Accelerating the Sweep3D for a Graphic Processor Unit

Chunye Gong*, Jie Liu*, Haitao Chen*, Jing Xie** and Zhenghu Gong*

Abstract—As a powerful and flexible processor, the Graphic Processing Unit (GPU) can offer a great faculty in solving many high-performance computing applications. Sweep3D, which simulates a single group time-independent discrete ordinates (Sn) neutron transport deterministically on 3D Cartesian geometry space, represents the key part of a real ASCI application. The wavefront process for parallel computation in Sweep3D limits the concurrent threads on the GPU. In this paper, we present multi-dimensional optimization methods for Sweep3D, which can be efficiently implemented on the fine-grained parallel architecture of the GPU. Our results show that the overall performance of Sweep3D on the CPU-GPU hybrid platform can be improved up to 4.38 times as compared to the CPU-based implementation.

Keywords—Sweep3D, Neutron Transport, GPU, CUDA

1. INTRODUCTION

When the first GPU was introduced in 1999, the GPU mainly had been used to transform, light and to rasterize triangles in three dimension (3D) graphics applications [1]. The performance of GPU doubles about every six to nine months, which means that it outperforms the Central Processing Unit (CPU) by a lot [2]. The modern GPUs are throughput-oriented parallel processors that can offer peak performance up to 2.72 Tflops single-precision floating-point and 544 Gflops double-precision floating-point [3]. At the same time, the GPU programming models, such as NVIDIA's Compute Unified Device Architecture (CUDA) [4], AMD/ATI's Streaming Computing [5] and OpenCL [6], have matures and they simplify the processing of developing non-graphics applications. The enhancement of computing performance, and the development of programming models and software makes GPU more and more suitable for general purpose computing. At present, GPU has been successfully applied to medical imaging, universe exploration, physics simulation, linear system solutions, and other computation intensive domains [7].

There is a growing need to accurately simulate physical systems whose evolutions depend on the transport of subatomic particles coupled with other complex physics [8]. In many simulations, particle transport calculations consume the majority of the computational resources. For example, the time devoted to particle transport problems in multi-physics simulations takes up

※ This research work is supported by the National Natural Science Foundation of China under grant No.60673150, also by the 973 Program of China under grant No. 61312701001. We would like to thank the anonymous reviewers for their helpful comments.

Manuscript received September 30, 2010; accepted February 22, 2011.

Corresponding Author: Chunye Gong

* Dept. of Computer Sciences, National University of Defense Technology, Changsha, P.R. China (gongchunye, @nudt.edu.cn, liujie@nudt.edu.cn, bugkiller@126.com, gzh@nudt.edu.cn)

** School of Information, XI'AN University of Finance and Economics, Xi'an, 710100, P.R. China (xaxj710@126.com)

Copyright ⓒ 2011 KIPS (ISSN 1976-913X)

50-80% of the total execution time for many of the realistic simulations on Department of Energy (DOE) systems [9, 10]. So parallelizing deterministic particle transport calculations is recognized as an important problem in many applications targeted by the DOE's Accelerated Strategic Computing Initiative (ASCI). The benchmark code, Sweep3D, [11] represents the heart of a real ASCI application that runs on the most powerful supercomputers such as Blue Gene [12] and Roadrunner [13]. Sweep3D is a challenging application for large-scale systems, in that it exhibits parallelism at different levels of granularity and the single-core efficiency is the basis [10].

Sweep3D has been ported to accelerators like CELL (processor name) and GPU. Sweep3D runs as a stand case on the first Pflops heterogeneous supercomputer Roadrunner [13]. Petrini et al [18] implemented Sweep3D on CELL (also called CBE, Cell Broadband Engine). They exploited 5 dimensions of parallelism, including process-level, thread-level, data streaming, and vector and pipeline parallelism, to achieve good performance. All the technologies make full use of CELL's processing elements, data traveling, and it's hierarchical memory and they achieve 4.5-20 times speedup compared with different kind of processors. Barker et al. [13] employed Sweep3D as a case study to exploit Roadrunner's hybrid architecture. Roadrunner is the first heterogeneous supercomputer to run Linpack at a sustained speed in excess of one Pflops. The most impressive thing in their work is that the ported Sweep3D performs on preproduction Roadrunner at full scale. Lubeck et al [21] implemented Sweep3D on the double precision performance improved CELL (PowerXCell8i) using an intra-chip message passing model that minimizes data movement. The advantages and disadvantages of this programming model with a previous implementation using a master-worker threading strategy are compared. A micro architecture performance model was applied to predict overall CPI (cycles per instruction), and gives a detailed breakdown of processor stalls. We [22] presented multidimensional optimization methods for Sweep3D, which can be implemented on the fine-grained parallel architecture of the GPU. The multi-dimensional optimization methods include thread level parallelism, more threads and repeated computing, and using on-chip shared memory, etc.

Because the representation of Sweep3D and the potential computing performance of the new GPU platform for particle transport, we accelerated Sweep3D on GPU. This is an extended paper on the previous implementation of Sweep3D [22]. In this paper we describe our experiences of developing Sweep3D implementation for the CUDA platform, and we analyze the bottleneck of our GPU execution. Our GPU version is based on the Single Instruction Multiple Data (SIMD) and uses the massive thread level parallelism of GPU. Efficiently using registers and thread level parallelism can improve performance. We use the repeated computing and shared memory to schedule 64 times more threads, which improves performance with a 64n-cubed problem size. Our GPU version achieves 4.38 times speedup as compared to the original single CPU core version.

The remainder of this paper is organized as follows: in the next section, the overview of Sweep3D is presented. The architecture of GT200 GPU and programming model CUDA are described in Section 3. In Section 4 we present the multi-dimensional optimization methods. In Section 5 we present the experimental results and analysis. Finally we give the conclusion and plans for the future in Section 6.

2. An Overview of Sweep3D

Sweep3D [11] solves a three-dimensional neutron transport problem from a scattering source.

The basis of neutron transport simulation is the time-independent, multi-group, inhomogeneous Boltzmann transport equation. The numerical solution to the transport equation involves the discrete ordinates (S_n) method and the procedure of source iteration. In the S_n method, where N represents the number of angular ordinates used, the angular-direction is discretized into a set of quadrature points. In Cartesian geometries (XYZ), each octant of angles has a different sweep direction through the mesh, and all angles in a given octant sweep the same way. The sweep of the S_n method generically is named, “wavefront” [14]. The solution involves two steps: the streaming operator is solved by sweeps and the scattering operator is solved iteratively.

A S_n sweep for a given angle proceeds as follows: every grid cell has 4 equations with 7 unknowns (6 faces plus 1 central) and boundary conditions complete the system of equations. The solution is by a direct ordered solve known as a, “sweep.” Three known inflows allow the cell center and 3 outflows to be solved. Each cell’s solution then provides inflows to 3 adjoining cells (I, J, and K directions). This represents a wavefront evaluation with recursion dependence in all 3 grid directions. Sweep3D exploits parallelism via a wavefront process. First, a 2D spatial domain decomposition onto a 2D array of processors in the I and J-directions is used. Second, the sweeps of the next octant pair start before the previous wavefront is completed; the octant order required for reflective boundary conditions limits this overlap to two octant pairs at a time. The overall combination is sufficient to give good theoretical parallel utilization. The resulting diagonal wavefront is depicted in Fig. 1 just as the wavefront starts in the 4th octant and heads toward the 6th octant [14, 15].

On single core, there is no communication in Sweep3D. The Sweep() subroutine, computational core of Sweep3D, takes about 97.70% of the whole runtime. The structure of the Sweep() subroutine is listed in Fig. 2. The jkm loop (line 7 - line 22) in the Sweep() subroutine takes 99.05 % of the subroutine runtime. The major computation of each cell includes reading the source from spherical harmonics (P_n) moments, solving S_n equation recursively, updating flux from P_n moments, and updating diffusion synthetic acceleration (DSA) face currents.

A complete wavefront from a corner to its opposite corner is an iteration. The I-lines that is the jkm loop in the iteration can be solved in parallel on each diagonal. As showed in Fig. 1, the number of concurrent threads is 1, 3, 6 and 10. The relationship between maximum concurrent threads ($MCT(n)$) and problem size n-cubed ($n \in z^+$) shows in (1):

$$\lim_{n \to \infty} MCT(n) = 6n \tag{1}$$

Similarly, the relationship between average concurrent threads ($ACT(n)$) and problem size n-cubed shows in (2):

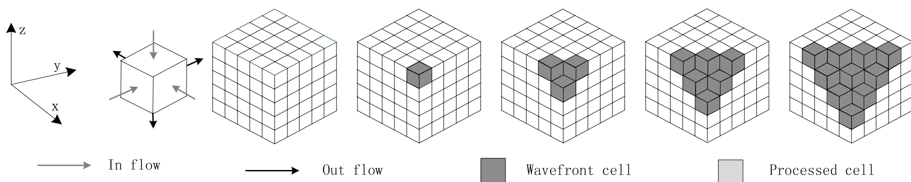

Fig. 1. Wavefront parallelism in 3D geometry

```
1   DO iq=1,8                          ! octants
2     DO mo=1,mmo                      ! angle pipelining loop
3       DO kk=1,kb                     ! k-plane pipelining loop
4         RECV E/W                     ! recv block I-inflows
5         RECV N/S                     ! recv block J-inflows
6         DO idiag=1,jt+nk-1+mmi-1     ! JK-diagonals with MMI pipelining
7           DO jkm=1,ndiag             ! I-lines on this diagonal
8             DO i=1,it                ! source (from Pn moments)
9             ENDDO
10            IF .NOT.do_fixups
11              DO i=i0,i1,i2          ! Sn eqn
12              ENDDO
13            ELSE
14              DO i=i0,i1,i2          ! Sn eqn fixups
15              ENDDO
16            ENDIF
17            DO i=1,it                ! flux (Pn moments)
18            ENDDO
19            DO i=1,it                ! DSA face currents
20            ENDDO
21          ENDDO
22        ENDDO
23        SEND E/W                     ! send block I-outflows
24        SEND N/S                     ! send block J-outflows
25      ENDDO
26    ENDDO
27  ENDDO
```

Fig. 2. Structure of Sweep() subroutine

$$\lim_{n \to \infty} ACT(n) = 3n \tag{2}$$

3. Architecture of Nvidia GT200 and CUDA

3.1 Architecture of Nvidia GT200

The architecture of GPU is optimized for rendering real-time graphics, a computation and memory access intensive problem domain with enormous inherent parallelism. Unlike CPU, a much larger portion of a GPU's resources are devoted to data processing rather than to caching or controlling flow.

The NVIDIA GT200 chip (Fig. 3) contains 240 Streaming-Processor (SP) cores running at

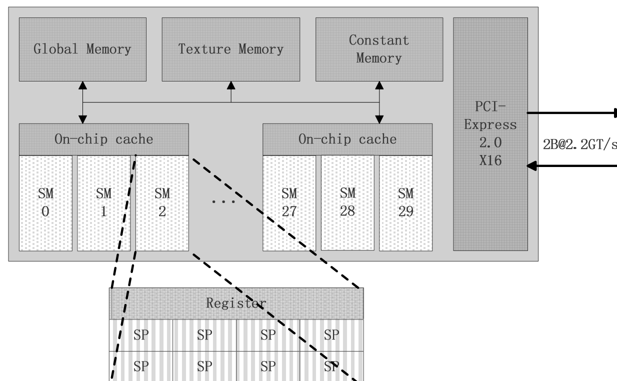

Fig. 3. Architecture of GT200 GPU

Table 1. Technical specifications of a Tesla S1070 and M2050 GPU

GPU model	S1070	M2050
CUDA compute capability	1.3	2.0
Total amount of global memory	4G	2.8G
Number of multiprocessors (SM)	30	14
Number of CUDA cores (SP)	240	448
Total amount of constant memory	64 KB	64 KB
Total amount of shared memory per block	16 KB	48 KB
Total number of registers available per block	16 KB	32 KB
Warp size	32	32
Clock rate	1.44 GHz	1.15 GHz

1.44 GHz. Eight SPs form one Streaming Multiprocessors (SMs or Multiprocessors) and each SP run in a SIMD manner. There are ten independent processing units called, "Thread Processing Clusters" (TPC) and each TPC contains a geometry controller, a SM controller, three SMs, and a texture unit. Multiprocessors creates, manages, and executes concurrent threads in hardware with near zero scheduling overheads and can implement barrier synchronization. The Single Instruction Multiple Thread (SIMT) unit, which is akin to SIMD vector organizations, creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps [4, 19].

The particular and useful specifications of Tesla S1070 and M2050 are listed in Table 1. The ideal peak performance depends on how many operations can be performed per cycle. One stream processor technically supports one MAD (multiply-add) and one MUL (multiply) per cycle, which would correspond to 1.04 Tflops per GPU. There is only one double-precision unit in a SM and the peak double-precision floating-point performance is about 78 Gflops per GPU. There are 4 GPUs in a Tesla S1070. So the peak single- and double-precision floating-point performances are 3.73 to 4.14 Tflops and 311 to 345 Gflops [19].

The application performance on the GPU is directly associated with Maximum Thread Blocks (MTB) per Multiprocessor. Most compute capability 1.3 run eight thread blocks (MTB_{warp}) because of the restriction of warp. As listed in Table 1, the shared memory and registers also limits maximum thread blocks. For example, the maximum thread blocks limited by registers (MTB_{reg}) is shown in (3):

$$MTB_{reg} = \frac{\text{Total number of registers available per block}}{\text{(Threads per block)*(Registers per thread)}} \tag{3}$$

3.2 Programming Model and Software

The programming model is a bridge between hardware and application. As a scalable parallel programming model, CUDA [4] does not abstract the whole computing system in an ideal level. The hybrid system is separated into host and device. CUDA uses the kernel function, which is a SPMD (Single Program Multiple Data) computation with a potentially large number of parallel threads, to run efficiently on hardware. The concept of thread block in thread hierarchy makes the CUDA programming model independent of the number of a GPU's SMs.

CUDA exposes the Multiprocessors on the GPU for general purpose computation through a minimal set of extensions to C programming language. Compute intensive components of a program can run as kernel function. Kernel functions are executed many times with different input data. The software managed on chip cache or shared memory in each SIMD core and barrier synchronization mechanism let local data sharing and synchronization in a thread block become a reality.

4. Multi-Dimensional Optimization Methods

Although there is no data send and receive on a single GPU through MPI, we still keep the process level parallelism as Fig. 2 shows. This will guarantee the portability of the existing software. As mentioned in Section 2, the jkm loop takes the most runtime. So the strategy for the implementation on this hybrid system is GPU-centric. That is to say, we have the compute intensive part run on GPU and CPU do little computation. The architecture optimization of hybrid computing of this application can be seen in Fig. 4.

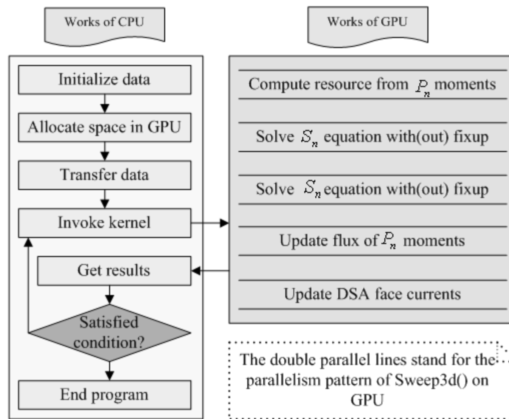

Fig. 4. Architecture of hybrid CPU-GPU computing

4.1 Stage 1: Thread Level Parallelization

Fig. 2 indicates that there are two levels of parallelism of the subroutine Sweep(). One is the I-lines on JK-diagonals with MMI pipelining (the jkm loop, line 7 - line 20) that can be processed in parallel, without any data dependency. The other one is the inner loop in the jkm loop, including reading the source from P_n moments (line 8, 9), updating the flux of P_n moments (line 17, 18) and DSA face currents (line 19, 20). There are two reasons for why we don't choose exploiting the parallelism of the most inner loop. First, the inner loop is limited by the X dimension size of the space geometry (the value of 'it'). Another shortcoming of parallelizing the most inner loop is that CPU calls kernel functions a lot more times and keeps too much temp local data. So, exploiting the parallelism of the jkm loop becomes the suitable choice.

In the jkm loop, there are two main branches: .NOT.do_fixups or do_fixups (line 10) and the variable of data-dependent loops increases or decreases (line 11, 14). The branches don't affect

Table 2. Runtime on CPU and GPU (seconds)

Size (cubed)	64	128	192	256
CPU	2.43	28.72	107.85	254.33
Stage1	4.44	41.51	79.79	426.20

the performance of the kernel's execution, but they make the whole kernel too big to program and debug on GPU. So we divide the jkm loop into four different kernels (jkmKernelNotFixUp, jkmKernelNotFixDown, jkmKernelFixUp, jkmKernelFixDown) and let CPU deal with the branches. Each time CPU invokes "ndiag" threads, which are divided into 1 to $\lceil \text{ndiag} \cdot 1.0/\text{threadBlockSize} \rceil$ thread blocks. The structure of every kernel is the same as line 8 - line 20 except for two branches.

The runtime of 64 times-cubed problem size on both CPU and GPU are listed in Table 2. When the problem size is 128-cubed, the runtime of the Sweep3D on GPU is 41.51 seconds, which is slower than running on CPU. The following parts in Section 4 illustrate how to accelerate Sweep3D on Tesla S1070.

4.2 Stage 2: More Threads and Repeated Computing

Although we achieve 1.35 times speedup on problem size 192-cubed, the performance of 128-cubed and 256-cubed is relatively poor. Taking 192-cubed as an example, the average number of concurrent threads $ACT(192)$ is about 576. The jkmKernelNotFixUp and jkmKernelNotFixDown use 41 registers per thread while jkmKernelFixUp and jkmKernelFixDown use 52 registers per thread. According to (3), we have $\lfloor 16\text{KB}/(64 \cdot 41) \rfloor = 6$ and $\lfloor 16\text{KB}/(64 \cdot 52) \rfloor = 4$. That is to say, six or four blocks can concurrently run on every multiprocessor. However, unfortunately, there are only 576 threads and 9 total blocks that cannot make full use of 30 multiprocessors in 256-cubed problem size, let alone a smaller problem size.

There is no data-dependence in the loops from the reading source from P_n moments, updating the flux and DSA face currents. We use 64 times more threads to do the same work that one thread does in stage 1. As mentioned in Section 2, the average number of concurrent threads in n-cubed is about 3n in stage 1. Here, invocating a kernel has 3n thread blocks with 64 threads in

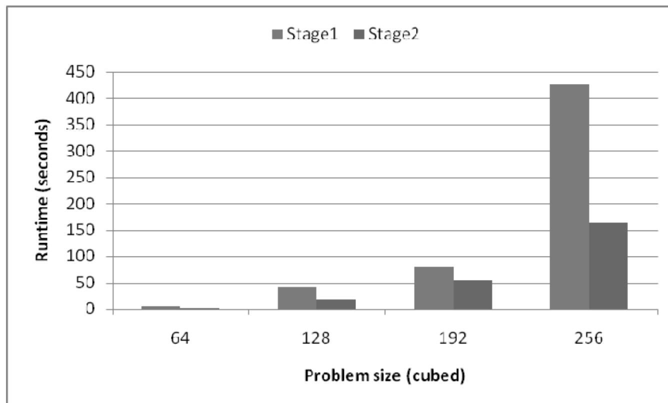

Fig. 5. Performance improvement using more threads and repeated computing on S1070

each block.

It's efficient to use the global memory bandwidth when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction of 128 bytes. However, the data-dependent S_n loop must be executed by one thread, so there are two barrier synchronizations before and after the S_n loop. The performance improvement is depicted in Fig. 5.

4.3 Stage 3: Using Shared Memory

The shared memory is on-chip and software managed cache can reduce the memory access time of the reusing data. However, most computations in the program lack data reuse. It is important to exploit data reuse. We found that the results of the computing source from P_n moments are reused in the S_n loop and medial results in the S_n loop are reused in the updating flux of P_n moments and DSA face currents. We utilize the shared memory to store the corresponding reusable vectors instead of accessing the global memory.

A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp. When accessing local or global memory, there is an additional 400 to 600 clock cycles of memory latency [4]. A multiprocessor takes 6 cycles per warp instead of the usual 4 to access the shared memory [20]. There are three reasons why using the shared memory cannot run up to 100 times faster than using global memory. First, because of the double data type, there exists two ways of bank conflicts in shared memory [4]. Second, the global memory access is coalesced and thread blocks scheduling can hide latency. Third, the real memory access operations are more complicated than theory analysis. The performance improvement is depicted in Fig. 6.

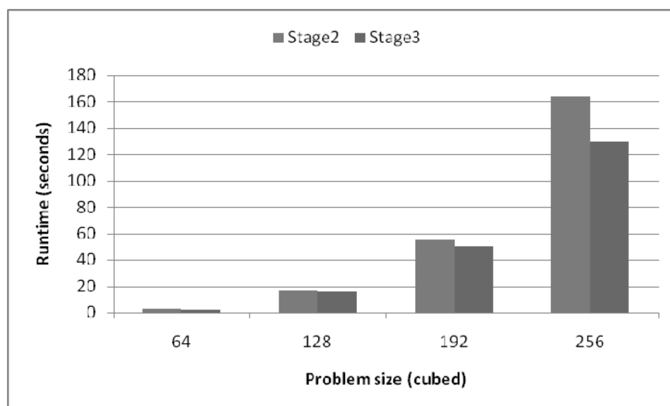

Fig. 6. Performance improvement by using shared memory on S1070

4.4 Stage 4: Other Methods

Other methods include more work on GPU, communication overlapping computation, autotunner in the running time, and the using of texture memory. More work on GPU can avoid data movement between CPU and GPU. Communication overlapping computation can hide some time that is spent on communication. Autotunner is valuable when ndiag is very small. Texture memory has a high-speed cache that can reduce the memory access time of the neighboring data. All the methods above reduce the runtime of 256-cubed by 5.69 seconds.

5. PERFORMANCE RESULTS AND ANALYSIS

In this section, we compare the runtime of our GPU and CPU implementations from a wide range of problem sizes and present speedup and precision errors.

The platform consists of Intel(R) Core(TM)2 Quad CPU Q6600 2.40 GHz processors, 5 GB of main memory, and the Red Hat Enterprise Linux Server release 5.2 operating system. The Tesla S1070 high performance computing platform consists of 4 GTX200 GPU, clock rate 1.44 GHz, with 4 GB frame buffer memory each, making a total GPU memory of 16GB.

For the purpose of comparison, we measure the speedup provided by one GTX200 GPU and compare it to a serial code running on the CPU (consequently only one core). Both codes run on double-precision floating-point arithmetic, and are compiled using the GNU gfortran compiler version 4.1.2, and the nvcc compiler provided by CUDA version 3.1. In both cases, level two optimization has been performed. Sweep3D runs 4 iterations, half with flux fix-ups, and half without.

The performance improvement of each optimization stage on S1070 is illustrated in Fig. 7. Stage 1 ports kernel part of Sweep3D to GPU and uses thread level parallelism. Stage 2 does some repeated computing and uses 64 times more threads to access global memory. Stage 3 uses shared memory to store local array. Stage 4 mainly puts additional work on GPU. Sweep3D runs 2.25 times faster on S1070 than on the single core of Q6600. The performance comparisons between different experimental platforms are shown in Fig. 8. The new GPU M2050 performs 2 times better than the S1070. And the peak performance improvement of Sweep3D on M2050 than on the single core of Q6600 is up to 4.38.

There are mainly four procedures in each cell's computation. The four procedures are the reading source from P_n moments named "source"; solving the S_n equation recursively, which is named "S_n"; updating the flux from P_n moments named, "flux"; and updating DSA face currents named, "face". Except for these four procedures, the additional computation and control are named "other" procedure. The percentage of each procedure in the whole runtime is shown in Fig. 9. Their quotient on GPU and GPU are absolutely different. The flux holds up to 32.29% of the whole runtime on Q6600; and the S_n procedure holds 83.24% to 86.68% of the whole run-time. It is obvious that the S_n procedure is the bottleneck for Sweep3D on the GPU.

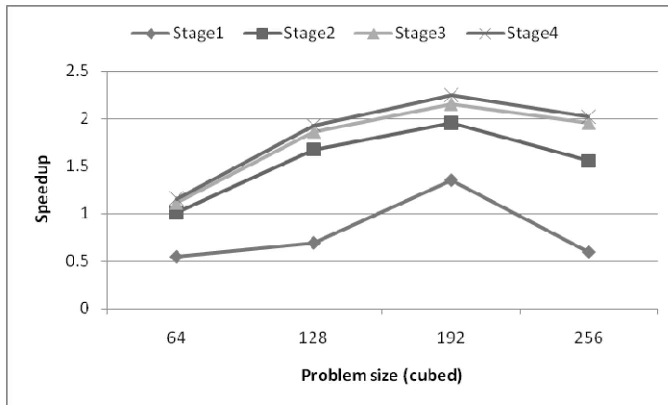

Fig. 7. Speedup of whole Sweep3D application on S1070

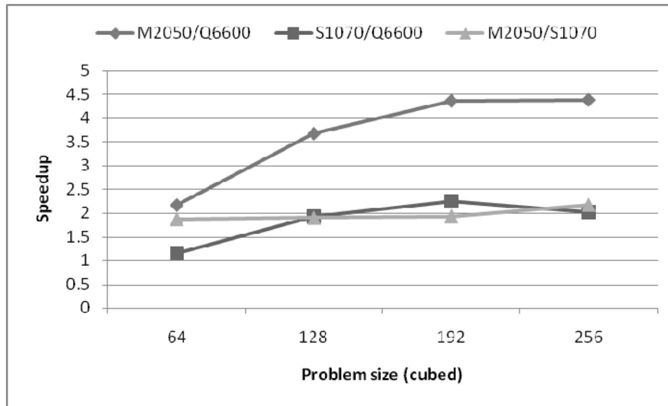

Fig. 8. Performance comparisons between different experimental platforms

Some double precision mathematical operations on GTX200 are IEEE-754 round-to-nearest-even, or even worse. This kind of system error cannot be avoided. The maximum error of 4 iterations at problem size 128-cubed is smaller than 10-13, which is acceptable.

Petrini got that the CELL BE is approximately 4.5 and 5.5 times faster than the 1.9 GHz Power5 and 2.6 GHz AMD Opteron on 50-cubed problem size. The optimized runtime is 1.33 seconds on CELL BE without knowing how many iterations there are. Consequently the runtime of Power5 and Opteron are 5.985 seconds and 7.315 seconds. The runtime of Intel Q6600 2.40 GHz is 3.10 seconds with 12 iterations, which is the default on 50-cubed input size. So the CELL BE is about 2.33 times faster than Intel Q6600 processors. If the proportion can scale to a 192-cubed input size then the Sweep3D on M2050 is 1.88 times faster than on CELL BE. However, Lubeck's implementation on PowerXCell8i (a double precision improved CELL) is much faster than Petrini's implementation [21]. The runtime of 10 iterations with fixups and dsa off for cubed-50 problem size is 0.19 seconds. The according runtime on Q6600 is 2.31 seconds. The speedup of Swee3D on PowerXCell8i is up to 12.16. Subsequently, the CELL performs much better than GPU at present implementation. So, the accelerated GPU implementation needs further optimization.

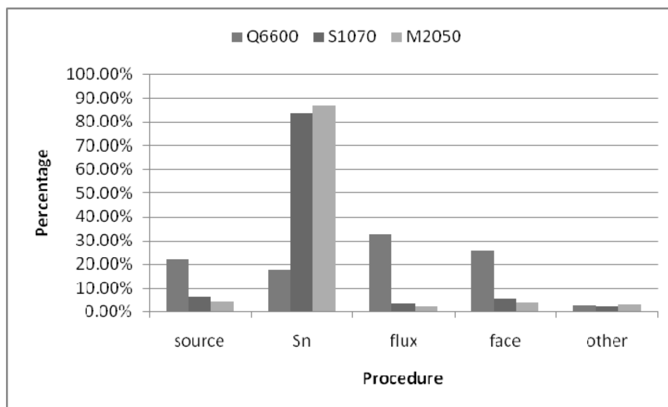

Fig. 9. Percentage of each procedure in the whole runtime

6. Conclusion

An optimized GPU based implementation of Sweep3D has been presented and up to 4.38 times speedup has been achieved compared to its CPU implementation. The implementation efficiently uses the features of a hybrid system and explores its multi-dimension optimizations. To the best of our knowledge, our work has revealed a new implementation of neutron transport simulation problems and wavefront processes of parallelism on GPU. Other similar and complex wavefront algorithms or applications are also likely to benefit from our experience with the CPU-GPU hybrid system.

After careful comparison, the bottleneck for Sweep3D on the GPU is distinguished. The bottleneck consumes more the 80% of the total runtime. Obviously, breaking this bottleneck must be a part of the future work. Furthermore, the scalability and performance issues on heterogeneous CPU-GPU clusters will be investigated also.

References

[1] H. Nguyen, "GPU Gems 3," Addison Wesley, 2007.

[2] D. Kirk, "*Innovation in graphics technology," Talk in Canadian Undergraduate Technology Conference*, 2004.

[3] AMD Corporation, "ATI Radeon HD 5870 Feature Summary," http://www.amd.com/, 2010.

[4] NVIDIA Corporation, "CUDA Programming Guide Version 3.1," 2010.

[5] AMD Corporation, "ATI Stream Computing User Guide Version 2.0," 2010.

[6] A. Munshi, "The OpenCL Specification Version: 1.1," Khronos OpenCL Working Group, 2010.

[7] NVIDIA Corporation, "Vertical solutions on CUDA," http://www.nvidia.com/object/vertical solutions.html, 2010.

[8] M.M. Mathis, N. Amato, M. Adams, W. Zhao, "*A General Performance Model for Parallel Sweeps on Orthogonal Grids for Particle Transport Calculations," Proc. ACM Int. Conf.* Supercomputing, 2000, pp.255-263.

[9] A. Hoisie, O. Lubeck, H. Wasserman, *"Scalability analysis of multidimensional wavefront algorithms on large-scale SMP clusters," The 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999, pp.4-15.

[10] A. Hoisie, O. Lubeck, H. Wasserman, "Performance and scalability analysis of teraflop- scale parallel architectures using multidimensional wavefront applications," *International Journal of High Performance Computing Applications*, Vol.14, No.4, 2000, pp.330-346.

[11] The Los Alamos National Laboratory, "Sweep3D," http://wwwc3.lanl.gov/pal/software/sweep3d/, 2010.

[12] K. Davis, A. Hoisie, G. Johnson, D.J. Kerbyson, M. Lang, M. Pakin, F. Petrini, "*A Performance and Scalability Analysis of the BlueGene/L Architecture," Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004, pp.41-50.

[13] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, J.C. Sancho, "*Entering the petaflop era: the architecture and performance of Roadrunner," Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp.1-11.

[14] E.E. Lewis, W.F. Miller, "Computational Methods of Neutron Transport," American Nuclear Society, LaGrange Park, 1993.

[15] K. Koch, R. Baker, R. Alcouffe, "Solution of the First-Order Form of Three-Dimensional Discrete Ordinates Equations on a Massively Parallel Machine," *Transactions of American Nuclear Society*, V65, 1992, pp.198-199.

[16] M.M Mathis, D.J. Kerbyson, "A General Performance Model of structured and Unstructured Mesh Particle Transport Computations," *Journal of Supercomputing*, Vol.34, 2005, pp.181-199.

[17] D.J. Kerbyson, A. Hoisie, "Analysis of Wavefront Algorithms on Large-scale Two level Heterogene-

ous Processing Systems," *Workshop on Unique Chips and Systems*, 2006, pp.259-279.

[18] F. Petrini, G. Fossum, J. Fernandez, A.L. Varbanescu, N. Kistler, M. Perrone, "*Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine," The 21st International Parallel and Distributed Processing Symposium*, 2007.

[19] NVIDIA Corporation, "NVIDIA Tesla S1070 1U Computing System," http://www.nvidia.com/object/product tesla s1070 us.html, 2010.

[20] V.Volkov, J.W. Demmel, *"Benchmarking GPUs to tune dense linear algebra," Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[21] O. Lubeck, M. Lang, R. Srinivasan, G. Johnson, "Implementation and performance modeling of deterministic particle transport (Sweep3D) on the IBM Cell/BE," *Scientific Programming*, Vol.17, No.1, 2009.

[22] C. Gong, J. Liu, Z. Gong, J. Qin, J. Xie, "Optimizing Sweep3D for Graphic Processor Unit," C.-H. Hsu, L. Yang, J. Park, S.-S. Yeo (Eds.), Algorithms and Architectures for Parallel Processing, Vol.6081 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2010, pp.416-426.

Chunye Gong

Chunye Gong received his BS degree in Applied Mathematics and his MS degree in Computer Science from the National University of Defense Technology (NUDT). He is now a PhD candidate in Computer Science at NUDT. He focuses on: large scale scientific computing, heterogeneous computing on hybrid architecture, non-linear numerical simulation, and stochastic simulation.

Jie Liu

Jie Liu received his doctorate degree in Computer Science at NUDT. He is now an associate professor at NUDT. He focuses on: large scale scientific computing.

Haitao Chen

Haito Chen received his doctorate degree in Computer Science at NUDT. He is now an assistant professor at NUDT. He focuses on: numerical simulation and system software.

Jin Xie

Jin Xie is an assistant professor at XI'AN University of Finance and Economics and PhD candidate in Computer Science at NUDT. He focuses on: distributed computing and network security.

Zhenghu Gong

Zhenghu Gong is a respected professor at NUDT. Professor Gong focuses on: pervasive computing, high performance networks, and information security.