Surfing the optimization space of a multiple-GPU parallel implementation of a X-ray tomography reconstruction algorithm

Abstract

The increasing popularity of massively parallel architectures based on accelerators have opened up the possibility of significantly improving the performance of X-Ray Computed Tomography (CT) applications towards achieving real-time imaging. However, achieving this goal is a challenging process, as most CT applications have not been designed for exploiting the amount of parallelism existing in these architectures. In this paper we present the massively parallel implementation and optimization of Mangoose++, a CT application for reconstructing 3D volumes from 2D images collected by scanners based on cone-beam geometry. The main contribution of this paper are the following. First, we develop a modular application design that allows to exploit the functional parallelism inside the application and to facilitate the parallelization of individual application phases. Second, we identify a set of optimizations that can be applied individually and in combination for optimally deploying the application on a massively parallel multi-GPU system. Third, we present a study of surfing the optimization space of the modularized application and demonstrate that a significant benefit can be obtained from employing the adequate combination of application optimizations. We show that improving the performance of Mangoose++ is not a straightforward process, as optimization interference complicates the inference of the best combination of optimizations to be applied. We advocate that understanding the architecture, especially the underlying memory hierarchy and data paths through the system, is crucial, as communication needs to be reduced by exploiting data locality and appropriately distributed and balanced in order to avoid congestion. Finally, we conclude that improving the performance of Mangoose++ has to be done in a holistic way, as side-effects of optimizations are difficult to predict.

Keywords:
CT reconstruction, tomography, GPGPU, SSD, multicore, optimization, paralellism

1. Introduction

X-Ray Computed Tomography (CT) is a medical imaging technique that produces tomographic images, i.e. slices of the body, based on measuring the attenuation of X-rays as they pass through the body [1]. Many small animal X-ray computed tomography scanners are based on cone-beam geometry with a flat-panel detector orbiting in a circular trajectory. A mathematical algorithm is used to reconstruct the 3D volume from the measured data.

Image reconstruction in these systems is usually performed based on the algorithm proposed by Feldkamp, Davis, and Kress (FDK) [2] because of its straightforward implementation and computational efficiency. The evolution of the detector panels has resulted in an increase of detector elements density, which produces a higher amount of data to process [3]. Together with that, there is an increasing interest for faster reconstructions to address CT applications that require soft real-time imaging, like image assisted surgery. This motivates the need to look for optimizations that can handle the complexity and demand of the reconstruction task.

One method to speed up the reconstruction process is by designing faster and more scalable algorithms. A first group of algorithms are based on regridding the projection data in the Fourier domain into a Cartesian grid in order to be able to use inverse 2D-FFT directly. An alternative to avoid the distortion introduced in the regridding step is based on hierarchical decomposition and angular downsampling of the backprojection operation [4]. These algorithms have been reported to reach a speed up of 40, although the quality of the image is degraded and they lack generality due to their dependency on the image properties [5]. Another performance improving strategy of FDK algorithms is through parallel computing techniques and architectures. In this direction there are many approaches, some of which are rigid and costly, as the use of application-specific integrated circuit (ASIC) or FPGA devices [6]. Currently, one promising alternative is exploiting the parallelism inherent to the General-Purpose computation on Graphics Processing Units (GPGPUs).

This work explores the implementation and the performance optimization process of a multiple GPGPU-based FDK reconstruction algorithm called Mangoose++. We target to understand the impact of applying combinations of independent optimizations on the application performance and to identify the limitations of current hardware architectures that need to be addressed for overcoming performance bottlenecks.

The main contributions of this paper are the following. First, we develop a modular implementation of an FDK reconstruction algorithm [7] that allows to exploit the functional parallelism inside the application and to facilitate the parallelization of individual application phases. Our approach achieves high computational resource utilization of heterogeneous resources by employing a hierarchical approach based on CPU-level and GPU-level problem decomposition. Second, we identify a set of optimizations that can be applied individually and in combination for optimally deploying the application on a massively parallel multi-GPU system. Our highly modularized solution facilitates the independent combination of optimizations for I/O, memory transfers, and multi-GPU operations, as shown in the paper. Third, we present a study of surfing the optimization space of the modularized application and demonstrate that a significant benefit can be obtained from employing the adequate combination of application optimizations. Finally, the paper includes an analysis of the performance of our implementation compared with some current state-of-the-art solutions, showing clear enhancements over them.

The remainder of this paper is organized as follows. Section 5 reviews related work. Section 2 contains a background on GPU hardware and programming. The detailed design and implementation of our solution is presented in Section 3. The experimental results are presented and discussed in Section 4. Finally, we conclude in Section 6.

2. GPU Background

In this section we provide a background of the GPU systems in two sections. Subsection 2.1 presents the NVidia GPU architecture. Subsection 2.2 discusses the main challenges faced in order to efficiently program multi-GPU systems.

2.1. CUDA architecture

The CUDA model of a GPU consists of a number of streaming multiprocessors (SM), as shown in Figure 1. CUDA architecture has experienced four huge improvements: G80, GT200, Fermi, and Kepler. In case of the Fermi architecture, each SM has one issue unit, eight scalar processors (SP), two single precision floating point units and one optional double precision floating point unit. The lasted architecture, (called Kepler) uses a similar design, but with a couple of key differences. Each SM is now a next-generation Streaming Multiprocessor, (which Nvidia abbreviates as SMX) and each SMX contains 192 CUDA cores.

The memory of a CUDA device can be classified broadly into memory accessible by all SMs and memory accessible only from within a SM. The memory accessible by all SMs consists of device memory, texture memory and read-only constant memory. On the other hand the threads scheduled inside one SM can access locally a number of registers, a shared memory and a read-only constant cache (speeding up the accesses to the constant memory of the device).

GPUs supports massive data and instruction parallelism by allowing the interleaved execution of threads on cores. The interleaving of threads on the cores is done by a hardware scheduler. Basic instruction level parallelism can be achieved through a simple pipeline, which allows only for in-order execution of the instructions issued by one thread. There is no branch prediction and instruction reordering such as in a CPU core.

2.2. GPU programming challenges

Several well-known challenges have to be overcome for employing GPU-based systems. In the case of a single GPU, the most prominent challenges are programmability and performance efficiency. For multiple GPUs the

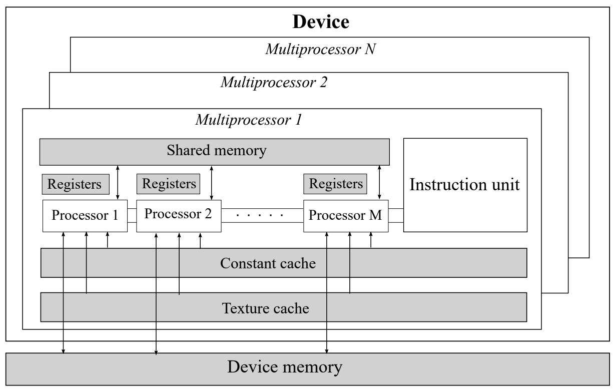


Figure 1. CUDA memory architecture. The figure shows how streaming multiprocessors access the different types of memory: device, shared, constant, and texture memory.

additional challenges over the single GPU case are load-balancing and efficient I/O management, which can be addressed by efficiently exploiting various available types of parallelism. These types include internal GPU parallelism and external parallelism among heterogeneous GPU types. Additionally, overlapping CPU-GPU transfers with GPU and CPU computation requires to focus on both structure and dynamics of the CPU and GPU computation and to efficiently use the asynchronous transfer mechanisms offered by GPUs.

An efficient utilization of a multi GPU architecture requires to efficiently schedule the CPU-GPU transfers targeting two goals: a) to reduce PCIe contention b) to increase the overlap between communication and computation. Shaa et al. showed that contention may have a cost of 15-20% of the aggregate throughput of concurrent transfers over PCIe [8]. Finally, disk I/O may cause an additional source of contention over PCIe.

In general efficiently exploiting all these types of parallelism is markedly complicated due to the difficulty on achieving load balance and high resource utilization.

3. Mangoose++

The algorithm proposed by Feldkamp, Davis, and Kress, commonly referred to as the FDK algorithm is a well-known 3D reconstruction approach based on an approximation of the 2D filtered back-projection algorithm for the case when 3D data are acquired with a cone beam geometry, e.g. flat-panel detectors and circular trajectory.

Mangoose++, is a new implementation of the FDK algorithm, targeting hybrid multi-core multi-accelerator systems. The proposed implementation takes advantage of parallel computing capabilities in order to speed up the two main stages of the algorithm implementation. The algorithm proposed by Feldkamp, Davis, and Kress, commonly referred to as the FDK algorithm is a well-known 3D reconstruction approach based on an approximation of the 2D filtered back-projection algorithm for the case when 3D data are acquired with a cone beam geometry, e.g. flat-panel detectors and circular trajectory.

To get a final volume of a CT scanner, the Mangoose++ reconstruction process has been designed as a pipeline consisting of six main phases that are executed in sequence:

1. Read I/O (R), that brings 2D projections from a file into the host main memory.
2. Scattering (S) 2D projections to parallel processing units, to transfer data from host memory to the device memory.
3. Filtering (F) the 2D projections, by iteratively process a number of 2D projections.
4. Backprojection (B), that reconstructs partial a 3D volume.
5. Aggregation (A) partial volumes into the final volume, by transferring sub-volumes from device memory to the host memory.
6. Write I/O (W), to transfer the volume in the host memory to persistent storage.

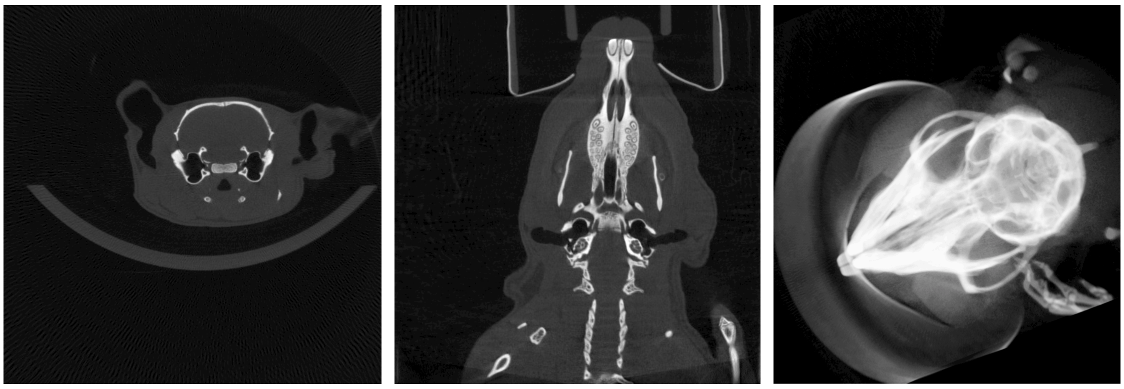

(a) (b) (c)

Figure 2. Coronal view (a), axial view (b), and volume render (c) of a mouse in a standard resolution study.

An example of a volume reconstructed by Mangoose++ is shown in Figure 2. We part form a set fo 2D projections (Figure 2 (a) and (b)) and the final result is a reconstructed 3D volume.

$$f(u,v,z) = \frac{1}{2}\int_{\theta=0}^{2\pi} W_2(v)\left[\int_{w=-\infty}^{\infty}\left(\int_{s=-\infty}^{\infty}[p_\theta(s,z)\cdot W_1(z,u)]\cdot e^{-j2\pi sw}ds\right)\cdot|w|\cdot e^{j2\pi ws}\cdot dw\right]\cdot d\theta \tag{1}$$

$$W_1 = \frac{SO}{\sqrt{SO^2+z^2+u^2}} \tag{2}$$

$$W_2 = \frac{SO}{(SO-v)^2} \tag{3}$$

The FDK formula is described in Equations (1), (2), and (3). The formula is derived by simply introducing a third (axial) coordinate in the FBP equation, in such a way that all the rays can be considered [9].

With corresponding to the pixel value in the reconstructed image at coordinates (u, v, z), to the projection data for angle θ and position (s, z) in the detector, and W_1 and W_2 to the weighting factors introduced to compensate for the different ray lengths, the formula involves a Fourier Transform and an Inverse Fourier Transform step. SO is the distance from the source to the detector, z is the axial coordinate, common for both detector and reconstructed volume reference frames, s is the radial coordinate in the detector, and u, v are the Cartesian coordinates in the reconstructed volume. The weighting factors W_1 and W_2 are introduced to compensate for the different ray lengths.

The reconstruction module can be seen as a pipeline consisting of six main phases (as shown in Figure 3). Filtering and backprojection phases are highly parallelizable due to the lack of data dependencies and high number of data parallel operations. However, the main problem of FDK algorithm is that it introduces a high computational load when coping with large volumes. Thus, it is very important to make optimizations to exploit the available parallelism. In the following subsections, we present the application design, the parallelization strategy, and the optimized implementation of Mangoose++.

3.1. Application design

In overview, our approach is based on hierarchical problem decomposition and overlapping computation, communication, and storage I/O. The main design goal is to improve the flexibility of tuning the performance and scalability of the application, while achieving a high degree of resource utilization.

Figure 4 shows the Mangoose++ implementation structure together with the data flow through different system memories. The Mangoose++ architecture consists of six phases: read I/O (R), scattering 2D projections to parallel processing units (S), filtering the 2D projections (F), backprojection to partial 3D volumes (B), aggregate partial volumes into the final volume (A), and write I/O (W). These phases can be partially overlapped (functional parallelism)

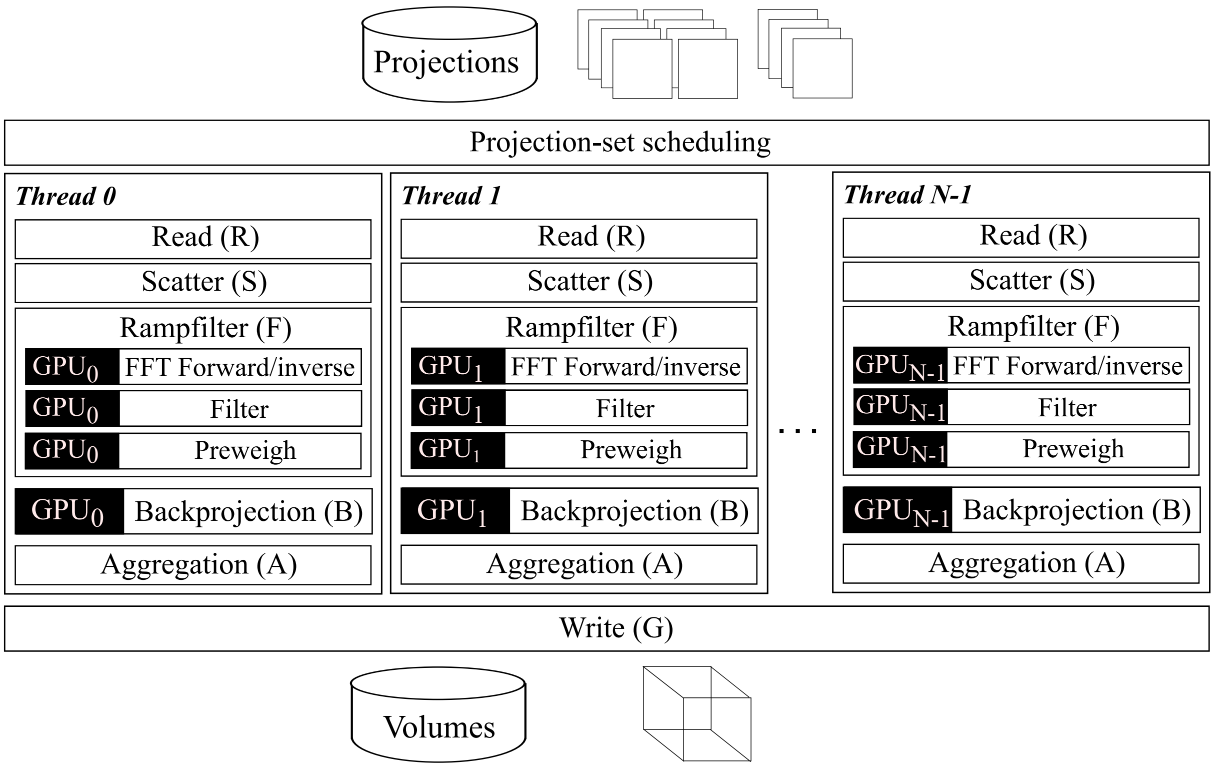


Figure 3. Design of a multi-GPU implementation of Mangoose++. ”e show the phases decomposition of Mangoose++. Each GPU is attached to a specific OpenMP thread, which is in charge of manage GPU context.

Table 1. Mangoose++ optimization table. The first column represents the scope of the optimization. The second column provides a notation for each optimization. The optimized application phases are shown in the third column. The optimization name is given in the fifth column. Finally, the last column provides the variables of each optimization.

Scope	Notation	Descripcion	Phases	Variables
I/O	O_1	Prefetching	Read (R)	read-ahead window
	O_2	Pinned vs pageable memory access	Scatter (S), Aggregation (A)	enabled/disabled
Memory	O_3	GPU shared memory	Backprojection (B)	slab size
	O_4	Write-combined	Scatter (S), Aggregation (A)	enabled/disabled
	O_5	Kernel overlaping	Filter (F), Backproyection (B)	enabled/disabled
GPU	O_6	Multi-GPU	Filter (F), Backprojection (B)	gpu_{nr}

and there is a lot of available concurrency in each phase. In Subsection 3.3 we present implementation details of these phases and we discussed the optimizations applied to each of them.

Filtering and backprojection phases are highly parallelizable due to the lack of data dependencies and high number of data parallel operations. However, the main challenge is to highly utilize the different types of available parallelism from a multiple GPU system.

3.2. *Parallelization strategy*

One of the most popular parallelization methodologies [10] consists of the following four steps: decomposition, assignment, orchestration, and mapping. While this methodology can be straightforwardly applied to a given functional module, Mangoose++ requires a more complex approach for at least four reasons. First, Mangoose++ consists

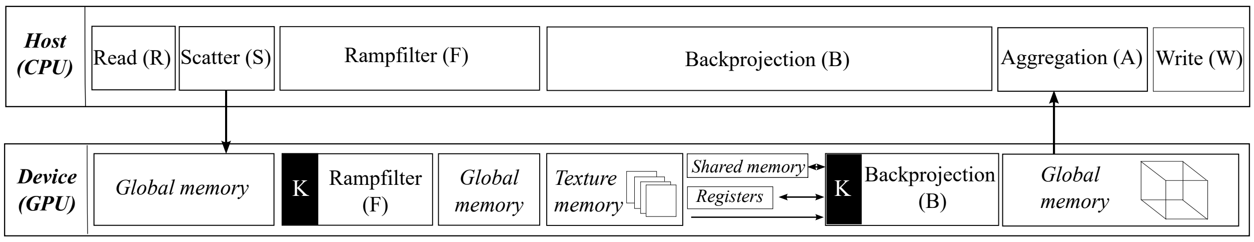


Figure 4. Design of a multi-GPU implementation of Mangoose++. We show memory host-device transfers.

of several functional modules, which may require different parallelization granularities. Second, the performance dependence between optimizations of modules shows non-linear effects, as it will be seen in the evaluation section. Third, our main goal is to map the application on a hybrid architecture with a complex memory hierarchy. Fourth, there is a number of optimizations that can be applied at the interface between functional units (e.g.: pinned memory, mapped memory).

In the remainder of this subsection we will discuss the parallelization strategy of Mangoose++ through the prism of the methodology discussed above, while highlighting the main differences.

3.2.1. Decomposition

In standard parallelization methodology a task is a unit of sequential processing. In our approach the decomposition of the problem into tasks is done at two levels. At the first level the GPU is considered as one unit, and we decompose the sequential problem into supertasks, having a minimum granularity of one projection (a group of projections in the general case). At the second level a supertask is decomposed into finer granular tasks, which are to be processed independently on the GPU cores. In CUDA terminology a supertask is a kernel and a task is a GPU thread.

3.2.2. Assignment

The first-level assignment is done through dynamic scheduling. Each idle GPU is dynamically assigned a supertask (a group of projections). The tasks of each supertask are further assigned to GPU threads, which are running on the GPU cores. We consider two criteria for assigning tasks to GPU processing elements targeting to: (i) fully occupy the processing elements in order to archive full thread parallelism (high occupancy), (ii) hide the memory latency by assigning more tasks than processing elements (warps).

3.2.3. Orchestration

The orchestration phase addresses the interactions between threads and targets to reduce the cost of synchronization and communication. The sequential program shows a low number of data dependencies, which require a small amount of synchronization (e.g. synchronization is needed when computing the total volume). However, there are two important non-trivial parallel strategies that need to be orchestrated for achieving a good resource utilization by the parallel algorithm: I/O strategy and memory layout strategy. The I/O strategy requires synchronization between accessing data from persistent storage, transferring them to GPGPUs and processing. The memory layout strategy has the potential of reducing the memory access time by exploiting the inter-task locality.

3.2.4. Mapping

The mapping phase is straightforward. A supertask (kernel) runs in an OpenMP thread, which is mapped on a CPU core. A task runs in a GPU thread and is mapped on a GPU core.

3.3. Implementation and optimizations

The implementation leverages both application-level pipeline parallelism (i.e. the functional parallelism between phases of the application) and the hardware-level parallelism (i.e. CPU multi-core parallelism, multiple GPU parallelism, inner GPU multi-core parallelism). Table 1 provides an overview of a subset of the optimizations provided by our implementation. For brevity, these optimizations are numbered from O1 to O6.

In the *reading phase* (R), the algorithm calculates the subset of the projection data required to reconstruct the volume of interest (VOI), according to geometrical considerations. The required part of each projection is read from the storage device into the main memory. The granularity of the read-ahead projections may vary between one 2D projection and the total number of projections. The R phase is implemented either based on POSIX synchronous operation (`read`) or asynchronous I/O operations (`aio_read`). For the asynchronous read I/O optimization (O1), the read I/O overlaps in time with the other phases. The maximum amount of data to be read ahead is limited by a maximum prefetch window size, which is a configurable parameter. Two of the questions of the optimization process is whether to use synchronous or asynchronous I/O and how to choose optimal values of the read-ahead parameter.

The *scattering phase* (S) transfers data from main memory to the memory devices. The transfer can be overlapped with kernel execution by pinning host memory pages, and, therefore, avoiding swapping. This is an approach recommended by NVidia to improve the transfer performance between host and device [11]. Our implementation provides as an option the possibility of using pinned memory in S phase (O2).

The *filtering phase* (F) iteratively processes a number of 2D projections. The computation is decomposed into groups of projections, which are processed in a loop, whose iterations are scheduled by OpenMP. Each iteration consists of four pipelined steps, which are implemented as CUDA kernels: 1) direct FFT based on CUFFT (the Fast Fourier Transform library provided by NVIDIA) 2) multiplication 3) inverse FFT based on CUFFT 4) weighting by a first geometrical factor.

The result produced by F phase in used as input to the *backprojection phase* (B). The B phase consists of spreading back the filtered projection values along each ray. It is implemented for each projection angle slice by slice (z index), following a voxel-driven approach. For each voxel in the slice, the position index pair (x,y) is rotated by means of a rotation matrix corresponding to the projection angle. The algorithm then calculates the cone-beam projection onto the detector and computes the corresponding backprojection value by bi-linear interpolation of the four neighboring projection pixels. Finally, the voxel value is incremented with the result of this interpolation and weighted by a second geometrical factor. The B phase input is passed through the GPU memory, i.e. it does not involve any transfer between host and devices. In the same fashion as the F phase, the B phase is decomposed into groups of projections, which are processed in a loop, whose iterations are scheduled statically or dynamically by OpenMP. Each iteration consists of two pipelined steps: the interpolation and the proper backprojection. The interpolation is based on a CUDA kernel, which leverages the GPU texturing hardware for improving the performance. The 2D projections are interpolated fast by fetching them through CUDA arrays bound to texture units. The proper backprojection is a CUDA kernel, which aggregates the interpolated 2D projection into a partial volume calculation. Optionally, our implementation exploits the temporal locality of the access pattern by calculating the partial volumes in tiles of configurable sizes stored in shared memory (O3). Finally, given that CUDA offers asynchronous memory transfers and concurrent CUDA kernels, the scattering, filtering and backprojection phases can be executed concurrently (O5).

The *aggregating phase* (A) transfers partial volume calculations from the device memory to the host memory and adds them into the final volume. As in the case of S phase, the transfer can be overlapped with kernel execution by pinning the host memory. Write-combining page-locked memory (O4) frees up the CPU's L1 and L2 cache resources, making more cache available to the rest of the application [11].

Finally, the *write phase* (W) stores the resulting volume to a file.

4. Experimental results

Data were acquired with the CT subsystem of an ARGUS PET/CT scanner [12], based on cone-beam geometry. In order to provide standard timings, we have used two acquisition protocols with 360 projections of 512x512 pixels (standard resolution) and 2048x2048 pixels (high resolution), respectively. The high resolution dataset was used to evaluate the effect of handling big data volumes (2880 MBytes of input data) on the performance. For both cases the reconstructed volume had a size of 512x512x512 pixels (given the memory limitation of the employed GPUs), resulting in a file of 268 MBytes.

The computer system used in the evaluation is equipped with two Intel Xeon E5640 (2.67 GHz quad-core processors) and 64 GBytes of RAM. The motherboard supports up to four PCIe 2.0 x16 buses. The source code was compiled with Intel 12 compiler with all the optimization flags activated and with CUDA version 4.2. The evaluation was done using five GPU models, whose attributes are provided in Table 2. The Linux kernel used in all evaluations

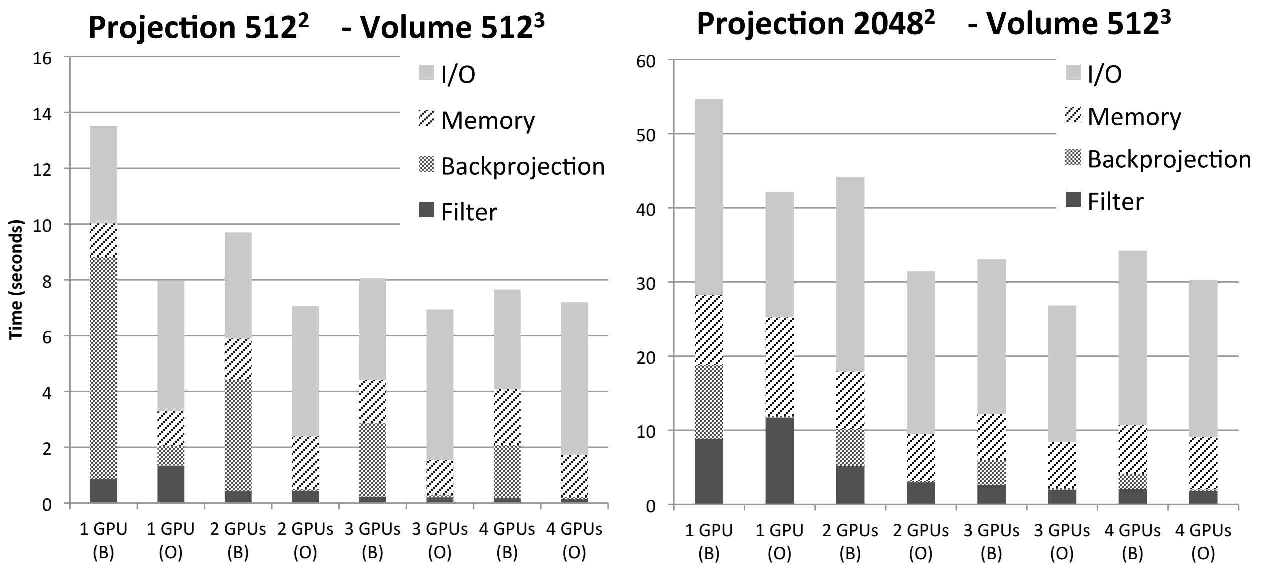


Figure 5. Breakdown of application time for reconstructing 3D volumes with 512^3 resolution from 2D projections of sizes 512^2 (left) and 2048^2 (right). We provide the results for the baseline (B) and full optimized (O) versions. We plot results for 1, 2, 3, and 4 GPUs for filter (dark gray), backprojection, memory, and I/O stages.

Table 2. Comparison of the evaluated GPU models.

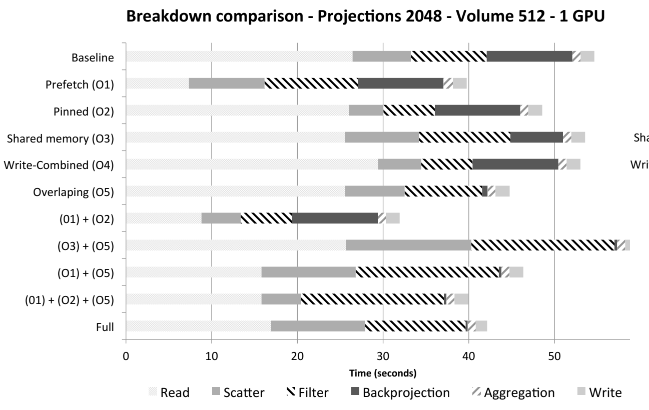
Specification[1]	**GT 610**	**Tesla C2050**	**GTX 470**	**GTX 680**	**GTX 760**
Nvidia family	Kepler	Fermi	Fermi	Kepler	Kepler
GPU Clock Speed (GHz)	0.81	1.15	1.22	1.41	1.32
CUDA cores	48	448	448	1536	1152
Performance (TFlops)	0.15	1.8	1.0	3.0	3.0
Global memory (MBytes)	1024	2687	1280	2048	2048
Bandwidth device to device (GBytes/sec)	14.4	144	133.9	192.2	192.2
bandwidth host to device (GBytes/sec)	3.0	5.8	5.8	5.8	6.0
Memory Clock (MHz)	600	1500	1674	1848	1848
L2 Cache Size (Kbytes)	64	64	64	512	512

was 2.6.35. The local file system on both the hard disk and SSD was ext4. The file system caches were flushed before running the experiments. For storage we employed a Seagate Constellation ES ST31000524NS hard disk device (HDD) and a Kingston SV300S37A120G solid state device (SSD).

4.1. Mangoose++ timing breakdown

In this subsection we present the breakdown of Mangoose++ time for reconstructing 3D volumes with 512^3 resolution from 2D projections of sizes 512^2 (left) and 2048^2 (right). Figure 5 shows the results of running Mangoose++ with baseline and full optimized versions for 1, 2, 3, and 4 GPUs. This experiment was carried out using two GTX 470 and two Tesla C2050 (configured in this order), given the architecture and performance similarities. We note that the backprojection and filtering phases scale almost linearly with the number of GPUs. The memory transfer does not scale for the small image (512^2) and scales for the larger image as the 2048^2 overhead of starting the transfer is amortized. Finally, the I/O phase does not scale with the number of GPUs, as the disk is a sequential bottleneck.

[1]Source: NVidia web site and the *deviceQuerry* tool included in the CUDA toolkit.



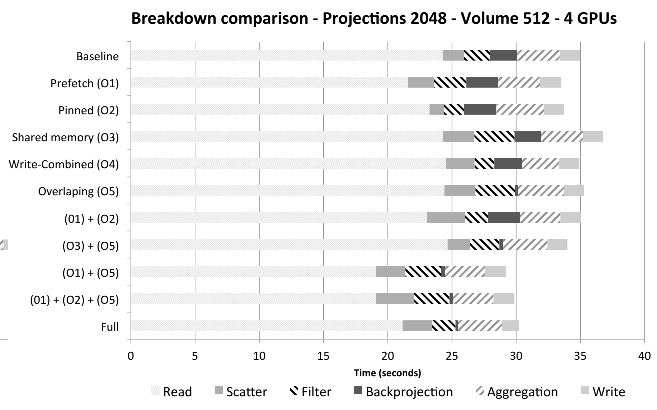


Figure 6. Breakdown of application time for reconstructing 3D volumes with 512^3 resolution from 2D projections of size 2048^2. We provide results for 1 GPU (left panel) and 4 GPUs (right panel). For each case we provide the results for the baseline (no optimization), applying one of the five optimizations O1-O5, promising combinations of two optimizations (O1+O2 and O3+O5), a promising combination of three optimizations (O1+O2+O5), and all optimizations (full).

We observe that for large projections the total time of the application is dominated by I/O. Reducing this time by using a parallel file system or different type of persistent memory such as Solid State Devices (SSDs) can substantially increase the performance and help achieve almost linear scalability, as we show in Subsection 4.3.

4.2. Surfing the optimization space

This subsection presents the experiments conducted for better understanding the effect of individual optimizations and combinations of them on the Mangoose++ performance. Due to space limitations we chose from the two cases evaluated in the previous subsection the one involving the larger amounts of data (2D projections of size 2048^2). The results are shown in Figure 6 in the left panel for 1 GPU and in the right panel for 4 GPUs with the same GPU configuration of the previous subsection. In each panel we plot the execution results for: one execution of Mangoose++ without optimizations (baseline), several executions of Mangoose++ with exactly one optimization enabled, two executions of Mangoose++ with two optimizations selected based on the promising results of one optimization, one execution of Mangoose++ with a combination of three optimizations, and, finally, for facilitating the comparison, the execution with all optimizations enabled discussed in the previous section. In all the experiments carried out, the computer was equipped with a magnetic hard disk, previously described in Subsection 4.

Applying exactly one optimization for both 1 GPU and 4 GPUs, I/O prefetching (O1) mostly improves the total time by overlapping file I/O with other operations such as computation or memory transfers. The next best performing optimizations are O5 (overlapping F and B phases) and O2 (pinned memory) for both cases. Further, O3 (shared memory) significantly reduces the B phase execution time.

In order to evaluate the interactions of optimizations, we picked up the most promising optimizations and applied them in pairs. For space reasons Figure 6 shows only O1+O2, O3+O5, and O1+O5 combinations, which were the ones that produced best results. Against intuition, for both 1 and 4 GPUs, we identified combinations of two optimizations that performed better than the case of applying all optimizations: O1+O2 for 1 GPU and O1+O5 for 4 GPUs. For 1 GPU the combination of prefetching from disk to CPU RAM memory (O1) and pinned memory for transfers from CPU RAM memory to GPU memory (O2) achieves the best overlap between disk I/O, memory transfers, and computation. However, for the same combination of optimizations (O1+O2) in the case of 4 GPUs the PCI contention for transfering data from CPU RAM to the 4 GPUs significantly reduces the efficiency of prefetching data from disk and the pinned memory does not add any benefit. On the other hand, for 4 GPUs the best results are obtained when disk I/O prefetching (O1) and overlapping scattering, filtering, and backprojection (O5). On the other hand the same combination of optimizations does not bring any benefit to 1 GPU case, as it appears that the prefetching does not provide enough data so that the subsequent phases are executed efficiently. We observe a performance decrease in case of optimization (3) and (5) in the 4 GPUs case comparing with the baseline implementation. For both cases, the lack of memory transfer optimizations affects the scatter phase, due to these optimizations change the communication

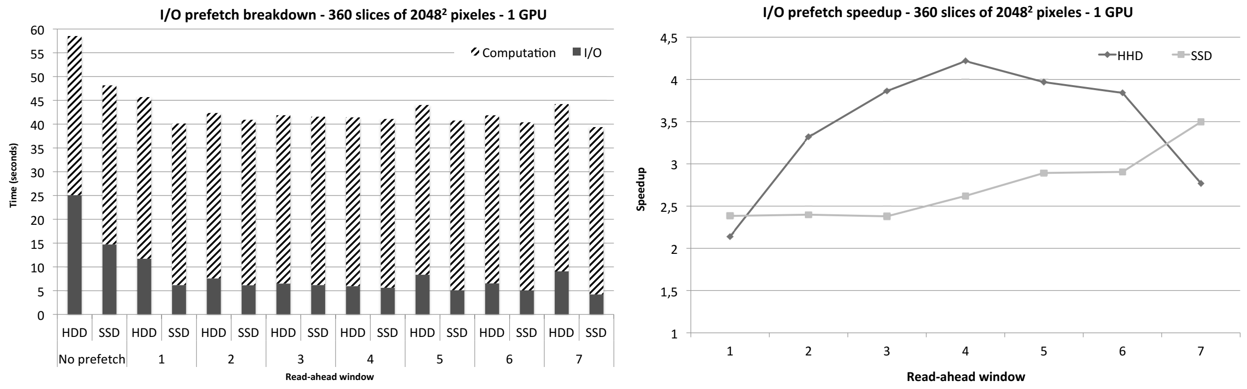


Figure 7. On the left panel, breakdown of application time into read I/O and other operations (computation, memory access, etc.) for reconstructing 3D volumes with 512^3 resolution from 2D projections of size 2048^2. The read ahead window has been varied from 0 (no prefetch) to 7 blocks. The best result of overlapping I/O with other operations is obtained for 4 blocks for current magnetic disks. On the right panel, speedup of HDD and SSD for an increasing read-ahead window.

pattern between host and device. Finally, we combined the best three optimizations (O1, O2, and O5) and obtained timings worse than when applying two optimizations and slightly better than the total times when all optimizations are applied for both 1 and 4 GPUs.

4.3. Asynchronous I/O

The asynchronous I/O read optimization (O1) was shown in the previous subsection to have the greatest impact on the total performance of the application. Here we discuss the experiments we have performed for obtaining the best trade-off between file prefetching and overlapping I/O with computation and memory access. Intuitively, prefetching too few data reduces the overlap of I/O and other operations, while prefetching too much can interfere with other operations such as transfers between the main memory and GPU memory. Figure 7 shows the results for applying only the prefetching optimization (O1), while varying the read-ahead window from 0 blocks (no prefetching) to 7 blocks and the storage device. The left panel of the figure shows the breakdown into compute time and storage I/O time, while the right panel plots the speedup over the case with no prefetching. A lower storage I/O time and a speedup larger than 1 indicates a better overlap between computation, memory transfers, and storage I/O.

In the base case (no prefetch), as expected, the HDD is around 66% slower than the SSD. Note that prefetching hides a significant amount of the storage I/O overhead for both the HDD and the SSD. When using magnetic disks, prefetching 4 blocks achieves the best trade-off between file I/O read overlapping and interference with other operations. In the case of using SSD, as we increase the read-ahead window, we reduce the I/O time for all the read-ahead windows. In the best case, the combination of SSD and prefechting can reduce the I/O time up to 60% and the total execution time by 48%. The speedup for magnetic disk is higher than for SSD due to the fact that the I/O throughput of the magnetic disk is significantly worse. However, the best improvement in absolute terms is offered by the solution using SSD and the maximum evaluated value of the prefetch window.

In general for each hardware configuration it is advisable to determine the optimal trade-off by running the experiment described in this section before combining the prefetching optimization with other optimizations.

4.4. CPU and GPUs performance comparison

In this subsection, we study and compare the performance archived by Fermi and Kepler architectures. Figure 8 plots a comparison of performance between a CPU-based implementation and GPUs over different NVidia's families, including Fermi and Kepler. A detailed specification of each evaluated card is summarized in Table 2. We conduct an evaluation using five different GPUs to quantify architectural impact of GPU core and memory bandwidth.

In order to study the performance of GPUs, we compare the execution time with a multi-core CPU-based version, in where filtering and backprojection phases have been parallelized using OpenMP (fine grain) and all the computation loops have been vectorized. Comparing to the performance between GPU and CPU, current GPU models obviously

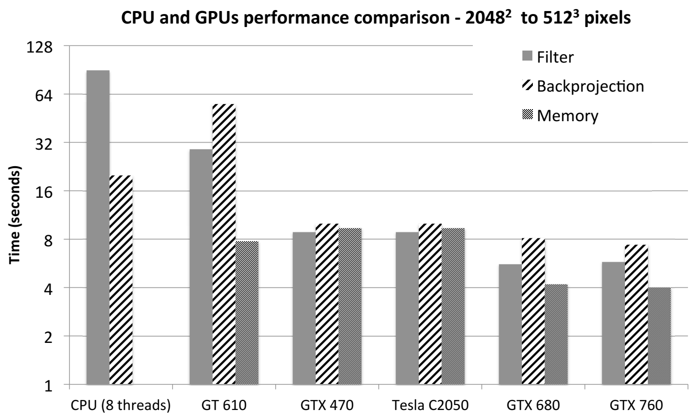


Figure 8. Breakdown of application time for reconstructing 3D volumes with 512^3 resolution from 2D projections of sizes 2048^2. We show results for a vectorized multi-core based and four NVidia GPU models. In order, the figure plots the filter, backprojection, and memory transfers time.

exceeds current CPU. However we note that in case of desktop-base cards, the multi-core base implementation outperforms a GPU in the backprojection phase. We observe that the current Kepler cards reduce the memory transfer time by two. In all the graphics cards, independently of the amount of processing cores, memory transfers are as expensive as computation. However, we note that as GPUs increase the amount of cores, the speedup obtained does not scale linearly. GTX 680 reduces execution time by 58% instead of the theorical 242%, comparing the number of cores of each card. As we describe in Table 2, current Kepler cards increase by 43% the effective memory bandwidth. Therefore, improving the internal memory bandwith is a key challenge for current and future GPGPU-based memory-bounded scientific applications.

4.5. Discussion

These non-intuitive results demonstrate the difficulty of choosing combinations of optimizations based on the results of independently applying optimizations. However, our results clearly demonstrate that the prefetching-based asynchronous I/O has the highest impact on performance and is present in all successful combinations of optimizations. On the other hand we note that the main key of achieving high-performance is to find the right optimization combination that achieves a high utilization of the system data paths, while avoiding contention. Striking this balance of data flow is non-intuitive and requires empirical evidence about the bottlenecks in the system. However, once this evidence is available, it is also possible to draw some conclusions about the proper architectures on which the application would perform best. From our results it is clear that storage I/O represented an important overhead for these configurations. Using parallel file systems to speed-up the storage I/O would result in further considerable performance benefits. Further, the PCI Express (PCIe) bus represents an important limitation when using 4 GPUs as all transfers to GPU's memory and disk I/O share this data path. Two possible solutions to this bottleneck could be a more scalable bus architecture or distributing the GPUs over several nodes. In the first case a hardware architect has to design the proper solution and the software engineer has to balance the data flow of application phases to achieve high utilization. The second case requires finding a trade off between two possible solutions: a) distribute the volume into sub-volumes over several nodes, replicate the projections, compute the sub-volumes, and join the sub-volumes into the final volume b) distribute the projections over several nodes to calculate partial sums of volumes and use a reduction operation to calculate the final volume from the partial sums of volumes. These solutions overcome the existing bus overhead at the cost of inter-node communication. We are aware of no study that approached this trade-off and we are currently working in this direction.

Finally, most optimization studies have focused on optimizing the filtering and the backprojection phases. However, this study has shown that these stages may require only a small part of the total execution time. Additionally, the optimizations applied to these phases can interact with other optimizations from other phases, and an application developer must carefully study these interactions in order to understand the side effects of each optimization. In general each optimization needs to be studied in the context of the data flow through the system data paths in order to properly balance the traffic and avoid congestion.

Table 3. Backprojection performance in the literature and in this paper. The column S compares implementations in terms of normalized speed up.

Authors	Year	Hardware	Projections	VOI	BP (s)	S(%)
Zhu et al. [13]	2012	3xC2050	1024^2x720	1024^3	13.5	1
Zhang et al. [14]	2012	2xGTS480	1024^2x360	512^3	7.27	0.10
Scherl et al. [15]	2012	GTX280	1240^2x543	512^3	6.70	0.16
Papenhausen et al. [16]	2011	GTX480	512^2x360	512^3	6.07	0.95
Papenhausen et al. [17]	2013	GTX480	512^2x360	512^3	6.07	0.95
Zinsser et al. [18]	2013	GTX680	512^2x468	512^3	0.99	0.22
This work	2013	2xGTX470 + 2xC2050	2048^2x360	512^3	0.57	1.16
This work	2013	2xGTX680	1024^2x720	1024^3	6.58	3.02

5. Related work

There are several works that focus on optimization of image processing algorithms based on GPGPUs (e.g. [19], [20], [15]). As far as we know, Zhu et al. [13] and Okitsu et al. [21] are among the few authors to take into account the overall reconstruction time, including disk I/O and memory transfers between GPU and CPU memory . Regarding disk I/O, Zhu et al. study only the benefits of hiding the latency of persistently storing the 3D volume through asynchronous writes. However, as we show in Section 4, reading projections from disk is as important as storing the reconstructed volume, especially for high resolution studies (e.g. 2048^2 projections).

Zhang et al. [14] present a parallel execution flow for multiple GPUs . The workflow aims to increase the utilization factor of GPUs by running multiple kernels concurrently. Our solution differs from this work by overlapping GPU computation and memory transfers between host and GPU. Papenhausen et al. [16] speed-up the GPU memory transfers by optimizing CPU-GPU memory copies through page-locked allocated memory, also known as pinned memory. Mukherjeet et al. [22] and Zinsser et al. [18] pointed out that transferring between CPU and GPU can cause a significant overhead and overlapping communication and computation could become critical for performance. Papenhausen et al. [17] proposed a fine-tunnig tool that aims to optimize GPU source code by using the ant colony optimization algorithm. The modular design of Mangoose++ allows to cover all the application optimization space. As we show in the evaluation section, we study in detail the efects of I/O, memory, and computation optimizations.

Although the previous works advocate for increasing the degree of parallelization of GPGPUs, we show that there is an increasing need to improve data transfer in the memory hierarchy (storage, host memory, and GPU device memory).

Given that the backprojection phase is one of most studied part of the reconstruction process, in Table 3 we contrast our approach with other implementations from the literature. We propose as a comparative metric the normalized speedup as showed in Equation (4), where nr_proj is the number of projections processed, VOI_size the size of the volume of interest in pixels, nr_gpus the number of GPUs, and t_bproj the time needed to backproject the whole volume.

$$S = \frac{nr\_proj_1}{nr\_proj_2} * \frac{VOI\_size_1}{VOI\_size_2} * \frac{nr\_gpus_2}{nr\_gpus_1} * \frac{t\_bproj_2}{t\_bproj_1} \tag{4}$$

Due to the differences in the employed hardware platforms and image sizes, a direct comparison of the reconstruction time is not possible. However, these values give an informative view of performance of the reconstruction algorithms. Our implementation of the backprojection phase is 16% faster than the best solution proposed in the literature. It was not possible to compare other phases as most papers do not provide breakdowns of the total time of the reconstruction process.

6. Conclusions

In this work, we have presented a novel modular implementation of a FDK-based algorithm for multiple GPGPU systems. The phases of the applications are organized into functional modules, which can be individually parallelized. This approach facilitates the development of new optimizations and the evaluation of the impact of combination of optimizations on the application performance. Further, the paper describes the a selection of optimizations developed for each of the application phases. Subsequently, we study the influence of several optimizations on the performance considering them isolated and in interaction with each other. Our experiments demonstrate that for current problem sizes, the storage I/O and memory transfers are the major limiting factors in heterogeneous systems. This is an important observation, as many studies presented in the related work section concentrate only on the optimization of the computation, while ignoring the storage I/O and memory-to-memory transfer performance.

We show that optimization process is not straightforward, as most optimizations are not additive, i.e. there are non-linear effects that complicate the inference of the best combination of optimizations to be applied [23]. Understanding the architecture, especially the underlying memory hierarchy and data paths through the system, is crucial, as communication needs to be reduced by exploiting data locality and appropriately distributed and balanced in order to avoid congestion.

Our future work will focus on an iterative reconstruction approach which can be easily achieved by extending the work presented in this paper. Further, we plan to develop novel optimization techniques for improving the I/O performance through parallel I/O techniques.

Acknowledgment

This work was partially funded by the Spanish Ministry of Science and Technology under the grant TIN2010-16497, the AMIT project (CEN-20101014) from the CDTI-CENIT program, RECAVA-RETIC Network (RD07/0014/2009), projects TEC2010-21619-C04-01, TEC2011-28972-C02-01, and PI11/00616 from the Spanish Ministerio de Ciencia e Innovacion, ARTEMIS program (S2009/DPI-1802), from the Comunidad de Madrid.

References

[1] W. Kalender, Computed Tomography: Fundamentals, System Technology, Image Quality, Applications, Wiley, 2011.

[2] L. A. Feldkamp, L. C. Davis, J. W. Kress, Practical cone-beam algorithm, J. Opt. Soc. Am. A 1 (6) (1984) 612–619.

[3] X. Zhao, J.-J. Hu, P. Zhang, GPU-based 3D cone-beam CT image reconstruction for large data volume, Journal of Biomedical Imaging 2009 (2009) 8:1–8:8.

[4] S. Basu, Y. Bresler, O(n/sup 3/ log n) backprojection algorithm for the 3-d radon transform, Medical Imaging, IEEE Transactions on 21 (2) (2002) 76–88. doi:10.1109/42.993127.

[5] T. Pipatsrisawat, A. Gacic, F. Franchetti, M. Puschel, J. Moura, Performance analysis of the filtered backprojection image reconstruction algorithms, in: Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on, Vol. 5, 2005, pp. v/153–v/156 Vol. 5.

[6] K. Jin, S. Song, FPGA-based forward and back-projection operators for tomographic reconstruction , Medical Imaging 2013: Physics of Medical Imagingdoi:10.1117/12.2007533.
URL + `http://dx.doi.org/10.1117/12.2007533`

[7] M. Abella, J. Vaquero, A. Sisniega, J. Pascau, A. Udias, V. Garcia, I. Vidal, M. Desco, Software Architecture for Multi-Bed FDK-based Reconstruction in X-ray CT Scanners, Computer methods and programs in biomedicine, in press.

[8] D. Schaa, D. Kaeli, Exploring the multiple-gpu design space, in: IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12. doi:http://dx.doi.org/10.1109/IPDPS.2009.5161068.

[9] A. C. Kak, M. Slaney, Principles of Computerized Tomographic Imaging, IEEE Press, 1998, available online at http://www.slaney.org/pct/pct-toc.html.

[10] D. Culler, J. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, 1st Edition, Morgan Kaufmann, 1998, the Morgan Kaufmann Series in Computer Architecture and Design.
URL `http://www.amazon.com/Parallel-Computer-Architecture-Hardware-Software/dp/1558603433`

[11] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA Corporation, 2007.

[12] J. Vaquero, S. Redondo, E. Lage, M. Abella, A. Sisniega, G. Tapias, M. Montenegro, M. Desco, Assessment of a New High-Performance Small-Animal X-Ray Tomograph, IEEE Transactions on Nuclear Science 55 (3) (2008) 898 –905.

[13] Y. Zhu, Y. Zhao, X. Zhao, A multi-thread scheduling method for 3d ct image reconstruction using multi-gpu, Journal of X-Ray Science and Technology 20 (2) (2012) 187–197. doi:10.3233/XST-2012-0328.
URL `http://dx.doi.org/10.3233/XST-2012-0328`

[14] H. Zhang, B. Yan, L. Lu, L. Li, Y. Liu, High performance parallel backprojection on multi-gpu, in: 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2012, 2012, pp. 2693–2696. doi:10.1109/FSKD.2012.6234177.

[15] H. Scherl, M. Kowarschik, H. G. Hofmann, B. Keck, J. Hornegger, Evaluation of state-of-the-art hardware architectures for fast cone-beam ct reconstruction, Parallel Computing 38 (3) (2012) 111 – 124. doi:10.1016/j.parco.2011.10.004.
URL `http://www.sciencedirect.com/science/article/pii/S0167819111001359`

[16] E. Papenhausen, Z. Zheng, K. Mueller, GPU-accelerated back-projection revisited: Squeezing performance by careful tuning, in: Workshop on High Performance Image Reconstruction (HPIR), 2011, pp. 19–22.

[17] E. Papenhausen, Z. Zheng, K. Mueller, Creating optimal code for gpu-accelerated ct reconstruction using ant colony optimization, Medical Physics 40 (3) (2013) 031110. doi:10.1118/1.4773045.
URL `http://link.aip.org/link/?MPH/40/031110/1`

[18] T. Zinsser, B. Keck, Systematic performance optimization of cone-beam back-projection on the kepler architecture, in: F. committee (Ed.), Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, 2013, p. 225228.
URL `http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2013/Zinsser13-SPO.pdf`

[19] P. B. Noel, A. M. Walczak, J. Xu, J. J. Corso, K. R. Hoffmann, S. Schafer, GPU-based cone beam computed tomography, Computer Methods and Programs in Biomedicine 98 (3) (2010) 271 – 277, hP-MICCAI 2008.

[20] J. I. Agulleiro, J. J. Fernandez, Fast tomographic reconstruction on multicore computers, Bioinformatics 27 (4) (2011) 582–583. doi:10.1093/bioinformatics/btq692.
URL `http://dx.doi.org/10.1093/bioinformatics/btq692`

[21] F. Ino, Y. Okitsu, T. Kishi, S. Ohnishi, K. Hagihara, in: Biomedical Imaging: From Nano to Macro, 2010 IEEE International Symposium on, title=Out-of-core cone beam reconstruction using multiple GPUs, 2010, pp. 792–795.

[22] S. Mukherjeet, N. Moore, J. Brock, M. Leeser, Cuda and opencl implementations of 3d ct reconstruction for biomedical imaging, in: IEEE Conference on High Performance Extreme Computing (HPEC), 2012, 2012, pp. 1–6. doi:10.1109/HPEC.2012.6408674.

[23] J. Sim, A. Dasgupta, H. Kim, R. Vuduc, A performance analysis framework for identifying potential benefits in gpgpu applications, SIGPLAN Not. 47 (8) (2012) 11–22.
URL `http://doi.acm.org/10.1145/2370036.2145819`

Javier Garcia Blas has been a teaching assistant of the University Carlos III of Madrid since 2005. He has cooperated in several projects with researchers from various high performance research institutions including HLRS (funded by HPC-Europe program) DKRZ, and Argonne National Laboratory. He is currently involved in various projects on topics including parallel I/O and parallel architectures. He received the MS degree in Computer Science in 2007 at the University Carlos III of Madrid. He also received a PhD in Computer Science from University Carlos III in 2010.

Monica Abella obtained a PhD in Telecommunication Engineering by the Universidad Politecnica of Madrid. Her main interests are artifact correction and reconstruction algorithms for high-resolution tomography systems CT, PET, and SPECT. She has worked in prestigious research institutions in USA and Europe collaborating with key professional in the tomography image reconstruction field. She is assistant professor for the new Biomedical Engineering degree at Carlos III University in Madrid since 2010, where she has been involved in the degree start-up phases.

Florin Isaila has been an Associate Professor of the University Carlos III of Madrid since 2005. He is currently a visiting scholar at Argonne National Laboratory in the MCS Division (2013-2015) as a Marie Curie fellow. He has been a visiting scholar at Argonne National Laboratory (2007-2008) and at Northwestern University in Center for Ultra-scale Computing and Information Security (2006). He received a PhD in Computer Science from University of Karlsruhe (Germany) in 2004 and a MS from Rutgers The State University of New Jersey in 2000. His primary research interests include high-performance computing, cloud computing, distributed systems, and data mining.

Jesus Carretero is full professor of Computer Architecture and Technology at the Universidad Carlos III de Madrid (Spain), where he is responsible for that knowledge area since 2000. He is also Director of the Master in Administration and Management of Computer Systems, that he founded in 2004. He serves as a Technology Advisor in several companies. His major research is in parallel and distributed systems, real time systems and computing systems architecture. He is Senior Member of IEEE. He has participated in many conference organization committees, and in the last three years he has been General Chair of HPCC 2011 and MUE 2012, and Program Chair of EuroMPI 2013 and ISPA 2012.

Manuel Desco is doctor, senior engineer Telecommunication specialist in Nuclear Medicine, and a PhD from the Complutense University of Madrid. He is currently Head of the research department of the Hospital General Universitario Gregorio Maranon and Professor, Department of Bioengineering and Aerospace Engineering at the University Carlos III of Madrid. He currently leads the Network of Technological Innovation in Hospitals, Institute of Health Carlos III. He has worked in various sectors, through the care medicine, and finally as an entrepreneur in biomedical research. His interest has focused on medical imaging, both advanced techniques for obtaining image (MRI, positron emission tomography, etc).