Realizing Extremely Large-Scale Stencil Applications on GPU Supercomputers

Toshio Endo Yuki Takasaki Satoshi Matsuoka
Tokyo Institute of Technology, Japan
Email: {*endo, matsu*}*@is.titech.ac.jp*

***Abstract*—The problem of deepening memory hierarchy towards exascale is becoming serious for applications such as those based on stencil kernels, as it is difficult to satisfy both high memory bandwidth and capacity requirements simultaneously. This is evident even today, where problem sizes of stencil-based applications on GPU supercomputers are limited by aggregated capacity of GPU device memory. There have been locality improvement techniques such as temporal blocking to enhance performance, but integrating those techniques into existing stencil applications results requires substantially higher programming cost, especially for complex applications and as a result are not typically utilized. We alleviate this problem with a run-time GPU-MPI process oversubscription library we call HHRT that automates data movement across the memory hierarchy, and a systematic methodology to convert and optimize the code to accommodate temporal blocking. The proposed methodology has shown to significantly eases the adaptation of real applications, such as the whole-city airflow simulator embodying more than 12,000 lines of code; with careful tuning, we successfully maintain up to 85% performance even with problems whose footprint is four time larger than GPU device memory capacity, and scale to hundreds of GPUs on the TSUBAME2.5 supercomputer.**

I. Introduction

One of the most serious problems in the exascale era is the deepening memory hierarchy, as the exascale architectures will be composed of limited amount of upper-tier high bandwidth memory and hierarchies of larger capacity but decreasingly lower bandwidth memory underneath. While such an architecture would be fine for compute-intensive applications, it would become a significant obstacle for scaling bandwidth-intensive applications such as weather simulation, organ simulation in medical applications, and various disaster management applications such as earthquakes and tsunamis. Although finer-grained meshes to achieve higher resolution will require significant increase in memory capacity of upper-tier high-bandwidth memory to maintain performance, such increase will be slower than the performance growth of the processing power of the CPUs, largely restricted by device physics, packaging limitations, as well as power restrictions.

In fact, we already suffer from this problem especially on heterogeneous systems equipped with accelerators such as GPUs or Xeon Phi processors. On such systems, while processing speed and memory bandwidth are high (around 1∼2TFlops and 250∼300 GB/s per accelerator circa 2015), memory capacity per accelerator is limited to only several gigabytes, while the host CPUs embody slower DRAMs are equipped with much higher capacity, being tens or even hundreds of gigabytes. As a result, although various types of stencil-based applications have been demonstrated successfully on GPU clusters, their problem sizes have been limited to fit within the GPU device memory, and have not utilized the large capacity host DRAM at all [1], [2], [3], [4], [5].

This problem in the capacity can be conceptually mitigated algorithmically with *locality improvement* techniques that can effectively utilize the memory hierarchy by reducing the bandwidth requirements. We have demonstrated that stencil computations can enjoy both higher performance and larger problem size with such an approach[6], [7]. To enlarge the problem sizes of stencil kernels, we place the problem in the host memory, whose size is much larger than the device, as shown in the architectural sketch of a GPU computing node in Figure 1. Here, naive usage of the host memory would result in disastrous performance, since they are accessible from GPU cores only via PCI-Express bus (hereafter PCI-e), which is 10x to 30x slower than bandwidth of device memory. Instead, we apply a locality improvement technique called *temporal blocking*, which has been proposed mainly for cache locality improvement[8], [9], [10], to this hierarchy. This approach was shown to work well to effectively enlarge the problem size beyond device memory capacity in toy programs such as a 7-point stencil; however, it is still not clear whether the approach will be easily applicable to real-world simulation applications, which typically consist of thousands lines of code, and embody complex structures including sophisticated boundary conditions and inter-node communication.

The goal of this work is to devise a systematic methodology to achieve high performance, large problem sizes, and low programming cost *simultaneously* in real-world bandwidth-intensive simulation applications on GPU supercomputers. We consider two large-scale stencil-based simulation applications as examples, namely the whole-city turbulent airflow simulation based on advanced Lattice Boltzmann Method[4], and dendritic solidification simulation based on the phase-field method[5], the latter of which was awarded the ACM Gordon Bell Prize in 2011. Although both are written with MPI and CUDA[11] and demonstrate almost linear scaling and petaflops performance by utilizing thousands of GPUs of TSUBAME2.5 petascale supercomputer, nonetheless they are mutually very different in terms of the underlying numerical algorithms. Both assume the high bandwidth of the GPU device memory to achieve high performance, and as a result their problem sizes are limited by the aggregated capacity of device memory of the GPUs used; moreover, both are well over thousands of lines

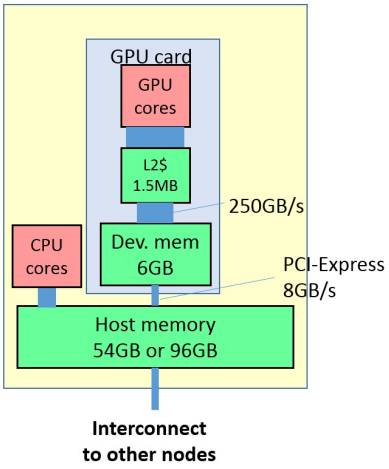

Fig. 1. Memory hierarchy of a GPGPU computing node. This illustrates the simplified architecture of a node of TSUBAME2.5 supercomputer we used in our evaluation.

of code, and thus incorporating temporal blocking would be extremely cumbersome and error-prone.

In order to minimize the code refactoring costs of such real applications to incorporate temporal blocking, we utilize a run-time library for GPUs called *Hybrid Hierarchical Runtime (HHRT)* that allows effective co-management of GPU device memory and host memory[12]. The current HHRT library is a wrapper library of MPI and CUDA; its functionality includes virtualization of device memory capacity and automated memory swapping between the hierarchies in memory. HHRT allows the users to effectively refactor the code by distinguishing (1) the code change that is really required for locality improvement of algorithm from (2) other auxiliary functions that can be offloaded to HHRT, allowing the user programmer to focus on (1). While this approach has been applied to a simple 7-point stencil benchmark[12], we have extended our methodology to cover above real-world applications, requiring further optimizations and feature extensions to the HHRT runtime.

By applying temporal blocking with the aid of HHRT, experiments on the TSUBAME2.5 supercomputer equipped with NVIDIA K20X GPUs, shows that the airflow simulation with 4x larger problem size than device memory achieves 85% of performance compared to smaller problems confined within the GPU, being significantly faster than CPU-only implementation, while code refactoring was relatively simple with small amount of code change. Similar result was achieved with the dendrite simulation, achieving 74% of performance for a problem beyond device memory capacity. These demonstrate that with the proposed methodologies and techniques real-world bandwidth-intensive applications can enjoy higher performance and larger problem size, with moderate programming costs, without significant increase in high-bandwidth memory capacity increase, but rather with deep memory hierarchy of today towards exascale.

II. Background

A. Memory Hierarchy of GPU Machines

We first give a brief overview of memory hierarchy of GPU-based machines. Here we focus on NVIDIA GPUs, but it would be applicable to other accelerated architectures such as AMD FireStream and Intel Xeon Phi. In such architectures, the host CPU memory has large capacity, ranging from several 10s to 100s of GigaBytes (GB for short), while the bandwidth is limited to 50∼100 GB/s. By contrast, accelerator CPUs such as GPUs embody significantly higher bandwidth, currently ranging at 200 300 GB/s, while their capacity would be limited to 4 to 16 GB. Moreover the bandwidth of the accelerators is expected to be elevated significantly to TeraByte (TB)/s range due to the adoption of 3D memory packaging such as HBM and HMC, but the capacity increase will be limited. In order to overcome the capacity limitations, the CPU host memory is usually regarded as a lower-tier memory in the overall memory hierarchy. Also, currently the transfer speed between the memory tiers is restricted to PCI-e bandwidth of 8∼16 GB/s; although there are future developments to overcome this limitation, nonetheless efficient transfer of data between the tiers will remain a problem irrespective. For our TSUBAME2.5, the GPU memory bandwidth is 250GB/s and the capacity is 6 GB, for CPU they are 51.2GB/s and 54∼96 GB respectively, and CPU-GPU bandwidth is 8 GB/s.

B. Stencil Computation

Stencil computation is a very typical kernel that appears in solvers for various HPC applications such as CFD. The simulation domain is partitioned into regular-sized grids, and repeated computation is performed to update the value of each point in the grid. The value of a point is determined locally by the surrounding points in a fixed 'stencil structure' and thus was named so. Since the update can be computed entirely in parallel, it is highly parallelizable, but typically is memory bound due to the requirements of reading many data points to compute just the data of a single point. Various optimization strategies such as spatial caching ad double buffering are typically employed to increase locality and significantly boost performance at the risk of code complexity.

Figure 2 (a) shows a simple example of a 7-point stencil program using MPI and CUDA, where the data are confined within the GPU device memory for highest efficiency. By contrast, Figure 2 (b) shows the case when data is larger than GPU device memory; here, the overall grid is partitioned into subgrids (or subdomaints), and they are proactively swapped for the computation to proceed. We observe that, not only there is the MPI communication, but we must perform two CPU-GPU transfers for every subgrid per each step, leading to significant slowdown due to slow CPU memory as well as even slower CPU-GPU transfer bandwidth.

C. Temporal Blocking

Temporal blocking is a locality-improving technique for stencil computations[8]. Here, we subdivide the grid into small subgrids as before, but advance the timesteps of each subgrid multiple times independent of others. This improves the locality of stencil computations considerably—although used for CPUs in the context of improving the cache hit

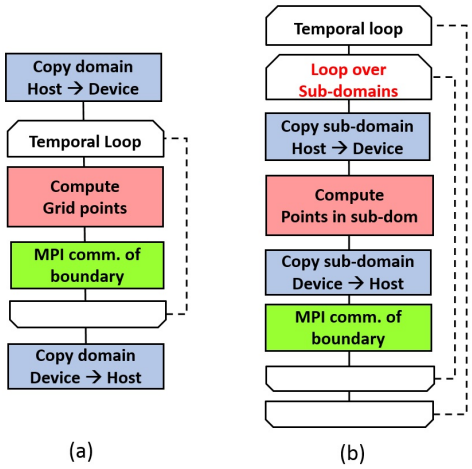

Fig. 2. (a) Stencil with MPI+CUDA, (b) Large Scale Stencil Algorithm

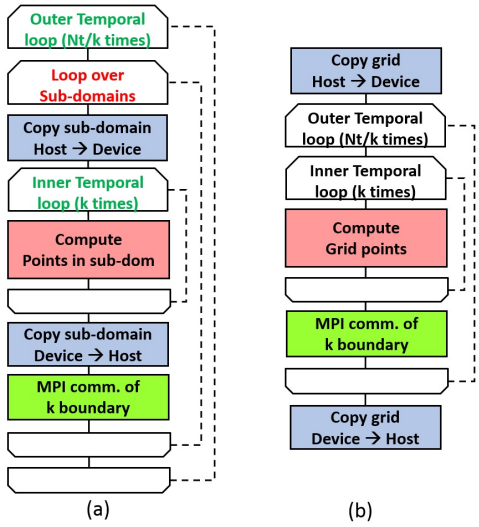

Fig. 3. Large Scale Temporal Blocking Algorithm. (a) Hand-coded Implementation, (b) An Implementation Combined with HHRT

ratio, we have applied the technique effectively to GPU-based computation to reduce the traffic of GPU-CPU communication significantly [6], [7].

Figure 3 (a) shows the improved algorithm that applies the temporal blocking technique to a GPU-based stencil program. We observe that there is a temporal loop inside the loop that enumerates over the subgrids. If we have k to be the temporal *width* of the temporal blocking (also called the *temporal blocking factor*), and we advance the timestep Nt times for the whole application, since each subgrid will compute k times in the inner temporal loop, the exterior temporal loop will enumerate Nt/k times. Compared to Figure 2(b), we have reduced the number of GPU-CPU transfer to be $1/k$ times, despite that the amount of transferred data remains constant (entire inner sub region + halo region). As such, the amount of data transferred during the application run is reduced to $1/k$ of the original program.

It is fairly simple to rewrite the program in Figure 2(b) to its temporal blocked version in Figure 3 (a) for toy program, however, for real applications that involve thousands or even tens of thousands of lines of code, embodying multiple complex stencils with varying boundary conditions and sophisticated communication patterns, such a rewrite would be overwhelming in the programming cost. In order to ease such a rewrite, we utilize the HHRT runtime that allows virtualization of GPU device memory in a way such that transfer management is largely hidden from the programmer.

D. HHRT

HHRT (Hybrid Hierarchical Runtime)[12] is a runtime that virtualizes the application GPU+MPI processes, and multiplexes them by supporting automated memory swaps between the tiers of the memory hierarchy, in order to reduce the cost of programming by eliminating the user-managed memory transfers. The current version of HHRT supports code written with CUDA and MPI to automate the GPU-CPU memory swaps, but will be extended to support other inter-tier transfers as well as other processor types such as Intel Xeon Phi. Figure 4 compares the typical execution model of applications on CUDA and MPI, to that on HHRT. Instead of letting each MPI process occupy a GPU, we let several MPI processes share a GPU. HHRT virtualizes and allows multiple CUDA-MPI processes to share GPUs and utilize memory space larger than the device memory, by appropriately swapping the processes in-out on demand, usually at the time of MPI communication. For details, readers are referred to [12].

There are various uses of HHRT, but in particular when applied to temporal blocking, it largely automates the memory hierarchy management. Although it does not automate the temporal blocking itself, our previous work showed that the program can be significantly simplified as seen in Figure 3 (b) compared to Figure 3 (a) [12]. The programmer essentially converts the program to a temporal blocked GPU+MPI process by merely adding an inner temporal loop with the halo increased in width to the temporal blocking factor k, as if the process still occupies the entire GPU device memory; such a conversion was found to be relatively simple, even for real applications we have seen so far. All the complexities associated with memory management, especially transfer between device and host memory, are automatically handled by the HHRT runtime.

However, although the conversion was simple for a toy, 7- and 9-point stencil programs with no boundary conditions, whether HHRT is ubiquitously applicable to (a) significantly reduce the programming burden for real stencil-based applications, and (b) whether we can attain performance levels of upper-tier high-bandwidth memory while growing the problem size beyond their size, up to the capacity of lower-tier memory, had not been investigated. Moreover, it has not been shown whether such programs, which are typically weak-scaling applications that could scale to machines with thousands of GPUs such as ORNL Titan or Tokyo Tech. TSUBAME2, exhibiting petaflops of performance, would scale similarly with HHRT-based temporal blocking.

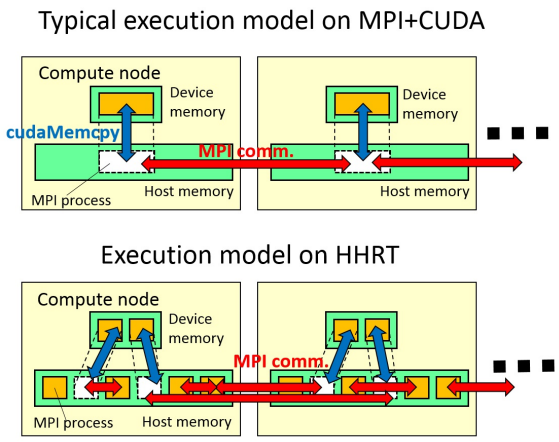

Fig. 4. Execution model on typical MPI+CUDA and execution model on HHRT library.

III. Target Stencil Applications

We next describe our target stencil applications. Both currently scale to petaflop(s) on TSUBAME2.5 using over 4,000 GPUs, provided that the entire dataset is confined to be within GPU device memory.

A. Whole-City LES-LBM Turbulant-Flow Airflow Simulation

Our first application is the whole-city turbulent flow airflow simulation presented in [4]. Numerically it is a massively parallel LBM (Lattice-Boltzmann Method) – LES (Large-Eddy Simulation) utilizing the coherent structured SGS model to effectively handle turbulent flows. The $10km \times 10km \times 500m$ region of central Tokyo had been digitized, including the terrain and all the constructions such as the roads and buildings. The spatial resolution of the simulation is 1 meter, or 5×10^{11} degrees of freedom, allowing accurate simulation of flows along the roads and rivers, as well as turbulent flows around the skyscrapers (Figure 5).

LBM is a naturally stencil-based computation, but its stencil is significantly complex, as 19 or 27-directional velocities of the virtual particles are expressed as lattice data. More concretely, the airflow simulation embodies the following characteristics, which makes the rewrite more difficult than a toy benchmark:

- This simulation requires a pair of 19 or 27 three-dimentional arrays for double buffering, and additional 16 three-dimentional arrays for physical values.
- The 7-point stencil only has one kernel function, whereas the whole-city simulation embodies six kernels that are computed in sequence.
- The 7-point stencil has a simple fixed (Dirichlet) boundary condition on all axis, whereas the whole-city simulation has a periodic boundary condition on the X-axis, and Neumann boundary condition on the Y- and Z-axis.

There are various other factors involved, and as a result the code size of the airflow simulation is 12,155 lines of C+CUDA+MPI code (counted by the CLOC tool [1]).

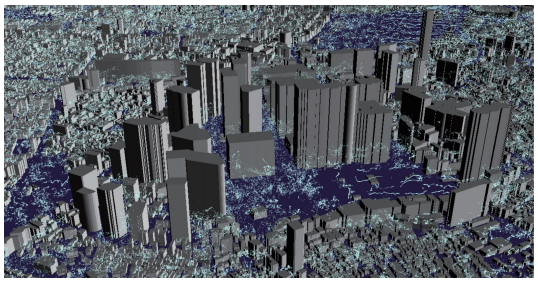

Fig. 5. Visualization of the Whole-City Airflow Simulation (by Courtesy of Takayuki Aoki)

B. Dendritic Solidification Simulation based on the Phase-Field Method

Our second simulation is the dendritic solidification simulation presented in [5], which was awarded the 2011 ACM Gordon Bell Prize. The simulation of microstructure of materials is required to use finer-grained grids as well as coverage of significant physical size in order to properly understand their mechanical properties. In order to fulfill the requirements, the particular simulation uses the phase-field method, which is known as the most powerful method to simulate the micro-scale dendritic growth during solidification (Figure 6).

Numerically, the simulation is also based on stencil computation on a 3-D array as is with the whole-city simulation, but the underlying stencil computation is very different. The followings are the detailed characteristics of the dendrite simulation.

- The simulation expresses the phase field ϕ and the concentration c as pairs of 3-dimensional arrays for double buffering. It additionally uses three 3-dimensional arrays.
- The simulation calls a single GPU kernel written in CUDA per iteration. The kernel is highly optimized and consists of around 270 lines of code.
- It employs sophisticated GPU-CPU as well as MPI communication strategies to hide the MPI communication latency.

The code size of the dendrite simulation is 4,684 lines of C+CUDA+MPI code.

IV. Temporal Blocking of Real-World Applications using HHRT

We now describe the general methodology we have devised to apply HHRT to implement temporal blocking on real-world stencil application codes. The basic strategy is fundamentally the same as toy programs, in that the application kernels in Figure 2(a) are converted to HHRT-enabled code as seen in Figure 3(b), and executed on top of the HHRT runtime, but further optimizations are performed to achieve performance close to that of device memory only execution. In order to discuss programming costs, we show the number of changed lines of code step by step for both applications in Tables I.

[1] http://cloc.sourceforge.net

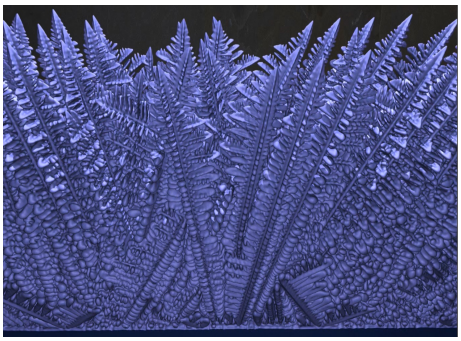

Fig. 6. Visualization of the Dendritic Solidification Simulation (by Courtesy of Takayuki Aoki)

TABLE I. THE NUMBER OF CHANGED LINES OF SIMULATION CODE.

The Airflow Simulation
(The original code consists of 12,155 lines)

	modified	added
Enabling HHRT	0	16
Temporal blocking	104	32
Comm. opt.	416	488
Annotation	0	9

The Dendrite Simulation
(The original code consists of 5,295 lines)

	modified	added
Enabling HHRT	0	5
Temporal blocking	50	68
Annotation	0	15

A. HHRT Enabling of the Code

As a first step, we link the HHRT runtime, thereby allowing the aggregate memory usage of all the processes to exceed the device memory size. Without HHRT, the programmer divides up the original array implementing the grid into sub-regions, and explicitly code the CPU-GPU memory transfer that overlaps with computation, requiring considerable restructuring of the loop in the kernels as seen in Figure 2(b), also considering inter-node MPI communication. With HHRT this becomes unnecessary, as the transfers will be automatically handled by the HHRT runtime.

B. Temporal Blocking Feasibility Check and Implementation

Not all stencil computations are subject to locality improvement via temporal blocking and thus subject to HHRT optimization; it is only possible when the value of point x in the lattice can be updated in the next temporal step by using only the values of the locally surrounding points (i.e. stencil) of x in the previous timestep. When an algorithm requires some global operation per timestep, e.g., the Conjugate Gradient method in which a global dot product must be computed for each timestep, temporal blocking optimization is not possible. In our examples, code reviews were conducted for the kernels of both the whole-city airflow simulation and dendrite simulation, and found that in both cases temporal blocking optimizations are applicable.

We then hand-converted the codes from Figure 2(a) to Figure 3(b); as we quantitatively assess later, in terms of the number of lines in the code changes were fairly localized and small. More explicitly, the following changes were made:

1) The exterior single temporal loop was made into a double nested loop according to the temporal blocking width k (we call each of double loops "outer" one and "inner" one).
2) MPI communication was moved to reside in-between the outer and inner temporal loops, and made to exchange k steps worth of halo region all at once.
3) The loop that enumerated over the stencil elements spatially was modified to decrease the loop interval by one step on both the loop start and end so that the halo size would appropriately increase by one step.
4) Computation of the Neumann boundary condition was left to remain within the inner temporal loop; for periodic boundary condition, the required MPI communication was moved to be between the inner and outer temporal loops in the same manner to above

By contrast, we did not have to modify the code of the numerical algorithm itself within the loop body at all, which of course is usually the most dominating part and quite error prone in their modifications. Code motion was only necessary for MPI communication in steps 3 and 4 above. As shown in Tables I, the amount of source code change was only 139 lines out of the 12,000 total for the whole-city simulation, and 118 out of 5,000 for the dendrite simulation.

C. Optimizing MPI communication with HHRT Swaps

Although temporal blocking was incorporated, we next conduct a series of simple set of optimizations in a systematic manner to improve performance, aided by HHRT optimization features. On monitoring the actual execution of the HHRT-enabled temporal blocking whole-city simulation code on TSUBAME2.5, we experienced three times more HHRT swaps than anticipated, resulting in significant overhead. A similar issue has been observed with the dendrite simulation. This was due to the communication structure of the original code: HHRT conducts swaps between the virtual processes when MPI blocking communication is called. The whole-city airflow simulation requires communication of the halo region with 26 other processes out of the 27 directions in the LBM lattice, and as the lattices are decomposed three-dimensionally, three MPI blocking communications are conducted in X-, Y-, and Z-directions in sequence. This results in five HHRT swaps (three swap outs and two swap ins) per each temporal loop instead of once, resulting in significant PCI-e transfer overhead.

To alleviate this problem, we took different approaches for the two applications. For the dendrite simulation, we added HHRT API calls `HH_setDevMode` to specify the code region where the virtual process never calls the GPU kernel functions. During execution of such code regions, skipping HHRT swaps is harmless since data on device memory are not touched by the kernel function. HHRT will cease to conduct swap-in subsequently after a virtual process calls `HH_setDevMode (HHDEV_NOTUSED)`. Then, when the process calls `HH_setDevMode (HHDEV_NORMAL)`, HHRT executes immediate swap-in, and subsequently the process is allowed to call GPU kernels again. The changes required for the dendrite simulation was adding only two `HH_setDevMode` calls, which is indicated in the 'Annotation' field in Table I.

On the other hand, we found the same approach was inapplicable to the whole-city simulation, since it uses GPU kernel functions in the boundary communication part to re-order the boundary data on device memory. MPI blocking communications and GPU kernel functions for X-, Y-, and Z- directions are interleaved. Instead, we modified the MPI communication so that 26 non-blocking MPI sends and receives are issued at once, whose completion is determined by one `MPI_Waitall()`. Although there are 52 non-blocking communications issued, this resulted in significantly faster execution as we benchmark later. In practice, the amount of required code modifications were 904 lines, which is much bigger that the temporal blocking itself, due to the fact that the original three-dimensional communication code was fairly complex. In practice the modification was simple, despite the fairly large number of lines, as it was systematic and also closer to the 'native ' behavior of the algorithm. Nonetheless, as a possible improvement, we could combine `HH_setDevMode` API and re-ordering of the GPU kernel functions and MPI communications, which would significantly reduce the code changes[2].

D. Minimizing Memory Space of HHRT Swap

By default HHRT swaps all the GPU data in device memory onto host memory. However, not all data need to be transferred; thus we can optimize the overall CPU-GPU memory tier communication time by reducing its amount. For this purpose, HHRT has an API `HH_madvise` that allows designation of data not to be swapped in/out. More concretely, `HH_madvise` takes the pointer to the array, data size, and transfer state as arguments, allowing switching on/off of the transfer at swap time by specifying `HHMADV_NORMAL / HHMADV_CANDISCARD` as the state argument, the latter designation allowing HHRT not to transfer the data. While `HH_devSetMode` described above specifies 'when' HHRT can skip swapping in/out, `HH_madvise` specifies 'which data' should be swapped in/out.

A simple observation for general double-buffered stencil computation is that, only one buffer needs value preservation, and thus the other does not need to be swapped out. In realistic applications, however, such an optimization does not result in a performance gain. Instead, such applications typically embody multiple arrays that are used temporally and do not need to be carried over to the next timestep. Since HHRT swap occurs only at the point of blocking MPI communication, at which time the interior computation of a sub grid is already completed for that time step, we have an opportunity of significant savings if we can identify many such arrays. For both the whole-city simulation and dendrite simulation, we were able to identify several such arrays, allowing us to conduct our optimization, which is only a two-line addition per array to call `HH_madvise()` as in the 'Annotation' rows of Tables I.

V. Performance Evaluation

A. Evaluation Environment

We tested our HHRT-enabled temporal blocked stencil applications at scale on the TSUBAME2.5 supercomputer hosted by the Global Scientific Information and Computing Center, Tokyo Institute of Technology. Each node of TSUBAME2.5 facilitates two Intel Xeon 5670 2.93GHz (6 cores) CPUs with 54GB or 96GB of DDR3-1333 memory, and three NVIDIA Tesla K20X GPUs, each with 6GB, 250GB/s GDDR5 memory. The nodes are interconnected with a dual-rail Infiniband QDRx4 supporting 80Gbits/s injection bandwidth. In our experiments, we utilized 1 GPU per node, with a 96GB node for single host experiments, and 54GB nodes for multi-node experiments. The software environment is SUSE Linux 11 sp3, gcc 4.3.4, OpenMPI 1.6.5, and CUDA 6.5.

B. Evaluation on a Single GPU

We first conduct a single-node, single-GPU performance analysis by comparing the four versions of the two stencil application programs:

- ORG: the original program that does not use HHRT nor temporal blocking
- HH: The HHRT-enabled version of ORG
- +TB: Temporal Blocking added to HH
- +TB+Opt: MPI streamlining/optimization added to +TB (only for the whole city simulation)
- +TB(+Opt)+Anno: `HH_madvise` and/or `HH_devSetMode` optimization added to reduce CPU-GPU transfer

Figures 7 and 8 show the performance of the two simulation applications for each of the program instances above, varying the problem sizes, tuned to optimal temporal blocking factors and partitioning. As the K20X GPU device memory size is limited, ORG cannot execute problem sizes beyond 6GB, whereas HH can execute problem sizes up to approximately 48GB. However, this is at the cost of significant performance degradation; compared with ORG, it exhibits only 5% performance in the whole-city simulation and 1.5% in the dendrite simulation. This is due to the occurrence of HHRT swap per every timestep. By contrast, the +TB or the temporal blocked version shows 4.9 times (in the whole-city simulation) speedup over HH, or approximately 27% of ORG. Since +TB conducts 3 swaps per timestep whereas the +TB+Opt only once as described earlier, we observe another 2.4 times speedup, reaching to 64% of ORG. We also observe +TB of the dendrite simulation suffers from swap costs. Finally by reducing unnecessary PCI-e traffic during HHRT swap with the HHRT advice mechanism, +TB+Opt+Anno demonstrates additional 1.4 times speedup, to achieve 85% of ORG performance when we use problem size of 16GB. The final version of the dendrite simulation is also largely improved, reaching 74% of ORG, when the problem size is 11GB.

We do note that we do observe slow degradation of performance as the problem size grows; this is largely due to MPI process management cost and implicit device memory consumption when processes are bound tGPUs. Currently NVIDIA device driver requires about 72MB of management area in the GPU device memory per process, thus compromising useable device memory as we add more processes for optimal temporal blocking. Also HHRT allows the usage of only half the host memory to accommodate swap capacity. In both cases we reach the limit of temporal blocking factors

[2]The camera ready version will show the results of this improvement

due to memory capacity, thus underachieving the potential performance gains had we been able to further increase the blocking factors.

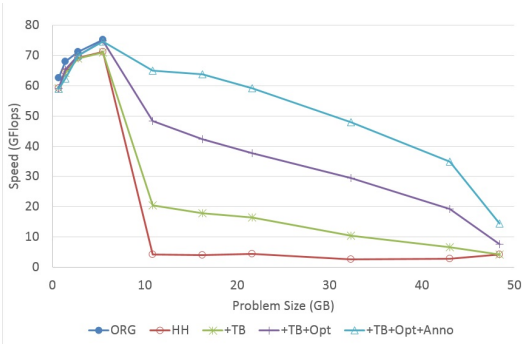

Fig. 7. Performance Results of Airflow Simulation on a Single GPU

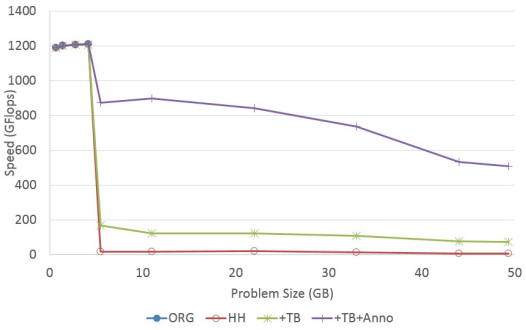

Fig. 8. Performance Results of Dendrite Simulation on a Single GPU

In order to see the effect of the number of temporal blocking factors k, Figure 9 shows the performance variations of the airflow simulation, when we alter k for +TB+Opt and +TB+Opt+Anno (final version). Due to the reduction of the HHRT swap overhead with HH_madvise, optimal blocking factors differ, with the latter requiring smaller blocking factor due to the lower latency of CPU-GPU tier memory transfer. Also we observe dramatic decrease in performance when blocking factor is further increased; this is due to the redundant computation occurring as a result of temporal blocking overwhelming the benefits in reducing the memory transfer cost.

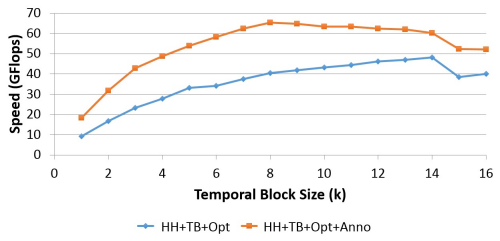

Fig. 9. The Effect of Varying Temporal Blocking Factors in the Airflow Simulation

C. Scaling to Multiple Nodes

Figures 10 and 11 show weak scaling performance on multiple nodes of TSUBAME2.5. Here one GPU per node is used for computation. ORG is configured so that each process occupies less memory than device memory capacity. Other lines correspond to the HHRT-enabled final version, and the problem size per node is described by the legends. Although there is some performance degradation (up to 16% with the Airflow simulation, and up to 47% with the Dendrite simulation) as we saw for the single node case, we observe that all the cases scale linearly without exhibiting abnormal performance slowdowns or tailoffs. Thus, our proposed method can significantly increase the problem capacity of large-scale parallel GPU machines, even if one can no longer utilize additional GPUs to increase the total available GPU device memory without sacrificing performance significantly.

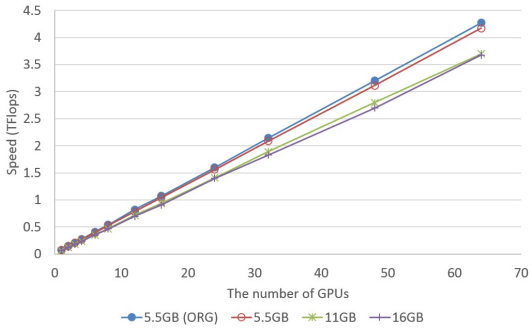

Fig. 10. Weak Scaling Results of the Airflow Simulation

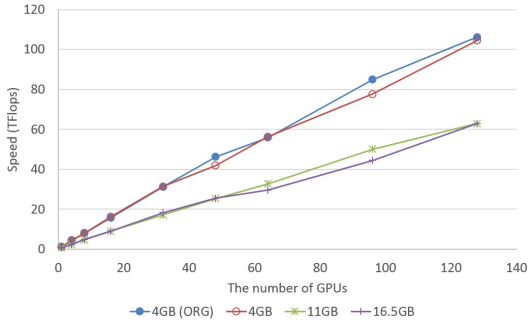

Fig. 11. Weak Scaling Results of the Dendrite Simulation

VI. Related Work

Many applications kernels are stencil based, and there have been various recent efforts to utilize the high memory bandwidth of GPUs to scale the real applications on large GPU-based supercomputers such as TSUBAME2.5[4], [5]. Although they have demonstrated performance scalability to petascale, their problem size remained limited due to the small capacity of high-bandwidth device memory on GPUs.

There have also been numerous optimization techniques proposed for stencil computation, mostly based on spatial and/or temporal blocking. Temporal blocking has been originally proposed to increase locality in the context of better CPU cache utilization [8], [9], [10]. For GPUs, there have been work

to reduce the memory traffic between device memory and on-chip shared memory[13]. Mattes et. al. proposed a temporally blocked FDTD implementation to reduce device-host memory communication [14]. Our previous work proposed additional optimization techniques for temporal blocking stencil code on GPUs, as well as demonstrated scalability to large parallel GPU supercomputers with thousands of GPUs [6], [7].

All such work however, does not focus on how to reduce the overall programming cost, both in terms of porting as well as tuning. It has been our experience that, unlike for toy programs, when the programmer ports and then attempts to tune the code for real stencil applications, he would have to significantly rewrite the temporal as well as spatial loops, and consider various interactions especially with MPI. By contrast, HHRT runtime offers automated memory hierarchy management features that not only allows easy porting of MPI+CUDA stencil code, requiring only small changes to the temporal loop and MPI communication, but still preserving majority of the performance when we exceed device memory, and moreover allowing weak scaling to a large number of GPUs.

Alternative approach to easing temporal blocking and other optimizations for stencil codes have been proposed with a family of DSL (Domain Specific Langauge)s, such as Physis[15] and ExaStencil[16]. Such approaches require that the entire application kernels be rewritten with DSLs, and differ from our runtime approach where we preserve much of the original code. DSLs do offer further optimization opportunity at compile time, however, and it will be subject to future work whether such optimization can also be attained with HHRT by auto-tuning methods.

VII. Conclusion

We have applied HHRT, a runtime for virtualizing GPU-MPI parallel programs and some of its novel features towards making it easy for the programmers to modify the codes so that they could run programs far beyond device memory capacity while preserving performance. By utilizing the HHRT features, programmers can systematically rewrite the temporal loops and conduct optimizations in an easy manner for MPI as well as memory hierarchy transfer to significantly increase performance of real stencil applications, and moreover makes it subject to easy tuning of the parameters. Porting experiences of two real petascale stencil with very different program characteristics nonetheless showed that HHRT significantly reduces such efforts. Benchmarks on a large-scale GPU supercomputer TSUBAME2.5 showed that, both applications largely preserve the performance and the scalability of code that originally only used the GPU device memory.

There are various future work ahead. One is to construct a higher-level framework such as C++ template metaprogramming to further reduce the programming and tuning cost. HHRT process management should be further optimized and/or automated to detect unnecessary transfers, and moreover should be merged with coherent memory features of upcoming GPUs and Xeon Phis. Other stencil programs involving different solvers should be examined to investigate applicability and possible new features. Finally, the current HHRT only handles memory management for a two-level hierarchy, but extended to handle three or more given the upcoming memory-class NVMs (non-volatile memory) and others that will further deepen the memory hierarchy to attain high capacity with lower power and cost.

VIII. Acknowledgements

This research is funded by JST-CREST, "Software Technology that Deals with Deeper Memory Hierarchy in Post-petascale Era". We also greatly thank Naoyuki Onodeara, Takashi Shimokawabe, and Takayuki Aoki who have kindly provided us with the simulation codes.

References

[1] E. H. Phillips and M. Fatica: Implementing the Himeno Benchmark with CUDA on GPU Clusters, in proceedings of IEEE IPDPS10, pp. 1-10 (2010).

[2] Dana A. Jacobsen, Julien C. Thibault, Inanc Senocak: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters, in proceedings of 48th AIAA Aerospace Sciences Meeting, Orlando (2010).

[3] M. Bernaschi, M. Bisson, T. Endo, M. Fatica, S. Matsuoka, S. Melchionna, S. Succi: Petaflop Biofluidics Simulations On A Two Million-Core System, in proceedings of IEEE/ACM SC11, 12pages, Seattle (2011).

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

[5] 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, in proceedings of IEEE/ACM SC11, 11pages, Seattle (2011).

[6] G. Jin, T. Endo and S. Matsuoka: A Multi-level Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPU. in proceedings of AsHES workshop, in conjuncition with IEEE IPDPS2013, pp.1080-1087 (2013).

[7] G. Jin, T. Endo and S. Matsuoka: A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPUs. in proceedings of IEEE CLUSTER2013, pp. 1–8, (2013).

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

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

[10] T. Minami, M. Hibino, T. Hiraishi, T. Iwashita and H. Nakashima: Automatic Parameter Tuning of Three-Dimensional Tiled FDTD Kernel. in proceedings of iWAPT2014, 8pages (2014).

[11] NVIDIA CUDA Toolkit, `https://developer.nvidia.com/cuda-toolkit`

[12] Toshio Endo and Guanghao Jin: Software Technologies Coping with Memory Hierarchy of GPGPU Clusters for Stencil Computations. in proceedings of IEEE CLUSTER2014, pp. 132–139, (2014).

[13] 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 SC10, 13pages (2010).

[14] L. Mattes and S. Kofuji: Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids. in proceedings of ICMMT10, pp.1536–1539 (2010).

[15] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, in proceedings of IEEE/ACM SC11, 12pages, Seattle (2011).

[16] S. Apel, M. Bolten, A. Grösslinger, F. Hannig, H. Köstler, C. Lengauer, U. Rude, and J.Teich. ExaStencils: Advanced Stencil-Code Engineering. inSiDE, 12(2):60-63 (2014)