Characterizing Multi-Chip GPU Data Sharing

SHIQING ZHANG, Ghent University, Belgium
MAHMOOD NADERAN-TAHAN, Ghent University, Belgium
MAGNUS JAHRE, Norwegian University of Science and Technology (NTNU), Norway
LIEVEN EECKHOUT, Ghent University, Belgium

Multi-chip GPU systems are critical to scale performance beyond a single GPU chip for a wide variety of important emerging applications. A key challenge for multi-chip GPUs though is how to overcome the bandwidth gap between inter-chip and intra-chip communication. Accesses to shared data, i.e., data accessed by multiple chips, pose a major performance challenge as they incur remote memory accesses possibly congesting the inter-chip links and degrading overall system performance. This paper characterizes the shared data set in multi-chip GPUs in terms of (1) truly versus falsely shared data, (2) how the shared data set scales with input size, (3) along which dimensions the shared data set scales, and (4) how sensitive the shared data set is with respect to the input's characteristics, i.e., node degree and connectivity in graph workloads. We observe significant variety in scaling behavior across workloads: some workloads feature a shared data set that scales linearly with input size, while others feature sublinear scaling (following a $\sqrt{2}$ or $\sqrt[3]{2}$ relationship). We further demonstrate how the shared data set affects the optimum last-level cache organization (memory-side versus SM-side) in multi-chip GPUs, as well as optimum memory page allocation and thread scheduling policy. Sensitivity analyses demonstrate the insights across the broad design space.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; • **Networks →** *Network on chip.*

Additional Key Words and Phrases: Graphics Processing Unit (GPU), multi-GPU systems, data sharing

ACM Reference Format:
Shiqing Zhang, Mahmood Naderan-Tahan, Magnus Jahre, and Lieven Eeckhout. xxx. Characterizing Multi-Chip GPU Data Sharing. *ACM Trans. Arch. Code Optim.* xxx, xxx, Article 1 (January xxx), 25 pages. https://doi.org/XXX

1 INTRODUCTION

The pace of technology scaling is slowing down, imposing a limit on the number of transistors available to build single-die monolithic Graphics Processing Unit (GPU) architectures [2]. At the same time, the processing and memory requirements of important GPU-compute applications such as map-reduce [9], virtual reality [10] and deep learning [15] are growing. Addressing this mismatch by increasing GPU die sizes is unattractive because lower yield results in production costs increasing super-linearly with die size [22]. The go-to strategy for enabling continued performance scaling in the post-technology-scaling era is hence to revert to multi-chip GPU architectures which provide the illusion of a single GPU while consisting of multiple GPU chips. Multi-chip GPUs

Authors' addresses: Shiqing Zhang, Ghent University, Belgium, shiqing.zhang@ugent.be; Mahmood Naderan-Tahan, Ghent University, Belgium, mahmood.naderan@ugent.be; Magnus Jahre, Norwegian University of Science and Technology (NTNU), Norway, magnus.jahre@ntnu.no; Lieven Eeckhout, Ghent University, Belgium, lieven.eeckhout@ugent.be.

1544-3566/xxx/1-ART1 $15.00
https://doi.org/XXX

are hence attractive as they provide the compute and memory resources required by emerging workloads while avoiding the production cost overhead.

Multi-chip GPU architectures can be broadly classified as (i) multi-socket GPUs where the system contains multiple GPU and memory chips that are connected with each other through the Printed Circuit Board (PCB), and (ii) Multi-Chip Module (MCM) GPUs [2, 7], where multiple GPU and memory dies, called chiplets, are interconnected using silicon interposers or organic substrates within a single package [22]. Commercial examples of the former are Nvidia's DGX systems [16, 18]; commercial examples of the latter include the Nvidia H100 [19], the AMD Instinct MI200 [1] and the Intel Ponte Vecchio [5]. In both multi-socket and chiplet-based GPU architectures, the GPUs are connected to each other via inter-chip links, and each GPU chip is connected to a local memory module. The key difference is the bandwidth available between GPU chips. MCM-GPUs offer the highest inter-chip bandwidth, but also incur the highest cost while providing limited memory capacity. In contrast, multi-socket GPUs incur lower cost and provide higher memory capacity, but offer lower inter-chip bandwidth.

A key challenge in multi-chip GPUs, both multi-socket GPUs and chiplet-based GPUs, is how to best overcome the bandwidth non-uniformity in inter-chip versus intra-chip bandwidth. Because of the bandwidth disparity, it is paramount that to unleash their high performance capabilities, multi-chip GPUs should avoid saturating the inter-chip links. This implies that most memory accesses should be local rather than remote, i.e., the Streaming Multiprocessors (SMs) in a GPU should predominantly access the local memory module, and not remote memory modules, to avoid congesting the inter-chip links. The data-parallel programming model of GPUs — in which each kernel is divided into a grid of Cooperative Thread Arrays (CTAs) with each CTA being responsible for performing the kernel's computation on a subset of the input data — is hence a good fit for multi-chip GPUs as long as the computation performed by each CTA primarily operates on a chip's local data and only rarely requires access to chip-remote data. This is mostly the case for a first-touch memory page allocation policy [2], under which the GPU driver allocates a memory page to the local memory module of the GPU chip containing the SM that first requests data from the page. Under this policy, all memory accesses are local to a chip as long as the data is *private* to that chip, i.e., no other chip is accessing the data. Because there are no remote memory accesses, only local accesses, applications with only private data are hence the perfect workloads for multi-chip GPUs as they provide unlimited scaling opportunities.

Unfortunately, not all workloads operate on private-only data. Many important GPU-compute workloads operate on data that incurs remote memory accesses, i.e., a chip is accessing data in a remote memory module. In other words, the data is *shared* by different chips. Shared data has a substantial impact on multi-chip GPU performance and scalability, as we will quantify in this paper. Intuitively speaking, if remote accesses would be equally fast as local accesses, there would be no scalability issues. It is hence critical that architects and software developers deeply understand how inter-chip data sharing affects performance to design architectures and develop software that consistently achieves high performance on multi-chip GPUs.

The goal of this work is to characterize the shared data set in multi-chip GPUs. In particular, we analyze how the shared data set varies with input size (i.e., how does the shared data set change when scaling the input size, in both absolute and relative terms?); how the shared data set scales across the various dimensions of the input (i.e., how does the shared data set scale along the x, y and z dimensions of the input?); how the shared data set varies with varying input characteristics (i.e., how does the shared data set change as properties of, for example, a graph input change in terms of the number of edges per nodes and node connectivity?). When doing so, we make a distinction between *truly shared* versus *falsely shared* data. Truly shared data means that different chips access the exact same cachelines; this leads to remote memory accesses and (possibly) camping effects

when different chips access the truly shared data around the same time to a single copy in cache. Falsely shared data on the other hand means that only a single chip is accessing the cacheline but because it is mapped to the same memory page as another cacheline accessed by at least one other chip, it leads to remote memory accesses.

We further demonstrate that the shared data set has a substantial impact on the optimum multi-GPU architecture — thereby re-enforcing the importance and relevance of understanding the shared data set in the context of multi-chip GPU systems. In particular, we compare a memory-side last-level cache (LLC) against an SM-side LLC and demonstrate that the preference of either LLC organization correlates strongly with a workload's shared data set properties and size. Because a memory-side LLC only caches data from a chip's local memory module, all accesses to remote data have to traverse the inter-chip links. An SM-side LLC on the other hand caches data from both local and remote memory modules for the local chip to access. We find that both LLC organizations perform comparably (apart from coherence overhead) if a workload only accesses private data. However, if a workload features a non-negliglible shared data set, the SM-side LLC is the best-performing organization if the shared data set is relatively small. In contrast, if the shared data set is large, the memory-side LLC is the winner. We observe a monotonic shift from the SM-side to the memory-side LLC as a function of a workload's input and shared data set size. The reason is that an SM-side LLC caches remote data locally, hence falsely shared data is cached closer to the SMs operating on the data, while truly shared data is replicated across the different chips that access it. If the shared data set is small relative to the LLC size, this turns out to be beneficial. However, if the shared data set is too large relative to the LLC's capacity, data replication of the truly shared data set initiates LLC thrashing, thereby degrading overall system performance.

Finally, we show that the impact of data sharing increases with system size (i.e., the number of chips in a multi-chip GPU system), and we find this to be the case for both strong and weak scaling scenarios. Furthermore, the scaling bottleneck increases with workload input size. This further illustrates the importance of understanding the shared data set as industry is moving forward towards larger multi-chip GPU systems.

Overall, we make the following contributions in this work:

- We characterize the shared data set in multi-chip GPUs including its relationship with a workload's input size, dimensions, and characteristics. We make a distinction between truly and falsely shared data, and quantify their scaling behavior. While the shared data set scales linearly with input size for some workloads, it follows a sublinear ($\sqrt{2}$ or $\sqrt[3]{2}$) relationship for others. We explain these scaling trends based on the dimensionality of the workloads and their inputs.
- We consider both uniform and non-uniform input scaling in which we scale all or selective input dimensions proportionally with input size, respectively. This provides insight into how input scaling affects shared data set size and performance.
- The amount of shared data is not only determined by the size of the input but also its characteristics. We provide insight into the impact of the input characteristics (node degree and connectivity) on data sharing by studying two graph workloads.
- We demonstrate that the shared data set has a substantial impact on the optimum LLC organization (memory-side versus SM-side) in multi-chip GPU architectures. The SM-side LLC is uniformly the best performing organization if the shared data set is relatively small such that replicating the truly shared data set does not surpass the LLC's capacity.
- We provide a variety of sensitivity analyses that confirm our findings across a range of inter-chip and memory bandwidth settings. We also explore how memory page allocation and CTA scheduling affect data sharing. Our analysis confirms that the state-of-the-art first-touch page

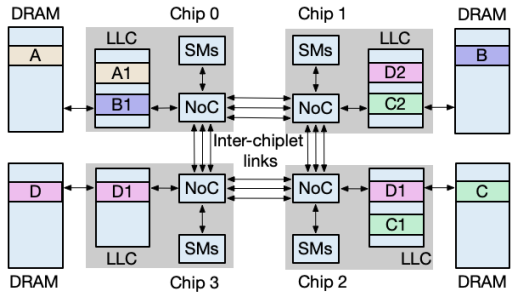


Fig. 1. Illustrating inter-chip GPU data sharing. *(1) Memory page A is allocated and accessed locally by chip 0. (2) Memory page B is allocated in chip 1 but accessed remotely only by chip 0; although cacheline B1 is not shared, it leads to remote accesses. (3) Memory page C is allocated in chip 2, and contains cacheline C1 accessed locally (by chip 2) and cacheline C2 accessed remotely by chip 1 — C1 and C2 are falsely shared. (4) Memory page D contains a truly-shared cacheline D1 accessed by both chips 2 and 3, and a falsely-shared cacheline D2 accessed remotely by chip 1.*

allocation and distributed-batched CTA scheduling policies are indeed the best performing multi-GPU policies [2]. However, we novelly correlate and explain this result by analyzing the impact these policies have on the shared data set.

2 MOTIVATION

Data sharing is common in GPU applications due to their nature of parallel processing. A thread can share data with other threads within the same thread block (i.e., CTA) or in different CTAs. When CTAs with a shared data set are allocated to different GPU chips, sharing between CTAs results in inter-chip data sharing. We now describe the different types of inter-chip data sharing, and we quantify its impact on performance.

2.1 Inter-Chip Data Sharing Classification

Data sharing across chips is the key reason why multi-chip GPU systems are sensitive to inter-chip bandwidth. Akin to the well-known concept of true versus false sharing at the cacheline granularity in CPU coherence protocols, we identify two forms of inter-chip GPU data sharing. *True sharing* occurs when different chips access the same cacheline. *False sharing* on the other hand occurs when different chips access different cachelines from the same memory page. In other words, a cache line is truly shared if it is accessed by multiple chips, and a cacheline is falsely shared if it is accessed by a single chip only, but at least one cacheline within the same memory page is accessed by another chip. *No sharing* occurs when all cachelines from a given page are accessed by the same chip; this can be a local or a remote chip.

Note that the degree of inter-chip data sharing and its impact on performance is affected by how memory pages are allocated and how CTAs are distributed across chips. More specifically, CTA scheduling and memory page allocation are key to eliminate no-sharing and reduce falsely shared data by allocating CTAs and the memory pages they access to the same chip and corresponding memory module. True sharing cannot be reduced through CTA scheduling and memory page allocation, simply because of the difference in granularity of allocation (i.e., memory page of 4 KB) versus access (i.e., cacheline of 128 B). We consider the state-of-the-art first-touch memory page allocation and distributed-batched CTA scheduling policies [2] throughout this work, unless mentioned otherwise (see also Section 8 for a detailed analysis) — these policies are the best performing ones because they eliminate no-sharing and minimize the amount of false sharing.

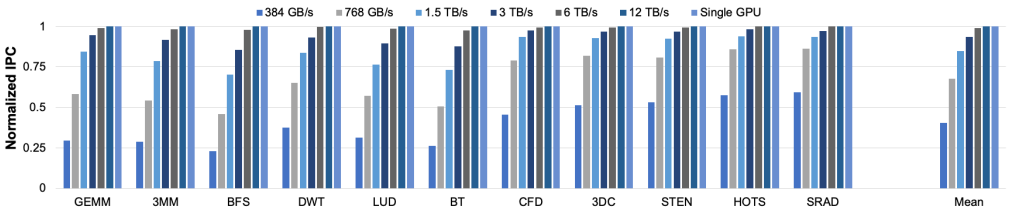


Fig. 2. Performance for 4-chip GPU as a function of the inter-chip bandwidth and in comparison to an unrealistic monolithic GPU with the same resources for our benchmarks with the 8× input. *Inter-chip data sharing significantly impacts multi-chip GPU performance when inter-chip bandwidth is limited.*

The illustrative example in Figure 1 considers four memory pages A, B, C and D, allocated in different memory partitions. Memory page A is allocated in chip-0's memory module, and only chip 0 accesses data from this memory page. In other words, this memory page is accessed locally and not shared with other chips. Memory page B is allocated in chip-1's memory module, and only remote chips access cache lines from this memory page, e.g., only chip 0 accesses data from memory page B. In other words, this page is accessed remotely. Memory page C is allocated in chip-2's memory module, and contains cachelines that are accessed locally (e.g., C1) and remotely (e.g., C2). Note that if C would have been allocated in chip 1, C1 would be remote while C2 would be local. In other words, no matter where page C is allocated, i.e., in chip 1 or 2, some accesses are local while others are remote, i.e., a case of false sharing. Lastly, memory page D is allocated in chip-3's memory partition. D1 is accessed by both chips 2 and 3, while D2 is accessed by chip 1. D1 is truly shared by chips 2 and 3, while D2 is falsely shared.

2.2 Inter-Chip Bandwidth Sensitivity

To quantify the impact data sharing has on multi-chip GPU performance, we now analyze how performance scales for a four-chip GPU system as a function of the aggregate inter-chip network bandwidth ranging from 384 GB/s to 12 TB/s, normalized to an unrealistic monolithic GPU with the same compute and memory resources (i.e., aggregate SM count, cache capacity, NoC, and memory bandwidth). We assume four chips with 64 SMs each (256 SMs in total), 4 MB of memory-side LLC per chip (16 MB in total), and a local GDDR6 memory partition with 8 channels each per chip (1.75 TB/s aggregate memory bandwidth over 32 channels). (We describe our experimental setup in detail in Section 3.)

As shown in Figure 2, multi-chip GPU performance is highly sensitive to inter-chip bandwidth. When the aggregate inter-chip bandwidth equals 384 GB/s, the average performance is only 40.3% of the unrealistic monolithic GPU. To close this performance gap, the aggregate inter-chip bandwidth needs to reach at least 6 TB/s. The high sensitivity is a result of the significant amount of inter-chip network traffic. Indeed, if all memory accesses would be local accesses, i.e., private-only data in the local memory partitions, then performance would be insensitive to inter-chip network bandwidth. In contrast, we note a sharp sensitivity to inter-chip bandwidth. The reason is the shared data set which incurs a significant amount of memory accesses to remote memory modules. The high sensitivity of multi-chip GPU performance to inter-chip bandwidth and its relation with the shared data set provides the motivation for this work to characterize multi-chip GPU data sharing and its impact on performance.

3 EXPERIMENTAL SETUP

Before diving into the analysis, we first describe our experimental setup in terms of the simulated system configuration and benchmarks.

Parameter	Value
Number of chips	4
Number of SMs	64 per chip, 256 in total
GPU frequency	1 GHz
Warp scheduler	Greedy-Then-Oldest (GTO) [21]
Inter-chip Bandwidth	768 GB/s ring, 12 bidirectional links in total 6 links per chip, 64 GB/s bidirectional per link
LLC Bandwidth	64 slices, 16 TB/s in total
DRAM Bandwidth	32 channels, 1.75 TB/s in total
L1D cache size	128 KB per SM
LLC capacity	4 MB per chip, 16 MB in total
Page placement policy	First-touch [2]
CTA allocation policy	Distributed-batched CTA scheduler [2]

Table 1. Simulated 4-chip GPU baseline configuration.

System configuration. We extend GPGPU-sim [3] to model a GPU system with four GPU chips as described in Table 1 and shown in Figure 1. Each GPU chip consists of 64 SMs and 16 LLC slices providing a per-chip LLC capacity of 4 MB; this is in line with prior work [2, 14] and recent commercial GPUs [17]. Each SM features a private L1 cache that connects with the LLC slices through a crossbar Network-on-Chip (NoC). Each chip also contains eight GDDR6 memory controllers for accessing the chip's local memory partition. We consider a ring topology for the inter-chip network in which each chip connects to two neighboring chips through six links with 64 GB/s bidirectional bandwidth each; this is similar to the second-generation NVLink [20]. The total bandwidth between two neighboring chips amounts to 192 GB/s (three links), and the total aggregate inter-chip network bandwidth amounts to 768 GB/s (twelve links in total). The end-points of the inter-chip links are connected to the NoC of each chip. To balance out memory bandwidth utilization, our baseline leverages the state-of-the-art PAE [13] randomized address mapping scheme. We consider a number of sensitivity analyses. In Section 7, we evaluate performance sensitivity to memory bandwidth (HBM2 and HBM3 with 2.3× and 6.9× higher memory bandwidth, respectively), inter-chip bandwidth (4th generation NVLink with 4× higher bandwidth), and LLC capacity (64 MB which is 4× our baseline). In Section 9, we explore the impact of system size by increasing the number of GPU chips from 4 to 8 and 16, while also varying the inter-chip network topology (fully connected versus ring).

The private L1 caches of each SM are write-through whereas the LLC follows a write-back policy. We assume software coherence as is common in GPUs [2, 14]. A software coherence protocol maintains coherence by flushing (invalidating) caches upon compiler-inserted cache control operations (e.g., at kernel boundaries). It is sufficient to flush the private L1 caches when the GPU adopts a memory-side LLC organization (our baseline) because the LLC slices only cache data from the chip's local memory partition. A data element is therefore cached in at most one LLC slice, and LLC slices are hence always coherent with respect to each other. If the GPU adopts an SM-side LLC organization on the other hand, the LLC slices must also be invalidated. The reason is that the LLC slices within a chip cache data on behalf of the chip's SMs. Shared data can hence be replicated across chips which can cause coherence violations without invalidation, e.g., if a replica is written to after a synchronization point.

Benchmarks. We select benchmarks from multiple widely-used benchmark suites, i.e., Rodinia [6], Polybench [8] and Parboil [23], as shown in Table 2. The benchmarks are selected such that we cover

Bench.	Input Set	Shared Set	Min Input	Max Input	Min Shared Set (MB)	Max Shared Set (MB)	Min CTA Count	Max CTA Count	CTA Size
3DC [8]	3D	2D (x, z)	(128, 128, 128)	(512, 512, 512)	0.5	6	32K	2M	(8, 8, 1)
STEN [23]	3D	2D (x, z)	(128, 128, 128)	(512, 512, 512)	0.5	6	32K	2M	(8, 8, 1)
CFD [6]	3D	2D (x, z)	(48, 48, 48)	(192, 192, 192)	1	13	3456	216K	(8, 4, 1)
GEMM [8]	2D	2D (x, y)	(1K, 1K, 1)	(8K, 8K, 1)	5	269	4K	256K	(32, 8, 1)
3MM [8]	2D	2D (x, y)	(768, 768, 1)	(6K, 6K, 1)	7	440	2304	144K	(32, 8, 1)
DWT [6]	2D	2D (x, y)	(768, 768, 3)	(6K, 6K, 3)	3	201	27K	1728K	(64, 1, 1)
LUD [6]	2D	2D (x, y)	(2K, 2K, 1)	(16K, 16K, 1)	9	517	1K	64K	(32, 32, 1)
SRAD [6]	2D	1D (x)	(1K, 1K, 1)	(8K, 8K, 1)	0.2	2	4K	256K	(16, 16, 1)
HOTS [6]	2D	1D (x)	(1K, 1K, 1)	(8K, 8K, 1)	0.2	2	4K	256K	(16, 16, 1)
BFS [6]	1D	1D (x)	(512K, 1, 1)	(32M, 1, 1)	3	180	1K	64K	(512, 1, 1)
BT [6]	1D	1D (x)	(256K, 1, 1)	(16M, 1, 1)	3	92	1K	64K	(256, 1, 1)

Table 2. Benchmarks used in this study along with the dimensionality of the input and truly shared data set, the minimum and maximum input set sizes, and the minimum and maximum CTA count and CTA size.

a variety in dimensionality for the input and truly shared data set. Some benchmarks, such as 3DC, STEN and CFD, feature a 3D input, while the shared data set is a 2D data structure. Others, such as GEMM, 3MM, DWT and LUD have a 2D input as well as a 2D shared data set. Two benchmarks, SRAD and HOTS, feature a 2D input while the shared data set is 1D. Finally, two benchmarks feature a 1D input with a 1D shared data set. The variety of dimensionality of input sets and shared data sets will prove to be instrumental for the remainder of the paper.

To characterize and analyze how the shared data set scales with the input, we consider a range of inputs from a minimum to a maximum size, as reported in Table 2. The minimum and maximum input size denote the scale of the problem, and depending on the specific workload, this is the number of data elements per dimension, the number of nodes in the input graph, etc. As we will discuss in more detail later, we will consider both uniform and non-uniform input scaling in which we scale all dimensions of the input uniformly or non-uniformly, as this leads to different scaling behavior. We consider the smallest (minimum) input as our baseline unless mentioned otherwise. The resulting memory footprint varies from 12 MB (for BT and its smallest input) to 1.5 GB (for SRAD and its largest input), as we will report later in Figure 4. The memory footprint is hence sufficiently large to stress the multi-GPU's memory subsystem, including on-chip caches, inter-chip network (due to remote memory partition accesses) and memory partitions. The rightmost columns in Table 2 report the minimum and maximum number of CTAs, as well as CTA size (i.e., the number of threads per CTA). The benchmarks are large enough in terms of the number of CTAs to meaningfully exercise our baseline multi-GPU system with a total of 256 SMs (four chips with 64 SMs each). We run one benchmark at a time, i.e., benchmarks execute in isolation without concurrently executing workloads.

4 SHARED DATA SET SCALING

To analyze how the shared data set scales with input size, we consider both uniform and non-uniform input set scaling, starting with uniform scaling.

4.1 Uniform Input Scaling

Recall from Section 3 that different workloads come with different input dimensionality, i.e., some workloads have a 1D input, others have a 2D input, and yet others have a 3D input. We first explore the impact of uniform scaling, i.e., scaling all dimensions of the input equally. We do so by increasing the total input size by powers of 2 from a minimum to a maximum size. For the 1D, 2D and 3D inputs, we scale each dimension by a factor of 2, $\sqrt{2}$, and $\sqrt[3]{2}$, respectively; doing so scales the total input size by a factor of 2. Uniform input scaling aligns well with what the inputs

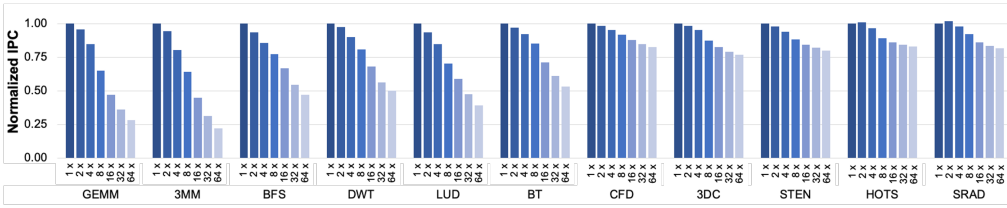
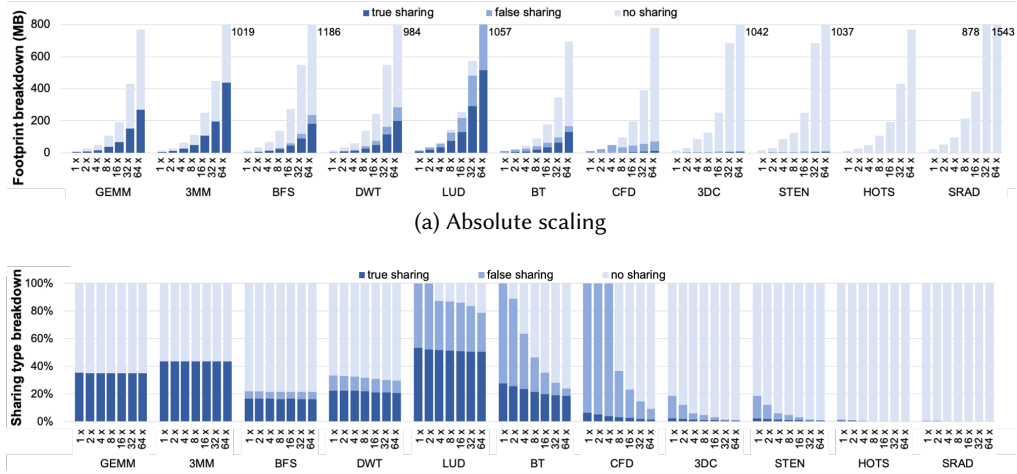


Fig. 3. Normalized performance (IPC) as a function of input size. *Performance decreases with increasing input.*



(a) Absolute scaling

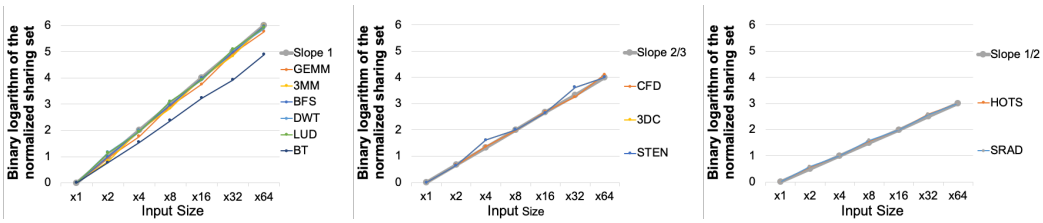
(b) Relative scaling

Fig. 4. Data footprint scaling in (a) absolute terms and (b) relative terms as a function of input size. *Both the truly and falsely shared data sets increase in absolute terms, while remaining constant or decreasing in relative terms.*

look like in existing benchmark suites, i.e., all dimensions of the input are equal in size. Note that because the number of CTAs along each dimension needs to be an integer number, we need to approximate the above general scaling. For example, for 3DC which features a 3D input, when doubling the input from the minimum $128 \times 128 \times 128$ input, the input becomes a $160 \times 160 \times 160$ matrix (i.e., $160 \approx 128 \times \sqrt[3]{2}$). Similarly, for GEMM which is a 2D-input workload, doubling the minimum 1024×1024 matrix yields a 1536×1536 matrix (i.e., $1536 \approx 1024 \times \sqrt{2}$).

Performance. Figure 3 reports performance for our baseline multi-chip GPU system as a function of input size, normalized to the smallest (1×) input. Performance is measured in number of instructions executed per cycle (IPC). Unsurprisingly perhaps, performance degrades with input size. Indeed, a larger input increases contention for on-chip cache capacity, leading to more misses and memory accesses, potentially in a remote memory partition. The benchmarks on the left-hand side are more sensitive compared to the benchmarks on the right-hand side.

Shared data set scaling. The performance trend is a result of how the shared data set scales with input size, see Figure 4. Increasing the input leads to a commensurate increase in the absolute footprint (in MB), see Figure 4a; indeed, we witness an approximate doubling in footprint as we double the input. The truly and falsely shared data set tends to increase as well in absolute terms, at least for about half the benchmarks, see the leftmost benchmarks from GEMM to BT, which are



(a) Linear scaling (b) Sublinear $\sqrt[3]{2}$ scaling (c) Sublinear $\sqrt{2}$ scaling

Fig. 5. Truly shared data set scaling trends as a function of input size. *Different workloads exhibit different scaling trends: linear, sublinear $\sqrt[3]{2}$, and sublinear $\sqrt{2}$ scaling.*

also the benchmarks for which performance degrades the most with increasing input size. Note that for some of the benchmarks on the right-hand side in Figure 4a, in particular 3DC, STEN, HOTS and SRAD, the shared data set is hardly visible. Nevertheless, as reported in Table 2, the truly shared data set varies from a couple hundreds of KBs to several MBs from the smallest to the largest input for these benchmarks. Because the LLC per chip equals 4 MB in our baseline, this also explains why these benchmarks are sensitive to relatively low inter-chip bandwidth as previously reported in Figure 2.

In relative terms though, see Figure 4b, we note different scaling trends for different benchmarks. For some benchmarks we note that the truly shared data set remains constant, see for example the leftmost benchmarks from GEMM to LUD, while it decreases for others, see the benchmarks in the middle from BT to STEN. We observe a similar trend for the falsely shared data set, with some benchmarks exhibiting a constant trend and others exhibiting a decreasing trend as a function of input size. Note though that the scaling trend for the truly and falsely shared data sets differs for some of the benchmarks, e.g., for LUD, the truly shared data set remains (roughly) constant while the falsely shared data set decreases in relative terms as a function of the input size.

The different scaling trends lead to interesting observations and important implications. In particular, a shared data set that increases proportionally with input size, while remaining constant in relative terms, suggests that the impact of shared data remains equally important irrespective of input size. In other words, no matter how big or how small the input is, the impact of shared data is similar. On the other hand, a shared data set that increases with input size in absolute terms, while decreasing in relative importance, suggests that the impact of shared data becomes less significant for bigger inputs. These findings have important implications for computer architects and software developers trying to understand how to scale GPU system and workload performance. If the shared data set scales proportionally with input size, one should be considerate of the impact of shared data. If on the other hand, the shared data set scales sub-proportionally, one can be less attentive to the shared data set and its impact on performance as we scale a workload's input.

Analyzing the shared data set. It is interesting to dive deeper and analyze shared data set scaling trends. To do so, we compute the size of the truly shared data set, normalized to the smallest data set, and visualize it in a log-log plot: Figure 5 reports the logarithm (with base 2) of the normalized shared data set size as a function of the input size on a logarithmic scale. The three subfigures denote the different scaling trends observed across the different benchmarks. It is interesting to relate these scaling trends to the dimensionality of the shared data set as previously reported in Table 2. If the dimensionality of the input set is the same as the dimensionality of the shared data set, e.g., both are 2D or both are 1D, we do note proportional scaling and a slope around one as shown in Figure 5a for the 2D-scaling benchmarks GEMM, 3MM, DWT and LUD, and the 1D-scaling

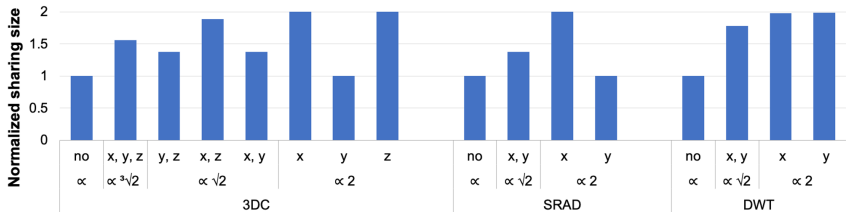


Fig. 6. Normalized shared data set size as we scale the inputs along different dimensions for 3DC, SRAD and DWT. *The shared data set increases as we scale along the sharing-sensitive input dimensions.*

benchmarks BFS and BT. If on the other hand, the dimensionality of the shared data set is smaller than the dimensionality of the input set, we observe sub-proportional scaling. If the input is 3D and the shared data set is 2D, we note a slope around $2/3$ as in Figure 5b for 3DC, STEN and CFD. This implies that the shared data set scales with a factor $\sqrt[3]{2}$ as a function of the input size. If the input is 2D while the shared data set is 1D, we note a slope around $1/2$ as in Figure 5c for SRAD and HOTS. This implies that the shared data set scales with a factor $\sqrt{2}$ as a function of the input size.

4.2 Non-Uniform Input Scaling

We now consider non-uniform input scaling, i.e., we scale the input dimensions x, y and z differently, as opposed to what we assumed in the previous section. Depending on which dimensions we scale in the input (the "Shared Set" column in Table 2 reports along which dimensions the shared data set scales), we observe different scaling behavior for the shared data set as shown in Figure 6 which reports the normalized size of the shared data set as we scale along different dimensions for three example benchmarks. (We observed similar results for the other benchmarks but these results are omitted due to space constraints.)

For 3DC, the 2D shared data set scales along the x and z dimensions. As a result, scaling the input along the x dimension (or z dimension) while keeping the other dimensions constant, i.e., 1D input scaling, leads to doubling the shared data set size. For the same reason, increasing the x *and* z dimensions with a factor $\sqrt{2}$ while keeping the y dimension constant, i.e., 2D input scaling, (almost) doubles the shared data set size. On the contrary, doubling the input along the y dimension alone does not affect the shared data set size. Increasing the input along two dimensions including the non-sharing-sensitive y dimension, i.e., scaling along the x and y dimensions or the y and z dimensions, leads to a shared data set that is roughly a factor $\sqrt{2}$ larger. For reference, uniform scaling leads to a shared data set that is a factor $\sqrt[3]{2}$ larger.

For SRAD (recall: 2D input with 1D shared data set), the sharing-sensitive input dimension is x. As a result, doubling the x dimension leads to doubling the shared data set, while doubling the y dimension does not affect the shared data set. Increasing both dimensions uniformly leads to a $\sqrt{2}$ increase in shared data set. For DWT (recall: 2D input with 2D shared data set), doubling either the x or the y dimension leads to doubling the shared data set. Increasing both dimensions uniformly by a factor $\sqrt{2}$ leads to a shared data set that is (almost) twice as big.

The significance of this analysis is that it provides insight into how the shared data set scales as a function of input scaling. This enables computer architects to understand how non-uniform input scaling affects shared data set size and performance. Or conversely, to understand how a particular architecture feature interacts with shared data, the architect could purposely compose an input that stresses the sharing degree of the input. Likewise, this analysis facilitates software developers to understand workload scaling as a function of how and along which dimensions one may expect the inputs to scale in the future; or, if appropriate, the software developer may want to reorganize

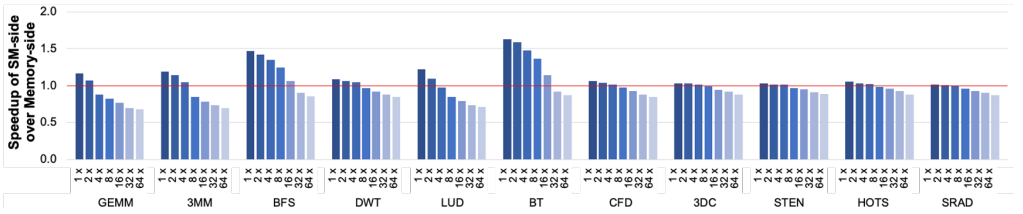


Fig. 7. Normalized performance for SM-side LLC versus memory-side LLC as a function of input size. *A memory-side LLC is the preferred organization for large inputs while a SM-side LLC is preferred for small inputs. The relative gap is larger for the benchmarks on the left-hand side due to their larger shared data sets.*

the code and/or data structure to be less sensitive to data sharing for better performance scaling to larger multi-GPU systems.

5 IMPACT ON MULTI-CHIP GPU SYSTEM ARCHITECTURE

How the shared data set scales with input size affects the optimum multi-chip GPU system architecture, which we now illustrate. We consider two multi-chip GPU architectures featuring a memory-side LLC versus an SM-side LLC [26], and we demonstrate that which architecture yields the highest performance is a function of the input's shared data set. A memory-side LLC only caches data from the local memory partition, hence all the accesses to remote data have to go through inter-chip links with limited bandwidth. Meanwhile, LLC slice conflict and queueing occur when several chips access the same shared data concurrently. The alternative configuration is an SM-side LLC, which caches data on behalf of the local chip from both the local and remote memory partitions. By caching remote data in a local LLC, SM-side LLC helps reduce remote traffic and increase LLC slice parallelism. However, remote data occupies extra cache lines in the SM-side LLC and results in more memory accesses. It also brings coherence overhead, which is not the case for a memory-side configuration.

We now explore this trade-off in more detail while considering uniform input scaling (we obtained similar results for non-uniform input scaling, omitted due to space constraints). Figure 7 reports normalized performance (or speedup) for the SM-side LLC compared to the memory-side LLC. A speedup larger than one means that the SM-side LLC is the best-performing organization, while a speedup below one means that the memory-side LLC is the winner. We find that, while the SM-side LLC is the best performing organization for the smallest inputs, the preferred LLC organization shifts towards the memory-side LLC organization for larger inputs. In fact, the memory-side LLC is uniformly the best performing organization for the largest inputs. For the smallest inputs, either the SM-side is the best performing organization or both organizations perform comparably. This can be understood intuitively as follows. As the input gets larger, the pressure on cache capacity increases, and because the data no longer fits inside the caches, more requests need to access main memory, possibly over the inter-chip network. An SM-side organization is hence preferred if the data set fits inside the cache because it incurs fewer remote memory accesses. However, as the input gets larger, a memory-side LLC reduces pressure on capacity by caching cachelines in only a single cache, and is therefore the preferred organization.

What makes the analysis interesting in the context of this paper is that the performance delta between the SM-side and memory-side LLC depends on the sharing degree: the benchmarks on the left-hand side in Figure 7 are much more sensitive to the LLC organization compared to the benchmarks on the right-hand side. This is to be explained based on the relative importance of the shared data set as previously reported in Figure 4. Indeed, the truly shared data set gets replicated in the chips' caches in an SM-side organization, i.e., all chips are able to cache the shared data set

locally. If the shared data set is frequently accessed and substantial in size but small enough to fit inside the LLC, the SM-side LLC is likely to be a clear winner compared to the memory-side LLC. This is exactly what we observe for the leftmost benchmarks, from GEMM to BT. If on the other hand, the shared data set is small (negligible) in size, the effect of the LLC organization is less pronounced and both organizations perform fairly similarly, as we observe for the rightmost benchmarks, from CFD to SRAD.

6 HOW INPUT CHARACTERISTICS AFFECT DATA SHARING

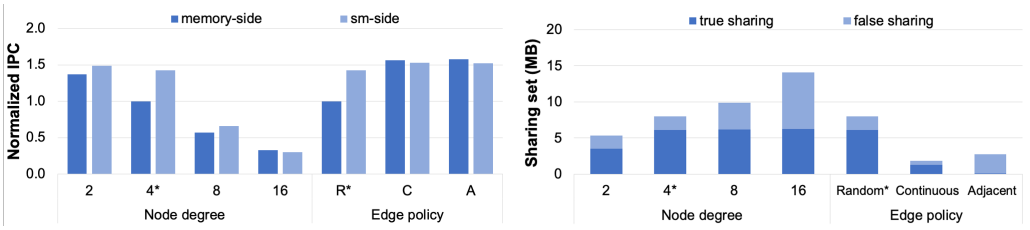
So far, we solely focused on how the shared data set scales with input size. We now investigate how, for a given input size, the input characteristics affect data sharing. Graph algorithms are a notable example of workloads where the characteristics of the input matter, in addition to its size. We now aim at gaining insight into the impact of the input characteristics on data sharing by studying two graph workloads in more detail, namely Breadth-First Search (BFS) and B+ Tree (BT).

6.1 Breadth-First Search (BFS)

The BFS benchmark used in this study is taken from Rodinia [6], and visits all nodes in the input graph following a breadth-first policy. While Rodinia provides a data set generator that creates input sets with varying number of nodes, the *node degree* or the number of edges per node, is fixed to four; and the *edge policy* or the policy for generating edges between nodes, is random, i.e., all nodes are equally likely to have an edge to any other node. To study how input graph characteristics affect data sharing, we implement support for generating input graphs with user-specified node degree and edge policy. More specifically, our input set generator supports the *continuous* and *adjacent* policies in addition to the default *random* policy. Under the continuous policy, we randomly create edges between a target node n_i and another node n_j such that the identifier j is within a range of 2.5% of the total number of nodes relative to i (i.e., j is selected within a range of 12.5 K node identifiers around i with our node count of 500 K). The adjacent policy arranges the nodes in a two-dimensional matrix according to their identifiers and then only considers four of its neighbors when randomly creating edges. More specifically, we create an $n \times n$ matrix, allocate threads row-wise to the matrix, and then consider the four threads $(i \pm 1, j)$ and $(i, j \pm 1)$ as candidates for edges of the thread assigned to matrix element (i, j). This yields a 50/50 probability for spatially long versus spatially short edges in terms of thread identifiers since $(i \pm 1, j)$ are spatially close to (i, j), while $(i, j \pm 1)$ are spatially distant.

Sharing analysis. Each thread in BFS processes a single node, and the edges between nodes thus generate inter-thread sharing. The CTA grid of BFS is one-dimensional, and each CTA contains threads of consecutively numbered nodes. Inter-CTA sharing thus occurs when the input graph has edges between nodes in different thread identifier intervals. Assuming 32 threads per warp, threads T_0 to T_{31} will be mapped to CTA_0, and threads T_{32} to T_{63} will be mapped to CTA_1. An edge between T_0 and T_{32} hence results in inter-CTA data sharing because they are part of CTA_0 and CTA_1, respectively. In contrast, an edge between T_0 and T_1 does not result in inter-CTA sharing because both threads are part of CTA_0. Each node in BFS contains an edge list which consists of `destination_node` and `weight` pairs as well as other fields such as `visited`, `cost` and `mask`. A node shares the `visited` field with the nodes it has edges to, and the `cost` and `mask` fields with the node that first visited it during the breadth-first traversal.

Inter-chip sharing occurs when CTAs share data and are mapped to different chips, e.g., a thread in CTA_i mapped to chip 0 shares the `visited`, `cost` or `mask` fields with a thread mapped to CTA_j on chip 1. Modifying the node degree parameter hence changes the number of nodes that each thread must access which in turn proportionally changes CTA and inter-chip sharing. Changing



(a) Memory-side and SM-side LLC performance (b) Sharing set size

Fig. 8. Performance and shared data set of BFS for input sets with the same size but different characteristics. (The default graph characteristics are shown with an asterisk. Performance results are normalized to a node degree of 4, the random edge policy, and memory-side LLC.) *Performance is inversely proportional to the amount of data sharing.*

the edge policy on the other hand affects the probability of inter-thread and inter-CTA sharing. As long as nodes mostly have edges to nodes that are spatially close in terms of CTA identifiers, inter-chip sharing is unlikely.

Sharing impact. Figure 8 shows the impact of graph characteristics on BFS performance while keeping the number of nodes constant at 512 K. The default input set of BFS has an average node degree of 4 and we additionally explore node degrees of 2, 8 and 16. We further explore the impact of the edge policy by changing it from the default *random* to the *continuous* and *adjacent* policies. Figure 8a reports IPC as a function of node degree and edge policy for the memory-side and SM-side LLC organizations normalized to the default. The key take-away is that input graph characteristics, both the node degree and edge policy, have a significant impact on performance. Figure 8b show that the performance impact is explained by changes in the amount and nature of inter-chip data sharing. In particular, comparing Figure 8a to 8b shows that performance is inversely proportional to the size of the shared data set.

Figure 8a also shows that 50% higher performance is achieved with the continuous and adjacent policies compared to the default random policy. Again, the performance improvement is caused by significantly less sharing, i.e., Figure 8b shows that the size of the shared data set is reduced by 77.0% and 65.5% with the continuous and adjacent policies, respectively, compared to random. The performance for the continuous and adjacent policies is similar because they result in similar amounts of sharing. The continuous policy mainly results in true sharing because the range that nodes are selected from overlaps with nodes that are spatially close; it is hence likely that nodes have edges to the same nodes. In contrast, the adjacent policy primarily results in false sharing because nodes are likely to have edges to spatially distant nodes that are spatially close to each other and hence in the same memory page. More specifically, node (i, j) has a 25% chance of generating an edge to node $(i, j + 1)$, while node $(i + 1, j)$ is equally probable to generate an edge to $(i + 1, j + 1)$. While $(i, j + 1)$ and $(i + 1, j + 1)$ are distinct nodes, they are adjacent to each other in terms of thread identifiers and hence typically stored in the same memory page — thereby resulting in false sharing rather than true sharing. The random policy causes significantly more sharing than the continuous and adjacent policies because it does not take spatial proximity into account when generating edges. The probability for inter-chip sharing is hence proportional to the number of chips under the random edge selection policy because (i) nodes are evenly distributed across chips, and (ii) a node is equally likely to have an edge to any other node.

Impact on architecture evaluation. Figure 8a shows input graph characteristics can impact the conclusions of architectural evaluation. The performance impact can be significant. More

specifically, adopting an SM-side LLC organization with a node degree of 4 yields a speedup of 42.3% compared to a memory-side LLC because the architecture has sufficient LLC capacity to replicate the true shared data set across chips. When the node degree increases to 8 or 16, LLC capacity becomes insufficient and the SM-side configuration experiences substantial LLC thrashing. This effect is so severe with node degree 16 that the memory-side LLC yields 9.1% higher performance than the SM-side LLC.

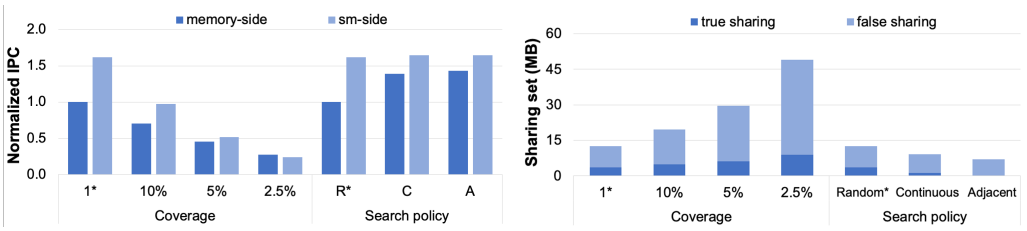
We observe a similar effect with respect to the edge policy. For the random policy, LLC capacity is sufficient to replicate the shared data set which results in SM-side improving performance by 42.3% compared to memory-side. For the continuous and adjacent policies on the other hand, there is not enough sharing for replication to yield a performance benefit. The memory-side LLC hence provides 2.7% and 4.0% higher performance than the SM-side LLC because the latter incurs coherence overhead while the memory-side LLC does not.

6.2 B+Tree (BT)

The BT benchmark used in this work is also taken from Rodinia [6], and its specific function is to carry out a fixed number of searches on an existing B+ tree. The main difference between a B+ tree and a standard B-tree is that a B+ tree stores copies of keys in internal nodes. The input set of BT hence consists of (1) a number of values to store in the B+ tree, (2) the number of searches to perform, and (3) the policy used to select the values to search for. The B+ tree built by BT contains all integers from 0 to n when instructed to generate a tree with n values; n hence determines both the number of values and the actual values to store in the tree. In contrast to BFS, *coverage* is a key parameter since BT (typically) will not search for all values in the tree. The B+ tree is self-balancing and its structure — and hence the amount of data sharing — is thus determined by the number of stored values. This sharing however only affects performance if the *search policy* searches for values that require accessing the shared data.

Sharing analysis. Data sharing occurs in BT when the search paths traversed by threads allocated to different CTAs overlap, i.e., they visit the same node(s) in the tree. The B+ tree stores ranges of values in leaf nodes, and each leaf node contains up to 255 values. The B+ tree of BT allocates a sufficient number of internal nodes such that (1) each node has between 1 and 256 child pointers, and (2) a path exists from the root node to every leaf node. All non-leaf nodes store the ranges of values that can be stored in each of its child sub-trees. Searching for a value hence starts at the root node and traverses the tree by selecting the child node which stores the range of values that the target value falls within. The search ends when reaching the leaf node with the target value.

Each thread in BT searches for a single target value, and threads share data when they access the same nodes during the search. The root node is hence shared by all threads. Similarly, threads with target values in the same leaf node will visit (and share) the same nodes. The root node and higher-level intermediate nodes are therefore shared between most threads and varying the search policy mainly affects the degree to which leaf nodes are shared. The default search policy is *random* in which the target value is a random integer between 0 and n (where n is the maximum value stored in the B+ tree). Under this policy, the search path taken by one thread has no specific relation to the paths of threads that are spatially close to it. Similar to BFS, BT has a one-dimensional grid structure, and we hence use the thread identifiers to create search policies that result in different sharing characteristics. Similar to the continuous edge policy in BFS, the *continuous* search policy randomly selects a target value close to the thread's identifier (i.e., within the range $\pm 2.5\% \times n$ where n is the maximum value stored in the tree). The *adjacent* policy allocates thread identifiers row-wise to an $n \times n$ matrix and then randomly selects the target value of T_i to be the thread identifier of one of the four vertical or horizontal neighbors of T_i.



(a) Memory-side and SM-side Performance (b) Sharing set size

Fig. 9. Performance and sharing behavior of BT for input sets with the same number of searches but different characteristics. *Performance is approximately inversely proportional to the amount of data sharing with the memory-side LLC, whereas the SM-side LLC is resilient to sharing when the shared data set (and replicas) fit in the LLC.*

Sharing impact. Figure 9 shows the impact of graph characteristics on BT performance when we keep the number of searches constant at 256K. The default input set of BT has a coverage of 100%, and we additionally explore coverage values of 10%, 5% and 2.5% by scaling the number of values stored in the tree. We further explore the impact of the search policy by changing it from the default random policy to continuous and adjacent policies. Figure 9a reports IPC as a function of coverage and search policy for the memory-side and SM-side LLC organizations normalized to our default configuration. Graph coverage has a significant performance impact, e.g., reducing coverage to 2.5% with the memory-side LLC yields 72.6% lower IPC compared to the default 100% coverage. Figure 9b shows that the performance impact of coverage is explained by changes in inter-chip data sharing. When comparing Figure 9a to Figure 9b, we note that performance as a function of coverage is approximately inversely proportional to the size of the shared data set.

Figure 9a also shows that performance increases by 39.2% and 42.8% with the continuous and adjacent policies, respectively, compared to the default random policy for the memory-side LLC organization. Interestingly, the performance of the SM-side LLC organization is practically constant. The reason is that the memory-side LLC is sensitive to both true and false sharing whereas the SM-side LLC stores recently-used shared cache blocks locally and hence avoids inter-chip accesses upon reuse. This also means that the SM-side LLC replicates truly shared cache blocks across the LLCs of the chips that access the same shared cache block — and thereby hides the performance impact of sharing when the LLCs have sufficient capacity to store the shared working set and all replicas. Figure 9b shows that the truly shared data set is relatively small for all search policies, and the cache capacity overhead of replication is hence low for this input set.

With the memory-side LLC, the higher performance with the continuous and adjacent policies is primarily due to significantly less true sharing, i.e., Figure 9b shows that the size of the truly shared data set is reduced by 65.8% and 98.3% with the continuous and adjacent policies, respectively, compared to random. The adjacent policy yields slightly (3%) better performance than the continuous policy because the continuous policy has more inter-thread true sharing. More specifically, the adjacent policy only shares data between threads that are spatial neighbors whereas the continuous policy shares data among threads that are spatially close to each other (but not necessarily neighbors). The falsely shared data set is also smaller for the adjacent and continuous policies than with the default random policy because leaf node accesses become more dense, i.e., leaf nodes are stored contiguously in memory and the leaf nodes of spatially close threads are hence mostly stored in the same memory page.

Impact on architecture evaluation. Figure 9a shows that BT's input set characteristics can impact the conclusions of architecture studies. For example, the SM-side LLC organization improves performance by 62.7% compared to the memory-side LLC organization when coverage is 100% because there is sufficient LLC capacity to cache the complete shared working set in this case. A coverage of 2.5% yields the opposite conclusion, i.e., the memory-side LLC improves performance by 13.6% compared to the SM-side LLC. The reason is that the LLCs do not have sufficient capacity to cache the shared working set which results in extensive thrashing in the SM-side configuration due to replication. With respect to the search policy, the SM-side LLC consistently outperforms the memory-side LLC because the LLCs have sufficient capacity to store the shared data set and replicas under all policies.

7 SENSITIVITY ANALYSIS: MEMORY HIERARCHY

So far, we considered a baseline configuration with a GDDR6 memory subsystem, inter-chip links modeled after NVLink generation-2, and an aggregate 16 MB LLC. We now explore the sensitivity of our findings to inter-chip bandwidth, memory bandwidth and LLC capacity. In particular, we assume an inter-chip network with NVLink generation-4 link bandwidth, i.e., 256 GB/s per link and total inter-chip bandwidth of 3 TB/s, which is 4× higher than the baseline. For the memory subsystem, we assume high-bandwidth memory HBM2 and HBM3 with 1 TB/s and 3 TB/s per module and total aggregate memory bandwidth of 4 TB/s and 12 TB/s, respectively.

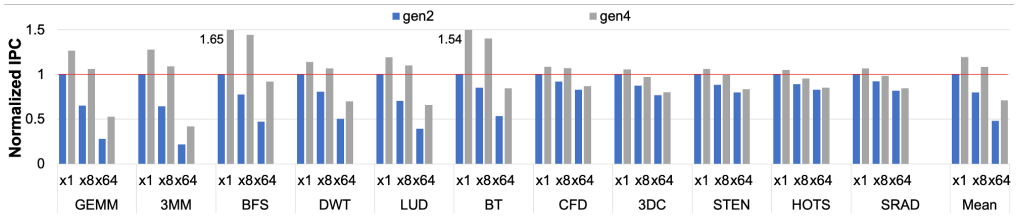
Inter-chip bandwidth. Figure 10a reports normalized performance for the various benchmarks and three inputs (1×, 8× and 64×) for the generation-2 and -4 inter-chip network configurations. Performance is normalized to the generation-2 configuration and the smallest input. The key takeaway is that we observe similar performance trends for both the generation-2 and -4 configurations, and while the trend is slightly less pronounced for the generation-4 case compared to the generation-2 case, it is still substantial. On average, performance is 41.7% lower for the largest input compared to the smallest input for the generation-4 case, while being 51.8% lower for the generation-2 case. In other words, while higher inter-chip bandwidth dampens the effect of inter-chip data sharing, it still is an important factor that has a significant impact on performance. More specifically, we note that the leftmost benchmarks, from GEMM to BT, remain highly sensitive to the inter-chip bandwidth as the input goes from the smallest to the largest. For the rightmost benchmarks, from CFD to SRAD, performance is less sensitive to inter-chip bandwidth. This is consistent with our previous conclusion from Figure 7.

Memory bandwidth. We achieve similar results when increasing memory bandwidth, see Figure 10b. While the performance trend is slightly less pronounced for HBM2 and HBM3 compared to GDDR6, it is still significant. On average, performance is 40.6% and 33.9% lower for the largest input compared to the smallest input for HBM2 and HBM3, respectively, compared to 51.8% for GDDR6. Again, while higher memory bandwidth dampens the effect, data sharing still significantly impacts performance.

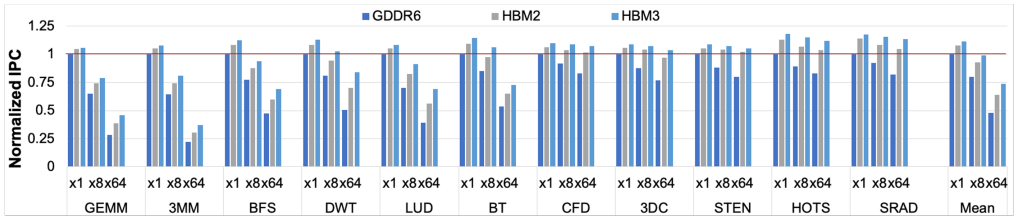
LLC capacity. Similarly, increasing the LLC capacity does not remove the impact inter-chip data sharing has on performance, see Figure 10c. When the LLC capacity increases from 16 MB to 64 MB, the performance decrease for the largest input compared to the smallest reduces from 51.8% to 35.8%, which is still significant.

8 MEMORY PAGE AND CTA ALLOCATION POLICIES

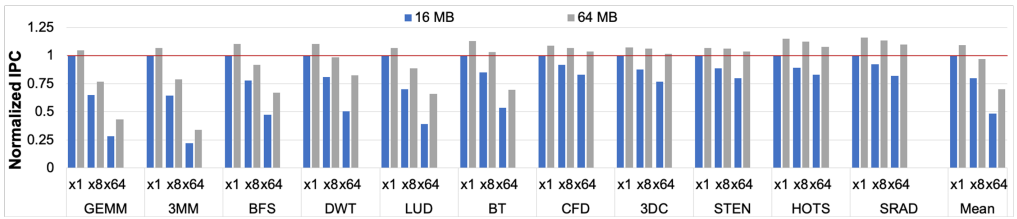
As aforementioned, memory page allocation and CTA scheduling affect shared data set characteristics. A variety of memory page allocation and CTA scheduling policies have been proposed and evaluated in the literature, see for example [11, 12, 24]. The purpose of this section is twofold: (1)



(a) Inter-chip bandwidth



(b) Memory bandwidth



(c) LLC capacity

Fig. 10. Sensitivity analysis for (a) inter-chip bandwidth, (b) memory bandwidth and (c) LLC capacity. (Performance is normalized to generation-2 inter-chip bandwidth, GDDR6, 16 MB LLC and the smallest 1× input.) *Data sharing continues to have a substantial effect on multi-chip GPU performance with higher inter-chip bandwidth, memory bandwidth, and LLC capacity.*

demonstrate that our simulated configuration (first-touch page allocation and distributed-batched CTA scheduling) is the best performing baseline, and (2) demonstrate the correlation between a policy's performance and the shared data set size, which to the best of our knowledge, no prior work reported nor analyzed.

8.1 Memory Page Allocation

We first focus on memory page allocation in this section, and discuss CTA scheduling in the next. Two commonly used page allocation policies are round-robin and first-touch [2]. Round-robin allocates memory pages to subsequent memory partitions in a round-robin fashion. First-touch on the other hand allocates memory pages to the memory partition local to the chip that first accesses (touches) the memory page. A variety of memory page allocation policies have been proposed that further enhance these two basic policies, see in particular sub-page round-robin [12], delayed first-touch [4], local-then-balanced [29], or hand-tuned API [24]. These memory page allocation policies have in common that they aim at optimizing the effective memory access bandwidth and latency (i.e., allocate data to local memory partitions), while at the same time balancing out the access distribution across all memory partitions, so that all memory partitions serve an equal share

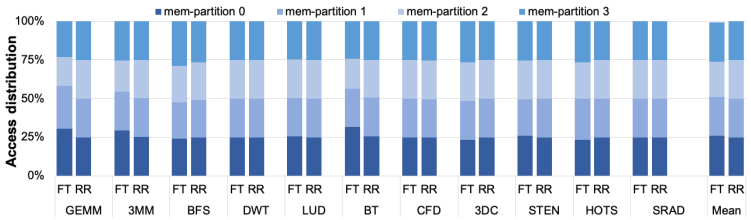
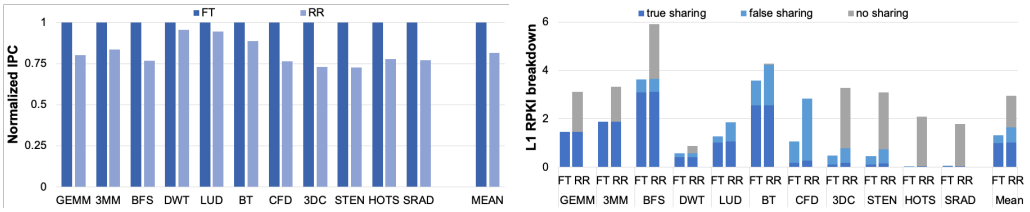


Fig. 11. Access distribution to the four memory partitions under first-touch (FT) and round-robin (RR) memory page allocation. *Memory accesses are fairly uniformly distributed under first-touch page allocation.*



(a) Performance for RR normalized to FT (b) RPKI breakdown

Fig. 12. (a) Normalized performance and (b) the number of remote memory accesses per thousand instructions (RPKI) for round-robin (RR) relative to first-touch (FT) page allocation (assuming distributed-batched CTA scheduling). *Round-robin page allocation leads to a higher sharing degree, which, because of the larger number of remote memory accesses, leads to lower performance compared to first-touch.*

of the total memory access stream. Unfortunately, these enhanced memory page allocation policies either require additional hardware support [4, 12, 29] or software modifications [24], which is undesirable in practice and which falls beyond the scope of this paper. Moreover, we find that the access distribution under first-touch page allocation delivers high performance while achieving a balanced access distribution across all memory partitions for our set of benchmarks, see Figure 11. We hence limit our evaluation to the two fundamental memory page allocation policies, namely first-touch and round-robin.

First-touch (FT) page allocation significantly outperforms round-robin (RR), see Figure 12a. The reason is the reduced number of remote memory accesses, i.e., a larger fraction of the memory accesses are local under first-touch compared to round-robin. Indeed, Figure 12b reports the number of remote memory accesses (i.e., L1 misses that lead to a remote memory access) per thousand instructions (RPKI) for the first-touch and round-robin page allocation policies. The number of remote memory accesses correlates inversely with the relative performance difference between RR and FT. More precisely, the performance drop under RR compared to FT is the largest for the benchmarks with a significant number of non-shared remote accesses under round-robin (see GEMM, 3MM, BFS, 3DC, STEN, HOTS and SRAD) or a significant increase in the number of remote accesses to falsely shared cachelines under round-robin compared to first-touch (see BT and CFD). Conversely, round-robin achieves relatively high performance for benchmarks with a small relative increase in remote accesses (see DWT and LUD).

Remote accesses incur inter-chip traffic and originate from accesses to truly shared, falsely shared, and non-shared cache lines, see Figure 12b. True sharing is frequent for several benchmarks, see for example GEMM, 3MM, BFS and BT, and is unaffected by the memory page allocation policy. False sharing constitutes an important fraction of the remote accesses for several benchmarks, and is more substantial for round-robin compared to first-touch, see in particular LUD, BT, CFD,

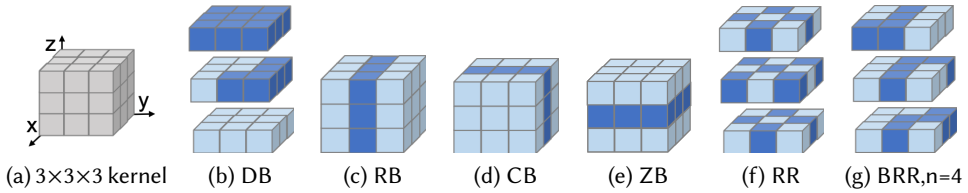


Fig. 13. CTA scheduling policies for a 3D kernel in a 2-chip GPU system. *The light blue CTAs are assigned to chip 0 and the dark blue CTAs are assigned to chip 1.*

3DC and STEN. Finally, several benchmarks suffer from non-shared cache lines allocated in remote memory partitions under round-robin, as is the case for most of the benchmarks including GEMM, 3MM, BFS, 3DC, STEN, HOTS and SRAD. Of course, first-touch lacks no-sharing.

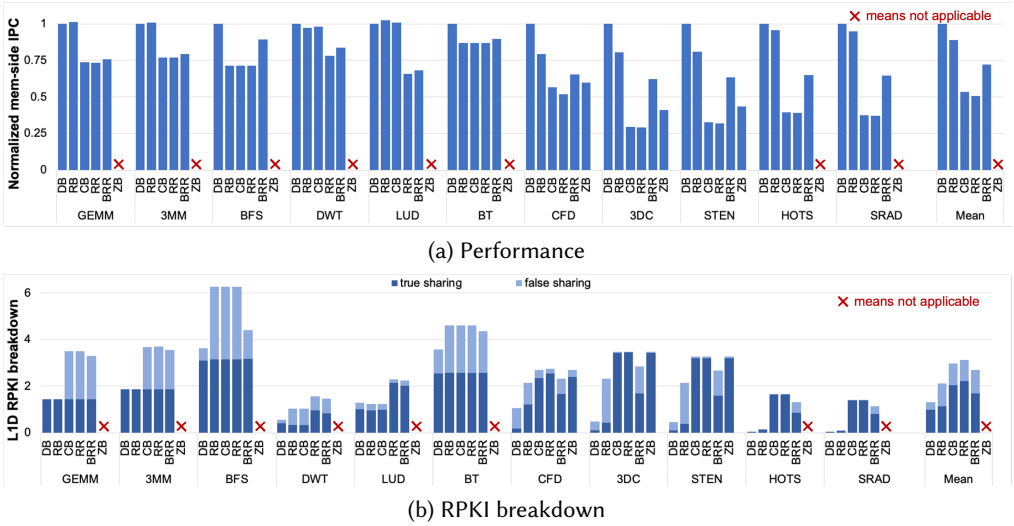
The overall conclusion from this analysis is that memory page allocation affects the degree of false sharing and no sharing, which in turn affects the number of remote accesses and overall performance. Since first-touch is the best performing memory page allocation policy, we considered it as the baseline in this paper.

8.2 CTA Scheduling

How CTAs are allocated to chips directly affects the sharing degree. To better understand how CTA scheduling affects data sharing, we evaluate the following CTA scheduling policies proposed in prior work, as illustrated in Figure 13:

- *Distributed-Batched (DB)* [2] divides the CTAs into C groups of consecutive CTAs (with C the number of chips), and assigns the CTAs of each group to the same chip. In the illustrative example shown in Figure 13b, this means that CTAs 0 through 13 (i.e., the fourteen light blue CTAs with the smallest ID) are assigned to chip 0, while CTAs 14 through 26 (i.e., the thirteen dark blue CTAs with largest ID) are assigned to chip 1.
- *Dimension-Binding* groups CTAs along a certain dimension, and then assigns CTA groups to chips in turn [11]. We consider Row-Binding (RB), Column-Binding (CB) and Z-dimension binding (ZB).[1] RB assigns CTAs to chips by row (y dimension), as illustrated in Figure 13c: CTAs in the first and third row (i.e., the eighteen light blue CTAs with the y value equal to 0 and 2) are assigned to chip 0, while CTAs in the second row (i.e., the nine dark blue CTAs with the y value equal to 1) are assigned to chip 1. Similarly, CB assigns CTAs to each chip in turn by column (x dimension), as illustrated in Figure 13d: CTAs in the first and third columns are assigned to chip 0, while CTAs in the second column are assigned to chip 1. We consider Z-dimension Binding (ZB) for the 3D benchmarks (CFD, 3DC and STEN) by assigning CTAs to chips in turn along the z dimension, as illustrated in Figure 13e.
- *Round-Robin (RR)* [2] allocates CTAs to chips in turn sequentially, as illustrated in Figure 13f: even-numbered CTAs (in light blue) are assigned to chip 0, and odd-numbered CTAs (in dark blue) are assigned to chip 1.
- *Batched Round-Robin (BRR)* [12, 24] first groups CTAs in batches of n ($n = 4$ in our setup) consecutive CTAs, and then allocates these batches to chips in a round-robin fashion, as illustrated in Figure 13g.

[1]While the illustrative $3 \times 3 \times 3$ example in Figure 13 might suggest that dimension-binding introduces a potential load imbalance issue, we find this not to be the case for the benchmarks considered in this paper with many more CTAs along the various dimensions.



(a) Performance

(b) RPKI breakdown

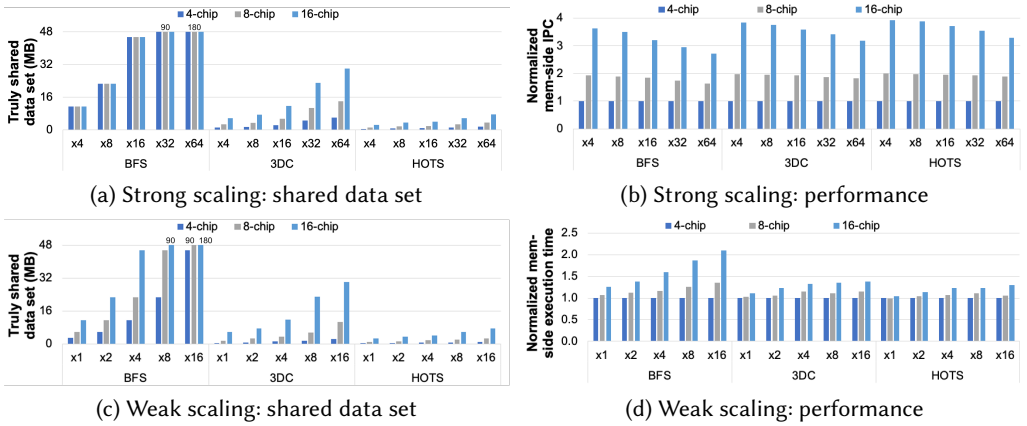
Fig. 14. CTA scheduling policy evaluation: (a) performance normalized to DB, and (b) RPKI breakdown. *DB is the best performing CTA scheduling policy because it minimizes inter-chip data sharing.*

Figure 14a reports how CTA scheduling affects performance assuming first-touch page allocation. DB is the best performing policy: the other CTA scheduling policies RB, CB, RR, and BRR yield 11%, 47%, 50% and 28% less performance than DB, respectively. For the 3D benchmarks, ZB yields 52% lower performance compared to DB. It is worth correlating these performance figures against the number of remote memory accesses (i.e., L1 misses that result in a remote access) per thousand instructions or RPKI, and its breakdown as shown in Figure 14b. The interesting observation here is that performance across the various CTA scheduling policies correlates inversely with the number of remote memory accesses, i.e., the lower the RPKI, the higher performance. The breakdown of RPKI indicates both the true and false sharing degrees. For most of the benchmarks we note a varying degree of true sharing behavior across the different CTA scheduling policies, see for example DWT, LUD, CFD, 3DC, STEN, HOTS and SRAD. Distributed-batched scheduling (DB) yields the lowest true sharing degree while round-robin (RR) achieves the highest. Performance correlates inversely with the true sharing degree because true sharing leads to inter-chip accesses and queueing delays in front of the LLC slices if multiple chips access truly shared cache lines around the same time. We note an inverse correlation between the degree of false sharing and performance due to increased inter-chip link traffic, see for example GEMM, 3MM, BFS and BT.

The overall conclusion is that distributed-batched scheduling (DB) is the best performing CTA scheduling policy, as previously reported [2], however, we novelly report how CTA scheduler performance (negatively) correlates with the sharing degree. Because DB is the best performance CTA scheduling policy, we considered it as our default policy.

9 SENSITIVITY ANALYSIS: SYSTEM SIZE

Having explored the shared data set for our baseline system with four GPUs, we now evaluate how the shared data set scales with system size and how it affects system performance. In addition, we evaluate how network topology affects these scaling trends. We consider system setups with 4, 8 and 16 GPUs with each GPU chip configured as in our baseline. We consider three benchmarks,



(a) Strong scaling: shared data set

(b) Strong scaling: performance

(c) Weak scaling: shared data set

(d) Weak scaling: performance

Fig. 15. Truly-shared data set size and normalized performance as we scale the number of chips. The top row assumes strong scaling while the bottom assumes weak scaling. *The shared data set increases with system size and input size, and so does its impact on performance, for both strong and weak-scaling workloads.*

one per truly shared data set scaling category following the analysis in Figure 5, i.e., BFS (linear scaling), 3DC ($\sqrt[3]{2}$ scaling) and HOTS ($\sqrt{2}$ scaling).

Chip count scaling. To evaluate how system size affects the shared data set and its impact on performance, we consider both strong scaling and weak scaling. Strong scaling means that we keep the workload constant while scaling the system. Weak scaling means that we scale the workload (and its problem size) with the size of the system. We consider five input sets for both strong and weak scaling. We use the ×4 to ×64 inputs under strong scaling, and keep the input constant as we scale system size. Under weak scaling, we consider the ×1 to ×16 inputs for the smallest system size with 4 GPUs, and then increase the input as we increase system size. For example, we use the ×1 input for 4 GPUs, the ×2 input for 8 GPUs, and the ×4 input for 16 GPUs; we repeat this process for all inputs up to using the ×16 input for 4 GPUs, the ×32 input for 8 GPUs, and the ×64 input for 16 GPUs. (Because the amount of work (number of instructions executed) does not precisely double when doubling the input, we rescale the performance numbers under weak scaling such that the amount of work is proportional to system size.)

Several interesting observations can be made. Under strong scaling, the truly shared data set size remains (nearly) constant for BFS, while increasing (almost) proportionally with system size for 3DC and HOTS, see Figure 15a. The proportional increase for 3DC and HOTS is a result of the work being distributed across more chips. If the mapping of CTAs to chips is done along the dimension along which sharing occurs, the shared data set increases accordingly. While the data structure accessed by multiple CTAs remains constant in size, distributing the CTAs across multiple GPU chips hence results in an increased truly shared data set. This is not the case for BFS where the shared data set remains (nearly) constant because all CTAs access the same data structure(s), hence irrespective of how CTAs are mapped to chips, this leads to a similar degree of sharing.

Under weak scaling, we note a compounding effect. First, when system size increases, so does the input and its truly shared data set (as previously reported). Second, distributing the work across more GPUs also increases the truly shared data set (as for strong scaling). The end result is that the truly shared data set increases linearly for BFS, and super-linearly for 3DC and HOTS, see Figure 15c.

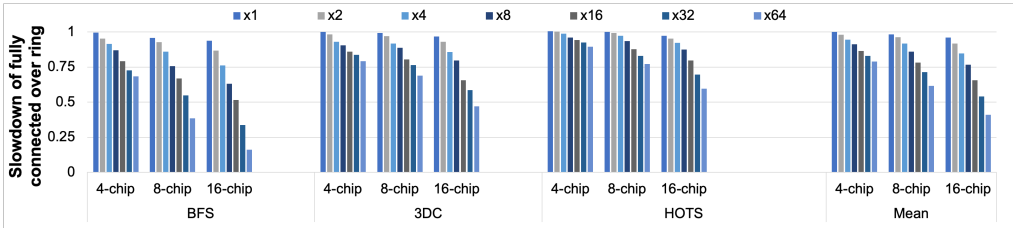


Fig. 16. Slowdown of the fully connected topology over the ring topology. *The larger the shared data set, the greater the performance degradation for the fully connected network compared to the ring.*

The impact on performance is commensurate. Under strong scaling, we observe that the impact of the shared data set increases with system size. Indeed, Figure 15b reports normalized performance, which under perfect scaling would equal 2× and 4× for the 8-chip and 16-chip systems, respectively. We note that the scaling behavior deteriorates with system size, e.g., for BFS and the ×64 input, the 8-chip system achieves 88% of its maximum speedup while the 16-chip system achieves 69% of its maximum speedup. We make a similar observation for weak scaling, see Figure 15d where we report normalized execution time. Under perfect scaling, execution time would be constant (i.e., normalized execution time equal to one) as we scale system size because the problem size scales proportionally. However, we observe that execution time increases with system size, e.g., for BFS and the ×16 input for 4 chips, execution time increases by 1.4× and 2.1× for the 8-chip and 16-chip systems, respectively. The poor scaling behavior further deteriorates with input size, for both strong and weak scaling.

The overall conclusion from this analysis is that the shared data set increases with system size, and so does its impact on performance. This is the case for both strong and weak-scaling workloads. And in addition, the performance bottleneck further worsens with a workload's input size.

Inter-chip network topology. Continuing with the three benchmarks, we now study how the inter-chip network topology affects the impact of sharing on performance. Figure 16 reports slowdown for the fully connected topology compared to the ring topology (our baseline) for BFS, 3DC and HOTS while varying system and input size. The fully connected topology achieves lower performance compared to the ring because the latter achieves a higher effective inter-chip bandwidth. Indeed, inter-chip traffic is not uniform across nodes. In fact, we observe more communication between neighboring chips. We note further that the performance gap increases with increasing system size and problem size, which is consistent with our earlier findings.

10 RELATED WORK

We quantitatively compared key CTA schedulers for multi-chip GPUs in Section 8.2, and we found that distributed-batched CTA scheduling is the top performer because it incurs the least amount of sharing; this observation is consistent with prior work [2]. The key benefit of the CTA scheduling algorithms that we evaluate in this paper is that they make their decision solely based on CTA identifiers. If it is acceptable to profile the application, change the compiler or involve the programmer, sharing can however be further reduced. For example, the Locality Descriptor [24] lets the programmer label data structures as having intra-thread locality, inter-thread locality or no-reuse, and then use this information at runtime to improve locality within a chip which in turn reduces inter-chip sharing. CODA [12] on the other hand analyzes inter-thread sharing within the compiler (to capture static sharing information) and with profiling (to capture dynamic information), and then modifies the address translation mechanism and the CTA scheduler to (as

much as possible) co-locate CTAs with the data they access. LADM [11] focuses on minimizing inter-chip traffic and uses compiler analyses to categorize data structures as having no data-block locality, DRAM row or column locality, or intra-thread locality, and then exploits these patterns to optimize CTA scheduling.

Another line of prior work focuses on making multi-chip GPUs more resilient to shared data by modifying the GPU memory system. CARVE [25] uses part of the chip's local memory to replicate remote data and thereby reduces inter-chip communication at the cost of increasing the application's memory footprint. Another option is to modify the cache structure, and MCM-GPU [2] adds an additional cache between the L1 cache and the LLC that exclusively caches remote data. In contrast, the dynamic LLC [14] allocates LLC capacity to local and remote data to optimize bandwidth beyond the LLC, while Adaptive LLC [27] and SelRep LLC [28] replicates shared data across LLC slices within a chip to optimize NoC bandwidth.

11 CONCLUSION

A workload's shared data set incurs a substantial impact on multi-GPU system performance: remote memory accesses possibly congest the low bandwidth inter-chip links and/or lead to camping effects in front of a memory-side LLC, thereby degrading overall system performance. This paper categorizes the shared data set in true versus false sharing, and quantifies how the shared data set scales with input size, along which input dimensions the shared data set scales, and which input properties affect the shared data set (i.e., node degree and connectivity for graph inputs). We further note that the optimum last-level cache organization in multi-GPU systems is a function of the shared data set. The insights provided in this paper help computer architects and software developers better understand how inter-chip data sharing affects multi-GPU performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. Shiqing Zhang is supported by a CSC scholarship (Grant No. 201903170128), and Magnus Jahre is supported by the Research Council of Norway (Grant No. 286596). Lieven Eeckhout is supported in part by the UGent-BOF-GOA grant No. 01G01421, and the European Research Council (ERC) Advanced Grant agreement No. 741097.

REFERENCES

[1] AMD. 2021. AMD Instinct MI200 Series Accelerator. https://www.amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf. [Online; accessed 2023-03-02].

[2] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 320–332.

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

[4] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A Mojumder, José L Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 596–609.

[5] David Blythe. 2021. Xehpc Ponte Vecchio. In *Proceedings of Hot Chips 33 Symposium (HCS)*. IEEE Computer Society, 1–34.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54.

[7] William J Dally, C Thomas Gray, John Poulton, Brucek Khailany, John Wilson, and Larry Dennison. 2018. Hardware-Enabled Artificial Intelligence. In *Proceedings of the International Symposium on VLSI Technology and Circuits (VLSI)*. IEEE, 3–6.

[8] John Cavazos Scott Grauer-Gray. 2015. PolyBench/GPU 1.0. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. [Online; accessed 2022-04-16].
[9] Hai Jiang, Yi Chen, Zhi Qiao, Tien-Hsiung Weng, and Kuan-Ching Li. 2015. Scaling Up MapReduce-Based Big Data Processing on Multi-GPU Systems. *Cluster Computing* 18 (2015), 369–383.
[10] David Kanter. 2015. Graphics Processing Requirements for Enabling Immersive VR. *AMD White Paper* (2015), 1–12.
[11] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G Rogers. 2020. Locality-Centric Data and Threadblock Management for Massive GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, 1022–1036.
[12] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. 2018. Coda: Enabling Co-Location of Computation and Data for Multiple GPU Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 3 (2018), 1–23.
[13] Yuxi Liu, Xia Zhao, Magnus Jahre, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Lieven Eeckhout. 2018. Get Out of the Valley: Power-Efficient Address Mapping for GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 166–179.
[14] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-Aware GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, 123–135.
[15] Saiful A Mojumder, Marcia S Louis, Yifan Sun, Amir Kavyan Ziabari, José L Abellán, John Kim, David Kaeli, and Ajay Joshi. 2018. Profiling DNN Workloads on a Volta-Based DGX-1 System. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. IEEE, 122–133.
[16] Nvidia. 2016. NVIDIA DGX-1: Essential Instrument of AI Research. https://www.nvidia.com/en-gb/data-center/dgx-systems/dgx-1/. [Online; accessed 2022-04-16].
[17] Nvidia. 2016. NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. [Online; accessed 2022-04-16].
[18] Nvidia. 2018. NVIDIA DGX-2: Break Through the Barriers to AI Speed and Scale. https://www.nvidia.com/en-us/data-center/dgx-2/. [Online; accessed 2022-04-16].
[19] Nvidia. 2022. NVIDIA H100 Tensor Core GPU: Unprecedented Performance, Scalability, and Security for Every Data Center. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet. [Online; accessed 2023-03-02].
[20] Nvidia. 2022. NVIDIA NVLINK: High-speed GPU Interconnect. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/. [Online; accessed 2022-04-16].
[21] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, 72–83.
[22] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 14–27.
[23] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. IMPACT-12-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign. 29 pages.
[24] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. 2018. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 829–842.
[25] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, 339–351.
[26] Shiqing Zhang, Mahmood Naderan-Tahan, Magnus Jahre, and Lieven Eeckhout. 2023. SAC: Sharing-Aware Caching in Multi-Chip GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 1–13.
[27] Xia Zhao, Almutaz Adileh, Zhibin Yu, Zhiying Wang, Aamer Jaleel, and Lieven Eeckhout. 2019. Adaptive Memory-Side Last-Level GPU Caching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 411–423.
[28] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Selective Replication in Memory-Side GPU Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, 967–980.
[29] Xia Zhao, Magnus Jahre, Yuhua Tang, Guangda Zhang, and Lieven Eeckhout. 2023. NUBA: Non-Uniform Bandwidth GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 544–559.

Received xxx; revised xxx; accepted xxx