Analyzing Memory Management Methods on Integrated CPU-GPU Systems

Mohammad Dashti
University of British Columbia, Canada
mdashti@ece.ubc.ca

Alexandra Fedorova
University of British Columbia, Canada
sasha@ece.ubc.ca

Abstract

Heterogeneous systems that integrate a multicore CPU and a GPU on the same die are ubiquitous. On these systems, both the CPU and GPU share the same physical memory as opposed to using separate memory dies. Although integration eliminates the need to copy data between the CPU and the GPU, arranging transparent memory sharing between the two devices can carry large overheads. Memory on CPU/GPU systems is typically managed by a software framework such as OpenCL or CUDA, which includes a runtime library, and communicates with a GPU driver. These frameworks offer a range of memory management methods that vary in ease of use, consistency guarantees and performance. In this study, we analyze some of the common memory management methods of the most widely used software frameworks for heterogeneous systems: CUDA, OpenCL 1.2, OpenCL 2.0, and HSA, on NVIDIA and AMD hardware. We focus on performance/functionality trade-offs, with the goal of exposing their performance impact and simplifying the choice of memory management methods for programmers.

CCS Concepts • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Computing methodologies** → *Parallel programming languages*

Keywords Heterogeneous systems, GPGPU, APU

1. Introduction

Copying data over the Peripheral Component Interconnect (PCI) bus is often limiting performance on discrete GPU systems. Integrated CPU-GPU systems address this problem by placing the CPU and the GPU on the same die (Fig. 1), with the added advantage of reduced area and power footprint. In integrated CPU-GPU systems, both the CPU and the GPU share the physical memory of the system. Therefore, it might appear that this tight integration allows for seamless data sharing. As we explain in this paper, this is not the case. There is still a lot of software and hardware restrictions on CPU-GPU integrated systems that forbid complete and seamless unification of memory.

Software frameworks such as CUDA and OpenCL provide a number of memory management methods to simplify programmability. For example, one method might allocate a memory area that can be seamlessly accessed from either the CPU or the GPU without the need to manually translate the pointers or copy data; another might provide a memory area that can be used for atomic accesses. These memory management methods are hidden in the runtime library and the GPU driver. Their implementation details are poorly understood, and the effects on performance remain somewhat of a mystery. For example, to enable seamless sharing of memory between the CPU and the GPU, a memory buffer could be configured uncacheable or access-protected by the driver, which means that an interrupt will be handled every time the memory is accessed. Although many of these design choices are justifiable from the standpoint of programmability, they can introduce large performance overheads.

The variety of memory management methods and poor understanding of their internals introduce the uncertainty as to which method a programmer should use, and whether reduced performance is worth the improved programmability. In this work we investigate how these different memory management methods behave in terms of performance and functionality. Our main contribution is revealing the performance trade-offs that these memory management strategies make and explaining the technical reasons for these performance effects, which required thorough analysis and some reverse-engineering effort due to lack of documentation. For our study we use an NVIDIA system with the CUDA programming framework and an AMD APU with the OpenCL 1.2, OpenCL 2.0 and HSA programming frameworks. We believe that our choice of platforms and frameworks covers a wide spectrum of current and future integrated CPU-GPU systems.

The rest of the paper provides background on GPU systems (Sec. 2), presents our evaluation of a NVIDIA/CUDA platform (Sec. 3 and 4) and of an AMD APU system with OpenCL and HSA

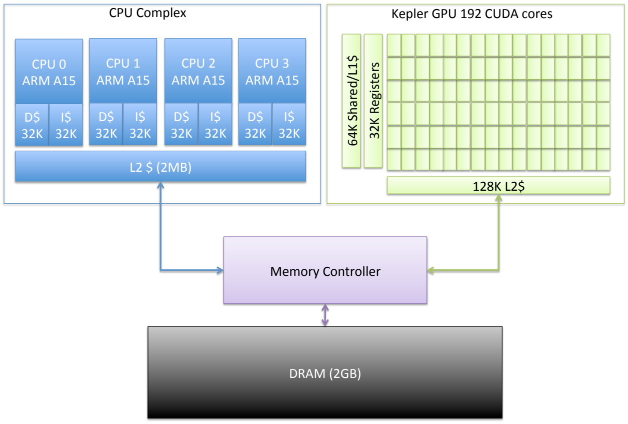

Figure 1. A high level view of the NVIDIA integrated CPU-GPU system used in this work.

ISMM'17, June 18, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5044-0/17/06...$15.00
http://dx.doi.org/10.1145/3092255.3092256

(Sec. 5 and 6). Sec. 7 discusses related work and Sec. 8 summarizes our findings.

2. Background

GPUs are massively parallel processors containing thousands of cores that aim at hiding latency by having thousands of instructions in-flight to ensure that there will always be something executing on the available cores. If, for example, a thread running on one GPU core issues a memory instruction and the data is not immediately available, the thread will be de-scheduled and a new thread will start executing immediately. On the contrary, a typical CPU consists of tens of processing cores that are more complex because they implement architectural techniques, such as deep out-of-order pipelines, branch predictors and multi-level cache hierarchies, aiming to reduce the latency of a sequential instruction stream.

Applications to be accelerated on a GPU typically have sequential and parallel regions. Sequential regions do not benefit from running on the GPU, since its specialty is massively parallel SIMT (Single Instruction Multiple Threads) code. Parallel regions can either be offloaded to the GPU, or continue running on the CPU. The portion of code that runs on the GPU is called a *kernel*. In CUDA terminology, each kernel is composed of a number of *blocks*. The blocks are further broken down into *warps*, which consist of 32 *threads* each. In other words, a GPU kernel consists of blocks of warps of threads. In the OpenCL and HSA models the blocks are called *workgroups* that are subdivided into *wavefronts*, each consisting of 64 threads or *work items*. The decision of how to best distribute the parallel regions between the CPU and the GPU can be very complex as it depends on multiple factors such as the offloading overhead, the cost of data transfer, the acceleration potential on the GPU, and memory requirements.

On current systems, interacting with the GPU occurs through a programming framework, which typically consists of a programming language (or an extension to a language), a runtime, and a driver. The two most widely used programming frameworks are CUDA and OpenCL. Furthermore, AMD introduced HSA, a lower-level framework that utilizes hardware features such as shared pageable memory and user-space work queues [7]. All frameworks have APIs that allow specifying how to allocate memory and how to launch kernels on the GPU.

At the hardware level, GPUs schedule warps (or wavefronts) rather than individual threads for execution. GPUs follow a single-instruction multiple-thread (SIMT) paradigm. Hence, a group of 32 (or 64) threads execute instructions in lockstep. Divergence between the individual threads is allowed and tracked by the GPU so that possible re-convergence can be resumed at a later point in execution. This type of divergence is referred to as *thread divergence*. There is some overhead with divergence, so it is best for all threads to follow the same execution path. Another type of divergence is referred to as *memory divergence*. It is best for the threads within a warp to access consecutive words in memory. This way of accessing memory allows the GPU to *coalesce* these memory accesses into a single access which loads or stores all of the consecutively accessed data. If, however, each thread tries to access a random memory location, the memory unit of the GPU will generate multiple memory requests and performance can degrade.

Although this study targets platforms that integrate CPUs and GPUs, it is applicable to any system in which two types of heterogeneous units need to communicate with each other. Instead of relying on simulators, we chose to present our analysis on real systems employing different software models and implementing different CPU-GPU coherency schemes. Future systems ranging from mobile devices to exascale computing are expected to house such integrated chips to accommodate a variety of applications [18].

3. The NVIDIA/CUDA System

The NVIDIA system used in our study includes a Kepler GPU with 192 cores and an ARM quad-core CPU; further details are provided in Section 4. There is no hardware coherency between the CPU and the GPU caches; software must take care of keeping the data consistent. The programming framework used on this platform is CUDA. We discover that memory management methods offered by CUDA sometimes apply restrictions on how the data is cached and whether it can be accessed concurrently by the CPU and the GPU. We now explain how these methods work.

`cudaMalloc`: The first and most commonly used method of managing memory is to allocate it on the host (CPU) using `malloc()` and on the device (GPU) using `cudaMalloc()`. This method does not utilize shared virtual memory, so the CPU uses a different virtual address space than the GPU. Therefore, two different buffers need to be allocated with different pointers pointing to each buffer. Data allocated on the host cannot be accessed from the device and vice versa, but it can be copied back and forth between the two buffers using `cudaMemcpy()`; the DMA copy engine will perform the copying from one location on the unified physical memory to another. It is up to the programmer to explicitly invoke `cudaMemcpy()` when the data is needed. As a result, the resulting code may end up cluttered with allocations and copy commands and can be difficult to debug and maintain. We will refer to this allocation scheme throughout the paper as the *default* scheme.

`cudaHostAlloc`: On integrated systems, data allocated with `cudaHostAlloc()` is placed in an area that is accessible from both the CPU and the GPU without the need to perform explicit memory copies. From our experiments, we found that the data is allocated on a device-mapped region `/dev/nvmap`; `nvmap` is the GPU memory management driver on integrated NVIDIA systems. This region physically resides in the system's main memory, shared by the CPU and the GPU, but has different access properties than other virtual regions. Our investigations revealed that this region is configured uncacheable on the CPU and the GPU. As a result, any read or write from or to that memory region goes directly to main memory, bypassing caches. We will refer to this allocation scheme as `hostalloc`.

`cudaMallocManaged`: *Managed* memory was introduced in CUDA 6.0 to eliminate the need for explicit data copying between the CPU and the GPU by providing a unified shared address space across the CPU and GPU. This unified address space allows seamless pointer sharing across both processing units. A managed memory buffer is allocated via `cudaMallocManaged()`, and similarly to `hostalloc`, there is no need to explicitly copy data between the host and device. Furthermore, just like `hostalloc`, the data is mapped to a `/dev/nvmap` region. In contrast to `hostalloc`, data is cached in CPU caches, and in the GPU L2 caches. The price of unrestricted caching is that the memory cannot be accessed concurrently by the CPU and the GPU. Once the host launches a GPU kernel, it is not allowed to access data in the same managed region as the one accessed by the GPU until the kernel completes; an attempt to do so will cause a SEGFAULT. If the host wishes to access a managed area after the GPU kernel completes, it must issue an explicit CUDA API synchronization call, which flushes caches to ensure a consistent view of the data. This restriction is not an issue for programs where the CPU and the GPU execute sequentially, but can be a limitation for programs where the CPU and the GPU concurrently access data in the same allocated region (even if the actual data is disjoint). We will refer to this allocation scheme as `managed`.

These memory management schemes fall into different parts of the spectrum of the trade-off between the functionality and ease of use and restrictions on memory accesses (which leads to performance degradation, as we will show later). While the default scheme places no restrictions on memory accesses, it is the most

```
// Initialize context
cudaFree(0);
// Inform Linux that next allocation will be sharedalloc
gpu_hook(INIT);
// Allocate data using the same syntax as cudaHostAlloc
cudaHostAlloc(data, size, flags);
// Work on data using CPU
cpu_work(data);
// Flush CPU caches to push data to main memory
gpu_hook(FLUSH_CPU);
// Launch kernel to manipulate data on the GPU
kernel<<<blocks, threads>>>(data, ...);
// Wait for kernel to finish
cudaDeviceSynchronize();
// Flush GPU caches to push data to main memory
gpu_hook(FLUSH_GPU);
```

Figure 2. `sharedalloc` usage example

difficult to use for the programmer, because a pointer allocated on the CPU cannot be accessed on the GPU (and *vice versa*), so data must be manually copied if sharing between the devices is desired. The `managed` scheme obviates the need for copying, but prevents concurrent access to data by the CPU and the GPU, so that caches can be flushed on kernel invocation boundary. This is a coarse consistency management method, which ensures that the programmer sees consistent copies of shared data at the expense of restrictions when the memory can be accessed. Finally, `hostalloc` makes another choice with respect to consistency management: it does not restrict concurrent access, but “shields” the programmer from accidentally reading inconsistent data by disabling caching.

We observe that there is another point in this spectrum that is not covered by the available memory management methods: allowing the programmer seamless memory sharing without concurrency or caching restrictions, but requiring them to manually flush caches when consistency management is required. To explore this point, we introduce a new configuration scheme that we call `sharedalloc`.

`sharedalloc`: Like `hostalloc`, it allows concurrent accesses to data from the CPU and the GPU, but at the same time allows caching, like the `managed` method. To implement `sharedalloc` we modified the Tegra Linux kernel code to introduce a new system call, `gpu_hook()`, that hooks into the GPU driver and allows us to control caching and explicitly flush caches to maintain data consistency.

We control caching by setting two flags that are passed as arguments to `ioctl()`. These flags make data cacheable on the CPU and the GPU. We discovered how to use these flags by tracing calls to `ioctl()` and examining the (limited) publicly available source code from NVIDIA. Once these flags are set, we can simply use `cudaHostAlloc()` to allocate memory without caching restrictions.

In our implementation of `sharedalloc`, the programmer must manually flush the CPU/GPU caches on kernel launch boundaries by invoking `gpu_hook()`. However, in programs adhering to a data-race-free synchronization model, the invocation of this function could be embedded into synchronization functions. Figure 2 shows an example using `sharedalloc`. The difference between using `sharedalloc` and `hostalloc` is the additional `gpu_hook()` system calls.

A summary of caching and concurrency trade-offs for the different memory allocation schemes is shown in Table 1.

Table 1. Summary of allocation schemes

Allocation scheme	CPU caching	GPU caching	CPU-GPU concurrency
default	cached	cached	difficult + copy overhead
hostalloc	not cached	not cached	easy
managed	cached	cached	not possible
sharedalloc	cached	cached	easy

4. Evaluation of the NVIDIA/CUDA System

4.1 Hardware/Software

We performed our experiments on the NVIDIA Jetson TK1 board, a K1 SoC integrating a CPU and a GPU. The CPU is a "4-Plus-1" 2.32GHz ARM quad-core Cortex-A15, and the GPU is a Kepler "GK20a" GPU with 192 SM3.2 CUDA cores. There is 2GB of DDR3L DRAM running at 933MHz with 64-bit data width. All 192 CUDA cores share 32K registers, 64KB L1 cache/shared memory, and a 128KB L2 cache. Global memory is normally cached only in L2, unless the application is compiled with a flag enabling caching in L1. The cache line size is 32 bytes. The system software is Linux For Tegra R21.4 with CUDA 6.5.

In all of our experiments, we set the board in the "performance" mode and we maximize the CPU and GPU clock rates. We also maximize the system and memory bus clocks.

4.2 Benchmarks and Experiments

We run four types of tests to analyze the memory management behaviour on this platform:

Bandwidth Experiments: We run microbenchmarks measuring read and write bandwidth from the CPU and the GPU under various memory management schemes and highlight the performance differences.

Rodinia Benchmarks: We port a number of applications from the Rodinia benchmark suite [5] to use the memory management methods described in the previous section. The original implementation of Rodinia uses the default method.

GPU Caching Behaviour: We explore the behaviour of caching on the GPU using microbenchmarks.

Concurrent Workload Experiments: We analyze the performance of applications where the CPU and GPU concurrently access portions of data from the same allocated region. To that end, we wrote a microbenchmark and modified four applications from Rodinia to perform concurrent work by the CPU and the GPU.

4.2.1 Bandwidth Experiments

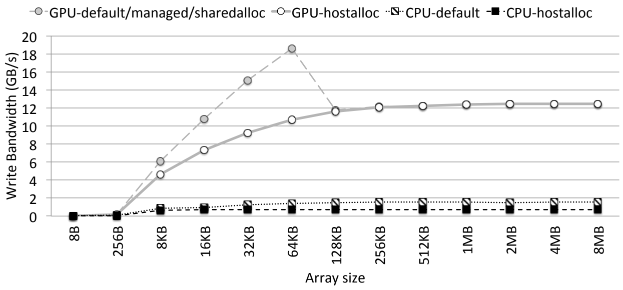

Figure 3. Write bandwidth under different memory management methods.

To demonstrate the impact of caching restrictions on performance, we measured the read and write memory bandwidth achieved by either the CPU or the GPU with the different memory management methods. For the CPU tests, a single CPU thread sequentially reads or writes a data array, which we vary in size. For the GPU tests, we launch many threads to read or write the array in parallel, each thread reading its own array entry. We vary the size of the array, and the number of threads varies with it. For example, for an 8MB array, we have 2^{20} of 8-byte entries, and so we launch that many GPU threads to access these entries. We perform 10 million iterations over the array in total, so caching will be beneficial where it is used and when the array fits. The reported bandwidth is the total number of accessed bytes divided by the run time.

Figure 3 shows the write bandwidth. We omit the read bandwidth numbers, because they are almost identical. The legend reports the device performing the access and the memory management method. For example, the line labeled *GPU-hostalloc* refers to the GPU accessing a `hostalloc`'ed region; the line labelled *CPU-hostalloc* refers to the same region accessed by the CPU. (The CPU and GPU tests are run, separately, not concurrently.) We show a single line labeled *GPU-default/managed/sharedalloc*, because the performance of these memory management methods on the GPU is the same in this simple experiment, as expected; neither of them disables caching.

We observe that caching restrictions imposed by the `hostalloc` method have a significant performance impact on both the CPU and the GPU. The CPU loses more than half the bandwidth when caching is disabled. For the GPU, the `managed`, `default` and `sharedalloc` scheme are able to exceed the available bandwidth of roughly 12GB/s when the data fits into the cache (the array size is smaller than 128KB), while `hostalloc` runs much slower due to the lack of caching. When the data no longer fits into the cache, all systems become limited by the hardware throughput and deliver the same performance (notice the sharp drop at 128KB).

The `managed`, default, and `sharedalloc` deliver the highest read/write bandwidth. The restriction on the CPU/GPU concurrency imposed by `managed` is not evident from this very simple test. However, many recent studies demonstrated benefits of truly concurrent execution by the CPU and the GPU [4, 6, 11, 12, 16]; for these applications, the restriction on CPU/GPU concurrency may be a limitation. We demonstrate a potential effect of these limitations with *Concurrent* experiments later in this section.

4.2.2 Rodinia Benchmarks

Figure 4 shows the running time under different memory management methods, normalized to default for Rodinia benchmarks. Figure 4a shows the overall runtime, Figure 4b shows the runtime of the GPU kernel only. In addition to the memory management schemes described in the previous section we also experiment with `hostalloc-cache`. This is an intermediate option between `hostalloc` and `sharedalloc` that enables caching on the CPU, but not on the GPU.

Due to caching restrictions, `hostalloc` causes performance degradation for most benchmarks; *lud* suffers the largest slowdown of $9.5\times$. The slowdown can be attributed to CPU caching, GPU caching, or both. We can deduce which effect is dominant by comparing `hostalloc-cache`, which only enables CPU caching, to `sharedalloc`, which also lifts restrictions on GPU caching. Additionally, we can compare Figures 4a and 4b, which separate the overall runtime (affected by CPU and GPU caching) from the runtime of the GPU kernel (affected by GPU caching only). For example, the `hostalloc` version of *kmeans* does not suffer any slowdown on the GPU side, but its overall runtime is affected, leading us to conclude that the culprit is lack of caching on the CPU side. *Particlefinder*, *lud* and streamcluster suffer from both CPU and GPU caching restrictions. For many benchmarks, the lack of GPU caching is the main source of overhead and this is what the kernel runtime figure shows. We also verified that the actual costs of the allocation calls are negligible; it is the cost of accesses that have the major impact on performance.

The `managed` scheme performs closely to the default, because both use identical caching configurations. The exception is *dwt2d*, which actually uses `hostalloc`, and not default, in the baseline implementation. `Managed` may result in worse performance than default in scenarios where the cost of cache flushing through the synchronization call is high; with the default scheme the data is copied by the programmer, so they have the option of cherry-picking what to copy, while the `managed` scheme will always flush caches on kernel launches to maintain consistency. However, for the applications presented here, the costs of the infrequent cache flushing and the costs of copying data between the CPU and the GPU was measured to be very small across the board (not having to go through the PCIe bus certainly helps), so this effect is not obvious. We will show later in the paper that for concurrent CPU-GPU applications the frequent cache flushing can have high overheads.

Summary: For applications without concurrent CPU/GPU accesses to the same data, the `managed` method provides equivalent performance to default with substantially better programmability. `Hostalloc` takes a toll on performance because it disables caching, and `sharedalloc` does not deliver any performance benefits over `managed` because the applications analyzed here do not have any CPU/GPU concurrency.

4.2.3 GPU Caching Behaviour

In this section, we attempt to shed light on GPU caching restrictions implemented by `hostalloc`. The details of its behaviour are not publicly available, so we infer them from simple microbenchmarks.

Figure 5 shows the performance of a GPU kernel, launched with an increasing number of threads, that accesses an array managed either by `hostalloc` or `managed`. Each entry in the array is either padded to ensure that each entry is 32 bytes (the size of the cache line in the L2 cache) or not padded. Figure 5a shows a version of the kernel where each thread accesses its respective entry in the array (`data[tid]++`); Figure 5b shows a version where all threads access the first entry (`data[0]++`).

This experimental design enables us to observe how performance changes as more and more threads access the same cache line. In the padded version where each thread accesses its own array element, each thread also accesses its own cacheline. In the unpadded version of the same experiment, several threads access the same cache line. In the experiment where all threads access the same element, the cache line is shared by all threads.

We observe that as the number of threads accessing the same L2 cache line increases, performance gets worse with `hostalloc`, but is unaffected when the `managed` scheme is used. `Hostalloc` aims for a stricter consistency model between the GPU and the CPU, and to that end it restricts caching on the GPU by disabling L2 caching altogether. This explains why writing to the same cache line slows down the execution: there is more competition so writing to the same cache line causes significant slowdown since every write has to go to main memory to ensure consistency. Hence, `hostalloc` is more appropriate for code that performs fine-grained data sharing. We explore this implication in the next section.

Figure 6 shows the results when data is cached in L1 caches in addition to L2. By default, global data are not cached in the L1 cache on Kepler class GPUs. However, this can be overridden using a compiler flag. The effects are less apparent when enabling L1 caching; however as the data set size increases and spills to the L2 cache, the behavior becomes similar to the previous case. Furthermore, enabling the caching in L1 for global data can negatively affect applications which rely on shared (local) memory. The L1 cache is used for caching shared memory by default, so having global memory cached as well can reduce the effective usage of this cache.

4.2.4 Concurrent Workload Experiments

In this section we demonstrate the utility of `sharedalloc` and `hostalloc` for workloads that utilize both the CPU and the GPU concurrently. We use a microbenchmark, and modify four Rodinia applications to have the CPU and GPU collaboratively work on allocated data.

In the microbenchmark, we allocate a 128 MB integer array (32M entries) using the different allocation methods. The array is

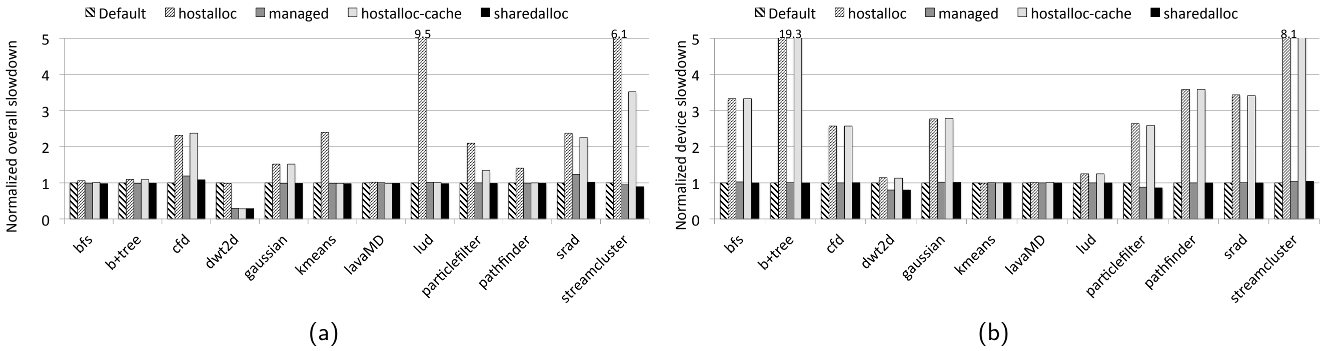

Figure 4. Performance of Rodinia applications with the different global memory allocation methods. (a) Overall. (b) Kernel-only.

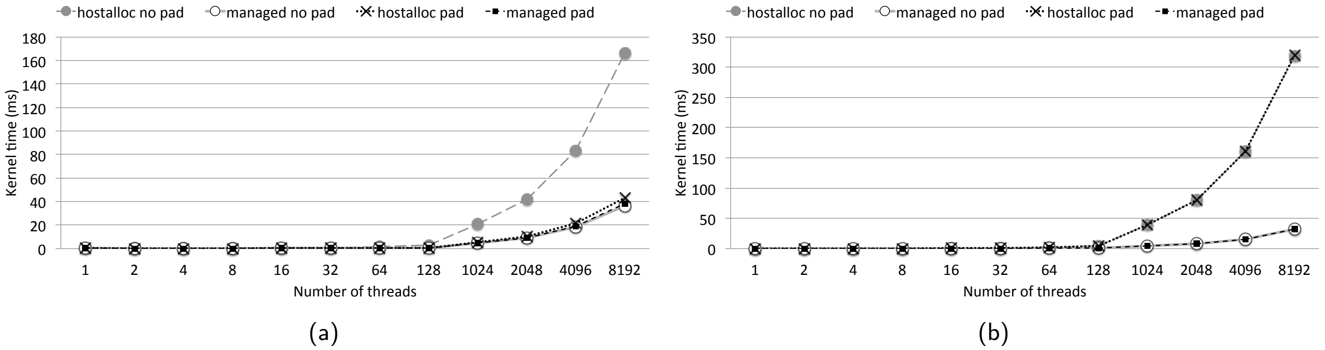

Figure 5. Accesses are cached in L2. (a) Each GPU thread accesses its respective array index. (b) All GPU threads access the same array index.

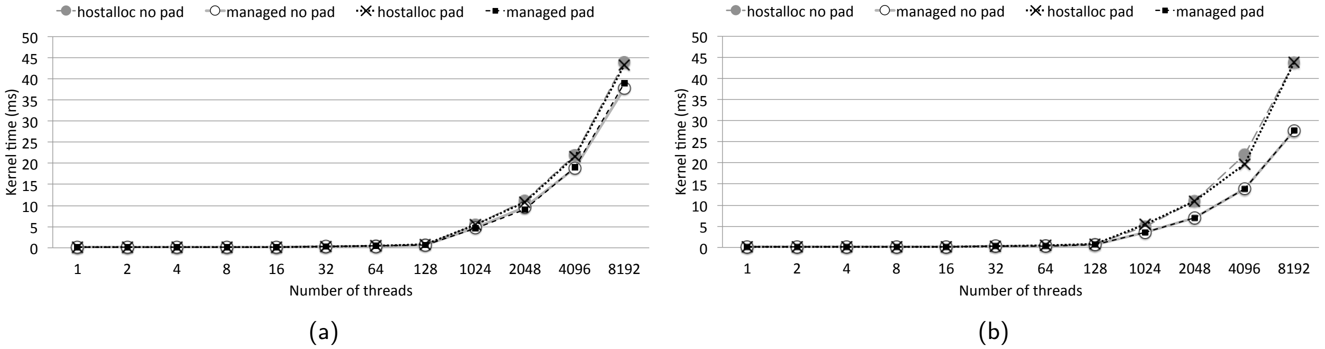

Figure 6. Accesses are cached in L1. (a) Each GPU thread accesses its respective array index. (b) All GPU threads access the same array index.

accessed 100 times in total. In the first iteration, the GPU increments the entries of the first half of the array and the CPU increments the entries of the second half. In the second iteration, the CPU works on the first half, and the GPU works on the second. The pattern continues until all iterations are completed. Each iteration is executed concurrently by the CPU and the GPU. However, when the data is allocated using `managed`, the CUDA runtime does not allow the GPU and the CPU to access the allocated region concurrently. One way to overcome this limitation is to split the allocated region into multiple allocations, so that the CPU and the GPU work on different allocations at any given time. While this would be feasible for very simple programs where the boundaries of data division are known *a priori*, for more complicated scenarios where shared data is arbitrarily split across threads, such a solution would be substantially more complicated to implement. Therefore, for the `managed` case, we execute the GPU and the CPU portions in lockstep: i.e., the GPU kernel finishes execution, then the CPU code begins to run. Under default, the data has to be copied between the CPU and GPU in-between iterations since each iteration depends on the values from the previous iteration.

The default applications in Rodinia do not impose much interactions and sharing between threads running on the CPU and GPU. This has been explored in some details in a recent paper [8]. So, we modified a number of applications to introduce such interactions. In the modified Rodinia applications, the kernel works on half of the data by launching half as many GPU threads as in the original. The second half of the data is processed by CPU threads (via OpenMP) concurrently with the GPU threads, except for the `managed` scheme where they run in lockstep.

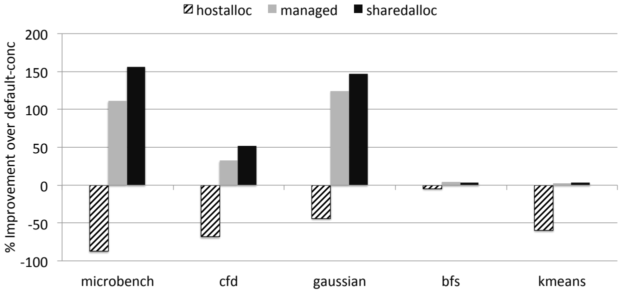

Figure 7. CPU-GPU concurrent benchmarks. Both the CPU and GPU work on independent data. Default-conc (y-axis) is the default implementation of the concurrent version of the benchmark where data has to be explicitly copied between the CPU/GPU

Figure 7 shows the performance under all memory management methods compared to default. In terms of programmability, the default version is the most cumbersome since data accesses can be non-trivial and figuring out which portion of the data to copy back and forth may be difficult. For `managed`, `hostalloc`, and `sharedalloc`, the data is visible on both the CPU and GPU without the need of explicit copies. `sharedalloc` improves performance by more than 156% over the default method for the microbenchmark, and over 45% over the `managed` scheme. For `hostalloc`, the main cause of the performance degradation is restricted caching. Although concurrency is possible with `hostalloc`, the overhead of not caching data on the CPU and GPU far exceeds concurrency benefits. Comparing `sharedalloc` and `managed`, `sharedalloc` is superior to `managed` because the CPU and the GPU are able to work concurrently. *Cfd*, and *gaussian* gain 25-50% performance relative to `managed`; *bfs* and *kmeans* gain a modest 4%.

The applications presented in Figure 7 perform coarse-grained data sharing. That is, threads run a computation on disjoint sets of data and any data exchanges between threads (i.e., where one thread reads the data that another thread wrote) are infrequent. In the following experiment we explore a fine-grained sharing scenario and demonstrate the behaviors of the `hostalloc` and `sharedalloc` schemes in this case.

To support a realistic scenario, we implemented a locking algorithm based on the Peterson's lock [14] with some additions to account for the specifics of the GPU execution model. The algorithm uses three locks: a CPU-local lock (a TAS lock), a GPU-local lock (a TAS lock), and a shared Peterson's lock. Threads running on the CPU will first compete for the CPU-local lock; threads running on the GPU will first compete for the GPU-local lock. The thread that acquired each lock is called the *leader* thread. The leader threads on the CPU and the GPU will then compete to acquire the shared Peterson's lock. This arrangement is needed to avoid a deadlock on the GPU, which would occur if multiple threads within the warp contended for the shared lock.

Once the lock is acquired, the thread in the critical section accesses a shared data region, allocated either with `hostalloc` or `sharedalloc`. Following the critical section, the thread performs accesses to a non-shared region. We vary the number of accesses in the non-critical section relative to the critical section. Figure 8 illustrates the fine-grained scenario.

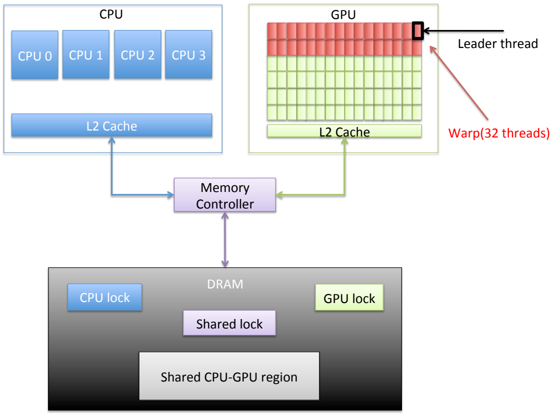

Figure 8. The locking scheme used in fine-grained sharing

While the shared region itself can be allocated using `hostalloc` or `sharedalloc`, the shared lock must be always allocated using `hostalloc`, because it requires strict consistency between the CPU and the GPU.

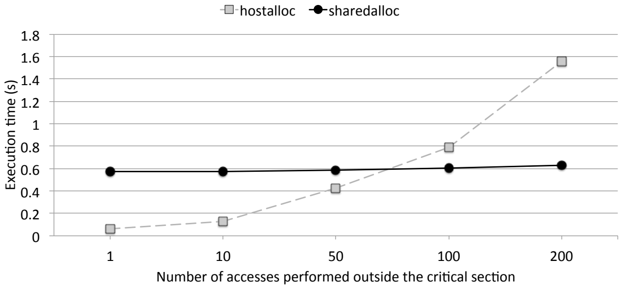

Figure 9. Fine-grained sharing benchmark showing the performance of `hostalloc` and `sharedalloc`

Figure 9 shows the results. When the number of accesses outside the critical section is small, `hostalloc` is superior to `sharedalloc`. Since `hostalloc` does not cache the data, we don't need to flush the caches after we release the shared lock since the latest copy of the data is already in main memory. However, for `sharedalloc`, we need to flush the caches after releasing the lock. `sharedalloc` flushes the entire cache, not discriminating the data that was actually modified in the critical section, causing higher overhead. However, as the number of accesses outside the critical section increases (the critical section becomes relatively less important), this advantage of `hostalloc` evaporates, and the caching advantage of `sharedalloc` begins paying off as better performance.

Summary: The `managed` memory management method is a significant improvement in programmability relative to default because it enables accessing the same memory on the CPU and the GPU without explicit copy statements. However, the downside is that the CPU and the GPU are unable to access the same allocated region concurrently even if they access disjoint data. The `hostalloc` method does not impose the concurrency restriction, but at the expense of disabling caching on the CPU and the GPU. `sharedalloc` neither restricts caching nor prevents concurrency, but needs to flush the caches when data written on one device must be made visible on another. Cache flushing calls could be embedded into the CUDA API so there would be no negative impact on programmability. The downside of `sharedalloc` is in indiscriminate flushing of the entire cache when only a small portion of the data needs to be synchronized. This becomes evident in fine-grained sharing scenarios, where `hostalloc` might be advantageous depending on the size of the shared buffer and the number of accesses performed in the critical section.

5. AMD Kaveri APU

In this section, we turn our focus to another platform that implements a unified memory system. We evaluate an AMD A10-7850K Kaveri APU. There are several programming frameworks available for the AMD APU. We briefly describe the programming frameworks used on the AMD APU and then evaluate their performance.

5.1 OpenCL 1.2 and OpenCL 2.0

Memory management in OpenCL 1.2 is similar in programmability to the default scheme in CUDA. The CPU and the GPU do not share the same virtual address space. Therefore memory buffers have to be explicitly managed and communicated between the two devices. A pointer on the CPU is inaccessible on the GPU and vice versa.

OpenCL 2.0 makes memory management considerably simpler by introducing Shared Virtual Memory (SVM). With SVM, the CPU and the GPU see a unified virtual address space, similarly to CUDA's `managed` method.

OpenCL 2.0 allows allocating a memory buffer either as *coarse-grained* or *fine-grained*. Coarse-grained buffers are allocated for access primarily on a single device (the CPU or the GPU), while fine-grained buffers are suitable for fine-grained sharing by the two devices. Coarse-grained buffers need to be mapped and unmapped using API calls, which act as synchronization points to ensure proper address translation and consistency when one type of device accesses the data after it might have been written by another. Fine-grained buffers do not need to be mapped or unmapped and are guaranteed to be consistent at kernel launch and kernel completion. Fine-grained buffers can also be declared with an atomic flag; in that case, they would be kept consistent between the CPU and the GPU even during kernel execution with the proper use of synchronization primitives.

5.2 HSA: Heterogeneous System Architecture

The HSA framework is a low-level software layer that gives programmers finer control over the underlying integrated hardware. Similar to OpenCL 2.0, HSA employs a unified virtual address space, where pointers can be seamlessly accessed from both the CPU and GPU.

Using a special compiler, CL Offline Compiler [2], unmodified OpenCL kernels can be compiled to HSAIL (HSA Intermediate Language) and BRIG (a binary representation of HSAIL) which can then be loaded and run by the HSA runtime. However, the host-side source has to be modified, as OpenCL APIs are different from those in HSA. The main modifications are related to the initialization of HSA and in launching the kernels.

On the Kaveri system, there are three different memory buses, named *GARLIC*, *ONION* and *ONION+* (see Figure 10), and memory accesses on the GPU may take one of these three paths [15]. If coherence with the CPU is not required as in the case for coarse-grained buffers, memory requests take the GARLIC path which leads directly to main memory without communication with the CPU. If kernel-granularity coherence with the CPU is required, as would be the case with fine-grained buffers, memory requests take the ONION path. If instruction-granularity coherence is required (fine-grained buffers with atomics), the requests take the ONION+ path. This path allows the bypassing of the L1/L2 caches from the GPU side and snooping to invalidate the CPU caches [15]. For the workloads that we used in the paper, both fine-grained and fine-grained-with-atomics memory allocations had identical performance, so we only present the analysis for fine-grained. Fine-grained-with-atomics need to be used along with synchronization primitives to ensure consistency. This would be similar to the fine-grained concurrent setting on the NVIDIA system. We do not present our evaluation of such concurrent scenarios on the AMD platform in this paper; however, performance trends are similar.

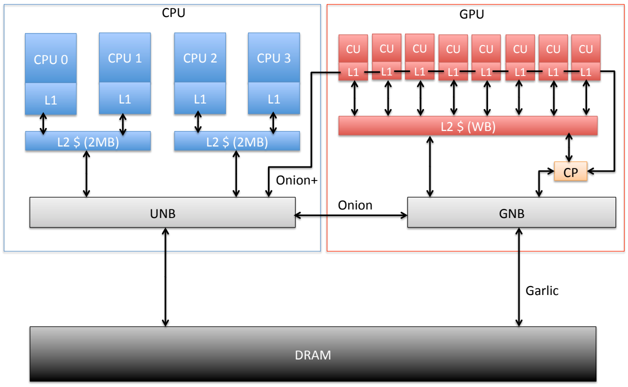

Figure 10. Memory system architecture of AMD Kaveri. Figure reproduced from [15]. CP: Command Processor, GNB: Garlic North Bridge, UNB: Unified North Bridge.

6. Evaluation of the AMD APU System

6.1 Hardware/Software

We perform our OpenCL and HSA experiments on an AMD A10-7850K Kaveri Quad-Core 3.7 GHz system. Each pair of CPU cores share a 2MB L2 cache. The integrated GPU is an AMD Radeon R7 720 MHz Sea Islands family (GCN 1.1). It has eight Compute Units (CU). The GPU L2 cache is implemented as a physically partitioned and distributed cache among the CUs. The total L2 cache size is 512 KB with each cache slice 64-128 KB [1]. The OpenCL software stack uses the proprietary FGLRX driver, while the HSA framework uses the Radeon Open Compute Platform (ROCm).

6.2 Benchmarks and Experiments

First, we compare the performance of the memory management methods available in OpenCL 1.2, OpenCL 2.0, and HSA using microbenchmarks. Then, we explore the caching behaviour of GPU accesses. Finally, we study the performance of Rodinia benchmarks, which we ported to OpenCL 2.0 and HSA.

The AMD platform offers us more data points to compare relative to the NVIDIA platform, because there are three programming frameworks and three types of memory allocation schemes (coarse-grained, fine-grained and fine-grained with atomics). To make the results easier to read and qualitatively comparable to the other system, we focus on the following combinations of the frameworks and the memory management methods.

1. *OpenCL 1.2, coarse-grained*: The memory is allocated using `clCreateBuffer` with the `CL_MEM_READ_WRITE` flag. This method is qualitatively similar to the default scheme on the NVIDIA/CUDA platform. It is the most cumbersome in terms of programmability and provides no data consistency between the devices and no sharing of virtual addresses.
2. *OpenCL 2.0, SVM, fine-grained*: The memory is allocated using `clSVMAlloc` specifying a fine-grained buffer. SVM is similar

to CUDA's managed in that it provides a unified virtual address space for the CPU and the GPU. Fine-grained buffers are kept consistent between the two devices at kernel invocation boundaries at the expense of performance, similarly to CUDA managed. Unlike CUDA managed, this configuration does not restrict concurrent access by the CPU and the GPU; however there is no guarantee on data consistency if the data is accessed concurrently by both devices[1].

3. *HSA, fine-grained*: This is similar to the above method but utilizes the HSA framework. The memory is allocated using the default system malloc call, which is a major portability advantage for HSA.

From now on, we refer to the three chosen configurations simply as OpenCL 1.2, OpenCL 2.0 and HSA, for brevity.

6.2.1 Bandwidth Experiments

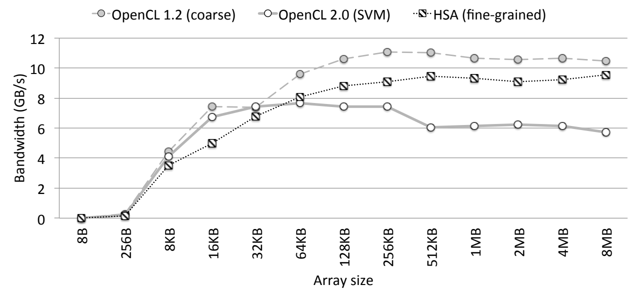

Figure 11. Write memory bandwidth for OpenCL 1.2 (coarse), OpenCL 2.0 (SVM) and HSA (fine-grained).

We measure the read and write bandwidth under the three chosen configurations using the same test as in Section 4: we vary the size of the allocated array, and each GPU thread reads or writes its own dedicated 8-byte entry. The number of threads is, therefore, the same as the number of array entries. We verified both experimentally and by reading the code and documentation that none of the memory management methods on the AMD system disable CPU caching or affect memory access latency on the CPU, so we only report the results of experiments on the GPU.

Figure 11 shows the results for the write bandwidth test. (We do not present the read bandwidth results, because they were almost identical to the write test.) We observe that the three different configurations deliver vastly different performance. For large buffer sizes, OpenCL 1.2 delivers the highest bandwidth, limited only by the speed of the hardware. HSA delivers roughly 10% lower throughput, and OpenCL 2.0 is almost 2× worse. For small buffer sizes, the performance differences are smaller, with HSA slightly trailing the OpenCL versions.

HSA enhances programmability over OpenCL 1.2 in two main ways: (1) HSA provides a unified virtual address space between the CPU and the GPU, (2) HSA configures the memory to be coherent between the CPU and the GPU. Both of these programmer-friendly features come at a cost. Providing a unified virtual address space requires the GPU to perform address translation using CPU page tables. As described in a recent article that includes several co-authors from AMD [17], the GPU on the Kaveri system has a multi-level TLB hierarchy that is accessed for address translation. Upon a TLB miss, a translation request is dispatched to the IOMMU (Input/Output Memeory Managament Unit) that has its own TLB hierarchy and walks CPU page tables upon a TLB miss. If the translation is not found in the page table, IOMMU dispatches a page fault handling request to the CPU. The address translation process has overhead; and this is part of the reason why HSA delivers lower throughput for large array sizes.

The second cause of overhead is the cost of maintaining coherency between the CPU and GPU. We couldn't confirm that the platform implements hardware features enabling coherency, so this cost is either the actual cost of hardware coherency or the cost of the software driver to functionally emulate hardware coherency. We refer to this cost as *functional hardware coherency*. To isolate the overhead of coherency from the address translation overhead, we perform a simple experiment where multiple GPU threads concurrently update a single memory location. This way, the overhead of address translation is negligible. We compare two configurations: (1) the memory is allocated (with HSA) as a fine-grained buffer, just like in Figure 11, and (2) the memory is allocated (with HSA) as a *coarse-grained* buffer. While the first configuration enables functional hardware coherence, the second one does not. Figure 12 shows the results, and demonstrates that functional hardware coherence, either implemented in software or hardware, does incur 15-25% overhead, which increases with the number of concurrent accesses.

Getting back to Figure 11, for small buffer sizes, the address translation overhead is small, so HSA trails the OpenCL implementations because of the functional hardware cache coherency overhead. With larger buffer sizes address translation overhead becomes dominant, so HSA benefits from the the functional hardware support relative to OpenCL 2.0 (which handles address translation in software), but loses to OpenCL 1.2, which provides no unified virtual address space and this incurs a smaller address translation overhead.

Table 2 presents the number of data TLB loads and the number of interrupts, both measured on the CPU-side, for the experiment from figure 11 for the 1MB array size. Indeed, OpenCL 1.2 performs a lot fewer TLB accesses than OpenCL 2.0 and HSA; that is because it uses a coarse-grained allocation. OpenCL 2.0 and HSA perform a similar number of TLB loads if memory allocation is configured as coarse-grained. What we find surprising is that OpenCL 1.2 generates the most interrupts and yet delivers the best performance. That could be explained by a high cost of address translation on the Kaveri platform (dissected in details in a recent work [17]); OpenCL 1.2 wins because it performs fewer of these expensive operations on a coarse-grained memory buffer.

Table 2. Data TLB loads and Interrupts

Metric	OCL 1.2 (coarse)	OCL 2.0 (SVM)	HSA (fine)
dTLB loads (billion)	17.7	28.2	25.9
K interrupts/s	21.1	14.7	7.6

OpenCL 2.0 delivers the worst performance of all, which is attributed to the runtime/driver implementation. OpenCL 2.0 SVM implementation provides the convenience of a unified virtual address space and data consistency in software (runtime/driver). Profiling confirmed that a significant portion of the time spent by OpenCL 2.0 workloads is attributed to the driver. The GPU driver is responsible for producing a unified virtual address space on systems with no associated hardware support. The driver allocates a GPU page table for a given range of CPU virtual addresses and performs address translation. These driver activities incur performance overhead, which could explain poor performance of OpenCL 2.0. Furthermore, OpenCL 2.0 flushes caches once the kernel has finished running to ensure that the CPU sees consistent data. That, and other software overhead is responsible for lower performance with this framework. Table 2 shows that OpenCL 2.0 performs more TLB loads than both OpenCL 1.2 and HSA, and generates almost twice as many

[1] For truly concurrent and consistent accesses the programmer is expected to use fine-grained buffers with atomics along with synchronization primitives.

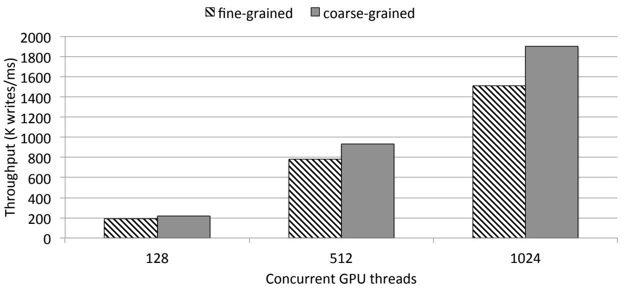

Figure 12. Write throughput as multiple threads concurrently update a single memory location allocated as *coarse* (non-coherent) or *fine* (coherent) with HSA.

interrupts as HSA, further pointing to driver overhead in this framework. Unfortunately, the OpenCL stack on our platform is proprietary, so we could not find further details on software overhead by examining the source code.

Summary: Similarly to our analysis of the NVIDIA platform, we observe that programmability comes at a cost. Unified virtual address space requires performing address translation. Enabling the CPU and the GPU to see consistent copies of data incurs overhead. OpenCL 2.0 pays a substantial cost for supporting these features whereas HSA delivers them at a substantially lower overhead.

6.2.2 Exploring GPU Caching Behaviour

In order to further understand the performance differences related to GPU caching, similarly to our experiments with the NVIDIA platform we varied the padding of array entries written by the threads. Unlike our NVIDIA experiments, we show the results for a single array size (as opposed to showing the performance across array sizes), to highlight interesting performance effects that are unique to the AMD platform. In this experiment, each thread writes its own eight-byte array entry in a 64MB array. The entry is either not padded, padded to 32 bytes, or to 64 bytes. The size of the L2 cache line is 64 bytes, so we vary how many cache lines are accessed. Figure 13a shows the results. Unlike the NVIDIA platform, the "no padding" case has much better performance across the board. The reason is that the GPU can utilize memory coalescing to read/write each of the eight array entries with a single memory access. This effect was not obvious on the NVIDIA platform, but is significant here.

Once the entries are padded to 32 bytes, performance deteriorates for HSA and OpenCL 2.0 because the GPU can no longer coalesce memory accesses since each cache line now contains only two entries. The second reason becomes apparent if we compare the OpenCL 2.0 bars in the figure. Except for the "no padding" case which utilizes coalescing, whenever the cache line contains more than one entry that might be accessed concurrently by more than one GPU thread, performance drops drastically especially for OpenCL 2.0 (and to some extent for HSA).

We believe that the observed caching behaviour is similar to the cache behaviour we noticed with `hostalloc` memory on the NVIDIA platform. We do not see these effects with OpenCL 1.2 since the buffer is not shared between the CPU and GPU. With OpenCL 2.0, this software-triggered cache behaviour maintains the consistency of SVM buffers and hence the drastic drop in performance. HSA also seems to experience similar behaviour but to a lesser extent. As discussed earlier, HSA utilizes some hardware features which eliminate some software coherency overheads. However, we notice that the 64-byte padding case seems to perform worse than OpenCL 2.0. As the number of cache lines used increases (which is the case with 64-byte array entries), any hardware mechanism would be inferior to simply flushing the cache.

Figure 13b further demonstrates the case of performance deterioration for OpenCL 2.0 whenever coalescing cannot be utilized and the cacheline contains multiple accessed entries. Since only even-numbered threads will write to the array entries, the "no padding" case means that the cacheline contains 4 accessed entries, and hence the huge performance drop for OpenCL 2.0. Finally, figure 14 shows the performance slowdown of OpenCL 2.0 compared to OpenCL 1.2 as we increase the padding size (i.e. decrease the number of entries in each cacheline). Except for utilizing coalescing (the 0-padding case), the figure confirms that the more the entries accessed on the same cacheline, the larger is the slowdown.

6.2.3 Rodinia Benchmarks

To further compare the memory management methods on the AMD system, we ported four applications (*BFS*, *Gaussian*, *kmeans*, *streamcluster*) from their original implementation in OpenCL 1.2 to OpenCL 2.0 and HSA. For the OpenCL 2.0 version, we allocate all the data used by the CPU and GPU as SVM fine-grain buffers. Also, we remove all data copies since they become unnecessary once data is allocated as SVM-fine buffers. Programmability is increased as the lines of code associated with data copies and the multiple CPU and GPU allocations are eliminated. For HSA, we port by using the HSA APIs for initialization and kernel launching on the CPU side. The application kernels themselves are not modified from the implementations in OpenCL 1.2 and 2.0. We use the CL Offline Complier (CLOC) to generate a BRIG binary module that can be read using an HSA API call.

Figure 15 shows the running times of the three implementations normalized to the running time with OpenCL 1.2. Based on the analysis of microbenchmarks in the previous sections, the performance slowdowns when using OpenCL 2.0 and HSA are expected. For *BFS* and *kmeans* the slowdowns are mainly attributed to slower executions of the GPU kernel due to the effects highlighted earlier. Although the OpenCL 1.2 version actually does copy data between the CPU and GPU in this realistic test in contrast to the previous section's microbenchmarks, the copying is performed only once in each direction. The kernels are executed multiple times, amortizing the overhead of data copying.

For *streamcluster*, the overhead of HSA is higher than OpenCL 2.0. The reason is because the HSA version does not use local memory on the GPU in contrast to the OpenCL 1.2 and OpenCL 2.0 versions. We were not able to allocate local memory in HSA, so we used global memory instead. This caused the HSA version to have a higher overhead than the OpenCL 2.0 version.

The performance speedup of HSA compared to OpenCL 1.2 for Gaussian is because the initialization of HSA takes less time than the initialization of OpenCL (25 ms vs 300 ms). Since the application runs for only about 2 seconds, the initialization overhead is significant.

Summary: Performance overheads shown with the microbenchmarks that stress the memory system do not necessarily apply to real applications that perform substantially more computation. *Programmability does not always come at a higher cost.* Exploring the overheads of programmability in applications with concurrent access to data by the CPU and the GPU, such as those introduced in recent work [4, 6, 11, 12, 16], would further elucidate this trade-off. We defer this analysis for future work.

7. Related Work

Prior to our work, Hestness, Keckler and Wood [9] analyzed the CPU and GPU memory system behavior on a simulator. Their analysis focused on how the memory behaviour of an application changes when it runs on the CPU vs on the GPU; they find significant differ-

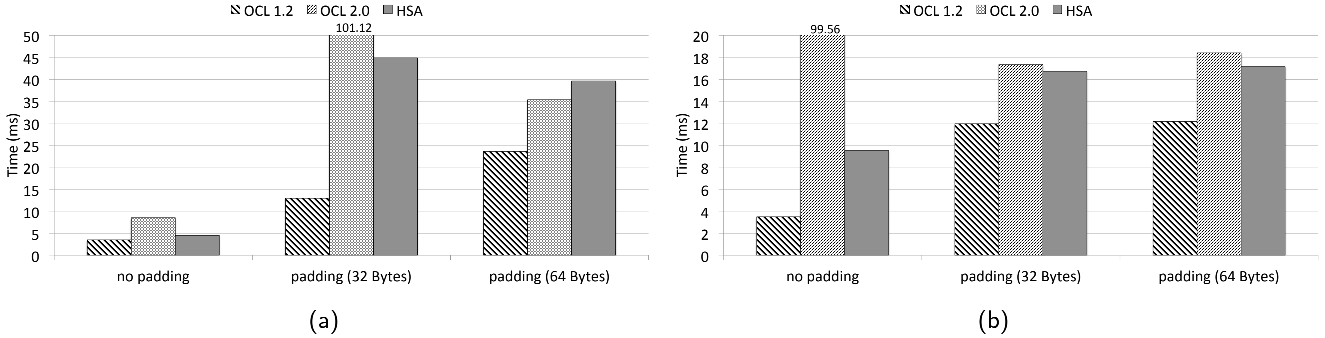

Figure 13. Latency of OpenCL 1.2, OpenCL 2.0, HSA kernel execution time. Array size = 64 MB and each array entry is either not padded, padded to 32 Bytes, or padded to 64 Bytes (cache line size). (a) Kernel: `memory[idx]++;`. (b) Kernel: `if(idx %2 == 0 ) memory[idx]++;`.

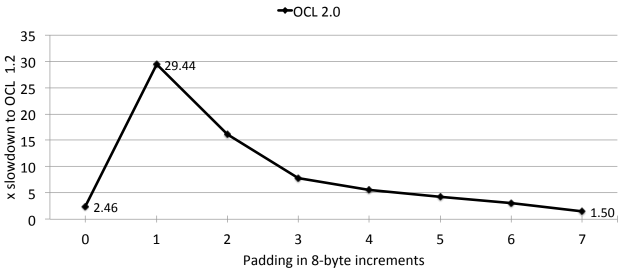

Figure 14. Slowdown of OpenCL 2.0 compared to OpenCL 1.2 as we increase the padding size of array entries. Kernel: `memory[idx]++;`.

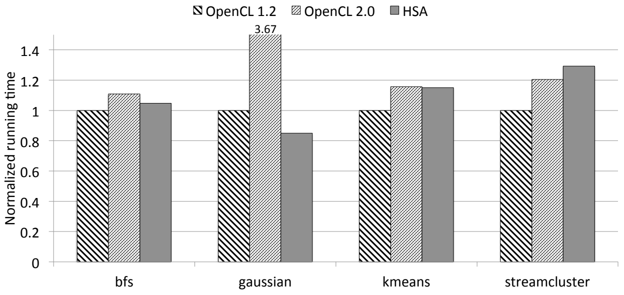

Figure 15. Runtime of Rodinia applications normalized to OpenCL 1.2

ences and attribute them to the differences in memory hardware. Our work studies the cost of memory accesses in different programming frameworks and analyses how programmability enhancements may affect their overhead.

The same authors as in [9] also analyzed the impact of hardware cache coherency on applications [10]. In contrast to our work, they used a simulated system. Similarly to our experience, they found that many existing GPGPU applications do not benefit much from the physically unified memory, because they were written at the time when this convenience was not available on GPU systems. To accommodate their study, they modify a number of applications to take advantage of the unified physical memory and cache coherence, and show that adapting the software is key to capturing the benefits of the hardware.

Implementing a full-blown CPU-like cache coherency on integrated CPU-GPU systems has practical limitations, so prior work looked at alternative software/hardware methods to implement partial coherency [3]. The authors relied on a simulator to evaluate their scheme which eliminates the need for CPU-GPU hardware coherency. Our evaluation on a real system showed that software-only designs may incur large overheads.

Similar to our performance analysis of the AMD platform, the authors in [13] characterize the benefits of the new features of OpenCL 2.0 and HSA. Our work focuses mainly on the cost of memory accesses under the different allocation methods on multiple frameworks.

The authors in [16] present a benchmark suite that explores the collaborative execution patterns on the CPU and GPU using an AMD APU. In our work, we evaluate the performance of existing platforms from a memory management scope. We briefly touch on concurrency workloads but mainly focus on memory access performance.

8. Conclusions and Future Work

We presented an analysis of memory management methods on real integrated CPU-GPU systems. We analyzed the cost of programmability enhancements, such as the unified virtual address space and data consistency. Future platforms promise hardware support for these features, which will enable comparing the cost of these enhancements in software and in hardware. We see two potential avenues for future work. *(1):* Hardware improvements to decrease the overhead of virtual address translation and cache coherency. *(2):* Improvements to software frameworks that discretely use these expensive features. However, to enable meaningful analysis in these future studies, there is a need for applications that truly rely on and exercise the features such as the unified virtual address space and cache coherency.

Acknowledgements

We would like to thank all the anonymous reviewers for their insightful and constructive reviews.

References

[1] AMD Graphics Core Next (GCN) Architecture. `https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`, 2012.

[2] CL Offline Compiler. `https://github.com/HSAFoundation/CLOC`, 2017.

[3] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler. Selective gpu caches to eliminate cpu-gpu hw cache coherence. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 494–506. IEEE, 2016.

[4] R. Barik, R. Kaleem, D. Majeti, B. T. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai. Efficient mapping of irregular c++ applications to integrated gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 33. ACM, 2014.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization IISWC*, 2009.

[6] J. Choi, A. Chandramowlishwaran, K. Madduri, and R. Vuduc. A cpu-gpu hybrid implementation and model-driven scheduling of the fast multipole method. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 64. ACM, 2014.

[7] O. Gabbay. AMD HSA kernel driver. `https://lwn.net/Articles/605153`, 2014.

[8] J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, A. J. Pena, et al. Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–10. IEEE, 2016.

[9] J. Hestness, S. W. Keckler, and D. A. Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *IISWC*. IEEE, 2014.

[10] J. Hestness, S. W. Keckler, and D. A. Wood. Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors. In *IEEE International Symposium on Workload Characterization (IISWC), 2015*, pages 87–97. IEEE, 2015.

[11] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 151–162. ACM, 2014.

[12] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[13] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli. A comprehensive performance analysis of hsa and opencl 2.0. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 183–193. IEEE, 2016.

[14] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[15] P. Singh, C.-R. M, P. Raghavendra, T. Tye, D. Das, and A. N. Platform coherency and soc verification challenges. In *13th International System-on-Chip (SoC) Conference, Exhibit & Workshops*, 2013.

[16] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–10. IEEE, 2016.

[17] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogenous systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016.

[18] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, et al. Design and analysis of an apu for exascale computing. In *HPCA*, 2017.