NUMA-Aware Queue Scheduler for Multi-Chiplet GPUs

Leveraging the Command Processor for Queue Scheduling in multi-chiplet GPUs

by
Neeraj Surawar

A project submitted in partial fulfillment
of the requirements for the degree of
Masters of Science
Computer Sciences

University of Wisconsin-Madison
December 2024

Acknowledgments

I would like to extend my heartfelt gratitude to my research advisor, Professor Matt Sinclair. His constant support, valuable mentorship, and encouragement have been instrumental throughout my research journey. Prof. Sinclair is one of the most diligent and hard-working individuals I have ever encountered, and his work ethic has inspired me to dedicate myself more fully to my research. His guidance has been truly inspirational and has tremendously impacted how I approach and tackle difficult research challenges. Working on exciting projects under his supervision has been an enriching experience, and I truly appreciate the opportunity to learn and grow under his mentorship. I also want to thank all the members of the HAL Research Group sincerely. Their insightful discussions, collaborative spirit, and readiness to help have enriched my learning experience and made the research process both rewarding and enjoyable. I am grateful to be part of such an exceptional research team.

I am also grateful to the Computer Sciences Department at the University of Wisconsin-Madison for fostering an environment rich with opportunities.

Last but not least, I would like to thank my parents and sister for believing in me, encouraging me, and always supporting me throughout my academic career.

On Wisconsin!

Abstract

Chiplet-based architectures have recently emerged as a technique to improve yields and enable continued performance scaling. However, the increased modularity and scalability they offer also requires rethinking system design. Compared to monolithic designs, chiplet-based architectures face challenges around how computation is scheduled and how data movement is coordinated across chiplets. These challenges introduce additional Non-Uniform Memory Access (NUMA) complexities in multi-chiplet systems that can impact performance. Consequently, exploiting locality is a significant bottleneck in multi-chiplet systems. Although multi-chiplet CPUs overcome this inefficiency through complex coherence protocols or OS support, accelerators (e.g., GPUs) utilize relatively lightweight coherence and OS support. Thus, inter-chiplet NUMA effects affect them more – especially at phase boundaries where accelerators often utilize heavyweight operations to ensure correctness. In recognition of these challenges, prior multi-chiplet GPU works introduced mechanisms to improve data locality or reduce synchronization overhead. However, these techniques perform these optimizations in isolation, limiting their benefits. Conversely, we propose **CAQS**, a novel **C**ache-**Aware** **Q**ueue **S**cheduler that intelligently utilizes **both** locality and synchronization information when deciding where to schedule application phases to reduce the impact of inter-chiplet NUMA effects. Overall, across 18 popular GPU workloads CAQS improves geomean performance (30%, 28%, 6%), energy efficiency (36%, 19%, 27%), and reduces network traffic (80%, 61%, 80%), over modern GPUs and the state-of-the-art CPElide and LADM, respectively. Moreover, CAQS's advantages grow for more concurrent streams.

Contents

List of Figures

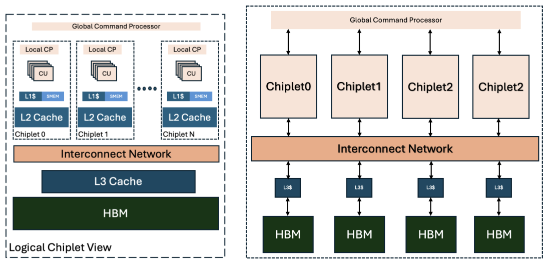
List of Tables

Chapter 1

Introduction

Workloads including high performance computing (HPC) and machine learning (ML) continue to exhibit voracious demands for compute and memory [1, 2]. Concurrently, the waning of Moore's Law and end of Dennard's Scaling limit the performance benefits that transistor scaling traditionally provided [3]. Thus, modern systems are facing challenges above from applications and below from the slowing of transistor scaling [4]. As a result, systems are embracing heterogeneous mixes of conventional cores and specialized accelerators to continue scaling performance and energy efficiency. Specialized accelerators are frequently used to improve the efficiency of computations that run inefficiently on conventional, general-purpose processors. To keep pace with the insatiable demand for performance, accelerators are growing larger, integrating more compute cores, specialized processing units, and sophisticated memory hierarchies [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]. However, this monolithic scaling approach faces challenges in manufacturing yield, cost, and thermal limits [15, 16, 17, 18].

Given these challenges, designers are embracing chiplets [19, 20], also known as multi-chip modules (MCMs) [15, 21, 22] (Figure 1.1a). Chiplets combine multiple smaller chips into a large, aggregated system using interposers [23, 24, 16, 25] or other packaging techniques such as crosslinks [4, 26]. By breaking down large, complex chips into smaller, interconnected semiconductor dies, chiplet architectures enable more flexible manufactur-



(a) MCM GPU (b) Distributed Memory

Figure 1.1: MCM GPU system's high-level view.

		Implicit Synchronization at Kernel Boundaries	
		Always Required	**Sometimes Required**
Data Access Optimizations	**No**	Current GPUs	CPElide [27]
	Yes	CODA[28], GRIT[29], LADM[17], Milic, et al. [30], SAC[31], SelRep [32]	***CAQS***

Table 1.1: Mechanisms to improve inter-kernel reuse on chiplet-based GPUs.

ing, improved yield, enhanced design flexibility, and reduced costs. Thus, chiplets offer a modular strategy to continue scaling performance and offer a potentially transformative architectural approach to address the escalating challenges of monolithic chip design.

Although chiplet-based designs offer a number of benefits over more conventional, monolithic designs, they also introduce unique challenges. In particular, maintaining data locality and minimizing latency overheads is more challenging in chiplet-based designs. Compared to monolithic designs, chiplets introduce an additional level of indirection in the memory hierarchy. As shown in Figure 1.1a, chiplets utilize private L1 and L2 caches located within the chiplets, as well as shared L3 caches. Furthermore, chiplet-based designs also distribute memory across chiplets (Figure 1.1b). This introduces significant indirection and NUMA (Non-Uniform Memory Access) overheads whenever inter-chiplet communication is required. Thus, NUMA-aware design is essential to optimize performance and fully harness the potential of chiplet-based architectures.

Although all chiplet-based designs face challenges from inter-chiplet NUMA effects, vendors have demonstrated how they can leverage a) complex coherence protocols (e.g., MOESI) that minimize the overhead of synchronization and increase locality [4] and b) OS support to mitigate NUMA effects [33, 34, 35] in homogeneous, multi-chiplet CPUs. Unfortunately, these solutions clash with how accelerators are typically designed. While monolithic accelerators had some NUMA overheads, multi-chiplet designs exacerbate these NUMA effects. Most accelerators assume relatively flat memory hierarchies, have limited OS support, and utilize relatively simple coherence protocols with heavyweight synchronization operations [36, 37, 38, 39, 40, 41]. Consequently, multi-chiplet accelerators cannot use the same solutions as multi-chiplet CPUs. In this work, we focus on the increasingly ubiquitous multi-chiplet GPUs due to their combination of programmability, performance, and energy efficiency. However, since many accelerators suffer from NUMA effects and utilize similar approaches [42, 43, 44, 45], CAQS also applies to them (discussed further in Chapter 6).

Previous NUMA-aware Approaches: Prior work has shown that the impact of NUMA effects on chiplet-based GPUs is severe: **29%-54% average performance loss** versus equivalently sized monolithic accelerators [27, 17, 39, 46]. Accordingly, they have made significant improvements to reduce NUMA penalties. Broadly, these efforts can be divided into two categories: scheduling work close to the data (*Data Access Optimizations*) and reducing the overhead of implicit GPU synchronization at kernel boundaries (*Implicit Synchronization at Kernel Boundaries*). We use these factors as axes in Table 1.1 to describe the state-of-the-art in reducing NUMA overheads (related work discussed further in Chapter 7).

Current multi-chiplet GPUs: At kernel boundaries, current multi-chiplet GPUs [10, 12, 14] must implicitly invalidate all valid data and flush all dirty data from all caches in a chiplet to keep the data consistent across chiplets [47, 48, 49]. In chiplet-based GPUs like those in Figure 1.1, this means that all data in both L1 **and** L2 caches, across all chiplets, must be invalidated/flushed at kernel boundaries. Thus, unlike in monolithic GPUs, in modern chiplet-based GPUs data cannot be retained in the L2 cache across kernels – significantly hurting performance. Although they do allow programmers to bind GPU streams to a specific chiplet, and offer virtualization techniques such as MIG [50] and MxGPU [51] to provide isolated access to a subset of the GPU's resources, these techniques are often unable to improve per-chiplet locality and must implicitly synchronize at kernel boundaries.

Data Access Optimizations: Prior work such as CODA [28], GRIT [29], Locality-Aware Data Management (LADM) [17], and Milic, et al. [30] proposed various page placement policies to co-locate data and compute, including first touch placement, on-touch migration, access counter-based migration, and page duplication to reduce NUMA penalties by reducing remote accesses. LADM attempts to schedule threads on the chiplet(s) where the data resides utilizing a compile-time static index analysis of a GPU program to dynamically make decisions about data placement, thread scheduling, and remote caching [17]. Conversely, GRIT attempts to dynamically identify the appropriate page placement schemes instead of using a single page placement policy [29]. Other approaches such as SAC and SelRep improve either memory- or SM-side LLC (e.g., L3) cache bandwidth [31, 32]. While each of these schemes improves locality, all must implicitly synchronize at kernel boundaries. Thus, they cannot exploit inter-kernel L2 reuse.

Reducing Implicit Synchronization Overhead: The above approaches optimize data accesses at the L3 cache or main memory. This approach is common because at kernel boundaries GPUs must implicitly invalidate all valid data and flush all dirty data from all caches in a chiplet to keep the data consistent across chiplets [47, 48, 49]. However, recent work, CPElide (discussed further in Chapter 2.1.2), reduces the overhead of these implicit synchronizations [27]. CPElide observed that many implicit kernel synchronizations are redundant, and reduces the overhead from these flushes/invalidations by tracking the kernel accesses using the GPU's Command Processor's (CP, discussed further in Chapter 2.1.1) to perform implicit synchronization only when required. Thus, CPElide keeps more data in the chiplet's L2 caches and reduces the overhead of implicit, kernel boundary synchronization. However, since CPElide does not consider the data locality when scheduling kernels, it cannot always avoid NUMA penalties.

Our Approach: CAQS: Prior work either require coarse-grained, redundant implicit synchronization or do not consider data placement optimizations when scheduling work. To overcome the shortcomings of both approaches, we propose **CAQS**, which considers

both locality and synchronization information when making GPU scheduling decisions. Like CPElide, we exploit the fact that the GPU CP already *has a global view of what work groups (WGs) are being sent to each chiplet and what data structures each work group (WG) in a GPU kernel accesses at a given time* [27]. However, neither modern GPUs nor CPElide leverage this information when the CP's queue scheduler (stream scheduler in NVIDIA parlance) decides which chiplet(s) to schedule a GPU kernels WGs on. Thus, CAQS introduces a novel queue scheduler into the GPU CP that leverages the CP's dynamic tracking information to schedule GPU kernels on chiplet(s) to improve L2 reuse, avoid expensive inter-chiplet communication, and reduce NUMA penalties.

To demonstrate CAQS's efficacy we evaluate it over 18 workloads from traditional GPGPU, graph analytics, and HPC. Compared to modern multi-chiplet GPUs and the state-of-the-art CPElide and LADM, CAQS significantly improves geomean performance (30%, 28%, 6%), improves average L2 hit rate (27%, 25%, 26%), reduces geomean energy consumption (36%, 19%, 27%), and decreases geomean network traffic (80%, 61%, 80%), respectively. Moreover, as concurrent GPU streams increase, CAQS outperforms the best baseline, LADM, by 29% geomean for 4-stream and 22% geomean for 6-stream workloads. To the best of our knowledge, CAQS is the first multi-chiplet GPU queue scheduler to combine locality- **and** synchronization-awareness to combat multi-chiplet GPU NUMA effects. Additionally, because CAQS is integrated into the GPU CP, it does not require hardware changes and can adapt to changing workload behavior.

Chapter 2

Background

2.1 Multi-Chiplet GPU Architecture

Modern GPU systems are often made of multiple GPUs that are connected via high-bandwidth interconnects such as PCIe [52], NVLink [53], or xGMI [54]. Each GPU in this multi-GPU system is composed of multiple chiplets, known as a multi-chiplet GPU or an MCM GPU. In a multi-chiplet GPU, multiple chiplets communicate over high-bandwidth interconnects (Figure 1.1a). Each of these chiplets has multiple Streaming Multiprocessors (SMs)/Compute Units (CUs) and a cache hierarchy that is connected to the L3 cache and high bandwidth memory (HBM). While all chiplets share the L3 cache and main memory, the L3 and HBM banks are physically distributed across multiple chiplets (Figure 1.1b). These distributed resources cause non-uniform memory accesses (NUMA) – local chiplet accesses are faster than remote ones [39, 15]. In this work, we focus on alleviating NUMA penalty impacts within a chiplet-based GPU. However, as we discuss in Chapter 6, CAQS could also be applied to multi-GPU systems that combine multiple chiplet-based GPUs.

2.1.1 GPU Command Processors

As shown in Figure 1.1, modern GPUs utilize an embedded, programmable RISC microprocessor, the CP, to act as the interface between the host and accelerator. In modern multi-chiplet GPUs, GPU vendors typically have a CP per chiplet [12, 14], and elect

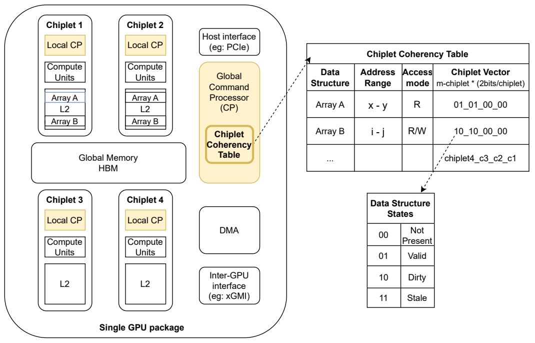
Algorithm 1: Default Queue Scheduler.

```
Input: Function running on each CPU cur_cpu
for all HW queues do
    Pick the highest priority queue, $Q_i$, from all the HW queues
end
forall work groups (WGs) in kernel K at the head of Queue $Q_i$ do
    // Round-Robin Scheduler
    schedule on chiplet C = (lastChiplet + 1) % totalChiplets
    lastChiplet = C
end
```

one as the *leader*. A GPU driver (e.g., AMD's ROCm [55]) maps the GPU program into software queues and enqueues the program's kernels, along with any memory management and inter-kernel synchronization, as a packet(s). The CP's **packet processor** then maps each kernel onto a hardware compute queue using its **queue scheduler**. A GPU queue scheduler orchestrates the efficient allocation and management of computational tasks across GPU resources, including determining task execution order and resource mapping. When dispatching kernel invocations, it must manage the critical interface between application-level workload requirements and hardware-level execution capabilities: balancing competing objectives: minimizing kernel launch latency, maximizing hardware utilization, and ensuring fair resource distribution. As shown in Algorithm 1, modern GPU queue schedulers typically dispatch all WGs from a kernel in round-robin (RR) fashion across the available CUs [56, 57] before switching to another kernel. GPUs also support multiple hardware queues to manage independent work submitted asynchronously with GPU streams [58, 59, 60]. Typically each stream is mapped to a queue and each queue holds one or more kernels from that stream. The CP maintains intra-stream and inter-kernel dependencies but often executes different streams concurrently.

2.1.2 CPElide

Since CAQS builds on key ideas from CPElide, we first discuss how CPElide works. Figure 2.1 demonstrates the overall CPElide architecture [27]. In addition to having local,



Chiplet Coherency Table			
Data Structure	Address Range	Access mode	Chiplet Vector m-chiplet * (2bits/chiplet)
Array A	x - y	R	01_01_00_00
Array B	i - j	R/W	10_10_00_00
...			chiplet4_c3_c2_c1

Data Structure States	
00	Not Present
01	Valid
10	Dirty
11	Stale

Figure 2.1: CPElide architecture [27].

```
typedef tuple<Addr_t, Addr_t, LogicalChipletID>
rangeChiplet;
vector<rangeChiplet> C_ranges(numSchedChip) =
  {make_tuple(C_d[start], C_d[mid], 0),
   make_tuple(C_d[mid+1], C_d[end], 1)};
vector<rangeChiplet> A_ranges(numSchedChip) =
  {make_tuple(A_d[start] , A_d[mid], 0),
   make_tuple(A_d[mid+1] , A_d[end], 1)};
hipSetAccessModeRange(square, C_d, 'R/W', C_ranges);
hipSetAccessModeRange(square, A_d, 'R', A_ranges);
hipLaunchKernelGGL(square,..., C_d,  A_d, N);
```

Listing 2.1: CPElide's proposed API calls to label the memory accesses in a sample kernel [27].

per-chiplet CPs like modern GPUs (Chapter 2.1.1), CPElide further splits the CPs functionality by adding a *global CP* to handle communication with the host. The global CP and leader CP are similar, except the global CP does not need to manage local, per-chiplet functionality. Moreover, CPElide leverages information already available in the CP to monitor kernel memory accesses across chiplets. Specifically, CPElide inspects each queue entry's kernel object to identify coarse-grained data structure access information. Programmers or compilers provide this information to the (global) CP via an API, which CPElide tracks in its *Chiplet Coherency Table* (*CCT*).

Listing 2.1 shows an example of how CPElide adds new API calls to AMD's open-source ROCm GPU API to pass this information to the CP [27]. Unlike coherence protocols, CPElide tracks access information coarsely per data structure, and only updates it at kernel boundaries. Thus, it conservatively estimates what data may be in each chiplet's L2 caches. When launching a kernel, CPElide's queue scheduler uses the CCT's information to decide which chiplets require implicit synchronization to ensure correctness, and which ones do not (L1 caches must still be invalidated and flushed). Thus, CPElide implicitly synchronizes only on the chiplets that require it, reducing the overhead of implicit synchronization and increasing L2 reuse in chiplets. However, CPElide does not consider locality when scheduling. Consequently, it only realizes significant benefits from retaining data in a chiplet's L2 cache when a kernel happens to be scheduled on a chiplet with reusable data. However, in Chapter 5 we show such serendipitous occurrences are infrequent.

Chapter 3

Design

3.1 Architecture

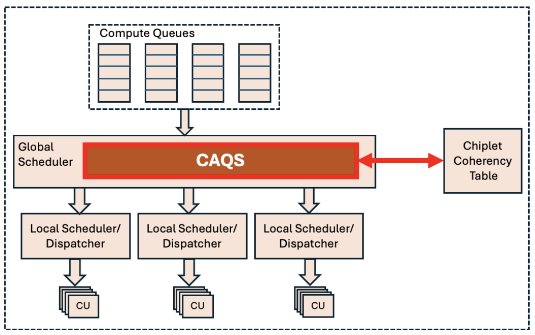

Compute Queues
Global Scheduler
CAQS
Chiplet Coherency Table
Local Scheduler/ Dispatcher
Local Scheduler/ Dispatcher
Local Scheduler/ Dispatcher
CU
CU
CU

Figure 3.1: CAQS Design (changes in red).

Figure 3.1 presents the CAQS's overall architecture. CAQS's primary objective is to reduce the NUMA penalty in chiplet-based GPUs by dynamically scheduling kernels from queues to increase inter-kernel cache reuse and retention. Unlike approaches (e.g., CODA, LADM) that rely on static profiling or predictions, CAQS leverages runtime information to make adaptive scheduling decisions. We leverage information on how data structures are being accessed by GPU kernels to determine what data has been accessed by recent kernels and where (i.e., which chiplet(s)) it has been accessed on. This per data structure access information is already available in the GPU (global) CP, but neither modern GPUs (Chapter 2.1.1) nor CPElide (Chapter 2.1.2) use it in their queue schedulers.

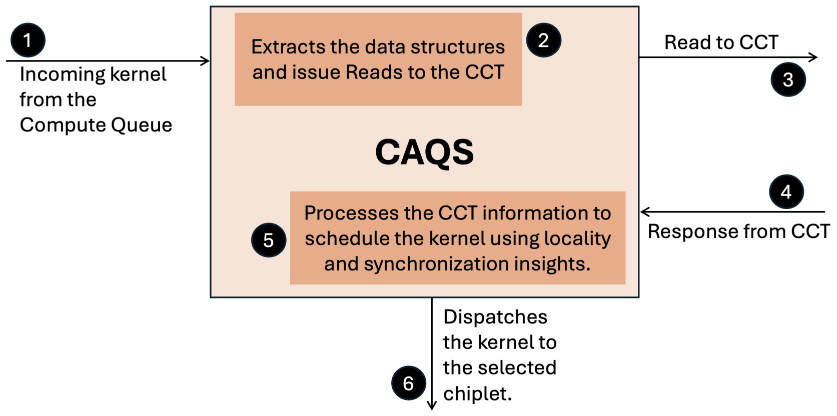

1
Incoming kernel from the Compute Queue
Extracts the data structures and issue Reads to the CCT
2
Read to CCT
3
CAQS
4
Response from CCT
5
Processes the CCT information to schedule the kernel using locality and synchronization insights.
6
Dispatches the kernel to the selected chiplet.

Figure 3.2: CAQS Req/Resp Flow

CAQS's key insight is that queue schedulers can leverage this coarse-grained information to cheaply, quickly determine when inter-kernel reuse may be possible in multi-chiplet GPUs. Thus, CAQS creates a novel, NUMA-aware queue scheduler in the GPU's global CP that builds on CPElide. CAQS uses CPElide to eliminate implicit synchronization operations, enabling L2 caches to retain previous kernel's data beyond kernel boundaries. These data structures are tracked in CPElide's CCT (Chapter 2.1.2). However, unlike CPElide, CAQS's new queue scheduler inspects the CCT to identify whether a data structure resides in any chiplet's L2 cache. Specifically, for each kernel at the front of a hardware compute queue (Chapter 2.1.1), CAQS's identifies all data structures the kernel will access, potentially including their address ranges. Then CAQS checks the CCT for potential inter-kernel reuse for these data structures, and finalizes its scheduling decision.

3.2 Dynamic Queue Scheduling Mechanism

Figure 3.2 illustrates CAQS's high-level request/response flow for an incoming kernel. Likewise, Algorithm 2 shows how we implement CAQS. When a kernel reaches the head of a hardware compute queue (1), CAQS extracts the data structure(s) the kernel will access (2). CAQS only considers chiplets with available resources – if a chiplet is completely utilized, CAQS will not attempt to schedule this kernel's WGs on it. Given the list of chiplets with available resources, CAQS then queries the CCT to determine the presence

Algorithm 2: CAQS Queue Scheduler. For simplicity, this algorithm assumes a given kernel is scheduled on a single chiplet.

```
Input: Function running on global CP
forall HW queues do
    Pick the highest priority queue, $Q_i$, from all the HW queues
end
schedOptions = NULL
$K$ = head[$Q_i$] // kernel $K$ at the head of Queue $Q_i$
// Extract all the Data Structures from $K$
kernelDataStructures = extractDataStructures($K$)
foreach data structure ($D_j$) in kernelDataStructures do
    foreach chiplet $C_i$ do
        // Check data structure in CCT for $C_i$
        cctData = readCCT($D_j$, $C_i$)
        // Check if $D_j$ may be in $C_i$ and is valid/dirty
        if cctData.match and (cctData.valid or cctData.dirty) then
            schedOptions.pushBack($C_i$)
        end
    end
end
// Determine which chiplet is best fit from available options
bestScore = 0
bestChiplet = 0
foreach chiplet $O_i$ in schedOptions do
    score = calcScore($K$, $O_i$)
    if score > bestScore then
        bestScore = score
        bestChiplet = $O_i$
    end
end
if bestScore > 0 then
    ... // Perform appropriate implicit synchronization on $C_i$
    forall work groups (WGs) in $K$ do
        Schedule on chiplet bestChiplet
    end
    lastChiplet = bestChiplet
end
else
    forall work groups (WGs) in $K$ do
        // Round-Robin Scheduler
        Schedule on chiplet $C$ = (lastChiplet + 1) % totalChiplets
    end
    lastChiplet = $C$
end
```

```
// A Workload with 3 kernels:
// Kernel1 with Array A (R) as
// input and Array B (R/W) as output
hipSetAccessMode(Kernel1, B_d, 'R/W');
hipSetAccessMode(Kernel1, A_d, 'R');
hipLaunchKernelGGL(Kernel1,..., B_d, A_d, N);

// Kernel2 with Array B (R) as
// input and Array C (R/W) as output
hipSetAccessMode(Kernel2, C_d, 'R/W');
hipSetAccessMode(Kernel2, B_d, 'R');
hipLaunchKernelGGL(Kernel2,..., C_d, B_d, N);

// Kernel3 with Array D (R) as
// input and Array E (R/W) as output
hipSetAccessMode(Kernel3, E_d, 'R/W');
hipSetAccessMode(Kernel3, D_d, 'R');
hipLaunchKernelGGL(Kernel3,..., E_d, D_d, N);
```

Listing 3.1: An example workload with 3 kernels.

and state of each data structure required by the kernel in those chiplets' L2 caches (3). CCT sends the response back to CAQS (4). If the CCT indicates that a data structure has been previously accessed, CAQS checks whether the data is still valid or dirty (5). If the data is valid or dirty, CAQS identifies this chiplet as one to consider scheduling the kernel on. If a read-only data structure is valid on multiple chiplets, CAQS selects the first chiplet as a tiebreaker (not shown in Algorithm 2 for simplicity). Moreover, if a required data structure has not been previously accessed or is not valid or dirty, CAQS defaults to RR scheduling. This strategy enables CAQS to balance the load across multiple chiplets instead of binding a stream to any one chiplet, which could negatively impact performance [33]. Once the queue scheduler has made its decision and before the kernel is dispatched, CPElide performs any necessary implicit acquire and release operations. Finally, CAQS schedules the kernel onto the identified chiplet(s) (6).

3.3 Putting It All Together

Consider a single-stream workload with three kernels, as described in Listing 3.1, running on a 2-chiplet GPU with an L2 cache that can hold two large data structures. In Listing 3.1, we use the same API calls as CPElide to pass per-kernel data structure (`A_d`, `B_d`, `C_d`, `D_d`, `E_d`) access information, since our approach builds on it.

Baseline System: In the baseline system (Chapter 2), the first kernel, `Kernel1`, is scheduled on Chiplet 0. Before `Kernel1` is launched, all chiplets are invalidated, and upon completion, any dirty data (e.g., `B_d`) is flushed from each chiplet. Next, the second kernel (`Kernel2`) is scheduled on Chiplet 1 following the baseline's round robin queue scheduling policy, undergoing a similar invalidate-and-flush process at kernel boundaries. Finally, the third kernel (`Kernel3`) is subsequently scheduled back on Chiplet 0, repeating the implicit invalidates-and-flushes at its boundaries.

System with CAQS: When `Kernel1` is enqueued at the global CP, CAQS extracts its data structures (`A_d`, `B_d`) and checks their status in the CCT. Since this is the first kernel, the CCT has no entries. Thus, CAQS defaults to scheduling `Kernel1` on Chiplet 0. Before scheduling `Kernel1`, CAQS checks and determines that no acquire or release operations are required since the CCT is empty (unlike in the baseline); thus it dispatches `Kernel1` WGs to Chiplet 0. When `Kernel1` completes the L2 cache on Chiplet 0 retains `A_d` and `B_d`.

Subsequently `Kernel2` is enqueued. For `Kernel2`, CAQS extracts `B_d` and `C_d` and queries their status in the CCT. Since `Kernel1` wrote `B_d` in Chiplet 0, it may still be present in Chiplet 0's L2 cache (Chapter 2.1.2). Thus, CAQS schedules `Kernel2` on Chiplet 0. Moreover, since `Kernel2` is scheduled on Chiplet 0 this reduces the overhead of flushing Chiplet 0 (unlike in the baseline). Accordingly, `Kernel2` benefits from reusing `B_d` in Chiplet 0's L2 cache, reducing main memory accesses and avoiding synchronization overhead. After `Kernel2` completes, Chiplet 0 retains `B_d` and `C_d`, while `A_d` is evicted (per LRU).

For `Kernel3`, CAQS similarly extracts `D_d` and `E_d` and queries them in the CCT. Neither of these structures has been accessed previously. To preserve the contents of Chiplet 0's L2 cache, particularly `B_d` and `C_d`, CAQS schedules `Kernel3` on Chiplet 1 (Chapter 2.1.2). Thus, Chiplet 0 retains `B_d` and `C_d` while accessing `D_d` and `E_d` on Chiplet 1 – balancing the cached data structures across the chiplets. Hence, CAQS reduces evictions, and retains more data in the L2 cache, increasing L2 reuse opportunities in subsequent kernels. Additionally, this strategy ensures that CAQS effectively load balances, avoiding rigidly binding streams to chiplets. In Chapter 5, we show that CAQS's relatively simple changes to the GPU queue scheduler provides the best of both worlds: balancing locality like LADM and avoiding implicit synchronization overheads like CPElide, without the downsides of either.

3.4 Overheads

Queue Scheduler Overhead: CAQS reads the CCT for each data structure associated with a kernel, which adds overhead. We model this overhead within our system. Given that the size of the CCT is ≈2KB [61], CAQS also fits into the global CP's private memory. The number of reads required varies based on the number of data structures accessed by each kernel. However, prior work has shown that most GPU programs access eight or fewer data structures per kernel [62] [63]. In our experiments, the average number of data structures accessed per kernel is four. Furthermore, because CAQS schedules kernels on chiplet(s) where the L2 cache may already contain the required data, it reduces memory accesses and avoids the invalidations and flushes typically seen in CPElide when subsequent kernels are scheduled on different chiplets. Thus, CAQS's runtime mechanism has minimal overhead (discussed further in Chapter 4.2) and negligible impact (Chapter 5).

Delayed Writebacks: Unlike the baseline, which incurs overhead from implicit flushes and invalidations for the per-chiplet L2 caches at kernel boundaries, CAQS delays these operations to encourage inter-kernel L2 reuse. However, subsequent kernels may evict this dirty data to make space for other data. Thus, CAQS may have write-back overheads. Since these writebacks occur during kernel execution in CAQS, they may be on the critical path. However, the performance impact of these writebacks is usually limited, and the increased reuse far outweighs their cost (Chapter 5).

False Positives: Unlike coherence protocols, CPElide's CCT coarsely tracks information at the data structure granularity, and only updates its state when kernels are launched.

Thus, CAQS may encounter false positives if some of the data is no longer in a chiplet's L2 cache (e.g., due to cache evictions). However, in this situation CAQS provides the same behavior as CPElide and LADM, which also do not perform fine-grained tracking. Moreover, in Chapter 5 we show that our approach provides significant benefits.

Chapter 4

Methodology

4.1 Baseline GPU Architecture

Similar to prior work, we evaluate CAQS using a tightly coupled CPU-GPU architecture with a unified address space with shared memory and coherence caches [27]. Figure 1.1 illustrates our GPU system, which is similar to prior work [27, 17, 39, 26]. All CPU cores and GPU CUs are connected via a shared, inclusive L3, which also serves as the directory. Each GPU chiplet has a private L1 cache and LDS per CU, and an L2 cache shared across the chiplet's CUs. The HBM and L3 cache are distributed across chiplets as shown in Figure 1.1b. The interconnect connects the chiplets, which routes L2 misses to the appropriate "home chiplet" for the address [39].

4.2 System Setup

As noted in prior work, GPU CPs can be reprogrammed [27, 64, 65, 66, 67]. However, since GPU vendors have not disclosed an API, doing so is difficult. As a result, prior work like CPElide simulate their CP extensions in gem5 [27]. Since we compare against CPElide (Chapter 4.4), we utilize the same approach to reduce the impact of system differences when comparing approaches. Thus, we simulate CAQS in gem5, which recent work has extended to support multi-chiplet GPUs [27, 68]. Moreover, while other popular simulators support modern GPUs [69, 70], gem5 has the most detailed CP model and models GPUs with high fidelity [71].

GPU Feature	**Configuration**
GPU Clock	1801 MHz
CP Latency	31 cycles
Num Chiplets	6
CUs/Chiplet; Complexes/Chiplet	60; 1
SE/Chiplet, SA/SE	4, 1
Total CUs	360
Num SIMD units/CU	4
Max WF/SIMD unit	10
Vector/Scalar Reg. File Size / CU	256/12.5 KB
Num Compute Queues	256
LI Instruction Cache / 4 CU	16 KB, 64B line, 8-way
L1 Data Cache / CU	16 KB, 64B line, 16-way
L1 Latency	140 cycles
LDS (Local Data Share) Size / CU	64 KB
LDS Latency	65 cycles
L2 Cache/chiplet	8 MB, 64B line, 32-way (8 banks per chiplet)
Local/Remote L2 Latency	269/390 cycles
L2 Write Policy	Write-back with write allocate
L3 Size	16 MB, 64B line, 16-way (64 banks)
L3 Latency	330 cycles
Main Memory	16 GB HBM, 4H stacks, 1000 MHz (64 banks)
Inter-chiplet Interconnect BW	768 GB/s

Table 4.1: Simulated baseline GPU parameters.

Specifically, we utilize CPElide's publicly available artifact [61] as our starting point. We implemented CAQS into its gem5 GPU global CP, including adding support for communicating between the global scheduler (Chapter 2.1.2) and CPElide's *CCT* (Chapter 2.1.2). Moreover, like CPElide we use ROCm 1.6 [55], gem5 v21.1 [72, 73], and configure the GPU to emulate CPElide's GPU setup. Table 4.1 summarizes the common key system parameters, which are based on an AMD Radeon VII GPU. To measure the energy consumption we leverage prior work's per-access GPU energy models [74, 75, 76, 77, 67], scaled for multi-chiplet GPUs [27]. In our simulated system the CP frequency is 1.5 GHz [78] and the CPs private memory's access latency is 31 cycles [79]. The global CP and local CP are connected via a high bandwidth crossbar, with 65 cycles of unicast latency and 100 cycles of broadcast latency. Similar to CPElide and prior work, the modeled local/global CP latency is 2 μs [80, 52, 57]. We also model CAQS's over-

Application	**Input**
Square [81, 82]	512K 1 2 2048 256
BabelStream [83, 84]	512K
BFS [85, 86]	graph128k.txt
Gaussian [85, 86]	16x16
Backprop [85, 86]	64K
Color-max [87]	AK.gr
HACC [88]	0.5 0.1 512 0.1 2 N 12 rcb
Hotspot [85, 86]	512 2 20 temp_512 power_512
LUD [85, 86]	512.dat
LULESH [88]	1.0e-2 10
Pennant [88]	noh.pnt
SSSP [87]	AK.gr
FW [87]	512_65536.gr
BTree [85, 86]	mil.txt
BC [87]	AK.gr
DWT2d [85, 86]	rgb.bmp 4096x4096
Pathfinder [85, 86]	200000 100 20
SRAD_v2 [85, 86]	2048 2048 0 127 0 127 0.5 2

Table 4.2: Evaluated Benchmarks

heads (Chapter 3.4). However, since CAQS only affects internal CP communication paths (e.g., *CCT*-queue scheduler communication), these changes do not affect the area beyond CPElide's original ∼2 KB [27], which fits in the CP's private memory and does not change the GPU's area. Specifically, our changes affect the CP's processing latency by ≈100ns, which our simulations factor in when processing new GPU kernels. However, this overhead is hidden for all but an application's first kernel since the kernel runtimes are much larger.

4.3 Workloads

We evaluate CAQS across 18 traditional GPGPU [85, 86], graph analytics [87], and HPC [88, 89] applications with diverse memory access patterns from gem5-resources [81]. Table 4.2 summarizes these workloads, which launch up to 450 kernels. Like our modeled system (Chapter 4.1), all gem5-resource's workloads use unified virtual memory (UVM) and page-aligned memory allocations [90]. We also selected these benchmarks to overlap with LADM and CPElide's (Chapter 4.4) studied benchmarks. Like CPElide, these workloads never overflow the *CCT*. We scaled the input sizes for all workloads to stress the L2 cache capacity [91]. Moreover, to evaluate CAQS's behavior in various scenarios, our workloads range from high or moderate inter-kernel reuse to low inter-kernel reuse [92, 91, 93]. We also simulated a subset of these workloads for 4 and 6 streams to evaluate the scalability of CAQS. Some of our evaluated works (HACC, Pennant) use multiple streams by default. Thus, we exclusively study them when we evaluate larger numbers of streams (Section 4.6).

4.4 Configurations

We compare CAQS against 3 state-of-the-art methods using the system described in Chapter 4.2:

Baseline: Baseline models a modern, multi-chiplet GPU [94, 95, 10, 12, 96, 14]. It uses a gem5's VIPER coherence protocol [80], which is similar to modern GPUs [36, 37, 38, 39, 40, 97, 41], extended for multi-chiplet GPUs [68]. It forwards remote requests to the home chiplet, writes through remote stores, and writes back local stores. Moreover, it uses a RR queue scheduler where each subsequent kernel is scheduled on the next chiplet (Chapter 2.1.1).

CPElide: CPElide [27] is a state-of-the-art approach for avoiding unnecessary implicit synchronization at kernel boundaries in chiplet-based GPUs. CPElide uses **Baseline**'s coherence, forwarding policy, queue scheduler, and write policies, but elides synchronizations by tracking data structure access information in its global CP's *CCT*.

LADM: LADM [17] is a state-of-the-art approach for placing pages and scheduling WGs on multi-chiplet GPUs (Chapter 1) which we implemented in gem5. LADM also uses the VIPER coherence protocol. We optimistically model LADM's best-case scenario when threads are always scheduled on the same chiplet where the data is present. Thus, our LADM implementation does not need its underlying LASP page placement policy. Instead we can utilize a first touch page allocation policy to allocate the pages wherever the data is first accessed [15, 39]. As a result, our implementation may overstate LADM's efficiency when this is not possible.

CAQS: Our proposed CAQS approach (Chapter 3) implemented on top of CPElide. Like

CPElide, CAQS tracks the kernel accesses in the *CCT*, and leverages it to schedule a kernel in a way that optimizes the L2 reuse and retention.

4.5 Design Decisions

We also made the following design choices for all the different configurations (Chapter 4.4):

Page Placement Policy: Since CAQS's focuses on improving inter-kernel reuse, we utilize the same page placement policy for all configurations. Specifically, we use a First Touch page placement policy since it usually performs well on modern multi-chiplet GPUs [15, 27, 39]. The first touch policy determines the home node (chiplet) for a given physical address. Moreover, different page placement policies can be used with CAQS (discussed further in Chapter 6).

Scheduling Kernels on One Chiplet: Modern multi-chiplet GPUs can be configured to schedule kernels on a single chiplet or across chiplets [10, 12, 14]. However, for many systems, as well as virtualization techniques like MIG [50] and MxGPU [51], it is common to partition the GPU at chiplet boundaries to improve locality. Thus, we model this behavior and do not split a given kernel's WGs across multiple chiplets (alternatives discussed further in Chapter 6).

4.6 Sensitivity Study: Number of Streams

Modern GPU workloads also often utilize multiple streams to improve the utilization. To understand the impact of this behavior and test the scalability of CAQS, we study the sensitivity of different numbers of streams in the system for all of our applications: 1, 4, and 6 streams. As in prior work, we run a subset of our benchmarks (Table 4.2) to run multiple parallel streams where each stream performs independent work [27, 98].

Chapter 5

Results

Figures 5.1 and 5.2 present the relative speedup and absolute L2 hit percentage, respectively, for all single-stream workloads across various configurations on a 6-chiplet GPU. Figure 5.3 depicts the normalized network traffic, broken into local and remote traffic. Figure 5.4 shows the normalized energy consumption for each configuration, broken down into main memory, L1, L2, remote, and the remaining (e.g., core) energy components. Overall, *CAQS* demonstrates significant improvements over *Baseline*, *CPElide*, and *LADM*. *CAQS* provides a geomean speedup (30%, 28%, 6%), average increase in L2 hits (27%, 25%, 26%), geomean energy consumption reduction (36%, 19%, 27%), and geomean network traffic reduction (80%, 61%, 80%) over *Baseline*, *CPElide*, and *LADM*, respectively. These results demonstrate the benefits of considering both locality and synchronization overheads in multi-chiplet GPU scheduling decisions. Moreover, for 4- and 6-stream workloads that further stress the system, *CAQS*'s advantages over the alternatives become even more substantial: *CAQS* outperforms the best baseline, *LADM*, by 29% geomean for 4-stream workloads and 22% geomean for 6-stream workloads. Thus, *CAQS* scales better and provides greater benefits even as GPU configurations grow.

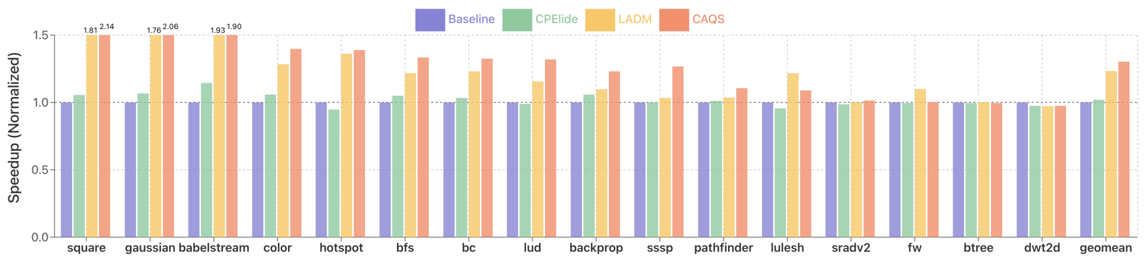


Figure 5.1: Single stream apps' speedup for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

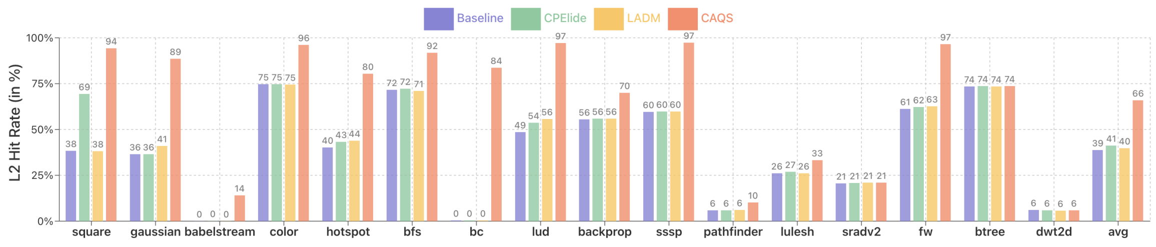


Figure 5.2: Single stream apps' L2 hit rates for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

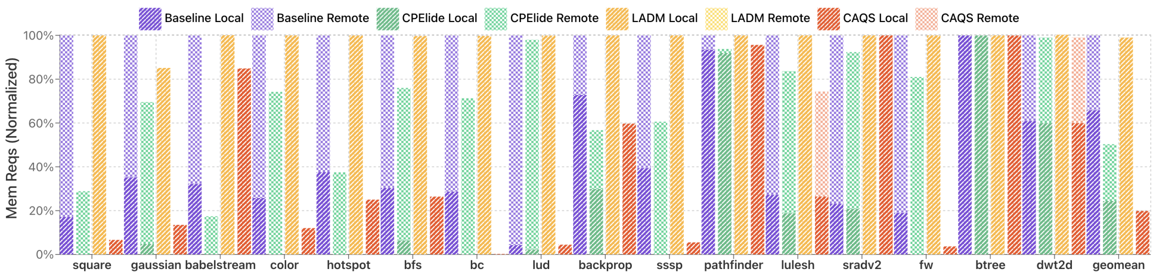


Figure 5.3: Single stream apps' network traffic for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

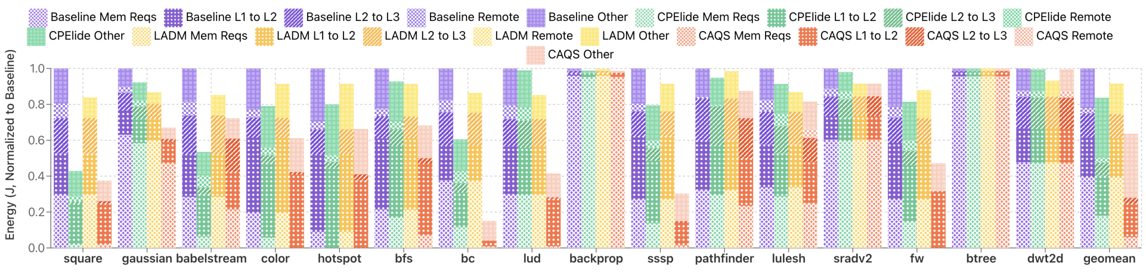


Figure 5.4: Single stream apps' energy usage for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

5.1 Single Stream

5.1.1 CAQS vs *Baseline*

Figure 5.1 compares *Baseline*'s and *CAQS*'s normalized speedups for single-stream simulations. Unlike *Baseline*, *CAQS*'s consideration of locality and implicit synchronization information enables it to increase cache hit rates, reduce memory and remote accesses, and reduce overhead from implicit flushes and invalidations at kernel boundaries. As a result, *CAQS* provides a 30% geomean speedup over the *Baseline*. This is further emphasized by Figure 5.2, which shows that *CAQS* consistently enhances L2 cache reuse: 27% average L2 hit rate improvement versus *Baseline*. Importantly, *CAQS*'s L2 hit rate is always equal to or better than *Baseline*. However, the relationship between speedup and L2 hit rate improvement is not linear. While higher L2 hit rates generally benefit performance, workloads with different sensitivities to memory latency exhibit varying speedups.

Applications with Inter-Kernel Reuse: Figure 5.3 demonstrates *CAQS*'s ability to improve reuse and keep data local within a chiplet. In particular, for workloads with opportunities for inter-kernel reuse (all but BTREE, DWT2D, Pathfinder, and SRAD_v2), CAQS effectively identifies and leverages reuse opportunities in the *CCT* from previously executed kernels (Chapter 3.2). This enables *CAQS* to keep the stream localized on that chiplet. Thus, *CAQS* significantly reduces remote accesses (80% geomean less network traffic than *Baseline*). Similarly, *CAQS* improves L2 hit rates for many applications with inter-kernel reuse: six benchmarks achieved L2 hit rates exceeding 90%, including LUD, SSSP, and FW reaching ≈97%.

CAQS is also effective for BC (84% L2 hit rate) and Square (94% L2 hit rate). Although both are memory-bound workloads, Square has a regular memory access pattern while BC is highly irregular. Nevertheless, *CAQS* captures their inter-kernel reuse patterns, reduces NUMA effects, and significantly improves their performance. Interestingly, Babelstream demonstrates *CAQS*'s ability to balance locality and synchronization Although Babelstream has substantial inter-kernel reuse, it only achieves a modest 14% L2 hit rate improvement with *CAQS* since its working set exceeds the L2 cache capacity, leading to cache thrashing. Despite this, *CAQS* improves Babelstream's performance by 90% over *Baseline* because *CAQS* keeps the accesses for the L2 misses local on the same chiplet.

However, not all workloads with inter-kernel reuse opportunities obtain significant benefits with *CAQS*. For example, FW's high, irregular memory parallelism hides memory latency through parallelism, rendering L2 hit rate improvements less impactful on overall performance. Thus, it obtains limited performance gains despite significant L2 hit rate improvements (Figure 5.2).

Applications with Limited Inter-Kernel Reuse: Unsurprisingly, applications (e.g., BTREE, DWT2D, Pathfinder, and SRAD_v2) with little or no inter-kernel reuse opportunities often obtain little speedup from *CAQS*. Accordingly, like *Baseline*, *CAQS* schedules the kernels in a RR fashion across the chiplets (the final else path in Algorithm 2), resulting in remote accesses similar to the *Baseline* to access pages first touched on another chiplet. However, even in these situation *CAQS* provides roughly the same performance as *Baseline*. DWT2D is the only workload where *CAQS* incurs a slight performance drop (3%). DWT2D has no inter-kernel reuse and minimal intra-kernel reuse ($\approx$6%). Interestingly, in DWT2D *CAQS*'s avoiding implicit synchronization hurts performance. Unlike *Baseline*, *CAQS* uses CPElide and retains modified data in the writeback L2 cache across kernel boundaries (Chapter 2.1.2). When a new kernel is scheduled, the chiplet must write back this data to memory, adding significant writeback traffic. This writeback traffic falls on a subsequent kernel's critical path (Chapter 3.4), unlike *Baseline*, which concurrently flushes and invalidates data at kernel boundaries. Thus, flushing or invalidating data at kernel boundaries could benefit workloads with no inter-kernel reuse. However, other applications with limited inter-kernel reuse like BTREE have high intra-kernel L2 reuse ($\approx$74%). This reduces L2 evictions and writeback traffic, avoiding DWT2D's slowdown.

Energy: While improving the cache hit rate does not always improve performance, it usually reduces energy. CAQS not only reduces the remote accesses (like prior work), it also reduces the energy consumed by L2 misses that go to the L3 cache or memory.

Overall, Figure 5.4 shows that *CAQS* reduces geomean energy consumption by 36% over *Baseline*. A significant portion of these savings stems from reductions in L2 to L3 requests, memory accesses, and remote traffic. As expected, the energy consumption for L1 to L2 accesses remains the same as the *Baseline*.

However, the energy reduction benefits vary with workload characteristics such as data size and inter-kernel reuse. For instance, Babelstream, which generates high writeback traffic but reduces remote traffic, obtains significant energy savings. Similarly, workloads like BC, which exhibit zero intra-kernel reuse but high inter-kernel reuse, experience an 85% reduction in energy consumption with *CAQS* compared to the *Baseline*. Moreover, *CAQS* significantly reduces energy even for workloads like FW and SRAD_v2, which did not obtain speedups. However, for workloads with low inter-kernel reuse, such as DWT2D and BTREE, *CAQS* and *Baseline* consume similar energy because *CAQS* resort to *Baseline*'s RR scheduling in the absence of reuse.

5.1.2 CAQS vs *LADM* vs *CPElide*

LADM reduces remote accesses, while *CPElide* minimizes implicit invalidations and flushes. Thus, both also improve performance, energy, and network traffic compared to *Baseline*. However, *CAQS* achieves both by retaining data in the cache and optimally scheduling kernels. As a result, *CAQS* improves geomean speedup (6%, 28%), average L2 Hit rate (26%, 25%), and reduces geomean energy consumption (27%, 19%), geomean network traffic (80%, 61%) over *LADM* and *CPElide*, respectively. Thus, *CAQS* provides the benefits of both of these prior approaches and reduces NUMA effects in multi-chiplet GPUs.

Applications with Inter-Kernel Reuse: Figure 5.1 shows that *CAQS* always equals or outperforms *CPElide* and outperforms *LADM* for all but 3 workloads. As expected, *CAQS*'s improvements stem from: (1) enhanced L2 reuse (as shown in Figure 5.2), which both *CPElide* cannot fully capitalize on and *LADM* does not provide, and (2) reducing remote accesses versus *CPElide* (Figure 5.3). As discussed in Chapter 4.4, in *LADM* we optimistically bind the stream to a specific chiplet, eliminating remote accesses. However, despite this optimal configuration, *CAQS* still outperforms *LADM* by effectively exploiting cache reuse.

Although *CAQS* effectively balances locality and implicit synchronization overheads in most workloads with inter-kernel reuse, *LADM* outperforms *CAQS* for FW (10%), Babelstream (3%), and LULESH (13%). For FW and Babelstream, like Babelstream in Chapter 5.1.1, *CAQS* 's delayed writebacks of dirty data increases the critical path of subsequent kernels (versus *LADM* implicitly synchronizing at the kernel boundaries). Moreover, FW's high memory parallelism reduces its sensitivity to L2 cache misses, limiting *CAQS*'s overall impact. Conversely, LULESH has low, irregular inter-kernel reuse. As a result, *CAQS* often resorts to the default RR scheduler. However, *CAQS* obtains some reuse for a few kernels (9% improvement in the L2 hit rate versus *LADM* and *CPElide*). Despite this improvement, the combination of this behavior with the first touch page policy results in remote accesses, which *LADM* avoids in its best-case scenario. Thus, *LADM* outperforms *CAQS* for LULESH. Nevertheless, in aggregate *CAQS* significantly reduces the impact of NUMA effects compared to *CPElide* and *LADM*.

Conversely, *CPElide*'s NUMA-unawareness hurts it in several scenarios. In our 6-chiplet system, *CPElide*'s RR scheduler takes 5 more kernels to return to the same chiplet. As a result, cache invalidations are likely to be triggered by one of the intervening kernels, reducing data reuse opportunities. Thus, *CPElide* only outperforms *LADM* in terms of L2 hit rate and network traffic (but not performance) for Square. Without considering locality – like *CAQS* – *CPElide* often cannot fully harness L2 cache reuse, and thus suffers from additional NUMA effects.

<u>*Applications with Limited Inter-Kernel Reuse*</u>: For the workloads (e.g., BTREE, DWT2D, Pathfinder, and SRAD_v2) with little or no inter-kernel reuse opportunities, *CPElide*, *LADM*, and *CAQS* all provide relatively similar speedups, L2 hit rates, and network traffic. For BTREE, since *CAQS* resorts to a RR scheduler in the absence of inter-kernel reuse, it has more remote accesses than *LADM* (Chapter 5.1.1). However, these remote accesses are <1% of total requests; hence they have minimal impact. For DWT2D, like

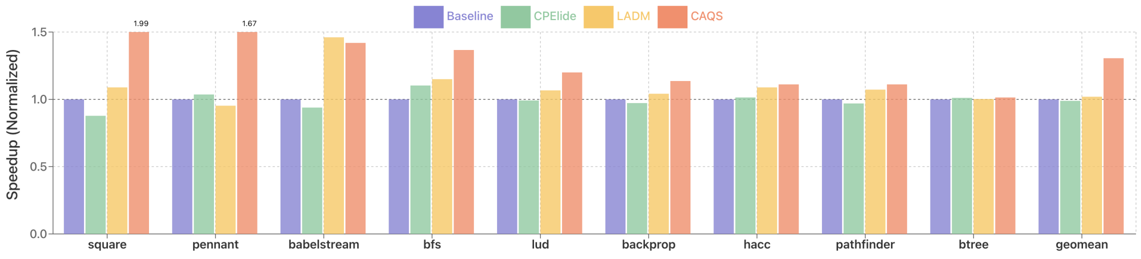


Figure 5.5: Four stream apps' speedup for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

CAQS, *CPElide* and *LADM* are also worse than *Baseline* (Chapter 5.1.1). For *CPElide*, the drop in performance comes from the same reason as *CAQS* – high writeback traffic. *LADM* reduces DWT2D's remote accesses, but the benefits are not enough to overcome the overhead of implicit synchronization on the same chiplet.

Energy: Figure 5.4's results confirm the speedup, L2 hit rate, and network traffic trends discussed above. Overall, *CAQS* significantly reduces energy consumption (Figure 5.4) over both *LADM* and *CPElide*. *CAQS* consumes less energy than *CPElide* by lowering remote accesses and improving the L2 hit rate. Compared to the best-case scenario of *LADM*, *CAQS* does not always reduce remote accesses, but it still leads to energy savings for workloads with high inter-kernel reuse due to its improved L2 hit rate. Interestingly, even workloads where *LADM* provides higher speedups than *CAQS* (e.g., FW, LULESH), *CAQS* still reduces energy versus *LADM* due to improved L2 reuse and fewer memory accesses in *CAQS*.

As before, there are a small number of outliers to this overall trend. *CAQS* consumes more energy than *CPElide* for Babelstream, due to its high write-back traffic (Chapter 5.1.1), which *CPElide* avoids due to its round robin queue scheduling across chiplets. Similarly, *CAQS* consumes more energy than *LADM* for DWT2D. Here, DWT2D's low inter-kernel reuse increases remote accesses in *CAQS* (Chapter 5.1.1) versus *LADM*, increasing energy.

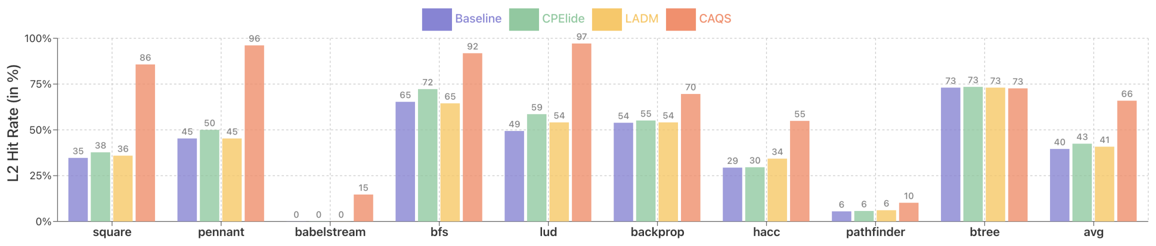


Figure 5.6: Four stream apps' L2 hit rates for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

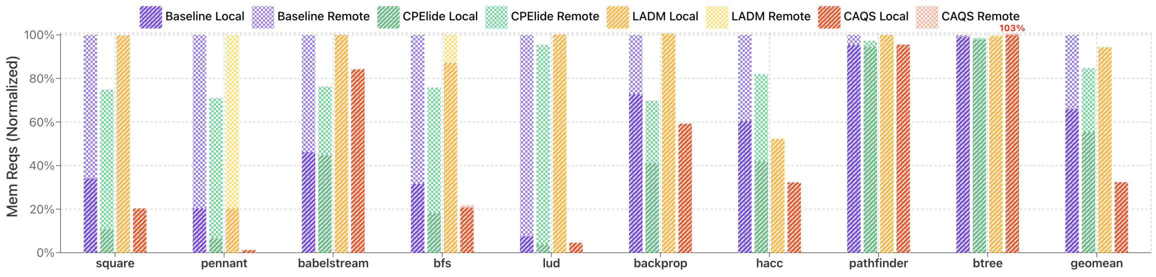


Figure 5.7: Four stream apps' network traffic for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

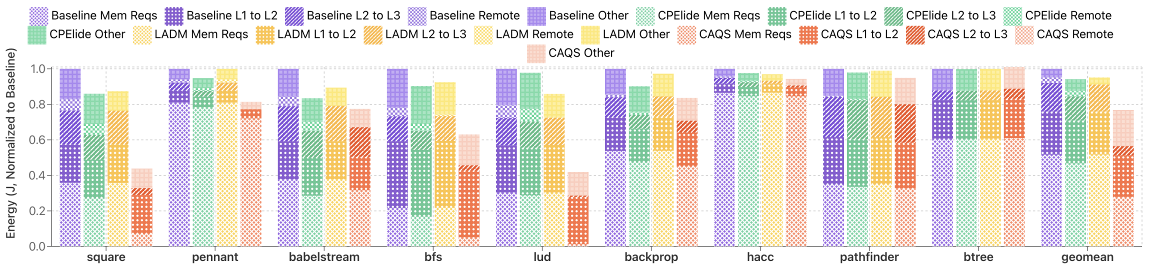


Figure 5.8: Four stream apps' energy usage for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

5.2 4-Stream Workloads

Multi-stream workloads increase shared resource contention in chiplet-based GPUs, including the L3 cache, main memory, and interconnect. With multiple streams active, implicit synchronization overheads also increase since every kernel from every stream will flush the entire system. Nevertheless, Figures 5.5-5.8 show *CAQS* meets this challenge: with 4 concurrent streams *CAQS* provides even better performance gains over *Baseline*, *LADM*, and *CPElide*. Overall, *CAQS* improves geomean performance (30%, 31%, 29%), average

L2 hits (26%, 23%, 25%), reduces energy consumption (24%, 18%, 19%), and decreases network traffic (68%, 62%, 66%) over *Baseline*, *CPElide*, and *LADM*, respectively.

Applications with Inter-Kernel Reuse: Most of these workloads obtain substantial increases in L2 hit rate with *CAQS*. *CAQS* keeps the data in each chiplet's L2 caches and schedules subsequent kernels from the same stream on the same chiplet when the *CCT* identifies reuse opportunities. Thus, the L2 hit rate, energy, and network traffic trends are similar to the single stream trends (Chapter 5.1). However, the increased network congestion due to writeback traffic when 4 concurrent streams are active impacts some application's behavior relative to a single stream (Chapter 5.1). While *Baseline*'s flushes increase, the writebacks in *CAQS* are on the critical path of kernel execution and have a more significant negative impact on performance – especially for Babelstream and Square. For Babelstream, the increased congestion, combined with its large working set and L2 cache thrashing (Chapter 5.1), reduce *CAQS*'s benefit over *Baseline* from 90% with one stream to 42% with four streams. Conversely, Square's L2 hit rate is much higher, resulting in significantly lower writeback traffic and thus fewer NUMA effects.

Interestingly, *LADM* and *CPElide* suffer even more than *CAQS* with four streams. *CPElide*'s NUMA-unaware, RR queue scheduling leads to high writeback traffic with four streams and limited L2 hit rate improvements. Conversely, *LADM* is able to preserve locality, significantly reducing remote accesses. However, because it is unable to exploit inter-kernel L2 reuse, it also increases congestion during flushes. Accordingly, its relative speedup versus *Baseline* drops from 23% for a single stream to 2% for 4 streams. Thus, *CAQS*'s ability to avoid both remote accesses and implicit synchronization make it much better at handling NUMA effects.

Applications with Limited Inter-Kernel Reuse: Since these workloads have little or no reuse opportunities, they are relatively unaffected by the increased number of streams. For example, just like with one stream (Chapter 5.1), workloads like BTREE obtain similar performance with *Baseline*, *CPElide*, *LADM*, and *CAQS*.

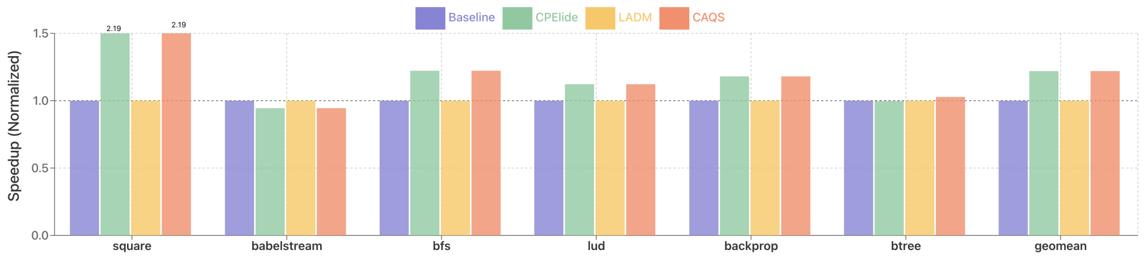


Figure 5.9: Six stream apps' speedup for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

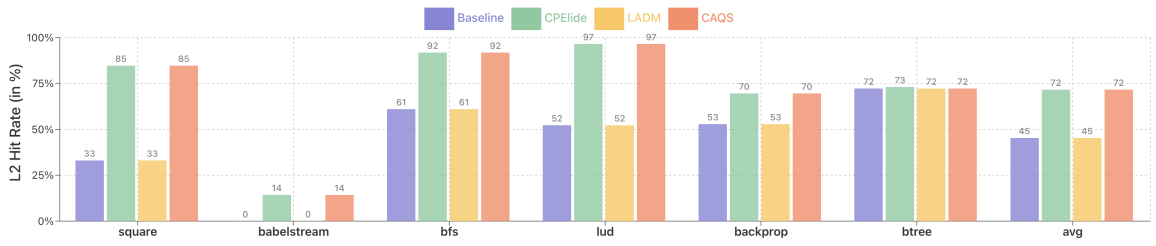


Figure 5.10: Six stream apps' L2 hit rates for *Baseline*, *CPElide*, *LADM*, & *CAQS* in a 6-chiplet GPU, normalized to *Baseline*.

5.3 Saturating the System: 6-Stream Workloads

Finally, Figures 5.9 and 5.10 examines the behavior of our configurations for workloads with 6 concurrent streams, which fully utilize our 6-chiplet GPU. Six stream applications effectively cause the RR queue scheduler to bind each stream to a specific chiplet in our system (Section 4.2). Consequently, *Baseline* emulates *LADM*'s behavior, while *CPElide* emulates *CAQS*'s functionality. Overall, both *CAQS* and *CPElide* provide a geomean speedup of 22% and average L2 hit rate improvement of 27% over *LADM* and *Baseline*. Although most workloads exhibit similar patterns to the transition to four streams, unsurprisingly Babelstream experiences a further performance drop in *CAQS* due to writebacks (Chapter 5.1, 5.2). However, at full utilization contention for shared resources reduces *CAQS*'s overall geomean speedup from 29% (4-streams) to 22% (6-streams). Nevertheless, *CAQS*'s increased L2 reuse and ability to avoid implicit synchronization overhead continues to enable it to equal or outperform all other approaches.

Chapter 6

Discussion

Page Placement Policies: We evaluate CAQS with a First Touch page placement policy (Chapter 4.5). However, as discussed in Chapter 1, there are a number of alternatives including Feng [99], LADM, and GRIT [29]. Unfortunately, this prior work has demonstrated how no single page placement policy is universally best for multi-chiplet GPU workloads [99, 29]. In particular, workloads with low inter-kernel reuse or sparse access patterns sometimes prefer alternate page placement policies. Although this reduces the number of NUMA accesses, they focus on page placement, whereas CAQS focuses on inter-kernel reuse (e.g., at the L2). Thus, these page placement policies are orthogonal to CAQS and could be applied on top of CAQS to further reduce NUMA penalties.

Scheduling Kernels Across Chiplets: In this work, we schedule a kernel on a single chiplet. Although this approach is common in modern multi-chiplet GPUs, it can limit performance for workloads (e.g., Babelstream) with large working sets. As discussed in Chapters 5.1.1 and 5.2, even when these workloads obtain significant inter-kernel reuse, they also suffer from thrashing at the L2 – lowering L2 hit rates and causing bursty write-back traffic. Splitting a kernel across chiplets can reduce the stress on L2 and alleviate the impacts of L2 thrashing, leading to further performance improvements. However, splitting a kernel requires finer grained data structure tracking. A given WG often accesses a smaller portion of each data structure. Accordingly, CAQS must track these accesses at the same granularity to avoid additional, redundant flushes and invalidations, which would decrease reuse. Unfortunately, this would also increase the area overhead at the global CP.

Chiplet-based GPUs versus Multi-GPU Designs: As mentioned in Chapter 2, in this work we focus on chiplet-based GPUs of multi-GPU systems each with multiple chiplets because queue schedulers typically only schedule work for the specific multi-chiplet GPU they receive work for. Moreover, prior work (and our results in Chapter 5) highlights how there are significant opportunities for reuse within a single multi-chiplet GPU. Thus, by improving a multi-chiplet GPU versus state-of-the-art solutions like CPElide and LADM, CAQS can also potentially improve the efficiency of multi-GPU systems.

CAQS Applicability to Other Accelerators: Although we evaluate CAQS on AMD multi-chiplet GPUs, CAQS can also be applied to other vendor's multi-chiplet GPUs, which also suffer from NUMA effects and interface with CPs. More broadly, other accelerators are also being split across multiple chiplets. Although kernels are a GPU-specific way to partition work, other accelerators also partition work into multiple phases and iterate through the phases. Moreover, many accelerators [43, 42, 44, 100, 45, 9] also use embedded microprocessors (like CPs) as an interface and work scheduler. Thus, CAQS works for a wide range of accelerators.

Chapter 7

Related Work

Since CPElide and LADM are most closely related to CAQS, we quantitatively compared against them in Chapter 5 and discussed them and other closely related work in Chapter 1. Here we qualitatively discuss other related approaches in the context of Table 1.1.

Reducing Chiplet-based GPU NUMA Penalties: As discussed in Chapter 1, other prior work on multi-chiplet GPUs have examined alternative approaches to reducing NUMA penalties through more intelligent data locality and placement [28, 30, 101, 46, 31]. For example, CARVE extends the GPU cache capacity to improve NUMA GPU performance [46]. Similar to LADM, other work optimized WG scheduling and/or placement algorithms [15, 28]. However, LADM outperforms them and they do not target implicit synchronization. Thus, since CAQS outperforms LADM (Chapter 5), CAQS should also outperform them. Conversely, SAC dynamically reduces NUMA penalties in multi-chiplet GPUs by improving the cache bandwidth [31]. However, SAC runs at the LLC (e.g., L3) level and does not impact per-chiplet L2 caches. Thus, CAQS is less affected by LLC caching changes. However, SAC or its ancestor SelRep [32] could be combined with CAQS to improve LLC bandwidth.

Other Optimizations for Chiplet-Based Accelerators: Other recent work has improved chiplet-based GPU cache coherence [102, 39], compression [103], design [15, 21], domain specialization [104, 22], and memory management [105, 106, 107, 108, 109]. However, these designs are largely orthogonal to CAQS.

Monolithic GPU Queue/WF Schedulers: Other works significantly increase reuse on monolithic GPUs via better wavefront (WF) [110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122] and queue [123, 124, 125, 126, 127, 128, 67, 129] scheduling. However, unlike CAQS, these solutions are focused on monolithic GPUs. Thus, they are unable neither able effectively provide inter-kernel reuse per chiplet nor are they able to manage the overheads of implicit synchronization – since those overheads are specific to chiplet-based GPUs, these approaches did not face them. Nevertheless, CAQS could be integrated with these WF schedulers to further improve performance.

CPU-based NUMA Approaches: Multi-chiplet GPUs are not the first device to face challenges from NUMA effects. For example, multi-core CPUs have a rich history of reducing the impact of NUMA effects, including Beckmann & Wood [130], D-NUCA [131], NuRapid [132], TD-NUCA [133], and TLC [134]. Similarly, CPU OS's consider waking up a thread on a core that they believe can provide cache benefits or utilize thread affinity to bind a thread to a core [34, 35]. While CAQS leverages some similar concepts to these works, such as load balancing and avoiding binding a stream to a chiplet [33], they require different support. These approaches work well in monolithic, multi-core CPUs, but are less effective in multi-chiplet GPUs which do not possess complex coherence protocols to improve locality and avoid synchronization overheads or OS support to identify where to migrate the threads to. Moreover, they were designed assuming inter-phase synchronization is relatively cheap – which is not the case in accelerators like GPUs. Thus, they cannot fully preserve inter-kernel reuse like CAQS. To preserve reuse in a chiplet-based GPU, they would need run-time scheduling information, which CAQS leverages via the CP.

Chapter 8

Conclusion

The increasing preponderance of multi-chiplet GPUs offers improved yield and continued performance scaling. However, the additional level of indirection chiplets incurs also introduces significant challenges, especially NUMA latencies across chiplets, which are difficult for GPUs to overcome. Recent work has reduced its overhead but has been constrained to levels below the L2 cache due to the implicit synchronization mechanism of GPUs. However, emerging techniques such as CPElide have opened up new possibilities by enabling data to be retained in the L2 cache beyond the execution of a single kernel. In this work, we present CAQS. CAQS's co-designed approach is the first GPU queue scheduler to combine both locality and synchronization information to reduce the impact of NUMA overheads in multi-chiplet GPUs. Consequently, CAQS provides substantial improvements over *Baseline* and the state-of-the-art *CPElide* and *LADM* approaches for geomean performance (30%, 28%, 6%, respectively), L2 cache hits (27%, 25%, 26%), energy efficiency (36%, 19%, 27%), and network traffic reduction(80%, 61%, 80%). Moreover, CAQS's gains increase for increased GPU streams – underscoring the importance of balancing **both** locality and synchronization when designing multi-chiplet GPU schedulers.

References

[1] James Ang et al. “Reimagining Codesign for Advanced Scientific Computing: Report for the ASCR Workshop on Reimagining Codesign”. In: *DOE ASCR Workshop on Reimagining Codesign* (Apr. 2022). DOI: 10.2172/1822199. URL: https://www.osti.gov/biblio/1822199.

[2] Amir Gholami. *Memory Footprint and FLOPs for SOTA Models in CV/NLP/Speech.* ”https://github.com/amirgholami/ai_and_memory_wall”. 2021.

[3] Stephen W. Keckler. *Life After Dennard and How I Learned to Love the Picojoule.* Keynote at MICRO. 2011.

[4] Samuel Naffziger et al. “Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product”. In: *ACM/IEEE 48th Annual International Symposium on Computer Architecture.* ISCA. New York, NY, USA: Association for Computing Machinery, 2021, pp. 57–70. DOI: 10.1109/ISCA52012.2021.00014.

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

[6] Jack Choquette, Olivier Giroux, and Denis Foley. “Volta: Performance and Programmability”. In: *IEEE Micro* 38.2 (2018), pp. 42–52. DOI: 10.1109/MM.2018.022071134.

[7] Norm Jouppi et al. “TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings”. In: *Proceedings of the*

50th Annual International Symposium on Computer Architecture. ISCA. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: `10.1145/3579371.3589350`. URL: `https://doi.org/10.1145/3579371.3589350`.

[8] Norman P. Jouppi et al. "Ten lessons from three generations shaped Google's TPUv4i". In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA. Virtual Event, Spain: IEEE Press, 2021, pp. 1–14. ISBN: 9781450390866. DOI: `10.1109/ISCA52012.2021.00010`. URL: `https://doi.org/10.1109/ISCA52012.2021.00010`.

[9] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: `10.1145/3079856.3080246`. URL: `http://doi.acm.org/10.1145/3079856.3080246`.

[10] Gabriel H. Loh et al. "A Research Retrospective on AMD's Exascale Computing Journey". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: `10.1145/3579371.3589349`. URL: `https://doi.org/10.1145/3579371.3589349`.

[11] Thomas Norrie et al. "The Design Process for Google's Training Chips: TPUv2 and TPUv3". In: *IEEE Micro* 41.2 (2021), pp. 56–63. DOI: `10.1109/MM.2021.3058217`.

[12] NVIDIA. "NVIDIA H100 Tensor Core GPU Architecture". In: *Proceedings of GPU Technology Conference*. GTC. 2022.

[13] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. "Modeling deep learning accelerator enabled gpus". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS. IEEE. Piscataway, NJ, USA: IEEE Press, 2019, pp. 79–92.

[14] Alan Smith et al. "Realizing the AMD Exascale Heterogeneous Processor Vision : Industry Product". In: *51st ACM/IEEE Annual International Symposium on Computer Architecture*. ISCA. Piscataway, NJ, USA: IEEE, 2024, pp. 876–889. DOI: 10.1109/ISCA59077.2024.00068. URL: https://doi.org/10.1109/ISCA59077.2024.00068.

[15] Akhil Arunkumar et al. "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA. Toronto, ON, Canada: ACM, 2017, pp. 320–332. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080231. URL: http://doi.acm.org/10.1145/3079856.3080231.

[16] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel H. Loh. "Exploiting Interposer Technologies to Disintegrate and Reintegrate Multicore Processors". In: *IEEE Micro* 36.3 (2016), pp. 84–93. DOI: 10.1109/MM.2016.53.

[17] Mahmoud Khairy et al. "Locality-Centric Data and Threadblock Management for Massive GPUs". In: *53rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Washington, DC, USA: IEEE Computer Society, 2020, pp. 1022–1036. DOI: 10.1109/MICRO50266.2020.00086.

[18] Saptadeep Pal et al. "Architecting Waferscale Processors - A GPU Case Study". In: *25th IEEE International Symposium on High Performance Computer Architecture*. HPCA. Piscataway, NJ, USA: IEEE Press, 2019, pp. 250–263. DOI: 10.1109/HPCA.2019.00042.

[19] Bryan Black. *Chiplets: How to Utilize Them, Some of Their Challenges and What They Can Do*. Annual IMAPS Symposium Keynote. 2020.

[20] Thiruvengadam Vijayaraghavany et al. "Design and Analysis of an APU for Exascale Computing". In: *Proceedings of the IEEE 23rd International Symposium on High Performance Computer Architecture*. HPCA. Piscataway, NJ, USA: IEEE Press, Feb. 2017, pp. 85–96. DOI: 10.1109/HPCA.2017.42.

[21] Akhil Arunkumar et al. “Understanding the Future of Energy Efficiency in Multi-Module GPUs”. In: *25th IEEE International Symposium on High Performance Computer Architecture*. HPCA. Piscataway, NJ, USA: IEEE Press, 2019, pp. 519–532. DOI: `10.1109/HPCA.2019.00063`.

[22] Yakun Sophia Shao et al. “Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 14–27. ISBN: 9781450369381. DOI: `10.1145/3352460.3358302`. URL: `https://doi.org/10.1145/3352460.3358302`.

[23] Natalie Enright Jerger et al. “NoC Architectures for Silicon Interposer Systems: Why Pay for more Wires when you Can Get them (from your interposer) for Free?” In: *47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Piscataway, NJ, USA: IEEE, Dec. 2014, pp. 458–470. DOI: `10.1109/MICRO.2014.61`.

[24] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel H Loh. “Enabling Interposer-based Disintegration of Multi-core Processors”. In: *48th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. IEEE Press. Piscataway, NJ, USA, 2015, pp. 546–558.

[25] Gabriel H. Loh et al. “Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems”. In: *Proceedings of the 2015 International Symposium on Memory Systems*. MEMSYS. Washington DC, DC, USA: ACM, 2015, pp. 3–10. ISBN: 978-1-4503-3604-8. DOI: `10.1145/2818950.2818951`. URL: `http://doi.acm.org/10.1145/2818950.2818951`.

[26] Skyler J Saleh et al. “GPU Chiplets Using High Bandwidth Crosslinks”. US20200409859A1. Dec. 2020.

[27] Preyesh Dalmia, Rajesh Shashi Kumar, and Matthew D. Sinclair. “CPElide: Efficient Multi-Chiplet GPU Implicit Synchronization”. In: *Proceedings of 57th IEEE/ACM*

International Symposium on Microarchitecture. MICRO. Los Alamitos, CA, USA: IEEE Computer Society, 2024.

[28] Hyojong Kim et al. “CODA: Enabling Co-Location of Computation and Data for Multiple GPU Systems”. In: *ACM Trans. Archit. Code Optim.* 15.3 (Sept. 2018). ISSN: 1544-3566. DOI: 10.1145/3232521. URL: https://doi.org/10.1145/3232521.

[29] Yueqi Wang et al. “GRIT: Enhancing Multi-GPU Performance with Fine-Grained Dynamic Page Placement”. In: *IEEE International Symposium on High-Performance Computer Architecture*. HPCA. Washington, DC, USA: IEEE Computer Society, 2024, pp. 1080–1094. DOI: 10.1109/HPCA57654.2024.00085.

[30] Ugljesa Milic et al. “Beyond the Socket: NUMA-aware GPUs”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Cambridge, Massachusetts: ACM, 2017, pp. 123–135. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3124534. URL: http://doi.acm.org/10.1145/3123939.3124534.

[31] Shiqing Zhang et al. “SAC: Sharing-Aware Caching in Multi-Chip GPUs”. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589078. URL: https://doi.org/10.1145/3579371.3589078.

[32] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. “Selective Replication in Memory-Side GPU Caches”. In: *53rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 967–980. DOI: 10.1109/MICRO50266.2020.00082.

[33] Josep Torrellas, Andrew Tucker, and Anoop Gupta. “Benefits of Cache-affinity Scheduling in Shared-memory Multiprocessors: a Summary”. In: *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer*

Systems. SIGMETRICS. Santa Clara, California, USA: Association for Computing Machinery, 1993, pp. 272–274. ISBN: 0897915801. DOI: 10.1145/166955.167038. URL: https://doi.org/10.1145/166955.167038.

[34] Qiuming Luo et al. "Characteristic Analysis of Operating Systems for Large Scale Hierarchical NUMA System". In: *Sixth International Symposium on Parallel Architectures, Algorithms and Programming*. SPAA. New York, NY, USA: Association for Computing Machinery, 2014, pp. 114–117. DOI: 10.1109/PAAP.2014.55.

[35] Ben Verghese et al. "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers". In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1996, pp. 279–289. ISBN: 0897917677. DOI: 10.1145/237090.237205. URL: https://doi.org/10.1145/237090.237205.

[36] Johnathan Alsop et al. "Lazy release consistency for GPUs". In: *49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Los Alamitos, CA, USA: IEEE Computer Society, 2016, 26:1–26:13. DOI: 10.1109/MICRO.2016.7783729. URL: https://doi.org/10.1109/MICRO.2016.7783729.

[37] B.A. Hechtman et al. "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs". In: *20th International Symposium on High Performance Computer Architecture*. HPCA. Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2014, pp. 189–200. DOI: 10.1109/HPCA.2014.6835930.

[38] Konstantinos Koukos et al. "Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead". In: *ACM Trans. Archit. Code Optim.* 13.1 (Mar. 2016), 1:1–1:22. ISSN: 1544-3566. DOI: 10.1145/2889488. URL: http://doi.acm.org/10.1145/2889488.

[39] Xiaowei Ren et al. "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems". In: *26th IEEE International Symposium on*

High Performance Computer Architecture. HPCA. Washington, DC, USA: IEEE Computer Society, 2020, pp. 582–595. DOI: `10.1109/HPCA47549.2020.00054`.

[40] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models". In: *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. New York, NY, USA: Association for Computing Machinery, Dec. 2015, pp. 647–659.

[41] Inderpreet Singh et al. "Cache Coherence for GPU Architectures". In: *19th International Symposium on High Performance Computer Architecture*. HPCA. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 578–590. DOI: `http://doi.ieeecomputersociety.org/10.1109/HPCA.2013.6522351`.

[42] Renée St. Amant et al. "General-Purpose Code Acceleration with Limited-Precision Analog Computation". In: *ACM/IEEE 41st International Symposium on Computer Architecture*. ISCA. Piscataway, NJ, USA: IEEE, 2014, pp. 505–516. DOI: `10.1109/ISCA.2014.6853213`.

[43] Apple. *Dispatch*. `https://developer.apple.com/documentation/dispatch`. 2024.

[44] Lucian Codrescu et al. "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications". In: *IEEE Micro* 34.2 (2014), pp. 34–43. DOI: `10.1109/MM.2014.12`.

[45] HSA Foundation. *HSA Platform System Architecture Specification*. `https://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf`. 2021.

[46] Vinson Young et al. "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Washington, DC, USA: IEEE Computer Society, 2018, pp. 339–351. DOI: `10.1109/MICRO.2018.00035`.

[47] Benedict R. Gaster, Derek Hower, and Lee Howes. "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models". In: *ACM Trans. Archit. Code Optim.* 12.1 (Apr. 2015), 7:1–7:26. ISSN: 1544-3566. DOI: 10.1145/2701618. URL: http://doi.acm.org/10.1145/2701618.

[48] Derek R. Hower et al. "Heterogeneous-race-free Memory Models". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. Salt Lake City, Utah, USA: ACM, 2014, pp. 427–440. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541981. URL: http://doi.acm.org/10.1145/2541940.2541981.

[49] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. "A Formal Analysis of the NVIDIA PTX Memory Consistency Model". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 257–270. ISBN: 9781450362405. DOI: 10.1145/3297858.3304043. URL: https://doi.org/10.1145/3297858.3304043.

[50] NVIDIA Corp. *NVIDIA Multi-Instance GPU (MIG)*. https://docs.nvidia.com/cuda/mig/index.html. 2021.

[51] AMD. *AMD MxGPU and VMware*. https://drivers.amd.com/relnotes/amd_mxgpu_deploymentguide_vmware.pdf. 2020.

[52] Rolf Neugebauer et al. "Understanding PCIe Performance for End Host Networking". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 327–341. ISBN: 9781450355674. DOI: 10.1145/3230543.3230560. URL: https://doi.org/10.1145/3230543.3230560.

[53] NVIDIA Corp. *NVLink Fabric: A Faster, More Scalable Interconnect*. https://www.nvidia.com/en-us/data-center/nvlink/. 2018.

[54] Dave James. *AMD's answer to Nvidia's NVLink is xGMI, and it's coming to the new 7nm Vega GPU*. Sept. 2018. URL: `https://www.pcgamesn.com/amd-xgmi-vega-20-gpu-nvidia-nvlink`.

[55] AMD. *ROCm: Open Platform For Development, Discovery and Education around GPU Computing*. `https://gpuopen.com/compute-product/rocm/`. 2021.

[56] Sooraj Puthoor et al. "Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture". In: *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. GPGPU. Barcelona, Spain: ACM, 2016, pp. 53–62. ISBN: 978-1-4503-4195-0. DOI: `10.1145/2884045.2884052`. URL: `http://doi.acm.org/10.1145/2884045.2884052`.

[57] Sooraj Puthoor et al. "Oversubscribed Command Queues in GPUs". In: *Proceedings of the 11th Workshop on General Purpose GPUs*. GPGPU. Vienna, Austria: ACM, 2018, pp. 50–60. ISBN: 978-1-4503-5647-3. DOI: `10.1145/3180270.3180271`. URL: `http://doi.acm.org/10.1145/3180270.3180271`.

[58] AMD. *HIP: Heterogeneous-computing Interface for Portability*. `https://github.com/ROCm-Developer-Tools/HIP/`. 2018.

[59] NVIDIA. *NVIDIA, CUDA Stream Management*. 2024. URL: `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__STREAM.html`.

[60] Justin Luitjens. "CUDA Streams: Best Practices and Common Pitfalls". In: *GPU Technology Conference*. GTC. 2014.

[61] Preyesh Dalmia, Rajesh Shashi Kumar, and Matthew D. Sinclair. *Artifact: CPElide: Efficient Multi-Chiplet GPU Implicit Synchronization*. `https://github.com/hal-uw/cpelide-micro24-artifact`. Nov. 2024.

[62] Reese Kuper, Suchita Pati, and Matthew D. Sinclair. "Improving GPU Utilization in ML Workloads Through Finer-Grained Synchronization". In: *3rd Young Architects Workshop*. YArch. Apr. 2021.

[63] Oreste Villa et al. "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 372–383. ISBN: 9781450369381. DOI: 10.1145/3352460.3358307. URL: https://doi.org/10.1145/3352460.3358307.

[64] Michael LeBeane et al. "ComP-Net: Command Processor Networking for Efficient Intra-Kernel Communications on GPUs". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243179. URL: https://doi.org/10.1145/3243176.3243179.

[65] Michael LeBeane et al. "Extended Task Queuing: Active Messages for Heterogeneous Systems". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC. Piscataway, NJ, USA: IEEE Press, 2016, pp. 933–944. DOI: 10.1109/SC.2016.79.

[66] Vinay Ramakrishnaiah et al. "Cache Cohort GPU Scheduling". In: *Proceedings of the 16th Workshop on General Purpose Processing Using GPU*. GPGPU. Edinburgh, United Kingdom: Association for Computing Machinery, 2024, pp. 19–25. ISBN: 9798400718175. DOI: 10.1145/3649411.3649415. URL: https://doi.org/10.1145/3649411.3649415.

[67] Tsung Tai Yeh et al. "Deadline-Aware Offloading for High-Throughput Accelerators". In: *27th IEEE International Symposium on High Performance Computer Architecture*. HPCA. Washington, DC, USA: IEEE Computer Society, 2021, pp. 479–492. DOI: 10.1109/HPCA51647.2021.00048.

[68] Bobbi W. Yogatama, Matthew D. Sinclair, and Michael M. Swift. "Enabling Multi-GPU Support in gem5". In: *3rd gem5 Users' Workshop*. June 2020.

[69] Yuhui Bao et al. "NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs". In: *Proceedings of the International Conference on Parallel Architectures*

and Compilation Techniques. PACT '22. Chicago, Illinois: Association for Computing Machinery, 2023, pp. 333–345. ISBN: 9781450398688. DOI: 10.1145/3559009.3569666. URL: https://doi.org/10.1145/3559009.3569666.

[70] Mahmoud Khairy et al. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling". In: *ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA. Piscataway, NJ, USA: IEEE Press, 2020, pp. 473–486. DOI: 10.1109/ISCA45697.2020.00047.

[71] Charles Jamieson et al. "GAP: gem5 GPU Accuracy Profiler". In: *4th gem5 Users' Workshop*. June 2022.

[72] Nathan Binkert et al. "The gem5 simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[73] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: 2007.03152 [cs.AR].

[74] Preyesh Dalmia, Rohan Mahapatra, and Matthew D. Sinclair. "Only Buffer When You Need To: Reducing On-Chip Memory Traffic Using Local Atomic Buffers on GPUs". In: *28th IEEE International Symposium on High-Performance Computer Architecture*. HPCA. Washington, DC, USA: IEEE Computer Society, 2022, pp. 676–691.

[75] William J. Dally. *Hardware for Deep Learning*. SysML Keynote. Feb. 2018.

[76] Song Han et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA. Seoul, Republic of Korea: IEEE Press, 2016, pp. 243–254. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.30. URL: https://doi.org/10.1109/ISCA.2016.30.

[77] Mike O'Connor et al. "Fine-grained DRAM: Energy-efficient DRAM for Extreme Bandwidth Systems". In: *Proceedings of the 50th Annual IEEE/ACM International*

Symposium on Microarchitecture. MICRO. ACM. New York, NY, USA: ACM, 2017, pp. 41–54.

[78] NVIDIA. "NVIDIA RISC-V Story". In: *4th RISC-V Workshop* (2016). URL: `https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf`.

[79] Jagadish B. Kotra et al. "Increasing GPU Translation Reach by Leveraging Under-Utilized On-Chip Resources". In: *54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1169–1181. ISBN: 9781450385572. DOI: `10.1145/3466752.3480105`. URL: `https://doi.org/10.1145/3466752.3480105`.

[80] Anthony Gutierrez et al. "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level". In: *IEEE International Symposium on High Performance Computer Architecture*. HPCA. Washington, DC, USA: IEEE Computer Society, 2018, pp. 608–619.

[81] Bobby R. Bruce et al. "Enabling Reproducible and Agile Full-System Simulation". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 183–193.

[82] AMD. *HIP-Examples*. `https://github.com/ROCm-Developer-Tools/HIP-Examples`. 2023.

[83] Tom Deakin et al. "GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models". In: *High Performance Computing*. Ed. by Michela Taufer, Bernd Mohr, and Julian M. Kunkel. Cham: Springer International Publishing, 2016, pp. 489–507. ISBN: 978-3-319-46079-6.

[84] Tom Deakin et al. "Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream". In: *Int. J. Comput. Sci. Eng.* 17.3 (Jan. 2018), pp. 247–262. ISSN: 1742-7185.

[85] Shuai Che et al. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *IEEE International Symposium on Workload Characterization.* IISWC. Washington, DC, USA: IEEE Computer Society, Oct. 2009, pp. 44–54. DOI: `10.1109/IISWC.2009.5306797`.

[86] Shuai Che et al. "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads". In: *IEEE International Symposium on Workload Characterization.* IISWC. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. DOI: `10.1109/IISWC.2010.5650274`.

[87] Shuai Che et al. "Pannotia: Understanding Irregular GPGPU Graph Applications". In: *IEEE International Symposium on Workload Characterization.* IISWC. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2013, pp. 185–195. DOI: `10.1109/IISWC.2013.6704684`.

[88] Lawrence Livermore National Labs. *CORAL-2 Benchmarks.* `https://asc.llnl.gov/coral-2-benchmarks`. 2020.

[89] R. D. Hornung, Jeff A. Keasler, and M. B. Gokhale. *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Tech. rep. LLNL-TR-490254. Livermore, CA: Lawrence Livermore National Laboratory, 2011, pp. 1–28.

[90] Johnathan Alsop et al. "Optimizing GPU Cache Policies for MI Workloads". In: *IEEE International Symposium on Workload Characterization.* IISWC. Piscataway, NJ, USA: IEEE Press, 2019.

[91] Bodun Hu and Christopher J. Rossbach. "Altis: Modernizing GPGPU Benchmarks". In: *IEEE International Symposium on Performance Analysis of Systems and Software.* ISPASS. Piscataway, NJ, USA: IEEE Press, 2020, pp. 1–11. DOI: `10.1109/ISPASS48437.2020.00011`.

[92] Joel Hestness, Stephen W. Keckler, and David A. Wood. "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior". In: *IEEE International Symposium on Workload Characterization.* IISWC. Piscataway, NJ, USA: IEEE Press, 2014, pp. 150–160. DOI: `10.1109/IISWC.2014.6983054`.

[93] Mahmoud Khairy, Mohamed Zahran, and Amr Wassal. "SACAT: Streaming-Aware Conflict-Avoiding Thrashing-Resistant GPGPU Cache Management Scheme". In: *IEEE Transactions on Parallel and Distributed Systems* 28.6 (2017), pp. 1740–1753. DOI: `10.1109/TPDS.2016.2627560`.

[94] AMD. *Introducing AMD CDNA™ 2 Architecture.* `https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf`. 2022.

[95] AMD. *AMD CDNA™ 3 Architecture.* 2023. URL: `https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf`.

[96] NVIDIA. *NVIDIA Blackwell Architecture Technical Brief.* `https://resources.nvidia.com/en-us-blackwell-architecture/blackwell-architecture-technical-brief`. 2024.

[97] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture.* ISCA. Toronto, ON, Canada: ACM, 2017, pp. 161–174. ISBN: 978-1-4503-4892-8. DOI: `10.1145/3079856.3080206`. URL: `http://doi.acm.org/10.1145/3079856.3080206`.

[98] Liu Ke et al. "RecNMP: Accelerating Personalized Recommendation with near-Memory Processing". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture.* ISCA. Piscataway, NJ, USA: IEEE Press,

2020, pp. 790–803. ISBN: 9781728146614. URL: https://doi.org/10.1109/ISCA45697.2020.00070.

[99] Yuan Feng. “Understanding Scalability of Multi-GPU Systems”. In: *15th Workshop on General Purpose Processing Using GPU*. GPGPU. Montreal, Canada: ACM, Feb. 2023.

[100] Qualcomm. *Qualcomm Hexagon DSP*. https://developer.qualcomm.com/sites/default/files/docs/adreno-gpu/developer-guide/dsp/dsp.html. 2021.

[101] Harini Muthukrishnan et al. “Efficient Multi-GPU Shared Memory via Automatic Optimization of Fine-Grained Transfers”. In: *ACM/IEEE 48th Annual International Symposium on Computer Architecture*. ISCA. Piscataway, NJ, USA: IEEE Press, 2021, pp. 996–1009.

[102] Saiful A. Mojumder et al. “HALCONE : A Hardware-Level Timestamp-based Cache Coherence Scheme for Multi-GPU Systems”. In: *arXiv preprint arXiv:2007.04292* (2020). arXiv: 2007.04292 [cs.AR].

[103] Mohammad Khavari Tavana et al. “Exploiting Adaptive Data Compression to Improve Performance and Energy-Efficiency of Compute Workloads in Multi-GPU Systems”. In: *IEEE International Parallel and Distributed Processing Symposium*. IPDPS. Piscataway, NJ, USA: IEEE Press, 2019, pp. 664–674. DOI: 10.1109/IPDPS.2019.00075.

[104] Yaosheng Fu et al. “GPU Domain Specialization via Composable On-Package Architecture”. In: *ACM Trans. Archit. Code Optim.* TACO 19.1 (Dec. 2021). ISSN: 1544-3566. DOI: 10.1145/3484505. URL: https://doi.org/10.1145/3484505.

[105] Trinayan Baruah et al. “Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems”. In: *26th IEEE International Symposium on High Performance Computer Architecture*. HPCA. Piscataway, NJ, USA: IEEE Press, 2020, pp. 596–609. DOI: 10.1109/HPCA47549.2020.00055.

[106] Trinayan Baruah et al. “Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance”. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. PACT. Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 455–466. ISBN: 9781450380751. DOI: 10.1145/3410463.3414639. URL: https://doi.org/10.1145/3410463.3414639.

[107] Bingyao Li et al. “Improving Address Translation in Multi-GPUs via Sharing and Spilling Aware TLB Design”. In: *54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1154–1168. ISBN: 9781450385572. DOI: 10.1145/3466752.3480083. URL: https://doi.org/10.1145/3466752.3480083.

[108] Jinhui Wei et al. “Dynamic GMMU Bypass for Address Translation in Multi-GPU Systems”. In: *Network and Parallel Computing*. Ed. by Xin He, En Shao, and Guangming Tan. Cham: Springer International Publishing, 2021, pp. 147–158. ISBN: 978-3-030-79478-1.

[109] B. Pratheek, Neha Jawalkar, and Arkaprava Basu. “Designing Virtual Memory System of MCM GPUs”. In: *2022 55th IEEE/ACM International Symposium on Microarchitecture*. MICRO. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2022, pp. 404–422. DOI: 10.1109/MICRO56248.2022.00036. URL: https://doi.ieeecomputersociety.org/10.1109/MICRO56248.2022.00036.

[110] Aamer Jaleel et al. “CRUISE: Cache Replacement and Utility-Aware Scheduling”. In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. London, England, UK: Association for Computing Machinery, 2012, pp. 249–260. ISBN: 9781450307598. DOI: 10.1145/2150976.2151003. URL: https://doi.org/10.1145/2150976.2151003.

[111] Timothy G. Rogers, Mike O’Connor, and Tor M. Aamodt. “Cache-Conscious Wavefront Scheduling”. In: *45th Annual IEEE/ACM International Symposium on Mi-*

croarchitecture. MICRO. Piscataway, NJ, USA: IEEE Press, 2012, pp. 72–83. DOI: 10.1109/MICRO.2012.16.

[112] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. "CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA. Portland, Oregon: Association for Computing Machinery, 2015, pp. 515–527. ISBN: 9781450334020. DOI: 10.1145/2749469.2750418. URL: https://doi.org/10.1145/2749469.2750418.

[113] Wilson W. L. Fung et al. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. New York, NY, USA: ACM, 2007, pp. 407–420. DOI: 10.1109/MICRO.2007.30.

[114] James A. Jablin et al. "Warp-aware Trace Scheduling for GPUs". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT. Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 163–174. ISBN: 9781450328098. DOI: 10.1145/2628071.2628101. URL: https://doi.org/10.1145/2628071.2628101.

[115] Nagesh B. Lakshminarayana and Hyesoon Kim. "Effect of Instruction Fetch and Memory Scheduling on GPU Performance". In: *Workshop on Language, Compiler, and Architecture Support for GPGPU*. 2010.

[116] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. "Divergence-Aware Warp Scheduling". In: *46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Piscataway, NJ, USA: IEEE Press, 2013, pp. 99–110.

[117] Yulong Yu et al. "A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-level Parallelism in GPGPUs". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS. Newport Beach, California, USA:

Association for Computing Machinery, 2015, pp. 15–24. ISBN: 9781450335591. DOI: 10.1145/2751205.2751234. URL: https://doi.org/10.1145/2751205.2751234.

[118] Benjamin Hao and David Pearson. "Instruction Scheduling and Global Register Allocation for SIMD Multiprocessors". In: *2nd Int'l Workshop on Parallel Algorithms for Irregularly Structured Problems*. 1995, pp. 81–86.

[119] Adwait Jog et al. "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 395–406. ISBN: 9781450318709. DOI: 10.1145/2451116.2451158. URL: https://doi.org/10.1145/2451116.2451158.

[120] Veynu Narasiman et al. "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling". In: *44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. Piscataway, NJ, USA: IEEE Press, Dec. 2011, pp. 308–317.

[121] Qiumin Xu and Murali Annavaram. "PATS: Pattern Aware Scheduling and Power Gating for GPGPUs". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT. Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 225–236. ISBN: 9781450328098. DOI: 10.1145/2628071.2628105. URL: https://doi.org/10.1145/2628071.2628105.

[122] J. Liu, J. Yang, and R. Melhem. "SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers". In: *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. New York, NY, USA: Association for Computing Machinery, 2015, pp. 383–394.

[123] Quan Chen et al. "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Pro-*

gramming Languages and Operating Systems. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 17–32. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037700. URL: http://doi.acm.org/10.1145/3037697.3037700.

[124] Quan Chen et al. "Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. Atlanta, Georgia, USA: ACM, 2016, pp. 681–696. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872368. URL: http://doi.acm.org/10.1145/2872362.2872368.

[125] Pin Gao et al. "Low Latency RNN Inference with Cellular Batching". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys. Porto, Portugal: ACM, 2018, 31:1–31:15. ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190541. URL: http://doi.acm.org/10.1145/3190508.3190541.

[126] Connor Holmes et al. "GRNN: Low-Latency and Scalable RNN Inference on GPUs". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys. Dresden, Germany: ACM, 2019, 41:1–41:16. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303949. URL: http://doi.acm.org/10.1145/3302424.3303949.

[127] Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. "GPUSync: A Framework for Real-Time GPU Management". In: *2013 IEEE 34th Real-Time Systems Symposium*. Piscataway, NJ, USA: IEEE Press, Dec. 2013, pp. 33–44. DOI: 10.1109/RTSS.2013.12.

[128] Shinpei Kato et al. "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments". In: *USENIX Annual Technical Conference*. USENIX ATC. Portland, OR: USENIX Association, June 2011. URL: https://www.usenix.org/conference/usenixatc11/timegraph-gpu-scheduling-real-time-multi-tasking-environments.

[129] Jacob T Adriaens et al. “The Case for GPGPU Spatial Multitasking”. In: *18th IEEE International Symposium on High-Performance Computer Architecture*. HPCA. IEEE. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1–12.

[130] Bradford M. Beckmann and David A. Wood. “Managing Wire Delay in Large Chip-Multiprocessor Caches”. In: *37th International Symposium on Microarchitecture*. MICRO. Piscataway, NJ, USA: IEEE Press, 2004, pp. 319–330. DOI: 10.1109/MICRO.2004.21.

[131] Changkyu Kim, Doug Burger, and Stephen W. Keckler. “An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches”. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. San Jose, California: Association for Computing Machinery, 2002, pp. 211–222. ISBN: 1581135742. DOI: 10.1145/605397.605420. URL: https://doi.org/10.1145/605397.605420.

[132] Zeshan Chishti, Michael D. Powell, and T.N. Vijaykumar. “Distance Associativity for High-performance Energy-efficient Non-uniform Cache Architectures”. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. New York, NY, USA: Association for Computing Machinery, 2003, pp. 55–66. DOI: 10.1109/MICRO.2003.1253183.

[133] Paul Caheny et al. “TD-NUCA: Runtime Driven Management of NUCA Caches in Task Dataflow Programming Models”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. SC. Piscataway, NJ, USA: IEEE Press, 2022, pp. 1–15. DOI: 10.1109/SC41404.2022.00085.

[134] Bradford M. Beckmann and David A. Wood. “TLC: Transmission Line Caches”. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. New York, NY, USA: ACM, 2003, pp. 43–54. DOI: 10.1109/MICRO.2003.1253182.