Exploring CPU-GPU Coherence

Gokul Subramanian, Urmish Thakker, Swapnil Haria, Rohit Shukla, Han Lin

May 14, 2015

Abstract

AMD, ARM and other members of the Heterogeneous Systems Architecture Foundation are focusing on integrated CPU-GPU systems with shared memory, to improve the programmability of heterogeneous systems. Such integration is also necessary to eliminate the energy and latency costs associated with conventional heterogeneous computation. This work investigates the relevance of CPU-GPU coherence for current heterogeneous workloads.

We propose certain modifications to Heterogeneous Systems Coherence (HSC), one of the few feasible coherence schemes aimed at such systems, to simplify the hardware overheads and improve performance. Furthermore, we evaluate upcoming workloads like Sirius Suite to understand the application structure needed to unlock the potential of CPU-GPU coherence. Our experiments show that while integrated CPU-GPU systems with coherent shared memory improves application performance, current coherence protocols are infeasible due to the high coherence bandwidth consumed by the GPU. Our modified HSC design uses region coherence to filter GPU-side coherence requests by up to 40%.

1 Introduction

General-purpose graphics processing units (GPGPUs) bear great promise for accelerating an increasing number of applications due to their throughput-oriented compute and memory systems. This trend has been recognized in the industry, with the result that commodity GPUs are becoming more closely coupled with the CPUs. GPUs are being physically integrated with CPUs on the same chip [6], as well as logically integrated through a unified shared address space [9]. The Heterogeneous Systems Architecture (HSA) by AMD [1] introduced this logical integration as a means to simplify CPU-GPU programming models (by preserving the pointer is a pointer abstraction) and improve performance by minimizing data transfer between the two devices.

Maintaining coherence in an unified address space can be done naively through explicitly disabling private caches (as in NVIDIA’s Kepler architecture), or in software through the use of appropriate barriers and cache flush operations, among other methods. However, these techniques result in performance degradation and burden the programmer. As a result, various schemes have been proposed in literature to support hardware coherence between the CPU and the GPU [13, 15, 5].

Most of our current work focuses on Heterogeneous Systems Coherence (HSC) [13], which leverages region coherence to limit the massive coherence bandwidth needed by the throughput-oriented GPUs. We propose several modifications to the original HSC design to lower the barrier to adoption. Our design utilizes region coherence only on the GPU-side memory network, while leaving the CPU-side intact. Furthermore, region prefetching is employed to exploit the streaming nature of GPU memory requests in a bid to improve GPU performance. This has been implemented in gem-gpu and evaluated comprehensively using heterogeneous system workloads like Rodinia [2] and Sirius Suite [4].

Additionally, we have also analyzed memory interactions between the CPU and the GPU on other fronts. The influence of coherent memory interconnect between the CPU and GPU on overall system performance has been evaluated to motivate a case for coherent CPU-GPU systems. Applications from the fields of machine learning and computer vision have been ported for execution in an integrated CPU-GPU environment, in order to identify new workloads which will benefit from coherent GPUs. Various CPU-GPU coherence schemes proposed in literature are compared in terms of relative performance and coherence activity.

The main contributions of this work are -

• Evaluating the limits of conventional coherence protocols in integrated CPU-GPU environments
• Comparing the performance and coherence activity under different CPU-GPU coherence schemes
• Simplifying the HSC design, proposing extensions to improve performance, and evaluating these in gem5-GPU

• Porting benchmarks in the Sirius Suite from CUDA to OpenCL and HSA implementations, and running these on real hardware

This report is structured in the following manner. Section II presents a brief background of CPU-GPU coherence. Section III motivates the need for CPU-GPU coherence, while Section IV is a brief discussion of our proposal for this project. Section V describes our implementation of Heterogenous Systems Coherence design. The methodology followed is detailed in Section VI, while Section VII summarizes the key results obtained by running heterogeneous workloads on gem5-GPU and real hardware. A brief overview of the related work is presented in Section VIII. Finally, Section IX concludes the report and presents directions for future work.

2 Background

2.1 Baseline Heterogeneous System

The baseline heterogeneous system would ideally consist of two clusters: a CPU cluster, which can contain any number of CPUs, and a GPU cluster made up of thousands of individual scalar units. In order to simplify the implementation, we choose to implement a single CPU core and a single GPU SM, which itself consists of multiple Compute Units. The SIMT nature of the GPU leads to tremendous memory parallelism, which puts strain on the memory system to deliver high bandwidth.

The CPU and the GPU both utilize non-inclusive and shared L2 caches. Both maintain coherence with each other and as well as with lower levels (directory/memory). The CPU L1 caches are implemented similar to the L2 cache while the GPU L1 caches are non-coherent in terms of hardware. They are kept coherent by writing through dirty data and flash invalidating the L1 caches when kernel begins.

The directory implements a hammer protocol - it does not contain any directory state and instead broadcasts requests to all the nodes in the system. In parallel, it fetches the data from the DRAM and forwards the response to the requestors. A directory hit guarantees that there is at least one cache in the system that has a valid copy of the requested block. Thus, for writes, those identified copies must be invalidated; for reads, one copy is probed directly for valid data. A directory miss requires the system to probe all caches.

2.2 Region Coherence

The key idea behind regional coherence is that allocating permissions to nodes based on region (which consists of multiple cache blocks) reduces the coherence traffic at the directory for permissions to each cache line in the block [10, 11]. The fundamental assumption in this is that data is shared at a coarse granularity and significant spatial locality exists in most applications, especially those running on GPGPUs.

A separate structure is implemented, which can sit prior to the directory or be distributed across the caches, which maintains information of permission for each region. When cache misses occur, this structure is first looked into, and if permissions exist for that region to the requesting node, then requests can be forwarded directly to the memory without increasing the traffic at the directory nor creating new traffic to check/invalidate other nodes.

When compared to normal directory coherence, directory-based region coherence will show a large reduction in directory bandwidth if the address stream exhibits spatial locality. Directory-based region coherence behaves similarly to region coherence after permissions are obtained for a region. On cache misses, the region buffer is queried; if the correct permissions are available for the request, it is forwarded directly to memory without any directory involvement.

3 Motivation

Currently, most code written for CPU-GPU systems follow a similar and rather limited programming model. There is preprocessing code which executes on the CPU, which handles the marshalling of complex pointer-based data structures like linked lists. The marshalled or serialized data is transferred to the GPU, which operates on this data. Lastly, the results of the GPU computation are transferred back to
the CPU memory, and unmarshalling is performed to recreate the relevant data structures. Although it is theoretically possible to overlap data movement between the CPU and GPU with computation, expressing dependencies is difficult in the current programming models. This leads to degradation of application performance as computation and data movement are serialized.

This is better expressed by means of a representative illustration. The code snippet presented below uses a ‘cudamemcpy’ command to transfer data between the CPU and the GPU. As seen in the code, pointers to data structures prior to the memcpy are different from those after the memcpy, thereby complicating the program structure. Also, software based coherence is unsuitable for switching within a heterogeneous system at fine granularities, as the overhead involved becomes very significant. This, for instance, rules out the possibility of having a fine-grained producer-consumer model between the CPU and the GPU.

```c
printf("Performing CPU computation\n");
bpnn_layerforward(net->input_units, net->hidden_units,net->input_weights, in, hid);

printf("Performing Mem Cpy\n");
cudamemcpy(input_cuda, net->input_units,(in+1) * sizeof(float), cudaMemcpyHostToDevice);
cudamemcpy(input_hidden_cuda, input_weights_one_dim, (in + 1) * (hid + 1) * sizeof(float), cudaMemcpyHostToDevice);

printf("Performing GPU computation\n");
bpnn_layerforward_CUDA<<<grid,threads>>>(input_cuda,output_hidden_cuda, input_hidden_cuda, hidden_partial_sum, in, hid);
```

There are certain inherent limitations in this model. There is significant amount of data transfer to and from the GPU. As Steve Keckler [8] reported in 2011, off-chip data movement consumes about 20-40x the energy spent in on-chip computations. Furthermore, this data transfer incurs a latency cost. As seen in Figure 1, for a majority of the Rodinia workloads simulated by us in gem5-GPU, this latency can be as much as or greater than the kernel execution time on the GPU. These high costs in terms of latency and energy consumption need to be amortized by having greater simultaneous computation on the GPU. This results in the current coarse-grained pattern of CPU-GPU coordination and computation. These penalties are greatly reduced in the case of integrated GPUs. However, the
lack of a cache-coherent memory hierarchy between the CPU and the GPU necessitates software-enforced coherence and synchronization through software flushes and barriers. These complicate the programming of CPU-GPU systems, and result in concurrency bugs. The problem of copying data between address spaces is also present in such systems.

All of these factors greatly motivate the need for a fully cache-coherent integrated CPU-GPU system. Consequently, AMD and other HSA Foundation members have committed to providing hardware coherence for heterogeneous systems. Hardware coherence allows the implementation of a shared address space for the CPU and GPU, thus simplifying data sharing. A shared virtual address space also allows the GPU to address a much larger memory space. Most importantly, such systems enable the ‘Pointer is a Pointer’ abstraction, which simplifies programming significantly, by allowing the GPU to access and use complex pointer-based data structures natively (without the need for marshalling). This greatly reduces the pre as well as post-processing required on the CPU-side, and allows efficient computation.

However, the throughput-oriented nature of GPU computation lead to several challenges in supporting coherence between the CPU and the GPU caches. The Single Instruction Multiple Thread (SIMT) execution model of GPGPUs which is supported by a high-bandwidth memory interconnect overwhelm conventional coherence structures like the memory-side directory and miss status handling registers (MSHR). Also, the minimum number of coherent requests generated per cycle was found to be two. Given that a directory supports a complicated state machine, multiporting the directory is very difficult and expensive.

In fully-coherent CPU-GPU systems, the directory emerges as a critical bottleneck, due to the difficulty in supporting multiple requests per cycle at the directory. Multiple requests can be handled by banking the directory, however this has limited potential for scaling as banking consumes significant power and area. This effect can be clearly seen in Figure 2 where the rate of GPU requests is 100-10000x the rate of CPU coherence requests. The rate of GPU coherence requests is calculated considering GPU execution time for a fair comparison, as GPU execution time is a small fraction of overall system execution time. In addition, Jason et al. [13] observed that tens of thousands of MSHR entries are needed to handle the flood of parallel requests from the GPU, in order to avoid stalls. To put this in perspective, the number of MSHRs in commercial systems currently extend to a few dozen registers, as MSHRs are built using power and area-intensive Content Addressable Memory (CAM).

As a result, our proposal, which will be elaborated in Section V is based around region coherence, which tracks coherence at coarse-granularities, and helps filter the large number of GPU coherence requests to a manageable amount.

4 Proposal

Our project proposal consists of three distinct avenues of exploration in the field of CPU-GPU coherence, each of which will be briefly discussed below and elaborated upon in the rest of the paper.
4.1 Extensions to Heterogeneous Systems Coherence

While work on studying CPU-GPU cache coherence has been carried out in [13], the GPU simulator used is a proprietary simulator based on the AMD Graphics Core Next architecture. Our initial aim was to implement and study these coherence issues on the open source gem5-GPU simulator to open up opportunities for further development on the same. While implementing this in gem5-GPU, further insight allowed us to propose modifications to the original design, to minimize hardware overheads and improve performance while mitigating the high bandwidth coherence traffic emerging from the GPU. Our design adds coarse-grained tracking of coherence only to the GPU-side, and attempts to improve performance with optimistic prefetching.

4.2 Suitable Workload Identification

We have been investigating newer workloads which could benefit from the fine-grained sharing enabled by CPU-GPU coherence. As part of this, we have been using the Sirius Suite [4], which contains individual algorithmic modules of a speech and vision based intelligent personal assistant (IPA), resembling Apple’s Siri [16]. Broadly, the suite consists of independent programs built for speech recognition, image matching and question answering.

4.3 Understanding Limitations of conventional protocols

Using heterogeneous system workloads from the Rodinia suite, we did a comparative study of the various conventional coherence protocols as applied to a CPU-GPU system. Towards this end, we considered VI_Hammer, MOESI_Hammer, and MESI_Two_Level coherence protocols, and evaluated it against a baseline non-coherent system. The metrics under consideration are relative execution times, and the coherence traffic observed by the directory.

5 Implementation

We explore the different coherence protocols listed below and compare the performance gain and coherence activity with each other and with the baseline software based coherence.

The following three coherence protocols have been evaluated -

- MOESI Hammer
- MESI Two Level
- VI Hammer

These protocols are briefly described in this section, while in-depth information about these can be found in [14].

5.1 MOESI Hammer

This is an implementation of the AMD’s Hammer protocol used in Opteron or Athlon 64 Processor. It is based on the HyperTransport protocol and utilizes a ProbeFilter along with a full-bit directory. It implements a private 2-level cache hierarchy with separate Instruction and Data L1 caches, and a unified L2 cache for each core.

MOESI Hammer does not maintain a directory state and instead broadcasts requests to all the processors in the system while fetching the data from DRAM in parallel.

5.2 MESI Two Level

This models a two level cache hierarchy with private L1 and Shared L2. Directory is responsible for on chip cache coherence and the directory information is co located with the corresponding cache blocks in the shared L2 cache.
5.3 VI Hammer

Similar to the MOESI hammer but employs a VI protocol, the hammer implementation remains the same. We chose this protocol to implement region coherence as it closely models a real life System on Chip design.

5.4 Region Coherence

All of the above protocols saw significant performance improvement, as seen in Figure 7. However, GPUs stream large amounts of data and often require read/write permissions to large data blocks. This does not sit well in the standard heterogeneous system because requests issued by the GPU hog the directory bandwidth not only impacting the performance of all the shader cores but also proving detrimental to the performance of the CPU. Figure 2 shows a comparison of the coherence activity from the GPU in comparison to the CPU. Note that these are not absolute values measured over execution time but are "requests/cycle". We believe that considering absolute value of coherence requests at the directory is a poor metric. This is because the kernel times for GPUs are very small as compared to the execution time on the CPU. This may lead to the ‘total’ coherence requests between the GPU and the CPU to be the same, while the rate of requests may show multiple orders of magnitude difference. As seen in the graph below, the activity from the GPU side may be hundreds of 1000s of times more than that from the CPU side.

In order to mitigate the immense bandwidth needed to support coherent GPUs, we implement region based coherence in our baseline system. For our baseline, we use a VI hammer based directory system. As shown in Figure 3a, we assume a separate CPU/GPU cluster with its private L2 cache. We add a region based array to the GPU L2 Controller (Figure 3b) to track access permissions at regional granularity. GPU L2 misses will first query the region buffer. If a valid permission for that region (shared for reads, private for writes) is found in the region buffer, requests are sent directly to memory via a direct bus. Else, requests are forwarded to the directory in order to acquire permissions for the region. The region directory connects the region buffers and tracks and arbitrates the permissions of all the regions on-chip. By obtaining permission at region granularity, this system routes the majority of L2 cache misses directly to memory, which reduces directory bandwidth. Figure 4, borrowed from [13], shows a representative diagram of how the lookup would happen in the Cache Controller. The hardware complexity of this system is modest, adding less than 1% to the total chip cache area.

In order to understand the working of such a system, let us consider a read request being initiated from the GPU side which results in an L2 Miss. The request is first looked up in the region coherence array to identify whether we have permissions for this region. If the valid bit is not set in the region coherence array, then multiple requests are generated to acquire permissions for all the cache lines within that region. Ideally, one would expect the directory to generate these requests. However, in our implementation we simulate this behavior by sending out individual requests for each cache line within a region via the region coherence array. We do this in order to handle the complexity of implementation in gem5-GPU. Once we have all the permissions, the request is served like a normal L2 cache miss. All
Figure 4: L2 Cache with Region Coherence Buffers

Figure 5: GPU L2 Cache Controller - Allocation
the subsequent access to the cache lines within that region would either be served by the GPU L2 or by memory directly without involving the directory.

Figure 5 shows the GPU L2 Cache Controller State Machine as implemented in VI Hammer. We integrate our Region Coherence Array with this state machine in order to achieve our objective. When a GETS or a GETX request arrives, a lookup into our Region Buffer (Region Coherence Array structure) is initiated. If region permissions are not already obtained, we initiate a corresponding PREFETCH_GETS or a PREFETCH_GETX for the region. This secures exclusive permissions for a region, and caches the entire region in the GPU L2 cache. The region invalidation in the original HSC design gets exclusive permission for a region. As all the cache lines are already invalidated in the CPU cache, optimistic prefetching of the region allows us to further improve performance.

As a result in our implementation, the Region Coherence is implemented via a prefetching request. Our experiments show that this is a reasonable assumption to make. We chose this method of implementation because our initial approach towards a permission-only based region coherence array was determined unfeasible when we hit some corner case scenarios which were extremely difficult to correct within the SLICC infrastructure in gem5-GPU. Although we were able to run few benchmarks using the permission based approach, in order to achieve a fully verified methodology, we switched to implementing prefetching based mechanism. The blue regions in Figure 5 show the states that had to be modified in order to allocate an entry to a region coherence array. Similarly states had to be modified in order to deallocate an entry in scenarios where a cache line has been removed from GPU L2. These states are highlighted in red in Figure 6.

6 Methodology

To evaluate the coherence protocols we ran Rodinia [2] and Sirius [4] benchmarks in gem5-GPU and on AMD R7200 Kaveri GPUs.

6.1 Platforms

6.1.1 gem5-GPU

The gem5-GPU simulator [12] was developed at University of Wisconsin-Madison to model tightly integrated CPU-GPU system. gem5-GPU is built on gem5 and gpgpu-sim and the simulator can run CUDA programs. For gem5-GPU the baseline configuration that we chose has been shown in Table 1. We
model a simple CPU, along with an AMD like GPU and all memory requests between CPU and GPU are through gem5 Ruby model.

6.1.2 AMD Hardware

We also ran some of our workloads on real hardware. To evaluate the benchmarks on real hardware, we chose AMD R7200 Kaveri GPUs. Our system consisted of a dual boot OS, first one being an Ubuntu OS with HSA drivers and the other one being OpenSUSE OS with OpenCL drivers. To run the workloads all of the CUDA programs were converted to OpenCL and HSA. ¹

6.2 Workloads
6.2.1 Sirius Suite

In order to find an application with fine grained sharing, we looked at the Sirius benchmark suite. Sirius is an end-to-end IPA (Intelligent Personal Assistants) web-service application that accepts queries in the form of voice and images, and responds with natural language. This workload is meant to investigate the implications of accelerator-based server architectures spanning traditional CPUs, GPUs, manycore throughput co-processors, and FPGAs. Sirius Suite contains 7 different applications, which are based on logically distinct modules from their personal assistant program. Table 2, summarizes the 7 benchmarks present in sirius along with the availability of CPU and GPU implementations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GPU Implementation Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Mixture Model (Audio Speech Recognition)</td>
<td>Yes</td>
</tr>
<tr>
<td>Deep Neural Network (Audio Speech Recognition)</td>
<td>Yes</td>
</tr>
<tr>
<td>Stemmer (Question-Answering)</td>
<td>Yes</td>
</tr>
<tr>
<td>Conditional Random Field classifier (Question-Answering)</td>
<td>No</td>
</tr>
<tr>
<td>Regular Expression (Question-Answering)</td>
<td>No</td>
</tr>
<tr>
<td>Feature Extraction using SURF (Image Matching)</td>
<td>Yes</td>
</tr>
<tr>
<td>Feature Description with ANN (Image Matching)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2: Summary of benchmarks present in Sirius Suite

- **Gaussian Mixture Model**: This benchmark replicates the automatic speech recognition algorithm implemented for Google voice. It reads the input speech data from a text file and uses an Expectation Maximization based approach to compute the scores for identifying the speech pattern. The EM step is performed as a separate function for CPU and GPU. While the GPU handles score computation, rest of the functions such as file reading or data pre-processing is done in CPU.

- **Deep Neural Network**: This benchmark implements speech recognition using deep neural networks. It does not include a training phase and reads the learnt weight values from a file. The computations are performed on GPU using the Caffe Library [7].

¹The changes will be submitted for inclusion in the main repository of sirius benchmark.
• **Stemmer**: Stemmer is one of the three query answering modules implemented in sirius benchmark. Stemmer uses an NLP based approach for removing the commoner morphological and inflexional endings from words in English to find the root of a word.

• **Feature Extraction**: This is one of the first steps towards image matching. Sirius-suite implements an openvc based SURF feature extraction technique. The executable takes image as an input and computes SURF based features either in CPU or in GPU.

• **Feature Description**: Once features are extracted, it passes through a description phase that characterizes a feature. The benchmark computers SURF descriptor either on a CPU or a GPU using the OpenCV library.

6.2.2 Rodinia Suite

Rodinia benchmark suite has been developed for evaluating the performance of heterogeneous systems. We use the following benchmarks from this suite -

• **Back propagation**: Back propagation is a machine-learning algorithm which trains the weights of connected nodes on a layered neural network.

• **Breadth first search (BFS)**: BFS is a breadth-first-search algorithm which can be implemented in parallel.

• **Hotspot**: Hotspot computes the final state of cell grids when the initial conditions temperature and power dissipation per cell is given.

• **LU decomposition (LUD)**: LUD decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix.

• **Needleman wunsch (NW)**: NW is a dynamic-programming algorithm for sequence alignments.

• **Kmeans**: K-means is a clustering algorithm which uses the data partitioning method to do dense linear algebra calculations.

• **Srad**: Srad is a diffusion method used in ultrasonic and radar imaging applications. It is used to remove locally correlated noise known as speckles without destroying important image features.

• **Gaussian**: Gaussian elimination computes results row by row and solves all the variables in a linear system.

• **LavaMD**: LavaMD computes particle potential and relocation due to mutual forces between particles within a large 3D space.

• **Nearest Neighbour (NN)**: NN evaluates the k nearest neighbors using euclidean distances.

• **Pathfinder**: Pathfinder uses dynamic-programming algorithm to find shortest path of a 2D grid.

7 Evaluation

7.1 Rodinia benchmarks

Figure 7 shows a comparison of the performance obtained from the different hardware coherence protocols along with a software based coherent (with cuda memcpy) model as obtained from running benchmarks from the Rodinia suite on gem5-GPU.

It can be observed that in most benchmarks, at least one or all of the hardware coherence protocols perform better (in terms of kernel time) as compared to the software based coherence protocol implementation. This validates our assumption that without a coherent model, the software overheads involved will have a significant impact on performance benefits of fine grained data sharing.

It is important to note here, that we show performance in terms of GPU ticks i.e. kernel execution time alone and not in terms of total CPU+GPU time. This is because in most current day benchmarks such as the Rodinia suite, there is not sufficient fine grained sharing of data. The GPU kernels are rather only to work on discrete pieces of code meant for specific purposes which are usually decoupled from the code execution on the CPU. But with a bigger push towards more closely integrated CPU-GPU
systems from different directions (e.g., AMD), we believe that there will be new opportunities to exploit fine-grained sharing thereby increasing the need for coherence models or additional features to support fine-grained coherence.

Figure 8 clearly highlights why region coherence is important. It shows the reduction in coherence activity at the directory when we implement a region buffer on the GPU side.

In the above graph, a value of 1 represents the activity for the GPU/directory in the case of VI Hammer and as can be seen above, a number of benchmarks show a significant reduction in coherence activity when using a region buffer. As explained earlier, streaming accesses from GPUs make region level permissions useful in bypassing the directory after permission is obtained and thereby reducing the pressure at the directory.

### 7.2 Sirius benchmarks

To run sirius-suite workload on gem5-GPU, one of the tasks was to have a statically compiled object file. Only GMM and Stemmer are evaluated on gem5-GPU as we were not able to find a static library for many of the dependencies of other benchmarks.

Table 3 shows the statistics obtained for executing the GMM benchmark on gem5-GPU. As we can
Table 3: Coherence requests from CPU and GPU side while executing the GMM benchmark for sirius-suite.

<table>
<thead>
<tr>
<th></th>
<th>GPU w/ cudaMemCpy</th>
<th>GPU w/o cudaMemCpy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-GETX</td>
<td>198246</td>
<td>132209</td>
</tr>
<tr>
<td>OTHER-GETX</td>
<td>66065</td>
<td>31</td>
</tr>
<tr>
<td>CPU-GETS</td>
<td>2189053</td>
<td>2110798</td>
</tr>
<tr>
<td>OTHER-GETS</td>
<td>126267</td>
<td>60271</td>
</tr>
</tbody>
</table>

Observe that when cudaMemCpy command is implemented, the number of OTHER-GETX and OTHER-GETS requests sent by GPU is considerably high when compared with implementation that does not have cudaMemCpy command implemented.

Sirius Benchmarks were also evaluated on real hardware (Figure 9), with the OpenCL implementation having a split memory hierarchy between the CPU and the GPU while the HSA implementation uses the coherent interconnect. The HSA implementation is created by compiling the OpenCL kernel using a CLOC tool provided by AMD, and rewriting the CPU side code. We have been unable to explain the relatively poor performance of the HSA implementation, given that the same kernel is used as the OpenCL implementation. Also, little documentation or support is provided by AMD for their HSA drivers and compiler. Also, the relatively limited program structures supported by the HSA compiler precluded us from generating HSA code for all but 2 Sirius workloads.

Figure 9: Performance of Sirius benchmarks on real hardware

Since the benchmarks were written assuming a split CPU GPU system, we do not see a lot of fine grained sharing. However, such applications show a great potential of using fine grained sharing if they are provided with the right software and hardware support. We hope that region coherence would be one of the few steps that need to be considered for this goal.
8 Related Work

8.1 GPU Coherence

Temporal coherence [15] proposes a cache coherence scheme based on synchronized counters within the GPU to improve performance. The use of these globally synchronized counters enable coherence transitions to happen synchronously without the need for coherence messages and thus eliminates race conditions. CPU-style read for ownership coherence is rejected by Hechtman et. al. [5] as unsuitable for streaming workloads. Their QuickRelease approach uses separate read and write buffers to enforce order between GPU memory requests, and achieve fine-grained memory synchronization.

8.2 Managing Broadcast Bandwidth

The idea of minimizing broadcast bandwidth has been explored before, for traditional CPU coherence. JETTY [11] proposed by Moshovos et al. uses knowledge of contiguous regions to filter out incoming snoops into the cache. On a similar note, RegionScout [10] adds a non-shared region table, a intellectual precursor to the region coherence array described in this work, to track regions known to be exclusive. Additionally, a bloom filter called a cached region hash holds a superset of the regions cached in each local cache. These two structures are used in order to filter out some of the broadcast bandwidth.

9 Conclusion and Future Work

Several key takeaways have emerged from our explorations in the area of CPU-GPU coherence. While CPU-GPU coherence is essential to simplify GPU programming while reducing the energy and latency costs associated with data transfers, conventional coherence schemes cannot be extended naively to include GPU caches. The high bandwidth of coherence requests generated due to the SIMT execution model can overwhelm the coherence network, and expose the directory as a critical bottleneck. Based on our experiments, the rate of GPU coherence requests is about 100-10000X the rate of CPU coherence
requests. This problem can be mitigated by employing region coherence, which is an efficient way of reducing the coherence traffic emerging from GPUs by almost 10X in our simulations. We proposed extensions to the original HSC design to minimize the hardware overheads, and improve performance.

Furthermore, we found certain limitations in the hardware implementation of the HSC design, which haven’t been described in the original paper. The region invalidate command from the directory to the L2 cache controllers invokes a number of invalidation requests, one per cached line from the region. This necessitates additional hardware in the L2 cache controller to handle the simultaneous multiple invalidations, and also consumes significant memory bandwidth. This could also lead to a serialization of the invalidation acknowledgements or writebacks sent to the directory, which could adversely affect the performance of the system.

However, we realized that the main performance benefits of a unified coherent address space can be exploited only by having fine-grained data sharing between CPU and the GPU. This is in sharp contrast to the current model of GPU programming, which came about as a way to get around the limitations of separate CPU and GPU memory systems. For this work, we tried finding such workloads to correctly gauge the true potential of fully coherent systems, but only found one, Sirius Suite, which while not exactly satisfying the requirements, was amenable to change. In Figure 10, we show an example of the DNN benchmark from Sirius Suite, modified with fine-grained sharing between the CPU and the GPU, in order to better exploit CPU-GPU coherence.

Our belief is that software developers need to move towards newer models of CPU-GPU programming, to create greater impetus for hardware manufacturers to support better coherence schemes. However, even after two years of its launch, there has not been any significant push from the community to achieve this as is visible from this [3] empty web page on the Heterogeneous Foundation Website describing applications based on coherent CPU-GPU Systems.

10 Statement of Work

The statement of work for the project signed by all the team members is presented below.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Gokul</th>
<th>Umish</th>
<th>Swapnil</th>
<th>Rohit</th>
<th>Han</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementing modified HSC in gem5-gpu</td>
<td>20%</td>
<td>50%</td>
<td>30%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>Porting Sirius-Suite to gem5-gpu</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>80%</td>
<td>20%</td>
<td>100%</td>
</tr>
<tr>
<td>Converting Sirius-Suite to OpenCL and HSA</td>
<td>0%</td>
<td>0%</td>
<td>80%</td>
<td>0%</td>
<td>20%</td>
<td>100%</td>
</tr>
<tr>
<td>Evaluation on gem5 - gpu</td>
<td>50%</td>
<td>10%</td>
<td>10%</td>
<td>20%</td>
<td>10%</td>
<td>100%</td>
</tr>
<tr>
<td>Literature Study</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>100%</td>
</tr>
<tr>
<td>Simulator setup + config + related overhead</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>100%</td>
</tr>
</tbody>
</table>

References


