Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization

Daniel Lustig and Margaret Martonosi
Princeton University
{dlustig,mrm}@princeton.edu

Abstract

GPUs are seeing increasingly widespread use for general purpose computation due to their excellent performance for highly-parallel, throughput-oriented applications. For many workloads, however, the performance benefits of offloading are hindered by the large and unpredictable overheads of launching GPU kernels and of transferring data between CPU and GPU.

This paper proposes and evaluates hardware and software support for reducing overheads and improving data latency predictability when offloading computation to GPUs. We first characterize program execution using real-system measurements to highlight the degree to which kernel launch and data transfer are major sources of overhead. We then propose a scheme of full-empty bits to track when regions of data have been transferred. This dependency tracking is fast, efficient, and fine-grained, mitigating much of the latency uncertainty and cost of offloading in current systems. On top of these full-empty bits, we build APIs that allow for early kernel launch and proactive data returns. These techniques enable faster kernel completion, while correctness remains guaranteed by the full/empty bits.

Taken together, these techniques can both greatly improve GPU application performance and broaden the space of applications for which GPUs are beneficial. In particular, across a set of seven diverse benchmarks that make use of our support, the mean improvement in runtime is 26%.

1. Introduction

General-purpose computation on graphics processing units is a heterogeneous computing paradigm seeing increasingly wide use. In it, general-purpose (non-graphics) computational kernels are offloaded from a host CPU to a nearby GPU in order to improve the runtime, throughput, or performance-per-watt of the computation as compared to the original CPU implementation. As a testament to GPU performance potential, three of the top ten supercomputers in the TOP500 [38] and four of the top five in the Green 500 [15] use GPUs or GPU-like Intel MICs [22] in their clusters.

However, GPU-based acceleration is no silver bullet, and offloading to the GPU is not free. In both discrete and integrated GPUs, there are relatively large overheads associated with data transfer, kernel launch, and synchronization. While the most conspicuous of these is the data transfer to and from a discrete GPU, many other operations are also high in overhead, even when CPUs and GPUs share the same chip or memory hierarchy. Driver delays, uncertainty about data arrival times, and coarse-grained synchronization each add latency overheads regardless of the placement of the GPU relative to the CPU.

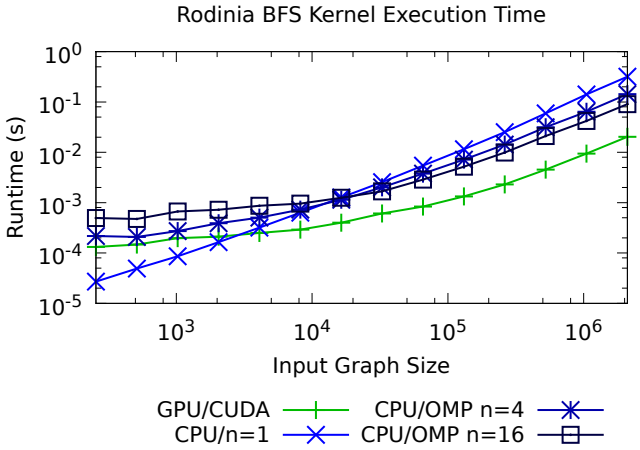


Figure 1: Runtime of breadth-first search [8] for varying input set sizes and hardware. The GPU is a discrete NVIDIA GTX 580 running CUDA and the CPU is an Intel Xeon X7560 running sequential code or a multithreaded OpenMP version. As input size changes, the type of hardware providing the fastest runtime changes: a single-threaded CPU is fastest for smallest inputs, and the discrete NVIDIA GPU is fastest at large sizes.

Consequently, it is beneficial to offload only when the transfer+launch overhead is outweighed by the performance gain achieved by executing the kernel on the GPU. GPUs are known to give excellent performance for large workloads which are *highly parallel* and *throughput oriented*: large kernels amortize the overhead of offloading, and throughput-oriented code can more easily hide the latency and variability of each individual operation. GPUs thus far have focused on improving the performance of such coarse-grained kernels and are heavily optimized for bandwidth. Workloads which are less parallel or which are more latency sensitive have been viewed as out of scope for GPU implementations. *The goal of this work is to reduce the duration, unpredictability, and performance impact of GPU onload/offload overhead in order to broaden the scope of applications that see performance benefits from GPUs.*

1.1. Motivating Example

To motivate our work, Figure 1 shows performance results for `BFS`, a breadth-first-search kernel. (Both axes are log scale.) For small input sets, the sequential CPU implementation has the smallest runtime, while for larger input sets, the GPU becomes fastest. The crossover point at which offload becomes beneficial occurs for graphs with roughly 4,000 nodes.

A primary reason GPUs are not always faster than CPUs

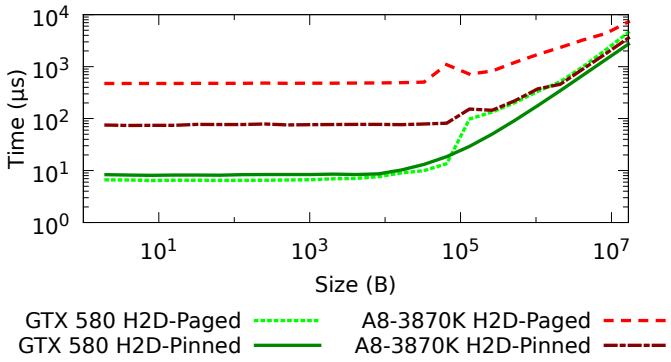


Figure 2: Memory copy latency between CPU and GPU for NVIDIA GTX580 (discrete) and AMD A8-3870K (integrated) GPUs. For small transfer sizes, the discrete GPU actually has lower latency than the integrated case.

is that kernel launch and data transfer introduce delays and variabilities that are added no matter the size of the kernel. For smaller workloads, this delay is significant or even prohibitive. Both the GPU and OpenMP implementations of the kernel have fixed minimum costs, depicted by the lines flattening out to the left of Figure 1, while the single-threaded CPU implementation has no such overhead and scales proportionally.

While the results shown here are for a discrete (off-chip) GPU, subsequent studies later in the paper show that the unpredictability of data arrival times can cause similar scenarios in integrated GPUs with smaller underlying data latencies. Our paper therefore aims to reduce these fixed costs for GPUs as well as the bottlenecks they cause, in order to broaden the set of applications that benefit from GPU acceleration.

1.2. Discrete vs. Integrated GPUs

An emerging trend is for smaller, lower-throughput GPUs to be integrated onto the same die as the CPU. AMD Fusion [3] and Intel Ivy Bridge [21] both provide OpenCL-programmable GPUs integrated onto the same die as the CPU. In currently available models, the memory space may be logically shared or partitioned between the CPU and the GPU; many modern systems continue to use partitioned memory spaces and pay the overhead of copying between the spaces every time data is transferred. Optimizations such as CPU pinned memory [30] were therefore developed to hide some of the latency and to improve bandwidth.

While on-chip integration is promising, it may not necessarily be a full solution. For example, in current systems, integrated CPU-GPU pairs do not always have smaller packet transfer latencies than discrete GPUs. To demonstrate this, Figure 2 shows real-system measurements of memory copy latency between CPU and GPU for different implementations and for different transfer sizes (note the log scales on the axes). Our measurements show that the AMD Fusion—an integrated GPU—actually has larger latencies than the discrete GPU for small packet sizes. Similar results have been obtained by previous work as well [10].

For future models, the HSA Foundation has published a roadmap [4] describing the expected features of future integrated GPUs: shared page tables and fully coherent memory, improved scheduling and command queuing, and more. Three upcoming features stand out in particular. First, a coherent shared memory space eliminates the need for memory copy operations. The two other features, user-level hardware-based GPU command queues and GPU virtual addressing with full paging support, have the potential to eliminate the need for a kernel driver completely.

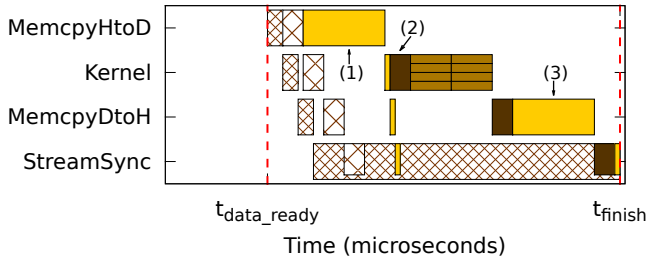
Nevertheless, some need for fine-grained data-oriented synchronization will still remain. First of all, discrete GPUs remain larger and more powerful than integrated GPUs, and memory copies will still be needed in domains such as HPC which require more raw throughput. Even when the memory system is shared, data may still be copied into or out of specialized (e.g., noncoherent) regions of memory during execution. Second, and most importantly, a large part of the overhead comes about from the coarse granularity of synchronization, rather than the transfer latency itself, and this fact is not addressed by integration roadmaps. Our exploration of fine-grained synchronization applies to both integrated and discrete GPUs (as well as to any producer-consumer relationship) and delivers performance benefits for both.

1.3. Contributions

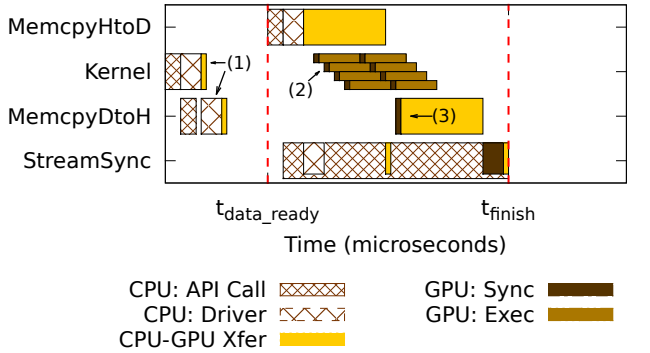
This paper proposes hardware and software support to improve data staging and synchronization in CPU to GPU communication. First, we propose full-empty bits (F/E bits) in GPU DRAM as an efficient, fine-grained data dependency tracking mechanism. On top of this hardware, we add program-level API enhancements to enable GPU execution overlap with data transfer, even within a single kernel. In particular, we evaluate an "overlap-start" technique that supports early sending of GPU kernel code from the CPU, with computation starting as soon as the F/E bits indicate data is ready. Likewise, API support for an "overlap-finish" technique allows subsets of GPU data results to get transferred back to the CPU proactively, rather than waiting for the entire kernel to finish. Together, these techniques deliver performance improvements of 26% on average across a range of benchmarks.

The performance improvements we demonstrate help to broaden the space of kernels that can benefit from CPU to GPU offloading. For example, as described in Section 7.2, we find that for `vectorAdd`, the original GPU delivers faster performance than a CPU for all vector sizes greater than 128K elements. With our techniques, the GPU performance improves for all vector sizes, and in fact even surpasses the CPU performance for vectors as small as 32K elements. In this way, even input sizes for which the GPU was originally slower now get performance benefits by offloading to our enhanced GPU.

To demonstrate our ideas, we built an enhanced and accurate simulation infrastructure. We created a CPU-GPU communication simulator that works with a version of GPGPU-Sim [6] modified to model F/E bits. We validate our CPU-GPU transfer times against real-system measurements and on average find agreement within 5% of total runtime. The simulator is released for open use.



(a) Baseline. Labeled portions described in Section 2.1.



(b) Performance Improvement using the "full-overlap" scenario. Labeled portions described in Section 2.3.

Figure 3: Timelines depicting how fine-grained synchronization removes many of the latencies involved in offloading kernels to the GPU. Operations such as kernel launch are taken off of the critical path, and communication and computation can be overlapped.

2. Faster GPU Offloading

Many throughput-oriented applications currently hide latency by pipelining GPU computation with communication to and from the GPU. More specifically, the GPU can be working on one of several independent kernels while others are being staged for transit to or from the GPU. For a balanced computation-to-communication ratio and with sufficient kernel-level parallelism, this pipeline can stay full. Similarly, with sufficient thread-level parallelism, the latency of memory requests can be hidden by efficient context switching, keeping each core active with other work until the response is received. These techniques require very detailed and device-specific tuning, however, and are limited to applications with particularly amenable characteristics. Our goal is to provide support to broaden the sphere of applications that can use GPUs easily and successfully.

2.1. Offloading to GPU: Baseline

Figure 3a shows a timeline diagram for the stages of a GPU computation including the code and data transfers from CPU to GPU. (This diagram is primarily aimed at discrete GPUs.) First, on the CPU side, a `memcpyHtoD` (Memory Copy from Host to Device) command is called to transfer data from host (CPU) to device (GPU). A second command to launch the code kernel to be executed on the GPU side is invoked as well. Data is made available to the GPU in one of two ways: either it is copied into the GPU memory space (labeled as 1 in the diagram), or the GPU directly accesses CPU memory (not pictured). Although no GPU execution has started yet, the CPU also launches the `memcpyDtoH` command by which data will be copied back from device (GPU) to host (CPU). The runtime system will ensure that this `memcpyDtoH` command does not actually execute until the computation has completed, so that data is not copied back prematurely. As the `memcpyHtoD` executes, the input data is written into the GPU memory space. Even after the data is present, due to the weak memory consistency models in use, it is not safe for GPU programs to use this data until a coarse-grained memory synchronization operation has completed (2). This means that all computation is held up until the last piece of data arrives at the GPU. Only then can threads start to execute. Then, any memory requests to read this data must travel from the GPU core to DRAM and back, costing hundreds of extra cycles. Finally, when the GPU kernel completes execution, it writes to the output array. When all the writes have completed and synchronized, the array is copied over the interconnect back to the host CPU (3).

2.2. Causes of Latency

As the timeline illustrates, many factors combine to cause the latency of offloading to GPUs to be so large. First of all, the kernel launch command itself takes time to execute, regardless of the size of the computation being done. Second, there is a delay between when data physically arrives at its destination versus when it is ready to use. This is because a synchronization operation must occur in between in order to guarantee that the computation will read the correct data. Besides these factors, API and driver overheads introduce delays with each call made. Several of these factors remain important in integrated GPUs as well.

Also, discrete GPUs communicate via commodity expansion ports. Such communication links focus mainly on bandwidth, and many optimizations for high bandwidth, such as write-combining buffers, can significantly increase latency for certain individual pieces of data. Nevertheless, PCIe is still the standard for discrete graphics cards. Packet-based protocols such as PCIe add packetization overhead—up to 90% of PCIe latency comes from higher layers in the stack [29], similar to the analogous cost in other packet-switched protocols [26]. Furthermore, while the bandwidth of PCIe is consistently improving with each generation, the *latency* often stays the same or even gets worse [29].

2.3. Offloading to GPU: Our Approach

Ideally, our goal is: as soon as the input data becomes available (e.g., as some previous computation creates it), we wish to complete this entire kernel computation with the lowest possible latency. In Figure 3, this corresponds to the difference

$$runtime = t_{finish} - t_{data\_ready}.$$

Note that the time at which the first command is launched is not counted in this ideal equation. In existing systems, it is only safe to launch the kernel *after* the data is ready at time t_{data_ready}, as in Figure 3a. However, our proposal removes this dependency from software by allowing F/E bits to more efficiently handle it in hardware. In other words, *kernel launch is no longer on the critical path of the computation*, and it can safely happen *in advance of the arrival of data at the GPU*.

Figure 3b shows a potential timeline if this paper's hardware and API proposals are applied. In particular, we can proactively begin kernel execution even if not all of the data has arrived (1). Synchronizing data using F/E bits means that we can overlap computation on the already-present data with arrival of the remaining data (2). As the data is copied into its destination, the target F/E bits are set to full. At this point, any dependent GPU instructions can continue. Likewise, as results are computed and written into the output array, the associated F/E bits are marked full. As this occurs, results can begin being copied back from device to host early (3). Individual portions of the copy will stall if the F/E bits indicate that certain data items have not yet been produced.

2.4. Our Proposal

Thus far, we have motivated the fact that CPU-GPU transfers of code and data can often impose too much overhead on GPU kernel executions. Waiting for *all* of a data block to be ready is the only currently safe method for code to execute, but this often results in a performance slowdown. If a kernel thread could block only on the specific addresses it uses, better performance and execution overlap can be achieved in both discrete and integrated GPUs.

With the goal in mind of increasing overlap and optimizing for producer-consumer data, this paper explores the implementation issues of F/E bits applied to GPU memory. These bits act as guards on memory operations. Accessing a location that is not yet marked as "full" causes the thread to stall. When a write occurs that causes the location to be considered full, any memory requests and threads waiting on that data are unblocked. While F/E bits have been previously proposed in other architectures (see Section 8), they have not been explored for GPU applications, and our results show considerable leverage. Our work explores the performance potential they offer, as well as their implementation issues.

3. GPU Overview

3.1. GPU Hardware

GPUs execute a large number of threads in parallel, with computations performed by a set of SIMD and/or VLIW multiprocessors. A memory system provides spaces private to each thread ("per-thread local memory" or "private memory"), spaces private to each thread block ("per-block shared memory" or "local memory"), and a global memory space shared by all CPU and GPU threads. GPUs are generally optimized for throughput, making use of aggressive context switching to cover the latency of memory requests.

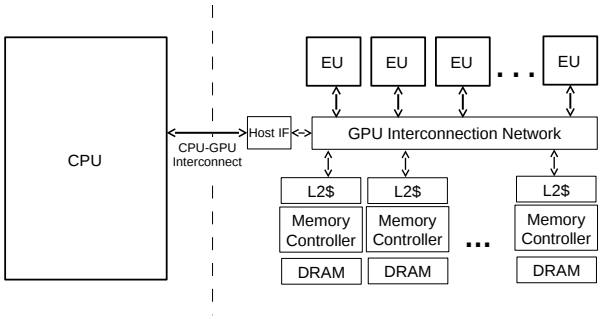


Figure 4: Typical CPU and GPU architecture. The CPU-GPU interconnect may be an on-chip bus for integrated GPUs or an peripheral card interface such as PCIe for discrete GPUs.

Figure 4 shows a typical GPU architecture. Each execution unit (EU) consists of a set of SIMD execution pipelines connected to an L1 demand-fetched cache, shared memory scratchpad, a constant/texture cache, and an interface to the GPU interconnect. Across the GPU interconnect are a set of memory partitions containing a coherent L2 cache, a memory controller, and a connection to off-die global memory DRAM. In integrated GPUs, the CPU and GPU communicate via an on-chip bus, and in discrete GPUs, via an expansion card slot. While the details vary by manufacturer (e.g., Intel GPUs have no scratchpad, some AMD GPUs are VLIW in addition to SIMD), the general approach is similar.

3.2. GPU Programming Model

Programmers write GPU programs in terms of thread blocks to be offloaded to the GPU. Each thread block consists of a set of software threads organized in a one-, two-, or three-dimensional grid. As hardware compute units become available, a hardware scheduler on the GPU dispatches each thread block to available cores. Since the order of execution and the interleaving of threads is impossible to know in advance, it is generally difficult or impossible for threads in different blocks to directly communicate with each other during execution.

Before each kernel or data transfer operation can begin, all of the necessary data must be ready to use. This synchronization takes place at various levels. The coarsest granularity of synchronization occurs between commands in a queue or stream: each command in a stream must complete entirely before the next can execute. Although this does provide synchronization across the entire kernel, it is very slow, as it often even involves a round trip to the CPU for API call completion. Finer-grained barriers can also be used to synchronize control (not data) within a thread block.

Previous work has studied the use of atomic operations to implement a custom synchronization primitive [13, 41]. However, these techniques are not guaranteed to be correct, due to the weak consistency model (see below). The F/E bits we propose can guarantee correct ordering of the producer and consumer requests to each location.

4. Full/Empty Bits for GPUs

4.1. Overview

F/E bits have been proposed in several settings as hardware support for efficient producer-consumer data accesses (as discussed in Section 8). With this technique, every region of memory of some specified granularity has an associated F/E bit that is checked and updated in parallel with the data. This bit can have one of two states: "full" or "empty", denoting whether the location contains valid data.

Associated with each incoming memory request are two new components: the *trigger condition* and the *update action*. The trigger condition defines how to handle each request based on the status of the targeted F/E bit. It can either be *wait for full*, *wait for empty*, or *non-triggered*. The non-triggered case causes the system to ignore F/E bits and default to normal operation. When the request is processed, the update action directs whether the F/E bit should be *filled*, *emptied*, or *left unchanged* as a result.

In choosing triggers and actions for each request, we categorize memory requests into three classes. First, for CPU requests, the triggers and actions are explicitly specified by the user via the proposed API extensions of Section 4.4. For GPU requests, reads have a fixed trigger of waiting for data to be marked as full, and they perform no action, while GPU writes have no trigger and have an implicit action of marking their target location as full. These choices reflect the most common status of the GPU as providing hardware support for operations being coordinated by the CPU. Other choices are possible and are discussed in Section 4.5.

The most common usage scenario is a strict producer-consumer relationship, such as the one between the CPU and the GPU. For example, suppose the GPU (the consumer) wishes to read data provided by the CPU (the producer). The GPU issues a read request with a *wait for full* trigger condition. Until the producer sends the data, the F/E bit for the memory location is set to empty, and the GPU requests blocks. When the producer actually writes the data, the F/E bit gets set, allowing the GPU read requests to proceed safely. For coalesced requests, responses are returned when all the relevant F/E bits indicate readiness. Other scenarios work similarly.

4.2. Placement

We chose to place F/E bits with GPU DRAM (global memory), as this is along the offloading critical path between the CPU and the GPU. As data is sent from the CPU to the GPU, it must first be written into the GPU DRAM. Only then can it be sent to the GPU cores and placed into the cache and/or scratchpad. Furthermore, many GPU kernels operate on a lot of "touch-once" data, and as a result the cache or shared memory are often not even used for such data [23].

Every four bytes of memory get a single associated F/E bit. If even the modest 3% DRAM overhead of such an approach is too high, one could choose to implement F/E bits only on a smaller subset of "synchronized global memory" rather than on the full DRAM. Since CUDA already supports various memory spaces, such an approach would be quite feasible [30], and could allow for a much smaller F/E bit structure to be embedded into the memory controller itself. Section 9 discusses these issues further.

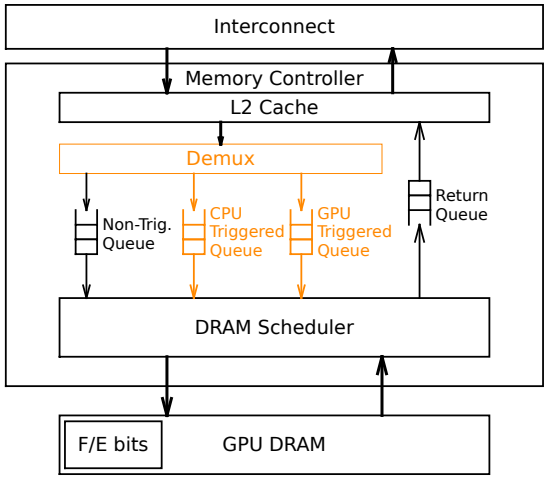


Figure 5: Full/Empty bits, GPU DRAM, and the GPU memory controller. Changes made to support fine-grained synchronization are shown in orange.

4.3. Memory Controller Architecture

Figure 5 illustrates the proposed GPU memory controller. Memory requests arrive on the interconnect, pass through the L2 cache, and then enter one of three queues (explained below). Requests stall at the head of their assigned queue until their trigger condition (if specified) is satisfied. By default, requests have no trigger, and these proceed through the memory controller in the normal manner. On the other hand, if a specified trigger condition is not yet satisfied (e.g., the GPU tries to read an empty location), the request stalls at the head of its queue. Once the condition is satisfied by a request (e.g., a CPU write to the same location) passing through a different queue, the original request is allowed to proceed. The requests then merge back into a single channel in the DRAM scheduler. At that point, if an update action was specified, the F/E bit is updated accordingly. Except for the full/empty bits and the two new queues in the middle, this process is entirely the same as for existing GPUs.

Each of the DRAM scheduler queues is strictly ordered for the sake of hardware simplicity. The DRAM scheduler checks the trigger conditions (if specified) of the requests at the head of each queue; only the head of each queue is ever observed. If the trigger of a request is not satisfied, the DRAM scheduler will stall that request (and therefore all other requests queued behind it in the same queue), and it will then snoop the requests passing through the other queues for an action which causes the trigger to become satisfied. When this occurs, the previously blocked request is then released. Other queued requests, if there are any, can then proceed as well.

Multiple queues are used to avoid head-of-line blocking

while ensuring that the memory controller can correctly enforce dependencies on the F/E bits. In contrast, consider if there were only a single queue, and a GPU read request was made to a currently empty memory location, in anticipation of the CPU later filling that location. If the CPU request sat behind the GPU request in the same queue, then it would never be able to bypass the GPU request to fill the location. The use of multiple queues allows the CPU write to proceed past the blocked GPU read.

Our implementation uses three queues. The first handles all non-triggered requests, ensuring at least one non-blocking path from the CPU or GPU to DRAM. The other two categories, triggered requests from the CPU and triggered requests from the GPU, must also be split into their own queues so that GPU requests can bypass blocked CPU requests. For example, in the overlap-finish strategy that will be described in Section 5.2, the *memcpyDtoH* requests may be sent before some threads of the kernel make read requests. Splitting triggered CPU requests and triggered GPU requests into different queues ensures that of the three parallel commands (`memcpyHtoD`, kernel launch, `memcpyDtoH`), no one will block any others.

The scenario in which a large number of requests go out to and then stall in the memory system due to unsatisfied triggers is handled similarly to existing scenarios in which many threads experience cache misses that take a long time to return. MSHRs and the rest of the memory system store the outstanding requests and track the arrival of the data. No extra buffering is used for this purpose. Should the MSHRs fill up, the cores would simply stall on the structural hazard of the memory system and the lack of available MSHRs and hence not issue any new requests until space again becomes available.

As with CUDA and OpenCL, our design is geared towards producer-consumer CPU-GPU accesses, as these are the most common relationship type. Arbitrary interleavings of CPU and GPU requests to the same address are rare and discouraged in GPGPU, due to the unpredictable execution order of threads, the weak consistency model, and the fact that all threads may not even launch simultaneously. Our implementation targets and correctly handles the common case in which the producer is the CPU and the consumer is the GPU, or vice versa. However, if needed, our approach scales linearly with the number of independent streams that need to execute simultaneously.

4.4. API Extensions for Full/Empty Bits

Our API extensions allow programmers to specify the update action and trigger explicitly for CPU requests. For update actions, the default flag `cudaMemcpyActionNone`[1] specifies that the F/E bit should not be changed when memory in this region is written or read. The other two flags, `cudaMemcpyActionFill` and `cudaMemcpyActionEmpty`, specify that the bit should be set or cleared, respectively.

[1] Our notation draws from CUDA terminology here, but OpenCL has analogous calls. For instance, OpenCL contains flags like `CL_MEM_READ_ONLY` and `CL_MEM_WRITE_ONLY` flags earmarking areas for particular uses. Our work extends such flags to apply to F/E bits as well.

For trigger conditions, the first specification, `cudaMemcpyTriggerNone`, specifies that a request should simply ignore the status of the F/E bit. The other two, `cudaMemcpyTriggerFilled` and `cudaMemcpyTriggerEmptied`, specify what state the F/E bits for the requested location must be in before the request completes. As with any software synchronization operation, a correct program must pair these with an update action request to *set* or *empty* the F/E at some point.

We maintain the CUDA notion of a stream of dependent operations such that each command cannot execute until its predecessor has completed. However, data dependencies that are now handled in hardware by F/E bits no longer need to be in the same API-level stream, as current systems would require. In fact, moving such operations into separate streams is exactly what enables the hardware to overlap kernel launch, data transfer, and execution efficiently. In OpenCL, which uses event wait lists rather than streams, similar semantics could be created using the flag `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` and choosing wait lists in a similar manner, i.e., intentionally not listing overlapping events in wait lists.

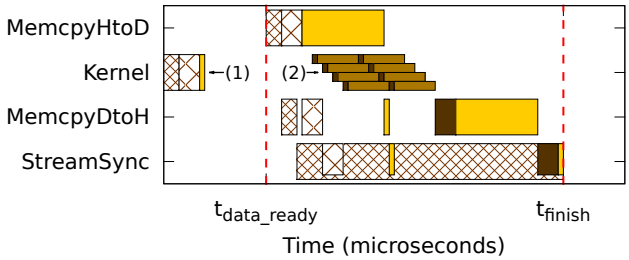
4.5. Discussion and Future Extensions

As mentioned, we focus on the very common case of producer-consumer writes between CPU and GPU. Although additional support for other features could be added, their less frequent use makes it harder to justify supporting them in hardware. For completeness, we describe some issues here.

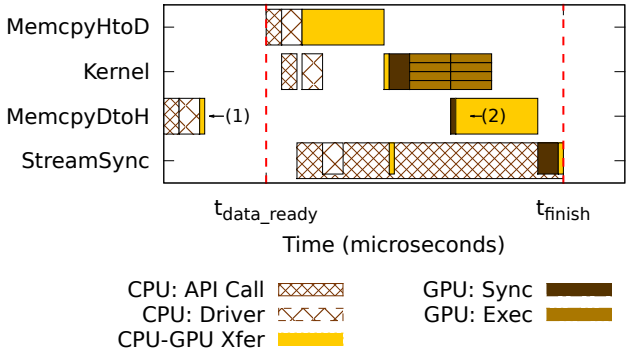
For programs where data is read or written more than once before being discarded, the simplest F/E mechanism would be insufficient. To handle such multiple-empty scenarios, marking a bit "empty" could be caused either by explicit programmer request at the GPU instruction level or by an event such as the end of a kernel, instead of by a single read. Furthermore, a F/E bit could be replaced with multiple counter bits, at the cost of higher storage overhead. Another possibility would be the introduction of override commands, which could either toggle F/E bits without affecting the data, or could simply release all pending requests regardless of the status of their trigger conditions. Finally, for code where using F/E bits seems particularly awkward, user can simply avoid F/E bits and default to existing software-based synchronization mechanisms.

Another issue is the case that an incorrect or highly-unusual GPU program might try to read empty GPU memory that is never written by either the CPU or the GPU. (We know of no benchmark that does this intentionally or unintentionally.) Without F/E bits, the read result would be undefined data, but with F/E bits, it can lead to queueing interlock problems if a write to fill the location cannot bypass the read in the GPU request queue. Our solution again prioritizes hardware simplicity: our GPU startup routine (one-time, not per kernel) includes an option to zero out memory and mark it as full.

Other memory controller design variations are also possible. For example, looking past the head of DRAM scheduler queues could help performance at the cost of power and complexity. Other options would include the use of timeouts or



(a) Overlap-start strategy. Labeled portions described in Section 5.1



(b) Overlap-finish strategy. Labeled portions described in Section 5.2

Figure 6: Usage scenarios in which fine-grained synchronization allows execution and communication to overlap for a single command queue. By shortening the critical path, the latency of offloading is reduced significantly. These strategies combine to form the full-overlap case shown in Figure 3b.

NACKs to alert the core of a deadlock situation, at which time the request could be retried or an exception handler could be triggered. Future work may explore such possible additions.

Finally, we note that the most sophisticated potential variations of F/E bits (e.g., in which GPU requests also have triggers and update conditions explicitly specified) may require changes to the GPU ISA. However, GPUs already frequently change both ISAs and microarchitectures, as both are hidden under more portable software interfaces (such as PTX for CUDA); such a requirement is not problematic.

5. Usage Scenarios

In our experiments, certain patterns of API calls are particularly common, and so we describe them in more detail. Figure 6 illustrates two strategies for making use of fine-grained synchronization. (Recall that Figure 3a presents the no F/E bit baseline as implemented in current GPUs.)

5.1. Overlap-Start Scenario

In Figure 6a, the kernel launch itself no longer has to be on the critical path, and so we start it in advance (1). At some later time, once the input data is prepared and ready on the CPU side, the CPU sends it to the GPU. Each data item fills its associated F/E bits as it arrives. Consequently, the threads waiting for that data can resume execution as soon as the F/E

```
//  In advance - nothing can be done!
//  When input data is ready
memcpyAsync(d_indata, h_indata, HtoD, stream[0]);
kernel<<<grid, tds, smem, stream[0]>>>();
memcpyAsync(h_outdata, d_outdata, DtoH, stream[0]);
streamSynchronize(stream[0]);
```

(a) Baseline (no F/E bits) approach on default GPU.

```
//  In advance, pre-load kernel
kernel<<<grid, tds, smem, stream[0]>>>();
//  When input data is ready
memcpyAsync(d_indata, h_indata, HtoD, stream[1],
  TriggerNone, ActionFill);
memcpyAsync(h_outdata, d_outdata, DtoH, stream[0]);
streamSynchronize(stream[0]);
```

(b) Overlap-start semantics on GPU with F/E bits.

```
//  In advance, send the command to copy
//    back the output data
memcpyAsync(d_indata, h_indata, DtoH, stream[1],
  TriggerFull);
//  When input data is ready
memcpyAsync(h_outdata, d_outdata, HtoD, stream[0],
  TriggerNone, ActionFill);
kernel<<<grid, tds, smem, stream[0]>>>();
streamSynchronize(stream[1]);
```

(c) Overlap-finish semantics on GPU with F/E bits.

```
//  In advance, send two commands
kernel<<<grid, tds, smem, stream[0]>>>();
memcpyAsync(d_indata, h_indata, DtoH, stream[2],
  TriggerFull);
//  When input data is ready
memcpyAsync(h_outdata, d_outdata, HtoD, stream[1],
  TriggerNone, ActionFill);
streamSynchronize(stream[2]);
```

(d) Full-overlap semantics on GPU with F/E bits.

Figure 7: Pseudocode snippets with proposed API extensions. Names shortened for legibility. The overlap-start scenario moves the host-to-device `memcpy` into a separate stream so that it overlaps with kernel execution.

bits are marked full (2). We call this strategy *overlap-start*.

Figures 7a and 7b show code for the baseline (Figure 3a) and overlap-start (Figure 6a) cases. In the baseline case without F/E bits, each operation in the sequence depends on the previous completing. Consequently, all operations occur in-order and in the same stream. The overlap-start case has three changes. First, kernel launch is moved to the beginning of the sequence, before the input data is all ready. The F/E bits enforce that kernel computations will not proceed past a point where they depend on data that is not ready, but other aspects of kernel launch and computation can begin as soon as possible. Second, the host-to-device `memcpyHtoD` command is placed into stream 1 so that it can overlap with the kernel launch in stream 0. Third, the *memcpyHtoD* command sets its action to "fill" so that it will fill the target F/E bits.

5.2. Overlap-Finish Scenario

Figures 6b and 7c illustrate the overlap-finish strategy, which exploits that some portions of the results array are ready before others. Therefore, we preemptively send the command to copy back the results from GPU to CPU in advance (1), with the trigger condition that result data must be "full" before it is copied back. Later, as pieces of the result array are filled by the GPU, they are indeed copied back to the CPU before the rest of the kernel may have completed (2). Computation completes when all of the results array has been sent back.

5.3. Full-Overlap Scenario

Finally, while Figures 6a and 6b show the strategies applied separately, they can also be used together. When both overlap-start and overlap-finish are used, we refer to this as the full-overlap strategy. This was pictured earlier, in Figure 3b, and the pseudocode is shown in Figure 7d. In the full-overlap scenario, both the kernel launch command and the command to copy back the results are sent earlier. As data arrives, it is processed as in the overlap-start case, and as results are ready, they are sent back as in the overlap-finish case. This strategy combines the benefits of the previous two, giving the best performance.

6. Simulator Infrastructure

Our work builds from existing simulators by adding detailed and real-system-validated models where needed to track the full path through our proposed techniques. The baseline GPU architectural model and the functional (non-timing) implementation of CPU-GPU communication are provided by GPGPU-Sim 3.0.2 [6]. To this, we add a detailed timing model for CPU-GPU communications (Section 6.1) that has been validated via real-system measurements. We also add a GPU host interface module tracking memory synchronization (Section 6.2), and the F/E bits themselves, plus the necessary enhancements in the memory controller to support them (Section 6.3).

6.1. CPU and Interconnect Timing Model

An often-critical part of measuring the performance of low-latency workloads is the timing of CPU-GPU communication. Simulators such as GPGPU-Sim, however, do not measure the timing of these operations, focusing instead on modeling the GPU itself. Other simulators are discussed in Section 8.

The simulator must accurately model several aspects of CPU-GPU communication. Some of the latency comes from data transfer itself. At the physical layer, this is simply the size of the data being sent divided by the bandwidth of the link. However, this is not the only component of the cost. Each API call, including `memcpy` and kernel launch commands, incurs multi-microsecond overhead in the CUDA/OpenCL driver. Even asynchronous calls in which control returns to the program "immediately" take microseconds to enqueue. For small messages these costs are non-negligible.

Rather than calculating a single fixed latency for each operation, we instead model operations as progressing through a set of stages, incurring some latency at each step. We model three stages: (i) time spent to enqueue a command into the driver, (ii) time in the driver preparing the command for execution, and (iii) time executing the command. A command may be stalled between stages or during execution when necessary. This breakdown allows us to represent the progression of asynchronous API calls in particular, as there may be a significant delay between a command being queued and its execution. As an example, the latency of a 128KB `memcpy` from host to device is broken down as follows: (i) 1.2 μs to enqueue the command, followed by a wait for the driver to become available; (ii) 7 μs to process the command; (iii) and finally a delay of (128KB) / (6.8 GB/s) to transmit the data.

With this methodology, the total latency is not a single number calculated ahead of time, but instead it is a sum of components that may vary dynamically depending on the state of the system. As described in Section 6.4, we microbenchmarked a real GPU system to obtain accurate timing for each of these components.

6.2. GPU Host Interface

The second new simulator component is a host interface module on the GPU. This GPU module, known in NVIDIA terminology as the CUDA Work Distributor, is responsible for processing commands received from and sent to the CPU. We are particularly interested in the time to synchronize between dependent operations. As memory requests from the CPU are processed, the GPU must wait for all of the requests to be successfully ACKed by the memory system before allowing subsequent operations from the same stream to begin.

The host interface module is simulated with a queue per stream of computation, as is done in modern GPUs [32]. As each command is processed, the module tracks any new pending ACKs or thread blocks that it must wait for. As the ACKs return from the memory system, they are removed from the queue for their associated stream. When the queue for a stream is empty, the next command can be processed.

6.3. Full/Empty Bits and Memory Controller

The F/E bits are placed with GPU DRAM, as described in Section 4.2. We also implement the three-queue memory controller described in Section 4.3. If requests from more than one of the queues are pending at once, a simple priority scheme arbitrates: non-triggered requests have the highest priority, followed by triggered CPU requests, and then triggered GPU requests.

6.4. Choosing Simulator Parameters

We use microbenchmarking on real systems to select simulation timing parameters that match a real NVIDIA GTX 580 hardware baseline. Accurate measurement methodology on real GPU systems is non-trivial, but our detailed approach offers good accuracy with low perturbation. The natural first step might be to use the built-in hardware profiler counter capability in the GPU. However, enabling these GPU profiler counters adds a very large overhead and possible distortion in program runtime. Furthermore, the profilers force kernels

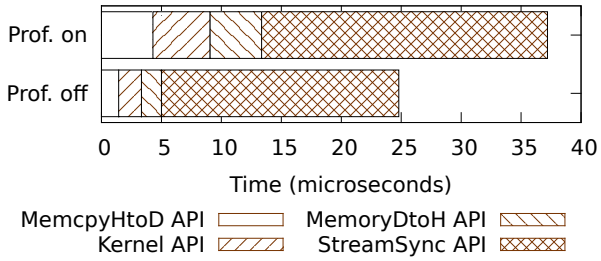


Figure 8: Real-system timelines of a program running with hardware profiler counters both on and off. Note that all `memcpy` calls and kernel launch are asynchronous, so the times are of the API calls only. The overhead of the counters is too large for the counters to be used as a validation target.

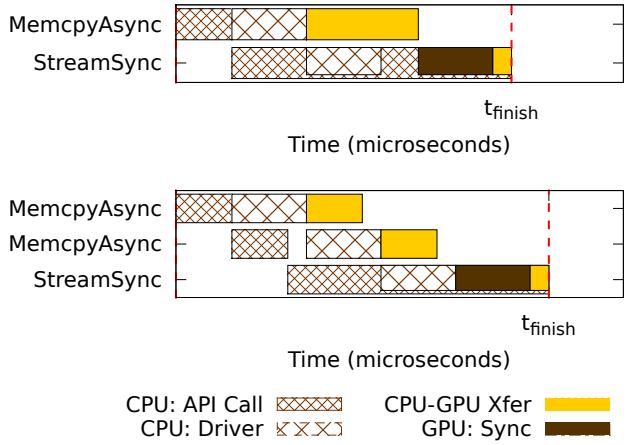


Figure 9: Illustration of time components of GPU API call and driver latency. Only API calls can be measured directly. In this example, the difference between a single `memcpyAsync` of size N and two `memcpyAsync` calls of size $N/2$ gives the overhead of each driver call.

to be serialized, when this is exactly the behavior we wish to avoid. Figure 8 shows the difference in timing results seen for FIBLookup when measured via profiling counters (top) and via lightweight timers placed only at API calls (bottom). Clearly, for accurate views of low-latency kernels, we cannot rely on profiling counters.

For better accuracy, we instead use the CPU's real-time clock to time API call and driver overhead. We time `memcpy` calls of various sizes and flags, kernel launches, and synchronization requests under various circumstances. Using carefully-chosen sequences of API calls, we extract the timing of each latency component. Figure 9 illustrates our approach. We perform two separate but similar sequences of operations. For the first, we observe a `memcpyAsync` of size N followed by a `StreamSynchronize` operation. In the second, we split the original `memcpyAsync` into two halves, otherwise copying the same host data to the same GPU locations. Taking the difference of the total latency for the two scenarios gives the driver latency of processing a single `memcpyAsync` command.

Table 2 lists the values we use in our simulator. These reflect the NVIDIA GTX 580 system from Table 1.

	Discrete	**Integrated**
CPU	Intel Xeon E3-1245 Sandy Bridge	AMD A8-3870K Llano
GPU	NVIDIA GTX 580	
GPU Cores	512	400
GPU Freq.	772MHz (core)	600 MHz
GPU DRAM	1.5GB	256MB (local)
CPU-GPU Interconnect	PCIe 2.0 16x	Fusion Compute Link / Radeon Memory Bus
GPU Mem. BW	192 GB/s	29.8 GB/s

Table 1: GPU Architectural Specifications.

API Call	**Call Latency**	**Driver Latency**
cudaMemcpy	$7\mu s + \frac{data\_size}{6.8GB/s}$	
cudaMemcpyAsync	$1.2\mu s$	$6\mu s + \frac{data\_size}{6.8GB/s}$
cudaLaunch	$1.5\mu s$	$3\mu s$
cudaStream Synchronize	$1\mu s$, then $1\mu s$ after sync'ed	
cudaDevice Synchronize	$1\mu s$, then $1\mu s$ after sync'ed	

Table 2: API call latencies used in simulator. Non-blocking calls have two latency components: (i) API call itself (after which control returns to the application), and (ii) driver call (after which the operation is complete). Blocking calls show a single combined latency, since control does not return to the application until the operation is complete.

6.5. Model Validation

Figure 10 shows latency measurements for synchronous and asynchronous `cudaMemcpy` calls of various sizes. Our measurements corroborate the interconnect timing model we describe in Section 6.1. Namely, the latency of these calls can be considered as the sum of the actual data transfer time (as a function of bandwidth) plus a fixed overhead cost.

As a final check, we validate application runtimes given by model against the real NVIDIA GTX 580 system described in Table 1. To validate, we compare the runtime of our benchmark suite on real hardware to the simulated system being modeled. Across the measured benchmarks and for the full application runtime, the simulator and real-system measurements agree with outstanding accuracy—to within 5% on average, with a maximum deviation of 15%.

6.6. Benchmark Suite

We test our proposal on a diverse set of benchmarks taken from various sources, as shown in Table 3. These benchmarks range from microbenchmarks to full workloads, with consistent performance improvement across the suite. Our goal is to enable the offloading of workloads previously considered "too small to offload"; we therefore include some smaller workloads intentionally, not to skew our results, but to demonstrate

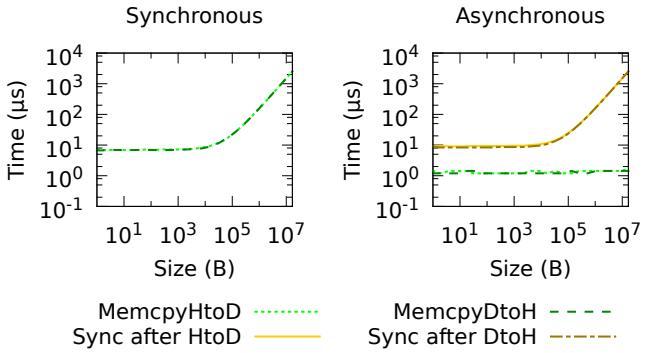


Figure 10: Real-system execution time for cudaMemcpy and associated calls based on the size of the data being transferred. In the synchronous case, the execution time is equal to a fixed factor of $7\mu s + size/BW_{eff}$. In the asynchronous case, the API call itself takes $1.2\mu s$ and the actual transfer and synchronization is $9\mu s + size/BW_{eff}$.

Suite	Benchmark	Dataset
NVIDIA SDK	vectorAdd	8K-256K elts./vec.
NVIDIA SDK	matrixMul	64 × 64 elts./mat.
NVIDIA SDK	histogram64	256KB
NVIDIA SDK	BlackScholes	64K options
Rodinia	BFS	4K-node graph
Rodinia	HotSpot	64×64 grid
Custom	FIBLookup	512 threads

Table 3: Benchmarks used in our study

the considerable new opportunity.

Four benchmarks are from the NVIDIA SDK [31]: `vectorAdd`, `matrixMul`, `histogram64`, and `BlackScholes`. These represent standard computational kernels that see widespread use as components of other algorithms. In addition, we use `BFS` [19] and `HotSpot` [20] from Rodinia [8] to represent more complex benchmarks. Finally, we write a `FIBLookup` (packet forwarding) benchmark, similar to one analyzed in previous work [18], to explore the behavior of a very latency-sensitive workload with close to zero data locality among threads. We use the simulator infrastructure developed in Section 6 as our execution environment. We run our benchmarks to completion in order to account for all execution and all CPU-GPU communication. To take advantage of F/E bits, we modify the source code of the benchmarks to implement each of the four use cases described in Figure 6: baseline (using asynchronous `memcpy` calls), overlap-start, overlap-finish, and full-overlap. We then present results for each benchmark for each of these cases.

7. Results

7.1. Comparing to Baseline GPU

Figure 11 shows the application runtimes using F/E bits and related API support, normalized to the GPU baseline case. (Lower is better.) In the GPU baseline, no F/E bits are available, so standard synchronization fences are used. For these applications, the GPU baseline already represents a performance improvement over a single-threaded CPU, even when CPU-GPU communication is included.

All scenarios using F/E bits perform as well or better than the GPU baseline case. Of the different strategies, the full-overlap scenario provides an average of 26% speedup.

The overlap-start and overlap-end scenarios are each frequently helpful, but show variability in performance gain across the different benchmarks. In `histogram64`, for example, overlap-start provides a significant speedup, while overlap-finish provides little. Conceptually, the input data can be sent to the correct histogram bin at any time, while the total histogram results are not final until all input data has been processed. On the other hand, for `matrixMul` overlap-finish provides slightly better gains than overlap-start. Similarly, the benefits of full-overlap over the individual components varies by benchmark. For `histogram`, most gains come from overlap-start, while for `vectorAdd` the benefit goes beyond what either of the two components provides independently.

7.2. Dataset Size and CPU-GPU Tradeoffs

In addition to the excellent performance gains F/E bits offer, we also want to understand how input size influences the GPU vs. CPU performance tradeoffs. To study this, Figure 12 shows the performance of `vectorAdd` for a variety of vector sizes. All runtimes are compared to a single-threaded CPU baseline[2]. First, considering the GPU baseline curve (no F/E bits), the CPU is the faster choice for all vector sizes smaller than about 128K, and the GPU is the better choice for larger vectors. However, if our proposals for fine-grained synchronization are adopted, the crossover point changes dramatically: the GPU with full-overlap becomes more efficient at 32K element vectors. Consequently, the breakeven point where GPUs are beneficial has shifted dramatically towards smaller data sets. In addition to raw performance improvements, this ability to be beneficial across a wide range of applications and data is another important benefit of our proposed approach.

8. Related Work

As GPUs have come into more general use [37] since projects such as Brook [7] demonstrated their potential, research on them has included some efforts related to our proposal. For example, prior work has considered improving GPU synchronization via atomic operations [13, 36, 41]. Others have proposed GPU architectural support for precise exceptions and speculative execution [28] and for GPU programming models using persistent threads [17]. The proposed techniques are not, however, data-oriented like our F/E bits. As a result, they are not as effective for kernels with varying fetch latencies. Dymaxion [9] overlapped computation and communication by breaking data transfers and kernels into overlapping chunks; however, their technique works only for purely data parallel kernels which can be easily divided, and it does not address GPU-to-CPU communication.

[2] In our experiments, parallelization of the CPU code with OpenMP was slower due to the overhead of thread creation.

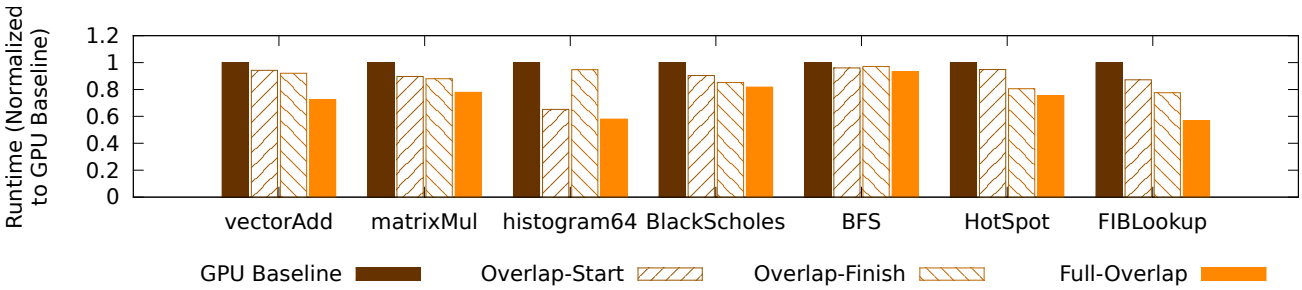


Figure 11: Normalized runtime of various benchmarks using full/empty bits instead of global synchronization. (Lower is better.)

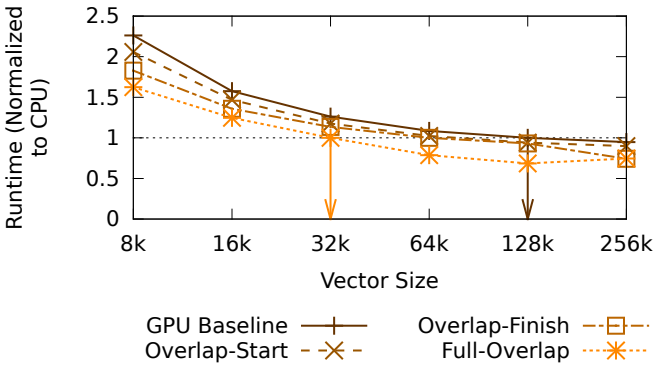


Figure 12: Normalized runtime of `vectorAdd` for different input set sizes. (Lower is better.) Exploiting F/E bits changes the breakeven point between CPU and GPU implementations. Previously, the GPU began outperforming CPU at 128K, but with F/E bits, it begins outperforming at 32K element vectors. This broadens the space where GPUs are beneficial, and provides a new best option for all input sizes above 32K.

Previous work has noted the importance of CPU-GPU communication time when measuring GPU performance, [16, 27], but little attention has gone to simulator models and their validation. FusionSim [43] models a combined CPU-GPU system, using GPGPU-Sim as its GPU model and PTLSim [42] as its CPU model, but has a simpler interconnect model and less real-system validation. MacSim [25] simulates a heterogeneous CPU-GPU system, but uses trace-based simulation, which does not capture CPU-GPU communication well. Multi2Sim [39] simulates both the CPU and the GPU, but CPU-GPU communication is simulated functionally, while we provide a detailed timing model.

F/E bits have long been studied in non-GPU scenarios. For example, the MIT Alewife processor [1] contained a F/E bit per 32-bit word in memory, while the Multi-ALU Processor (MAP) [24] placed F/E bits in the register file. The Heterogeneous Element Processor (HEP) [35] uses a single full/empty bit for each register and memory word, while the Message-Driven Processor (MDP) [11] marks each 32b word in registers and memory with a 4b tag, which acts as either a full/empty bit or a type indicator. The Tera/Cray MTA [2] and Eldorado [14] computers contained a set of four synchronization bits per 64-bit word in memory. The first two, a F/E bit and a trap bit used as support, indicated whether data was present or not. A second trap bit marked unallocated words to help in debugging. Finally, a forwarding bit indicated that the data line itself contained an address. On accesses to lines that were not ready, the architecture would retry the access a number of times before passing control to a trap handler. More recently, the Godson-T processor [12] uses full/empty bits along with a hardware data transfer agent as part of an efficient mechanism for transferring data between its cores. Finally, we note that other throughput-oriented approaches like the Cell processor [33] might benefit from F/E bits in addition to their DMA support.

Full/empty bits have been incorporated in other ways as well. One study integrated F/E bits with cache coherence to support fine-grained synchronization for shared memory multiprocessors [40]. Others have focused on algorithms research using F/E bits [5]. Our work takes F/E bits to GPUs, with the specific goal of improving performance and CPU-GPU overlap. In the GPU arena, the only prior use of F/E bits was as part of a framebuffer proposal, in which they were used for reliability rather than performance enhancements [34].

9. Discussion

Our study of fine-grained synchronization for GPUs established the large (26%) payoff for F/E bits and related API changes. Given the large payoff, there are further tradeoffs to explore in future work. For example, adjusting the granularity at which data is tracked and transmitted will allow architects to find the best balance between hardware overhead and performance. Applications like `BFS` benefit from fine granularity to properly handle less-predictable data structures. In fact, graph and sparse matrix calculations of this type, in addition to scientific applications, are seeing increasing commercial usage. On the other hand, for dense numeric code, associating F/E bits with larger blocks of addresses might be sufficient, as each individual piece of data would only briefly be delayed waiting for the rest of its block to finish. Similarly, the total size of the memory space for which data is tracked with F/E bits will influence the placement of the bits themselves. If the total storage required is small enough, it may become beneficial to move the F/E bits into the memory controller itself in exchange for being able to track a smaller space.

It is also critical to understand the best interface to these software bits from software. Our work used a simple host API interface while leaving the F/E bit triggers and actions fixed for GPU code. Additional benefits might come from giving the

GPU more detailed control of the F/E bits, thereby enabling an even more customizable CPU-GPU producer-consumer relationship. In addition, a further application-driven study of possible override conditions may help more benchmarks make better use of them.

Finally, our results serve as a reminder that the communication between CPU and GPU is in fact a critical component of the runtime of GPU workloads. As the architectures of both discrete and integrated GPUs continue to evolve rapidly, it is crucial that both the latency and the throughput behavior of the chips be taken into account. This will enable GPUs be used to speed up the performance of a wide range of workloads.

10. Conclusion

Overall, our work makes several important research contributions. First, we demonstrate that F/E bits and related APIs can have substantial performance impact on a range of applications. Across our suite of latency-sensitive kernels, we measure improvements of 26% on average, and 43% in the best case. More broadly, by reducing the data transfer and launch overheads associated with GPU offloading, our work has the potential to considerably broaden the space of applications that can benefit from GPU acceleration. For example, in a `vectorAdd` kernel, our techniques shifted the GPU breakeven point from vectors of 128K elements down to vectors of 32K elements. This increases the applicability of GPU acceleration and makes GPU adoption less likely to lead to performance surprises. Finally, our work is underpinned by detailed simulations and real-system measurements. Our experimental infrastructure can be valuable to other researchers on similar topics and is publicly available.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0627650 and CNS-07205661. The authors also acknowledge the support of the Gigascale Systems Research Center, one of six centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. Finally, we acknowledge the support of a research gift from Intel Corp.

References

[1] A. Agarwal *et al.*, "The MIT Alewife Machine: Architecture and Performance," *ISCA*, 1995.

[2] G. Alverson *et al.*, "Tera Hardware-Software Cooperation," *SC*, 1997.

[3] AMD, "Memory System on Fusion APUs." Available: http://developer.amd.com/afds/assets/presentations/1004_final.pdf

[4] AMD, "The Programmer's Guide to a Universe of Possibility," *AFDS Keynote*, 2012. Available: http://hsafoundation.com/publications

[5] D. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-Connectivity on the Cray MTA-2," *ICPP*, 2006.

[6] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," *ISPASS*, 2009.

[7] I. Buck *et al.*, "Brook for GPUs: Stream Computing on Graphics Hardware," *SIGGRAPH*, 2004.

[8] S. Che *et al.*, "Rodinia: a Benchmark Suite for Heterogeneous Computing," *IISWC*, 2009.

[9] S. Che, J. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," *SC*, 2011.

[10] M. Daga, A. Aji, and W.-C. Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing," *SAAHPC*, 2011.

[11] W. Dally *et al.*, "The Message-Driven Processor: a Multicomputer Processing Node with Efficient Mechanisms," *Micro, IEEE*, vol. 12, no. 2, 1992.

[12] D.-R. Fan *et al.*, "Godson-T: An Efficient Many-Core Architecture for Parallel Program Executions," *J. Comp. Sci. and Tech.*, vol. 24, 2009.

[13] W.-C. Feng and S. Xiao, "To GPU Synchronize or Not GPU Synchronize?" *ISCAS*, 2010.

[14] J. Feo *et al.*, "Eldorado," *Comp. Frontiers*, 2005.

[15] Green500, "Green500 List," November 2011. Available: www.green500.org

[16] C. Gregg and K. Hazelwood, "Where Is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer," *ISPASS*, 2011.

[17] K. Gupta, J. Stuart, and J. Owens, "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads," *InPar*, 2012.

[18] S. Han *et al.*, "PacketShader: a GPU-Accelerated Software Router," *SIGCOMM*, 2010.

[19] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," *HiPC*, 2007.

[20] W. Huang *et al.*, "HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design," *VLSI*, 2006.

[21] Intel, "Ivy Bridge." Available: http://ark.intel.com/products/codename/29902/Ivy-Bridge

[22] Intel, "The Intel Xeon Phi Coprocessor 5110P: Highly-parallel Processing for Unparalleled Discovery," 2012. Available: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

[23] W. Jia, K. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-Fetched Caches in GPUs," *ICS*, 2012.

[24] S. Keckler *et al.*, "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," *ISCA*, 1998.

[25] H. Kim *et al.*, "MacSim: Simulator for Heterogeneous Architecture," 2012. Available: http://code.google.com/p/macsim

[26] S. Larsen *et al.*, "Architectural Breakdown of End-to-End Latency in a TCP/IP Network," *IJPP*, 2009.

[27] V. W. Lee *et al.*, "Debunking the 100x GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *ISCA*, 2010.

[28] J. Menon, M. de Kruijf, and K. Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," *ISCA*, 2012.

[29] D. Miller, P. Watts, and A. Moore, "Motivating Future Interconnects: a Differential Measurement Analysis of PCI Latency," *INCS*, 2009.

[30] NVIDIA, "CUDA C Programming Guide." Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[31] NVIDIA, "Cuda Toolkit 4.2." Available: http://developer.nvidia.com/cuda-downloads

[32] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110." Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[33] D. Pham *et al.*, "The Design and Implementation of a First-generation CELL Processor," *ISSCC*, 2005.

[34] J. Sheaffer, D. Luebke, and K. Skadron, "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors," *Graph. HW.*, 2007.

[35] B. J. Smith, "Architecture and Applications of the HEP multiprocessor Computer System," 1982.

[36] J. A. Stuart and J. D. Owens, "Efficient Synchronization Primitives for GPUs," *arXiv CoRR*, vol. abs/1110.4623, no. 1110.4623v1, Oct. 2011.

[37] S. Suneja *et al.*, "Accelerating the Cloud with Heterogeneous Computing," *HotCloud*, 2011.

[38] TOP500 Project, "TOP500 List," November 2012. Available: www.top500.org

[39] R. Ubal *et al.*, "Multi2Sim: a Simulation Framework for CPU-GPU Computing," *PACT*, 2012.

[40] V. Vlassov *et al.*, "Support for Fine-Grained Synchronization in Shared-Memory Multiprocessors," in *Parallel Computing Technologies*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4671, pp. 453–467. Available: http://dx.doi.org/10.1007/978-3-540-73940-1_45

[41] S. Xiao and W.-C. Feng, "Inter-Block GPU Communication via Fast Barrier Synchronization," *IPDPS*, 2010.

[42] M. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," *ISPASS*, 2007.

[43] V. Zakharenko, "FusionSim Simulator," 2012. Available: www.fusionsim.ca