Implementation and Evaluation of 3D Finite Element Method Application for CUDA

Satoshi Ohshima, Masae Hayashi, Takahiro Katagiri, and Kengo Nakajima

The University of Tokyo, 2-11-16 Yayoi, Bunkyo-ku, Tokyo, Japan
{ohshima,masae,katagiri,nakajima}@cc.u-tokyo.ac.jp

Abstract. This paper describes a fast implementation of a FEM application on a GPU. We implemented our own FEM application and succeeded in obtaining a performance improvement in two of our application components: Matrix Assembly and Sparse Matrix Solver. Moreover, we found that accelerating our Boundary Condition Setting component on the GPU and omitting CPU–GPU data transfer between Matrix Assembly and Sparse Matrix Solver slightly further reduces execution time. As a result, the execution time of the entire FEM application was shortened from 44.65 sec on only a CPU (Nehalem architecture, 4 cores, OpenMP) to 17.52 sec on a CPU with a GPU (TeslaC2050).

1 Introduction

The performance of GPUs is rapidly improving, attracting greater and greater attention. Through various projects, numerous numerical applications and libraries have been optimized for GPUs. On the other hand, there are many applications which are not suitable for GPUs, and some kinds of applications are suitable for both CPUs and GPUs. Therefore, it is necessary to implement and evaluate various real applications on GPUs.

Our specific interest is in investigating the acceleration of numerical applications though the use of GPUs and creating numerical libraries based on our results. Our current aim to accelerate our 3D finite element method (FEM) application on a NVIDIA GPU using CUDA[1]. We proposed and implemented some optimization techniques specific to our FEM application for a GPU and demonstrated that better performance can be attained than for a multi-core CPU.

The remainder of this paper is as follows. Section 2 describes the abstractions of CUDA and our target FEM application. Section 3 describes special characteristics of our FEM application and proposes three types of optimization techniques, which we implement and evaluate the effectiveness of. Section 4 is the Conclusion section.

2 CUDA and FEM Application

2.1 CUDA and NVIDIA GPU

CUDA is an architecture and application development framework of NVIDIA GPUs. It provides a GPGPU program development environment using an extension of the C/C++ programming language and a corresponding compiler. There is only a slight difference between C/C++ and CUDA in terms of language specification, but CUDA has unique specifications in its hardware model, memory model, and execution model. Therefore, the program optimization techniques for CPUs and GPUs (CUDA) are greatly different.

Some optimization techniques and strategies for GPU programming are already well known[2]. For example, GPU programs must have high parallelism in order to hide latencies of memory access. Also, reducing the number of branch instructions and random memory accesses is important because the associated performance penalties are larger for GPUs than for CPUs.

2.2 FEM Application

FEM is widely used in various scientific applications, and the acceleration of FEM is important and in high demand. There are many FEM applications and various libraries are used to accelerate these. Generally, Sparse Matrix Solver and Matrix Assembly take up a significant portion of the execution time of FEM applications.

Our target FEM application is based on the GeoFEM program[3], which is an existing FEM program for CPUs. The target problem is a 3D solid mechanics problem. This application has been parallelized for a multi-core CPU and PC cluster by using OpenMP and MPI. We use the modified OpenMP version [4] as a target of GPU acceleration.

Figure 1 shows the structure of our target FEM application, which has five parts. The execution time breakdown is shown in Figure 2. The execution environment is described in Table 1. This environment has a Xeon W3520 CPU (Nehalem architecture, 4 cores, 2.67 GHz) and a Tesla C2050 GPU (Fermi architecture, 448 SPs, 1.15 GHz). The number of elements is 512,000 (=$80\times80\times80$). The most time-consuming part is Sparse Matrix Solver, for both sequential execution (90.36%) and parallel execution (80.15%). In this part, this program uses a Conjugate Gradient Solver (CG solver) with a simple block diagonalization preconditioner as a sparse matrix solver. The second most time-consuming part is Matrix Assembly. Therefore, we focused mainly on trying to accelerate Sparse Matrix Solver and Matrix Assembly, which are described in Section 3.

Our FEM program has a special memory structure. Our matrix format is similar to the Compressed Row Storage (CRS) and blocked CRS formats. Matrices are partitioned into 3×3 blocks, and also divided into diagonal matrix, upper matrix, and lower matrix parts (Figure 3). This is based on physical problem setting that is stress strain. This 3×3 blocks structure is effective in terms of the memory requirements. Moreover, because 3×3 blocks structure improves cache hit rate, execution time is shorter than non-3×3 blocks structure.

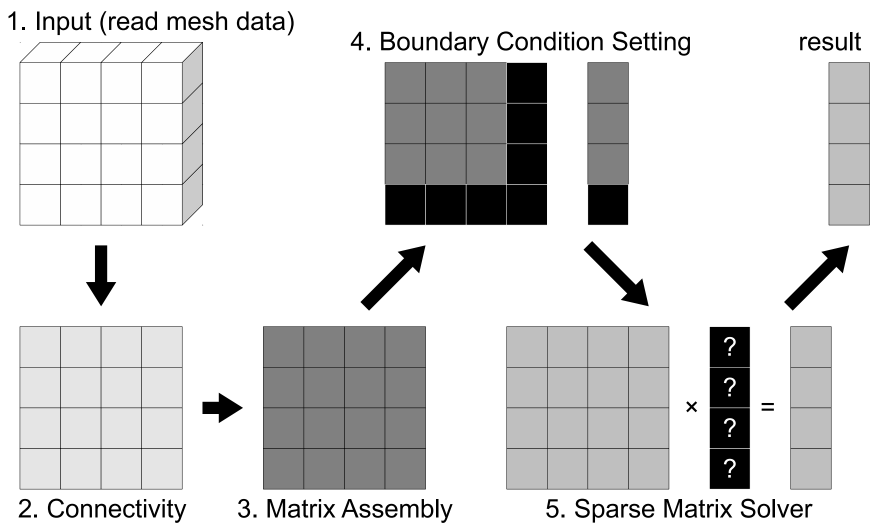


Fig. 1. Structure of our target FEM program

2.3 Related Works

Accelerating a CG solver requires speeding up matrix and vector calculations — for example, summation and multiplication of a vector and scalar, and multiplication of a matrix and vector — and these calculations are easy to parallelize. Also, these calculations are suitable for GPUs. Thus, there have been many studies of sparse matrix solvers (CG solvers) for GPUs over the years [5] [6] [7]. Also, some libraries for executing a CG solver on a GPU have been published. For example, the CUSP library[8] provides a fast CG solver and some useful data structures and calculation methods as a C++ class library.

One example of matrix assembly on a GPU is Cris[9].

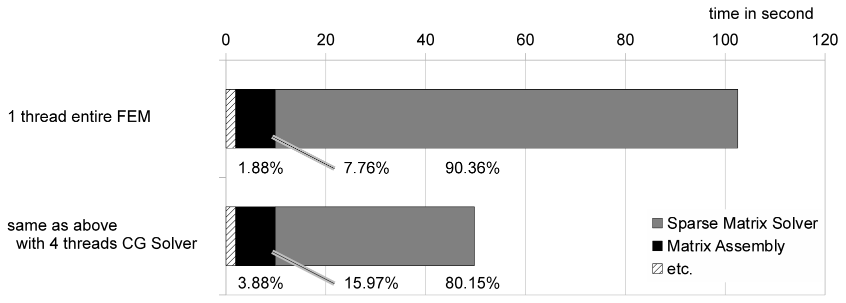


Fig. 2. Breakdown of execution time on a CPU

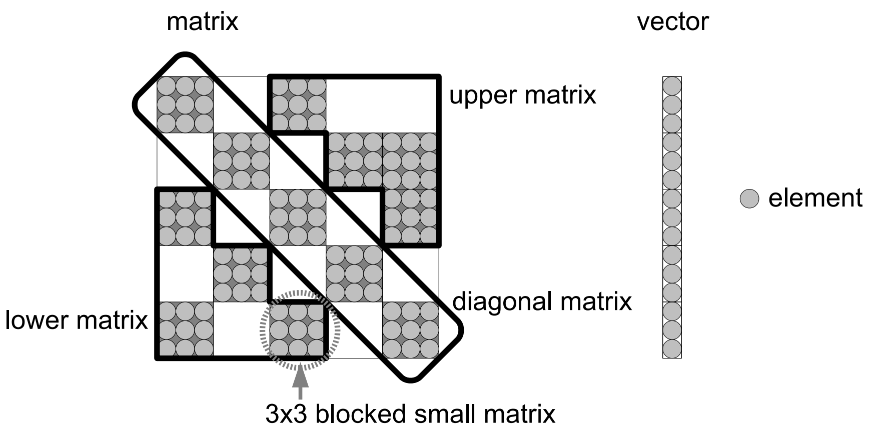


Fig. 3. Matrix structure

Table 1. Evaluation environment

CPU	Intel Xeon W3530 (Nehalem, 4cores, 2.67GHz)
Main memory	PC3-10600(DDR3-1333) 12GB
GPU	NVIDIA Tesla C2050 (Fermi, 448SPs, 1.15GHz)
Video memory	DDR5 3GB
connection	PCI-Express x16 (Gen 2)
OS	CentOS5.7 (kernel 2.6.18)
Compiler	gcc version 4.4.4 20100726 (Red Hat 4.4.4-13) (GCC) Cuda compilation tools, release 4.0, V0.2.1221

Our work has a specific target application and focuses on optimization techniques for that application. There are no existing reports of accelerating GeoFEM-based FEM applications on a GPU. We only are trying to accelerate a 3×3-block CG solver. Few studies have considered the implementation of boundary condition setting on a GPU or evaluating the CPU–GPU data transfer time between matrix assembly and sparse matrix solver components.

3 Implementation

3.1 Optimization of Sparse Matrix Solver

In this section, we describe three kinds of optimization strategies and implementations. The first is the optimization of Sparse Matrix Solver.

As described in Section 2, for our FEM application, Sparse Matrix Solver, which uses a CG solver with a matrix partitioned into 3×3 blocks, is the most time-consuming part. The CG solver involves only a few types of calculation, the most time-consuming of which is sparse matrix–vector multiplication (SpMV).

Therefore, in this section, we mainly focusing on describing acceleration on a GPU for a matrix partitioned into 3×3 blocks.

SpMV calculation with the CRS format is very easy to parallelize and accelerate, whether for CPU or GPU calculation, because the calculation of each row is independent and can be executed in parallel. However, it is difficult to obtain very good performance because SpMV calculation requires random memory accesses. A GPU can execute SpMV quickly and simply by dividing the matrix based on rows and assigning them to CUDA Threads. Our problem is how to divide and assign the 3×3 block partitioned matrix to CUDA Threads.

A simple strategy is dividing the block based on rows and assigning each block to CUDA ThreadBlock and each row to a CUDA Thread (Figure 4(a)). Although this strategy can obtain almost the same performance as SpMV calculation without 3×3 blocking on a GPU, it is not sufficiently optimized. This strategy cannot perform coalesced memory access and does not create sufficient parallelism.

Our optimization strategy is to assign each 3×3 block to three CUDA Threads. Each CUDA Thread reads matrix data in coalesced rule (Figure 4(b)) and has data in the shared memory, and these CUDA Threads calculate multiplication and addition using SharedMemory. Moreover, parallelism is increased by assigning multiple 3×3 blocks to each CUDA TreadBlock. These strategies are simple and effective.

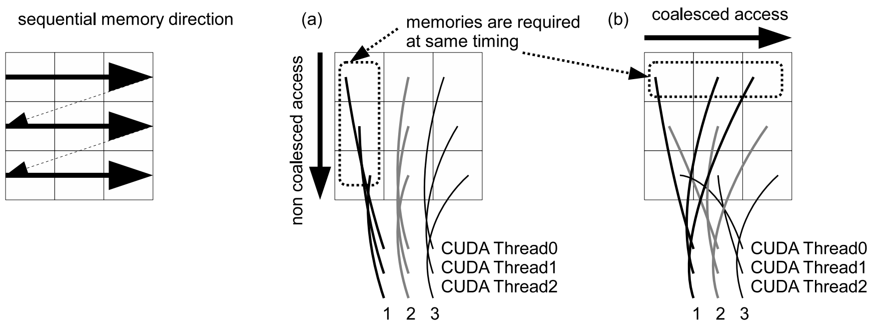


Fig. 4. Optimization strategy for SpMV (coalesced memory access)

Figure 5(a) shows the SpMV performance. The evaluation environment and problem setting are same as in Section 2. As a result, 3×3 blocked SpMV obtained 3.20 times better performance on a GPU than on a CPU. The performance ratio is much smaller than that for FLOPS (515 GFLOPS/42.7 GFLOPS = 12.06). Rather, it is closer to the memory bandwidth ratio (144 GB/s / 32 GB/s = 4.50). Moreover, by applying the same strategies to vector summation, multiplication, and dot product calculation, the execution time of the entire CG solver was shortened from 39.90 sec to 14.15 sec (Figure 5(b)).

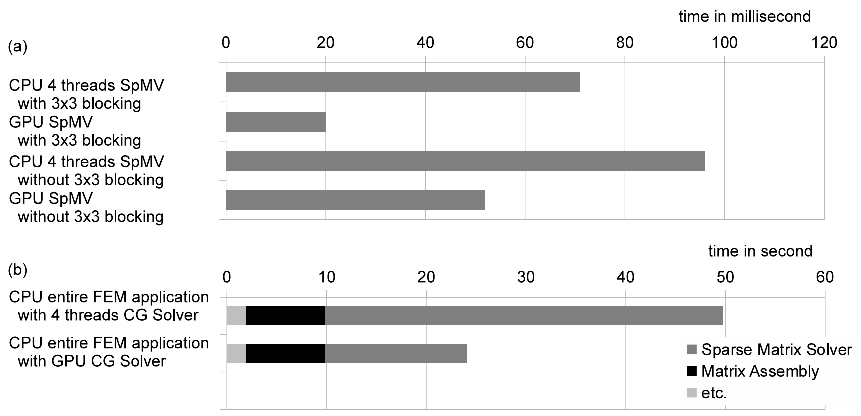


Fig. 5. Performance evaluation 1 (SpMV calculation and entire FEM application)

3.2 Optimization of Matrix Assembly

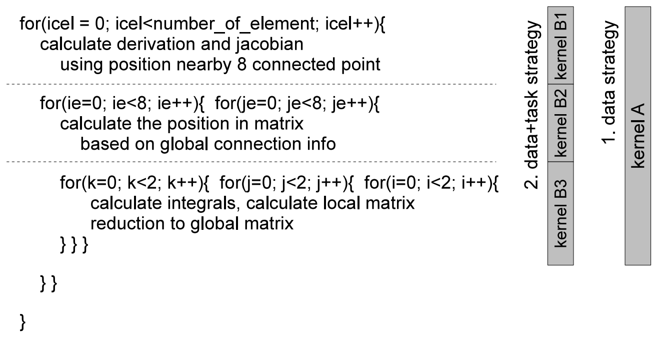
As shown in Section 2, Matrix Assembly was the second most time-consuming part of the FEM application. As the result of accelerating the CG solver on a GPU, the relative amount of time used by Matrix Assembly increased. In this subsection, we describe the acceleration of Matrix Assembly.

Figure 6 shows the structure (flow of source code) of Matrix Assembly. The flow of Matrix Assembly is more complicated than SpMV, and a hierarchical loop structure (loopA, loopB, and loopC in Figure 6) is characteristic. The problem of how to assign calculations to the GPU is more important than the problem of Sparse Matrix Solver. Because Matrix Assembly has dependencies between each matrix element, coloring computation is used to obtain parallelism. In this study, we make the CPU execute coloring computation (multicolor method) and make the GPU execute parallel calculation after coloring.

We tried to implement the calculation in two styles. The first style (data strategy) performs the entire calculation in one GPU kernel, and the second style (data+task strategy) divides the calculation into three parts. While the data+task strategy can make each GPU kernel simple and small, the CPU and GPU have to synchronize their kernels, which may degrade performance. According to the results of our implementation and evaluation, the data+task strategy obtained better performance than the data strategy (Figure 7).

3.3 Optimization of Entire FEM Application

The execution time of the FEM application is shortened by accelerating Matrix Assembly and Sparse Matrix Solver on a GPU. However, some parts of the FEM



```
for(icel = 0; icel<number_of_element; icel++){
  calculate derivation and jacobian
    using position nearby 8 connected point
--------------------------------------------------------------
  for(ie=0; ie<8; ie++){  for(je=0; je<8; je++){
    calculate the position in matrix
      based on global connection info
--------------------------------------------------------------
    for(k=0; k<2; k++){  for(j=0; j<2; j++){  for(i=0; i<2; i++){
       calculate integrals, calculate local matrix
       reduction to global matrix
    }}}
  }}
}
```

Fig. 6. Assignment of Matrix Assembly to the GPU

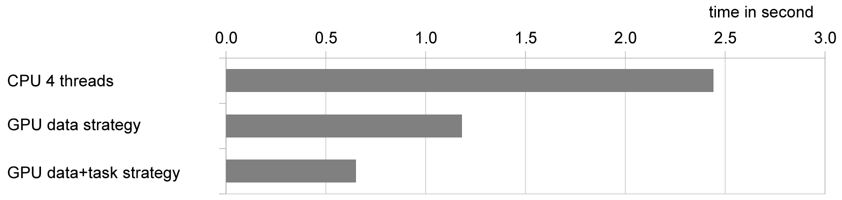


Fig. 7. Performance evaluation 2 (Matrix Assembly)

application are still executed on the CPU, but the execution times of these parts are not so time-consuming. However, we think that accelerating more parts on the GPU is important in order to obtain the best performance in the CPU with a GPU environment.

Of the five parts of the FEM application, Sparse Matrix Solver and the latter half of Matrix Assembly are already implemented to execute on the GPU. Here, we implement Boundary Condition Setting on the GPU. Because our FEM application has a simple boundary condition, the computation time for the boundary condition is small and we can implement it easily on the GPU. However, if we implement Boundary Condition Setting on the GPU, the data transfer between CPU and GPU after Matrix Assembly and before Sparse Matrix Solver can be omitted and the performance may improve. In order to obtain correct result, we modify Matrix Assembly and Sparse Matrix Solver to omit the data management computation. Therefore, the CPU only performs control computations, such as kernel launching of the GPU and loop control in the CG solver from after the coloring procedure of Matrix Assembly to the end of Sparse Matrix Solver (Figure 8).

Figure 9 shows the resulting execution times. The middle bar shows the results of the above-described optimization. It is true that the effect of this optimization is not large, but it is significant: a 5.14% performance improvement.

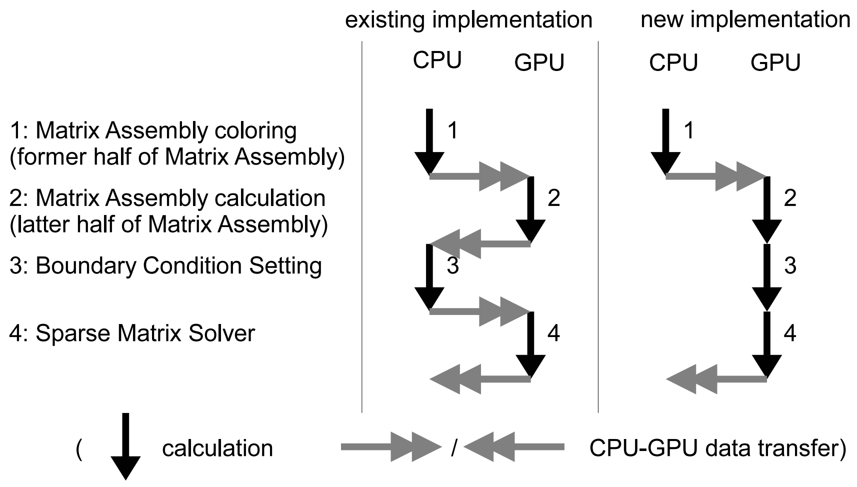

existing implementation
new implementation
CPU
GPU
CPU
GPU
1: Matrix Assembly coloring (former half of Matrix Assembly)
2: Matrix Assembly calculation (latter half of Matrix Assembly)
3: Boundary Condition Setting
4: Sparse Matrix Solver
1
2
3
4
(calculation / CPU-GPU data transfer)

Fig. 8. Assignment more parts to the GPU

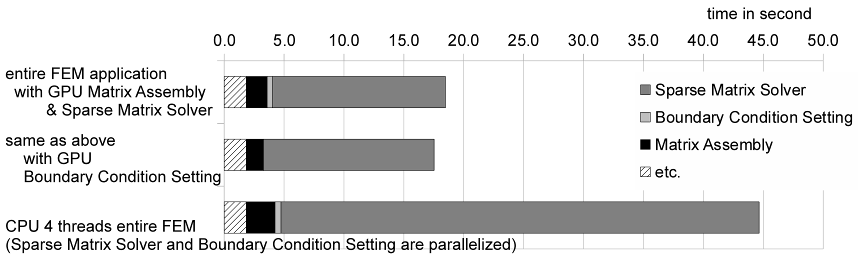


Fig. 9. Performance evaluation 3 (entire FEM application)

4 Conclusion

In this paper, we described the acceleration of a FEM application on a NVIDIA GPU with CUDA. We implemented three components of the application, Sparse Matrix Solver, Matrix Assembly, and Boundary Condition Setting, on a GPU. The execution time of Sparse Matrix Solver was shortened from 39.30 sec to 14.15 sec, and the execution time of Matrix Assembly was shortened from 2.44 sec to 0.65 sec. By implementing the Boundary Condition Setting on the GPU and omitting the CPU–GPU data transfer, the execution time of the entire FEM application was reduced. The most important technique for accelerating execution was memory assignment. Exact assignment obtained good performance. As a result, the execution time of entire FEM application was shortened from 44.65 sec on only a CPU (Nehalem architecture, 4 cores, OpenMP) to 17.52 sec on a CPU with a GPU (TeslaC2050).

There remains room for improvement and some challenges for FEM applications on a GPU. For example, coloring computation, complex preconditioners, and complex boundary conditions are difficult for a GPU to accelerate. Utilizing multiple GPUs is also an advanced topic. These remain as future work of our project.

Acknowledgment. This work is partially supported by “Framework and Programming for Post Petascale Computing (FP3C)” (JST, ANR) and “ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications with Automatic Tuning” (JST CREST).

References

1. NVIDIA: NVIDIA Developer Zone (CUDA ZONE), `http://developer.nvidia.com/category/zone/cuda-zone`
2. NVIDIA: NVIDIA CUDA C Programming Guide
3. Research Organization for Information Science & Technology (RIST): GeoFEM Homepage, `http://geofem.tokyo.rist.or.jp/`
4. Nakajima, K.: Parallel iterative solvers of geofem with selective blocking preconditioning for nonlinear contact problems on the earth simulator. In: ACM/IEEE Proceedings of SC 2003 (2003)
5. Bolz, J., Farmer, I., Grinspun, E., Scheróder, P.: Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In: Proceedings of ACM SIGGRAPH 2003, pp. 917–924 (2003)
6. Krüger, J., Westermann, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In: Proceedings of ACM SIGGRAPH 2003, pp. 908–916 (2003)
7. Cevahir, A., Nukada, A., Matsuoka, S.: High performance conjugate gradient solver on multi-gpu clusters using hypergraph partitioning. Computer Science - Research and Development 25, 83–91 (2010)
8. cusp-library: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, `http://code.google.com/p/cusp-library/`
9. Cecka, C., Lew, A.J., Darve, E.: Assembly of finite element methods on graphics processors. International Journal for Numerical Methods in Engineering 85(5) (2011)