Enhanced System-Level Coherence for Heterogeneous Unified Memory Architectures

Anoop Mysore Nataraja
University of Washington
mysanoop@uw.edu

Ricardo Fernández-Pascual
University of Murcia
rfernandez@ditec.um.es

Alberto Ros
University of Murcia
aros@ditec.um.es

***Abstract*—Heterogeneous Unified Memory Architectures (HUMA) provide a unified memory space for on-die CPUs, GPUs, and other hardware accelerators. Such architectures improve performance and energy efficiency by obviating explicit data transfers between processors. An important feature of such architectures is Heterogeneous System Coherence (HSC) which simplifies the programming model by reducing the explicit synchronizations otherwise expected of the programmers of such systems. However, due to differences in the memory models and bandwidth requirements of CPUs and GPUs, hardware implementation of coherence for such systems is often complex and comes at high power, performance, and area trade-offs.**

This paper optimizes the existing heterogeneous coherence mechanism in early AMD Accelerated Processing Units, approximately modeled in the gem5 simulator. It introduces precise sharing information in the system-level directory, which monitors both CPU and GPU cache lines, and implements a new write-back shared last-level cache (LLC). The original implementation consisted of a stateless system-level directory and a write-through LLC. Our evaluation results with a set of collaborative heterogeneous benchmarks reveal, on average, a 14.4% performance improvement and 80.8% and 50.4% reduced probing traffic and main-memory interactions, respectively. Through optimizations and adaptation of the evaluated benchmarks, this work aims to reduce the barriers to entry into HSC research.

***Index Terms*—Heterogeneous system coherence, collaborative heterogeneous applications, architectural simulator**

I. Introduction

Recent leaps in accelerating parallel workloads, and sustainable computing trends have motivated the development of faster and, importantly, more energy-efficient computing hardware. This is especially true with the rising demands from recent advancements in Large Language Model training and inference demands. Accordingly, GPUs (GPGPUs), manycores, and specialized accelerators have seen tremendous advancements and have returned profitably to mainstream GPU vendors like Nvidia and AMD [20], [22], and other hardware manufacturers. However, such systems often falter with irregular workloads with fine-grained synchronizations and in scenarios that prioritize energy efficiency.

This has motivated the development of single-die heterogeneous systems on chip (SoCs) composed of CPUs, GPUs, and specialized accelerators to reduce communication latency and energy expenses arising from explicit data transfers and synchronizations across the processors. An example of such systems is the AMD Ryzen 3 2200G Accelerated Processing Unit (APU) [5] composed of Zen CPUs and VEGA GPU with a unified memory system implemented according to the Heterogeneous System Architecture (HSA). HSA [25] is a comprehensive framework and set of standards designed to improve the efficiency and performance of systems integrating different processors such as CPUs, GPUs, and/or hardware accelerators, often on a single die. A key characteristic of such architectures is HSA Unified Memory Architectures (HUMA), which provides a unified memory space for constituent processors. Having a unified memory space obviates explicit data transfers and complex synchronizations that is traditionally handled by the programmers of such systems. An important feature of HUMA is Heterogeneous System Coherence (HSC) [23] which is a mechanism to provide cache coherence across different processor caches respecting each of their memory model expectations.

Despite sustained industrial advances in heterogeneous systems, academic research opportunities have remained restricted due to a lack of access to and insights into state-of-the-art models. The available simulators widely used in academia with complete memory system implementations and scope for heterogeneous configurations have room for improvement. It is our strong belief that this frontier gap between academia and industry needs to close to enable symbiotic research that can greatly reduce reinventing the wheel, which, ultimately, more ubiquitously values human development time.

However, despite the frontier gap between academia and industry, the translation of some academic ideas can sometimes fulfill short-term industrial utility and less frequently be a definitive arc in the development of new ideas. With that in mind, we propose enhancements to heterogeneous coherence in gem5 [7], [19], which is one of the most accessible, academically relevant, and composable heterogeneous system simulators currently available. The proposed enhancements target the original implementation of the AMD VEGA GPU model and the VIPER GPU coherence protocol [14].

We then evaluate the enhancements in the context of collaborative heterogeneous workloads in terms of both execution time, which equates to performance ceteris paribus, and network traffic, both between the directory and the main memory and between the directory and serviced L2s, which directly affects energy consumption. As such, our

background study of suitable benchmarks returned a few established benchmark suites [8], [12], [28], [29]. We focus mainly on CHAI [12] benchmarks that have high levels of collaboration between the CPU and the GPU threads with both data parallelism and fine- and coarse-grain task parallelism exhibited in the applications therein. In addition, CHAI benchmarks implement different types of atomics-based synchronization primitives [27] which provides a holistic evaluation target for the enhancements proposed.

This paper makes the following contributions:

- Provide detailed description of the CPU-GPU heterogeneous coherence protocol in gem5 (section II).
- Propose enhancements to the protocol by implementing a write-back LLC with a state-tracking system-level directory tailored to the intricacies of AMD's MOESI and VIPER coherence protocols (sections III and IV).
- Characterization and performance evaluation of the enhancements to the protocol through adaptation of CHAI benchmark suite (sections V and VI).

Our modifications to the CHAI benchmark suites and the necessary modifications to gem5 to support precise state tracking are open-sourced [1].

Through our detailed description, enhancement, and evaluation enablements through benchmark adaptations, the paper aims to reduce the barriers to entry into Heterogeneous Systems research.

II. Background

Gem5 [7], [19] is a modular architectural simulator providing models of various CPU and GPU architectures, network-on-chip implementations, and memory systems with elaborate timing information. Gem5 is a well-accepted platform for evaluating new hardware designs and improvements. This paper specifically focuses on AMD's VEGA GPU model combined with an X86O3CPU CPU model, which is a detailed out-of-order CPU model. The memory system is provided by Ruby and is a combination of AMD's implementation of MOESI on the CPU memory side and VIPER coherence protocol [14] on the GPU memory side. The need for different coherence protocols on the CPU and GPU caches arises due to the differences in the complexity of access patterns and expectations on the cache system. CPU accesses exhibit a high degree of temporal locality and complex sharing characteristics that can benefit from latency-optimizing protocols, whereas the GPU's throughput-oriented parallel access patterns would see complex protocols like MOESI as excessive and inefficient.

We approximately model the memory configuration of AMD APUs such as the Ryzen 3 2200, in particular, a heterogeneous system composed of a CPU cluster, a GPU cluster, and dedicated memory systems for each, with a shared system-level directory and the last-level cache, along with various other components such as TLBs, DMA engines, etc. The block diagram in Figure 1 represents a simplified system with one CPU cluster and a GPU cluster, with other components omitted for brevity. Ryzen 3 2200G is a commercial desktop-grade APU composed of 4 Zen microarchitecture-based out-of-order CPUs and 8 AMD VEGA 8 compute units (CUs). Table III lists the other parameters as configured.

[1]Modified Gem5: https://github.com/CAPS-UMU/gem5_GPU.git.

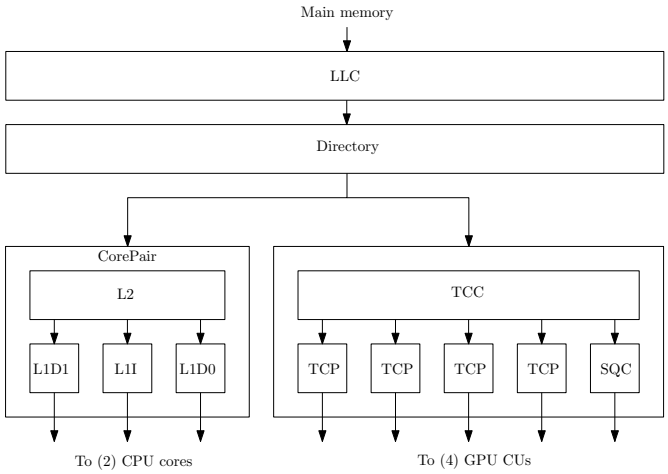


Fig. 1. Block diagram of a simple CPU-GPU heterogeneous memory system with a CorePair instance and a 4-CUs GPU instance. Constant cache is omitted for brevity

The CUs each consist of 4 16-lane SIMD units. Each CU is serviced by a Texture Cache per Pipe (TCPs) equivalent to their L1 cache, with all of them together sharing the Texture Cache per Channel (TCC) or equivalently, their L2, as well as a Sequencer Cache (SQC), or equivalently their instruction cache. In addition to these, there's a scalar constant cache for constant data that does not participate in coherence, and is not represented in the block diagram, and local scratch-pads, Local Data Share (LDS) within the CUs for CU-local data sharing.

The CPU cluster is composed of 4 CorePair subsystem instances consisting of a cache system and two X86O3CPU cores each, described in subsection II-B.

The shared directory is situated at the system level and services requests from the L2 caches within the CorePairs, the TCC(s), and the DMA engine. The TCC can also bypass requests from TCPs, which means the TCC can be non-inclusive of the TCPs. Besides the configuration as described above, there are parameters to modify functional blocks in the system; relevant parameters are: *WB_L1* and *WB_L2* to configure the TCP and TCC respectively as write-back caches from a default write-through configuration, which enables scoped synchronizations and memory interactions, and *useL3OnWT* to enable system-level atomics and write-throughs from the TCC to also write to the shared last-level cache (LLC) which, by default, bypasses it and writes directly to the main memory.

A. Coherence Protocol Description

This section explains the GPU VIPER coherence protocol. We use the following abbreviations for brevity during the

following discussions. L2 refers to L2 caches in the CorePairs shared by two CPU cores. LLC refers to the last-level cache. The Directory at the system level is backed by the LLC, and can service requests from the DMA engines, L2s, and TCC(s), and can receive the following requests from the L2s:

- RdBlk, a read permission request that can be granted either a shared or an exclusive status,
- RdBlkS, a read permission request that is specifically for a shared status often sent as a result of I cache misses,
- RdBlkM, a write permission request,
- VicDirty, a dirty victim write-back, or,
- VicClean, a clean victim write-back.

It can also receive the following requests from the TCC:

- RdBlk, a read permission request; if exclusive status is granted, it is ignored by the TCC,
- Flush request from TCP (orchestrated by the TCC) for supporting Store Release,
- Atomic request, for performing atomic updates on cache lines, executed at *system*-level visibility[2],
- A write-through (WT) request for writes to be performed at system-level visibility, which also doubles as a write-back request for writing back modified cache lines when TCC is configured as a write-back cache.

The directory also handles DMA reads (DMARd) and writes (DMAWr) from the DMA engine.

B. CorePair

CorePair refers to the subsystem composed of 2 CPU cores serviced by a shared, context-sensitive L1 instruction cache (L1I) and dedicated L1 data caches (L1D0 and L1D1) both of which are backed by a shared, inclusive L2 cache. The simulation-time parameters can configure the number of CorePair instances in the system and the cache properties of the individual caches. The CorePair caters to requests from the CPU cores through the L1Ds and the shared L1I cache, all backed by the L2. The CorePair can request the directory when there is a miss in its L2, or when it needs to victimize a line from its L2. The lines held in the L1 and L2 caches can be granted any of Modified, Owned, Exclusive, Shared, or Invalid statuses, thus implementing a MOESI coherence protocol. Any read request from the instruction cache that misses at the L2 sends a RdBlkS request, and any misses from data caches send either a RdBlk request which can be granted a Shared or an Exclusive status, or a RdBlkM which is granted a Modified status. Note that exclusive cache lines in the CorePair can silently turn modified; there is no intimation to the directory of its happening.

C. GPU Caches

The GPU cache hierarchy consists of Sequencer cache (SQC), Texture Cache per Pipe (TCP), and Texture Cache per Channel (TCC). SQC is a read-only instruction cache, and implements a simple VI-like protocol. TCP and TCC are read-write data caches and also implement simple VI-like protocols (Valid/Invalid states) with support for Global-Level Coherent (device-scope) atomics with GPU visibility. System-Level Coherent (SLC) Atomics are implemented at the directory for full-system visibility. Incoming requests with the SLC characteristics are bypassed through the TCC, and this leads to non-inclusive behavior of the TCC. TCP and TCC also have transient states for handling atomic requests. Both of them also offer write-back and write-through configurations. In the write-back configuration, TCCs offer scoped synchronizations. In both cases, TCC does not forward modified data when probed by the directory controller; however, consequently, TCC does invalidate itself.

D. Directory and Last-Level Cache

The system-level directory is modeled as a stateless directory, which means that it does not track the states of the cache lines in the L2s, TCC, or the LLC. The protocol implementation in gem5 models the LLC as a write-through, non-exclusive, victim cache.

The victim nature arises due to LLC writes only happening on write-backs from L2s, and optionally TCC, if configured to support it. In other words, missing requests receiving data from the main memory do not write to the cache, and neither do any probe-acknowledgments containing data. Consequently, LLC is not an inclusive cache.

The write-through nature arises due to writes to the LLC also necessarily writing to the memory, the utility essentially being faster reads when there is good locality in LLC accesses and guaranteed coherence between main memory and the LLC. The write-through nature is discussed further in subsection III-C

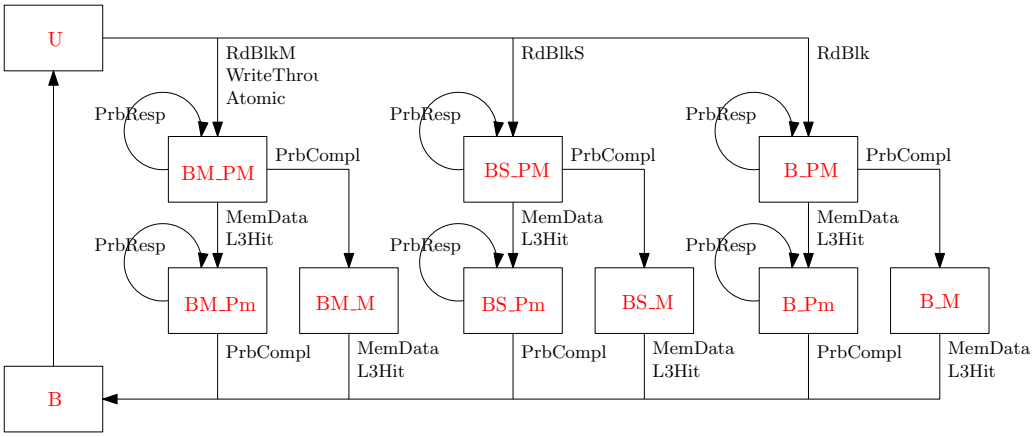
The LLC is not an exclusive cache with respect to the L2s because victim write-backs, irrespective of whether clean or dirty, do not ensure exclusivity before writing to the LLC, so there may be addresses cached simultaneously in both the LLC and one or more L2s. Note that multiple copies of an L2 line may be dirty in Shared state with another L2 line designated the Owned status, responsible for ensuring reconciliation with the LLC and main memory, upon eviction. Understandably, the possibility of clean victims implies evictions from L2s are noisy.

The LLC, as modeled, does not record or maintain the states of the cache lines, except, implicitly, the Valid bit.

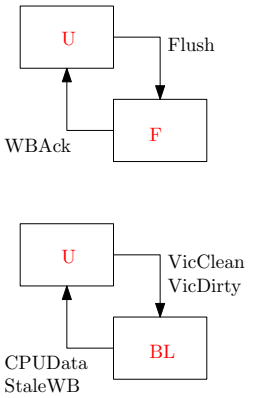
Figure 2 represents the system-level directory's state machine for handling permission requests. Some implicit actions are omitted for brevity. The default state is the unblocked U state for every line. B is the blocked state waiting for either an unblock from the core or an alternative implicit action when TCC is involved[3]. The figure represents all possible transitions brought about by incoming requests. States with _PM suffix indicate the state is blocked on both probe acknowledgments (P) and memory responses (M); _Pm indicates states blocked

[2]Atomic requests that are to be executed at *global*-level visibility (within the GPU caches) are handled entirely within the TCC and any consequential requests to the directory are treated as regular TCC requests

[3]This is done via an internal trigger-queue data structure.



(a) Permission request handling



(b) GPU flush handling (top) and CPU eviction handling (bottom)

Fig. 2. State-machine for handling incoming requests at the system-level directory

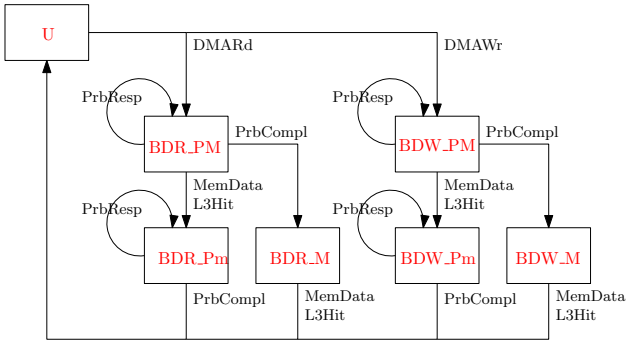


Fig. 3. DMA request handling at the Directory

only on probes; and _M indicates states blocked only on memory responses which can be resolved with data from either the LLC or main-memory. Transitions from U to any of BM_PM, BS_PM, B_PM, BDR_PM, or BDW_PM initiate probe requests broadcast to the L2s and TCCs[4]

E. DMA

The directory also processes read and write requests from the DMA engines. Figure 3 represents the state machine for handling DMA requests. Note that in the baseline, requests from DMA also broadcast probes. DMA writes also additionally probe the GPU caches.

III. Enhancements to the heterogeneous coherence protocol

In this section, we discuss three optimizations to the coherence protocol, followed by the implementation of precise sharing information in the system-level directory. We implement them on the Ruby memory system implementation on the gem5 simulator. All the enhancements target the directory state machine.

A. Early response on dirty probe acknowledgment

Baseline behavior : In the baseline behavior of the system-level directory, upon receipt of a permission request from the cores or DMA engines, the directory sends out probe requests to the L2 caches as well as a read request to the LLC which may in turn request the memory upon a miss in the LLC. The probes can be either invalidating probes or downgrading probes. Invalidating probes are sent out when the incoming requests are of types: DMAWr, RdBlkM, WT or Atomic requests. Downgrading probes are sent out when incoming requests are of types: DMARd, RdBlk, or RdBlkS. And only upon the return of both the LLC/memory read response and all the probes acknowledgments, is the original requester responded to. A receipt of a subsequent unblock signal from the requester signals the end of the coherence transaction, and the line is unblocked (state U) for further accesses.

Enhancement: For the downgrade probes, it is unnecessary to wait for all the probes to return. Upon receipt of the first probe acknowledgment with dirty data, if so is the case, the directory could safely respond to the original requesting cache with the dirty data. This enhancement is useful in cases where a second or later L2 or TCC requests for a line that has already been cached and modified by an L2 (not TCC) and has not yet been victimized. This is a commonly occurring case in heterogeneous collaborating applications with multiple CPU threads, for example, operating on a common data structure. The improvement is even more pronounced when the latency of memory or LLC access is significantly higher than the probe request round-trip, or when there are multiple sharers of the line with varying levels of business.

B. No write-back of clean victims to memory

In the baseline implementation, when an L2 sends a victim to be written, the directory controller writes it both to the

[4]Read-permission requests initiate downgrade probes which may not include the TCC as a probe destination. Write-permission requests such as RdBlkM, WT, Atomic, and DMAWr broadcast invalidating probes and include the TCC.

memory and to the LLC. However, writing to the memory is unnecessary in case of clean data; and when so, it can save a memory access. The victims will necessarily be written to the LLC. If the LLC has no free cache lines available in the corresponding set, an appropriate LLC victim line will be replaced. Note that dirty victims are not affected by this optimization and do write to memory.

1) No write-back of clean victims to LLC: We additionally evaluate the performance implications if the clean L2 victims are not cached in the LLC, despite being a victim cache, and instead only caching dirty victims from CPU L2s. Clean victims would essentially be lost *in the air* since the LLC does not cache lines on the request/refill path from the main memory, and the next time a read/write miss is incurred for the then flushed clean victim, the LLC will need to re-fetch from memory.

We hypothesize that this optimization can be advantageous in situations where multiple non-colocated L2 threads are requesting read-only accesses to a common data structure in memory. In such cases, clean victims resulting from capacity-induced evictions from L2s could be prevented from polluting the LLC with data with low to no locality of usage. Even when not, compiler optimizations that are aware of this optimization in the hardware cache system could restructure applications to have distinct data structures for in-place modified common data structures across threads. This kind of memory access patterns are common with inference and encoding in large ML models where weights or encoding vectors are read but never modified immediately.

In cases where multiple L2s or TCCs are requesting read-only accesses to the same lines one after the capacity-induced eviction of the other, this may be detrimental to performance, since it could be especially advantageous to have clean victims cached in the LLC in such cases. This situation may be possible during streaming applications, movement of computation from CPU to GPU, or when there are multiple threads processing on common data structures with wide temporal separation. When we evaluated this idea, we found inconsistent improvement and degradation across different benchmarks which we believe were due to the two cases mentioned above.

In the best case, this optimization could save an LLC write and a potential writing of LLC victim to memory with significant energy savings, lead to lesser pollution of the cache with temporally unusable content, and increased effective cache capacity for the LLC. In the worst case, if later threads request for the cleanly victimized line, they would then need the directory to re-fetch from the memory.

C. LLC as a write-back victim cache

As previously detailed, LLC is modeled as a write-through cache. Since the LLC is monolithic, there are no obvious coherence advantages from the write-through policy. Added to the fact that the DMA accesses do not update the L3, it is logical to instead have the LLC as a write-back cache. The main advantage of a write-back LLC is the resulting significantly fewer accesses to the main memory, which has both a potentially high energy efficiency improvement and a minor performance implication. The performance implication is minor because in either case, the writes or write-backs to the memory are non-blocking since the only interface from the LLC to the memory (through the directory) is ordered, however, it may expend a few clock cycles delegating the request to the memory and potentially stalling when this interface is blocked.

We modify the directory from the optimization proposed in subsection III-B to not only not write clean victims to memory, but also to not write dirty victims to memory, and to instead only write to the LLC with a corresponding dirty bit set to signal that the written victim was dirty. The dirty bit is set at the first dirty L2 victim write, which if set, will function as an indication for the LLC line to be written back to memory if and when victimized itself. Note that the block sizes for L2 and LLC are the same. It follows that when victimizing a clean line, it will not be necessary to write back the data, since it will be guaranteed to be coherent with the memory. Obviously, this depends on no other caches or DMA agents being able to update the memory transparently to the directory. The only case that would violate this assumption is GPU write-throughs and atomic operations which need to be manually re-configured to write to the LLC using simulator parameters. Understandably, evictions from the LLC would be on the critical path. So, there is an additional minor latency penalty incurred during processing L2 victims when the conflicting LLC line is dirty. It is a minor penalty due to write-back of the LLC being non-blocking and write-backs from the LLC happening transparently to L2s.

IV. Precise state-tracking system-level directory

In the traditional GPU programming model, tracking statuses of cache lines in the system-level directory often comes at a high cost of space efficiency. This is because, typically, there is little shared data that can prove beneficial when tracked, and GPUs typically access vast amounts of data, most of which is not shared and need not be tracked by the directory. Tracking GPU lines in such cases can lead to severe thrashing of CPU lines. However, with collaborative applications where GPUs are not consuming large amounts of data, tracking cache lines can be a valid consideration.

However, in heterogeneous unified memory systems, such as the one configured, effective collaborations between threads on different heterogeneous compute elements depend critically on optimal cache performance. Optimized heterogeneous system coherency can offer new opportunities for increasing system performance and energy efficiency for programmers to effectively write programs [3]. State information, when used effectively in the directory controller, can alleviate network activity and reduce main memory interactions. In addition, information on the sharers of a line can avoid broadcasting invalidating probes to instead multi-cast them to only sharers of a line, which can considerably reduce network activity and potentially improve performance. Understandably, this comes at the cost of more area for tracking sharers, and a marginally

more complex state machine in the directory controller. We evaluate these two strategies in the following subsections.

A. Tracking ownership

We propose a system-level directory that can track the ownership status of the cached lines in the processor caches, that is, caches other than the LLC. The proposed mechanism for coherence is similar in principle to an MSI protocol with a few special cases explicitly handled to support MOESI on the L2s and VIPER on TCCs.

Our coherence algorithm tracks cache lines in three stable states:

- I: line not cached in the processor caches,
- S: cache line in processor caches in shared (S) state,
- O: line modified (M) with potential dirty sharers (O), or exclusive (E) in one of the processor caches.

There is an additional transient busy state (B) for when the corresponding line in the directory is being evicted, and the directory controller shall not accept intermediate incoming requests until the completion of outstanding eviction. The transient states represented in 2a and Figure 3 are implicit.

When the directory line is in the S state, it is guaranteed that the data contained in the corresponding L2 caches is not modified with respect to the LLC. Therefore, read-only requests from any cache clients can be safely serviced from the LLC which is guaranteed to be coherent with the processor caches. Thus, probes to sharer caches can be safely elided in such cases. This can be beneficial when there are many CorePairs configured in the system since the wait times on returning probes and network traffic would increase substantially. Note that if the incoming request is a RdBlk, to a line that is in S state, it should be assigned directly a shared status without assessing if exclusive status can be granted since the response is forced to be from the LLC. Normally, only responses serviced from other shared/owned caches would grant shared status. Note that in the implementation described here, the point of allocating a directory entry for RdBlkS is to determine if it would be necessary to broadcast invalidating probes for future write-permission requests to the line under consideration.

When the line is in O state, one of the caches in possession of the line may have it modified with respect to the LLC, and if there are sharers, each of the sharers is guaranteed to possess the modified line under exactly one designated owner which stewards the responsibility of writing back to the LLC. In the implementation described, O state at the directory conservatively includes cache lines that are held in exclusive states since exclusive cache lines can turn modified silently. In such cases, it would be incorrect to service a request for a line from the LLC since the LLC can potentially be stale. However, probing all the system caches for a read-only access would be wasteful. Probing only the owner cache suffices and reduces network occupancy/traffic by as many packets as the number of CorePairs and TCCs. However, if there is a write-permission request, it would be necessary to broadcast invalidation probes to all the caches in the system to ensure that there are no stale copies of the line in the L2s if cached. The paper later discusses an optimization that proposes tracking sharers of a line that will improve this situation by restricting broadcast probes to a multicast over only caches that have cached the modified line. Note that if the incoming request is a RdBlk, the fact that the response has arrived from a cache automatically denies exclusive status in the response.

When the line is in I state, it is guaranteed that no caches are in possession of the line in question. And so, it would be unnecessary to probe any of the caches. It would be sufficient to directly read the line from the LLC. The LLC can potentially miss and request a refill from the main memory. In the original design, all compulsory misses in the directory lead to probes being broadcast to all caches. Eliding these probes contribute the most to the performance improvements observed in Section VI, with pronounced effects for long-running applications, or running on a warmed-up cache will provide more realistic values of improvement. Note that this optimization obviates the optimization in subsection III-A.

1) Directory inclusion policy: An important choice in designing an owner-tracking directory is the inclusion policy [30]. Inclusive directories enable filtering incoming requests to avoid unnecessary probing routines or memory accesses. However, in order to guarantee inclusivity, certain LLC accesses would then necessitate propagation of backward invalidations to the owning/sharing L2s — which adds additional latency, and network traffic, and reduces effective cache capacity in the L2s. Non-inclusive directories [9] do not help with filtering incoming requests in case of directory misses, since the data may reside in any of the L2s.

Recall that the LLC is a non-inclusive cache. As previously reported [26], a perfect inclusive directory would require the sum of both the entries and the associativity in L2 (and TCC). This would result in a highly associative directory. An alternative mechanism is to model the directory as a cache [13], resorting to broadcasting probes in the event of directory eviction.

In addition to the base set of transitions described in Figure 2a, the enhanced directory will need to support additional transitions to deallocate a directory entry to accommodate a new incoming request when the directory is full. Depending on the status of the deallocated directory line, there may be directory-induced invalidating probes to ensure that there are no stale copies of the deallocated line in any of the L2s.

2) State-machine description: The working of the directory owner-tracking can be inferred from Table I. The directory starts with all cache lines in state I. Upon receiving a request, the state of the line is checked. Once it has the state, and the input request, the transitions and necessary actions are as per the state-transition table in I.

If the directory line is not present and the directory is full, the newly added transition will ensure that an appropriate directory line has been victimized and deallocated. As a consequence of the deallocation, the directory may induce

TABLE I
STATE MACHINE FOR SHARER TRACKING IMPLEMENTATION IN ISO COHERENCE

State	Request	Sharer-list	State-transition	Probe	Memory RW, Additional details
I	RdBlkS	Add new sharer	S	None	R
	RdBlk	Clear	O (owner established)		
	RdBlkM | WT | Atomic		O (owner established)		
	DMARd | DMAWr		No change		
S	RdBlkS[a]	Clear	No change	None	R
	RdBlk	Add new sharer	No change	None	R, No Exclusive grant
	RdBlkM | DMAWr WT | Atomic	Clear	O (owner established)	Multicast Invalidate	R
	DMARd	No change	No change	None	R
	VicClean	Remove sharer[b]	I if sharers list empty S otherwise	None	W
O	RdBlkS[c]	If probe-resp clean: Add owner, & new sharer[d] Else: Add new sharer	If probe-resp clean: S[d] Else: No change	Unicast downgrade[e]	R + dirty probe override
	RdBlk				R + dirty probe override No exclusive grant
	RdBlkM | WT | Atomic	Clear	No change Owner updated	Multicast Invalidate	R + dirty probe override
	DMARd	If probe-resp is clean: Add owner as entry Otherwise: No change	If probe-resp is clean: S Otherwise: No change[f]	Unicast downgrade	R + dirty probe override
	DMAWr	Clear	I	Multicast invalidate	R + dirty probe override
	VicClean	Clear	I[g]	None	W
	VicDirty	Remove sharer[b]	I if sharers list empty S[h] otherwise	None	W

[a]Requester cannot be a sharer. This case would be internally handled at the CorePair by transferring requested line from L2 to L1D.
[b]In case of limited-pointer directory, a sharer should not be removed when list is full to preserve broadcasts to untracked sharers.
[c]If the requester is the same as the owner, it signals E to S transition in the L2 because of an I$ miss.
[d]This is the case where O is conservative and L2 line is actually in E, contrary to expectation. The requester is the owner.
[e]In the above case where requester=owner, there cannot have been other sharers, since the line in L2 was still E.
[f]If the L2 line was E, it will be transition to S upon receipt of downgrade probe.
[g]Any line in O state can send a VicClean if the line held in the L2 was E. [h]Other L2s can still have a dirty, shared line

invalidations to the L2s and TCC to maintain inclusivity with respect to the private caches.

At this point, the directory should have a free line available for accommodating the incoming request. Subsequently, the directory controller follows the state-transition table to safely service the request through probes and/or LLC accesses. Note that DMA requests do not lead to any state alteration since DMA engines do not cache the lines and therefore not participate in coherence.

Issuing probes consumes energy and time. We propose to alleviate the time spent probing and consequential network traffic by tracking ownership of a line. The main benefit of owner-tracking is in the case where a multiple caches have copies of a line, and start updating the lines. In such cases, the default action is to also read the LLC. Since the LLC would not have had the line cached on the refill path previously, every LLC read will miss, and require reading from main-memory. With owner tracking, the LLC reads are elided in this case.

B. Tracking sharers for more efficient probing

A further enhancement over owner tracking is to also track a list of sharers of a cache line. With the knowledge of the sharers of a cache-line, the directory would need to only probe-invalidate the sharers (a multicast instead of a broadcast) on a write-permission request (RdBlkM, WT, Atomic, DMAWr) for a line that is tracked in the directory, thereby rendering broadcasts unnecessary. In addition, backward probe-invalidations resulting from directory replacements can also be restricted to only the caches registered in the sharers list and the owners, if any. Table I tabulates the state machine decisions for all cache-line states and incoming request types. Note that there are a few special cases that need to be handled atypically due to the differing implementations of the MOESI and VIPER protocols. These are elaborated in-place in the form of footnotes. Missing transitions, such as VicDirty when cache line is in state S, are illegal and therefore omitted. Understandably, exhaustively tracking sharers is area-expensive and may overshoot the point of diminishing performance return. In such cases, our observation is that tracking only the owners would be advantageous. For the area and energy expense, whether the resulting lower network traffic is considerably profitable is a decision left to domain-specific use cases. We however present the performance improvements and the implications on the network activity through the reduction in probes sent out of the directory in Section VI.

Table I describes the detailed state machine decision diagram for sharers-tracking. An important consideration is the size of the sharers list which can be easily limited to a fixed

TABLE II
CACHE CONFIGURATIONS AS CONFIGURED

Cache	Directory Cache	LLC	L2 L1D L1I	TCC TCP SQC
Cache size	256 KB	16 MB	2 MB 64 KB 32 KB	256 KB 16 KB 32 KB
Associativity	32	16	8 2 2	16 16 8
Block size	1 B	64 B		
Repl. policy	Tree PLRU	Tree PLRU		
Access latency	20 cy	20 cy	1 cy 1 cy 1 cy	8 cy 4 cy 1 cy

maximum number of sharers (limited pointer directory [2] or coarse vector [13]). In our implementation, we tested a sharers list with as many entries as there are L2s, i.e., a full-map sharing code. The advantage of such linear scaling is in the possible use of bitmaps to track sharers, as opposed to multi-bit identifiers for each sharer in the implementation for a limited pointer directory.

V. METHODOLOGY

We implemented the proposed changes in gem5 [19] develop branch with commit `0bb7a355` and adjusted the parameters to match the configuration declared in Table III. In our testing, the CHAI benchmarks [12] have shown, on average, greater collaboration through finer-grain data sharing and synchronization than the other benchmarks we considered. We also evaluated the benchmarks part of HeteroSync [28], and Lulesh [15], available as open-source benchmarks [10]. However, the effects of the enhancements are not prominent due to their limited collaborative properties. We emphasize the reasoning of heavily and frequently collaborating applications having more opportunities to reap benefit from faster coherence communications. CHAI benchmarks are open source and available in both OpenCL [17] and CUDA [21] implementations. For our use case, we converted them to HIP [4], with minor modifications to CPU and GPU thread counts, and randomization seeds for deterministic execution and fair evaluation of optimizations. Moreover, the simulator accepts HCC-compiled kernels through the supported ROCm runtime framework. This framework comes installed within the supplied Docker image. In our exploration of the different platforms to evaluate our results, we decided on the VEGA Syscall Emulation (SE) mode as opposed to a Full System (FS) mode of the gem5 simulator for efficiency reasons. SE mode is simpler and perfectly adequate for our purposes because syscalls do not play a significant role in the region of interest of our benchmarks.

We focus our evaluation on 10 CHAI benchmarks: Bezier Surface (bs), Canny Edge-Detection (cedd), Padding (pad), Stream-Compaction (sc), Task Queue System (tq), Histogram with both input and output partitioning (hsti, hsto), and In-Place Transposition (trns), and Random Consensus (rscd, rsct).

TABLE III
SYSTEM CONFIGURATIONS SIMULATED

Parameter	Assignment
#CUs / #SIMDs per CU	8 / 16
#TCPs per CU	1
#TCCs	1
#CorePairs / #CPUs	4 / 8
CPU freq.	3.5 GHz
GPU freq.	1.1 GHz

These benchmarks allow configuration of varying GPU and CPU threads to carry out the computation with a mix of different kinds of parallelisms (data, fine-grained task, coarse-grained task). These benchmarks also implement different kinds of synchronization via std threads such as unpaired work-queues, non-ordering flags, etc.

However, despite our efforts to have all the CHAI benchmarks evaluated, we were unable to get 4 of 14 benchmarks running in the baseline model. Debugging the reasons for failure leads to spurious failures in waking CPU threads in the O3 CPU implementation within gem5 which we were unable to resolve without invasive changes to the simulator. Additionally, 2 of 14 benchmarks (data and task parallel Random Sample Consensus — rscd, rsct) failed verification even without our changes. Nevertheless, we believe the remaining benchmarks exhibit a balance of both data and task parallelism and can still reflect fairly on the potential of the proposed optimizations.

VI. RESULTS

We now discuss the quantitative performance, energy, and network activity improvements following the proposed enhancements to the coherence protocol implemented in gem5. Table III tabulates the system configuration along with important parameters that describe the behavior of the runtime system. Most noticeable improvements can be had with optimizations in Sections III-B and III-C. Early probe responses do not produce significant improvements.

Figure 4 is a bar graph of speedup achieved through reduction in runtime (number of cycles simulated) by each proposed optimization. The reductions are in relation to the baseline, which is the default unmodified HSC implementation in gem5. We see varying improvements across all benchmarks. Improvements over benchmarks with highly data-parallel implementations, such as bs, pad, hsti, hsto, rscd, are limited because of their low coherence activities. On average, we see a performance improvement of 1.68% without precise state tracking. Figure 5 is a bar graph of the number of memory requests made by the directory controller through optimizations in Sections III-B and III-C. The number of memory accesses are directly proportional to energy decrements. An average of 50.38% reduction in memory accesses is observed resulting from obviating memory accesses on every LLC write across all the benchmarks. Figure 5 also captures the performance difference between enabling TCC write-throughs to be written to the LLC instead of just the main memory through the

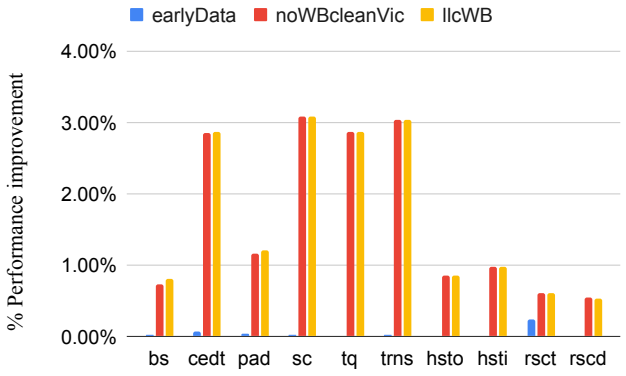


Fig. 4. Performance increments of each of the 3 optimizations for the tested benchmarks in %saved simulated cycles over baseline

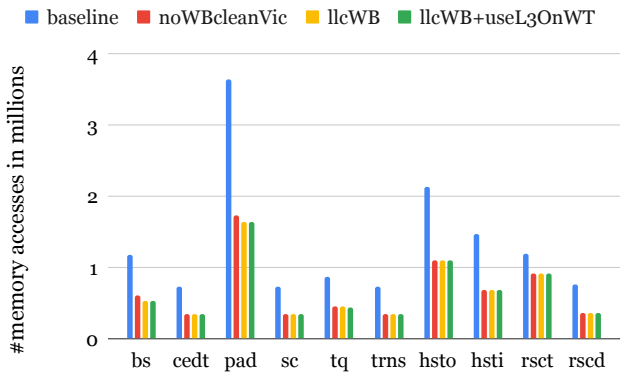


Fig. 5. Reduction in memory reads and writes from the directory in #reads and #writes over baseline

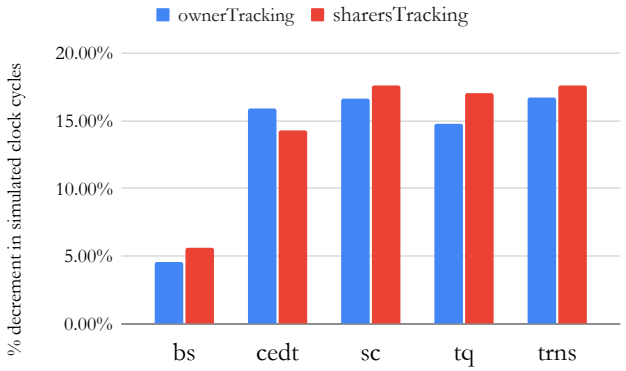


Fig. 6. Performance increments of owner and sharers tracking for the tested benchmarks in %saved simulated cycles over baseline

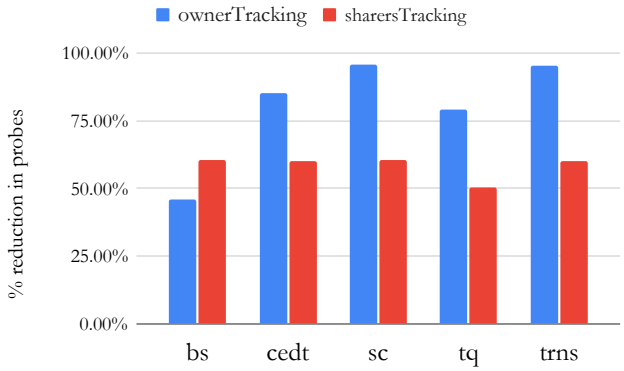


Fig. 7. Reduction in network traffic in % reduction in probes sent out from the directory.

last two bars for each benchmark. There are no noticeable differences in memory accesses due to the shorter duration of the benchmarks simulated.

Figure 6 captures the performance improvement through state tracking and sharer tracking, with the average over the five benchmarks tested being 14.4%. This improvement is a result of avoiding unnecessary probes in many cases. Figure 7 is a representation of the number of probes sent out from the directory as optimizations of state-tracking and sharers-tracking are implemented. There is a marked reduction in the number of probes sent, which reflects a reduction in network activity. In 4 of the benchmarks, sharers-tracking is not contributing much to lowering network traffic. On average, there is 80.3% reduction in probes in the five benchmarks tested.

If tracking sharers in the granularity of the individual caches is expected to pass the point of diminishing performance return on the area expenditure for a specific use case, an option is to stick to tracking ownership only since tracking sharers scales area linearly.

VII. Future work

The paper aims to provide a fertile development platform to shorten practical evaluations of sophisticated ideas to advance heterogeneous coherence research. As such, there is a scope for improvement on the proposed new baseline with state-tracking system directories, and we expect continued active development. Additionally, the coherence protocols themselves are very loosely coupled, which presents opportunities for tighter integration of the different component coherence protocols. In this section, we discuss three ideas that we reserve as future work.

The choice of the replacement policy in the directory cache is crucial so the directory controller does not unnecessarily evict lines with many sharers or lines in modified statuses since that would lead to prolonged increase in immediate network activity following the eviction in re-caching lost lines. A replacement policy that prioritizes unmodified lines with least sharers, cascaded with a tree-PLRU in case of multiple candidate filtrates, can be expected to perform better than tree-PLRU, which is otherwise the default.

Another point of consideration is that, when receiving dirty victims at the directory, it is guaranteed to be from owned or

modified lines in serviced L2s. In such cases, de-allocating the corresponding entry from the directory need not invalidate dirty sharers, if any. Dirty sharers will not forward data when probed, and hence are guaranteed to be the last users of the cache line for that transaction.

For better scalability, distributed directories have been proposed. The state-tracking directory can be made compatible with distributed directories.

VIII. Related Work

Alternative open-source simulators for GPUs architectures include AccelSim [16], NaviSim [6], and an outdated gem5-GPU simulator [1]. AccelSim and NaviSim are both GPU models that lack direct CPU integration and heterogeneous coherent cache system implementation. Gem5's VEGA GPU model along with its heterogeneous system coherence implementation provides a good baseline for evaluating our proposed enhancements.

Among heterogeneous benchmark suites, other than CHAI, there are HeteroSync [28], Rodinia [8], and Hetero-Mark [29] among a few other benchmarks. In our testing of Hetero-Mark and HeteroSync, we did not find useful levels of collaboration between the CPU and the GPU to evaluate the benefits of the proposed enhancements. The collaborative nature of the benchmarks could most easily be established with CHAI benchmarks owing to their more interspersed CPU and GPU workloads and the parameterizability of the applications themselves towards launching more/fewer CPU/GPU threads for insights on the applicability of the enhancements.

Among related works, Koukos et al. [18] improve the performance of the CPU and GPU by optimizing the coherence protocol with private/shared tracking at the TLB hierarchy. HMG [24] implements a hierarchical state-tracking system-level directory for tracking cache lines in multi-GPU systems with a VI-like protocol. The Compound Memory Models work [11] describes a compositional memory model that allows component devices to retain their behavior in their original memory models when combined. We did not find any previous attempts to improve the implementation of heterogeneous system coherence in gem5.

IX. Conclusion

In this work, we described the working of Heterogeneous System Coherence in AMD APUs as currently modeled in the gem5 architectural simulator. We adapted a selection of CHAI benchmarks to the open-source gem5 simulator and characterized their performance. We implemented enhancements to the heterogeneous coherence protocol, and a state-tracking system-level directory implementation, to make the coherence model more representative of advanced architectures. Our enhancements optimize energy efficiency and improve performance by 14.4% and lower the barriers to entry into heterogeneous systems coherence research. Future work involves a detailed analysis of the directory replacement policies. We also reserve for future work, investigation of the advantages of not tracking certain read-only memory pages and accesses that are guaranteed to be read-only.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ECHO, grant agreement No 819134), from the CIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grant PID2022-136315OB-I00), and from the MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (grant TED2021-130233B-C33).

References

[1] Gem5-gpu simulator. [Online]. Available: https://github.com/gem5-gpu/gem5-gpu

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*, 1988, pp. 280–289.

[3] J. Alsop, W. T. Na, M. D. Sinclair, S. Grayson, and S. V. Adve, "A case for fine-grain coherence specialization in heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 3, pp. 41:1–41:26, 2022.

[4] AMD. Hip documentation. [Online]. Available: https://rocm.docs.amd.com/projects/HIP/en/latest/

[5] ——, *Ryzen 3 2200G*, https://www.amd.com/en/newsroom/press-releases/2018-1-7-amd-redefines-high-performance-computing-with-new-.html, 2018.

[6] 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 *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2023, p. 333–345.

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

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[9] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "An efficient, self-contained, on-chip, directory: Dir1-sisd," in *24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 317–330.

[10] gem5. gem5-resources. [Online]. Available: https://github.com/gem5/gem5-resources

[11] A. Goens, S. Chakraborty, S. Sarkar, S. Agarwal, N. Oswald, and V. Nagarajan, "Compound memory models," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591267

[12] J. Gómez-Luna, I. El Hajj, V. Chang, Li-Wen Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[13] A. Gupta, W.-D. Weber, and T. Mowry, *Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes*. Boston, MA: Springer US, 1992, pp. 167–192.

[14] 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. G. Rogers, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 608–619.

[15] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[16] 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 (ISCA)*, 2020, pp. 473–486.

[17] Khronos. The opencl specification. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html

[18] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building heterogeneous unified virtual memories (uvms) without the overhead," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1:1–1:22, Mar. 2016.

[19] 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. Castrill'on, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. 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. M"uck, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. 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 'Eder F. Zulian, "The gem5 simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020.

[20] Nvidia. 2022 q1 financial results. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-announces-financial-results-for-first-quarter-fiscal-2024/

[21] ——. Cuda. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[23] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated cpu-gpu systems," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 457–467.

[24] 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 *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 582–595.

[25] P. Rogers, "Heterogeneous system architecture overview," in *2013 IEEE Hot Chips 25 Symposium (HCS)*, 2013, pp. 1–41.

[26] A. Ros, M. E. Acacio, and J. M. García, "A scalable organization for distributed directories," *Journal of Systems Architecture (JSA)*, vol. 56, no. 2-3, pp. 77–87, Mar. 2010.

[27] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 161–174.

[28] ——, "Heterosync: A benchmark suite for fine-grained synchronization on tightly coupled gpus," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017, pp. 239–249.

[29] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[30] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, "Ncid: A non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies," in *7th ACM International Conference on Computing Frontiers*, 2010, p. 121–130.