

Spring 2021

Rethinking Cache Hierarchy And Interconnect Design For Next-Generation Gpus

Mohamed Assem Abd ElMohsen Ibrahim

William & Mary - Arts & Sciences, mohamedassemibrahim@gmail.com

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Ibrahim, Mohamed Assem Abd ElMohsen, "Rethinking Cache Hierarchy And Interconnect Design For Next-Generation Gpus" (2021). *Dissertations, Theses, and Masters Projects*. William & Mary. Paper 1627047836.

<http://dx.doi.org/10.21220/s2-a01c-6214>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Rethinking Cache Hierarchy and Interconnect Design for Next-generation GPUs

Mohamed Assem Abd ElMohsen Ibrahim

Cairo, Egypt

Bachelor of Science, Cairo University, 2010
Master of Science, Cairo University, 2016

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

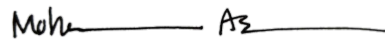
Department of Computer Science

College of William & Mary
May 2021

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy



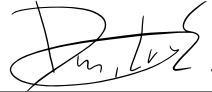
Mohamed Assem Ibrahim

Approved by the Committee, May 2021



Committee Chair

Adwait Jog, Assistant Professor, Computer Science
College of William & Mary



Dmitry Evtushkin, Assistant Professor, Computer Science
College of William & Mary



Bin Ren, Assistant Professor, Computer Science
College of William & Mary



Andreas Stathopoulos, Professor, Computer Science
College of William & Mary



Asit Mishra, Senior Deep Learning Computer Architect
NVIDIA

ABSTRACT

To match the increasing computational demands of GPGPU applications and to improve peak compute throughput, the core counts in GPUs have been increasing with every generation. However, the famous memory wall is a major performance determinant in GPUs. In other words, in most cases, peak throughput in GPUs is ultimately dictated by memory bandwidth. Therefore, to serve the memory demands of thousands of concurrently executing threads, GPUs are equipped with several sources of bandwidth such as on-chip private/shared caching resources and off-chip high bandwidth memories. However, the existing sources of bandwidth are often not sufficient for achieving optimal GPU performance. Therefore, it is important to conserve and improve memory bandwidth utilization.

To achieve this goal, this dissertation focuses on improving on-chip cache bandwidth by managing cache line (data) replication across L1 caches via rethinking the cache hierarchy and interconnect design. First, this dissertation shows that efficient inter-core communication can exploit data replication across the L1s to unlock an additional source of on-chip bandwidth, which we call remote-core bandwidth. We propose to exploit this remote-core bandwidth by investigating: a) which data is replicated across cores, b) which cores have the replicated data, and c) how to fetch the replicated data as soon as possible. Second, this dissertation shows that if data replication is eliminated (or reduced), then the L1s can effectively cache more data, leading to higher hit rates and more on-chip bandwidth. We propose designing a shared L1 cache organization, which restricts each core to cache only a unique slice of the address range, eliminating data replication. We develop lightweight mechanisms to: a) reduce the inter-core communication overheads and b) to identify applications that prefer the private L1 organization and hence execute them accordingly. Finally, to improve the performance, area, and energy efficiency of the shared L1 organization, this dissertation proposes a DC-L1 (DeCoupled-L1) cache, an L1 cache separated from the GPU core. We show how the decoupled nature of the DC-L1 caches provides an opportunity to aggregate the L1s and enables low-overhead efficient data placement designs. These optimizations reduce data replication across the L1s and increase their bandwidth utilization.

Altogether, this dissertation develops several innovative techniques to improve the efficiency of the GPU on-chip memory system, which are necessary to address the memory wall problem. The future work will explore other designs and techniques to improve on-chip bandwidth utilization by considering other bandwidth sources (e.g., scratchpad and L2 cache).

TABLE OF CONTENTS

Acknowledgments	v
Dedication	vi
List of Tables	vii
List of Figures	viii
1 Introduction	2
1.1 Problem Statement	3
1.2 Opportunity	4
1.3 Contributions	7
1.3.1 Unlocking Remote-core Bandwidth in GPUs	7
1.3.2 A Case for Shared L1 Caches in GPUs	7
1.3.3 A Case for Aggregated Decoupled L1 Caches in GPUs	8
1.4 Dissertation Organization	8
2 General Background on Graphics Processing Units (GPUs)	10
3 Analyzing and Leveraging Remote-core Bandwidth in GPUs	14
3.1 Introduction	15
3.2 Motivation and Analysis	17
3.2.1 Inter-core Communication Message Flow	17
3.2.2 Potential Benefits of Remote-core Bandwidth	18

3.3	Inter-core Communication in GPUs	20
3.3.1	Baseline Architecture and Communication Fabric	21
3.3.2	Communication Knobs: Probe Coverage and Probe Rate	21
3.3.3	Which Remote Cores Have the Data?	22
3.3.4	How is the Data Fetched?	24
3.3.5	Is the Data Shared?	27
3.3.6	Implementation Details	30
3.4	Experimental Setup	33
3.5	Experimental Results	33
3.5.1	Sensitivity Studies	40
3.6	Related Work	42
3.7	Chapter Summary	43
4	Analyzing and Leveraging Shared L1 Caches in GPUs	44
4.1	Introduction	45
4.2	Motivation and Analysis	47
4.2.1	Analysis of Wasted L1 Cache Space	47
4.2.2	A Case for Shared L1 Caches	49
4.3	Shared L1 Caches: Design, Analysis, and Optimizations	51
4.3.1	Terminology and Address Mapping	51
4.3.2	Shared L1 Caches Design	51
4.3.3	Performance Analysis and Optimizations	54
4.4	A Dynamic Mechanism for Handling Private-friendly Applications	57
4.4.1	Sampling Methodology	58
4.4.2	Sampled Metrics	61
4.4.3	Hardware Overhead	64
4.5	Experimental Setup	64

4.6	Experimental Results	65
4.6.1	Sensitivity Studies	69
4.6.2	Case Study: Deep-Learning Applications	72
4.6.3	Case Study: Crossbar-based Shared L1 Cache Design	72
4.7	Related Work	76
4.8	Chapter Summary	77
5	Analyzing and Leveraging Decoupled L1 Caches in GPUs	78
5.1	Introduction	79
5.2	Motivation and Analysis	82
5.2.1	Inefficiency#1: Cache Line Replication across L1 Caches	82
5.2.2	Inefficiency#2: Low L1 Cache Utilization	84
5.2.3	Solution: Decouple and Aggregate L1 Caches	85
5.3	Decoupled-L1 (DC-L1) Design	85
5.4	Private DC-L1 Caches	88
5.4.1	Designing Private DC-L1 Caches	88
5.4.2	Evaluating Private DC-L1 Caches	88
5.5	Shared DC-L1 Caches	93
5.5.1	Designing Shared DC-L1 Caches	93
5.5.2	Evaluating Shared DC-L1 Caches	94
5.6	Clustered Shared DC-L1 Caches	96
5.6.1	Designing Clustered Shared DC-L1 Caches	97
5.6.2	Evaluating Clustered Shared DC-L1 Caches	98
5.6.3	Frequency-boosted Clustered Shared DC-L1 Design	101
5.7	Experimental Setup	103
5.8	Experimental Results	103
5.8.1	Sensitivity Studies	109

5.9	Related Work	112
5.10	Chapter Summary	114
6	Conclusion and Future Research Directions	115
6.1	Summary of Dissertation Contributions	115
6.2	Future Research Directions	117
6.2.1	Short-term Direction: Improved On-chip Bandwidth Utilization	117
6.2.2	Long-term Direction: Graphics Pipeline & Workloads	117
	Bibliography	118
	Vita	138

ACKNOWLEDGMENTS

“He is not thankful to God who is not thankful to people” – Prophet Muhammad

This dissertation would not have been possible without the help and support of many people. First, I would like to thank my dissertation advisor, Adwait Jog, for guiding me along this journey. He always encouraged me to pursue interesting questions, carry out deep investigations, and submit papers in the top venues. Adwait always made time for me and helped me with his insights. Most importantly, he cared about my personal development and welfare. I also thank his family for their support. I would also like to thank Onur Kayiran, who alongside Adwait laid the foundations of this dissertation. I deeply admire his detailed advice, useful insights, and hands-on collaboration. On top of being a collaborator in this dissertation, I was lucky to have Onur as a mentor during my internships at AMD Research. I am highly indebted to Adwait and Onur for what they have given me so far.

I am grateful to the members of my dissertation committee, Andreas Stathopoulos, Bin Ren, Dmitry Evtvushkin, and Asit Mishra for their generous support and attentive feedback. Their insightful comments and helpful discussions have greatly improved this dissertation. Also, I would like to thank my internship mentors at AMD Research for providing a stimulating and encouraging environment for me to work. In this regard, I thank Onur Kayiran, Yasuko Eckert, Gabriel Loh, Shaizeen Aga, Nuwan Jayasena, Brad Benton, and Jieming Yin. My internships at AMD Research had been great learning experience. I am also thankful to my mentors, Hatem El-Boghdadi, Tamer ElBatt, Nayer Wanas, and Hesham ElGamal who sparked my interest in research and led me to start this journey.

My Ph.D. research was performed using the computing facilities at William & Mary. I thank the entire technical staff for managing those facilities. In particular, I thank Eric Walter for always helping me with my requests. I also thank the entire administrative staff of the Computer Science department; Vanessa Godwin, Dale Hayes, and Jacquelyn Johnson, for being efficient, professional, and above all caring.

I was fortunate to have a group of dear friends who enriched my Ph.D. experience. I am especially thankful to my lab-mates Gurunath Kadam, Hongyuan Liu, and Haonan Wang for the support, companionship, and the countless interesting discussions. I would also like to thank all my research collaborators and colleagues with whom I have interacted.

Finally, I have been gifted with a caring, patient, and supportive family. I am thankful to my father Assem Abd ElMohsen, my mother Faida Kamel, my brother Tarek Assem for their unconditional love, support, encouragement, and everything they did. I am deeply grateful to my beloved wife Yousra Rehab for her endless patience, care, and support throughout the Ph.D. journey. In the end, I want to thank my precious son, Yehia, for all the joy he brought to my life. I am forever indebted to my family.

To my family

LIST OF TABLES

3.1	Probing/Communication scenarios.	18
3.2	Configuration parameters of the simulated GPU.	34
4.1	Configuration parameters of the simulated GPU.	64
5.1	NoC size and peak L1 bandwidth reduction under different private DC-L1 configurations.	89
5.2	Configuration parameters of the simulated GPU.	103

LIST OF FIGURES

1.1	The scope of inter-core locality (i.e., cache line replication) in GPUs. . .	4
1.2	Potential performance benefits of (a) unlocking remote-core bandwidth with ideal inter-core communication and (b) eliminating replication across L1 caches with a hypothetical cache design. Results are normalized to a baseline with the conventional memory hierarchy. . .	6
2.1	Overview of GPU architecture.	11
2.2	Private and shared cache organizations.	12
3.1	L1 read miss handling when inter-core communication is (a) disabled and (b) enabled.	18
3.2	Performance benefits of remote-core bandwidth for various scenarios. Section 3.4 has the details on the experimental methodology.	19
3.3	The loss of inter-core locality (remote hit rate) for various scenarios. . .	20
3.4	Illustrating (a) inter-core locality (normalized to the PP scenario) and (b) request bandwidth (normalized to the IP(5,1,1)) under IP($C,1,1$) averaged across the evaluated applications.	22
3.5	Supplier heat map for (a) C-BFS, (b) R-CFD, (c) S-SpMV, (d) L-BH, and (e) PP-2MM under the baseline 6×6 mesh NoC. L2 partitions (and MCs) are highlighted using thick borders. For these applications, the maximum value in the heat map is 1.94× the minimum, on average. . .	23

3.6	Performance of selection criteria under IP(27,1,1) averaged across the evaluated applications. Results are normalized to a baseline with no inter-core communication.	24
3.7	Performance of C-BFS with index-based and supplier-based selectors under IP(27,S,P). Results are normalized to a baseline with no inter-core communication.	24
3.8	Illustrating how two-level probing works. The dotted red lines represent the order of searching the cores in this scenario. Gray nodes are connected to L2s and MCs.	26
3.9	Performance with supplier-based selector and two-level probing under IP(C,S,P) averaged across the evaluated applications. Results are normalized to the baseline with no inter-core communication.	27
3.10	Remote hits vs. remote misses for different PC under (a) C-BFS, and (b) P-2DCONV. The numbers on each bar represent the total remote read accesses per PC.	28
3.11	Two-bit PC-based sharing predictor. t_S refers to a <i>Sharing</i> transition, while t_{NS} refers to a <i>Non-Sharing</i> transition.	29
3.12	Hardware organization of our proposal. The shaded components are used for inter-core communication. The gray components are added to support our proposal.	30
3.13	The effect of the proposed schemes on IPC and reply bandwidth. . . .	36
3.14	Illustrating (a) precision and recall for RP(5,2,0.5) and (b) effect of prediction precision on IP.	37
3.15	The effect of the proposed schemes on the request and reply NoC link utilization.	39

3.16	The effect of the proposed schemes on applications with low inter-core locality. Results are normalized to the baseline with no inter-core communication.	39
3.17	Sensitivity studies on (a) CTA scheduling, (b) NoC resources, and (c) NoC size.	40
4.1	Performance of the evaluated applications in terms of L1 miss rate, line replication ratio, and IPC improvement under 16× the L1 cache size (normalized to baseline). The left-hand y-axis represents cache line replication ratio and raw L1 miss rate.	48
4.2	Private and shared cache organizations.	49
4.3	IPC and reply bandwidth of a hypothetical cache design that eliminates data replication across local L1 caches. Results are normalized to the private L1 baseline. Section 4.5 has the details on the experimental methodology.	50
4.4	L1 miss rate of a hypothetical cache design that eliminates data replication across local L1 caches. Results are normalized to the private L1 baseline.	50
4.5	Request/Reply flow in a shared L1 organization. The L1 Arbitrator and the In/Out queues, shown in black in (a), are newly added to support our proposal. Dashed lines represent L1/MSHR bypassing.	52
4.6	Performance of a realistic shared L1 organization. Results are normalized to the private L1 baseline.	55
4.7	Fraction of useful bytes within a cache line.	56
4.8	Non-shared-friendly applications under Shared++. Results are normalized to the private L1 baseline.	57
4.9	Sampling phase of the dynamic scheme.	58

4.10	Effect of different metrics on the dynamic scheme.	62
4.11	The effect of the proposed solutions on IPC. Results are normalized to the private L1 baseline.	66
4.12	The effect of the proposed solutions on IPC as S-curve. Results are normalized to the private L1 baseline.	66
4.13	The effect of the proposed solutions on L1 miss rate. Results are normalized to the private L1 baseline.	67
4.14	The effect of the proposed solutions on number of replicas.	68
4.15	Execution timeline under DynEB for (a) C-BFS and (b) C-NN. <i>S</i> refers to a sampling phase. <i>Ex-Sh</i> and <i>Ex-Pr</i> refer to an execution under Shared++ and Private, respectively.	69
4.16	Sensitivity study on L1 access latency.	71
4.17	Sensitivity study on core and uncore components.	71
4.18	Analyzing deep-learning applications in terms of L1 miss rate, line replication ratio, and performance improvement under DynEB.	73
4.19	Performance of the shared L1 organization in terms of IPC under a crossbar-based system. <i>NS</i> refers to non-shared-friendly applications. Results are normalized to a crossbar-based system with private L1 organization.	74
4.20	The effect of the proposed solutions on IPC as S-curve. Results are normalized to a crossbar-based system with private L1 organization.	75
4.21	Crossbar-based system scalability.	75

5.1	Performance of the evaluated applications in terms of IPC improvement under $16\times$ the L1 cache (normalized to baseline), L1 miss rate, and cache line replication ratio. The left-hand y-axis represents replication ratio and raw L1 miss rate. The experimental methodology is detailed in Section 5.7.	83
5.2	Illustrating performance of a hypothetical cache design that eliminates replication across L1s on replication-sensitive applications (normalized to baseline).	84
5.3	L1 cache data port and NoC link utilization.	85
5.4	Decoupled-L1 (DC-L1) node and NoC design.	86
5.5	Pr40 design.	89
5.6	Performance under private DC-L1 design. Results are normalized to the private L1 baseline.	90
5.7	Illustrating the potential performance of DC-L1 under private DC-L1 design. Results are normalized to the private L1 baseline.	91
5.8	NoC area and static power under private DC-L1 design. Results are normalized to the private L1 baseline.	92
5.9	Sh40 design.	93
5.10	Performance under Sh40. Results are normalized to the private L1 baseline.	95
5.11	Performance of replication-insensitive applications under Sh40. Results are normalized to the private L1 baseline.	96
5.12	Sh40+C10 design.	98
5.13	Performance of Sh40 under different cluster counts. Results are normalized to the private L1 baseline.	99
5.14	NoC area and static power under different cluster counts. Results are normalized to the private L1 baseline.	100

5.15 Performance of poor-performing replication-insensitive applications under Sh40+C10. Results are normalized to the private L1 baseline.	101
5.16 Illustrating the maximum frequency of various crossbars.	102
5.17 The effect of the proposed designs on IPC. Results are normalized to the private L1 baseline.	105
5.18 The effect of the proposed designs on IPC as S-curve. Results are normalized to the private L1 baseline.	106
5.19 The effect of the proposed designs on L1 miss rate. Results are nor- malized to the private L1 baseline.	106
5.20 The effect of the proposed solutions on number of replicas.	107
5.21 L1/DC-L1 cache data port utilization.	107
5.22 NoC power under Sh40+C10+Boost. Results are normalized to the private L1 baseline.	108
5.23 Area overhead/savings under Sh40+C10+Boost. Results are normal- ized to the private L1 baseline.	109
5.24 Sensitivity study on using a hierarchical crossbar.	110
5.25 Sensitivity study on L1/DC-L1 access latency.	111

Rethinking Cache Hierarchy and Interconnect Design for
Next-generation GPUs

Chapter 1

Introduction

Graphics Processing Unit (GPU) architectures have become increasingly popular for general purpose computing because of their capability to provide high compute throughput at a competitive power budget [5, 89, 69, 6, 74, 113, 68, 70, 99, 2, 1, 18, 56, 118]. Therefore, GPUs have been employed in many computing systems, including many supercomputers on Top500 [125] and Green500 lists [124]. Additionally, GPUs have become the default choice for accelerating a large number of data-parallel applications in various fields such as artificial intelligence [98, 117, 88], image/video processing [120, 31, 94, 95], physical simulation [108, 14, 139], gene sequencing [85, 110, 129, 112, 127], financial computing [111, 87, 83], medical science [101, 118, 84, 29], and security-critical cryptographic workloads [25, 76, 32, 130].

Unlike CPUs, which typically have limited multi-threading capabilities, GPUs launch thousands of threads across multiple GPU cores to exploit the high thread-level parallelism available in GPGPU applications and to mask the long memory latency of a single thread. To serve these thousands of concurrently executing threads with their required data, GPUs are dependant on the high bandwidth provided by their memory system. The GPU memory system consists of two levels of on-chip hardware-managed caches, namely a per-core private local L1 cache and a shared L2 cache. Additionally, the GPU memory system

employs an on-chip software-managed scratchpad¹. The L2 cache is shared across multiple cache banks connected to memory channels. These memory channels work independently of each other and the connected L2 bank(s) caches the data served by the corresponding memory channel.

1.1 Problem Statement

To match the increasing computational demand of GPGPU applications and to improve the peak compute throughput, the core counts in GPUs have been increasing as the manufacturing technology improves. Another recent trend of scaling up the GPUs enables more kernels from the same or different applications to concurrently execute on the same GPU to improve the peak compute throughput. However, GPU performance ultimately faces the famous memory wall [138]. For many workloads, these attempts to boost the GPU compute capability is limited by memory bandwidth. As a matter of fact, the scaling in the core count and the multi-kernel executions aggravates the problem. In particular, with more cores and kernels, more threads can be launched to take advantage of such compute power, which puts more pressure on the GPU memory hierarchy. Therefore, as mentioned before, to serve the increasing memory-demands of thousands of concurrently executing threads, many modern GPUs utilize several sources of bandwidth such as on-chip private/shared caches and off-chip high bandwidth memories. However, the existing sources of on-chip bandwidth are usually not sufficient and not efficiently utilized [47, 51, 96, 105, 60, 3]. A straightforward approach to mitigate this issue is to scale the on/off-chip memory resources. However, memory bandwidth scaling is limited by the cost and power budgets of the system [144, 126, 19, 93]. Additionally, the off-chip memories are constrained by their I/O specifications. This makes the memory bandwidth a scarce and valuable resource. Therefore, it is important to conserve and improve memory bandwidth utilization. In this dissertation, we observe that the conventional memory hierarchy and

¹Scratchpad is referred to as Shared Memory in CUDA terminology.

interconnect design leads to several limitations and inefficient utilization of the available on-chip bandwidth sources. We aim to understand these limitations and inefficiencies to improve the utilization of the on-chip sources of bandwidth instead of (along with) naively scaling the resources.

1.2 Opportunity

With the conventional GPU memory hierarchy, each GPU core is attached to a private local (L1) cache and all cores in the GPU share banked L2 caches [10]. All cores (and L1 caches) are connected to L2 banks. However, there is no explicit connection/communication between cores. We observe that such a cache/interconnect design limits the efficient use of the on-chip cache bandwidth. Specifically, we observe that the data required by one of the GPU cores (e.g., L1 read misses) can be also found in the local L1 caches of other remote cores. This is essentially *inter-core locality* [59, 72, 28, 55], which exists mainly because other GPU cores have previously requested the same data (exact sharing) or nearby data in the same cache line (false sharing) and placed it in their local caches [59]. We formally define inter-core locality as the ratio of L1 misses that can be found in other L1 caches to total L1 misses. To illustrate the volume of inter-core locality (i.e., cache line replication), we evaluate 37 applications from different benchmark suites [86, 20, 26, 100, 48] under a baseline with the conventional memory hierarchy. Figure 1.1 shows that most of the evaluated applications possess varying degrees of inter-core locality (up to 95% for AlexNet).

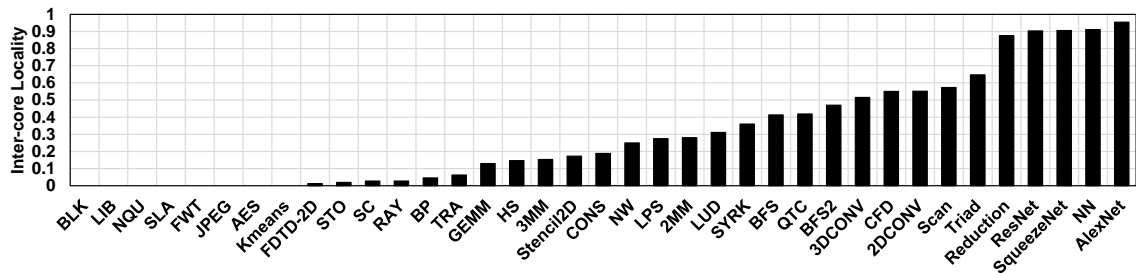


Figure 1.1: The scope of inter-core locality (i.e., cache line replication) in GPUs.

In this dissertation, we analyze and leverage such cache line (data) replication across the L1 caches to improve their utilization, hence boost the on-chip bandwidth and the overall throughput. Specifically, we consider inter-core locality as a double-edged sword. On the one hand, inter-core locality can provide additional source of bandwidth. In particular, the remote cores are capable of supplying the data (given the replicated cache line is stored in their L1 caches) and hence a potential source of memory bandwidth, namely remote-core bandwidth. On the other hand, inter-core locality can be considered as waste of caching resources, which leads to less effective cache capacity, and ultimately leading to more requests to L2/memory and increasing the bandwidth consumption. Therefore, in this dissertation, we dissect inter-core locality to boost the on-chip bandwidth in two ways.

1. We consider inter-core locality as a positive opportunity and utilize the data copies in the remote cores as additional sources of bandwidth. Instead of serving the L1 read miss at the L2, we can fetch the required data from a nearby remote core. To show the potential performance benefits of unlocking remote-core bandwidth, we evaluate an ideal scenario where a given core can fetch replicated data from a remote core in zero cycles (i.e., no communication overhead). Figure 1.2a shows the reply bandwidth received by each core in terms of L1 reply bandwidth and L2 reply bandwidth, and the performance in terms of IPC.² Both metrics are normalized to a baseline with the conventional memory hierarchy. We observe an increase in the L1 reply bandwidth because of utilizing the remote-core bandwidth. This results in up to $2.1\times$ improvement in performance. Additionally, we observe that exploiting remote-core bandwidth reduces the pressure on L2 and off-chip memory. To this end, this dissertation investigates how to efficiently exploit remote-core bandwidth to boost on-chip bandwidth and overall performance.

2. We consider inter-core locality as a waste and aim to reduce/eliminate it to effectively

²L1 (L2) reply bandwidth is the number of replies received from L1 (L2) over the total execution time.

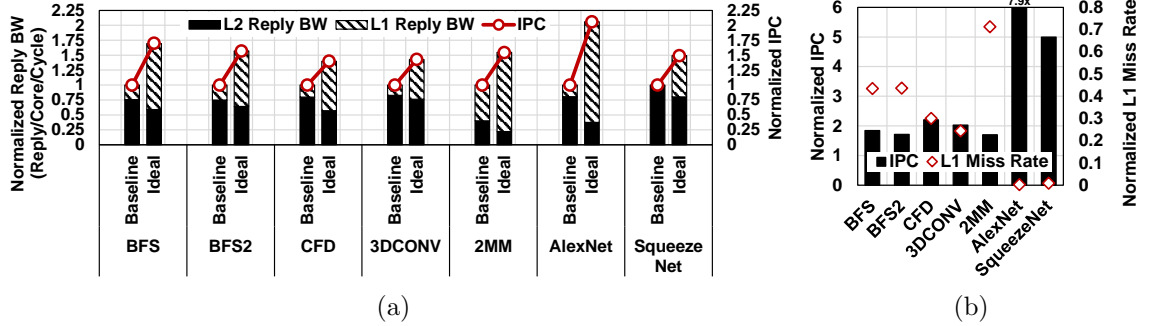


Figure 1.2: Potential performance benefits of (a) unlocking remote-core bandwidth with ideal inter-core communication and (b) eliminating replication across L1 caches with a hypothetical cache design. Results are normalized to a baseline with the conventional memory hierarchy.

increase the on-chip caching capacity, hence cache more data. For that, we study cache and interconnect designs so that the GPU cores (and their L1s) collectively reduce (or eliminate) data replication. This leads to higher hit rates, which improve the on-chip bandwidth and overall performance. To show the potential performance benefits of eliminating data replication across the L1 caches, we evaluate a hypothetical cache design where all GPU cores access a single L1 cache (while maintaining the total L1 cache capacity and bandwidth) to ensure no replication. Figure 1.2b shows the performance scope under such impractical system in terms of IPC and L1 miss rate normalized to a baseline with the conventional memory hierarchy. We observe a significant drop in the L1 miss rate by up to 99%, which translates to a performance boost of up to $7.9\times$. To this end, this dissertation investigates how to efficiently eliminate/reduce data replication to boost on-chip bandwidth and overall performance.

To enable both of these directions, we need to efficiently enable inter-core communication in GPUs. The following section details our contributions.

1.3 Contributions

This dissertation focuses on utilizing the inter-core locality in the means mentioned above. First, this dissertation shows how to utilize the replicated data in GPUs via efficient inter-core communication to search the remote cores and fetch the required data. Second, this dissertation makes a case for shared L1 caches in GPUs to eliminate data replication across L1 caches and improve their effective capacity. Finally, this dissertation investigates a renovated cache hierarchy and interconnect in GPUs to manage data replication across the L1 caches while reducing the area and energy requirements. In the rest of this section, we discuss these contributions in details.

1.3.1 Unlocking Remote-core Bandwidth in GPUs

This dissertation proposes to efficiently coordinate the data movement across cores in GPUs to exploit the remote-core bandwidth [41]. However, this dissertation finds that the efficient detection and utilization of the remote-core bandwidth presents several challenges. Specifically, this dissertation addresses:

- Which data is shared (replicated) across GPU cores? To this end, this dissertation proposes a novel PC-based Sharing Predictor.
- Which remote cores have the shared data? To this end, this dissertation proposes a novel Supplier-based Core Selector.
- How to efficiently fetch the shared data? To this end, this dissertation proposes a novel Two-level Probing technique.

1.3.2 A Case for Shared L1 Caches in GPUs

This dissertation introduces a new shared L1 cache organization, where all GPU cores collectively cache a single copy of each cache line, eliminating data replication [39]. This is achieved by allowing each core to cache only a non-overlapping slice of the entire address range. Such a design is useful for significantly improving the collective L1 hit rates but

incurs latency overheads from additional communication when a core requests data that does not belong to its assigned address range slice. While many workloads can tolerate this additional latency, several workloads show performance sensitivities. Therefore, this dissertation develops lightweight communication optimization techniques and a run-time mechanism that considers the latency tolerance characteristics of applications to decide which applications should execute in private versus shared L1 cache configuration and re-configures the caches accordingly.

1.3.3 A Case for Aggregated Decoupled L1 Caches in GPUs

This dissertation finds that the main source of the inefficient utilization of L1 caches stems from the tightly-coupled design of GPU cores with L1s [40]. This leads to data replication (as discussed before) and low per-core L1 bandwidth utilization. To address these inefficiencies, this dissertation renovates the conventional GPU cache hierarchy by proposing a new **DC-L1 (DeCoupled-L1)** cache – an L1 cache separated from the GPU core. This dissertation shows how decoupling the L1 cache from the GPU core provides opportunities for aggregating the DC-L1s to reduce data replication across the L1s and increase their bandwidth utilization. This dissertation investigates:

- How can the DC-L1s be aggregated?
- How can the data placement be managed across the aggregated DC-L1s?
- How can the DC-L1s be efficiently connected to the GPU cores and the L2 banks?

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 provides a general background for GPUs and the GPU memory hierarchy. In Chapter 3, we present a novel set of schemes to utilize the inter-core locality via efficient inter-core communication to unlock and exploit the remote-core bandwidth. In Chapter 4, we present our novel shared L1 cache design to eliminate data replication across the L1s and show how a lightweight

dynamic scheme can improve performance across a wide set of applications. In Chapter 5, we present our novel decoupled L1 cache design and show that how we utilize such flexible design to manage data replication using clustering. Finally, in Chapter 6, we conclude this dissertation and discuss future research directions.

Chapter 2

General Background on Graphics Processing Units (GPUs)

In this chapter, we provide a general background of Graphics Processing Units (GPUs). Specifically, we focus on the baseline GPU architecture and memory hierarchy.

GPU Core Architecture. GPUs achieve high throughput as it is capable of executing a large number of threads concurrently. To facilitate this, GPUs consists of a large number of processing elements (PEs), as shown in Figure 2.1. A group of PEs are clustered into a GPU core (known as Streaming Multiprocessor (SM) in NVIDIA terminology or Compute Unit (CU) in AMD terminology). The PEs within each core are supported by a large register file and other caching resources. These private caching resources include multiple hardware-managed L1 caches (data, instruction, read-only constant, and read-only texture) and a software-managed scratchpad. Upon launching a kernel from a given GPGPU application, the threads from the launched kernel are uniformly distributed on the cores at the granularity of a cooperative thread array (CTA). Each core is capable of handling threads from multiple CTAs and executes them at the granularity of a warp (known as wavefront in AMD terminology). A warp is a collection of (usually 32) individual threads that execute in a lock-step manner on the PEs of the same core.

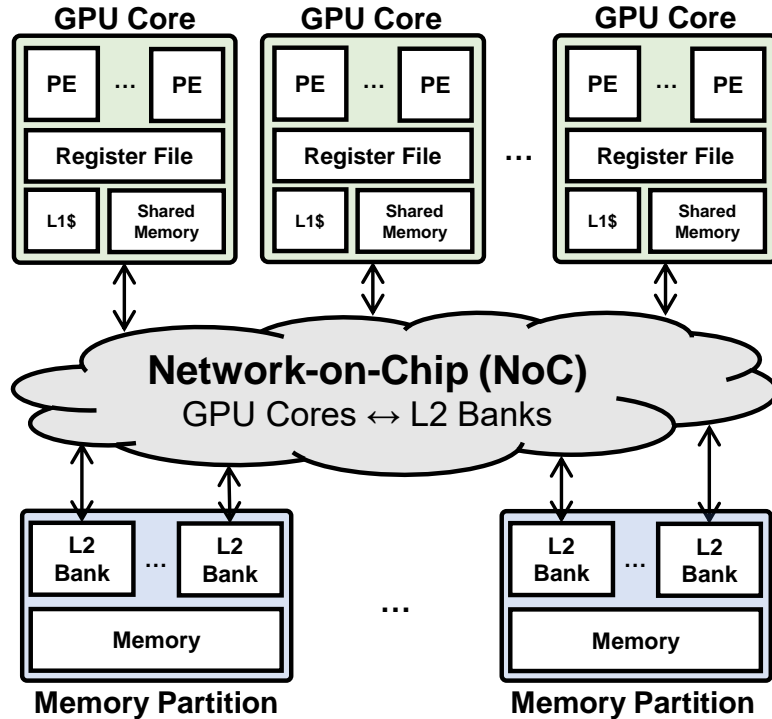


Figure 2.1: Overview of GPU architecture.

These threads from the same warp execute a single instruction at a time on different data (i.e., implement SIMD). Multiple warps residing on the same core facilitate in hiding long memory latencies by executing in a pipelined and multiplexed manner and hence improving the utilization/throughput of the core. To support such high thread-level parallelism (TLP), the PEs make use of the per-core register file for saving the context of a large number of concurrent threads to minimize the overhead of context switching.

GPU Memory Hierarchy. We consider the memory hierarchy under a generic GPU architecture consisting of many GPU cores (and their caching resources), which are connected to few memory partitions via a Network-on-Chip (NoC) as shown in Figure 2.1. Each memory partition hosts an off-chip high-bandwidth memory module to support fast data access for the large number of concurrent threads. Additionally, each memory partition is associated with a bank (or more) of the shared L2 cache for faster access

to the required data. A memory controller (MC) within each memory partition is responsible for scheduling L2 cache misses (i.e., memory requests sent from the L2 cache) to the GPU memory. The data of the running GPGPU application is interleaved across multiple memory partitions to achieve high memory bandwidth.

Private and Shared Cache Organizations. As discussed before, the L1 caches in the baseline GPU architecture are organized as private caches. However, the default for the L2 caches is shared cache organization. The difference between the private and shared cache organization lies in the placement of the data in the caches. Specifically, with a private cache organization, each L1 cache can store any cache line. For example, in Figure 2.2a, given four different address ranges represented by different shades, a private L1 cache can store any line from all four address ranges. On the other hand, with a shared organization, the entire address range is interleaved across all the L2 banks. In other words, each L2 bank caches data from a non-overlapping address range. For example, as shown in Figure 2.2b, the address range represented by white can be cached by only L2-0, and the address range represented by black can be cached by only L2-3.

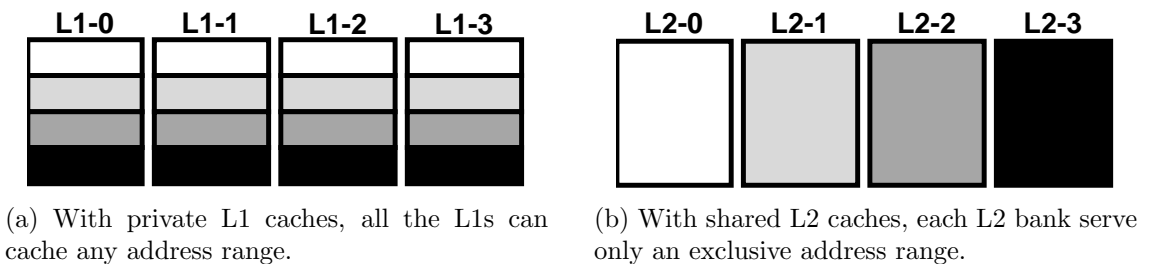


Figure 2.2: Private and shared cache organizations.

Data Locality in GPUs. Under a baseline GPU architecture, the data fetched by a GPU core exhibits one of the following locality types: no locality, intra-thread, intra-warp, inter-warp, inter-CTA, and inter-core. No data locality is observed if each thread fetches data into the cache and never reuse it. Intra-thread is observed if there is a reuse of data by the same thread. If there is data reuse across threads belonging to the same warp,

then intra-warp locality is exhibited. If data reuse extends to threads from other warps, then inter-warp locality is observed. Similarly, inter-CTA locality occurs if there is data reuse between warps that belong to different CTAs. Finally, if these CTAs are executing on different cores, then inter-core locality is exhibited.

Chapter 3

Analyzing and Leveraging Remote-core Bandwidth in GPUs

Bandwidth achieved from local/shared caches and memory is a major performance determinant in Graphics Processing Units (GPUs). These existing sources of bandwidth are often not enough for optimal GPU performance. Therefore, to enhance the performance further, we focus on efficiently unlocking an *additional* potential source of bandwidth, which we call as remote-core bandwidth. The source of this bandwidth is based on the observation that a fraction of data (i.e., L1 read misses) required by one GPU core can also be found in the local (L1) caches of other GPU cores. In this work, we propose to efficiently coordinate the data movement across cores in GPUs to exploit this remote-core bandwidth. However, we find that its efficient detection and utilization presents several challenges. To this end, we specifically address: a) which data is shared across cores, b) which cores have the shared data, and c) how we can get the data as soon as possible. Our extensive evaluation across a wide set of GPGPU applications shows that significant performance improvement can be achieved at a modest hardware cost on account of the additional bandwidth received from the remote cores.

3.1 Introduction

Graphics Processing Unit (GPU) architectures are becoming an inevitable part of every computing system [125] because of their ability to provide orders of magnitude faster execution. They have become the default choice for accelerating innovations in various fields [29, 101, 118, 84, 111, 87, 83, 88, 95] such as high-performance computing (HPC), artificial intelligence, deep learning, and virtual/augmented reality. Traditionally, GPUs have relied on bandwidth to achieve high throughput [47, 10, 44, 45, 137, 21]. However, the current sources of bandwidth such as local/shared caches, scratchpad, and memory are often not sufficient for achieving the peak GPU throughput [47, 51, 96, 105, 60, 3]. In this work, we focus on dynamically identifying and exploiting an additional source of bandwidth in GPUs, which we call as *remote-core bandwidth*. The source of this additional bandwidth stems from *inter-core locality* [59, 72, 28, 55] that allows the data required by one of the GPU cores (i.e., L1 read misses) to be also found in the local L1 caches of remote GPU cores. Our analysis shows that this additional source of bandwidth leads to significant improvement in performance, however, can only be leveraged if an efficient inter-core communication is enabled. However, there are several challenges towards designing efficient inter-core communication, which have not been addressed by prior works. In particular, this work addresses the following research questions.

I) How to determine which data can also be found in the local caches of remote cores?

Traditionally, a cache line requested by a core is always found in the GPU memory, as it stores the data required by the kernel(s). However, the requested data may or may not be found in the L1 cache of the remote cores due to static data sharing characteristics or runtime state of the caches [55, 71, 59, 72, 28]. A mechanism that correctly predicts if the data is shared would reduce unnecessary inter-core communication.

II) How to determine which cores have the data of the requester core? Even if it is known that the data is shared across cores, determining which cores have the shared data is critical. A naive approach of sending request *probes* to all the cores to fetch the

data can incur significant latency and consume interconnect bandwidth. Therefore, it is important to determine which cores are likely to have the requested data to reduce the communication overhead.

III) How to get the data as soon as possible without congesting the interconnect? Finally, it is important to search the cores such that we do not saturate the interconnect bandwidth while still reducing the search latency. This latency can be tolerated to a certain extent; however, long latencies can hurt performance [47]. Moreover, long search delays decrease the probability of finding the shared data due to cache evictions at the remote core.

Contributions: To the best of our knowledge, this is the first work that systematically addresses these questions. Specifically, this work makes the following contributions:

- We observe a bi-modal distribution of inter-core locality across different load instructions – some instructions use data that is shared across cores and some do not. We leverage this observation and use the program counter (PC) to predict which L1 read misses are likely to be satisfied by the L1 caches of remote cores.
- We develop a low-overhead mechanism that can locally predict which cores are likely to have the shared data. It is based on our key observation that the data required by a core is generally shared across *only a few* cores, which can be detected via sampling a limited number of core replies.
- We develop a novel two-level probing mechanism that searches the identified cores in parallel while considering the interconnect bandwidth consumption.
- Our combined schemes take advantage of the untapped remote-core bandwidth, leading to 21% improvement (up to 40%) in performance if the data is *a priori* known to be shared, and 10% (up to 26%) with our PC-based predictor. These results are averaged across 11 diverse GPGPU applications that exhibit inter-core locality and achieved at a modest area overhead of 0.058 mm^2 per core (determined by detailed RTL synthesis). Additionally, our proposed schemes do not affect the performance of applications that possess low inter-core locality.

3.2 Motivation and Analysis

Many important graph and HPC applications are known to be cache sensitive with significant reuse. To capture this reuse, much attention has been given to improving local cache performance in GPUs (e.g., [105, 47, 46, 42, 51]). However, limited focus is given to another type of locality, called as inter-core locality [55, 59, 72, 28] (i.e., the data required by a core can be found in the local L1 caches of other cores). Inter-core locality primarily results from each core independently requesting data without consulting the L1 cache of nearby cores. We find that in many cases, other GPU cores have previously requested the same data (exact sharing) or nearby data in the same cache line (false sharing) and placed it in their local caches [59]. Consequently, they are also capable of supplying the data and a potential source of memory bandwidth, which we refer to as *remote-core bandwidth*. To unlock this additional bandwidth, efficient inter-core communication is essential.

3.2.1 Inter-core Communication Message Flow

We first provide a high-level overview of how L1 read miss requests are routed to other cores to exploit inter-core locality. Under a baseline GPU where inter-core communication is not enabled (Figure 3.1a), a read request which misses in L1 goes through the Network-on-Chip (NoC) and accesses L2 cache. L2 cache either responds with data or forwards the request to its associated memory channel. When inter-core communication is enabled (Figure 3.1b), a read request which misses in L1 (i.e., the requester L1) can probe other L1 caches (i.e., supplier L1s).¹ An L1 read miss goes through the NoC to probe other L1 caches. If a supplier L1 has the data, it will respond with data; if not, it will send a NACK. If no supplier L1 responds with the data (or NACK) in a given amount of time (we define this as *Timeout*), the requester L1 will fall back to the default scenario shown in Figure 3.1a to probe the L2 cache.

¹The inter-core communication in our proposal is enabled for the read requests only and thus can co-exist with the existing cache coherence mechanism. A write request to a shared data in L1 is handled by the default cache coherence mechanism.

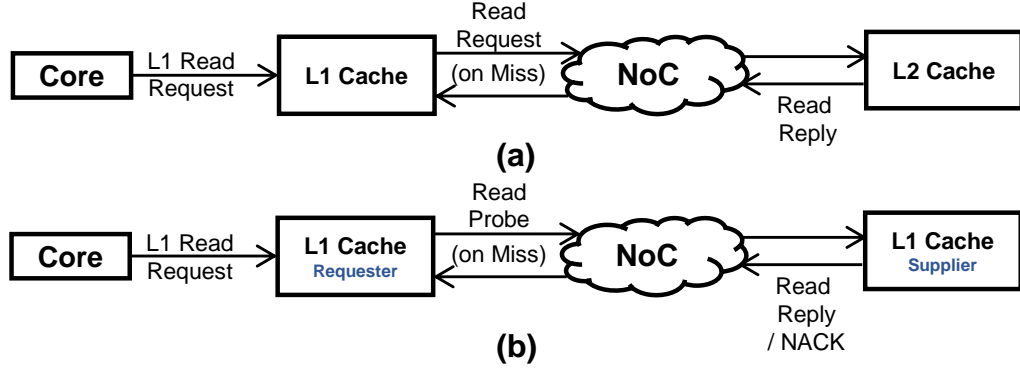


Figure 3.1: L1 read miss handling when inter-core communication is (a) disabled and (b) enabled.

3.2.2 Potential Benefits of Remote-core Bandwidth

To illustrate the benefits of inter-core communication in GPUs, we consider three different scenarios for probing other GPU cores, as tabulated in Table 3.1. These scenarios are formed based on the questions we raised in Section 3.1: (1) is the data shared?; (2) which remote cores have the data?; and (3) how should the data be fetched? We start with the assumption that the answer to the first question is known *a priori* (we will relax this assumption later in Section 3.3). In other words, we assume a perfect predictor that determine if the required data exists in the L1 cache of at least one remote core.

Table 3.1: Probing/Communication scenarios.

Scenario	Is the data shared?	Which remote cores have the data?	How is the data fetched?
Perfect Probing (PP)	Known	Known	Zero-cycle communication
Direct Probing (DP)	Known	Known	Direct communication with the nearest supplier
Naive Indirect Probing (n-IP)	Known	Search all the cores	Sequentially search the cores one-by-one

The first scenario, called as *Perfect Probing (PP)*, assumes that we oracularly know which cores have the shared data, and this data can be fetched in zero cycles (i.e., no communication overhead). In the next scenario, called as *Direct Probing (DP)*, we still assume that the location of the shared data is known, but a mechanism is required to

probe the nearest core that shares the data and fetch it. Finally, in the *Indirect Probing* mechanism (*IP*), we assume that the location of the shared data is unknown, and a single probe request has to sequentially search all remote cores one-by-one to fetch the data. This is a naive implementation of IP, and hence mentioned as *Naive IP* (*n-IP*) in Table 3.1. Section 3.3 discusses our final probing scenario (not shown in Table 3.1), called as *Realistic Probing* (*RP*), which adopts intelligent IP mechanisms to efficiently fetch data from the remote cores, and also a technique to determine if a cache line is shared by other remote cores.

Figure 3.2 shows the reply bandwidth received by each core in terms of L2 reply bandwidth and remote-core reply bandwidth, and the performance in terms of IPC (both normalized to the baseline with no inter-core communication) under the aforementioned probing scenarios. Four observations are in order. First, on average, the total reply bandwidth is higher under PP scenario compared to other scenarios. Therefore, IPC is also the maximum in this scenario. Specifically, because $IPC \propto BW/MPKI$, where MPKI is misses-per-kilo-instruction [45, 134], unlocking the remote-core bandwidth shall increase the overall available bandwidth, which in turn improves IPC. Thus, even if the overall memory bandwidth can be increased by adding more memory partitions, the additional on-chip bandwidth from remote cores can further enhance performance.

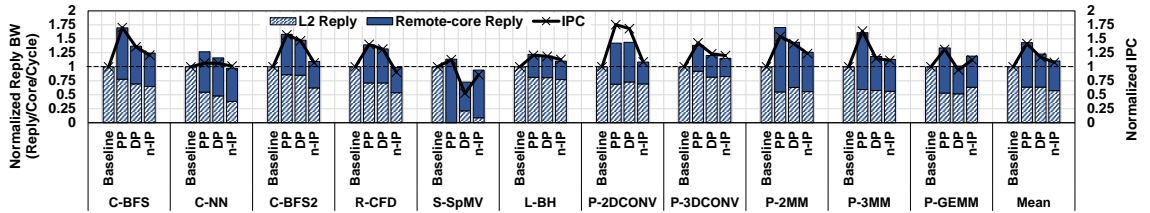


Figure 3.2: Performance benefits of remote-core bandwidth for various scenarios. Section 3.4 has the details on the experimental methodology.

Second, the remote-core bandwidth under DP is lower in many applications compared to PP. This is due to the overhead of fetching the data from remote cores. This overhead is not only in terms of latency of fetching the data; in some cases, the data is no longer

present in the cache by the time a probe reaches the remote destination. As shown in Figure 3.3, this results in a loss in remote hit rate (i.e., inter-core locality), which is defined as the ratio of replies received from the remote cores to L1 read misses. Figure 3.3 results are normalized to PP with the raw inter-core locality numbers of PP shown at the top of each application. Third, with n-IP, the overhead of naive searching is more significant because of the NoC contention, which further decreases the remote-core bandwidth of n-IP, and thus its performance.

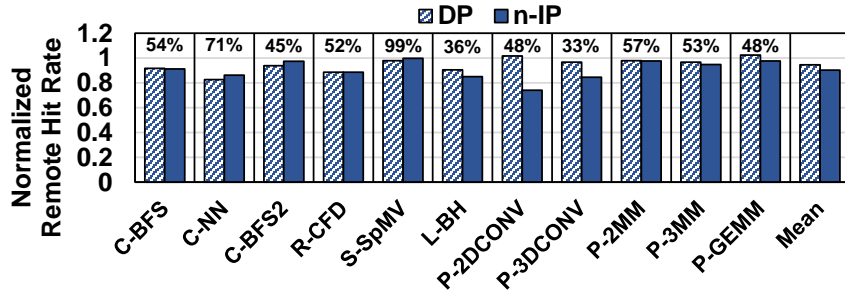


Figure 3.3: The loss of inter-core locality (remote hit rate) for various scenarios.

Finally, the reply bandwidth for P-2DConv is slightly higher with DP than PP, however, IPC with PP is higher than DP. This is attributed to the runtime state of the caches such as cache evictions [59]. Specifically, using $IPC \propto BW/MPKI$, the runtime state of the cache affects $MPKI$, which may decrease IPC. Also, using zero-cycle communication is the main performance booster in PP. In summary, utilizing remote-core bandwidth boosts overall performance and is complementary to the bandwidth received from the memory partitions.

3.3 Inter-core Communication in GPUs

In this section, we discuss the design of inter-core communication policies, which are required to exploit the inter-core locality opportunities discussed before.

3.3.1 Baseline Architecture and Communication Fabric

Our baseline GPU consists of 28 cores connected to 8 L2 slices and memory channels via NoC. Each core has a local L1 cache, which is connected to its associated NoC interface. There is a shared L2 cache that is interleaved across 8 banks. Each L2 bank is connected to a NoC interface for the incoming L2 requests and to its corresponding memory controller for forwarding the requests to memory in case of L2 misses. We use two separate NoCs: request and reply NoCs to avoid protocol deadlock [10]. The L2 requests, probes, and the NACKs use the request NoC, while the replies from cores or L2 use the reply NoC. Similar to recent works [11, 53, 142, 97] in GPUs, we model a 2D mesh NoC for connecting cores to memory channels because it inherently enables core-to-core communication. Additionally, a 2D mesh NoC is scalable as the number of cores increases because it is modular and easier to lay out on a chip [141, 10, 16].

3.3.2 Communication Knobs: Probe Coverage and Probe Rate

To address the performance overheads of inter-core communication discussed in Section 3.2, we consider modulating the number of cores to search (i.e., controlling the *probe coverage*) and/or the rate at which the cores are searched (i.e., controlling the *probe rate*). Formally, we define $IP(C,S,P)$, where S probes are sent per read miss with a probability of P ($0 \leq P \leq 1$), or $S - 1$ probes per read miss are sent with a probability of $1 - P$, to search C cores in the GPU system. For example, $IP(15,2,0.2)$ implies that a core searches 15 remote cores by sending 2 probes per request for around 20% of its L1 read misses and 1 probe per request for the rest. In the case of two (or more) probes per request, the target cores (i.e., the cores to be probed) are disjointly divided among the probes as equally as possible to be searched in parallel. For example, under $IP(15,2,0.2)$, the first probe searches 8 cores and the second probe searches 7 cores. Probe coverage is determined by the value of C and the probe rate is determined by the value of the pair (S,P) . Note that both probe coverage and rate affect the consumption of request NoC

bandwidth (Request/Core/Cycle), which is inherently limited. Therefore, it is important to control each of these parameters carefully (C , S , and P) to optimize performance.

3.3.3 Which Remote Cores Have the Data?

Effect of Probe Coverage. Figure 3.4 shows the effect of probe coverage on the remote hit rate and the request bandwidth under $IP(C,1,1)$. The request bandwidth has three components: a) requests sent to L2, b) probe requests sent to remote cores, and c) forwarded probe requests from remote cores. We observe that probing a limited number of cores can reduce the consumption of the request bandwidth at the cost of reducing inter-core locality. Therefore, it is important to carefully select the number of target cores that balances the available inter-core locality and the NoC overhead (e.g., $C = 15$ in Figure 3.4).

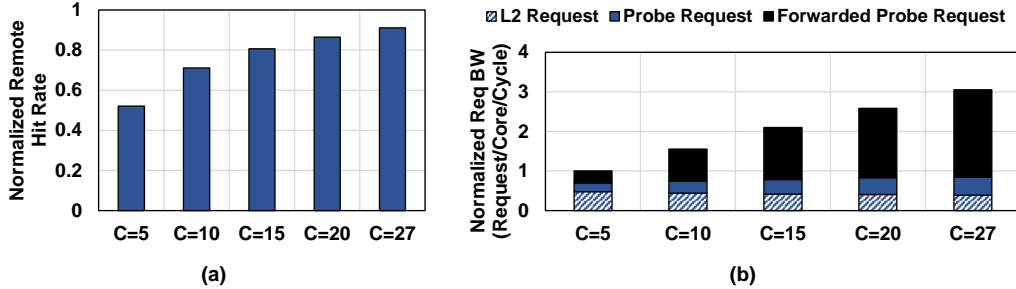


Figure 3.4: Illustrating (a) inter-core locality (normalized to the PP scenario) and (b) request bandwidth (normalized to the $IP(5,1,1)$) under $IP(C,1,1)$ averaged across the evaluated applications.

Which Cores to Probe? Our next goal is to identify the target cores. This step consists of predicting which cores have a high probability of providing the shared data and selecting a subset of them to probe. Figure 3.5 shows the heat map of cores that can supply data to requester cores for representative applications. Each cell in the heat map represents how many times a particular core is able to respond to an incoming probe with data. A requester core is any core that had at least one remote request during execution. This data is collected assuming that probes can be sent in zero cycles. We observe from this

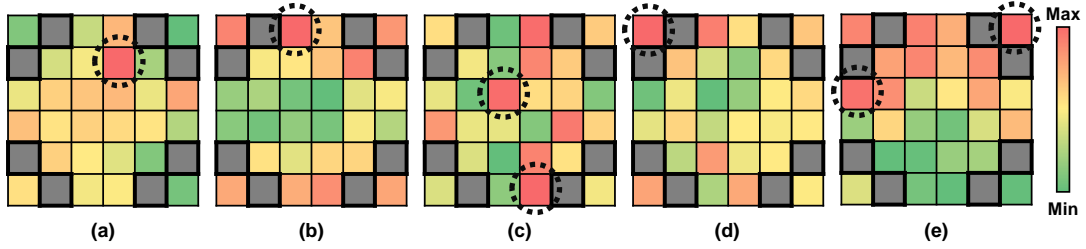


Figure 3.5: Supplier heat map for (a) C-BFS, (b) R-CFD, (c) S-SpMV, (d) L-BH, and (e) PP-2MM under the baseline 6×6 mesh NoC. L2 partitions (and MCs) are highlighted using thick borders. For these applications, the maximum value in the heat map is $1.94 \times$ the minimum, on average.

figure that some cores can provide the data more than the others. For example, in C-BFS, the highlighted core is more likely to provide the data. Similar behavior is observed in the other applications as shown in Figure 3.5. Therefore, probing the cores that have a higher probability of responding with data is potentially beneficial because it would maintain inter-core locality, and reduce the request NoC bandwidth consumption.

Selection Criteria. There are multiple design choices when selecting the set of target cores. Figure 3.6 shows performance of $IP(C=27,1,1)$ under two selector mechanisms, where 27 is the maximum number of cores that can be searched in our 28-core baseline architecture. In *index-based*, which is used in n-IP, a probe sequentially searches the cores assigned to it based on the core index in ascending order. We propose a *supplier-based* selector. In this mechanism, each core locally and periodically collects the number of data replies received from other cores. This information is then used to assign probability values for selecting the target cores.² To reduce the bias in the selection process, (1) the collected data is reset at the end of each period, and (2) the cores that have not replied with data during the current period are given a very small probability (half of the lowest collected non-zero probability) to be selected as target cores. Then, C target cores are selected for probing based on the collected and modified probability of finding data in each core. We observe from Figure 3.6 that our *supplier-based* selector outperforms the

²For example, in a four-core system, if Core1, Core2, and Core3 responded to Core0 with data 5, 3, and 2 times during a period, respectively, then Core0 will select Core1, Core2, and Core3 as target cores with 50%, 30%, and 20% probability, respectively.

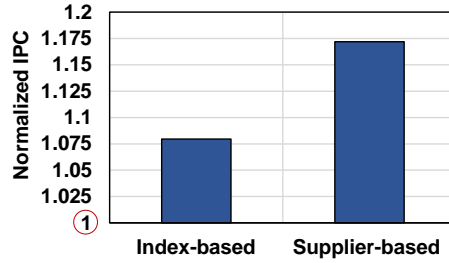


Figure 3.6: Performance of selection criteria under $IP(27,1,1)$ averaged across the evaluated applications. Results are normalized to a baseline with no inter-core communication.

index-based selector because of its ability to adapt to the dynamic changes in the sharing patterns.

3.3.4 How is the Data Fetched?

Effect of Probe Rate. We study the effect of probe rate with the help of Figure 3.7 that shows the performance of $IP(27,S,P)$ for C-BFS under index-based and supplier-based selection criteria. In the index-based case, we obtain the highest IPC when $S = 1$ and $P \leq 1$. In other words, if we send only one probe for a portion of the read miss requests, while the rest are directly sent to L2, then performance can improve; with multiple probes per request, performance drops.

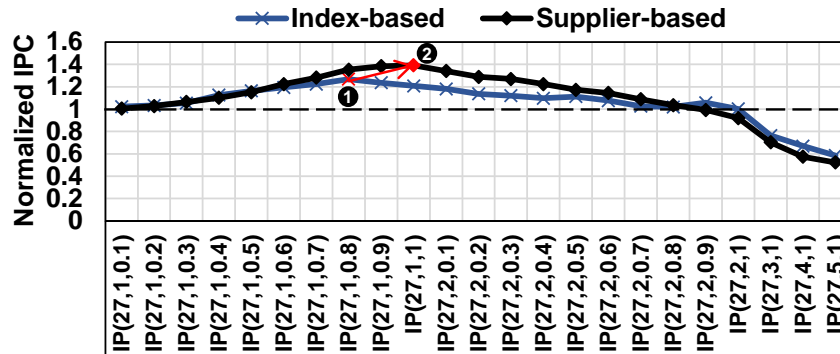


Figure 3.7: Performance of C-BFS with index-based and supplier-based selectors under $IP(27,S,P)$. Results are normalized to a baseline with no inter-core communication.

In the supplier-based case, we observe that the peak performance for C-BFS is shifted to the right (from ❶ to ❷). This confirms that selecting which cores to search first

has a positive impact on performance. However, performance still drops when using $S > 1$. This is because multiple probes can cause contention in the request NoC resources (e.g., links, buffers, virtual channel (VC) allocation, switch (SW) allocation). In addition, multiple parallel probes may lead to redundant replies, thereby congesting the reply NoC further. Therefore, it is important to modulate the probe rate carefully while handling the redundant replies.

One way to improve performance in the presence of parallel probes is to limit the number of data replies to one, so that reply NoC is not further congested. Based on this idea, we propose a novel *Two-level Probing* scheme.

Two-level Probing. Our two-level probing scheme overcomes the issue of redundant replies by leveraging two probe types. The first type is the *Leader Probe*, which looks for the data in its assigned target cores and returns once the data is found (similar to a normal probe). The second type is the *Scout Probe*, which also looks for data within its target cores; however, once it finds the data, it does not return with data. Instead, it appends the core identifier to the candidate suppliers list and then searches the rest of the assigned cores. The scout probe returns once it completes searching. If the leader does not return with the data, then the requester initiates the *second-level of probing* by injecting a leader-like probe to search all the candidate suppliers sequentially and return if it finds the data (or failed). There is a singular leader probe in our scheme, while the rest of the parallel probes are scouts.

To illustrate how two-level probing works, let us consider an example in Figure 3.8. Assume that $S = 2$; the leader probe searches the shaded cores, while the scout probe searches the others. Assuming that the data is present in cores **A**, **B**, **C**, and **D**, the leader returns with data (from **B**) after searching three cores, and the scout searches all the assigned fourteen cores and returns with candidate suppliers **A** and **D**. However, because the data is found by the leader, these candidates are ignored. In another scenario, assume that data is only found in **A** and **D**. In this case, the leader searches all the assigned cores and returns with a NACK back to the requester. The scout returns with the candidate

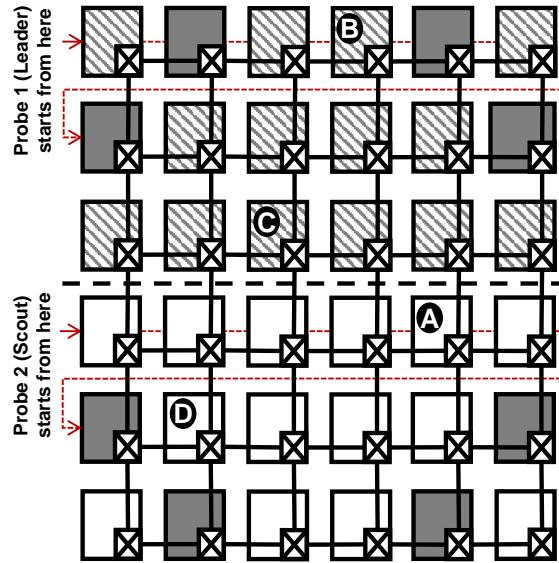


Figure 3.8: Illustrating how two-level probing works. The dotted red lines represent the order of searching the cores in this scenario. Gray nodes are connected to L2s and MCs.

suppliers (**A** and **D**), so the requester injects a leader-like probe that searches **A**. On failing to find the data (for example, evicted by the time the probe reaches **A**), it searches **D**. In summary, the advantage of two-level probing is the elimination of redundant replies from different remote L1 caches.

Discussion. Figure 3.9 shows the average performance under $IP(C, S, P)$ when S and P (probe rate) are varied, while C (probe coverage) is set to 5, 10, 15, 20, or 27. Since the request NoC bandwidth is a function of the number of probes sent and the number of cores to search, decreasing the number of target cores is expected to release more NoC resources to accommodate more probes. In that case, we observe a further shift to the right in the peak performance (i.e., we observe better performance when more than one probe search in parallel). Using $C \geq 20$, we barely observe any benefits from using $S \geq 2$. We can still get benefits from sending a mix of one or two probes, but not beyond two probes. On the other hand, using $C = 15$, we observe a lower reduction in performance even with $S \geq 2$. Both $C = 10$ and $C = 5$ lead to better performance with $S \geq 2$ compared to $C \geq 15$. To summarize, a trade-off between the number of

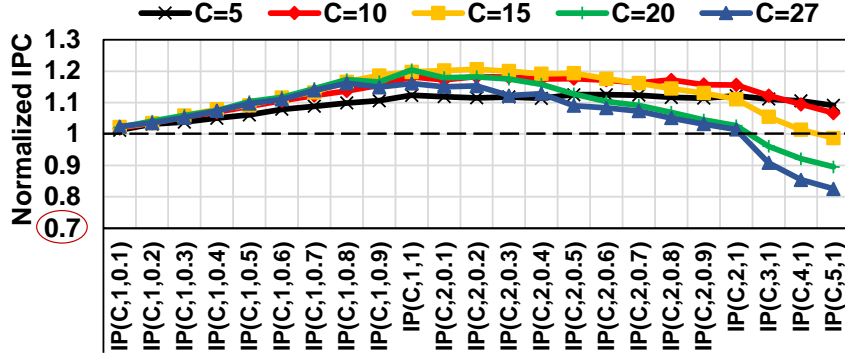


Figure 3.9: Performance with supplier-based selector and two-level probing under $IP(C, S, P)$ averaged across the evaluated applications. Results are normalized to the baseline with no inter-core communication.

cores to search and the parallel probes to inject is required to balance the overall request bandwidth and to control the forward request bandwidth.

3.3.5 Is the Data Shared?

We have so far assumed that a requester core had *a priori* knowledge of whether the data it requests is cached by remote cores. In this section, we propose a two-bit predictor that uses the Program Counter (PC) information to predict, locally at each core, if the required data exists in a remote L1 cache. If our predictor anticipates that the data is shared, the supplier-based core selector and the two-level probing techniques are utilized to search for the required data. Otherwise, the request is sent directly to L2.

Why Prediction? We start by studying the need for a predictor. From Figure 3.3, we observe that the raw volume of inter-core locality is not 100% of the read misses. Additionally, falsely assuming that a read miss is shared causes latency overhead for the request sent to L2, as probing remote L1 caches imposes a search delay. As a result, if we assume every read miss is shared, it will cause unnecessary search overhead in the cases when the data is not shared. For example, in C-BFS, the percentage of shared read miss request is around 54%. Thus, if we probe remote L1 caches on every read miss, we will end up with a failed search for 46% of the requests. In other words, almost half of the

requests will endure unnecessary delay and consume request NoC bandwidth whereas the data is not shared.

PC and Inter-core Locality. As a first step to designing a sharing predictor, we need to identify a simple local parameter to use. We investigated multiple parameters, and we found that request origin PC is a good metric to consider. Figure 3.10a shows the volume of remote hits for each PC value in C-BFS. We observe that out of nine PCs, only two have inter-core locality ($PC = 80$, $PC = 288$), and one of them ($PC = 288$) features $> 90\%$ remote hits out of 350120 remote read accesses.

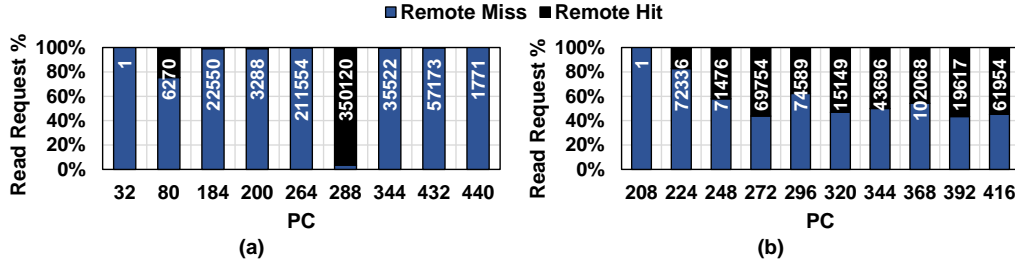


Figure 3.10: Remote hits vs. remote misses for different PC under (a) C-BFS, and (b) P-2DCONV. The numbers on each bar represent the total remote read accesses per PC.

To confirm this finding, we inspected the PTX code generated from C-BFS and found the instructions with PC values of 80 and 288. Algorithm 1 shows a pseudo code snippet from C-BFS, where the orange lines 4 and 5 correspond to PC 80 and 288, respectively. Line 5 ($PC = 288$) possesses high inter-core locality as the graph nodes connected to node $nextNodeID$ loads $g_graph_visited[nextNodeID]$, which creates several copies of the same data in different cores. We observe similar behavior in other evaluated applications.

Algorithm 1 CUDA-BFS code snippet

```

1: procedure BFS
2:    $nodeID = blockIdx.x * BLOCK\_DIM + threadIdx.x$ 
3:   for  $i$  in all edges connected to current  $nodeID$  do
4:      $nextNodeID = g\_graph\_edges[i]$ 
5:     if ( $!g\_graph\_visited[nextNodeID]$ ) then
6:        $g\_cost[nextNodeID] = g\_cost[nodeID] + 1$ 

```

This observation leads to the design of our PC-based predictor. If we keep track of the

number of probe requests sent and the core replies received per PC, then we can develop a local scheme that predicts if the data is shared.

Two-bit PC-based Predictor. Figure 3.11 shows the finite state machine for our proposed predictor. It keeps track of four different states (hence two-bit) per PC. Specifically, the states are *Strong Shared*, *Weak Shared*, *Weak Non-shared*, and *Strong Non-shared*. The predictor optimistically assumes sharing and starts from a *Strong Shared* state. If a given PC fails to show a dominant sharing behavior, it will end up in the most restrictive state *Strong Non-shared*. Each state utilizes three variables (W , S , and T). These variables are used along with the inter-core replies count (R) to decide the next state. Given state i , W_i sets the number of read misses to be considered during state i . S_i sets the number of read misses that are assumed to be shared out of W_i requests ($W_i \geq S_i$). Once W_i requests are processed, we compare the number of core replies R_i to the threshold T_i and based on that, the next state is determined. Based on the current state, if $R_i \geq T_i$, then the next state is set as the state that provides more sharing. On the contrary, if $R_i < T_i$, then the next state is the more restrictive state.

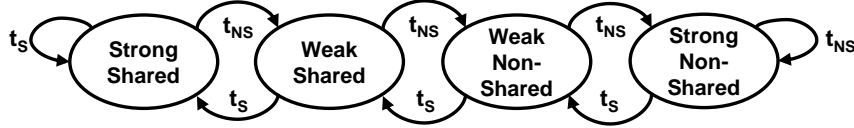


Figure 3.11: Two-bit PC-based sharing predictor. t_s refers to a *Sharing* transition, while t_{NS} refers to a *Non-Sharing* transition.

Discussion. We will discuss the effectiveness of the proposed predictor and its accuracy in Section 3.5. However, we want to point out one possible concern with our predictor. In Figure 3.10b, we show the volume of remote hits for each PC value in P-2DCONV. In contrast to C-BFS, P-2DCONV does not have a few dominant PC values. Specifically, eight out of ten PCs have around 50% remote hits. Additionally, such behavior is spread throughout the execution (not shown). As a result, it is difficult to have high accuracy under such application behavior.

3.3.6 Implementation Details

Figure 3.12 shows the architectural diagram of our proposal. We start by explaining the design choices and scenarios in our system. Then, we study the area, power, and communication overheads.

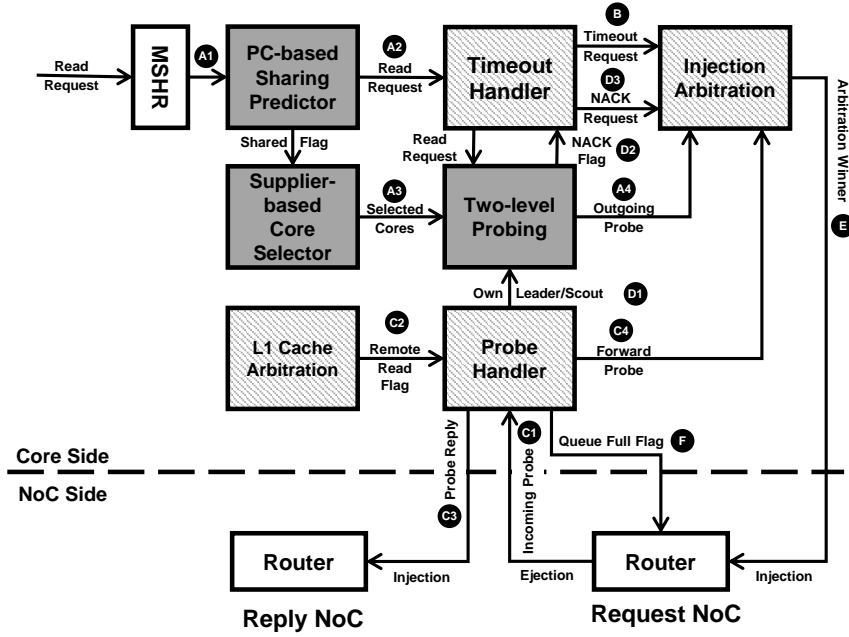


Figure 3.12: Hardware organization of our proposal. The shaded components are used for inter-core communication. The gray components are added to support our proposal.

Probe Injection. On an L1 read miss, a request is added to miss status holding register (MSHR) to be passed down the memory hierarchy. First, the request is sent to the *PC-based Sharing Predictor* (A1) to locally predict if the data is present in remote L1 caches. If the request is predicted to be shared, it will be (1) added to a queue (*Selective L2 Requests*) in the *Timeout Handler* (A2) that selectively sends the request to L2 if needed, and (2) sent to the *Supplier-based Core Selector* to select the target cores for probing (A3). Then, the *Two-level Probing* mechanism determines how many probes to send (based on S and P), assigns the target cores to the generated probes, and adds the probes to a queue (*Outgoing Probe Requests*) holding the core’s own probes for injection arbitration (A4).

Selective L2 Request Timeout. In some cases, probe requests take a long time to

return (with data or NACK). This might be due to several reasons related to NoC congestion and queuing. We need a failsafe mechanism to ensure forward progress. Therefore, for every read miss predicted as shared, a corresponding L2 request is also generated, and placed into *Selective L2 Requests* queue. Every cycle, the *Timeout Handler* checks if the head of the queue timed out. Timeout means that the injected probe(s) failed to retrieve the data from the target cores in a timely manner. In that case, the head of the *Selective L2 Requests* queue competes for injection to be sent to L2 **B**.

Handling Other Cores' Probes. On receiving an incoming probe from a remote core, the probe is added to a queue (*Incoming Probe Requests*) in the *Probe Handler* module **C1**. The forwarded probe is processed to differentiate between a leader probe, a scout probe, or a received NACK. In case of a leader or a scout, the *Probe Handler* consults the *L1 Cache Arbitration* module that prioritizes the local cache accesses over remote reads.³ In case of no local cache access, the *L1 Cache Arbitration* module informs the *Probe Handler* **C2** to check the L1 cache if the required data is cached.

If the incoming probe is a leader, and the data is not found, the probe is added to a queue (*Forwarded Incoming Probes*) to forward it to the next target core (or the requester if no more target cores). However, if the data is found locally, then a probe reply is added to a queue (*Replies to Incoming Probes*) holding the replies to be sent to the requester cores. The rationale behind this queue is to mitigate the head-of-line blocking that can occur in the *Incoming Probe Requests* queue if the reply failed to find space for injection into the reply NoC. The head of the *Replies to Incoming Probes* is pushed into the reply NoC **C3**. On the other hand, a scout probe updates its candidate supplier list if the data is found, and is always added to the *Forwarded Incoming Probes* queue to be sent to the next target core (or the requester if no more target cores). The head of the *Forwarded Incoming Probes* contends for injection into the request NoC **C4**.

In case of a returning own leader/scout, the *Probe Handler* notifies the *Two-level*

³Dual ported caches may be needed for applications where L1 bandwidth is not sufficient [57]. However, we do not observe L1 bandwidth as a bottleneck in our applications and hence arbitration is sufficient.

Probing module **D1** to keep track of the injected probes per request. If all outstanding probes are received without data reply or candidate suppliers, then the *Two-level Probing* module informs the *Timeout Handler* **D2**. If the timeout of the failed request has not fired yet, it is retrieved from the *Selective L2 Requests* queue to compete for injection to be sent to L2 **D3**.

Injection Arbitration. Our design supports different types of messages to be injected into the request NoC. Consequently, to keep the system stable, we must maintain the injection rate into the NoC. We do so by arbitrating between five different request types (ordered from the highest to the lowest priority): non-shared requests, selective L2 requests, forwarded probes, processed NACKs, and outgoing probes. The *Injection Arbitration* selects the winner of the arbitration to be injected into the request NoC based on the priorities of the competing requests **E**.

Deflection of Incoming Probes. To control the queuing delay at the core, a mechanism is required to limit the number of probes received by a given core. If the *Incoming Probe Requests* queue is full, we deflect the incoming probes at the NoC level by passing a signal from the core to the NoC router to convey the unavailability of queue space **F**. The router then deflects the probe request to its next target cores or to its requester if no more target cores exist.

Overhead. The *PC-based Sharing Predictor* supports up to 64 *PC* values. We empirically select the values of W , S , and T based on the following, $W_i = 32 \times 2^i$, $S_i = W_i/4^i$, $T_i = \text{ceil}(S_i/8)$, where $0 \leq i \leq 3$. Both *Timeout Handler* and the *Two-level Probing* modules track up to 32 outstanding requests, which is the MSHR size. The *Supplier-based Core Selector* monitors the replies from 27 remote cores (in our 28-core baseline GPU) over a period of 8192 cycles. Finally, we empirically choose 2048 cycles as the timeout value in the *Timeout Handler*. Under this timeout, only 0.7% of the probe requests fail to return with a reply (or a NACK).

To estimate the area overhead, we differentiate between the hardware used to enable inter-core communication (shaded components in Figure 3.12), and the hardware used

to optimize such communication (gray components in Figure 3.12). We faithfully synthesized the RTL design of the hardware required for the inter-core communication and our schemes using the 65nm TSMC libraries in the Synopsys Design Compiler. We use these synthesized Verilog models for the area and leakage power. Additionally, we use DSENT [119] to estimate the NoC dynamic power assuming a 45nm technology. The area overhead for inter-core communication is 0.089 mm^2 per core, while the area overhead for our schemes is 0.058 mm^2 per core. The total leakage power overhead is 2.022 mW per core. The difference in the dynamic power compared to the baseline is 0.05794 W .

In terms of communication overhead, we add 1-bit in the request to mark as a probe, and 1-bit to identify as a leader or scout. A 32-bit group identifier is added to uniquely identify the probes belonging to the same request. Additionally, up to fifteen target cores need to be searched, and each core needs $\text{ceil}(\log_2 27)$ bits, that is 75 bits required in total. All this overhead in the request fits in the baseline flit size of 32 bytes.

3.4 Experimental Setup

Simulated System. We model our schemes and inter-core communication using a cycle-level simulator – GPGPU-Sim v.3 [10]. A detailed platform configuration is described in Table 3.2.

Evaluated Applications. We use sixteen applications from five benchmarks suites (CUDA SDK (C) [86], Rodinia (R) [20], SHOC (S) [26], Lonestar (L) [15], and Poly-Bench (P) [100]) for evaluation. Eleven out of sixteen applications have inter-core locality greater than 30% (Figure 3.2). The rest of the applications have inter-core locality less than 10%.

3.5 Experimental Results

In Section 3.3, we studied the effect of both probe coverage C and probe rate (S, P) on the efficiency of the inter-core communication under a mesh-based system. We proposed

Table 3.2: Configuration parameters of the simulated GPU.

Core Features	1400MHz core clock, 28 cores (CUs), SIMD width = 32 (16×2)
Resources / Core	48KB scratchpad, 32KB register file, Max. 1536 workitems (48 wavefronts, 32 workitems/wavefront)
L1 Caches / Core	16KB 4-way L1 data cache 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
L2 Cache	8-way 128 KB/memory channel (1MB in total) 128B cache block size
Features	Memory coalescing and inter-wavefront merging enabled, immediate post dominator based branch divergence handling
Memory Model	8 GDDR5 memory controllers (MCs) FR-FCFS scheduling, 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock, Global linear address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing [38], $t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$
Interconnect	6×6 mesh topology, 700MHz interconnect clock, 32B flit size, 1 VC per port, 8 flits/VC, iSLIP VC and switch allocators

three techniques (supplier-based core selector, two-level probing, and PC-based sharing predictor) to exploit the remote-core bandwidth via efficient inter-core communication. We evaluate IP($C=15, S=2, P=0.2$), an IP scenario that incorporates supplier-based core selector and two-level probing under a perfect sharing predictor. Although IP knows the sharing information *a priori*, we investigate it thoroughly as it gives an attainable upper bound of the inter-core communication benefits via our schemes. In order to reach such upper bound, we evaluate RP($C=5, S=2, P=0.5$), a *Realistic Probing* scenario that does not need any software support and adopts PC-based sharing predictor in addition to supplier-based core selector and two-level probing.

We choose IP(15,2,0.2) as it balances the trade-off between losing inter-core locality (due to searching fewer cores) and incurring latency (due to searching more cores). In general, given an arbitrary GPU, searching 35%-55% of the cores is a valid choice to maintain the required balance under IP scenario. Also, using two probes parallelizes the search process without overwhelming the request NoC resources. For RP(5,2,0.5), we reduce the number of target cores ($C = 5$) because we use a realistic PC-based predictor.

Specifically, if we use $C = 15$, any misprediction will result in searching fifteen cores even though the data is not shared. This leads to unnecessary latency overhead for the whole data fetching process. In general, under RP, searching 15%-25% of the cores balances the inter-core locality and the request NoC bandwidth consumption. Also, to further reduce the search overhead, RP(5,2,0.5) uses a higher probe rate. We compare these mechanisms against:

- **DP** utilizes a perfect sharing predictor and sends a probe request to the oracularly known nearest sharer (Section 3.2).

- **IP(27,1,1)**, which is equivalent to n-IP, uses a perfect sharing predictor, however, it searches all the cores sequentially based on core index to find the shared data (Section 3.2).

- **Cooperative Caching Network (CCN)** [28] uses a ring NoC to connect all the cores. On a read miss, CCN traverses the ring and searches the cores sequentially. To limit the search overhead, a throttling scheme based on the ratio between replies received and requests sent, over a sampling window, is used. Since CCN NoC is a crossbar augmented with a ring, we emulate it by using index-based core selector under RP(27,1,1).

- **Locality-Aware Last-Level Cache (LA-LLC)** [146] utilizes a locality-aware L2 that records the last sharer core. Upon receiving a read request from a core, the locality-aware L2 forwards the request to the last sharer in case of a hit, instead of serving the request.

Effect on Performance. Figure 3.13 shows the performance of our proposed schemes in terms of IPC and total reply bandwidth received by a core (in terms of L2 reply bandwidth and remote-core reply bandwidth), respectively. The results are normalized to the baseline architecture with no inter-core communication. We draw five main observations. First, IP(15,2,0.2) achieves 21% and 8% IPC improvement over the baseline and IP(27,1,1), respectively. The superiority of IP(15,2,0.2) over the baseline comes from unlocking the remote-core bandwidth, thus increasing the total available on-chip bandwidth. However, higher performance compared to IP(27,1,1) comes from searching fewer cores for the required data with higher confidence. Also, the possibility of sending two parallel probes

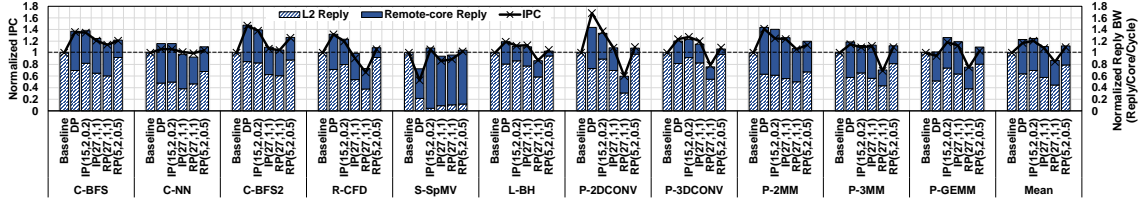


Figure 3.13: The effect of the proposed schemes on IPC and reply bandwidth.

helps in improving the performance as it cuts down the search latency. Second, DP yields better performance compared to IP(15,2,0.2) for almost all evaluated applications except S-SpMV and P-GEMM (also observed in Figure 3.2). Such counter-intuitive behavior for these two applications is due to the existence of only a few supplier cores for the majority of the requests (Section 3.3.3), leading to NoC hotspots near some cores under DP. Consequently, the remote-core bandwidth is reduced. In contrast, under IP(15,2,0.2), if a given target core is busy, the request is deflected to the next target core (Section 3.3.6) thereby alleviating hotspots. Moreover, DP is dependent on a single target core, thus it risks the possibility of not finding the data due to eviction and falls back to probing L2/memory. On the other hand, IP(15,2,0.2) searches more cores, so even if a target supplier core evicted the data, the probe moves to the next core in its supplier list.

Third, the performance of RP(27,1,1) is lower than the baseline. This is because of the misprediction overhead. The overhead of searching 27 cores for each misprediction causes a 15% drop in IPC. Therefore, searching less number of cores mitigates the misprediction overhead. Fourth, RP(5,2,0.5) performs better than IP(27,1,1), that utilizes perfect sharing predictor, because of its lower search overhead. Specifically, RP(5,2,0.5) searches only 5 cores compared to 27 cores in case of IP(27,1,1). Also, RP(5,2,0.5) divides the search process among two probes. As a result, even in case of failing to find the data, the smaller search space and the parallel search lessens the overhead. Fifth, the total reply bandwidth follows the same trend as IPC. This conforms to what we discussed in Section 3.2. Additionally, the reply bandwidth from the remote cores in RP(5,2,0.5) is less compared to the other schemes. This is because RP(5,2,0.5) searches 5 cores only,

thus perceives lower inter-core locality (refer to Figure 3.4a).

Figure 3.14a shows the precision and recall of RP(5,2,0.5).⁴ In general, we find precision and recall to be high for many applications, except a few ones. These applications do not have a few dominant *PC* values as previously discussed in Figure 3.10b. On average, RP(5,2,0.5) achieves 72% precision and 88% recall. Since the precision controls the misprediction volume, we investigate the sensitivity to different precision values by studying an imperfect IP. Figure 3.14b shows the effect on IPC using imperfect IP(5,2,0.5) and imperfect IP(15,2,0.2), respectively, under different precision values (100%, 95%, 90%, 80%, and 70%). These precision values are achieved by injecting non-shared requests into the NoC. A precision of $X\%$ under IP means that $(100 - X)\%$ of the non-shared requests are considered as shared. We observe that the drop in IPC in IP(15,2,0.2) increases with less precise predictors (up to 85% performance loss). This is because the unnecessary overhead per mispredicted request is high (searching 15 cores). However, in IP(5,2,0.5), the drop is less severe (up to 45%) due to lower misprediction overhead (searching 5 cores).

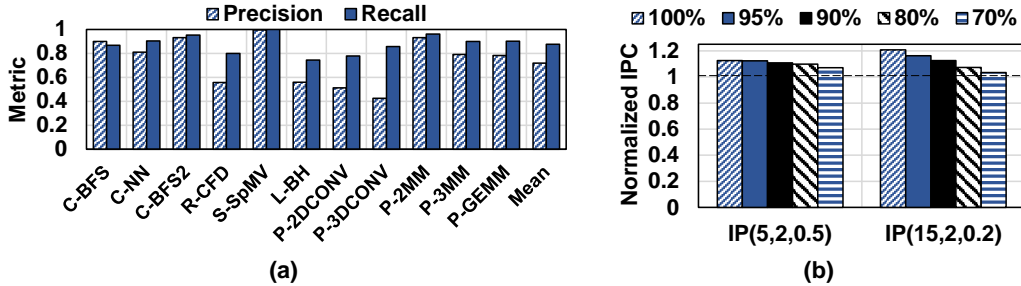


Figure 3.14: Illustrating (a) precision and recall for RP(5,2,0.5) and (b) effect of prediction precision on IP.

We can further bridge the gap between RP(5,2,0.5) and IP(15,2,0.2) if a software-based technique or a programmer input is utilized to provide sharing insight. For example, if a software-based mechanism provides the sharing PC information (instead of using the PC-based predictor), we can achieve performance improvement more than RP(5,2,0.5). Specifically, for C-BFS2 and S-SpMV, an IPC improvement of 37% and 5% is achieved

⁴Precision measures the percentage of the shared predictions that were truly shared. Recall measures the percentage of the truly shared cases the predictor identified.

respectively, compared to 38% and 6% in the case of IP(15,2,0.2). To conclude, any increase in the prediction precision helps improving the performance of RP(5,2,0.5).

Finally, we evaluate RP(5,2,0.5) against LA-LLC. On average, RP(5,2,0.5) achieves 10% IPC improvement compared to 2% from LA-LLC. LA-LLC uses the existence of the data in L2 as sharing indicator and forwards the read request to the last sharer core instead of serving at L2. However, the data may be evicted by the time the request reaches the last sharer. This degrades LA-LLC overall prediction precision to an average of 60% and as low as 40% for applications like P-3MM, and P-GEMM. Also, considering only the last sharer, vs. five cores in RP(5,2,0.5), in the search space decreases the chances of finding the data.

In summary, using IP(15,2,0.2) allows for higher performance as it balances the trade-off between searching more cores vs. sending more probes. However, searching fewer cores as in RP(5,2,0.5) is favored if a low-overhead option is required to balance out any penalty due to mispredictions.

Effect on Link Utilization. Figure 3.15 shows the effect of IP(15,2,0.2) and RP(5,2,0.5) on the request and reply NoC link utilization. We choose three applications as representatives and compare both mechanisms to baseline and DP. Two observations are in order. First, in the request NoC, both IP(15,2,0.2) and RP(5,2,0.5) have higher link utilization compared to baseline and DP. This is a result of utilizing the links to communicate among cores for searching and retrieving the required data. IP(15,2,0.2) achieves better link utilization in a couple of applications (e.g., C-BFS) due to searching more cores. Second, in the reply NoC, IP(15,2,0.2) and RP(5,2,0.5) have similar behavior in the highly utilized links, however, the lowest utilization in IP(15,2,0.2) is higher than in RP(5,2,0.5). This is because IP(15,2,0.2) searches more cores compared to RP(5,2,0.5), thus enabling more sources to deliver replies. Subsequently, more links are used to retrieve data from the target cores.

Performance Impact on Applications with low Inter-core Locality. Some applications have either low inter-core locality or none. Figure 3.16 shows the performance of

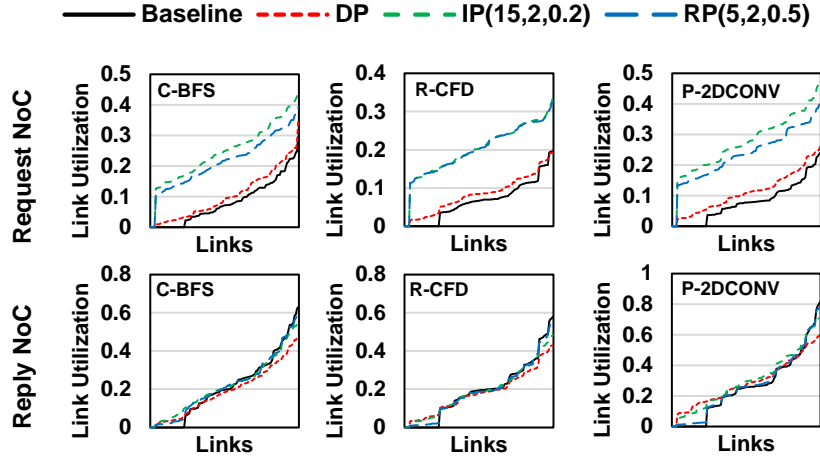


Figure 3.15: The effect of the proposed schemes on the request and reply NoC link utilization.

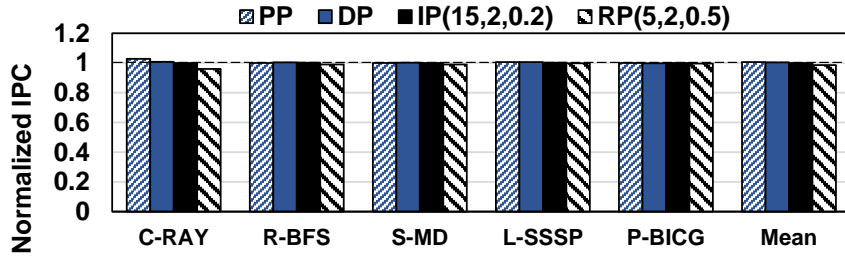


Figure 3.16: The effect of the proposed schemes on applications with low inter-core locality. Results are normalized to the baseline with no inter-core communication.

five applications, from different benchmarks suites, with $< 10\%$ inter-core locality under PP, DP, IP(15,2,0.2), and RP(5,2,0.5). Two observations are in order. First, the performance gain from PP, DP, or IP(15,2,0.2) is less than 1%. This is due to the reduced scope of inter-core locality. Second, our RP(5,2,0.5) does not affect the evaluated applications negatively. On average, IPC under RP(5,2,0.5) drops 1% for these applications. This is because the small scope of sharing drives the PC-based sharing predictor towards the most restrictive *Strong Non-shared* state which assumes less shared requests over a larger window of requests. This shows that our predictor can handle the absence of inter-core locality without degrading performance.

3.5.1 Sensitivity Studies

Effect of CTA Scheduling. We use the widely-used round-robin CTA scheduler to achieve better load balancing of CTAs across cores [47]. However, our proposal should still be effective under different CTA scheduling mechanisms. For example, a CTA scheduler that assigns nearby CTAs on the same core [46] still leaves a significant room to exploit inter-core locality. Figure 3.17a shows the portion of remote hit requests that have CTA distance ≤ 8 (with the nearest supplier core) and above. We observe that for nine out of eleven applications, the portion with CTA distance > 8 is more than 50% of the requests with at least one remote hit. We conclude that even with a CTA scheduler that assigns up to eight consecutive CTAs on the same core, we still have a large scope for inter-core communication to unlock the remote-core bandwidth.

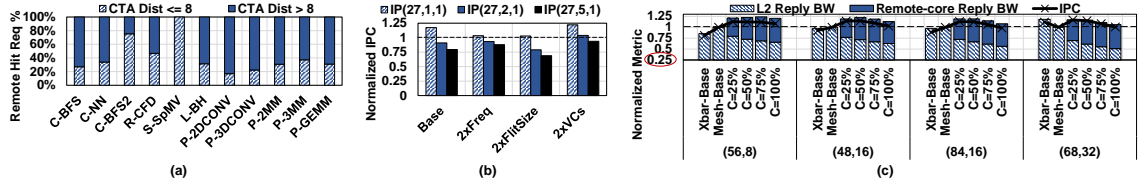


Figure 3.17: Sensitivity studies on (a) CTA scheduling, (b) NoC resources, and (c) NoC size.

Effect of NoC Resources. Figure 3.17b shows the sensitivity when increasing the NoC resources. We consider three configurations; double the NoC frequency, double the flit size, and double the virtual channels. We also show the results of the baseline NoC used so far (Section 3.3.1), denoted as *Base*. $IP(27,S,1)$ is evaluated under each of them and normalized to the corresponding configuration baseline. First, we observe that our schemes are still beneficial even with double the NoC resources. Second, increasing the number of probes (S) under 27 cores is still not helpful. Third, our schemes benefit the most under double the VCs. This is because searching cores and pushing more probes cause contention at the VC allocator and SW allocator. Thus, doubling the VCs may mitigate the VC allocation contention but at the cost of extra hardware.

Effect of NoC Size. We study the scalability of our schemes using 8×8 mesh and

10×10 mesh under two different configurations. Figure 3.17c shows the IPC and reply bandwidth (both normalized to the configuration mesh baseline) under $IP(C\%,1,1)$, where $C\%$ represents the percentage of cores to be searched. The used notation in the figure is (number of cores, number of L2/memory partitions). We observe that the IPC follows a similar trend to what we observed using the baseline 6×6 mesh. Specifically, searching 25% or 50% of the cores leads to higher performance in terms of both IPC and reply bandwidth.

Effect of Additional Memory Partitions. Figure 3.17c shows the effect of increasing the number of memory partitions (this increases the total L2 capacity, L2 bandwidth, and memory bandwidth) in the system. For an 8×8 mesh, we study systems with 8 and 16 memory partitions. For a 10×10 mesh, we study systems with 16 and 32 memory partitions. We observe that even with more memory partitions, our proposal enhances IPC due to efficiently unlocking the remote-core bandwidth.

Effect of Core to Memory Partition Ratio. Figure 3.17c studies varying the ratio of core to memory partition count. We observe that our schemes can boost IPC in all systems. Even in a large (68,32) system, $IP(C=25\%,1,1)$ achieves 17% IPC improvement over the baseline 10×10 mesh.

Comparison against a Crossbar-based Baseline. In Figure 3.17c, we observe that our schemes perform better than a crossbar-based baseline in terms of both IPC and reply bandwidth under (56,8), (48,16), and (84,16) systems. Under a large (68,32) system, a crossbar-based baseline performs close to, but still not as good as, our schemes. Note that for such large systems, the complexity of the crossbar is high. Also, the performance difference between the mesh-based baseline and the crossbar-based baseline is in line with a simple bisection bandwidth analysis for both systems.⁵

We conclude that our design is robust and can perform well across a wide range of hardware mechanisms and system configurations, such as CTA scheduling policies,

⁵For the systems we consider in this work, the ratio of crossbar bisection bandwidth to 2D mesh bisection bandwidth is equal to the ratio of the number of memory partitions to twice the mesh dimension.

L2/memory bandwidth, and core to memory partition ratio. It also outperforms the crossbar-based baseline.

3.6 Related Work

In this section, we briefly discuss works that are the most relevant to this study.

Intra-core Locality in GPUs. There is a large body of work that focuses on exploiting the locality that exists within a GPU core [105, 106, 47, 51, 114, 73, 46, 44, 53, 96, 52, 81, 67, 78, 140]. In this work, we specifically focus on the locality that exists across cores. Multiple prior CTA schedulers [65, 8, 122, 136, 71] used different heuristics to exploit the locality across CTAs. However, as shown by prior works [131, 8, 148], there is no single ideal CTA scheduling policy that benefits all applications. This is because inter-CTA locality, data access pattern, and execution time of CTAs are hard to know at compile time, which increases the complexity of the CTA scheduling problem. Hence, we choose the round-robin CTA scheduler as it is the most commonly used. Our analysis shows that the data sharing across L1 caches is pervasive and hence our solutions are effective.

Inter-core Locality in GPUs. Prior works proposed mechanisms to exploit inter-core locality in GPUs by allowing communication between multiple L1s by connecting the cores via a ring NoC [28] or using the L2 cache to forward the read request to a supplier L1 [146]. Other works proposed coherence-like mechanisms [123] to enable communication across L1 caches. Inter-core locality information has also been used to propose a packet coalescing mechanism to reduce NoC pressure [55]. Although these works either identify inter-core locality, propose architectures to enable inter-core communication, or utilize coherence-like mechanisms, they do not provide a way to (1) probe multiple L1 caches in parallel, and (2) identify which L1 caches to probe for high probe success rate. Our schemes allow the inter-core communication to be low-latency due to parallel probes, and low bandwidth-demanding due to the reduced number of useless probes sent. Finally, previous works studied coherence communication predictors based on address [79, 64],

instruction [49], or both [77, 50]. These works focused on tracking coherence events at the directories. Our work uses an effective PC-based predictor to filter the read misses that have less probability of sharing across the GPU cores.

3.7 Chapter Summary

Traditionally, GPUs have been depending on the bandwidth from local/shared caches and memory to achieve high performance. Going forward, other sources of bandwidth need to be explored and leveraged given that the issue of bandwidth is going to be even more critical in large-scale GPU-based systems. Our detailed analysis in this work shows that remote-core bandwidth can significantly improve the GPU performance within a single GPU node. However, there are several challenges in unlocking this remote-core bandwidth, which this work systematically addresses. First, we leverage the bi-modal distribution of inter-core locality across PCs to determine which data is expected to be shared across cores. Second, we dynamically generate an inter-core locality map that guides the probing mechanism to determine which cores to probe for increasing the probability of finding the shared data. Finally, we develop a novel two-level probing technique to get the data as soon as possible without saturating the interconnect. We conclude that our efficient inter-core communication provides a significant improvement in performance and on-chip bandwidth at a modest hardware cost.

Chapter 4

Analyzing and Leveraging Shared L1 Caches in GPUs

Graphics Processing Units (GPUs) concurrently execute thousands of threads, which makes them effective for achieving high throughput for a wide range of applications. However, the memory wall often limits peak throughput. GPUs use caches to address this limitation, and hence several prior works have focused on improving cache hit rates, which in turn can improve throughput for memory-intensive applications. However, almost all of the prior works assume a conventional cache hierarchy where each GPU core has a *private* local L1 cache and all cores share the L2 cache. Our analysis shows that this canonical organization does not allow optimal utilization of caches because the private nature of L1 caches allows multiple copies of the same cache line to get replicated across cores.

We introduce a new *shared* L1 cache organization, where all cores collectively cache a single copy of the data at only one location (core), leading to zero data replication. We achieve this by allowing each core to cache only a non-overlapping slice of the entire address range. Such a design is useful for significantly improving the collective L1 hit rates but incurs latency overheads from additional communications when a core requests data not allowed to be present in its own cache. While many workloads can tolerate this

additional latency, several workloads show performance sensitivities. Therefore, we develop lightweight communication optimization techniques and a run-time mechanism that considers the latency-tolerance characteristics of applications to decide which applications should execute in private versus shared L1 cache organization and reconfigures the caches accordingly. In effect, we achieve significant performance and energy efficiency improvements, at a modest hardware cost, for applications that prefer the shared organization, with little to no impact on other applications.

4.1 Introduction

Graphics Processing Units (GPUs) have emerged as very effective general-purpose accelerators for a wide range of applications. They have been successful because they provide very high throughput at a competitive power budget. High-bandwidth memories provide the foundation for supporting the fine-grain multithreading that GPUs rely upon for achieving high throughput. However, the well-known memory wall [138] is often the performance-limiting factor for GPUs. Traditionally, a popular approach to address the memory wall problem has been to employ on-chip memories such as caches. In CPUs, caches have been very effective in cutting down memory latencies. In GPUs, however, latency is not often the first-order challenge for many applications because of the high level of multithreading. Still, GPUs are equipped with both software-managed (scratchpad) and hardware-managed on-chip memories (caches) to reduce traffic to the lower levels of the memory hierarchy. An increase in on-chip memory hit rate can lead to a proportional decrease in memory traffic, translating into performance improvements for memory-intensive programs [82, 134]. Therefore, researchers in the past have invested significant efforts in improving cache performance via hardware and software methods [105, 47, 46, 42, 51, 58, 148].

GPUs typically employ a two-level cache hierarchy, where each core is associated with a private local L1 cache, and all cores in the GPU share a banked L2 cache. An interconnect connects all cores to the L2 caches and memory partitions. The L1 caches are responsible

for reducing traffic to the interconnect and L2 cache, while the L2 cache helps to reduce memory traffic. This work challenges such a conventional cache organization and reveals inefficiencies in the existing cache hierarchy in the context of GPUs. In particular, we focus on addressing the inefficiencies associated with GPUs' *private* local L1 caches. Specifically, because of the private nature of the L1 caches, the same cache lines can be requested by different cores, leading to high inter-core locality [71, 59, 72, 28]. This data (cache line) replication reduces the effective aggregate capacity of the L1 caches across all cores, leading to their lower bandwidth utilization as we will show in Section 4.2.

To address these challenges, we propose and evaluate *shared* local L1 caches in GPUs. The key idea is to ensure only one copy of data exists across L1 caches, thereby eliminating data replication and making better use of the finite cache capacity. We propose to realize the shared L1 caches by making minimal changes to the existing L1 cache controller and address mapping policies, with *no* changes to the L1 caches. Normally, each core can cache any data from the entire address range. Instead, our shared L1 cache design restricts each core to cache only a unique slice of the address range. Consequently, each core caches data from non-overlapping address ranges, which eliminates data replication across local caches.

Although such a design is attractive for GPUs, it requires inter-core communication if one core requests data that is not mapped to its allocated address range. In such situations, additional latency will be incurred to fetch the data from the L1 cache of a remote core. Fortunately, thanks to the latency-tolerance of many GPGPU applications, an increase in latency often has a negligible impact on performance. However, not all applications a) can tolerate long memory latencies, b) exhibit data replication, or c) are sensitive to cache capacity (i.e., their working sets fit in L1 cache or they stream with little-to-no locality). Consequently, shared local caches can have negative or no effect on such applications' performance. To address these concerns, we develop lightweight mechanisms to a) reduce the inter-core communication overhead and b) identify applications that prefer the private L1 organization and hence execute them accordingly.

Contributions: This work contributes the following:

- We propose shared L1 caches in GPUs. To the best of our knowledge, this is the first work that performs a thorough characterization of shared L1 caches in GPUs and shows that they can significantly improve the collective L1 hit rates and reduce the bandwidth pressure to the lower levels of the memory hierarchy.

- We develop GPU-specific optimizations to reduce inter-core communication overheads. These optimizations are vital for maximizing the benefits of the shared L1 cache organization.

- We develop a GPU-specific lightweight dynamic scheme that classifies application phases and reconfigures the L1 cache organization (shared or private) based on the phase behavior.

- We extensively evaluate our proposal across 28 GPGPU applications. Our dynamic scheme boosts performance by 22% (up to 52%) and energy efficiency by 49% for the applications that exhibit high data replication and cache sensitivity without degrading the performance of the other applications. This is achieved at a modest area overhead of $0.09 \text{ mm}^2/\text{core}$.

- We make a case to employ our dynamic scheme for deep-learning applications to boost their performance by $2.3\times$.

4.2 Motivation and Analysis

In this section, we first quantify the data replication problem associated with private L1s in GPUs (as described in Section 4.1) and then make a case for shared L1s to address this inefficiency.

4.2.1 Analysis of Wasted L1 Cache Space

Figure 4.1 shows the line replication ratio under the baseline private L1 organization for the evaluated applications (methodology detailed in Section 4.5). The line replication

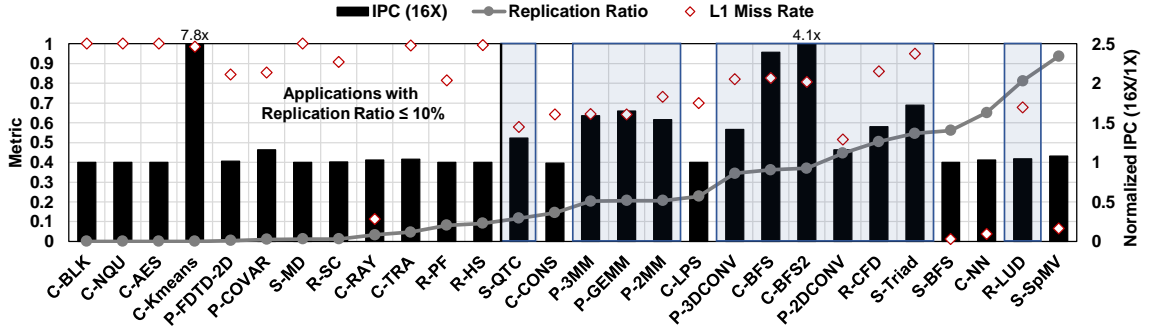


Figure 4.1: Performance of the evaluated applications in terms of L1 miss rate, line replication ratio, and IPC improvement under $16\times$ the L1 cache size (normalized to baseline). The left-hand y-axis represents cache line replication ratio and raw L1 miss rate.

ratio is defined as the ratio of L1 misses that can be found in other L1 caches to total L1 misses. We observe that the replication ratio varies across the applications. Specifically, some applications have no replication (e.g., C-BLK) or low replication (e.g., C-RAY), while others have high replication (e.g., C-BFS).

Identifying Target Applications. The waste due to data replication may not affect all applications. Only the applications that are sensitive to larger cache space are *expected* to benefit if the wasted cache space is reduced/eliminated. Therefore, we study their performance under a $16\times$ larger L1 cache in Figure 4.1. We observe that 13 applications are both capacity-sensitive and possess high data replication. To identify the subset of the capacity-sensitive applications that are sensitive to data replication, we study their L1 miss rates. Applications with low L1 miss rates (e.g., C-NN and S-SpMV) may not suffer under private L1 caches because the majority of their requests can be satisfied locally. These applications tend to have working sets smaller than the baseline L1 cache capacity. In general, we consider an application to be sensitive to data replication if it 1) has a replication ratio of $>10\%$, 2) has an L1 miss rate of $>50\%$, and 3) observes a speedup of $>5\%$ with $16\times$ capacity.¹ Based on these criteria, we observe that 11 applications are sensitive to data replication (marked by the blue boxes in Figure 4.1). These are our target applications.

¹This criteria is empirical and is not used by our proposed scheme in Section 4.4.

4.2.2 A Case for Shared L1 Caches

One way to eliminate data replication is to enable a shared cache organization across the local L1 caches. Under a private L1 organization, each core can cache any line. For example, given four different address ranges represented by different shades in Figure 4.2a, a private L1 cache can store any cache line from all four address ranges. However, under a shared L1 organization, the entire address range is interleaved across all cores and such mapping is fixed. In other words, each core caches data from a non-overlapping address range. For example, as shown in Figure 4.2b, the address range represented by white can be cached by only L1-0, and the address range represented by black can be cached by only L1-3. Because an exclusive slice of the address range maps to a single L1, the shared L1 organization ensures no cache line replication across L1s. However, to fully unlock the potential of the shared L1 organization, the cores need to communicate to fetch the data that do not belong to their assigned address ranges.

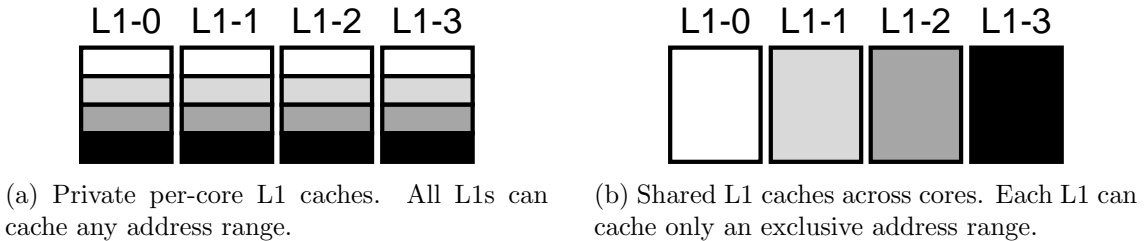


Figure 4.2: Private and shared cache organizations.

Sources of Benefits. To understand the scope of potential performance benefits of the shared L1 organization, we set up a hypothetical design where all cores can communicate with each other with zero cycle overhead and share their L1 caches ensuring no data replication. Figure 4.3 and Figure 4.4 show performance in terms of IPC and L1 miss rate for the identified target applications executing on this hypothetical system, normalized to the private L1 baseline. We observe, in Figure 4.3, that such a hypothetical configuration improves performance by between 14% and 83% across these applications, and 39% on average. The main reason for such a performance boost is the significant 79% reduction

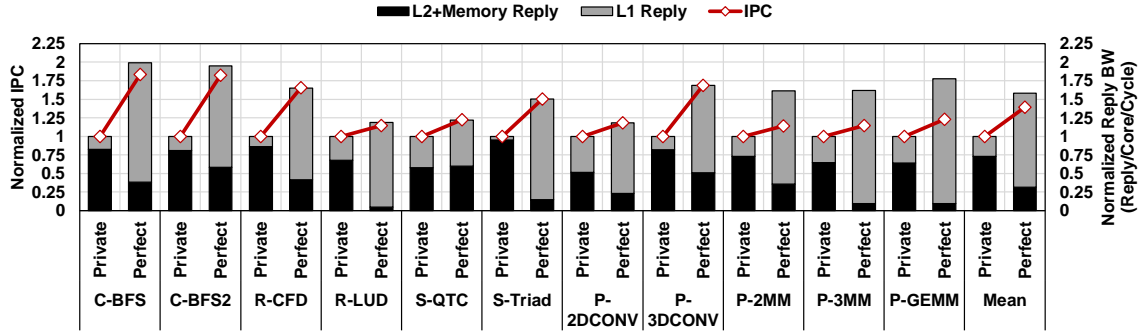


Figure 4.3: IPC and reply bandwidth of a hypothetical cache design that eliminates data replication across local L1 caches. Results are normalized to the private L1 baseline. Section 4.5 has the details on the experimental methodology.

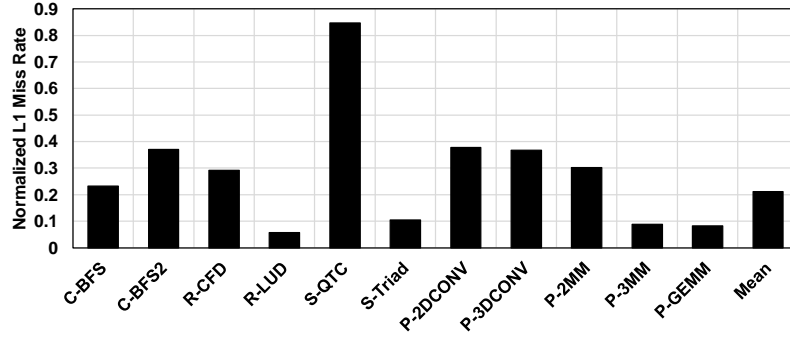


Figure 4.4: L1 miss rate of a hypothetical cache design that eliminates data replication across local L1 caches. Results are normalized to the private L1 baseline.

in the collective L1 miss rate (shown in Figure 4.4) that results in better L1 bandwidth utilization (i.e., total collective *useful* bandwidth received from the L1 hits is higher than the baseline).² Consequently, the L2 and memory bandwidth consumption is reduced, thereby making the L1s more effective at addressing the memory wall challenge.

Overall, we conclude that the shared L1 organization eliminates wasted L1 cache capacity and thus can lead to a significant performance improvement for the target applications. Therefore, we refer to them as *shared-friendly* applications. Next, we propose a shared L1 cache design that aims to achieve the performance of this hypothetical cache design for the shared-friendly applications.

²L1 and L2+Memory reply bandwidth represent the number of replies received from L1 and L2+Memory, respectively, over the total execution time.

4.3 Shared L1 Caches: Design, Analysis, and Optimizations

In this section, we describe our design that enables both private and shared L1 organizations, and demonstrate the potential performance of a realistic shared L1 organization.

4.3.1 Terminology and Address Mapping

Under a shared L1 organization, we define two terms that we use in this paper: *requester* and *home* cores. A requester is a core that requests a given cache line and the home is the core that can cache that line. For example, in Figure 4.2b, the home core of a line that falls in the black address range is core $L1 - 3$. If core $L1 - 3$ requests that line, then the core is both the requester and the home for that line. Additionally, a typical memory access under a shared cache organization can be either *local* or *remote*. An access is considered local if the requester core is also the home core. Otherwise, an access is remote. For example, in Figure 4.2b, if core $L1 - 0$ requests a cache line from the black address range, then it will send a remote request to the home core $L1 - 3$.

Selecting the Home Core. To select the home core for a given cache line, we use *core bits*. These core bits are selected from the physical address of a request. The process of selecting these bits is analogous to selecting the DRAM bank bits based on the physical address. In the private L1 cache organization, there are no core bits because the requester is always the home. In a system with N local L1 caches (each attached to one core) that are organized in a shared fashion, we use the least significant $\lceil \log_2(N) \rceil$ bits of the tag as core bits to select the home core for a given cache line. Because the core count or the cache being organized as private or shared does not affect the number of tag bits according to this mapping policy, our system always uses 20 bits as tag.

4.3.2 Shared L1 Caches Design

Figure 4.5 shows the communication flow in a simple design that enables the L1 caches to be organized as either private or shared. Each core is connected to an L1 cache, which has

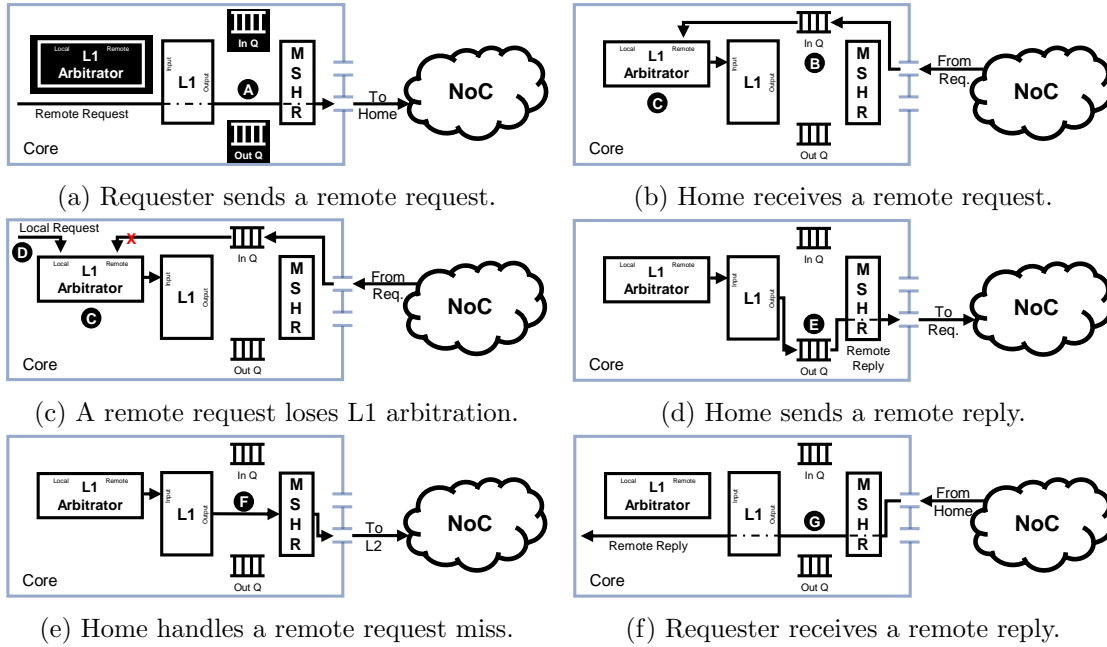


Figure 4.5: Request/Reply flow in a shared L1 organization. The L1 Arbitrator and the In/Out queues, shown in black in (a), are newly added to support our proposal. Dashed lines represent L1/MSHR bypassing.

associated MSHR to track pending L1 misses. The MSHRs are connected to the network-on-chip (NoC) that routes L1 misses to the L2. In the baseline private L1 organization, each core sends requests to its local L1, and the misses go through its local MSHR to access the L2 via the NoC.

Handling Read Requests. With a shared L1 organization, a remote read request skips the L1 cache of the requester core because the data cannot be there. It then also skips the local MSHR **A** and goes through the NoC to reach the home core. The home core queues the received remote request **B** and consults its local L1 cache arbitrator **C**, which prioritizes the local cache requests over remote requests. If there are no local requests, the remote request accesses the L1 cache of the home core. Otherwise, the local request is processed **D** and the remote request remains queued. If the request hits in the home L1 cache, then the L1 queues the read reply **E** for injection into the NoC back to the requester. If the request misses in the home L1 cache, then the home core sends the

request to the L2 cache through the NoC **F**.³ Once a read reply is received from the L2 via the NoC, the home core installs the reply in its local L1 and concurrently queues the reply **E** to be injected to the requester through the NoC. Finally, the requester core receives and processes the remote read reply without caching it locally **G**.

Handling Write Requests. With a shared L1 organization, a remote write request follows the same flow as a remote read request. However, a remote write request always skips the MSHR of both the requester and the home. Also, we use write-through and no-write-allocate policies in the L1 caches (Section 4.5). Therefore, on a write hit, a given write request modifies the cache line in the home core. The modified cache line is forwarded to the L2 cache through the NoC. However, on a write miss, no cache line is allocated at the home core and the updated data is delivered to the L2 cache. Once a write ACK is received from the L2, the home core forwards the write ACK to the requester core via the NoC.

Handling Coherence. With a shared L1 organization, only a single copy of a cache line may exist across L1s. Therefore, there may not be a need for coherence mechanisms within a single GPU.

Handling Non-L1 Requests All non-L1 (instruction, texture, and constant cache) misses from the GPU core are not affected by the shared cache organization. Non-L1 misses are simply forwarded to the L2 via the NoC as in the private L1 baseline.

Handling Atomic Operations. In the baseline, atomic operations skip the L1 cache and are handled at L2/MC [10]. Similarly, in our design, atomic operations skip the requester and home L1 caches and are handled at the unaltered L2/MC.

Communication Fabric. We evaluate shared L1 caches with a mesh interconnect [11, 53, 142, 97] in Section 4.3.3 and present a case study of a crossbar-based system in Section 4.6.3. Other interconnect topologies that allow inter-core communication can be used to unlock the full potential of shared L1 caches, but we leave the study of such

³A single MSHR entry is allocated for a unique cache line address at the home's L1 to allow coalescing of misses originating from both local and remote read requests.

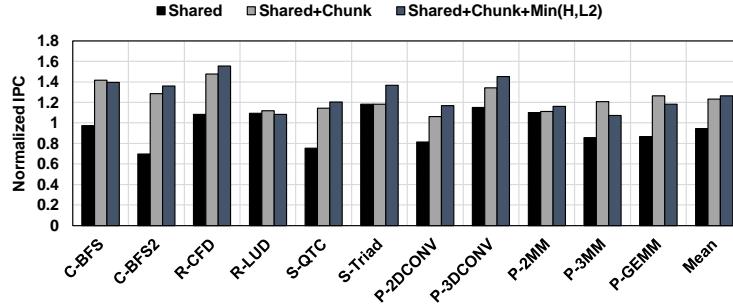
topologies for future work.

4.3.3 Performance Analysis and Optimizations

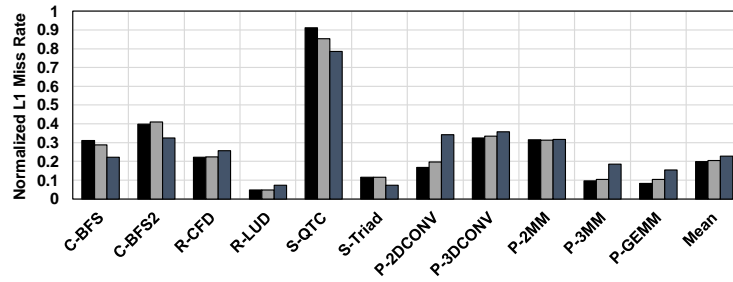
We analyze the impact of shared L1 cache design on the shared-friendly applications in terms of performance, L1 miss rate, and the reply network latency as shown in Figure 4.6. We observe that although the shared design (denoted as *Shared*) helps in significantly reducing the L1 miss rate by 80% (as expected per our discussion in Section 4.2.2), it does not translate into performance improvement over the private L1 baseline. In fact, we observe a performance degradation of 5%. This is because of the overhead incurred (average packet latency of the reply traffic increases by $2.2\times$) due to the additional communication. Therefore, it is essential to analyze this overhead and propose optimizations to alleviate it.

Optimization I: Reducing Wasted NoC Bandwidth. Because the requester does not install data for remote requests in its own L1, fetching the requested data at a full cache line granularity from the home core wastes NoC bandwidth if only a portion of the line is actually requested by the requester. Figure 4.7 shows how much data within a line is used by the requester cores for shared-friendly applications. “Access=N” denotes that N bytes out of 128, which is the cache line size, are used by the requester. We observe that many applications do not need the entire cache line data and in fact need only a quarter of it most of the time. We apply this known observation [104] in a different context for reducing interconnect traffic between cores. Based on this observation, we design the system such that the data reply from the home to the requester only carries the data requested by the requester, not the entire line. The key idea is to reduce unnecessary data movement and to also avoid wasting precious NoC bandwidth. With the help of this optimization (denoted as *Shared+Chunk*), we observe a significant speedup of 23% for shared-friendly applications as shown in Figure 4.6a.

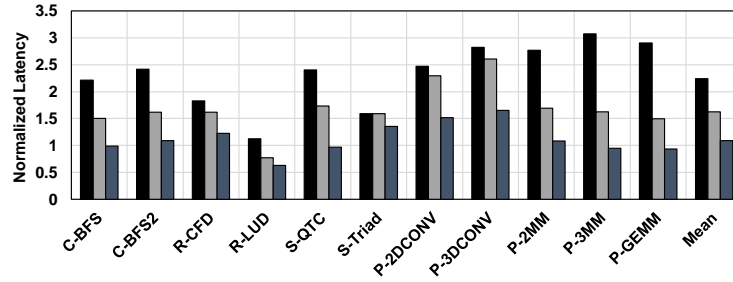
Optimization II: Better Distribution of Requests. The next optimization ($Min(H,L2)$) balances the interconnect traffic by selectively routing the L1 requests to



(a) IPC



(b) L1 miss rate



(c) Average latency on the reply network.

Figure 4.6: Performance of a realistic shared L1 organization. Results are normalized to the private L1 baseline.

either the home L1 or to L2, whichever is fewer hops away. The key idea is to better utilize both the home cores' bandwidth and the L2 bandwidth and cut down latency by going to the nearest source of data. In Figure 4.6, *Shared+Chunk+Min(H,L2)* shows the effect of applying *Min(H,L2)* on top of *Shared+Chunk*. In this experiment, we apply Optimization I (*chunking*) on the traffic from either the home core or L2 to the requester core. We make several key observations. First, with *Shared+Chunk+Min(H,L2)*, we improve the performance benefits to 26%. This is because of the better distribution of interconnect traffic and reduced latency. In *Shared+Chunk*, all requests go to home cores, which has

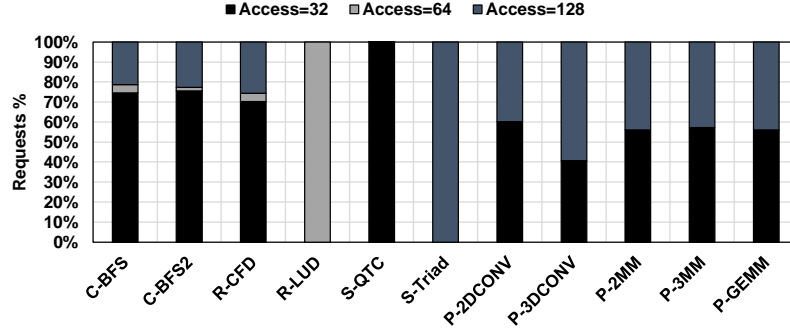


Figure 4.7: Fraction of useful bytes within a cache line.

the potential to create network hotspots and limit the achieved bandwidth from the home cores. With Shared+Chunk+Min(H,L2), there is a better balance between requesters obtaining their data from home cores and L2. Second, Shared+Chunk+Min(H,L2) does not provide significant L1 hit rate benefits, compared to Shared+Chunk. Its performance benefit is mainly because of a more uniform distribution of traffic on the chip, not due to reduced cache contention at the home caches. Finally, Shared+Chunk+Min(H,L2) reduces the latency overhead to 9%, mainly because of lower hop counts and more uniform traffic distribution. We conclude that, for shared-friendly applications, our optimizations can reduce the wasted bandwidth, provide a good balance between miss rate reduction and network latency, and show promising performance improvements. For the rest of the paper, we will refer to Shared+Chunk+Min(H,L2) as *Shared++*.

Evaluating Non-shared-friendly Applications. So far, we have proposed an optimized shared L1 organization and validated its usefulness on the shared-friendly applications. For completeness, we evaluate Shared++ further on other applications (17) that are not classified as shared-friendly (denoted as *non-shared-friendly* applications). Figure 4.8 shows the performance of these applications normalized to the private L1 baseline. Three observations are in order. First, most of these applications perform as well as the private L1 baseline and are hence classified as *insensitive*. These applications are likely to have a high tolerance to the latency overhead induced by the shared L1 organization. Second, two of these applications (C-Kmeans and P-COVAR) perform better than the private L1 or-

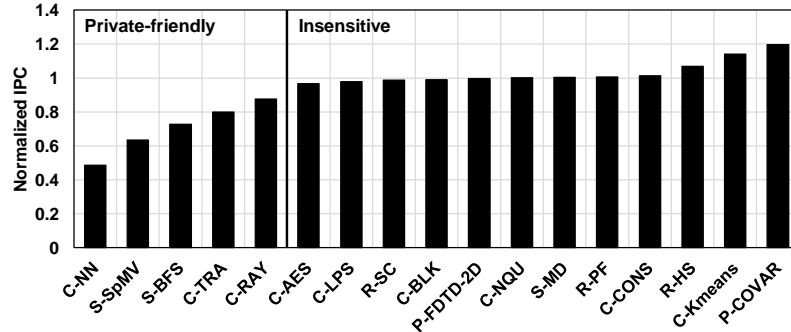


Figure 4.8: Non-shared-friendly applications under Shared++. Results are normalized to the private L1 baseline.

ganization. C-Kmeans achieves a 14% performance improvement because of the Min(H,L2) optimization. C-Kmeans has high sensitivity to cache size and no replication across cores. Thus, by bypassing the home core and directly going to L2, we effectively increase the cache capacity (increase the L1 hit rate). As for P-COVAR, its 20% improvement is because of the work imbalance between the cores in some kernels under the private L1 baseline. Specifically, some kernels do not have enough CTAs for all the cores, which leaves the L1 caches of some cores not utilized in the baseline. However, with Shared++, all the L1 caches serve the requests based on the required address range. Finally, five applications suffer a drop in performance under the proposed shared L1 organization (minimum = 12%, maximum = 51%). We observe that these applications either have high L1 cache locality leading to low L1 miss rates (< 10%) or low latency tolerance. To make a strong case for the shared L1 organization, we need a mechanism that identifies such *private-friendly* applications and executes them in a private L1 organization.

4.4 A Dynamic Mechanism for Handling Private-friendly Applications

In this section, we present a per-core lightweight dynamic scheme that *locally* classifies an application at runtime as shared-friendly or private-friendly and executes the application

on a shared or private L1 organization accordingly. Our dynamic scheme utilizes a two-step process: a sampling phase followed by an execution phase. During the sampling phase of a core, it simultaneously collects runtime metrics for both shared and private organizations. Once the sampling phase of a core ends, it evaluates the locally collected information and chooses the desired L1 organization during the next execution phase. After concluding an execution phase, a new sampling phase starts. By repeating this two-step process, our scheme can adapt to the changing behavior of the application.

4.4.1 Sampling Methodology

In this section, we discuss the details of the sampling mechanism and the per-core collected information as shown in Figure 4.9.

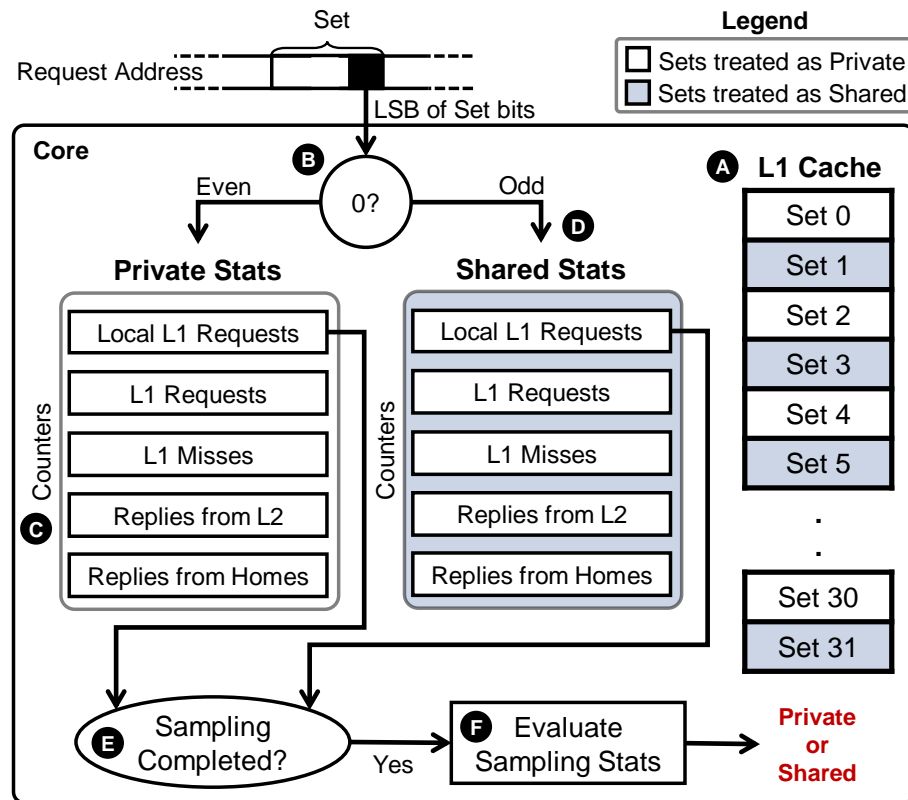


Figure 4.9: Sampling phase of the dynamic scheme.

Concurrent Evaluation of Private and Shared L1 Organization. Our scheme

concurrently evaluates both a shared and a private L1 organization using the local L1 cache during the sampling phase. We accomplish such simultaneous evaluation by treating half of the L1 cache sets as shared and the other half as private. We assign the even sets and the odd sets to be treated as private and shared, respectively **A**. We interleave the set indexing between private and shared at a fine granularity to decrease the bias of requests focusing on a subset of the cache sets. Note that this approach is not a dynamic cache partitioning scheme, thus we do not have the associated overheads [109]. We do not change the indexing of the cache as the set bits are the same. We use the least significant bit (LSB) of the set bits to determine if the required set is even (to be treated as private) or odd (to be treated as shared) **B**.

Sampling Phase. During the sampling phase, we use counters to gather information that is crucial for classifying the running application **C**. For example, we count the number of accesses and misses to the local L1 cache to estimate the L1 miss rate at the end of the sampling phase. Because we evaluate both shared and private cache organizations concurrently, we use two groups of counters for each option, and only the corresponding counters are updated based on the LSB of the set bits. For example, if a core receives a read reply from L2 to install in an odd set, then the replies from L2 counter for the sets that are treated as shared is incremented **D**. The sampling phase continues until both the shared and private groups each process at least R_S local L1 accesses ($R_S = 512$ requests) **E**, where a local L1 access occurs when a core generates a request that is destined to its local L1 cache (i.e., *requester = home*). This makes the time interval for the sampling phase variable. Also, this ensures that each group observes enough requests to have a fair evaluation between the two options.

Execution Phase. Once the sampling phase ends, the counters from each group are used to evaluate which cache organization to use **F**. The evaluation is based on the metrics discussed in Section 4.4.2. After evaluation, the execution phase starts under the desired L1 organization. The next sampling phase starts after processing R_{Ex} local L1 accesses

($R_{EX} = 16384$ requests).⁴

Selecting the Home Core. Due to the self-paced nature of our dynamic scheme, a given core may be in either sampling or execution phase. Additionally, a core locally chooses the preferred L1 organization. Nevertheless, as discussed in Section 4.3.1, a core under a shared L1 organization (during sampling or execution) still uses the core bits to determine the home core, even if the home core is under private L1 organization or in a sampling phase.

Handling Coherence. The coherence protocol utilized in the private L1 baseline is used in our dynamic mechanism. Specifically, both the private L1 baseline and our dynamic scheme employ flushing-based software coherence [92, 90, 5, 103, 121, 140, 116].⁵ This is ensured by the usage of 1) a write-through L1 cache that is invalidated and flushed at every kernel boundary or at synchronization points, and 2) a shared L2 cache that is inherently coherent. Such system-wide flushing of the L1 caches does not differentiate between a core that is under execution phase (private or shared) or sampling phase. In other words, all L1 caches in the system will be invalidated and flushed indiscriminately at kernel boundary or synchronization points to ensure coherence.

Handling Private-to-Shared Transition. In case shared L1 organization is desired for the execution phase, then some leftover cache lines may exist in the cache. A leftover line is a cache line that was cached during sampling in the sets treated as private but does not belong to the assigned address range of the core. However, if a leftover line is requested, then the core will skip its local L1 cache (as *requester* \neq *home*) and forward the request to the home core. Thus, these leftovers lines are not utilized by the requester core during the execution phase. Additionally, a request destined to a cache set storing a leftover line will always lead to a tag mismatch with the leftover line as the core bits are different. We employ a lazy invalidation scheme instead of migrating the leftover lines or flushing the

⁴ R_S and R_{Ex} values are empirically chosen based on the insight to have longer execution phases to minimize any sampling overheads.

⁵If a hardware-based coherence protocol is used, the directory at L2 will correctly keep track of the list of sharer cores and the invalidations will only be sent to the sharers.

L1 cache because of its simplicity. However, the cache replacement policy may be updated to consider the leftover lines for victim-selection. These lines can be identified by using either the core bits or by setting an extra 1-bit per cache line during sampling. Such a policy should replace the leftover lines sooner leading to better cache utilization.

4.4.2 Sampled Metrics

In this section, we assess the effectiveness of two possible metrics that can be used in classifying an application to be either shared-friendly or private-friendly. A good metric should clearly distinguish between shared-friendly and private-friendly applications with minimum overhead in terms of the sampled information.

Metric I: Average Memory Access Time (AMAT) is a well-known metric used to analyze memory system performance in the CPU domain. AMAT is a good candidate for evaluation as it covers the cache capacity aspect (via the miss rate) and reports the average overall latency. For our scheme, AMAT is defined as:

$$AMAT = L1_{HitLatency} + \left(\frac{L}{L+R} \times L1_{LocalMissRate} \times L2_{AccessLatency}\right) + \left(\frac{R}{L+R} \times AMAT_{Home}\right) \quad (4.1)$$

$$AMAT_{Home} = Home_{AccessLatency} + (L1_{RemoteMissRate} \times L2_{AccessLatency}) \quad (4.2)$$

where L is the number of a core's own local L1 accesses, and R is the number of a core's own remote L1 accesses. $L/(L+R)$ represents a fraction of the given core's own requests that belong to its assigned address range. Similarly, $R/(L+R)$ represents a fraction of the core's own requests that do not belong to its assigned address range.

At the end of the sampling phase, we evaluate AMAT for both shared and private L1 organizations and choose the option with the lower AMAT. Figure 4.10a shows the effectiveness of AMAT to choose between shared and private L1 organization using four non-shared-friendly applications (one insensitive and three private-friendly) and four shared-friendly applications. We observe that for the non-shared-friendly applications, AMAT

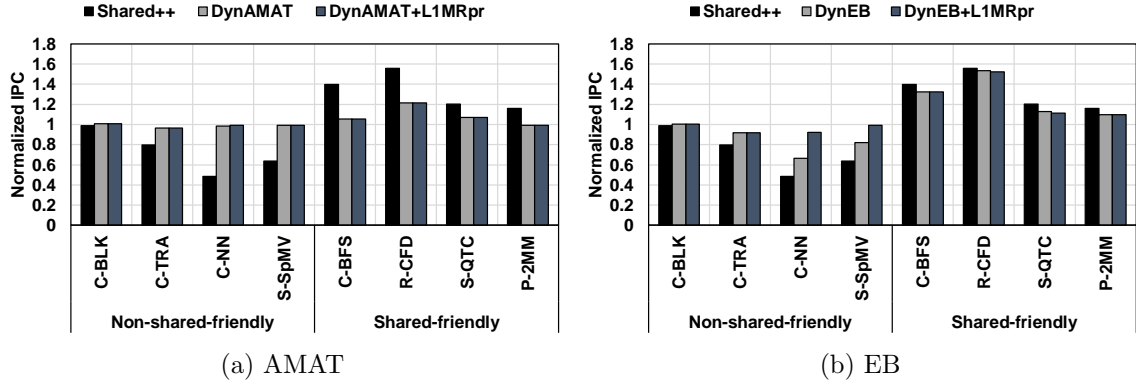


Figure 4.10: Effect of different metrics on the dynamic scheme.

(*DynAMAT*) performs as well as the baseline by clearly identifying the insensitive and private-friendly applications. It also performs better than Shared++ for C-TRA, C-NN, and S-SpMV. However, DynAMAT performs poorly with the shared-friendly applications, losing the performance benefits gained by using the shared L1 organization. This is because AMAT is oblivious to latency tolerance in GPUs. Thus, even with the latency overhead imposed by the shared L1 organization to access remote home cores, GPUs may be able to hide such an increase in latency due to their huge parallelism. This makes a case for using another metric.

Metric II: Effective Bandwidth (*EB*) is defined as the ratio of bandwidth to miss rate and is calculated based on the level of memory hierarchy under consideration. At a given core, EB is computed as BW/CMR , where $CMR = L1_{MissRate} \times L2_{MissRate}$. EB is a good candidate for the following reasons. First, Wang *et al.* [134] showed that $IPC \propto EB$. Thus by optimizing for a higher EB, we aim for a higher IPC as well. Second, EB is sensitive to the change in the L1 effective capacity as it has an L1 miss rate aspect. Third, EB accounts for latency tolerance in GPUs as well by considering bandwidth. In other words, even if some requests end up incurring high latency, more requests may be processed within the same time interval, increasing the overall received bandwidth. Finally, using EB, we can distinguish the performance impact of requests being cached using a shared or a private organization. However, doing so by using a direct

performance metric (e.g., IPC) would be difficult because our scheme deals with requests, not instructions. Furthermore, performance metrics might vary due to reasons other than L1 performance (e.g., bandwidth obtained from software-managed caches [45]), which can lead to an inaccurate classification of applications during runtime. In our scheme, our proxy EB is defined as:

$$EB = \frac{L2_{Replies}}{L1_{MissRate}} + Home_{Replies} \quad (4.3)$$

where $L2_{Replies}$ and $Home_{Replies}$ are the number of read/write replies from L2 and home core(s), respectively.

At the end of the sampling phase, we evaluate EB for both shared and private L1 organizations and choose the option with higher EB. Figure 4.10b shows the effectiveness of EB in choosing between shared and private L1 organizations. We observe that EB (*DynEB*) achieves the performance improvement of a shared L1 organization for the shared-friendly applications. As for the non-shared-friendly applications, EB performs as well as private for C-BLK and C-TRA. However, for C-NN and S-Spmv, EB falls behind the private L1 organization by up to 33%. To remedy that, we utilize our observation (Section 4.3.3) that such applications have significantly low L1 miss rates (< 10%) and low latency tolerance.

Optimization. We augment our *DynEB* by checking if the sets treated as private have an L1 miss rate lower than $L1MR_{Threshold}$ (= 10% in our evaluation). *DynEB+L1MRpr* denotes the updated *DynEB* in Figure 4.10b. *DynEB+L1MRpr* performs as well as the private L1 organization for the non-shared-friendly applications while maintaining the IPC improvement for the shared-friendly applications. We also updated the AMAT-based metric with the L1 miss rate optimization and, as shown in Figure 4.10a, *DynAMAT+L1MRpr* is still not effective with the shared-friendly applications.

4.4.3 Hardware Overhead

As discussed in Section 4.3.2 and Section 4.4.1, our optimized shared L1 organization and DynEB do not change the L1 caches or the NoC. We only update the request handling architecture to manage the remote accesses. We synthesized the RTL design of the hardware required for our optimized shared L1 organization using the 65nm TSMC libraries in the Synopsys Design Compiler and estimated the area overhead to be 0.085 mm^2 per core. DynEB leads to an additional area overhead of 0.005 mm^2 per core.

4.5 Experimental Setup

Simulated System. Our baseline architecture assumes a generic GPU, consisting of multiple cores that have private local L1 caches. These caches are connected to multiple address-sliced L2 cache banks via a NoC. We use two separate networks: request and reply networks to avoid protocol deadlocks [10]. We faithfully model our shared L1 cache organization, inter-core communication, and other mechanisms using a cycle-level simulator – GPGPU-Sim v.3 [10]. A detailed platform configuration is described in Table 4.1.

Table 4.1: Configuration parameters of the simulated GPU.

Core Features	1400MHz core clock, 28 cores (CUs), SIMD width = $32 (16 \times 2)$
Resources / Core	48KB scratchpad, 32KB register file, Max. 1536 workitems (48 wavefronts, 32 workitems/wavefront)
L1 Caches / Core	16KB 4-way Write-through L1 data cache - Latency = 28 cycles [54] 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
L2 Cache	8-way 128 KB/memory channel (1MB in total) 128B cache block size - Latency = 120 cycles
Features	Memory coalescing and inter-wavefront merging enabled, immediate post dominator based branch divergence handling
Memory Model	8 GDDR5 memory controllers (MCs) FR-FCFS scheduling, 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock, Global linear address space is interleaved among partitions in chunks of 256 bytes [33] Hynix GDDR5 Timing [38]
Interconnect	6×6 mesh topology, 700MHz interconnect clock, 32B flit size, 4 VCs per port, 4 flits/VC, iSLIP VC and switch allocators

Evaluated Applications. We evaluate 28 benchmarks from four suites (CUDA-SDK (C) [86], Rodinia (R) [20], SHOC (S) [26], and PolyBench (P) [100]).

4.6 Experimental Results

In this section, we evaluate and compare the following against a private L1 organization baseline:

- **Shared++:** Our shared L1 organization augmented with the optimizations in Section 4.3.3.

- **DynEB:** Our EB-based dynamic scheme, *augmented* with the *L1MRpr* optimization (Section 4.4.2), to classify applications either as shared-friendly or private-friendly.

- **Best(Private,Shared++):** This configuration statically captures the best of both private and shared L1 organizations by picking the organization that achieves higher IPC.

Effect on Performance. Figure 4.11 shows the IPC performance of our proposed solutions normalized to the private L1 baseline. We observe the following. First, DynEB exploits the benefits of the shared L1 organization for shared-friendly applications. Specifically, DynEB enhances IPC by 22% on average over the private baseline and is within 3% of Best(Private,Shared++) for the shared-friendly applications. This is because DynEB significantly reduces data replication, thus it increases the effective L1 cache capacity. Second, DynEB compensates for the IPC loss of the private-friendly applications under Shared++. As discussed in Section 4.3.3, these applications have a significantly low L1 miss rate and high sensitivity to latency. Thus, their performance suffers because, with Shared++, even a cache hit may have to go through the NoC. DynEB identifies these applications and prefers a private L1 organization for them. Finally, for the insensitive non-shared-friendly applications, DynEB improves performance by 1%, 2%, and 4% over Best(Private,Shared++), Shared++, and private L1 baseline, respectively. This is because DynEB enables each core to adapt to the changing behavior of the executing application and obtain the advantages of both shared and private L1 organizations during different

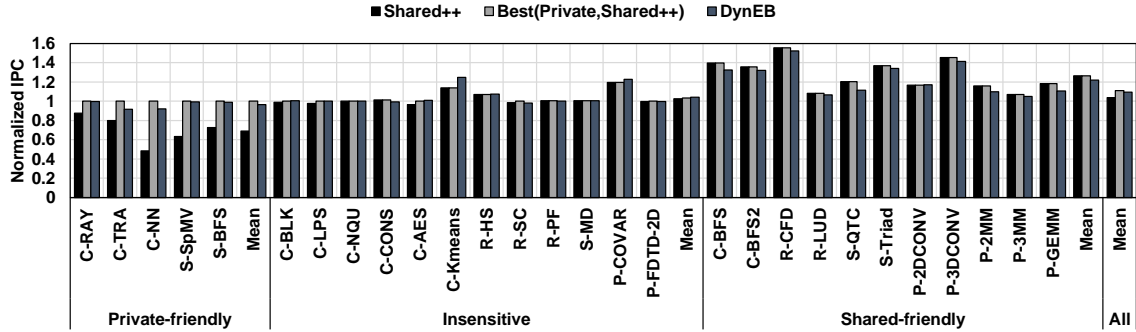


Figure 4.11: The effect of the proposed solutions on IPC. Results are normalized to the private L1 baseline.

phases of execution.

Overall, DynEB improves performance of all evaluated applications by 9%. To demonstrate that, in Figure 4.12, we show normalized speedup for the evaluated applications sorted ascendingly. This is under the shared L1 organization (*Shared*), the optimizations in Section 4.3.3 (*Shared+Chunk* and *Shared++*), and the dynamic scheme (*DynEB*). We observe that although *Shared+Chunk* and *Shared++* push the tail of the S-curve toward the private L1 organization, they still suffer due to the private-friendly applications. However, DynEB can recover the performance loss of these applications.

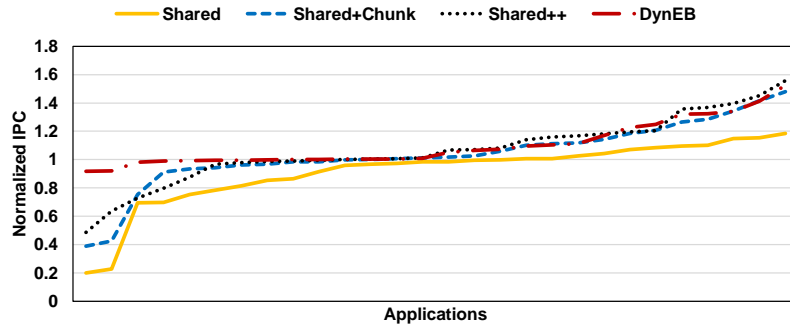


Figure 4.12: The effect of the proposed solutions on IPC as S-curve. Results are normalized to the private L1 baseline.

Effect on L1 Miss Rate. Figure 4.13 shows how effective our solutions are for decreasing L1 miss rate. The results are normalized to the private L1 baseline. We observe the following. First, *Shared++* leads to lower L1 miss rates compared to the private L1 orga-

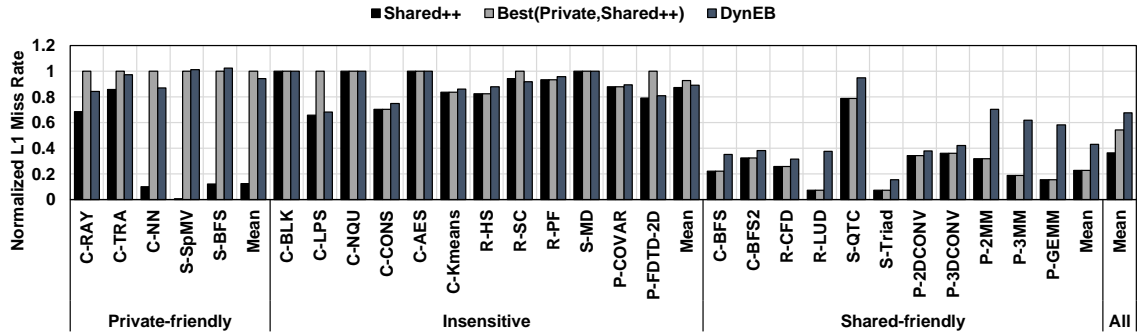


Figure 4.13: The effect of the proposed solutions on L1 miss rate. Results are normalized to the private L1 baseline.

nization because of the extra effective capacity achieved using a shared L1 organization. Specifically, with Shared++, the L1 miss rate drops by 77% and 88% for shared-friendly and private-friendly applications, respectively. As for the insensitive non-shared-friendly applications, Shared++ reduces the L1 miss rate by only 13% as these applications possess low data replication (Figure 4.1).

Second, for shared-friendly applications, DynEB decreases the L1 miss rate by 57% compared to a private L1 organization. This is because DynEB aims to adapt to the shared-friendly nature of these applications and executes them under a shared L1 organization. However, DynEB causes a 88% increase in the L1 miss rate compared to Shared++ because it runs half the cache sets as private during sampling. Additionally, some cores may end up running under a private L1 organization during some execution phases, which may lead to replication across cores, and thus less effective capacity and higher L1 miss rate. Specifically, Figure 4.14 quantifies the number of replicas across the cores under both private and shared L1 cache organizations and under DynEB. As expected, with Shared++, we maintain only a single copy of the data. However, under *Private*, each core can cache any data from the address range, which may lead to more replications across the cores (2.7 replicas on average). DynEB maintains fewer replicas compared to Private but more compared to Shared++ (1.4 replicas on average). This result conforms with the L1 miss rate increase under DynEB compared to Shared++. Finally, for the private-friendly

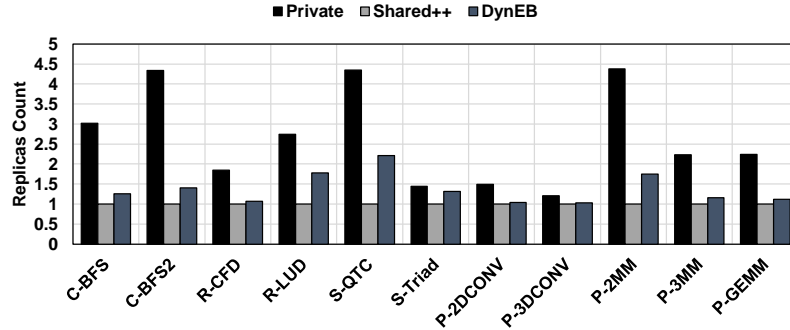


Figure 4.14: The effect of the proposed solutions on number of replicas.

applications, DynEB achieves an L1 miss rate similar to the private L1 baseline, as shown in Figure 4.13. These applications prefer a private L1 organization due to their high L1 hit rate and latency sensitivities, and DynEB runs them under their preferred organization.

Effect on Energy. The shared L1 organization introduces inter-core traffic. However, the chunking optimization (Section 4.3.3) reduces such overheads by only sending the data requested by the requester, not the entire line. Moreover, our proposed schemes reduce L2 and off-chip memory traffic. Using flit and hop counts as well as L2 and memory access counters, we use DSENT [119] and GPUWatch [70] to estimate energy consumption. Overall, the total power under DynEB is similar to baseline, with $<1\%$ reduction averaged across all evaluated applications. Given the improvement in the overall throughput and execution time, the average energy savings under DynEB is 9% compared to the baseline. Therefore, DynEB improves performance-per-watt by 9% and the energy efficiency (performance-per-energy) by 20%, on average across all evaluated applications. For the shared-friendly applications, DynEB maintains the total power consumption (similar to baseline) and saves energy by 18%. Therefore, DynEB enhances performance-per-watt and energy efficiency for the shared-friendly applications by 22% and 49%, respectively.

Effect on Latency. Our private L1 baseline and proposed solutions assume a local L1 access latency of 28 cycles. The shared L1 cache organization imposes a latency overhead of 54 cycles, on average, for the communication between the requester and the home cores. Such inter-core communication overhead is insignificant compared to the 247 cycles, on

average, to communicate with L2 in the baseline. Also, such latency overhead does not negatively affect the evaluated applications because of their latency-tolerant nature.

Adaptability of DynEB. The performance results so far show the versatility of DynEB. This is because DynEB utilizes a repeated two-step process of sampling and execution. Thus, DynEB adapts to the changing characteristics of a given application’s execution. Also, DynEB is local per core. Hence, each core independently monitors application needs and decides the desirable mode of execution. To visualize this adaptive nature, Figure 4.15 shows how DynEB changes the execution mode under C-BFS and C-NN for a representative core. For both applications, DynEB identifies the desirable mode of execution and sticks to it for almost the entire execution.

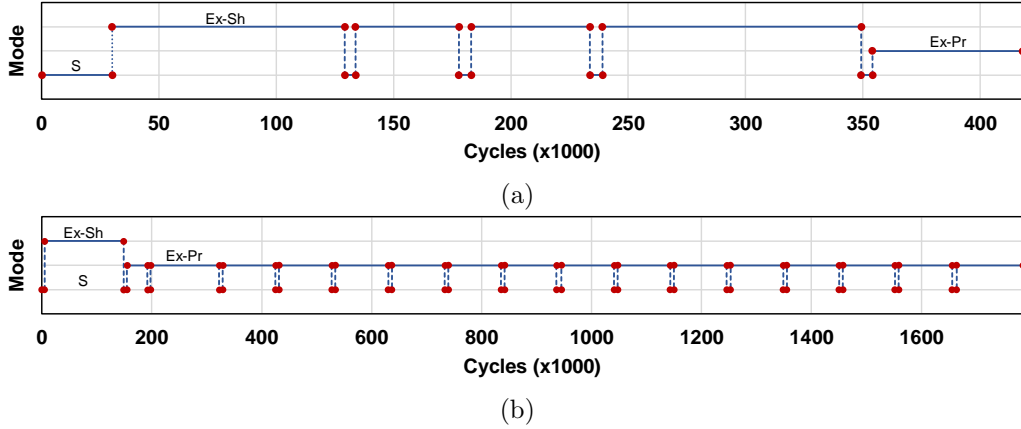


Figure 4.15: Execution timeline under DynEB for (a) C-BFS and (b) C-NN. *S* refers to a sampling phase. *Ex-Sh* and *Ex-Pr* refer to an execution under Shared++ and Private, respectively.

4.6.1 Sensitivity Studies

Effect of L1 Cache Size. We evaluate the effect of doubling the L1 cache size per core on the performance of our schemes. We observe that Shared++ and DynEB achieve around 11% improvement for the shared-friendly applications while maintaining the private performance of the non-shared-friendly applications, over a private baseline with $2\times$ L1 cache size. The lower scope of the improvement is because the working set of some

shared-friendly applications can now fit in the larger L1 cache. Additionally, some of these applications are latency-sensitive, making a shared L1 organization less desirable for them under $2\times$ L1 cache size. We also compare Shared++ and DynEB, for the shared-friendly applications, under the baseline L1 cache size (Table 4.1) against a private L1 organization with double the L1 cache size, denoted as *Private(2x)*. We observe that Shared++ and DynEB improves IPC over *Private(2x)* by 8% and 4%, respectively. This shows that by enabling a shared L1 organization, we can perform better compared to a system with double the L1 cache resources without the extra cost/overhead of increasing the L1 cache size (84% cache area overhead).⁶

Effect of L2 Cache Size. We evaluate a boosted private L1 baseline with double the L2 cache size. We observe almost no performance improvement for the shared-friendly applications compared to the baseline. This is because performance is limited by the L2 reply bandwidth bottleneck [146, 97, 148]. Such a bottleneck is relieved with Shared++ and DynEB as the shared L1 organization utilizes the remote cores as an additional source of bandwidth.

Effect of L1 Access Latency. In our baseline and proposed schemes, we assume 28 cycles access latency for the L1 caches. Figure 4.16 shows average performance with DynEB under different L1 access latency, ranging from 8 to 64 cycles, each normalized to its respective private L1 baseline. We observe that DynEB achieves 17% performance improvement for the shared-friendly applications even under an L1 access latency of 8 cycles while maintaining the performance of the non-shared-friendly applications.

Effect of Core Count. We study the scalability of Shared++ and DynEB using 8×8 mesh and 10×10 mesh NoCs under two different configurations. Figure 4.17 shows performance of both Shared++ and DynEB normalized to their respective private L1 organization baseline. The notation in the figure is (number of cores, number of memory partitions). We observe that IPC follows a similar trend to what we observed using the baseline (28,8) 6×6 mesh. Specifically, with DynEB, we gain significant IPC improvement

⁶The cache area overhead is estimated using CACTI 6.5 [80].

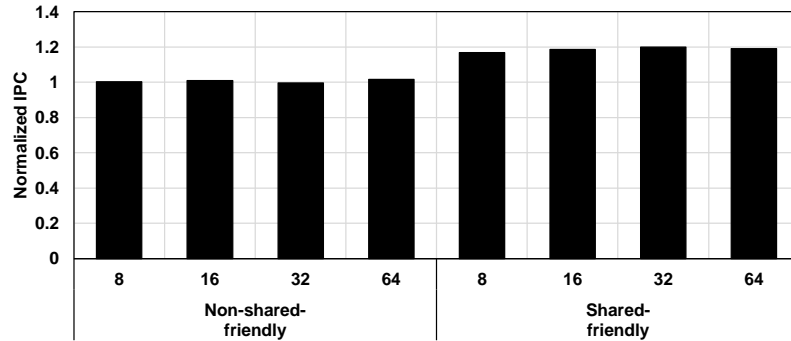


Figure 4.16: Sensitivity study on L1 access latency.

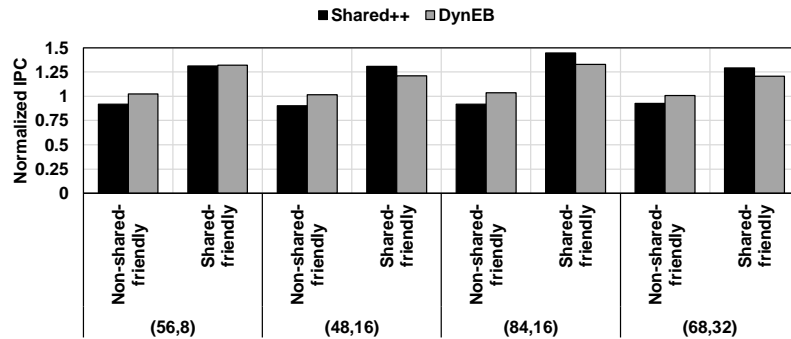


Figure 4.17: Sensitivity study on core and uncore components.

for the shared-friendly applications and maintain the private performance for the non-shared-friendly applications. For example, for an (84,16) system, DynEB improves IPC by 33% and 4%, on average, for the shared-friendly and non-shared-friendly applications, respectively. We observe higher IPC improvement under increased core count because the overall L1 capacity increases with more cores, thus the available collective L1 bandwidth increases under shared L1 organization. Also, with more cores, the *home camping* effect is reduced. Home camping, which is similar to partition camping [4], is caused by memory accesses that are skewed towards a subset of the home cores, which may degrade the performance. Thus, by increasing the core count, each core is assigned a smaller slice of the address range which should likely lead to a uniform traffic distribution among the home cores and hence scales performance.

Effect of Additional Memory Partitions. Figure 4.17 shows the effect of increasing the memory partitions count (this increases total L2 capacity, L2 bandwidth, and memory

bandwidth). For an 8×8 mesh, we study systems with 8 and 16 memory partitions. For a 10×10 mesh, we study systems with 16 and 32 memory partitions. We observe that for the systems with a smaller number of memory partitions, our schemes achieve performance boost at least as good as the systems with a greater number of memory partitions. This is because our schemes are more beneficial with more cores.

Effect of Core to Memory Partition Ratio. Figure 4.17 shows that our schemes can boost IPC for the shared-friendly applications under varying core-to-memory partition ratio. Even in a large (68,32) system, DynEB achieves 21% IPC improvement over the baseline (68,32) 10×10 mesh.

4.6.2 Case Study: Deep-Learning Applications

In this section, we briefly characterize three popular deep-learning workloads from Tango benchmark suite [48], namely AlexNet (AN), ResNet (RN), and SqueezeNet (SN). Additionally, we evaluate their performance under DynEB assuming a big 76-core system with 24 memory partitions (using 10×10 mesh) to mimic recent GPUs oriented to processing deep-learning applications. Figure 4.18a characterizes the evaluated applications in terms of L1 miss rate and line replication ratio (Section 4.2). We observe high replication ratio (up to 98% for SN) and high L1 miss rate (up to 98% for SN) in the evaluated applications, making them perfect candidates for our proposed schemes. Reducing this significant replication across the L1s enables more data to be cached on-chip, which boosts the L1 hit rate, on-chip bandwidth, and overall performance, as shown in Figure 4.18b. Specifically, on average, DynEB reduces the L1 miss rate by 79% for these applications, thus improving their performance by up to $3.9\times$ and by $2.3\times$ on average.

4.6.3 Case Study: Crossbar-based Shared L1 Cache Design

In this section, we evaluate the shared L1 organization under a crossbar NoC. A conventional crossbar connecting cores on one side of the crossbar to L2 slices on the other does not support inter-core communication. Therefore, in this case study, we investigate

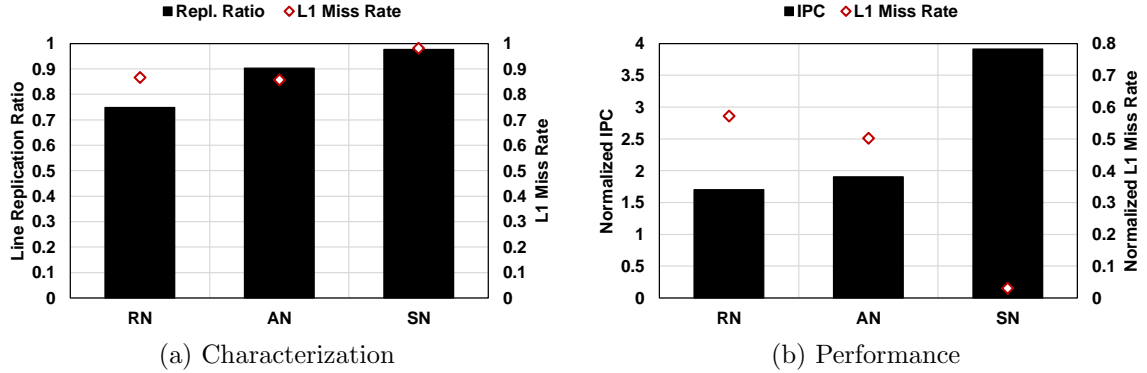


Figure 4.18: Analyzing deep-learning applications in terms of L1 miss rate, line replication ratio, and performance improvement under DynEB.

enabling such communication via the L2 slices. Then, we propose using work distribution crossbar [30, 34], which is already utilized in the graphics (rendering) pipeline, to forward inter-core traffic.

Inter-core Communication via L2 Slices. We update the L2 slices to simply receive a remote request/reply from a requester/home core and forward it back to the target home/requester core. We observe that using L2 to forward the inter-core traffic reduces performance by 23% compared to the private L1 organization. This is due to the contention between L2 replies and forwarded remote traffic, thereby significantly delaying the remote traffic and thus losing performance.

Inter-core Communication via Work Distribution Crossbar. We propose to utilize the work distribution crossbar [30, 34], which already exists and is used by the graphics pipeline, to handle inter-core traffic instead of using the L2 slices. The work distribution crossbar is a scalable multistage butterfly NoC that supports 1) the distribution of triangle and fragment work necessary for load balancing and 2) the synchronization communication necessary for ordering in the graphics pipeline [30]. Therefore, the work distribution crossbar inherently enables inter-core communication. A multistage butterfly (k -ary n -fly) supports a system with up to k^n nodes organized in n stages, where each stage has k^{n-1} switches with a radix k (i.e., $k \times k$ crossbar switch). For our 36-node baseline system (28 cores and 8 memory partitions), we assume a 6-ary 2-fly butterfly NoC. Figure 4.19

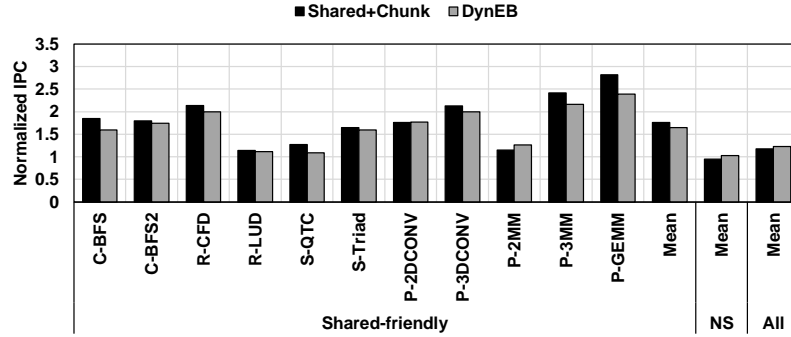


Figure 4.19: Performance of the shared L1 organization in terms of IPC under a crossbar-based system. *NS* refers to non-shared-friendly applications. Results are normalized to a crossbar-based system with private L1 organization.

shows performance of *Shared+Chunk* (Section 4.3.3) and *DynEB* (Section 4.4.2) under the work distribution crossbar. We observe the following. First, *Shared+Chunk* and *DynEB* improve performance of the shared-friendly applications, on average, by 76% and 65%, respectively. Second, for the non-shared-friendly applications (denoted as *NS* in Figure 4.19), *Shared+Chunk* incurs a 5% performance drop, on average. However, *DynEB* maintains these applications’ private performance and offers a 2% performance improvement, on average. This is because *DynEB* obtains the advantages of both shared and private L1 organizations per each application needs.

Overall, *Shared+Chunk* and *DynEB* improve performance of all evaluated applications (denoted as *All* in Figure 4.19) by 18% and 23%, respectively. To demonstrate that, Figure 4.20 summarizes the effect of the shared L1 organization (*Shared*), the proposed chunking optimization (*Shared+Chunk*), and the dynamic scheme (*DynEB*) on the evaluated applications sorted ascendingly. Similar to the mesh-based system, *Shared* and *Shared+Chunk* can provide performance benefits for the shared-friendly applications, but they fail to push the tail of the S-curve towards the private L1 baseline due to the private-friendly applications. On the other hand, *DynEB* recovers the lost performance of the private-friendly applications, while improving the shared-friendly applications.

Scalability. We study the scalability of *Shared+Chunk* and *DynEB* for a 64-node system under two different configurations. Specifically, we evaluate a 48-core system with

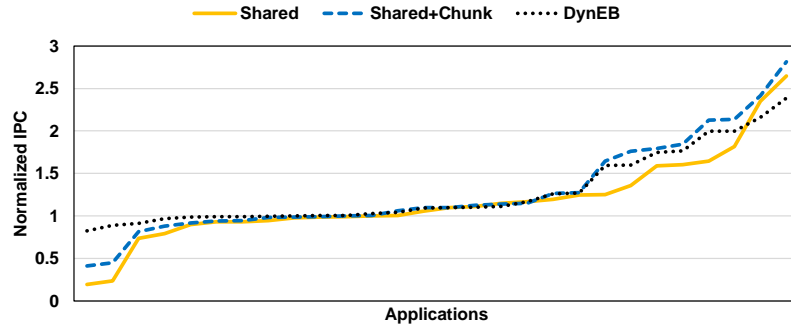


Figure 4.20: The effect of the proposed solutions on IPC as S-curve. Results are normalized to a crossbar-based system with private L1 organization.

16 memory partitions and a 56-core system with 8 memory partitions. For both configurations, we assume a 4-ary 3-fly butterfly NoC. Figure 4.21 shows performance of both Shared+Chunk and DynEB normalized to their respective private L1 baseline. The notation in the figure is (number of cores, number of memory partitions). We observe a similar trend to what we observed with the 36-node system. In particular, Shared+Chunk significantly boosts performance of shared-friendly applications while falling short for non-shared-friendly applications. On the other hand, DynEB matches the performance boost of the Shared+Chunk for shared-friendly applications and maintains the private performance for non-shared-friendly applications. Additionally, similar to our observation in Section 4.6.1, performance improvement under the 56-core system is higher compared to the 48-core system. This is because our proposed shared L1 organization benefits more in the presence of more L1 caches.

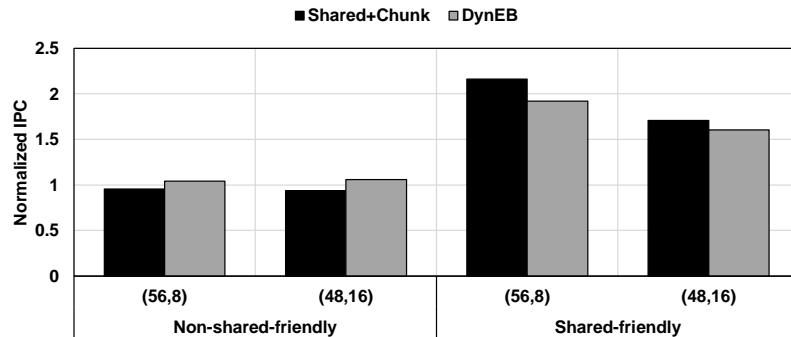


Figure 4.21: Crossbar-based system scalability.

4.7 Related Work

To our knowledge, this is the first work to make a case for using shared L1 caches in GPUs. In this section, we briefly discuss works that are most relevant to this study.

Intra-core Locality in GPUs. There is a large body of work that focuses on exploiting the locality that exists within a private local L1 cache in GPUs [105, 106, 47, 51, 114, 73].

In this work, we specifically focus on the locality that exists across L1 caches. Multiple prior CTA schedulers [65, 8, 122] have used different heuristics to exploit the locality across CTAs. However, they are not ideal [46, 71, 131], and the fundamental problem of cache line replication across private L1 caches remains. While the goal of these schedulers is to improve cache performance, our approach 1) is not dependent on any scheduling algorithm, 2) does not require any software support to determine private and shared data, and 3) does not only reduce replication but can eliminate it. In general, prior L1 cache capacity management works based on bypassing [122, 60], sectoring [104], or compression [9] do not ensure zero data replication across L1s. However, they can continue to improve the performance of local L1 caches while our shared L1 organization can facilitate coordination across L1s for their better utilization.

Inter-core Locality in GPUs. Prior works proposed mechanisms to exploit inter-core locality in GPUs by allowing communication between multiple L1s via connecting L1s through a ring network [28], using the L2 cache to forward inter-core traffic [146], or coherence-like mechanisms [123]. Although these works identified and exploited inter-core locality via inter-core communication, they do not provide a way to reduce or eliminate data replication across L1 caches as we do. Our shared L1 organization utilizes inter-core communication to eliminate the L1 cache wastage without the need for searching or prediction. Zhao *et al.*[148] boost performance of applications with high degrees of data sharing between cores by replicating the shared cache lines across different L2 slices. This is complementary to our work as ours improves the L1 bandwidth utilization while their work improves the L2 bandwidth.

Replication Control in CPUs. Many works have investigated the trade-offs between shared and private caches in the context of CPUs. These works use a flavor of replication control [75, 37, 143, 22, 12, 61, 128], cooperative capacity management mechanisms across cores [17, 102, 107, 27, 36, 62], hybridized shared/private designs [145, 63], OS-level techniques [23, 35], or focus on different architectures/components [115, 13]. Our work differs from those in multiple aspects. First, most of the replication management works in the CPU context consider latency as an important metric for controlling replication. We show that using a latency metric (i.e., AMAT) performs poorly in GPUs as it does not consider the latency-tolerance property of applications. Therefore, we investigate a GPU-oriented metric (i.e., EB) to gauge an application’s affinity towards a private or shared L1 organization. Second, all works in the CPU context investigate the aforementioned approaches for the last-level caches as L1 caches always aim to reduce latency. Due to the latency-tolerant and throughput-oriented behavior of GPUs, optimizing for hit rate (and hence bandwidth) is usually more important than optimizing for latency, so we consider using a shared cache organization for L1 caches. Finally, our mechanism is entirely locally managed, and no coordinated mechanisms are needed to make a decision.

4.8 Chapter Summary

In this work, we show that using a shared L1 cache organization in GPUs is attractive in terms of performance for many applications. We also address the challenges related to applications that lose performance from such an organization with low-overhead communication optimization techniques and a lightweight dynamic mechanism that gauges an application’s affinity towards a private or shared L1 organization and configures the L1 caches accordingly. We show that our techniques can boost performance and can be even more beneficial for future large GPUs with many cores. We hope that this work will open up new research directions in sharing other resources in the GPU (e.g., software-managed caches).

Chapter 5

Analyzing and Leveraging Decoupled L1 Caches in GPUs

Graphics Processing Units (GPUs) use caches to provide on-chip bandwidth as a way to address the memory wall. However, they are not always efficiently utilized for optimal GPU performance. We find that the main source of this inefficiency stems from the tightly-coupled design of cores with L1 caches. First, such a design assumes a per-core private local L1 cache in which each core independently caches the required data. This allows the same cache line to get replicated across cores, which wastes precious cache capacity. Second, due to the many-to-few traffic pattern, the tightly-coupled design leads to low per-core L1 bandwidth utilization while L2/memory is heavily utilized.

To address these inefficiencies, we renovate the conventional GPU cache hierarchy by proposing a new **DC-L1** (**DeCoupled-L1**) cache – an L1 cache separated from the GPU core. We show how decoupling the L1 cache from the GPU core provides opportunities to reduce data replication across the L1s and increase their bandwidth utilization. Specifically, we investigate how to aggregate the DC-L1s; how to manage data placement across the aggregated DC-L1s; and how to efficiently connect the DC-L1s to the GPU cores and the L2/memory partitions. Our evaluation shows that our new cache design boosts the useful L1 cache bandwidth and achieves significant improvement in performance and en-

ergy efficiency across a wide set of GPGPU applications while reducing the overall NoC area footprint.

5.1 Introduction

Graphics Processing Unit (GPU) architectures are a critical component in most high-performance computing systems [125] as they provide faster and more energy efficient execution for many general purpose applications [29, 101, 118, 84, 111, 87, 83, 88, 95]. GPUs employ a conventional two-level cache hierarchy where each core incorporates a private L1 cache and all the GPU cores are connected via a Network-on-Chip (NoC) to a shared and banked L2 cache. The L1 and L2 caches are used to boost the on-chip bandwidth as a means to address the well-known memory wall problem [138]. An increase in the on-chip bandwidth translates into performance improvements for memory-sensitive applications [82, 134, 132]. Therefore, prior research efforts developed hardware and software schemes to improve cache performance [105, 47, 46, 51, 42, 58, 148, 133]. However, we find that the conventional cache hierarchy leads to inefficient utilization of the valuable on-chip caches. Specifically, the tight coupling between the GPU cores and the L1 caches, under this conventional cache organization, results in the following two inefficiencies.

The first inefficiency stems from the many-to-few communication between the L1s and the L2 banks. This puts more pressure on the few L2s and less pressure on the many per-core L1s, which results in a low bandwidth utilization for the per-core L1s [43]. The second inefficiency is due to the private nature of the L1 caches. This may lead to high cache line (data) replication across the L1 caches [59, 28, 72, 71] as each GPU core may independently cache the same cache line. Such replication effectively wastes the overall L1 cache capacity, leading to lower L1 hit rates and hence reduces its useful bandwidth. If cache line replication is reduced, then the L1 caches can effectively provide more capacity to cache more data, leading to higher hit rates, more delivered on-chip bandwidth, and

reduced pressure on the L2 and memory.

In this work, we address these two inefficiencies by breaking the tight coupling between the GPU cores and the L1 caches. To achieve that, we renovate the GPU two-level cache hierarchy and propose **DeCoupled-L1 (DC-L1)** caches, where we separate the L1 caches from the GPU cores. The decoupled nature of the DC-L1 caches enables aggregating the DC-L1 caches into bigger caches (while maintaining the total L1 cache capacity), in which each DC-L1 cache is accessed by multiple GPU cores. Aggregating DC-L1 caches improves their individual bandwidth utilizations and reduces data replication across the DC-L1s as more cores are accessing a given DC-L1. Although extreme aggregation of DC-L1s (all cores accessing one DC-L1) eliminates replication and improves DC-L1 bandwidth utilization, it can drastically reduce the overall peak L1 bandwidth and hence performance. In this work, we use the aggregation granularity as a knob to reduce replication and improve cache bandwidth utilization while managing the overall peak L1 bandwidth.

Once we achieve a suitable aggregation granularity, we propose managing data placement across the DC-L1s to further reduce replication. Specifically, we evaluate a shared DC-L1 cache design to eliminate replication across the DC-L1 caches. With a shared DC-L1 cache design, each DC-L1 *exclusively* caches a unique slice of the address range. This ensures only one copy of data exists across DC-L1s, thereby eliminating replication and making better use of the finite cache capacity. However, we show that the shared DC-L1 cache design requires all-to-all communication between the GPU cores and the DC-L1s, which imposes significant NoC area/power overheads. Additionally, it imposes both scalability and NoC clocking challenges. Therefore, we propose to vary the sharing granularity using a **Clustered DC-L1** cache design to balance the trade-off between the replication waste and the NoC overheads. With such a design, we group the DC-L1 caches into clusters and enable the shared cache organization only within each cluster instead of enabling a fully shared cache across all DC-L1s. Therefore, we eliminate replication within the DC-L1 cluster and reduce replication across all the DC-L1s in a controlled fashion. This improves overall GPU throughput while reducing the overall GPU area and energy

requirements.

To enable these DC-L1-based cache designs, a revamped NoC design is also required to connect the DC-L1 caches to the GPU cores and the memory partitions. The updated NoC design depends on the granularity of DC-L1 aggregation and the granularity of sharing under the clustered DC-L1 cache design. Also, given the shared nature of the L2 slices and the unique address range assigned to each DC-L1 within a cluster, each DC-L1 will communicate only with a few L2 slices. This further reduces the overall area and energy requirements.

Contributions: To our knowledge, this is the first work that performs a thorough design space exploration and characterization of decoupled L1 caches in GPUs to improve their bandwidth utilization. We address the inefficiencies of the conventional GPU caches and propose renovating both the cache and NoC design. In particular, by decoupling and aggregating the L1 caches, we enable a practical clustered shared L1 cache design that reduces data replication and boosts the L1 bandwidth utilization. Such cache design is supported by a two-level NoC design that is optimized to save the overall area and energy. Overall, this work contributes the following:

- We propose Decoupled-L1 caches (DC-L1) where we dissociate the L1 caches from the GPU cores and aggregate them.
- We propose co-designing the DC-L1 caches and the NoC to build a shared DC-L1 cache organization that *eliminates* cache line replication across the DC-L1 caches. We show that our holistic approach significantly improves the collective L1 hit rates and reduces the bandwidth pressure to the lower levels of the memory hierarchy for the applications that are sensitive to high replication volume.
- To address the drawbacks of the shared DC-L1 organization (NoC area/power overheads, scalability, and clocking challenges), we propose a clustered shared DC-L1 cache design that *limits* data replication. This cache design enables a cluster of GPU cores to access a cluster of shared DC-L1 caches, thus eliminating data replication within the cluster and reducing it across the GPU.

- We evaluate our clustered DC-L1 design across 28 GPGPU and deep-learning applications. On average, our proposal boosts performance by 75% (up to $8\times$) for the applications that are sensitive to high data replication without degrading performance of the applications that are insensitive. Additionally, our proposal reduces the total NoC area by 50% and improves energy efficiency by 95%.

5.2 Motivation and Analysis

In this section, we discuss the inefficiencies of tightly coupled L1 caches in GPUs and make a case for separating and aggregating the L1 caches to address those drawbacks.

5.2.1 Inefficiency#1: Cache Line Replication across L1 Caches

With the baseline private L1 cache design, each GPU core satisfies its L1 requests from the local L1 cache. On a miss, each core *independently* fetches the required data from the L2 cache. This may lead to replication across L1s if the cores request the same cache line, leading to wasted cache capacity.

Wasted L1 Cache Capacity. The volume of replication of the evaluated applications is shown in Figure 5.1 in terms of *Replication Ratio*, sorted in ascending order. Replication ratio is defined as the ratio of L1 misses that can be found in other L1 caches to total L1 misses. We observe that the replication ratio varies across the evaluated applications. Specifically, some applications have no replication (e.g., C-BLK) or low replication (e.g., C-RAY), while others have high replication (e.g., C-BFS). We also observe that deep-learning applications (T-AlexNet, T-ResNet, and T-SqueezeNet) have significantly high replication. For example, T-AlexNet has a replication ratio of 95%.

Identifying Replication-sensitive Applications. The waste due to data replication may not affect all applications. Only the applications that are sensitive to larger cache space are *expected* to benefit if the wasted cache space is reduced/eliminated. Therefore, we study performance of the evaluated applications under a $16\times$ larger L1 cache in Fig-

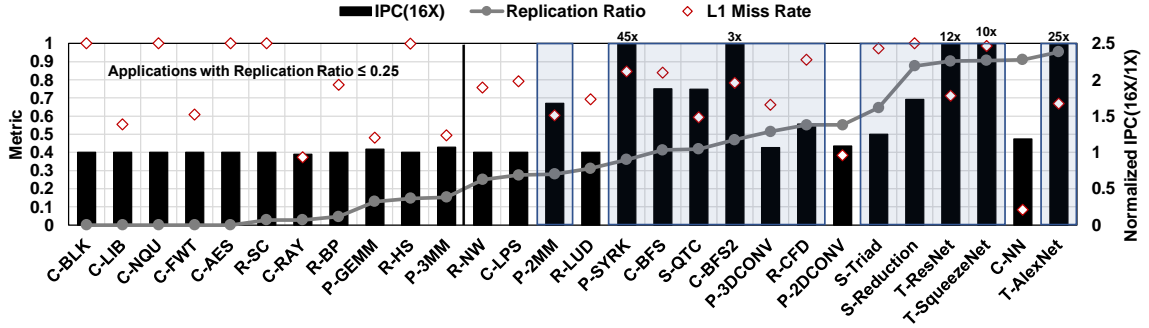


Figure 5.1: Performance of the evaluated applications in terms of IPC improvement under $16\times$ the L1 cache (normalized to baseline), L1 miss rate, and cache line replication ratio. The left-hand y-axis represents replication ratio and raw L1 miss rate. The experimental methodology is detailed in Section 5.7.

ure 5.1. We observe that 15 applications are both capacity-sensitive and possess high data replication. To identify the subset of the capacity-sensitive applications that are *replication-sensitive*, we study their L1 miss rates. Applications with low L1 miss rates (e.g., C-NN) may not suffer under private L1 caches because the majority of their requests can be satisfied locally. In general, we consider an application to be replication-sensitive if it 1) has a replication ratio of $>25\%$, 2) has an L1 miss rate of $>50\%$, and 3) observes a speedup of $>5\%$ with $16\times$ capacity.¹ Based on these criteria, we observe that 12 applications are replication-sensitive (marked by the blue boxes in Figure 5.1).

Effect of Eliminating Replication. To estimate the potential performance benefits of eliminating data replication for the replication-sensitive applications, we evaluate a hypothetical design where all GPU cores access a *single* L1 cache (while maintaining the total L1 cache capacity and bandwidth) to ensure no replication in Figure 5.2. We observe that the L1 miss rate is reduced significantly by an average of 89.5% under such design. This is because removing replication allows for more data to be cached in L1s, thus improving L1 hit rates. For deep-learning applications, we observe an exceptional 99% reduction in the L1 miss rates as they have high replication volume as shown in Figure 5.1. Overall, for the replication-sensitive applications, the significant reduction in

¹This criteria is empirical and is not used by our proposed designs.

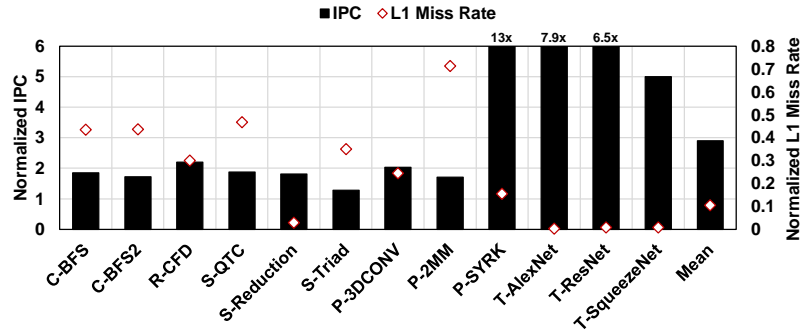


Figure 5.2: Illustrating performance of a hypothetical cache design that eliminates replication across L1s on replication-sensitive applications (normalized to baseline).

L1 miss rates leads to more delivered on-chip bandwidth from the L1s, which translates to an IPC improvement of $2.9\times$ on average.

5.2.2 Inefficiency#2: Low L1 Cache Utilization

The tight coupling of the L1 caches and GPU cores along with the many-to-few communication pattern (between the L1s and the L2 banks) puts more pressure on the few L2 banks and less pressure on the many L1 caches. This leads to low bandwidth utilization of the per-core L1 caches. We define the per-core L1 bandwidth utilization as the ratio of a core’s L1 accesses (requests) over the total cycle count. Figure 5.3 shows the maximum bandwidth utilization of the L1 cache data port, across all L1s, under all the evaluated applications sorted in ascending order. We observe that the highest bandwidth utilization of the L1 data ports is 18%. The low bandwidth utilization of the L1 caches is also shown by recent work [43, 54] where they show a significant gap between the theoretical peak L1 cache bandwidth and the achieved L1 cache bandwidth.² Additionally, we verified our findings in Figure 5.3 using nvprof 9.2 [91] on a Quadro P6000 GPU and observed the maximum L1 bandwidth utilization to be 11.1% on average. For a comprehensive view, we also study the utilization of NoC links that carry the data replies from the L2 to the GPU cores. Figure 5.3 shows the maximum NoC link utilization, across all links connected to

²The higher measured L1 bandwidth reported by [43] is due to using a microbenchmark designed to pressure the data load requests on the L1 cache.

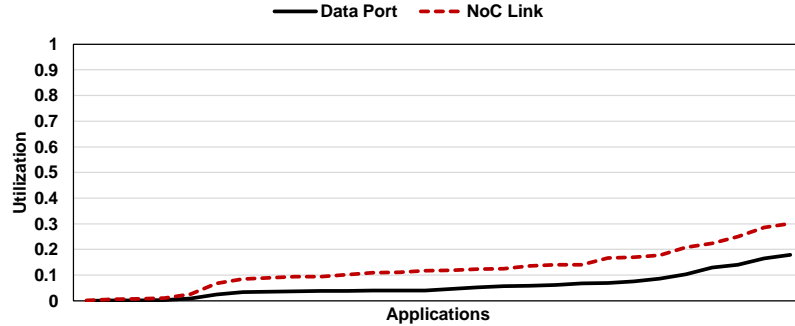


Figure 5.3: L1 cache data port and NoC link utilization.

GPU cores, for all evaluated applications in ascending order. Similar to the data port, the maximum link utilization is low (30%), which further shows the underutilization of the per-core L1s.

5.2.3 Solution: Decouple and Aggregate L1 Caches

We point out that a major cause for the inefficiencies is the tight coupling between the GPU core and the L1 cache within it. Therefore, we propose a **DeCoupled-L1** cache (**DC-L1**) – an L1 cache separated from the GPU core (Section 5.3). This breaks the tight coupling between these entities and enables optimizations to reduce replication across the L1s and boost their bandwidth utilizations. These optimizations include aggregating the DC-L1 caches (Section 5.4) and managing data placement across the aggregated caches (Section 5.5 and Section 5.6).

5.3 Decoupled-L1 (DC-L1) Design

In this section, we describe Decoupled-L1 (DC-L1) caches and demonstrate how the DC-L1s, GPU cores, and L2/memory are connected. Also, we discuss the request/reply flow under the DC-L1-based design.

DC-L1 Node and NoC Design. Figure 5.4 **A** shows our DC-L1 node design. A DC-L1 node simply contains the DC-L1 cache (DC-L1\$), two queues to handle the traffic from/to the GPU core, and two queues to handle the traffic to/from the L2 and memory

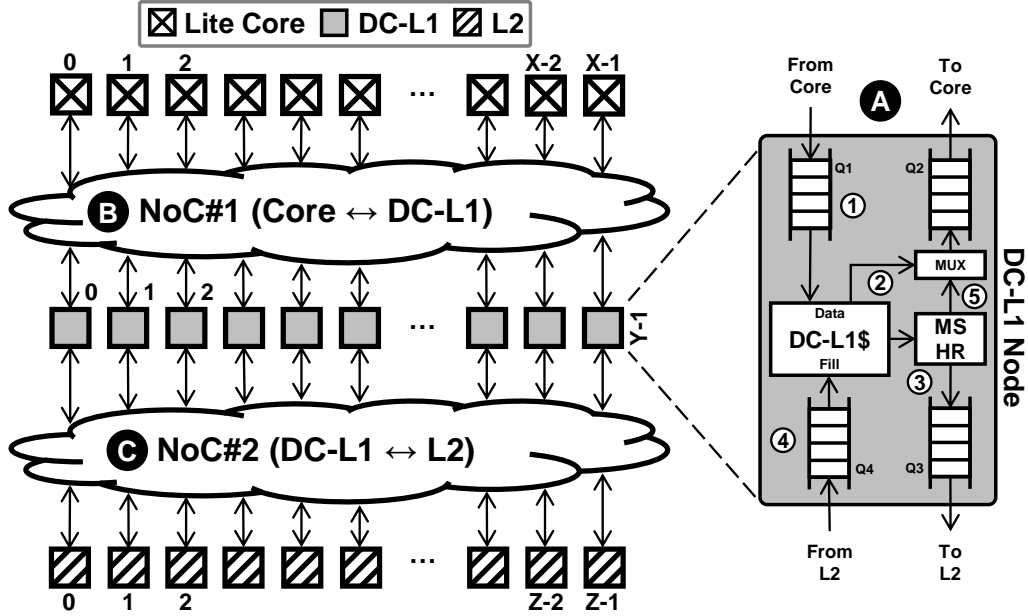


Figure 5.4: Decoupled-L1 (DC-L1) node and NoC design.

partitions.³ A GPU core in our design is a *Lite Core*. A lite GPU core is similar to the baseline GPU core but without the L1 data cache and the associated MSHR. Because the L1 caches are now separated from the GPU cores, we breakdown the NoC into two parts. The first NoC **B** (NoC#1) connects the GPU cores and the DC-L1 nodes. The second NoC **C** (NoC#2) connects the DC-L1 nodes and the L2/memory. The design of both NoCs is determined by the number of DC-L1 nodes and the cache organization.

Handling Read Requests. With a DC-L1-based design, an L1 read request is injected into NoC#1 to the target DC-L1 node as the GPU core does not have an L1 cache (and associated MSHR) anymore. The target DC-L1 node queues the received request into $Q1$ ① to be served by the DC-L1\$ in FIFO manner. The request at the head of the queue accesses the DC-L1\$. If the request hits in the DC-L1\$, then the DC-L1 node queues the read reply into $Q2$ ② for injection into NoC#1 back to the GPU core. If the request misses in the DC-L1\$, then the DC-L1 node queues the request into $Q3$ ③ to be forwarded to the L2 cache through NoC#2. Once a read reply is received from the L2 via NoC#2, the

³The hardware overhead of the DC-L1 node is discussed in Section 5.8.

DC-L1 node queues the reply into Q_4 ④ to be installed in its DC-L1\$. Concurrently while caching the reply, the DC-L1 queues the reply to be injected to the requester GPU core through NoC#1 ⑤. This read reply is not necessarily full cache line sized; it only needs to provide the data requested by the requester GPU core. This is because, in our design, a GPU core does not have an L1 cache to install the data in. Hence, if the whole cache line is not required by the core, then sending the read reply at a full cache line granularity from the DC-L1 will waste NoC#1 bandwidth [104]. We refer to this optimization as *Sectoring*.

Handling Write Requests. A write request follows the same flow as a read request. However, because we use a write-evict policy for the DC-L1 caches (Section 5.7), on a write hit, a given write request evicts the cache line from the DC-L1\$. The evicted cache line is forwarded to the L2 cache through NoC#2. On a write miss, no cache line is allocated at the DC-L1\$ and the updated data is delivered to the L2 cache as we use a no-write-allocate policy. Once a write ACK is received from the L2, the DC-L1 node forwards the write ACK to the requester GPU core via NoC#1.

Handling Non-L1 Requests. Because the DC-L1 node is on the path to L2, all the instruction, texture, and constant cache misses from the GPU core must go through the DC-L1 node to be forwarded to the L2 via NoC#2. These non-L1 requests do not access the DC-L1\$. A given non-L1 request is simply moved from Q_1 to Q_3 bypassing the DC-L1\$. Similarly, a non-L1 reply is moved from Q_4 to Q_2 . For clarity, the bypassing of DC-L1\$ is not shown in Figure 5.4.

Handling Atomic Operations. In the baseline, atomic operations skip the L1 cache and are handled at L2/MC [10]. Similarly, in our design, atomic operations skip the DC-L1 cache and are handled at the unaltered L2/MC.

5.4 Private DC-L1 Caches

In this section, we address the inefficiencies discussed in Section 5.2. Specifically, to reduce data replication across the DC-L1s and improve their individual bandwidth utilizations, we investigate aggregating the DC-L1s into larger DC-L1s.

5.4.1 Designing Private DC-L1 Caches

Given a system with X GPU cores and X DC-L1 nodes, where each DC-L1 node hosts a single DC-L1\$ with size C , we aggregate these X DC-L1\$ to form Y bigger DC-L1\$ ($X > Y$). Each of the Y larger DC-L1\$ has a size of $(X \times C)/Y$ and is hosted in a DC-L1 node.⁴ Under this design, each DC-L1 node is accessed privately by a group of $N = X/Y$ cores via an $N \times 1$ crossbar in NoC#1. We refer to this private aggregated DC-L1 design as **PrY**. In that sense, we can vary Y to control the granularity of aggregation (X/Y). Table 5.1 shows the different NoC configurations of a private DC-L1 design using different Y values under our 80-core baseline (Section 5.7). For example, Figure 5.5 shows the design of Pr40, where 80 DC-L1s are aggregated into 40 DC-L1s each with double the cache capacity. With Pr40, each DC-L1 node is privately accessed by two cores via a 2×1 crossbar in NoC#1. Such a private cache organization allows each DC-L1 to cache any line. For example, given the different address ranges represented by different patterns in Figure 5.5, a private DC-L1 cache can store any line from all address ranges.

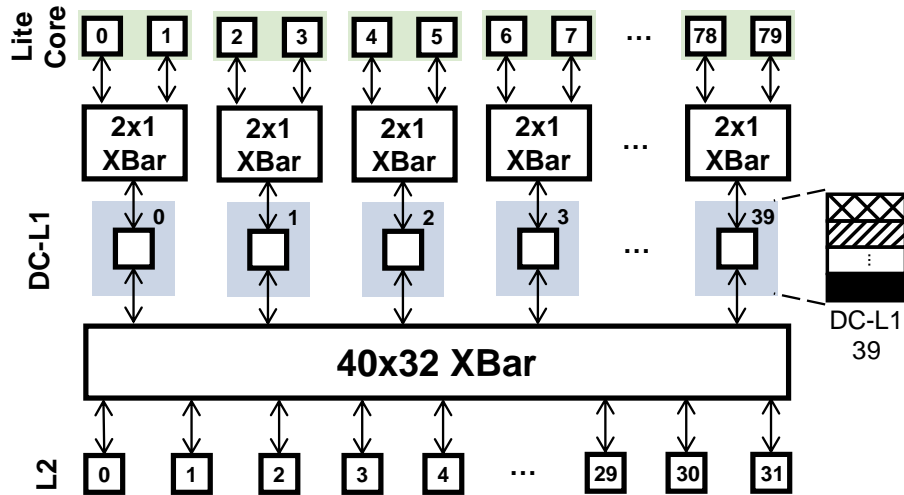
5.4.2 Evaluating Private DC-L1 Caches

Performance. We evaluate performance of a private DC-L1 cache design on the replication-sensitive applications in terms of IPC and DC-L1 miss rate, normalized to the private L1 baseline in Figure 5.6. We start with Pr80 where we decouple the L1 caches without any aggregation. As shown in Table 5.1, Pr80 connects the GPU cores to the corresponding DC-L1 nodes using *32-Byte* direct links in NoC#1, while connect-

⁴The size of the MSHR is scaled proportionally when aggregating the DC-L1s. For example, when aggregating two DC-L1s, the associated MSHR is doubled in size.

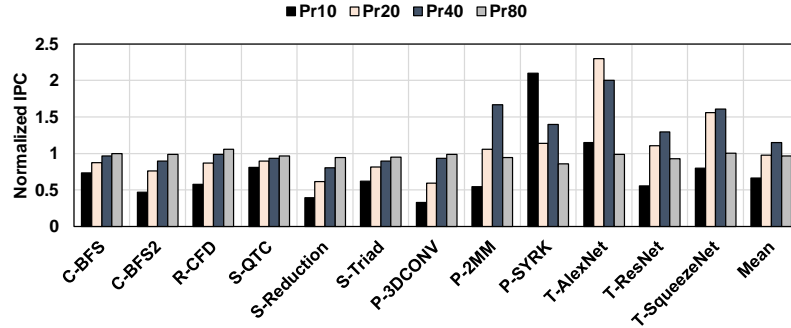
Table 5.1: NoC size and peak L1 bandwidth reduction under different private DC-L1 configurations.

Config.	NoC#1 Crossbars	NoC#2 Crossbars	Peak L1 BW	Peak L1 BW Drop
Baseline	NA	($\times 1$) 80×32 XBar	$\frac{128 \text{ Bytes}}{1 \text{ Cycle}} \times 80$	-
Pr80	($\times 80$) Direct Links	($\times 1$) 80×32 XBar	$\frac{128 \text{ Bytes}}{4 \text{ Cycles}} \times 80$	$4\times$
Pr40	($\times 40$) 2×1 XBar	($\times 1$) 40×32 XBar	$\frac{128 \text{ Bytes}}{4 \text{ Cycles}} \times 40$	$8\times$
Pr20	($\times 20$) 4×1 XBar	($\times 1$) 20×32 XBar	$\frac{128 \text{ Bytes}}{4 \text{ Cycles}} \times 20$	$16\times$
Pr10	($\times 10$) 8×1 XBar	($\times 1$) 10×32 XBar	$\frac{128 \text{ Bytes}}{4 \text{ Cycles}} \times 10$	$32\times$

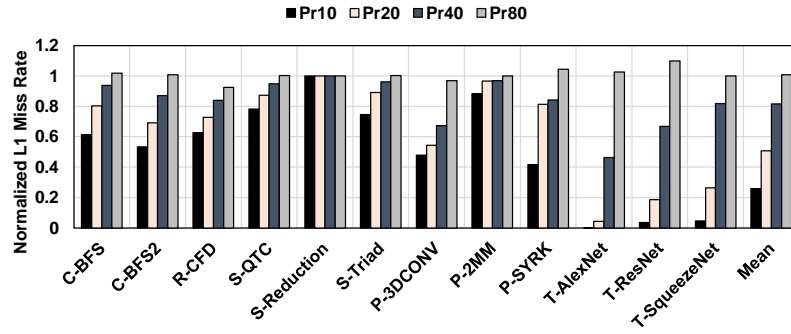
**Figure 5.5:** Pr40 design.

ing the DC-L1 nodes to L2/memory using a 80×32 crossbar in NoC#2.⁵ Because of the 32B links, the 128B cache line fetched from a given DC-L1 will be decomposed into four 32B chunks (assuming no control metadata) to be delivered sequentially to a requester core. Therefore, the peak theoretical DC-L1 cache bandwidth is $4\times$ less than baseline (Table 5.1). Nonetheless, as shown in Figure 5.6a, performance of Pr80 in terms of IPC is similar to baseline (3% drop on average). This is attributed to the latency tolerance property of GPGPU applications. This also shows that the peak L1 bandwidth is suffi-

⁵The 32B links are inline with the baseline link width in Section 5.7.



(a) IPC



(b) L1 miss rate

Figure 5.6: Performance under private DC-L1 design. Results are normalized to the private L1 baseline.

ciently abundant (as shown in [43, 54]) that even reducing it by $4\times$, we still achieve similar throughput. However, Pr80 does not reduce DC-L1 miss rate as shown in Figure 5.6b. This indicates no reduction in data replication across the DC-L1s. This is expected as, under Pr80, we do not aggregate the DC-L1s.

Under aggregated DC-L1s, a group of cores access a common caching resource (a single DC-L1\$). For example, under Pr40, two cores access a single DC-L1\$. As there is no cache line replication within a single cache, DC-L1 aggregation should reduce replication and enhance the collective hit rate of the DC-L1s. This is shown in Figure 5.6b where the DC-L1 miss rate drops by 19%, 49%, and 74% under Pr40, Pr20, and Pr10, respectively. In terms of throughput, Pr40 improves the IPC of the replication-sensitive applications by 15%, on average, compared to baseline. This is because of the reduction in data replication, hence higher DC-L1 hit rates, which leads to an increase in the on-chip bandwidth and

overall performance. On the other hand, Pr20 and Pr10 reduce average performance by 3% and 34%, respectively. This is due to the significant drop in their peak L1 bandwidth (Table 5.1) and the lower NoC bandwidth due to using smaller crossbars in NoC#2, which limits their bisection bandwidth.

To understand the scope of the private DC-L1 design under different DC-L1 node counts, we assume a perfect DC-L1\$ with 100% hit rate. Figure 5.7 shows the average IPC improvement for the replication-sensitive applications under both normal and perfect DC-L1\$ normalized to the private L1 baseline. Three observations are in order. First, Pr10 under perfect DC-L1\$ still leads to a drop in performance by 28% due to the reduced DC-L1 cache and NoC bandwidth. Second, Pr20 and Pr40 improve performance under perfect DC-L1\$ by 40% and 90%, respectively, compared to their normal DC-L1\$ counterparts. However, Pr40 has a higher IPC boost of $2.2\times$ compared to the baseline with normal L1 cache. Finally, Pr80 under perfect DC-L1\$ boosts performance by $3.3\times$ compared to Pr80 with normal DC-L1\$. However, it does not match the $5.2\times$ improvement of having a perfect L1 cache in the baseline private case (denoted as *Base*). This is due to the $4\times$ drop in the peak L1 bandwidth under Pr80.

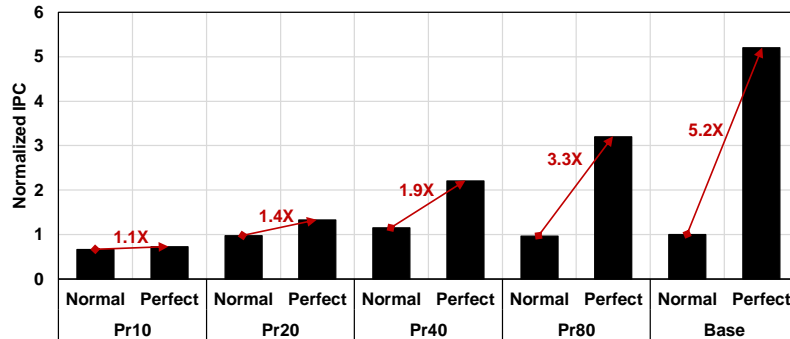


Figure 5.7: Illustrating the potential performance of DC-L1 under private DC-L1 design. Results are normalized to the private L1 baseline.

Area & Power. We study NoC area and static power breakdown under different private DC-L1 configurations, normalized to the private baseline in Figure 5.8.⁶ We use

⁶We estimate the NoC dynamic power in Section 5.8.

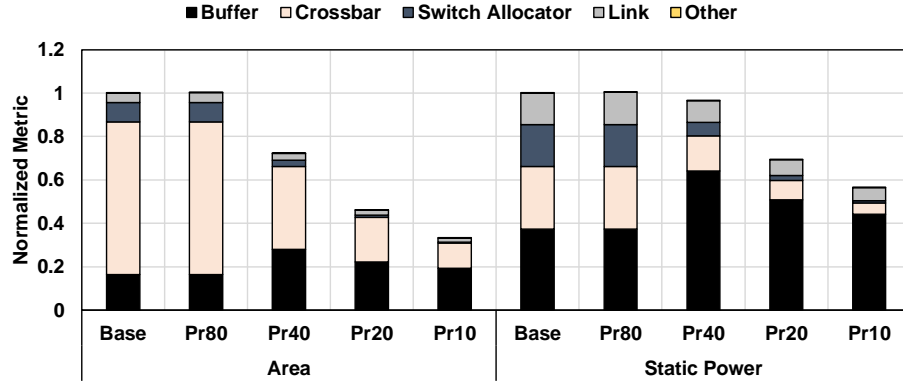


Figure 5.8: NoC area and static power under private DC-L1 design. Results are normalized to the private L1 baseline.

DSENT [119] to model the crossbars in both NoC#1 and NoC#2, assuming a 22nm technology and assuming that all the evaluated crossbars can operate at the same clock frequency. We observe the following. First, Pr80 adds insignificant area and static power overhead compared to the baseline. This is because Pr80 only adds links to connect a given GPU core to its corresponding DC-L1 node. Second, Pr40, Pr20, and Pr10 reduce the NoC area by 28%, 54%, and 67%, respectively. This is due to breaking down the baseline 80×32 crossbar into smaller crossbars, thus reducing the NoC area [148, 147]. Specifically, using smaller crossbars leads to a lower area overhead from *Crossbar* and the *Switch Allocator* within each router. Third, the static power reduction under Pr40 is just 4%. This is because the small crossbars of Pr40 reduces the per-router static power from the *Crossbar* and the *Switch Allocator* components; however, it increases the static power from *Buffer* components (due to more routers). Finally, the static power reduction under Pr20 or Pr10 is more than Pr40. This is because Pr40 uses more small crossbars in NoC#1 and a bigger crossbar in NoC#2 (Table 5.1).

Verdict. Because Pr40 improves throughput while reducing the NoC area and maintaining the power consumption (compared to baseline), we choose 40 DC-L1 nodes for the rest of this work. However, to bridge the Pr40 performance gap between normal and perfect DC-L1, we need to investigate other innovative ways to reduce data replication, thus further improving the DC-L1s collective hit rates.

5.5 Shared DC-L1 Caches

To eliminate data replication across the DC-L1 caches, we investigate enabling a shared DC-L1 cache organization. Under a shared organization, the entire address range is interleaved across all the DC-L1s and such mapping is fixed. In other words, each DC-L1 exclusively caches data from a non-overlapping address range. A DC-L1 that can cache a given cache line is the *home DC-L1* of that line. For example, Figure 5.9 shows that the black address range can be cached by only DC-L1 39. In other words, DC-L1 39 is the home of the black address range. Because an exclusive slice of the address range maps to a single DC-L1, the shared organization ensures no cache line replication across DC-L1 caches.

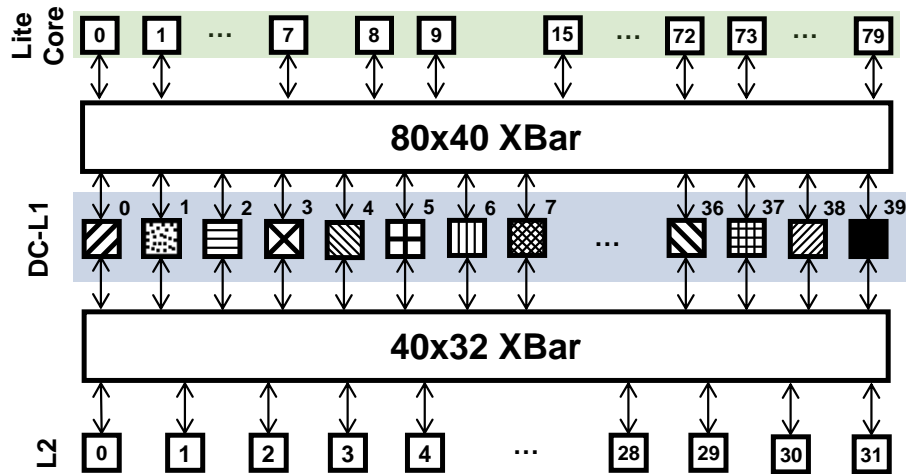


Figure 5.9: Sh40 design.

5.5.1 Designing Shared DC-L1 Caches

To enable a shared DC-L1 organization, any core needs access to any DC-L1 node. Figure 5.9 shows one possible design to achieve that under our setup (Section 5.7). In this design, 80 GPU cores are connected to 40 DC-L1 nodes via an 80×40 crossbar in NoC#1. We refer to this design as **Sh40** (or **ShY** in general, where Y is the total number of DC-L1 nodes).

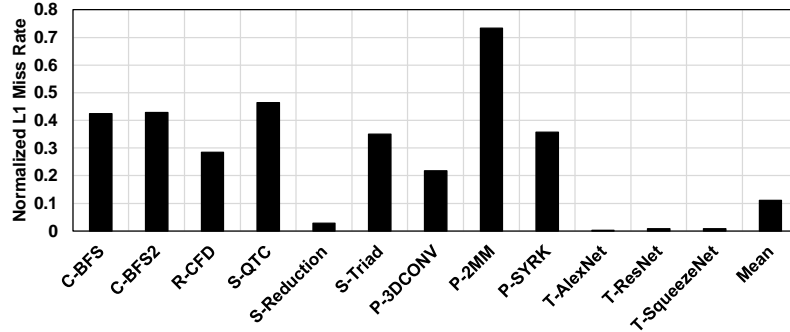
Selecting the Home DC-L1. To select the home DC-L1 for a given cache line, we use the *home bits*. These home bits are selected from the physical address of the request. The process of selecting these bits is analogous to selecting the appropriate L2 bank based on the physical address. In general, Sh Y design requires $\lceil \log_2(Y) \rceil$ home bits to identify the home DC-L1.

Handling Requests. An L1 or a non-L1 request/reply (read or write) follows the same flow in Section 5.3. The only difference is that the request/reply is forwarded to the home DC-L1.

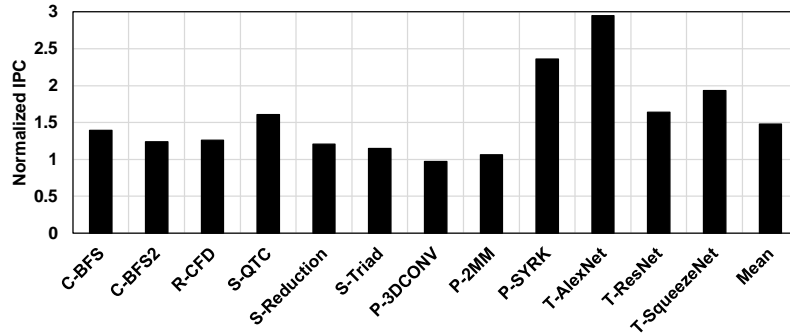
5.5.2 Evaluating Shared DC-L1 Caches

Performance. We evaluate the performance of Sh40 on the replication-sensitive applications in terms of DC-L1 miss rate and IPC, normalized to the private L1 baseline in Figure 5.10. Under Sh40, the DC-L1 miss rate drops significantly by an average of 89% (minimum = 27%, maximum = 99%). The significant drop in the DC-L1 miss rate is expected as these applications have high data replication across the DC-L1s (Section 5.2), which is eliminated under shared DC-L1 design. This effectively provides L1 cache capacity to store more cache lines, thus improving the L1 hit rate and the on-chip bandwidth. The boosted on-chip bandwidth from the DC-L1s is translated into a throughput boost of 48% on average (up to $2.9\times$ for T-AlexNet) as shown in Figure 5.10b.

However, two replication-sensitive applications (P-2MM, P-3DCONV) do not benefit from Sh40. Specifically, P-2MM achieves only 6% speedup because Sh40 can lead to a *partition camping* problem. Partition camping [4] is caused by cache accesses that are skewed toward a subset of the DC-L1 nodes. This leads to load imbalance between the DC-L1s and limits the benefits of the shared cache design. As for P-3DCONV, it suffers a 3% performance loss with Sh40 due to its sensitivity to available L1 cache bandwidth. Specifically, the traffic in NoC#1 is high due to the absence of the L1 caches from the GPU cores and the high DC-L1 hit rate with Sh40. Thus, the reduced peak cache bandwidth with 40 DC-L1s (Table 5.1) limits the performance benefits of P-3DCONV under Sh40.



(a) L1 miss rate



(b) IPC

Figure 5.10: Performance under Sh40. Results are normalized to the private L1 baseline.

Area & Power. Although Sh40 improves performance of the replication-sensitive applications, it uses an 80×40 crossbar in NoC#1 to route the traffic from/to any GPU core to/from any DC-L1 node. This crossbar, in addition to a 40×32 crossbar in NoC#2, incurs a NoC area overhead of 69% and a NoC static power overhead of 57% compared to the private baseline.

Replication-insensitive Applications. We evaluate performance of Sh40 on the applications that are classified as *replication-insensitive* in Figure 5.11. We observe the following. First, most of these applications perform as well as the baseline (e.g., R-LUD and C-BLK). These applications have a high tolerance to the latency overhead induced by the DC-L1 design. Second, R-SC performs better than the baseline. This is because R-SC suffers from work distribution imbalance as some cores are assigned more CTAs. This leads to imbalance in L1 cache accesses across the cores under the baseline. How-

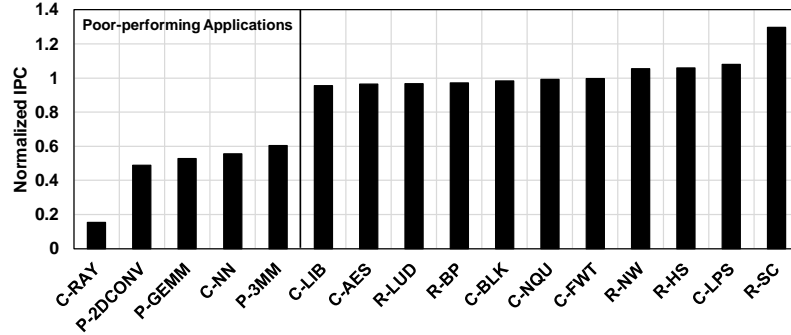


Figure 5.11: Performance of replication-insensitive applications under Sh40. Results are normalized to the private L1 baseline.

ever, given the shared nature of the DC-L1 under Sh40, such imbalance in DC-L1 cache accesses is mitigated, which reduces the bottlenecks and improves performance. Second, five applications suffer a drop in performance with Sh40 (minimum = 40%, maximum = 85%). We refer to these applications as *poor-performing applications* in Figure 5.11. These applications either have high L1 hit rates and low latency tolerance (C-NN), suffer from partition camping (C-RAY, P-3MM, and P-GEMM), or are sensitive to the reduced peak L1 cache bandwidth (P-2DCONV).

Verdict. Although Sh40 significantly improves performance of replication-sensitive applications, it incurs a considerable NoC area and static power overhead. Also, some replication-insensitive applications suffer significant performance loss with Sh40. Therefore, to make a strong case for DC-L1-based designs, we need to address these two issues.

5.6 Clustered Shared DC-L1 Caches

We need a design that provides performance boost for the replication-sensitive applications while reducing area and energy requirements. Also, it should *not* negatively affect the replication-insensitive applications. Therefore, in this section, we investigate limiting replication instead of eliminating it. Also, we utilize the fact that smaller crossbars in NoC#1 can be operated at a higher frequency to boost performance of both replication-sensitive and replication-insensitive applications.

5.6.1 Designing Clustered Shared DC-L1 Caches

The main reason behind the NoC area and power overhead of Sh40 is the 80×40 crossbar used in NoC#1. This crossbar is essential to enable the fully-shared cache organization. Specifically, because each DC-L1 exclusively caches a non-overlapping slice of the address range, a communication path is required between any GPU core and any DC-L1 node. On the other hand, with Pr40, replication is still high, but the overall NoC area and static power is reduced. This presents a trade-off between the reduction in replication and the NoC area/power requirements. Therefore, we propose a cluster-based shared DC-L1 design where we enable the shared cache model across a cluster of DC-L1 caches instead of all of them. This eliminates replication across the DC-L1s of the same cluster, as shown in Figure 5.12. However, it still allows replication across the DC-L1s in different clusters. Using the number of clusters as a design parameter, we can limit replication while controlling the NoC area and power requirements.

With this design, a cluster of M DC-L1 nodes is accessed by N cores via an $N \times M$ crossbar in NoC#1. We refer to this design as **ShY+CZ**, where Y is the total number of DC-L1 nodes and Z is the number of clusters ($Z = Y/M$). Additionally, because of the shared nature of the L2 slices and that each DC-L1 within a cluster is assigned a unique address range, a given DC-L1 will communicate only with a few L2 slices. Therefore, instead of using a full $Y \times L$ crossbar in NoC#2 to connect the Y DC-L1 nodes to the L L2 slices ($L \geq M$, $L \bmod M = 0$), a given DC-L1 will communicate only with $O = L/M$ L2 slices via an $Z \times O$ crossbar in NoC#2. For example, Figure 5.12 shows the design of Sh40+C10 with 40 DC-L1s and 10 clusters. Each cluster consists of 8 cores accessing 4 shared DC-L1s via an 8×4 crossbar in NoC#1. The 40 DC-L1s are connected to the 32 L2 slices via four 10×8 crossbars in NoC#2. Specifically, all the 10 DC-L1s across the clusters that are assigned the same address range access the 8 L2 slices that serve such address range via a 10×8 crossbar in NoC#2. To illustrate, in Figure 5.12, the DC-L1s that serve the cross-hatched address range are connected to the L2 slices 0 to 7 (shown as

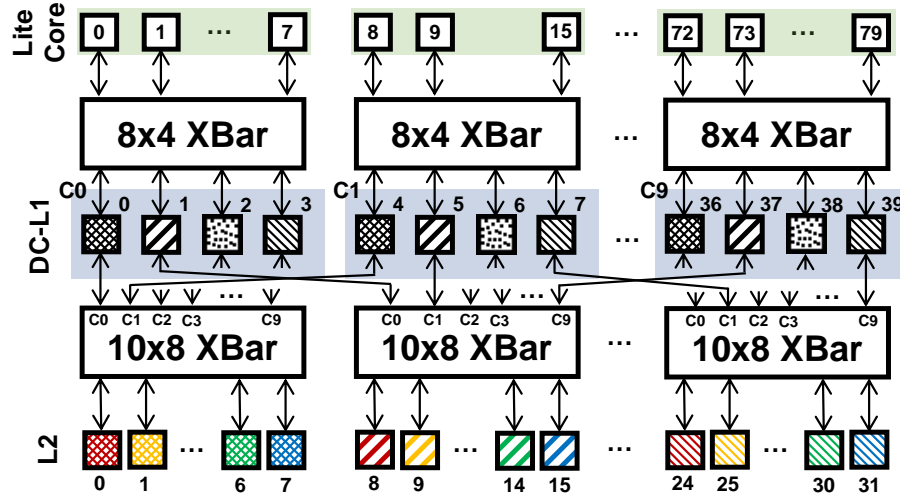


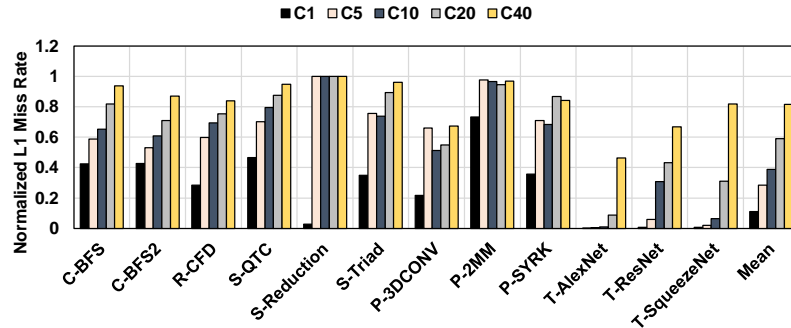
Figure 5.12: Sh40+C10 design.

different colors) that jointly serve such address range.

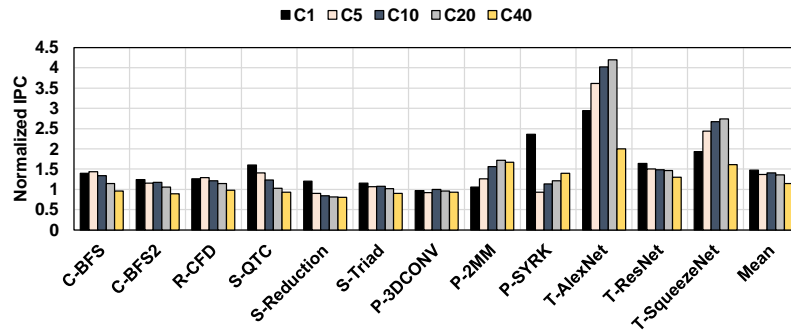
Selecting the Home DC-L1. As discussed in Section 5.5.1, the selection of the home DC-L1 is based on the home bits. The only difference is the number of bits used from the physical address of a request. Specifically, Sh Y + CZ design requires $\lceil \log_2(Y/Z) \rceil$ home bits.

5.6.2 Evaluating Clustered Shared DC-L1 Caches

Performance. In Figure 5.13, we evaluate performance of the clustered shared DC-L1 cache design on the replication-sensitive applications under different cluster counts. The results are normalized to the private L1 baseline. In this figure, $C1$ and $C40$ are equivalent to Sh40 and Pr40 designs, respectively. We make several observations. First, the L1 miss rate is higher when cluster count is more than one ($C > 1$). This is due to increased replication compared to the $C1$ case that keeps only a single copy of a given cache line across the DC-L1s. Specifically, up to 5, 10, 20, 40, and 80 copies of a cache line can exist across the DC-L1s with $C5$, $C10$, $C20$, $C40$, and baseline, respectively. This leads to an average L1 miss rate reduction (compared to baseline) of 72%, 61%, and 41% with $C5$, $C10$, and $C20$, respectively.



(a) L1 miss rate



(b) IPC

Figure 5.13: Performance of Sh40 under different cluster counts. Results are normalized to the private L1 baseline.

Second, the performance improvement is still significant under $C5$, $C10$, and $C20$ even with the smaller reduction in the L1 miss rate (compared to $C1$). For example, $C10$ improves performance by 41%, on average, over the private L1 baseline. This represents a 5% drop in performance compared to $C1$. Third, the majority of the replication-sensitive applications perform better with $C1$ because of their sensitivity to the additional effective cache capacity achieved by eliminating the replication. On the other hand, some applications (e.g., T-AlexNet) perform better with clustering. This is because the controlled replication using clustering balances the useful L1 bandwidth from the additional cache capacity and from having multiple copies (hence sources) of a given cache line. This shows that controlled data replication may not negatively affect, and can even help improve, overall performance.

Finally, P-3DCONV does not obtain speedup with the clustered design and S-Reduction

loses performance (15% drop). As discussed in Section 5.5.2, the low performance of P-3DCONV is due to its sensitivity to the reduced peak cache bandwidth with the DC-L1-based designs. As for S-Reduction, its performance improves only with the fully-shared C1 design due to its replication pattern. This is evident by the 97% drop in L1 miss rate with C1. With other clustering options, L1 miss rate does not drop, which means no reduction in replication. Thus, neither the on-chip bandwidth nor performance is boosted. On the contrary, due to the decoupled nature of the DC-L1 design and the latency intolerance of the application, we observe performance degradation.

Area & Power. We evaluate the NoC area and static power of different cluster counts, normalized to the private L1 baseline, in Figure 5.14. Similar to our observation in Section 5.4.2, breaking the 80×40 crossbar in NoC#1 with C1 (Sh40) and using smaller crossbars to form the clusters leads to savings in the NoC area and static power. Also, using smaller crossbars in NoC#2 instead of the 40×32 crossbar further improves such savings. Specifically, compared to baseline, we observe a NoC area savings of 45%, 50%, and 45% with C5, C10, and C20, respectively. As for NoC static power, we observe a reduction of 15%, 16%, and 14% with C5, C10, and C20, respectively. Given the performance improvement and area/power savings of C10, we choose this design for the rest of this work. We refer to it as **Sh40+C10**.

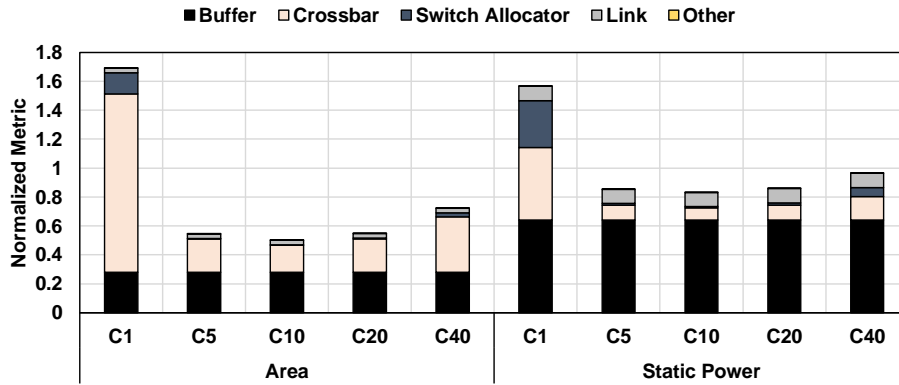


Figure 5.14: NoC area and static power under different cluster counts. Results are normalized to the private L1 baseline.

Replication-insensitive Applications. Figure 5.15 shows performance of the poor-

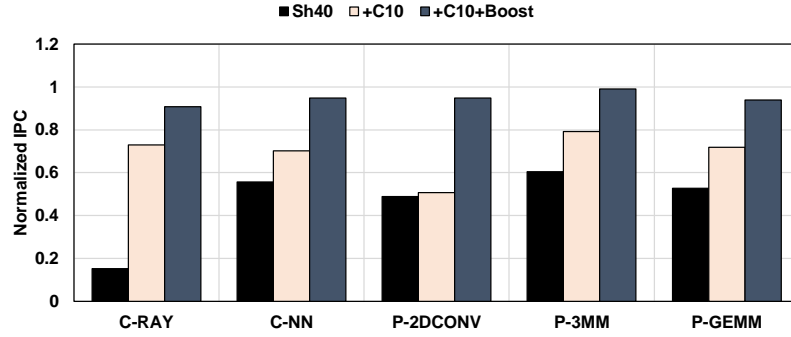


Figure 5.15: Performance of poor-performing replication-insensitive applications under Sh40+C10. Results are normalized to the private L1 baseline.

performing applications that significantly suffered with Sh40. We observe that Sh40+C10 drastically improves performance of three of these applications compared to Sh40. Specifically, C-RAY, P-3MM, and P-GEMM benefit as the effect of partition camping is lower with the clustered design. In other words, the DC-L1 contention from partition camping is relieved by having multiple home DC-L1s (ten under Sh40+C10). However, even with this improvement, performance losses in these five applications are still significant compared to the private L1 baseline with a maximum drop of 49% in P-2DCONV. Therefore, we need to further improve the performance of these applications.

5.6.3 Frequency-boosted Clustered Shared DC-L1 Design

To further boost performance of both replication-sensitive and replication-insensitive (especially poor-performing) applications, we improve the performance of NoC#1. This NoC between the GPU cores and the DC-L1 nodes is busy with request and reply traffic due to the absence of L1 caches in the GPU cores and the high hit rate of the DC-L1s under the clustered shared design. To improve performance, we utilize the fact that our Sh40+C10 uses smaller crossbars in NoC#1. This enables us to boost the frequency of these small crossbars with minimal effect on the overall NoC dynamic power (evaluated in Section 5.8). Using DSENT, we estimate the maximum operating frequency of different crossbars used in our designs in Figure 5.16. We observe the low maximum operating frequency of the

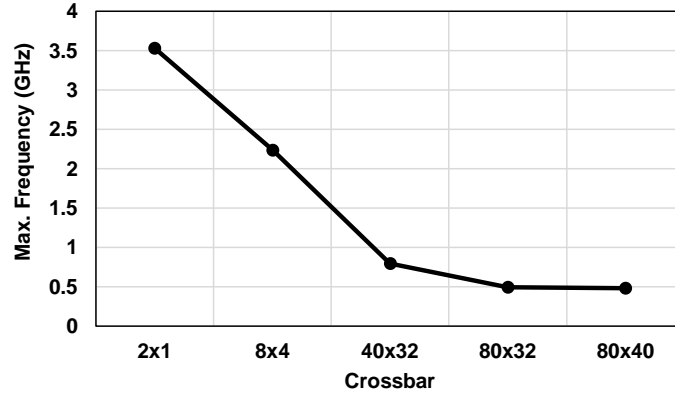


Figure 5.16: Illustrating the maximum frequency of various crossbars.

80×32 crossbar used in the baseline and the 80×40 crossbar used in Sh40. On the other hand, the small crossbars used in Pr40 (2×1) and Sh40+C10 (8×4) can operate at significantly higher frequencies. Therefore, in Sh40+C10, we double the baseline frequency of the 8×4 crossbars in NoC#1 while keeping the baseline frequency of the 10×8 crossbars in NoC#2 the same.⁷ We refer to this design as **Sh40+C10+Boost**.

From Figure 5.15, we observe that the frequency-boosted design improves performance of the poor-performing replication-insensitive applications significantly. The performance impact is particularly evident in P-2DCONV as it is sensitive to the available peak cache bandwidth (discussed in Section 5.5.2). By doubling the frequency of NoC#1, Sh40+C10+Boost partially compensates for the drop in the peak cache bandwidth due to using 40 DC-L1s (Table 5.1). Specifically, instead of enduring $8 \times$ peak cache bandwidth reduction with Sh40+C10 (compared to baseline), Sh40+C10+Boost has a $4 \times$ reduction. **Verdict.** Sh40+C10+Boost is a balanced design that limits replication to at most 10 replicas. It achieves significant performance improvements for the replication-sensitive applications while reducing the NoC area and static power. Additionally, the boosted frequency in NoC#1 recovers most of the lost performance of the poor-performing applications. We evaluate this design in Section 5.8.

⁷We do not boost NoC#2 frequency as it has less traffic due to the high hit rate of the DC-L1s. We evaluate a frequency-boosted baseline in Section 5.8.1.

5.7 Experimental Setup

Simulated System. Our *private L1 baseline* assumes a generic GPU, consisting of multiple cores that have private L1 caches. These caches are connected to multiple address-sliced L2 cache banks via a NoC. We use two separate networks (request and reply) to avoid protocol deadlocks [10]. Our baseline and proposed designs assume a separate scratchpad memory and L1 data cache. The software-managed scratchpad memory is local per-core and its performance characteristics (latency and bandwidth) are unchanged across all designs. We model our baseline and proposed designs using GPGPU-Sim v.3 cycle-level simulator [10]. Table 5.2 provides a detailed platform configuration.

Table 5.2: Configuration parameters of the simulated GPU.

Core Features	1400MHz core clock, 80 cores (CUs), SIMD width = 32 (16×2)
Resources / Core	48KB scratchpad, 32KB register file, Max. 1536 workitems (48 wavefronts, 32 workitems/wavefront)
L1 Caches / Core	16KB 4-way Write-evict L1 data cache - Latency = 28 cycles [54] 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
L2 Cache	8-way 128 KB/memory channel (4MB in total) 128B cache block size - Latency = 120 cycles
Features	Memory coalescing and inter-wavefront merging enabled, immediate post dominator based branch divergence handling
Memory Model	16 GDDR5 memory controllers (MCs) FR-FCFS scheduling, 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock, Global linear address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing [38]
Interconnect	80×32 crossbar topology, 700MHz interconnect clock, 32B flit size, 4 VCs per port, 4 flits/VC, iSLIP VC and switch allocators

Evaluated Applications. We evaluate 28 applications from five representative and diverse benchmarks suites (CUDA-SDK (C) [86], Rodinia (R) [20], SHOC (S) [26], Poly-Bench (P) [100], and Tango (T) [48]).

5.8 Experimental Results

In this section, we evaluate and compare the following against the private L1 baseline:

- **Pr40:** The proposed private DC-L1 cache design (Section 5.4) in which we reduce the number of the DC-L1 nodes to 40 while maintaining the total DC-L1 cache capacity.

- **Sh40:** The proposed fully-shared DC-L1 cache design (Section 5.5) in which we enable a shared DC-L1 cache organization to eliminate data replication across the DC-L1s.

- **Sh40+C10:** The proposed cluster-based DC-L1 cache design (Section 5.6.1) in which we apply the shared cache design across a cluster of DC-L1s to eliminate data replication within the cluster and limit replication in the GPU.

- **Sh40+C10+Boost:** The proposed frequency-boosted Sh40+C10 (Section 5.6.3) that doubles the frequency of the *small* 8×4 crossbars in NoC#1 under Sh40+C10 design.

Effect on Performance. Figure 5.17 shows performance of our proposed designs in terms of IPC normalized to the private L1 baseline. We observe the following. First, all the proposed designs improve performance of the replication-sensitive applications by varying degrees. Specifically, an improvement of 15%, 48%, 41%, and 75% is achieved under Pr40, Sh40, Sh40+C10, and Sh40+C10+Boost, respectively. Second, performance of P-3DCONV only improves under Sh40+C10+Boost (31%). This is because the cache bandwidth sensitivity of P-3DCONV (Section 5.5.2) is addressed by the frequency boost in NoC#1, which partially compensates the lost peak cache bandwidth under the DC-L1-based designs. Third, performance of P-2MM improves with Sh40+C10(+Boost) as the DC-L1 contention from partition camping is alleviated by having 10 home DC-L1s. Fourth, S-Reduction still suffers a drop in performance (14%) under Sh40+C10+Boost due to its replication pattern that can only be eliminated/reduced under the fully shared Sh40, as discussed in Section 5.6.2. For the same reason, P-SYRK achieves a lower IPC improvement of 13% with Sh40+C10+Boost compared to $2.4\times$ with Sh40. Finally, even with excluding T-AlexNet, T-ResNet, and T-SqueezeNet from the replication-sensitive applications, Sh40+C10+Boost achieves 29% improvement over the private L1 baseline. These deep-learning applications possess high data replication (Figure 5.1) and therefore highly benefit from Sh40+C10+Boost ($4.4\times$).

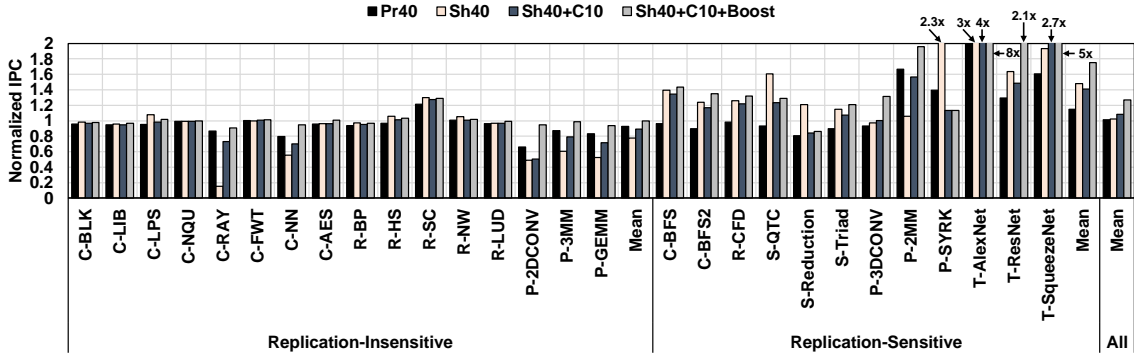


Figure 5.17: The effect of the proposed designs on IPC. Results are normalized to the private L1 baseline.

For the replication-insensitive applications, a 7%, 22%, and 11% drop in performance is incurred under Pr40, Sh40, and Sh40+C10, respectively. However, Sh40+C10+Boost maintains performance of these applications with an average IPC drop of only <1%. This is because of the frequency boost in NoC#1, which in return pushed performance of the poor-performing replication-insensitive applications (e.g., C-NN and P-2DCONV). For the remaining replication-insensitive applications, we observe a 1% performance drop under Pr40, and a 3%, 1%, and 2% improvement under Sh40, Sh40+C10, and Sh40+C10+Boost, respectively. Overall, Sh40+C10+Boost improves performance of all evaluated applications by 27%, as shown in Figure 5.17. To demonstrate that, we show the speedup of all evaluated applications sorted in ascending order under the proposed designs in Figure 5.18. This shows that Sh40+C10+Boost can provide performance benefits for the replication-sensitive applications. Also, Sh40+C10+Boost pushes the tail of the S-curve towards the private L1 baseline, thus maintaining performance of the replication-insensitive applications.

Effect on L1 Miss Rate. Figure 5.19 shows the effectiveness of our designs in reducing the DC-L1 miss rate. The results are normalized to the private L1 baseline. The reduction in the DC-L1 miss rate under Sh40+C10(+Boost) is higher compared to Pr40 and lower compared to Sh40. Such a reduction is directly proportional to the reduction in data replication. Figure 5.20 quantifies the number of replicas across the DC-L1s under the

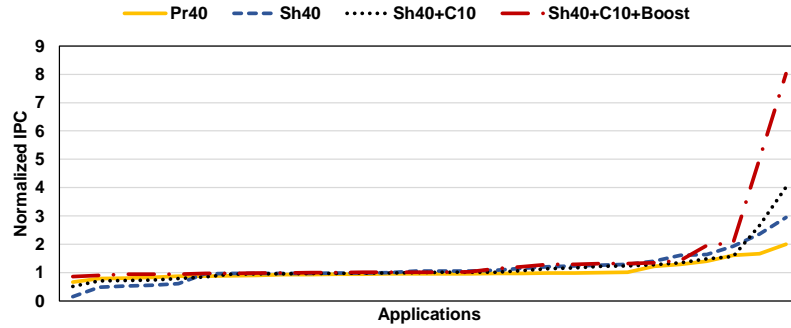


Figure 5.18: The effect of the proposed designs on IPC as S-curve. Results are normalized to the private L1 baseline.

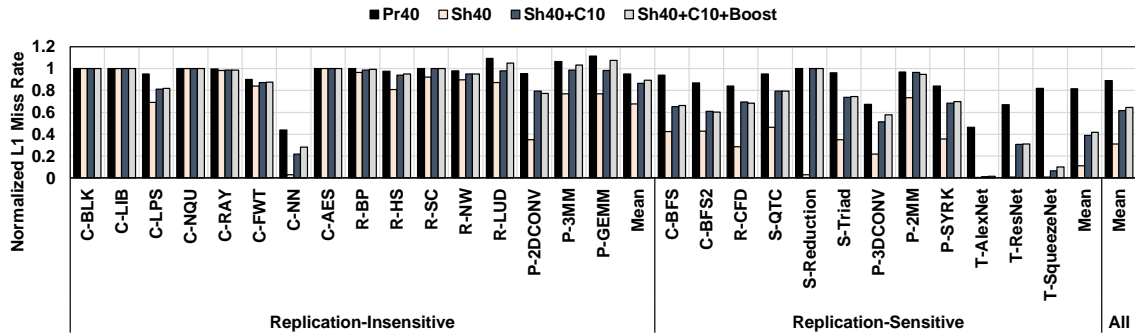


Figure 5.19: The effect of the proposed designs on L1 miss rate. Results are normalized to the private L1 baseline.

evaluated designs for the replication-sensitive applications. As expected, only a single copy of the data (i.e., zero replicas) is maintained under Sh40. However, under the private L1 baseline, each L1 can store any cache line, which may lead to more replicas across the L1s (7.7 replicas on average). Pr40 can reduce the replication compared to baseline (Section 5.4.2), which is shown by the reduction in the DC-L1 miss rate and the lower replica count (5.7 replicas on average). Sh40+C10+Boost strikes a middle ground between Pr40 and Sh40 (2.8 replicas on average).

Effect on L1 Utilization. Figure 5.21 illustrates the bandwidth utilization of the L1/DC-L1 cache data port (maximum per L1/DC-L1 accesses over the execution time) with our designs for all evaluated applications sorted in ascending order. We observe that all the proposed designs show higher DC-L1 data port utilization compared to baseline L1 data port utilization. This is from reducing the number of DC-L1 nodes (by aggre-

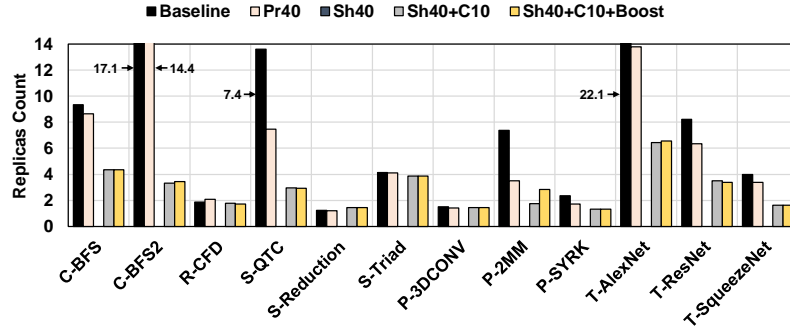


Figure 5.20: The effect of the proposed solutions on number of replicas.

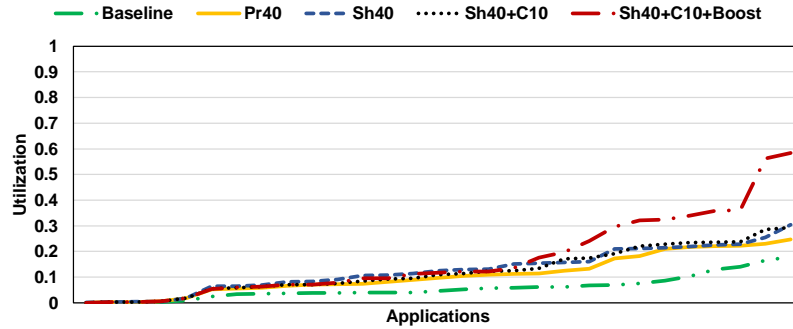


Figure 5.21: L1/DC-L1 cache data port utilization.

gating the DC-L1 caches), which leads to more requests served by each DC-L1 compared to baseline. For a comprehensive view, we also studied the utilization of NoC links that carry the data replies from the L2 to the DC-L1s and observed similar trends.

Energy Analysis. With Sh40+C10+Boost, an 8×4 crossbar (in NoC#1) connects a nearby cluster of 8 GPU cores and 4 DC-L1 nodes via short 3.3mm links. The DC-L1 nodes and the L2 slices are connected to the 10×8 crossbars (in NoC#2) via long 12.3mm links.⁸ We use DSENT to estimate the power consumption of the crossbars in both NoC#1 and NoC#2 assuming a 22nm technology. We use GPGPU-Sim to collect the flit count and NoC link activity to estimate the injection rates from the GPU cores, DC-L1 nodes, and L2 banks. We feed these estimates into DSENT to compute NoC dynamic power. Figure 5.22 shows the static, dynamic, and total NoC power breakdown for Sh40+C10+Boost normalized to the private L1 baseline. We observe the following. First,

⁸These conservative estimations are similar to prior work [147, 148].

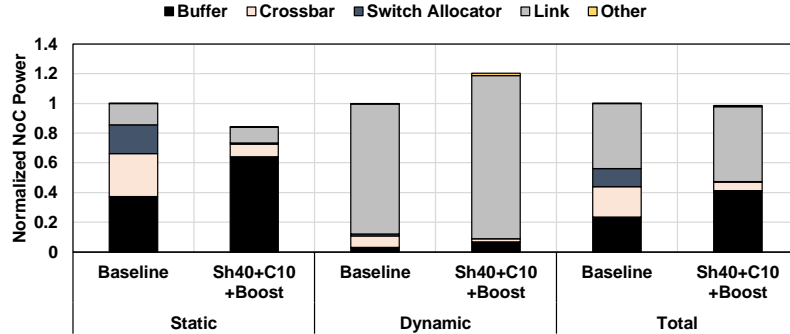


Figure 5.22: NoC power under Sh40+C10+Boost. Results are normalized to the private L1 baseline.

Sh40+C10+Boost reduces the NoC static power by 16% compared to baseline. Second, compared to baseline, the dynamic power of Sh40+C10+Boost is on average 20% higher because of the high traffic volume in NoC#1. Finally, even with the high dynamic power toll, the overall NoC power under Sh40+C10+Boost 2% lower than baseline. Given the improvement in the overall throughput and execution time, the average energy savings under Sh40+C10+Boost is 35% compared to baseline. Therefore, Sh40+C10+Boost improves performance-per-watt and energy efficiency (performance-per-energy), on average, by 29.5% and 95%, respectively.

Area Analysis. Figure 5.23 shows the area overhead/savings of Sh40+C10+Boost in terms of the queues within the DC-L1 nodes, the DC-L1 caches, and the NoC. We discussed the NoC area breakdown in Section 5.6.2 and showed that Sh40+C10 reduces NoC area requirements by 50%. The Boost optimization affects the NoC dynamic power and minimally affects the NoC area. As for the queues within the DC-L1 nodes, we use four queues (Figure 5.4) in each DC-L1 node. Each queue holds up to four 128B entries. All the queues impose an overhead of 6.25% compared to the total baseline L1 cache. This overhead is compensated by the 8% cache area savings from aggregating the DC-L1 caches in a fewer number of DC-L1 nodes. The cache area is estimated using CACTI 6.5 [80]. Even though the cache budget is maintained under Sh40+C10+Boost, we save area as we use fewer cache ports. Specifically, Sh40+C10+Boost has 50% less DC-L1 cache banks

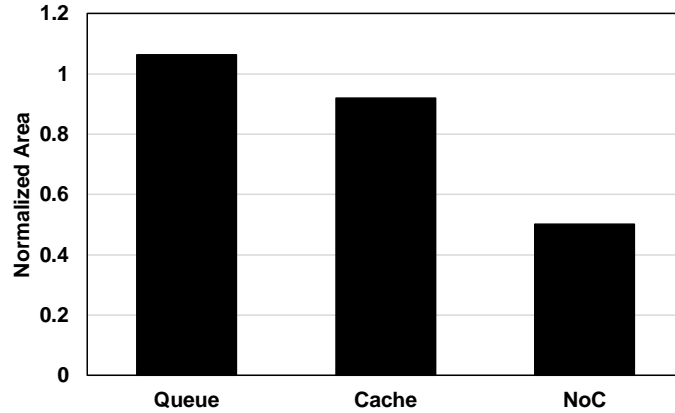


Figure 5.23: Area overhead/savings under Sh40+C10+Boost. Results are normalized to the private L1 baseline.

and hence less cache ports.

Latency Analysis. The decoupled nature of the DC-L1s imposes additional latency for the communication between the GPU cores and the DC-L1s. We estimated such latency under the evaluated applications with Sh40+C10+Boost, and observed an overhead of 54 cycles, on average. Another source of latency overhead is the aggregation of the DC-L1s. Specifically, with Sh40+C10+Boost, each DC-L1 cache is double the size of the baseline L1 cache, which adds a 7% increase in the DC-L1 access latency. Specifically, the DC-L1s with Sh40+C10+Boost have an access latency of 30 cycles, compared to 28 cycles L1 access latency in the baseline (Table 5.2). Such latency overheads do not negatively affect the evaluated applications because of the latency tolerance of the GPGPU applications. In fact, given the additional provided on-chip bandwidth from the DC-L1s with Sh40+C10+Boost, we observe a 53% reduction in the overall round trip time to fetch the required data, compared to the private L1 baseline.

5.8.1 Sensitivity Studies

Hierarchical Crossbar. Zhao *et al.* [147, 148] proposed a hierarchical two-stage crossbar design to improve the NoC scalability, area, and power. In Figure 5.24, we evaluate a hierarchical crossbar design similar to [147] (denoted as *CDXBar*) normalized to the

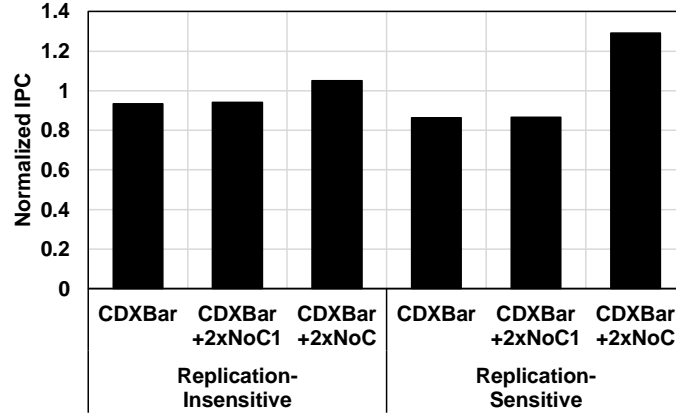


Figure 5.24: Sensitivity study on using a hierarchical crossbar.

private L1 baseline. We observe that both the replication-insensitive and the replication-sensitive applications incur performance degradation of 7% and 14% with CDXBar, respectively. This is because the main design goal of CDXBar is not performance. For a fair comparison with Sh40+C10+Boost, we study doubling the NoC frequency of the small crossbars in the first stage of CDXBar (denoted as *CDXBar+2xNoC1*) and observed a minor performance improvement (<1%) compared to CDXBar. This is because CDXBar (and *CDXBar+2xNoC1*) does not reduce data replication across the L1 caches, which puts pressure on the crossbars of the second stage of CDXBar. Hence, once we double the frequency of both stages of CDXBar (denoted as *CDXBar+2xNoC*), we observe performance improvement of 29% for the replication-sensitive applications. Such improvement is 26% lower compared to the 75% improvement under Sh40+C10+Boost. As for the replication-insensitive applications, *CDXBar+2xNoC* improves their performances by 5% compared to a slight <1% loss under Sh40+C10+Boost. However, *CDXBar+2xNoC* incurs higher dynamic NoC power overhead due to doubling the frequency of all the crossbars. In summary, Sh40+C10+Boost improves performance significantly compared to CDXBar-based designs, while achieving similar NoC area and power savings.

L1 Access Latency. In our baseline, we assume 28 cycles access latency for the L1 caches (Table 5.2). Figure 5.25 shows performance of Sh40+C10+Boost under different L1 (and DC-L1) access latency, ranging from zero to 64 cycles, normalized to its respective private

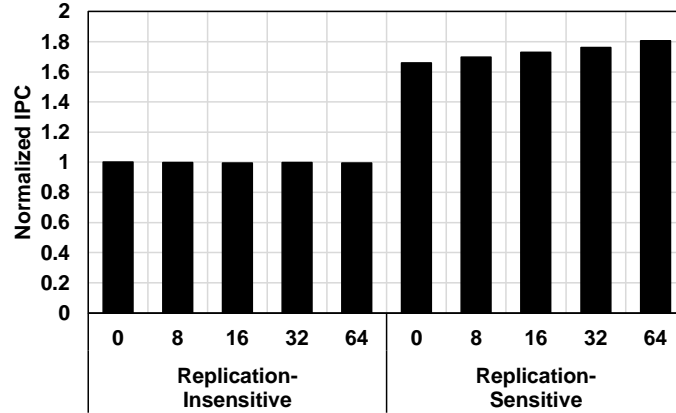


Figure 5.25: Sensitivity study on L1/DC-L1 access latency.

L1 baseline. We observe that Sh40+C10+Boost achieves a significant 66% performance improvement for the replication-sensitive applications even under zero access latency while maintaining the performance of the replication-insensitive applications (<1% drop).

CTA Scheduling. We evaluate the effect of the state-of-the-art distributed CTA scheduler [8] compared to the default round-robin CTA scheduler under Sh40+C10+Boost. Even with such scheduler, we observe 46% performance improvement for the replication-sensitive applications. The reduction in performance benefits is attributed to mapping the nearby CTAs to the same core which may reduce replication.

System Size. We study the scalability of our frequency-boosted clustered shared design (Section 5.6) by evaluating Sh60+C10+Boost under a 120-core system with 60 DC-L1s, 48 L2s, and 24 memory channels. We observe that performance follows a similar trend to the evaluated 80-core system. Specifically, we gain significant IPC improvement of 67% for the replication-sensitive applications, and maintain the private performance for the replication-insensitive applications.

Boosted Baseline. We investigate various *boosted* baselines with $2\times$ the per-core L1 cache capacity, $2\times$ the NoC frequency, and $5\times$ the flit size, respectively.⁹ For the replication-sensitive applications, we observe that these boosted baselines achieve performance improvement of 33%-36% normalized to the private L1 baseline. Such improve-

⁹The $5\times$ flit size boosted baseline delivers a given 128B read reply (or write request) in one flit.

ment is 22% lower compared to the 75% improvement under Sh40+C10+Boost. As for the replication-insensitive applications, the boosted baselines can improve their performance by 2%-6% compared to a slight <1% loss under Sh40+C10+Boost. However, these boosted baselines incurs significant area and power overheads. Specifically, using DSENT and CACTI, the cache-boosted baseline incurs a cache area overhead of 84%, and the flit-boosted baseline incurs a NoC area and static power overhead of $18.5\times$ and $4.2\times$, respectively. As for the frequency-boosted baseline, the 80×32 crossbar cannot be operated using $2\times$ the baseline frequency. Finally, our proposed designs are expected to improve performance with larger DC-L1s or boosted NoC resources.

Sectoring. We study the effect of disabling the *sectoring* optimization (Section 5.3) on the evaluated applications under Sh40+C10+Boost. This optimization aims to reduce NoC#1 bandwidth consumption by only sending the required portion of the cache line to the requester GPU core. We observe that even when disabling this NoC optimization, Sh40+C10+Boost improves performance of the replication-sensitive applications by 48% over a private L1 baseline. As for the replication-insensitive applications, on average, Sh40+C10+Boost suffers a 5% performance drop.

5.9 Related Work

To our knowledge, this is the first work to make a case for clustered shared decoupled L1 caches for GPUs. In this section, we briefly discuss works that are most relevant to this study.

Intra-core Locality in GPUs. Prior works focused on exploiting the locality that exists within a private L1 cache [105, 106, 47, 51, 73, 114]. In this work, we focus on the locality that exists across L1 caches. Other works proposed CTA schedulers [65, 8, 122] using different heuristics to exploit the locality across CTAs and improve cache performance. However, these schedulers are not ideal, and the problem of *uncontrolled* replication across L1 caches persists. Our proposed designs restrict replication to a preset limit (e.g., at

most 10 copies with Sh40+C10+Boost) and do not require any software support. In general, prior L1 cache capacity management techniques using bypassing [66, 122, 60], sectoring [104, 7], or compression [9] do not control replication across L1s. However, these works can improve performance of each individual DC-L1, while our designs facilitate coordination across DC-L1s for their better utilization.

Inter-core Locality in GPUs. Prior works focused on improving the private L1 bandwidth utilization by exploiting inter-core locality and enabling inter-core communication. This was achieved by using a ring to connect the GPU cores [28], using the L2 cache to forward inter-core traffic [146], or coherence-like mechanisms [123]. These works do not reduce replication across L1s. However, our designs reduce replication and eliminate the need for inter-core communication. Prior work [135, 24] proposed sharing an L1 data cache across a group of cores. This cache design is similar to the private DC-L1 cache design (Section 5.4) which suffers from high data replication compared to our design (Section 5.6). Specifically, Sh40+C10+Boost improves performance over such design by 52% for the replication-sensitive applications. Zhao *et al.* [148] utilized inter-core locality to address bandwidth bottlenecks at L2 by replicating cache lines across different L2 slices. This work is complementary to our work as it targets the L2 bandwidth, while ours improves the L1 capacity and its bandwidth utilization.

Replication Control in CPUs. Prior CPU works investigated the trade-offs between shared and private cache design for the last-level caches. These works proposed forms of replication management [75, 143, 22, 12, 37, 61, 128], cooperative capacity management mechanisms [17, 102, 107], hybrid shared/private designs [145, 63], coherence protocols to enable inter-core communication [36, 62], or OS-level techniques [23, 35]. Other works focused on different architectures and components [115, 13]. These works focused on latency as it is often the first-order challenge in CPU workloads. However, to our knowledge, our work is the first to propose replication control and clustered shared decoupled L1 cache design in GPUs in order to boost on-chip bandwidth.

5.10 Chapter Summary

In this work, we showed that rethinking the cache hierarchy and interconnect design in GPUs can be rewarding in terms of performance, area, and energy. Specifically, we introduced the DC-L1 cache, an L1 cache decoupled from the GPU core to address the low bandwidth utilization of the L1s and the wasted L1 cache capacity due to cache line replication across the L1 caches. We used the DC-L1s and proposed a clustered-based DC-L1 cache organization, where a cluster of GPU cores access a cluster of shared DC-L1s. With a clustered shared cache organization, we eliminated data replication within each cluster and limited the overall replication in the GPU. Our designs improve the effective L1 cache capacity, which significantly boosts on-chip bandwidth and overall performance.

Chapter 6

Conclusion and Future Research Directions

6.1 Summary of Dissertation Contributions

GPUs are designed to provide high compute throughput via high thread-level parallelism. For each generation of GPUs, the number of cores continuously grow, providing higher peak throughput. To support the continuous scaling of compute throughput, it is crucial to conserve and improve on-chip memory bandwidth utilization. The research proposed in this dissertation boosts the on-chip bandwidth via the following three contributions.

1. Unlocking Remote-core Bandwidth in GPUs. We develop probing mechanisms to efficiently unlock remote-core bandwidth – an additional source of bandwidth in GPUs. The key idea is to dynamically and locally track data replication across the cores to limit inter-core communication overhead. The proposed mechanisms achieve that by 1) leveraging the program counter (PC) information to predict which data is replicated across cores, 2) generating an inter-core locality map to predict which remote cores have the replicated data, and 3) employing a two-level probing technique to search the selected remote cores without crippling the interconnect. Our efficient inter-core communication

boosts the on-chip bandwidth and overall performance.

2. Shared L1 Caches in GPUs. We propose a renovated cache design that enables both private and shared models for L1 cache. The key idea is to dynamically detect the GPGPU applications (or phases within an application) that exhibit high data replication and utilize the shared L1 cache model for it. The shared L1 caches eliminate such replication across the L1s thus improving the L1 hit rates and boosting the on-chip bandwidth. The proposed cache design achieves that via a low-overhead dynamic scheme to configure the L1 cache as either shared or private based on locally tracking the application phases. Additionally, to efficiently utilize the additional on-chip bandwidth from shared L1 caches, we develop optimizations to reduce inter-core communication overheads. Our evaluation shows that the proposed cache design significantly reduces the data replication and improves the overall performance.

3. Aggregated Decoupled L1 Caches in GPUs. We propose co-designing the cache hierarchy and the interconnect and present DC-L1 cache – an L1 cache separated from the GPU core. The proposed cache hierarchy is still two-level with the DC-L1 as the first level of caching and shared L2 is the second level. The key idea is to break the tight coupling of the L1 caches and the GPU cores, which our analysis shows to be the main source of the L1 bandwidth underutilization. Decoupling the L1 cache from the GPU core enables aggregating the DC-L1s to reduce data replication across the L1s and increase their bandwidth utilization. Additionally, we further control data replication across DC-L1s using our frequency-boosted clustered DC-L1 design, where a cluster of DC-L1 caches are shared by a cluster of cores. Our evaluation shows that the proposed cache hierarchy and interconnect boost the on-chip bandwidth while reducing the area and energy requirements.

6.2 Future Research Directions

6.2.1 Short-term Direction: Improved On-chip Bandwidth Utilization

One line of future work continues exploring other research opportunities to improve on-chip bandwidth utilization in GPUs by considering other bandwidth sources. Specifically, there are several open research questions.

- What is the volume of data replication between the L1 and L2 caches? and how to develop techniques to address the inefficiencies caused by such replication?
- What is the volume of replication across the per-core software-managed scratchpad? and how to utilize such replication to boost their bandwidth utilization?
- How to leverage value locality and predictability to extend the scope of inter-core locality and unlock an additional source of bandwidth?

6.2.2 Long-term Direction: Graphics Pipeline & Workloads

This dissertation focuses on the general-purpose computing side of the GPUs. However, another critical and interesting aspect is investigating the graphics side of the GPUs. Graphics workloads are critical from business and research perspectives. From a business perspective, computer graphics represents huge market worth billions of dollars. It involves video games, augmented/virtual reality, content creation, and cloud gaming. From a research perspective, there is less focus on these workloads from the computer architecture community due to the inherent complexity involved in the rendering process, and lack of tools in the community. Therefore, to identify areas of improvement, it is critical to gain a low-level understanding on how these workloads work and their interaction with the graphics pipeline.

Bibliography

- [1] MOHAMMAD ABDEL-MAJEED AND MURALI ANNAVARAM. Warped Register File: A Power Efficient Register File for GPGPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.
- [2] MOHAMMAD ABDEL-MAJEED, DANIEL WONG, AND MURALI ANNAVARAM. Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.
- [3] NEHA AGARWAL, DAVID NELLANS, MIKE O’CONNOR, STEPHEN W KECKLER, AND THOMAS F WENISCH. Unlocking Bandwidth for GPUs in CC-NUMA Systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.
- [4] ASHWIN M. AJI, MAYANK DAGA, AND WU-CHUN FENG. Bounding the Effect of Partition Camping in GPU Kernels. In *Proceedings of the International Conference on Computing Frontiers (CF)*, 2011.
- [5] AMD. AMD RDNA Architecture White Paper, August 2019.
- [6] MANISH ARORA, SIDDHARTHA NATH, SUBHRA MAZUMDAR, SCOTT B BADEN, AND DEAN M TULLSEN. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro*, 2012.

- [7] A. ARUNKUMAR, S. LEE, AND C. WU. ID-Cache: Instruction and Memory Divergence Based Cache Management for GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2016.
- [8] AKHIL ARUNKUMAR, EVGENY BOLOTIN, BENJAMIN CHO, UGLJESA MILIC, EIMAN EBRAHIMI, ORESTE VILLA, AAMER JALEEL, CAROLE-JEAN WU, AND DAVID NELLANS. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.
- [9] AKHIL ARUNKUMAR, SHIN-YING LEE, VIGNESH SOUNDARARAJAN, AND CAROLE-JEAN WU. LATTE-CC: Latency Tolerance Aware Adaptive Cache Compression Management for Energy Efficient GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.
- [10] A. BAKHODA, G.L. YUAN, W.W.L. FUNG, H. WONG, AND T.M. AAMODT. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.
- [11] ALI BAKHODA, JOHN KIM, AND TOR M AAMODT. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2010.
- [12] BRADFORD M BECKMANN, MICHAEL R MARTY, AND DAVID A WOOD. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2006.
- [13] SRIKANT BHARADWAJ, GUILHERME COX, TUSHAR KRISHNA, AND ABHISHEK BHATTACHARJEE. Scalable Distributed Last-level TLBs Using Low-latency Interconnects. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018.

- [14] BENJAMIN BLOCK, PETER VIRNAU, AND TOBIAS PREIS. Multi-GPU Accelerated Multi-Spin Monte Carlo Simulations of the 2D Ising Model. *Computer Physics Communications*, 2010.
- [15] M. BURTSCHER, R. NASRE, AND K. PINGALI. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2012.
- [16] CEREBRAS. Cerebras Wafer Scale Engine, August 2019.
- [17] JICHUAN CHANG AND GURINDAR S. SOHI. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006.
- [18] N. CHATTERJEE, M. O’CONNOR, G. H. LOH, N. JAYASENA, AND R. BALASUBRAMONIAN. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.
- [19] NILADRISH CHATTERJEE, MIKE O’CONNOR, DONGHYUK LEE, DANIEL R JOHNSON, STEPHEN W KECKLER, MINSOO RHU, AND WILLIAM J DALLY. Architecting an Energy-Efficient DRAM System for GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2017.
- [20] SHUAI CHE, M. BOYER, JIAYUAN MENG, D. TARJAN, J.W. SHEAFFER, SANG-HA LEE, AND K. SKADRON. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009.
- [21] GUOYANG CHEN, BO WU, DONG LI, AND XIPENG SHEN. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

- [22] ZESHAN CHISHTI, MICHAEL D POWELL, AND TN VIJAYKUMAR. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.
- [23] SANGYEUN CHO AND LEI JIN. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2006.
- [24] KYOSHIN CHOO, WILLIAM PANLENER, AND BYUNGHYUN JANG. Understanding and Optimizing GPU Cache Memory Performance for Compute Workloads. In *Proceedings of the International Symposium on Parallel and Distributed Computing (IS-PDC)*, 2014.
- [25] A. E. COHEN AND K. K. PARHI. GPU Accelerated Elliptic Curve Cryptography in GF(2m). In *Proceedings of the International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2010.
- [26] ANTHONY DANALIS, GABRIEL MARIN, COLLIN MCCURDY, JEREMY S. MEREDITH, PHILIP C. ROTH, KYLE SPAFFORD, VINOD TIPPARAJU, AND JEFFREY S. VETTER. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the Workshop on General Purpose Processing Using GPU (GPGPU)*, 2010.
- [27] R. G. DRESLINSKI, D. FICK, B. GIRIDHAR, G. KIM, S. SEO, M. FOJTIK, S. SATPATHY, Y. LEE, D. KIM, NURRACHMAN LIU, M. WIECKOWSKI, GREGORY CHEN, T. MUDGE, D. SYLVESTER, AND D. BLAAUW. Centip3De: A 64-Core, 3D Stacked, Near-Threshold System. In *Proceedings of the Symposium on High Performance Chips (Hot Chips)*, 2012.
- [28] SAUMAY DUBLISH, VIJAY NAGARAJAN, AND NIGEL TOPHAM. Cooperative Caching for GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

- [29] ANDERS EKLUND, PAUL DUFORT, DANIEL FORSBERG, AND STEPHEN M LA-CONTE. Medical Image Processing on the GPU-Past, Present and Future. *Medical Image Analysis Journal*, 2013.
- [30] MATTHEW ELDRIDGE, HOMAN IGEHY, AND PAT HANRAHAN. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2000.
- [31] J. FARRUGIA, P. HORAIN, E. GUEHENNEUX, AND Y. ALUSSE. GPUCV: A Framework for Image Processing Acceleration with Graphics Processors. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*, 2006.
- [32] JOHANNES GILGER, JOHANNES BARNICKEL, AND ULRIKE MEYER. GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library. In *Proceedings of the International Conference on Information Security (ISC)*, 2012.
- [33] GPGPU-SIM v3.x. Address Mapping.
- [34] AYUB A. GUBRAN AND TOR M. AAMODT. Emerald: Graphics Modeling for SoC Systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.
- [35] NIKOS HARDAVELLAS, MICHAEL FERDMAN, BABAK FALSAFI, AND ANASTASIA AILAMAKI. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.
- [36] H. HOSSAIN, S. DWARKADAS, AND M. C. HUANG. Improving Support for Locality and Fine-grain Sharing in Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2008.

- [37] J. HUH, C. KIM, H. SHAFI, L. ZHANG, D. BURGER, AND S. W. KECKLER. A NUCA Substrate for Flexible CMP Cache Sharing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2007.
- [38] HYNIX. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0, 2009.
- [39] MOHAMED ASSEM IBRAHIM, ONUR KAYIRAN, YASUKO ECKERT, GABRIEL H. LOH, AND ADWAIT JOG. Analyzing and Leveraging Shared L1 Caches in GPUs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2020.
- [40] MOHAMED ASSEM IBRAHIM, ONUR KAYIRAN, YASUKO ECKERT, GABRIEL H. LOH, AND ADWAIT JOG. Analyzing and Leveraging Decoupled L1 Caches in GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2021. © 2021 IEEE. Reprinted, with permission.
- [41] MOHAMED ASSEM IBRAHIM, HONGYUAN LIU, ONUR KAYIRAN, AND ADWAIT JOG. Analyzing and Leveraging Remote-core Bandwidth for Enhanced Performance in GPUs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2019. © 2019 IEEE. Reprinted, with permission.
- [42] W. JIA, K. A. SHAW, AND M. MARTONOSI. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.
- [43] ZHE JIA, MARCO MAGGIONI, BENJAMIN STAIGER, AND DANIELE P. SCARPAZZA. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv*, April 2018.
- [44] ADWAIT JOG, EVGENY BOLOTIN, ZVIKA GUZ, MIKE PARKER, STEPHEN W. KECKLER, MAHMUT T. KANDEMIR, AND CHITA R. DAS. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In

- Proceedings of the Workshop on General Purpose Processing Using GPU (GPGPU)*, 2014.
- [45] ADWAIT JOG, ONUR KAYIRAN, TUBA KESTEN, ASHUTOSH PATTNAIK, EVGENY BOLOTIN, NILARDISH CHATTERJEE, STEVE KECKLER, MAHMUT T. KANDEMIR, AND CHITA R. DAS. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2015.
- [46] ADWAIT JOG, ONUR KAYIRAN, ASIT K. MISHRA, MAHMUT T. KANDEMIR, ONUR MUTLU, RAVISHANKAR IYER, AND CHITA R. DAS. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.
- [47] ADWAIT JOG, ONUR KAYIRAN, NACHIAPPAN C. NACHIAPPAN, ASIT K. MISHRA, MAHMUT T. KANDEMIR, ONUR MUTLU, RAVISHANKAR IYER, AND CHITA R. DAS. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
- [48] AAJNA KARKI, CHETHAN PALANGOTU KESHAVA, SPOORTHI MYSORE SHIVAKUMAR, JOSHUA SKOW, GOUTAM MADHUKESHWAR HEGDE, AND HYERAN JEON. Detailed Characterization of Deep Neural Networks on GPUs and FPGAs. In *Proceedings of the Workshop on General Purpose Processing Using GPU (GPGPU)*, 2019.
- [49] S. KAXIRAS AND J. R. GOODMAN. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

- [50] S. KAXIRAS AND C. YOUNG. Coherence Communication Prediction in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2000.
- [51] ONUR KAYIRAN, ADWAIT JOG, MAHMUT T. KANDEMIR, AND CHITA R. DAS. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2013.
- [52] ONUR KAYIRAN, ADWAIT JOG, ASHUTOSH PATTNAIK, RACHATA AUSAVARUNGNIRUN, XULONG TANG, MAHMUT T. KANDEMIR, GABRIEL H. LOH, ONUR MUTLU, AND CHITA R. DAS. μ C-States: Fine-grained GPU Datapath Power Management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.
- [53] ONUR KAYIRAN, NACHIAPPAN CHIDAMBARAM NACHIAPPAN, ADWAIT JOG, RACHATA AUSAVARUNGNIRUN, MAHMUT T. KANDEMIR, GABRIEL H. LOH, ONUR MUTLU, AND CHITA R. DAS. Managing GPU Concurrency in Heterogeneous Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.
- [54] M. KHAIRY, Z. SHEN, T. M. AAMODT, AND T. G. ROGERS. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.
- [55] KYUNG HOON KIM, RAHUL BOYAPATI, JIAYI HUANG, YUHO JIN, KI HWAN YUM, AND EUN JUNG KIM. Packet Coalescing Exploiting Data Redundancy in GPGPU Architectures. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2017.
- [56] DAVID KIRK AND W. W. HWU. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

- [57] J. KLOOSTERMAN, J. BEAUMONT, M. WOLLMAN, A. SETHIA, R. DRESLINSKI, T. MUDGE, AND S. MAHLKE. WarpPool: Sharing Requests with Inter-Warp Coalescing for Throughput Processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015.
- [58] RAKESH KOMURAVELLI, MATTHEW D. SINCLAIR, JOHNATHAN ALSOP, MUHAMMAD HUZAIFA, MARIA KOTSIFAKOU, PRAKALP SRIVASTAVA, SARITA V. ADVE, AND VIKRAM S. ADVE. Stash: Have Your Scratchpad and Cache It Too. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.
- [59] G. KOO, H. JEON, AND M. ANNAVARAM. Revealing Critical Loads and Hidden Data Locality in GPGPU Applications. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2015.
- [60] GUNJAE KOO, YUNHO OH, WON WOO RO, AND MURALI ANNAVARAM. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.
- [61] GEORGE KURIAN, SRINIVAS DEVADAS, AND OMER KHAN. Locality-Aware Data Replication in the Last-Level Cache. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.
- [62] GEORGE KURIAN, OMER KHAN, AND SRINIVAS DEVADAS. The Locality-aware Adaptive Cache Coherence Protocol. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.
- [63] WOO-CHEOL KWON, TUSHAR KRISHNA, AND LI-SHIUAN PEH. Locality-oblivious Cache Organization Leveraging Single-cycle Multi-hop NoCs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

- [64] AN-CHOW LAI AND B. FALSAFI. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.
- [65] M. LEE, S. SONG, J. MOON, J. KIM, W. SEO, Y. CHO, AND S. RYU. Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.
- [66] S. LEE AND C. WU. Ctrl-C: Instruction-Aware Control Loop Based Adaptive Cache Bypassing for GPUs. In *Proceedings of the International Conference on Computer Design (ICCD)*, 2016.
- [67] SHIN-YING LEE AND CAROLE-JEAN WU. CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014.
- [68] SHIN-YING LEE AND CAROLE-JEAN WU. Characterizing the Latency Hiding Ability of GPUs. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.
- [69] VICTOR W. LEE, CHANGKYU KIM, JATIN CHHUGANI, MICHAEL DEISHER, DAEHYUN KIM, ANTHONY D. NGUYEN, NADATHUR SATISH, MIKHAIL SMELYANSKIY, SRINIVAS CHENNUPATY, PER HAMMARLUND, RONAK SINGHAL, AND PRADEEP DUBEY. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010.
- [70] JINGWEN LENG, TAYLER HETHERINGTON, AHMED ELTANTAWY, SYED GILANI, NAM SUNG KIM, TOR M. AAMODT, AND VIJAY JANAPA REDDI. GPUWatch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

- [71] ANG LI, SHUAIWEN LEON SONG, WEIFENG LIU, XU LIU, AKASH KUMAR, AND HENK CORPORAAL. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [72] D. LI AND T. M. AAMODT. Inter-Core Locality Aware Memory Scheduling. *IEEE Computer Architecture Letters (CAL)*, 2016.
- [73] DONG LI, MINSOO RHU, DANIEL R JOHNSON, O MIKE, MATTAN EREZ, DOUG BURGER, DONALD S FUSSELL, AND STEPHEN W REDDER. Priority-Based Cache Allocation in Throughput Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.
- [74] ERIK LINDHOLM, JOHN NICKOLLS, STUART OBERMAN, AND JOHN MONTRYM. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 2008.
- [75] CHUN LIU, ANAND SIVASUBRAMANIAM, AND MAHMUT KANDEMIR. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2004.
- [76] S. A. MANAVSKI. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *Proceedings of the International Conference on Signal Processing and Communications (ICSPC)*, 2007.
- [77] M. M. K. MARTIN, P. J. HARPER, D. J. SORIN, M. D. HILL, AND D. A. WOOD. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003.

- [78] UGLJESA MILIC, ORESTE VILLA, EVGENY BOLOTIN, AKHIL ARUNKUMAR, EIMAN EBRAHIMI, AAMER JALEEL, ALEX RAMIREZ, AND DAVID NELLANS. Beyond the Socket: NUMA-Aware GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.
- [79] SHUBHENDU S. MUKHERJEE AND MARK D. HILL. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1998.
- [80] NAVEEN MURALIMANO HAR, RAJEEV BALASUBRAMONIAN, AND NORM JOUPLI. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007.
- [81] VEYNU NARASIMAN, MICHAEL SHEBANOW, CHANG JOO LEE, RUSTAM MIFTAKHUTDINOV, ONUR MUTLU, AND YALE N. PATT. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.
- [82] C. NUGTEREN, G. VAN DEN BRAAK, H. CORPORAAL, AND H. BAL. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.
- [83] NVIDIA. Computational Finance.
- [84] NVIDIA. How to Harness Big Data for Improving Public Health.
- [85] NVIDIA. What's in Your Genome? Startup Speeds DNA Analysis with GPUs.
- [86] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.
- [87] NVIDIA. NVIDIA Tesla GPUs Used by J.P. Morgan Run Risk Calculations in Minutes, Not Hours, 2011.

- [88] NVIDIA. Researchers Deploy GPUs to Build World’s Largest Artificial Neural Network, 2013.
- [89] NVIDIA. NVIDIA Tesla V100 GPU Architecture White Paper, August 2017.
- [90] NVIDIA. CUDA C++ Programming Guide, November 2019.
- [91] NVIDIA. nvprof Profiling Tool, November 2019.
- [92] NVIDIA. Parallel Thread Execution ISA Version 6.5, November 2019.
- [93] MIKE O’CONNOR, NILADRISH CHATTERJEE, DONGHYUK LEE, JOHN WILSON, ADITYA AGRAWAL, STEPHEN W KECKLER, AND WILLIAM J DALLY. Fine-grained DRAM: Energy-efficient DRAM for Extreme Bandwidth Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.
- [94] I. K. PARK, N. SINGHAL, M. H. LEE, S. CHO, AND C. KIM. Design and Performance Evaluation of Image Processing Algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2011.
- [95] SEUNG IN PARK, SEAN P PONCE, JING HUANG, YONG CAO, AND FRANCIS QUEK. Low-Cost, High-Speed Computer Vision Using NVIDIA’s CUDA Architecture. In *Proceedings of the Applied Imagery Pattern Recognition Workshop (AIPR)*, 2008.
- [96] ASHUTOSH PATTNAIK, XULONG TANG, ADWAIT JOG, ONUR KAYIRAN, ASIT K. MISHRA, MAHMUT T. KANDEMIR, ONUR MUTLU, AND CHITA R. DAS. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.
- [97] ASHUTOSH PATTNAIK, XULONG TANG, ONUR KAYIRAN, ADWAIT JOG, ASIT MISHRA, MAHMUT T. KANDEMIR, ANAND SIVASUBRAMANIAM, AND CHITA R. DAS. Opportunistic Computing in GPU Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.

- [98] PEILONG LI, YAN LUO, NING ZHANG, AND YU CAO. HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Deep Learning Algorithms. In *Proceedings of the International Conference on Networking, Architecture and Storage (NAS)*, 2015.
- [99] BHARATH PICHAI, LISA HSU, AND ABHISHEK BHATTACHARJEE. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
- [100] LOUIS-NOËL POUCHET. Polybench: The Polyhedral Benchmark Suite, 2012.
- [101] GUILLEM PRATX AND LEI XING. GPU Computing in Medical Physics: A Review. *The Journal of Medical Physics Research and Practice*, 2011.
- [102] MOINUDDIN K QURESHI. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.
- [103] X. REN, D. LUSTIG, E. BOLOTIN, A. JALEEL, O. VILLA, AND D. NELLANS. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.
- [104] MINSOO RHU, MICHAEL SULLIVAN, JINGWEN LENG, AND MATTAN EREZ. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.
- [105] TIMOTHY G. ROGERS, MIKE O’CONNOR, AND TOR M. AAMODT. Cache-Conscious Wavefront Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012.

- [106] TIMOTHY G. ROGERS, MIKE O'CONNOR, AND TOR M. AAMODT. Divergence-Aware Warp Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.
- [107] DYER ROLAN, BASILIO B FRAGUELA, AND RAMON DOALLO. Adaptive Set-Granular Cooperative Caching. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.
- [108] DIEGO ROSSINELLI, MICHAEL BERGDORF, GEORGES-HENRI COTTET, AND PETROS KOUMOUTSAKOS. GPU Accelerated Simulations of Bluff Body Flows Using Vortex Particle Methods. *Journal of Computational Physics*, 2010.
- [109] D. SANCHEZ AND C. KOZYRAKIS. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.
- [110] EDANS FLAVIUS O. SANDES AND ALBA CRISTINA M.A. DE MELO. CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.
- [111] IVY SCHMERKEN. Wall Street Accelerates Options Analysis with GPU Technology, 2009.
- [112] BERTIL SCHMIDT AND ANDREAS HILDEBRANDT. Next-generation Sequencing: Big Data Meets High Performance Computing. *Drug Discovery Today*, 2017.
- [113] ANKIT SETHIA, GANESH DASIKA, MEHRZAD SAMADI, AND SCOTT MAHLKE. APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2013.

- [114] ANKIT SETHIA AND SCOTT MAHLKE. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.
- [115] AMNA SHAHAB, MINGCAN ZHU, ARTEMIY MARGARITOV, AND BORIS GROT. Farewell My Shared LLC!: A Case for Private Die-stacked DRAM Caches for Servers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018.
- [116] INDERPREET SINGH, ARRVINDEH SHRIRAMAN, WILSON WL FUNG, MIKE O’CONNOR, AND TOR M AAMODT. Cache Coherence for GPU Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.
- [117] D. STEINKRAUS, I. BUCK, AND P. Y. SIMARD. Using GPUs for Machine Learning Algorithms. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*, 2005.
- [118] SAM S. STONE, JUSTIN P. HALDAR, STEPHANIE C. TSAO, WEN MEI W. HWU, BRADLEY P. SUTTON, AND ZHI-PEI LIANG. Accelerating Advanced MRI Reconstructions on GPUs. *The Journal of Parallel and Distributed Computing*, 2008.
- [119] C. SUN, C. H. O. CHEN, G. KURIAN, L. WEI, J. MILLER, A. AGARWAL, L. S. PEH, AND V. STOJANOVIC. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*, 2012.
- [120] NARAYANAN SUNDARAM, THOMAS BROX, AND KURT KEUTZER. Dense Point Trajectories by GPU-accelerated Large Displacement Optical Flow. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2010.

- [121] A. TABBAKH, X. QIAN, AND M. ANNAVARAM. G-TSC: Timestamp Based Coherence for GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.
- [122] ABDULAZIZ TABBAKH, MURALI ANNAVARAM, AND XUEHAI QIAN. Power Efficient Sharing-Aware GPU Data Management. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.
- [123] DAVID TARJAN AND KEVIN SKADRON. The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.
- [124] TOP500. Green500 Supercomputer Sites, June 2019.
- [125] TOP500. Top500 Supercomputer Sites, June 2019.
- [126] JELENA TRAJKOVIC, ALEXANDER V VEIDENBAUM, AND ARUN KEJARIWAL. Improving SDRAM Access Energy Efficiency for Low-power Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.
- [127] COLE TRAPNELL AND MICHAEL C. SCHATZ. Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment. *Parallel Computing*, 2009.
- [128] PO-AN TSAI, NATHAN BECKMANN, AND DANIEL SANCHEZ. Nexus: A New Approach to Replication in Distributed Shared Caches. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2017.
- [129] A. TUMEO AND O. VILLA. Accelerating DNA Analysis Applications on GPU Clusters. In *Proceedings of the Symposium on Application Specific Processors (SASP)*, 2010.

- [130] GIORGOS VASILADIS, ELIAS ATHANASOPOULOS, MICHALIS POLYCHRONAKIS, AND SOTIRIS IOANNIDIS. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.
- [131] N. VIJAYKUMAR, E. EBRAHIMI, K. HSIEH, P. B. GIBBONS, AND O. MUTLU. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.
- [132] H. WANG AND A. JOG. Exploiting Latency and Error Tolerance of GPGPU Applications for an Energy-Efficient DRAM. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2019.
- [133] HAONAN WANG, MOHAMED ASSEM IBRAHIM, SPARSH MITTAL, AND ADWAIT JOG. Address-Stride Assisted Approximate Load Value Prediction in GPUs. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2019.
- [134] HAONAN WANG, FAN LUO, MOHAMED ASSEM IBRAHIM, ONUR KAYIRAN, AND ADWAIT JOG. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.
- [135] JIANFEI WANG, LI JIANG, JING KE, XIAOYAO LIANG, AND NAIFENG JING. A Sharing-Aware L1.5D Cache for Data Reuse in GPGPUs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2019.
- [136] LU WANG, XIA ZHAO, DAVID KAEI, ZHIYING WANG, AND LIEVEN EECKHOUT. Intra-Cluster Coalescing to Reduce GPU NoC Pressure. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

- [137] BO WU, ZHIJIA ZHAO, EDDY ZHENG ZHANG, YUNLIAN JIANG, AND XIPENG SHEN. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.
- [138] WM A WULF AND SALLY A MCKEE. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 1995.
- [139] JUEKUAN YANG, YUJUAN WANG, AND YUNFEI CHEN. GPU Accelerated Molecular Dynamics Simulation of Thermal Conductivities. *Journal of Computational Physics*, 2007.
- [140] V. YOUNG, A. JALEEL, E. BOLOTIN, E. EBRAHIMI, D. NELLANS, AND O. VILLA. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.
- [141] G.L. YUAN, A. BAKHODA, AND T.M. AAMODT. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009.
- [142] J. ZHAN, O. KAYIRAN, G. H. LOH, C. R. DAS, AND Y. XIE. OSCAR: Orchestrating STT-RAM Cache Traffic for Heterogeneous CPU-GPU Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.
- [143] MICHAEL ZHANG AND KRSTE ASANOVIC. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.
- [144] TAO ZHANG, KE CHEN, CONG XU, GUANGYU SUN, TAO WANG, AND YUAN XIE. Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Re-

- thinking of Fine-grained Activation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.
- [145] LI ZHAO, RAVI IYER, MIKE UPTON, AND DON NEWELL. Towards Hybrid Last Level Caches for Chip-Multiprocessors. *ACM SIGARCH Computer Architecture News*, 2008.
- [146] X. ZHAO, Y. LIU, A. ADILEH, AND L. EECKHOUT. LA-LLC: Inter-Core Locality-Aware Last-Level Cache to Exploit Many-to-Many Traffic in GPGPUs. *IEEE Computer Architecture Letters (CAL)*, 2017.
- [147] X. ZHAO, S. MA, Z. WANG, N. E. JERGER, AND L. EECKHOUT. CD-Xbar: A Converge-Diverge Crossbar Network for High-Performance GPUs. *IEEE Transactions on Computers (TC)*, 2019.
- [148] XIA ZHAO, ALMUTAZ ADILEH, ZHIBIN YU, ZHIYING WANG, AAMER JALEEL, AND LIEVEN EECKHOUT. Adaptive Memory-Side Last-Level GPU Caching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.

VITA

Mohamed Assem Abd ElMohsen Ibrahim

Mohamed Assem Ibrahim is a Ph.D. Candidate in the Department of Computer Science at William & Mary under the supervision of Professor Adwait Jog. Mohamed's research interests lie in the broad area of computer architecture, with an emphasis on designing high-performance and energy-efficient GPU architectures. His Ph.D. research has been published at top venues: PACT 2019, PACT 2020, and HPCA 2021. Additionally, he has co-authored papers at other major computer architecture conferences such as MICRO and ICS. Mohamed worked as an intern with AMD Research in the summer of 2018 and the summer/fall of 2020. Before joining William & Mary, he received his bachelor's and master's degrees in Computer Engineering at Cairo University, Egypt.