Maximizing I/O Bandwidth for Reverse Time Migration on Heterogeneous Large-Scale Systems

Tariq Alturkestani, Hatem Ltaief, and David Keyes

Extreme Computing Research Center
King Abdullah University of Science and Technology
Thuwal, Saudi Arabia
`{firstname.lastname}@kaust.edu.sa`

Abstract. Reverse Time Migration (RTM) is an important scientific application for oil and gas exploration. The 3D RTM simulation generates terabytes of intermediate data that does not fit in main memory. In particular, RTM has two successive computational phases, i.e., the forward modeling and the backward propagation, that necessitate to write and then to read the state of the computed solution grid at specific time steps of the time integration. Advances in memory architecture have made it feasible and affordable to integrate hierarchical storage media on large-scale systems, starting from the traditional Parallel File Systems (PFS) to intermediate fast disk technologies (e.g., node-local and remote-shared Burst Buffer) and up to CPU main memory. To address the trend of heterogeneous HPC systems deployment, we introduce an extension to our Multilayer Buffer System (MLBS) framework to further maximize RTM I/O bandwidth in presence of GPU hardware accelerators. The main idea is to leverage the GPU's High Bandwidth Memory (HBM) as an additional storage media layer. The objective of MLBS is ultimately to hide the application's I/O overhead by enabling a buffering mechanism operating across all the hierarchical storage media layers. MLBS is therefore able to sustain the I/O bandwidth at each storage media layer. By asynchronously performing expensive I/O operations and creating opportunities for overlapping data motion with computations, MLBS may transform the original I/O bound behavior of the RTM application into a compute-bound regime. In fact, the prefetching strategy of MLBS allows the RTM application to believe that it has access to a larger memory capacity on the GPU, while transparently performing the necessary housekeeping across the storage layers. We demonstrate the effectiveness of MLBS on the *Summit* supercomputer using 2048 compute nodes equipped with a total of 12288 GPUs by achieving up to 1.4X performance speedup compared to the reference PFS-based RTM implementation for large 3D solution grid.

Keywords: Multilayer Buffer System, Reverse Time Migration, Asynchronous I/O Operations, Hierarchical Storage Media, Heterogeneous Systems.

1 Introduction

The execution rate of floating-point operations has typically increased by an order of magnitude every 4 years during the last 30 years of modern computing [4]. This exponential growth in terms of computational power has benefited from processor technology scaling. However, memory and storage systems have not maintained the same rate of technology scaling. Scientific applications that are traditionally compute-bound have undergone a paradigm shift toward an I/O-bound regime. This has been further exacerbated by the limited on-node memory capacity, where I/O operations become a major bottleneck for scaling up critical simulations.

Performing I/O operations is usually necessary for post-processing tasks (e.g., visualization [12]) or for checkpointing the entire application’s state (e.g., resilience [16]). For seismic depth imaging simulations in the context of oil and gas exploration, I/O operations play a pivotal role and one that is increasing with the technology scaling gap. The challenge with Reverse Time Migration (RTM), the *de facto* method for subsurface imaging, resides in continuously intertwining computational phases with I/O operations during the time integration.

Based on an adjoint-state formulation, the RTM relies on a finite difference stencil time integration explicit solver to simulate the forward and backward 3D wave propagations. RTM requires combining at regular time steps a forward-propagated source wavefield with a backward-propagated receiver wavefield. This process involves thus a first phase where the 3D domain solutions, or snapshots, of the source wavefield are computed and stored at predetermined imaging time steps. Then, in a second phase, the field data are injected at the receiver locations to compute the receiver wavefield. At each imaging time step, the source snapshots are retrieved from storage, brought in to main memory, and correlated with the receiver snapshots. The combination of the correlated snapshots results in the final image. RTM runs typically for thousands of time steps in production and thus requires out-of-core computations [27] since not all subsequent snapshots can be kept in main memory. To give the proper perspective, we profile representative test cases from a seismic imaging campaign: the I/O time spent during the two aforementioned phases account for about 70% of the entire execution time. The overall RTM performance may therefore be mostly driven by I/O storage subsystem bandwidth. Nor is this I/O dominance special to RTM. As described in [21], in monitoring over 17,000 executions of an earth science code on a DOE petascale system, more than half of the wall-clock time was spent on I/O.

Previously deployed large-scale systems include several layers of hardware storage media, from Dynamic Random Access Memory (DRAM) to node-local /remote-shared Burst Buffer [11, 19] to traditional Parallel File Systems (PFS), and all the way down to storage disks. More recently, especially with the advent of AI workloads, GPU-based supercomputers have been considered as one of the main sources of horsepower for the Exascale quest. In this paper, we propose to extend the Multilayered Buffer Storage (MLBS) software library [6] to support I/O operations involved in GPU application workloads. The objective of

Multilayer Buffer System (MLBS) is to maximize the bandwidth utilization of multiple hardware storage media layers, while prefetching the I/O operations for possible overlap with computations. The main idea herein consists of including the High Bandwidth Memories (HBMs) as an additional storage media layer to MLBS. In the context of the RTM application, this means the write (i.e., modeling phase) and read (i.e., propagation phase) operations may potentially be overlapped with the stencil computation kernels running on the GPU devices. In fact, the ultimate goal of the MLBS approach is to ensure that the data is always GPU-resident whenever the RTM needs it in order to mitigate the overhead of out-of-core computations. Our lightweight, non-intrusive MLBS framework provides simple APIs that enable the RTM application to decouple these I/O operations from its critical path, while eagerly pursuing its forward and backward time integration.

We demonstrate the performance superiority of the RTM application with MLBS using 2048 compute nodes, i.e., a total of 12288 NVIDIA V100 GPUs, on the *Summit* supercomputer. By using the GPU’s HBMs as an additional storage media layer, we report up to 1.4X performance speedup compared to the reference Parallel File System (PFS)-based RTM implementation for large 3D solution grid, respectively. Last but not least, we provide an online autotuner for MLBS to select at runtime the optimal storage media layers to maximize the overall I/O throughput.

The remainder of the paper is as follows. Section 2 discusses background on high performance storage media and related work. Section 3 states our main contributions. Section 4 gives details on the RTM application. Section 5 highlights the design and implementation of the MLBS GPU support. Section 6 analyzes the performance results of the MLBS-enabled RTM application on homogeneous and heterogeneous systems, i.e., *Shaheen-2* and *Summit* systems, respectively. Section 7 summarizes the paper and describes future work.

2 Background and Related Work

Burst Buffers and High Bandwidth Memory. Due to the importance of closing the gap between I/O and compute throughput, several production supercomputers are currently equipped with high bandwidth intermediate memory and storage subsystems, e.g., High Bandwidth Memory (HBM), DRAM and Burst Buffer (BB) [20]. In particular, many supercomputers are now composed of heterogeneous compute nodes that host multiple Central Processing Units (CPUs) equipped with General Processor Units (GPUs), as seen in Fig. 1. While most of the parallel storage systems in supercomputers are built with cost-effective, low-bandwidth, high-capacity spinning Hard Disk Drives (HDDs), Burst Buffers are built with more expensive but high-bandwidth, low-capacity Solid State Drive (SSD) or Non-Volatile Memory (NVMe) [2]. The two most common types of Burst Buffers are node-local and remote-shared. In the design of the former, as shown in Fig. 1(a), each node is equipped with a local SSDs or NVMe with limited capacity. In the design of the latter, as pictured in Fig. 1(b),

Fig. 1. Designs of (a) Remote-Shared and (b) Node-Local Burst Buffer (c) Heterogeneous compute node similar to *Summit* system.

arrays of SSDs or NVMe are grouped in separate Burst Buffer nodes. These Burst Buffer nodes are located on the interconnect fabric and compute nodes can access them through the network. For instance, on *Summit* supercomputer, each compute node has 6 NVIDIA V100 GPUs with a local 1.6 TiB NVMe, as highlighted in Fig. 1(c). On *Shaheen-2* supercomputer, all compute nodes share 1.5 PiT of distributed high bandwidth SSDs located on the remote Burst Buffer nodes.

Fig. 2. Observed compute and I/O traces for a master iteration of RTM using *Shaheen-2* supercomputer with Lustre Parallel file system. Red and pink are forward and backward computation, respectively. Blue and green show the write and read I/O operations, respectively.

Runtime opportunities for compute and I/O overlap. In Fig. 2, we provide a typical execution trace of the RTM application, which alternates each compute and I/O phase. In the forward phase, the application computes several iterations before triggering the I/O library to dump the grid solution. The library then conducts the actual writes to the closest storage media (e.g., HBM) before falling back to the next closest storage media (e.g., DRAM) in case the current storage media is full. Once the forward phase is over, in the backward simulation

and imaging phase, the application reads back previously written grids in Last In, First Out (LIFO) order and computes the so-called imaging condition at the same time step during the forward phase. Red and pink colored bars represent compute times for forward and backward phases, and blue and green represent **write** and **read**, respectively. We can clearly observe how the backward phase reads may be faster in the beginning since the latest snapshots may already be cached. But then, they eventually become slower due to snapshot cache misses. The objective is to overlap the red and pink bars with the blue and green bars to speed up the overall time to solution.

Current solutions. There are many system-level and hardware-level solutions to reduce the widening performance gap between I/O and compute operations. Many HPC centers have adopted multiple storage layers in their systems. Ranging from HBM to node-local and remote-shared NVMe and SSDs, these layers are helping to reduce the performance gap between I/O and compute operations. Systems such as BurstFS and BurstMem [33, 34] propose to redirect I/O calls from PFS to local NVMe and SSDs. Burst Buffer caching systems such as DDN IME [1], DataWarp [15], and Data Elevator [13] accelerate applications writes by redirecting writes from PFS to remote-shared Burst Buffers. GPUDirect Storage [29] is a recently proposed system that moves data between GPUs and a specific local or remote storage without involving the CPU’s memory, thereby removing the need for extra data copies. Other systems such as SSDalloc, NVMalloc [9, 31] extend the virtual memory space from DRAM only to a wider range of fast memory such as fast SSD and NVMe. Systems in [17, 25] have been proposed to show the benefit providing hints for read prefetchers in I/O libraries. GPUfs [28] is a file system that extends traditional file systems and enables asynchronous data swaps between GPUs memory and their internal I/O buffers. More recent works such as Hermes [18], UniviStor [32] and ARCHIE [14] provide buffering solution across all stages of storage layers. They use dedicated processes to perform the caching and prefetching. TB-RTM [5] is an approach that relies on StarPU [8] to taskify and overlap all routines of RTM applications, including I/O operations. Since it is application-oblivious, it does not take advantage of the First In, First Out (FIFO) and LIFO patterns seen in the RTM application.

In contrast to previous work, MLBS is a software-level library solution that consists in overlapping I/O and compute for out-of-core simulations. MLBS provide a holistic solution to enhance the time-to-solution of real multi-threaded distributed-memory GPU-based scientific applications such as RTM using both write behind [7] and prefetching mechanisms. These optimizations permit to stream data across several stacked memory/storage layers (i.e., HBM/DRAM /Burst Buffer (BB)/PFS), while maximizing the throughput at each encountered hardware layer. MLBS is able to properly balance the I/O and compute loads, with minimal and controlled impact on the application computation phases.

3 Contributions

The main contributions of this paper are as follows:

- We introduce the GPU support in MLBS for caching and prefetching I/O operations, in the context of the RTM application.
- We add support for HMBs as a new storage media layer, in addition to the CPU main memory and various BB layers, i.e., node-local and remote-shared, and evaluate its performance impact.
- We evaluate the MLBS scalability on homogeneous *Shaheen-2* and heterogeneous *Summit* systems using up to 2048 compute nodes.
- We deploy an online autotuner into MLBS that monitors and reorders the different storage layers based on their sustained I/O bandwidth.

4 Seismic Depth Imaging Application

High resolution depth images of the subsurface are critical for successful oil and gas exploration. Wave equation-based depth seismic migration techniques such as Reverse Time Migration (RTM) are often employed to meet the oil industry imaging objectives, particularly in complex geological environments such as subsalt exploration. Given a model of the subsurface and seismic data recorded at known receiver locations, an image of the subsurface is formed by combining with an imaging condition, a forward propagated source wavefield with a backward propagated receiver wavefield. Reviews of seismic migration techniques and imaging conditions can be found in [10,26]. The source and receiver wavefields are computed by solving the wave equation using time or frequency domain solvers. Time domain finite differences solvers are commonly used due to their simplicity and computational efficiency. Second to fourth order stencils are usually considered to compute time derivatives and spatial derivatives are computed using up to sixteenth order stencils. The RTM image is formed by cross-correlating the forward simulated source wavefield with an adjoint (i.e., reverse-time) simulation from receivers. In a first phase, the source wavefield is reversed in time by storing at predetermined imaging time steps (snapshots) the propagation history. Due to the huge volume represented by the snapshots, they are usually offloaded to disks. In a second phase, when the backward receiver propagation reaches one of the imaging time steps, the corresponding source snapshots are read back from disk, and correlated with the receiver snapshots to incrementally calculate the image condition, until the final image is eventually obtained. RTM usually suffers from I/O performance bottlenecks due to unnecessary system synchronizations, which prevent overlapping with computations. The main idea then consists in creating two swim lanes by decoupling the I/O operations from the computational kernels. The final image is formed by combining the image contribution from several seismic gathers which are simultaneously simulated in an embarrassingly parallel fashion.

Algorithm 1: MLBS API functions: Init, Write and Read

```

1 Function Init( datasize, ioOps)
2   SPAWNEngine(MLBSEngine, datasize, ioOps)
3 Function inGPUParallelCopy( src, dest, len)
4   idx = threadIdx.x + blockIdx.x * blockDim.x;
5   if idx < len then
6     dest[i] = src[i]
7 Function Write(filename, data, datasize):
8   dest ← FINDEMPTYSLOTINHBM() // blocking call
9   // GPU based parallel memory to memory copy. Similar method is used on CPUs
10  INGPUCOPY<<<DIMGRID,DIMBLOCK>>>>(data, dest, datasize)
11 Function Read(filename, data, datasize):
12   while file is not yet loaded in MLBS HBM // blocking busy loop
13   do
14     WAIT( )
15   src ← GETFILEPOINTER(filename)
16   SWAP(src, data)

```

---

5 Design and Implementation of a Multilayered Buffer System

MLBS handles all I/O requests for the RTM application while maintaining the following goals: 1) lightweight Application Program Interface (API), 2) overlapping I/O and compute phases using asynchronous executions, 3) prefetching across multi storage layers with minimal impact on the computation kernels. The design of MLBS relies on two components: an API, memory and storage interfaces, and a helper thread engine. MLBS is written in C++ and CUDA and is deployed as a light-wight library.

Application Interface. The main objectives of the MLBS APIs are ease-of-use, non-intrusiveness and high throughput copy from application space to the fastest layer allocation in MLBS space. MLBS uses a simple API that mimics C/C++’s **write()** and **read()** interfaces. Therefore, with minimal code changes, application developers can easily integrate MLBS into their codes. The API requires two inputs from the application at the initialization stage: 1) the size of each data file, and 2) the number of write and read requests. Once the parameters are set, the helper engines are spawned and start their tasks, as detailed in in Algorithm 2. Higher throughput for **write()** requests is achieved with a parallel copy of the targeted data using all hardware threads whether the data were located on a CPU or a GPU (see **write** method for GPU based kernel in Algorithm 1). On the other hand, **read()** requests depend heavily on the prefetching strategy employed by the helper engines, the MLBS first layer allocation and the rate of consumption (compute kernel speed) of prefetched data. Ideally, from the application’s point of view, a **read()** operation is as fast as a memory pointer swap (see **read** method in Algorithm 1).

Helper Threads. The objective of the helpers is to handle all I/O operations: 1) moving the data between the application space and MLBS space, 2) conducting the actual push and pull operations on all available memory and

Algorithm 2: MLBS Helper Engine for RTM on *Summit* System

Data: Size of grids, number of I/O writes and reads

```

1 begin
2   layers  $\leftarrow$  MLBSAuto() // At run time, survey all available memory and storage
                               // layers and create a list ranked from fastest to slowest
3   GridsInLayers  $\leftarrow$  ... // calculate the number of grids to keep in each layer
                               // based on grid size and available space
4   while numPushedGrids < numGridsToPush do
5     push grid to first layer
6     if HBM layer (1st layer) is full then
7       select oldest grid in HBM layer and push to DRAM layer
8       if DRAM layer (2nd layer) is full then
9         select oldest grid in DRAM layer and push to node-local BB layer
10        if node-local BB layer (3rd layer) is full then
11          select oldest grid in node-local BB layer and push to IBM Spectrum
              Scale (4th) layer // Depending on MLBS Auto survey 3rd
                               // and 4th layers could in the opposite order
12    numPushedGrids  $\leftarrow$  numPushedGrids + 1
13    // Switching to read mode
    numPulledGrids  $\leftarrow$  0
14    while numPulledGrids  $\neq$  numPushedGrids do
15      if first layer is not full // Read operation in the application space frees up
          first layer buffers
16      then
17        for  $i \leftarrow 1; i < \text{numOfLayers}$  do
18          pull newest grid from  $(i + 1)$ th layer to  $i$ th layer
19        numPulledGrids  $\leftarrow$  numPulledGrids + 1

```

---

storage layers, 3) moving data across storage layers such as HBM, DRAM and PFS from fastest to slowest, and 4) caching in and out data based on RTM grids (i.e. snapshots) access pattern. Algorithm 2 details how MLBS overlaps compute and I/O with LIFO data access pattern, which matches the I/O pattern of the RTM application. From the application developer's inputs, such as the total number of grids and the size of each grid, the number of write and read operations, MLBS helper can then manage the data traffic in the background, while the application carries on its computation. In particular, MLBS helper engine consists of three phases. In the first phase, the engine surveys all available memory and storage layers and creates a list of layers ranked from fastest to slowest (line 2). For example, on *Summit* system the order of BB and PFS (line 11) could be exchanged depending of the runtime sustained throughput as seen in the evaluation in Section 6. In the 2nd phase, the engine loops over the 1st layer of MLBS, and evicts oldest data to the next layer of MLBS, e.g., BB or PFS (line 7). It may further push down the oldest files from 2nd layer to the 3rd layer, and repeat to the last layer. In the 3rd phase, i.e., the read phase, the helper engine wait for the application to consume the data that are still in the MLBS 1st layer. As soon as a buffer slot is empty in the 1st layer, the engine fills up the buffer slot with the newest data file from the 2nd layer (line 16). It will also pop up the newest data file from the 3rd layer to the 2nd layer and repeat the same routine for all layers. MLBS maintains a full pipeline across storage layers, while maximizing I/O bandwidth and occupancy.

Algorithm 3: Integrating MLBS into RTM Code on *Summit*

Data: size of grids, number of I/O ops

```

1 Function main()
2   MLBSHandler  $\leftarrow$  MLBSINIT(datasize, ioOps)
   // setup all RTM related parameterscopy necessary RTM initial data to GPU memory
   // start forward modeling
3   for i = 0  $\rightarrow$  numOfSnaps do
4     INGPUCOMPUTEFWD<<<DIMGRID,DIMBLOCK>>>(data, datasize)
5     if (i mod snapRatio) == 0 then
6       MLBSHandler.WRITE(i, data, datasize)
   // start backward propagation
7   finalImage  $\leftarrow$  EMPTYDATAARRAY(datasize)
8   for i = numOfSnaps  $\rightarrow$  0 do
9     INGPUCOMPUTEBKW<<<DIMGRID,DIMBLOCK>>>(data, datasize)
10    if (i mod snapRatio) == 0 then
11      MLBSHandler.READ(i, datafwd, datasize)
12    finalImage  $\leftarrow$  CORRELATE(finalImage, data, datafwd)

```

MLBS Integration. The RTM application presented in this work has two computation kernel engines, GPU and CPU based kernels that rely on CUDA cores and OpenMP programming models for the parallel computational kernel implementation, i.e., the stencil computation kernel. MLBS uses a Pthread-based implementation for flexibility purposes. To ensure the CPU based kernel OpenMP threads and MLBS Pthreads properly coexist, the total number of threads should match the number of physical cores, which prevents oversubscription overheads. Only the single-threaded MLBS implementation is demonstrated in this paper to avoid performance slowdown on the compute kernel side, since it is part of the critical path. However, the MLBS Pthread-based implementation can support more threads. Ultimately, the makespan of the RTM should be minimized by balancing compute and I/O workloads accordingly through MLBS. Algorithm 3 presents the pseudo-code of the RTM application with MLBS API integration. This pseudo-code assumes a single shot simulation performed on a shared-memory node, without loss of generality, since multiple shots are processed in an embarrassingly parallel fashion. One observes the possible overlapping between lines 4 and 6 as well as between lines 9 and 11 during the forward and the backward integration, respectively. The final image is then generated thanks to the image condition, as shown in line 12.

Given these generic components, MLBS components may be further extended to support various applications, beyond the herein studied RTM application. Thanks to the helper threads, its API permits application developers to instrument the I/O accesses with the flexible memory interface, while customizing the helper engine to match the I/O patterns.

6 Performance Results

6.1 Hardware Settings

We have used the homogeneous Cray XC40 *Shaheen-2* and the heterogeneous IBM-built *Summit* supercomputers as our experimental platforms.

Shaheen-2 is a 6174-node system, each with a dual-socket 16-core Intel Haswell E5-2698 V3 CPU and 128 GB DRAM. *Shaheen-2* operates two PFSs, a 1.5 PiB remote-shared SSD-based DataWarp BB and a 17.6 PiB Hard Disk Drive (HDD)-based Lustre [22]. The sustained bandwidth of DRAM based on the Stream benchmark [24] is 105.9 GiB/s. The I/O bandwidth measured with the IOR benchmark [3] is 1.5 TiB/s and 500 GiB/s, for the DataWarp and Lustre, respectively.

Summit is a 4356-node system, each with a dual-socket 22-core IBM Power 9 CPU, 6 NVIDIA Volta V100 GPUs, and 512 GB DRAM. *Summit* operates two, parallel file systems, a 7.4 PiB node-local SSD-based BB and a 250 PiB HDD-based IBM Spectrum Scale file system. The sustained bandwidth of DRAM is reported to be 272.9 GiB/s [30]. The I/O bandwidth is 9.7 TiB/s and 2.5 TiB/s, for the BB and IBM Spectrum Scale, respectively.

6.2 RTM Application Setup

We evaluate MLBS performance impact on a realistic RTM code that we have developed. The code is written in C++/CUDA and uses MPI to assign each shot to a rank. The computation kernels in this research code use single-precision arithmetic and run on CPUs and/or GPUs. In this evaluation, we set the number of time steps to 2000 and we performed I/O write (in the forward phase) or read (in the backward phase) operations at every 10 time steps. We report only the I/O throughput, since the CPU and GPU kernels are not touched. Each measurement is repeated three times and the average is reported.

On *Shaheen-2*, each node works on one shot and we set the grid size of each shot to $800 \times 800 \times 800$. We then increase the number of nodes from 1 to 2048. With a total of 200 snapshots per shot, each node writes to storage 368 GiB of intermediate data in order to produce the desired seismic image. When 2048 shots are dispatched a total of 772.52 TiB is written to storage.

On *Summit*, each node works on 6 shots simultaneously and we set the grid size of each shot to $700 \times 700 \times 700$. We then increase the number of node from 1 to 2048. With a total of 200 snapshots per shot, each node writes to storage 2.3 TiB of intermediate data in order to produce the desired seismic image. When 12,288 shots are dispatched a total of 2.6 PiB is written to storage.

Additionally on *Summit* system, using one compute node and one GPU, we evaluate the performance of caching and prefetching using the HBM layer with two small grids, $256 \times 256 \times 64$ and $512 \times 512 \times 128$. This configuration is particularly of interest for domain decomposition when running in strong scaling mode of operation. The number of times steps is increased to 3,500 and we perform I/O at every 10 time steps. The former grid generates 10 GiB

Fig. 3. Per-node **write** (top) and **read** (bottom) throughput in GiB/s for the RTM app on *Shaheen-2*. Every node executes one shot with grid size $800 \times 800 \times 800$ and I/O at every 10th iteration. Burst Buffer is remote-shared.

of intermediate snapshots, which can fit entirely on the HBM, while the latter generates 87 GiB, that need to be caching in and out through the different layers of MLBS (i.e., DRAM and PFS). We report the corresponding I/O throughput and the compute times.

6.3 RTM on Shaheen-2 and Summit

We measure the aggregate and per-node I/O throughput of the RTM application on *Shaheen-2*'s PFSs, i.e., Lustre and DataWarp. All solution grids are written or read directly to/from the file system. We run the RTM application on *Summit* and collect I/O throughput as well. We then integrate the Pthread-based MLBS implementation into each rank of the RTM application on *Shaheen-2* and *Summit*. We remove one thread from the RTM pool of OpenMP threads and give it to MLBS engine as a helper thread. This does not affect the performance of the computational kernel on *Shaheen-2*, since the stencil kernel is memory-bound, while fully utilizing the vector units for SIMD. On *Summit*, the CPU is only in charge of launching the CUDA stencil kernel. MLBS can therefore use all IBM Power9 CPUs available in the hosts to improve the performance of the memory copies. We test the I/O throughput impact when we allocate 70% of the

Fig. 4. Per-node **write** (top) and **read** (bottom) throughput in GiB/s for the RTM app on *Summit*. Every node executes six shots, one shot/GPU with grid size $700 \times 700 \times 700$ and I/O at every 10th iteration. Burst Buffer is node-local. MLBS uses HMB and DRAM as the first two layers.

physical DRAM capacity for MLBS. In our benchmark campaign, we present the HDD-based PFS as the main storage. Then we repeat the experiments with SSD-based BB. Finally, we let MLBS helper engine select the storage media at runtime using its internal autotuner, based on the per-node I/O throughput.

Fig. 3 (top) shows per-node **write** throughput of the RTM application on *Shaheen-2*. We observe that the throughput, as seen by the application, is higher when MLBS uses the BB instead of Lustre. We also observe that the performance of **write** using MLBS with Lustre is almost identical to using the BB without MLBS. On the backward phase, as seen in Fig. 3 (bottom), MLBS helper engines succeed in prefetching the snapshots into memory all the times. The RTM in the backward phase shifts truly from IO-bound application to memory-bound (i.e., the kernel speed). Fig. 4 (top) and (bottom) show the aggregate and per-node **write** and **read** throughput of the RTM application on *Summit*. Unlike on *Shaheen-2*, the PFS brings a higher throughput to the application. However, as we increase the number of nodes to 2048, the node-local BB performs better. MLBS autotuner configuration is able to switch, at runtime, for the write I/O operations between the SSD-based and the HDD-based file systems at 2048 nodes. The similar switch happens for the read I/O operations during the backward

Fig. 5. Computation and I/O time breakdown of (a) small grid of size $256 \times 256 \times 64$ where all intermediate snapshot fit in HMB and (b) a larger $512 \times 512 \times 128$ grid where intermediate snapshots are cached and prefetched on HBM, DRAM and PFS

phase. However, the switch for read I/O operations would better benefit if the switch would have happened starting from 64 nodes. The MLBS autotuner eventually reveals that the RTM application should use different hierarchical storage media layers during the forward (i.e., write I/O) and backward (i.e., read I/O) wave propagation. As it is shown in Fig. 4 (bottom), when the number of nodes is 2048, MLBS autotuner read throughput outperforms the reference IBM Spectrum Scale based PFS by 18 times. Since the storage media for the write and the read I/O operations are currently coupled, MLBS autotuner is not able to change the storage media layer at runtime. We believe this is a reasonable extension to support online storage media switch in a future work, which would further maximize the obtained overall performance.

Fig. 5 (a) and (b) show the forward and backward computation times in addition to `write` and `read` times in seconds. In Fig. 5(a), we use a small grid with 350 snapshots that can fit entirely in the HBM while in Fig. 5(b), only 64 snapshots can be held at one instance. Therefore, continuous caching and prefetching has to take place which is translated into 20% of time to solution. MLBS can achieve up to 3X and 2.5X performance speedup compared to the reference PFS-based RTM implementation for the small and large 3D solution grids, respectively. Although this experiment is done on a single node, since RTM is embarrassingly parallel, one can attain similar improvement factors when running on several *Summit* nodes.

7 Summary and Future Work

We have introduced a GPU extension to our Multilayered Buffer System (MLBS) that further create opportunities to overlap expensive I/O operations with GPU computations in the context of the Reverse Time Migration (RTM) for seismic imaging. MLBS leverages the RTM I/O access patterns and asynchronously pushes and pulls solution grids across hardware stacked storage layers, such as

GPU high bandwidth memory/CPU main memory/Burst Buffer/Parallel File Systems. The resulting up and down pipelining of grids between storage layers overlaps with the main RTM computational kernel. It permits the application to carry on as if it does not require out-of-core computations, while achieving up to 2.5X and 1.4X throughput increase, on *Shaheen-2* and *Summit*, respectively. In cases where all intermediate data can fit in a GPU's HBM, which is a realistic case for domain decomposition and strong scaling mode of operations, we show that the computation can proceed as if there is no I/O at all. MLBS prefetching strategy for the read operations makes the RTM application behave as if the data resides in the CPU/GPU main memory, on both systems. MLBS has shown how hierarchical caching of intermediate data can close the gap between I/O and compute for iterative and data intensive applications such as Vector Particle-In-Cell (VPIC), Hardware Accelerated Cosmology Code (HACC) and RTM [6]. We plan to extend MLBS support for topology-aware data motion on heterogeneous systems to further maximize on-node bandwidth [23]. We also intend to further extend our newly proposed MLBS autotuner routine and enable it to switch between memory and storage layers at runtime during both write and read I/O operations.

Acknowledgment. For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia and the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We would like to thank Rached Abdelkhalak from NVIDIA for the insightful discussions and the anonymous reviewers for their constructive comments to improve this paper. This research was partially supported by Saudi Aramco through KAUST OSR contract #3226.

References

1. DDN IME, <https://www.ddn.com/products/ime-flash-native-data-cache/>
2. High-Performance Storage list, <https://www.vi4io.org/>, accessed February 2020
3. HPC IO Benchmark Repository, <https://github.com/hpc/ior>, accessed Dec 2019
4. TOP500 Supercomputer Lists, <https://www.top500.org/lists/top500/>, accessed February 2020
5. AlOnazi, A., Ltaief, H., Keyes, D., Said, I., Thibault, S.: Asynchronous Task-Based Execution of the Reverse Time Migration for the Oil and Gas Industry. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–11. IEEE (2019). <https://doi.org/10.1109/cluster.2019.8891054>
6. Alturkestani, T., Tonellot, T., Ltaief, H., Abdelkhalak, R., Etienne, V., Keyes, D.: MLBS: Transparent Data Caching in Hierarchical Storage for Out-of-Core HPC Applications. In: Proceedings of the 26th International Conference on High Performance Computing (HiPC). pp. 312–322. IEEE (2019). <https://doi.org/10.1109/hipc.2019.00046>
7. Arulraj, J., Perron, M., Pavlo, A.: Write-Behind Logging. Proceedings of the VLDB Endowment **10**(4), 337–348 (2016). <https://doi.org/10.14778/3025111.3025116>

8. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* **23**(2), 187–198 (2011). <https://doi.org/10.1002/cpe.1631>
9. Badam, A., Pai, V.S.: SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. p. 211–224. NSDI’11, USENIX Association, USA (2011)
10. Baysal, E., Kosloff, D.D., Sherwood, J.W.: Reverse Time Migration. *Geophysics* **48**(11), 1514–1524 (1983)
11. Bhimji, W., Bard, D., Romanus, M., Paul, D., Ovsyannikov, A., Friesen, B., Bryson, M., Correa, J., Lockwood, G.K., Tsulaia, V., et al.: Accelerating Science With the NERSC Burst Buffer Early User Program. *Proceedings of the Cray Users’ Group* (2016)
12. Byna, S., Hsu, K.J., Lin, K.W., Shoshani, A., Uselton, A., Wu, K., Chou, J., Rubel, O., Prabhat, Karimabadi, H., Daughter, W.S., Roytershteyn, V., Bethel, E.W., Howison, M.: Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation. In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE (nov 2012). <https://doi.org/10.1109/sc.2012.92>
13. Dong, B., Byna, S., Wu, K., Johansen, H., Johnson, J.N., Keen, N., et al.: Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. pp. 152–161. IEEE (2016). <https://doi.org/10.1109/hipc.2016.026>
14. Dong, B., Wang, T., Tang, H., Koziol, Q., Wu, K., Byna, S.: ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage. In: *2018 IEEE International Conference on Big Data (Big Data)*. pp. 211–220. IEEE (2018). <https://doi.org/10.1109/bigdata.2018.8622616>
15. Henseler, D., Landsteiner, B., Petesch, D., Wright, C., Wright, N.J.: Architecture and Design of Cray Datawarp. *Cray User Group CUG* (2016)
16. Ibtesham, D., Arnold, D., Bridges, P.G., Ferreira, K.B., Brightwell, R.: On the Viability of Compression for Reducing the Overheads of Checkpoint/Restart-Based Fault Tolerance. In: *2012 41st International Conference on Parallel Processing*. pp. 148–157. IEEE (2012). <https://doi.org/10.1109/icpp.2012.45>
17. Kim, S., Kim, H., Emc, D., Lee, J., Jeong, J., March, F., Clara, S., Lee, J., Jeong, J.: Enlightening the I/O Path: A Holistic Approach for Application Performance. *15th USENIX Conference on File and Storage Technologies (FAST 17)* (2017)
18. Kougkas, A., Devarajan, H., Sun, X.H.: Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. pp. 219–230. ACM (2018). <https://doi.org/10.1145/3208040.3208059>
19. Lee, K., Sullivan, M.B., Hari, S.K.S., Tsai, T., Keckler, S.W., Erez, M.: GPU Snapshot: Checkpoint Offloading for GPU-Dense Systems. In: *Proceedings of the ACM International Conference on Supercomputing*. p. 171–183. ICS ’19 (2019). <https://doi.org/10.1145/3330345.3330361>
20. Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., Maltzahn, C.: On the Role of Burst Buffers in Leadership-Class Storage Systems. In: *Proceedings of the 28th Symposium on Mass Storage Systems and Technologies*. pp. 1–11. IEEE (2012). <https://doi.org/10.1109/msst.2012.6232369>
21. Luu, H., Winslett, M., Gropp, W., Ross, R., Carns, P., Harms, K., Prabhat, M., Byna, S., Yao, Y.: A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In: *Proceedings of the 24th International Symposium*

- on High-Performance Parallel and Distributed Computing. pp. 33–44 (2015). <https://doi.org/10.1145/2749246.2749269>
22. Markomanolis, G.S., Hadri, B., Khurram, R., Feki, S.: Scientific Applications Performance Evaluation on Burst Buffer. In: International Conference on High Performance Computing. pp. 701–711. Springer (2017). https://doi.org/10.1007/978-3-319-67630-2_50
23. Martinasso, M., Kwasniewski, G., Alam, S.R., Schulthess, T.C., Hoefer, T.: A PCIe Congestion-Aware Performance Model for Densely Populated Accelerator Servers. In: SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 739–749. IEEE (2016). <https://doi.org/10.1109/sc.2016.62>
24. McCalpin, J.: Memory Bandwidth and Machine Balance in High Performance Computers. Technical Committee on Computer Architecture Newsletter pp. 19–25 (1995)
25. Patrick, C.M., Kandemir, M., Karaköy, M., Son, S.W., Choudhary, A.: Cashing in on Hints for Better Prefetching and Caching in PVFS and MPI-IO. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. pp. 191–202. HPDC ’10, ACM (2010). <https://doi.org/10.1145/1851476.1851499>
26. Sava, P., Hill, S.: Overview and Classification of Wavefield Seismic Imaging Methods. *The Leading Edge* **28**(2), 170–183 (2009). <https://doi.org/10.1190/1.3086052>
27. Scott, D.S.: Parallel I/O and Solving Out of Core Systems of Linear Equations. In: Proceedings of the 1993 DAGS/PC Symposium. pp. 123–130 (1993)
28. Silberstein, M., Ford, B., Keidar, I., Witchel, E.: GPUfs: Integrating a File System with GPUs. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 485–498. ASPLOS ’13, Association for Computing Machinery (2013). <https://doi.org/10.1145/2451116.2451169>
29. Thompson, A., Newburn, C.: GPUDirect Storage: A Direct Path Between Storage and GPU Memory (August 2019), <https://devblogs.nvidia.com/gpudirect-storage/>, accessed May 2020
30. Vazhkudai, S., et. al.: The Design, Deployment, and Evaluation of the CORAL Pre-exascale Systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. pp. 52:1–52:12. IEEE (2018). <https://doi.org/10.1109/SC.2018.00055>
31. Wang, C., Vazhkudai, S.S., Ma, X., Meng, F., Kim, Y., Engelmann, C.: NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines. In: Proceedings of the 26th International Parallel and Distributed Processing Symposium. IEEE (2012). <https://doi.org/10.1109/ipdps.2012.90>
32. Wang, T., Byna, S., Dong, B., Tang, H.: UniviStor: Integrated Hierarchical and Distributed Storage for HPC. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 134–144. IEEE (2018). <https://doi.org/10.1109/cluster.2018.00025>
33. Wang, T., Mohror, K., Moody, A., Sato, K., Yu, W.: An Ephemeral Burst-Buffer File System for Scientific Applications. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE (nov 2016). <https://doi.org/10.1109/sc.2016.68>
34. Wang, T., Oral, S., Wang, Y., Settlemeyer, B., Atchley, S., Yu, W.: Burst-Mem: A High-Performance Burst Buffer System for Scientific Applications. In: 2014 IEEE International Conference on Big Data (Big Data). IEEE (oct 2014). <https://doi.org/10.1109/bigdata.2014.7004215>