# Heterogeneous System Coherence for Integrated CPU-GPU Systems

Jason Power* powerjg@cs.wisc.edu

Arkaprava Basu* basu@cs.wisc.edu

Junli Gu† junli.gu@amd.com

Sooraj Puthoor† sooraj.puthoor@amd.com

Bradford M. Beckmann† brad.beckmann@amd.com

Mark D. Hill*† markhill@cs.wisc.edu

Steven K. Reinhardt† steve.reinhardt@amd.com

David A. Wood*† david@cs.wisc.edu

*Department of Computer Sciences
University of Wisconsin – Madison

†Advanced Micro Devices, Inc.

## ABSTRACT

Many future heterogeneous systems will integrate CPUs and GPUs physically on a single chip and logically connect them via shared memory to avoid explicit data copying. Making this shared memory coherent facilitates programming and fine-grained sharing, but throughput-oriented GPUs can overwhelm CPUs with coherence requests not well-filtered by caches. Meanwhile, region coherence has been proposed for CPU-only systems to reduce snoop bandwidth by obtaining coherence permissions for large regions.

This paper develops *Heterogeneous System Coherence* (HSC) for CPU-GPU systems to mitigate the coherence bandwidth effects of GPU memory requests. HSC replaces a standard directory with a region directory and adds a region buffer to the L2 cache. These structures allow the system to move bandwidth from the coherence network to the high-bandwidth direct-access bus without sacrificing coherence.

Evaluation results with a subset of Rodinia benchmarks and the AMD APP SDK show that HSC can improve performance compared to a conventional directory protocol by an average of more than 2x and a maximum of more than 4.5x. Additionally, HSC reduces the bandwidth to the directory by an average of 94% and by more than 99% for four of the analyzed benchmarks.

## Categories and Subject Descriptors

C.1.3 [**Processor Architectures**] Other Architecture Styles—Heterogeneous (hybrid) systems; B.3.2 [**Memory Structures**]: Design Styles—cache memories;

## General Terms

Performance, Design

## Keywords

GPGPU computing, Heterogeneous computing, Cache coherence, Coarse-grained coherence

## 1. INTRODUCTION

General-purpose graphics processing unit (GPGPU) computing offers great potential to accelerate many types of applications with high-bandwidth throughput-oriented cores [24]. Moreover, as compute capabilities increase, we expect memory bandwidth will also increase to satisfy their future demands. The High Bandwidth Memory (HBM) task group has been working to define a standard to deliver bandwidth ranging from 128 GB/s to 256 GB/s per DRAM stack [1]. In addition, die-stacking memory technologies have been proposed to stack multiple DRAM stacks[1] on the same die to provide bandwidth up to 1 TB/s.

At the same time, GPUs are becoming more tightly integrated with conventional CPUs in two ways. First, they are being *physically* integrated on the same chip. Current examples include AMD's Trinity [4] and Intel's Ivy Bridge processors. Such heterogeneous processors currently make up more than 50% of client PC orders and will become the vast majority by 2014 [30].

Second, GPUs are becoming more *logically* integrated with CPUs through support for a unified (shared) memory address space. This integration frees the programmer from using explicit copies and enables use of unmodified pointer-based data structures ("pointer is a pointer" [19]).

To make a shared address space appear sensible to application programmers, some type of coherence should be implemented in software (libraries), software-hardware, or in hardware. Cohesion [16] implements it by a combination of hardware and software coherence. Asymmetric distributed shared memory (ADSM) implements a logically shared address space between the CPU and GPU through a software implementation, GMAC [14]. However, these mechanisms can burden the programmer [16] and affect performance [14].

This paper focuses on supporting ***hardware coherence between CPUs and GPUs*** in a heterogeneous CPU-GPU system. We choose hardware because CPUs will continue to implement hardware coherence for the foreseeable future [21], and to ease software's burden. Additionally, AMD and other HSA Foundation members have committed to providing hardware coherence for

[1] eight stacks in Black et al. [10]

heterogeneous systems. Hardware coherence can enable new classes of applications to take advantage of GPGPU computing through low-overhead fine-grained data sharing.

Supporting coherence between CPUs and GPUs is challenging. GPGPU applications often require much higher memory bandwidth than CPU applications due to the massively parallel nature of their Single Instruction, Multiple Thread (SIMT) execution model. GPGPU applications also have different memory access patterns than most CPU applications. Due to the streaming nature of many GPGPU applications, they may exhibit higher spatial and lower temporal locality than CPU applications. Our performance evaluation of Rodinia benchmarks shows that the GPU shared L2 cache has an average miss rate of 58%, which is much higher than the average CPU miss rate of 14% [12].

Because of the high-bandwidth nature of heterogeneous systems, using coherence mechanisms created for CPU architectures is difficult. Traditional directory designs have two main problems when scaling to a high bandwidth. First, it is challenging to support more than one request per cycle at the directory. While it is possible to bank the directory heavily, replicating all of the required structures is expensive in terms of both power and area. Second, full coherence necessitates that each GPU request allocates a miss status handling register (MSHR) at the directory to handle potential races with competing CPU requests. Many MSHR entries are required to deal with the numerous parallel requests. For the Rodinia workloads, we find that many applications need *tens of thousands* of MSHR entries to avoid stalling the GPU. The number of required MSHRs is impractical for a real machine, which today has only a few dozen.

To reduce the bandwidth required in snooping-based symmetric multi-processor (SMP) systems, researchers have proposed region coherence [3, 11, 23, 31]. Region coherence exploits coarse-grained sharing patterns among processors only requiring a subset of memory requests to be broadcast to all cores. If permissions for the requested region have been obtained already, broadcasts are filtered and requests proceed directly to memory. Given that GPGPU applications employ mostly coarse-grained sharing between the CPUs and GPUs, the concept of region coherence lends itself well the requirements of hardware coherence in heterogeneous systems.

To this end, we propose Heterogeneous System Coherence (HSC), which is directory-based hardware coherence on heterogeneous CPU-GPU systems. HSC adds region buffers to both CPU and GPU L2 caches to track the regions over which the CPU or GPU currently hold permission. The region directory replaces the traditional block-level directory. In addition to reducing bandwidth by moving most coherent requests onto the incoherent *direct-access bus*, the region directory occupies less area and is simpler than a conventional block-level directory.

The main contributions of this paper are:

- A characterization of coherence bottlenecks in future high-bandwidth heterogeneous systems showing that limited directory resources are a significant bottleneck.
- A redesign of region-based coherence for heterogeneous architectures that addresses the challenge of heavy resource requirements.
- Optimizations for coherence-related traffic to reduce the directory congestion bottleneck by bypassing the directory and accessing memory directly for a majority of L2 cache misses.

We implemented HSC on a cycle-level simulator. Evaluation results show that HSC reduces the resource requirements while achieving an average performance improvement of 2x and a maximum performance improvement of 4.5x compared to the baseline coherent heterogeneous system. Bandwidth to the directory is reduced by an average of 94% and by more than 99% for most benchmarks with limited MSHRs.

In this paper, the next section introduces the heterogeneous CPU-GPU system that serves as the baseline system in this paper. Section 3 illustrates the bottlenecks of extending directory-based coherence protocols on the baseline system. Section 4 describes the HSC design and implementation and estimates the hardware complexity. Section 5 describes our simulation infrastructure and the benchmarks we used to evaluate HSC. Section 6 reports and discusses results. Section 7 reviews previous work on GPU coherence and region coherence. Section 8 contains concluding remarks.

## 2. BACKGROUND

This section introduces the baseline heterogeneous system based on current mainstream integrated CPU-GPU products. We also describe the extension of directory-based coherence to heterogeneous systems, which will serve as the baseline coherence implementation.

### 2.1 Baseline Heterogeneous System

Figure 1(a) shows an overview of our baseline system. There are two clusters: a CPU cluster, which can contain any number of CPUs, and a GPU cluster made up of thousands of individual scalar units. These scalar units are coalesced to form SIMD units that execute instructions in SIMT fashion. These SIMD units are grouped together into compute units (CUs). 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 GPU clusters have two separate, non-inclusive, shared L2 caches. The baseline coherence protocol is hierarchical coherence with a global block-level directory that has two bits for the sharing vector (one for each cluster).

The GPU uses the non-coherent direct-access bus for all incoherent memory traffic, which mostly includes graphics-related traffic. For coherent GPU requests, there is a separate coherent interconnect between the GPU and the directory. All general-purpose memory traffic (requests to the CPU-GPU shared address space) generated in GPGPU applications must use this interconnect and access the directory to stay coherent with the CPU caches. The GPU L1 caches are kept coherent by writing through dirty data and flash-invalidating the L1 caches at kernel begins. Section 5 provides a more detailed description of the simulated system.

### 2.2 Coherent Cache Implementation

Figure 1(b) shows an example architecture of the L2 cache tags and required hardware for lookups and probes. MSHRs are used to track the ongoing transactions of the outstanding requests. The MSHR table is implemented using content-addressable memory (CAM) to facilitate finding conflicts among the outstanding addresses. A request occupies an MSHR until it has been processed completely either by a hit in the L2 cache or an external response.

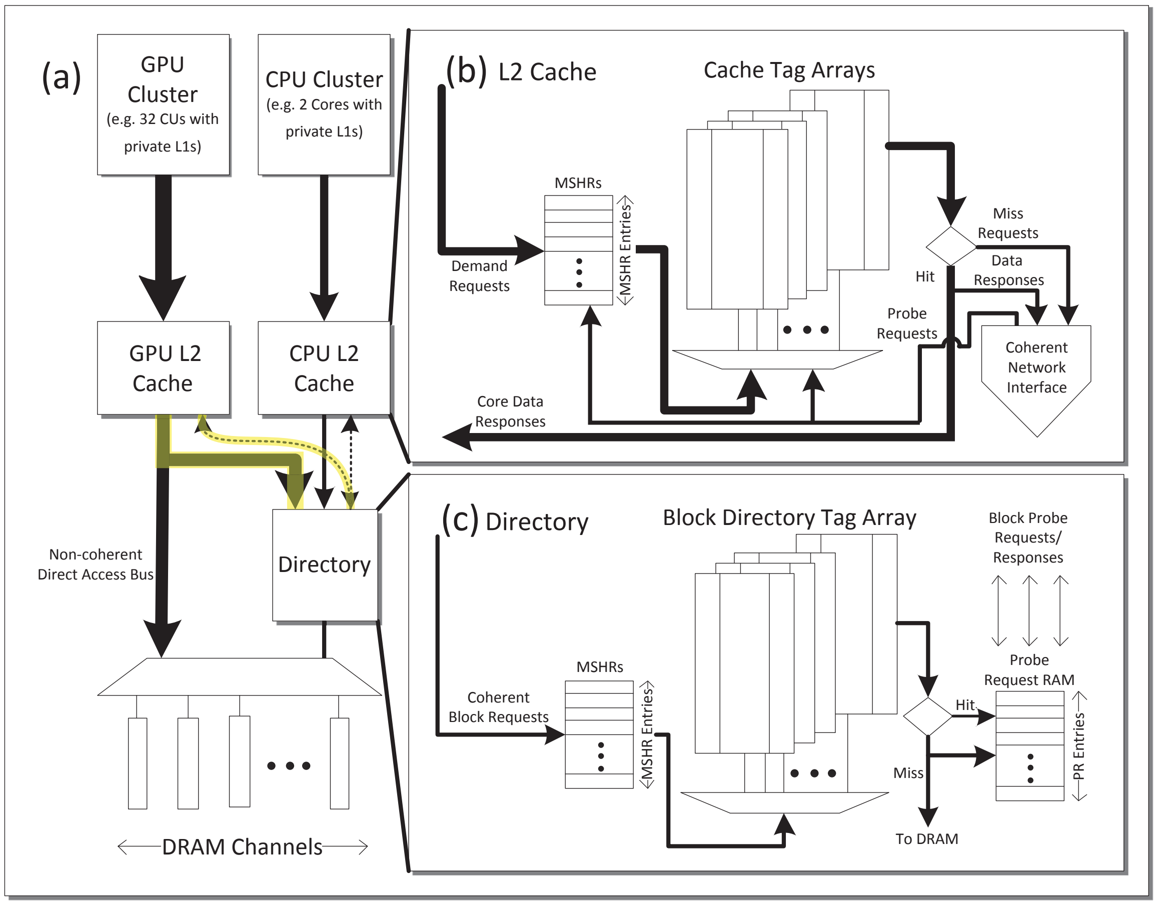

**Figure 1: Baseline heterogeneous system. The weight of the lines represents the bandwidth required between components. Connections added to support heterogeneous coherence are highlighted. (a) shows the baseline system under study in this paper. The dotted lines represent probe/invalidation requests and responses. (b) and (c) are detailed diagrams of the designs of the L2 cache and the directory, respectively, in our baseline system.**

Figure 1(c) shows the baseline directory implementation. The directory MSHR table, also implemented as a CAM, is similar in purpose to the MSHR table at the L2 cache. It is used to find any conflicts with outstanding coherence requests to both memory and L2 caches. The probe-request RAM (PRR) is used to track outstanding probes and is implemented as a RAM indexed on coherence transaction ID. The directory is mostly inclusive of all caches in the system. 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. To initiate a probe request, a PRR entry must be allocated and the PRR entry stays occupied until all required caches have replied to the probe requests. While scaling the PRR to support many outstanding probe requests is relatively easy, scaling the directory MSHR table to support many outstanding requests and multiple incoming requests per cycle is much more difficult and costly due to the CAM design.

The probe requests are sent to any shared L2 caches that have a valid copy of the data. On receiving the probe request, the L2 cache tag array must be read to determine the state of the block for that address. The L2 MSHR table also must be queried to deal correctly with the case of an outstanding request for that address. These probe requests interfere with any demand requests from the L1 caches and may cause increased latency for those requests. Also, arbitration logic between demand and probe requests is required. Scaling the cache tags and MSHR tables to support more than one probe per cycle is difficult and costly. As with the directory MSHR table, multi-porting the L2 MSHR table is costly.

An alternative implementation of the directory is a stateless or null directory [7]. In this configuration, there are no directory tags. Every request is treated as a directory hit and probes are broadcast to all caches in the system. This design simplifies the directory considerably, but suffers from high bandwidth of probe requests at the L2 caches.

## 2.3 Region Coherence

Region coherence was first proposed in 2005 to reduce the bandwidth required on snooping-based systems [3, 11, 23, 31]. Region coherence filters many broadcasts by tracking the sharing information on a coarse granularity (a region). Regions contain many (16–64) cache blocks. Coarse-grained coherence [11] adds a Region Coherence Array (RCA) to the local caches. The RCA is checked on cache misses, and if permissions exist for the region to which the request address belongs, the request is forwarded directly to memory without requiring a broadcast. On RCA misses, the request is broadcast and all other caches reply with the correct region-level permissions, which are inserted into the RCA. In this work, we build on the RCA design, but we expect our contributions to generalize to other region coherence implementations.

### 2.4 Heterogeneous Coherence

Historically, coherence among CPUs and GPU has been managed by software through explicit copies between address spaces. Explicitly managed software coherence complicates heterogeneous applications by requiring the programmer to reason about multiple addresses for each variable. Additionally, this software coherence makes using pointer-based data structures—like linked-lists and trees—difficult on heterogeneous hardware.

In addition to complicating the programming model, explicit software-managed coherence can perform poorly. Although it is theoretically possible to overlap data movement between the CPU and GPU with computation, expressing the dependencies is difficult in current programming models. This leads to inefficiencies in application performance as computation and data movement are serialized.

To mitigate these issues and increase performance, AMD and other HSA Foundation members have committed to providing hardware coherence for heterogeneous systems [25]. Hardware coherence allows the implementation of a shared address space for the CPU and GPU, which simplifies data sharing. A shared virtual address space also allows the GPU to address a much larger memory space (all of CPU memory). Therefore, our baseline system assumes hardware coherence between the CPU and GPU in our system.

## 3. HETEROGENEOUS SYSTEM BOTTLENECKS

We performed a characterization of the bottlenecks of the coherence protocol in the baseline heterogeneous system (Figure 1). The bottlenecks are primarily due to the high memory bandwidth of GPU computing, which generates more coherent requests than the directory can maintain and also requires expensive MSHR resources.

### 3.1 Memory Bandwidth

GPU compute units are throughput-oriented many-core processors targeting streaming-style applications. The streaming access pattern is quite different from most CPU applications and causes the GPU caches to be ineffective at filtering DRAM accesses. Thus, GPU computing often consumes high memory bandwidth to keep all the SIMD execution units busy. For example, recent discrete GPUs from NVIDIA and AMD support bandwidth close to 300 GB/s [6, 13]. Even though integrated GPUs are less powerful, the number of CUs on these GPUs will likely continue to scale in the future to increase computing power. Therefore, memory bandwidth must scale as well.

To meet the challenge, both academia and industry have proposed new memory technologies such as die-stacking memory. Assuming such technology will be incorporated in future systems, we target a high memory bandwidth for the rest of paper to reflect this design trend. Specifically, for a GPU composed of 32 CUs, we found that 700 GB/s eliminated the memory bandwidth bottleneck for all our workloads.

### 3.2 Directory Bandwidth

As memory bandwidth increases, other bottlenecks come to light. Specifically, we find that the number of coherent data requests is more than the directory can handle and requires a currently infeasible number of MSHRs. MSHRs are used to track the status of in-progress requests at the directory. This includes directory hits, for which an MSHR entry is occupied until all probe requests have completed, and directory misses, for which an MSHR entry is occupied until memory responds with data.

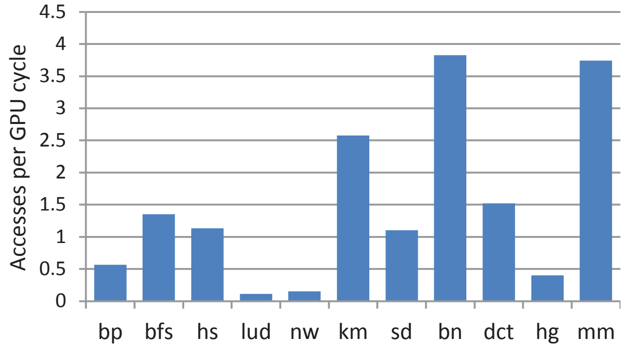

**Figure 2(a): Number of directory accesses per GPU cycle for directory-based coherence.**

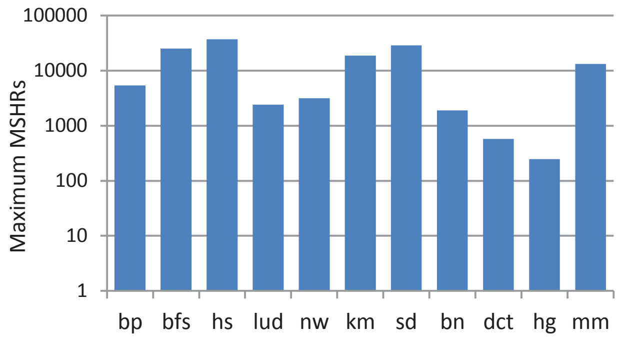

**Figure 2(b): Maximum resources used for directory-based coherence (log scale).**

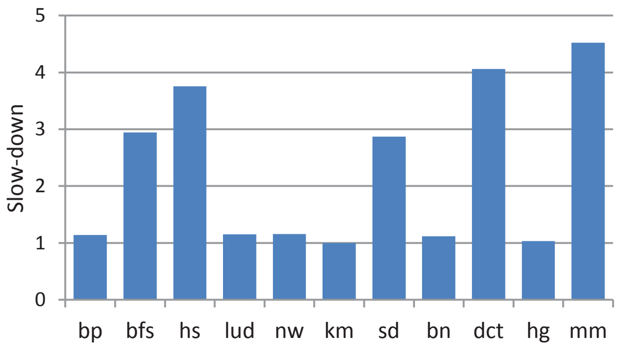

**Figure 2(c): Slowdown for baseline directory protocol when using constrained resources (32 MSHRs) compared to unconstrained resources.**

Figure 2(a) shows the average accesses per GPU cycle at the directory for our workloads (see Section 5 and Table 1 for simulation details). For many applications, the bandwidth is more than two accesses per cycle, which is difficult to support at the directory. While it is possible to bank the directory array heavily, splitting the pipeline for accessing the directory is much more difficult.

High memory bandwidth also requires numerous MSHRs to track all the simultaneous outstanding data requests. Because MSHRs are CAM structures, it is expensive in terms of both power and area. The MSHR structure is difficult to multi-port due to the large number of entries it must hold. One can imagine address-slicing the MSHRs *n* ways, but a lower value of *n* will encounter a significant number of conflicts while a higher value of *n* wastes many extra entries due to uneven request rates across the address slices.

To illustrate the high demand of MSHRs for future heterogeneous systems, Figure 2(b) shows the maximum MSHRs used by our workloads when given unlimited resources. As shown in the figure, these applications have a high variance in required MSHRs. Some applications, such as hg (histogram), need relatively few MSHRs (450) due to low utilization of the GPU, with an average of less than 20% of the GPU CUs used during execution. However, many applications need a massive number of resources. Six of our applications use more than 5,000 MSHRs at peak usage. These applications will suffer significantly from queuing delays when directory resources are constrained. Figure 2(c) shows the slowdown of these applications with 32 MSHRs. Matrix multiplication slows by more than 4x, and five workloads suffer a more than 2x slowdown. The average slowdown across all of the workloads is 2.25x.

To implement coherent caches for GPU workloads, the overheads must be reduced. Heterogeneous System Coherence, presented in the next section, mitigates many of these overheads.

## 4. HETEROGENEOUS SYSTEM COHERENCE

In this section, we discuss the design of HSC. We first extend region coherence to directory-based coherence protocol, and then we apply it to our baseline system. HSC adds region buffers to both CPU L2 and GPU L2 caches to track access permissions at the region granularity. All the L2 misses first query the region buffer. If valid permission for that region (shared for reads, private for writes) is found in the region buffer, data requests are sent directly to memory via the direct-access bus. If permission is not found, requests are forwarded to the region directory to acquire permission 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, HSC routes the majority of L2 cache misses directly to memory, which reduces directory bandwidth. The hardware complexity of HSC is modest, adding less than 1% to the total chip cache area.

### 4.1 Directory-based Region Coherence

To apply the principles from region coherence to future heterogeneous systems, similar to spatiotemporal coherence [3], we first adapt region coherence to a directory-based coherence protocol. To this end, we replace the block-level directory in the baseline with a region directory and add a modified RCA [11] at each shared L2 cache.

The region directory is conceptually similar to a normal block-level directory. The block tags are replaced with region tags, and the state associated with each entry is stored at a region granularity. The region directory tracks the state only at the region granularity; therefore, if a single block in the region is shared between two cores, the entire region is marked as shared. Figure 3(a) shows a breakdown of the directory entry.

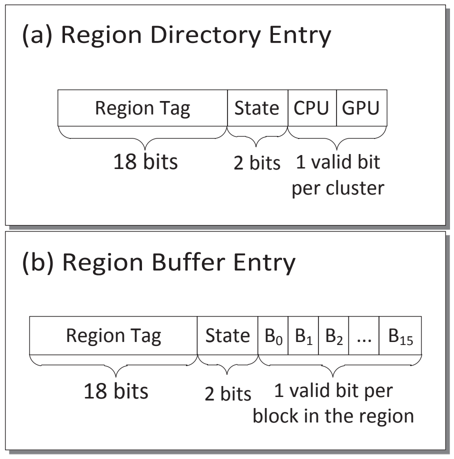

**Figure 3: Example region directory (a) and region buffer (b) entries. Assumes 1 KB regions (16 64-byte blocks), 40-bit physical addressing, and the configurations in Table 1.**

In coarse-grained coherence, the RCA sits behind the cache and tracks the region state and the sharers. However, in directory-based region coherence, there is no need for the sharers to be stored at the cache level. We replace the RCA with a simplified structure, the region buffer, which holds only the region state because the sharers are stored in the region directory. Like the RCA, the region buffer is inclusive of all entries stored in the caches above it. Figure 3(b) shows the region buffer entry.

The region directory needs to be accessed only on requests for regions that are not cached in the region buffer. 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.

### 4.2 Heterogeneous System Coherence

HSC takes directory-based region coherence and applies it to the baseline heterogeneous system. Figure 4(a) shows an overview of a system with HSC.

HSC takes the huge bandwidth on the coherence network in the baseline system and moves it to the incoherent direct-access bus by allowing *coherent direct access*. Coherent direct access is achieved by acquiring permission for the entire region when the first block is accessed. The subsequent accesses to any blocks in the same region can perform coherent direct access on the incoherent direct-access bus, which directly accesses memory without requiring a directory query. When there is high spatial locality in the memory access stream, such as for streaming applications, most requests will not need to access the region directory because the permissions already will have been obtained. All requests for which permissions exist in the region buffer can then use the high-bandwidth direct-access bus instead of the lower-bandwidth coherence network.

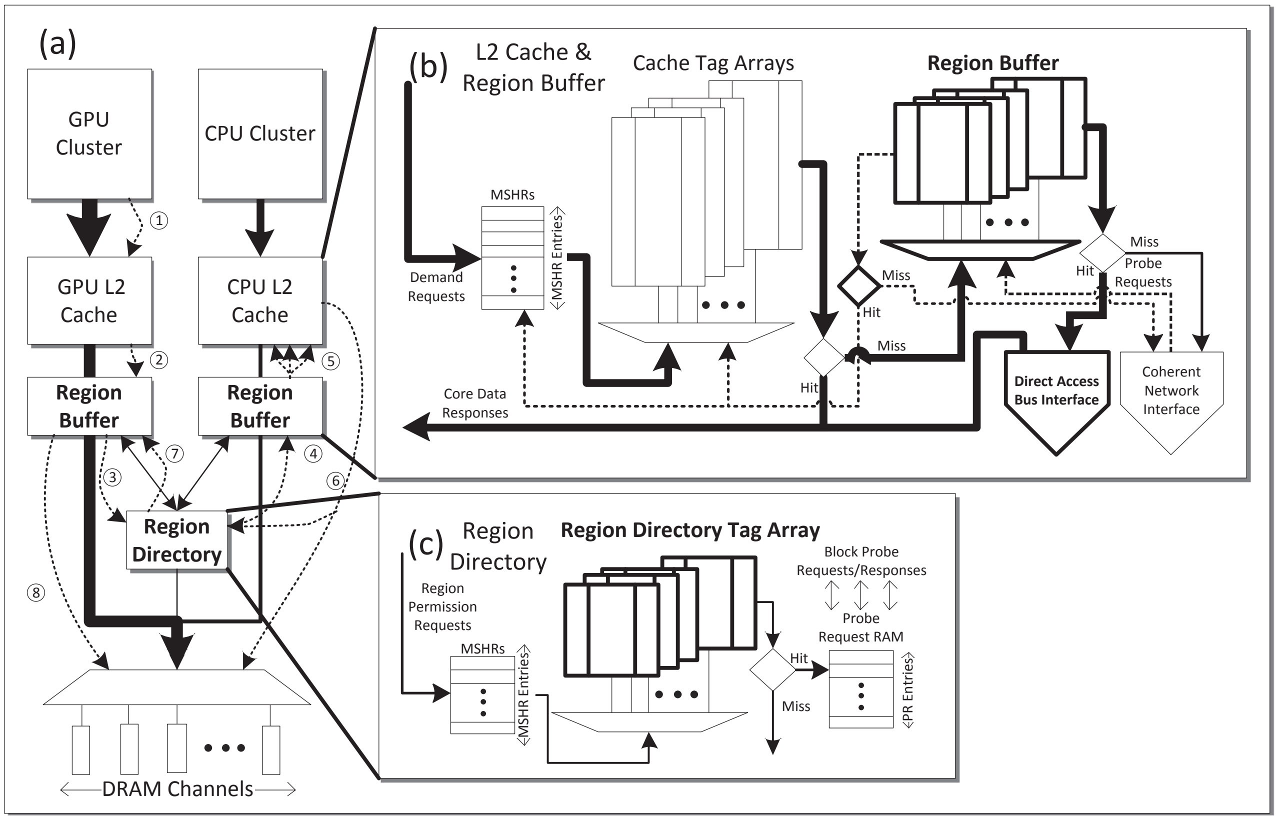

**Figure 4: Heterogeneous System Coherence overview. Line weight represents bandwidth. Changes from the baseline are bolded. (a) shows the system architecture of HSC. The circled numbers refer to the detailed example of a GPU memory request in Section 4.2. (b) shows a detailed representation of the L2 cache and Region Buffer. Dotted lines show probes. (c) shows a detailed representation of the region directory.**

Figure 4(a) shows an example of how a memory request flows through the system. ① The GPU issues a write request (exclusive request) for address A. ② The request misses in the GPU L2 cache and is forwarded to the region buffer. Address A is part of region R(A), which is not present in the GPU region buffer; therefore, ③ a region-exclusive request is sent to the region directory. When the request for R(A) reaches the region directory, there is a hit, which results in ④ an invalidate probe being sent to the CPU region buffer along with the demand request. On receiving the invalidate probe for region R(A), ⑤ the CPU region buffer forwards the demand request and invalidates every block that is part of the region and is valid in the CPU shared L2 cache. ⑥ The CPU shared L2 cache responds to the demand request and writes back the data for all of the blocks in the region that were valid. Once all of the blocks have been evicted from the CPU shared L2 cache, ⑦ the region directory responds with the data for the original request and gives private permissions to the GPU region buffer. For subsequent requests from the GPU cluster that miss in the GPU shared L2 cache and are part of region R(A), ⑧ the GPU region buffer sends the coherent request on the direct-access bus to memory, which returns the data.

## 4.3 Region Buffer

The region buffer contains the region tags, permissions, and other region metadata that are kept at the local caches. The modifications required at the shared L2 cache to implement the region buffer are highlighted in Figure 4(b).

The biggest change from the baseline shared L2 cache is the addition of the region buffer. This is a relatively small structure that is banked easily to support high-bandwidth accesses and is indexed by the region tag—the most significant bits of the address. Each entry in the region buffer holds the region permission: private with read-write permissions, shared with read-only permissions, or invalid. A bit-vector that is the size of the region (16 bits for a 1 KB region) is also included in each entry to track the valid blocks in the region to minimize the required invalidates sent to the L2 cache tags.

An interface to the direct-access bus is also added to the shared L2 cache. This interface is quite similar to the one present in GPUs to carry graphics memory requests.

With the addition of the region buffer, any requests that miss in the cache tags must also query the region buffer. These look-ups can be done in parallel, at the cost of some power, or in series, at the cost of some extra latency on cache misses. If the request is a hit in the region buffer and the necessary permissions exist, the request is issued on the direct-access bus as a coherent direct access request. On misses in the region buffer, the request is forwarded to the region directory, which satisfies the request and responds to the region buffer with region permissions depending on the request type.

The region buffer is placed between the cache tags and the region directory to filter probe requests. All probes that the region

| | |
|---|---|
| CPU Clock | 2 GHz |
| CPU Cores | 2 |
| CPU L1 Data Cache | 64 KB (2-way banked) |
| CPU L1 Instruction Cache | 64 KB (2-way banked) |
| CPU Shared L2 Cache | 2 MB (16-way banked) |
| GPU Clock | 1 GHz |
| Compute Units | 32 |
| Compute-unit SIMD Width | 64 scalar units by 4 SIMDs |
| GPU L1 Data Cache | 32 KB (16-way banked) |
| GPU L1 Instruction Cache | 32 KB (8-way banked) |
| GPU Shared L2 Cache | 4 MB (64-way banked) |
| L3 Memory-side Cache | 16 MB (16-way banked) |
| DRAM | DDR3, 16 channels, 667 MHz |
| Peak Memory Bandwidth | 700 GB/s |
| Baseline Directory | 262,144 entries (8-way banked) |
| Region Directory | 32,768 entries (8-way banked) |
| MSHRs for All Directories | 32 entries |
| Region Buffer | 16,384 entries (64-way banked) |

**Table 1: Simulation Parameters**

directory sends to the region buffer are for regions, not block-level data requests. This allows a single request to be sent on the coherence network, and any block-level probes, which need to be forwarded to the local L2 cache, are handled without the need for interconnect traffic.

### 4.4 Region Directory

Also added in HSC is the region directory (Figure 4(c)), which replaces the block-level directory in the baseline system. This structure is organized very similarly to the baseline block-level directory, except that the region directory is indexed by region tags instead of block tags. Similar to the baseline directory, each region directory entry contains a bit-vector of sharers (one bit for each cluster) and the state associated with each region.

Because the region directory is no longer required to support high bandwidth, all of the associated structures (the directory MSHR table and the PRR) can be simplified greatly. The region directory does not need to support many outstanding region permission requests, so these structures can be quite small, reducing the area and power of the directory.

### 4.5 Hardware Complexity

The HSC design adds two region buffers and one region directory to the baseline system. HSC extends the CPU and GPU L2 caches to make all L2 caches interact with the region buffers. Our evaluation results show that HSC reduces the resource requirements (MSHRs) to a reasonable number. Thus, the hardware complexity of HSC mainly lies in the storage overheads of region buffers and region directory and the extension of L2 caches.

The extension of the L2 cache includes modifying related cache transactions to perform the memory request flow with HSC. For example, for L2 miss, instead of being forwarded to directory, the data request is forwarded to the region buffer to check permission. An interface to the direct-access bus is also added to the shared L2 cache to route traffic directly to memory. HSC does not require any changes to the coherence states. Thus, no additional storage is needed for L2 caches.

Based on the detailed description in Figure 3, each region buffer entry requires 36 bits of storage and each region directory entry takes 22 bits, for a 1 KB region. In our configuration, we used 32,768 region directory entries and 16,384 region buffer entries for each region buffer with region size of 1 KB. The storage overhead for each region buffer is 74 KB, which is less than 5% for a 2 MB CPU L2 cache. In total, the storage overhead is 238 KB, which is less than 1% of the total on-chip memory hierarchy. In comparison, the baseline directory includes 262,144 directory entries and occupies 820 KB storage, significantly more than the region directory.

## 5. EXPERIMENTAL SET-UP

In this section, we describe the simulation infrastructure and the workloads we used to evaluate our HSC design.

### 5.1 Simulation

For simulating the CPU, we used the in-order non-pipelined CPU model from the gem5 [9] simulation infrastructure. For simulating the GPU, we used a proprietary simulator based on the AMD Graphics Core Next architecture [20]. The heterogeneous system is simulated by combining the memory systems of gem5 and our proprietary GPU simulator. The simulator runs in timing mode and produces detailed statistics including simulator cycles, directory, and cache traffic.

The simulated CPU-GPU processor has two CPU cores and 32 GPU CUs. As described in Section 3.1, future heterogeneous processors likely will scale to provide increasing computing power. We use a large number GPU CUs and extreme memory bandwidth of 700 GB/s to reflect the trend. Table 1 shows the parameters used in the simulations. In each protocol, the simulated memory system has a shared memory-side L3 cache. The CPU memory system uses MOESI states for cache blocks; the GPU caches are write-through and use a VI (valid/invalid)-based protocol for coherence.

We implemented the HSC protocol on this simulated architecture. Regions can be in one of three stable states: private (read-write permissions), shared (read-only permissions), and invalid. For comparison purposes, we evaluated three coherence protocols:

- Broadcast: Broadcast-based null directory protocol
- Baseline: Block-based directory protocol (Figure 1)
- HSC: Region-based directory protocol (Figure 4) with a region size of 1 KB or 16 64-byte blocks

### 5.2 Workloads

We make use of two benchmark suites to evaluate HSC design. Seven of the 14 Rodinia [12] benchmarks were ported to our simulator. These include back propagation (bp), a machine-learning algorithm; breadth-first search (bfs), HotSpot (hs), a thermal simulation for processor temperatures; LU Decomposition (lud); Needleman-Wunsch (nw), a global optimization method for DNA sequence alignment; kmeans (km), a clustering algorithm used in data mining; and, speckle-reducing anisotropic diffusion (srad), a diffusion algorithm. We also ported four AMD APP SDK [5] benchmarks to our simulator: bitonic sort (bn), discrete cosine transform (dct), histogram (hg), and matrix multiplication (mm). The other Rodinia and AMD SDK benchmarks have not been ported to the shared address space APU architecture.

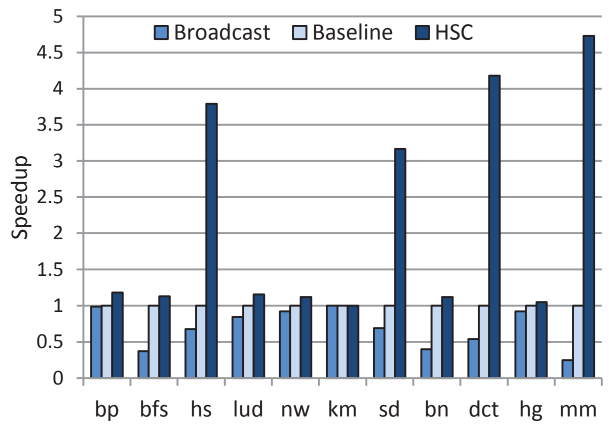

**Figure 5: Execution time of all protocols normalized to baseline directory protocol.**

All benchmarks are written in OpenCL. The workloads were modified to utilize our fully coherent system by removing all explicit memory copies and using pointers instead. All benchmarks kept the same general structure after the modifications. All of the workloads use all CUs on the simulated GPU but exercise only one CPU while executing.

## 6. EXPERIMENTAL RESULTS

In this section, we present the experimental results of HSC by running the simulation in detailed mode. To focus on our region of interest, the results presented in this section do not contain statistics of CPU operations such as initialization, reading files, and other similar operations that do not involve CPU-GPU interaction. The results include the reduction in directory bandwidth, application execution time, and MSHR consumption compared to a block-based directory protocol.

### 6.1 Execution Time

Figure 5 shows the performance of the coherence protocols across the execution of the entire region of interest (both CPU and GPU execution). HSC significantly increases performance for many of the applications. The average speedup is 2x, with a maximum speedup of more than 4.5x.

The key reason HSC outperforms the baseline directory protocol is that it alleviates the queuing delays caused by too few MSHR entries. As discussed in Section 3.2, constraining the MSHRs at the directory causes a significant slowdown for some applications. HSC reduces the pressure on MSHR entries and the applications can obtain high performance with modest hardware costs (32 MSHR entries).

Most applications see at least a slight slowdown when using the broadcast protocol. This is because issuing the probes to all L2 caches for every directory access causes pressures on the directory and L2 cache structures. The number of needed MSHRs increases as latency for requests increases because every request must wait for all L2 caches to respond. Additionally, probe requests to the L2 caches interfere with demand traffic. Also, for some applications the PRR entries become a bottleneck, introducing more stalls.

Although some applications see a significant performance improvement with HSC, some applications do not. This is because these applications are CPU-bound and do not spend a significant amount of time running on the GPU. Figure 6 shows the percent of total execution time each application spends running on the GPU and CPU. Applications that do not experience performance improvements with HSC are those that have a very low percent of total execution time on the GPU, and thus have relatively low directory bandwidth demand. As GPUs become easier to program, future GPGPU workloads likely will spend a significant percentage of their run-time on the GPU; for these workloads, HSC improves performance.

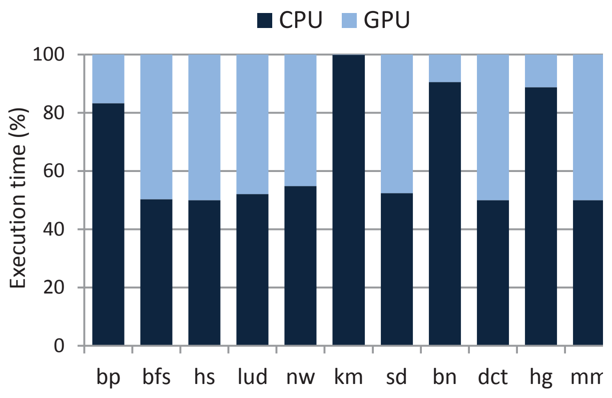

**Figure 6: Percent of application execution time spent on GPU and CPU.**

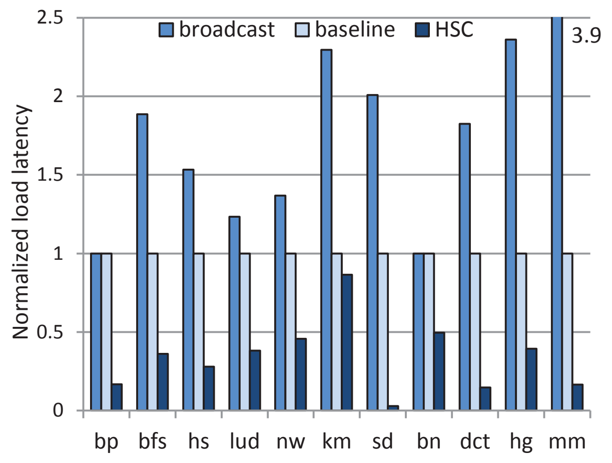

**Figure 7: Latency of loads normalized to the baseline directory protocol.**

Most of HSC's performance improvement comes from its ability to reduce the latency of loads issued from the GPU. The average load latency is reduced because most requests proceed directly to memory via the direct-access bus, bypassing the directory. Figure 7 shows the decrease in load latencies with HSC. The write-through memory system of the GPU caches allows instantaneous stores; hence, store latency is not critical to the performance of the simulated system.

Although the bfs benchmark has significant time on GPU, it does not get significant benefit from HSC (only 9%) due to its short kernel run-times. bfs launches many very short kernels to the GPU and the overheads of kernel launching and completion hide the benefits of HSC.

nw also shows little benefit from HSC (10%) even though the GPU time is significant. nw suffers because of low region density. Most

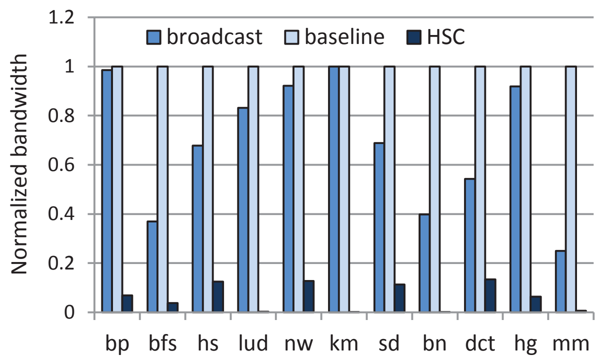

**Figure 9: Directory bandwidth normalized to baseline directory protocol**

benchmarks have an average region density—the number of valid blocks currently in the local cache—of nearly 75% (12 of 16 blocks in the region are valid). However, nw shows much lower densities, with an average of slightly more than one block valid in the region. nw has poor region density due to its pattern of memory accesses.

Region density is a good indicator of the effectiveness of HSC. We expect that most GPU applications are not like nw, and exhibit high spatial locality because low spatial locality leads to memory divergence and poor memory bandwidth utilization on GPU architectures.

### 6.2 Directory Bandwidth

Reducing the directory bandwidth eases implementation of coherence for a heterogeneous system. Figure 9 shows that HSC accomplishes this goal with an average of 94% reduction in the directory bandwidth compared to the baseline directory protocol.

For some benchmarks, the reduction in directory accesses is more than 99%. The maximum reduction in directory accesses expected is only 94% directory requests elided; however, many benchmarks see a greater reduction. This large reduction come from the region buffers holding permissions for regions that currently are not valid in the cache. Because of the compactness of the region buffers, it is easy to overprovision them such that the region buffer has a much larger reach than the attached caches. Because of this larger reach, the region buffer caches permissions for some regions in which the cache does not contain any valid blocks, and these permissions are re-used when the request for that block misses in the cache.

### 6.3 Resource Requirements

Figure 8 shows the reduction in directory resources achieved by HSC. This data was generated by allowing unlimited MSHRs in the HSC protocol. In the baseline protocol, to not cause stalls, the directory MSHR table needs, on average, at least 10,000 entries. In the worst case, to cause no stalls in hotspot, the baseline directory would need in excess of 36,000 entries. HSC greatly reduces this constraint. On average, the reduction is more than 95%. The average maximum needed MSHR table in HSC is only 488 entries, and the maximum across all benchmarks is 1,888 entries. Although the maximum used MSHRs are still much larger than a reasonable amount of MSHRs (32), performance is improved by reducing queuing delays at the directory. With this huge reduction of resource requirements at the directory, HSC makes coherent caches between the CPU and GPU feasible without significant hardware resources.

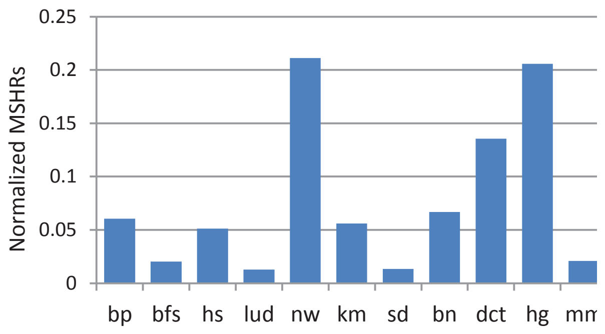

**Figure 8: Maximum number of MSHRs used by HSC normalized to baseline**

## 7. RELATED WORK

### 7.1 GPU Coherence

Historically, coherence between the CPU and GPU in a system has been managed by software, which is largely an artifact of the memory partitioning in these systems. However, now that the CPU and GPU are becoming more closely integrated, there have been some proposals for providing the programmer with a more coherent view of the shared memory. PTask [26] proposes a task graph library that removes the need for programmers to manage the memory explicitly. The underlying coherence mechanism is implemented in a software library. Cohesion [16] implements a combination of software and hardware coherence in which the data can migrate between these two coherence domains dynamically. This system works well for current GPGPU applications that use the bulk-synchronous communication pattern. A potential bottleneck in the Cohesion design is that on every block request that misses in the higher-level caches, the directory must be queried to determine if the block is in the software- or hardware-managed coherence domain. Asymmetric distributed shared memory (ADSM) implements a logically shared address space between the CPU and GPU [14]. Gelado et al. provide GMAC, a software implementation of ADSM. These proposals come with potential programming model complications or performance issues. Due to this, and because AMD and other HSA Foundation members have committed to providing hardware coherence, we focus on only hardware coherence in this paper. Additionally, Singh et al. developed a coherence protocol for intra-GPU coherence [28]. Their work uses a timestamp-based protocol called temporal coherence. HSC focuses on coherence between an integrated CPU and GPU. Our results are compatible with other forms of intra-GPU coherence.

### 7.2 Reducing Broadcast Bandwidth

Many systems have been developed to reduce the bandwidth required in snooping systems. Although we focus on a directory-based coherence protocol, these ideas were influential. Moshovos et al. proposed JETTY [22], which filters incoming snoops to the cache based on contiguous regions. Stream registers [27] implement similar structures to JETTY except the regions can be sized dynamically. Also, high bandwidth for snoop requests is supported by duplicating the logic for the snoop filters at every port into the cache. RegionTracker [31] replaces the local cache tags with region-based tags. RegionScout [23], a precursor to some of the region coherence work, adds a non-shared region table (NSRT), which caches regions known not to be held in any other caches, and the cached region hash (CRH), which is a Bloom filter that holds a superset of regions cached in the local cache. These structures filter the broadcasts required. Other solutions have been proposed as well, including virtual tree coherence [15], subspace snooping [17], and in-network coherence filtering [2]. These works relate to the design of HSC.

### 7.3 Reducing Directory Look-ups

TurboTag [18] focuses on reducing energy consumed by the directory by using Bloom filters to reduce the number of directory look-ups. Each directory bank has a counting Bloom filter that is accessed prior to the directory look-up. If the entry is not present in the Bloom filter, then it cannot be present in the directory and the look-up is elided. This work could help reduce the high bandwidth from the GPU to the directory tags, but it does not limit the number of MSHRs required.

### 7.4 Reducing Directory Resources

Many scalable directory protocols have been studied. Spatiotemporal Coherence Tracking (SCT) [3] aims to reduce the size of the directory by leveraging spatially local regions. SCT is dual-grained, enabling the tracking of permissions at both the region and block levels. SCT reduces the required number of directory entries by tracking private data at the region level.

Basu et al. extend region coherence to directory-based systems [8]. This work proposes a dual-granular directory design that tracks both block- and region-level permissions. HSC is a different implementation for directory-based region-level tracking; however, HSC focuses on heterogeneous CPU-GPU systems instead of CMPs and primarily aims to reduce the bandwidth to the directory, not the directory size.

### 7.5 Reducing Miss-handling Resources

Scalable miss handling [29] proposes a hierarchical miss-handling architecture. The authors observe that when constructing a high-bandwidth miss-handling architecture, banking the miss-handling registers does not provide enough benefits due to bank access imbalance. Therefore, the authors construct a high-bandwidth miss-handling architecture by adding per-bank miss-handling registers and allowing them to overflow into a large centralized set of miss-handling registers. To reduce the bandwidth to the centralized file of miss-handling registers, the authors add a per-bank Bloom filter that holds all of the entries in the centralized file. On a miss in the per-bank miss-handling registers, if the address is not in the Bloom filter then the centralized file does not need to be accessed.

HSC solves a similar problem by limiting the total number of MSHRs required. However, HSC also reduces the total bandwidth to both the directory and its MSHR structure by moving most requests onto the direct-access bus, bypassing the directory completely.

## 8. CONCLUSIONS

Heterogeneous CPU-GPU processors potentially can be more programmable and more efficient with the support of hardware coherence. However, limited directory resources will be a significant bottleneck due to a GPU's high memory bandwidth requirements and unique patterns of memory accesses. We introduce Heterogeneous System Coherence, or HSC, which implements directory coherence at a region granularity. Given the high spatial locality of GPU data, obtaining coherence permissions at coarse granularity (compared to traditional block-level) enables the elision of the majority of directory accesses.

Evaluation results show that HSC achieves an average performance improvement of 2x compared to a baseline directory design. Bandwidth to the directory is reduced by an average of 95% and more than 99% for four of the benchmarks. Thus, HSC provides a practical hardware solution to full coherence in a heterogeneous CPU-GPU system.

## 9. ACKNOWLEDGEMENTS

This work was performed at AMD, including student internships. Wisconsin authors improved the paper's presentation while being partially supported with NSF grants CCF-0916725, SHF-1017650, CNS-1117280, and CCF-1218323. The views expressed herein are not necessarily those of the NSF. Professors Hill and Wood have significant financial interests in AMD.

## 10. REFERENCES

[1] 3D-ICs: *http://www.jedec.org/category/technology-focus-area/3d-ics-0*. Accessed: 2013-09-18.

[2] Agarwal, N. et al. 2009. In-network coherence filtering. (2009), 232.

[3] Alisafaee, M. 2012. Spatiotemporal Coherence Tracking. (Dec. 2012), 341–350.

[4] AMD Inc. 2010. *AMD Fusion$^{TM}$ Family of APUs*.

[5] AMD Inc. 2008. *AMD Stream SDK*.

[6] AMD Radeon$^{TM}$ HD 7970 Graphics: *www.amd.com/radeonHD7970*.

[7] Archibald, J. and Baer, J.-L. 1984. An Economical Solution to the Cache Coherence Problem. *Proceedings of the 11th Annual International Symposium on Computer Architecture* (Jun. 1984), 355–362.

[8] Basu, A. et al. 2013. *CMP Directory Coherence: One Granularity Does Not Fit All*. Technical Report #CS-TR-2013-1798. Univ. of Wisconsin Computer Sciences.

[9] Binkert, N. et al. 2011. The gem5 simulator. *Computer Architecture News (CAN)*. (2011).

[10] Black, B. et al. 2006. Die Stacking (3D) Microarchitecture. (Dec. 2006), 469–479.

[11] Cantin, J.F. et al. 2005. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. *Proceedings of the 32nd Annual International Symposium on Computer Architecture* (Jun. 2005).

[12] Che, S. et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. *2009 IEEE International Symposium on Workload Characterization (IISWC)* (2009).

[13] GeForce GTX 780: 2013. *http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications*.

[14] Gelado, I. et al. 2010. An asymmetric distributed shared memory model for heterogeneous parallel systems. (2010), 347.

[15] Jerger, N.D.E. et al. 2008. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. *MICRO 41* (Nov. 2008), 35–46.

[16] Kelm, J.H. et al. 2010. Cohesion: A Hybrid Memory Model for Accelerators. *Proccedings of the 37th Annual Internal Symposium on Computer Architecture (ISCA)* (Jun. 2010).

[17] Kim, D. et al. 2010. Subspace snooping. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), 111–122.

[18] Lotfi-Kamran, P. et al. 2010. TurboTag: lookup filtering to reduce coherence directory power. *Proceedings of the 16th*

*ACM/IEEE international symposium on Low power electronics and design* (2010), 377–382.

[19] Mantor, M. 2011. *Fusion and the Future of Heterogeneous Computing*.

[20] Mantor, M. and Houston, M. 2011. *AMD Graphics Core Next*.

[21] Martin, M.M.K. et al. 2012. Why on-chip cache coherence is here to stay. *Communications of the ACM*. 55, 7 (Jul. 2012), 78.

[22] Moshovos, A. et al. 2001. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture* (Jan. 2001).

[23] Moshovos, A. 2005. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. *Proceedings of the 32nd Annual International Symposium on Computer Architecture* (Jun. 2005).

[24] Owens, J.D. et al. 2008. GPU Computing. *Proceedings of the IEEE*. 96, 5 (2008).

[25] Rogers, P. 2013. Heterogeneous System Architecture Overview. *HOT CHIPS* (2013).

[26] Rossbach, C.J. et al. 2011. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. *Proc. of the 23nd ACM Symp. on Operating System Principles* (Oct. 2011).

[27] Salapura, V. et al. 2007. Improving the accuracy of snoop filtering using stream registers. *Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture* (2007), 25–32.

[28] Singh, I. et al. 2013. Cache Coherence for GPU Architectures. *The 19th IEEE International Symposium on High Performance Computer Architecture*. (2013).

[29] Tuck, J. et al. 2006. Scalable Cache Miss Handling for High Memory-Level Parallelism. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2006).

[30] Wilkins, M. *NVIDIA Jumps on Graphics-Enabled Microprocessor Bandwagon*.

[31] Zebchuk, J. et al. 2007. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2007), 314–327.