NetCrafter: Tailoring Network Traffic for Non-Uniform Bandwidth Multi-GPU Systems

Amel Fatima
University of Virginia
Charlottesville, Virginia, USA
af3szr@virginia.edu

Yang Yang
University of Virginia
Charlottesville, Virginia, USA
yangyang@virginia.edu

Yifan Sun
William & Mary
Williamsburg, Virginia, USA
ysun25@wm.edu

Rachata Ausavarungnirun*
MangoBoost Inc.
Skokie, Illinois, USA
r.ausavarungnirun@gmail.com

Adwait Jog
University of Virginia
Charlottesville, Virginia, USA
ajog@virginia.edu

Abstract

Multiple Graphics Processing Units (GPUs) are being integrated into systems to meet the computing demands of emerging workloads. To continuously support more GPUs in a system, it is important to connect them efficiently and effectively. To this end, emerging multi-GPU systems are adopting a hierarchical approach – a group of GPUs with high affinity are connected with higher-bandwidth networks, while multiple groups of GPUs are connected with lower-bandwidth networks to support the scaling of GPUs. Unfortunately, such a non-uniform bandwidth configuration leads to significant performance bottlenecks, especially across lower-bandwidth networks. We present NetCrafter, a combination of novel approaches to deal with the network traffic. NetCrafter is based on three observations: a) not all flits in the network fully utilize the network bandwidth, b) not all requested flits are even necessary – they are requested in the hope that their data might be useful later, c) some flits are more latency-sensitive than others and must be prioritized in the network. NetCrafter leverages these observations to reduce the network traffic by stitching compatible flits that are partly filled, and trimming the number of flits by not fetching flits that are unnecessary. NetCrafter also effectively manages network traffic by sequencing flits such that latency-sensitive flits reach their destinations faster. Although our proposed techniques are generic and can be applied to any network, they are especially useful in alleviating the bottlenecks presented by lower-bandwidth networks connecting multiple groups of GPUs. Overall, NetCrafter significantly improves multi-GPU performance, thereby contributing to efficient scaling of GPU-based systems.

CCS Concepts

• Computer systems organization → Single instruction, multiple data; **• Networks → Network performance analysis**.

*The author contributed to this work while he was at KMUTNB in Thailand.

Keywords

GPUs, Virtual Memory, Bandwidth, Performance

ACM Reference Format:
Amel Fatima, Yang Yang, Yifan Sun, Rachata Ausavarungnirun, and Adwait Jog. 2025. NetCrafter: Tailoring Network Traffic for Non-Uniform Bandwidth Multi-GPU Systems. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731040

1 Introduction

Graphics Processing Units (GPUs) are widely used in modern computing systems to accelerate performance for a wide range of applications [14, 38, 47, 76, 92]. However, scaling the performance of a single GPU is becoming increasingly challenging. This difficulty stems primarily from limitations in transistor scaling and the growing complexity of large die sizes [5, 22, 39, 84, 94]. To sustain continued performance scaling, researchers have considered two primary architectural strategies to scale-up the GPU systems: multi-chip modules (MCM) [5, 29, 61, 71, 98], which integrate multiple GPU chiplets within a single package, and multi-GPU systems, which connect discrete GPUs via networks with varying bandwidth configurations [26, 30, 31, 33, 62, 66, 83, 86, 90].

Both MCM and multi-GPU strategies are complementary and increasingly combined in modern systems using hierarchical networks to create large, unified GPU complexes [25, 26, 33, 83]. For instance, Figure 1 shows the Frontier system [6, 26], where tightly integrated AMD GPU chiplets (or simply referred as "GPU" in the rest of the paper) within a cluster (e.g., GPU 0 and GPU 1) communicate over higher-bandwidth networks, while chiplets across clusters are connected via lower-bandwidth networks. This results in inherent bandwidth non-uniformity—a design trend likely to persist—where high-affinity GPUs enjoy fast communication, and communication between distant GPU clusters is slow. NVIDIA's GPU-based Summit [83, 90] and Intel's GPU-based Aurora node [25] are also examples of multi-GPU-based HPC nodes connected by networks with varying bandwidth configurations.

Such multi-GPU architectures deliver high theoretical computing throughput, but are hindered by communication bottlenecks between GPUs connected over lower-bandwidth networks, a challenge absent in single-GPU setups. These bottlenecks can severely

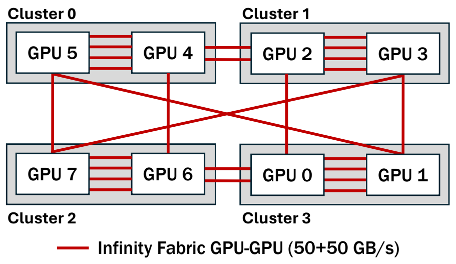

Figure 1: AMD GPUs in a Frontier node. GPUs within the same cluster are connected with Infinity Fabric GPU-GPU with a peak bandwidth of 200 GB/s. GPUs in different clusters are connected with Infinity Fabric GPU-GPU, where the peak bandwidth ranges from 50-100 GB/s.

limit application performance by introducing non-uniform memory access (NUMA) overheads arising from data sharing and communication across GPUs [25, 33, 65, 83, 90] connected over the lower-bandwidth networks. Optimizing communication efficiency among different clusters is, therefore, critical for enhancing the performance of large-scale workloads. Our goal in this work is to effectively reduce and manage network traffic across the lower-bandwidth networks connecting the GPUs. We make several observations to achieve this goal: (a) not all flits (which represent the amount of data transferred per flow-control unit) fully utilize the available network bandwidth, (b) some flits are fetched as part of the same cache block but are not always needed, and (c) certain flits are more latency-sensitive and require prioritization. Based on these observations, we introduce NetCrafter, which reduces network traffic using two key techniques: 1) *Stitching* and 2) *Trimming*, and effectively manages the network traffic with a third technique: 3) *Sequencing*. Below, we outline our high-level strategies:

Stitching Network Traffic. To minimize network traffic, we propose our first technique, *Stitching*, which we apply to the lower-bandwidth network. By analyzing the network traffic, we identified that many flits contain empty bytes because the packet size is not always a multiple of the flit size. Many flits are even less than half full, causing a significant waste of bandwidth. To this end, we propose to stitch (i.e., combine) multiple flits' useful information (e.g., payloads, header) to fewer flits when possible, thereby reducing the overall number of flits and improving bandwidth utilization.

Selecting the flits to stitch has a significant performance implication and needs careful design. We take three approaches, including: (1) Traffic category-facilitated candidate selection. We categorize the network traffic into six distinct categories and find that packets of different categories (e.g., read response, page table response) usually have different packet sizes. Continuously stitching flits across certain categories can yield high utilization. (2) Flit Pooling. Flit Pooling delays the ejection of a flit into the network for an appropriate number of cycles, waiting for a suitable stitching candidate to arrive. (3) Prioritizing latency-sensitive packets. To prevent increased latency of Flit Pooling from negatively impacting certain applications, we identify latency-critical flits and exclude them from Flit Pooling.

Trimming Network Traffic. A major part of the inter-GPU data access happens between the L1 and the L2 cache (we assume L2 cache is shared across GPUs [9, 24, 42, 71]). The current cache design fetches entire cache lines to the L1 cache even if only a few bytes are needed, wasting bandwidth. We take advantage of the fact that, for many applications, GPU wavefronts (groups of 64 adjacent GPU threads) typically request fewer than 16 bytes of cache line data at a time. This insight allows us to trim parts of the network packet that carries data in lower-bandwidth networks, saving network bandwidth and reducing network traffic. Note that we only trim when the request has to traverse the lowest-bandwidth network. This ensures that we reduce network flits without significantly degrading L1 cache performance.

Sequencing Network Traffic. Not all packets have the same level of criticality in terms of performance impact. Our experiments revealed that end-to-end execution time is sensitive to the latency of page table walks (PTWs). This is because PTWs require fetching virtual-to-physical memory translations from the page table, placing these look-ups on the critical path of data accesses [7, 28, 44, 68, 81]. Moreover, we found that PTW-related packets (e.g., page table requests and responses) account for only a small fraction (on average 13%) of the total data traversing the lower-bandwidth network connecting the GPUs. These latency-critical PTW-related packets often get blocked behind data packets, degrading overall application performance. Based on these insights, we propose sequencing the network traffic by prioritizing PTW-related flits that stall read accesses over the lower-bandwidth network. This selective prioritization prevents data flits from blocking latency-critical PTW-related flits, allowing them to pass through lower-bandwidth networks smoothly and facilitating faster completion of PTWs, thus improving performance.

We summarize the contributions of this work as follows:

- We conduct an in-depth performance evaluation of a non-uniform bandwidth multi-GPU setup, demonstrating that bandwidth non-uniformity creates significant performance bottlenecks, particularly due to the constraints of lower-bandwidth networks.
- We introduce NetCrafter, a novel approach designed to address network traffic inefficiencies in multi-GPU systems, particularly those constrained by lower-bandwidth networks. NetCrafter is built on three key observations: (a) not all flits fully utilize available network bandwidth, b) not all requested flits are even necessary – they are requested in the hope that they might be useful later, and (c) certain flits are more latency-sensitive and require prioritization. NetCrafter leverages these observations to a) reduce the network traffic (flits) by stitching compatible flits that are partly filled, b) trimming network traffic by not fetching flits that are unnecessary, and c) effectively managing network traffic by sequencing flits in the network in such a way that latency-sensitive flits reach their destinations faster.
- To evaluate NetCrafter, we model a non-uniform bandwidth multi-GPU system (Figure 2), inspired by AMD's frontier compute node [26], using the MGPUSim [85] simulator. We evaluate NetCrafter across 15 GPU applications, achieving up to 64% speedup and an average of 16% over our baseline non-uniform multi-GPU configuration.

2 Background

In this section, we describe our multi-GPU baseline architecture, followed by providing details on CTA scheduling, data placement, and virtual memory support. The evaluation methodology is discussed later in Section 5.

2.1 Baseline Architecture

Inspired by the Frontier supercomputer node [6, 18, 26], we model our baseline non-uniform bandwidth multi-GPU setup as shown in Figure 2. We assume a unified virtual memory (UVM)-managed multi-GPU system, where the GPUs share the unified virtual memory address space (more details in Section 2.3). Each GPU in our multi-GPU setup contains several Compute Units (CUs), also called Streaming Multi-processors (SMs) in NVIDIA terminology. It also has a highly banked L2 cache shared among all CUs, alongside HBM/GDDR memory stacks. The available threads in a program are grouped as Cooperative Thread Arrays (CTAs), also known as workgroups or thread blocks, each containing up to 1024 threads. Each CTA executes exclusively on a single compute unit (CU). Each CU is composed of multiple SIMD units. These units share an L1 cache and execute a group of threads in a lockstep manner. This group of threads is referred to as wavefront (warp in NVIDIA terminology) and its size is 64 in our setup. Memory accesses are coalesced using a hardware coalescer before reaching the L1 cache, with misses going to lower levels of the memory hierarchy.

We assume that all GPUs in our multi-GPU setup share the L2 Cache and HBM/GDDR to aggregate the cache and memory capacity [5, 17, 42, 61, 62, 71, 91, 98, 99]. Therefore, threads executing on a CU have access to the L2 cache and memory across all GPUs in the system. The GPUs are connected with a higher-bandwidth network within each cluster. Conversely, inter-cluster networks between GPUs operate at a lower bandwidth.[1] An access served by the local memory partition without using the GPU-GPU network is termed a *local access*. In contrast, an access that requires data from another GPU and must traverse the GPU-GPU network is termed a *remote access*. Therefore, in our setup, all *inter-GPU* accesses, including *inter-GPU-cluster* and *intra-GPU-cluster* accesses are referred to as remote accesses. For these remote accesses, the requested data is facilitated by a per-GPU Remote Direct Memory Access (RDMA) engine [9]. The accesses to L2 caches or HBMs on a remote GPU are slower than accesses to the local L2 cache or HBM. We assume that remote L2 data is not cached in the local L2 partition (however brought to L1 cache) [9, 24, 29, 56, 58, 59, 62, 71, 93].

To understand the access flow in our baseline architecture, Figure 2 illustrates the steps involved in servicing an inter-GPU-cluster read request originating from GPU 3 (inside Cluster 1) towards GPU 1 (inside Cluster 0). First, threads within a wavefront generate accesses for addresses residing in the same cache line. The hardware coalesces these accesses into a single read request, which is then issued to the L1 cache (❶). Second, a miss occurred in the L1 cache (❷). The data required is located in GPU 1, necessitating an inter-GPU-cluster remote access to retrieve the data (❸). Upon receiving the read request, GPU 1 services it by sending back the entire cache line data associated with the read request (❹). The process from 4(a)-4(e) details the steps involved in returning the data from the remote GPU (GPU 1) to the requesting GPU (GPU 3): The RDMA

[1]In this paper, we refer higher-bandwidth networks within a cluster as intra-GPU-cluster networks and lower-bandwidth networks across different clusters as inter-GPU-cluster networks.

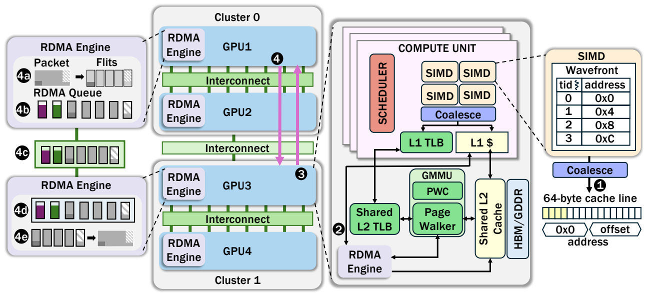

Figure 2: Illustrating baseline architecture and access flow. Our baseline multi-GPU setup pairs high-affinity GPUs with higher-bandwidth network, while groups of GPUs connect via lower-bandwidth network to enable scalable expansion. Access flow is shown for an inter-GPU-cluster remote request generated from GPU 3 towards GPU 1.

engine on GPU 1 receives the cache line data as a read response packet (4a). The read response packet is segmented into flits before being transmitted through the network (4b). The flits are forwarded through the network (4c). GPU 3’s RDMA engine receives the flits from the network (4d). The received flits are reassembled into a read response packet, which is then delivered to the requesting compute unit and cached in its local cache (4e). Overall, this access flow helps in establishing a baseline for future discussions.

2.2 CTA Scheduling and Data Placement

We assume a unified multi-GPU model where the CTAs of a kernel are launched across all the available GPUs [5, 9, 61, 85, 93, 96]. Recent works utilize compile-time static analysis of kernels to infer data access patterns, aiding in CTA scheduling and data placement. An example of such an approach is Locality-Aware Scheduling and Placement (LASP) by Khairy et al. [42], which leverages static index analysis to reduce remote memory accesses. LASP classifies kernels by data access patterns, schedules CTAs to GPUs, and places corresponding data pages locally to reduce remote accesses. Our baseline design adopts LASP’s CTA scheduling and data placement strategies, which are also utilized in other recent works [29, 71].

2.3 Virtual Memory Support

To support virtual memory, each CU in our design is equipped with a private L1 Translation Lookaside Buffer (TLB), while a shared L2 TLB and GPU Memory Management Unit (GMMU) serve all CUs within a GPU. The GMMU includes a Page Walk Cache (PWC), which accelerates page table walks (PTWs) by caching entries from the upper levels (levels 1–3) of the radix-tree page table. It also features a set of parallel page table walkers that handle virtual-to-physical address translation by traversing the page tables stored in HBM/GDDR memory. If the required page table entry (PTE) resides on GPUs in other clusters, these accesses may involve communication across the inter-GPU-cluster network.

On a load or store request, the CU first queries its private L1 TLB for a virtual-to-physical address translation. If the translation is not found, the request is forwarded to the shared L2 TLB. Upon a L2 TLB miss, the PWC is accessed to perform a longest-prefix match on the Virtual Page Number (VPN), potentially resolving a portion of the

PTW. Depending on the prefix match length, a PTW through the four-level page table may require 1 to 4 memory accesses, handled by dedicated page table walkers. Once the translation is obtained, it is inserted into both the L1 and L2 TLBs to avoid future misses, and then returned to the CU, which issues the load or store to the translated physical address. In our design, PTEs are also cached in the L2 cache along with data.

While LASP co-locates CTAs with their data to reduce remote accesses, it does not address PTE placement. We extend LASP by aligning PTE pages with data placement: each PTE page, mapping a 2MB virtual address region, is placed on the GPU where the first data page in that region resides. This ensures that data and its translation metadata are co-located, minimizing remote translation overhead. Our approach reflects Linux's NUMA-aware PTE placement [1] and aligns with prior GPU virtual memory work [71].

3 Motivation and Analysis

In this section, we first quantify the challenges posed by non-uniform bandwidth in multi-GPU systems. Subsequently, we analyze network traffic across lower-bandwidth networks to reduce and manage that traffic.

3.1 Analyzing Network Bottleneck

To evaluate the impact of non-uniform bandwidth on multi-GPU performance, we compare our baseline setup with an ideal configuration, where all GPUs are connected via high-bandwidth networks (see Section 5 for details on the bandwidth numbers). Note that the *ideal* configuration is not practical; it indicates the optimization potential. As shown in Figure 3, the ideal network configuration achieves an average 1.5× speedup over our baseline.

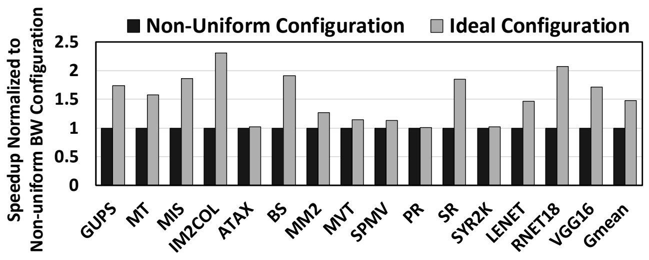

Figure 3: Performance of a non-uniform multi-GPU system compared to an ideal setup with only high-bandwidth networks.

The substantial performance difference is primarily attributed to network bottleneck, as illustrated by high network utilization (which suggests significant congestion) across lower-bandwidth networks (see Figure 4). This heightened network utilization also contributes to higher average inter-GPU memory access latency, as evidenced in Figure 5, which consequently degrades overall system performance. These findings underscore how non-uniformity in the bandwidth exacerbates communication bottlenecks within multi-GPU systems, thereby significantly impacting application performance. Therefore, effective strategies to reduce and manage network traffic on lower-bandwidth networks connecting GPUs across clusters are critical. In the subsequent section, we delve into the insights that guided the development of our optimization strategies aimed at mitigating and managing network traffic on these lower-bandwidth networks.

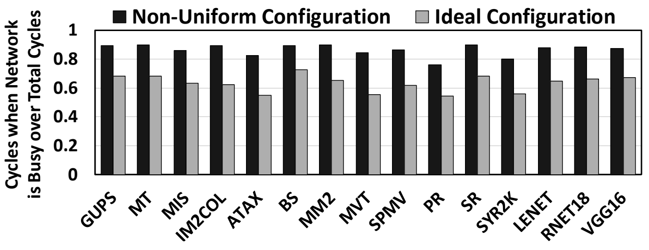

Figure 4: Network utilization of non-uniform & ideal config.

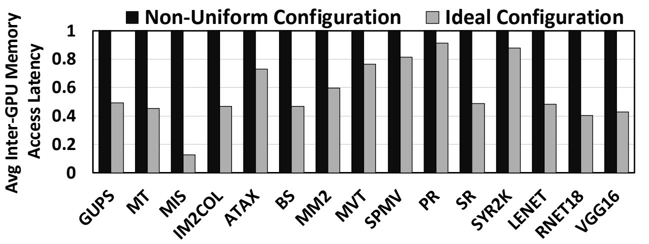

Figure 5: Average inter-GPU-cluster memory access latency of non-uniform and ideal configuration (normalized to the non-uniform configuration).

3.2 Reducing Network Traffic

Our first strategy reduces pressure on lower-bandwidth networks by minimizing network traffic, primarily through eliminating redundant or unnecessary data. This is driven by two key observations:
Observation 1: Substantial empty bytes in network flits. We classify network traffic (illustrated in Table 1) generated by GPU accesses into six distinct types, each varying in packet size and content: 1) read request packet (containing an 8-byte address), 2) read response packet (containing 64 bytes of cache line data), 3) write request packet (containing both 64 bytes of cache line data and an 8-byte address), 4) write response packets (containing acknowledgment information, typically embedded in the packet header), 5) page table request packet (containing an 8-byte address), and 6) page table response packet (containing 8-byte physical address, significantly smaller than read response data for an entire cache line). Additionally, each packet includes a fixed 4-byte metadata header. The *Bytes Required* column in Table 1 reflects the total number of bytes needed by each packet type.

Table 1: Categorizing 16B flits by type and size as observed in MGPUSim [85]. Each flit has multiple useless (padded) bytes.

Request Type	Bytes Occupied	Bytes Required	Bytes Padded	Flits Occupied
Read Req	16	12	4	1
Write Req	80	76	4	5
Page Table Req	16	12	4	1
Read Rsp	80	68	12	5
Write Rsp	16	4	12	1
Page Table Rsp	16	12	4	1

These distinctions highlight the diverse information carried by each packet type. When segmented into flits, these packets are

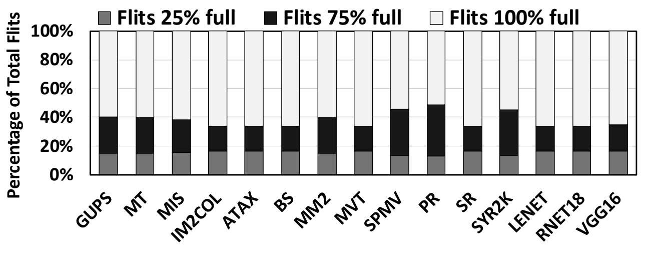

Figure 6: Distribution of flits categorized by occupancy levels.

often padded if the data packet size is not a multiple of the native flit size, leading to unnecessary traffic load across the network and additional wasted bandwidth. The issue of wasted bandwidth due to fragmentation—caused by splitting packets into flits—has also been acknowledged in a recent prior work [62]. In our effort to optimize the lower-bandwidth networks between the GPUs, we identify these padded bytes as a source of redundant data. To quantify our findings, Figure 6 illustrates the percentage of total flits with either 25% or 75% padded bytes in the lower-bandwidth network. Our analysis reveals that, on average, 42% of flits in the lower-bandwidth network contain either 25% or 75% redundant (padded) data, revealing opportunities for traffic reduction in these networks.

Observation 2: Partial cache-line utilization by wavefronts. We analyzed cache line utilization by all threads of a wavefront after coalescing. Figure 7 shows the fraction of inter-GPU-cluster read requests based on the cache line bytes required by the wavefront. The analysis reveals that many evaluated applications needed only 16 bytes or fewer from the 64-byte cache line fetched from the GPU. This results in inefficient use of network bandwidth, adding unnecessary traffic to already congested networks. This observation underscores another opportunity to optimize network traffic.

3.3 Managing Network Traffic

Our second strategy to optimize lower-bandwidth networks between GPUs focuses on the efficient management of network flits. This approach aims to minimize total application execution time by handling flits based on their latency requirements. This approach is guided by the following two key observations:

Observation 3: PTW-related flits lie on the critical path of data accesses. To efficiently manage network flits on the lower-bandwidth networks, it is crucial to identify and address the latency requirements of each flit category. We investigated the types of flits to classify them based on their impact on the overall performance. By prioritizing a constant number of each flit category and measuring the resulting performance, our experiments revealed that read PTW-related flits are more latency-critical and have a greater impact on application performance. This is because read PTWs require translating virtual addresses to physical addresses, which can stall critical data read accesses while the translation completes.

Figure 8 quantifies our findings, comparing the performance of prioritizing read PTW-related accesses in the network versus prioritizing an equal number of data accesses. The results show that prioritizing data accesses over read PTW-related accesses worsened performance, while prioritizing read PTW-related accesses improved performance compared to the baseline. This shows that

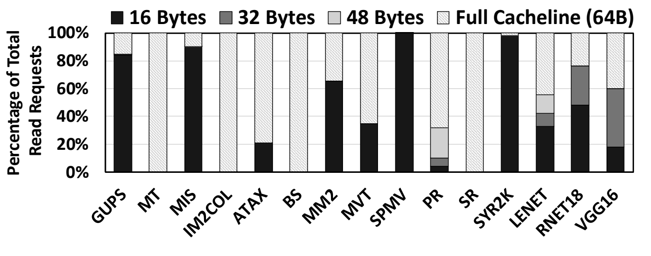

Figure 7: Categorization of inter-GPU-cluster read requests based on the number of bytes required from the cache line.

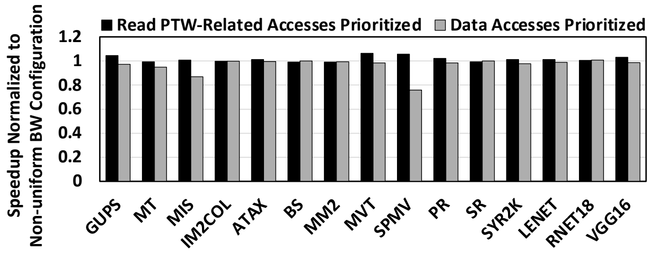

Figure 8: Performance characterization after prioritizing read PTW-related accesses and the same fraction of data accesses.

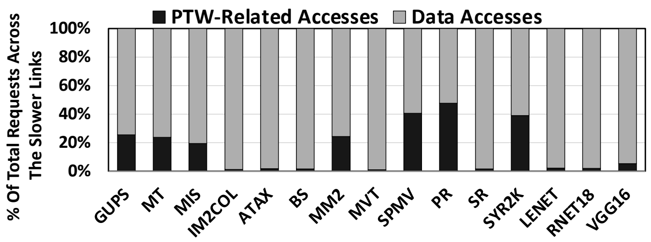

Figure 9: Ratio of data and PTW-related accesses in the lower-bandwidth network.

read PTW-related accesses are more latency-sensitive than data accesses, highlighting the benefit of prioritizing read PTW-related flits to improve performance over lower-bandwidth networks.

Observation 4: PTW-related flits make up a small fraction of flits on lower-bandwidth networks. After identifying read PTW-related flits as latency-critical, it is crucial to assess their frequency and data volume relative to the total network flits before proposing an optimization strategy. Page table requests contain only the address, making them very small. Page table responses contain only the translated physical address and are significantly smaller than read response requests, which include the entire cache line. Thus, we found that Page table requests and responses occupy a small fraction of the total bytes in the network. Figure 9 shows the fraction of total traffic in the lower-bandwidth network that comprises PTW-related accesses (page table requests and responses) versus data accesses. The distribution indicates that PTW-related accesses occupy a small fraction of the total network accesses, averaging around 13% of the total accesses. From our analysis, we conclude that PTW-related flits are not only latency-critical but also occupy a small fraction of the total flits by weight and volume, making them well-suited for prioritization in the network (i.e., reinforcing the previous observation 3).

4 NetCrafter: Design and Implementation

Based on the observations discussed in the previous section, we propose NetCrafter – a unique combination of three key mechanisms (Stitching, Trimming, and Sequencing) that are applied together to optimize network traffic. In this section, we present their design and implementation details. It is important to note that our proposed methods are specifically applied to inter-GPU-cluster networks, with the design details and protocol modifications tailored for a simplified PCIe-like protocol. However, our proposed modifications can be adapted based on the protocol in use.

4.1 Packet and Flit Structures

Figure 10(a) shows the baseline packet structure. We assume a simplified PCIe-style packet consists of a header and a payload. The header of each packet has essential information, alongside unused bits which we repurpose to realize our mechanisms. The payload has the data. We assume the header is 12 bytes[2], out of which 4 bytes are metadata and the remaining 8 bytes are for the address. The 4-byte header metadata contains a 5-bit type field indicating the packet type, destination routing information, and a unique identification tag (ID) for each packet. Out of 64 address bits (8 bytes), 48 bits are used for memory address in a PCIe-style packet, and the remaining 16 bits are unused.

A packet is segmented into fixed-size flits before being transmitted across the network. Figure 11(a) shows various packet types and how they are divided into flits. For example, the Read-Response packet is divided into five flits: the first flit contains the header and partial payload data, followed by flits consisting of the remaining payload data. The fifth (final) flit also has empty bytes. Further, different types of network flits contain varying amounts of empty bytes, which are padded with redundant data before transmission, as discussed in Section 3.2. In our simulated system, the Read-Response packet is 80 bytes (5 flits) out of which only 68 bytes in total are occupied by the header (dark colors in Figure 11(a)) and payload (light colors in Figure 11(a)). The remaining 12 bytes are empty (white-gray patterns in Figure 11(a)), which are filled by redundant data to pad the empty bytes in the flit.

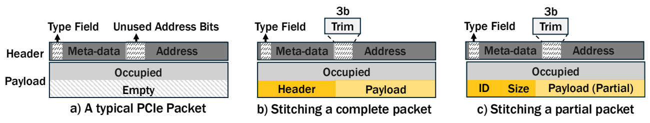

Figure 10: a) A simplified PCIe packet structure, b) NetCrafter packet stitched with another (whole) packet, c) NetCrafter packet stitched with another (partial) packet. Stitched packets are shown in yellow.

4.2 Design of Stitching Mechanism

To address the identified flit-level inefficiency, we propose *Stitching*, which reduces overall network traffic by sending useful data from other outstanding packets instead of redundant data in the network.

[2]Per [12, 62, 70], the packet consists of a header (up to 16 bytes), 12 bytes of additional information (e.g., CRCs, framing, and sequence number), and payload. To be compatible with the MGPUSim simulation framework, we are not considering additional information and using the header size of 4 or 12 bytes based on packet type: 4 bytes for Read Rsp and Write Rsp and 12 for others types (Table 1).

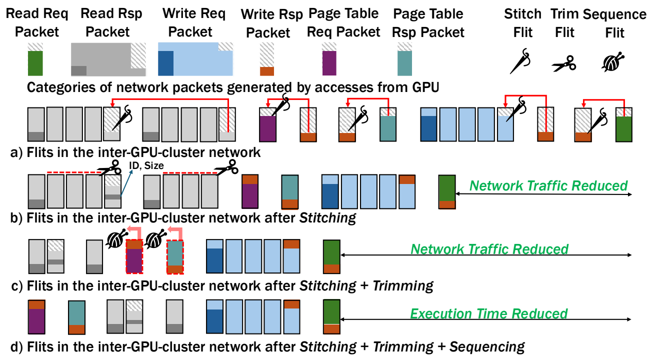
Read Req Packet
Read Rsp Packet
Write Req Packet
Write Rsp Packet
Page Table Req Packet
Page Table Rsp Packet
Stitch Flit
Trim Flit
Sequence Flit
Categories of network packets generated by accesses from GPU
a) Flits in the inter-GPU-cluster network
ID, Size
Network Traffic Reduced
b) Flits in the inter-GPU-cluster network after Stitching
Network Traffic Reduced
c) Flits in the inter-GPU-cluster network after Stitching + Trimming
Execution Time Reduced
d) Flits in the inter-GPU-cluster network after Stitching + Trimming + Sequencing

Figure 11: Reducing & managing network traffic with NetCrafter. Dark colors represent headers; light colors represent payloads & white-gray patterns represent empty bytes.

When a flit (called *parent* flit) is retrieved from the queue to be sent into the network, it is assessed for the number of empty bytes and its destination. We then search for stitching candidates in the queue that meet specific criteria: (1) the candidate flit should contain fewer than or equal to the number of empty bytes in the *parent* flit, ensuring a fitting merge, and (2) the candidate flit should share the same destination.

Once an ideal candidate flit is identified, it is stitched into the parent flit using one of two techniques. (1) If the candidate flit belongs to a larger packet and contains only payload data, we first prepend it with metadata—specifically, an identification tag (ID) and size field (Size)—both inferred from the header of the original packet. This augmented flit is then stitched into the parent flit. This enables the receiver side to correctly reassemble the candidate flit with its remaining packet using the ID, and to accurately un-stitch it using the Size field. (2) If the candidate flit contains a complete packet (including both header and payload), it is stitched directly without additional bytes. The offset of the stitched packet can be determined by inferring the size of the *parent* packet from its header. Multiple candidates can be stitched into the parent flit, as long as their total size fits within the parent flit's remaining space.

Figure 11(b) shows the benefits of Stitching when applied to the stream of flits shown in Figure 11(a). Five different kinds of stitching scenarios are shown. In the first scenario, the last flits in two back-to-back read response packets have empty bytes (e.g., contain the trailing end of data from a read response packet) and hence are the candidates for stitching. However, given that header information is not available in the last flit of the second read response packet, we will need additional information to stitch it so that it can be identified during un-stitching. Figure 11(b) shows the status of flits after stitching is performed, along with some additional bytes that contain an ID and a Size field. In the remaining four scenarios, stitching between compatible flits is straightforward, as the entire packet—header and payload— is available, requiring no additional metadata to be added. Overall, as noticed from Figure 11(b), significant network traffic can be reduced with Stitching.

Packet Structure that Supports Stitching. To implement our mechanisms, the communication protocol needs minimal updates to the packet structure (see Figure 10(a)). We leverage unused encoding (out of several unused combinations available [62]) of the

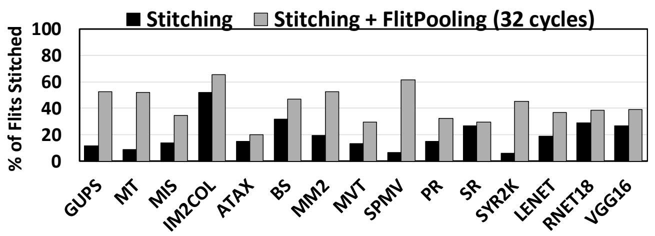

Figure 12: Percentage of total flits stitched before and after application of Flit Pooling with Stitching. Flit Pooling significantly improves the percentage of total stitched flits.

type field in the PCIe-style packet header to denote a stitched flit. Figure 10(b) and Figure 10(c) show two types of stitched packets that incorporate these changes. The payload of a parent flit is modified by including stitching candidate flits as multiple sub-packets concatenated within a single transaction. In the first type, a complete packet is stitched (highlighted in yellow) with the parent flit. In the second type, only part of the packet is stitched (highlighted in yellow), which does not contain its own header. Therefore, we include additional information (ID and Size of partial payload). Note that NetCrafter stitches only those flits that follow the same route, allowing them to share header fields and simplifying the design.

Optimization I: Flit Pooling. While flit Stitching is promising in reducing overall traffic, our experiment indicates that only a small fraction of packets can be stitched (see Figure 12). This limitation arises from the limited availability of suitable stitching candidates in the queue under study at the time of flit ejection. To enhance the opportunity of stitching, we introduce *Flit Pooling*. Leveraging the latency tolerance of GPU architectures, Flit Pooling temporarily postpones the ejection of a network flit if no suitable stitching candidate is found. This delay allows more time for a suitable stitching candidate to arrive in the queue. After the delay period, the deferred flit is re-evaluated for potential stitching. If a suitable candidate is found, stitching is performed; otherwise, the flit is ejected without stitching. We performed a design space exploration by varying the delay period and measuring its impact on overall performance (Figure 18). The delay period in the final design was optimized to achieve the best trade-off between latency increase and improved stitching percentage.

Optimization II: Selective Flit Pooling. While GPUs are generally not sensitive to latency, they are sensitive to the latency of certain traffic categories (see Observation 3). To ensure broad applicability and minimal disruption to critical applications, we employ *Selective Flit Pooling*. This approach exempts latency-sensitive flits from the Flit Pooling delay. During *Selective Flit Pooling*, the flit undergoes assessment to determine if it is latency-critical. If the flit is considered latency critical, we exclude it from Flit Pooling, otherwise, the ejection is delayed temporarily while subsequent flits in the queue are processed. Currently, we consider all the PTW-related flits (see Figure 8) latency critical, as guided by Observation 3. Selective Flit Pooling mitigates the performance degradation caused by increased latency associated with Flit Pooling. This improvement is demonstrated later in Section 5.

4.3 Design of Trimming and Sequencing Mechanisms

In Section 3.2, we described that threads within a wavefront often do not require all bytes in a 64-byte cache line fetched from a remote GPU. To address the inefficiencies of transmitting unnecessary bytes, especially over lower-bandwidth networks, we propose *Trimming*. *Trimming* involves modifying network packets to eliminate superfluous data. When the remote GPU hosting the cache line is ready to satisfy the packet request, we analyze it to determine the exact number of bytes requested by the wavefront. If this number is 16 bytes or fewer, and the response packet has to traverse an inter-GPU-cluster network, we trim the packet to include only the necessary bytes, discarding the excess. If the number is above 16 bytes, or the packet won't traverse an inter-GPU-cluster network, we do not discard any flits to preserve spatial locality advantages. Trimming packets minimizes the transmission of redundant data, which previously polluted the network channel. Figure 11(c) shows the benefits of Trimming, which discards the flits that are not required by the application. For example, we show two such scenarios where a number of flits (payload related) in two Read Response packets are not required and hence not sent via the network, thereby reducing network traffic.

Packet Structure that Supports Trimming. The communication protocol requires minimal updates to the packet structure (see Figure 10(a)) to support trimming. NetCrafter's updated packet format (see Figure 10(b) and (c)) re-purposes three unused bits (Trim) in the *address* field [46, 50, 52]. One bit specifies whether the request needs 16 bytes or a full cache line of data, while the other two bits indicate the offset within the 64B cache line.

Implications of Trimming on Cache Design. To complement our trimming optimization, we take advantage of the concept of sectored cache design [35, 74], which has been explored in the context of a single-GPU scenario. This cache breaks each cache line into smaller, independently addressable sectors or sub-blocks of 16 bytes — the same granularity at which Trimming brings data into the cache. On an L1 cache miss that requires an inter-GPU-cluster request to service the miss, only the necessary sectors are loaded from the main memory of the relevant inter-cluster-GPU into the L1 cache, reducing data transfer and enhancing the efficiency of our proposed Trimming technique.

Impact of Trimming on the Cache Performance. Trimming does not entirely negate the spatial locality benefits of natural fetching of the cacheline. In NetCrafter, a full cache line is always retrieved from the L2 cache for all accesses. However, the NetCrafter controller employs Trimming only to inter-cluster GPU requests. For intra-cluster GPU requests, where bandwidth is higher, the standard fetching mechanism remains unchanged, allowing full cache line retrieval without introducing additional overhead.

Design of Sequencing Mechanism. Based on our analysis in section 3.3, we found that PTW-related flits are not only latency-sensitive but also constitute a small fraction of total network flits in terms of both their weight and volume. Given these insights, we propose a strategy termed *Sequencing*. This approach prioritizes the handling of PTW-related accesses that potentially delay read requests, ensuring they are addressed promptly. Since the PTW-related accesses occupy a small proportion of accesses on lower-bandwidth networks, prioritizing them there does not adversely affect the queuing latency for data requests. Instead, it optimizes traffic flow, thereby enhancing overall performance without significant trade-offs. Figure 11(d) provides a high-level overview of sequencing, illustrating how PTW-related flits are prioritized over other network flits to minimize the end-to-end execution time of applications (implementation details are up next).

4.4 Implementation of NetCrafter

The overall architecture of NetCrafter is encapsulated within the NetCrafter Controller, as illustrated in Figure 13. The NetCrafter controller performs Stitching, Trimming, and Sequencing and is located inside each cluster's switch. It comprises of three key components: 1) a Trim Engine for Trimming, 2) a Cluster Queue (CQ), and 3) a Stitching Engine for Stitching.

Trim Engine. It is responsible for refining network packets by Trimming read response packets to match the specific bytes requested by the GPU. As previously discussed, we modify the packet structure to include *trim* bits. These bits indicate if Trimming is required and, if so, specify the offset within the 64-byte cache line. The Trim Engine uses the *pkt.trim* bits (shown in Figure 13) as control signals to trim read response packets and outputs them to the Stitching Engine for further processing.

Cluster Queue (CQ). It is an SRAM structure located at the inter-GPU-cluster network egress port. It selectively buffers flits destined to traverse lower-bandwidth networks, preparing them for stitching. The Cluster Queue uses a two-level virtual structure: the first level, *CQ.dst*, groups flits by destination cluster, while the second level, *CQ.type*, subdivides each *CQ.dst* by request type (as defined in Table 1). The request type determines the number of empty bytes available for stitching. The appropriate cluster queue partition for a flit is selected based on control signals (*pkt.dst* and *pkt.type*) from the packet, specifying the destination cluster and packet type. A round-robin scheduler allocates service turns across these partitions. Each Cluster Queue entry occupies 16 bytes, the fixed size of each flit in our network.

Stitching Engine. The stitching engine operates on two flits: a parent flit and a stitching candidate flit. It first examines the candidate to determine whether it is a complete packet—containing both header and payload—or a partial packet consisting only of payload. If the candidate is a complete packet, it is directly stitched into the parent flit. Otherwise, if the candidate includes only a payload segment, the engine appends lightweight metadata: a unique ID identifying the original packet and a Size field indicating the number of bytes included. Finally, to signal that the resulting packet contains stitched content, the NetCrafter engine repurposes an otherwise unused combination in the 5-bit type field of the parent packet header [62]. The final output is a stitched flit, ready for transmission across the network. The stitching engine at the receiver end also performs un-stitching once a stitched flit reaches its destination. It begins by examining the 5-bit type field in the flit header to determine whether the flit contains stitched data. If the flit is identified as stitched—using the repurposed unused type value—it is flagged for further processing. The engine then parses the flit to

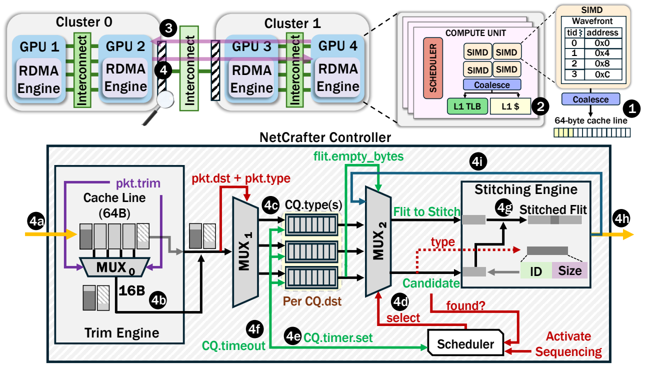

Figure 13: NetCrafter in Action.

extract metadata associated with each stitched flit, including the ID and Size fields. This information enables the engine to locate and separate all constituent flits embedded within the stitched packet. Finally, the engine reunites each extracted flit with the remaining portion of its original packet by matching the ID appended in the parent flit with the ID of the remaining packet of the stitching candidate from their header.

Putting it all Together. Figure 13 provides a step-by-step example of the NetCrafter workflow. A wavefront from Cluster 1 GPU 4 generates a read request for an address in Cluster 0 GPU 2, requiring data from the first 16 bytes of the cache line (1). The L1 cache misses this address, prompting a remote request to GPU 2 (2). The request is then forwarded to GPU 2 through the network (3). GPU 2 services the read request, generates a read response with a 64-byte cache line, and sends the data back to GPU 4 (4).

To illustrate our design, we detail the steps for sending the read response packet from GPU 2 to GPU 4. The packet is partitioned into flits, which are processed by the NetCrafter controller before being ejected into the network (4a). In this case, the read response required only 16 bytes from the total 64 bytes fetched, so the Trimming engine examines the packet, checks its destination, and trims the remaining bytes since it is traversing the lower-bandwidth network (4b). Each flit is placed into the appropriate cluster queue partition based on its destination and type. The type determines the empty bytes within the flit, except for PTW-related flits, which are placed in a separate queue (4c). The flit is retrieved based on the cluster queue's turn, scheduled in a round-robin fashion (with a bias towards prioritizing the cluster queue containing PTW-related flits). The retrieved flit passes through a multiplexer, which also forwards the stitching candidate based on the first flit's empty bytes (4d). If no candidate is found for stitching, a signal is sent to the scheduler, setting a timer for the queue that ejected the flit to be stitched. This timer delays retrieval from that cluster queue until it expires. The timer is never set for the PTW-related cluster queue, as PTW-related flits are latency-critical and exempt from the timer. If no candidate is found for PTW-related flits, they are ejected regardless (4e). After the timeout, the scheduler resumes the cluster queue's turn (4f). When a stitching candidate is found, the stitching engine evaluates it. If the candidate contains both a header and payload, the engine stitches both flits. If the candidate only contains the payload, the stitching engine generates an ID and a Size field and appends it with the stitched flit (4g). The flit is pushed out of the network after our techniques are applied (4h). Note that after stitching, the flit can still be stitched again if enough empty bytes remain (4i).

4.5 Other Design Considerations

Hardware Overhead of NetCrafter. Each GPU cluster is equipped with a NetCrafter controller, which is integrated into the cluster's switch. This controller requires 16KB of SRAM for the cluster queue. Additionally, the stitch engine requires 16B SRAM buffer to hold the flits. Overall, the overhead for the NetCrafter controller sums up to 16.02KB per cluster. This accounts for about 0.098% of the L2 cache size (16MB) of the AMD Instinct MI250X GPU used in the Frontier node [6, 26]. With a programmable switch like the Intel Tofino ASIC [3] (64MB SRAM), the overhead drops to 0.024%. Note that this overhead only accounts for the extra SRAM from NetCrafter, excluding timer and multiplexer unit overhead, as their overhead may not be significant.

Latency Overhead of NetCrafter. NetCrafter Controller is designed as a pipelined system, enabling efficient parallel processing of operations as shown in Figure 13. The trim and stitch operations are conservatively assumed to overlap with the switch's processing pipeline (more details on the switch processing latency in Section 5.1). However, selective Flit Pooling introduces latency overhead that varies with the pooling window size, potentially affecting overall performance, as shown in Figure 18 and Figure 19.

Coherency Implications of NetCrafter. Hardware managed coherence maintains cache line consistency by tracking sharers and invalidating remote copies during writes. In contrast, software-managed coherence relies on flush operations, where dirty cache lines are written back upon triggers from software-inserted synchronization primitives or at kernel boundaries. Currently, software coherence is the predominant approach in commercial products [61, 62, 64, 98, 101], while hardware coherence has been explored primarily in research on multi-GPU systems [73]. Our baseline assumes this software-managed cache coherence and also does not cache remote data in local L2 [9, 24, 29, 56, 58, 59, 62, 71, 93]. NetCrafter can also seamlessly complement any underlying hardware coherence mechanisms. The fine-grained nature of hardware coherence traffic presents additional opportunities for stitching, a direction we leave for future work. At the same time, Trimming can function independently by incorporating a write mask into invalidation requests across GPUs.

Handling Deadlocks in NetCrafter. In our simulations, we did not observe any scenarios where an application is not making forward progress. To prevent protocol-level deadlocks, the stitching engine only stitches flits that are traveling in the same direction, thereby avoiding potential cyclic dependencies. Additionally, we carefully size buffers and queues to ensure sufficient capacity under varying traffic conditions. While NetCrafter allows stitching across different packet types—such as combining request and response packets—they are always destined for the same endpoint.

5 Experimental Results

In this section, we first describe our experimental setup, followed by the evaluation of our proposed mechanisms.

5.1 Methodology

We model our cache organization, non-uniform GPU networks, and other system details (all discussed in Section 2) using the cycle-level simulator MGPUSim [85]. The detailed baseline system specifications are presented in Table 2. Our baseline architecture assumes a non-uniform bandwidth system consisting of four GPUs. Each pair of GPUs within a cluster is interconnected by a higher-bandwidth network of 128 GB/s (intra-GPU-cluster network). In contrast, accesses between GPUs across clusters utilize a lower-bandwidth network of 16 GB/s (inter-GPU-cluster network), creating a bandwidth ratio of 8:1. However, to thoroughly evaluate the impact of varying bandwidth ratios and values on overall performance with NetCrafter, we conduct an extensive sensitivity analysis. This study, detailed in Section 5.5, examines different bandwidth ratios and higher bandwidth numbers to provide a comprehensive analysis.

Table 2: Baseline Multi-GPU Configuration.

Parameter	Value
Compute Unit (CU)	1 GHz, 64 per GPU
L1 Cache (per CU)	64KB write-through L1 vector cache, 20 cycle lookup, 32-entry MSHR
	32KB inst. and 16KB scalar cache shared b/w 4 CUs
L1 TLB (per CU)	32 entry fully assoc., 1 cycle lookup [29, 56, 57, 71], 8-entry MSHR
L2 TLB	512 entry TLB/GPU, 8 way, 10 cycle lookup [56–59], 64-entry MSHR [71]
L2 Cache	4MB per GPU, 16 banks, 16 way [71], 64-entry MSHR
	100 cycle lookup [43, 69], 64B cache line, write-back
DRAM	1 TBps, 100 ns latency [29, 71]
Page Table Walk	GMMU 16 shared page table walkers per GPU (default) [29, 71]
Page Walk Cache	32 entry fully assoc. page cache, 10 cycle lookup [71]
CTA/Page Scheduling	LASP [29, 42, 71]
Heterogeneous Interconnect	Inter-GPU-cluster - 16GBps, bi-directional (controlled by switch)
	Intra-GPU-cluster - 128GBps, bi-directional (controlled by switch)
Network Switch Parameters	30 cycle processing latency, 1024 entry I/O buffer size
NetCrafter Parameters	Cluster Queue - 1024 Entries (16B each) equally partitioned/dst cluster

Table 3: List of Evaluated Applications

Abbr.	Application	Access Pattern	Benchmark Suite
GUPS	multi-threaded, random access	Random	MGPUSim [85]
MT	matrix transpose	Gather	AMDAPPSDK [4]
MIS	max. independent set	Random	Pannotia [13]
IM2COL	image to column	Adjacent	DNN-Mark [23]
ATAX	matrix transpose & vector multiplication	Scatter	Polybench [32]
BS	blackscholes	Partitioned	AMDAPPSDK [4]
MM2	2D matrix multiplications	Gather	Polybench [32]
MVT	matrix vector product and transpose	Scatter,Gather	Polybench [32]
SPMV	sparse matrix vector multiplication	Random	SHOC [16]
PR	page rank algorithm	Random	Hetero-Mark [87]
SR	shoc-reduction	Gather	SHOC [16]
SYR2K	rank-2k of a symmetric matrix	Adjacent	Polybench [32]
VGG16	deep CNN for large-scale image recognition	-	DNN-Mark [23]
LENET	CNN for digit recognition	-	DNN-Mark [23]
RNET18	RESNET18 - deep CNN with residual connections	-	DNN-Mark [23]

MGPUSim integrates cycle-level network simulation from the Akita framework to model GPU networks. The switches route flits across multiple ports, with each flit passing through a data processing pipeline with a 30-cycle latency, while maintaining a throughput of 1 flit/cycle/port. After pipeline processing, flits wait in a buffer (1024 entries) for routing. The switch, implemented similarly to a crossbar, sends and receives one flit per cycle per port. If the outgoing buffer is full, routing is paused, causing back pressure that can propagate to upstream switches. Flits are transmitted from the outgoing buffer at 1 flit/cycle/port. We adopt a 16-byte flit size, the smallest used among commercial network technologies (e.g., PCIe, NVLink, Infinity Fabric), and previous works that use specific flit sizes. [10, 35, 53, 54, 62].

For our evaluation, we select a diverse set of workloads representing various multi-GPU memory access patterns [9, 86]: *random* *access* (GUPS, SPMV, PR, MIS), *adjacent access* (SYR2K, IM2COL), *partitioned access* (BS), *gather access* (MT, MM2, SC), and *scatter access* (ATAX, MVT). To ensure our evaluation covers a wide range of real-world access patterns across diverse multi-GPU frameworks, we also evaluate NetCrafter on deep neural network (DNN) training workloads, specifically VGG16, LENET, and RESNET18, using data parallelism. For the RESNET18 and VGG16 evaluations, we use the Tiny-ImageNet-200 [48] dataset, while for LENET, we utilize the MNIST [49] dataset. We did not use large-scale datasets due to prohibitively long simulation times. Table 3 summarizes the workloads and their access patterns.

To ensure an unbiased baseline unaffected by suboptimal workload mapping, we conducted an analysis of memory access distribution within each GPU, categorizing accesses as local (from the same GPU) or remote (from other GPUs). Our analysis revealed that LASP [42] in our baseline effectively maximizes local accesses and balances remote accesses across GPUs. Despite efficient mapping, remote accesses are unavoidable as systems scale, highlighting the persistent challenge of managing remote memory accesses in large-scale GPU systems.

5.2 Overall Performance Analysis

Figure 14 shows the performance improvements achieved through our three optimizations: Stitching (which includes Stitching with selective Flit Pooling (32 cycles)), Trimming, and Sequencing. Overall, NetCrafter achieves a speedup of up to 64%, with an average improvement of 16%, compared to our baseline non-uniform multi-GPU configuration. We observe significant improvement in workloads constrained by the network bandwidth. While Stitching and Sequencing consistently benefit all bandwidth-constrained workloads, Trimming offers targeted improvements for workloads that partially utilize the fetched cache lines. For instance, workloads such as MIS, SPMV, and GUPS not only experience network bandwidth constraints but also display a pattern of mostly using 16 bytes from an entire cache line fetched, as illustrated in Figure 7. This makes them particularly well-suited to benefit from Trimming. Moreover, reduction in network traffic helps in reducing the overall inter-GPU-cluster memory access latency as shown in Figure 15.

5.3 Trimming and L1 Sectored Cache Baseline

Figure 14 also includes a bar illustrating the performance improvement of an L1 sector cache design over our baseline, enabling a direct comparison with our Trimming approach. To ensure a fair comparison with Trimming, which operates at 16-byte granularity, we modify our L1 vector cache to function as a sector cache with 16B sectors. We developed our sector cache baseline model by building upon the single-GPU sectoring implementation in Accel-Sim [43], extending it to support sectoring in our multi-GPU system baseline in MGPUSim. The sector cache baseline (16B) adopts an *all trimming* approach that brings data to L1 cache at the granularity of 16 bytes, regardless of whether the associated requests traverse lower (inter-GPU-cluster) or higher (intra-GPU-cluster or within the GPU) bandwidth networks. In contrast, our Trimming approach selectively reduces traffic only on lower-bandwidth inter-GPU-cluster networks, achieving a better balance between

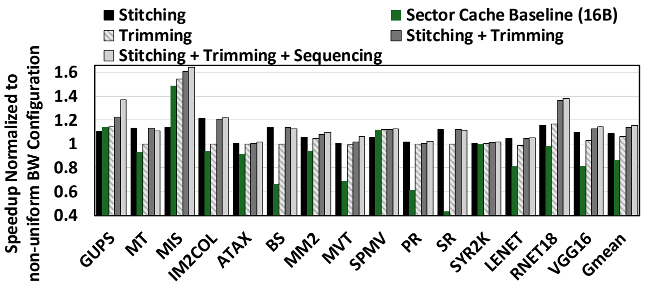

Figure 14: Overall performance improvement with NetCrafter normalized to the baseline non-uniform bandwidth multi-GPU configuration.

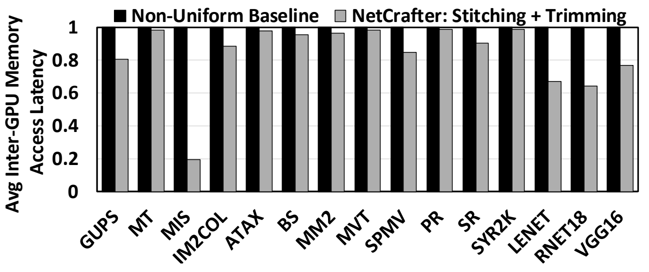

Figure 15: Average Inter-GPU-Cluster memory access latency for non-uniform configuration vs. NetCrafter

the potential increase in cache miss rate (due to sectoring) and performance gains from reduced network traffic.

From Figure 14, we observe that the sector cache baseline (16B) improves performance for workloads such as MIS, GUPS, and SPMV, as they benefit from the fine-grained cache responses. However, for workloads benefiting from coarser-grain cache responses, sector cache leads to an increase in cache misses (Figure 16), resulting in decreased performance compared to our Trimming approach. Although PR, a graph workload with an inherently sparse access pattern, experiences a performance drop with 16-byte sectors, performance could improve with larger sector sizes. As shown in Figure 7, request distributions often span 16, 32, or 48 bytes within a cache line, suggesting that larger sectors may better suit such workloads. We further investigate this issue with Large GEMM Kernels. For these kernels, Figure 17 shows the impact of Trimming on Cache misses-per-kilo-instructions (MPKI) as a function of trimming granularity / sector size. Although the Trimming approach reduces L1 cache MPKI compared to an all-trimming strategy, we leave the exploration of finding an optimal granularity/sector size as a part of future work.

5.4 Sensitivity Study with Flit Pooling

Stitching With Flit Pooling. Figure 18 shows the impact of varying Flit Pooling times from 32 to 128 cycles. While increasing the pooling time improves the likelihood of successful flit stitching (Figure 12), it also adds latency for flits unable to find suitable stitching candidates in real-time. Achieving an optimal Flit Pooling time requires balancing reduced network traffic from improved stitching against increased inter-GPU-cluster latency for pooled flits. Figure 18 indicates 32 cycles as a sweet spot for optimal performance

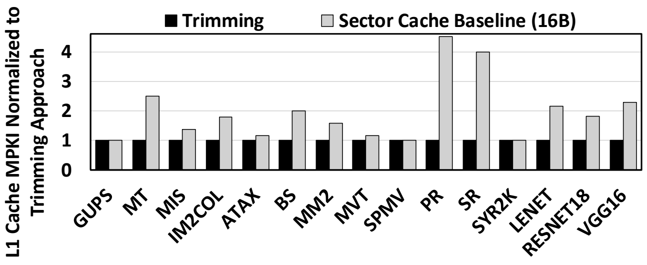

Figure 16: L1 Cache MPKI comparison for trimming in NetCrafter versus a sector cache design (16 byte sector size.

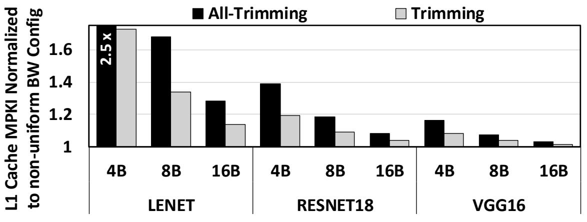

Figure 17: L1 Cache MPKI comparison for trimming in NetCrafter versus an all trimming approach, evaluated at trimming granularity of 4, 8, and 16 bytes.

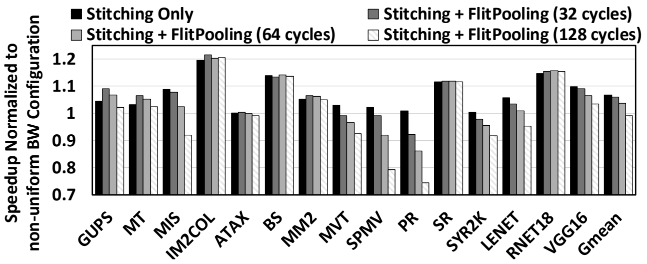

Figure 18: Performance improvements with Stitching alone and Stitching combined with Flit Pooling, evaluated across Flit Pooling times from 32 to 128 cycles, normalized to a baseline non-uniform bandwidth multi-GPU configuration.

with Stitching and Flit Pooling. However, workloads like PR exhibit performance degradation even at 32 cycles. To address this, we employ Selective Flit Pooling (Section 4) and perform a similar sensitivity analysis (discussed next).

Stitching With Selective Flit Pooling. Selective Flit Pooling is realized by applying Flit Pooling only to non-latency-critical flits (Section 4). Figure 19 shows that with a selective Flit Pooling time of 32 cycles, performance improves through better stitching and reduced latency overhead. This approach prevents performance degradation in workloads like PR and SYR2K, which no longer suffer from the latency issues seen before with standard Flit Pooling at 32 cycles (Figure 18). Figure 20 shows the reduction in network bytes through Stitching and Stitching with Selective Flit Pooling. Stitching offers significant savings, with further reductions from Selective Flit Pooling. However, as the Selective Flit Pooling window increases, savings become constant. Optimal savings occur at a Selective Flit Pooling window of 32 cycles.

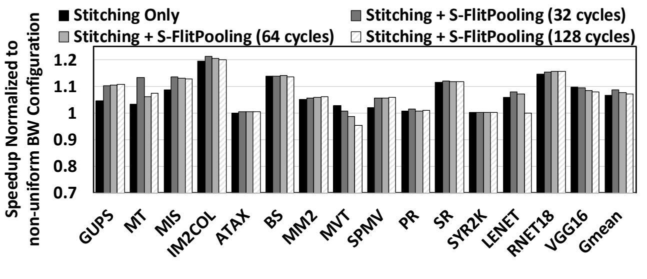

Figure 19: Performance improvements with Stitching alone and Stitching combined with Selective Flit Pooling, evaluated across Flit Pooling times from 32 to 128 cycles, normalized to a baseline non-uniform bandwidth multi-GPU configuration.

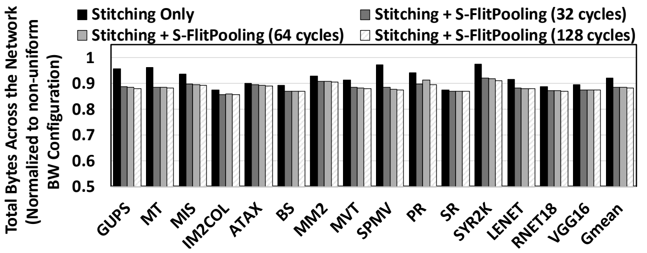

Figure 20: Reduction in network bytes with Stitching alone and Stitching + Selective Flit Pooling, evaluated over 32–128 cycle Flit Pooling intervals, normalized to a baseline non-uniform bandwidth multi-GPU configuration.

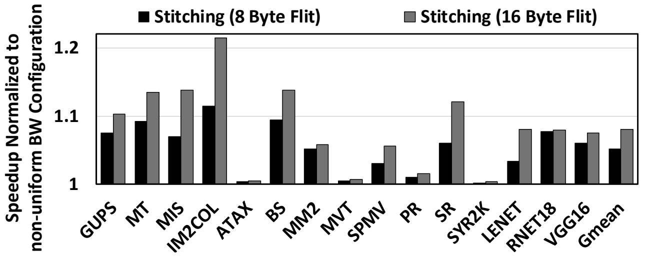

Figure 21: Performance improvement with Stitching and Selective Flit Pooling for 8 and 16 Byte Flit Size.

5.5 Other Sensitivity Studies

Flit Size. Figure 21 compares the performance of stitching with selective Flit Pooling using a smaller flit size (8B) against our baseline (16B). As shown in Figure 21, the benefits of stitching become less pronounced with a smaller flit size due to the reduced opportunity for stitching additional data. Nevertheless, stitching continues to show performance improvements, underscoring its effectiveness even when operating with significantly reduced flit sizes.

Bandwidth Ratios/Numbers. In our baseline, intra-GPU-cluster networks have 128 GB/s bandwidth, and inter-GPU-cluster networks have 16 GB/s, resulting in an 8:1 ratio. We analyze bandwidth ratios from 8:1 to 2:1, and values ranging from 128–512 GB/s for intra-GPU-cluster networks and 16–64 GB/s for inter-GPU-cluster networks. Additionally, we assess performance under a homogeneous 32 GB/s configuration for both networks. Figure 22 highlights consistent performance gains with NetCrafter across all tested bandwidth configurations, including homogeneous setups.

The most significant gains occur in bandwidth-constrained scenarios [60, 63, 77, 102], where large-scale workloads may saturate network bandwidth.

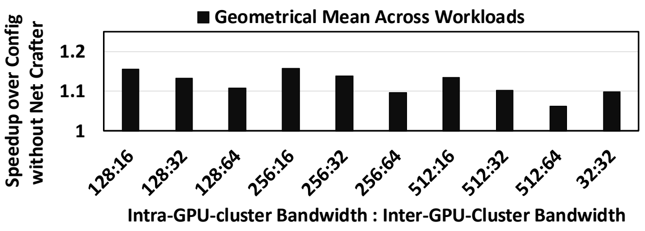

Figure 22: Performance improvements of NetCrafter with varying inter and intra-GPU-cluster bandwidth ratios, network bandwidths, and a homogeneous configuration.

6 Related Work

Network Optimizations. Finepack [62] proposed to dynamically coalesce and compress small writes/stores into a larger I/O message for reducing link-level protocol overhead. DUALOPT [10] combined fine-grained remote data transfers with coalescing, thereby maximizing inter-GPU bandwidth efficiency. Batching [41, 80] in Network Interface Cards (NICs) reduces DMA overhead and CPU load by accumulating packets. A similar technique is also deployed in NVIDIA Mellanox NICs, referred to as adaptive interrupt moderation [67, 89]. TCP piggybacking [79, 88] reduces overhead by combining control messages like ACKs, but may incur extra overhead (e.g., 12-byte TCP Timestamp Option [37]), thereby reducing data transmission efficiency. Our stitching and pooling techniques operate at a flit level in multi-GPU networks, addressing inefficiencies from mismatched packet and flit sizes.

CTA Scheduling. Arunkumar et al. [5] proposed a distributed CTA scheduling method with a first-touch page placement strategy, allocating contiguous CTAs based on locality. However, this method relies on slow GPU page faults, causing performance degradation with high SM stalling times [29, 42, 45, 71]. To mitigate this issue, recent studies have employed compile-time static analysis to infer data access patterns, enabling efficient CTA scheduling and data placement without extensive page migrations. CODA [45] uses compiler index analysis to align thread-blocks with their accessed data, while LASP (Locality-Aware Scheduling and Placement) [42] classifies data structures according to access patterns to enhance locality. LASP demonstrates improved performance compared to previous state-of-the-art methods, such as Batch+First-Touch [5] and CODA [45], establishing itself as a solid baseline for our analysis and recent works [29, 71]. NetCrafter principles are equally applicable under all CTA scheduling and page placement approaches.

Data Placement. Dashti et al. [19] presented a memory management algorithm that leverages interleaving, page replication, and page migration to address the traffic congestion issue and mitigate the cost of remote wire delays. Agarwal et al. [2] proposed a mechanism for placing pages in a hybrid memory system at runtime that detects and acts on hot and cold pages. Grit [93] optimized page placement by fine-tuning different page placement schemes for pages at runtime. Baruah et al. [9] proposed a page migration system for migrating data between GPUs in multi-GPU systems, while Khairy et al. [42] suggested creating a virtual uniform architecture consisting of discrete GPUs that are internally MCMs. Mitosis [1] proposed replicating pages across all remote nodes to improve performance. Furthermore, significant research has been conducted on developing affinity-targeted thread mapping policies [8, 11, 15, 20, 21, 34, 40, 51, 55, 72, 75, 95].

Cache Optimizations. Prior works have explored caching remote data to reduce NUMA traffic, including CC-NUMA [82], S-COMA [78], and Reactive NUMA [27], which use different granularities for on-chip caching. CARVE [97] proposed caching remote data in GPU video memory, while Milic et al. [61] adapt caching and interconnect policies based on application phases. Another line of work investigates shared versus private LLC and L1 cache designs, along with sectored cache approaches. [35, 36, 74, 99, 100]

Last-Level Cache (LLC) in GPUs can be organized as memory-side caches near memory partitions serving all SMs, or as SM-side caches on each chip serving local SMs with potentially remote data. Memory-side caching maximizes effective capacity by avoiding duplication but suffers from latency and low-bandwidth remote accesses. SM-side caching offers higher bandwidth by placing data closer to compute units, but can duplicate cache lines and add coherence complexity. Our techniques are orthogonal to the choice of LLC organization and are designed to optimize inter-GPU traffic in both memory-side and SM-side cache architectures. Notably, SM-side caching introduces coherence overhead, which creates further opportunities for our optimizations, particularly in SM-side and Shared Aware Caching [98] designs. We leave a deeper exploration of coherence-aware optimizations to future work.

In contrast to previous research that has primarily focused on cache optimizations (e.g., sectoring), efficient data placement, and message/packet coalescing (e.g., stitching and pooling) to address data movement overheads, this work, to the best of our knowledge, is the first to propose a complementary, application-specific approach that manages inter-GPU network traffic by Stitching, Trimming, and Sequencing flits as they traverse lower-bandwidth networks in a multi-GPU system.

7 Conclusions

We perform a detailed characterization of multi-GPU network traffic to demonstrate that network bandwidth across groups of multiple GPUs is a critical performance determinant. To address this bottleneck, we carefully characterized the network traffic and showed that flits traveling through the network are not fully utilized. Moreover, a large fraction of flits are never needed (or not needed immediately), while some flits are more latency-sensitive than others. Equipped with these observations, we developed NetCrafter that leverages three newly developed crafting techniques (stitching, trimming, and sequencing) to significantly reduce and streamline network traffic. As a result, NetCrafter significantly improves multi-GPU performance across a wide range of workloads.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This material is based upon work supported by the US National Science Foundation awards (#2316694, #2402804, #2402805) and performed using the computing facilities at the University of Virginia.

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. *SIGPLAN Not.* 52, 4 (2017), 631–644. doi:10.1145/3093336.3037706

[3] Anurag Agrawal and Changhoon Kim. 2020. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS).* IEEE Computer Society, 1–32.

[4] AMD. 2015. AMD APP SDK OpenCL Optimization Guide.

[5] 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. *ACM SIGARCH Computer Architecture News* 45 (06 2017), 320–332. doi:10.1145/3140659.3080231

[6] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23).* Association for Computing Machinery, New York, NY, USA, Article 52, 16 pages. doi:10.1145/3581784.3607089

[7] 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 *the Proceedings of 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Williamsburg, VA.* 503–518.

[8] Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. 2009. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Oper. Syst. Rev.* 43, 2 (apr 2009), 56–65. doi:10.1145/1531793.1531803

[9] 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 *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 596–609. doi:10.1109/HPCA47549.2020.00055

[10] Leul Belayneh, Haojie Ye, Kuan-Yu Chen, David Blaauw, Trevor Mudge, Ronald Dreslinski, and Nishil Talati. 2023. Locality-Aware Optimizations for Improving Remote Memory Latency in Multi-GPU Systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22).* Association for Computing Machinery, New York, NY, USA, 304–316. doi:10.1145/3559009.3569649

[11] 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 (10 2010). doi:10.1007/s10766-010-0136-3

[12] Ravi Budruk, Don Anderson, and Tom Shanley. 2004. *PCI express system architecture.* Addison-Wesley Professional.

[13] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. *2013 IEEE International Symposium on Workload Characterization (IISWC)* (2013), 185–195.

[14] Cen Chen, Kenli Li, Aijia Ouyang, Zhuo Tang, and Keqin Li. 2017. GPU-Accelerated Parallel Hierarchical Extreme Learning Machine on Flink for Big Data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* PP (04 2017), 1–14. doi:10.1109/TSMC.2017.2690673

[15] 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 20th Annual International Conference on Supercomputing* (Cairns, Queensland, Australia) *(ICS '06).* Association for Computing Machinery, New York, NY, USA, 353–360. doi:10.1145/1183401.1183451

[16] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3).* Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/1735688.1735702

[17] Sina Darabi, Mohammad Sadrosadati, Negar Akbarzadeh, Joël Lindegger, Mohammad Hosseini, Mohammad Hosseini, Jisung Park, Juan Gómez-Luna, Onur Mutlu, and Hamid Sarbazi-Azad. 2023. Morpheus: Extending the Last Level Cache Capacity in GPU Systems Using Idle GPU Core Resources. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22).* IEEE Press, 228–244. doi:10.1109/MICRO56248.2022.00029

[18] Sajal Dash, Isaac R Lyngaas, Junqi Yin, Xiao Wang, Romain Egele, J. Austin Ellis, Matthias Maiterth, Guojing Cong, Feiyi Wang, and Prasanna Balaprakash. 2024. Optimizing Distributed Training on Frontier for Large Language Models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference).* 1–11. doi:10.23919/ISC.2024.10528939

[19] 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 Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13).* Association for Computing Machinery, New York, NY, USA, 381–394. doi:10.1145/2451116.2451157

[20] 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 *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC).* 491–496. doi:10.1109/HPCC.2010.114

[21] Wei Ding, Yuanrui Zhang, Mahmut Kandemir, Jithendra Srinivas, and Praveen Yedlapalli. 2013. Locality-aware mapping and scheduling for multicores. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 1–12. doi:10.1109/CGO.2013.6495009

[22] Shi Dong, Xiang Gong, Yifan Sun, Trinayan Baruah, and David Kaeli. 2018. Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18).* Association for Computing Machinery, New York, NY, USA, 96–106. doi:10.1145/3184407.3184423

[23] Shi Dong and David R. Kaeli. 2017. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. *Proceedings of the General Purpose GPUs* (2017).

[24] 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 *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23).* Association for Computing Machinery, New York, NY, USA, Article 45, 13 pages. doi:10.1145/3579371.3589080

[25] Argonne Leadership Computing Facility. 2022. Aurora. https://www.alcf.anl.gov/support-center/aurora-sunspot

[26] Oak Ridge Leadership Computing Facility. 2022. Frontier User Guide - System Overview. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#id2

[27] B. Falsafi and D.A. Wood. 1997. Reactive NUMA: A Design For Unifying S-COMA And CC-NUMA. In *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture.* 229–240. doi:10.1145/264107.264205

[28] Amel Fatima, Sihang Liu, Korakit Seemakhupt, Rachata Ausavarungnirun, and Samira Khan. 2023. vPIM: Efficient Virtual Address Translation for Scalable Processing-in-Memory Architectures. In *2023 60th ACM/IEEE Design Automation Conference (DAC).* 1–6. doi:10.1109/DAC56929.2023.10247745

[29] Yuan Feng, Seonjin Na, Hyesoon Kim, and Hyeran Jeon. 2024. Barre Chord: Efficient Virtual Memory Translation for Multi-Chip-Module GPUs. In *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA).*

[30] Nitin A. Gawande, Joshua B. Landwehr, Jeff A. Daily, Nathan R. Tallent, Abhinav Vishnu, and Darren J. Kerbyson. 2017. Scaling Deep Learning Workloads: NVIDIA DGX-1/Pascal and Intel Knights Landing. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 399–408. doi:10.1109/IPDPSW.2017.36

[31] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *ArXiv* abs/1706.02677 (2017).

[32] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar).* 1–10. doi:10.1109/InPar.2012.6339595

[33] Mert Hidayetoglu, Simon Garcia De Gonzalo, Elliott Slaughter, Yu Li, Christopher Zimmer, Tekin Bicer, Bin Ren, William Gropp, Wen-Mei Hwu, and Alex Aiken. 2024. CommBench: Micro-Benchmarking Hierarchical Networks with Multi-GPU, Multi-NIC Nodes. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24).* Association for Computing Machinery, New York, NY, USA, 426–436. doi:10.1145/3650200.3656591

[34] Joshua Hursey, Jeffrey M. Squyres, and Terry Dontje. 2011. Locality-Aware Parallel Process Mapping for Multi-core HPC Systems. In *2011 IEEE International Conference on Cluster Computing.* 527–531. doi:10.1109/CLUSTER.2011.59

[35] Mohamed Assem Ibrahim, Onur Kayiran, Yasuko Eckert, Gabriel H. Loh, and Adwait Jog. 2020. Analyzing and Leveraging Shared L1 Caches in GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and*

Compilation Techniques (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 161–173. doi:10.1145/3410463.3414623

[36] Mohamed Assem Ibrahim, Onur Kayiran, Yasuko Eckert, Gabriel H. Loh, and Adwait Jog. 2021. Analyzing and Leveraging Decoupled L1 Caches in GPUs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 467–478. doi:10.1109/HPCA51647.2021.00047

[37] Van Jacobson, Robert Braden, and David Borman. 1992. TCP Extensions for High Performance. RFC 1323, https://www.rfc-editor.org/rfc/rfc1323.txt. doi:10.17487/RFC1323 Accessed: 2024-11-20.

[38] Colin L. Jermain, Graham E. Rowlands, Robert A. Buhrman, and Daniel C. Ralph. 2015. GPU-accelerated micromagnetic simulations using cloud computing. *Journal of Magnetism and Magnetic Materials* 401 (2015), 320–322.

[39] Hai Jiang, Yi Chen, Zhi Qiao, Tien-Hsiung Weng, and Kuan-Ching Li. 2015. Scaling up MapReduce-based Big Data Processing on Multi-GPU systems. *Cluster Computing* 18, 1 (mar 2015), 369–383. doi:10.1007/s10586-014-0400-1

[40] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 395–406. doi:10.1145/2451116.2451158

[41] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.

[42] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G. Rogers. 2020. Locality-Centric Data and Threadblock Management for Massive GPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1022–1036. doi:10.1109/MICRO50266.2020.00086

[43] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. doi:10.1109/ISCA45697.2020.00047

[44] Hoda Aghaei Khouzani, Pouya Fotouhi, Chengmo Yang, and Guang R. Gao. 2017. Leveraging access port positions to accelerate page table walk in DWM-based main memory. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*. 1450–1455. doi:10.23919/DATE.2017.7927220

[45] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. 2018. CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems. *ACM Trans. Archit. Code Optim.* 15, 3, Article 32 (Sept. 2018), 23 pages. doi:10.1145/3232521

[46] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based Always-On Heap Memory Safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1153–1166. doi:10.1109/MICRO50266.2020.00095

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (may 2017), 84–90. doi:10.1145/3065386

[48] Ya Le and Xuan Yang. 2015. Tiny imagenet visual recognition challenge. *CS 231N* 7, 7 (2015), 3.

[49] Yann LeCun and Corinna Cortes. 2005. The mnist database of handwritten digits.

[50] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing gpu via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 27–41.

[51] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 260–271. doi:10.1109/HPCA.2014.6835937

[52] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 253–267. doi:10.1145/3466752.3480076

[53] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. doi:10.1109/TPDS.2019.2928289

[54] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 191–202. doi:10.1109/IISWC.2018.8573483

[55] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 297–311. doi:10.1145/3037697.3037709

[56] Bingyao Li, Yanan Guo, Yueqi Wang, Aamer Jaleel, Jun Yang, and Xulong Tang. 2023. IDYLL: Enhancing Page Translation in Multi-GPUs via Light Weight PTE Invalidations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1163–1177. doi:10.1145/3613424.3614269

[57] Bingyao Li, Yueqi Wang, Tianyu Wang, Lieven Eeckhout, Jun Yang, Aamer Jaleel, and Xulong Tang. 2024. Improving Multi-Instance GPU Efficiency via Sub-Entry Sharing TLB Design. *arXiv preprint arXiv:2404.18361* (2024).

[58] 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 *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 456–470. doi:10.1109/HPCA56546.2023.10071054

[59] Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. 2021. Improving Address Translation in Multi-GPUs via Sharing and Spilling Aware TLB Design. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[60] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement learning with In-Switch Computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 279–291.

[61] 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 *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 123–135.

[62] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. 2023. FinePack: Transparently Improving the Efficiency of Fine-Grained Transfers in Multi-GPU Systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 516–529. doi:10.1109/HPCA56546.2023.10070949

[63] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. doi:10.1145/3230543.3230560

[64] NVIDIA. 2016. NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascalarchitecture-whitepaper.pdf Accessed: 2017-09-19.

[65] NVIDIA. 2017. NVIDIA DGX-1 System Architecture White paper.

[66] NVIDIA. 2018. *NVIDIA DGX-2*. https://www.nvidia.com/en-us/data-center/dgx-2/

[67] NVIDIA Corporation. 2024. Performance Tuning for Mellanox Adapters. https://enterprise-support.nvidia.com/s/article/performance-tuning-for-mellanox-adapters. Accessed: 2024-11-20.

[68] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every walk's a hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 128–141. doi:10.1145/3503222.3507718

[69] Junhyeok Park, Osang Kwon, Yongho Lee, Seongwook Kim, Gwangeun Byeon, Jihun Yoon, Prashant J. Nair, and Seokin Hong. 2024. A Case for Speculative Address Translation with Rapid Validation for GPUs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 278–292. doi:10.1109/MICRO61859.2024.00029

[70] PCI-SIG. 2017. PCI-SIG Releases PCIe® 4.0, Version 1.0. https://pcisig.com/pci-sig-releases-pcie%C2%AE-40-version-10. Accessed: August 2, 2024.

[71] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2022. Designing Virtual Memory System of MCM GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 404–422. doi:10.1109/MICRO56248.2022.00036

[72] 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. *SIGARCH Comput. Archit. News* 40, 1 (mar 2012), 235–248. doi:10.1145/2189750.2151002

[73] 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 *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 582–595. doi:10.1109/HPCA47549.2020.00054

[74] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 86–98.

[75] 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 *2009 IEEE Symposium on Computers and Communications*. 811–817. doi:10.1109/ISCC.2009.5202271

[76] Ahmed Sanaullah, Saiful A. Mojumder, Kathleen M. Lewis, and Martin C. Herbordt. 2016. GPU-accelerated charge mapping. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2016.7761599

[77] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. In *Symposium on Networked Systems Design and Implementation*.

[78] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. 1995. An argument for simple COMA. In *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*. 276–285. doi:10.1109/HPCA.1995.386535

[79] Luca Scalia, Fabio Soldo, and Mario Gerla. 2007. PiggyCode: a MAC layer network coding scheme to improve TCP performance over wireless networks. In *IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference*. IEEE, 3672–3677.

[80] Henry N Schuh, Arvind Krishnamurthy, David Culler, Henry M Levy, Luigi Rizzo, Samira Khan, and Brent E Stephens. 2024. CC-NIC: a Cache-Coherent Interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 52–68.

[81] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[82] P. Stenstrom, T. Joe, and A. Gupta. 1992. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings the 19th Annual International Symposium on Computer Architecture*. 80–91. doi:10.1109/ISCA.1992.753306

[83] C. B. Stunkel, R. L. Graham, G. Shainer, M. Kagan, S. S. Sharkawi, B. Rosenburg, and G. A. Chochia. 2020. The high-speed networks of the Summit and Sierra supercomputers. *IBM Journal of Research and Development* 64, 3/4 (2020), 3:1–3:10. doi:10.1147/JRD.2020.2967330

[84] Yifan Sun, Nicolas Bohm Agostini, Dong Shi, and David Kaeli. 2019. Summarizing CPU and GPU design trends with product data. *arXiv preprint arXiv:1911.11313* (2019).

[85] 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 Kaeli. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 197–209.

[86] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Rafael Ubal, Xiang Gong, Shane Treadway, Yuhui Bao, Vincent Zhao, José L Abellán, et al. 2018. Mgsim+ mgmark: A framework for multi-gpu system research. *arXiv preprint arXiv:1811.02884* (2018).

[87] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter Mccardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. doi:10.1109/IISWC.2016.7581262

[88] Konstantin Taranov, Fabian Fischer, and Torsten Hoefler. 2022. Efficient RDMA Communication Protocols. *arXiv preprint arXiv:2212.09134* (2022).

[89] Mellanox Technologies. [n. d.]. Performance Tuning Guidelines for Mellanox Network Adapters.

[90] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. 2018. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 661–672. doi:10.1109/SC.2018.00055

[91] Guan Wang, Chuanqi Zang, Lei Ju, Mengying Zhao, Xiaojun Cai, and Zhiping Jia. 2018. Shared Last-Level Cache Management and Memory Scheduling for GPGPUs with Hybrid Main Memory. *ACM Trans. Embed. Comput. Syst.* 17, 4, Article 77 (jul 2018), 25 pages. doi:10.1145/3230643

[92] Jun Wang, Eric Papenhausen, Bing Wang, Sungsoo Ha, Alla Zelenyuk, and Klaus Mueller. 2017. Progressive clustering of big data with GPU acceleration and visualization. In *2017 New York Scientific Data Summit (NYSDS)*. 1–9. doi:10.1109/NYSDS.2017.8085036

[93] Yueqi Wang, Bingyao Li, Aamer Jaleel, Jun Yang, and Xulong Tang. 2024. GRIT: Enhancing Multi-GPU Performance with Fine-Grained Dynamic Page Placement. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1080–1094. doi:10.1109/HPCA57654.2024.00085

[94] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. 2015. Deep Image: Scaling up Image Recognition. *ArXiv* abs/1501.02876 (2015).

[95] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 76–88. doi:10.1109/HPCA.2015.7056023

[96] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 339–351. doi:10.1109/MICRO.2018.00035

[97] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 339–351. doi:10.1109/MICRO.2018.00035

[98] Shiqing Zhang, Mahmood Naderan-Tahan, Magnus Jahre, and Lieven Eeckhout. 2023. SAC: Sharing-Aware Caching in Multi-Chip GPUs. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 43, 13 pages. doi:10.1145/3579371.3589078

[99] Xia Zhao, Almutaz Adileh, Zhibin Yu, Zhiying Wang, Aamer Jaleel, and Lieven Eeckhout. 2019. Adaptive Memory-Side Last-level GPU Caching. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 411–423.

[100] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Selective Replication in Memory-Side GPU Caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 967–980. doi:10.1109/MICRO50266.2020.00082

[101] 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*. 544–559.

[102] Huan Zhou, Vladimir Marjanovic, Christoph Niethammer, and José Gracia. 2015. A Bandwidth-Saving Optimization for MPI Broadcast Collective Operation. In *2015 44th International Conference on Parallel Processing Workshops*. 111–118. doi:10.1109/ICPPW.2015.20