DGNA: Dissecting GPU NUMA Architecture through Microbenchmarking and Data Analysis

CHANGXI LIU*, The Hong Kong University of Science and Technology (Guangzhou), China
YUN CHEN, The Hong Kong University of Science and Technology (Guangzhou), China
TREVOR E. CARLSON, National University of Singapore, Singapore

Graphics Processing Units (GPUs), due to their immense parallel processing capabilities, have become essential across various fields, including gaming and artificial intelligence. With significant advancements in GPU cores, GPU memory efficiency has lagged, resulting in bottlenecks that can limit workload efficiency. To bridge this gap, a deep understanding of GPU memory architectures, particularly Non-Uniform Memory Access (NUMA) mechanisms within L2 and DRAM, is essential for optimizing applications, designing new architectures, and building accurate simulators. However, the latest GPU hardware from vendors like NVIDIA and AMD is still a black-box, making it challenging for researchers to understand the details of their design.

In this paper, we introduce DGNA, a methodology designed to unveil the NUMA architecture of the GPU memory hierarchy through microbenchmarking and data analysis. Specifically, we propose an approach to measuring the latency of L2 caches and DRAM without relying on the intrinsic instructions of the architecture and apply a Gaussian mixture model to filter out outliers and accurately determine latency distributions. We apply DGNA on NVIDIA's A100 and H100 GPUs, revealing NUMA node architecture, SM-NUMA relationships, and NUMA-aware memory allocation strategies used to maintain cache coherence. To the best of our knowledge, this is the first paper to detail the NUMA architecture within the GPU memory subsystem.

CCS Concepts: • **Computing methodologies** → **Model development and analysis**; **Graphics processors**.

Additional Key Words and Phrases: GPU Architecture, Benchmarking, Memory Hierarchy, NUMA Systems

ACM Reference Format:
Changxi Liu, Yun Chen, and Trevor E. Carlson. 2026. DGNA: Dissecting GPU NUMA Architecture through Microbenchmarking and Data Analysis. *ACM Trans. Arch. Code Optim.* 1, 1 (January 2026), 20 pages. https://doi.org/10.1145/3841173

1 Introduction

GPUs have become essential across a broad range of fields, including gaming [36, 38], graphics rendering [39], ray tracing [18], and artificial intelligence [1, 11, 37]. Rapid advancements in GPU core development have enabled these processors to achieve remarkable performance levels, reaching up to 33.5 peak FP-64 TFLOPS [33]. However, while computational power has surged, GPU memory has not kept pace, creating a gap that can limit GPU workload efficiency.

Understanding GPU memory is essential for several reasons. First, it allows programmers to develop highly efficient code that maximizes the utilization of GPUs. Second, it aids researchers in understanding the latest GPU designs,

*This work was completed while the author was at the National University of Singapore.
New Paper, Not an Extension of a Conference Paper.

Authors' Contact Information: Changxi Liu, changxiliu@hkust-gz.edu.cn, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China; Yun Chen, yunchen@hkust-gz.edu.cn, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China; Trevor E. Carlson, National University of Singapore, Singapore, tcarlson@nus.edu.sg.

Manuscript submitted to ACM

enabling the design of more sophisticated and optimized architectures. Third, a deep understanding of GPU design is essential for developing accurate simulators.

However, a significant challenge lies in the closed-source nature of advanced GPUs from major vendors, including NVIDIA and AMD. While numerous prior works [19, 20, 28, 45] have proposed methods to reveal memory hierarchy details, including cache line size, hit latency and bandwidth, there is a lack of approaches aiming to reveal L2 and DRAM Non-Uniform Memory Access (NUMA) mechanisms on GPUs. In contrast, recent research works [2, 24, 26, 27, 29, 40, 49, 50, 52] have focused on improving GPU NUMA architecture to optimize memory access patterns, minimize contention, and enhance scalability. To illustrate the methodologies employed by industry and to establish a solid baseline for researchers, approaches that provide insight into the NUMA architecture of GPUs are required.

In this paper, we introduce DGNA, a methodology designed to reveal the NUMA architecture in GPU memory subsystems. DGNA measures L2 and DRAM latencies independently from vendor-provided intrinsic instructions, which we show can provide misleading results in specific instances. Then we develop a methodology to automatically filter out outliers, often triggered by factors such as DRAM refresh, and subsequently derive accurate and reliable memory latency values using a Gaussian mixture model [14, 44]. DGNA analyzes the distribution of memory accesses across different levels of the memory hierarchy and their NUMA nodes. Additionaly, by examining the NUMA architecture, DGNA illustrates the correspondence between Graphics Processing Clusters (GPCs), GPU streaming multiprocessors (SMs), and L2 and DRAM NUMA nodes. Furthermore, DGNA unveils mechanisms of read and write instructions on NUMA nodes, providing deeper insights into the memory accesses. To demonstrate the broad applicability of DGNA across various GPUs, we apply DGNA to the latest NVIDIA GPU architectures, including Ampere (A100) and Hopper (H100).

First, our analysis reveals several key findings for the A100 and H100 GPUs based on the latency distribution. Using a Gaussian mixture model, we derive the latencies for reading data on local and remote DRAM NUMA nodes, which are 385 cycles and 546 cycles on the A100, and 554 cycles and 728 cycles on the H100, respectively. Additionally, reading data on local and remote L2 NUMA nodes results in latencies of 218 cycles and 379 cycles on the A100 and 295 cycles and 470 cycles on the H100, respectively.

We observe that L2 NUMA nodes on both the H100 and A100 operate under a first-touch mechanism, where the latency for an SM accessing a data block is affected by previous SMs that accessed the same block. Furthermore, we also find that H100 incorporates sub-NUMA nodes [4] within all L2 NUMA nodes, introducing an access latency gap 32 cycles accessing sub-NUMA nodes. In contrast, the A100 lacks sub-NUMA nodes, which we attribute to the larger L2 cache size on the H100 (50 MB), compared to the A100 (40 MB).

By analyzing latency gaps, we conclude that SMs access remote L2 and DRAM NUMA nodes through interconnections between L2 NUMA nodes as the observed NUMA latency gaps for accessing remote NUMA node of L2 and DRAM are similar. Specifically, on the A100, we observe a NUMA latency gap of 161 cycles for DRAM and 161 cycles for L2 caches, while on the H100, the NUMA latency gaps on DRAM and L2 are 174 cycles and 175 cycles, respectively.

Second, DGNA reveals the corresponding relationship between SMs, GPCs, L2 NUMA nodes, and DRAM NUMA nodes. We find that 2 GPCs, each containing a group of SMs, connect to a sub-NUMA node, while 4 GPCs connect to an L2 NUMA node on both the A100 and H100. The distribution of SMs on the A100 is imbalanced across different L2 NUMA nodes, while the H100 addresses this imbalance for L2 NUMA nodes. However, H100 still exhibits imbalances when it comes to the sub-NUMA nodes.

Third, DGNA reveals the read and write mechanisms of NUMA nodes on A100 and H100 GPUs. The home node for a data block refers to the node that stores its values [15]. We define the *DRAM home node* as the node that contains the

physical page of the data block. The *L2 home node* is the node directly connected to the DRAM home node, while the *home SM* refers to the SM node that is directly connected to the L2 home node. The terms *DRAM remote node*, *L2 remote node*, and *remote SM* refer to the reverse of the home nodes.

Our findings indicate that for read operations, home SMs directly read the data block into the L2 home node, while remote SMs buffer data into both L2 home and remote NUMA nodes. For write operations, home SMs write data blocks solely to the L2 home node. In contrast, remote SMs update both local and remote NUMA nodes if a cache hit occurs in the L2 home node. If the cache hit occurs only on the L2 remote node, remote SMs only update the L2 remote node.

In summary, we make the following contributions:

- We propose DGNA, a framework to reveal the NUMA architecture of the GPU memory hierarchy through microbenchmarking and data analysis. We will open-source DGNA to support further research.
- We propose an independent latency measurement methodology. DGNA accurately captures L2 and DRAM latencies without relying on vendor-specific intrinsic instructions, which can introduce misleading values at certain data points. We employ a mathematical approach, the Gaussian mixture model, to automatically detect and remove outliers caused by various factors across different architectures. By preprocessing the dataset, we can derive a more accurate understanding of memory access behavior.
- Our analysis reveals the corresponding relationship among SMs, L2 and DRAM NUMA nodes and the NUMA allocation mechanisms. We find that L2 NUMA nodes in both GPUs operate based on a first-touch mechanism, affecting latency depending on prior SM access. Furthermore, we identify that the H100 includes sub-NUMA nodes within its L2 NUMA nodes, likely due to its larger cache size, while A100 lacks this sub-structure.
- We demonstrate the read and write mechanisms of L2 NUMA on the A100 and H100. We conclude that home SMs perform direct reads and writes to home L2 nodes, while remote SMs apply specific buffering and updating strategies across both local and remote NUMA nodes.

2 Background

GPU memory hierarchy. The GPU memory hierarchy is structured to optimize data access and computational efficiency, beginning with registers at the core level to the slowest levels of GPU memory. GPU registers are used to store temporary data for warps, providing the quickest access to frequently used values during computation. Next, each SM contains a shared memory, which is a fast, user-managed memory, allowing high-speed data exchange between warps within the same thread block. Each SM also has its own L1 cache, a small, local SRAM, to provide fast access to frequently used data. On advanced GPUs like the A100 and H100, shared memory is dynamically allocated from the L1 cache according to kernel execution requirements.

The L2 cache is shared across all SMs and serves as a large, global cache, typically organized using NUMA architecture since L2 caches are significantly large on modern GPUs. The largest and slowest level of the memory hierarchy is global memory, typically organized using NUMA architecture and composed of high-bandwidth DRAM such as GDDR6, HBM2, or HBM3. However, vendors, including AMD and NVIDIA, do not provide detailed information about the L2 and DRAM NUMA architecture in their public documentation.

Dissecting GPU Architecture. Understanding GPU architecture is attracting more and more attention from researchers as dissecting the details of GPU design aids in fields, including improving GPU application performance and developing novel GPU architectures. Luo et al [28] introduce a benchmarking suite designed to unveil the tensor core performance and programming features of the Hopper architecture. Similar studies have been conducted on other GPU architectures, such as Turing [19], Volta [20], and Ampere [45] architectures. However, these works do not demonstrate

the details of the NUMA architecture and cache coherence protocol of GPUs, which is crucial for a comprehensive analysis GPU performances.

In addition to these works, GPU security researchers have also contributed on dissecting GPU architectures. For example, *Spy in the GPU-box* [16] reveals that memory locations are cached only on the GPU where the memory is allocated, specifically in the Pascal and Volta GPU architectures. INVALIDATE+COMPARE [54] uses cache maintenance instructions to reverse engineer some properties of the cache hierarchy. However, these works primarily focus on attack methodologies, with the dissected aspects of the GPU memory hierarchy serving to support their approach. In contrast, DGNA presents a comprehensive methodology for revealing the full details of the GPU NUMA architecture.

NUMA structure and relative GPU works. Non-Uniform Memory Access (NUMA) is commonly used in many-core systems where memory is divided into multiple nodes, each associated with one or more cores. In NUMA systems, processors can access their local memory more quickly than memory located on other nodes, leading to varying access times according to the distance of memory to the processor.

The NUMA architecture in GPU memory has gained attention from researchers seeking to improve GPU performance. Prior works have explored the methodology to maintain the cache coherence [5–7, 13, 23, 41] and improve the NUMA performance [2, 8, 24, 26, 27, 29, 40, 42, 49, 51–53] on GPUs. Zhao et al. [55] propose the Non-Uniform Bandwidth Architecture to improve the bandwidth of the last-level cache. Milic et al. [29] propose NUMA-aware multi-socket GPUs to minimize the impact of NUMA effects. Runtime methodologies [2, 49] reduce the NUMA overhead via optimizing the page placements. A deeper understanding of the NUMA architecture in commercial GPUs can help researchers contribute to optimizing GPU NUMA systems.

3 Motivation and Challenges

Due to the high demands of GPU workloads, such as the training of large language model [34, 43, 48], GPU memory and caches are becoming larger and larger. Therefore, the introduction of NUMA architecture becomes inevitable due to factors, such as the physical limitations of a single node. However, NUMA introduces imbalanced access latencies between different NUMA nodes, and accessing remote NUMA nodes has a higher latency than that of the local NUMA node. Therefore, numerous works have been proposed to mitigate these NUMA overheads [2, 15, 24, 26, 27, 29, 40, 49, 50, 52]. However, due to the black-box nature of commercial GPUs, the gap between prior NUMA research and the commercial implementation of NUMA in GPUs is implicit, which limits the progress of the community in this area.

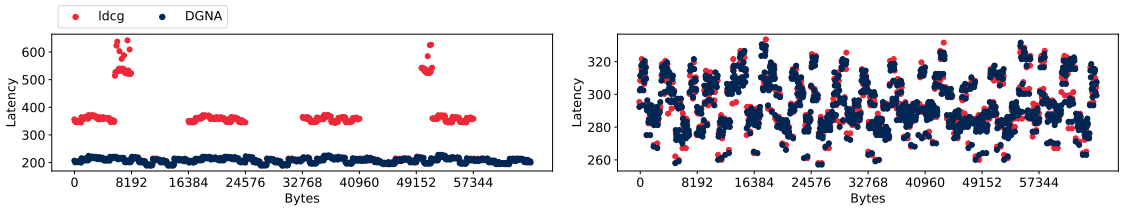
A deep understanding of GPU NUMA architecture is essential across various fields. For example, researchers can leverage knowledge of GPU NUMA architecture to improve the performance of GPU workloads [42, 51, 53]. Furthermore, researchers focusing on GPU NUMA architecture [2, 24, 26, 27, 29, 40, 49, 50, 52] can advance their work based on insights into commercial GPUs. Additionally, GPU simulator [22, 46] can keep pace with the latest GPUs and provide an accurate foundation for researchers to build on. GPU security [16, 54] researchers also require hardware knowledge to design their attack and defense mechanisms.

However, dissecting GPU NUMA architecture poses significant challenges. Modern GPUs feature complex memory hierarchies and virtual memory subsystems, which complicate isolating and testing individual components. For example, to accurately measure L2 cache latency, it is necessary to account for and minimize the influence of memory address translation overhead. Moreover, the data required to be fetched from DRAM into L2 should bypass L1 caches to isolate the L2 cache's specific latency.

Prior works, such as *spy in the GPU-box* [16] and *uncovering real GPU NoC characteristics* [21], rely on specialized load primitives, such as *__ldcg*(), to constrain data accesses to L2 cache and thereby measure L2 latency. However,

Table 1. The corresponding relationship among CUDA code, PTX and SASS instructions

CUDA code	PTX	SASS (A100, H100)
__ldcg(ptr)	ld.global.cg.u8	LDG.E.U8.STRONG.GPU
*ptr	ld.global.u8	LDG.E.U8



(a) A100 (b) H100

Fig. 1. The L2 latencies measured by *__ldcg* [16, 21] and DGNA on the A100 show clear gaps, suggesting that *__ldcg* introduces additional operations for certain L2 memory accesses. For the H100, the similar results indicate that *__ldcg* may directly fetch data from the L2 cache. These findings suggest that vendor-provided intrinsics contain undisclosed implementation details that can impact final results.

these vendor-specific primitives may not be supported on other GPU architectures. Moreover, the PTX and SASS instructions generated by *__ldcg*() differ from those of ordinary memory accesses, as shown in Table 1. Consequently, using the memory instructions generated by *__ldcg*() to analyze the behavior of ordinary memory accesses has inherent limitations, raising questions about the exact feature being measured.

The results, as shown in Figure 1, indicate that the latencies measured by DGNA and *__ldcg* [16] differ on the A100 but are similar on the H100. The difference in latency measurements between *__ldcg* and ordinary instructions lies in the instructions shown in Table 1, with all other is the same. The L2 latencies of DGNA are consistently around 200 cycles as these data blocks are fetched into the local L2 NUMA node by another SM on A100. However, with *__ldcg*, some latencies are concentrated around 350 cycles and some others are around 500 cycles, indicating that these data blocks are either cached in a different L2 NUMA node or still reside in DRAM. We conclude that utilizing ordinary memory instructions offers more reliability.

Moreover, the latency variance in GPU DRAM and L2 cache subsystems can further complicate analysis. For example, DRAM cells require periodic refreshing [30, 31] to maintain data integrity which can interfere with latency measurements and lead to inaccurate conclusions. The DRAM page policy of opening and closing pages also affects the results. Additionally, factors such as TLB misses, temperature fluctuations, and power states of GPUs also introduce noise, skewing latency measurements. To accurately analyze GPU NUMA architectures, a general methodology is required to filter out outliers and ensure reliable results.

4 Methodology

In this section, we present the methodology used to uncover the complexity of GPU NUMA architecture. First, we describe how to measure L2 cache and DRAM latencies without relying on specific intrinsic instructions. Next, we introduce the approach to dissect the NUMA architecture topology. Finally, we explain the techniques used to reveal the read and write mechanisms on GPU NUMA architecture.

```
#define LDCG_TEST_READ 1
#define TEST_READ 2
#define TEST_WRITE 3
template <int MODE>
__global__ void cuTestLat(int smid, int *smSet,int *d0, char * dMem) {
    unsigned int cur_smid;
    __shared__ unsigned char shared[1];
    asm("mov.u32 %0, %smid;" : "=r"(cur_smid));
    //ensuring only executing on the SM ("smid")
    if (smid != cur_smid) return;
    //ensuring only one warp from a single thread block executing
    int old = atomicAdd(&smSet[smid], 1)
    if ( old != 0 ) return;
    if (threadIdx.x == 0) {
        uint64_t start = clock();
        if (MODE == LDCG_TEST_LOAD) {
            shared[0]+=__ldcg(dMem[0]);
        } else if (MODE == TEST_LOAD) {
            shared[0] += dMem[0];
        } else if (MODE == TEST_WRITE) {
            dMem[0] = 1;
            __threadfence();
        }
        d0[0] = clock() - start;
}}
int *d0, *dMem, *smSet;
template<int MODE>
int TestLatency(int smid, int idx) {
    int lat;
    int * addr = &dMem[idx]
    cudaMemset(smSet, 0, NUM_of_SM*sizeof(int));
    cuTestLat<MODE><<<1024,32>>>(smid,smSet,d0,addr);
    cudaMemcpy(&lat, d0, 4, cudaMemcpyDeviceToHost);
    return lat;
}
template<int MODE>
void TestLatencies( vector<int> &l2latencies,  vector<int> &dramlatencies, vector<int> sms) {
    int step = max(DRAMtoL2TransSize,L1CacheLineSize);
    for (auto sm_1 : sms) {
        for (auto sm_2 : sms) {
            malloc(d0, dMem, smSet)
            for (int i = 0 ; i < Size ; i+=step ) {
                //measure DRAM latency and warmup L2.
                int lat1 = TestLatency<MODE>(sm_1,i);
                //measure L2 latency.
                int lat2 = TestLatency<MODE>(sm_2,i);
                dramlatencies.push_back(lat1);
                l2latencies.push_back(lat2);
            }
            cudaDeviceReset();
}}}
```

Listing 1. The code that measures the latency of accessing one byte in L2 cache and DRAM. We use the `malloc` function to allocate GPU memory after resetting the device, ensuring that each measurement does not affect the others.

DGNA assumes that GPUs do not employ hardware prefetching. We argue that future commercial GPUs are unlikely to include prefetchers for several reasons. First, a GPU can be viewed as an execution-driven prefetcher as massive thread-level parallelism allows it to issue memory requests for future accesses early enough to hide DRAM latency, thereby reducing the need for complex prefetching hardware, which is commonly used in CPUs. Second, adding prefetching hardware would increase complexity and reduce the number of cores that can fit on a given die size. Finally, modern GPUs are often bandwidth-limited, and prefetching can introduce unnecessary memory traffic, potentially

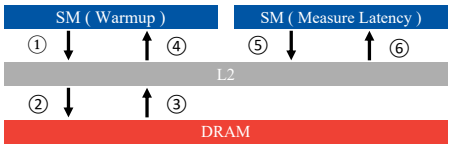


Fig. 2. DGNA leverages the fact that L2 caches are shared among all SMs, while each SM has its own private L1 cache, to measure L2 latency. DGNA uses one SM to warm up the L2 cache and another SM to measure the L2 latency, as the L1 caches are not shared between these two SMs.

degrading performance rather than improving it [10, 35]. Rather than using prefetchers, some GPUs employ the Tensor Memory Accelerator (TMA) [12], a specialized unit that asynchronously issues strided and tiled memory accesses, to improve bandwidth utilization.

4.1 Measuring Latency

Measuring latencies for GPU NUMA architecture poses a significant challenge, as vendors do not provide specific instructions that allow programmers to control which NUMA nodes and memory levels are accessed by memory instructions. Although vendors, such as NVIDIA and AMD, provide intrinsics intended to access data at particular memory levels, the exact hardware implementations of these instructions remain undisclosed, leaving researchers uncertain about the mechanisms underlying them.

Instead of relying on vendor-specific intrinsic instructions, DGNA introduces a new methodology to measure latencies for each level of the memory hierarchy independently of these specific instructions. Figure 2 illustrates how DGNA measures the latencies of L2 and DRAM when accessing the L2 cache and DRAM. First, DGNA employs one SM (*SM (Warmup)*) to fetch data from DRAM into both the L2 cache and its own L1 cache. Since the L2 cache is shared across all SMs, the data resides in L2. DGNA then uses a different SM (*SM (Measuring Latency)*) to issue memory requests and measure the latency of accessing the L2 cache, as each SM has its own private L1 cache.

List 1 presents the code used to analyze the L2 and DRAM latencies. Specifically, DGNA uses one SM to load a data block into the L2 cache and a different SM to subsequently fetch this data block from L2, as each SM maintains its own L1 cache, isolating the access paths and ensuring accurate measurement of latency between cache levels.

The function *cuTestLat*, running on GPUs, measures the memory access latency for the memory address *d_mem* on the particular SM (*smid*). Given that GPU schedulers cannot be directly controlled by programmers, DGNA launches a large number of thread blocks (2048 in our experiments), each containing a single warp. It then filters out thread blocks that are not scheduled on the target SM (Lines 6–10). Because multiple thread blocks may still be scheduled on the same SM, a counter (Lines 12–13) is used to restrict measurements to a single warp from a single thread block. Only thread 0 within the warp reads/writes data (Lines 16-23), which prevents memory requests from other threads from interfering with latency measurements. The fetched value is added to shared memory (*shared*) or *__threadfence* is used to ensure the memory access fully completes.

The function *TestLatency*, running on the CPU, resets the warp counter for each SM (Line 34) before launching *cuTestLat* to measure memory access latency. The *MODE* parameter specifies whether the measurement mode is for read, write, or read operations through vendor-specified intrinsics.

The function *TestLatencies* measures memory latency values across all pairwise combinations of SMs for a sequence of memory addresses as each factor influences latency results. The first call to *TestLatency* in *TestLatencies* (Line 45)

measures the latency of fetching data from DRAM and warmup the L2 cache, while the second call (Line 47) specifically measures the L2 cache latencies. The function *cudaDeviceReset* is called when either *sm_1* or *sm_2* changes to prevent prior measurements from affecting current results. We do not reset the device for each new address to accelerate the measurement process as, based on our observations, the A100 and H100 GPUs lack prefetch mechanisms.

After collecting the dataset using DGNA, we preprocess the data to filter out outliers caused by factors, such as TLB misses, DRAM refreshing. Assume that latencies are only influenced by NUMA architecture, the latency distribution can be modeled as a Gaussian mixture, given by

$$p(\mathbf{lat}) = \sum_{i=1}^{K} \pi_i \mathcal{N}(\mathbf{lat}|\mu_i, \Sigma_i) \tag{1}$$

where lat represents the latencies, $p(lat)$ is the probability density function, K is the number of Gaussian components in the mixture, π_i denotes the proportion of latencies associated with each NUMA node's data block access by SMs, μ_i is the mean latencies for each component, and Σ_i is the variance for each Gaussian component.

Since we assume that NUMA architecture is the primary factor influencing latencies, the means μ_i represent the latencies of different SMs accessing various NUMA nodes, and these may differ. However, we expect the variances Σ_i to be similar across different components, as NUMA architecture is assumed to be the main latency factor. During the analysis, we observe that Σ_0 (the variance for SMs accessing their local NUMA nodes) remains stable, while the variances Σ_i for $i > 0$ (representing remote accesses) are more unstable. This is because other factors tend to increase latencies for memory accesses. Therefore, we use Σ_0 as an approximation for the Σ in other Gaussian components.

Since the probability of a Gaussian distribution for values beyond $3 \times \Sigma$ is small, we remove the outliers by filtering latencies that are larger than $\mu_K + 3 \times \Sigma$ since outliers tend to be very large. After this filtering, we obtain the final latency dataset. This Gaussian mixture model (GMM) separates latencies from SMs accessing different NUMA nodes, providing a framework to filter out outliers without needing knowledge of varying factors, such as TBL misses, DRAM refreshing, that can differ across GPUs.

When scaling beyond two NUMA nodes, the latency distribution may become multimodal rather than bimodal, potentially with higher variance. A potential solution can overcome the higher variance through the following steps. First, remove outliers for each address to reduce measurement noise. Second, combine the filtered measurements into a latency vector, where each dimension corresponds to an address-specific latency. In the high-dimensional latency space, memory regions mapped to different NUMA nodes naturally form distinct clusters due to their structural latency differences. Finally, apply a random projection to map the latency vectors into a scalar space, allowing the GMM to remove the outliers further.

4.2 NUMA Architecture Topology

DGNA first investigates the NUMA architecture hierarchy within the GPU by analyzing the statistical distribution of memory latency values. We derive the latencies of accessing home and remote NUMA nodes and the possibility of sub-NUMA nodes [4] within the DRAM and L2 NUMA node, based on the Gaussian mixture model. We observe that the mapping between allocated virtual pages and NUMA nodes remains stable after resetting the device (Line 50 in Listing 1). We leverage this stable mapping to remove the effects of virtual memory, thereby unveiling the details of the NUMA architecture. The details of this mapping are illustrated in Section 5.3.

Next, we leverage the latency differences when accessing various DRAM, L2 NUMA and L2 sub-NUMA nodes to analyze the GPU memory topology. Figure 3 illustrates the workflow of DGNA for analyzing NUMA architecture

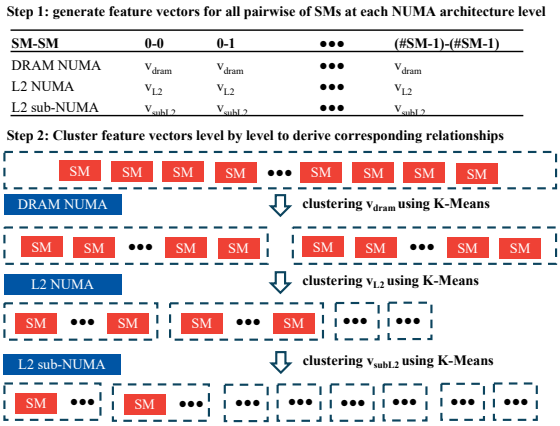


Fig. 3. The workflow for analyzing the NUMA architecture topology. The first step is to generate feature vectors for any pair of SMs accessing data blocks across a sequence of memory addresses at different NUMA levels: DRAM NUMA, L2 NUMA and L2 sub-NUMA. The feature vectors consist of latency values for accessing different memory addresses. The second step is to cluster the feature vectors obtained from DRAM NUMA to L2 sub-NUMA. It maps out the connections within the NUMA architecture, establishing which SMs are associated with specific DRAM, L2 and sub-L2 NUMA nodes.

topology. We first generate feature vectors, which consist of latency values for accessing a sequence of memory addresses with all SM pairs.

Then, DGNA first groups SMs based on the access latencies to DRAM NUMA nodes. This initial grouping is performed since DRAM latencies by each SM can be directly measured, while L2 NUMA nodes that buffer data depend on the SMs that initially access the data, complicating the relationship analysis between SMs and L2.

Next, DGNA groups SMs with their corresponding L2 NUMA nodes, further dividing these SMs into smaller groups. This is feasible as SM pairs on the same local NUMA node exhibit different latency characteristics compared to SM pairs on different L2 NUMA nodes. Additionally, for GPUs with sub-NUMA architecture, DGNA further splits SMs into groups corresponding to L2 sub-NUMA nodes [4]. As observed, the L2 sub-NUMA on the H100 employs a memory interleaving scheme, in which data is mapped to different sub-NUMA nodes according to their addresses. By comparing L2 access latencies from different SMs, as measured using the methodology in Listing 1, we can reconstruct the topology of SMs relative to the L2 sub-NUMA nodes.

DGNA uses the K-Means algorithm [3] to automatically analyze and cluster these feature vectors. By incorporating feature vectors that consist of latency measurements across a sequence of memory addresses, DGNA helps mitigate the impact of this variance, enabling more accurate clustering.

Although the current hardware exposes only two NUMA nodes, our methodology is generic and not restricted to this configuration. We agree that future GPUs may expose more than two NUMA nodes, driven by the increasing demand for memory capacity and cache buffering as GPU workloads continue to scale. For environments with multiple NUMA nodes, DGNA can be naturally extended by using latency vectors instead of scalar latency measurements. Specifically, we can first measure a vector of access latencies from different addresses, which are mapped to different NUMA nodes. Next, we could project the latency vectors into a scalar space using a random projection matrix. Utilizing this methodology, DGNA could be extended to scenarios with more than two NUMA nodes without fundamental changes.

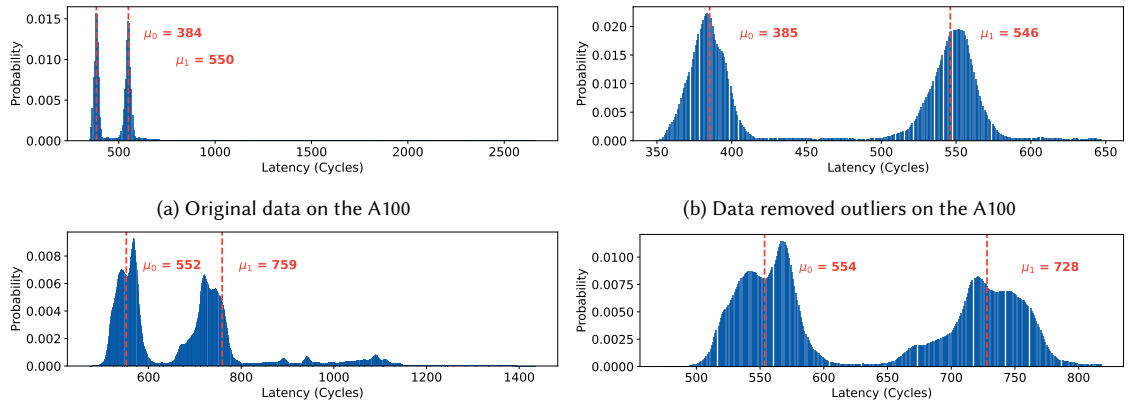
```
#define WARMUP_HOME 1
#define WARMUP_REMOTE 2
#define WARMUP_BOTH 3
#define READ_HOME 1
#define READ_REMOTE 2
#define WRITE_HOME 1
#define WRITE_REMOTE 2
template<int MODE>
void WarmUP() {
    if (MODE == WARMUP_HOME) {
        TestLatency<TEST_READ>(sm_home,0);
    } else if (MODE == WARMUP_REMOTE) {
        TestLatency<TEST_READ>(sm_remote,0);
    } else if (MODE == WARMUP_BOTH) {
        TestLatency<TEST_READ>(sm_home,0);
        TestLatency<TEST_READ>(sm_remote,0);
}}
template<int MODE>
void TestWrite() {
    if (MODE == WRITE_HOME) {
        TestLatency<TEST_WRITE>(sm_home,0);
    } else if (MODE == WRITE_REMOTE){
        TestLatency<TEST_WRITE>(sm_remote,0);
}}
template<int MODE>
void ReadAfterWrite() {
    if (MODE == READ_HOME) {
        TestLatency<TEST_READ>(sm_home,0);
    } else if (MODE == READ_REMOTE) {
        TestLatency<TEST_READ>(sm_remote,0);
}}
template<int WARMMODE, int WRITEMODE, int READMODE>
void Test(int &lat_write, int &lat_raw) {
    Warmup<WARMMODE>();
    lat_write = TestWrite<WRITEMODE>();
    lat_raw = ReadAfterWrite<READMODE>();
}
```

Listing 2. The code to dissect the write mechanisms within GPU's NUMA architecture.

4.3 Read and Write Mechanism on L2 NUMA

The read mechanism on GPUs can be divided into four cases, categorized by the types of SMs used to warm up and test latencies of L2 caches. For a given data block, we define two types of SMs, including (1) *Home-SM*, which is directly connected to the L2 NUMA node, which in turn is directly connected to the DRAM containing the physical page of the data block. (2). *Remote-SM.*, which are not Home-SM for the specific data block. Therefore, we categorize these four types by leveraging Home-SM and Remote-SM for warm-up and measuring the L2 NUMA nodes.

Then we analyze the write mechanism on GPUs. As shown in Listing 2, DGNA tests 12 different cases to measure the write mechanisms. The process involves three steps. (1). L2 cache warmup. DGNA begins by warming up the L2 caches in 3 possible cases: home-SM, remote-SM, or both. (2). Write latency measurement. Once the L2 cache is warmed up, DGNA measures the latencies of writing data to the same memory location. The write operation is tested using two cases: home-SM write or remote-SM write. A longer write latency generally indicates that more NUMA nodes are being updated during the write operation compared to a shorter write latency. 3). Read latency measurement after write (RAW). After the write operation, DGNA tests the latencies of reading the data to analyze which NUMA nodes are updated during the write. Short read latencies indicate that the current NUMA node being accessed is already up to



(a) Original data on the A100

(b) Data removed outliers on the A100

(c) Original data on the H100

(d) Data removed outliers on the H100

Fig. 4. The distribution of access latencies for all SMs on GPUs accessing all DRAM NUMA nodes follows a mixed Gaussian distribution for both the A100 and H100 GPUs. These latencies are clusterred around two central values: 385 cycles (μ_0) and 546 cycles (μ_1) for the A100, and 554 cycles (μ_0) and 728 cycles (μ_1) for the H100 after removing outliers.

date. All these steps help dissect the impact of write operations on the NUMA architecture, revealing how and where data is updated within the GPU memory subsystem.

5 Evaluation

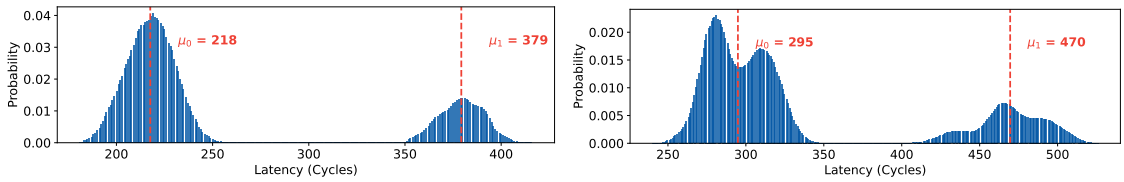
In this section, we describe the findings on the A100 and H100. using DGNA.

5.1 Experimental Setup

We evaluate DGNA on two modern GPU architectures, A100 [32] and H100 [33]. H100 is the most advanced GPU architecture that can be purchased at the time. The configuration details of A100 and H100 are shown in Table 2. Both GPUs are equipped with 5 HBM stacks, with each stack having 2 memory controllers. Furthermore, both A100 and H100 have 2 L2 NUMA nodes. The transaction size from L2 to L1 is 32B on both the A100 and H100. NVIDIA provides an API to configure the default transaction size from DRAM to L2, and for this evaluation, we set it to the default value of 64B on both A100 and H100 [47].

Table 2. The architecture configurations for the GPUs evaluated, including A100 [32] and H100 with SXM5 [33].

GPUs	A100	H100 with SXM5
Architecture	Ampere	Hopper
GPC	7	7 or 8
SM × cores/SM	108 × 64	132 × 128
L2	40MiB	50MiB
L2 NUMA nodes	2	2
Memory	40GiB HBM2	80GiB HBM3
HBM stack	5	5
Memory Controller	10 512-bit	10 512-bit



(a) A100 (b) H100

Fig. 5. The latency distribution for all SMs accessing all L2 NUMA nodes, after removing outliers, follows a mixed Gaussian distribution on both the A100 and H100 GPUs. The access cycles cluster around two central values: 218 cycles (μ_0) and 379 cycles (μ_1) for the A100, and 295 cycles (μ_0) and 470 cycles (μ_1) for the H100. The distribution of latencies on the H100 demonstrates that there are sub-NUMA nodes within the local NUMA node.

5.2 Results of Measuring Latencies

We measures latencies for accessing DRAM and L2 caches. We first demonstrate the distribution of raw data and data after removing outliers on DRAM latencies. Then we analyze the data distribution and give the the average latencies on DRAM, L2 NUMA nodes and L2 sub-NUMA nodes.

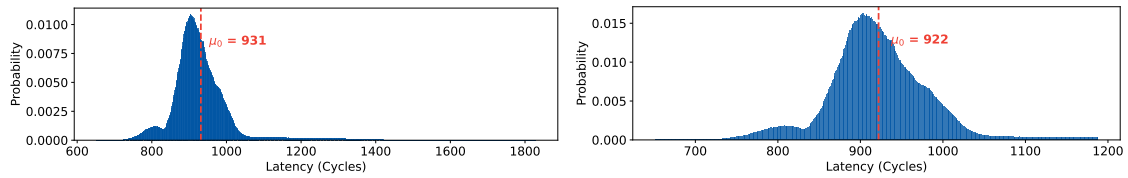
Filtering out outliers. Figure 4a and Figure 4c illustrate the raw latency distributions for DRAM accesses on the A100 and H100, respectively. After filtering out outliers, the refined datasets are shown in Figure 4b and Figure 4d. The raw data on the H100, for example, exhibits some extreme values, leading to a distortion in the mean latencies. These outliers significantly shift the second mean to a higher value, 759 cycles. Although the percentage of outliers is small, their impact on the values is significant. By filtering out these outliers, we obtain more accurate measurements, revealing the true means of accessing each DRAM NUMA node.

DRAM latencies. The DRAM latencies on GPUs exhibit a mixed Gaussian distribution, with two distinct means observed for each architecture: 385 cycles and 546 cycles for the A100, and 554 cycles and 728 cycles for the H100. The mixed Gaussian distribution with two means occurs because, despite the architectural differences, both the Ampere and Hopper GPUs employ a structure with 2 L2 NUMA nodes. Accessing DRAM through a local L2 NUMA node results in lower latencies, while accessing it through a remote L2 NUMA node incurs higher latencies.

L2 latencies. In Figure 5a, we observe that the latency distribution, after removing outliers, for the A100 follows a mixed Gaussian distribution, whose latencies are around 218 cycles and 379 cycles. For the H100, however, the latency distribution diverges from that of the A100, suggesting the presence of sub-NUMA nodes within the L2 cache. The average latencies for accessing local and remote NUMA nodes of the L2 cache on the H100 are 295 cycles and 470 cycles, respectively.

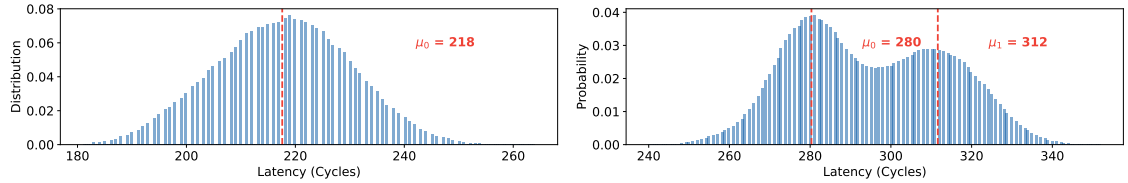
L2 sub-NUMA latencies On the H100, L2 latencies when accessing the local NUMA node show a mixed Gaussian distribution, while the A100 follows a single Gaussian distribution, as shown in Figure 7. For the H100, the memory latency values exhibit two mean values, 280 cycles and 312 cycles. We attribute this to the larger L2 cache size of the H100 (50 MiB) compared to the A100 (40 MiB), leading NVIDIA to implement a sub-NUMA microarchitecture within the local NUMA node on H100. Additionally, the latency overhead for accessing a remote sub-NUMA node is minimal, approximately 32 cycles.

RTX 5090 We further apply DGNA to the RTX 5090, which is a desktop GPU based on the Blackwell architecture and released in January 2025. As shown in Figure 6, DGNA reveals that the RTX 5090 employs a single L2 cache partition,



(a) Original data on the RTX 5090. (b) Data removed outliers on the RTX 5090

Fig. 6. The distribution of access latencies for all SMs on GPUs accessing all DRAM NUMA nodes follows a Gaussian distribution for RTX 5090. These latencies are clusterred around a single central value: 922 cycles for the RTX 5090 after removing outliers.



(a) A100 (b) H100

Fig. 7. The latency distribution for all SMs accessing their local L2 NUMA nodes on the H100 follows a mixed Gaussian distribution, while on the A100 follows a Gaussian distribution. The mean latency values are 280 cycles (μ_0) and 312 cycles (μ_1), respectively, representing the two main latency groups within the local L2 NUMA access pattern for L2 sub-NUMA nodes on the H100.

in contrast to the H100 and A100, which feature multiple L2 partitions. In addition, DGNA enables more accurate measurement of DRAM latencies by effectively filtering out outliers.

5.3 NUMA Architecture Topology

We next investigate the collected latency data to reveal the NUMA architecture and the correlations between DRAM and L2 NUMA nodes with the SMs on the A100 and H100.

DRAM NUMA Architecture. We derive the relationship between GPU cores (SMs) and DRAM NUMA nodes on both A100 and H100 GPUs. Our analysis demonstrates two different patterns in relation to DRAM NUMA nodes.

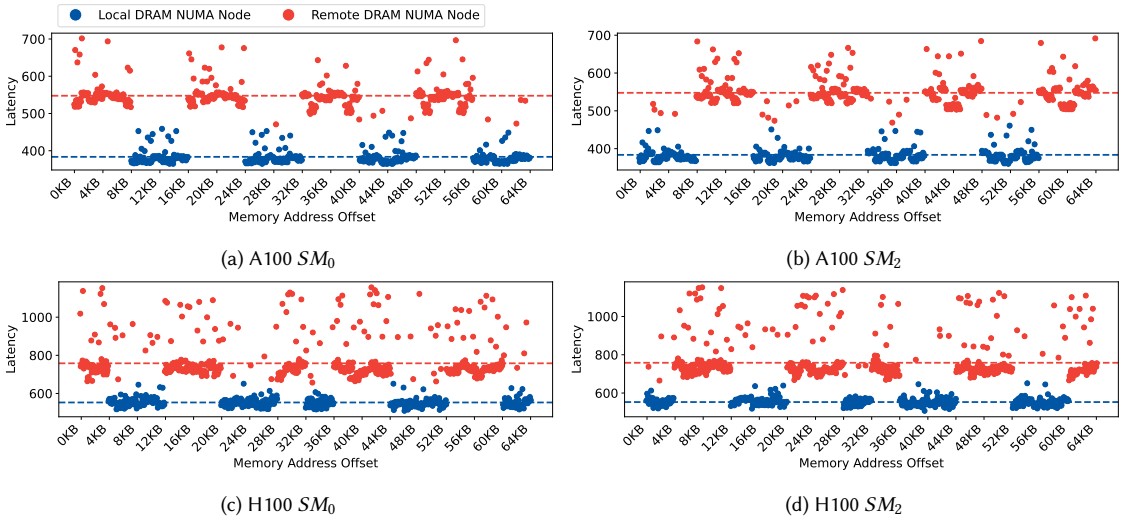
Figure 8 demonstrates the NUMA allocation mechanisms on the A100 and H100. Each SMs on GPUs that access data from their local DRAM NUMA node exhibit lower memory latency values compared to those accessing data from a remote NUMA node. On the A100, the allocation alternates every 8KB, where each 8KB portion corresponds to a specific NUMA node. On the H100, The NUMA memory is allocated based on a formula that determines the allocation pattern by address offset. The formula is $ID_{NUMA} = \lfloor \frac{x+1-H(x-8)}{2} \rfloor mod\ 2$ where $x = \lfloor \frac{address\ mod\ 64KB}{4KB} \rfloor$ and H(x-8) is the Heaviside step function $H(x-8) = \begin{cases} 1, & x \geqslant 8 \\ 0, & x < 8 \end{cases}$.

Furthermore, we group DRAM NUMA portions and the corresponding GPU cores using K-Means and the feature vectors. Table 3 demonstrates that on the A100, 46 SMs are assigned to DRAM NUMA node 0 and 62 SMs are assigned to DRAM NUMA node 1. On the H100, both DRAM NUMA nodes have 66 corresponding SMs.

This result demonstrates the differences between the A100 and H100 in handling die manufacturing errors. NVIDIA employs the floorsweeping technique [9] to salvage imperfect dies by disabling defective hardware blocks. According to NVIDIA's official documentation [32, 33], the GA100 full GPU consists of 128 SMs, whereas the GH100 full GPU

Table 3. The corresponding relationship between GPU cores and NUMA memory partitions for A100 and H100. We utilize ID_{SM} to represent the SM ID. On the A100, we summarize the distribution of SMs using $\mathbf{X = ID_{SM} \mod 14}$. On the H100, we use $\mathbf{Y = ID_{SM} \mod 16}$ when $\mathbf{ID_{SM} < 62}$, and $\mathbf{Z = (ID_{SM} - 62) \mod 14}$ when $\mathbf{62 \le ID_{SM} < 120}$. The number of corresponding SMs for each NUMA node is 62 and 46 for the A100, which is imbalanced, whereas it is 66 and 66 for the H100, which is balanced.

GPUs	DRAM NUMA Node 0	DRAM NUMA Node 1
A100	$0 \le X < 2$ $6 \le X < 12$	$2 \le X < 6$ $12 \le X < 14$
H100	$0 \le Y < 2$ $8 \le Y < 14$ $8 \le Z < 14$ $120 \le ID_{SM} < 122$ $124 \le ID_{SM} < 132$	$2 \le Y < 8$ $14 \le Y < 16$ $0 \le Z < 8$ $122 \le ID_{SM} < 124$



(a) A100 SM_0 (b) A100 SM_2

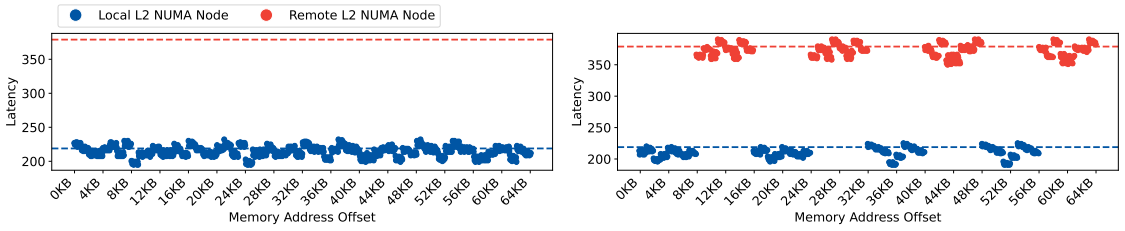
(c) H100 SM_0 (d) H100 SM_2

Fig. 8. We measure the latency feature vectors by accessing consecutive memory addresses through DRAM NUMA nodes. SM_i indicates the SM whose ID is i. The dots are marked as blue if they are close to the average latency of the local DRAM NUMA node, and as red if they are close to the average latency of the remote DRAM NUMA node.

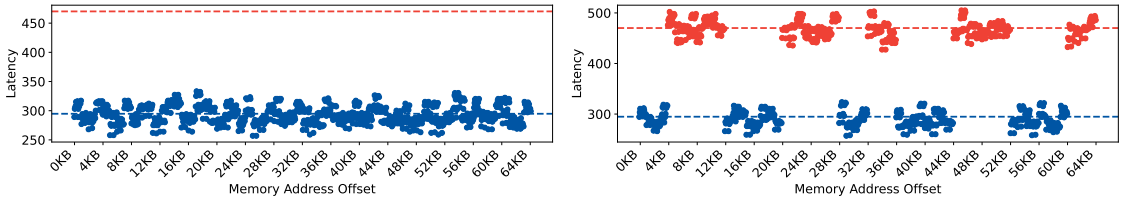
features 144 SMs. The A100 appears to combine two dies with different defective rates, while the H100 utilizes two dies with similar rates. This implies that NVIDIA improved the technique on the H100 to reduce the die defective rate compared to the A100.

L2 NUMA architecture. We then utilize the feature vectors of L2 NUMA nodes to further cluster groups. However, we find that the results are identical to those obtained from clustering based on DRAM NUMA nodes. We conclude that L2 and DRAM are directly connected, and the information transfer between them occurs through the connections between L2 NUMA nodes. As a result, L2 NUMA nodes cannot further split SMs into smaller groups.

Another piece of evidence supporting this conclusion is the latency gap between fetching data from the local L2 NUMA node versus a remote NUMA node, which is 161 cycles and 174 cycles for the A100 and H100, respectively. Meanwhile, the overhead for SMs accessing remote L2 NUMA nodes is 161 cycles for the A100 and 175 cycles for the

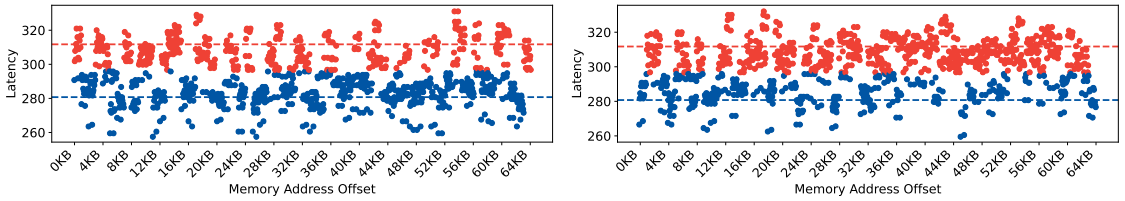


(a) A100: using the SM on the same NUMA node to warmup. (b) A100: using the SM on the different NUMA node to warmup.



(c) H100: using the SM on the same NUMA node to warmup. (d) H100: using the SM on the different NUMA node to warmup.

Fig. 9. Accessing consecutive memory addresses to derive the L2 NUMA node information. For measuring L2 latency within the same NUMA node, SM_0 fetches data into the L2 cache and then SM_1 is used to measure L2 latencies. To assess latency across different NUMA nodes, SM_0 fetches the data, but SM_2 is used to measure L2 latencies. The dots are marked as blue if they are close to the local L2 NUMA latency, and as red if they are close to the remote L2 NUMA latency.



(a) using the SM on the same sub-NUMA node to warmup. (b) using the SM on the different sub-NUMA node to warmup.

Fig. 10. The sub-NUMA information is obtained by accessing consecutive memory addresses using different SMs to warm up and measure L2 latencies. Our findings reveal that the H100 contains sub-NUMA L2 nodes within each L2 local NUMA node. We mark the dots as blue if they are close to the average latency of the local sub-NUMA node, and as red if they are close to the average latency of the remote sub-NUMA node. The results indicate that the first 2KB of memory addresses exhibit high latency in (a) but low latency in (b), followed by alternating between these two patterns

H100. This Network-on-Chip (NoC) overhead is similar to the L2 and DRAM latency for SMs accessing remote NUMA nodes, suggesting that the same wires are used for both.

L2 sub-NUMA architecture. Figure 10 presents the results of using SM_0 to warm up the L2 cache, followed by measuring L2 access latency from both SM_0 and SM_8. Since the L1 cache is flushed after kernel execution, all these measurements reflect L2 latency. Specifically, the latency to access the local sub-NUMA node is lower than that for accessing the remote sub-NUMA node, indicating that SM 0 and SM 8 are connected to different sub-NUMA nodes.

We further cluster SMs into four groups based on their relationship with sub-NUMA nodes, as shown in Table 4. Next, we use the thread block cluster on the H100 to analyze the SM IDs. Our analysis reveals that SMs connected to the same sub-NUMA node always belong to two GPU processing clusters (GPCs), indicating that two GPCs are

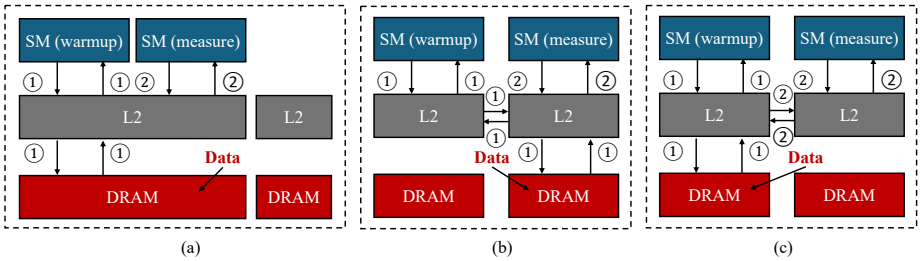


Fig. 11. The mechanism of GPUs to read data on L2 NUMA includes 3 cases: **(a).** The home-SM warms up the local NUMA node, and another home-SM accesses it. **(b).** The remote-SM warms up both the local and remote NUMA nodes, and the home-SM accesses it. Remote-SM with remote-SM is similar to this case. **(c).** The home-SM warms up only the local NUMA node, and the remote-SM accesses it.

Table 4. The corresponding relationship between SMs and NUMA/sub-NUMA nodes on the H100 reveals that SMs connected to a sub-NUMA node belong to one of the two GPCs. This indicates that GPCs are the fundamental units responsible for connecting to sub-NUMA nodes.

NUMA	sub-NUMA	#SM	GPC	SM ID
NUMA 0	sub-NUMA 0	32	GPC_0	2, 3, 18, 19, 34, 35, 50, 51, 64, 65, 78, 79, 92, 93, 106, 107.
			GPC_1	4, 5, 20, 21, 36, 37, 52, 53, 66, 67, 80, 81, 94, 95, 108, 109.
	sub-NUMA 1	34	GPC_2	6, 7, 22, 23, 38, 39, 54, 55, 68, 69, 82, 83, 96, 97, 110, 111.
			GPC_3	14, 15, 30, 31, 46, 47, 62, 63, 76, 77, 90, 91, 104, 105, 118, 119, 122, 123.
NUMA 1	sub-NUMA 2	34	GPC_4	0, 1, 16, 17, 32, 33, 48, 49, 124, 125, 126, 127, 128, 129, 130, 131.
			GPC_5	12, 13, 28, 29, 44, 45, 60, 61, 74, 75, 88, 89, 102, 103, 116, 117, 120, 121.
	sub-NUMA 3	32	GPC_6	8, 9, 24, 25, 40, 41, 56, 57, 70, 71, 84, 85, 98, 99, 112, 113.
			GPC_7	10, 11, 26, 27, 42, 43, 58, 59, 72, 73, 86, 87, 100, 101, 114, 115.

connected to each sub-NUMA node on the H100. Table 4 also illustrates that SMs within each L2 NUMA node are still imbalanced, 32 or 34 SMs per sub NUMA node. We conclude that this imbalance arises because GPCs are composed of texture processing clusters, each of which contains two SMs. As a result, no sub-NUMA node can be connected to exactly 33 SMs.

5.4 Read and write mechanisms on the L2 NUMA

We then analyze the read and write mechanism on the L2 NUMA. Using the NUMA allocation results in Section 5.3, we can control the home, remote SM, L2 and DRAMs.

Read operations. As shown in Figure 9, Since SM_2 resides on the same L2 NUMA node as SM_1, their L2 access latencies are similar, both being close to 218 cycles for the A100 and 295 cycles for the H100.

Moreover, we observe that L2 access latency varies across addresses when accessed by SM_2. Comparing these results with Figure 8, we find that the L2 access latency matches that of a local L2 NUMA node when the SM warms up the L2 cache using data from a remote DRAM NUMA node. In contrast, when the data originates from the local DRAM NUMA node, the observed L2 access latency is closer to that of a remote L2 NUMA node.

We conclude that the read mechanism on L2 NUMA of both the A100 and H100 operate as shown in Figure 11. There are 3 cases for L2 NUMA nodes reading data. (1). As shown in Figure 11a, the home-SM warms up only the local L2 NUMA node, and another home-SM accesses it, resulting in L2 access latency similar to that of accessing

the local NUMA node. (2). As shown in Figure 11b, the home-SM warms up only the local NUMA node, and the remote-SM accesses it, resulting in L2 access latency similar to that of accessing the remote NUMA node. (3). As shown in Figure 11c, the remote-SM warms up both the local and remote NUMA nodes, the latency of both remote-SM and home-SM accessing data block is similar to that of accessing the local NUMA node.

We also notice that, on both the A100 or H100, the L1 cache is flushed after the kernel finishes executing, but the data inside the L2 NUMA node remains. When utilizing the same SM to warm up and measure data, the results are similar to those of accessing the local L2 NUMA node.

Table 5. The results of Listing 2 measure the write mechanism of GPUs on L2 NUMA nodes. The warmup phase includes three different cases: warmup by home-SM, remote-SM, and both. We test writes by home-SM and remote-SM under after warmup. Subsequently, we measure the latency of read-after-write for both home-SM and remote-SM. The results reveal four patterns on both the A100 and H100, indicating that home-SM always writes to the local NUMA node if there is a cache hit. Remote-SM only updates the remote-NUMA if and only if the remote-NUMA cache hits, otherwise it updates all L2 NUMA nodes.

Warmup	Home-SM		Remote-SM		Both	
Write	Home-SM	Remote-SM	Home SM	Remote-SM	Home-SM	Remote-SM
A100-Write	281	425	385	407	411	446
A100-RAW-Home-SM	223	209	223	365	205	208
A100-RAW-Remote-SM	388	205	384	206	366	205
H100-Write	426	595	502	591	556	594
H100-RAW-Home-SM	281	299	279	472	301	297
H100-RAW-Remote-SM	444	295	447	295	473	298

Write operations. The results, as shown in Table 5, present the write latency for different cases on the A100 and H100. These cases include: (a) whether the data is inside all L2 NUMA nodes or only in the local or remote L2 NUMA node, and (b) the location of the data inside DRAM, either in the SM's local DRAM NUMA node or in the remote DRAM NUMA node. Additionally, Table 5 presents the read latencies following a write, considering two different cases on the A100 and H100: (a) reading using the home SM, and (b) reading using the remote SM.

In the case of *Home-SM (Warmup)-Home-SM (Write)*, the home SM fetches data only into the local NUMA node, resulting in the lowest write latency, as it only needs to update the local NUMA node. When the home SM reloads the data from the local NUMA node, the latency remains low for Home-SM (RAW). However, for Remote-SM (RAW), the remote NUMA node does not buffer this cache block, leading to a high latency for the remote SM. For *Home-SM (Warmup)-Remote-SM (Write)*, the remote SM updates all L2 NUMA nodes. Subsequently, when SMs reload the data from the L2 caches, the latency remains low for both Home-SM (RAW) and Remote-SM (RAW).

In the *Remote-SM (Warmup)-Home-SM (Write)* case, the remote SM fetches data into both the remote and local NUMA nodes. The home SM updates only the local NUMA node, invalidating the remote NUMA node. Thus, Home-SM (RAW) latency remains low. For Remote-SM (RAW), its latency is high as the remote NUMA node does not buffer the latest cache block. For *Remote-SM (Warmup)-Remote-SM (Write)*, the remote SM updates only the remote L2 NUMA nodes. When the home SM reloads the data, it requires access to the remote L2 NUMA, resulting in high latency for Home-SM (RAW). However, Remote-SM (RAW) latency is low, as the L2 cache buffers the most recent data block.

For *Both (Warmup)-Home-SM (Write)*, the write operation only updates home NUMA node. But for *Both (Warmup)-Remote-SM (Write)*, both remote and home NUMA nodes are updated. The difference between *Remote (Warmup)-Remote-SM (Write)* and *Both (Warmup)-Remote-SM (Write)* is because when the remote SM first touches the data, the primary

NUMA node for the data block is maintained in the remote SM's L2 node, despite also buffering data into the home L2 NUMA node [17, 25]. However, in *Both (Warmup)-Home-SM (Write)*, the primary NUMA node is the home L2.

6 Conclusion

In this work, we present DGNA, a framework to uncover the NUMA architecture within the GPU memory hierarchy through microbenchmarking and data analysis. DGNA independently measures L2 and DRAM latencies without relying on vendor-specific instructions, ensuring accurate metrics by using a Gaussian mixture model to filter outliers. Our findings identify the presence of sub-NUMA nodes on the H100. Additionally, we uncover the read and write mechanism on the L2 NUMA nodes, which employs different mechanisms across home and remote NUMA nodes for home and remote SMs. DGNA will be open-sourced to support continued research.

Acknowledgments

We would like to thank everyone who took the time to provide detailed suggestions that help to improve this work. In particular, we would like to acknowledge the help of Yifan Sun and Alen Sabu. We also thank the anonymous reviewers for their constructive feedback. This project is generously supported by A*STAR under the RIE2025 Energy-aware Accelerated Computing (EAC) program (Award H25-MSR3439), the RIE2025 Innovation & Enterprise (I&E) Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) (Award H25-MCP3442), the Ministry of Education, Singapore, Tier 3 (Award MOE-MOET32024-0003), the NUS Artificial Intelligence Institute (NAII) (Award NAII-SF-2024-004), the Guangdong Municipal Science and Technology Project (Nos. 2024QN11X196) and the Guangdong Basic and Applied Basic Research Foundation (Nos. 2025A1515110151).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

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

[3] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. 2020. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics* (2020).

[4] Christie L Alappat, Johannes Hofmann, Georg Hager, Holger Fehske, Alan R Bishop, and Gerhard Wellein. 2020. Understanding HPC benchmark performance on Intel Broadwell and Cascade Lake processors. In *International Conference on High Performance Computing (ISC)*.

[5] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *International Symposium on Computer Architecture (ISCA)*. 439–451.

[6] Johnathan Alsop, Marc S Orr, Bradford M Beckmann, and David A Wood. 2016. Lazy release consistency for GPUs. In *International Symposium on Microarchitecture (MICRO)*. 1–14.

[7] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *International Symposium on Computer Architecture (ISCA)*.

[8] 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 *International Symposium on Computer Architecture (ISCA)*.

[9] Joshua Bakita and James H Anderson. 2023. Hardware compute partitioning on NVIDIA GPUs. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[10] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadat, and Onur Mutlu. 2022. Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction. In *International Symposium on Microarchitecture (MICRO)*. 1–18.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.

[12] Jack Choquette. 2022. Nvidia hopper gpu: Scaling performance. In *Hot Chips 34 Symposium (HCS)*.

[13] Preyesh Dalmia, Rajesh Shashi Kumar, and Matthew D Sinclair. 2024. CPElide: Efficient Multi-Chiplet GPU Implicit Synchronization. In *International Symposium on Microarchitecture (MICRO)*.

[14] AC Damianou, Carl Henrik Ek, MK Titsias, and ND Lawrence. 2012. Manifold relevance determination. In *International Conference on Machine Learning (ICML)*.

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

[16] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2023. Spy in the GPU-box: Covert and side channel attacks on multi-GPU systems. In *International Symposium on Computer Architecture (ISCA)*.

[17] Antonio Franques, Apostolos Kokolis, Sergi Abadal, Vimuth Fernando, Sasa Misailovic, and Josep Torrellas. 2021. Widir: A wireless-enabled directory cache coherence protocol. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[18] Yicheng Gu, Yun Wang, Yunfan Sun, Yuxin Xiang, Xuyan Hu, Zhengwei Qi, and Haibing Guan. 2024. {gVulkan}: Scalable {GPU} Pooling for {Pixel-Grained} Rendering in Ray Tracing. In *USENIX Annual Technical Conference (USENIX ATC)*.

[19] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[20] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[21] Zhixian Jin, Christopher Rocca, Jiho Kim, Hans Kasan, Minsoo Rhu, Ali Bakhoda, Tor M Aamodt, and John Kim. 2024. Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture. In *International Symposium on Microarchitecture (MICRO)*.

[22] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486.

[23] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building heterogeneous unified virtual memories (uvms) without the overhead. *Transactions on Architecture and Code Optimization (TACO)* (2016).

[24] Jiwon Lee, Ju Min Lee, Yunho Oh, William J Song, and Won Woo Ro. 2023. Snakebyte: A tlb design with adaptive and recursive page merging in gpus. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[25] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1990. The directory-based cache coherence protocol for the DASH multiprocessor. In *International symposium on Computer Architecture (ISCA)*.

[26] Bingyao Li, Yueqi Wang, and Xulong Tang. 2023. Orchestrated scheduling and partitioning for improved address translation in gpus. In *Design Automation Conference (DAC)*. IEEE, 1–6.

[27] Bingyao Li, Jieming Yin, Anup Holey, Youtao Zhang, Jun Yang, and Xulong Tang. 2023. Trans-fw: Short circuiting page table walk in multi-gpu systems via remote forwarding. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[28] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and Dissecting the Nvidia Hopper GPU Architecture. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

[29] 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 *International Symposium on Microarchitecture (MICRO)*.

[30] Janani Mukundan, Hillery Hunter, Kyu-Hyoun Kim, Jeffrey Stuecheli, and José F Martínez. 2013. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. In *International Symposium on Computer Architecture (ISCA)*.

[31] Prashant Nair, Chia-Chen Chou, and Moinuddin K Qureshi. 2013. A case for refresh pausing in DRAM memory systems. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[32] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[33] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core.

[34] OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[35] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *International Symposium on Computer Architecture (ISCA)*.

[36] Seonghyun Park, Adil Ahmad, and Byoungyoung Lee. 2020. Blackmirror: Preventing wallhacks in 3d online fps games. In *ACM Conference on Computer and Communications Security (CCS)*. 987–1000.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Conference on Neural Information Processing Systems (NeurIPS)*. 8024–8035.

[38] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. 2016. Improving mobile gaming performance through cooperative CPU-GPU thermal management. In *Proceedings of the annual design automation conference (DAC)*.

[39] Xiaowei Ren and Mieszko Lis. 2021. Chopin: Scalable graphics rendering in multi-gpu systems via parallel image composition. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[40] 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 *International Symposium on High Performance Computer Architecture (HPCA)*.

[41] 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 *International Symposium on High Performance Computer Architecture (HPCA)*.

[42] Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, Abhinav Vishnu, Dipanjan Sengupta, and Xu Liu. 2018. Numa-caffe: Numa-aware deep learning neural networks. *Transactions on Architecture and Code Optimization (TACO)* (2018).

[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning (ICML)*.

[44] Guoli Song, Shuhui Wang, Qingming Huang, and Qi Tian. 2017. Multimodal Gaussian process latent variable models with harmonization. In *International Conference on Computer Vision (ICCV)*.

[45] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2023. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *Transactions on Parallel & Distributed Systems (TPDS)* (2023).

[46] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David R. Kaeli. 2019. MGPUSim: enabling multi-GPU performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. 197–209.

[47] G Thomas-Collignon and V Mehta. 2020. Optimizing cuda applications for nvidia a100 gpu. In *NVIDIA GPU Technology Conference*.

[48] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* (2023). arXiv:2302.13971

[49] Yueqi Wang, Bingyao Li, Aamer Jaleel, Jun Yang, and Xulong Tang. 2024. GRIT: Enhancing Multi-GPU Performance with Fine-Grained Dynamic Page Placement. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[50] Chenhao Xie, Xin Fu, Mingsong Chen, and Shuaiwen Leon Song. 2019. OO-VR: NUMA friendly object-oriented VR rendering framework for future NUMA-based multi-GPU systems. In *International Symposium on Computer Architecture (ISCA)*.

[51] Chenhao Xie, Fu Xin, Mingsong Chen, and Shuaiwen Leon Song. 2019. OO-VR: NUMA friendly object-oriented VR rendering framework for future NUMA-based multi-GPU systems. In *International Symposium on Computer Architecture (ISCA)*.

[52] 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 *International Symposium on Microarchitecture (MICRO)*.

[53] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Symposium on principles and practice of parallel programming (PPoPP)*.

[54] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. 2024. {Invalidate+ Compare}: A {Timer-Free}{GPU} Cache Attack Primitive. In *USENIX Security Symposium (USENIX Security)*.

[55] Xia Zhao, Magnus Jahre, Yuhua Tang, Guangda Zhang, and Lieven Eeckhout. 2023. NUBA: Non-uniform bandwidth GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Received 22 July 2025; revised 23 February 2026; accepted 22 July 2026