GPU-based Parallel Implementation of SAR Imaging

Xingxing Jin
Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Canada
xing.jin@usask.ca

Seok-Bum Ko
Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Canada
seokbum.ko@usask.ca

Abstract— **Synthetic Aperture Radar (SAR) is an all-weather remote sensing technology and occupies a great position in disaster observation and geological mapping. The main challenge for SAR processing is the huge volume of raw data, which demands tremendous computation. This limits the utilization of SAR, especially for real-time applications. On the other hand, recent developments in Graphics Processing Unit (GPU) technology, which obtain general processing capability, high parallel computation performance, and ultra wide memory bandwidth, offer a novel method for computationally intensive applications. This work proposes a parallel implementation of SAR imaging on GPU via Compute Unified Device Architecture (CUDA), and provides a potential solution for SAR real-time processing. The results show that the proposed method obtained a speedup of 31.72, compared to a CPU platform.**

Keywords- SAR, parallel computation, GPU, CUDA

I. INTRODUCTION

Synthetic Aperture Radar (SAR) is a remote sensing technique for obtaining high resolution images of the earth's surface [1]. As a radar sensor, SAR provides an all-weather remote sensing means and plays a great role in disaster observation, geological mapping, and strategic surveillance of military targets.

One of the difficult problems with SAR is the tremendous amount of signal processing to form a final image from raw echo data. The remote sensing community and the space agencies spend yearly a considerable amount of time and money to implement efficient and accurate processors for SAR data [2]. Even worse, the volume of actual SAR data is increasing rapidly, as are the demands of real time processing. This makes SAR processing systems more complicated to design and build, resulting in larger cost.

Recent years, many-core concepts and platforms draw a lot of attention. The switch to parallel microprocessors is a milestone in the history of computing [3]. Particularly, in November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture, leveraging the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU [4].

In this work, a GPU-based parallel implementation of SAR imaging is proposed, which provides a promising solution for real time SAR processing. The general methodology that accelerates complex tasks with GPU is demonstrated, with the process of GPU-based SAR implementation, as well as CUDA design and optimization techniques, including SIMT, concurrent transfer and compute, coalesced access, and free bank conflict access.

This work is organized as follows: Section 2 provides some background first and then gives the implementation and optimization of SAR imaging on GPU. Section 3 discusses the results in detail. Conclusions are left to Section 4.

II. IMPLEMENTATION OF SAR IMAGING ON GPU

GPU is flexible to program using high level languages and APIs, however, due to their throughput-oriented design, GPU requires much more emphasis on parallelism and scalability than CPU, which is very challenging. For those applications that have plenty of data level parallelism and the data can be processed independently on different processing elements for a similar set of operations such as filtering, aggregating, ranking, GPU can give more remarkable performance [5, 6]. The range-Doppler (RD) algorithm belongs to this type of programs.

RD algorithm is the most widely used method for digital processing of SAR. Figure 1 shows the block diagram of RD algorithm. The detailed implementation of the main blocks will be explained in the next part, following the order of the flow chart.

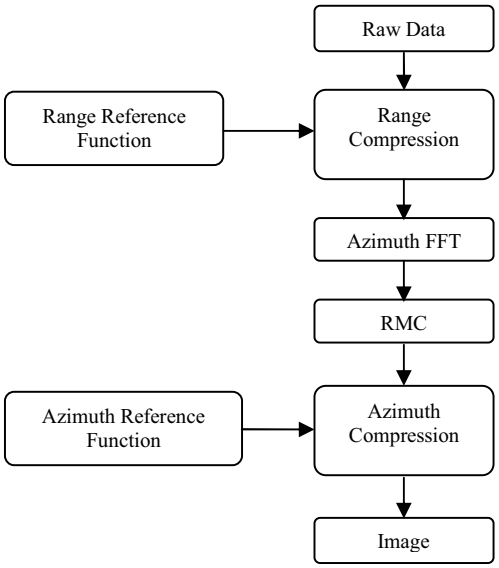

Figure 1. The block diagram of RD algorithm.

978-0-7695-4902-6/12 $26.00 © 2012 IEEE
DOI 10.1109/ISED.2012.35

A. Raw data transfer and reference function generation

First, the huge amount of raw data needs to be transferred from host memory to device memory. To obtain the highest bandwidth between host and device, CUDA provides a runtime API *cudaMallocHost()* to allocate pinned memory [7]. For PCIe×16 Gen2 GPU cards, the obtainable transfer rate could be more than 5 GBps.

Reference function is generated based on the sent signals and system parameters. Its phase-frequency characteristics satisfy the conjugate match condition with that of the sent signals.

CUDA provides a technique: asynchronous transfers, which can enable overlap of data transfer and computation on devices. It demands to replace blocking transfer API *cudaMemcpy()* with non-blocking variant *cudaMemcpyAsync(),* in which control is returned immediately to the host. Besides, the asynchronous transfer version contains an additional argument: stream ID. A stream is simply a sequence of operations that are performed in order on the device. To achieve concurrence of data transfer and device computation, they must use different non-default streams. The codes below demonstrate how the concurrence is performed.

CUDA codes for concurrence of data transfer and device computation

```
1. cudaStreamCreate(&stream1);
2. cudaStreamCreate(&stream2);
3. cudaMemcpyAsync(Echo_D,Echo_H,size,
cudaMemcpyHostToDevice,stream1);
4. RefGenerate
<<<Blocks_1D,Threads_1D,0,stream2>>>(Ref);
5. cufftSetStream(plan1, stream2);
6. cufftExecC2C(plan1, Ref, Ref,
CUFFT_FORWARD);
7. RefGenerate_
Conj<<<Blocks_1D,Threads_1D,0,stream2>>>(R
ef);
```

Data transfer is executed by stream 1 (line 3) and reference function generation including FFTs is performed by stream 2 (lines 4-7). Figure 2 depicts the difference of sequential and concurrent modes and we can see much time is saved from the overlapping streams.

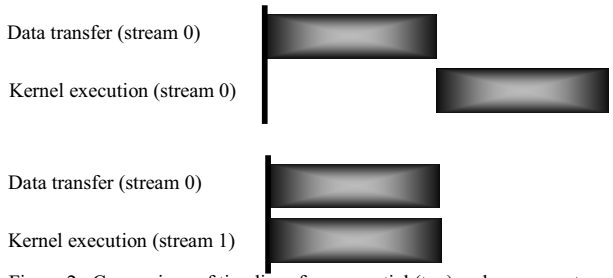

Figure 2. Comparison of timelines for sequential (top) and concurrent (bottom) data transfer and reference function generation.

B. Range Conpression

Range compression essentially is a matched filter through multiplying the range transformed data by the range reference function, which is same for all azimuth cells and can be multiple-used efficiently. This block is executed in parallel by a kernel in which each thread reads an element from Fourier transformed raw data and a corresponding element from range reference function, and then applies multiplication to them and save product to the global memory.

In particular, access to the global memory needs to be paid careful attention. The global memory is organized into aligned segments of 16 and 32 words. The global memory access by threads of a half warp is coalesced by the device. So, for a certain access pattern, if all the access addresses of threads within a half warp are within the same global memory segment, then one single 16 words transaction is enough. Otherwise, a wasted 32 words transaction or one 16 words transaction plus one 8 words transaction occurs, which will greatly degrade the bandwidth efficiency. The simplest coalesced access pattern is: the k-th thread access the k-th word in a segment, which is implemented in the excerpt of the range compression kernel.

We can see the coalesced access to the global memory (lines 6-9), as well as the usage of the shared memory aiming to store intermediate values and reduce communication costs.

Kernel codes for range compression

```
1. __global__ void RangeCompress(Complex*
InPut,Complex* RangeRef, Complex* OutPut){
2. __shared__float Dre[ThreadNumberPerBlock];
3. __shared__float Dim[ThreadNumberPerBlock];
4. unsigned int xIndex = blockIdx.x *
ThreadNumberPerBlock + threadIdx.x;
5. unsigned int xIndex_Ref = (xIndex % Num_Range);
6. Dre[threadIdx.x] = InPut[xIndex].x * RangeRef
[xIndex_Ref].x - InPut[xIndex].y *
RangeRef[xIndex_Ref].y;
7. Dim[threadIdx.x] = InPut[xIndex].x *
RangeRef[xIndex_Ref].y + InPut[xIndex].y *
RangeRef[xIndex_Ref].x;
8. OutPut[xIndex].x = Dre[threadIdx.x];
9. OutPut[xIndex].y = Dim[threadIdx.x];}
```

C. Tranpose

Matrix transpose is essential in RD algorithm, and it will consume a lot of time if the transpose is not handled properly. Since the amount of SAR data is huge, it is necessary to rearrange the data in the physical memory to facilitate further processing, instead of just exchanging the index of all elements in the matrix.

To efficiently implement transpose, coalescing data access to global memory and avoiding shared memory bank conflicts are the keys.

Here are some size specifics: the size of the complex matrix is 4096×4096, and it is divided into many tiles. The dimension of each tile is 16×16, same size as each thread block. Each block transposes a tile in parallel.

All loads from raw matrix are coalesced because each half-warp reads 32 contiguous words (a row of a tile) and one single 128-byte transaction is sufficient. When writing to global memory, each warp writes a row of a tile to different segments of global memory, and results in 16 separate memory transactions. One solution is to read the data into shared memory, and have each half warp access noncontiguous locations in shared memory in order to write contiguous data to global memory. There is no performance penalty for noncontiguous access patterns in shared memory, while there is huge penalty in global memory.

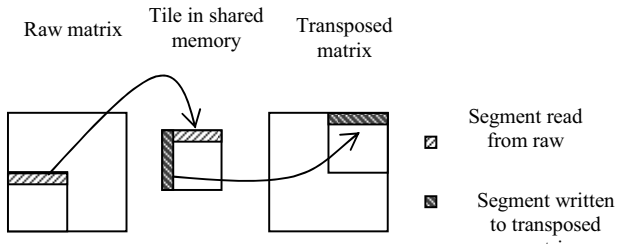

Figure 3. Coalesced matrix transpose.

The half warp writes one row of the raw matrix tile to the shared memory tile indicated by the brighter segments, as shown in figure 3. After all writings to tile are finished, the half warp writes one column of tile to one row of the transposed matrix, indicated by the darker segments, which are coalesced.

A shared memory access is another critical aspect of parallel transpose design. To get high bandwidth through concurrent accesses, shared memory is broken into 16 equally sized segments. Within a half warp, if more than one thread access the same bank of shared memory, then a bank conflict occurs and hardware will split the memory access into as many sequential and separate conflict-free accesses as necessary [8].

The coalesced transpose uses a 16×16 shared memory of complex, then all elements in one column are mapped to the same bank, indicated by the brighter blocks in figure 4 (The top left block is covered by gray blocks and dark blocks at the same time). As a result, when writing a column from tile in shared memory to a row in global memory, the half warp meets a 16-way bank conflict. The solution this work adopted is to pad the shared memory tile by one column:

__shared__ Complex Tile[Dim_block][Dim_block+1];

Then a 16-way bank conflict is changed to a 2-way bank conflict, indicated by the darker blocks in figure 4.

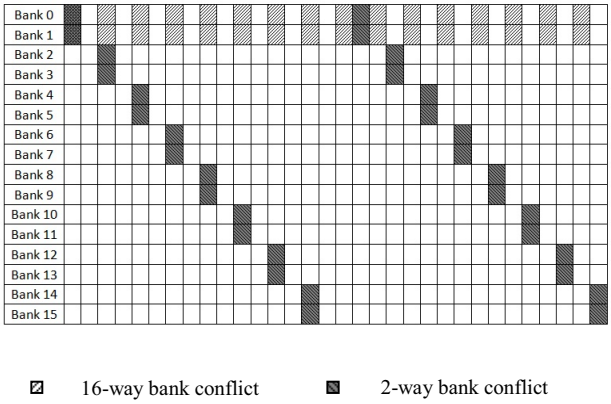

Figure 4. Comparison of 16-way bank conflict and 2-way bank conflict.

D. Range Cell Migration Correction (RCMC)

For a perfect focusing of the image, the degree of range cell migration has to be taken into account. This effect can be compensated by applying a complex interpolator to the range spectrum, but in general interpolation affects the final image quality and the computation is time consuming.

If the range migration phenomenon is assumed constant for the overall range scale, it can be compensated by filtering the SAR data in the two-dimensional (2-D) frequency domain with an elaborate 2-D reference function [9]. Since CUDA has developed a very efficient FFT library, we choose the 2-D frequency domain algorithm only involving FFT and multiplications.

III. RESULTS

The quality of the results and the speed of imaging are mainly considered. To get a credible conclusion, we test the program 100 times and obtain the average time as final results.

The CPU platform is Intel Core i7-930 processor at 2.8 GHz, 4 GB RAM on 64-bit Windows 7 OS. The GPU platform is NVIDIA Tesla C1060, which has 30 multiprocessors and each multiprocessor has 8 cores at 1.3 GHz. GPU and CPU are connected through two PCIe×16 ports.

A. Quality of Imaging Results

Table I shows the main system parameters:

TABLE I. SYSTEM PARAMETERS

Parameter	Value
Wave length	0.03125 m
Sample frequency	100 MHz
Pulse width	20 us
Band width	70 MHz
Pulse repetition frequency	700 Hz
Samples in range/azimuth	4096/4096
Number of azimuth patches	10

Two types of raw SAR data are used to test the performance of the GPU-based platform. The first one is generated by simulator and one point target is placed in the scene. The second one is a surface target obtained from an actual device.

1) Point target situation

Figure 5 shows the imaging result of GPU for the point target.

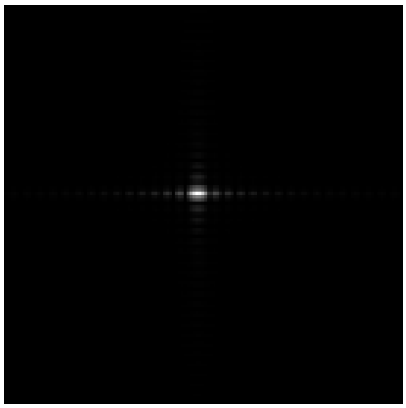

Figure 5. Imaging result of point target.

From the range and azimuth profiles of the point target, we got the peak sidelobe ratio (PSLR), as well as the resolutions in two dimensions, which are illustrated in table II and table III respectively. The quality of images obtained under two different platforms is almost identical.

TABLE II. PEAK SIDELOBE RATIO

	Range	Azimuth
PSLR (GPU)	-13.28 dB	-13.28 dB
PSLR (CPU)	-13.28 dB	-13.28 dB

TABLE III. RESOLUTION

	Range	Azimuth
Theoretical resolution	1.8986 m	0.5288 m
Measured resolution (GPU)	1.8973 m	0.5294 m
Measured resolution (CPU)	1.8973 m	0.5294 m

2) Surface target situation

Figure 6 shows the imaging result of GPU for the surface target with many hills. From the image, we can determine the location of ridges, shoulders and valleys clearly.

B. Speed of imaging

The Tesla C1060 shows 933 GFLOPs of processing performance and comes with 4 GB of GDDR3 memory at 102 GBps bandwidth. For Intel Core i7-930 processor, only one thread from one core is applied in this work and works at 2.8 GHz.

The average processing time of CPU platform is 62 717.3 ms, while the average processing time of GPU-based platform is 1 977.292 ms, resulting in a speedup of 31.72.

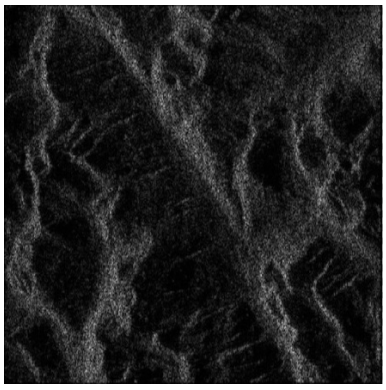

Figure 6. Imaging result of surface target.

The effective bandwidth between CPU and GPU is 5.5 GBps and reaches the best practical performance of PCIe×16 ports. For unpinned memory transfer, the effective bandwidth can only reach 3.0 GBps and consumes 395ms more than pinned memory transfer, as shown in table IV.

TABLE IV. COMPARISON OF PINNED AND UNPINNED DATA TRANSFER

	Data size	Copy data to GPU (ms)	Copy data to CPU (ms)	Total (ms)	Effective bandwidth (GBps)
Unpinned	1280MB	338.8	520.6	859.4	3.0
Pinned	1280MB	222.5	241.8	464.3	5.5

Figure 7 shows the sequential and concurrent operations for data copy and reference function generation. We can see that 61.4 ms is saved. The reason that all the computation time cannot be hided is the overhead of streams management.

Figure 7. Sequential and concurrent operations for data copy and reference function generation

Table V shows the performance of this work and a recently published work [10].

First, let's limit the scope within this work. For a range compression block, the moderate speedup results from the huge time consumption of data transfer between CPU and GPU, which takes 80.2% of the total processing time of this block. This is also true for azimuth matched filter, and the higher speedup is achieved for a higher computation-memory rate. In the future, GPU will probably be located on-chip with the CPU, thus reducing communication overhead [11]. For the following two blocks, their encouraging speedups mainly come from highly efficient parallel FFT

and multiply operations, which greatly contribute to the final speedup of the whole project.

TABLE V. PERFORMANCE OF THIS WORK AND THE REFERENCE WORK

	This work			Reference work		
CUDA kernel	GPU (ms)	CPU (ms)	Speed-up	GPU (ms)	CPU (ms)	Speed-up
Range compression	285.7	1178.2	4.1	900	6600	7.3
Azimuth matched filter	353.5	4487.6	12.7	590	5000	8.5
RCMC & Azimuth compression	209.7	10997.2	52.4	700	31300	18.4
FFTs	479.8	23746.1	49.5	440	21000	47.7
Total	1977.3	62717.3	31.7	4400	70100	15.9

Compared with the reference work, the CPU capacity (Intel i7-930 at 2.8 GHz VS Intel Core 2 Duo E6850 at 3.0 GHz) is in the same level, considering only one thread is applied. Other parameters like GPU platform, image size and computation word length are all the same.

For the RCMC block, this work adopts the 2-D frequency domain algorithm instead of inefficient interpolation algorithm. From the table we can see that 1 490 ms is saved, which is the main contribution to the performance improvement. There is another advantage: the main operations of the frequency domain algorithm are FFTs and those enable us to further accelerate the program using the efficient CUDA FFT library.

In total, we adopt pinned memory transfer method and save 395ms; 64.1ms is hide by concurrent data transfer and device computation; Almost 1500ms is cut off by using the 2-D frequency domain algorithm instead of the interpolation algorithm. Taken together, more than half of the processing time from the reference work is eliminated.

IV. CONCLUSIONS

In this work, a GPU-based parallel implementation of SAR imaging is proposed, with the detailed implementation using CUDA design and optimization techniques. This method obtained the same quality image as the one under the traditional processing platform, but saved nearly 97% processing time, providing a potential solution for real time SAR applications.

V. ACKNOWLEDGMENT

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] J. C. Curlander and R. N. McDonough, Synthetic Aperture Radar - Systems and Signal Processing, Wiley, 1991.

[2] C. Clemente, M. D. Bisceglie, N. Ranaldo, M. Spinelli, “Processing of Synthetic Aperture Radar Data with GPGPU,” IEEE Workshop on Signal Processing Systems (SiPS 2009), Oct. 2009, pp. 309-314.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley,” Technical Report UCB/EECS-183, 2006.

[4] NVIDIA CUDA C Programming Guide 3.1, 2010.

[5] M. Garland and D. B. Kirk, "Understanding Throughput-Oriented Architectures," Communications of the ACM, vol. 53, issue 11, pp. 58-66, Nov. 2010.

[6] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal and P. Dube, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," International Symp. on Computer Architecture (ISCA 2010), June 2010, pp. 451-460.

[7] NVIDIA CUDA C Best Practices Guide 3.1, May 2010.

[8] NVIDIA Optimizing Matrix Transpose in CUDA, Jan. 2009.

[9] G. Franceschetti, G. Schirinzi, “A SAR Processor Based on Two Dimensional FFT Codes,” IEEE Transactions on Aerospace and Electronic Systems, vol. 26, pp. 356-365.

[10] M. D. Bisceglie, M. D. Santo, C. Galdi, R. Lanari, N. Ranaldo, “Synthetic Aperture Radar Processing with GPGPU,” IEEE Signal Processing Magazine, vol. 27, pp.69-78.

[11] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," IEEE Symp. on Application Specific Processors (SASP 08), June 2008, pp. 101-107.