Accelerating the calculation of scattering of complex targets from background radiation with CUDA, OpenACC and OpenHMPP

Xing Guo[1], Zhensen Wu[1*], Jiaji Wu[2]
1. School of Physics and Optoelectronic Engineering, Xidian University, Xi'an, Shaanxi, China
2. Institute of Intelligent Information Processing, Xidian University, Xi'an, Shaanxi, China
[*]Corresponding author : wuzhs@mail.xidian.edu.cn

***Abstract*—Graphics Processing Unit (GPU) is used to accelerate the calculation of scattering of complex target from background radiation in infrared spectrum. Compute Unified Device Architecture (CUDA), OpenACC, and Hybrid Multicore Parallel Programming (OpenHMPP) implementations are presented. In all our implementation, scattering of background radiation in different directions are calculated in parallel. A personal desktop with 2 NVIDIA GTX GeForce 590 with an Intel i7 CPU is used in our experiment. In CUDA, by using shared memory to buffer the background radiation and BRDF parameters and tuning the grid organization, we achieve a speedup of 197x. OpenACC implementation is realized by inserting the parallel loop construct with reduction clause before the loop in original serial code. By utilization of data clause and tuning number of gangs used, a speedup of 158.9x is obtained. In OpenHMPP implementation, the loop iterating over incident direction of original code is transformed to the codelet function and we achieve a speedup of 160.7x. Our effort makes the calculation of complex target in real time possible.**

Keywords- scattering of target, background radiation, GPU, CUDA, OpenACC, OpenHMPP

I. INTRODUCTION

Scattering characteristic of complex target from background radiation is of great value for infrared radiation, imaging, remote sensing and detection and tracking of target and has been widely studied [1-3]. In 2002, Anan Liu analyzed scattering characteristic of simple target such as board with surface which is of uniform Lambertian reflectance from celestial and terrestrial radiation [4, 5]. Complex target with non-lambert surface was studied in [6, 7]. As the background radiation comes in all directions, the processing is quite time consuming thus limits its application in engineering.

In recent years after the release of compute unified frameworks such as Compute Unified Device Architecture (CUDA) and Open Compute Language (OpenCL), programming GPUs has become much easier and more popular. GPU has been used to accelerate a wide range of research applications such as, image processing [8], feature tracking, coding and decoding of video [9], Monte Carlo simulation of 2D and 3D Ising model [10], medical image processing, like CT reconstruction [11], image segmentation [12], image visualization [13], visualization of ultrasound from CT images [14], radiative transfer model for the infrared atmospheric sounding interferometer [15, 16], etc.

Calculation of scattering of target has high inherent parallelism and is very suitable for GPU implementation where the scattering of background radiation from different incident directions can be calculated in parallel. Previous effort of CUDA implementation of scattering of complex target has been done by Li in 2011 [17]. Based on Li's work further optimization is made and higher speedup is obtained in this work.

However both of the two models mentioned above are low level framework, and require the programmer to have a deep understanding of the underlying architecture. Additionally a larger number of codes have to be rewritten including compute kernels, memory allocation and free of accelerator, data transferred to and from the accelerator and necessary synchronization, etc. So using GPU needs quite a lot programming effort Due to the success of OpenMP in shared memory platform, a lot of efforts have been made to develop high-level programming models for heterogeneous system. Most recently various such models have been proposed as OpenHMPP, OpenACC, HiCUDA, OpenMPC, PGI Accelerator, etc. And there're have been many reports about these high-level parallel programming models[18,19].

Additionally in this paper, OpenACC and OpenHMPP versions of the applications are created to drive the automatic generation of the GPU code from sequential code. The results demonstrate a significant speedup using each parallelization method.

The remainder of this paper is organized as follows: Next section presents the brief description of calculation process of the scattering of complex target from background radiation. Section 3 makes a simple introduction of CUDA, OpenACC and OpenHMPP programming models. This is followed by analysis of implementations of CUDA, OpenACC, and OpenHMPP and comparison with CPU version in section 4. Finally section 5 concludes the paper and outlines plans for future work.

II. SCATTERING OF COMPLEX TARGET

To calculate the scattering of target, first of all, the geometrical model of the target is constructed by using 3DMAX. Then the surface of target is divided into thousands of triangular facets each of which is treated as surface with small roughness. And scattering of each facet from surface ground and sky radiation both of which can be obtained with the software MODTRAN is calculated based on the rough surface theory. Finally the contributions of all visible facets are added to get the total scattering radiance.

1521-9097/13 $31.00 © 2013 IEEE
DOI 10.1109/ICPADS.2013.125

BRDF (Bidirectional Reflectance Distribution Function) is defined as the ratio of the scattering radiance to the incident irradiance .The Five-parameter BRDF model, one of the many popular models to represent different materials, is presented and adopted in our implementation. [17]

According to the definition of BRDF and rough surface theory, the scattering of each facet from background radiation from a given incident direction with a given wavelength can be expressed as

$$dL_{\lambda,\text{facet}}(\theta_s,\phi_s) = f_r(\theta_s,\theta_i,\phi,\lambda)L_\lambda(\theta_i,\phi_i)d\omega_i \quad (1)$$

where $d\omega_i = \cos\theta_i \sin\theta_i d\phi_i d\theta_i$, BRDF of each facet is $f_r(\theta_s,\theta_i,\phi,\lambda)$, $L_\lambda(\theta_i,\phi_i)$ presents background radiation in incident direction referenced by zenith angle θ_i and azimuth angle ϕ_i with wavelength λ , $dL_{\lambda,\text{facet}}(\theta_s,\phi_s)$ represents the scattering radiation of each facet in direction referenced by θ_s and ϕ_s .

It is worth noting that θ_s,θ_i,ϕ in f_r are in self-coordinates of each facet so transformation of the standard incident and observation direction have to be made.

As surface ground and sky radiation comes in all directions, we need to calculate the scattering from each incident zenith and azimuth angle then integrate over the whole space,

$$L_{\lambda,\text{facet}}(\theta_s,\phi_s) = \int_0^{2\pi} d\phi_i \int_0^{\pi} f_r(\theta_s,\theta_i,\phi,\lambda)L_\lambda(\theta_i,\phi_i)\cos\theta_i \sin\theta_i d\theta_i \quad (2)$$

We can obtain the total scattering radiance of target in a certain spectrum by integration of spectral

$$L(\theta_s,\phi_s) = \int_{\lambda_1}^{\lambda_2} \left(\sum_{k=1}^{n} L_{\lambda,\text{facet}}(\theta_s,\phi_s)(S_k * \cos\theta_s) / \sum_{k=1}^{n} S_k * \cos\theta_s \right) d\lambda \quad (3)$$

III. PROGRAMMING MODELS

In this section, brief introduction of three programming models including CUDA, OpenACC, and OpenHMPP is presented.

A. CUDA

CUDA is a general purpose parallel computing platform and programming model introduced by NVIDIA and implemented by GPUs they produce. CUDA allows users to program the GPU directly. It is a small set of extensions to standard programming languages like C and Fortran. The CUDA model supports heterogeneous computation where CPU called host run the serial portions of applications while GPU called device run the parallel regions which are implemented as kernel functions. CUDA runtime provides library functions for device memory management and data transfer between the host and the device.

Threads are organized into three-level hierarchy. A thread is the fundamental building block of a parallel program [20]. Threads are grouped into blocks and the number of threads per block is limited by shared and register memory used. Grid as the highest level consists of blocks. The exact organization of a grid is determined by the execution configuration parameters of the kernel launch statement.

Each thread has private local memory and register. Shared memory is visible to all threads in a block which is expected to be much faster than global memory and enables threads within the same thread block to cooperate and facilitates extensive reuse of on-chip data. All threads have access to global memory which has the largest space but relatively slow access speed. Additionally there're two kinds read-only cached memory accessible by all threads: the constant and texture memory. The constant and texture memory are always optimization candidates for different memory usages. The CUDA grid organization and memory model is presented in.

Actually CUDA is a practical and most reliable approach as of now [21]. This approach also delivers quite a high performance. However, as can be seen from above it requires a lot programming efforts including allocation and free of device memory, and data transfer between host and device and necessary synchronization thus may affect its usability and productivity.

B. OpenACC

OpenACC as a parallel programming standard was proposed by NVIDIA, Cray, Portland Group and CAPS [34] in 2011 to make it easier for programmers to take advantage of hardware computing accelerators. OpenACC provides compiler directives, library routines and environment variables for specifying regions of an algorithm to be offloaded from the host CPU to an attached accelerator device in C, C++ and Fortran programs [39]. The model is portable across operating systems and various types of CPUs, accelerators and compilers. Programmers can use accelerators without allocating and free device memory, managing data transfer between host and device. Programs using OpenACC can be generated for both NVIDIA and AMD GPU as it supports both CUDA and OpenCL target language.

OpenACC has directives for specifying parallel regions that will be run on device and managing data location and how they transferred to device. OpenACC provides two types of constructs to define parallel region: the kernels construct and parallel construct. Both constructs have clauses to manage data transfer, to indicate whether the host and accelerator operate in an asynchronous mode. OpenACC defines three levels of parallelism; gang, worker, and vector. However the OpenACC model does not provide enough control over various optimizations and translations, and it also does not offer directives to express architecture-specific features and support for multi-GPU [22].

C. OpenHMPP

Like OpenACC, OpenHMPP is another directive-based parallel programming interface for heterogeneous system which targets both CUDA and OpenCL and is initially developed by CAPS enterprise. CAPS Compilers contains C and FORTRAN compiler drivers, target code generator, and

runtime to simplify the utilization of GPUs and many cores system. The code generator is responsible to extract the parallelism and translate them into CUDA and OpenCL. The HMPP runtime provides libraries to initialize hardware accelerator (HWA), allocate data memory on HWA, manage data transfer between the host and HWA and call the remote procedure (RPCs) [23].

The basic concepts of OpenHMPP are codelet and callsite and the simplest use case of OpenHMPP directives is composed of only two directives made of a codelet declaration and callsite marker [23]. Codelet defines functions that can be remotely executed on hardware accelerators and the callsite directive declares where the codelet function is called. The programmer always needs to rewrite or reconstruct original serial code to create codelet function. Also OpenHMPP provides region directive which is a merge of the codelet and callsite directive to avoid code restructuration to build the codelet. Group directive declares a group of codelets which allows data sharing between different codelets. The asynchronous clause of callsite directive is useful to overlap operation on the accelerator with that on the host so that it's always a choice to further optimize OpenHMPP applications.

To optimize the generation and mapping of codelets into target code, OpenHMPP provides a rich set of directives, called HMPP Codelet Generator Directives [24] and started with #pragma hmppcg. These directives allow programmers to use CUDA special memories like constant and shared memory, to manual determine block size of the a loop nest, and to control complex loop mapping and loop transformations.

IV. IMPLEMENTATION AND RESULTS DISCUSSION

Our GPU-based parallel implementation of the scattering of complex targets is performed on a PC desktop equipped with an Intel i7 2600K CPU and 2 NVIDIA GeForce GTX 590 cards each of which has 1024 CUDA cores. The specification of GTX 590 is listed in Table 1.

CUDA 4.2 is used to for compiling GPU code. The CAPS Compilers 3.3.0 from CAPS Enterprise is used to compile and run the two directive-based parallel implementations. And both OpenACC and OpenHMPP implementations are targeting CUDA language.

For the speedup measures all time measurement includes memory transfer between host and accelerator and is obtained as an average of calculation of 10 different observation directions. For all speedup, the time cost in serial implementation is used as the base time. The airplane model and its scattering are shown in Fig. 1 and Fig. 2 respectively.

Table 1 Specification of NVIDIA GeForce GTX 590

CUDA cores	1024
GPU clock rate	1.23GHz
Global memory	1535MB
Shared memory (per SM)	48KB or 16KB
L1 Cache (per SM)	16KB or 48KB
L2 Cache size	768KB

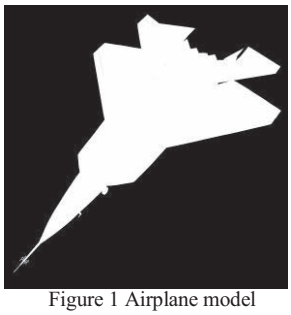

Figure 1 Airplane model

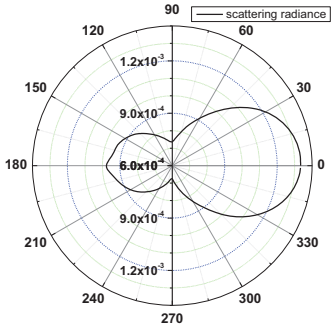

Figure 2 Scattering of the airplane target

In our calculation one hypothesis is made that in infrared bands the average celestial radiation can be approximately replaced by background radiation whose zenith angle is 45°and 135°background upward radiation can replace the average terrestrial radiation [5]. In MODTRAN, we set the atmosphere model as 1976 US standard, without consideration of cloud or rain. Observe time is at 2.5 on July 1st, 2007(Greenwich time), and observe position is 45° N，120° E at 2 KM height. The spectrum is from 3 to 5 with a step of 0.075. Aerosol model used is rural with a VIS of 23KM.

A. Serial implementation

In our implementation, a function named Intensity gives the scattering of the target from background radiation in a given incident zenith and azimuth direction in which ks is the detection direction, W, Bter, Bcel are surface ground and sky radiation, KB, KR, KD, AA, BB are the parameters of BRDF, Nx, Ny, Nz, S denote the normal line and area of each facet.. And function area calculates the projection area in observation direction.

Table 2 Sketch of CPU serial implementation

```
99 for (ia=0; ia<angleNum; ia++){
100       float radiance = 0.0f;
101       for(ij = 0; ij < 180 * 360; ++ij) {
102       float theta = ij / 360 * 1.0f;
103       float phi = ij % 360 * 1.0f;
104       radiance += Intensity(ks, W, Bter, Bcel, KB, KR,
    KD, AA, BB, Nx, Ny, Nz, S, FacetNum, WaveNum,
    theta, phi);
105       }
106       float areap = area(); //projection area
107       radiance = radiance / areap;
108 }
```

To calculate the scattering of target in a given direction, we need a loop to cumulative sum of scattering radiation from different directions. So in our implementation, there're double loops, the outer loop iterates over observing directions, whereas the inner loop defines incident directions. Both of the two loops are stepped with 1 degree. A sketch of the CPU implementation is shown in Table 2. The serial version is compiled using GCC 4.6.3 with flag –O2. The calculation of scattering of target for a given observe direction costs 196 seconds.

B. CUDA implementation

The parallelism in this problem makes it very suitable for the GPU implementation. In our implementation, the construction of geometrical model and calculation of background radiation by MODTRAN, and the calculation of normal vector and area of each facet are done by the host CPU. The calculation of the scattering from different incident directions as the most time-consuming part is processed in parallel by different threads on device where each thread computes the scattering of all facets from background radiation in a given incident direction. Both the incident zenith and azimuth angle are divided into intervals of 1 degree. Thus total the number of threads equals the numbers of 180*360. Through reduction, the sum of scattering in a block are obtained then transferred back to host to get the total scattering of the target. To calculate the scattering in different observing directions, the kernel has to be called multiple times. We used 256 threads for each of 256 blocks and achieve a speedup of 188.2x shown in Table 3.

1) Optimization 1 by Tuning block size

To evaluate the effect of block size on performance, the block size varies from 32 to 512. Registers used per thread and shared memory used per block are given, the block size determines the occupancy of the stream multiprocessor. As can be seen from Fig. 3, best performance is obtained with the block size as 64. When using 64 threads in each block the time cost is 1.038 seconds which means a speedup of 188.8x is achieved shown as Table 3.

2) Optimization 2 by Using shared memory

The bandwidth of on-chip shared memory is much higher than that of global memory. All threads in the same block can have access to the same shared memory, so threads can data in shared memory loaded from global memory by other threads within the same block so as to reduce the accessing of the global memory.

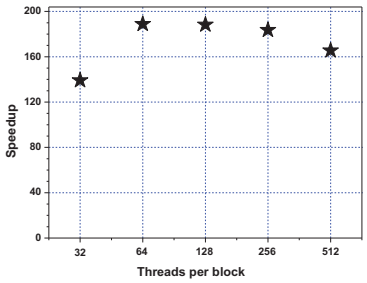

Figure 3 Effect of block size on performance

Table 3 Performance of CUDA implementation

	Processing time[s]	Speedup
CPU serial	196	
CUDA	1.04	188.2x
Optimization1	1.038	188.8x
Optimization2	0.991	197.7x

In our implementation, shared memory in each block is used to buffer the background radiation and BRDF parameters. In this CUDA version, the block size is kept as 64. Experiment is conducted, and this version costs 0.991seconds and a speedup of 197.7x is achieved shown as Table 3.

C. OpenACC implementation

In our first OpenACC implementation, we insert the parallel construct with reduction clause in original serial code as #pragma acc parallel loop reduction(+:Result) present_or_copyin(W[0:27], Bter[0:27], Bcel[0:27], KB[0:27], KR[0:27], KD[0:27], AA[0:27], BB[0:27], Nx[0:4708], Ny[0:4708], Nz[0:4708], S[0:4708], FacetNum, WaveNum) copyin(h_K[0:3]) copy(radiance) before line 100. Considering background radiation, BRDF parameters and normal vectors and areas for each facet are identical for different detection direction, present_or_copyin clause is used to indicate these data need be transferred to device only once and need not be transferred back to host. While the unit normal vector of detection direction differs with detection direction, copyin clause is applied to indicate to transfer the argument h_K to device every time at the entry of compute region. The argument result, as reduction clause parameter, copy clause was used to show it should be copied to device to assign initial value on device and copied back to host after calculation. A speedup of 115.6x is obtained.

As can be seen from above in our implementation, the parallel section is located in a loop, which means that it will be called multiple times, such that all the memory (de)allocations on the device and data transfers will occur when the section is entered and exited, which incurs an absolutely non-negligible overhead.

1) Optimization 1 by using data clause

Thanks to the data clause, its main purpose is to avoid multiple re-allocations of GPU memory and transfers between the host and the GPU [40]. With data clause device memory is allocated and initialized with the host's value once before the loop, and transferred back to the host and freed when the data section ends. Our effort increases the speedup to 116.7x.

2) Optimization 2 by tuning number of gangs and workers used

As is known, like CUDA the organization of grid and block has an important effect on performance. OpenACC provides num_gangs and num_workers clause in parallel construct and gang, worker clause in loop construct which define the number of parallel gangs and the number of workers within each gang that will execute the region.

In our implementation, the default gangs and workers are 192 and 256 respectively.

Table 4 Performance of OpenACC implementation

	Processing time[s]	Speedup
CPU serial	196	
OpenACC	1.70	115.6x
Optimization 1	1.70	116.7x
Optimization 2	1.23	158.8x

We have done experiments and found that 1024 gangs and 128 workers give highest speedup of 158.8x which means calculating scattering of target form a given direction cost approximately 1.23 seconds. Performances of three versions of OpenACC implementations are listed in Table 4 from which we can see that the number of gangs and workers used has the significant effect on performance.

D. OpenHMPP implementation

In our implementaion we outline the ij loop of line 101 in original serial code to create a codelet function with HMPPCG reduce clause which specifies a reduction operation is performed. #pragma hmpp target codelet target=CUDA is added before the definition of the codelet function. As the codelet function does not return value, so a function with a pointer to save the result is created. Obviously, the pointer pointed to result needs to be copied back to host every time the codelet is called, so args[].io = out and args[].transfe =atcall is applied on argument. The unit normal vector of detection direction needs be transferred to accelerator for different observe directions so args[].io=in and args[].transfe =atcall is applied. Similar with OpenACC implementation, background radiation and normal vectors and areas for each facet are identical for different detection directions, they need only be transfer to accelerator memory once and they need not be transferred back to host, so args [].transfer = firstcall, and args [].io = in is used. After compiled with CAPS compiler, we get compilation message as loop 'ij' not gridified: Inter-iterations dependencies found. So #pragma hmppcg gridify(ij) is adopted to help compiler identify parallel loops and how they should be mapped to the GPU. In the main function, #pragma hmpp target callsite is added before the call of the codelet function.

Like CUDA, the organization of block size has a significant effect on performance. So HMPPCG directives allow the user to explicitly define the block size of a loop. Clause blocksize 128x1 is added in the HMPPCG directives. Experiments with different block size are conducted and results are depicted in Fig. 4.

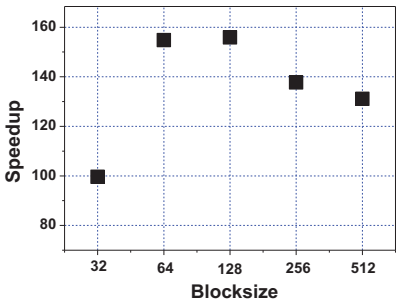

Figure 4 Effect of block size on performance of HMPP

V. CONCLUSIONS AND FUTURE WORK

The scattering of complex target from background radiance is studied in this paper. And CUDA, OpenACC and OpenHMPP implementations are made. CUDA gives the best performance. It's clear that OpenACC and OpenHMPP improve the programmer productivity in terms of reduced number of lines of code rewritten. In OpenACC implementation, only parallel directives are inserted in original serial code without changing it at all. And in OpenHMPP implementation one loop is outlined to create codelet function and the callsite directive is added before the call of the codelet function with much less code rewritten than that of CUDA implementation.

Now quite a number of graphics cards have more than one GPU, so our next work is to use this architecture to verify the whether it can help accelerate the speedup.

ACKNOWLEDGMENT

This work has been supported by the National Natural Science Foundation of China under Grant NO.61172031 and the Fundamental Research Funds for the Central Universities (NO. K5051207001).

REFERENCES

[1] Z. Yandong and W. Zhensen, "Scattering of radiation of the sun and the sky from two dimensional rough sea surfaces," International Journal of Infrared and Millimeter Waves, vol. 25, 2004, pp. 1013-1022.

[2] W. Zhensen, Z. Caixia, C. Hui, and Z. Yandong, "Computation of Near Sea Target Scattering from Complex Background Radiation," International Journal of Infrared and Millimeter Waves, vol. 24, 2003, pp. 1989-1998.

[3] Z. Zhu, H. Liang, A. Pan, B. Song, G. Xu, and G. Ni, "Detection and acquisition of small targets with low signal-to-clutter ratio," in Signal and Data Processing of Small Targets, 1999, pp. 564-569.

[4] Z. Wu, A. Liu, and Y. Xue, "Scattering of solar and atmospheric background radiation incident to target," in International Symposium on Remote Sensing, 2002, pp. 354-361.

[5] Z. Wu and A. Liu, "Scattering of solar and atmospheric background radiation from a target," International Journal of Infrared and Millimeter Waves, vol. 23, 2002, pp. 907-917.

[6] Y. Yufeng, "Scattering characteristics of complex background infrared radiation from a non-lambertian target," Infrared and Laser Engineering, 2011, p. 05.

[7] Y.-f. Yang, Z.-S. Wu, and L.-C. Li, "The effect of complex background infrared radiation on target scattering radiance," in Antennas Propagation and EM Theory (ISAPE), 2010 9th International Symposium on, 2010, pp. 749-752.

[8] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on CUDA," in Computer Science and Software Engineering, International Conference on, 2008, pp. 198-201.

[9] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," Machine Vision and Applications, vol. 22, 2011, pp. 207-217.

[10] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising

model," Journal of Computational Physics, vol. 228, 2009, pp. 4468-4477.
[11] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the common unified device architecture (CUDA)," in Nuclear Science Symposium Conference Record, 2007. NSS'07. IEEE, 2007, pp. 4464-4466.
[12] L. Pan, L. Gu, and J. Xu, "Implementation of medical image segmentation in CUDA," in Information Technology and Applications in Biomedicine. ITAB International Conference on, 2008, pp. 82-85.
[13] Y. Heng and L. Gu, "GPU-based volume rendering for medical image visualization," in Engineering in Medicine and Biology Society, 2005. IEEE-EMBS. 27th Annual International Conference of the, 2006, pp. 5145-5148.
[14] O. Kutter, R. Shams, and N. Navab, "Visualization and GPU-accelerated simulation of medical ultrasound from CT images," Computer methods and programs in biomedicine, vol. 94, 2006, pp. 250-266.
[15] B. Huang, J. Mielikainen, H. Oh, and H.-L. Allen Huang, "Development of a GPU-based high-performance radiative transfer model for the Infrared Atmospheric Sounding Interferometer (IASI)," Journal of Computational Physics, vol. 230, pp. 2207-2221, 2011.
[16] J. Mielikainen, B. Huang, and H.-L. Huang, "GPU-accelerated multi-profile radiative transfer model for the infrared atmospheric sounding interferometer," Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of, vol. 4, 2011, pp. 691-700.
[17] Liangchao Li, et al., "Parallel calculation for background infrared irradiation from aerial complex targets,"Systems Engineering, and Electronics, vol. 33, 2012, pp. 2573-2576.
[18] S. Lee and J. S. Vetter, "Early evaluation of directive-based gpu programming models for productive exascale computing," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, p. 23
[19] M. Lee, H. Jo, and D. H. Choi, "Towards high performance and usability programming model for heterogeneous HPC platforms," in Computing Technology and Information Management (ICCM), 2012 8th International Conference on, pp. 51-57.
[20] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs: Newnes, 2012.
[21] . M. Lee, H. Jo, and D. H. Choi, "Towards high performance and usability programming model for heterogeneous HPC platforms," in Computing Technology and Information Management (ICCM), 2012 8th International Conference on, pp. 51-57.
[22] "The OpenACC™ Application Programming Interface Version 1.0," 2011.
[23] CAPSCompilers-3.3_OpenHMPP_ReferenceManual," 2013.
[24] "CAPSCompilers-3.3_HMPPCG_ReferenceManual,"2013.