CPElide: Efficient Multi-Chiplet GPU Implicit Synchronization

Preyesh Dalmia
ECE Department
NVIDIA, University of Wisconsin-Madison
Santa Clara, CA, USA
pdalmia@nvidia.com

Rajesh Shashi Kumar
ECE Department
ARM, University of Wisconsin-Madison
Austin, TX, USA
rajesh.shashikumar@arm.com

Matthew D. Sinclair
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI, USA
sinclair@cs.wisc.edu

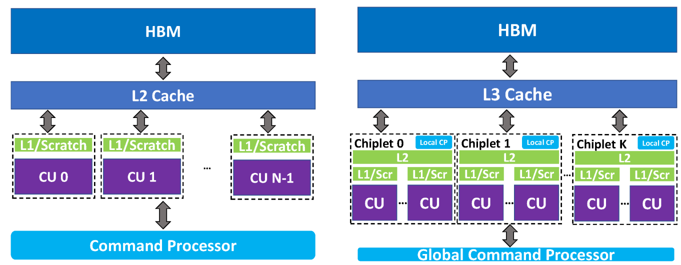
ABSTRACT

Chiplets are transforming computer system designs, allowing system designers to combine heterogeneous computing resources at unprecedented scales. Breaking larger, monolithic chips into smaller, connected chiplets helps performance continue scaling, avoids die size limitations, improves yield, and reduces design and integration costs. However, chiplet-based designs introduce an additional level of hierarchy, which causes indirection and non-uniformity. This clashes with typical heterogeneous systems: unlike CPU-based multi-chiplet systems, heterogeneous systems do not have significant OS support or complex coherence protocols to mitigate the impact of this indirection. Thus, exploiting locality across application phases is harder in multi-chiplet heterogeneous systems. We propose **CPElide**, which utilizes information already available in heterogeneous systems' embedded microprocessor (the command processor) to track inter-chiplet data dependencies and aggressively perform implicit synchronization only when necessary, instead of conservatively like the state-of-the-art HMG. Across 24 workloads CPElide improves average performance (13%, 19%), energy (14%, 11%), and network traffic (14%, 17%), respectively, over current approaches and HMG.

***Index Terms*—GPGPU, Chiplets, Synchronization, Coherence.**

I. INTRODUCTION

Systems ranging from smartphones to supercomputers are embracing heterogeneity to improve efficiency. The underlying technology is also changing. For decades vendors used transistor scaling to fit an order of magnitude more transistors on a die per technology generation. For example, modern GPUs often have hundreds of compute units (CUs), 4× more CUs than the previous decade [64], and run applications with millions or billions of threads. However, continuing to scale performance and energy efficiency for future heterogeneous systems is hampered by technology scaling and Moore's Law slowing [63]. General-purpose CPUs and accelerators also often have different timing, density, and bandwidth requirements. Thus, integrating them on a single die is difficult. Moreover, cost, die, and yield limitations make designing larger, monolithic systems difficult [16], [60], [64], [106].



(a) Monolithic. (b) Proposed chiplet-based.

Fig. 1: Overall heterogeneous system.

Recent research has combined multiple smaller chips into a large, aggregated system, an approach known as multi-chip modules (MCMs) or chiplets [16], [17], [40], [64], [89], [118], [129], as shown in Figure 1. Industry has shown chiplet-based CPUs can continue scaling performance [93]. Since the chiplets are smaller, they avoid the die and yield challenges monolithic systems face. Moreover, integrating multiple chiplets together (e.g., using interposers [56], [59], [60], [83] or other packaging technologies [117]) improves memory bandwidth, memory capacity, and I/O scalability [53], [54], [64], [103], [122], [129]. This enables closer integration of components than was previously possible, without the technology integration challenges monolithic designs faced.

In particular, two key bottlenecks are bandwidth limited inter-chiplet links [116] and more expensive inter-kernel reuse caused by additional cache levels being subject to implicit synchronization [64], [142]. To examine these issues we focus on chiplet-based GPUs [54], [108], [117], [129] because GPUs have become the general-purpose accelerator of choice due to their wide availability and ease of programming. However, the issues also apply to other accelerators [7], [14], [29], [50], [58], [111] (discussed further in Section VI).

As shown in Figure 1b, multi-chiplet GPUs have an additional level of cache. Thus, GPU L2 caches are now shared across CUs within a chiplet, and the L3 cache is a shared LLC across all chiplets. As a result, synchronization operations are even more expensive in multi-chiplet GPUs than monolithic GPUs (discussed further in Section II-C). Although most GPU applications only have implicit, coarse-grained synchronization at kernel boundaries, they still must invalidate all valid data from local caches at kernel launches (an implicit acquire) and write through all dirty data from local caches (an implicit release) at kernel completion to ensure correctness [41], [48], [86]. In monolithic GPUs, this overhead was relatively small because the L2 cache was shared across all CUs, and GPU L1 caches typically use write-through or write-no-allocate policies. However, in chiplet-based GPUs the L3 is the shared ordering point across chiplets. Thus, the per-chiplet L2 caches must also be invalidated and flushed at kernel boundaries.

This increased indirection hurts performance: unlike monolithic GPUs, chiplet-based GPUs cannot exploit inter-kernel locality at the L2. To understand inter-kernel shared L2 reuse's impact, we compared performance of our workloads (Section IV-D) on a 4-chiplet GPU to an equivalent (but infeasible to build) monolithic GPU. Figure 2 shows the loss of inter-kernel shared L2 reuse is significant: 54% on average (similar to prior work that shows 29%-45% average performance loss [116], [142]). Consequently, efficiently moving data is challenging in chiplet-based heterogeneous systems and will become even more acute as systems scale to more chiplets.

To address this inefficiency we propose CPElide. CPElide leverages the key insight that, although current systems do not exploit it (Section II-B), the GPU's Command Processor (CP) *has a global view of what work groups (WGs) are being sent to each chiplet and what data structures each WG accesses at a given time*. Modern, monolithic GPUs often utilize a centralized, integrated programmable processor (the CP) to interface between the programmable accelerator and software (Figure 1a). However, currently these CPs are limited to parsing work contexts and latency-blind scheduling (discussed further in Section II). We propose to redesign the CP to utilize this information, in concert with software information (from the compiler or programmer) to determine when implicit synchronization must be performed. Given this, CPElide generates the appropriate per-chiplet L2 acquire and release operations at kernel launches to ensure data is invalidated and/or flushed right before another chiplet needs it. Effectively, CPElide converts conservative, per-kernel, GPU-wide implicit acquires and releases into aggressive, chiplet-specific, on demand acquires and releases – increasing reuse and reducing the implicit synchronization penalty.

However, modern CPs view the accelerators monolithically, even though accelerators are often distributed across multiple chiplets. Thus, we propose to partition the global, centralized CP's responsibilities between a global CP and local, per-chiplet CPs (Figure 1b). Modern chiplet-based GPUs have a local, per-chiplet CP and elect one of these local CPs to manage work across the GPU [88], but they do not utilize all of their available information to intelligently partition work – which we propose to do. The local CPs have access to dynamic, micro-second scale information about their chiplet, which they communicate to the global CP. Likewise, the global CP has a global view of all CUs, including synthesizing the information from the local CPs, and all work being sent to the GPU. Additionally, since the global CP has access to the GPU's kernel objects' metadata [50], it knows which data structure(s) each kernel accesses, as well as what chiplet(s) each kernel will be assigned to. The global CP also knows what data structure(s) subsequent kernel(s) will access and which chiplet(s) those kernel(s) will be scheduled on. Thus, the global CP has a complete picture of what data may still be in the chiplet's L1 and L2 caches. CPElide uses the global CP's information to track, at a data structure granularity, when and where data structures are being accessed and issue the appropriate per-chiplet synchronization operations.

Prior work examined chiplet-based GPU and multi-GPU (MGPU) coherence. Hierarchical Multi-GPU (HMG) [116] extend existing GPU coherence protocols from monolithic GPUs (Section II-C) for MGPU systems by hierarchically tracking sharers. Similarly, Halcone [91] hierarchically extends timestamp-based monolithic GPU coherence protocols for MGPU systems. However, we show HMG's complexity is unnecessary and sometimes hurts performance (Section V). CPElide also shares some similarities with CPU distributed shared memory approaches, although they either incur significant latency overheads or require additional copies compared to CPElide. We discuss related work further in Section VII.

Overall, across 24 benchmarks from traditional GPGPU, graph analytics, ML, and HPC workloads, on average CPElide improves performance by 13% and 19% (17% and 20% for workloads with moderate or higher inter-kernel reuse), energy by 14% and 11%, and network traffic by 14% and 17%, over the baseline 4-chiplet GPU and the state-of-the-art HMG, respectively. Furthermore, for applications without significant reuse CPElide provides equivalent performance to the baseline, and CPElide's benefits scale with the number of chiplets. To the best of our knowledge, CPElide is the first to leverage CP information to mitigate synchronization overheads in multi-chiplet GPUs. Moreover, CPElide effectively monitors intra- and inter-chiplet behavior without hardware changes.

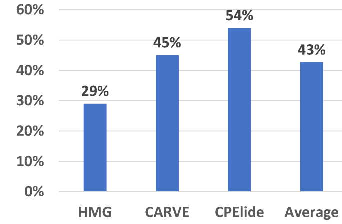


Fig. 2: Average performance lose due to lack of L2 cache inter-kernel reuse in multi-chiplet GPUs versus equivalent monolithic GPU, from both prior work [116], [142] and ours.

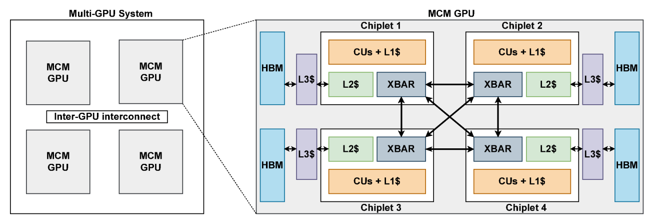
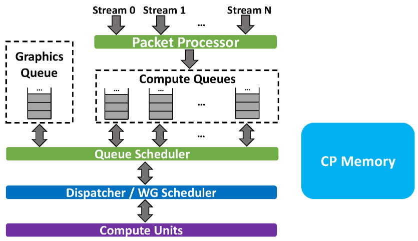
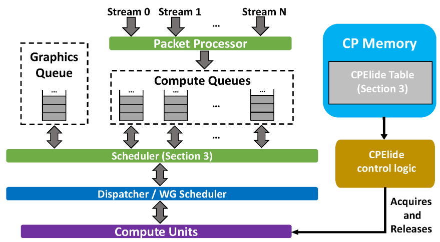


Fig. 3: Current Multi-Chiplet Single GPU ("MCM-GPU") and Multi-GPU (MGPU) Architectures.



(a) Current Command Processor.



(b) Proposed Command Processor.

Fig. 4: Current and proposed CP for chiplet-based systems.

II. Background

A. Multi-Chiplet GPU Architecture

Each GPU in a MGPU system is composed of multiple chiplets (e.g., 4 chiplets in Figure 3). Hence, such a GPU is often called an MCM-GPU. MGPUs connect the MCM-GPUs via an inter-GPU interconnect [53], [103], [122]. MCM-GPU and MGPU designs have several important differences. Since MGPU systems connect multiple MCM-GPUs, if unified virtual memory (UVM) is desired they must keep data coherent and consistent across GPUs. Thus, MGPUs focus on communication across GPUs and introduce an additional level of hierarchy to a variety of components, including cache coherence [91], [116], memory consistency, scheduling [64], and synchronization. While this work is important, we focus on single GPU, multi-chiplet designs (i.e., a MCM-GPU) because significant performance is lost within a MCM-GPU (Figure 2). Moreover, solving the unique challenges MCM-GPUs present to retain more data locally could also potentially benefit MGPUs (Section VI).

Each GPU chiplet in a MCM-GPU has dedicated CUs, each with a private L1 cache, and a shared L2 cache (Figure 1b, Figure 3 breakout). In some setups each L2's banks are coherent within a single chiplet but incoherent with the rest of the system [116], while in others all banks in the system are coherent [117], [129], [141]. Chiplet-based GPUs also introduce an additional level to the memory hierarchy: an LLC which is shared across all chiplets, although LLC banks and the device's HBM (High Bandwidth Memory) are also divided across chiplets. A MCM-GPU's memory subsystem is NUMA and its inter-chiplet links do not provide full aggregated LLC/HBM bandwidth to each chiplet [16]. As a result, accesses to another chiplet's memory incur additional latency. Accordingly, inter-chiplet bandwidth is limited [116] and bulk flush and invalidation operations are expensive. Thus, when accessing data modified by another chiplet current designs either: a) incur additional latency to access a shared cache's remote bank [116] or b) flush dirty data from the producer chiplet, allowing consumer chiplet(s) to subsequently fetch the data from a shared LLC or global memory [117]. Since both approaches incur significant overhead, we investigate alternatives that retain more data in each chiplet's L2 cache.

B. GPU Command Processors

Figure 4a shows a simplified CP [8], [42].[1] For GPUs, the programmable CP interfaces between the software, via the driver and runtime (e.g., AMD's ROCm [11] stack), and the hardware. Since CPs are programmable, vendors can adjust their functionality without changing hardware. Once a user has written their GPU program [9], [69], [101], the underlying GPU driver and runtime create software queues and enqueue the program's GPU kernels, along with any memory management and inter-kernel synchronization, as packet(s).

The CP's *packet processor* maps each packet onto a hardware *compute queue* using its *queue scheduler*. The *queue scheduler* maps t kernels to M CUs while maximizing resource utilization. To meet this goal GPUs support multiple hardware queues [8], [81], [97], [109] to manage independent work submitted asynchronously with GPU streams [9], [85], [99]. Typically each stream is mapped to a queue and each queue holds one or more kernels from that stream. The CP maintains intra-stream, inter-kernel dependencies but often executes different streams concurrently. Within a queue, a queue entry describes a given kernel including thread dimensions, register usage, scratchpad size, and pointers to kernel arguments being accessed. The CP's *WG scheduler* reads these fields to dispatch WGs to CUs. Generally, WG schedulers issue all WGs from a kernel in round-robin fashion across the available CUs [109], [110] before switching to another kernel. Multi-chiplet GPUs have per-chiplet CPs [88] to handle local scheduling decisions like which CU on a chiplet to schedule a given WG and which WF to schedule on a given CU. However, to the best of our knowledge they do not have a global CP.

C. GPU Coherence & Consistency

Most GPU applications are highly parallel, infrequently share data globally, and infrequently synchronize, often only implicitly at the beginning and end of a kernel. Thus monolithic GPUs use simple, software-driven, VI-like coherence protocols where data is either in the Valid (V) or Invalid (I) state [4], [46], [76], [116], [119]–[121]. This coherence protocol invalidates L1 caches on acquires (synchronization reads) while releases (synchronization writes) ensure all previous writes complete. Acquires and releases must also ensure data reaches a memory level shared by all the CUs (often the L2, Figure 1b). Although these synchronizations are heavyweight, since most GPU applications synchronize infrequently the overheads of flushing/invalidating L1 caches are acceptable.

Since synchronization is so expensive, GPU consistency models extend the popular sequentially consistent for data-race-free (SC-for-DRF) CPU consistency model [20], [87]

[1] Without loss of generality, we use AMD terminology when discussing CPs. NVIDIA utilizes embedded RISC cores for similar purposes [1], [98].

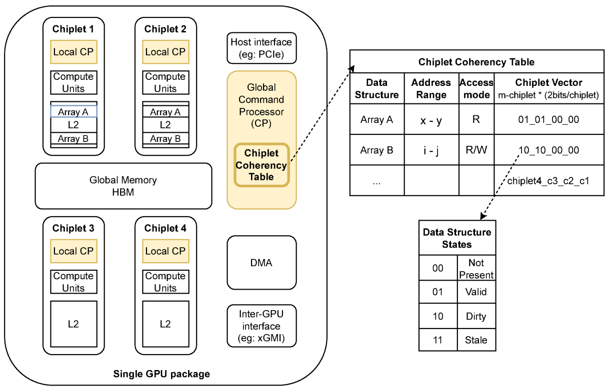


Fig. 5: Proposed CPElide architecture (changes in yellow).

with scopes. Thus, it is called sequentially consistent for heterogeneous-race-free (SC-for-HRF) [3], [41], [48]–[50], [86], [120], [136]. We focus on SC-for-HRF's global scope since implicit kernel boundary synchronization uses it. Globally scoped synchronization is visible to all GPU threads by invalidating all valid data in local caches (e.g., L1s) at acquires and waiting for all writes to complete at releases.

III. Design

A. Proposed CPElide Architecture

Figure 5 shows CPElide's overall design. To track memory accesses from data structures (e.g., arrays) across chiplets and elide implicit synchronization, CPElide implements a *Chiplet Coherence Table* in the global CP's private memory. This involves the GPU's coherence protocol and consistency model (Section II-C). Each table row has 4 fields: data structure (e.g., array base address), address range(s) per chiplet, access mode, and a bit vector indicating which chiplets are accessing this data structure and their access state (Section III-B). Each entry requires 1 byte (chiplet vector), 1 bit (access mode), 28 bytes (address range(s)), and 4 bytes (base address).

Sizing Chiplet Coherence Table: Prior work found most GPU programs access 8 or fewer data structures per kernel and these data structures are reused within 4 kernels [77], [130]. To confirm this we analyzed our application's (Table II) inter-kernel access patterns. Most reused data structures within 1-2 kernels, except the RNNs which occasionally reused data up to 4 kernels later. Moreover, our applications have an average of 4 input arguments per kernel (max 8). Thus, we conservatively sized CPElide to hold 8 unique data structures per kernel for 8 kernels based on our analysis. As a result, CPElide can simultaneously track 64 entries. The total space for a 4-chiplet system is ∼2 KB. However, if access patterns changes in the future, since CPs are programmable data structure tracking can be increased (at the cost of additional CP memory).

Since all work dispatched to the GPU goes through the global CP (Section II-B), the global CP has a global view of what data is being accessed in each chiplet at a given time. The global CP gets this information for the kernel argument via the metadata retrieved from kernel packets. Next, when the global CP dispatches the WGs to chiplets,

```
//Square Kernel with Array A (R) as
//input and Array C (R/W) as output
hipSetAccessMode(square, C_d, 'R/W');
hipSetAccessMode(square, A_d, 'R');
hipLaunchKernelGGL(square,..., C_d, A_d, N);
```

Listing 1: Proposed new API calls to mark array access modes.

```
typedef tuple<Addr_t, Addr_t, LogicalChipletID>
rangeChiplet;
//If kernel to be launched on 2 chiplets
//Each chiplets works on half of input & output
//numSchedChip: # chiplets to schedule kernel on
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: Proposed API calls to mark both access modes and address ranges for data structures. Although the programmer does not know which chiplets the kernel will map to, the number of chiplets it will use is configurable.

it updates the *Chiplet Coherence Table* for each data structure the kernel accesses. However, kernel packets do not always provide all necessary information [9], [101], [124]. Thus, we utilize software information (either from the programmer or the compiler) about access mode and address ranges. Each chiplet a kernel is scheduled on accesses this information (Section III-B). The global CP also uses this information to estimate what data may still be in the chiplet's L1 and L2 caches at the end of a given kernel. Accordingly, instead of performing implicit acquires and releases on all chiplets at each kernel boundary, CPElide uses the *Chiplet Coherence Table's* information to generate the appropriate per-chiplet acquires and releases to invalidate and/or flush per-chiplet L2 data just before another chiplet needs it. However, since CPElide does not modify the coherence protocol, the L1 caches must still be invalidated/flushed at kernel boundaries.

B. Proposed Changes

To perform the operations described in Section III-A, CPElide requires several key changes:I

Command Processor (CP): As discussed in Section II-B, modern GPUs have per-chiplet CPs. However, they do not exploit all of the information available to them. Figure 4b shows our redesigned CP, which adds a global CP, splits the CP's functionality between the local, per chiplet CPs and global CP, and adds new functionality into the local CP per chiplet. In addition to managing local scheduling decisions, our local CPs also pass runtime information back to the global CP. The global CP acts as the interface with the host and dispatches work across chiplets. It also issues CPElide's acquires and releases, and houses CPElide's *Chiplet Coherence Table*. We discuss these changes further in Section III-C.

Labeling Memory Accesses: To identify each global memory data structure, like prior work [5], [27], [92], [119], [120] we label each data structure and their access mode: Read-Only (*R*) or Read/Write (*R/W*). Although monolithic GPUs generally only need *R* and *R/W* labels [76], [119], chiplet-based GPUs must also know **where** these accesses are scheduled. Without scheduling information it is unknown which chiplet(s) have the most up-to-date copy and thus the system must conservatively generate acquires/releases for all chiplets.

There are several ways for the compiler/programmer to pass this information to the CP. Listing 1 shows an example of the new API calls we added to HIP's open source ROCm [11] for this purpose. We use `hipSetDevice` to bind a stream i to chiplet(s) j. Specifically, for each data structure in a given kernel, the programmer uses our new `hipSetAccessMode` call to specify if the data structure will be *R* or *R/W* in that kernel. We extended ROCm to add this information to the kernel packet, allowing the global CP's packet processor to access it. Optionally, programmers can instead use our new `hipSetAccessModeRange` to specify finer-granularity information by providing both access mode and address range(s) within a data structure that chiplet(s) will be operating on (Listing 2). Although the compiler/programmer should be able to determine most GPU access patterns statically (e.g., record and replay [107], virtual ISAs [75], or effect inference [128]), when this is not possible `hipSetAccessMode` should be used. For example, if a kernel accesses different data structures depending on control flow, the software must specify all regions that may be accessed by the kernel. Similar to GPU consistency models, the compiler/programmer must correctly mark the ranges or the outputs may be incorrect.

Tracking Accesses in CP: Figure 5 shows how CPElide uses a $2n$-bit bit-vector (n is the number of chiplets) to track which data structures are accessed, their mode (R, R/W), and by what chiplets. Each table row tracks a data structure, and the columns specify the virtual address ranges for different chiplets, the access mode, and the $2n$-bit bit-vector tracks what state (**States**) the data structure will be in.

Coarsening Data Structure Labels: CPElide tracks up to 8 data structures per kernel (Section III-A). If a kernel accesses more than 8 data structures CPElide coarsens the information for before adding it to the *Chiplet Coherence Table*. To do this we first search the table to find if any data structures are contiguous in memory. If any are found, CPElide combines their entries. The combined entry tracks all chiplets any of these data structures were assigned to, and its data structure identifier in the chiplet vector stores the more conservative of the states to ensure correctness. For example, if one data structure is *R* and the other is R/W, the combined state of the chiplet vector will be R/W. However, if no such structure is found, then we coarsen the data structures closest to one another in memory. Although this may perform more acquire/releases than necessary, since the memory between them is not accessed, it ensures correctness. Finally, while not observed, if parts of a data structure are accessed in different modes and software cannot statically determine their ranges, CPElide creates a chiplet vector per range.

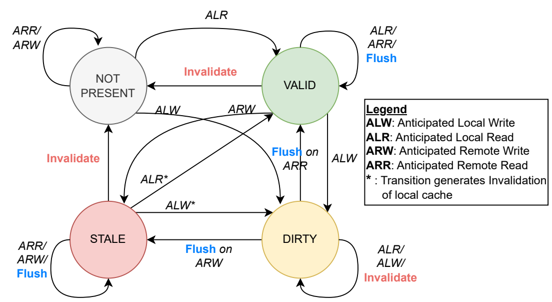


Fig. 6: Internal state tracking mechanism for a data structure in the CPElide Table for a given chiplet.

States: Each *Chiplet Coherence Table* entry has four possible states, represented by 2 bits per chiplet in the chiplet vector:

- *Not Present (00)*: This state indicates that a data structure does not exist in chiplet i's L2 cache.
- *Valid (01)*: After a kernel that only reads (access mode *R*) a data structure, if its data is in chiplet i's L2 cache, it will be *Valid*. Thus, if later chiplet j wants to write this data structure, chiplet i's copy must be invalidated or marked as *Stale*.
- *Dirty (10)*: After a given kernel that reads or writes (access mode R/W) a data structure, if its data remains in chiplet i's L2 cache, its values may be *Dirty*. Thus, if later chiplet j wants to access this data, we must first flush it from chiplet i.
- *Stale (11)*: The *Stale* state indicates when a data structure might be in chiplet m's cache, but its values are not the most up-to-date (unlike *Not Present*, which guarantees the data is not in a chiplet's caches). For example, if another chiplet k wrote the data structure after chiplet m accessed it, and chiplet m has not accessed the data again subsequently, the data structure may still be in m's L2 cache. Thus, chiplet m must be invalidated before it is safe for m to access it again.

Figure 6 shows CPElide's state diagram, which tracks the state of each data structure in the *Chiplet Coherence Table*. The circles denote the aforementioned 4 possible states of a data structure in the *Chiplet Coherency Table*. The arrows indicate the required state transition when the conditions indicated by the text alongside the arrows occur. For example, the *Valid* state's loop indicates that the data structure's state remains in *Valid* whenever one of the following conditions (indicated as *ALR/ARR/Flush* in Figure 6) occur: *ALR*: a local read; *ARR*: a remote read; or a *Flush*: an on-going L2-cache operation that may have been initiated by updates on a different data structure. In the table, each *Chiplet Vector* entry represents the state at a particular chiplet for one data structure. Unlike most coherence protocols, CPElide does not need transient states since it is not waiting for operations to complete – instead it denotes how the data is being accessed in each chiplet. The transitions show how a *Chiplet Coherency Table* entries' state changes when different events like Acquires, Releases, Reads, or Writes occur. This state may or may not be the same as the actual state of the corresponding cache lines in the cache since the table entries' state is a conservative, coarse-grained estimate of a data structure in a given chiplet's L2. Moreover, the table's state transitions occur at kernel launches. For example if a data structure was *Valid* in chiplet 0's cache at the end of kernel 1, and chiplet 0 now receives a kernel that writes the same data structure, the state for this data structure in the table transitions to *Dirty* for chiplet 0, even though the kernel has not started. Some key Figure 6 state transitions are:

- *Valid* $\rightarrow$ *Stale*: When the address range on a chiplet i will be modified by another chiplet j in a soon-to-be-launched kernel, CPElide marks i's entry as *Stale* for this data structure. Thus, chiplet i's data in its L2 is now incoherent. However, CPElide will invalidate it before subsequent uses.
- *Dirty* $\rightarrow$ *Stale*: When an address range on a chiplet i will be written by another chiplet j in a soon-to-be-launched kernel, we must flush the dirty data from chiplet i's L2 cache before chiplet j accesses it. Accordingly, CPElide issues a flush (release), then transitions to *Stale*. Also, since the baseline coherence protocol keeps the a clean copy of the line in chiplet i's L2 cache, if these addresses will be subsequently read by chiplet i then CPElide further generates an invalidate (acquire) to ensure no stale data is accessed and transitions to *Invalid*.
- *Stay in Valid on remote accesses*: CPElide elides unnecessary invalidations by allowing caches to retain clean copies if other chiplets are also only reading data from a given address range.
- *Stay in Dirty*: CPElide's software-range based tracking helps increase L2 reuse. For example, if chiplet i accesses the same data structures across kernels, it is unnecessary to flush i's dirty data. Instead CPElide elides this release, increasing reuse.

Lazy Acquire/Release: Current GPUs perform releases at the end of kernels. CPElide instead lazily performs releases as needed. Specifically, CPElide only generates a release for a specific chiplet j once CPElide observes a subsequent kernel being scheduled on a different chiplet k will access a data structure in *Dirty* on chiplet j. Similarly, CPElide only generates an acquire for a specific chiplet i once CPElide observes that subsequent kernels (kernel A) caused a data structure(s) accessed in chiplet i to become stale, and another subsequent kernel (kernel B) wants to access the same data structure on chiplet i. Additionally, CPElide performs the release after the acquire associated with the start of a subsequent kernel, but before the next kernel issues any memory accesses. In the baseline coherence protocol when a fully dirty line is written back, the cache retains a clean copy of the line and transitions to a shared state. Delaying the release helps CPElide retain the lines written by the previous kernel. Moreover, this change still produces SC-compliant results for programs with no heterogeneous races (i.e., SC-for-HRF programs), since no subsequent accesses are performed before any necessary release (flush)

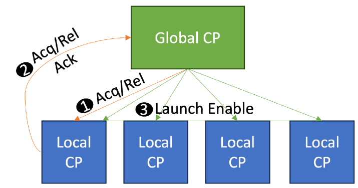


Fig. 7: Communication between local CPs and Global CP.

and acquire (invalidation) of these operations are completed.
Consistency Model Impact: We assume the standard SC-for-HRF consistency model. Thus, like current GPUs, any correctly synchronized, simultaneously executing threads will either not be writing the same address or must explicitly synchronize to guarantee correctness. Explicit synchronization operations do not impact the CP's tracking of implicit synchronization: even programs with explicit synchronization must still implicitly synchronize at kernel boundaries.

C. Functionality

Launching Kernels: Since CPElide removes the implicit acquires and releases at kernel boundaries for L2 caches, we adjust how the global CP launches kernels. Once a kernel is reaches the head of a hardware queue in the CP's packet processor, the queue scheduler attempts to schedule the kernel's WGs onto the currently available resources (Section II-B). Before dispatching any WGs the global CP inspects all data structures the kernel accesses by checking its *Chiplet Coherence Table* to determine what state these data structures are in on all chiplets. These checks happen once per kernel – ensuring no redundant *Chiplet Coherence Table* updates. If any prior kernels accessed the same data structure(s), the global CP determines if acquire and/or release operations are necessary to ensure that the data accessed by those kernel(s) is flushed or invalidated from the chiplets that accessed it. For any data structures that were accessed by the prior kernels, the global CP generates synchronization operations for the appropriate chiplet(s) to ensure correctness. The global CP sends these synchronization operations to the local CP(s), which send these requests to its corresponding L1 and L2 caches to invalidate or flush data. This approach could also be augmented by distinguishing between read-only and not read-only data [119] to only generate releases if one of the kernels writes the data structure. The local CPs will not launch WGs from the next kernel until the appropriate acknowledgments for acquires and releases are received by the global CP, which then sends a “launch enable” message to the local CPs. Since these acquire/release acknowledgments and the final launch signal are on the critical path, we model their overhead (Section IV). Finally, once the acquires and releases complete the global CP resets the appropriate bit-vector to *Not Present* to avoid generating further acquires/releases for the same data structure.
Local/Global CP Communication: The global CP and local CPs are connected through a crossbar (Figure 3). If CPElide generates any acquires or releases when launching a kernel, these requests are sent across the crossbar from the global CP to the local CP (Figure 7). After sending out these operations,

the global CP uses an acknowledgments (ACKs) count to ensure they complete, as discussed in **Launching Kernels**.

Generating Release Requests: CPElide only sends a release (flush) when a data structure will be accessed in a new kernel and that kernel will be scheduled on chiplet(s) other than those where it is in *Dirty*. This ensures a chiplet never reads a stale value. However, if the next kernel accessing this data is scheduled on the same chiplet(s) as the previous kernel, CPElide elides the release if the WGs continues accessing the same address range(s) in a data structure, on the same chiplets.

Generating Acquire Requests: CPElide only sends an acquire (invalidation) when a new kernel will access a data structure and this kernel will be scheduled on a chiplet(s) where the data structure is in *Stale*. This ensures that a chiplet does not read stale values, without requiring implicit acquire requests across all chiplets before each kernel.

Removing Entries: When CPElide generates an acquire or release, it also updates the chiplet vectors per Figure 6. If the chiplet vector's state is *Not Present* (00) for all chiplets, we remove the entry from the table. Thus, if an acquire/release is generated for all chiplets, all table entries will be removed.

Indirect & Irregular Accesses: Although most GPGPU applications have regular access patterns [43], [133], [139], some are more irregular [23], [26], [61], [90], [105], [135].[2] For irregular accesses, CPElide identifies how a data structure will be accessed in a given kernel. In Section V, we show that CPElide effectively supports workloads with indirect accesses. However, if this information cannot be determined statically for pointer-based accesses, like the baseline CPElide conservatively performs implicit acquires and releases at kernel boundaries. Although dis-contiguous address ranges are not common in irregular GPGPU workloads, CPElide supports both contiguous and dis-contiguous address ranges. However, applications with dis-contiguous address ranges may increase *Chiplet Coherence Table* entries, which we could compensate for by increasing the table size (Section III-A) if necessary. Similarly, although rare, CPElide also supports GPGPU workloads that use recursion [115], [132]. For languages like OpenCL, CPElide can utilize access mode information for all data structures in the kernel declarations (Listing 1). For languages like CUDA and HIP, which do not have per-kernel labels, CPElide would require conservatively labeling the data structures and access modes for all kernels that may be recursively called. Alternatively, in GPU ISAs where additional instructions could be added we could add special instructions to perform this labeling per kernel.

IV. Methodology

A. Baseline GPU Architecture

We model a tightly coupled CPU-GPU architecture with a unified shared memory address space and coherent caches. All CPU cores and GPU CUs are connected via an inclusive L3, which is also the directory. Figures 1 and 3 illustrate our baseline GPU, which is similar to prior work [116], [117]. Each GPU chiplet has an L1 cache and LDS per CU, and an L2 cache shared across the chiplet's CUs. The per-chiplet L2 caches are connected via inter-chiplet links using a crossbar [116]. Section IV-C1 discusses more design choices.

GPU Feature	Configuration
GPU Clock	1801 MHz
CUs/Chiplet; Complexes/Chiplet	60; 1
SE/Chiplet, SA/SE	4, 1
Num Chiplets	2, 4, 6
Total CUs	120, 240, 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
Local/Remote L2 Latency	269/390 cycles
L2 Write Policy	Write-back with write allocate
L3 Size	16 MB, 64B line, 16-way
L3 Latency	330 cycles
Main Memory	16 GB HBM, 4H stacks, 1000 MHz
Inter-chiplet Interconnect BW	768 GB/s
Scheduling Policy	Static Kernel Partitioning

TABLE I: Simulated baseline GPU parameters

B. System Setup

Although CPElide could be implemented in existing GPUs by re-programming the CPs, GPU vendors have not disclosed an API [79], [80], [114], [140]. Thus, we simulate CPElide in gem5 [42], which our group recently extended to support multiple chiplets [141].[3] Although other simulators also support modern GPUs [18], [65], [126], we chose gem5 because it has the most detailed CP model and models GPUs with high fidelity [42]. Specifically, we use ROCm 1.6 [11] and gem5 v21.1 [19], [84], which we extended to model local and global CPs, and implemented CPElide in the global CP. We modified the workloads to label the address ranges and access modes similar to the Section III-B examples.

Table I summarizes the common key system parameters, which is based on an AMD Radeon VII GPU and which our group previously validated to tune gem5 relative to real hardware [55], [65], [112], [113], [137]. To measure energy consumption we leverage prior work's per-access GPU energy models [30], [31], [45], [104], [140], scaled to handle multi-chiplet GPUs. Since CPElide only impacts the memory subsystem, we only report energy numbers for it. Similar to prior work, the modeled local/global CP latency is 2 μs [42], [96], [110]. The CPs frequency is 1.5 GHz [98] and the CPs private memory's access latency is 31 cycles [74]. Since our tables use $\sim$2 KB (Section III-A), they fit in the CP's private memory and do not change the GPU's area. The global CP and local CP are connected via high bandwidth crossbar, with 65 cycles of unicast latency and 100 cycles of broadcast latency.

[2] The data structures in all applications we study (Table II) use either 1D or 2D arrays.

[3] Although our group's prior work refers to the gem5 support as "multi-GPU", it does not have an inter-GPU interconnect and its chiplets are configured similar to a MCM-GPU in Figure 3 [116], [117].

We factor this overhead into CPElide. However, since there are few messages and communication only happens at the start of some kernels, the overall impact is negligible.

Although our changes (Section III-B) add some complexity to CPs, they only add $\sim$2% to the CP's total lines of code. We also estimate CPElide's computational requirements by executing our changes (Section III-B) on a CPU with similar specifications to the CP. Specifically, CPElide's algorithm consists of: reading/writing the *Chiplet Coherence Table* for the kernel's data structures and generating the appropriate acquires/releases. Although the first component varies with number of data structures per kernel, on average our applications access 4 data structures per kernel. Overall, the CP requires 6 μs to perform CPElide's new operations. We incorporate this overhead into CPElide, although since GPU's enqueue kernels before launch and nearly all kernel's runtime exceeds 6 μs, this latency is usually hidden for all but the first kernel. Thus, our changes have a small impact on the CP's complexity and performance.

C. Configurations

To determine CPElide's efficacy, we evaluate the following configurations: (scheduling and page placement policies discussed in Section IV-C1):

Baseline: Baseline implements the multi-chiplet GPU described in IV-A. It use gem5's VIPER GPU coherence protocol, extended for chiplet-based GPUs [141]. Baseline forwards remote requests to the home node and writes through remote stores and writes back local stores.

CPElide: Our proposed CPElide approach (Section III) uses **Baseline**'s coherence protocol, forwarding policy, and write policies, but elides acquires and releases as appropriate.

HMG (NHCC): HMG [116] is a state-of-the-art chiplet-based, MGPU coherence protocol. Although HMG was primarily designed for MGPU systems, since we focus on a MCM-GPU (Section II-A), we compare against HMG's MCM-GPU variant. Since HMG's code is not publicly available, we implemented it in gem5. Our HMG uses a L2 coherence directory with 12K entries for each GPU chiplet, with each entry covering four cache lines (i.e., the directory covers 64K cache lines). This sizing is equivalent to largest directory size HMG studied.[4] In HMG the home node always contains each memory location's most up-to-date value. Thus, unlike **Baseline** and **CPElide**, **HMG** writes through all caches entries to its home node. Further, HMG also sends writes through to memory and retains a valid copy in the home and sender L2 caches. Although HMG evaluates write-through caches, it also discusses a potential write back L2 cache variant. We also implemented and evaluated this variant, but it performed significantly worse (13% geomean) than the write through L2 variant because it reduces HMG's precise tracking benefits. Thus we use HMG's write through variant in our evaluation.

[4]gem5 uses 64B cache lines, while NVArchSim uses 128B cache lines. Thus, gem5 has double the number of cache lines for a given cache size. As a result, in gem5 HMG has twice as many directory entries as NVArchSim for a given cache size, reducing HMG's directory pressure.

Application	Input
Moderate-to-high inter-kernel reuse	
BabelStream [32], [33]	524288
Backprop [25]	65536
BFS [25]	graph128k.txt
Color-max [26]	AK.gr
FW [26]	512_65536.gr
Gaussian [25]	256x256
HACC [78]	0.5 0.1 512 0.1 2 N 12 rcb
Hotspot3D [25]	512 8 20 power_512x8 temp_512x8
Hotspot [25]	512 2 20 temp_512 power_512
LUD [25]	512.dat
Lulesh [78]	1.0e-2 10
Pennant [78]	noh.pnt
RNN-GRU [94], [95]	BS:4, TS:2, Hidden Layers: 256
	BS:16, TS:4, Hidden Layers: 512
RNN-LSTM [94], [95]	BS:4, TS:2, Hidden Layers: 256
	BS:16, TS:4, Hidden Layers: 512
Square [12], [21]	524288 1 2 2048 256
SSSP [26]	AK.gr
Low inter-kernel reuse	
BTree [25]	mil.txt
CNN (Conv+Pool+FC) [35]	128x128x3, BS:4
DWT2d [25]	rgb.bmp 4096x4096
NW [25]	8192 10
Pathfinder [25]	200000 100 20
SRAD_v2 [25]	2048 2048 0 127 0 127 0.5 2

TABLE II: Evaluated Benchmarks

1) Design Decisions: We also made the following design choices for the different configurations (Section IV-C):

Scheduler: We use static, kernel-wide WG partitioning to divide a kernel's WGs into groups [16], [89]. These groups are sent to individual chiplets, where the local CP's local dispatcher round robin schedules them onto individual CUs. Although Locality & Data Movement (LADM) [64] proposes more nuanced compile-time static analysis of kernels, we use static kernel-wide partitioning since it is most common.

Page Placement Policy: To isolate CPElide's effects as much as possible, all configurations use the state-of-the-art First Touch page placement policy [16], [116]. The first touch policy determines the home node (chiplet) for a given physical address. However, sometimes first touch is ineffective [38] and different placement policies can skew performance.

D. Benchmarks

We examine 24 popular traditional GPGPU, graph analytics, HPC, and ML applications with diverse memory access patterns from gem5-resources [21]. Table II summarizes these workloads, which have up to 510 dynamic kernels and 11 *Chiplet Coherence Table* entries, and never overflow the *Chiplet Coherence Table*. We excluded unsupported applications in gem5 (e.g., those using textures [25]). For all applications we configured their input sizes to ensure the chiplet-based GPU had reasonable occupancy and memory footprints [51]. Moreover, given our modeled system (Section IV-A) we modified all applications to use UVM. We also updated all applications to use page-aligned memory allocations to reduce unintentional false sharing [6]. Like prior work [47], [51], [66] we group the applications into those with: (a) moderate to high inter-kernel reuse and (b) low to no inter-kernel reuse. We compute this by calculating the miss rate reduction from inter-kernel reuse with no flush/invalidation overhead.

E. Sensitivity Study: Number of Chiplets

The ROCm version integrated with gem5 for multi-chiplet experiments [141] only supports up to 7 chiplets, due to ROCm 1.6 memory aperture size constraints. Thus, to understand the impacted of the number of chiplets, we evaluated all applications and configurations for 2, 4, 6, and 7 chiplets. In Section VI we discuss how CPElide applies to systems with additional chiplets. Moreover, we use strong scaling – same amount of work, but divided across the chiplets – since this is representative of an application running on a chiplet-based GPU of a given size.

V. Results

Figure 8 shows the *Baseline*'s, *HMG*'s, and *CPElide*'s normalized performance, across all applications, for 2-, 4-, 6- and 7- chiplet GPUs. We subdivide this figure into two groups: moderate to high inter-kernel reuse and low inter-kernel reuse. Figure 9 shows the memory subsystem's normalized energy consumption for a 4-chiplet GPU, divided into L1 instruction and data caches, LDS, L2 cache, NOC, and DRAM. Figure 10 shows the normalized network traffic for a 4-chiplet GPU, measured in flits and divided into multiple components: L1-to-L2, L2-to-L3, and remote. Overall, *CPElide* improves performance (13%, 19%), energy consumption (14%, 11%), and network traffic (14%, 17%) over both the *Baseline* and *HMG*, for 4-chiplet GPUs. These trends also continue for 2-, 6-, and 7-chiplet GPUs. Moreover, *CPElide* does not hurt performance for applications with little or no reuse.

A. 4-Chiplet GPUs: CPElide vs Baseline

Moderate-to-High Inter-Kernel Reuse: *CPElide* usually improves performance for workloads with larger ($> 15\%$) inter-kernel reuse in 4-chiplet GPUs (Figure 8). Since these applications have significant inter-kernel reuse, they benefit from *CPElide* preserving their inter-kernel locality. However, the results vary with each application's access patterns. Applications with iterative GPU kernels and uniform access patterns (e.g., BabelStream and Square) can easily divide WGs into chunks that can be scheduled on independent chiplets with limited remote accesses and their working sets fit into the chiplet's aggregate L2 capacity. Thus, *CPElide* outperforms *Baseline* by 31% on average for them. Likewise, the RNN's have producer-consumer style inter-kernel reuse, including input matrix weights. *CPElide* preserves this reuse, improving their performance by 11% on average. Finally, Hotspot3D performs a memory bound 3D stencil; inter-kernel L2 reuse for its read-only arrays help *CPElide* outperform *Baseline* by 37%.

More irregular applications like Color, SSSP, and BFS have many read-only memory accesses [52]. Thus, avoiding unnecessary acquires improves their inter-kernel reuse and performance: 16% for Color, 14% for SSSP, and 6% for BFS (BFS has less potential inter-kernel reuse). Similarly, Pennant and Lulesh use indirect addressing or have unstructured data structures causing irregular memory access patterns [78]. However, since these accesses are limited to a subset of addresses that fit into the aggregate L2 capacity, *CPElide* improves their performance by 38% and 16%, respectively.

GPU applications also frequently access data in three phases: loading data into the LDS, performing compute operations on the data, and finally writing data back to global memory. Here inter-kernel cache locality only helps for the first (read into LDS) and the last (write to global memory) phases. Thus, these application's benefits depend on ratio between the phases: compute-bound applications see little benefit, whereas memory-bound applications with few ALU operations benefit more: e.g., Backprop (10%) and LUD (48%).

Other applications have weak correlation between inter-kernel reuse and performance. Hotspot is compute-bound with sufficient on-chip memory bandwidth to keep the CUs busy – hence *CPElide*'s speedup for it is low. Hotspot is also bottlenecked by compute stalls. Thus, loading the LDS faster via more L2 hits does little to alleviate this problem. Moreover, sometimes (e.g., FW, Gaussian, HACC) there is sufficient memory-level parallelism to hide the L2 cache misses caused by implicit kernel boundary synchronization. Thus, although *CPElide* improves their L2 inter-kernel reuse, other accesses must go to main memory. Consequently, hitting more in the L2 cache does not significantly improve their performance.

Low-to-No Inter-Kernel Reuse: Unsurprisingly, *CPElide* and *Baseline* perform similarly for workloads (e.g., BTree, CNN, DWT2D, NW, and Pathfinder) with limited or no inter-kernel reuse. Since these applications do not have significant reuse, eliding acquires and releases does not significantly affect them.

B. 4 Chiplet System: CPElide vs. HMG vs. Baseline

Moderate-to-High Inter-Kernel Reuse: For applications with little to no remote accesses (e.g., BabelStream, Square), *CPElide* elides all flushes and invalidations except the final ones (Figure 8). However, since *HMG* uses write-through L2s, it always writes through to memory, generating much more L2-L3 traffic than *CPElide*. This significantly slows down *HMG* versus *CPElide*: 37% for BabelStream and 40% for Square. Compared to *Baseline*, *HMG* caches remote traffic, evicting some local data from the cache and generating invalidation traffic. Consequently, *HMG* performs slightly worse than *Baseline*, which cannot provide inter-kernel reuse.

Likewise Color, SSSP, and FW have input-dependent memory accesses which cause many remote accesses, since the first-touch page policy is subpar when the access pattern is irregular [38]. *HMG* caches all remote accesses at their home node. Thus when the data locality in remote accesses is low, it generates considerable invalidation traffic, and reduces space for that chiplet's local reads and writes, preventing *HMG* from reusing more local data. On average, *CPElide* is 26% faster than *HMG* for the graph analytics workloads. *Baseline* also sometimes outperforms *HMG* for these workloads. Although *Baseline* cannot provide inter-kernel reuse (unlike *HMG*), it better leverages intra-kernel L2 cache reuse for local reads and writes due to *HMG* caching data in the home node. Lulesh's irregular access patterns cause considerable *HMG* invalidation traffic, enabling *CPElide* to outperform *HMG* by 33%.

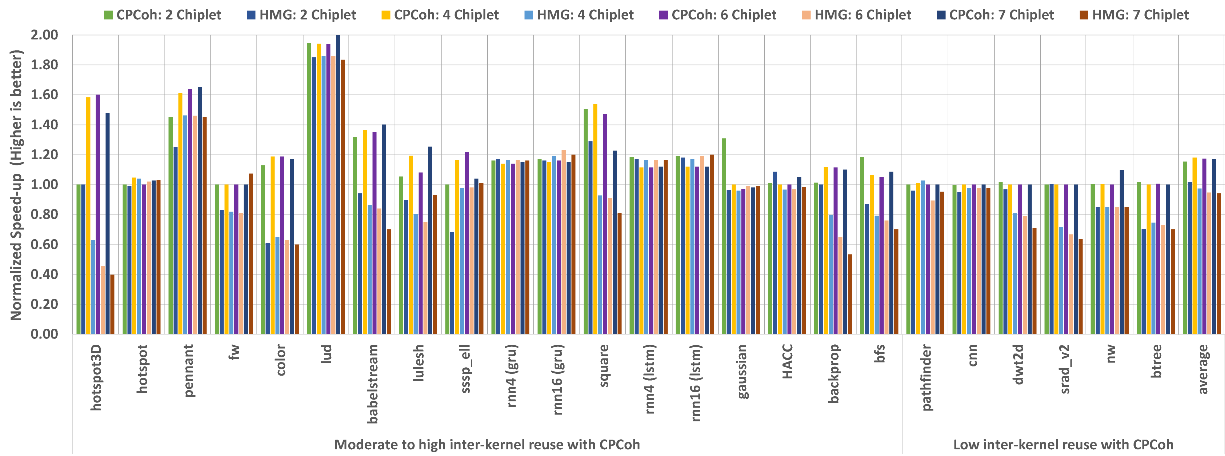


Fig. 8: Performance of *CPElide* and *HMG* on a 2-, 4-, 6- and 7-chiplet GPU, normalized to *Baseline* for each number of chiplets.

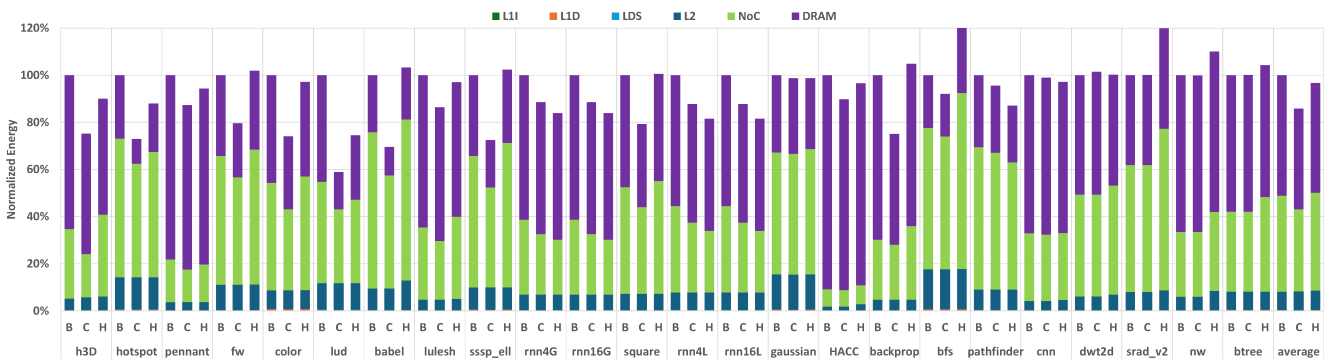


Fig. 9: 4-chiplet GPU memory subsystem energy consumption for *Baseline* (B), *CPElide* (C), and *HMG* (H), normalized to *Baseline*.

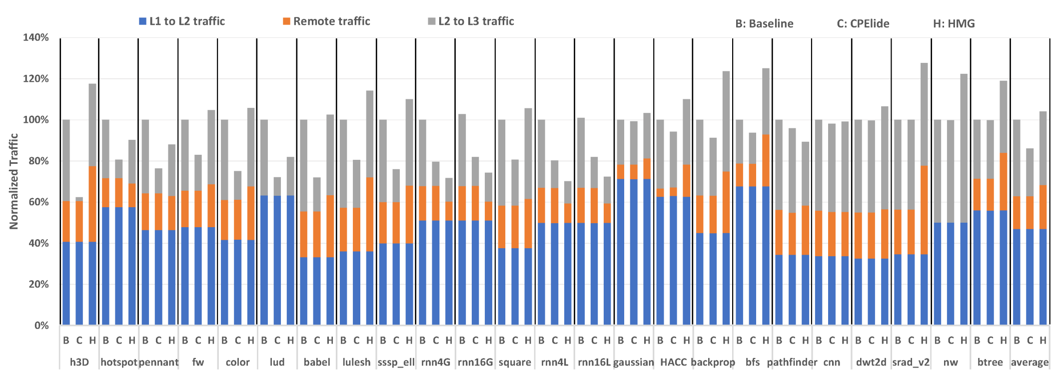


Fig. 10: Interconnect traffic for *Baseline* (B), *CPElide* (C), and *HMG* (H) on a 4-chiplet GPU, normalized to *Baseline*.

Pennant (38%) and LUD (48%) have significant inter-kernel reuse. However, *CPElide* and *HMG* perform similarly for them since both capture inter-kernel reuse and have low invalidation traffic in *HMG*. For LUD, *Baseline*, *HMG* and *CPElide* all have ∼0% remote traffic because it has many LDS accesses, the working set fits in the shared LLC, and the 4 chiplets perfectly partition the work. *CPElide* and *HMG* also perform similarly for compute-bound benchmarks (CNNs and Hotspot) and benchmarks with limited inter-kernel reuse (Pathfinder, DWT2D, and HACC). The RNNs have good remote read locality from the shared input weights and intermediate results, enabling *HMG* to slightly outperform (3%) *CPElide* since *CPElide* does not cache remote reads. *HMG* also outperforms *Baseline* by improving inter-kernel locality and providing remote read intra-kernel locality. For most applications both *CPElide* and *HMG* reduce L2-L3 traffic versus *Baseline* (Figure 10). Thus, to varying degrees both *CPElide* and *HMG* preserve inter-kernel reuse. However, *CPElide* reduces L2-L3 traffic by 37% versus *HMG* because *CPElide* does not cache remote data, which often has low locality. *CPElide* also leverages producer-consumer info without maintaining directory sharer's lists like *HMG*. *HMG* also has 23% more remote traffic than *CPElide* due to invalidations from tying four cache lines to a directory entry, which causes additional invalidations and evicts local data on remote accesses. Overall *CPElide* reduces network traffic by 17% over *HMG*.

Low-to-No Inter-Kernel Reuse: For limited inter-kernel reuse applications (e.g., BTree, SRAD_v2), ***Baseline* often outperforms *HMG***. Here *HMG* binding four cache lines to one directory entry causes many directory evictions. These evictions also generate many remote invalidations, hurting *HMG*'s performance. Consequently, *Baseline* outperforms *HMG* for these workloads by 15% on average, while *Baseline* and *CPElide* perform similarly. Although initially surprising, recent work corroborated that *HMG* suffers in these situations [38]. Thus, while *HMG* sometimes outperforms *CPElide*, in aggregate *CPElide* outperforms *HMG* by intelligently eliding synchronization operations. *HMG* fares much better against *Baseline* by leveraging inter-kernel reuse. However *HMG*'s write policy, remote read caching, and binding 4 cache lines to one directory entry sometimes hurt it compared to *Baseline*.

Energy Consumption: Overall, in a 4-chiplet GPU *CPElide* reduces average energy consumption by 14% and 11% over *Baseline* and *HMG*, respectively (Figure 9). Unsurprisingly, neither *CPElide* nor *HMG* significantly improves energy consumption for the L1 or LDS relative to *Baseline* since neither affects their behavior. Moreover, since these accesses are cheap their overall energy contribution is low. For example, even though LUD has many LDS accesses, the LDS's energy per access is much lower relative to the NOC and DRAM. Thus LDS does not significantly affect LUD's energy footprint. Interestingly, neither *CPElide* nor *HMG* significantly affect L2 energy – since the L2 must be accessed in both regardless of whether the access hits or misses. Thus, *CPElide*'s and *HMG*'s main differences come from reducing network traffic and main memory accesses. While both *CPElide* and *HMG* reduce these components energy over *Baseline*, *CPElide*'s ability to retain more data in chiplets relative to *HMG*, via eliding acquires and releases, reduces its average DRAM energy by 4% over *HMG*. Similarly reducing network traffic reduces *CPElide*'s NOC energy by 7% on average over *HMG*. Generally, the reasons for these differences are similar to Section V-B's performance discussion. For example, *HMG*'s better reuse for the RNNs helps it to provide slightly lower energy consumption for them, while *CPElide* provides much better energy consumption for applications like BFS where *HMG*'s write-through L2s generate much more L2-L3 traffic, increasing NOC energy.

C. Number of Chiplets

Generally, the 4-chiplet GPU *CPElide* and *HMG* trends also hold for 2-, 6- and 7- chiplet GPUs (Figure 8). However, there are some exceptions. For example, *CPElide* does not improve Backprop's, Hotspot3D's, and SSSP's 2-chiplet performance since its aggregate L2 cache capacity is insufficient for their larger memory footprint. Conversely, *HMG* fairs considerably better for SRAD_v2 and DWT2D with 2-chiplets – since there are fewer places for remote requests to go, there is less invalidation traffic and fewer directory invalidations. *HMG* also improves performance for benchmarks which suffered from low locality in remote reads, since fewer remote cache lines are cached since there are fewer remote nodes. Thus, local cache line reuse increases. Consequently, for 2-chiplet GPUs *CPElide*'s overall improvement over *HMG* decreases by 9% relative to the 4-chiplet GPU. For 6- and 7- chiplet GPUs benchmarks like Hotspot3D and LUD have better hit rates with *CPElide*, slightly improving their performance over 4-chiplets. Conversely, with more number of chiplets *HMG*'s performance for benchmarks like Hotspot3D suffers: its remote traffic significantly increases, reducing remote locality, since the working set is distributed across more chiplets. However, as in Section V-B, *HMG* slightly outperforms *CPElide* (4%) for the RNNs due to their good remote read locality and *CPElide* not caching remote reads. Broadly these performance trends continue for 7 chiplets: *CPElide*'s normalized average speedup over *Baseline* is 17% and *HMG* normalized average speedup is 6% worse than *Baseline* – largely due to the increased number of remote nodes. Overall for 6- and 7-chiplets *CPElide*'s average performance improvement over *HMG* increases by 1% and 2% respectively relative to the 4-chiplet configuration, continuing to show *CPElide* scales better than *HMG* as chiplets increase. Since future GPUs are likely to have more chiplets per GPU, and *CPElide*'s gains increase (slightly) as chiplets scale, *CPElide* improves scalability. The performance improvements percentages are similar for higher chiplet configurations since we use strong scaling (Section IV-E). Moreover, Figure 8 normalizes the results to *Baseline* for each number of chiplets – as chiplets increase, absolute runtime decreases. In Section VI we also show *CPElide* improves performance for multi-stream applications and that *CPElide*'s additional overhead is small for systems with even more chiplets – further demonstrating *CPElide*'s benefits and scalability.

VI. Discussion

Chiplet-based GPU versus Multi-GPU systems: To the best of our knowledge, CPElide is the first to leverage CP information to mitigate synchronization overheads in chiplet-based GPUs. As discussed in Section II-A, MCM-GPUs and MGPUs present different challenges. We focus on a single GPU with multiple chiplets because there are opportunities for improved reuse within a single GPU (Section V). However, CPElide can also be applied to MGPU systems where each GPU has multiple chiplets – by improving MCM-GPU performance, CPElide can also potentially help MGPU systems.

CPElide Scalability: Due to issues with the version of ROCm we use (Section IV-E) we were only able to simulate systems with up to 7 chiplets. However, to study how CPElide scales to larger numbers of chiplets, we performed a scaling study where we added additional acquires and/or releases at kernel boundaries to mimic those additional chiplets would need. For example, for hypothetical 8- or 16-chiplet configurations we use 2 and 4 sets of acquires/releases at kernel boundaries, respectively. This study is conservative since it adds additional sequential overhead to the 4-chiplet system – some of these acquires/releases would be performed in parallel in a larger system, but are serialized in our study. Thus, it overestimates CPElide's overhead for systems with more chiplets. Nevertheless, the additional overhead for hypothetical 8- and 16-chiplet systems (graph omitted for space) are small: 1% and 2% average slowdown for 8- and 16-chiplet systems, respectively. Accordingly, CPElide continues to scale well for systems larger numbers of chiplets.

Fine-grained Hardware Range Based Flush: Although CPElide uses range-based tracking to determine which addresses to flush/invalidate, it must still flush or invalidate the entire cache even if it is only necessary for some cached addresses since CPElide is in the global CP. To avoid flushing or invalidating the entire cache requires additional address translation support: CPElide's software hints track virtual addresses but GPU L2 caches are physically addressed. Thus to perform hardware range-based flushes, CPElide would need to translate the virtual address ranges to physical addresses. Since most GPU vendors use page-aligned array allocations, flushes/invalidations of these address ranges can be broken into page-wise requests. These requests can be sent to the core, translated into the corresponding physical pages, and then bypass the L1 cache (which must always be flushed at synchronization points) to perform targeted flushes at the L2. This technique may require multiple cache walks depending on the address range's size. However, if writeback time exceeds cache walk time, then critical path latency will be unaffected.

Annotation Implications: CPElide requires access mode and optionally address range software hints (via Section III-B's programming interface) – which may be challenging. However, prior work observed that many GPGPU workloads have simple, linear/affine data structures [43], [133], [139]. Thus, identifying this information in most GPGPU workloads is relatively straightforward. Moreover, recent compiler and runtime work showed that identifying such information can potentially be automated, especially for workloads with relatively simple access patterns (like most GPGPU workloads) [36], [71], [72], [75], [107]. Accelerators (including GPGPUs) also increasingly utilize high-level frameworks [2], [13] or libraries [10], [67], [100], [102], [134]. These libraries and frameworks core code is largely written by expert developers who can provide the appropriate hints. Accordingly, most programmers can utilize these highly tuned kernels with embedded access information. Thus, while like others [5], [27], [92] we manually annotate programs (Section IV-D) to demonstrate CPElide's efficacy, library, high-level framework, or compiler integration will avoid requiring most programmers to modify applications.

Multi-Stream Workloads: Although our workloads do not use multiple streams, CPElide will also help multi-stream workloads that concurrently run independent kernels from different streams. Data movement and locality are also challenging here since concurrent kernels may contend for shared caching resources. Accordingly, CPElide's ability to track data placement and elide unnecessary implicit synchronization can improve performance. To demonstrate this we evaluated *streams*, the only GPU benchmark in gem5-resources [21] that used multiple streams. We also extended a subset of our benchmarks (Table II) to run multiple parallel streams to mimic concurrent jobs, similar to prior work [62]. Overall, on average *CPElide* outperformed *HMG* by 12% for these workloads (graph not shown due to space constraints) for 4-chiplet systems. Largely the trends mirror single-stream workloads with similar access patterns, although for some with moderate-to-high inter-kernel reuse *CPElide* and *HMG* see additional benefits over *Baseline* due to higher synchronization costs. Thus, *CPElide* also helps multi-stream workloads.

Managing Implicit Synchronization at Driver: Like the CP, the GPU driver also knows which data structures each kernel accesses. Thus, the GPU driver could manage implicit synchronization. However, since the driver does not know which chiplet(s) a kernel's WGs will be scheduled on, the CP would have to frequently send this information to the driver (as discussed in Section VII). Prior work has shown this adds significant latency, hurting performance [28], [79], [140]. Conversely, CPElide is tightly integrated with the GPU at the global CP, where scheduling decisions are made. Nevertheless, applying CPElide at the GPU driver would also be novel.

Directories: Although CPElide's tracking mechanism bears some similarity to directory-style coherence protocols, they serve different, complementary purposes. Directory protocols primarily focus on fine-grained, cache line granularity coherence ordering between requests. Conversely, CPElide's tracks larger coarse-grained data structures. Instead CPElide only enforces ordering at implicit synchronization points, not on a request-by-request basis like many directories. Thus, CPElide complements existing directory protocols, focusing on when to perform and elide synchronization operations.

Kernel Fusion: GPU software frequently use optimizations such as kernel fusion [34], [37], [39], [68], [123], [131], [144] to combine operations into a single kernel to avoid reduce data movement and redundant global memory accesses. However, kernel fusion can increase register and LDS pressure and may limit parallelism. Thus, for larger applications it may not scale and the application still requires implicit synchronization.

Other Coherence Protocols: We focused on applying CPElide to the existing GPU coherence and consistency. However, since CPElide targets kernel boundary overheads, it can also be applied to other GPU coherence protocols. For example, CPElide is compatible with HMG and Halcone [91], and monolithic GPU coherence protocols like hLRC [4], hUVM [76], DeNovo [119], or Spandex [5]. However, we compared against HMG because it is the state-of-the-art for multi-chiplet GPUs. CPElide's benefits will likely be strongly correlated with the cost of implicit synchronization at kernel boundaries in each coherence protocol. For example, CPU style coherence protocols with active sharer tracking have low cost acquires and releases, but additional overhead via more states and invalidation traffic.

Other Accelerators/GPUs: Kernels are a GPU-specific way of partitioning work. Other accelerators partition work into different types of phases and granularities. Nevertheless, since

Feature	HMG [116]	Spandex [5], [119]	hLRC [4]	Halcone [91]	SW DSM [57], [143]	HW DSM [82], [138]	CPElide
No coherence protocol changes	X	X	X	X	X	X	✓
No L2 cache structure changes	X	X	X	X	✓	X	✓
Reduces Kernel Boundary synchronization overhead	✓	✓	✓	✓	✓	✓	✓
Avoids remote coherence traffic	X	X	X	✓	X	X	✓
Designed for chiplet-based systems	✓	X	X	X	X	X	✓
Access to scheduling information to reduce overhead	X	X	X	X	X	X	✓

TABLE III: Comparing CPElide to prior work.

CPElide targets phase (kernel) boundary synchronization, it can be applied to other accelerators that utilize a similar interface. Importantly, many accelerators [7], [14], [29], [50], [58], [111] as well as ARM and NVIDIA GPUs also use embedded microprocessors (like CPs) as an interface. However, since accelerators access memory differently and often prefer different levels of integration [5], this may require a flexible coherence interface [5], [15], [125]. Regardless, CPElide can work with a wide range of accelerators and GPUs.

VII. Related Work

Table III compares CPElide to prior work across several important metrics. This prior work significantly advanced the field, but either do not target implicit synchronization like CPElide or cannot provide all of the same benefits.

GPU Coherence & Consistency: Halcone [91] and HMG [116] designed MGPU chiplet-based GPU coherence protocols. However, as shown in Section V CPElide outperforms HMG. Halcone [91] extends timestamp-based monolithic GPU coherence protocols for multi-GPU systems by adding hierarchical timestamps. However, it is unclear how Halcone works in a single GPU with multi-chiplets and it assumes low bandwidth links between GPUs, which is less important in a single GPU with multi-chiplets. Thus, CPElide provides benefits over the state-of-the-art and is the first to target kernel boundary synchronization overheads in chiplet-based GPUs and redesign the CP to track access information. Furthermore, CPElide is compatible with many monolithic GPU coherence protocols (Section VI).

Multi-core CPU Coherence: Prior multi-core CPU work like BulkSC [24] and DeNovo [27] use dynamic sets of instructions or software information to reduce explicit synchronization overhead. Although this bears some similarity to CPElide's eliding of implicit kernel boundary synchronization, neither BulkSC nor DeNovo target implicit synchronization.

Shadow Tags: Shadow tags could reduce the overhead of invalidating valid data [127], but have sizable storage overhead, accessing the shadow tag structure affects the critical path, and flushing per-chiplet dirty data at kernel boundaries would still be expensive.

Reducing Chiplet-based GPU NUMA Penalty: CARVE improves NUMA GPU performance by extending the GPU cache capacity [142], while LADM uses static analysis to improve intra-kernel locality via better scheduling [64]. Both CARVE and LADM corroborate that implicit synchronization at kernel boundaries ruins the inter-kernel locality, hurting performance. Other work optimized WG scheduling and/or placement algorithms [16], [70]. Intelligent schedulers like these could be used in conjunction with CPElide, which has detailed information about where data is being accessed and tight coupling with the WG scheduler. However, intelligent schedulers do not target implicit synchronization. AMD proposed an architecture where the LLC (the L3) and HBM are logically shared across the chiplets, but physically sub-divided across them – each chiplet has a portion of the L3 and the HBM [117]. CPElide is more attractive with this architecture: unlike HMG, CPElide will not incur remote latencies for non-local data. TD-NUCA tracks and optimizes block placement across shared LLC banks to mitigate non-uniform latency effects [22]. However, it does not preserve inter-kernel reuse within private caches like CPElide. To preserve reuse in a chiplet-based GPU, run-time scheduling information is required, which CPElide leverages via the CP.

Coarse-grained Tracking in Distributed Shared Memory: *Software and Hybrid DSM's*: CPElide also shares some similarities with software and hybrid Distributed Shared Memory (DSM), which also perform coarse-grained memory tracking – often via software coherence at a page granularity [44], [57], [73], [143]. However, these software (or hybrid) level approaches require additional support (e.g., duplicate page copies). They also require runtime scheduling information to accurately track data structure states (Section III-B) – which is unavailable at the GPU compiler/software level. This information could be passed at runtime to the host software by extending ROCm. However, there would be a significant latency penalty to wait for the host software, hurting performance [28], [79], [140]. Conversely, CPElide leverages low level access and scheduling information available in the CP to synchronize only when necessary, at different granularities, without additional copies, and without host-side software latency overheads. Although CPElide could be enhanced by static compiler analysis [44], [64], it is not always possible.

Hardware DSM's: Hardware-based DSMs monitor the coherence status of large, aligned memory regions in hardware to snoop external requests and provide region snoop responses [82]. However, this requires a warm-up phase and can lead to false sharing if the regions are not appropriately sized. This is unnecessary in CPElide, which leverages scheduling, access mode, and data range information to make coherence decisions before a kernel starts. Other proposals such as DirSW shift some of the coherence burden to software to identify independent regions [138]. However, this is difficult in GPUs since many kernels use complex data indexing mechanism leveraging multi-dimensional thread grid structures. Most GPU's also lack OS support, which DirSW relies on.

VIII. Conclusion

Emerging chiplet-based heterogeneous systems presents challenges: the additional hierarchy they introduced makes implicit synchronization even more expensive and hampers inter-kernel reuse. To overcome this we propose CPElide, which redesigns GPU CPs to track which chiplets access specific data and intelligently elides implicit acquires and releases – only performing them when and where required. Overall, on average CPElide improves performance (13%, 19%), energy (14%, 11%), and network traffic (14%, 17%) over current approaches, respectively, without requiring hardware changes.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback that helped improve this paper. This work is supported in part at the University of Wisconsin-Madison by a Fall Research Competition grant, as well as the National Science Foundation under grant SHF-2238608. Sinclair is also a Visiting Research Scholar at AMD Research & Advanced Development. However, this work was solely performed at the University of Wisconsin and did not involve Sinclair's work at AMD.

REFERENCES

[1] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, *General-Purpose Graphics Processor Architectures*. Morgan and Claypool, Synthesis Lectures on Computer Architecture, 2018.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[3] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU Concurrency: Weak Behaviours and Programming Assumptions," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 577–591. [Online]. Available: https://doi.org/10.1145/2694344.2694391

[4] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy release consistency for gpus," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016, pp. 26:1–26:13. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783729

[5] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA. Piscataway, NJ, USA: IEEE Press, 2018, pp. 261–274. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00031

[6] J. Alsop, M. D. Sinclair, A. Gutierrez, S. Bharadwaj, X. Zhang, B. Beckmann, A. Dutu, O. Kayiran, M. LeBeane, B. Potter, S. Puthoor, and T. T. Yeh, "Optimizing GPU Cache Policies for MI Workloads," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, 2019.

[7] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-Purpose Code Acceleration with Limited-Precision Analog Computation," in *ACM/IEEE 41st International Symposium on Computer Architecture*, ser. ISCA, 2014, pp. 505–516.

[8] AMD, "AMD's Asynchronous Shaders White Paper," 2012. [Online]. Available: https://developer.amd.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf

[9] ——, "HIP: Heterogeneous-computing Interface for Portability," https://github.com/ROCm-Developer-Tools/HIP/, 2018.

[10] ——, "rocBLAS Library," https://rocm-documentation.readthedocs.io/en/latest/ROCm_Tools/rocblas.html, 2020.

[11] ——, "ROCm: Open Platform For Development, Discovery and Education around GPU Computing," https://gpuopen.com/compute-product/rocm/, 2021.

[12] ——, "HIP-Examples," https://github.com/ROCm-Developer-Tools/HIP-Examples, 2023.

[13] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 929–947. [Online]. Available: https://doi.org/10.1145/3620665.3640366

[14] Apple, "Dispatch," https://developer.apple.com/documentation/dispatch.

[15] ARM, "AMBA 5 CHI Architecture Specification Architecture Specification," https://developer.arm.com/documentation/ihi0050/c/, 2018.

[16] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 320–332. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080231

[17] A. Arunkumar, E. Bolotin, D. Nellans, and C.-J. Wu, "Understanding the Future of Energy Efficiency in Multi-Module GPUs," in *25th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2019, pp. 519–532.

[18] Y. Bao, Y. Sun, Z. Feric, M. T. Shen, M. Weston, J. L. Abellán, T. Baruah, J. Kim, A. Joshi, and D. Kaeli, "NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 333–345. [Online]. Available: https://doi.org/10.1145/3559009.3569666

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[20] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2008, p. 68. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1375581.1375591

[21] B. R. Bruce, A. Akram, H. Nguyen, K. Roarty, M. Samani, M. Fariborz, T. Reddy, M. D. Sinclair, and J. Lowe-Power, "Enabling Reproducible and Agile Full-System Simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS, 2021.

[22] P. Caheny, L. Alvarez, M. Casas, and M. Moreto, "TD-NUCA: Runtime Driven Management of NUCA Caches in Task Dataflow Programming Models," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2022, pp. 1–15.

[23] D. Cederman, B. Chatterjee, and P. Tsigas, "Understanding the Performance of Concurrent Data Structures on Graphics Processors," in *International Conference European Conference on Parallel Processing*, ser. Euro-Par. Springer, 2012, pp. 883–894.

[24] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 278–289. [Online]. Available: https://doi.org/10.1145/1250662.1250697

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Comput-

ing," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, 2009, pp. 44–54.
[26] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, Sept 2013, pp. 185–195.
[27] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. USA: IEEE Computer Society, 2011, p. 155–166. [Online]. Available: https://doi.org/10.1109/PACT.2011.21
[28] Y. Choi and M. Rhu, "PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units," in *26th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2020, pp. 220–233.
[29] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications," *IEEE Micro*, vol. 34, no. 2, pp. 34–43, 2014.
[30] W. J. Dally, "Hardware for Deep Learning," SysML Keynote, Feb 2018.
[31] P. Dalmia, R. Mahapatra, and M. 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*, ser. HPCA, 2022.
[32] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models," in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds. Cham: Springer International Publishing, 2016, pp. 489–507.
[33] ——, "Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream," *Int. J. Comput. Sci. Eng.*, vol. 17, no. 3, p. 247–262, jan 2018.
[34] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Y. Hannun, and S. Satheesh, "Persistent RNNs: Stashing Recurrent Weights On-Chip," in *Proceedings of the 33nd International Conference on Machine Learning*, ser. ICML, 2016, pp. 2024–2033.
[35] S. Dong and D. Kaeli, "DNNMark: A Deep Neural Network Benchmark Suite for GPUs," in *Proceedings of the General Purpose GPUs*, ser. GPGPU. New York, NY, USA: ACM, 2017, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038239
[36] A. Ejjeh, L. Medvinsky, A. Councilman, H. Nehra, S. Sharma, V. Adve, L. Nardi, E. Nurvitadhi, and R. A. Rutenbar, "HPVM2FPGA: Enabling True Hardware-Agnostic FPGA Programming," in *IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors*, ser. ASAP, 2022, pp. 1–10.
[37] I. El Hajj, J. Gomez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism," in *49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2016, pp. 1–12.
[38] Y. Feng, "Understanding Scalability of Multi-GPU Systems," in *ACM Workshop on General Purpose GPUs*, ser. GPGPU'23, 2023.
[39] J. Fousek, J. Filipovič, and M. Madzin, "Automatic Fusions of CUDA-GPU Kernels for Parallel Map," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, p. 98–99, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2082156.2082183
[40] Y. Fu, E. Bolotin, N. Chatterjee, D. Nellans, and S. W. Keckler, "GPU Domain Specialization via Composable On-Package Architecture," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1145/3484505
[41] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 1, pp. 7:1–7:26, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2701618
[42] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2018, pp. 608–619.
[43] D. Ha, Y. Oh, and W. W. Ro, "R2D2: Removing ReDunDancy Utilizing Linearity of Address Generation in GPUs," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589039
[44] H. Han and C.-W. Tseng, "Compile-time Synchronization Optimizations for Software DSMs," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pp. 662–669.
[45] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30
[46] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs," in *20th International Symposium on High Performance Computer Architecture*, ser. HPCA, Feb 2014, pp. 189–200.
[47] J. Hestness, S. W. Keckler, and D. A. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, 2014, pp. 150–160.
[48] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free Memory Models," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS. New York, NY, USA: ACM, 2014, pp. 427–440. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541981
[49] L. Howes and A. Munshi, "The OpenCL Specification, Version 2.0," Khronos Group, 2015.
[50] HSA Foundation, "HSA Platform System Architecture Specification," https://hsafoundation.com/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf, 2021.
[51] B. Hu and C. J. Rossbach, "Altis: Modernizing GPGPU Benchmarks," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS, 2020, pp. 1–11.
[52] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, "Inter-Kernel Reuse-Aware Thread Block Scheduling," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3406538
[53] D. James, "AMD's answer to Nvidia's NVLink is xGMI, and it's coming to the new 7nm Vega GPU," Sep 2018. [Online]. Available: https://www.pcgamesn.com/amd-xgmi-vega-20-gpu-nvidia-nvlink
[54] ——, "Nvidia has "de-risked" multiple chiplet GPU designs – "now it's a tool in the toolbox"," https://www.pcgamesn.com/nvidia/graphics-card-chiplet-designs, September 2019.
[55] C. Jamieson, A. Chandrashekar, I. McDougall, and M. D. Sinclair, "GAP: gem5 GPU Accuracy Profiler," in *4th gem5 Users' Workshop*, June 2022.
[56] N. E. Jerger, A. Kannan, Z. Li, and G. H. Loh, "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*, ser. MICRO, Dec 2014, pp. 458–470.
[57] D. Jiang, H. Shan, and J. P. Singh, "Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors," in *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 217–229. [Online]. Available: https://doi.org/10.1145/263764.263792
[58] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing,

M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[59] A. Kannan, N. E. Jerger, and G. H. Loh, "Enabling Interposer-based Disintegration of Multi-core Processors," in *48th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO. IEEE, 2015, pp. 546–558.

[60] ——, "Exploiting Interposer Technologies to Disintegrate and Reintegrate Multicore Processors," *IEEE Micro*, vol. 36, no. 3, pp. 84–93, 2016.

[61] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[62] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with near-Memory Processing," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA. IEEE Press, 2020, p. 790–803. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00070

[63] S. W. Keckler, "Life After Dennard and How I Learned to Love the Picojoule," Keynote at MICRO, 2011.

[64] M. Khairy, V. Nikiforov, D. Nellans, and T. G. Rogers, "Locality-Centric Data and Threadblock Management for Massive GPUs," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2020, pp. 1022–1036.

[65] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA, 2020, pp. 473–486.

[66] M. Khairy, M. Zahran, and A. Wassal, "SACAT: Streaming-Aware Conflict-Avoiding Thrashing-Resistant GPGPU Cache Management Scheme," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1740–1753, 2017.

[67] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, "MIOpen: An Open Source Library For Deep Learning Primitives," 2019.

[68] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization," in *Proceedings of 51st IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2018.

[69] Khronos Group, "OpenCL," http://www.khronos.org/opencl/.

[70] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, "CODA: Enabling Co-Location of Computation and Data for Multiple GPU Systems," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3232521

[71] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing Indirect Memory References with milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 299–312. [Online]. Available: https://doi.org/10.1145/2967938.2967948

[72] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The Tensor Algebra Compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133901

[73] L. Kontothanassis, R. Stets, G. Hunt, U. Rencuzogullari, G. Altekar, S. Dwarkadas, and M. L. Scott, "Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, p. 301–335, aug 2005. [Online]. Available: https://doi.org/10.1145/1082469.1082472

[74] J. B. Kotra, M. LeBeane, M. T. Kandemir, and G. H. Loh, "Increasing GPU Translation Reach by Leveraging Under-Utilized On-Chip Resources," in *54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2021, pp. 1169–1181.

[75] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "HPVM: Heterogeneous Parallel Virtual Machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP. New York, NY, USA: ACM, 2018, pp. 68–80. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178493

[76] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 1:1–1:22, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2889488

[77] R. Kuper, S. Pati, and M. D. Sinclair, "Improving GPU Utilization in ML Workloads Through Finer-Grained Synchronization," in *3rd Young Architects Workshop*, ser. YArch, April 2021.

[78] Lawrence Livermore National Labs, "CORAL-2 Benchmarks," https://asc.llnl.gov/coral-2-benchmarks, 2020.

[79] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt, and L. K. John, "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*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243179

[80] M. LeBeane, B. Potter, A. Pan, A. Dutu, V. Agarwala, W. Lee, D. Majeti, B. Ghimire, E. V. Tassell, S. Wasmundt, B. Benton, M. Breternitz, M. L. Chu, M. Thottethodi, L. K. John, and S. K. Reinhardt, "Extended Task Queuing: Active Messages for Heterogeneous Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2016, pp. 933–944.

[81] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March-April 2008.

[82] M. H. Lipasti, B. Falsafi, J. F. Cantin, J. E. Smith, and A. Moshovos, "Coarse-Grain Coherence Tracking: RegionScout and Region Coherence Arrays," *IEEE Micro*, vol. 26, no. 01, pp. 70–79, jan 2006.

[83] G. H. Loh, N. E. Jerger, A. Kannan, and Y. Eckert, "Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15. New York, NY, USA: ACM, 2015, pp. 3–10. [Online]. Available: http://doi.acm.org/10.1145/2818950.2818951

[84] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[85] J. Luitjens, "CUDA Streams: Best Practices and Common Pitfalls," 2014.

[86] D. Lustig, S. Sahasrabuddhe, and O. 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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 257–270. [Online]. Available: https://doi.org/10.1145/3297858.3304043

[87] J. Manson, W. Pugh, and S. V. Adve, "The Java Memory Model," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 378–391. [Online]. Available: https://doi.org/10.1145/1040305.1040336

[88] T. Martin, R. Litwiller, N. Pathak, and R. W. Ramsey, "United States Patent Application #2023/0376318 A1: Distributed Geometry," U.S. Patent 2023/0 376 318 A1, November, 2023.

[89] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the Socket: NUMA-aware GPUs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50. New

York, NY, USA: ACM, 2017, pp. 123–135. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124534

[90] P. Misra and M. Chaudhuri, "Performance Evaluation of Concurrent Lock-free Data Structures on GPUs," in *IEEE 18th International Conference on Parallel and Distributed Systems*, ser. IPDPS, 2012, pp. 53–60.

[91] S. A. Mojumder, Y. Sun, L. Delshadtehrani, Y. Ma, T. Baruah, J. L. Abellán, J. Kim, D. Kaeli, and A. Joshi, "HALCONE : A Hardware-Level Timestamp-based Cache Coherence Scheme for Multi-GPU Systems," *arXiv preprint arXiv:2007.04292*, 2020.

[92] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO. New York, NY, USA: ACM, 2019, pp. 1009–1022. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358254

[93] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "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*, ser. ISCA, 2021, pp. 57–70.

[94] S. Narang, "DeepBench," https://github.com/baidu-research/DeepBench, 2016.

[95] S. Narang and G. Diamos, "An update to DeepBench with a focus on deep learning inference," https://svail.github.io/DeepBench-update/, 2017.

[96] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe Performance for End Host Networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 327–341. [Online]. Available: https://doi.org/10.1145/3230543.3230560

[97] NVIDIA, "CUDA HyperQ Example," http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2013.

[98] ——, "NVIDIA RISC-V Story," *4th RISC-V Workshop*, 2016. [Online]. Available: https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf

[99] ——, "Nvidia, cuda stream management," 2018. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__STREAM.html

[100] ——, "NVIDIA cuDNN: GPU Accelerated Deep Learning," https://developer.nvidia.com/cudnn, 2018.

[101] ——, "NVIDIA CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/, NVIDIA Corp., 2024.

[102] NVIDIA Corp., "NVIDIA cuBLAS," https://developer.nvidia.com/cublas, 2016.

[103] ——, "NVLink Fabric: A Faster, More Scalable Interconnect," https://www.nvidia.com/en-us/data-center/nvlink/, 2018.

[104] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: energy-efficient dram for extreme bandwidth systems," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 41–54.

[105] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: An Unbalanced Tree Search Benchmark," in *Languages and Compilers for Parallel Computing*, G. Almási, C. Caşcaval, and P. Wu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 235–250.

[106] S. Pal, D. Petrisko, M. Tomei, P. Gupta, S. S. Iyer, and R. Kumar, "Architecting Waferscale Processors - A GPU Case Study," in *25th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2019, pp. 250–263.

[107] K. Parasyris, G. Georgakoudis, E. Rangel, I. Laguna, and J. Doerfert, "Scalable Tuning of (OpenMP) GPU Applications via Kernel Record and Replay," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607098

[108] U. Pirzada, "NVIDIA Next Generation Hopper GPU Leaked – Based On MCM Design, Launching After Ampere," https://wccftech.com/nvidia-hopper-gpu-mcm-leaked/, November 2019.

[109] S. Puthoor, A. M. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann, and G. Rodgers, "Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16. New York, NY, USA: ACM, 2016, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/2884045.2884052

[110] S. Puthoor, X. Tang, J. Gross, and B. M. Beckmann, "Oversubscribed Command Queues in GPUs," in *Proceedings of the 11th Workshop on General Purpose GPUs*, ser. GPGPU-11. New York, NY, USA: ACM, 2018, pp. 50–60. [Online]. Available: http://doi.acm.org/10.1145/3180270.3180271

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

[112] V. Ramadas, D. Kouchekinia, N. Osuji, and M. D. Sinclair, "Closing the Gap: Improving the Accuracy of gem5's GPU Models," in *5th gem5 Users' Workshop*, June 2023.

[113] V. Ramadas, D. Kouchekinia, and M. D. Sinclair, "Further Closing the GAP: Improving the Accuracy of gem5's GPU Models," in *6th Young Architects' Workshop*, ser. YArch, April 2024.

[114] V. Ramakrishnaiah, B. Beckmann, P. Ehrett, R. Van Oostrum, and K. Lowery, "Cache Cohort GPU Scheduling," in *Proceedings of the 16th Workshop on General Purpose Processing Using GPU*, ser. GPGPU '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 19–25. [Online]. Available: https://doi.org/10.1145/3649411.3649415

[115] B. Ren, S. Balakrishna, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, "Extracting SIMD Parallelism from Recursive Task-Parallel Programs," *ACM Trans. Parallel Comput.*, vol. 6, no. 4, Dec 2019. [Online]. Available: https://doi.org/10.1145/3365663

[116] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems," in *26th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2020, pp. 582–595.

[117] S. J. Saleh, S. Naffziger, M. S. Bhagavat, and R. Agarwal, "GPU Chiplets Using High Bandwidth Crosslinks," December 2020, uS Patent App. 16/456,287.

[118] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–27. [Online]. Available: https://doi.org/10.1145/3352460.3358302

[119] M. D. Sinclair, J. Alsop, and S. 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*, ser. MICRO, December 2015, pp. 647–659.

[120] ——, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA. New York, NY, USA: ACM, 2017, pp. 161–174. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080206

[121] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *19th International Symposium on High Performance Computer Architecture*, ser. HPCA, 2013, pp. 578–590.

[122] R. Smith, "NVIDIA Develops NVLink Switch: NVSwitch, 18 ports for DGX-2 & more," Mar 2018. [Online]. Available: https://www.anandtech.com/show/12581/nvidia-develops-nvlink-switch-nvswitch-18-ports-for-dgx2-more

[123] M. Springer, P. Wauligmann, and H. Masuhara, "Modular Array-Based GPU Computing in a Dynamically-Typed Language," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 48–55. [Online]. Available: https://doi.org/10.1145/3091966.3091974

[124] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[125] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A

Coherent Accelerator Processor Interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7:1–7:7, 2015.
[126] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 197–209. [Online]. Available: https://doi.org/10.1145/3307650.3322230
[127] Sun Microsystems, Inc., "OpenSparc T2 System-on-chip (SoC) Microarchitecture Specification," http://www.opensparc.net/opensparc-t2/index.html, May 2008.
[128] A. Tzannes, S. T. Heumann, L. Eloussi, M. Vakilian, V. S. Adve, and M. Han, "Region and Effect Inference for Safe Parallelism (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2015, pp. 512–523.
[129] T. Vijayaraghavany, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, "Design and Analysis of an APU for Exascale Computing," in *2017 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, Feb 2017, pp. 85–96.
[130] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 372–383. [Online]. Available: https://doi.org/10.1145/3352460.3358307
[131] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, ser. GREENCOM-CPSCOM '10. USA: IEEE Computer Society, 2010, p. 344–350. [Online]. Available: https://doi.org/10.1109/GreenCom-CPSCom.2010.102
[132] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, ser. ISCA, June 2016, pp. 583–595.
[133] K. Wang and C. Lin, "Decoupled Affine Computation for SIMT GPUs," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 295–306. [Online]. Available: https://doi.org/10.1145/3079856.3080205
[134] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-Performance Graph Processing Library on the GPU," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851145
[135] Z. Wei and J. JaJa, "Optimization of Linked List Prefix Computations on Multithreaded GPUs using CUDA," in *IEEE International Symposium on Parallel & Distributed Processing*, ser. IPDPS, 2010, pp. 1–8.
[136] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-Scope Promotion: Clarified, Rectified, and Verified," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 731–747. [Online]. Available: https://doi.org/10.1145/2814270.2814283
[137] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems Software*, ser. ISPASS, 2010, pp. 235–246.
[138] D. A. Wood, S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt, "Mechanisms for Cooperative Shared Memory," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 156–167. [Online]. Available: https://doi.org/10.1145/165123.165151
[139] T. T. Yeh, R. N. Green, and T. G. Rogers, "Dimensionality-Aware Redundant SIMT Instruction Elimination," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1327–1340. [Online]. Available: https://doi.org/10.1145/3373376.3378520
[140] T. T. Yeh, M. D. Sinclair, B. M. Beckmann, and T. G. Rogers, "Deadline-Aware Offloading for High-Throughput Accelerators," in *27th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2021, pp. 479–492.
[141] B. W. Yogatama, M. D. Sinclair, and M. M. Swift, "Enabling Multi-GPU Support in gem5," in *3rd gem5 Users' Workshop*, June 2020.
[142] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2018, pp. 339–351.
[143] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 75–88. [Online]. Available: https://doi.org/10.1145/238721.238763
[144] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip," in *Proceedings of 6th International Conference on Learning Representations*, ser. ICLR, 2018.