# Software Technologies Coping with Memory Hierarchy of GPGPU Clusters for Stencil Computations

Toshio Endo Guanghao Jin
Global Science Information and Computing Center,
Tokyo Institute of Technology/JST-CREST, Japan
Email: *endo@is.titech.ac.jp, jin.g.ab@m.titech.ac.jp*

***Abstract*—Stencil computations, which are important kernels for CFD simulations, have been highly successful on GPGPU clusters, due to high memory bandwidth and computation speed of GPU accelerators. However, sizes of the computed domains are limited by small capacity of GPU device memory. In order to support larger domain sizes, we utilize the memory hierarchy of GPGPU clusters; larger host memory is used for maintain large domains. However, it is challenging to achieve all of larger domain sizes, high performance and easiness of program development. Towards this goal, we combine two software technologies. From the aspect of algorithm, we adopt a locality improvement technique called temporal blocking. From the aspect of system software, we developed a MPI/CUDA wrapper library named HHRT, which supports memory swapping and finer grained programming model. With this combination, we demonstrate that our goal is achieved through evaluations on TSUBAME2.5, a petascale GPGPU supercomputer.**

## I. Introduction

One of important issues in constructing high performance system towards exascale era is the memory wall problem; the improvement of capacity and/or bandwidth of memory is slower than that of processors. This problem will be a significant obstacle in making larger and finer scale simulations in weather, medical and disaster measurement area on future supercomputers.

We already suffer from this problem especially on heterogeneous systems equipped with accelerators such as GPUs or Xeon Phi processors. On these systems, while processing speed and memory bandwidth are high (around 1TFlops and 100 to 250 GB/s per accelerator), memory capacity per accelerator is limited to 1 to 8 GB. 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 [1], [2], [3], [4].

This problem in the capacity can be mitigated by *locality improvement* techniques in algorithm so that we can effectively harness *memory hierarchy* in architecture. We have demonstrated that both higher performance and larger problem size are achieved in this approach [5], [6] as follows. To enlarge the problem sizes of stencil kernels, we harness the host memory, whose size is typically dozens of Gigabytes per node. An example of architecture of a GPGPU computing node is shown in Figure 1. It is natural that naive usage of the host memory is harmful, since its contents is only accessible from GPU cores via PCI-Express bus (hereafter PCIe), which is x10 to 30

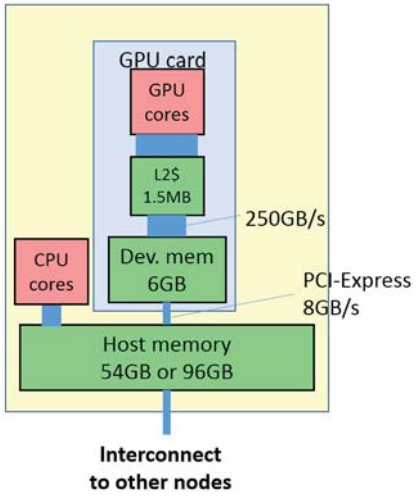


Fig. 1. Memory hierarchy of a GPGPU machine from the viewpoint of GPU cores. This illustrates the simplified architecture of a node of TSUBAME2.5 supercomputer we used in our evaluation.

slower than bandwidth of device memory. Instead, we adopted a locality improvement technique called *temporal blocking*, which has been proposed mainly for cache locality improvement [7], [8]. This approach worked well in the performance and problem size, but there is still an issue in *programming cost*. When we introduce temporal blocking technique into existing applications written in NVIDIA CUDA [10] and MPI, we found that we need intensive code rewriting as we will discuss in Section II-B.

The goal of this paper is to achieve the all of higher performance, larger problem size, and lower programming cost in stencil applications on GPU clusters. For this goal, we distinguish (1) the code change that is really required for locality improvement of algorithm from (2) other auxiliary code changes. By delegating the work related to (2) to the underlying library, user programmers can focus on code rewriting for (1). For this purpose, we designed and implemented a package named *Hybrid Hierarchical Runtime (HHRT)*. HHRT, used as a wrapper of CUDA and MPI, performs memory swapping between device memory and host memory. It supports a finer grained execution model, which

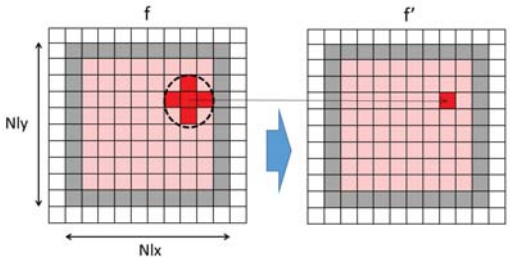


Fig. 2. Data structure that each MPI process maintains in the simple stencil computation. Data in dark grey areas are sent to neighbor processes, and white areas are used to receive data from neighbors.

```
 1: Copy array f from host to device
 2: for t in [0, Nt) {
 3:   Copy halo of f from host to device
 4:   Exchange halo of f with neighbors
      by MPI
 5:   Copy halo of f from device to host

      // done on GPU: START
 6:   forall (x, y) in [1,Nlx+1)x[1,Nly+1) {
        // compute one element
 7:     f'[x,y] := func(f[x,y],
                        f[x-1,y], f[x+1,y],
                        f[x,y-1], f[x,y+1])
 8:   }
      // done on GPU: END
 9:   Swap f and f' on GPU
10: }
11: Copy array f from device to host
```

Fig. 3. A pseudo code of "basic" stencil implementations on GPU clusters.

also contributes to mitigation of the programming cost.

Through the experiments on the TSUBAME2.5 petascale GPGPU supercomputer, equipped with NVIDIA K20X GPUs, we demonstrate that we can achieve higher performance, larger problem size, and lower programming cost.

## II. Stencil Computations and Temporal Blocking

### A. Stencil Computations and Problems on GPU Clusters

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. In order to simulate time evolution, time steps are repeated. In each time step, all the grid points are calcuated by using values of adjacent points in the previous time step. In this section, we focus on very simple "five-point" stencil computation on two dimensional grids [1] , where each point is calculated as:

```
f'[x,y] := func(f[x,y], f[x-1,y], f[x+1,y],
                f[x,y-1], f[x,y+1])
```

where $f$ is an array that corresponds to the old grid and $f'$ is new one.

[1]though we will use "seven-point" stencil on three dimensional grids in the evaluation

```
 1: for t in [0, Nt) {
 2:   Exchange halo of f with neighbors by MPI

 3:   Decompose domain [1,Nlx+1)x[1,Nly+1)
      into several sub-domains

 4:   for sd in sub-domains {
        // here we assume sd corresponds to
        // [sx, ex)x[sy,ey) of f
 5:     Copy [sx-1, ex+1)x[sy-1,ey+1) of f
        from host to device
        // done on GPU: START
 6:     forall (x, y) in [sx,ex)x[sy,ey) {
 7:       f'[x,y] := func(f[x,y],
                          f[x-1,y], f[x+1,y],
                          f[x,y-1], f[x,y+1])
 8:     }
        // done on GPU: END
 9:     Copy [sx,ex)x[sy,ey) of f'
        from device to host
10:   }
11:   Swap f and f' on host
12: }
```

Fig. 4. A naive stencil implementation for larger domain (named "Decomp"). The domain of each process is further decomposed so that each sub-domain fits into the device memory. This is very slow due to heavy PCIe cost.

First we discuss a basic implementation of this computation on GPU clusters. We assume that the code is written in CUDA and MPI, and each process is bound to a single GPU. The grid to be simulated is distributed among MPI processes, and the divided local arrays $f$ and $f'$ are illustrated in Figure 2. Figure 3 illustrates the behavior of each process. Here $Nt$ is the number of total time steps to be simulated, and $Nlx, Nly$ correspond to the size of area to be computed. At the beginning, the initial contents of the array $f$ are copied from host to device memory (line 2), and then we start the temporal loop. Before computation in each step, we need MPI communication of process boundary region (called *halo*) between adjacent processes, due to data dependency between adjacent grid points (lines 3 to 5). After that, the process computes all the elements of the local array $f'$ by using the GPU cores in parallel (lines 6 to 8).

The limitation of this implementation is in problem size; we cannot execute it if the domain size per GPU (total size of arrays f and f') is larger than the capacity of device memory.

A naive approach to support larger domain (Figure 4) is as follows. Each process basically holds the local arrays, which may be larger than device memory, on the host memory. Then the arrays are decomposed into several sub-domains, so that each sub-domain is smaller than device memory. Every time step, the process takes a sub-domain and copies it to device (line 5), compute its elements (line 6 to 8) and copies it back to host (line 9). This implementation suffers from costs for PCIe communication of the whole array every step, which is far heavier than halo exchange.

Figure 5 demonstrates the problems of the above two implementations. It shows the performance for varying problem sizes, using a single K20X GPU with 6GB device memory. Obviously the first one ("Basic" in the graph) cannot support problems larger than 6GB. The second one ("Decomp") works,

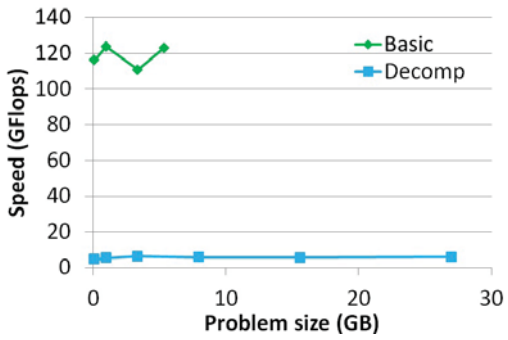


Fig. 5. Performance of implementations in Figures 3 (Basic) and 4 (Decomp). "Basic" cannot support larger domains and "Decomp" exhibits too low performance.

but it is about 20 times slower.

### B. Temporal Blocking

In order to support larger domain sizes with lower costs, we adopt a known technique, called *temporal blocking*. Some researchers have used it for improving cache hit ratio[7], [8], and others coped with limited capacity of GPU device memory [5], [6], [9], which is the target of this paper.

With temporal blocking, we divide the arrays into sub-domains as done in "Naive" implementation. Unlike "Naive", however, we execute the computation of a single sub-domain for several time steps at once, instead of moving to the next sub-domain immediately.

Figure 6 shows an example of a stencil implementation with temporal blocking, named Hand-TB. Here $k$ is *temporal block size*, which is the number of steps computed at once. In order to proceed a single sub-domain for $k$ steps at once, we introduce doubly nested temporal loop; the outer loop starts at line 1 in the pseudo code and the inner loop starts line 6. Now PCIe communications (lines 5 and 13) are kicked out of the inner loop, and thus frequency and amount of PCIe communication reduced to $1/k$ of that of "Decomp" (Figure 4).

This concept is very close to classical iteration blocking technique introduced in dense matrix computations[11], however, the characteristic of stencil computation, where each point computation involves adjacent points, makes things more complicated as shown in Figure 7. In order to obtain the results at $(t0+k)$-th time steps of a sub-domain, we have to prepare "extra halo" area for the sub-domain at $t0$-th time step as input. For this purpose, the area size to be computed (specified in line 7 and 8) changes in the inner loop.

Here let us mention that this characteristic introduces redundant computation for the overlapped area between sub-domains. Although we have also presented that removing the redundant computation improves the performance further[5], we omit it in this paper.

### C. Issues in Hand-coding Temporal Blocking

With temporal blocking, we can achieve both higher performance and larger problem sizes. However, there still remains

```
   // outer temporal loop
 1: for t0 in [0, Nt) with stride k {
 2:   Exchange k-halo of f with neighbors by MPI

 3:   Decompose domain [k,Nlx+k)x[k,Nly+k)
      into several sub-domains

 4:   for sd in sub-domains {
        // here we assume sd corresponds to
        // [sx, ex)x[sy,ey) of f
 5:     Copy [sx-k, ex+k)x[sy-k,ey+k) of f
        from host to device

        // inner temporal loop
 6:     for t in [t0, t0+k) {
 7:       r := k-1-(t-t0) // r changes k-1 ... 0
          // done on GPU: START
 8:       forall (x, y) in
              [sx-r,ex+r)x[sy-r,ey+r) {
 9:         f'[x,y] := func(f[x,y],
                            f[x-1,y], f[x+1,y],
                            f[x,y-1], f[x,y+1])
10:       }
          // done on GPU: END
11:       Swap f and f' on GPU
12:     }

13:     Copy [sx,ex)x[sy,ey) of f
        from device to host
14:   }
15: }
```

Fig. 6. A hand-coded implementation of temporal blocking (named Hand-TB). This has a largely different structure than "Basic" (Figure 3).

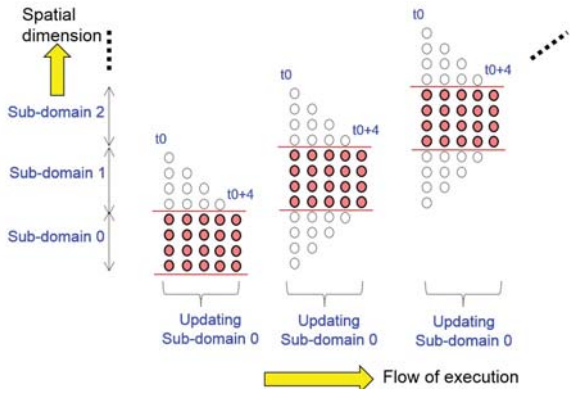


Fig. 7. Temporal blocking technique introduces redundant computation for overlapped area. Here the temporal block size $k$ is 4. For simplicity, the figure assumes one dimensional domain.

an issue in programming costs. Let us assume that we already have a working stencil application written in the "Basic" style as in Figure 3, and then we try to rewrite the code to support large scale domain. In this rewriting process towards the "Hand-TB" implementation (Figure 6), we have to take care of the following issues.

First, we need to add two types of loops, the inner temporal loop and the sub-domain loop. Also dividing local domains into sub-domains itself will involve extensive modification of data structure in the existing code. Finally, we have to

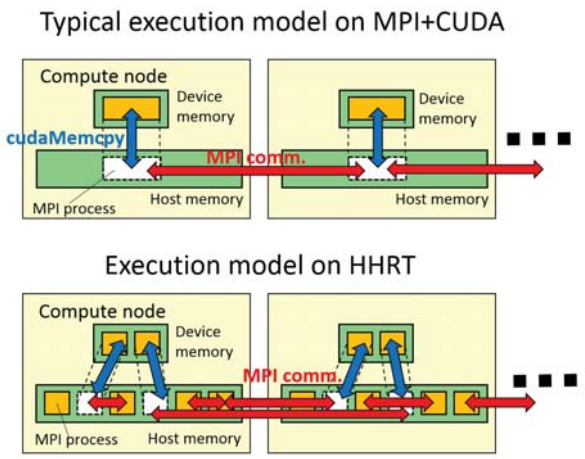


Fig. 8. Execution model on MPI/CUDA and execution model on HHRT library.

change the timings of memory copy between host and device, which was outside the temporal loop in "Basic", while it is included in the sub-domain loop in "Hand-TB". With all these changes, the structure of "Hand-TB" heavily differs from that of "Basic".

Against this issue, our approach is to distinguish (1) the code change that is really required for locality improvement from (2) other auxiliary code changes. By delegating the work related to (2) to the underlying HHRT library described in the next section, user programmers can focus on code rewriting for (1).

## III. The HHRT Library

The role of the Hybrid Hierarchical Runtime (HHRT) library is to execute applications written with CUDA and MPI on top of it, while hiding the limitation of the capacity of GPU device memory. In order to play this role, HHRT includes memory swapping mechanism, which evicts the data on device memory into larger host memory when free space size of the device memory is insufficient. The role of this mechanism is the same as that of OS, but unlike OS's page-wise swapping, we adopt "process-wise" swapping as described later. With this mechanism, we can execute programs like Figure 3 without being annoyed with limitation of device memory capacity. HHRT is implemented as a wrapper library of CUDA and MPI, thus it has the same APIs as CUDA and MPI, except additional ones for performance improvement introduced in Section IV-B.

Figure 8 compares the typical execution model of applications on CUDA and MPI, and that on HHRT. Instead of letting each MPI process occupy a GPU, we let several MPI processes share a GPU. When users execute their application, they have to 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 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.

It is natural that 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) as follows; when a running process $p$ is selected as the victim of swapping out, HHRT suspends its execution and copies contents of all $p$'s memory regions on device memory into the dedicated buffer (swap buffer) on the host memory. Then HHRT frees those memory region on device and make process $p$ "sleeping mode" (now another sleeping process may wake up). Afterwards, when the size of free space in device memory becomes sufficient, the sleeping process $p$ can wake up after restoring contents of device memory regions. With this swapping mechanism, $P_s$ processes share the limited capacity of device memory in a transparent fashion. In Figure 8, two processes are running and other four processes are sleeping per node.

We noticed that we should restrict the timing of swapping out, in order to avoid too frequent swapping in/out, which causes heavy PCI-Express communication. For this objective, each process may be swapped out only when it executes one of the following operations:

- Blocking MPI communication operations such as `MPI_Send, MPI_Wait, MPI_Barrier`
- Device memory allocation such as `cudaMalloc`

According to the design described above, we have implemented a prototype HHRT library. In order to realize swapping mechanism, blocking MPI operations and device memory allocation are hooked so that it checks the free space size of device memory, and execute swapping out operation if needed.

## IV. Writing Temporal Blocking on HHRT

### A. Implementation on top of HHRT

This section describes how programmers write stencil computation that supports larger problem sizes efficiently on top of HHRT. Our goal is not to propose new blocking technique; our motivation is that although temporal blocking is well-known, it is still hard to implement for programmers. Our approach to relax this problem is to harness a runtime library with memory swapping function; thus programmers can focus on locality improvement. While "Hand-TB" (Figure 6) included explicit memory movement and the loop over sub-domains, we delegate such code changes to the HHRT library.

To remove the sub-domain loop from user's code, we simply substitute sub-domains to MPI processes; a each process deals with a single domain that is smaller than device memory size. And we let several MPI processes share a single GPU as is recommended by the execution model of HHRT.

Now data swapping of each sub-domain between host and device is automatically achieved by HHRT's process-wise swapping mechanism. This means that we can execute the "Basic" implementation in Figure 3 on top of HHRT in order to accommodate larger problem sizes [2].

[2]Of course this case exhibits significant low performance due to heavy PCIe communication cost.

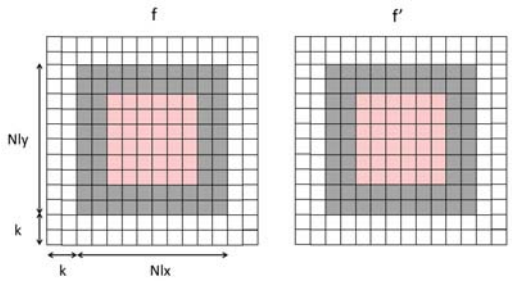


Fig. 9. Data structure that each MPI process maintains in HHRT-TB implementation. Here $k = 2$.

The next step is to implement temporal blocking, in a much simpler fashion than in "Hand-TB". The new implementation is named "HHRT-TB", and its data structure is shown in Figure 9 and the pseudo code is in Figure 10. Here each MPI process maintains arrays, which are not divided into sub-domains. Unlike Figure 2, the halo (process boundary) regions are expanded so that they have the width of $k$, and we call them "$k$-halo" in the pseudo code.

Figure 10 shows the changes compared with "Basic". Modified lines are marked with "$*$", and newly added lines are marked with "$\#$". The following are descriptions of the changes:

- The temporal loop is now doubly nested. The outer starts at line 2 and the inner starts at line 6.
- The area to be computed is configured similarly to "Hand-TB", in order to maintain data dependency (lines 7-8).
- Halo exchange is done for expanded halo with $k$ width (k-halo) at once. And the exchange is done out of the inner loop (lines 3-5).

We do not insist these changes are negligible; however, we consider it is important that the entire code structure is preserved, and programmers only require code rewriting in a local and step-by-step fashion. Contrarily, the hand-coded "Hand-TB" has a largely different structure than "Basic".

Here we discuss why PCIe costs are reduced in the new implementation. Note that due to the swapping rule of HHRT described in Section III, implicit process-wise swapping, involving PCIe costs, happens only when processes execute MPI communication, which appears at line 4 in this code. Thus each process can proceed computation for $k$ time steps without being swapped out.

### B. Optimization with Programmers' Hints

With the implementation described above, we successfully support large problem sizes. However, we found that its PCIe communication cost is about doubly larger than hand written implementation even if we use the same temporal block size $k$. It turned out that this difference is owing to that both implementations are based on *double buffer* technique; where we prepare two arrays for the grid, $f$ and $f'$.

At the point when the inner temporal loop finishes (line 12 in Figure 10), the contents of $f$ are alive while $f'$ can be discarded. In the hand code version, it is natural to evacuate only $f$ from device memory to host memory (line 13 in Figure 6). However, on top of HHRT, HHRT evacuates the contents of all memory regions that the victim process holds, increasing the PCIe communication amount.

```
  1: Copy array f from host to device
     // outer temporal loop
* 2: for t0 in [0, Nt) with stride k {
* 3:   Copy k-halo of f from host to device
* 4:   Exchange k-halo of f by MPI
* 5:   Copy k-halo of f from device to host
       // (a)
       // inner temporal loop
# 6:   for t in [t0, t0+k) {
# 7:     r := k-1-(t-t0) // r changes k-1 ... 0
       // done on GPU: START
* 8:     forall (x, y) in
             [k-r,Nlx+k+r)x[k-r,Nly+k+r) {
  9:       f'[x,y] := func(f[x,y],
                           f[x-1,y], f[x+1,y],
                           f[x,y-1], f[x,y+1])
 10:     }
       // done on GPU: END
 11:     Swap f and f' on GPU
#12:   }
       // (b)
 13: }
 14: Copy array f from device to host
```

Fig. 10. A stencil implementation with temporal blocking technique on top of HHRT (named HHRT-TB). Compared with Figure 3, modified lines are marked with $*$, and newly added lines are marked with $\#$.

In order to improve the swapping cost, we prepared a new API of HHRT so that programmers can supply "hints" as follows.

```
int HH_madvise(void *p, size_t size,
               int kind);
  'kind' is one of following constants:
    HHMADV_NORMAL:
      [p, p+size) is the normal region
    HHMADV_CANDISCARD:
      the runtime can discard the contents
      of [p, p+size)
```

By using this API, programmers can specify the memory regions whose contents can be discarded by HHRT. We can improve the HHRT-TB implementation (Figure 10) by adding two lines:

```
HH_madvise(f', end of f', HHMADV_NORMAL),
HH_madvise(f', end of f', HHMADV_CANDISCARD)
```

at the points (a) and (b), respectively (we call the resultant version "HHRT-TB-Hint").

With this information, we can reduce the costs of swapping. This `HH_madvise` API is inspired by `madvise` Linux systemcall, however, `HH_madvise` allows more intensive optimization since the current specification of `madvise` does not allow the disposal of memory contents. The effects on performance of this optimization are evaluated in the next section.

TABLE I. SYSTEM SOFTWARE USED ON TSUBAME2.5

| | |
|---|---|
| OS | SUSE Linux 11 sp1 |
| Compiler | gcc 4.3.4 |
| MPI | OpenMPI 1.6.3 |
| CUDA | 5.5 |

## V. PERFORMANCE EVALUATION

### A. Evaluation Conditions

For the performance evaluation, we used the TSUBAME2.5 petascale supercomputer installed in Tokyo Institute of Technology. The system mainly consists of 1,408 HP Proliant SL390s compute nodes, each of which is equipped with two Intel Xeon X5670 CPUs (6core, 2.93GHz) and three NVIDIA K20X GPUs [3]. Each K20X GPU consists of 2,688 CUDA cores (732MHz), which are enclosed in 14 SMXs (Streaming Multiprocessor eXtreme), and the peak performance is 3.95TFlops in single precision and 1.31TFlops in double precision. CUDA cores share on-board device memory with 6GB capacity and 250GB/s bandwidth. CPUs and GPUs are connected via PCI-Express 2.0 x16 bus, whose theoretical bandwidth is 8GB/s, about 30 times slower than device memory bandwidth. Each node is equipped with 54GB or 96GB host memory, which is 9x to 16x larger than device memory of a GPU. The list of system software is shown in Table I.

As stencil benchmarks, we implemented several versions of a simple 7-point stencil programs. The computed grids are three-dimensional arrays, whose elements have float data type. In the following evaluations, the grid shapes are regular cubes, which are decomposed in two-dimension in multi-process cases. The problem sizes are computed as $2 \times n^3 \times sizeof(float)$ (bytes), where $n$ is the length of cube edges. Here we include space for double buffering but not for extra halo region introduced for temporal blocking. Also we do not count the redundant computation to obtain the performance number in GFlops.

### B. Evaluation on a Single GPU

We compare the performance of the following stencil programs on a single GPU on a TSUBAME node with 96GB host memory:

- **Hand-TB**: Temporal blocking is hand-coded as in Figure 6.
- **Hand-TB-Opt**: Based on Hand-TB, and extensive optimization techniques, such as redundant computation elimination, are implemented. The implementation is even more complex than Figure 6. For details, refer to [5].
- **HHRT-Basic**: The "Basic" implementation (Figure 3) is simply executed on top of HHRT.
- **HHRT-TB**: Temporal blocking is implemented as in Figure 10 and executed on HHRT.
- **HHRT-TB-Hint**: Based on HHRT-TB, and programmers' hints are used as in Section IV-B.

[3]In this paper, one GPU per node is used.

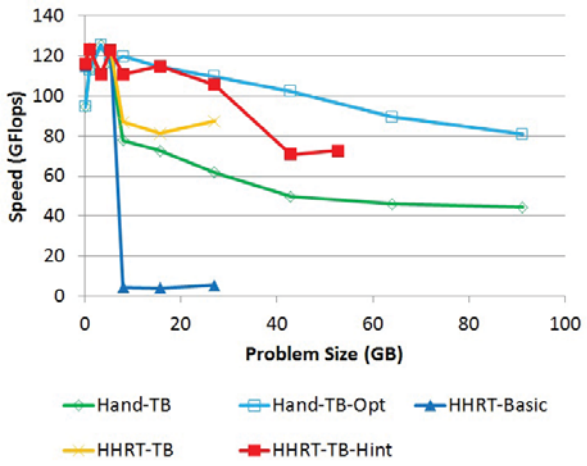


Fig. 11. Performance evaluation with various problem sizes on a single GPU on a node with 96GB host memory.

The graphs in Figure 11 shows relationship between problem sizes (in GB) and performance (in GFlops). In each case, execution parameters, temporal blocking size $k$ and the number of processes sharing a GPU $P_s$ are configured so that we can obtain the best performance.

We observe that the performance of "HHRT-Basic" largely drops when the problem size exceeds the device memory capacity of 6GB. With other four implementations with temporal blocking, the performance drop is dramatically mitigated. We observe HHRT-TB and HHRT-TB-Hint exhibits better performance than hand-coded Hand-TB, although programming cost is reduced. We found that this gap is related to overlapping between computation and communication. Overlapping is not explicitly coded in all of Hand-TB, HHRT-TB and HHRT. However, with HHRT library, the overlapping is automatically realized since memory swapping of a process can be done simultaneously while another process is running. Comparing HHRT-TB and HHRT-TB-Hint, the latter is 21 to 42% faster.

We see the performance of hand-optimized Hand-TB-Opt surpasses two versions on HHRT, and the gap between HHRT-TB-Hint and Hand-TB-Opt reaches up to 31%. In order to shrink this gap, instrumenting advanced optimizations such as redundancy reducing on HHRT is included in our future work.

Figure 11 also shows that versions on HHRT still have limitations in problem sizes. In hand written versions, where all the memory regions are maintained by programmers, we can support problem sizes around 90GB on 96GB memory node. On the other hand, since HHRT itself consumes host memory for its swapping buffers, apart from programmers' region, the affordable problem sizes are reduced. Comparing HHRT-TB and HHRT-TB-Hint, programmers' hints are also helpful for enlarging problem sizes, since we can reduce the total size of required swapping buffers. The maximum problem size in HHRT-TB-Hint is around 53GB, which is 55% of the host memory capacity. We are planning to improve this point as future work; a possible approach would be to use Flash memory or memory on other compute nodes in order to hold swap buffers.

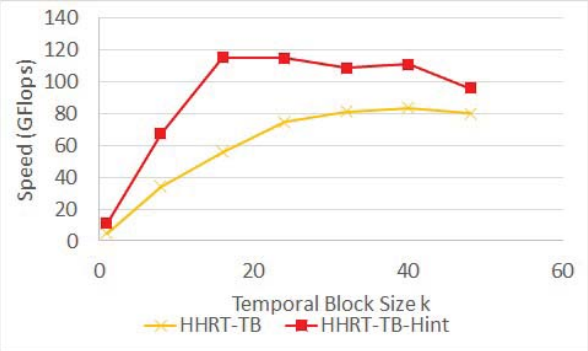


Fig. 12. Impacts of the temporal block size $k$ on performance.

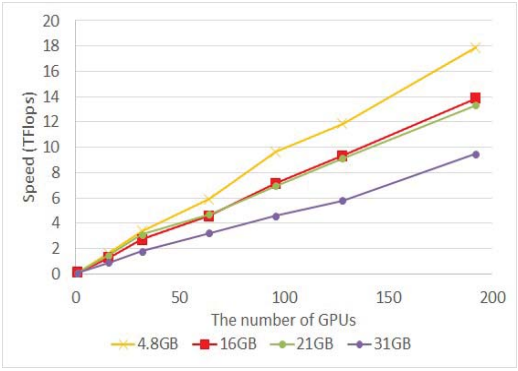


Fig. 13. Weak scability of our implementation on TSUBAME2.5 up to 192GPUs.

### C. Impacts of Temporal Block Size

Generally, when we introduce temporal blocking technique, we have to carefully tune execution parameters, especially the temporal block size $k$. If $k$ is too small, the performance suffers from costs for frequent data movement. If $k$ is too large, the computational cost increases for redundant computation.

Figure 12 shows the performance of HHRT-TB and HHRT-TB-Hint with various $k$, with the problem size of 16GB. We observe the tradeoff described above with both versions, but the "optimal" $k$ differs; it is 40 in HHRT-TB and 16 in HHRT-TB-Hint. The performance with smaller $k$ suggests this difference comes from the difference in impact of data movement cost, which is twice larger in HHRT-TB. With larger $k$, we see the performances of two versions get closer, since the performance is largely determined by computational cost, which is common in both versions.

### D. Evaluation on Multiple Nodes

This section evaluates the performance of our implementation on HHRT by using multiple nodes. Here we used compute nodes with 54GB host memory [4], and one GPU on each node. Figure 13 demonstrates weak scalability of the HHRT-TB-Hint implementation. The graph shows four cases and legends correspond to the problem sizes per GPU. In the smallest case (4.8GB), no swapping out occurs. In other cases, we observe we suffer from 22 to 47% overhead, due to memory swapping costs. In spite of this overhead, the graph shows that our implementation exhibits good scalability even with swapping. When the problem size reaches 31GB per GPU (5.95TB in total), we achieve 9.5TFlops. This result indicates that HHRT enables users to execute simulations with problem sizes 5.2 times larger than available device memory capacity with reasonable overhead.

## VI. Related Work

Locality improvement of time iterative simulations is one of recent hot topics against issues of memory wall, which will be getting higher towards exascale era. In this context, temporal blocking for stencil computations has been proposed and implemented in various computer architectures [7], [8]. While most previous papers have focused on improving cache hit ratio, our objective is to reduce PCI-Express communication cost between GPUs and CPUs. While the basic concept is common, this difference heavily impacts on parameter tuning; while the temporal block size for cache tuning is typically 2 to 8, we have shown that best parameter in our case is 16 to 40. Like our work, Mattes et al. has focused on locality improvement on GPU machines[9]. Although we skipped the detail, our previous paper described multi-level temporal blocking, which reduces both device memory traffic and PCI-Express traffic[5], [6]. Anyway in previous papers, programming costs for implementing temporal blocking have been merely mentioned.

In order to write stencil applications, we could use domain specific framework, instead of using MPI and CUDA directly. Such an example is the Physis stencil framework [18]. With this framework, programmers write stencil applications in for-all style. After programmers write a code fragment for a single grid point update, the framework produces the iteration code, communication code and supports for GPU automatically. This approach is highly promising since it would be possible to implement optimization techniques including temporal blocking in a user-transparent style. On the other hand, if users already have application codes written in MPI and CUDA [1], [2], [3], [4], they have to completely 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.

Another approach for efficient applications execution on GPU systems is to use fine grained task scheduler such as StarPU[13], PaRSEC[14] or PULSAR[15]. These systems assume that applications are described in direct acyclic graphs (DAGs) that consist of fine grained tasks, which are targets of scheduling, and relationship among tasks. Their assumptions differ from ours, where the main targets are existing codes written in MPI and CUDA.

The design of our HHRT library is largely inspired by Adaptive MPI [16], implemented on top of CHARM++ [17]. Adaptive MPI and HHRT have a common point in the execution model; several MPI processes share the limited resources on computer systems. However, the objective is different since Adaptive MPI's main target is dynamic load balancing between compute nodes, while our target is to use limited capacity of GPU device memory efficiently. Also, Adaptive MPI itself is

[4]Although single GPU evaluation used a 96GB node, the system does not have sufficient number of 96GB nodes for this evaluation

not for GPU clusters.

Recently NVIDA has released CUDA version 6 with new mechanism called Unified Memory[10]. With Unified Memory, programmers can allocate memory regions whose contents are moved between device memory and host memory transparently. This has a similar effect as the swapping mechanism of HHRT. However, the current Unified Memory does not compromise the necessity of HHRT, since it does not support either regions larger than device memory, or arbitration between processes sharing a GPU, which HHRT supports. When such limitations are relaxed in future CUDA version, we will reconsider the design of HHRT.

## VII. Conclusion

We have demonstrated that we can execute stencil computations on GPU clusters, while maintaining high performance, large problem sizes, and low programming costs. The key is combining temporal blocking technique, which is efficient locality improvement optimization, and underlying software layer that supports memory swapping between host and device transparently.

Through the performance evaluation on the TSUBAME2.5 GPGPU supercomputer, we can support problem size 9 times larger than GPU device memory with moderate overhead. The resultant implementation is also highly scalable with a large number of GPUs; it achieved 13.9TFlops with 192GPUs for the problem size of 3TB, which is 2.6 times larger than the total capacity of used GPUs. Also programmers' hints that specify the liveness of memory regions are helpful to improve performance and problem size.

As future work, it will be challenging to implement advanced optimization techniques, including redundancy reduction, while keeping the programming costs low. We could support even larger problem sizes than host memory, if computers are equipped with fast flash memory devices like FusionIO ioDrive with $>$ 1GB/s bandwidth [19]. Also we are planning to demonstrate the feasibility of our approach on Xeon Phi clusters.

The basic concept of our approach is to harness the high bandwidth of device memory and the large capacity of host memory. This concept is not flashy or temporal one for non-ordinary computers, but general and important one against the memory wall problem. In order to expand bandwidth of memory devices towards future supercomputers, 3D-stacked memory architecture is receiving high attention. However, it will be difficult to achieve both high bandwidth and large capacity with monolithic memory architecture, and we will need heterogeneous memory as is the case in the accelerated hybrid clusters. We will continue to investigate locality improvement and communication reducing techniques on changeful supercomputer architectures towards the exascale era.

**Acknowledgements**

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

## References

[1] Everett H. Phillips and Massimiliano Fatica: Implementing the Himeno Benchmark with CUDA on GPU Clusters, IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 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, 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, IEEE/ACM Supercomputing (SC11), pp. 1-12, Seattle (2011).

[4] 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 Supercomputing (SC11), pp. 1-11, Seattle (2011).

[5] Guanghao Jin, Toshio Endo and Satoshi Matsuoka: A Multi-level Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPU. The Third International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), in conjuncition with IEEE IPDPS2013, pp.1080-1087 (2013).

[6] Guanghao Jin, Toshio Endo and Satoshi Matsuoka: A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPUs . In Proceedings of IEEE Cluster Computing (CLUSTER2013), pp. 1–8, (2013).

[7] 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).

[8] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In Proceedings of IEEE/IEEE Supercomputing (SC10), 13pages (2010).

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

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

[11] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, `http://www.netlib.org/benchmark/hpl/`

[12] C. Ding, Y. He: A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems, IEEE/ACM Supercomputing (SC01), Denver (2001).

[13] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation. Practice and Experience, Special Issue: Euro-Par 2009, **23** pp. 187–198 (2011).

[14] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, J. Dongarra: PaRSEC: Exploiting Heterogeneity to Enhance Scalability. IEEE Computing in Science and Engineering, Vol. 15, No. 6, 36-45 (2013).

[15] PULSAR: Parallel Ultra Light Systolic Array Runtime. `http://icl.cs.utk.edu/pulsar/`

[16] 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).

[17] 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).

[18] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, IEEE/ACM Supercomputing (SC11), pp. 1-12, Seattle (2011).

[19] Fusion-io ioDrive2, `http://www.fusionio.com/products/iodrive2`