resumes normally. If the data is compressed, the compressed cache line is inserted into the L1 cache. The encoding of the compressed cache line and the warp ID are relayed to the Assist Warp Controller (AWC), which then triggers the AWS (❷) to deploy the appropriate assist warp (❸) to decompress the line. During regular execution, the load information for each thread is buffered in the coalescing/load-store unit [241, 248] until all the data is fetched. We continue to buffer this load information (❹) until the line is decompressed.

After the CABA decompression subroutine ends execution, the original load that triggered decompression is resumed (❹).

The Compression Mechanism. The assist warps to perform compression are triggered by store instructions. When data is written to a cache line (i.e., by a store), the cache line can be written back to main memory either in the compressed or uncompressed form. Compression is off the critical path and the warps to perform compression can be scheduled when the required resources are available.

Pending stores are buffered in a few dedicated sets within the L1 cache or in available shared memory (❺). In the case of an overflow in this buffer space (❺), the stores are released to the lower levels of the memory system in the uncompressed form (❻). Upon detecting the availability of resources to perform the data compression, the AWC triggers the deployment of the assist warp that performs compression (❷) into the AWB (❸), with *low priority*. The scheduler is then free to schedule the instructions from the compression subroutine. Since compression is not on the critical path of execution, keeping such instructions as low priority ensures that the main program is not unnecessarily delayed.

L1 Access. On a hit in the L1 cache, the cache line is already available in the uncompressed form. Depending on the availability of resources, the cache line can be scheduled for compression or simply written to the L2 and main memory uncompressed, when evicted.

L2/Memory Access. Data in memory is compressed at the granularity of a full cache line, but stores can be at granularities smaller than the size of the cache line. This poses some additional difficulty if the destination cache line for a store is already compressed in main memory. Partial writes into a compressed cache line would require the cache line to be decompressed first, then updated with the new data, and written back to main memory. The common case—where the cache line that is being written to is uncompressed initially—can be easily handled. However, in the worst case, the cache line being partially written to is already in the compressed form in memory. We now describe the mechanism to handle both these cases.

Initially, to reduce the store latency, we assume that the cache line is uncompressed, and issue a store to the lower levels of the memory hierarchy, while buffering a copy in L1. If the cache line is found in L2/memory in the uncompressed form (❶), the assumption was correct. The store then proceeds normally and the buffered stores are evicted from L1. If the assumption is incorrect, the cache line is retrieved (❼) and decompressed before the store is retransmitted to the lower levels of the memory hierarchy.

Realizing Data Compression. Supporting data compression requires additional support from the main memory controller and the runtime system, as we describe below.

Initial Setup and Profiling. Data compression with CABA requires a one-time data setup before the data is transferred to the GPU. We assume initial software-based data preparation where the input data is stored in CPU memory in the compressed form with an appropriate compression algorithm before transferring the data to GPU memory. Transferring data in the compressed form can also reduce PCIe bandwidth usage.[17]

Memory-bandwidth-limited GPU applications are the best candidates for employing data compression using CABA. The compiler (or the runtime profiler) is required to identify those applications that are most likely to benefit from this framework. For applications where memory bandwidth is not a bottleneck, data compression is simply disabled.

Memory Controller Changes. Data compression reduces off-chip bandwidth requirements by transferring the same data in fewer DRAM bursts. The memory controller (MC) needs to know whether the cache line data is compressed and how many bursts (1–4 bursts in GDDR5 [135]) are needed to transfer the data from DRAM to the MC. Similar to prior work [263, 290], we require metadata information for every cache line that keeps track of how many bursts are needed to transfer the data. Similar to prior work [290], we simply reserve 8MB of GPU DRAM space for the metadata (~0.2% of all available memory). Unfortunately, this simple design would require an additional access for the metadata for every access to DRAM effectively doubling the required bandwidth. To avoid this, a simple *metadata (MD) cache* that keeps frequently-accessed metadata on chip (near the MC) is required. Note that this metadata cache is similar to other metadata storage and caches proposed for various purposes in the memory controller, e.g., [113, 201, 224, 263, 269, 294]. Our experiments show that a small 8 KB 4-way associative MD cache is sufficient to provide a hit rate of 85% on average (more than 99% for many applications) across all applications in our workload pool.[18] Hence, in the common case, a second access to DRAM to fetch compression-related metadata can be avoided.

5.5 Use Cases

5.5.1 Memoization

Hardware memoization is a technique used to avoid redundant computations by reusing the results of previous computations that have the same or similar inputs. Prior work [17, 24, 282] observed redundancy in inputs to data in GPU workloads. In applications limited by available compute resources, memoization offers an opportunity to trade off computation for storage, thereby enabling potentially higher energy efficiency and performance. In order to realize memoization in

[17]This requires changes to the DMA engine to recognize compressed lines.

[18]For applications where MD cache miss rate is low, we observe that MD cache misses are usually also TLB misses. Hence, most of the overhead of MD cache misses in these applications is outweighed by the cost of page table lookups.

hardware, a look-up table (LUT) is required to dynamically cache the results of computations as well as the corresponding inputs. The granularity of computational reuse can be at the level of fragments [24], basic blocks, functions [16, 18, 76, 134, 305], or long-latency instructions [71]. The CABA framework is a natural way to implement such an optimization. The availability of on-chip memory lends itself for use as the LUT. In order to cache previous results in on-chip memory, look-up tags (similar to those proposed in [110]) are required to index correct results. With applications tolerant of approximate results (e.g., image processing, machine learning, fragment rendering kernels), the computational inputs can be hashed to reduce the size of the LUT. Register values, texture/constant memory or global memory sections that are not subject to change are potential inputs. An assist warp can be employed to perform memoization in the following way: (1) compute the hashed value for look-up at predefined trigger points, (2) use the load/store pipeline to save these inputs in available shared memory, and (3) eliminate redundant computations by loading the previously computed results in the case of a hit in the LUT.

5.5.2 Prefetching

Prefetching has been explored in the context of GPUs [23, 154, 155, 184, 188, 221, 298] with the goal of reducing effective memory latency. With memory-latency-bound applications, the load/store pipelines can be employed by the CABA framework to perform opportunistic prefetching into GPU caches. The CABA framework can potentially enable the effective use of prefetching in GPUs due to several reasons: (1) Even simple prefetchers such as the stream [157, 252, 310] or stride [31, 104] prefetchers are non-trivial to implement in GPUs since access patterns need to be tracked and trained at the granularity of warps [188, 298]. CABA could enable fine-grained book-keeping by using spare registers and assist warps to save metadata for each warp. The computational units could then be used to continuously compute strides in access patterns both within and across warps. (2) It has been demonstrated that software prefetching and helper threads [3, 51, 74, 136, 136, 184, 212, 318] are very effective in performing prefetching for irregular access patterns. Assist warps offer the hardware/software interface to implement application-specific prefetching algorithms with varying degrees of complexity without the additional cost of hardware implementation. (3) In bandwidth-constrained GPU systems, uncontrolled prefetching could potentially flood the off-chip buses, delaying demand requests. CABA can enable flexible prefetch throttling (e.g., [94, 95, 310]) by scheduling assist warps that perform prefetching, only when the memory pipelines are idle. (4) Prefetching with CABA entails using load or prefetch instructions, which not only enables prefetching to the hardware-managed caches, but also simplifies usage of unutilized shared memory or register file as prefetch buffers.

5.5.3 Redundant Multithreading

Reliability of GPUs is a key concern, especially today when they are popularly employed in many supercomputing systems. Ensuring hardware protection with dedicated resources can be expensive [214]. Redundant multithreading [232, 270, 341] is an approach where redundant threads are used to replicate program execution. The results are compared at different points in execution to detect and potentially correct errors. The CABA framework can be extended to redundantly execute portions of the original program via the use of such approaches to increase the reliability of GPU architectures.

5.5.4 Speculative Precomputation

In CPUs, speculative multithreading ([103, 217, 268]) has been proposed to speculatively parallelize serial code and verify the correctness later. Assist warps can be employed in GPU architectures to speculatively pre-execute sections of code during idle cycles to further improve parallelism in the program execution. Applications tolerant to approximate results could particularly be amenable towards this optimization [368].

5.5.5 Handling Interrupts and Exceptions.

Current GPUs do not implement support for interrupt handling except for some support for timer interrupts used for application time-slicing [245]. CABA offers a natural mechanism for associating architectural events with subroutines to be executed in throughput-oriented architectures where thousands of threads could be active at any given time. Interrupts and exceptions can be handled by special assist warps, without requiring complex context switching or heavy-weight kernel support.

5.5.6 Profiling and Instrumentation

Profiling and binary instrumentation tools like Pin [213] and Valgrind [239] proved to be very useful for development, performance analysis and debugging on modern CPU systems. At the same time, there is a lack [19] of tools with same/similar capabilities for modern GPUs. This significantly limits software development and debugging for modern GPU systems. The CABA framework can potentially enable easy and efficient development of such tools, as it is flexible enough to invoke user-defined code on specific architectural events (e.g., cache misses, control divergence).

[19]With the exception of one recent work [311].

5.6 Methodology

We model the CABA framework in GPGPU-Sim 3.2.1 [32]. Table 6 provides the major parameters of the simulated system. We use GPUWattch [192] to model GPU power and CACTI [322] to evaluate the power/energy overhead associated with the MD cache (Section 5.4.3) and the additional components (AWS and AWC) of the CABA framework. We implement BDI [264] using the Synopsys Design Compiler with 65nm library (to evaluate the energy overhead of compression/decompression for the dedicated hardware design for comparison to CABA), and then use ITRS projections [139] to scale our results to the 32nm technology node.

System Overview	15 SMs, 32 threads/warp, 6 memory channels
Shader Core Config	1.4GHz, GTO scheduler [274], 2 schedulers/SM
Resources / SM	48 warps/SM, 32768 registers, 32KB Shared Memory
L1 Cache	16KB, 4-way associative, LRU replacement policy
L2 Cache	768KB, 16-way associative, LRU replacement policy
Interconnect	1 crossbar/direction (15 SMs, 6 MCs), 1.4GHz
Memory Model	177.4GB/s BW, 6 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling, 16 banks/MC
GDDR5 Timing [135]	$t_{CL} = 12, : t_{RP} = 12, : t_{RC} = 40, : t_{RAS} = 28,$ $t_{RCD} = 12, : t_{RRD} = 6 : t_{CLDR} = 5 : t_{WR} = 12$

Table 6: Major parameters of the simulated systems.

Evaluated Applications. We use a number of CUDA applications derived from CUDA SDK [244] (*BFS, CONS, JPEG, LPS, MUM, RAY, SLA, TRA*), Rodinia [61] (*hs, nw*), Mars [127] (*KM, MM, PVC, PVR, SS*) and lonestar [52] (*bfs, bh, mst, sp, sssp*) suites. We run all applications to completion or for 1 billion instructions (whichever comes first). CABA-based data compression is beneficial mainly for memory-bandwidth-limited applications. In computation-resource limited applications, data compression is not only unrewarding, but it can also cause significant performance degradation due to the computational overheads associated with assist warps. We rely on static profiling to identify memory-bandwidth-limited applications and disable CABA-based compression for the others. In our evaluation (Section 5.7), we demonstrate detailed results for applications that exhibit some compressibility in memory bandwidth (at least 10%). Applications without compressible data (e.g., sc, SCP) do not gain any performance from the CABA framework, and we verified that these applications do not incur any performance degradation (because the assist warps are *not* triggered for them).

Evaluated Metrics. We present Instruction per Cycle (*IPC*) as the primary performance metric.

We also use *average bandwidth utilization*, defined as the fraction of total DRAM cycles that the DRAM data bus is busy, and *compression ratio*, defined as the ratio of the number of DRAM bursts required to transfer data in the compressed vs. uncompressed form. As reported in prior work [264], we use decompression/compression latencies of 1/5 cycles for the hardware implementation of BDI.

5.7 Results

To evaluate the effectiveness of using CABA to employ data compression, we compare five different designs: (i) *Base* - the baseline system with no compression, (ii) *HW-BDI-Mem* - hardware-based *memory bandwidth compression* with dedicated logic (data is stored compressed in main memory but uncompressed in the last-level cache, similar to prior works [263, 290]), (iii) *HW-BDI* - hardware-based *interconnect and memory bandwidth compression* (data is stored uncompressed only in the L1 cache) (iv) *CABA-BDI* - Core-Assisted Bottleneck Acceleration (CABA) framework (Section 5.3) with all associated overheads of performing compression (for both interconnect and memory bandwidth), (v) *Ideal-BDI* - compression (for both interconnect and memory) with no latency/power overheads for compression or decompression. This section provides our major results and analyses.

5.7.1 Effect on Performance and Bandwidth Utilization

Figures 50 and 51 show, respectively, the normalized performance (vs. *Base*) and the memory bandwidth utilization of the five designs. We make three major observations.

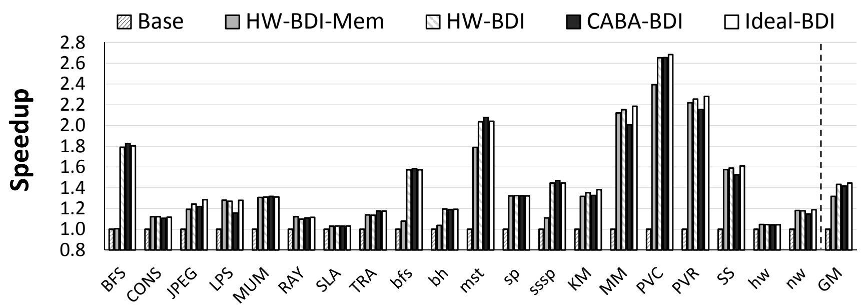


Figure 50: Normalized performance.

First, all compressed designs are effective in providing high performance improvement over the baseline. Our approach (CABA-BDI) provides a 41.7% average improvement, which is only 2.8% less than the ideal case (Ideal-BDI) with none of the overheads associated with CABA. CABA-BDI's performance is 9.9% better than the previous [290] hardware-based memory bandwidth compression design (HW-BDI-Mem), and *only* 1.6% worse than the purely hardware-based design (HW-BDI) that performs both interconnect and memory bandwidth compression. We conclude that

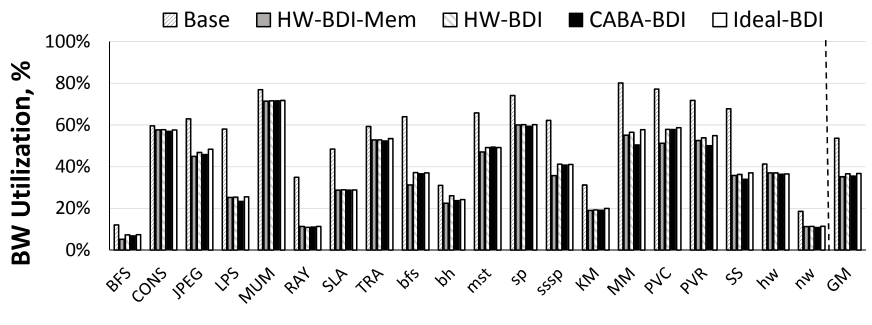


Figure 51: Memory bandwidth utilization.

our framework is effective at enabling the benefits of compression without requiring specialized hardware compression and decompression logic.

Second, performance benefits, in many workloads, correlate with the reduction in memory bandwidth utilization. For a fixed amount of data, compression reduces the bandwidth utilization, and, thus, increases the effective available bandwidth. Figure 51 shows that CABA-based compression 1) reduces the average memory bandwidth utilization from 53.6% to 35.6% and 2) is effective at alleviating the memory bandwidth bottleneck in most workloads. In some applications (e.g., *bfs* and *mst*), designs that compress *both* the on-chip interconnect and the memory bandwidth, i.e. CABA-BDI and HW-BDI, perform better than the design that compresses only the memory bandwidth (HW-BDI-Mem). Hence, CABA seamlessly enables the mitigation of the interconnect bandwidth bottleneck as well, since data compression/decompression is flexibly performed at the cores.

Third, for some applications, CABA-BDI performs slightly (within 3%) better than Ideal-BDI and HW-BDI. The reason for this counter-intuitive result is the effect of warp oversubscription [26, 163, 164, 274]. In these cases, too many warps execute in parallel, polluting the last level cache. CABA-BDI sometimes reduces pollution as a side effect of performing more computation in assist warps, which slows down the progress of the parent warps.

We conclude that the CABA framework can effectively enable data compression to reduce both on-chip interconnect and off-chip memory bandwidth utilization, thereby improving the performance of modern GPGPU applications.

5.7.2 Effect on Energy

Compression decreases energy consumption in two ways: 1) by reducing bus energy consumption, 2) by reducing execution time. Figure 52 shows the normalized energy consumption of the five systems. We model the static and dynamic energy of the cores, caches, DRAM, and all buses (both on-chip and off-chip), as well as the energy overheads related to compression: metadata (MD) cache and compression/decompression logic. We make two major observations.

First, CABA-BDI reduces energy consumption by as much as 22.2% over the baseline. This

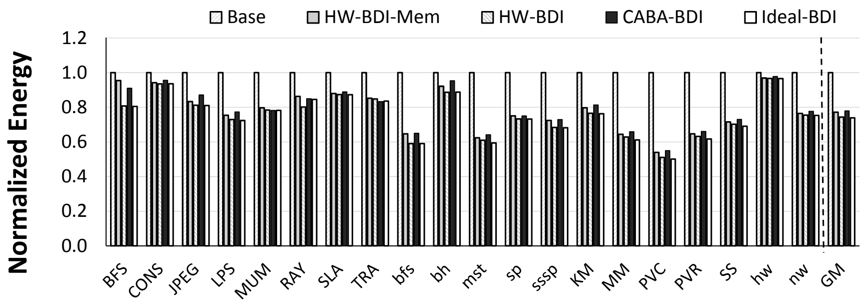


Figure 52: Normalized energy consumption.

is especially noticeable for memory-bandwidth-limited applications, e.g., *PVC*, *mst*. This is a result of two factors: (i) the reduction in the amount of data transferred between the LLC and DRAM (as a result of which we observe a 29.5% average reduction in DRAM power) and (ii) the reduction in total execution time. This observation agrees with several prior works on bandwidth compression [263, 300]. We conclude that the CABA framework is capable of reducing the overall system energy, primarily by decreasing the off-chip memory traffic.

Second, CABA-BDI's energy consumption is only 3.6% more than that of the HW-BDI design, which uses dedicated logic for memory bandwidth compression. It is also only 4.0% more than that of the Ideal-BDI design, which has no compression-related overheads. CABA-BDI consumes more energy because it schedules and executes assist warps, utilizing on-chip register files, memory and computation units, which is less energy-efficient than using dedicated logic for compression. However, as results indicate, this additional energy cost is small compared to the performance gains of CABA (recall, 41.7% over Base), and may be amortized by using CABA for other purposes as well (see Section 5.5).

Power Consumption. CABA-BDI increases the system power consumption by 2.9% over the baseline (not graphed), mainly due to the additional hardware and higher utilization of the compute pipelines. However, the power overhead enables energy savings by reducing bandwidth use and can be amortized across other uses of CABA (Section 5.5).

Energy-Delay product. Figure 53 shows the product of the normalized energy consumption and normalized execution time for the evaluated GPU workloads. This metric simultaneously captures two metrics of interest—energy dissipation and execution delay (inverse of performance). An optimal feature would simultaneously incur low energy overhead while also reducing the execution delay. This metric is useful in capturing the efficiencies of different architectural designs and features which may expend differing amounts of energy while producing the same performance speedup or vice-versa. Hence, a lower Energy-Delay product is more desirable. We observe that CABA-BDI has a 45% lower Energy-Delay product than the baseline. This reduction comes from energy savings from reduced data transfers as well as lower execution time. On average, CABA-BDI is within only 4% of Ideal-BDI which incurs none of the energy and performance overheads of the CABA framework.

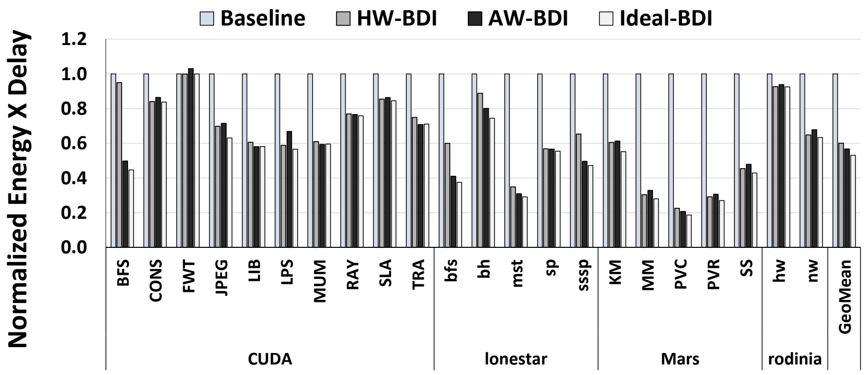


Figure 53: Energy-Delay product.

5.7.3 Effect of Enabling Different Compression Algorithms

The CABA framework is *not limited to a single compression algorithm*, and can be effectively used to employ other hardware-based compression algorithms (e.g., FPC [13] and C-Pack [66]). The effectiveness of other algorithms depends on two key factors: (i) how efficiently the algorithm maps to GPU instructions, (ii) how compressible the data is with the algorithm. We map the FPC and C-Pack algorithms to the CABA framework and evaluate the framework's efficacy.

Figure 54 shows the normalized speedup with four versions of our design: *CABA-FPC*, *CABA-BDI*, *CABA-C-Pack*, and *CABA-BestOfAll* with the FPC, BDI, C-Pack compression algorithms. CABA-BestOfAll is an idealized design that selects and uses the best of all three algorithms in terms of compression ratio for *each cache line*, assuming no selection overhead. We make three major observations.

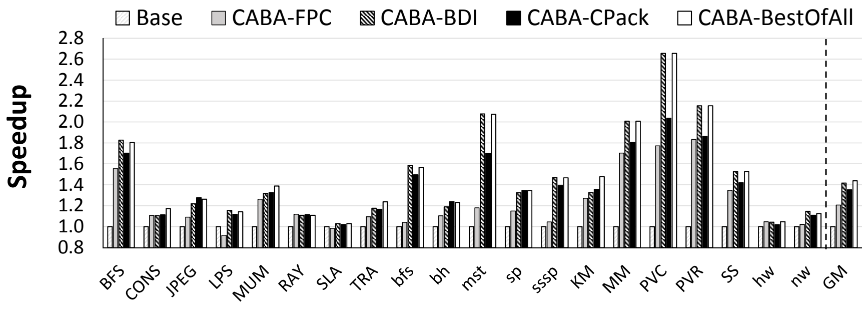


Figure 54: Speedup with different compression algorithms.

First, CABA significantly improves performance with any compression algorithm (20.7% with FPC, 35.2% with C-Pack). Similar to CABA-BDI, the applications that benefit the most are those that are both memory-bandwidth-sensitive (Figure 51) and compressible (Figure 55). We conclude that our proposed framework, CABA, is general and flexible enough to successfully enable different compression algorithms.

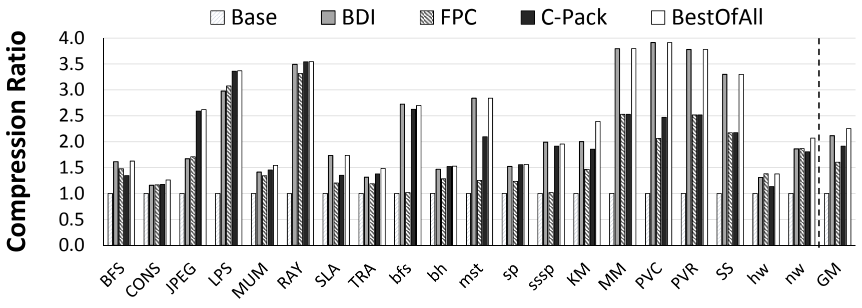


Figure 55: Compression ratio of algorithms with CABA.

Second, applications benefit differently from each algorithm. For example, *LPS, JPEG, MUM, nw* have higher compression ratios with FPC or C-Pack, whereas *MM, PVC, PVR* compress better with BDI. This motivates the necessity of having *flexible data compression* with different algorithms within the same system. Implementing multiple compression algorithms completely in hardware is expensive as it adds significant area overhead, whereas CABA can flexibly enable the use of different algorithms via its general assist warp framework.

Third, the design with the best of three compression algorithms, CABA-BestOfAll, can sometimes improve performance more than each individual design with just one compression algorithm (e.g., for *MUM* and *KM*). This happens because even within an application, different cache lines compress better with different algorithms. At the same time, different compression related overheads of different algorithms can cause one to have higher performance than another even though the latter may have a higher compression ratio. For example, CABA-BDI provides higher performance on *LPS* than CABA-FPC, even though BDI has a lower compression ratio than FPC for *LPS*, because BDI's compression/decompression latencies are much lower than FPC's. Hence, a mechanism that selects the best compression algorithm based on *both* compression ratio and the relative cost of compression/decompression is desirable to get the best of multiple compression algorithms. The CABA framework can flexibly enable the implementation of such a mechanism, whose design we leave for future work.

5.7.4 Sensitivity to Peak Main Memory Bandwidth

As described in Section 5.2, main memory (off-chip) bandwidth is a major bottleneck in GPU applications. In order to confirm that CABA works for different designs with varying amounts of available memory bandwidth, we conduct an experiment where CABA-BDI is used in three systems with 0.5X, 1X and 2X amount of bandwidth of the baseline.

Figure 56 shows the results of this experiment. We observe that, as expected, each CABA design (**-CABA*) significantly outperforms the corresponding baseline designs with the same amount of bandwidth. The performance improvement of CABA is often equivalent to the doubling the off-chip memory bandwidth. We conclude that CABA-based bandwidth compression, on average, offers almost all the performance benefit of doubling the available off-chip bandwidth with only modest complexity to support assist warps.

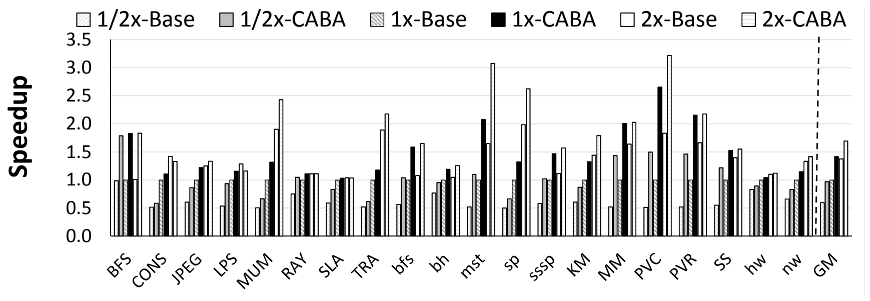


Figure 56: Sensitivity of CABA to memory bandwidth.

5.7.5 Selective Cache Compression with CABA

In addition to reducing bandwidth consumption, data compression can also increase the *effective capacity* of on-chip caches. While compressed caches can be beneficial—as higher effective cache capacity leads to lower miss rates—supporting cache compression requires several changes in the cache design [13, 66, 262, 264, 287].

Figure 57 shows the effect of four cache compression designs using CABA-BDI (applied to both L1 and L2 caches with 2x or 4x the number of tags of the baseline[20]) on performance. We make two major observations. First, several applications from our workload pool are not only bandwidth sensitive, but also cache capacity sensitive. For example, *bfs* and *sssp* significantly benefit from L1 cache compression, while *TRA* and *KM* benefit from L2 compression. Second, L1 cache compression can severely degrade the performance of some applications, e.g., *hw* and *LPS*. The reason for this is the overhead of decompression, which can be especially high for L1 caches as they are accessed very frequently. This overhead can be easily avoided by disabling compression at any level of the memory hierarchy.

5.7.6 Other Optimizations

We also consider several other optimizations of the CABA framework for data compression: (i) avoiding the overhead of decompression in L2 by storing data in the uncompressed form and (ii) optimized load of *only useful* data.

Uncompressed L2. The CABA framework allows us to store compressed data selectively at different levels of the memory hierarchy. We consider an optimization where we avoid the

[20]The number of tags limits the effective compressed cache size [13, 264].

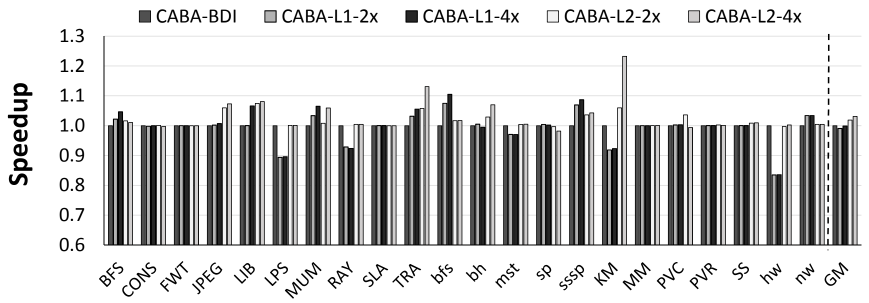


Figure 57: Speedup of cache compression with CABA.

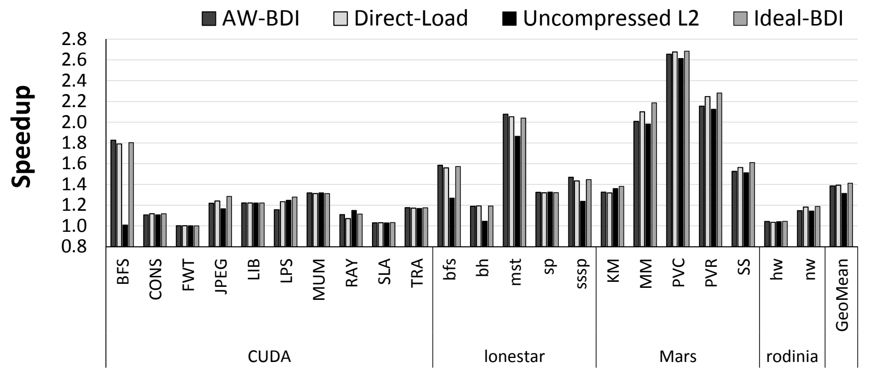


Figure 58: Effect of different optimizations (Uncompressed data in L2 and Direct Load) on applications' performance.

overhead of decompressing data in L2 by storing data in uncompressed form. This provides another tradeoff between the savings in on-chip traffic (when data in L2 is compressed – default option), and savings in decompression latency (when data in L2 is uncompressed). Figure 58 depicts the performance benefits from this optimization. Several applications in our workload pool (e.g., *RAY*) benefit from storing data uncompressed as these applications have high hit rates in the L2 cache. We conclude that offering the choice of enabling or disabling compression at different levels of the memory hierarchy can provide higher levels of the software stack (e.g., applications, compilers, runtime system, system software) with an additional performance knob.

Uncoalesced requests. Accesses by scalar threads from the same warp are coalesced into fewer memory transactions [242]. If the requests from different threads within a warp span two or more cache lines, multiple lines have to be retrieved and decompressed before the warp can proceed its execution. Uncoalesced requests can significantly increase the number of assist warps that need to be executed. An alternative to decompressing each cache line (when only a few bytes from each line may be required), is to enhance the coalescing unit to supply only the correct *deltas* from within each compressed cache line. The logic that maps bytes within a cache line to the appropriate registers will need to be enhanced to take into account the encoding of the compressed line to determine the size of the *base* and the *deltas*. As a result, we do not decompress the entire cache lines and only extract the data that is needed. In this case, the cache line is not inserted into the L1D cache in the uncompressed form, and hence every line needs to be decompressed even if it is found in the L1D cache.[21] *Direct-Load* in Figure 58 depicts the performance impact from this optimization. The overall performance improvement is 2.5% on average across all applications (as high as 4.6% for *MM*).

5.8 Related Work

To our knowledge, this work is the first to (1) propose a flexible and general cross-layer abstraction and framework for employing idle GPU resources for useful computation that can aid regular program execution, and (2) use the general concept of *helper threading* to perform memory and interconnect bandwidth compression. We demonstrate the benefits of our new framework by using it to implement multiple compression algorithms on a throughput-oriented GPU architecture. We briefly discuss related works in helper threading and bandwidth compression.

Helper Threading. Previous works [3, 51, 58, 59, 73, 74, 90, 91, 136, 159, 170, 211, 212, 318, 378, 387] demonstrated the use of *helper threads* in the context of Simultaneous Multithreading (SMT) and multi-core processors, primarily to speed up single-thread execution by using idle SMT contexts or idle cores in CPUs. These works typically use helper threads (generated by the software, the hardware, or cooperatively) to pre-compute useful information that aids the execution of the primary thread (e.g., by prefetching, branch outcome pre-computation, and cache management). No previous work discussed the use of helper threads for memory/interconnect bandwidth compression or cache compression.

While our work was inspired by these prior studies of helper threading in latency-oriented architectures (CPUs), developing a framework for helper threading (or *assist warps*) in throughput-oriented architectures (GPUs) enables new opportunities and poses new challenges, both due to the massive parallelism and resources present in a throughput-oriented architecture. Our CABA framework exploits these new opportunities and addresses these new challenges, including (1) low-cost management of dozens of assist warps that could be running concurrently with regular program warps, (2) means of state/context management and scheduling for assist warps to maximize effectiveness and minimize interference, and (3) different possible applications of the concept of assist warps in a throughput-oriented architecture.

In the GPU domain, CudaDMA [34] is a recent proposal that aims to ease programmability by decoupling execution and memory transfers with specialized DMA warps. This work does not provide a general and flexible hardware-based framework for using GPU cores to run warps that aid the main program.

Compression. Several prior works [15, 22, 262, 263, 290, 300, 323] study memory and cache

[21]This optimization also benefits cache lines that might *not* have many uncoalesced accesses, but have poor data reuse in the L1D.

compression with several different compression algorithms [13,22,66,138,264,364], in the context of CPUs or GPUs. Our work is the first to demonstrate how one can adapt some of these algorithms for use in a general helper threading framework for GPUs. As such, compression/decompression using our new framework is more flexible since it does not require a specialized hardware implementation for any algorithm and instead utilizes the existing GPU core resources to perform compression and decompression. Finally, assist warps are applicable beyond compression and can be used for other purposes.

5.9 Summary

This work makes a case for the Core-Assisted Bottleneck Acceleration (CABA) framework, which employs *assist warps* to alleviate different bottlenecks in GPU execution. CABA is based on the key observation that various imbalances and bottlenecks in GPU execution leave on-chip resources, i.e., computational units, register files and on-chip memory, underutilized. CABA takes advantage of these idle resources and employs them to perform useful work that can aid the execution of the main program and the system.

We provide a detailed design and analysis of how CABA can be used to perform flexible data compression in GPUs to mitigate the memory bandwidth bottleneck. Our extensive evaluations across a variety of workloads and system configurations show that the use of CABA for memory compression significantly improves system performance (by 41.7% on average on a set of bandwidth-sensitive GPU applications) by reducing the memory bandwidth requirements of both the on-chip and off-chip buses.

We conclude that CABA is a general substrate that can alleviate the memory bandwidth bottleneck in modern GPU systems by enabling flexible implementations of data compression algorithms. We believe CABA is a general framework that can have a wide set of use cases to mitigate many different system bottlenecks in throughput-oriented architectures, and we hope that future work explores both new uses of CABA and more efficient implementations of it.

Chapter 6

Conclusions and Future Work

6.1 Future Work

This dissertation opens new avenues for research. In this section, we describe several such research directions in which the ideas and approaches in this thesis can be extended to address other challenges in programmability, portability, and efficiency in various systems.

6.1.1 Enabling rapid and fine-grain code adaptation at runtime, driven by hardware

As we increasingly rely on clouds and other virtualized environments, co-running applications, unexpected contention, and *lack of visibility* into available hardware resources make software optimization and dynamic recompilation limited in effectiveness. The hardware today cannot easily help address this challenge since the existing hardware-software contract requires that hardware rigidly execute the application as defined by software.

With the abstractions proposed in this dissertation, we cannot adapt the code itself, but only the underlying system and hardware based on the program properties. Future work would involve enabling the system/hardware to dynamically change computation depending on availability of resources and runtime program behavior. The idea is to have the application only convey higher-level functionality, and then enable a *codesigned* hardware-software system to dynamically change the implementation as the program executes. The benefit of enabling such capability in hardware is greater efficiency in adapting software and more fine-grain visibility into dynamic hardware state and application behavior. For example, the program describes a potentially sparse computation (e.g., sparse matrix-vector multiply). The hardware then dynamically elides computation when it detects zero inputs. Other examples include changing graph traversal algorithms to maximize data locality at runtime or altering the implementation of a forward pass in each neural network layer, based on resource availability and contention. The challenge is in designing a *general* hardware-software system that enables many such runtime optimizations and integrates flexibly with frameworks such as Halide [271], TensorFlow [4], and Spark [375].

The approach would be to design a clearly defined hardware-software abstraction that expresses what computation can be changed based on runtime information (resource availability/bottleneck). This abstraction should integrate well into common building-block operations of important applications to obtain generality. Another approach is to determine how to effectively abstract and communicate fine-grained dynamic hardware state, bottlenecks, and contention, to enable software frameworks, databases, and other software systems to dynamically adapt applications accordingly.

6.1.2 Widening the scope of cross-layer and full stack optimizations

This dissertation has demonstrated the significant *performance, portability,* and *productivity* benefits of cross-layer abstractions that are carefully architected to enable *full-system coordination and communication* to achieve these goals, from the programming model, compiler and OS, to each hardware component (cache, memory, storage, and so on).

Future work would investigate enabling full-system coordination and communication to achieve other challenges such as security, quality-of-service (QoS), meeting service-level objectives (SLOs), and reliability. The challenge is in designing cross-layer abstractions and interfaces that enable very disparate components (e.g., cache, storage, OS thread scheduler) to communicate and coordinate, both horizontally and vertically in the computing stack, to meet the same goal

The first steps would involve looking into enhancing the existing compute stack to enable these holistic designs and cross-layer optimizations. This will involve determining how to express application-level requirements for these goals, how to design low-overhead interfaces to communicate these requirements to the system and each hardware component, and then enhance the system accordingly to achieve the desired goal. Insights from this thesis may be directly applicable to solutions here.

The long-term research goal is to research *clean-slate approaches* to building systems with modularized components that are designed to provide full-system guarantees for performance isolation, predictability, reliability, and security. The idea is to compose the overall system from smaller modules where each module or *smallest unit* is designed to provide some guarantee of (for example) strict performance isolation. Similarly the interfaces between module should preserve the guarantees for the overall system. Initial research questions: What are the semantics that define a module? What are the semantics that define interactions between modules?

6.1.3 Integrating new technologies, compute paradigms, and specialized systems

Future systems will incorporate a diverse set of technologies and specialized chips, that will rapidly evolve. This poses many new challenges across the computing stack in the *integration* of new technologies, such as memristors, persistant memory, and optical devices, and new paradigms, such as quantum computing, reconfigurable fabrics, application-specific hardware, and processing-in-memory substrates. Below are extensions to this thesis, relating to different aspects of this problem.

Enabling applications to automatically leverage new systems and architectures (*a software approach*). Automatically generating high-performance code for any new hardware accelerator/specialization today, without fully rewriting applications, is a challenging task. Software libraries have been demonstrated to be very inefficient and are not general. Automatic code generating frameworks and optimizing compilers (e.g., [65, 182, 316]) are promising approaches to target a changing set of architectures, without rewriting application code. However, such tool chains use a common intermediate representation (IR) to summarize the application and then use *specialized backends* that optimize the IR according to the characteristics of the new architecture and produce target code. Integration of new architectures or even new instructions/functionality in CPUs/GPUs (e.g., to add instructions to leverage a processing-in-memory accelerator) into these systems takes significant time, effort, and cross-layer expertise.

The goal is to develop frameworks to enable automatic generation of high performance code for specialized/new hardware, without rewriting existing applications. This would significant reduce the effort required to evaluate and deploy new hardware innovations including new architectures, substrates, or finer granularity hardware primitives (e.g., a faster in-memory operation).

The first steps would be to design an abstraction that enables easy and flexible description of the performance characteristics, constraints, and the *semantics* of the interface to hardware, i.e., the functionality implemented by the instruction-set architecture (ISA) or new primitive. Next, develop tools that enable automatic integration into existing compiler IRs to generate backends based on the description. Existing IRs should also be enhanced to capture more semantic content if required.

A longer-term goal is to develop hardware-software frameworks that enable flexible evaluation and design space exploration of how to abstract new hardware technologies and substrates in terms of the overall programmability (how many applications can leverage the technology), portability (how easily we can enhance the architecture without changing the interface), and performance (efficiency of the new architecture).

Enabling software-transparent hardware specialization and reconfiguration (*a hardware approach*). Today, even within general-purpose cores, architects are turning to different forms of hardware customization and reconfigurability for important computation as a means to drive improvements in performance and energy efficiency. Examples include Tensor Cores within GPUs to speed up tensor operations in machine learning, accelerating data intensive computation using processing-in-memory technologies, or reconfigurable fabrics (e.g., CGRAs) to accelerate important computations.

The goal is to enable seamless integration of specialization and reconfigurability into general-purposes architectures, *transparently* to the software stack. This would enable specialized designs to continuously evolve without creating new primitives/instructions each time (and hence no recompilation/rewriting is required). This addresses the critical portability and compatibility challenges associated with these approaches. For example, this would enable seamless addition of specialized/reconfigurable hardware support within general-purpose cores for: sparse and irregular computations (e.g, graph processing), managed languages (e.g., support for object-based programming, garbage collection), critical computations (e.g, numerical loops in machine learning) and frequent operations (e.g, queries in databases), among very many possibilities. Abstractions proposed in this dissertation, such as Expressive Memory, are not sufficiently rich to include specialized *computation* as opposed to just memory access.

The first steps would be to enhance general-purpose architectures to *(1)* allow flexible customization and reconfiguration of different components such as the compute units, memory hierarchy, and coherence protocols; and *(2)* enable seamless transition between general-purpose and specialized computation, and enable safe reconfiguration of hardware components at runtime. The next steps would be to design a rich (and future-proof) programming abstraction that captures sufficient application information to enable this flexible customization and runtime reconfiguration.

6.2 Conclusions

In this dissertation, we observed that the *interfaces* and abstractions between the layers of the computing stack—specifically, the hardware-software interface—significantly constrain the ability of the hardware architecture to intelligently and efficiently manage key resources in CPUs and GPUs. This leads to challenges in programmability and portability, as the application software is forced to do much of the heavy lifting in optimizing the code for performance. It also leaves significant performance on the table, as the application has little access to key resources and architectural mechanisms in hardware, and little visibility into available resources in the presence of virtualization or co-running applications.

We proposed rich low-overhead cross-layer abstractions that communicate higher-level program information from the application to the underlying system software and hardware architecture. These abstractions enable a wide range of hardware-software cooperative mechanisms to optimize for performance by managing critical resources more efficiently and intelligently. More efficient resource management at the hardware and system-level makes performance less sensitive to how well an application is optimized for the underlying architecture. This reduces the burden on the application developer and makes performance more portable across architecture generations. We demonstrated how such cross-layer abstractions can be designed to be *general*, enabling a wide range of hardware-software mechanisms, and *practical*, requiring only low overhead additions to existing systems and interfaces. Using 4 different contexts in CPUs and GPUs, we validate the thesis: *a rich low-overhead cross-layer interface that communicates higher-level application* *information to hardware enables many hardware-software cooperative mechanisms that significantly improve performance, portability, and programmability.*

First, we proposed Expressive Memory, a rich cross-layer interface in CPUs to communicate higher-level semantics of data structures and their access semantics to the operating system and hardware. We demonstrated its effectiveness in improving the portability of memory system optimizations and in enabling a wide range of cross-layer optimizations to improve memory system performance. Second, we introduced the Locality Descriptor, a cross-layer abstraction in GPUs that enables expressing and exploiting data locality in GPU programs. We demonstrated significant performance benefits by enabling the hardware to leverage knowledge of data locality properties, while reducing programming effort when optimizing for data locality. Third, we proposed Zorua, a framework that decouples GPU programming models from hardware resource management. We demonstrated how Zorua enhances programmability, portability, and performance, by enabling hardware to dynamically manage resources based on the program requirements. Finally, we introduced the Assist Warp abstraction in GPUs to effectively leverage idle memory and compute bandwidth to perform useful work. We hope that the ideas, analyses, and techniques presented in this dissertation can be extended to address challenges in hardware-software codesign and the design of cross-layer interfaces in future computer systems.

References

[1] Memory management optimizations on the Intel® Xeon Phi™ coprocessor. In `https://software.intel.com/sites/default/files/managed/b4/24/mem_management_dgemm.pdf`. Intel Compiler Lab, 2015.

[2] μC-States: Fine-grained GPU Datapath Power Management. In *PACT*, 2016.

[3] Tor M. Aamodt, Paul Chow, Per Hammarlund, Hong Wang, and John P. Shen. Hardware support for prescient instruction prefetch. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 84–, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. OSDI'16, 2016.

[5] Bulent Abali, Hubertus Franke, Dan E. Poff, Robert A. Saccone Jr., Charles O. Schulz, Lorraine M. Herger, and T. Basil Smith. Memory Expansion Technology (MXT): Software Support and Performance. *IBM J.R.D.*, 2001.

[6] Mohammad Abdel-Majeed and Murali Annavaram. Warped register file: A power efficient register file for gpgpus. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 412–423, Washington, DC, USA, 2013. IEEE Computer Society.

[7] Advanced Micro Devices, Inc. AMD Accelerated Parallel Processing OpenCL Programming Guide, 2011.

[8] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W Keckler, and Thomas F Wenisch. Unlocking bandwidth for GPUs in CC-NUMA systems. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[9] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. ASPLOS '15, 2015.

[10] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, 2015.

[11] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *ISCA*, 2015.

[12] Berkin Akin, Franz Franchetti, and James C Hoe. Data Reorganization in Memory Using 3D-Stacked DRAM. In *ISCA*, 2015.

[13] Alaa R. Alameldeen and David A. Wood. Adaptive Cache Compression for High-Performance Processors. In *ISCA-31*, 2004.

[14] Alaa R. Alameldeen and David A. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. *Tech. Rep.*, 2004.

[15] Alaa R. Alameldeen and David A. Wood. Interactions between compression and prefetching in chip multiprocessors. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 0:228–239, 2007.

[16] Carlos Álvarez, Jesús Corbal, Esther Salamí, and Mateo Valero. On the potential of tolerant region reuse for multimedia applications. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 218–228, New York, NY, USA, 2001. ACM.

[17] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7):922–927, July 2005.

[18] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Dynamic tolerance region computing for multimedia. *IEEE Trans. Comput.*, 61(5):650–665, May 2012.

[19] AMD. AMD-V nested paging.

[20] AMD. Amd accelerated parallel processing opencl programming guide. In *www.developet.amd.com/GPU/AMDAPPSDK*, 2011.

[21] Gene M Amdahl, Gerrit A Blaauw, and FP Brooks. Architecture of the IBM System/360. *IBM JRD*, 1964.

[22] Angelos Arelakis and Per Stenstrom. SC2: A statistical compression cache scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA, 2014.

[23] José-María Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Boosting mobile gpu performance with a decoupled access/execute fragment processor. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, 2012.

[24] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. *SIGARCH Comput. Archit. News*, 42(3):529–540, June 2014.

[25] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[26] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. *PACT*, 2015.

[27] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.

[28] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayiran, Gabriel H Loh, Chita R Das, Mahmut T Kandemir, and Onur Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *PACT*, 2015.

[29] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. *MICRO*, 2017.

[30] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. *ASPLOS*, 2018.

[31] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, May 1995.

[32] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pages 163–174, April 2009.

[33] Bin Bao and Chen Ding. Defensive loop tiling for shared cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[34] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[35] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2012.

[36] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Structure slicing: Extending logical regions with fields. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[37] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[38] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[39] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2013.

[40] Gordon Bell, A. Kotok, Thomas N. Hastings, and Robert A Hill. The Evolution of the DEC System 10. *CACM*, 1978.

[41] Kristof Beyls and Erik D'Hollander. Compile-time cache hint generation for EPIC architectures. In *2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2)*, 2002.

[42] Kristof Beyls and Erik H D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 2005.

[43] Rishiraj A. Bheda, Thomas M. Conte, and Jeffrey S. Vetter. Improving DRAM bandwidth utilization with MLP-Aware OS paging. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS, 2016.

[44] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2006.

[45] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *ASPLOS*, 1995.

[46] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[47] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, 2008.

[48] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*, 2018.

[49] Edward Brekelbaum, Jeff Rupley, Chris Wilkerson, and Bryan Black. Hierarchical Scheduling Windows. In *MICRO*, 2002.

[50] Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding. Pacman: Program-assisted cache management. In *ISMM*, 2013.

[51] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. Speculative precomputation on chip multiprocessors. In *In Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.

[52] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *IISWC*, 2012.

[53] J. Cabezas, L. Vilanova, I. Geladeno, T. B. Jablin, N. Navarro, and W. m. Hwu. Automatic execution of single-GPU computations across multiple GPUs. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2014.

[54] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B. Jablin, Nacho Navarro, and Wen-mei W. Hwu. Automatic parallelization of kernels in shared-memory multi-GPU nodes. In *International Conference on Supercomputing (ICS)*, 2015.

[55] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a smarter memory controller. In *Proceedings. Fifth International Symposium On High Performance Computer Architecture (HPCA)*, 1999.

[56] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 2007.

[57] Kevin K Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA*, 2016.

[58] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 186–195, Washington, DC, USA, 1999. IEEE Computer Society.

[59] Robert S. Chappell, Francis Tseng, Adi Yoaz, and Yale N. Patt. Microarchitectural support for precomputation microthreads. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 74–84, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[60] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[61] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[62] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. Porple: An extensible optimizer for portable data placement on gpu. In *MICRO*, 2014.

[63] Li-Jhan Chen, Hsiang-Yun Cheng, Po-Han Wang, and Chia-Lin Yang. Improving GPGPU performance via cache locality aware thread block scheduling. *IEEE Computer Architecture Letters (CAL)*, 2017.

[64] T. F. Chen. An effective programmable prefetch engine for on-chip caches. In *MICRO*, 1995.

[65] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: an automated end-to-end optimizing compiler for deep learning. OSDI, 2018.

[66] Xi Chen, Lei Yang, R.P. Dick, Li Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. In *VLSI Systems, IEEE Transactions on*, volume 18, pages 1196 –1208, Aug. 2010.

[67] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient GPU computing. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[68] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. Application-specific memory management for embedded systems using software-controlled caches. DAC, 2000.

[69] T. Chiueh. Sunder: a programmable hardware prefetch architecture for numerical loops. In *SC*, 1994.

[70] Hyojin Choi, Jaewoo Ahn, and Wonyong Sung. Reducing off-chip memory traffic by selective cache management scheme in GPGPUs. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2012.

[71] Daniel Citron, Dror Feitelson, and Larry Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 252–261, New York, NY, USA, 1998. ACM.

[72] Stephanie Coleman and Kathryn S McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Notices*, 1995.

[73] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 306–317, Washington, DC, USA, 2001. IEEE Computer Society.

[74] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. *SIGARCH Comput. Archit. News*, 29(2):14–25, May 2001.

[75] Jason Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman, and Yi Zou. An energy-efficient adaptive hybrid cache. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, 2011.

[76] D.A Conners and W.-M.W. Hwu. Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO-32*, pages 158–169, 1999.

[77] Henry Cook, Krste Asanović, and David A. Patterson. Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. Technical report, EECS Department, University of California, Berkeley, Sep 2009.

[78] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, 2002.

[79] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM JRD*, 1981.

[80] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. S. Yousif, and C. R. Das. Performance and Power Optimization through Data Compression in Network-on-Chip Architectures. In *HPCA*, 2008.

[81] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2013.

[82] Andrew Davidson and John Owens. Toward techniques for auto-tuning GPU algorithms. In *International Workshop on Applied Parallel Computing*, 2010.

[83] Davide De Caro, Nicola Petra, and Antonio G. M. Strollo. High-performance special function unit for programmable 3-d graphics processors. *Trans. Cir. Sys. Part I*, 56(9):1968–1978, September 2009.

[84] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 2013.

[85] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 1970.

[86] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. *ASPLOS*, 2015.

[87] Wei Ding, Diana Guttman, and Mahmut Kandemir. Compiler support for optimizing memory bank-level parallelism. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[88] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *GPGPU 2007*, 2007.

[89] Yuri Dotsenko, Sara S. Baghsorkhi, Brandon Lloyd, and Naga K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. *SIGPLAN Not.*

[90] M. Dubois. Fighting the memory wall with assisted execution. In *CF*, 2004.

[91] Michel Dubois, Michel Dubois, Yong Ho Song, and Yong Ho Song. Assisted execution. Technical report, USC, 1998.

[92] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys, 2016.

[93] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[94] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Jose A Joao, Onur Mutlu, and Yale N Patt. Prefetch-aware shared resource management for multi-core systems. *ISCA*, 2011.

[95] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *MICRO*, 2009.

[96] Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[97] Magnus Ekman and Per Stenstrom. A Robust Main-Memory Compression Scheme. In *ISCA*, 2005.

[98] Mattan Erez, Brian P Towles, and William J Dally. Spills, Fills, and Kills - An Architecture for Reducing Register-Memory Traffic. Technical Report TR-23, Stanford Univ., Concurrent VLSI Architecture Group, 2000.

[99] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, 2012.

[100] Carlos Flores Fajardo, Zhen Fang, Ravi Iyer, German Fabila Garcia, Seung Eun Lee, and Li Zhao. Buffer-integrated-cache: A cost-effective sram architecture for handheld and embedded platforms. DAC '11, 2011.

[101] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Conference on High Performance Networking and Computing (SC)*, 2006.

[102] E. A. Feustel. On the advantages of tagged architecture. *TC*, 1973.

[103] Manoj Franklin. *Multiscalar Processors*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[104] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 102–110, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[105] Adi Fuchs, Shie Mannor, Uri Weiser, and Yoav Etsion. Loop-aware memory prefetching using code block working sets. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO-47, 2014.

[106] Jayesh Gaur, Raghuram Srinivasan, Sreenivas Subramoney, and Mainak Chaudhuri. Efficient management of last-level caches in graphics processors for 3d scene rendering workloads. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[107] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. *SIGARCH Comput. Archit. News*, 39(3):235–246, June 2011.

[108] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. A hierarchical thread scheduler and register file for energy-efficient throughput processors. 2012.

[109] Mark Gebhart, Stephen W. Keckler, and William J. Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 465–476, New York, NY, USA, 2011. ACM.

[110] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 96–106, Washington, DC, USA, 2012. IEEE Computer Society.

[111] Mohsen Ghasempour, Aamer Jaleel, Jim D Garside, and Mikel Luján. Dream: Dynamic re-arrangement of address mapping to improve the performance of DRAMs. In *Proceedings of the Second International Symposium on Memory Systems*, 2016.

[112] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. arxiv:1802.00320 [cs.AR], 2018.

[113] Mrinmoy Ghosh and Hsien-Hsin S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. MICRO-40, 2007.

[114] A. Glew. MLP yes! ILP no. *ASPLOS WACI*, 1998.

[115] Xiang Gong, Zhongliang Chen, Amir Kavyan Ziabari, Rafael Ubal, and David Kaeli. Twinkernels: an execution model to improve gpu hardware scheduling at compile time. In *International Symposium on Code Generation and Optimization (CGO)*, 2017.

[116] Antonio María González Colás, José González González, and Mateo Valero Cortés. Virtual-physical registers. In *HPCA*, 1998.

[117] Edward H Gornish and Alexander Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *International Conference on Parallel Processing (ICPP)*, 1994.

[118] GPGPU-Sim v3.2.1. GPGPU-Sim Manual.

[119] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *HotPar*, 2012.

[120] Xiaoming Gu, Tongxin Bai, Yaoqing Gao, Chengliang Zhang, Roch Archambault, and Chen Ding. P-OPT: program-directed optimal cache management. LCPC, 2008.

[121] P. H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM JRD*, 1983.

[122] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, Maria J. Garzaran, and David Padua. Programming with tiles. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2008.

[123] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C Hoe, and Franz Franchetti. 3D-Stacked Memory-Side Acceleration: Accelerator and System Design. In *WoNDP*, 2014.

[124] T. D. Han and T. S. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[125] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. *ACM SIGARCH Computer Architecture News*, 37(3):184–195, 2009.

[126] Ari B. Hayes and Eddy Z. Zhang. Unified on-chip memory allocation for simt architecture. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, 2014.

[127] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.

[128] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013.

[129] Marius Hillenbrand, Mathias Gottschlag, Jens Kehne, and Frank Bellosa. Multiple physical mappings: Dynamic DRAM channel sharing and partitioning. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017.

[130] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. *ASPLOS*, March 2011.

[131] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[132] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *NSDI*, 2016.

[133] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.

[134] Jian Huang and David Lilja. Exploiting basic block value locality with block reuse. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 106–, Washington, DC, USA, 1999. IEEE Computer Society.

[135] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0.

[136] Khaled Z. Ibrahim, Gregory T. Byrd, and Eric Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 179–, Washington, DC, USA, 2003. IEEE Computer Society.

[137] Intel Corporation. Enabling Intel virtualization technology features and benefits.

[138] Mafijul Islam, Sally A McKee, and Per Stenström. Zero-value caches: Cancelling loads that return zero. Technical report, 2009. 18.

[139] ITRS. International technology roadmap for semiconductors. 2011.

[140] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 1998.

[141] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted cache replacement mechanisms for embedded systems. ICCAD, 2001.

[142] Prabhat Jain, Srini Devadas, and Larry Rudolph. Controlling cache pollution in prefetching with software-assisted cache replacement. *Comptation Structures Group, Laboratory for Computer Science CSG Memo*, 2001.

[143] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *ISCA-37*, 2010.

[144] Hyeran Jeon, Gunjae Koo, and Murali Annavaram. CTA-aware prefetching for GPGPU. *Computer Engineering Technical Report Number CENG-2014-08*, 2014.

[145] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. Gpu register file virtualization. In *MICRO*, 2015.

[146] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[147] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[148] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[149] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *International Conference on Supercomputing (ICS)*, 2012.

[150] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010.

[151] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Bottleneck Identification and Scheduling in Multithreaded Applications. In *ASPLOS*, 2012.

[152] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Utility-based Acceleration of Multithreaded Applications on Asymmetric CMPs. In *ISCA*, 2013.

[153] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *GPGPU*, 2014.

[154] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[155] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Orchestrated scheduling and prefetching for GPGPUs. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[156] Adwait Jog, Onur Kayiran, Nachiappan C. Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.

[157] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM.

[158] J. C. Juega, J. I. Gomez, C. Tenllado, and F. Catthoor. Adaptive mapping and parameter selection scheme to improve automatic code generation for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, 2014.

[159] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. *SIGPLAN Not.*, 46(3):393–404, March 2011.

[160] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. Profiling a warehouse-scale computer. In *ISCA*, 2015.

[161] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, 2000.

[162] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[163] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.

[164] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing GPU Concurrency in Heterogeneous Architectures. In *MICRO*, 2014.

[165] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[166] Mahmoud Khairy, Mohamed Zahran, and Amr Wassal. SACAT: Streaming-aware conflict-avoiding thrashing-resistant GPGPU cache management scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[167] Mahmoud Khairy, Mohamed Zahran, and Amr G Wassal. Efficient utilization of GPGPU cache hierarchy. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2015.

[168] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[169] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Trans. Archit. Code Optim.*, January 2013.

[170] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. *SIGPLAN Not.*, 37(10):159–170, October 2002.

[171] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. Toward standardized near-data processing with unrestricted data placement for GPUs. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2017.

[172] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *CAL*, 2016.

[173] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies. *BMC Genomics*, 2018.

[174] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2011.

[175] Kyu Yeun Kim, Jinsu Park, and Woongki Baek. IACM: Integrated adaptive cache management for high-performance and energy-efficient GPGPU computing. In *International Conference on Computer Design (ICCD)*, 2016.

[176] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[177] John Kloosterman, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. WarpPool: Sharing requests with inter-warp coalescing for throughput processors. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[178] P. M. Kogge. EXECUBE—A New Architecture for Scaleable MPPs. In *ICPP*, 1994.

[179] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. Stash: Have your scratchpad and cache it too. ISCA, 2015.

[180] Gunjae Koo, Hyeran Jeon, and Murali Annavaram. Revealing critical loads and hidden data locality in GPGPU applications. In *International Symposium on Workload Characterization (IISWC)*, 2015.

[181] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. Access pattern-aware cache management for improving data utilization in GPU. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[182] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: Heterogeneous parallel virtual machine. PPoPP, 2018.

[183] Bo-Cheng Charles Lai, Hsien-Kai Kuo, and Jing-Yang Jou. A cache hierarchy aware thread mapping methodology for GPGPUs. *IEEE Transactions on Computers*, 2015.

[184] Nagesh B Lakshminarayana and Hyesoon Kim. Spare register aware prefetching for graph algorithms on GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[185] Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu. Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-data-port DRAM. In *PACT*, 2015.

[186] HyoukJoong Lee, Kevin J Brown, Arvind K Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware Mapping of Nested Parallel Patterns on GPUs. In *MICRO*, 2014.

[187] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[188] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *International Symposium on Microarchitecture (MICRO)*, 2010.

[189] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[190] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[191] Shin-Ying Lee and Carole-Jean Wu. Ctrl-C: Instruction-aware control loop based adaptive cache bypassing for GPUs. In *International Conference on Computer Design (ICCD)*, 2016.

[192] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. *SIGARCH Comput. Archit. News*, 41(3):487–498, June 2013.

[193] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA, 2007.

[194] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-aware CTA clustering for modern GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[195] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2015.

[196] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic GPU cache bypassing. In *International Conference on Supercomputing (ICS)*, 2015.

[197] Dong Li, Minsoo Rhu, Daniel R Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S Fussell, and Stephen W Redder. Priority-based cache allocation in throughput processors. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[198] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-based hybrid memory management. 2017.

[199] Yun Liang, Xiaolong Xie, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (ICCAD)*, 2015.

[200] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[201] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. ISCA, 2012.

[202] Lei Liu, Zehan Cui, Yong Li, Yungang Bao, Mingyu Chen, and Chengyong Wu. BPM/BPM+: Software-based dynamic memory partitioning mechanisms for mitigating DRAM bank-/channel-level interferences in multicore systems. *ACM Trans. Archit. Code Optim.*

[203] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012.

[204] Lei Liu, Yong Li, Zehan Cui, Yungang Bao, Mingyu Chen, and Chengyong Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[205] Peng Liu, Jiyang Yu, and Michael C. Huang. Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads. *ACM Trans. Archit. Code Optim.*

[206] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving DRAM refresh-power through critical data partitioning. *SIGPLAN Not.*, 46(3), March 2011.

[207] Yangguo Liu, Junlin Lu, Dong Tong, and Xu Cheng. Locality-aware bank partitioning for shared DRAM MPSoCs. In *22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[208] Yixun Liu, Eddy Z Zhang, and Xipeng Shen. A cross-input adaptive framework for gpu program optimizations. In *IPDPS*, 2009.

[209] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent Data Structures for Near-Memory Computing. *SPAA*, 2017.

[210] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, April 2012.

[211] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[212] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 40–51, New York, NY, USA, 2001. ACM.

[213] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[214] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 467–478. IEEE, 2014.

[215] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, et al. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *ASPLOS*, 2015.

[216] M. Manivannan, V. Papaefstathiou, M. Pericas, and P. Stenstrom. Radar: Runtime-assisted dead region management for last-level caches. In *HPCA*, 2016.

[217] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 77–84, New York, NY, USA, 1998. ACM.

[218] Jiří Matela, Martin Šrom, and Petr Holub. Low gpu occupancy approach to fast arithmetic coding in jpeg2000. In *Mathematical and Engineering Methods in Computer Science*, pages 136–145. Springer, 2011.

[219] Robert McGill, John W Tukey, and Wayne A Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[220] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. Tile size selection revisited. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[221] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA-37*, 2010.

[222] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. igpu: Exception support and speculative execution on gpus. *SIGARCH Comput. Archit. News*, 2012.

[223] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[224] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. In *CAL*, 2012.

[225] Wei Mi, Xiaobing Feng, Jingling Xue, and Yaocang Jia. Software-hardware cooperative DRAM bank partitioning for chip multiprocessors. *Network and parallel computing*, 2010.

[226] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 50–61. ACM, 2015.

[227] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[228] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. Beyond the socket: NUMA-aware GPUs. In *International Symposium on Microarchitecture (MICRO)*, 2017.

[229] Sparsh Mittal. A survey of cache bypassing techniques. *Journal of Low Power Electronics and Applications*, 2016.

[230] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 321–334, New York, NY, USA, 2001. ACM.

[231] Shuai Mu, Yandong Deng, Yubei Chen, Huaiming Li, Jianming Pan, Wenjun Zhang, and Zhihua Wang. Orchestrating cache management and memory scheduling for GPGPU applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2014.

[232] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *ISCA*.

[233] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Whirlpool: Improving dynamic cache management with static data classification. In *ASPLOS*, 2016.

[234] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[235] O. Mutlu. Memory scaling: A systems architecture perspective. In *IMW*, 2013.

[236] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.

[237] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, 2003.

[238] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO*, 2011.

[239] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.

[240] Nintendo/Creatures Inc./GAME FREAK inc. Pokémon. `http://www.pokemon.com/us/`.

[241] B.S. Nordquist and S.D. Lew. Apparatus, system, and method for coalescing parallel memory requests, February 17 2009. US Patent 7,492,368.

[242] NVIDIA. Programming Guide.

[243] NVIDIA. PTX ISA Version 6.0. In `http://docs.nvidia.com/cuda/parallel-thread-execution/#cache-operators`.

[244] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.

[245] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, 2011.

[246] NVIDIA. CUDA Programming Guide. 2018.

[247] NVIDIA Corp. CUDA.

[248] L. Nyland, J.R. Nickolls, G. Hirota, and T. Mandal. Systems and methods for coalescing memory accesses of parallel threads, December 27 2011. US Patent 8,086,806.

[249] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt. How to Fake 1000 Registers. In *MICRO*, 2005.

[250] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. APRES: Improving cache efficiency by exploiting load characteristics on GPUs. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[251] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, 2013.

[252] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[253] Abhisek Pan and Vijay S. Pai. Runtime-driven shared last-level cache management for task-parallel programs. SC '15, 2015.

[254] Vassilis Papaefstathiou, Manolis GH Katevenis, Dimitrios S Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *ICS*, 2013.

[255] Heekwon Park, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2013.

[256] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *SIGARCH Comput. Archit. News*.

[257] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 1997.

[258] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *HPDC*, 2014.

[259] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities. In *PACT*, 2016.

[260] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. A case for toggle-aware compression for GPU systems. In *HPCA*, 2016.

[261] Gennady Pekhimenko, Evgeny Bolotin, Mike O'Connor, Onur Mutlu, Todd C Mowry, and Stephen W Keckler. Toggle-Aware Compression for GPUs. In *HPCA*, 2016.

[262] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Exploiting compressed block size as an indicator of future reuse. In *HPCA*, 2015.

[263] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Linearly Compressed Pages: A Low Complexity, Low Latency Main Memory Compression Framework. In *MICRO-46*, 2013.

[264] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Todd C. Mowry, Phillip B. Gibbons, and Michael A. Kozuch. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *PACT*, 2012.

[265] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[266] Ping-Che Chen. N-queens solver. 2008.

[267] L. Pouchet. Polybench: The polyhedral benchmark suite.

[268] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[269] Moinuddin K Qureshi and Gabe H Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[270] Moinuddin K Qureshi, Onur Mutlu, and Yale N Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. DSN, 2005.

[271] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *PLDI*, 2013.

[272] Rajiv Ravindran, Michael Chu, and Scott Mahlke. Compiler-managed partitioned data caches for low power. LCTES '07, 2007.

[273] Scott Rixner, John D Owens, Peter Mattson, Ujval J Kapasi, and William J Dally. Memory access scheduling. In *ISCA*, 2000.

[274] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.

[275] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE CAL*, 2011.

[276] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 37–, Washington, DC, USA, 2001. IEEE Computer Society.

[277] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, 2008.

[278] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded gpu. CGO, 2008.

[279] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen mei W. Hwu. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing*, 68(10), 2008.

[280] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *ASPLOS*, 2018.

[281] R. Sakai, F. Ino, and K. Hagihara. Towards automating multi-dimensional data decomposition for executing a single-GPU code on a multi-GPU system. In *International Symposium on Computing and Networking (CANDAR)*, 2016.

[282] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[283] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[284] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 2014.

[285] Daniel Sánchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA*, 2013.

[286] Muhammad Husni Santriaji and Henry Hoffmann. Grape: Minimizing energy for gpu applications with performance requirements. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[287] Somayeh Sardashti and David A. Wood. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-optimized Compressed Caching. In *MICRO-46*, 2013.

[288] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. Cooperative cache scrubbing. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2014.

[289] Jennifer B Sartor, Subramaniam Venkiteswaran, Kathryn S McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. In *INTERACT-9*, 2005.

[290] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *PACT*, 2012.

[291] Katsuto Sato, Hiroyuki Takizawa, Kazuhiko Komatsu, and Hiroaki Kobayashi. Automatic tuning of CUDA execution parameters for stencil processing. In *Software Automatic Tuning*. 2011.

[292] Christoph A Schaefer, Victor Pankratius, and Walter F Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *Euro-Par*. 2009.

[293] V. Seshadri and O. Mutlu. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers, Volume 106*. Academic Press, 2017.

[294] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[295] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Fast Bulk Bitwise AND and OR in DRAM. *CAL*, 2015.

[296] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. RowClone: Fast and Energy-efficient In-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.

[297] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.

[298] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2013.

[299] Ankit Sethia and Scott Mahlke. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 647–658, Washington, DC, USA, 2014. IEEE Computer Society.

[300] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. MemZip: Exploring Unconventional Benefits from Memory Compression. In *HPCA-20*, 2014.

[301] D. E. Shaw. The NON-VON Database Machine: A Brief Overview. *IEEE Database Eng. Bull.*, 1981.

[302] Jonas Skeppstedt and Michel Dubois. Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP)*, 1997.

[303] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for HPC with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[304] B. Smith. A Pipelined, Shared Resource MIMD Computer. In *ICPP*, 1978.

[305] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. *SIGARCH Comput. Archit. News*, 25(2):194–205, May 1997.

[306] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News*, 2009.

[307] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[308] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning for opencl devices. In *Euro-Par*. 2010.

[309] SPEC CPU2006 Benchmarks. http://www.spec.org/.

[310] S. Srinath, O. Mutlu, Hyesoon Kim, and Y.N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA*, 2007.

[311] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O'Connor, and Stephen W Keckler. Flexible software profiling of GPU architectures. In *ISCA*, 2015.

[312] H. S Stone. A Logic-in-Memory Computer. *IEEE TC*, 1970.

[313] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012.

[314] John A Stratton, Christopher Rodrigues, I-Jui Sung, Li-Wen Chang, Nasser Anssari, Geng Liu, W Hwu Wen-mei, and Nady Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *Computer*, (8):26–32, 2012.

[315] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *ASPLOS*, 2010.

[316] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 2014.

[317] M Aater Suleman, Onur Mutlu, José A Joao, Yale N Patt, et al. Data Marshaling for Multi-core Architectures. In *ISCA*, 2010.

[318] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. *SIGPLAN Not.*, 35(11):257–268, November 2000.

[319] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *ISCA*, 2014.

[320] Xun Tang, Maha Alabduljalil, Xin Jin, and Tao Yang. Partitioned similarity search with cache-conscious data traversal. *TKDD*, 2017.

[321] J. E. Thornton. Parallel Operation in the Control Data 6600. In *AFIPS FJCC*, 1964.

[322] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Laboratories, 2008.

[323] Martin Thuresson, Lawrence Spracklen, and Per Stenstrom. Memory-Link Compression Schemes: A Value Locality Perspective. *IEEE Trans. Comput.*, 57(7), July 2008.

[324] Bradley Thwaites, Gennady Pekhimenko, Hadi Esmaeilzadeh, Amir Yazdanbakhsh, Onur Mutlu, Jongse Park, Girish Mururu, and Todd Mowry. Rollback-free value prediction with approximate loads. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 493–494. ACM, 2014.

[325] Yingying Tian, Sooraj Puthoor, Joseph L Greathouse, Bradford M Beckmann, and Daniel A Jiménez. Adaptive GPU cache bypassing. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2015.

[326] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2014.

[327] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. Dependent partitioning. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[328] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. Jenga: Software-defined cache hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, 2017.

[329] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[330] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-mei W. Hwu. *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, chapter CUDA-Lite: Reducing GPU Programming Complexity. 2008.

[331] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. Tida: High-level programming abstractions for data locality management. In *International Conference on High Performance Computing (ISC)*, 2016.

[332] Hans Vandierendonck and Koenraad De Bosschere. Xor-based hash functions. *IEEE Transactions on Computers*, 54(7):800–812, 2005.

[333] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs. In *ISCA*, 2018.

[334] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. Zorua: A Holistic Approach to Resource Virtualization in GPUs. In *MICRO*, 2016.

[335] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. Decoupling the Programming Model from Resource Management in Throughput Processors. In *Many-Core Computing: Hardware and Software, IET*, 2018.

[336] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B Gibbons, and Onur Mutlu. A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory. In *ISCA*. 2018.

[337] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C Mowry, and Onur Mutlu. A case for core-assisted bottleneck acceleration in gpus: enabling flexible data compression with assist warps. In *ISCA*, 2015.

[338] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C Mowry, and Onur Mutlu. A Framework for Accelerating Bottlenecks in GPU Execution with Assist Warps. *Advances in GPU Research and Practices, Elsevier*, 2016.

[339] Oreste Villa, Daniel R. Johnson, Mike O'Connor, Evgeny Bolotin, David W. Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, Stephen W. Keckler, and William J. Dally. Scaling the power wall: A path to exascale. 2014.

[340] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. Bump: Bulk memory access prediction and streaming. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[341] Jack Wadden, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. Real-world design and evaluation of compiler-managed gpu redundant multithreading. ISCA '14, 2014.

[342] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, 2002.

[343] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S. Vetter. Exploring hybrid memory for gpu energy efficiency through software-hardware co-design. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2013.

[344] Bin Wang, Weikuan Yu, Xian-He Sun, and Xinning Wang. DaCache: Memory divergence-aware GPU cache management. In *International Conference on Supercomputing (ICS)*, 2015.

[345] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Laperm: Locality aware scheduler for dynamic parallelism on GPUs. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[346] Zhenlin Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003.

[347] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2002.

[348] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *HPCA*, 2016.

[349] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 2001.

[350] S. P. Vander Wiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*, 1999.

[351] M. V. Wilkes. The memory gap and the future of high performance memories. *CAN*, 2001.

[352] Steven JE Wilton and Norman P Jouppi. CACTI: An enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, 1996.

[353] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *ASPLOS*, 2002.

[354] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *International Conference on Supercomputing (ICS)*, 2015.

[355] Ping Xiang, Yi Yang, and Huiyang Zhou. Warp-level divergence in gpus: Characterization, impact, and mitigation. In *HPCA*, 2014.

[356] Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[357] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on GPUs. In *International Conference on Computer-Aided Design, (ICCAD)*, 2013.

[358] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing for GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[359] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: A language and compiler for DSP algorithms. In *ACM SIGPLAN Notices*, 2001.

[360] Jun Yan and Wei Zhang. Virtual Registers: Reducing Register Pressure Without Enlarging the Register File. In *HIPEAC*, 2007.

[361] Jun Yan and Wei Zhang. Exploiting Virtual Registers to Reduce Pressure on Real Registers. *TACO*, 2008.

[362] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.

[363] Hongbo Yang, Ramaswamy Govindarajan, Guang Gao, and Ziang Hu. Compiler-assisted cache replacement: Problem formulation and performance evaluation. *LCPC*, 2004.

[364] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent Value Compression in Data Caches. In *MICRO-33*, 2000.

[365] Yi Yang, Ping Xiang, Jingfei Kong, Mike Mantor, and Huiyang Zhou. A unified optimizing compiler framework for different gpgpu architectures. *ACM Trans. Archit. Code Optim.*, June 2012.

[366] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. *PLDI*, 2010.

[367] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. Shared memory multiplexing: a novel way to improve gpgpu throughput. In *PACT*, 2012.

[368] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. Rfvp: Rollback-free value prediction with safe-to-approximate loads. *TACO*, 2016.

[369] Hangin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row buffer locality-aware data placement in hybrid memories. In *ICCD*, 2011.

[370] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit. In *ISCA*, 2016.

[371] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient DRAM caching via software/hardware cooperation. MICRO '17, 2017.

[372] Zihao Yu, Bowen Huang, Jiuyue Ma, Ninghui Sun, and Yungang Bao. Labeled RISC-V: A new perspective on software-defined architecture. In *CARVV*, 2017.

[373] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. CGO, 2010.

[374] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[375] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12, 2012.

[376] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Two-level Hierarchical Register File Organization for VLIW Processors. In *MICRO*, 2000.

[377] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, 2008.

[378] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 85–95, Washington, DC, USA, 2007. IEEE Computer Society.

[379] Yang Zhang, Zuocheng Xing, Cang Liu, Chuan Tang, and Qinglin Wang. Locality based warp scheduling in GPGPUs. *Future Generation Computer Systems*, 2017.

[380] Yang Zhang, Zuocheng Xing, Li Zhou, and Chunsheng Zhu. Locality protected dynamic cache allocation scheme on GPUs. In *Trustcom/BigDataSE/ISPA*, 2016.

[381] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *International Symposium on Microarchitecture (MICRO)*, 2000.

[382] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. Breaking address mapping symmetry at multi-levels of memory hierarchy to reduce DRAM row-buffer conflicts. *The Journal of Instruction-Level Parallelism*, 2001.

[383] Chen Zhao, Fei Wang, Zhen Lin, Huiyang Zhou, and Nanning Zheng. Selectively GPU cache bypassing for un-coalesced loads. In *International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[384] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Adaptive cache and concurrency allocation on GPGPUs. *IEEE Computer Architecture Letters (CAL)*, 2015.

[385] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *TPDS*, 2014.

[386] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L. Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs. *ACM Trans. Archit. Code Optim.*, 2016.

[387] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. *SIGARCH Comput. Archit. News*, 29(2):2–13, May 2001.

[388] William K Zuravleff and Timothy Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order, 1997. US Patent 5,630,096.