A copy of this technical report can be obtained by sending a request to h1tseng@cs.ucsd.edu

# Gullfoss: Accelerating and Simplifying Data Movement among Heterogeneous Computing and Storage Resources

Hung-Wei Tseng Yang Liu Mark Gahagan
Jing Li Yanqin Jin Steven Swanson
Computer Science and Engineering Department
University of California, San Diego
{h1tseng,yal036,mgahagan,jil261,y7jin,swanson}@cs.ucsd.edu

## ABSTRACT

High-end computer systems increasingly rely on heterogeneous computing resources. For instance, a datacenter server might include multiple CPUs, high-end GPUs, PCIe SSDs, and high-speed networking interface cards. All of these components provide computing resources and operate at a high bandwidth. Coordinating the movement of data and scheduling computation across these resources is a complex task, as current programming models require system developers to explicitly schedule data transfers. Moving data is also inefficient in terms of both performance and energy costs: some applications running on GPU-equipped systems spend over 55% of their execution time and 53% of energy moving data between the storage device and the GPU.

This paper proposes Gullfoss, a system that provides a simplified programming model for these heterogeneous computing systems. Gullfoss provides a high-level interface for specifying an application's data movement requirements, and dynamically schedules data transfers while accounting for current system load and program requirements. Our initial implementation of Gullfoss focuses on data transfers between an SSD and a GPU, eliminating wasteful transfers to and from main memory as data moves between the two. This saves memory energy and bandwidth, leaving the CPU free to do useful work or operate at a lower frequency to improve energy efficiency.

We implement and evaluate Gullfoss using commercially available hardware components. Gullfoss achieves 1.46× speedup, reduces energy consumption by 28%, and improves energy-delay product by 41%, comparing with systems without Gullfoss. For multi-program workloads, Gullfoss shows 1.5× speedup. Gullfoss also improves the performance of a GPU-based MapReduce framework by 10%.

## 1. INTRODUCTION

The growing size of application data, coupled with the introduction of heterogeneous computing resources and high-performance storage and network devices, is reshaping the computing landscape. Increasingly, application performance is determined by how efficiently data can can be moved between diverse hardware components. For instance, we found that moving data accounts for 55% of execution time for the Rodinia GPU benchmark suite [1].

As high-speed interconnects like PCIe enable fast, low-latency communication between devices, applications still must rely on the entrenched CPU-centric programming and execution model in which the CPU first transfers data from the source to main memory and then from main memory to its destination. Many modern systems support a more efficient transfer mechanism – peer-to-peer transfer over PCIe [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] – but software support for this approach is inconsistent and, even if it is present, the programmer must choose between transferring data through main memory or using peer-to-peer communication. The result is that programming heterogeneous systems is cumbersome and it is difficult for programmers to make the best use of available peripherals and PCIe and memory bandwidth.

This paper presents *Gullfoss*, a framework that maximizes the efficiency of data movement between peripherals within a system. The Gullfoss interface allows programmers to specify data transfer requirements at a high level and then the Gullfoss runtime implements them efficiently using the available communication mechanisms and taking into account the current state of the system.

Since Gullfoss automatically selects the most efficient route of carrying application data, applications that use Gullfoss are agnostic to the underlying interconnect technology. As new interconnects emerge, Gullfoss could be extended to utilize them as well without requiring changes to applications.

One of Gullfoss's strengths is its ability to leverage peer-to-peer transfers between peripherals. Avoiding a round trip through memory provides multiple benefits. First, it reduces transfer latency by removing the main memory and the CPU from the data path. Second, it frees the CPU to perform useful work rather than managing data movement. Third, it reduces contention for TLB (Translation Look-aside Buffer) entries and memory bandwidth. Finally, if there is no useful work to do on the CPU, the CPU can shift to a low power mode to save energy.

To demonstrate the effectiveness of Gullfoss, we have used it to optimize the data transfer between NVM-Express (NVMe) [14] high-speed solid-state disks (SSDs) [14, 15, 16] and general-purpose GPUs (GPGPUs). To accomplish this we also implemented an extension of the existing NVMe interface (called DirectNVMe) to support peer-to-peer trans-

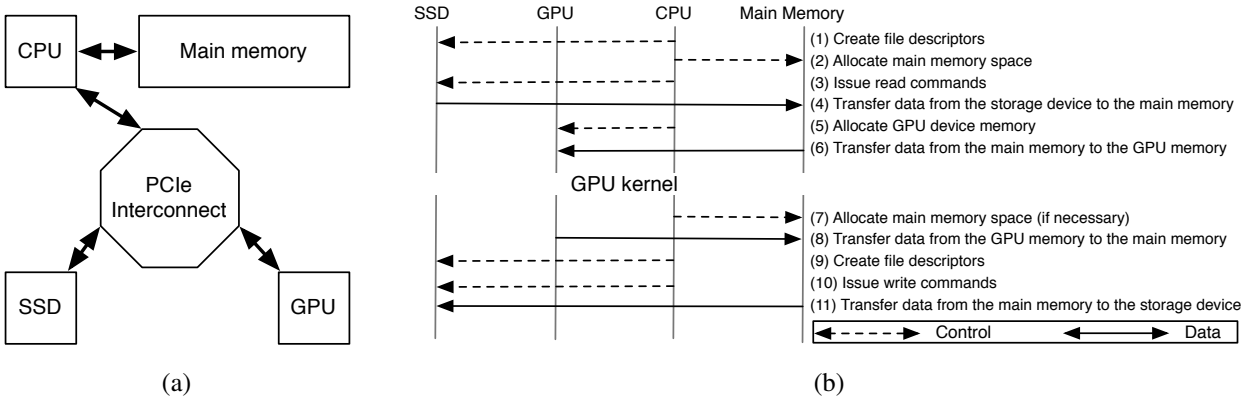


**Figure 1: (a) The conventional GPGPU platform and (b) the process of moving data between the GPU and the SSD on this platform**

fers between NVMe SSDs and the GPUs.

This paper makes the following contributions:

- It presents Gullfoss, a framework that provides a simple, easy-to-use programming model and runtime system for managing data movement in heterogeneous computing systems.
- It proposes and implements DirectNVMe, an NVMe extension to support peer-to-peer transfers between SSDs and other devices (e.g., GPUs). This is the first system we know of that implements direct data transfers between commercially available NVMe SSDs and GPUs.
- It evaluates Gullfoss and demonstrates the improvement in performance and energy-efficiency that it enables.

We implement Gullfoss and evaluate it in a system that includes an Intel Xeon processor, an NVIDIA K20 GPU, and a high-end NVMe SSD. The experiments demonstrate that for a single GPU-accelerated application, Gullfoss speeds up application end-to-end latencies by 1.46× on average. Using the default power management policy in Linux, Gullfoss reduces energy consumption by 28% and improves the energy-delay product by 41%. With slower CPUs that consume less power, Gullfoss becomes even more beneficial, speeding up applications by 1.56× on average.

The performance advantage of Gullfoss is more significant in a multi-program environment, as Gullfoss dynamically spreads transfers across the available data paths when multiple processes are sharing the same set of hardware resources. Gullfoss can accelerate the execution multi-program of GPU applications by 50% on average. Even for the GPU-MapReduce framework that is highly optimized for heterogeneous computing system [17], Gullfoss still improves the performance of the framework by 10%

The remainder of this paper is organized as follows. Section 2 describes the CPU-centric data transfer model currently used in heterogeneous computer systems and identifies the importance of data movement to system efficiency. Section 3 introduces the architectural supports that Gullfoss relies on. Section 4 describes Gullfoss's design, implementation, and programming model in detail. Section 5 describes the system and the benchmark applications that we used to evaluate Gullfoss. Section 6 evaluates Gullfoss's performance and energy efficiency. Section 7 places Gullfoss in the context of previous efforts. Section 8 concludes this paper.

## 2. HETEROGENEOUS COMPUTING SYSTEMS

Heterogeneous computing systems bring together diverse hardware resources such as GPGPUs and SSDs to improve performance and efficiency. Figure 1(a) presents a typical architecture for a heterogeneous computing platform equipped with a GPGPU and an SSD. The GPU executes application kernels that require high degrees of parallelism. The CPU manages the execution of the kernels, performs computations that are not suited to the GPU, executes system libraries, and manages the DRAM-based main memory. The high-performance SSD holds input and output data and can sustain several GB/sec of bandwidth. These system components communicate through PCIe.

While this architecture allows programs to leverage the performance of specialized components and provides more computing resources than using APUs (Accelerated Processing Units) [18, 19, 20], it also incurs costs in moving data between these devices during execution. This data can add large overheads to data-intensive applications.

In the current programming models for heterogeneous computing systems (e.g., CUDA or OpenCL), the CPU serves as both control plane and data plane. The application requests a data transfer, for example, from an SSD to a GPU. The SSD's driver then transfers the data to a buffer in DRAM via DMA before the GPU driver transfers it to the GPU, again via DMA.

Figure 1(b) illustrates this process. Steps (1–6) transfer data between the SSD and the GPU. In Steps(1–3), the application creates file descriptors, allocates DRAM memory buffers, and issues read() system calls. After the SSD directly accesses main memory in Step (4), the CPU allocates space in the GPU (Step (5)) and, in Step (6), copies the data

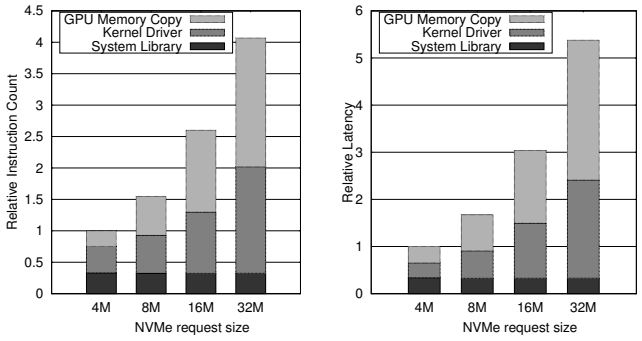


**Figure 2: The (a) instruction count and (b) latency breakdown for NVMe read commands under different request sizes**

from the main memory to the GPU memory. Steps (7–11) show the same process in reverse as the program moves the results of the GPU kernel back to the SSD.

The result is two useless copies that consume CPU time, waste memory bandwidth, pollute TLBs and caches, and consume DRAM. There are ancillary costs as well in the form of extra run-time overheads and exception handling.

Figure 2 details the run-time overhead of this model on a machine described in Section 5.1. We examine the number of CPU instructions and latencies required for read requests varying from 4 MB to 32 MB. We start showing the result from 4 MB since our smallest input file is 7 MB. Because the NVMe SSD we use here can accept at most 32 MB data access in each I/O command, we stop at 32 MB. Figure 2 breaks down the CPU instructions and latencies into three parts: "System library" covers most operations in Step (1), including opening file descriptors and calculating the LBAs (Logical Block Addresses) of requesting files. "Kernel driver" represents the operations of Steps (3–4) that require the processor to set up DMA resources and interact with the SSD. "GPU Memory Copy" includes the run-time system costs of moving data from DRAM to GPU device memory. We exclude the overheads of allocating DRAM and GPU memory spaces in these tests. As Figure 2 presents, the conventional programming model devotes 25%–50% of its CPU instructions to moving data from DRAM to GPU using the run-time system. This extra run-time overhead accounts for 35%–55% of the latencies in our tests.

In addition to the system overhead, the conventional model, combined with current SSD and GPU interfaces, prevents programmers from utilizing the more efficient data paths provided by emerging high-speed interconnects. For example, PCIe allows each device to send data packets directly from one device to another if both peripherals provide architectural supports. AMD's DirectGMA [3] and NVIDIA's GPUDirect [4] interface support peer-to-peer transfers to network cards and, with some additional effort (See Section 4.4), SSDs can support them as well. However, the current programming model requires significant re-engineering to make use of these more direct transfers.

Figure 3 quantifies the significance of data movement in nine GPGPU applications from the Rodinia benchmark

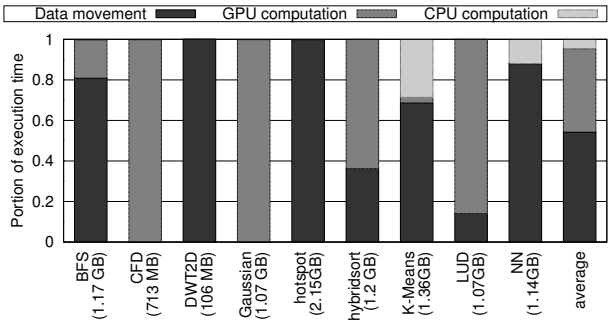


**Figure 3: The execution time breakdown of Rodinia applications**

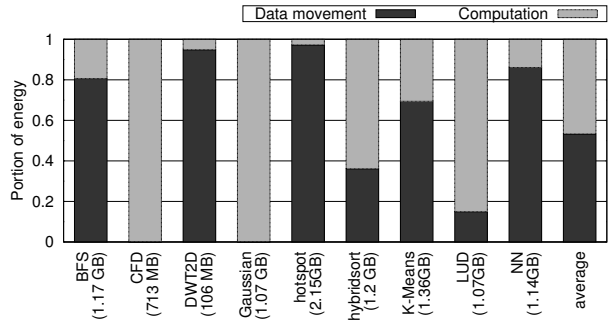


**Figure 4: The energy breakdown of Rodinia applications**

suite [1] running on a machine described in Section 5.1. It breaks down the execution time into three different parts, in which "Data movement" includes buffer allocation, data transfer over PCIe and the memory bus, and access time for the SSD and main memory. We modify Rodinia inputs to use binary file formats to improve overall performance and use `read` or `write` primitives to access the main memory locations of application data. The size of the input data set is shown in parentheses. Our experiments show that, on average, applications spend more than 55% of execution moving data. Figure 4 breaks down compute and data movement energy usage in the same workloads. Data movement accounts for 53% of total system energy.

# 3. ARCHITECTURAL SUPPORT FOR Gullfoss

To establish direct data transfers between SSDs and GPUs, Gullfoss requires architectural supports from NVMe-compliant SSDs and GPUs [4] communicating through the PCIe interconnect. This section provides a brief introduction to these architectural supports.

## 3.1 GPU supports for Gullfoss

PCIe allows each device to send data packets directly from one device to another, bypassing CPU and the main memory. Supporting this peer-to-peer data transfer model requires the devices to be able to map their device memory to PCIe base address registers (BARs) in the PCIe controller/switch.

AMD's DirectGMA and NVIDIA's GPUDirect [3, 4] are

technologies that provide interfaces for software to program PCIe peer-to-peer communication for GPUs. These technologies work together with the GPU hardware to first create a pinned location in the GPU memory. After this operation succeeds, the device driver programs the PCI BARs; the PCIe controller then makes the GPU memory region available to other PCIe devices connecting to the same controller. Any PCIe device can issue read or write requests to the addresses that GPU makes available in the PCI BAR for data interchange.

These GPU hardware and library supports allow Gullfoss to make GPU device memory available in the PCIe interconnect. However, conventional SSDs cannot access GPU device memory directly without supports like DirectNVMe that we will discuss in Section 3.2 and Section 4.4.

## 3.2 NVMe SSDs and DirectNVMe

NVM Express (NVMe) is the emerging standard for PCIe SSDs [14]. NVMe avoids the disk-centric legacy of SATA and SCSI interfaces and leverages PCIe to provide scalable bandwidth. For example, 16-lane Gen3 PCIe supports up to 15.754 GB/sec full-duplex data transfer, while SATA can typically only achieve 600 MB/sec. NVMe also supports more concurrent I/O requests than SATA or SCSI by maintaining a software command queue holding up to 64K entries for each processor core, and its command set includes scatter-gather data transfer operations with out-of-order completion, further improving performance.

Current NVMe SSDs cannot provide native support for peer-to-peer data transfers with other PCIe devices, since NVMe uses a doorbell model for PCIe communication and does not map device memory for data accesses. This is because SSDs are block devices: Their internal data arrays are not byte-addressable and require the SSD controller to translate the logic block addresses into physical block addresses. Furthermore, allowing other PCIe peripherals to access the SSD's data array directly and bypass the host operating system could result in data integrity issues.

To overcome this limitation, this paper proposes and develops *DirectNVMe*. DirectNVMe is an extension of the NVMe interface that adds new `ioctl`-based read and write commands to send data directly between SSDs and GPUs (or, in principle, other devices). These read/write commands resemble conventional read/write requests except that they use GPU memory instead of main memory as the DMA target. The conversion takes place in the DirectNVMe driver: It ensures that the GPU memory address is available for PCIe peer-to-peer transfers and then issues normal NVMe read and write commands using GPU memory addresses. When the SSD receives the command, it reads or writes data directly from or to the GPU without further involvement of the CPU or main memory. Because DirectNVMe still relies on the CPU code to issue read/write commands, DirectNVMe does not incur any new file system integrity issues.

# 4. GULLFOSS

Gullfoss decouples the control plane and the data plane for applications and manages the low-level details of scheduling and executing data transfers. This frees the CPU for more important tasks and lowers the burden on the programmer.

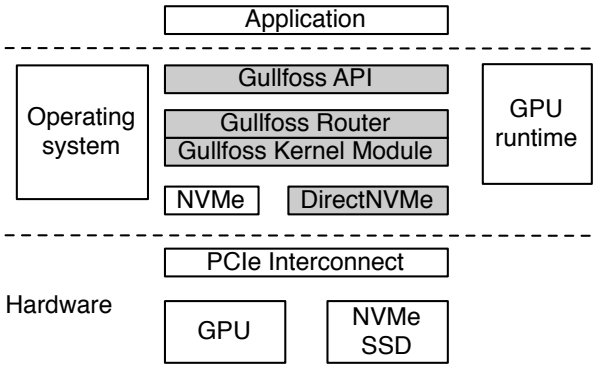


**Figure 5: The system components of Gullfoss**

In many cases, this will improve latency by eliminating unecssary copies to and from main memory. This, in turn, frees up CPU time and main memory bandwidth for more important tasks. Finally, it can save energy, since the CPU can, if there is no other work for it to do, run at a lower clock rate or be put into a sleep mode.

Gullfoss relies on four components (shown in grey in Figure 5):

*Gullfoss API:* Gullfoss provides an API for programmers to specify the sources and destinations for data transfers.

*Gullfoss Router:* The router selects the best route to carry desired data transfers. It understands the file layout in the storage devices and monitors resource availability on the destination device to decide which transfer method to use.

*Gullfoss Kernel module:* The Gullfoss kernel module maintains the runtime information from all processes using Gullfoss and allows the router to make inter-process decisions about resource allocation.

*DirectNVMe:* Gullfoss uses DirectNVMe (described in Section 3.2) to perform peer-to-peer transfers between the SSD and GPU.

We describe each of these in more detail below.

## 4.1 Gullfoss API

The Gullfoss API allows programmers to provide only the sources and the targets of data transfers, and rely on Gullfoss to handle the details.

Table 1 documents the API. The functions initialize the environment for Gullfoss, create data access requests, perform data transfers, and release the resources when Gullfoss is finished. The Gullfoss API interacts with the underlying file system, operating systems, and the Gullfoss kernel module to acquire permission for accessing files.

Figure 6 compares the source code of a simple GPU application using CUDA (a) and Gullfoss (b). The Gullfoss code is much shorter. It forgoes the `malloc` call in line 2 , the `cudaMalloc` calls of line 6, and `cudaMemcpy` calls of line 7 and 10.

The resulting code (Figure 6(b)) is much simpler. After the call to `gullfoss_init()` initializes the Gullfoss runtime system, `gullfoss_send()` sends data from the SSD to the GPU (Line 2). Once the function finishes, a

| Synopsis | Description |
|---|---|
| `int gullfoss_init()` | The `gullfoss_init()` function initializes the Gullfoss runtime. |
| `size_t gullfoss_send(const char *filename, void **gpuMemPtr, size_t offset, size_t size)` | The `gullfoss_send()` function creates the Gullfoss task of sending data from `filename` with `offset` to GPU memory location `gpuMemPtr`. If the user passes a `gpuMemPtr` containing *NULL*, Gullfoss automatically allocates the GPU memory space fits the input file or requested size. |
| `size_t gullfoss_recv(const char *filename, void *gpuMemPtr, size_t offset, size_t size)` | The `gullfoss_recv()` function creates the Gullfoss task of sending `size` bytes of data from GPU memory location `gpuMemPtr` to the `filename` with `offset`. |
| `int gullfoss_deinit()` | This function releases the resource that Gullfoss uses for an application. |

**Table 1: The Gullfoss API.**

```
1:  fp = open(filename, O_RDONLY|O_DIRECT);
2:  m = (float*) malloc(sizeof(float)*matrix_dim*matrix_dim);
3:  read(fp,m,sizeof(float)*matrix_dim*matrix_dim);
4:  close(fp);
5:
6:  cudaMalloc((void**)&d_m,
      matrix_dim*matrix_dim*sizeof(float));
7:  cudaMemcpy(d_m, m, matrix_dim*matrix_dim*sizeof(float),
         cudaMemcpyHostToDevice);
8:  lud_diagonal<<<1, matrix_dim>>>(d_m, matrix_dim);
9:
10: cudaMemcpy(m, d_m, matrix_dim*matrix_dim*sizeof(float),
                                    cudaMemcpyDeviceToHost);
11: store_matrix(m, matrix_dim);
```

(a)

```
1:  gullfoss_init();
2:  gullfoss_send(filename, (void **)&d_m, 0, 0);
3:
4:  lud<<<1, matrix_dim>>>(d_m, matrix_dim);
5:
6:  gullfoss_recv(output_filename, d_m, 0,
                    sizeof(float)*matrix_dim*matrix_dim);
7:  gullfoss_deinit();
```

(b)

**Figure 6: The CUDA and the Gullfoss programming model (a) A simplified CUDA example LU Decomposition (LUD) code (b) the Gullfoss code**

kernel running on the GPU can access the memory in `d_m` (Line 4). When the kernel completes, `gullfoss_recv()` moves data directly from the GPU to the SSD (Line 6), and releases any resources Gullfoss was using with `gullfoss_deinit()` (Line 7).

## 4.2 Gullfoss Router

The Gullfoss router receives data transfer tasks from the API, chooses the data transfer method, and schedules these tasks. The Gullfoss system provides two routes for data transfers. The preferred method is to move data using a DirectNVMe peer-to-peer transfer. The secondary, more conventional option is to send data to their destination through the main memory.

Gullfoss uses the secondary option only if the GPU has insufficient memory space to receive the incoming data. This can happen when the GPU is under heavy load. In this case, rather than holding the data while waiting to initiate a peer-to-peer transfer, Gullfoss will move the data to main memory. When the GPU memory becomes available, Gullfoss initiates a transfer from main memory to the GPU. This results in lower overall latency by overlapping SSD accesses with ongoing GPU computation; in addition, the later data transfer leverages the GPU and main memory bandwidth, which is higher than SSD bandwidth.

Likewise, upon receiving an SSD write request that writes from a GPU memory address, Gullfoss uses DirectNVMe as the default data route. However, if the Gullfoss kernel module reports that pending data read requests are already buffered because of limited GPU memory, Gullfoss will use the conventional NVMe to write data to the main memory and free up the occupied GPU location. In this case, Gullfoss can shorten the transition time between GPU kernels and then overlap the upcoming GPU kernel computation with the rest of the data write request.

To provide fair sharing among processors, an NVMe SSD periodically polls the software-maintained NVMe command queue for each processor [21]. As a result, the SSD can under-utilize both internal access and outgoing bandwidth if only one or two processes are issuing commands to the SSD. The Gullfoss router fixes this problem by querying the occupancy of the SSD's NVMe command queues. If the queues are nearly empty, the router boosts performance by running multiple DirectNVMe transfers in parallel, improving bandwidth.

## 4.3 Gullfoss kernel module

The Gullfoss kernel module serves as a synchronization point that collects information about SSD resource utilization from all running processes using the Gullfoss framework.

In Gullfoss, the kernel module forwards the I/O requests from the Gullfoss router to the underlying DirectNVMe or NVMe system components. Therefore, the Gullfoss kernel module can track the number of pending tasks and the utilization of NVMe command queues from the system.

## 4.4 DirectNVMe

DirectNVMe allows Gullfoss to perform peer-to-peer data transfers between the SSD and the GPU. Figure 7 illustrates how Gullfoss uses DirectNVMe to avoid unnecessary copies to and from main memory. After the Gullfoss API initiates a file transfer and obtains a file descriptor from the operating system (Step (1)), the Gullfoss router requests a region in the GPU memory (Step (2)). In Step (3), DirectNVMe issues NVMe read commands that include the GPU memory addresses to the SSD. Finally in Step (4), the SSD pushes data directly from the SSD to the GPU using DMA, without any CPU involvement.

To write computations to the SSD, DirectNVMe follows

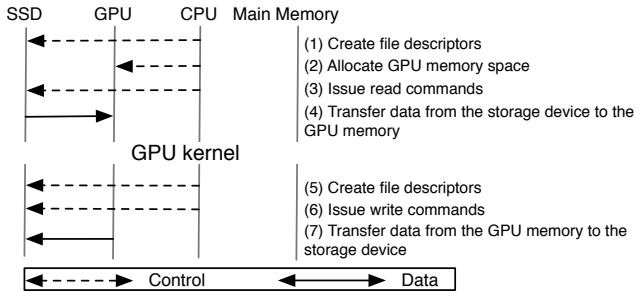


**Figure 7: The process of establishing direct data access channels for Gullfoss**

Steps (5–7), which resemble Steps (1–4) except that DirectNVMe issues NVMe write commands instead of read commands and pulls data directly from the GPU memory.

DirectNVMe requires pinning GPU device memory to a BAR in the PCIe controller, which is a slow operation [4]. Therefore, the DirectNVMe module maintains a *pinbuffer*. When an application calls `gullfoss_init()`, Gullfoss pre-allocates a pinned memory region in the pinbuffer. Future Gullfoss transfers reuse this pinned GPU memory and avoid the overhead of manipulating PCIe BARs.

Some GPUs (e.g., the Tesla K20) use a BAR to limit the GPU memory size that can be visible. For these systems, the Gullfoss router performs transfers in chunks that fit within those limits. When all the chunks have been transferred, Gullfoss uses device-to-device memory copy to move data between the pinned region of GPU memory and the actual target GPU memory. More recent GPU accelerators such as the Tesla K40 do not suffer from this limitation on BAR size.

# 5. EXPERIMENTAL METHODOLOGY

To evaluate Gullfoss, we use a test platform that contains an Intel Xeon processor, the NVIDIA Tesla K20 GPU, and a high-end PCIe-attached SSD using a PMCS controller. We select nine applications in the Rodinia benchmark suite and four workloads from the GPMR MapReduce framework and integrate the Gullfoss library into these applications. This section describes our test bed, benchmark applications, and the process of evaluating the selected benchmark applications.

## 5.1 Experimental Platform

The testing platform uses an Intel Xeon E5-2609V2 processor that contains 4 cores and a 10 MB shared L3 cache. Each processor core runs at 2.5GHz by default. Each core has its own private 32KB L1 D-cache, 32KB L1 I-cache, and 256KB L2 cache. The on-chip memory controller in this processor manages the 64GB DRAM main memory.

The GPU in our test bed is an NVIDIA Tesla K20 GPU accelerator. The GPU contains 2496 CUDA cores and 5GB GDDR5 memory on board. The GPU connects to the rest of the components in the system through 16 lanes of the PCIe interconnect, providing 8 GB/sec I/O bandwidth. The card BIOS allows at most 192 MB of device memory to be mapped to the PCIe BARs for peer-to-peer PCIe accesses.

| | Input Size | | |
|---|---|---|---|
| Application Name | Small | Medium | Large |
| Breadth-First Search (BFS) | 37 MB | 587 MB | 1.17 GB |
| Computational Fluid Dynamics (CFD) | 16 MB | 570 MB | 713 MB |
| 2D Discrete Wavelet Transform (DWT2D) | 7 MB | 27 MB | 106 MB |
| Gaussian Elimination (Gaussian) | 67 MB | 268 MB | 1.07 GB |
| HotSpot | 34 MB | 537 MB | 2.15 GB |
| Hybrid Sort | 40 MB | 200 MB | 1.2 GB |
| Kmeans | 41 MB | 136 MB | 1.36 GB |
| LU Decomposition (LUD) | 17 MB | 268 MB | 1.07 GB |
| k-Nearest Neighbors (NN) | 71 MB | 570 MB | 1.14 GB |
| GPMR-IntCount | 128 MB | 256 MB | 512 MB |
| GPMR-Kmeans | 128 MB | 512 MB | 1 GB |
| GPMR-LinReg (Linear Regression) | 128 MB | 512 MB | 1 GB |
| GPMR-MM (Matrix Multiplication) | 128 MB | 512 MB | 2 GB |

**Table 2: The applications and the input data sizes that we used in this paper.**

We use an SSD supporting NVMe 1.1 commands as the data storage. This SSD contains SLC chips with 768GB capacity available for applications. It can sustain 2.2 GB/s for both read and writes. The SSD uses 4 PCIe 3.0 lanes to provide 4GB/sec bandwidth.

We use Linux 3.16.3 in our experiments. The system uses the default "performance" governor in the Linux Intel CPU driver as the power management policy. This policy dynamically optimizes the frequencies of processor cores from 1.2 GHz to 2.5 GHz. To measure the power consumption, we use a Wattsup power meter that allows us to read the total system power each second. The test system consumes 117.6 W when idle.

## 5.2 Benchmarks

We select nine CUDA applications from the Rodinia benchmark suite [1]. This set contains all the applications in the suite that accept files as inputs and provide generators to create input data files of arbitrary sizes. Table 2 lists the applications we select and the sizes of input data we use in our experiments. We store the input data files using the binary format to eliminate the overhead of ASCII to binary translation. We use the same data input files for the "baseline" and Gullfoss-enabled versions of our benchmarks. The applications spend, on average, 55% of execution time moving data.

For each benchmark, we generate three different sizes of input data. For the small input data set, we generate files that are close to 100 MB. For the medium input data set, we generate input files with sizes ranging from 136MB–587MB. DWT2D and CFD are exceptions. For DWT2D, we use a 27 MB file for the medium input data and a 106 MB file for the large input data. For CFD, we use a 713MB input file for the large dataset. In both cases, this is because if we use GB-scale input files the working set sizes of their GPU kernels will exceed the memory capacity of the K20 GPU. This is not a limitation of Gullfoss.

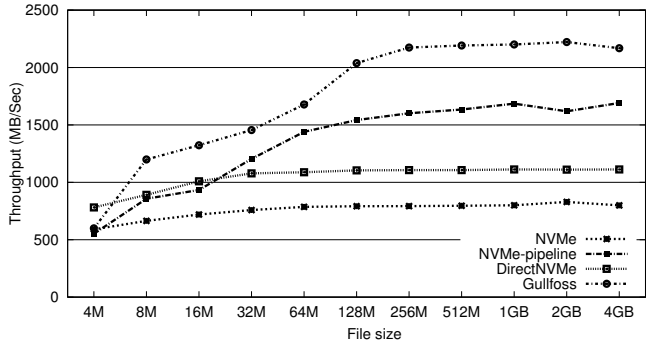


**Figure 8: The throughput of different data paths in Gullfoss**

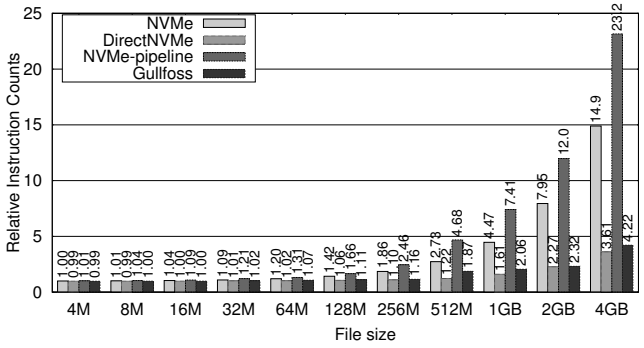


**Figure 9: The total CPU instruction counts for different data paths in Gullfoss**

We also include GPMR [17], a multiprogrammed MapReduce framework that utilizes GPU to concurrently process MapReduce tasks. As with our modifications to the Rodinia benchmark suite, when adapting GPMR we modify only the file I/O and code related to data transfers between SSDs and GPUs. We run four workloads—IntCount, K-Means, Linear Regression and Matrix multiplication—that GPMR provides. Table 2 contains additional details.

To integrate Gullfoss into these applications, we replace traditional I/O and data movement commands with with `gullfoss_send()` and/or `gullfoss_recv()`. To produce multi-program workloads from the Rodinia benchmark suite, we run them concurrently and modify the applications to ensure that the GPU kernels have sufficient input data and output memory space to complete the GPU tasks without blocking.

# 6. RESULTS

This section measures the impact Gullfoss has on performance and efficiency using microbenchmarks and the benchmarks we described in the previous section. First, we measure the impact of DirectNVMe on data transfer performance, then we report results for individual Rodinia benchmarks, multi-program workloads, and our Gullfoss-enabled MapReduce framework.

## 6.1 Performance of DirectNVMe and Gullfoss

Gullfoss supports peer-to-peer communication between the GPU and the SSD using DirectNVMe. This section presents the performance advantages of this route option and run-time optimization from the Gullfoss framework.

Figure 8 compares the throughput of moving data from the SSD to the GPU using Gullfoss against standard NVMe (NVMe), pipelined NVMe (NVMe-pipeline) that overlaps SSD access with GPU memory copy and the single channel, peer-to-peer DirectNVMe (DirectNVMe). We report the data transfer throughput under different file sizes, excluding the overhead of allocating all necessary resources (e.g. memory buffers) along the data paths. Because the K20 GPU has only 4.8GB device memory available for applications, we examine these route options with file sizes under 4GB.

Since the Gullfoss router dynamically creates multiple simultaneous DirectNVMe data transfers using free NVMe command queues, Gullfoss outperforms all other route options. The performance advantage of Gullfoss becomes more significant as file size increases. When transferring a 4GB file between the SSD and the GPU, a DirectNVMe that performs file access requests using a single NVMe command queue only achieves bandwidth of 1110 MB/sec, due to the under-utilized NVMe SSD resources. Gullfoss, on the other hand, offers up to 2221 MB/sec bandwidth.

NVMe-pipeline improves the performance of standard NVMe by compensating for latencies with multiple data transfers. However, NVMe-pipeline can still only achieve a throughput of 1691 MB/Sec between the SSD and GPU for 4GB files, 34% slower than Gullfoss, because NVMe-pipeline requires more CPU resources.

To present the CPU overheads that different data movement mechanisms consume, Figure 9 depicts the number of CPU instructions given in the experiments in Figure 8. Standard NVMe and NVMe-pipeline require the GPU runtime system to move data between DRAM and GPU memory, so they require more CPU instructions than DirectNVMe and Gullfoss. To transfer a 4GB file, standard NVMe uses $4\times$ the instructions used by DirectNVMe and $3\times$ the instructions used by Gullfoss. To streamline requests, NVMe-pipeline partitions file accesses into smaller chunks to balance pipeline stages, resulting in 60% more CPU instructions than standard NVMe.

Enabling simultaneous DirectNVMe channels achieves the best bandwidth in our tests, so we configure Gullfoss to dynamically use all free NVMe command queues for a single process by default. To make fair comparisons, the following paragraphs still present Gullfoss results in which applications use only single DirectNVMe connections, to highlight the performance difference arising from utilizing multiple command queues.

## 6.2 Single Process Workload

In this section, we evaluate the effect of Gullfoss when the system is running a single GPU process. In these experiments, we execute only one instance of these applications at a time. Therefore, Gullfoss always uses DirectNVMe to

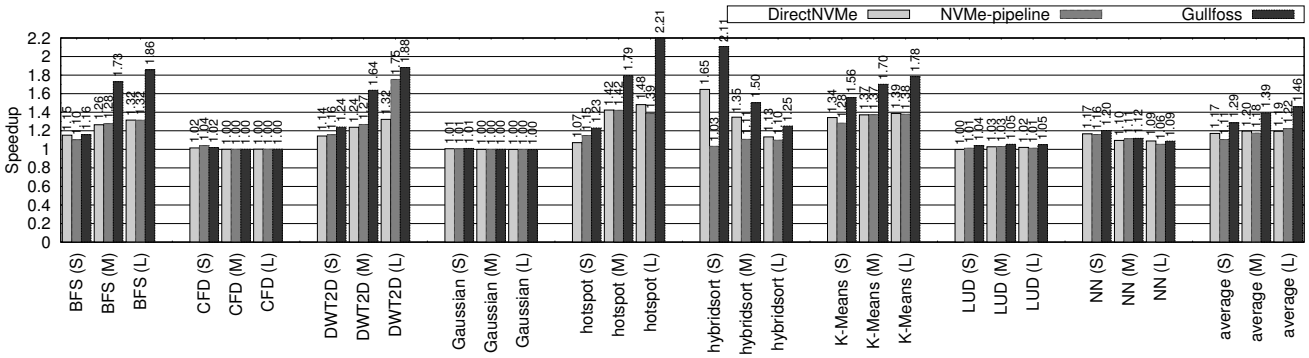


**Figure 10: The speedup of benchmark applications using Gullfoss**

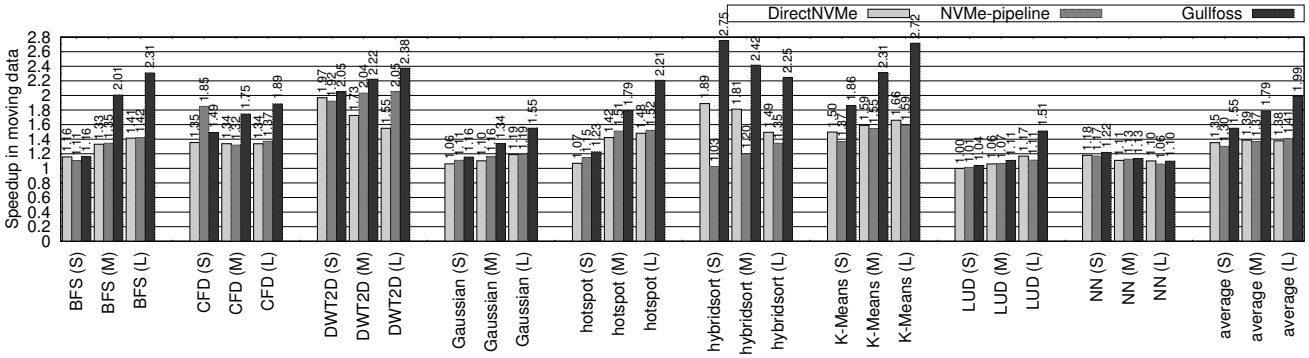


**Figure 11: The speedup of data movement in benchmark applications using Gullfoss**

move data since there is no competition for GPU resources.

### *6.2.1 Performance*

Figure 10 illustrates the speedup of using Gullfoss on our benchmarks. We measure the end-to-end latencies for each application. Gullfoss achieves an average speedup of 1.46× for the large input set. For the medium input data set and the small input data set, Gullfoss speeds up the whole application by 1.39× and 1.29× respectively. If we limit each process to use only one NVMe queue (DirectNVMe), Gullfoss still achieves an average speedup of 1.19× for both the medium and the large inputs, and 1.16× for the small data set. With pipelining, standard NVMe can slightly outperform DirectNVMe using large input data, but still falls behind Gullfoss. Figure 11 only presents the speedup of data movement in applications. We measure the end-to-end latencies for all data movement operations. Gullfoss speeds up the data movement by 2× for the large inputs.

In most cases, the impact of Gullfoss increases with data set size, since larger input files help amortize the initialization costs in Gullfoss runtime. In hybridsort and NN, the impact of Gullfoss becomes less significant because the percentage of data transfer time decreases as input sizes grow.

Table 3 shows how effectively Gullfoss reduces the main memory usage and the overhead of handling page faults. This table presents the relative number of page faults of applications using Gullfoss compared with the baseline implementation. Gullfoss reduces the number of page faults by

| Application | Relative Page Faults | | |
|---|---|---|---|
| | Small | Medium | Large |
| BFS | 67.71% | 53.04% | 49.66% |
| CFD | 100.85% | 85.24% | 83.14% |
| DWT2D | 40.60% | 40.78% | 39.37% |
| Gaussian | 81.30% | 86.63% | 90.08% |
| HotSpot | 65.23% | 57.35% | 48.27% |
| HybridSort | 83.23% | 77.48% | 71.74% |
| Kmeans | 83.10% | 83.03% | 78.06% |
| LUD | 99.77% | 94.17% | 62.91% |
| NN | 89.44% | 87.20% | 87.17% |
| Average | 79.03% | 73.88% | 67.82% |

**Table 3: The relative number of page faults of applications with different input data sets using DirectNVMe.**

21% for the small data set, 26% for the medium data set and 32% for the large data set.

### *6.2.2 Energy and Power*

Gullfoss reduces the load on CPUs during data transfers, enabling processor cores to operate at lower clock rates or perform more useful work. To explore the potential energy and power savings using Gullfoss, we measure the total system power and calculate the energy used during the entire running time of these applications.

Figure 12 presents the relative total system energy consumption of our benchmark applications using Gullfoss, Gullfoss with single DirectNVMe connection, and NVMe with pipelining, compared with a baseline that uses only conventional NVMe. We report that Gullfoss can reduce

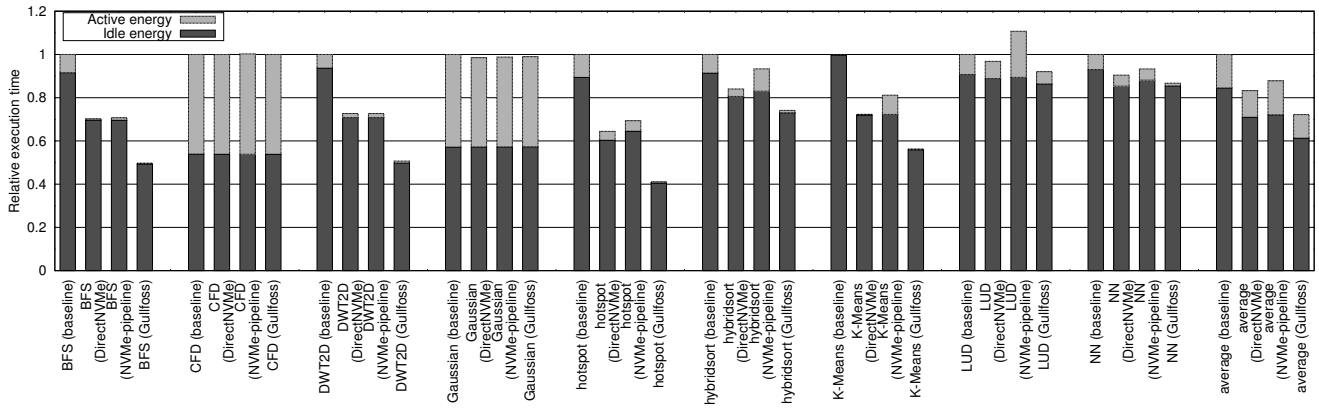


**Figure 12: The relative energy consumption of benchmark applications using Gullfoss**

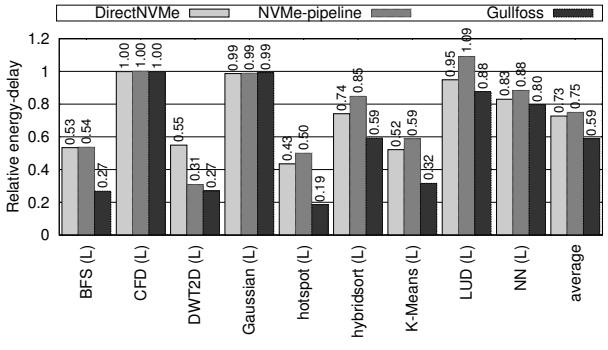


**Figure 13: The relative energy–delay product of benchmark applications using Gullfoss**

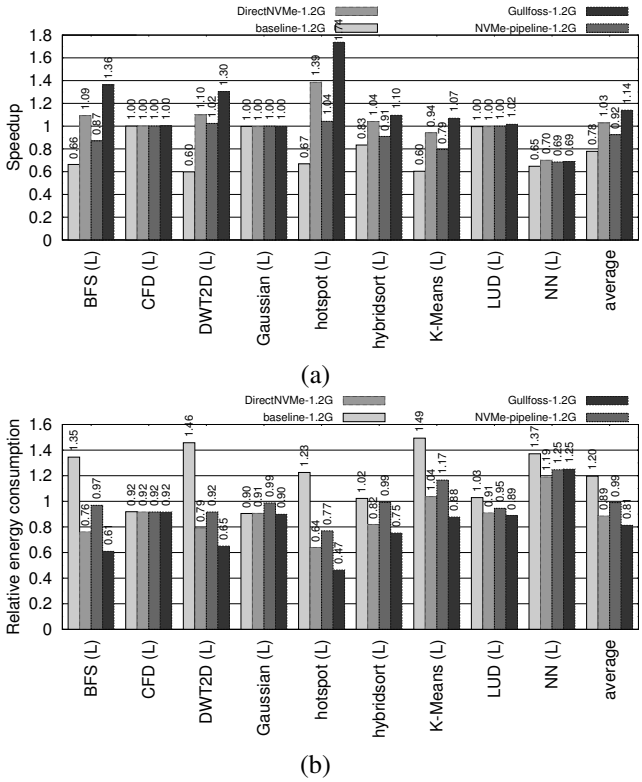


**Figure 14: The (a) speedup and (b) relative energy consumption with baseline running at 1.2GHz, single connection DirectNVMe at 1.2GHz, NVMe-pipeline at 1.2GHz and full-fledged Gullfoss at 1.2GHz**

total energy consumption by 28% when using large input data set for these applications. If we limit each process to use only one DirectNVMe connection, DirectNVMe still reduces energy consumption by 17%.

This figure also breaks down energy consumption into idle energy and application energy. We calculate application energy by subtracting system idle energy from total energy consumption. Since DirectNVMe improves the execution time of programs, we observe 27% reduction in idle energy. Even with an Intel CPU driver that optimizes for energy efficiency, Gullfoss still reduces total system power consumption by up to 8% and saves 30% of application energy over the baseline. On the other hand, while NVMe-pipeline also reduces execution time and idle energy, it consumes 2% more application energy than the baseline because of increased power consumption.

Figure 13 reports the energy–delay product. Gullfoss improves the energy–delay of applications by 41% for large data sets. Even with DirectNVMe only using one NVMe queue, we still see 27% improvement in energy–delay. Since the Intel CPU driver already aggressively optimizes for both performance and power, we see only a marginal (2%) improvement in energy consumption when using more aggressive power management policies in Gullfoss.

### *6.2.3 Gullfoss and Lower-End CPUs*

Gullfoss does not rely heavily on the CPU to exchange data between the SSD and GPU. Therefore, the system can also use a simpler but more energy-efficient processor if target workloads require only a small amount of CPU computation. This section explores the potential of using Gullfoss in servers with lower-end CPUs.

In this section, we restrict the CPU frequency in our test bed to 1.2GHz, the lowest clock rate that the processor supports. Figure 14(a) examines the performance of a baseline

system running at 1.2 GHz ("baseline-1.2G"), Gullfoss with only single DirectNVMe connections running at 1.2 GHz ("DirectNVMe-1.2G"), standard NVMe with pipelining running at 1.2 GHz ("NVMe-pipeline-1.2G"), and full-fledged Gullfoss running at 1.2 GHz ("Gullfoss-1.2G"). All of these are compared against the baseline running at 2.5 GHz. Lowering the CPU frequency to 1.2 GHz results in a 22% slowdown for baseline-1.2G, since the 52% slower clock rate increases the CPU computation time by 237% and data movement time by 41%. We also observe 8% overall performance slowdown in NVMe-pipeline for similar reasons.

Using Gullfoss, applications still suffer from the same increase in CPU computation time, but Gullfoss speeds up data movement time over baseline-2.5G by 1.33×. As a result, Gullfoss-1.2G achieves 14% performance gain. In this lower-end server setup, Gullfoss shows more performance advantages. If we use execution time of baseline-1.2G as the baseline, the Gullfoss-1.2G achieves 1.56× speedup. With the limitation of using a single DirectNVMe channel for each process, DirectNVMe-1.2G still reduces data movement time by 12% and delivers similar performance to baseline-2.5G.

Figure 14(b) shows the relative energy consumption of the above three configurations, compared with baseline-2.5GHz. For DirectNVMe-1.2G and Gullfoss-1.2G, the system can reduce energy consumption by 11% and 18%. However, baseline-1.2G consumes 20% more energy in total, even with a lower-power processor, due to the increased execution time.

## 6.3 Multi-program workload

To demonstrate the effectiveness of the Gullfoss router in a multi-program environment, we create multiprocess workloads to mimic the usage of heterogeneous computing servers. In this series of experiments we submit 8 homogeneous Rodinia benchmark tasks simultaneously with the large input data set.

Figure 15 shows the performance of these server workloads. We use the unmodified applications as the baseline and measure the end-to-end latencies of programs. To measure the effects of Gullfoss system components, we compare four different configurations of Gullfoss.
(1) DirectNVMe: we route I/O requests so as to always use a single DirectNVMe channel.
(2) DirectNVMe with Gullfoss Router: we enable the Gullfoss router but still use a single DirectNVMe channel for each process.
(3) Gullfoss w/o router: we force Gullfoss to always select DirectNVMe and establish multiple DirectNVMe channels if possible.
(4) Gullfoss: we enable all Gullfoss features.

If we allow Gullfoss to utilize multiple queues and enable the router (Gullfoss), we can achieve 1.5× speedup. Without the router dynamically selecting routes (Gullfoss w/o router), we can improve only 31% over the baseline.

If we force Gullfoss to use a single DirectNVMe connection for each process (DirectNVMe), we find that DirectNVMe can achieve only 10% performance gain over these multiprocess workloads. For hotspot and hybridsort, we observe performance loss. This is because a running GPU task uses all available GPU memory in these two applications and as a result, the DirectNVMe cannot initiate data transfers until the GPU task completes. In the baseline implementation, applications can send I/O data to main memory before the GPU resource becomes available; the main memory provides larger bandwidth than SSDs, thereby reducing the GPU idle time between tasks.

By allowing the router to dynamically select routes but requiring that it still use a single DirectNVMe for each process (DirectNVMe with Gullfoss router), Gullfoss improves the performance of DirectNVMe and achieves 1.3× of speedup—a 17% increase over DirectNVMe. This result points out that the effect of the Gullfoss router is especially important for heterogeneous servers with heavily loads.

## 6.4 GPU-MapReduce workload

To understand the impact of Gullfoss in highly-optimized GPU server workloads, we tailor the GPMR [17] framework that supports MapReduce applications on GPU to use Gullfoss for data transfers between SSDs and GPUs. For each workload, we run 4 processes in parallel. We present this set of results in Figure 15. Even though the baseline aggressively reduces file I/O operations and optimizes the overlapping between GPU data transfers and GPU kernels, Gullfoss still improves the performance by 2%–27%.

Most of these GPMR workloads, except for matrix multiplication, contain relatively fewer MapReduce tasks and file I/O operations. As a result, Gullfoss achieves limited performance gain in IntCount and K-Means on GPMR. For matrix multiplication, the workload spawns thousands of tasks and each task reads relatively larger chunks from the input files. This allows Gullfoss to achieve 1.27× speedup in matrix multiplication workload on GPMR.

Comparing the result of using Gullfoss and single connection DirectNVMe in GPMR, the improvement of I/O performance allows the GPMR to outperform single connection DirectNVMe by shrinking the gaps between GPU matrix multiplication kernels. We see little difference in performance on rest workloads, as the number of file accesses in these workloads is small.

The effect of the Gullfoss router in GPMR is limited. As GPMR aggressively preserves resources for the input data in GPU tasks but generates tasks conservatively, the Gullfoss router seldom selects the alternative route of DirectNVMe. However, we expect the router can still be beneficial if we use GPMR to perform larger-scale workloads.

# 7. RELATED WORK

Gullfoss improves performance by optimizing data transfers within the system. Most previous works considering the importance of data transfer in a GPGPU computing node focus on the bottleneck between CPU and GPU [22, 23, 24]. Several works also propose solutions [25, 26, 27, 28] to mitigate the CPU–GPU bottleneck using runtime scheduling and pipelining to allow data transfer time and computation time to overlap. Gullfoss leverages the efforts made in these works to create efficient data paths from storage device to GPU via main memory as an alternative to DirectNVMe. As a great deal of the research on high-performance computing focuses on reducing computation time on GPU [29, 30,

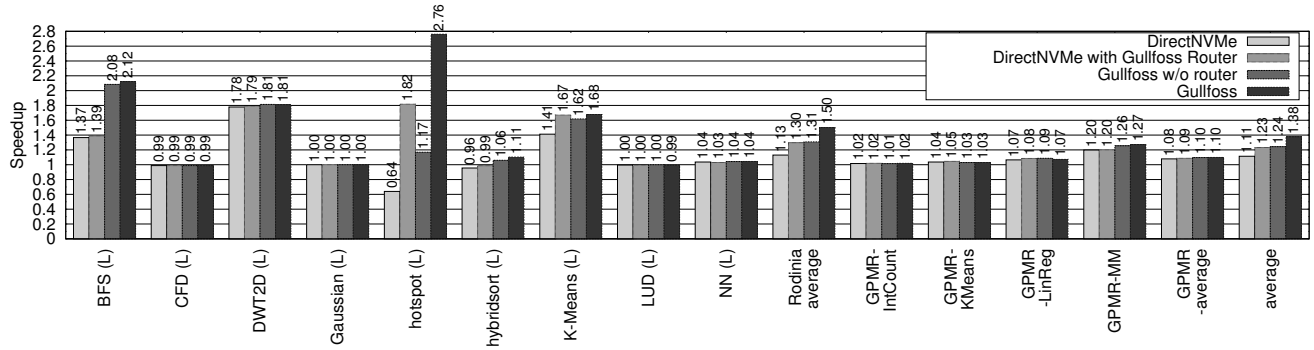


**Figure 15: The speedup of Gullfoss under multiprogram workload**

31], we expect the data movement problem to become more important.

Gullfoss is the first system we know of that implements direct data transfers between commercially available NVMe SSDs and GPUs, although AMD's DirectGMA or NVIDIA's GPUDirect provides peer-to-peer communication between two GPUs or between the GPU and an Infiniband device [3, 4] to improve inter-node communication within GPU clusters [5, 6, 7] or intra-node communication between GPUs or other devices [8, 9, 10, 11, 12, 13]. GPUDrive [32] provides similar functionality, but it uses a customized PCIe switch to provide access to SATA SSDs.

Gullfoss reduces the importance of CPU performance in heterogeneous computing platforms. The result of Gullfoss encourages researchers to revisit the design of server architectures [33]. Several server systems including FAWN [34], Gordon [35] and Blade [36] also prove the concept of using low-end CPUs for servers running data-intensive applications. Our experiments show that Gullfoss is especially effective for GPU servers using low-end CPUs.

Heterogeneous System Architecture (HSA) [37] provides another approach to eliminating data transfers between CPU and GPU. HSA integrates the CPU and GPU on the same chip and uses a unified virtual address space to share the memory. Several research papers demonstrate the potential benefits of HSA in analytical workloads and database operations [38, 39, 40]. However, with the limitation in the shared main memory bandwidth and the power constraint from dark silicon problem [41, 42], such integrated GPUs [20] can accommodate less than 20% of the streaming units and so deliver only moderate performance compared with high-end, discrete GPUs [18, 19]. In fact, due to their streaming nature [43], many GPGPU applications require higher memory bandwidth and have different memory access patterns than most CPU applications. Gullfoss will help systems with discrete GPUs deliver better performance and energy efficiency on data-intensive and GPU-intensive applications, by significantly reducing the data transfer overhead as well as by capitalizing on the superior internal GPU memory bandwidth.

# 8. CONCLUSION

The widespread use of accelerators, high-speed storage devices and peripherals in heterogeneous computing platforms calls for more advanced intra-computer interconnect as well as a thorough rethinking of the data transfer model.

This paper presents the Gullfoss system stack, which efficiently handles data transfers inside heterogeneous computers. As programming network data transfers requires only the source data and destination network address, Gullfoss needs only the source and destination locations in the computer. Gullfoss makes wise choices regarding data routes, including DirectNVMe, to directly send data between SSDs and GPUs without consuming host CPU cycles and main memory.

As Gullfoss works for commercially available hardware, we evaluate its performance on a real system. We achieve an average speedup of 1.46× for single applications processing files larger than a few GBs. Gullfoss reduces energy consumption by 28% and energy–delay product by 41% for GB-scale input data. Gullfoss is especially effective for data center servers, improving server performance by 50% for multi-program GPU workloads and 10% for MapReduce workloads on the GPU. However, even a lower performance CPU-powered server using Gullfoss can still outperform a high-end server by 14%.

# 9. REFERENCES


[1] M. B. S. Che, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '09, pp. 44–54, Oct 2009.

[2] PCI-SIG, "PCI-Express Specification." https://www.pcisig.com/specifications/pciexpress/.

[3] Advanced Micro Devices, Inc., "FirePro DirectGMA Technical Overview." http://developer.amd.com/tools-and-sdks/graphics-development/firepro-sdk/firepro-directgma-sdk/, 2014.

[4] NVIDIA Corporation, "Developing a Linux Kernel Module Using RDMA for GPUDirect." http://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf, 2014.

[5] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters," *Computer Science - Research and Development*, vol. 26, no. 3-4, pp. 257–266, 2011.

[6] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. Lo Cicero, A. Lonardo, E. Mastrostefano, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Gpu peer-to-peer techniques applied to a cluster interconnect," in *IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pp. 806–815, 2013.

[7] J. Jenkins, J. Dinan, P. Balaji, N. Samatova, and R. Thakur, "Enabling fast, noncontiguous gpu data movement in hybrid

mpi+gpu environments," in *2012 IEEE International Conference on Cluster Computing*, CLUSTER, pp. 468–476, Sept 2012.

[8] J. Stuart and J. Owens, "Multi-GPU MapReduce on GPU Clusters," in *2011 IEEE International Parallel Distributed Processing Symposium*, IPDPS ' 11.

[9] S. Potluri, H. Wang, D. Bureddy, A. Singh, C. Rosales, and D. Panda, "Optimizing mpi communication on multi-gpu systems using cuda inter-process communication," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, IPDPSW ' 12.

[10] L. Oden and H. Froning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *2013 IEEE International Conference on Cluster Computing*, CLUSTER ' 13, 2013.

[11] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 2595–2605, Oct 2014.

[12] R. Bittner, E. Ruf, and A. Forin, "Direct GPU/FPGA Communication Via PCI Express," *Cluster Computing*, vol. 17, pp. 339–348, June 2014.

[13] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pp. 170–178, 2013.

[14] Amber Huffman, "NVM Express Revision 1.1." http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf, 2012.

[15] Intel, "Intel(R) Solid-State Drive DC P3700 Series Specifications." http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf, 2015.

[16] Samsung, "Industry's First NVMe Solid State Drive." http://www.samsung.com/global/business/semiconductor/file/product/XS1715_ProdOverview_2014_1.pdf, 2012.

[17] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, (Washington, DC, USA), pp. 1068–1079, IEEE Computer Society, 2011.

[18] AMD, "AMD FirePro(TM) W9100 Workstation Graphics." http://www.amd.com/Documents/FirePro_W9100_Data_Sheet.pdf, 2014.

[19] NVIDIA, "Whitepaper NVIDIA's Next Generation CUDA(TM) Compute Architecture: Kepler TM GK110/210." http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf, 2014.

[20] AMD, "Compute Cores White Paper." https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, 2014.

[21] D. B. M. Jisoo Yang and F. Hady, "When Poll Is Better than Interrupt," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST' 2012, p. 3, 2012.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General-purpose Applications on Graphics Processors Using CUDA," *J. Parallel Distrib. Comput.*, vol. 68, pp. 1370–1380, Oct. 2008.

[23] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *2011 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pp. 134–144, April 2011.

[24] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pp. 163–174, April 2009.

[25] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pp. 354–365, Feb 2013.

[26] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pp. 55–62, 2012.

[27] F. Song and J. Dongarra, "A scalable approach to solving dense linear algebra problems on hybrid cpu-gpu systems," *Concurrency and Computation: Practice and Experience*, 2014.

[28] R. Mokhtari and M. Stumm, "BigKernel – High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications," in *IEEE 28th International Symposium on Parallel and Distributed Processing Symposium*, IPDPS '14, pp. 819–828, May 2014.

[29] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code," in *2010 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2010.

[30] R. Salomon-Ferrer, G. A. W., D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald," *Journal of Chemical Theory and Computation*, vol. 9, no. 9, pp. 3878–3888, 2013.

[31] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pp. 117–128, 2012.

[32] M. Shihab, K. Taht, and M. Jung, "Gpudrive: Reconsidering storage accesses for gpu acceleration," in *Workshop on Architectures and Systems for Big Data*, 2014.

[33] M. Arora, S. Nath, S. Mazumdar, S. Baden, and D. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, vol. 32, pp. 4–16, Nov 2012.

[34] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A Fast Array of Wimpy Nodes," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 1–14, 2009.

[35] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," in *the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pp. 217–228, 2009.

[36] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pp. 315–326, 2008.

[37] AMD, "Heterogeneous System Architecture: A Technical Review." http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf, 2012.

[38] M. Daga, A. Aji, and W.-C. Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing," in *2011 Symposium on Application Accelerators in High-Performance Computing*, pp. 141–149, July 2011.

[39] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "CPU-assisted GPGPU on Fused CPU-GPU Architectures," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pp. 1–12, 2012.

[40] J. He, M. Lu, and B. He, "Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture," *Proc. VLDB Endow.*, vol. 6, pp. 889–900, Aug. 2013.

[41] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pp. 365–376, 2011.

[42] M. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *49th ACM/EDAC/IEEE Design Automation Conference*, DAC ' 2012.

[43] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 457–467, 2013.