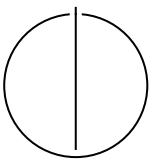
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Disaggregated, Vendor-Generic APUs with CXL

Victor Fritz Trost



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Disaggregated, Vendor-Generic APUs with CXL

Disaggregierte, Herstellerunabhängige APUs unter Verwendung von CXL

Author:	Victor Fritz Trost
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Dr. Anatole Lefort, M.Sc. Nicolò Carpentieri
Submission Date:	10.10.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 10.10.2025 Victor Fritz Trost

Abstract

Modern, data-intensive, heterogeneous workloads, such as AI training and inference, demand large amounts of memory, particularly GPU memory. As the demand has exceeded the capabilities of current hardware, solutions have emerged. While these increase effective VRAM they do so at the cost of overhead, through automatic memory movement between devices, or by increasing programming complexity through manual memory management.

Unified Memory (UM) resolves these shortcomings, by combining CPU and GPU memory. Though, UM is currently only realized in tightly integrated chips, which do not allow for memory expansion. In contrast, the Compute Express Link (CXL) promises memory expansion, but CXL-enabled GPUs are not yet available. Further, CXL has only been evaluated for monolithic memory expansion.

To address this, we propose a novel approach of leveraging CXL memory to create a heterogeneous system with a cache coherent UM. By using a shared memory pool we create a CXL enabled Accelerated Processing Unit (APU). To achieve this we utilize coherence bridges to efficiently translate between the CXL and the coherence protocols of the CPU or GPU. We employ a directory based MESIF protocol as the CPU coherence and leverage *gem5*'s VIPER protocol for GPU coherence. Our model is designed to minimize the impact of latency introduced by the used remote memory. As a proof-of-concept, we implement our model for the *gem5* simulator and analyze its performance using the Rodinia benchmark. We compare our model with the *gem5*'s default APU, finding them to be on-par. Further, we investigate the impact of increasing Compute Express Link (CXL)-Bus latency on performance. We find latency to scale simulated time linearly, though this is dependent on cache hit-rates. Overall, we show our approach is a valid solution to the memory problem modern heterogeneous systems face. Thus, this work paves the way for cost-effective, scalable systems in the AI-era. Further, it enables easier programming of such systems.

Source code for our proof-of-concept simulation can be found at `https://github.com/TrostV/cxl-apu`.

Contents

1 Introduction

With the rise of AI workloads new challenges emerge, for the increasingly heterogeneous systems [1] tasked with training and executing these workloads. They must satisfy the huge memory demand of model training, while ensuring efficient interoperability of hosts and accelerators. The memory demands of state-of-the-art AI models have exceeded what current hardware provides. Consider OpenAI's `GPT-3` model, which uses 175 billion parameters [2]. Assuming 2 bytes per parameter (i.e. `FP16`), this equates to 350 GB. However, the `H200`, one of NVIDIA's top-of-the-line data center GPUs, can only provide 141 GB of VRAM [3].

A common approach to overcoming the discrepancy between required and provided VRAM is moving memory between the CPU and GPU. Developers can either do this explicitly, via `memcpy()`-like APIs or use the transparent Unified Virtual Memory (UVM) approach. UVM creates a virtual contiguous address space and transfers data on demand from CPU to GPU memory. This is done in software. A similar, though hardware based approach, is a Unified Memory (UM). With UM the CPU and GPU share physical memory. Traditionally, UM is achieved using a tightly integrated chip combining CPU, GPU and memory on one die. One example of such an architecture is AMD's Instinct `MI300A` Accelerated Processing Unit (APU), which combines 24 CPU cores with 228 GPU Comput Units (CUs) and 128 GB of high bandwidth memory [4]. Another technology used to scale GPU-intensive applications is *NVLink* which creates a point-to-point interconnect between multiple GPUs [5]. This increases not only the GPU computing power, but also augments the effective VRAM. To expand memory only, the open CXL standard exists. It promises scalability by allowing remote memory and, since version 3.0, memory pooling. This means multiple hosts can access the same remote memory bank [6]. Furthermore, CXL was originally designed for host-device communication and memory share.

While these solutions are promising the current state of the art only provides partial solutions, which demand trade-offs: manual memory management is difficult and error-prone. UVM solves this, but compared to the explicit approach, may introduce overheads [1], due to its software-based nature [7]. While a true, hardware-based, UM eliminates this overhead, it is only available in highly integrated APUs. Thus APUs allow for expanding of VRAM, but neither memory, CPU, nor GPU can be replaced or

[1] This is highly dependent on the memory access patterns and kernel design [7]

expanded without replacing the other components. Similarly, *NVLink* does not scale compute resources independently. And while CXL is a perfect fit, little research has been conducted on CXL-enabled GPUs and no hardware is available yet. However, CXL-GPU[8] introduced a real silicon CXL-enabled GPU whose memory can be extended remotely. Though the authors of CXL-GPU do not explore the opportunities of a cache-coherent unified address space.

Thus, there is a need for a system that:

1. Easily expands memory both on the CPU and GPU.
2. Supports moving of data between the CPU and GPU without introducing large overhead.
3. Reduces the complexity of manual memory management for programmers.

We address this need, by extending an APU system for general purpose server hardware using CXL. Our approach creates a cache coherent UM domain spanning the CPU and the GPU, using the capabilities of CXL interconnect protocols. Since CXL enabled GPUs are not available yet, we evaluate the impact of remote memory on GPU–CPU interoperability and performance using a proof-of-concept simulation. This provides a controlled environment, which allows us to model ideal scenarios for variables such as latency. Particularly interesting to us is the comparison of a classical APU model to a CXL simulation. For the proof-of-concept simulation, we leverage existing models and adapt them for CXL. To mitigate latency penalties introduced by remote memory, we employ the MESIF cache coherence protocol on the CPU caches.

We implement our model in the *gem5* simulator and make use of its CPU, GPU as well as precise cache simulation capabilities.

We use the existing VIPER protocol, which we adapt to interface with the CXL domain, to keep coherence on the GPU. For the same purpose, on the CPU side, we implement a directory-based MESIF protocol. We use the *ProtoSLICC* compiler to generate the MESIF protocol.

compiler to implement the CPU coherence protocol. To create the shared memory we employ the CXL protocol and a memory pool. We use the *ProtoSLICC* provided CXL implementation for our simulation.

We evaluate our approach using the Rodinia benchmark suite, a standard for assessing performance in heterogeneous systems. We compare this to a baseline of the default *gem5* APU. We do this to understand the impact CXL has on heterogeneous computing. We investigate the simulated run time, cache hit-rates and Instructions per Cycle. Furthermore we also explore effect of increasing latency on performance.

The contributions we make are threefold:

1. **A Proof-of-Concept Simulation Model.** We develop, to the best of our knowledge, the first simulation model of a fully cache coherent APU interconnected via CXL.
2. **A Comprehensive Performance Evaluation.** We provide a detailed performance evaluation of our CXL-based UM and qualify its performance with comparisons to the default *gem5* APU as baseline.
3. **A Reusable MESIF Implementation.** We implement a reusable MESIF protocol specification using the *ProtoSlicc* compiler, which does not only allow for easy reuse and adaptation, but also allows generation of a coherence bridge component with any arbitrary protocol.

2 Background

2.1 Cache Coherence

Cache coherence protocols ensure a coherent view of a memory location when it is accessed by multiple instances (usually processor cores). In other words, a cache coherence protocol provides rules for how and when to synchronize between caches. This is done to prevent the access of outdated (stale) data. Synchronization may be as simple as invalidation or as complex as supplying an updated value [9], [10].

Cache coherence protocols can broadly be divided into two categories: **snooping** and **directory**-based. The snooping approach uses an arbitrated, shared bus that the cache coherence controllers can monitor ("snoop"). This ensures consistent global ordering of messages, and allows the controllers to infer the global state. This is approach is in contrast to a directory-based, in which a directory keeps track of the global state. The directory is therefore involved in all requests. It issues commands to the caches/controllers to maintain coherence. Directory-based protocols increase scalability by reducing bus contention. In snooping protocols, this is caused by the large number of broadcasts. As more cores are in use, the number of messages on the bus increases dramatically [10].

Regardless of whether snooping or directory-based, cache coherence protocols are usually designed as finite state machines and described by their stable states. However, since most operations can't be performed in an atomic manner in practice, transient states are introduced. Transient states are states which occur while transitioning from one stable state to another. Such a transient state may, for example, occur while waiting for a response from memory. While transient states introduce complexity in both the design and the implementation of protocols, they are an absolute necessity for correct operation [9].

2.1.1 The MESI Cache Coherence Protocol

The Modified-Exclusive-Shared-Invalid (**MESI**) protocol, first introduced in 1984 [11], is a common cache coherence protocol. It improves upon the simpler Modified-Shared-Invalid (**MSI**) protocol [10]. The MESI protocol uses only the stable states it is named after:

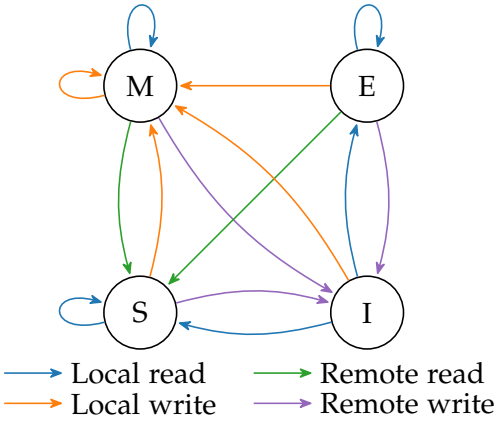


Figure 2.1: Finite state machine of the MESI stable state protocol

Modified (M) State: The M state is the only state that holds dirty data (i.e. data that has been written to). Furthermore it is exclusive. To ensure this, all caches holding the same line have to be invalidated before transitioning to the M state and writing. A line in the M state can be read or written to without further action. When another cache tries to read the line, the cache in the M state must write-back to memory and transition to the S state [9].

Exclusive (E) State: As its name suggests, the E state is exclusive. However, unlike the M state, it holds clean data. A cache transitions to the E state when it reads a line that no other cache holds. Reading from the E state can be done without further action. The E state is the main optimization over MSI as, due to its exclusivity, the cache can transition to M silently, when writing in a E state. This means it does not have to invalidate or inform other caches of this operation. However, if another cache attempts to read, the E state loses its exclusivity and thus must transition to the S state [9].

Shared (S) State: The S state holds valid, clean data. It suggests, though doesn't guarantee, that there are other caches holding the same cache line in the same state. Since the data is always valid and clean, a cache may read without further actions [9].

Invalid (I) State: The I state represents a cache line which does not hold any valid data and thus, if used, must be fetched from memory [9].

Action \ State	M	E	S	I	F
Local Read	*hit!*	*hit!*	*hit!*	GetS → F	*hit!*
Local Write	*hit!*	→ M	Invalidate sharers →M	GetM → M	Invalidate sharers →M
Remote Read (GetS)	Write-back, Forward data →S	Forward data → S	–	–	Forward data → S
Remote Write (GetM)	Forward data → I	Forward data → I	→ I	–	Forward data → I

Table 2.1: Stable state protocol of MESIF adapted from [12]

2.1.2 The MESIF Cache Coherence Protocol

The Modified-Exclusive-Shared-Invalid-Forward (**MESIF**) protocol, which we use as our CPU coherence protocol, was introduced by Intel as an extension of MESI. It improves upon MESI by introducing the Forwarding (**F**) state. The F state, a specialized version of the S state, reduces the time a cache miss takes. It accomplishes this by enabling neighboring caches to share data with each other, instead of fetching it from memory. MESIF was designed to take advantage of Intel's QuickPath interconnect, achieving a two-hop cache-to-cache latency, when used with it [12]. Furthermore, it was designed to enhance performance in a distributed memory system. The following invariants have been identified to maintain coherence with the MESIF protocol [12]:

1. There must always be exactly one forwarder per cache line. However, note that if no cache holds the line in a forwarding state, the forward property is implicitly at the home node (i.e., the directory or memory).
2. Only the forwarder can award Data and access permissions.
3. The forwarded value must be the same as the source value just before the start of the forward.
4. There can be either a single writer *or* multiple readers.

These invariants are upheld by the MESIF state machine. Just like in the MESI protocol, both the E and M states are exclusive. Thus they easily act as forwarders. When multiple caches share the line, one cache is marked as the forwarder. This cache is in

the F state, which, just as the S state, holds only clean data. Whenever data is forwarded to another cache, the forwarding property is passed along with it. Thus, a cache in the F state transitions to the S state on a remote read, while the reading cache transitions from S to F [12]. This can be seen in the stable-state protocol shown in Table 2.1.

2.1.3 Memory Consistency

Cache coherence alone is not enough to ensure parallel programs behave as expected. Cache coherence only performs operations on a single cache line. Memory consistency, in contrast, is needed to ensure proper ordering between cache lines. Thus cache coherence provides the foundation needed for Memory consistency. However full memory consistency requires additional hardware mechanisms.

There are multiple consistency models, such as sequential consistency. It is the most intuitive, though most restrictive, consistency model. With sequential consistency, the result of execution is the same as if all operations were executed in some, globally agreed-upon, order. Additionally, each operation issued by a core is performed in the order specified by its program. This is a so-called strong memory ordering model [9]. In contrast are weak consistency models, also called relaxed memory order ease the aforementioned requirements to allow optimizations. One such model is the so-called release-consistency model, which introduces two operations: `acquire` and `release`. Release-consistency guarantees that each operation before a `release` becomes visible to any core performing a `acquire` after this point [9].

It is important to note that while memory consistency is usually defined by the Instruction Set Architecture (ISA) and known to the program, the cache coherence protocol is transparent and may differ by processor model.

2.2 The *gem5* simulator

The simulator we use for our proof-of-concept APU model is *gem5*. *gem5* is an open-source simulation framework for exploring and evaluating new hardware possibilities. It has found widespread acceptance in academia and industry, where it is used to explore and evaluate in the early stages of the design process [13]. *gem5* is the result of a merge of the *M5* and *GEMS* simulators and allows for two simulation modes[14]:

FS mode: The full system (FS) mode simulates the entire OS stack. This allows for a detailed analysis of performance, especially regarding system-level interactions like context switches or IO.[14]

SE mode: In system-call emulation (SE) mode *gem5* simulates only the targeted program, while interaction with the OS is emulated. This allows for greater simula-

tion performance, making it perfect for ideal application-level analysis. This is particularly useful in scenarios where OS interactions are not of interest or are limited.[14]

gem5 supports simulating multiple architectures, such as `x86`, `ARM`, and `RISC-V`. It provides models with varying levels of speed and accuracy [14]. Most interesting for this thesis is *gem5*'s Ruby cache subsystem. Ruby allows for defining custom cache coherence protocols for the CPU and the GPU. SLICC, a Domain Specific Language (DSL) for describing the state machine of cache coherence protocols is employed for this purpose. SLICC works at cache block granularity, as to ease the definition/implementations of protocols [14]. However, implementing SLICC protocols is challenging, because a lot of boilerplate code must be written, and states[1] and their transitions must be explicitly defined [14], [15]. The gem5 simulator provides out-of-the-box implementations of popular cache coherence protocols, such as a MESI protocol with three caching levels [13].

gem5 offers a variety of default implementations for coherence and multiple models for CPUs, GPUs, and many other devices. Its modular design and Python-based configurations using allow for easy configuration and extension of simulator [13].

2.2.1 GPU simulation in *gem5*

gem5 allows for the simulation of General-Purpose Computing on Graphics Processing Units (GPGPU) tasks. The used model was first introduced with support for AMD's Graphics Core Next (GCN)3 architecture. Though it now supports the Vega ISA[2] [16]. However the model is flexible enough to allow extension to other ISAs [13]. In its execution model, threads are combined into wavefronts; in CUDA, these are called warps. These, in turn, are combined to form work groups, also known as thread blocks in CUDA. A GPU kernel may consist of multiple thread blocks. In hardware terms, a wavefront maps to a single SIMD unit, and a work group maps to a CU [13].

The GPU model works in both FS and SE modes. In both cases AMD's ROCm software stack is used. Note that reliance on ROCm limits the CPU side to the `x86` architecture. In SE mode, the driver is emulated; in FS mode, the entire stack, including the drivers is simulated [17]. The FS mode is primarily used to simulate discrete GPUs, while the SE mode is used to simulate APUs.

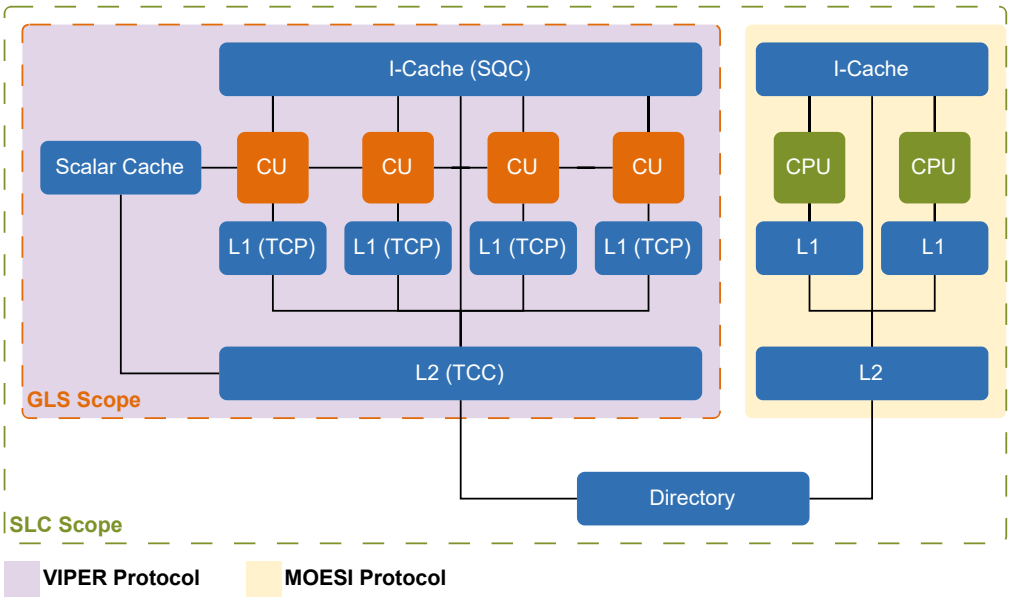


Figure 2.2: Overview of the gem5 APU model adapted from [18]

2.3 The GPU VIPER Protocol

The GPU VIPER protocol is the cache coherence protocol used in *gem5*'s GPU simulation. It provides cache coherent Unified Memory APU [18], but also allows for simulation of discrete GPUs [16]. In addition to the cache coherence protocol, VIPER includes a a wavefront-level coalescer. The coalescer is used to coalesce, meaning bundle, multiple spatially related memory reads or writes [18].

Within Ruby, VIPER manages coherence between multiple GPU caches: the private L1 caches (Texture Cache per Pipe/TCP by AMD), the shared L2 (Texture Cache per Channel/TCC), the shared instruction cache (Sequencer Cache/SQC), and the scalar cache. VIPER uses a scoped release-consistency model, with two scopes, System Level Coherence (SLC) and Global Level Coherence (GLC) [18]. It does so in accordance with the GCN ISA, which calls for SLC and GLC bits [19]. The extent of the scopes can be seen in Figure 2.2.

By default, VIPER uses a write-through scheme, meaning that when a write occurs, the cache line is invalidated and the write passed on to the higher cache or memory [18].

[1]This includes transient states
[2]Note Vega is a superset of GCN3

Though a write-back TCC option was introduced to enable multi-GPU support [20]. This write-through approach allows for instant writes, meaning the TCP does not wait for permission or acknowledgment. Similarly, atomics are also performed immediately and in memory. In case they are GLC-scoped and write-back mode is enabled they are performed in the TCC [18], [21]

The VIPER protocol doesn't make use of the `acquire` and `release` events provided by Ruby. Instead, the corresponding fences are mapped differently. The coalescer initiated `invTCP` walks and invalidates the entire TCP, performing an acquire fence operation. A release fence is performed by waiting for all outstanding write and atomic callbacks [18].

VIPER was designed to integrate with AMD's Modified-Owned-Exclusive-Shared-Invalid (MOESI) implementation and its stateless directory. The MOESI protocol works similarly to the MESIF Protocol but instead of a clean F state, it uses a dirty O state.

2.4 The *ProtoSLICC* Compiler

ProtoSLICC is a recently proposed compiler designed to simplify the development of cache coherence controllers for Ruby subsystem of *gem5*'s. Traditionally, writing a coherence controller in SLICC required extensive boilerplate to describe stable states and transient states, their interactions, as well as message types. This process is not only time consuming but also error-prone [15].

ProtoSLICC adress this by introducing a DSL to define the atomic specifications of the cache coherence protocol. Furthermore, the DSL specifies protocol and architecture parameters. Using this the compiler generates transient states and the state-machine implementations. Additionally *ProtoSLICC* generates the *gem5* Python configuration script with the provided parameters [15].

ProtoSLICC builds upon the earlier *ProtoGen* tool, which is limited to generating a private cache and directory only [22]. Furthermore, *ProtoSLICC* improves upon *ProtoGen* by compiling to SLICC and generating state machines for a broader range of caches. Specifically, it generates the L1, L2, and private instruction caches, in addition to the directory [15].

By automating the generation of transient state and multiple levels of caches, *ProtoSLICC* eases the development of cache coherence controllers, which would otherwise be quite laborious.

2.5 The Compute Express Link

The Compute Express Link (CXL) standard, released by the CXL consortium, is used as an interconnect between a host CPU and devices such as GPUs or accelerators. Intel first released it in 2019 [23]. Since then multiple major industry players such as AMD and NVIDIA have joined the consortium's board of directors [24]. CXL was developed to enable cache coherent memory access between devices and host memory. However, it has evolved to enable memory pooling with multi-level switching and memory expansion [25]. Although it is built atop the existing Peripheral Component Interconnect Express (PCIe) physical infrastructure, CXL provides lower latency than PCIe [23]. CXL achieves this with three sub-protocols:

CXL.io: Based on PCIe, CXL.io is used for IO related tasks such as device discovery, error reporting, and Direct Memory Access (DMA). It is the only mandatory protocol for any CXL device [6], [23].

CXL.cache: To allow a device to cache host memory CXL.cache utilizes a MESI coherence protocol with 64-byte cache lines. Thus a device can read or write locally cached copies of data residing in host memory, while remaining coherent with the host [23].

CXL.mem: The CXL.mem protocol enables the use of so called Host managed Device Memory (HDM). HDM can be of different types: HDM-H, where coherence is only affected by the host; and HDM-D, which allows the device to affect coherence. HDM-H's is used for simple memory expenders, while HDM-D is used for accelerators, such as GPUs. CXL version 3.0 introduces HDM-DB, which extends HDM-D with back-invalidation. This enables the support of multi-host and disaggregated environments [23].

Using these protocols, the CXL standard defines three device types [6]:

Type 1 devices only use CXL.io and CXL.cache. They cache host data, but don't have HDM. A typical example of a type 1 device is a Network Interface Controller (NIC).

Type 2 devices use CXL.io, CXL.cache and CXL.mem; thus they provide fully coherent HDM. An accelerator, such as a GPU, fulfills these requirements.

Type 3 devices use CXL.io and CXL.mem, thus function as a memory buffer.

Together these device types and protocols form a versatile standard for unified memory in heterogeneous systems. They even allow for memory expansion, remote memory

and memory pooling [6]. For this thesis we use both CPU and GPU as CXL hosts. Thus both use CXL.mem to communicate with the CXL memory. Due to the multi-host nature we need to use a HDM-DB memory device

Note that in this thesis we will refer to CXL.mem and CXL.cache operations by their canonical names, where possible. This means we will refer to `GetS` instead of `MemRd` with a meta field value of `A` or `GetM` instead of `MemRd` with a meta field value of `S`.

3 Overview

Our proof-of-concept APU-like architecture integrates CXL to create a cohesive, high-performance system. It mitigates the high latency associated with CXL remote memory. More importantly it enables scalable, unified memory pooling.

3.1 Design Goals

We designed our model with the following design goals in mind:

1. **High-Efficiency Data Transfers:** Achieve minimal latency for transferring data between CPU and GPU.
2. **Independent Memory Scalability:** Enable the system to scale the memory resources of both the CPU and GPU independently of their compute power.
3. **Platform Independence:** Make use of open technology so that our model can be applied independently of and across vendors.
4. **Transparent Memory Management:** Minimize the manual memory management and movement between devices, to enable developers to focus on application logic.

Adhering to these goals addresses the needs of modern heterogeneous systems that strive to be not only efficient, but also scalable.

3.2 System Overview

As seen in Figure 3.1 our architecture can be split into three distinct components: The CPU, the GPU and the CXL domain. The CPU and GPU domains contain their respective computational units and caches. In contrast to this the CXL domain contains a CXL directory with CXL memory. However, each of these domains utilizes its own cache coherence protocol. The CPU uses the MESIF protocol, and the GPU employs the VIPER protocol. The CXL domain utilizes the CXL.cache and CXL.mem protocols,

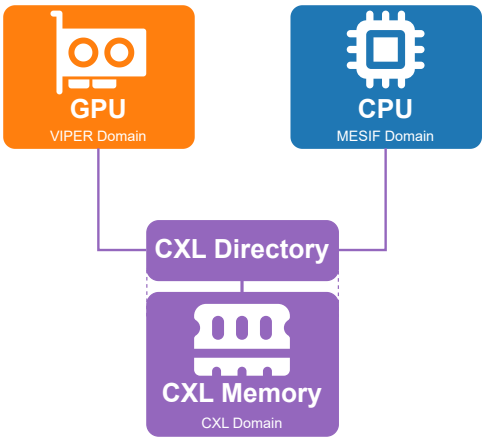


Figure 3.1: Overview of the CXL APU design

which are based on the MESI protocol. We use the MESIF protocol for the CPU coherence and we elected VIPER for the GPU coherence.

Since we only model remote memory, both the CPU and GPU are hosts in the view of CXL. The directory with the attached memory forms the CXL device[1].

3.3 System Components

As mentioned above, our model consist of three primary components, each with a distinct function:

CPU/MESIF Domain: The CPU domain employs the MESIF cache coherence protocol. We make use of MESIF's forwarding property to reduce the latency introduced by the CXL memory. With MESIF, the CPU adheres to a typical coherence model to meet the expectations of the programmer. As this is an `x86` CPU, a strong memory order is in use.

GPU/Viper Domain: The GPU domain uses *gem5*'s VIPER GPU protocol to maintain coherence. In contrast to the CPU domain it utilizes a more relaxed memory order. This is in accordance to the GPU's highly parallel nature and maximizes its throughput.

[1] In this scenario the device is a memory expand, thus a type 3 device.

CXL Domain: The CXL domain acts as fabric, connecting both the CPU and GPU domains coherently using the CXL.cache and CXL.mem protocols. Furthermore, the CXL domain also provides the remote memory used by both the CPU and GPU. Thus, all memory interactions happen through CXL.

3.3.1 System Workflow

In our proposed architecture, the typical workflow of an application proceeds as follows: First, an application is launched on the CPU. Next, space is allocated on the shared, remote memory pool for the data the application needs. This data is subsequently either loaded from storage or generated dynamically. Because the data is located on shared CXL memory, it is coherently accessible to the CPU and GPU. The CPU's multiple cores share this data among themselves using the MESIF protocol, so the CPU only has to load from the CXL-domain once.

Once data preparation is complete, the application launches the GPU kernel. Since the data resides in the shared memory or is coherently cached, no memory transfer has to happen to make it available to the GPU. The GPU acquires the data directly and as needed via CXL. The GPU's cache hierarchy uses VIPER to move data between caches and keep coherence between CUs.

If the GPU produces results it wants to make available to the CPU, this can be achieved through a simple store operation marked as SLC. This propagates the dirty data throughout the CXL domain. This can happen at the GPU kernel end, as in traditional approaches, or at arbitrary times. The CPU can access these results at any time, again via CXL. The CPU will cache any accessed data locally. As before, cores share this data with other cores to avoid re-fetching from remote memory.

From the programmer's perspective all data movement between the CPU and GPU is expressed by regular memory operations without the need for explicit transfers. Furthermore, application designed for memory architectures with segregated address spaces can still benefit from the shared memory: when a runtime, such as CUDA, is issued a device-to-device copy, it can perform a simple memory copy within the shared memory pool instead of an expensive transfer over PCIe.

4 Design

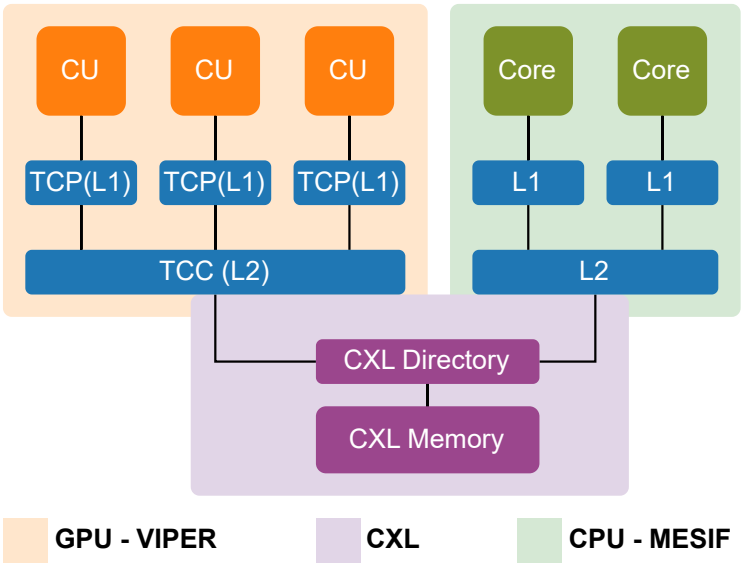


Figure 4.1: CXL-APU Architecture

Unlike in a typical APU architecture or the default *gem5* APU model, our CXL-APU model does not use a single coherence protocol for both the CPU and GPU. This can be seen in Figure 4.1, which also illustrates the independent nature of the CPU and GPU. They never communicate directly and only interact via CXL. Therefore, we will introduce, both the GPU and CPU domains independently and explain how these interact with CXL, providing a full picture of our model.

4.1 Choice of CPU Coherence Protocol

We selected the MESIF protocol for our CPU cache coherence protocol due to its ability to reduce latency caused by our use of CXL remote memory. MESIF accomplishes this by leveraging the forwarding property. By forwarding data already read by a local cache, MESIF minimizes the expensive memory accesses. This is an optimization

even in regular memory systems. However, it can lead to greater improvements in our remote memory context, where memory accesses have even higher latency. This contrasts with the MESI protocol, which would require reading from memory and thus leading to slowdowns. To implement the MESIF protocol we generate it based on its specification using the *ProtoSlicc* compiler.

Furthermore we adapt the by-the-book MESIF implementation to a directory-based approach.

4.1.1 Discussion of Directory vs Snoopy

Since this choice is nontrivial, we compare our directory approach to the classic Snoopy approach below.

A directory-based approach increases the cache-to-cache latency from the two hops (requester $\rightarrow$ forwarder $\rightarrow$ requester), as mentioned in Subsection 2.1.2, to three (requester $\rightarrow$ directory $\rightarrow$ forwarder $\rightarrow$ requester). It also introduces additional storage overhead, since the directory must store global state. However, this approach simplifies our implementation, because we can implement our bridges at the directory level, which already contains a state machine. Although MESIF was designed to be more scalable than traditional snoopy protocols [12], the directory will still improve its scalability. Changing the multicast to a unicast minimizes network traffic. The impact of broadcasts scales not only with the number of cores, but also with the latency between them. Thus, the directory-based approach is even more advantageous in multi-node environments. Furthermore, coherence interconnects are usually not present between sockets in multi-socket environments. In this scenario, the directory approach eliminates the need for a hybrid approach or snooping filters.

4.2 Choice of GPU Coherence Protocol

We elected to use the VIPER Protocol for the GPU coherence because it is the only currently implement coherence protocol for the *gem5* APU. Furthermore, the *gem5* GPU model requires the usage of specific semantics. However this is, if documented, only sparsely so, making the implementation of a new coherence protocol hard. For example, for the acquire fence operation, the protocol relies on an `InvTCP` [18] event generated by the coalescer instead of the Ruby-provided `Acquire` operation. The release fence is even less portable. Instead of using the Ruby-provided `Release` operation, the model waits for all outstanding requests to finish [18]. This is also done by the coalescer. Moreover, the VIPER coherence protocol itself is barely documented, and, to our knowledge, dose not have a formal definition. Nevertheless we still need to bridge the VIPER domain to the CXL domain. We do so manually, as using the *ProtoSlicc*

compiler would require creating a formal definition of the protocol. Furthermore this would also require *ProtoSlicc* to be either adapted to the VIPER coalescer, or a new coalescer to be implemented. This is beyond the scope of this thesis.

4.3 Coherence Bridges

As mentioned above both, the CPU and GPU domain need to interact with the CXL domain in a coherent manner. Thus, we introduce coherence bridges.

These bridges are hardware components that translate states and messages from one coherence protocol to another. They accomplish this by maintaining a separate state for each domain. This is ensures that the invariants of both protocols are preserved, and neither protocol has to be adapted. Just like the cache coherence protocols, we implement bridges using finite state machines and denote a bridged state as follows: `E_C_M`. In this example, the left protocol is in the E state, andthe protocol to the right of the bridge is in the M state.

4.3.1 Example: SI–MSI Bridge

To further the understanding of coherence bridges we will introduce the example of bridging a simple Shared-Invalid (**SI**) protocol to a Modified-Shared-Invalid(**MSI**) protocol.

the SI protocol's S and I states are trivially translated into the MSI protocol; the `S_C_S` and `I_C_I` states represent this translation. However, when data is written in the SI protocol, it is immediately committed either to memory or a higher cache in the hierarchy. This behavior is not mirrored in the MSI protocol. While it is possible to match this behavior in the bridge (e.g., by issuing a `GetM` immediately followed by a `PutM`), this is inefficient. We can optimize this, by making our bridge a cache. While the bridge holds the dirty data, it is in the `M_C_S` state, appearing clean to the SI domain and dirty to the MSI domain.

Now that basic operation of this bridge is clear, we will review the sequence of events shown in Figure 4.2. There is exactly one MSI cache, one SI cache, and the bridge. Initially only the MSI cache holds the clean data, as seen in 1. The SI cache then attempts to load but misses. It sends a `GetS` to the bridge, which, as it also does not hold the data, reads from memory. After the memory read, in 2, the bridge holds clean data and transitions to `S_C_S`. The bridge sends the data to the requesting cache with a `GetS_Ack`, and the SI cache transitions to the S state. In the next timestep 3, the SI cache writes. As there is no dirty state in the SI protocol, it sends the dirty data to the bridge. The SI cache transitions to the I state. Since the MSI domain does

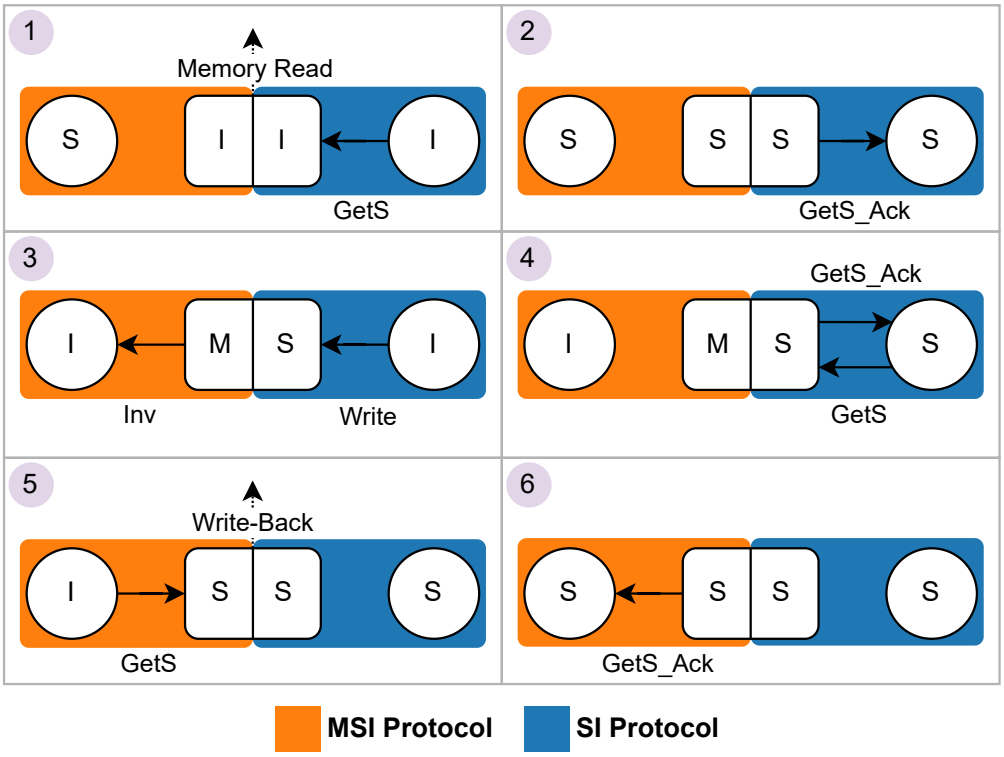


Figure 4.2: Sequence of events with a MSI–SI bridge

have a dirty state, the bridge transitions to the `M_C_S` state and now holds dirty data. However, before this can happen the MSI cache must be invalidated, because the M state is exclusive. Following this, in 4, the SI cache again tries to load and sends another `GetS` to the bridge. The bridge can respond to this with a `GetS_Ack`. The MSI domain remains unaffected. In 5 the MSI cache tries to read. It requests access by sending a `GetS` to the bridge. However, since the bridge contains dirty data, it first must write-back to memory. This causes the bridge to contain clean data and move to the `S_C_S` state. The SI side of the bridge is unaffected since, to it, the data was always clean. Finally, in 6 the bridge sends the data alongside its acknowledgment to the MSI cache, causing it to transition to the S state.

5 CPU Implementation - Directory-Based MESIF

In this section, we discuss and explain the MESIF implementation we use for our CPU model. We use *ProtoSLICC* to generate SLICC code. We have opted for a directory-based approach. To better model modern day systems we also choose an unordered interconnect network.

5.1 Basic Operation

First, we introduce the basic operation of our implementation of this protocol.

When a cache requests shared or exclusive access from the directory, the directory forwards the request to the cache with forwarding property (i.e., the cache in the F, E or M state). We refer to the regular requests to the directory as `GetS` or `GetM`, while we call the forwarded messages `Fwd_GetS` or `Fwd_GetM`. A cache that receives a forwarded message will send ("forward") its data to the requester. Then, it moves to either the S or I state, as can be seen in Table 2.1. In the case of a `Fwd_GetM`, the directory will provides the number of sharers to the forwarder and sends invalidation (`Inv`) messages to each of them. The forwarder, in turn, sends the number of sharers alongside its data to the requester. The requester has to wait for the appropriate number of invalidate acknowledgments (`InvAck`) before proceeding. Additionally if a cache is in the M state and receives a `Fwd_GetS` it must write-back its dirty data. The directory expects and waits for this write-back. However, the data is forwarded immediately. In the cases of `Fwd_GetM`, the dirty data is not written back; instead it forwarded as dirty. Thus, the inevitable write-back becomes the responsibility of the new M state cache.

To implement this, our directory has to mirror the states of the caches in the directory. We accomplish this with the following directory states:

M/E: A single cache holds the line exclusively. The cache-line is either dirty (M) or clean (E).

F: One cache is the designated forwarder, and other caches may share the line.

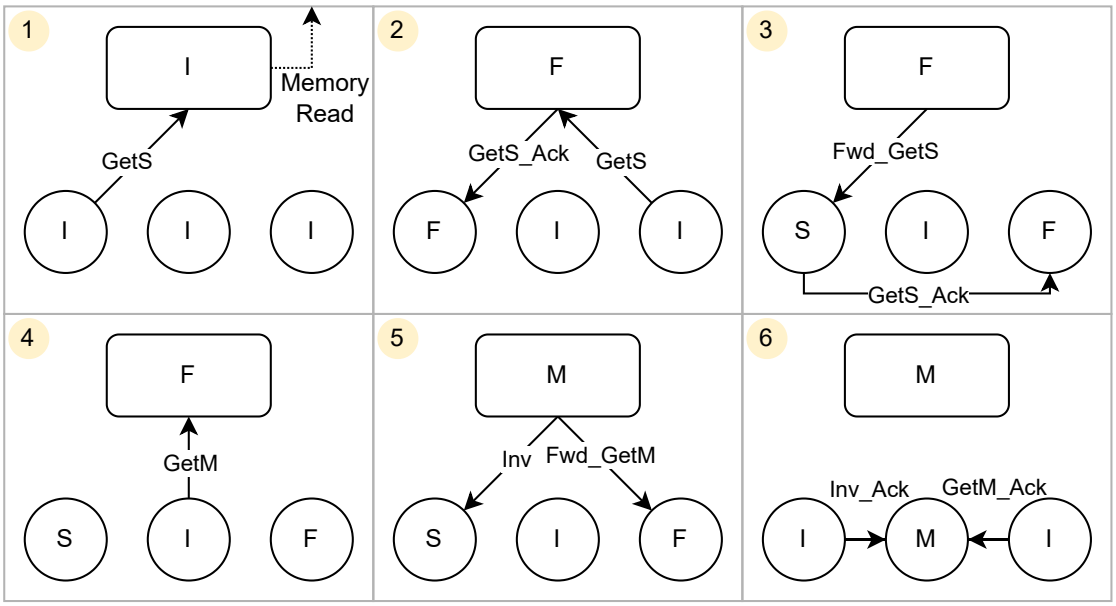


Figure 5.1: Example sequence of operations of the MESIF coherence protocol

S: There is at least one cache holding the line in the S state. However, there is no designated forwarder. Thus the directory has to provide data. This is done by fetching from memory.

I: No cache holds the line.

But to send forwarded requests and invalidations to the correct caches, the directory also needs to keep track which cache is the forwarder and which are sharers. Thus, when forwarding a request, the directory records the requester as the new forwarder. In the case of a `GetS`, the directory adds the old forwarder to its list of sharers.

To keep the forwarder reference and sharer list up-to-date, the directory also must be informed of evictions. To accomplish this a cache will send a message to the directory, indicating its eviction and previous state. We use `PutS`, `PutE`, `PutF`, and `PutM` as these messages. As all but the M state hold clean data only on `PutM` data is written to the memory by the directory. This data is send alongside the `PutM` message.

5.1.1 Example: Operation of Directory-Based MESIF

With the basic operation of the employed directory-based MESIF protocol explained, we provide a simple example to illustrate its function. Note, however, that this operation is simplified, as will become apparent later.

In the initial state 1 all caches and the directory are in the I state. The first cache sends a `GetS` to the directory, because the corresponding core issues a load. Since the directory itself does not contain any valid data, it reads from memory. After the reading, in 2, the directory sends a `GetS_Ack` to the requesting cache. Alongside this message, the data it just fetched is sent. Since the directory knows the requesting cache will transition to the F state, it, too, moves to the F state. Furthermore, the directory keeps track of which cache the forwarder is. In this case, it is the first cache. After receiving the data the requesting cache moves to the F state, as anticipated by the directory. This completes the first cache's load. At the same time, also in 3, the third cache also initiates a load operation, by sending the `GetS` message to the directory. Again, the directory forwards this `GetS` request to the forwarder (the first cache) via the `Fwd_GetS`, as seen in 3. The directory now knows the third cache is the new forwarder. The original forwarder sends the `GetS_Ack` directly to the requesting cache. This again can be seen in 3. The first cache moves from the F to the S state and the third cache from the I to the F state. This completes this interaction.

In 4 one can see the second cache send a `GetM` request to the directory. The directory then responds by forwarding this request to the designated forwarder (cache three). Furthermore, it sends an invalidation request to the first cache. This can be seen in 5. As mentioned above, the directory sends the number of sharers, in this case one, alongside its `Fwd_GetM` message. In 6 this amount is sent to the second cache with the requested data via the `GetM_Ack` message. The second cache receives this acknowledgment and waits until all shares are invalidated. To be sure this is done it relies on the number of shares it was informed about. In this case, the second cache receives the only `Inv_Ack` after the first cache invalidates itself. Thus, the second cache can transition into the exclusive M, state while both the first and third caches move into the I state, concluding this example.

5.2 Introducing Unblock

If further care is not taken, the protocol described thus far can lead to violations of the invariants mentioned in Subsection 2.1.2. Consider the situation as shown in Figure 5.2: In the initial timestep 1, the first cache is in the F state and wants to write; thus, it sends a `GetM` to the directory. In the next timestep, 2, the second cache wants to read. As it is in the I state, it sends a `GetS`. While this occurs the directory transitions to the M state and sends a `GetM_Ack` to the first cache, to allow it to write; however, this message is delayed. In 3, the directory handles the second caches `GetS` by transitioning to F, and sending a `Fwd_GetS`. Due to the unordered nature of the interconnect used,

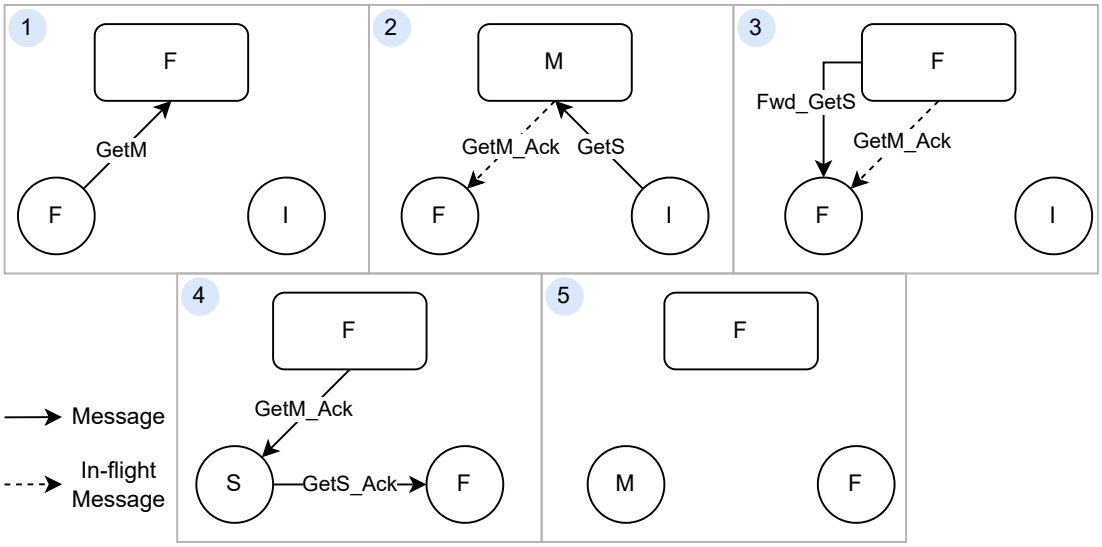


Figure 5.2: Problematic sequence of events in a `Unblock` free MESIF implementation

this message may "overtake" the original `GetM_Ack`. Since there is no guarantee that the `GetS` happened after its request, the first cache handles the `Fwd_GetS` as per the protocol specification and forwards its data to the other cache. This can be seen in 4. Also in this timestep, the first cache receives the acknowledgment of its `GetM` and thus transitions to the M state. This final state, seen in 5, poses a deadlock. This is because the directory is waiting for a write-back from the first cache since it believes it evicted an M state with the message sent in 3. Furthermore, it also violates two invariants of MESIF as defined in Subsection 2.1.2: by having multiple caches in forwarding states it violates the only one forwarder invariant. It also violates the single writer, multiple reader invariant, by having a modified state (writer) and a forward state (reader) at the same time.

To prevent this illegal state from occurring we introduce an `Unblock` message, which is sent from any cache giving up its forwarding property to the directory. While waiting for the `Unblock` message the directory stalls all other request on the same cache line. However, when a M state cache receives `Fwd_GetS`, the cache sends a write-back (`WB`) containing the dirty data. In this case the `WB` messages doubles as an `Unblock`.

5.3 Introducing Put Acknowledgment

While the `Unblock`, introduced in the previous subsection, fixes one deadlock, it introduces another. Consider the situation portrayed in Figure 5.3. In 1 the first cache,

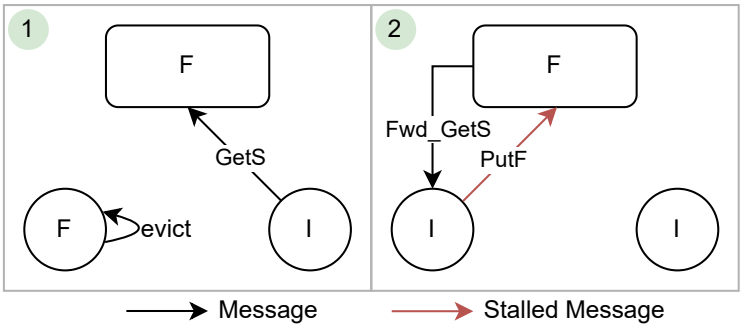


Figure 5.3: Problematic sequence of events in a `PutAck` free MESIF implementation

which is in the F state, triggeres the eviction of its line. Simultaneously, the second cache sends a `GetS` message to the directory. The next timestep, 2, shows the `Fwd_GetS` forwarded by the directory and the `PutF` message sent by the evicting cache. However, the `PutF` is stalled at the directory, because it is waiting for an `Unblock` message. Furthermore, the `Fwd_GetS` message is received in an I state, which it cannot forward.

To avoid this state, we introduce the put-acknowledgment messages. Like the `Put` messages, these are state dependent. This means the directory will respond to a `PutF` with a `PutF_Ack`. A cache may only transition to the I state after receiving such a message. This also means that a cache has to fulfill its forwarder property even while attempting to evict. In the scenario seen in Figure 5.3 at 2, instead of moving to the I state directly, the first cache will be in the transient state F_evict (i.e. in a F state waiting for an acknowledgment of its eviction). When receiving the `Fwd_GetS` it will forward its data and move to the S_evict transient state. Then finally after the cache receives the acknowledgment from the directory, the can move to the I state.

5.4 Silent upgrade

An important optimization preserved from the MESI protocol is the silent upgrade from the E to the M state. This operation does *not* involve the directory. Consequently, the directory may be in the E state while the cache is in the M state. Since the E state is exclusive, this does not break coherence. However, when in the E state, the directory must take special care; it needs to expect a `PutE`, as well as a `PutM`, in case the cache has silently transitioned to the M state. As mentioned in Section 5.2, a `Unblock` after a cache receives `Fwd_GetS` in the E state. In the M state, however, this is replaced by a `WB`. Thus, the directory must handle both situations if it is in the E state.

5.5 MESIF–CXL Bridge

MESIF State	CXL State
M	M/E
E	E
S	S
I	I
F	S

Table 5.1: Unoptimized bridge mapping between MESIF and CXL MESI

In order to share data cache coherently with the memory pool, we need to bridge our directory-based MESIF protocol to the CXL-domain. Due to CXL itself being MESI-based and MESIF being an extension of MESI, this can be achieved with minimal effort.

Table 5.1 shows the mapping between the states of the MESIF protocol and the CXL MESI-based protocol. All stable states of the MESIF protocol map directly to their respective MESI counterpart except for the F state. There is no direct equivalent of the F state in the MESI protocol. However, since the F state is clean, it can be translated to the MESI S state.

The bridge also acts as a higher-level cache to the MESIF domain and thus functions as a MESIF directory. By serving as the sole point of interaction between the CXL and MESIF domains, the bridge ensures all invariants of the MESIF protocol are fulfilled. For this the bridge can take on the role of the forwarder. It can do so even if it does not hold data, by translating any request into the MESI domain.

Optimizations

The bridge functions as a cache itself and can store data not held by lower MESIF caches. This enables us to optimize the amount of memory access. For example, when a MESIF cache relinquishes its M state, the bridge transitions to `M_C_I`, as it does not need to write-back data. It can remain in this state until this line gets replaced or is otherwise evicted. This allows us to read from the bridge if another access occurs on the same line, instead of reading from memory. Additionally the bridge can give the exclusive E state while itself being in the M, E or S states in the CXL domain. It then moves into the `M_C_E`, `E_C_E`, or `S_C_E` state.

5.6 Implmentation using *ProtoSLICC*

We use *ProtoSLICC* to implement the cache and directory behavior for the *gem5* simulator. We specify the cache and directory behavior using the *ProtoSLICC* DSL. Thus, we only specify the stable state protocol, while transient states are generated automatically. However, manual intervention in the SLICC code is needed to uphold the forwarding in transient states, as mentioned in Section 5.3. This is a limitation of ProtoSLICC; transient state transitions cannot be specified explicitly. Though, this only requires a simple change of a transition. Instead of remaining in the forwarding transient state (e.g., F_evict), we transition to its non-forwarding counterpart(e.g., S_evict). Furthermore, by specifying the CXL protocol we can also generate the bridge between CXL and MESIF.

In summary, MESIF meets our need to hide memory latency. Furthermore, our directory-based MESIF implementation effectively addresses the scalability challenges of traditional MESIF implementations. With *ProtoSLICC*, our implementation can easily be realized and bridged to CXL to suit our needs.

6 GPU Implementation - VIPER–CXL Bridge

The primary challenge of integrating VIPER into our simulation model is bridging it to the CXL-domain, because it is already implemented in the gem5 simulator. Unlike MESIF, the VIPER protocol is not based on MESI. It is fundamentally different, making it more difficult to bridge to the CXL-domain. Furthermore, to our knowledge, the VIPER protocol does not have a formal definition; thus, we cannot easily define it for the *ProtoSLICC* compiler. Therefore, we will create the bridge manually.

We bridge the VIPER and CXL protocols at the TCC, the shared L2 cache of the GPU. This is the only point of interaction between the GPU domain caches and the CXL domain. As explained in Section 2.3, the VIPER protocol uses the System Level Coherence (SLC) scope, which is coherent with the CPU, and the Global Level Coherence (GLC) scope, which is only coherent with the other caches in the GPU. The scoped nature is integral to creating a bridge between VIPER and CXL. However, to understand our bridge one must first understand the stable states of the VIPER TCC:

1. **Invalid (I):** The cache holds no valid data.
2. **Valid (V):** The cache holds valid, clean data.
3. **Modified (M):** The cache holds dirty data.
4. **Written (W):** The cache contains dirty data marked for write-back. This state cannot be read, but it can be amended with another write. Atomics are not allowed.

In the original VIPER implementation, only GLC operations are cached in the TCC[1]. Thus all cached values in the TCC are non-coherent to the directory. Furthermore, all states, except the I state, can only be caused by a GLC operation. This is because the directory performs SLC reads, writes, and atomics. Additionally, SLC messages invalidate all caches, including the TCC, on their path.

With this information, we can create our VIPER TCC bridge to CXL. Note, however, that we do not include the W state in our bridge. The W state is an optimization that

[1]However, in case the `WB` property is *not* enabled, stores are instantly committed to memory

represents a deferred write-back. CXL provides a system-coherent M state, which, on a remote-CXL `GetS` will write-back, thus making the W state redundant. Furthermore, excluding the W state simplifies the state machine of our bridge.

In the following chapter, we will denote the bridge state as `VIPER State_C_CXL State`.

6.1 Basic Bridging

Unlike the default VIPER implementation, we opted to cache data, gained by operations in SLC scope, in the TCC. This helps us overcome the remote memory latency. Thus, we translate an invalid state caused by an SLC load to a coherent clean state in the CXL domain. Consequently, after an SLC load, we are either in the `I_C_S` or, if there are no other sharers, the `I_C_E` states. In a similar fashion, we bridge and improve upon the SLC store by using the CXL coherent M state. We thus cache dirty data coherently and transition into `I_C_M`.

While mapping SLC actions is straightforward, the system-incoherent GLC operations require special care. These operation contain data which is not coherent with the CPU. Though, data can still be loaded from memory. To tackle this we introduce a new present (P) state. The P state is a CXL side state that we use to indicate data is present but not coherent within the CXL domain. Crucially, from the perspective of others in the CXL domain, a cache in the P state is considered invalid. This state is necessary because any other valid CXL state would cause the data to be system-coherent. With this new state, a load in the GLC scope must first acquire data in a non-coherent manner (denoted in the table as GetP), unless the bridge is already in the P state. To do so, we perform a non-cachable load, realized in the CXL.cache protocol using the `SnpCur` (Snoop Current) message [6]. The bridge then transitions to the `V_C_P` state. Similarly, after a store, the cache moves into the `M_C_P` state.

6.2 Mixing SLC and GLC

In order to properly mirror the functionality of VIPER, we also must handle a mixed sequence of GLC and SLC scoped operations.

All transitions in the following section as well as those already described above can be observed in Table 6.1 and Table 6.2. In each table, the rows represent the request initiated by the CU, while the columns represent the current stable state of the bridge.

Action \ State	V_C_M	V_C_E	V_C_S	V_C_P	M_C_P	M_C_E
SLC load	→I_C_M	→I_C_E	→I_C_S	GetS →I_C_E/I_C_S	WB+GetS →I_C_E/I_C_S	→I_C_M
SLC store	→I_C_M	→I_C_M	GetM →I_C_M	GetM →I_C_M	GetM →I_C_M	→I_C_M
GLC load	*hit!*	*hit!*	*hit!*	*hit!*	*hit!*	*hit!*
GLC store	WB+PutM →M_C_P	→M_C_E	PutS →M_C_P	→M_C_P	*hit!*	*hit!*

Table 6.1: State mapping table, while the VIPER–CXL bridge is in a GLC state

6.2.1 GLC scoped operations after SLC scoped operations

For a GLC load performed after SLC operations, the TCC already contains valid data (i.e. the CXL side of the bridge is not in the I state). We can avoid losing the permissions of these states by moving directly into a corresponding V_C_M, V_C_E or V_C_S state. However, note that if an invalidate occurs on the CXL side in any of these states, we will either acknowledge or write-back. Then, to keep the GLC-specified non-coherence with the CPU, we move to V_C_P.

A GLC store after a SLC load can be handled similarly. We can transition to the M_C_S or M_C_E state. owever, as an optimization we bypass the M_C_S state entirely and transition directly to the M_C_P state. The S state does not provide any benefit here; any SLC operation making use of such a system coherent state would have to write-back the GLC-written data first. This would require requesting exclusive access, so the S state and its associated permission are redundant. By proactively moving to the P state, the directory can grant another cache exclusive access without having to invalidate the TCC. This avoids overhead. In contrast to the M_C_S state, the M_C_E state, benefits from the exclusive access of the CXL E state. This exclusivity allows the dirty GLC-scoped data to be instantly committed if an SLC option occurs. Such an operation then yields an I_C_M state. However when a GLC store happens after a SLC store special care needs to be taken. After a SLC store the data in the TCC is system-coherent and dirty. Directly committing GLC dirty data on top of this would lead to inconsistencies: if we receive a FwdGetM request from the CXL domain, the SLC data is expected to be written back, while the GLC data should still remain non-coherent. To avoid such inconsistencies, we only write-back the dirty SLC data to memory, satisfying the system level coherence. We do so before writing the GLC dirty data to the TCC and moving into the M_C_P state. This avoids the M_C_M state.

6.2.2 SLC scoped operations after GLC scoped operations

Action \ State	I_C_M	I_C_E	I_C_S	I_C_I
SLC load	*hit!*	*hit!*	*hit!*	GetS →I_C_E/I_C_S
SLC store	*hit!*	→I_C_M	GetM →I_C_M	GetM →I_C_M
GLC load	→V_C_M	→V_C_E	→V_C_S	GetP →V_C_P
GLC store	PutM+WB →M_C_P	→M_C_E	PutS →M_C_P	GetP →M_C_P

Table 6.2: State mapping table, while the VIPER–CXL bridge is in a SLC state

In the original VIPER protocol, an SLC operation will always invalidate, any value written or cached in the GLC scope, and write-back, if applicable. Therefore, GLC operations will always encounter an I state after an SLC operation. To mirror this, we implement an SLC load by transitioning from V_C_P and M_C_P to the appropriate coherent states I_C_S/I_C_E or I_C_M, when an SLC load is issued. Thus, in the CXL domain, a GetS or GetM is issued. The data already present is replaced by freshly acquired clean data in the case of a V_C_P. In contrast, it is kept as new system-wide dirty data in the case of M_C_P. Replacing the data is necessary as it may not be up to date. This can occur because the CXL-domain does not inform the bridge of changes the CPU might make. Because the bridge is, from its view, invalid. In contrast to this we keep the dirty data as a coherent CXL M state, as this equates to writing back in the original VIPER protocol. When loading with SLC scope in the V_C_S, V_C_E, or V_C_M states, data replacement can be omitted. The data is loaded directly, since it is already coherent and up to date.

Similarly, an SLC store at the V_C_E or V_C_M states does not need to acquire data or exclusive permission and can move directly to I_C_M[2]. The same applies to the M_C_E state, as moving to I_C_M equates to a write-back of the GLC written data. If the bridge is in the V_C_S or M_C_P state before storing in the SLC scope, exclusive access has to be acquired via GetM in the CXL scope. In both cases however, the already present data is reused, again equating to a write-back for M_C_P. In the V_C_P the data must be replaced by data acquired using the GetM request.

[2]The CXL E state can silently upgrade to M.

6.3 Example: Operation of VIPER–CXL Bridge

To further the understanding of our bridge, we will introduce an example.

Initially, the bridge is in the I_C_I state. The first operation initiated by a CU occurs, and results in an SLC load. As coherent data is required, the bridge sends a GetS in the CXL domain. Once the data has been received by the bridge, it will move into the I_C_S state and relay the data to the requester. The next operation reaching the TCC is a GLC load by another CU: as the bridge already has the current data, there is no need to load form memory. It can therefore directly move to the V_C_S state and send the data to the corresponding TCP. Now, the same CU writes to the cache-line. The data is already present in the bridge. However, as per the previously introduced optimization, it does not transition to the M_C_S state, rather than to the M_C_P state. For this, a PutS is issued in the CXL domain. To complete this operation, a acknowledgment of the store is sent to the CU. After this the first CU again tries to load in the SLC domain. Since the bridge is in the M_C_P state, a GetM is issued. After gaining exclusive access, the GLC modified data is sent to the CU. Subsequently the bridge moves to I_C_M, but the TCC cache line does not change. Finally, an SLC store occurs. As the bridge is already in the I_C_M state no further action is required besides the storing of data.

6.4 Invalidations

Invalidations can be initiated from both the VIPER and the CXL domain. When coming from the VIPER domain, they are sent by the TCP via the InvCache message. However, in the original VIPER implementation, this is only defined for the V and I states – notably, *not* the M state. As such, it is trivially adapted for our bridge: if a InvCache message arrives at the bridge in a V_C_M, V_C_E, or V_C_S state, it will move to an I_C_M, I_C_E, or I_C_S state respectively, and send an acknowledgment. Furthermore, if an InvCache message arrives at the V_C_P state, we invalidate it completely and transition to I_C_I. If the bridge is in any other state, it simply sends an acknowledgment.

When coming from the CXL domain, two requests may lead to cache invalidation: the BISnpInv message, a simple request to invalidate caches, which is equivalent to requesting exclusive access (GetM), and the BISnpData message, which is a request for shared access (GetS) [6]. The BISnpInv message, unlike the InvCache message, always invalidates the CXL side of the bridge. Thus, after handling a BISnpInv message, the bridge must be in the I_C_I, V_C_P or M_C_P state. To achieve this when reviving the invalidation on V_C_E or V_C_S, the bridge transitions directly to V_C_P. Similarly, the I_C_E, I_C_S, and M_C_E states move to the I_C_I and M_C_P states respectively. In contrast, to this the I_C_M and V_C_M states have to write-back their dirty data before

transitioning.

While handling the BISnpData message, the sharing permission can be retained: the I_C_M and I_C_E states transition to the I_C_S state. Similarly, the V_C_M and V_C_E will relinquish their exclusive access and transition to the V_C_S state. To accomplish this, if the CXL side of the bridge is in the M state, dirty data must be written-back. Although it is also possible to move to the M_C_S state from the M_C_E state, we transition to the M_C_P state, as to adhere to the optimization mentioned above.

6.4.1 Conflicts

Receiving a CXL invalidation while in a transient state (i.e., waiting for data from the CXL domain) results in a conflict. We handle this according to tje CXL specification [6]: We immediately send a BIConflict message. Upon receiving the subsequent BIConflictAck there are two options: either the BIConflictAck is received before the completion acknowledgment of the original request (early conflict) or after it (late conflict). In the case of an early conflict, the invalidation must occur before receiving the data. Thus, the cache must act upon the conflict as if in the state before entering the transient state. If the conflict is a late conflict, the cache must handle the invalidation in the now new state which occurred after exiting the transient state.

6.4.2 Replacements

Replacements occur when a new cache-line needs to be allocated, but none are available. In this case, a victim line is chosen and fully invalidated, moving into the I_C_I state. When performing a replacement, M states both in the CXL and VIPER domains must be written back using a PutM. Furthermore, since the replacement operation is cache-initiated, the CXL directory must also be informed of that the clean states E and S are relinquished. This is done by sending a clean evict message.

6.5 Atomic Operations

Since atomics are part of the VIPER Protocol, we also have to bridge them to CXL. In VIPER GLC atomics are performed in the TCC. This is trivially translated in our bridge to perform the atomics in the M_C_P, M_C_E or M_C_S state. SLC atomics, in the default VIPER implementation are performed in the directory. Since our bridge is coherent and can acquire exclusive access, we perform atomics in the TCC in the I_C_M state.

7 Evaluation

7.1 Evaluation Setup & Methodology

For our evaluation, we use *gem5* version 25.0 and analyze the statistics it provides. The statistics are appropriately aggregated between GPU kernels. To configure the simulation we use the `apu_se.py` configuration, used by the default VIPER protocol. For this, we extend its already existing logic. We mirror the configuration of the of the original GPU. Adding only the appropriate CXL message buffers. However, we do change the CPU configuration to allow for our generated MESIF protocol. The original configuration models pairs of cores with two private L1 caches and one shared L2 cache. Additionally, the default VIPER protocol also models an L3 cache. In contrast to the original, we do not group cores into pairs. We, too, use a private L1 cache per core, but we use a single shared L2 cache. This configuration is provided by the ProtoSlicc compiler. Internally, the caches of both CPU and GPU are connected via `IntLinks`. They are also connected to the CXL-directory through `ExtLinks`. We make the latency property of `ExtLinks` available to the user as a command-line argument, thus allowing for easy modeling of different latencies on the CXL bus. Note that, while we model the latency of the buses via the network, the original VIPER implementation does so at the cache processing level. It only applies a latency of 60ns (120 cycles) when a TCC request is issued to the directory. In contrast, the CPU and the directory to GPU direction of the original implementation does not have such a delay[1]. Our approach models these delays.

When running benchmarks, we use the default parameters provided. Where not indicated otherwise stated, we use four CPU cores and four GPU CUs. Furthermore, we use a CXL latency of 60ns, as to be comparable to the original VIPER implementation. All other configuration inputs are kept at their defaults. Specifically, this means we use a cacheline size of 64Bytes, a L1 size of 64KiB, and a L2 size of 2MiB.

[1] A delay exists, though it is very small, i.e., 5 cycles/2.5ns.

7.2 Benchmark-suite: Rodinia

We employ the Rodinia benchmark suite to evaluate our proposed APU-CXL model. Originally introduced by Che et al. in [26], Rodinia is designed to evaluate the performance of heterogeneous systems. The suite spans a multitude of applications from different domains, including physics simulation, data mining, and image processing. Furthermore, Rodinia provides OpenCL, OpenMP, and CUDA implementations, enabling execution on both the CPU and GPU [26].

For our purposes, we will only evaluate using the Rodinia GPU implementation. However, since gem5's APU model uses the ROCm stack, it is unable to run CUDA applications. Thus, we convert the CUDA implementation of Rodinia to the ROCm equivalent, HIP. As HIP syntax is based on CUDA, this can be done via a simple text replacement on Rodinias source code using AMD's `hipify-perl` tool.

However, it is important to note that the HIP compiler, `hipcc`, does only seldom, if ever, generate memory instructions with the SLC bit enabled. Therefore, we investigate the performance of the SLC part of our model separately in Section 7.7.

7.3 Overall Performance

Overall, a geometric mean speedup of 0.99, with a standard deviation of 0.10, relative to the default VIPER implementation, can be observed[2]. The individual speedups can be seen in Figure 7.1. It can be concluded that our model is comparable in performance to the original implementation if the same latency applies, though there is a minor slowdown. This means our approach can effectively minimize the latency between the CPU and our CXL directory.

While most of the benchmarks have a slightly longer simulated execution times than the original implementation, the `hotspot` and `myocyte` are faster in our model. Notably, the speedup of the `myocyte` benchmark is more than $1.4\times$.

In the following, we will investigate how these differences in performance arise.

7.4 GPU – TCC Hit-Rate

When comparing the hit-rate of our adapted TCC to the original, we can observe a superior performance of our model. This can be seen in Figure 7.2. Note, we did not include the `bfs` and `nn` benchmarks in Figure 7.2, as these produce a hit-rate of 0 in both our model and in the original. The `bfs` benchmark does not issue load or

[2]When excluding the outlier, `myocyte`, the geometric mean is ~ 0.97 which is also the median.

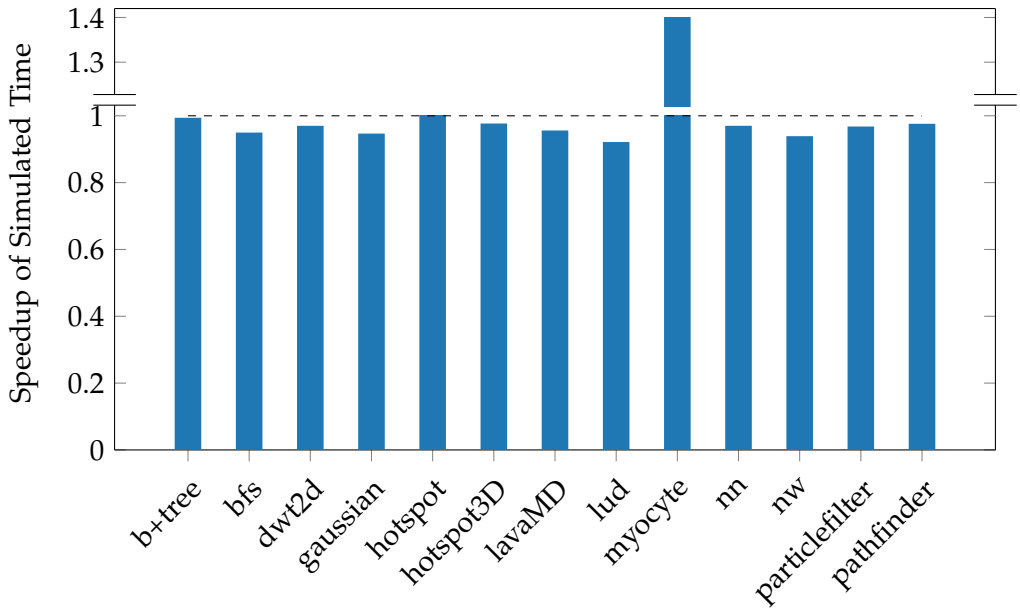


Figure 7.1: Speedup of simulated time of our model, relative to the default VIPER-MOESI implementation.

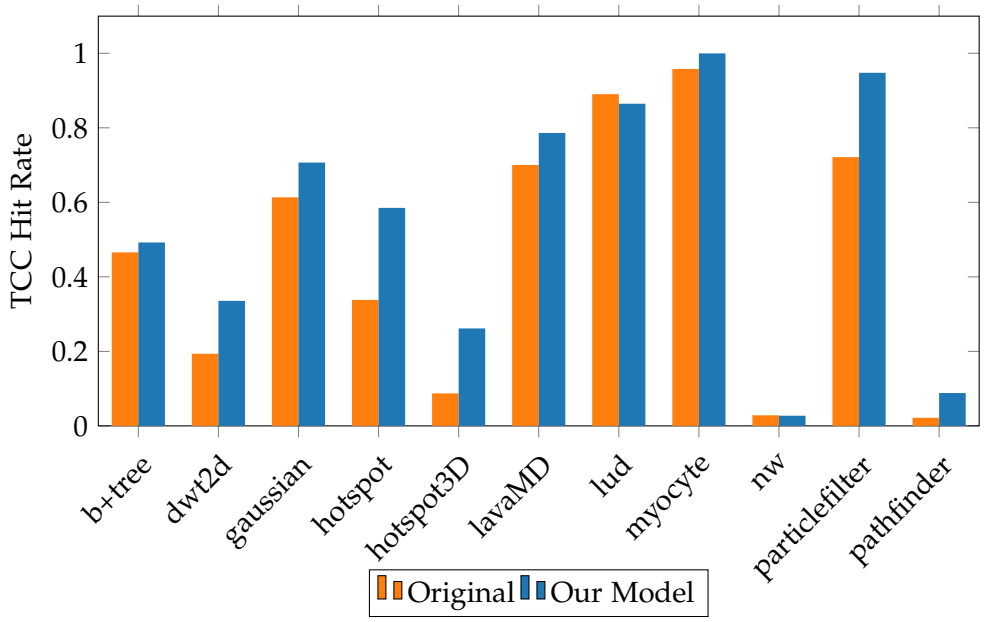


Figure 7.2: Hit-rates in the TCC of the original and our model

write operations on the GPU[3] and the `nn` benchmark only issuing loads for untouched memory locations (i.e., only generates misses).

Our model outperforms the original VIPER implementation in eight out of eleven evaluated cases. Notably, we achieved a near-perfect 0.9997 hit ratio in the `myocyte` benchmark. This contributes to the significant speedup observed in Figure 7.1. Furthermore, our model increases the hit-rate of `pathfinder` more than threefold (305%). Significant gains were also observed when running the `hotspot` program. It received the highest absolute hit-rate increase of 0.25. Conversely, the original model performs slightly better than ours in both the `nw` and `lud` benchmarks. However, the absolute difference is negligible for `nw` (~ 0.0012) and minute for `lud` (~ 0.028).

The average increase in hit-rate of 0.10, can be explained by the write-through nature of the default VIPER implementation. Workloads that exhibit a read-after-write access pattern, such as the read-heavy `particlefilter`, benefit the most from this. Because, unlike the original VIPER, our bridge does not invalidate written-to data.

It is noteworthy that, when enabling the write-back option of the original VIPER implementation, the margins between hit ratios shrink. Nevertheless, our model still comes out on top: for the `particlefilter` application, the hit-rate difference drops from 0.23 to 0.02. However, we also observe an increase: the hit-rate difference rises from -0.03 to 0.17 when running the `lud` benchmark.

This unusual hit-rate difference can be attributed to a significantly higher replacement rate in our implementation. The replacement rate difference can be attributed to the original VIPER implementation not setting the Most Recently Used (MRU) cache on read/store misses which synergies with the access pattern of the `lud` benchmark[4].

7.5 CPU – Instructions Per Cycle

To understand the impact of our CPU coherence protocol choice, consider the Instructions Per Cycle (IPC) of the simulated core 0, as shown in Figure 7.3. We only analyze core 0 because the Rodinia benchmarks rarely utilize the other cores.

As can be expected, when comparing remote memory with local memory, our model has an overall higher IPC. This is due to the increased number and longer duration of memory stalls introduced by remote memory. In nine of the thirteen evaluated benchmarks, we observed a decrease in IPC. This results in an average absolute IPC change of -0.001. The greatest relative drop was observed in the `gausian` and

[3] The `bfs` benchmark loads data into the GPU via DMA. And only uses the non-coherent scratch memory.

[4] The `lud` benchmark often stores values to new matrices in memory with a single access per cell. By not marking the cache-line containing the cell as MRU, other used cache-lines are retained longer. It is unclear if this is a conscious optimization or due to an oversight.

`bfs` benchmarks, both of which decreased the IPC by 5%. However, for the `myocyte` and `hotspot` benchmarks, the IPC of our model exceeds that of the original VIPER implementation, by 22% and 2%, respectively.

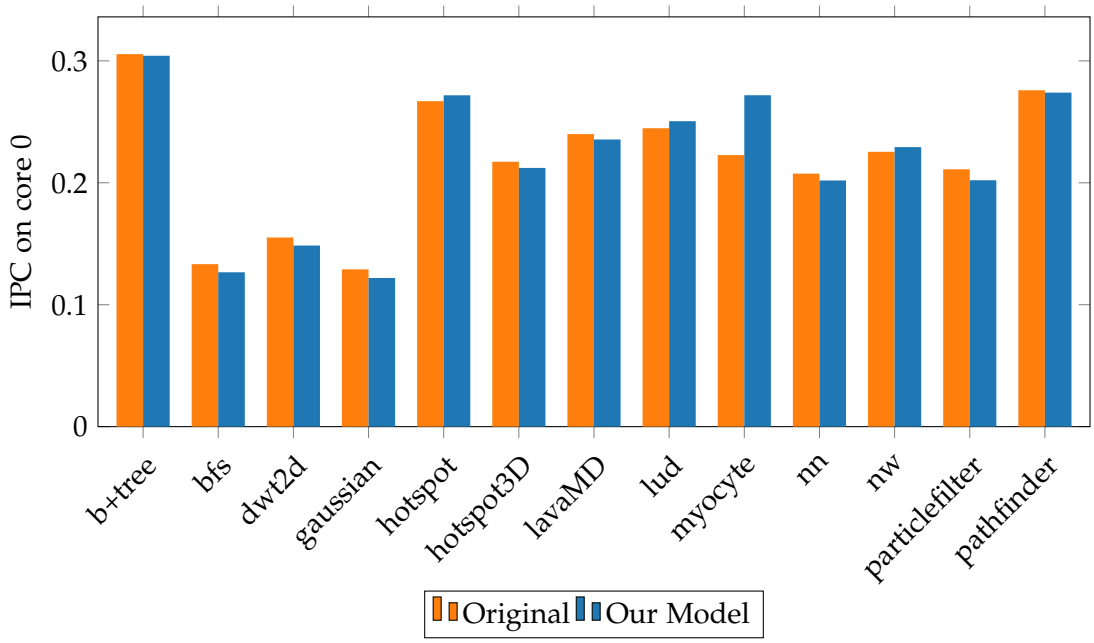


Figure 7.3: IPC on core0 comparison between our and the original model

7.6 Impact of Latency

To investigate the unavoidable impact of physical link latency with remote memory on heterogeneous computing, we simulated the benchmarks with increasing CXL latency. As can be seen in Figure 7.4, we used latencies ranging from 60ns to 100ns. We used a run with the latency of 60ns as our baseline and calculated the geometric mean across the runtime of all benchmarks. Figure 7.4 shows a strong linear relationship ($R^2 = 0.98$) between latency and runtime. This means the impact of latency on performance is easily predictable, thus enabling system designers to easily create systems that adhere to given specifications. However, as can be seen in Figure 7.4, the error of our measurement increases with higher latencies. This indicates the impact of latency varies between benchmarks. We see the biggest slow down with the `nw` benchmark, with an average -6.65% per 10ns of latency. This can easily be explained by the low TCC hit-rate of this benchmark. A low hit-rate causes more and, because of the increasing latency, longer

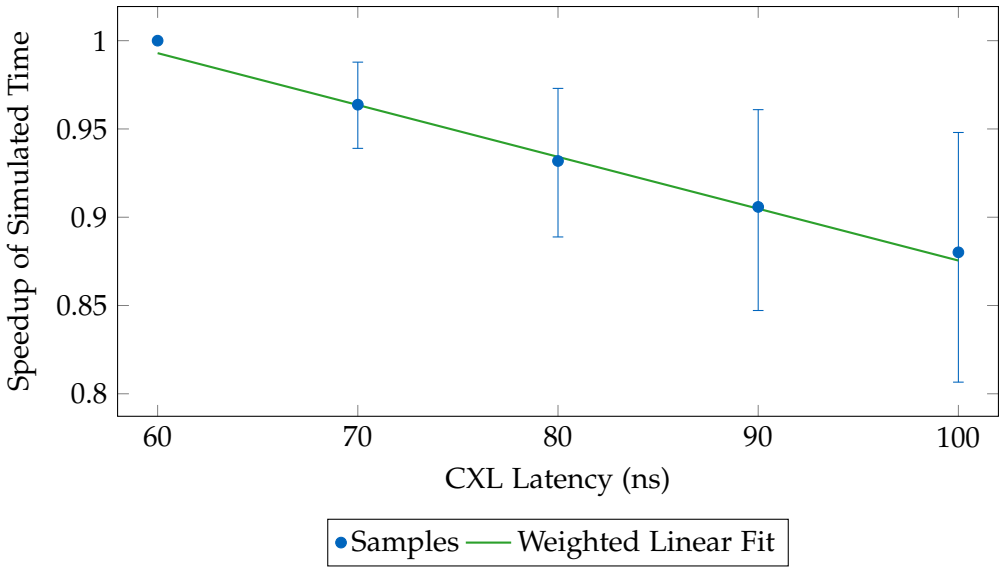


Figure 7.4: Geometric mean speedup of our model in relation to latency

fetches from memory. The `hotspot` benchmark exhibits the smallest penalty at -0.23% per 10ns, this can be attributed to the combination of high hit-rate in the TCC and the high CPU IPC, which indicates a high CPU cache hit-rate. Thus, the `b+tree` benchmark avoids costly fetches through its cache locality.

Further we investigate the impact of higher latencies on the `hotspot` benchmark. For this we run the benchmark at latencies typical for different CXL applications. We again use a baseline of 60ns and compare it to latencies typical for Non Uniform Memory Architecture (NUMA) of 150ns, local CXL systems (250ns, 450ns) and, even remote CXL (600ns). [27] The speedup of these latencies over the base line can be seen in Figure 7.5. We still observe a linear slowdown in realization to latency. This indicates the linear relationship holds even in high latency environments.

With this information, we can confidently conclude that the limiting factor for latency scaling is the hit-rate of the evaluated program, both in the GPU and CPU.

7.7 SLC Accesses

The `hipcc` compiler rarely generates SLC-scoped memory instructions, because it assumes them to be very slow. However, we sought to improve upon this performance by coherently caching them in the TCC. To investigate this performance, and due to the lack of SLC instructions emitted when compiling Rodinia, we devised a micro-

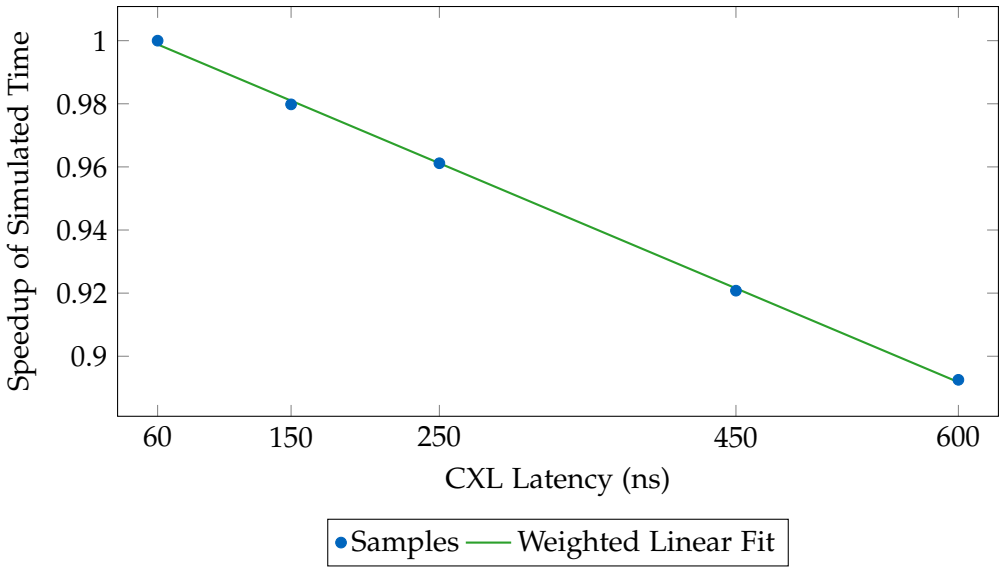


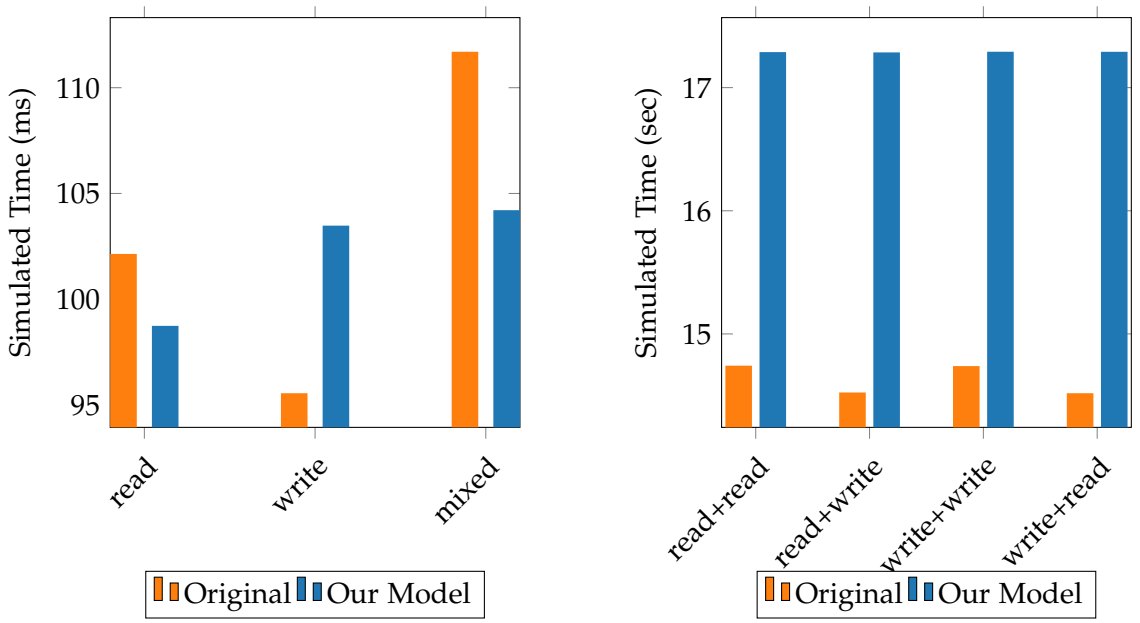
Figure 7.5: Speedup of the `hotspot` benchmark in relation to high latencies

benchmark. Our micro-benchmark consists of three kernels to evaluate the scenarios of a sequence of loading, a sequence of storing, and a mixed sequence. All of these follow the same scheme, which can be seen in Figure 7.6. It depicts the mixed sequence kernel. For the loading and storing kernels, lines 6 and 7 are omitted, respectively. We ran these kernels 50 times consecutively and with 32 blocks containing 32 threads each.

```
1 __global__ void read_write_bench(float4 *inout) {
2   int idx = blockIdx.x * blockDim.x + threadIdx.x;
3   float4 data;
4
5   for (unsigned i = 0; i < 100; i++) {
6     flat_load_dwordx4_slc (&inout[idx + i], data);
7     flat_store_dwordx4_slc(&inout[idx + i], data);
8   }
9 }
```

Figure 7.6: HIP kernel code illustrating mixed sequences of SLC loads and stores

In our model, we found that the first executed kernel had significant overhead. This is to be expected, as the original VIPER only writes-through or reads-through on SLC instructions. In contrast to this our approach acquires access via CXL, which may



(a) Runtime without CPU accesses

(b) Runtime with CPU accesses

Figure 7.7: Simulated runtime of benchmark kernels executed 50 times

involve invalidating sharers. However, we observed a significant performance increase with consecutive SLC read instructions on the same line, since our model caches this data in the TCC. The same applies also when performing store operations. However, the difference is much less. Thus, a greater number of stores is needed to overcome the initial overhead. This can be seen in Figure 7.7a: one can observe the overhead of our model increases from the read to the write test due to acquiring exclusive access. Furthermore, one can observe the overhead remains the same between the write and the mixed kernel. The original implementation exhibits better performance in the write benchmark, due to not acquiring exclusive access in the first execution of the kernel. Subsequent executions show an improvement of our model over the original. However, the difference is less than in the read benchmark, since for writes the original implementation does not have to wait for a response from memory.

Performance degrades massively when CPU and GPU accesses are run in parallel. This is due to an increase in coherence traffic. This was observed in both our model and the original. We evaluated this by running reads or writes in the same fashion as on the GPU, on the CPU. This can be seen in Figure 7.7b. In this test, the original VIPER implementation outperforms our model by a large margin in these scenarios. However, this is to be expected. The higher latency between the CPU and directory in our model

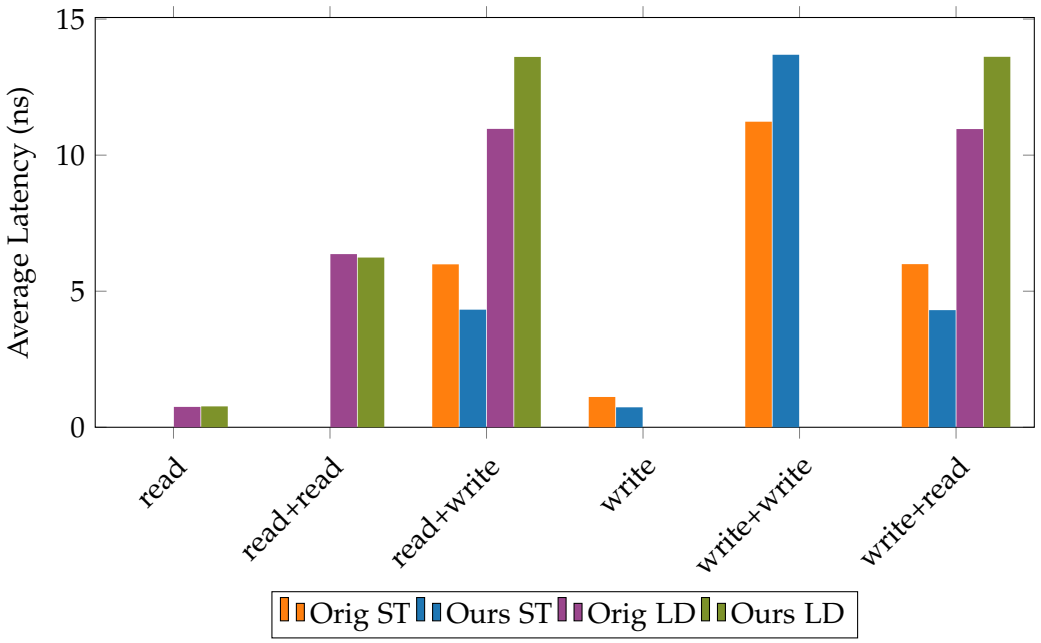


Figure 7.8: Average latency of stores (ST), loads (LD)

is exacerbated by the large amount of coherence traffic. We observe the benefits of the store-through and write-through: due to not caching SLC data in the TCC, the CPU does not have to invalidate or acquire access from it. This contrasts with our bridge which caches any access coherently in the TCC, and thus must be invalidated by the CPU. Both the original VIPER implementation and our approach perform better when the CPU is writing.

Furthermore, we examined and compared the average latency of loads (LD) and stores (ST) between the original model and ours. Note, that latency is aggregated between the GPU and CPU. The results can be found in Figure 7.8. We found latency to be similar if not better in the GPU only cases. We see a reduction in latency for stores in our model. The exception to this is the `write+write` benchmark, in which the coherence traffic added by out SLC-caching CXL implementation shows higher latencies. In combined read and write benchmarks, we see increases in latency of loads in our model which is caused by the added invalidation.

Overall, our approach provides sufficient, if not better, performance when using SLC instructions. Though, significant regressions can be observed when frequently accessing used memory with the CPU. Since CPU accesses are the primary function of SLC instructions, this behavior is unsatisfactory and should be improved in the future. One can imagine an approach using non-caching SLC stores, with the knowledge of

whether data will be accessed by the CPU.

8 Summary and Conclusion

As modern applications are becoming increasingly data-intensive and heterogeneous, the need for scalable memory, both on the CPU and GPU, has arisen. But current solutions to this problem either increase the complexity for the programmer or introduce overhead. They require manual intervention for memory management, or use automated systems to transfer data to and from the GPU.

We overcome the limitations of previous approaches by using CXL memory in combination with an APU architecture. In our model, both the CPU and the GPU are treated as CXL hosts, while the shared memory pool is the CXL device. We utilize coherence bridges to translate the CXL.mem to CPU's and GPU's cache coherence protocols. We chose the MESIF (Modified, Exclusive, Shared, Invalid, Forward) protocol for our CPU coherence protocol. We do this as an effort to minimize the impact of the added latency of CXL memory by means of the forwarding property. However, we do not implement the standard MESIF protocol; we opt for a directory based implementation, not only to increase the scalability but also to ease the design of the coherence bridge. For this we use the *ProtoSlicc* compiler and its DSL to generate the implementation not only of our caches but also of the coherence bridge between MESIF and CXL. Furthermore, *ProtoSlicc* allows us to only specify the stable state protocol and generate transient states.

Since we want to simulate our model using the gem5 simulator and its APU model, we choose its VIPER protocol as the GPU cache coherence protocol. As it is already implemented in the gem5 simulator, we only design a bridge to CXL. However, our design optimizes upon the original TCC by caching even system level dirty data. Thus, minimizing the the traffic over the high latency CXL-bus. To implement this, we modify the existing VIPER TCC to communicate with our CXL directory. Notably, we introduce a new P state to indicate valid data, while renaming incoherent in the CXL domain. Such a state can be produced by GLC operations.

We evaluated our approach by simulating the Rohdinia benchmark suite and comparing the performance of our model to the default *gem5* APU model. We found our approach to almost match the performance of the original APU model when matching the latency of the original. Furthermore, we observed an increased hit-rate in the TCC, which can largely, though not entirely, be attributed to the write-through nature of the VIPER protocol. To understand the impact of MESIF and CXL have on the CPU,

we examined its IPC count and found it to slightly decrease when compared to the MOESI-based original model. We attribute this to the higher physical link latency, as well asincreased communication overhead caused by the CXL bus. When we investigated the impact of rising latencies on the runtime of applications in our model, we found a linear correlation. This allows for easy modeling. However, applications with low hit-rates in both the CPU and GPU caches were more impacted by increased latency. We found SLC instructions to be seldom used but performing adequately. With the exception of accessing the same memory from the CPU and GPU simultaneously. This showed regressions compared to the original model. This unsatisfactory behavior should be improved upon in the future. Such an improvement may be achieved by avoiding the invalidation, for example by not caching in the GPU, when CPU accesses are imminent.

In conclusion, one can say our model, which combines CPU and GPU memory via CXL, achieves seamless memory scaling for both. It maintains nearly equivalent performance to traditional APUs. Furthermore, with the UM of our system, the burden on the programmer is decreased: no explicit memory movements between the CPU and GPU are necessary. All of this is achieved using the open CXL standard, thus enabling implementations across vendors. Overall, our model of a CXL-APU addresses the shortcomings of current heterogeneous systems.

9 Related Work

9.1 Unified Memory in APUs

APUs with unified memory, like we extend with CXL, are already established in heterogeneous systems. In [28] Whalgren et al. investigate the performance of one such APU, the AMD MI300A APU, paying special regard to its unified memory. They determine a plethora of memory properties, such as latency and cache coherence overhead. They also examine memory savings and propose a strategy for porting classical applications to APUs. Whalgren et al. find CPU memory latency to be lower at all cache levels. This is exacerbated by the missing GPU L3. The authors of [28] find significant coherence overhead when running CPU and GPU threads in parallel. However, they find the CPU is more impacted by this. These findings confirm the results observed in Section 7.7. Using their proposed porting strategies, the authors find memory usage to decrease significantly. They achieve this by eliminating duplicate buffers.

9.2 CXL in Heterogeneous Systems

Similar to our proposal, Lee et al. propose a synergy between NVIDIA's unified memory and CXL in [29]. They introduce two scenarios: in scenario one, CXL is used to supplement GPU memory, in the manner of a memory expander. Data is then moved to the CXL remote memory when the GPU tries to access data. Scenario two sees the CXL memory as main memory of the CPU, such that data is moved from CXL memory to GPU local memory when the GPU tries to access it. The authors, however, only simulate a decline in GPU performance with increasing page fault latency. Unlike this thesis, where we simulate an entire CXL system. They find CXL to be promising, especially given the simplified programming model NVIDIA's unified memory provides [29].

Another paper investigating CXL in a similarly heterogeneous scenario is [30]. To evaluate the performance of CXL in a tightly integrated, heterogeneous system, the authors of [30] create a benchmark. This benchmark uses a Field Programmable Gate Array (FPGA) to decompress a file containing values, which a GPU uses to calculate

π. In a traditional, less tightly coupled, system, this involves an initial transfer from the CPU to the FPGA. Followed by data movement from the FPGA to the CPU, which in turn, transfers its data to the GPU. A final data movement occurs from the GPU, after it finishes calculations, to the CPU. This is greatly improved when using CXL: the intermediate transfers from the FPGA to the CPU and from the CPU to the GPU can be eliminated. The authors also propose a balanced pipeline approach where send/receive can overlap with the kernel. They evaluate their approach using a model based on high-level performance expectations of CXL and compare it to a the PCIe DMA model. When examining the isolated transfer times, the authors find CXL to be faster than PCIe for smaller transfer sizes bandwidth (Though CXL outperformend PCIe into the `KiB` range). They find significant speedups($\geq$ 14) when using CXL's cache-line size of 64 bytes. When running the entire model, the speedup of CXL over DMA was found to be 1.31$\times$ and 1.45$\times$ with the pipeline [30].

9.3 CXL GPUs

Although CXL-GPUs are not yet available, they are integral to realizing a CXL-APU. In [8] Gouk et al. propose the first step toward real CXL-enabled GPUs hardware: a silicon-based CXL controller, which enables them to expand GPU memory with CXL using DRAM or SSDs in CXL-GPU [8]. To optimize the CXL round-trip latency, two solutions are introduced: speculative reads and deterministic stores. The speculative read prefetches data into a ring buffer on GPU local memory. While the deterministic store uses a similar process for writes, coalescing them in memory if the device is busy. To evaluate their approaches, they simulate and compare them with a GPU with NVIDIA's UVM, a GPU with GPUDirect storage, and an ideal GPU with infinite DRAM. They use multiple workloads, as well as different backing media.

When DRAM is used as backing media, they found their simple approach to deliver performance close to the ideal GPU. This equates to a 50$\times$ performance improvement over UVM. When using SDDs, the speculative approach shows significant performance gains over the simple approach, which already outperforms the GPUDirect. Using deterministic stores further improves performance [8].

10 Future Work

This thesis has successfully detailed and evaluated a clear, and novel model for heterogeneous computing. While we have laid the foundation, future work still has multiple avenues to explore.

10.1 Specific Workloads

We have demonstrated the performance of our model using the Rodinia benchmark, however performance of real-world application is often more nuanced than benchmarks can capture. We would like to measure the performance of real-world applications, such as AI models and high-performance computing (HPC) applications. However, the gem5 simulation we utilize is limited in this regard: it relies on an outdated version of ROCm, and is also slow due to the precises nature of gem5.

To address the software stack of our CXL-APU model, we plan to port our gem5 SE simulation to the gem5 FS simulation environment. This will allow us to use simulated drivers[17], thus newer ROCm versions. Furthermore, we plan to employ trace-based simulation to increase simulation speed.

10.2 Interoperability with Local Memory

Another option that we have not explored is the utilizing of on-board memory in addition to our proposed remote memory. GPUs would benefit greatly from the bandwidth of local memory, because GPU throughput is determined by bandwidth[31]. There are multiple ways to establishing such interoperability: one might use separate regions for shared and local memory. However, this would also reintroduce some of the burdens of manual memory management: either the programmer or software would have to decide if and when to transfer data to and from the shared memory. Another valid approach would be using the fast GPU memory as an extended cache with a replacement policy like Least Frequently Used.

10.3 Multi-Device Setup

In theory, our model can support an arbitrary number of devices because they are totally independent. Each device would be connected to one shared memory pool. This includes CPUs, GPUs, and other accelerators. We plan to extend our simulation to also include multi-CPU setups. This would include both separate hosts and a single host, sharing cache lines via the MESIF protocol (i.e., multi-CPU clusters). Moreover we would also like to evaluate multi-GPU setups and compare these to the performance of GPU interconnects, such as *NVLink*. As gem5 has first class support for multi-CPU systems[14] and also provides support for multi-GPU simulation[20], these extensions of out model should prove simple.

Abbreviations

APU Accelerated Processing Unit

CU Comput Unit

CXL Compute Express Link

DMA Direct Memory Access

DSL Domain Specific Language

GCN Graphics Core Next

GLC Global Level Coherence

HDM Host managed Device Memory

IPC Instructions Per Cycle

ISA Instruction Set Architecture

PCIe Peripheral Component Interconnect Express

TCC Texture Cache per Channel

TCP Texture Cache per Pipe

SLC System Level Coherence

UM Unified Memory

UVM Unified Virtual Memory

List of Figures

List of Tables

Bibliography

[1] A. Tekin, A. Durak, C. Piechurski, D. Kaliszan, F. A. Sungur, F. Robertsén, and P. Gschwandtner, "State-of-the-art and trends for computing and interconnect network solutions for hpc and ai," Partnership for Advanced Computing in Europe, Tech. Rep., Jan. 2021. DOI: 10.5281/zenodo.5717283.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, 2020. arXiv: 2005.14165 [cs.CL].

[3] NVIDIA Corporation, *Nvidia h200 tensor core gpu datasheet*, https://resources.nvidia.com/en-us-data-center-overview-mc/en-us-data-center-overview/hpc-datasheet-sc23-h200, Accessed: 2025-08-22, 2024.

[4] Advanced Micro Devices, Inc., *Amd instinct mi300 apu datasheet*, https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/data-sheets/amd-instinct-mi300a-data-sheet.pdf, Accessed: 2025-08-23, 2025.

[5] N. corporation, *Nvlink & nvswitch: Fastest hpc data center platform*, https://www.nvidia.com/en-us/data-center/nvlink/, Accessed: 2025-09-21, 2025.

[6] Compute Express Link Consortium Inc., "Compute Express Link (CXL) Specification Revision 3.2 (Evaluation Copy)," Compute Express Link Consortium, Tech. Rep., Oct. 2024, © 2019–2024 Compute Express Link Consortium. Available: https://computeexpresslink.org/wp-content/uploads/2024/11/CXL-Specification_rev3p2_ver1p0_2024October2_evalcopy.pdf.

[7] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6. DOI: 10.1109/HPEC.2014.7040988.

[8] D. Gouk, S. Kang, S. Lee, J. Kim, K. Nam, E. Ryu, S. Lee, D. Kim, J. Jang, H. Bae, and M. Jung, "Cxl-gpu: Pushing gpu memory boundaries with the integration of cxl technologies," *IEEE Micro*, pp. 1–8, 2025. DOI: 10.1109/MM.2025.3582433.

[9] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence* (Synthesis Lectures on Computer Architecture 49), 2nd ed. Springer Nature Switzerland AG, 2022, ISBN: 978-3-031-01764-3. DOI: 10.1007/978-3-031-01764-3.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128119055.

[11] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 348–354, Jan. 1984, ISSN: 0163-5964. DOI: 10.1145/773453.808204.

[12] A. W. Hay, "Mesif cache coherence protocol," M.S. thesis, University of Auckland, 2012.

[13] 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 '. F. Zulian, "The gem5 simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020. arXiv: 2007.03152.

[14] 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. DOI: 10.1145/2024716.2024718.

[15] N. Carpentieri, A. Lefort, D. Schall, and P. Bhatotia, "ProtoSLICC: Automated Synthesis of Gem5-based Cache Coherence Controllers," in *ISCA 2025 gem5 Workshop*, 2025.

[16] M. D. Sinclair, "Running (AMD) GPU experiments in gem5," in *HPCA 2023 gem5 Workshop*, Available at https://www.gem5.org/assets/files/hpca2023-tutorial/gem5-tutorial-hpca23-gpu.pdf, 2023.

[17] M. Poremba, *Moving to full system simulation of gpu applications*, https://www.gem5.org/2023/02/13/moving-to-full-system-gpu.html, Accessed: 2025-09-20, 2023.

[18] T. Gutierrez, S. Puthoor, T. Ta, M. Sinclair, and B. Beckmann, "The AMD gem5 APU Simulator: Modeling GPUs using the Machine ISA," in *ISCA 2018 gem5 Workshop*, Available at https://old.gem5.org/wiki/images/1/19/AMD_gem5_APU_simulator_isca_2018_gem5_wiki.pdf, Jun. 2018.

[19] Advanced Micro Devices, Inc., "Reference Guide: Graphics Core Next Architectiure, Generation Next, Revision 1.1," Advanced Micro Devices, Tech. Rep., Aug. 2016, © 2016 Advanced Micro Devices, Inc. Available: https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/gcn3-instruction-set-architecture.pdf.

[20] B. W. Yogatama, M. D. Sinclair, and M. M. Swift, *Enabling multi-gpu support in gem5*, https://www.gem5.org/2020/05/30/enabling-multi-gpu.html, Accessed: 2025-09-15, 2020.

[21] V. Ramadas, D. Kouchekinia, N. Osuji, and M. D. Sinclair, "Closing the GAP: Improving the Accuracy of gem5's GPU Models," in *ISCA 2023 gem5 Workshop*, Available at https://www.gem5.org/assets/files/workshop-isca-2023/slides/closing-the-gap.pdf, 2023.

[22] N. Oswald, V. Nagarajan, and D. J. Sorin, "Protogen: Automatically generating directory cache coherence protocols from atomic specifications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 247–260. DOI: 10.1109/ISCA.2018.00030.

[23] D. Das Sharma, R. Blankenship, and D. Berger, "An introduction to the compute express link (cxl) interconnect," *ACM Comput. Surv.*, vol. 56, no. 11, Jul. 2024, ISSN: 0360-0300. DOI: 10.1145/3669900.

[24] C. Consortium, *Our members - compute express link*, https://computeexpresslink.org/our-members/, Accessed: 2025-09-21, 2025.

[25] C. Chen, X. Zhao, G. Cheng, Y. Xu, S. Deng, and J. Yin, *Next-gen computing systems with compute express link: A comprehensive survey*, 2025. arXiv: 2412.20249 [cs.DC].

[26] 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. DOI: 10.1109/IISWC.2009.5306797.

[27] J. Liu, H. Hadian, H. Xu, D. S. Berger, and H. Li, *Dissecting cxl memory performance at scale: Analysis, modeling, and optimization*, 2024. arXiv: 2409.14317 [cs.OS].

[28] J. Wahlgren, G. Schieffer, R. Shi, E. A. León, R. Pearce, M. Gokhale, and I. Peng, *Dissecting cpu-gpu unified physical memory on amd mi300a apus*, 2025. arXiv: 2508.12743 [cs.DC].

[29] J. Lee and J. Kim, "Synergizing cxl with unified memory for scalable gpu memory expansion," in *2024 International Conference on Electronics, Information, and Communication (ICEIC)*, 2024, pp. 1–4. DOI: 10.1109/ICEIC61013.2024.10457110.

[30] A. M. Cabrera, A. R. Young, and J. S. Vetter, "Design and analysis of cxl performance models for tightly-coupled heterogeneous computing," in *Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions*, ser. ExHET '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, ISBN: 9781450393447. DOI: 10.1145/3529336.3530817.

[31] M. A. Ibrahim, H. Liu, O. Kayiran, and A. Jog, "Analyzing and leveraging remote-core bandwidth for enhanced performance in gpus," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '19, Seattle, WA, USA: IEEE Press, 2024, pp. 257–270, ISBN: 9781728136134. DOI: 10.1109/PACT.2019.00028.