GPU Acceleration of a Fully 3D Iterative Reconstruction Software for PET using CUDA

J. L. Herraiz, S. España, S. García, R. Cabido, A. S. Montemayor,
M. Desco, J. J. Vaquero, and J. M. Udias

***Abstract*–**A CUDA implementation of the existing software FIRST (Fast Iterative Reconstruction Software for (PET) Tomography) is presented. This implementation uses consumer graphics processing units (GPUs) to accelerate the compute-intensive parts of the reconstruction: forward and backward projection. FIRST was originally developed in FORTRAN, and it has been migrated to C language to be used with NVIDIA C for CUDA, as well as for a straightforward implementation and performance comparison between the C versions of the code running on the CPU and on the GPU. We measured the execution time of the CUDA version compared to the fastest available CPU. The CUDA implementation includes a loop re-ordering and an optimized memory allocation, which improves even more the performance of the reconstruction on the GPUs.

I. Introduction

TOMOGRAPHIC reconstruction is computationally very demanding, specially when iterative methods based on realistic models of the emission and detection of the radiation are used [1]. Modern multi-core processors have reduced the computational time required to run tomographic reconstruction codes thanks to parallel task execution. Tomographic reconstruction codes can be easily parallelized, as the two main time-consuming parts of the reconstruction task (forward and backward projection) can be recoded into SIMD (single instruction multiple data) tasks and distributed among the available processor units, assigning some parts of the data to each of them. This can be done as the acquisition data are arranged in sets that are largely independent among them. FIRST [1], a Fast Iterative Reconstruction Software for (PET) Tomography was developed in our group using this strategy, and it is been used on several commercial preclinical systems [5].

Manuscript received October 27, 2009. This work was supported in part by MEC (FPA2007-62216), CDTEAM (Programa CENIT, Ministerio de Industria), UCM (Grupos UCM, 910059), CPAN (Consolider-Ingenio 2010) CSPD-2007-00042 and the RECAVA-RETIC network.

J. L. Herraiz and J. M. Udias are with the Grupo de Física Nuclear, Dpto. Física Atómica, Molecular y Nuclear, Universidad Complutense de Madrid, Spain (telephone: +34-91-394-4484, email: joaquin@nuclear.fis.ucm.es).

S. España was with the Grupo de Física Nuclear,Universidad Complutense de Madrid, Spain. He is now with the Department of Radiation Oncology, Massachusetts General Hospital and Harvard Medical School, Boston, MA, USA (S.España e-mail: samuel@nuclear.fis.ucm.es)

S. García, R. Cabido and A.S. Montemayor are with the Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, Madrid, Spain (A.S. Montemayor e-mail: antonio.sanz@urjc.es).

M. Desco and J. J. Vaquero are with the Unidad de Medicina y Cirugía Experimental, Hospital General Universitario Gregorio Marañón, Madrid, Spain (J.J. Vaquero e-mail: juanjo@hggm.es).

Nevertheless, the increasing complexity of new scanners with larger number of response lines and reconstructed voxels, as well as more sophisticated acquisitions protocols, like dynamic studies or PET acquisitions with time-of-flight information, requires new approaches in order to maintain reasonable reconstruction times without the need of large computer clusters.

The Graphics Processing Unit (GPU) can handle large data sets in parallel when working in single instruction multiple data (SIMD) mode. GPU development has been much faster than CPU one and nowadays the processing capability of GPU is considerably superior to the CPU one. The recent advances in the programmability of GPU has made it possible that certain general purpose computations usually done at the CPU can be implemented on GPU with a much faster speed. CUDA [2], developed by NVIDIA, offers a unified hardware and software solution for parallel computing on CUDA-enabled NVIDIA GPUs supporting the standard C programming language together with high performance computing numerical libraries aimed to ease the job of coding and running complex computational problems on the GPU. When programmed through CUDA, the GPU can be viewed as a computing device capable of executing a very high number of threads in parallel, and the power of the GPU for handling multiple instances of the same operation on multiple data is transparently accessed.

Attempts to code tomographic reconstruction for CT and PET have been made in the past [3], [4] also with CUDA, but usually from scratch, using different algorithm and codes that would have been employed on the CPU. Thus, a direct comparison of performance as well as the resulting images between GPU and CPU codes is difficult.

FIRST has proved to be a successful implementation of a tomographic code for high-resolution small animal PET scanners [1], [6]. It is based on a realistic model of the scanner obtained from the Monte Carlo simulation code PeneloPET [5]. This model improves the image quality. For example, it obtains a more uniform resolution along the FOV. Some of its key advantages are that it implements a method for handling the large size of the model to fit into RAM memory, improving the performance of the code. It is also coded to run in parallel in an arbitrary number of processors and further the number of pixels of the reconstructed image can be chosen freely.

In this work we aimed to obtain a straightforward implementation of the same code, targeted for CPU optimized execution, into the GPU. Our main goal was to obtain a significant acceleration of the code without compromising the

quality of the reconstructed images. Additionally we wanted to obtain a simple and flexible code, without specific optimizations, which would allow for possible future modifications without much additional effort.

The program was written to a large extent independently on the NVIDIA GPU model that will execute the code. However, attention was paid to considerations about memory allocation and access, in order to achieve optimal performance on GPUs.

II. Materials and Methods

A. Description of the CPU Implementation

For a more detailed description of the reconstruction code FIRST, the reader is referred to [1]. We will describe here its main components and characteristics for the sake of completeness. The code implements a fully-3D iterative reconstruction of PET data based on a realistic model of the radiation emission and detection.

This model was generated using the Monte Carlo code PeneloPET [5] based on PENELOPE [7]. In order to fit these probability distributions called system response matrix (SRM) into RAM memory, all symmetries present in our system were exploited. This imply that only some probability distributions along the LOR (commonly called Tubes-of-Response, TOR) have to be computed and stored, as symmetrically-equivalent LORs will have the same probability distribution, as shown in Fig. 1. Even with this approach, the size of the probability coefficients exceeded the available RAM memory of common computers. This was solved using what we called quasi-symmetries [1]: LORs with very similar angle respect to the scintillator crystals will have a very similar probability distribution, as shown in fig. 1. Therefore, probability distributions along chosen LORs, called Super-LORs, were simulated and stored, reducing the initial storage requirements by more than a factor 20 without compromising the quality of the reconstructed images.

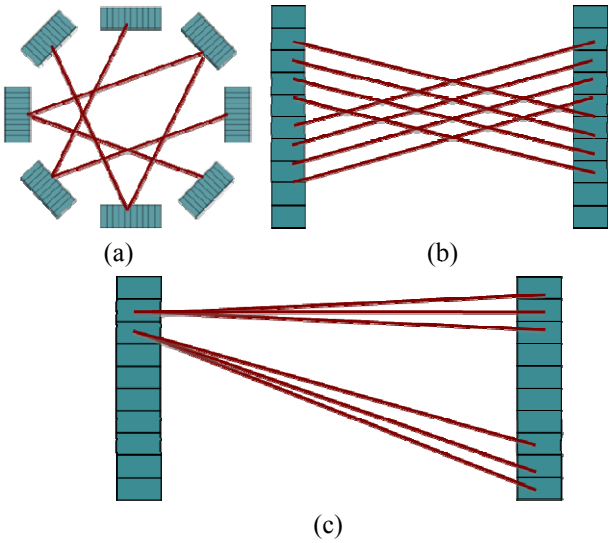

Fig. 1 – Symmetries: (a) Rotational (b) Translational and Reflections, and (c) Quasi-symmetries, used to reduce the number of LORs that needs to be stored in memory. All these LORs have a similar probability distribution.

In the reconstruction algorithm used in this work, called OS-EM [8], the reconstructed image X_j is iteratively multiplied by a weighted average of a subset S of corrective factors (Eq. 1). These factors are obtained as the ratio of the measured data Y_i and the data estimated P_i, from the image and the weighting factors C_{ij} comes from the SRM.

$$X_j^{(t+1)} = X_j^{(t)} \cdot \frac{\sum_{i \epsilon S} C_{ij} \cdot \frac{Y_i}{P_i}}{\sum_{i \epsilon S} C_{ij}} \quad ,, \; P_i = \sum_{i \epsilon S} C_{ij} \cdot X_j^{(t)} \qquad (1)$$

For this comparison, a version of the code suited for a high-resolution small animal PET scanner with a pair of rotating detectors in coincidence. The number of voxels in the reconstructed images was 175x175x59 and the number of bins in the sinogrammed data were 175 (radial bins) x 130 (angles) x 900 (oblique sinograms). The data used in this study came from a PeneloPET simulation of a Derenzo-like phantom, with rods of different diameters filled with activity. The original code, written in FORTRAN, was translated into ANSI C. In both cases the Intel compiler [9] was used and the best optimization options available were chosen. The execution speed of both the FORTRAN and C version was similar.

In this work the time required for reconstructing one image was studied for one single CPU. Nevertheless we expect good parallelization as reported in previous works [1], so the reconstruction time required in a cluster of computer can be easily estimated.

B. Description of the GPU implementation with CUDA

With the large number of threads that can be used for parallel computation in GPUs, the usual bottleneck in these kind of implementations is memory accession. Table I describes the different types of memory available for the GPU and exposed by CUDA. It can be seen that access to registers and textures is much faster than access to global memory.. In recent versions of CUDA, 3D textures became available which makes the implementation of our code straightforward.

TABLE I. GPU TYPES OF MEMORY AND REQUIRED CYCLES TO ACCESS THEM

GPU Memory Type	Number of Cycles
Register	1
Shared Memory	1
Constant Memory (in cache)	1-10
Constant Memory (not in cache)	10-100
Texture Memory (in cache)	1-10
Texture Memory (not in cache)	10-100
Global Memory	400 - 600

We have defined three 3D-textures, one for the reconstructed image, another one for the System Response Matrix and a third one corresponding to the corrections obtained after comparing the data with the projections.

The system response 3D-texture uses the large number of symmetries present in the sinograms acquired from a rotating scanner, making the total number of necessary Super-LORs to be quite small.

Forward and backward projections are by far, the most time-consuming parts of the code. More than 90% of the reconstruction time is spent in these two steps. Fortunately these parts can be easily distributed in several processors as many forward or backward projections can be computed in parallel. These two subroutines were implemented as CUDA kernels, and called from the C code. The implementation is schematically described in Fig. 2:

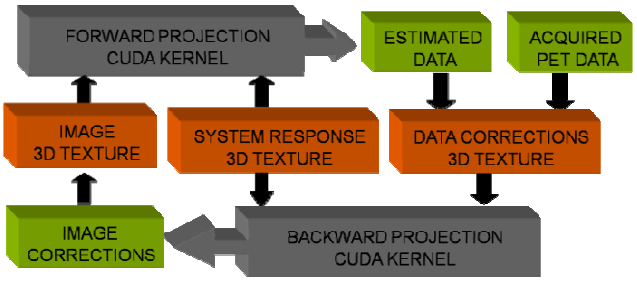

Fig. 2. Diagram of the code implemented.

In order to reach the best performance with the GPU, the code implementation should reduce memory accesses wherever possible. Therefore, the code was designed with this goal in mind, and some efforts were devoted to reorder the data to improve memory access and to rearrange loops in the iterative algorithm.

In this work we have used the 2.3 version of CUDA with the 191.07 version of the NVIDIA drivers.

C. *Forward Projection*

In the implemented CUDA kernel, each thread projects one LOR, adding the contribution from all the 175x5x3 voxels connected with it. Several angles (with all radial and axial bins) can be projected at the same time. In this work, a total of 175(radial) x 59(axial) x 10(angles) LORs are projected simultaneously. The possibility of changing the total number of LORs that can be projected simultaneously, will allow maintaining a good scalability in future more powerful GPUs.

```
For each LOR  {
 Find its corresponding Super-LOR
 (0,yc,zc) = Center of the LOR (Before rotation)
 [θ,δ] = Polar and Axial Angle of the LOR
 For each POINT (i,j,k) in the LOR  {
  (Super-LOR, i,j,k) → (XP,YP,ZP) [Probab.Index]
  VALUE_PROB = tex3D(texProb,XP,YP,ZP);
  (x0,y0,z0)=(0,yc,zc)+(i,j,k)
  (x,y,z) =Rotation[θ,δ] (x0,y0,z0)
  VALUE_IMAGE = tex3D(texImg,x,y,z);
  SUM_PROJ+=VALUE_PROB*VALUE_IMAGE;
  }
 PROJECTION[LOR]=SUM_PROJ;
}
```

Fig. 3. Pseudo-code of the Forward Projection.

Fig. 3 shows the pseudo-code of the Forward Projection algorithm.

D. *Backward Projection*

The ratio between the measured data and the projections are stored in a 3D texture of corrections with radial, axial and angular indexes. This way of storing these corrections is much more efficient than keeping them in a vector in Global Memory. Not only the values which should be back-projected are more easily accessed, but also linear-interpolation in radial and axial directions is obtained from textures without requiring additional code.

In the backward projection kernel, each thread back-projects one voxel, adding the contribution from all the 5x3x10 previously projected LORs connected with it. Due to the overlap between LORs caused by the thickness of the probability distribution, for each angle a voxel is connected with 5x3 LORs. All the corrections obtained are summed and stored into an image of corrections and an image of sensitivity, both necessary for the OS-EM algorithm.

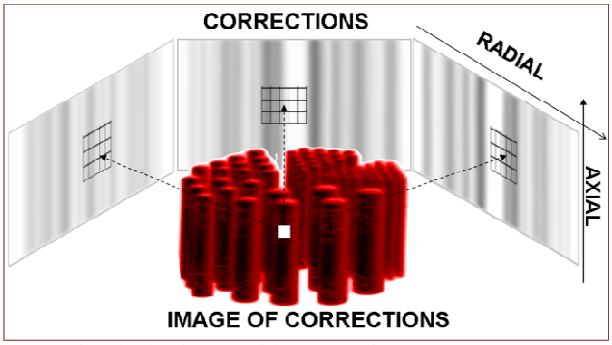

Fig. 4. Backprojection in a voxel from several LORs.

Figure 4 shows schematically the backprojection step for a voxel, where the corrections from several LORs are summed up and stored. Figure 5, describes the pseudo-code for this kernel.

```
For each LOR  {
 Find its corresponding Super-LOR
 For each Voxel  {
   (xc,yc,zc) = Voxel Coordinates
  For each projected  [θ,δ] angle  {
   (x0,y0,z0) =Rotation[-θ,-δ] (xc,yc,zc)
   (y0,z0)  Represents Detector Coordinates
   For each  (j,k) transversal point in the LOR {
   (x,y,z) = (x0,y0,z0) + (0,j,k)
   VALUE_CORR = tex3D(texCorr,y,z, [θ,δ]);
   From [θ,δ] and [y,z] → Super-LOR
   (Super-LOR, x,-j,-k)→(XP,YP,ZP)
   VALUE_PROB = tex3D(texProb,XP,YP,ZP);
   valor_imagen_corr+= valor_corr*valor_prob;
    valor_imagen_prob+= valor_prob;
   }
  }
  IMG_CORR[Voxel]+=valor_imagen_corr;
  IMG_SENS[Voxel]+=valor_imagen_prob;
}
```

Fig. 4. Pseudo-code of the Backward Projection.

III. RESULTS

Table II shows the time required for reconstructing one acquisition in different architectures. The codes for the CPU were compiled with the Intel C compiler producing 64 bit code. Times for the CPU are given for a single core. The processors where these codes were run are among the fastest in the market. The CUDA code was run in several GPUs, obtaining significant performance differences between them. The reconstructed images can be viewed in Fig. 3.

TABLE II – RECONSTRUCTION TIME FOR ONE BED, ONE-FRAME IN DIFFERENT ARCHITECTURES. THE SPEED-UP FACTOR IS COMPARED AGAINST THE FASTEST CPU.

Architecture	Time (s)	Speed-up factor
CPU–Intel® Xeon™ (3.00GHz)	2048	-
CPU–Intel® Core™ i7 (2.93GHz)	1561	1x
GPU - GT 120 1.0 GB – 32 stream processors	271	6x
GPU - 8800 GTS 640MB – 96 stream processors	95	16x
GPU - FX5600 1.5GB – 128 stream processors	65	23x
GPU - GTX260 896MB – 216 stream processors	45	35x

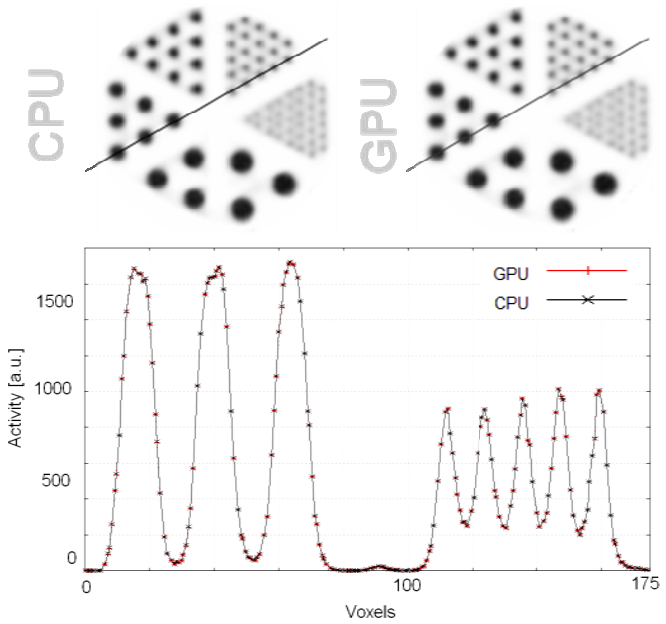

Fig. 3. Image reconstructed from CPU and GPU. A line profile through both images is also shown.

IV. CONCLUSIONS

The iterative fully-3D reconstruction software FIRST has been successfully implemented in CUDA and a significant improvement in the reconstruction time has been achieved. This is remarkable as FIRST has been shown to be already a highly optimized reconstruction code.

In can be noticed from Table II that the same code running on different GPUs can obtain significant time reductions. This is one of the main advantages of CUDA implementations, its scalability to different GPU families with no needs of re-coding.

The following generation of GPUs will probably outperform by an order of magnitude the current ones, so reconstruction software implemented on CUDA will surely benefit from these improvements without effort. Some further strategies to optimize the reconstruction code to take full advantages of GPU capabilities are current under investigation.

REFERENCES

[1] J. L. Herraiz et al., "FIRST: Fast Iterative Reconstruction Software for (PET) Tomography," Phys. Med. Biol., vol. 51, pp. 4547-4565, 2007.

[2] CUDA - Nvidia Corp.: Nvidia CUDA Programming Guide v.2.3 (2009) - http://www.nvidia.com/object/cuda_home.html

[3] G. Pratx, G. Chinn, P.D. Olcott, and C.S. Levin, "Fast, accurate and shift-varying line projections for iterative reconstruction using the GPU," IEEE Trans. Med. Imag., vol 28(3), pp. 435-445, Mar 2009.

[4] Venkatesh Bantwal Bhat, "High-Speed Reconstruction of Lowdose CT using Iterative Techniques for Image-Guided Interventions", Thesis, 2008

[5] E. España et al., "PeneloPET, a Monte Carlo PET simulation toolkit based on PENELOPE," Phys. Med. Biol. vol. 54, pp. 1723–1742, 2009

[6] E. Lage et al., "Design and performance evaluation of a coplanar multimodality scanner for rodents imaging," Phys. Med. Biol, vol. 54, pp. 5427-5441, 2009.

[7] J Baró et al., " PENELOPE: an algorithm for Monte Carlo simulation of the penetration and energy loss of electrons and positrons in matter," Nucl Inst Meth In Phy Res B, vol. 100, pp. 31-46, 1996

[8] H. M. Hudson, and R. S. Larkin, "Accelerated image reconstruction using ordered subsets of projection data," IEEE Trans. Med. Imag., vol. 13, pp. 601–609, 1994

[9] http://software.intel.com/en-us/intel-compilers