

Copyright
by
Kishore Punniyamurthy
2021

The Dissertation Committee for Kishore Punniyamurthy
certifies that this is the approved version of the following dissertation:

**Data Movement Optimizations for GPU-based
Non-Uniform Processing-in-Memory Systems**

Committee:

Andreas Gerstlauer, Supervisor

Lizy K. John

Mattan Erez

George Biros

Bradford M. Beckmann

**Data Movement Optimizations for GPU-based
Non-Uniform Processing-in-Memory Systems**

by

Kishore Punniyamurthy,

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2021

Dedicated to my parents.

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. Andreas Gerstlauer. My PhD would not have been possible without his expert guidance and support. He is a person with high standards and a dedication towards perfection. Working with him has helped me evolve both as a human and a researcher. I would like to specially thank him for supporting me to venture into research topics different from our bread and butter areas. My PhD journey had more downs than ups and I could not have navigated it without his endless patience.

I was fortunate to learn from and work with professors who had genuine interest in my development and success during the course of my PhD. After my advisor, I have worked closely the most with Dr. Lizy K. John during my time at UT. She gave me exposure to paper reviewing, supported and guided me constantly from the days when I was working on performance modelling to present day, I am extremely grateful to her. I would like to thank Prof. Mattan Erez for his suggestions and guidance. I learned a lot from Prof. Mattan from his course and while TAing for him. I would also like to thank Prof. George Biros for the enthusiastic discussions about hardware/software co-design and his suggestions and guidance on my thesis and projects. Being a TA is special for me and is something I enjoyed doing during my time at UT. I would like

to thank Prof. Yale N. Patt for providing me my first opportunity to be a TA. His computer architecture lecture was my first graduate lecture at UT and also the first course I got to TA. Listening to his lectures and TAing for him has been a wonderful and rewarding experience. I would like to specifically thank Dr. Jonathan W. Valvano for encouraging me and giving me a chance to teach an entire lecture under his supervision. It remains as one of my best memories at UT.

During the course of my education I got a chance to work with wonderful people as part of internships. The guidance and insight I received from them during my internships were vital in shaping my PhD. I wish to thank Brad Beckmann, Sooraj Puthoor, Benoy Alexander, Tarun Nakra, Shomit Das, Jagadish Kotra, Sergey Blagodurov, Mohammad Seyedzadeh for all their mentorship, insight, and valuable feedback.

Over the course of my graduate studies I had the pleasure of making many great friends who made navigating my PhD journey easier. First of all, I would like to thank my research lab mates: Kamyar Mirzazad, Zhuoran Zhao, Xinnian Zheng, Mochamad Asri, Alex Cathis, Susana Alcorta Lozano, Dimitrios Gourounas, Sabine Francis, Wooseok Lee, Seogoo Lee and Dongwook Lee. Having them around for brainstorming ideas, discussing PhD progress/troubles and festivities helped me endure difficult times. I would also like to thank Behzad Boroujerdian, Reena Panda, Jiajun Wang, Aditya Karanam, Esha Choukse, Sangkug Lym for helping me out during my PhD.

Most importantly, I would like to thank my parents and my brother for

letting me undertake this journey and supporting me throughout its course. I got admitted as a Masters student and my parents shouldered the financial burden during the initial days, while being close to retirement themselves. During the course of my PhD, my parents and brother would put up a brave front despite the illness, injury and troubles they endured, handling the problems among themselves to ensure that I am not worried and can focus on my PhD. Anytime they sense that I am feeling low, they would encourage me to march on, assuring me that things will workout well. It is the sacrifices they made that kept me going against the setbacks. At the very least, I had to try to make my PhD worth their efforts.

Data Movement Optimizations for GPU-based Non-Uniform Processing-in-Memory Systems

Publication No. _____

Kishore Punniyamurthy, Ph.D.
The University of Texas at Austin, 2021

Supervisor: Andreas Gerstlauer

Recent technological trends have aided the design and development of large-scale heterogeneous systems in several ways: 1) 3D-stacking has enabled opportunities to place compute units into memory stacks, and 2) advancements in packaging technology now allow integrating high-bandwidth memory in the same package as compute. These trends have opened up a new class of non-uniform processing-in-memory (NUPIM) system architectures. NUPIM systems consist of multiple modules each integrating (2.5D or 3D stacked) memory and compute together in the same package and interconnected via an off-chip network. Such modularity allows system scalability, but also exacerbates the performance and energy penalty of data movement. Inter-module data movement becomes the limiting factor for performance and energy-efficiency scaling.

Existing approaches to address data movement either do not account for dynamic, performance-critical application and system interactions, or incur

high overhead that does not scale to NUPIM systems. My work focuses addressing both the cause and the effect of data movement in NUPIM systems by collecting and exploiting knowledge about application and system behavior using scalable, low-overhead software and hardware techniques. Specifically, my research addresses data movement by: 1) accelerating critical data to mitigate traffic impact, 2) reducing the number of data bits moved, and 3) eliminating the need to move data in the first place.

To mitigate traffic impact, I first propose a low-overhead yet scalable scheme for congestion management in off-chip NUPIM networks. This approach dynamically tracks the congested links and memory divergence using low-overhead techniques, and then accelerates the performance-critical data traffic. The collected information is further used to dynamically manage link widths and save I/O energy. Results show that the proposed scheme achieves on average 16% (and up to 33%) improvement over baseline and 10% (and up to 29%) improvement over other congestion mitigation schemes.

To reduce I/O link traffic in NUPIM systems, I further propose cache-line utilization-aware link traffic compression (CUALiT). CUALiT exploits the variation in temporal and spatial utilization of individual cacheline words to achieve higher compression ratios. I utilize a novel mechanism to predict utilization of cachelines across warps at word granularity. The unutilized words are pruned, latency-critical words are traditionally compressed and words with temporal slack are coalesced across cachelines and compressed lazily to achieve higher compression ratios. Results show that CUALiT achieves up to 24%

lower system energy and on average 11% (up to 2x) higher performance over traditional compression schemes.

Finally, to help eliminate the need to move data, knowledge about application locality is critical in co-locating data and compute. I propose TAFE, a framework for accurate dynamic thread address footprint estimation of GPU applications. TAFE combines minimal static address pattern annotations with dynamic data dependency tracking to compute threadblock-specific address footprints of both data-dependent and -independent access patterns prior to kernel launch. I propose pure software as well as hardware-assisted mechanisms for lightweight dependency tracking with minimal overhead. Furthermore, I develop compiler support for the framework to improve its applicability and reduce programmer overhead. Simulator-based evaluations show that TAFE achieves 91% estimation accuracy across a range of benchmarks. TAFE-assisted page/threadblock mapping improves performance 32%-45% across different configurations. When evaluating TAFE on a real multi-GPU system, results show that TAFE-based data-placement hints reduce application runtime by 10% on average while minimizing programmer effort.

Table of Contents

Acknowledgments	v
Abstract	viii
List of Tables	xiv
List of Figures	xv
Chapter 1. Introduction	1
1.1 Data Movement	2
1.2 Mitigating Data Traffic Impact	4
1.3 Reducing Amount of Data Moved	5
1.4 Eliminating Need to Move Data	6
1.5 Thesis Statement	7
1.6 Contributions	8
Chapter 2. Background	11
Chapter 3. Related Work	15
3.1 Network Congestion Management	15
3.2 Data Compression and Variable Cachelines	18
3.3 Data/Thread Locality Tracking and Optimizations	21
Chapter 4. Congestion Management in Off-Chip Networks	24
4.1 NUPIM Networks	25
4.1.1 Congestion Behavior	25
4.1.2 Divergence Behavior	28
4.2 Off-chip Congestion Management	29
4.2.1 Critical Path Detection	30

4.2.2	Path-based Prioritization	33
4.2.3	Divergence-based Prioritization	34
4.2.4	Link Width Management	35
4.3	Experimental Setup	37
4.3.1	Benchmarks	38
4.4	Evaluation	42
4.4.1	Performance Results	42
4.4.2	Energy Results	43
4.5	Summary and Conclusions	44
Chapter 5.	Cacheline Utilization-Aware Link Compression	46
5.1	CUALiT Overview	47
5.2	Variation in Cacheline Utilization	48
5.3	CUALiT	50
5.3.1	Utilization Prediction	51
5.3.2	Spatial Utilization-aware Pruning	55
5.3.3	Slack-aware compression	56
5.3.4	Decompression	58
5.4	Evaluation	59
5.4.1	Utilization Predictor	61
5.4.2	Off-chip Traffic	62
5.4.3	Energy Results	63
5.4.4	Performance Results	64
5.5	Summary and Conclusions	65
Chapter 6.	Thread Address Footprint Estimation	67
6.1	TAFE Overview	68
6.2	Thread Address Footprints	70
6.2.1	Thread ID-Dependent Accesses	72
6.2.2	Input Data-Dependent Accesses	73
6.3	Basic TAFE Framework	75
6.3.1	Static Code Annotations	77
6.3.2	Dynamic Data Dependency Tracking	81

6.3.3	Basic TAFE Limitations	86
6.4	Hardware-Assisted Dependency Tracking	88
6.4.1	Hardware Architecture	88
6.4.2	Limitations	91
6.5	TAFE Generalization and Automation	92
6.5.1	TAFE Generalization	92
6.5.2	Compiler support for TAFE	97
6.5.3	Extended TAFE Limitations	101
6.6	Evaluation	103
6.6.1	Simulator Setup	104
6.6.2	Address Footprint Accuracy	107
6.6.3	TAFE Hardware Overhead	110
6.6.4	TAFE-Based Optimizations	113
6.6.5	Evaluation on Multi-GPU System	115
6.7	Summary and Conclusions	117
Chapter 7.	Summary, Conclusions and Future Research	119
7.1	Dissertation Summary	119
7.2	Future Work	120
	Bibliography	123
	Index	144
	Vita	146

List of Tables

4.1	Simulation parameters.	39
4.2	Benchmarks.	40
5.1	Benchmarks.	60
5.2	Simulation parameters.	61
6.1	TAFE API.	77
6.2	APD Entries.	81
6.3	Extended TAFE API.	93
6.4	Simulation parameters.	105
6.5	Benchmarks.	106
6.6	TAFE hardware costs.	111
6.7	Benchmarks for compiler-assisted TAFE.	116

List of Figures

1.1	NUPIM system architecture.	1
2.1	NUPIM configurations.	13
4.1	Interconnect congestion stalls.	26
4.2	Distribution of divergent memory accesses.	27
4.3	Delay between first and last divergent response.	28
4.4	Critical path detection.	32
4.5	Simulation setup.	37
4.6	Average (PS_{avg}) vs. maximum (PS_{max}) page sharing of benchmark applications.	41
4.7	Performance across different congestion management schemes.	41
4.8	Energy breakdown.	44
5.1	CUALiT Overview.	48
5.2	Cacheline utilization.	49
5.3	CUALiT Block Diagram.	50
5.4	Access granularity variation.	52
5.5	Utilization prediction example.	53
5.6	Granularity predictor	54
5.7	Pruning and compression.	56
5.8	Lazy batch compression.	57
5.9	Decompression.	59
5.10	Simulated system.	60
5.11	Prediction mechanism performance.	62
5.12	Normalized off-chip traffic.	63
5.13	Normalized Energy saving.	64
5.14	IPC with 68B and 32B flitsizes.	65

6.1	TAFE overview.	68
6.2	Generic array access patterns.	71
6.3	Thread ID-dependent access.	72
6.4	Data-dependent access.	74
6.5	CSR example (<i>Bfs</i>).	76
6.6	Access pattern directory (APD) and entries.	80
6.7	CSR example: populated APD table.	80
6.8	Extent table.	83
6.9	TAFE hardware for tracking primary arrays.	89
6.10	TAFE compiler support.	98
6.11	Simulated exascale node configuration.	103
6.12	TAF tracking accuracy.	109
6.13	TAF estimation accuracy.	110
6.14	TAFE runtime overhead.	112
6.15	Off-chip traffic.	113
6.16	Warp memory load latency.	114
6.17	IPC.	114
6.18	Normalized multi-GPU runtime.	117

Chapter 1

Introduction

The growing demand for computational capability in scientific and high-performance computing demands larger systems to meet requirements. It is projected that exascale computing will require thousands of nodes where each node performs at least 10 TFLOPS of computation and has 5 TB of memory [108]. These requirements cannot be satisfied by traditional node architectures [114]. Fabricating larger chips is challenging as it results in lower yield rendering such designs expensive and unscalable. By contrast, advancements in packaging and 3D-stacking technology have aided the design and development of modular nodes each consisting of multiple interconnected modules. 3D-stacking has enabled opportunities to place memory and com-

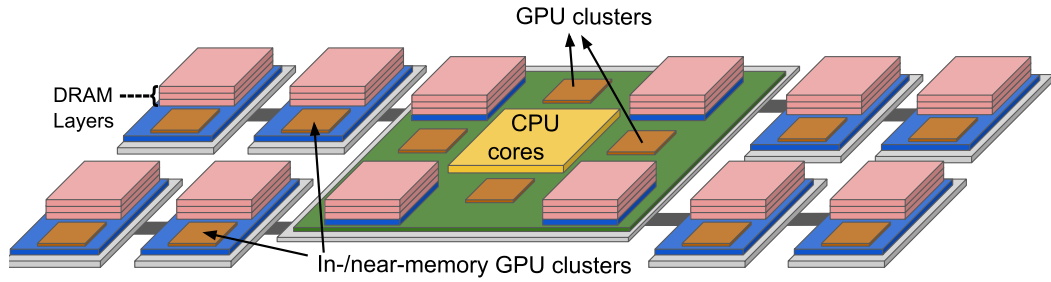


Figure 1.1: NUPIM system architecture.

pute units on top of each other such that in-memory compute units have significantly greater bandwidth access to memory provided by through-silicon vias (TSVs). In addition, packaging technology has evolved to now allow integrating high-bandwidth memory in the same package as compute, e.g using silicon interposers. These trends have opened up a new class of non-uniform processing-in-memory (NUPIM) system architectures (Fig. 1.1).

NUPIM systems consist of multiple modules each integrating (2.5D or 3D stacked) memory and compute together in the same package interconnected via an off-chip network. Such modularity allows system scalability, but also exacerbates the performance and energy penalty of data movement. Inter-module data movement becomes the limiting factor for performance and energy-efficiency scaling. As such, managing and reducing data movement is critical to ensure scaling with system size. In addition, both individual stacked modules as well as complete systems deploying NUPIM architectures have strict power and thermal budgets. Therefore, any solution should incur low overhead while being scalable.

1.1 Data Movement

Data movement is not an unknown problem. In the past [57, 117], researchers have projected the performance and energy overhead of data movement and the need to address the same. Keckler *et al.* [57] projected the energy required to access operands from external DRAM to be 200x greater than double-precision fused-multiply add operation. These concerns have been

corroborated by recent studies [114, 108, 51].

My research addresses data movement by: 1) accelerating critical data to mitigate traffic impact, 2) reducing the amount of data bits moved, and 3) eliminating the need to move data. Several optimizations have been proposed to address data movement, but existing approaches have limited effectiveness and applicability for NUPIM systems. On the one hand, many of these approaches are either agnostic towards performance-critical application and system interactions or only capture them statically, thus limiting the performance and energy benefits that can be achieved. On the other hand, approaches that are aware of dynamic application and system behavior rely on costly runtime techniques or additional resources that are not feasible in NUPIM systems.

My research focuses on reducing, eliminating and mitigating both the cause and the effect of data movement for NUPIM systems by collecting and exploiting knowledge about application and performance-critical system behavior using scalable, low-overhead software and hardware techniques. GPUs have been widely proposed as a choice for in- and near-memory compute [125, 48] as they are throughput-oriented processors, capable of utilizing the high bandwidth available from memory while having a uniform thermal dissipation (without thermal hotspots). Therefore, I focus on GPU-based NUPIM systems in this thesis.

1.2 Mitigating Data Traffic Impact

One way to address data movement is by alleviating the adverse effects of data movement. These optimizations involve identifying performance-critical data requests and accelerating/prioritizing them. The performance impact of a data request depends on many different criteria such as network conditions [88, 66, 58], processor behavior [43, 15] and application characteristics [112]. Critical data traffic can be prioritized during instruction [41, 69], network [90, 63, 10], or memory [43, 73] scheduling. The major advantage of such approaches is that they are user-transparent. Critical data traffic can be identified purely using hardware metrics like stall count, queue occupancy and completion latency without requiring any software modification.

The inherently distributed nature of NUPIM systems and the resulting off-chip data movement makes them particularly susceptible to network congestion issues. Off-chip network congestion is often a major factor that adversely affects their performance. Traditional solutions to reduce congestion involve adaptive routing over alternate routes in the network [39, 66]. However, both individual stacked modules as well as complete systems deploying NUPIM architectures have strict power and thermal budgets [108, 125], where I/O links and buffers already contribute to high energy overhead constituting 73% of network [51] and 31-35% of router energy [118], respectively. As such, adding additional links between modules or monitoring/overlay networks is typically not feasible. Alternative approaches that rather aim to mitigate the adverse effects of congestion have been proposed for memory networks [90],

but they don't address any-to-any traffic patterns seen in NUPIM systems with distributed compute/memory modules. In this dissertation, I instead propose a low-overhead yet scalable scheme for congestion management in off-chip NUPIM networks. This approach dynamically tracks the congested links and memory divergence using low-overhead techniques, and then accelerates the performance-critical data traffic.

1.3 Reducing Amount of Data Moved

While managing data traffic can improve performance and reduce system energy consumption (due to reduced application runtime), associated runtime optimizations incur overhead. Furthermore, they do not reduce the actual amount of data moved and therefore have limits to the benefits that can be achieved. Alternatively, inter-module data traffic can be efficiently packed and/or compressed to reduce the bits moved. Data re-arrangement approaches [77, 21] efficiently pack irregularly/sparsely accessed data to make them cache friendly. By contrast, compression approaches [60, 55, 87, 29] exploit the redundancy in data values and reduce the bits required for representing them. Finally, lack of spatial locality can result in cachelines being accessed that are not completely utilized resulting in unnecessary data being moved. Variable cacheline size techniques [98, 64] can prevent unnecessary bytes from being accessed by dynamically adjusting the cacheline size depending on the spatial utilization of the data accessed. Like data traffic management, all these approaches are also user-transparent and can be implemented

purely in hardware.

In NUPIM systems, the inter-module data traffic is dominated by memory responses, which are suitable for compression. While many different compression approaches have been proposed in the past [87, 55, 104], they typically only exploit redundancy in data values, but otherwise treat all words in a cacheline identically. At the same time, general purpose graphics processing unit (GPGPU) applications exhibit significant variation in temporal and spatial cacheline utilization [49, 98]. In this dissertation, I therefore propose cacheline-utilization aware compression, where varying spatial and temporal utilization of cacheline words depending on application and scheduling behavior is exploited to achieve higher compression ratios.

1.4 Eliminating Need to Move Data

The effectiveness of approaches to reduce data bits moved depends on values stored in memory (data content) and therefore may not be applicable for all workloads. Further, these approaches only address the symptom, but not the cause for data movement. The necessity for inter-module memory accesses itself can be reduced/eliminated by ensuring all data required for a computation is mapped within the same module as the compute. This requires knowledge about the data/thread locality in the executed applications, which can either be obtained from software or with hardware assistance. Software approaches for capturing locality information include programmer-hints [116, 115], compiler analysis [72, 89] or dedicated new languages [40, 24].

By contrast, hardware techniques involve tracking data accesses [35, 111]. Approaches for mapping of data and compute [82, 71, 115] then exploit locality available in applications to assign computations and their corresponding data closer together.

In NUPIM systems, mapping data and associated compute into the same module will keep all their communication local and thus eliminate the need for inter-module data traffic. This requires capturing data/thread locality of GPGPU applications. Existing software-based approaches [89, 115, 40] either require rewriting applications or can only track static, data-independent accesses. GPGPU application, however, often contain dynamic and data-dependent indirect access patterns. Existing hardware solutions for dynamic tracking [82, 61] have to learn patterns, make predictions and apply optimizations after a kernel has launched, which incurs runtime overhead and limits benefits. In this dissertation, I instead propose a framework for accurately estimating both static, data-independent and dynamic, data-dependent thread address footprints (TAFs) prior to thread and kernel execution. I propose pure software as well as hardware-assisted mechanisms for lightweight dependency tracking with minimal overhead. Furthermore, I develop compiler support for the framework to improve its applicability and reduce programmer overhead.

1.5 Thesis Statement

Scalable and accurate hardware-assisted tracking of system and application behavior can significantly enhance the effectiveness and efficiency of

data movement optimizations. This can be achieved by a) tracking and prioritizing performance-critical data to mitigate the impact of network congestion, b) utilization-aware compression to reduce the amount of data moved, and c) estimating the dynamic thread address footprints to better exploit locality and reuse.

1.6 Contributions

The specific contributions of this dissertation are as follows:

- *Off-chip Congestion Management* [95]: I propose a low-overhead yet scalable congestion management scheme to identify and prioritize critical packets across the network. I first study data movement and congestion within off-chip networks, of GPU-based NUPIM systems. Results show that NUPIM systems are susceptible to a form of tree saturation resulting in critically congested paths comprised of several individual links. I further propose a scheme that dynamically predicts congested paths within the network. Once a critically congested path has been identified, flits traversing along it are prioritized over other flits to improve performance. The critical path information is also used to reduce interconnect energy by aggressively and adaptively varying the width of non-critical links without significantly impacting performance. Furthermore, the scheme exploits the differing performance impact of data responses due to memory divergence. Non-divergent data responses are

prioritized over divergent responses to maximize performance benefits. The proposed scheme achieves on average 16% (and up to 33%) improvement over baseline and 10% (and up to 29%) improvement over other congestion mitigation schemes for benchmarks with high data sharing. Using link width management, up to 29% system energy can be saved while achieving up to 30% performance improvement.

- *Cacheline Utilization-aware Link Traffic Compression (CUALiT)* [93, 34]: I propose an extended compression scheme to exploit the variation in spatial and temporal utilization of cachelines. In this work, I first analyze the variation in spatial and temporal utilization across cacheline words for GPGPU applications. I further propose a novel mechanism that allows fine-grained prediction of cacheline utilization for GPGPU applications. Predicted unutilized words are dropped from responses. Immediately accessed words are compressed using traditional methods while words with temporal slack in access time are coalesced across cachelines and compressed lazily to achieve higher compression ratios. Evaluations show that CUALiT achieves up to 24% lower system energy and on average 11% (up to 2x) higher performance over traditional compression scheme. Furthermore, under high bandwidth demand, the scheme provides 16% more performance improvement, demonstrating its effectiveness to reduce bandwidth demand.
- *Thread Address Footprint Estimation (TAFE)* [96]: I propose TAFE, a framework for accurately estimating both static, data-independent

and dynamic, data-dependent *thread address footprints (TAFs)* prior to thread and kernel execution. I first introduce the concept of TAFs and propose *TAF relations* to succinctly represent TAFs as a relation of thread IDs and input data dependencies. My TAFE framework constructs TAF relations using a combination of minimal static code annotations and dynamic data dependency tracking. I propose low-overhead software mechanisms and optional hardware extensions to collect the data dependency information required for TAF estimation just-in-time before launch of a kernel. I also develop compiler support for TAFE to expand its applicability and reduce its programming overhead. I evaluate TAFE’s accuracy and demonstrate its benefits by using TAF information for data/threadblock co-location for multiple NUMA GPU system configurations. My evaluations show that TAFE achieves 91% estimation accuracy across all benchmarks. Moreover, TAFE can accurately track data-dependent access patterns with only 1.5% average error while incurring low (3%) overhead. Performing page/threadblocking mapping using TAFE improves performance by 45% and 32% across homogeneous and heterogeneous system configurations, respectively. For data-dependent applications, TAFE-assisted mapping can achieve 10% and 22% speedup compared to data-independent schemes across different configurations. When evaluating TAFE to provide data placement hints on a real multi-GPU system, results show that application runtime reduces by 10% on average.

Chapter 2

Background

In this chapter, the NUPIM systems evaluated in this dissertation are described along with constraints and assumptions pertaining to them. Vijayaraghavan *et al.* [114] show that the growing demand of scientific and high-performance computing cannot be satisfied by systems with a single monolithic system-on-chip. They projected that a single 3D-stacked DRAM module will not be sufficient to meet the capacity requirements. As such, NUPIM-type systems consisting of multiple interconnected modules are required. Off-chip data movement is a significant concern for such modular systems as it inhibits performance and energy-efficiency scaling with system size. This is corroborated by the findings in [114, 51].

In general, NUPIM systems can have different configurations consisting of multiple packages of varying sizes, each having different proportion of memory and compute integrated together as shown in Fig. 2.1. Configurations can range from having a centralized compute (Fig. 2.1a) where all the compute units are consolidated within a single large package. Alternatively, some compute can be distributed across other modules. Depending on the distribution, configurations can have modules with heterogeneous (Fig. 2.1b) or

homogeneous (Fig. 2.1c) compute-to-memory ratios. Different configurations will be suitable for different application behavior. Assuming the same total amount of compute and memory, high data sharing applications may result in lower off-chip data traffic in systems with centralized compute or with compute units aggregated in a smaller number of modules. This is because such systems can accommodate a large number of compute threads that share data or have varying memory footprints in the same module with lower off-chip accesses. By contrast, applications with little sharing, uniform accesses and high locality will benefit from distributing threads across a large number of homogeneous modules with the same compute to memory ratios.

In this dissertation, we perform evaluations of NUPIM systems with specific configuration and topology assumptions as described in the following. We base our NUPIM system on the exascale node architecture proposed in [114]. The system has a shared memory address space partitioned across modules. Specifically, we make the following assumptions:

Homogeneous compute-to-memory ratio All the modules within the NUPIM systems under consideration in this dissertation have the same compute-to-memory ratios, even though individual modules may vary in size. Such systems are suitable for regular applications with low sharing where data and computations can be cleanly partitioned. For irregular, high data sharing applications, there is no single optimal configuration as the required compute-to-memory ratio distribution can vary significantly and dynamically. Our

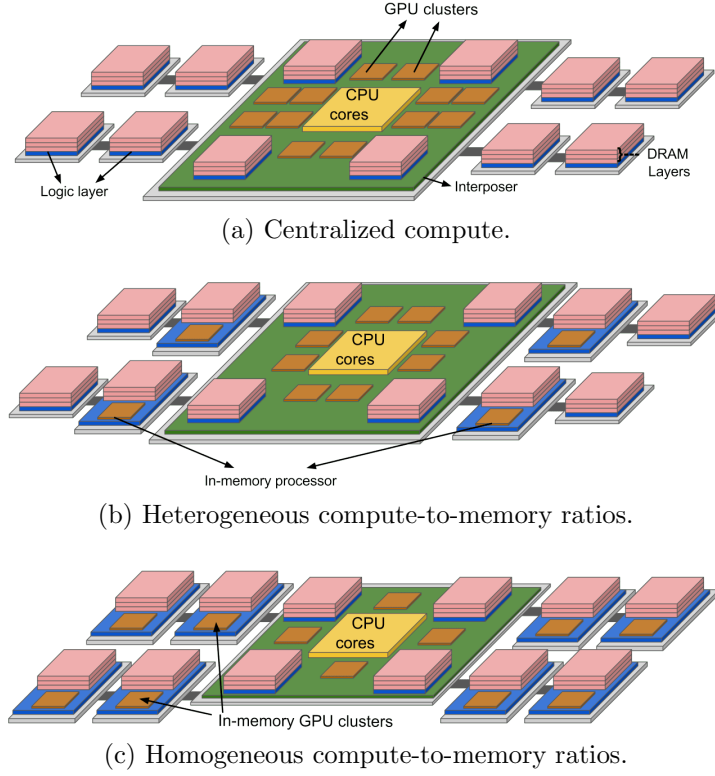


Figure 2.1: NUPIM configurations.

evaluations [94] show that while such applications could benefit from some distributed compute due to the reduced network contention, the benefits are ultimately limited by increased off-chip accesses and performance varies only minimally across different configurations. By contrast, regular applications can significantly benefit from a fully homogeneous distribution of compute. As such, we restrict our evaluations to homogeneous NUPIM systems and deploy alternate optimizations to address data movement for high data sharing applications.

Minimally-connected topologies As already mentioned in Chapter 1, both individual stacked modules as well as complete systems deploying NUPIM architectures have strict power and thermal budgets [108, 125]. Jian *et al.* [51] found that I/O links contribute to high energy overhead constituting 73% of network energy. Furthermore, buffers consume a significant proportion (31-35%) of router energy [118]. As such, adding additional links between modules or monitoring/overlay networks may not be feasible. In this dissertation, I conservatively evaluate NUPIM systems with minimally-connected tree topologies similar to [51].

GPUs as in-/near-memory compute Researchers have proposed a variety of compute units as a choice for in-/near-memory processors such as CPU [9], FPGA [78], ASIC accelerators [8] and GPUs [48]. In this dissertation, I consider GPUs as the choice of compute as they are throughput-oriented processors, capable of utilizing the high bandwidth available from memory while having a uniform thermal dissipation.

Chapter 3

Related Work

In this chapter, related works for each specific contribution of this dissertation are discussed. First, prior work regarding different approaches to identify and track network congestion, performance-critical data and associated optimizations are discussed. Then, related work on eliminating redundant and unnecessary data bytes from being accessed using compression and by varying cacheline sizes are discussed. Finally, prior research on gathering locality information using programmer-hints, compiler analysis, and hardware metrics are examined. We also discuss prior research exploring the different ways locality information can be utilized.

3.1 Network Congestion Management

Pfiser *et al.* [88] were among the first to study the issue of tree saturation within multistage networks. They showed that occurrence of tree saturation due to non-uniform traffic spots can eventually degrade the performance of the entire network. Subsequently, many researchers have tried tackling the issue of tree saturation. Adaptive routing techniques [66] have been proposed and widely studied as such an optimization. These approaches

involve diverting data traffic along non-minimalistic routes to avoid congested network regions [32, 39]. Adaptive routing approaches assume topologies that provide multiple, alternate routes between nodes. This may not be feasible in a NUPIM system due to higher I/O energy required for any additional off-chip links [51].

Adaptive routing techniques use congestion information to make routing decisions. Congestion-aware routing policies can be further classified based on whether they rely on purely local congestion information or take into account congestion status at other points in the network. Approaches considering local congestion [6] use information readily available at a given node such as queue length at output ports or counts of available virtual channels. Gratz *et al.* [44] proposed use of non-local congestion information for making routing decisions. They used a low-bandwidth monitoring network to propagate congestion information among adjacent routers. Researchers have also proposed approaches to track remote congestion without requiring dedicated monitoring or overlay networks e.g. by using credit round trip latency [59] as a proxy or via indirect adaptive routing techniques [54] for large interconnect networks. We use a similar piggybacking approach as in [54] to communicate 1-bit information across routers over packets, but our approach carries information needed for detecting congested paths and not for identifying alternate routes. As such, we send information (tracker bits) only along congested links and do not broadcast it across all routers. Furthermore, our approach only stores state about input-output port pairs and does not need to track the state

of other routers.

As an alternative to adaptive routing, traffic throttling and prioritization have been proposed to mitigate congestion impact. If the congestion is caused due to over-subscription of end-points, researchers have proposed speculative reservation [52] to prevent over-subscription of any network destination. [58] proposes an approach which detects endpoint congestion within the network and throttles the traffic from source nodes. However, we find that congestion in NUPIM systems is mostly due to over-subscription of links and not of end-points.

Prior works on prioritization have proposed many metrics to determine packets to be prioritized. [75] prioritizes traffic based on packet type. Others have suggested prioritization based on temporal locality of crossbar connections [10]. [90] proposes distance-based prioritization where data packets are prioritized based on the destination distance, but this does not address any-to-any traffic patterns seen in NUPIM systems with distributed compute/memory modules. Our approach instead considers the behavior of both the network (congestion) and application (divergence) to prioritize packets. Traffic can be prioritized and accelerated in many different ways within a network. One option is to bypass some or all pipeline stages within routers. [63] allows packets traversing a certain number of hops in a particular direction to skip the internal processing pipeline of routers and thus creating an effect of bypassing intermediate routers. Others have suggested skipping switch allocation stages [10] within routers. Another way to accelerate packets is by prioritizing

them during scheduling. [90] prioritizes data packets during switch and virtual channel allocation phases. Our approach prioritizes packets during VC and switch allocation, but any other means of prioritization can be equally used.

Recently, researchers have analyzed the energy overhead of interconnect networks and ways to address it. [51] explores different power management techniques for memory networks such as DVFS and variable width links (VWL) under network-unaware and -aware scenarios. Their approach relies on the assumption that upstream links have higher traffic than immediate downstream links. This does not apply for distributed NUPIM architectures as each module could have both memory and compute.

Our approach considers not only the behavior of the network but also of the core issuing requests while prioritizing traffic. Specifically, our approach also prioritizes non-divergent memory accesses over divergent ones. Memory divergence has been widely studied and many approaches have been proposed using warp scheduling [99], cache management [15, 14] and memory scheduling [27] to solve divergence issues. In contrast to these approaches, we do not target reducing memory divergence itself, but exploit memory divergence information to accelerate packets with the highest performance impact.

3.2 Data Compression and Variable Cachelines

Compression has been proposed as a viable way to reduce the amount of data moved by exploiting redundancy in data patterns. Researchers have proposed many compression schemes mainly focusing on minimizing decom-

pression latency for cache and link compression. Most of the earlier proposed works tend to exploit the significant amount of redundancy in data accessed by real-world applications [12, 55]. Zeroes are one of the most commonly seen data values and various approaches [37, 38] have been proposed aimed at removing redundant zeroes. Prior analysis [105] has also shown that a single value may be repeated multiple times in memory, which can be exploited to compress data. There have been works which exploit frequent values [55] or frequent patterns within cachelines [29, 85]. Some compression techniques [12] aim at smaller data values stored in larger data types (over-provisioning of data types) to reduce the amount of storage required. B Δ I [87] is based on the observation that most cachelines have a low dynamic range for data and hence, can be represented as base and series of offsets using fewer bytes than the actual cacheline size. Some schemes have suggested a data transformation step that results in data with higher compression ratio [60]. All these approaches only exploit the redundancy in data values and do not consider the utilization characteristics (word utilization, access slack) of cacheline words.

Some approaches have proposed using combinations of compression schemes. LATTE-CC [13] aims at aggressively compressing individual cachelines by switching between slow but effective and fast but relaxed compression schemes depending on the parallelism available in GPUs. Cachelines are compressed differently depending on available warps during cacheline fills. Large numbers of warps (high parallelism) allow for more latency hiding and can tolerate higher decompression latency allowing the scheme to use a slow but

effective compression technique. However, this approach requires hardware support for multiple compression techniques to provide benefits.

Data is accessed from memory at cacheline granularity, thus allowing spatial locality to be captured. However, this can result in unnecessary bytes being accessed from memory for cases where applications have low spatial locality (irregular workloads) or when processors cannot capture spatial locality effectively. One class of solutions to handle this involves rearranging data in memory. Prior works [77, 21] have proposed rearranging techniques where memory regions that would have been irregularly accessed by the processor are rearranged on demand into cache-friendly blocks in an in-memory buffer. These approaches can be effective for irregular applications, but they do not provide any benefit for cases where spatial locality cannot be effectively captured (e.g. due to cache conflicts).

Another way to solve this issue is by using sector caches [100] with a smaller block/sub-sector size. The memory is accessed always at block/sub-sector granularity, preventing un-utilized bytes from being accessed at the risk of losing spatial locality. Another approach involves evicting unused cache line words from caches [97], which increases the effective cache capacity, but does not prevent unused words from being accessed. Alternatively, researchers have proposed variable cacheline techniques which allow for partial cache lines to be accessed from memory based on predicted locality, thus reducing link bandwidth usage and potentially increasing the effective size of the cache. LAMAR [98] proposes dynamically switching between performing 32B cache-

line sector accesses against an entire 128B cacheline access depending on the locality prediction. Further, they also suggest modifications to caches to increase their effective size. Our approach does aim to reduce bandwidth requirements depending on the spatial locality captured (cacheline utilization) similar to these approaches, but we perform much finer granularity accesses (as small as 4B). Such small granularity accesses make predicting the number of words to be accessed and which words to be accessed more challenging. We do not propose changing the effective cache size but such modifications can very well be performed. Amoeba-cache [64] proposes variable cacheline size to reduce link bandwidth consumption and cache capacity utilization. This work proposes a prediction mechanism to perform fine-grained accesses, but the mechanisms presented in their work are intended for CPUs and can perform poorly for GPUs as they don't consider the similarity in memory access behavior across threads/warps.

3.3 Data/Thread Locality Tracking and Optimizations

The need for moving data off-chip can be eliminated by exploiting the locality within the application to map the computation and its corresponding data within the same module. This requires knowledge about the addresses accessed by individual compute threads. Traditional solutions learn or predict the available locality online usually using hardware techniques. The approach presented in [61] assumes strided accesses within a threadblock. A leading warp from every threadblock is executed to determine the base address ac-

cessed and the addresses accessed by remaining threads are then derived by adding a stride. [62] tracks accesses to cachelines brought in by a single warp to determine the locality and reuse specific to individual load instructions. Some approaches [82] suggested extending TLB structures to include counters that track the number of accesses to pages on an LLC miss. AMS [111] uses hardware utility monitors to profile per-thread miss curves, which reflect the number of misses a thread would incur under different cache sizes. TOM [48] proposes a solution for discrete GPU systems, where the access patterns are profiled to identify the mapping, and copying of data to device memory is delayed for that duration. Such online approaches inherently have runtime overhead. Furthermore, since they can only be applied after the kernel has been launched, their benefits are limited, and they additionally incur costs of applying optimizations at runtime (e.g. TLB shutdown during page-remapping).

Alternatively, software-based solutions have been proposed to determine locality. A wide range of programming languages have been introduced to allow explicit expression of locality [24, 26, 40, 106, 113, 20, 122, 76]. Such approaches require the programmers to be familiar with the new language and re-write the application. By contrast, TAFE can be applied over existing programming models, requiring programmers to only specify the arrays to be tracked without affecting the functionality of the application. [89, 110] uses compiler (trip count and induction variable) analysis to determine the locality of memory regions accessed, but unlike TAFE they don't support data-dependent accesses. [103] proposes compiler analysis for execution-time

pre-processing of accesses in loops, the results of which are later used for optimizations. While TAFE performs kernel code pre-processing, it can capture data dependencies during array initialization itself and is decoupled from the kernel being optimized.

Several works have proposed means to allow programmers to provide hints about access patterns. [115] presents a way to express data locality within applications. The authors propose descriptors consisting of *C-tiles* and *D-tiles* to let programmers explicitly specify the inter- and intra-thread locality, which is then utilized to perform data and compute mapping for NUMA systems. Similarly, [22] presents user-level APIs allowing the programmer to specify access patterns. These works are able to capture static, thread ID-dependent accesses, but they do not support complex dynamic, data-dependent accesses. While TAFE also relies on programmer annotated APIs, it does not require explicit mapping and data sharing information from programmers like [115] and can capture dynamic data-dependent access patterns. [101] handles specific complex access patterns using an *index mapper* that performs analyses by re-reading the entire primary data structure. This solution was proposed for discrete GPU systems, where reloading primary arrays by the host will result only in local accesses. However, such an approach will incur significant overhead in systems with unified memory architectures, limiting scalability. TAFE can handle a wider range of access patterns with little overhead even for unified memory architecture systems.

Chapter 4

Congestion Management in Off-Chip Networks

In this chapter¹, we present our congestion management scheme for off-chip NUPIM networks [95]. We first study data movement and congestion within off-chip networks of GPU-based NUPIM systems. We find that NUPIM systems exhibit any-to-any inter-module traffic patterns in their off-chip networks, where individual links can become oversubscribed and saturated, and are therefore susceptible to a form of tree saturation [88]. The branches of multiple overlaid saturated trees are comprised of individually congested links, where the effect of congestion increases along some paths more than others due to back pressure. As such, traffic along such paths needs higher bandwidth to minimize performance impact. Allocating bandwidth proportional to congestion requires detecting the critically congested paths in a scalable manner. We propose an approach to track the critical path in a distributed manner, where the information about the entire critically congested path is not available in any single switch but is distributed among the routers allowing the technique to

¹Contents of this chapter previously appeared in:
[95] Kishore Punniyamurthy and Andreas Gerstlauer. “Off-Chip Congestion Management for GPU-based Non-Uniform Processing-in-Memory Networks”, in International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2020. I proposed, designed and implemented the main ideas and collaborated with Dr. Gerstlauer on writing this paper.

scale with the network size. In addition, since servicing non-divergent memory responses is more likely to contribute to warp progress, our approach also identifies data requests that originate from non-divergent memory instructions and tracks their corresponding responses. The data traffic traversing through the detected paths and non-divergent memory responses are then prioritized during switch and virtual channel arbitration. Furthermore, the proposed scheme uses critical path information to reduce interconnect energy by aggressively and adaptively varying the width of less congested links without significantly impacting performance.

4.1 NUPIM Networks

We analyze the NUPIM network behavior to identify performance-critical data traffic. Specifically, we focus on data which has higher performance impact due to congestion within network and memory divergence. This requires an understanding of the different network traffic paths, varying performance impact of data packets traversing the link and the extent of divergence in memory responses due to distributed memory.

4.1.1 Congestion Behavior

We first study congestion behavior to demonstrate the varying importance of different traffic paths within NUPIM networks. Congestion heatmaps for *BFS* and *CFD* benchmarks executed on a NUPIM system with central module *M0* and external modules *M1-M8* are shown in Fig. 4.1. The colors

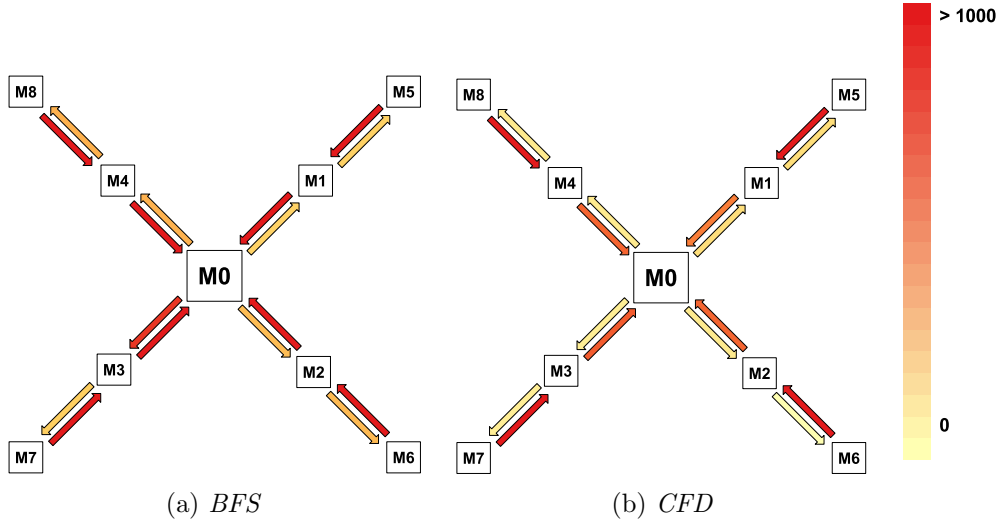


Figure 4.1: Interconnect congestion stalls.

of the links indicate the number of stalls experienced by flits traversing along the specific link due to a buffer being full. They are normalized against the average congestion stalls experienced by a single connection to an in-package stack. We can make the following observations: 1) Individual congested off-chip links can form a large critically congested path. E.g., in *BFS* (Fig.4.1a), $M6 \rightarrow M2 \rightarrow M0 \rightarrow M3$ is a critical path; 2) Due to back-pressure, a small number of stalls at the end of a congested path snowballs into a large number of stalls at the source of the path as evident from the increasing intensity of congestion from *M0* to *M5* along the critical path in *CFD* (Fig. 4.1b); and 3) The critical path can vary across applications as seen from the differences between Fig. 4.1a and 4.1b.

These observations show that distributed NUPIM systems are vulnera-

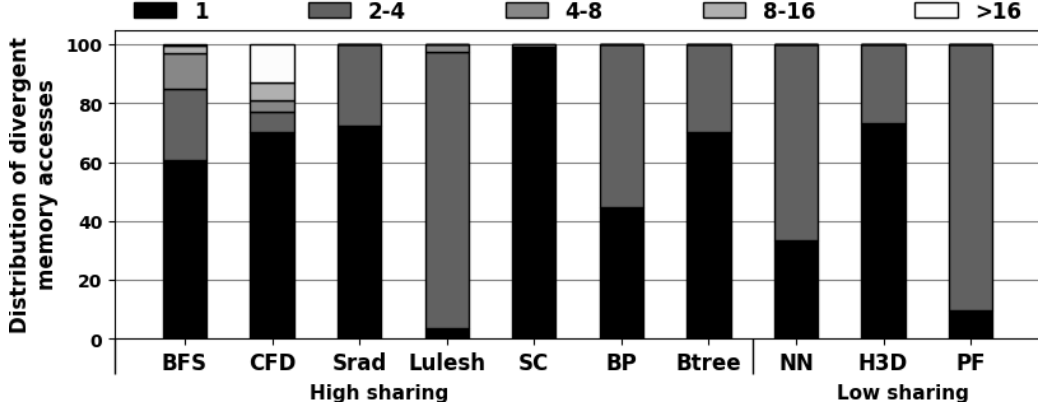


Figure 4.2: Distribution of divergent memory accesses.

ble to a form of tree saturation [88, 66]. While tree saturation can occur from both end-point and network congestion [58], we find network-congestion due to over-subscription of links around the central module to be the dominant cause. In our applications, traffic was roughly equal across all modules (the average coefficient of variation in the number of flits across different nodes is $< 7\%$) and no end-point hot-spots were observed. This is expected because of interleaved data-mapping [79] and similarity in thread memory accesses. Even if a different mapping results in hot-spots, network congestion is likely to remain the main cause for saturation due to limitations on the number of off-chip links in NUPIM systems. All in all, links comprising the critically congested paths have larger impact on performance than other links. As such, prioritizing traffic on congested over non-congested paths allow reducing congestion impact and improving performance.

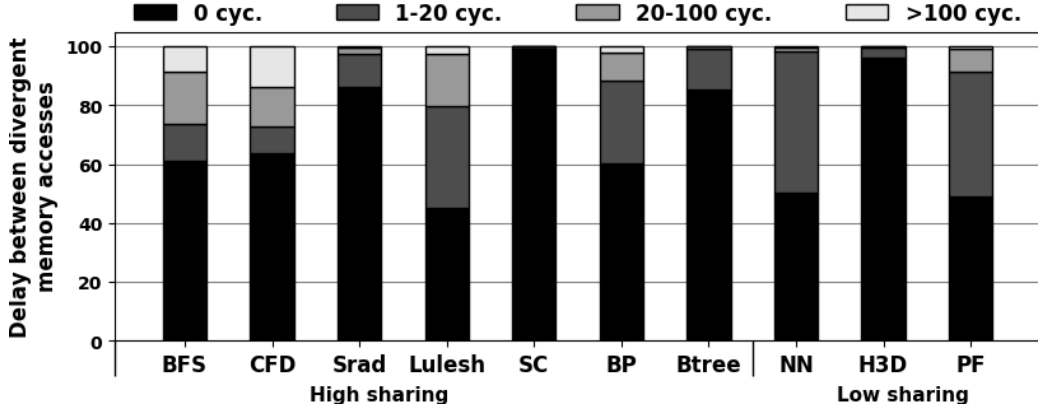


Figure 4.3: Delay between first and last divergent response.

4.1.2 Divergence Behavior

In GPUs, a warp load instruction can result in up to 32 memory accesses due to memory divergence [15], where all resulting memory requests must be serviced for the warp instruction to complete. Fig. 4.2 shows the distribution of divergent memory accesses resulting from individual warp instructions for different benchmarks. On average, 46% of warp load instructions result in more than one memory access. Memory divergence is more prevalent in high-sharing applications² due to their irregular access patterns. For example, *BFS* and *CFD* have 15% and 23% of their memory instructions resulting in 4 or more accesses, respectively.

In case of multiple accesses, there can be a significant delay between divergent responses. Fig. 4.3 shows the distribution of delay between the first

²Applications are classified as high and low sharing based on data shared between thread-blocks. Please refer to Section 4.3.1

and the last responses (excluding L1 hits) received per memory instruction for different benchmarks. A delay of 0 indicates that only 1 request was serviced at L2 or memory. It can be seen that up to 27% of memory responses have more than 20 cycles delay between first and last response. This indicates that even if 1 divergent response returns, there can be a significant delay before the instruction completes. Such delays are only expected to exacerbate as NUPIM system size scales. As such, servicing requests with lower memory divergence has higher probability to contribute to warp progress. Prioritizing responses for load instructions with lower divergence reduces more memory stalls and increases the chance of making forward progress.

4.2 Off-chip Congestion Management

We propose a low-overhead and scalable way to detect critically congested paths and divergent data responses to intelligently manage congestion within a NUPIM network while minimizing cost. We detect critical paths using a novel lightweight technique to link congested ports across switches. On detecting the critical paths, we prioritize the packets traversing along the critical path during the virtual channel (VC) and switch allocation stages of the router pipeline to improve performance. Additionally, the flits originating from non-divergent instructions are identified and prioritized over other flits traversing along the same link. Finally, knowledge about congested paths is further used to save energy by aggressively varying link width [7] for non-critical links while not affecting performance significantly.

4.2.1 Critical Path Detection

As discussed earlier, a critical path is a combination of individual congested links. Detecting a critical path requires identifying: 1) the congested output ports in every switch, and 2) the input ports that are linked to congested ports of adjacent switches. This information should be gathered in a scalable manner to ensure feasibility across networks of varying sizes. Congestion in output ports can be tracked by having stall counters for every output port [44]. Every switch has counters to track the number of stalls due to congestion (i.e., output buffer full). These counters get reset at regular intervals of time (epoch) and the counter values are used to determine the congestion for each port in a specific epoch.

Traditional solutions to reduce congestion in on-chip networks approaches employ separate monitoring networks to communicate stall counters to adjacent switches and thus achieve regional congestion awareness [44]. We propose a novel approach to track globally congested paths without the need for additional communication overhead. Our approach uses tracker bits attached to regular flits to identify the input ports that are linked to congested ports of adjacent switches and thus establish congested paths.

Our approach requires 1 additional bit for bookkeeping in the header of every flit. Since the packet size is not usually a multiple of the flit size, 1 extra bit does not usually result in additional flits per packet. Tracker flits are regular flits whose tracker flag is set. Any regular flit can act as a tracker flit when its tracker flag is set. At the beginning of each epoch, a tracker flit

is sent along the most congested downstream port in every switch. This is done by setting the tracker flag of a flit that is about to traverse the selected link. The flag is cleared once it reaches the subsequent router. When a tracker flit is sent along an output port, the port is marked as critical by setting the bit corresponding to the port in a critical port bitmask that is introduced in every switch. Every switch also sends a tracker flit along its most congested downstream port on receiving a tracker flit. Hence, a chain of tracker flits are sent along the sequence of congested links tracking the critical path. Every time a tracker flit is triggered due to an incoming tracker flit, the input port from where the incoming tracker was received and the output port along which the outgoing tracker was sent are recorded in a list of critical input-output port pairs per switch. The critical port pair list is a list of input-output port tuples that are part of a critical path, where the information about the entire critical path is not available in any single switch but is distributed among the routers allowing the technique to scale with network size.

To prevent the same output port from being repeatedly selected in the same epoch, every switch maintains a bit mask of output ports along which tracker flits have been previously sent. When a new incoming tracker is received, an outgoing tracker flit is sent along the most congested downstream port that has not yet been tracked. To limit the number of ports being tracked, ports should have at least $k\%$ of the stalls of the port with highest congestion to be considered as a candidate, where k is a tunable parameter.

Fig.4.4 shows our critical path detection scheme in action. Consider a

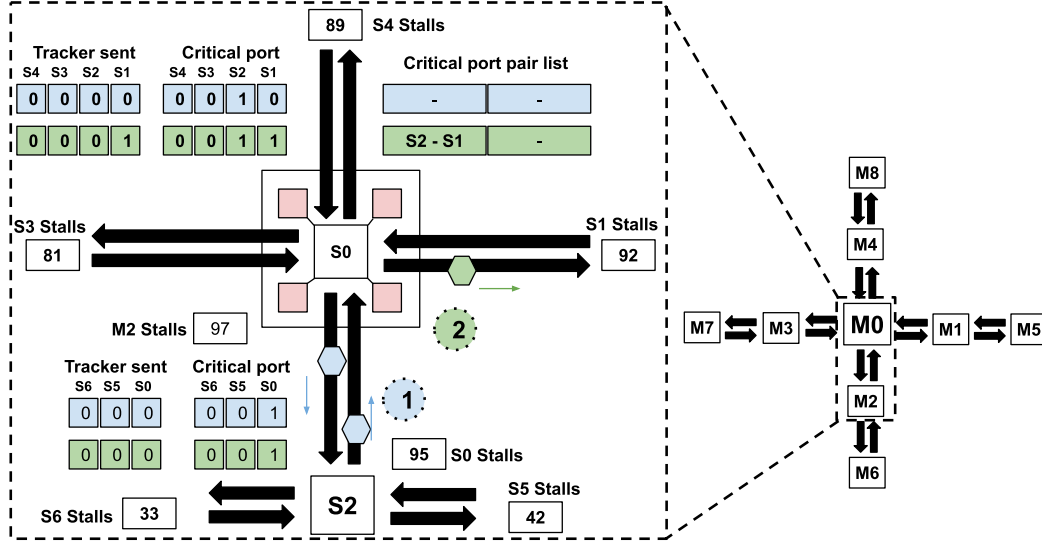


Figure 4.4: Critical path detection.

NUPIM system in a tree topology, specifically focusing on S0 and S2 switches. For the purpose of this explanation, flits from S5 and S6 to S2 are not considered. Initially, the critical port and tracker sent bitmasks for every port are cleared. Furthermore, the critical port pair list is empty and the congestion stalls incurred by every output port in the previous epoch are assumed to be available. The tracking threshold is assumed to be $k = 50\%$. At the beginning of an epoch (step 1 in the figure), modules M2 and M0 dispatch tracker flits along their most congested output ports, which are assumed to be along S0 and S2. The corresponding critical bits for the ports are set in each switch. As these tracker flits reach the destination switches, they in turn trigger additional tracker flits in those switches (step 2 in Fig. 4.4). In switch S0, a tracker flit is sent along S1 since it is the next most congested downstream

port that has not been tracked and has congestion above the threshold. The critical bit for the corresponding port is set and the input-output port tuple is inserted into the critical port pair list. In switch S2, no additional tracker flit is sent since the congestion along all downstream ports is lower than 50% of the highest congested port.

At the end of each epoch, the tracker sent bitmask and stall counters are cleared. The entries in the critical port pair list and critical port bitmask need to be cleared to prevent saturation, but such that the entries don't fluctuate in and out of the list across epochs. This is achieved by adding a h -bit hysteresis counter. List and bitmask entries are cleared only if they are not selected for 2^h consecutive epochs. A 6-bit hysteresis counter was determined suitable for our design empirically and is used in all our evaluations.

4.2.2 Path-based Prioritization

At the end of each epoch, every switch knows the the input-output port tuples that are part of the critically congested path(s) and the relative extent of congestion along each. The switch then uses this information to prioritize the flits traversing along the identified critical input-output port pairs during the VC and switch allocation stages. This in turn reduces congestion along the critical path at the cost of data moved along other paths. The manner in which the most congested ports in each switch are identified ensures that the input-output port tuples are inserted into the critical port pair list in decreasing order of congestion. In the proposed scheme, packets are prioritized according

to the order of their input-output port pairs in the list. Therefore, packets traveling along highly congested paths are selected over other congested paths. Since a strictly priority-based scheme is used, fairness and starvation can be potential issues. However, we did not observe any deadlocks in the applications evaluated. We experimented with fairness-enforcing schemes that select non-critical packets with some probability, but they did not perform better.

4.2.3 Divergence-based Prioritization

As mentioned earlier, not all types of data request traffic contribute equally to performance. In our scheme, we prioritize requests/responses with no divergence (i.e. instructions resulting in 1 coalesced memory access) over others traversing along the same link as they have higher chance of contributing to forward warp progress. Other prioritization heuristics are possible. For example, prioritizing the slowest among the divergent requests. However, identifying such requests is not simple and individual memory instruction latency does not significantly impact GPU performance as long as some threads can make progress and hide it. Identifying non-divergent memory requests is simpler and prioritizing them ensures that their corresponding warp will make progress. Note that our prioritization does not reduce memory divergence.

The coalescer within each SM generates memory requests for every memory instruction. In our scheme, the coalescer also sets a bit to indicate if the request is divergent or not as part of its header information. A 4-entry priority buffer is added per input port, which is exclusively used by

flits of non-divergent requests/responses. The priority buffer acts as small virtual channel with higher priority that allows non-divergent responses to bypass other traffic traversing along the same link. We reduce the size of normal VCs by the size of the priority buffer to keep the total buffer overhead constant. During VC allocation, non-divergent flits are allocated entries within the priority buffer if space is available. Otherwise, normal VCs are used similar to other traffic. When a port is selected during VC or switch allocation, the flits within the priority buffer are selected over those from normal VCs. Since the priority-buffer is only used by non-divergent requests/responses, it is vulnerable to under-utilization. However, since the applications evaluated have significant proportion of non-divergent traffic (54%), under-utilization of the priority buffer is not a significant problem. For a general case, it is possible to have designs where the priority buffer acts as normal VC when not in use, as previously done in [65].

4.2.4 Link Width Management

Knowledge about critical paths is further exploited for dynamic and adaptive optimization of energy consumption. In order to save energy, the width of interconnect links can be varied dynamically [7], where their bandwidth and energy changes proportionally. We adopt this approach to reduce links down to $1/2$, $1/4$ or as low as $1/8$ of their original width every epoch. The penalty for varying the width is set as $1\mu s$ [7], epoch length is set to trade off responsiveness versus per-epoch link management overhead. We selected

a length of $40\mu\text{s}$ (40x larger than the link adjustment time) to balance and amortize the overhead. Link width is determined differently for critical and non-critical links. For critical links, the width is adjusted in accordance with the intensity of data traffic. By contrast, for non-critical links, the width is adjusted aggressively to save energy and less importance is given to traffic intensity.

The link width for critical links is determined based on the flit rate and downstream traffic intensity. We obtain the flit rate by counting the number of credits received at each output port in an epoch. In addition, the congestion stalls for each output port reflect the traffic intensity downstream along the port. The traffic intensity is compared against the flit rate for each output port and the link width (and in turn the bandwidth and energy) is set to its maximum if the former is higher than the latter. Otherwise, the width is set such that the resulting bandwidth is a tier higher than the minimum required to support the data rate downstream. More specifically, let FR_p and T_p be the flit rate and traffic intensity (number of buffer full stalls) measured along port p , and let W and BW be default/maximum link width and the full bandwidth possible along any link. The link width W_p along a critical port p is then determined as:

$$W_p = \begin{cases} W, & T_p > FR_p \\ W, & FR_p > BW/4 \\ W/2, & FR_p > BW/8 \\ W/4, & \text{otherwise} \end{cases}$$

For non-critical links, the width is only determined by the downstream traffic intensity. If the traffic intensity along a port is more than a preset

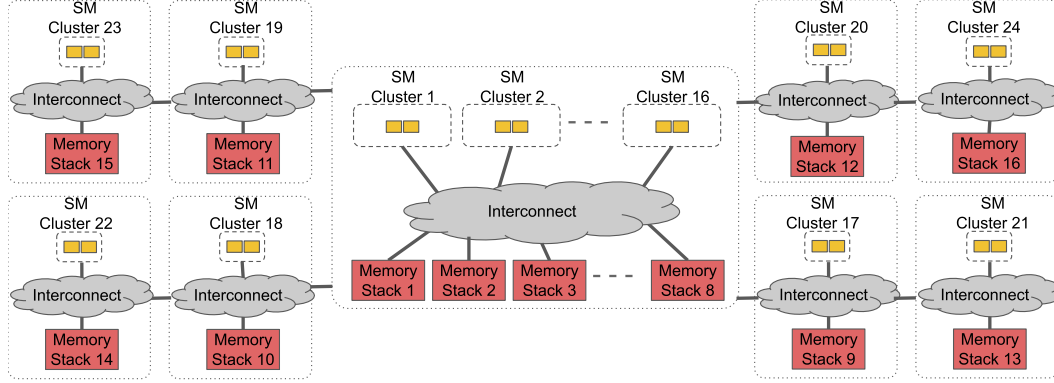


Figure 4.5: Simulation setup.

threshold w times that of the port with the highest stalls, then the width is raised to the next higher tier. Otherwise, the link width is reduced to the next lower tier until it is set to the lowest width. The threshold w is set to 0.9 for our evaluations. Let the width of W_p along a non-critical port p be initially set to its default/maximum W . Furthermore, let $T_{max} = \max_p T_p$ be the highest stalls incurred along any output port in a specific epoch. The link width is adjusted as:

$$W_p = \begin{cases} \min(2W_p, W), & T_p > 0.9T_{max} \\ \max(W_p/2, W/8), & \text{otherwise} \end{cases}$$

4.3 Experimental Setup

We evaluate the impact of our optimizations on performance and energy for a distributed NUPIM system architecture over a range of benchmarks. We use GPGPUSim v3.2.2 [18] coupled with GPUWattch [70] for modeling of SM power consumption and DRAMPower [25] to model the memory and I/O power. The simulator has been modified to replicate our system configurations

and congestion management scheme. We configure GPGPUSim to model 3D-stacked memory according to Hybrid Memory Cube (HMC) specifications [31] since HMC allows integration of units into a network. The capacity of each stack is the same and total memory is set to be equal to the memory footprint of each application evaluated. We assume that up to 4 GTX480 SMs fit within the logic layer of external memory stacks. Considering the area available in HMC logic layers [50] and the size of vault controllers [17] and the area for GTX480 SMs [2]. 4 SMs can comfortably fit within the logic layer [86] without violating thermal and power considerations [125]. While many NUPIM configurations are possible, we analyze a NUPIM system in which multiple modules having homogeneous memory-to-compute ratios are interconnected together. This configuration can easily be scaled with growing compute and memory requirements. Our simulation setup is shown in Fig. 4.5, and the exact simulator parameters for the configurations evaluated are summarized in Table 4.1.

4.3.1 Benchmarks

We use a range of benchmarks from Rodinia [28] and Lulesh [56] benchmark suites to evaluate our scheme. The complete list of benchmarks is shown in Table 4.2. Benchmarks are simulated with specified inputs to completion or for 1 billion instructions, whichever occurred first.

Since congestion in a NUPIM network is greatly dependent on application behavior and partitioning of data and compute, it is necessary to

Table 4.1: Simulation parameters.

Architectures		
System parameters	Total SMs	64
	Total DRAM stacks	Centrl:8, Ext.:8
	Centrl pkg SMs	32
	SMs/Ext. stack	4
SM Configuration		
Core configuration	1.4Ghz, GTO warp scheduler [18]	
Private L1 cache	32kB, 4-way, write through[48]	
Shared L2 cache	1MB, 16-way, write through [48]	
Interconnect		
Frequency	2.5Ghz	
Topology	Ext: Tree, Centrl pkg: Fully connected	
Switch	islip allocation, credit-based flow control [18], Min routing[53]	
Virtual channels(VC)	2 VC per port	
Bandwidth		
In-pkg bandwidth	160 GB/s per stack [48]	
Off-chip bandwidth	80 GB/s [48]	
Memory Stack		
Memory stack configuration	16 memory stacks, 16 vaults/stack, 16 banks/vault, 64 TSVs/vault [48]	
Scheduling policy	FR-FCFS	
DRAM Timing	DDR3 [83]	

understand application behavior and data mapping used for evaluating such systems. As mentioned earlier, applications can vary from having negligible to significant inter-thread sharing. To understand and differentiate application memory access patterns, we define the average page sharing of an application as $PS_{avg} = \frac{1}{N} \sum_{k=1}^N Tb_k$, where Tb_k is the number of unique threadblocks accessing page k and N is the total pages in the application's working set. Thus, PS_{avg} indicates the average number of unique threadblocks sharing a page. A PS_{avg} value of 1 indicates that no two threadblocks access same page. For applications with multiple kernels, the PS_{avg} of each kernel is weighted and

Table 4.2: Benchmarks.

Benchmark	Input
BFS [28]	1MW
CFD [28]	0.2M
Srad [28]	100, 0.5, 502, 458
Lulesh [56]	Default
Streamcluster (SC) [28]	Default
Backprop (BP) [28]	256k
Btree [28]	mil.tx
NN [28]	list5120k_512.txt
Pathfinder (PF) [28]	100000,100,20
Hotspot3D (H3D) [28]	512, 8, 100, power512x8, temp512x8

averaged based on the dynamic memory instruction count.

In some applications, a small set of pages are shared by a large number of threadblocks while the vast majority of pages are shared among a few. This results in a small PS_{avg} value despite having page sharing. Therefore, we also define the maximum page sharing $PS_{max} = \max_k Tb_k$ as the maximum number of threadblocks any page is shared with.

PS_{avg} and PS_{max} for different benchmarks are plotted in Fig.4.6. We distinguish between low-sharing applications that can be cleanly partitioned and high-sharing applications that share pages among threadblocks throughout this paper. In reality, applications usually don't fall into a single category completely. We therefore categorize applications based on their PS_{avg} and PS_{max} values. *NN*, *Hotspot3D*, and *Pathfinder* have small PS_{avg} and PS_{max} and are categorized as low-sharing while others are considered as high-sharing. Previous work [36] suggested that a locality-based data mapping suits regular benchmarks with low page sharing. Hence, while evaluating low-sharing

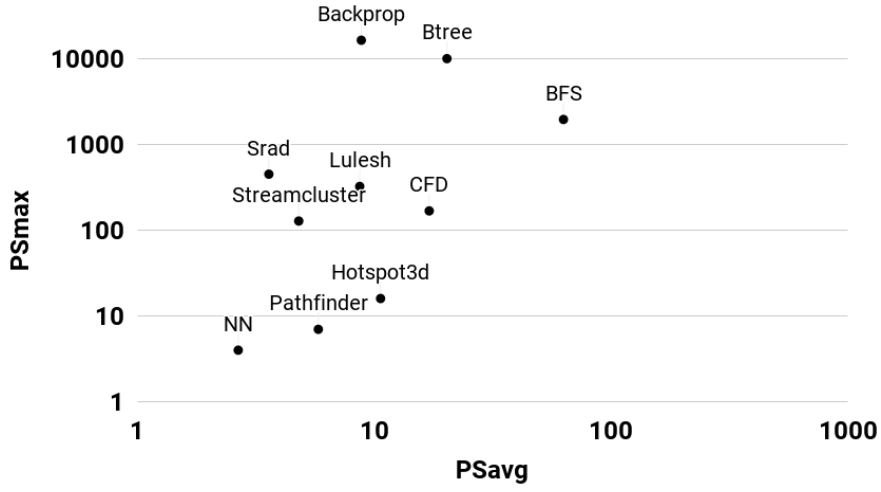


Figure 4.6: Average (PS_{avg}) vs. maximum (PS_{max}) page sharing of benchmark applications.

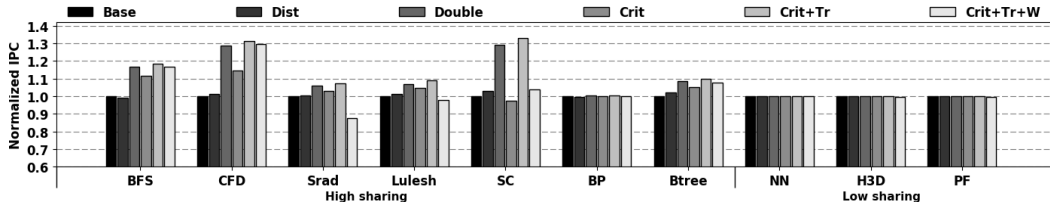


Figure 4.7: Performance across different congestion management schemes.

benchmarks, we map data based on the locality policy presented in [36]. We use pre-collected memory traces to statically place the pages into appropriate memory stacks. For irregular benchmarks with high page sharing, other work has demonstrated that interleaving data provides best performance [79]. For such high-sharing benchmarks, we thus interleave data across modules in chunks of 256 Bytes for our evaluations.

4.4 Evaluation

We apply our congestion management scheme on a homogeneous NUPIM configuration as described in Section 4.3 to evaluate its effectiveness in alleviating the adverse effects of off-chip congestion. We compare a baseline configuration without improvements (*Base*) to a configuration that performs only critical path prioritization (*Crit*), a configuration that performs both critical path and non-divergent traffic prioritization (*Crit+Tr*) and a setup that additionally performs dynamic link width management (*Crit+Tr+W*). We compare our schemes against distance-based (*Dist*) arbitration from [90] and a baseline NUPIM configuration with double the number of virtual channels and buffers (*Double*). Note that doubling the virtual channels introduces high area and power overheads (up to 74%) [124] that would violate NUPIM constraints and thus reduce the available budget for in-memory compute.

4.4.1 Performance Results

Fig. 4.7 shows the IPC of different benchmarks across different configurations normalized to *Base*. For high-sharing benchmarks, we can notice that *Dist* only achieves marginal performance improvement (up to 3%). This is because it does not account for variations in congestion between paths of the same length and treats all data as equally important. *Crit* achieves 5% average (and up to 15%) improvement, while *Crit+Tr* achieves 16% average (and up to 33%) improvement over baseline. *Crit+Tr* outperforms *Dist* by 10% on average (and up to 29%) and exceeds the performance obtained by *Double*,

indicating that the proposed congestion management scheme is effective in eliminating the adverse impact of off-chip traffic congestion while adding minimum overhead. Our link width management scheme ($Crit+Tr+W$) achieves lower benefits (6% on average and up to 30%) due to reduced link widths, but saves I/O and system energy as will be shown later. The extent of improvement observed across benchmarks varies depending on the impact of congestion on their performance. For example, congestion is more severe for *CFD* compared to *Backprop* and *CFD* thus benefits more from our proposed optimizations. The *Srad* benchmark experiences a drop in performance since it has multiple kernels that are smaller than the epoch length, where our scheme responds slowly.

Low-sharing benchmarks are not affected by our proposed scheme as they have low congestion. This is because data can be partitioned cleanly among modules for these benchmarks, thereby avoiding the congestion problem in the first place.

4.4.2 Energy Results

Fig. 4.8 shows the normalized system energy consumed by baseline, $Crit+Tr$ and $Crit+Tr+W$ configurations. We break system energy down into GPU, memory I/O, and other memory components. It can be noted that memory energy is dominated by its I/O component. Similar observations have been made in [51]. Our approach for link width management ($Crit+Tr+W$) consumes 78% of original I/O energy on average and reduces system energy

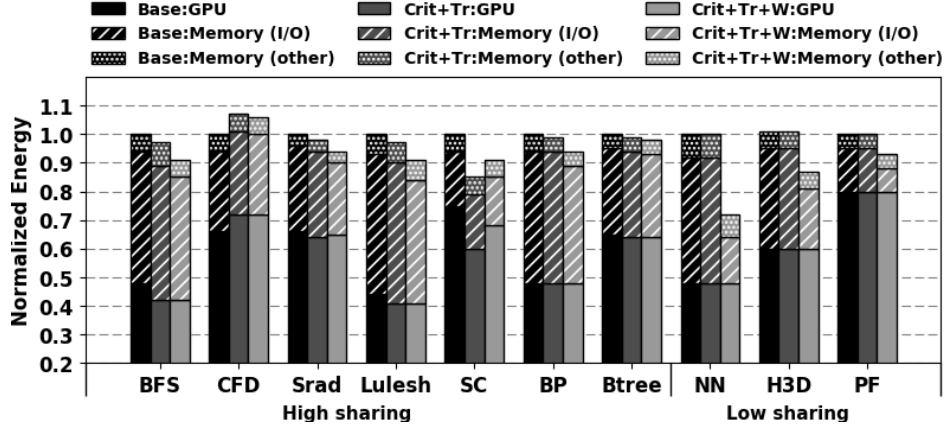


Figure 4.8: Energy breakdown.

by 8% (and up to 29%). For *Streamcluster*, *Crit+Tr* consumes less energy because of its higher performance and smaller execution time. Energy consumed increases for *CFD* because of increased utilization of GPU resources.

4.5 Summary and Conclusions

In this chapter, we showed that NUPIM systems are susceptible to tree saturation due to network congestion resulting in critically congested paths comprised of several individual links. We exploit the varying congestion across paths as well as the differing impact of data packets on performance due to memory divergence in NUPIM networks. We propose a novel, lightweight scheme that exploit such variations to mitigate congestion effects while reducing energy without incurring the overhead of additional links. Our scheme achieves on average 16% (and up to 33%) improvement over baseline and 10% (and up to 29%) improvement over other congestion mitigation schemes for

high-sharing benchmarks. Our scheme with link width management can save up to 29% system energy while achieving up to 30% performance improvement.

Chapter 5

Cacheline Utilization-Aware Link Compression

In this chapter¹, we present CUALiT [93], an approach for Cacheline Utilization-Aware Link Traffic compression that exploits both fine-grain spatial and temporal variation in cacheline utilization to reduce interconnect traffic. We first analyze the variation in spatial and temporal utilization across cacheline words for GPGPU applications. Words filled in a cache as responses are accessed immediately (critical), after a delay (non-critical) or remain unutilized during their lifetime. We then propose a novel mechanism that allows prediction of cacheline utilization for GPGPU applications. Predicted unutilized words are dropped from responses. Critical words are compressed using traditional methods while words with temporal slack in access time are coalesced across cachelines and compressed lazily to achieve higher compression ratios.

¹Contents of this chapter previously appeared in:
[93] Kishore Punniyamurthy, Shomit Das and Andreas Gerstlauer. “Cacheline Utilization-Aware Link Traffic Compression for Modular GPUs”, in International Conference on VLSI Design and Embedded Systems (VLSID), 2020. I proposed, designed and implemented the main ideas and collaborated with co-authors on writing this paper.

5.1 CUALiT Overview

Fig. 5.1 shows an overview of CUALiT. General purpose graphics processing unit (GPGPU) applications exhibit significant variation in temporal and spatial cacheline utilization [49, 98]. On the one hand, there are words that are accessed either immediately after the cacheline is brought in, or have some timing slack before they are first accessed (temporal utilization). On the other hand, there are words that are never accessed during their lifetime in the cache (spatial utilization). CUALiT uses a novel mechanism for fine-grained prediction of cacheline utilization and exploits the variations in utilization to reduce the amount of data moved. Predicting spatial utilization involves identifying cacheline words that will not be accessed during the lifetime of the cacheline, while predicting temporal utilization involves identifying whether words will be accessed immediately or with some slack once the cacheline response is filled. The words predicted to be unutilized during the lifetime of a cacheline are dropped from memory responses. Words with predicted slack in access times are batched across cachelines and lazily compressed. This enables us to harness data content locality [85] across different cachelines being accessed at the same time, resulting in higher effective compression ratios and subsequent traffic reduction. The critical words are handled in similar manner as in traditional compression schemes.

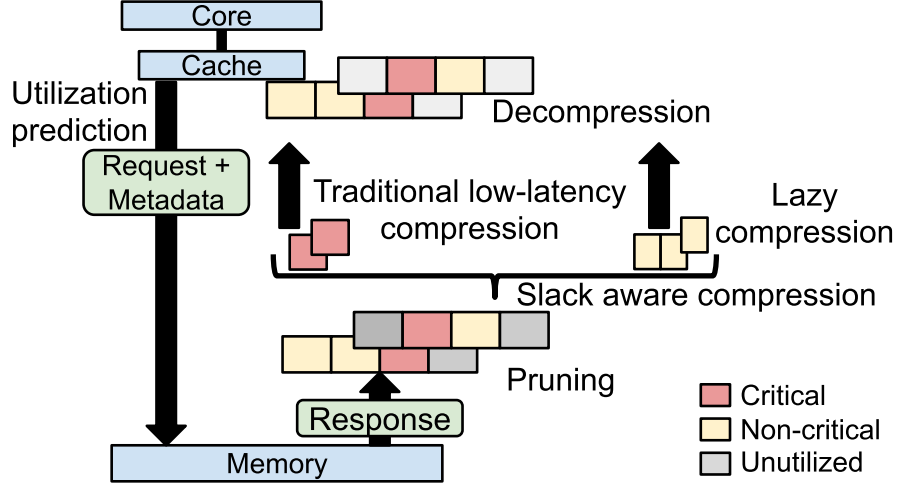


Figure 5.1: CUALiT Overview.

5.2 Variation in Cacheline Utilization

Fig. 5.2 shows the distribution of word utilization in cachelines across different GPGPU benchmarks. We track the time of first access to each word relative to the time the cacheline was brought into the cache. Not surprisingly, we can observe that the different words within a cacheline are requested at different times. Based on utilization, they can be categorized into three groups: *Critical words*: which are accessed immediately after cacheline becomes active in cache (31% on average). *Non-critical words*: which have a delay between cacheline filling and being accessed. We see that $\sim 10\%$ of words are accessed within 50 cycles of filling cacheline and $\sim 17\%$ of words have a slack of more than 100 cycles. *Unutilized words*: which are never accessed before the line gets evicted ($\sim 40\%$). In conclusion, we see that cachelines exhibit variation in both spatial and temporal utilization. Such variations can be leveraged to

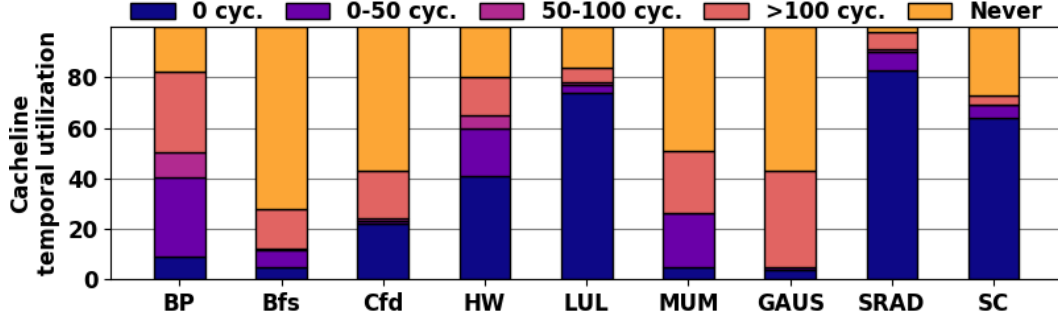


Figure 5.2: Cacheline utilization.

reduce the data movement. Spatial variation can be used to prune the unutilized words in memory responses while the slack available due to temporal variation can be used to increase data compression opportunities. However, there are challenges in doing so. 1) *Accurate utilization prediction*: accurate prediction of utilization is vital since an aggressive incorrect prediction will result in additional accesses and performance loss while conservative prediction will limit benefits. Prior works have proposed spatial utilization prediction but they provide coarse granularity prediction [98] or are intended for CPU applications [64] and may not work as well for GPU applications, as explained later. We address this by introducing a novel granularity predictor which can predict utilization and prune the unutilized words from memory responses. 2) *Exploiting slack to reduce data movement with minimum overhead*: traditional compression schemes aim to compress cachelines under stringent latency constraints (latency-sensitive). Conversely, exploiting temporal variation requires schemes which optimize for compression ratio with long compression/decompression latency and low hardware overhead. We propose a lazy batch com-

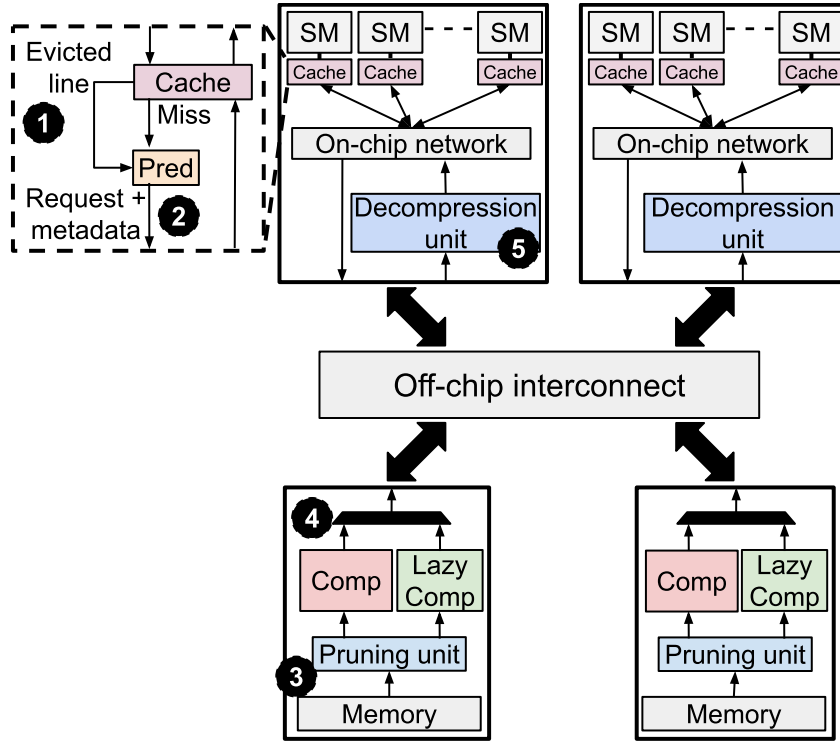


Figure 5.3: CUALiT Block Diagram.

pression scheme which delays responses in order to find and harness data content locality across cachelines and thereby achieving higher compression ratio by exploiting the slack.

5.3 CUALiT

CUALiT combines granularity prediction, utilization-aware pruning, and compression to reduce the amount of data moved across the interconnect; thereby saving bandwidth and energy consumption while improving performance. Fig. 5.3 shows the overview of our scheme.

Our scheme begins with an accurate prediction of the utilization of the requested cacheline during its lifetime in the cache. To do so, the prediction mechanism needs to be trained. Caches are augmented with book-keeping data to track the utilized words within the cacheline. Every time a cacheline gets evicted, its utilization is used to calibrate the granularity predictor ❶. When a memory read request is generated due to a cache miss, the utilization of the cacheline is predicted and then the request along with this metadata is sent across the interconnect to the memory ❷. Once the responses have been accessed from the memory, they are pruned ❸ and compressed based on their criticality ❹ before sending into the interconnect. Finally, the compressed responses are decompressed within each GPU module before being filled into the cache ❺. The metadata included with the responses specify the valid words within each response. We now delve into the individual components.

5.3.1 Utilization Prediction

Benefit from CUALiT hinges on reliable and fine-grained prediction of cacheline utilization. In a GPU, since the threads are mostly identical, it is reasonable to expect the same instructions across warps to behave similarly. We validate this by tracking the variation in the utilization of cachelines accessed by individual instructions (program counter (PC) specific) and plot the standard deviation (SD) of utilization (in bytes) averaged across all PC for different benchmarks as shown in Fig. 5.4. The maximum and minimum SD observed are plotted as error bars. We see that for most benchmarks,

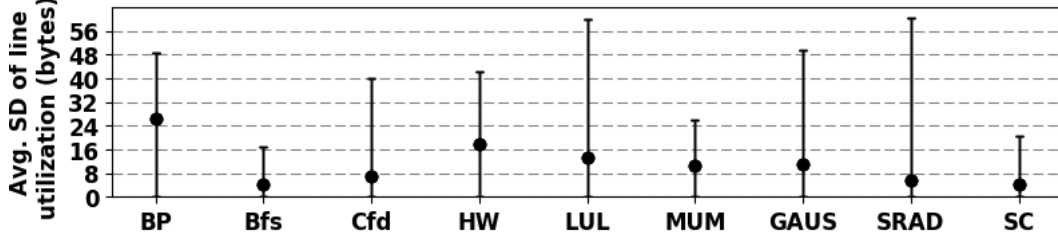


Figure 5.4: Access granularity variation.

the variation in utilization of cachelines accessed by same PC is $\sim 10\text{B}$ or less for a 128B cacheline, indicating that instructions with same PC have similar utilization in most cases. We exploit this observation in our scheme.

Prior works have proposed fine-grained utilization prediction by selecting useful words as the set of all words between touched words seen in history, while utilizing critical word index [64]. Such schemes may not work well for GPU applications. Consider the example shown in Fig. 5.5 consisting of 4 warps (W_0 - W_3) each accessing 2 words in cacheline A (32B cacheline) on executing instruction with program counter (PC) p . Consider a case where cacheline A is filled in to the cache and gets evicted by the time only 2 warps (W_0 - W_1) have completed their accesses. Prior PC-based spatial prediction solutions [64] use the utilized words to predict the mask for next access, which is $\langle 0:3 \rangle$ in this case. Next consider that warp W_2 executes PC p , now the prediction will be either $\langle 0:3 \rangle$ in which case the prediction is wrong, or if the predictor uses critical word index then no prediction will be made and result in conservative access. Similar miss predictions may happen when warps access other cachelines while executing PC p instruction.

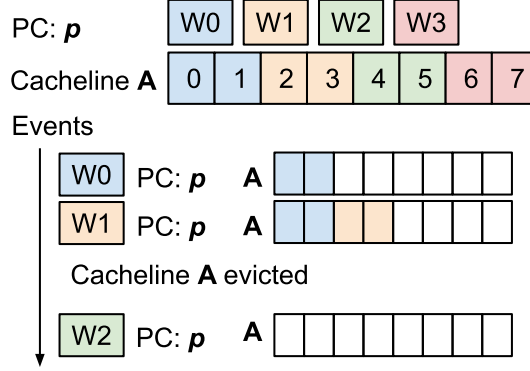


Figure 5.5: Utilization prediction example.

We propose a novel prediction mechanism capable of reliably predicting the number of words (4B) which will be accessed from the requested cacheline before its eviction. The predictor outputs the granularity for the access and not the pattern of useful bytes within cacheline. Consider accessing 128B cacheline at different granularities for example, 1x128B, 2x64B, 4x32B and so on forming a hierarchy. The predictor predicts the optimal granularity (4/8/16/32/64/128 bytes) and only the predicted granularity size segments containing the requested words are moved. For example, if the coalesced request for the cacheline requires the 0th and 5th word and the predicted granularity is 16B. Then all the words within the 16B segments containing requested words (i.e., words 0-7) are brought in response. For the example shown earlier, our prediction mechanism will learn the granularity to be 4 words and would predict the 4 word chunk containing the requested words (i.e., $\langle 4:7 \rangle$) when W_2 executes.

Fig. 5.6 shows the working of the granularity predictor. It is a PC-

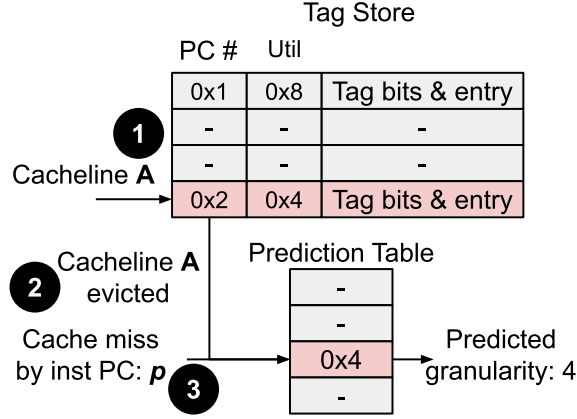


Figure 5.6: Granularity predictor

based predictor consisting of a prediction table and metadata tracking as part of the cache tag store. The prediction table is indexed by PC bits and holds the predicted granularity for the PC. The tag store is extended to store bits from PC making the request and to track the cacheline words accessed.

Predictor training is a two step process. Every time a new cacheline is allocated ①, bits from PC of the corresponding instruction are stored as part of tag entry. We use bits <10:7> from PC. On a line eviction ②, the PC bits and utilization metadata are used to update the prediction table. The PC bits are used to index into the table and the corresponding entry is updated with the new granularity (4/8/16/32/64/128 bytes) depending on the number of words accessed in the evicted cacheline. Since the update to the predictor table happens on eviction, it does not affect the critical path and therefore, does not add to the memory latency. We found that predictor table with 16 entries worked well for our experiments. Finally on a cache miss ③, bits from

the requesting PC are used to index into the granularity prediction table to obtain the granularity of request. If no prediction is available, full cacheline is selected by default. The predicted granularity along with the bytes required by the coalesced request is used to determine the actual utilization of the cacheline. The total overhead of metadata sent along with each request is 35 bits (32b bitmask + 3b for granularity).

The above scheme predicts the spatial utilization of cachelines and not its temporal utilization. We mark the words required in the coalesced request as critical and others as non-critical. We do so for simplicity as tracking variation in slack across words will require a complicated design and additional book-keeping. As results (in a later section) show, this simple differentiation is sufficient to get benefits.

5.3.2 Spatial Utilization-aware Pruning

The predicted spatial utilization is used to prune the unutilized words from the responses. This is simply done by dropping the words from the memory responses and updating the metadata to indicate the valid words.

Fig. 5.7 shows an example of the same with two 32B responses with predicted granularity of 16B and 32B respectively. The unutilized words (shown in gray) from memory responses are cropped by the pruning unit. Instead of pruning the memory response, it is possible to read only the required bytes, resulting in potential lower memory access latency. However, in our experiments we read the entire cacheline from memory. Unlike spatial utilization,

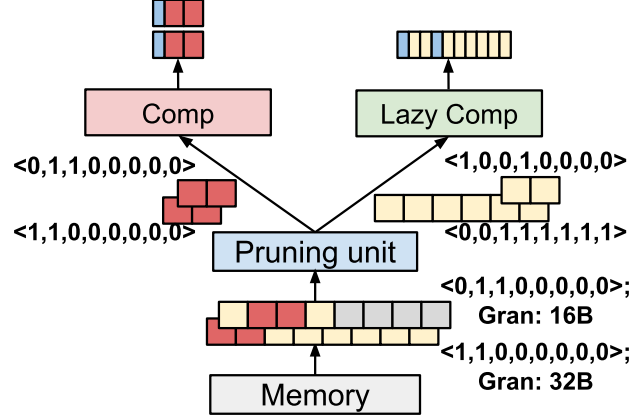


Figure 5.7: Pruning and compression.

exploiting temporal utilization is not trivial since all the words are required but have different slack time. Traditional compression schemes optimize for latency and treat all words equally except for their data content. On the contrary, we need schemes which optimize for compression ratio at the expense of long latency.

5.3.3 Slack-aware compression

We perform slack-aware compression where words are compressed differently based on the available slack (criticality). As mentioned earlier, words are marked critical if they are required by the request causing the cache miss (using metadata in each request). In our example (Fig. 5.7), words $\langle 0:3 \rangle$ of the first response are predicted to be utilized based on its granularity with words $\langle 1:2 \rangle$ as critical while the second response has entire cacheline predicted to be utilized with words $\langle 0:1 \rangle$ as critical. Critical words which are required immediately are compressed using the traditional scheme by the *Comp* unit. We

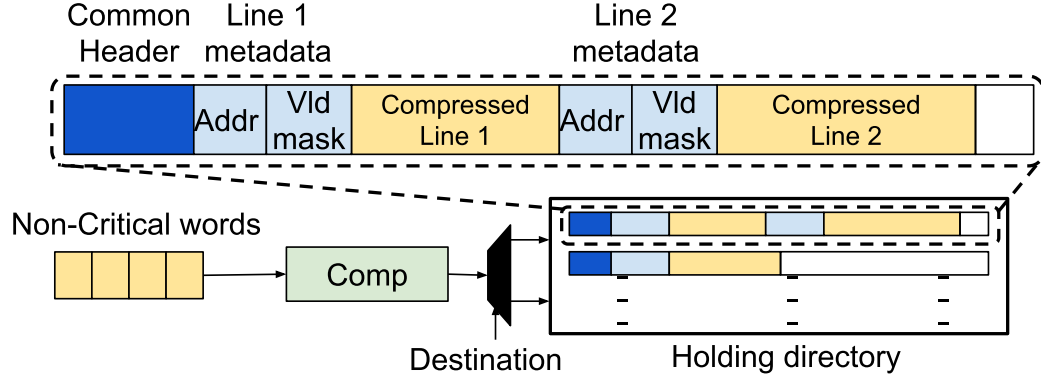


Figure 5.8: Lazy batch compression.

use C-PACK as our standard compression scheme but any other scheme can equally be used. The non-critical (and any mispredicted critical) words are batched together and compressed lazily. Each compressed data also contains metadata (a bit mask) indicating the number and position of valid words.

Fig. 5.8 shows the *lazy batch* compression unit in detail. Lazy batch compression tries to compress non-critical words across cachelines with an aim to achieve higher compression ratio. It uses a library based compression scheme (C-PACK) to compress words while reusing dictionary across cache-lines to exploit data-content locality and thereby achieving higher compression ratio. To be able to compress across cachelines, the responses are delayed and batched together by storing them in the *holding* buffer for a certain duration (a parameter). Delaying and batching responses to compress more words allows this scheme to trade-off slack for compression ratio. In a multi-module GPU system memory responses might be destined to different modules and should be batched separately. We do so, by having a *holding* directory with

a buffer for each destination. The buffer size is set at 256B to limit overhead. When a new response with slack reaches the unit, the appropriate buffer is selected based on the destination, the incoming words are compressed using the destination-specific dictionary and inserted into the buffer if space available. For every new cacheline inserted, its valid mask (32b) and address (48b) are also stored. The responses are dispatched to their destinations once the delay duration is over or the buffer is full. If there is insufficient space in buffer, the words are treated the same as critical words and dispatched immediately without delay. We use 32B dictionary per destination and delay duration of 200 cycles to keep the overhead nominal.

5.3.4 Decompression

The compressed responses received from the interconnect are decompressed at each module before filling them into caches. Since lazy batch compression compresses lines to same module (not same cache), decompression is done at the I/O interface of each module as shown in Fig. 5.3. The metadata included with every response received contains information about the number and position of valid words. Fig. 5.9 shows an example of decompression in action. *Metadata1* provides the address (*Addr1*) and valid mask ($\langle 0,1,1,0,0,0,0 \rangle$) for critical compressed line which is used for its decompression. Lazy batch compressed responses with multiple cachelines are decompressed one at a time using the metadata pertaining to individual cacheline. The decompression mechanism is same as the original scheme[29] except the

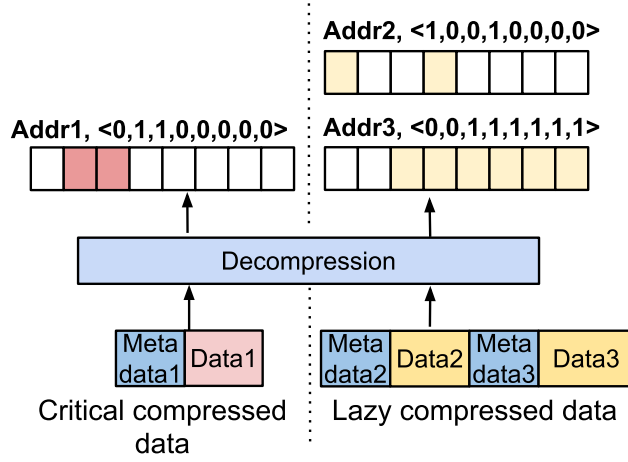


Figure 5.9: Decompression.

number of words are determined by valid mask. The extracted responses are then sent to their respective caches within each module where they are filled into the cache along with their active word masks which are stored in tag arrays. We add compression/decompression latency overhead of 10 cycles[29]. If a read request wants a word which is not active in the cache, it is treated as a miss. Since this work aims at reducing interconnect traffic, caches in our experimental setup do not support variable cacheline size. Therefore, the effective cache capacity remains the same.

5.4 Evaluation

We evaluate our scheme's impact on link bandwidth and consequent system performance and energy, on a multi-module system where each module consists of streaming multiprocessors (SMs) and in-package memory in the form of Hybrid Memory Cubes (HMCs) as shown in Fig. 5.10. We use

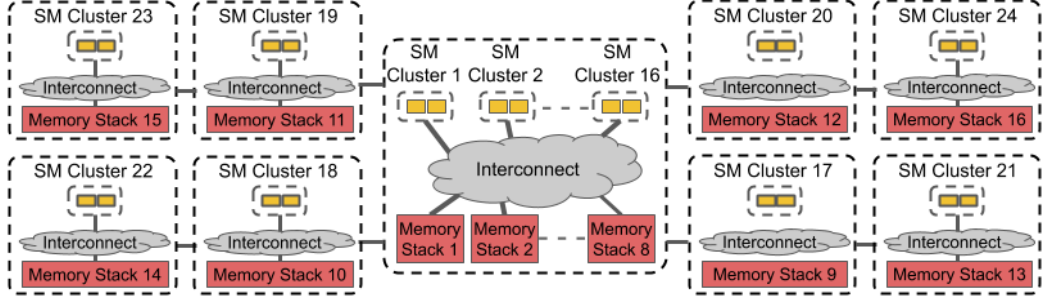


Figure 5.10: Simulated system.

Table 5.1: Benchmarks.

Benchmark	Input
Backprop (BP) [28]	256k
Bfs [28]	1MW
Cfd [28]	0.2M
Heartwall (HW) [28]	test.avi, 5
Lulesh (LUL) [56]	Default
Mummer (MUM) [28]	NC_003997.q25bp.50k.fna
Gaussian (GAUS) [28]	matrix512
Sradv1 (SRAD) [28]	100, 0.5, 502, 458
Streamcluster (SC) [28]	Default

GPGPUSim v3.2.2 [18] for our experiments. GPGPUSim is coupled with GPUWattch [70] for modeling SM energy consumption and DRAMPower [25] to model the memory and I/O power. We configure GPGPUSim to model 3D stacked memory according to HMC specifications [31] since HMC allows integration of units into a network. The capacity of each stack is the same and total memory is set to be equal to the memory footprint of each application evaluated. The benchmarks evaluated are shown in Table 5.1. The simulator parameters are summarized in Table 5.2. The benchmarks are simulated to

Table 5.2: Simulation parameters.

System Configuration	
Total SMs	64 (16 clusters with 4 each)
Memory stacks	16
SM Configuration	
Core configuration	1.4Ghz, GTO warp scheduler[18]
Private L1 cache	32kB, 4-way, 128B cacheline[48]
Shared L2 cache	1MB, 16-way, 128B cacheline[48]
Interconnect	
Frequency	2.5Ghz
Topology	Ext: Tree, Central pkg: Fully connected
Bandwidth	
In-pkg bandwidth	160 GB/s per stack [48]
Off-chip bandwidth	80 GB/s [48]
Memory Stack	
Stack configuration	16 vaults/stack, 64 TSVs/vault [48]
Scheduling policy	FR-FCFS
Timing	DDR3 [48]

completion or till 1 billion instructions whichever occurred first.

5.4.1 Utilization Predictor

We first compare the effectiveness of our prediction mechanism against PC-based prediction scheme used in [64]. We limit to PC-based schemes so that the patterns learnt from a warp can be used across all the warps. The responses are pruned to bring only the predicted bytes. No compression is applied. Fig. 5.11 shows the off-chip traffic observed using proposed granularity prediction and prior work. We can observe that our scheme performs better with average 22% (up to 56%) lower data movement. The prior work

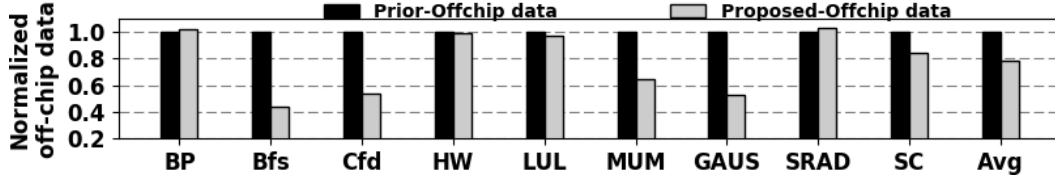


Figure 5.11: Prediction mechanism performance.

uses critical word index on top of PC for indexing, resulting in conservative behavior with lower benefits. Our aggressive fine grained prediction results in higher data movement for *Backprop* and *Sradv1* but the increase is small ($<4\%$).

5.4.2 Off-chip Traffic

We compare the reduction in actual off-chip data (not including padding bytes) moved under different configurations as shown in Fig. 5.12. The baseline (*Base*) has no compression and accesses complete cachelines. *Comp* uses traditional compression (C-PACK). *CompMask* uses granularity prediction and compresses the pruned data using traditional compression scheme and *CompMask-Lazy* uses granularity prediction and employs slack-aware compression on pruned response. *RQ* and *RSP* indicate data traffic type: memory request and response respectively. We can see that on average *Comp* reduces 28% of total offchip data traffic, *CompMask* gets higher (33%) reduction indicating spatial utilization can be exploited to obtain bandwidth saving. The effectiveness of lazy batch compression is evident from the average 42% traffic reduction obtained by *CompMask-Lazy*. Moreover, *CompMask-Lazy* reduces

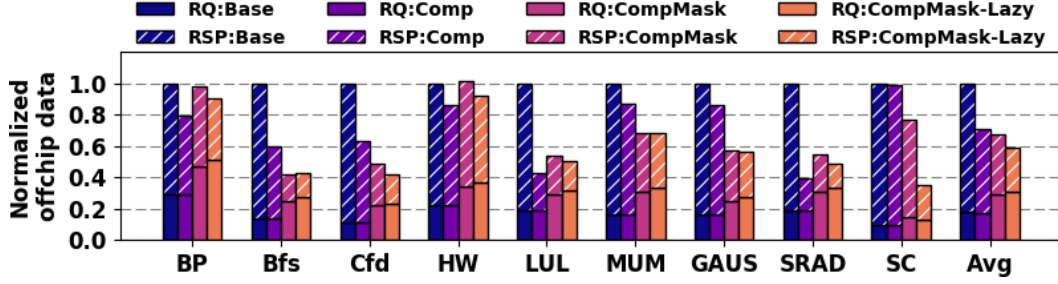


Figure 5.12: Normalized off-chip traffic.

12% higher off-chip traffic than *CompMask* indicating the additional compression opportunities captured by our lazy batch compression scheme. The request traffic for all configurations is in general higher than *Base* (except for *Comp*) due to the metadata (requested word mask (4 bytes) and granularity (1 byte)) overhead. This overhead is often compensated by the reduction in the response sizes resulting in net reduction in traffic. However, there are some cases when the net traffic increases e.g., *Heartwall* and *Backprop* for *CompMask*. HMC uses packet switch interconnect and will have overhead of padding bytes if response sizes are not multiple of flit size. Schemes like [60] can be deployed to eliminate the same but has not been implemented in this work.

5.4.3 Energy Results

We now compare the system energy savings obtained with different configurations. I/O link energy saving methods have been proposed by *X. Jian et al.* [51], which save energy at the cost of bandwidth. Implementing these solutions is not the focus of this work and we assume that I/O component of

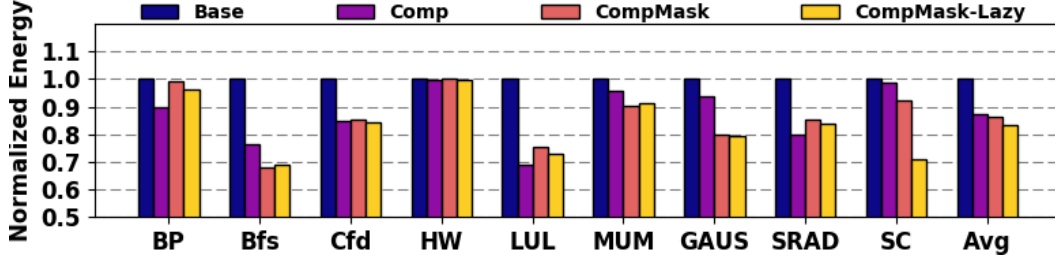


Figure 5.13: Normalized Energy saving.

link energy is reduced proportional to the reduction in link traffic. Fig.5.13 shows the normalized system energy across different configurations. Comparing the energy savings obtained we see that *Comp* reduces system energy by 12% while *CompMask* and *CompMask-Lazy* get 14% and 17% energy saving respectively. *CompMask-Lazy* reduces 4% (up to 24%) more system energy than *CompMask* scheme.

5.4.4 Performance Results

Fig. 5.13 shows the normalized IPC across different configurations with 68B and 32B flit sizes. We see that with flit size of 68B, the reduction in traffic is not reflected in IPC benefits in general, this is because, using a flitsize of 68 bytes diminishes the benefit obtained due to padding overhead and lower bandwidth pressure. The response size is 128 bytes (data) + 8 bytes (header) resulting in 2 flits per response. *Cfd* and *Streamcluster* still benefit from traffic reduction. On average *Comp* provides 5% (up to 32%) performance improvement, while *CompMask* and *CompMask-Lazy* provide 7% (up to 48%) and 16% (up to 2x) improvement respectively. *Streamcluster* benefits significantly

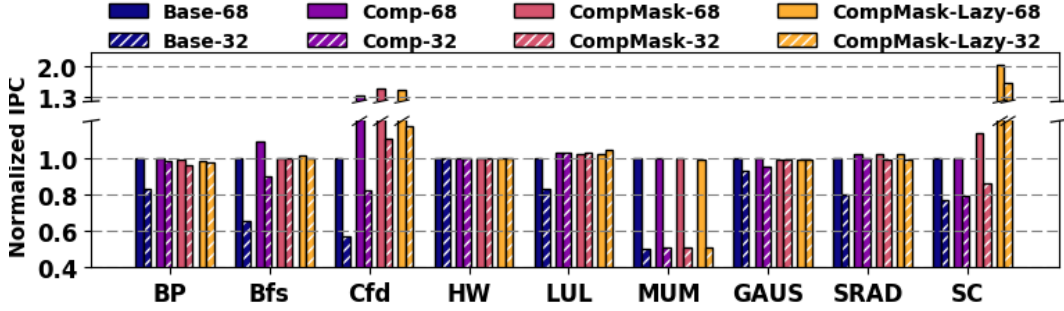


Figure 5.14: IPC with 68B and 32B flitsizes.

due to its high data-content locality. We study the performance impact under smaller flit sizes. We reduce the flit size from 68B to 32B while keeping other network parameters same, this reduces the padding overhead and increases the bandwidth demand since every response now requires 5 flits. Fig.5.14 shows the IPC under different flit sizes normalized to 68B (*Base-68*). We can see that performance drops for many benchmarks due to increased bandwidth demand with 32B flits and in most cases our scheme is able to accommodate the increased bandwidth demand while achieving similar or higher performance. We find that under higher bandwidth demand *Comp* provides 20% average improvement while *CompMask* and *CompMask-Lazy* provide 23% and 36% improvement respectively.

5.5 Summary and Conclusions

In this chapter, we presented CUALiT - Cacheline Utilization Aware Link Traffic Compression. The scheme exploits the variation in temporal and spatial utilization to achieve higher compression ratio. CUALiT uses a novel

granularity predictor to predict the cacheline utilization at a very fine granularity. This information is then used to prune unutilized words, batch and lazily compress words with slack to harness data content locality across cachelines. Our evaluations show that CUALiT achieves up to 24% lower system energy and on average 11% (up to 2x) higher performance over a compression-only scheme. Furthermore, under high bandwidth demand, the scheme provides 16% more performance improvement, demonstrating its effectiveness to reduce bandwidth demand.

Chapter 6

Thread Address Footprint Estimation

In this chapter¹, we present TAFE [96], a framework for accurately estimating both static, data-independent and dynamic, data-dependent *thread address footprints (TAFs)* prior to thread and kernel execution. Data-dependent patterns require tracking of input data, which, if not carefully done, can result in overhead proportional to the input problem size. Our framework relies on a combination of easily-derivable code annotations to capture regular, static access patterns with dynamic tracking of input data dependencies. We propose both pure software as well as hardware-assisted mechanisms for lightweight dependency tracking with minimal overhead. Furthermore, we develop compiler support to automate code annotation and minimize programmer overhead. We show that with this information, the relation between thread IDs and addresses accessed (the TAFs) can be accurately estimated prior to the launch of a GPU kernel. TAFs enable the OS/runtime to derive the spatial sharing of data across threads/warps/threadblocks without explicit input from users. As

¹Contents of this chapter previously appeared in:
[96] Kishore Punniyamurthy and Andreas Gerstlauer. “TAFE: Thread Address Footprint Estimation for Capturing Data/Thread Locality in GPU Systems.”, in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2020. I proposed, designed and implemented the main ideas and collaborated with Dr. Gerstlauer on writing this paper.

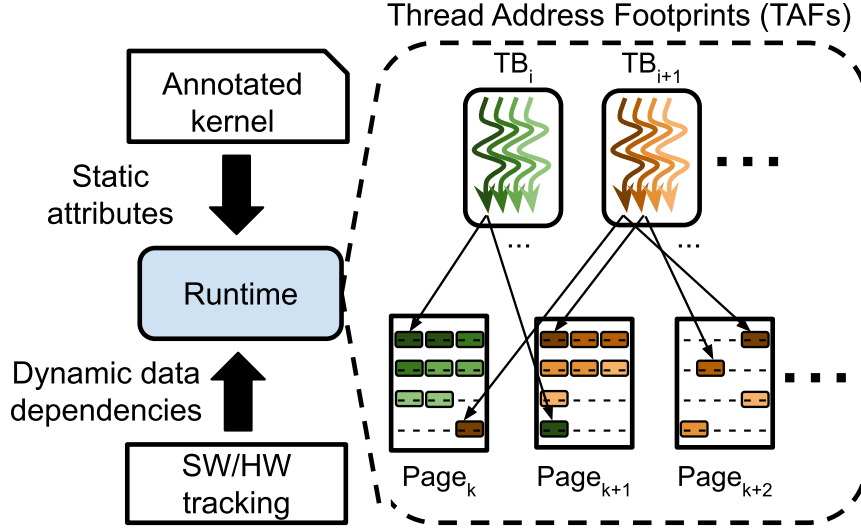


Figure 6.1: TAFE overview.

a case study, we demonstrate that using this information, an OS/runtime can achieve better data/thread co-location in NUMA GPU systems. Such TAFs can in turn also be used for prefetching pages in unified, e.g. CUDA-managed memory systems, transformations of irregular (scatter/gather) patterns into regular remote memory accesses via local page caching and re-arrangement, or reducing coherency traffic [126, 115, 19].

6.1 TAFE Overview

An overview of TAFE is shown in Fig. 6.1. We focus on arrays as representation of arbitrary data structures contiguously laid out in virtual memory, which are the pre-dominant types of data structures in GPGPU applications. We distinguish between array accesses that are either: 1) *Thread ID-dependent*,

where final array addresses are a function of the thread index only, or 2) *Input data-dependent*, where addresses accessed depend on both the thread ID and external input data passed into the thread by the host. The TAFs for the former can be derived from the coefficients of thread indices and offsets used to compute array indices. This information is readily available in the code and can either be programmer- or compiler-annotated. TAFE uses them to determine the exact addresses accessed by each thread and thereby any spatial sharing of data without needing the programmer to explicitly specify it.

For input-data-dependent accesses, the TAF is typically established through one or more levels of nested and indirect accesses, where an externally initialized primary array is accessed in a specific pattern, and the result is used to compute the index into secondary arrays. Obtaining the TAF for such accesses requires establishing the relationships between 1) the thread ID and the primary array index accessed, 2) the primary array index and the value stored in it, and 3) the primary array value and the secondary array address being accessed. The first and last can be obtained statically in the same way as for purely thread ID-dependent accesses. Tracking 2), however, can only be done dynamically. In order to know the primary array values before thread or kernel execution the primary array values must be known before the thread or kernel launches. We therefore track primary array values and relate primary indices to values stored in them during their initialization. Directly capturing this relationship is, however, infeasible. It would incur overhead that is proportional to size of the primary array itself. Our framework proposes

mechanisms to capture data-dependent TAFs and their required information with low overhead. Once a primary array is initialized, TAFE uses the tracked primary array values along with the static information from 1) and 2) to compute the TAF for data-dependent accesses.

TAFE estimates the individual array indices that will be accessed by each threadblock. As such, it is capable of tracking addresses at array element-size granularity. While the addresses and TAFs can be estimated per thread, we restrict TAFE to threadblock granularity to limit overhead. Different optimizations will require TAFs to be estimated at different granularity. In this dissertation, we evaluate TAFE for data/threadblock co-location, where TAFs are estimated at page granularity for each threadblock. Alternative optimizations, such as cache replacement policies would require TAFs at threadblock to cacheline granularity, which is supported by TAFE.

6.2 Thread Address Footprints

In order to capture data/thread locality, it is necessary to estimate the address footprint of different threads. In general, threads can access arrays with arbitrary levels of nesting as shown in Fig. 6.2. In the first level (level 0), thread IDs are used to determine the indices accessed in the first array. In each subsequent level, values stored in primary arrays are used to compute the indices into the next secondary array. At each level l , the indices accessed in different arrays can be captured and represented by relations $R_{l,n} \subseteq (values, index), 0 \leq n < N_l$ that map thread IDs or one or more pri-

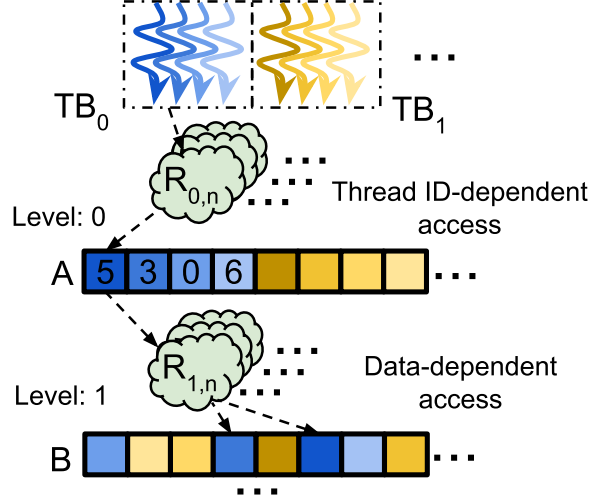


Figure 6.2: Generic array access patterns.

mary array values to sets of secondary array indices. Each relation is specific to one secondary array, where every array can be accessed using one or more relation. Fig. 6.2 shows such relations $R_{l,n}$, where l is the nesting level and n is relation ID within level l . The application-specific relations $R_{l,n}$ determine the access patterns and can range from simple identity functions to arbitrary code. TAFs can be established by propagating thread IDs through relations across nesting levels.

In TAFE, we estimate the TAFs of all threads accessing the same data structure by capturing and successively evaluating individual relations throughout all nesting levels. If multiple relations are associated with an array, we assume that all relations contribute towards all nested accesses. In practice, for every threadblock we find their mapping into sets of array indices and use that to determine the addresses and pages accessed. In the following,

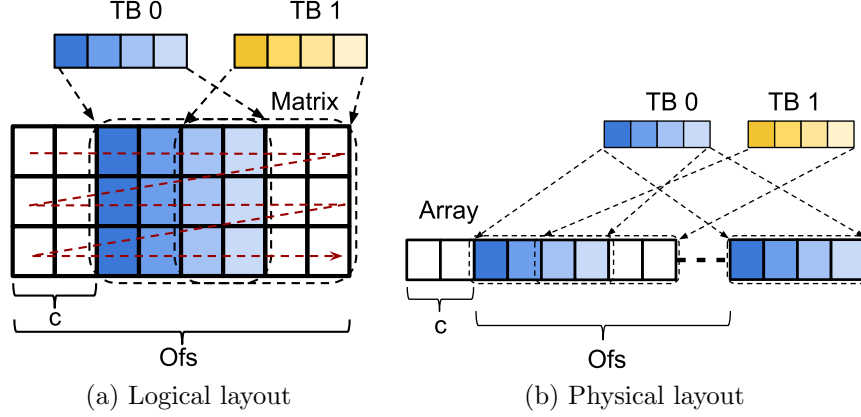


Figure 6.3: Thread ID-dependent access.

we identify the information and overhead required to capture TAF relations for commonly found access patterns.

6.2.1 Thread ID-Dependent Accesses

At first level ($l = 0$), the relations ($R_{0,n}$) take thread IDs as input and define a mapping to array indices for each thread. A commonly seen pattern include linear relations including iterative accesses within loop control structures of the following form:

$$R_{0,n}(tid) = \{\mathbf{m} \cdot \mathbf{ID}(tid) + c + i \cdot ofs \mid 0 \leq i < I\},$$

where, $R_{0,n}(tid)$ is the set of all indices accessed by thread ID tid for a given array, $\mathbf{m}$ is a coefficient vector, $\mathbf{ID}(tid)$ is a function that maps a global thread ID into multi-dimensional block and thread indices, c is a constant, I is the iteration count of the loop and ofs is the offset added to the index in each loop iteration. In the simplest case, the array index is computed as an affine func-

tion of the thread ID (N and ofs are set to 0, i.e. each thread only accesses one array element). The above relation represents patterns where threadblocks access sections of an array separated by an offset iteratively as shown in Fig. 6.3. Such patterns are commonly seen in linear algebra, dynamic programming or sliding window applications [101].

6.2.2 Input Data-Dependent Accesses

Nested array accesses ($l > 0$) use data stored in primary arrays to determine the secondary indices accessed. There can be one or more primary arrays supplying the input values. There are two common variants of nested relations $R_{l,n}$:

Indirect In this variant, the result of one memory access (primary access) is used to compute the address for the next memory access (secondary access). The TAF relation for such patterns with single primary array is:

$$R_{l,n}(r) = \{m \cdot P[r] + c\}.$$

Here, P is the primary array, r is the primary array index, m is a coefficient, c is a constant, and $R_{l,n}(r)$ maps indices of P into the set of secondary array indices indirectly accessed. Indices r are in turn determined by relations $R_{l-1,q}$ attached to P in nesting level $l-1$, going recursively back all the way to original thread IDs. This captures patterns like $B[A[i]]$ (scatter/gather operations), where the value stored in array A is used to index into array B as shown in Fig. 6.4a. Such patterns can be seen in algorithms such as link-list traversals,

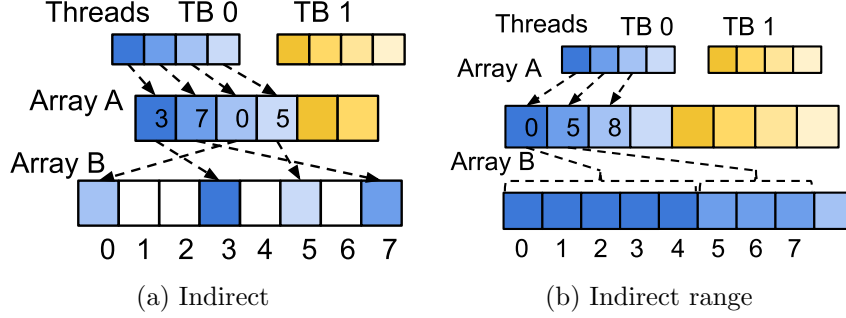


Figure 6.4: Data-dependent access.

hash search [47], radix sort [46] and COO format-based sparse matrix vector multiplication [33].

Indirect Range Alternatively, primary arrays can provide bounds of a continuous range of indices accessed in a nested array. The TAF relation for patterns where two primary arrays provide the start and end indices for accessing a secondary array is as follows:

$$R_{l,n}(r_s, r_e) = \{k \mid P_s[r_s] + c_s \leq k < P_e[r_e] + c_e\}.$$

Here, P_s and P_e are the primary arrays containing starting and ending indices (can be identical), r_s and r_e are indices of corresponding start and end elements, c_s , and c_e are constants, and $R_{l,n}(r_s, r_e)$ is the set of secondary indices at level l spanned by the values of start and end elements in P_s and P_e . Fig. 6.4b shows a specific case in which start and end indices are represented by adjacent values in a single primary array as commonly seen in compact adjacency list representations of graphs [45] and compressed sparse

row (CSR) formats [4, 119]. Such patterns are seen in quadratic parallelization based graph traversal algorithms [81], sparse matrix multiplications [11] and shortest path graph algorithm (SSSP, APSP) implementations [45].

While not exhaustive, the relations presented above cover commonly seen access patterns [101]. In general, accesses can have non-linear relations, nested accesses with multiple primary arrays or with specific dependencies across nesting levels, more complex control flow, etc.

6.3 Basic TAFE Framework

In this section, we explain our framework for estimation of TAFs in detail. In heterogeneous systems involving CPUs and GPUs, the data is usually prepared (and/or initializes) by the host or earlier GPU kernels and then the main kernel(s) are launched across SMs. The required memory for data structures is usually dynamically allocated by the host. Our framework constructs the TAF relations for different data structures before their initialization and passes the information to the OS/runtime, which then use them to estimate TAFs and make optimization decisions before kernel launch. Thread ID-based patterns can be represented using few coefficients and can be statically evaluated to obtain the addresses accessed by threads. Data-dependent access patterns, by contrast require dynamic information (primary array values) to be estimated. Tracking primary arrays can incur overhead proportional to array size if not done carefully.

We use a simplified example derived from the *Bfs* benchmark in [28]

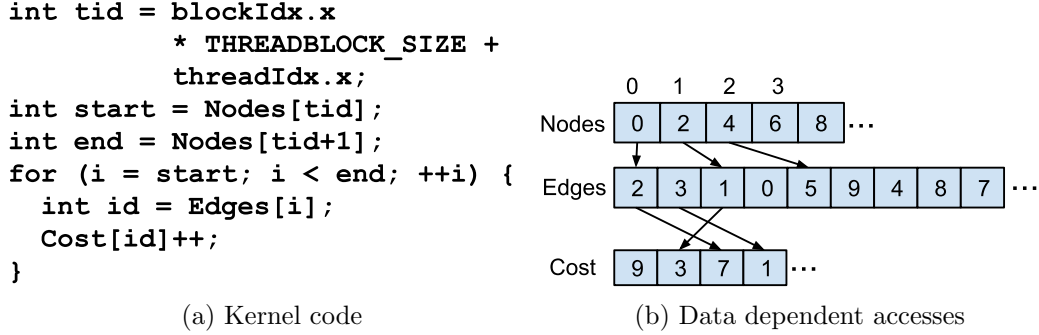


Figure 6.5: CSR example (*Bfs*).

to demonstrate the working of TAFE throughout this section. The example accesses data stored in CSR format, which is widely used to represent sparse matrices [4, 3] and graphs [5, 119]. A snippet of the example *Bfs* kernel code is shown in Fig. 6.5a. Array **Nodes** contains source offsets and is accessed in thread ID-dependent manner. **Edges** contains destination indices, which have data-dependent (indirect range) access patterns dependent on **Nodes**. Finally, the **Cost** array also exhibits data-dependent (indirect) access patterns, but in turn dependent on **Edges**. In the example in Fig. 6.5b, thread ID 0 accesses **Nodes**[0], **Edges**[0-1] and **Cost**[2-3]. In this example, TAFE can be used to estimate the **Nodes**, **Edges** and **Cost** array indices accessed by individual threadblocks. This information can then be used to co-locate threadblocks with the data regions they access. For example, **Nodes**[0] and **Edges**[0-1] are exclusively accessed by thread 0 and thus can be mapped into the same module as the thread. However, elements of **Cost** could be accessed by multiple threads and thus cannot be cleanly partitioned.

Table 6.1: TAFE API.

API	Description
taf_paramSetup (BlkDim, GridDim)	Passes the kernel launch parameters to the framework. The launch parameters passed are stored in the APD header. <i>BlkDim</i> , <i>GridDim</i> : block and grid dimensions of the application kernel.
taf_register (Base, size, TID_DEP, T, m , c, ofs, N, Pri)	This registers a thread ID-dependent TAF relation for an array with our framework. The arguments passed are stored within the APD entry. <i>Base</i> : array starting address, <i>Size</i> : array size, <i>TID_DEP</i> : thread ID-dependent access pattern, <i>T</i> : element size, m : pattern coefficients, <i>c</i> : constant, <i>ofs</i> : loop offset, <i>N</i> : loop iteration count, <i>Pri</i> : Primary array
taf_register (Base, size, INDIR, T, PBase, m, c, Pri)	Same as above but for Indirect access patterns. <i>Base</i> : array starting address, <i>Size</i> : array size, <i>INDIR</i> : Indirect access pattern, <i>PBase</i> : primary array base address, <i>m</i> : pattern coefficient, <i>c</i> : constant added to primary array values.
taf_register (Base, size, INDIR_RANGE, T, <i>PBase_s</i> , <i>c_s</i> , <i>PBase_e</i> , <i>c_e</i> , Pri)	Same as above but for Indirect range access patterns. <i>Base</i> : array starting address, <i>size</i> : array size, <i>INDIR_RANGE</i> : Indirect range access pattern, <i>PBase_s</i> : base address of primary array with starting indices, <i>c_s</i> : constant added to starting array values. <i>PBase_e</i> : base address of primary array with ending indices, <i>c_e</i> : constant added to ending array values.
taf_setRdy (Base)	Indicate that all the information required to evaluate the TAF for an array is available. For a data-dependent pattern, this call needs to be made after the last store to its primary array. This API stalls until all pending stores are completed before marking the array as ready. <i>Base</i> : array starting address.
taf_reset (Base)	Remove registered array and reset APD and extent table entries. <i>Base</i> : array starting address.

6.3.1 Static Code Annotations

TAFE provides APIs to collect static attributes about arrays from the application and store them in an *access pattern directory (APD)* within OS memory. API calls are annotated in the host code by the programmer. Note that the programmer does not need to have any knowledge about the architecture of the system on which the application will be executed. Annotated TAFs and TAF relations represent application-specific but architecture-independent

information. Therefore, the compiled code can be ported to different systems without further modifications.

API The complete list of TAFE APIs with their parameters is described in Table 6.1. The snippet of host code corresponding to the *Bfs* kernel launch with TAFE API calls is shown in Listing 6.1. The application allocates memory for arrays, initializes them and launches the kernel which operates on them. The annotated code required for TAFE is shown in bold. We first setup the kernel parameters for the framework using a `taf_paramSetup()` call. Each of the arrays are then registered with the framework along with the attributes of their access patterns. **Nodes** is registered as an array with its base address, size (`numNodes+1`), thread ID-dependent access (`TID_DEP`), element-size (`sizeof(int)`) and coefficient vector. The constant, loop offset and loop iteration count are set to 0. **Edges** and **Cost** are similarly registered, except that the base address of their primary array is also provided. **Edges** is registered with **Nodes** as both its starting and ending primary arrays to indicate a CSR like format. **Nodes** and **Edges** are set as primary arrays during registration. The `taf_setRdy()` call is used to mark the arrays ready. Unlike thread ID-dependent arrays, data-dependent arrays are marked ready only after its primary array is initialized, as can be seen for **Edge** and **Cost**. **Nodes** and **Edges** are assigned values within the loop, details of which have been omitted for simplicity. The values stored to primary arrays (**Nodes** and **Edges**) are tracked before kernel launch as explained later. Once the entire

primary array has been initialized and tracked, the TAF of the dependent arrays can be estimated.

```

taf_paramSetup(dimBlock, dimGrid);
int *Nodes = malloc(numNodes+1)
taf_register(Nodes, numNodes+1, TID_DEP, sizeof(int),
              [1,0,0,THREADBLOCK_SIZE,0,0],0,0,0,1);
taf_setRdy(Nodes);
int *Edges = malloc(numEdges);
taf_register(Edges, numEdges, INDIR_RANGE,
              sizeof(int), Nodes, 0, Nodes, 0, 1);
for (int i = 0; i < numNodes+1; i++) {
/* Initialize "Nodes" array */
    Nodes[i] = prepNodeData(i);
}
taf_setRdy(Edges);
int *Cost = malloc(numNodes);
taf_register(Cost, numNodes, INDIR, sizeof(int), Edges, 1, 0, 0);
for (int i = 0; i < numEdges; i++) {
/* Initialize "Edges" array */
    Edges[i] = prepEdgeData(i);
}
taf_setRdy(Cost);
for (int i = 0; i < numNodes; i++) {
    Cost[i] = prepCostData(i);
}
kernel<<<dimGrid,dimBlock>>>(Nodes, Edges, Cost);

```

Listing 6.1: CSR example: host code with TAFE APIs.

Access Pattern Directory Our framework maintains API-annotated information pertaining to individual access patterns and their TAF relations within APD entries. Fig. 6.6 shows the APD organization and fields within each entry are described in Table 6.2. The APD first stores kernel launch parameters (grid and threadblock size) to be used as inputs to TAF relations. It further consists of multiple entries, where each entry is used to capture a

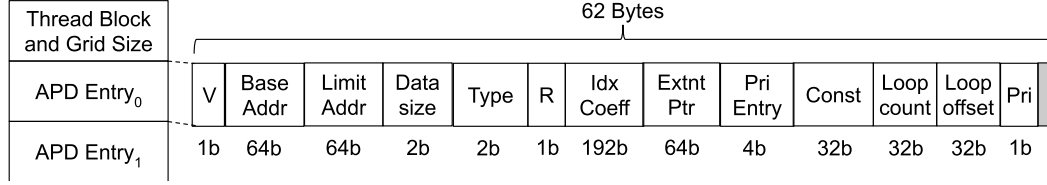


Figure 6.6: Access pattern directory (APD) and entries.

dimBlock, dimGrid	V	Base Addr	Limit Addr	Data Size	Type	R	Idx Coeff	Extnt Ptr	S-Pri Entry	E-Pri Entry	Const	Loop count	ofs	Pri
Nodes Entry	1	Nodes	Nodes + (numNodes+1)*4	2'b10	TID_DEP	1	[1, 0, 0, THREADBLOCK_SIZE, 0, 0]	Addr ₀	-	-	0	0	0	1
Edges Entry	1	Edges	Edges + numEdges*4	2'b10	INDIR_ RANGE	1	NA	Addr ₁	x0	x0	0	0	0	1
Cost Entry	1	Cost	Cost + numNodes*4	2'b10	INDIR	1	NA	-	x1	-	0	0	0	0

Figure 6.7: CSR example: populated APD table.

specific access pattern of a data structure. If a data structure is accessed with multiple patterns, it can be registered multiple times for different patterns. The APD is stored in memory (where the APD start address, for example, is contained in the Process Control Block), such that it can be accessed during API calls. Each APD entry fits into one cacheline allowing for efficient access.

The populated APD table after array registration for the *Bfs* example is shown in Fig. 6.7. In addition to the *dimBlock* and *dimGrid* dimensions stored in the header, the APD contains 3 entries, one for each array (**Nodes**, **Edges** and **Cost**), which are populated with the information provided during their corresponding (`taf_register()`) calls. In addition, all *R* bits are set (through `taf_setRdy()` calls), indicating that information required to evaluate their TAF is ready.

Table 6.2: APD Entries.

Field	Description
V	Valid bit.
Base Addr	Starting virtual address.
Limit Addr	Ending virtual address.
Data Size	Array element size (8b/16b/32b/64b).
Type	Access pattern. (TID_DEP/INDIR/INDIR_RANGE).
R	Bit indicating information is ready to estimate TAF.
Idx Coeff	Coefficients m .
Extnt Ptr	Pointer to Extent table.
S-Pri Entry	Index of APD entry of primary array with starting indices.
E-Pri Entry	Index of APD entry of primary array with ending indices.
Const	Constant <i>c</i> .
Loop Count	Iteration count <i>N</i> .
Loop Offset	Iteration offset <i>ofs</i> .
Pri	Bit indicating if the array is primary array.

6.3.2 Dynamic Data Dependency Tracking

Data-dependent access patterns use values stored in primary arrays to compute the indices for secondary arrays. This requires capturing the relationship between thread indices and primary array values. However, if not carefully done, the overhead involved in tracking primary array values can grow proportional to the array size, E.g. consider a data-dependent access $B[A[tid]]$. Here, the index of array B accessed by thread tid is $A[tid]$. To track the indices of B accessed by a given threadblock TB , all $A[tid]$, $tid \in TB$ need to be tracked. This overhead exacerbates in case of nested data-dependent accesses. Instead of tracking the relation between primary array values and their indices and later relating thread IDs to these indices, we can reformulate the problem as directly tracking the primary array values referenced by every threadblock or group of threadblocks. This problem is similar to the one faced

in file systems. A file system directory also faces the challenge of tracking disk block IDs pertaining to individual files while keeping the overhead small to reduce file system wastage. The XFS file system [109] specifically uses extents for managing space. An extent corresponds to one or more adjacent blocks on the disk. Instead of tracking each individual block using bitmaps or a linked allocation, a B+ tree of extents is maintained to indicate which blocks belong to a file or the free list.

Extent Table We propose an extent-based mechanism inspired by XFS to capture the values stored in primary arrays without enduring overhead proportional to the array size. Instead of a tree, we use an extent table (ET) per primary array, organized as list of extents stored in memory as shown in Fig 6.8a. Since GPUs perform scheduling and mapping at threadblock granularity, we track the primary array values for each threadblock or threadblock group. An extent captures the range of values in the primary array accessed by the threads in one threadblock or threadblock group. Extents contain a valid byte and the tuple of minimum and maximum values stored. To limit storage overhead, we track primary array values using a single extent per threadblock. Stored value is inserted into the extent corresponding to the threadblock which will access the index into which the value is being stored. There is an extent table for every primary array to track the values stored, allowing TAFE to support nested input-data dependent access patterns, since the extent table of immediate primary array is sufficient for estimating the address footprint.

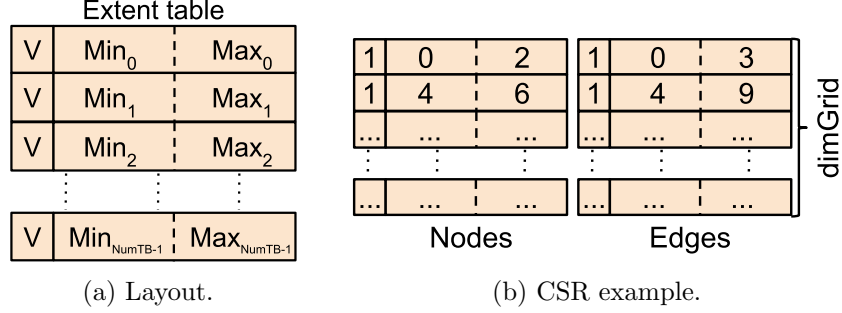


Figure 6.8: Extent table.

Primary Array Tracking The values stored in primary arrays have to be processed to populate the extent table appropriately. This can be done either by re-reading the primary arrays prior to kernel launch or by overloading their initialization and avoiding additional loads. The approach involving re-reading primary arrays requires 1) iterating across all the thread IDs in the GPU kernel, 2) computing the TAF relations to obtain the indices accessed by each thread, and 3) loading the elements from the indices and inserting them into their corresponding extent. In case of thread ID-dependent primary array, the range of array indices accessed by every threadblock (or threadblock group) is calculated using the attributes from the APD entry and kernel launch parameters. For data-dependent primary arrays, the mapping between threadblock IDs and range of indices accessed by them is present in the extent entries of their nested primary array. Re-reading of primary arrays can be performed as part of `taf_setRdy()` calls before the secondary array is even initialized. Re-reading of primary arrays can be done on the host CPU or by exploiting parallelism (using an additional kernel) on the GPU itself. The latter also

supports cases when primary arrays are initialized on the GPU.

Alternatively, primary array stores can be overloaded to track the stored values without the need for re-reading values from memory. On every overloaded store, this will require 1) computing the array element index into which the value is stored, 2) computing the inverse TAF relation to obtain the index to threadblock mapping, and 3) inserting the value into the corresponding threadblock extent. Array element indices can be computed using their base address and element size (from APD entries). Inverse TAF computation, by contrast, is more complex and can only be evaluated if the TAF relation is invertible, i.e. each primary array index is accessed by a single threadblock. If the TAF relation is non-invertible, we track only one of the threadblocks and other accesses will be false negatives when using overloading. Inverse TAF computation otherwise depends on the access pattern of the primary array. On every store, the element offset (primary array index) is compared against the range of array indices corresponding to each threadblock. For thread ID-dependent primary arrays, the range of array indices accessed can be computed from attributes in the APD entry. By contrast for data-dependent primary arrays, the required indices are present in the nested extent entries. On a match, the threadblock ID is selected. This approach works, but results in considerable overhead since the array indices for each threadblock or group need to be computed and compared against the element offset. This penalty scales with the number of threadblocks or groups.

In either case of re-reading or overloading, the stored values are inserted

by checking against the extent corresponding to the determined threadblock (or threadblock group) to see if its range already encompasses the value. If not, the extent is expanded accordingly. APD entries of primary arrays contain the starting address of the corresponding extent table (*Extnt Ptr*). ETs for different primary arrays are aligned at cacheline boundaries. The address of the cacheline containing extent for threadblock TB can thus be computed as $Extnt\ Ptr + ((TB / \#extents\ per\ cacheline) \times cacheline\ size)$. If the primary array values inserted are not continuous, it can potentially result in over-estimation. *INDIR_RANGE* patterns access contiguous indices, so a single extent is sufficient to perfectly capture them. However, the same is not true for *INDIR* patterns and may result in false positives. For the benchmarks evaluated, we find that 1 extent per threadblock is sufficient and results in only 1.5% average error in TAF estimation. Considering 1 extent per threadblock, a 64B cacheline can hold extents for 7 threadblocks. Each threadblock takes $1 + 2 \times 4 = 9$ bytes of space to store its extent and validity information. For a kernel with 64k threadblocks, each primary array would thus require ~0.57MB of memory. Alternatively, stores pertaining to threadblocks assigned to same SM or module can be tracked together reducing the extent table size.

Populated extent tables after initialization of **Nodes** and **Edges** arrays in the *Bfs* example are shown in Fig. 6.8b, assuming each threadblock consists of 2 threads accessing 2 elements of **Nodes** (i.e., *THREADBLOCK_SIZE* in Listing 6.1 is 2). Since threadblock 0 will access **Nodes**[0-1], its first extent in the **Nodes** table covers the value range (0, 2). Similarly, threadblock 1 accesses

`Nodes[2-3]` with extent $(4, 6)$. Values in `Nodes` in turn determine the starting indices of the indirect ranges that threadblocks access in `Edges`, where the end of a threadblock’s range is determined by the start of the next threadblock’s range. Threadblocks 0 and 1 will access `Edges[0-3]` and `Edges[4-7]` resulting in extents $(0, 3)$ and $(4, 9)$, respectively. Indices for `Cost` are determined by the minimum and maximum value of the corresponding threadblock’s extent, e.g. threadblock 0 and 1 will be assumed to access `Cost[0-3]` and `Cost[4-9]`. Since the values stored in `Edges[4-7]` are not continuous (see Fig. 6.5), it results in false positives (`Cost[6]` and `Cost[7]`).

6.3.3 Basic TAFE Limitations

The framework described thus far, directly supports only predefined access patterns to facilitate programmer-assisted API annotation and hardware-assisted tracking. However, applications often have unique access patterns that will lead to estimation inaccuracies when capturing them using predefined patterns. In the following, we discuss some of such application characteristics. While modifying source code is always a solution, we present possible approximations.

Section 6.2 describes access patterns directly supported by our TAFE framework. Other patterns can be approximated using a combination of supported patterns to achieve the tightest possible fit. Patterns can be over- or under-approximated depending on the use-case requirements, e.g. to capture non-linear relations in linear form. Consider an access pattern `A[tid*4 + j`

```

if( threadIdx.x >= i+1 && threadIdx.x <= BLOCKSIZE-i-2)) {
    idx = blockIdx.x*sbcsls + threadIdx.x + cols*(start+i);
    result[threadIdx.x] = shortest + gpuWall[idx - b]; }

```

Listing 6.2: Control divergence example (Pathfinder).

+ $i*N$], $0 \leq j < 4$, $0 \leq i < N$. The tightest fit can be achieved by under-approximating this relation as $\{tid \cdot 4 + i \cdot N \mid 0 \leq i < N\}$. Patterns like $A[tid \% N]$, where $N_t > N$ is the total threads in the kernel, can be represented as thread ID dependent relation of the form $\{tid + i \cdot (-N) \mid 0 \leq i < (N_t/N)\}$. Evaluating this relation will result in underflow of indices/addresses, which are filtered using *Base* and *Limit* APD entries.

Loop structures are only supported in thread-ID dependent accesses at level 0. Branching behavior with control divergence can be over-approximated by capturing relations for all possible branches. Listing 6.2 shows an example from the *Pathfinder* benchmark, where *result* and *gpuWall* are only accessed by some threads. We over-estimate this access by registering the TID_DEP patterns in the branch unconditionally for all threads. In the worst case, unsupported patterns require marking the entire array as part of the footprint.

Applications with multiple kernels are supported by tracking the access patterns of the entire application as a whole in a single APD. This can accurately capture the individual kernel patterns if they have the same threadblock and grid dimensions. Applications consisting of kernels with varying dimensions (e.g. *Lud*) can be approximated by either identifying a threadblock and grid dimension that allows non-contradictory mappings across kernels or us-

ing the dominant kernel dimensions. Alternatively, TAFE can be extended to support kernels with different dimensions by tracking each kernel separately (requires an APD per kernel).

The described TAFE implementation also has some limitations beyond accuracy effects. Obtaining TAFs requires primary arrays to be initialized before and not modified during kernel execution.

6.4 Hardware-Assisted Dependency Tracking

As described previously, primary array values can be tracked purely in software in the host CPU or GPU. This, however, requires software modifications and incurs overhead. We propose an optional lightweight hardware extension to support transparent tracking of primary array stores and reduce software overhead. Hardware assistance is only made possible by restricting support to predefined access patterns. Arrays are transparently tracked while they are being initialized avoiding additional loads.

6.4.1 Hardware Architecture

Optional TAFE hardware extensions are inserted between the cache and core as shown in Fig. 6.9. They can be potentially combined with prefetchers by sharing logic and functional units [30, 123]. TAFE hardware extension consist of components for comparison, inverse TAF computation and ET insertion.

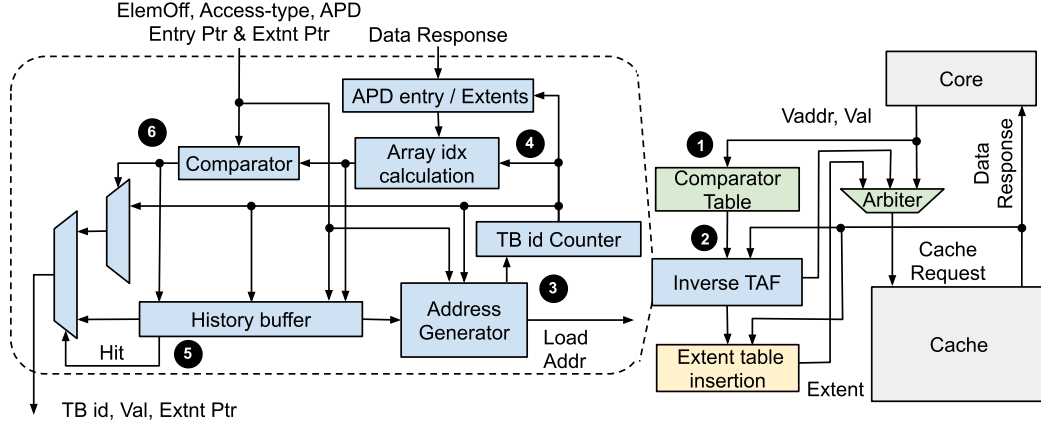


Figure 6.9: TAFE hardware for tracking primary arrays.

Comparator Table The comparator table is responsible for detecting stores to primary arrays and computing the element index. As part of registration (`taf_register()`) calls for primary arrays, the framework populates the primary array’s base and limit address, element size, access type and a pointer (APD entry pointer for TID_DEP or extent pointer otherwise) into a hardware comparator table. If the primary array itself has a nested data-dependent access pattern, the *Extnt Ptr* of the nested, second-level primary array is also stored. APD entries of primary arrays can be pinned into the cache until kernel launch as part of array registrations for faster access. A single table entry can be reused for tracking multiple primary arrays as long as their initialization is not interleaved, i.e. sequential. An entry is reserved during the registration call of primary array and freed again as a part of the secondary array’s `taf_setRdy()` call. If initializations are not sequential, the size of the comparator table determines the maximum number of primary arrays that

can be tracked simultaneously in the system. Our results show that tracking 2 primary arrays is sufficient for the benchmarks evaluated.

Every time a store reaches L1 cache, the address is then compared against the address ranges in the comparator table ❶. An address within a tracked range indicates a store to a primary array, and the element offset is computed using the base address and element size. The element offset, value to be stored, access type, pointer (APD entry/extent pointer) and optional nested extent pointer are then passed to an inverse TAF block ❷.

Inverse TAF Computation This block computes the inverse TAF to determine the correct ET entry to be updated. As described previously, this requires computing the index ranges for each threadblock and matching them against the primary array index. In order to reduce the overhead, we exploit the fact that primary array initialization is mostly performed sequentially (within a loop). This means that the threadblock accessing the index on which a store is performed will mostly be same as that of the previous store. Further, we assume that the subsequent primary array indices are accessed by threadblocks in incremental order, i.e. the indices accessed by a threadblock are larger than those accessed by preceding threadblocks (e.g. in TID_DEP, INDIR_RANGE primary access patterns). This avoids the need of iterating across all threadblocks. We use a single-entry history buffer capable of storing one range of array indices (extent) and its corresponding threadblock ID. On the first store to a primary array, the range of array indices are computed for

the first threadblock (or threadblock group). In case of a thread ID-dependent primary array, its APD entry is first loaded ③ and the attributes from the APD entry are used to calculate the range indices ④. For a data-dependent array, the extent of its nested primary array is loaded (using the extent pointer from comparator metadata) to obtain the indices. The range of indices and the threadblock ID are inserted into the history buffer. On subsequent stores, the primary index is compared against the entry in the history buffer ⑤. If it is a hit, the matching overhead is avoided. On a miss, the threadblock ID is incremented and a new range of indices is computed and compared against the element offset ⑥ before being updated in history buffer. In our experiments, we observe a >99% history buffer hit rate thereby significantly reducing the overhead.

Extent Table Insertion Finally, the threadblock ID, primary array extent pointer along with value being stored are passed to an extent table insertion block, where the value is inserted into the appropriate entry within the extent table of the primary array. A write-combining buffer is used to merge writes to the same extent reducing the cache transactions.

6.4.2 Limitations

In order to facilitate hardware dependency tracking, TAFE is restricted to directly support predefined access patterns as presented in Section 6.2. Supporting arbitrary access patterns would require proportionally complex

hardware which may not be feasible. Furthermore, our proposed optional hardware tracking currently only supports nested primary array accesses with a single TID_DEP pattern or INDIR_RANGE pattern using the same starting and ending arrays and hence cannot support arbitrary nested accesses.

6.5 TAFE Generalization and Automation

The basic TAFE framework as described earlier is inherently restricted to directly support only certain access patterns and requires others to be approximated as a combination of supported patterns as described in Section 6.3.3. Moreover, TAFE relies on programmer assistance for annotating API calls to collect static attributes about access patterns. Although the attributes are easily-derivable and straightforward to annotate, understanding the different access patterns within the application and annotating corresponding APIs could become an arduous task for users especially for complex applications. To address these concerns, we further extend and generalize TAFE to support arbitrary access patterns while also developing compiler support to expand its applicability.

6.5.1 TAFE Generalization

In order to support arbitrary access patterns, the TAF relation for every array access is captured and provided to TAFE as a pointer to a standalone *TAF function* in the code. A TAF function takes a thread ID, kernel launch dimensions, and kernel arguments as inputs and computes the indices

Table 6.3: Extended TAFE API.

API	Description
tafe_paramSetup (kName, BlkDim, GridDim)	Passes the kernel launch parameters to the framework. <i>kName</i> : Name of kernel for which the launch parameters are provided. <i>BlkDim</i> , <i>GridDim</i> : block and grid dimensions.
tafe_register (kName, arrName, patternID, tfptr)	Register an access pattern for array <i>arrName</i> in kernel <i>kName</i> . <i>patternID</i> : Unique ID for pattern, <i>tfptr</i> : pointer to TAF function representing the pattern.
tafe_priVal (kName, arrName, patternID, grpId, minPtr, maxPtr)	Access the extent of primary array values for pattern <i>patternID</i> of primary array <i>arrName</i> in kernel/function <i>kName</i> corresponding to threadblock group <i>grpId</i> . <i>minPtr</i> , <i>maxPtr</i> : pointers to hold the value extent limits on function return.
tafe_evaluate (kName, arrName, grpId, strtBlk, endBlk, <ArgList>)	Computes and returns the TAF of array <i>arrName</i> in kernel <i>kName</i> for a range of threadblocks as a pair. <i>grpId</i> : Unique ID for threadblock group, <i>strtBlk</i> , <i>endBlk</i> : Start and end threadblock ID of the range, <i>ArgList</i> : arguments of <i>kName</i> .
tafe_trackPriArray (kName, arrName, patID, pBase, size)	Tracks and populates extents of primary array <i>arrName</i> of kernel <i>kName</i> in accordance with pattern <i>patID</i> by re-reading the array. <i>pBase</i> : Base address of primary array, <i>size</i> : size of primary array.
tafe_trackPriArray (kName, arrName, patID, idx, stVal)	Tracks and populates the extent of primary array for a single store. <i>idx</i> : primary array index to which values is stored, <i>stVal</i> : value being stored.

accessed by the thread. The body of a TAF function can take any form and supports arbitrary code. Unlike basic TAFE, which used static attributes (e.g. coefficients, constant, loop counts) to capture an access pattern, the extended TAFE uses TAF function to represent an access thus supporting generic patterns. This generality however, is achieved at the cost of increased programming overhead as TAF functions have to be created for every access pattern tracked, based on the kernel code.

Table 6.3 presents APIs for the extended TAFE. The API calls mostly resemble that of basic TAFE (e.g. paramSetup, register) and perform similar operations. The register call is different in that it takes in a function pointer

instead of attributes. Unlike basic TAFE, hardware-assisted primary array tracking is not feasible for extended TAFE, so, new API calls have been added to expose software-based tracking of primary arrays to the users. We also provide APIs for users to evaluate and query TAFs. In addition to generalized TAF functions, we include support for tracking individual kernels/device (GPU) functions within the application thus allowing different launch parameters (threadblock and grid dimensions) for different kernels.

Listing 6.3 shows the CSR example with the new APIs. We still need to setup the kernel launch parameters using the `tafe_paramSetup()` call, but this is now done for every kernel (or device function) tracked. The listing shows parameters being setup for kernel *kernel*. The most significant change is in array registration. The arrays to be registered are identified using a combination of their name and kernel/function name instead of base address. This allows kernel-specific access pattern tracking. `tafe_register()` calls don't require static attributes. Instead the TAF function representing the pattern is passed. We support multiple access patterns for each array per kernel by identifying them using a unique pattern ID. The example shows pattern 0 being registered for arrays *NodesDev* and *EdgesDev* represented by *tafFuncNodes* and *tafFuncEdges*, respectively. The TAF functions have to be created based on application code. *tafFuncNodes* represents the thread ID-dependent access pattern of *NodesDev* where the indices accessed by a thread are computed and returned as an extent. Similarly, *tafFuncEdges* represents the data-dependent access pattern of *EdgesDev*. This function uses the values stored in

```

// TAF functions for Nodes and Edges arrays
void tafFuncNodes( unsigned sblkidx, unsigned sblkidy, unsigned sblkidz,
    unsigned eblkidx, unsigned eblkidy, unsigned eblkidz,
    unsigned blkDimx, unsigned blkDimy, unsigned blkDimz,
    unsigned grpId, int* Nodes, int* Edges, int* Cost,
    unsigned* min, unsigned* max) {

    *min = sblkidx * blkDimx
    *max = (blkDimx-1) + eblkidx * blkDimx; }

void tafFuncEdges( unsigned sblkidx, unsigned sblkidy, unsigned sblkidz,
    unsigned eblkidx, unsigned eblkidy, unsigned eblkidz,
    unsigned blkDimx, unsigned blkDimy, unsigned blkDimz,
    unsigned grpId, int* Nodes, int* Edges, int* Cost,
    unsigned* min, unsigned* max) {

    tafe_priVal(kernel, NodesDev, 0, grpId, min, max); }

void main() {
    tafe_paramSetup(kernel, dimBlock, dimGrid);

    int *Nodes = malloc(numNodes+1);
    int *Edges = malloc(numEdges);
    tafe_register(kernel, NodesDev, 0, tafFuncNodes);
    tafe_register(kernel, EdgesDev, 0, tafFuncEdges);

    std::pair<unsigned,unsigned> NodeTaf, EdgeTaf;
    NodeTaf = tafe_evaluate(kernel, NodesDev, /*grpId*/ 0,
        /*threadblock range*/ strtBlkid, endBlkid,
        /*kernel arguments*/ Nodes, Edges, Cost);

    for (int i = 0; i < numNodes+1; i++) {
        Nodes[i] = prepNodeData(i);
    }
    tafe_trackPriArray(kernel, NodesDev, 0, &Nodes[0], numNodes+1);

    EdgeTaf = tafe_evaluate(kernel, EdgesDev, 0, strtBlkid, endBlkid,
        Nodes, Edges, Cost);
    ...
    kernel<<<dimGrid,dimBlock>>>(Nodes, Edges, Cost);
}

__global__ void kernel(int* NodesDev, int* EdgesDev, int* CostDev){
... // kernel body

```

Listing 6.3: CSR example: host code with Extended TAFE APIs.

primary array (*NodesDev*) to compute its TAF using a `tafe_priVal()` call. Each `tafe_register` call allocates an entry within the APD storing the array name, kernel name, pattern ID and TAF function pointer within them. APD entries now have 2 extent tables, one for array values and the other to store the computed TAFs.

We also provide means for users to compute and access the TAF which can be used for user-specified optimizations [102]. `tafe_evaluate()` calls evaluate the TAF function passed as part of a prior registration call to compute the TAF of the specific array access pattern for a range of threadblocks. This user-specified range of threadblocks identified using the group ID determines the granularity at which TAFs are computed and stored (minimum 1 threadblock). TAF functions can be evaluated for every thread, but that may result in overhead proportional to the kernel itself. We approximate TAF computation within `tafe_evaluate()` to limit overhead. For thread ID-dependent access patterns, TAF functions are evaluated for threads at threadblock boundaries and interpolated to obtain the TAF for the user-specified threadblock range. This can cause errors in TAF estimations if different threadblock groups access interleaved addresses. For data-dependent access patterns, TAFs are evaluated for the extents tracked per threadblock group and linearly interpolated. This usually does not increase the TAF estimation errors as in most cases, secondary array indices are linearly derived (e.g. via the identity function) from the primary array values, e.g. in case of gather operations. As TAF functions are evaluated only at boundaries, their computation is performed on the host

CPU where other data structures such as APDs and extents are stored. Alternatively, if `tafe_evaluate()` computation requires higher parallelism, it can be executed on GPUs.

In the example shown, `tafe_evaluate()` calls `compute` the TAF for *NodesDev* and *EdgesDev* for the *strtBlkid* to *endBlkid* range of threadblocks. The computed TAF is returned and stored as an extent specific to the group ID within the APD entry for future queries. For data-dependent accesses, the threadblock groups and their group IDs should match that of their primary arrays. The kernel arguments are also passed to the evaluate call to satisfy access patterns that require them as inputs.

Finally, the API provides ways to track primary arrays in software either by re-reading the array or overloading the stores. Tracking can be performed on either CPUs or GPUs, and we evaluate the options as part of Section 6.6.3. The listing shows `tafe_trackPriArray()` call to track the *NodesDev* primary array using re-reading. Primary arrays must be evaluated and tracked before computing the TAF for their corresponding secondary array.

6.5.2 Compiler support for TAFE

Thus far, we described the extensions made to TAFE to support arbitrary access patterns, but in order to do so, the user is expected to provide the TAF function for every pattern tracked. In doing so, the user has to understand the underlying access patterns and develop appropriate functions

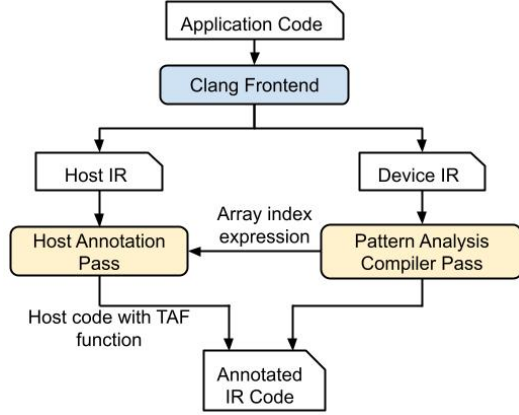


Figure 6.10: TAFE compiler support.

replicating them, which can be difficult. To address this, we have developed compiler support to aid the users in determining the access patterns and creating the TAF functions. The users can simply identify the kernel and array to be tracked using a pragma, and the compiler will transparently create the TAF functions for the specified array. This is achieved by introducing 2 additional compiler (LLVM [121]) passes as shown in Fig. 6.10. A *Pattern analysis pass* examines the device (GPU) code to identify and extract the access pattern information for the specified array. It examines the intermediate representation (IR) of the kernel body, identifies accesses to the array specified by the user and extracts the expression (sequence of instructions) that computes the array index. The sequence of instructions is derived by traversing the reverse dependency chain, starting from the array index variable and terminating at in-built variables (e.g. `threadIdx.x`, `blockIdx.x`) or kernel arguments.

```

void main() {
    tafe_paramSetup(kernel, dimBlock, dimGrid);

    int *Nodes = malloc(numNodes+1);
    int *Edges = malloc(numEdges);

    std::pair<unsigned, unsigned> NodeTaf, EdgeTaf;
    NodeTaf = tafe_evaluate(kernel, NodesDev, 0, strtBlkid, endBlkid,
        Nodes, Edges, Cost);

    for (int i = 0; i < numNodes+1; i++) {
        Nodes[i] = prepNodeData(i);
    }
    tafe_trackPriArray(kernel, NodesDev, 0, &Nodes[0], numNodes+1);

    EdgeTaf = tafe_evaluate(kernel, EdgesDev, 0, strtBlkid, endBlkid,
        Nodes, Edges, Cost);
    ...
    kernel<<<dimGrid,dimBlock>>>(Nodes, Edges, Cost);
}

TAFEREGISTER(kernel, NodesDev)
TAFEREGISTER(kernel, EdgesDev)
__global__ void kernel(int* NodesDev, int* EdgesDev, int* CostDev) {
    ... // kernel body
}

```

Listing 6.4: CSR example: compiler support for extended TAFE.

The *Host annotation pass* uses the information from the *pattern analysis pass* to transparently build the TAF function, which takes in kernel arguments, launch dimensions, and thread ID as inputs and computes their corresponding TAF. This function can then be called by users as part of the evaluate calls, thus relieving them of the burden of analyzing access patterns within the application code and generating the TAF functions.

Listing 6.4 shows the CSR example with compiler support for the extended TAFE APIs. No user provided TAF functions are required. Instead,

pragma statements (`TAFE_REGISTER`) indicating to the compiler to track arrays *Nodes* and *Edges* are added by the user. The compiler tracks the access patterns for the arrays and transparently generates their TAF functions. The function that would be generated by the compiler for *Nodes* is shown in Listing 6.5 as an example. This function computes the TAF for 1 thread, and will be executed multiple times to compute the TAF for a range of thread-blocks. Furthermore, the `tafe_register()` calls are no longer required as the compiler takes care of annotating these calls. The compiler can potentially also annotate calls to track primary arrays, but this can be challenging due to pointer aliasing and uncertainty in determining appropriate annotation locations. As such, we expect the user to annotate primary array tracking calls in our current implementation.

```
define void @tafFuncNodes_0(i32 %tidx, i32 %tidy, float %tidz,
    i32 %blkidx, i32 %blkidy, float %blkidz,
    float %blkDimx, float %blkDimy, i32 %blkDimz, i32 %grpID,
    i32* %Nodes, i32* %Edges, i32* %Cost, i32* %min, i32* %max) {
base:
    %0 = mul i32 %blkidx, %blkDimx
    %1 = add i32 %0, %tidx
    store i32 %1, i32* %min
    store i32 %1, i32* %max
    ret void
}
```

Listing 6.5: Example of compiler-generated TAF function for Nodes.

6.5.3 Extended TAFE Limitations

Extended TAFE with compiler support addresses many of the shortcomings experienced in basic TAFE. However, it has some limitations of its own.

TAF function sampling Currently, TAF functions are evaluated and stored per user-specified group of threadblocks (minimum of 1 threadblock). For thread ID-dependent accesses, a TAF function is evaluated only for threads at threadblock boundaries. For data-dependent patterns, TAF functions are evaluated only for the extent boundaries tracked. TAFs are then linearly interpolated in between. Depending on the access pattern, this may result in estimation errors.

Hardware support Arbitrary access patterns cannot be handled by simple fixed-function hardware extensions. As such, the extended TAFE sacrifices hardware-assisted primary array tracking to allow capturing arbitrary access patterns. Primary arrays are only tracked in software by re-reading or overloading the stores.

TAFE overhead Since there is no hardware support for primary array tracking and TAF evaluation, TAF estimation may result in overhead comparable to re-running the kernel, especially for applications dominated by data-dependent accesses. If an application accesses the entire primary array once

and these accesses dominate the kernel execution time, such applications can incur TAFE overhead comparable to re-running kernel itself rendering TAFE not beneficial.

Access pattern coverage The applicability of the extended TAFE is only as generic as the TAF function supplied or the underlying compiler pass for pattern analysis. If an access pattern requires an input other than in-built variables or kernel arguments to compute the array index, such patterns are not supported. The current compiler implementation does not capture control divergence due to branches and such patterns would be approximated, similar to the basic TAFE (see Section 6.3.3). Further, the compiler pass can currently not handle external function calls as part of the sequence of instructions that compute the array index. Any function call present can be in-lined to circumvent this restriction.

Restricted TAFE benefits TAFE cannot enable every optimization. It currently estimates only spatial and not temporal locality. As such, it cannot enable corresponding temporal optimizations like cacheline pinning. For the optimizations enabled, TAFE may not always result in performance improvements. Benefits achieved depend on both the application behavior and the optimizations performed. For example, if most of the data is shared across threads, then static data partitioning will only achieve limited benefits and may not amortize the cost for estimating TAFs. However, read-only shared

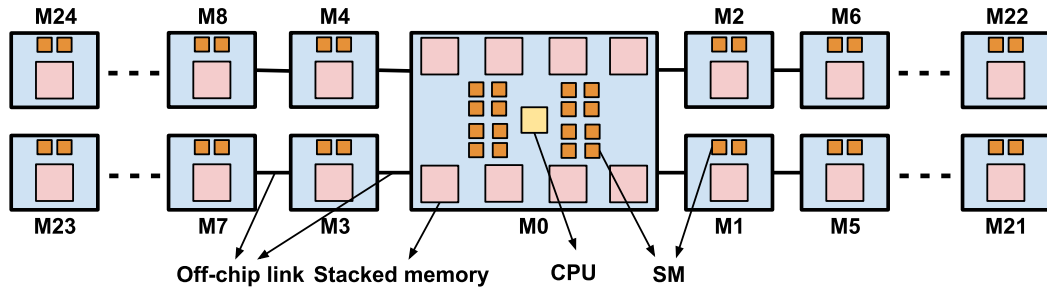


Figure 6.11: Simulated exascale node configuration.

data can still be replicated to achieve benefits. Furthermore, applications that are dominated by kernel launch latencies will not benefit from TAFE, as it will only add to the overhead.

6.6 Evaluation

We evaluate the benefits of TAFE by using the estimated TAFs for co-locating threadblocks/data in multi-GPU systems. We first evaluate basic TAFE using a simulator-based setup where estimated TAFs are used for both threadblock and data mapping across different NUMA GPU systems. We also evaluate its estimation accuracy and the impact of hardware-assisted dependency tracking. We then evaluate extended TAFE with compiler support on an actual multi-GPU system, where the estimated TAFs are used to provide hints to improve memory placement [102].

6.6.1 Simulator Setup

Our simulation setup models 2 different NUMA GPU system configurations: 1) A multi-GPU/-chiplet system with identical modules interconnected by off-chip links and 2) a heterogeneous multi-chiplet module (MCM) configuration representing recently proposed [80, 114, 95] exascale node architectures (Fig. 6.11). The configurations consist of a host CPU and GPU with unified memory distributed across interconnected modules. Details of our simulated configurations are provided in Table 6.4. We use `gem5-gpu` [91] (`VI_hammer_fusion`) for simulating our system. We augmented the CUDA runtime to include APIs required by TAFE. Support for the new APIs and for TAFE hardware modifications are made in the simulator. We model an APD with 16 entries, extent tables with 1 extent per threadblock, and a comparator table with 2 entries. The memory allocated by the application is uniformly distributed across HMC stacks.

We evaluated TAFE for applications mostly from the *Rodinia-nocopy* [28] benchmark suite. The list of applications including lines of code (LOC) added for TAFE annotations and storage overhead of TAFE metadata (APD and extent tables) are shown in Table 6.5. The applications are written/compiled in CUDA v3.2. Radix sort [46] is executed the prefix sum in the host machine instead of the GPU. The applications are annotated with the TAFE API calls to obtain the required information while maintaining the original functionality. Except for over-approximations to capture control divergence in *Hotspot*, *Lud* and *Pathfinder* (with 1, 4 and 1 instances, respectively), all patterns encoun-

Table 6.4: Simulation parameters.

Configuration 1: Multi-GPU/-chiplet	
NUMA modules	8 (all identical)
SM count	64 (8 per module)
Memory stacks	8 (1 per module)
Topology	Tree
Configuration 2: Exascale node	
NUMA modules	25
SM count	64 (16 in M0, 2 each in others)
Memory stacks	32 (8 in M0, 1 each in others)
Topology	Ext: Tree, Central pkg: Fully connected
Memory	
Bandwidth	Local: 160 GB/s, off-chip: 80 GB/s [48]
Memory model	HMC_2500 [16, 120]
Core Configuration	
CPU	OOO model, 2Ghz
SM	1.4Ghz, 48kB shared memory
Cache	L1:16kB, 4-way (pvt.), L2:1MB, 16-way (shared)

tered in benchmarks are directly supported by TAFE. Results are obtained by running each application (in addition to all host CPU instructions prior to first kernel launch) for 1B GPU instructions or to completion, whichever occurs first.

Threadblock and Data Mapping Optimizations Prior work has shown that mapping contiguous threadblocks within SMs benefits locality [84]. However, for multi-dimensional kernel grids, to capture maximum locality, threadblocks accessing the closest array elements should be mapped together. For thread ID-dependent arrays, we map threadblocks in row-, column- or depth-major order depending on the relative step size defined by block ID coefficients along x, y and z dimensions. Threadblocks are equally divided across SMs for

Table 6.5: Benchmarks.

Benchmark	Input	Access Pattern	TAFE Storage	TAFE LOC
Backprop (BP) [28]	256k	TID_DEP	256 B	25
Lud [28]	2048	TID_DEP	192 B	26
NN [28]	640k	TID_DEP	128 B	13
Pathfinder (PF) [28]	Default [91]	TID_DEP	192 B	19
Srad [28]	Default [91]	TID_DEP	768 B	32
Hotspot (HS) [28]	4096	TID_DEP	192 B	13
Bfs-1M [28]	1M nodes	TID_DEP, INDIR & INDIR_RANGE	1136 B	40
Bfs-4M [28]	4M nodes	TID_DEP, INDIR & INDIR_RANGE	2.8 kB	40
Cfd [28]	0.2M	TID_DEP & INDIR	942 B	52
Spmv-in [11]	inline [1]	TID_DEP, INDIR & INDIR_RANGE	4.7 kB	42
Spmv-ser [11]	serena [1]	TID_DEP, INDIR & INDIR_RANGE	12.4 kB	42
Radix Sort (RD) [46]	256k	TID_DEP & INDIR	1098 B	27

load-balancing. Since most applications have similar trends across different arrays, the decision is made based on the array registered first. Data is later mapped/allocated to complement the determined threadblock mapping. If data is already mapped before threadblock mapping is decided (e.g. threadblock remapping across multiple kernel calls), threadblocks is mapped to the module containing pages accessed by them. The module to which a page is mapped can be identified from its physical address, requiring a address translation per page or OS support as done in [110]. Using the page mapping and TAFs, appropriate threadblocks can be co-located with their data-pages. For kernels with data-dependent arrays, a specific order (e.g. x-dimension first) is selected, allowing the runtime to determine the exact threadblocks mapped to

each module. The primary array values now can be tracked per module instead of every threadblock, reducing the extent table size overhead (proportional to the number of modules).

Once the threadblock mapping has been determined (coefficient-based or pre-selected), the runtime uses estimated TAFs to allocate memory to modules while reducing off-chip data movement. This requires computing TAFs and performing page allocation before arrays are initialized on the host CPU as doing otherwise would require remapping of pages, which defeats the purpose of optimizing the mapping before kernel launch. TAFE supports TAF estimation and optimal page mapping prior to host initialization as all the required information is available during `taf_setRdy()` calls. We use a straightforward allocation algorithm: If a page is accessed by only one module, it is allocated in that specific module. If a page is accessed by multiple modules, it is mapped to the module with the shortest distance to all other accessing modules. We currently do not replicate read-only data shared and accessed by multiple modules. All page mappings are subject to memory capacity restrictions.

6.6.2 Address Footprint Accuracy

The primary purpose of TAFE is to estimate address footprints accurately prior to kernel launch. We begin by studying the accuracy with which TAFE is able to estimate the address footprints for different benchmarks. Evaluated benchmarks are not affected by limitations of hardware-assisted tracking. As such, software- and hardware-based tracking have the same ac-

curacy. Fig. 6.12 shows TAFE accuracy in form of a color map for Exascale node configuration. *NN* has a very similar color map as *Backprop*, *Bfs-4M* and *Spmv-ser* have similar characteristics as *Bfs-1M* and *Spmv-in* respectively and are omitted for space reasons. For every combination of page and module, we show when TAFE correctly estimated whether a page is actually accessed by a module (*true-pos*, shown in black) or not (*true-neg*, shown in white). We also show pages that were accessed but that TAFE captured otherwise (*false-neg*, shown in red), and pages not accessed but tracked by TAFE as accessed (*false-pos*, shown in yellow). The first 2 categories indicate correct tracking by TAFE, while the latter 2 indicate wrong tracking. We define accuracy as the proportion of page-module pairs that were estimated correctly by TAFE out of all possible combinations.

It can be seen that for most benchmarks, TAFE is able to track the address footprint accurately, including those with data-dependent accesses. The quantitative distribution of TAFE’s estimation accuracy by access patterns for configuration 1 is further summarized in Fig. 6.13. TAFE has on average 91% accurate estimations across all benchmarks. In data-dependent benchmarks, TAFE can estimate TAFs with on average 1.5% (and up to 8%) error. Most errors are due to conservative over-estimation under control divergence (*Pathfinder*, *Hotspot*) or limits of extent-based data tracking (*Cfd*, *Spmv-in*). *Lud* has a significant incorrect estimations because it has multiple kernel calls with varying launch parameters and control divergence over memory accesses, which does not accurately map to the patterns currently supported by TAFE.

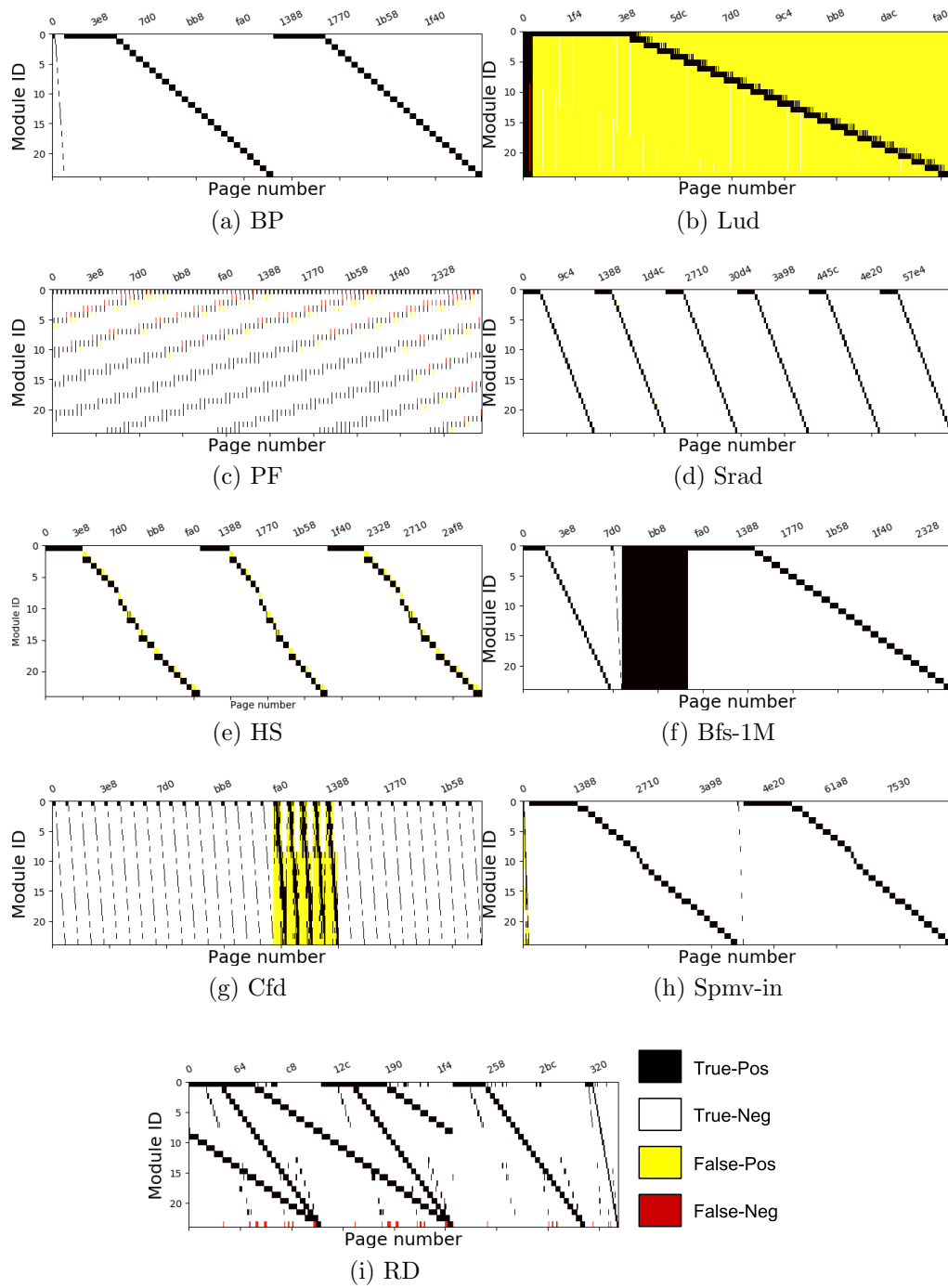


Figure 6.12: TAF tracking accuracy.

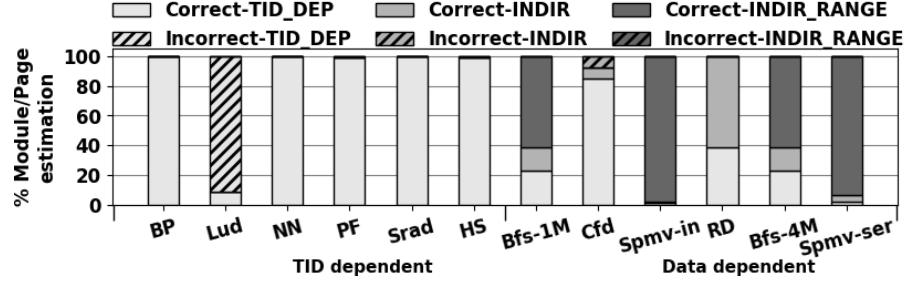


Figure 6.13: TAF estimation accuracy.

We measured the average value sparsity across indirectly (INDIR) accessed array indices for irregular graph and sparse matrix benchmarks. The value sparsity $1 - \frac{\text{Num. indices accessed}}{(\text{Max. index value} - \text{Min. index value})}$ for *Bfs-1M*, *Bfs-4M*, *Spmv-in* and *Spmv-ser* is 0.98, 0.99, 0.95 and 0.99, respectively. Note that TAFE accuracy ultimately depends on the granularity at which TAFs are requested. TAFE achieves high accuracy at page/module granularity for sparse applications since the probability that at least one word from each page within the range will be accessed is high, resulting in a near-contiguous range.

6.6.3 TAFE Hardware Overhead

In this section, we discuss the overhead of implementing TAFE. Estimating TAFs for thread ID-dependent access patterns can be done using static annotations and incurs overhead of API calls and memory space to store the APD table. Data-dependent access patterns can be tracked either in software or using optional hardware as described in Section 6.3. For software, we use re-reading of immediate primary arrays to process the values stored. This approach is more flexible and avoids limitations of inverse TAF compu-

Table 6.6: TAFE hardware costs.

Component	Storage	Logic	Delay
Comparator Table	68 B	4 CMP, 1 ADD, 1 SHFT	1 cyc. (2 pipeline stages)
Inverse TAF	133 B	2 MUL, 2 ADD, 1 counter, 1 CMP	Hit: 1 cyc., Miss: 8 cyc. + load
ET Insertion	34 B	2 CMP, 1 ADD	1 cyc.
Pipeline	76 B	control/stall logic	-

tations. We evaluate overhead of software tracking in the host CPU or GPU and hardware-assisted tracking in the CPU.

Table 6.6 summarizes the cost for TAFE hardware support. We use McPAT [74] to estimate the area overhead to be 2% of an Alpha21364 reference host core. Existing prefetcher components can be shared with TAFE tracking logic to further reduce the hardware overhead. CPU prefetchers [30, 123] already include hardware for tracking cache accesses, functional units for computing prefetch addresses and issue loads. Note that TAFE hardware overhead is independent of the number of SMs and data size, and the information obtained can be reused for different optimizations. The 3 stages for tracking stores (comparator table, inverse TAF and ET insertion) can happen in parallel to cache accesses and can be pipelined to support best-case throughput of 1 store per cycle on history buffer hits. Only one extra arbiter delay is added on the critical path of cache accesses. In rare cases of history buffer misses, the tracking pipeline and cache accesses need to be stalled until the miss is served. We assume an inverse TAF miss delay of 20 cycles for our evaluations.

Fig. 6.14 compares the TAFE overhead when implemented in software

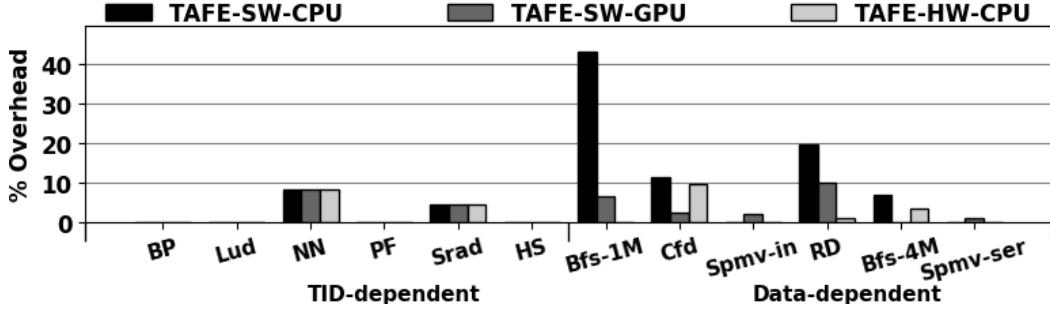


Figure 6.14: TAFE runtime overhead.

within the host CPU (TAFE-SW-CPU) or GPU (TAFE-SW-GPU) and when using optional hardware support in the host CPU (TAFE-HW-CPU) for exascale node configuration. Overhead is compared against host CPU runtime without TAFE. For thread ID-dependent benchmarks that do not require primary array tracking, both hardware and software versions of TAFE incur the same small overhead for API calls. For data-dependent benchmarks, TAFE-SW-CPU experiences larger overhead (up to 43%) for many benchmarks (*Bfs-1M*, *Cfd*, *RD*) due to the inability of the host CPU to hide the latency. TAFE-SW-GPU has low overhead (3% on average and max. of 10%) for most cases, indicating that TAFE-SW-GPU is an efficient and also more flexible approach to track primary array values. TAFE-HW-CPU allows transparent tracking and incurs the lowest overhead (2% on average), but requires extra hardware.

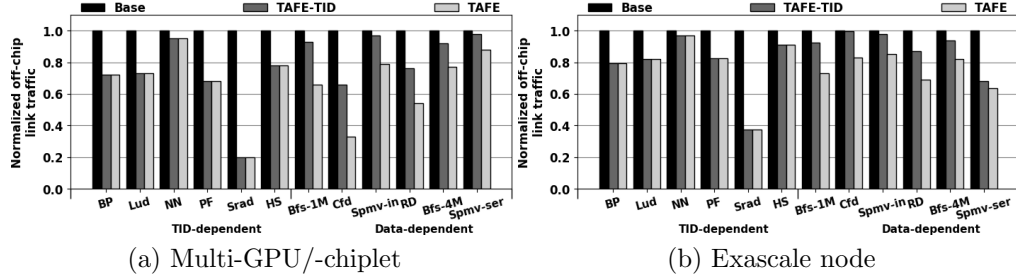


Figure 6.15: Off-chip traffic.

6.6.4 TAFE-Based Optimizations

We further study the benefits of utilizing TAFE for data/thread co-location for both system configurations. We model page co-location with compute by swapping pages prior to writing data rather than modifying memory allocation to ensure all untracked pages are in their default locations. We compare our framework against a *Base* configuration that has no knowledge about data/thread locality and uses the default threadblock scheduling and page mapping. Furthermore, we evaluate the benefits of tracking data-dependent access patterns (*Bfs-1M*, *Cfd*, *Spmv-in*, *Radix*, *Bfs-4M* and *Spmv-ser*) by comparing against a TAFE-TID scheme, which only tracks static, thread ID-dependent access patterns. This is similar to prior work [115]. In TAFE-TID, data-dependent data structures are treated as being accessed by all modules.

Off-Chip Link Traffic Fig. 6.15 shows the off-chip link traffic across different benchmarks normalized against *Base* for both configurations. The off-chip link traffic is the total amount of data moved across all off-chip (inter-module)

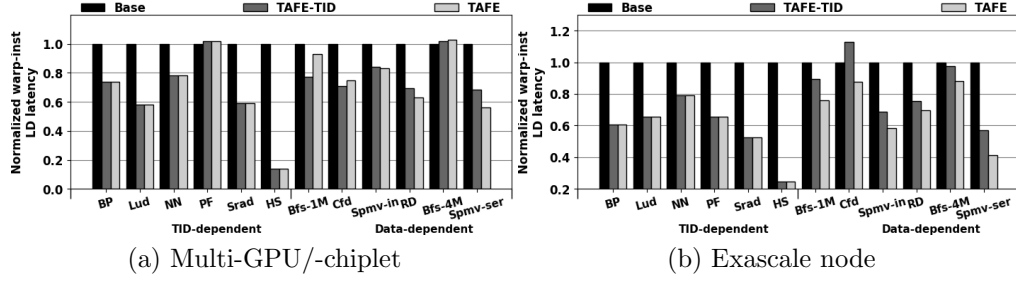


Figure 6.16: Warp memory load latency.

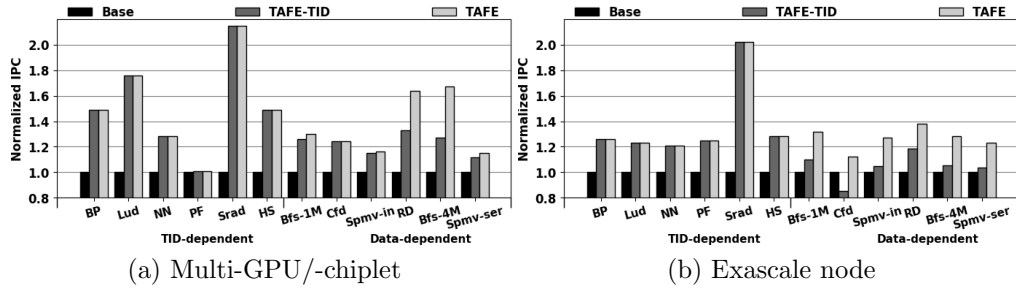


Figure 6.17: IPC.

links during the course of the application. As expected, the overall traffic decreases due to improved mapping. On average, TAFE reduces the off-chip link traffic by 33% (up to 80%) for the multi-GPU/-chiplet and 23% (and up to 62%) for exascale node configuration. In data-dependent benchmarks, TAFE reduces traffic by 34% and 24% on average while TAFE-TID reduces it by 13% and 10%, respectively.

Warp Memory-Load Latency Fig. 6.16 compares the average time taken by a warp-instruction reading global memory to complete (all 32 thread requests must complete) for both configurations. For the multi-GPU/-chiplet

case, both TAFE and TAFE-TID achieve similar reduction in warp memory-load latency (on average 21%), but in some cases TAFE-TID achieves higher reduction than TAFE (e.g. *Bfs-1M*, *Cfd*). This is because the latency is dominated by on-chip accesses for such cases. For exascale node system, TAFE-TID reduces warp memory-load latency by 29% on average while TAFE reduces it by 36%. In data-dependent benchmarks, TAFE-TID and TAFE achieve 16% and 30% reduction, respectively.

Performance Fig. 6.17 shows the average SM IPC across different benchmarks normalized against the *Base* case for both configurations. On an average, TAFE provides 45% (up to 2.1x) and 32% (and up to 2x) IPC improvement for multi-GPU/-chiplet and exascale node cases, respectively. This improvement is both because of fewer off-chip accesses and lower off-chip link congestion. In multi-GPU/-chiplet case, TAFE achieves on avg. 10% (up to 32%) higher performance than TAFE-TID for data-dependent benchmarks, while it performs 22% (up to 31%) better than TAFE-TID in exascale node case. The improvement is lower in multi-GPU/-chiplet system because the off-chip accesses have lower performance impact due to the smaller number of distributed modules.

6.6.5 Evaluation on Multi-GPU System

We evaluate the extended TAFE with compiler support for data placement optimizations on a real multi-GPU system consisting of an Intel Xeon

Table 6.7: Benchmarks for compiler-assisted TAFE.

Benchmark	Input
Maximal Independent Set (MIS) [23]	europe_osm [1]
Connected Components (CC) [23]	europe_osm [1]
Mri-Q [107]	large [107]
Pagerank (PGRNK) [23]	europe_osm [1]
Stencil [107]	default [107]
SGEMM [107]	medium [107]
LBM [107]	large [107]

CPU E5-2620 v4 host with 4x Nvidia 1080-Ti GPUs, running CUDA v10.0 (driver: 460.32.03). We use LLVM v10 [67] for developing the compiler passes. The list of benchmarks evaluated are listed in Table 6.7.

Since the aim is to evaluate the benefits of using TAFE for data placement, the single GPU benchmarks are modified to run on multiple GPUs by splitting the kernel grid across available GPUs (4 in our setup) and launching individual kernels on each device with no explicit memory mapping/place-ment code. The benchmarks are changed to use CUDA unified memory i.e `cudaMalloc()` calls are replaced with `cudaMallocManaged()` calls. This al-lows user-transparent memory management, where the CUDA driver is re-sponsible for mapping and moving data across host CPU and GPUs within the system. In our evaluation, we use the estimated TAFs to provide data-placement hints for CUDA unified memory usage [102] . The TAFs are used to derive the address range argument required by the `cudaMemAdvise()` calls to specify the addresses accessed by individual GPUs. We compare it against a baseline which uses the default driver behavior to place data without any user input.

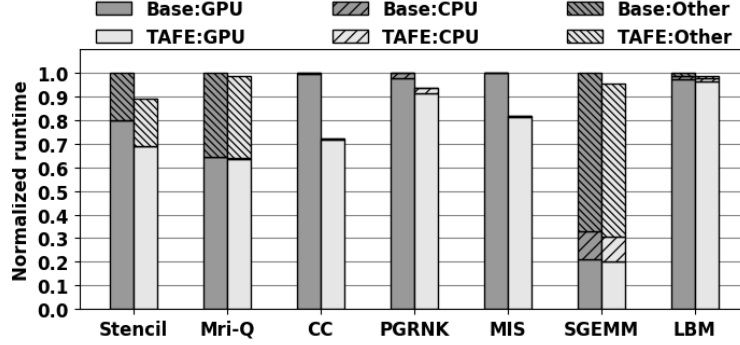


Figure 6.18: Normalized multi-GPU runtime.

Fig. 6.18 shows the normalized runtime of the benchmarks with data-placement hints using TAFE. The *GPU* component of runtime includes the kernel duration, computations within host code is captured within the *CPU* component and time spent in file I/O and CUDA API calls are captured as Other. We can see a 10% reduction in runtime on average. The address ranges for data-placement hints were derived using compiler-assisted TAFE and did not require the programmer to understand the access patterns in the application code. The runtime improvement achieved shows that compiler-assisted TAFE minimizes programmer overhead while capturing data/thread locality effectively.

6.7 Summary and Conclusions

In this chapter, we target the problem of estimating, in a scalable manner, the relationship between threads in GPGPU systems and both static, data-independent and dynamic, data-dependent addresses accessed by them

prior to kernel launches. We introduce the concept of thread address footprints (TAFs) and TAF relations as a means to concisely represent this relationship. We propose TAFE, a framework for evaluating TAF relations and estimating TAFs. Our framework constructs TAF relations using a combination of static attributes and dynamic data dependencies collected through application code and dependency tracking. We show that dynamic data dependencies can be tracked in GPU software with low overhead and further provide an optional hardware extension to capture data-dependencies. The TAFs can be used by a OS/runtime to compute TAFs prior to kernel launch. We evaluate TAFE’s accuracy and demonstrate its benefits by using TAF information for data/threadblock co-location for multiple NUMA GPU system configurations. Further, we develop compiler support for TAFE to minimize programmer overhead and improve its applicability. Our evaluations show that TAFE achieves 91% estimation accuracy across all benchmarks. Moreover, TAFE can accurately track data-dependent access patterns (only 1.5% average error) while incurring low (3%) overhead. Performing page/threadblocking mapping using TAFE improves performance by 45% and 32%. For data-dependent applications, TAFE assisted mapping can achieve 10% and 22% speedup compared to data-independent schemes across different configurations. Our evaluations further show that TAFE-based data placement hints reduce application runtime by 10% on a real multi-GPU system while minimizing the required programmer effort.

Chapter 7

Summary, Conclusions and Future Research

In this chapter, we summarize our contributions and key results, based on which we then further discuss possible future research directions.

7.1 Dissertation Summary

In this dissertation, we propose optimizations aimed at addressing both the cause and effect of data movement for GPU-based NUPIM systems. We show that scalable and hardware-assisted tracking of dynamic system and application behavior can significantly enhance the effectiveness and efficiency of existing data movement optimizations. We tackle the data movement issue by 1) managing the data traffic through network prioritization, 2) reducing the bits to represent data using cacheline compression, and 3) eliminating the need to move data by better capturing locality. Firstly, for network traffic management, we show that dynamic knowledge about network congestion and application behavior collected through low-overhead, scalable methods can better mitigate the adverse performance and energy impact of network traffic. Results show that using such schemes achieve on average 16% higher performance while saving up to 29% system energy. Secondly, for cacheline

compression, we show that knowledge about the spatial and temporal variation in cacheline utilization can be exploited to increase compression ratios and reduce interconnect traffic. We found that utilization-aware compression can reduce link traffic by 42% while boosting performance by 16%. Thirdly, for capturing locality, we show that tracking the access patterns within the application code using software-, compiler- and hardware-assisted tracking methods can be a scalable yet accurate way of capturing locality. The information collected in this way can enable wide range of optimizations such as thread/data co-location. Our results show that the proposed framework to estimate thread access footprints achieves 91% accuracy and the data mapping optimizations performed using the information achieve 45% and 32% performance improvement across different NUMA configurations.

7.2 Future Work

In this section, we discuss future research directions.

For reducing the amount of data moved, as an alternative to traditional compression, we can leverage the fact that computations are cheaper than memory accesses, and therefore aim to replace the latter with the former. Statistical and machine learning based techniques can be used to accurately or approximately fit curves over datasets, thereby representing data as a function. The function takes in an easily derivable input (e.g. address offset, neighbouring memory content) and generates the corresponding data element. The function (curve fitted) can then be evaluated to compute the re-

quired data value effectively reducing the data moved at the cost of additional computations.

For TAFE, the compiler-assisted variant can be expanded to handle access patterns with control-divergence and validate its applicability by evaluating more benchmarks and applications. Further, not all array accesses have equal contribution to performance. It is better to only track the performance-critical array accesses and perform optimizations based on them to eliminate unnecessary tracking overhead and prevent noisy information to partitioning and mapping heuristic. Currently, TAFE expects the user to mark the arrays to be tracked. Instead, light-weight performance models [42, 92] can be used to identify critical array accesses for TAFE to track.

Our current evaluation only uses TAFE information to optimize data placement by adjusting page mapping or providing memory usage-hints to the runtime. The full potential of TAFE for thread/data co-location can only be achieved by performing both thread and data mapping based on estimated TAFs. There have been prior work which allow the users to write a single kernel and transparently split the work across multiple devices [68] without burdening the user. They consider data-transfer cost to decide whether to partition the kernel or not but don't optimize the partitioning to reduce the data-transfer. For kernels with iterative thread ID and data-dependent access patterns, grouping non-sequential threadblocks can result in lower data-transfer than a sequential partition. Information from TAFE can enable optimal threadblock grouping and mapping, thus boosting the performance gains

with no additional programmer overhead. Further, TAFs can in turn also be used for prefetching pages in unified, e.g. CUDA-managed memory systems, transformations of irregular (scatter/gather) patterns into regular remote memory accesses via local page caching and re-arrangement, shared data replication or reducing coherency traffic [126, 115, 19]. The impact of TAFE in enabling such optimizations can be explored in future.

Bibliography

- [1] SuiteSparse Matrix Collection. <https://sparse.tamu.edu/>.
- [2] ATI and NVIDIA face off-obliquely. <https://www.cnet.com/news/ati-and-nvidia-face-off-obliquely>, October 2009.
- [3] clSPARSE documentation. <http://clmathlibraries.github.io/clSPARSE/index.html>, 2016.
- [4] NVIDIA cuSPARSE CUDA toolkit documentation. <https://docs.nvidia.com/cuda/cusparse/index.html>, 2019.
- [5] NVIDIA nvGRAPH CUDA toolkit documentation. <https://docs.nvidia.com/cuda/nvgraph/index.html>, 2019.
- [6] A. Singh, W. J. Dally, A. K. Gupta, and B. Towles. GOAL: a load-balanced adaptive routing algorithm for torus networks. In *International Symposium on Computer Architecture (ISCA)*, 2003.
- [7] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. In *International Symposium on Computer Architecture (ISCA)*, 2010.
- [8] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and

- R. Das. Compute caches. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.
- [9] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *International Symposium on Computer Architecture (ISCA)*, 2015.
- [10] M. Ahn and E. J. Kim. Pseudo-circuit: Accelerating communication for on-chip interconnection networks. In *International Symposium on Microarchitecture (MICRO)*, 2010.
- [11] A. Aithal and S. Bharadwaj. Sparse-matrix-vector-multiplication documentation. <https://github.com/aneesh297/Sparse-Matrix-Vector-Multiplication>, 2018.
- [12] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *International Symposium on Computer Architecture (ISCA)*, 2004.
- [13] A. Arunkumar, S. Lee, V. Soundararajan, and C. Wu. LATTE-CC: Latency tolerance aware adaptive cache compression management for energy efficient GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
- [14] A. Arunkumar, S. Lee, and C. Wu. Id-cache: instruction and memory divergence based cache management for GPUs. In *International Symposium on Workload Characterization (IISWC)*, 2016.

- [15] 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. In *International Conference on Parallel Architectures and Compilation (PACT)*, 2015.
- [16] E. Azarkhish, C Pfister, D. Rossi, I. Loi, and L. Benini. Logic-base interconnect design for near memory computing in the smart memory cube. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(1):210–223, January 2017.
- [17] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. High performance AXI-4.0 based interconnect for extensible smart memory cubes. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.
- [18] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.
- [19] A. Basu, S. Puthoor, S. Che, and B. M. Beckmann. Software assisted hardware cache coherence for heterogeneous processors. In *International Symposium on Memory Systems (MEMSYS)*, 2016.
- [20] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

- [21] J. C. Beard. The sparse data reduction engine: Chopping sparse data one byte at a time. In *International Symposium on Memory Systems (MEMSYS)*, 2017.
- [22] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin. Memory access patterns: The missing piece of the multi-GPU puzzle. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.
- [23] M. Burtcher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *International Symposium on Workload Characterization (IISWC)*, 2012.
- [24] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
- [25] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens. System and circuit level power modeling of energy-efficient 3D-stacked wide I/O DRAMs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013.
- [26] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

- [27] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramanian. Managing dram latency divergence in irregular GPGPU applications. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.
- [28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*, 2009.
- [29] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. C-Pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1196–1208, 2010.
- [30] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Transactions on Computer Systems*, 22(2):214–280, May 2004.
- [31] Hybrid Memory Cube Consortium. Hybrid memory cube specification 2.1. Technical report, 2015.
- [32] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, 1993.

- [33] H. Dang and B. Schmidt. The sliced COO format for sparse matrix-vector multiplication on CUDA-enabled GPUs. In *International Conference on Computational Science (ICCS)*, 2012.
- [34] S. N. Das, K. Punniyamurthy, M. Tomei, and B. M. Beckmann. Device and method for cache utilization aware data compression, U.S. Patent 10838727, Nov. 2020.
- [35] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
- [36] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Locality vs. balance: Exploring data mapping policies on numa systems. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2015.
- [37] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *International Conference on Supercomputing (ICS)*, 2009.
- [38] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *International Symposium on Computer Architecture (ISCA)*, 2005.

- [39] M. Farrens, B. Wetmore, and A. Woodruff. Alleviation of tree saturation in multistage interconnection networks. In *SC*, 1991.
- [40] 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 *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2006.
- [41] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture (ISCA)*, 2001.
- [42] C. Gerum, O. Bringmann, and W. Rosenstiel. Source level performance simulation of gpu cores. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.
- [43] S. Ghose, H. Lee, and J. F. Martínez. Improving memory scheduling via processor-side load criticality information. In *International Symposium on Computer Architecture (ISCA)*, 2013.
- [44] P. Gratz, B. Grot, and S. W. Keckler. Regional congestion awareness for load balance in networks-on-chip. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2008.
- [45] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using CUDA. In *International Conference on High*

Performance Computing (HiPC), 2007.

- [46] M. Harris, S. Sengupta, and J. D. Owens. CUDA-based parallel radix sort. http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/radix_sort_cuda.cc.
- [47] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2007.
- [48] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *International Symposium on Computer Architecture (ISCA)*, 2016.
- [49] C. Huang and V. Nagarajan. Increasing cache capacity via critical-words-only cache. In *International Conference on Computer Design (ICCD)*, 2014.
- [50] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *Symposium on VLSI Technology (VLSIT)*, 2012.
- [51] X. Jian, P. K. Hanumolu, and R. Kumar. Understanding and optimizing power consumption in memory networks. In *International Symposium*

on *High Performance Computer Architecture (HPCA)*, 2017.

- [52] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally. Network congestion avoidance through speculative reservation. In *International Symposium on High-Performance Comp Architecture (HPCA)*, 2012.
- [53] N. Jiang, D.U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, J. Kim, and W. J. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.
- [54] N. Jiang, J. Kim, and W. J. Dally. Indirect adaptive routing on large scale interconnection networks. In *International Symposium on Computer Architecture (ISCA)*, 2009.
- [55] Jun Yang, Youtao Zhang, and R. Gupta. Frequent value compression in data caches. In *International Symposium on Microarchitecture (MICRO)*, 2000.
- [56] I. Karlin, A. Bhatele, B. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, , and D. Wong. LULESH programming model and performance ports overview. Technical report, Lawrence Livermore National Lab, 2012.
- [57] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17,

2011.

- [58] G. Kim, C. Kim, J. Jeong, M. Parker, and J. Kim. Contention-based congestion management in large-scale networks. In *International Symposium on Microarchitecture (MICRO)*, 2016.
- [59] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *International Symposium on Computer Architecture (ISCA)*, 2008.
- [60] J. Kim, M. Sullivan, E. Choukse, and M. Erez. Bit-plane compression: Transforming data for better compression in many-core architectures. In *International Symposium on Computer Architecture (ISCA)*, 2016.
- [61] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram. Cta-aware prefetching and scheduling for GPU. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2018.
- [62] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram. Access pattern-aware cache management for improving data utilization in GPU. In *International Symposium on Computer Architecture (ISCA)*, 2017.
- [63] A. Kumar et al. Express virtual channels: towards the ideal interconnection fabric. In *International Symposium on Computer Architecture (ISCA)*, 2007.
- [64] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-cache: Adaptive blocks for eliminating waste in

- the memory hierarchy. In *International Symposium on Microarchitecture (MICRO)*, 2012.
- [65] W. Kwon, S. Yoo, J. Um, and S. Jeong. In-network reorder buffer to improve overall NoC performance while resolving the in-order requirement problem. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009.
- [66] T. Lang and L. Kurisaki. Nonuniform traffic spots (NUTS) in multistage interconnection networks. *Journal of Parallel and Distributed Computing*, 10(1):55–67, September 1990.
- [67] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.
- [68] J. Lee, M. Samadi, Y. Park, and S. Mahlke. SKMD: single kernel on multiple devices for transparent CPU-GPU collaboration. *ACM Transactions on Computer Systems*, 33(3), August 2015.
- [69] S. Lee and C. Wu. CAWS: criticality-aware warp scheduling for GPGPU workloads. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014.
- [70] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: enabling energy optimizations

- in GPGPUs. In *International Symposium on Computer Architecture (ISCA)*, 2013.
- [71] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal. Locality-aware CTA clustering for modern GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [72] C. Li, Y. Yang, Z. Lin, and H. Zhou. Automatic data placement into GPU on-chip memory resources. In *International Symposium on Code Generation and Optimization (CGO)*, 2015.
- [73] D. Li and T. M. Aamodt. Inter-core locality aware memory scheduling. *IEEE Computer Architecture Letters*, 15(1):25–28, 2016.
- [74] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *International Symposium on Microarchitecture (MICRO)*, 2009.
- [75] Z. Li, J. Wu, L. Shang, R. P. Dick, and Y. Sun. Latency criticality aware on-chip communication. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009.
- [76] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *Interna-*

tional Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008.

- [77] S. Lloyd and M. Gokhale. In-memory data rearrangement for irregular, data-intensive computing. *Computer*, 48(8):18–25, 2015.
- [78] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [79] A. Mariano, M. Diener, C. Bischof, and P. O. A. Navaux. Analyzing and improving memory access patterns of large irregular applications on numa machines. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2016.
- [80] NNSA ASC 2014 PI MEETING. The AMD FASTFORWARD project. <https://asc.llnl.gov/fastforward/AMD-FF.pdf>, February 2014.
- [81] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2012.
- [82] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh. Heterogeneous memory architectures: A HW/SW approach for

- mixing die-stacked and off-package memories. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.
- [83] Micron. 4gb: x4 x8 x16 ddr3 sdram. https://classes.engineering.wustl.edu/cse362/images/8/8b/4Gb_DDR3_SDRAM.pdf, 2011.
- [84] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans. Beyond the socket: NUMA-aware GPUs. In *International Symposium on Microarchitecture (MICRO)*, 2017.
- [85] B. Panda and A. Sez nec. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *International Symposium on Microarchitecture (MICRO)*, 2016.
- [86] R. Panda, Y. Eckert, N. Jayasena, O. Kayiran, M. Boyer, and L. K. John. Prefetching techniques for near-memory throughput processors. In *International Conference on Supercomputing (ICS)*, 2016.
- [87] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.
- [88] G. F. Pfister and V. A. Norton. “hot spot” contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, 1985.

- [89] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler support for selective page migration in NUMA architectures. In *International Conference on Parallel Architectures and Compilation (PACT)*, 2014.
- [90] M. Poremba, I. Akgun, J. Yin, O. Kayiran, Y. Xie, and G. H. Loh. There and back again: Optimizing the interconnect in networks of memory cubes. In *International Symposium on Computer Architecture (ISCA)*, 2017.
- [91] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A heterogeneous CPU-GPU simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.
- [92] K. Punniyamurthy, B. Boroujerdian, and A. Gerstlauer. Gatsim: Abstract timing simulation of gpus. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017.
- [93] K. Punniyamurthy, S. N. Das, and A. Gerstlauer. Cacheline utilization-aware link traffic compression for modular GPUs. In *International Conference on VLSI Design and Embedded Systems (VLSID)*, 2020.
- [94] K. Punniyamurthy and A. Gerstlauer. Exploring non-uniform processing in-memory architectures. In *Workshop on Hardware/Software Techniques for Minimizing Data Movement (MinMove)*, 2017.

- [95] K. Punniyamurthy and A. Gerstlauer. Off-chip congestion management for gpu-based non-uniform processing-in-memory networks. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020.
- [96] K. Punniyamurthy and A. Gerstlauer. TAFE: thread address footprint estimation for capturing data/thread locality in GPU systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020.
- [97] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2007.
- [98] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *International Symposium on Microarchitecture (MICRO)*, 2013.
- [99] T. G. Rogers, M. O’Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *International Symposium on Microarchitecture (MICRO)*, 2013.
- [100] J. B. Rothman and A. J. Smith. The pool of subsectors cache design. Technical Report UCB/CSD-99-1035, EECS Department, University of California, Berkeley, Jan 1999.

- [101] E. Rubin, E. Levy, A. Barak, and T. Ben-Nun. MAPS: optimizing massively parallel applications using device-level memory abstraction. *ACM Transactions on Architecture and Code Optimization*, 11(4), December 2014.
- [102] N. Sakharnykh. Unified memory on Pascal and Volta. <https://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf>, 2017.
- [103] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.
- [104] V. Sathish, M. J. Schulte, and N. S. Kim. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.
- [105] Y. Sazeides and J. E. Smith. The predictability of data values. In *International Symposium on Microarchitecture (MICRO)*, 1997.
- [106] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. 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.

- [107] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, V. Chang, N. Anssari, and W. W. Hwu G. D. Liu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois, at Urbana-Champaign, 2012.
- [108] ASCAC Subcommittee. The opportunities and challenges of exascale computing. Technical report, U.S. Dept. of Energy, 2010.
- [109] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *USENIX Annual Technical Conference*, 1996.
- [110] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy. Data movement aware computation partitioning. In *International Symposium on Microarchitecture (MICRO)*, 2017.
- [111] P. Tsai, C. Chen, and D. Sánchez. Adaptive scheduling for systems with asymmetric memory hierarchies. In *International Symposium on Microarchitecture (MICRO)*, 2018.
- [112] E. Tune, Dongning Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.
- [113] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Micheliogiannakis, Ann S. Almgren, and John Shalf. TiDA: High-level pro-

- gramming abstractions for data locality management. In *International Supercomputing Conference (ISC)*, 2016.
- [114] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan. Design and analysis of an APU for exascale computing. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.
- [115] N. Vijaykumar, E. Ebrihami, K. Hsieh, P. B. Gibbons, and O. Mutlu. The locality descriptor: A holistic cross-layer abstraction to express data locality in GPUs. In *International Symposium on Computer Architecture (ISCA)*, 2018.
- [116] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *International Symposium on Computer Architecture (ISCA)*, 2018.
- [117] T. Vogelsang. Understanding the energy consumption of dynamic random access memories. In *International Symposium on Microarchitecture (MICRO)*, 2010.
- [118] H. Wang, L. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. In *International Symposium on Microarchitecture (MICRO)*, 2003.

- [119] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing*, 4(1), August 2017.
- [120] C. Weis, A. Mutaal, O. Naji, M. Jung, A. Hansson, and N. Wehn. DRAMSpec: a high-level DRAM timing, power and area exploration tool. *International Journal of Parallel Programming*, 45(6):1566–1591, December 2017.
- [121] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt. gpucc: An open-source GPGPU compiler. In *International Symposium on Code Generation and Optimization (CGO)*, 2016.
- [122] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *International Conference on Languages and Compilers for Parallel Computing (LCPC)*, 2009.
- [123] C. Yang and A. R. Lebeck. *The Push Architecture: a Prefetching Framework for Linked-Data Structure*. PhD thesis, USA, 2001. AAI3095642.
- [124] Y. J. Yoon, N. Concer, M. Petracca, and L. Carloni. Virtual channels vs. multiple physical networks: A comparative analysis. In *Design Automation Conference (DAC)*, 2010.

- [125] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: throughput-oriented programmable processing in memory. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.
- [126] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

Index

- Abstract, viii
- Acknowledgments*, v
- Address Footprint Accuracy*, 107
- Background*, 11
- Basic TAFE Framework*, 75
- Basic TAFE Limitations*, 86
- Benchmarks*, 38
- Bibliography*, 143
- Cacheline Utilization-Aware Link Compression*, 46
- Compiler support for TAFE*, 97
- Congestion Behavior*, 25
- Congestion Management in Off-Chip Networks*, 24
- Critical Path Detection*, 30
- CUALiT*, 50
- CUALiT Overview*, 47
- Data Compression and Variable Caches*, 18
- Data Movement*, 2
- Data/Thread Locality Tracking and Optimizations*, 21
- Decompression*, 58
- Dedication*, iv
- Dissertation Summary*, 119
- Divergence Behavior*, 28
- Divergence-based Prioritization*, 34
- Dynamic Data Dependency Tracking*, 81
- Eliminating Need to Move Data*, 6
- Energy Results*, 43, 63
- Evaluation*, 42, 59, 103
- Evaluation on Multi-GPU System*, 115
- Experimental Setup*, 37
- Extended TAFE Limitations*, 101
- Future Work*, 120
- Hardware Architecture*, 88
- Hardware-Assisted Dependency Tracking*, 88
- Input Data-Dependent Accesses*, 73
- Introduction*, 1
- Limitations*, 91
- Link Width Management*, 35
- Mitigating Data Traffic Impact*, 4
- Network Congestion Management*, 15
- NUPIM Networks*, 25
- Off-chip Congestion Management*, 29
- Off-chip Traffic*, 62
- Path-based Prioritization*, 33
- Performance Results*, 42, 64
- Reducing Amount of Data Moved*, 5
- Related Work*, 15
- Simulator Setup*, 104
- Slack-aware compression*, 56
- Spatial Utilization-aware Pruning*, 55

Summary and Conclusions, 44, 65,
 117
*Summary, Conclusions and Future
 Research*, 119

TAFE Generalization, 92
*TAFE Generalization and Automa-
 tion*, 92
TAFE Hardware Overhead, 110
TAFE Overview, 68
TAFE-Based Optimizations, 113
*Thread Address Footprint Estima-
 tion*, 67
Thread Address Footprints, 70
Thread ID-Dependent Accesses, 72

Utilization Prediction, 51
Utilization Predictor, 61

Variation in Cacheline Utilization,
 48

Vita

Kishore Punniyamurthy received the B.Tech degree in Electrical and Electronics Engineering from the National Institute of Technology, Tiruchirappalli in 2014 and the MS degree in Electrical and Computer Engineering from University of Texas at Austin in 2017, where he is currently pursuing his Ph.D. degree. During the course of his education, he has interned at Intel (2015), Samsung (2016) and AMD Research (2018 and 2020). His research interests are GPUs, network of PIMs specifically managing and reducing data movement within a network of PIMs/near-data processors.

Permanent address: 2501 Lake Austin Blvd
Austin, Texas 78703

This dissertation was typeset with $\text{\LaTeX}^\dagger$ by the author.

[†] $\text{\LaTeX}$ is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's $\text{\TeX}$ Program.