Korean Journal of Remote Sensing, Vol.28, No.6, 2012, pp.683~691
http://dx.doi.org/10.7780/kjrs.2012.28.6.8

Performance Study of Satellite Image Processing on Graphics Processors Unit Using CUDA

In-Kyu Jeong*, Min-Gee Hong*, Kwang-Soo Hahn, Joonsoo Choi** and Choen Kim*** †**

*Department of Applied Information Technology, Graduate School, Kookmin University
**Department of Computer Science, Kookmin University, ***College of Forest Science, Kookmin University

Abstract : High resolution satellite images are now widely used for a variety of mapping applications including photogrammetry, GIS data acquisition and visualization. As the spectral and spatial data size of satellite images increases, a greater processing power is needed to process the images. The solution of these problems is parallel systems. Parallel processing techniques have been developed for improving the performance of image processing along with the development of the computational power. However, conventional CPU-based parallel computing is often not good enough for the demand for computational speed to process the images. The GPU is a good candidate to achieve this goal. Recently GPUs are used in the field of highly complex processing including many loop operations such as mathematical transforms, ray tracing. In this study we proposed a technique for parallel processing of high resolution satellite images using GPU. We implemented a spectral radiometric processing algorithm on Landsat-7 ETM+ imagery using CUDA, a parallel computing architecture developed by NVIDIA for GPU. Also performance of the algorithm on GPU and CPU is compared.

Key Words : GPU, CUDA, Parallel Processing, High-Resolution Satellite Imagery

1. Introduction

Computational power and remotely sensing performance for data acquisition has been growing steadily during recent decades. Accordingly, the application of processing the acquired images was variously developed to be able to more easily access and process the data (Christophe and Inglada, 2009). As a result, researchers have been benefited from the quick processing of more data. But as the latest image processing techniques are becoming more complex, the amount of data grows more huge. These advanced tasks are forced to be processed more complicatedly, thus requiring more powerful computation. However, researchers have faced a limit on the financial aspect, needing a new solution.

The answer for these problems is parallel systems. As emphasized in a perceptive report from Berkeley,

Received October 23, 2012; Revised November 18, 2012; Accepted November 19, 2012.
† Corresponding Author: Choen Kim (choenkim@kookmin.ac.kr)

the improvement in computational power recently takes a parallel approach (Asanovíc *et al*., 2006; Christophe *et al*., 2011). Parallel processing techniques have been developed continuously for improving the performance of image processing along with the development of the computational power. However, entering the twenty-first century, semiconductor scaling limits, electrical power ceiling and exothermic problems have been in the way against the growth of single-core microprocessors.

Because of this, most semiconductor vendors turn instead to multicore-chip organizations. A case in actual-industry is that the computing vendor changed procedure in 2005 when Intel followed the lead of IBM's Power 4 and Sun Microsystems Niagara processor announced that its high performance microprocessors would henceforth rely on multiple processors or cores (Asanovíc *et al*., 2006). Multicore will obviously help program workloads which contain a mix of independent sequential tasks, even though the benefits of multicore can only be realized when the programmer or compiler explicitly parallelizes the software.

Recent tendency of Graphics Processing Unit (GPU) has gained great popularity in the field of high-performance processing for scientific and engineering applications such as image processing (Song *et al*., 2011). As shown simple example in Fig. 1, recently available high-end GPU has hundreds of computing cores meanwhile mainstream Central Processing Unit (CPU) has quad cores only. GPU-based desktop computer's advantage of low cost, compact size hardware, high memory bandwidth and high parallelism are what make this system an appealing alternative to massively parallel system made up of commodity CPUs (Lindholm *et al*., 2008). In this context, GPUs become attractive because they offer considerable material even for non-visual, general-purpose computations. In fact, GPUs are really multicore processors, with hundreds of processing units (Che *et al*., 2008). At the present stage, high-end graphics manufactures are connected to GPU library, i.e., NVIDIA's CUDA and ATI's STREAM for efficient parallel processing. In order to improve a performance of image processing offered by CPU, we can propose the parallelism by using GPU library.

When it appeared, the floating-point computing power has grown beyond the rate of Moore's Law (Owens *et al*., 2005), for example, the existing NVIDIA Geforce GTX 680 has a single-precision floating-point computing power up to 3.08 Terra FLoating-point Operations Per Second (TFLOPS). GPU for general-purpose computing has the following advantages (Zhao and Jhou, 2011). First, a large number of parallel cores, which can provide more computing resources than CPU. Compute Unified Device Architecture (CUDA) represents the coprocessor as a device that can run a large number of threads. Second, high memory bandwidth, which can get a higher data throughput than CPU. Third, little thread switching overhead, so a lot of threads can be quickly executed.

A good example for GPU parallelism programming language CUDA is given in NVIDIA. CUDA has remarkably increased programmability to escape from classical General-Purpose computing on Graphics Processing Units (GPGPU) way of transformation to graphics Application Programming Interface (API), for example Open Graphics Library (OpenGL) and DirectX (Kessenich *et al*., 2012). If one wants to overcome the previously mentioned limitations, one can run some code on GPU while developing applications, using the programming language. Our goal in this paper inspired by CUDA is to examine the effectiveness of CUDA as a tool to express parallel computation with performance characteristics on GPU. And we propose a new

Fig. 1. The GPU devotes more transistors to data processing (NVIDIA, 2012).

platform in this study for improving the speed of high-resolution satellite image data processing.

2. CUDA Overview

CUDA is both the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, Open Computing Language (OpenCL), DirectCompute, and other languages (NVIDIA, 2012). The researchers encode a single program that contains both the CPU (host) and the GPU (device) code.

These two parts are automatically separated and compiled by the CUDA compiler tool chain (NVIDIA, 2011). Researchers using CUDA allows writing device code in C++ functions called kernels. The kernel is disparate from a regular function in that it is executed by many GPU threads in a Single-Instruction Multiple-Data (SIMD) style. This style is called Single-Instruction Multiple-Threads (SIMT) that each thread executes the entire kernel once. Fig. 2 shows an example that performs serial code execution on the CPU while parallel code executes on the GPU. Each GPU thread is given a special thread ID that is accessible within the kernel, through the built-in variables *blockIdx* and *threadIdx* (Han and Abdelrahman, 2011).

Threads have access to varied GPU memories during execution to a kernel. Each thread can read and write, or either read or write its private registers and local memory. In addition, single-cycle access times, registered in the GPU memory hierarchy, are the fastest. In contrast, local memory in the GPU memory hierarchy is the slowest. Each thread block has its private shared memory. All threads have read and write access to the global memory, and read-only access to the constant memory and the texture memory (Han and Abdelrahman, 2011). Since GPU threads can't access the host memory, the data by a kernel must be copied in the above mentioned GPU memories before it is executed.

NVIDIA's Fermi architecture is designed to support both graphics and general purpose computing. Current Fermi architecture products can support up to trillions concurrent threads. Each streaming multiprocessor consists of 8 processing elements, called Stream Processors or SPs. Each warp of 32 threads operates in lockstep and these 32 threads are quad pumped on the 8 SPs. Multithreading is then achieved through a hardware thread scheduler in each SP. Every cycle of this scheduler selects the next warp to execute. Divergent threads are handled

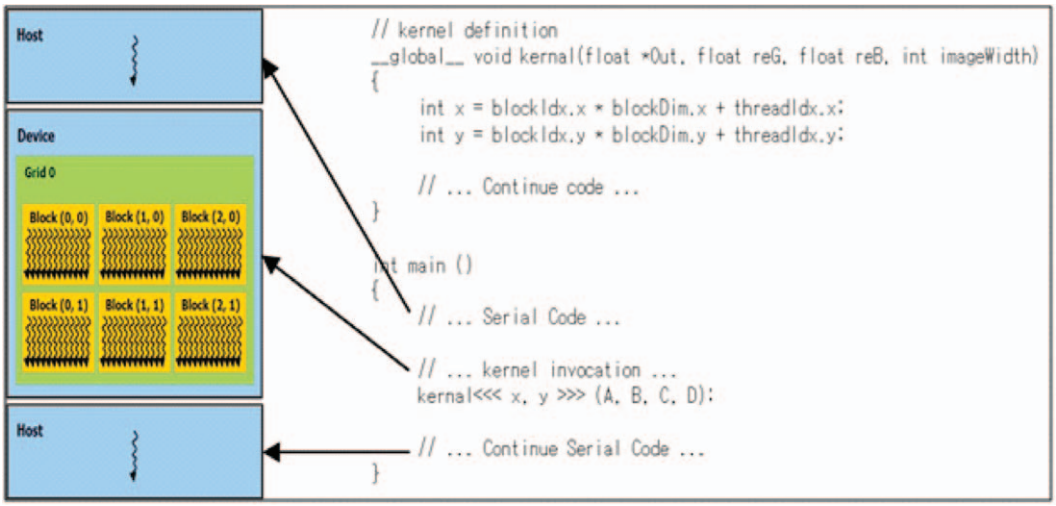

Fig. 2. The host code and kernel code in running process.

using hardware masking until they are reconverted.

Different warps in a thread block need not operate in lockstep, but if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously (Che *et al*., 2008). NVIDIA provides the occupancy calculator for the user-friendliness (NVIDIA, 2012a).

3. Methodology

This chapter is evaluating the degree of performance improvement for high-resolution image processing using GPU. In this study, we measured the processing speed for thread-block model and image size changing. Experiments for the optimization of the performance differences depending on the configuration of blocks and threads are needed due to the nature of parallel processing to use a GPU. We measure each processing speed changing the size of images.

In our experiment, we tested the speed of both the CPU and GPU versions of the conversion to Landsat-7 ETM+ spectral radiance calibration algorithm. During radiometric calibration, pixel values from raw, unprocessed imaged data are converted to units of absolute spectral radiance using 32-bit floating-point calculations (Markham *et al*., 2004). The absolute radiance values are scaled to 8-bit numbers representing before output to distribution media. Conversion from Q_{cal} in Level1 products back to ETM+ sensor spectral radiance (L_λ) requires knowledge of the lower and upper limit of the original rescaling factors. The following equation is used to perform the Q_{cal}-to-L_λ conversion for Level 1 products (Chander *et al*., 2009):

$$L_\lambda = \left(\frac{LMAX_\lambda - LMIN_\lambda}{Q_{cal\,max} - Q_{cal\,min}}\right)(Q_{cal} - Q_{cal\,min}) + LMIN_\lambda \quad (1)$$

Or

$$L_\lambda = G_{rescale} \times Q_{cal} + B_{rescale} \quad (2)$$

equation's element $G_{rescale}$ and $B_{rescale}$ can be expressed as follows:

$$G_{rescale} = \frac{LMAX_\lambda - LMIN_\lambda}{Q_{cal\,max} - Q_{cal\,min}} \quad (3)$$

and

$$B_{rescale} = LMIN_\lambda - \left(\frac{LMAX_\lambda - LMIN_\lambda}{Q_{cal\,max} - Q_{cal\,min}}\right) Q_{cal\,min} \quad (4)$$

The meaning of the equation elements and

Table 1. Meaning of conversion equation elements

Elements	Mean [measure unit]
L_λ	Spectral radiance at the sensor's aperture [W/(m^2 sr μm)]
Q_{cal}	Quantized calibrated pixel value [DN]
$Q_{cal\ min}$	Minimum quantized calibrated pixel value corresponding to $LMIN_\lambda$ [DN]
$Q_{cal\ max}$	Maximum quantized calibrated pixel value corresponding to $LMAX_\lambda$ [DN]
$LMIN_\lambda$	Spectral at-sensor radiance that is scaled to Q_(cal min) [W/(m^2 sr μm)]
$LMAX_\lambda$	Spectral at-sensor radiance that is scaled to Q_(cal max) [W/(m^2sr μm)]
$G_{rescale}$	Band-specific rescaling gain factor [W/(m^2 sr μm)/DN]
$B_{rescale}$	Band-specific rescaling bias factor [W/(m^2 sr μm)]

Table 2. Performance testing environment

	CPU	GPU
Product model	Intel Core i5-760	GTX 550Ti
Num of cores	4 cores / 4 threads	192 cores / 4MPs
Clock speed	3.20 GHz	0.98 GHz
Shared memory	N/A	48 KB per MP
Library	OpenCV 2.1	CUDA 4.2
Tool kit	Microsoft Visual Studio 2010 v10.0.40219.1 SP1Rel	

measure units are shown in Table 1. Other elements data to post-calibration dynamic ranges, and mean solar irradiance is given in Markham *et al*. (2004). Landsat-7 ETM+ sensor has a spatial resolution of 30 m for the six reflective bands, 60m for the thermal band, and includes a panchromatic band with a 15 m resolution.

If one wants to use the NVIDIA's GPU library, CUDA, the CPU host code and the GPU kernel code will be needed. Host code is responsible for the role of overall handling such as GPU variable declarations, initialization, and kernel code execution. In this paper, the kernel code is executed by the internal threads in the GPU at the same time for operations that convert the digital number values of the image processing. This system is called the kernel function in the host code using variable thread and block for parallel processing. In this study, we are tested performance using single core at CPU but using hundreds of Arithmetic and Logic Unit (ALU) at GPU for parallelization.

When remote sensing imagery is processed by calibration algorithm, the image region will be calculated by a kernel (above equation) in applicable channel. The procedure of computing one pixel in the target image is independent of computing other pixels. We enable one thread to compute one pixel of original image with CUDA. Assume that the original image has width x height pixels with channels, and the kernel is a template. In the procedure, the CUDA kernel (grid) is divided blocks and each block is composed of threads. Then each thread can be executed independently to compute the relevant pixel of target image. For given experiments in this paper, we first compare the performance of Open Computer Vision (OpenCV)-based CPU with that of CUDA-based GPU. We then measured the processing speed by changing thread block models and image sizes. The above algorithm performance was conducted to compare the timestamp method by using between cudaEvenElapsedTime API at CUDA and GetTickCount API at OpenCV. In addition, note that

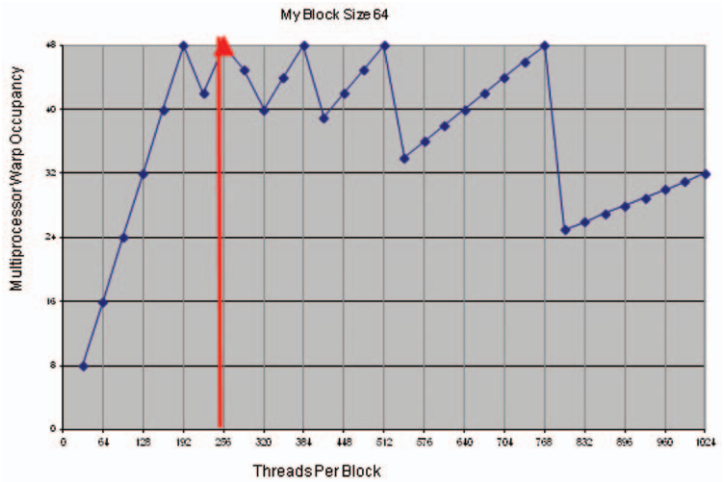

Fig. 3. The results of occupancy calculator running (NVIDIA, 2012a).

Table 3. The comparison of processing time (sec) according to image size and thread block model

	Structure		Processing time for each image size		
	Block	Thread	1,000 × 1,000	5,000 × 5,000	10,000 × 10,000
GPU	4	48	0.68	3.12	10.23
	16	96	0.58	2.87	9.86
	32	192	0.41	2.49	8.54
	48	192	0.42	2.53	8.74
	64	256	0.45	2.88	8.58
CPU	N/A	N/A	0.52	2.88	11.58

tested overall source codes are the same in both cases at CPU and GPU except for measuring performance of the above algorithm. Table 2 shows the development environment of our experiment.

4. Experimental Results

GTX 550Ti belongs to NVIDIA's mainstream graphics card, but it prominently improves processing speed. CUDA affects the performance, depending on the configuration of the thread and block. We need to consider observation of speed changes according to the configuration of thread and block to find the optimal combination of performance improvement. In this study, we were experiment through a two-step to find the optimal model. Firstly, we were performance comparison based on the optimal model found using occupancy-calculator provided by CUDA with CPU. Secondly, we were found optimal model through performance comparison by change of the model structure.

First, optimal thread block model was applied using the occupancy calculator provided by NVIDIA. Occupancy calculator running results were shown in Fig. 3 that the marker (see red arrow) was the optimal model as the number of 256 threads per block. Comparison of the computation speed of the CPU and CUDA depend on optimal model is as Fig. 4. At this time, the size of the data has changed to compare the change in the amount of data processing speed increase. A kernel function has only simple arithmetic

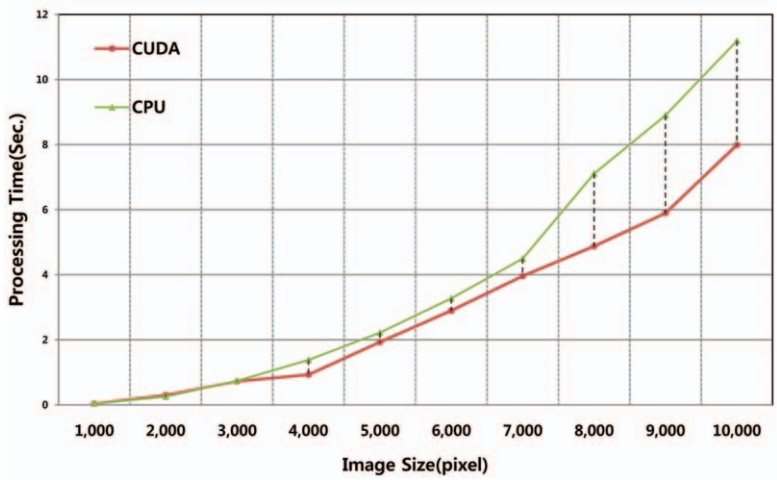

Fig. 4. Processing time comparison between CPU and CUDA according to image size.

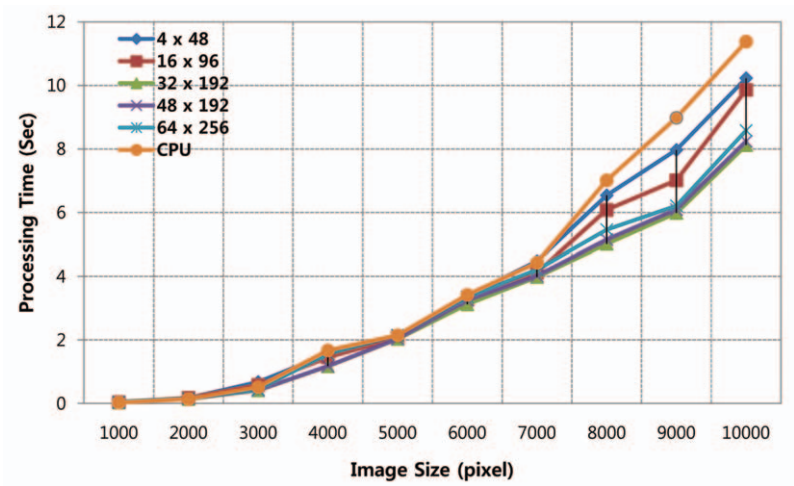

Fig. 5. Processing time comparison between CPU and CUDA depending on thread-block structure.

operations handling the pixel value. The GPU parallelization method can be applied to remotely sensed data to improve processing performance.

Second, Table 3 shows the results for the processing time of the CPU and GPU depending on thread-block model. The processing time in Table 3 is the average of 100 independent runs. Experiments are carried out by increasing the size of the image by 1,000 from 1,000x1,000 to 10,000x10,000. There is a difference in speed depending on the configuration of the block and threads. Therefore blocks and threads configuration are very important. In this study configuration of block and threads are called one-dimensional block and thread model. Parallelization is simple and with the size of the data is small, but showed performance improvement of about 10% better than based on CPU.

5. Conclusion

To process high resolution imagery using general methods on standard computing environment is

difficult because of very large capacity. The capacity of satellite images (several gigabytes) makes processing by usual methods inapplicable on standard research environment. It is not helpful or worthwhile to copy the entire image into memory before doing any processing. In this situation, it is necessary to copy only part of the image and process it before saving the result to the disk and proceeding to the next part. This process relies on the capability of each algorithm to determine the size of the input needed to produce the output required by the downstream process (Christophe *et al*., 2011).

We must perform various calibrations to acquire information. If one wants the repeatedly same-operations for each pixel, one can perform a parallel processing method for multiple data. In this paper, we proposed an efficient parallel processing using CUDA. Now we mostly use as general-purpose high-performance graphics cards on the PC. Therefore a parallel processing by GPU is an important to improve the efficiency of data processing dramatically. Of course, a single-CPU is a current trend that supports a parallel processing to configure more than 4 ALU. However, CPU has lower performance compared to GPU which has more than hundreds ALU specialized in floating-point arithmetic. This allows for adapting GPU to be executed in order to improve image processing, instead of the most expensive part (namely, CPU) in a processing unit. With a minimum investment (hardware cost is around US$200, the software used here is free and open source), performance gains can attain on the critical portion of the processing.

The data transfer rate lower than processing speed was one of the old problems of parallel system, and the problem still exists in a GPU parallel system. Simultaneously using multiple processors can handle large amounts of data and, of course, are faster. However it is difficult to achieve the desired speedup due to data bottlenecks. In this paper, we conducted experiments changing the threads-block model in order to minimize these problems. In this study, it didn't show a big speed improvement compared with the C programming because processing performed by the GPU kernel function is relatively simple. However, it is clearly demonstrated the speed improvements.

As mentioned above, the GPU parallelization of CUDA is more efficient to overcoming repeatability of the work than the complexity of algorithm. However, the advantage from this massive parallelization requires one to carefully select those algorithms which fit well in the GPU computing architecture, identify the critical sections to optimize, and look closely at how things are implemented. Also, we can be made in that direction by proposing a mechanism to switch seamlessly from CPU to GPU versions of algorithms depending on available hardware. In the future, a meaningful research for optimizing the performance of CUDA will be considered.

Acknowledgements

The authors would like to thank the two reviewers for their valuable comments and their helpful contribution in improving this article. Page charges of the manuscript are supported by the Industry-Academic Cooperation Foundation, Kookmin University.

References

Asanovíc, K., R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick,

2006. The landscape of parallel computing research: A view From Berkeley, Tech. Rep. No. UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Dec. 18, 2006, p. 54.

Chander, G., B.L. Markham, and D.L. Helder, 2009. Summary of current radiometric calibration coefficients for Landsat MSS, TM, ETM+, and EO-1 ALI sensors, *Remote Sensing of Environment*, 113: 893-903.

Che, S., M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron, 2008. A performance study of general-purpose applications on graphics processors using CUDA, *Journal of Parallel and Distributed Computing*, 68: 1370-1380.

Christophe, E. and J. Inglada, 2009. Open source remote sensing: Increasing the usability of cutting-edge algorithms, *IEEE Geoscience Remote Sensing Newsletter*, 9-15.

Christophe, E., J. Michel, and J. Inglada, 2011. Remote sensing processing: From multicore to GPU, *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(3): 643-652.

Han, T.D. and T.S. Abdelrahman, 2011. hiCUDA: High-level GPGPU programming, *IEEE Transactions on Parallel and Distributed Systems*, 22(1): 78-90.

Kessenich, J., D. Baldwin, and R. Rost, 2012. The OpenGL shading language, *http://www.opengl.org/documentation/glsl*.

Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym, 2008. NVIDIA Tesla: A unified graphics and computing architecture, *IEEE Micro*, 28(2): 39-55.

Markham, B.L., K. Thome, J. Barsi, E. Kaita, D. Helder, J. Barker, and P. Scaramuzza, 2004. Landsat-7 ETM+ On-orbit reflective-band radiometric stability and absolute calibration, *IEEE Transactions on Geoscience and Remote Sensing*, 43: 2810-2820.

NVIDIA, 2011. The CUDA Compiler Driver NVCC V4.1, *http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/nvcc.pdf*.

NVIDIA, 2012. NVIDIA CUDA C Programming Guide v4.2, *http://developer.download.nvidia.com /compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf*.

NVIDIA, 2012a. CUDA Occupancy Calculator, *http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation*.

Owens, J.D., D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell, 2005. *A Survey of General-Purpose Computation on Graphics Hardware*, Eurographics 2005, State of the Art Reports, 21-51.

Song, C., Y. Li, and B. Huang, 2011. A GPU-accelerated wavelet decompression system with SPIHT and Reed-Solomon decoding for satellite images, *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 4(3): 683-690.

Zhao, J. and H. Jhou, 2011. Design and optimization of remote sensing image fusion parallel algorithms based on CPU-GPU heterogeneous platforms, *IEEE International Congress on Image and Signal Processing*, Shanghai, China, Oct. 2011, 1623-1627.