2016 IEEE International Conference on Cluster Computing

Realizing Out-of-Core Stencil Computations using Multi-Tier Memory Hierarchy on GPGPU Clusters

Toshio Endo
Global Science Information and Computing Center,
Tokyo Institute of Technology, Japan
Email: *endo@is.titech.ac.jp*

***Abstract*—The memory wall problem is one of major obstacles against the realization of extremely fast and large scale simulations. Stencil computations, which are important kernels for CFD simulations, have been highly successful on GPU clusters in speed, due to high memory bandwidth and computation speed of accelerators. However, their problem scales have been limited by small capacity of GPU device memory. In order to support larger domain sizes than not only device memory capacity but host memory, we extend our approach that combines locality improved stencil computations and a runtime library that harnesses memory hierarchy. This paper describes the extended version of HHRT library that supports multi-tier memory hierarchy, which consists of device memory, host memory and high speed flash SSD devices. And we demonstrate our approach effectively realizes out-of-core execution of stencil computations, whose problem scales are three time larger than host memory capacity.**

I. Introduction

With the existence of many-core accelerators including GPUs and Xeon Phi processors, exascale supercomputers will be realized in a few years to accommodate high performance simulations in weather, medical and disaster measurement area. On the other hand, the scales of those simulations will be limited by the memory wall problem[1]; the improvement of capacity and/or bandwidth of memory is slower than that of processors. It will be a significant obstacle in making larger and finer scale simulations.

We already suffer from this problem especially on clusters with many-core accelerators. In current high-end products, while computation speed and memory bandwidth are high (around 1 to 2 TFlops in double precision and 200 to 400 GB/s per accelerator), memory capacity per accelerator is limited to 6 to 16 GiB. Owing to the advantage in performance of GPUs, many stencil-based applications have been executed successfully on general purpose GPU (GPGPU) clusters, however, the problem sizes have been limited by capacity [2], [3], [4].

In order to realize extremely fast and large scale simulations, we need properly designed approaches to harness deeper memory hierarchy. An example of architecture of a GPGPU computing node is shown in Figure 1. If we harness both of high performance of upper memory layer and large capacity of lower layer, fast and large scale simulations could be realized. In this direction, we have demonstrated an approach of combination of application programs with locality improvement techniques and underlying runtime library to perform data swapping between memory layers[5], [6]. Stencil-based application programs, the whole-city airflow simulation[7] and dendritic solidification simulation[8] have been executed on top of a run-time library called *Hybrid Hierarchical Run-time (HHRT)* in order to enable larger problem scales that surpass the GPU device memory capacity by effectively using capacity of larger host memory. In order to reduce data swapping overhead between memory layers, a locality improvement technique known as *temporal blocking* [9], [10], [11], [12], [13] for stencil computations has been applied.

The objective of this work is to expand the problem scales of such simulations even further, by harnessing recent flash SSD devices. While access bandwidth of traditional SSD devices have been limited by SATA or SAS buses, recent products achieve bandwidth more than 1GB/s. Especially, some m.2 SSD devices achieve this performance at a low price around $500. Figure 1 shows an example of node architecture equipped both with a GPU accelerator and a high performance flash SSD device. Here we observe three-tier memory hierarchy that consists of device memory, host memory and flash SSD. We expect that we can realize application execution with problem scales larger than 100GiB by using this memory hierarchy.

Toward this objective, this paper describes an extension to the HHRT library to support SSD devices. This is done by using SSD devices to accommodate application data when HHRT executes data swapping implicitly. We describe required modification in order to exceed the host memory capacity, including swapping host memory data and treatment of communication buffers. Also we support parallel data swapping on compute nodes with multiple SSD devices. These modifications are performed inside the HHRT library, and we do not need modification to application code.

The relation between performance and problem scales of a stencil benchmark is evaluated on two GPGPU computing environments equipped with SSD devices. One is a PC server with a high performance m.2 SSD shown in Figure 1. The other is a cluster called TSUBAME-KFC/DL[14], where each node has Tesla K80 GPUs and SATA SSD devices. We show that our approach enables out-of-core execution with problem scales larger than host memory capacity; we successfully execute the stencil benchmark with 192GiB problem size, three times larger than host memory. Also we analyze the effects of access speed of SSDs on performance, and relation between performance and problem sizes in detail. Our approach realizes

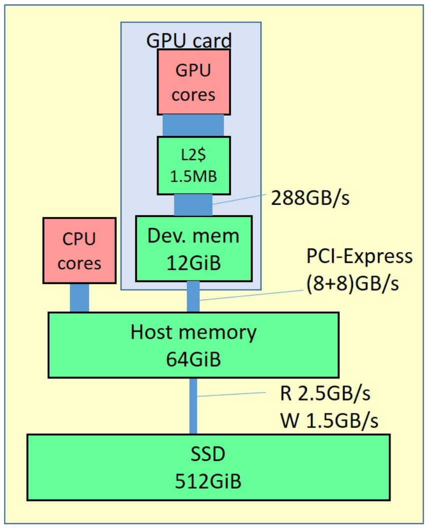


Fig. 1. Memory hierarchy of a GPGPU machine from the viewpoint of GPU cores. Here an SSD with bandwidth of >1GB/s is equipped. This figure illustrates a PC server used for performance evaluation in Section V.

good scalability, which paves the road toward extremely fast and large scale simulations in coming exascale era.

II. HHRT: Hybrid Hierarchical Runtime

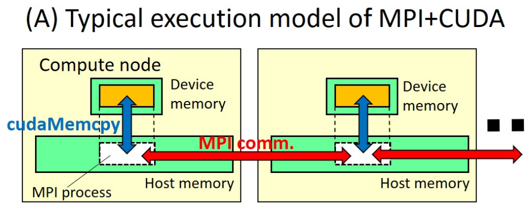
A. Concept of HHRT

The objective of Hybrid Hierarchical Runtime (HHRT) library is *to extend applications' supportable problem scales*[5]. The main targets of HHRT are applications whose problem scales have been limited by the capacity of upper memory layer, such as GPU device memory in GPU clusters. For instance, such applications include simulation software based on stencil computations written for GPUs[2], [3], [4], [8]. They have been enjoyed high computing speed and memory bandwidth of GPUs, however, most of them are designed as "in-core", or supported problem sizes are determined by device memory capacity as shown in Figure 2 (A), in spite of the existence of larger memory (storage) layers, including host memory and file systems. While the problem sizes are expanded by using multiply GPUs and compute nodes, they are still limited by the aggregated amount of used device memory capacity.

The problem scales of such applications are expanded by executing them with only slight modifications on top of HHRT library. Basically we assume that the target applications of HHRT have the following characteristics:

- The applications consist of multiple processes working cooperatively.
- Data structure that is frequently accessed by each process is put on upper memory layer.

Many stencil applications on GPU clusters described above have already these characteristics, since they are written in MPI to support multiple nodes, and regions to be simulated are distributed among processes so that each process has smaller local region than device memory capacity.



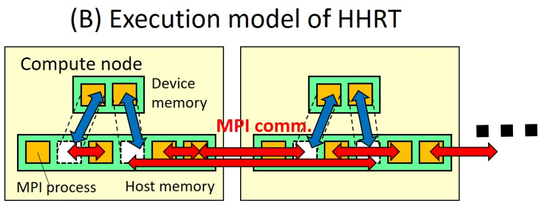


Fig. 2. Execution model on typical MPI/CUDA and execution model on HHRT library.

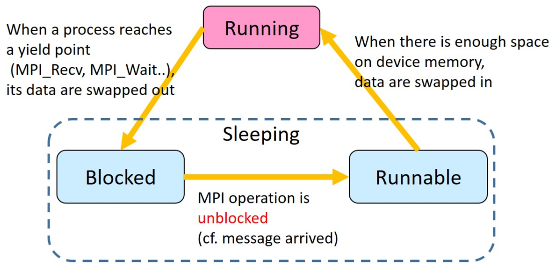

Running
When a process reaches a yield point (MPI_Recv, MPI_Wait..), its data are swapped out
When there is enough space on device memory, data are swapped in
Sleeping
Blocked
Runnable
MPI operation is unblocked (cf. message arrived)

Fig. 3. State transition of each process on HHRT.

In the following discussion, we focus on the current HHRT library, which targets applications written in MPI for inter-process communication and CUDA for using GPUs. Now HHRT library is implemented as a wrapper library of CUDA and MPI, thus it has the same APIs as CUDA and MPI, except some additional APIs for performance improvement[5].

On the execution model of HHRT, each GPU is shared by multiple MPI processes as illustrated in Figure 2 (B). This is contrary to the typical execution method shown in Figure 2 (A).

When users execute their application on top of HHRT, they would typically adjust the number of MPI processes so that *the data size per each process is smaller than the capacity of device memory*. Hereafter P_s denotes the number of processes sharing a single GPU, which is 6 in the figure. By invoking plenty number of processes per GPU, we can support larger problem sizes than device memory in total.

This *oversubscribing* model itself, however, does not support larger problem sizes. We cannot hold all the data of P_s processes on the device memory at once, when P_s is large enough. Instead, we execute swapping out of memory regions of some processes from the device memory (process-wise swapping).

On HHRT, swapping is tightly coupled with process scheduling. Swapping out may occur at *yield points*, where process may start sleeping, instead of individual memory accesses. In current library, based on CUDA and MPI, yield points correspond to blocking operations of MPI, such as `MPI_Recv`, `MPI_Wait`, and so on.

Figure 3 illustrates state transition of each MPI process. Each process is in one of states, "running", "blocked" or "runnable" [1] .

When a running process p reaches a yield point, it starts swapping out contents of all the regions that the process holds on the device memory into some dedicated place (called swap buffer hereafter) on the lower memory layer, such as host memory. Then the process releases the capacity of device memory so that it can reused by other processes, and the process starts sleeping. While the MPI operation that have let the process p start to sleep is still blocked, the process p remains in "blocked" state. Even when the operation is unblocked (for example, a message has arrived), the process p may not start running immediately if the capacity of device memory is insufficient; here p is in the "runnable" state. Afterwards, when the size of free space in device memory becomes sufficient, the sleeping process p can start swapping-in; it allocates the heap region again on device memory, copying user data from swap buffer to the heap on device memory. Then p can exit from the yield point, which is an MPI blocked operation function. Now p is in the "running" state.

With this swapping mechanism, P_s processes share the limited capacity of device memory in a transparent fashion from application programs.

Generally, we can obtain better performance if more than one process out of P_s processes can be in "running" state, since such a situation enables overlapping of swapping processes and running processes. This can be done by configuring the data size of each process to be less than half of device memory capacity. Figure 2 shows such a situation where two processes are running and other four processes are sleeping.

B. The Base Implementation of HHRT

Here the base implementation of HHRT [5] is briefly described. As described before, this implementation is for application programs written with CUDA and MPI. To support them, the HHRT library is implemented as a wrapper library of CUDA and MPI. Hereafter we distinct APIs provided by HHRT library called by application processes directly, from those of the underlying CUDA or MPI by prefix. For example, `HHcudaMalloc` and `HHMPI_Recv` are provided by HHRT, and `cudaMalloc` and `MPI_Recv` come from the underlying CUDA or MPI.

The version described in this section takes memory hierarchy of device memory and host memory into account, and the capacity of device memory is time-shared among processes. This will be extended to support even larger problem scales than host memory in Section III.

[1] In the actual implementation, there are two transient states, "swapping-in" and "swapping-out".

In order to realize the concept of HHRT, we have implemented the following mechanisms.

1) Memory Management: When a process's data is swapped out, all the memory regions that reside on the device memory are copied into swap buffer on the lower layer. For this purpose, those regions allocated by `HHcudaMalloc` and its variants have to be tracked by HHRT library. We have implemented memory management functions in the HHRT library. In the initialization phase, HHRT allocates a large heap region by using the underlying CUDA. When `HHcudaMalloc` is called by application processes, it finds a free region from the heap region, and returns it to the caller.

2) Heap Management: The heap region of each process has to be managed with the following conditions.

- After the heap region is released in swapping-out, the reclaimed capacity can be reused by another process.
- After the heap region is allocated again in swapping-in, the address of the heap has to be same as previous one. This condition is required for the transparent execution of application processes.

For these conditions, we take the following approach. In the initialization phase, one of P_s processes (for example, the process that has the least MPI rank) becomes the representative, and it allocates the heap region by (underlying) `cudaMalloc`. Then P_s processes share the region by using CUDA IPC mechanism. The representative exports the region by `cudaIpcGetMemHandle` and the other processes obtain the address of the region by `cudaIpcOpenMemHandle` API.

After preparing shared memory on device memory, HHRT keeps track of the current user process of the region. Information of those management information is located on (host) shared memory between processes. When a process releases its heap in swapping-out, it updates the heap user variable on the shared memory by -1 (that means no-user). Other sleeping (runnable) processes periodically checks the variable, and when it finds heap user is -1, it atomically updates the variable to its rank. By using this mechanism, the device memory capacity is time-shared by processes.

III. Extension to HHRT for Out-of-Core Computations

This section describes extension to HHRT library in order to support larger application data than the capacity of host memory. The basic idea is simple; we use files on file systems as swap buffer in addition to host memory. If those files are put on fast disk devices such as m.2 flash SSDs, the cost of swapping is expected to be moderate.

The extensions are done inside the HHRT library, enabling expansion of supportable data size without modifying application code.

A. Swapping Data on Host Memory

In order to exceed the host memory capacity, we should consider swapping out *data allocated on host memory*, in addition to data on device memory for the following reason. When an application process allocates considerable amount of

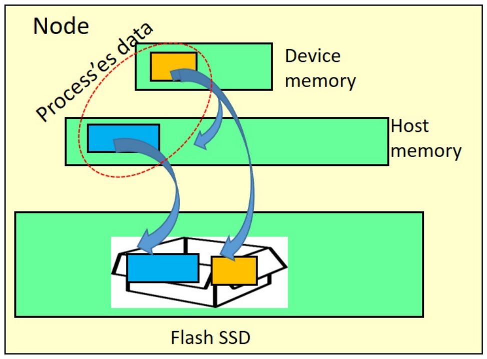


Fig. 4. Usage of three tiers memory hierarchy. Application data both on device memory and host memory are swapped out to lower layer.

data on host memory, the number of processes on a compute node is limited and the total application data size is limited, without swapping those data out elsewhere. For our main target applications including stencil computations, the data structures mainly used for computation tend to be placed on device memory rather than host memory. However, they may still consume host memory for data I/O, data initialization, or MPI communication. In order to enlarge data sizes in such cases, data both on device memory and on host memory should be swapped out to lower memory layers as shown in Figure 4.

Our heap management for host memory is implemented in similar ways to that for device memory described in Section II-B. In addition to `cudaMalloc`, wrapper functions for `malloc`, `calloc`, `free` etc., from libc are implemented.

The heap management should be done while satisfying the following conditions:

- After the heap region is released in swapping-out, the reclaimed capacity can be reused by another process.
- After the heap region is allocated again in swapping-in, the address of the heap has to be same as previous one.

For this purpose, we chose to use `mmap`/`munmap` systemcalls for heap allocation. In swapping-in phase, each process calls `mmap` to allocate a region for heap on host memory. In order to preserve the address of the heap, we specify the start address explicitly. After each process in swapping-out phase calls `munmap`, another process can reuse the capacity.

B. *Using Files as Swap Buffers*

In swapping out, all data on heaps are evacuated to swap buffers created on lower memory layer. For this purpose, each process internally creates two files as swap buffers, each of which are for device memory heap and host memory heap, respectively. When a process opens these files, we use `O_DIRECT` flag in order to suppress the effects of file caching, which consumes host memory. In the swapping-in/out operations, HHRT library read/write systemcalls for file access.

If the compute node has several SSDs, each process on the node can be configured to use one of them in a round-

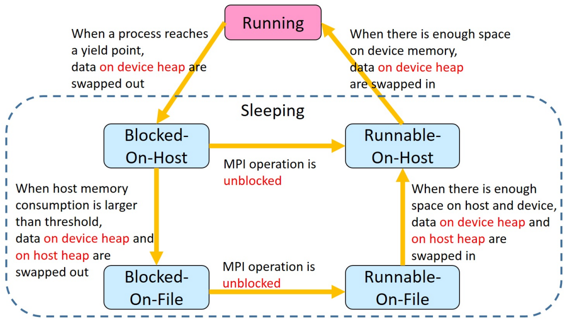

Running
When a process reaches a yield point, data on device heap are swapped out
When there is enough space on device memory, data on device heap are swapped in
Sleeping
Blocked-On-Host
Runnable-On-Host
MPI operation is unblocked
When host memory consumption is larger than threshold, data on device heap and on host heap are swapped out
When there is enough space on host and device, data on device heap and on host heap are swapped in
Blocked-On-File
Runnable-On-File
MPI operation is unblocked

Fig. 5. State transition of each process on the extended HHRT. It includes states in which data are on files.

robin style, to enable parallel swapping to improve HHRT applications performance.

C. *Behavior of Processes*

With mechanisms explained so far, application data both on device memory and host memory can be swapped out to lower memory layer. This section describes how we schedule swapping to keep application performance higher. Here we assume that transfer speed between host memory and files are lower than that between device and host. As shown in Figure1, it is the case even if we use the latest m.2 SSD, whose access bandwidth is around 75% lower than PCIe.

With this assumption, our strategies are:

- To evacuate data from device memory as fast as possible, since we regard device memory as the most precious resource.
- To reduce the frequency of swapping that involve access to files.

Based on this discussion, the behavior of each process described in Section II-A is modified as in Figure 5. Here ”Blocked” state is extended to ”Blocked-On-Host” and ”Blocked-On-File”, and ”Runnable” state is treated similarly. When a running process p reaches a yield point, it first evacuates data on device memory into host memory, then it is in Blocked-On-Host state. Now the capacity of device memory can be reused by other processes. Afterward, if p finds the host memory consumption is high, it evacuates data both on device memory and host memory into files, which work as swap buffers, and moves to Blocked-On-File state. On the other hand, if the MPI operation is unblocked before the above transition occurs, the process p moves to Runnable-On-Host state directly, without involving file access.

D. *Treatment for Communication Buffers*

We have described our basic implementation of our extension to HHRT. However, we noticed another issue occurs when HHRT simply releases regions on heap memory in swapping-out phases. Let us consider a process calls `MPI_Recv` specifying a heap pointer (hereafter $p1$) as a communication buffer. Since `MPI_Recv` is a blocking operation, HHRT may swap all the data out during `MPI_Recv` is ongoing. Here if $p1$ is included in the host heap, whose memory region is discarded after swapping out, it would break the behavior of the underlying MPI since `MPI_Recv` tries to write received data to $p1$. Thus we need special treatments to exclude these buffers from the targets of swapping out conceptually.

Our current solution is to make additional buffers on host memory to duplicate data for communication. For example, `HHMPI_Recv` in HHRT makes the duplication before calling the underlying `MPI_Recv`. Instead of $p1$, we pass a pointer of the duplicated buffer ($p1'$) as a parameter of `MPI_Recv`. Similarly, such duplications are done in other communication operations, including non-blocking ones or collective ones. These duplicated buffers (such as one pointed by $p1'$) are kept on host memory until the corresponding MPI communication finishes. After that they are released.

E. Current Limitations

Our intention in designing and implementing HHRT is to support generic parallel applications written in CUDA and MPI. However, there are still limitations and assumptions on the applications. Elimination and relaxation of these limitations are included by our future work.

- Each process can use up to one GPU.
- Each process has to be single-threaded.
- Only MPI-1 APIs are supported. Especially one-side communication APIs in MPI-2 are not supported.
- The current version does not support some CUDA features including unified virtual memory, texture cache, Hyper-Q, etc.
- Global variables (on device or on host) are not targets of swapping, thus they consume memory capacity and may limit the total problem scale.
- Basically source files have to be re-compiled with `#include "hhrt.h"` before linking. If some memory regions are allocated by other parts, such as third-party libraries, those regions are not targets of swapping.
- C++ `new` is not supported yet.
- `malloc` invocations inside CUDA kernel functions are not considered.

IV. A Stencil Benchmark for Evaluation

We evaluate performance of out-of-core execution with the extended HHRT library by executing a "seven-point" stencil benchmark on top of it. This is the same one evaluated in our previous publication[5]; without modifying the application code, we can expand the stencil size larger than the host memory capacity. This section briefly describes the benchmark and *temporal blocking*, which is a locality improvement technique.

Stencil computations are commonly found kernels in CFD and engineering simulations. The target area to be simulated is expressed as a regular grid, and all grid points are computed in each time step. The new value of each grid point is calculated based on values of its adjacent points in the previous time step.

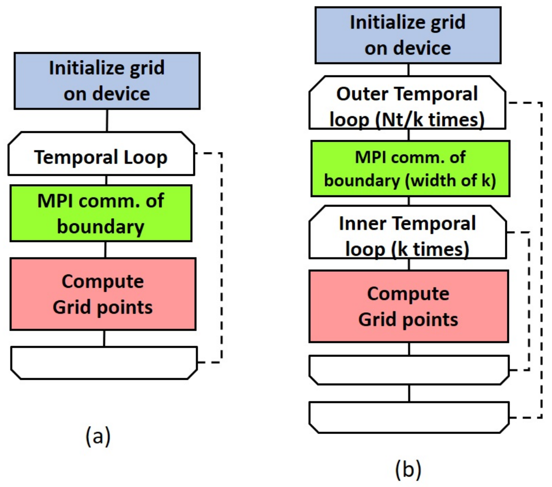

Initialize grid on device
Temporal Loop
MPI comm. of boundary
Compute Grid points
(a)
Initialize grid on device
Outer Temporal loop (Nt/k times)
MPI comm. of boundary (width of k)
Inner Temporal loop (k times)
Compute Grid points
(b)

Fig. 6. Behavior of each process in stencil computations on GPU clusters. (a) without temporal blocking, (b) with temporal blocking.

Our benchmark is designed for GPU clusters by using CUDA and MPI, and like typical stencil applications, it distributes the grid to be simulated among MPI processes. And each local grid per process is usually smaller than device memory capacity. The behavior of the first version (without temporal blocking) is shown in Figure 6 (a). Before computation in each time step, we need MPI communication of process boundary region (called *halo*) between adjacent processes, due to data dependency between adjacent grid points.

Such stencil implementations are known to have poor locality. On top of HHRT, this issue introduces very frequent swapping since each time step involves blocked MPI operations for boundary exchange. As a result, performance gets significant low with larger problem sizes than device memory capacity.

In order to improve locality of stencil computations, we use a known technique called *temporal blocking*[9], [10], [11], [12], [13]. We have shown this technique can be implemented easier on top of HHRT than totally hand-coded cases[5], [6].

Figure 6 (b) shows the process behavior in this approach. Here a parameter k denotes the *temporal blocking size*. When we start the computation for a relatively small sub-grid, we continuously perform the computation for it for k steps without interruption. In the algorithm in Figure 6 (b), the frequency of MPI operations, which work as yield points on HHRT, is reduced to $1/k$. Instead, the cost of each boundary communication becomes k times larger in order to prepare sufficient boundary data for local k steps computation.

Also this implementation of temporal blocking introduces redundant computation costs for extra boundary area, which gets heavier with larger k. Techniques such as wavefront blocking and diamond blocking[15], [16] have been proposed to eliminate such redundant computation; applying them on top of HHRT is included in our future work.

TABLE I. PLATFORMS USED FOR EVALUATION

	PC server	TSUBAME-KFC/DL
# of nodes	1	40
GPU	NVIDIA Tesla K40	NVIDIA Tesla K80
SP peak perf. (GFlops)	4.29 (5.0 w/ boost)	2.8 (4.37 w/ boost)
Device memory BW (GB/s)	288	240
Device memory size (GiB)	12	12
# of GPUs/node	1	8 (4 boards)
CPU	Intel Core i7-6700K	Intel Xeon E5-2620 v2
# of CPUs/node	1	2
CPU-GPU connection	PCIe gen3 x8	PCIe gen3 x16
Peak BW (GB/s)	8+8	16+16
Host memory size (GiB)	64	64
SSD	Samsung 950PRO m.2	Intel DC S3500
Read BW (GB/s)	2.5	0.50
Write BW (GB/s)	1.5	0.41
Size (GB)	512	480
# of SSDs/node	1	2
Network Interface	Gigabit Ethernet	4x FDR InfiniBand
OS	CentOS 7.2	CentOS 6.4
MPI	MPICH 3.2	MVAPICH2 2.2a
CUDA	7.5	7.0

V. PERFORMANCE EVALUATION

A. Evaluation Conditions

Our performance evaluation has been conducted on two GPGPU platforms shown in Table I. One is a PC server equipped both with a NVIDIA K40 GPU and a m.2 SSD whose bandwidth is 2.5GB/s (Read), 1.5GB/s (Write). Its memory hierarchy has been shown in Figure 1. Since the device memory capacity is 12GiB and the host memory capacity is 64GiB, the host swapper is necessary for problem sizes $\geq$12GiB, and so is the file swapper for $\geq$64GiB problems.

The other platform is a 40-node GPGPU cluster named *TSUBAME-KFC/DL* [14]. Each node has four NVIDIA K80 gemini boards, thus eight GPUs are available [2]. In this paper, a single GPU per node is used to evaluate the effect of memory hierarchy. The sizes of device memory and host memory are same as those of the PC server described above. Note that the application performance is significantly affected by SSD access performance in "out-of-core" cases. Each TSUBAME-KFC/DL node is equipped with two SATA SSDs. Their bandwidth is severely lower than the m.2 SSD described above; 0.5GB/s for read and 0.41GB/s for write per SSD. We use two SSDs per node for parallel file swapping as described in III-B.

As the benchmark program, we used a seven-point stencil program with temporal blocking described in Section IV linked with our HHRT library. The computed grids are three-dimensional arrays, whose elements have float data type. The 3D grid is decomposed in two-dimension among MPI processes. In the following discussion, we evaluate the performance for various problem sizes. The problem size is given as the aggregated sizes of 3D arrays that should be computed, excluding extra halo region introduced for temporal blocking. Also we do not count the redundant computations to obtain the performance number in GFlops, while the redundant computations increase the execution time. These assumptions are made for fairness in comparison during various temporal blocking size k.

[2]When [14] is published, the node had four K20X GPUs. They have been upgraded later

TABLE II. THE NUMBER OF MPI PROCESSES THAT SHARE A SINGLE GPU FOR EACH PROBLEM SIZE. THE NUMBERS IN PARENTHESIS ILLUSTRATE PROCESS GRID SIZES FOR 2D DECOMPOSITION.

Problem size	P_s
6(GiB)	1 (=1×1)
8	1 (=1×1)
12	4 (=1×4)
16	4 (=1×4)
24	6 (=2×3)
32	12 (=3×4)
48	16 (=4×4)
64	24 (=4×6)
96	32 (=4×8)
128	48 (=6×8)
192	80 (=8×10)

B. Evaluation on a Single GPU

The graphs in Figure 7 show the performance of the stencil program on a single GPU on the PC server and a TSUBAME-KFC/DL node. "TB" in graphs corresponds to the cases with temporal blocking, and "NoTB" does not use temporal blocking (thus $k = 1$). In TB cases, we show the fastest cases with varying k. Also P_s, the number of processes to be oversubscribed is configured for each problem size so that each process's data on GPU is sufficiently smaller than device memory capacity; the numbers are shown in Table II.

On both machines, we have successfully executed the program with problem sizes of up to 192GiB, which is three times larger than host memory capacity, and 16 times larger than device memory. This is owing to HHRT's facility for oversubscription and swapping.

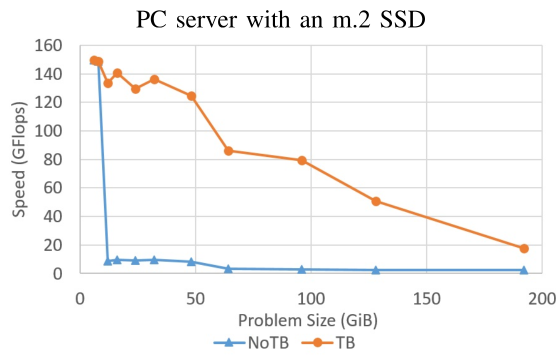
While NoTB is impractically slow with 12GiB problem sizes or larger for too heavy swapping cost, the situation is significantly better on TB for better locality. Comparing the two graphs, we observe differences between the two machines in TB performance when swapping occurs. When only host swapper is required (12 to 48GiB), the slow down compared with "in-(GPU)core" case is 6 to 17% on the PC server, and 19 to 46% on KFC/DL node. This result looks counterintuitive, since the latter machine provides faster PCIe communication. Through preliminary experiments, we observed that this difference is mainly due to host-to-host memory transfer speed. On a PC server with a newer CPU gave faster memory copy speed.

In the context of this paper, the performance difference with file swapper is more important. In the case of 96GiB problem, the slow down compared with 6GiB case is 47% on the PC server and 75% (TB (2SSDs))on the KFC/DL node. This difference directly comes from the bandwidth of SSDs. Even when two SSDs on KFC/DL are used in parallel, the aggregated bandwidth is still about the half of that of the m.2 SSD. When only one SSD is used (TB (1SSD)), the performance is roughly halved.

With larger problem sizes (128 or 192GiB), the speeds are degraded on both machines; this is analyzed in the below section.

C. Discussion on Performance Limiting Factor

Considering the characteristics of our benchmark with temporal blocking on top of HHRT, we have to carefully



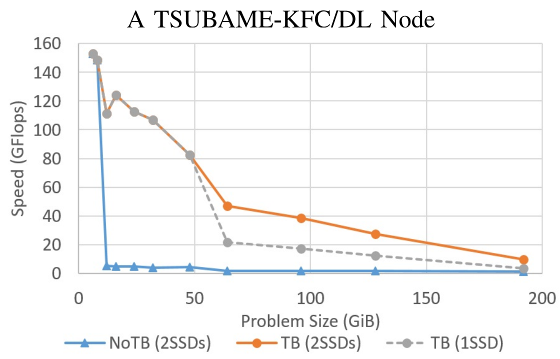


Fig. 7. Performance evaluation with various problem sizes on a single GPU.

choose the temporal block size k. If k is too small, the performance suffers from large swapping costs due to more frequent "yield" points. If k is too large, the computational cost increases for redundant computation.

There are additional issues that limit both the performance and problem scale; memory consumption of user processes inflates as k gets larger for the following reasons:

1) Each MPI process allocates its local grids surrounded by halo regions whose width is k. Thus device memory consumption is increased.
2) Each MPI process allocates MPI buffers for halo exchange on host memory. As described in SectionIII-D, their copies occupy host memory even in "On-File" states. Their sizes are inflated as k is larger.

Table III shows the performance of the stencil benchmark with various k and various problem sizes on two machines. We discuss the effects of k according to the following three categories.

- **Smaller than device memory (6 to 8GiB):** The speed performances are best when k=1 and degraded as k gets larger, since the amount of redundant computation increases. There is no swapping.
- **Smaller than host memory (12 to 48GiB):** The performances with k=1 are less than 10GFlops due to heavy swapping costs. They are improved with larger k, since the frequency of swapping is $O(1/k)$. After hitting "suit spots", the performance is degraded by redundant computation costs. The best k is around 32 on the PC server and around 64 on a KFC/DL node. The exception is the case of 48GiB on KFC/DL node. Here due to the issue 1) above, the memory consumption in increased with k and there are swaps to files if $k \geq 48$ (the performance numbers in the table are in italic font). Thus the performances in those cases are degraded, which makes $k = 32$ the peak as a result.
- **Larger than host memory (64 to 192GiB):** The performances with k=1 are even lower for larger costs for file swapping. They are improved with larger k, however, the improvement is slower than the above case. With further larger k, the execution fails, as shown as "OOM" in the table. This is mainly due to the issue 2), which causes inflation of host memory consumption even if almost all the processes are at "On-File" states. This issue is severer with larger problem sizes; if it is 192GiB, the executions with $k \geq 16$ fail. This explains the performance degradation we observed in Figure 7. When problem sizes are larger, the upper limit of k that enables successful executions is decreased, which also limits the highest performance.

Finally, in addition to the above two issues, we observed the following issue that also limits our problem scales:

3) We observed that when a GPU is used by multiple processes, considerable amount of device memory is consumed besides explicitly allocated region. In our platforms, the amount of such implicit consumption is around 75MiB per process. We consider they are buffers used for internal management by CUDA.

This memory pressure gets more significant with larger P_s (the number of processes that share a GPU). For example, if $P_s = 80$ as in the case with 192GiB problem, $75 \times 80 = 6000$ MiB, which is around 50% of device memory, is occupied implicitly, which is not available to user programs. The basic idea to support larger problem scales on HHRT is to increase the number of oversubscribed processes P_s. However, we cannot use larger P_s than 160 on our platforms, since the aggregated internal buffers use up the device memory. As a result, the supportable problem sizes are limited.

We expected that this problem would be alleviated by using CUDA MPS (multi process servers), which works as a multiplexer of CUDA device. Unfortunately, current MPS only supports up to 16 client processes, thus we abandoned it.

D. Evaluation on Multiple Nodes

This section evaluates the performance by using multiple nodes. Here we used 32 TSUBAME-KFC/DL compute nodes at maximum. Like the previous sections, we used one GPU per node.

Figure 8 demonstrates weak scalability. The graph shows the case of 24GiB problem size per node, and 96GiB per node. The temporal block size k is 32 in the former case, and it is 12 in the latter case. It is smaller than in the single GPU evaluations; since the effects of issue 2) described in

TABLE III. EFFECTS OF TEMPORAL BLOCK SIZE k ON PERFORMANCE FOR VARYING PROBLEM SIZES.

PC server with an m.2 SSD

	k=1	8	16	24	32	48	64	96
6(GiB)	149	148	145	142	137	134	129	119
8	149	147	145	142	139	133	129	121
12	8.72	65.7	101	130	134	132	126	114
16	9.39	63.2	108	138	141	135	130	106
24	9.37	63.3	98.8	122	125	130	122	110
32	9.79	58.3	89.5	121	136	127	121	98 .3
48	8.12	61.7	88.7	116	125	*72.7*	*87.9*	*91.5*
64	*3.23*	*22.3*	*34.4*	*49.0*	*57.7*	*85.9*	*82.6*	*75.5*
96	*2.68*	*20.7*	*33.4*	*47.7*	*53.4*	*79.3*	OOM	OOM
128	*2.67*	*18.8*	*38.4*	*45.4*	*50.6*	OOM	OOM	OOM
192	*2.55*	*17.7*	OOM	OOM	OOM	OOM	OOM	OOM

A TSUBAME-KFC/DL Node (2 SSDs are used)

	k=1	8	16	24	32	48	64	96
6(GiB)	153	145	145	144	137	134	130	121
8	149	148	146	144	142	136	133	126
12	5.23	39.1	65.1	84.0	94.3	103	111	95.3
16	4.54	37.4	63.2	91.0	88.9	116	124	101
24	4.75	35.3	56.5	77.2	97.6	106	112	107
32	3.91	31.6	53.8	65.9	90.5	92.9	107	*51.1*
48	4.37	30.7	53.4	66.8	82.3	*40.8*	*43.5*	*47.4*
64	*1.54*	*10.9*	*18.1*	*25.2*	*32.7*	*40.8*	*46.9*	OOM
96	*1.51*	*9.79*	*18.5*	*23.7*	*29.2*	*38.4*	OOM	OOM
128	*1.37*	*9.79*	*17.9*	*22.6*	*27.3*	OOM	OOM	OOM
192	*1.35*	*9.67*	OOM	OOM	OOM	OOM	OOM	OOM

- Each number represents the speed in GFlops.
- The underlined number shows the fastest case in each row.
- The *italic* numbers correspond to cases when the file swapper is used.
- ”OOM” means execution failure due to ”Out Of Memory”.

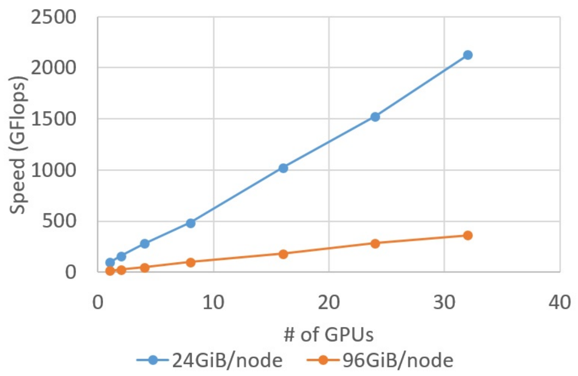


Fig. 8. Weak scability evaluation on TSUBAME-KFC/DL.

Section V-C are heavier in multiple node evaluations, we took the above numbers.

We observe the scalability is pretty good in both cases. Compared with a single GPU performance (97.6GFlops in 24GiB case and 15.2GFlops in 96GiB case), we got 21.8 times and 23.9 times speed up on 32 GPUs. In the latter case, we have successfully executed the problem of $96 \times 32 = 3072$ GiB size by using limited computing resources.

Currently, the resultant speed 363GFlops is not so high due to the insufficient speed of SATA SSDs. If each node were equipped with fast m.2 SSDs as the PC server we used, we could obtain around 1TFlops for 3TiB problem. It is possible to constructing such clusters in low price, since the cost of an m.2 SSD we used is around \$400, while we require mother boards that support m.2.

VI. RELATED WORK

Non volatile memory devices, especially NAND flash memory devices, have been widely attracted attention since they fill the performance/capacity gap between traditional DRAM and hard disks. There are lots of software projects and products that harness flash; some use them as accelerators of hard disks, such as DDN's Infinite Memory Engine [3]. On the other hand, this work uses flash devices in order to expand available capacity of host memory and GPU device memory. However, we do not use the standard swapping mechanism of OS, but process-wise swapping mechanism of HHRT for performance.

The HHRT model is largely inspired by Adaptive MPI (AMPI) [17], implemented on top of CHARM++ [18]. Both AMPI and HHRT provide execution model where several MPI processes share the limited resources on computer systems. On the other hand, AMPI itself does not support swapping between memory hierarchy.

Our HHRT implementation has lots of common ideas with the virtual memory based runtime by Becchi et al[19]. Their system provides time-sharing facility of GPU device memory capacity by data swapping. Their main focus is to maintain multiple distinct GPU applications on limited number of GPUs, while our focus is to support MPI parallel applications that consist of multiple processes, which have inter-process dependency. For this difference, HHRT has features that Becchi's system does not have: yielding mechanism coupled with MPI communication operations and treatment for communication buffers.

From the applications' viewpoint, temporal blocking for stencil computations have a long history and have been implemented in various computer architectures [9], [10], [11]. While most previous papers have focused on improving cache hit ratio, Mattes et al. and we have previously demonstrated the effects of (completely hand written) temporal blocking in order to reduce data transfer costs between device memory and host memory[12], [13]. While based on these results, our objective is to support multi-tier memory hierarchy, GPU device memory, host memory and flash devices, while reducing swapping costs.

Midorikawa et al.[20] implemented and evaluated out-of-core stencil computations that consider host memory and flash devices. They have successfully obtained performance gain with temporal blocking. Our work differs from it since we take GPU device memory into account, and thus in-core performance used as ”baseline” is much higher on GPUs than on CPUs.

Generally, programming locality improvement techniques tends to be troublesome for end programmers; most of the above projects have been involved with this issue. One of approaches to relieve it is to use domain specific frameworks, such as Physis [21] or Shimokawabe's framework[22]. With this framework, programmers write stencil applications in for-all style. After programmers write a code fragment for a single grid point update, without considering details of underlying architectures. This approach is highly promising since it would be possible to implement optimization techniques including

[3]http://www.ddn.com/products/infinite-memory-engine-ime14k/

temporal blocking in a user-transparent style. On the other hand, if users already have application codes written in MPI and CUDA[2], [3], [4], [8], they have to rewrite the code for the framework. Our approach with HHRT allows users to use such existing code as a start point, and to improve it step by step.

Recently NVIDA has announced upgraded version of *Unified Memory* mechanism that comes with CUDA version 8 [4]. With this mechanism, programmers can allocate memory regions whose contents are moved between device memory and host memory transparently. Also those regions may be larger than device memory capacity. This has a similar effect as the swapping mechanism of HHRT. However, Unified Memory does not support arbitration between processes sharing a GPU. Also currently it does not support the extension to flash devices, which HHRT supports.

VII. Conclusion

This paper has demonstrated an extension to the HHRT library, a data swapping library between deeper memory hierarchy on GPU clusters, in order to support the hierarchy of flash devices. On top of this extended library, we successfully executed extremely large scale stencil computations, by integrating an algorithm level locality improvement, temporal blocking.

Through the performance evaluation on two computing environments equipped with GPUs and flash devices, we can support problem size 3 times larger than host memory and 16 times larger than GPU device memory. The performance overhead for such larger sizes largely depends on access performance of flash devices. With the latest m.2 SSDs with bandwidth larger than 1GB/s, we observed the overhead for the problem size of 96GiB is 47%, showing that the out-of-core execution involving flash devices is realistic.

Also we have demonstrated the effects of parallel swapping using multiple flash devices and high scalability using multiple compute nodes. In spite of the above results, there are still rooms to improve problem scales and performance. Through the detailed measurements, we discussed several scale limiting factors mainly related to host memory consumption.

Future work includes performance evaluation of real simulation workloads as done in [6], and improvement of the HHRT implementation both in scale and performance, by resolving current limiting factors, toward extremely high performance and large simulations on top of next generational Exascale supercomputers.

Acknowledgements

This research is funded by JST-CREST, "Software Technology that Deals with Deeper Memory Hierarchy in Post-petascale Era".

References

[1] R. Lucas et al.: Top Ten Exascale Research Challenges, DOE ASCAC Subcommittee Report (2014).

[2] E. H. Phillips, M. Fatica: Implementing the Himeno Benchmark with CUDA on GPU Clusters, IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1-10 (2010).

[3] D. A. Jacobsen, J. C. Thibault, I. Senocak: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters, 48th AIAA Aerospace Sciences Meeting, 16pages (2010).

[4] M. Bernaschi, M. Bisson, T. Endo, M. Fatica, S. Matsuoka, S. Melchionna, S. Succi: Petaflop Biofluidics Simulations On A Two Million-Core System, IEEE/ACM SC'11, 12 pages (2011).

[5] T. Endo, G. Jin: Software Technologies Coping with Memory Hierarchy of GPGPU Clusters for Stencil Computations. IEEE Cluster Computing (CLUSTER2014), pp. 132-139 (2014).

[6] T. Endo, Y. Takasaki, S. Matsuoka: Realizing Extremely Large-Scale Stencil Applications on GPU Supercomputers. IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015), pp. 625-632 (2015).

[7] N. Onodera, T. Aoki, T. Shimokawabe, T. Miyashita, H. Kobayashi: Large-Eddy Simulation of Fluid-Structure Interaction using Lattice Boltzmann Method on Multi-GPU Clusters, 5th Asia Pacific Congress on Computational Mechanics and 4th International Symposium on Computational Mechanics (2013).

[8] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, S. Matsuoka: Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer, IEEE/ACM SC'11, 11 pages (2011).

[9] M. E. Wolf and M. S. Lam: A Data Locality Optimizing Algorithm. ACM PLDI 91, pp. 30–44 (1991).

[10] M. Wittmann, G. Hager, and G. Wellein: Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7pages (2010).

[11] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. IEEE/ACM SC'10, 13 pages (2010).

[12] L. Mattes, S. Kofuji: Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids. International Conference on Microwave and Millimeter Wave Technology (ICMMT), pp.1536–1539 (2010).

[13] G. Jin, T. Endo, S. Matsuoka: A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPUs. IEEE Cluster Computing (CLUSTER2013), 8 pages, (2013).

[14] T. Endo, A. Nukada, S. Matsuoka: TSUBAME-KFC: a Modern Liquid Submersion Cooling Prototype towards Exascale Becoming the Greenest Supercomputer in the World. IEEE International Conference on Parallel and Distributed Systems (ICPADS 2014), pp. 360-367 (2014).

[15] V. Bandishti, I. Pananilath, U. Bondhugula: Tiling stencil computations to maximize parallelism. IEEE/ACM SC'12, 11 pages (2012)

[16] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, D. Keyes: Multicore-Optimized Wavefront Diamond Blocking for Optimizing Stencil Updates. SIAM J. Sci. Comput., 37(4), C439–C464 (2015).

[17] C. Huang, O. Lawlor, L. V. Kale: Adaptive MPI, Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science Volume 2958, pp 306-322 (2004).

[18] L. V. Kale, S. Krishnan: CHARM++: A Portable Concurrent Object Oriented System Based On C++, ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 91-108 (1993).

[19] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi and S. Chakradhar: A Virtual Memory Based Runtime to Support Multi-tenancy in Clusters with GPUs, ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC '12), pp 97-108 (2012).

[20] H. Midorikawa, H.Tan: Locality-Aware Stencil Computations Using Flash SSDs as Main Memory Extension, IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2015), pp. 1163-1168 (2015).

[21] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, IEEE/ACM SC'11, 12 pages (2011).

[22] T. Shimokawabe, T. Aoki and N. Onodera: High-productivity Framework on GPU-rich Supercomputers for Operational Weather Prediction Code ASUCA, IEEE/ACM SC'14, pp. 251-261 (2014).

[4]https://developer.nvidia.com/cuda-toolkit