NVMMU: A Non-Volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures

Jie Zhang[1], David Donofrio[3], John Shalf[3], Mahmut Kandemir[2], and Myoungsoo Jung[1*]
[1]School of Integrated Technology, Yonsei Institute Convergence Technology, Yonsei University
[2] Department of EECS, The Pennsylvania State University
[3] Computer Architecture Laboratory, Lawrence Berkeley National Laboratory
jie@yonsei.ac.kr, ddonofrio@lbl.gov, jshalf@lbl.gov, kandemir@cse.psu.edu, m.jung@yonsei.ac.kr

***Abstract*—Thanks to massive parallelism in modern Graphics Processing Units (GPUs), emerging data processing applications in GPU computing exhibit ten-fold speedups compared to CPU-only systems. However, this GPU-based acceleration is limited in many cases by the significant data movement overheads and inefficient memory management for host-side storage accesses. To address these shortcomings, this paper proposes a non-volatile memory management unit (NVMMU) that reduces the file data movement overheads by directly connecting the Solid State Disk (SSD) to the GPU.**

We implemented our proposed NVMMU on a real hardware with commercially available GPU and SSD devices by considering different types of storage interfaces and configurations. In this work, NVMMU unifies two discrete software stacks (one for the SSD and other for the GPU) in two major ways. While a new interface provided by our NVMMU directly forwards file data between the GPU runtime library and the I/O runtime library, it supports non-volatile direct memory access (NDMA) that pairs those GPU and SSD devices via physically shared system memory blocks. This unification in turn can eliminate unnecessary user/kernel-mode switching, improve memory management, and remove data copy overheads. Our evaluation results demonstrate that NVMMU can reduce the overheads of file data movement by 95% on average, improving overall system performance by 78% compared to a conventional IOMMU approach.

***Keywords*-GPU; SSD; DMA; OS; NVM;**

I. Introduction

The general purpose graphics processing units (GPUs) are becoming increasingly popular as data processing co-processors with high computation parallelism and comparatively low power consumption. As the computing capabilities of GPUs is improving faster than CPUs [1] , they are increasingly being used in a wide variety of database, scientific, and big-data analytic applications as well as in-memory computing [2] [3] [4] [5] [6] [7]. In a modern GPU, hundreds of processing cores share execution control rather than maintaining their own instruction registers. This single-instruction multiple-thread (SIMT) architecture is capable of processing large sets of data, performing identical operations on numerous pieces of data via thread-level parallelism and data-level parallelism. In addition, it can offload host-side computations from the CPU to the GPU to take advantage of massive device-level parallelism [8]. As a consequence

* Corresponding author: M. Jung (m.jung@yonsei.ac.kr)

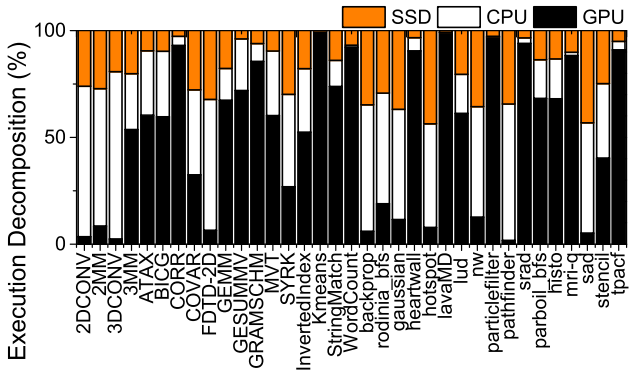


Figure 1: Execution time analysis for major GPU applications on a heterogeneous GPU and SSD system.

of this efficacy in managing application execution by employing different types of parallelism, GPUs exhibit 2x $\sim$ 72x speedups compared to CPU-only systems in big-data analytic frameworks like MapReduce [9] and data-intensive applications [10] [11].

Thanks to their massively parallel computation capabilities, today, GPUs can process more data than they have ever had before, and the volume of such data is expected to be immensely growing. However, the GPU devices have no external data storage medium and employ on-board memory whose size is relatively smaller compared to the host memory. This limits the scale of experiments in numerous data-intensive applications by incurring significant data movement overheads associated with file accesses, uploading and downloading data sets, kernel launching, and synchronizations. Specifically, in cases where the applications need to fetch data sets from the underlying storage drive, data transfer rates between the CPU and the GPU can degrade as much as 95%, which might not be acceptable in many application domains that employ GPUs [12]. Further, the input data that the GPU-kernels of such applications will process should be available in pageable/non-pageable host-side memory before the non-preemptive direct memory access (DMA) begins to transfer them to from the GPU side [13], which further increases the data movement overheads.

To defend against the performance degradation associated with such file-resident data movement, one straightforward solution is to directly attach a high-performance solid state disk (SSD) to the GPU. While modern SSDs can offer a

few GB/sec throughput, which matches well with the GPU interface bandwidth, there are two main challenges to realize this heterogeneous GPU-SSD solution: i) unlike a main memory, all conventional SSDs are *block-based* devices that require help of operating system (OS), and ii) GPUs are not standalone devices that can freely access the external storage device on their own. In contrast to dynamic random access memory (DRAM), GPU applications can access the file data on SSDs only with assistance of the host-side storage software stack, including an I/O runtime library and a file system. Unfortunately, this storage software stack is completely different from the GPU software stack due to their different security, functionality and responsibility. We observed that the overheads related to this software stack separation introduce unnecessary user/kernel-mode context switches and redundant data copies, which consume 1.68x and 4.25x more CPU time, compared to the SSD and the GPU, respectively. Figure 1 illustrates the execution breakdown that consists of the SSD, CPU, and GPU latency values for four GPU benchmarks [9] [14] [15] [16]. One can be observe from this figure that the overheads of CPU intervention related to the file-resident data movement account for 5% $\sim$ 80% of the total execution time, which can prevent the GPU from achieving the full benefits of its massive parallelism.

Motivated by this, we propose *NVMMU*, a novel non-volatile memory management unit that reduces the file-resident data movement overheads by directly connecting the SSD to the GPU. As shown in Figure 2, our NVMMU directly transfers the file data between the GPU and SSD devices, whereas the conventional approach (named IOMMU in this work) needs to make multiple data copies and imposes OS context switching overheads on both the software stacks. Specifically, our NVMMU unifies the two discrete software stacks (one for the SSD and the other for the GPU) via two different kernel components: i) non-volatile direct memory access (NDMA) and ii) unified runtime library (UIL). NDMA manages a physical memory map, physically shared by the involved GPU and SSD devices, while UIL is capable of directly forwarding file data between the two devices without severe CPU intervention. This in turn can eliminate user/kernel-mode switching, memory management overheads, and unnecessary data copies. In addition, our NVMMU provides a simplified programming model for the GPU-accelerated data enabling applications to handle both underlying GPU runtime and I/O runtime libraries over an easy-to-use interface.

We evaluate NVMMU on a high-end NVIDIA GPU system (Tesla K20) [17] with commercially available SSDs by considering three types of storage configurations: i) NVMe SSD, ii) SATA SSD, and iii) SSD array. In cases where we apply our NVMMU to the NVMe SSD device, it shortens average execution time and reduces file-resident data movement overheads by 78% and 95%, respectively. While leveraging an NVMe SSD device would be the most up-front approach to bridge the bandwidth gap between the

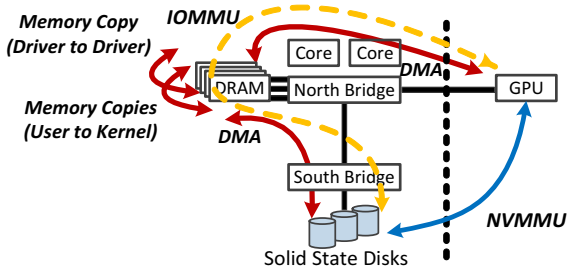


Figure 2: A high-level view of a heterogeneous GPU-SSD architecture.

GPU and the SSD, it is typically 2x$\sim$3x more expensive than moderate GPUs and require 1.5x$\sim$5x more idle/active power than enterprise-scale storage arrays. This high-cost and power-inefficiency may be a barrier to deploying flash storage in GPU computing. To counter this, we also employ our NVMMU in multiple commodity SSD products that expose aggregate performance to the GPU via Direct Media Interface (DMI), whose bandwidth similar to PCI Express (PCIe) interfaces.

Our main **contributions** can be summarized as follows:

• *GPU and SSD stacks unification.* We quantitatively analyze the performance overheads imposed by the current system software stacks atop two different (GPU and SSD) devices. We then unify these two different software stacks in an attempt to directly move the target data between the GPU and SSD without unnecessary user/kernel mode switching and resource management overheads. We observe from our empirical evaluations that NVMMU completely eliminates the performance disparity in cases where the amount of each kernel's input data is greater than 16 MB. In addition, its data-transfer rates account for 60% $\sim$ 90% of the GPU data-transfer rates when GPU applications access the underlying storage media with data sizes ranging from 512 KB to 8 MB.

• *GPU and SSD memory management.* To further reduce the CPU intervention involved in the file-resident data movement, our NVMMU employs *non-volatile direct memory access (NDMA)* that manages multiple kernel CPU-memory chunks mapped to an I/O submission region for both the GPU and the SSD through a scatter/gather memory list. Specifically, NDMA engine leverages the system memory blocks mapped to the GPU-memory by GPUDirect; this memory blocks are also mapped to the appropriate baseline address register associated with the physical memory pages of the SSD. Putting our unified stack interface and NDMA together, NVMMU successfully removes unnecessary memory allocations/releases and redundant data copies across the SSDs, CPU and GPU devices, which can also save 37% energy compared to the conventional IOMMU approach.

• *Simplified programming model for GPU data movement.* Our NVMMU provides a set of programming interfaces, which simplifies the IO runtime, GPU runtime, and memory management operations involved in transferring file data between the GPU and the SSD. While our simplified program-

ming interface is orthogonal to other types of IO runtime interfaces, it gives an opportunity to GPU applications to take full advantage of the direct storage accesses by hiding the complexities of the underlying kernel driver modules.

• *GPU, CPU and I/O service pipelining for SSD arrays.* To protect against system-level storage failures, most fault-tolerance mechanisms ensure that redundancy data (e.g., parity blocks) are stored in persistent storage within the target array. While reliable, this colocation of data and parity bits can result in significant performance degradation and exacerbate the already problematic disparity between the GPU and the SSD. To cope with this, we first quantify the magnitude of the performance degradation exhibited by a popular fault-tolerant mechanism. We then address this performance degradation by re-designing that fault-tolerance mechanism so that it issues its activities in a pipeline fashion alongside CPU and GPU computational kernel activities.

II. SSD+GPU System Organization

In this section, we describe our target heterogeneous GPU-SSD system, the software stack on a host which facilitates data movements, and advanced data transfer methods.

A. Hardware Architecture

Figure 2 illustrates a high-level view of typical system architecture that employs both GPU and SSD. In this system, the GPUs are typically connected to the CPU-side on-chip *northbridge* controller, and their GPU-side memory (hereafter referred to as *GPU-memory*) can be accessed via high performance PCIe links. On the other hand, SSD(s) can be connected to the host off-chip *southbridge* via either PCIe or a thin storage interface such as SATA [18]. Even though GPU and SSD can offer extremely high bandwidth compared to other external devices by taking advantage of different levels of parallelisms, these devices are normally considered just like a conventional peripheral from a CPU viewpoint. As a consequence, the communication protocols transfer data from the CPU to the GPU (and/or SSD) through memory mapped I/O or copy technique. As currently there is no way to directly forward file data between GPU and SSD, the host-side memory, called *CPU-memory*, is utilized to accommodate such data as an intermediate storage medium. The multiple memory interface boundaries can lead to ill-tuned memory management and redundant memory copies, even within the same host machine. While modern GPUs offer excellent performance, often an order of magnitude better than CPU-only systems for data-intensive workloads [8] because of the highly parallel and throughput-oriented nature of their architecture, the overheads associated with communication and data movement prevent GPUs from being able to take full advantage of the inherent massive parallelism in such scenarios.

B. Software Architecture

Because of the different functionalities and purposes of the GPU and the SSD, there are two discrete I/O runtime and GPU runtime libraries, which co-exist on the same host machine and are both utilized in GPU applications. All SSD accesses and file services are managed by modules on the *storage software stack*, while all GPU-related activities including memory allocations and data transfers are handled by modules on the *GPU software stack*. Figure 3c illustrates these two software stacks.

```
00: nImageDataSize = imageWidth * imageHeight * imageChannels * sizeof(float);
01: fd_r = open(r_fileName, O_RDONLY);
02: fd_w = open(w_fileName, O_WRONLY);
03: pSSDInput = malloc(nImageDataSize ); // user-level CPU-memory
04: pSSDOutput = malloc(nImageDataSize );
05: cudaMalloc(&pGPUInput, nImageDataSize ); // GPU-memory
06: cudaMalloc(&pGPUOutput, nImageDataSize );
07: read(fd_r, pSSDOutput, nImageDataSize );
08: cudaMemcpy(pGPUInput, pSSDOutput, nImageDataSize, cudamemcpyHostToDevice);
09: kernel<<dimGrid, dimBlock>>(pGPUInput, pGPUOutput,
                                deviceMask, imageChannels, imageWidth, imageHeight);
10: cudaMemcpy(pSSDInput, pGpuOutput, nImageDataSize, cudamemcpyHostToDevice);
11: write(fd_w, pSSDInput, nImageDataSize);
```

Figure 4: An example of simplified code segment that transfers file data between SSD and GPU.

Storage Software Stack. We now explain how the storage software stack handles storage commands. When a GPU application calls an I/O runtime library through a POSIX interface, the runtime library stores all the user-level contexts and jumps to the underlying *virtual file system* (VFS), which is a kernel module in charge of managing all standard UNIX file system calls. VFS can select an appropriate native file system and initiate file I/O requests. This is done by calling an I/O system function pointer extracted from a file-operation function container, a kernel-level OS data structure. The native file system then checks the actual physical location associated with the file requests, and composes block-level I/O service transactions by calling another function pointer that can be retrieved from a block-device-operation data structure. Finally, disk driver issues I/O requests to the underlying SSD through the PCIe or AHCI controller. Once I/O services have completed, the target data are returned to the GPU application via the aforementioned modules, but in reverse order.

GPU Software Stack. The GPU runtime library, on the other hand, is mainly responsible for executing GPU-kernels and copying data between the CPU and GPU memories. Unlike the storage software stack, this GPU runtime library creates GPU device commands at the user-level and directly submits them with the target data to the kernel-side GPU driver via an *ioctl*, which is another system call enabling device-specific I/O operations. Depending on the GPU commands, the GPU device driver can map a kernel-memory space (CPU-memory) to GPU-memory and/or translate addresses (like CPU-memory's virtual addresses) to physical addresses of the GPU-memory. These GPU-specific data movement activities include *base address register* (BAR) and *graphics address remapping table* (GART) management. Once the address translations/mappings finish, an on-device microprocessor in the GPU facilitates data movement between the CPU and GPU memories. While a file operation

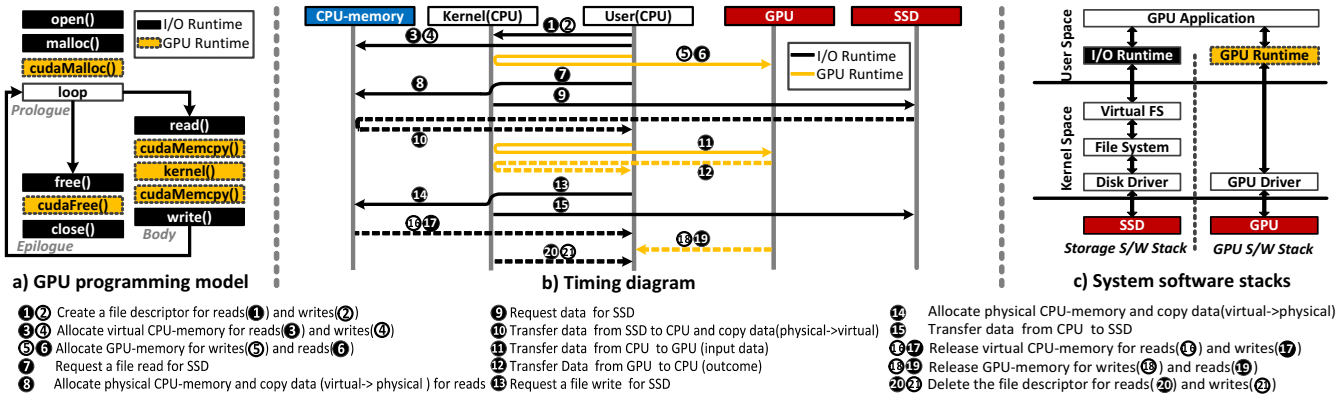


Figure 3: GPU programming model (a), timing diagram illustrating the transfer of file-resident data among CPU, GPU and SSD (b), and system software stacks for the GPU and the SSD devices (c).

on the storage software stack normally travels slower than the memory operations, the GPU software stack has no control of the GPU-specific data transfer activities until the file data arrives. This can clearly prevent the GPU application from being able to get full benefits its massive parallelism.

C. Programming Model

Due to these discrete software stacks, a GPU application must cope with different library interfaces exposed by those stacks. Figures 4 and 3a respectively illustrate an example of the simplified code segment and the corresponding programming model, which explains how a GPU application can manage the file data movement on these two different software stacks. To execute GPU-kernels with a large-scale data set, the application needs to first open a file descriptor. It then allocates user memory in order to read/write data from/to the underlying SSD using the CPU-memory (lines $1 \sim 4$). In this prologue code segment, it must also allocate GPU-memory for the data transfers between the GPU and the CPU (lines 5 and 6). The GPU application then performs calls to an I/O runtime library API (e.g., read, write) by specifying the file descriptor and the CPU-memory address as prepared in the previous steps (line 7). Once the target data is brought into the CPU-memory, the application can initiate data transfers and execute a GPU-kernel by calling the GPU runtime library (e.g., cudaMemcpy, kernel) with a specific number of threads and memory address pointers (lines 8 and 9). We refer to this process (after the prologue, but before the GPU execution) as the *upload*. In cases where the application needs to retrieve the results generated by the GPU, it can do so by copying the corresponding data from the GPU-memory to the user buffer (CPU-memory), and subsequently write them to the SSD (lines 10 and 11). We refer to this process as the *download*. Note that this entire process (within the "Body" of the loop) may be executed multiple times. Finally, in the epilogue, the application cleans up all the CPU- and GPU-memory allocations and file descriptors.

Ill-Tuned Data Path. Figure 3b illustrates the process of transferring file data among the CPU, GPU and SSD devices, based on the programming model we explained above. In this figure, the activities related to I/O runtime and GPU runtime libraries are represented by the solid lines with a different color, while device responses are indicated by the dotted line. The application (working on user-level CPU) always needs to request the I/O or memory operations from the underlying kernel-level modules. Once the modules are done with file-related operations, the disk driver exchanges file data between the SSD and the GPU, using the CPU-memory as an intermediate storage. These numerous hops through layers ill-tuned to handle the raw bandwidth and low latency GPUs expect makes storage data transfers cumbersome and slow. Specifically, as shown in Figure 1, these communication overheads, redundant data copies and CPU intervention overheads among the GPU, CPU and SSD can take as much as 4.21x and 1.68x, respectively, of the CPU execution time taken by GPU and the SSD.

D. Advanced Data Transfer Mechanisms

While the file-resident data movements are relatively new challenges in the heterogeneous GPU-SSD architecture, GPUs and SSDs individually offer advanced data transfer methods that can alleviate the data movement overheads for each.

GPUDirect. GPUDirect [19] supports a direct path for communication between the GPU and a peer high-performance device using the standard PCIe interface. This is typically used to handle peer-to-peer data transfers between the multiple GPU devices. In addition, GPUDirect offers non-uniform memory access (NUMA) and remote direct memory access (RDMA), which can be used for accelerating the data communication with other devices such as network and storage devices. While GPUDirect can be used for managing GPU-memory in transferring a large data set between the GPU and the SSD, it has three shortcomings: i) all the SSD and GPU devices should use PCIe and should exist under the same root complex [19], ii) GPUDirect is incompatible with conventional IOMMU [19] and iii) the file data accesses

should still pass through all the components in the storage software stack.

NVMe. NVM Express (NVMe) [20] is a scalable and high performance interface for NVM systems, which offers an optimized register interface, command and feature sets. NVMe can accommodate both standard-sized PCIe-based SSDs and SATA Express (SATAe) SSDs connected to either northbridge or southbridge in the conventional system organizations. As a consequence, it does not require the SSD and GPU devices to exist under the same root complex like what GPUDirect insists. While NVMe is originally oriented towards managing the data transfers between the CPU and the SSD, its system memory blocks, referred to as physical page regions (PRP) can be shared by the SSD and other third-party devices [20].

AHCI. Advance Host Controller Interface (AHCI) [21] is one of the advanced storage interfaces that employ both SATA and PCIe links in the southbridge controller. AHCI defines a system memory structure, which allows the OS to move data from the CPU-memory to an SSD without significant CPU intervention. Unlike the traditional host controller interfaces, AHCI can expose the high bandwidth of the underlying SSD to the northbridge controller through Direct Media Interface (DMI) that shares many characteristics with PCIe. Specifically, the transfer rate of DMI 3.0 can be as high as 8 GT/sec, which matches well with the high-bandwidth of multiple SSD devices. Further, AHCI's system memory blocks are pointed by physical region descriptor (PRD), whose capabilities are similar to those of PRP.

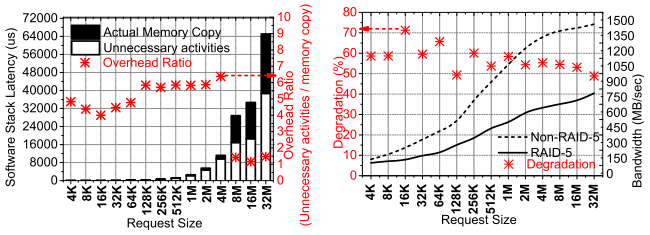
It should be noted that, even though the above advanced data transfer techniques can reduce the CPU intervention on memory or I/O services for each device to some extent, they are *not* aware of the process of transferring file data.

III. NVMMU

The overarching design goal of our NVMMU is to enable data-intensive applications to take full advantage of GPU acceleration by minimizing storage access and data movement overheads. In order to achieve this, we have three specific design and implementation goals; i) *reducing unnecessary CPU-memory activities and user/kernel-mode switching* in current system stacks without major software overhauls, ii) *directly forwarding data from different types of SSD system to the GPU* without any underlying hardware modifications, and iii) *addressing potential performance degradation issues within SSD array systems* to maintain maximum storage performance throughout execution. In this section, we will lay out how we achieve these design and implementation goals by detailing our system stack unification, direct non-volatile memory access, and file handling model.

A. Unifying Software Stacks

One of the main problems with the heterogeneous architecture considered in this paper is that the SSD and GPU devices are completely disconnected from each other, and they are managed by different software stacks. Consequently, many redundant memory allocations/releases and data copies



(a) System Stack Overheads (b) Performance Degradation

Figure 6: Empirical evaluations for performance overheads of the current system stacks (a) and performance degradation imposed by RAID (b).

exist between the user-space and the kernel-space on the SSD and GPU system stacks back and forth, as shown in Figure 3b. To better understand the inefficiency of the current system stacks, we compare the latency of actual data movements with the execution times unrelated to GPU uploads/downloads. One can see from the results plotted in Figure 6a that the execution time spent for unnecessary data copies is longer than the latency of the actual data movements by 16%∼537%, which may not be acceptable in many application domains that employ GPUs. It should be noted that, since it is not possible for a kernel module to directly access user memory space[1], the memory management and data copy overheads between the kernel-space and the user-space imposed by I/O services are unavoidable during the offload of the file-associated data to GPU. Further, the kernel- and user-mode switching overheads along with the data copies also contribute to long latency of the file-driven data movements.

However, this inefficiency can be addressed if we can remove some of the I/O runtime library calls from user-level GPU applications; even though the host-side data-intensive applications work upon two different runtime libraries, *the target data fetched by the storage software stack are only needed by GPU-kernels, not by the host-side GPU-accelerated applications themselves*. Consequently, memory allocations/ releases and data copies between user-spaces and kernel-spaces on the same CPU-memory are *not* necessarily required if there is a mechanism to directly forward such data from the storage software stack to the GPU software stack without an intervention of the GPU applications. In addition, the programming model for the GPU applications can also be simplified. Motivated by this observation, we redesign the current system stacks by employing a virtual file system driver, referred to as *unified interface library (UIL)*. UIL directly forwards the target file-associated data from the storage software stack to the GPU software stack, as depicted in Figure 5b. Specifically, our UIL reads/writes the target file contents from the native file system via reserved kernel system buffers provided by our non-volatile direct memory access support (explained

[1]There is no guarantee that the current kernel is running in the process that the I/O request was initiated.

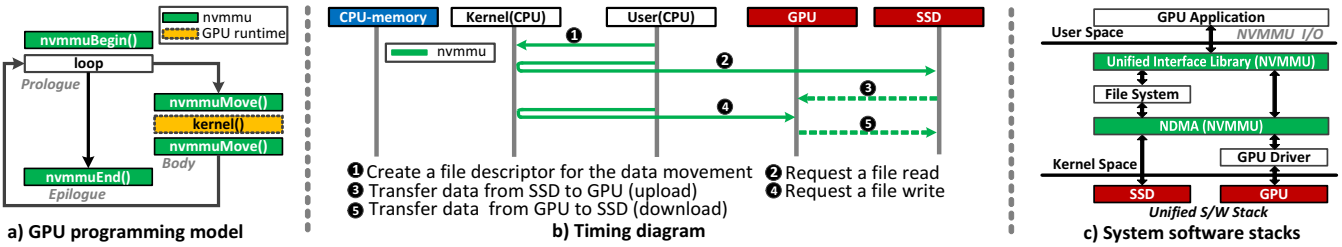


Figure 5: NVMMU-assisted programming model (a), new timing diagram in transferring file-associated data among CPU, GPU and SSD (b) and new system software stacks for the GPU and SSD devices (c).

shortly). Using the kernel system buffer mapped to GPU-memory has the same effect as directly uploading the target data from the SSD to the GPU-memory (and vice versa). This system stack redesign behind our NVMMU is able to remove the need for the redundant memory copies required by user- and kernel-mode switching and the two different runtime libraries' communications. Therefore, our NVMMU can significantly reduce the overheads involved in managing the file-associated data and expose true performance of our SSD testbeds to the GPU.

B. Non-volatile Direct Memory Access

While UIL can minimize the user- and kernel-mode switching overheads, it still requires CPU intervention to relocate data across multiple kernel modules. For example, UIL pulls the data from the file-related OS kernels such as a native file system buffer or I/O scheduler and pushes them into the GPU-memory region through another kernel driver. We could also remove this CPU intervention and reduce the overheads imposed by the data transfers, if there is a specific direct memory access (DMA) method, which can directly forward those data between the GPU and the SSD. One of the challenges behind this DMA-enabled data forwarding scheme is that the logical block address (LBA) spaces (also referred to the device-visible virtual address) are managed by the native file systems located atop the disk controller driver. While the file descriptors and corresponding LBA are determined by the file system, the actual user data should go to the target device through specific I/O submission regions that the disk controller driver handles. As a result, it is difficult to synchronize all the operations through a system call such as *ioctrl* from within the user-level GPU applications.

To address this, we propose Non-volatile DMA (NDMA), which is a modified disk controller driver, that manages multiple kernel CPU-memory chunks mapped to an I/O submission region for both the GPU and the SSD. Specifically, NDMA leverages the kernel buffer regions mapped to the GPU-memory by GPUDirect through a scatter/gather system memory list, and this memory region is also mapped to the appropriate baseline address register (BAR) that represents physical page address spaces of the underlying SSD. The NDMA mapping method can be re-configured based on the interface or controller employed (e.g., NVMe/AHCI). This scattered and mapped kernel buffers are exposed to our upper kernel module, especially, UIL. The UIL recomposes the user data of the incoming I/O requests using such scattered memory buffers if the requests are related to the file-driven data transfers between the GPU and the SSD. Otherwise, it bypasses the requests to the underlying kernel module (i.e., native file systems) in the storage software stack. We note that this request re-composition is only performed for user data, and the corresponding I/O services such as address translation and meta-data are handled by other proper modules in the stack. As a consequence, NDMA can deliver the user data to an appropriate I/O submission region or GPU-mapped memory space, which enables NVMMU to directly transfer data between the SSD and the GPU without any major software overhead or severe CPU intervention.

C. RAID Support

NVMe offers a straightforward DMA strategy that can be applied to GPUDirect, as most NVMe SSDs can be connected to the root complex, where GPUs are employed. In addition, the bandwidth of modern NVMe SSDs can be as high as 3.3 GB/sec, which almost matches that of the GPU interface. Consequently, the easiest way to bridge the bandwidth and latency gap between GPU and storage is to exploit NVMe SSDs. However, we observe that the *power consumption of NVMe SSD devices exceeds the poewr consumption when using a conventional storage interface by 4x $\sim$ 6x*, and the price range of NVMe SSDs is from five thousand to thirty thousand dollars, which is *2$\sim$11 times more expensive than the high-end GPU device itself*. These drawbacks can prevent one from employing NVMe SSDs as a solution to the bandwidth/latency gap between the GPUs and the traditional storage.

To hedge against these shortcomings, one possible solution would be to employ an SSD array that consists of multiple commodity SSDs via a low-power conventional storage interface such as AHCI. The SSD array we tested enables the system to reach a lower cost-per-GB (in the ballpark of what an HDD-based storage array offers) and also consumes 10%$\sim$74% less power than NVMe SSDs. However, we observe that the aggregate performance of the SSD array significantly varies based on the array control logic employed as well as I/O access type exhibited. For example, we compare the data-transfer rate of a RAID-5 based SSD array (consisting of four SATA SSDs) with that of a non-RAID SSD array that has no failure recovery

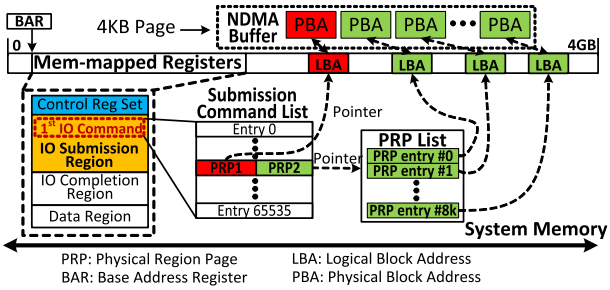


Figure 7: The data structures for NVMe system memory management.

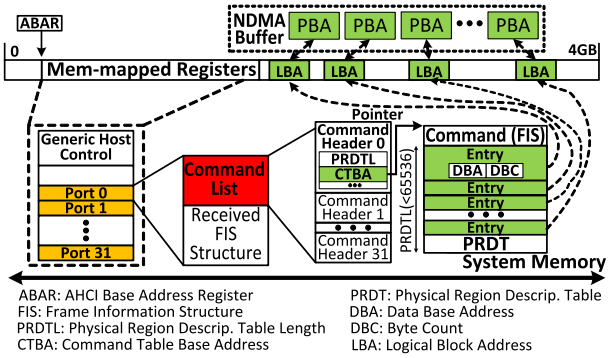


Figure 8: The data structures for AHCI system memory management.

scheme. One can observe from the result plotted in Figure 6b that GPU download performance on the RAID-applied SSD array ranges from only 70 MB/s to 750 MB/s, which are 49%∼71% worse than the non-RAID SSD array. This is because RAID-5 needs to check old parity data and write new parity blocks to the array for every data update request, which introduces additional read-computation-write activities in tandem. We address these performance issues by overlapping the CPU, GPU and I/O activities at the disk controller driver.

IV. Implementation

Our NVMMU unifies the GPU and SSD software stacks via two different kernel components: i) NDMA and ii) UIL. To minimize the modifications made to the underlying system, we updated two Linux drivers for the system memory management (NDMA) and unified interface/programming model (UIL). In this section, we explain our modification to the Linux drivers, and discuss our buffer management management policies and the corresponding data structures.

A. System Memory Management

NVMMU has three different types of memory management policies for transferring data between the GPU and the SSD based on the address space type: i) NVMe SSD address space, ii) AHCI SSD address space, and iii) GPU-memory space.

NVMe SSD to/from GPU-Memory. The communication protocol for NVMe SSD is similar to the memory management protocol that GPUDirect employs. NDMA (of our NVMMU) leverages the system memory blocks mapped to the GPU-memory (with an assistance of GPUDirect) for data transfers from the host to the SSD.

Figure 7 shows how the actual data on the system memory block can be brought from the host to the SSD (or vice versa) through the NVMe protocol. Specifically, the memory-mapped registers that the disk controller driver manages are indicated by the PCIe's *baseline address register (BAR)*. The control register set starting from the BAR is used for managing the NVMe work such as updating the doorbell registers (related to the command submission) and interrupt management (mask set/clear). There are multiple I/O submission queues and completion queues right below the control register set. These submission queues can have many items, each having two *physical region page (PRP) entries*, PRP1 and PRP2. PRP1 entry points to the 4K physical page, which contains the data that NDMA needs to deliver to the target SSD. In cases where the amount of data is greater than 4KB, they can be referred by the pointers on the PRP list, which is indicated by the PRP2 entry.

In our NDMA implementation, the system memory blocks pointed by each PRP entry are mapped to the pinned memory spaces allocated by GPUDirect. Specifically, these NDMA's pinned memory buffers can be mapped to the GPU-memory through `nvidia_p2p_get_pages()`, and we export these pre-allocated memory spaces to UIL, so that it can directly upload (or download) the GPU data, while letting the other kernel components serve the file-related work such as LBA translation in an appropriate manner. Since these NDMA buffers are managed as a pre-allocated memory pool, they should not be released until all data movement activities involving file data are over. To implement this, we also modified the interrupt service routine (ISR) registered at the driver's NVMe initialization time.

AHCI SSD to/from GPU Memory. Unlike NVMe, AHCI has a different data management structure, but still employs a similar strategy for the data transfers between the host and the SSD. Figure 8 explains our system memory management and the corresponding data structures on AHCI. Specifically, *AHCI base address register (ABAR)* indicates the start offset of the memory-mapped registers that contains the generic host control and multiple port registers. Each port represents an individual SSD device in an array. Within the port register, there are two meta-data structures: i) the command list and ii) received FIS structure. The command list consists of 32 command headers, each referring to *physical region descriptor table (PRDT)*, while the received frame information structure (FIS) is used for handshaking control such as a device-to-host (D2H) acknowledge FIS. There are 65536 entries in PRDT, each indicates a system memory block that our NDMA manages. While the maximum buffer size of each PRDT entry can be 4MB, we split the buffer into multiple 4KB physical pages in an attempt to make them compatible with the PRP management policy employed by the GPU. As the DMI of AHCI shares physical

```
00: nImageDataSize = imageWidth * imageHeight * imageChannels * sizeof(float);
01: nvmmuBegin(tid, w_filename); //Pipelining initialization
02: cudaMalloc(&pGPUInP2P, nImageDataSize ); // GPU-memory
03: cudaMalloc(&pGPUOutP2P, nImageDataSize );
04: nvmmuMove(r_filename, pGPUInP2P,0,nImageDataSize , H2D);
05: kernel<<dimGrid, dimBlock>>(pGPUInP2P, pGPUOutP2P,
                                deviceMask, imageChannels, imageWidth, imageHeight);
06: nvmmuMove(w_filename, pGPUoutP2P,0,nImageDataSize , D2H);
07: nvmmuEnd(tid);
```

Figure 9: An example NVMMU-assisted GPU application.

characteristics of the PCIe links, the interrupts delivered by FIS are converted to a PCIe interrupt packet, which allows NDMA to manage ISR in a similar fashion to what is done in NVMe.

B. Unified Interface and Programming Model

Our unified interface library (UIL) is a modified driver that exposes a simple API set, which moves the file data between the GPU and the SSD. As shown in Figure 5c, UIL is placed on top of a native file system and exposes the file-related GPU program interfaces to user GPU applications, removing unnecessary user- and kernel-mode switching overheads between the user-space and the kernel-space. UIL neither uses user-level memory space, nor copies file data back and forth between the user-level spaces and the kernel-level spaces while transferring them between the GPU and CPU. Instead, it handles file accesses and memory buffers that NDMA provides by overriding a conventional virtual file system switch. Note also that UIL directly pulls and pushes file data across two different system software stacks by interacting with the underlying NDMA driver. In addition, UIL piggybacks information for RAID-applied SSD arrays to distinguish read operations used for a system-level failure protection mechanism from normal I/O operations. We will explain this pipelining technique shortly. There are two main advantages of this kernel-level approach; since the file-associated GPU-specific operations are implemented as a VFS extension, UIL-assisted GPU applications can be complied just like a normal GPU program (*no compiler modification is required*). In addition, all the functionality of the I/O runtime and GPU runtime libraries are still available, which means that our NVMMU is fully compatible with all existing GPU applications.

Table I gives a description of our UIL, and Figure 9 shows an illustrative example of the corresponding NVMMU programming model. At the very begining of data process, a GPU application can initialize our NVMMU through `nvmmuBegin`. This function keeps the requester's thread id for the internal resource management, and sends piggyback information about the parity block pipelining before starting the movement of file data (line 1). It then allocates GPU memory and calls `nvmmuMove` by specifying the file name, offset and number of byte (length) for transferring data from the SSD to the GPU (lines 2 $\sim$ 4). Once the GPU is done with GPU-kernel execution, it can download data from the GPU to the SSD with the same function, but give a different `direction` parameter. Finally, `nvmmuEnd` releases all the resources that UIL and NVMMU used for this thread and file (line 6).

C. GPU, CPU and I/O Service Pipeline

For the RAID-based SSD array, we modify a software-based array controller driver to abstract the multiple SSD components as a *single virtual storage device*. This help us to manage device failures and hide the overheads associated with failure management. Since GPU devices have neither OS nor resource management capabilities, the host-side GPU applications in practice have all of the information regarding file data movements, such as the target data sizes, file locations, and timing for the data downloads prior to even beginning GPU-kernel execution. The `nvmmuBegin` function passes `downloadFileName` to UIL, and the latter feeds this information to our array controller driver; the array driver then reads the old version of the target file data and the corresponding parity blocks at an early stage of the GPU body code-segments using them.

Consequently, the array controller driver can load the old data and prepare new parity blocks while the GPU and the CPU prepare for data movement and execution of the GPU-kernel. This *parity block pipelining* strategy enables all parity block preparations to be done in parallel with performing data movements between the GPU and the CPU and/or executing GPU-kernels, thereby eliminating the performance degradation exhibited by conventional RAID systems.

V. Evaluation Results

A. Experimental Setup

System Configuration. To evaluate a heterogeneous GPU-SSD architecture, we employ Tesla K20 GPU, MLC-based Samsung 830 (AHCI-SSD), Intel P3600 (PCIe-SSD) and an SSD array (RAID-SSD) that consists of four AHCI-SSD devices with a system failure recovery mechanism (RAID-5). The GPU device is connected to our host evaluation platform, which has an Intel Core i7 and 16GB DDR3, through the PCIe 2.0 16 lanes. All AHCI-SSD and RAID-SSD testbeds employ SATA 3.0 on AHCI, while our PCIe-SSD is connected to the host through NVMe 1.1 [20]. To broaden our comparison (for especially device-level dynamic power and throughput) analysis, we also evaluate an HDD array (RAID-HDD) that consists of four 7500 RPM enterprise-scale HDDs. In this evaluation, the dynamic power values are captured by our in-house PCIe power analyzer. The important characteristics of our testbeds are given in Table II. With these, we configure five different memory management units as follows:

- NVMe-IOMMU: PCIe-SSD, using a conventional IOMMU (baseline).
- AHCI-NVMMU: AHCI-SSD, using NVMMU.
- NVMe-NVMMU: PCIe-SSD, using NVMMU.
- RAID-NVMMU: RAID-SSD, using NVMMU without our parity block pipelining.

Interface	Description
`int nvmmuBegin(int threadId, char *downloadFileName)`	The `nvmmuBegin` function initializes UIL and NDMA. If `downloadFileName` is not NULL, it will also initiate pipelining for the parity blocks. This function keeps the record of threadId and returns 1 upon success. Otherwise, it returns 0.
`int nvmmuMove(char *fileName, void *gpuMemPtr, int offset, int length, int direction)`	The `nvmmuMove` function creates data path between SSD and GPU based on the GPU-memory address `gpuMemPtr` and moves data taking into account the `fileName`, `offset`, and the amount of data (`length`). The `direction` can be either D2H (device-to-host) or H2D (host-to-device); D2H and H2D indicate the data movement "from GPU to SSD" and "from SSD to GPU", respectively. The `nvmmuMove` returns the number of transferred bytes upon success. Otherwise, it returns 0.
`int nvmmuEnd(int threadId)`	The `nvmmuEnd` cleans up the resources that UIL and NDMA use for the `threadId` thread.

Table I: The interface functions in our NVMMU library.

	Total Cost	Cap. (TB)	Device	Protocol	# of Dev.	Failure Re-cover
RAID-HDD	$1085	2	HDD	SATA	4	YES
AHCI-SSD	$194	0.25	MLC	SATA	1	NO
PCIe-SSD	$5990	0.36	MLC	NVMe	1	NO
RAID-SSD	$1760	2	MLC	SATA	4	YES

Table II: Salient characteristics of the evaluated SSDs.

Bechmarks	Input (MB)	Output (MB)
PolyBench	1520.6	234.5
Mars	292.5	282
Rodinia	479.6	523
Parboil	123.1	94.9

Table III: Average data sizes of the GPU benchmarks tested.

- RAID-NVMMU-P: RAID-SSD, using NVMMU with our parity block pipelining.

We compare our NVMMU against a baseline system that employs PCIe SSD with the conventional IOMMU support. Note that, in our tests, IOMMU is disabled for all devices that use NVMMU since we often faced kernel panic while performing DMA; we conjuncture that this is because the current version of GPUDirect needs all physical memory regions managed by IOMMU, which makes our system incompatible with the conventional IOMMU.

GPU Workloads. We use 39 benchmark applications compiled from *Ploybench* [16], *Mars* [9], *Rodinia* [14], and *Parboil* [15]. *PolyBench* is a collection of benchmarks containing static control parts, and applications we tested are mainly related to convolution and linear algebra on large data sets. *Mars* is a MapReduce framework on GPU that covers several web applications such as string match or inverted index on hdml file sources. *Rodinia* takes into account emerging accelerated-computing applications ranging from bioinformatics to data mining to classical algorithms. *Parboil* is designed toward capturing different scientific and commercial kernels including image processing, biomolecular simulation, fluid dynamics, and astronomy. The upload and download data sizes we tested for our workloads are listed in Table III. For the device-level analysis, we also composed in-house micro-benchmarks by modifying a CUDA sample code [13]. *bench-rdrd* and *bench-sqrd* generate fourteen different request sizes for uploading file data with random and sequential access patterns, respectively. In contrast, *bench-rdwr* and *bench-sqwr* generate download scenarios with random and sequential access patterns, respectively. Finally, to evaluate file-associated GPU application performance, we put all GPU input data (e.g., keys, images, etc) in the storage system in a "binary" format before evaluation begins, and check the data at the end of each evaluation by reading them from the underlying storage system.

B. File-Resident Data Movement Latency

Figure 10 shows the latency values in transferring the file data for the GPU applications tested. For better comparison, we *normalize* the evaluation results of each configuration to NVMe-IOMMU, and show the corresponding average speedup for each benchmark category in Figure 12a. One can observe from these figures that our NVMMU dramatically reduces the overheads of the file-resident data movement for all the benchmarks. Specifically, compared to NVMe-IOMMU, NVMe-NVMMU reduces the latency of the data movement costs for Polybench, Mars, Rodina and Parboil by 202%, 70%, 112% and 108%, respectively. While the baseline SSD array (RAID-NVMMU) exhibits 1.1x speedup for PolyBench on average, it introduces much longer latency values in transferring the file data for Mars, Rodina and Parboil. This is because all the benchmarks except PloyBench have a large amount of output data (accounting for 50% of the total data), which introduces significant performance degradation in the system that has a system-level failure recovery mechanism, due to read-write intermixed access patterns on the parity block computation. In contrast, RAID-NVMMU-P successfully isolates read operations from the place where the GPU applications try to download the output, and pipelines the computation on parity blocks. As shown in Figure 12a, RAID-NVMMU-P reduces the overheads originating from file data movement by 218%, 16%, 74% and 71%, for PolyBench, Mars, Rodinia and Parboil, respectively. This latency reduction on the file data movement is similar to what NVMe-NVMMU achieves, but it consumes 79% less dynamic power and is 82% less expensive than NVMe-NVMMU, as discussed in Section V-D. We also note that, even though AHCI-NVMMU only uses a single AHCI-SSD, it exhibits 10% better data movement latency in PolyBench, compared to PCIe-SSD with the conventional IOMMU. This is because NVMMU removes redundant memory copies and user-/kernel-mode switching overheads, which play a more critical role in read-intensive applications compared to write-intensive applications.

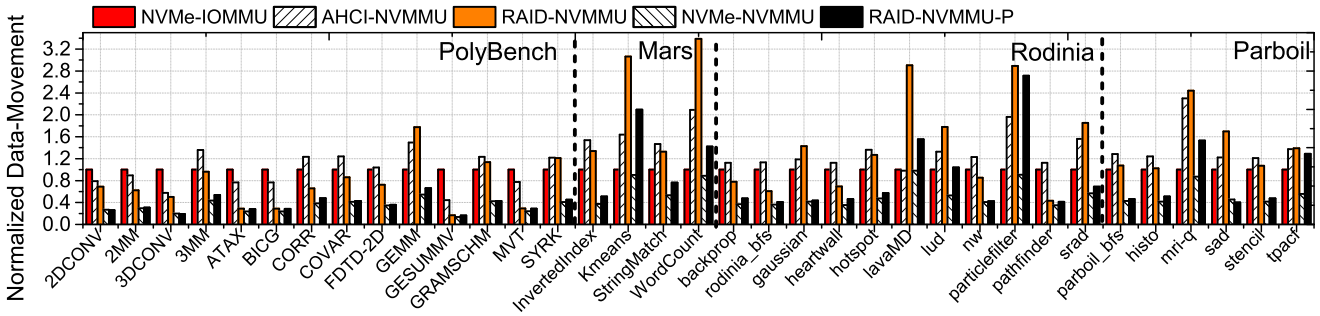


Figure 10: Data movement overhead analysis.

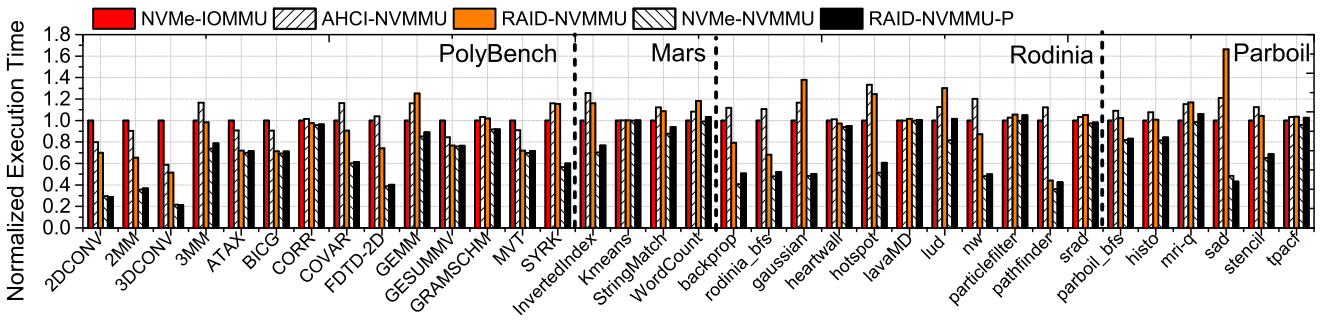
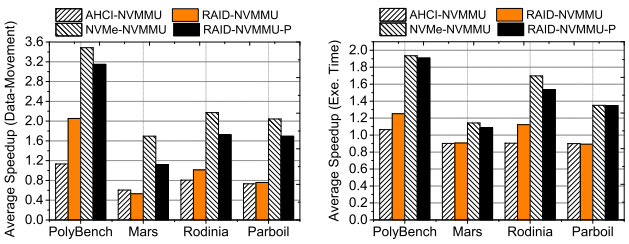


Figure 11: Performance improvement analysis.



(a) Data movement (b) Execution time

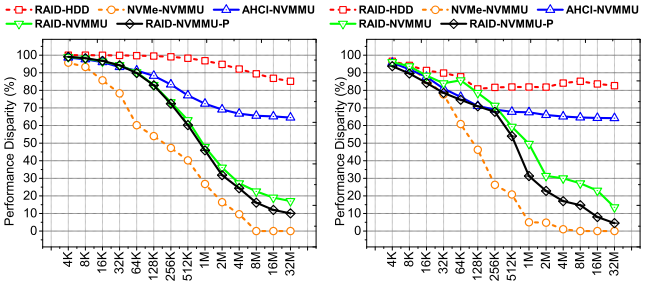
Figure 12: Average speedup analysis.

C. Application Performance Improvement

The performance improvements in terms of application execution time, and the corresponding average speedup values for different GPU benchmarks are plotted in Figures 11 and 12b, respectively. As our NVMMU directly forwards data from SSD to GPU by minimizing the memory copy penalty and the user-/kernel-mode switching overheads, it significantly improves the overall performance in most GPU benchmarks tested. Specifically, NVMe-NVMMU enhances the application performance, compared to NVMe-IOMMU, by 192%, 14%, 69% and 37%, for PolyBench, Mars, Rodinia and Parboil, respectively. *3DCONV* of the PolyBench benchmark is composed of a three-level nested for-loop and spends most of it execution time on accessing the storage and moving the input data. The speedup brought by our NVMMU is as high as 5x in *3DCONV*, which is one of good examples that demonstrate our NVMMU's performance superiority over the conventional IOMMU. We also note that, even though *kmeans* of *Mars* benchmark is a clustering algorithm used extensively in data mining, the total amount of input data is around ten megabytes and the converged result sizes are less than tens kilobytes. Because of this, we observe only a slight performance benefit for *kmeans* (in Mars). It should also be noted that low-end AHCI-SSD with our NVMMU exhibits an excellent performance, similar to what IOMMU with PCIe-SSD delivers, and shows even better performance under heavy read-intensive workloads such as *2DCONV*, *MM* and *3DCONV* in PolyBench. We also note that NVMMU successfully hides all overheads behind our SSD array testbed that employs a system-failure recovery mechanism. Specifically, RAID-SSD with our NVMMU (RAID-NVMMU-P) results in an application performance, which is very similar to that of NVMe-NVMMU; it only has 1.6%, 7%,12% and 1% performance degradation compared to NVMe-NVMMU for PolyBench, Mars, Rodinia and Parboil, respectively. Considering the device-level characteristics in terms of the dynamic power and cost-per-bit, NVMMU can make RAID-SSD more promising in a heterogeneous GPU and SSD architecture, as described below.

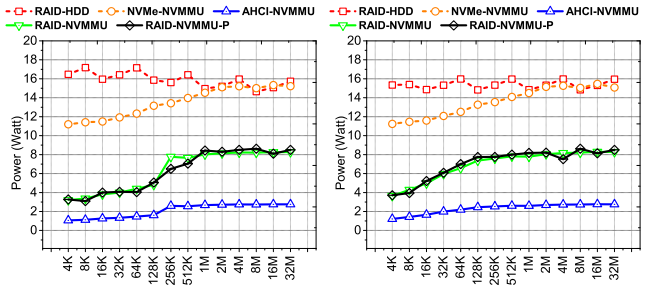
D. Device-Level Performance Analysis

Upload performance. Figure 13 shows the percentage performance disparity between the CPU and the GPU during file data movements. As shown in this plot, the performance of RAID-HDD only accounts for 4% and 1% of the GPU data-transfer throughput in cases where the I/O request size is relatively small (4K $\sim$ 256K). As the parity block computation is negligible on uploads (reads), both RAID-NVMMU and -P can expose full, aggregate performance by taking advantage of the array-level parallelism. Specifically, under



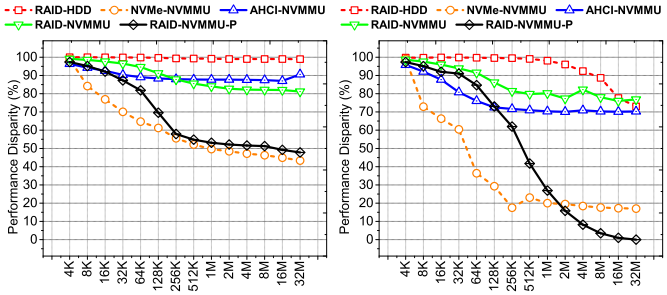
(a) bench-rdrd (b) bench-sqrd

Figure 13: Upload performance disparity analysis.



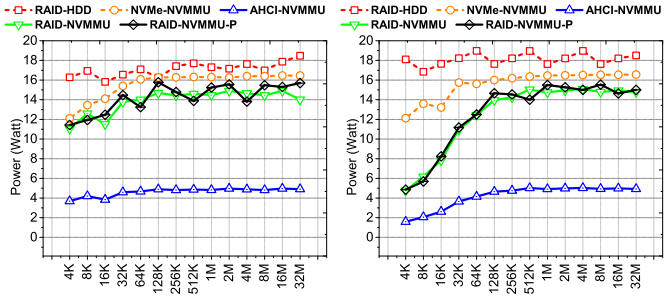
(a) bench-rdrd (b) bench-sqrd

Figure 14: Upload dynamic power consumption.



(a) bench-rdwr (b) bench-sqwr

Figure 15: Download performance disparity analysis.



(a) bench-rdwr (b) bench-sqwr

Figure 16: Download dynamic power consumption analysis.

the large-size I/O patterns, RAID-NVMMU and -P reduce the performance disparity between the CPU and the GPU on *bench-rdrd* and *bench-sqrd* by 10% and 15%, respectively. On the other hand, NVMe-NVMMU can eliminate all the performance gap between the CPU and the GPU if the sizes of incoming I/O requests are large (8MB $\sim$ 32MB).

Upload power. Even though PCIe SSD (with our NVMMU support) is the best solution in terms of performance, dynamic power consumption paints an entirely different picture. Figure 14 shows dynamic power consumption on both *bench-rdrd* and *bench-sqrd*. The dynamic power values of NVMe-NVMMU accounts for 67% $\sim$ 105% of RAID-HDD, and becomes even worse in cases where the upload size is grater than 1MB. This shortcoming can be addressed if we employ RAID-SSD. Specifically, RAID-NVMMU and -P consume low dynamic power, ranging from 3.2 watts to 8.6 watts while reducing the performance disparity between the CPU and the GPU by an average of 57%. The SSD array options consume 80% less dynamic power than RAID-HDD and, even compared to the single-device system, they offers 61% lower power consumption than NVMe-NVMMU under all the workloads tested. On the other hand, AHCI-SSD with our NVMMU (AHCI-NVMMU) achieves the lowest power consumption across all storage drives. Considering both power consumption behavior and performance benefits, we believe that RAID-NVMMU-P is the most promising solution in our testbeds.

Download performance. We observe that all storage drives exhibit low performance on random writes, and therefore, the reduction rate of performance disparity between the SSD and the GPU (with help of NVMMU) can be at most 66%, even under the large size I/O request patterns. As shown in Figure 15a, the performance disparity reduction is also limited by RAID-SSDs that need to keep redundant data for the system failure recovery.

Figure 15b plots the performance disparity between the GPU and CPU *bench-sqwr* workloads. Even with the sequential access pattern, the performance disparity reduction for RAID-NVMMU averages only on 20%, which is the most critical bottleneck in SSD arrays. This is because the parity block handling for each and every I/O request makes the array write performance faraway from that, GPU download transfer-rates. In contrast, our parity block pipelining technique (RAID-NVMMU-P) can reduce the performance disparity by 98% for *bench-sqwr*, which is even better than NVMe-NVMMU in cases where the size of requests is grater than 2MB. RAID-NVMMU-P performs the read operations related to the parity block update much earlier through UIL (`nvmmmuBegin`) and strips writes over multiple AHCI-SSDs, which can dump the GPU output data as fast as NVMe-NVMMU. One can observe from Figure 15b that, RAID-NVMMU-P completely removes the performance disparity between the GPU and the SSD in the case of large I/O request (32MB) as it is much easier to achieve high array-level parallelism with the large I/O requests.

Download power. Unlike the GPU upload dynamic power, the power consumption on GPU downloads varies based on I/O sizes as well as the access patterns of our GPU workloads. Under the *bench-rdwr* workloads, the dynamic power values of NVMe-NVMMU and RAID-NVMMU account for 91% $\sim$ 84% of RAID-HDD, respectively. While the parity

block pipelining technique performs very well in improving the SSD array performance, the power consumption characteristics of RAID-NVMMU-P are not much different from that of RAID-NVMMU. We believe that this is because the pipelining is not a main factor that reduces the number of requests; it only isolates reads from writes for parity block computation.

For *bench-rdwr*, NVMe-SSD has a power consumption of around 17 watts, which is similar to what RAID-HDD consumes, while our RAID-NVMMU and -P reduce power consumptions from 3% to 83% irrespective of the request sizes. One of the reasons behind this low power consumption is that the SSD array leverages a standard low-power interface on southbridge rather than high-performance PCIe bus on northbridge. PCIe allows a maximum power consumption of 25 watts for modern PCIe SSDs, which makes them not only high performance but also much more power-hungry devices.

VI. Acknowledgement

This research is supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the "IT Consilience Creative Program" (IITP-2015-R0346-15-1008) supervised by the NIPA (National IT Industry Promotion Agency). This work is also supported in part by DOE grant DE-AC02-05CH1123, NSF grants 1213052, 1205618, 1302557, 1526750, 1409095, and 1439021.

VII. Conclusion

In this paper, we proposed a NVMMU that addresses the performance disparity between SSD and GPU caused mainly by file-resident data movements. We implemented and evaluated NVMMU on a high-end NVIDIA GPU system with three types of commercially available SSD devices. With our NVMMU support, the GPU applications tested exhibit 78% performance improvement, and require 37% less dynamic power, compared to a baseline that employs the conventional IOMMU.

References

[1] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[2] R. Wu, B. Zhang, and M. Hsu, "Gpu-accelerated large scale analytics," in *Workshop on UnConventional High Performance Computing (UCHPC)*, 2009.

[3] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, 2008.

[4] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[5] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, 2006.

[6] C. Ji, Y. Li, W. Qiu, U. Awada, and K. Li, "Big data processing in cloud computing environments," in *International Symposium on Pervasive Systems, Algorithms and Networks (ISPAN)*, 2012.

[7] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *ACM Transactions on Graphics (TOG)*, 2003.

[8] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[9] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2011.

[10] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[11] NVIDIA, "Gpu-accelerated applications," http://nvidia.com/content/tesla/pdf/gpu-accelerated-apps-for-hpc.pdf, 2013.

[12] M. Shihab, K. Taht, and M. Jung, "Gpudrive: Reconsidering storage accesses for gpu acceleration," 2014.

[13] NVIDIA, "Nvidia cuda library documentation," http://docs.nvidia.com/cuda/, 2013.

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

[15] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[16] L.-N. Pouchet, "Polybench: the polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/*, 2012.

[17] NVIDIA, "Nvidia's next generation cuda compute architecture: Kepler gk110 whitepaper," http://nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf, 2013.

[18] Chen, Feng and Koufaty, David A and Zhang, Xiaodong, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *ACM SIGMETRICS Performance Evaluation Review*, 2009.

[19] Mellanox, "Nvidia gpudirect technology accelerating gpu-based systems," http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf, 2013.

[20] D. Cobb and A. Huffman, "Nvm express and the pci express ssd revolution," 2012.

[21] E. Ooi, "Method and apparatus for using advanced host controller interface to transfer data," 2004.