Optimization of Two Bottleneck Programs in SAR System on GPGPU

Yang Zhang(✉), Zuocheng Xing, Cang Liu, Chuan Tang, Lirui Chen, and Qinglin Wang

National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, China
{zhangyang,zcxing,liucang,tc8831,chenlirui14}@nudt.edu.cn, wangqinglin.thu@gmail.com

Abstract. The Synthetic Aperture Radar (SAR) system is a kind of modern high-resolution microwave imaging radar used in all-weather and all day long to provide remote sensing means and generate high resolution images of the land under illumination of radar beam. Unlike optical sensors, SAR algorithm needs a post-processing process on the data acquired to form the final image. In this article, we use the General Purpose Graphic Processing Units (GPGPU) to accelerate two of SAR algorithms, PGA (Phase Gradient Autofocus) and PDE (Partial Differential Equations), which are two computational intensive algorithms in the post-processing process for the system. Our work shows that the GPU architecture has different acceleration effects on the two algorithms. PGA can achieve an acceleration of 21.7% and PDE can get a speed up of 2.58× on GPGPU. We analyse the reasons for the results and conclude that GPU is a promising platform to accelerate the SAR system.

Keywords: SAR system · PGA · PDE · GPU · Acceleration

1 Introduction

Synthetic aperture radar (SAR) is a kind of modern high-resolution microwave imaging radar used in all-weather and all day long. It uses the principle of synthetic aperture, pulse compression technology and the method of signal processing. And make use of real aperture antenna to obtain distance and azimuth bidirectional high resolution remote sensing imaging. Unlike optical (Landsat) data, the SAR data requires extensive two-dimensional, space variant signal processing before an image is formed [1]. As we all know, it occupies an absolutely important position in imaging radars. With the widespread application of SAR, more attentions have been increasingly paid on its image post-processing technology because of its complicated process. In the SAR image post-processing, PGA algorithm and PDE algorithm are two key algorithms that are always the bottleneck operations. Our goal is to optimize the two algorithms and relieve the bottlenecks.

W. Xu et al. (Eds.): NCCET 2016, CCIS 666, pp. 115–124, 2016.
DOI: 10.1007/978-981-10-3159-5_11

GPUs is well-known as its powerful computing capability and high energy efficiency. It is originally designed to exploit the concurrency inherent in graphics workloads. Now it has been widely used in high performance computing (HPC) platform as an accelerator [2–4]. As we known, it is featured with thousands of processors and few control units. For example, state-of-the-art Maxwell architecture has up to 3072 cuda cores and few megabytes on-chip memory [5]. The massive cores are used to tolerate long memory latency and to deliver high throughput.

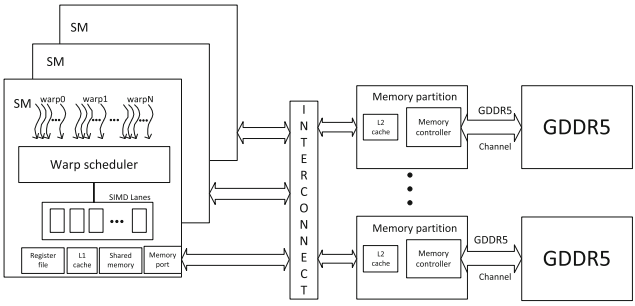

Fig. 1. Baseline architecture of our GPU.

The emerge of CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) programming models makes GPUs an easy platform for computing and boost its further development [6,7]. In CUDA, threads take a hierarchical structure, that is many threads are grouped into a thread block and several thread blocks constitute a thread grid. A large workload can be divided into several blocks which are further divided into a number of threads. A warp is made up of 32 threads and its threads execute in a lockstep style. Several warps are scheduled by warp scheduler for high performance. The memory also has a hierarchical structure, which includes on-chip memory such as register file, L1 cache, shared memory, L2 cache and off-chip memory such as global memory. Among them, global memory is a large but slow memory where data between different stream multi-processors (SM) can communicate and shared memory stores data which belong to the same thread block. Threads in a block run in an SIMT (Single Instruction Multiple Threads) manner and they can synchronize among themselves through barriers. Figure 1 is the baseline architecture of our GPU.

ArrayFire is a software platform which is developed by AccelerEyes for users and programmers to quickly develop data parallel programs in C, C++, Fortran, and Python [8]. ArrayFire provides simple high-level functions instead of low-level GPU APIs such as CUDA, OpenCL and OpenGL to allow scientists, sociologists and economists to take full advantage of the computation ability of GPU. Combined with friendly interface to users, automatic memory management, real-time compilation, parallel loop structure for GPU,

and hardware-accelerated graphic library interface, ArrayFire becomes ideal for rapid parallel algorithm prototyping and end to end application establish. We use ArrayFire to accelerate PDE algorithm for noise reduction processing.

[1] has described the signal processing operations in a digital processor which has been built to produce images from the Seasat-A SAR data. [9] presents a new frequency scaling processing algorithm for spotlight SAR data processing. In [10], the two-dimensional exact transfer function (ETF) is calculated and range-variant phase corrections have been calculated in order to process many azimuth lines per block. [11] introduces a novel and efficient Synthetic Aperture Radar processor. The previous works mainly focus on the SAR algorithm itself. As a contrast, we put an emphasis on the simple implementation of SAR algorithm on GPU and the optimization of their performance.

The remainder of this paper is organized as follows: In Sect. 2, the CUDA programming model and arrayfire are described. Sect. 3 proposes the implementation of the two algorithms in GPU. The simulation result is presented in Sect. 4. Section 5 provides some conclusions.

2 CUDA Programming Model and Arrayfire

We use CUDA and arrayfire to optimize the programs. CUDA is a popular software model to develop applications for GPU and arrayfire is a new software which is based on CUDA to develop GPU program in an easier and faster way.

2.1 CUDA Programming Model

In 2007, Nvidia released the CUDA architecture [6]. Since then, CUDA is a mainstream of parallel programming model that is used by many fields in parallel computing. Because the majority of GPU resources are used for calculation and the execution pattern meets the SIMT model, it uses SIMT model for the development of highly parallel programs. CUDA makes GPU an easier and more powerful programming platform for parallel computing over CPU.

CUDA is a software programming model which allows programmers to exploit the parallel potential in GPU. It is a minimal extension to C and C++ and has few instructions but very fast execution. It adopts the single instruction multiple data mode. As Fig. 2 shows, kernels execute over a set of parallel threads and threads are organized in a hierarchy structure. Each block can have up to 3 dimensions and contains up to thousands of threads. Threads within one block can share the shared memory and synchronize. If the memory bandwidth is not enough, throughput will be limited by the memory copy process from CPU to GPU or vice versa. The fast on-chip resources such as registers, shared memory and constant memory can be used to reduce the off-chip memory access. Shared memory can store data on chip to reduce memory access time. By allowing data reuse between data blocks and reducing bank conflicts in shared memory, there will be less long-latency global memory accesses. In addition, there are two restrictions. Firstly, one block can only use the shared memory within one SM

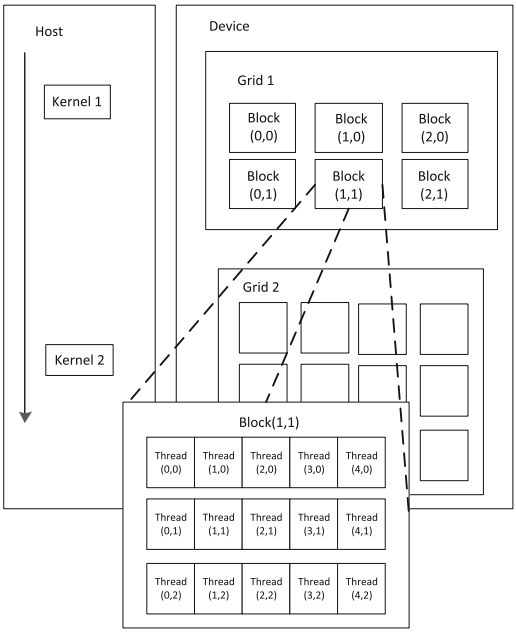

Fig. 2. CUDA programming model.

and shared memory can't be shared by blocks in different SMs. Secondly, the capacity of shared memory in each SM is limited and it is divided by multiple blocks. Thus, using excessive shared memory may reduce the number of concurrent blocks mapped onto one SM. Therefore, reasonable distribution of work over multiple cores to reduce long latency is a challenging work.

2.2 Arrayfire

Over the past decades, GPU has been more and more common over consumers and computer developers. Despite of the growing number of successful projects, the GPU software number increases slowly. That is mainly due to the difficulties in GPU programming. In the early times, the advent of CG, GLSL, HLSL and Brook stream programming marked the beginning of stream programming. They are the precursors of GPU programming. But the calculations of them need to be mapped to the graphics pipeline, which restricts their applications. After that, CUDA and OpenCL introduced a more general programmable software architecture which is easier than stream programming. However, even with these advances, it is still difficult for the average programmer to learn CUDA and OpenCL, because they are more difficult than the standard single-threaded C and C++ programming.

Some former companies are also trying to achieve this goal. One of the first such companies, Peak-Stream, has built a C/C++ runtime library functions to

provide GPU developers with a rich set of tools. RapidMind has developed a flexible intermediate layer to support a variety of front-end languages and back-end hardware. These are attempts to bridge the gap between hardware and software developers. ArrayFire is the newest development platform for GPU in an attempt to bridge this gap and transplant high-level functions to the bottom hardware. Our goal is to provide more programmability and portability for the program without sacrificing performance.

3 Implementation

To accelerate the two algorithms, we need to make use of the inherent parallelism and the different memory architecture in GPU. The parallelism in GPU is a kind of lightweight parallelism and it can be assigned by programmer. When a loop is encountered in the program, we can explicitly dispatch threads as many as possible to execute the loop concurrently. For the memory part, we mainly use two kinds of memories those are the shared memory and the global memory. Global memory is persistent across kernel launches by the same application. Data stored in the global memory is also the only way to perform data exchange between different SMs. For comparison, shared memory has much higher bandwidth and much lower latency than global memory. Programmer can directly allocate the capacity of shared memory in one SM. It can be shared by multiple threads in one block but not between multiple blocks because the life time of shared memory is as long as the block's.

3.1 PGA

We use Matlab program to finish pre-simulation and acquire the results of the two algorithms. In order to carry out GPU acceleration, we first make an appropriate conversion from Matlab program to C language program. Since Matlab is a more abstract high-level language, its invoked libraries can not be reused in CUDA. So to make the programs more efficient, we rewrite and add several functions in C language from the Matlab library functions. In this process, we ensure functional coherence between C programs and Matlab programs, while maintaining the programs' correctness through the conversion. The redesigned functions in PGA are as follows.

- Calculation functions: variance computation cov(), averaging computation mean(), circulate and shift computation circshift_real(), and the sort of array.
- Transform functions: fast Fourier transformation and fast inverse-Fourier transformation.
- Fitting functions: binomial curve fitting.

We revise and optimize the C implementation of PGA and make sure the correctness and the high efficiency. The optimization process is as follows.

- We isolate the functions which have many computations and are easy to parallel for acceleration on GPU.

– We set up local and global variable carefully. To reduce data transmission of kernel functions, we take local variable as much as possible.
– Taking the effect of computer architecture to the program's efficiency into consideration, we optimize time locality and space locality to enhance IPC (Instruction Per Cycle).

We put the optimized C programs onto GPU to make full use of its powerful computing capability and efficient storage characteristics. The CUDA code first makes a serial execution on the host CPU side. When encountering the kernel function, the code will be loaded onto the GPU for execution. By specifying the amount of threads and blocks in the kernel, we can parallelize the data process explicitly. When the kernel is finished, the data will be copied from GPU to the CPU side, where the serial execution continues. Therefore, the placement method of data and the parallelism optimization are particularly important. So we take the following steps to finish the GPU implementation.

– We migrate the C language program without parallelism directly to Nvidia's CUDA programming platform to make the program run on the CPU-GPU heterogeneous platforms, following CUDA programming syntax rules and various API interfaces. In this way, this part of program still runs on the CPU and the cost of data movement between CPU and GPU is reduced.
– When conducting fast Fourier transform, a one-dimensional plan, namely cufftHandle plan, is created, followed by a Fourier transform cufftExecC2C. CUFFT_FORWARD, as a function parameter, shows the Fourier transform is performed, while CUFFT_INVERSE indicates an inverse Fourier transform is performed.
– Use the multi-threaded feature of GPU to accelerate Fourier transform parts in the program.
– Use a variety of memory hierarchy, such as global memory, shared memory and etc., to reduce data access time and optimize memory access efficiency.

We have attempted to put all functions (the whole PGA program) into the GPU for execution. However we find the execution efficiency is low. That is because the program flow is too complicated and the too many functions with a number of parameters further complicate the program. Since each function call and access to local variables involves function stack operation, so it is easy to cause stack overflow. Although the algorithm has some parallelism, i.e., the image data can be processed in parallel blocks, in the realization of the program, we find because this process is a complex serial procedure and each block contains a number of function calls, the algorithm's parallel process is not very suitable for the GPU's lightweight thread feature.

Since we can not put all the functions in GPU, we adopt a compromise approach to place the most complex computational process, FFT and IFFT transformation, onto GPU for acceleration. Each time the algorithm needs FFT and IFFT transformation, data are moved from CPU to GPU through cudaMemcpy() and it calls the cufft library for transformation. After the calculation is completed, the results are moved from GPU to CPU by cudaMemcpy() to take the next step of the calculation. The whole process is illustrated as Fig. 3.

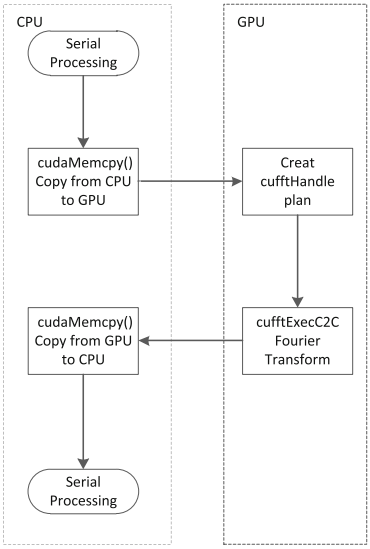

Fig. 3. Execution process of PGA.

3.2 PDE

To accelerate PDE, we use open source software to accelerate the critical parts in it. PDE contains input matrix transformation and regularization, multiple iterations of arithmetic operations and two-dimensional convolution calculation. Since the input matrix transformation, regularization and the matrix operation are not the bottlenecks, and to avoid the redundant overhead of data transfer between CPU and GPU, these operations are carried out in CPU. Two-dimensional convolution operation takes a long time and is done several iterations, so we put it to the GPU to accelerate its process and reduce latency.

Arrayfire comes with library functions to optimize two-dimensional convolution operation. First, we need to call the header file of arrayfire.h. After the procedure calls the function of convolve2(), the underlying functions will finish the copy of data to the GPU and the two-dimensional convolution operation. The convolution process includes the development of parallelism and the optimization of storage. After the calculation is completed, the results will be passed to the CPU for the following index calculation, multiplication and division.

4 Throughput Performance and Analysis

We take an image of 16k * 4k as the input of PGA and compare the results before and after acceleration. The CPU and GPU configurations are in Table 1 and the two platforms' performance are comparable. The results show that the time spent in GPU is 48.345 s, in contrast to 58.84 s on CPU. GPU can get 21.7% improvement over the CPU implementation.

Table 1. Configuration of the Platform

CPU	Intel core i5	3.4 GHz	8G DDR4
GPU	(GTX660Ti)	(1006/1084 MHz)	2G GDDR5

Although we take parallelized method for PGA processing and get good performance with less efforts, it still cannot get a considerable speed-up ratio beyond the serial implementation as we expected. The main reason is that, although the entire program has several pieces to parallelize, each piece is a complex procedure that contains a number of serial function calls. The program data can be divided into pieces for parallel process but the computational process is hard to parallelize. Therefore PGA algorithm can only get limited improvement (Fig. 4).

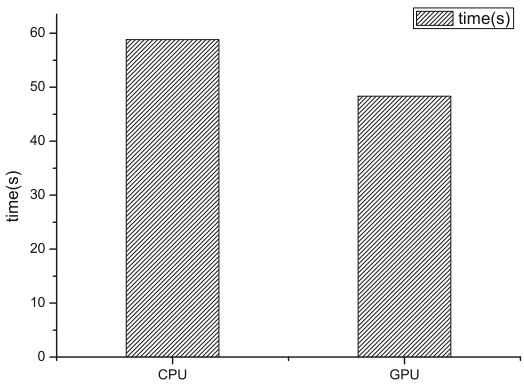

Fig. 4. The acceleration effect of PGA on GPU.

However we can get further improvement through the following method. One solution is to optimize the image processing algorithm to reduce the number of function calls, making it more suitable for GPU's characteristics. Another solution is to use other platforms for acceleration, such as MIC, which has more control logic to handle complex procedures (Fig. 5).

The input data of PDE is an image of 12k * 4k and it runs on a computer with a configuration as Table 1 shows. The running results of PDE program show that the time spent on CPU is 49.21 s, while the time spent on GPU is 19.072 s. We can achieve an acceleration of 2.58×.

It can be seen that, since the two-dimensional convolution has high parallelism inherently, we can take the parallel method to get the desired speed-up ratio. The main reason is that the algorithm itself has a high degree of parallelism, and each thread has a relatively simple calculation process. Meanwhile, there are no bottleneck operations and critical paths. This massive parallelism is suitable for parallel processing, and this also makes the optimization of on-chip

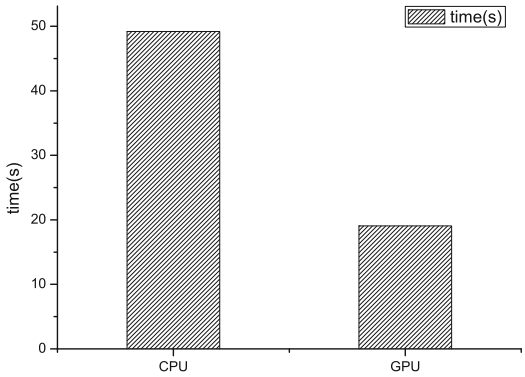

Fig. 5. The acceleration effect of PDE on GPU.

data storage possible. Thus PDE algorithm is very suitable for the acceleration on GPU platform.

5 Conclusion

In this article, we use the General Purpose Graphic Processing Units (GPGPU) to accelerate two of SAR algorithms, PGA and PDE, which are two bottleneck algorithms in the post-processing process for the system. We first inspect the characteristics of the two algorithms, those are their computing process, the parallelism of the algorithms and the memory access patterns. Then, we take distinct methods to accelerate the two algorithms. One way is to use CUDA library functions to simplify the process and to optimize the throughput of PGA, the second way is to use open source software to accelerate the critical parts of PDE. The two methods adapt to different characteristics of the two algorithms and both achieve good performance. So our work show that GPU is a promising accelerator for the SAR system. Our next work is to put the whole SAR system on GPU and to achieve better performance.

Acknowledgments. This work is supported by National Science Foundation of China (Grant No. 61170083, 61373032) and Specialized Research Fund for the Doctoral Program of Higher Education (Grant No. 20114307110001).

References

1. Cumming, I.C., Bennett, J.R.: Digital processing of SEASAT SAR data. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 1979, vol. 4. IEEE, pp. 710–718 (1979)
2. http://www.top.500.org
3. http://www.green500.org

4. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proc. IEEE **96**(5), 879–899 (2008)
5. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x
6. Nvidia: Nvidia Cuda C programming guide v7.5 (2015). http://developer.nvidia.com/nvidia-gpu-computing-documentation
7. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, Reading (2010)
8. Malcolm, J., Yalamanchili, P., McClanahan, C., Venugopalakrishnan, V., Patel, K., Melonakos, J., Arrayfire: a GPU acceleration platform. In: SPIE Defense, Security, and Sensing, p. 84 030A. International Society for Optics and Photonics (2012)
9. Mittermayer, J., Moreira, A., Loffeld, O.: Spotlight SAR data processing using the frequency scaling algorithm. IEEE Trans. Geosci. Remote Sens. **37**(5), 2198–2214 (1999)
10. Eldhuset, K.: A new fourth-order processing algorithm for spaceborne SAR. IEEE Trans. Aerosp. Electron. Syst. **34**(3), 824–835 (1998)
11. Liu, B., Wang, K., Liu, X., Yu, W.: An efficient SAR processor based on GPU via CUDA. In: 2nd International Congress on Image and Signal Processing, CISP 2009, pp. 1–5. IEEE (2009)