U.P.B. Sci. Bull., Series C, Vol. 73, Iss. 2, 2011 ISSN 1454-234x

PERFORMANCES STUDY OF TOMOGRAPHIC RECONSTRUCTION IMPLEMENTED ON NVIDIA GRAPHIC PROCESSOR SYSTEMS

Adrian SIMA[1]

Procesul de reconstrucție tomografică necesită un cost computațional foarte mare datorită procedurii de "proiecție înapoi" (backprojection). În acest articol am implementat un astfel de algoritm de reconstrucție pentru o geometrie paralelă 2D a razelor X. Implementarea a fost realizată în 4 variante: utilizând o singură unitate de calcul, 4 unități de calcul (Intel Q6600 la 2.4 GHz şi 8GB memorie ram) împreună cu tehnologia OpenMp, o placă video ce este compatibilă cu tehnologia CUDA şi utilizează memoria globală a acesteia şi ultima variantă aceeaşi placă video însă de data aceasta utilizând memoria texturilor. Atât implementările pe CPU, cât şi cele pe placa grafică folosesc datele în simplă precizie. Cel mai bun timp de calcul a fost obținut în cazul utilizării plăcii video cu memoria texturilor.

Tomographic reconstruction requires a very high computational cost due to its time-consuming backprojection step. In this paper, we implement 2D parallel-beam backprojections on single core processo,on four cores of the same CPU using OpenMP technology and a graphics card Nvdia Tesla 870C using global memory and textures memory CUDA (Compute Unified Device Architecture) architecture. For parallel beam reconstruction, the pixel-driven backprojection program running on the CPU and GPU uses single-precision binary floating-point arithmetic and linear interpolation. With the help of the texture unit we obtained the best computation time.

Keywords: CT reconstruction, Backprojection, Parallel beam, GPU, CUDA

1. Introduction

In tomographic reconstruction, exact or approximate, analytical or iterative algorithms have been developed and implemented. Most of these algorithms use backprojection as a core operation which dominates the computational cost. Generally, the backprojection is a straightforward pixel-driven operation, which has few dependencies among different pixels and is computed as an array operation within a loop. With high memory bandwidth, programmable graphics processors can execute this operation at high speeds. Graphics applications also typically consist of largely independent computations and data-intensive

[1] PhD student, National Institute for Lasers, Plasma and Radiation Physics, Bucharest-Magurele, Romania, e-mail: adi.sim@gmail.com

operations. Accelerating the backprojection step on graphics hardware was studied since 1990s [1], which employed texture mapping hardware of the graphics processor for the acceleration of tomographic reconstruction and the evolution of floating points pc graphics processing unit (GPU). With the evolution of graphics processors, the speed of reconstruction has been improved; however, their limited capabilities and inadequate programming model were a barrier to increasing the speed of reconstruction.

A recent development in this viewpoint is the release of CUDA by NVidia Corporation, which is a new hardware and software architecture for issuing and managing general computations on the GPU as a data-parallel computing device [2]. The aim of this paper is to implement 2D parallel beam backprojection algorithms through Intel uni and quad processors and CUDA for the Tesla C870 GPU and to benchmark their performances. This GPU is is organized into 16 highly threaded Streaming Multiprocessors (SMs). A pair of SMs form a building block (Fig. 1). Each SM has 8 streaming processors (SPs), for a total of 128 (16*8). Each SP has a multiply-add (MAD) unit, and and an additional multiply (MUL) unit, all running at 1.35 gigahertz (GHz). This means that there are almost 367 gigaflops for the MADs and a total of over 500 gigaflops if you include the MULs as well. In addition, special function units perform floating point functions such as SQRT and RCP SQRT as well as transcendental functions. Each GPU currently comes with 1.5 megabytes of DRAM. This memory space differ from the system memory DIMM DRAMs on the motherboard because it is essentially the frame buffer memory that is used for graphics. For graphics applications, it hold high-definition video images, and texture information for 3D rendering as in games. But for computing, it works like very high bandwidth off-chip cache, though with somewhat more latency regular cache or system memory. If the chip is programmed properly, the high bandwidth makes up for the large latency.

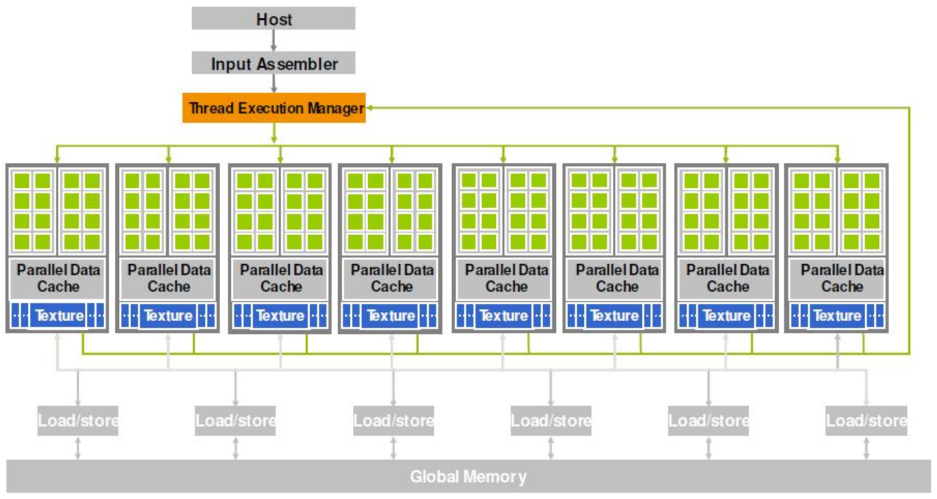


Fig. 1. Architecture of a CUDA-capable GPU

2. CUDA programming model

From a point of viewof a CUDA programmer, the computing system consists of a *host* that is a traditional Central Processing Unit (CPU), such an Intel Architecture microprocessor in personal computers, and one or more *devices* that are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, there are often program sections that exhibit rich amount of data parallelism, a property where many arithmetic operations can be safely performed on program data structures in a simultaneous manner.

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. The program supplies a single source code encompassing both host and device code. The NVIDIA C Compiler (NVCC) separates the two. The host code is straight ANSI C code and it is compiled with the host's standard C compilers and it runs as an ordinary process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures. The device code is typically further compiled by the NVCC and executed on a GPU device. The kernel functions, or simply kernels, typically generate a large number of threads to exploit data parallelism. CUDA threads are of much lighter weight than the CPU threads. CUDA programmers can assume that these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically take thousands of clock cycles to generate and schedule.

The execution of a typical CUDA program is illustrated in Fig. 2. The execution starts with host (CPU) execution. When a kernel function is invoked, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a *grid*. Fig. 2 shows the execution of two girds of threads. When all threads of a kernel complete their execution, the corresponding grid terminates, the execution continues on the host until another kernel is invoked.

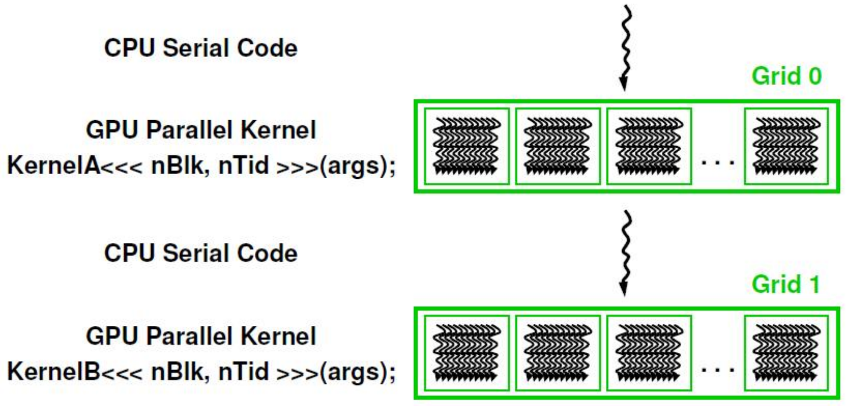


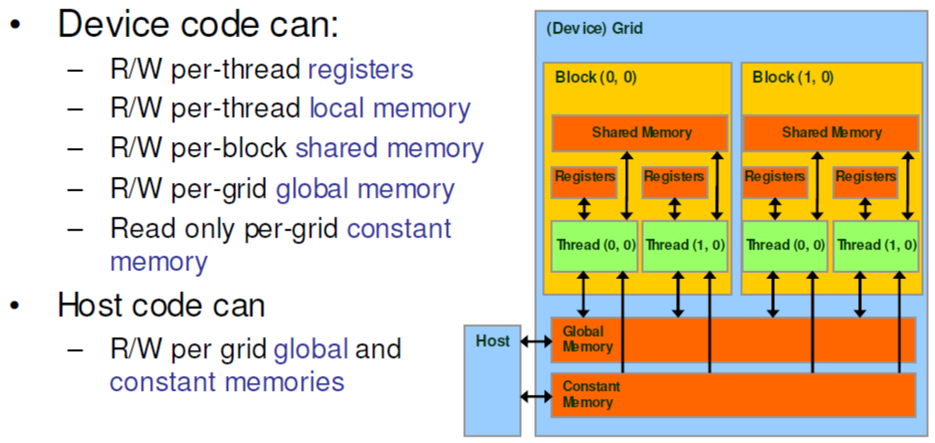
Fig. 2. Execution of a CUDA program

In CUDA, host and devices have separate memory spaces. This reflects the reality that devices are typically hardware cards that come with their own Dynamic Random Access Memory (DRAM). For example, the NVIDIA TESLA C870 card that we use
comes with 1500 MB (million bytes, or mega-bytes) of DRAM. In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer the data from the host memory to the allocated device memory. Similarly, after device execution, the programmer needs to transfer result data from device back to the host and free up the device memory that is no longer needed. The

CUDA runtime system provides Application Programming Interface (API) function calls to perform these activities.

Fig. 3 shows an overview of the CUDA device memory model and the various memory types available on a device. At the bottom of the picture, we see global memory and constant memory. These are the memories that the host code can write (W) to and read (R) from. Constant memory allow read-only access by the device. The concept CUDA memory model is supported by the API functions that can be used for allocating and de-allocating device Global Memory.

Once a program has allocated device memory for the data objects, it can request that data be transferred from the host to the device memory. This is accomplished by calling one of the CUDA API functions for data transfer between memories. Next step is the invocation of kernel functions. In CUDA, a kernel function specifies the code to be executed by all threads of a parallel phase. Since all threads of a parallel phase execute the same code, CUDA programming is an instance of the well-known Single-Program Multiple-Data (SPMD) parallel programming style, a popular programming style for massively parallel computing systems.



- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - R/W per grid global and constant memories

Fig. 3. Overview of the CUDA device memory model

The second notable extension to ANSI C is the reference to the thread indices of a thread. All threads execute the same kernel code. There needs to be a mechanism to allow them to distinguish themselves and direct themselves towards the particular parts of the data structure they are designated to work on. These keywords allow a thread to access the hardware registers associated with it at runtime that provides the identity to the thread. When a kernel is invoked, *or launched*, it is executed as *grid* of parallel threads.

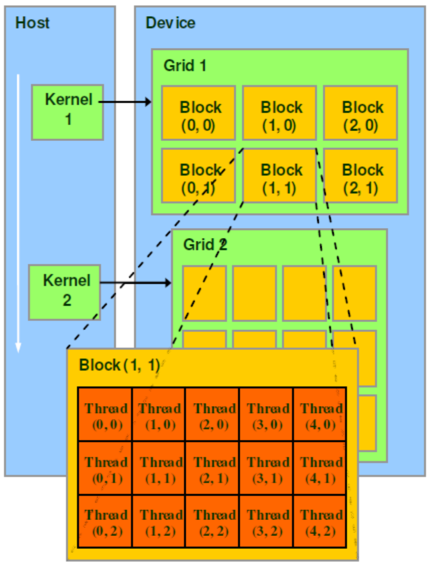


Fig. 4.CUDA thread organization

In Fig. 4, the launch of Kernel 1 creates Grid 1. Each CUDA thread grid typically comprises thousands to millions of lightweight GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism. Threads in a grid are organized into a two-level hierarchy, as illustrated in Fig. 4. For simplicity, the number of threads shown in Fig. 4 is set to be small. In reality, a grid will typically consist of many more threads. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads. In Fig. 4, Grid 1 consists of 6 thread blocks that are organized into a 2x3 two-dimensional array of threads. Each thread block has a unique two dimensional coordinate given by the CUDA specific keywords. All thread blocks must have the same number of threads organized in the same manner. Each thread block is in turn organized as a three dimensional array of threads with a total size of up to 512 threads. The coordinates of threads in a block are uniquely defined by three thread indices. Not all applications will use all the three dimensions of a thread block. In Fig. 4, each thread block uses only two of the dimensions and is organized into a 3x5 array of threads. This gives Grid 1 a total of 15*6=90 threads.

3. Tomographic reconstruction principle. Implementation of parallel beam algorithm

X-ray Computed Tomography (X-CT) is a nondestructive technique for visualizing interior features within solid objects, and for obtaining digital information on their geometries and properties. X-CT imaging consists of directing X-rays through an object from multiple orientations and measuring the decrease in intensity along a series of linear paths. For a monochromatic X-ray energy beam through a homogeneous material this decrease is characterized by Beer's Law,

$$I = I_0 \cdot e^{-\mu \cdot d} \tag{1}$$

where I_0 and I are the initial and final X-ray intensity, μ is the material's linear attenuation coefficient (units 1/length) and d is the length of the X-ray path.

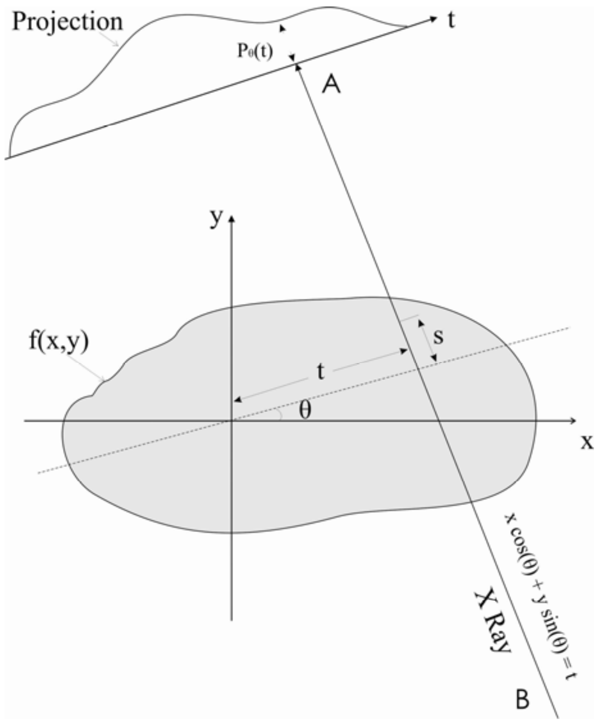


Fig. 5. An object, f(x, y),and its projection, $P_\theta(t)$, are shown for an angle of θ

In this paper, we study the reconstruction tomography algorithm for parallel geometry [2]-[4]. Projections of the object are captured by a one-dimensional sensor. For each angle θ (θ = 1..180) a projection of the object is acquired from the sensor(Fig. 5 - $P_\theta(t)$). After the measurement it will result a picture (sinogram) with the number of pixels of the sensor size - lengthwise - and the number of angles used - the width. The image obtained consists of various levels of gray. The gray levels in the sinogram correspond to X-ray attenuation, which reflects the proportion of X-rays scattered or absorbed as they pass through the object. X-ray attenuation is primarily a function which depends on X-ray energy source and composition of the studied material.

To achieve the object image is necessary to implement a backprojection algorithm. The Filtered Back Projection algorithm uses Fourier theory to arrive at a closed form solution to the problem of finding the linear attenuation coefficient at various points in the cross-section of an object. A fundamental result linking Fourier transforms to cross-sectional images of an object is he Fourier Slice Theorem. Let x,y represent the coordinates inside the object (Fig. 5), and f(x,y) the density (attenuation coefficient) in rectangular co-ordinates of the object under consideration at the cross-section at which the imaging has to be done. Let equation (3) represent the projection of the object at distance, t, from the center. The equation of the line AB is:

$$t = x\cdot\cos\theta + y\cdot\sin\theta \qquad (2)$$

Then, the projections are defined as:

$$P_\theta(t) = \int_{(\theta,t)line} f(x,y)\cdot ds \qquad (3)$$

It has been shown [3] that the above equation can be written using a delta function as:

$$P_\theta(t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\cdot\left[\delta(x\cdot\cos\theta + y\cdot\sin\theta - t)\cdot dx\cdot dy\right] \qquad (4)$$

This function is known as the Radon transform of f(x,y) [3]. A set of these functions for constant angle are the projections of the object at the cross-section.

$$f(x,y) = \int_{0}^{\pi}\int_{-\infty}^{\infty}\left[S_\theta(w)\cdot|w|\cdot e^{j2\pi(x\cos\theta + y\sin\theta)}\cdot dw\right]\cdot d\theta \qquad (5)$$

In the equation (5), the terms inside the square brackets (the operation indicated by the inner integral) represents a filtering operation and evaluate the filtered projections, and the operation being performed by the outer integral evaluate the back-projections, which basically represents a smearing of the filtered projections back on to the object and then finding the mean over all the angles.

The filtered back projection algorithm can therefore be thought of as a three step process:

1. Finding the Fourier Transform (FT) in 1D of the projections.
2. Finding the filtered projections. This essentially means multiplying the results of step 1. with a response of low pass filtering function in the frequency domain, and then finding the inverse Fourier Transform (IFFT). This step is essentially the same as carrying out convolution in the time domain. It can be represented mathematically as:

$$Q_\theta(t) = \int_{-\infty}^{\infty} S_\theta(w) \cdot |w| \cdot e^{j2\pi wt} \cdot dw \tag{6}$$

3. Finding the back projections. This step is the smearing of the filtered projections back on to the object, and is mathematically represented by:

$$f(x,y) = \int_0^\pi Q_\theta(x\cos\theta + y\sin\theta) \cdot d\theta \tag{7}$$

These three steps represent the filtered back projection algorithm(Fig. 6). In the discrete domain, the algorithm changes only slightly. The important steps are outlined below:

1. Find the 1D FT of the projections for each angle.
2. Multiply the result of step 1 above with the response of low pass filtering function in the frequency domain (equivalent to convolving with the response function in the time domain).
3. Find the IFFT of the results in step 2. This gives us the filtered projections in the discrete domain and correspond to Q(n), where the Q's are taken at the various angles at which the projections were taken, and "n" is the ray number at which the line projection was taken.
4. Back-project. The integral of the continuous time system now becomes a summation, and we get:

$$f(x,y) = \frac{\pi}{N_p} \cdot \sum_{i=1}^{N_p} Q_{\theta_i}\left(x\cos(\theta_i) + y\sin(\theta_i)\right) \tag{8}$$

where *f*(*x, y*) is the image and Q_θ is the filtered projection data, θ_i for i=1, 2, ..., N_p are the angles of the measured projections since the projection angles are also just available as discrete values. The normalization in front of the sum in (9) is accomplished by the discretization of the angle element

$$d\theta \to \Delta\theta = \frac{\pi}{N_p} \tag{9}$$

Projections filtering can be done with the filters Ram-Lak, Shepp-Logan, Hamming, Hanning, Cosine, both in Fourier space and Cartesian space.

It should be noted here that (x,y) are chosen by the program while back projecting. So the value of $x\cos\theta + y\sin\theta$ may not correspond exactly to a value of "n" for the filtered projections which may have been calculated in the previous step. Therefore interpolation has to be done, and usually linear interpolation is quite sufficient [3]. It may be noted that in terms of computational time, this step consumes the maximum time (about 80%).

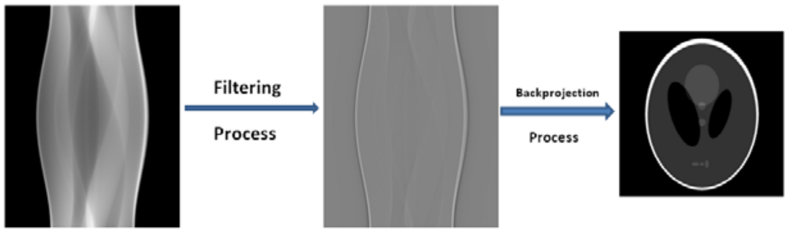


Fig. 6. The filtered back projection algorithm

Correlation coefficient between reconstructed image and the well-known "head phantom" is a measure of reconstruction quality(10).

4. Implementations of backprojections algorithm-parallel beam and the obtained performances

For parallel beam reconstruction, we used the pixel-driven backprojection programs running on the CPU and GPU with single-precision binary floating-point arithmetic and linear interpolation. It can be seen from the publications that GPUs are widely used for general purpose computing, beyond the original target of computer graphics and gaming industry. The earliest attempt to accelerate CT reconstruction using graphics hardware dates back to 1994, when Calbral et al. utilized the texture mapping hardware for reconstruction [1]. With the introduction of modern GPUs in the last 5 years, both analytical and iterative methods have been implemented on graphics hardware.

In this work all data from these programs are used in single precision for reasons of compatibility with video card([5]-[7]) and the projections were filtered using Shepp-Logan filter type.

A. Single core CPU implementation

In the bellow program, as in the following, *N* is the number of pixels of the image that we reconstruct, *m* represents the number of angles and *n* is the size of linear detector. Thus we have the size of projection image [n, m]. In the vectors R and IMG are stored the filtered projections and reconstructed image, respectively.

```
for each number of angles ($\theta = 0 : N_p$)
    for each pixel in reconstructed image on length ($i_x = 1:N$)
        for each pixel in reconstructed image width ($i_y = 1:N$)
            $x = i_x - N/2;$
            $y = i_y - N/2;$
            $t = y*\cos(\theta) - x*\sin(\theta);$
            $f(x,y) = f(x,y) + q(\theta);$
```

```
                    end
            end
    end
    for each pixel in reconstructed image on length ($i_x$ = 1:N)
                    for each pixel in reconstructed image width ($i_y$ = 1:N)
                            f(x,y) = f(x,y) * $\pi/N_p$
                    end
    end
```

The result of this program is the reconstructed image.

B. 4 cores CPU implementation using OpenMP

The quad-core processors together with OpenMP[6] technology allowed to obtain a reduced computation time(Table 1). By using of OpenMP technology, each "for" loop was was split up between all 4 threads.

```
    starting parallel zone with shared (f(x,y))
            omp for
                    for each number of angles ($\theta$ = 0 : $N_p$)
                            for each pixel in reconstructed image on length ($i_x$ =
    1:N)
                                    for each pixel in reconstructed image width
    ($i_y$ = 1:N)
                                            x = $i_x$ - N/2;
                                            y = $i_y$ - N/2;
                                            t = y*cos($\theta$) - x*sin($\theta$);
                                            f(x,y) = f(x,y) + q($\theta$);
                                    end
                            end
                    end
            omp for
                    for each pixel in reconstructed image on length ($i_x$ = 1:N)
                                    for each pixel in reconstructed image width
    ($i_y$ = 1:N)
                                            f(x,y) = f(x,y) * $\pi/N_p$
                                    end
                    end
    end parallel zone
```

C. A first CUDA implementation using global memory

The programming model changed with implementation of the reconstruction algorithm on video card with CUDA support. As mentioned in section 2 of this paper, the main point of this equipment is to create and use a large number of threads that are grouped into blocks. This execution blocks form the grid. These threads are executed in parallel by the multiprocessors available on video card. Thus each pixel of the reconstructed image can be represented (parallel computed) by such a thread. Comparing the code implemented on a single CPU with the one implemented on the graphics card we can see that the last two main "for" loops representing the reconstructed image pixel indices (*for (ix = 0; ix <N; ix = ix +1)* and *for (iy = 0; iy < N, iy = iy +1)*) have disappeared. They were replaced with statement of grid of threads blocks that are the indices of the pixels of the reconstructed image:

```
int x = blockDim.x * blockIdx.x + threadIdx.x;
int y = blockDim.y * blockIdx.y + threadIdx.y;
```

Variable *blockDim.x* and *blockDim.y* " represent the threads block size on the x and y axis. "blockIdx.x" and "blocIdx.y" are variables designating indices of the blocks of threads within the grid. Grid can have no more than two dimensions: *blockDim.x* ,*blockDim.y*. Indices within each thread block are represented by the variables: *threadIdx.x, threadIdx.y*. So for an image reconstruction will now use a grid of threads of execution that will have the same size as the reconstructed image and each pixel of the image will be computed by one thread of execution. For the samples code C and code D we used the variable *imgD_f* for storage of projections and *imgD* for reconstructed image. The grid size is given in main program, before starting of reconstruction subroutine.

```
dim3 bloc_i(16,16,1);
dim3 grid_i(N / bloc_i.x, N / bloc_i.y);
interpolare<<<grid_i,bloc_i>>>(...);
```

Dimensions of a execution block are established in the main program, in our case, this block is composed of a 16x16 threads. Grid itself is also a structure formed from such blocks. Depending on the desired reconstructed image size, grid size may vary. In our case this grid is N / 16 x N / 16 (we have to reconstruct an image consisting of NxN pixels, and one single block can reconstruct a sub imagine of 16x16 pixels). Because we used a global memory video card, the program performance is not remarkable (Table 1) and we are referring here to the reconstruction time.

```
GPU function declaration
    x = gpu x grid declaration
    y = gpu y grid declaration
    for each number of angles ($\theta = 0 : N_p$)
        xx = x + 0.5 - N/2;
        yy = y + 0.5 - N/2;
        t = yy *cos(θ) - xx*sin(θ);
        sum = sum + q(θ);
    end
```

```
        sum = sum * $\pi/N_p$
        f(x,y) = sum;
    end Gpu function
```

D. CUDA Implementation using texture space memory

Another way is to implement this algorithm using texture memory. The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. Another quality of the texture unit is that it knows how to make an interpolation of data: nearest neighbor and linear interpolation. So we tried an improvement of reconstruction time using texture memory for storing projection data already filtered. Computation of texture address was performed by texture unit using a linear interpolation method. Because of these hardware abilities the reconstruction time was substantially improved (Table 1) comparing with the other implementations.

We attached to the texture unit a CUDA array in which is stored the filtered projections. We used a two-dimensional texture (N pixels on Ox and Oy N pixels) and the mode of address was set on "clamp." Also we used texture in the non-normalized form.

```
    2D GPU texture unit declaration
    GPU function declaration
        x = gpu x grid declaration
        y = gpu y grid declaration
        tex _x = x - N/2;
        tex_y  = y - N/2;
        for each number of angles ($\theta = 0 : N_p$)
            t =  tex_y * cos() - tex_x *sin();
            sum = sum + unit_texture2D(t, $\theta$)
        end
        sum = sum * $\pi/N_{p;}$
        f(x,y) = sum;
    end Gpu function
```

5. Results

Backprojections alogorithm is used for achieving of the reconstructed images from the filtered projections with the following sizes: 711x1800, 1419x1800, 1453x1800, 2833x1800, 2901x1800 and 5797x1800 pixels. They represent both the number of angles at which projections were taken from 0 to

179.9 with a step of 0.1 degree and the number of pixels of a single projection. The obtained reconstructed images have the following sizes: 500x500, 1000x1000, 1024x1024, 2000x2000, 2048x2048, 4000x 4000 pixels and represent Sheep-Logan Phantom. For implementation of the backprojection algorithm we used a single core processor (Intel Q6600 2.4 GHz, 8 GB RAM.), four cores of the same CPU using OpenMP technology, a graphics card Nvidia Tesla 870C using only global memory and textures memory.

Table 1 presents both reconstruction times of the initial phantom image and correlation coefficient of the reconstructed image compared with original image. We mention that the computing time was registered since the beginning of the program until its full completion; this time includes the time for calculating the reconstruction image and the time required to achieve operations and read / write of the data input / output Image correlation factor obtained in relation to the initial phantom image represents the percentage of similarity of two images. This factor was calculated by the following formula for each reconstruction in part:

$$corr = \frac{N_p^2 \cdot \sum_i \left(f_i^{rec} \cdot f_i^{ph}\right) - \sum_i \left(f_i^{rec}\right) \cdot \sum_i \left(f_i^{ph}\right)}{\left[N_p^2 \cdot \sum_i \left(f_i^{rec}\right)^2 - \left(\sum_i f_i^{rec}\right)^2\right]^{0.5} \cdot \left[N_p^2 \cdot \sum_i \left(f_i^{ph}\right)^2 - \left(\sum_i f_i^{ph}\right)^2\right]^{0.5}} \tag{10}$$

Where f_i^{rec} represents the pixel value of reconstructed image, f_i^{ph} is the pixel value from the phantom image and N_p is the total number of pixels.

Table 1

Reconstruction times of the initial phantom image and correlation coefficient of the reconstructed image compared with original image.

SIZE of parallel projection /reconstructed image	CUDA - texture reconstruction time[ms] / correlation factor	CUDA - global memory reconstruction time[ms] / correlation factor	CPU - 4 cores with OpenMP reconstruction time[ms] / correlation factor	CPU - 1 core reconstruction time[ms] / correlation factor
S_711_1800/I_500_500	109 / 0.96	531 / 0.98	1546 / 0.95	5812 / 0.95
S_1419_1800/I_1000_1000	328 / 0.98	2062 / 0.99	6187 / 0.97	23500 / 0.97
S_1453_1800/I_1024_1024	343 / 0.98	2140 / 0.99	6468 / 0.97	24671 / 0.97
S_2833_1800/I_2000_2000	1156 / 0.99	8078 / 0.99	24500 / 0.99	93656 / 0.99
S_2901_1800/I_2048_2048	1203 / 0.99	8515 / 0.99	26000 / 0.99	98500 / 0.99
S_5797_1800/I_4096_4096	4656 / 0.99	34390 / 0.99	102546 / 0.99	389531 / 0.99

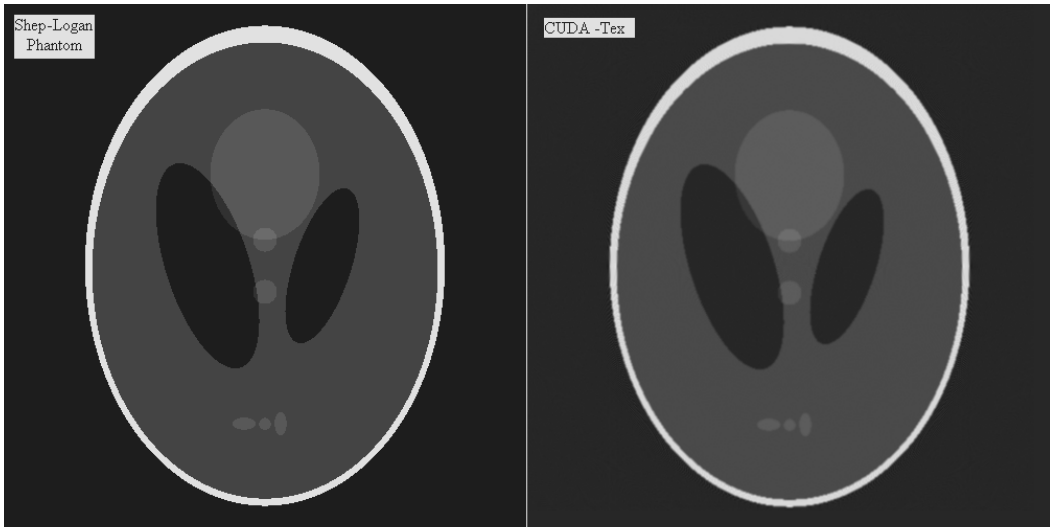


Fig. 7. Reconstruction results and initial Shepp-Logan Phantom

One can observe that the best time of reconstruction is achieved if the texture memory is used; from the image above (Fig. 7) and the study of correlation coefficients result that no visible differences between accomplished reconstructions in the four cases and initial Shepp-Logan Phantom. It changes the reconstruction time, which is 83 times lower for GPU texture in comparison with the single CPU computation for a 4096 x 4096 pixel image. In Table 2 we present the accelerating factors of the reconstruction time for each size of reconstructed image separately. It was assumed that the reconstruction values for GPU - texture is set to 1.

Table 2

Accelerating factors of the reconstruction time for each size of reconstructed image separately

SIZE of reconstructed image	CPU - 1 core	CPU - 4 cores with OpenMP	CUDA - global memory	CUDA - texture
I_500_500	1	4.87	14.18	53.32
I_1000_1000	1	6.27	18.86	71.64
I_1024_1024	1	6.24	18.86	71.93
I_2000_2000	1	6.99	21.19	81.02
I_2048_2048	1	7.08	21.61	81.88
I_4096_4096	1	7.38	22.02	83.66

6. Conclusion

In this work we made a comparison quality / reconstruction time to implement a parallel beam tomographic reconstruction from different programming environments CPU, GPU. All reconstructions were performed with single-precision binary floating-point arithmetic and linear interpolation of the projections data. Best reconstruction time was obtained for CUDA with texture memory. Though it has been used a video card with average performances (TESLA C870 - using the first version of CUDA) it obtained an acceleration by a factor of 83x over the time of image reconstruction made on a powerful computer.

R E F E R E N C E S

[1] *B. Cabral, N. Cam, J. Foran*, “Accelerated volume rendering and tomographic reconstruction using texture mapping hardware,” 1994 Symp. Volume Visualization, pp. 91-98, 1994

[2] *Jiang Hsieh*, "Computed Tomography: Principles, Design, Artifacts, and Recent Advances, Second Edition", SPIE Publications, 2009

[3] *Avinash C. Kak, M. Slaney,* "Principles of Computerized Tomographic Imaging", ch. 3 http://www.slaney.org/pct/pct-toc.html

[4] *D. Kirk, Wen-mei Hwu,* "Programming Massively Parallel Processors: A Hands-on Approach", Elsevier Science Ltd 2010

[5] *J. Sanders, E. Kandrot,* "CUDA by Example. An Introduction to General-Purpose GPU Programming", Addison-Wesley, 2010

[6] NVidia Corporation, “NVidia CUDA Compute Unified Device Architecture Programming Guide” http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[7] NVidia Corporation, http://www.nvidia.com/docs/IO/43395/C870-BoardSpec_BD-03399-001_v04.pdf

[8] *T. Mattson, et al* “Patterns for Parallel Programming,” Addison Wesley, 2005.