Overlapping Host-to-Device Copy and Computation using Hidden Unified Memory

Anonymous Author(s)

Abstract

In this paper, we propose a runtime, called HUM, which hides host-to-device memory copy time without any code modification. It overlaps the host-to-device memory copy with host computation or CUDA kernel computation by exploiting Unified Memory and fault mechanisms. HUM provides wrapper functions of CUDA commands and executes host-to-device memory copy commands in an asynchronous manner. We also propose two runtime techniques. One checks if it is correct to make the synchronous host-to-device memory copy command asynchronous. If not, HUM makes the host computation or the kernel computation waits until the memory copy completes. The other subdivides consecutive host-to-device memory copy commands into smaller memory copy requests and schedules the requests from different commands in a round-robin manner. As a result, the kernel execution can be scheduled as early as possible to maximize the overlap. We evaluate HUM using 51 applications from Parboil, Rodinia, and CUDA Code Samples and compare their performance under HUM with that of hand-optimized implementations. The evaluation result shows that executing the applications under HUM is, on average, 1.21 times faster than executing them under original CUDA. The speedup is comparable to the average speedup 1.22 of the hand-optimized implementations for Unified Memory.

Keywords GPU, CUDA, Unified memory, Runtime, Data transfer and computation overlap, Device driver

1 Introduction

Heterogeneous computing uses different types of processors together to gain performance and energy efficiency. The processors include CPUs, GPUs, FPGAs, DSPs and accelerators of other types. GPU is one of the most popular accelerators and many programming models have been proposed to use it efficiently. CUDA[24] is one of the popular programming models for GPUs.

CUDA Unified Memory (UM) is a memory pool that has a single address space and can be accessed by both the host and the GPU[29]. A UM object is allocated by invoking `cudaMallocManaged()` in a CUDA program. When UM is used, a CUDA program does not need to explicitly move data between the host and the device. In other words, there is no need to use `cudaMemcpy()` or `cudaMemcpyAsync()` in the CUDA program. The UM system exploits the page fault engine in the GPU[27], and it automatically migrates accessed pages between the host and the GPU. UM significantly lessens the burden of a programmer to manage data distribution across the host and the GPU. However, using UM solely does not guarantee good performance. To fully exploit UM and improve performance, the programmer needs to add user hints to the source code to prefetch pages that are going to be accessed during the kernel execution. For example, to give the user hint, the programmer manually inserts `cudaMemPrefetchAsync()` before the kernel is executed to prefetch memory to a specified destination GPU.

By exploiting CUDA UM and fault mechanisms in both the CPU and the GPU, overlapping data transfers and computation can be well controlled. This can be a solution to one of the major challenges in heterogeneous computing: hiding the memory transfer time between the host and the device as much as possible. In this paper, we propose a runtime, called *HUM* (Hidden Unified Memory), as a solution of this problem. It automatically hides the *host-to-device memory copy* (in short, *H2Dmemcpy* hereafter) time by overlapping it with *host computation* or *kernel computation*.

Here, the host computation is the execution of the host code that does not depend on H2Dmemcpy commands. It includes CPU computation, host memory allocation/deallocation, file I/O, etc. To copy data from the host memory to the device memory, CUDA provides both synchronous (blocking) commands (*e.g.*, `cudaMemcpy()`) and asynchronous (non-blocking) commands (*e.g.*,`cudaMemcpyAsync()`). The asynchronous function call is asynchronous with respect to the host, hence the call may return before the copy completes while the synchronous function call returns after the copy has completed.

Overlapping H2Dmemcpy and host computation. For the best application performance, the programmer is recommended to use asynchronous memory copy commands to perform useful CPU tasks in parallel with the memory copy. However, it is difficult for the programmer to safely replace a synchronous memory copy command with an asynchronous one. By exploiting UM and the page fault engine, when a H2Dmemcpy command is synchronous, HUM makes it non-blocking (asynchronous). As a result, the H2Dmemcpy and some CPU computation are overlapped. To guarantee safety for the overlap, HUM exploits the segmentation fault mechanism in the host side at run time. When the host tries

PPoPP'20, February 22–26, 2020, San Diego, California, USA
2020. ACM ISBN …$15.00
https://doi.org/

to access a page in the source host memory object of the H2Dmemcpy command that has not been copied to the device memory space yet, a segmentation fault occurs. HUM catches it and makes the host wait until the H2Dmemcpy operation on that page completes. Then, the host side computation continues.

Overlapping H2Dmemcpy and kernel computation. While UM supports automatic overlapping of kernel execution and data transfers between the host and the device, it requires a programmer to explicitly use UM allocation commands (*e.g.*, `cudaMallocManaged()`) in the source code. In addition, to fine-tune the transfers, user hints (*e.g.*, `cudaMemPrefetchAsync()`) are also required. However, HUM does not require any explicit UM command and user hint. By exploiting the GPU page fault mechanism and UM, HUM automatically overlaps the H2Dmemcpy and the kernel computation without regards to if the copy command is synchronous or asynchronous. Even if the copy command is asynchronous, it is still beneficial to use HUM for performance.

To the best of our knowledge, HUM is the first work that automatically hides the H2Dmemcpy time by overlapping it with host computation or kernel computation without any explicit UM command and any modification of the source code. Major contributions of this paper are summarized as follows:

- We propose a runtime, called HUM, which exploits CUDA UM and fault mechanisms of both the host and the GPU. It automatically hides the H2Dmemcpy time by overlapping it with the host or kernel computation. We describe its design and implementation.
- We propose a runtime technique that exploits the host side page protection mechanism and checks if it is correct to make a synchronous H2Dmemcpy command asynchronous.
- We propose a runtime technique that subdivides consecutive H2Dmemcpy commands into smaller memory copy requests and executes the requests from different commands in a round-robin manner. As a result, the kernel execution can be scheduled as early as possible to maximize its overlap with the H2Dmemcpy commands.
- We evaluate HUM using 51 CUDA benchmark applications from Parboil[34], Rodinia[3], and CUDA Code Samples[23]. The evaluation result shows that executing the applications under HUM is, on average, 1.21x faster than executing them under original CUDA. The speedup is comparable to the average speedup of 1.22 that is obtained by manually porting and optimizing the applications with Unified Memory.

2 Related Work

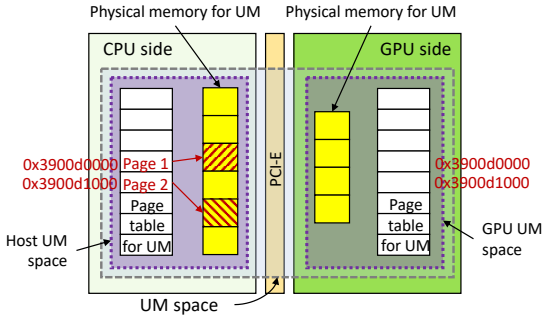
There are some previous studies related to CUDA Unified Memory (UM)[1, 19, 20]. Landaverde et al.[19] and Li et al.[20] evaluate the performance of UM using Parboil[34] and Rodinia[3] benchmark suites. As Parboil and Rodinia do not provide UM version, they make the UM version of the benchmark applications on their own and compare their performance with the existing non-UM version. The results from both studies show that the UM version is slower than the non-UM version. One of the reasons of the slowness is that both studies use GPUs of the NVIDIA Kepler architecture[21] that does not fully support UM. The NVIDIA Pascal architecture[27] and its successors[22, 28] fully support UM. The Kepler architecture is two generations earlier than the NVIDIA Pascal architecture. Kepler architecture does not support GPU page faults. Moreover, in the Kepler architecture, all pages in the host UM space have to be migrated to the GPU UM space before a kernel is executed even if some pages are not actually accessed by the kernel.

Awan et al. propose OC-DNN[1] that exploits UM on GPUs of NVIDIA Pascal and Volta architectures[22, 27]. They port one of the well-known DNN frameworks, Caffe[32], to UM and optimize it manually by adding various CUDA user-hint API functions. OC-DNN provides comparable performance to Caffe for popular Deep Neural Networks (DNNs), such as AlexNet[18], GoogLeNet[35], VGG-19[33], and ResNet-50[13]. HUM exploits CUDA UM to automatically overlap the host-to-device memory copy and the computation of the host or the device without exposing UM to the programmer and without hurting performance.

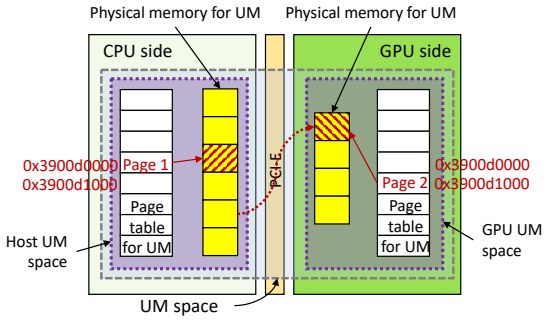
Many studies have been performed to detect memory reuse[4, 14, 15, 37]. All these previous studies focus on data-reuse analysis at compile time. Cong et al.[4] and Issenin et al.[14, 15] statically analyze data reuse and try to hide memory latency by placing frequently reused data in scratchpad memory. HUM is different from those previous approaches in that it detects modifications to previously defined data. Moreover, it performs the detection at run time and exploits the segmentation fault mechanism.

Many techniques for overlapping host-GPU data transfers and GPU kernel computation have been proposed[2, 9, 16, 17, 30, 31]. While they require a user to manually overlap the data transfers and the kernel computations, our framework automatically does it without any code modification.

Overlapping communication and CPU/GPU computation in a cluster has also been widely studied[5–7, 10, 36]. White III and Dongarra[36] show the effect of overlapping CPU/GPU computation, inter-node communication, and CPU-GPU communication. Danalis et al.[5], Fishgold et al.[7], and Danalis et al.[6] introduce compiler techniques that transform MPI code to overlap inter-node communication and CPU computation. Gysi et al.[10] propose a framework that automatically overlaps inter-node communication and GPU computation. HUM focuses on automatic overlapping of data transfers and GPU computation in a node by exploiting Unified Memory.



(a) After the host has accessed page 1 and page 2.



(b) After the GPU has accessed page 2.

Figure 1. CUDA unified memory.

3 CUDA Unified Memory

CUDA Unified Memory (UM) provides ease-of-programming by enabling CUDA programs to access the host memory and the GPU memory without the need to manually copy data from one to the other. UM behaves as if the programmer had a single address space between the host and the GPU[12]. It allows a CUDA application to allocate memory objects that can be read or written from both the host and the GPU. The NVIDIA Pascal architecture and later NVIDIA GPU architectures fully support UM.

As shown in Figure 1(a), physical memory spaces are allocated to UM in both the host side and the GPU side. Pages in the host side space are pinned. UM page tables in the host side and the GPU side are managed by the CUDA runtime. To allocate a UM object, the CUDA program invokes `cudaMallocManaged()`, an allocation function that returns a pointer to the memory object. The pointer is accessible from both the host and the GPU. However, the memory object may not be physically allocated when the call to `cudaMallocManaged()` returns. In other words, the pages and page table entries of the memory object may not be created until it is accessed by the GPU or the CPU.

Pages in a UM object are automatically migrated between the host and the GPU on demand. This automatic page migration exploits page faults. The host reads and writes pages in the host memory and the GPU reads and writes pages in the device memory. The CUDA runtime takes care of the page migration, hence there is no need to call `cudaMemcpy()` or `cudaMemcpyAsync()` at all.

For example, suppose that a UM object has been allocated by `cudaMallocManaged()` and that the host has accessed two pages of the object, page 1 (at virtual address `0x3900d0000`) and page 2 (at virtual address `0x3900d1000`). Figure 1(a) shows the current status of page tables and physical memory spaces of UM. Now, suppose that the GPU accesses page 2 at virtual address `0x3900d1000`. Since page 2 is not residing in the GPU side, a page fault occurs and a page fault interrupt signal is raised. The page fault is handled by the NVIDIA display driver. It catches the signal and migrates the faulted page, page 2, between the host UM space and the GPU UM space as shown in Figure 1(b). Then, it makes the GPU replay the access. To avoid excessive page faults, the NVIDIA driver uses some heuristics for the page migration[12].

The GPU memory can be oversubscribed with UM allocations, *i.e.*, if allocated by `cudaMallocManaged()`. The size of oversubscription is typically limited to the size of the host physical memory. Another benefit of using UM is that it guarantees data consistency between the host memory and the GPU memory.

HUM exploits the same page fault mechanism to detect the case when a CUDA kernel accesses a UM page that has not been transferred from the host UM space to the GPU UM space.

Table 1. Representative CUDA commands used in this paper.

`cudaError_t cudaMalloc(void** devPtr, size_t size)` allocates `size` bytes on the device and then returns in `*devPtr` a pointer to the allocated memory. It is a synchronous function.
`cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)` copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy. We are interested in `cudaMemcpyHostToDevice` as the value of `kind` in this paper.
`cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, ...)` behaves the same as `cudaMemcpy()` except that it is asynchronous with respect to the host.
`cudaError_t cudaMallocManaged(void** devPtr, size_t size, ...)` allocates `size` bytes on the device and returns in `*devPtr` a pointer to the allocated memory that is automatically managed by the UM system.
`cudaError_t cudaMemPrefetchAsync(const void* devPtr, size_t count, int dstDevice, ...)` prefetches UM memory to the specified destination device. `devPtr` is the base pointer of the UM memory space to be prefetched and `dstDevice` is the destination device. `count` specifies the number of bytes to prefetch. It is asynchronous with respect to the host.

Representative CUDA commands[25] used in this paper are summarized in Table 1. The way of handling other CUDA memory management commands by HUM is similar to the way of handling those listed above.

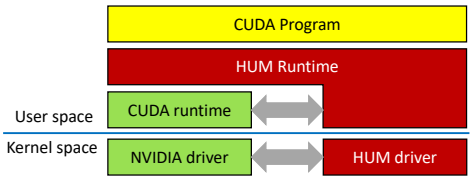


Figure 2. Components of HUM.

4 Design and Implementation

In this section, we present the design and implementation of HUM. HUM exploits the page fault mechanism of UM to automatically overlaps *host-to-device memory copy (H2Dmemcpy) and host computation* or *H2Dmemcpy and kernel computation* without any code modification.

As shown in Figure 2, HUM consists of two components: *HUM runtime* and *HUM driver*. The NVIDIA driver[26] is a part of the CUDA framework that bridges the CUDA runtime and NVIDIA GPUs. It resides in the kernel address space. Similar to the NVIDIA driver, the HUM driver resides in the kernel address space. It intercepts signals going into the NVIDIA driver and takes some actions. Then, it calls appropriate NVIDIA driver functions for the signals if needed. The HUM runtime provides wrapper functions of CUDA API functions and interacts with the HUM driver and the CUDA runtime.

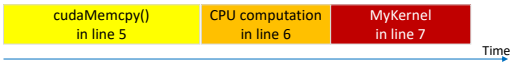
In CUDA, a stream is a sequence of commands that execute in issue-order on the GPU[11]. Commands in different streams may execute out of order with respect to one another or concurrently. When a CUDA program generates a request to create a new stream, the HUM runtime creates a stream object that is a wrapper of a new CUDA stream and provides it to the CUDA program. The HUM stream object is managed by HUM, and a host thread in the HUM runtime, called the *command scheduler*, periodically visits all existing streams in a round-robin manner. The HUM runtime also has several *worker threads*. When the command at the front of each stream is ready to execute, the command scheduler takes it from the stream and dispatches it to a worker thread. The worker thread executes the command (note that the command is actually the wrapper function of a CUDA command) and enqueues the CUDA command to the CUDA stream managed by the CUDA runtime. Finally, the CUDA runtime executes the command.

4.1 Overlapping H2Dmemcpy and Computation

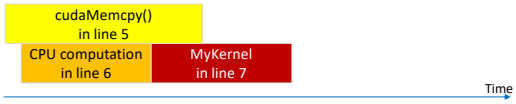
Synchronous H2Dmemcpy. Figure 3 shows some examples of the memory copy commands. In Figure 3(a), the CUDA program allocates a host memory space, say *hA*, pointed to by `hostA` using `malloc()` in line 2 and a device memory space, say *dA*, pointed to by `devA` in line 3. It writes some data to *hA* in line 4. Then, it copies the contents of *hA* to *dA* by invoking synchronous `cudaMemcpy()` in line 5. After the memory copy has completed and some host computation has been performed in line 6, a kernel `MyKernel` that accesses *dA* is launched in line 7. Figure 3(b) shows the timeline of executing the code in Figure 3(a) under CUDA.

```
1: ...
2: hostA = malloc(size);
3: cudaMalloc(&devA, size);
4: ... // write to hostA
5: cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);
6: ... // some CPU computation
7: MyKernel<<<...>>>(devA);
8: ...
```

(a) Overlapping H2Dmemcpy and CPU computation and overlapping H2Dmemcpy and kernel computation.



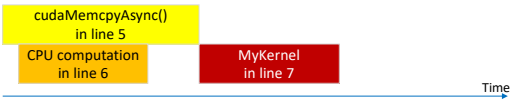
(b) Executing the code in (a) under CUDA.



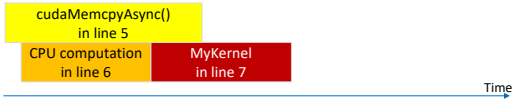
(c) Executing the code in (a) under HUM.

```
1: ...
2: hostA = malloc(size);
3: cudaMalloc(&devA, size);
4: ... // write to hostA
5: cudaMemcpyAsync(devA, hostA, size, cudaMemcpyHostToDevice);
6: ... // some CPU computation
7: MyKernel<<<...>>>(devA);
8: ...
```

(d) Overlapping H2Dmemcpy and kernel computation.



(e) Executing the code in (d) under CUDA.



(f) Executing the code in (d) under HUM.

Figure 3. Overlapping H2Dmemcpy and computation.

When the same code is executed under HUM, `cudaMemcpy()` returns immediately after initiating the memory copy even though the copy has not completed. This enables overlapping the memory copy in line 5 and the host computation in line 6. It may further overlaps the memory copy in line 5 and the kernel execution in line 7. Figure 3(c) shows the timeline of executing the code in Figure 3(a) under HUM. Compared to the timeline under CUDA in Figure 3(b), `cudaMemcpy()` in line 5 is fully overlapped with the CPU computation in line 6 and partially

overlapped with the kernel computation in line 7. As a result, the total execution time is significantly reduced.

Even though the kernel starts its execution before the memory copy in line 5 completes, the kernel correctly executes under HUM. The reason is that a page fault is raised at the device side when the kernel accesses a page that has not been copied yet to the device side. The page fault is handled by the HUM driver and it makes the kernel waits until the faulted page is copied to the device side. Then the page access request from the kernel is replayed.

However, if the host computation in line 6 modifies *hA*, the memory copy in line 5 and the host computation in line 6 may not be overlapped to guarantee data consistency and correctness. In this case, the timeline of executing the code in Figure 3(a) under HUM is the same as that under CUDA in Figure 3(b). The HUM runtime detects such a case using a simple runtime technique. The technique will be described later in Section 4.2.

Asynchronous H2Dmemcpy. In Figure 3(d), the CUDA program calls asynchronous `cudaMemcpyAsync()` in line 5, hence the memory copy is performed in the background. As a result, the host side computation in line 6 can be overlapped with the memory copy in line 5. However, the kernel launched at line 7 cannot be overlapped with the memory copy in line 5 because all tasks placed in one stream are executed sequentially (the default behavior of CUDA). Figure 3(e) shows the timeline of executing the code in Figure 3(d) under CUDA.

When the same code is executed under HUM, even if the asynchronous memory copy in line 5 has not finished yet, the GPU may start executing the kernel in line 7. This enables overlapping the H2Dmemcpy and the kernel computation for the same reason as the case of overlapping the synchronous H2Dmemcpy and computation mentioned above. Figure 3(f) shows the timeline of executing the code in Figure 3(d) under HUM. As a result, we see that the total execution time is significantly reduced.

Note that even though the HUM runtime overlaps the H2Dmemcpy and the host or kernel computation, it preserves the CUDA semantics of synchronization commands, such as `cudaDeviceSynchronize()`. `cudaDeviceSynchronize()` in the HUM runtime is also a wrapper function and invokes the original CUDA command.

4.2 Data Consistency and Correctness

Consider the CUDA program in Figure 4. After performing `cudaMemcpy()` to copy the contents of the memory object, say *hA*, pointed to by `host_A` to the device memory object, say *dA*, pointed to by `dev_A` in line 5, the program modifies the contents of *hA* or frees *hA* in line 7. Under the CUDA semantics, this program has no problem at all. However, it may cause a problem under HUM. The data transfer caused by `cudaMemcpy()` to the device may still continue when the

```
01: ...
02: host_A = malloc(size);
03: cudaMalloc(&dev_A, size);
04: ... // write to host_A
05: cudaMemcpy(dev_A, host_A, size, cudaMemcpyHostToDevice);
06: ...
07: ... // write to host_A or free host_A
08: ...
09: MyKernel<<<...>>>(dev_A);
10: ...
```

Figure 4. A problematic scenario.

contents of *hA* is modified in line 7. Thus, the device may receive some pages that contain the modified contents. As a result, the kernel may access inconsistent and incorrect data.

To solve this problem, the HUM runtime exploits the access protection of pages using a POSIX function `mprotect()`[8] that changes the access protection of the memory pages of the calling process. When the H2Dmemcpy caused by `cudaMemcpy()` or `cudaMemcpyAsync()` is initiated, the HUM runtime changes the protection of pages in the source host memory object to *read-only*. For example, the protection of the pages in the object pointed to by `host_A` in Figure 4 is changed to *read-only* when the H2Dmemcpy of `cudaMemcpy()` is initiated.

When the CUDA program in Figure 4 modifies a page in *hA* in line 7 in a manner (*e.g.*, write) that violates the protection, the linux kernel generates a SIGSEGV signal. The signal handler installed by the HUM runtime handles the signal. When it receives the signal, it waits until the memory copy for the page completes. After completion, it restores the protection of the page in *hA* to *writable*. Then, the modification to the page in *hA* starts. This method allows the HUM runtime to execute H2Dmemcpy commands in an asynchronous manner without any data consistency violation or any segmentation fault.

4.3 HUM Driver

Intercepting interrupts. To overlap H2Dmemcpy and kernel execution, HUM makes the GPU pend when the page accessed by the GPU has not been transferred to the GPU yet. In this case, a GPU page fault occurs in HUM. The HUM driver handles the page fault. The HUM driver hooks the interrupt handler of the NVIDIA display driver and intercepts the page fault signal. In Linux for the x86 architecture, the *interrupt descriptor table* (IDT) contains all information about interrupts, such as interrupt number, interrupt name, address of the interrupt handler, interrupt flags, etc. When the HUM driver is installed, HUM looks up the existing IDT entries and finds the entry for the NVIDIA interrupt handler. HUM replaces the entry with the information of its own interrupt handler.

Handling page faults. Figure 5 shows the actions occurring when the HUM interrupt handler handles a page fault in the GPU side. When the HUM interrupt handler receives an

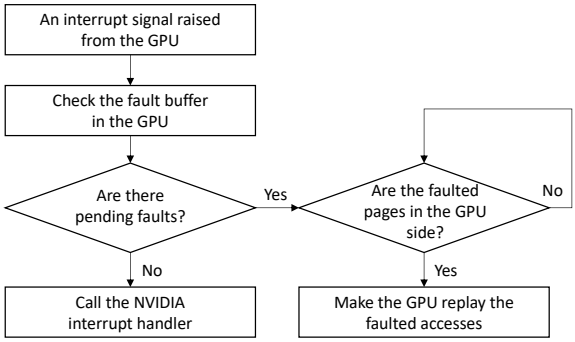


Figure 5. Actions of the HUM interrupt handler.

interrupt signal, it checks the fault buffer in the GPU if there is a pending GPU page fault. The fault buffer is a circular queue implemented in the GPU by NVIDIA. It stores page faults information from the GPU. If there is no pending fault in the fault buffer, the HUM interrupt handler invokes the original NVIDIA interrupt handler because the interrupt is not a page fault and there is nothing to do for the HUM interrupt handler. Otherwise, if there are pending faults in the fault buffer, the HUM interrupt handler waits until all the faulted pages arrive and are mapped to the GPU. Then, the HUM driver sends a replay signal to the GPU so that the GPU replay the faulted memory accesses.

4.4 HUM H2Dmemcpy Mechanism

When the GPU accesses a page that has not been copied from the host side to the GPU side, the HUM runtime makes the GPU waits until the page arrives. As a result, a kernel can be executed even the transfer of the data to be accessed by the kernel is still ongoing. However, to implement the H2Dmemcpy in HUM, we may not use `cudaMemcpy()` and `cudaMemcpyAsync()` because they cause a serious interrupt handling problem.

Problems of CUDA memory copy commands. For example, suppose that the HUM driver uses `cudaMemcpy()` to copy data from the host to the device and that the GPU is trying to read a page that has not been yet copied to the GPU side. Then, a read page fault is raised and the HUM driver catches it. The HUM driver waits until the page comes to the GPU side. When the page arrives, calling `cudaMemcpy()` triggers a write page fault because the page has not been mapped to the GPU yet. The HUM driver catches the write page fault and maps a blank page to the GPU UM space. Then, it sends a replay signal to the GPU. This makes the GPU reads stale data in the blank page. In turn, the page arrived updates the GPU UM space. Since interrupts caused by memory requests are processed sequentially one by one in the GPU, the kernel reads the stale data in the blank page first, and the page update by the memory copy follows this read. To get the correct result, the memory copy should have completed before the kernel reads the stale page. However,

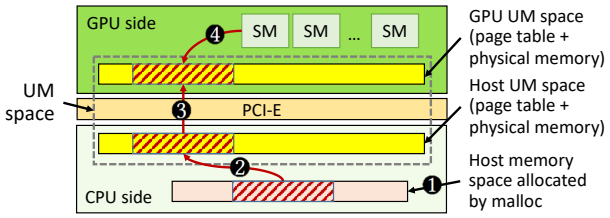


Figure 6. How the HUM H2Dmemcpy function works.

changing the order of interrupt processing is not supported by the current NVIDIA driver.

HUM H2Dmemcpy functions. To solve this problem, the HUM driver has its own H2Dmemcpy function. Figure 6 shows how the HUM H2Dmemcpy function works. A CUDA program first writes data to the host memory space that is generally allocated through `malloc()` (❶). Suppose that the program uses `cudaMemcpy()` or `cudaMemcpyAsync()` to perform the H2Dmemcpy. As mentioned before, the HUM runtime implements wrappers of `cudaMemcpy()` and `cudaMemcpyAsync()`. In the wrappers, the HUM runtime calls the HUM driver rather than calling the original CUDA `cudaMemcpy()` or `cudaMemcpyAsync()`.

The HUM driver first copies the data from the host memory space to the host UM space (❷ in Figure 6). Then, it invokes the page migration function provided by the NVIDIA driver to migrate the pages in the host UM space to the GPU UM space (❸). To use the migration function, source pages of the migration must reside in the host UM space. The page migration function is synchronous and migrates maximum 512 pages at a time, *i.e.*, maximum 2 MB at a time. When the migration completes, the pages are mapped to the GPU, and the GPU can access the pages without any page fault (❹).

When there is a H2Dmemcpy request of size M MB ($M > 2$), the HUM driver divides the request into multiple requests of size 2 MB. We take the maximum size because frequent memory-copy requests cause heavy copy initiation overhead.

```
01: ...
02: host_A = malloc(size);
03: host_B = malloc(size);
04: host_C = malloc(size);
05: ... // write to host_A and host_B
06: cudaMalloc(&dev_A, size);
07: cudaMalloc(&dev_B, size);
08: cudaMalloc(&dev_C, size);
09: ...
10: cudaMemcpyAsync(dev_A, host_A, size,
11:                 cudaMemcpyHostToDevice);
12: cudaMemcpyAsync(dev_B, host_B, size,
13:                 cudaMemcpyHostToDevice);
14: ...
15: vec_add<<<...>>>(dev_A, dev_B, dev_C);
16: ...
```

Figure 7. Vector addition program.

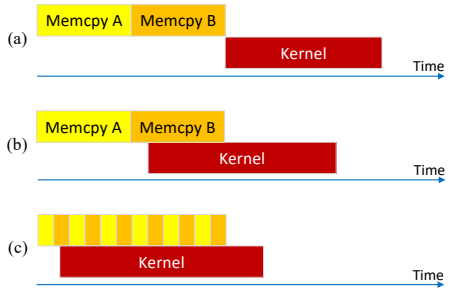


Figure 8. Executing the vector addition program in Figure 7. (a) Under normal CUDA semantics. (b) Under HUM discussed so far. (c) Optimization under HUM.

4.5 Parallelizing Memory Copy Commands

Consider a vector addition CUDA program in Figure 7. It adds two vectors A and B, and the result is stored in vector C. Figure 8 shows timelines of executing the program. Since the memory copy command and the kernel execution command are issued in the same stream to guarantee correctness, they are sequentially executed as shown in Figure 8(a) under CUDA semantics.

The timeline of executing the vector addition program under the HUM design discussed so far is shown in Figure 8(b). HUM may execute the kernel as early as possible when the memory copy for vector B has initiated. As a result, the time when the kernel completes under HUM maybe much earlier than that under normal CUDA. Since the page migration function provided by NVIDIA driver used in the HUM H2Dmemcpy function is synchronous, the memory copy for vector B has to be initiated after the memory copy for vector A has completed.

Using the HUM H2Dmemcpy function, the time spent on memory copying is much larger than using `cudaMemcpy()` or `cudaMemcpyAsync()`. This is because HUM copies the data twice: from the host memory space to the host UM space, and then to the GPU UM space.

To reduce the copy time from the host memory to the host UM space (❷ in Figure 6), HUM exploits multiple host threads for the memory copy. The multiple threads simultaneously copy different parts of the source host memory to the host UM space. HUM divides the source host memory object into multiple 2MB chunks and each thread takes care of copying a 2MB memory chunk to the host UM space at a time.

4.6 Scheduling Memory Copy Commands

When more than one CUDA H2Dmemcpy commands are issued consecutively from a CUDA program, the HUM runtime copies their divided 2MB chunks from the host UM space to the device UM space in a round-robin manner. In the HUM runtime, there is a pool of *page migration queues* (PMQs) to queue the page migration requests of 2MB chunks. Moreover, there exists a different PMQ for each CUDA H2Dmemcpy command issued.

For a H2Dmemcpy command from the CUDA program, after dividing the source host memory object into 2MB chunks and copying them to the host UM space with multiple threads, the page migration request of each chunk from the host UM space to the GPU UM space is inserted in the associated PMQ. A host thread called the *page migration thread* (PMT) is taking care of visiting non-empty PMQs in the pool in a round-robin manner. The PMT processes the page migration request at the head of each PMQ by calling the page migration function provided by the NVIDIA driver.

In this case, there must not exist any dependence between destination locations of the consecutively issued CUDA H2Dmemcpy commands. However, such dependences are hardly found in real CUDA applications. Since the HUM runtime has all information about the CUDA H2Dmemcpy commands issued from a CUDA program, the HUM runtime can easily check the dependence at run time.

By doing so, we can schedule the kernel launch as early as possible. As a result, the kernel may access required pages sooner and its execution may finish earlier. This case is illustrated in Figure 8(c). The kernel execution can be initiated after the execution of the H2Dmemcpy command of the vector B has been initiated. In general, with regards to H2Dmemcpy commands, the execution of a kernel command K under HUM can be initiated as early as possible at the time point that satisfies all of the following conditions:

- The last command preceding K in the same stream is a CUDA H2Dmemcpy command, say C, on which K's arguments depend.
- The execution of C has been initiated.
- All target pages of C in the device UM space have been unmapped once to the GPU after the initiation of executing C.

Table 2. System configuration.

CPU	2 × Intel 2.10 Ghz 16-core Xeon Gold 6130
Main memory	256GB DDR4
OS	CentOS 7.6.1810 (kernel 3.10.0-957)
GPU	4 × NVIDIA Tesla V100 PCIe (16GB device memory for each GPU)
GPU driver	NVIDIA display driver 410.48
CUDA version	10.0

5 Evaluation

In this section, we evaluate HUM with various GPU applications and analyze the results. We compare the performance of HUM with that of manual optimizations.

5.1 Methodology

System configuration. We use NVIDIA Tesla V100 GPUs (Volta architecture)[22] for our experiment. Detailed system configuration is summarized in Table 2.

Suite	No.	Name	Sync or async	CPU/H2D overlap	CPU/H2D overlap (HUM)
Parboil	1	bfs	S	N	Y
	2	cutcp	S	N	Y
	3	histo	S	N	Y
	4	lbm	S	N	N
	5	mri-gridding	S	N	Y
	6	mri-q	S	N	Y
	7	sad	S	N	Y
	8	sgemm	S	N	Y
	9	spmv	S	N	Y
	10	stencil	S	N	Y
	11	tpacf	S	N	N
	12	backprop	S	N	Y
	13	b+tree	S	N	Y
	14	cfd	S	N	N
	15	dwt2d	S	N	Y
	16	gaussian	S	N	Y
	17	heartwall	S	N	Y
Rodinia	18	hotspot	S	N	Y
	19	hotspot3D	S	N	Y
	20	huffman	S and A	N	Y
	21	hybridsort	S	N	Y
	22	kmeans	S	N	Y
	23	lavaMD	S	N	Y
	24	leukocyte	S	N	N
	25	lud	S	N	Y
	26	mummergpu	S	N	Y
	27	myocyte	S	N	Y
	28	nn	S	N	Y
	29	nw	S	N	Y
	30	particlefilter	S	N	Y
	31	pathfinder	S	N	Y
	32	srad	S	N	Y
	33	streamcluster	S	N	Y
	34	alignedTypes	S	N	Y
CUDA Code Samples	35	BlackScholes	S	N	Y
	36	eigenvalues	S	N	Y
	37	fastWalshTransform	S	N	Y
	38	matrixMul	S	N	Y
	39	MC_SingleAsianOptionP	S	N	Y
	40	mergeSort	S	N	Y
	41	MonteCarloMultiGPU	A	Y	Y
	42	nbody	S	N	Y
	43	reduction	S	N	Y
	44	scalarProd	S	N	Y
	45	scan	S	N	Y
	46	SobolQRNG	S	N	Y
	47	sortingNetworks	S	N	Y
	48	threadFenceReduction	S	N	Y
	49	transpose	S	N	Y
	50	vectorAdd	S	N	Y
	51	warpAggregatedAtomicsCG	S	N	Y

Figure 9. Characteristics of applications.

Benchmark applications. We use 51 applications from various sources: 11 applications from Parboil[34], 22 applications from Rodinia[3], and 18 applications from CUDA Code Samples[23]. While we use all the applications from Parboil and Rodinia, we choose only 18 out of 170 applications in CUDA Code Samples. We exclude 152 applications in CUDA Code Samples because of the following reasons:

- They use CUDA graphics or driver API,
- They have neither CUDA kernel execution nor H2Dmemcpy,
- They use additional CUDA libraries (cuBLAS, cuFFT, cuSPARSE, cuSOLVER, nvGRAPH),
- They appear in Parboil or Rodinia, or
- Their kernel execution times are too small (less than 1ms) to see the effect of overlapping H2Dmemcpy and CUDA kernel computation.

Figure 9 shows the characteristics of applications in each benchmark suites. The column Sync or async shows the type of H2Dmemcpy commands each application uses. The column CPU/H2D overlap shows if the application is designed to overlap CPU computation and H2Dmemcpy. The column CPU/H2D overlap (HUM) shows if HUM can overlap the CPU computation and the H2Dmemcpy.

Most of the applications use synchronous H2Dmemcpy and hence, they are unable to overlap CPU computation and H2Dmemcpy when running under normal CUDA. On the other hand, HUM can overlap the CPU computation and the H2Dmemcpy in most of the cases except some applications that modify the contents of the source host memory object of the H2Dmemcpy or frees it after the H2Dmemcpy (lbm and tpacf in Parboil, cfd and leukocyte in Rodinia).

We use the largest dataset that fits in the GPU memory for each application, hence, most of the datasets used for the experiment are hundreds of megabytes to a few gigabytes. As the goal of HUM is performance improvement without any code modification, no source code of the applications is modified.

5.2 Results

Speedup. Figure 10 shows the speedup of each application with various optimization schemes on a single V100 GPU. The speedup is obtained over running the original version of each application (this setup is called CUDA hereafter). The optimization schemes are described as follows:

- CUDA-async is a manually optimized version where synchronous memory copy functions in the original application is transformed to corresponding asynchronous ones when the transformation is safe.
- CUDA-UM is a naive UM implementation of each application. We change all `cudaMalloc()` functions to `cudaMallocManaged()`. Then, we remove all CUDA memory copy functions, such as `cudaMemcpy()` and `cudaMemcpyAsync()`, because data will be automatically transferred between the host and the device by CUDA UM.
- CUDA-UM-opt is a manually optimized version of CUDA-UM using user hints (*e.g.*, `cudaMemPrefetchAsync()` and `cudaMemAdvise()`). We add `cudaMemPrefetchAsync()` as early as possible before the CUDA kernel launch so that memory copy and kernel computation can be overlapped. `cudaMemPrefetchAsync()` is also used to map blank pages to the GPU if the pages are first accessed for write by the GPU. This prevents excessive write page faults in the GPU side. We add `cudaMemAdvise()` to avoid page migration if the pages are read by both the CPU and the GPU without any write (*i.e.*, read-only accesses).

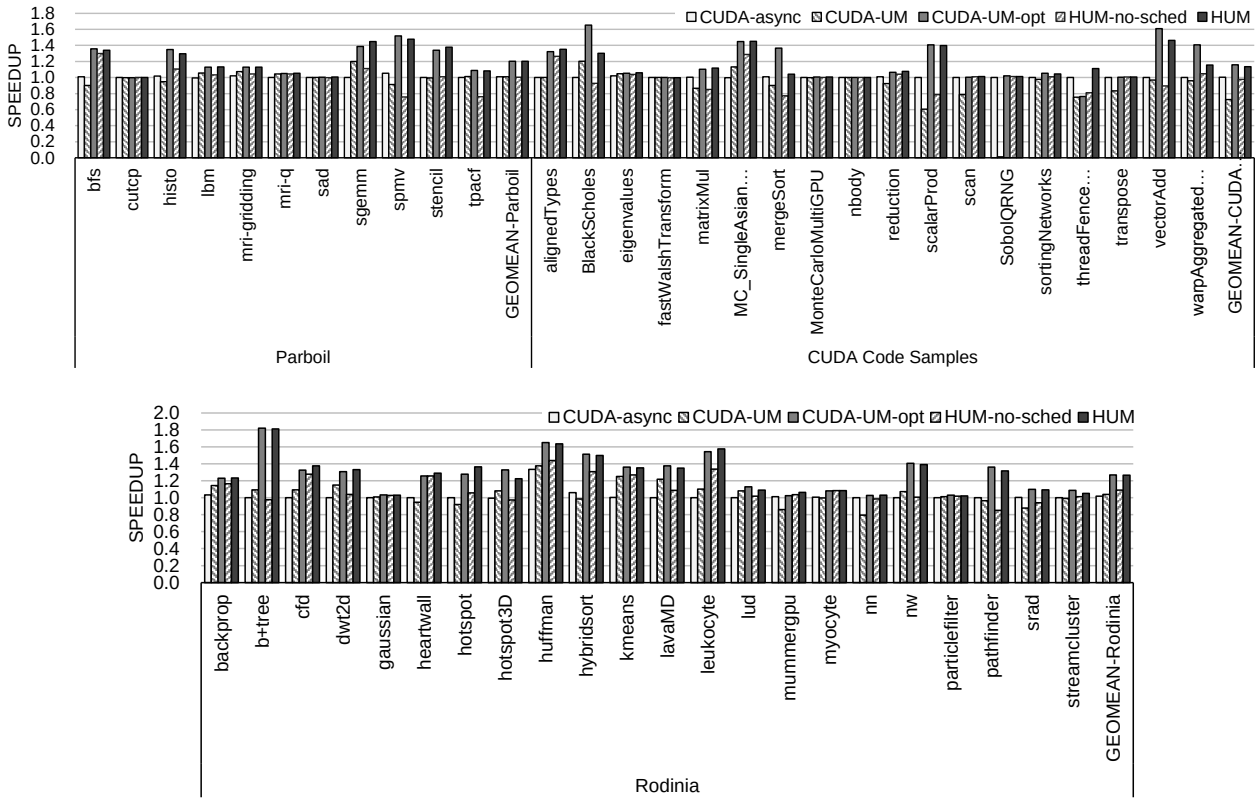


Figure 10. Speedup of each application with a single V100 GPU.

- HUM-no-sched runs the applications under HUM without any H2Dmemcpy command scheduling described in Section 4.6.
- HUM runs the applications under HUM with all the HUM techniques described in Section 4.

The number of memory-copy threads mentioned in Section 4.6 is set to eight in both HUM-no-sched and HUM. When we measure the total execution time, we exclude the file I/O time in each application to clearly see the effect of overlapping.

For all applications, CUDA-UM-opt and HUM outperform CUDA, CUDA-async, and CUDA-UM. Some applications show marginal speedup under HUM and CUDA-UM-opt. This happens when the H2Dmemcpy time takes a very little portion of the total execution time. For example, the host computation time dominates the total execution time of cutcp in Parboil. The kernel execution time dominates the total execution time of mri-q in Parboil, gaussian and particlefilter in Rodinia, MonteCarloMultiGPU, nbody, scan, and transpose in CUDA Code Samples. The D2Hmemcpy time dominates the total execution time of sad in Parboil, SobolQRNG in CUDA Code Samples.

On the other hand, some applications show very good speedup under HUM and CUDA-UM-opt. The applications sgemm and spmv in Parboil, b+tree, hybridsort, and leukocyte in Rodinia have enough kernel computation time to hide the H2Dmemcpy time. The application huffman in Rodinia mainly benefits from overlapping the H2Dmemcpy and the host computation.

CUDA-UM-opt is much better than HUM in BlackScholes, vectorAdd, and warpAggergatedgAtomicsCG in CUDA Code Samples. This is due to the prefetching heuristics used in the NVIDIA driver for page migration. When a GPU page fault occurs, the NVIDIA driver actively prefetches some pages around the faulted page from the host UM space to the GPU UM space according to the prefetching heuristics (note that the heuristics are not publicly known).

CUDA-async is a little bit better than CUDA for Parboil on average, but there is no difference between CUDA-async and CUDA for Rodinia and CUDA Code Samples on average. This is because few applications in Rodinia and CUDA Code Samples have some host computation to hide between the H2Dmemcpy command and the kernel launch command.

CUDA-UM is a little bit better, on average, than CUDA-async for Parboil and Rodinia because of the prefetching heuristics used in the NVIDIA driver for the Unified Memory. CUDA-UM is much worse than CUDA-async for CUDA Code Samples on average because of SobolQRNG. In SobolQRNG, CUDA-UM is 88 times slower than CUDA-async. The 4GB write-only data accessed by the kernel in SobolQRNG incur a lot of page faults in the GPU side for CUDA-UM. This does not happen for CUDA-UM-opt because to avoid the

write page faults, CUDA-UM-opt maps the data pages to the GPU using `cudaMemPrefetchAsync()` before the kernel execution is initiated.

While CUDA-UM-opt achieves the average speedup of 1.22x for all applications, the average speedup of HUM is 1.21x (1.20x for Parboil, 1.26x for Rodinia, and 1.13x for CUDA Code Samples). Thus, the speedup under HUM is comparable to that of CUDA-UM-opt.

Effect of H2Dmemcpy command scheduling. HUM-no-sched is slower than HUM consistently. Even HUM-no-sched is slower than CUDA for some applications. One such a case is when the memory-copy time dominates the execution time. When the kernel computation time is not large enough, overlapping the H2Dmemcpy and the kernel computation cannot fully amortize the slowdown in H2Dmemcpy due to copying the memory object twice from the source host memory space to the host UM space, and then from the host UM space to the device UM space. tpacf in Parboil, nn, pathfinder, and srad in Rodinia, BlackScholes, mergeSort, scalarProd, and threadFenceReduction in CUDA Code Samples fall in this category.

Another case is when the CUDA kernel launch is not scheduled as early as possible. spmv in Parboil, b+tree in Rodinia, matrixMul and vectorAdd in CUDA Code Samples fall in this category. For example, as mentioned in Section 4.6, in vectorAdd, there are two memory objects to transfer from the host to the device (vector A and vector B). Without the memory-copy command scheduling, the kernel execution cannot be scheduled until the entire vector A has been copied to the device.

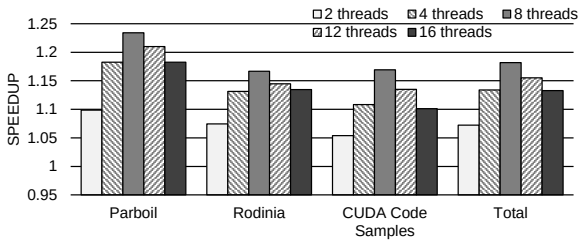


Figure 11. Average speedup obtained by varying the number of memory-copy threads.

The number of memory-copy threads. As mentioned in Section 4.5, HUM uses multiple threads to copy the source host memory object to the host UM space to execute a H2Dmemcpy command. To find the optimal number of threads, we vary the number of memory-copy threads from 1 to 16 and measure the overall performance. Figure 11 shows the average speedup obtained over one thread for each benchmark suite. We see that, on average, eight is the optimal number of memory-copy threads.

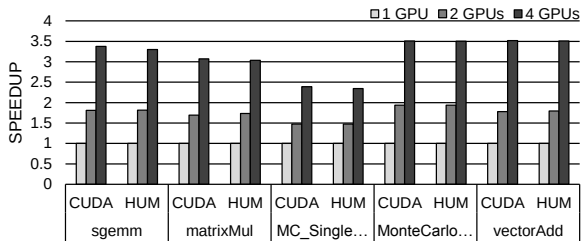


Figure 12. Speedup on multiple GPUs.

Multi-GPU environments. To show that HUM works well with multi-GPU environments, we choose the applications whose speedup under HUM with a single GPU is greater than 1.10 and whose workload can be easily distributed across multiple GPUs. These applications include sgemm in Parboil, and matrixMul, MC_SingleAsianOptionP, and vectorAdd in CUDA Code Samples. We implement the multi-GPU version of them. In addition, we choose MonteCarloMultiGPU in CUDA Code Samples because it is originally designed to support multiple GPUs.

Figure 11 shows the speedup obtained by varying the number of GPUs for these applications. We do not vary the workload for multiple GPUs, hence Figure 11 shows the result of strong scaling for both CUDA and HUM. The speedup is obtained over the case of a single GPU for each of CUDA and HUM. The result indicates that HUM achieves scalable performance in the multi-GPU environment. The major reason for this strong scaling is that page faults occurred in different GPUs are handled by different host threads.

6 Conclusions

HUM hides the host-to-device memory copy time by automatically overlapping it with the host computation or the kernel computation. It exploits CUDA Unified Memory and fault mechanisms of both the host and the GPU. HUM's Unified Memory is hidden to the programmer and there is no need to modify the source code.

Since CUDA is proprietary and not open source, it is impossible to modify the CUDA runtime and the CUDA display driver. Thus, we implement the proposed techniques in the HUM runtime and driver that exploit the CUDA runtime and driver. The techniques can be easily incorporated into the CUDA runtime and the CUDA display driver.

With 51 applications from Parboil, Rodinia, and CUDA Code Samples benchmark suites, we evaluate HUM. We compare their performance under HUM with that of their hand-optimized implementations. The evaluation result shows that HUM is quite effective and practical. On average, HUM achieves 1.20x for applications in Parboil, 1.26x for Rodinia, and 1.13x for CUDA Code Samples. The average speedup of all applications under HUM is 1.21, which is comparable to the average speedup 1.22 of the hand-optimized implementations for Unified Memory.

References

[1] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K. Panda. 2018. OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 143–152. https://doi.org/10.1109/HiPC.2018.00024

[2] Massimo Bernaschi, Mauro Bisson, and Davide Rossetti. 2013. Benchmarking of communication techniques for GPUs. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 250 – 255. https://doi.org/10.1016/j.jpdc.2012.09.006

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[4] Jason Cong, Hui Huang, Chunyue Liu, and Yi Zou. 2011. A reuse-aware prefetching scheme for scratchpad memory. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 960–965.

[5] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. 2005. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 58–58. https://doi.org/10.1109/SC.2005.75

[6] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. 2009. MPI-aware Compiler Optimizations for Improving Communication-computation Overlap. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 316–325. https://doi.org/10.1145/1542275.1542321

[7] Lewis Fishgold, Anthony Danalis, Lori Pollock, and Martin Swany. 2006. An automated approach to improve communication-computation overlap in clusters. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 7 pp.–. https://doi.org/10.1109/IPDPS.2006.1639590

[8] Free Software Foundation. 2019. mprotect(2) - Linux manual page. Website. (2019). http://man7.org/linux/man-pages/man2/mprotect.2.html

[9] Serban Georgescu and Hiroshi Okuda. 2010. Conjugate gradients on multiple GPUs. *International Journal for Numerical Methods in Fluids* 64, 10âĂŘ12 (2010), 1254–1273. https://doi.org/10.1002/fld.2462 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.2462

[10] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. 2016. dCUDA: Hardware Supported Overlap of Computation and Communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 52, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014974

[11] Mark Harris. 2012. How to Overlap Data Transfers in CUDA C/C++. Website. (2012). https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

[12] Mark Harris. 2017. Unified Memory for CUDA Beginners. Website. (2017). https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 770–778.

[14] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. 2004. Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (DATE '04)*. IEEE Computer Society, Washington, DC, USA, 10202–. http://dl.acm.org/citation.cfm?id=968878.968995

[15] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. 2007. DRDU: A Data Reuse Analysis Technique for Efficient Scratch-pad Memory Management. *ACM Trans. Des. Autom. Electron. Syst.* 12, 2, Article 15 (April 2007). https://doi.org/10.1145/1230800.1230807

[16] Ali Khajeh-Saeed and J. Blair Perot. 2012. Computational Fluid Dynamics Simulations Using Many Graphics Processors. *Computing in Science Engineering* 14, 3 (May 2012), 10–19. https://doi.org/10.1109/MCSE.2011.117

[17] Ki-Hwan Kim and Q-Han Park. 2012. Overlapping computation and communication of three-dimensional FDTD on a GPU cluster. *Computer Physics Communications* 183, 11 (2012), 2364 – 2369. https://doi.org/10.1016/j.cpc.2012.06.003

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[19] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. 2014. An investigation of Unified Memory Access performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2014.7040988

[20] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 1092–1098. https://doi.org/10.1109/CCGrid.2015.105

[21] NVIDIA. 2014. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210. Whitepaper. (2014). https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[22] NVIDIA. 2019. Artificial Intelligence Architecture | NVIDIA Volta. Website. (2019). https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/

[23] NVIDIA. 2019. CUDA Code Samples. Website. (2019). https://developer.nvidia.com/cuda-code-samples

[24] NVIDIA. 2019. CUDA Parallel Computing Platform. Website. (2019). https://developer.nvidia.com/cuda-zone

[25] NVIDIA. 2019. CUDA Runtime API: Memory Management. Website. (2019). https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

[26] NVIDIA. 2019. NVIDIA Driver Downloads. Website. (2019). https://www.nvidia.com/Download/index.aspx

[27] NVIDIA. 2019. Pascal GPU Architecture. Website. (2019). https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/

[28] NVIDIA. 2019. Professional Graphics Solution and Turing GPU Architecture | NVIDIA. Website. (2019). https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/

[29] NVIDIA. 2019. Unified Memory Programming. Website. (2019). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

[30] Everett H. Phillips and Massimiliano Fatica. 2010. Implementing the Himeno benchmark with CUDA on GPU clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–10. https://doi.org/10.1109/IPDPS.2010.5470394

[31] James C. Phillips, John E. Stone, and Klaus Schulten. 2008. Adapting a message-driven parallel application to GPU-accelerated clusters. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–9. https://doi.org/10.1109/SC.2008.5214716

[32] Berkeley AI Research. 2019. Caffe: Deep learning framework. Website. (2019). http://caffe.berkeleyvision.org/

[33] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv 1409.1556* (09 2014).

[34] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign, Urbana.

[35] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[36] J. B. White III and J. J. Dongarra. 2011. Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. In *2011 IEEE International Parallel Distributed Processing Symposium*. 59–67. https://doi.org/10.1109/IPDPS.2011.16

[37] Doran Wilde and Sanjay Rajopadhye. 1996. Memory reuse analysis in the polyhedral model. In *Euro-Par'96 Parallel Processing*, Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–397.