NUBA: Non-Uniform Bandwidth GPUs

Xia Zhao
Artificial Intelligence Research Center
Defense Innovation Institute
Academy of Military Science
China

Magnus Jahre
Department of Computer Science
Norwegian University of Science and Technology (NTNU)
Norway

Yuhua Tang
State Key Laboratory of High Performance Computing
College of Computer Science and Technology
National University of Defense Technology
China

Guangda Zhang
Artificial Intelligence Research Center
Defense Innovation Institute
Academy of Military Science
China

Lieven Eeckhout
Department of Electronics and Information Systems
Ghent University
Belgium

ABSTRACT

The parallel execution model of GPUs enables scaling to hundreds of thousands of threads, which is a key capability that many modern high-performance applications exploit. GPU vendors are hence increasing the compute and memory resources with every GPU generation — resulting in the need to efficiently stitch together a plethora of Symmetric Multiprocessors (SMs), Last-Level Cache (LLC) slices and memory controllers while maximizing bandwidth and keeping power consumption and design complexity in check. Conventional GPUs are Uniform Bandwidth Architectures (UBAs) as they provide equal bandwidth between all SMs and all LLC slices. UBA GPUs require a uniform high-bandwidth Network-on-Chip (NoC), and our key observation is that provisioning a NoC to match the LLC slice bandwidth incurs a hefty power and complexity overhead.

We propose the Non-Uniform Bandwidth Architecture (NUBA), a GPU system architecture aimed at fully utilizing LLC slice bandwidth. A NUBA GPU consists of partitions that each feature a few SMs and LLC slices as well as a memory controller — hence exposing the complete LLC bandwidth to the SMs within a partition since they can be connected with point-to-point links — and a NoC between partitions — to enable access to remote data. Exploiting the potential of NUBA GPUs however requires carefully co-designing system software, the compiler and architectural policies. The critical system software component is our Local-And-Balanced (LAB) page placement policy which enables the GPU driver to place data in local partitions while avoiding load imbalance. Moreover, we propose Model-Driven Replication (MDR) which identifies read-only shared data with data-flow analysis at compile time. At run time, MDR leverages an architectural mechanism that replicates read-only shared data across LLC slices when this can be done without pressuring cache capacity. With LAB and MDR, our NUBA GPU improves average performance by 23.1% and 22.2% (and up to 183.9% and 182.4%) compared to iso-resource memory-side and SM-side UBA GPUs, respectively. When the NUBA concept is leveraged to reduce overhead while maintaining similar performance, NUBA reduces NoC power consumption by 12.1× and 9.4×, respectively.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575745

CCS CONCEPTS

• **Computer systems organization → Multiple instruction, multiple data**.

KEYWORDS

GPU; Non-Uniform Bandwidth Architecture (NUBA)

ACM Reference Format:
Xia Zhao, Magnus Jahre, Yuhua Tang, Guangda Zhang, and Lieven Eeckhout. 2023. NUBA: Non-Uniform Bandwidth GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3575693.3575745

1 INTRODUCTION

GPUs are widely used in high-performance computing, machine learning and data analytics. To continuously increase the raw computing power, GPU vendors keep adding evermore Streaming Multiprocessors (SMs) to their GPUs. In addition, they increase the amount of Last-Level Cache (LLC) and memory bandwidth to satisfy the data bandwidth demand of the larger number of SMs. For example, Nvidia scaled the number of SMs from 16 to 108 (7× increase), the LLC size from 768 KB to 40 MB (53× increase), and memory bandwidth from 177 GB/s to 1.5 TB/s (9× increase) from Fermi [55] to A100 [64]; the GPUs from other vendors follow similar trends [2, 5]. GPU memory systems are hence becoming increasingly complex and power-hungry, and our goal is to redesign the GPU system architecture to improve performance while reducing power consumption and design complexity.

Conventional GPUs are *Uniform Bandwidth Architectures (UBAs)* because they provide equal bandwidth between the SMs and all

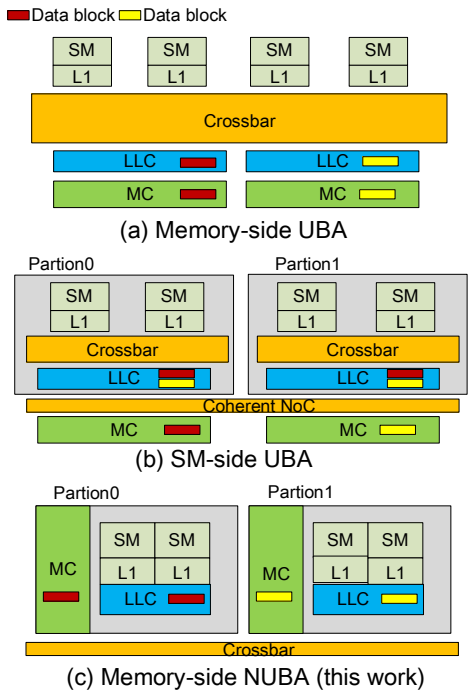


Figure 1: A memory-side Uniform Bandwidth Architecture (UBA), an SM-side UBA and our memory-side Non-Uniform Bandwidth Architecture (NUBA); the area of the crossbar/NoC indicates power overhead. *The memory-side UBA is limited by the crossbar overheads whereas the SM-side UBA does not scale because LLC partitions must be kept coherent. NUBA is scalable and provides high bandwidth at low power overhead.*

Last-Level Cache (LLC) slices. The most predominant LLC organization in modern-day GPUs from both Nvidia [55, 56, 57, 58, 60, 59] and AMD [3, 4] is a *memory-side UBA* in which each LLC slice caches a subset of the memory address space as specified by the address mapping policy [49]. A high-bandwidth crossbar NoC connects all SMs and LLC slices (see Figure 1a). Unfortunately, a high-bandwidth crossbar incurs significant power overheads, which may force architects to set the NoC bandwidth (much) lower than the LLC bandwidth to meet the power budget. The root cause is that crossbar overhead scales quadratically with the number of endpoints [22, 70, 69, 79]. To reduce NoC overhead, recent GPUs such as Nvidia's A100 [64] have adopted an *SM-side UBA* organization where each LLC slice caches data from all memory partitions (see Figure 1b). While this reduces NoC overhead, it comes at the cost of maintaining coherence among LLC partitions — adding undesirable design complexity and limiting scalability with an increasing number of partitions.

In this work, we propose the *Non-Uniform Bandwidth Architecture (NUBA)*, a GPU system architecture that distributes the SMs and LLC slices evenly across memory controllers as shown in Figure 1c; we refer to the co-located SMs, LLCs and memory controller as a *partition*. NUBA exposes the full bandwidth of the LLC slices to the SMs within the same partition as they are connected via point-to-point links, while providing access to all remote LLC slices and memory partitions via the NoC. The advantage of NUBA over UBA is twofold. First, NUBA offers a clear performance advantage over UBA by providing higher effective bandwidth to the local LLC slices. Second, since most accesses are local, NUBA is less NoC-bandwidth-sensitive which enables reducing NoC bandwidth, and thus NoC power and complexity, without adversely affecting performance. NUBA with lower NoC bandwidth yields higher (or at least similar) performance than UBA with higher NoC bandwidth.

NUBA is the GPU equivalent of the Non-Uniform Cache Architecture (NUCA) [43, 35, 41, 93, 31] and Non-Uniform Memory Architecture (NUMA) [87, 47, 71, 67, 51, 52, 27] proposals in the CPU domain. However, the motivation is fundamentally different. GPUs are highly bandwidth-sensitive as SMs switch between a large number of Cooperative Thread Arrays (CTA) when encountering a stall on one of them. This execution model implies that memory bandwidth in GPU systems is (practically) independent of latency, and it is hence critical to map frequently accessed data locally to maximize the effective memory bandwidth. In contrast, because of the limited number of concurrent threads per core — and hence limited Memory-Level Parallelism (MLP) — the primary motivation behind NUCA and NUMA in the CPU domain is not to maximize memory bandwidth but to minimize access latency by mapping data to nearby caches and memories. Moreover, the need for high bandwidth in GPUs results in power-hungry and complex NoCs whereas CPU NoCs are typically much simpler. In other words, NUBA and NUCA/NUMA embrace non-uniformity for fundamentally different reasons: NUBA maximizes the effective memory bandwidth while reducing power and complexity overhead for GPUs, whereas NUCA/NUMA minimizes the effective memory latency for CPUs.

While NUBA creates the architectural foundation for exploiting non-uniformity in GPUs, the highly parallel execution model means that we have to carefully co-design system software, the compiler, and the architectural policies to fully unleash the NUBA potential. The first challenge is to allocate memory pages such that requests mostly access the local LLC slices (to capitalize on their bandwidth benefit), while retaining load balance (to ensure that the aggregate LLC and memory bandwidth is used efficiently). Memory page allocation in GPUs is done in system software, i.e., the GPU driver on the host CPU allocates a memory page to a particular memory module upon its first access. The most commonly used page allocation policies are *first-touch* and *round-robin* [32], but they are both ineffective for NUBA GPUs. First-touch places a memory page in the local memory module of the SM that first accesses it and may cause severe load imbalance in NUBA GPUs. Round-robin evenly distributes pages across modules and does not benefit from non-uniformity. We propose *Local-And-Balanced (LAB)* page allocation to exploit the high local bandwidth of NUBA GPUs while simultaneously ensuring that pages are evenly distributed across modules. LAB tracks the number of memory pages allocated to each memory module. If the pages are not sufficiently evenly allocated, LAB allocates the page to the module that currently has the lowest number of allocated pages. Conversely, LAB allocates the page to

the memory module of the requesting partition when pages are sufficiently well distributed. LAB achieves the best of both worlds and performs similarly to first-touch for low-sharing applications and similarly to round-robin for high-sharing applications, yielding a 90.9% and 14.9% performance improvement compared to first-touch and round-robin, respectively.

The second challenge is that shared data in a NUBA GPU must necessarily be mapped to a single memory partition and hence it is remote for at least one sharer, limiting the effective bandwidth. Replicating shared data across partitions in NUBA GPUs has the potential to greatly improve the effective memory bandwidth. However, data replication may also pressure the effective cache capacity if the shared data set is large. We hence propose *Model-Driven Replication (MDR)* to trade-off the improvement in effective bandwidth versus the degradation in effective cache capacity. MDR leverages data-flow analysis within the compiler to identify read-only shared data. We focus on read-only data sharing because (i) it is common in GPUs [90, 95], and (ii) read-only data can be replicated without creating cache coherence issues. At runtime, MDR leverages a novel hardware component which employs an analytical model to balance the bandwidth benefit of shared data replication against the potential bandwidth cost of increased LLC miss rates. MDR is effective for high-sharing applications, yielding a performance improvement of 18.4% on average (and up to 183.9%) compared to no replication in a NUBA GPU baseline under LAB.

The NUBA concept can be leveraged in various ways, (1) to improve performance, (2) to reduce NoC power consumption and complexity, or (3) to improve performance *and* reduce NoC complexity, and we provide experimental results for the three cases. (1) When both the NUBA and the memory-side and SM-side UBAs are configured with a 1.4 TB/s crossbar NoC, NUBA improves performance by 23.1% and 22.2% on average (and up to 183.9% and 182.4%), respectively. (We note that keeping memory accesses mostly local is also beneficial in terms of energy, as NUBA reduces GPU energy consumption by 16.0% and 13.1% on average compared to iso-resource memory-side and SM-side UBA GPUs, respectively.) (2) When configuring the NUBA with a 700 GB/s NoC, NUBA reduces NoC power consumption by 12.1× and 9.4× compared to the memory-side and SM-side UBAs with a 5.6 TB/s NoC (as for the A100 [64]), respectively, for similar performance. (3) NUBA with a 700 GB/s NoC outperforms the memory-side and SM-side UBAs with a 1.4 TB/s NoC by 12.7% and 11.3%, while at the same time reducing NoC power by 2.3× and 1.6×, respectively.

2 BACKGROUND

GPU architecture. We now return to Figure 1 to explain the GPU architectures in more detail. Both the UBA and NUBA GPUs contain 64 SMs, 64 LLC slices and 32 memory controllers, and there is hence a 2:2:1 ratio of SMs, LLC slices and memory controllers. The partitions shown in Figure 1c are hence exactly the same as in our NUBA baseline except that our baseline has 32 partitions whereas the simplified NUBA GPU in Figure 1c has only two partitions. We refer to the LLC slices and memory controller within the partition as *local*, and the LLC slices and memory controllers in other partitions as *remote*. The memory-side UBA GPU uses a crossbar NoC to connect the L1 caches of all SMs to all LLC slices, and point-to-point

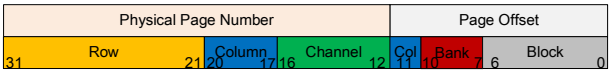


Figure 2: Partition-aware address map. *The channel bits are selected outside of the memory page offset and not randomized to give the GPU driver control over memory page placement.*

links between the LLC slices and memory controllers. The SM-side UBA GPU uses a crossbar to connect the SMs' L1 caches to the LLC slices within one of two partitions; hardware coherence keeps the two partitions (containing of 32 LLC slices each) consistent. The NUBA architecture uses a crossbar to connect all LLC slices to all other LLC slices. The interconnection network between the SMs' L1 caches and the LLC slices are low-complexity point-to-point links: we do not need input buffers and virtual channels as in typical NoC routers. For requests, routing requires inspecting the appropriate memory address bits on the L1-side, and control is provided by a round-robin arbiter on the LLC-side (which also handles requests from the inter-partition NoC). All architectures use hierarchical crossbars to minimize their power and area cost [94]. Both the UBA and NUBA GPUs can be clustered [89] (i.e., L1s in UBA and LLC slices in NUBA share NoC ports to reduce area and power overheads at the cost of reduced bandwidth), yet we use a one-to-one mapping in our setup.

Address mapping policy. Conventional UBA GPUs employ randomizing address mapping policies to evenly distribute non-shared requests across memory channels and banks [49]; shared data is necessarily located in a single memory channel which can lead to intermittent congestion [95, 94]. In a NUBA GPU, the address mapping policy needs to be designed such that the GPU driver can influence the placement of memory pages to be able to capitalize on its non-uniform bandwidth. More specifically, the channel bits need to be selected outside of the memory page as shown in Figure 2. Moreover, the address mapping policy must copy these bits directly to preserve the channel mapping. However, it can still harvest address entropy across the row, channel and bank bits, and use this to randomize the bank bits as in the PAE address mapping policy [49]; we use the least significant bank bit(s) to select the LLC slice. To ensure a fair comparison, we use the same fixed-channel address mapping policy for both the UBA and NUBA GPUs in this work, even if UBA performs slightly (3.1%) better with PAE since it also randomizes the channel bits. We show in our sensitivity analysis that our performance-optimized NUBA GPU still outperforms UBA with PAE by a significant margin (see Section 7.5 for details).

Workload sharing behavior. Before delving into the details of how to design NUBA GPUs, we first categorize our set of benchmarks based on their sharing degree as we find it useful to discuss NUBA performance. In particular, we evaluate the degree to which memory pages are shared between different SMs for the benchmarks we consider in this work. We perform this evaluation on our baseline 64-SM GPU with a 4 KB page size; we will consider other page sizes in Section 7.5.

Figure 3 shows the degree to which memory pages are shared across SMs. A page is shared if it is accessed by more than one SM

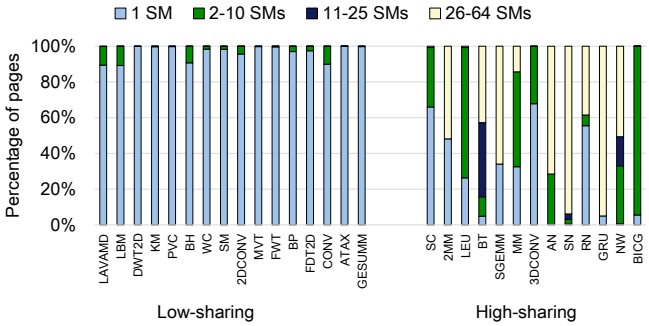


Figure 3: Memory page sharing degree for different GPU applications. *For low-sharing applications, most of the memory pages are accessed by only one SM; for high-sharing applications, a large portion of memory pages are accessed by tens of SMs.*

during execution, and we count the number of SMs that access each shared page. Benchmarks exhibit different sharing behavior, and we partition the benchmarks into two groups based on their sharing degree, see Figure 3. For the *low-sharing* applications, more than 80% of the memory pages are accessed by a single SM and are hence not shared. The *high-sharing* benchmarks feature a reasonably large fraction of shared pages. The degree of sharing varies significantly within this group. For example, ~30% of pages are shared by 2–10 SMs for SC, whereas more than 70% of the memory pages are shared by 26–64 SMs for AN, SN and GRU. In contrast to what one may expect, some applications with irregular memory access patterns exhibit low-sharing characteristics (such as MVT, ATAX and GESUMM), while others feature high-sharing characteristics (such as NW and BICG). The key difference is whether the irregular memory accesses of different SMs target the same shared memory pages or not.

The sharing degree is an important workload characteristic in the context of a NUBA architecture. NUBA will maximize performance for the low-sharing applications because most memory accesses will be local provided that (i) Cooperative Thread Arrays (CTAs) are evenly distributed across SMs, and (ii) the memory pages that these CTAs access are mapped to the local memory partition. For the high-sharing applications on the other hand, irrespective of how CTAs are distributed across SMs and how memory pages are mapped across the different memory partitions, a significant fraction of memory accesses will be remote. These remote accesses traverse the NoC and hence experience lower effective bandwidth than local accesses. It is hence critical to map memory pages optimally and maximize the effective bandwidth for applications with substantial sharing.

3 THE NON-UNIFORM BANDWIDTH ARCHITECTURE

Implementing NUBA. We first describe how NUBA supports local and remote memory accesses in the LLC slices. The SM translates virtual memory addresses to physical addresses using the TLBs (or possibly page tables). After address translation, the SM accesses the L1 cache which returns the data upon a cache hit. Otherwise, the memory request needs to access the LLC, and the LLC slice that the

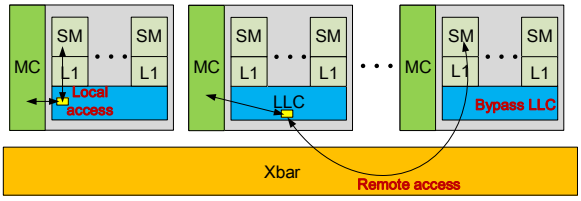


Figure 4: NUBA local and remote memory access example.

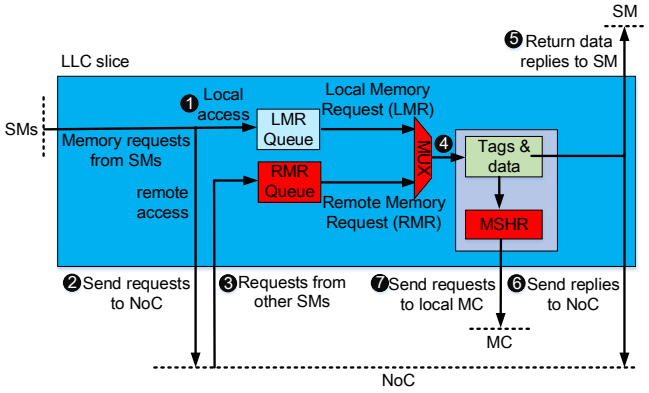


Figure 5: NUBA LLC slice microarchitecture.

data can reside in is uniquely identified by the architecture's address mapping scheme (see Section 2). Figure 4 shows an example of local and remote memory accesses in NUBA. If the memory page is mapped to the local memory partition, the data is cached by a local LLC slice and the request is sent directly to this slice. The local LLC slice responds with the data upon a cache hit. Otherwise, the request is passed on to the local memory controller, and the requested data is inserted into the cache when it returns. If the request maps to a memory page that is stored in a remote memory partition, the request bypasses the local LLC slice and instead accesses the LLC slice in the remote partition.

Figure 5 shows the LLC slice design that we use in NUBA to support local and remote memory accesses. Upon receiving a memory request, the LLC inspects the memory address of the request and uses this to determine its destination. If the request is local, it is inserted into the Local Memory Request (LMR) queue to eventually access the LLC's tag and data arrays ❶. Otherwise, the memory request is forwarded to the NoC which routes it to the appropriate remote LLC slice based on its memory address ❷. The LLC slice also needs to service remote requests that it receives through the NoC. These requests are stored in the Remote Memory Request (RMR) queue ❸. The LLC needs to arbitrate between the LMR and RMR as it can issue one request per cycle to the tag and data arrays. We use a round-robin selection policy ❹, i.e., if both the LMR and RMR contain requests we alternate between the queues in subsequent cycles.

If the memory request hits in the LLC, the data is returned to the SM directly if the request is local ❺ or inserted into the NoC if the request is remote ❻. LLC misses on both local and remote requests reserve an entry in the LLC's Miss Status Holding Register

(MSHR) and the access is forwarded to the local memory controller ❼. When the data returns, it is installed in the LLC if the request was from a local SM or forwarded to the requesting SM over the crossbar otherwise.

The NUBA design space. Our baseline NUBA GPU consists of 32 partitions with two SMs, two LLC slices and one memory controller in each partition. This is only one point in the design space though, and we now discuss how to scale NUBA with resource availability. The limiting factor is typically the number of memory controllers that can be accommodated as adding more memory channels is relatively more costly than adding compute resources. In particular, increasing the number of memory channels requires more pins which in turn requires more expensive packages in single-package designs [12], while it increases integration costs in multi-chip modules [42].

The next step is to decide upon the ratio of SMs and LLC slices per memory channel within a partition. If the architecture focuses on compute-intensive applications, it may be appropriate to have relatively many SMs per memory channel (e.g., 4 SMs per channel) whereas our baseline is more memory-oriented with 2 SMs per channel. How many LLC slices to distribute the aggregate cache capacity across also needs to be decided. Current architectures typically have twice as many LLC slices as there are memory controllers [64]. The number of LLC slices is a trade-off between providing more bandwidth — as slices can be accessed in parallel — versus the overheads of connecting the slices to each other and duplicating the access circuitry (i.e., everything except the tag and data arrays in Figure 5). We explore the performance of various NUBA configurations in Section 7.5.

We propose to keep the ratio of SMs and LLC slices to memory controllers constant when scaling NUBA GPUs. The reason is that adding more SMs increases the bandwidth demand and the architecture would hence quickly become imbalanced if compute resources are scaled faster than memory resources, and vice versa. We hence scale NUBA by increasing the number of partitions, and we adjust the focus on compute versus memory by adjusting the ratio of SMs to LLC slices and memory controllers within each partition.

NUBA die layout. We believe that manufacturing a NUBA GPU is feasible. In commercial products such as Nvidia's A100, where LLCs are concentrated in the middle of the die, all SMs already have LLC slices that are physically close versus LLC slices that are relatively far away [25]. It is hence unnecessary to re-distribute LLCs across the die to enable a NUBA architecture. Moreover, each LLC slice is already connected to a nearby memory controller in the A100. A NUBA GPU can hence be implemented by linking nearby SMs to nearby LLC slices to form partitions, and the LLC slices can retain their memory controller links. The LLC slices will then need to be connected to each other using the crossbar, but this crossbar is similar to the crossbar used to connect the SMs to the LLC slices in the conventional UBA GPU.

4 LOCAL-AND-BALANCED PAGE ALLOCATION

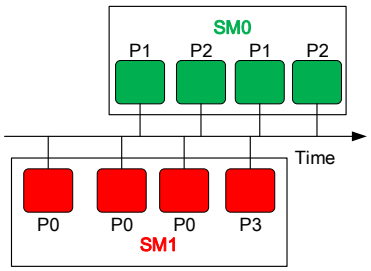
A NUBA GPU will only outperform a UBA GPU if it is able to predominantly steer memory requests to the local LLC slices and memory channels while achieving good load balance. We first explain why the existing round-robin and first-touch page allocation policies [6] fall short before we describe the details of our newly proposed Local-And-Balanced (LAB) policy.

First-touch and round-robin page allocation. The page placement policy has limited performance impact in UBA GPUs as a randomized address mapping policy will distribute memory accesses evenly across LLC slices and memory channels [49]. However, page mapping policies have been studied in the context of multi-chip module GPUs. More specifically, the designers of MCM-GPU [6] considered the *round-robin* and *first-touch* page allocation policies. Under round-robin, the GPU driver distributes memory pages evenly across memory partitions, whereas first-touch allocates a page in the memory partition closest to the SM that first accessed the page. The key difference between our NUBA architecture and a multi-chip module is the granularity of allocation.

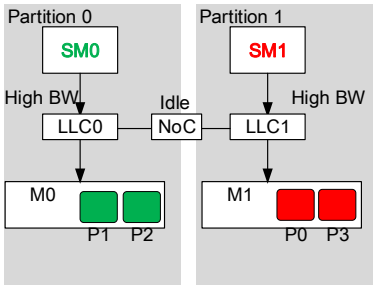
The fine-grained resource distribution in NUBA GPUs results in neither round-robin nor first-touch page allocation being effective. The reason is that low-sharing applications prefer first-touch because distributed CTA scheduling already distributes CTAs evenly across SMs while maximizing locality. The CTAs of low-sharing applications allocated to the same SM hence mostly access the same memory pages so it is beneficial to place them in the local memory partition. In contrast, first-touch is detrimental to performance for high-sharing applications because the locality optimization of distributed CTA scheduling tends to allocate heavily shared pages to relatively few memory channels. This leads to poor bandwidth utilization as few channels are heavily congested whereas others are lightly loaded. While round-robin avoids this pathological behavior, it is also unable to leverage the non-uniform bandwidth of the NUBA since it is equally likely to allocate a page to any memory channel.

We now explain this behavior in more detail with an example. Figure 6a shows a timeline of accesses for two SMs, SM0 and SM1, that are allocated to different NUBA partitions. There is no data sharing as SM0 accesses page P1 and P2 whereas SM1 accesses page P0 and P3; we number the pages in the order they are accessed by the SMs. Pages accessed by SM0 are green, pages that are accessed by SM1 are red, and shared pages are blue. Figure 6b illustrates how the first-touch policy allocates pages to memory modules on a simplified NUBA architecture with two partitions and one SM, one LLC slice, and one memory channel per partition given the access order in Figure 6a. Under first-touch, the GPU driver first allocates P0 to M1 because it is accessed by SM1. Then, it allocates P1 to M0, P2 to M0, and finally P3 to M1, hence succeeding in keeping all memory requests within the module. Figure 6c shows that round-robin is ineffective as it first allocates P0 to M0, P1 to M1, P2 to M0, and finally P3 to M2. While the load on the NoC and memory channels is equal, it is suboptimal since NoC bandwidth is lower than the local LLC bandwidth.

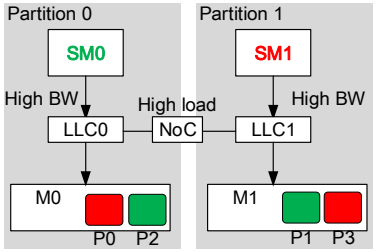
Figure 6d shows the access order of a high-sharing application in which SM0 and SM1 access the same pages. Again, first-touch allocates pages into the memory of the SM that accessed the page first (see Figure 6e). SM1 accesses P0 and P2 first and these are hence placed in M1. Then, SM0 accesses P1 which results in the GPU driver placing it in M1. SM1 then accesses P3, and it is allocated to M1.



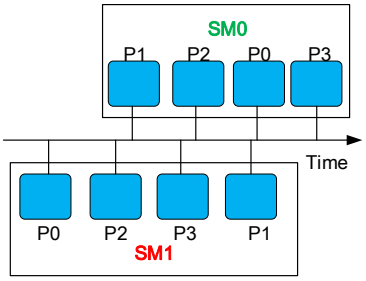
(a) Low-sharing application timeline



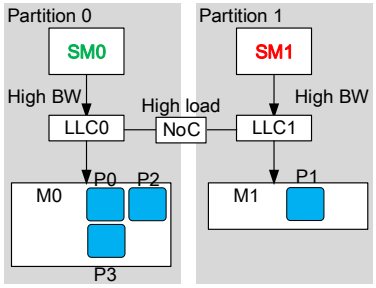
(b) Low-sharing application with first-touch



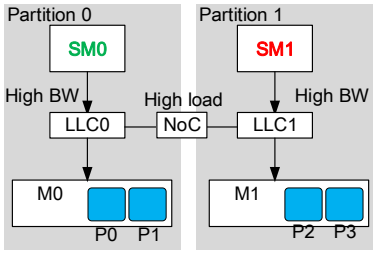
(c) Low-sharing application with round-robin



(d) High-sharing application timeline



(e) High-sharing application with first-touch



(f) High-sharing application with round-robin

Figure 6: Page allocation scheme for low-sharing and high-sharing applications. *Round-robin page allocation distributes pages evenly across memory channels while first-touch scheme allocates the page in the partition near the SM that first accesses the page.*

First-touch is hence suboptimal as M0 achieves a disproportionately large fraction of the memory memory pages. Round-robin works better for high-sharing applications as it distributes pages evenly across modules (i.e., P0 and P2 go to M0 and P1 and P3 go to M1 as shown in Figure 6f). The fundamental problem is that shared data necessarily will be accessed by multiple SMs and it is hence critical to distribute this load across channels.

Local-and-balanced page allocation. We now describe our *Local-And-Balanced (LAB)* page allocation policy which steers accesses to the local LLC slices and memory channels while at the same time distributing pages across channels to avoid load imbalance when necessary. More specifically, LAB employs first-touch as long as it can without creating load imbalance, while reverting to *least-first* page allocation otherwise. Least-first allocates a page to one of the modules that have the lowest number of pages allocated to it, breaking ties arbitrarily. Once the pages are sufficiently evenly allocated to memory modules, LAB reverts back to first-touch. In other words, while LAB improves locality through first-touch, it steers pages to the most lightly loaded memory partitions when the allocation is imbalanced — by doing so, LAB optimizes locality while retaining sufficient balance among the memory partitions.

LAB, like first-touch and round-robin, is implemented within the GPU driver. Unlike first-touch and round-robin though, LAB keeps track of the number of memory pages it has allocated to each of the memory partitions. The GPU driver runs on the host CPU and tracking the number of allocated pages hence requires allocating a 32-entry array in CPU memory for our baseline (as it has 32 channels). Under LAB, each time the GPU driver allocates a new page in GPU memory, it first computes the Normalized Page Balance (NPB):

$$\text{NPB} = \frac{1}{n} \times \sum_{i=1}^{n} \frac{P_i}{\max(P_1, P_2, \ldots, P_n)}. \quad (1)$$

More specifically, we first compute the ratio of the number of pages P_i in partition i to the maximum number of pages currently allocated to any partition. We then take the sum of these ratios across all n partitions and finally divide by n to normalize and compute NPB. NPB is a number between $1/n$ and 1 where 1 means that the memory pages are evenly allocated and $1/n$ means that all pages are allocated to a single partition. In this work, LAB adopts the first-touch policy if NPB is above 0.9 and uses least-first otherwise. We empirically determined that a threshold of 0.9 balances the performance gain across high-sharing and low-sharing applications (see Section 7.5 for details).

5 MODEL-DRIVEN DATA REPLICATION

Frequent accesses to shared data pose a challenge for NUBA GPUs. Shared data structures accessed by multiple CTAs are allocated to a single memory partition and are hence cached in its corresponding LLC slice. This creates two problems. (i) SMs in other partitions need to traverse the NoC to access the shared data in a remote LLC slice, which is suboptimal in terms of effective bandwidth compared to a local access. (ii) Multiple SMs accessing the shared data around the same time leads to camping in front of the LLC slice. We propose *Model-driven Data Replication (MDR)* of read-only shared data to alleviate this bandwidth bottleneck.

5.1 Data Replication Trade-Off

Replicating shared data across LLC slices increases the effective bandwidth as multiple SMs can then access the shared data locally and in parallel, as opposed to requiring a remote access and possibly serializing in front of an LLC slice. We propose to only replicate read-only shared data as the vast majority of shared cache lines is read-only [90, 95]. Read-write shared data is not replicated to avoid coherence overheads for keeping the replicated copies up to date. We rely on compiler analysis to identify accesses to read-only shared data structures within a kernel (as described later).

Data replication leads to a bandwidth versus cache capacity trade-off. Replicating data increases the effective bandwidth (improving performance) while at the same increasing pressure on cache capacity (degrading performance). In fact, replicating a small shared data set is most likely to be beneficial whereas replicating a large shared data set may lead to cache thrashing. MDR makes this bandwidth-capacity trade-off dynamically in hardware. If the predicted bandwidth benefits of replication outweigh the increase in LLC miss rate, MDR replicates the data; this is done on demand and on a per-cacheline basis, i.e., whenever an SM requests a shared read-only cacheline, the cacheline is cached locally and hence replicated if deemed beneficial by MDR. If not, the shared data is not replicated and SMs will need to traverse the NoC to access the shared data in a remote partition.

MDR divides time into fixed-length epochs (e.g., 20 K clock cycles) and re-evaluates the policy's selection at epoch boundaries. Profiling results collected during the previous epoch are used as input to the performance model to predict whether or not to replicate in the next epoch. MDR compares the estimated bandwidth consumption with versus without data replication. It estimates the effective bandwidth of full replication while running under no replication, and vice versa. The configuration that yields the highest effective bandwidth is adopted.

No Replication: The effective bandwidth under no replication (BW_{NoRep}) is estimated as the weighted sum of the effective bandwidth to the local and remote partitions:

$$\begin{aligned} BW_{NoRep} &= Frac_{local} \cdot BW_{local} + Frac_{remote} \cdot BW_{remote} \\ BW_{local} &= LLC_{hit} \cdot BW_{LLC} + BW_{LLC\_miss} \\ BW_{LLC\_miss} &= Min\{LLC_{miss} \cdot BW_{LLC},\ BW_{MEM}\} \\ BW_{remote} &= Min\{BW_{NoC},\ LLC_{hit} \cdot BW_{LLC} + BW_{LLC\_miss}\} \end{aligned}$$

The effective local bandwidth BW_{local} is the sum of the LLC hit and miss bandwidth. The former is the LLC hit rate times the raw LLC bandwidth, while the latter is the minimum of the raw memory bandwidth and the LLC miss rate times the raw LLC bandwidth. The effective remote bandwidth BW_{remote} is computed in a similar way except that it is further constrained by the NoC bandwidth as remote requests need to traverse the NoC. For simplicity, we assume that the LLC hit and miss rates are the same for local and remote requests.

Full Replication: Under full replication, the effective bandwidth ($BW_{FullRep}$) is estimated as the sum of the LLC hit and miss bandwidth as all L1 misses access the local LLC slices:

$$\begin{aligned} BW_{FullRep} &= LLC_{hit} \cdot BW_{LLC} + BW_{LLC\_miss} \\ BW_{LLC\_miss} &= Min\{LLC_{miss} \cdot BW_{LLC},\ BW_{local/remote}\} \\ BW_{local/remote} &= Frac_{local} \cdot BW_{MEM} + Frac_{remote} \cdot BW_{remote} \\ BW_{remote} &= Min\{BW_{NoC},\ BW_{MEM}\} \end{aligned}$$

The LLC miss bandwidth is computed as the minimum of the LLC miss rate times the raw LLC bandwidth and the effective memory bandwidth ($BW_{local/remote}$), which is a weighted sum of the local and remote memory bandwidth, with the latter being the memory bandwidth derated with the NoC bandwidth.

Note that some inputs to the above formulas are specific to the microarchitecture, i.e., the bandwidth numbers of the LLC, NoC and memory. Other parameters are a function of the workload and whether replication is applied or not, i.e., the fraction local versus remote accesses as well as the LLC hit and miss rates. The latter parameters are estimated during profiling using dynamic set sampling [75] in which we profile 8 sets in a single LLC slice. The hardware overhead to collect the required profiling information is limited to 384 bytes, i.e., 8 sets times 16 ways times 24 bits per entry. The MDR model decides whether read-only shared data should be replicated in the next 20K-cycle epoch by evaluating the above model in hardware once per epoch. MDR assumes custom circuitry with two fixed-point ALUs and control logic. We can hence compute the no-replication and full-replication cases in parallel in 116 cycles.[1] The runtime overhead for evaluating the model is hence negligible; the overhead for filling the cache with replicated cache lines is faithfully modeled in our simulation setup.

[1] Computing the LLC hit/miss rate and fraction local/remote accesses, and computing the model equations involves 4 divisions of 25 cycles each, 4 multiplications of 3 cycles each, and 2 additions and 2 comparisons of 1 cycle each (116 cycles in total).

5.2 Compiler Analysis and Runtime Support

MDR relies on the compiler to identify read-only shared data structures for the hardware to replicate if deemed appropriate. As in prior work [95], the compiler employs data flow analysis at the PTX intermediate code level [62] to identify accesses to read-only data structures within a kernel boundary. If a data structure is never written to within a kernel, it is marked as read-only; otherwise, it is marked as read-write. Note that a data structure marked as read-only in one GPU kernel can be read-write in another kernel, e.g., the output of one kernel can serve as input to another kernel. Load operations accessing read-only data structures using the `ld.global` instruction are then replaced by a newly introduced `ld.global.ro` instruction, indicating to hardware that these instructions operate on read-only data that can be replicated.

To differentiate requests to read-only and read-write shared data at runtime, the instruction decoder in hardware, upon decoding an `ld.global.ro` instruction adds a read-only bit to the memory request metadata which MDR uses to identify candidates for replication. This extra bit does not add overhead because a read request only needs to transmit the address to be fetched (8 bytes in our setup), while a write request needs to transmit both the address and the data (16 bytes). In other words, there are spare bits among the request links to mark accesses to read-only shared data. Replicating cachelines under MDR in hardware is as simple as caching remote cache lines locally, i.e., we route requests to remote shared read-only cachelines to the local LLC first and then to the remote LLC (and memory) upon a miss. We rely on the cache replacement policy to keep the hottest cachelines in the local LLC, i.e., we treat replicated cachelines equally to other cachelines.

5.3 Cache Coherence Implications

GPUs typically employ software cache coherence protocols with write-through L1 caches [61, 53, 90, 77], which is what we assume in this work. At synchronization boundaries, e.g., a `sync` instruction or kernel boundary, the SMs flush (invalidate) their L1 cache to ensure that the LLC contains the most recent values. When employing replication, our compiler support guarantees that the local LLC only caches read-only data; dirty replicas hence do not exist. We do however flush the LLC on kernel boundaries as read-only data in the current kernel can become read-write in the next kernel. (We faithfully model this overhead in our evaluation.) NUBA GPUs handle atomic instructions the same way as UBA GPUs. More specifically, UBA GPUs use the raster operation units located in the LLCs to handle atomic instructions [1, 33].

Table 1: Simulated GPU architecture.

No. SMs	64 SMs
SM resources	1.4 GHz, 32 SIMT width, 96 KB shared memory Max. 2048 threads (64 warps/SM, 32 threads/warp)
Scheduler	2 warp schedulers per SM, GTO policy
L1 data cache	48 KB per SM (6-way, 64 sets), 128 B block, 128 MSHR entries, write-through, write-no-allocate
L1 TLB	128 entries per SM, single port, 1-cycle latency, LRU
LLC	6 MB in total (64 slices, 16-way, 48 sets), 120 clock cycles latency, write-back
L2 TLB	512 entries in total, 16-way set-associative, 10-cycle latency, LRU, 2 ports
Page table walker	shared, 64 concurrent walkers
NoC	64 × 64 crossbar, 1.4 TB/s
Memory stack configuration	350 MHz, 4 memory stacks, 8 channels/stack, FR-FCFS, 64 entries/queue, 16 banks/chan., 720 GB/s
HBM Timing [20, 44]	tRC=24, tRCD=7, tRP=7, tCL=7, tWL=2, tRAS=17,tRRDl=5, tRRDs=4, tFAW=20 tRTP=7, tCCDl=1, tCCDs=1, tWTRl=4, tWTRs=2

Table 2: GPU-compute benchmarks.

Benchmark	Abbr.	Sharing Degree	Memory Footprint / Read-Only Shared
LavaMD [21]	LAVAMD	Low	7 MB / 0.9 MB
Lattice-Boltzmann [82]	LBM	Low	389 MB / 33 MB
DWT2D [21]	DWT2D	Low	302 MB / 0.01 MB
Kmeans [21]	KMEANS	Low	136 MB / 0.1 MB
Page View Count [36]	PVC	Low	1,081 MB / 0.6 MB
Black-Scholes [65]	BH	Low	48 MB / 5.3 MB
Wordcount [36]	WC	Low	542MB / 0.9 MB
Stringmatch [36]	SM	Low	146 MB / 1.2 MB
2DConvolution [34]	2DCONV	Low	1,074 MB / 17 MB
Mvt [34]	MVT	Low	6,443 MB / 0.1 MB
FastWalshTransform [65]	FWT	Low	269 MB / 0.01 MB
Backprop [21]	BP	Low	75 MB / 0.4 MB
Fdtd2D [34]	FTD2D	Low	51 MB / 0.07 MB
Convolution Separable [65]	CONVS	Low	151 MB / 20 MB
ATAX [34]	ATAX	Low	1,342 MB / 0.08 MB
Gesummv [34]	GESUMM	Low	1,073 MB / 0.1 MB
Streamcluster [21]	SC	High	302 MB / 8 MB
2MM [34]	2MM	High	84 MB / 6 MB
Leukocyte [21]	LEU	High	2 MB / 1 MB
B+tree [21]	BT	High	39 MB / 36 MB
SGemm [82]	SGEMM	High	9 MB / 8 MB
Matrixmul [65]	MM	High	8 MB / 7 MB
3DConvolution [34]	3DCONV	High	1,074 MB / 68 MB
AlexNet[86]	AN	High	1 MB / 0.4 MB
SqueezeNet [86]	SN	High	1 MB / 0.9 MB
ResNet [86]	RN	High	4 MB / 0.7 MB
Gated Recurrent Unit [86]	GRU	High	2 MB / 0.4 MB
Needleman-Wunsch [21]	NW	High	16 MB / 10 MB
BICG [34]	BICG	High	2,013 MB / 472 MB

6 METHODOLOGY

Simulated System: We substantially modified GPGPU-sim v3.2.2 [13] to evaluate NUBA. In particular, to faithfully model the memory subsystem, we have integrated Ramulator [44] into GPGPU-sim. We also added TLB and MMU support to simulate unified memory. A two-level TLB design is used where each SM has its private L1 TLB and all SMs share an L2 TLB as in prior work [8, 80, 81, 9, 91]. A TLB miss triggers a page table walk upon a page fault; up to 64 concurrent page walkers are supported. A fixed latency penalty (20 µs) is assumed to model the overhead of page fault handling and page eviction [96]. Similar to previous work, we assume 4 KB memory pages [96, 73, 81, 72, 88]; 2 MB page size is evaluated in the sensitivity analysis. We use our fixed-channel address mapping policy as discussed in Section 2 unless mentioned otherwise.

The simulated UBA and NUBA architectures feature 64 SMs, 64 LLC slices, 32 memory channels, and 4 HBM stacks, i.e., 8 memory controllers per HBM stack and 2 LLC slices per memory controller. The SM-side UBA features 2 LLC partitions with 32 LLC slices each as in Nvidia's A100 [64]. In the NUBA configuration, we assume that a partition consists of two SMs that are locally connected to two LLC slices and one memory controller. All simulated configurations assume a 1.4 TB/s hierarchical crossbar NoC [94], unless stated

otherwise, built by assembling a total of 16 8 × 8 crossbars. Each 8×8 crossbar incurs a 4-cycle latency and 16 B link width. The reply data packet size equals 136 bytes (128 bytes data plus 8 bytes control). The NoC bandwidth and latency are hence exactly the same for all iso-resource NoCs. The point-to-point links between the L1 caches and their respective local LLC slice under NUBA provide 2.8 TB/s.

We further assume distributed CTA scheduling [6] to maximize data locality within an SM (for the UBA GPU) and within a partition (for NUBA). Table 1 provides further details about the simulated GPU architectures. We use GPUWattch [48] and DSENT [83] to estimate GPU and NoC power, respectively, assuming a 22 nm technology node. The compile-time support needed by NUBA analyzes PTX code (CUDA's intermediate code representation) as compiled using nvcc 4.0 [63].

Workloads: We use a wide range of GPU benchmarks including regular applications from Rodinia [21] and CUDA SDK [65], as well as irregular applications from Mars [36], deep learning applications from Tango [86] and additional sources [34, 82]. Details are provided in Table 2. We use the default input data set for each benchmark and cover a wide range of different memory footprints (as large as 6.4 GB). The benchmarks are classified as low-sharing versus high-sharing based on the memory page sharing characteristics across SMs as discussed before. We simulate 1 billion instructions to obtain stable and representative results following common practice [7, 45, 6, 94], and we confirm that this is representative. We compute average speedup using the harmonic mean and then report average improvement as a percentage.

7 EVALUATION

We evaluate the following four proposals:

- **Memory-side UBA:** The conventional memory-side UBA GPU baseline architecture.
- **SM-side UBA:** The SM-side UBA GPU architecture.
- **NUBA-No-Rep:** The proposed NUBA GPU architecture with LAB page allocation but without MDR.
- **NUBA:** The proposed NUBA architecture with LAB page allocation and MDR-driven data replication.

7.1 Overall Performance

Iso-resource GPU architectures. Figure 7 reports performance for the two NUBA variants normalized to the conventional UBA baseline in iso-resource configurations (i.e., all architectures use the same 1.4 TB/s crossbar NoC). We focus the comparison against our baseline memory-side UBA because the SM-side UBA only marginally outperforms the baseline in cases where data replication across LLC partitions is beneficial (by 1.0% on average). NUBA provides a substantial performance speedup for both the low-sharing and high-sharing applications: we report an average performance improvement of 30.4% on average (and up to 97.9%) for the low-sharing applications versus an average performance improvement of 15.1% (and up to 183.9%) for the high-sharing applications. Overall, across our complete set of benchmarks, NUBA improves performance by 23.1% on average.

The performance improvement correlates strongly with the effective bandwidth (replies/cycle) perceived by the SMs, see Figure 8.

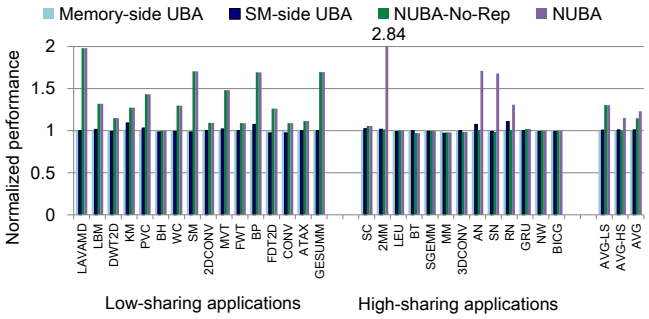


Figure 7: Performance improvement obtained by NUBA and NUBA-No-Rep over UBA. *NUBA improves performance by 30.4% on average for the low-sharing applications and by 15.1% for the high-sharing applications, and by 23.1% overall.*

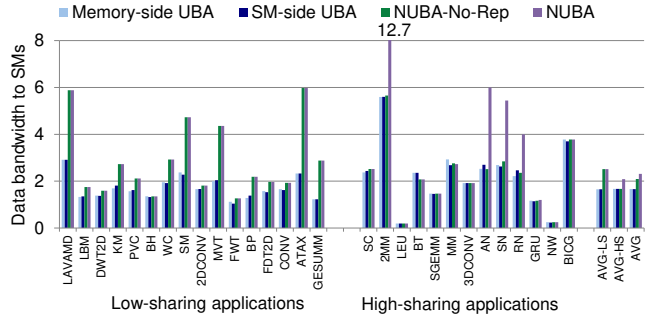


Figure 8: Memory bandwidth (replies/cycle) perceived by the SMs under UBA, NUBA-No-Rep and NUBA. *NUBA significantly improves the effective memory bandwidth.*

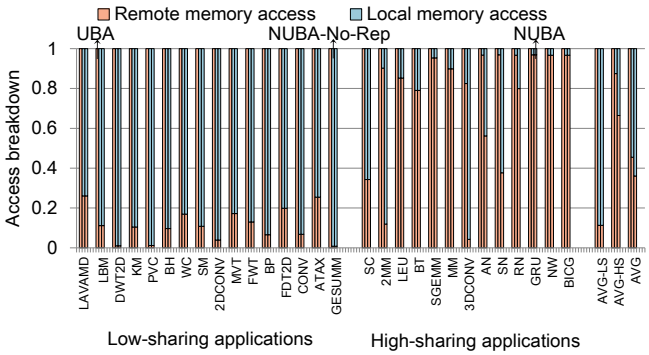


Figure 9: L1 miss breakdown under UBA (memory-side and SM-side UBA yield the same breakdown), NUBA-No-Rep and NUBA. *NUBA turns the majority of remote memory accesses into local high-bandwidth accesses.*

NUBA increases the perceived bandwidth by 51.7% on average (and up to 117.9%) for the low-sharing applications and by 24.7% on average (and up to 137.5%) for the high-sharing applications. Overall, NUBA increases the effective perceived memory bandwidth by 38.9% on average.

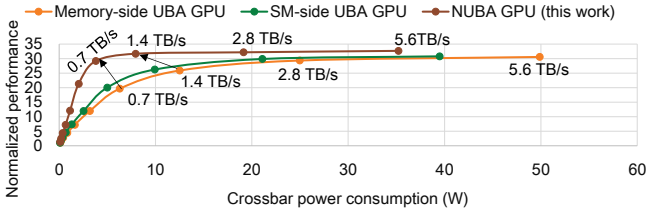


Figure 10: Performance improvement versus NoC power consumption for UBA and NUBA. *NUBA retains high performance while significantly reducing NoC power and complexity.*

The fundamental reason for the increase in perceived bandwidth is explained in Figure 9. In a conventional UBA architecture, all L1 misses have to traverse the 1.4 TB/s NoC and are hence remote. Under NUBA, the vast majority of L1 misses turn into accesses to the local LLC/memory partition over the high-bandwidth 2.8 TB/s point-to-point links. This is particularly true for the low-sharing applications; for some of the high-sharing applications, selective data replication under NUBA turns a large fraction of the accesses to shared read-only data into local accesses as well. This leads to a significant performance improvement for 2MM, AN, SN and RN. (In spite of the dramatic increase of local accesses for 3DCONV due to data replication, this does not translate into a significant performance improvement as this benchmark is relatively bandwidth-insensitive.) Overall, under NUBA, 63.9% of the L1 misses turn into local accesses.

Performance versus NoC power/complexity trade-off. While comparing iso-resource NUBA and UBA configurations enables a fair comparison, it does not exploit NUBA's potential for reducing NoC power and complexity. The fact that NUBA is much less sensitive to NoC bandwidth than the UBA architectures provides an opportunity to reduce NoC power/complexity while providing similar (or higher) performance. More specifically and as shown in Figure 10, NUBA with a 700 GB/s NoC provides similar performance as the memory-side and SM-side UBAs with a 5.6 TB/s NoC for a 12.1× and 9.4× reduction in NoC power consumption, respectively; note that a 5.6 TB/s NoC provides similar bandwidth to the NoC in the SM-side Nvidia A100 UBA [64]. NUBA can also improve performance *and* at the same time reduce NoC power consumption, see for example how NUBA with a 700 GB/s NoC outperforms the memory-side and SM-side UBAs with a 1.4 TB/s NoC by 12.7% and 11.3%, while at the same time reducing NoC power by 2.3× and 1.6×, respectively.

7.2 LAB Page Allocation

As shown in Figure 11, LAB page allocation outperforms conventional first-touch (FT) and round-robin (RR) page allocation. For the low-sharing applications, LAB and FT map memory pages to the memory partition where it is accessed locally, thereby increasing the effective memory bandwidth. In contrast, RR spreads out memory pages across the system which leads to a large fraction of remote accesses. For the high-sharing applications, FT leads to (very) heavily skewed page placement, i.e., some frequently accessed memory pages are mapped to the same memory partition, thereby leading to a severe bandwidth bottleneck when the other

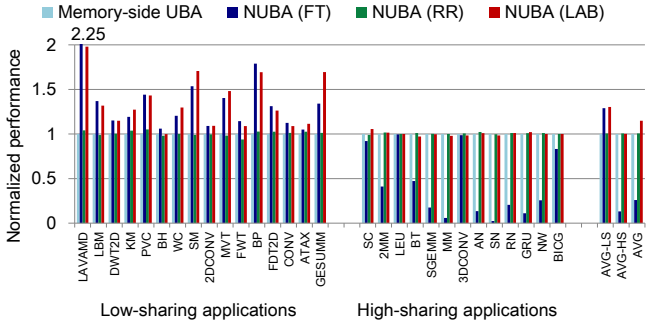


Figure 11: The impact of page allocation on NUBA performance. *LAB achieves high performance for both the low- and high-sharing applications.*

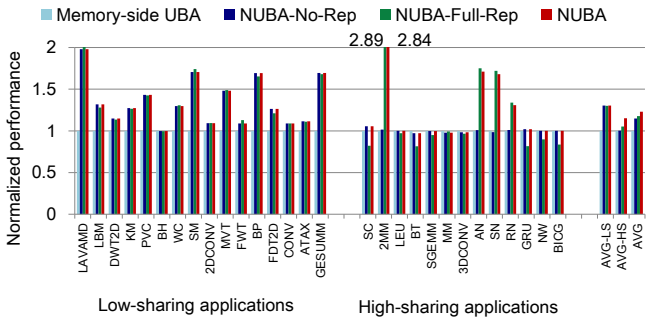


Figure 12: The impact of data replication on NUBA performance. *MDR effectively balances maximizing bandwidth to shared read-only data against reducing pressure on cache capacity.*

partitions access the data. LAB and RR alleviate this problem by evenly distributing memory pages across the different memory partitions. In NUBA, LAB improves average performance by 88.9% and 14.3% compared to first-touch and round-robin, respectively. Overall, LAB improves performance by 14.8% on average compared to UBA.

7.3 Model-Driven Data Replication

MDR is critical for the high-sharing applications. Figure 12 reports the performance impact of data replication for three configurations: no replication (No-Rep), full replication (Full-Rep) and MDR, assuming LAB page allocation. Whereas full replication dramatically improves performance for a number of benchmarks, e.g., 2MM (189.9%), AN (75.1%), SN (72.0%) and RN (33.9%), due to increased effective bandwidth to shared data. However, it also significantly degrades performance for others, e.g., SC (17.9%), BT (18.6%), GRU (18.3%) and BICG (16.5%) because of increased pressure on cache capacity, i.e., LLC miss rate increases by 10.7% (SC), 21.9% (BT), 90.0% (GRU), 10.8% (BICG). MDR only replicates cache lines if deemed beneficial, which leads to an average performance improvement of 15.1% compared to no replication.

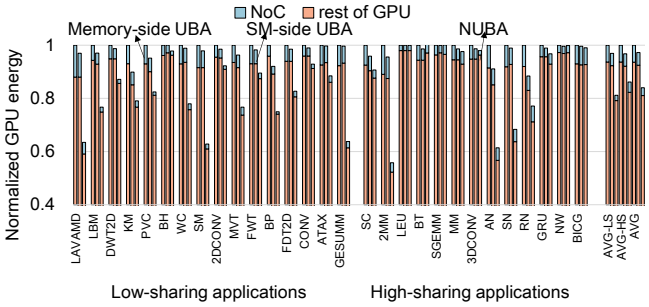


Figure 13: Normalized GPU energy consumption: NoC versus the rest of the GPU. *NUBA decreases NoC energy by 54.5% and total GPU energy by 16.0% on average.*

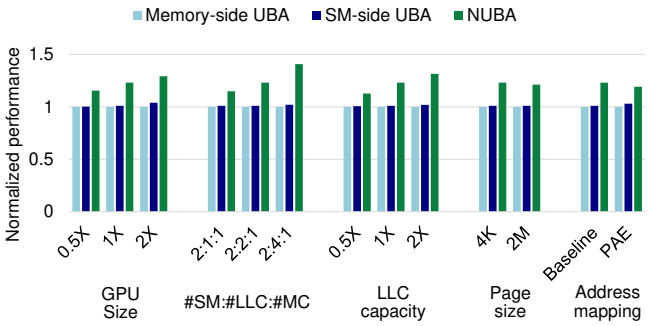


Figure 14: Sensitivity analyses. *NUBA improves performance across the broad design space while varying the number of SMs, partition setup, LLC capacity, page size and address mapping policy.*

7.4 Energy Consumption

NUBA reduces NoC energy compared to UBA by 54.5% on average by having fewer requests traverse the NoC, i.e., most accesses are to the local memory partition. Due to the reduction in execution time, NUBA leads to a reduction in overall GPU energy by 16.0% on average and up to 44.2%, see Figure 13. This is a substantial improvement over SM-side UBA which reduces NoC energy by 25.9% and total GPU energy by 2.9% on average.

7.5 Sensitivity Analyses

We now evaluate NUBA's effectiveness across the broad design space, see also Figure 14.

GPU size: We assume a 2:2:1 ratio of SMs, LLC slices and memory channels per partition as we scale system size by 0.5×, 1× (baseline) and 2×, i.e., we scale compute, LLC and memory bandwidth proportionally while keeping LLC slice capacity constant (we thus effectively scale LLC capacity). NUBA's average performance benefit increases from 15.9% to 23.1% and 30.1%, respectively. Increased LLC capacity increases the number of accesses to the local LLC slices, which increases the opportunity for NUBA to outperform UBA.

Partition: We assumed a 2:2:1 ratio of SMs, LLC slices and memory controllers throughout the paper. We now change the number of LLC slices per partition while keeping total LLC capacity constant. Increasing the number of LLC slices increases the effective bandwidth that the LLC can provide to the local SMs, which in turn increases the opportunity for NUBA to outperform UBA. We report a performance improvement of 15.1%, 23.1% and 41.2% for 1, 2 and 4 LLC slices per partition, respectively.

LLC capacity: NUBA performs better for systems with larger LLC capacity, changing from 12.9% at 0.5× to 31.7% at 2× the baseline capacity. The reason is that increased LLC capacity increases the fraction of local memory accesses under NUBA.

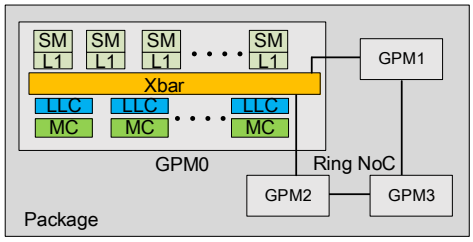
Page size: Increasing the memory page size from 4 KB to 2 MB slightly reduces the performance benefit obtained through NUBA to 21.6%. A large page size does not affect NUBA's effectiveness for low-sharing applications. For high-sharing applications, a large page size increases the sharing degree, thereby decreasing the number of memory accesses that can be turned into local accesses. Fortunately, NUBA's selective data replication largely counters this effect.

Address mapping: Changing the UBA address mapping policy from our fixed-channel policy as described in Section 2 to PAE [49] to randomize the channel bits in addition to the bank bits does not significantly affect NUBA's performance benefit (19.7% average improvement).

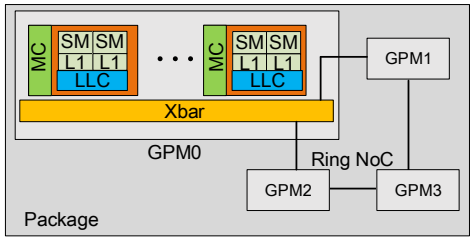
LAB threshold: LAB uses a normalized page balance threshold to tune its affinity for placing data locally. Increasing the threshold slightly improves the performance for high-sharing applications while slightly reducing the performance for low-sharing applications. The reason is that LAB tolerates less page imbalance with a higher threshold and hence places fewer pages locally. Conversely, decreasing the threshold benefits low-sharing applications and reduces performance for high-sharing applications. The performance effect is limited, i.e., LAB with thresholds of 0.95 and 0.8 improves average performance by 13.1% and 14.5%, respectively, compared to UBA, which is only slightly worse than the 14.8% average performance improvement LAB achieves with our default threshold of 0.9.

7.6 NUBA in Alternative Configurations

MCM-GPU systems: Multi-GPU systems have recently been proposed to continue to scale GPU performance in spite of Moore's Law slowing down. GPUs can be connected through the PCB (e.g., Nvidia NVLink and NVSwitch [66]) or via an interposer (e.g., multi-chip-module (MCM) GPUs [6], see also Figure 15). We find that NUBA is even more important for MCM-GPUs than it is for monolithic GPUs as the inter-module link bandwidth is smaller than the NoC bandwidth. We find that NUBA improves performance by 40.0% on average for an MCM-GPU with 128 SMs, 128 LLC slices and 64 memory controllers distributed across four modules with 720 GB/s bidirectional inter-module links, versus 30.1% for a monolithic GPU with the same number of SMs, LLC slices and memory controllers, see Figure 16. For the low-sharing applications, the performance improvement in an MCM-GPU context through NUBA is similar to what we observe for monolithic GPUs. In contrast, we note substantially higher performance improvements for the high-sharing applications as shared data replication across modules becomes even more important due to lower inter-module link bandwidth versus on-chip NoC bandwidth.



(a) Memory-side UBA MCM



(b) NUBA MCM

Figure 15: UBA and NUBA Multi-Chip Module (MCM) GPUs.

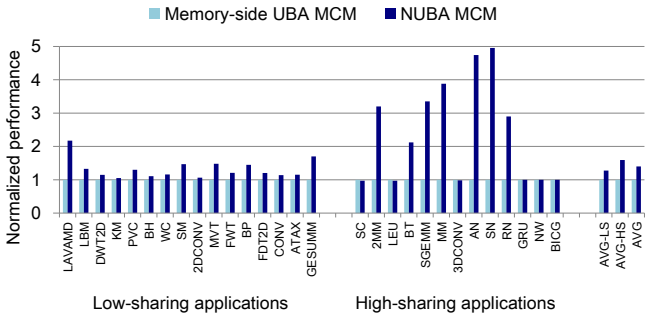


Figure 16: The impact of NUBA on MCM-GPU performance. *NUBA is even more important for MCM-GPU systems.*

Alternative page allocation: Prior work proposed page migration based on page access counts over a predefined interval [14], and page replication if there is sufficient free memory, trading off memory capacity versus memory bandwidth [27]. Both techniques could be used as alternative solutions for the LAB page allocation policy proposed in this work. We find that for the low-sharing applications, page migration and replication deliver a performance improvement of around 26% as these techniques indeed bring the data close to the requesting SMs. For the high-sharing applications, we find that page migration and replication degrade performance by up to 80.4% (2MM) and 60.1% (3DCONV). Page migration incurs high migration overhead when different SMs access the same memory pages. In contrast, LAB does not migrate pages, instead it evenly distributes pages across the system. Page replication leads to severe cache thrashing effects. In contrast, MDR replicates at the cache line level, not the page level, to maximize bandwidth to read-only shared data.

8 RELATED WORK

The most closely related work to NUBA targets MCM-GPUs [6] and multi-socket GPU systems [53]. Their goal is primarily to overcome manufacturing limitations, yet it indirectly leads to a non-uniform GPU architecture where each module or package has local LLC and memory resources. Young et al. [90] propose caching remote data in the local video memory of a GPU module to reduce communication overhead; Baruah et al. [14] propose the Griffin page migration system for migrating data between GPUs in multi-GPU systems; and Khairy et al. [42] propose to create a virtual uniform architecture consisting of discrete GPUs that internally are MCMs. This body of prior work focuses on non-uniformity across packages and modules whereas we focus on exposing and exploiting non-uniformity across NUBA partitions within a single package or module. The key difference is that NUBA has (much) more fine-grained resource partitioning, i.e., there are 64 SMs per module in MCM-GPU [6] compared to two (or four) SMs per NUBA partition — leading to more severe load imbalance and more common inter-partition data sharing.

Another branch of related work focuses on reorganizing the internal resources within a GPU module. More specifically, they explore shared versus private LLC and L1 cache organizations [39, 40, 95, 94]. These architectures are all UBAs and do not exploit that supplying uniform bandwidth between SMs and caches can be inefficient.

As mentioned before, NUBA is the GPU equivalent of NUCA [43] in the CPU domain, yet NUBA and NUCA are fundamentally different as the cost of non-uniformity is bandwidth in GPUs versus latency in CPUs. The goal in the CPU domain is to minimize latency, and many works have focused on favorably combining shared and private LLC organizations to get the capacity advantage of the shared LLC and the latency advantage of a private LLC. For example, researchers have proposed to combine intelligent data placement with dynamic migration [35, 41, 93], selective data replication [19, 24, 92, 15, 37] and adaptive cache capacity sharing [31, 74]. For GPUs, the key issues are to (i) distribute memory pages across partitions such that memory bandwidth utilization is maximized (i.e., LAB), and (ii) leverage that the GPU programming model enables effective compiler-guided replication (i.e., MDR). Non-uniformity maximizes the effective memory bandwidth to the SMs in a GPU while at the same time reducing NoC complexity.

The challenge of allocating threads to cores while simultaneously optimizing locality and load-balance also occurs in the CPU domain. One line of prior work focuses on developing affinity-targeted thread mapping policies [78, 29, 17, 30, 23, 84, 11, 17, 76, 38, 46]. Another line of work focuses on data mapping, i.e., the assignment of memory pages to memory controllers. Basic data mapping mechanisms such as home-node allocation, page-interleaved allocation, and first-touch allocation have been proposed [47, 68]. Other prior work exploits data allocation, migration, and replication techniques [54, 16, 71, 50, 67], or relies on hardware performance counters to analyze memory access behavior which in turn supports decision-making mechanisms in the operating system or hardware [51, 52, 18, 87, 85, 10, 28, 26]. The primary goal of this prior work is to minimize the effective memory access latency. In contrast, NUBA's primary goal is to maximize the effective memory

bandwidth within each partition while at the same time balancing bandwidth demand across all partitions to maximize aggregate memory system bandwidth.

9 CONCLUSION

We have introduced the Non-Uniform Bandwidth Architecture (NUBA) which, in contrast to the current Uniform Bandwidth Architecture (UBA) paradigm, exposes the complete bandwidth of LLC slices to SMs without incurring prohibitive NoC power and area overheads. Realizing the potential of NUBA GPUs requires a full-stack approach. First, we need to place data in local NUBA partitions — to capitalize on the higher bandwidth to local LLC slices compared to remote LLC slices — while avoiding load imbalance. We hence propose the Local-And-Balanced (LAB) page allocation policy which places memory pages in local memory partitions while maintaining sufficient load balance. Second, our Model-Driven Replication (MDR) scheme further increases the proportion of local accesses by replicating shared read-only data across partitions when the shared data set is sufficiently small. We showed that the NUBA concept can be used to (1) improve performance compared to iso-resource UBA GPUs, (2) reduce NoC power consumption significantly by maintaining similar performance to UBA GPUs with much higher NoC bandwidth, and (3) both improve performance *and* reduce NoC complexity and power overhead.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their thoughtful and valuable feedback. Xia Zhao is supported by the National Natural Science Foundation of China (Grant No. 62102438) and sponsored by the Beijing Nova Program. 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] Tor M. Aamodt, Wilson W. L. Fung, and Timothy G. Rogers. 2018. *General-Purpose Graphics Processor Architectures.* Morgan & Claypool Publishers.

[2] AMD. 2012. AMD Graphics Core Next. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf.

[3] AMD. 2019. Introducing RDNA Architecture. https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[4] AMD. 2020. Introducing AMD CDNA Architecture. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf.

[5] AMD. 2021. AMD Radeon PRO V620. https://www.amd.com/en/products/server-accelerators/amd-radeon-pro-v620.

[6] 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)*. 320–332.

[7] Akhil Arunkumar, Shin-Ying Lee, Vignesh Soundararajan, and Carole-Jean Wu. 2018. 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)*. 221–234.

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 136–150.

[9] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 503–518.

[10] Manu Awasthi, David Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. 2010. Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 319–330.

[11] Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. 2009. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 56–65.

[12] Ali Bakhoda, John Kim, and Tor M. Aamodt. 2010. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 421–432.

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

[14] 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)*. 596–609.

[15] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. 2006. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 443–454.

[16] François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-Andr Wacrenier, and Raymond Namyst. 2010. Structuring the Execution of OpenMP Applications for Multicore Architectures. In *Proceedings of the International Symposium on Parallel Distributed Processing (IPDPS)*. 1–10.

[17] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. 2010. ForestGOMP: an Efficient OpenMP Environment for NUMA Architectures. *International Journal of Parallel Programming* 38, 5 (2010), 418–439.

[18] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1994. Scheduling and Page Migration for

Multiprocessor Compute Servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 12–24.

[19] Jichuan Chang and Gurindar S. Sohi. 2006. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 264–276.

[20] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R. Johnson, Stephen W. Keckler, Minsoo Rhu, and William J. Dally. 2017. Architecting an Energy-Efficient DRAM System for GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*. 73–84.

[21] 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)*. 44–54.

[22] Gregory K. Chen, Mark A. Anders, and Himanshu Kaul. 2017. Scalable crossbar apparatus and method for arranging crossbar circuits. US Patent 9,577,634.

[23] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. 2006. MPIPP: An Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *Proceedings of the International Conference on Supercomputing (ICS)*. 353–360.

[24] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2005. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 357–368.

[25] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 2021. 3.2 The A100 Datacenter GPU and Ampere Architecture. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, Vol. 64. 48–50.

[26] Ilaria D. Gennaro, Alessandro Pellegrini, and Francesco Quaglia. 2016. OS-Based NUMA Optimization: Tackling the Case of Truly Multi-thread Applications with Non-partitioned Virtual Page Accesses. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 291–300.

[27] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 381–394.

[28] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. KMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*. 277–288.

[29] Matthias Diener, Felipe Madruga, Eduardo Rodrigues, Marco Alves, Jorg Schneider, Philippe Navaux, and Hans-Ulrich Heiss. 2010. Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors. In *Proceedings of International Conference on High Performance Computing and Communications (HPCC)*. 491–496.

[30] Wei Ding, Yuanrui Zhang, Mahmut Kandemir, Jithendra Srinivas, and Praveen Yedlapalli. 2013. Locality-aware Mapping and Scheduling for Multicores. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 1–12.

[31] Haakon Dybdahl and Per Stenstrom. 2007. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 2–12.

[32] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM* 58, 12 (2015), 59–66.

[33] David B. Glasco, Peter B. Holmqvist, George R. Lynch, Patrick R. Marchand, Karan Mehra, and James Roberts. 2012. Cache-based Control of Atomic Operations in Conjunction With an External ALU Block. US Patent 8,135,926 B1.

[34] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Proceedings of Innovative Parallel Computing (InPar)*. 1–10.

[35] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 184–195.

[36] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 260–269.

[37] Christopher Hughes, Changkyu Kim, and Yen-Kuang Chen. 2010. Performance and Energy Implications of Many-Core Caches for Throughput Computing. *IEEE Micro* 30, 6 (November 2010), 25–35.

[38] Joshua Hursey, Jeffrey M. Squyres, and Terry Dontje. 2011. Locality-Aware Parallel Process Mapping for Multi-core HPC Systems. In *Proceedings of International Conference on Cluster Computing (CLUSTER)*. 527–531.

[39] Mohamed A. Ibrahim, Onur Kayiran, Yasuko Eckert, Gabriel H. Loh, and Adwait Jog. 2020. Analyzing and Leveraging Shared L1 Caches in GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 161–173.

[40] Mohamed A. Ibrahim, Onur Kayiran, Yasuko Eckert, Gabriel H. Loh, and Adwait Jog. 2021. Analyzing and Leveraging Decoupled L1 Caches in GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 2–12.

[41] Mahmut Kandemir, Feihui Li, Mary Jane Irwin, and Seung Woo Son. 2008. A Novel Migration-based NUCA Design for Chip Multiprocessors. In *Proceedings of the Conference on Supercomputing (SC)*. 1–12.

[42] 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)*. 1022–1036.

[43] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 211–222.

[44] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (January 2016), 45–49.

[45] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 307–319.

[46] Argonne National Laboratory. 2013. Using the Hydra Process Manager. https://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager.

[47] Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. 2009. Affinity-on-next-Touch: An Extension to the Linux Kernel for NUMA Architectures. In *Proceedings of the International Conference on Parallel Processing and Applied Mathematics (PPAM)*. 576–585.

[48] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 487–498.

[49] 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)*. 166–179.

[50] Henrik Löf and Sverker Holmgren. 2005. Affinity-on-next-Touch: Increasing the Performance of an Industrial PDE Solver on a Cc-NUMA System. In *Proceedings of the International Conference on Supercomputing (ICS)*. 387–392.

[51] Jaydeep Marathe and Frank Mueller. 2006. Hardware Profile-Guided Automatic Page Placement for CcNUMA Systems. In *Proceedings of the International Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 90–99.

[52] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-directed Page Placement for ccNUMA via Hardware-generated Memory Traces. *J. Parallel and Distrib. Comput.* 70, 12 (2010), 1204–1219.

[53] 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)*. 123–135.

[54] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade;eacute;. 2000. Is Data Distribution Necessary in OpenMP?. In *Proceedings of the Conference on Supercomputing (SC)*. 47–61.

[55] NVIDIA. 2009. NVIDIA's Next Generation CUDA Compute Architecture. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[56] NVIDIA. 2012. NVIDIA GeForce GTX 680. https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf.

[57] NVIDIA. 2014. NVIDIA GeForce GTX 980. https://www.microway.com/download/whitepaper/NVIDIA_Maxwell_GM204_Architecture_Whitepaper.pdf.

[58] NVIDIA. 2016. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[59] NVIDIA. 2016. NVIDIA Turing GPU Architecture. https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[60] NVIDIA. 2017. NVIDIA Tesla V100 Volta Architecture. http://www.nvidia.com/object/volta-architecture-whitepaper.html.

[61] NVIDIA. 2018. VOLTA Architecture and performance optimization. http://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf.

[62] NVIDIA. 2019. Parallel Thread Execution ISA Version 6.5. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[63] NVIDIA. 2020. CUDA COMPILER DRIVER NVCC. https://docs.nvidia.com/pdf/CUDA_Compiler_Driver_NVCC.pdf.

[64] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[65] NVIDIA. 2022. NVIDIA CUDA SDK Code Samples. https://developer.nvidia.com/cuda-downloads.

[66] NVIDIA. 2022. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.

[67] Takeshi Ogasawara. 2009. NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 377–390.

[68] Oracle. 2010. Solaris OS Tuning Features. https://docs.oracle.com/cd/E18659_01/html/821-1381/aewda.html.

[69] Giorgos Passas, Manolis Katevenis, and Dionisis Pnevmatikatos. 2010. A 128 x 128 x 24Gb/s Crossbar Interconnecting 128 Tiles in a Single Hop and Occupying 6% of Their Area. In *Proceedings of the International Symposium on Networks-on-Chip (NoCS)*. 87–95.

[70] Giorgos Passas, Manolis Katevenis, and Dionisios Pnevmatikatos. 2011. VLSI Micro-Architectures for High-Radix Crossbar Schedulers. In *Proceedings of the International Symposium on Networks-on-Chip (NoCS)*. 217–224.

[71] Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler Support for Selective Page Migration in NUMA Architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*. 369–380.

[72] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. 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)*. 743–758.

[73] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 568–578.

[74] Moinuddin K Qureshi. 2009. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 45–54.

[75] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 167–178.

[76] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2012. Optimal Task Assignment in Multithreaded Processors: A Statistical Approach. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 235–248.

[77] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 582–595.

[78] Eduardo R. Rodrigues, Felipe L. Madruga, Philippe O. A. Navaux, and Jairo Panetta. 2009. Multi-core Aware Process Mapping and its Impact on Communication Overhead of Parallel Applications. In *Proceedings of the International Symposium on Computers and Communications (ISCC)*. 811–817.

[79] Korey Sewell, Ronald G. Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F. Wenisch, Dennis Sylvester, David Blaauw, and Trevor Mudge. 2012. Swizzle-Switch Networks for Many-Core Systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2, 2 (June 2012), 278–294.

[80] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 180–192.

[81] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 352–363.

[82] 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. University of Illinois at Urbana-Champaign.

[83] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. 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)*. 201–210.

[84] David Tam, Reza Azimi, and Michael Stumm. 2007. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 47–58.

[85] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2004. Using Hardware Counters to Automatically Improve Memory Performance. In *Proceedings of the International Conference on Supercomputing (SC)*. 46–46.

[86] San Jose State University. 2019. Tango: A Deep Neural Network Benchmark Suite for Various Accelerators. https://gitlab.com/Tango-DNNbench/Tango.

[87] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 279–289.

[88] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[89] Lu Wang, Xia Zhao, David Kaeli, Zhiying Wang, and Lieven Eeckhout. 2018. Intra-Cluster Coalescing to Reduce GPU NoC Pressure. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 990–999.

[90] 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)*. 339–351.

[91] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. 2020. HPE: Hierarchical Page Eviction Policy for Unified Memory in GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2461–2474. https://doi.org/10.1109/TCAD.2019.2944790

[92] Michael Zhang and Krste Asanovic. 2005. Victim Replication: Maximizing Capacity While Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 336–345.

[93] Yuanrui Zhang, Wei Ding, Mahmut Kandemir, Jun Liu, and Ohyoung Jang. 2011. A Data Layout Optimization Framework for NUCA-based Multicores. In *Proceedings of International Symposium on Microarchitecture (MICRO)*. 489–500.

[94] 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)*. 411–423.

[95] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Selective Replication in Memory-Side GPU Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 967–980.

[96] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 345–357.

Received 2022-07-07; accepted 2022-09-22