論文 / 著書情報
Article / Book Information

題目(和文)	GPUクラスタにおけるメモリ階層を考慮した時間依存性があるアルゴリズムのための最適化手法
Title(English)	Optimization Methods for Efficient Utilization of Memory Hierarchy for Algorithms with Temporal Dependences on GPU Clusters
著者(和文)	金光浩
Author(English)	GUANGHAO JIN
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第9423号, 授与年月日:2014年3月26日, 学位の種別:課程博士, 審査員:松岡　聡,青木　尊之,遠藤　敏夫,増原　英彦,渡辺　治
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第9423号, Conferred date:2014/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Optimization Methods for Efficient Utilization of Memory Hierarchy for Algorithms with Temporal Dependences on GPU Clusters

Guanghao Jin

Department of Mathematical an Computing Sciences

Tokyo Institute of Technology

A thesis submitted for the degree of

Doctor of Philosophy (PhD)

2014 February

Acknowledgements

I am very grateful to Prof. Satoshi Matsuoka for being my academic advisor and his continuous support, suggestion and comment. He encouraged and supported me through the period of my research; I wouldn't be able to write this thesis without his help and guidance. As a top researcher in supercomputing field, my supervisor helped me build up my whole view of this field. Furthermore, he supported me for many research activities that include attendances of international and domestic conferences. The research environment and devices which are essential to my research are also supported by him. When I met difficulty, he was the first person that gives me support and suggestion. He is also the person who I want to be in the future because of his professional attitude and passion. It was a great honor to be his student and I hope to retain this kind of relationship in future which can benefit my life.

I also want to give grateful thanks to Prof. Toshio Endo who supervised me about my research. He gave me endless support which made me possible to achieve success in research and continue the study in Tokyo Institute of Technology. His financial support was essential to me. His guidance was the key point of my achievement. He gave suggestion and comment to my papers and presentations which made me possible to be a real researcher in the supercomputing field. My research started from his great idea and he gave me right direction. With his continuous supervision, I can see my improvement day by day. He is the spiritual leader of mine and I will follow his direction which will make me to be a real researcher in future.

I also want to express my greatest gratitude to the people who had helped me during my study in Tokyo institute of technology. All members of Matsuoka lab had given me support and suggestion during my study here. I will remember all support from them and all happy time with them. They gave me suggestion and comment during seminars and rehearsals which made me more confident at the presentation of conference. I also learned knowledge of programming and technology of supercomputer from the members which were essential to implement of my project. It was a great honor to work with them and I hope to keep this relationship in future.

Contents

Chapter 1. Introduction

1.1 Motivation of this thesis

Graphics Processing Units (GPUs) which also called visual processing unit are very efficient at manipulating computer graphics which was popularized by NVIDIA in 1999. Although GPUs have been developed for rendering computer graphics, since Compute Unified Device Architecture (CUDA) was released by NVIDIA as a GPGPU programming framework in 2006, programming GPUs for scientific computing has been made possible without knowing graphics-oriented APIs. GPGPU is well-designed to solve problems that can be expressed as data-parallel computations [1]. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. GPGPU has emerged as a high-performance computing device to accelerate a wide variety of applications and become an active research area in parallel computing. Heterogeneous computing with both conventional CPUs and vector-oriented GPU accelerators is becoming common because of superior raw performance as well as power-performance efficiencies [2]. In many CPU-GPU architectures where GPU acts as a subsidiary device of CPU, the memory capacity of GPU is smaller than that of CPU. Commonly, the size of computational domain is limited by the memory capacity of GPUs. In the real scientific simulations, it needs to increase the computational domain for these reasons. The first reason is that bigger computational domain means bigger computational area at the same accuracy which is important to simulations like weather forecast. In those simulations, bigger computational area means capability to predict wider range of area which is important to protection from environment disasters. The second reason is that bigger computational domain means higher accuracy at the same computational area if it use like stencil method. More accuracy can show more details of the simulations which make it possible to find more details and uncover some new field of research. The third reason is that some scientific simulations are constituted of multiple processes that consume the system resource. It is the key challenge that how to extend the limitation of GPU memory capacity in those cases.

For all those reasons, we need to solve the contradiction between the demand of bigger computational domain and the limitation of GPUs memory capacity. The solution can benefit wide range scientific simulations.

Most of the GPU clusters are constituted of nodes and the connection between those nodes. In many cases, it uses like fat-tree network to connect those nodes by using InfiniBand mechanism [3], [4]. At each node, CPU acts as a controller which manages the connection between nodes and resource of different mechanism. Each node has some sockets that make it easier to input new devices. Devices are inserted to the sockets and the mechanism enables the connection between those devices. In this kind of system architecture, it is possible to increase the capacity of memories by inserting more memory cards into the sockets. For GPU device, the memory card and computing units is integrated as a whole device. For this reason, it is difficult to increase the memory capacity of GPU. It only depends on the manufactory. Commonly, the memory capacity of CPU is bigger than the memory capacity of GPU in most GPU based supercomputer systems.

As a common way to use GPUs to compute scientific simulations, the domain is sent from CPU side to GPU side to compute. On the GPU side, it uses different methods that focus on the full utilization of GPU architecture. The size of domain needs to fit the memory capacity of GPUs as it needs to send the initial to the GPU. For complicated simulations, as each process consume the resource of GPU, the memory limitation of GPU also makes it sensitive to the problem size [5]. The memory capacity of CPU is bigger than that of GPU as it only depends on the number of sockets. So, it gives us a possibility to enable the computation on the bigger domain by initialize the domain on the CPU side. We can contain the bigger domain on the CPU side and separate the domain into some sub-domains. Then, we can compute each sub-domain by GPU. By this kind of methods, it can compute bigger domain by GPUs. This is brief view of solving the bigger size applications. To achieve the objective, it needs to consider these factors. The first factor is to consider the communication between CPU and GPU. The connection is made up by the PCI express. Currently, the PCI express has a high speed which is almost 8GB/s. Yet, compared with the throughput of the GPU, the connection speed still remains as a problem. It reduces the performance especially in frequent communication case. The second factor is the performance of kernels on the GPU side. To improve the performance of the kernels, it uses the methods like spatial blocking, utilization of cache, and utilization of registersor shared memory on GPU side. The third factor is to consider the connection between the GPUs. It needs to exchange information via the connection between GPUs in some applications. The communication cost is high compared with the throughput of kernels. The fourth factor is the total executing time of the applications which is the key demand to use supercomputer systems. The fifth is the easiness and portability of programming.

With all those consideration, it is possible to figure out how to compute bigger size applications by the limited memory capacity of GPUs.

1.2 Contribution of this thesis

This thesis wants to enable the computation on the bigger domains on the GPU cluster while maintaining high performance. We select two kinds of real application to figure out optimization methods. We evaluate the optimization methods on TSUBAME system to analyze the result. The first one is stencil computation which is widely applied as a base computational kernel for scientific and engineering simulations [6], [7]. The second one is band sparse matrix vector multiplication which is widely applied as a base kernel in different kinds of simulations include equation solver [8], [9]. We implement the optimization methods to catch the characteristic the two applications and their same points. Those optimization methods are to efficiently utilize the memories to achieve high performance in real applications. We explain those contributions as below.

- For stencil computation, we enable the computation on the domain that is bigger than the memory capacity of GPUs while maintaining high performance. In single node case, it enables the computation on the domain which is 27 times bigger than the memory capacity of single GPU while reaches high performance. We developed a series of optimization methods and the best one achieves 1.35 times better than other methods. In multiple nodes case, we also developed a series of optimization methods which enable the computation on the domain that is 40 times bigger than the memory capacity of GPUs. The best one achieves 1.72 times better performance than other methods. We evaluate the methods for 7-point and 19-point stencil. The evaluation result shows that it can compute bigger domain while maintaining high performance in both stencil cases.
- For band sparse matrix multiplication case, we developed a series of new methods to achieve high performance. With the evaluation on those special matrices, we

conclude some important points and we plan to extend those methods to wide range matrix cases.

As the evaluation result proves, we enable the computation of bigger size applications while maintaining high performance. We use memory of CPUs to contain bigger domain and use GPUs to achieve high performance. By the efficient utilization of the memory of CPUs and GPUs, we enable the computation on the bigger domain while maintaining high performance.

This work is very important because of the contradiction between limited system resource and the demand of bigger size simulations. As the application size increased by the demand of the industry, it should face to the bigger size simulations. The size increases because of scientific demands or improvements. All those things may independently make the simulation size become bigger. Currently, the simulation size is limited by the total memory capacity of GPUs in GPU cluster. To increase the memory capacity of GPUs in GPU cluster, it needs to upgrade the system which consumes extraordinary time and fund. For this reason, the improvement of the system is slower than the growth of simulation size. Our work can efficiently solve that problem by enable the computation on the bigger domain by current GPU cluster. As introduced former, it is easier to increase the memory size of CPUs by inserting more memory cards into the sockets. Our optimization methods only depend on the memory size of CPUs as it needs to contain the initial of the bigger domain. Furthermore, we believe that it can also applied by different kinds of supercomputer systems. As the GPU cluster has limited number of GPUs, we believe that our methods can benefit all those GPU clusters.

We also discuss some limitation of our work in the next step. This includes the easiness and portability of programming. Also the further improvement of performance by understanding the characteristic of GPU device will be mentioned in the next.

Chapter 2. Background

2.1 Application: stencil computation

Stencil computation (SC) is widely applied as a base kernel in scientific and engineering simulations [1], [2], [5], [6], [7]. SC performs nearest neighbor computation on a spatial domain. For example, stencil computation is the base kernel for weather forecast as the below figure shows. The accuracy of the solution depends on the size of domain when using stencil computation.

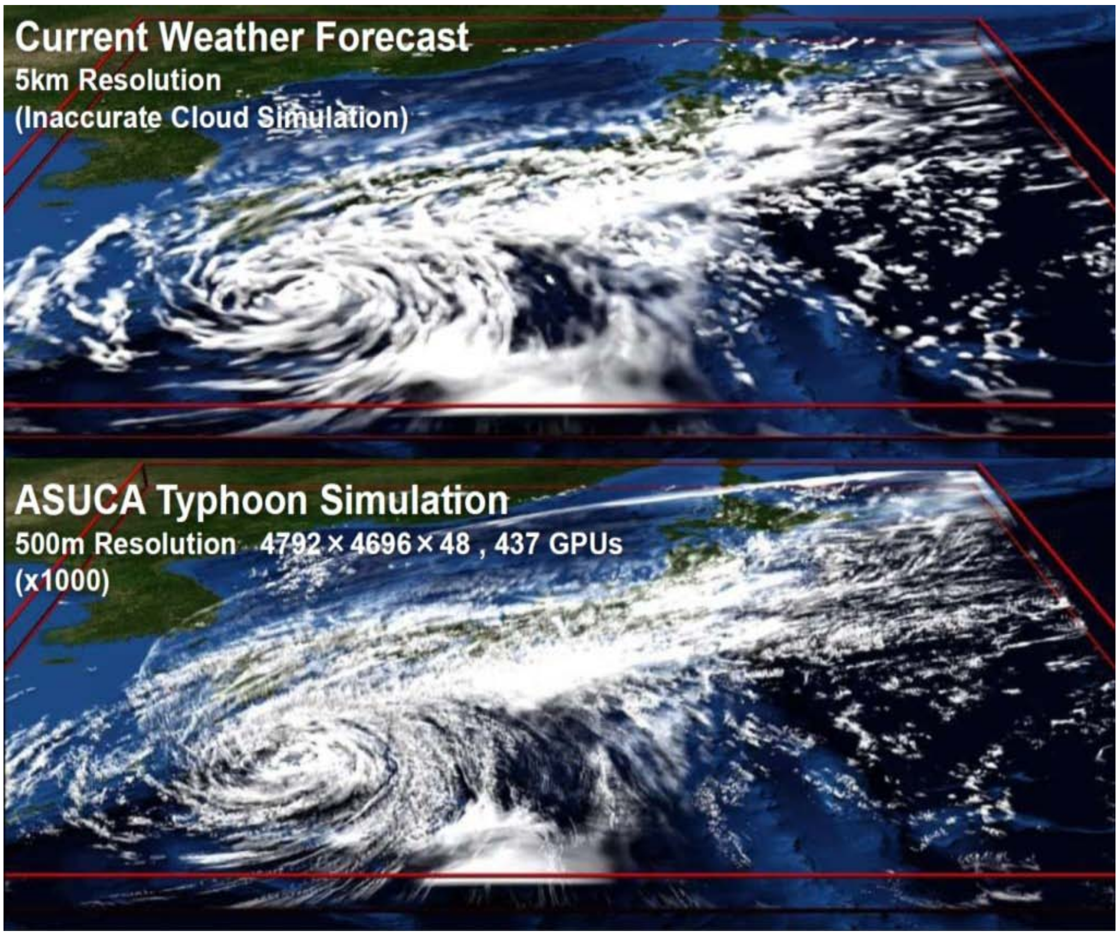

Current Weather Forecast
5km Resolution
(Inaccurate Cloud Simulation)
ASUCA Typhoon Simulation
500m Resolution 4792 × 4696 × 48 , 437 GPUs
(x1000)

Fig.1. Weather forecast that uses stencil computation

SC sweeps through the entire domain multiple times, called time steps, updating each domain point with the calculations of its nearest neighbors. At n-th time step, 7-point stencil computes all of the grid points for the next (n+1)-th time step. To compute each point, it needs to read seven points include itself and adjacent two points in each dimension as follows:

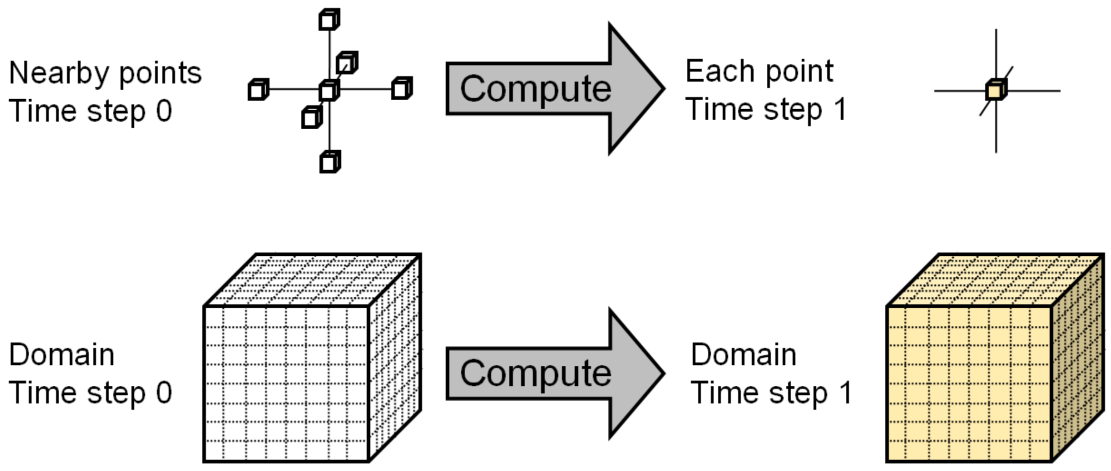


Fig.2. 7 point stencil

At n-th time step, in the 19-point stencil, the computation of each point involves adjacent 18 points (involves corner points). As the below figure shows, the computation of each point needs its nearest neighbors. At each time step, it needs to update each point through the whole domain.

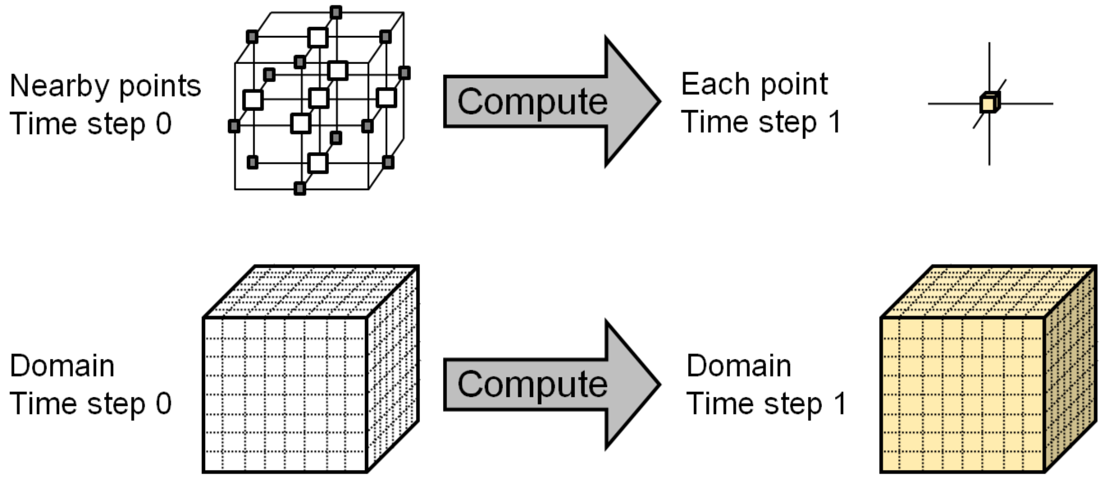


Fig.3. 19 point stencil

Here, we check the condition to compute each point at one time step. As we can see in below figure, if the nearby points have not been updated, it cannot compute the point for the next time step. Also, if there are no nearby points on the same memory, it also cannot compute the point for the next time step. If some nearby points are on the outside of the whole domain, it can use boundary condition to continue the computation.

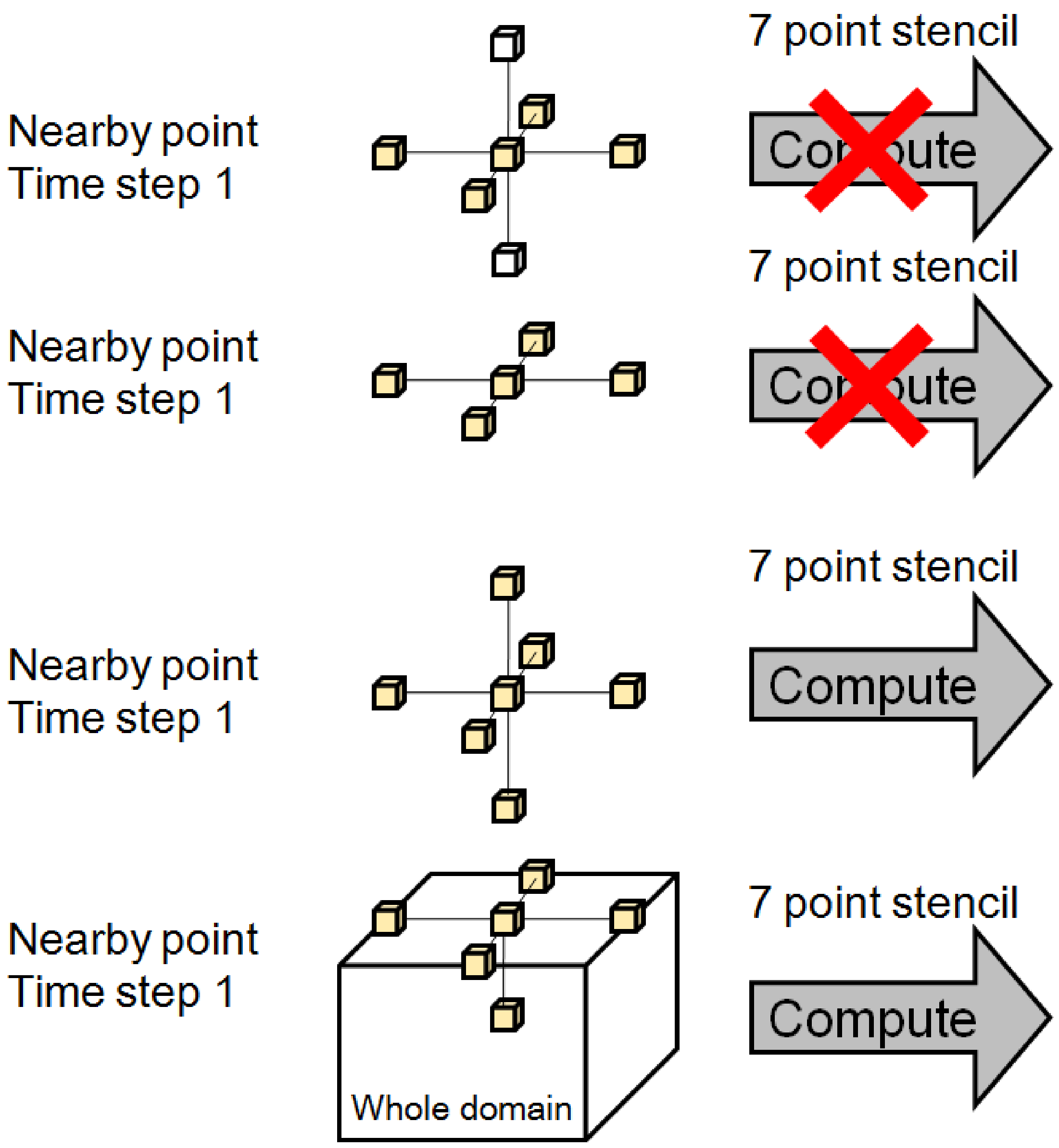
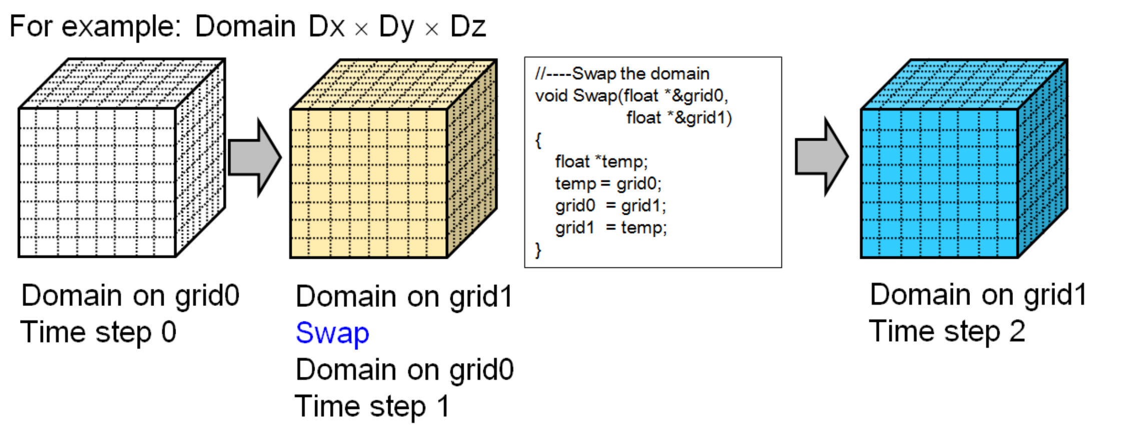


Fig.4. The conditions that each point can be computed in 7 point stencil.

To efficiently perform the stencil computation, it normally uses double buffering method to save and read the domain at each time step. It uses two grids, one grid is assigned to read the initial of the domain and the other one is assigned to save the result of the domain. For the next time step, it swaps two grids. So, the grid that reads the initial is now assigned to save the result; the grid that saves the result is now assigned to read the initial. It uses two grids to contain the initial and result to efficiently perform the stencil computation on the GPU side. It consumes 2 times space on the GPU side. We give the example of double buffering method on domain as below:



```
//----Swap the domain
void Swap(float *&grid0,
             float *&grid1)
{
   float *temp;
   temp = grid0;
   grid0  = grid1;
   grid1  = temp;
}
```

Fig.5. Double buffering method

After the computation of each time step, it swaps the two grids to continue the computation. To simplify the explanation, we will not identify the domain on which grid.

If the domain is divided into sub-domains, each point of the boundary needs adjacent nearby points which may belong to the other sub-domains. It calls those points on the other sub-domains as ghost boundary.

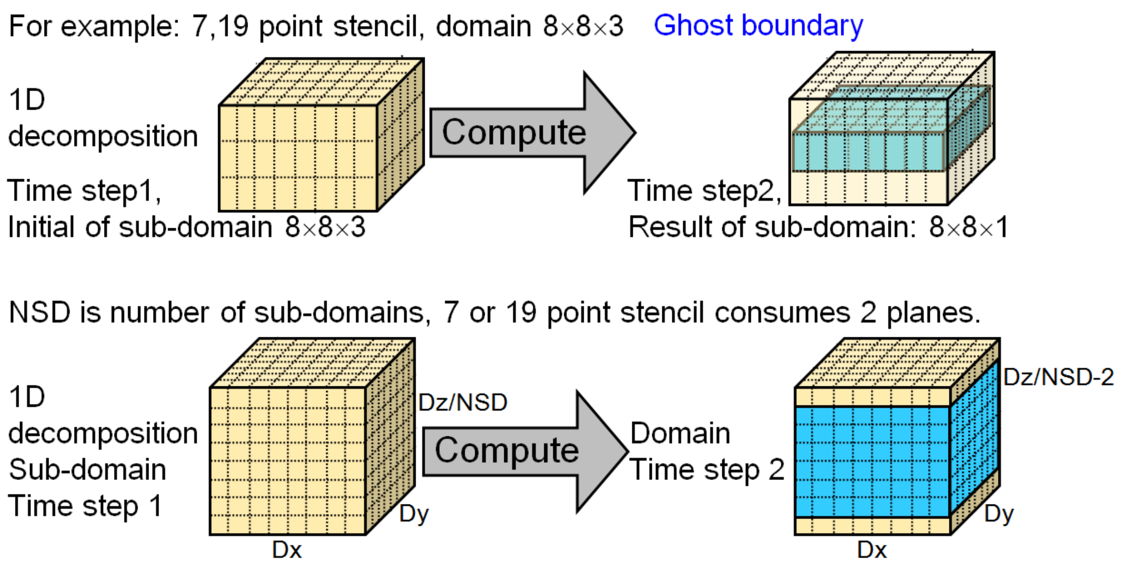
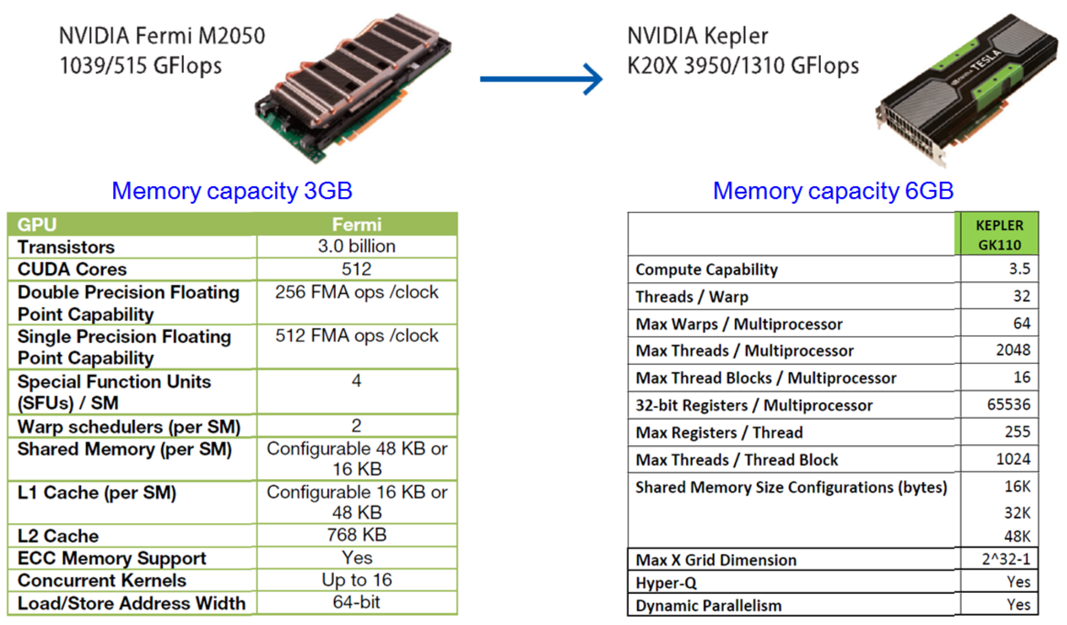

For example: 7,19 point stencil, domain 8×8×3
Ghost boundary
1D decomposition
Time step1,
Initial of sub-domain 8×8×3
Compute
Time step2,
Result of sub-domain: 8×8×1
NSD is number of sub-domains, 7 or 19 point stencil consumes 2 planes.
1D decomposition
Sub-domain
Time step 1
Dz/NSD
Dy
Dx
Compute
Domain
Time step 2
Dz/NSD-2
Dy
Dx

Fig.6. Ghost boundary for 1D or 3D decomposition.

Here Dx, Dy, Dz is the size of the X, Y, Z dimension. NSD is the number of sub-domains. If the domain is divided into sub-domains, each point of the boundary needs adjacent nearby points which may belong to the other sub-domains. It calls those points on the other sub-domains as ghost boundary.

2.2 GPU and CUDA program model

Graphics Processing Units for general-purpose computation (GPGPU) is proved to be a high-performance computing device to accelerate a wide variety of scientific and engineering simulations. A central processing unit (CPU) is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system. The reason of floating-point capability difference between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control [4].


NVIDIA Fermi M2050
1039/515 GFlops
NVIDIA Kepler
K20X 3950/1310 GFlops
Memory capacity 3GB
Memory capacity 6GB

GPU	Fermi
Transistors	3.0 billion
CUDA Cores	512
Double Precision Floating Point Capability	256 FMA ops /clock
Single Precision Floating Point Capability	512 FMA ops /clock
Special Function Units (SFUs) / SM	4
Warp schedulers (per SM)	2
Shared Memory (per SM)	Configurable 48 KB or 16 KB
L1 Cache (per SM)	Configurable 16 KB or 48 KB
L2 Cache	768 KB
ECC Memory Support	Yes
Concurrent Kernels	Up to 16
Load/Store Address Width	64-bit

	KEPLER GK110
Compute Capability	3.5
Threads / Warp	32
Max Warps / Multiprocessor	64
Max Threads / Multiprocessor	2048
Max Thread Blocks / Multiprocessor	16
32-bit Registers / Multiprocessor	65536
Max Registers / Thread	255
Max Threads / Thread Block	1024
Shared Memory Size Configurations (bytes)	16K 32K 48K
Max X Grid Dimension	2^32-1
Hyper-Q	Yes
Dynamic Parallelism	Yes

Fig.7. NVIDIA GPUs

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with high arithmetic intensity – the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In fact, many algorithms are accelerated by data-parallel processing.

In November 2006, NVIDIA introduced CUDA™, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language. Other languages or application programming interfaces are supported, such as CUDA FORTRAN, OpenCL, and DirectCompute.

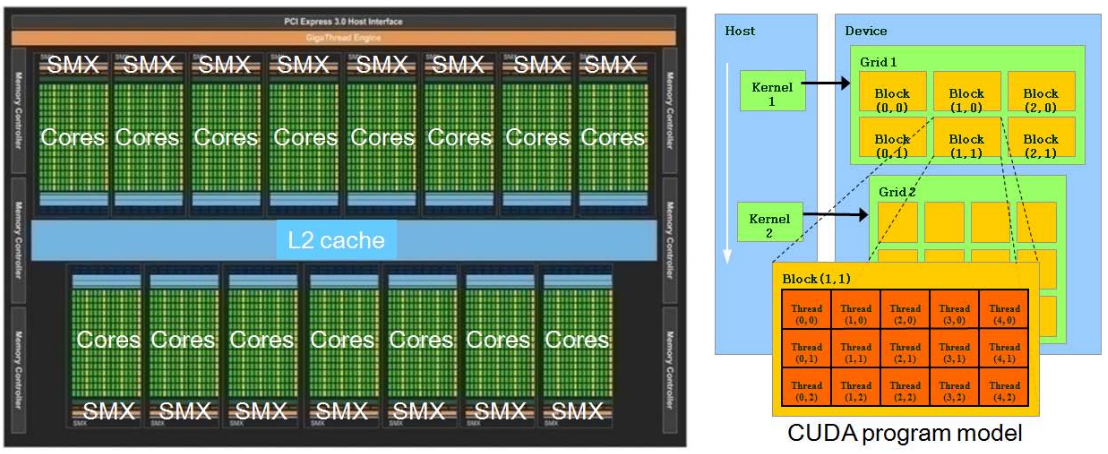


Fig.8. GPU architecture and CUDA program model

The advent of multi-core CPUs and many core GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, transparently scale their parallelism to many core GPUs with widely varying numbers of cores. The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores and only the runtime system needs to know the physical processor count.

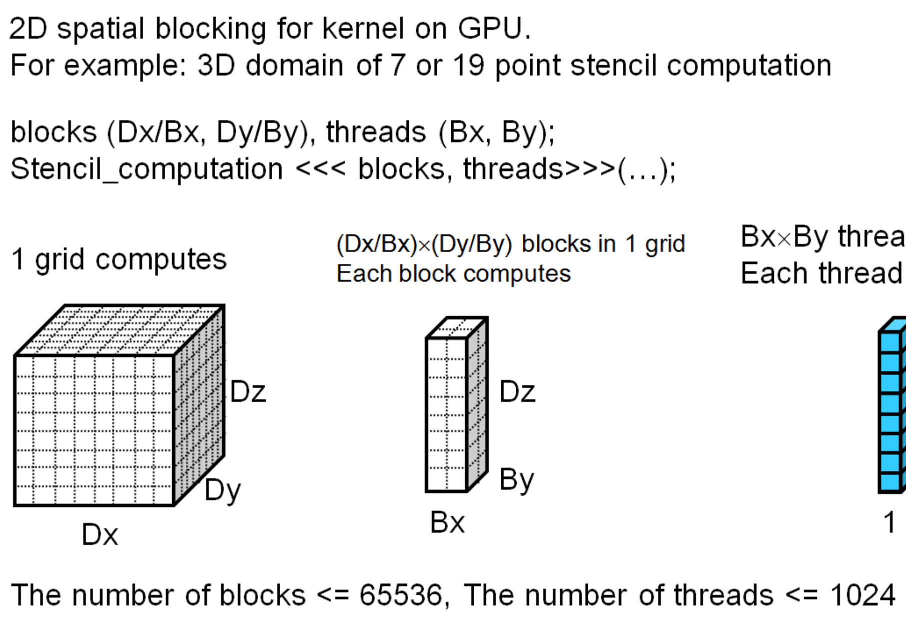
CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<…>>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain.

2.3 2D spatial blocking method for stencil computation

2D spatial blocking [5], [11] is an efficient way to perform stencil computation on the GPU size. Since the computation involves a large number of memory accesses, it should reduce access to the global memory of the GPU. Thus, it should reuse data in registers efficiently. Moreover, it should invoke sufficiently larger number of threads than that of physical CUDA cores in order to hide latency of memory accesses.


2D spatial blocking for kernel on GPU.
For example: 3D domain of 7 or 19 point stencil computation

blocks (Dx/Bx, Dy/By), threads (Bx, By);
Stencil_computation <<< blocks, threads>>>(…);

The number of blocks <= 65536, The number of threads <= 1024

Fig.9. 2D spatial blocking of stencil computation

To fulfill those requirements, it designed a kernel function that calculates the points of a computational part of size of (Dx, Dy, Dz) for a single time step. The kernel function is invoked on a GPU with (Dx/Bx, Dy/By) thread blocks, each of which has (Bx, By) threads. Bx×By should no more than the number of threads that a single block can contain. It is better to set Bx more than By to improve data locality. It divides the given part into pieces of size of (Bx, By, Dz) as shown in upper figure. Here each thread calculates Dz points in the Z direction; a thread that corresponds to (i, j) updates grid points (i, j, k) where k varies from 0 to Dz. When a thread computes a point in the k + 1-th plane, some points to be referred have been already accessed by itself in the previous k-th plane computation. It reuses such data by holding them in registers to reduce the global memory access. On the other hand, it does not use on-chip shared memory to store data shared by several neighbor threads. It is called as 2D spatial blocking.

2.4 TSUBAME system

TSUBAME2.0 supercomputer is developed by Global Scientific Information and Computing Center (GSIC) at Tokyo Institute of Technology [10]. TSUBAME 2.0 ranked world number two on "The Green 500 List" of 2010 which are the world's most powerful supercomputers in the power efficiency. TSUBAME2.0 also ranked number 4 on "The Top 500 List" by the absolute performance which is 2.4 PFLOPS. It consists of thin, medium and fat computing node which have different system specifications. The nodes are interconnected by dual QDR InfinitBand network with a full bisection-bandwidth fat-tree topology. The nodes mainly consist of two Intel Xeon Westreme-EP 2.9 GHz CPUs and three NVIDIA M2050 GPUs. We call the memory of CPU side as host memory and the memory of GPU side as device memory. Each GPU has 3GB device memory and connects to the CPU via PCI express. There are 1408 thin computing nodes, 24 medium nodes and 10 fat nodes. Each thin node has 54GB host memory, each medium node has 128 GB host memory and each fat node has 256 GB or 512 GB host memory. The local storage (SSD) of each thin node is 120 GB or 240 GB. The local storage of medium and fat node is 480GB. As a high performance supercomputer system, TSUBAME2.0 supplies peta scale throughput and storage which makes most of current simulations can be executed and get correct result in an endurable time. The key point to use TSUBAME2.0 is the multiple GPUs which can get high throughput by the data-parallel programming model of GPU architecture. The CPU is installed separately from the host memory and both of them are inserted into the sockets of the motherboard at each node. The host memory is also inserted into the socket of motherboard as a memory card. It is possible to increase the capacity of host memory by inserting into more memory card. Instead, the device memory is integrated to the GPU device which makes it impossible to increase the memory capacity.

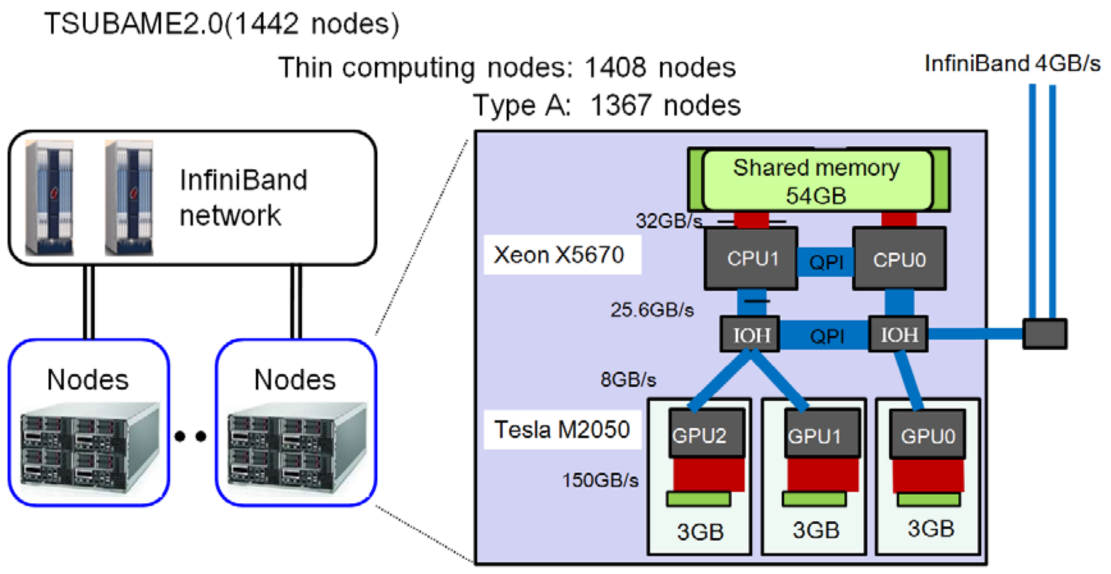


Fig.10. TSUBAME2.0 architecture

Commonly, the serial code executes on the host while parallel code executes on the device to efficiently utilize the strong point of CPU and GPU. The domain is initialized on the CPU side and sent to GPU side to compute. This is the common way to use GPU cluster to compute real applications on single node. For the applications on multiple nodes, it needs to copy data between CPU to GPU to continue the computation or exchange data between GPUs. As we mentioned above, TSUBAME2.0 is consisted of nodes and the connection between those nodes. On each node, there are CPU memory and GPU memory. Recently, TSUBME2.0 is upgraded to TSUBAME2.5. It will be upgraded to TSUBAME3.0 in the future. TSUBAME2.5 upgraded TSUBAME2.0 by exchanging the Tesla M2050 to Tesla K20X which means the throughput of each GPU is upgraded. The CPUs and the connections between nodes and inside nodes remain the same. The Tesla M2050 is Fermi and Tesla K20X is Kepler in NVIDIA product series.

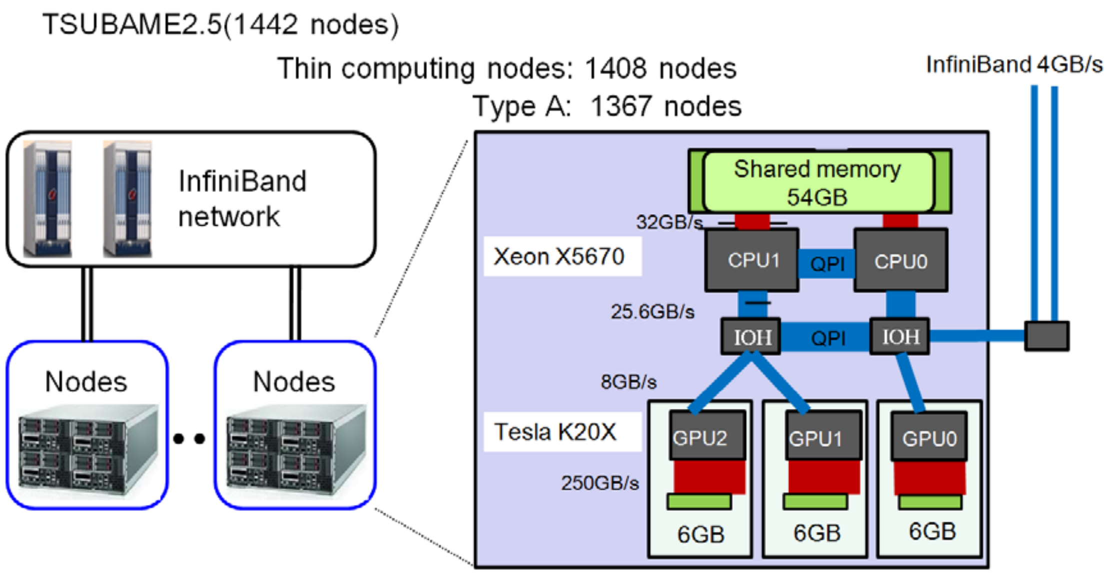


Fig.11. TSUBAME2.5 architecture

Fermi introduced an L1 cache in addition to the shared memory which is available since the earliest CUDA-capable GPUs. In Fermi, the shared memory and the L1 cache share the same physical on-chip storage, and a split of 48 KB shared memory / 16 KB L1 cache or vice versa can be selected per application or per kernel launch. Kepler continues this pattern and introduces an additional setting of 32 KB shared memory / 32 KB L1 cache, the use of which may benefit L1 hit rate in kernels that need more than 16 KB but less than 48 KB of shared memory per multiprocessor. L1 caching in Kepler GPUs is reserved only for local memory accesses, such as register spills and stack data. In Kepler GPUs, global loads are only cached in L2 only. In Fermi GPUs, the global loads are cached in L1 and L2 caches which mean more data locality in caches to improve data hit rate. Kepler has more computing units which are called SMs and has bigger L2 cache. So, Kepler ensures performance improvement in most application cases, and Fermi has strong point in data caching. For the programs that developed on Fermi GPUs, it also needs to consider how to revise the program to Kepler version to fully utilize the strong point of Kepler.

TSUBAME 3.0 is next generation supercomputer which will come in 2015-2016 to target green computing, extra big data computing, and extra scale resilience.

2.5 Bigger computational domain

In the real scientific simulations, it needs to increase the computational domain for these reasons.

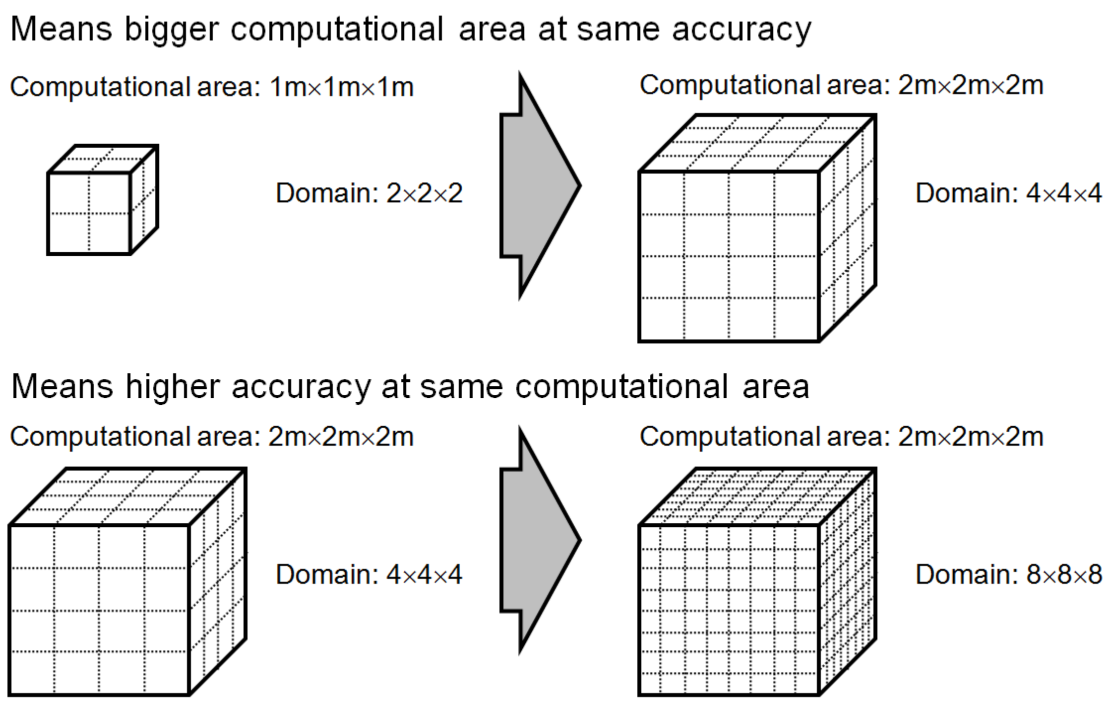

Means bigger computational area at same accuracy
Computational area: 1m×1m×1m
Domain: 2×2×2
Computational area: 2m×2m×2m
Domain: 4×4×4
Means higher accuracy at same computational area
Computational area: 2m×2m×2m
Domain: 4×4×4
Computational area: 2m×2m×2m
Domain: 8×8×8

Fig.12. Bigger computational domain

The first reason is that bigger computational domain means bigger computational area which is important to simulations like weather forecast. In those simulations, bigger computational area means capability to predict wider range area which is important to protection from disasters. The second reason is that bigger computational domain means higher accuracy if it use stencil method. More accuracy can show more details of the simulations which make it possible to find more details and uncover some new field of research.

As we can see in the upper figure, if we use stencil computation method to simulate the problem, bigger domain means bigger computational area at the same accuracy. The bigger domain also means higher accuracy at the same computational area. For all those reasons, we need to solve the contradiction between the demand of bigger computational domain and the limitation of GPUs memory capacity. The solution can benefit wide range scientific simulations.

2.6 Common way to use GPUs and limitation

The common way to use GPU to compute the domain is to initialize the whole domain on the CPU side. Then, it copies the whole domain to the GPU side.

On the GPU side, it computes the whole domain for multiple times which is called time steps. By this way it can utilize the throughput of GPU totally and only needs to send the whole domain to GPU side at first time step. It copies the result at last time step. So, it can efficiently reduce the communication between CPU and GPU.

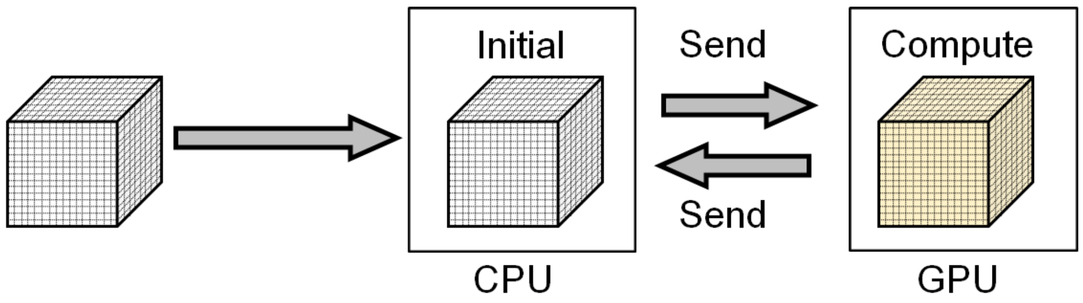


Fig.13. Common way for using single GPU on single node

As the whole domain is copied to GPU side, the domain should fit the memory capacity of GPU. If the domain is smaller than the memory capacity of GPU, it does not need to consider the limitation of GPU memory capacity. If the domain is bigger than the memory capacity of GPU, it cannot directly copy the domain to GPU side as below.

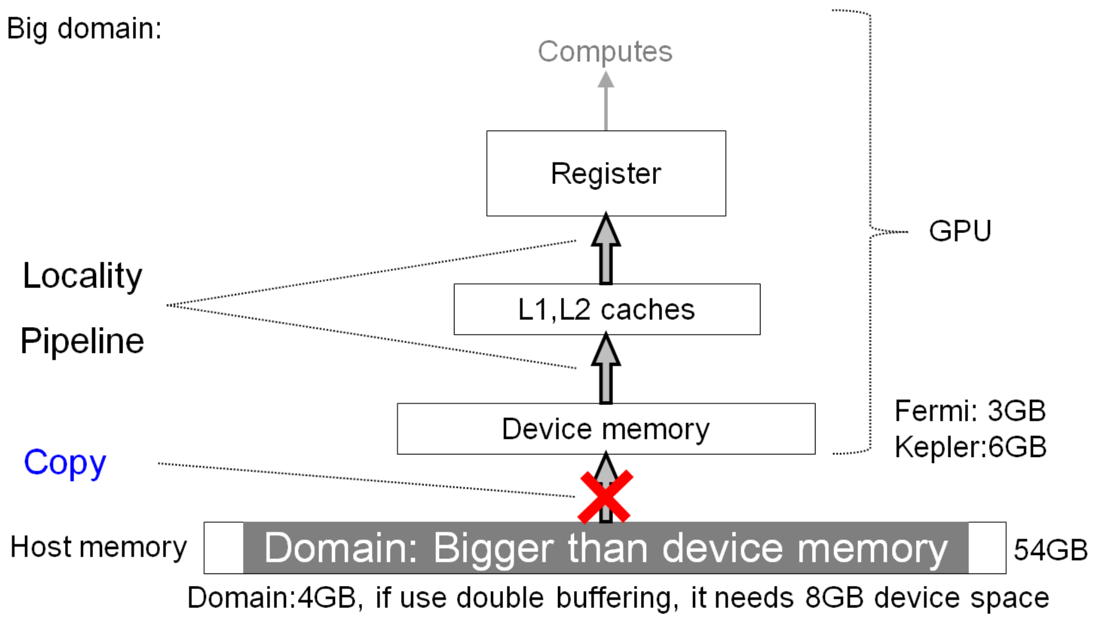


Fig.14. Memory limitation of common way

If domain is bigger than the memory capacity of GPU, it commonly uses multiple GPUs to extend the memory capacity as below:

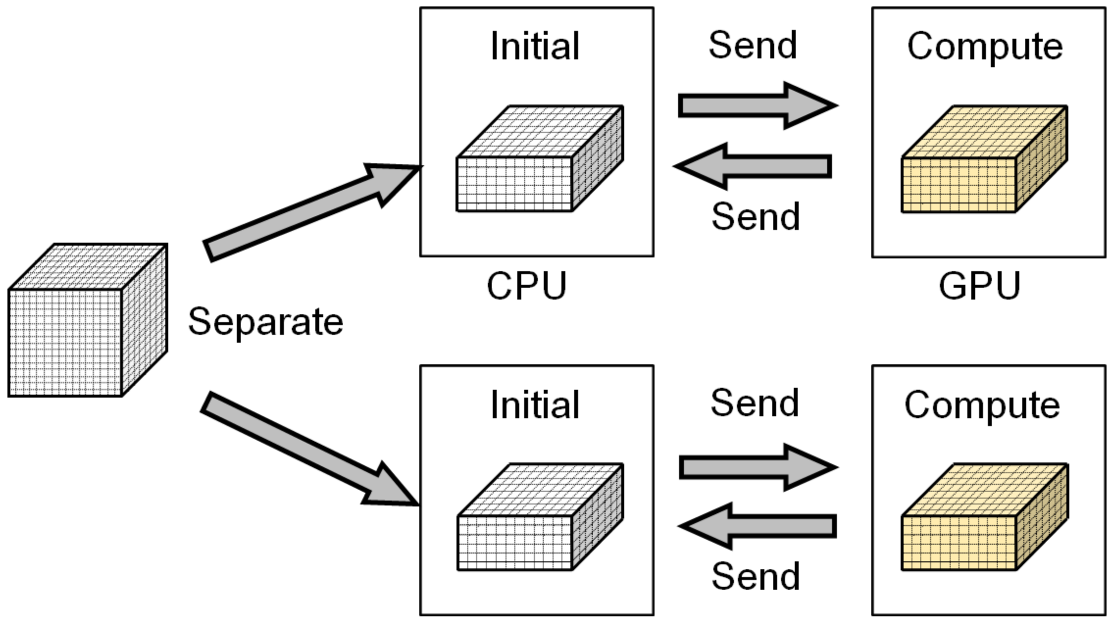


Fig.15. Common way to extend the memory capacity of GPUs

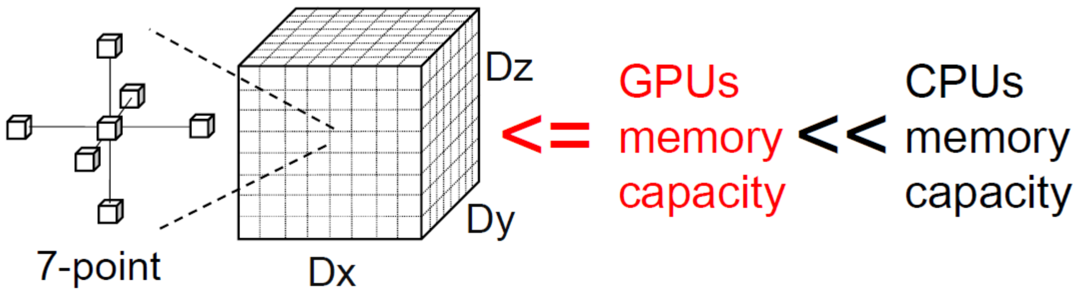


Fig.16. The limitation of domain size in common way case

As the domain should fit the memory capacity of GPUs in common way case, the domain size is limited by the total memory capacity of GPUs as the upper figure shows. As the total memory capacity of CPUs is bigger than the memory capacity of GPUs, it has possibility to enable the computation on the domain that is bigger than the memory capacity of GPUs by efficiently use the memory capacity of CPUs.

2.7 Objective: efficiently use the memories of GPU cluster

Our objective is to efficiently use the memories of GPU cluster to enable the computation on the domain that is bigger than the memory capacity of GPUs. For example, if the domain is 4GB and use double buffering method, it cannot copy the domain to the GPU side to compute as the below figure shows. We separate the domain into sub-domains and compute each sub-domain by the same GPU as below.

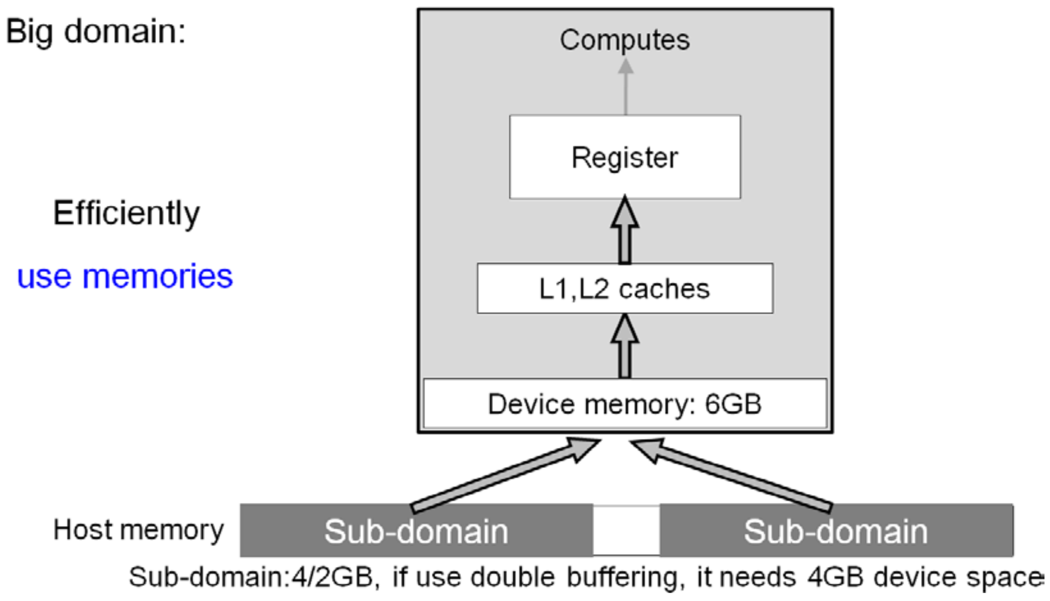


Fig.17. Efficiently use the memories.

Chapter 3. Related work

3.1 Communication Avoiding Kernels

Jim Demmel [29], [30] proposed the communication-avoiding algorithm for matrix multiplication. In CPU case he has mentioned that the communication cost includes moving data between levels of a memory hierarchy and processors over a network as below figure shows.

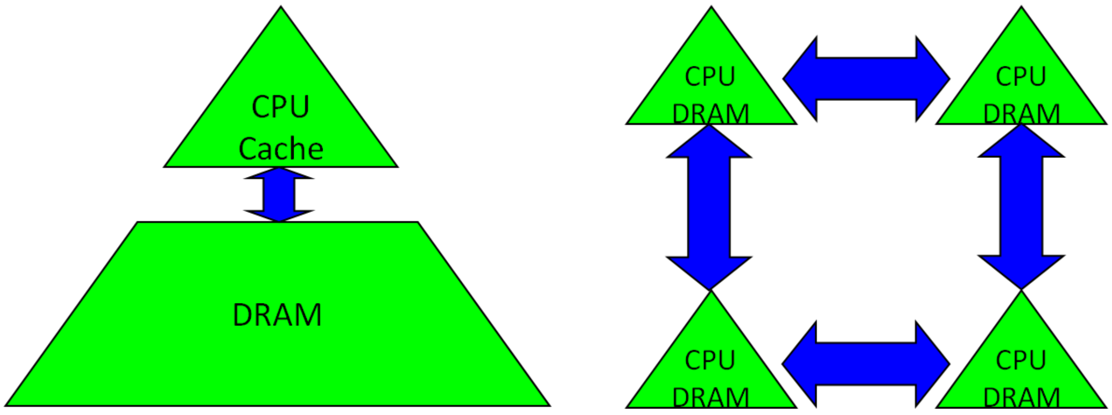


Fig.18. Communication between different processors

In tri-diagonal case, his algorithm Improves performance and accuracy on extreme-Scale computing systems. On modern computer architectures, communication between processors takes longer than the performance of a floating point arithmetic operation by a given processor. ASCR researchers have developed a new method, derived from commonly used linear algebra methods, to minimize communications between processors and the memory hierarchy, by reformulating the communication patterns specified within the algorithm. This method has been implemented in the TRILINOS framework, a highly-regarded suite of software, which provides functionality for researchers around the world to solve large scale, complex multi-physics problems.

The focus of his method is to develop a systematic approach for designing and analyzing dense linear algebra algorithms, paying particular attention to avoiding communication costs as much as possible, in order to optimize performance on a wide range of platforms. To maintain general applicability of algorithmic ideas, he considers one sequential and one parallel architectural model that are simple enough to facilitate coarse asymptotic communication analysis of algorithms but accurate enough to predict performance of implementations on real hardware.

Communication-avoiding '2.5D' algorithms take advantage of the extra available memory and reduce the bandwidth cost of many algorithms in numerical linear algebra. Generally, 2.5D algorithms can use a factor of c more memory to reduce the bandwidth cost. The theoretical communication reduction translates to a significant improvement in strong-scalability (scaling processor count with a constant total problem size) on large supercomputers.

In many cases, he proves lower bounds on the communication required of a computation on a particular machine model, thereby setting a target for algorithmic performance. Establishing lower bounds and developing and improving algorithms is often a simultaneous process, with the objective of identifying algorithms that are provably communication optimal. After an efficient algorithm has been developed for the modeled machine, it will rely on automatic performance tuning, or auto-tuning, to tweak the parameters of the algorithm for optimal performance on a particular platform. The ultimate goal of his work is to deliver the improved performance of the new and improved algorithms to scientists across many disciplines and other users by integrating the ideas into the state-of-the-art libraries. The k iterations is $y = A \cdot x$ with $[Ax, A^2x \ldots A^kx]$. The sequential and parallel algorithms are below in matrix case:

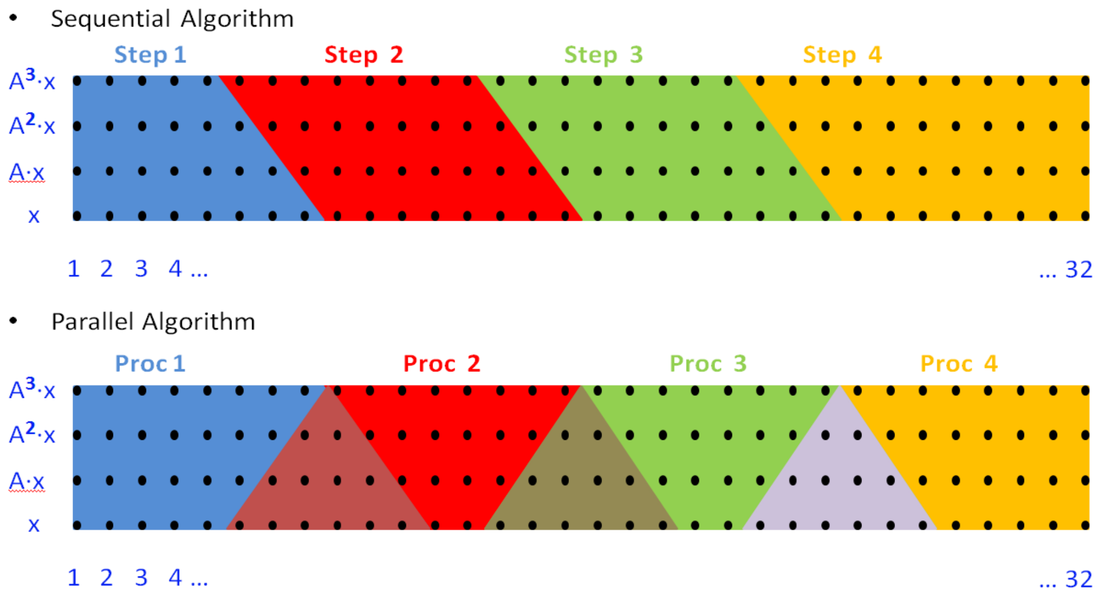


Fig.19. Sequential and parallel algorithms

3.2 Temporal blocking method

Leonardo Mattes [25], [26], [27] proposed optimization method for finite difference time domain problem which focuses on overcoming the GPU memory limitation on FDTD through the use of overlapping sub-grids. The method finite difference time domain (FDTD) is widely used in electromagnetic simulations to solve problems of microwave tomography, radar and telecommunications. Since this method is a data intensive and computation intensive problem, there are a lot of initiatives to improve the scalability and performance of the FDTD. In many problems of High-performance computing that need large memory capacity, the evident solution is the division of the problem in small parts, in order to solve it in a fragmentary way. In other words: store the entire data in the CPU and systematically transfer part of these data to the GPU.

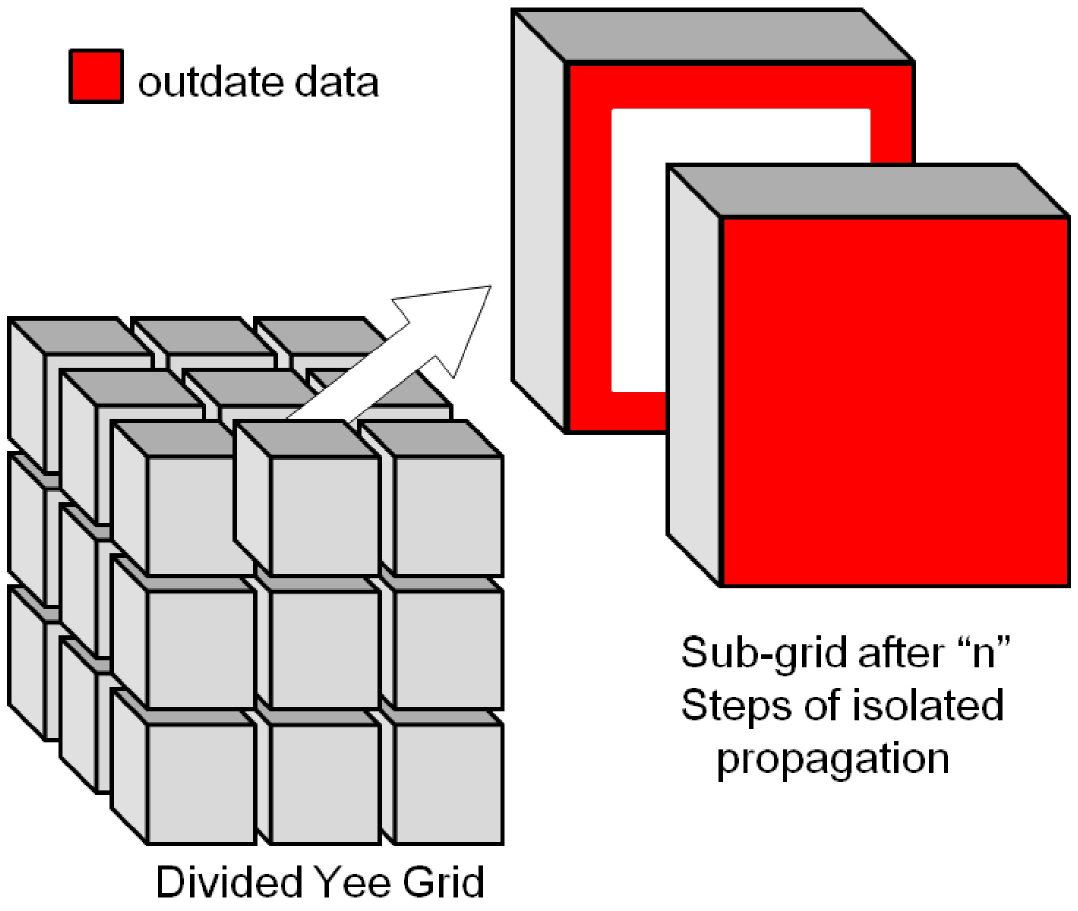


Fig.20. Yee sub-grid after n time steps of simulations. For each time steps propagation a line of outdated data arises from the board of the division.

Specifically to FDTD, store all Yee grids in the CPU memory to transfer part of that to GPU to calculate the cell updates, then the simulation could occur in steps and the memory amount of the simulation could exceed the GPU capacity (but not the CPU capacity).The challenges they had overcome:

- High latency of data transfer from CPU to GPU: Minimize the memory transfer rate between CPU and GPU;
- Dependence of adjacent cells to calculate the propagation of the Yee grid: The calculation of the electric and magnetic fields in a cell of the Yee grid on time step t depends on the values of the adjacent cells on time step t-1. Thus, when a Yee sub-grid is propagated, it is necessary to seek the updated values of the cells in the border of the division. If these values are not available, the correct calculation in the border of the division is compromised, an error that could be propagated in a way that, for each time step, a line of outdated data arises from the board of the division. Below figure shows this kind of error, in which after an isolated propagation of one Yee sub-grid for each time steps, a line of outdated data arises from the board of the division.

Together, the high data transfer latency and the dependence of adjacent cells to calculate the propagation of the Yee grid is a challenge to apply the FDTD on a GPU in a fragmented. They proposed a model of Yee grid division with areas of redundancy, creating overlapping sub-grids that allow the propagation of the Yee sub-grids for multiple time steps without compromising the integrity of the simulations. After the Yee division, each sub-grid is increased by a redundancy area of n lines of width. Thus, after its n time steps propagation, the area of outdated values correspond to the redundancy area that is discarded without compromising the integrity of the simulation. In according of the proposed solution, the FDTD propagation is made through the following steps:

His work can avoid communication cost between CPU and GPU and can overcome the memory limitation of GPU. Yet, as he mentioned, his method has to initialize a bigger sub-grid which cause more communication and computation. We call his method as temporal blocking method and the more communication and computation is called as redundancy problem. It is important to solve the redundancy problem to improve the performance.

3.3 Other related works

3.3.1 Efficiently use the shared memory of GPU

Anthony Nguyen [11] proposed and implement a 3.5D blocking scheme for stencil kernels by performing a 2D spatial and 1D temporal blocking to convert bandwidth bound kernels into compute bound kernels. The resultant stencil computation scales well with increasing core counts and is also able to exploit data-level parallelism using SIMD operations. Additionally, he provides a framework that determines the various blocking parameters – given the byte/op of the kernel, peak bytes/op of the architecture and the on-chip caches available to hold the blocked data. As a result, the performance of 7-point stencil is comparable or better than the fastest reported in the literature for GPUs. With the future architectural trend of increasing compute to bandwidth ratio, his blocking scheme would become even more important for stencil based kernels.

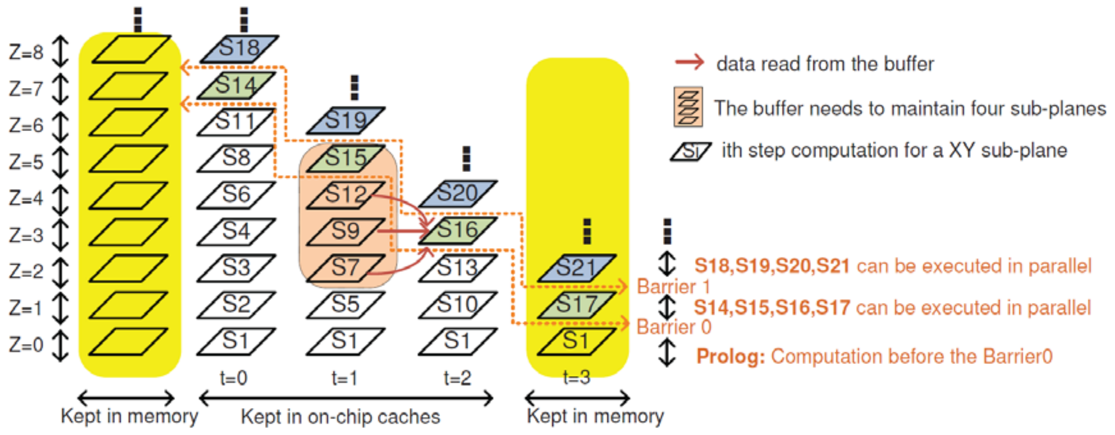
He presented a novel 3.5D blocking algorithm that performs a 2D spatial blocking and an additional temporal blocking of the input grid into on-chip memory. His method which is called 3.5D spatial blocking blocks in two dimensions and streams through the third dimension. Thereby it increases the blocking size and resulting in better utilization of the on-chip memory. The temporal blocking performs multiple time steps of SC, re-using the data blocked into shared memory and registersfiles for GPUs. This reduces the effective bandwidth requirement and allows for full utilization of the available compute resources. The resultant algorithm is amenable to both thread-level and data-level parallelism, and scales near-linearly with SIMD width and multiple cores. He applies this 3.5D blocking algorithm to two specific examples. He gives the example is a 7-point stencil for 3D grids. The performance of previously optimized implementations of 7-point stencil is bandwidth bound on state-of-the-art CPUs and GPUs. He made the following contributions:

- It presents the most efficient blocking mechanism that reduces memory bandwidth usage and reduces overhead due to blocking. His stencil computation (SC) implementation is no longer memory bandwidth bound even for very large grids.
- By making SC compute bound, his implementation effectively utilizes the available thread- and data level parallelism on modern processors. It scales near linearly with multiple cores and SIMD width on both CPUs and GPUs.
- It presents a flexible load-balancing scheme that distributes the grid elements equally

amongst the available threads, all of which perform the same amount of external memory read/write and stencil computations.

- On NVIDIA GTX 285 GPU, his 7-point stencil for single precision inputs is 1.8X faster than previously reported numbers. His GPU implementation employs an efficient work distribution across thread warps, thereby reducing the overhead of parallelization.

Although 2D spatial blocking ensures near-optimal usage of memory bandwidth, it still does not guarantee full utilization of the computational resources. Since the original application executes multiple time steps (usually hundreds to thousands), the only way to reduce bandwidth is to execute several time steps of the blocked data so that the intermediate data can reside in the cache, and then store the resultant output to the main memory. This proportionately reduces the amount of bandwidth.



Z=8
Z=7
Z=6
Z=5
Z=4
Z=3
Z=2
Z=1
Z=0
S18
S14
S11
S8
S6
S4
S3
S2
S1
S19
S15
S12
S9
S7
S5
S20
S16
S13
S10
S21
S17
t=0
t=1
t=2
t=3
Kept in memory
Kept in on-chip caches
Kept in memory
data read from the buffer
The buffer needs to maintain four sub-planes
ith step computation for a XY sub-plane
S18,S19,S20,S21 can be executed in parallel
Barrier 1
S14,S15,S16,S17 can be executed in parallel
Barrier 0
Prolog: Computation before the Barrier0

Fig.21. 3.5D spatial blocking

Although temporal blocking has been used in the past [26], [27], he provides: (a) algorithm for combining temporal blocking with 2D spatial blocking to achieve the compute and bandwidth friendly 3.5D blocking, (b) thread-level and data level parallel algorithms to fully exploit the available compute resources. For temporal blocking, denote dimT as the number of time steps, it needs to block before writing the data to the external memory. To accommodate for dimT time steps, it extends the buffer to store the XY sub-planes for dimT time instances (starting from 0 to (dimT -1)). After the (dimT −1) time steps, the output for the next step is written out to the external memory.

3.3.2 Efficiently use the registersof GPU

In Wai Teng Tang's paper [28], he explores a different data access strategy to improve the performance of stencil kernels. In particular, he proposed a novel algorithm for computing stencil operations based on an in-plane method as opposed to the more commonly used forward-plane method. One of the main difficulties of stencil computations is the loading of boundary data elements known as the halo regions. With the appropriate tuning, his in-plane method is able to obtain optimal parameters on different GPU platforms, and is able to achieve better performance than prior methods.

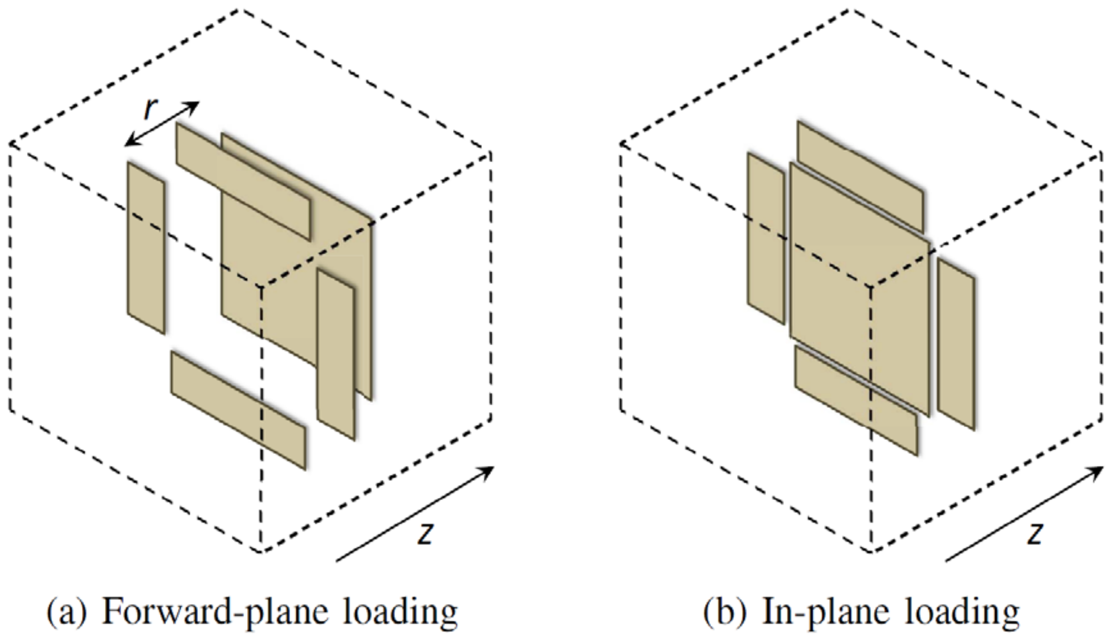


Fig.22. Forward and in-plane loading methods

In the in-plane method that he proposed, the plane in which the interior elements are read is made to coincide with the plane containing the halos. This opens up a few possibilities in which the memory loading patterns can be varied. The main difference between the two methods is that the former fetches all neighbor values for calculating each output element, while his method performs incremental updates of the output element from partial reads of the neighbor data. With in-plane loading, the number of flops per stencil element is increased while the number of data references remains the same. Note that since the z values are also stored in registers as in the 2D blocking method, the bandwidth required by the in-plane method is similar to that of 2D blocking method.

3.3.3 Implementation on TSUBAME system

In Takashi Shimokawabe and Takayuki Aoki's paper [5], they describe their implementation of the phase field simulation. They describe a single-GPU implementation first and then their implementation strategies with multiple GPUs over the TSUBAME 2.0 nodes including techniques to improve scalability. They describe their strategies of multi-GPU computation. Using multiple GPUs is necessary not only for higher performance, but for accommodating larger-size problems beyond the capacity of video memory on a single GPU (i.e., 3 GB on M2050 GPU).

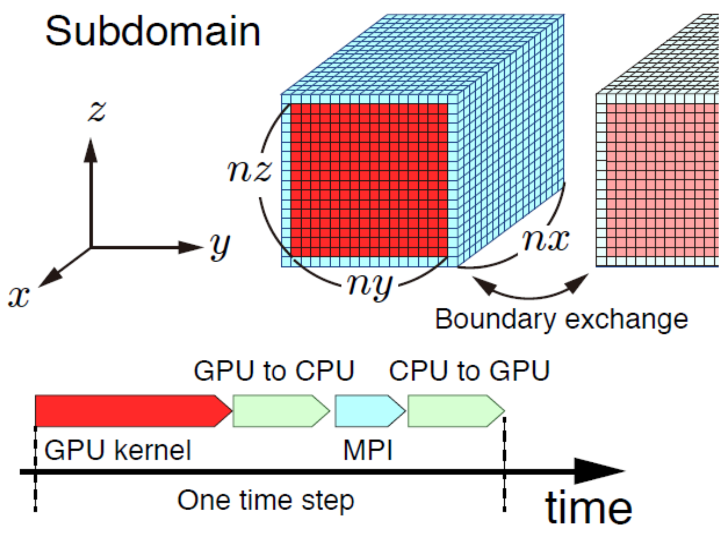


Fig.23. Scheme of the GPU-only method.

They decompose the whole computational domain in both y- and z-directions (2D decomposition) and allocate each sub-domain to one GPU. We have chosen this method since 3D decomposition, which looks better to reduce communication amount, tends to degrade GPU performance due to complicated memory access patterns for data exchanges between GPU and CPU. Similar to conventional multi-CPU implementations with MPI, their multi-GPU implementation requires boundary data exchanges between sub-domains. Because a GPU cannot directly access to the global memory of other GPUs, host CPUs are used as bridges for data exchange. For inter-node cases, this data exchange is composed of the following three steps: (1) the data transfer from GPU to CPU using CUDA APIs such as cudaMemcpy, (2) the data exchange between nodes with MPI, and (3) the data transfer back from CPU to GPU with CUDA APIs.

Chapter 4. Optimization methods for stencil computation on single node

4.1 Introduction

In this chapter, we first introduce existing optimization methods that can compute bigger domain by single GPU while maintaining high performance. Then, we will introduce our optimization methods. In single node case, we only use 1 GPU and only use 1D decomposition to simplify the implementation. We assume the points are saved in an order of X, Y, Z dimension. Double buffering is the method to save initial and result of domain. 2D spatial blocking is the method to implement the kernel on GPU side. We use double buffering method and 2D spatial blocking as default in 3D domain case.

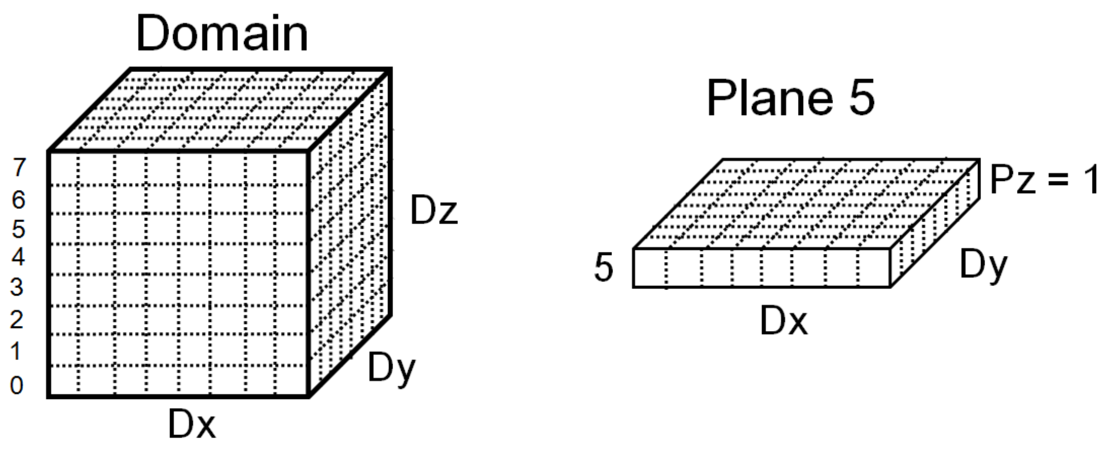


Fig.24. Set the number to each plane.

We set the number to the planes to explain the optimization method as upper figure shows. Each plane has a unique number. As upper figure shows, in domain Dx×Dy×Dz, each plane is Dx×Dy×1. We use single node of TSUBAME2.0 and TSUBAME2.5 to evaluate the optimization methods. The difference between two systems is that the memory capacity of each GPU is increased from 3 GB to 6GB. The memory capacity of each GPU on TSUBAME2.0 is 3GB; the memory capacity of each GPU on TSUBAME2.5 is 6GB. We assume the computation precision is floating.

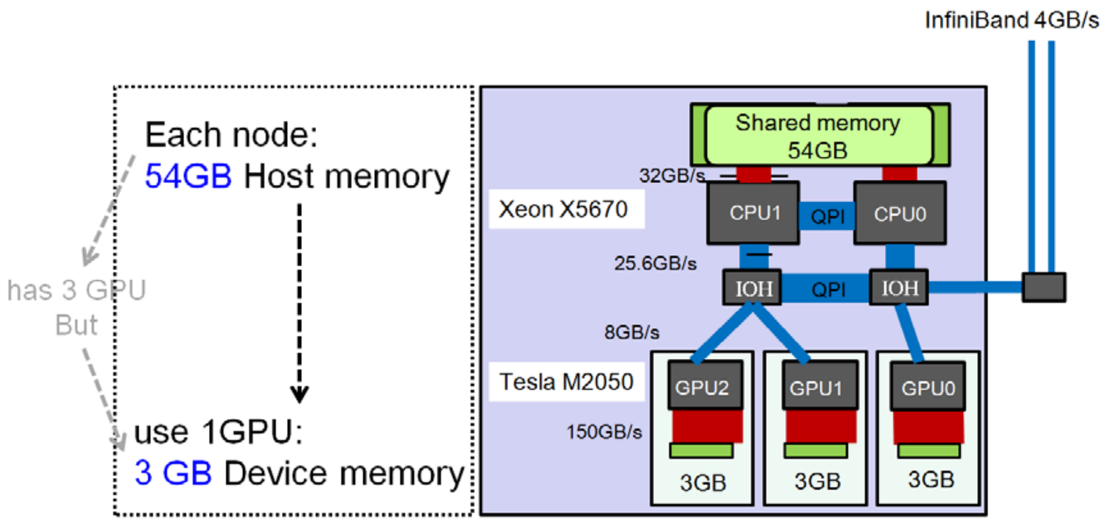


Fig.25. Use single node of TSUBAME2.0 to evaluate the optimization methods

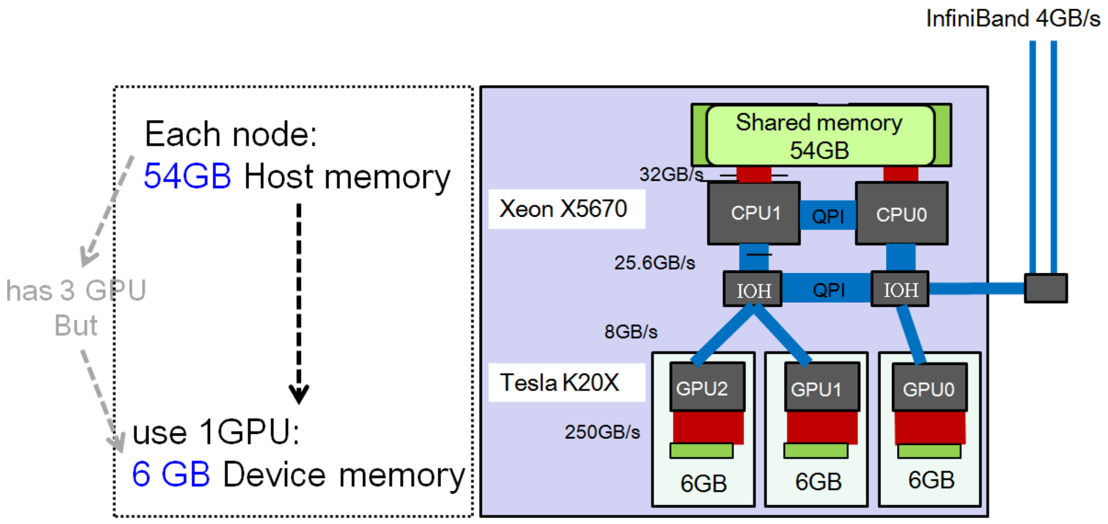


Fig.26. Use single node of TSUBAME2.5 to evaluate the optimization methods

4.2 Existing: Common way

The common way to use single node to compute the domain of stencil computation is to initialize the domain on the CPU side. Then, it copies the whole domain to the GPU side.

On the GPU side, it computes the domain for multiple times which is called time steps. By this way, it can utilize the throughput of GPU totally and the communication between CPU and GPU only occurs at first and last time step. It can efficiently reduce the cost of the communication between CPU and GPU via PCI express. As it only computes the domain that is smaller than the memory capacity of GPU, it can only compute small domain as below figure shows. On the small domain area, it has high performance. If use double buffering method, domain size is limited by the formula: Dx×Dy×Dz×2 <= GPU memory capacity.

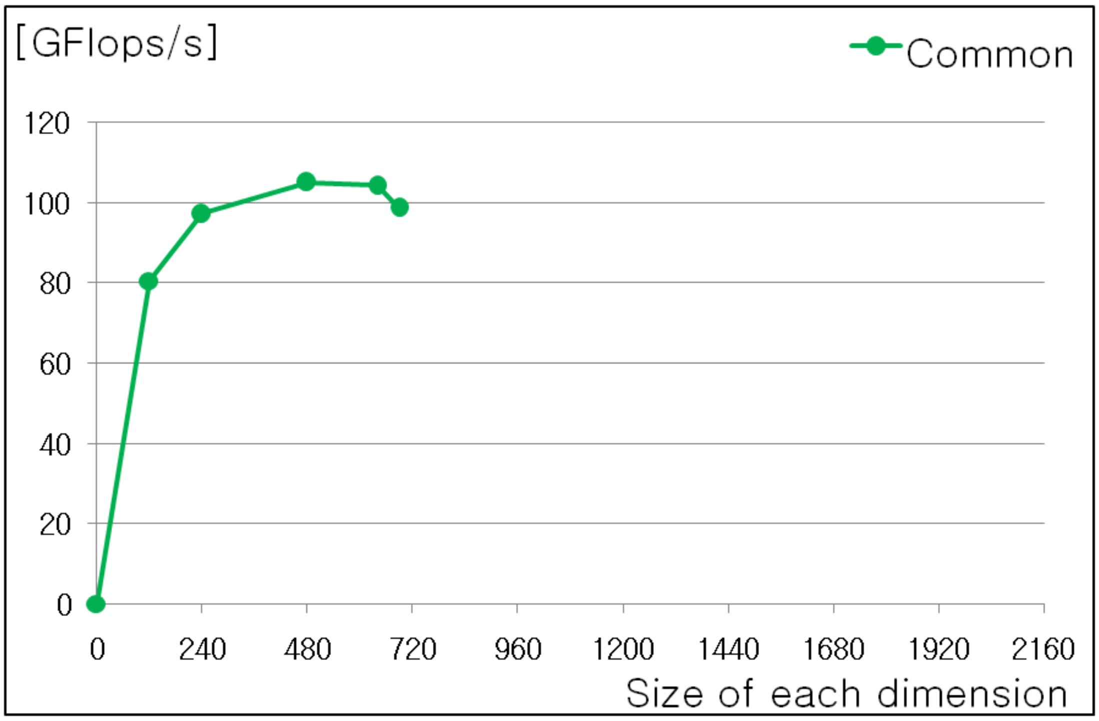


Fig.27. Evaluation of common way on TSUBAME2.0

As it uses double buffering method, the domain should smaller than 1/2 of GPU memory capacity. For Tesla M2050, Dx×Dy×Dz×2 <= 700×700×1400 in floating precision case.

4.3 Existing: 1D-N method

We implement the 1D-decomposition version of naive method [31] which is called as 1D-N method. Different from the original naive method, it only separates the domain by one dimension. So, it is called as 1D-N. The whole domain is separated into sub-domains by 1D decomposition. In 3D domain case, it separates the whole domain by Z dimension. In 2D case, it can separate the domain by Y dimension. If there is no data dependence, it can compute multiple time steps on the GPU side. If there is data dependence like stencil computation case, it can only compute 1 time step on GPU side.

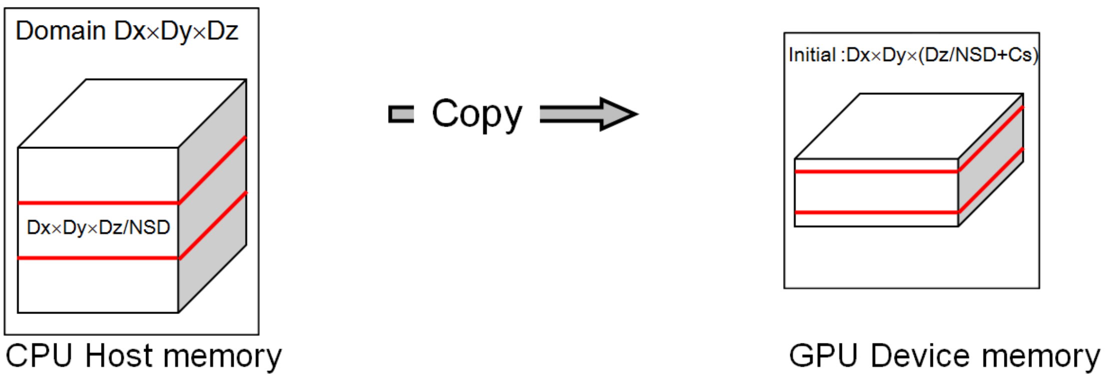


Fig.28. Copy initial to GPU side

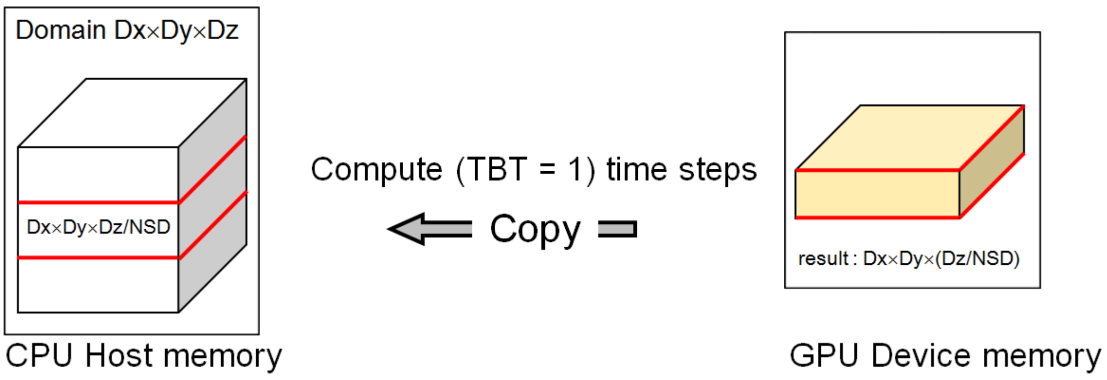


Fig.29. Compute 1 time step and copy the result back

1D-N can only compute one time step. Then, it needs to copy the result back and copy the next sub-domain to the GPU side. So, to compute the domain for one time step, it has to copy each sub-domain to the GPU and copy the result back to the CPU at each time step. This frequent communication reduces the performance. As the computation of single time step is faster than the communication between CPU and GPU, the total time is occupied by the communication cost. After all of the sub-domains are computed, it can continue to compute the next time step. So, it causes frequent communication between CPU and GPU. The process of 1D-N is below:

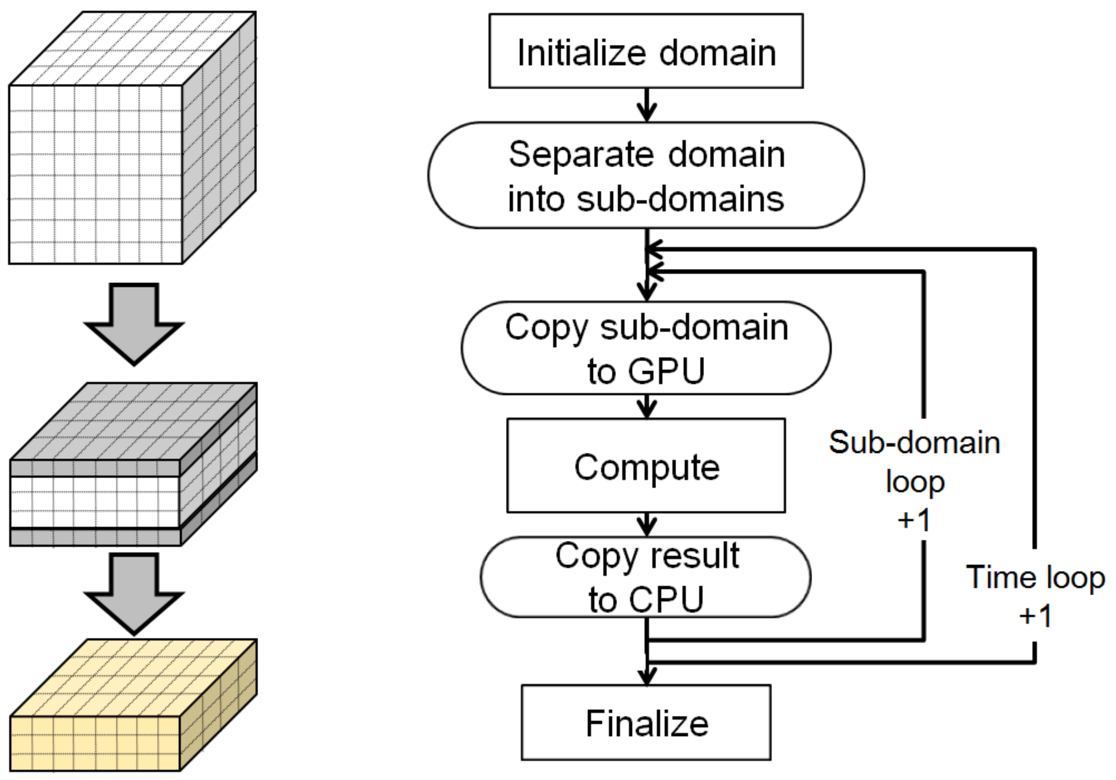


Fig.30. Process of 1D-N method

As the figure shows, after it computes all sub-domains, it can continue to computes the next time step. We give the pseudo code of 7-point stencil as below:

```
Allocate Grid on CPU to read initial of domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += 1)
{
    For (j = 0; j < NSD; j += 1)
    {
        Copy sub-domain j with ghost boundary from CPU to Grid0;
        Compute1 time step and save result to Grid1;
        Copy result of sub-domain j from Grid1 to CPU;
    }
}
```

TSI is the iteration times; NSD is the number of sub-domains. If use double buffering, Dx×Dy×(Dz/NSD+Cs)×2 <= GPU memory capacity. We evaluated 1D-N for 7 point stencil on single node of TSUBAME2.0. The domain is from 240×240×240 to 2160×2160×2160. For Tesla M2050, Dx×Dy×(Dz/NSD+2)×2 <= 700×700×1400 in floating computation case.

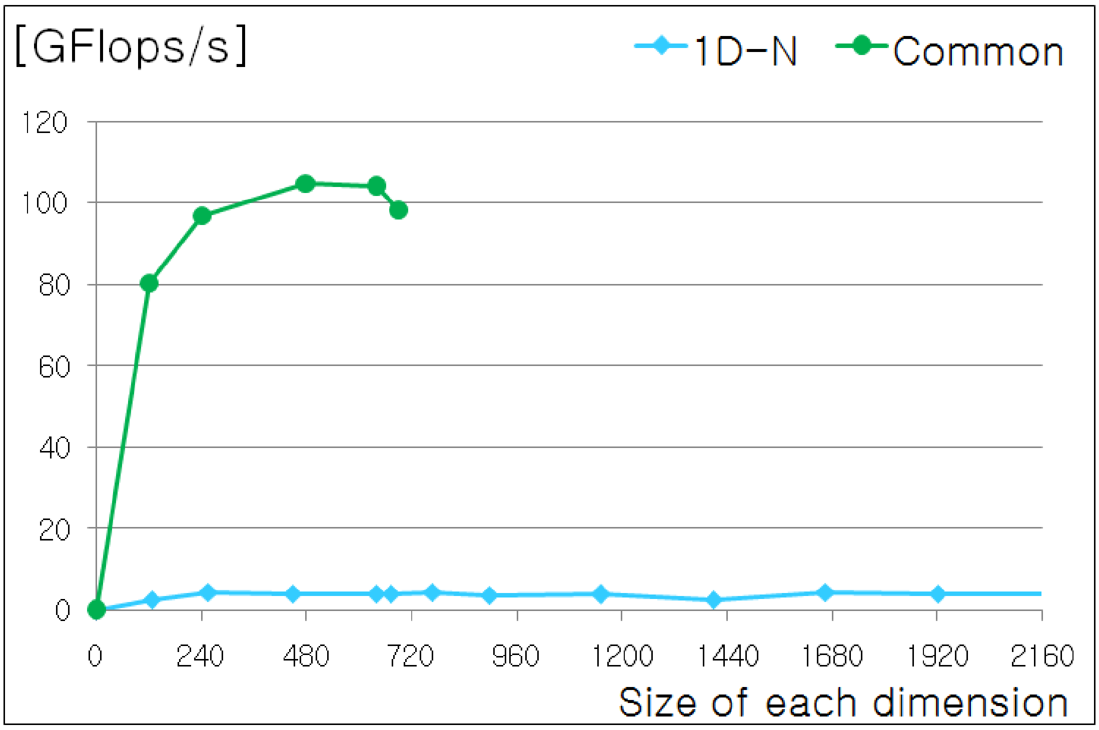


Fig.31. Evaluation on TSUBAME2.0

As we can see in the upper figure, 1D-N can compute big domain. But, it has low performance because of the frequent communication between CPU and GPU. We should reduce the communication cost between CPU and GPU to improve the performance.

4.4 Existing: 1D-T method

As we introduced, temporal blocking method [25], [26], [27], [28], [29], [30], [31] is an efficient way to avoid the communication between CPU and GPU. The main point of this method is to send more information that is needed to GPU. Then, it computes multiple time steps on the GPU side. In stencil case, it copies more boundary points. For example, as below figure shows, the domain is Dx×Dy×Dz. Each sub-domain is Dx×Dy×Dz/NSD. It first copy initial Dx×Dy×(Dz/NSD+Cs×TBT). Then it can compute TBT time steps and get the result of sub-domain Dx×Dy×Dz/NSD. By this way it can compute multiple time steps on the GPU side. It sends sub-domain with more ghost boundaries as initial to GPU side. To compute more time steps on GPU side, it should copy more ghost boundaries.

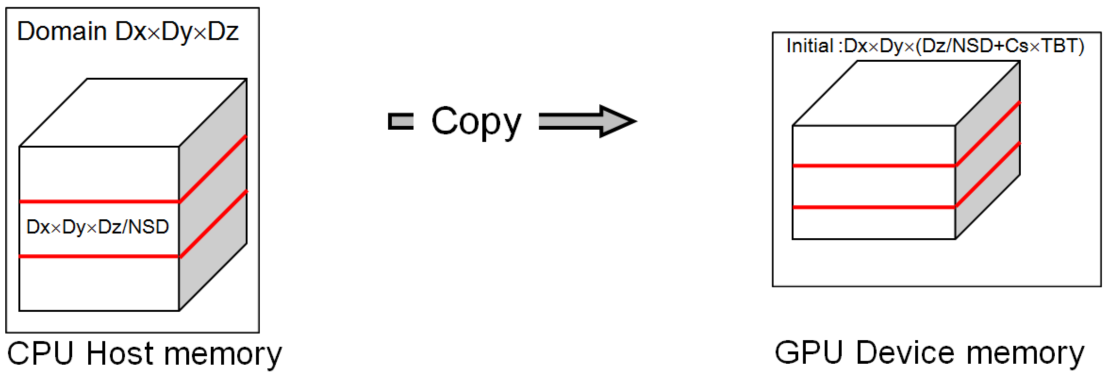


Fig.32. Copy bigger initial to GPU side

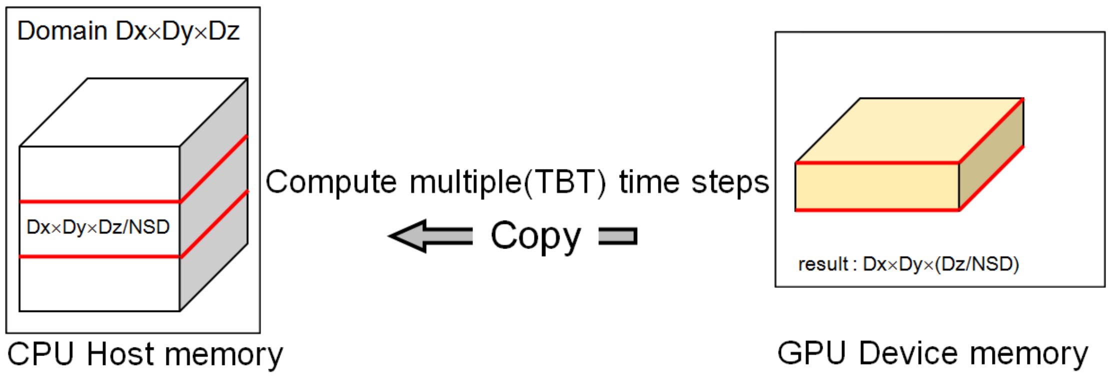


Fig.33. 1D-T can compute multiple time steps on the GPU side.

Both of the ghost boundaries and the sub-domain consume space on the GPU side. More ghost boundaries mean more time steps can be computed on the GPU side. Yet, it decreases the size of sub-domain which makes it separate the whole domain into smaller sub-domains. The ghost boundaries are only needed to continue the computation of sub-domain. So we should balance the number of ghost boundaries and the size of sub-domain to get higher performance.

For bigger domain, the size of each dimension is increased. So, it needs to separate the domain into more sub-domains to fit the GPU memory capacity. Ghost boundaries become bigger as the size of plane grows. So it can contain less ghost boundaries. The memory limitation remains the same. So, it needs to rebalance both sub-domain and the ghost boundaries. It has to separate the domain into smaller sub-domains and contain less the ghost boundaries. The process of this method is below:

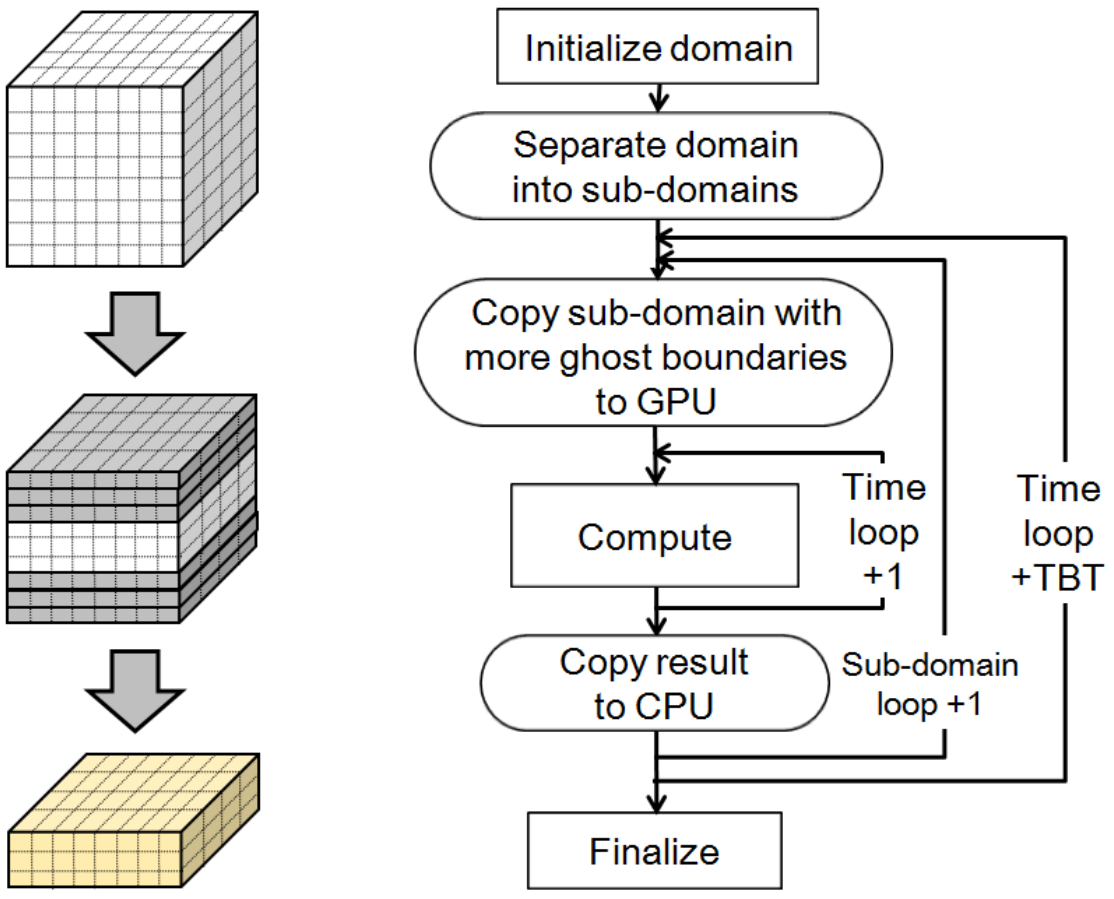


Fig.34. 1D-T method

As the upper figure shows, it first needs to separate the domain into sub-domains. Then, it needs to copy each sub-domain with more ghost boundaries to GPU side. On the GPU side, it can compute multiple time steps. Then, it copies result back to the CPU side. It copies next sub-domain with more ghost boundaries to continue the computation. It needs to copy more ghost boundaries for each sub-domain. As the object is to compute the sub-domains, the ghost boundaries are only for the temporal blocking method. The redundant communication and computation of ghost boundaries degrade the performance. More ghost boundaries cause more redundant cost. For a bigger domain, there are more sub-domains and ghost boundaries which cause more redundant cost. So, it needs to consider the cost of communication and ghost boundaries. We give the redundant cost of 3 time steps computation as below.

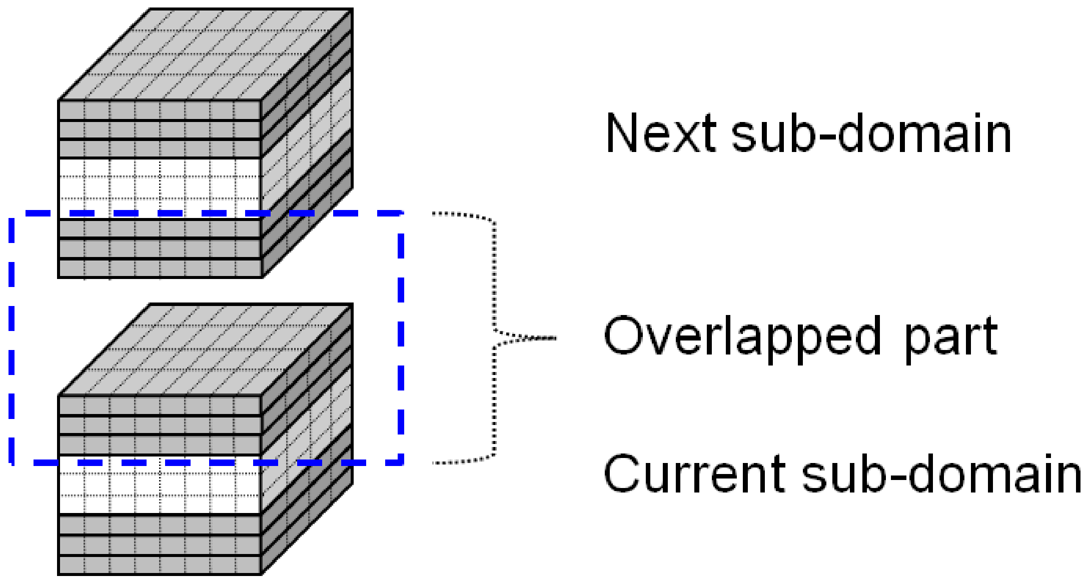


Fig.35. Redundant cost of 1D-T method

We give the example of 7-point stencil as below:

```
Allocate Grid on CPU to read initial of domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += TBT) {
  For (j = 0; j < NSD; j += 1){
    Copy sub-domain j with more ghost boundaries from CPU to Grid0;
    For(k = 0; k<TBT; k+=1){
      Compute1 time step and save result to Grid1;
      Swap Grid0 and Grid1;
    }
    Swap Grid0 and Grid1;
    Copy result of sub-domain j from Grid 1 to CPU;
}}}
```

Here TSI is the iteration times; NSD is number of sub-domains; TBT is temporal blocking times. If use double buffering, $Dx \times Dy \times (Dz/NSD+Cs \times TBT) \times 2 <=$ GPU memory capacity. We evaluated the 1D-T method for 7 point stencil on single node of

TSUBAME2.0. The domain is from 240×240×240 to 2160×2160×2160. 1D-T can compute bigger domain and has high performance. For Tesla M2050, Dx×Dy×(Dz/NSD+2×TBT) × 2 <=700×700×140 in floating computation case.

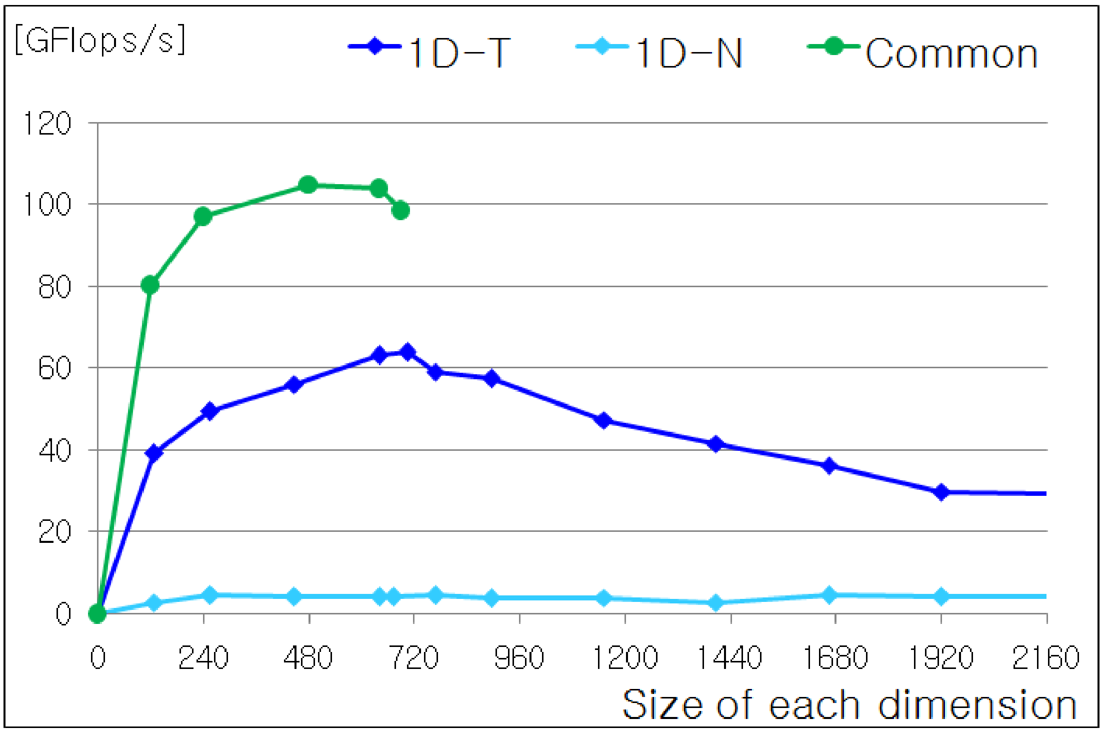
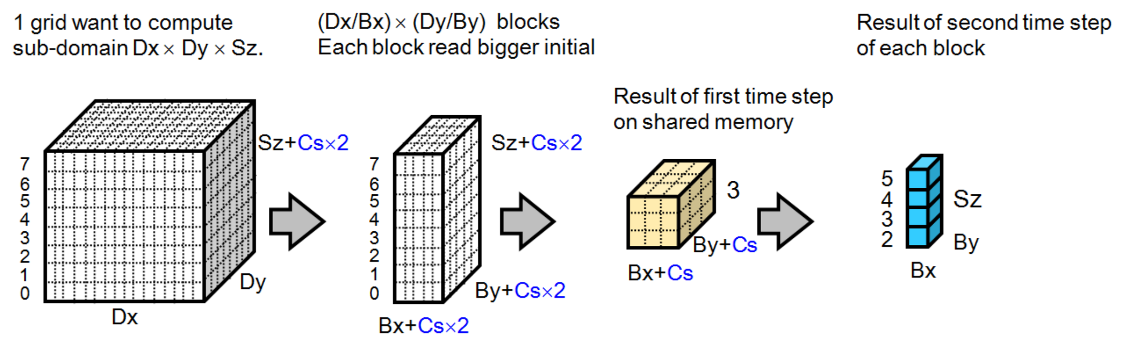


Fig.36. Evaluation on TSUBAME2.0

4.5 Contribution: 1D-2T method

In this section, we try to adopt the temporal blocking method for kernel [36] to improve the performance of the kernel on GPU side. The main idea of this method is to reduce the access time to the global memory of GPU device. We apply temporal blocking method for GPU kernel to improve the performance. It temporally saves the result into shared memory and reuses them to compute next time step.

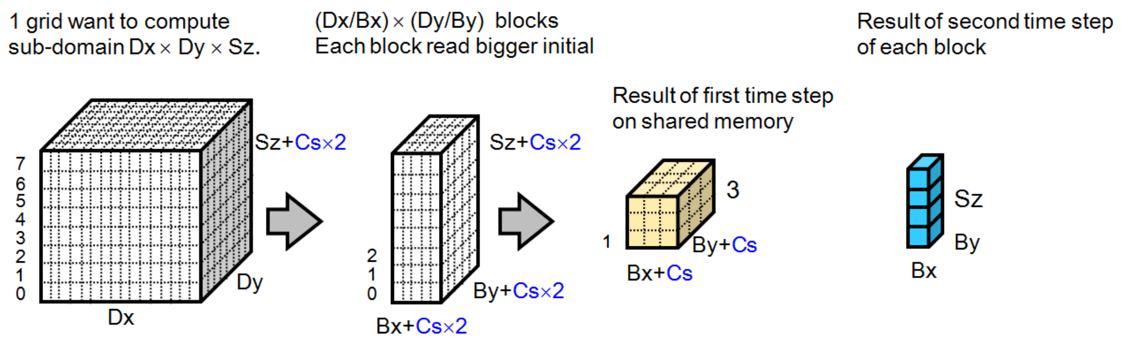
Now we need to determine the number of time steps that can be computed by each kernel. Considering limited capacity of the shared memory of GPU, we configured the number as two. The result of first time step will be temporally saved on the shared memory. By the result on the shared memory, it can compute the next time step. By efficiently reusing the space of shared memory, it can continue the computation to get the result of second time step. We allocate the space on the shared memory to contain three small XY-planes for 7-point stencil computation as below:



1 grid want to compute sub-domain Dx × Dy × Sz.
(Dx/Bx) × (Dy/By) blocks Each block read bigger initial
Result of second time step of each block
Result of first time step on shared memory
Sz+Cs×2
7 6 5 4 3 2 1 0
Dy
Dx
Sz+Cs×2
7 6 5 4 3 2 1 0
By+Cs×2
Bx+Cs×2
3
By+Cs
Bx+Cs
5 4 3 2
Sz
By
Bx

Fig.37. Allocate space to contain 3 small XY-planes.

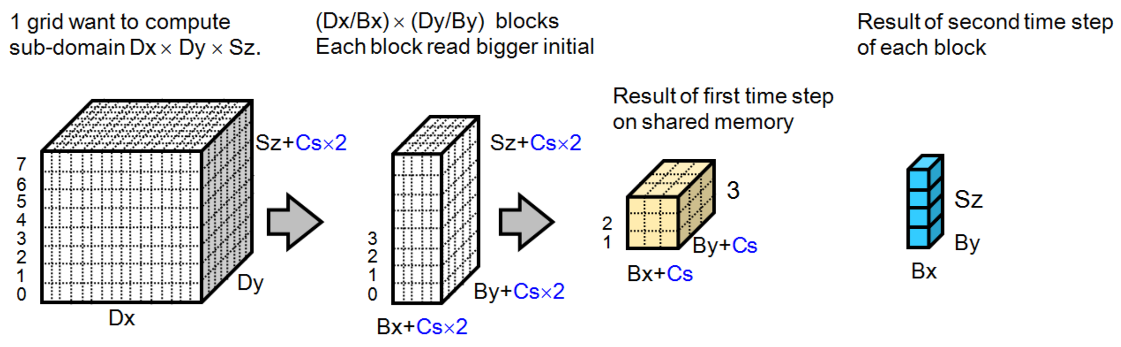
The initial should be bigger than the final result area to continue the computation.



1 grid want to compute sub-domain Dx × Dy × Sz.
(Dx/Bx) × (Dy/By) blocks Each block read bigger initial
Result of second time step of each block
Result of first time step on shared memory
Sz+Cs×2
7 6 5 4 3 2 1 0
Dy
Dx
Sz+Cs×2
2 1 0
By+Cs×2
Bx+Cs×2
3
1
By+Cs
Bx+Cs
Sz
By
Bx

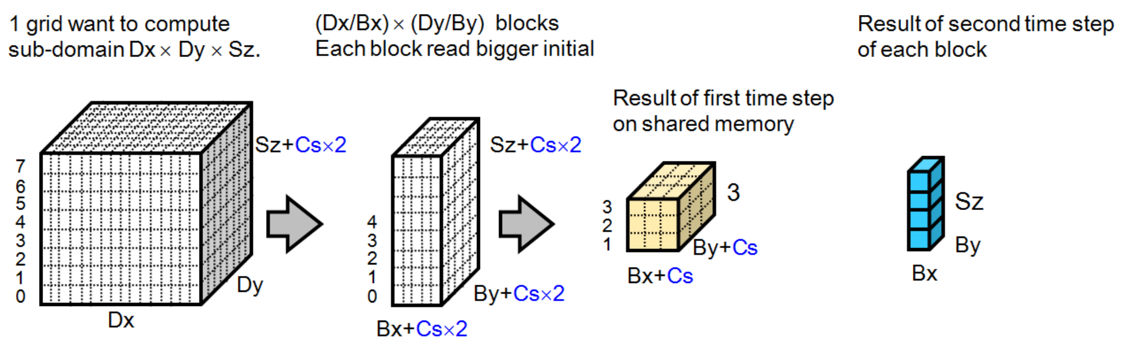
Fig.38. Each block uses 3 bigger XY-planes to compute 1 time step to shared memory.

For example, to compute Dx×Dy×Sz domain in upper figure, we should copy initial Dx×Dy×(Sz+Cs×2). Each block read small XY-planes (0,1,2) which size is (Bx+Cs×2)×(By+Cs×2) as initial. Then, it can compute the plane 1 of time first time step which size is (Bx+Cs)×(By+Cs) .



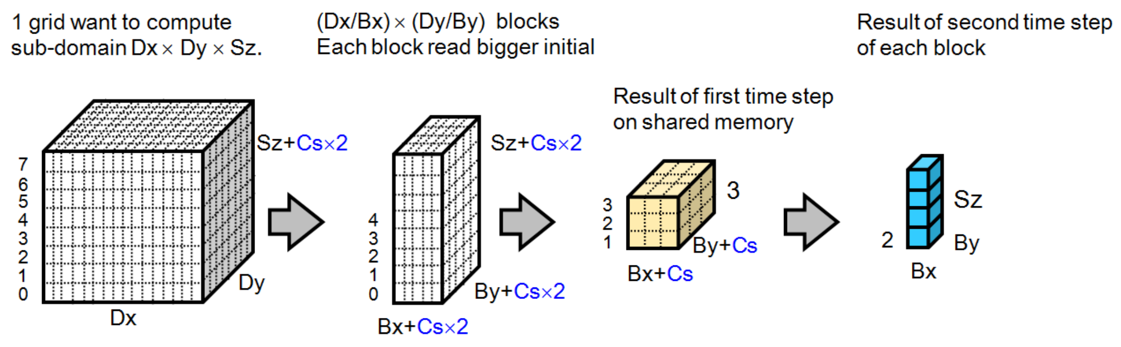
1 grid want to compute sub-domain Dx × Dy × Sz.
(Dx/Bx) × (Dy/By) blocks Each block read bigger initial
Result of second time step of each block
Result of first time step on shared memory
Sz+Cs×2
7 6 5 4 3 2 1 0
Dy
Dx
Sz+Cs×2
3 2 1 0
By+Cs×2
Bx+Cs×2
3
2 1
By+Cs
Bx+Cs
Sz
By
Bx

Fig.39. Continue to compute small XY-planes to shared memory



1 grid want to compute sub-domain Dx × Dy × Sz.
(Dx/Bx) × (Dy/By) blocks
Each block read bigger initial
Result of second time step of each block
Result of first time step on shared memory
7 6 5 4 3 2 1 0
Sz+Cs×2
Dy
Dx
4 3 2 1 0
Sz+Cs×2
By+Cs×2
Bx+Cs×2
3 2 1
3
By+Cs
Bx+Cs
Sz
By
Bx

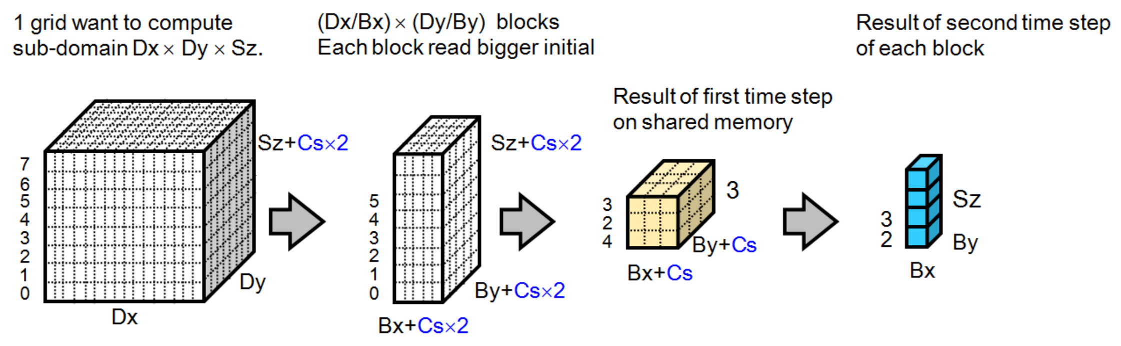
Fig.40. When the number of smaller XY-planes is 3, it can compute the second time step



1 grid want to compute sub-domain Dx × Dy × Sz.
(Dx/Bx) × (Dy/By) blocks
Each block read bigger initial
Result of second time step of each block
Result of first time step on shared memory
7 6 5 4 3 2 1 0
Sz+Cs×2
Dy
Dx
4 3 2 1 0
Sz+Cs×2
By+Cs×2
Bx+Cs×2
3 2 1
3
By+Cs
Bx+Cs
2
Sz
By
Bx

Fig.41. Compute the second time step by using data on the shared memory.

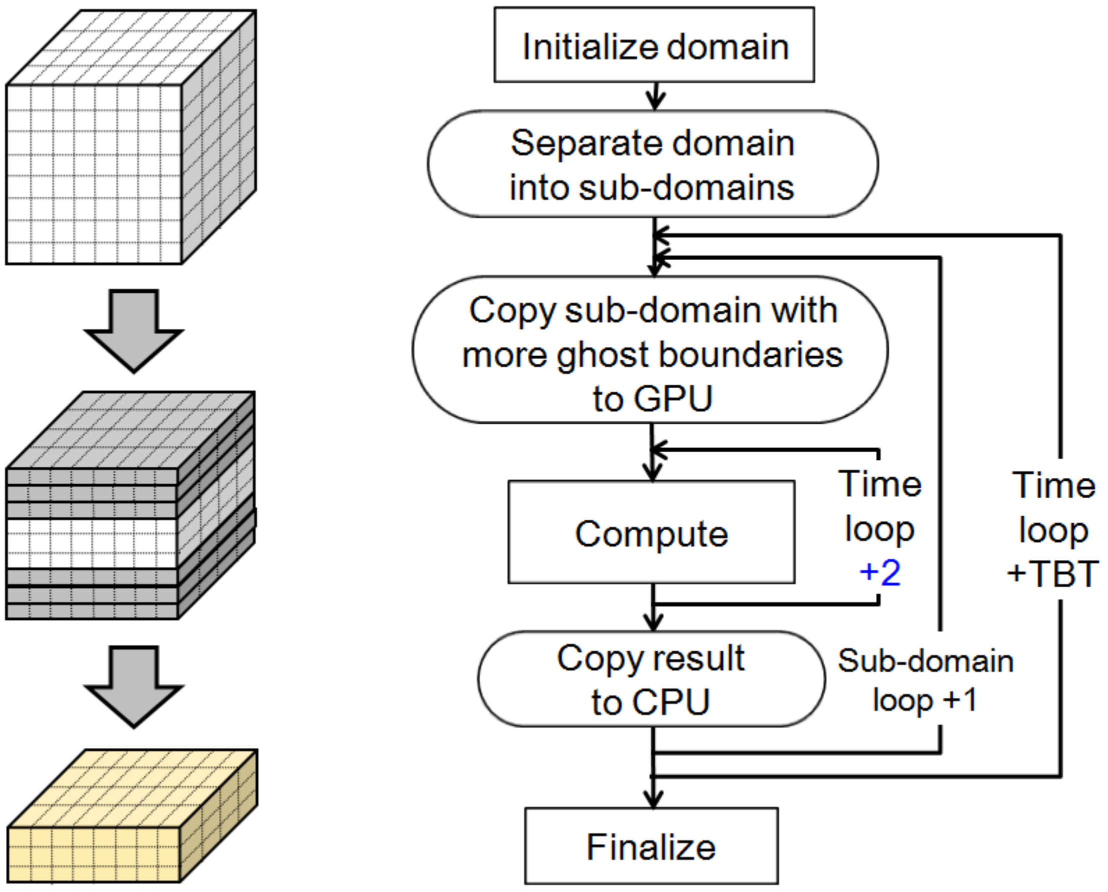
As the memory capacity of shared memory is limited, it reuses the space of the shared memory as below:



1 grid want to compute sub-domain Dx × Dy × Sz.
(Dx/Bx) × (Dy/By) blocks
Each block read bigger initial
Result of second time step of each block
Result of first time step on shared memory
7 6 5 4 3 2 1 0
Sz+Cs×2
Dy
Dx
5 4 3 2 1 0
Sz+Cs×2
By+Cs×2
Bx+Cs×2
3 2 4
3
By+Cs
Bx+Cs
3 2
Sz
By
Bx

Fig.42. Reuse the space of the shared memory to continue the computation.

If it uses temporal blocking method for kernel, it only improves the performance of kernel. It computes 2 time steps by 1 kernel. If we use temporal blocking method for kernel to 1D-T, we call the method as 1D-2T.



As the capacity of shared memory is 32×32×3 currently. We can get the pseudo code of 1D-2T method as below:

```
Allocate Grid on CPU to read initial of domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += TBT)
{
  For (j = 0; j < NSD; j += 1)
  {
    Copy sub-domain j with more ghost boundaries from CPU to Grid0;
    For(k = 0; k<TBT; k+=2){
      Compute 2 time steps by one kernel and save result to Grid1;
      Swap Grid0 and Grid1;
    }
    Swap Grid0 and Grid1;
```

```
        Copy result of sub-domain j from Grid 1 to CPU;
    }
}
```

If use double buffering, Dx×Dy×(Dz/NSD+Cs×TBT) ×2 <= GPU memory capacity. We evaluated the 1D-2T method for 7 point stencil on single node of TSUBAME2.0. The domain is from 240×240×240 to 2160×2160×2160. 1D-2T can compute bigger domain and it achieves 1.16 times higher performance than 1D-T. For Tesla M2050, Dx×Dy× (Dz/NSD+2×TBT)×2 <=700×700×140 in floating computation case.

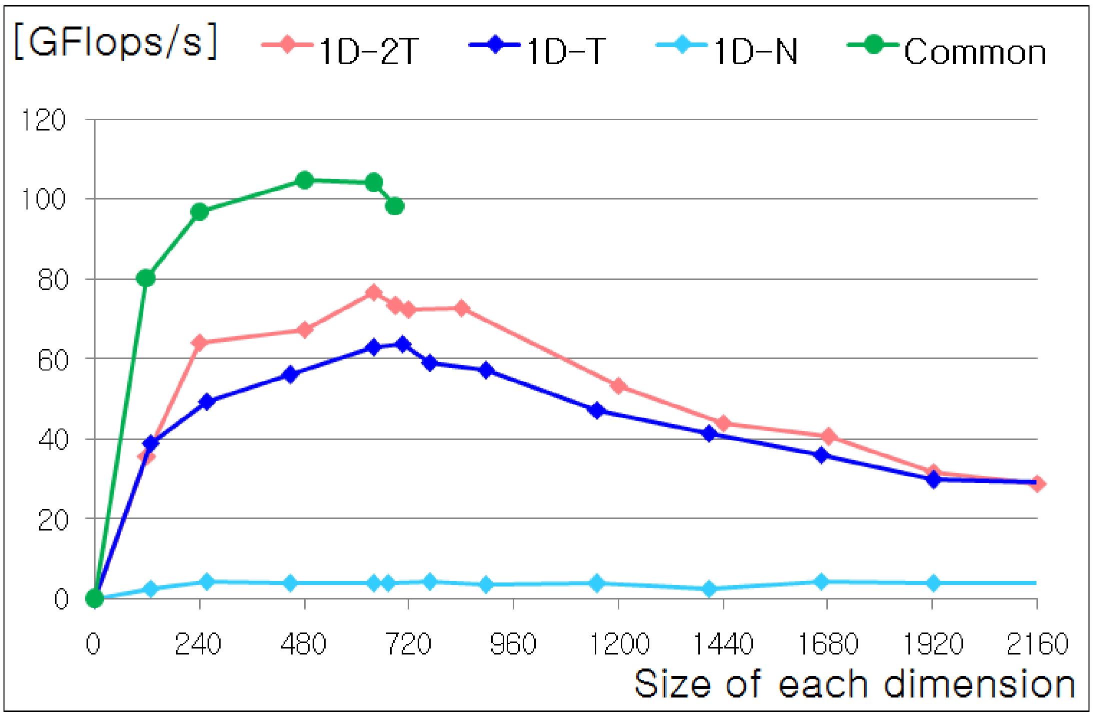


Fig.43. Evaluation on TSUBAME2.0

4.6 Contribution: 1D-TBM method

4.6.1 Buffer copy method

Temporal blocking method can reduce the communication times to reduce the communication cost. But as we explained, there is redundancy cost because of the more ghost boundaries. The redundancy cost is to compute multiple time steps on the GPU side. The more ghost boundaries cause more redundant cost. As the more ghost boundaries are essential to reduce the communication cost, we should focus on how to solve the redundancy problem. We found that there are overlapped part between ghost boundaries and sub-domains. It computes same part multiple times on the device memory. So, we think about how to reduce this redundant cost by saving and reusing the overlapped part.

The overlapped part is copied and computed twice. So, we consider if there is a way to reuse them to avoid the overlapped work. As the overlapped part has been computed when computing current sub-domain, we consider containing some result on the GPU side. When computing next sub-domain that has the overlapped part, it can reuse saved result to avoid overlapped work. Still, we need to find the least XY-planes that need to save and reuse. As the reused part is stored on the GPU side, it also needs to allocate space for the reused part. Those reused part compete space with sub-domain and ghost boundaries which may degrade the performance of temporal blocking. So, we need to balance the space for reused part, sub-domain and the ghost boundaries.

We first consider how to save and reuse those overlapped part to solve redundancy problem. The 7-point stencil computation consumes 2 XY-planes at each time steps, so we try to reuse 2 XY-planes and supply them to next sub-domain. We explain this step by step. When we compute current sub-domain, at time step 0, we save two XY-planes to the buffer on GPU along the border line as the below figure shows. When computing the next time step, we also save two XY-planes along the border line. This phase will continue until the last time step. So, it first allocates a buffer for those XY-planes on the GPU side as below:

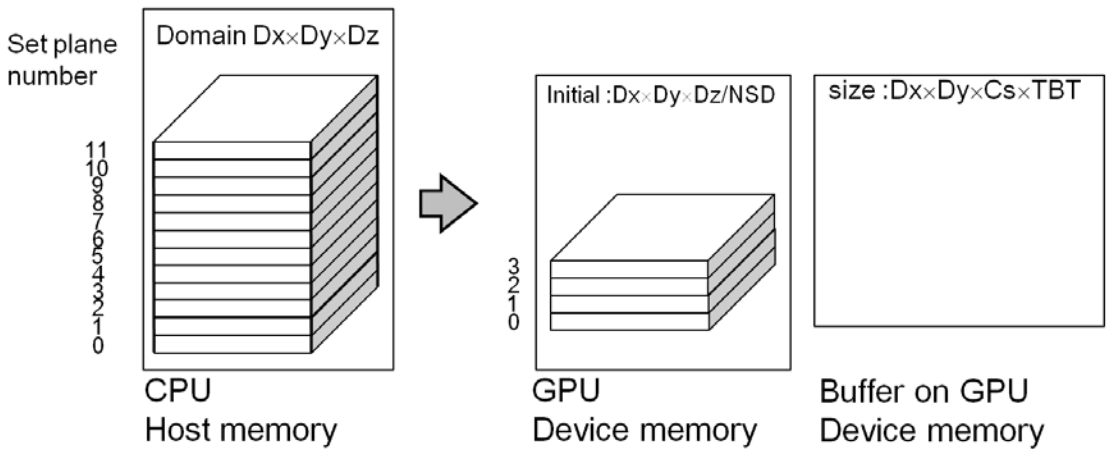


Fig.44. It first allocates a buffer on the GPU side

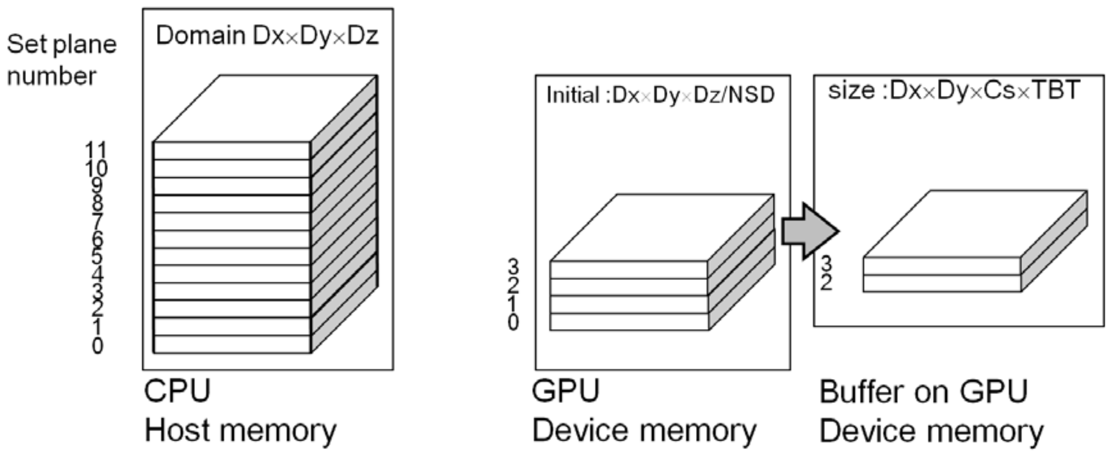


Fig.45. It saves XY-planes (2, 3) to the buffer at time step 0.

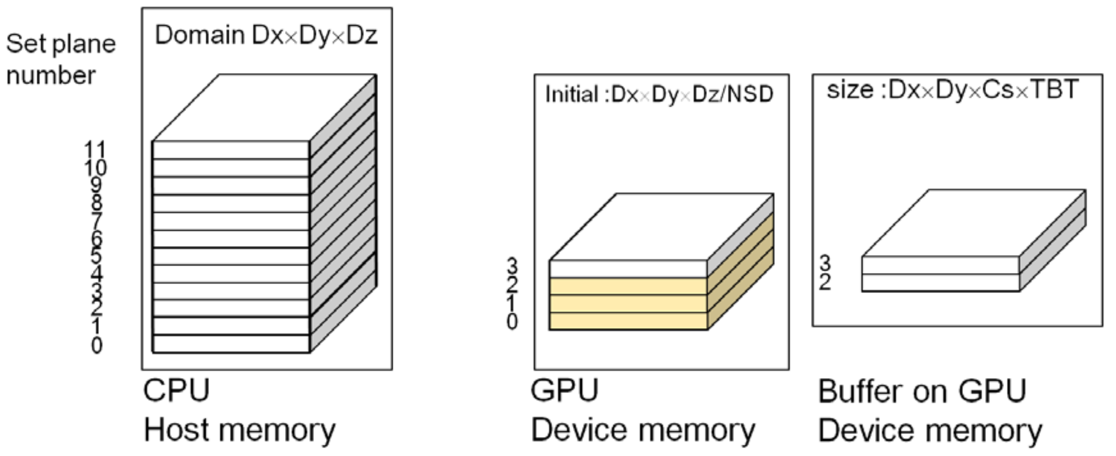


Fig.46. Then, it computes time step 1.

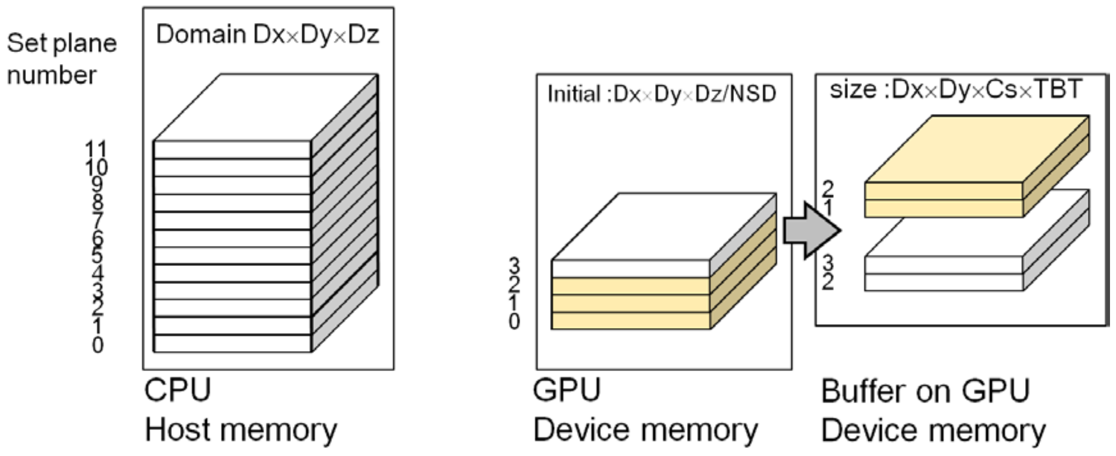


Fig.47. It also saves XY-planes (1, 2) to the buffer on GPU.

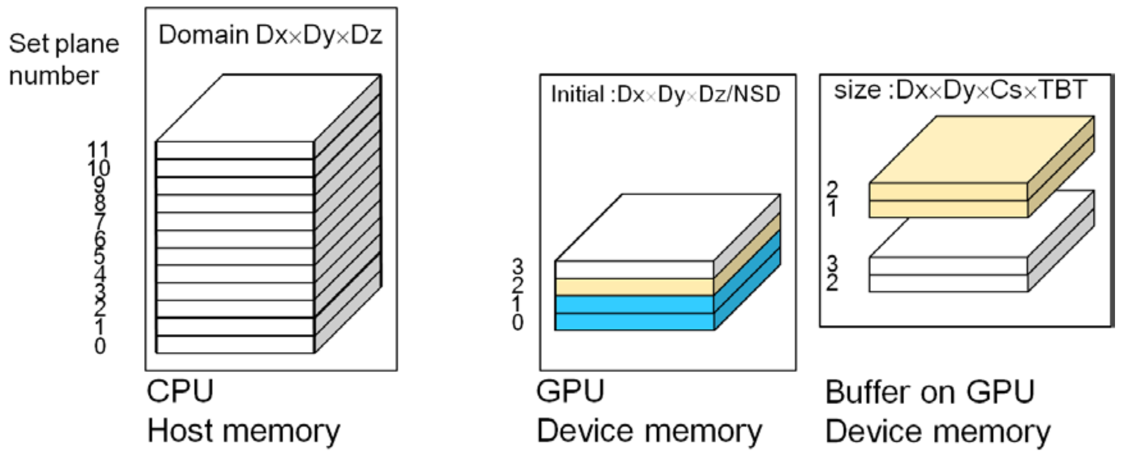


Fig.48. Then, it computes time step 2.

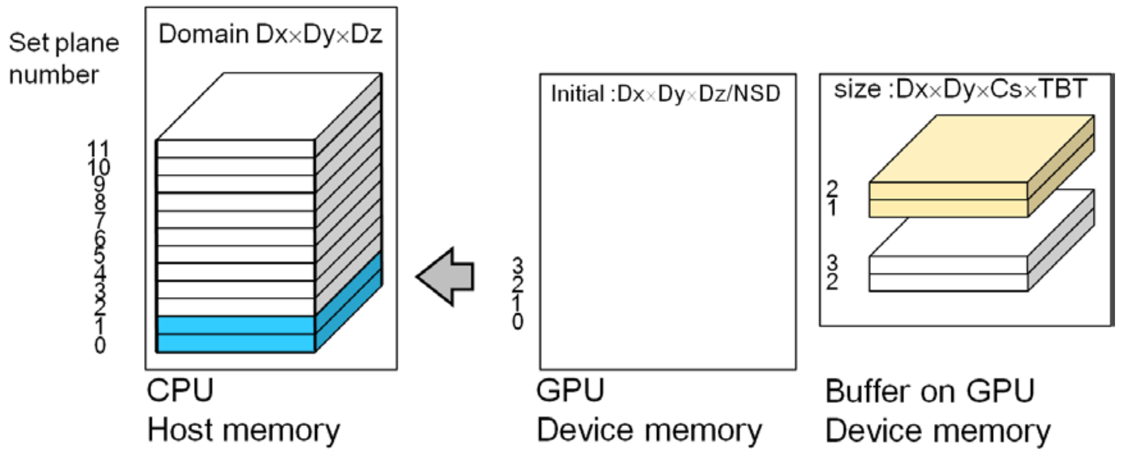


Fig.49. It copies the result back to CPU side.

When computing the next sub-domain, at time step 0, we copy sub-domain1 from CPU to GPU as below figure shows. Then we read two XY-planes from the buffer on the GPU. We compute time step 0 with the sub-domain and the two XY-planes from buffer. So, we get the result of time step 0. Then we also read two XY-planes of time step 1 from buffer. With those XY-planes, it can continue to compute time step 1. We continue this kind of method until last time step.

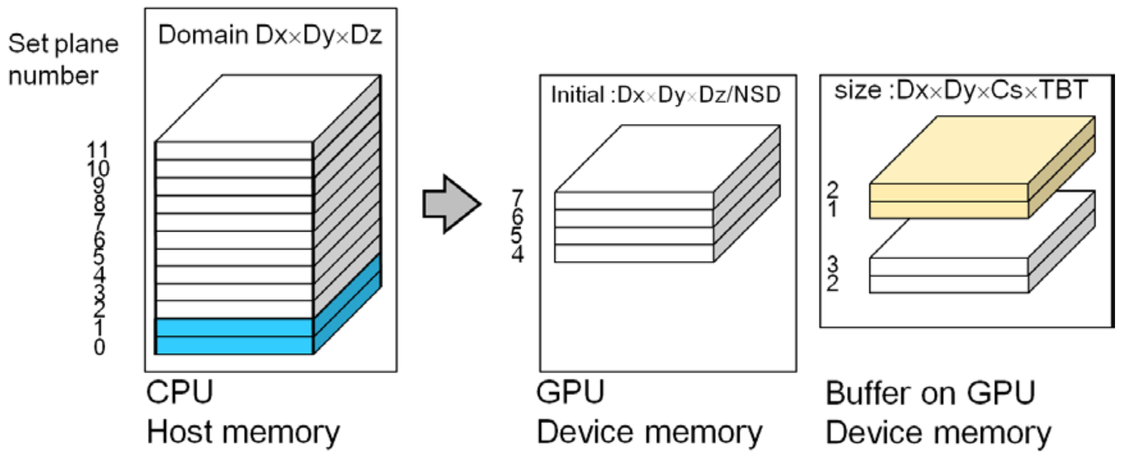


Fig.50. It copies the initial XY-planes (4,5,6,7) of next sub-domain

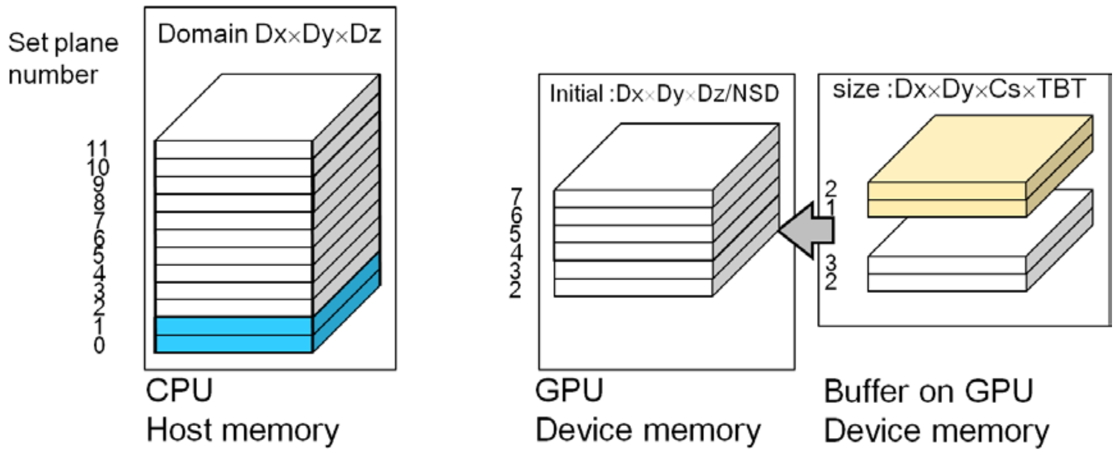


Fig.51. It reads XY-planes(2,3) from buffer

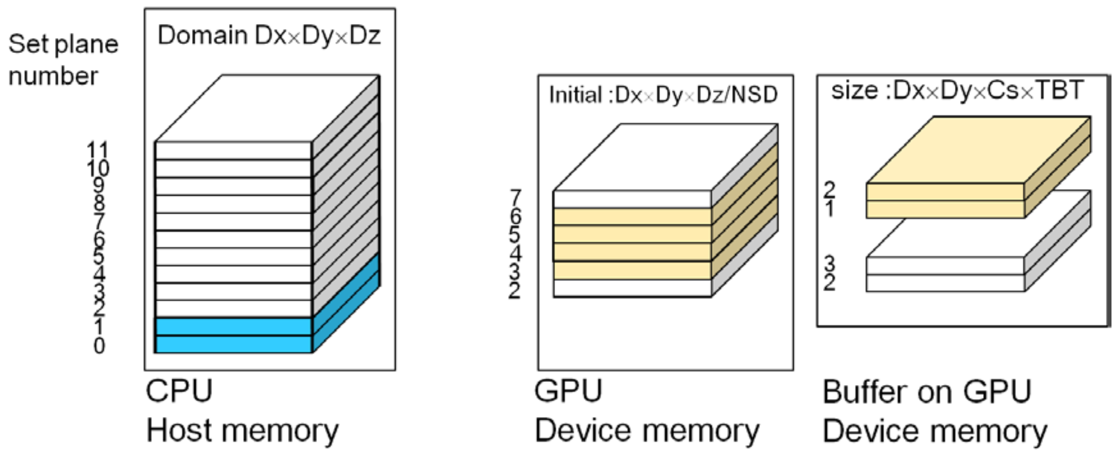


Fig.52. It computes time step 1.

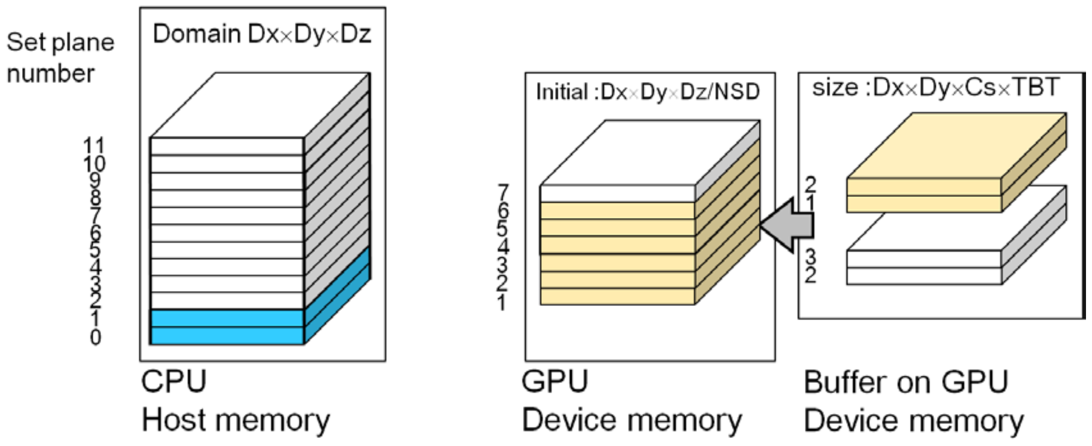


Fig.53. It reads XY-planes (1, 2) of time step 1 from buffer.

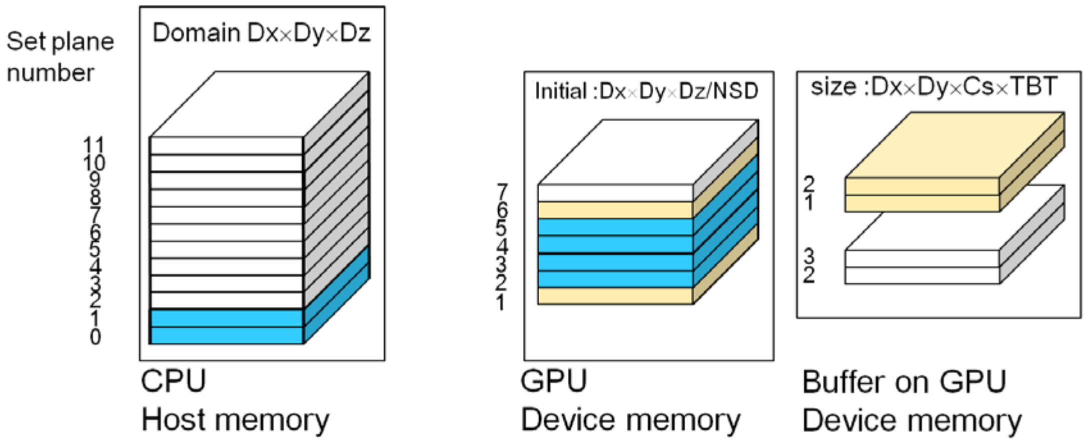


Fig.54. It computes time step2

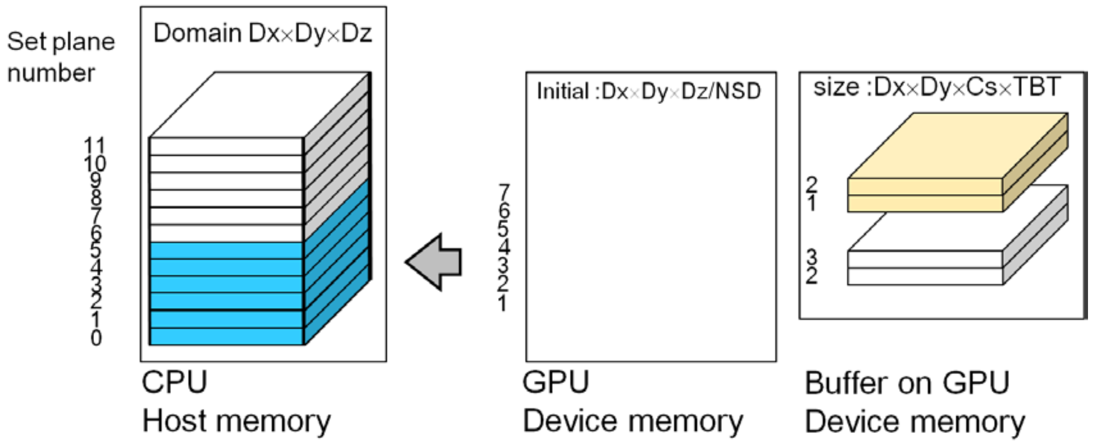


Fig.55. It copies the result back to CPU side.

As it reads two XY-planes along the border line and the stencil computation consumes two XY-planes, it can continue the computation till to get the final result. The next figure gives the difference of computation and communication between temporal blocking method and our method. We call this method as buffer-copy method [36], [37]. Note that this buffer-copy method does not degrade computation accuracy, since it stores some result of current and reuse them to solve redundant problem. It can work out the correct result, while it resolves redundancy problem. Therefore, our method can be adopted by other stencil forms. The space for the buffer should big enough to contain the reused XY-planes. The buffer can be reused by next sub-domain. When computing next sub-domain, it first reads two XY-planes from the buffer. After reading 2 XY-planes for current sub-domain, the space for those two XY-planes can be reused for next sub-domain. So it can reuse the buffer for each sub-domain. By this way, it can use less space to perform the buffer-copy method. We call the 1D-T with buffer-copy method as 1D-TB method.

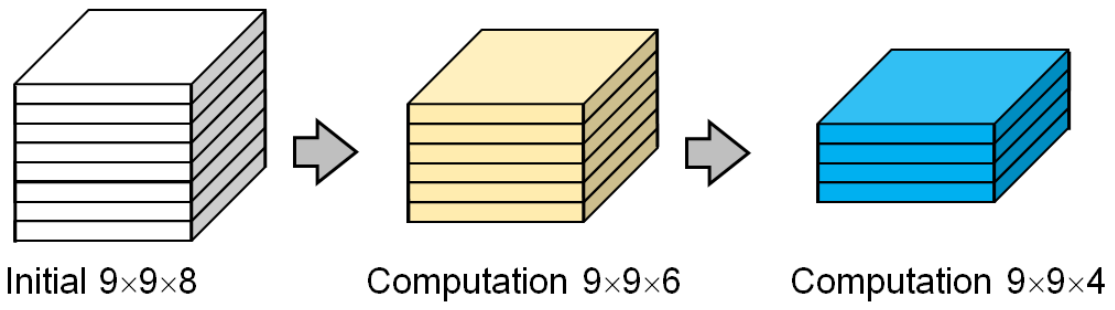


Fig.56. The communication and computation cost of 1D-T on example sub-domain.

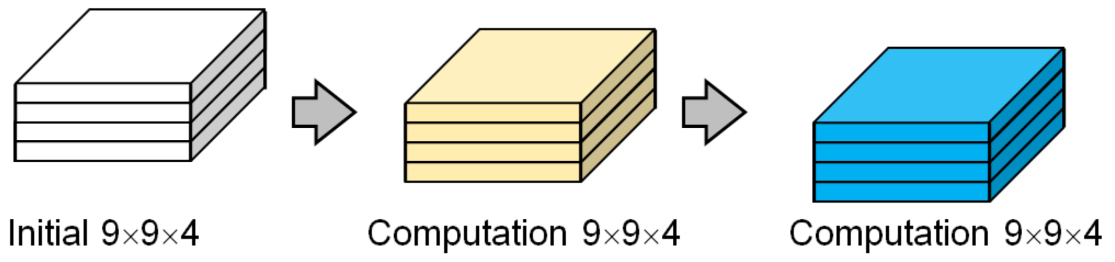


Fig.57. The communication and computation cost of 1D-TB on example sub-domain.

As upper figure shows, the computation and communication cost of 1D-TB is less than 1D-T. The additional cost is to save two XY-planes to the GPU and read two XY-planes from the buffer. Compare with the communication between CPU and GPU, the cost of access to GPU memory is lower. So by this way, it can efficiently solve the redundancy problem. For the first sub-domain, as there is no former sub-domain, it no needs to read from the buffer. But it needs to save two XY-planes to the buffer for next sub-domain. For the last sub-domain, as there is no next sub-domain, it no needs to save two XY-planes to the buffer. But it still needs to read two XY-planes from the buffer. The middle sub-domains need to read two XY-planes from the buffer and save two XY-planes to the buffer at each time step. Then we can give the process as below: it first separates the whole domain into sub-domains. Then, it only copies sub-domain from GPU to CPU. In this phase, it can decrease the communication cost. Then, it reads two XY-planes from the buffer. After it computes 1 time step, it also saves two XY-planes to the buffer. It only accesses to the device memory of GPU and the computation of current time step does not affect the data on the buffer. So it ensures the correctness of the result.

For different kinds of stencil forms, we can save and read related number of XY-planes. For example, 13-point stencil depends on 4 points at each dimension on 3D domain. In 13-point stencil case, it consumes 4 XY-planes at each time step. To continue the computation, it needs to reads 4 XY-planes from the buffer. Also, it needs to save 4 XY-planes to the buffer for next sub-domain. So the number of XY-planes that need to be buffered is decided by the stencil form. Although it needs to access those XY-planes at every time step, compared with the redundant communication of first time step and the redundant computation at every time step, the cost of reading those XY-planes are lower. This is the whole process of the buffer-copy method. With the buffer-copy method, the temporal blocking method can efficiently reduce the communication and computation cost.

4.6.2 Memory-saving method

Double buffering is a common way to perform the stencil computation on the GPU. One grid is assigned to read and the other one is assigned to save. When save to or read from the grid, each point is to save to the same position during the whole iteration. So each XY-plane is to save to the same space of the grid during the whole iteration. As 1D-TB method only computes un-overlapped parts, there is blank on the grid after each time step. Those blanks cannot be reused during the computation of current sub-domain. To save those spaces, we propose memory-saving method [37]. At each time step, it shifts the planes to fill blank as below:

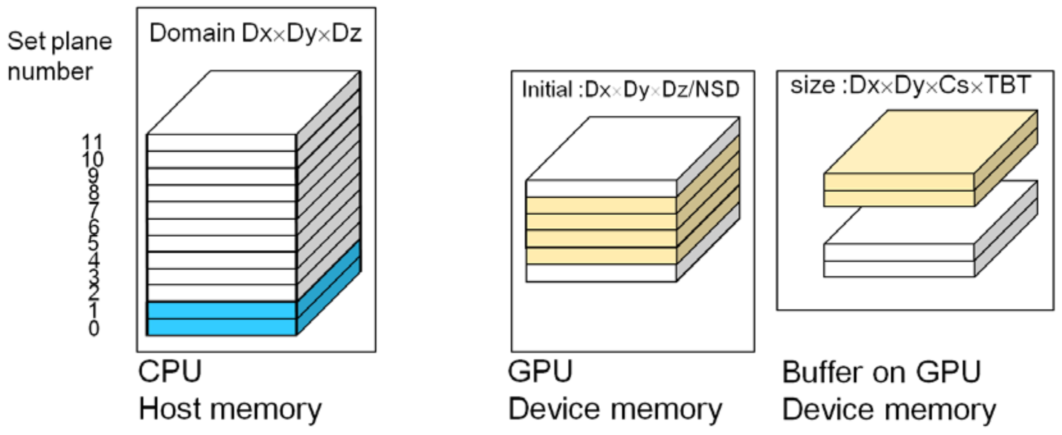


Fig.58. The computation of first time step.

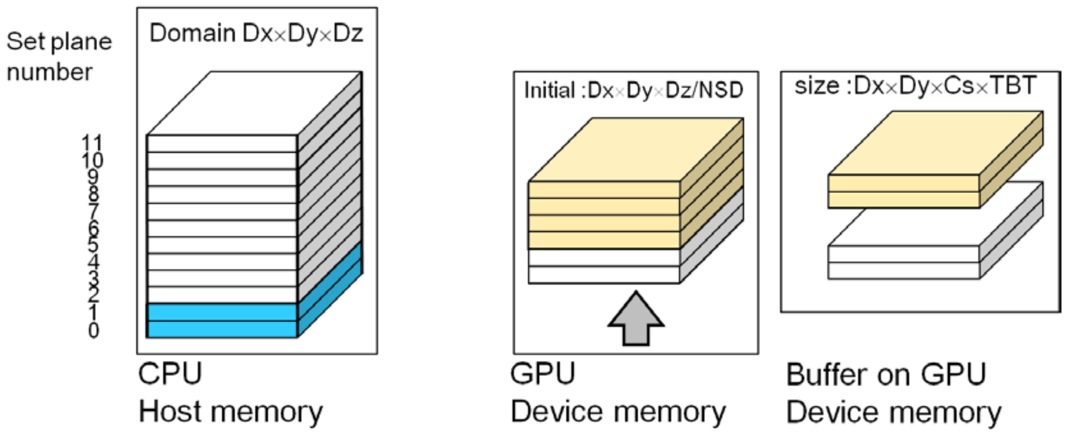


Fig.59. It shifts the result to fill the blank after the computation of each time step.

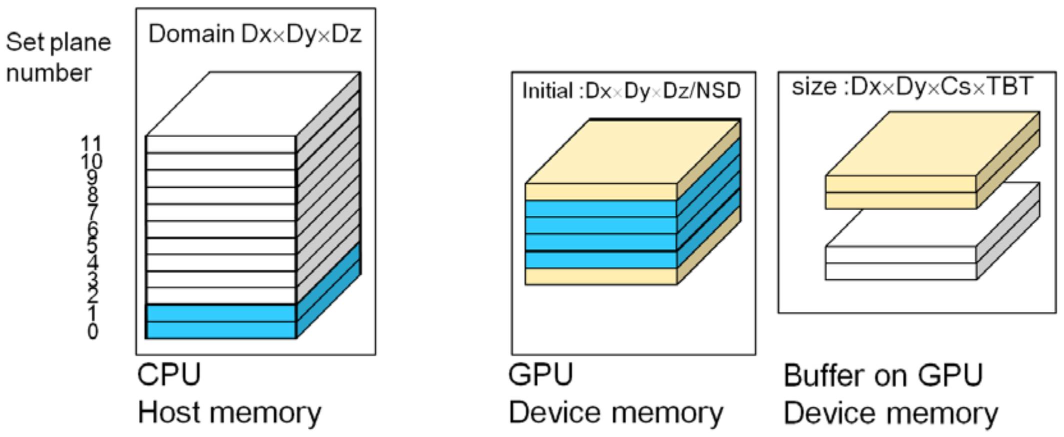


Fig.60. Then, it continues the computation of next time step.

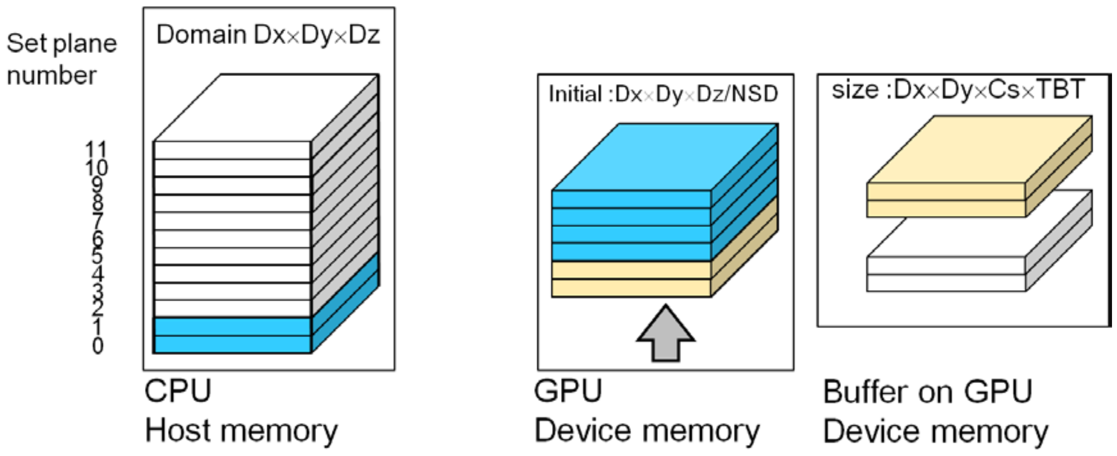


Fig.61. It also shifts the result till final time step.

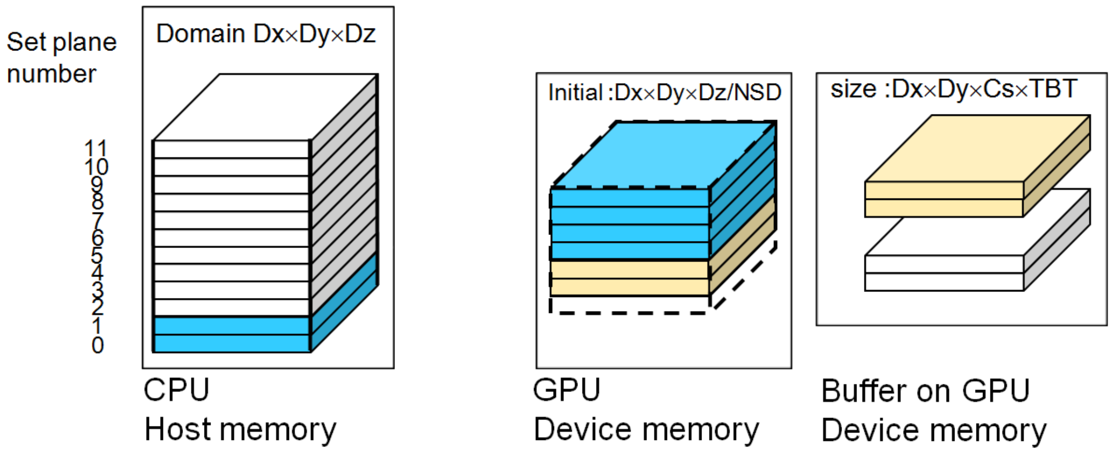


Fig.62. It can save (Cs×TBT-Cs) × Dx × Dy space than 1D-T

We call this method as memory-saving method [37]. Compared with the buffer-copy method, the memory-saving method can save space while get same effect. Saving space is important because we can use saved space to contain more ghost boundaries. The saved space can also be used to contain bigger sub-domain. Both of them can improve the performance. Bigger sub-domain means that it needs fewer phases to compute the whole domain.

Although the cost of buffer-copy method is less than the communication cost between CPU and GPU, it also needs to consider reducing the cost of buffer-copy. We call 1D-TB method with memory-saving method as 1D-TBM method. So, the whole process of 1D-TBM is below:

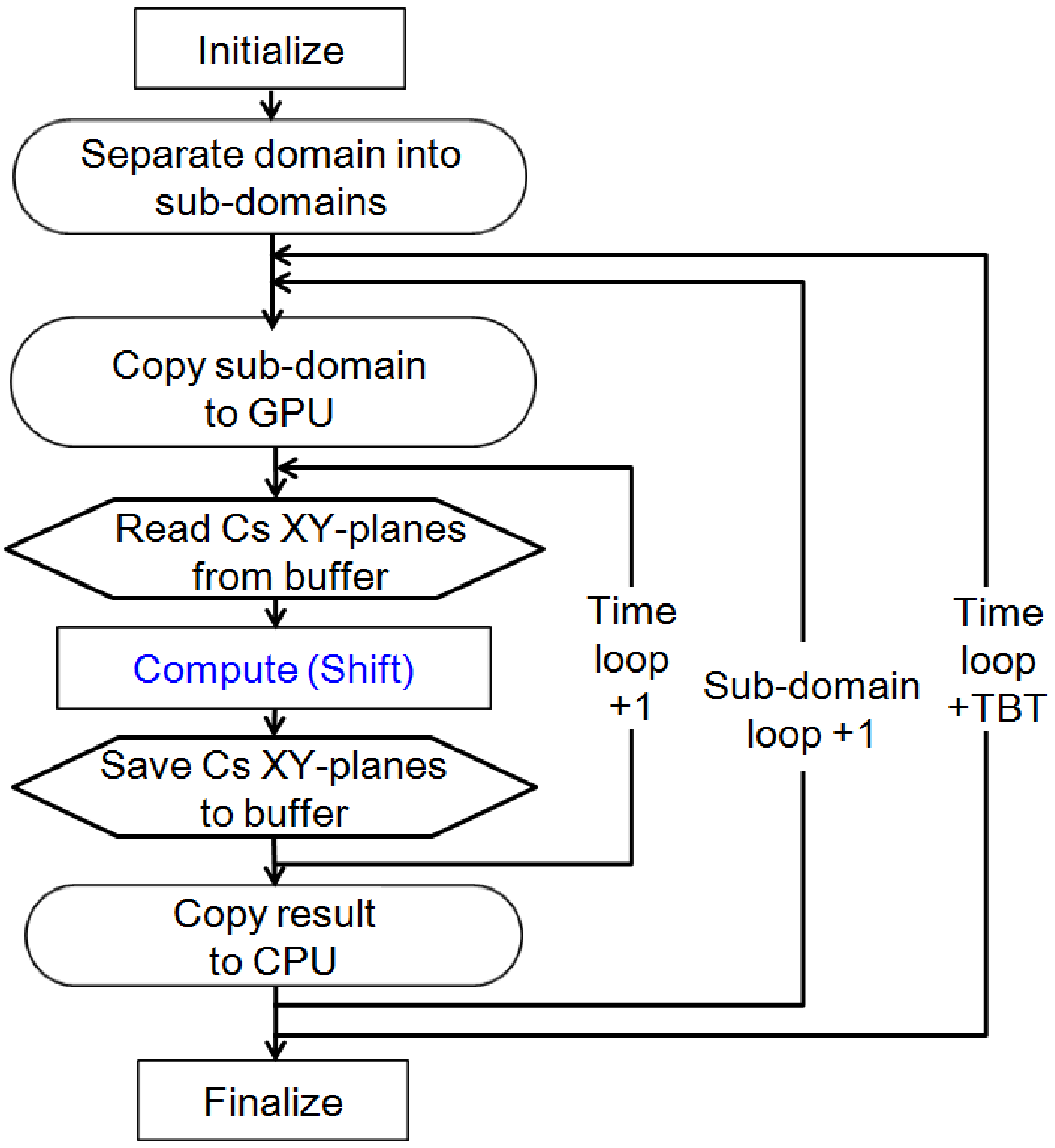


Fig.63. The process of 1D-TBM method

As upper figure shows, it separates the domain into sub-domains. Then it copies sub-domain from CPU to GPU. Then it uses the temporal blocking with buffer-copy and memory-saving method. The difference between with or without memory-saving method is that it shift the XY-planes to fill the blank when saving the result to the global memory.

The buffer-copy method cost less communication and computation. Furthermore, by the memory-saving method, it consumes less space to perform the buffer-copy method. By the temporal blocking method, it can reduce the communication times to reduce the communication cost. So we can give the pseudo code of 1D-TBM method as below:

```
Allocate Grid on CPU to read initial of domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Allocate Buffer on GPU;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += TBT)
{
    For (j = 0; j < NSD; j += 1)
    {
        Copy sub-domain j from CPU to Grid0;
        For(k = 0; k<TBT; k+=1)
        {
            Read Cs XY-planes from Buffer;
            Compute1 time step;
            Save result to the shifted position on Grid1;
            Save Cs XY-planes to Buffer;
            Swap Grid0 and Grid1;
        }
        Swap Grid0 and Grid1;
        Copy result of sub-domain j from Grid1 to CPU;
    }
}
```

4.7 Contribution: 1D-2TBM method

By applying the buffer-copy and memory-saving method to 1D-2T, we can get 1D-2TBM method. As it computes 2 time steps, it needs to read Cs×2 XY-planes from the buffer of GPU. Also it needs to save Cs×2 XY-planes to the buffer of GPU. It reduces the loop time by 2. So the total time step of iteration should be divided by 2. Also the position that needs to be shift is Cs×2 planes. The program model of 1D-2TBM is below:

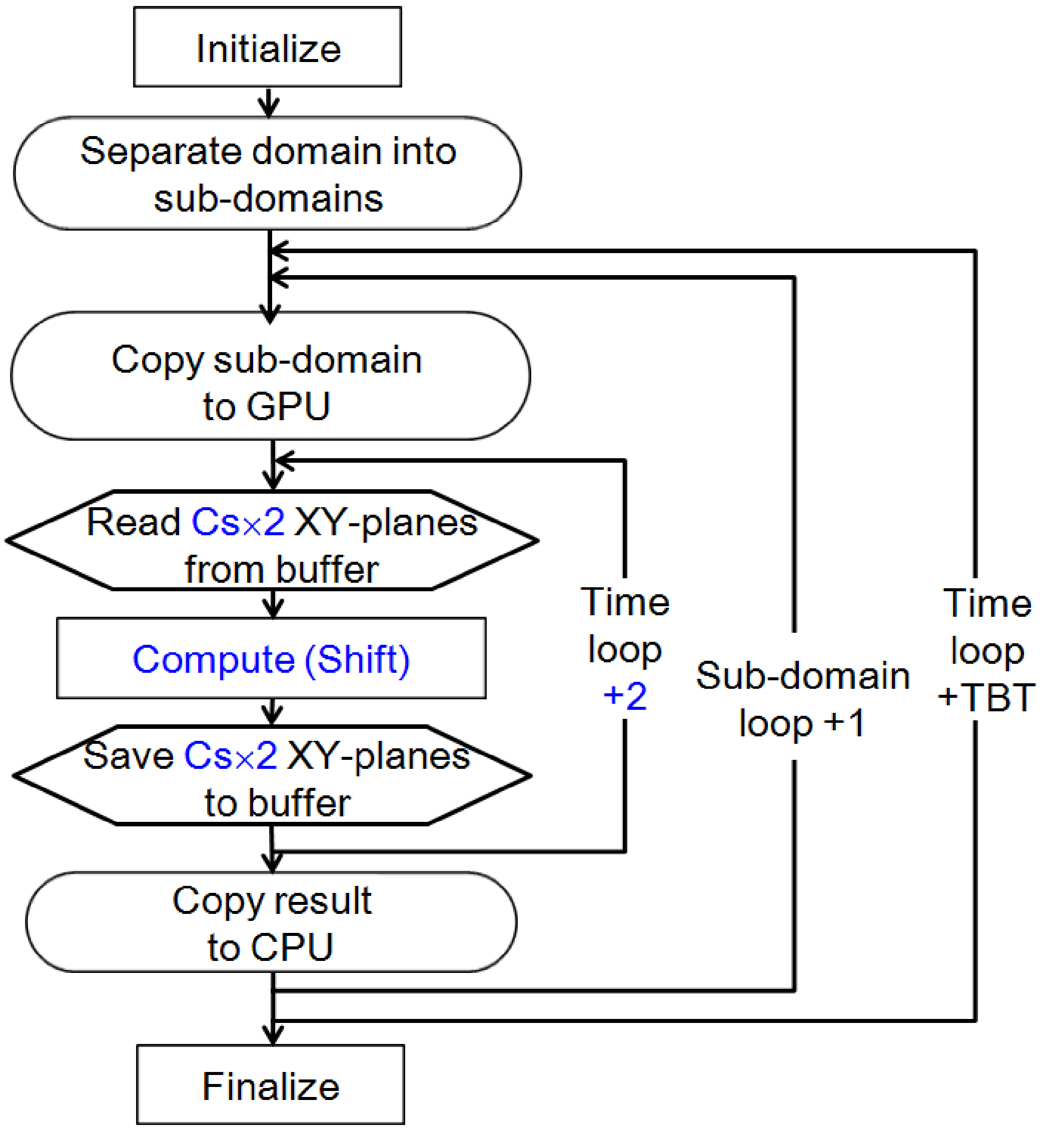


Fig.64. The process of 1D-2TBM method

We can get the pseudo code of 1D-2TBM method as below:

```
Allocate Grid on CPU to read initial of domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Allocate Buffer on GPU;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += TBT)
{
    For (j = 0; j < NSD; j += 1)
```

```
  {
      Copy sub-domain j from CPU to Grid0;
      For(k = 0; k<TBT; k+=2)
      {
          Read Cs×2 XY-planes from Buffer;
          Compute 2 time step by one kernel;
          Save result to shifted position on Grid1;
          Save Cs×2 XY-planes to Buffer;
          Swap Grid0 and Grid1;
      }
      Swap Grid0 and Grid1;
      Copy result from Grid1 to CPU;
}}
```

We evaluated the 1D-P-2TBM method for 7 point stencil on single node of TSUBAME2.0. The domain is from 240×240× 240 to 2160×2160× 2160. 1D-P-2TBM can compute bigger domain and has higher performance. It has 1.44 times better performance than 1D-2T.

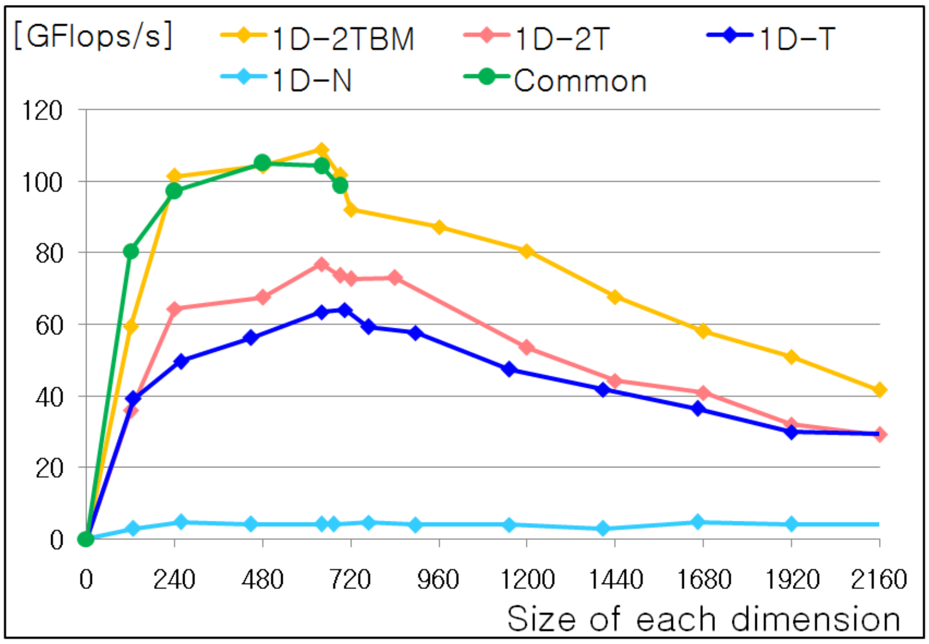


Fig.65. Evaluation on TSUBAME2.0

4.8 Contribution: 1D-P-2TBM method

As the domain grows, the sub-domain and buffer for buffer-copy method grows. As the GPU memory is limited, it contains smaller sub-domain or fewer XY-planes to reuse. Smaller sub-domain means more phases to compute and fewer XY-planes on buffer means fewer time steps that can be computed locally on GPU. Both of them degrade the performance. We parallel the communication with the computation to solve this problem.

The communication is paralleled during the computation. The computation uses two grids to execute. So, it should use other space to perform the communication. As it should send initial to the GPU and copy result from the GPU, the additional space is to send and receive those data. So, two buffers on the GPU are to perform the communication. We call those buffers as Buffer0 and Buffer1. The first buffer is to receive the initial from the CPU. The second buffer is to send the result to CPU. So by those two buffers, we can perform communication during the computation as they are independent.

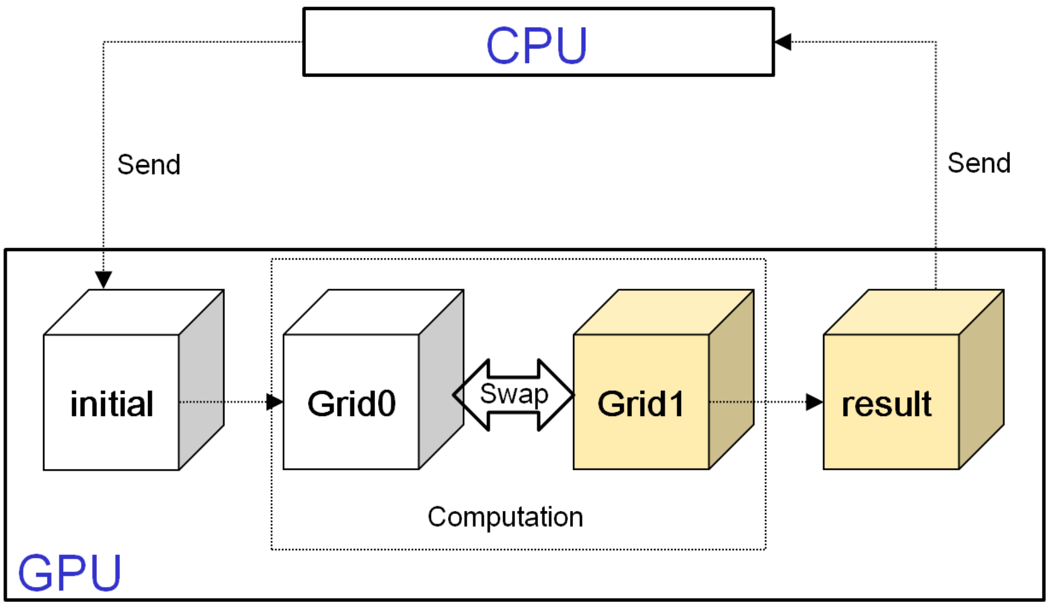


Fig.66. Parallel the communication with computation by additional buffers

We call this new method as 1D-P-TBM. 1D-TBM method needs two grids to perform double buffering. So the size of the space for initial, result and computation are same. As the below figure shows, it also separates the domain into sub-domains. Then, it copies sub-domain from CPU to GPU. Then, it can also use the temporal blocking with buffer-copy and memory-saving method.

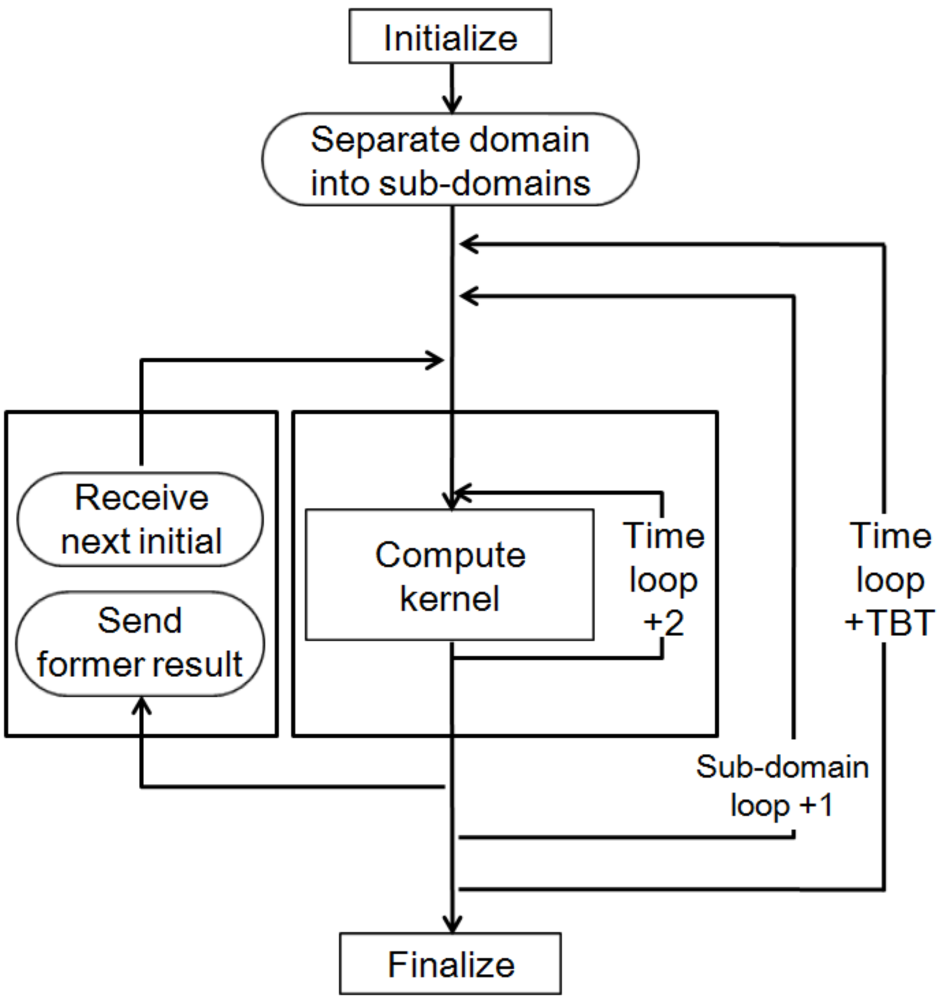


Fig.67. The process of 1D-P-2TBM

As there are two additional buffers to receive the initial and send the result, the communication can be paralleled with the computation. The current computation can be started only if there is the initial of current computation. The result is obtained after the computation of final time step. So sending the initial and result of current computation cannot be paralleled with the current computation. After the end of current computation, there comes the computation of next sub-domain. So, the initial for next sub-domain is required. So, we consider sending the initial for next sub-domain during the computation of current sub-domain. Also, the result of former sub-domain is prepared during the computation of current sub-domain. So, we consider sending the result of former sub-domain during the computation of current sub-domain. By adopting the temporal blocking method to the kernel, we can get 1D-2TBM method [37]. As it computes 2 time steps, it needs to read Cs×2 XY-planes from the buffer of GPU. Also it needs to save

Cs×2 XY-planes to the buffer of GPU. It reduces the loop time by 2. So the total time step of iteration should be divided by 2. Also the position that needs to be shift is Cs×2 XY-planes. We can get the pseudo code of 1D-P-2TBM method as below:

```
Allocate Grid on CPU to read initial of domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Allocate Buffer on GPU;
Allocate Buffer0 and Buffer1 on GPU;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += TBT) {
    For (j = 0; j < NSD; j += 1){
        Read initial from Buffer0;
        Parallel Do {
            Send next initial from CPU to Buffer0;
            Send former result from Buffer1 to CPU;
            For(k = 0; k<TBT; k+=2)
            {
                Read Cs×2 XY-planes from Buffer;
                Compute 2 time step by one kernel;
                Save result to shifted position on Grid1;
                Save Cs×2 XY-planes to Buffer;
                Swap Grid0 and Grid1;
            }
        }
        Swap Grid0 and Grid1;
        Save result to Buffer1;
    }
}
```

We evaluated the 1D-P-2TBM method for 7 point stencil on single node of TSUBAME2.0. The domain is from 240×240× 240 to 2160×2160× 2160. 1D-P-2TBM can compute bigger domain and has higher performance. 1D-P-2TBM has 1.35 times better performance than 1D-2TBM.

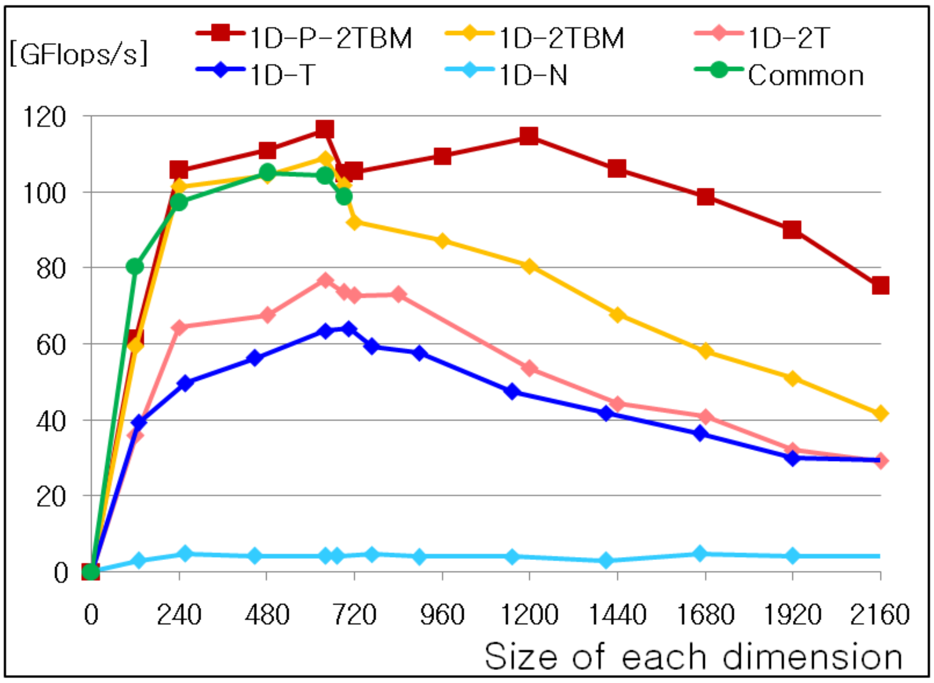


Fig.68.　Evaluation on TSUBAME2.0

4.9　Reevaluation on TSUBAME2.5

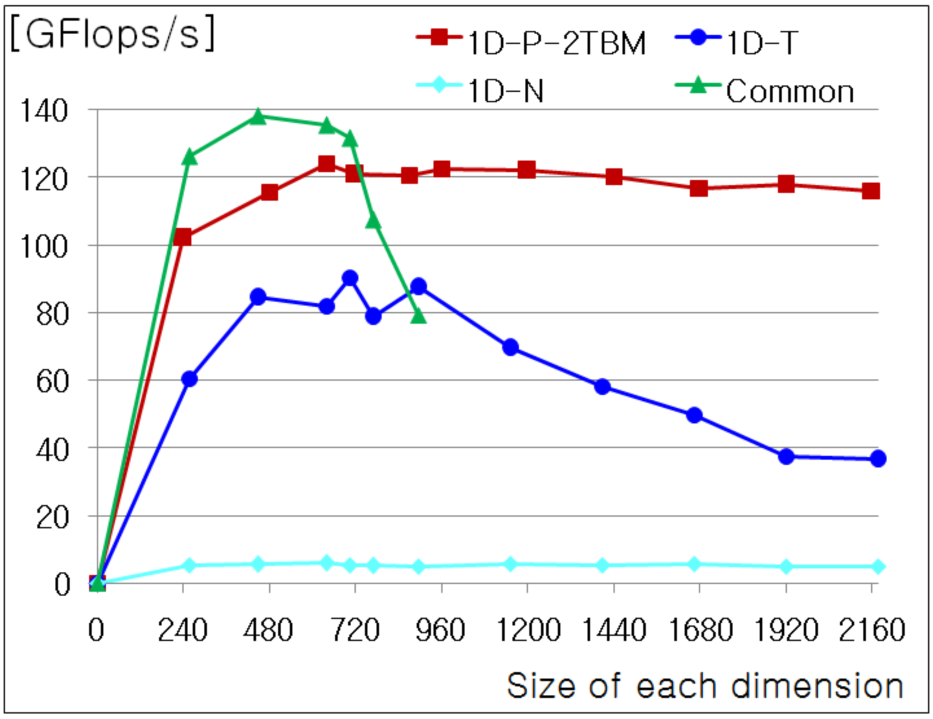


Fig.69.　Reevaluation on TSUBAME2.5

We evaluated the 1D-P-2TBM method for 7 point stencil on single node of TSUBAME2.5. The domain is from 240×240× 240 to 2160×2160× 2160. 1D-P-2TBM can compute bigger domain and has higher performance. 1D-P-2TBM is 2.26 times higher than 1D-T on TSUBAME2.0. 1D-P-2TBM is 1.94 times higher than 1D-T on TSUBAME2.5

The domain that common way can compute is 700×700×700 on TSUBAME2.0. The domain that common way can compute is 900×900×900 on TSUBAME2.5. The domain size that can be computed by common way is increased. But, it still cannot compute bigger domain as below figure shows.

We evaluated the 1D-P-2TBM method for 19 point stencil on single node of TSUBAME2.5. The domain is from 240×240× 240 to 2160×2160× 2160. 1D-P-2TBM can compute bigger domain and has higher performance. 1D-P-2TBM is 1.78 times higher than 1D-T on TSUBAME2.5

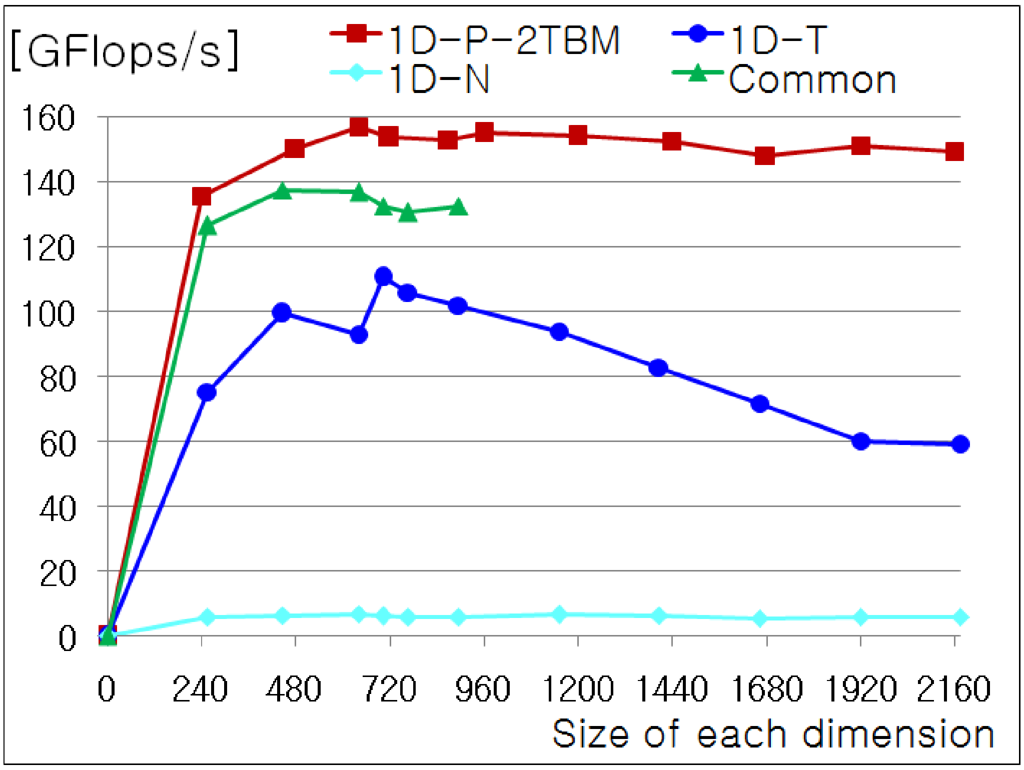


Fig.70. Reevaluation on TSUBAME2.5

Here we want to give how to decide the parameters for 1D-P-2TBM, Dx is size of X dimension, Dy is the size of Y dimension, Dz is the size of Z dimension, and our objective is to get max TBT. Here, NSD is the number of sub-domain; TBT is the temporal blocking times. GPU memory is shared by 2 grids (sub-domain), 2 buffers (communication), 1 buffer (buffer-copy) . So we can get the formula as below:

$$Dx \times Dy \times (Dz/NSD + Cs \times 2) \times 4 + Dx \times Dy \times TBT \times Cs \leq \text{GPU memory capacity} \quad (1)$$

$$TBT < Dz / NSD, \text{ get maxt } TBT \quad (2)$$

Dimension size		240	480	630	720	840	960	1200	1440	1680	1920	2160
1D-P-2TBM	NSD	2	2	2	2	3	4	8	15	24	32	48
	TBT	120	240	314	360	280	240	150	96	70	54	44

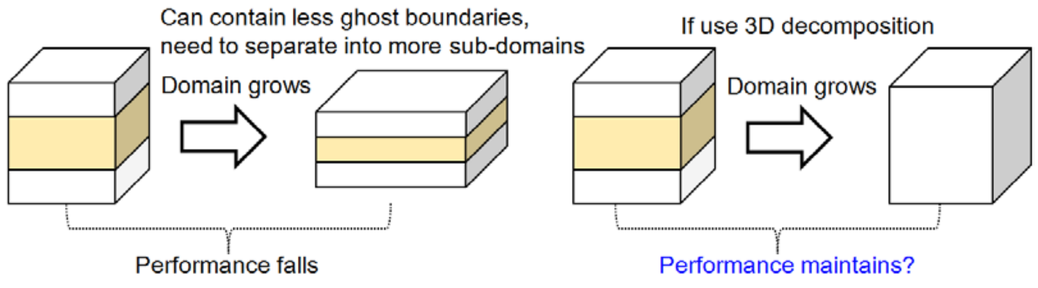


Fig.71. Parameters on TSUBAME2.0

As using 1D temporal blocking method causes lower performance in bigger domain area, we think 3D temporal blocking method can maintain high performance since the sub-domain size is the same.

4.10 Existing: compiler loop unrolling

Software pipelining is a type of out-of-order execution technique which is a loop optimization technique to make statements within iteration independent of each other. The goal is to remove dependencies so that seemingly sequential instructions may be executed in parallel. Software pipelining is often used in combination with loop unrolling. Here we give the technique of loop unrolling by CUDA compiler which symbol is #pragma unroll. By default, the compiler unrolls small loops with a known trip count. The #pragma unroll directive however can be used to control unrolling of any given loop. It must be placed immediately before the loop and only applies to that loop. It is optionally followed by a number that specifies how many times the loop must be unrolled. For example, in this code sample:

```
#pragma unroll 4

    for (int i = 0; i < n; ++i)
```

The loop will be unrolled 4 times. The compiler will also insert code to ensure correctness (in the example above, to ensure that there will only be n iterations if n is less than 4, for example). It is up to the programmer to make sure that the specified unroll number gives the best performance. #pragma unroll 1 will prevent the compiler from ever unrolling a loop. If no number is specified after #pragma unroll, the loop is completely unrolled if its trip count is constant, otherwise it is not unrolled at all.

We evaluate the unrolling (use #pragma unroll) by the CUDA compiler on different GPUs. The first GPU type is Kepler K20C on raccoon cluster of Tokyo Institute of Technology. The second GPU type is Kepler K20X on TSUBAME2.5. Here we give the architecture of the two GPUs.

	K20C	K20X
CUDA version	5.5	5.0
Peak single precision floating point performance	3.52 TFlops	3.95 TFlops
Shader processing units	2496	2688
Memory size	5GB	6GB
Bandwidth max	208 GB/s	250GB/s
L1 cache + shared	64KB/32 cores	64KB/32 cores
L2 cache	1310KB	1536KB

4.10.1 Evaluation: different stencil forms

We first evaluate the compiler loop unrolling on K20X GPU. We select 7 point stencil and 19 point stencil on domain 640×640×640 of 1D-TBM. NSD equals to 2. Temporal blocking times equal to 300. The threads in each block are (128, 8). Unrolling number 0 means it does not use the unrolling.

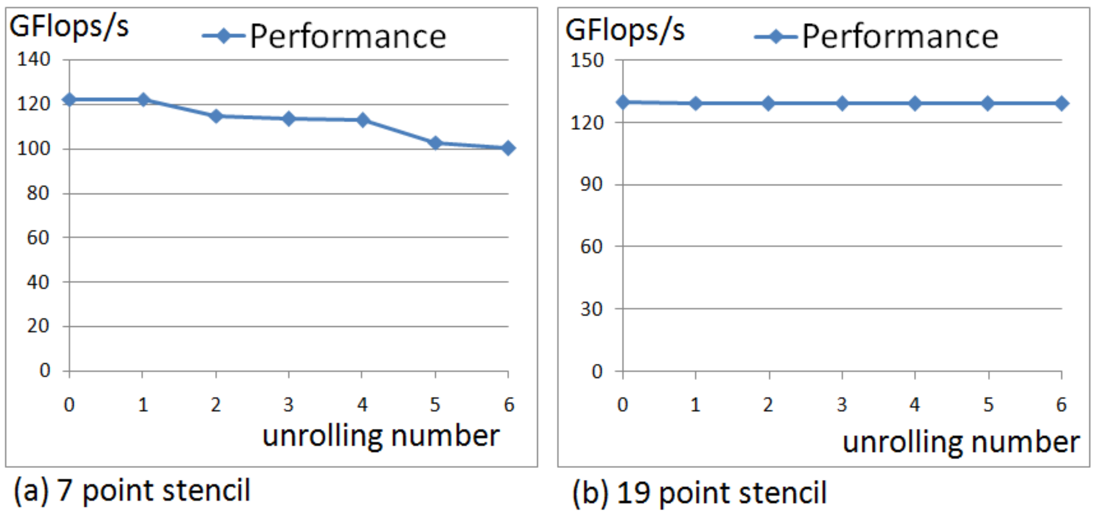
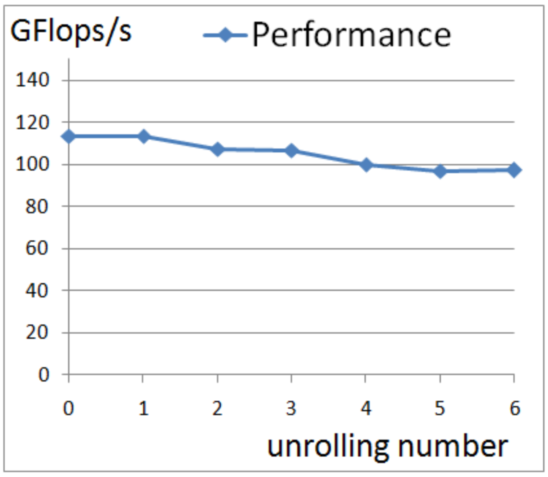


Fig.72. Performance of 7 point and 19 point stencil

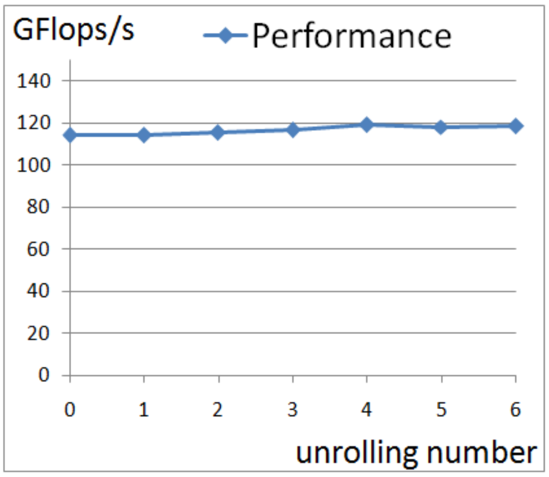
In 7 point stencil case, we found that the performance falls as it increases unrolling number. In 19 point stencil case, we found that the performance degrades less than 7 point case.

4.10.2 Evaluation: different GPUs

We evaluate the compiler loop unrolling on K20C. We select 7 point stencil and 19 point stencil on domain 640×640×640 of 1D-TBM. NSD equals to 2. Temporal blocking times equal to 300. The threads in each block are (128, 8). Unrolling number 0 means it does not use the unrolling.



(a) 7 point stencil



(b) 19 point stencil

In 7 point stencil case, we found that the performance also falls as it increases unrolling number. In 19 point stencil case, we found that the performance is improved a little.

4.10.3 Evaluation: different parameters for threads in each block

We evaluate the compiler loop unrolling on K20X. We select 7 point stencil on domain 768×768×768 of 1D-TBM. NSD equals to 2. Temporal blocking times equal to 240. Loop unrolling number is 0, 2 and 4. We set threads in each block are (256, 4), (128, 8), (64, 16), (32, 32) for spatial blocking method that computes kernel. As we can see in the below figure, we change the parameters of threads in each block. It shows that it cannot improve the performance by compiler loop unrolling in 7 point stencil case when it changes the parameters of threads in each block.

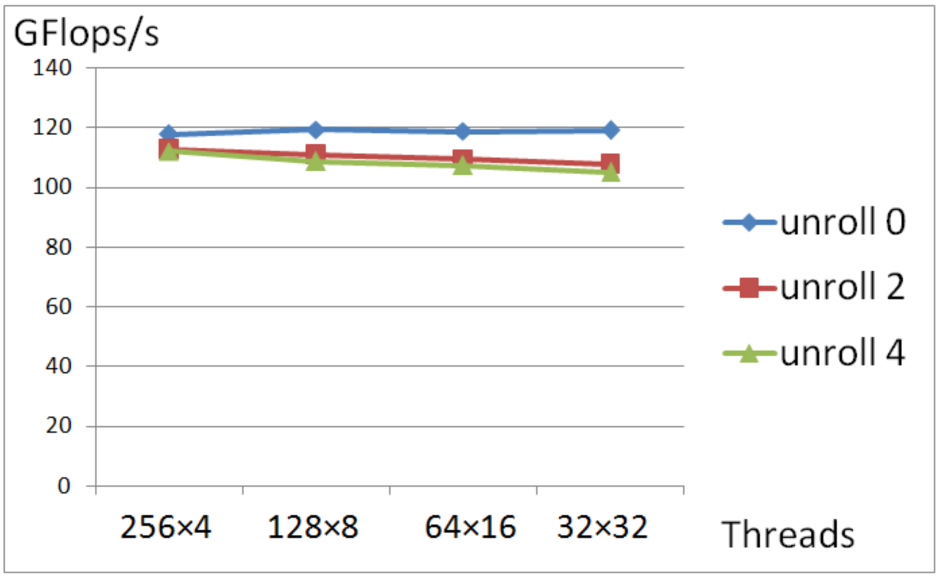


Fig.73. Change the parameters of threads in each block.

4.10.4 Evaluation: different domains

We evaluate the compiler loop unrolling by the compiler on K20X. We select 7 point stencil of 1D-TBM. Loop unrolling number is 0, 2 and 4. The domain is from 256×256×256 to 2176×2176×2176. As we can see in the below figure, the unroll 0 is better than the others in 7 point stencil case.

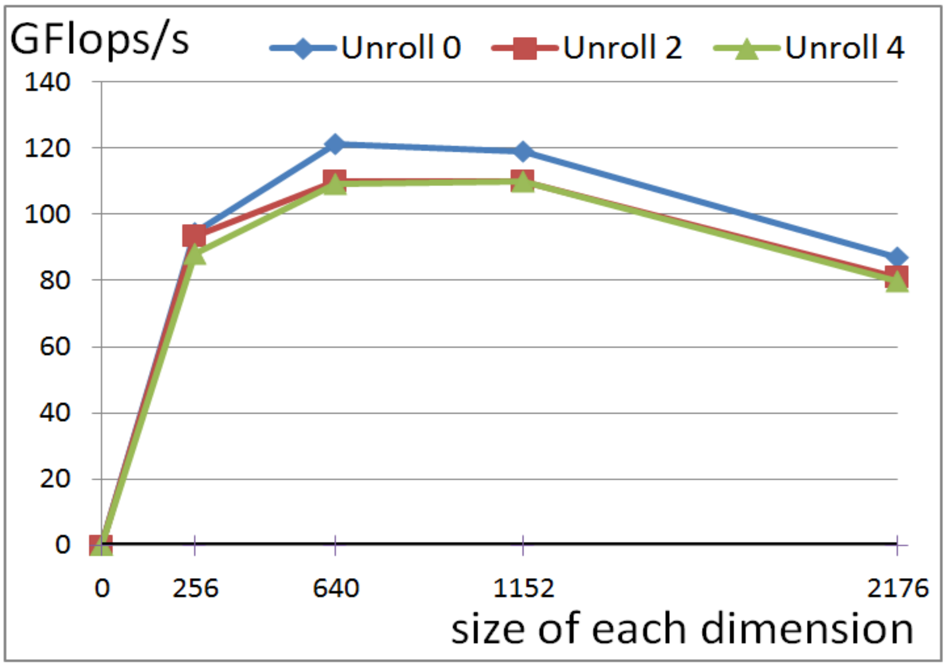


Fig.74. Loop unrolling on different domains.

4.10.5 Evaluation: cache hit rate

In this section, we use the NVIDIA visual profiler to analyze the loop unrolling by compiler. The NVIDIA visual profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. First introduced in 2008, Visual Profiler supports all 350 million+ CUDA capable NVIDIA GPUs shipped since 2006 on Linux, Mac OS X, and Windows. The NVIDIA Visual Profiler is available as part of the CUDA Toolkit.

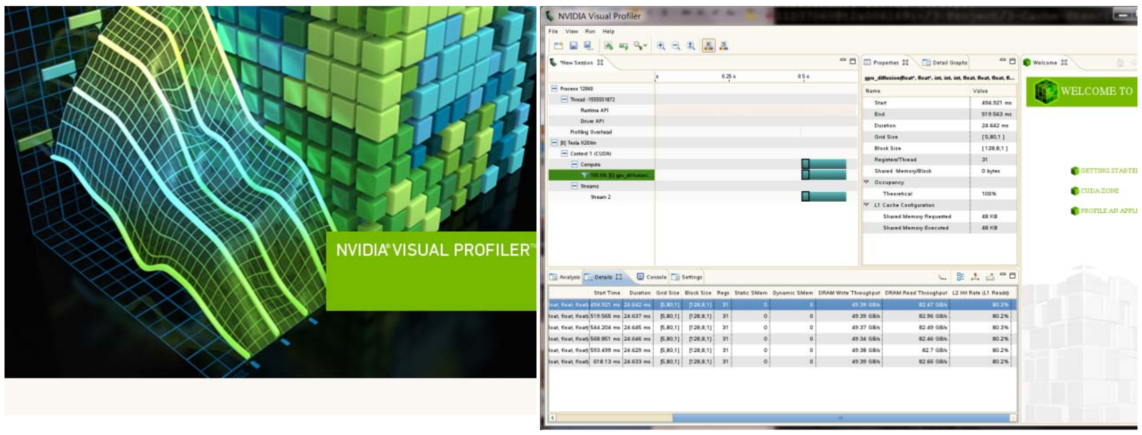


Fig.75. NVIDIA visual profiler

We evaluate the cache hit rate, DRAM reading and writing throughput (reading and writing the device memory by the kernel) to analyze the unrolling methods. We select the time step as 10, 20, 50 and 100 and get the average values. We evaluate the unrolling by compiler for kernel of 1D-TBM. We evaluate the 7 point and 19 point stencil on K20X GPU of TSUBAME2.5. The domain size is 640×640×640. The number of sub-domains is 2.

7 point stencil		Cache hit rate	Reading[GB/s]	Writing[GB/s]	Registers
Unrolling 1D-TBM by compiler	Unroll 1	80.3%	82.33	49.35	16
	Unroll 2	83%	61.9	46.1	24
	Unroll 4	84.3%	58.4	43.92	32

19 point stencil		Cache hit rate	Reading[GB/s]	Writing[GB/s]	Registers
Unrolling 1D-TBM by compiler	Unroll 1	94.1%	24.76	17.29	21
	Unroll 2	94.2%	23.66	17.26	29
	Unroll 4	94.3%	23.25	17.23	36

As we can see in the up table, the hit rate grows as the unroll number grows. But, it needs to allocate more registers to execute each thread. So, it affects the number of threads that can be executed concurrently which cause the DRAM reading and writing throughput falls. As the hit rate 19 point is high, it affects less than 7 point stencil. So, we need to consider manual unrolling method to improve the performance.

4.11 Contribution: manual loop unrolling, 1D-TBMR, 1D-P-TBMR

By using shared memory to compute 2 time steps in 1 kernel, it can improve the performance of kernel. But, it should mange the data on the shared memory which may cause difficulty of programming. As the upper evaluation shows, the performance cannot be improved obviously by the compiler loop unrolling. So, to easier programming while improving the performance of kernel, we try to use the manual loop unrolling method that uses registers to improve the performance of kernel as the related works has introduced.

4.11.1 Manual loop unrolling: use the registers to improve kernel

Here we give the manual unrolling technique. When we use the 2D spatial blocking method at the kernel on the GPU, The data should be prepared to compute each point on the thread. When we compute upper point, it also needs to prepare some overlapped data. So, there are overlapped reading between 2 points.

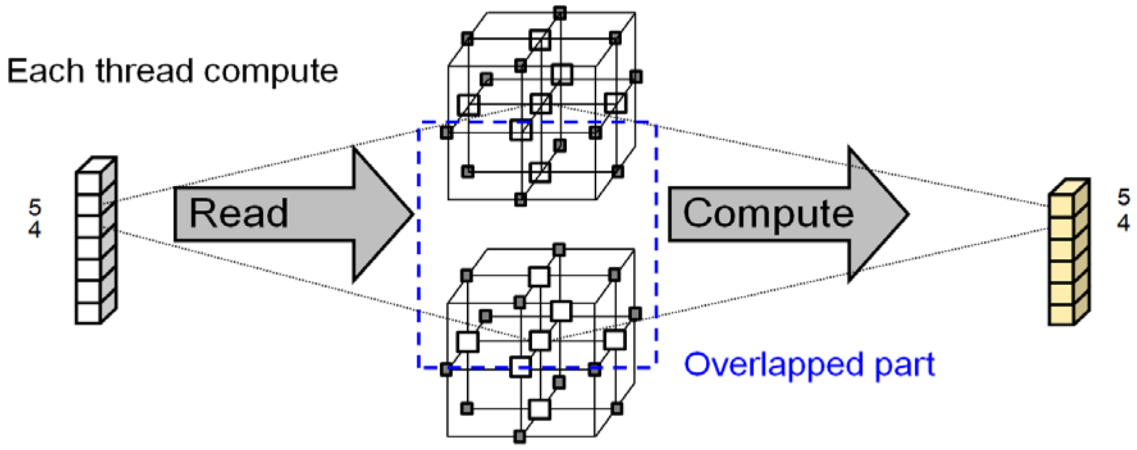


Fig.76. Overlapped reading between two points

To reduce the overlapped reading, it save some of them on the registers, and reuse them when computing upper point.

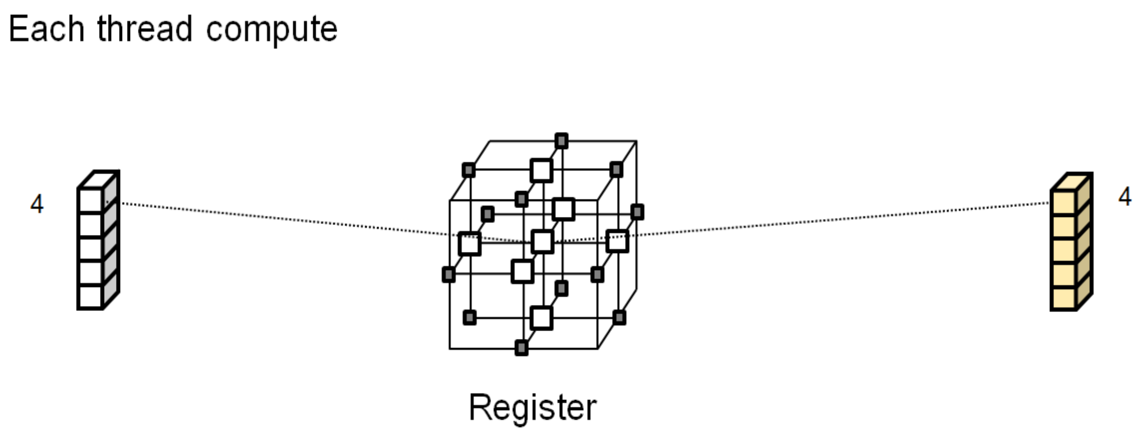


Fig.77. Read the nearby points for point 4 on the register.

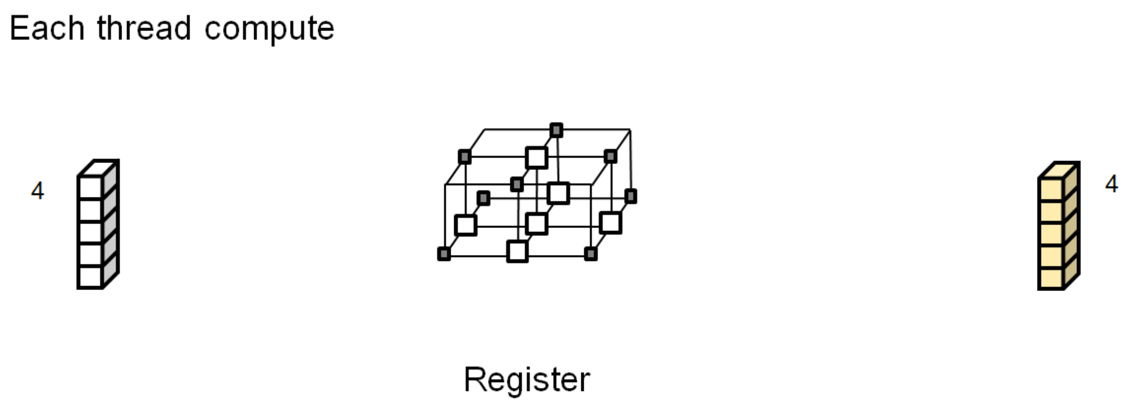


Fig.78. Save some part of initial of point 4 on registers.

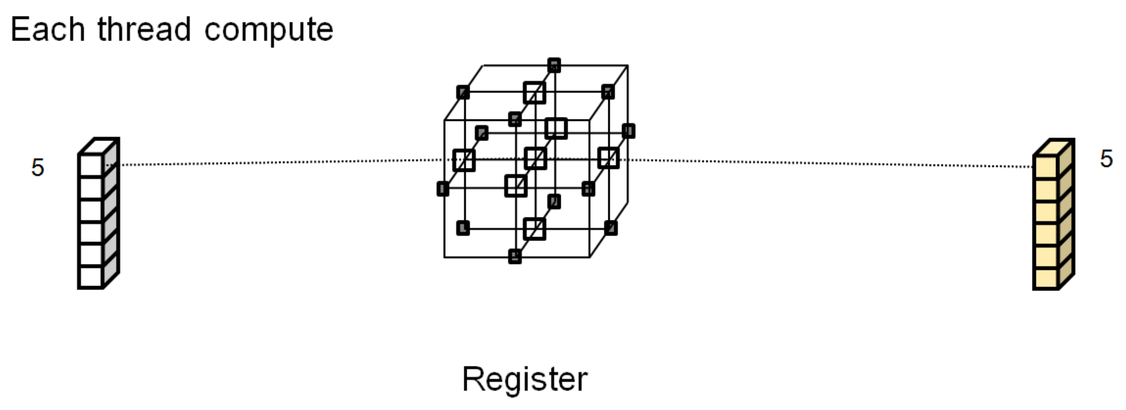


Fig.79. Reuse the saved part on registers for point 5.

Here we give the example code for using registers

```
__global__   void   kernel(grid0, grid1,…)
{
        int jx = blockDim.x * blockIdx.x + threadIdx.x;
        int jy = blockDim.y * blockIdx.y + threadIdx.y;
        for(jz = zstart;jz < zend; jz++)
        {
                ji = Dx*Dy*jz + Dx*jy + jx;
                je = ji+1; jw = ji-1;   jn = ji+Dx; js =ji-Dx;
                tempt       = grid0[ji];
                grid1[ji]    = tempo + cc*tempc + ct*tempt + cb*tempb;
                tempb       = tempc;
                tempc       = tempt;
                tempo       = ce*grid0[je] + cw*grid0[jw]
                             + cn*grid0[jn] + cs*grid0[js];
        }
}
```

As we can see in the up table, it reads 5 points instead of 7 points from the device memory. Also, it completes the computation by reading the data from register. It only needs to allocate 4 more points on the registers to unroll 2 loops. It also reduces the overlapped reading which is different from simple unrolling by compiler. By using registers instead of using shared memory to improve the performance of kernel, we can get 1D-TBMR (instead of 1D-2TBM), 1D-P-TBMR (instead of 1D-P-2TBM). The pseudo code is in Appendix A.1 and A.2.

4.11.2 Evaluation: 1D-P-TBMR vs other methods

We evaluated the optimization methods for 7 point and 19 point stencil on single node of TSUBAME2.5. The domain is from 240×240×240 to 2160×2160×2160 of floating computation. As we can see in the below figure, 1D-P-TBMR method gets 2.63 times higher performance than 1D-T method in 7 point stencil case. 1D-P-TBMR method gets 2.98 times higher performance than 1D-T method in 19 point stencil case. It shows that manual unrolling can efficiently improve the performance of kernel than the unrolling by compiler.

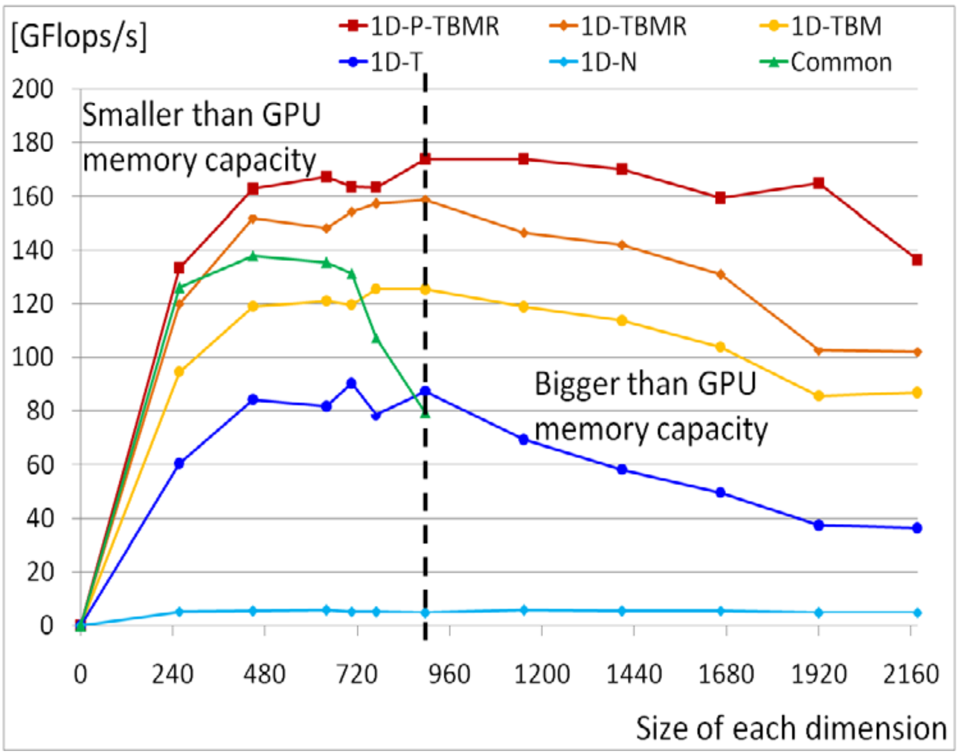


Fig.80. 7 point stencil, 1D-P-TBMR vs others

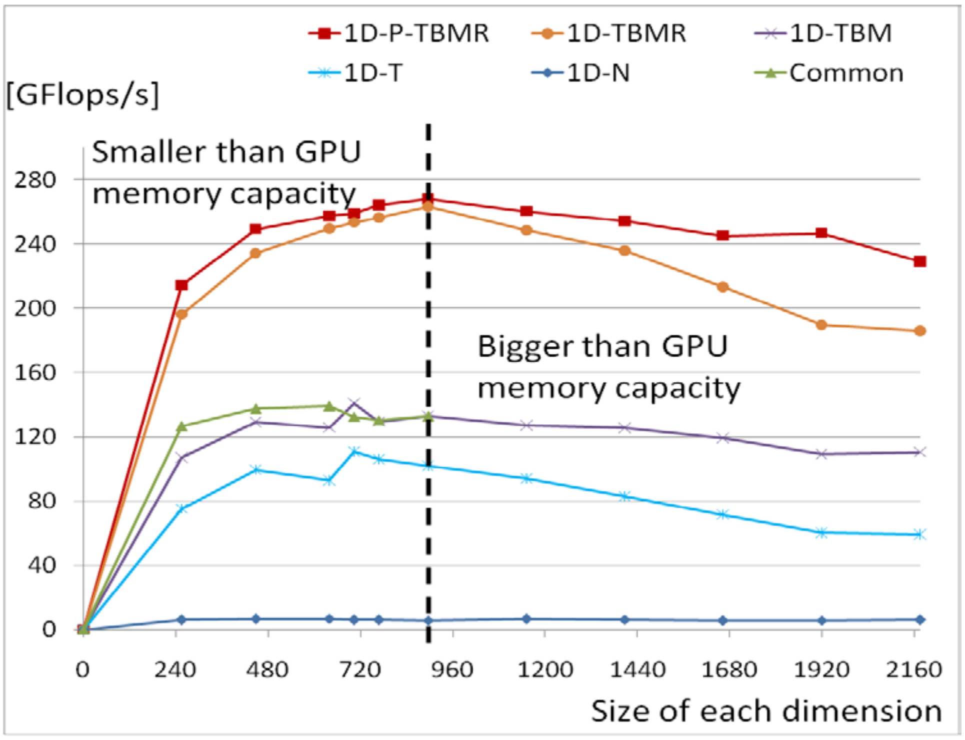


Fig.81. 19 point stencil, 1D-P-TBMR vs others

When we use the manual unrolling method, it can unroll the loop and read less data from the device memory. As it reads some data from registers to complete the computation, it can benefit writing the result to device memory. By those two reasons, it can improve the performance of kernel.

4.12 Discussion

4.13.1. Performance model

In this section, we first discuss the performance model of 1D-T and 1D-TBM.

We build the model of 1D-T as below:

$T_{TBT} = T_{C2G}(\text{domain})+T_{C2G}(\text{ghost boundaries})+T_{G2C}(\text{domain})$

$+ T_{computation}(\text{domain})+T_{computation}(\text{ghost boundaries})$

To get high performance, it should minimize $T_{TBT}\times$Iteration / TBT

We build the model of 1D-TBM as below:

$T_{TBT} = T_{C2G}(\text{domain})+T_{G2C}(\text{domain})+T_{computation}(\text{domain})$

To get high performance, it should minimize $T_{TBT}\times$Iteration / TBT.

We evaluate the actual time on the single node of TSUBAME2.5 to compare with the model. The domain is 640×640×640; the number of sub-domains is 2.

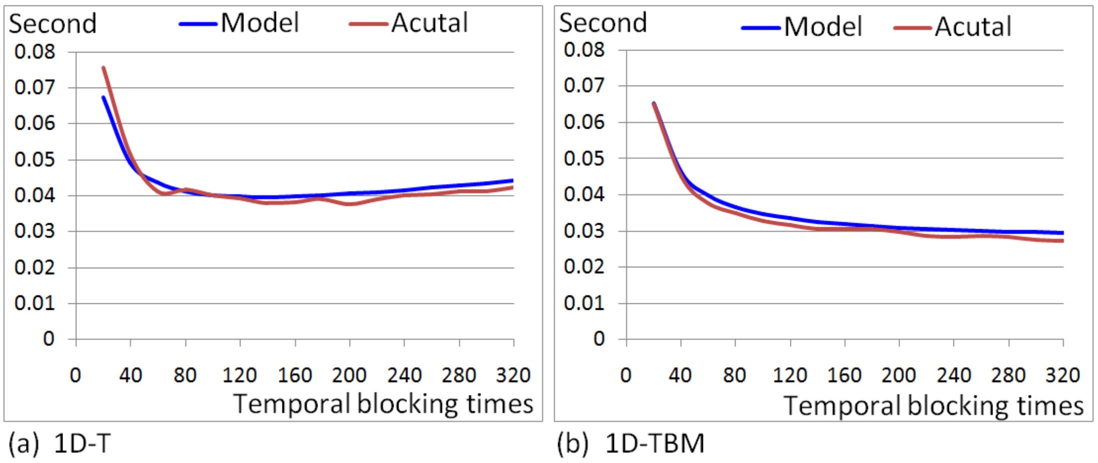


Fig.82. The model and actual of T_iteration/TBT on single node of TSUBAME2.5

As we can see in the upper figure, the models are similar to the actual evaluations which can be used to decide the parameters.

4.13.2. Other stencil forms

Then, we discuss about how to apply optimization methods to other stencil forms. As below figure shows, we define the area of computing 1 point is Px×Py×Pz. The nearby points are all included in this Px×Py×Pz area. For example, the area of computing one point in 7 point stencil case is 3×3×3.

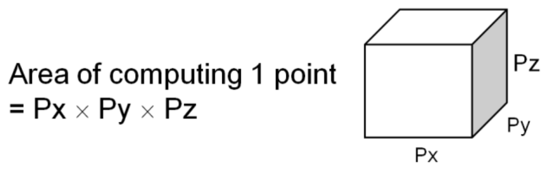


Fig.83. The area of computing 1 point.

So, we can get Cs = Pz – 1, space for 1D-P-2TBM:

Dx×Dy×(Dz/NSD+Cs×2)×4+Dx×Dy×TBT×Cs ≤GPU memory capacity (1)

TBT< min{Dz/NSD, Communication(1)/computation(1)/Ratio} (2)

Get max TBT, Ratio = (0,1)

If the TBT is big enough, it cannot reduce the communication significantly. So, we set TBT< min {Dz/NSD, Communication(1)/computation(1)/Ratio}. For example, Ratio can be the value from 0 to 1 which means it is to reduce the communication cost enough. Our objective is to get max TBT until min{Dz/NSD, Communication(1)/computation(1)/Ratio}.

4.13.3. Other domains: 2D domain

The second discussion is to apply optimization methods to computation on other dimension. As below figure shows, we define the area of computing 1 point is Px×Py in 2D dimension case. The nearby points are all included in this Px×Py area.

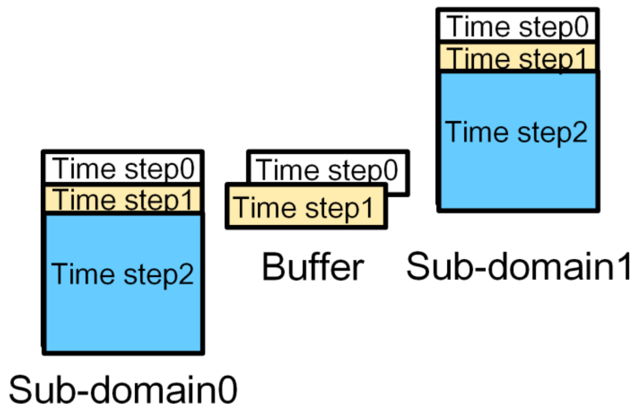


Fig.84. The optimization methods for 2D dimension case.

So we can get Cs = Py - 1 , data reuse Dx×Cs, space for 1D-P-2TBM:

Dx×(Dy/NSD+Cs×2)×4+Dx×TBT×Cs ≤GPU memory capacity (1)

TBT< min{Dy/NSD, Communication(1)/computation(1)/ Ratio} (2)

Get max TBT, Ratio = (0,1)

4.13.4. Multi-D temporal blocking: 2D temporal blocking

The third discussion is to apply multi-dimension temporal blocking method. As the below figure shows, we give the example for 2D temporal blocking method for 2D domain. It separates the domain into sub-domains by 2D decomposition. Then, when it goes from left sub-domain to right sub-domain, it reuses some data by the buffer on the GPU as the upper figure shows. Also, when it goes from bottom sub-domain to upper sub-domain, it also reuses some data by the buffer on the GPU. So, it can apply 2D temporal blocking method to 2D domain.

When it copies sub-domain to the GPU side, it should rearrange the data to continuous space. Also, when it reuses the data by the buffer, it should manage the data form. Both of them increase the difficulty of implementation.

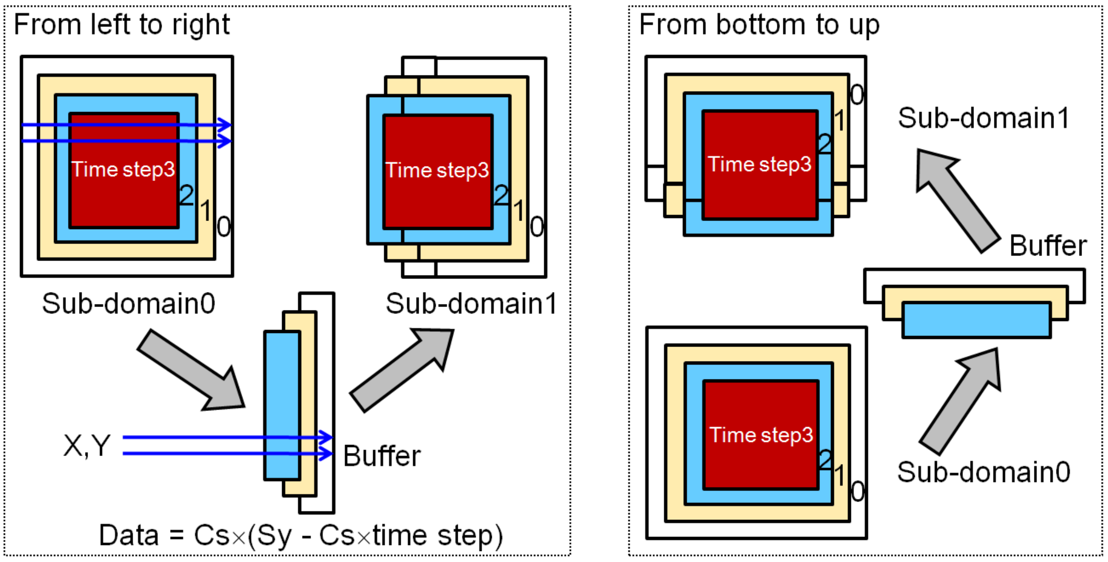


Fig.85. 2D temporal blocking method for 2D domain.

4.13.5. Limitation of multi-D temporal blocking

Here we discuss 2D temporal blocking method for 2D domain.

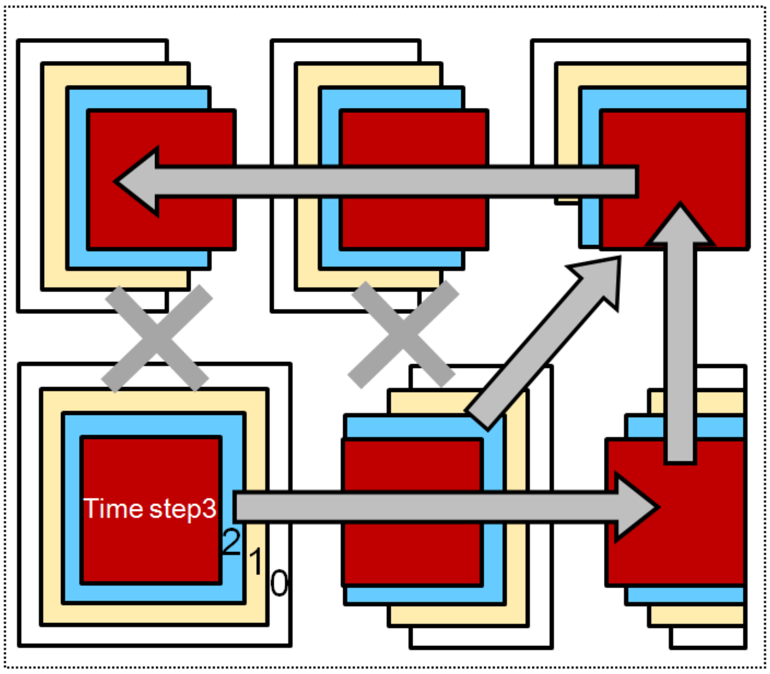


Fig.86. The limitation of 2D temporal blocking method for 2D domain.

The domain is separated into sub-domains by the 2D decomposition method. Then, it uses 2D temporal blocking method between sub-domains. Between the sequential sub-domains, it can use TBM method (temporal blocking+buffer-copy+memory-saving) to reduce the redundant cost. But, between the sub-domains that are not computed sequentially, it cannot reduce the redundant cost since it does not save overlapped part between those sub-domains as the upper figure shows. So, we can build performance model for 2D-T and 2D-TBM as below:

The model of 2D-T:

$$T_{TBT} = T_{C2G}(\text{domain})+T_{C2G}(\text{ghost boundaries})+T_{G2C}(\text{domain}) + T_{computation}(\text{domain})+T_{computation}(\text{ghost boundaries})$$

The model of 2D-TBM as below:

$$T_{TBT} = T_{C2G}(\text{domain})+T_{C2G}(\text{ghost boundaries/2})+T_{G2C}(\text{domain}) + T_{computation}(\text{domain})+T_{computation}(\text{ghost boundaries/2})$$

To get high performance, it should minimize T_{TBT}×Iteration / TBT. 2D-T cannot reduce the redundant cost between any sub-domains. 2D-TBM can reduce the redundant cost between 2 sequential sub-domains. So, it can reduce 1/2 of the redundant cost. Below shows the estimated time on TSUBAME2.5 by those models. As the below figure shows, 2D-TBM can reduce more redundancy cost than 2D-T method.

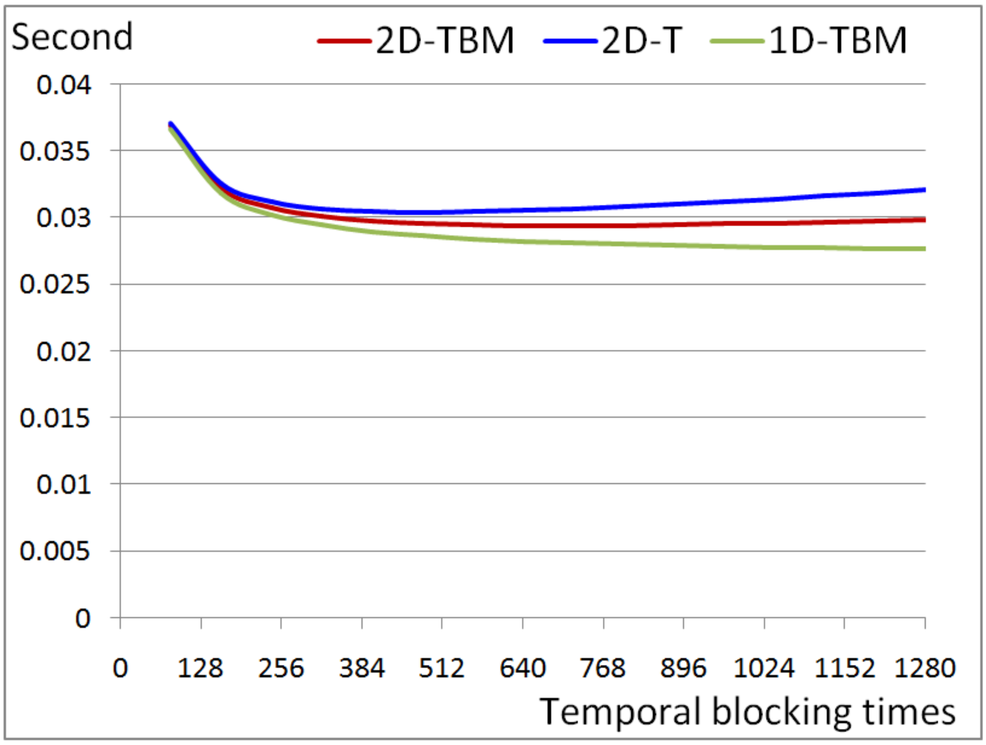


Fig.87. The estimated time of 2D-TBM, 2D-T, 1D-TBM, domain is 16384×16384.

So, we can decide whether use 2D-TBM or 1D-TBM. In 1D-TBM case, the time is reduced as the temporal blocking times grow. In 2D-TBM case, the time rebounds as the temporal blocking times grow. So we can use the 2D-TBM instead of 1D-TBM only if Min_time(1D-TBM)/Min_time(2D-TBM)>Ratio, Ratio>1.0. NSD_x means the number of sub-domains in X dimension, NSD_y means the number of sub-domains in Y dimension,

In 1D-TBM case, the max TBT for Min_time(1D-TBM) is decided by below formula.

Dx×(Dy/NSD+Cs)×2+Dx×TBT×Cs≤GPU memory capacity (1)

TBT<min{Dy/NSD,Communication(1)/computation(1)/Ratio0} ,Ratio0=(0,1) (2)

In 2D-TBM case, the max TBT for Min_time(2D-TBM) is decided by below formula.

$(Dx/NSD_x+Cs)\times(Dy/NSD_y+Cs)\times2+Max\{Dx,Dy\}\times TBT\times Cs\leq$ GPU memory capacity (1)

TBT that get minimum $T_{TBT}\times$Iteration / TBT (2)

The below figure is an example. The domain is 16384×16384. In small domain case, it is better to use 1D-TBM instead of 2D-TBM.

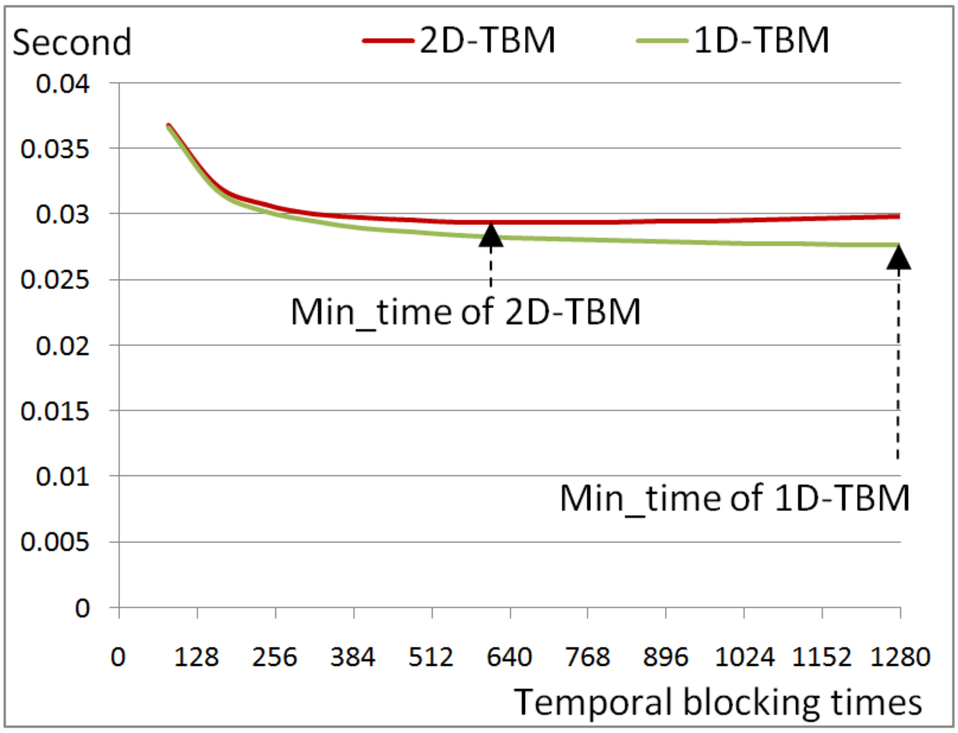


Fig.88. An example of small domain

The below figure is an example. The domain is 491520×491520. In big domain case, it is better to use 2D-TBM instead of 1D-TBM.

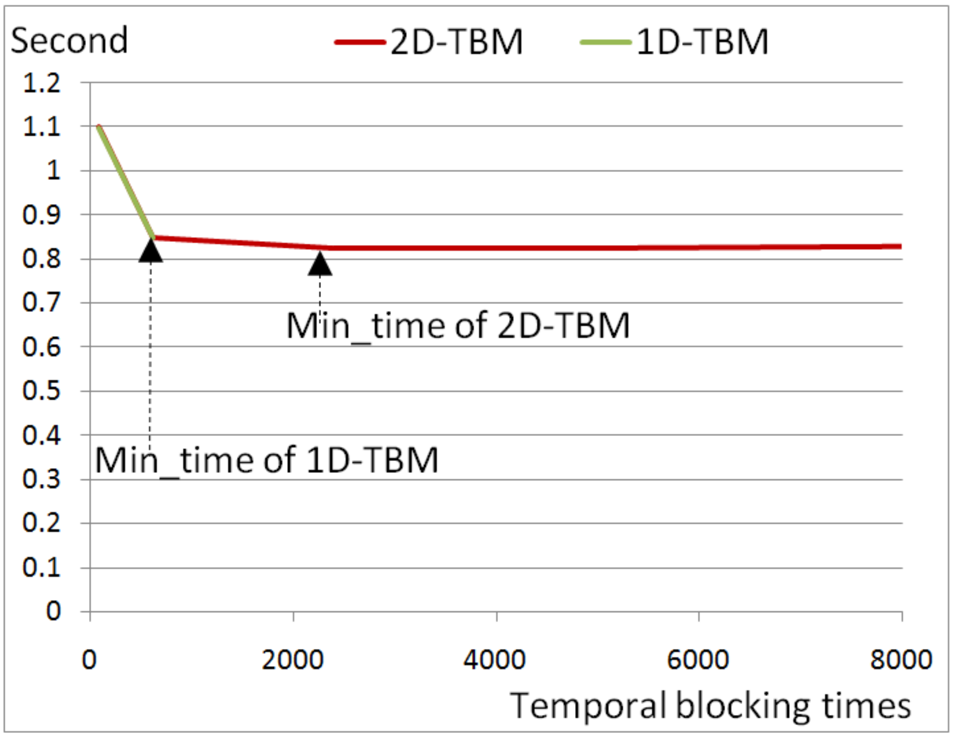


Fig.89. An example of big domain

4.13 Summary

In this chapter, we proposed 1D-2T, 1D-TBM, 1D-2TBM, 1D-P-2TBM, 1D-TBMR, 1D-P-TBMR methods. Those methods enable the computation on the domain while maintaining high performance on single node of TSUBAME system.

The naive method can enable the computation on the domain that is bigger than the memory capacity of GPUs. But, it causes frequent communication between CPU and GPU. Temporal blocking method can reduce the communication cost between CPU and GPU. But, it causes redundant communication and computation cost more by the more ghost boundaries. We propose buffer-copy method which saves some result to the buffer on GPU and reuse the result to solve redundancy problem. We also propose buffer-copy method to save space. We use shared memory to improve the performance of kernel. We also evaluate the unrolling method by compiler and manual to improve the performance of kernels. We use NVIDIA visual profiler to analyze the unrolling methods. The result shows it is better to use manual unrolling to our optimization methods. Then we overlap the communication with computation to improve the performance of the optimization methods. By the combination of those methods, we can get different optimization methods which get higher performance than other optimization methods.

In discussion part, we figure out performance model and optimization methods for other stencil forms and other dimensional domains. Those works can be used to decide parameters and optimization methods in other stencil or domain case. We published 2 papers depend on our new optimization methods.

[36] 金 光浩，遠藤 敏夫，松岡 聡． GPU メモリ容量を超える問題規模に対応する高性能ステンシル計算法 ．ハイパフォーマンスコンピューティングとアーキテクチャの評価に 関する北海道ワークショップ (HOKKE-20), 情報処理学会研究報告, 2012-ARC-194/HPC-137, 6 pages, 2012.12. (Domestic, non refereed, published)

[37] Guanghao Jin, Toshio Endo, Satoshi Matsuoka. A Multi-level Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPU. In Proceedings of The Third International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), in conjunction with IEEE IPDPS 2013, Boston, May 2013. (International, refereed, published)

Chapter 5. Optimization methods for stencil computation on multiple nodes

5.1 Introduction

To apply optimization methods on multiple nodes, we first propose decomposition methods among nodes. We only use 1 GPU in each node. First, we implement existing optimization methods which called 2D-1D-N, 2D-1D-T. Then, we propose our optimization method which called 2D-1D-TBM and 2D-1D-TBMR. We evaluate 7, 19 point stencil on 3D domain of floating computation on TSUBAME2.0 and TSUBAME2.5.

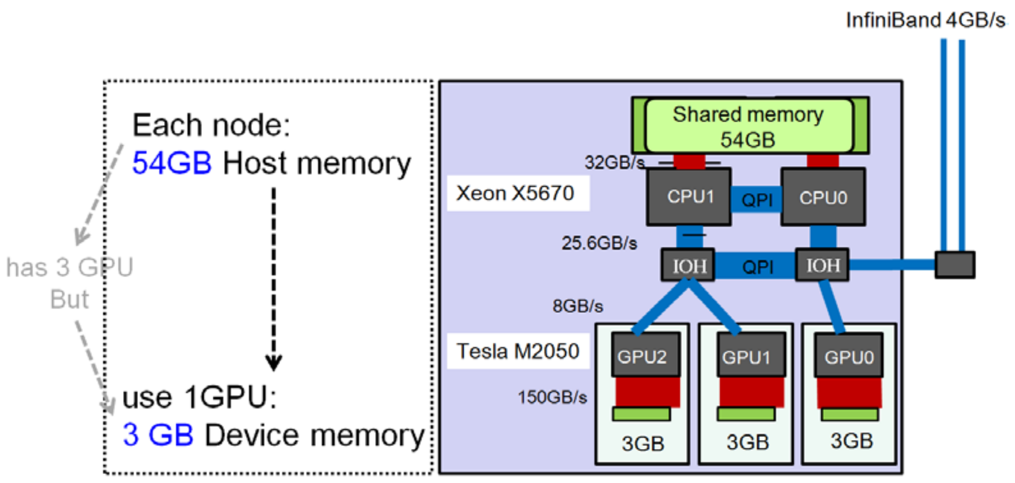


Fig.90. TSUBAME2.0

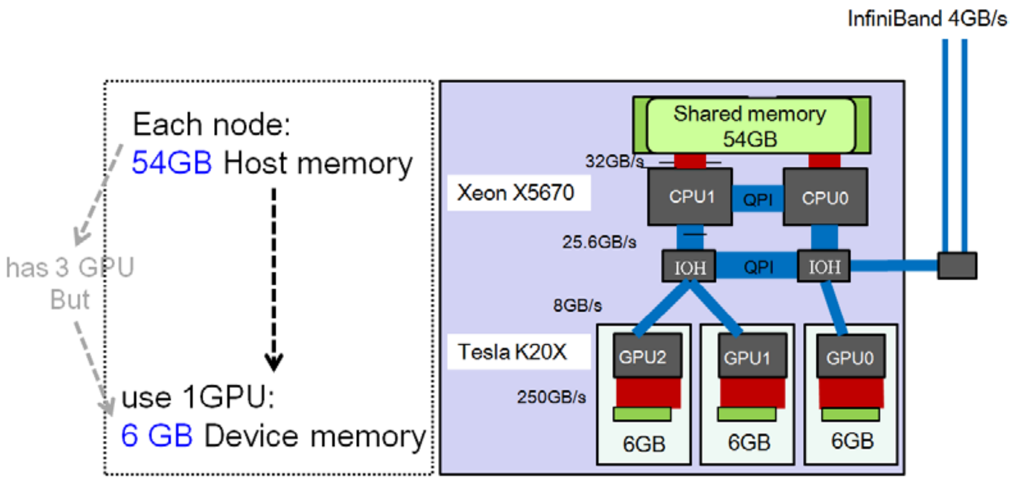


Fig.91. TSUBAME2.5

5.2 Contribution: decomposition

To enable the computation on the bigger domain by multiple nodes, we first need to consider how to separate the whole domain among the nodes. Our upper optimization methods achieve scalability in Z dimension as it separate the domain by one dimension. In 3D domain case, to achieve scalability in three dimensions, it is better to separate the domain in X and Y dimensions among GPUs. So, we separate the domain in the other two dimensions among nodes. Our objective is to enable the computation on the bigger domain. If we can enable the computation on each sub-domain that is bigger than the memory capacity of each GPU, it means that the whole domain is bigger than the memory capacity of GPUs. So, we should enable the computation on the sub-domain which is bigger than the memory capacity of each GPU. Then we use each GPU to compute this bigger sub-domain. As each sub-domain is bigger than the memory capacity GPU, we also need to separate the sub-domain into parts to fit the memory capacity of GPU.

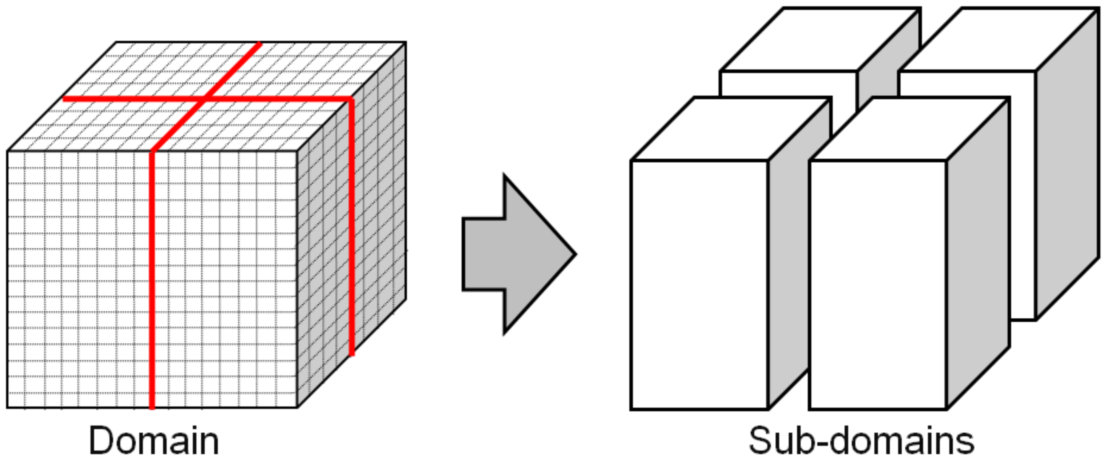


Fig.92. Separate the domain by two dimensions for 3D domain

To use the optimization method like 1D-N, 1D-T, 1D-TBM, it is better to separate the sub-domain by one dimension. As you can see in the below figure, we also separate each sub-domain to parts. We send each part which is smaller than the memory capacity of GPU to the GPU to compute. The size of each part depends on the optimization method. For example, if we only use naive method, it only needs space to contain each part. If we use temporal blocking method, it also needs to allocate the space for multiple ghost boundaries. If we use temporal blocking with buffer-copy method, it needs to allocate additional buffer for buffer-copy method.

The common 2D decomposition method is to separate the domain in Y and Z dimensions. As it saves the domain points in X, Y, Z order, 2D decomposition in Y and Z dimension can benefit the implementation and boundary exchange in common way case. When using 1D optimization method between sub-domain and parts, the computational part on the GPU size is changed at each time step. To contain the points in continuous space on GPU side, it is better to separate the domain in X, Y dimension among nodes.

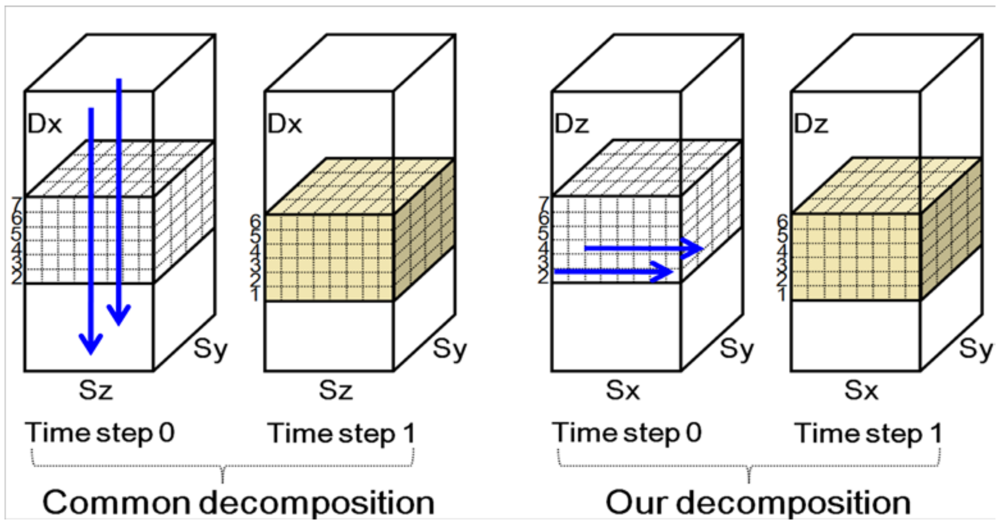
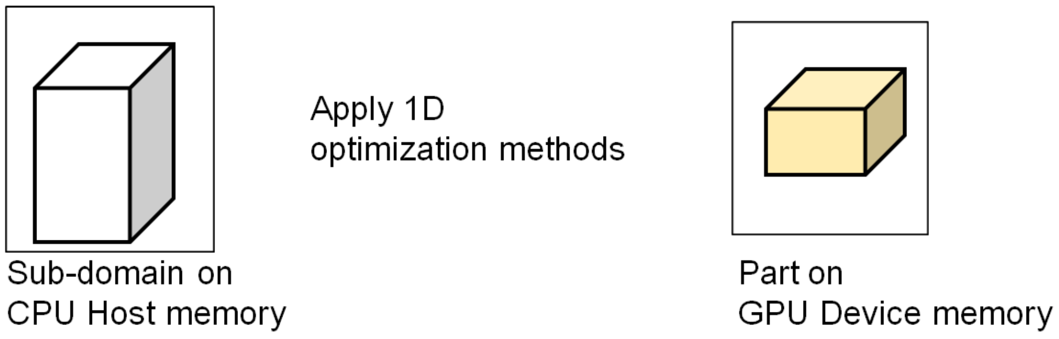


Fig.93. Common decomposition VS our decomposition

5.3 Apply 1D optimization methods

We need to consider how to apply the optimization methods when computing the big sub-domain by each GPU. When compute each sub-domain, we separate it into parts and send each part to GPU to compute. So, it is similar to the 1D optimization methods that have been introduced.



Apply 1D optimization methods
Sub-domain on CPU Host memory
Part on GPU Device memory

Fig.94. Separate each sub-domain into parts

5.4 Exchange the boundary

When each part is computed by each GPU, it needs to exchange the boundary information during the computation. As the computation of each point needs information of the nearest neighbors, the boundary of each part also needs the updated information of its neighbors. The neighbors are updated by the adjacent part on the other nodes. It should exchange boundary information during the iteration. So the boundary exchange is executed as below figure. As we can see in the figure, each part should exchange the boundary information to continue the computation. Each GPU computes the same part of each sub-domain. So after the computation of each time step, it should update the information of the boundary. By exchanging the boundary information, each part gets the information to continue the iteration. To deliver the boundary, it compresses the boundary to a vector. As it is divided in X and Y dimension, it is more difficult to compress the boundaries. Then, it can send the boundary to the CPU side. It delivers the boundary to the remote CPU. Then it sends the boundary from the remote CPU to remote GPU.

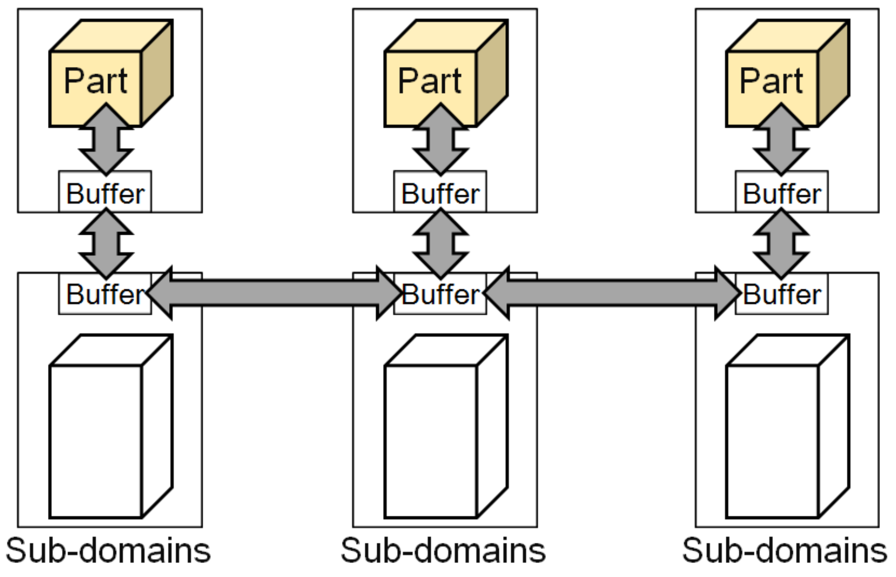


Fig.95. Exchange the boundary information through the CPUs.

5.5 Overlap the communication with computation

When it performs communication, it also computes inside part to reduce the communication cost.

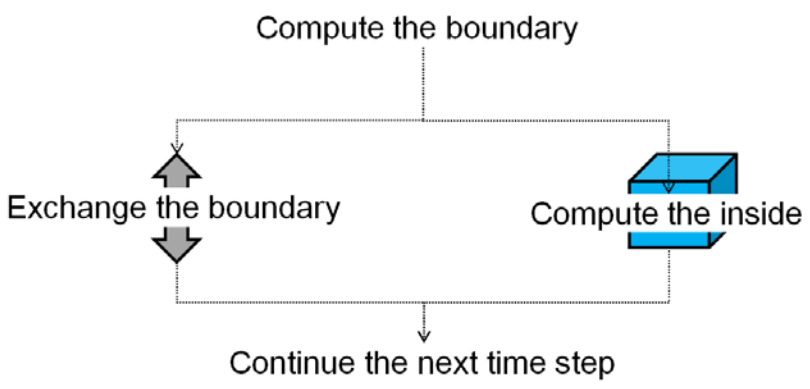


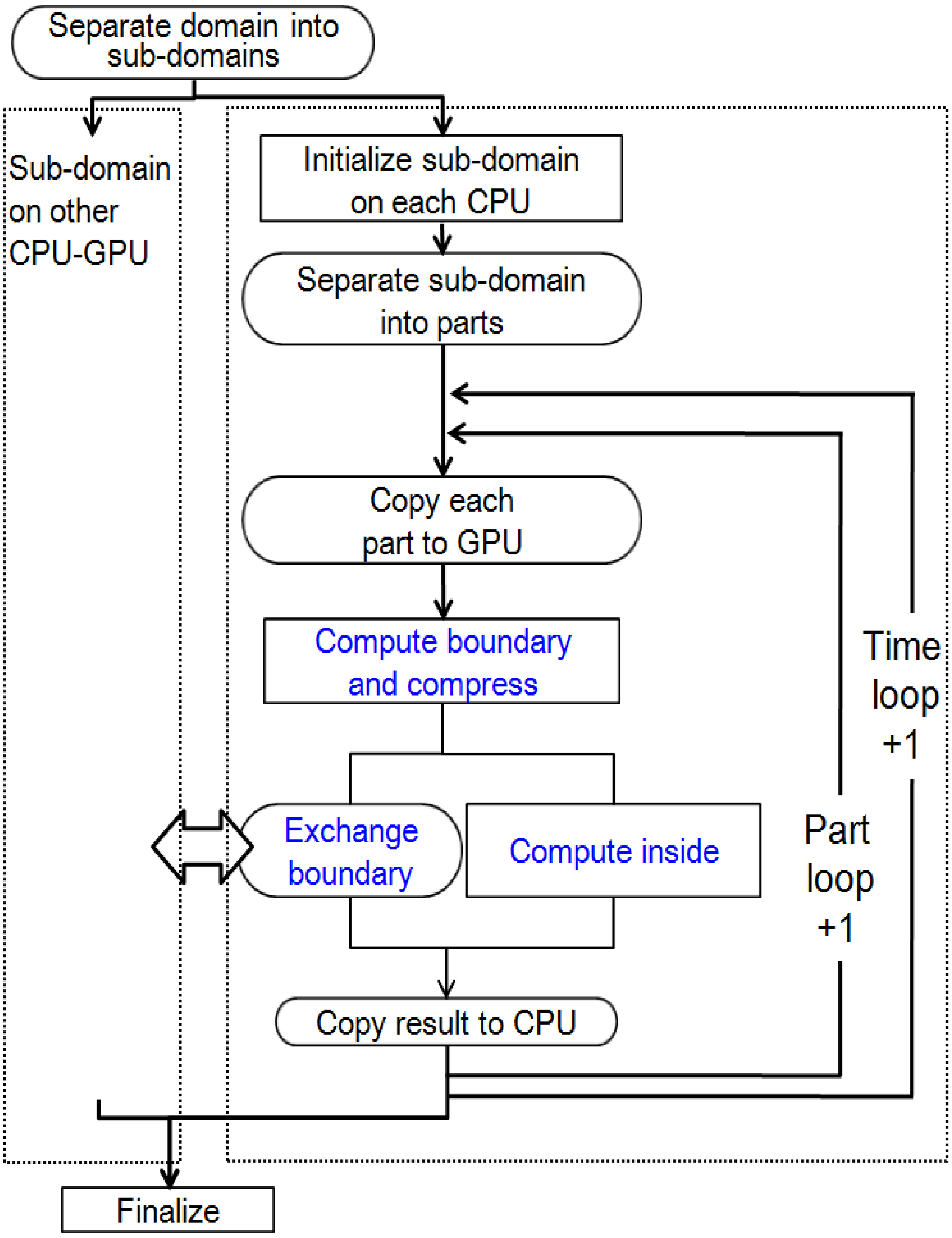
Fig.96. Overlap the communication with inside computation.

5.6 Existing: 2D-1D-N method

We can figure out the pseudo code as below:

```
Separate the domain into sub-domains;
Allocate Grid on each CPU to read initial of sub-domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Separate domain into parts;
For (i = 0; i< TSI; i += 1)
{
    For (j = 0; j < NPS; j += 1)
    {
        Send initial from CPU to Grid0;
        Compute boundary;
        Compress boundary;
        Parallel Do
        {
            Compute inside;
            Exchange boundary;
        }
        Read result from Grid1 to CPU;
    }
}
```

On each node, it copies each part to GPU side and only computes 1 time step and copy the result back. So, it also cannot get high performance because of frequent communication between host and device memory. We first implement the 2D-1D-N method as below:



Separate domain into sub-domains
Sub-domain on other CPU-GPU
Initialize sub-domain on each CPU
Separate sub-domain into parts
Copy each part to GPU
Compute boundary and compress
Exchange boundary
Compute inside
Copy result to CPU
Time loop +1
Part loop +1
Finalize

Fig.97. The process of 2D-1D-N method

As we can see in the upper figure, it separates the domain into sub-domains. Then, it initializes the sub-domain on the CPU side. Then it separates the sub-domain into parts. It sends each part to the GPU and only computes 1 time step. Then it sends the result back to CPU. So it degrades the performance because of the frequent communication.

5.7 Existing: 2D-1D-T method

The process of the 2D-1D-T method is below:

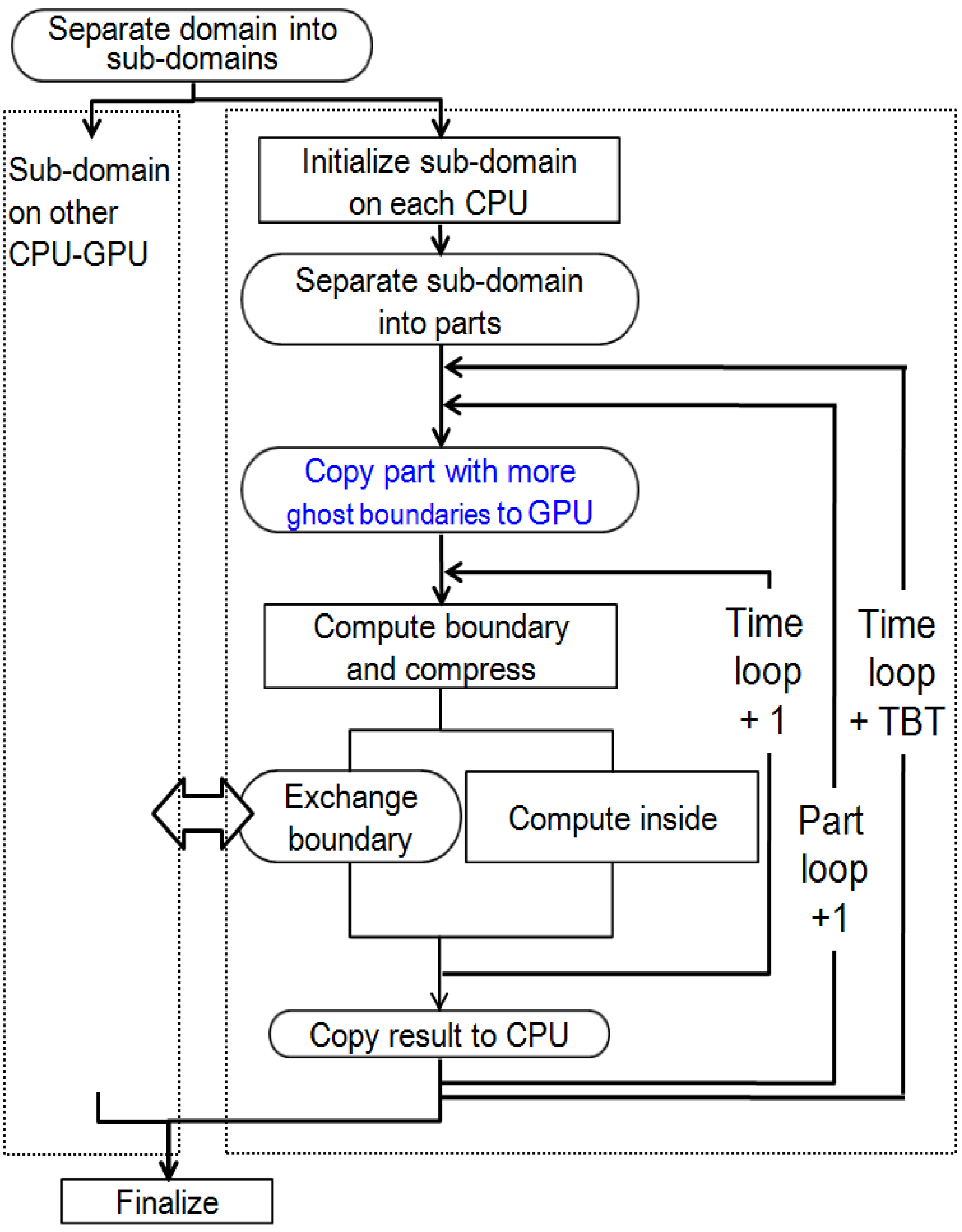

Separate domain into sub-domains
Sub-domain on other CPU-GPU
Initialize sub-domain on each CPU
Separate sub-domain into parts
Copy part with more ghost boundaries to GPU
Compute boundary and compress
Exchange boundary
Compute inside
Time loop + 1
Time loop + TBT
Part loop +1
Copy result to CPU
Finalize

Fig.98. The process of 2D-1D-T method

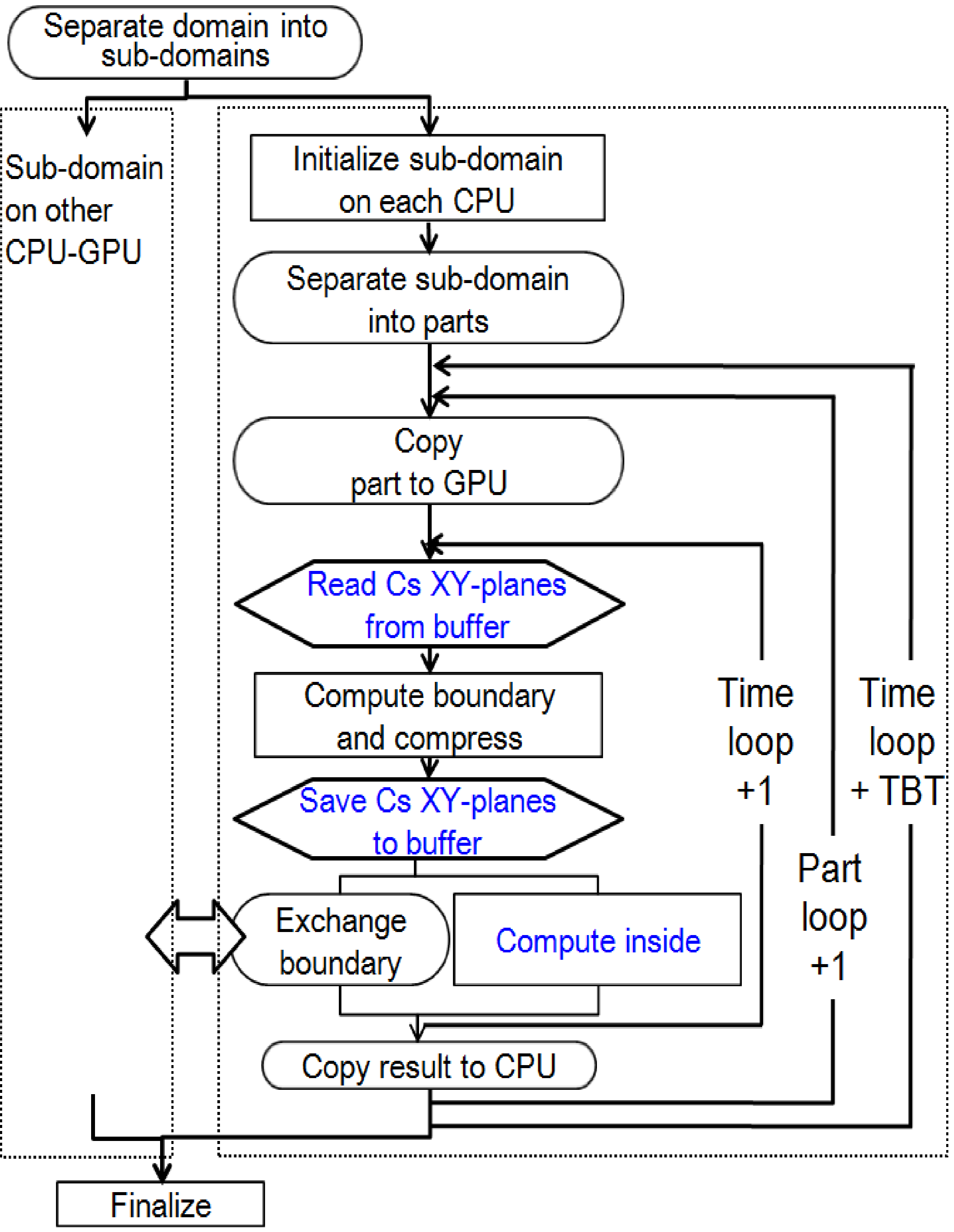
We can figure out the pseudo code as below:

```
Separate the domain into sub-domains;
Allocate Grid on each CPU to read initial of sub-domain;
Allocate Grid0 and Grid1 on each GPU to perform double buffering;
Separate domain into parts;
For (i = 0; i< TSI; i += 1)
{
    For (j = 0; j < NPS; j += 1)
    {
        Copy part j with more ghost boundaries from CPU to Grid0;
        For(k = 0; k<TBT; k+=1){
            Compute boundary;
            Compress boundary;
            Parallel Do
            {
                Compute inside;
                Exchange boundary;
            }
            Swap Grid0 and Grid1;
        }
        Swap Grid0 and Grid1;
        Copy result of sub-domain j from Grid 1 to CPU;
    }
}
```

On each node, it copies initial to the GPU side and compute multiple time steps on GPU side, and copy the result back. So, it can get high performance but also causes redundant cost.

5.8 Contribution: 2D-1D-TBM method

The process of the 2D-1D-TBM method is below:



Separate domain into sub-domains
Sub-domain on other CPU-GPU
Initialize sub-domain on each CPU
Separate sub-domain into parts
Copy part to GPU
Read Cs XY-planes from buffer
Compute boundary and compress
Save Cs XY-planes to buffer
Exchange boundary
Compute inside
Copy result to CPU
Time loop +1
Part loop +1
Time loop + TBT
Finalize

Fig.99. The process of 2D-1D-TBM method

We can figure out the pseudo code as below:

```
Separate the domain into sub-domains;
Allocate Grid on each CPU to read initial of sub-domain;
Allocate Grid0 and Grid1 on each GPU to perform double buffering;
Allocate Buffer on each GPU;
Separate domain into parts;
For (i = 0; i< TSI; i += 1)
{
    For (j = 0; j < NPS; j += 1)
    {
        Copy part j from CPU to Grid0;
        For(k = 0; k<TBT; k+=1){
            Read Cs XY-planes from Buffer;
            Compute boundary;
            Compress boundary;
            Save Cs XY-planes to Buffer;
            Parallel Do
            {
                Compute inside;
                Exchange boundary;
            }
            Swap Grid0 and Grid1;
        }
        Swap Grid0 and Grid1;
        Copy result of sub-domain j from Grid 1 to CPU;
    }
}
```

On the each device memory, it can reduce the communication cost between the GPU and CPU. By reusing the data on the buffer of GPU, it can solve redundancy problem. To perform 2D-1D-TBM, we should allocate 2 grids (performs computation) and 1 buffer (performs buffer-copy) and buffer for boundary on GPU side. We name TBT as temporal blocking times on GPU side. We name Dzp as the Z dimension size of each part. Here NPS stands for the number of parts of each sub-domain. We should set TBT as big as possible, and it should not bigger than Dzp (Z size of part). Dzp = Dz / NPS, Cs = 2. If use double buffering, (Dx/Nx)×(Dy/Nx)×(Dzp+Cs)×2+(Dx/Nx)×(Dy/Ny)×TBT× Cs+(Dx+Dy) ×(Dzp)×2 ≤ GPU memory capacity, TBT < Dzp, Get max TBT.

5.9 Evaluation on TSUBAME2.0

5.9.1 Strong scalability of 7-point and 19-point

We first evaluated 2D-1D-TBM method on multiple nodes of TSUBAME2.0. We select 7-point and 19-point stencil to evaluate. The 3D domain is 5120×5120×2560. We increase the number of GPUs from 16 to 256.

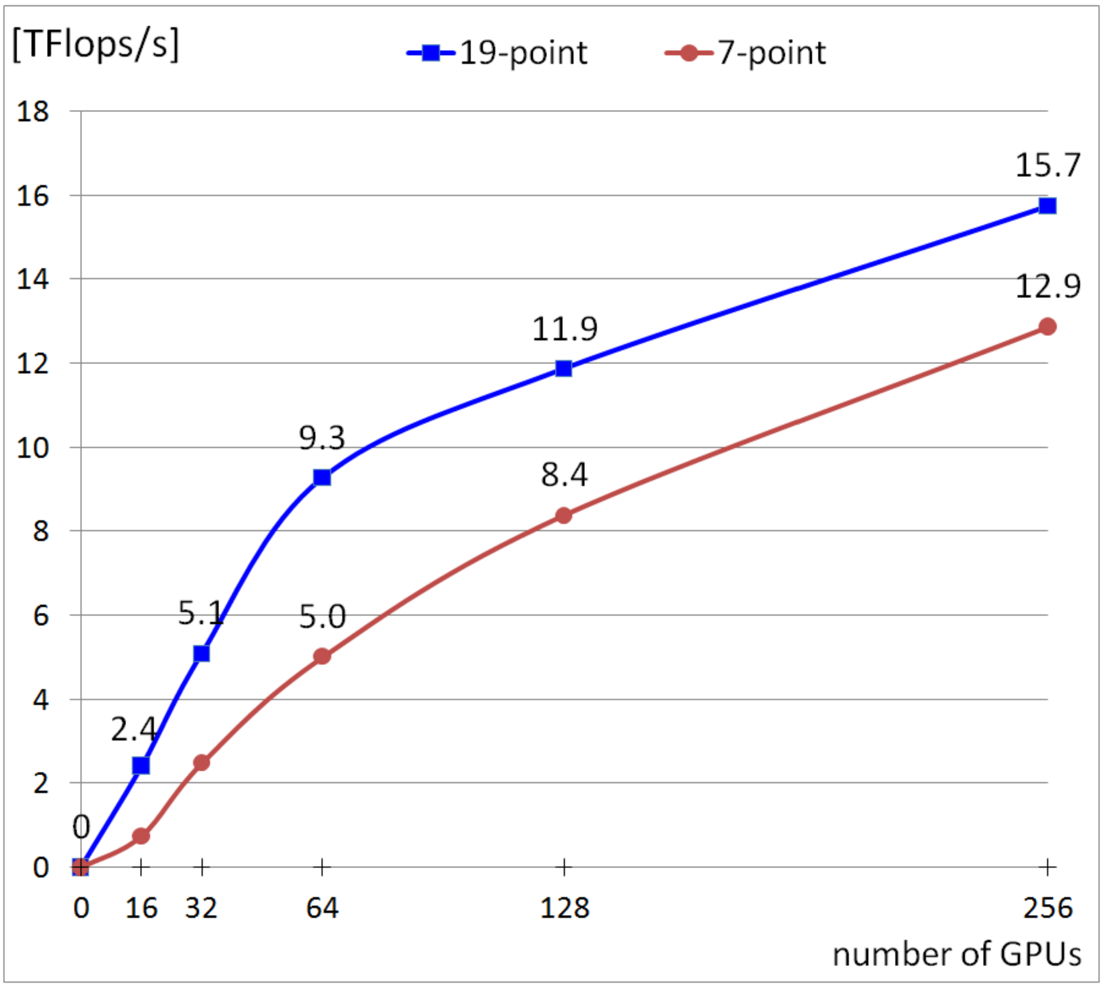


Fig.100. Strong scalability of 2D-1D-TBM for 7-point and 19-point

As we can see that it achieves high performance when the number of GPUs increases. Common way needs 205 GPUs to compute domain 5120×5120×2560. 2D-1D-TBM method can compute 40 times bigger domain than the common way. The domain size is only depends on the memory capacity of CPUs when using 2D-1D-TBM method.

5.9.2 Weak scalability of 7-point and 19-point

We evaluated 2D-1D-TBM method on multiple GPUs of TSUBAME2.0. We select 7-point and 19-point stencil. The sub-domain on each CPU is (640×640×1920). We increase the number of GPUs from 16 to 256.

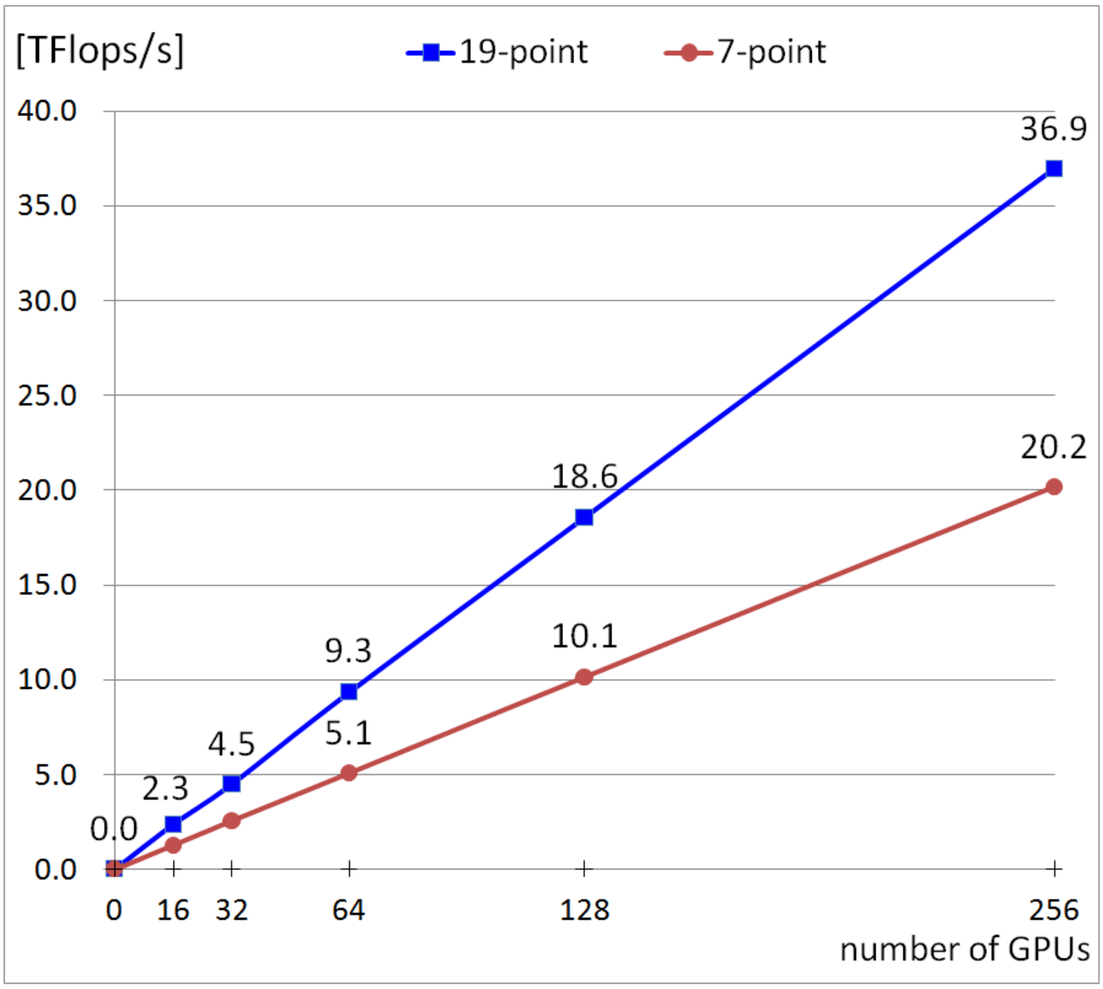


Fig.101. Weak scalability of 2D-1D-TBM for 7-point and 19-point

As we can see that it achieves high performance when the number of GPUs increases. It can compute bigger domain than the common way while reaches high performance.

5.9.3 Strong scalability: 2D-1D-TBM Vs other methods

We evaluated 2D-1D-TBM, 2D-1D-T, and 2D-1D-N method on multiple GPUs of TSUBAME2.0. We select 19-point stencil forms. The 3D domain is 5120×5120×2560. We increase the number of nodes from 16 to 256.

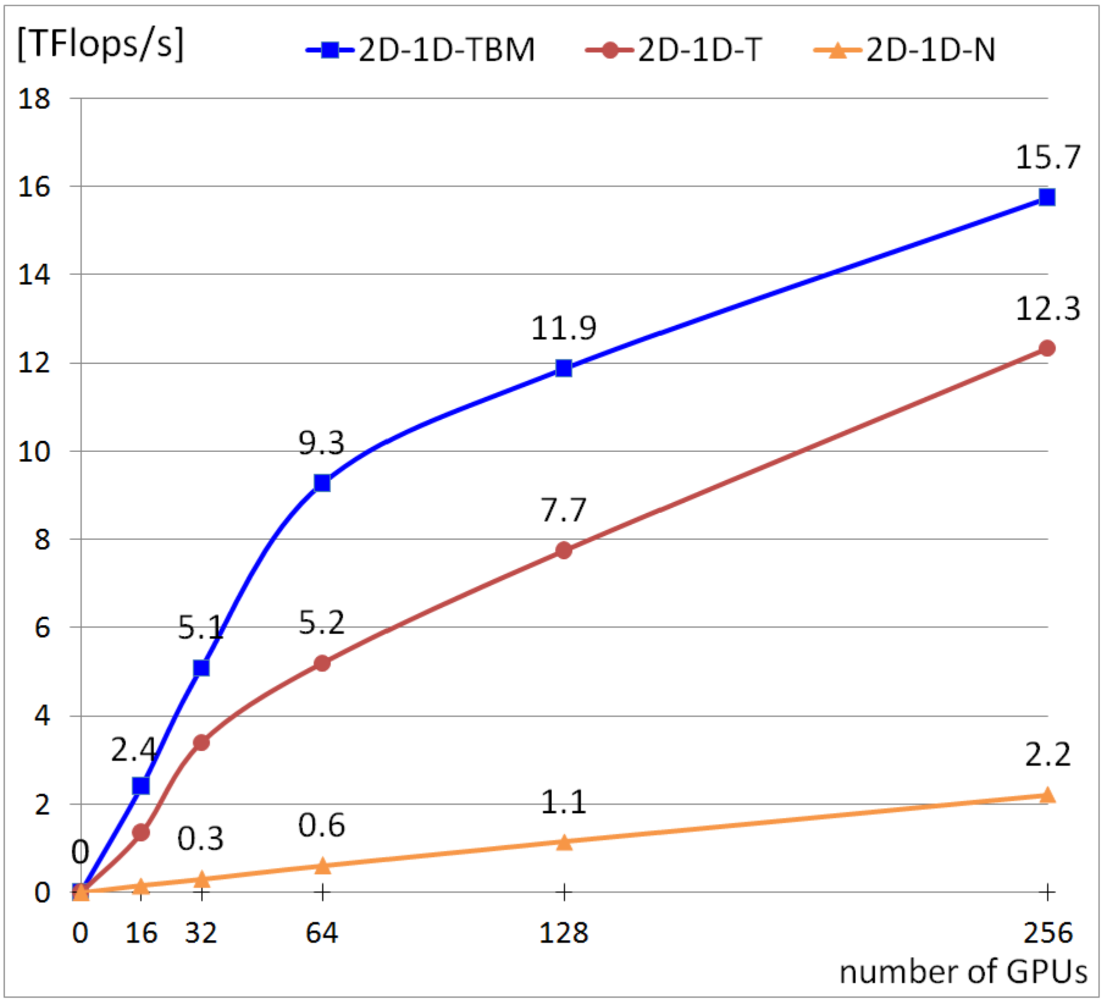


Fig.102. Strong scalability of 2D-1D-TBM, 2D-1D-T, 2D-1D-N for 19-point

As we can see that 2D-1D-TBM has 1.58 times better performance than the other methods. Common way needs 205 GPUs to compute domain 5120×5120×2560. 2D-1D-TBM method can compute 40 times bigger domain than the common way. The domain size is only depends on the memory capacity of CPUs when using 2D-1D-TBM method.

5.9.4 Weak scalability: 2D-1D-TBM Vs other methods

We evaluated 2D-1D-TBM, 2D-1D-T, and 2D-1D-N method on multiple GPUs of TSUBAME2.0. We select 19-point stencil. The sub-domain on each CPU is 640×640×1920. We increase the number of GPUs from 16 to 256.

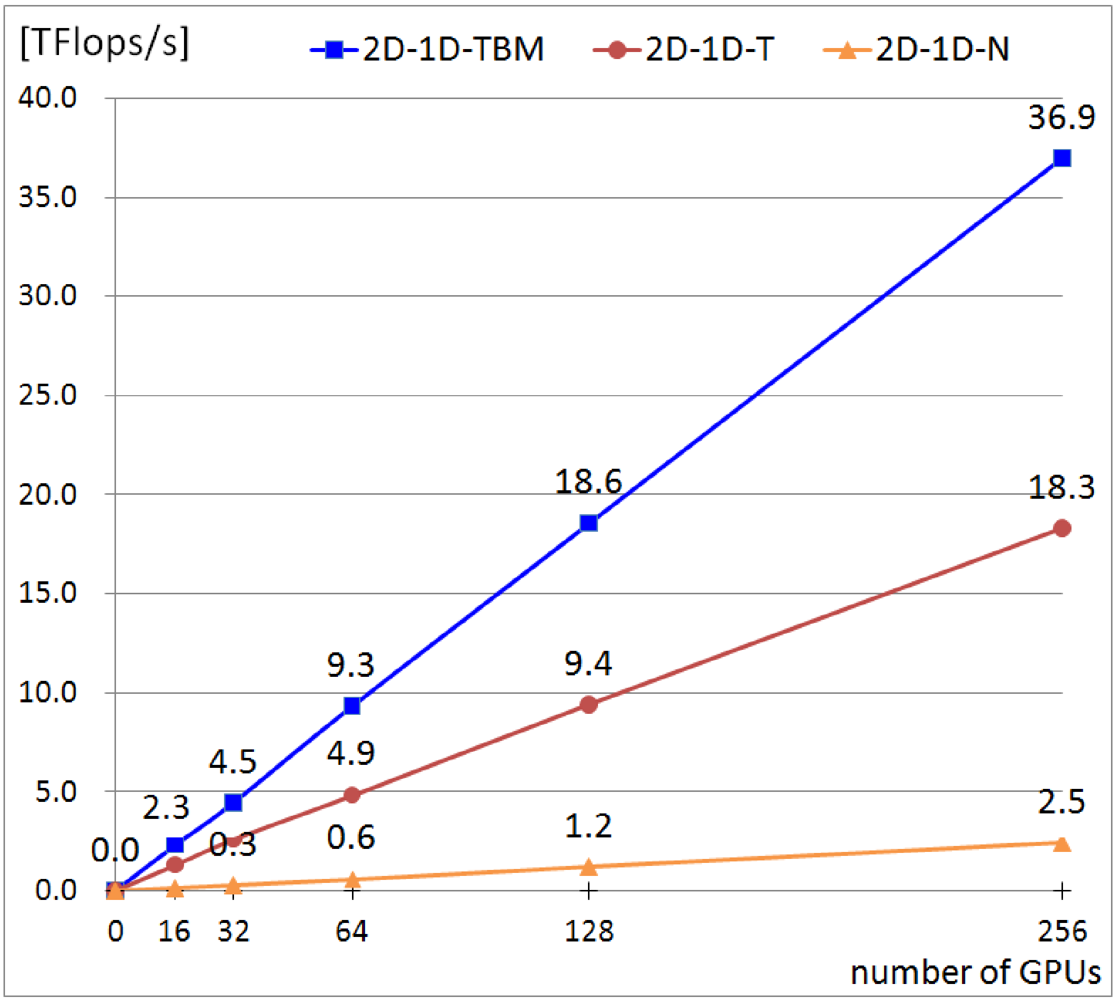


Fig.103. Weak scalability of 2D-1D-TBM, 2D-1D-T, 2D-1D-N for 19-point

As we can see that 2D-1D-TBM has 1.88 times better performance than the other methods. 2D-1D-TBM, 2D-1D-T, 2D-1D-N can compute bigger domain than the common way while reaches high performance.

5.9.5 Various domains

We evaluated 2D-1D-TBM on multiple GPUs of TSUBAME2.0. We select 19-point stencil form. As it achieves scalability in two dimensions among the nodes, we only need to evaluate the scalability in the other dimension. So, we set the domain size is

2560×2560×Z. Then we increase the size of Z to evaluate our method. We set the number of nodes is 16.

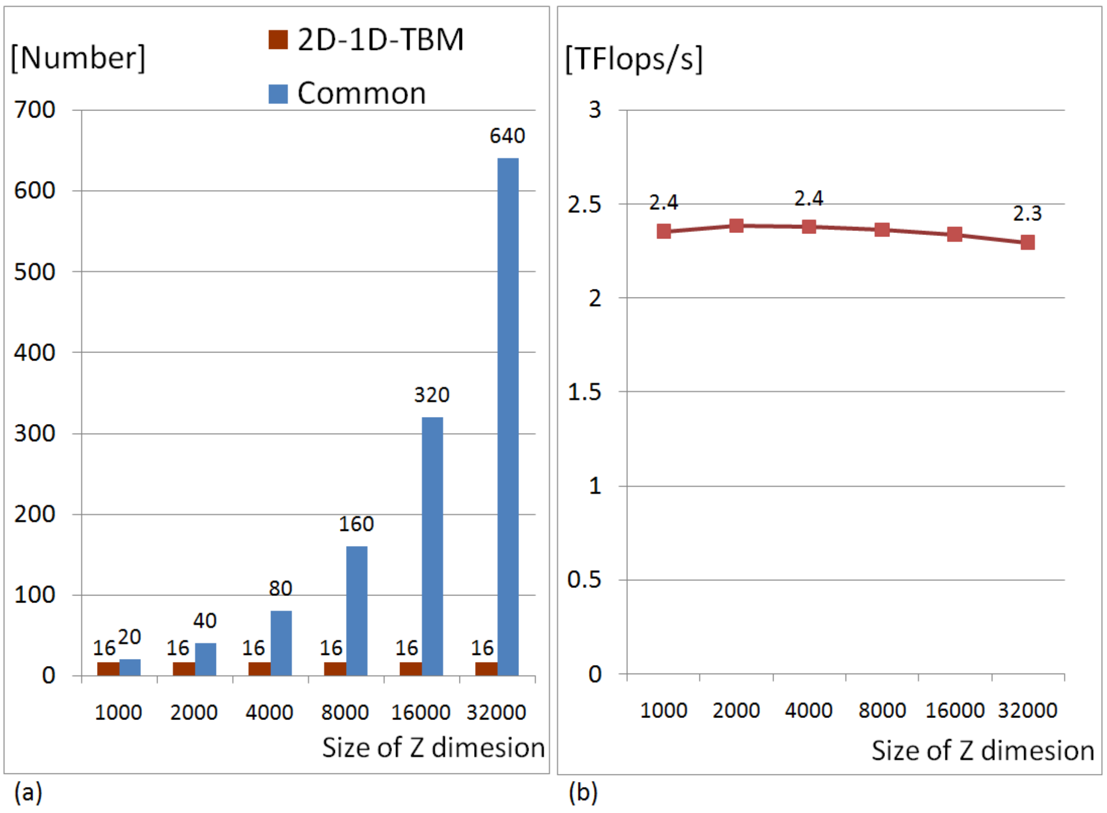


Fig.104. (a) The number of nodes that is needed (b) Related performance

We increase the Z from 2000 to 32000. As the domain increases, the common way needs more GPUs to compute big domain. Yet, 2D-1D-TBM method only uses 16 GPUs to compute this big domain. Finally, it can compute 40 times bigger domain than the common way while reaches high performance.

5.9.6 Time evaluation

Below figure (a) is the elapsed time of strong scalability of 2D-1D-TBM for 19-point stencil. Figure (b) is the elapsed time of weak scalability of 2D-1D-TBM for 19-point stencil. The 3D domain is 5120×5120×2560. We increase the number of nodes from 16 to 256. We measure the time for 760 time steps. We try to figure out the relationship between total, communication, computation time. We observe that the total time is similar to the time of computation. It proves that the inside computation well covers the boundary exchange and the cost of communication between CPU and GPU is reduced. The cost of sending data between CPU and GPU is reduced by optimized temporal blocking method. The computation time is the main factor that affects the performance. So, we need to improve the performance of computation in future.

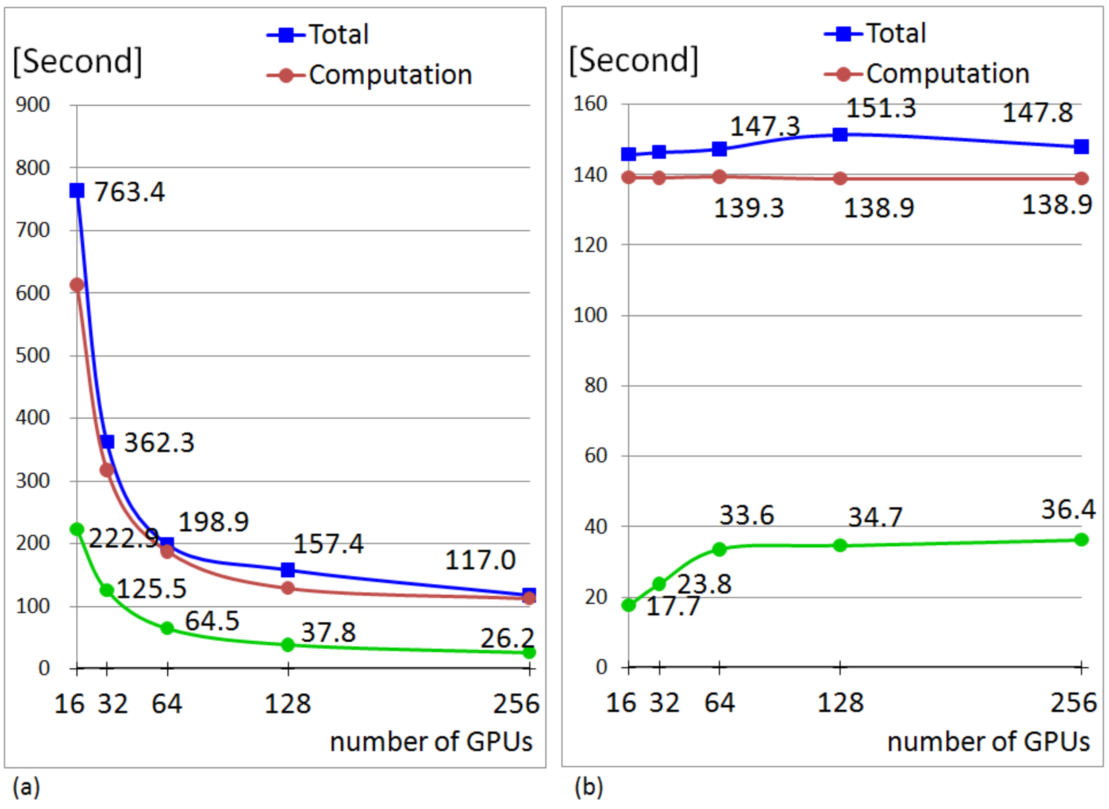


Fig.105. Elapsed time of strong scalability and weak scalability

5.9.7 Parameters

To perform 2D-1D-TBM, we should allocate 2 grids (performs computation) and 1 buffer (performs buffer-copy) and boundaries on GPU side. We name TBT as temporal blocking times on GPU side. We name Zp as the Z dimension size of each part. Here NPS stands for the number of parts of each sub-domain. To reduce communication times between CPU and GPU, we should set TBT as big as possible. So, we can get the (TBT, NPS) pairs for different number of GPUs. Figure (a) shows the relationship between NPS and TBT. As NPS grows, Zp decrease and TBT grows. As we set TBT no more than Zp, TBT grows until it reaches the Zp. Figure (b) shows the performance of those (NPS, TBT) pairs. As it shows, the max point of TBT gets max performance. Also, if the TBT remains big enough, the communication time between CPU and GPU remains small since it is divided by TBT.

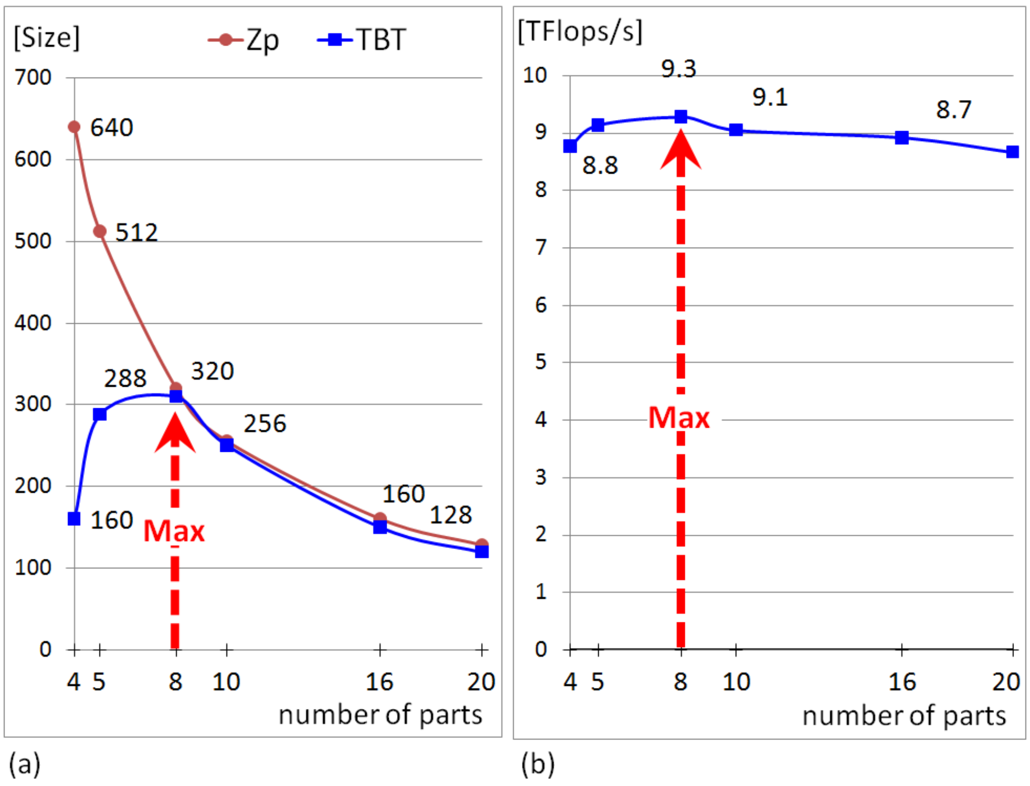


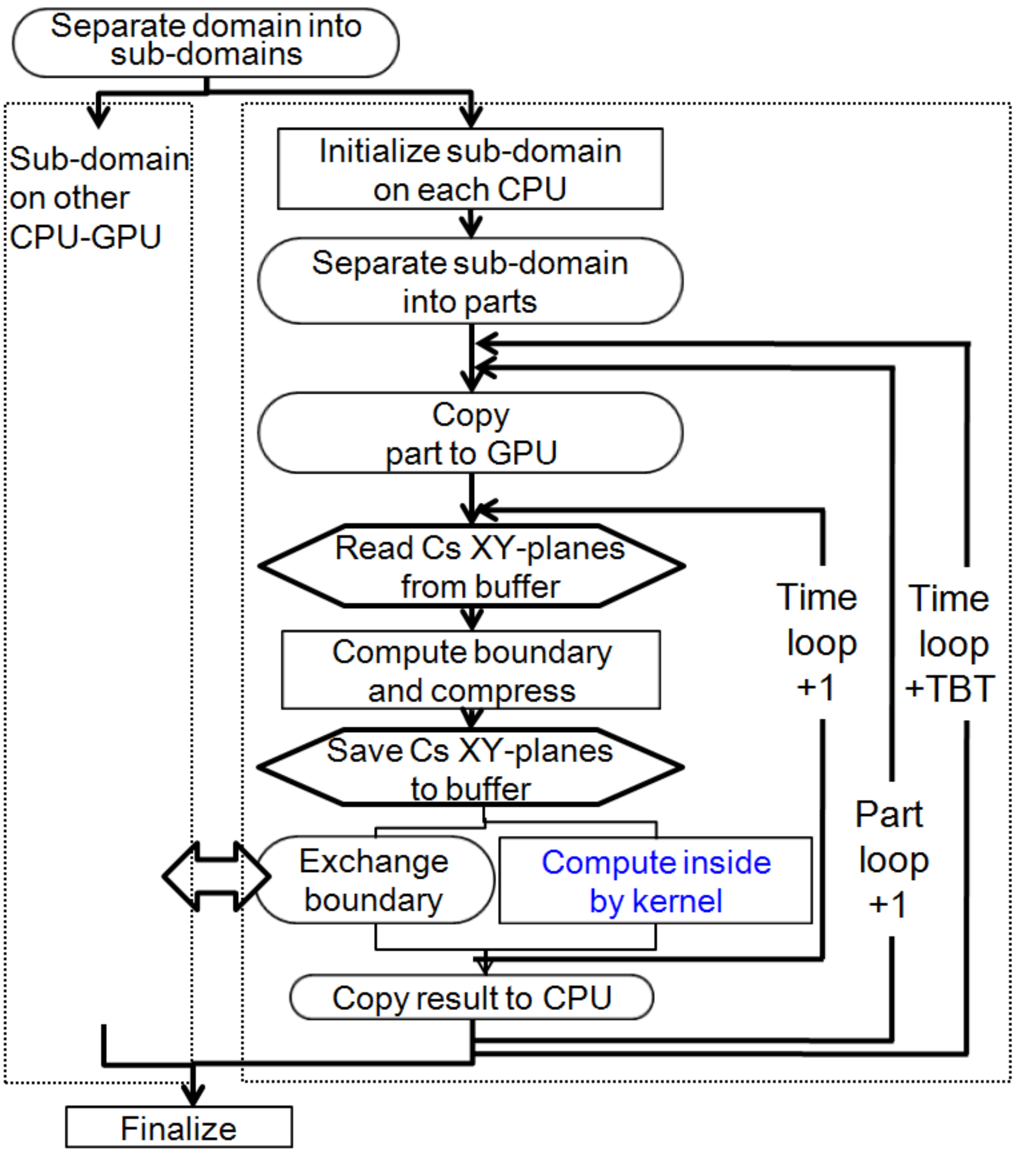
Fig.106. (a) Relationship of parameters (b) Related performance

5.10 Contribution: 2D-1D-TBMR method

Here Dx is X dimension size of sub-domain, Dy is y dimension size of sub-domain. It also reuses the space on the register. We can figure out the pseudo code as below:

```
Separate the domain into sub-domains;
Separate domain into parts;
For (i = 0; i< TSI; i += 1)
{
    For (j = 0; j < NPS; j += 1){
        Copy part j from CPU to Grid0;
        For(k = 0; k<TBT; k+=1){
            Read Cs XY-planes from Buffer;
            Compute boundary;
            Compress boundary;
            Save Cs XY-planes to Buffer;
            Parallel Do{
                Compute inside with in-plane and memory-saving method;
                Exchange boundary;}
            Swap Grid0 and Grid1;
        }
        Swap Grid0 and Grid1;
        Copy result of sub-domain j from Grid 1 to CPU;
}}
```

On each GPU, it computes multiple time steps to reduce the communication cost. It also efficiently uses the registersto solve redundancy problem. The process of the 2D-1D-TBMR method is below:



Separate domain into sub-domains
Sub-domain on other CPU-GPU
Initialize sub-domain on each CPU
Separate sub-domain into parts
Copy part to GPU
Read Cs XY-planes from buffer
Compute boundary and compress
Save Cs XY-planes to buffer
Exchange boundary
Compute inside by kernel
Copy result to CPU
Time loop +1
Time loop +TBT
Part loop +1
Finalize

Fig.107. The process of 2D-1D-TBMR method

5.11 Reevaluation on TSUBAME2.5

We reevaluate our method 2D-1D-TBMR with former 2D-1D-TBM and the existing 2D-1D-T and 2D-1D-N. The domain is 5120×5120×2560. We select 7 point stencil of floating computation.

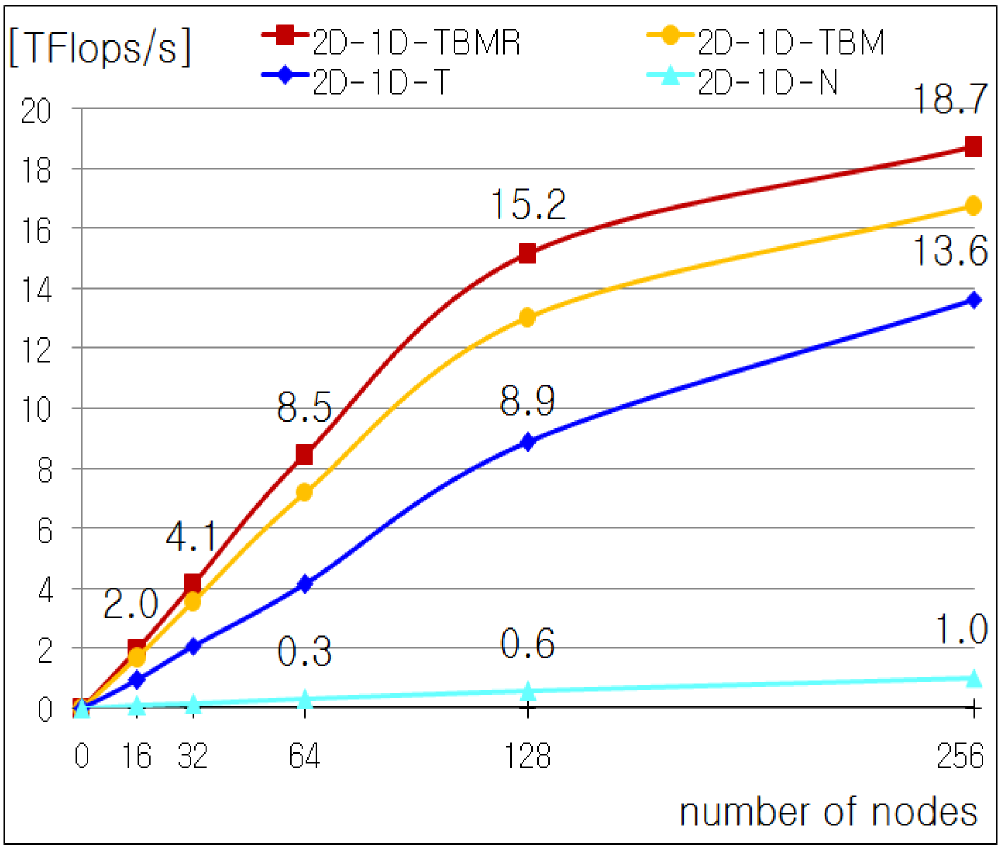


Fig.108. Evaluation on TSUBAME2.5

As we can see in the upper figure, 2D-1D-TBMR method gets 1.59 times higher performance than 2D-1D-TBM and 1.84 times higher performance than 2D-1D-T method in 7 point stencil case.

We reevaluate our method 2D-1D-TBMR with former 2D-1D-TBM and the existing 2D-1D-T and 2D-1D-N. The domain is 5120×5120×2560. We select 19 point stencil of floating computation. In 19 point stencil case, 2D-1D-TBMR method gets 1.71 times higher performance than 2D-1D-TBM and 2.68 times higher performance than 2D-1D-T. As 19 point stencil reads more nearby points, it is more important to use registers to reduce access time.

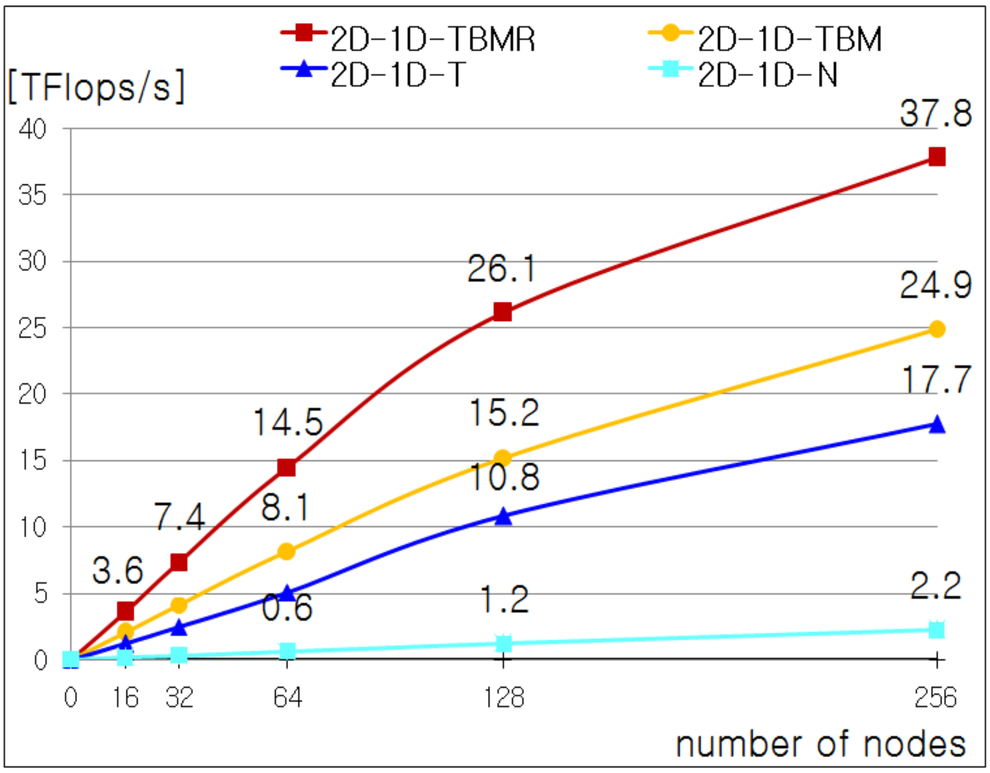


Fig.109. Evaluation on TSUBAME2.5

The each GPU on TSUBAME2.0 has 3GB memory. The each GPU on TSUBAME2.5 has 6GB memory. So, the max point on TSUBAME2.5 comes earlier than on TSUBAME2.0.

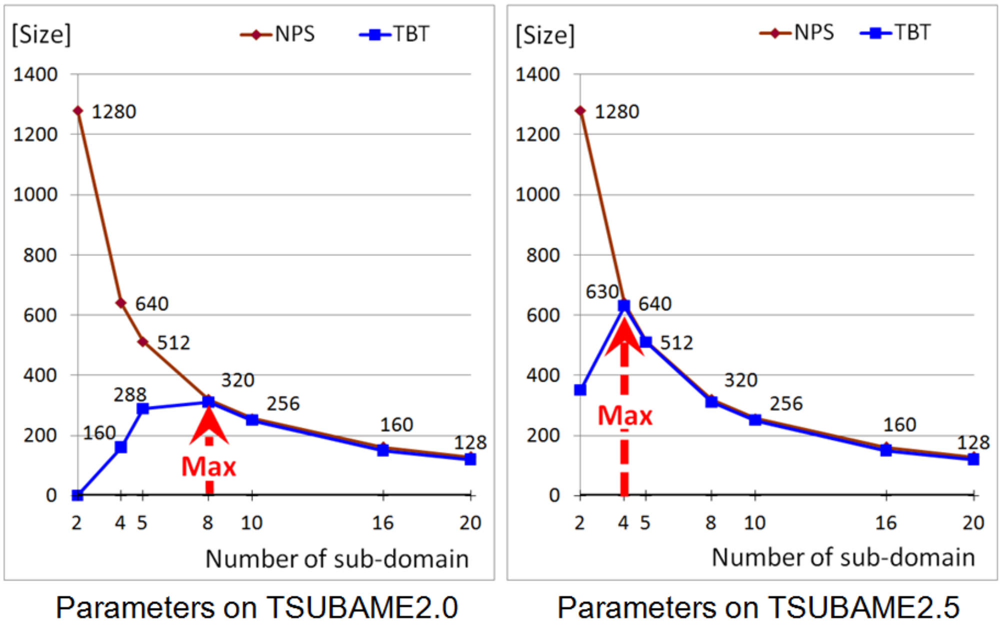


Fig.110. Parameters on TSUBAME2.0 and TSUBAME2.5

5.12 Summary

In this chapter, we proposed 2D-1D-TBMR, 2D-1D-TBM methods for multiple nodes. The evaluation on multiple nodes of TSUBAME system shows that our optimization methods get higher performance than the existing optimization methods.

To achieve 3D scalability on 3D domain, we first propose new 2D decomposition method to separate the domain among the nodes. Different from the existing 2D optimization methods, it separates the domain by X and Y dimension. The existing decomposition method decomposition method separates the domain by Y and Z dimension which can benefit the implementation of common way. As the optimization methods only compute un-overlapped part on the GPU side, our decomposition method can benefit the implementation of applying 1D optimization methods on each node.

On each node, we apply the 1D optimization methods to reduce the communication cost between CPU and GPU. We overlap the boundary exchange with the inside computation to reduce the communication between nodes.

By the combination of 2D decomposition and 1D optimization methods for 3D domain, it achieves 3D scalability and gets different optimization methods in multiple nodes case. As it computes sub-domain that is bigger than the memory capacity of GPU on each node, it can compute the domain that is bigger than the memory capacity of GPUs on multiple node.

The evaluation of the performance shows that our optimization methods can compute bigger domain than the common way while maintaining higher performance than existing optimization methods on multiple nodes on TSUBAME systems. We also analyze the performance and parameters difference between TSUBAME2.5 and TSUBAME2.0. Although it increased the memory capacity of each GPU on TSUBAME2.5, it also needs to compute bigger domain by using optimization methods. We published 1 papers depend on 2D-1D-TBM method.

[38] Guanghao Jin, Toshio Endo, 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 2013, Indianapolis, September 2013.
(International, paper, refereed, and already published)

Chapter 6. Optimization methods for band SpMV on single node

6.1 Introduction

The common way to use GPU to compute matrix vector multiplication is to initialize the matrix and vector on the CPU [12], [13], [14], [15], [16], [17]. Then, it copies the matrix and vector to the GPU side. On the GPU side, it continues the computation for multiple time steps.

The compressed sparse row (CSR) format is perhaps the most popular general-purpose sparse matrix representation [18], [19], [20], [21], [22]. CSR explicitly stores column indices and nonzero values in arrays indices and data. A third array of row pointers, ptr, allows the CSR format to represent rows of varying length. Below figure illustrates the CSR representation of an example matrix.

CSR format:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \qquad \begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Fig.111. CSR format

One-Thread-One-Row maps each dot production between row and vector to a thread. One-Warp-One-Row maps each dot production between row and vector to one warp of threads as bellow:

```
_global_ void spmv_cscvectockernel (const int, num_rows, const int *Ptr, const int
*Indices, const float *Data, const float *x, float *y) {
   _shared_ float vs[32];
   int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
   int warp_id = thread_id / 32;
   int la = thread_id & (32 - 1);
   int row = warp_id;
   if(row < num_rows){
    int row_start = ptr[row];
    int row_end = ptr[row + 1];
    vs[threadIdx.x] = 0;
    for(int j = row_start + la; j < row_end; j += 32)
        vs[threadIdx.x] += data[j] * x[indices[j]];
    if(la < 16) vs[threadIdx.x] += vs[threadIdx.x + 16];
    if(la < 8) vs[threadIdx.x] += vs[threadIdx.x + 8];
    if(la < 4) vs[threadIdx.x] += vs[threadIdx.x + 4];
    if(la < 2) vs[threadIdx.x] += vs[threadIdx.x + 2];
    if(la < 1){
        vs[threadIdx.x] += vs[threadIdx.x + 1];
        y[row] = vs[threadIdx.x];
}}}
```

If a group of cooperating threads is assigned to only one row, then the number of blocks required to process the entire matrix may be more than 65535. To complement this approach more than one row per cooperating thread group can be processed.

```
__global__ void csrmv (float *values, int *rowPtrs, int *colIdxs, float *x, float *y, int
dimRow, int repeat, int coop) {
  int i = (repeat*blockIdx.x*blockDim.x + threadIdx.x)/coop;
  int coopIdx = threadIdx.x%coop;
  int tid = threadIdx.x;
  extern __shared__ volatile float sdata[];
  for (int r = 0; r<repeat; r++) {
    float localSum = 0;
    if (i<dimRow) {
      // do multiplication
      int rowPtr = rowPtrs[i];
      for (int j = coopIdx; j<rowPtrs[i+1]-rowPtr; j+=coop) {
        localSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
      }
```

```
        // do reduction in shared mem
        sdata[tid] = localSum;
        for(unsigned int s=coop/2; s>0; s>>=1) {
          if (coopIdx < s) sdata[tid] += sdata[tid + s];
        }
        if (coopIdx == 0) y[i] = sdata[tid];
          i += blockDim.x/coop;
}}}}
```

6.1.1 Diagonal matrix

In diagonal case, as there is no data dependence between the sub-vectors, it can continue the computation locally on GPU side without the communication.

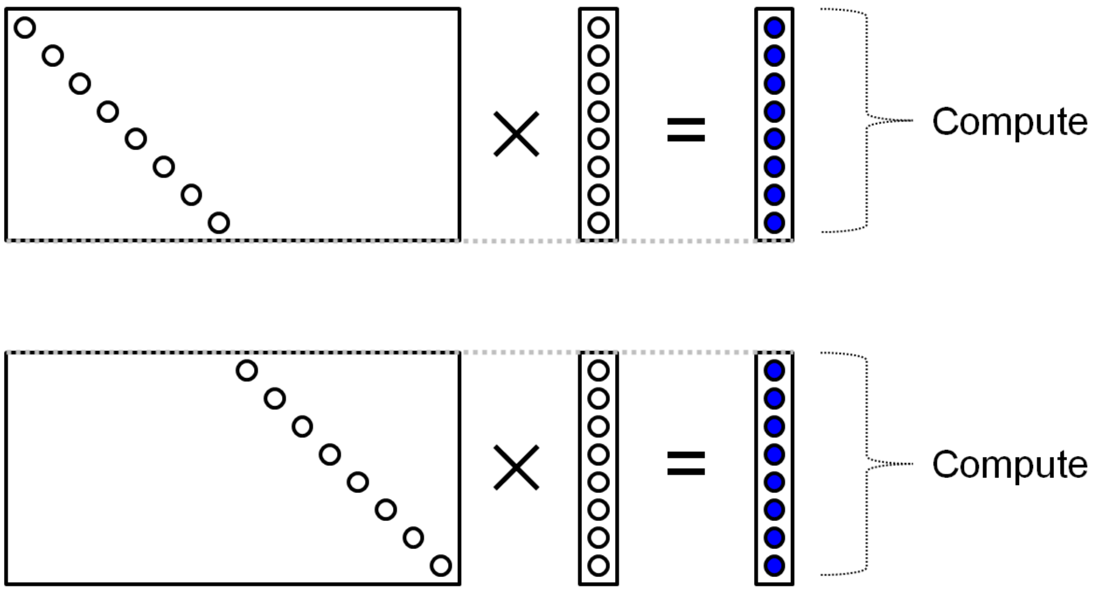


Fig.112. Decomposition of diagonal matrix

6.1.2 Tri-diagonal matrix

In some case like sparse tri-diagonal matrix case, there is data dependence between separated sub-vectors by the structure of the tri-diagonal matrix. For this kind of case, the sub-vectors need to exchange the dependent data to continue the computation as below.

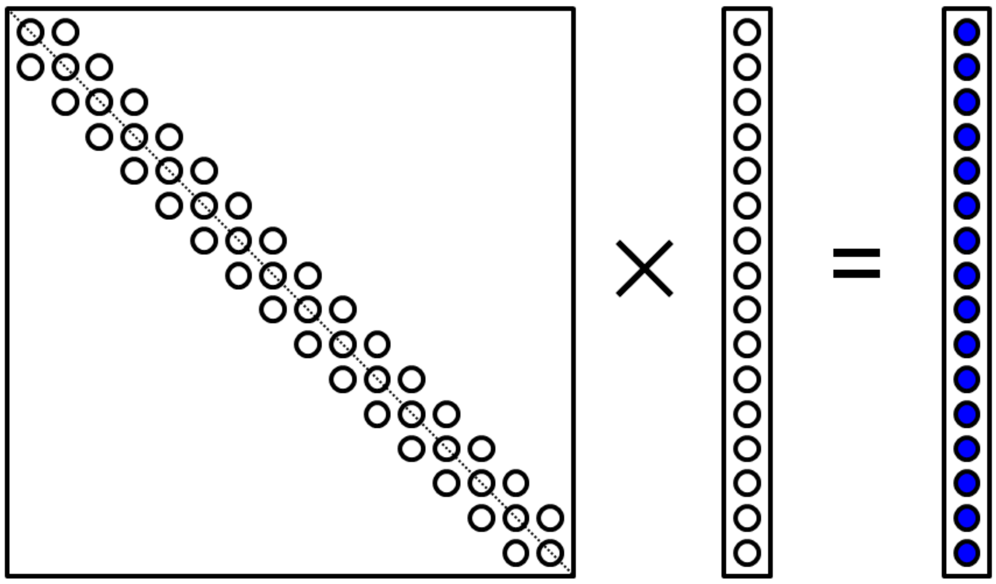


Fig.113. Tri-diagonal matrix

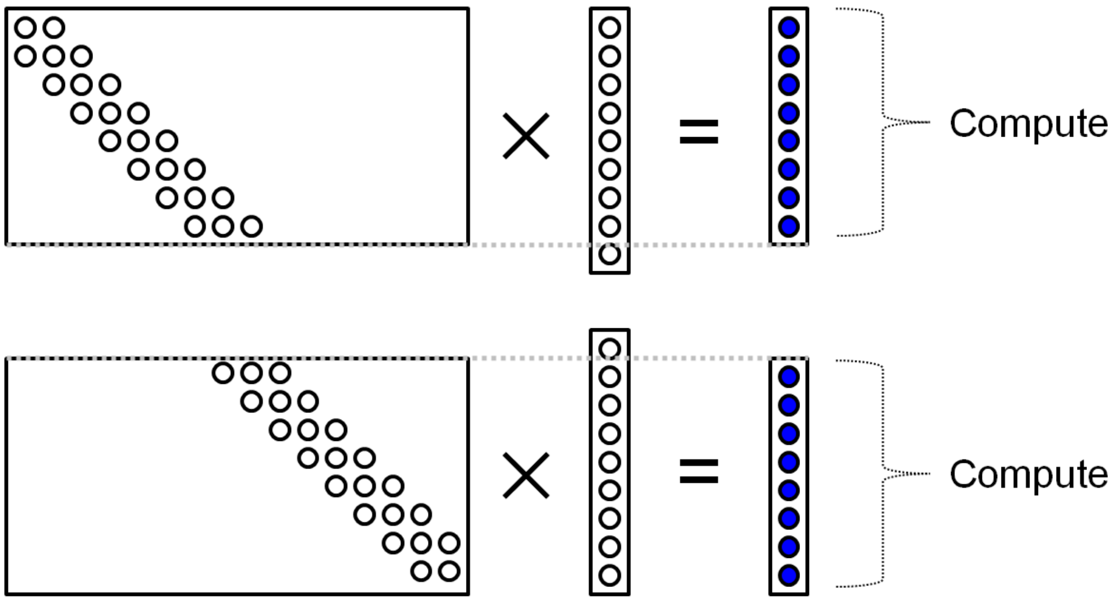


Fig.114. Decomposition of tri-diagonal matrix

To efficiently use the host and device memory, it needs to separate the whole matrix and vector into sub-matrix and sub-vector to the device memory. So, we need to consider the data dependence. We start the some special SpMV on single node as the base of our research.

In this section, we apply optimization method to sparse matrix case on single node. The optimization methods include 1D-N, 1D-T and 1D-TB (temporal blocking + buffer-copy).

6.1.3 Existing: 1D-N method

1D-N method separates the whole domain (matrix and vector) into sub-domain (sub-matrix and sub-vector). Then copy each sub-domain to GPU to compute 1 time step. So, it causes frequent communication between CPU and GPU.

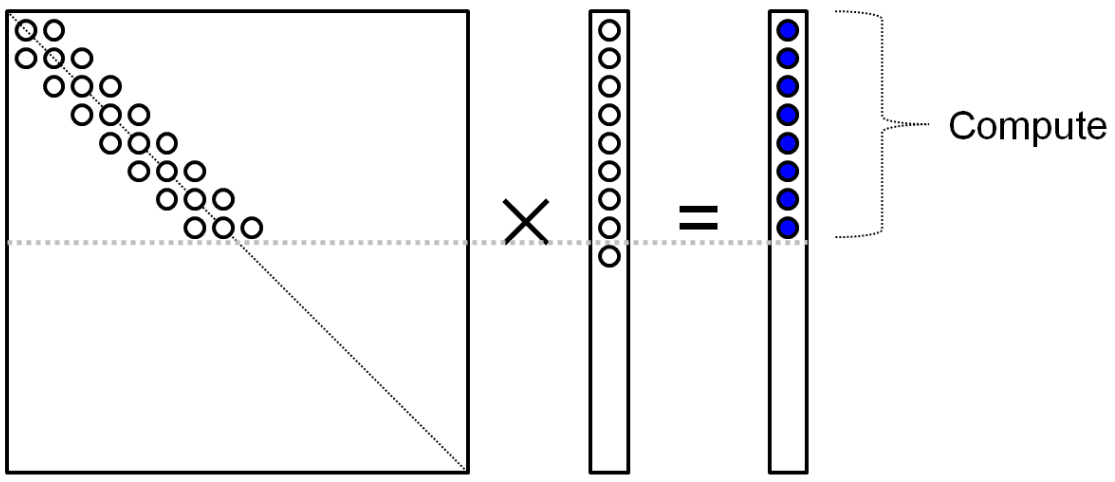


Fig.115. Compute current sub-domain for 1 time step

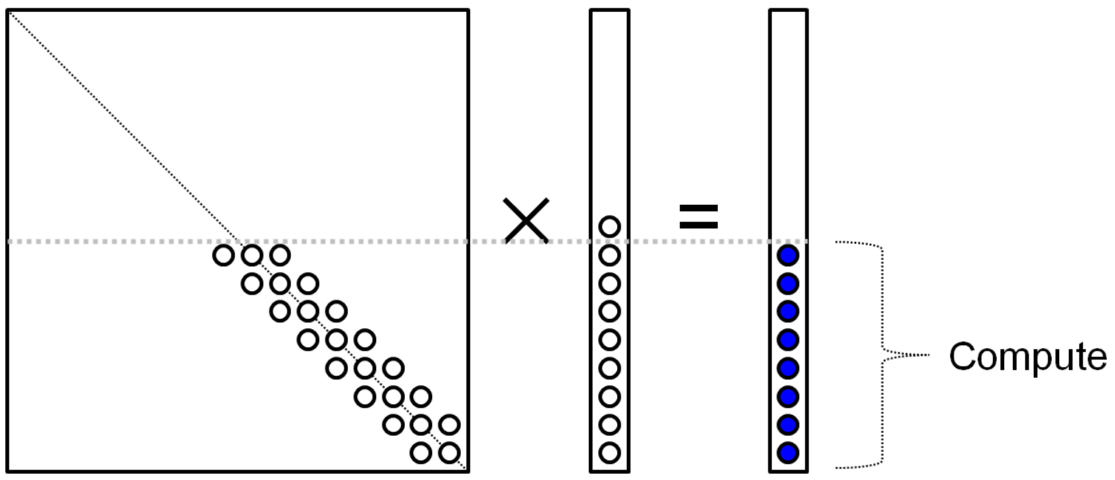


Fig.116. Compute next sub-domain for 1 time step

As we can see in the upper figure, the program model for SpMV is the same as the stencil case. It proves that our program model is correct for 1D-N in data dependence case.

6.2 Existing: 1D-T method

1D-T method separates the whole domain (matrix and vector) into sub-domain (sub-matrix and sub-vector). When it copies each sub-domain to GPU, it copies more rows of matrix and vector. So it can compute multiple time steps on the GPU.

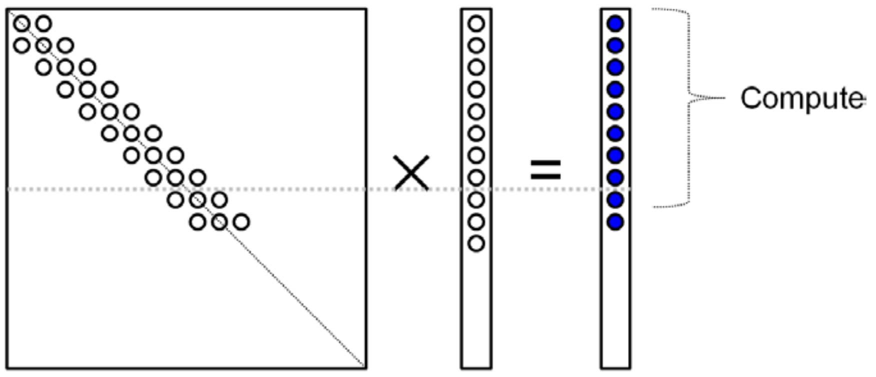


Fig.117. Compute current sub-domain for first time step

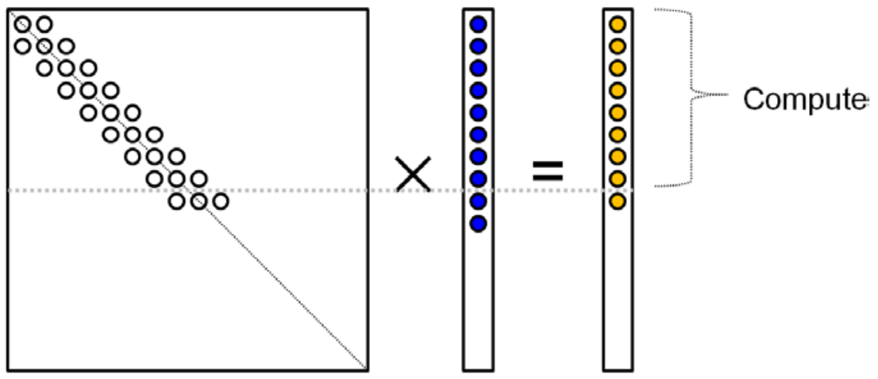


Fig.118. Compute current sub-domain for second time step

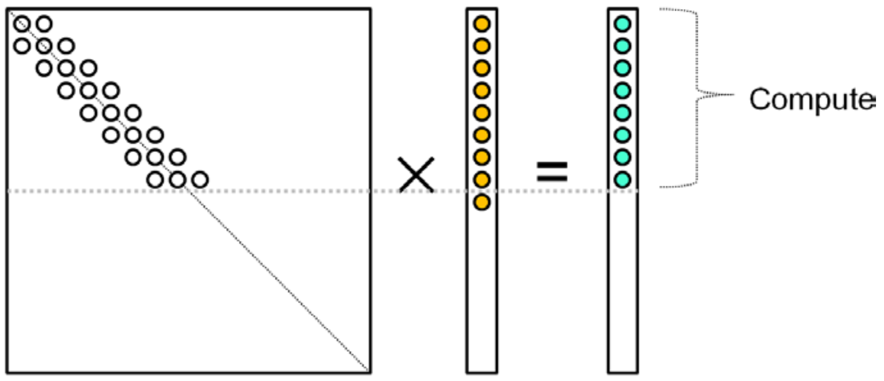


Fig.119. Compute current sub-domain for third time step

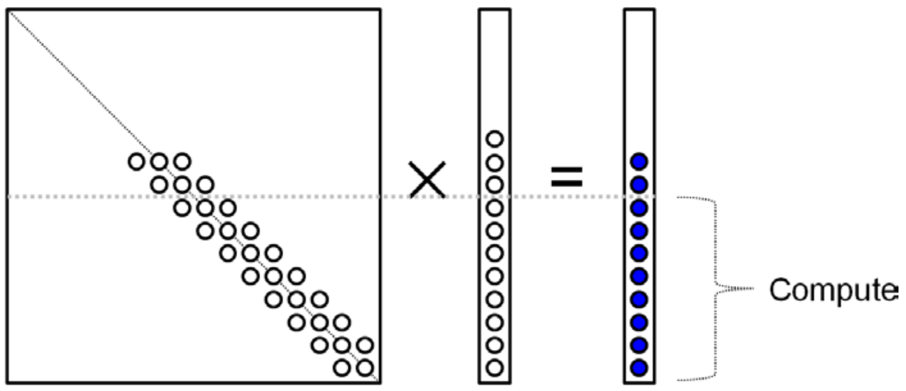


Fig.120. Compute next sub-domain for first time step

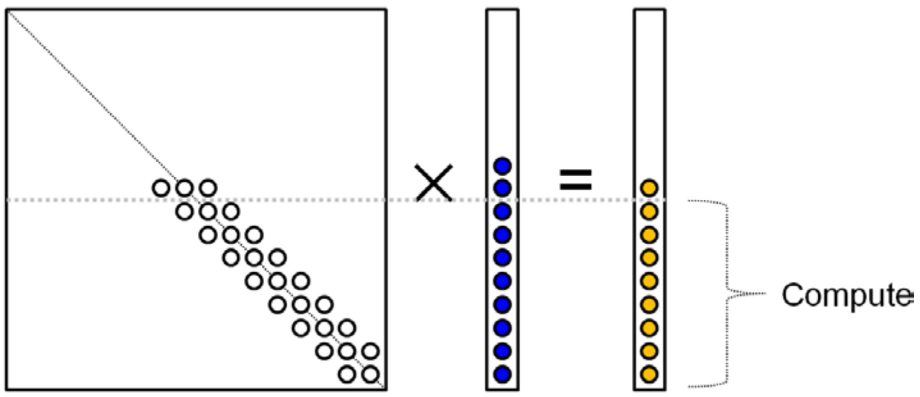


Fig.121. Compute next sub-domain for second time step

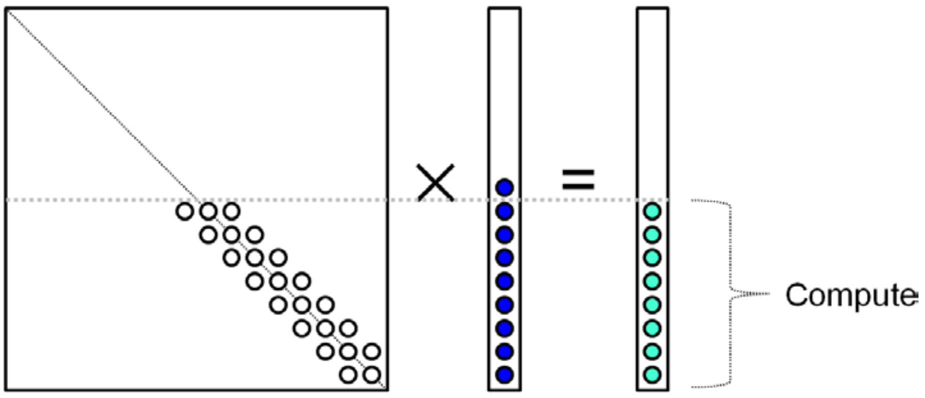


Fig.122. Compute next sub-domain for third time step

As we can see in the upper figure, the program model for SpMV is the same as the stencil case. It proves that our program model is correct for 1D-T in data dependence case.

6.3 Contribution: 1D-TB method

1D-TB (Temporal blocking + buffer-copy) separates the whole domain (matrix and vector) into sub-domain (sub-matrix and sub-vector). When it copies each sub-domain to GPU, it copies more rows of matrix and vector. So it can compute multiple time steps on GPU. To reduce the computation cost, it saves some result of current sub-domain to buffer at each time step. When it computes next sub-domain, it reads from buffer. By reusing those results, it can do less computation.

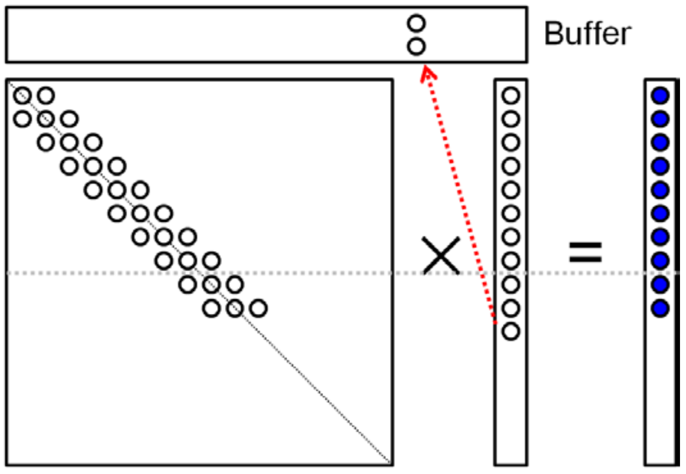


Fig.123. Compute current sub-domain for first time step and save to buffer

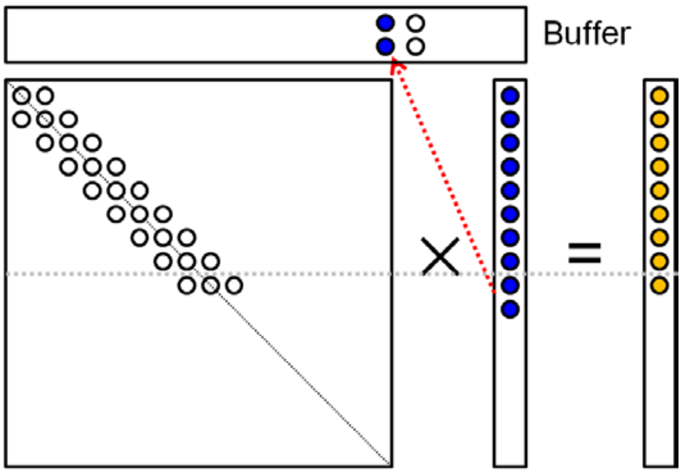


Fig.124. Compute current sub-domain for second time step and save to buffer

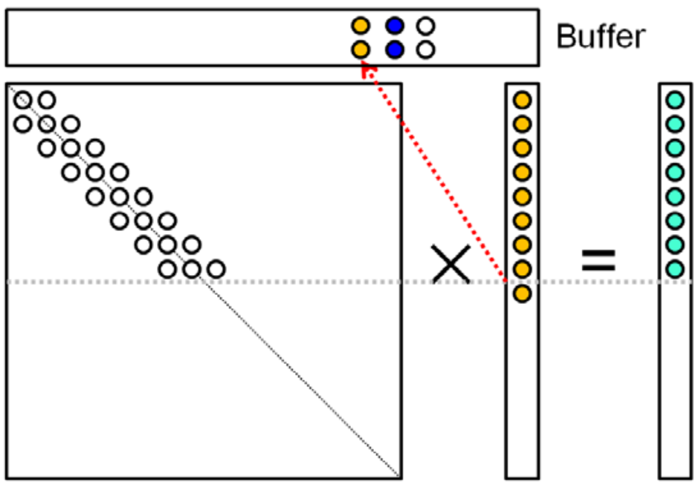


Fig.125. Compute current sub-domain for third time step and save to buffer

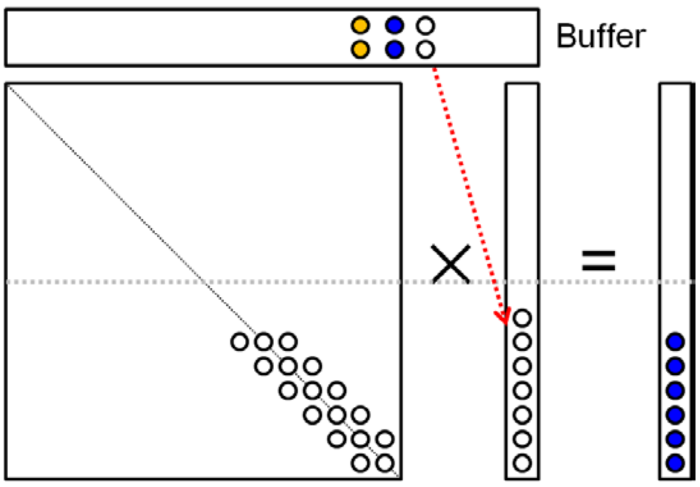


Fig.126. Read from buffer and compute next sub-domain for first time step

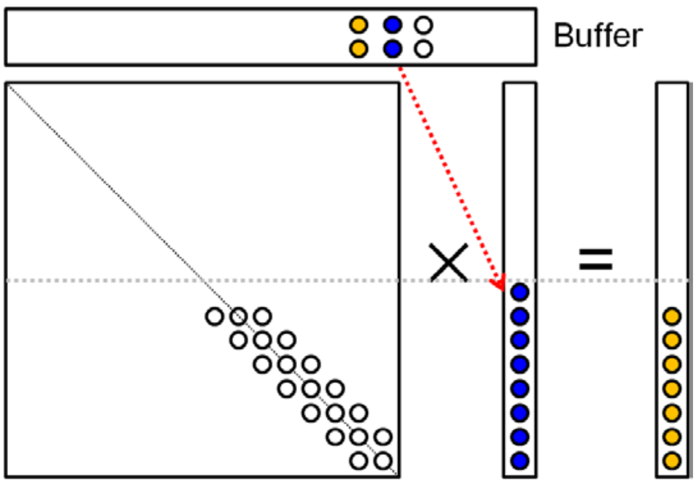


Fig.127. Read from buffer and compute next sub-domain for second time step

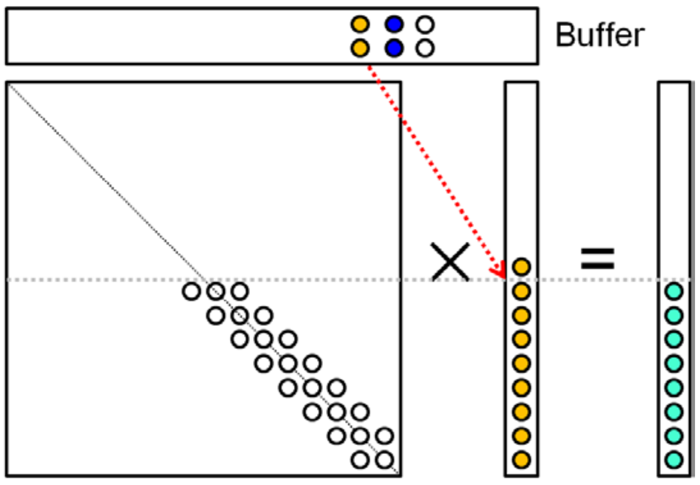


Fig.128. Read from buffer and compute next sub-domain for third time step

As we can see in upper figure, the program model for SpMV is the same as the stencil case. It proves that our program model is correct for 1D-T in data dependence case.

6.4 Evaluation on TSUBAME2.5

We evaluate optimization methods on single GPU of TSUBAME2.5. Bandwidth means the width of diagonal matrix.

6.4.1 1D-N vs 1D-TB

We set Bandwidth = 64, temporal blocking times = 800, time steps = 800 and increase the size of vector to compare Usual and TB method. In small matrix case, TB method performs 77% of usual method. In big matrix case, TB performs 15.2 times better than usual method. TB totally performs 8 times better performance than usual method.

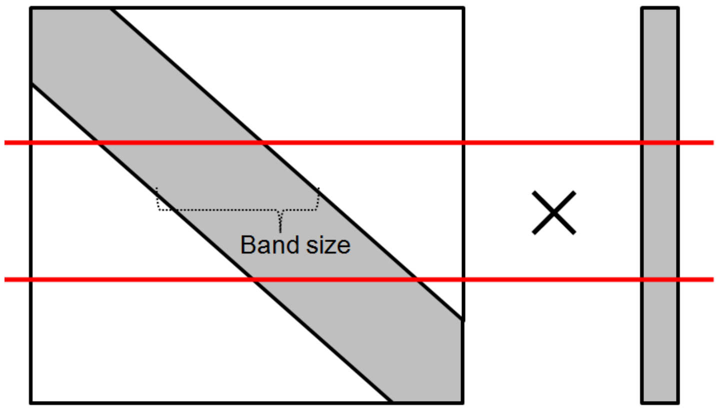
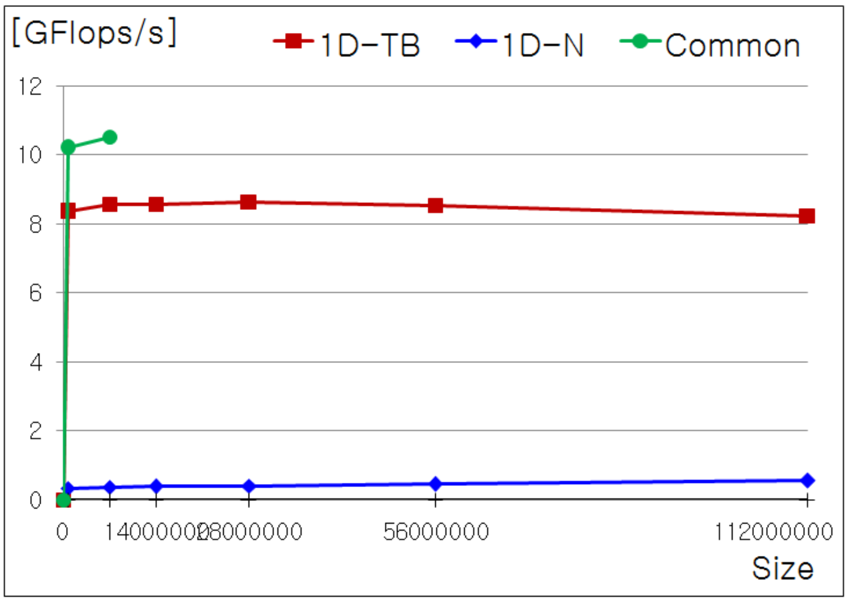


Fig.129. Band matrix and 1D decomposition.



6.4.2 1D-T vs 1D-TB

We set Bandwidth = 64, size of vector = 7×1.0e06, number of sub-domains =2, time steps = temporal blocking times, and increase temporal blocking times to compare 1D-TB and 1D-T method. 1D-TB has a little better performance than 1D-T.

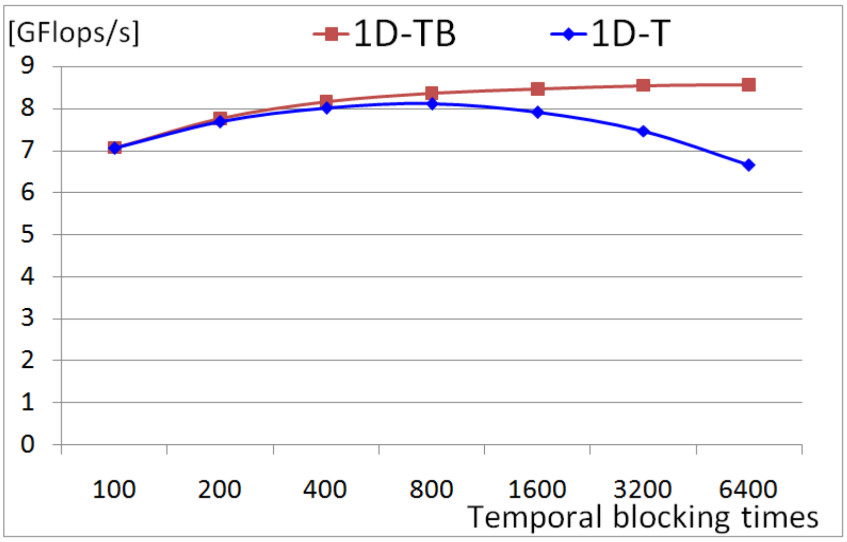


Fig.130. 1D-T vs 1D-TB

In 1D-T and 1D-TB method case, we found that when the temporal blocking time is big enough, the performance will not improved as the communication cost is reduced enough and will less affect the total time. So, we set TBT as below:

TBT< min{Dz/NSD, Communication(1)/computation(1)/Ratio} , get max TBT.

Chapter 7. Conclusion and directions for the future

In this thesis, we proposed optimization methods for stencil computation on 3D domain for single node and multiple nodes, and compared them with the existing optimization methods. The result of those optimization methods shows that our optimization methods enable the computation on the domain that is bigger than the memory of GPUs while maintaining higher performance than existing optimization methods. We combine the existing methods that can efficiently use the shared memory and registersof GPU to the optimization methods to maintain higher performance. We also evaluate our methods on TSUBAME2.0 and TSUBAME2.5. The result shows that our methods can achieve high performance on both TSUBAME2.0 and TSUBAME2.5. We analyzed the performance and parameters differences between TSUBAME2.0 and TSUBAME2.5. The memory capacity of each GPU is increased on TSUBAME2.5. But, it still cannot compute bigger domain. So, the optimization methods also are important to TSUBAME2.5.

In this thesis, we only use 1 GPU on each node. There are 3 GPUs in each node on TSUBAME2.5. It also needs to consider how to use those GPUs which is important to improve the performance and extend the memory capacity of GPUs. Using multiple GPUs on each node, it can get higher performance. But, how to efficiently use the GPUs of each node is also a challenge. It also needs to consider better way to use shared memory and registersof GPUs to improve the performance of kernels. In different stencil case, it needs to consider the better method that which part can be reused by shared memory or register. Also, as the size of shared memory or registersis limited, it needs to consider reusing minimum part to improve the performance. As the optimization methods are complicated to be implemented, it also needs to consider the program easiness. Our optimization methods combine multiple methods which include like temporal blocking, buffer-copy, memory-saving. All of them will increase the difficulty of programming. As the optimization methods are designed for specific stencil computations, it also needs to consider how to apply the optimization methods to wide range stencil computations. In different kinds of stencil, the consumed planes are different. Also, the dimension of domain becomes different. So, it needs to consider how to apply optimization method in different stencil case. All of those directions should be solved in future.

References

[1] NVIDIA CUDA C Programming Guide, Version 4.0, 2011

[2] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka, “Physis: An implicitly-parallel programming model for stencil computing on large-scale GPU-accelerated supercomputers” , IEEE SC11,2011.

[3] Gomez, C. , Gilabert, F.,Gomez, M.E.,Lopez, P. ,Duato, J. “RUFT: Simplifying the Fat-Tree Topology”, IEEE International Conference on Parallel and Distributed Systems, pp.153 – 160, 2008.

[4] Bogdanski B, Johnsen B.D., Reinemo S.-A. , Sem-Jacobsen F.O., “Discovery and Routing of Degraded Fat-Trees” , PDCAT, pp.697 – 702, 2012.

[5] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Akinori Yamanaka, Akira Nukada, Toshio Endo, Naoya Maruyama, and Satoshi Matsuoka, “Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer, ” In Proceedings of IEEE/ACM International Conference on Supercomputing (SC11), pp.1-11, 2011.

[6] Christen, M. , Schenk, O., Yifeng Cui, “PATUS for Convenient High-Performance Stencils: Evaluation in Earthquake Simulations”, In Proceedings of IEEE/ACM International Conference on Supercomputing (SC12), pp. 1-10, 2012.

[7] Schneible, J. ,Riha, L.,Malik, M., El-Ghazawi, T. , Schneible, J. “A method for communication efficient work distributions in stencil operation based applications on heterogeneous clusters” High Performance Computing and Simulation (HPCS), pp. 468 - 474 ,2012.

[8] Mei Yang ,Cheng Sun ,Zhimin Li,Dayong Cao,” An improved sparse matrix-vector multiplication kernel for solving modified equation in large scale power flow calculation on CUDA” Power Electronics and Motion Control Conference (IPEMC), pp. 2028 – 2031,2012.

[9] de Castro Martins, T., de Miranda Kian, J., Kubagawa Sato, A., de Sales Guerra Tsuzuki, M.” Matrix-vector multiplication and triangular linear solver using GPGPU for symmetric positive definite matrices derived from elliptic equations”, Soft Computing and Intelligent Systems (SCIS) and 13th International Symposium on Advanced Intelligent Systems (ISIS), pp. 1286 – 1291,2012.

[10] TSUBAME2.0 User’s Guide, version 1.1

[11] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey, “3.5-D blocking optimization for stencil computations on modern CPUs and GPUs,” IEEE SC10, 2010.

[12] Adam Dziekonski, Michal Mrozowski, “Tuning Matrix-Vector Multiplication on GPU”,

Information Technology (ICIT), pp.167-170, 2010.

[13] Nathan Bell, Michael Garland, “Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors”, High Performance Computing Networking, Storage and Analysis, pp.1-11, 2009.

[14] Zhuowei Wang Xianbin Xu WuqingZhao Yuping Zhang Shuibing He, “Optimizing Sparse Matrix-Vector Multiplication on CUDA”, Education Technology and Computer (ICETC), pp. V4-109 - V4-113, 2010.

[15] Wei Cao, Lu Yao, Zongzhe Li, Yongxian Wang, Zhenghua Wang, “Implementing Sparse Matrix-Vector Multiplication using CUDA based on a Hybrid Sparse Matrix Format”, 2010 International Conference on Computer Application and System Modeling (ICCASM 2010),V11.pp.161-165,2010.

[16] Noboru Tanabe, Yuuka Ogawa, Masami Takata and Kazuki Joe, “Scaleable Sparse Matrix-Vector Multiplication with Functional Memory and GPUs” , 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp.101-108,2011.

[17] Kiran Kumar Matam, Kishore Kothapalli, “Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU” , International Conference on Parallel Processing,pp.612-621,2011.

[18] Xiaowen Feng, Hai Jin, Ran Zheng, Kan Hu, Jingxiang Zeng, Zhiyuan Shao, “Optimization of Sparse Matrix-Vector Multiplication with Variant CSR on GPUs” , IEEE 17th International Conference on Parallel and Distributed Systems, pp.165-172,2011.

[19] Walid Abu-Sufah, Asma Abdel Karim, “An Effective Approach for Implementing Sparse Matrix-Vector Multiplication on Graphics Processing Units” , IEEE 14th International Conference on High Performance Computing and Communications, pp.453-460, 2012

[20] Xiangzheng Sun, Yunquan Zhang, Ting Wang, Xianyi Zhang, Liang Yuan, and Li Rao, “Optimizing SpMV for Diagonal Sparse Matrices on GPU” , International Conference on Parallel Processing, pp.492-501,2011

[21] Istv á n Reguly, Mike Giles, “Efficient sparse matrix-vector multiplication on cache-based GPUs”, Innovative Parallel Computing, pp.1-12, 2012.

[22] Gao Huan, Zhang Qian, “ A New Method of Sparse Matrix-Vector Multiplication on GPU” , Computer Science and Network Technology (ICCSNT), pp.954-958, 2012

[23] M. Frigo and V. Strumpen, “The memory behavior of cache oblivious stencil computations,” J. Supercomput., vol. 39, no. 2, pp. 93–112, 2007.

[24] J. Habich, T. Zeiser, G. Hager, and G. Wellein, “Enabling temporal blocking a lattice boltzmann flow solver through multicore-aware wavefront parallelization,” 21st International Conference on Parallel Computational Fluid Dynamics, pp. 178–

182, 2009.

[25] Leonardo Mattes, Leonardo C. Militelli, João Antonio Zuffo, "Platform to enforce multiple access control policy in grid hosting environment ", Proceedings of the third international conference on security and privacy in communication networks, pp.199-205, 2007.

[26] Leonardo Mattes and Sergio Kofuji, "Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids," ICMMT, pp.1536 – 1539, 2010.

[27] Leonardo Mattes and Sergio Kofuji, "The use of overlapping subgrids to accelerate the FDTD on GPU devices,"Radar Conference, pp. 807 –810, 2010.

[28] Wai Teng Tang, Wen Jun Tan, Krishnamoorthy, R., Yi Wen Wong, Shyh-hao Kuo, Goh, R.S.M. , Turner, S.J., Weng-Fai Wong, "Optimizing and Auto-Tuning Iterative Stencil Loops for GPUs with the In-Plane Method ", Parallel & Distributed Processing (IPDPS), pp. 452 – 462, 2013.

[29] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Ben Lipshitz, Oded Schwartz, and Omer Spillinger, " Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication", Proceedings of the IEEE International Parallel & Distributed Processing Symposium(IPDPS), 2013.

[30] Grey Ballard, Aydin Bulu, James Demmel, Laura Grigori, Ben Lipshitz Oded Schwartz, and Sivan Toledo,"Communication Optimal Parallel Multiplication of Sparse Random Matrices", Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures(SPAA), pp. 222 – 231, 2013.

[31] Toshio Endo and Satoshi Matsuoka, " Massive supercomputing coping with heterogeneity of modern accelerators," In Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008), 10pages, April 2008.

[32] 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, April 2010.

[33] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann and Holger Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," Computer Software and Applications Conference, vol.1, pp. 579 – 586, 2009.

[34] Tomoki Kawamura,Naoya Maruyama, and Satoshi Matsuoka, "Performance model for automatic optimization of communication in data-parallel stencil computations," IPSJ SIG Technical Report , vol.2012-HPC-135, 8pages, 2012

[35] Takeshi Minami , Takeshi Iwashita , Yasuhito Takahashi, and Hiroshi Nakashima, "Cache-aware performance improvement of FDTD kernel," IPSJ SIG Technical Report , vol.2010-HPC-124 No.5, 7pages, 2010.

[36] 金 光浩，遠藤 敏夫，松岡 聡．GPU メモリ容量を超える問題規模に対応する高性能ステンシル計算法 ．ハイパフォーマンスコンピューティングとアーキテクチャの評価に 関する北海道ワークショップ(HOKKE-20), 情報処理学会研究報告, 2012-ARC-194/HPC-137, 6 pages, 2012 年 12 月．

[37] 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,” AsHES/IPDPS 2013, 2013.

[38] Guanghao Jin, Toshio Endo, 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 2013, Indianapolis, September 2013.

Appendix A Other optimization methods

Here, we give some other optimization methods.

A.1 1D-TBMR method

So, we can get the pseudo code of 1D-TBMR method as below:

```
Allocate Grid on CPU to read initial of domain;
Allocate Grid0 and Grid1 on GPU to perform double buffering;
Allocate Buffer on GPU;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += TBT)
{
    For (j = 0; j < NSD; j += 1)
    {
        Copy sub-domain j from CPU to Grid0;
        For(k = 0; k<TBT; k+=1)
        {
            Read Cs XY-planes from Buffer;
            Compute 1 time step by one kernel with In-plane method;
            Save result to shifted position on Grid1;
            Save Cs XY-planes to Buffer;
            Swap Grid0 and Grid1;
        }
        Swap Grid0 and Grid1;
        Copy result of sub-domain j from Grid1 to CPU;
    }
}
```

The difference between 1D-2TBM and 1D-TBMR is the optimization method for the kernel. 1D-2TBM uses shared memory to reduce the access time to the global memory. 1D-TBMR method uses registersto reduce the access time to the global memory.

A.2 1D-P-TBMR method

As the domain grows, the sub-domain and ghost boundaries grow. As the GPU memory is limited, it contains smaller sub-domain or fewer ghost boundaries. Smaller sub-domain means more phases to compute and fewer ghost boundaries mean fewer time steps that can be computed locally on GPU. Both of them degrade the performance. We parallel the communication with the computation to solve this problem.

The pseudo code of 1D-P-TBM method is below:

```
Allocate Grid to read initial of domain;
Allocate Grid0 and Grid1 to perform double buffering;
Allocate Buffer on the GPU memory;
Allocate Buffer0 and Buffer1 on the GPU memory;
Separate domain into sub-domains;
For (i = 0; i< TSI; i += TBT) {
    For (j = 0; j < NSD; j += 1){
        Read initial from Buffer0 to Grid0;
        Parallel Do {
            Send next initial from CPU to Buffer0;
            Send former result from Buffer1 to CPU;
            For(k = 0; k<TBT; k+=1)
            {
                Read Cs XY-planes from Buffer to Grid0;
                Compute1 time step;
                Save result to shifted position on Grid1;
                Save Cs XY-planes from Grid0 to Buffer;
                Swap Grid0 and Grid1;
            }
        }
        Swap Grid0 and Grid1;
        Read result from Grid1 to Buffer1;
}}}
```

A.3 2D-1D-TBMS method

We can figure out the pseudo code as below:

```
Separate the domain into sub-domains;
Allocate Grid on each CPU to read initial of sub-domain;
Allocate Grid0 and Grid1 on each GPU to perform double buffering;
Allocate Buffer on each GPU;
Separate domain into parts;
For (i = 0; i< TSI; i += 1)
{
    For (j = 0; j < NPS; j += 1)
    {
        Copy part j from CPU to Grid0;
        For(k = 0; k<TBT; k+=2){
            Read Cs×2 XY-planes from Buffer;
            Compute boundary of two time step;
            Compress boundary;
            Save Cs×2 XY-planes to Buffer;
            Parallel Do
            {
                Compute inside of two time steps with memory-saving method;
                Exchange boundary;
            }
            Swap Grid0 and Grid1;
        }
        Swap Grid0 and Grid1;
        Copy result of sub-domain j from Grid 1 to CPU;
    }
}
```