Journal of Systems Architecture 160 (2025) 103339

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

REC: Enhancing fine-grained cache coherence protocol in multi-GPU systems

Gun Ko, Jiwon Lee, Hongju Kal, Hyunwuk Lee, Won Woo Ro [*]

Yonsei University, 50 Yonsei-ro Seodaemun-gu, Seoul, 03722, Republic of Korea

ARTICLE INFO

Keywords:
Multi-GPU
Data sharing
Cache coherence
Cache architecture

ABSTRACT

With the increasing demands of modern workloads, multi-GPU systems have emerged as a scalable solution, extending performance beyond the capabilities of single GPUs. However, these systems face significant challenges in managing memory across multiple GPUs, particularly due to the Non-Uniform Memory Access (NUMA) effect, which introduces latency penalties when accessing remote memory. To mitigate NUMA overheads, GPUs typically cache remote memory accesses across multiple levels of the cache hierarchy, which are kept coherent using cache coherence protocols. The traditional GPU bulk-synchronous programming (BSP) model relies on coarse-grained invalidations and cache flushes at kernel boundaries, which are insufficient for the fine-grained communication patterns required by emerging applications. In multi-GPU systems, where NUMA is a major bottleneck, substantial data movement resulting from the bulk cache invalidations exacerbates performance overheads. Recent cache coherence protocol for multi-GPUs enables flexible data sharing through coherence directories that track shared data at a fine-grained level across GPUs. However, these directories limited in capacity, leading to frequent evictions and unnecessary invalidations, which increase cache misses and degrade performance. To address these challenges, we propose REC, a low-cost architectural solution that enhances the effective tracking capacity of coherence directories by leveraging memory access locality. REC coalesces multiple tag addresses from remote read requests within common address ranges, reducing directory storage overhead while maintaining fine-grained coherence for writes. Our evaluation on a 4-GPU system shows that REC reduces L2 cache misses by 53.5% and improves overall system performance by 32.7% across a variety of GPU workloads.

1. Introduction

Multi-GPU systems have emerged to meet the growing demands of modern workloads, offering scalable performance beyond what a single GPU can deliver. However, as multi-GPU architectures scale in size and complexity [1,2], managing memory across multiple GPUs becomes increasingly challenging [3–7]. One of the primary challenges arises from the bandwidth discrepancy between local and remote memory, commonly known as the Non-Uniform Memory Access (NUMA) effect [3,4]. To mitigate the NUMA penalty, GPUs generally rely on caching remote memory accesses, allowing them to be served with local bandwidth [5,8–10]. This caching strategy is often extended across multiple levels of the cache hierarchy, including both private on-chip caches and shared caches [3,4,11,12], to better accommodate the diverse access patterns of emerging workloads.

While remote data caching offers significant performance benefits in multi-GPU systems, it also requires extending coherence throughout the cache hierarchy. Conventional GPUs rely on a simple software-inserted bulk-synchronous programming (BSP) model [11], which performs cache invalidation and flush operations at the start and end of each kernel. However, as recent GPU applications increasingly require more frequent and fine-grained communication both within and across kernels [11,13–15], these frequent synchronizations can lead to substantial cache operation and data movement overheads. Additionally, precisely managing the synchronizations places additional burdens on programmers, complicating the optimization of multi-GPU systems.

Ren et al. [11] proposed HMG, a hierarchical cache coherence protocol designed for L2 caches in large-scale multi-GPU systems. HMG employs coherence directories to record cache line addresses and their associated sharers upon receiving remote read requests. Any writes to these addresses trigger invalidations. Once capacity is reached, existing entries are evicted from the directory, triggering invalidation requests to the sharer GPUs. These invalidations are unnecessary, as the corresponding cache lines do not immediately require coherence to be maintained. When GPUs access data across a wide range of addresses, significant directory insertions lead to a number of unnecessary invalidations for cache lines that have not yet been fully utilized. Subsequent accesses to these cache lines result in cache misses, requiring data to be fetched again over bandwidth-limited inter-GPU links.

[*] Corresponding author.
E-mail address: wro@yonsei.ac.kr (W.W. Ro).

https://doi.org/10.1016/j.sysarc.2025.103339
Received 10 September 2024; Received in revised form 27 December 2024; Accepted 5 January 2025
Available online 9 January 2025

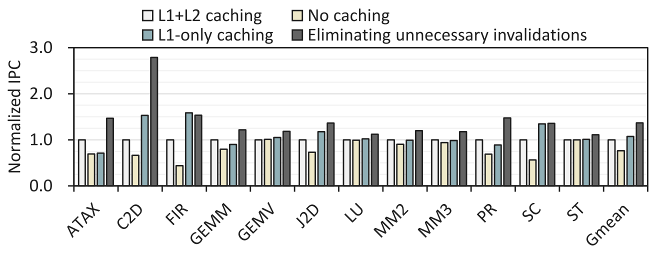

Fig. 1. Performance of each caching scheme normalized to a system that enables remote data caching in both L1 and L2 caches using software and hardware coherence protocols, respectively. "No caching" refers to a system that disables remote data caching, simplifying coherence.

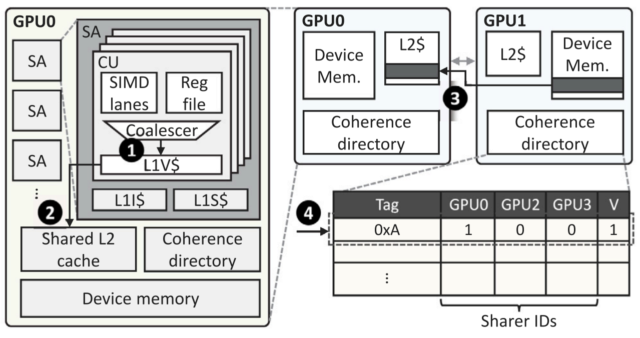

Fig. 2. Baseline multi-GPU system. Each GPU has a coherence directory that records and tracks the status of shared data at given addresses along with the corresponding sharer IDs.

To evaluate the implications of the coherence protocol, we measure the performance impact of unnecessary invalidations on a 4-GPU system that caches remote data in both L1 and L2 caches. L1 caches are assumed to be software-managed, while L2 caches are managed under fine-grained invalidation through coherence directories. As Fig. 1 shows, there exists a significant performance opportunity in eliminating unnecessary invalidations caused by frequent directory evictions. Increasing the size of the coherence directory can delay evictions and the corresponding invalidation requests, but at the cost of increased hardware. Our observations indicate that to eliminate unnecessary invalidations, the size of the coherence directory would need be substantially increased, accounting for 30.4% of the L2 cache size. As the size of GPU L2 caches continues to grow [16,17], the aggregate storage overhead of coherence directories becomes substantial, causing inefficiency in scaling for multi-GPU environment (discussed in Section 3.3).

In this paper, we propose Range-based Directory Entry Coalescing (REC), an architectural solution that mitigates unnecessary invalidation overhead by increasing the effective tracking capacity of the coherence directory without incurring significant hardware costs. Our key insight is that since directory updates are performed upon receiving remote read requests, leveraging memory access locality provides an opportunity to coalesce multiple tag addresses of shared data based on their common address range. To achieve this, we employ a coherence directory design, which aggregates data from incoming remote reads that share a common base address within the same address range, storing only the offset and the sharer IDs. We reduce the storage requirements of directory entries by designing them in a base-and-offset format, recording the common high-order bits of addresses and using a bit-vector to indicate the index of each coalesced entry within the target range. For incoming writes, if they are found in the coherence directory, invalidations are propagated only to the corresponding address, maintaining fine-grained coherence in multi-GPU systems.

To summarize, this paper makes the following contributions:

- We identify a performance bottleneck of fine-grained shared data tracking mechanisms in multi-GPU systems. Our analysis demonstrates that such methods generate unnecessary invalidations at coherence directory evictions, which incurs a significant performance bottleneck due to increased cache miss rates.
- We show that simply employing larger coherence directories incurs significant storage overhead. Our analysis shows that the baseline multi-GPU system requires a 12× increase in the directories to eliminate redundant invalidations.
- We propose REC which increases effective coverage of the coherence directory by enabling each entry to coalesce and track multiple memory addresses along with the associated sharers. By reducing the L2 cache misses by 53.5%, REC improves overall performance by 32.7% on average across our evaluated GPU workloads.

2. Background

2.1. Multi-GPU architecture

The slowdown of transistor scaling has made it increasingly difficult for single GPUs to meet the growing demands of modern workloads. Alternatively, multi-GPU systems have emerged as a viable path forward, offering enhanced performance and memory capacity by leveraging multiple GPUs connected using high-bandwidth interconnects such as PCIe and NVLink [18]. However, these inter-GPU links are likely to have bandwidth that falls far behind the local memory bandwidth [3,4,8]. The NUMA effect that arises from this large bandwidth gap can significantly impact multi-GPU performance, making it crucial to optimize remote access bottlenecks to maximize efficiency.

Fig. 2 illustrates the architectural details of our target multi-GPU system. Each GPU is divided into several SAs, with each comprising a number of CUs. Every CU has its own private L1 vector cache (L1V$), while the L1 scalar cache (L1S$) and L1 instruction cache (L1I$) are shared across all CUs within an SA. Additionally, each GPU contains a larger L2 cache that is shared across all SAs. When a data access misses in the local cache hierarchy, it is forwarded to either local or remote GPU memory, depending on the data location. For local memory accesses, the cache lines are stored in both the shared L2 cache and the L1 cache private to the requesting CU. In the case of remote-GPU memory accesses, the data can be cached either only in the L1 cache of the requesting CU [4,5,8] or in both the L2 and L1 caches [3,11,12]. Caching data in remote memory nodes helps mitigate the performance degradation caused by accessing remote memory nodes.

2.2. Remote data caching in multi-GPU

While caching remote data only in the L1 cache can save L2 cache capacity, it limits the sharing of remote data among CUs. As a result, such an approach provides lower performance gain when unnecessary invalidation overhead is eliminated in its counterpart, as shown in Fig. 1. For this reason, in this study, we assume the baseline multi-GPU architecture allows caching of remote data in both L1 and L2 caches.

A step-by-step process of remote data caching is shown in Fig. 2. Upon generating a memory request, an L1 cache lookup is performed by the requesting CU (❶). When data is not present in the L1, an L2 cache lookup is generated to check if the remote data is cached in the L2 (❷). If the data is found in the L2 cache, it is returned to the requesting CU and cached in its local L1 cache. If the data is not found in the L2 cache, the request is forwarded to the remote GPU memory at the given physical address. Subsequently, the requested data is returned at a cache line granularity and cached in both the L1 and L2 caches (❸). At the same time, the coherence directory, which maintains information about data locations across multiple GPUs, is

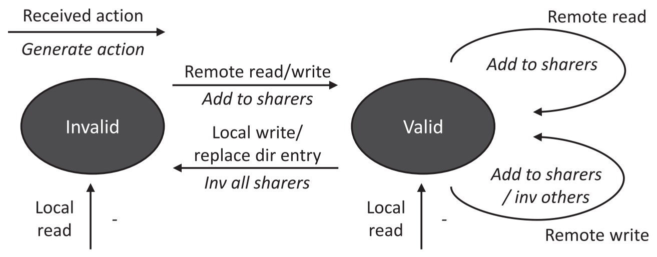

Fig. 3. Coherence protocol flows in detail. The baseline hardware protocol has two stable states: valid and invalid, with no transient states or acknowledgments required for write permissions.

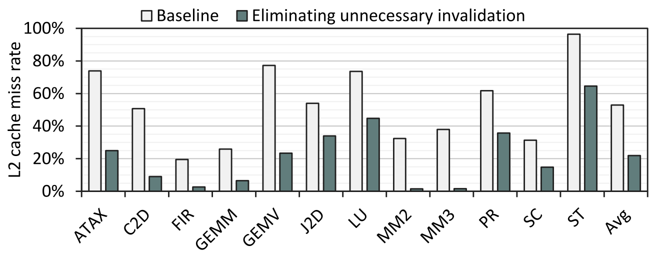

Fig. 4. L2 cache miss rates in baseline and idealized system where no invalidations are propagated by coherence directory evictions. Cold misses are excluded from the results.

updated with the corresponding entry and the sharer GPU (❹). Writes to remote data in the home GPU are also performed in the local L2 cache, following the write-through policy, as the corresponding GPU may access the written data in the future. Remote writes arriving at the home GPU trigger invalidation messages to be sent out to the sharer GPU(s), and the requesting GPU is recorded as a sharer (❹).

2.3. Cache coherence in multi-GPU

Existing hardware protocols, such as GPU-VI [19], employ coherence directories to track sharers (i.e., L1s) and propagate write-initiated cache invalidations within a single GPU. Bringing the notion into multi-GPU environments, Ren et al. proposed HMG [11], a hierarchical design that efficiently manages both intra- and inter-GPU coherence. HMG includes two layers for selecting home nodes to track sharers: (1) the inter-GPU module (GPM) level that selects a home GPM within a GPU and (2) the inter-GPU level that selects a home GPU across the entire system. A GPM is a chiplet in multi-chip module GPUs. With this, HMG reduces the complexity of tracking and maintaining coherence across a large number of sharers. HMG also optimizes performance by eliminating all transient states and most invalidation acknowledgments, leveraging weak memory models in modern GPUs [11].

Each GPU has a coherence directory attached to its L2 cache, managed by the cache controllers. The directory is organized in a set-associative structure, and each entry contains the following fields: tag, sharer IDs, and coherence state. The tag field stores the cache line address for the data copied and fetched by the sharer. The sharer ID field is a bit-vector representing the list of sharers, excluding the home GPU. Each entry is in one of two stable states: valid or invalid. Unlike HMG [11], the baseline coherence directory tracks one cache line per each entry. In contrast, a directory entry in HMG is designed to track four cache lines using a single tag address and sharer ID field, which limits its ability to manage each cache line at a fine granularity. Consequently, a write to any address tracked by a directory entry may unnecessarily invalidate other cache lines within the same range, potentially causing inefficiencies in remote data caching. We discuss the importance of reducing unnecessary cache line invalidations in detail in Section 3.1. Like typical memory allocation in multi-GPU systems, the physical address space is partitioned among the GPUs in the system. Therefore, data at any given physical address is designated to one GPU (i.e., the home GPU), and every access by a remote GPU references the coherence directory of the home GPU. For example, in Fig. 2, GPU0 requests data at address 0xA from GPU1, which is the home GPU; the corresponding entry is then inserted into the directory of GPU1 with the relevant information.

Fig. 3 shows the detailed state transitions and actions initiated by the coherence directory. Note that *local* and *remote* refer to the sources of memory requests received: *local* refers to accesses from the local CUs, and *remote* refers to accesses from the remote GPUs.

Local reads: Local read requests arriving at the L2 cache are directed to either locally- or remotely-mapped data. On cache hits, the data is returned and guaranteed to be consistent because it is either the most up-to-date data (if mapped to local DRAM) or correctly managed by the protocol (if mapped to remote GPU). On cache misses, the requests are forwarded to either local DRAM or a remote GPU. In all cases, the directory of the requesting GPU remains unchanged.
Remote reads: For remote reads that arrive at the home GPU, the coherence directory records the ID of the requesting GPU at the given cache line address. If the line is already being tracked (i.e., the entry is found and valid), the directory simply adds the requester to the sharer field and keeps the entry in the valid state. If the line is not being tracked, the directory finds an empty spot to allocate a new entry and marks it as valid. When the directory is full and every entry is valid, it evicts an existing entry and replaces it with the new entry (discussed below).
Local writes: Local writes to data mapped to the home GPU memory look up the directory to find whether a matching entry at the line address exists. If found, invalidations are propagated to the recorded sharers in the background, and the directory entry becomes invalid.
Remote writes: By default, L2 caches use a write-back policy for local writes. As described in Section 2.2, remote writes update both the L2 cache of the requester and local memory, similar to a write-through policy. Consequently, the directory maintains the entry as valid by adding the requester to the sharer list and sends out invalidations to other sharers recorded in the original entry.
Directory entry eviction/replacement: Coherence directories are implemented in a set-associative structure. Thus, capacity and conflict misses occur as directory lookups are initiated by the read requests continuously received from remote GPUs. To notify that the information in the evicted entry is no longer traceable, invalidations are sent out as with writes.
Acquire and release: At the start of a kernel, invalidations are performed in L1 caches as coherence is maintained using software bulk synchronizations. However, the invalidations are not propagated beyond L1 caches, as L2 caches are kept coherent with the fine-grained directory protocol. Release operations flush dirty data in both L1 and L2 caches.

3. Motivation

In multi-GPU systems, coherence is managed explicitly through cache invalidations to ensure data consistency across multiple GPUs. When invalidation requests are received, sharer GPUs must look up and invalidate the corresponding cache lines. Subsequent accesses to these invalidated cache lines result in cache misses, which are then forwarded to the home GPU. This, in turn, can negate the performance benefits of local caching as it undermines the effectiveness of caching mechanisms intended to reduce remote access bottlenecks. In this section, we analyze the behavior of cache invalidation and its impact on the overall

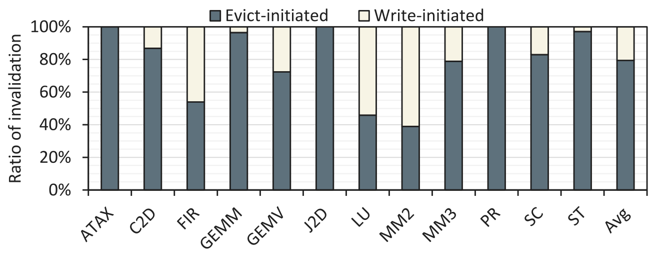

Fig. 5. Fraction of evict-initiated and write-initiated invalidations in the baseline multi-GPU system. The results are based on invalidation requests that hit in the sharer-side L2 caches.

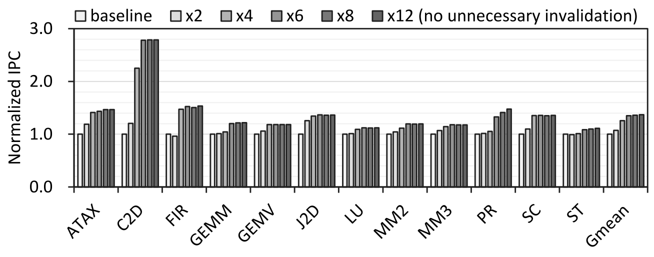

Fig. 6. Performance impact of increasing coherence directory sizes. To eliminate unnecessary invalidations, GPUs require a directory size up to 12× larger than the baseline.

performance of multi-GPU systems. We identify the sources of invalidation and explore a straightforward solution to mitigate the associated bottlenecks. Our experiments are conducted using MGPUSim [20], a multi-GPU simulation framework that we have extended to support the hardware cache coherence protocol. The detailed configuration is provided in Table 2.

3.1. Impact of cache invalidation

To ensure data consistency across multiple GPUs, invalidation requests are propagated by the home GPU in two cases: (1) when write requests are received and (2) when an entry is evicted from the coherence directory due to capacity and conflict misses. Invalidation requests triggered by writes are crucial for maintaining data consistency, as they ensure that no stale data is accessed in the sharer GPU caches. On the other hand, invalidations generated by directory eviction aim to notify the sharers that the coherence information is no longer traceable, even if the data is still valid. A detailed background on the protocol flows with invalidations is given in Section 2.3.

Broadcasting invalidations does not significantly impact cache efficiency if the cache lines are already evicted or no longer in use. However, when applications exhibit frequent remote memory accesses, the generation of new directory entries increases invalidation requests from eviction, invalidating the associated cache lines prematurely. These premature invalidations lead to higher cache miss rates, as subsequent accesses to the invalidated cache lines result in misses. As remote data misses exacerbates NUMA overheads, they need to be reduced to improve multi-GPU performance.

Fig. 4 shows the impact of cache miss rate when eliminating unnecessary invalidations across the benchmarks listed in Table 3 running on a 4-GPU system. The figure demonstrates that the baseline system experiences a cache miss rate more than double (average 2.4×) that of the idealized system without the unnecessary invalidation. This increase is mainly due to frequent invalidation requests, which prematurely invalidate cache lines before they can be fully utilized, leading to an increase in the number of remote memory accesses. The result strongly motivates us to further study the source of these frequent invalidations to improve efficiency of remote data caching in multi-GPU systems.

To demonstrate the performance opportunity, Fig. 1 presents a study showing the performance of idealized caching without the invalidation overhead. With no invalidations to unmodified cache lines, remote data can be fully utilized as needed until they are naturally replaced by the typical cache replacement policy. The performance of the baseline and ideal system is represented in the first and fourth bars, respectively, in Fig. 1. The result shows that an ideal system with no unnecessary cache invalidation overheads outperforms the baseline by up to 2.79× (average 36.9%). As demonstrated by Figs. 1 and 4, reducing premature cache invalidations is crucial in improving efficiency of remote data caching in multi-GPU systems.

3.2. Source of premature invalidation

As described in Section 2.3, when a coherence directory becomes full, the GPU needs to evict an old entry and replace it with a new one upon receiving a remote read request; an invalidation request must be sent out to the sharer(s) in the evicted entry. Fig. 5 shows the distribution of invalidations triggered by directory eviction and write requests, referred to as evict-initiated and write-initiated invalidations, respectively. The measurements are taken based on the invalidations that are hit in the sharer-side L2 caches after receiving the requests. We observe that significant amount of invalidations (average 79.5%) are performed by the requests from directory evictions in the home GPUs. These invalidations, considered unnecessary as they do not require immediate action, should be delayed until remote GPUs have full use of the data.

We also show the percentage of write-initiated invalidations in Fig. 5. One can observe that applications such as `FIR`, `LU`, and `MM2` experience a significant number of invalidations due to write requests. These workloads exhibit fine-grained communication within and across dependent kernels, necessitating the invalidation of corresponding cache lines in the remote L2 cache upon any modification to the shared data. Although the applications exhibit a high percentage of write-initiated invalidations, their impact on cache miss rates may be negligible if the GPUs do not subsequently require access to the invalidated cache lines. Nonetheless, the results from Fig. 4 clearly demonstrate the importance of minimizing unnecessary cache invalidations.

So far, we have discussed how prematurely invalidating remote data leads to increased cache miss rates, which negatively impacts multi-GPU performance. We also show that a large fraction of invalidation requests stems from directory evictions, which frequently occur due to the high volume of remote accesses. These accesses trigger numerous directory updates, overwhelming the baseline coherence directory's capacity to effectively manage coherence. A straightforward solution to mitigate premature invalidations is to increase the size of the coherence directory, providing more coverage to track sharers and reducing eviction rates. In the following section, we analyze the performance impact of larger coherence directory sizes. It is important to note that this paper primarily focuses on delaying invalidations caused by directory evictions, as write-initiated invalidations are necessary and must be performed immediately for correctness.

3.3. Increasing directory sizes

A simple approach to delay directory evictions, thereby minimizing premature invalidations, is to increase the size of coherence directories. Limited directory sizes lead to significant evict-initiated invalidations, which can undermine the performance benefits of local caching. To quantify the benefits of larger directories, we conduct a quantitative analysis of performance improvements with increasing directory sizes.

In our simulated 4-GPU system, each GPU has an L2 cache size of 2 MB, with each cache line being 64B. Each coherence directory tracks

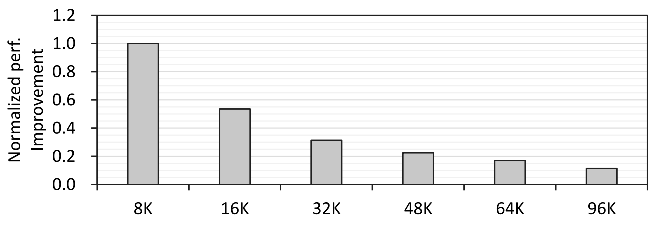

Fig. 7. Average performance improvement per increased directory storage in the baseline coherence directory design. The results are normalized to the system with 8K-entry coherence directory.

the identity of all sharers excluding the home GPU (i.e., three GPUs). To cover the entire L2 cache space for three GPUs, an ideal coherence directory would require approximately 96K entries, or about 12× the baseline 8K entries.

Fig. 6 illustrates the normalized performance for increasing the directory sizes by 2×-12× the baseline. With an ideal directory size, unnecessary invalidations from directory evictions can be eliminated, leaving only write-initiated invalidations. The results show that applications exhibit significant performance gains as the directory size increases, with some benchmarks (e.g., ATAX, PR, and ST) requiring 8×-12× the baseline size to achieve the highest speed-up. Specifically, benchmarks such as PR and ST show irregular memory access patterns that span a wide address range, leading to higher chances of conflict misses when updating coherence directories. Most other tested benchmarks require up to six times the baseline directory size to achieve maximum attainable performance; the average speedup with six times the size is 1.35×.

Each entry in the coherence directory comprises a tag, sharer list, and coherence state. We assume 48 bits for tag addresses, a 3-bit vector for tracking sharers, and one bit for the directory entry state; thus, each entry requires a total of 52 bits of storage. Our baseline directory implementation has 8K entries and occupies approximately 2.5% of the L2 cache [11]. Therefore, the storage cost of the baseline directory in each GPU is 52 × 8192/8/1024 = 52 kB, assuming 8 bits per byte and 1024 bytes per kilobyte. From our observation in Fig. 6, applications require directory sizes from 6× up to 12× the baseline to achieve maximum performance. This corresponds to a total storage cost of 312-624 kB, which is an additional 15.2–30.4% of the L2 cache size. While increasing directory size can significantly improve performance, the associated hardware costs are substantial. To show the inefficiency of simply scaling directory sizes, we calculate the performance per storage using the results in Fig. 6 and the number of directory entries. Fig. 7 illustrates the results relative to the baseline with 8K entries, showing that performance improvements per increased storage do not scale proportionally with larger coherence directories. Additionally, since GPU applications require different directory sizes to achieve maximum performance, simply increasing the directory size is not an efficient solution. Moreover, as GPU L2 caches continue to grow [16,17], the cost of maintaining proportionally larger coherence directories will only amplify these overheads. Therefore, improving coherence directory coverage without significant storage overhead motivates the need for more efficient fine-grained hardware protocols in multi-GPU systems.

4. REC architecture

This work aims to enhance coherence directory coverage while avoiding significant hardware overhead, overall reducing unnecessary cache invalidations in multi-GPU systems. We introduce REC, an architecture that coalesces directory entries by leveraging the spatial locality in memory accesses observed in GPU workloads. In this section, we provide an overview of REC design and discuss its integration with existing multi-GPU coherence protocols.

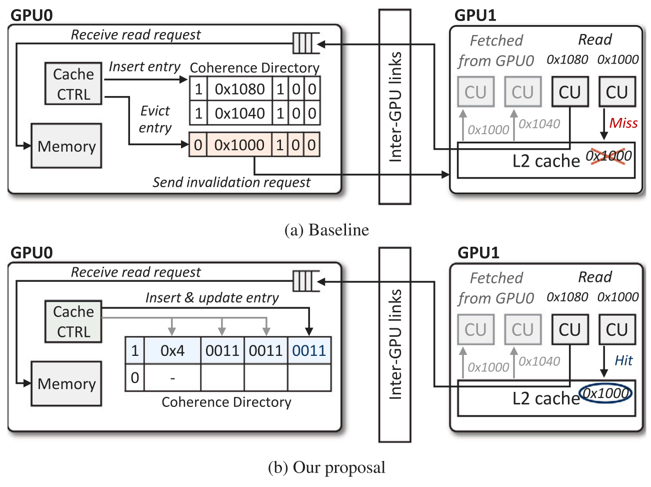

Fig. 8. A high-level overview of (a) baseline and (b) proposed REC architecture with simplified 2-entry coherence directories. The figure illustrates a scenario where GPU1 accesses memory of GPU0 in order of 0 × 1000, 0 × 1040, 0 × 1080, and 0 × 1000 by each CU. In the baseline directory, entry that tracks status of data at 0 × 1000 is evicted for recording the address 0 × 1080. The proposed directory coalesces three addresses with same base address into one entry.

4.1. Hardware overview

As shown in Section 3.2, a significant fraction of cache invalidations are generated by the frequent directory evictions. These invalidations lead to increased cache misses, as data is prematurely invalidated from the cache, requiring subsequent accesses to fetch the data from remote memory. While simply increasing the directory size can address this bottleneck, the associated cost of hardware can become substantial. To address this, we propose REC, an architectural solution that compresses remote GPU access information, retaining as much data as possible before eviction occurs. It aggregates data from incoming remote read requests so that (1) multiple reads to the same address range share a common base address, storing only the offset and source GPU information, and (2) the coalescing process does not result in any loss of information, maintaining the accuracy of the coherence protocol. We now discuss the design overview of REC and the details of the associated hardware components.

Fig. 8(a) shows how the baseline GPU handles a sequence of incoming read requests. The cache controller records the tag addresses and the corresponding sharer IDs in the order that the requests arrive. When the coherence directory reaches its capacity, the cache controller follows a typical FIFO policy to replace the oldest entry with a new one within the set. Once an entry is evicted, the information it held can no longer be tracked, triggering an invalidation request to be sent to the GPU listed in the entry. Upon receiving this request, the sharer GPU checks its L2 cache and invalidates the corresponding cache line, leading to a cache miss on any subsequent access to the cache line.

To delay invalidations caused by directory evictions without significant hardware overhead, we introduce the REC architecture, which enhances the baseline coherence directory by leveraging spatial locality for merging multiple addresses into a single entry. As illustrated in Fig. 8(b), REC stores tag addresses with common high-order bits as a single entry using a base-plus-offset format. When a new read request matches the base address in an existing entry, the offset and sharer information are appended to that entry, reducing the need for additional entries and delaying evictions. The base address represents the shared high-order bits, covering a range of addresses and reducing the storage required compared to storing full tag addresses individually. Additionally, REC uses position bits to efficiently track multiple addresses within the specified range, further minimizing storage overhead.

Table 1
Trade-offs between addressable range and storage for each entry. Note that one valid bit, not shown in the table, is included in the overall calculation.

	Addressable range				
	64B	128B	256B	1 kB	4 kB
Base address bits	48	41	40	38	36
Position/Sharer bits	–/3	2/6	4/12	16/48	64/192
Total bits per entry	52	50	57	103	293

Table 2
Baseline GPU configuration.

Parameter	Configuration
Number of SAs	16
Number of CUs	4 per SA
L1 vector cache	1 per CU, 16 kB 4-way
L1 inst cache	1 per SA, 32 kB 4-way
L1 scalar cache	1 per SA, 16 kB 4-way
L2 cache	2 MB 16-way, 16 banks, write-back
Cache line size	64B
Coherence directory	8K entries, 8-way
DRAM capacity	4 GB HBM, 16 banks
DRAM bandwidth	1 TB/s [11]
Inter-GPU bandwidth	300 GB/s, bi-directional

Table 3
Tested workloads.

Benchmark	Abbr.	Memory footprint
Matrix transpose and vector multiplication [21]	ATAX	128 MB
2-D convolution [21]	C2D	512 MB
Finite impulse response [22]	FIR	128 MB
Matrix-multiply [21]	GEMM	128 MB
Vector multiplication and matrix addition [21]	GEMV	256 MB
2-D jacobi solver [21]	J2D	128 MB
LU decomposition [21]	LU	128 MB
2 matrix multiplications [21]	MM2	128 MB
3 matrix multiplications [21]	MM3	64 MB
PageRank [22]	PR	256 MB
Simple convolution [23]	SC	512 MB
Stencil 2D [24]	ST	128 MB

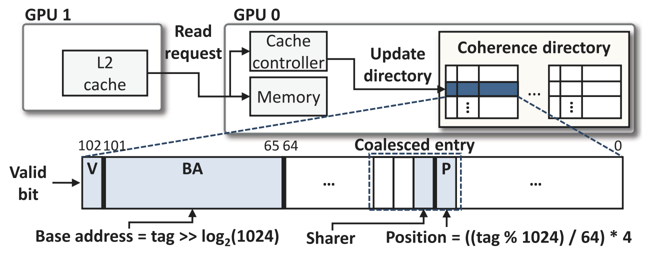

Fig. 9. Coherence directory entry structure for 64B cache lines. In our design, each entry stores up to 16 coalesced entries based on 1 kB range.

Determining the address range within which REC coalesces entries is one of the key design considerations, as it directly impacts the number of bits required for each entry. Table 1 shows a list of design choices for implementing REC with varying addressable ranges and their potential trade-offs. The number of required base address bits is calculated using 2^n = addressable_range, where n is the number of bits right-shifted from the original tag address. Also, the number of required position bits is determined by the maximum number of coalesceable cache line addresses within the target range, assuming 64B line size. Then, the number of sharer bits required is (n-1)×num_position_bits, where n is the number of GPUs. For example, if REC is designed to coalesce with addressable range of 256B, each entry would require 40, 4, and 12 bits for base address, position, and sharer fields, respectively. Lastly, one valid bit is added to each entry. In Table 1, we show the total bits required per entry under the addressable ranges from 128B to 4 kB

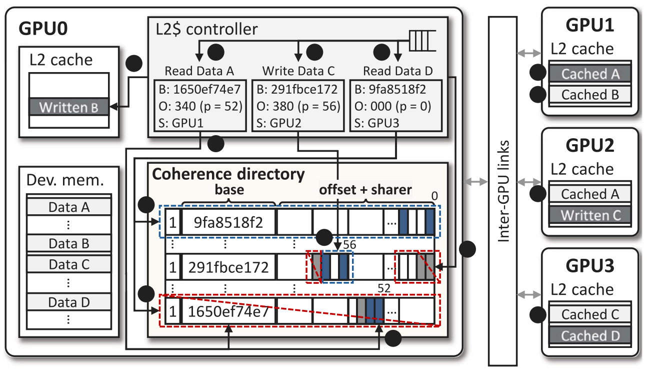

Fig. 10. Overview of the REC protocol flows. In the example coherence directory, entry insertion and offset addition operations are highlighted in blue, while eviction and offset deletion operations are shown in red.

for comparing the storage costs. REC designs with larger addressable ranges can benefit from increased directory coverage but at the cost of storage. In the evaluation of this paper, we tested various addressable ranges for REC. Each design is configured to coalesce the maximum number of offsets within its specified range. Later in the results, we confirm that a 1 kB coalesceable range offers the best trade-off, balancing reasonable size overhead per entry with the ability to coalesce a significant number of entries before evictions occur (discussed in Section 5.2).

Based on these findings, the format of a directory entry is as illustrated in Fig. 9. Each entry comprises a base address, coalesced entries, and a valid bit. When the first remote read request arrives at the home GPU, the cache controller sets the base address by right-shifting the tag address by the number of bits needed to represent the offset within the specified range. For a 48-bit tag, the address is right-shifted by 10 bits (considering a 64B-aligned 1 kB range), and the resulting bits from positions 64 to 101 are used to store the base address. The coalesced entry is identified using the offset within the 1 kB range, represented by a position bit, followed by three bits for recording the sharers. The position bit is calculated as:

$$p = \left(\frac{\text{Tag mod } m}{64} \right) \times (n + 1)$$

where m denotes the coalescing range, and n is the number of sharers, which are set to 1 kB and 3, respectively. Once the position is determined, the corresponding position and the sharer bit are set to 1 using bitwise OR operation. Given that the 1 kB range allows each entry to record up to 16 individual tag addresses, we use the lower 64 bits to store the coalesced entries. Furthermore, the position bit can also function as the valid bit for each coalesced entry, meaning only one valid bit is necessary to indicate whether the entire entry is valid or not.

4.2. REC protocol flows

The baseline coherence protocol operates with two stable states-valid and invalid-allowing it to remain lightweight and efficient. In our proposed coherence directory design, each entry represents the validity of an entire address range instead of tracking individual tag addresses and associated sharers. This enables the state transitions to be managed at a coarser granularity during directory evictions. Additionally, REC supports fine-grained control over write requests by tracking specific offsets within these address ranges, avoiding the need to evict entire entries. Fig. 10 highlights the architecture of REC and how it differently handles the received requests with the baseline. REC does not require additional coherence states but instead modifies the transitions triggered under specific conditions.

Remote reads: When the GPU receives the read request from the remote GPU, the cache controller extracts the base and offset from the tag address (🅐). The controller then looks up the coherence directory for an entry with the matching base address (🅑). If a valid entry is found, the position bit corresponding to the offset calculated using the formula in Section 4.1 and the associated sharer bit are set (🅒). For example, the position bit is $340_{16}/64 \times 4 = 52$ representing the 14th cache line within the specified 1 kB range. The sharer bit is determined by the source GPU index (e.g., GPU1). Therefore, bit 52 and 53 are set to 1. It can happen that the position bit is already set; nevertheless, the controller still performs a bitwise OR on the bits at the corresponding positions. Since the entry already exists in the directory, it remains valid. Otherwise, if no valid entry is found, a new entry is created with the base address, and the position and sharer bits are set. With the insertion of a new entry, the state transitions from invalid to valid.
Local writes: When the write request is performed locally (🅓), the cache controller must determine whether it needs to send out invalidation requests to the sharers that hold the copy of data. For this, the controller again looks up the directory with the calculated base address and offset (🅔). If an entry is found and the offset is valid (i.e., the position bit is set), the invalidation request is generated and propagated to the recorded sharers immediately (🅕). The state transition is handled differently based on two conditions. First, when another offset is tracked under the common address range, the directory entry should remain valid. Thus, the controller clears only the position and sharer bits for the specific offset of the target address. For example, in Fig. 10, the directory entry has another offset (atp = 56) recorded under the same base address. Once the invalidation request is sent out to GPU1, the controller only clears bits 0 and 1. If the cleared bits are the last ones, the entire directory entry transitions to an invalid state to make room for new entries.
Remote writes: For the remote write request, the cache controller begins the same directory lookup process by calculating the base and offset from the tag (🅖). In our target multi-GPU system, the source GPU also performs writes to the copy of data in its local L2 cache (discussed in Section 2.2). Therefore, the controller handles remote write requests differently from local writes. When an entry already exists in the directory (i.e., hits), there may be two circumstances: (1) the target offset is invalid but the entry has other valid offsets and (2) the target offset is already valid and one or more sharers are being tracked. If the target offset is invalid, the controller simply adds the offset and the sharer to the entry in the same way it handles remote reads. If the offset is valid, the controller adds the source GPU to the sharer list by setting its corresponding bit and clearing other sharer bits (🅗), then sends invalidation requests to all other sharers (🅘). In Fig. 10, the entry and the target offset (atp = 56) both are already recorded. The controller, thus, additionally sets bit 58 to add GPU2 as a sharer while clearing the bit 59 and sends the invalidation request to GPU3. In either cases, the directory entry remains valid. When the directory misses, the cache controller allocates a new entry to record the base, offset, and sharer from the write request. Then, the entry state transitions to valid.
Directory entry eviction/replacement: When the coherence directory becomes full, it needs to replace an entry with the newly inserted one. The baseline coherence directory uses a FIFO replacement policy. However, for workloads that exhibit irregular memory access patterns, capturing locality becomes a challenge. To address this, REC adopts the replacement policy, similar to LRU, to better retain entries that are more likely to be accessed again. As the cache controller receives the remote read request and does find an entry with the matching base address (🅙), it determines an entry for replacement (🅚). The evicting entry is then replaced with the new entry from the incoming request (🅛). Meanwhile, the controller retrieves the base address, every merged offset from the evicting entry and reconstructs the original tag addresses. Invalidation requests are propagated to every recorded sharer associated with each tag address (🅜). Lastly, the entry transitions to an invalid state.

4.3. Discussion

Overheads: In our design, the coherence directory consists of 8K entries, with each entry covering a 1 kB range of addresses. Each entry comprises a 38-bit base address field, a 64-bit vector for offsets and sharers, and a valid bit (detailed in Table 1). Thus, the total directory size is $8192 \times 103/8/1024 = 103$ kB. We also estimate the area and power overhead of the coherence directory in REC, using CACTI 7.0 [25]. The results show that the directory is 3.94% area and has 3.28% power consumption compared to GPU L2 cache. REC requires no additional hardware extensions for managing the coherence directory. The existing cache controller handles operations such as base address calculation and bitwise manipulation efficiently.
Comparison to prior work: As discussed in Section 2.3, HMG [11] designs each coherence directory entry to track four cache lines at a coarse granularity. We empirically show, in Section 3.3, that GPUs require a directory size up to 12× the baseline to eliminate unnecessary cache line invalidations. Since REC coalesces up to 16 consecutive cache line addresses per entry, REC can track a significantly larger number of cache lines compared to the prior work. Moreover, REC precisely tracks each address by storing the offset and sharer information. Thus, REC fully support fine-grained management of cache lines under write operations.
Scalability: REC requires modifications to its design in large-scale systems, specifically to the sharer bit field. For an 8-GPU system, REC requires $(8\text{-}1)\times 16 = 112$ bits to record sharers in each entry. Then, the size of each entry becomes $112 + 38 + 16 + 1 = 167$ bits, which is approximately three times the baseline size, where each entry costs 56 bits, including a 4-bit increase for sharers. Similarly, for a 16-GPU system, REC requires 295 bits per entry, roughly five times the baseline size. However, as observed in Section 3.3, an ideal GPU requires up to 12 times the baseline directory size even in a 4-GPU system, implying that simply increasing the baseline directory size is insufficient to meet scalability demands.

5. Evaluation

5.1. Methodology

We use MGPUSim [20], a cycle-accurate multi-GPU simulator, to model baseline and REC architecture with four AMD GPUs connected using inter-GPU links of 300 GB/s bandwidth [26]. The configuration of the modeled GPU architecture is detailed in Table 2. Each GPU includes L1 scalar and instruction caches shared within each SA, while the L1 vector cache is private to each CU, and the L2 cache is shared across the GPU. We extend remote data caching to the L2 caches, allowing data from any GPU in the system to be cached in the L2 cache of any other GPU. Since MGPUSim does not include a support of hardware cache coherence, we extend the simulator by implementing a coherence directory managed by the L2 cache controller. The coherence directory is implemented with a set-associative structure to reduce lookup latency. Since the baseline coherence directory is decoupled from the caches, its way associativity as well as the size can be scaled independently. In our evaluation, the coherence directory is designed with an 8-way set-associative structure to reduce conflict misses, containing 8K entries in both the baseline and REC architectures. Upon receiving remote read requests, the cache controller updates the coherence directory with recording the addresses and the associated sharers. Once capacity of the directory is reached, the cache controller evicts an entry and sends out invalidation requests to the recorded sharers. For receiving write requests, the controller looks up the directory to find whether data with matching addresses are shared by remote GPUs. If the matching entries are found, invalidation requests are propagated to the sharers except the source GPU. Additionally, since L2 caches are managed by coherence directories, acquire operations do not perform invalidations on L2 caches, but release operations flush the L2 caches. We use workloads from a diverse set of benchmark suites, including AMDAPPSDK [23], Heteromark [22], Polybench [21], SHOC [24]. Table 3 lists the workloads with their memory footprints.

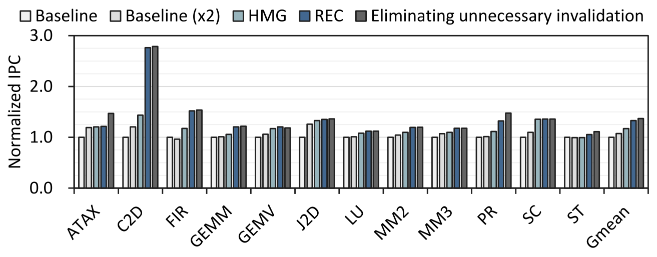

Fig. 11. Performance comparison of the baseline with double-sized coherence directory, HMG [11], REC, and an idealized system with zero unnecessary invalidations. Performance is normalized to the baseline with 8K-entry coherence directory.

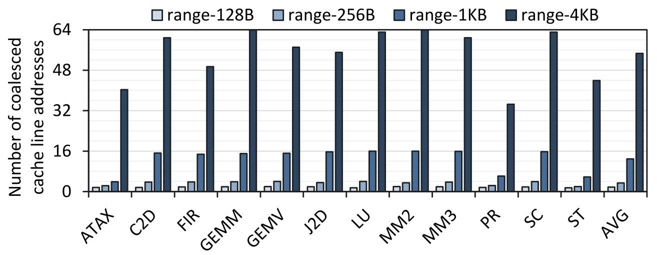

Fig. 12. Number of coalesced cache line addresses at directory entry eviction under REC with varying addressable ranges. REC in this work coalesces with 1 kB addressable range.

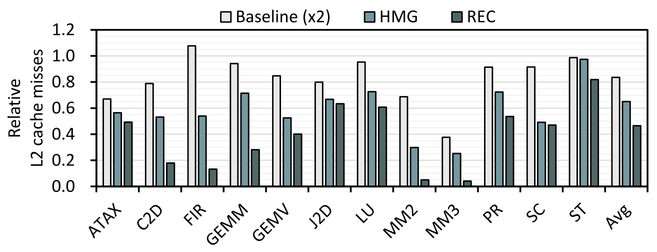

Fig. 13. Total number of L2 cache misses in the baseline with double-sized coherence directory, HMG [11], and REC relative to the baseline.

5.2. *Performance analysis*

Fig. 11 shows the performance of the baseline with coherence directory double in size, HMG [11], REC, and an ideal multi-GPU system with zero unnecessary invalidations relative to the baseline. First, we include the performance of baseline with double in coherence directory size to compare REC with the same storage cost. The result shows that the baseline with double the size of directory achieves average speedup of 7.3%. The baseline coherence directory tracks each remote access individually, on a per-entry basis. As discussed in Section 3.3, doubling the size of coherence directory does not mitigate the unnecessary cache line invalidations for applications with significant directory evictions. Also, results show that HMG and REC achieve average speedup of 16.7% and 32.7% across the evaluated workloads. We observe that REC outperforms the prior scheme for two reasons. First, REC delays directory evictions by allowing each entry to record more cache line addresses for a wider range. Since HMG uses each directory entry to track four cache lines, an entire coherence directory can track cache lines up to 4× the baseline. On the other hand, the directory in REC can record up to 16× the number of entries. Second, REC manages write operations to shared cache lines at a fine granularity by searching the directory with exact addresses and sharers, propagating invalidations only when necessary. Since each directory entry of HMG stores only a single address and sharer ID field that cover for four cache lines, writes to any of these cache lines trigger invalidation requests to every cache line and recorded sharer which leads them to be false positives. In contrast, REC does not allow any false positives and performs invalidations only to the modified cache lines and the associated sharers. As a result, REC reduces unnecessary invalidations on cache lines that are actively being accessed by the requesting GPUs, minimizing redundant remote memory accesses. To investigate the effectiveness of REC under different addressable ranges listed in Table 1, we also measure the number of coalesced cache line addresses when an entry is evicted and plot in Fig. 12. We observe that the directory entries capture an average of 1.8, 3.4, 12.9, and 54.7 addresses until eviction under REC with 128B, 256B, 1 kB, and 4 kB coalesceable ranges. Specifically, REC captures more than 14 addresses before directory eviction for applications with strong spatial locality.

Fig. 12 also illustrates the characteristics of limited locality for certain workloads where REC benefits less. In ATAX, PR, and ST, REC coalesces 3.9, 6.1, and 5.8 addresses, respectively. This is because the applications exhibit locality challenging to be captured due to their irregular memory access patterns that span across a wide range of addresses. To delay the eviction of entries in irregular workloads, we design our proposed coherence directory with an LRU-like replacement policy (discussed in Section 4.2). Another interesting observation is that the performance improvement of GEMV with REC is higher than the improvement seen when eliminating unnecessary invalidations. Our approach delays invalidations, but still performs them when the directories become full. During cache line replacement, the controller prioritizes invalid cache lines before applying the LRU policy. As a result, this delays the replacement of useful cache lines, thereby improving cache efficiency.

L2 cache misses: The performance improvement of REC is largely attributed to the reduction in cache misses caused by unnecessary invalidations from frequent evictions in the coherence directory of home GPUs. Fig. 13 shows the total number of L2 cache misses in the baseline with double-sized directory, HMG, and REC relative to the baseline. Cold misses are excluded from the results. We observe that REC reduces L2 cache misses by 53.5%. In contrast, the baseline with double-sized directory and HMG experience 1.79× and 1.40× higher number of cache misses than REC since neither approach is insufficient to delay evict-initiated cache line invalidations. The result is closely related to the reduction in remote access latency, as the corresponding misses are forwarded to the remote GPUs. Addressing the remote GPU access bottleneck is performance-critical in multi-GPU systems.

Unnecessary invalidations: In the baseline, invalidation requests propagated from frequent directory evictions in the home GPU lead to a higher chances of finding the corresponding cache lines still valid in the sharer-side L2 caches. This results in premature invalidations of cache lines that are actively in use, exacerbating the cache miss rate. In REC, the invalidation requests generated by directory eviction reduce the chances of invalidating valid cache lines. Fig. 14 shows that the number of unnecessary invalidations performed in remote L2 caches (i.e., where they are hits) is reduced by 84.4%. Since REC significantly delays evict-initiated invalidation requests, many cache lines have already been evicted from the caches by the time these requests are issued.

Inter-GPU transactions: The reduction in unnecessary invalidations enhances the utilization of data within the sharer GPUs and minimizes redundant accesses over inter-GPU links. Fig. 14 shows the total number of inter-GPU transactions compared to the baseline. As illustrated, REC reduces inter-GPU transactions by an average of 34.9%. The reduced inter-GPU transactions directly contributes to the overall performance improvement in multi-GPU systems.

Bandwidth impact: Fig. 15 shows the total inter-GPU bandwidth costs of invalidation requests. As presented in Section 3.2, a large fraction of invalidation requests are propagated due to frequent directory evictions. Since REC delays invalidation requests from directory evictions by allowing each entry to coalesce multiple tag addresses, the

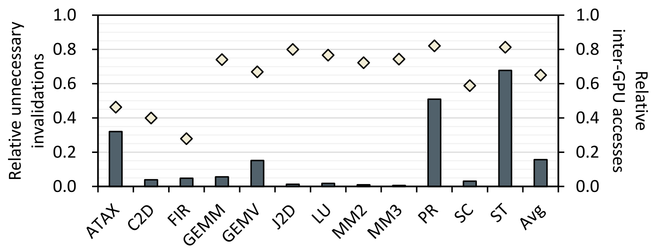

Fig. 14. Total number of unnecessary invalidations (bars) and inter-GPU transactions (plots) relative to the baseline.

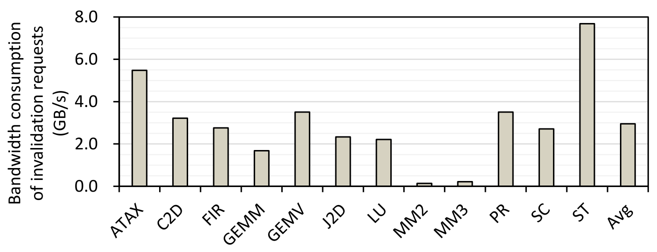

Fig. 15. Total bandwidth consumption of invalidation requests.

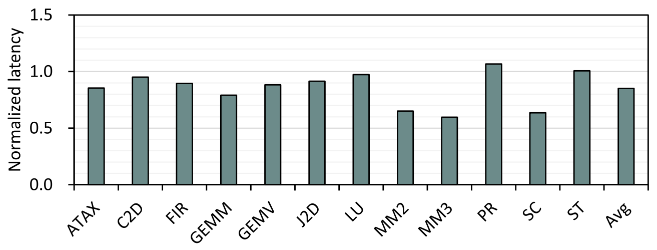

Fig. 16. L2 cache lookup latency.

bandwidth in most of the workloads becomes only a few gigabytes per second.

Cache lookup latency: Fig. 16 illustrates the average L2 cache lookup latency of REC normalized to the baseline. The results show that the lookup latency reduces by 14.8% compared to the baseline. REC affects the average lookup latency as evict-initiated invalidation requests are propagated in burst. However, since REC significantly delays directory eviction by coalescing multiple tag addresses, the overall latency decreases for most of the evaluated workloads.

5.3. Sensitivity analysis

Coalescing range: One important design decision in optimizing REC is determining the range over which to coalesce when remote read requests are received. As discussed in Section 4.1, the trade-off exists between the range an entry coalesces and the number of bits required: the larger the range, the more bits are needed to store the remote GPU access information. Fig. 17(a) shows that the performance of REC improves as the coalescing range increases, with performance gains beginning to saturate at 1 kB. For our applications, a 1 kB range is sufficient to capture the majority of memory access locality within the workloads. Since coalescing beyond 4 kB incurs excessive overhead in terms of bits required per entry (with 4 kB already requiring nearly 6× the baseline size), the potential performance improvement may not be substantial to offset the additional cost. Therefore, we choose a 1 kB range for our implementation.

Entry size: In our evaluation, we use a directory size of 8K entries to match the baseline coherence directory. Fig. 17(b) shows the performance REC with varying entry sizes, ranging from 2K to 32K. On average, REC outperforms the baseline, even with reduced entry sizes compared to the baseline system with 8K-entry coherence directory. This is because the coverage of each coherence directory in REC can increase by up to 16× when locality is fully utilized. Although applications with limited locality show performance improvements as the directory size increases, these gains are relatively modest when considered against the additional hardware costs.

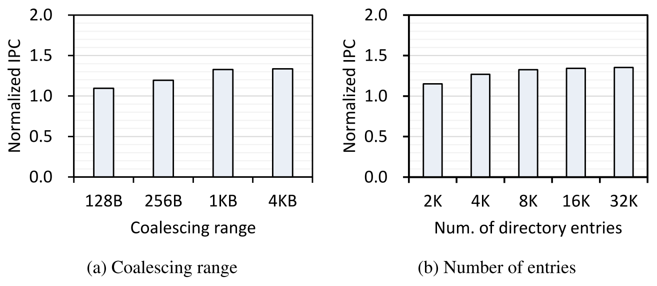

Fig. 17. Performance of REC under varying (a) coalescing address ranges and (b) number of directory entries. Results are shown relative to the baseline with an 8K-entry coherence directory.

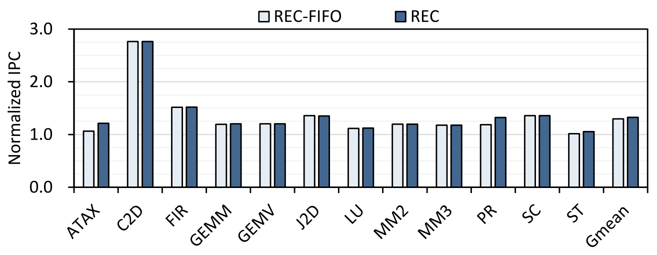

Fig. 18. Performance comparison of REC using FIFO and LRU replacement policies. Performance is normalized to the baseline coherence directory with FIFO policy.

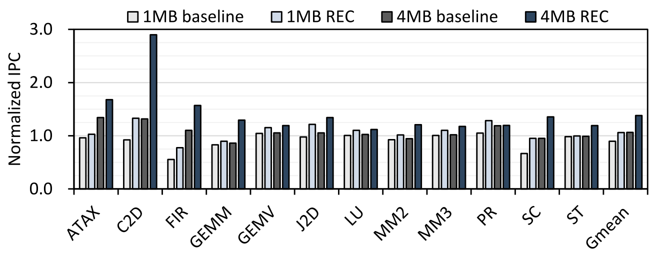

Fig. 19. Performance impact of different L2 cache sizes in the baseline and REC. Performance is normalized to the baseline with 2 MB L2 cache.

FIFO replacement: Fig. 18 represents the performance of REC with a FIFO replacement policy. Our evaluation shows that the choice of replacement policy has a relatively small impact on the overall performance. For the workloads with regular and more predictable memory access patterns, using the FIFO replacement policy is already effective in coalescing sufficient number of addresses under the target ranges (shown in Fig. 12). However, for some applications, such as ATAX, PR, and ST, performance is lower with FIFO compared to REC due to their limited locality patterns. These applications, therefore, benefit from using an LRU-like replacement policy.

L2 cache size: The performance impact of different L2 cache sizes is shown in Fig. 19. The results are normalized to the baseline with a 2 MB L2 cache. The benefits from increasing L2 cache capacity are limited by the baseline coherence directory. In contrast, the performance of REC improves as L2 cache size increases, demonstrating its ability to leverage larger caches effectively. Another observation is that performance improvement with smaller L2 capacity is less significant compared to larger L2 caches. This is because the coverage of the

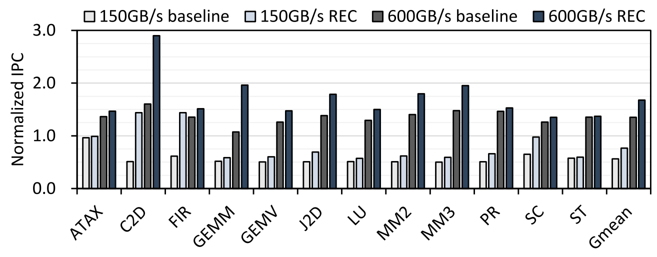

Fig. 20. Performance impact of different inter-GPU bandwidth in the baseline and REC. Performance is normalized to the baseline with 300 GB/s inter-GPU bandwidth.

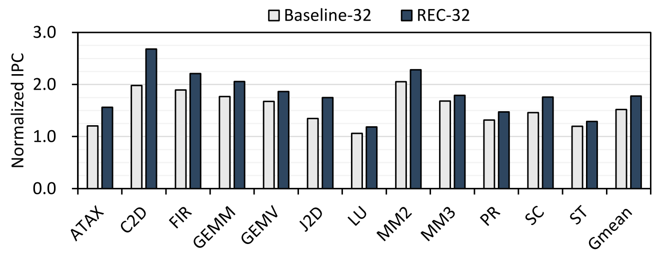

Fig. 21. Performance of REC with different number of SAs normalized to the baseline with 16 SAs.

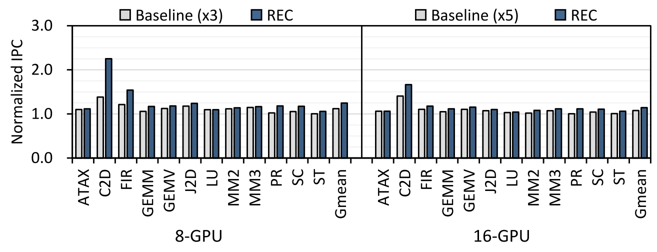

Fig. 22. Performance comparison of REC and the baseline with equal storage cost under different number of GPUs. Performance is normalized to the baseline with 8K entries.

baseline coherence directory relatively increases as the L2 cache size decreases. To further explore the performance sensitivity to different L2 cache sizes, we evaluate REC in systems with L2 cache sizes of 0.5 MB and 8 MB. We find that REC achieves an average performance improvement of 6.3% and 26.7% compared to the baseline with 0.5 MB and 8 MB L2 caches, respectively. Additionally, the performance trend of REC decreases as the L2 cache size increases since the effectiveness of REC also reduces larger caches. Nevertheless, the results emphasize the importance of coherence protocol in improving cache efficiency.

Inter-GPU bandwidth: The bandwidth of inter-GPU links is a critical factor in scaling multi-GPU performance. Fig. 20 shows the performance of the baseline and REC under different inter-GPU bandwidths, relative to the 300 GB/s baseline. The results demonstrate that REC outperforms the baseline, even in applications where performance begins to saturate with increased bandwidth.

Number of SAs: We also evaluate REC with increasing the number of SAs as shown in Fig. 21. The performance improvement of REC decreases compared to the system with 16 SAs since the increased number of SAs improves thread-level parallelism of GPUs. However, the system with a larger number of SAs also elevates the intensity of data sharing thus, increases the frequency of coherence directory evictions. As a result, REC outperforms the baseline with 16 SAs by 17.1%.

Number of GPUs: We evaluate REC in 8-GPU and 16-GPU systems, as shown in Fig. 22. To ensure a fair comparison, we do not change the workload sizes. The results show that REC provides performance improvements of 24.7% and 14.7% over the baseline in 8-GPU and 16-GPU systems, respectively. We observe that the performance improvement decreases as the number of GPUs increases. This is because, with more GPUs, the application dataset is more distributed, and the amount of data allocated to each GPU's memory decreases, resulting in reduced pressure on each coherence directory for tracking shared copies. Additionally, we compare REC with the baseline configured with different directory sizes to match equal storage costs (discussed in Section 4.3). We observe that REC achieves performance improvements of 2.04× and 1.83× over the baseline with directory sizes increased by 3× and 5×, respectively. The results confirm that simply increasing directory sizes is not an efficient approach, even in large-scale multi-GPU systems.

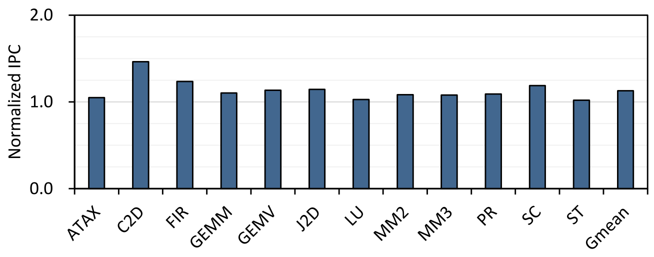

Fig. 23. Performance of REC in different GPU architecture.

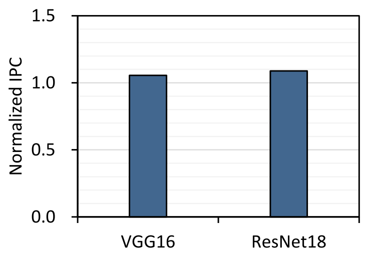

Fig. 24. Performance of REC with DNN applications.

5.4. *REC with Different GPU Architecture*

We extend the evaluation of REC to include a different GPU architecture by adapting the simulation environment to a more recent NVIDIA-styled GPU [27]. This involves increasing the number of computation and memory resources compared to the AMD GPU setup. Specifically, we change the GPU configuration to include 128 CUs, each with a 128 kB L1V cache. The L2 cache size is increased to 72 MB with the cache line size adjusted to 128B. With the increased cache line size, we configure the addressable range of REC to 2 kB, allowing for coalescing up to the same number of tag addresses. We also scale the input sizes of the workloads until the simulations remain feasible. The performance results, in Fig. 23, show that REC achieves a 12.9% performance improvement over the baseline. This indicates that our proposed REC also benefits the NVIDIA-like GPU architecture.

5.5. *Effectiveness of REC on DNN applications*

We evaluate the performance improvement of REC in training two DNN models, VGG16 and ResNet18, using Tiny-Imagenet-200 dataset [28]. As shown in Fig. 24, REC outperforms the baseline for training VGG16 and ResNet18 by 5.6% and 8.9%, respectively. The results imply that REC also has benefits in multi-GPU training on DNN workloads. Additionally, GPUs have recently gained significant attention for training large language models (LLM). The computation of LLM training comprises multiple decoder blocks with each primarily having series of matrix and vector operations [29]. In our evaluation, we observe that REC improves multi-GPU performance by 20.2% and 20.4% on GEMM and GEMV workloads, respectively. Considering real-world LLM training, the memory requirements can become significant with large parameters which can pressure memory systems and lead to under-utilization of computation resources [29]. Since REC improves the cache efficiency in multi-GPU systems, we expect a higher performance potential from REC in real-world LLM training.

6. Related work

Several prior works have proposed GPU memory consistency and cache coherence mechanisms optimized for general-purpose domains [13–15,19,30–32]. GPU-VI [19] reduces stalls at the cache controller by employing write-through, write-no-allocate L1 caches and treating loads to the pending writes as misses. To maintain write atomicity, GPU-VI adds transient states and state transitions and requires invalidation acknowledgments before write completion. REC is implemented based on the relaxed memory models commonly adopted in recent GPU architectures, which do not require acknowledgments to be sent or received over long-latency inter-GPU links. HMG [11] proposes a lightweight directory protocol by addressing up-to-date memory consistency and coherence requirements. HMG integrates separate layers for managing inter-GPM and inter-GPU level coherence, reducing network traffic and complexity in deeply hierarchical multi-GPU systems. REC primarily addresses the increased cache misses to remotely fetched data caused by frequent invalidations. Additionally, REC can be extended to support hierarchical multi-GPU systems posed by HMG without significant hardware modifications.

Other efforts aim to design efficient cache coherence protocols for other processor domains. Wang et al. [33] suggested a method to efficiently support dynamic task parallelism on heterogeneous cache coherent systems. Zuckerman et al. [34] proposed Cohmeleon that orchestrates the coherence in accelerators in heterogeneous system-on-chip designs. HieraGen [35] and HeteroGen [36] are automated tools for generating hierarchical and heterogeneous cache coherence protocols, respectively, for generic processor designs. Li et al. [37] proposed methodologies to determine the minimum number of virtual networks for cache coherence protocols that can avoid deadlocks. However, these studies do not address the challenges of redundant invalidations in the cache coherence mechanisms of multi-GPU systems.

Significant research has addressed the NUMA effect in multi-GPU systems by proposing efficient page placement and migration strategies [5,6,38], data transfer and replication methods [4,7,8,10,39,40], and address translation schemes [41–43]. In particular, several works have focused on improving the management of shared data within the local memory hierarchy. NUMA-aware cache partitioning [3] dynamically allocates cache space to accommodate data from both local and remote memory by monitoring inter-GPU and local DRAM bandwidths. The authors also extend software coherence with bulk invalidations to L2 caches and evaluate the overhead associated with unnecessary invalidations. SAC [12] proposes reconfigurable last-level caches (LLC) that can be utilized as either memory-side or SM-side, depending on predicted application behavior in terms of effective LLC bandwidth. SAC evaluates the performance of both software and hardware extensions for LLC coherence. In contrast, REC specifically targets the issue of unnecessary invalidations under hardware coherence, which can undermine the efficiency of remote data caching. It introduces a new directory structure, carefully examining the trade-off between performance and storage overhead.

Recent studies on multi-GPU and multi-node GPU systems also address challenges in various domains. Researchers proposed methods to accelerate deep learning applications [44], graph neural networks [45], and graphics rendering applications [46] in multi-GPU systems. Na et al. [47] addressed security challenges in inter-GPU communications under unified virtual memory framework. Barre Chord [48] leverages page allocation schemes in multi-chip-module GPUs to reduce address translation overheads, and [47]. Villa et al. [49] studied designing trustworthy system-level simulation methodologies for single- and multi-GPU systems. Lastly, NGS [50] enables multiple nodes in a data center network to share the compute resources of GPUs on top of a virtualization technique.

7. Conclusion

In this paper, we propose REC to improve the efficiency of cache coherence in multi-GPU systems. Our analysis shows that the limited capacity of coherence directories in fine-grained hardware protocols frequently leads to evictions and unnecessary invalidations of shared data. As a result, the increase in cache misses exacerbates NUMA overhead, leading to significant performance degradation in multi-GPU systems. To address this challenge, REC leverages memory access locality to coalesce multiple tag addresses within common address ranges, effectively increasing the coverage of coherence directories without incurring significant hardware overhead. Additionally, REC maintains write-initiated invalidations at a fine granularity to ensure precise and flexible coherence across GPUs. Experiments show that REC reduces L2 cache misses by 53.5% and improves overall system performance by 32.7%.

CRediT authorship contribution statement

Gun Ko: Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Jiwon Lee:** Formal analysis, Conceptualization. **Hongju Kal:** Validation, Conceptualization. **Hyunwuk Lee:** Visualization, Validation. **Won Woo Ro:** Supervision, Project administration, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2024-00402898, Simulation-based High-speed/High-Accuracy Data Center Workload/System Analysis Platform)

Data availability

The authors are unable or have chosen not to specify which data has been used.

References

[1] NVIDIA, NVIDIA DGX-2, 2018, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-2/dgx-2-print-datasheet-738070-nvidia-a4-web-uk.pdf.

[2] NVIDIA, NVIDIA DGX A100 system architecture, 2020, https://download.boston.co.uk/downloads/3/8/6/386750a7-52cd-4872-95e4-7196ab92b51c/DGX%20A100%20System%20Architecture%20Whitepaper.pdf.

[3] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, D. Nellans, Beyond the socket: NUMA-aware GPUs, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2017, pp. 123–135.

[4] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, O. Villa, Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2018, pp. 339–351.

[5] T. Baruah, Y. Sun, A.T. Dinçer, S.A. Mojumder, J.L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, D. Kaeli, Griffin: Hardware-software support for efficient page migration in multi-GPU systems, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2020, pp. 596–609.

[6] M. Khairy, V. Nikiforov, D. Nellans, T.G. Rogers, Locality-centric data and thread-block management for massive GPUs, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2020, pp. 1022–1036.

[7] H. Muthukrishnan, D. Lustig, D. Nellans, T. Wenisch, GPS: A global publish-subscribe model for multi-GPU memory management, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 46–58.

[8] L. Belayneh, H. Ye, K.-Y. Chen, D. Blaauw, T. Mudge, R. Dreslinski, N. Talati, Locality-aware optimizations for improving remote memory latency in multi-GPU systems, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2022, pp. 304–316.

[9] S.B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, K. Barker, Spy in the GPU-box: Covert and side channel attacks on multi-GPU systems, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2023, pp. 633–645.

[10] H. Muthukrishnan, D. Lustig, O. Villa, T. Wenisch, D. Nellans, FinePack: Transparently improving the efficiency of fine-grained transfers in multi-GPU systems, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2023, pp. 516–529.

[11] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, D. Nellans, HMG: Extending cache coherence protocols across modern hierarchical multi-GPU systems, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2020, pp. 582–595.

[12] S. Zhang, M. Naderan-Tahan, M. Jahre, L. Eeckhout, SAC: Sharing-aware caching in multi-chip GPUs, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2023, pp. 605–617.

[13] B.A. Hechtman, S. Che, D.R. Hower, Y. Tian, B.M. Beckmann, M.D. Hill, S.K. Reinhardt, D.A. Wood, QuickRelease: A throughput-oriented approach to release consistency on GPUs, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2014, pp. 189–200.

[14] M.D. Sinclair, J. Alsop, S.V. Adve, Efficient GPU synchronization without scopes: Saying no to complex consistency models, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2015, pp. 647–659.

[15] J. Alsop, M.S. Orr, B.M. Beckmann, D.A. Wood, Lazy release consistency for GPUs, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2016, pp. 1–13.

[16] NVIDIA, NVIDIA TESLA V100 GPU architecture, 2017, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[17] NVIDIA, NVIDIA A100 tensor core GPU architecture, 2020, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[18] NVIDIA, NVIDIA NVLink high-speed GPU interconnect, 2024, https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/.

[19] I. Singh, A. Shriraman, W.W.L. Fung, M. O'Connor, T.M. Aamodt, Cache coherence for GPU architectures, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2013, pp. 578–590.

[20] Y. Sun, T. Baruah, S.A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A.K. Ziabari, Z. Chen, R. Ubal, J.L. Abellán, J. Kim, A. Joshi, D. Kaeli, MGPUSim: Enabling multi-GPU performance modeling and optimization, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2019, pp. 197–209.

[21] T. Yuki, L.-N. Pouchet, Polybench 4.0, 2015.

[22] Y. Sun, X. Gong, A.K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, D. Kaeli, Hetero-mark, a benchmark suite for CPU-GPU collaborative computing, in: Proceedings of IEEE International Symposium on Workload Characterization, 2016, pp. 1–10.

[23] AMD, AMD app SDK OpenCL optimization guide, 2015.

[24] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter, The Scalable Heterogeneous Computing (SHOC) benchmark suite, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 63–74.

[25] R. Balasubramonian, A.B. Kahng, N. Muralimanohar, A. Shafiee, V. Srinivas, CACTI 7: New tools for interconnect exploration in innovative off-chip memories, ACM Trans. Archit. Code Optim. 14 (2) (2017) 14:1–25.

[26] NVIDIA, NVIDIA DGX-1 with tesla V100 system architecture, 2017, pp. 1–43.

[27] NVIDIA, NVIDIA ADA GPU architecture, 2023, https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf.

[28] Y. Le, X. Yang, Tiny ImageNet visual recognition challenge, 2015, https://http://vision.stanford.edu/teaching/cs231n/reports/2015/pdfs/yle_project.pdf.

[29] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, J. Park, NeuPIMs: NPU-PIM heterogeneous acceleration for batched LLM inferencing, in: Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2024, pp. 722–737.

[30] K. Koukos, A. Ros, E. Hagersten, S. Kaxiras, Building heterogeneous Unified Virtual Memories (UVMs) without the overhead, ACM Trans. Archit. Code Optim. 13 (1) (2016).

[31] X. Ren, M. Lis, Efficient sequential consistency in GPUs via relativistic cache coherence, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2017, pp. 625–636.

[32] S. Puthoor, M.H. Lipasti, Turn-based spatiotemporal coherence for GPUs, ACM Trans. Archit. Code Optim. 20 (3) (2023).

[33] M. Wang, T. Ta, L. Cheng, C. Batten, Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2020, pp. 173–186.

[34] J. Zuckerman, D. Giri, J. Kwon, P. Mantovani, L.P. Carloni, Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous SoCs, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 350–365.

[35] N. Oswald, V. Nagarajan, D.J. Sorin, HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2020, pp. 888–899.

[36] N. Oswald, V. Nagarajan, D.J. Sorin, V. Gavrielatos, T. Olausson, R. Carr, HeteroGen: Automatic synthesis of heterogeneous cache coherence protocols, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2022, pp. 756–771.

[37] W. Li, A.G.U. of Amsterdam, N. Oswald, V. Nagarajan, D.J. Sorin, Determining the minimum number of virtual networks for different coherence protocols, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2024, pp. 182–197.

[38] Y. Wang, B. Li, A. Jaleel, J. Yang, X. Tang, GRIT: Enhancing multi-GPU performance with fine-grained dynamic page placement, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2024, pp. 1080–1094.

[39] M.K. Tavana, Y. Sun, N.B. Agostini, D. Kaeli, Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-GPU systems, in: Proceedings of IEEE International Parallel and Distributed Processing Symposium, 2019, pp. 664–674.

[40] H. Muthukrishnan, D. Nellans, D. Lustig, J.A. Fessler, T.F. Wenisch, Efficient multi-GPU shared memory via automatic optimization of fine-grained transfers, in: Proceedings of the ACM/IEEE International Symposium on Computer Architecture, 2021, pp. 139–152.

[41] B. Li, J. Yin, Y. Zhang, X. Tang, Improving address translation in multi-GPUs via sharing and spilling aware TLB design, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 1154–1168.

[42] B. Li, J. Yin, A. Holey, Y. Zhang, J. Yang, X. Tang, Trans-FW: Short circuiting page table walk in multi-GPU systems via remote forwarding, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2023, pp. 456–470.

[43] B. Li, Y. Guo, Y. Wang, A. Jaleel, J. Yang, X. Tang, IDYLL: Enhancing page translation in multi-GPUs via light weight PTE invalidations, in: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2015, pp. 1163–1177.

[44] E. Choukse, M.B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, Buddy compression: Enabling larger memory for deep learning and HPC workloads on GPUs, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2020, pp. 926–939.

[45] Y. Tan, Z. Bai, D. Liu, Z. Zeng, Y. Gan, A. Ren, X. Chen, K. Zhong, BGS: Accelerate GNN training on multiple GPUs, J. Syst. Archit. 153 (2024) 103162.

[46] X. Ren, M. Lis, CHOPIN: Scalable graphics rendering in multi-GPU systems via parallel image composition, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2021, pp. 709–722.

[47] S. Na, J. Kim, S. Lee, J. Huh, Supporting secure multi-GPU computing with dynamic and batched metadata management, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2024, pp. 204–217.

[48] Y. Feng, S. Na, H. Kim, H. Jeon, Barre chord: Efficient virtual memory translation for multi-chip-module GPUs, in: Proceedings of ACM/IEEE International Symposium on Computer Architecture, 2024, pp. 834–847.

[49] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, Need for speed: Experiences building a trustworthy system-level GPU simulator, in: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2021, pp. 868–880.

[50] J. Prades, C. Reaño, F. Silla, NGS: A network GPGPU system for orchestrating remote and virtual accelerators, J. Syst. Archit. 151 (2024) 103138.

Gun Ko received the B.S. degree in electrical engineering from Pennsylvania State University in 2017. He is currently pursuing the Ph.D. degree with the Embedded Systems and Computer Architecture Laboratory, School of Electrical and Electronic Engineering, Yonsei University, Seoul, South Korea. His current research interests include GPU memory systems, multi-GPU systems, and virtual memory.

Jiwon Lee received the B.S. and Ph.D. degrees in electrical and electronic engineering from Yonsei University, Seoul, South Korea, in 2018 and 2024, respectively. He currently works in the memory division at Samsung Electronics. His research interests include virtual memory, GPU memory systems, and storage systems.

Hongju Kal received the B.S. degree from Seoul National University of Science and Technology and Ph.D. degree from Yonsei University in school of electric and electronic engineering, Seoul, South Korea in 2018 and 2024, respectively. He currently works in the memory division at Samsung Electronics. His current research interests include memory architectures, memory hierarchies, near memory processing, and neural network accelerators.

Hyunwuk Lee received his B.S. and Ph.D. degrees in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2018 and 2024, respectively. He currently works in the memory division at Samsung Electronics. His research interests include neural network accelerators and GPU systems.

Won Woo Ro received the B.S. degree in electrical engineering from Yonsei University, Seoul, South Korea, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from the University of Southern California, in 1999 and 2004, respectively. He worked as a Research Scientist with the Electrical Engineering and Computer Science Department, University of California, Irvine. He currently works as a Professor with the School of Electrical and Electronic Engineering, Yonsei University. Prior to joining Yonsei University, he worked as an Assistant Professor with the Department of Electrical and Computer Engineering, California State University, Northridge. His industry experience includes a college internship with Apple Computer, Inc., and a contract software engineer with ARM, Inc. His current research interests include high-performance microprocessor design, GPU microarchitectures, neural network accelerators, and memory hierarchy design.