Software Assisted Hardware Cache Coherence for Heterogeneous Processors

Arkaprava Basu Sooraj Puthoor Shuai Che Bradford M. Beckmann

AMD Research
Advanced Micro Devices, Inc.
{arkaprava.basu, sooraj.puthoor, brad.beckmann, shuai.che}@amd.com

ABSTRACT

Current trends suggest that future computing platforms will be increasingly heterogeneous. While these heterogeneous processors physically integrate disparate computing elements like CPUs and GPUs on a single chip, their programmability critically depends upon the ability to efficiently support cache coherence and shared virtual memory across tightly-integrated CPUs and GPUs. However, throughput-oriented GPUs easily overwhelm existing hardware coherence mechanisms that long kept the cache hierarchies in multi-core CPUs coherent.

This paper proposes a novel solution called Software Assisted Hardware Coherence (SAHC) to scale cache coherence to future heterogeneous processors. We observe that the system software (Operating system and runtime) often has semantic knowledge about sharing patterns of data across the CPU and the GPU. This high-level knowledge can be utilized to effectively provide cache coherence across throughput-oriented GPUs and latency-sensitive CPUs in a heterogeneous processor. SAHC thus proposes a hybrid software-hardware mechanism that judiciously uses hardware coherence only when needed while using software's knowledge to filter out most of the unnecessary coherence traffic. Our evaluation suggests that SAHC can often eliminate up to 98-100% of the hardware coherence lookups, resulting up to 49% reduction in runtime.

CCS Concepts

• CCS → Computer systems organization → Architectures → Parallel architectures → Single instruction, multiple data

Keywords

Cache coherence; Heterogeneous processor; GPGPU; Virtual memory; Operating system.

1. INTRODUCTION

Many current and future processors are becoming increasingly heterogeneous. Large commercial processor manufacturers like AMD®, Intel®, and Qualcomm® ship millions of processors with CPUs and GPUs tightly coupled together. This trend is likely to continue in the near-future with highly-capable accelerator-like GPU being tightly integrated inside a processor.

Ease of programming these heterogeneous processors is key to harness their full potential. Cache coherence and shared virtual memory are two critical features that greatly enhance programmability of any system. It is thus no surprise that several major hardware vendors like AMD®, ARM®, and Qualcomm® promise shared virtual memory and cache coherence in their heterogeneous processors [22].

However, realizing the promise of full cache coherence across the CPU and GPU in a heterogeneous processor is challenging. Traditional approaches have either burdened the application programmer to explicitly manage coherence (software cache coherence) or put the onus of maintaining coherence entirely on the hardware (hardware cache coherence). Software cache coherence is more appealing for niche accelerators programmed by ninja programmers while the hardware cache coherence is the norm for more generic and easily programmable CPUs. Unfortunately, neither of these two approaches is readily extensible to heterogeneous processors that should be programmable en masse. Software cache coherence seriously impedes programmability. The hardware-only coherence is hard to scale to meet the demand from throughput-oriented GPUs [15]

Researchers have proposed several designs to address this challenge in the past. For example, GMAC [7] proposes a software only solution that enables coherence between a CPU and GPU with a software library, but adds significant performance overhead. Cohesion [10] proposes a software-hardware co-design that allows data to be kept coherent either by hardware or by software. However, to achieve high performance, Cohesion requires application programmers to manage coherence and adds significant hardware overheads (e.g., adds region table). HSC [15] proposes a hardware-only solution that revamps the hardware cache coherence mechanism by proposing to maintain cache coherence at a larger granularity of region (e.g., 1KB) instead of traditional cache blocks (e.g., 64 bytes). Given that current cache coherence protocols are already hard to verify, the significant changes proposed by HSC will be challenging to adopt.

Ideally, heterogeneous processors would retain the ease of programming enabled by hardware cache coherence but without adding significant performance overhead and without requiring a re-design of conventional hardware. To this end, we propose software-assisted hardware-managed coherence (SAHC). The key observation is that the system software (Operating system, runtime) knows how data is shared across the CPU and the GPU. This knowledge can aid the hardware cache coherence. Furthermore, since memory is allocated by operating system at the granularity of

MEMSYS '16, October 03-06, 2016, Washington, DC, USA

pages we observed that this semantic knowledge is captured well at the page-granularity (e.g., 4KB). Thus, SAHC piggybacks on virtual memory's page-permission checks to enforce coherence between the CPU and the GPU cache hierarchies. SAHC then dynamically falls back to the traditional block-granular hardware cache coherence only for frequently-shared data across CPU and GPU. Specifically, SAHC extends the set of page permissions (e.g., read/write/no-execute) to encode page-granular coherence permissions. These page-grain coherence permissions encode whether a given page is currently accessible by CPU or by GPU or by both. The existing virtual memory hardware for enforcing page permissions is then minimally extended to enforce newly-added page-grain coherence permissions. However, if a page contains data that is frequently-shared across CPU and GPU, then it will trigger an excessive number of costly page-permission changes. To avoid any potential performance degradation under such scenario, SAHC dynamically falls back to conventional block-granular hardware cache coherence only for pages that are frequently shared across the CPU and the GPU. Our analysis (Section 4) shows that only small fraction of data is frequently-shared across CPU and GPU (e.g., synchronization variable)

SAHC's software-hardware co-design enables several important benefits over the state-of-art. First, SAHC enables traditional block-granular hardware cache coherence to scale to GPU's memory bandwidth requirements by judiciously using it only when necessary. Thus, unlike hardware-only coherence for heterogeneous processors SAHC does not require re-designing and re-verifying hardware cache coherence mechanism. Second, SAHC does not burden application writers to maintain cache coherence unlike software cache-coherence mechanisms [10]. Instead, SAHC minimally extends system software (OS and runtime) to manage coherence. Finally, our evaluation shows that SAHC can eliminate 98-100% of all hardware coherence directory lookup and can reduce application runtime by up to 50%

Our contributions are as follows:

1. We analyze and quantify bottlenecks in the hardware cache coherence in a heterogeneous system.
2. We analyze and characterize cache coherence needs across the CPU and GPU for several applications. This analysis further leads to classification of CPU-GPU interactions in heterogeneous applications.
3. Guided by this analysis, we propose a novel software hardware co-designed coherence scheme (SAHC) for emerging heterogeneous systems.
4. Finally, we evaluate and analyze the efficacy of proposed SAHC scheme in heterogeneous processor.

2. Background

In this section, we describe our baseline heterogeneous system that is broadly modeled after commercial heterogeneous systems. We then discuss basics of virtual memory.

2.1 Baseline Heterogeneous Processor

Figure 1 shows an overview of our baseline system. There are two clusters -- a CPU cluster (left) and a GPU cluster (right). A CPU cluster can contain any number of CPU cores (two in our experiments) and private L1 caches (per core). A GPU cluster is made up of several compute units (CUs), each comprising of several SIMD execution units. Each SIMD unit is 64-lanes wide and executes instructions in lock-step fashion. Each CU has with private L1 caches.

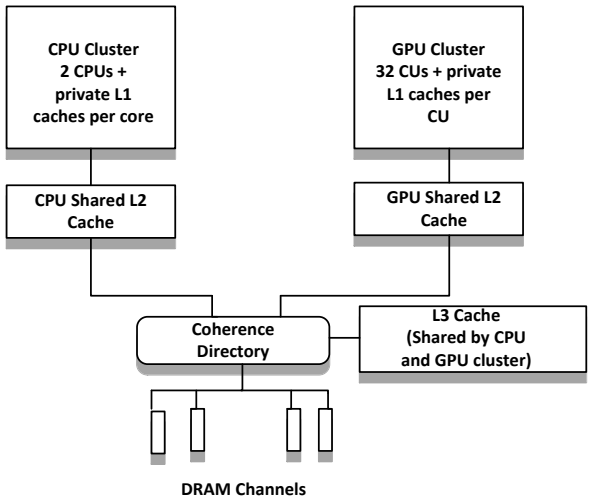


Figure 1. Baseline system architecture.

Both the CPU and GPU clusters have their private L2 caches. A block-granular inclusive coherence directory maintains the coherence between the CPU and GPU L2 caches and each directory entry has a two-bit sharing vector (one bit for each cluster). A request to directory allocates an entry in Miss Handling Registers (MSHR) to keep track of outstanding requests. An entry in MSHR is deallocated only when the directory request completes.

Memory accesses from CPU or GPU that miss in their respective local cache hierarchy look up the coherence directory. A lookup in the coherence directory identifies whether the remote cache hierarchy may contain the requested cache block. The local cache hierarchy of GPU consists of per-CU L1 caches that shares a common L2 cache. The CPU local cache hierarchy consists of a per-CPU L1 and a shared L2. GPU's local cache hierarchy is called the remote cache hierarchy for any request originated from the CPU and vice-versa. If a request misses in the coherence directory then it looks up the unified L3 cache before going off chip to the DRAM.

2.2 Virtual Memory Basics

The software generates a memory access with a virtual address but the hardware uses physical address to lookup requested data or instruction. The virtual memory subsystem translates the virtual addresses to the corresponding physical addresses. Address translation is performed at the granularity of a page (e.g., 4KB). Each page is also assigned a set of permissions (e.g., read-only, read-write) that controls accesses to the page. A virtual-to-physical address mapping and the associated access permissions are assigned by the OS during memory allocation or modified later on an explicit request (e.g., on *mprotect*() system call in Linux). The hardware is responsible for performing the address translation and to enforce page permissions on every memory access. The hardware caches recently-used address translation entries in a hardware structure called a Translation Lookaside Buffer (TLB) to fasten the translation. The hardware raises a page-fault exception to the OS if the address of an access does not have a valid translation or if it lacks requisite access permissions. On a page fault, the OS can then either create the necessary translation entry or raise an error (e.g., segmentation fault) to the application. Unfortunately, the process of servicing such page faults is slow as it involves generating an interrupt to the OS, switching from user to supervisor mode, executing the page fault routine, and (potentially)

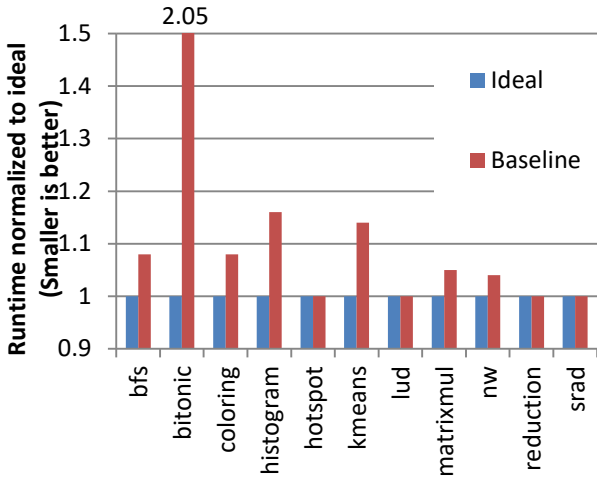


Figure 2. Performance overhead of coherence in heterogeneous systems (Baseline).

performing a TLB shootdown before finally resuming the execution of the application.

3. Motivation and Goal

In this section, we analyze the overheads of cache coherence in the baseline heterogeneous processor Figure 1).

Overheads of enforcing coherence: To understand the overhead of enforcing cache coherence, we first simulate an *ideal* cache coherence mechanism. An ideal system maintains cache coherence across CPU and GPU caches, like in the baseline system using the block-granular hardware directory, but *without any latency of doing so*. The ideal cache coherence mechanism can fetch coherence permissions, invalidate blocks in other caches, or send coherence acknowledgements without incurring any delay. Figure 2 shows performance of the baseline system normalized to that of a system implementing ideal cache coherence mechanism for applications described in Section 6.1 (bars for ideal is always 1). We observe that seven out of the eleven workloads we studied suffer from non-negligible performance overheads due to cache coherence in the baseline architecture, compared to the ideal coherence. We expect future GPUs with larger number of CUs will suffer larger performance degradation due to increased coherence activity across CPU's and GPU's caches. Although not quantified here, coherence lookups and coherence messages adds to energy budget, too.

Our analysis attributes the performance loss primarily to queuing delay at the coherence directory. This is illustrated in Figure 2. We found that the delay at the coherence directory is caused by contention for MSHRs at the caches, for banks at the directory, and due to the latency of servicing coherence probes. In summary, the traditional coherence directory schemes that worked well for CPU designs is not extensible to a heterogeneous system with throughput-oriented accelerators like GPUs.

Goals: SAHC aims to scale the traditional block-grain hardware cache coherence to heterogeneous processors without re-designing the coherence mechanisms and, without sacrificing ease of programming.

More specifically, following are the goals of this work:

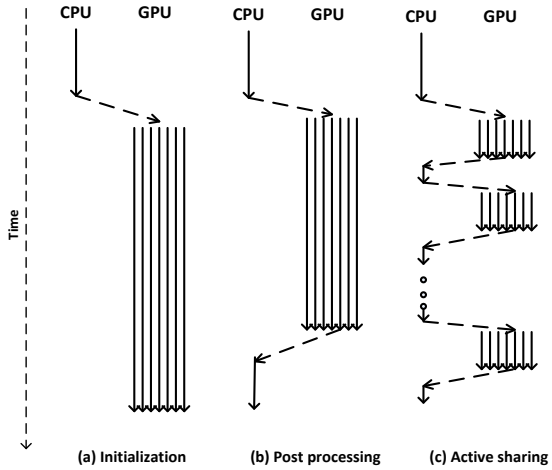


Figure 3. Interaction patterns between CPU and GPU.

1. Scale hardware cache coherence mechanism to the needs of a heterogeneous processor by reducing lookups at the coherence directory.
2. Minimize modifications to the conventional hardware cache coherence mechanisms to avoid validation and re-design cost.
3. Avoid application level cache coherence management to enhance ease of programming.

4. Application Analysis and Observations

In this section, we first analyze the characteristics of heterogeneous applications and examine their communication patterns across the CPU and the GPU. This helps us understand how heterogeneous application may use the coherence mechanism.

We choose heterogeneous applications from the Rodinia [5] and the AMD APP SDK [20] benchmark suites (see Section 6.1) for this study. We characterize when a data *toggles* from the CPU to the GPU and vice-versa. We define a *toggle* to have occurred when a given piece of data that is most recently accessed by a CPU is next accessed by the GPU and vice-versa. A toggle suggests the need for latest data to be communicated across the CPU and the GPU. We characterize the amount and timing of these toggles for different workloads both qualitatively and quantitatively.

Qualitative Analysis: First, we analyze several heterogeneous applications to understand when and why a data need to be communicated across the CPU and the GPU due to the algorithmic needs.

We find three primary communication patterns across the CPU and the GPU in a heterogeneous application (Figure 3). The most common patterns occur when the CPU initializes a piece of data before the GPU works on that data (Figure 3 (a)). Data is initialized either explicitly by the application or implicitly (zeroed) by the OS. The second most common communication pattern (Figure 3 (b)) occurs when CPU processes the result regenerated by the GPU. We call this pattern *post processing.* In the third commonly occurring pattern, applications *actively share* data (Figure 3 (c)) across the CPU and the GPU during program execution. We observe that such active sharing occurs when either the GPU iteratively performs computations on a piece of data while the CPU subsequently performs reduction for each iteration or when there are synchronization across the CPU and the GPU.

Table 1. Frequency and classification of toggles of pages (4KB) between CPU and GPU.

Application	Number of domain toggles per 10K coherence directory lookups	Toggle pattern break down (%)		
		Initialization	**Post Processing**	**Active Sharing**
bfs	67.4	11.10	11.11	77.79
bitonic	0.56	100	0	0
coloring	12.06	99.63	0.06	0.32
histogram	73.93	99.22	0.7	0.16
hotspot	15.06	99.7	0.03	0.0
kmeans	25.8	47.64	0.78	52.27
lud	3.5	99.91	0.09	0.0
matrixmul	5.9	85.6	14.40	0.0
nw	77.59	75	25.0	0
reduction	77.57	99.79	0.21	0.0
srad	83.69	85.53	14.47	0.0

Quantitative Analysis: Next, we quantitatively analyze these aforementioned different communication patterns. We perform measurements at the boundary of pages (4KB in our study) since memory buffers are allocated by the OS at the granularity of pages. Specifically, we seek to answer following two questions: (1) how frequently a page toggles between the CPU and the GPU, and (2) which of the three communication patterns lead to such behavior? The answer to the first question helps estimate how frequently coherence actions are needed due to application's communication needs across the CPU and the GPU. The answer to the second question helps determine which communication patterns to optimize.

We measure the frequency of toggles relative to coherence directory lookups in a baseline system with traditional block-grain hardware cache coherence. By normalizing it to the number of coherence directory lookups, this analysis provides an estimate of relative frequency of interactions between the CPU and the GPU compared to extraneous directory lookups performed in the baseline. The first data column of Table 1 lists the number of toggles between the CPU and GPU for every 10,000 coherence directory lookups in the baseline. The next three columns in the table classify these toggles based on the patterns shown in Figure 3.

The data in Table 1 can be summarized into two important observations:

1. The need to communicate data across the CPU and the GPU (toggle) is significantly less than the coherence directory lookup performed in the baseline. For example, we observe none of the applications requires more than 83 toggles per 10,000 directory lookups. This, in turn, suggests that the majority of the coherence directory lookup are extraneous and can be eliminated if semantic knowledge about communication pattern is utilized.
2. The most common cause for the communications between the CPU and the GPU is initialization of data by the CPU before the GPU starts processing it. The second most common reason is post-processing of data at the CPU after computation on GPU is complete. Two out of eleven applications (bfs and kmeans) show significant active sharing across CPU and GPU.

5. Design and Implementation

In this section, we first describe our design principle for scaling hardware coherence to the needs of heterogeneous processors. We then detail our design and implementation of proposed software-assisted-hardware-coherence (SAHC). Finally, we discuss multiple optional optimizations that can further improve the design in the presence of specific CPU-GPU communication patterns found in a few applications.

5.1 Design Principles

The primary design principle for SAHC is to make use of virtual memory's mechanism for page permission checking to enforce coherence permission whenever possible. SAHC dynamically falls back to the traditional block-grain hardware coherence only if necessary for better performance (e.g., for actively shared synchronization variable).

Our design principle is guided by the observations made in Section 4 and also by how page permissions are enforced in today's hardware. In particular, we observed that communications between the CPU and the GPU are generally infrequent. However, when a GPU (CPU) will accesses a data that is most recently touched by the CPU (GPU), is unknown to both the hardware and the system software. Thus, every memory access needs to be checked to determine if the latest version of the data it requests may be in the remote cache hierarchy (e.g., in GPU caches for a CPU access). However, our analysis in the previous section shows that such occasions are relatively infrequent.

We further note that page access permissions (e.g., read-write, read-only) are checked and enforced by hardware on every memory access. Hardware detects possible violations of assigned page permissions while the OS acts on such violation. Any modifications to the assigned page permissions had to be performed by the OS. Thus, page permission checks are fast while page permission modifications are slow. Such division of responsibilities between hardware and software worked well for virtual memory since the page permission needs to be enforced on ever memory access, but page permission violations and modifications are infrequent.

The above observations suggest that the current virtual memory subsystem design aligns well with the coherence needs across the CPU and the GPU in a heterogeneous processor. Thus, we propose

Page Attribute \ Access Type	CPU Access (Read/Write)	GPU Access (Read/Write)
CPU_ONLY	Access can proceed. *No* h/w directory lookup needed.	Raise permission fault to OS.
GPU_ONLY	Raise permission fault to OS.	Access can proceed. *No* h/w directory lookup needed.
CPU_GPU	Access can proceed. But *may* require h/w directory lookup on local cache miss.	

Figure 4 Page-grain Coherence Permission Access Matrix.

to minimally extend existing mechanisms for a page access permission check to also include coherence permission checks. SAHC will be able to fall back to traditional block grain cache coherence for memory regions that frequently toggles between CPU and GPU to avoid overheads of modifying permission.

5.2 Software-Assisted Hardware Coherence (SAHC)

The SAHC has three major components. First, the SAHC introduces new page-grain cache coherence permissions in the page table entry (PTE) and the hardware is responsible to enforce these new permissions along with already-existing page-grain access permissions. Second, the SAHC introduces the policy and the mechanism for assignment of the newly introduced page permissions. This part is implemented in the OS's page fault handler. Third, the SAHC exploits runtime's knowledge about work scheduled on GPU to reduce the frequency of modification to page-grain coherence permissions. The following sections describe each of these major components in detail

5.2.1 New Page-grain Coherence Permissions

The SAHC extends page permissions to include the notion of cache-ability of blocks belonging to a page by the CPU and/or by the GPU in their respective local cache hierarchies. This requires change in the PTE format to include two additional bits and extension to the page fault mechanism as follows.

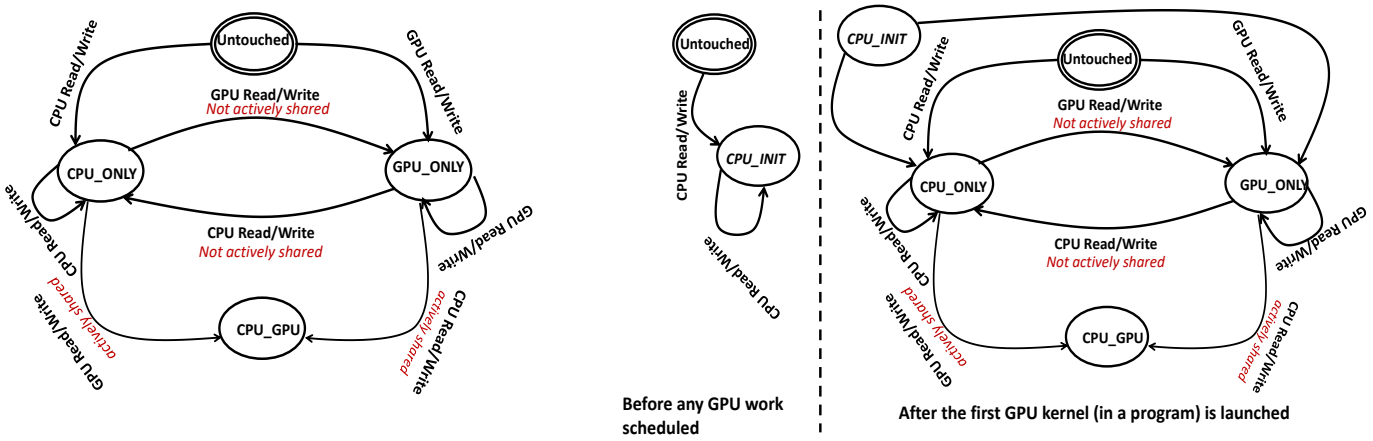
The SAHC adds three new page permissions that indicate ownership of a page by the CPU or by the GPU or by both -- CPU_ONLY, GPU_ONLY, and CPU_GPU. A page with CPU_ONLY permission is accessible only by the CPU and cache blocks in that page is cacheable only in the CPU's local cache hierarchy. The GPU is not permitted to access content of the page. Similarly, a page with GPU_ONLY permission is cacheable only in GPU cache hierarchy. A page with CPU_GPU permission can be accessed and cached by both CPU and GPU.

These page permission attributes are enforced by the hardware along with the conventional page permission checks and any violation (e.g., the CPU attempting to access a GPU_ONLY page) generates a page fault to the OS. The OS can then take a corrective action or generate an error in the same way traditional page access permissions are enforced. Figure 4 presents the access matrix for the newly introduced page permissions. For accesses in the green box, the coherence between CPU and GPU caches are enforced efficiently by page permission checks only *without needing coherence directory lookup*. The accesses in the red box generate page permission faults to the OS due to coherence permission violation. The accesses in the yellow boxes are managed by the hardware coherence mechanism without software intervention and may need hardware cache coherence directory lookup as in the baseline block-grain coherence protocol.

Thus, SAHC makes use of page permission checking capability in today's hardware to enforce coherence permissions between the CPU and GPU caches at a page granularity. Additionally, it uses existing permission fault mechanism of the OS to catch any coherence permission violations.

5.2.2 Coherence Permission Assignment

SAHC implements the policy for assigning the newly introduced page-grain coherence permissions in the OS. This allows SAHC to keep policy separate from the hardware mechanisms and allows flexibility in changing policies without changes in the hardware.



(a) Un-optimized state transition diagram.

(b) State transition with application-transparent optimization.

Figure 5. State transition diagrams for page permission assignment.

Figure 5(a) depicts a state machine diagram for the assignment of newly introduced page permission attributes. The first access to an untouched page will result in a page fault and the OS assigns CPU_ONLY (or GPU_ONLY) permission if the access is by the CPU (GPU). The subsequent accesses from CPU (GPU) on a CPU_ONLY (GPU_ONLY) page proceeds without any software intervention. However, if CPU (GPU) accesses a page with GPU_ONLY (CPU_ONLY) permission then the hardware generates a permission fault which is captured by the OS's page fault handler. The modified page fault handler then assigns CPU_ONLY (GPU_ONLY) page permission to the faulting page or assigns CPU_GPU permission. Assignment of CPU_GPU permission depends upon whether the faulting page is deemed *actively shared*. In the current implementation of the SAHC, we deem a page as actively shared when a given page incurs at least three page permission faults due to change in coherence permissions. We keep a count of number of coherence permission faults per page by adding a 2-bit saturating counter in the PTE. If a page has CPU_GPU page permission then accesses to cache blocks belonging to that page are kept coherent by the traditional block-grain hardware coherence mechanisms.

5.2.3 *Optimizations*

Guided by the analysis of communication patterns between the CPU and the GPU (Section 4), we propose several optional optimizations for the SAHC. We propose two types of optimization -- application-transparent optimizations and application-apparent optimizations.

Application-transparent optimization: This type of optimization does not require application modification, but needs updates to the runtime, the OS page fault handler, and to the hardware.

Analysis in Section 4 showed that a large fraction of toggles between CPU and GPU is due to initialization of data in the CPU and subsequent use of the data by the GPU. These toggles can trigger costly page permission faults in SAHC. The goal of this optimization is to alleviate these page permission faults by exploiting runtime's knowledge about when a program launches its first kernel on the GPU.

Specifically, we introduce another new page permission called *CPU_INIT*. Figure 5 (b) depicts an optimized version state transition diagram of page permission assignment policy. It has two state diagrams. The first state transition diagram (Figure 5(b) left) remains in effect before any work is scheduled on the GPU by a given program while the second one (Figure 5(b) right) comes into effect after a kernel is launched. A page is assigned coherence permission CPU_INIT before any work is scheduled on the GPU by a given program. After the first kernel is scheduled on the GPU by a given heterogeneous program, if the hardware page walker finds a page table entry with access permission as CPU_INIT then it updates the page permission to CPU_ONLY or to GPU_ONLY depending on whether the access is from the CPU or from the GPU,

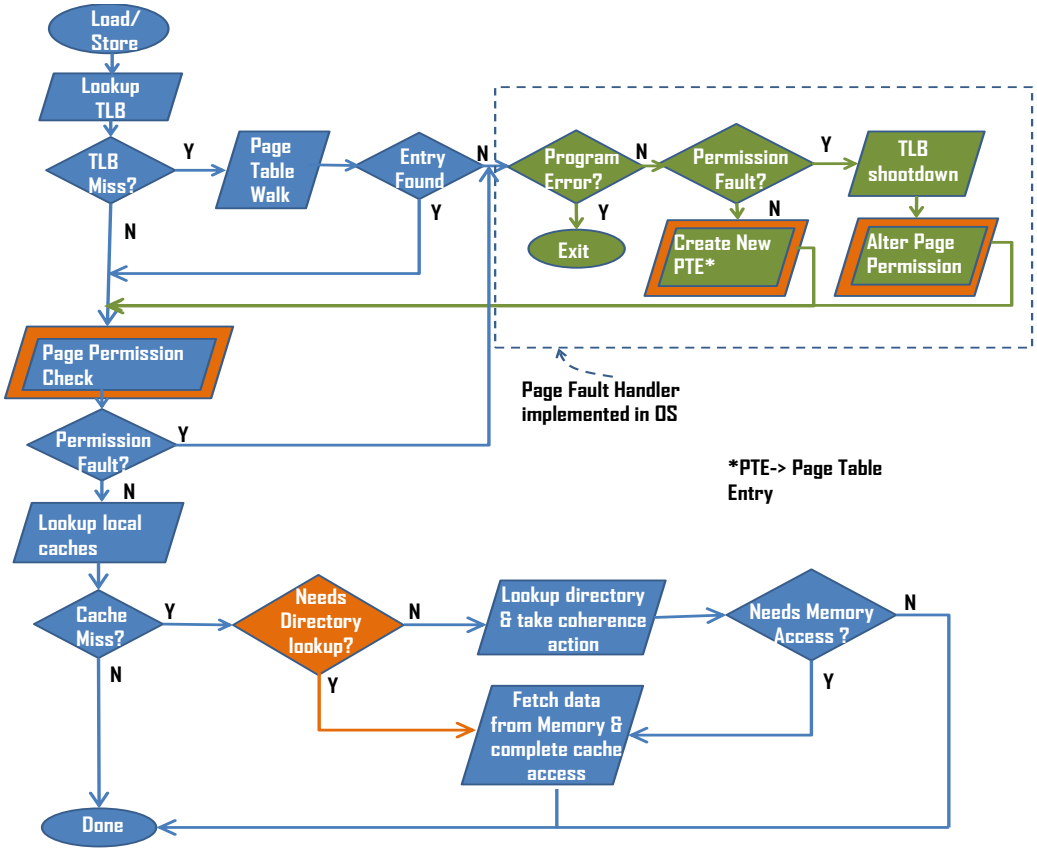


Figure 6. Flowchart of SAHC's working on a load/store on CPU/GPU. The green box and blue box signifies a mechanism implemented in software and hardware, respectively. Boxes with orange border signify mechanism that is modified in SAHC and an orange box signifies a new mechanism introduced by SAHC.

no page fault is generated. Thus, when a page is initialized in the CPU and then is accessed by the GPU, no page fault is triggered. The runtime also flushes the CPU cache hierarchy and the TLBs when a program schedules its first kernel on the GPU to ensure that later accesses from CPU or GPU gets latest data. This also ensures that a page table walk is always triggered on the first access to a page after any work is scheduled on the GPU.

We observe that the above optimization can be easily implemented because the driver and/or the runtime knows when a heterogeneous program schedules a kernel. SAHC modifies runtime to set a single bit register (default un-set) when the very first GPU kernel is launched by a given program and resets it whenever a program finishes execution. Note that these actions happen only once during a program's lifetime.

Application-apparent optimization: Application-apparent optimization proposed in SAHC needs very minimal modifications to application code (often limited to one or two lines of annotation).

Driven by our analysis in Section 4, we propose a very simple *optional* optimization in the SAHC that alleviates page faults due to post-processing in CPU (Figure 3(b) and Table 1). We introduce a new runtime API called *gpuWorkFinish().* An application can invoke *gpuWorkFinish*() to indicate that the given program will not use GPU for processing any further. The runtime then flushes GPU cache hierarchy and sets a single-bit register called *gpu_work_finish*.

If this register is set the hardware does not raise page fault on a CPU access to a page with GPU_ONLY permission. Thus, this can avoid permission faults that would have otherwise happened due to post-processing of the GPU's data at the CPU. This register is reset when a program finishes execution. Note that this optimization needs *only single line of code change* in an application.

5.2.4 Putting All Together

We now describe how all components of SAHC work together.

Figure 6 shows flowchart of how SAHC hardware and software works on execution of a load or store instruction from a CPU or a GPU. First, the local TLB is looked up on every memory instruction (e.g., load, store) from the CPU or the GPU to find the desired address translation as in today's hardware. On a TLB hit, the page permissions are checked. SAHC extends page permission check to include coherence permission checks as described in the access matrix (Figure 4). On a permission violation the OS's page fault handler routine is invoked on a page fault as is in today's system. On a TLB miss the hardware page walker attempts to find the desired PTE in the memory-resident page table. If the entry is found, the page table walker populates the TLB accordingly. Otherwise, the OS's page fault handler is invoked as in today's hardware.

The page fault handler has two responsibilities – (1) raise exception (e.g., segmentation fault) if the access is deemed illegal, and (2) if the access is legal then create or update the corresponding page table entry with requisite page permissions. SAHC modifies page fault handler to implement the policy of page permission assignment described in Sections 5.2.2 and 5.2.3. Once a memory instruction has the requisite page permissions to proceed, the cache hierarchy is looked up to find the desired cache block just like in baseline hardware. If the desired cache block is found in the local cache hierarchy (e.g., CPU request finds the block in CPU cache hierarchy), then the request completes. However, on a miss in the local cache hierarchy one of the two following actions happen. If the access permission of the page on which the requested cache block falls contains CPU_GPU then the hardware coherence directory is looked up to ascertain if the remote cache hierarchy (i.e., GPU cache hierarchy for a CPU instruction and vice-versa) contains the requested cache block as in the baseline. Otherwise, the hardware coherence directory is bypassed and the request is sent to unified L3 cache and to the off-chip memory in parallel. This is correct as the page access permission confirms that the cache block cannot be in the remote cache hierarchy. Since the page access permission is not usually available at the cache hierarchy the SAHC augments cache request messages with a single bit that encodes whether requested cache block falls in a page with CPU_GPU access permission.

Benefits: The above-mentioned design philosophy has several important benefits that align well with the goal of our work as mentioned in Section 3.

1. SAHC can bypass the hardware coherence for large fraction of accesses by utilizing page permission checking for determining the need for a coherence transaction.
2. SAHC alleviates the need to revamp cache coherence for heterogeneous systems as the hardware coherence is exercised only selectively.
3. Use of hardware cache coherence only for data that frequently moves between CPU and GPU enables good performance by selectively avoiding large number of potential page permission modifications.

6. Evaluation

In this section, we describe our evaluation methodology and then detail our evaluation of the SAHC.

6.1 Evaluation Methodology

Simulation: We simulate a heterogeneous processor with out-of-order CPU cores and a GPU based on AMD Graphics Core Next architecture [3] using gem5 simulator [2].

The simulated CPU-GPU processor has two CPU cores and 32 GPU CUs. We model extreme memory bandwidth of 700 GB/s to model growing trend of ever increasing bandwidth on such system. Table 2 shows the parameters used in the simulations. 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 SAHC in the above-mentioned simulation framework. Table 2 shows the simulation parameters used.

Workloads: We draw applications from two benchmark suites to evaluate SAHC design. We evaluated seven Rodinia [5] benchmarks and four AMD APP SDK [20] workloads. We modified these programs to work with a shared memory model, which is similar to the model introduced in OpenCL™ 2.0 and HSA [22]. In the modified version of the programs, we allocate only one copy of the memory buffer for each data structure, and use it for both CPU and GPU computation. This is different from the original implementation where separate memory buffer is allocated on the device memory and data is copied between the two buffers. We remove the use of OpenCL memory buffer calls, such as clCreatebuffer and clEnqueueRead/Writebuffer. The pointers to the host buffers are directly passed to the kernel for subsequent GPU.

The programs from Rodinia suite include a machine-learning algorithm; breadth-first search (bfs), HotSpot (hotspot), a thermal simulation for processor temperatures; LU Decomposition (lud); Needleman-Wunsch (nw), a global optimization method for DNA

Table 2. Simulation Parameters.

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)
Page permission fault latency	5000 CPU cycles (Avg.)

sequence alignment; kmeans (km), a clustering algorithm used in data mining; and, speckle-reducing anisotropic diffusion (srad), a diffusion algorithm. Programs from AMD APP SDK includes: bitonic sort (bitonic), histogram (histogram), graph coloring (coloring), and matrix multiplication (matrixmul).

6.2 Results

Performance Comparison: Figure 7 depicts the performance comparison across four configurations – ideal, baseline, SAHC with application-transparent optimizations (*SAHC-T*) and SAHC with application-apparent optimization (*SAHC-A*). The ideal and baseline configurations are as described in Section 3. More specifically, ideal mimics an unrealistic system with no coherence overheads while baseline simulates conventional system where coherence is enforced by the hardware coherence directory protocols (Figure 1). SAHC-T implements page assignment policy depicted in Figure 5 (b). SAHC-A adds the ability to notify end of work in the GPU by an application on top of SAHC-T.

We observe that several workloads, like bitonic, matrixmul, and kmeans, can incur significant performance overhead due to cache coherence in the baseline configurations. SAHC-T significantly reduces or almost eliminates coherence overheads for bitonic, coloring, histogram, and kmeans. However, SAHC-T can potentially hurt performance for bfs, matrixmul, and nw due to costly page faults (detailed in next subsection). Many of these page faults occur due to post-processing of GPU-produced data on the CPU. SAHC-A can eliminate many of these costly page permission faults using application provided hint on when no further work is scheduled on GPU. Note, this requires only a single line code modification in the application code.

Analysis: In this subsection, we analyze the results described above. Specifically, we measure the fraction of the coherence directory lookup avoided due to SAHC and also measure overheads introduced by SAHC.

The first two result columns of Table 3 show the percentage of hardware coherence directory lookup saved by SAHC-T, and SAHC-A. We observe that often SAHC saves 99-100% of coherence directory lookups needed in baseline architecture. This

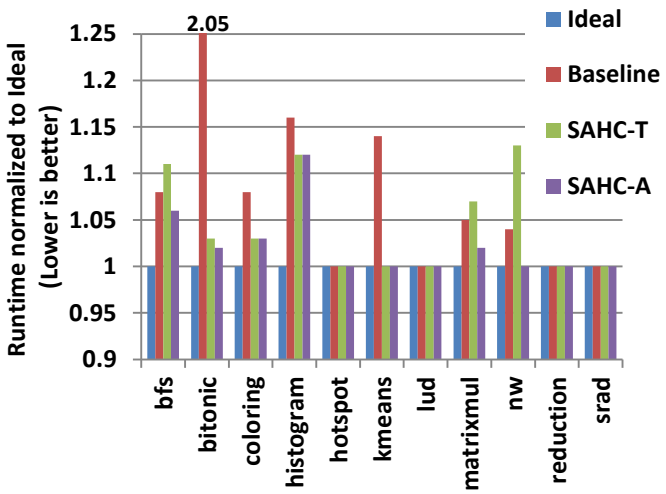


Figure 7. Performance normalized to ideal system.

is expected as our analysis in Section 4 showed that very large fraction of directory lookups is unnecessary.

SAHC incurs performance overhead due to slow page faults triggered by changes in the page-grain coherence permissions. The last two columns of Table 3 shows the number of such page permission faults introduced by the two flavors of SAHC-T and SAHC-A. We observe that for applications like bfs, nw, SAHC-T incurred significant number of page faults and this explains the overheads seen in Figure 7 for these applications. Further, we observe that for many of these applications SAHC-A eliminates large number of these page permission faults by eliminating the ones caused due to post-processing of data by CPUs.

Table 3. Hardware Directory lookup saving and page permission faults by SAHC.

	Percentage of hardware directory lookup saved		Number of page permission faults	
	SAHC-T	**SAHC-A**	**SAHC-T**	**SAHC-A**
bfs	98.64	99.61	9530	10
bitonic	100	100	1	1
coloring	99.44	99.44	9	9
histogram	100	100	130	129
hotspot	100	100	1	1
kmeans	99.01	99.01	617	617
lud	100	100	1	1
matrixmul	100	100	259	0
nw	100	100	4100	0
reduction	100	100	18	17
srad	100	100	262	0

7. Related Work

Researchers have proposed several ways to enable coherence in emerging heterogonous systems. One such proposal is Cohesion [10], which implements heterogeneous coherence using a combination of software and hardware techniques where data is allowed to dynamically migrate between coherence domains

owned by the CPU and GPU. The approach is shown to work well for bulk-synchronous data sharing, but the design requires all misses with the local cache hierarchies to query the system-level directory and determine what coherence domain owns the block. More recently, Power et al. [15] proposed using a region-granular directory to avoid unnecessary lookup at the system-level directory. However, their approach always manages coherence at the region granularity and it requires a significant redesign of today's block-based cache coherence mechanism. In addition, Singh et al. proposed an intra-GPU coherence protocol that uses timestamps to manage cache block permissions [17]. Their work focused on intra-GPU coherence, whereas SAHC focuses on inter-CPU-GPU coherence.

In addition to heterogeneous coherence solutions that leverage hardware, recent software-only solutions have also been proposed. In particular, PTask [16] introduced a task graph library that manages the memory coherence without programmer intervention. Similarly, the "Unified Memory" support in CUDA 6 automatically migrates data between CPU and GPU memories by leveraging system software and existing virtual address translation support [21]. In addition, Asymmetric Distributed Shared Memory (ADSM) and Gelado et al.'s software solution, called GMAC, also provide a logically shared address space between the CPU and GPU [7]. In comparison, SAHC strives to avoid the performance overhead of software-only coherence, while requiring less hardware support than prior hardware-based solutions.

Beyond heterogeneous and GPU coherence proposals, several prior efforts investigated reducing coherence bandwidth in CPU-based systems. For example, JETTY [14] and stream registers [9] proposed filtering out cache snoops using specialized hardware structures. Meanwhile others have proposed region-based hardware coherence [13], [19], as well as virtual tree coherence [6], subspace snooping [11], in-network coherence filtering [1], and directories based on bloom filters [12].

Reducing directory resources in CPU-based systems has also been a recent focus of researchers. For instance, spatiotemporal coherence tracking [2] reduces directory size by tracking private data at a region-granularity instead of at the cache block granularity. Similarly, multigrain coherence directories opportunistically use region-granular entries to reduce storage overhead [18].

Finally, R-NUCA [8] proposed to use OS page classification for achieving better cache hit rates and/or lower cache access latencies in large NUCA caches. R-NUCA uses page classification information to guide replication, migration, and placement of cache blocks according to dynamic behavior of the data/instructions. Similar to R-NUCA, our proposal makes use of OS page classification technique for very different purposes. SAHC uses page classification for efficient cache coherence in heterogeneous systems while R-NUCA strives to better manage NUCA caches.

8. Conclusion

Efficiently supporting cache coherence in emerging heterogeneous architectures without necessitating application management of coherence is challenging. SAHC addresses this challenge by extending virtual memory subsystem in novel way to encode coherence permissions in page attributes. SAHC then uses minimal modifications to system software (OS and runtime) and hardware to enable efficient coherence across CPU and GPU caches. More importantly, unlike previous proposals, SAHC does not need revamping cache coherence hardware nor does it necessitate application-management of coherence.

9. ACKNOWLEDGMENTS

Authors thank Ayse Yilmazer for her help with the gem5 simulator. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. The format ARM® is a registered trademark of ARM Limited. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

10. REFERENCES

[1] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-Network Coherence Filtering: Snoopy coherence without broadcasts," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. MICRO-42*, 2009, pp. 232–243.

[2] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2012, pp. 341–350 [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.39.

[3] AMD Radeon Graphics Technology, "{AMD Graphics Cores Next (GCN) Architecture White Paper}," Jun. 2012.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[5] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *2010 IEEE International Symposium on Workload Characterization (IISWC)*, 2010, pp. 1–11.

[6] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti, "Virtual Tree Coherence: Leveraging Regions and In-network Multicast Trees for Scalable Cache Coherence," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2008, pp. 35–46 [Online]. Available: http://dx.doi.org/10.1109/MICRO.2008.4771777.

[7] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2010, pp. 347–358 [Online]. Available: http://doi.acm.org/10.1145/1736020.1736059.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2009, pp. 184–195 [Online]. Available: http://doi.acm.org/10.1145/1555754.1555779.

[9] N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally, "Stream register files with indexed access," in *Software, IEE Proceedings-*, 2004, pp. 60–72.

[10] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: A Hybrid Memory Model for Accelerators," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2010, pp. 429–440 [Online]. Available: http://doi.acm.org/10.1145/1815961.1816019.

[11] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace Snooping: Filtering Snoops with Operating System Support," in *Proceedings*

of the 19th International Conference on Parallel Architectures and Compilation Techniques, New York, NY, USA, 2010, pp. 111–122 [Online]. Available: http://doi.acm.org/10.1145/1854273.1854292.

[12] P. Lotfi-Kamran, M. Ferdman, D. Crisan, and B. Falsafi, "TurboTag: Lookup Filtering to Reduce Coherence Directory Power," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, New York, NY, USA, 2010, pp. 377–382 [Online]. Available: http://doi.acm.org/10.1145/1840845.1840929. [Accessed: 25-Nov-2014]

[13] A. Moshovos, "RegionScout: exploiting coarse grain sharing in snoop-based coherence," in *32nd International Symposium on Computer Architecture, 2005. ISCA '05. Proceedings*, 2005, pp. 234–245.

[14] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2001, p. 85– [Online]. Available: http://dl.acm.org/citation.cfm?id=580550.876432. [Accessed: 25-Nov-2014]

[15] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2013, pp. 457–467 [Online]. Available: http://doi.acm.org/10.1145/2540708.2540747. [Accessed: 20-Nov-2014]

[16] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions to Manage GPUs As Compute Devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2011, pp. 233–248 [Online]. Available: http://doi.acm.org/10.1145/2043556.2043579.

[17] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," *IEEE Micro*, vol. 34, no. 3, pp. 69–79, May 2014.

[18] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain Coherence Directories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2013, pp. 359–370 [Online]. Available: http://doi.acm.org/10.1145/2540708.2540739.

[19] J. Zebchuk, E. Safi, and A. Moshovos, "A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2007, pp. 314–327 [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.5.

[20] "AMD App SDK" [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/

[21] "CUDA:Unified Memory." [Online]. Available: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[22] "HSA Foundation." [Online]. Available: http://www.hsafoundation.com/