Synthetic Aperture Radar imaging on a CUDA-enabled mobile platform

Massimiliano Fatica and Everett Phillips
NVIDIA Corporation
Santa Clara, CA 95050, USA

***Abstract*—This paper presents the details of a Synthetic Aperture Radar (SAR) imaging on the smallest CUDA-capable platform available, the Jetson TK1. The results indicate that GPU accelerated embedded platforms have considerable potential for this type of workload and in conjunction with low power consumption, light weight and standard programming tools, could open new horizons in the embedded space.**

I. Introduction

The use of GPUs in high performance computing, sometimes referred to as *GPU computing*, is becoming very popular due to the high computational power and high memory bandwidth of these devices coupled with the availability of high level programming languages and tools. The GPU capabilities have not been unnoticed in the high performance embedded computing space. For example, Baralli et al. [1] showed the capabilities of GPUs to process real time synthetic aperture sonar on board of autonomous underwater vehicles (AUV), allowing a higher level of autonomy and data driven missions.

While there are embedded and ruggedized platforms with discrete GPUs on the market capable of running CUDA [3], this is the first time that a System on Chip (SOC) featuring a CUDA-capable GPU is available for development and possible deployments.

This paper presents the implementation of a SAR imaging algorithm on the Jetson TK1 platform, analyzing both the CPU only performance and the CPU/GPU implementation. Our main interest is in the capabilities of the platform, both from the hardware and software point of view, and not in the SAR algorithm processing itself. For this paper we have decided to use the material available from the MIT OpenCourseware class “Build a Small Radar System Capable of Sensing Range, Doppler, and Synthetic Aperture Radar Imaging” [2]. The material has raw radar data and MATLAB scripts to preprocess the data and generate the SAR image.

II. Tegra K1 and Jetson TK1

The tests were performed on the new Jetson TK1 platform, a small (5 inches by 5 inches) board designed for development of embedded and mobile applications depicted in Figure 1. The Jetson TK1 is powered by the Tegra K1, the first mobile processor to feature a CUDA-capable GPU. It features a GK20A Kepler GPU with 192 cores and a quad-core ARM Cortex-A15 32bit CPU. An upcoming version of the Tegra K1 will offer two custom ARM 64bit cores and it will be pin compatible with the 32bit version. Jetson has 2GB of system memory and a Gigabit Ethernet port plus other peripheral ports (GPIO, mini PCI-e, HDMI, serial, USB). The board consumes 2.2W when idle, around 4-5W when using the CPU cores and up to 11W when using the GPU cores. It uses a Linux distribution derived from Ubuntu and all the normal CUDA developement tools are available. Being ARM-based, there are no many ISV applications available, all the software and tools are usually open-source based. Since MATLAB is not available on this platform, we used Octave 3.6.

III. CUDA

CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

CUDA follows the data-parallel model of computation. Typically each thread executes the same operation on different elements of the data in parallel. Threads are organized into a 1D, 2D or 3D grid of blocks. Each block can be 1D, 2D or 3D in shape, and can consist of up to 1024 threads on current hardware. Threads within a thread block can cooperate via the on-chip shared memory or in the latest generations (compute capability greater than 3.0) via shuffle instructions. Thread blocks are executed as smaller groups of threads known as "warps".

Kernel invocations in CUDA are asynchronous, so it is possible to run CPU and GPU in parallel. Data movement can also be overlapped with computations and the GPU can DMA directly from page-locked CPU memory. There are also a large number of CUDA based libraries, from linear algebra to

Figure 1. NVIDIA Jetson TK1 developer board.

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

random number generation. Two libraries that are particularly relevant to this porting are CUFFT [4] and Thrust [5], a C++ template library for CUDA similar to the Standard Template Library (STL). Thrust includes data parallel primitives like scan, sort and reduce, which can be used as building blocks for parallel algorithms, hiding the complexity of the underlying GPU implementation.

A. Calling CUDA code from Octave

At the HPEC 2007 conference, Fatica and Jong [6] showed for the first time how to interface MATLAB and CUDA using mex files. Since then, Mathworks has added GPU capabilities in MATLAB and in the latest versions it is possible to use high level constructs operating on GPU arrays as well as call CUDA kernels directly from MATLAB scripts. Unfortunately, MATLAB is not available for Linux on ARM so we had to use Octave and go back to the mex interface. The basic idea is to use the nvcc compiler to generate the object file and link the necessary libraries with *mkoctfile*, the equivalent *mex* command in Octave.

It is also important to remember that MATLAB and Octave store complex arrays as two consecutive arrays, one for the real part and one for the imaginary part. However, CUDA C and the CUDA libraries are expecting the data in an interleaved format. Once we move the data to the GPU, we need to change the storage format. The calls to CUFFT also need the dimensions swapped, since MATLAB and Octave store the data in column-major format while CUFFT is expecting them in row-major format.

IV. SAR

The SAR imaging has 2 distinct phases:

1) Acquisition and pre-processing of the raw SAR data
2) Processing of the SAR data

An important point, is that this work is only intended to measure performance of the algorithm and ease of implementation, not to improve the SAR algorithm.

A. Pre-processing of raw data

The original MATLAB script, *SBAND_RMA_opendata.m*, is quite slow even on a workstation but when run unmodified (aside from additional timing information) takes almost 18 minutes on the Jetson. Being able to reduce the preprocessing time is crucial if the SAR image needs to be generated during the mission. Moreover, most of the operations are carried out in a serial fashion and to efficiently use the GPU cores, we need to express as much computation as possible in a parallel way.

We can identify three distinct phases:

- Reading the wav file containing the raw radar data. The original data is represented in 16-bits. The reading time for the .wav file with 15570944 samples is 5.5 seconds. The reading time will depend on the media used, the standard filesystem on Jetson is on a flash module, but it is also possible to connect a SATA harddisk or a SSD.
- Parse data by position: looking for silence in the data. If Nrp is the minimum number of samples between range profiles, we are looking at the first points with absolute value bigger than the mean that are preceded by Nrp points below the mean. For the pulse time and the frequency used during the data acquisition, Nrp is equal to 11025. The original script is looping through all the points and for each eligible point computes the sum of the preceding Nrp.

```
%parse data here by position
%(silence between recorded data)
 rpstart = abs( trig )>mean(abs(trig ));
count = 0;
Nrp = Trp*FS; %min # samples between range profiles
for ii = Nrp+1:size( rpstart ,1)-Nrp
 if rpstart ( ii )==1 & sum(rpstart( ii -Nrp:ii-1))==0
   count = count + 1;
   RP(count ,:)      =     s( ii : ii +Nrp-1);
   RPtrig(count ,:) = trig ( ii : ii +Nrp-1);
 end
end
```

This is the phase that consumes most of the time, 887.31 seconds. It is possibile to find these voids using prefix sums and element-wise multiplications, both very amenable to parallel processing. The idea is shown in figure 2.

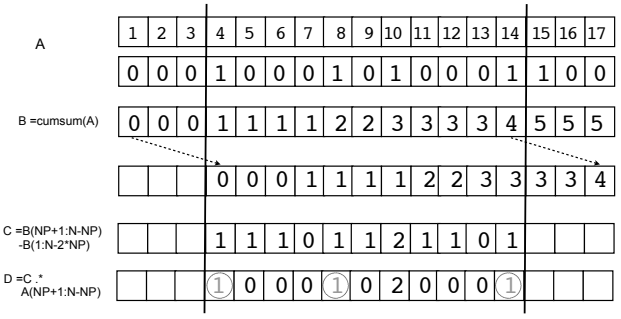

Figure 2. Find voids in parallel: we are looking for elements of the array A that have 3 or more zeros before the ones with value 1. The elements that should be selected are indices 4, 8 and 14.

If the array A stores the values 0 (point below the mean) and 1 (point above the mean), we will construct a new array B containing the prefix sum (also known as cumulative sum) of A. The difference between the element i and the element i-Nrp of B will tell us the number of non zero values of A between these two points. If this difference is multiplied by the corresponding value of A, the points with the value of 1 are the points we are looking for. Before writing the CUDA code, we implemented the new approach in MATLAB syntax:

```
psum=cumsum(rpstart);
dis=psum((Nrp+1:size( rpstart ,1)-Nrp))
    -psum(1    : size ( rpstart ,1)-2*Nrp);
dis=dis.* rpstart ((Nrp+1:size( rpstart ,1)-Nrp));
ind=find(dis==1);
```

```
for ii = 1:size(ind)
    istart =ind(ii)+Nrp;
    iend   = istart +Nrp-1;
    RP(ii,:)      =     s( istart :iend );
    RPtrig( ii ,:) = trig ( istart :iend );
end
```

The elapsed time for the improved version is now 0.63 seconds and the script gives the correct results. For further speed-up we could also pre-allocate RP and RPtrig since we now know their dimensions before entering the loop. This result was obtained in Octave running on the CPU but we now have an algorithm that is parallel and can be easily ported to the GPU.
The CUDA implementation of this phase is quite simple to write using Thrust to compute the cumulative sum and to select the elements of the array equal to 1.

```
/* Compute the mean */
  trig_s_kernel <<<16,128>>>(buf,trig,s,avg,f);
/* Compute rpstart */
  rpstart_kernel <<<16,128>>>(trig,avg, rpstart ,f);
/* Compute cumsum with Thrust */
  thrust :: inclusive_scan ( dp_rpstart , dp_rpstart +f,dp_psum);
/* Compute final value of array */
  ind_kernel<<<16,128>>>(rpstart,psum,ind,count_h,f,sr /4);
/* Select elements equal to 1 with Thrust */
  thrust :: copy_if(dp_ind,dp_ind+f,dp_ind,is_pos ());
```

A standalone CUDA implementation runs in 0.1 seconds. Calling the CUDA code from inside Octave will add some overhead, since we can't use pinned memory.

- Parse data by pulse: after locating the rising edge of the sync pulses, accumulate them and apply a Hilbert transform. This phase takes 162.2 seconds. The original script uses a logic similar to the previous phase.

```
%parse data by pulse
thresh = 0.08;
for jj = 1:size(RP,1)
    SIF = zeros(N,1); %clear SIF;
    start = (RPtrig( jj ,:)> thresh );
    count = 0;
    for ii = 12:(size( start ,2)-2*N)
        [Y I] = max(RPtrig(jj , ii : ii +2*N));
        if mean(start( ii -10:ii-2)) == 0 & I == 1
            count = count + 1;
            SIF = RP(jj , ii : ii +N-1)' + SIF;
        end
    end
    %hilbert transform
    q = ifft (SIF/count);
    sif ( jj ,:) = fft (q(size(q,1)/2+1: size(q ,1)));
end
sif (find(isnan( sif ))) = 1E-30; %set all Nan values to 0
```

We use an approach similar to the one we used to find the voids in parallel.

```
%parse data by pulse
thresh = 0.08;
for jj = 1:size(RP,1)
    SIF = zeros(N,1); %clear SIF;
    start = (RPtrig( jj ,:)> thresh );
    psum=cumsum(start(1:Nrp-2*N));
    dis=1+psum(10:Nrp-2*N-2)-psum(1:Nrp-2*N-11);
    dis=dis.* start (12:Nrp-2*N);
    ind=find( dis==1)+11;
    count = 0;
    for ii =1:size(ind,2)
        myind=ind(ii);
        [Y I] = max(RPtrig(jj,myind:myind+2*N));
        if I ==1
            count = count + 1;
            SIF = RP(jj,myind:myind+N-1)' + SIF;
        end
    end
    %hilbert transform
    q = ifft (SIF/count);
    sif ( jj ,:) = fft (q(size(q,1)/2+1: size(q ,1)));
end
sif (find(isnan( sif ))) = 1E-30; %set all Nan values to 0
```

Elapsed time is 0.165 seconds, running Octave on CPU. At the end of the preprocessing step, the raw input data has been reduced to a complex array of size (55x441). Our better pre-processing algorithm improved the processing time from 18 minutes to 6.2 seconds even without using the GPU. The time spent in the reading of the file (5.5 seconds) could be reduced using a better disk (instead of the on board flash card) or with a different file format.

B. Processing of the SAR data

The next script, SBAND_RMA_IFP.m, takes the SAR data and generates the final image. The SAR imaging is composed of the following phases:

- Apply windowing function, pad the data so that the radar can squint, apply 1D FFT and FFTSHIFT: At the end of this phase, the original input data of dimension (55x441) is transformed in a array of dimension (2048x441).
- Apply matched filter.
- Perform Stolt interpolation: correct for range curvature. At the end of this phase, the array is of dimension (2048x1024). An additional Hanning function is also applied to clean up the data.
- Perform inverse 2D FFT on a padded array four times larger than the original.

The first problem we encountered was the amount of memory needed by the script. The final data is a complex array of size 8192x4096 and there is not enough memory to carry on all the computation in double precision. Just before the last inverse 2D FFT transform, we convert the data to single precision. We optimized the script as much as possible, pre-allocating arrays and removing redundant operations. If we run the script using the standard CPU-only Octave, the elapsed time is 110 seconds:

```
octave:1> SBAND_RMA_IFP
  Along track FFT in      0.204203 seconds.
  Matched filter in      40.384115 seconds.
  Stolt interpolation in 20.263402 seconds.
  2D inverse FFT         14.344351 seconds.
  SAR processing in     110.220000 seconds.
```

Running the same script on a MacBook Pro with a 2.3 GHz Intel Core i7 and MATLAB 2013b takes 8 seconds:

```
matlab> SBAND_RMA_IFP
 Along track FFT in       0.056983 seconds.
 Matched filter in        0.073197 seconds.
 Stolt interpolation in   1.185652 seconds.
 2D inverse FFT           0.741271 seconds.
 SAR processing in        8.080000 seconds.
```

Some of the differences in runtime are due to the different software, for example Octave seems to be very inefficient in executing the following code for the matched filter phase:

```
%step through each time step row to find phi_mf
for ii = 1:size(S,2)
    %step through each cross range
    for jj = 1:size(S,1)
        phi_mf(jj,ii) = Rs*sqrt((Kr(ii))^2 - (Kx(jj))^2);
    end
end
%apply matched filter to S
smf = exp(j*phi_mf);
S_mf = S.*smf;
```

The next step in our porting was to move all these phases to the GPU using a mex file and custom CUDA kernels. The kernels are quite simple, for example the following shows the CUDA kernel that performs the matched filter with 2D blocks:

```
__global__ void matched_filter(cufftDoubleComplex *S,
                               int M, int N, double Rs, double Kx0,
                               double dx, double Kr0, double dr)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    double c,s;

    if (i<M && j<N)
    {
     double kx=Kx0+i*dx;
     double kr=Kr0+j*dr;
     double angle=Rs*sqrt(kr*kr-kx*kx);
     sincos(angle,&s,&c);
     double Sx=S[i+j*M].x*c-S[i+j*M].y*s;
     double Sy=S[i+j*M].x*s+S[i+j*M].y*c;
     S[i+j*M].x=Sx;
     S[i+j*M].y=Sy;
    }
}
```

The GPU-accelerated code is now capable of generating the SAR image in 3 seconds. If we look at a detail output, most of the time is spent in copying the results back and in the setup of the FFT library. The long setup time is due to the size of the library, but it only affects the first time the library is called.

```
octave:1>
zpad=2048;
v= sar_gpu(sif,zpad,Kr,Rs,delta_x);
Data is complex, M=55, N=441, ZPAD=2048
***** cudaMalloc time:  0.088 seconds
***** fftplan    time:  0.654 seconds
***** warmup GPU time:  0.040 seconds
***** copy H2D   time:  0.001 seconds
***** GPU Comp   time:  0.332 seconds
***** Copy D2H   time:  1.316 seconds
***** unpack     time:  0.232 seconds
```

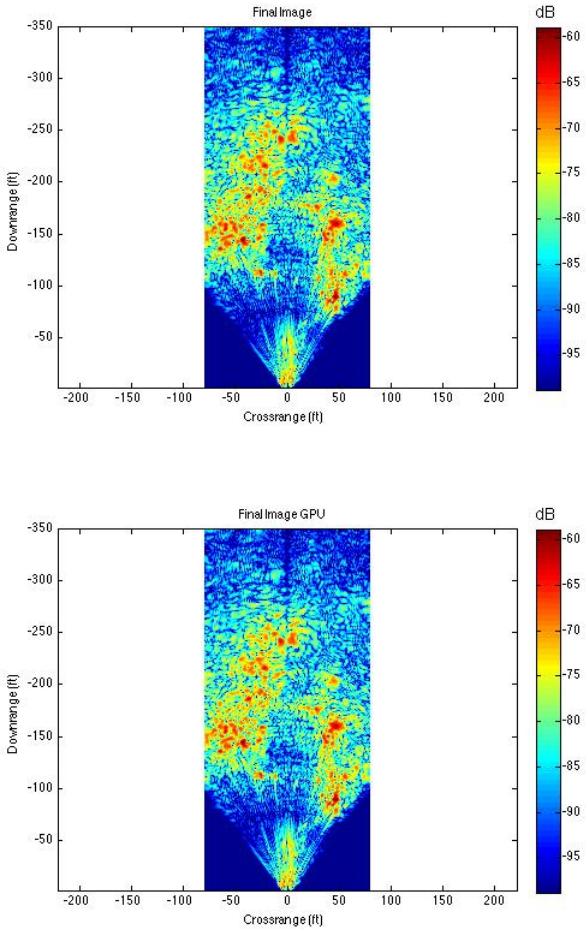

Figure 3. Final SAR images: the top one is obtained on CPU, the one on the bottom on the GPU

```
***** free       time:  0.072 seconds
***** Total      time:  2.936 seconds
  GPU 3.033000 seconds.
```

Using the nvprof profiler tool, we can also look at the individual kernel times:

```
Time(%)      Time     Name
 79.02%  1.37451s     CUDA memcpy DtoH]
  8.06%  140.22ms     spVector8192D::fftDirection_t=1
  6.75%  117.38ms     spRadix0064B::fftDirection_t=1
  1.69%  29.466ms     [CUDA memcpy DtoD]
  1.33%  23.151ms     set_to_zero(float2*, int, int)
  1.21%  21.068ms     matched_filter(double2*,...
  0.97%  16.866ms     interp_kernel_float(double2 ..
  0.77%  13.467ms     dpVector2048D::fftDirection_t=-1
  0.19%  3.3606ms     along_track(double*,  ...)
  0.00%  73.833us     [CUDA memcpy HtoD]
```

The generated SAR images shown in figure 3 are pretty much indistinguishable. Only plotting the relative error between the two datasets, we are able to distinguish them. Figure 4 shows this difference along a vertical cut.

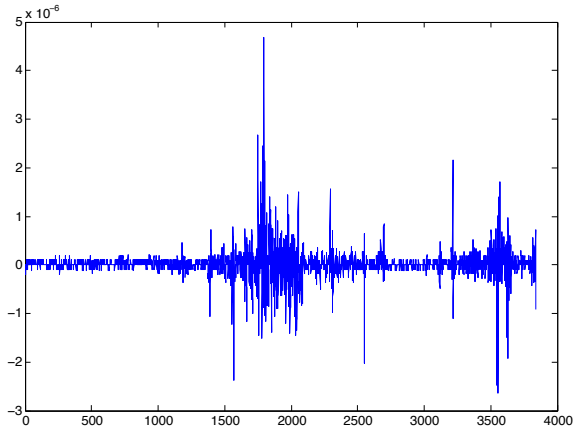

Figure 4. Difference between CPU and GPU results along vertical cut at I=1023

Since most of the time is spent in the data transfer back to the CPU and most of the data is discarded before generating the figure, we can implement an additional optimization by moving the postprocessing of the image to the GPU. We flip, rotate, clip, and scale the data on the GPU. Now we only bring back to CPU memory the data required to generate the plot.

```
octave:1> SAR_gpu
Data is complex, M=55, N=441, ZPAD=2048
***** cudaMalloc time:  0.086784 seconds
***** fftplan    time:  0.507067 seconds
***** copy H2D   time:  0.000811 seconds
***** warmup GPU time:  0.002435 seconds
***** set_zero   time:  0.020761 seconds
***** along_trk  time:  0.002271 seconds
***** 1D FFT     time:  0.006936 seconds
***** mat filter time:  0.010135 seconds
***** Stolt      time:  0.008533 seconds
***** 2D IFFT    time:  0.204319 seconds
***** flip rot   time:  0.026690 seconds
***** scale      time:  0.010833 seconds
***** thrust max time:  0.004489 seconds
maxx = -5.886394e+01
***** TOT GPU    time:  0.295100 seconds
***** Copy D2H   time:  0.157    seconds
***** free       time:  0.042222 seconds
***** Total      time:  1.118386 seconds

  GPU 1.142000 seconds.
```

This reduced the SAR generation to just 1.1 seconds, starting from the preprocessed data. The FFT library initialization is now the bottleneck. If we compute multiple images, the subsequent ones will be processed in just 0.6 seconds.

We also wrote a complete implementation in CUDA, without using Octave. The whole processing, from reading the file to generating the data for the image is now completed in less than 1.5 seconds.

V. Conclusion and Future Plans

The results are quite encouraging, the Jetson platform is very capable and thanks to a standard software toolchain, application porting is very quick. If CUDA code is already available, it is just a matter of recompiling. We were able to process SAR images on this very low power platform (less than 11W) in significantly less time than on a fast laptop. The main limitation of the platform is the available memory, only 2GB. Future platforms with 64bit ARM processors will relax this limitation. We are also looking at the possibility of putting the Jetson on a UAV with the low-cost radar from MIT [2]. While the software has been developed for the Jetson platform, it could also be used on other CUDA capable platforms if the power budget constraint can be relaxed.

References

[1] Francesco Baralli, Michel Couillard, Jesus Ortiz, Darwin G Caldwell, "GPU-based real-time synthetic aperture sonar processing on-board autonomous underwater vehicles", OCEANS-Bergen, June 10th 2013 MTS/IEEE.

[2] Charvat, Gregory, Jonathan Williams, Alan Fenn, Steve Kogon, and Jeffrey Herd. RES.LL-003 "Build a Small Radar System Capable of Sensing Range, Doppler, and Synthetic Aperture Radar Imaging", January IAP 2011. (MIT OpenCourseWare: Massachusetts Institute of Technology), *http://ocw.mit.edu/resources/res-ll-003-build-a-small-radar-system-capable-of-sensing-range-doppler-and-synthetic-aperture-radar-imaging-january-iap-2011*.

[3] CUDA Toolkit, *http://developer.nvidia.com/cuda-toolkit*

[4] CUFFT Library, *http://docs.nvidia.com/cuda/cufft*

[5] THRUST Library, *http://docs.nvidia.com/cuda/thrust*

[6] Massimiliano Fatica and Won-Ki Jeong,"Accelerating MATLAB with CUDA", HPEC 2007, September 18-20 2007, Lexington Massachusetts.