Applying Recursive Temporal Blocking for Stencil Computations to Deeper Memory Hierarchy

Toshio Endo
GSIC, Tokyo Institute of Technology /
RWBC-OIL, AIST
Email: *endo@is.titech.ac.jp*

***Abstract*—Recent high performance computer architecture has deeper memory hierarchy including 3D stacking memory and non-volatile memory. In order to achieve higher application performance, optimizations in application algorithm level are required. This paper takes stencil computations as the target application, and focus on a technique called temporal blocking. In order to consider multiple hierarchy, we apply recursive temporal blocking alorithm. By using it, high performance is obtained without hand tuning considering memory architecture parameter. The evaluation of this approach is done on a server equipped with GPU device memory of HBM2, DDR4 host memory and 3D-XPoint based SSD.**

I. Introduction

Access performance and capacity of memory systems have strong direct impacts on performance and scales of simulations in weather, medical and disaster measurement area. However, it is hard for a single memory layer to achieve both since the improvement of capacity and/or bandwidth of memory is slower than that of processors [1]. For example, current high-end GPGPU achieves memory bandwidth of near 1TB/s with 3D stacked DRAM technique, however, memory capacity per accelerator is limited to 8 to 32 GiB, which is much smaller than that of DDR memory of typical high performance servers as shown in Figure 1. Conversely, recent non-volatile devices with Flash or 3D-XPoint technology [2] have around 1TB capacity. On the other hand, they are implemented as disk devices, and even with high performance devices, access performance is limited to a few GB/s due to PCI-express bus.

Our targe applications are *stencil computation kernels*, which are a class of computations frequently used in simulations with time evolution including fluid dynamics simulations, structure analysis of materials and so on. They are memory bandwidth centric, and thus many stencil-based applications have higher speed performance on general purpose GPU (GPGPU) clusters[3], [4], [5], [6] than on general CPU clusters. However, the problem sizes have been limited by capacity of GPU device memory.

In order to realize extremely fast and large scale simulations, we need approaches to harness deeper memory hierarchy; high performance of upper memory layer and large capacity of lower layer should be exploited. In order to achieve that, memory access locality of the algorithm should be considered, and a technique called *temporal blocking* [7], [8], [9], [10], [11], [12] have been explored. With temporal blocking, the simulated area is divided into blocks both in temporal dimension and spatial dimensions to improve memory access locality. We have demonstrated this technique is useful to improve stencil performance in "out-of-core" scales while harnessing high performance of GPUs[13], [14].

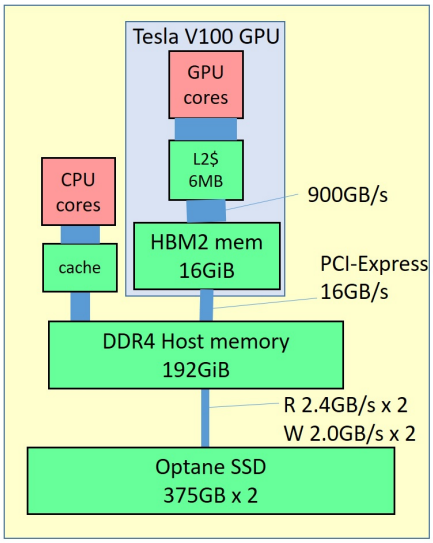

Fig. 1. Memory hierarchy of an exemplar GPGPU machine from the viewpoint of GPU cores. Here 3D-XPoint based Optane SSDs with bandwidth of >2GB/s are equipped. This figure illustrates a compute node used in our performance evaluation.

While temporal blocking is useful, it requires parameter tuning of temporal/spatial block sizes, which considers property of memory architecture. The costs of tuning get even higher if we take deeper memory hierarchy into consider, in order to configure block sizes for each memory layer [15], [16]. If the implementation of blocking is programmed manually, programming costs also get higher.

An approach that does not require architecture aware parameters is the *cache oblivious stencil* algorithm [17]. Here the simulated region that consists of temporal dimension and spatial dimensions are divided recursively. While the algorithm was originally designd for cache access improvement, we ex-

pect this approach effectively supports deep memory hierarchy because we do not have to specify parameters for different layers.

The objectives of this paper are as follows. We implement a stencil kernel with recursive temporal blocking algorithm for memory systems that consists of HBM2 device memory, DDR4 host memory and 3D-XPoint SSDs. Through the performance evaluation using this implementation, the recursive approach has advantages in speed performance over non-recursive (called single temporal blocking in this paper) on deep memory hierarchy. Also we discuss that configuration of a threshold to finish recursive call affects cache locality.

The current implementation is a simple 3-dimensional 7-point stencil, we expect that this approach will pave the road towards extreme large-scale and high-performance simulations on deep memory hierarchy with non-volatile memory.

II. Stencil Computations and Temporal Blocking

A. Stencil Computations

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 "three-point" stencil computation on one dimensional grids[1]. The simplified algorithm is as follows.

```
for (t = 0; t < nt; t++)
  for (x = 1; x < nx-1; x++)
    f[(t+1)%2][x] := c1 * f[(t%2)][x-1] +
                     c2 * f[(t%2)][x] +
                     c3 * f[(t%2)][x+1]
```

Here a technique known as double buffering used; f[0] represents simulation data for "even" time steps, and f[1] corresponds to odd time steps. Update of a single point at x requires data of previous points at $x-1$, x and $x+1$ as shown in Figure 2, introducing neighbor dependency.

Ovbiously the above implementation has less memory access locality, since the entire arrays are traversed for every time step. Thus when the total size of arrays exceed the capacity of upper memory layer (GPU device memory or cache memory), it suffers from heavy data movement.

B. Temporal Blocking

In order to improve memory access locality, a technique known as temporal blocking (or time-space tiling) has been proposed [7], [8], [9], [10], [11], [12]. With temporal blocking, we divide the arrays into blocks in space dimensions. Then we execute the computation of a single block for several (k) time steps at once, which improves locality. Here k is called temporal block size. The basic idea is simple, we need to take care of neighbor dependency. In order to preserve the dependency, we cannot use "rectangle" block shape. Instead, various shapes have been proposed, such as parallelogram shape and trapezoid shape [11] as shown in Figure 3. In the figure, the numbers in blocks represents possible execution order.

[1]though we will use "3-dimensional seven-point" stencil on three dimensional grids in the evaluation

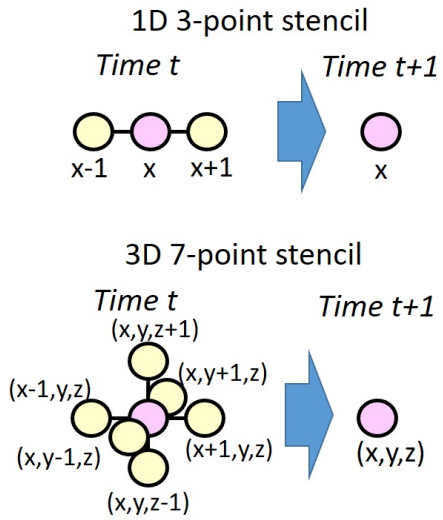

Fig. 2. A single point update in 1-dimensional 3-point stencil and 3-dimensional 7-point stencil.

Between these two shapes, parallelogram shape tends to introduce longer dependency chain among blocks, and thus trapezoid shape has advantages for scalable parallel executions. This paper mainly focuses on blocking with trapezoid shape.

Roughly speaking, when block size is well tuned, access amount to lower memory layer is reduced to $1/k$ of the base case without blocking. Thus larger k is desirable for speed performance, however, we cannot choose infinitely large k for the following reason. In order to compute a single block efficiently, the memory footprint size (w in the figure) should not exceed the upper memory capacity. Due to the property of shapes shown in the figure, the following relations must hold: $w > k$ in parallelogram shape and $w > 2k$ in trapezoid shape.

In order to support deep memory hierarchy efficiently, several researchers have tried blocking with mutiple levels [15], [16]. However, this approach requires tuning of block sizes for each level considering capacity of each memory layer.

C. Recursive Temporal Blocking

An approach that does not require architecture aware parameters is the *cache oblivious stencil* algorithm described by Frigo et al [17], called recursive temopral blocking hereafter. Here the region to be computed are divided recursively both in spatial dimension and temporal dimension.

The algorithm starts with the entire block, which is represented as a product space of spatial area and temporal range to be computed. The algorithm divides current target block

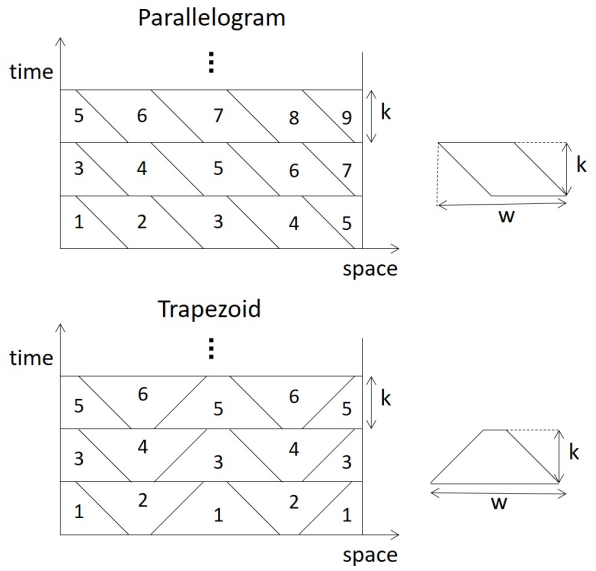

Fig. 3. Examples of block shapes in temporal blocking (one-dimensional space).

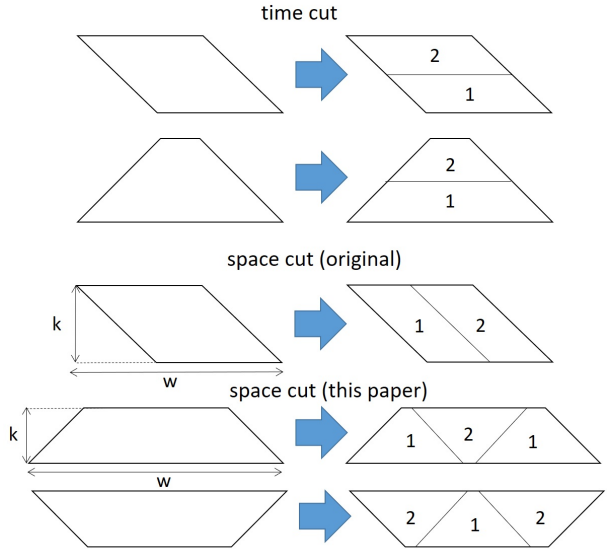

Fig. 4. Block division methods in recursive temporal blocking.

by using either ”time cut” or ”space cut” illustrated in Figure 4. If the footprint of the current block is sufficiently larger than block height (for example $w > 2k$), the block is ”space cut”. Otherwise, the block is ”time cut”. The divisions are recursively repeated until the algorithm reaches a leaf case. In Frigo’s work, a leaf case corresnponds to a block with height k of 1.

In this paper, we prefer trapezoid shapes that support parallelism among blocks. For this direction, we slightly modified space cut method as shown in the lower part of Figure 4. Here space cut is possible if $w > 4k$.

III. Implementation

This section describes our prototype implementation of stencil code with recursive temporal blocking for deep memory hierarchy. The code computes 3-dimensional (x, y and z) space with 7-point stencil, where a single point update requires neighbor 7 points in the previous time step as illustrated in the lower figure in Figure 2. Each array element has float data type (single precision). For the simplicity of implementation, we currently divide blocks only in z-dimension and t (time) dimension. Our target system architecture is a comput node equipped with NVIDIA GPU and Intel 3D-XPoint based SSDs shown in Figure 1.

A. Implementation of Recursive Algorithm

In the original Frigo’s algorithm, the recursive partition is repeated until height of a target block (the block size in temporal dimension) becomes 1. However, we found that this causes too small partition, which introduces higher costs for recursive function call. Also if a block is too small, many cores on a GPU are not efficiently utilized. Instead, in our algorithm, recursive partition stops if the footprint of the current block is smaller than a pre-defined threshold. In this aspect, our method is not parameter-tuning free. However, we consider configuring a single threshold is still much easier than tuning block sizes for each memory layer. The effects of varying the threshold are evalulated in Section IV.

The computation kernel on the leaf case is implemented using NVIDIA CUDA programming environment [18]. A compute kernel running on the GPU takes two pointers to partial arrays for double buffering, and updates all inner points. In the compute kernel, CUDA threads are utilized to harness massive number of CUDA cores in a GPU. Also threads are aligned so that they can perform coalesced memory accesses.

B. Management of Memory Hierarchy

Our main target is out-of-core cases, where the total size of arrays exceeds both GPU memory capacity and host memory capacity. We need to seek for a memory management method, which can support out-of-core cases. For this objective, we surveyed automatic memory management methods provided by NVIDIA and Intel. NVIDIA Unified memory mechanism [18] provides automatic data movement functionality between device memory and host memory, while Intel Memory Drive Technology (IMDT) transparently expand capacity of host memory by harnessing capacity of 3D-XPoint based Optane SSDs. If these two worked simultaneously, we could enjoy automatic data transfer among device memory, host memory and SSDs. Unfortunately, we found that the machine may crash when Unified memory and IMDT are used in the current system software (see Table I).

To avoid this issue, our current implementation uses only IMDT, while data movement between device memory and host memory is done by manual coding. Each of two arrays for double buffering is allocated by `malloc`, which may be larger than capacity of physical host memory. In the recursive algorithm, if footprint of the current target block is

TABLE I
THE COMPUTE NODE USED FOR EVALUATION

GPU	NVIDIA Tesla V100
SP peak perf. (TFlops)	15.7
Device memory BW (GB/s)	900
Device memory size (GiB)	16
# of GPUs/node	2 (1 used)
CPU	Intel Xeon Gold 6140
# of CPUs/node	2
CPU-GPU connection	PCIe gen3 x16
Peak BW (GB/s)	16+16
Host memory size (GiB)	192
SSD	Intel Optane SSD DC P4800X
Size (GB)	375
Read BW (GB/s)	2.4
Write BW (GB/s)	2.0
Read latency (us)	10
Write latency (us)	10
# of SSDs/node	2
OS	CentOS 7.3
System software	CUDA 9.1
	NVIDIA driver 390.30
	IMDT 8.5.1955

smaller than device memory, contents of the block are copied (swapped-in) to device memory explicitly using CUDA APIs (`cudaMemcpy`). Note that recursive partitioning may repeated even after swapping-in, until each block gets smaller than the abovementioned threshold.

C. Current Limitations

The implementation described so far realizes stencil computation that supports larger scale than host memory. The current prototype implementation has the following limitations, which will be improved in the future.

- While we use 3-dimensional arrays, they are divided only in z-dimension. Frigo et al. have described multi-dimensional recursive partition, which would improve access locality.
- Computation and data movement between device and host are not overlapped. Since our implementation is based on trapezoid shape that can coexist with inter-block parallelism, this can be improved relatively easily.
- Only a single GPU is utilized for computation. Thus while we use intra-block parallelism by CUDA threads, inter-block parallelism is not utilized currently.

IV. PERFORMANCE EVALUATION

A. Evaluation Conditions

Our performance evaluation has been conducted on a GPGPU node shown in Table I. Its memory hierarchy has been shown in Figure 1. The device memory capacity of 16GiB and the host memory capacity of 192GiB. In this paper, a single GPU is used for computation. The node is equipped with two Optane SSDs, which are used automatically as host memory expansion by IMDT.

In the next subsection, the following algorithms are compared.

- *Base* means the base stencil algorithm without temporal blocking.
- *Single (Sxx)* means the stencil with temporal blocking with a single block size, as shown in "Trapezoid" in Figure 3. The spatial block size is set so that the footprint of a block fits the device memory capacity. Several temporal block sizes k are compared.
- *Recursice (Rxx)* means the recursive temporal blocking. We configure several thresholds to stop recursive processing.

For each condition, we compare execution speed in "the number of updated points per second", which is denoted as "GUP/s" (giga updated points per second) later.

B. Evaluation

The graphs in Figure 5 show the performance of the stencil program for several array sizes. In the first graph, the simulated space size is $1024 \times 1024 \times 1024$, and the total size of double arrays is $1024 \times 1024 \times 1024 \times$ sizeof(float) $\times 2 =$ 8GiB, which corresponds to "in-core" case since it is smaller than GPU device memory of 16GiB. In this condition, we observe the Base case achieves 30.5GUP/s. Here the effective bandwidth is $30.5 \times 10^9 \times$ sizeof(float) $\times 2 = 244$GB/s. "Single (Sxx)" algorithm is the same as "Base" in in-core case, and we see it shows the same performance. On the other hand, "Recursive (Rxx)" achieves better performance, because recursive algorithm can harness GPU cache effectively. The speed depends on the threshold, and it is 67.7GUP/s with 64MB threshold, which is 2.22times fater than "Base" or "Single".

The second graph shows computation of $2048 \times 2048 \times 2048$ space, where total array size is 64GiB. It is larger than GPU device memory and smaller than physical host memory. The performance of Base case is critically low; only 0.36GUP/s. We observe temporal blocking is useful for performance improvement, and S128 achieves 20.1GUP/s. The performance is higher with larger temporal block size, however, the execution failed with block size larger than 128. The recursive algorithm shows improved performance and reaches 23.0GUP/s with 256MB threshold. These speeds will be improved if limitations on the current implementation described in Section III are solved. Especially, the execution time of 60% or more are consumed by data copy between device memory and host memory, even in R128MB, optimizations in data movement will have large impacts. We also observe the sensitivity of varying threshold weaker than in 8GB scale. The reason will be investigated in future.

The third graph shows large computation of $2048 \times 2048 \times 8192$ space, where total array size is 256GiB, which is larger than physical host memory. Here the program suffers from access costs to Optane SSD via IMDT. Thus the performance of "Base" case, 0.24GUP/s, is even slower than above. "Single" temporal blocking improves the performance, however, the highest value is 12.1GUP/s with block size of 96. With "S128", the execution failed since it failed to make block shapes with $k = 128$. We observe "Recursive" algorithm

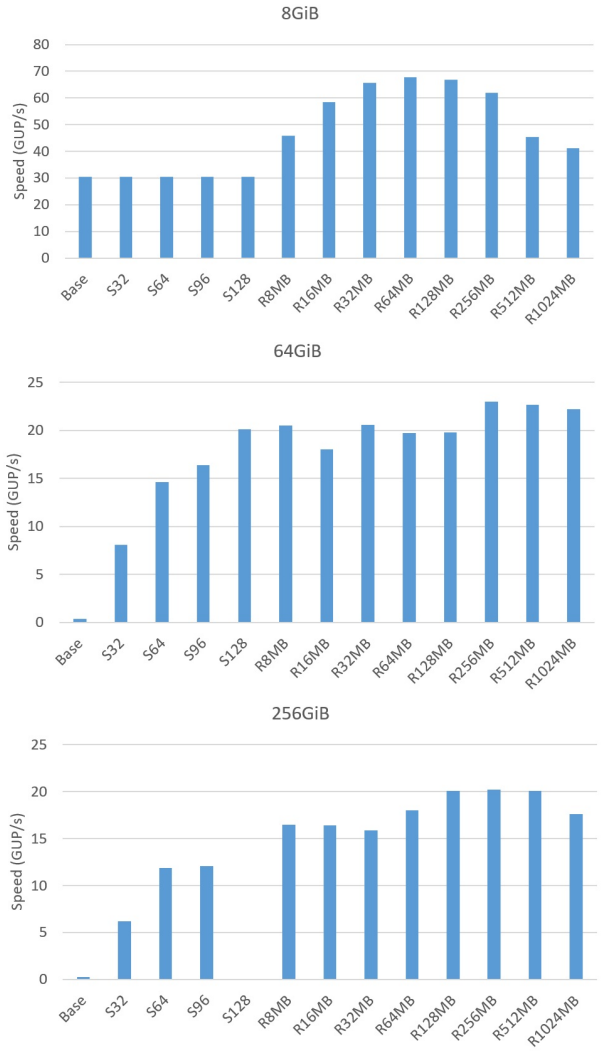

Fig. 5. Performance evaluation with various problem sizes. Each graph compares "Base" algorithm without temporal blocking, "Single (Sxx)" temporal blocking with various temporal block sizes and "Recursive (Rxx)" temporal blocking with various thresholds.

achieves 20.2GUP/s with 256MB threshold, which is 1.67 times faster than S128. The performance gap between "Recursive" and "Single" is larger than in 64GB scale, which demonstrates that recursive algorithm has advantages with deeper memory hierarchy including non-volatile memory with near terabytes capacity.

V. Related Work

Non volatile 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 [2]. On the other hand, this work uses NVM devices in order to expand available capacity of host memory and GPU device memory for scientific applications.

Temporal blocking for stencil computations have a long history and have been implemented in various computer architectures [7], [8], [9]. 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[10]. While based on these results, our objective is to support multi-tier memory hierarchy, GPU device memory, host memory and NVM, while reducing swapping costs.

Adapting stencil computation to multi-tier hierarchy has been done by Midorikawa et al.[15] implemented and evaluated out-of-core stencil computations that consider CPU cache, host memory and flash devices. This approach requires block sizes tuning for each memory layer. Thus they also implemented auto-tuning mechanisms considering memory architecture parameters. On the other hand, one of our objectives is to minimize number of parameters to be tuned.

VI. Conclusion

This paper has discussed performance of huge scale stencil computations in order to assess the applicability of 3D-XPoint based NVM devices to scientific simulation applications. In order to harness deep memory hierarchy efficiently, improvement of memory access locality, temporal blocking technique in our context, is mandatory. Especially recursive temporal blocking algorithm well fits deep memory hierarchy that consists of GPU device memory, host memory and NVM devices. The recursive approach achieves 1.67 times better performance than a simpler temporal blocking algorithm in "out-of-core" case.

In addition to improvements of the current prototype implementation, there are several directions to be explored in future.

- To integrate the locality improvement technique into stencil execution frameworks/DSLs [19], [20], [21] or polyhedral compiler tools [22], [23]. These integration will be required for real simulation applications with large code bases to realize extremely large and fast executions.
- This work used 3D-XPoint based SSD devices, which are essentially block devices. In near future, byte-addressable NVM devices will be available; we are going to assess and evaluate the perforamcne impact of recursive blocking approach on such new devices.

Acknowledgements

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

References

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

[2] Intel Corporation: Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

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

[3] 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).

[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 Supercomputing (SC11), pp. 1-12, Seattle (2011).

[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, IEEE/ACM Supercomputing (SC11), pp. 1-11, Seattle (2011).

[6] 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).

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

[8] 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).

[9] 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).

[10] 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).

[11] David Wonnacott and Michelle Strout: On the Scalability of Loop Tiling Techniques. In Proceedings of International Workshop on Polyhedral Compilation Techniques (IMPACT 2013), 9pages (2013).

[12] Takeshi Fukaya,Takeshi Iwashita: Time-space tiling with tile-level parallelism for the 3D FDTD method. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2018), pp. 116-126 (2018).

[13] Toshio Endo, Yuki Takasaki and Satoshi Matsuoka: Realizing Extremely Large-Scale Stencil Applications on GPU Supercomputers. In Proceedings of The 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015), pp. 625-632 (2015).

[14] Toshio Endo: Realizing Out-of-Core Stencil Computations using Multi-Tier Memory Hierarchy on GPGPU Clusters. In Proceedings of IEEE Cluster Computing (CLUSTER2016), pp. 21-29 (2016).

[15] Hiroko Midorikawa: Blk-Tune: Blocking Parameter Auto-Tuning to Minimize Input-Output Traffic for Flash-based Out-of-Core Stencil Computations. International Workshop on Automatic Performance Tuning iWAPT2016, In Proceedings of IEE IPDPSW2016, pp. 1516-1526 (2016).

[16] 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).

[17] Matteo Frigo and Volker Strumpen: Cache Oblivious Stencil Computations. In Poceedings of ACM International Conference on Supercomputing (ICS'05), pp. 361-366 (2005).

[18] NVIDIA Corporation: `https://developer.nvidia.com/cuda-toolkit`.

[19] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), pp. 1-12 (2011).

[20] Takashi Shimokawabe, Takayuki Aoki and Naoyuki Onodera: High-productivity Framework on GPU-rich Supercomputers for Operational Weather Prediction Code ASUCA, The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC14), pp. 251-261 (2014).

[21] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, Thomas C. Schulthess: STELLA: A domain-specific tool for structured grid methods in weather and climate models. The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), Article No. 41 (2015).

[22] Tobias Grosser, Armin Groesslinger, Christian Lengauer: Polly - Performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters, 22:04 (2012).

[23] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gomez, C. Tenllado. F. Catthoor: Polyhedral parallel code generation for CUDA. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 9, No. 4, Article No. 54 (2013).