PROCESSING OF SYNTHETIC APERTURE RADAR DATA WITH GPGPU

Carmine Clemente, Maurizio di Bisceglie, Michele Di Santo, Nadia Ranaldo, Marcello Spinelli

Università degli Studi del Sannio, Piazza Roma 21, 82100 Benevento, Italy.

ABSTRACT

Synthetic aperture radar processing is a complex task that involves advanced signal processing techniques and intense computational effort. While the first issue has now reached a mature stage, the question of how to produce accurately focused images in real–time, without mainframe facilities, is still under debate. The recent introduction of general–purpose graphic processing units seems to be quite promising in this view, especially for the decreased per-core cost barrier and for the affordable programming complexity. The authors explain, in this work, the main computational features of a range–Doppler Synthetic Aperture Radar (SAR) processor, trying to disclose the degree of parallelism in the operations at the light of the CUDA programming model. Given the extremely flexible structure of the Single Instruction Multiple Threads (SIMT) model, the authors show that the optimization of a SAR processing unit cannot reduce to an FFT optimization, although this is a quite extensively used kernel. Actually, it is noticeable that the most significant advantage is obtained in the range cell migration correction kernel where a complex interpolation stage is performed very efficiently exploiting the SIMT model. Performance show that, using a single Nvidia Tesla-C1060 GPU board, the obtained processing time is more than fifteen time better than our test workstation.

Index Terms— Synthetic Aperture Radar, parallel processing, GPU, CUDA.

1. INTRODUCTION

Synthetic Aperture Radar (SAR) is an imaging radar for earth observation from satellite and airborne manned/unmanned platforms; it is currently operational in recently launched polar-orbiting platforms such as TerraSAR-X, RadarSAT-2 and Cosmo SkyMed as well as in previous missions. Applicatons are tailored to disaster observation and management, mapping of renewable resources, geological mapping, snow/ice mapping and strategic surveillance of military sites. The data stream produced by high resolution SAR systems may exceed 1 Gb/s and the real-time or near real-time processing represents a demanding requirement for on-board or even ground-based processing systems. 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. Moreover, the scientific community is more and more oriented to a wide range of applications where the first step is the focalization of SAR data [1].

A large amount of solutions based on parallel/distribuited architectures, FPGA boards and degraded resolution algorithms have been developed by industrial and research companies and universities. In [2] both a coarse-grain parallelism and a fine-grain one are adopted. The first is obtained by dividing raw radar data in blocks and implementing the range-Doppler algorithm (RDA), while the latter by using the chirp-scaling algorithm. An interesting approach is presented in [3] where azimuth and range processing are distribuited over different CPU. In [4] the range-doppler alogrithm has been parallelized using a network of workstation; this approach parallelizes both functional blocks and data, with excellents performances. All these architectures turn out to be very expensive and complicated to manage.

The recent development and diffusion of multicore platforms opens new horizons and breaks barriers in the design of architectures for massively parallel processing of SAR data, without loosing in resolution and/or accuracy.

In this work the design of a SAR processor is presented. Our results are related to a CUDA implementation but the design stage is more generally abstracted to General-purpose computing on graphics processing units (GPGPU).

The remainder of this paper is organized as follows. Section 2 provides some background on the SAR acquisition geometry and on the RDA. Section may be skipped by a reader who has some knowledge of these topics. Section 3 introduces coding design strategies with focus on those aspects that are relevant to the proposed prototype. Section 4 explains the SAR processor implementation and optimization with final results, comments and optimization details left to Section 5.

2. SAR GEOMETRY AND PROCESSING DETAILS

The basic geometry of a SAR system is shown in Figure 1. The antenna system, looking across the flight direction, transmits a short chirped waveform, with a pulse repetition time 1/PRF much longer than the waveform duration; the echoes reflected by the earth surface are received through the antenna

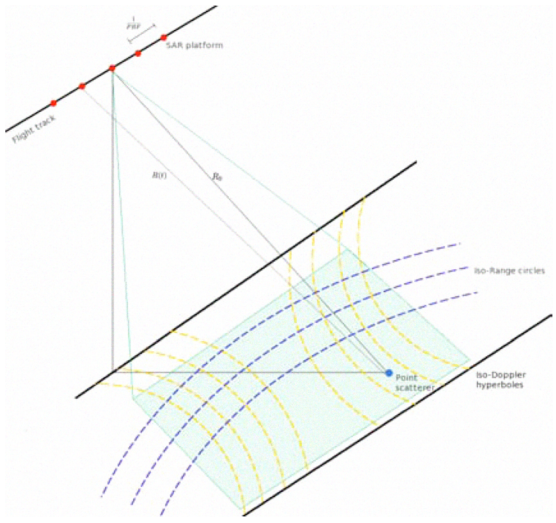

Fig. 1. Syntetic Aperture Radar system geometry

pattern and digitized line by line at each platform position as a two–dimensional array of samples. The Azimuth direction refers to the along-track direction, i.e. parallel to the flight direction while Range refers to the across-track direction. The equirange surfaces are concentric spheres whose intersection with the flat terrain generates concentric circles. Surfaces with identical Doppler shift are coaxial cones with the flight line as the axis and the radar platform as the apex. The intersection of these cones with the flat terrain generates hyperbolas. Objects lying along the same hyperbole will provide equi-Doppler returns.

Since the received signal can be viewed as a superposition of returns from elementary scatters, the processing problem consists in a deconvolution of the received signal that is spread out in both range and azimuth directions.

In essence, the acquisition geometry makes the problem intrinsically space-varying. This implies that the deconvolution process would require a different two-dimentional processing for each image point. We may understand this concept thinking to the response from a single point placed on a non-reflecting surface. Suppose that, at a time $t = 0$, the radar starts acquiring the leading edge of the signal from the point reflector; it is in the position $x = x_0, y = y_0, z = h$ with x the along track coordinate, y the cross track coordinate and h the vertical coordinate. At each acquisition, the radar moves along track, of a step v/PRF where v is the platform velocity. At a new position, say $x = x_0 + kv/\text{PRF}, y = y_0, z = h$ with k an integer, the acquisition time of the leading edge is changed (shortned if the radar moves toward the point, lengthened if the radar moves forward). The result is that the returns from a single point are not aligned from azimuth to azimuth position; this effect, usually known as range migration, is depicted in Figure 1. Range migration is a slowly varying range dependent process, it is an hyperbolic function of the azimuth coordinate and can be accurately removed by a two-dimensional processing stage.

SAR data processing consists of a set of procedures for obtaining the final spatial and radiometric resolutions from the instrument. It should satisfy requirements of accuracy, computational complexity and technical feasibility. In the relatively small set of available techiques for SAR data processing (also referred to as SAR focusing), the range-Doppler algorithm and its variants is probably the most widely used. It was first developed by MacDonald Dettwiler and Associates (MDA) and the Jet Propulsion Lab (JPL) in 1979 for the processing of SEASAT data [5, 6]. The algorithm is designed to achieve block processing efficiency, using frequency domain operations in both range and azimuth, while maintaining the simplicity of the one-dimensional operations. Block processing efficiency is also achieved for the Range Cell Migration Correction (RCMC), operation because it is performed in the so called *range–Doppler plane*, as explained later in this section. The sequence of core steps of the algorithm are shown in Figure 2 where it is worth to note that the deconvolution is split in the range and azimuth directions. This is possible because the range between the radar and a given point target is assumed as fixed for a given pulse (start-stop approximation). The range focusing is a filtering stage carried out in

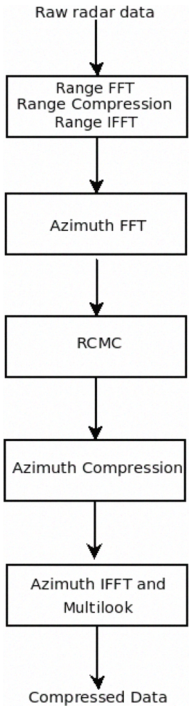

Fig. 2. Range-Doppler SAR processing algorithm

three steps:

1. the **Range FFT**, that is a set of one–dimensional Fast Fourier Transforms in the range direction, one for each ac-

quisition of the sensor at the time n/PRF.

2. the **Range Compression**, namely a matched filtering obtained by multiplying the range transformed data by the range reference function in the frequency domain.

3. **Range IFFT**, the inverse one–dimenional set of FFT to re–obtain data in the time domain.

The azimuth focusing is a range dependent matched filtering stage that includes RCMC. It includes:

1. the **Azimuth FFT** that is a set of one–dimensional Fast Fourier Transforms in the azimuth direction; the resulting data lie in the range–Doppler plane.

2. **Range Cell Migration Correction** In the frequency domain, variations of the point–to–radar distance induce a range-dependent doppler frequency shift in the received signal. This stage includes range shifts and interpolation operations to catch the exact azimuth trajectories of the received echoes.

3. **Azimuth Compression** namely a matched filtering obtained by multiplying the azimuth transformed data by the range–dependent azimuth reference function in the frequency domain.

4. **Azimuth IFFT** the inverse one–dimenional set of FFT to obtain the final image.

3. CODE DESIGN GUIDELINES

Many core GPU-based platforms and architectures are the enabling technology for massively parallel processing of SAR data, so that the most accurate processing techniques will be rapidly available for real–time applications at a low cost. The main barrier to the diffusion of parallel applications working on many core systems is that writing multi-threaded programs requires to be quite confident with more specialized techniques, like synchronization, data partitioning and behavior of access patterns through memory and management of data location.
The recent introduction of CUDA (Compute Unified Device Architecture) programming environment represents an important contribution towards the reduction of GPU programming complexity without sacrificing performance. CUDA is based on a Single Instruction Multiple Threads (SIMT) programming model; it is very similar to SIMD (Single Instruction Multiple Data), where a single code instruction can be executed by different threads on many data. To achieve maximum efficiency, CUDA applications can use a great number of threads working on small chunks of data, organized in different subsets called *blocks*. Because the size of the subsets in a SIMT architecture is much smaller than in a SIMD one, the environment turns out to be more flexible and efficient [7, 8]. The complexity of the programming environment moved us to adopt the following guidelines in the algorithm design.

1. Identify the computational domain of interest as a structured hierarchy of threads that are executable in parallel.

2. Write kernels that execute threads. In a SIMT model, the code of a kernel describes the actions of a single thread, (with CUDA, the calling of a kernel also specifies the total number of threads to be created and executed).

3. Carefully plan memory usage and data transfers among the different levels of memory hierarchy. Global memory has a high latency (hundreds of clock cycles) and is prone to access conflicts when many threads work on the same data. Performance can improve even of an order of magnitude when accesses from groups of threads meet alignment and ordering criteria, a feature referred to as *coalescing*. Moreover the use of the shared memory allows to obtain better performance when different threads in a block need to use the same data.

4. Accurately manage the divergences within each kernel code. Thread blocks are dynamically assigned to the parallel processing units of the GPU, where threads are instantiated by hardware thread scheduling mechanisms and executed. The threads of a block start all at the same instruction, and execute one common instruction at a time. In order to avoid efficiency penalties, code should be organized so that threads in the same block (in particular in the same *warp* in CUDA) do not give rise to divergent conditional branches or at least minimize their number.

5. Maximize the computational density, avoiding memory accesses. Recomputing sometimes is less expansive than read data from memory.

6. Use of all instruments provided by the API as the intrinsic functions in the Special Function Unit, loop unrolling, syncronizations, streamings and texture.

4. SAR PROCESSING WITH GPGPU

Our approach will be necessarily related to a CUDA implementation, because the prototype and its results have been developed under such a platform; nonetheless, the discussion will be abstracted when platform–independent functional structures can be defined.
The GPU implementation of the SAR processing is organized by sectioning the data matrix in subswaths, (see Figure 3) and executing FFT computations in batched mode. This means that many FFTs are executed in parallel or, more correctly, that data blocks belonging to many FFT computations are executed in parallel. The batched–mode FFT kernel will be illustrated later in Section 4.3.

4.1. Range compression and range–Doppler representation

The implementation of the range filtering stage follows the steps depicted in Section 2 with forward and inverse FFT,

along N_a azimuth gates, executed in batched mode. Define a range gate as the set of radar returns that are mapped into a column of the data matrix. Equivalently, an azimuth gate is the set of radar returns that are mapped into a row of the data matrix. The matched filtering in the frequency domain can be computed very efficiently because all azimuth cells require the same (Fourier transformed) range reference function. This task is executed in parallel by a kernel where each thread reads a range gate, executes the product between range gate values and the corresponding value of the range reference function and writes the result in the global memory. Data in the range–Doppler domain are obtained after N_r forward FFTs in batched mode, along the range gates.
The optimum number of parallel executions per block depends on the hardware GPU resources in terms of available threads and memory and is targeted to saturate the available bandwidth. Thus, the value of the variables N_a and N_r is subject to machine–dependent optimization and will be assessed in the following section. To optimize global memory bandwidth usage, reading-from and writing-to global memory are coalesced: the region of memory accessed by a thread block is contiguous. This is obtained by opportunely using thread indices to write memory access instructions so that the k^{th} thread in each half warp accesses the k^{th} element in the block being read.

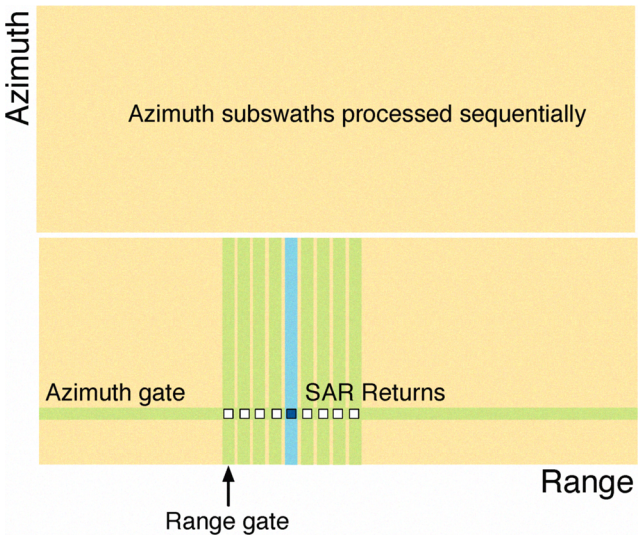

Fig. 3. Matrix deployment of SAR data and scheme of the block processing strategy

4.2. Range cell migration correction and azimuth compression

The azimuth processing encompasses, with no doubts, all the main features leading to the SAR image formation. The range cell migration correction in the range–Doppler plane is a one–dimensional interpolation problem where the signal values along the migration path are evaluated with an interpolation function of a sufficiently high order (we used an eight–order *sinc* kernel for the case at hand). It is efficiently coded using a *fine grained* parallelism by assigning one thread to each range gate and by transferring chunks of data from global memory to shared memory (this method, referred to as *tiling*, is used to circumvent the tradeoff between the large global memory and the small but distributed shared memory. Data are partitioned into smaller subsets so that each tile fits into the shared memory and the kernel computations on that tile can be carried out independently).

In our planning the kernel executes computations on a range gate and we have defined a block as a collection of 64 threads (a grid collects of as many blocks to include all the range gates). A tile is the subset of range cells spanned by a block. The kernel sequentially moves a tile from the global memory to the shared memory in coalesced mode, executes interpolation of the data values using eight surrounding samples (four from each side as shown in Figure 3) and writes the result in the global memory.

Access to data from different threads is subject to possible bank conflicts because memory–read requests to the same memory address cannot be served simultaneously; if this happens, the hardware serializes the accesses and performance decrease. To achieve maximum speed, memory–read is organized as in Figure 4 where it is evident that a memory address is always accessed sequentially. During the memory accesses $k, k+1 \ldots k+9$, thread 1 reads data values $rc(n-4), rc(n-3), \ldots rc(n+4)$ that are used sequentially for interpolation on the n–th sample.

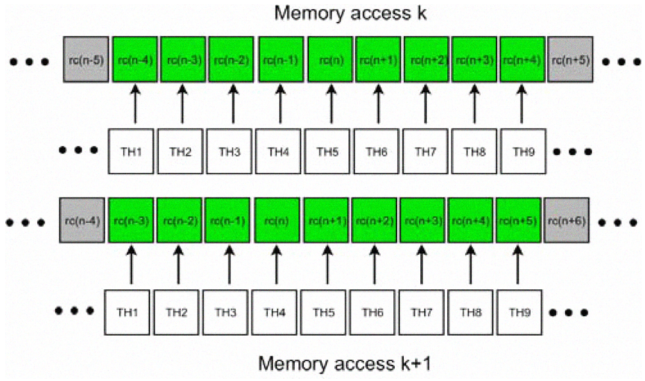

Fig. 4. Coalesced memory access for 9-samples interpolation required in Range Cell Migration Correction

The azimuth compression is obtained by multiplying each range migration corrected range gate by the corresponding azimuth matched filter.

4.3. FFT optimization

The range-Doppler algorithm makes an intensive use of one-dimensional FFT and IFFT for complex data of radix-2 size. In CUDA-based implementations, a possible approach is the direct use of NVIDIA's CUFFT [9], a library which delivers a parallel implementation of complex and real data transforms in one, two and three dimensions (for one dimension the batched execution is available).
The main limitation of CUFFT is that there are no specific optimizations with respect to the GPU resources. The horizon of imaginable improvements achievable by more tailored implementations moved us to investigate a very basic but quite optimized algorithm for FFT computation, with the aim to exploit at most the memory hierarchy and the high throughput computing resources of GPUs.
The proposed library is based upon the work presented in [10] by Volkov and Kazian, which, in turn, relies upon the Cooley and Tukey framework, and currently supports the transform of batched 1D arrays of complex values. The multi-threading implementation re-scales the original problem into a large number of small radix FFTs, where an efficient use of register files and per-block shared memory is exploited. The main features of the approach are: (1) the use a different kernels for managing different vector sizes; (2) the need of a final per-batch transposition; (3) the use of code optimizations, such as shiftings for power–of–two divisions or modulus operations, loop-unrollings and so on.
In order to compute the base-case transform, the model assigns each thread a little subset of values from the input arrays. For example, in the 4096 points per-batch FFT calculation, a kernel is required to compute radix-8 FFTs and a single thread is charged for an 8 points FFT. This simple task can be performed very efficiently with the following steps:

1. read data from global memory and save them in the register file (*code lines 5-7*);
2. evaluate the FFT using an hardcoded, optimized routine (*code line 8*);
3. multiply by the twiddle factor (*code lines 9-12*).

Finally, values are reordered and transferred back to global memory for a global per-batch transposition. This computation is performed by a separate kernel. To achieve the best performance in the transpose operation for a single vector, blocks of values are transferred in the shared memory trying to avoid bank conflicts and using coalesced write to and read from global memory.
The main drawback of the Cooley-Tukey FFT framework is the need of a final per-batch transposition. For these reasons, our intention is to switch to the Stockham algorithm that does not require any transposition but, conversely, requires twice the memory space. This will be rapidly manageable in the next–generation GPUs.

FFT kernel (excerpt)

```
1. __global__ void fftff_8_points(cmplxValue *v ) {
2.   cmplxValue temp[8];
3.  /* offset computation in the array v of the 8 values
4.    to store in the temp array ñ code omitted */
5.   #pragma unroll 8
6.   for( int i = 0; i < 8; i++ )
7.     temp[i] = v[i<<9];
8.   radix8_fft( temp );
9.   #pragma unroll 8
10.   for( int j = 1; j < 8; j++ )
11.     temp[j] = temp[j] * twiddleFactor_8(offset,
12.         j, 4096,FFTFF_FORWARD);
13.   #pragma unroll 8
14.   for( int i = 0; i < 8; i++ ){
15.     int ind = indexReverse_8(i);
16.     v[i<<9] = temp[ind];
17.   } ...
```

5. OPTIMIZATION AND RESULTS

Optimization is an essential and critical phase in the design of parallel procedures and an accurate tuning of the execution configuration is required to obtain close to hardware–limited performance. There are two main reasons because it is difficult to determine the optimum execution configuration for a GPU-based application; the first is that there is not a linear dependence between performance and block size [11], the second is that there is, generally, a domain of influence among the different parameters of a GPU. The optimization is, thus, carried out using an automatic profiling stage with a tuner embedded in the code. The tuner runs all the kernels with different settings and extracts the best execution configuration. To avoid an exhaustive search a simple branch–pruning strategy is implemented to constrain the optimization space and improve efficiency. Finally, the best configurations are stored in a database, with the data dimensions and the device features.
Our implementation is based on CUDA 2.0 [12, 13] and, besides the plethora of minor tricks, the main parameters to be considered are those related to the number of blocks in a grid (G) and the number of threads in a block (B).
The number of range gates N_r is a priori defined by the SAR acquisition system, thus we have

$$G = \left\lceil \frac{N_r}{B} \right\rceil .$$

If N_r is not an integer multiple of B, the number of required grids G becomes the next greater integer; this causes the number of threads that execute a kernel to exceed the range gates to be processed. In this instance, the possible exceeding threads should be managed by generating divergent branches.
The SAR processor has been tested with an ERS-2, July 14th 1995 pass, centred in the Campania region (Italy). The size of the ERS-2 raw data matrix is 26880×4912 azimut-range gates. Results exhibit the same numerical accuracy as those obtained with the original CPU based algorithm; the normalized mean squared error is in the order of 10^{-8}.
The processor has been tested using an Nvidia Tesla C1060 on a workstation equipped with Intel Core 2 Duo E6850 at 3.00

CUDA Kernel	# threads per block	GPU μs	CPU μs
Range compression kernel	64	1.1×10^6	6.6×10^6
Parameter estimation kernel	32	7.1×10^2	3.1×10^4
Azimuth matched filter kernel	64	5.9×10^5	5.0×10^6
RCMC and azimuth compression kernel	64	1.7×10^6	31.3×10^6

Table 1. Best configuration for an ERS-2 data, obtained with the embedded tuner.

GHz (FSB at 1333 MHz and 4 MB of L2 cache), 4GB RAM, 16x PCI-Express bus with Linux Ubuntu 9.04 OS. In this configuration the mean processing time is 4.65 seconds, whereas the CPU based SAR processor runs in 70.1 seconds. In terms of speedup, the presented processor achieves a speedup of about 15 with respect to the reference workstation.

The best execution configuration is shown in Table 1. Speedups for the functional blocks of the RDA are shown in Figure 5, using the execution configuration in Table 1. Results show that most of time is consumed by the Range compression kernel and by the RCMC and azimuth compression kernel. However, as long as the speedup obtained for the RCMC kernel and for the FFT computations is quite satisfying, the speedup for the range compression kernel is very moderate. Deeper investigation has shown that much of the time is spent in the data transfer from CPU to GPU: this is clearly a case where the *computational density* is small and the advantage gained by the FFT computation is lost in the data transfer operations.

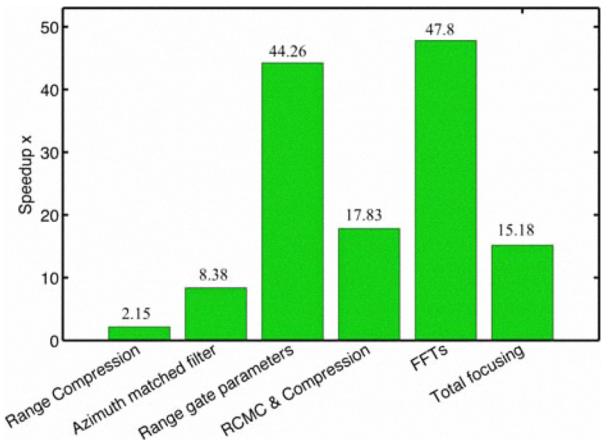

Fig. 5. Obtained speedups for the differents functional blocks of the RDA and total speedup. Processed azimuth patches are of 4096×4096 samples.

6. REFERENCES

[1] I.G. Cumming and F.H. Wong, *Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementation*, Artech House Publishers, first edition, 2005.

[2] X. Jang, Z. Minhui, W. Yirong, and P. Hailiang, "Parallel programming in SAR imaging processing," in *Geoscience and Remote Sensing Symposium, 1999. IGARSS '99 Proceedings. IEEE 1999 International*, 1999, pp. 567–568.

[3] R. Albrizio, G. Aloisio, A. Mazzone, and N. Veneziani, "Multiprocessors architectures for SAR data processing," in *Geoscience and Remote Sensing Symposium, 1991. IGARSS '91 Proceedings*, 1991, pp. 267–270.

[4] P.G. Meisl, M.R. Ito, and Ian G. Cumming, "Parallel synthetic aperture radar processing on workstation networks," in *Proceedings of IPPS apos;96, The 10th International*, 1996, pp. 716–723.

[5] C.Wu, "Processing of SEASAT SAR data," in *SAR Technology Symp., Las Cruces, NM*, September 1977.

[6] R. Bamler, H.Breit, U. Steinbrecher, and D. Just, "Algorithm for X-SAR processing," in *Geoscience and Remote Sensing Symposium, 1993. IGARSS '93 Proceedings, Tokyo, Vol.4*, August 1993, pp. 1589–1592.

[7] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[8] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[9] *CUDA, CUFFT Library 1.1.*

[10] V.Volkov and B. Kazian, "Fitting FFT onto G80 architecture," in *UC Berkeley CS258 project report*, May 2008.

[11] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu, "Program optimization space pruning for a multithreaded gpu," *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, p. 195–204, 2008.

[12] D. Kirk and W. Hwu, "Cuda textbook," in preparation, http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook.

[13] *NVIDIA CUDA Programming Guide 1.*