Building Heterogeneous Unified Virtual Memories (UVMs) without the Overhead

KONSTANTINOS KOUKOS, Uppsala University
ALBERTO ROS, University of Murcia
ERIK HAGERSTEN and STEFANOS KAXIRAS, Uppsala University

This work proposes a novel scheme to facilitate heterogeneous systems with unified virtual memory. Research proposals implement coherence protocols for sequential consistency (SC) between central processing unit (CPU) cores and between devices. Such mechanisms introduce severe bottlenecks in the system; therefore, we adopt the heterogeneous-race-free (HRF) memory model. The use of HRF simplifies the coherency protocol and the graphics processing unit (GPU) memory management unit (MMU). Our protocol optimizes CPU and GPU demands separately, with the GPU part being simpler while the CPU is more elaborate and latency aware. We achieve an average 45% speedup and 45% energy-delay product reduction (20% energy) over the corresponding SC implementation.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems;**

Additional Key Words and Phrases: Multicore, heterogeneous coherence, GPU MMU design, virtual coherence protocol, directory-less protocol

ACM Reference Format:
Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building heterogeneous unified virtual memories (UVMs) without the overhead. ACM Trans. Archit. Code Optim. 13, 1, Article 1 (March 2016), 22 pages.
DOI: http://dx.doi.org/10.1145/2889488

1. INTRODUCTION

As fused devices become state of the art (AMD APUs [AMD 2013], and Intel Sandybridge–Broadwell central processing units (CPUs) with integrated graphics), the need for easier-to-use, general-purpose programming models increases. A fused architecture is essentially a *heterogeneous*, *clustered* system, with two distinct clusters: the CPU and the graphics processing unit (GPU) cluster. Each has its own cache hierarchy, with private L1s and a shared L2, while the two clusters may share a global lower-level cache (e.g., L3) or directly the main memory. Each cluster has distinctly different characteristics and compute capabilities, which necessitate a different approach. For example, the CPU cluster is latency sensitive and optimized to run complex, highly irregular codes, while the GPU cluster is capable of tolerating high access latencies but

This work is supported by the Swedish Research Council UPMARC Linnaeus Centre, the EU Project LPGPU FP7-ICT-288653, the Spanish MINECO, as well as European Commission FEDER funds, under Grant No. TIN2012-38341-C04-03, and the "Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia" under the project "Jóvenes Líderes en Investigación" 18956/JLI/13.
Authors' addresses: K. Koukos, E. Hagersten, and S. Kaxiras, Uppsala University, Lägerhyddsvägen 2, 752 37, Uppsala, Sweden; emails: {konstantinos.koukos, erik.hagersten, stefanos.kaxiras}@it.uu.se; A. Ros, Computer Engineering Department, University of Murcia, 30100 Murcia, Spain; email: aros@ditec.um.es.

DOI: http://dx.doi.org/10.1145/2889488

restricted to a data-parallel programming model (e.g., CUDA [Nvidia 2015] or OpenCL [Munshi et al. 2011]).

Although both OpenCL and CUDA have made big strides to facilitate the programmer with the ability to easily offload data-parallel tasks to the GPU, synchronization and memory copies remain a burden and in many cases introduce inefficiencies (e.g., if memory copies do not overlap with computation). Since the data-parallel programming model was not designed with coherence in mind, the programmer is responsible to handle data migration explicitly—that is, between GPU shared memory and device memory, and between main memory and device memory (in non-fused contemporary systems). While caches exist both in CPUs and GPUs, there is currently no practical end-to-end (any cache to any other cache) coherence to simplify the programming models. Providing end-to-end hardware coherence for heterogeneous systems is proving to be a challenging task due to the different bandwidth/latency demands of CPUs and GPUs.

The first step to make those systems more programming friendly is to provide a cache-coherent unified address space. Research proposals either employ sequential consistency (SC) protocols to maintain coherence between devices (CPU–GPU) such as [Power et al. 2013, 2015] or exchange some programming ease for simplicity and adopt release consistency (RC) such as in Singh et al. [2013] and Hechtman et al. [2014]. To maintain SC across devices, every write performed by a GPU core is eagerly made visible to the device and to the entire system. For simplicity, this entails a write-through protocol for the GPU, with the downside of creating tremendous pressure on the GPU's own last-level cache, due to the massive parallelism of the streaming multiprocessors (SMs). Unsurprisingly, this pressure on the GPU L2 and the directory becomes one of the major roadblocks towards implementing system-wide coherence.[1]

Our experimentation shows that such coherence (which involves indirection and forwarding) introduces enormous access latencies to the CPU for those blocks that are owned by the GPU when its own L2 cache is under heavy congestion, as shown in Figure 1. For most applications, this latency is prohibitive (on average 5× slower than accessing the main memory). Although high latencies are not usually a serious problem for GPU cores, they become rather severe for the latency-sensitive CPU cores. Therefore, the CPU should avoid interaction with the GPU shared cache for latency purposes. The use of regional directories as proposed in heterogeneous system coherence (HSC) [Power et al. 2013] significantly reduces the bandwidth and MSHRs pressure of these systems but still maintains most of the complexity required to support SC, such as invalidations and indirections.

Our starting point to provide a unified virtual memory (UVM) for future heterogeneous systems without the overheads of SC proposals is to relax the memory consistency constrains. For that we adopt the heterogeneous-race-free (HRF) consistency model [Hower et al. 2014]. In contrast to SC that maintains a store order at any given point of the execution, HRF (similarly to SC for data-race free (DRF) [Adve and Hill 1990]) uses synchronization scopes and assumes race free execution of the program between synchronization points. Since a heterogeneous system is a *clustered*, *hierarchical* system, cores within a single cluster can synchronize among themselves, in groups or all together, while the whole cluster (device) can synchronize with another cluster hierarchically. In our approach we employ an asymmetrical cluster synchronization, which simplifies both the architecture and the programming model. This allows an individual CPU core to synchronize with any other CPU core or with the whole GPU cluster, and

[1]A significant contribution of Power et al. [2013] is to analyze heterogeneous system bottlenecks. Our observations about the system bottlenecks are in complete accordance with their findings despite using a different simulation infrastructure. The details are discussed in Section 5.

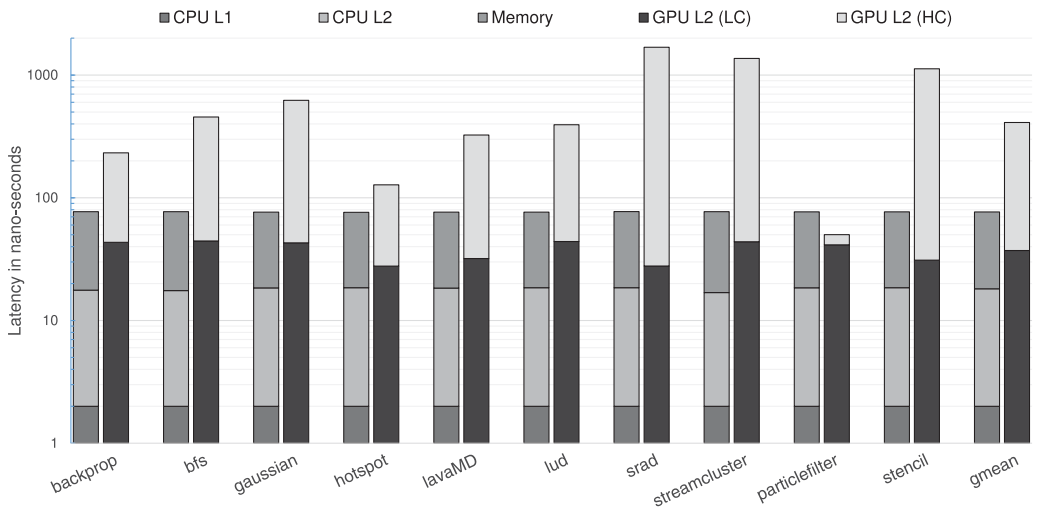

Fig. 1. Access latencies from a CPU core to some level of the memory hierarchy using a heterogeneous version of MOESI/VI [Power et al. 2015] protocol. *GPU L2 (LC)* is the latency measured from a CPU core to the GPU L2 under low congestion (e.g., at the end of a kernel execution), while *GPU L2 (HC)* is the latency under heavy congestion (e.g., when all GPU SMs generate traffic to it).

vice versa, while a single GPU core cannot synchronize with a single CPU core without enforcing device synchronization (for the whole GPU).

To efficiently support the HRF consistency model and introduce coherence in the system we use the VIPS-M [Ros and Kaxiras 2012] protocol and its hierarchical extension, VIPS-H [Ros et al. 2015]. These protocols are based on self-invalidation (during synchronization), and both rely on private/shared classification of data blocks, which is performed by the operating system (OS) using the page table and the translation lookaside buffer (TLBs). Dirty private blocks, update the next level using write-back (upon replacement), while shared blocks use write-through and self-invalidate at synchronization points.

Supporting address translations for fused heterogeneous systems is another challenging task. As proposed by Power et al. [2014], efficiently supporting GPU address translation requires a private TLB at each SM, a shared second-level TLB between cores, and a highly multithreaded page-walker. Although such an approach is efficient in performance, it comes with the tradeoff of using complex hardware structures (such as multiple private, fully associative L1-TLBs, and page-walkers), which increase the dynamic power of the core. Our approach simplifies the memory management unit (MMU) design with the introduction of virtual cache coherence [Kaxiras and Ros 2013] for the private caches, which eliminates the need for a private TLB at each GPU SM. Instead, we use a shared (among all SMs) TLB near the GPU L2 and virtually indexed, virtually tagged (VIVT) GPU L1 caches that eliminate the need for translations. This saves all the translation energy for the L1 hits, while it maintains the effectiveness of a shared GPU TLB [Power et al. 2014]. Since in our approach TLBs also keep page classification, the GPU TLB keeps the device classification between CPU/GPU. From the CPU, this information is available through a *Mirror-TLB*, which reflects if the page is accessed by the GPU. If a page is stored in the GPU TLB and is also accessed by the CPU, then it is shared. Else, if it is accessed solely by the CPU, then this page cannot exist in the GPU TLB (and therefore, in the Mirror-TLB) and is considered private to the CPU cluster (although it might be shared between CPU cores).

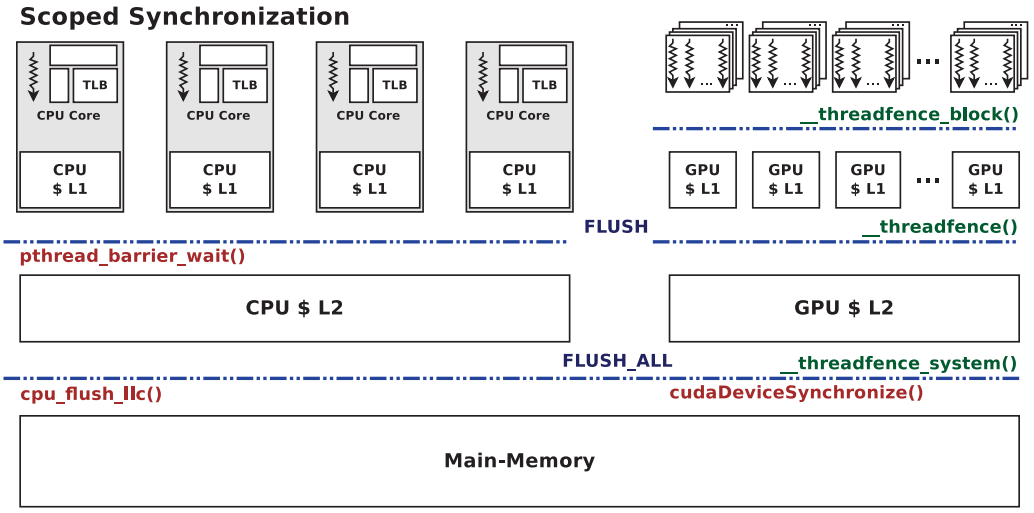

Fig. 2. Different synchronization primitives of a heterogeneous x86/CUDA system.

This article is organized in the following manner. Section 2 gives the overview of the synchronization model and the proposed architecture and describes the design of system components. Section 3 append the details of the experimentation setup. Section 4 evaluates our approach over contemporary systems while Section 5 provides a comparison with related research proposals. Section 6 is the related work, and, finally, Section 7 summarizes the goals of this article.

2. PROPOSED SYSTEM ARCHITECTURE

This section explains the proposed system architecture and all its required components. We start our analysis by introducing the memory consistency model supported by our architecture. Then we explain the adaptive heterogeneous data classification of our proposal and the coherence protocol (*VIPS-G*) that supports it. Afterwards, we append the detailed design of the caches and TLB and explain the required modifications in those structures to maintain the data classification. Finally, the last subsection summarizes our proposal and provides an overview of the system.

2.1. Scoped Synchronization

Contemporary GPUs use a weakly ordered memory model to make writes visible to different levels of the memory hierarchy. Therefore, heterogeneous applications require well-defined synchronization scopes, explicitly stated by the programmer. This approach has the advantage of reduced traffic, because stores are not required to become eagerly visible in the entire system. The tradeoff of this memory model, is a slightly increased programming complexity. Our observation is that the latest research proposals for future heterogeneous systems such as the HRF memory model, closely resemble existing GPGPU programming models such as CUDA and OpenCL. Although in contemporary heterogeneous systems there is a well-defined ordering model for the GPU cluster, this does not extend systemwide. Therefore, research proposals (e.g., HSC) employ sequential consistency for the CPU cores and between devices. Instead, we adopt the HRF model for the whole system by carefully interpreting the existing synchronization primitives of contemporary systems, as shown in Figure 2.

For simplicity and compatibility with existing applications and libraries, we adapt our programming model to the existing CPU/GPU synchronization primitives. This allows us to run most of the existing heterogeneous applications unmodified (or with

minimal modifications), while the architecture and the libraries abstract the details. Inside the CPU cluster we adopt a SC for DRF model which is supported by the VIPS coherence protocol [Ros and Kaxiras 2012]. From the programmer's perspective two threads running on different CPU cores can explicitly synchronize using pthread semantics[2]—for example, with a call to *pthread_barrier_wait()* or through *pthread_mutex_lock()/unlock()*. During synchronization we need to *FLUSH* the shared blocks of the CPU L1 caches involved, but as long as the data are not required to be visible to the GPU, no further action is required to the CPU L2 cache. To make all changes performed by any CPU thread (that has already committed them to L2) visible systemwide, some thread needs to explicitly call *cpu_flush_llc()*. This triggers a last-level (L2) cache flush[3] that writes back any remaining dirty shared blocks to the memory and invalidates all shared blocks.[4]

For the GPU we adopt the CUDA semantics unmodified. According to Nvidia [2015] *_threadfence_block()* forces writes performed by the calling thread to shared and global memory to become visible by all threads in the thread block. Additionally, reads from the calling thread see the updates performed by other threads in the thread block to shared and global memory. Since threads within the same block share the same L1 cache, no action is required by the memory system. Similarly, *_threadfence()* forces writes of the calling thread to shared and global memory to become visible by all threads in the device. In our approach this entails a GPU L1 cache flush. Since many GPU threads share the same L1 (within the thread block), it is important to ensure that they have all made their changes visible to the thread block before one of them can call *_threadfence()* (e.g., the first thread of each block that participates in the synchronization). Flushing the cache also ensures that loads after synchronization will read the updated values from other thread blocks, instead of stale data. Finally, there is also support for global synchronization across devices (CPU and GPU). For that, *_threadfence_system()* or *cudaDeviceSynchronize()* can be used to ensure that GPU last-level (L2) cache is flushed. The caller blocks until the device (GPU) has completed all preceding requested tasks. Before unblocking and after all kernels end an explicit call to *_threadfence_system()* is performed that flushes the GPU last-level cache, so writes become visible in the entire system. This also ensure that GPU threads after this fence will read the updated values for writes performed by the CPU.

Listing 1 gives a C/CUDA style (pseudocode) example of a multithreaded heterogeneous application.[5] The CPU has a few threads that are responsible for different jobs—for example, one of them may perform some I/O into a region of a shared buffer, while another could act as the GPU arbitrator which spawns GPU kernels and explicitly handles synchronization between devices. This is performed in two steps: (i) Ensure that CPU threads have flushed the corresponding private caches using the *pthread_barrier_wait* and the last level using *cpu_flush_llc()* and (ii) call cudaDeviceSynchronize() to wait for kernel completion and flush the GPU last-level cache. The GPU is running a typical GPGPU kernel that operates on global and shared data, and synchronize threads within a thread block using *__syncthreads()* or even across blocks with *__threadfence()*. The later is solely performed by the first thread of the thread-block and results to flushing L1 into the GPU L2. In this example flushing the

[2]To support the SC, for DRF model we modify the pthread synchronization primitives with explicit memory fences that flush private caches with the help of the protocol.

[3]A *FLUSH_ALL* message is forwarded to the CPU L2 and the core blocks until the operation is complete.

[4]The protocol, as explained in Section 2.2, is responsible for the classification of the blocks as private/shared which allows us to filter and selectively invalidate only shared blocks during synchronizations to reduce NoC traffic and memory bandwidth.

[5]The example omit the initialization details and in some cases we use a more abstract syntax to save space—for example, (args, ...) in function calls means *any arguments* that are not important for the example.

```
/* CPU thread code */
void *thread_func (void *t_data) {
    do {
        // if possible do some work

        // Sync and flush CPU private caches
        int res = pthread_barrier_wait(&barrier);
        if(res == PTHREAD_BARRIER_SERIAL_THREAD) {
            // Make writes visible system wide
            cpu_flush_llc();

            // Start GPU kernels ...
            kernel_run<<<dimX,dimY>>>(args, ...);
            cudaDeviceSynchronize();
        }
        else {
            // Error handling and
            // worker threads preparation
            // for the next iteration ...
        }
    } while ( ! complete );
}
```

```
/* GPU kernel code */
__global__ void kernel_run (args, ...) {
    __shared__ float local_data[SIZE];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx=threadIdx.x;
    int ty=threadIdx.y;

    // Do some work on local_data
    // and global data ...
    __syncthreads();

    // Do some more work on local_data
    __syncthreads();

    if(tx==0 && bx==0) {
        // Do some work for the whole
        // thread block and synchronize.
        __threadfence();
    }
}
```

Listing 1. Synchronization example of a multithreaded heterogeneous application at different levels.

GPU L2 is implicit at kernel end. If such synchronization is required before the kernel end, then *_threadfence_system()* can be used from within the GPU kernel.

2.2. Heterogeneous Data Classification

Weak consistency models such as DRF and HRF require flushing and self-invalidation of the caches that participate in the synchronization. Because self-invalidating and writing back every block stored in the cache at every synchronization point is an overkill, a filtering of the blocks that need to be flushed is required. In our approach, we perform selective flushing based on the observation that private blocks are not accessed by other processing units and, consequently, do not need to be flushed. Therefore, private blocks are allowed to cross synchronization points without the need to write-back dirty data and invalidate. On the contrary, shared blocks have to be written-back (if dirty) and self-invalidate in order for the cache to get an updated copy of the data in the next epoch. To reduce the amount of dirty shared blocks at synchronization points we use a coalesced (delayed) write-through policy for these blocks.

To detect private blocks, a simple and accurate classification is required. Based on previous proposals of Hardavellas et al. [2009], Cuesta et al. [2011], Ros and Kaxiras [2012], and Ros et al. [2015], we employ a classification of pages performed by the page table upon TLB misses. This classification can be flat or hierarchical. The flat classification [Hardavellas et al. 2009; Cuesta et al. 2011; Ros and Kaxiras 2012] is simpler since it only requires changing the state of each accessed page once, when it becomes shared, while the hierarchical [Ros et al. 2015] is more suitable for clustered hierarchies, but it requires multilevel classification and may involve multiple interrupts. For example, in a transition for a page that was initially shared only within a cluster and becomes globally shared among all clusters, all the cores previously sharing the page must be interrupted.

In prior work of Ros and Kaxiras [2012] and Ros et al. [2015], in both flat and hierarchical approaches, when a page becomes shared it does not revert to private unless if the page is evicted from memory. In fused devices, where the computation interleaves between the CPU and the GPU, all data eventually become shared, which jeopardizes the efficiency of the last-level caches in the system. Therefore, we enable classification adaptivity for device coherence (between CPU/GPU). A logical diagram of the proposed system is illustrated in Figure 3 (right), as opposed to a homogeneous non-adaptive system (left). Our approach suggests that (i) no classification using a

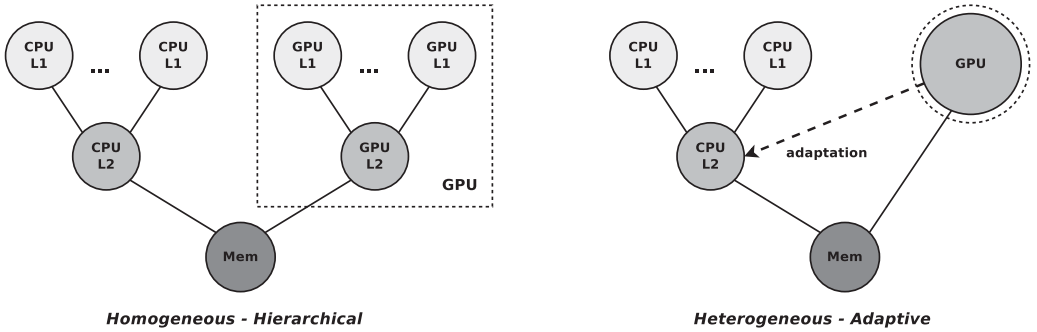

Fig. 3. Logical diagram of a homogeneous - hierarchical (on the left), versus an asymmetrical (heterogeneous) - adaptive (on the right) memory organization for data classification.

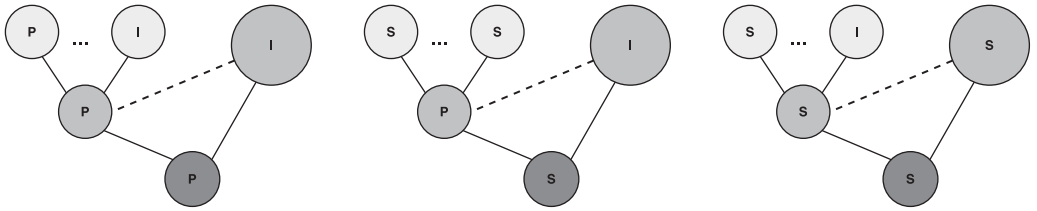

Fig. 4. Examples of different valid states of page classification in the system.

write-though protocol (2-state VI) for the GPU (between GPU L1s) and (ii) a heterogeneous classification (using a three-state protocol) for the CPU cores and the entire GPU, optimized to enable adaptation for across-device coherence.

2.2.1. No-Classification within the GPU. GPGPU applications benefit little from temporal locality (compared to traditional CPU applications) as shown in Che et al. [2009]. Therefore, we treat all data blocks in the GPU L1 caches as shared and use a simple write-through V/I protocol. Removing classification from the GPU L1s is an important simplification factor for a GPU design. This makes the entire GPU cluster look like a single node[6] in the logical diagram of Figure 3. To ensure program order, the programmer has to explicitly flush and self-invalidate each level of the cache using fences (as explained in Section 2.1).

2.2.2. Asymmetrical Classification for CPU Cores and the GPU. For the CPU cores and the entire GPU we use a hybrid classification, which is flat within the CPU cluster and the entire GPU and adaptive between devices. This is achieved by grouping all GPU cores within the same logical core and place it symmetrically to the CPU cores, as shown in Figure 3. The main difference between the CPU cores and the GPU is that the GPU bypasses the CPU L2 and instead reads from main memory (or potentially a third-level cache). This necessitates a hybrid approach to maintain coherence between devices as explained in Section 2.2.3, because neither a purely flat nor a hierarchical approach can be used efficiently. Figure 4 illustrates different valid states of page classification in the system.

The flat classification of the CPU cluster is similar to VIPS-M [Ros and Kaxiras 2012]. The first time that a core accesses a page and a TLB miss takes place (and

[6]Additionally, the use of virtual (VIVT) GPU L1 caches, in combination with a single shared GPU TLB near the L2 supports the logical placement of the entire GPU in the same level with the CPU cores (Section 2.5).

possibly a page fault), the page is classified as *Private* (Figure 4(a)). As long as no other core requests a block within the page, the page remains private and dirty data are written back only upon evictions. When a block that belongs to a private page is requested from a different core or device (e.g., GPU), the page becomes *Shared*. This transition happens with a *downgrade* (*Private to Shared*) request to the owner of the private page. When this occurs, all dirty blocks of this page have to be written back at the lower level of the memory hierarchy. When a page becomes shared, we do not distinguish between which cores or clusters it turned shared. Instead, it is considered universally shared, for the whole system using a flat classification (as in VIPS-M). This information is nonvolatile across the entire program execution, except for the pages that are evicted from the page-table, which will be classified as private to the first device that will access them again. Figure 4(b) shows an example of two CPU cores sharing a page. The page is marked as shared in the page-table and therefore, if GPU requests the page it will directly get it as shared without the need for a downgrade in the second level, as opposed to a purely hierarchical approach.

If the data owner is a CPU core and the data requestor is another CPU core, then the common level of sharing is the CPU L2, while if the second is the GPU, then the common level of sharing is the main memory. Therefore, we use a different type of (deep) write-back that forwards the writes to the main memory when the request comes from the GPU. The classification between the CPU cores and the entire GPU asymmetrically placed to the same level (as another logical core) is non-adaptive—that is, when a page becomes shared, it remains shared until it is evicted from the page table. This reduces the number of required downgrades based on prior execution history but may result in turning every page in the system eventually shared. To avoid that, we use a second level of classification between devices (CPU/GPU) which is volatile and offers adaptivity as explained below.

2.2.3. Adaptation Between Devices. The classification adaptation between devices is achieved by adjusting the CPU L2 coherence state based on the GPU L2 classification. If a page is accessed by both the CPU and the GPU, then this page is classified as shared both in the CPU TLBs and in the GPU TLB as shown in Figure 4(c). Assuming that the CPU L2 could easily look up the GPU TLB for a shared entry, then this entry could be directly used as the classification between devices (the existence of the page denotes that a page is shared). If the GPU does not participate in the shared classification, then no entry is available for this page in the GPU TLB, and this page can be considered private to the CPU cluster as shown in Figure 4(b). We enable the lookup of the GPU classification from the CPU L2 with the use of a *Mirror-TLB* near the CPU L2. *This mechanism enables the adaptive classification between devices* with the result of better filtering of the blocks that are flushed and self-invalidated in the CPU L2 (private blocks do not participate in this process).

Considering a page as private for the CPU in the absence of an entry in the GPU TLB generates an implicit transition at page replacements in the GPU TLB: an *upgrade* (*Shared to Private*) transition due to page eviction in the GPU TLB. Although this adaptivity is desired in the protocol and was the main target of our design, special attention is required to maintain correctness during *shared to private* transitions across devices. To avoid stale data in the CPU, one option would be to invalidate the data in the CPU so an updated copy of the data is refetched from memory after the transition. This approach suffers from a high invalidation penalty and also reduces the CPU L2 effectiveness since it may invalidate useful data.

To attack this problem, we use *Private*/*Shared* (P/S) bits in the L2 cache which are initialized based on the GPU classification. When we allocate a block in the cache, we also store its current coherence state locally (in the P/S bit). Two types of transitions

can occur: private to shared and shared to private. *Private to shared* transitions are performed by flushing dirty private blocks and mark them as shared in order to be invalidated at the next fence (similar to Ros and Kaxiras [2012]). *Shared to private* transitions do not require any action since blocks are already marked as shared (in the cache P/S bit) and will be self-invalidated at the corresponding fence.

2.3. False-Sharing Prevention

Our approach uses write-through for all blocks that belong to shared pages. Therefore, it is possible that two write requests for the same block are received concurrently at the lower level from different requesters. This occurs when two different cores or devices try to modify either the same word within the same cache line or different words that belong to the same cache line. The first is a true data race, and therefore the protocol cannot guarantee the correctness of the result because the execution is not DRF/HRF. In the second case no actual race exists, but if a coherence protocol operates at cache-line granularity, then merging the correct data becomes impossible. In our approach, we resolve false sharing at word granularity, using (i) a coalescing write-buffer and (ii) by sending diffs[7] on write messages.

Each message is of variable size from 1 word to a whole cache line, including a bitmap that encodes which are the dirty words in the corresponding cache line—for example, for a 128-Byte wide cache line we need 32 bits to encode all the dirty words in a message. Although the message packs all the dirty words of the cache line concatenated, the diff maintains the encoding for each word's position. Therefore, we can compress, extract, and merge all dirty words properly. We keep diffs only for the shared blocks, since only those blocks are prone to false sharing. This is achieved with the help of a coalescing write-buffer near each cache that keeps the dirty word bitmap while the data are updated directly in the cache.

When the buffer becomes full, we issue a write-back to the proper level of the memory hierarchy for the oldest entry in the write-buffer and free this entry. Because the cache already contains the correct values for the block, it is set clean when the lower level acknowledges the write-back. When the page upgrades from shared to private, the diffs at the write-buffer are no longer required, since false-sharing does not apply to private page blocks. Therefore, we free all write-buffer entries that belong to the newly upgraded private page, but we maintain the dirty bit in the cache. This helps towards minimizing the cost of a page upgrade transition and keeps up with the demands of the latency sensitive nature of the CPU.

2.4. TLB and Cache Design

The most important components to maintain coherence in our system are the TLBs and the page-table. Apart from the fields required for the translation (e.g., tag), our proposal extends the TLBs with a *Private*/*Shared* (P/S) bit to keep the page classification and a bitmap of *Valid*/*Invalid* (V/I) bits to indicate whether each cache line that is mapped to a page is valid or not.[8] We migrate the V/I bits from the cache to the TLB, for all those caches that the TLB keeps the page classification, and call these caches *state-less* (because the V/I state of each block is kept externally). Figure 5 shows the overall

[7]Diffs has been widely used in distributed shared memory systems like Cashmere [Dwarkadas et al. 1999], Treadmarks [Amza et al. 1996], and Argo [Kaxiras et al. 2015], as well as in DRF coherence protocols such as VIPS [Ros and Kaxiras 2012].

[8]The idea of decoupling cache fields from the cache structure has been extensively used in earlier work. Sembrant et al. [2013] proposed a way to make tag-less caches by mapping the index of a cache block in the TLB and additionally keep the cache way in which the block belongs, while Seshadri et al. [2014] propose a mechanism to store Dirty-Block Index in an external structure.

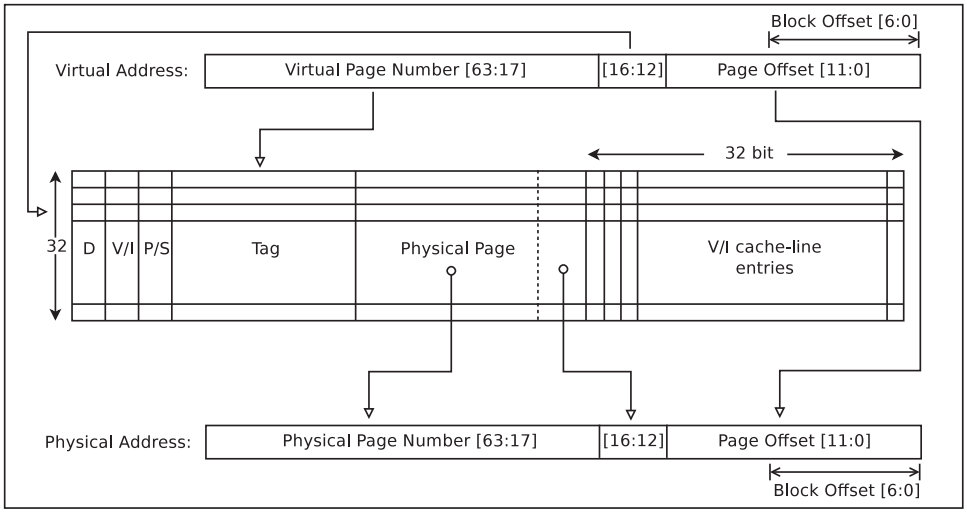

Fig. 5. Extended TLB design used to facilitate fast self-invalidations during synchronization.

design of the extended TLB in our approach.[9] This design provides an efficient way to bulk reset the shared blocks of each page selectively or for all shared pages (e.g., during self-invalidations). During synchronization, we reset the V/I bits for all blocks belonging to the page, while the tag and the classification bit remain valid in the TLB for future use. This facility comes with the tradeoff of cache underutilization for applications that have low spatial locality (e.g., valid-blocks span across pages).

Our system uses TLBs in combination with state-less caches for the CPU L1 data caches and the GPU L2 cache. These TLBs are indexed using virtual address. Adaptive classification (Section 2.2.3) between devices requires a mechanism to provide fast lookups of the GPU coherency state from the CPU L2 cache. Therefore, besides the main TLBs, we place a *Mirror-TLB*[10] near the CPU L2 cache, which allow us to retrieve the classification of the GPU TLB from the CPU. The mirror-TLB is also indexed using virtual addresses (VIPT). Since neither the GPU TLB nor the mirror-TLB is fully associative, we need to ensure that entries map to the same set in the GPU TLB and the mirror-TLB. Therefore, updates from the GPU TLB carries the index where the entry is stored in the GPU TLB and the physical address tag, which is also stored in the GPU TLB. Messages from L1 to L2 convey the virtual index along with the physical address tag, which is only a few bits (in our case 5 bits for a 32-set, 32-way TLB). The virtual index is then used to select the TLB set while the physical address is compared with the TLB tag. Since the CPU L2 already maintains classification (P/S bits), the mirror-TLB is used only to acquire the classification of newly allocated blocks (when no previous classification exist in the cache).

2.5. Overall Design and Address Spaces

This section summarizes the overall design of the system, which consists of two clusters: the CPU and the GPU, as shown in Figure 6. Each cluster may have different voltage/clock domains and meet different technology trends. The CPU design adopts

[9]The example in Figure 5 shows only one way of a 1024 entries 32-way set-associative TLB. The size of the V/I bitmap in this example is 32 because we use a cache line of 128 Bytes.

[10]A similar idea has been proposed in the *Piranha* architecture [Barroso et al. 2000], where in order to simplify coherence and avoid snooping at the L1s, duplicate tags and state are stored between L1s and L2.

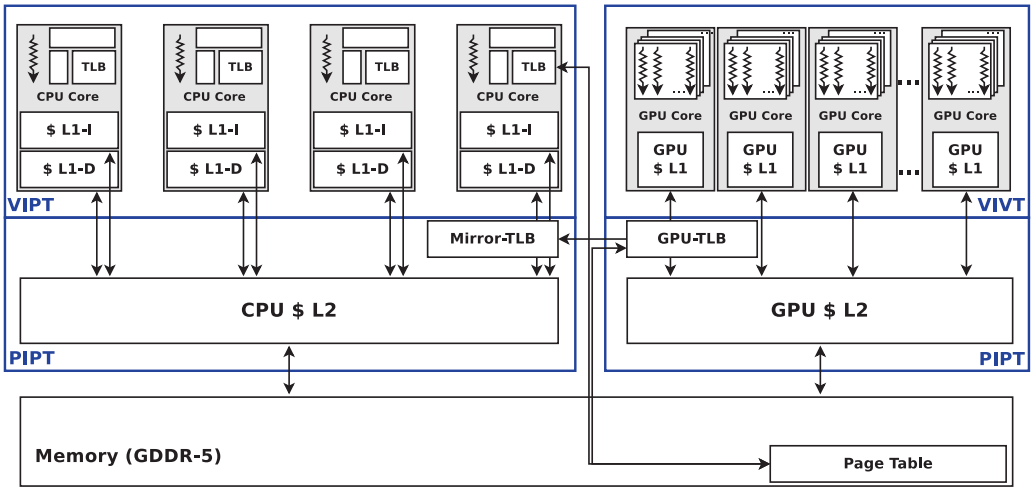

Fig. 6. The memory hierarchy proposed for a fused CPU/GPU system showing the address spaces, the caches, and the TLBs.

an existing state-of-the-art multicore CPU with private L1 instruction and data caches, a private TLB at each core, and a shared L2. In our approach, both data and instruction CPU L1 caches are virtually indexed, physically tagged (VIPT)—although a virtually indexed, virtually tagged (VIVT) configuration is also possible. For the data L1 caches we use a state-less design (V/I bits are external to the TLB). The TLB handles the translations and the data classification (P/S bits) and also retains the V/I bits of each state-less cache. For each access, along with the translation we also get the classification from the TLB which comes without extra latency overhead.

For the GPU L1, we use virtually indexed, virtually tagged (VIVT) caches, which eliminate the need for TLB translations upon a hit, which significantly simplify the GPU MMU design. For these caches, we do not employ a private/shared classification, due to the low temporal data reuse observed in GPU workloads, which makes private and shared blocks perform similarly. This suggests the use of a simple (V/I) write-through protocol, which is further optimized to coalesce writes on a small write-buffer. Unlike state-of-the-art proposals that use non-allocate writes for the GPU L1s, we employ a write-allocate policy for the misses. This is driven by the use of a shared TLB for the entire GPU that suggests minimizing L1 misses in order to reduce TLB pressure. As shown in our evaluation (Section 3), this policy improves the performance of all the evaluated GPGPU applications.

The GPU L2 is designed to handle the enormous write pressure of the GPU L1s and to maintain the coherence of the whole device with the rest of the system; therefore, it is highly multibanked. The GPU L2 is physically indexed and physically tagged (PIPT), and therefore it requires a translation before accessing it. For that, we use a shared GPU TLB (accessed before the L2) to handle the translations from GPU SMs and also to keep the classification for the whole device. Similarly to the GPU L2, the CPU L2 is also physically indexed, physically tagged (PIPT) and has the Mirror-TLB attached to assist the adaptive classification between devices. In addition to TLBs, the classification for P/S pages is also stored in the page-table. This allows us to retrieve the history of pages and classify them based on their previous state (when available), even when pages are not present in the TLBs due to a replacement.

Table I. Cache and Memory Specifications
DRAM configuration is based on Hynix H5GQ1H24AFR datasheet [Hynix 2013] MSHRs are per Bank—for example, CPU L2 has a total of 256 while GPU L2 has a total of 1536. GPU L1 is unified for both instruction and data.

Device	Type	Size	Assoc.	Banks	MSHRs/B.	Latency(ns)	Bandwidth
CPU L1	Data	64K	8	1	128	2ns	
CPU L1	Instr.	32K	4	1	128	2ns	
GPU L1	I & D	64K	8	1	64	8ns	
CPU L2	I & D	4M	64	8	32	16ns+2ns	
GPU L2	I & D	2M	32	16	96	18ns+8ns	
Memory	GDDR-5	4G				58ns+L1+L2	180GB/s

3. EXPERIMENTAL SETUP

This section describes the infrastructure and the settings used to evaluate our system. We evaluate our system proposal using gem5-gpu [Power et al. 2015]: a cycle-accurate simulator that integrates a gem5 [Binkert et al. 2011] CPU simulator with a GPGPU-Sim [Bakhoda et al. 2009] GPU simulator. The two simulators are connected using a *RUBY* memory system (integrated in gem5), which is used to simulate the memory hierarchy and the system coherence. The GPU energy is estimated using GPUWattch [Leng et al. 2013] power simulator, which is built in to gem5-gpu. To estimate the dynamic energy of the MMU (e.g., TLBs), the caches, and the NoC in higher detail, we use McPAT [Li et al. 2009] and Cacti [Thoziyoor et al. 2008; Muralimanohar et al. 2009]. We model a fused heterogeneous CPU/GPU architecture with unified DRAM for both devices, similarly to the default gem5-gpu implementation (VI_Hammer fusion). Timing simulation is performed in detailed mode, using an out-of-order (OoO) x86 architecture for the CPU and a Fermi-like GPU configuration. This takes into consideration the core type (CPU/GPU) and architecture, the cache latencies, the network congestion, as well as the TLB hit/miss penalties. We model our GPU TLB with the same size and hit/miss latencies with the reference gem5-gpu second-level TLB.

The simulation parameters are selected to depict a state-of-the-art CPU/GPU technology. For the CPU, we use 32KB private, instruction, and data caches, 8-ways set associative with low hit latencies as shown in Table I. For the GPU, we use a unified 64KB L1 cache similar to Power et al. [2015]. The latencies of the GPU caches are higher than those of the CPUs to depict technology trends. Table I reports latencies both in cycles (device local) and in nanoseconds since CPU and GPU are running in different frequency domains, 2GHz for the CPU and 700MHz for the GPU, respectively. GPU L2 handles the traffic from 16 write-though GPU L1s, which creates enormous pressure. To relieve this pressure and avoid long access latencies due to congestion, we split the L2 into 16 banks and use 96 MSHRs per bank. CPU L2 is taking significantly less pressure due to (i) application demands and (ii) much lower CPU L1 miss ratios, and therefore we split it into 8 banks and use 32 MSHRs per bank. Finally, we simulate 4GB GDDR-5 with 180GB/s bandwidth (based on Hynix H5GQ1H24AFR [Hynix 2013] specifications) as main memory for both devices. This decision is inspired by Sony PS4, which commercially introduced high-bandwidth GDDR-5 as system memory.

We perform our evaluation in a subset of the Rodinia [Che et al. 2009] benchmark suite, which has been modified by the gem5-gpu community to work in unified memory address space. Additionally, we include a 7-point stencil kernel from parboil [Stratton et al. 2012] benchmark suite, manually crafted to work with unified memory. Finally, we include *wave* and *octreepart* from the workloads used in Singh et al. [2013]. All these applications are written in CUDA with well-defined synchronization scopes that fits well with the HRF semantics proposed in this article.

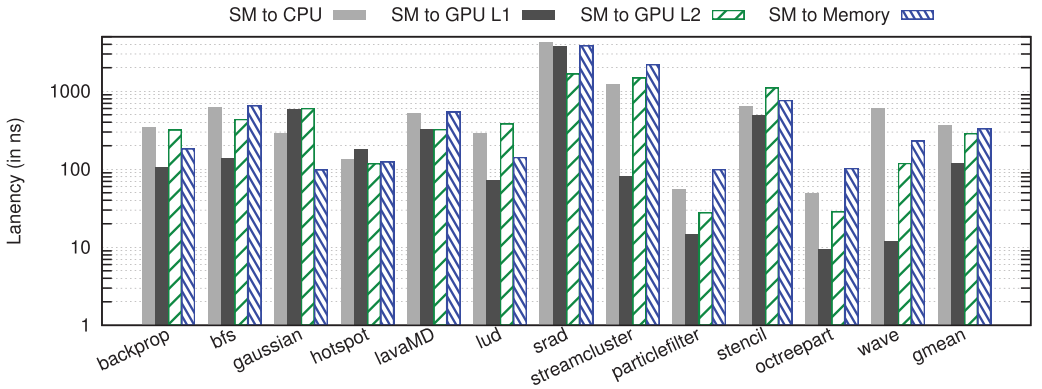

Fig. 7. Average latencies from GPU SMs to any level of the memory hierarchy for VI_Hammer.

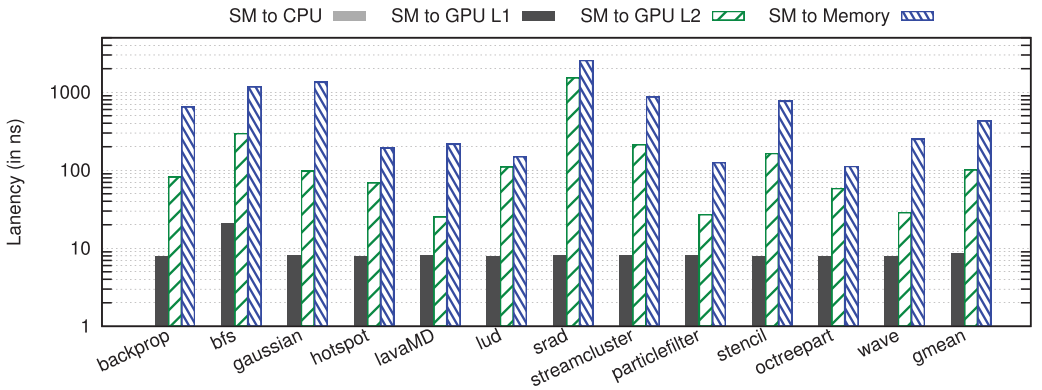

Fig. 8. Average latencies from GPU SMs to any level of the memory hierarchy for VIPS-G.

4. EVALUATION AND QUANTITATIVE COMPARISON

This section evaluates our approach of *VIPS-G* against a state-of-the-art heterogeneous system coherence implementation (*VI_Hammer*) in gem5-gpu. The later simulates a contemporary GPU using a write-through protocol, with write-invalidate and non-allocate writes. The CPU used is a contemporary OoO multicore running MOESI protocol. The classification between devices is kept in a shared directory. We start this section with an analysis of the system bottlenecks. Similarly to HSC, we observe that the greatest system bottleneck is the GPU L2 and for VI_Hammer the directory. Therefore, we shift our attention to the GPU. For the performance, we measure the GPU parallel total execution time as reported by the GPU simulator. We exclude the initialization part because it is performed solely by a single CPU core, which makes its performance very similar in both protocols. The reported time includes all synchronization and communication overhead between multiple calls of the same or different kernels. To explain the performance benefit of our approach versus the reference, we plot both the miss ratios and latencies at various levels of the memory hierarchy. Finally, we conclude our study with a performance and energy analysis of the proposed architecture.

4.1. Bottleneck Analysis

Similarly to Figure 1, which shows the access latencies from a CPU core to other system components, Figures 7 and 8 show the latencies from a GPU SM to any level of the

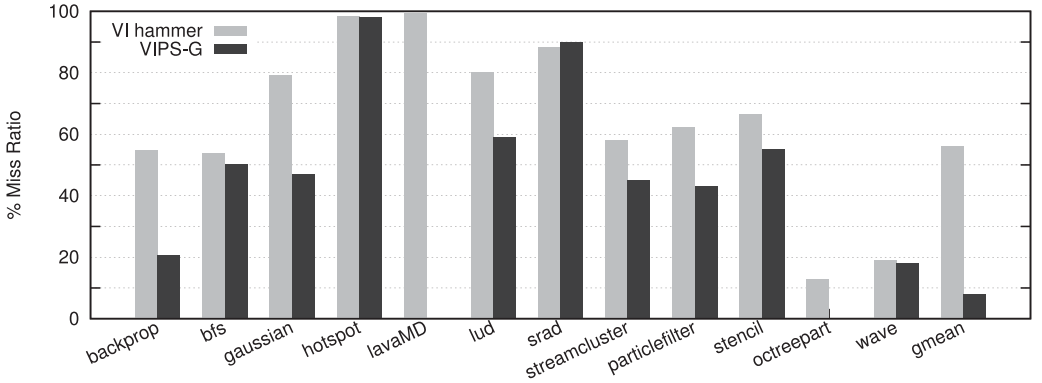

Fig. 9. GPU L1 miss ratio for VIPS-G versus VI_Hammer.

memory hierarchy that respond with the data upon a request. For example, *"SM to CPU"* is the average latency for a GPU SM to access a data-block which is originally owned by the CPU. *"SM to GPU L1"* is the latency of a GPU L1 hit, *"SM to GPU L2"* is the latency of a GPU L2, while *"SM to Memory"* is the memory access latency for a GPU SM. For VIPS-G, there is no *"SM to CPU"* bar in Figure 8 because we forbid direct CPU–GPU communication, as explained in Section 1. *A substantial difference between the two approaches is that VI_Hammer uses non-allocate writes for stores and always invalidate the block after a store (even on hits), while VIPS-G uses allocate writes and do not invalidate the block after a store.*

The first observation of Figures 7 and 8 is the deviation of the latencies measured compared to the nominal cache latencies that are shown in Table I. The measured latencies are fairly high, due to congestion in different levels of the memory hierarchy (especially in the GPU L2). For VI_Hammer, the accesses from a GPU SM to the CPU in most cases have a similar latency as accessing the main memory, while they increase the NoC pressure on the CPU side. Our approach minimizes this pressure with the use of the HRF consistency model in which, an updated copy of the data comes directly from the memory after synchronization. The only direct communication between CPU and GPU is a page downgrade (private to shared transition), which comes at a slightly higher cost compared to a single VI_Hammer coherence request but applies to all blocks of a page, since it is a bulk operation. Therefore, the corresponding downgrade latency per block is significantly lower.

Within the GPU cluster, we observe that VIPS-G has an almost-an-order-of-magnitude lower L1 access latency than VI_Hammer. This is because store hits do not strictly force the block to invalidate after use as in VI_Hammer. In the latter, only one application (*particlefilter*) is not severely penalized, while the rest are highly affected due to their high store/load ratio. On average, the non-allocate write policy introduces high penalties to VI_Hammer and contributes to the congestion of the GPU L2. This is shown in Figures 7 and 8 for bars *"SM to GPU L2,"* where we observe that VI_Hammer has a significantly greater penalty due to congestion by 4× on average. This makes the accesses to the GPU L2 cost almost the same as the memory accesses (and in some cases even more), and characterizes the GPU L2 and the directory as the bottlenecks of a system that uses VI_Hammer. VIPS-G, on the other hand, is *directory-less* and generates less burst traffic to the GPU L2 due to write-allocate non-invalidate policy of the L1s, which significantly reduces the L2 access latency.

Another important factor of performance is the hit ratio as shown in Figures 9 and 10 for the GPU L1 and L2, respectively. Our first observation is that the GPU L1

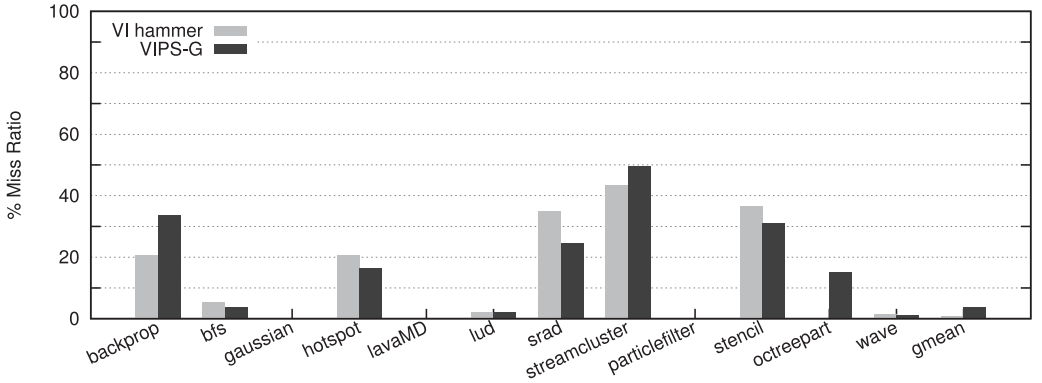

Fig. 10. GPU L2 miss ratio for VIPS-G versus VI_Hammer.

miss ratio is rather high (compared to the miss ratio observed in CPUs). We find that VI_Hammer has a dramatically higher average miss ratio (≈55%) compared to VIPS-G (≈ 8%), as shown in Figure 9. The reason is, again, the stores. VI_Hammer invalidates the block at every store (even when it is a hit); therefore, the load/store ratio of each application affects the total L1 miss ratio. The most extreme case is *lavaMD* [Szafaryn et al. 2011; Che et al. 2009] where VI_Hammer constantly invalidates all blocks due to stores, while for VIPS-G, the use of write-allocate policy reduces the miss ratio at less than 1%. Other applications that are also severely harmed by this policy are *backprop*, *gaussian*, and *particle-filter* while the rest of the evaluated workloads are less affected.

Figure 10 shows the GPU L2 miss ratios. Some applications like *hotspot*, *bfs*, *srad*, and *stencil* have a lower miss ratio in VIPS-G compared to VI_Hammer. There are two reasons for this: first, the higher hit ratio for stores in VIPS-G and, second, the high number of GET-S requests for blocks that are in a modified state of the MOESI protocol, which are eventually invalidated. On the other hand, there are some applications like *backprop*, *lavaMD*, and *streamcluster* for which VIPS-G has a higher miss ratio. The reason is the use of TLB to keep the V/I block, which comes with the cost of cache underutilization when there is low spatial block-locality per page, which in turn affects the miss ratio. Despite the higher L2 miss ratio in some applications, the lower cache access latency together with the higher L1 utilization, amortizes the penalty in performance.

4.2. Performance Analysis

As already discussed in Section 4.1, our proposed architecture has better cache hit ratio (for every application at the L1 and for some applications at the L2) and lower access latencies compared to *VI_Hammer*. This results in significant performance improvements, as shown in Figure 11. On average, we achieve a 45% speedup over VI_Hammer while some applications like *lavaMD* and *streamcluster* have an immense speedup of over 2.7×. Other applications, like *gaussian*, *srad*, and *stencil*, have a good speedup of ≈1.8× while the rest of the workloads benefit less. This leads to IPC improvement, with VIPS-G sustaining over 50% higher IPC on average compared to VI_Hammer, as shown in Figure 12.

4.3. Power Analysis

The high performance achieved by our approach, comes with the cost of higher power drain, due to the better utilization of SMs. For those applications that we significantly improve the performance, we observe an increase in the power consumption, as shown

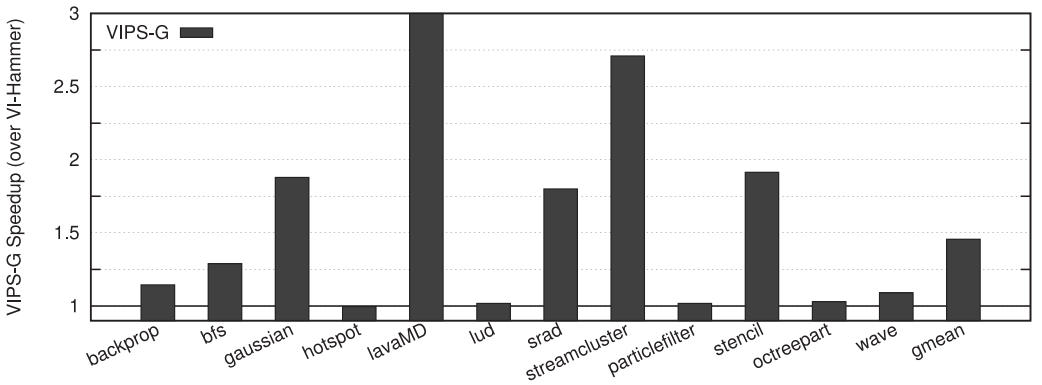

Fig. 11. Normalized speedup.

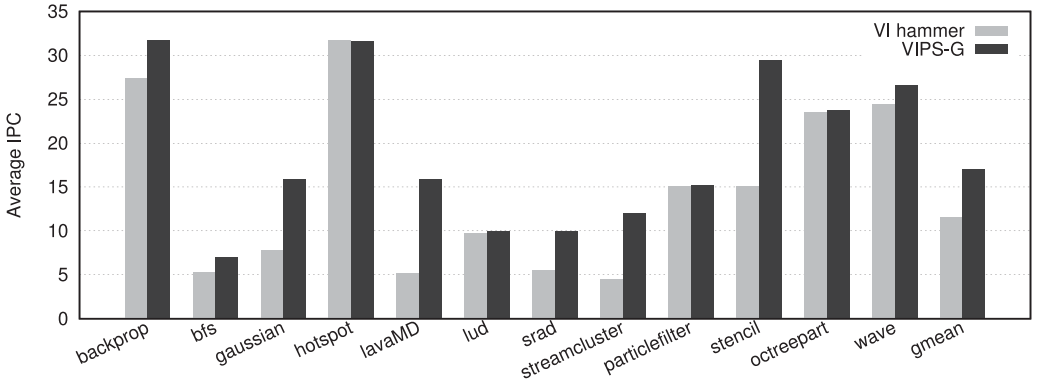

Fig. 12. Average IPC per GPU core.

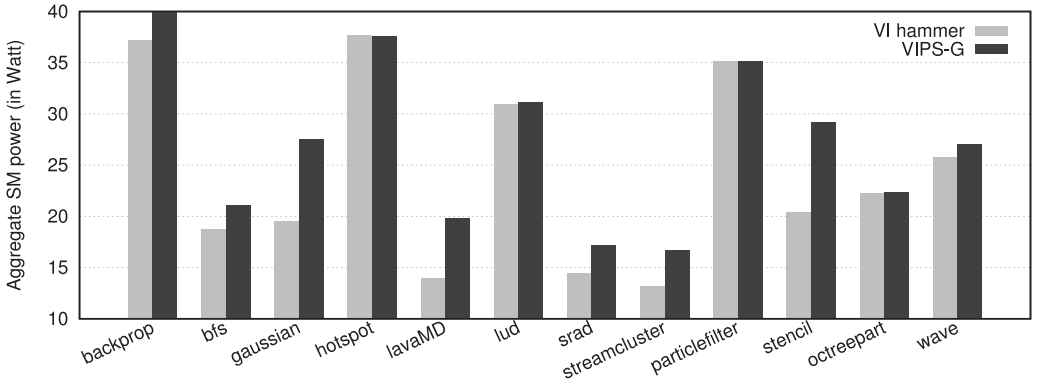

Fig. 13. Peak GPU core average power (in watts) for each protocol at 22nm as reported by GPUWattch [Leng et al. 2013] for a 16SM GPU.

in Figure 13. Despite the higher power consumption, the final energy and energy-delay product (EDP) are significantly lower (due to the performance improvements), as shown in Figure 14. To estimate the total power of the GPU we use GPUWattch [Leng et al. 2013], modeling 22nm technology for a 16SM integrated GPU. Figure 14 shows that our

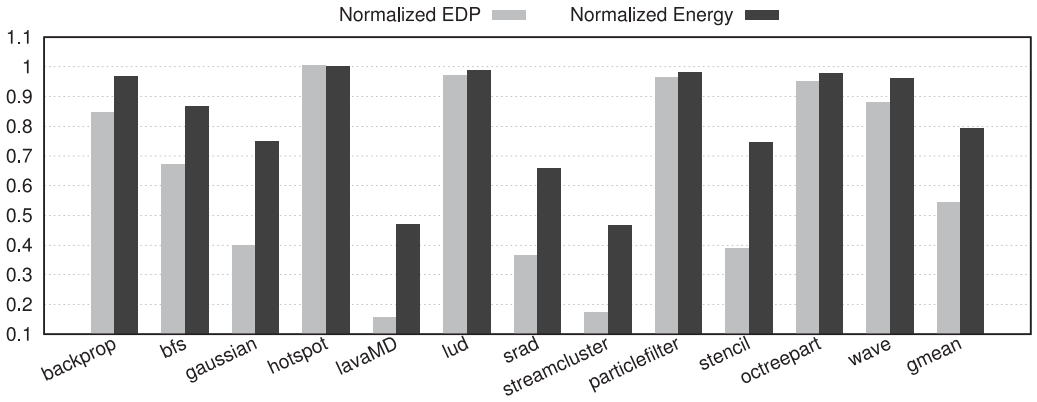

Fig. 14. Normalized total energy/EDP for VIPS-G (baseline is VI_Hammer).

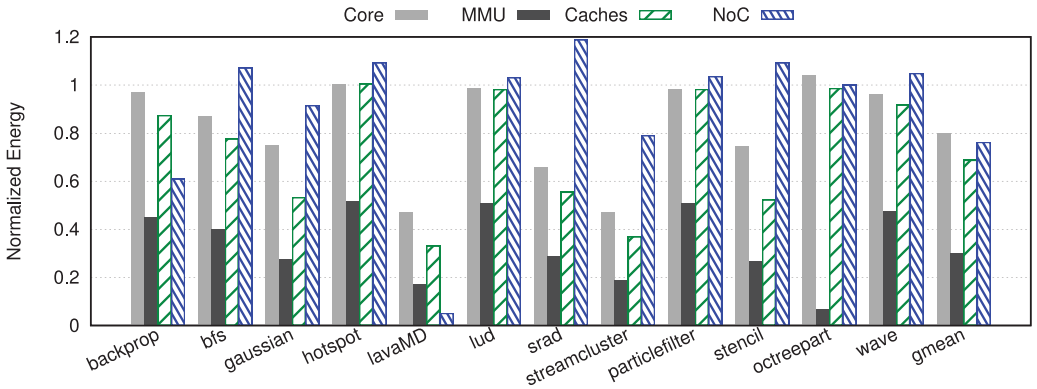

Fig. 15. Normalized per component energy for VIPS-G (baseline is VI_Hammer).

approach achieves a significant EDP reduction of 45% and around 20% on average lower energy, compared to VI_Hammer. For some applications like *lavaMD* and *streamcluster*, we observe over 50% energy reduction, due the performance improvements achieved.

Apart from the aggregate energy estimation for the GPU core (performed by GPUWattch), which yields an average 20% improvement, we perform a more detailed study of the energy consumption of various system components, such as the GPU-MMU, the caches, and the NoC, using *McPAT* [Li et al. 2009]. Figure 15 shows the normalized (to VI_Hammer) energy per evaluated component. Unsurprisingly, the use of virtual coherence and the elimination of private TLBs at each SM results in 70% dynamic energy reduction for the MMU (even by not accounting the page-walker energy for VI_Hammer). For the GPU caches, we observe that the energy savings follow the performance trend, because the total cache energy is dominated by leakage and is therefore heavily affected by the execution time instead of the total accesses to the cache. Finally, for the NoC energy of the GPU cluster, we observe that some applications have slightly higher dynamic NoC energy due to the higher traffic created by our decision to use a write-allocate policy in the GPU caches. Contrary to stores, for which write coalescing on the write-buffer and the use of variable-size messages can potentially reduce NoC traffic, data responses from the lower level send the whole block. This slightly increases the overall NoC traffic for some applications (e.g., srad,

hotspot, and bfs), while some other benefit significantly (e.g., lavaMD and backprop) by this policy. On average, we observe a 25% NoC energy reduction.

4.4. Area Reduction Analysis

One of the main goals of our proposal is to simplify the GPU MMU. This is achieved by removing the private TLBs from the SMs and instead use virtual L1s and a single, shared TLB close to the GPU L2 (with similar configuration to the GPU L2-TLB of VI_Hammer) for the translations of the entire cluster. A conservative[11] area analysis using *Cacti* shows that our MMU requires at least 49% less area compared to an approach that has private TLBs, such as the one proposed by Power et al. [2014] and used by Power et al. [2015]. For *VI_Hammer* we model 16 private TLBs with 32 entries, fully set-associative, and a shared 1024 entries 32-way set-associative TLB, while VIPS-G uses only a single 1024 entries 32-way set-associative *extended* TLB—that is, in the latter, we also account the extra area for the classification, owner, and V/I bits.

5. QUALITATIVE COMPARISON

In this section, we qualitatively compare our approach with prior research proposals.

To address the high bandwidth and MSRHs demands of future GPUs, *HSC* [Power et al. 2013] uses a regional directory and buffers in both CPU and GPU L2 caches to filter directory probes. If the permission can be acquired by the region buffer, then the requestor directly accesses the memory through a dedicated interconnect, otherwise it has to access the directory before accessing the memory. To support fused architectures, they adopt a GPU MMU design similar to Power et al. [2014] for the address translations. Compared to our approach, HSC simplifies device coherence since it removes the requirement for explicit synchronization between devices. Inside GPU, threads synchronize using the same primitives as in our work and in most contemporary GPUs. The advantage of our approach is simplicity. We achieve equally good performance without the need of region buffers by totally eliminating the directory and with a significantly simpler MMU design. The GPU TLB in our approach serves as a region directory (region size of one page) but without the extra hardware overhead. Furthermore, the use of virtual coherence in the GPU L1s is an additional simplification factor which reduces power and area by removing the private TLBs.

Temporal Coherence (TC) proposed by Singh et al. [2013] is an alternative that provides relaxed memory consistency.[12] Time-based coherence such as TC requires synchronized counters, which is (i) hard to implement across devices that run on different voltage–clock domains (such as CPU–GPU) and (ii) a single time-stamp mechanism is inefficient for irregular applications [Esteve et al. 2015], as those expected to run in the CPU on future heterogeneous systems [Arora et al. 2012]. The major roadblock of TC-weak is the write propagation time. Although writes are not delayed in TC-weak, the new values are visible to the other cores only after their time-stamp expires. In contrast, our approach employs atomic operations for races, which are guaranteed to update the corresponding cache level directly and make the changes visible to other threads with lower latency (without delays). This ensures the progress of other threads—for example, when loads are employed for spin-waiting.

QuickRelease (QR) [Hechtman et al. 2014] is a release consistency approach that use FIFO queues to enforce partial store order, so synchronization can complete without

[11]In our study, we model only the TLB area excluding the TLB page-walkers and their caches, due to limitations of the tools. This leads to a conservative estimation of the achieved benefit.

[12]Based on their findings TC-weak outperforms TC-strong by 30% and we therefore discuss TC-weak only. Since their proposal leaves CPU–GPU coherence as future work, we omit a direct quantitative comparison.

frequent cache flushes. In our approach, a write-buffer is used to minimize data-flushing during synchronization, similar to QR FIFOs. The insertion of fences in the write-buffer in QuickRelease is equivalent to flushing the write-buffer in our approach. One of the advantages of our approach is that we do not need separate read/write caches, and therefore the cache design is minimally affected. Furthermore, our approach does not require broadcasts to invalidate stale copies, because our caches self-invalidate at the corresponding synchronization operation.

6. RELATED WORK

As fused heterogeneous systems becomes the dominant future architecture, significant effort is spent both from the industry and academia to simplify their programming model. HSA Foundation members (e.g., AMD) are committed to provide heterogeneous coherence in future systems. NVIDIA also facilitates their latest CUDA versions with UVM (unified virtual memory) to eliminate memory copies from host to device. State-of-the-art academic proposals, such as *heterogeneous system coherence* (HSC) [Power et al. 2013], recognizes the need for an efficient hardware solution. Their study adopts a SC memory model for CPU inter-core and CPU–GPU coherence in combination with a simple write-through policy for the GPU. They enlighten various bottlenecks in the system, such as MSHRs and bandwidth pressure, and try to address directory design challenges for future systems with the use of region coherence. Our approach considers the benefits of regional coherence, while we try to apply it without the overhead of a directory, region buffers near the L2 caches, and dedicated interconnects between the L2s and the memory. For that, we adopt the VIPS-M [Ros and Kaxiras 2012] approach of page coherence. Our study shows that a fixed page-size granularity is very efficient in terms of performance and energy and comes with minimal extra cost for the system because it can be directly tracked in the TLB.

Hechtman and Sorin [2013] show that the memory consistency model on massively-threaded throughput-oriented processors (such as GPGPUs), has an insignificant impact on performance. However, there are various significant constrains to consider such as energy-efficiency, area, and implementation complexity. Our main goal is to simplify the architecture and we therefore adopt the HRF memory model [Hower et al. 2014]. We show that there is significant room for improvements in every aspect of the system and, in most of the cases, simplifications come even with a performance benefit. The use of the HRF memory model line up with various existing systems that already use scoped synchronization at various levels—for instance, Power7 [Kalla et al. 2010] architecture uses scoped broadcasts. On programming models, MPI [Gropp et al. 1999] provides a functionality for "scoped consistency synchronization," while CUDA synchronization semantics can be easily interpreted as scoped synchronization primitives for the GPU.

Inspired by Kaxiras and Ros [2013], we introduce virtual coherence for the GPU to eliminate most of the translation overhead and complexity. In contrast with the Power et al. [2014] proposal that relies on private TLBs at every GPU SM in coordination with a highly multithreaded page-walker for the translations, we use a single shared TLB for the whole GPU attached to the L2 and virtual (VIVT) address for the GPU L1s. This is possible with the use of a coherence protocol such as VIPS-G, which is based on self-invalidation and therefore does not involve upwards traffic as in the case of SC protocols.

Our protocol (VIPS-G) substantially differs from earlier VIPS protocol versions by being entirely heterogeneous. This is done by using only a subset of the protocol (V/I only) for an entire cluster (inside the GPU) and by designing device coherence to be asymmetrical and adaptive. Ros et al. [2015], in VIPS-H protocol, propose a generic (homogeneous) hierarchical approach. Such an approach involves the interruption of

several cores—for example, when a page transition from partially shared to globally shared occurs at L2. Instead, we employ an adaptive, hybrid data classification with the use of the GPU TLB and it's mirror-TLB to minimize software interactions for coherence purposes in the GPU. Finally, although not evaluated in this work, VIPS-G can also benefit from efficient spin-waiting techniques that do not rely on invalidations [Ros and Kaxiras 2015], thus allowing better performance without extra complexity.

7. CONCLUSIONS

Heterogeneous architectures pose new challenges for the design of future systems. To ease the programming models, device fusion along with the use of a unified memory address space becomes crucial. *Our goal is to show that this can be achieved efficiently by simplifying the architecture, instead of making future heterogeneous systems more complicated*. By carefully interpreting existing synchronization primitives, we allow legacy codes to run directly into our infrastructure and support future applications to be written in an easier way by taking advantage of the unified virtual memory. The introduction of virtual coherence for the GPU enables us to simplify the MMU design significantly, reduce the required TLB area by 50%, and achieve up to 70% MMU energy savings. By relaxing strict SC constraints with the use of the HRF consistency model, we are able to significantly simplify the protocol. For this, we propose VIPS-G, a heterogeneous protocol carefully designed to eliminate the bottlenecks created by the data-parallel GPU execution and maintain coherency efficiently inside the GPU cluster and between clusters. Thanks to the heterogeneous adaptive classification, our coherence approach is much simpler to implement and yet more efficient. Overall, we achieve an average 45% speedup, over 20% energy savings, and significantly lower EDP (over 45%).

REFERENCES

Sarita V. Adve and Mark D. Hill. 1990. Weak ordering – a new definition. In *Proceedings of the 17th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2–14.

AMD. 2013. APU ™. Retrieved from http://www.amd.com/en-us/innovations/software-technologies/apu.

Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared memory computing on networks of workstations. *IEEE Comput.* 29, 2 (Feb 1996), 18–28.

Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B. Baden, and Dean M. Tullsen. 2012. Redefining the role of the CPU in the Era of CPU-GPU integration. *IEEE Micro* 32, 6 (Nov 2012), 4–16.

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174.

Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 282–293.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Comput. Arch. News* 39, 2 (Aug. 2011), 1–7.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 International Symposium on Workload Characterization (IISWC)*. 44–54.

Blas Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José Duato. 2011. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 93–104.

Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. 1999. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proceedings of the 13th International Symposium on Parallel Processing (IPPS)*. 153–159.

Albert Esteve, Alberto Ros, Maria E. Gómez, Antonio Robles, and José Duato. 2015. Efficient tlb-based detection of private pages in chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (March 2015).

William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Vol. 1. MIT Press, Cambridge, MA.

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 184–195.

Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *Proceedings of the 20th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 189–200.

Blake A. Hechtman and Daniel J. Sorin. 2013. Exploring memory consistency for massively-threaded throughput-oriented processors. In *Proceedings of the 40th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 201–212.

Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free memory models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 427–440.

Hynix. 2013. Hynix H5GQ1H24AFR – 1Gb (32Mx32) GDDR5 SGRAM. (2013). http://www.hynix.com/.

Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. 2010. Power7: IBM's next-generation server processor. *IEEE Micro* 30, 2 (2010), 7–15.

Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 3–14.

Stefanos Kaxiras and Alberto Ros. 2013. A new perspective for efficient virtual-cache coherence. In *Proceedings of the 40th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 535–546.

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 487–498.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480.

Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, and Dan Ginsburg. 2011. *OpenCL Programming Guide*. Pearson Education.

Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories* (2009), 22–31.

Nvidia. 2015. CUDA C Programming Guide. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-fence-functions.

Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 457–467.

Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. 2015. Gem5-gpu: A heterogeneous CPU-GPU simulator. *Comput. Arch. Lett.* 14, 1 (Jan 2015), 34–36.

Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the 20th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 568–578.

Alberto Ros, Mahdad Davari, and Stefanos Kaxiras. 2015. Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies. In *Proceedings of the 21st IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 186–197.

Alberto Ros and Stefanos Kaxiras. 2012. Complexity-effective multicore coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 241–252.

Alberto Ros and Stefanos Kaxiras. 2015. Callback: Efficient synchronization without invalidation with a directory just for spin-waiting. In *Proceedings of the 42nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 427–438.

Andreas Sembrant, Erik Hagersten, and David Black-Shaffer. 2013. TLC: A tag-less cache for reducing dynamic first level cache energy. In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 49–61.

Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2014. The dirty-block index. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 157–168.

Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache coherence for GPU architectures. In *Proceedings of the 19th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 578–590.

John A. Stratton, Christopher Rodrigues, I.-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

Lukasz G. Szafaryn, Todd Gamblin, Bronis R. De Supinski, and Kevin Skadron. 2011. Experiences with achieving portability across heterogeneous architectures. *Proceedings of WOLFHPC, in Conjunction with ICS, Tucson* (2011).

Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008. CACTI 5.1. *HP Laboratories* 2 (Apr 2008).

Received May 2015; revised October 2015; accepted November 2015