CPU-GPU Heterogeneous System Simulator for Coherence Performance

Xinchao Zha
University of Michigan - Ann Arbor
Ann Arbor, United States
xinchaoz@umich.edu

Jianping Shen
University of Michigan - Ann Arbor
Ann Arbor, United States
jpshen@umich.edu

Wynn Kaza
University of Michigan - Ann Arbor
Ann Arbor, United States
wynnkaza@umich.edu

Taeyoon Kim
University of Michigan - Ann Arbor
Ann Arbor, United States
taeyoon@umich.edu

Shengwei Wang
University of Michigan - Ann Arbor
Ann Arbor, United States
shengwsw@umich.edu@umich.edu

***Abstract*—Hardware has continued to become more specialized, with companies always searching for ways to optimize energy-efficiency, power, and speed. Many future systems are being developed to be heterogeneous, and finding innovative ways to integrate the CPU and Accelerator to increase the performance of the system is becoming more important. Traditionally, the CPU and Accelerator maintain their own memory, and use a high-speed interconnect to communicate between the two processors. However, due to programmer complexity, there has been significant industry push, originating from the HSA Foundation, looking into unified coherent memory which allows for more data movement and data reuse in heterogeneous systems. The goal of such system would be to reduce communication latency between different processors, and remove the need for the programmer to explicitly move data between different device memories.**

This paper focuses on developing a configurable simulation for heterogeneous cache coherency architecture. The heterogeneous system simulator was designed to provide the user with opportunities to experiment with different coherency models and configurations of different heterogeneous architectures, with goals of having the ability to compare the recreates the heterogeneous cache coherency architecture [1].

***Index Terms*—HSA (Heterogeneous System Architecture), GPU, CPU, Shared Memory Systems, Coherency, Caches**

I. Introduction

As software workloads continue to become demanding, hardware acceleration has become an important research subject to improve computing performance. To handle new workloads, GPU and accelerators have become increasingly popular in data centers, focused on massive parallelization. Currently, most systems employ a separated memory system where the CPU will offload it's tasks to the GPU and GPU memory. In CUDA, a parallel programming model created by NVIDIA, the user is responsible for handling the transfer of data from CPU to GPU memory, and vice versa. HSA [4] hopes to remove programmer complexity by defining a unified virtual address space for the CPU, GPU, and any other accelerators.

Challenges arise when two devices employing different memory coherence and consistency models need to access the same memory, as each device utilizes a distinct read/write granularity tailored to its specific memory requirements. Managing coherence becomes complex when modifications are made at different granularity's, such as word versus line granularity. Traditionally, various device types have implemented different coherence protocols based on their unique demands. In certain heterogeneous system architectures, devices are required to adhere to MESI-based coherence protocols. While these protocols are well-supported for formal verification, they are notably inefficient due to their extensive cache line management requirements. This paper contributes to the problem by implementing a hierarchical coherence inspired by Spandex [1] which allows individual devices to use their own coherence protocols where our LLC is responsible for translating the request from each devices. Additionally, there is a pressing need for new coherence models tailored to diverse device types. Proposals for hierarchical heterogeneous systems have been made, but most development occurs within industry where the development is simulated on closed-sourced simulators. This restricts public access and modification, making it difficult for the general public to test, validate and optimize their systems. To address these issues, we developed a Python-based simulator that is straightforward to implement and can evaluate the performance of workloads for heterogeneous coherence designs.

II. Background

Heterogeneous system coherency is difficult due to the diversity of implementation of coherency protocols from different accelerators and CPUs. In scenarios where a company did not develop the coherency protocols for one or more of the devices in a heterogeneous system, it is difficult to integrate the devices with functional correctness. Any coherence design has different trade offs in terms of network bandwidth, complexity, invalidation, and communication. In this paper, because Spandex acts as a LLC, we assume a directory based protocol. In the next section, we describe some of the trade offs for the provided CPU and GPU protocols.

A. CPU Coherence Protocol

The MSI protocol helps to maintain coherence in a multiprocessor system by correctly handling incoming requests to mange its three states (Shared, Modified, and Invalid). The MSI protocol is write-back, exploiting cache locality. In a directory based system, all requests must first be sent to a directory, where the processor can learn about the state of the block, and know whether they are able to obtain write/read access to the block. While the base protocol only handles three stable states, the actually implementation of MSI features a multitude of more transient states which are used to handle the transitions between the stable states while waiting for messages from other processors or the directory. These transient stables help to eliminate deadlocks and provide correctness.

Whenever a processor has a read request for a cache line, it will send a request to the directory requesting for shared access for that cache line. In the case where the cache line is in the shared or invalid state, the directory will add it to the list of sharers and provide the data to the requesting processor. However, if the cache line is currently in the modified state, then it will send a downgrade request from modified to sharer to the owner of the cache line, and have that processor send the data to both the requesting processor and the directory. In this case, the cache line is now in the shared state with two sharers (previous owner and requester), and all processors have a clean copy of the cache line.

A processor can get a write access for a cache line by gaining ownership for the block. It will send a request to the directory (GetM) requesting for modified access for that cache line. If the cache line is currently invalid in other processors, it can obtain ownership, and will get a writable copy of the data from the directory. If the cache line is currently in the shared state, the directory must first send invalidation messages to all processors. Then, the requester node must wait for the data and the number of acknowledgements it needs before it can move into the modified state. After a processor receives an invalidation request, it will send a invalidation acknowledgement to the new owner. In the case where a write request is sent for a cache line which is currently in the modified state, it will invalidate the previous owner, and upgrade the current requester to be the new owner of the cache line.

B. GPU Coherence Protocol

GPU implements a simpler coherency protocol compared to CPU as the GPU does not need to maintain a coherent view of the memory compared to CPU. Because the GPU execution involves a high number of parallel threads, maintaining a coherent view of the shared memory would require complex and frequent communications or memory barriers which can be very bandwidth-intensive. For this reason, GPU does not normally require highly complex coherence protocol like MESI-based and mainly focus on execution of code within their own memory space, implementing a simple VI-based protocol instead. In the VI protocol, there are two states: valid, and invalid. Similar to the MSI protocol, transient states are required to maintain coherency between GPU processors.

In the case of a load request, if the GPU is in a valid state, it will not cause a state transition. If the GPU is in the invalid state, it must request and obtain the data from the directory. In the case of a store request, because the GPU protocol is write-through, store requests are sent to the directory.

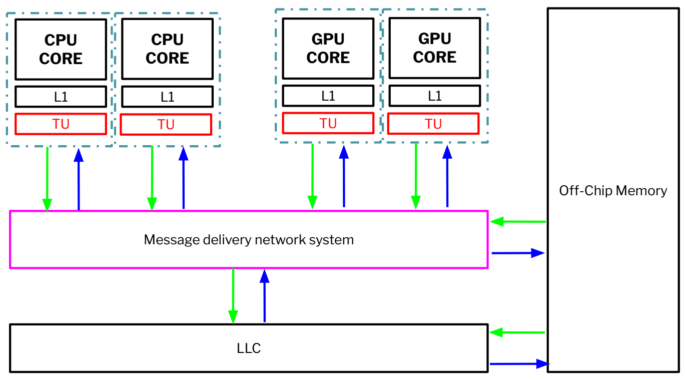

Fig. 1. Overview of System Architecture

III. Simulator

To easily validate and verify the heterogeneous protocol design, we designed a coherence simulator in Python. This simulator is highly customizable and configurable, allowing for the exploration of various protocol design trade-offs. Code for this project can be found at GitHub. To ensure the simulator's accuracy, we implemented our version of the heterogeneous protocol and conducted tests using the proposed simulator. An high-level architectural view of our system design can be found in **Fig. 1**, which demonstrates the simulators with two CPU cores, two GPU cores, and the LLC. Many open-source cycle-accurate simulators are available to assess the performance of various architectural designs. However, these simulators often require substantial modifications to integrate coherence protocols, which is the primary motivation for this paper. The key idea behind this work is the development of a message delivery network system that models the message passing in the coherence protocol. At each global tick, messages generated by each device are processed by the delivery network and forwarded to the appropriate destination node. If the destination node can process the message, the system then removes it from the device's message buffer. This streamlined approach provides protocol designers with a straightforward method to initially assess the performance of their designs, offering a first line of insight into their work.

A. Coherence Protocol Model

Different coherence protocols for each device are simulated by modeling state transitions within the cache controllers that are attached to the individual caches. The current simulator includes a GPU simulator which implements a VI protocol, a CPU simulator which implements a MSI protocol, and

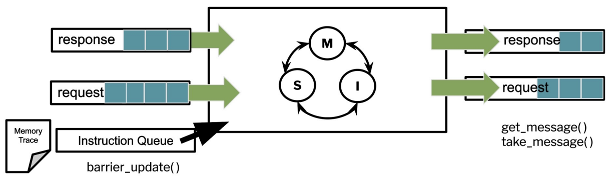

Fig. 2. Coherence Protocol Implemented on Processor Core

a shared LLC which implements the Spandex protocol. A diagram of our CPU simulator can be found at **Fig. 2** Additionally clock cycles are tracked through the progression of our workloads, however, due to lack of hardware of a system that integrates Spandex, we cannot compare the simulator clock accuracy against real hardware for the different workloads.

All caches are designed as a collection of cache entry class which includes important cache meta data like state, aligned address, and data. The cache class includes functions like allocation, de-allocation and setting LRU entry. It is also responsible for helping the cache controller decides whether the address is valid or needs to be replaced. The cache design here is independent of our controller design and can be used for all different types of coherence protocols and devices.

The simulation of memory operations is handled in parallel threads for each device, such as CPUs and GPUs. Special synchronization primitives, such as barriers, are employed to ensure that all the threads (either in the CPU or GPU, or across both) reach a certain point in the code before any of them proceeds. This prevents some threads in a CPU from reading or writing data that other threads are concurrently modifying from a GPU, eliminating potential data race. The GPU is assumed to operate under a data race-free consistency model, which hinges on the correct use of synchronization primitives by the programmer. When the memory trace implements barriers, it is expected that the barriers correctly control the execution order to maintain data consistency.

These memory instructions are first parsed through individual core and enqueued to the instruction queue at compile time. Additionally, the simulator implements message channels to handle communication messages. After the simulator starts running, at each global tick, messages on 3 message channels are handled based on their priority. Channel priority is given to the response channel, then request channel, and finally the instruction channel at each clock cycle. Priority ordering was required to remove deadlocks caused from a message waiting for a response but being blocked due to a different message. Only one outstanding message is allowed per channel, so if the channel has a message awaiting for a response or is empty, the handler function goes to the next channel. Using the peek function provided by the channel class, the first message of the queue from different channels are assessed and the system decides on which coherence event to take based on the message type or current cache state. These events then trigger the system to perform a state transition based on the state table defined. The state transitions will result in certain actions like allocation or deallocation of a cache entry or en-queuing a response or request message to a corresponding channel.

B. VI-based Protocol

The GPU simulator is capable of executing instructions such as loads, stores, and barriers, but also handling message requests from the LLC. On every clock cycle, the GPU will first check for a incoming instruction from the LLC. After handling a incoming instruction, the GPU will run it's own instructions which include loads, stores, and barriers from other processors.

For the GPU coherence system, we implement a VI-based coherence protocol, which can be found in **Fig. 3**. We decided on a simplified coherence model because of the nature of a GPU workload. Most GPU workloads focus on parallelism, and do not require complicated shared state between the difference processors.

The cache model used for the GPU handles configurable settings, with the ability to change the cache size, numbers of ways, line size, and the memory size. All of this information can be configured on cache initialization through a configuration file.

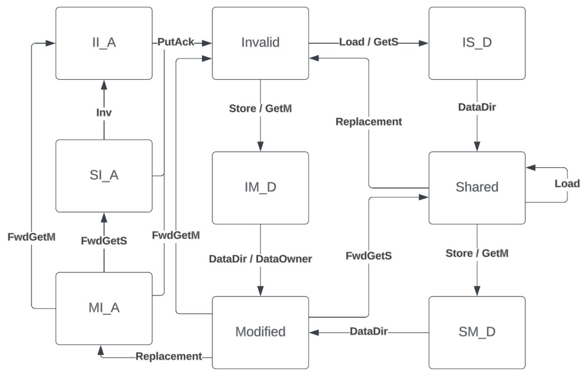

Fig. 3. Coherence Protocol Implemented on GPU

C. Modified MSI Protocol

The CPU simulator implements a modified MSI-protocol to better handle communication with the LLC. The protocol does not use an explicit PutS message for evictions in the shared state. Instead, when the CPU core have to replace a shared state entry, it does so without notifying the directory. Since the shared list in the directory can be different from the actual sharers, any processors that receive Invalidation message need to send a InvAck message to the directory. Additionally, unlike the GPU, the CPU also implements a translation unit which translates CPU requests to LLC coherency messages to handle communication between the two caches. These translation can be found in **Table I**. Multiple LLC Messages can be mapped to one entry on the CPU side.

For the CPU coherence system, we implement a modified version of the MSI protocol. We decided to modify the original protocol to reduce the complexity of the CPU, and to better

TABLE I
TRANSLATION BETWEEN CPU AND LLC INSTRUCTIONS

CPU ↔ LLC Translation	
LLC MSG	CPU Cache MSG
FwdReqS, FwdReqV	FwdGetS
FwdReqVE, FwdReq0data, FwdRvk0	FwdGetM
Data, DataOwner	RepRvk0
DataV	RepFwdV
DataOwnerV	RepFwdVE
GetS	ReqS
GetM	Req0data
PutM	ReqWB
RepS, Rep0data	DataDir
RepWB	PutAck

accurately handle the communication with the LLC based on our requirements. The implemented coherency protocol for the CPU is in **Fig. 4**. Similar to the cache model for the GPU, the CPU cache is configurable with the ability to modify any of the settings listed in the GPU section.

D. Last-Level Cache

The LLC in our system handles the heterogeneity of the system, and handles coherence between the CPU and GPU. As the LLC is similar to a directory cache, on top of having to implement the normal information for a cache, it must also have implementation details for the owner and sharers.

For the LLC coherence system, we implement a variation of the Spandex protocol which allows for flexibility and simplicity of implementation. The directory states are shown in our excel sheet.

There are 4 stables states in our LLC design. Invalid, Valid, Shared and Owned. Valid states in the LLC means that the data is up-to-date and no other devices have the data in either the Shared or Owned state. Shared stated is required to support any protocols with writer-invalidation and invalidation are required if other devices with write-invalidation protocol want to modify it. The Owned state in the LLC is equivalent to the Modified state where the target data is owned by the attached device and is relevant for any coherence protocol with ownership-based write.

LLC processes 5 different requests from the attached devices: ReqS, ReqV, ReqWT, ReqOdata, and ReqWB. It is responsible for sending forwarding requests to the other nodes : FwdReqS, FwdReqV, FwdReqVE, FwdReqOdata, FwdRvkO and Inv. Forwarding Revoke Ownership (FwdRvkO) messages occur when the LLC in the Owned state receives a ReqWT from the self-invalidation protocol. Invalidation occurs when the writer-invalidation plans to obtain ownership. In this case, the invalidation message is sent to all false sharer nodes kept by the LLC. Since the sharer list could be incorrect due to false-sharing, all nodes in the sharer list are responsible for sending back InvAck to the LLC to avoid any deadlocks. If the LLC is in the Owned state and receive a GetS, GetO or GetV request from the CPU or GPU, the data owner is responsible for responding with data to the requester. For GetS, GetV and GetO requests, the owner should not forward the request to the requester but instead to the directory.

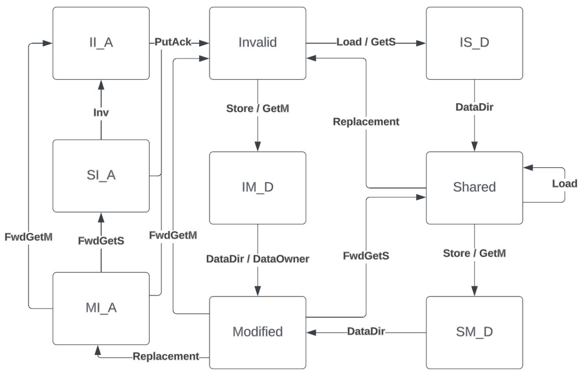

Fig. 4. MSI Coherence Protocol Implemented on CPU

IV. TRACE GENERATION

Due to the unavailability to find memory traces focused on heterogeneous systems, we implemented our own random memory trace generation that generates different workloads that would highlight protocol optimizations made in Section 3 for the heterogeneous system and based on the workloads listed in Spandex [1]. The trace generation handles the generation of memory traces for each of the devices (CPU, GPU), with three type of instructions: loads, stores, and barriers. For load and store instructions, the generator generates a random address and the type of instruction. For barrier instructions, the left number represents the barrier number, which is used to maintain ordering of barriers in the program. Additionally, the right number represents how many devices are required to wait for that barrier. Generated test cases can be found in `Simulator/Testcase`.

A. System Memory Space

To demonstrate the advantage of heterogeneous system, we selected a memory space that involves different segments of memory allocated for various CPUs and GPUs within a heterogeneous architecture. The first memory region is shared among all CPUs. Coherence between CPUs for this shared memory is maintained by a MSI writer invalidation protocol. This means that when a CPU writes data to a memory location, any cached copies of that data in other CPUs are marked as invalid, ensuring that stale data is not read. The next section is divided among multiple CPUs, with each having its private memory space. There is no contention between CPUs for accessing these spaces as they are isolated and private to individual CPUs. The last part of memory section is designated for GPU access only. The access to this region is controlled by synchronization barriers that we manually define and the coherence is maintained by the GPU self-invalidation protocol. This implies that only one CPU or GPU can access these memory blocks at a time and the coherence is maintained by the GPU invalidating its own cached data when required.

B. Workloads

In our test suite, we generate 6 different workloads to test the coherence and implementation of our simulator. Using the provided random trace generator, it would be possible to generate more test cases which covers the the memory space mentioned above, or modify the Python code to generate new types of workloads to evaluate the coherence protocol design with the simulator. The purpose behind these test cases is to first test whether the coherence models work correctly under different memory accesses and measure the performance of the protocol when both CPU and GPU access the shared memory space under the correct consistency. Test case 1 and 2 compare the distribution of work across GPU, and test case 3 and 4 test coherency in a system with multiple CPUs. Within these workloads, barriers were correctly generated to accurately handle the communication and consistency between the CPU and GPU.

V. Evaluations

A. Verification

The simulator provides the users with debug options that can be used to help check the correctness of the protocol. The debug class prints out all messages in the response and request queue as well as the message in transaction. Manual test cases were run with the designed heterogeneous protocol to test various corner cases to verify the correct functionality for the simulator. Up to 1000 random memory traces were generated and tested on the simulator to confirm functional verification coverage.

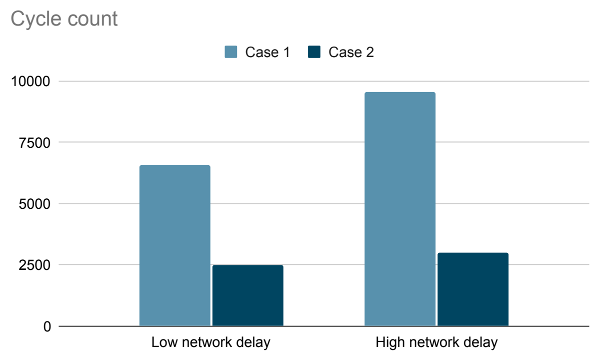

Fig. 5. Cycle count results from simulating network delay and GPU workload distribution

B. Results and Analysis

Performance of different coherence protocols were measured for heterogeneous system using cycle count and total message traffic during the runtime.

The simulator provide options to model different delays in the message passing network. **Fig. 5** shows the result of two different delays modeled in the system. Low network delay was set to be between 1-2 clock cycles while the high network delay was set to be between 5-10 clock cycles. The higher delay in the network causes much higher cycle count compared to the low network delay. **Fig. 5** also highlights how the simulator can correctly capture the workload distribution in the GPU. There are no contentions between the GPU cores and the simulator achieves up to 2.32x speed-up in the low delay network and 2.41x speed-up in the high delay network. The memory operations simulated up to 5000 instructions to account for normal GPU operations that typically involve much higher parallel execution and access the memory more frequently compared to the CPU.

Fig. 6 shows the cycle count of different workloads. Different line size was configured in the system to check the accuracy of the CPU coherence with the GPU coherence. As seen from **Fig. 6**, when the line size is increased, the number of cycles decrease. This behavior is expected in a correct system with multi-processors as CPUs benefit from higher line size update due to spatial locality. The private memory section of each CPU was reduced for test case 4, showing an increase in the cycle count.

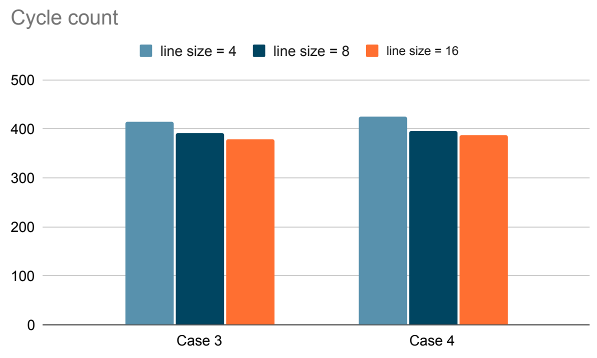

Fig. 6. Cycle count results from simulating different line sizes

The message traffic of different workloads in **Fig. 7** shows a similar result with cycle count. As the line size is increased, the CPU coherence takes advantage of the larger line, causing the message traffic to decrease. The advantage of using a MSI based coherence is more apparent from the total message count. In test case 4, we decrease the memory region of each CPU, which leads to an increase in the message traffic.

VI. Related Works

Both industry and academia have worked to look for profitable solutions for heterogenous architectures. Industry efforts like OpenCAPI, CXL, and NVLink all aim to achieve high bandwidth memory sharing and pooling through existing open standard bus. The use of an special FPGA (Field-Programmable Gate Array) controller manages data flow and help achieve high speed switching for different memory demands while ensuring coherency. Through our readings, we

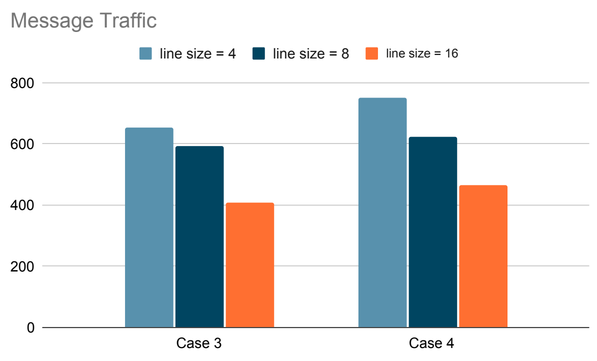

Fig. 7. Message traffic results from simulating different line sizes

noticed a lot of papers focused on a similar concept, remapping of the LLC to handle arbitrary requests from accelerators and CPU. However, different papers approach this problem in their own unique ways, focusing on different types of devices, security, and efficiency.

A. Crossing Guard: Mediating Host-Accelerator Coherence Interactions

Olsen *et al.* propose a coherence interface between the CPU and accelerator that standardizes the accelerator's coherence messages to provide bug-free cache coherency [2]. Coherency issues such as acknowledgements between devices, and handling race conditions are handled by crossing guard. However, Crossing Guard is mainly focused on potentially buggy, third-party accelerators, looking for ways to find ways to integrate it into the host CPU system.

B. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence

Alsop *et al.* put forth a novel alternative to MESI-based coherence for HSA [1]. With a focus on simplicity and flexibility, Spandex allows for CPU, GPU, and other accelerators to maintain their own coherency strategy for their workloads, but interface with each other through the Spandex protocol. The proposed protocol is able to reduce network traffic and execution time for CPU-GPU applications compared to a MESI coherence interface.

C. A Case for Fine-grain Coherence Specialization in Heterogeneous Systems

Aslop *et al.* build upon their previous work, Spandex, discussing methods to optimize memory requests to improve cache reuse in heterogeneous memory systems [3]. Compared to Spandex, this paper focuses on the advantages of a fine-grained coherence flexibility. This means that unlike Spandex, which allocates coherency flexibility at the device level, this paper allows for coherence flexibility to occur at the instruction level. To achieve this, the paper proposes multiple heuristic algorithms which find the best request to send to the LLC based on the static memory image of the code.

VII. Conclusion and Future Work

As hardware continues to evolve to handle new workloads and demand, it becomes increasingly important to find ways to accelerate memory coherency. We have created a simulator for heterogeneous coherence design that can be used to optimize different coherence protocols

To reduce the complexity of integrating different coherency protocols from different devices, we implement the Spandex protocol in our LLC. Our simulator is able to effectively handle running a modified version of the Spandex protocol in our LLC, while having our CPU and GPU run their own coherency protocols. Additionally, our simulator is able to generate 6 different types of random workloads with correct barrier integration which can be run on the simulator to observe the effectiveness of the heterogeneous architecture. Our work has demonstrated a new simulator for heterogeneous systems, which is able to effectively simulate different heterogeneous workloads on cycle granularity. Furthermore, due to the simulator's simplicity, it can be easily configured to test how different parameters affect the performance. We hope our elegant design can be a good solution to many who want to test our test heterogeneous protocol.

In our forthcoming research and exploration, we intent to broaden the accelerator to be able to run real-world workloads to effectively observe the beneficial gain provided over a homogeneous architecture. To test our simulator against hardware, we would be required to either obtain the memory traces from all devices and generate the barriers in the code in the correct location, or implement a execution based model, similar to AccelSim, which can handle running CUDA/C++ code out of the box within the simulator [5].

VIII. Teammate Contribution

Finally, you must include a brief statement of each team member's specific contribution to the project

Xinchao developed the simulator

Jianping worked on testcases and memory traces

Wynn worked on heuristic algorithm and memory

Taeyoon worked on MSI protocol simulator

Shengwei worked on memory traces

References

[1] J. Alsop, M. Sinclair and S. Adve, "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 2018, pp. 261-274, doi: 10.1109/ISCA.2018.00031.

[2] L. E. Olson, M. D. Hill, and D. A. Wood, "Crossing Guard," Computer architecture news, vol. 45, no. 1, pp. 163–176, Apr. 2017, doi: https://doi.org/10.1145/3093337.3037715.

[3] J. Alsop, Weon Taek Na, M. D. Sinclair, S. Grayson, and S. V. Adve, "A Case for Fine-grain Coherence Specialization in Heterogeneous Systems," ACM transactions on architecture and code optimization, vol. 19, no. 3, pp. 1–26, Aug. 2022, doi: https://doi.org/10.1145/3530819.

[4] "Heterogeneous System Architecture Foundation." https://hsafoundation.com/

[5] Mahmoud Khairy, Jason Shen, Tor M. Aamodt, and Timothy G. Rogers "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling", In The 47th International Symposium on Computer Architecture, May 2020