Accelerating Dock6's Amber Scoring with Graphic Processing Unit

Hailong Yang, Bo Li, Yongjian Wang, Zhongzhi Luan, Depei Qian, and Tianshu Chu

Department of Computer Science and Engineering,
Sino-German Joint Software Institute, Beihang University,
100191 Beijing, China
{hailong.yang,yongjian.wang,tianshu.chu}@jsi.buaa.edu.cn,
{libo,zhongzhil,depeiq}@buaa.edu.cn

Abstract. In the drug discovery field, solving the problem of virtual screening is a long term-goal. The scoring functionality which evaluates the fitness of the docking result is one of the major challenges in virtual screening. In general, scoring functionality in docking requires large amount of floating-point calculations and usually takes several weeks or even months to be finished. This time-consuming disadvantage is unacceptable especially when highly fatal and infectious virus arises such as SARS and H1N1. This paper presents how to leverage the computational power of GPU to accelerate Dock6 [1]'s Amber [2] scoring with NVIDIA CUDA [3] platform. We also discuss many factors that will greatly influence the performance after porting the Amber scoring to GPU, including thread management, data transfer and divergence hidden. Our GPU implementation shows a 6.5x speedup with respect to the original version running on AMD dual-core CPU for the same problem size.

1 Introduction

Identifying the interactions between molecules is critical both to understanding the structure of the proteins and to discovering new drugs. Small molecules or segments of proteins whose structures are already known and stored in database are called ligands. While macromolecules or proteins associated with a disease are called receptors [4]. The final goal is to find out whether the given ligand and receptor can form a favorable complex and how appropriate the complex is, which may inhibit a disease's function and thus act as a drug.

Scoring is the step after docking which is involved evaluating the fitness for docked molecules and ranking them. A set of sphere-atom pairs will be on behalf of an orientation in receptor and evaluated with a scoring function on three dimensional grids. At each grid point, interaction values are to be summed to form a final score. These processes need to be repeated for all possible translations and rotations. Tremendous computational power is required, as scoring for each orientation needs large amount of CPU cycles, especially dealing with floating-point. The advantage of amber scoring is that both ligands and active sites of the receptor can be flexible during the evaluation. While the disadvantage is also obvious, it brings tremendous intensive floating-point computations. When performing amber scoring, it calculates the internal energy of the ligand, receptor and the complex, which can be broken down into three steps:

- Minimization
- molecule dynamics (MD) simulation
- more minimization using solvents

The computational complexity of amber scoring is very huge, especially in the MD simulation stage. Three grids which individually have three dimension coordinates are used to represent the molecule during the orientation such as geometry, gradient and velocity. In each grid, at least 128 elements are required to sustain the accuracy of the final score. During the simulation, scores are calculated in three nested loops, each of which walks through one of the three grids.

While many virtual screening tools such as GasDock [5], FTDock [6] and Dock6 can utilize multi-CPUs to parallel the computations, the incapacity of CPU in processing floating-point computations still remains untouched. Compared with CPU, GPU has the advantages of computational power and memory bandwidth. For example, a GeForce 9800 GT can reach 345 GFLOPS at peak rate and has an 86.4 GB/sec memory bandwidth, whereas an Intel Core 2 Extreme quad core processor at 2.66 GHz has a theoretical 21.3 peak GFLOPS and 10.7 GB/sec memory bandwidth. Another important factor why GPU is becoming widely used is that it is more cost-effective than CPU.

Our contributions in this paper include porting the original Dock6 amber scoring to GPU using CUDA, which can archive a 6.5x speedup. We analyze the different memory access patterns in GPU which can lead to a significant divergence in performance. Discussions on how to hide the computation divergence on GPU are made. We also conduct experiments to see the performance improvement.

The rest of the paper is organized as follows. In section II, an overview of Dock6's amber scoring and analysis of the bottleneck is given. In section III, we present the main idea and implementation of the amber scoring on GPU with CUDA, and details of considerations about performance are made. Then we give the results, including performance comparisons among various GPU versions. Finally, we conclude with discussion and future work.

2 Analysis of the Amber Scoring in Dock6

2.1 Overview

A primary design philosophy of amber scoring is allowing both the atoms in the ligand and the spheres in the receptor to be flexible during the virtual screening process, generating small structural rearrangements, which is much like the actual situation and gives more accuracy. As a result, a large number of docked orientations need to be analyzed and ranked in order to determine the best fit set of the matched atom-sphere pairs.

In the subsection, we will describe the amber scoring program flow and profile the performance bottleneck of the original amber scoring, which can be perfectly accelerated on GPU.

2.2 Program Flow and Performance Analysis

Figure 1 shows the steps to score the fitness for possible ligand-receptor pairs in amber. The program firstly performs conjugate gradient minimization, MD simulation, and more minimization with solvation on the individual ligand, the individual receptor, and the ligand-receptor complex, then calculates the score as follows:

$$\text{Ebinding} = \text{Ecomplex} - (\text{Ereceptor} - \text{Eligand}) . \quad (1)$$

The docked molecules are described using three dimension intensive grids containing the geometry, gradient or velocity coordinates information. The order of magnitude of these grids is usually very large. Data in these grids is represented using floating-point, which has little or no interactions during the computation. In order to archive higher accuracy, the scoring operation will be performed repeatedly, perhaps hundreds or thousands times.

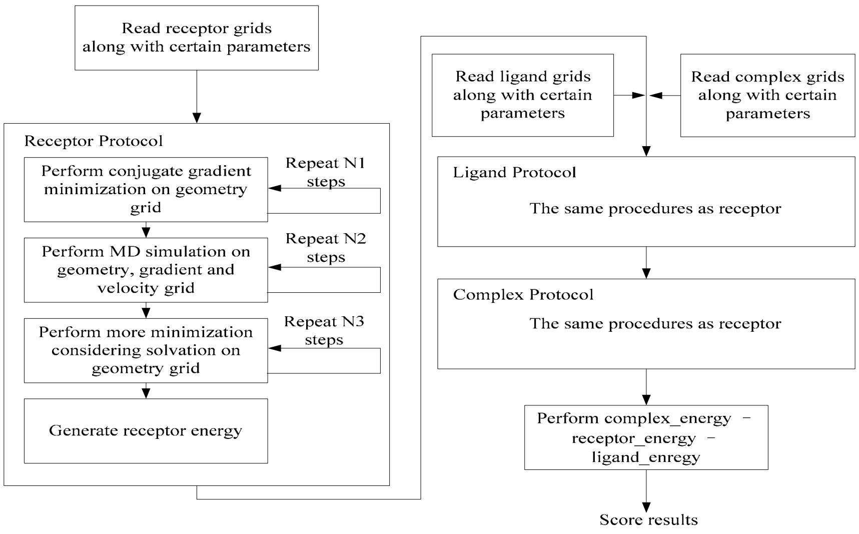
Read receptor grids along with certain parameters
Receptor Protocol
Perform conjugate gradient minimization on geometry grid
Repeat N1 steps
Perform MD simulation on geometry, gradient and velocity grid
Repeat N2 steps
Perform more minimization considering solvation on geometry grid
Repeat N3 steps
Generate receptor energy
Read ligand grids along with certain parameters
Read complex grids along with certain parameters
Ligand Protocol
The same procedures as receptor
Complex Protocol
The same procedures as receptor
Perform complex_energy − receptor_energy − ligand_enregy
Score results

Fig. 1. Program flow of amber scoring

Due to the characteristics of the amber scoring such as data independency and high arithmetic intensity, which are exactly the sweet spots of computing on GPU, it can be perfectly paralleled to leverage the computing power of GPU and gain preferable speedup.

3 Porting Amber Scoring to GPU

3.1 Overview

To determine the critical path of amber scoring, we conduct an experiment to make statistics about the cost of each step as Table 1 shows. We see that time spent on processing ligand is negligible, because ligand in docking always refer to small molecules or segments of protein whose information grids are small and can be calculated quickly. We also observe, however, that MD simulation on receptor and complex are the most time-consuming parts, which take up to 96.25 percentage of the total time. Therefore, in our GPU accelerated version, we focus on how to port the MD simulation to GPU, which could accelerate the bulk of the work.

Table 1. Run time statistics for each step of Amber scoring. 100 cycles are performed for minimization steps and 3,000 cycles for MD simulation step.

Stage		Run time (seconds)	Ratio of total (%)
Receptor protocol	gradient minimization	1.62	0.33
	MD simulation	226.41	45.49
	minimization solvation	0.83	0.17
	energy calculation	2.22	0.45
Ligand protocol	gradient minimization	≈0	0
	MD simulation	0.31	0.06
	minimization solvation	≈0	0
	energy calculation	≈0	0
Complex protocol	gradient minimization	8.69	1.75
	MD simulation	252.65	50.76
	minimization solvation	2.69	0.54
	energy calculation	2.22	0.45
Total		497.64	100

For the simplicity and efficiency, we take advantage of the Compute Unified Device Architecture (CUDA). We find the key issue to fully utilize GPU is high ratio of arithmetic operations to memory operations, which can be achieved through refined utilization of memory model, data transfer pattern, parallel thread management and branch hidden.

3.2 CUDA Programming Model Highlights

At the core of CUDA programming model are three key abstractions – a hierarchy of thread groups, shared memories and barrier synchronization, which provide fine-gained data parallelism, thread parallelism and task parallelism. CUDA defines GPU as coprocessors to CPU that can run a large number of light-weight threads concurrently. Threads are manipulated by kernels representing independent tasks mapped over each sub-problem. The kernel is invoked from the host side, most cases the CPU, as an asynchronized thread. The parallel threads collaborate through shared memory and synchronize using barriers.

In order to process on the GPU, data should be prepared by copying it to the graphic board memory first. Data transfer can be performed using deeply pipelined streams that overlap the kernel processing.The problem domain is defined in the form of a 2D grid of 3D thread blocks. The significance of thread block primitive is that it is the smallest granularity of work unit to be assigned to a single streaming multiprocessor(SM). Each SM is composed of 8 scalar processors (SP) that indeed run threads on the hardware in a time-slice manner. Every 32 threads within a thread block are grouped into warps. Within a warp the executions are in order, while beyond the warp the executions are out of order. However, if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously.

In addition to global memory, each thread block has its own private shared memory that is only accessible to threads within that thread block. Threads within a thread block could cooperate by sharing data among shared memory with low latency, while threads that belong to different thread blocks could only share data through global memory, which is slower by three orders of magnitude than shared memory. Synchronization within a thread block is implemented in hardware. Among thread blocks, synchronization can be achieved by finishing a kernel and starting a new one.

3.3 Parallel Thread Management

To carry out the MD simulation on GPU, a kernel needs to be written which is launched from the host (CPU) and executed on the device (GPU). A kernel is the same instruction set that will be performed by multiple threads simultaneously. By default, all the threads are distributed onto the same SM, which can't fully explore the computational power of the GPU. In order to utilize the SMs more efficiently, thread management must be taken into account.

We divide the threads into multiple blocks and each block can hold the same number of threads. In the geometry, gradient and velocity grids, 3D coordinates of atoms are stored sequentially and the size of the grid usually reach as large as 7,000. Calculation works are assigned to blocks on different SMs; each thread within the blocks computes the energy of one atom respectively and is independent of the rest (see Figure 2). We compose N threads into a block, which calculate N independent atoms in the grids. Assuming the grid size is M and M is divisible by N, there will happen to be M/N blocks.

While in most cases the grid size M is not divisible by N, we designed two schemes dealing with this situation. In the first scheme, there will be M/N blocks. Since there is M%N atoms left without threads to calculate, we will rearrange the atoms evenly to the threads in the last block. One more atom will be added to the threads in the last block until no atoms left, which is ordered by thread ID ascending. The second scheme is to construct M/N + 1 blocks. Each thread in the blocks still calculates one atom however the last block may contain threads with nothing to do. Control logic should be added to the kernel to judge whether the thread has some calculations or not through comparing the value of block ID * N + thread ID and M.

Our experiment proves the former scheme obtained better performance. This is caused by underutilized SM resources and branch cost in the second scheme. When there is a branch divergence, all the threads must wait until they reach the same instructions again. Synchronization instructions are generated by the CUDA complier automatically, which is time consuming.

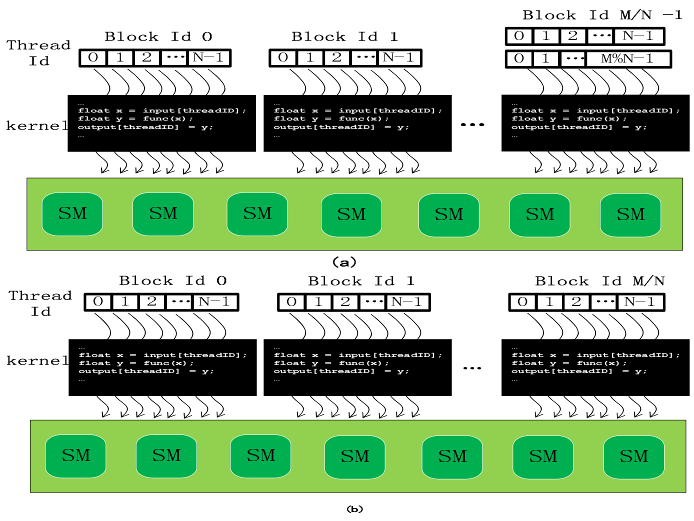

Fig. 2. Threads and blocks management about processing molecule grids on GPU: (a) blocks whose threads in the last block may calculate two atoms each (b) blocks whose threads in the last block may have nothing to do.

3.4 Memory Model and Data Transfer Pattern

The first step to perform GPU computations is to transfer data from host (CPU) memory to device (GPU) memory since the receptor, ligand and complex grids need to be accessible by all SMs during the calculations. There are two kinds of memory can be used to hold these grids. One is the constant memory, which can be read and written by the host but can only be read by the device. The other is the global memory, which can be read and written by both the host and device. One important distinction between the two memories is the access latency. SMs can get access to the constant memory magnitude order faster than the global memory. While the disadvantage of the constant memory is also obvious, it is much smaller, which is usually 64 KB compared to 512 MB global memory. Thus, a trade-off has to be made on how to store these grids.

During each MD simulation cycle, the gradient and velocity grids are read and updated. Therefore, they should be stored in global memory. While once entered the MD simulation process, the geometry grids are never changed by the kernel. Hence, they can be stored in constant memory (see Figure 3). Considering the out-bound danger which dues to the limited capacity of the constant memory, we observed the size of the each geometry grid. The receptor and complex geometry grids usually contain no more than 2,000 atoms each while the ligand geometry grid contains 700 atoms, which totally requires 2,000 * 3 * 4 * 2 + 700 * 3 * 4 bytes (56.4 KB) memory to store them. Since it is smaller than 64 KB, the geometry grids shall never go out-bound of the constant memory.

The time to transfer molecule grids from host to their corresponding GPU memory is likewise critical issue, which may degrade the benefit achieved from the parallel execution if without careful consideration [7]. For each MD simulation cycle, we

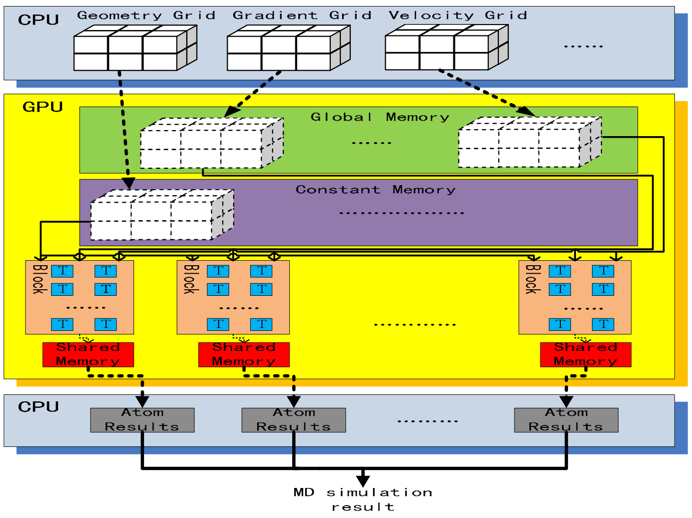

Fig. 3. Memory model and data transfer pattern during the MD simulation cycles. Grids are transferred only once before the simulation, which are stored in global memory and constant memory correspondingly. Atom results are first accumulated in the shared memory within the block. Then the accumulations per block are transferred into the host memory and summed up to the final MD simulation result for the molecule.

could transfer one single atom 3D coordinates in the geometry, gradient and velocity grids to device memory when they are required by the SMs. The other solution is to transfer the entire grids into the GPU memory before the MD simulation stage. When the simulation starts, these grids are already stored in device memory which can be accessed by simulation cycles performed on SMs.

Based on our experiment, significant performance improvements are obtained from the second scheme since the molecule grids are transferred only once for all before the MD simulation. When the calculations on the SMs are carried out, the coordinates of the atoms in the grids are already stored in the device memory. Therefore, the SMs don't have to halt and wait for the grids to be prepared, which obviously speeds up the parallel execution of the MD simulations by fully utilizing the SMs.

The MD simulations are executed parallel on different SMs, and threads within the different blocks are responsible for the calculations of their assigned atoms of the grids. The final simulation result is formed by accumulating all results. Our solution is to synchronize threads within the blocks, which generates atom results separately. Then a transformation is performed to store the atom results from shared memory to host memory in a result array, whose index is identical to the block ID. The molecule result shall be achieved by adding up all the elements in the array without synchronization since the results are transferred only when the calculations on device are accomplished.

3.5 Divergence Hidden

Another important factor that dramatically impact the benefits achieved by performing MD simulation on GPU is the branches. Original MD simulation procedure has involved a bunch of nested control logics such as bound of Van der Waals force and constrains of molecule energy. When the parallel threads computing on different atoms in the grids come to a divergence, a barrier will be generated that all the threads will wait until they reach the same instruction set again. The above situation can be time-consuming and outweigh the benefits of parallel execution.

We extract the calculations out of the control logic. Each branch result of the atom calculation is stored in a register variable. Inside of the nested control logic, only value assignments are performed, which means the divergence among all the threads will be much smaller, thus the same instruction sets can be reached with no extra calculation latency. Although this scheme will waste some computational power of the SMs since only a few branch results are useful in the end, it brings tremendous improvements in performance. These improvements can be attributed to that, in most cases, the computational power we required during the MD simulation is much less than the maximum capacity of the SMs. Hence, the extra calculations only consume vacant resources, which in turn speed up the executions. The feasibility and efficiency of our scheme have been demonstrated in our experiment.

4 Results

The performance of our acceleration result is evaluated for two configurations:

- Two cores of a dual core CPU
- GPU accelerated

The base system is a 2.7 GHz dual core AMD Athlon processor. GPU results were generated using an NVIDIA GeForce 9800 GT GPU card.

Table 2. CPU times, GPU times and speedups with respect to 3,000 MD simulation cycles per molecule protocol. The CPU version was performed using dual core, while GPU version with all superior scheme.

Stage		CPU	GPU	Speedup
Receptor protocol	gradient minimization	1.62	0.89	1.82
	MD simulation	226.41	31.32	7.23
	minimization solvation	0.83	0.15	5.53
	energy calculation	2.22	1.21	1.83
Ligand protocol	gradient minimization	≈0	0.02	——
	MD simulation	0.31	0.60	——
	minimization solvation	≈0	0.03	——
	energy calculation	≈0	≈0	0
Complex protocol	gradient minimization	8.69	2.88	3.02
	MD simulation	252.65	34.79	7.26
	minimization solvation	2.69	2.05	1.31
	energy calculation	2.22	1.47	1.51
Total		497.64	75.41	6.5

We referred to the Dockv6.2 as the original code, which was somewhat optimized in amber scoring. We also used the CUDAv2.1, whose specifications support 512 threads per block, 64KB constant memory, 16KB shared memory and 512MB global memory. Since double precision floating point was not supported in our GPU card, transformation to single precision floating point was performed before the kernel launched. With small precision losses, the amber scoring results were slightly different between CPU version and GPU version, which can be acceptable.

Table 2 compares the original CPU version with the GPU accelerated version in runtime for various stages. The MD simulation performed are 3,000 cycles each molecular stage. The overall speedup achieved for the entire amber scoring is over 6.5x.

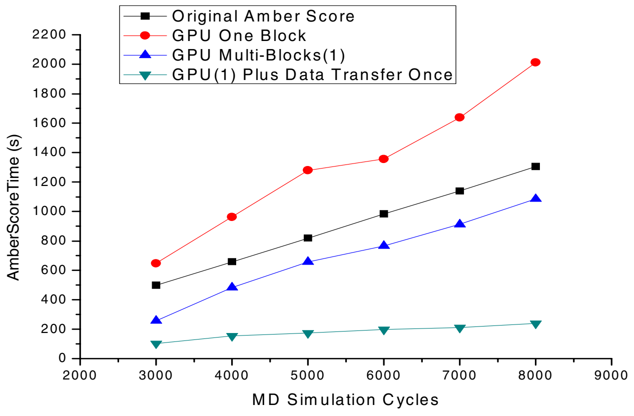

Fig. 4. Shown is a comparison of amber scoring time between original amber and different GPU versions whose speedup varies significantly as the MD simulation cycles increasing from 3,000 to 8,000

Figure 4 depicts the total speedups of different GPU schemes with respect to the range of increasing MD simulation cycles. As mentioned in section 3.3, the GPU version with only one block did not speedup, which was attributed to the poor management of threads since each block had a boundary of maximum active threads. In our experiment, GeForce 9800 GT specification limits each block can only hold 512 threads maximally. This limitation will force large amount of threads waiting on the only one block until other threads are served and release the SM resource. Since the MD simulation requires more than one block threads to calculate the atom results, the latency becomes more obviously as the MD simulation cycles scale. Fortunately, with multiple blocks, this kind of thread starvation latency can be greatly eased. Threads within multiple blocks can be scheduled onto different SMs so that calculations of independent atoms are executed parallel. The most significant performance improvements are achieved from transferring the molecule grids only once during the MD simulation in addition to the usage of multi-blocks.

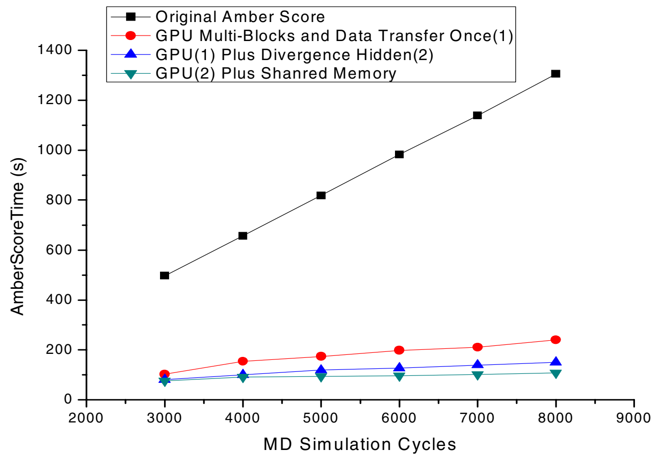

Fig. 5. Comparison of speedups among different GPU versions based on Figure 4 in addition to divergence hidden and shared memory

Figure 5 depicts the second speedup in performance comes from the utilizations of divergence hidden and synchronization on shared memory. Since the branch calculations are extracted out of the control logic and stored in temporary variables, only one single instruction will be performed which assigns corresponding values into the final result when divergences occur. This scheme greatly shortens the time consumed for all the threads to return to the same instruction sets. While threads within a block will accumulate atom simulation values into a partial result of molecule on shared memory, the result array transferred back to the host is very small. Performance improvements are obtained when summing up the elements in the array to form the molecule simulation result. We also notice that as the MD simulation cycles scales, the speedup becomes more considerable in our best GPU version.

5 Related Work

Exploiting GPUs for general purpose computing has recently gained popularity particularly as a mean to increase the performance of scientific applications. However most of the accelerations of science-oriented applications on GPU are in the fields of graphic processing and arithmetic algorithms. Kruger et al. [8] implemented linear algebra operators for GPUs and demonstrated the feasibility of offloading a number of matrix and vector operations to GPUs. Nathan Bell [9] demonstrated several efficient implementations of sparse matrix-vector multiplication (SpMV) in CUDA by tailoring the data access patterns of the kernels.

Studies on utilizing GPU to accelerate molecule docking and scoring problems are rare, the only work we find more related to our concern is in the paper of Bharat Sukhwani [10]. The author described a GPU-accelerated production docking code, PIPER [11], which achieves an end-to-end speedup of at least 17.7x with respect to a single core. Our contribution is different from the former study in two aspects. First, we focus our energy on flexible docking such as amber scoring while the previous study mainly work on rigid docking using FFT. Thus our work is more complex and competitive in the real world. Second, we noticed the logic branches in the parallel threads on GPU degraded the entire performance sharply. We also described the divergence hidden scheme and represented the comparison on speedup with and without our scheme.

Another attractive work needs to be mentioned is that Michael Showerman and Jeremy Enos[12] developed and deployed a heterogeneous multi-accelerator cluster at NCSA. They also migrated some existing legacy codes to this system and measured the performance speedup, such as the famous molecular dynamics code called NAMD[13, 14]. However, the overall speedup they achieved was limited to 5.5x since they could not utilize the computation power of GPU and FPGA simultaneously.

6 Conclusion and Future Works

In this paper we present a GPU accelerated amber score in Dock6.2, which achieves an end-to-end speedup of at least 6.5x with respect to 3,000 cycles during MD simulation compared to a dual core CPU. We find that thread management utilizing multiple blocks and single transferring of the molecule grids dominate the performance improvements on GPU. Furthermore, dealing with the latency attributed to thread synchronization, divergence hidden and shared memory can be elegant solutions which will additionally double the speedup of the MD simulation. Unfortunately the speedup of Amber scoring can't go much higher due to Amdahl's law. The limits are in multiple ways:

- With the kernel running faster because of GPU accelerating, the rest of the Amber scoring takes a higher percentage of the run-time
- Partitioning the work among SMs will eventually decrease the individual job size to a point where the overhead of initializing an SP dominates the application execution time

The work we presented in this paper only shows a kick-off stage of our exploration in GPGPU computation. We will proceed to use CUDA accelerating various applications with different data structures and memory access patterns and hope to be able to work out general strategies about how to use the manycore feature of GPU more efficiently.

Acknowledgment

Many thanks to Ting Chen for thoughtful discussions and comments about our implementation and paper work. This work was supported by the National Natural Science Foundation of China under the grant No. 90812001, Chinese National Programs for High Technology Research and Development under the grant No. 2006AA01A 124 and 2006AA01A118.

References

1. Dock6, `http://dock.compbio.ucsf.edu/DOCK_6/`
2. Wang, J., Wolf, R.M., Caldwell, J.W., Kollman, P.A., Case, D.A.: Development and testing of a general Amber force field. Journal of Computational Chemistry, 1157–1174 (2004)
3. NVIDIA Corporation Technical Staff.: Compute Unified Device Architecture - Programming Guide, NVIDIA Corporation (2008)
4. Kuntz, I., Blaney, J., Oatley, S., Langridge, R., Ferrin, T.: A geometric approach to macromolecule-ligand interactions. Journal of Molecular Biology 161, 269–288 (1982)
5. Lia, H., Lia, C., Guib, C., Luob, X., Jiangb, H.: GAsDock: a new approach for rapid flexible docking based on an improved multi-population genetic algorithm. Bioorganic & Medicinal Chemistry Letters 14(18), 4671–4676 (2004)
6. Servat, H., Gonzalez, C., Aguilar, X., Cabrera, D., Jimenez, D.: Drug Design on the Cell BroadBand Engine. In: Parallel Architecture and Compilation Techniques, September 2007, p. 425 (2007)
7. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: High-performance graphics coprocessor sorting for large database management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (2006)
8. Kruger, J., Westermann, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In: ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques (2003)
9. Nathan, B., Michael, G.: Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004 (Dec. 2008)
10. Bharat, S., Martin, C.H.: GPU acceleration of a production molecular docking code. In: Proceedings of 2nd Workshop on General Purpose Processing on GPUs, pp. 19–27 (2009)
11. PIPER, `http://structure.bu.edu/index.html`
12. Michael, S., Hwu, W.-M., Jeremy, E., Avneesh, P., Volodymyr, K., Craig, S., Robert, P.: QP: A Heterogeneous Multi-Accelerator Cluster. In: 10th LCI International Conference on High-Performance Clustered Computing (March 2009)
13. NAMD, `http://www.ks.uiuc.edu/Research/namd/`
14. Phillips, J.C., Zheng, G., Sameer, K., Kalé, L.V.: NAMD: Biomolecular Simulation on Thousands of Processors. In: Conference on High Performance Networking and Computing, pp. 1–18 (2002)