uc3m | Universidad Carlos III de Madrid

Doctoral Thesis

University Carlos III of Madrid
School of Engineering

Novel High Performance Techniques for High Definition Computer Aided Tomography

Author:
Estefanía Serrano López

Tutor:
Prof. PhD. Jesús Carretero Pérez

Advisors:
PhD. Francisco Javier García Blas
Prof. PhD. Jesús Carretero Pérez

PhD. in Computer Science and Technology
Leganés, September 2018

uc3m | Universidad Carlos III de Madrid

Tesis Doctoral

Universidad Carlos III de Madrid
Escuela Politécnica Superior

Novel High Performance Techniques for High Definition Computer Aided Tomography

Autor:
Estefanía Serrano López

Tutor:
Prof. Dr. Jesús Carretero Pérez

Directores:
Dr. Francisco Javier García Blas
Prof. Dr. Jesús Carretero Pérez

Doctorado en Ciencia y Tecnología Informática
Leganés, Septiembre 2018

Estefanía Serrano López
Email: esserran@inf.uc3m.es
©2018 Estefanía Serrano López

TESIS DOCTORAL

Novel High Performance Techniques for High Definition Computer Aided Tomography

AUTORA:	**Estefanía Serrano López**
TUTOR:	**Prof. Dr. Jesús Carretero Pérez**
DIRECTORES:	**Prof. Dr. Jesús Carretero Pérez**
	Dr. Francisco Javier García Blas

TRIBUNAL CALIFICADOR

PRESIDENTE:

VOCAL:

SECRETARIO:

CALIFICACION:

Leganés, a de de 2018

“All sorts of things can happen when you’re open to new ideas and playing around with things.”

Stephanie Kwolek

Agradecimientos

Han pasado ya tres años del comienzo de esta etapa que se cierra con la defensa de esta tesis. Aún asi, este viaje comienza unos cuantos años antes, años en los que he crecido y aprendido.

Este trabajo no hubiera sido posible sin el apoyo de mis directores Jesús y Javi, los cuales me han guiado y ayudado. Sin su confianza no hubiera conseguido ni empezar, ni terminar esta etapa. También quiero agradecer al grupo de "bio", gracias a los cuales he conseguido aprender todo lo que sé de imagen médica y que ha dado pie al caso de uso principal de esta tesis. Quiero destacar mi agradecimiento a Mónica y Claudia, por sus explicaciones y por sus respuestas a mis dudas, las cuales me han ayudado a entender mejor de que va esto del CT. Gracias también a Inés, a Álvaro y Nerea, por sus pruebas y sugerencias, siempre útiles.

Gracias a Leonel, Aleksandar, Roel y Tom por acogerme en Lisboa y Leuven y darme la oportunidad de trabajar con vosotros.

Gracias a la gente del laboratorio, por esos momentos inolvidables. Gracias a Carlos, Javi Prieto, Alfredo, Fran, Alberto, Garci y Saul, por aguantar mis tardes de delirios y mis quejas interminables, y por esos momentos de desahogo en el irlandés.

Gracias al grupo ARCOS, por acogerme desde el principio y por ayudarme con mis dudas y problemas.

Gracias a Silvina, sin tu ayuda y amistad esta tesis no hubiera sido posible. Gracias por estar ahí y por aguantar mis quejas y momentos maníacos en el despacho. Una tesis no puede con nosotras.

Gracias a mi eterno compañero de prácticas, Luis. Han sido unos años difíciles, pero por fin ha terminado. No más impresiones de pósters y trenes de fin de semana.

Gracias a mis amigas de toda la vida. A Bea, Andreas, Irene, Tania y Dani. Son muchos años de amistad y ni una tesis nos ha separado.

Y gracias a mi familia. A mis padres, por haberme apoyado en mi decisión de hacer la tesis y haberme animado siempre que no veía forma de terminarla. Desde pequeña me enseñasteis que podía conseguir lo que me proponía y aunque yo nunca llegué a creerlo vosotros siempre habéis creído en mi. Gracias a Jose, Maribel, Carlos, Alicia y Dani, porque habéis visto crecer esta tesis y vuestro ánimo ha hecho que, sin duda, avance más rápido.

Y en general, gracias a todos los que me han acompañado en este viaje, estén nombrados aquí o no. Muchas veces una simple distracción ha hecho mucho más por este trabajo de lo que pensaríais.

Abstract

Medical image processing is an interdisciplinary field in which multiple research areas are involved: image acquisition, scanner design, image reconstruction algorithms, visualization, etc. X-Ray Computed Tomography (CT) is a medical imaging modality based on the attenuation suffered by the X-rays as they pass through the body. Intrinsic differences in attenuation properties of bone, air, and soft tissue result in high-contrast images of anatomical structures. The main objective of CT is to obtain tomographic images from radiographs acquired using X-Ray scanners. The process of building a 3D image or volume from the 2D radiographs is known as reconstruction. One of the latest trends in CT is the reduction of the radiation dose delivered to patients through the decrease of the amount of acquired data. This reduction results in artefacts in the final images if conventional reconstruction methods are used, making it advisable to employ iterative reconstruction algorithms.

There are numerous reconstruction algorithms available, from which we can highlight two specific types: traditional algorithms, which are fast but do not enable the obtaining of high quality images in situations of limited data; and iterative algorithms, slower but more reliable when traditional methods do not reach the quality standard requirements. One of the priorities of reconstruction is the obtaining of the final images in near real time, in order to reduce the time spent in diagnosis. To accomplish this objective, new high performance techniques and methods for accelerating these types of algorithms are needed. This thesis addresses the challenges of both traditional and iterative reconstruction algorithms, regarding acceleration and image quality. One common approach for accelerating these algorithms is the usage of shared-memory and heterogeneous architectures. In this thesis, we propose a novel simulation/reconstruction framework, namely FUX-Sim. This framework follows the hypothesis that the development of new flexible X-ray systems can benefit from computer simulations, which may also enable performance to be checked before expensive real systems are implemented. Its modular design abstracts the complexities of programming for accelerated devices to facilitate the development and evaluation of the different configurations and geometries available. In order to obtain near real execution times, low-level optimizations for the main components of the framework are provided for Graphics Processing Unit (GPU) architectures.

Other alternative tackled in this thesis is the acceleration of iterative reconstruction algorithms by using distributed memory architectures. We present a novel architecture that unifies the two most important computing paradigms for scientific computing nowadays: High Performance Computing (HPC). The proposed architecture combines Big Data frameworks with the advantages of accelerated computing.

The proposed methods presented in this thesis provide more flexible scanner configurations as they offer an accelerated solution. Regarding performance, our approach is as competitive as the solutions found in the literature. Additionally, we demonstrate that our solution scales with the size of the problem, enabling the reconstruction of high resolution images.

Resumen

El procesamiento de imágenes médicas es un campo interdisciplinario en el que participan múltiples áreas de investigación como la adquisición de imágenes, diseño de escáneres, algoritmos de reconstrucción de imágenes, visualización, etc. La tomografía computarizada (TC) de rayos X es una modalidad de imágen médica basada en el cálculo de la atenuación sufrida por los rayos X a medida que pasan por el cuerpo a escanear. Las diferencias intrínsecas en la atenuación de hueso, aire y tejido blando dan como resultado imágenes de alto contraste de estas estructuras anatómicas. El objetivo principal de la TC es obtener imágenes tomográficas a partir estas radiografías obtenidas mediante escáneres de rayos X. El proceso de construir una imagen o volumen en 3D a partir de las radiografías 2D se conoce como reconstrucción. Una de las últimas tendencias en la tomografía computarizada es la reducción de la dosis de radiación administrada a los pacientes a través de la reducción de la cantidad de datos adquiridos. Esta reducción da como resultado artefactos en las imágenes finales si se utilizan métodos de reconstrucción convencionales, por lo que es aconsejable emplear algoritmos de reconstrucción iterativos.

Existen numerosos algoritmos de reconstrucción disponibles a partir de los cuales podemos destacar dos categorías: algoritmos tradicionales, rápidos pero no permiten obtener imágenes de alta calidad en situaciones en las que los datos son limitados; y algoritmos iterativos, más lentos pero más estables en situaciones donde los métodos tradicionales no alcanzan los requisitos en cuanto a la calidad de la imagen. Una de las prioridades de la reconstrucción es la obtención de las imágenes finales en tiempo casi real, con el fin de reducir el tiempo de diagnóstico. Para lograr este objetivo, se necesitan nuevas técnicas y métodos de alto rendimiento para acelerar estos algoritmos.

Esta tesis aborda los desafíos de los algoritmos de reconstrucción tradicionales e iterativos, con respecto a la aceleración y la calidad de imagen. Un enfoque común para acelerar estos algoritmos es el uso de arquitecturas de memoria compartida y heterogéneas. En esta tesis, proponemos un nuevo sistema de simulación/reconstrucción, llamado FUX-Sim. Este sistema se construye alrededor de la hipótesis de que el desarrollo de nuevos sistemas de rayos X flexibles puede beneficiarse de las simulaciones por computador, en los que también se puede realizar un control del rendimiento de los nuevos sistemas a desarrollar antes de su implementación física. Su diseño modular abstrae las complejidades de la programación para aceleradores con el objetivo de facilitar el desarrollo y la evaluación de las diferentes configuraciones y geometrías disponibles. Para obtener ejecuciones en casi tiempo real, se proporcionan optimizaciones de bajo nivel para los componentes principales del sistema en las arquitecturas GPU.

Otra alternativa abordada en esta tesis es la aceleración de los algoritmos de reconstrucción iterativa mediante el uso de arquitecturas de memoria distribuidas. Presentamos una arquitectura novedosa que unifica los dos paradigmas informáticos más importantes en la actualidad: computación de alto rendimiento (HPC) y Big Data. La arquitectura propuesta combina sistemas Big Data con las ventajas de los dispositivos aceleradores.

Los métodos propuestos presentados en esta tesis proporcionan configuraciones de escáner más flexibles y ofrecen una solución acelerada. En cuanto al rendimiento, nuestro enfoque es tan competitivo como las soluciones encontradas en la literatura. Además, demostramos que nuestra solución escala con el tamaño del problema, lo que permite la reconstrucción de imágenes de alta resolución.

Acronyms

AI	Arithmetic Intensity.
AIDR3D	Adaptive Iterative Dose Reduction.
AIR	Algebraic Iterative Reconstruction.
ART	Algebraic Reconstruction Technique.
ASIR	Adaptive Statistical Iterative Reconstruction from GE HealthCare.
BI-ART	Block Iterative Algebraic Reconstruction Technique.
BI-SMART	Block Iterative Simultaneous Algebraic Reconstruction Technique.
BiCGStab	BiConjugate Gradient Stabilized.
CAD	Computer Aided Design.
CARM	Cache-Aware Roofline Model.
CAT	Computed Axial Tomography.
CBCT	Cone-Beam Computed Tomography.
CE	Compute Engine.
CNN	Convolutional Neural Network.
COMPSs	COMP Superscalar.
CT	Computed Tomography.
CU	Compute Unit.
DDO	Distance Detector Object.
DMDA	Distributed Memory Distributed Array.
DSO	Distance Source Object.
DVFS	Dynamic Voltage and Frequency Scaling.
FBP	Filtered Backprojection.
FDK	Feldkamp,Davis, and Kress.
FMA	Fused Multiply Add.
FOV	Field Of View.
FP	Focal Plane.
FPGA	Field-programmable Gate Arrays.
FPU	Floating Point Unit.
GFS	Google File System.

GPC	Graphics Processing Clusters.
GPU	Graphics Processing Unit.
HDFS	Hadoop Distributed File System.
HPC	High Performance Computing.
IRIS	Image Reconstruction in Image Space.
MBIR	Model Based Iterative Reconstruction.
MIMD	Multiple Instruction Multiple Data.
ML-EM	Maximum Likelihood Expectation Maximization.
MPI	Message Passing Interface.
MRI	Magnetic Resonance Imaging.
PAPI	Performance API.
PICCS	Prior Image Constrained Compressed Sensing.
RDD	Resilient Distributed Dataset.
RMSE	Root Mean Square Error.
ROI	Region Of Interest.
SAFIRE	Sinogram Affirmed Iterative Reconstruction.
SART	Statistical Algebraic Reconstruction Technique.
SCCS	Surface Constrained Compressed Sensing.
SFU	Special Function Unit.
SIMD	Single Instruction Multiple Data.
SM	Streaming Multiprocesor.
SNR	Signal to Noise Ratio.
TBB	Intel Threading Building Blocks.
TV	Total Variation.
VOI	Volume Of Interest.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Medical image processing field includes topics from different disciplines ranging from medicine or electrical engineering to computer science. In the past years, new methods and techniques for the acquisition of medical images have been developed. The growth in the usage of digital equipment and detectors in all imaging modalities has also increased the possibilities of introducing digital image processing techniques at different stages. Additionally, the innovations achieved through the development of new image scanners and acquisition geometries demand new protocols and advances in reconstruction algorithms. The main objective of these advances is to obtain better images, in terms of quality, at faster rates. This applies also to the world of CT, an imaging modality based on the acquisition of images using X-Ray properties that has been used for decades to obtain tomographic images of objects, animals or patients. It is widely employed in non-destructive testing to recover a digital model of the object inspected, but its most famous application is Computed Axial Tomography (CAT) in health care.

In this modality, X-Ray images are obtained from the object or patient scanned from different positions, traditionally, from different angles around the object of interest. These X-Ray radiographs, also known as projections, are then digitally processed or reconstructed to recover the final tomographic image or volume.

Limitations of CT in certain medical circumstances, such as operations, in which images from all angles can not be acquired, have increased the amount of research on new acquisition protocols for these X-Ray systems in limited-data situations. Additionally, the increased use in the last decades of X-Rays as a diagnosis tool has created a trend that investigates the possibilities of reducing the amount of data acquired in order to decrease the radiation dose received by the patient. This radiation dose has been recently related to an increased cancer risk for different types of patients [1, 2]. In these situations all information desired from the patient can not be obtained. New acquisition protocols and X-Ray scanners, like the one shown in Figure 1.1, require the implementation of new reconstruction methods, capable of obtaining images with similar quality to those obtained in ideal situations.

Part of this thesis will focus on a simulation/reconstruction platform, capable of assessing the effectiveness of these new configurations, simulating the acquisition of these images and, at the same time, designing and evaluating different reconstruction algorithms over these data in terms of quality and execution time.

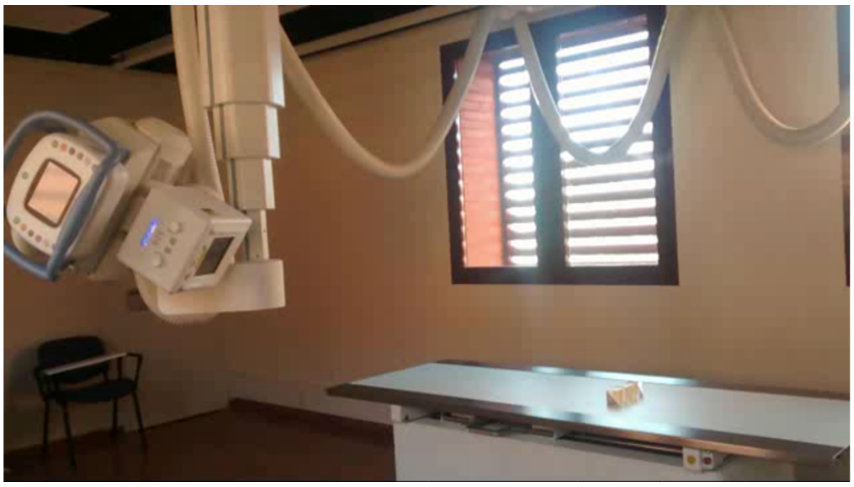


Figure 1.1: *Photo of a flexible X-Ray scanner. The X-Ray source can be moved and rotated around the room with a fixed bed in which the patient can lay.*

1.1 Motivation

The design of new acquisition protocols and configurations requires of the possibility of changing the acquisition parameters in a flexible manner. Although in most X-Ray scanners positions can be configured to meet different requirements, the acquisition of the images can represent a long process not suitable for research purposes. Additionally, exact positioning of these machines implies an additional calibration step. Therefore, advantages of emulating the operation of a flexible X-Ray scanner are straightforward since there are not any physical limitations in terms of positioning or acquisition times. The development of new flexible X-Ray systems benefits from computer simulations, an environment that enables performance to be checked before expensive real systems are implemented. Along with the new acquisition protocols, new reconstruction algorithms must be applied to obtain the final tomographic image.

Nevertheless, the task of developing a platform for simulating the acquisition of X-Ray images and reconstructing them has several difficulties to be tackled. First, in terms of emulating the physical effects, a realistic simulation of the acquisition of the X-Rays can not be done efficiently. Since for the development of new configurations and acquisition protocols the critical point is the geometrical positioning of the elements, a simplified model is required. Second, digital detectors have increased their size and density in the last years, thus incrementing the size of the data set to be processed. Luckily, memory and computational specifications have also increased and new hardware architectures have made it possible to obtain images in almost near real-time. The final problem is related to the new advanced reconstruction algorithms that have appeared recently. These algorithms, designed to work with limited-data configurations, are also computationally more expensive because of their extensive use of iterative methods.

For these reasons, it is necessary to explore the possibilities of developing these algorithms in non-traditional computing platforms with programming models and paradigms that can provide the necessary computational resources to obtain these images in lower times than those obtained in conventional systems.

The traditional approach consisted on porting this type of applications to HPC programming models, which are centered around the final performance obtained and the exploitation of the hardware and systems constructed for them. Programming models such as Message Passing Interface (MPI), OpenMP, Intel Threading Building Blocks (TBB), OMPSS and those including support for heterogeneous platforms, OpenACC, CUDA or OpenCL, have been widely employed to implement scientific applications. Nevertheless, in the last decade, a new paradigm that changes the focus from performance to throughput in data intensive applications, has appeared. The Big Data paradigm and its related programming models, from which we can highlight MapReduce, has attracted the attention of the users. Some of the advantages of the frameworks supporting the Big Data paradigm are: a better support for more scientific domain-like programming languages, a programming model much easier to exploit and implement than in HPC, automatic data management or fault tolerance. Image processing applications can be ideal candidates for the Big Data era. Most of the algorithms related to this field can be characterized as loosely coupled or even embarrassingly parallel algorithms, an optimal target for these frameworks. Moreover, the large quantity of images to be processed and their increasing sizes transform them into Big Data Applications. However, the main algorithms employed in medical image processing also require performance, which is not the main objective of the Big Data frameworks designed today. Therefore, medical image processing algorithms can be employed as use cases in the process of convergence between the Big Data paradigms and HPC.

The hypothesis of this thesis can be summarized in:

It is possible to develop new techniques for the reconstruction of high quality CT images in situations were the input data is limited or is not complete, such as in low-dose scenarios, obtaining the results in near real-time compared with previous traditional computation methods.

Throughout this thesis, we will work on the progressive optimization and implementation of different platforms supporting multiple reconstruction algorithms. Furthermore, different combinations of programming models, in search of a model of convergence, will be explored.

1.2 Objectives

The main objective of this thesis is the development of new techniques for reconstruction in situations where the acquired data from the scanner is not suitable for traditional reconstruction methods, in an optimized and flexible platform that can provide a meaningful reduction of the time spent for obtaining the final reconstructed image.

Additionally, other related objectives will also be fulfilled:

O1 **To design a flexible simulator/reconstructor for CT medical image processing with support for heterogeneous architectures**. We will strive towards a unified platform providing simulation and reconstruction capabilities for X-Ray images. It must be flexible in terms of X-Ray geometries and hardware support in order to be executed in all sort of environments with high performance capabilities.

O2 **To improve the performance of simulation/reconstruction algorithms**. To fully exploit the hardware it is necessary to obtain metrics and measures from the simulation/reconstruction platform that can provide an idea of the limitations of the design in order to overcome them and propose new heterogeneous hardware-driven optimizations than can increase the performance of the implemented algorithms.

O3 **To design an iterative reconstruction platform for low-dose systems**. From the previous optimized designs it is possible to extend the platform to support different types of iterative reconstruction algorithms. This platform must be easily maintainable and configurable to allow user-friendly experiments with multiple methods.

O4 **To reconstruct high resolution images**. At the end, we can provide the application with further performance and programmability enhancements employing novel frameworks. These frameworks can be combined with different HPC techniques to accelerate the execution of the algorithms and support the reconstruction of high resolution images.

1.3 Structure of the document

This thesis is structured as follows:

- Chapter 2 *State of the Art*: contains an extended state of the art of the topics related to this thesis. It also includes a brief background on medical image processing algorithms and programming models.
- Chapter 3 *Proposal of a new fast and flexible X-Ray simulator*: describes the design and implementation of FUX-Sim, an X-Ray simulation/reconstruction platform. A full description of its architecture, the different programming models supported as well as a performance evaluation are included.
- Chapter 4 *Enhanced cache-aware roofline model for GPU kernel characterization*: presents further optimizations for the main components in FUX-Sim, with a methodology based on the characterization of the application using Cache-Aware Roofline Model (CARM).
- Chapter 5 *Design of a fast iterative reconstruction framework for limited-data CT*: contains the description of the iterative algorithms implemented and the description of the reconstruction framework. A performance evaluation of all the algorithms comparing GPU and CPU implementations is also provided.
- Chapter 6 *A Distributed Iterative Reconstructor*: contains a description and evaluation of the implementation of an iterative reconstruction algorithm in a distributed environment. This chapter focuses on MPI and a specific library, PETSc, employed in this implementation.
- Chapter 7 *Iterative reconstruction framework based on the Big Data paradigm*: describes the steps to translate parts of an iterative reconstruction algorithm to Big Data frameworks, in this case Apache Spark. It presents an extended architecture of the framework that supports GPUs and several evaluations in CPU and GPU-based architectures.
- Chapter 8 *Conclusions*: contains a summary of this thesis, the main conclusions obtained from its development and the contributions made. Possible future directions for research related to the topic covered in this thesis are described.

Chapter 2

State of the art

This chapter includes an overview of the different fields related with this PhD thesis. Due to the interdisciplinary nature of this work, we will introduce each of the topics with a brief, but complete, background description. In the first section, we will explore the medical image processing field, focusing on X-Ray and CT reconstruction algorithms. We provide a review of the existing solutions and their main characteristics in comparison with the solutions presented in this thesis. These solutions take advantage of different programming models and specialized hardware architectures that will be explained in a second section. A literature review of this topic is also provided, with special emphasis in HPC, Big Data paradigms, and heterogeneous computing with their corresponding programming models.

2.1 Medical image processing

Medical image processing is an extensive field containing different techniques and disciplines whose main objective is to obtain and process images for medical diagnosis. It includes research from different backgrounds, such as signal processing, electric engineering, computer science, mechanical engineering, etc. In the scope of this thesis we will focus on X-Ray medical image processing, although most of the optimizations and techniques employed here can be applied to other type of image modalities.

The main objective of medical image processing methods is to modify and improve image signals taken for medical purposes [3]. It covers different image modalities, which obtain the images from different physical properties. Some of them employ magnetic fields to obtain internal physical images of an object or body, as is the case of Magnetic Resonance Imaging (MRI)). Others use X-Rays measurements to obtain an image based on the different attenuation ratios of the materials scanned, like in the case of CT. Although these methods can also be used for other purposes apart from those strictly medical (diagnosis), the requirements in terms of quality and the techniques for the image acquisition and posterior computational processing differ greatly depending on the final objective of the image/study. Accordingly, we will focus on those requirements specific to medical purposes, since they are the ones explored in this thesis.

More concretely, algorithms and applications presented here are related to Cone-Beam Computed Tomography (CBCT). This modality consists on the construction of tomographic images, internal images of an object or a patient, combining the information of multiple X-Ray images (commonly referred as projections) taken from different θ angles around the scanned object of interest (see Figure 2.1). The cone-beam term refers to the geometry formed by a beam of rays

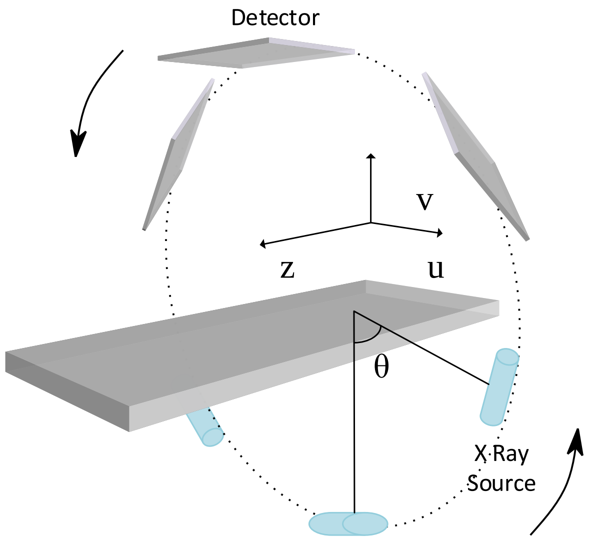


Figure 2.1: *An example of how a CT Scanner works. The detector panel, normally hidden inside the machine, and the X-Ray source rotate around the patient, normally lying in bed.*

going from the X-Ray source to the detector panel, which creates a cone geometry. This detector panel registers the signal as a radiography, similar to the one shown in Figure 2.2, which is later processed through different algorithms to construct the final tomographic image. This tomographic image is also known as a volume due to its 3 dimensional characteristics. This volume is divided into different values that represent the density of the object at that point (voxel).

The basic operation of the CT scanner shown in Figure 2.1 is not unique. Different acquisition techniques have been developed to allow the detector and source to move independently, or, even shift and rotate the bed in different angles.

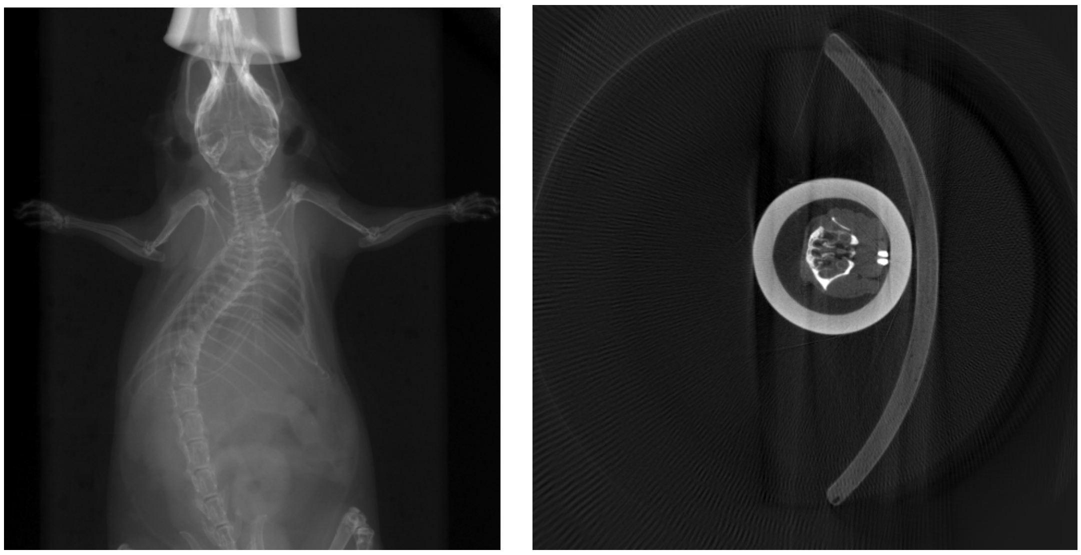


Figure 2.2: *Input radiography for the reconstruction algorithm (left).Reconstructed image of a rat (right).*

2.1.1 Traditional reconstruction algorithms

Traditional reconstruction algorithms are based on mathematical principles and consist on the correspondence of the reconstruction problem to the solution of a system of linear equations [4] of the type:

$$Ax = b \tag{2.1}$$

where A is the system matrix simulating the operation of acquiring images from a CT scanner, x is a vector containing the intensities of the final images, and b is the vector containing the set of values taken from the projections obtained from the scanner. The size of matrix A is $M \times N$, being M the number of pixels per projection multiplied by the number of projections; and N, the number of voxels of the final image. Therefore, x is a column vector of size $N \times 1$ and b another column vector of size $M \times 1$.

With this system matrix, we can obtain the reconstructed values of the final images based on the acquired projections in a single matrix-vector multiplication. However, obtaining this system matrix is time-consuming task and it depends on the geometrical factors of the acquisition, so it must be recomputed for each study. Additionally, it occupies a great amount of space for relatively small volumes (e.j a 64^3 elements of the reconstructed volume from $64^2 \times 720$ projection size given a system matrix occupying 2,880 GB) although it can be stored in compressed format due to its sparsity characteristics. The alternative employed in traditional reconstruction methods is the application of geometrical relations between the input data and the final reconstructed volume without pre-computing this system matrix [5]. One of the most well-known algorithms included in these traditional methods is the Filtered Backprojection (FBP) or Backprojection-Filtered algorithms. Both of them consist on two basic stages: filter and backprojection. The filter stage can be applied before or after the backprojection and its main objective is the enhancement of the acquired projections, if applied before; or of the final result, if carried out after. There are different types of filters that can be applied at each step. However, the most effective is the ramp-filter, which increases the value of higher frequencies in output or input images avoiding the apparition of blurring artefacts. The second stage, the backprojection, is the one that properly reconstructs the image, gathering information obtained from the initial radiographies. The way in which this information is gathered determines the type of backprojection algorithm employed. Canonically, this is performed through a transformation named Siddon's algorithm [6]. This algorithm is capable of calculating lengths of intersection with each of the voxels of the image that we are reconstructing. This is done simulating the exact path that each ray is taking, from the source to the detector. This approach can be adapted to different interpolation modes that transform the algorithm making the rays pass through specific structures. In Chapter 3, two of these interpolation modes will be addressed along with a full explanation of the backprojection kernel.

Another important stage involved in reconstruction algorithms is the projection. This operation takes a reconstructed volume and obtains projections from it simulating the operation of the scanner. Both backprojection and projection operations constitute the main components or kernels of many reconstruction applications.

Projection algorithms

Projection algorithms emulate the process of data acquisition in an X-Ray scanner, given an initial 3D volume and the geometrical parameters (distance from the detector to the object,

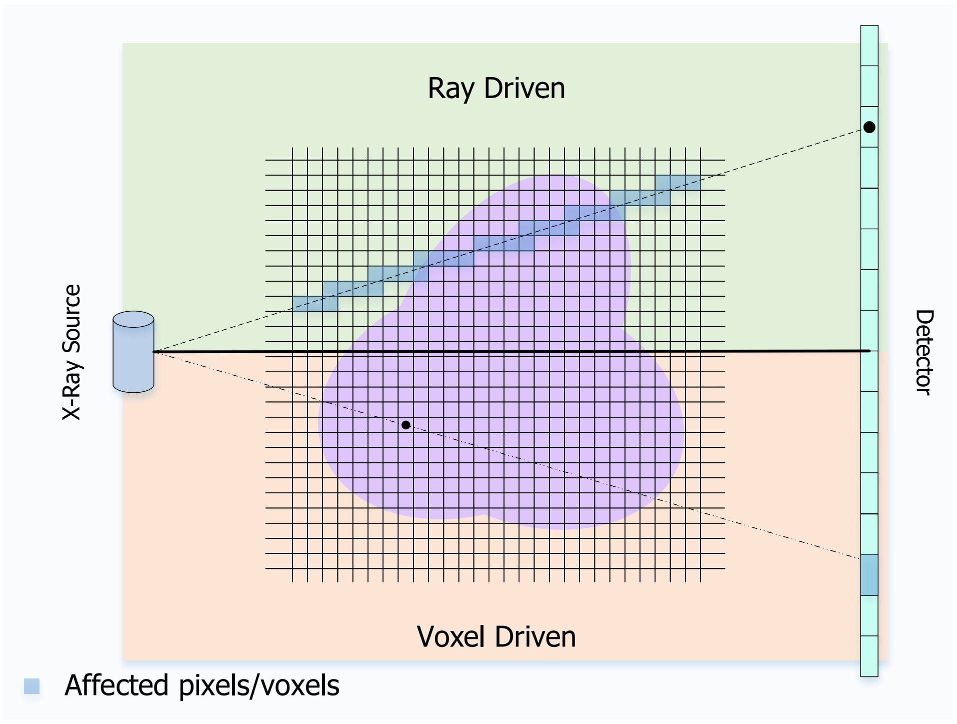


Figure 2.3: *Schematic of two different interpolation algorithms for reconstruction: ray driven (upper half) and voxel driven (lower half).*

distance from the source to the object, size of the volume, size of the projection, positions...) of the system. There are different methods to obtain the projected radiographs, although most of them consist on the computation of diverse trajectories of the X-Rays from the simulated source to the panel while traversing the 3D volume. Depending on the material and geometrical characteristics, these rays are affected by different phenomena. Correctness of the final output depends also on the type of interpolation used as well as the number of rays simulated.

In our case, we have focused on ray-driven and distance driven interpolation methods. Ray-driven methods are based on the computation of the trajectory of a ray from the X-Ray source to the center of the detector cell to be computed. The value obtained in this detector cell will be then the sum of $N \cdot step$ values along the X-Ray beam. A representation of this interpolation mode can be found in the upper half of Figure 2.3.

In general, the mathematical operations that describe this process are shown in Equations 2.2 and 2.3.

$$p(s,z,\theta) = \int\limits_{v=-\infty}^{\infty} vol\left(s \cdot cos(\theta) - v \cdot sin(\theta), s \cdot sin\theta + v \cdot cos\theta, W \cdot z\right) dv \tag{2.2}$$

$$W = \frac{DSO - v}{DSO + DDO} \tag{2.3}$$

where $p(s,z,\theta)$ is the value of the projection in position s, z at angle θ, vol is the 3D volume, and v is the position at axis v (see Figure 2.1). W can be identified as the magnification factor produced by the cone-beam geometry, Distance Source Object (DSO) is defined as the distance between source and object and Distance Detector Object (DOD) as the distance between detector and volume. Depending on the interpolation method s, v and z will be computed at different positions of the volume and detector.

Backprojection algorithms

As already explained, the backprojection algorithm builds a volume from different projections or radiographies obtained from the scanner or from the projection algorithm. The most employed backprojection methods are commonly based on the FDK algorithm [7] for cone-beam geometries. This algorithm is a version of an FBP algorithm in which voxel-driven interpolation is implemented. In voxel-driven interpolation, rays pass through the center of each of the voxels that form the volume. The value of this voxel will be the integral of all the values of the projections to which these rays intersect. The process for one projection is shown in the bottom half of Figure 2.3. A simplified mathematical formulation of the backprojection algorithm is shown below in Equations 2.4, 2.5 and 2.6:

$$f(u,v,z) = \frac{1}{2}\int_{\theta=0}^{2\pi} W_2(v)\left[\int_{w=-\infty}^{\infty}\left(\int_{s=-\infty}^{\infty}[p(s,z,\theta)\cdot W_1(z,u)]\cdot e^{-j2\pi sw}ds\right)\cdot|w|\cdot e^{j2\pi ws}\cdot dw\right]\cdot d\theta \tag{2.4}$$

$$W_1 = \frac{DSO}{\sqrt{DSO^2 + z^2 + u^2}} \tag{2.5}$$

$$W_2 = \frac{DSO}{(DSO - v)^2} \tag{2.6}$$

where $f(u,v,z)$ is the value of the image at position u,v,z, θ is the angle from which the projection is taken, $p(s,z,\theta)$ is the value of the projection at position (s,z), and W_1 and W_2 are the weighting factors introduced to compensate for the different ray lengths. DSO 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.

2.1.2 Iterative reconstruction algorithms

Although traditional reconstruction algorithms are still in use, advanced reconstruction algorithms have been developed in the last years. One of the objectives of these advanced reconstruction algorithms is to work with limited-data or limited angle span (total angle arc covered by the projection data). With this purpose, the most successful reconstruction algorithms have been the reconstruction algorithms based on iterative methods. Normally based on previous analytical methods, its main purpose is to continuously improve the quality of the final image through an iterative process. In this iterative process, prior information can be included to generate a better image. Each iteration of these algorithms consists roughly of the same steps that can vary depending on the purpose of the algorithm or the type of data that it has been fed on. A generic workflow for describing an iterative reconstruction algorithm contains the following steps:

1.- Obtaining of an initial guess: this initial guess consists of an approximated volume or even a dummy volume (zero initialized volume). In many cases, a traditional algorithm is employed to obtain this approximation, for example a FBP algorithm.

2.- Image processing: different characteristics of the volume are inferred or enhanced. In this step, prior information can be included, either providing a previously reconstructed volume of the patient or limited information, such as the delimitation of the volume with respect to the background.

3.- Projection of the volume: projections from the computed volume are acquired obtaining radiographs of the estimated image.

4.- Correction of the volume: based on the comparison of the obtained radiographs with the input projections, the radiographs are corrected. The projections can then be backprojected to obtain the corrected volume to which successive computations will be applied.

From the corrected projections new estimated volumes are generated, and the process is repeated iteratively, until the algorithm reaches the defined quality threshold. In many cases, this process also has to be supervised due to the possible overcorrection of the final images, which can lead to blurred images or the apparition of new artefacts or distortions in the image.

There are three types of iterative reconstruction algorithms [8]:

- Algebraic Iterative Reconstruction (AIR) algorithms: the exponent of the non-statistical group is the Algebraic Reconstruction Technique (ART). This technique consists on considering each ray from each projection and iteratively increase their quality until the difference between the projection of the estimated volume and the input data is consistently reduced. It is based on the resolution of Equation 2.1. This technique [9] has demonstrated to be good enough and reduce artifacts with less computing power than other techniques. However, it does not consider the additional noise produced by the process [10]. There are several variants of this technique introducing new operations or dividing the data in blocks that are processed simultaneously to accelerate the process and reduce the execution time. Examples of these variants are Block Iterative Algebraic Reconstruction Technique (BI-ART), and Block Iterative Simultaneous Algebraic Reconstruction Technique (BI-SMART) [11].

- Statistical iterative reconstruction algorithms: these iterative methods modify the previously mentioned equation to include the noise:

$$Ax + \epsilon = b \tag{2.7}$$

  where ϵ represents the noise. With this new formulation it is possible to account for the noise present in the images either by predicting or by modelling it. With an accurate model of the noise, then it is possible to remove it from the final image obtaining a volume with higher quality. This is specially useful in situations of limited-data or in which the angular span of the projections is narrow (120 degrees or less). These algorithms can also be divided in two groups: the ones that model the noise following a Gaussian distribution and the ones that model the noise following a Poisson distribution. Both of them try to obtain the final image that maximizes the probability of being the correct one. These methods assume that it is not possible to recover the true volume from the input data, so, statistical computation is applied to recover at least the closest one. At the end, we can convert the algorithm to an optimization problem that can be solved with the same methods employed in other similar statistical problems; for example, with conjugate gradient methods or steepest descent algorithms. Some of the algorithms belonging to this category are Maximum Likelihood Expectation Maximization (ML-EM) algorithms [12] and its different variants. The most used iterative methods nowadays are commercial products such as Adaptive Statistical Iterative Reconstruction from GE HealthCare (ASIR) from GE HealthCare [13], Model Based Iterative Reconstruction (MBIR) [11], Adaptive Iterative Dose Reduction (AIDR3D) [14],Image Reconstruction in Image Space (IRIS) [15], Sinogram Affirmed Iterative Reconstruction (SAFIRE) [16] and iDose (Philips) [17].

- Machine learning based algorithms: this last category contains the most recent algorithms based on models for reconstruction or image enhancement obtained through machine learning techniques, mainly using deep neural networks and especially Convolutional Neural Networks (CNNs) [18, 19, 20].The use of machine learning and deep learning algorithms have grown in the last years due to the increased performance given by the new developments in hardware and software. Tney have been proven useful in computer vision application and image and natural language processing. In the field of medical image applications some of them have been focused on merging traditional reconstruction algorithms or other types of iterative reconstruction algorithms for image enhancing [21, 22, 23], or even to estime CT reconstructed images from other medical image modalities like MRI [24]. However, this type of algorithms are still in an early stage and their applications are expected to grow in the next years.

2.2 Heterogeneous and homogeneous architectures

One of the objectives of this thesis is the design of new platforms for simulation of iterative reconstruction mechanisms in heterogeneous architectures by using high performance techniques. Performance can be obtained through multiple mechanisms: from the choice of the programming language in which the algorithm will be implemented to the hardware platform in which they will be executed. In general, the decision on the hardware architecture limits the possibilities in terms of programming languages and programming models that can be applied.

The need for carrying out specific tasks led to the design of unique hardware architectures for their execution. One of the first examples of this trend was the creation of the Floating Point Unit (FPU), which, although nowadays it is introduced into the processor, started as a co-processor in charge of particularly computing floating point operations. As new applications and algorithms were developed, other co-processors started to appear in the market. Nowadays, the most famous co-processor already present in almost all computers is the GPU, initially designed to manage the computer graphics. Their inclusion in a wide variety of systems has extended the concept of heterogeneous computing.

The use of a combination of different systems for purely computing is also not new. From the FPUs to the use of Field-programmable Gate Arrayss (FPGAs), scientific applications seeking high performance have been employing heterogeneous architectures. Apart from performance and purpose, the main difference between designing applications for homogeneous architectures or heterogeneous architectures is normally the level of programming difficulty. Programming for homogeneous architectures does not require to consider the architectural differences including an easier management of the data. Meanwhile heterogeneous computing normally requires specific libraries and a better knowledge of the underlying architecture. Heterogeneous architectures also have an extra difficulty in which they normally work with distributed memories that must be manually managed in order to obtain full performance. These difficulties do not only affect the implementation of applications for heterogeneous architectures but also imply a higher difficulty in the optimization phase that leads to the obtaining of less than the expected performance.

To overcome these problems, novel programming models have appeared in the last years attempting to facilitate programming for this type of architectures and to decrease the complications of optimizing the applications for specific hardware systems. Examples of these types of programming models will be seen in next sections.

Accelerators

Accelerators are one of the most used resources in computer hardware. They are designed to accelerate the execution of certain tasks increasing the performance obtained. There are many types of accelerators: for graphics tasks, for HPC tasks, for signal processing tasks, etc., even for general algorithms in the form of FPGAs.

Nowadays, probably the most popular accelerators are GPUs. At first designed for graphics management, their highly-parallel architecture and their support for fast floating point computations made them a perfect candidate for scientific computing and HPC. It is employed in numerous general purpose applications and its use has increased for the execution of machine-learning applications [25] and bitcoin mining [26, 27]. A standard architecture for GPUs does not exist, differing greatly between different vendor implementations and purposes. However, all GPUs are characterized by having a large number of processing units in which work can be parallelized. They can be classified into two main groups: desktop GPUs and mobile GPUs. Desktop GPUs can be found in all types of computers, from supercomputer nodes to laptops. Mobile GPUs are included in the same chip as CPU units in mobile processors and can be found in smartphones and micro computers such as Raspberry Pis or Odroids.

In the field of desktop GPUs there are two main vendors: NVidia and AMD. Both of them follow the same principle, to have a large number of processing units or cores inside the accelerator. The concept of core in GPUs is slightly different from the one employed in CPUs. A core in a CPU contains several units in charge of different kind of computations as well as architectural optimizations to cache, predict and dispatch instructions in order to execute faster. These units are not usually present in the GPU cores and if present they are included in a smaller amount than in CPUs. These GPU cores are grouped in blocks in which other shared hardware units are included. Therefore, caches, memory controllers, instruction decoders or registers are shared between multiple cores. This additional simplicity with respect to traditional microprocessors makes it possible to obtain devices with thousands of cores, when even high-end CPUs are only reaching 256 processing units.

In Figures 2.4 and 2.5 we show two representative examples of architectures of NVidia GPUs. The first aspect to notice is the division of the GPU in Graphics Processing Clusterss (GPCs), each containing all the components necessary for graphics processing. Between these components, we have the texture units, Special Function Units (SFUs) and most importantly, the SMs. Texture units are in charge of the specific pipeline for the texture mechanisms, a special data structure optimized for image processing. These texture units include additional optimized operations like pixel and voxel interpolation or spatial locality for 2D and 3D structures. Thus, the access to this type of data structures is faster than a user specific implementation in software of similar funcionality. SMs are in charge of executing the instructions, containing several cores, whose number depends on the type of GPU and the architecture generation selected. Typically, these cores are composed of FPUs for operations with 32 and 64 bit decimal numbers, arithmetical integer units and for the newest architecture (see Figure 2.5) even tensor cores for machine learning applications. This SM also contains a shared register file for all cores, which represents one additional restricted resource that has to be taken into account by the programmer. Another shared resource that is shared is the shared memory, a memory region programmed implicitly by the programmer that acts as a cache. Its use its advisable to avoid repeated access to the Global Memory of the GPU. Additionally, it also possess a L1 level cache, local to the SM and a L2 level cache, shared for different SMs, which are managed automatically by the GPU driver.

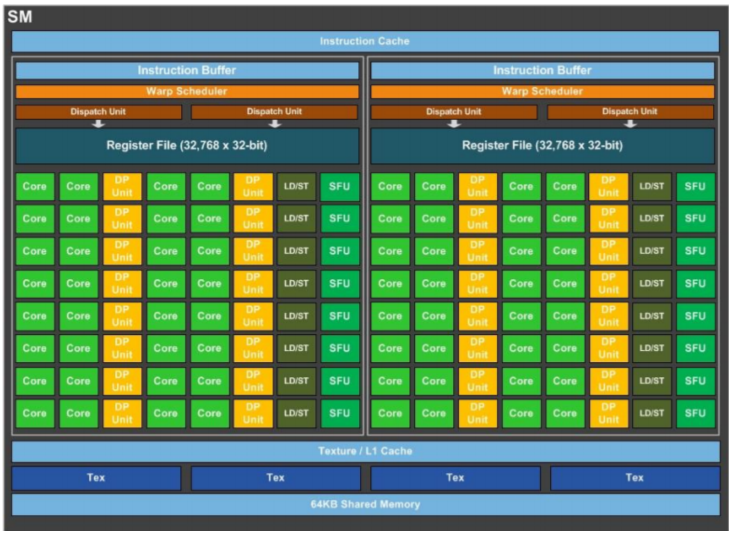


Figure 2.4: *Pascal architecture for the SM, NVidia [28].*

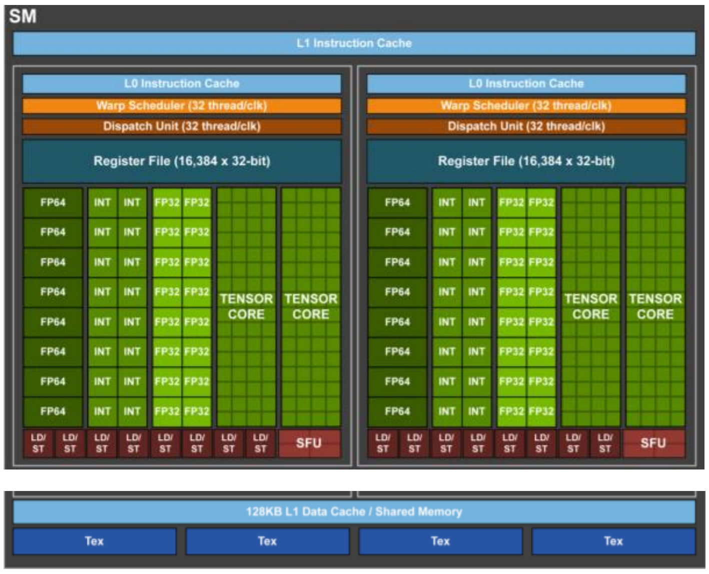

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure 2.5: *Volta architecture for the SM, NVidia [29].*

In the last years, apart from the addition of the previously mentioned enhancements for accelerating deep learning algorithms, NVidia GPU architecture does not change drastically from generation to generation. However, one thing to highlight is the disappearance of specific texture cache units in the last generations. Further details and the implications of this decision will be given in Chapter 4.

With respect to other vendors like AMD, the architecture is similarly organized. Instead of cores it has Compute Units (CUs) and Compute Engines (CEs), as we can see in Figure 2.6, instead of SMs. Memory hierarchies also are very similar, with a local L1 cache and a second shared level. Main device memory technologies have advanced in the same direction with support for GDDR5 and HBM memories. These technologies are normally faster with higher bandwidth than those that can be found in commodity computers, in order to be able to serve all the necessary data to the graphic cores. With this fast memories and the large number of cores present in these devices, the execution model employed is closer to the Single Instruction Multiple Data (SIMD) model as opposed to the Multiple Instruction Multiple Data (MIMD) model employed in multi-core systems.

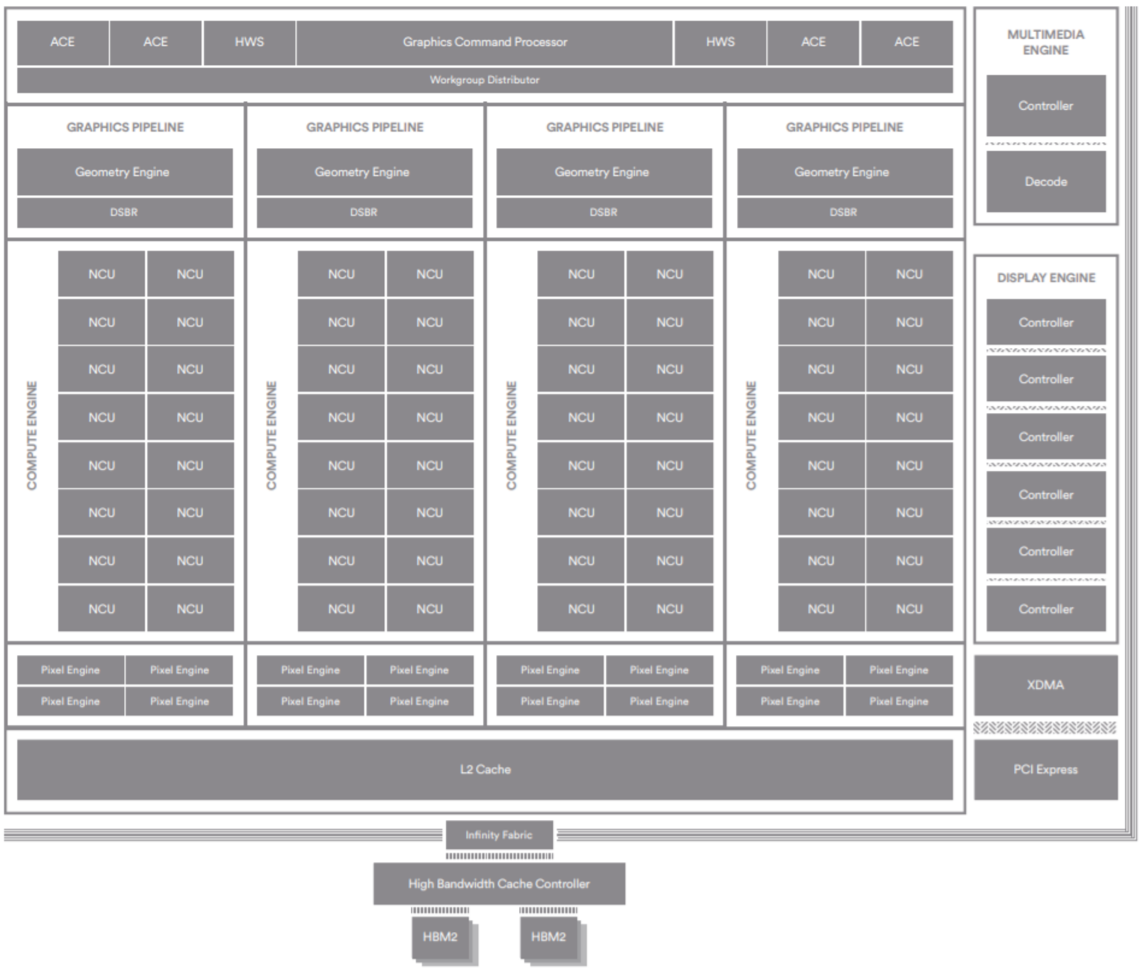


Figure 2.6: *Vega architecture, AMD [30].*

The combination of the differences in the execution models and the management of a separate memory hierarchy make programming these accelerators more difficult. Also, the lack of a standard programming model affects the portability of applications, which in some cases are restricted to a single vendor.

Another type of accelerator that has appeared in the last years is the coprocessor Intel

Xeon Phi. This coprocessor integrates a manycore architecture [31] that is composed by a larger number of cores than standard processor architectures, but without reaching the level of parallelism of GPUs. They also have a separate memory space, although last versions also are sold as main processors[32], thus converging with standard Intel processors. Their internal architecture is very similar to that of standard microprocessors (x86 architecture) with the exception of larger vectorization units to increase data parallelism. As many other Intel microprocessors; Intel Xeon Phi also possess the hyperthreading feature, which can expand the opportunity of parallelization to up to 256 threads.

Finally, there is the possibility of using FPGAs. FPGAs represent the more general solution for accelerating the execution of scientific applications. They enable the possibility of specifically programming the hardware for the algorithm to be executed thanks to their reconfigurable memories and datapaths [33]. They have recently been used for their high efficiency in terms of performance per watt [34]. It is the main competitor of GPUs in the field of energy efficiency, however, their main disadvantage is their increased complexity in terms of programming. Additionally, their price is normally higher than the price of a commodity GPU and it requires of specific programming tools that can increase the final price for development. Even so, the increased performance offered by FPGAs and the high variety of algorithms that need real time performance has increased their popularity, being used in a wide range of applications, from image processing [35] to deep learning algorithms [36]. Several architectures exist for FPGAs [37] as well as different programming methods. Even technology (SRAM-based, Flash or Antifuse) changes between different vendors and categories making it more difficult to standardize. However all of them are based on the reprogrammability of the different components of the board, including logical and I/O components.

As an evidence of the increasing importance of accelerators in scientific computing, the list of the TOP 500 supercomputers in the world is shown in Table 2.1. More than half of the top ten supercomputers includes an accelerator. From these supercomputers, the most used accelerators are the GPUs from NVidia with a residual presence of Intel Xeon Phi.

Programming models

Programming for different architectures poses a challenge that is increased when portability is desired [39]. In the last years a large number of programming languages and models have appeared with the objective of decreasing the difficulty of programming for heterogeneous systems. Some of them were created in order to totally substitute other programming models while others prefer to offer an adaptation to current programming languages without changing significantly the original applications.

The support for accelerators implies an inherent change of paradigms mainly due to the execution model employed and the existence of a separate memory hierarchy that has to be independently managed.

There are many examples of programming models created for programming for accelerators. They can be found for any kind of programming languages, although we can observe that most of them are mainly adapted for C and C++ languages, natively, with some of them still having support for Fortran due to its importance in legacy, scientific applications. We can classify these programming models in three main categories:

- Annotation-based: these programming models are meant to reduce the modifications on the original application code annotating the changes needed to adapt the code to the hardware

Table 2.1: *Top ten most powerful supercomputers from the TOP500 ranking as of June 2018 [38].*

Position	Name	TFlop/s	Power (kW)	Nvidia GPUs	Intel Xeon Phi
1	Summit DOE/SC/Oak Ridge National Laboratory United States	122,300.0	8,806	✓	
2	Sunway TaihuLight National Supercomputing Center in Wuxi China	93,014.6	15,371		
3	Sierra DOE/NNSA/LLNL United States	71,610.0		✓	
4	Tianhe-2A National Super Computer Center in Guangzhou China	61,444.5	18,482		
5	AI Bridging Cloud Infrastructure (ABCI) National Institute of Advanced Industrial Science and Technology (AIST) Japan	19,880.0	1,649	✓	
6	Piz Daint Swiss National Supercomputing Centre (CSCS) Switzerland	19,590.0	2,272	✓	
8	Titan DOE/SC/Oak Ridge National Laboratory United States	17,590.0	8,209	✓	
8	Sequoia DOE/NNSA/LLNL United States	17,173.2	7,890	✓	
9	Trinity DOE/NNSA/LANL/SNL United States	14,137.3	3,844		✓
10	Cori DOE/SC/LBNL/NERSC United States	14,014.7	3,939		✓

in the form of pragmas or annotations. They imply an automatic transformation of the code at compilation or pre-compilation time. However, due to the simplification of the code written and the limitations of the annotations it is impossible to totally manage the execution of the program by the programmer. The optimizations that can be made to the code are minor and therefore is difficult to obtain the full performance from the available hardware. Additionally, the parallel annotated versions can not assure equivalence with the sequential version of the application.

- Library-based: they rely on the use of libraries or modules including several functions to control the hardware and execute the code on it. Their use requires a larger number of modifications or even a reimplementation of the entire application. Programming models requiring the execution on non traditional architectures, like accelerators, are placed in this category.
- Parallel programming languages: they are programming languages designed with parallelism in mind. In this category no extra libraries or annotations are needed because the programming language already provides in its syntax all the tools to program in parallel. Exponents of this category would be Chapel [40] and Julia [41].

Some programming models can fit in one or more categories as well as parallel programming languages can provide additional levels of parallelism through other mechanisms.

One of the programming models most employed in scientific applications is OpenMP [42]. It is an annotation-based programming model widely used for C, C++ and Fortran applications. Apart from supporting the use of accelerators for offloading computation, although only from version 4.0, its main objective is the exploitation of the parallelism supported by the hardware. As we can see in Listing 2.1, it targets *for loops* whose iterations can be executed independently. These iterations then, can be executed by different execution threads in the microprocessor or as a separate piece of code, or kernel, in an accelerator. To mark the code to be parallelized, a pragma is located just before the parallel region, with additional annotations about the data employed inside the region or the target in which the code will be executed.

Listing 2.1: *Offloaded code block example for OpenMP 4.0.*

```
int main(){
        float a[N], b[N], c[N];
        #pragma omp target device(acc0) in(a,b) out(c)
        for(i=0;i<N;i++){
                a[i]=sin(b[i])+cos(c[i]);
        }
}
```

Listing 2.2: *Offloaded code block example for OpenACC*

```
int main(){
        float a[N], b[N], c[N];
        #pragma acc data copy(a,b,c)
        #pragma acc kernels
        for(i=0;i<N;i++){
                a[i]=sin(b[i])+cos(c[i]);
        }
}
```

Another example of annotation-based programming models is OpenACC [43]. Very similar to OpenMP, it aims at being a programming model for easily implementing scientific code for heterogeneous HPC hardware. It also supports automatic parallelization of for loops (see Figure 2.2) although it requires compilers that are less extended than in the case of OpenMP. In the context of specific programming languages the REPARA annotation extension for C++ [44], provides C++ with annotations for parallel code generation for multiprocessors, FPGAs and GPUs. A last example of an annotation based programming model would be OMPSS and PyCOMPSS [45, 46], compatible with C, C++ and Python programs.

However, if a total control of the accelerator device is required, a library-based programming model is required. These are normally released by the accelerator vendor in order to provide the programmer with tools that help her to make use of the hardware. In the case of GPUs there are two main examples used in GPU programming for computing, NVidia CUDA and OpenCL. There are also additional programming models and libraries that are capable of taking advantage of the hardware and can be used for computing, but their primary objective is graphics computing, as we will see later with OpenGL.

The standard programming model for programming for GPUs is OpenCL [47]. Created as a way of developing portable code between different devices, it supports the execution of parallel code in desktop GPUs, mobile GPUs, standard multicore processors and FPGAs. Opposite to what the previously presented programming models did, OpenCL requires of specific programming skills and transformations of code. The code to be executed in the accelerator must be separated from the rest in an isolated function called kernel. This kernel is identified by compilers as accelerator code and compiled independently for the targeted architecture. Moreover, the code is implicitly parallel, executed by the available threads independently. The execution is controlled by thread specific variables with which they can be identified. In OpenCL, there are ids that can be obtained through different functions (Listing 2.3 and the keyword identifying the kernel is *__kernel*. Even though it is supported by several vendors and represents a standard, not all companies implement all its features, thus leading to compatibility problems.

NVidia CUDA is very similar to OpenCL, it needs a separate piece of code, a kernel, in this case identified with the keyword *__global__* (see Listing 2.4). The identifiers for the threads can also be retrieved through a structure and it also requires memory transfers between the device and the host. These memory transfers can be explicitly scheduled by the programmer, as seen in OpenCL and in this example, but from version 6.0 of CUDA it is possible to avoid the manual memory management. This is a great advance in terms of programming with respect to OpenCL. It is also a more modern programming model and it takes better advantage of the GPU due to its compatibility restriction to NVidia GPUs.

Therefore, OpenCL is mainly used when portability is needed or when no other generic programming model is available and NVidia CUDA is normally chosen if a NVidia GPU is guaranteed to be present, as it happens in many supercomputers. In terms of performance, NVidia CUDA applications are in average faster than their OpenCL counterparts, even reaching 30% more performance [48]. Nevertheless, these advantages in performance are due to the explicit optimizations that can be made in CUDA applications and that are not available in OpenCL because of portability issues.

To help with the programming difficulties, additional programming models were included as part of CUDA and OpenCL. CUDA Thrust [49] is a C++ template library that incorporates CUDA functionalities to C++. A similar approach but with OpenCL is Khronos SYCL [50]. Both of them aim at using stardard features already present in C++ in order to provide easy access to heterogeneous resources. A similar objective has GRPPI [51], a parallel pattern programming interface developed for C++ that offers an easy interface for parallel programming with the most common patterns, including *map*, *reduce* or *stencil*, normally present to data parallel architectures as well as streaming patterns as *farm* or *pipeline*. In contrast with previous solutions, GRPPI supports different parallel back-ends extending the compatibility to several architectures through different abstracted programming models.

Whatever the model chosen, with this library-driven approach a deeper knowledge of the programming model is required. However, the control provided by the large number of functions present in the API allows to more advanced programmers to gain full control of the hardware below.

There are numerous examples of the use of accelerators and diverse programming models in the medical imaging field in the literature. For instance, Scherl et al. [52] used up to 6 GPU families, and also FPGAs, to reduce the reconstruction time. They concluded that the main problem of GPUs resides in the memory bandwidth available to cope with data and the computing resources. Other examples of using GPUs in CT reconstruction are [53] and [54]. Both show that using CUDA and NVidia GPUs leads to very good performance results. However, they also demonstrated that the performance of the application strongly depends on the optimizations applied, which, in many cases, are not straight-forward. For these reasons new approaches have arisen that employ programming frameworks with a more friendly interface and that help to

Listing 2.3: *OpenCL Kernel.*

```
int main(){
float a[N], b[N], c[N];
....
kernel = clCreateKernel(program, "kernel_example", &err);
err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,N*sizeof(float), a, 0,
     NULL, NULL);
err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,N*sizeof(float), b,
    0, NULL, NULL);

....
err  = clSetKernelArg(kernel, 0,N*sizeof(float), &d_a);
err |= clSetKernelArg(kernel, 1,N*sizeof(float), &d_b);
err |= clSetKernelArg(kernel, 2, N*sizeof(float), &d_c);
err |= clSetKernelArg(kernel, 3, N*sizeof(float), &n)
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &
    localSize,
0, NULL, NULL);
kernel_example<<N,1>>(a,b,c);
clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,N*sizeof(float), c, 0, NULL,
    NULL);
....
}

__kernel void kernel_example( __global float *a, __global float *b,
    __global float *c ) {
int tid = get_global_id(0);
a[tid]=sin(b[tid])+cos(c[tid]);
}
```

Listing 2.4: *CUDA Kernel.*

```
int main(){
        float a[N], b[N], c[N];
        ....
        cudamemcpy(da, a, N*sizeof(float),  cudaMemcpyHostToDevice);
        cudamemcpy(db, b, N*sizeof(float),  cudaMemcpyHostToDevice);
        ....
        kernel_example<<N,1>>(a,b,c);
        cudamemcpy(dc, c, N*sizeof(float),  cudaMemcpyDeviceToHost);
        ....
}

__global__ void kernel_example( float *a, float *b, float *c ) {
        int tid = blockIdx.x;
        a[tid]=sin(b[tid])+cos(c[tid]);
}
```

automatize the process of programming for accelerators. Nowadays, many works are focused on using this type of frameworks. One of the advantages of their use is that, as [55] demonstrates, they allow easily porting the applications to heterogeneous environments. But, as [56] shows with a comparison of an implementation in OpenCL and OpenACC, there is a trade-off between the lower development effort needed for the OpenACC version and the performance obtained with OpenCL native version. In Section 2.5 we extend this state of the art to other algorithms and approaches.

2.3 Performance models for characterization and optimization

To be able to optimize and implement complex algorithms in different architectures it is necessary to understand how the algorithm is behaving and what problems must be overcome. For this purpose, different tools and methodologies have appeared to assist the programmers about their applications.

Monitoring and tracing tools have been developed in order to better characterize applications and algorithms executed over different hardware. One of the first tools that are used when characterizing scientific applications is *perf* [57], mainly used for obtaining information of the execution of an application at both kernel and user levels. This includes access to information provided by the hardware counters if support for the Performance API (PAPI) [58] is available. In the case of accelerators one of the most well-known tools is NVProf [59] created by NVidia to obtain information from the execution of kernels on their devices.

Using the information provided by these profilers, it is possible to characterize applications to assist the optimization process [60, 61] or to obtain the maximum performance provided by the hardware [62, 63]. These problems are addressed through the use of performance models that can provide an aid into understanding the limits of the hardware and the bottlenecks present in the characterized application. The main difficulty of these performance models is generality, since they are defined for specific architectures and normally can not be generalized, as device-specific models are more accurate than those applicable to different architectures [64]. This is the reason why currently multiple performance models for different devices are available in the last years. One of the most important performance models for shared memory architectures is the Roofline Model [65], which pretended to be a visual tool for detecting the performance roof of the microprocessor considering also the DRAM bandwidth. This model has been translated successfully to GPU accelerators, as shown by Jia et al. [66] for both NVidia and AMD GPUs. Another similar example is the Quadrant-Split visual performance model [67] used by Konstantinidis et al. to predict the performance of different kernels in NVidia GPUs. Moreover, roofline models with Dynamic Voltage and Frequency Scaling (DVFS) applied to GPUs [68] have been created.

An important feature for creating performance models is the knowledge of the architecture. In many cases some of the key characteristics of the hardware are not explicitly described by the vendor, thus, making the optimization of the applications difficult. Efforts have been done in this line of research for NVidia GPUs in different works [69, 70, 71] obtaining information about the different cache levels and their internal organization. This additional information can give an idea of the performance limits that can be reached by the application and explore the best option for efficiently applying the optimization effort.

2.4 HPC vs Big Data programming models

The increment in the volume of data to be managed by current systems has led to the emergence of new paradigms and frameworks able to tackle the problems related to these data-intensive applications. The solution was obtained in the form of the Big Data paradigm and its related frameworks, which appeared in the last decade. These frameworks and runtimes have been centered in the execution of data-analytics or loosely-coupled applications in large scale environments. Their easy deployment on virtual environments, such as the cloud, has also spread their popularity during the last years. To program this kind of applications the most famous programming model is MapReduce.

First appeared as a tool [72] created by Google to process their prodigious amount of data for their web crawler, open source implementations did not take long to arrive. Apache Hadoop [73] created a whole new ecosystem based on the MapReduce idea, from the execution engine, to the resource manager YARN [74], and the file system HDFS [75] (similar to what Google already had with their Google File System (GFS)[76]). However, due to the reduced functionality provided by the MapReduce programming model other frameworks extended the support to other type of operations. Between these novel frameworks the one that obtained more attention is Apache Spark [77]. Spark introduced compatibility with the previous Hadoop ecosystem with the possibility of in-memory computing through a feature named Resilient Distributed Dataset (RDD) [78]. RDDs are an abstraction of the data management level, providing containers and functionalities for maintaining the locality of the data and communication. These RDDs incremented the performance of the framework through avoiding the usage of persistent storage in temporal computations and storing them when possible in main memory. Other frameworks already realized that, for iterative computations, avoiding going further in the memory hierarchy was the key. In the work of Ekanayake et al. [79], a framework for scientific computing applications with support for Iterative MapReduce computations was presented. These approaches proved to be faster than previous Hadoop implementations, although its limitation to MapReduce programming model has turned the balance in favor of Spark even for scientific applications. Another generic framework for Big Data is COMP Superscalar (COMPSs)[80], which has similar objectives to those presented by Spark with a large number of functionalities that move it away from the simplicity of the MapReduce model.

On the other side, the HPC paradigm has aimed to adapt and adopt the MapReduce programming model in order to simplify programming and data management in scientific applications. Many of these attempts have translated the MapReduce model to the MPI library, the de-facto standard for distributed computing in clusters and supercomputers. Some of these solutions have extended the MPI API to include *map* and *reduce* functions that include implicit data management that is abstracted from the programmer as it is the case of MR-MPI [81], the work by Hoefler et al. [82], MRO MPI [83] or Mimir [84]. Others have opted for including similar data structures to those obtained in Big Data frameworks in communication patterns, such as the key-value objects in Data MPI [85]. At the end, what these works are looking for is a common programming model to those found in Big Data frameworks with the performance advantages of an HPC computing library, like MPI.

This is all part of a trend, initiated in the last years, of convergence between both paradigms, an idea exposed by several authors [86, 87, 88] and also explored in the context of this thesis. The increase in the data volumes generated and the need of larger scale systems is directing both worlds to the Exascale era.

New methods to combine Big Data and HPC paradigms have appeared. Apart from trying to integrate libraries such as MPI in Big Data frameworks another way of approaching them is to increase the support for heterogeneous hardware in Big Data frameworks. Original implementations of Hadoop and Apache Spark did not consider the possibility of offloading computation to accelerators or coprocessors present in the machines. Thus, their use in scientific clusters and supercomputers is limited by their compatibility with these type of architectures. For Hadoop, solutions appeared that supported computation in accelerators through conversion tools, such as Hadoop+Aparapi solutions [89, 90], which converted Java code to OpenCL in order to be executed in GPUs. These solutions are constructed to take advantage of accelerators in applications that previously did not support them transforming sequential code in the framework tasks to parallel GPU tasks. In case previous code was already implemented in either CUDA or OpenCL programming models these can not be executed inside the framework. Hadoop+ [91] gave this possibility to Hadoop through code plugins that could be written for CUDA and OpenCL compatible devices and a global GPU resource manager in charge of assigning accelerator resources to the different tasks. Recently, other works have focused on extending Apache Spark to support GPUs. An example is HeteroSpark [92], a heterogeneous CPU/GPU Spark platform for machine learning algorithms. IBM extended Apache Spark for enabling GPUs [93], the same approach as the one applied in [94]. Recently, Fukutomi et al. [95] added to Apache Yarn, a resource manager mainly used in MapReduce frameworks, support for GPUs, although targeting Java applications. Even though without native support for GPUs in these frameworks Boubela et al. [96] managed to combine Big Data approaches for the pre-processing of large-scale MRI data using Apache Spark with separate GPU servers for accelerating specific steps of the processing pipeline. This approach more independent than to modify the frameworks allows to use the original framework without any changes, an advantage in closed environments where software can not be configured as freely, such as in supercomputers or private clusters.

2.5 Systematic comparison of accelerated implementations

Reconstruction algorithms can also be classified in terms of the programming models employed and the optimization techniques used. In Table 2.2 we summarized different works related to medical image processing, and more concretely to CT. These works have employed different techniques to accelerate or improve the execution time of the algorithms implemented. Some of them included focus on the optimization of specific operations, such as the projection [97, 98], or complete implementations of the entire iterative reconstruction algorithm [99].

Most of the works are limited to single node execution although most of them have used either a parallel programming model or accelerator hardware to increase the performance. In the case of implementations not supporting accelerators, many choose to use OpenMP programming model [109, 124, 130, 132]. Some also include other type of parallelization, such as SIMD mechanisms [135, 109] or the support of multi-GPUs [124]. Even in some implementations, this non-hardware-accelerated solutions can match the performance of their GPU-based counterparts [135]. When employing GPUs the preferred choice seems to be NVidia GPUs over other types of GPUs, probably because they are easily found in HPC environments, as will be seen later. In terms of programming models, although OpenCL has also been used, and it has compatibility advantages over other choices, most works have chosen NVidia CUDA. The kernel parallelization is normally straightforward in most of the cases, although some works have fully exploited the hardware with specific optimizations. For example, Lu et al. [104] focused on the optimization of the projection kernel in a GPU taking into account the specific memory hierarchy of the hardware. Their optimizations lead to speedups of 1.47× over previously non-optimized GPU

Table 2.2: *Summary of the different works in Medical Image reconstruction related to CT reconstruction methods and details about their implementation.*

Authors	Reference	Architectures	Operators	Distributed Framework
Chen Jian-Lin et al.	[100]	Shared Memory (one thread); GPU CUDA	Projection,Backprojection	
Simon Rit et al.	[98]	Shared Memory (one thread)	Projection	
Zhao et al.	[101]	Shared Memory (one thread); GPU CUDA	Backprojection	
Nguyen et al.	[102]	GPU CUDA	Projection; Backprojection	
Du et al.	[97]	GPU CUDA	Projection	
Naik et al.	[103]	Shared Memory (Parallel Matlab); GPU CUDA		
Lu et al.	[104]	GPU CUDA	Backprojection	
Agaian et al.	[105]	Shared Memory (one thread);		
Grant et al.	[106]	Shared Memory (one thread);		
Solomon et al.	[107]	Shared Memory (one thread);		
Khawaja et al.	[108]	Shared Memory (one thread);	Iterative	
Hu et al.	[99]	Shared Memory (Multithread)	Iterative;Backprojection;Projection	
Wang et al.	[109]	Shared Memory (OpenMP; SIMD)	Iterative	
Xie et al.	[110]	GPU CUDA	Projection; Backprojection	
Sabne et al.	[111]	GPU CUDA	Iterative	
Bai et al.	[112]	GPU CUDA		
Meng et al.	[113]	Distributed	Backprojection	Hadoop
Palenstijn et al.	[114]	Distributed	Iterative	MPI
Rosen et al.	[115]	Distributed	Iterative	MPI
Boubela et al.	[96]	GPU CUDA		Spark
Cao et al.	[116]	GPU CUDA		Spark
Bao et al.	[117]	GPU CUDA		Hadoop
Yang et al.	[118]	Distributed	Iterative	Hadoop
Van et al.	[119]	Shared Memory	Platform	
MedPy	[120]	Shared Memory	Platform	
Hansen et al.	[121]	Shared Memory	Platform	
Gursoy et al.	[122]	Shared Memory; GPU CUDA	Iterative;Backprojection;Projection	MPI
Van et al.	[123]	Shared Memory; GPU CUDA	Iterative;Backprojection;Projection	
Blas et al.	[124]	GPU CUDA	Backprojection	
Blas et al.	[125]	GPU CUDA	Backprojection	MPI
Deng et al.	[126]	Distributed	Backprojection	MPI
Domanski et al.	[127]	Shared Memory; GPU CUDA	Backprojection	
Fan et al.	[128]	GPU CUDA	Iterative;Backprojection;Projection	
Koestler et al.	[129]	GPU OpenCL		
Melvin et al.	[130]	Shared Memory (OpenMP)	Iterative; Backprojection	
Tirado et al.	[131]	Distributed		MPI
Treibig et al.	[132]	Shared Memory (OpenMP)	Backprojection	
Kole et al.	[133]	GPU	Iterative	
Pratx et al.	[134]	GPU OpenGL	Iterative	
Sampson et al.	[135]	Shared Memory (Multithread; SIMD)	Projection;Backprojection	
Park et al.	[136]	GPU CUDA	Backprojection	
Mukherjee et al.	[137]	GPU CUDA; GPU OpenCL	Backprojection	
Siegl et al.	[138]	GPU CUDA;GPU OpenCL	Backprojection	
Mendl et al.	[139]	Shared Memory (Multithread);GPU CUDA; GPU DirectX	Backprojection	
Fang	[140]	GPU OpenGL	Iterative; Backprojection; Projection	
Zhu et al.	[141]	GPU CUDA	Backprojection	
Biguri et al.	[142]	GPU CUDA	Platform	

versions. Data staging between GPU and host is also a challenge that has been covered in this type of applications, with different strategies to avoid data transfer latencies [124] or designing scheduling strategies to overlap computation and memory transfers [141]. Memory management is also an important topic, resulting in a limitation in some works that do not support datasets larger than the memory capacity of the GPU [143].

Most of the works compared here address the problem of optimizing specific algorithms. However, there are solutions that have been designed with the idea of providing the user with a unified platform containing diverse algorithms and techniques related to medical imaging. These solutions allow us to simulate the acquisition and/or reconstruction of tomographic studies. However, they generally offer restricted possibilities for positioning the source and the detector, thus reducing their ability to simulate new acquisition protocols based on non-standard setups. For instance, CT Sim [144] is an open source CT simulator that enables the projection of various artificial models (phantoms), although it is limited to 2D circular scans for parallel-beam and fan-beam geometries without any kind of misalignments. It provides analytical reconstruction methods (FBP and Direct Fourier), without supporting iterative reconstruction algorithms. A more flexible alternative is IRT, an open-source image reconstruction toolbox [145], which provides a number of iterative algorithms, together with tools for building new ones. The main drawback of IRT is that it is only focused on standard cone-beam CT systems and does not provide enough flexibility for the more sophisticated scanning geometries achievable with radiology systems. TomoPy [122] provides projection, reconstruction methods, and pre-processing and post-processing tools, such as filters and artefact removal algorithms. However, the geometries offered are again rather simple, with the possibility of only changing the centre of rotation for projection and reconstruction.

Another common drawback to the above mentioned approaches is that they are all limited to CPU implementations. Given the high computational burden of some of the algorithms used in simulation and reconstruction and, as we have seen from isolated implementations of these algorithms, it is widely accepted that parallel implementations are needed to achieve reasonable execution times. X-ray Sim [146], which has a basic open-source version in the CPU, lacks flexibility in the available system geometries and is based on the projection of digital Computer Aided Design (CAD) models, thus hindering the direct use of real acquired images. A similar drawback is found in ImaSim [147], where objects are based on specific geometrical shapes and not voxels, thus precluding handling of voxelized objects such as actual CT datasets. CONRAD [148] is a Java-based framework that uses GPU devices for hardware acceleration. It provides tools for simulating 4D studies, analytical reconstructions, and artifact correction. Flexible scanning geometries are supported, although not in a straightforward manner, since they are based on a projection matrix that needs to be obtained beforehand. A popular open-source solution is the ASTRA toolkit [149] offers a solution based on CUDA that can be used to develop advanced reconstruction algorithms and allows the user to experiment with customized geometries. However, it is limited to datasets that fit completely in the memory space of the GPU and to circular orbits, thus precluding simulation of new acquisition geometries such as those used in tomosynthesis. Finally, TIGRE toolbox [142] contains implementations a wide range of algorithms in the Statistical Algebraic Reconstruction Technique (SART) family, the Krylov subspace family, and a range of methods using total variation regularization as a Matlab Package. Additionally, it has support for GPUs through CUDA, being possibly the most complete alternative to the work proposed in this thesis.

2.6 Thesis scope

Simulation and reconstruction tools for CT already exist in many frameworks and implementations. However all of them lack different capabilities that can make them a good solution for diverse situations. We have seen that most of them are focused on one type of architecture, supporting GPUs and multicore CPUs through a unique programming model lacking portability and leading to compatibility issues. When they do support different architectures they do not address problems such as the reconstruction of non-standard acquisition geometries or the easy construction of new simulation or reconstruction methods focusing on one specific algorithm. Additionally, the majority of these applications are designed for shared memory architectures, which is limited in terms of performance and memory. Distributed memory implementations in this area have not demonstrated high scalability and performance with platforms not optimized for distributed heterogeneous architectures.

In this thesis we will cover the development of new reconstruction and simulation techniques in different environments aiming at providing maximum portability of our algorithms and the maximum performance from the hardware chosen. For this purpose, we will cover in each chapter different programming models and paradigms that complete the gaps seen in this review of the literature. In Figure 2.7 we show all the topics covered in this thesis in the scope of Computer Science and Computer Architecture.

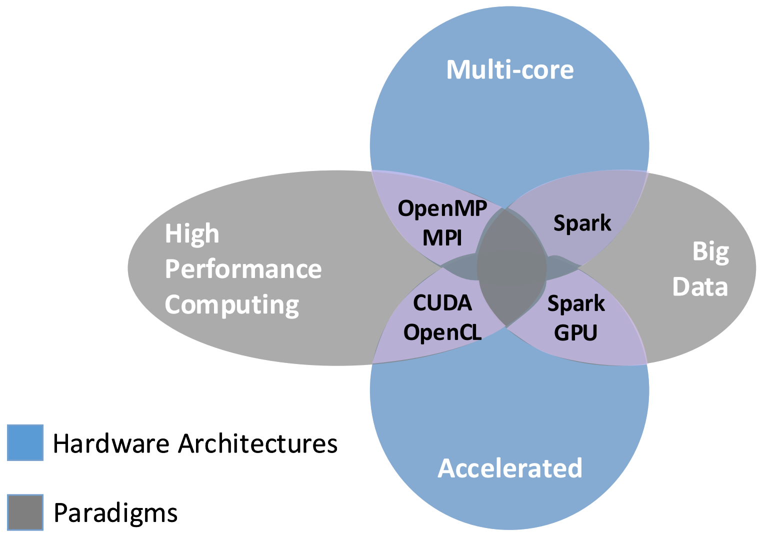


Figure 2.7: *Venn's Diagram representing the scope of this thesis. The rose colored areas are the ones addressed in this document.*

In terms of architectures we will cover both homogeneous and heterogeneous environments with a special focus on GPUs and more concretely on NVidia GPUs. Regarding the paradigms that will be introduced in different Chapters, we have to highlight HPC and Big Data paradigms. All these topics will not be covered in isolation but, as described by the figure, they will be combined in search of hybrid models that can best fit the medical image processing applications developed during this thesis.

Chapter 3

Proposal of a new fast and flexible X-Ray simulator

As we have seen in the review of the state of the art, previous works lack support for completely flexible geometries and/or compatibility with multiple programming models and platforms. To cope with this problem, this thesis proposes a new flexible X-Ray simulator and reconstructor, following the hypothesis that the development of new flexible X-ray systems can benefit from computer simulations, which may also enable performance to be checked before expensive real systems are implemented. The development of simulation/reconstruction algorithms in this context poses three main challenges. First, reconstruction algorithms for CT deal with large data volumes and are computationally expensive, thus leading to the need for hardware and software optimizations. Second, these optimizations are limited by the high flexibility required to explore new scanning geometries, including fully configurable positioning of source and detector elements. And third, the evolution of the various hardware setups increases the effort required for maintaining and adapting the implementations to current and future programming models.

This chapter is framed within the High Performance paradigm in heterogeneous and homogeneous platforms as shown in Figure 3.1.

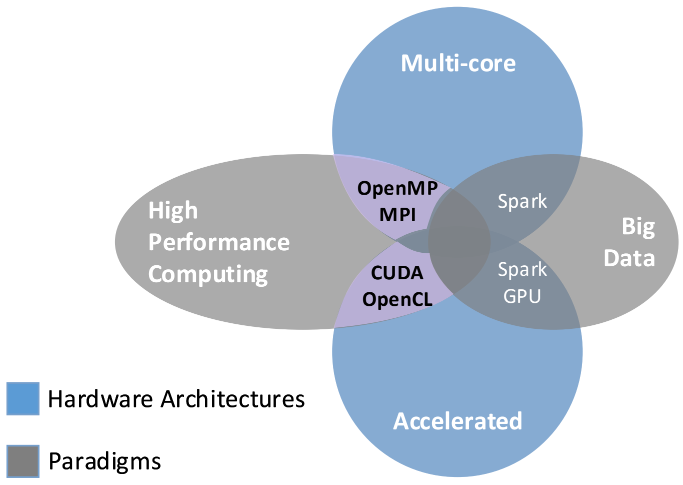


Figure 3.1: *Scope of Fux-Sim over the topics included in this thesis.*

3.1 Design goals

To cope with the end-user requirements, the three main goals to be fulfilled by the X-Ray simulator platform pursued in this thesis are:

- **Performance**: obtained through different parallel programming models that take advantage of the underlying hardware, including both multi-core CPUs and GPU accelerators.
- **Flexibility**: in terms of functionality and maintainability. It must be able to simulate and reconstruct with multiple geometries and flexible positioning parameters of source and detector. Since new reconstruction algorithms are being developed, the architecture must be flexible enough to allow the extension of the platform.
- **Compatibility**: with multiple current programming models and environments. It must support different architectures so it can be executed on different hardware platforms.

These three goals represent a trade-off that can not be fully solved. On the one hand, the achievement of full flexibility and compatibility affects performance. On the other hand, to obtain high performance, specialized implementations of the algorithms must be designed leading to an increased effort in order to support different hardware and acquisition geometries.

3.2 Framework architecture

The proposed X-Ray simulator, FUX-Sim, has been organized as a framework with a layered software architecture that provides support for different hardware and programming models in order to be modular and easily configurable, as shown in Figure 3.2.

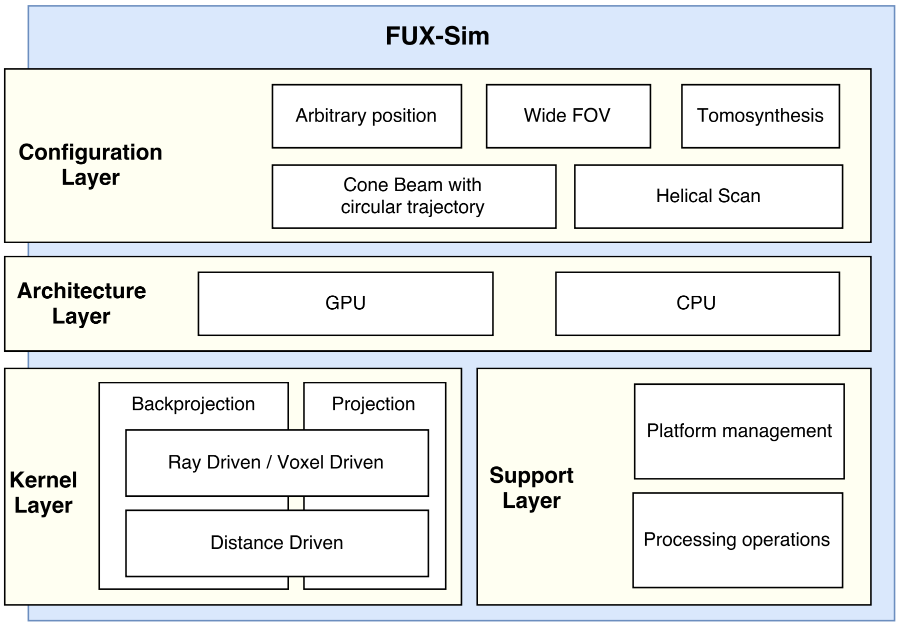


Figure 3.2: *Overview of the FUX-Sim platform and all its layers.*

The *configuration layer* allows the user to incorporate various system configurations including circular scan, arbitrary position, wide field of view, tomosynthesis, and helical scan. These configurations are mainly related to the geometries that can be defined in the platform, which can then be used for simulation or reconstruction. These geometries will define the Region Of Interest (ROI) of the result (specific areas of the output volume or projections) and the Field Of View (FOV) reached by the algorithm (visual range in which data can be obtained). Congurations can be customized through different parameters such as voxel and pixel size, detector and volume sizes, detector distances and movements, etc.

The *architecture layer* enables the execution of the simulator on different hardware architectures. For this purpose, all the algorithms are implemented in three programming models, namely, OpenMP, CUDA, and OpenCL, all of which are identical in terms of functionality and results. The *kernel layer* represents the execution core of the simulator and provides the main building blocks for the upper layers. At the same level, the *support layer* contains auxiliary processing operations (filters, matrix operations, etc.) and platform management modules to handle memory for different GPUs and CPUs. The *architecture layer* acts as a wrapper between optimized kernels and algorithms in lower layers. The execution of the simulator passes through the *architecture layer* to automatically reach the corresponding functionality in the *kernel layer* or *support layer*, depending on the availability of GPUs and the programming model chosen.

A detailed description of each layer can be found in the following sections.

3.2.1 Scanner configuration layer

The *scanner configuration layer* translates the parameters of the scanning geometry obtained from the command line or through the calibration file into a specific parameter set for the various system configurations.

Cone-beam with circular trajectory

The most standard configuration is a cone-beam system with the detector placed orthogonally to the line that passes through the source and the origin with the piercing point at its center, as shown in Figure 3.3 left, and with the source-detector pair following a circular trajectory with the object to be scanned at the origin of coordinates.

The implementation of this geometry is based on several calls to the projector/backprojector kernels for each view angle. The view angle is either calculated from the span angle and number of evenly spaced projections or read from the calibration file.

Helical scan

The helical configuration is implemented based on the circular cone-beam geometry described above, with the position of the volume for each projection changed to simulate the movement of the bed (Figure 3.3 right). For each angular position, θ, the shift of the voxels in the z direction is calculated by

$$shift_\theta = \frac{pitch \cdot span}{360 \cdot n} \cdot thick \tag{3.1}$$

where *pitch* is the displacement of the bed in one rotation, n is the number of projections per rotation, *span* is the total angle span covered during the acquisition, and *thick* is the slice thickness in the volume.

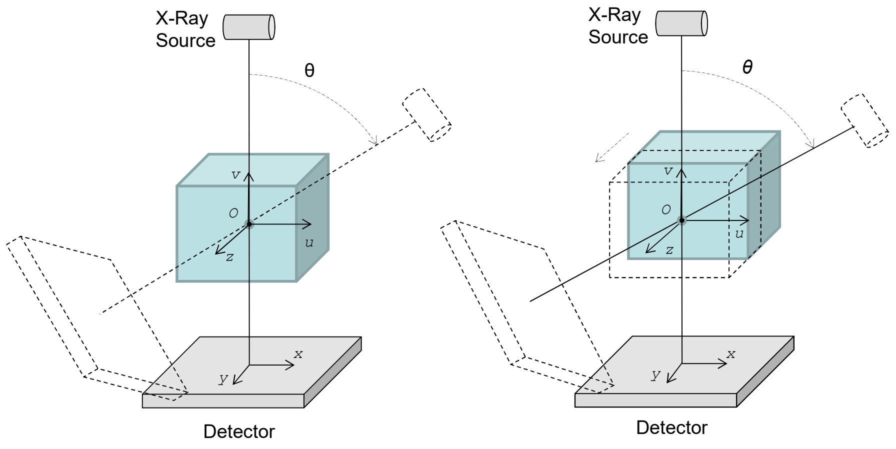


Figure 3.3: *Circular scan (left) and helical scan (right) configurations.*

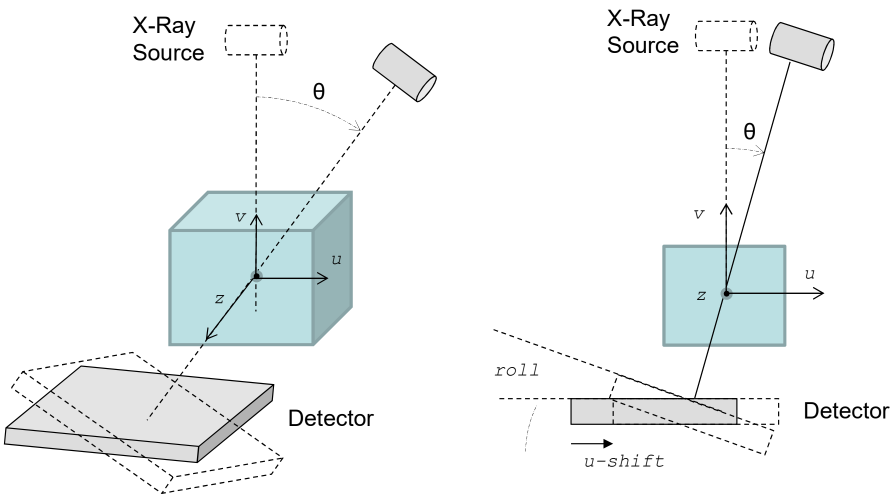


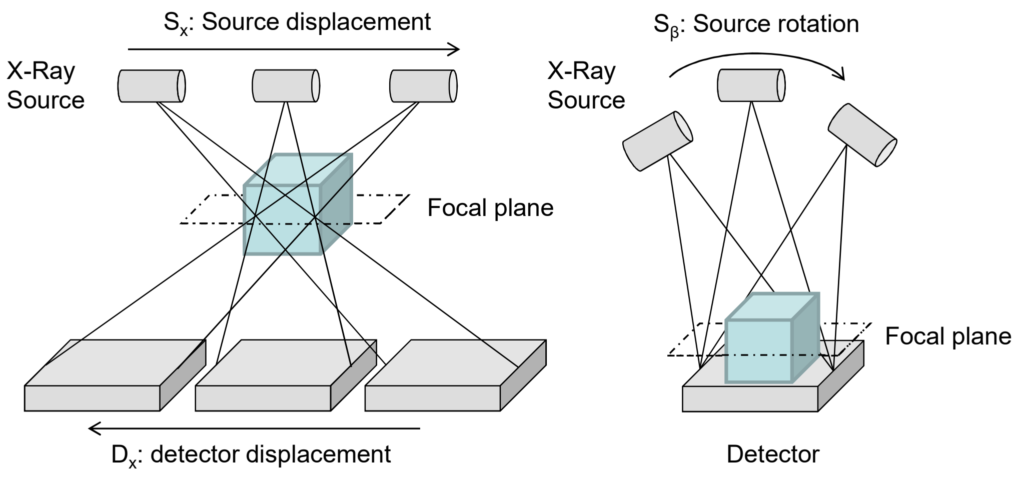
Figure 3.4: *Arbitrary positioning configuration.Translation of an arbitrary position of the detector into a set of geometrical non-idealities of a virtual detector. Dotted lines show source and detector in ideal intermediate positions.*

Arbitrary positioning

The arbitrary positioning configuration allows us to define an arbitrary trajectory for source and detector. Each position is translated into a set of linear displacements and angular inclinations from the ideal position (circular scan geometry), as shown in Figure 3.4. The translation is carried out in two steps: (1) u- and v-shifts are calculated so that the source-object line passes through the center of a virtual detector; and (2) inclinations (tilt and roll) are calculated as the angles formed between the real and virtual detectors around z and u axis, respectively.

Tomosynthesis

As shown in Figure 3.5, the simulator implements two system configurations for tomosynthesis: linear tomosynthesis, where the source follows a linear trajectory while the detector moves in the



S_x: Source displacement
X-Ray Source
Focal plane
D_x: detector displacement
S_β: Source rotation
X-Ray Source
Focal plane
Detector

Figure 3.5: *Linear (left) and Arc (right) tomosynthesis configurations.*

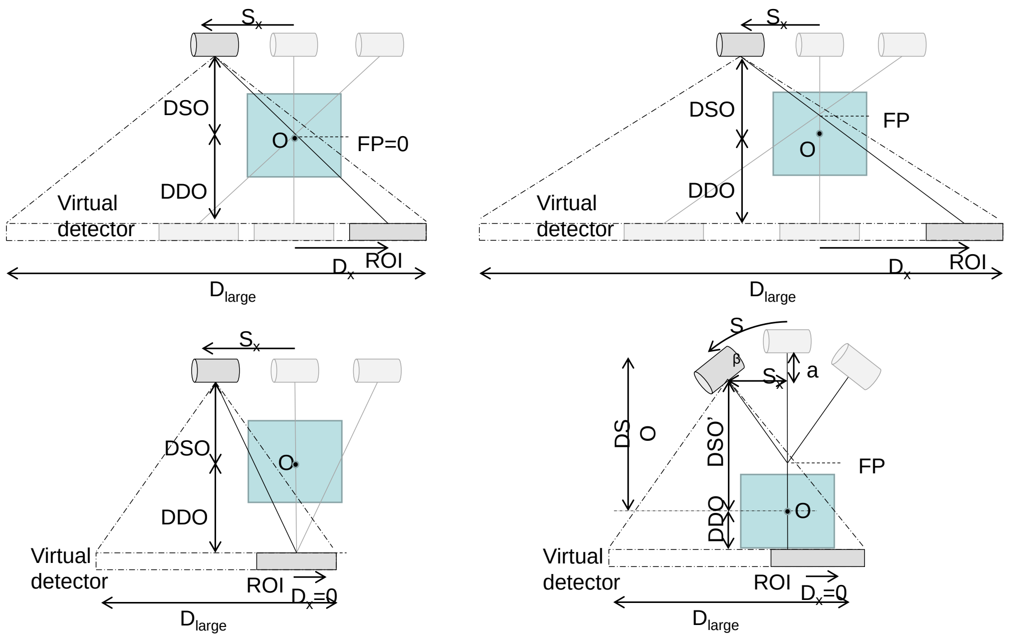


Figure 3.6: *Translation of tomosynthesis configurations into a set of geometrical non-idealities of a virtual detector that is larger than the real detector. A-C: Linear tomosynthesis with a different Focal Plane (FP). D: Arc tomosynthesis. DSO and DDO are the distance from the center of the FOV to the source and the detector, respectively, and ROI corresponds to the real detector.*

opposite direction (as in conventional tomography) and arc tomosynthesis, where the detector is static and the source follows a circular trajectory.

In both cases, the structures contained in the focal plane are projected into the same position of the detector, while structures in other planes appear at different locations in the projections.

The implementation of these configurations is based on the use of a virtual detector that is larger than the real detector, as shown in Figure 3.6.

In the case of linear tomosynthesis, the large detector size, D_{large}, is calculated as

$$D_{large} = 1 \times (S_x + D_x + \frac{D_{real}}{2}) = 2 \times S_x \times (1 + \frac{DDO + FP}{DSO - FP}) + D_{real} \tag{3.2}$$

where D_x and S_x are the displacements of the detector and source respectively, D_{real} is the actual detector size, DDO is the object-detector distance, and DSO is the source-object distance.

$$D_x = S_x \times \frac{DDO + FP}{DSO - FP} \tag{3.3}$$

For each projection, a ROI is calculated in the virtual detector equal to the real detector size centered at $D_x + S_x$.

For the case of Arc tomosynthesis, S_x and DSO are calculated for each projection as

$$sin(S_\beta) = \frac{S_x}{DSO - FP} \rightarrow S_x = sin(S_\beta) \cdot [DSO - FP] \tag{3.4}$$

$$cos(S_\beta) = \frac{DSO - FP - a}{DSO - FP} \rightarrow DSO - FP - a = cos(S_\beta) \cdot [DSO - FP] \tag{3.5}$$

$$DSO' = DSO - a = [cos(S_\beta) \cdot (DSO - FP)] + FP \tag{3.6}$$

$$D_{large} = D_{real} + (2 \cdot S_x) \tag{3.7}$$

where β is the angle rotated by the source.

Wide field of view

FUX-Sim enables the possibility of simulating an increased FOV, which is useful in scenarios where the detector is smaller than the scanning area. In these cases, two or more projections can be obtained and stitched together using a post-processing algorithm to build a larger image.

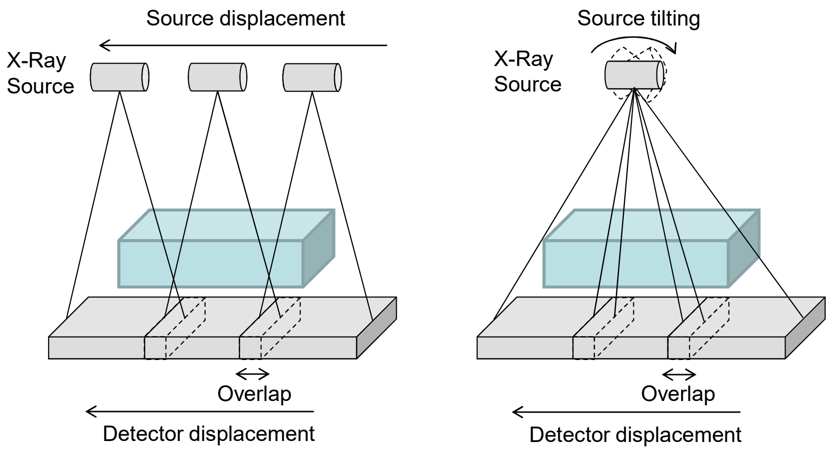


Figure 3.7: *Wide field of view configurations. Enhancement of FOV through linear displacement (left) and tilting (right) of the source.*

Depending on the movement of the source, FUX-Sim provides two models: linear displacement and tilting, as shown in Figure 3.7.

Linear displacement is based on the same idea as the helical scan: shift of the whole volume in the z-direction. The tilting configuration is based on defining a larger virtual detector, as in linear tomosynthesis. The large detector size, D_{large}, is calculated as

$$D_{large} = N \times (D_{real} - (N-1) \times Overlap) \tag{3.8}$$

where D_{real} is the detector size and N the total number of projections. For each projection at position n, we calculate a ROI on the virtual detector equal to the size of the real detector centered at D_x:

$$D_x = n \times D_{real} - (n-1) \times Overlap - \frac{D_{real}}{2} \tag{3.9}$$

where $Overlap$ is the overlap between two consecutive positions of the detector.

3.2.2 Architecture layer

The abstraction of the *architecture layer* makes it possible to create new configurations on multiple platforms (GPU, x86 CPU-based) and in different operating systems (Linux, Windows, and MacOS) without requiring a deep knowledge of accelerator architectures (see Figure 3.8). For this purpose, all algorithms and kernels were implemented according to three programming models: CUDA (for NVidia GPUs), OpenCL (for GPUs and ARM architectures), and OpenMP (for CPUs), thus enabling execution of the same algorithm in a parallel manner.

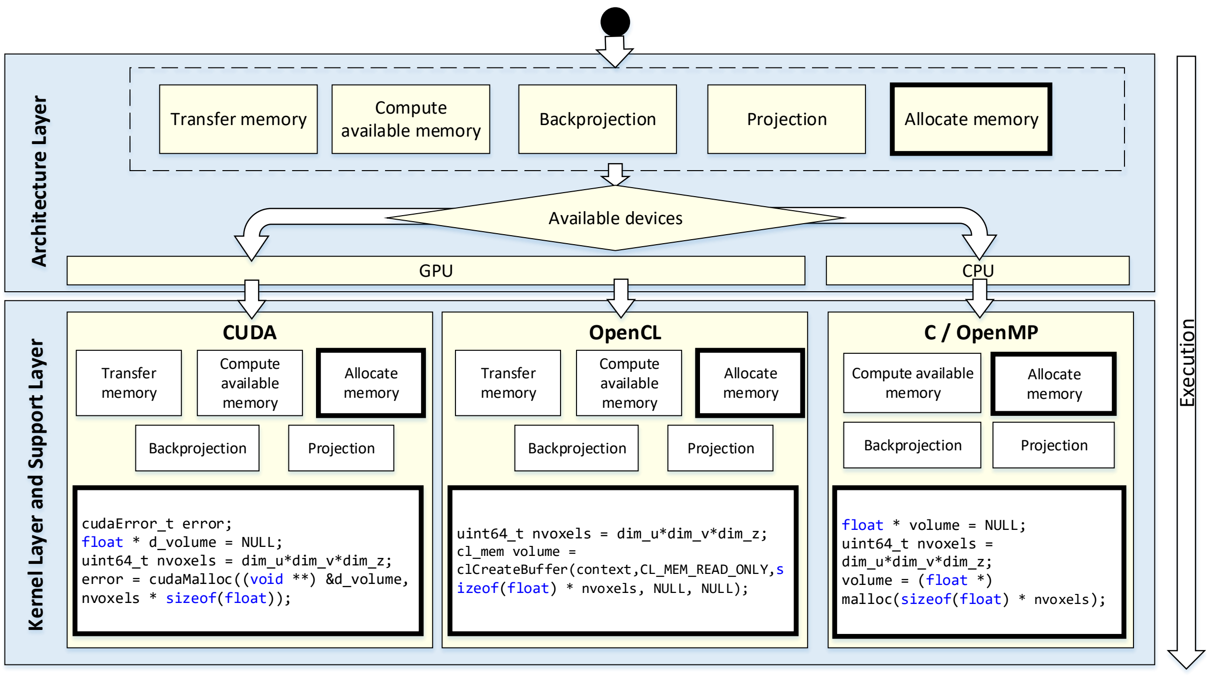
The *architecture layer* provides a wrapper for the specific version of the algorithms, which is configurable by the user depending on the available resources. The execution flow of the simulator passes through the *architecture layer* to automatically reach the corresponding functionality in the *kernel layer* or *support layer*, depending on the availability of the GPU and the programming model chosen. In the example shown in Figure 3.8, the allocate memory function in the *architecture layer* is translated into *cudamalloc*, *clcreatebuffer*, or *malloc* in the *kernel layer* and *support layers*, depending on the devices and the available programming models. Other important functions that depend on architecture are: memory transfers, obtain hardware characteristics or execute the accelerated portions of the code (kernels). All these cases follow the same path as with the allocation memory example being the upper-layers architecture-agnostic.

3.2.3 Kernel layer

The *kernel layer* constitutes the simulator core and contains the projection and backprojection computational kernels (see Figure 3.8, which are implemented based on a cone-beam geometry (see Figure 3.9).

It is possible to set all the system geometrical parameters (projection angle, source-object distance, detector-object distance, matrix and pixel size of the detector, matrix and voxel size of the volume), as well as the deviations from the ideal position of the detector (shifts, skew, roll, and tilt in Figure 3.9). The adjustment of these parameters enables the study of the effects of misalignments and the simulation of the scanner using non-regular geometries at arbitrary angular positions for other X-ray equipment such as a C-arm or tomosynthesis systems.

Linear shifts (x_{shift}, y_{shift}) and skew angle (ϕ) are applied with simple geometrical operations (shift or rotation of pixel coordinates):



```
cudaError_t error;
float * d_volume = NULL;
uint64_t nvoxels = dim_u*dim_v*dim_z;
error = cudaMalloc((void **) &d_volume,
nvoxels * sizeof(float));
```

```
uint64_t nvoxels = dim_u*dim_v*dim_z;
cl_mem volume =
clCreateBuffer(context,CL_MEM_READ_ONLY,s
izeof(float) * nvoxels, NULL, NULL);
```

```
float * volume = NULL;
uint64_t nvoxels =
dim_u*dim_v*dim_z;
volume = (float *)
malloc(sizeof(float) * nvoxels);
```

Figure 3.8: *Application's flow for the different programming models implemented in the platform.*

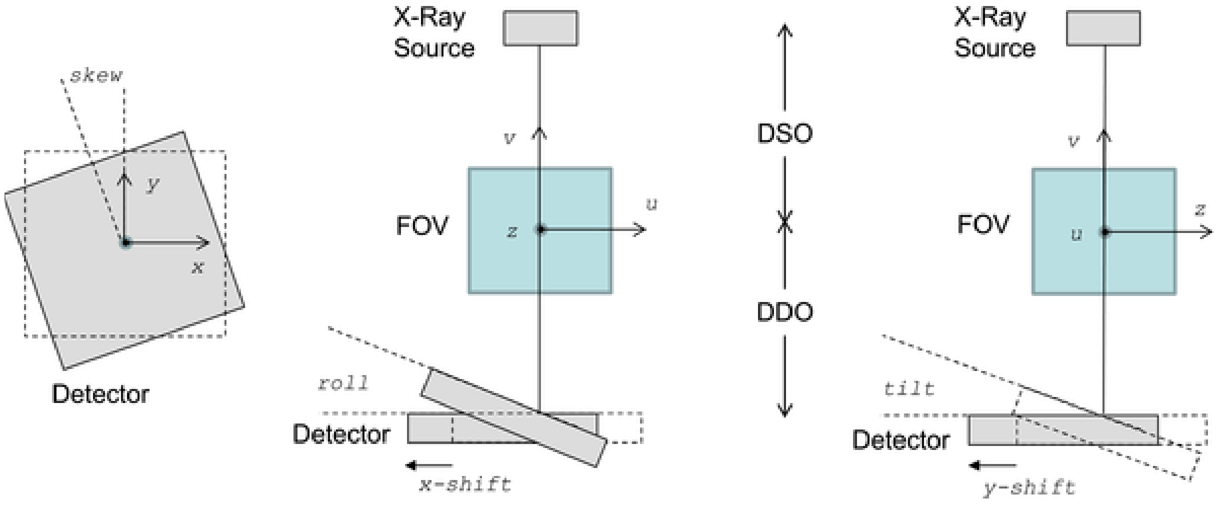


Figure 3.9: *Geometrical parameters used to parametrize deviations from the ideal position of the detector: Shifts, skew, roll, and tilt.*

$$\begin{pmatrix} x_{aux} \\ y_{aux} \end{pmatrix} = \begin{pmatrix} cos\phi - sin\phi \\ sin\phi cos\phi \end{pmatrix} \begin{pmatrix} x + x_{shift} \\ y + y_{shift} \end{pmatrix} \tag{3.10}$$

The effect of detector inclination (roll and tilt) is shown in Figure 3.10, where α is the inclination angle of the detector, A' is a pixel in the real detector, and A is the corresponding pixel in the ideal detector.

For each point in the ideal detector, we can calculate the corresponding point in the real

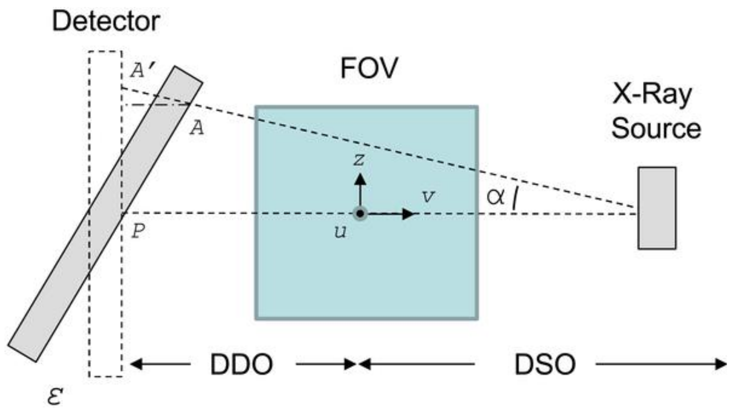


Figure 3.10: *The effect of detector inclination (roll and tilt).*

detector according to the expression:

$$|PA'|=\frac{|PA|}{cos(\epsilon)+sin(\epsilon)\cdot\frac{|PA|}{DSO+DDO}} \tag{3.11}$$

FUX-Sim implements ray-driven, voxel-driven, and distance-driven interpolation approaches. Ray-driven methods tend to introduce artifacts in the backprojection, whereas voxel-driven projection introduces grid artifacts into the projections [150]. With more accurate geometric modeling, distance-driven methods often lead to better image quality than ray-driven projection and voxel-driven back-projection [151]. This is achieved by projecting voxel and detector boundaries into the same axis and calculating the overlap between them (Figure 3.11), both for projection and for backprojection. Ray-driven and voxel-driven approaches rely on the computation of the trajectory corresponding to the center point of the voxel/pixel (black dot in Figure 3.11 for the case of voxel-driven backprojection), whereas distance-driven mode aims to obtain a more accurate representation of the contribution to the voxel/pixel by computing trajectories for its limits ($u1$ and $u2$ in Figure 3.11).

Given that the kernels are the most time-consuming components, this layer is where most of the optimizations were made, including the full parallelization of the ray trajectories. Two alternatives have been developed for projection and backprojection based on ray-/voxel-driven and distance-driven methods. Since each interpolation method needs a specific parallelization approach, two versions of each kernel were implemented in order to increase performance.

Backprojection

The backprojection kernel implements the integral along all the angles of the result of spreading back the projection values (sometimes after filtering or other pre-processing steps) along each ray, according to the following equation (if all the geometrical parameters are zero):

$$f(u,v,z)=\Delta\theta\cdot\sum_{\theta=ini}^{ini+nproj}p_\theta(Mag\cdot[ucos\theta-vsin\theta],Mag\cdot z) \tag{3.12}$$

where ini is the initial projection angle, $nproj$ is the total number of projections, $f(u,v,z)$ is the value in the back-projected volume at coordinates (u,v,z), $p\theta(x,y)$ the projection data for position (x,y) in the detector at angle θ, $\Delta\theta$ the step angle in radians, and Mag the magnification

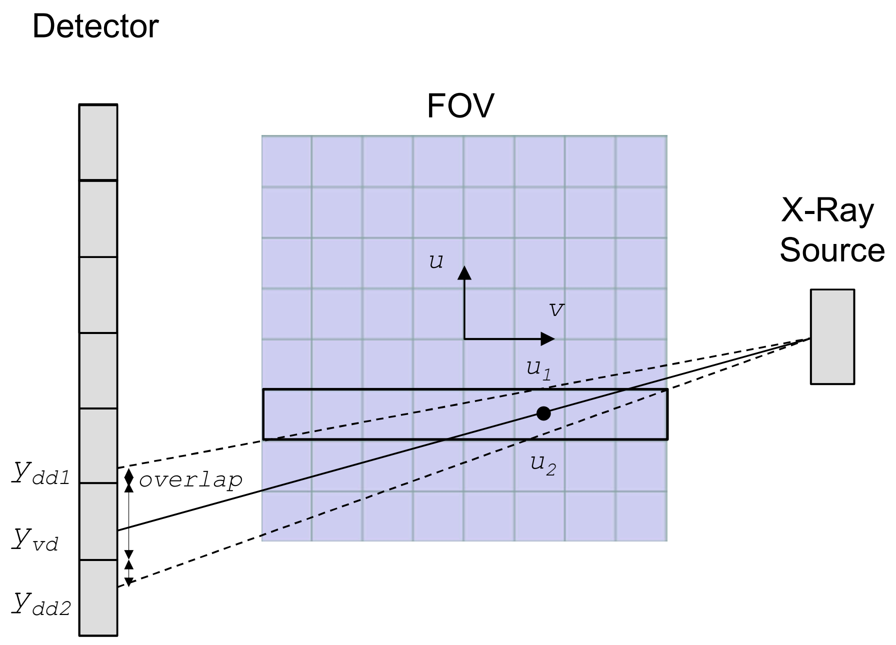


Figure 3.11: *Voxel-driven vs. distance-driven for backprojection. The contribution for the voxel (u,v) is calculated from the yvd value in the projection in the voxel-driven case and from ydd1 and ydd1 in the distance-driven case.*

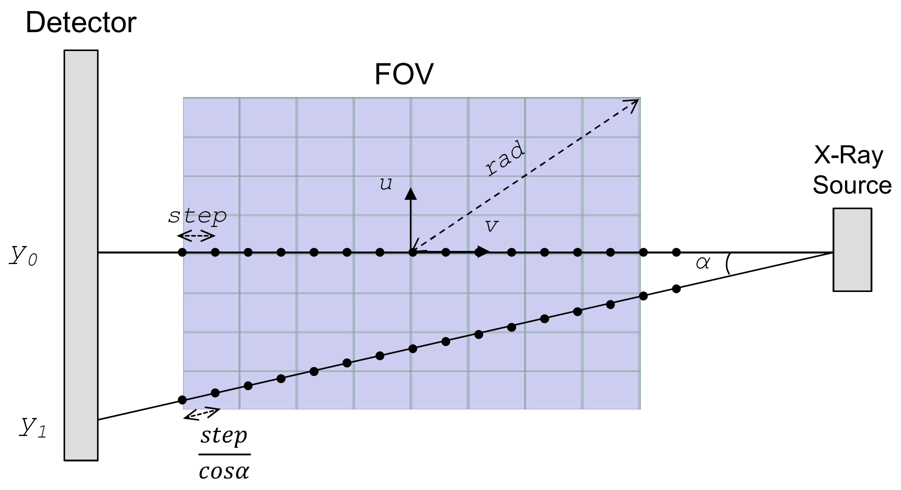


Figure 3.12: *Sampling scheme on the v-axis for the case of a non-isotropic voxel.* y_0 *corresponds to the central ray. Sampling points are indicated with dots.*

due to the cone shape of the beam given that

$$Mag = \frac{DSO + [usin\theta + vcos\theta]}{DSO + DDO} \tag{3.13}$$

where DSO and DDO are the distance from the center of the FOV to the source and the detector, respectively.

The implementation of the backprojection kernel is shown in Algorithm 1 for the ray-driven algorithm (orange color) and distance-driven algorithm (blue color).

Algorithm 1 Voxel-driven and distance-driven Backprojection.

```
1: procedure BACKPROJECTION(($projections, tilt, skew, shifts...$))
2:     for $u \leftarrow u\_vol\_size$ do
3:         for $z \leftarrow z\_vol\_size$ do
4:             Compute centered $u$ and $z$ coordinates in volume
5:             Compute centered u1,u2 and z1,z2 boundary coordinates
6:             for $\theta \leftarrow projections$ do
7:                 for $v \leftarrow v\_vol\_size$ do
8:                     Compute centered $v$ coordinates in volume
9:                     Compute real $SO$ and $DDO$ distances
10:                    Compute $u$ and $v$ rotated coordinates for $\theta$ angle
11:                    Compute u1,u2 and v rotated coordinates for $\theta$ angle
12:                    Compute magnification factor
13:                    Obtain ideal $x$ and $y$ coordinates
14:                    Obtain ideal x1,x2 and y1,y2 coordinates
15:                    if shift then
16:                        Apply x- and/or y-shift to $(x, y)$ coordinates
17:                        Apply x- and/or y-shift to (x1,y1) and (x2,y2) coordinates
18:                    end if
19:                    if tilt OR roll then
20:                        Apply tilt or roll to $(x, y)$ coordinates
21:                        Apply tilt or roll to (x1,y1) and (x2,y2) coordinates
22:                    end if
23:                    if skew then
24:                        Apply skew to $(x, y)$ coordinates
25:                        Apply skew to (x1,y1) and (x2,y2) coordinates
26:                    end if
27:                    for x_i > floor(x1) and x_i < ceil(x2):
28:                    Compute contribution for x_i
29:                    for y_i > float(y1) and y_i < ceil(y2):
30:                    Compute contribution for y_i
31:                    Update weighted value
32:                    $value \leftarrow Bilinearinterpolationofprojections(\theta, x, y)$
33:                    $volume(u, v, z) \leftarrow volume(u, v, z) + value$
34:                end for
35:            end for
36:        end for
37:    end for
38:    return $volume$
39: end procedure
```

Projection

The projection kernel emulates data acquisition in an X-ray system: the line integral is based on the computation of the sum of $N \cdot step$ values along the X-ray beam to update the contribution to the detector pixel:

$$p_\theta(x,y) = step \times \sum_{v_i=-\frac{rad}{step}}^{\frac{rad}{step}} \frac{1}{cos\alpha} \cdot f(\frac{1}{Mag}xcos\theta + vsim\theta, -\frac{1}{Mag}xsin\theta + vcos\theta, \frac{1}{Mag}y) \tag{3.14}$$

where rad is the maximum radius of the FOV (in mm), $f(u,v,z)$ is the voxel value in the sample at coordinates (u,v,z), $p\theta(x,y)$ is the projection data for position (x, y) in the detector at angle θ, α is the angle of the ray with respect to the central ray of the beam, and Mag is the magnification due to the cone angle, given by Equations 3.15 and 3.16.

$$x = arctg\frac{\sqrt{x^2+y^2}}{DSO+DDO} \tag{3.15}$$

$$Mag = \frac{DSO+v}{DSO+DDO} \tag{3.16}$$

where DSO and DDO are the distances from the center of the FOV to the source and the detector, respectively (see Figure 3.9). Sampling is performed along the v-axis given by $step$ (in mm), which is set by default to the minimum dimension of the pixel, covering $2 * rad$. The term $1/cos\alpha$ is included to compensate for the higher sampling in rays that are distant from the central ray, as shown in Figure 3.12 for the case of the ray that corresponds to y_1.

Algorithm 2 describes the projection kernel for both the ray-driven (orange color) and the distance-driven (blue color) algorithms.

Algorithm 2 Ray-driven and distance-driven projection.

```
1:  procedure PROJECTION()($volume, tilt, skew, shifts...$)
2:      for $\theta \leftarrow projections$ do
3:          for $x \leftarrow x\_proj\_size$ do
4:              for $y \leftarrow y\_proj\_size$ do
5:                  Compute centered x coordinate in projection
6:                  Compute centered y coordinate in projection
7:                  Compute centered x1 and x2 coordinate boundary in projection
8:                  Compute centered y1 and y2 coordinate boundary in projection
9:                  if skew then
10:                     Apply skew to $(x,y)$ coordinates
11:                     Apply skew to (x1,y1) and (x2,y2) coordinate
12:                 end if
13:                 if tilt OR roll then
14:                     Apply tilt or roll to $(x,y)$ coordinates
15:                     Apply tilt or roll to (x1,y1) and (x2,y2) coordinates
16:                 end if
17:                 if shift then
18:                     Apply x- and/or y-shift to $(x,y)$ coordinates
19:                     Apply x- and/or y-shift to (x1,y1) and (x2,y2) coordinates
20:                 end if
21:                 for $v \leftarrow v\_vol\_size$ do
```

```
                    Compute centered $v$ coordinate
                    Compute $(u, v)$ rotated coordinates for $\theta$ angle
                    Compute $(u1, v)$ and $(u2, v)$ rotated coordinates for $\theta$ angle
                    Compute real $SO$ and $DDO$ distances
                    Compute inverse magnification factor: $InvMag$
                    Obtain ideal $u$ coordinate: $InvMag \cdot u\_rot$
                    Obtain ideal $z$ coordinate: $InvMag \cdot z$
                    Obtain ideal x1 and x2 coordinate: $InvMag \cdot u\_rot1$ and $InvMag \cdot u\_rot2$
                    Obtain ideal y1 and y2 coordinate: $InvMag \cdot y1$ and $InvMag \cdot y2$
                    for x_i > floor(x1) and x_i < ceil(u2):
                    Compute contribution for x_i
                    for y_i > float(y1) and y_i < ceil(y2):
                    Compute contribution for y_i
                    Update weighted value
                    $value \leftarrow Trilinearinterpolationofvolume(u, v, z)$
                    $projection(\theta, x, y) \leftarrow value$
                end for
            end for
        end for
    end for
    return $projection$
end procedure
```

3.2.4 Support layer

The *support layer* contains two modules: processing operations (i.e., derivatives and filters) and the platform management.

Processing operations

The *support layer* provides basic processing operations for the customization of the simulation and auxiliary kernels needed for the reconstruction algorithms.

Customization includes functions for geometry computation and calculation of offsets for the definition of the ROI or Volume Of Interest (VOI). These functions are always executed in the CPU due to their low computational cost.

The *support layer* also includes auxiliary kernels responsible for matrix and element-wise operations such as arithmetic operations, derivatives, and computation of norms. Two important operations included here are the computation of the weighting factors W_1 and W_2, a necessary step for backprojection, and the application of a ramp filter to enhance high frequencies, which is an essential step in FDK-based methods and can be used to enhance high frequencies in other reconstruction methods.

Factors W_1 and W_2 are given by the following equations:

$$W_1 = \frac{DSO}{\sqrt{DSO^2 + x \times size\_x^2 + y \times size\_y^2}} \tag{3.17}$$

$$W_2 = (\frac{DSO}{DSO - v \times size\_v})^2 \tag{3.18}$$

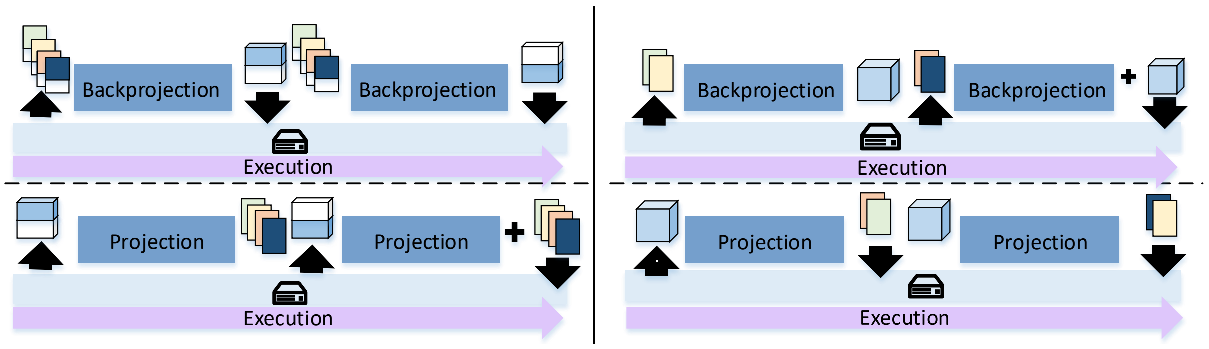


Figure 3.13: *Partitioning schemes implemented in Fux Sim. Partitioning by volume (left) for backprojection and projection, and partitioning projections (right) for both algorithms.*

where DSO is the distance from the source to the detector (in mm), x and y are coordinates in the projection, and v is the coordinate in the reconstructed volume (as shown in Figure 3.9), and $size\_x$, $size\_y$ and $size\_v$ are the pixel/voxel size in mm along x-, y- and v-axis, respectively.

The filtering operation involves Fourier transform and inverse Fourier transform steps, which are achieved by means of the cuFFT library [1] in CUDA and the clFFT library [2] in OpenCL. For the CPU, the filter is applied in the spatial domain through a convolution specifically implemented.

Platform management kernels

The platform management kernels are dedicated to operations such as memory allocation and deallocation in the GPU and the CPU, input/output operations, memory transfers between the GPU and host memory, and resource management.

The layout of the different structures of data is identical in both hots memory and GPU memory with the objective of facilitating the communication and transfers between both memory spaces. 3D volume structures are organized as a three dimensional matrix in row major order (first u coordinates, then v coordinates and finally z). Regarding projections, a set of 2 dimensional matrix was chosen in row major order (first x coordinates and then y coordinates).

Two partitioning strategies are proposed to address memory limitations in both the CPU and the GPU. The first consists in the division of the volume into multiple sub-volumes, called *chunks*, along the z-axis. The second consists in the division of the projections into *sets* (covering different angles). The decision on the number of projections included in one set fixes an upper threshold for the slot size, which is described in Section 3.3.2 (maximum number of projections transferred to the GPU).

These partitioning strategies, which can be combined, enable the execution of the kernel with partial volumes or projections in both the GPU and the CPU. They also provide the possibility of accelerating the execution using multiple GPUs, where each GPU is in charge of the backprojection of a chunk or projection of a projection set.

The chunk-partitioning strategy (Figure 3.13, left) is used for both projection and backprojection kernel executions. In the case of the backprojection kernel, each chunk is computed and stored to disk independently. In the case of the projection kernel, each chunk is read and com-

[1]https://developer.nvidia.com/cuFFT
[2]http://clmathlibraries.github.io/clFFT

puted for all the projection angles independently. The projections that result from each chunk are added and stored to disk.

The set-partitioning strategy (Figure 3.13, right) follows a similar logic. For the backprojection kernel, each set of projections is read and processed independently. The results are added in a final volume that is stored when all projections have been processed. In the case of the projection kernel, each set of projections is created from the volume and stored independently on disk.

The parameters chunk size and set size are calculated automatically by FUX-Sim at the beginning of the execution based on the hardware characteristics and current usage of the available resources, mainly memory capacity.

3.3 Parallelization techniques

The performance of the framework was optimized by applying different techniques, some of them dependent on the hardware platform, while others can be applied indistinctly to the GPU and the CPU.

3.3.1 Data interpolation

For the GPU version, FUX-Sim takes advantage of the texture memory in NVidia GPUs and in OpenCL-aware GPUs to reduce memory latencies and generate automatic bilinear or trilinear interpolations. The projections and volumes are uploaded to this memory space before the kernel execution.

For the CPU-based version, projection data are stored in the main memory, and accessed through an specific implementation of the bilinear or trilinear interpolation, which reduces the overall performance and consumes up to 25% of the total execution time.

3.3.2 GPU memory transfer pattern

The pattern for the memory transfers from the CPU to the GPU can dramatically affect execution time. Transferring larger datasets results in a more efficient exploitation of the bus capacity between the host and the GPU by taking advantage of the full memory bandwidth. Additionally, this approach enables simultaneous processing of various data and, therefore, a better exploitation of the available computational power of the GPU.

The memory transfers of projection data to the GPU device in the backprojection algorithm is one of the bottlenecks of kernel execution. Although the kernel is applied in each projection independently, if the GPU memory can hold one or more projections simultaneously, data are transferred in groups of projections. Projections belonging to each group are stored in the same array object (i.e., slot) concatenated vertically and separated with a padding zone, thus avoiding the use of values from the end of previous projections at the beginning of the current processed projection. The slot size is a configurable parameter selected by the user after taking into consideration the size of the projections and the underlying hardware. As demonstrated in a previous work [124], there is a trade-off between dataset size and performance for the case of the backprojection kernel. A huge dataset can be disadvantageous due to the overhead in kernel execution, since the number of projections present in the GPU affects the complexity of the kernel (line 6 of Algorithm 1).

In the case of the projection kernel, the subvolumes transferred to the GPU memory are formed directly by a group of contiguous axial slices (used for the 3D interpolation). In this case, the above mentioned trade-off does not hold: since the large number of axial slices does not affect the complexity of the GPU kernel (the kernel does not iterate over the z-axis), it does not imply an overhead in kernel execution.

After execution, output data (either projections or the volume) are transferred to the host memory for further processing or final storage.

3.3.3 Parallelization strategy

Parallelism represents the fundamental optimization implemented in the *kernel layer*. The strategy consists of dividing workload among different computational threads executed in parallel on either the CPU or the GPU. This work division differs depending on the interpolation method used. However, in both cases, parallelism exploits the data independence of the processing of each voxel or pixel, as described in [124].

To optimize memory access, the minimal computational thread in our parallel implementation is the iteration over the v-axis (black-delineated voxels in Figure 3.12 are computed by the same computational thread). Each of the parallel executions is identified by u and z in the case of the projection kernel, and by x and y in the case of the backprojection kernel (see first two loops in Algorithms 1 and 2, respectively).

The number of threads that can be scheduled is optimized by taking into account the number of required GPU registers. As the number of threads available for execution increases, the occupancy of the GPU is also increased, thus reducing the memory latency perceived [152]. The calculation of these rays trajectories for ray-driven and voxel-driven methods is shown in Algorithms 1 and 2, which are highlighted in italic font.

Parallelization of the distance-driven algorithm is highly limited by the intensive calculation of areas for each ray (shown in Figure 3.11). The computation of the boundaries, either on the volume or in the detector, adds four operations at each iteration. These boundaries are the limits of the voxels/pixels projected on each $u - z$ plane, as shown in Figure 3.12. Although independent, these boundaries have the same v-coordinate and access contiguous positions of the input data, thus increasing data locality when retrieving the values thanks to the memory layout. This loop is highlighted in bold font in Algorithms 1 and 2.

3.4 Evaluation

The performance of FUX-Sim was evaluated on two hardware architectures, namely, a high-performance workstation and a low-performance workstation. The high-performance workstation had an Intel(R) Xeon(R) E5-2630 processor with 32 cores at 2.4 GHz and 250 GB of RAM and an NVidia Tesla K40 with CUDA version 7.5 and OpenCL version 1.2. The low-end workstation was a commodity laptop equipped with an Intel Core i7 processor, 8 GB of DDR3 RAM, and a mobile GPU (NVidia GTX 965m).

Four studies have been used to evaluate our system: (1) standard-resolution; (2) high-resolution; (3) whole body versions of the Digimouse phantom[3]; and (4) a CT scan of the life-size human thorax phantom PBU-50 model (manufactured by Kyoto Kagatu), previously

[3]http://neuroimage.usc.edu/neuro/Digimouse

Table 3.1: *Detailed description of the studies used in the experimental evaluation.*

Study	**Standard resolution**	**High resolution**	**Whole body**	**Thorax study**
Detector pixel size (mm)	0.2*0.2	0.1*0.1	0.2*0.2	0.4*0.4
Detector matrix (pixels)	512*512	1024*1024	512*512	889*1080
Volume voxel size (mm)	0.1253	0.06253	0.1253	0.931*0.931*0.5
VOI (voxels)	512*512*512	1024*1024*1024	512*512*942	349*230*938
Chunk size	512*512*512	1024*1024*128	512*512*942	349*230*938
Set size	360/720	360/720	360	41
Circular Scan				
# projections	360/720	360/720	-	-
Tomosynthesis Scan				
# projections	-	-	-	41
Source displacement (mm)	-	-	-	150
Arc range (degrees)	-	-	-	10
Helical Scan				
# projections	-	-	360	-
Pitch	-	-	62	-

Table 3.2: *Processing times in seconds for CBCT configurations with backprojection and projection kernels for the different configurations and programming models evaluated.*

	Digimouse Standard Resolution (sec)						**Digimouse High Resolution (sec)**					
	Kernel execution			**Overall execution**			**Kernel execution**			**Overall execution**		
	CUDA	**OpenCL**	**CPU**	**CUDA**	**OpenCL**	**CPU**	**CUDA**	**OpenCL**	**CPU**	**CUDA**	**OpenCL**	**CPU**
Projection – circular scan												
360 proj	2.67	7.69	183.65	4.33	9.36	185.05	86.17	144.96	1456.07	92.39	150.92	1462.10
720 proj	5.12	15.19	377.03	7.95	18.05	379.71	170.52	269.08	2931.47	181.07	279.57	2942.57
Backprojection – circular scan												
360 proj	10.76	13.74	259.22	12.59	15.61	260.79	71.40	78.57	1995.03	83.82	90.90	2007.40
720 proj	21.32	26.67	630.14	23.43	28.90	632.17	136.55	149.19	4163.51	147.12	162.77	4177.13

acquired with a Toshiba Aquilion/LB CT scanner. Different configurations were simulated using ray-driven/voxel-driven interpolation modes with the parameters shown in Table 3.1. The last two rows show the configurable parameters set size and chunk size, which are calculated automatically during execution depending on data size.

On the one hand, Table 3.2 presents the results of the circular scans for both standard and high resolutions. On the other hand, Table 3.3 the results of helical, linear, and arc tomosynthesis. Both tables show the processing time in seconds for the kernel including memory transfers (kernel execution) and the process including I/O operations (overall execution).

The poorest performance was obtained with the CPU versions using OpenMP for parallelization of the core algorithms. Although OpenCL and CUDA used the same GPU and for high-resolution studies the performance was similar, OpenCL performed worse than CUDA for small volumes.

Table 3.3: *Processing times in seconds for the configurations and programming models evaluated for Digimouse study.*

	Digimouse Standard Resolution (sec)					
	Kernel execution			**Overall execution**		
	CUDA	**OpenCL**	**CPU**	**CUDA**	**OpenCL**	**CPU**
Helical scan	4.21	9.65	181.91	5.99	11.61	182.89
Linear tomosynthesis	3.18	4.34	19.57	11.32	12.61	26.07
Arc tomosynthesis	3.46	3.50	36.66	11.71	11.65	43.05

Table 3.4: *Comparison between Astra and Fux-Sim.*

	Astra	Fux-Sim
Support for flexible geometries	✓	✓
Support for tomographic images	✓	✓
Support for accelerated execution	CUDA in some algorithms	OpenMP; CUDA and OpenCL
Fully indepedent application	✗	✓
Bindings for other programming languages	Matlab, Python	Matlab
Support for scanner generated calibration files	✗	✓
Support for high resolution studies	✓	✓
Support for different interpolation modes	✗	✓
MultiGPU support	Only in one algorithm	✓
Support for configurable voxel and pixel sizes	Only in some kernels	✓

In the case of the circular scan, the total execution time of the projection kernel increased linearly with both number of projections (360 projections 2× faster than 720 projections) and resolution (standard resolution 32× faster than high resolution). Backprojection showed a different dependency on resolution, with the standard-resolution study only 8× faster than the high-resolution study. The reason for this is that the slot size is set to 1 to evaluate the most limited case; better results could be obtained by optimizing the slot size, as explained in [124].

Finally, the programming model that showed the best results, CUDA, was evaluated in the low-performance workstation. We employed the most demanding study, namely, backprojection of the high-resolution Digimouse with a circular trajectory. The configuration resulted in a total execution time of 376 seconds, which is 5× slower than the one obtained on a high-performance computer being limited by the size of the GPU memory. Chunk size and set size in this case are 1024×1024×286 (resulting in 4 chunks, the last one being slightly smaller) and 360 projections, respectively.

3.4.1 Comparison with Astra Toolbox

One well-known competitor in the field of flexible and accelerated simulation and reconstruction for X-Ray and CT is the Astra Toolbox [149]. Astra provides multiple implementations of different algorithms supporting CUDA-based acceleration for GPUs and it has bindings for Matlab and Python programming languages. In Table 3.4, we describe the main characteristics of Astra in comparison with FUX-Sim.

In terms of processing time, we have made an experimental comparison for the reconstruction of standard and high resolution studies, shown in Table 3.5. A comparison has been made between FUX-Sim with the usage of the kernels in CUDA and ASTRA 1.8[4] using the cone geometry and the reconstruction method BP3D_CUDA. In FUX-Sim we have fixed the slot size (number of projections processed by one call to the kernel) to 32. Astra employs a similar strategy with a group of projections of 32 as well. Results are favorable for Astra employing half of our solution's time. With Astra Toolbox, 3D reconstruction is not possible without support of CUDA GPUs. Additionally, the BP3D_CUDA method lacks of support for individual calibration for each projection, reducing the possibilities of reconstructing studies with misalignments or variable focus distance. This reduction in the capabilities of the reconstruction kernel impacts its complexity, thus explaining the bigger differences in time for the larger data sizes. In Table

[4]https://github.com/astra-toolbox/astra-toolbox

Table 3.5: *Processing times for backprojection in FUX-Sim and Astra Toolbox, both with CUDA support and the standard and high resolution studies.*

	Digimouse Standard resolution (sec)		Digimouse High Resolution (sec)	
	FUX-Sim CUDA	FDK_CUDA Astra	FUX-Sim CUDA	FDK_CUDA Astra
Backprojection				
360 projections	5.55	2.88	36.54	16.53
720 projections	7.72	4.56	49.27	23.67

Table 3.6: *Processing times for backprojection in FUX-Sim and Astra Toolbox, both with CUDA support and the standard and high resolution studies using one projection at a time.*

	Digimouse Standard resolution (sec)		Digimouse High Resolution (sec)	
	FUX-Sim CUDA	FDK_CUDA Astra	FUX-Sim CUDA	FDK_CUDA Astra
Backprojection				
360 projections	15.01	7.88	65.11	88.05
720 projections	27.57	13.75	105.82	126.75

3.7 we show the higher complexity in terms of floating point operations of FUX-Sim compared to Astra in case of the backprojection kernel. The number of instructions and floating point operations executed are 10× superior for FUX-Sim with a performance of 489.91 GFlop/s, 4× more than Astra Toolbox.

Table 3.7: *Performance metrics for both backprojection kernels.*

	FUX-Sim	Astra
GigaFlop	43.99	7.72
Instructions (millions)	33300	733
Memory Accesses (millions)	11000	3340
GigaFlops/s	489.91	135.73

Regarding the quality of the resulting backprojected image , we observe a significant difference between FUX-Sim and Astra in terms of Signal to Noise Ratio (SNR) metric. As seen in Table 3.8, FUX-Sim obtains 3× more SNR than Astra (the higher the better). For the Root Mean Square Error (RMSE), the difference still favours FUX-Sim although the variation between Astra and FUX-Sim results is not significant.

When studying the quality results slice by slice (see Figure 3.14), we observe that both metrics are stable, except on the initial and final slices. These slices are not completely covered by the cone beam geometry generating different artefacts that interfere with the obtained results.

Table 3.8: *Average RMSE and SNR for all 512 slices.*

	Avg. SNR (dB)	Avg. RMSE
True volume image vs FUX-Sim	3.33	0.13
True volume image vs Astra	1.20	0.17

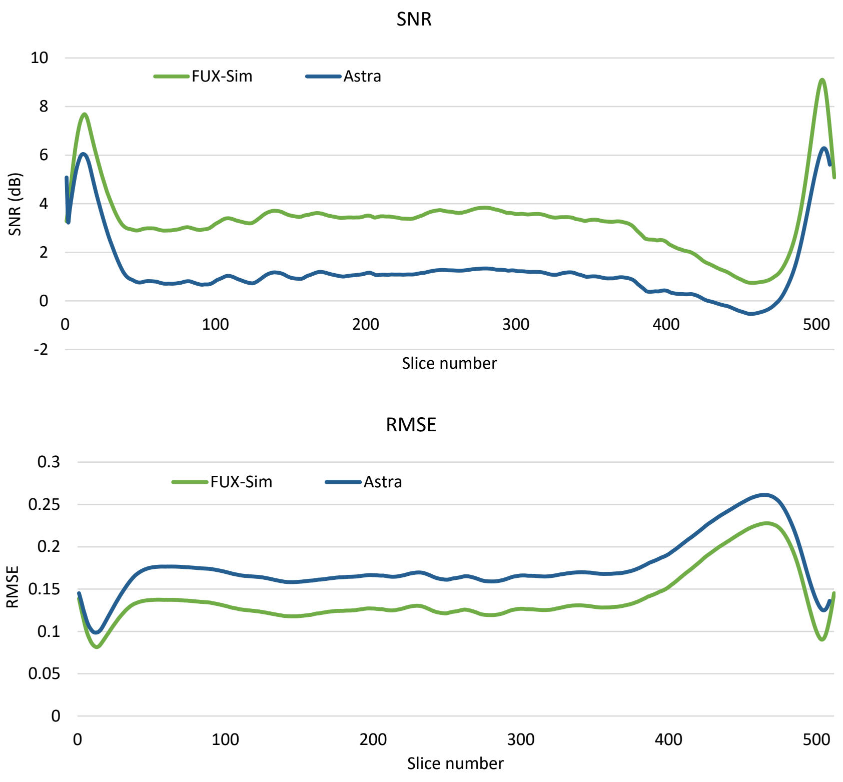


Figure 3.14: *SNR and RMSE of the results of the backprojection between Astra and the golden standard image. Results are shown for 512 slices of* 512^2 *pixels.*

3.5 Discussion

FUX-Sim was designed to address three key difficulties in the development of simulation/reconstruction algorithms: (1) the need to manage large data and computationally expensive volumes, thus needing hardware and software optimizations; (2) the limitation of optimizations by the high flexibility required to explore flexible scanning geometries, including fully configurable positioning of source and detector elements; and (3) the fast evolution of different hardware architectures, which increases the effort required to maintain and adapt implementations to current and future programming models.

Simulation and reconstruction algorithms require large memory capacity because of the need to allocate both projections and volumes in memory to ensure efficient computation. Memory limitations have been addressed by including two efficient partitioning strategies that allow the processing of small partitions of the input data. These strategies made it possible to run FUX-Sim on standard workstations with commodity hardware and low-memory/profile GPUs for both simulating or reconstructing large studies.

The optimized implementation for the different systems, i.e., programing models for the GPU (CUDA and OpenCL) and CPU, is achieved thanks to a modularized approach based on a layered architecture and parallel implementation of the algorithms in both GPU and CPU. The modular approach enables flexible and easy creation of new system configurations using existing kernels and utilities. This flexibility implies a trade-off with performance, as it prevents application of very specific optimizations. An example of this type of optimization would be the overlap of input/output operations and kernel execution, which would require tighter coupling between the *support layer* and the *kernel layer*, thus leading to a loss of modularity. Another example is the reduction in geometrical parameters used in the projection and backprojection kernels, such as detector shifts and rotations, in the case of simple geometries (e.g., ideal circular cone-beam scans). This simplification would imply the need for customized kernels for each geometry, thus hindering the creation of new system configurations. However, our evaluation showed that performance was similar to that of previous works thanks to the optimizations included in the different layers.

As expected, the worst performance was observed with the CPU version of FUX-Sim, even with the parallelization of the core algorithms using OpenMP. The GPU version of FUX-Sim was evaluated on both a laptop and a high-performance computer. The possibility of using a wide range of underlying hardware is an advantage over other simulation/reconstruction platforms presented in previous works, where, despite using the same acceleration device, execution with CUDA was 10% faster than with OpenCL when backprojecting high-resolution studies [139, 110]. However, we found a much larger difference in performance between CUDA and OpenCL when projecting smaller volumes: CUDA was 2$\times$ faster than OpenCL because hardware is used more efficiently with CUDA, which is compensated for when there is enough load to use the maximum computational capacity of the device.

Differences in hardware and software platforms make it difficult to compare execution times between studies. Nevertheless, an approximate comparison shows, for example, that FUX-Sim was around four times faster when projecting and five times faster when backprojecting than TIGRE [142]. We also obtained good results, even with our layered architecture, with respect to state-of-the-art implementations of the algorithms. Backprojection of similar volume sizes with FUX-Sim was more than two times faster than the CUDA and C implementation shown in [139]. Finally we showed that it was possible to simulate high-resolution studies in commodity computers, even when there is not enough memory to allocate the whole dataset.

The three configurable parameters that affect the overall performance of FUX-Sim are chunk size, set size, and slot size. Chunk size and set size are used for the optimization of memory transfers between the CPU and the GPU. Their value is automatically calculated based on the available resources of the computer (GPU global memory and CPU memory capacity). A low value for these parameters would increase the number of memory transfers and result in a low GPU utilization factor. The relationship between performance and slot size was studied in a previous work [124]. The value for this parameter is defined by the user after taking the texture memory capacity and GPU model into consideration.

The simulator can deal with a wide variety of scanning geometries, but does not include the source model (heel effect, polychromatic nature, focal spot) or detector model (noise model, intensity response), both of which could easily be included in the future as new modules of FUX-Sim in the *support layer*.

The architecture proposed in this thesis is significantly more flexible than that of previous simulators (CT Sim [144], IRT [145], TomoPy [122], X-ray Sim [146]), which do not allow the simulation of new acquisition protocols based on non-standard setups. The CONRAD [148] and ASTRA [149] toolkits allow flexible scanning geometries but present limitations. The simulation of non-standard geometries with CONRAD is less straightforward, as it is based on a projection matrix that needs to be previously obtained. We show the limitations of Astra in comparison with FUX-Sim in terms of functionalities, execution time, and resulting quality.

3.6 Summary

In conclusion, this chapter presented a new, highly flexible X-ray simulation/reconstruction framework that enables fully configurable positioning of source and detector elements. The implementation is optimized for two different families of GPUs (CUDA and OpenCL) and multi-core CPUs using a modularized approach based on a layered architecture and the parallel implementation of the algorithms in both devices. Consequently, FUX-Sim can be executed in most current hardware platforms, since OpenCL is supported by AMD and NVidia GPUs and by Intel and ARM processors, while CUDA is the most widely applied programming model for GPUs. The modular architecture also facilitates the maintenance and adaptation for current and future programming models. The execution times we measured were faster than other state-of-the-art implementations for different system configurations and hardware platforms. FUX-Sim can prove to be valuable for research on new configurations for X-ray systems with non-standard scanning orbits, new acquisition protocols, and advanced reconstruction algorithms. Moreover, the proposed framework will make it possible to obtain tomographic images from very few projections, thus enabling easy and inexpensive assessment before implementation in real systems.

The main contributions of this chapter are:

- FUX-Sim, a novel X-ray simulation/reconstruction framework that was designed to be flexible and fast.
- Optimized implementation of projection and backprojection algorithms for different families of GPU (CUDA and OpenCL) and multi-core CPUs.

The development of new advanced reconstruction algorithms based on iterative methods is an essential feature in any reconstruction platform. In subsequent chapters FUX-Sim is extended to support iterative reconstruction algorithms, but previously, further hardware specific optimizations are necessary. The main components of the simulator, as shown, are backprojection and projection and the remainder work will focus on their optimization to decrease their execution time. CUDA kernels of both components performed better in general and will be those employed in successive chapters of this thesis.

Part of this chapter has already been published in the work: *"FUX-Sim: Implementation of a fast universal simulation/reconstruction framework for X-ray systems"* [153].

Chapter 4

Enhanced cache-aware roofline model for GPU kernel characterization

The performance optimization and parallelization of scientific applications shape an important research topic for computer scientists. The impact of applying specific hardware optimizations in certain methods and algorithms can represent a large difference in terms of execution time.

A good characterization of the target application is necessary for optimizing or porting applications to new architectures. Profiling tools are available for numerous architectures and programming models, making it easier to spot possible bottlenecks in the source code. However, for a better interpretation of the performance and hardware metrics collected from these profilers, the usage of performance models have increased. One example of performance models are roofline models [65], which have been used for several years to assess performance of applications for different architectures considering hardware limitations. Different roofline models have been developed taking into account a limited set of characteristics such as input/output bandwidth, theoretical computational performance limits of the hardware, memory hierarchy, etc. These models are normally created based on two main metrics: performance (in Flops/s) and operational intensity (Flops/Byte). Additionally, more specific models focused on other parameters, such as energy efficiency have appeared [154]. Some of these models have been already included in well-known profiling tools such as Intel Advisor.

In this chapter, we present an extension of original GPU CARM that considers the influence of the texture memory of the GPU devices and specialized mathematical operations. Later, we demonstrate the application of the proposed GPU CARM for the performance characterization of CUDA-based applications. The work presented here has been carried out in collaboration with Professor Leonel Sousa and PhD. Aleksandar Ilic during a research stay in INESC-ID. We characterized two kernels that are part of an iterative reconstruction solution. These two kernels take most of the execution time of the whole method, being therefore suitable for a deeper analysis. We include the different configurations and parameters taken into account in order to decrease the execution time of each kernel.

4.1 Background of roofline models

The original roofline model [65] developed for traditional shared memory architectures was highly limited due to the simplification of the measurements taken to construct the model. Only Peak DRAM bandwidth was included and operational intensity was computed based on bytes only obtained from the traffic to and from the DRAM. Consequently, this model was limited and only insightful when studying one-level architectures in which caches were not used. However, the original roofline model gave a first insight into an easy differentiation between memory-bound and compute-bound regions which could be easily identified with this model.

An extended model, namely CARM [155], includes data from CPU caches, increasing the number of regions in which the application can be classified. This approach results in a better characterization of applications, which can be limited by other aspects different from the processor speed and the DRAM bandwidth. In Figure 4.1, we depict this extended model as presented by Intel Advisor tool. The different colored lines represent the different regions that can be generated from the model depending on the roof lines. It includes the peak bandwidth of all the caches present on the memory hierarchy as well as the different computational rooflines obtained from the computation of different operations on the processor.

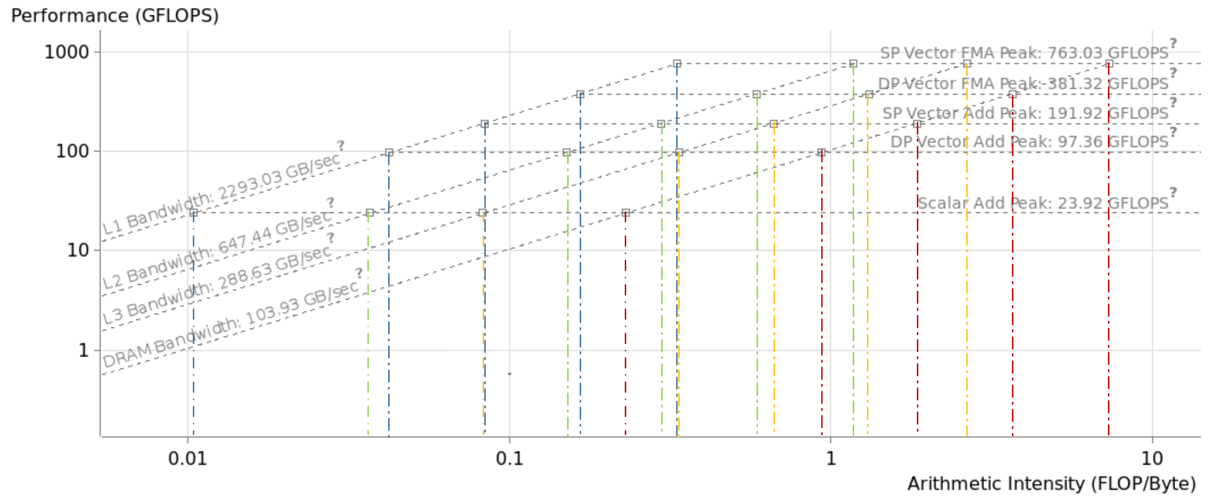


Figure 4.1: *Example of CARM as depicted by Intel Advisor. Bandwidth for the three levels of cache are shown as well as peak performance for Single Precission (SP) vector Fused Multiply Add (FMA), Double Precission (DP) Vector FMA, SP Vector Add, DP Vector Add and Scalar Add.*

Furthermore, this model can be transformed to characterize other traits of the application, such as energy efficiency [154]. This model was applied by Ilic et al. to NVidia GPUs creating the GPU CARM [71], which provided coherent results with respect other tools in terms of characterization. With this model it is possible to extend the characterization from the typical compute or memory bound result to a larger amount of intermediate results thanks to the inclusion of the different caches in the GPU devices.

4.2 Characterization and profiling method

In this section we will describe the methods employed in this study including the tools developed and the algorithms evaluated. Additionally an extension of the GPU CARM is presented

Table 4.1: *Main GPU hardware counters used for profiling and creating the GPU CARM.*

Floating point operations	$flop\_count_{sp}\_add$	Number of floating point add operations (single precision)
	$flop\_count_{sp}\_mul$	Number of floating point multiplication operation (single precission).
	$flop\_count_{sp}\_fma$	Number of floating point FMA operations (single precision).
	$flop\_count_{dp}\_add$	Number of floating point add operations (double precision).
	$flop\_count_{dp}\_mul$	Number of floating point multiplication operation (double precision).
	$flop\_count_{dp}\_fma$	Number of floating point FMA (Fused Multiply Add) operations (double precision).
	$flop\_count_{sp}\_special$	Number of floating point special operations. This category includes operations such as: fast division, cosines, sines or root squares.
Memory operations	inst_compute_ld_st	Number of instructions executed with load and store operations.

considering textures and special functions in GPU hardware. These rooflines were not previously taken into account in the previous model.

We will focus on the application of the CARM on GPUs [71]. The differences in the construction of the model for a CPU or for GPU mainly reside in the memory hierarchy used inside the GPUs. In GPUs the number of cache levels is reduced as well as the types of operations that can be performed. Therefore, the number of roofs obtained varies, although their meaning is identical, giving us the possibility of identifying the different bottlenecks in the explored kernels.

4.2.1 Profiling tool

We have developed an auxiliary tool for the automatization of the data model recollection and the generation of their corresponding GPU CARM visualization for their interpretation.

Performance information is obtained through hardware counters present in different NVidia architectures by using NVProf (NVidia profiler) tool. The counters are mainly chosen for their relation with the CARM in terms of performance (operations) or memory accesses (memory instructions). Detailed information of each counter and their meaning can be found in Table 4.1.

Thanks to these counters we can obtain the two main factors that influence the CARM: arithmetic intensity and performance. To compute the performance of the kernel we collected the total number of operations (either in double or single precision) using the following equations:

$$fp_{sp} = flop\_count_{sp}\_add + flop\_count_{sp}\_mul + 2 * (flop\_count_{sp}\_fma) + flop\_count_{sp}\_special \quad (4.1)$$

$$fp_{dp} = flop\_count_{dp}\_add + flop\_count_{dp}\_mul + 2 * (flop\_count_{dp}\_fma) \quad (4.2)$$

where FMA instructions ($flop\_count_{sp}\_fma$) are multiplied by two since it represents the execution of two operations: multiplication and addition in one step. We also obtain the execution time of the kernel and the number of both load and store instructions (inst_compute_ld_st) with the objective of calculating the performance in Equation 4.3 and the Arithmetic Intensity (AI) in Equation 4.4. For double precision operations equations are similar.

$$perf_{sp} = \frac{fp_{sp}}{ex\_time} \quad (4.3)$$

$$ai_{sp} = \frac{fp_{sp}}{inst\_compute\_ld\_st * 4} \tag{4.4}$$

Empirical roofs for the memory hierarchies of the GPUs employed in this chapter were already provided by Lopes et al. [71]. In the next sections we contribute with new benchmarks for the obtaining of new roofs. Additional information is gathered from these roofs, certainly when the kernels to be studied make use of these characteristics. Considering the type of operations present in medical image processing kernels we have focused on two characteristics: the usage of textures and the computation of trigonometric functions.

Extending GPU CARM: the texture cache roof

Another main difference between CPUs and GPUs in terms of architecture details is the presence of a texture pipeline that provides concrete functionalities related to the memory management (i.e., data access, inherit computation, etc). In some GPU architectures, there is also a separate texture cache in charge of caching and pre-fetching the context of the texture memory. To obtain the maximum bandwidth given by the texture cache, we have designed a benchmark kernel in charge of getting the maximum performance of the cache. This kernel (see Listing 4.1) takes advantage of the acquisition of consecutive elements in the 1D texture. Thus, no strides are introduced and caches should be used at its maximum capacity. To avoid interferences in the experiment, we have avoided the usage of interpolation functions by invoking the function *tex1Dfetch* instead of *tex1D*. The use of *volatile* variables has as objective to avoid possible compiler optimizations. This is also the objective of storing the 0 value in a variable instead of using it as a constant.

Listing 4.1: *Benchmark for obtaining max texture bandwidth*

```
template <class T>
__global__ void benchmark_tex_standard_1d(){
 const int glob_id =  blockIdx.x *blockDim.x +threadIdx.x;
 volatile T r0;
 int r1 = glob_id;
 __shared__ T shared[SHSIZE];
 const float r2 = 0;
 for(int j = 0; j<16384;j+=8){
  #pragma unroll
  for(int i=0; i<8;i++){
   r1 = r1 + r2;
   r0 = tex1Dfetch(tproj1d,r1);
  }
 }
 shared [threadIdx.x]= r0;
}
```

The maximum bandwidth obtained for the texture cache in an NVidia Tesla K40 is 1,153.4 GB/s. This experimental bandwidth was validated with the NVidia profile tool, using both the Texture Unit utilization level (HIGH) and the Texture Cache Hit Rate (100.00%) metrics.

Extending the GPU CARM: special trigonometric function roof

In medical imaging kernels, where different geometric operations must be performed in order to calculate target coordinates, built-in trigonometric functions are fundamentals. CUDA offers implementations for the most common functions, such as *sin*, *cos*, *tan*, etc. These functions can be executed in a fast way, using an approximation of the functions that reduced the precision of the operations, or with an exact version. In this section we review the maximum performance of the fast version (using `__<name_func>f` or including the *–fastmath* compilation flag), commonly employed to accelerate the execution of the kernels. In this case, the benchmark is based on the execution of the cosine function (see Listing 4.2).

Listing 4.2: *Benchmark to obtain maximum performance of cosf function.*

```
template <class T>
__global__ void benchmark_cosf(T in){
        __shared__ T shared[SHSIZE];
        T r0 = shared[threadIdx.x];
        T r1 = r0;
        T r2 = r0;
        T r3 = r0;
        for(int j = 0; j<1024;j+=8){
                #pragma unroll
                for(int i = 0; i<8; i++){
                        r0 = __cosf(r1);
                        r1 = __cosf(r2);
                        r2 = __cosf(r3);
                        r3 = __cosf(r0);
                }
        }
        shared[threadIdx.x] = r0;
}
```

The maximum performance obtained in a NVidia Telsa K40 is 256.063 GFlop/s, which is about 12 $\times$ less performance that the one obtained with singles precision FMA operations [71] (empirical maximum performance of the card).

4.2.2 Algorithms

In the previous chapter, we introduced and examined the FUX-Sim [153] framework in terms of execution time and performance. When analyzing the different components of FUX-Sim, the layer consuming a largest percentage of the total time was the Kernel layer. This layer contained the implementation of both backprojection and projection components. Both of them are present in all possible setups of the framework and take part in advanced iterative reconstruction algorithms. Thus, any possible optimization that can be applied, effectively reduces the overall execution time of many functionalities of FUX-Sim.

For both kernels in standard baseline versions, output data are stored in global memory while volume and projection data for input are stored in a CUDA array, bound to 3D or 2D textures. This texture-based implementation has proven to be much more efficient than global memory due to the caching and spatial arrangement of textures in memory and the automatic interpolation that is provided by the texture units [156, 157]. Geometric data are stored in constant memory, thus providing fast access to all threads.

Backprojection

An outline of the operations included in the backprojection kernel is shown in Pseudocode 1 (Chapter 3 Section 3.2.3). From the two types of interpolation modes that are available in the framework, we will focus on the ray-driven method. This interpolation is commonly used and represents a higher priority. The computational complexity of this kernel is therefore defined as $O(u\_vol\_size \times v\_vol\_size \times z\_vol\_size \times projections)$. The operations carried out in this kernel can be divided in three regions:

- Rotation of the volume and magnification. The rotated coordinates of the volume in axis u and v are computed for the selected angle. Magnification in z is also computed. In these operations several computations for equivalences between real measurements (mm) and voxel and pixel sizes are calculated.

- Computation of misalignments with respect to the ideal geometry. These operations are executed for every obtained projection coordinates. Control instructions are dominant in this region of the algorithm.

- Memory accesses and computation of the final value. The value on the selected coordinates is obtained from memory with its corresponding bilinear interpolation. Depending on the configuration of the kernel, this bilinear interpolation can be executed using hardware or software functions.

Furthermore, in all these regions special mathematical functions are also included (*sin*, *cos*, *tan*, *sqrt*, etc). These functions consist of several floating point operations that can be optimized by the compiler at compilation time. Parting from this initial pseudocode, different configurations of the kernel were created taking into account optimizations over the three regions of the code.

The configurations designed for this kernel are the following:

- **std** (Standard baseline): this configuration represents the initial implementation of the algorithm without any modification or profiling input. The algorithm followed is unchanged with respect to the one presented in the pseudocode. This standard version does not establish a maximum number of registers and is compiled with the –use_fast_math flag.

- **opt** (Branch optimized): this configuration modifies the computation of the geometrical distortion computation (tilt and roll). The main modification consists on reducing the computation of the misaligments, pre-computing the sign of the final coordinates before the actual branch appears. Therefore, we reduce the number of branches by two and generate only one computation independently of the direction of the tilt parameter.

- **mreg32** (Maximum number of registers 32) : with this configuration we limit the number of registers per thread to 32 using the compilation flag *–maxrregcount=32*. The number of register per thread depends on the number of variables present in the system and it is limited. A high number of registers per thread can decrease the occupancy of the device and the performance.

- **nofm** (Disabling fast-math): a version that eliminates the use of *fast-math* operations (special operations for division, cos, sin...). This specially affects to the computation of misalignments and rotated indexes.
- **gm** (Change texture memory fetching by global memory): in this version the projection data is stored in standard global memory and the interpolation functions are implemented as ___device___ functions.

Projection

The pseudocode for the algorithm implemented in the kernel is shown in Pseudocode 2 (Chapter 3 Section 3.2.3). Thread parallelization in the GPU is based on x and y coordinates of the resultant projection, executing a kernel for each angle. This approach effectively reduces the memory requirements being capable of only storing one projection at a time, a difference with the backprojection kernel.

Similarly to the backprojection, the projection can also be divided in three regions:

- Rotation of the volume and inverse magnification. The rotated coordinates of the volume in axis u and v are computed for the selected angle. The inverse magnification in z is also computed. In these operations, multiple computations for equivalences between real measurements (mm) and voxel and pixel sizes are calculated.
- Computation of misalignments with respect to the ideal geometry. These misalignments are computed over the projection coordinates, which are the initial coordinates of the GPU threads. Therefore, these additional computations that in the backprojection are computed for every coordinate in the depth axis v, in projection are now computed once for all v coordinates. It also contains a large quantity of control instructions.
- Memory access and computation of the final value. The final value is retrieved from memory with the computed coordinates of the volume. At this point a trilinear interpolation function is employed for non ideal coordinates.

The configurations implemented for this kernel are the following:

- **std** (Standard baseline): this configuration represents the initial implementation of the algorithm without any modification or profiling input. The algorithm followed is unchanged with respect to the one presented in Pseudocode 2. This standard version does not establish a maximum number of registers and is complied with the –use_fast_math flag.
- **opt** (Branch optimized): this configuration modifies the computation of the geometrical misalignments computation (tilt and roll). The main modification consists on reducing the computation of the misaligments, pre-computing the sign of the final coordinates before the actual branch appears. Therefore, we reduce the number of branches by two and generate only one computation independently of the sign. This configuration effectively reduces the number of control instructions executed by the kernel.
- **reg** (Manually unroll v loop to increase register usage): the v loop is unrolled with a factor of 2 in order to increase the register usage and the texture locality. This optimization also implies an increase in the number of registers employed in the kernel.

- **mreg32** (Maximum number of registers 32) : with this configuration we limit the number of registers per thread to 32 with the compilation flag *–maxrregcount=32*. In this kernel, standard versions do not surpass the maximum number of registers available per block, however, this configuration can be useful combined with others, such as, reg, which increase the number of registers per thread and can reduce the effective occupancy.

- **nofm** (Disabling fast-math): a version that does not use fast-math operations (special operations for division, cos, sin...). This affects specially to the computation of misalignments and rotated indexes.

- **gm** (Change texture memory fetching by global memory): in this version the volume data is stored in standard global memory and the interpolation functions are implemented as ___device___ functions.

4.3 Experimental results

We have evaluated both kernels with the different configurations explained in the previous sections. We employed two different families of GPUs, whose specifications are described in Table 4.2, with different number of cores and clock speed. In all cases we use CUDA 8.0 and NVidia Profiler version 8.0 to compile and obtain the main counters of the hardware. All experiments were executed 10 times and here we show the median execution time. We also provide the results for the CPU version of the kernels, also characterized through CARM.

We evaluated all configurations by combining 2D block configurations from 8 to 128 threads per dimension. We executed the algorithms over two studies datasets with different sizes: 512^2 projection size and 512^3 volume size; and 1024^2 projection size and 1024^3 volume size. In the case of the GTX 980, only the first image was obtained due to memory constraints.

Hardware counters are collected to later characterize the kernel in terms of performance (GFlops/s) and Arithmetic Intensity (Flops/Byte). For this purpose, we gather multiple metrics, all described in Table 4.1, in order to obtain the number of floating point operations executed in both simple and double precision inside the kernel. We finally construct the model based on this information.

Table 4.2: *Specifications of the GPUs employed in the evaluation.*

Specification	**GTX 980**	**Tesla K40c**
Architecture	Maxwell (GM204)	Kepler (GK110B)
Clock Speed (GHz)	1.126	0.745
Cores	2,048	2,880
Theoretical Performance (TFlops)	4.5	4.29
SMs	16	15
Global memory (GBytes)	4	12
Internal memory bandwidth (GB/s)	224	288
L2 Cache Size (KB)	2,048	1,536
Texture units	128	240
Texture cache	✗	✓
Compute Capability	5.2	3.5

4.3.1 High-end GPU evaluation

We started with a thorough evaluation on a NVidia Tesla K40. In Figure 4.2, we show the execution times for several configurations for projection and backprojection kernels compared to the *std* configuration time (in orange) for two data sizes. For each configuration the execution time from the block size combination with the best performance was taken. For the backprojection kernel, the configuration that gave best results was the *opt* version. The reduction of the computation on the internal branches of the algorithm helps to reduce the time spent in this part of the algorithm that, since it is inside the internal loop of the kernel, is executed v times per thread. For the projection kernel the same behaviour is not expected. The branches for the computation of the geometrical distortions are not inside the internal kernel, thus reducing its impact over the overall execution time. In this case, the best configuration is the *reg* configuration. This configuration increases the number of registers used inside the kernel, reducing the effect of register spilling and increasing the overall performance.

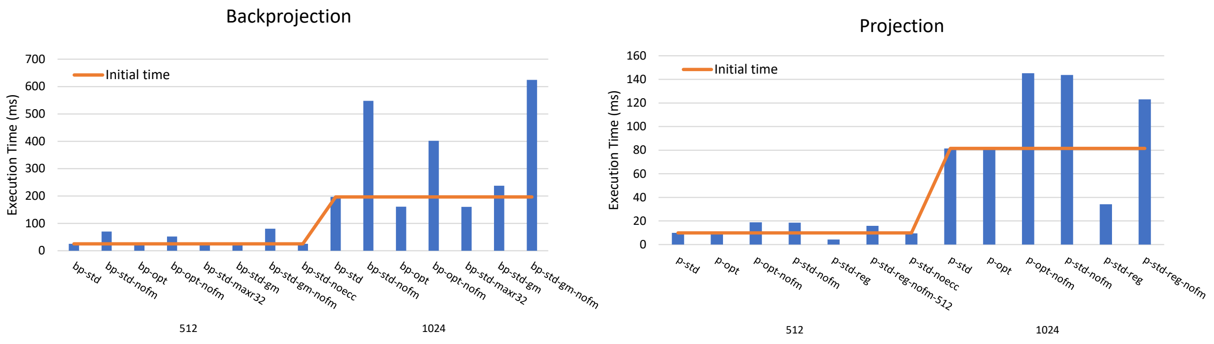


Figure 4.2: *Execution time for different configurations with their best block sizes. Results are shown for the backprojection (left) and the projection (right)*

After applying the equations described in Section 4.2, we obtained the characterization of the application inside the CARM of the Tesla K40. The models obtained are shown in Figure 4.3. Both of them are obtained with the data acquired from the configurations executed with the block sizes leading to the best performance. None of the kernels are in the compute-bound region, although *p-std-reg* is near. The execution of the kernels with different size do not reflect significantly in the model being obtaining in some case only slight better performance when executing with the larger data set.

In Figure 4.4 we plot the same models for the block size leading to the worst performance. The difference with respect to the best performance case is minimal. The position of the clusters of configurations do not change since the arithmetic intensity must be identical independently of the block size chosen (total floating point operations do not change independently of how the blocks are configured).

We compare the obtained results with the GPU CARM with the characterizations obtained from the NVidia Profiler. In Table 4.3, we summarize the different characterizations obtained for both kernels in some of the configurations, experimenting with the best and worst block dimensions. We found discrepancies between both tools with differences between compute-bound and memory-bound characterizations when the block dimension is altered.

NVidia classifies kernels taking into account the time spent in different types of instructions (compute instructions vs load/store instructions). However, CARMs does not take into account this mix of instructions, focusing on the performance of the whole kernel execution as well as

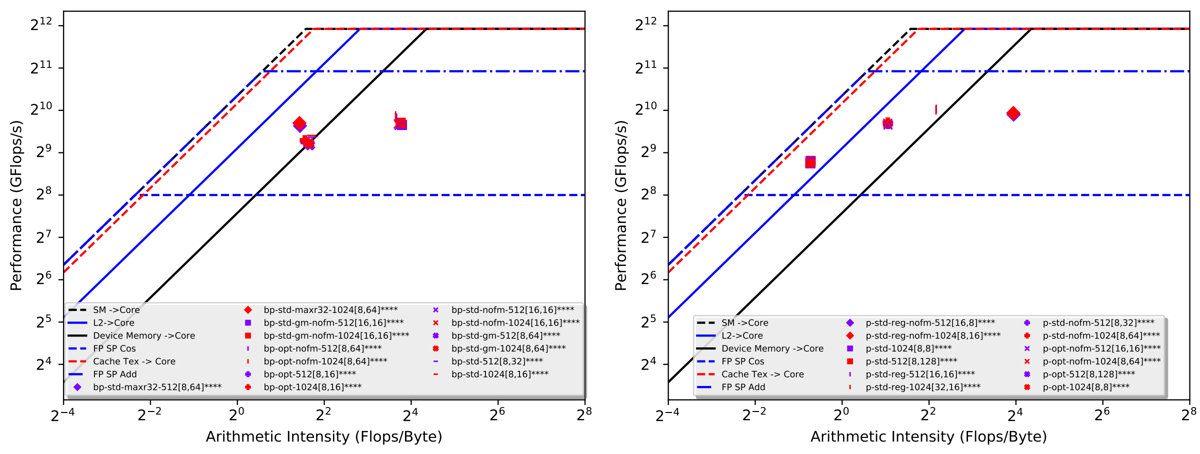


Figure 4.3: *CARM for NVidia Tesla K40 GPU for all configurations with the thread block dimensions that lead to best performance in backprojection (left), and projection (right).*

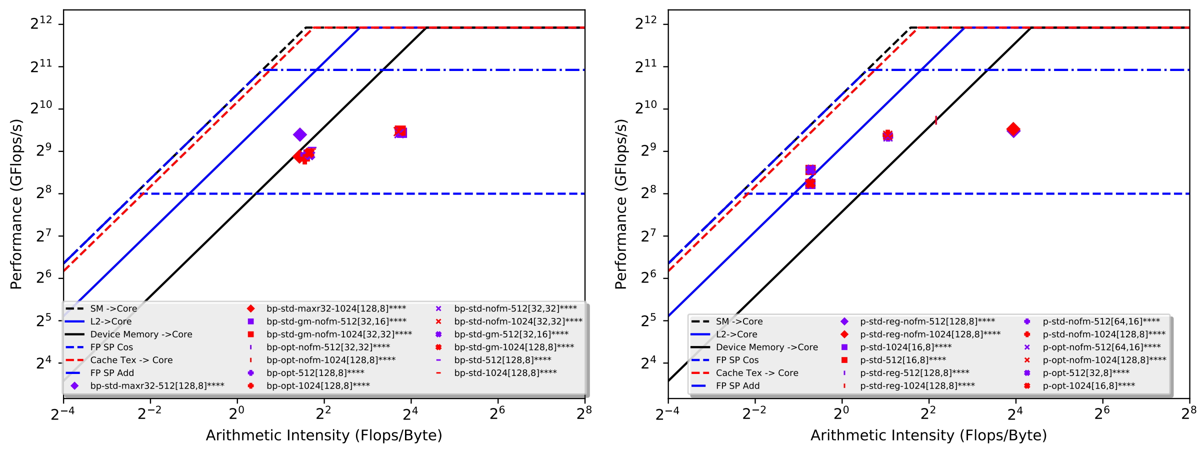


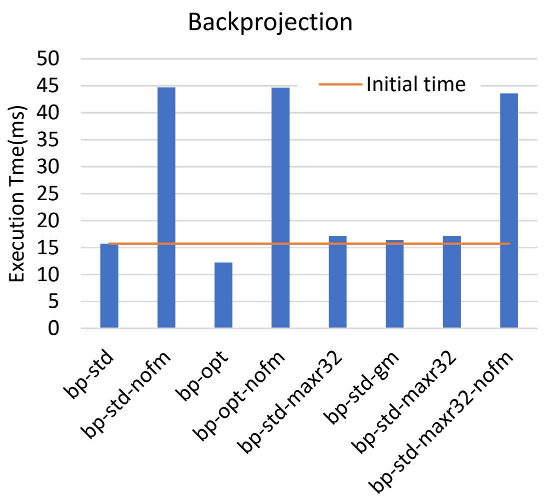
Figure 4.4: *CARM for NVidia Tesla K40 GPU for all configurations with the thread block dimensions that lead to worst performance in backprojection (left), and projection (right).*

the arithmetic intensity. Therefore, when employing a GPU CARM only, performance can be modified if the block dimensions are changed, causing a vertical movement in the plot. For NVidia, this also affects the time spent on memory operations, which can affect the characterization of the kernel. Therefore, for a full characterization of the kernel, additional features and information must be used in order to completely understand the behaviour of the application.

Additionally, roofs are computed based on the maximum performance that can be reached by the GPU, thus, with the maximum occupancy and level of parallelism that is possible inside the device. Therefore, for kernels in which possible trade-offs exists (occupancy vs number of registers/shared memory size) these roofs do not necessarily represent the real performance limits of the application and can change the characterization of the kernels. This is the reason why NVidia is capable of characterizing the kernels differently from the CARM. This affects the characterization of the kernels with different block sizes and configurations since these characteristics change the occupancy ratio on the GPU.

Table 4.3: *Table with the characterizations obtained for different configurations of the kernels in the Tesla K40c with the GPU CARM and with the NVidia profiler.*

Name	Characterization Best block size	Characterization Worst block size	Characterization CARM
bp-std	compute bound	instruction and memory latency	memory bandwidth
bp-opt	compute bound	instruction and memory latency	memory bandwidth
bp-std-nofm	compute bound	compute bound	compute bound
bp-opt-nofm	compute bound	compute bound	compute bound
bp-std-mreg32	compute bound	instruction and memory latency	compute bound
bp-std-gm	compute bound	instruction and memory latency	compute bound
p-std	memory bandwidth	memory bandwidth	memory bandwidth (L2)
p-opt	memory bandwidth	memory bandwidth	memory bandwidth (L2)
p-opt-nofm	instruction and memory latency	instruction and memory latency	memory bandwidth
p-std-nofm	instruction and memory latency	instruction and memory latency	memory bandwidth
p-std-gm	compute bound	instruction and memory latency	memory bandwidth
p-std-gm-nofm	compute bound	instruction and memory latency	memory bandwidth
p-reg	instruction and memory latency	instruction and memory latency	memory bandwidth
p-reg-nofm	compute bound	Instruction and Memory Latency	memory bandwidth



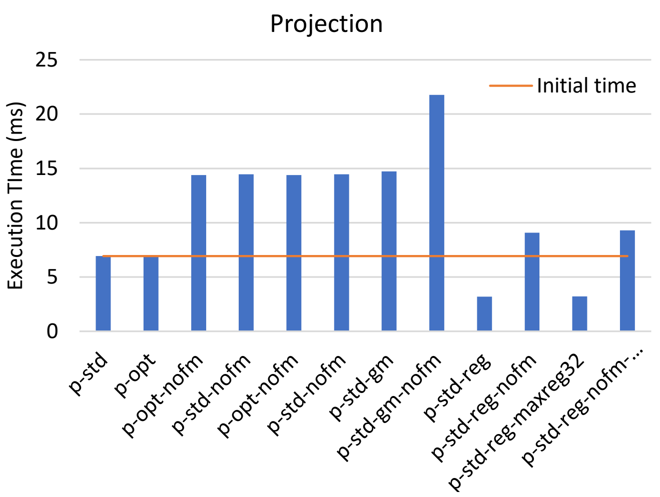


Figure 4.5: *Execution time for different configurations with their best block sizes. Results are shown for the backprojection (left) and the projection (right)*

4.3.2 Commodity GPU

With the aim of studying if the results obtained with the High-Profile GPU are also applicable to other GPUs using other architectures, we have replicated the evaluation on a NVidia GTX980. Total execution times are coherent with what has been obtained from the Tesla K40. In this case, only the small study of 512^3 volume dimension was evaluated due to the memory limitation

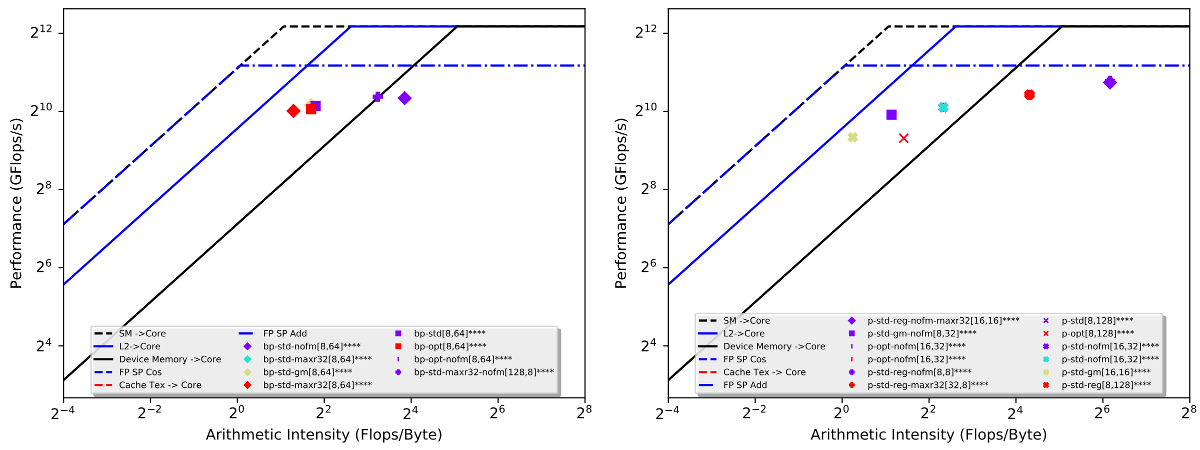


Figure 4.6: *CARM for GTX980 GPU for all configurations with the block sizes that lead to best performance in backprojection (left), and projection (right).*

of this device.

Speedups obtained for *bp-opt* and *p-reg* are significant (as we can see in Figure 4.5), obtaining a speedup of 1.28× for backprojection and 2.22× for projection. The worst execution time is still reached with the *nofm* versions of the different kernels. The penalization is larger for the backprojection kernel in which the nofm version can be 3× slower.

Representation of the different configurations in the CARM are less clustered than in the case of the Tesla option and are closer to the compute-bound region (see Figure 4.6).

Both models (Tesla K40 CARM and GTX980 CARM) are very similar in terms of how the configurations are ordered, although in general they obtain a better performance and arithmetic intensity. GTX 980, although with a newer architecture, is directed towards the commodity hardware segment meanwhile Tesla products are specific for HPC. Taking into account the differences between both architectures, and the larger number of cores of the Tesla model, the datasets provided take better advantage of the GTX980 hardware. The small study is capable of obtaining better advantage of the hardware occupying better the computation units. Additionally the highest clock frequency of the later model provides a better ratio of FLOPs/byte.

4.3.3 Multi-core CPUs

We also evaluated the performance of the kernels in a non-accelerated multi-core environment with a version of the standard configuration of each kernel. The evaluation was executed in a node with 2 Intel(R) Xeon(R) CPU E5-2630 v3, 32 virtual cores and 252 GB of DDR3 RAM.

To generate a CARM, the application was executed inside Intel Advisor 2017. This tool automatically constructs the model for the processor in which is executed (roofs are computed on the fly and can differ between executions). In Figure 4.7, we show the CARM plots for backprojection and projection. These CPU kernels are parallelized with OpenMP. These profiled kernels were not optimized specifically for the architecture in which they were executed and correspond to the gm configuration employed in the GPU kernels. However, Advisor shows that the projection kernel was automatically partially vectorized which can imply an additional performance difference with respect to the backprojection. Backprojection kernel can not be automatically vectorized due to the greater complexity of its internal loops.

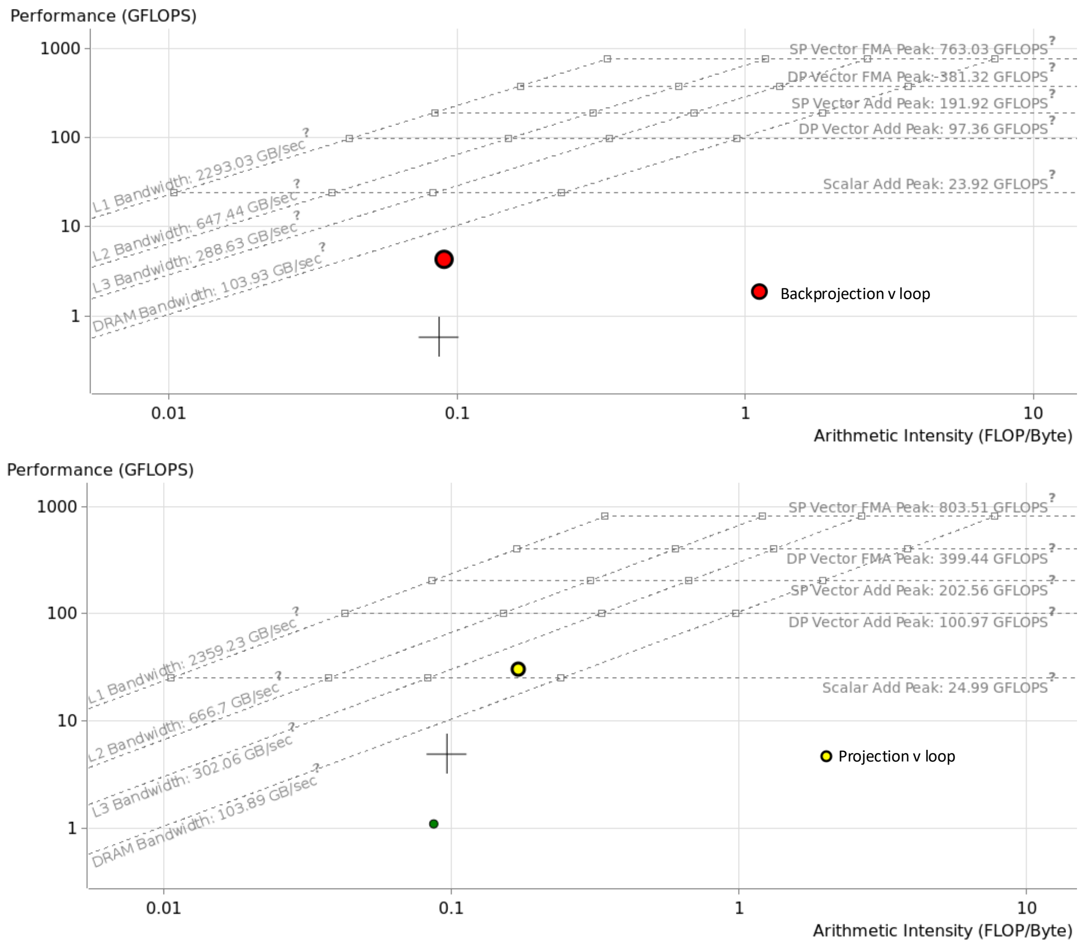


Figure 4.7: *CPU CARM model from Intel Advisor for the backprojection and projection algorithm with OpenMP.*

Both kernels are located in the memory-bound zone of the plot, although the projection kernel seems to be limited by the L3 bandwidth and not the main memory. This situation is similar to the one obtained in GPU CARM were projection was capable of obtaining a higher performance than the backprojection kernel, being this last one the most time consuming kernel.

In this case, only the main memory can be used, so no optimizations employing texture mechanisms can be applied. In Figure 4.8 we show the results for both kernels when executing them without OpenMP. Both kernels are situated in the same zone in despite of the lack of parallelism. Performance decreases due to the larger execution times. The arithmetic intensity varies slightly due to the different optimizations and organization of the code included by the compiler. Therefore, we observe a similar effect to what we have already described in the GPU case, with a large vertical movement in the model when the parallelism pattern is modified.

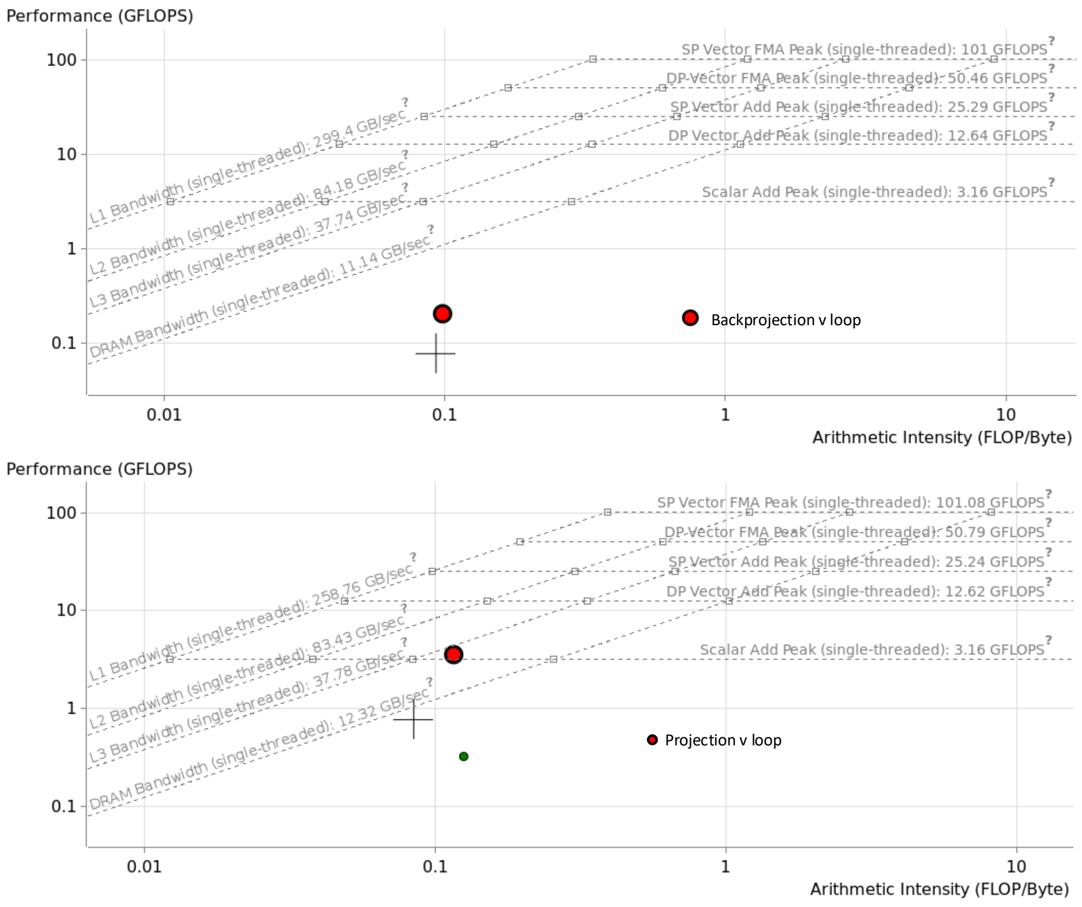


Figure 4.8: *CPU CARM model from Intel Advisor for the backprojection and projection algorithm with a single thread*

4.4 Summary

In this chapter we have presented a full characterization of two medical image processing kernels: backprojection and projection. These kernels, highly used in modern reconstruction algorithms, present certain characteristics that make them suitable for an extensive characterization. Both of them have been implemented in many versions and architectures, including the GPUs. When implemented many optimizations are applied to them to increase their performance. One example is the usage of texture mechanisms or cache optimizations. With respect to our baseline implementation we have been able to obtain a speedup of 2× for the projection kernel and 1.25× for the backprojection kernel.

We can summarize the main contributions of this chapter into the following points:

- Extended roofs for constructing GPU CARMs considering textures and special functions in GPU hardware.
- An study of the influence of parallelism in GPU CARM in terms of block sizes, and a study of the influence of the most used compilation flags for CUDA GPUs, *fastmath* and *maxreg*.
- Optimized backprojection and projection kernels for CUDA GPUs evaluated in different

architectures and performing up to 2× faster than previous versions.

Although some benchmark applications have been already characterized with the GPU CARM, here we have presented a real application and the effects of characterizing it in terms of assisting in the process of optimization to obtain better execution times. We present multiple versions of these kernels, with different optimization levels based on the data obtained from the NVidia profiler and the constructed CARM. We also presented a comparison with the CARM obtained from the CPU versions of these kernels.

As shown in this chapter, the most time consuming components have been improved in terms of performance. Once optimized, it is feasible to incorporate them into an iterative reconstruction solution that is mainly composed by repetitive backprojection and projection stages.

Chapter 5

Design of a fast iterative reconstruction framework for limited-data CT

The source-detector pair in conventional CBCT systems rotates around the patient through 360 degrees (full angular span) to acquire at least 360 projections. However, there is a trend towards reducing the number of projections needed to reconstruct a good quality image. In the case of the health sector, for people or animals, the aim of this trend is to reduce the total radiation received by the patients that go under CT. Another examples are the realization of CT during surgery, due to movement limitations, or in respiratory-gated CT, where only a few projections correspond to each gate. In those cases, the number of projections acquired is smaller than 360 and/or covers a smaller angular span (down to 150 degrees), which causes severe artefacts (streaks and/or edge distortion) when reconstructing with these limited data using the traditional method of FDK.

Due to this problem, it is necessary to use advanced reconstruction methods that compensate for the lack of data by including prior information about the study. The most common option for prior information is the assumption of local smoothness, which can be imposed by adding the minimization of the L_1 norm of the Total Variation (TV) term. However, since the TV term is not differentiable, the use of traditional reconstruction methods may be subject to instability problems [158]. Thus, iterative reconstruction algorithms that rely on the iterative refinement of the final image until a point of convergence or a minimal noise rate are reached are implemented.

One of the main features of an iterative algorithm is the progressive refinement of the image to be reconstructed, which is carried out through the application of specific operators: back-projection and projection. Thus, these iterative algorithms are, in many cases, computationally much more expensive than traditional analytic methods. The general approximation of iterative reconstruction algorithms is the continuous improvement of the final resulting image, taking into account the characteristics of the input radiographs. They are also algorithms that rely on a large amount of memory because they work with large and dense coefficient matrices. As the resolution of the available detectors increase, the size of these matrices becomes unmanageable in standard workstations.

To cope with this problem, Abascal et al. showed in [159] that reconstructing limited data in CT can be efficiently solved using the Split Bregman formulation [160], which reduces the optimization problem to a sequence of unconstrained and simpler problems that are updated iteratively. In any case, the main limitation of previous solutions [161] is that only 2D images can be reconstructed, due to computational and memory requirements. Thus, reconstruction of 3D images with these methods was not possible for two main reasons: (1) memory requirements of the algorithm and (2) long execution times, which hinder the reconstruction of standard size volumes in a reasonable amount of time.

To overcome those limitations, a new iterative framework for limited data, both considering a reduced angular span and number of projections, is proposed in this chapter of the thesis. It is based on an accelerated implementation of the split Bregman method [162] that includes a Krylov subspace solver [161]. The solution proposes the use of GPUs for the most time-consuming operations and includes a partitioning strategy to be able to handle large volumes, with a total footprint of several GB.

5.1 Iterative reconstruction framework

The iterative reconstruction framework has been designed and implemented as a module of FUX-Sim, extending its functionalities to enable iterative reconstruction for flexible geometries. The framework is placed at the *configuration layer* (see Figure 5.1) and can work with the different geometric configurations, which were already described in Chapter 3.

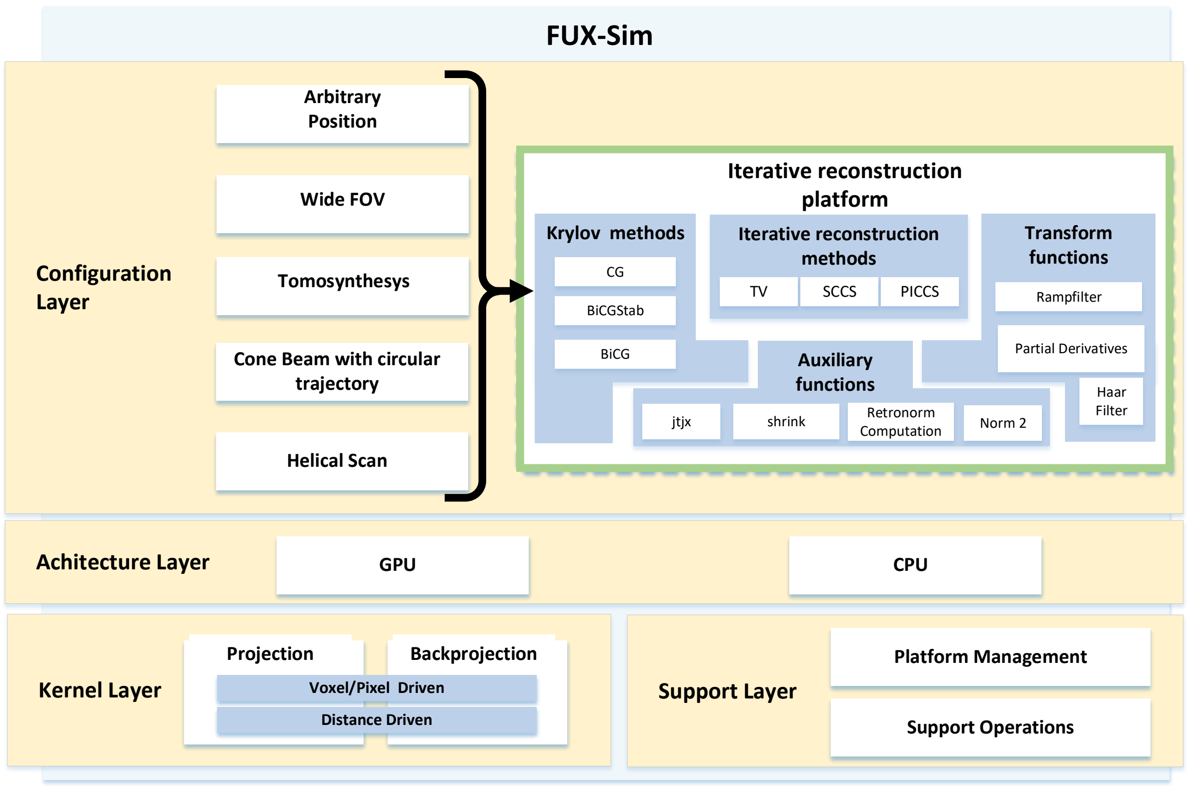


Figure 5.1: *Extended architecture of FUX-Sim including the iterative reconstruction framework.*

The framework is designed around three main components:

- Iterative reconstruction methods: providing the description of the different algorithms that can be executed for iterative reconstruction. At this moment three algorithms have been proposed: TV, Surface Constrained Compressed Sensing (SCCS), and Prior Image Constrained Compressed Sensing (PICCS).

- Krylov methods: this component contains the implementation of mathematical iterative methods to solve systems of linear equations in the space of Krylov. Currently, the matrix-free BiConjugate Gradient Stabilized (BiCGStab) method is supported.
- Transformation functions: providing the functions that can transform data volumes into different domains. The two main functionalities supported are: gradient computation and Haar Filter.

Additionally, the framework provides auxiliary functions that can be used in any of the components mentioned before. Therefore, the execution is characterized by (1) the election of an iterative reconstruction algorithm, (2) a Krylov method (if needed), and (3) one or more transform functions (if they are used inside the method). With this modular approach, it is possible to easily experiment with different algorithms and variations of their internal components. Moreover, the software architecture chosen facilitates the extension of the framework inserting new iterative reconstruction methods in the form of plugins.

The main purpose of the Krylov solver is to solve the equation system:

$$Ax = b \tag{5.1}$$

where A is the system matrix, x and b are vectors representing volume data. The system matrix A is a large matrix representing the relation between the value of the voxels in the volume and the pixel value in the detector. In medium and large studies the size of this matrix increases, as well as the cost to compute it. Thus, it is easier to employ backprojection and projection algorithms than their system matrix representation. For this reason we execute an independent function to compute Ax, which results in a much smaller vector than the system matrix A and requires the usage of a matrix-free compatible solver. We also provide the vector b, being an approximation of the final volume. The specific Krylov space solver employed in this work was BiCGStab [163] in its matrix-free form without any preconditioners.

Furthermore, all the methods and algorithms already present in FUX-Sim are available for its use inside the iterative reconstruction platform, being able to take advantage of the previously explained geometry configurations as well as the management functions. Thanks to this interoperability FUX-Sim is transformed into a completely extensible framework.

5.1.1 Algorithms

Currently there are three iterative reconstruction algorithms supported as methods inside the framework. All of them are based on the total variation minimization problem and make use of the Krylov method and one or more customizable transformation functions. They are included in the *Iterative reconstruction method* component of the *Iterative reconstruction platform* (see Figure 5.1). All methods employ a matrix-free approximation to the matrix A in the linear system $Ax = b$. The function chosen to approximate A is shown in Algorithm 3.

Algorithm 3 Callback function for Krylov space solver (bicgstab).

```
1: procedure JTJX($sol$)
2:     $gradsolx \leftarrow gradient_x(sol)$
```

```
3:      $gradsoly \leftarrow gradient_y(sol)$
4:      $bTV \leftarrow lambda * (gradient\_transp_x(gradsolx) + gradient\_transp_y(gradsoly))$
5:      $sol\_proj \leftarrow projection(sol)$
6:      $bFback \leftarrow backprojection(solMat\_proj)$
7:      $bF \leftarrow mu * bFback * backNormFactor$
8:      $bG \leftarrow beta * solMat$
9:      $Ksol \leftarrow bTV + bF + bG$
10:     return $Ksol$
11: end procedure
```

Total Variation reconstruction algorithm

The reconstruction problem follows the TV minimization method [164]:

$$min \left\|\nabla(u)\right\|_1 \quad s.t. \left\|Au - f\right\|_2^2 \leq \sigma^2, \quad u \geq 0, \quad u \in \Omega \tag{5.2}$$

where $\left\|\nabla(u)\right\|_1$ corresponds to the L_1 norm of the gradient of the reconstructed image u, A is the system matrix, f is the acquisition data, σ^2 is the image noise, and Ω is the subspace corresponding to the FOV.

Using the Split Bregman formulation [158], the L_1-constrained optimization problem shown in Equation (5.2) can be converted into the following unconstrained problems, which are solved at each iteration k:

$$\begin{aligned}(u^{k+1}, d_x^{k+1}, d_y^{k+1}) = min \left\|(d_x, d_y)\right\|_1 + \frac{\mu}{2}\left\|Au - f^k\right\|_2^2 + \\ + \frac{\lambda}{2}\left\|d_x - \nabla_x u - b_x^k\right\|_2^2 + \frac{\lambda}{2}\left\|d_y - \nabla_y u - b_y^k\right\|_2^2\end{aligned} \tag{5.3}$$

$$f^{k+1} = f^k + f - Au^{k+1} \tag{5.4}$$

$$b_x^{k+1} = b_x^k + \nabla_x u^{k+1} - d_x^{k+1} \tag{5.5}$$

$$b_y^{k+1} = b_y^k + \nabla_y u^{k+1} - d_y^{k+1} \tag{5.6}$$

where μ and λ are regularization parameters. Equation (5.3) can be split into two sub-problems. The first sub-problem contains only differentiable L_2-norm terms. By differentiating with respect to u and setting the result to 0, we obtain the following problem:

$$(\mu A^T A - \lambda \nabla^T \nabla) u^{k+1} = \mu A^T f^k + \lambda \nabla^T (d^k - b^k) \tag{5.7}$$

which can be summarized in the following problem:

$$Ku^{k+1} = rhs^k \tag{5.8}$$

that is solved iteratively using a Krylov space solver. In this step, an input parameter β controls the stability of the problem. The second sub-problem contains L_1 terms that are not differentiable. Therefore, it is tackled using analytical formulas (shrinkage operation), which need two additional input parameters α and λ. Finally, Equations (5.4, 5.5, 5.6) are the Bregman iterations that impose constraints for acquired data and total variation, respectively. A pseudocode for the implementation of the algorithm in our framework can be seen in Algorithm 4.

Algorithm 4 Total Variation iterative reconstruction.

```
 1: procedure RECOTV3D($f\_ini, FOV, alpha, mu, beta, lambda, iterations$)
 2:     $f\_back \leftarrow backprojection(f\_ini)$
 3:     $murf \leftarrow mu * f\_back * backNormFactor$
 4:     for $k \leftarrow iterations$ do
 5:         $rhs \leftarrow murf + lambda * gradient\_transp_x(dx - bx)+$
 6:         $+lambda * gradient\_transp_y(dy - by) + beta * u$
 7:         $u \leftarrow bicgstab(@jtjx, rhs, tolKrylov, MaxIter)$
 8:         $gradx \leftarrow gradient_x(u)$
 9:         $grady \leftarrow gradient_y(u)$
10:         $[dx, dy] \leftarrow shrinkage(gradx + bx, grady + by, alpha/lambda)$
11:         $bx \leftarrow bx + gradx - dx$
12:         $by \leftarrow by + grady - dy$
13:         $u \leftarrow u * FOV > 0$
14:         $u\_proj \leftarrow projection(u)$
15:         $f \leftarrow f + f\_ini - u\_proj$
16:         $f\_back \leftarrow backprojection(f)$
17:         $murf \leftarrow mu * f\_back * backNormFactor$
18:     end for
19:     $u \leftarrow uBest/(normFactor)$
20:     return $u$
21: end procedure
```

Surface Constraint Compressed Sensing reconstruction method

The SCCS reconstruction method is thoroughly explained in the work by Molina et al. [161]. Similarly to the previous algorithm it follows the idea of the TV minimization problem:

$$\begin{gathered} min \left\|\nabla(u)\right\|_1 s.t. u \in \omega \left\|Au - f\right\|_2^2 \leq \sigma^2 \\ \|\nabla(u)\|_1 = \sqrt{(\nabla_x u)^2 + (\nabla_y u)^{@}} \end{gathered} \tag{5.9}$$

where u is the reconstructed image, $\nabla_x u$ and $\nabla_y u$ are the gradients of u along the x and y directions, A is the system matrix, f is the acquisition data, σ^2 is the data noise and ω is the subspace that corresponds with the support of the sample. Using the Split Bregman formulation [162], L1-constrained optimization problems can efficiently solved by converting Equation 5.9 into a sequence of unconstrained problems:

$$\begin{aligned}(u^{k+1}, d_x^{k+1}, d_y^{k+1}, v^{k+1}) = min \left\|(d_x, d_y)\right\|_1 + \frac{\mu}{2}\left\|Au - f^k\right\|_2^2 + \\ + \frac{\lambda}{2}\left\|d_x - \nabla_x u - b_x^k\right\|_2^2 + \frac{\lambda}{2}\left\|d_y - \nabla_y u - b_y^k\right\|_2^2 + \phi(v \in \omega) + \frac{\gamma}{2}\left\|v - u - b_v^k\right\|_2^2\end{aligned} \tag{5.10}$$

$$f^{k+1} = f^k + f - Au^{k+1} \tag{5.11}$$

$$b_x^{k+1} = b_x^k + \nabla_x u^{k+1} - d_x^{k+1} \tag{5.12}$$

$$b_y^{k+1} = b_y^k + \nabla_y u^{k+1} - d_y^{k+1} \tag{5.13}$$

$$b_v^{k+1} = b_v^k + u^{k+1} - v^{k+1} \tag{5.14}$$

where μ, λ and γ are regularization parameters. Equations 5.10 to 5.13 are similar to those employed in the TV only method, but this algorithm adds a new constraint for data in the form of Equation 5.14. The first sub-problem is still solved using a Krylov space solver. The pseudocode for the implementation of the algorithm in our framework is shown in Algorithm 5.

Algorithm 5 SCCS iterative reconstruction.

```
1: procedure RECOTV3D($f\_ini, FOV, alpha, mu, beta, lambda, iterations$)
2:     $f\_back \leftarrow backprojection(f\_ini)$
3:     $murf \leftarrow mu * f\_back * backNormFactor$
4:     for $k \leftarrow iterations$ do
5:         $rhs \leftarrow murf + lambda * gradient\_transp_x(dx - bx) +$
6:         $+ lambda * gradient\_transp_y(dy - by) + beta * u + gamma * bv$
7:         $u \leftarrow bicgstab(@jtjx, rhs, tolKrylov, MaxIter)$
8:         $gradx \leftarrow gradient_x(u)$
9:         $grady \leftarrow gradient_y(u)$
10:        $[dx, dy] \leftarrow shrinkage(gradx + bx, grady + by, alpha/lambda)$
11:        $bx \leftarrow bx + gradx - dx$
12:        $by \leftarrow by + grady - dy$
13:        $bv \leftarrow bv + u * surface$
14:        $u \leftarrow u * FOV > 0$
15:        $u\_proj \leftarrow projection(u)$
16:        $f \leftarrow f + f\_ini - u\_proj$
17:        $f\_back \leftarrow backprojection(f)$
18:        $murf \leftarrow mu * f\_back * backNormFactor$
19:    end for
20:    $u \leftarrow uBest/(normFactor)$
21:    return $u$
22: end procedure
```

Prior Image Constrained Compressed Sensing reconstruction method

The Prior Image Constrained Compressed Sensing reconstruction method is thoroughly explained in the work by Abascal et al. [159]. In addition to solving the reconstruction problem using the TV minimization approach it includes a new constraint in the form of a prior image of the object.

$$min(1-\alpha)\left\|\nabla(u)\right\|_1 + \alpha\left\|T_2(u-u_p)\right\|_1 : \left\|Au-f\right\|^2 \le \sigma^2 \tag{5.15}$$

where u is the reconstructed image, σ accounts for the noise in the data and α weights the prior penalty function, $\nabla(u)$ represent the spatial discrete gradient that leads to TV and A is the forward operator (projection). For the T_2 function a gradient can also be chosen although other possibilities are explored, such as a wavelet transform. We can transform the problem into an equivalent unconstrained equation introducing Bregman iterations (b):

$$\begin{aligned}(u^{k+1}, d_x^{k+1}, d_y^{k+1}, w^{k+1}) = min(1-\alpha)\left\|(d_x, d_y)\right\|_1 + \alpha\left\|w\right\|\frac{\mu}{2}\left\|Au-f^k\right\|_2^2 + \\ +\frac{\lambda}{2}\left\|d_x-\nabla_x u - b_x^k\right\|_2^2 + \frac{\lambda}{2}\left\|d_y-\nabla_y u - b_y^k\right\|_2^2 + \\ +\frac{\lambda}{2}\left\|w-T_2(u-u_p)-b_w^k\right\|_2^2 + \phi(v\ge 0, v\in\omega) + \frac{\gamma}{2}\left\|v-u-b_v^k\right\|_2^2\end{aligned} \tag{5.16}$$

From Equation 5.16 we can obtain the following Bregman iterations:

$$f^{k+1} = f^k + f - Au^{k+1} \tag{5.17}$$

$$b_x^{k+1} = b_x^k + \nabla_x u^{k+1} - d_x^{k+1} \tag{5.18}$$

$$b_y^{k+1} = b_y^k + \nabla_y u^{k+1} - d_y^{k+1} \tag{5.19}$$

$$b_v^{k+1} = b_v^k + u^{k+1} - v^{k+1} \tag{5.20}$$

$$b_w^{k+1} = b_w^k + T_2(u_i^{k+1}) - w^{k+1} \tag{5.21}$$

where μ, λ and γ are regularization parameters. In this case the new constraint for the prior is expressed in Equation 5.21. The rest of constraints are solved using the same approaches as the ones presented for previous methods. The pseudocode for the implementation of the PICCS algorithm in our framework can be seen in Algorithm 6.

Algorithm 6 PICCS iterative reconstruction.

```
1: procedure RECOTV3D($f\_ini, FOV, alpha, mu, beta, lambda, iterations$)
2:     $f\_back \leftarrow backprojection(f\_ini)$
3:     $murf \leftarrow mu * f\_back * backNormFactor$
4:     $Tup \leftarrow T2(up)$
5:     for $k \leftarrow iterations$ do
6:         $rhs \leftarrow murf + lambda * gradient\_transp_x(dx - bx)+$
```

```
 7:         $+lambda * gradient\_transp_y(dy - by) + gamma * T2(dx + Tup - cx)$
 8:         $+gamma * T2(dy + Tup - cy)$
 9:         $u \leftarrow bicgstab(@jtjx, rhs, tolKrylov, MaxIter)$
10:         $gradx \leftarrow gradient_x(u)$
11:         $grady \leftarrow gradient_y(u)$
12:         $[dx, dy] \leftarrow shrinkage(gradx + bx, grady + by, (1 - gamma)/lambda)$
13:         $[gx, gy] \leftarrow shrinkage(gradx + Tup + cx, grady + Tup + cy, gamma/lambda)$
14:         $bx \leftarrow bx + gradx - dx$
15:         $by \leftarrow by + grady - dy$
16:         $cx \leftarrow cx + gradx - Tup - gx$
17:         $cy \leftarrow cy + grady - Tup - gy$
18:         $u \leftarrow u * FOV > 0$
19:         $u\_proj \leftarrow projection(u)$
20:         $f \leftarrow f + f\_ini - u\_proj$
21:         $f\_back \leftarrow backprojection(f)$
22:         $murf \leftarrow mu * f\_back * backNormFactor$
23:     end for
24:     $u \leftarrow uBest/(normFactor)$
25:     return $u$
26: end procedure
```

5.2 Accelerated implementation

An accelerated implementation is proposed for all the algorithms already presented (Algorithms 4, 5 and 6). The acceleration is obtained through the use of the extended architecture of FUX-Sim (see Figure 5.1). A workflow representation of the TV method and the approximation function *jtjx* for the Krylov solver is shown in Figures 5.2 and 5.3, respectively. The two main computational kernels are executed both in the external Bregman iteration (red boxes in Figure 5.2), and inside the *jtjx* function. These kernels are executed on the GPU thanks to the optimized implementation that has been shown in Chapters 3 and 4. Other operations that run on GPU are the gradients ($gradient_x$, $gradient_y$), the shrinkage operation (*shrink2*), and the L_2-norm calculation (using CUBLAS library). The remaining element-wise operations are vectorized by the compiler [165] and multi-thread CPU parallelized with OpenMP 4.0 due to their low computational load. Although suitable for problem resolution, the adoption of the Split Bregman approach generates multiple variables, which in the case of 3D iterative reconstruction. Everything must be converted into a 3D volume representation occupying a large portion of the memory. In the case of TV reconstruction we can count up to 8 volumes that need to be allocated simultaneously resulting in a large amount of memory. For the other two algorithms, the number of sub-problems obtained from the formulation is even larger leading to an increased number of variables and a higher complexity.

Given GPU memory restrictions, we designed a partitioning strategy in both backprojection and projection operations, which are the ones that require the highest amount of memory in the device. Similarly to the process devised in Chapter 3, the volumes are partitioned in chunks taking into account the available memory in the device. This is a dynamic process, for every time these functions are called, this partitioning process is activated in order to compute the best chunk size. In case is necessary, a second-level partitioning strategy is triggered, dividing the projections into different sets which are processed and transferred to or from the device.

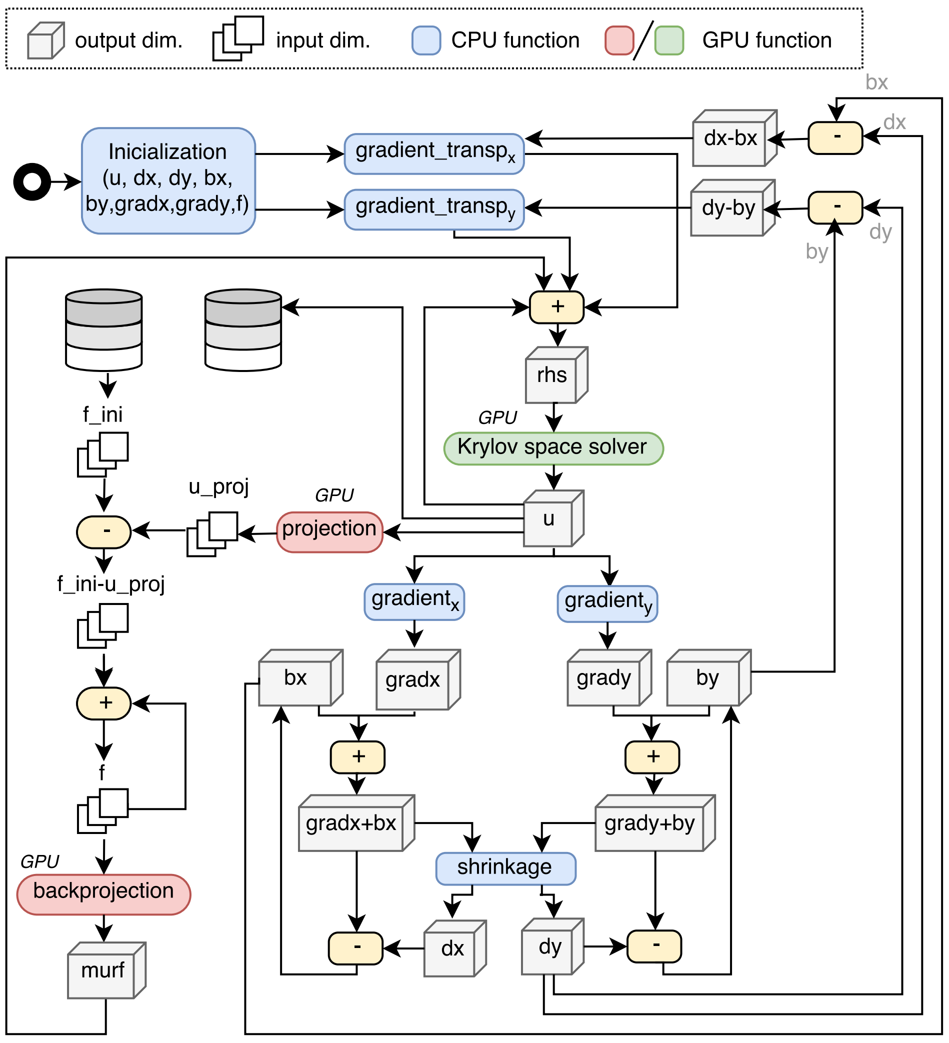


Figure 5.2: *TV3D iterative reconstruction workflow for TV algorithm.*

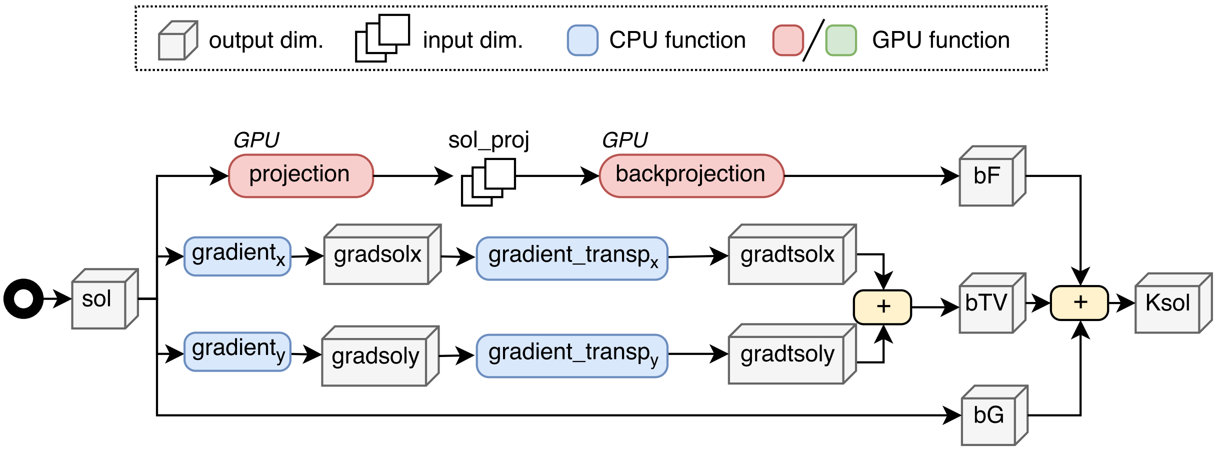


Figure 5.3: *Callback function for Krylov solver workflow for TV algorithm.*

The Krylov space solver is implemented using the BiCGStab method, BiCGStab [166], where the input matrix in Equations (5.7, 5.8) is substituted by the computation shown in Algorithm 3.

5.3 Evaluation

The three methods were evaluated in a computer with two Intel(R) Xeon(R) E5-2630 v3 processors at 2.40 GHz and one NVidia Tesla K40c GPU. Limited-data acquisitions were simulated from a previously acquired small-animal scan ($512 \times 512 \times 512$ pixels; 0.125 mm pixel size), as shown in Figure 5.4, left. We studied the following parameters: dependency on the number of projections with $NumProj$ = 60, 90, and 120 covering an angular span of 360 degrees and $DimProj$ = 512; dependency on angular span for $NumProj$ = 45 uniformly distributed in an angular span of 45, 60, 90, 180, and 270 degrees ($DimProj$ = 512); and the effects of the projection size, by considering $DimProj$ = 256, 512, and 1024 when 90 projections are obtained uniformly distributed in an angular span of 360 degrees. All simulations were generated using FUX-SIM [153].

5.3.1 Study of the TV method

The evaluation of the feasibility and quality of the reconstruction methods is based on the evaluation of the first method, the TV method. This is the basic method from which the rest are constructed and the one with worst results in terms of image quality.

The evaluation data were reconstructed with an FDK-based method [167] and the implemented TV method resulting in a volume of $DimProj \times DimProj \times DimProj$ pixels. For the latter, we used $\alpha = 0.003$, $\mu = 20$, $\beta = 3$, and $\lambda = 2$ as reconstruction parameters (see [159] for details on how to select these parameters). The number of iterations (*iterations* in Algorithm 4, line 4) was 35, selected high enough to ensure an error variation smaller than 1%.

Table 5.1: *Difference of the SNR (dB) between the FDK and the iterative reconstruction; RMSE between the iterative reconstruction and the reference image for different limited-data configurations.*

Angular span	Projections	SNR Difference (dB)	RMSE
135	45	20.79	0.268
150	45	23.20	0.220
360	45	28.27	0.154
360	90	26.17	0.153
360	120	25.98	0.151

Figure 5.4 shows the reference image (FDK reconstruction of the complete dataset) and the results of FDK and the proposed iterative method for two limited-data configurations. Image quality was assessed with two metrics. To evaluate the global image quality, we calculated the RMSE between the reference image and the intermediate solution u^k from the limited dataset. To evaluate the influence of streaks and noise in the reconstructed image, we measured the improvement of SNR obtained with the iterative method with respect to the FDK-based method in the homogeneous area indicated in Figure 5.4. Table 5.1 shows both metrics for different number of projections and angular span, with a noticeable improvement when increasing the angular span despite the low number of projections. Figure 5.5 plots the dependence of the RMSE with the number of iterations, k, for six different limited-data cases varying the angular

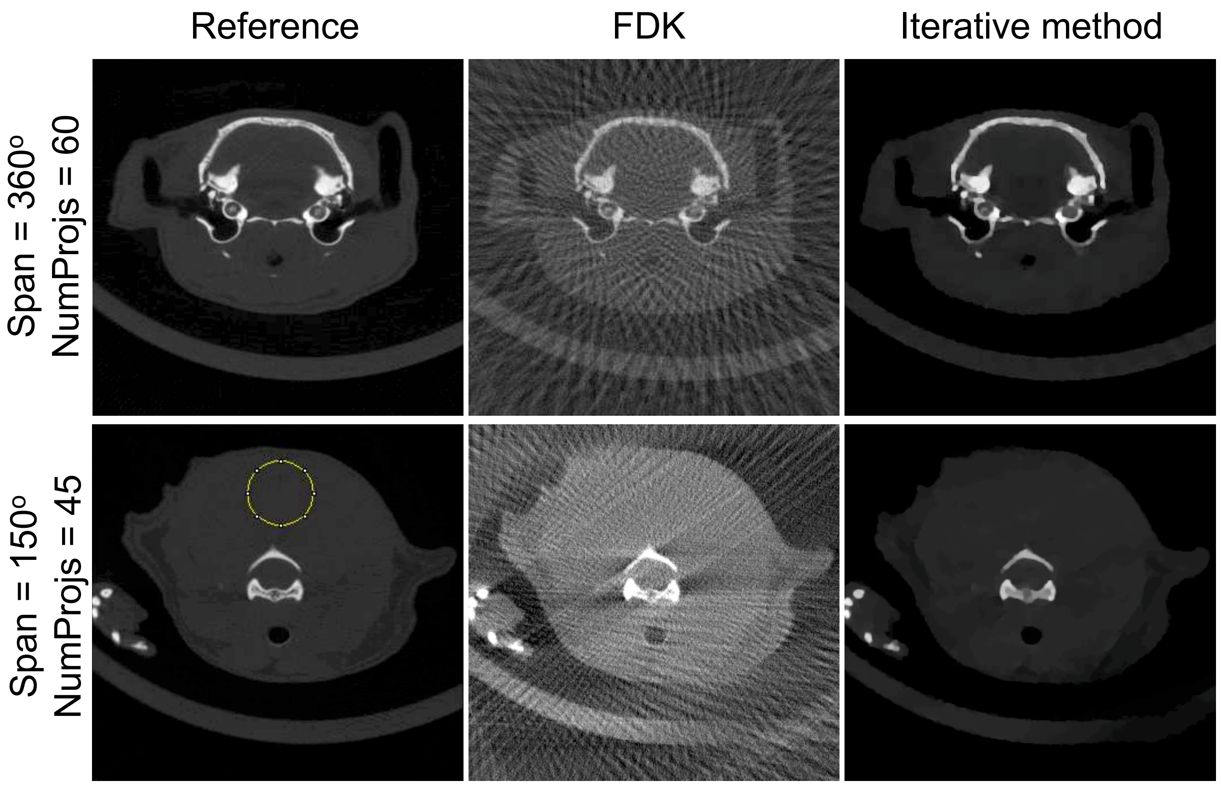


Figure 5.4: *From left to right: reference image, FDK-method and proposed iterative method. Top panel corresponds to the case of 60 projections covering an angular span of 360 degrees and bottom panel to the case of 45 projections covering an angular span of 150 degrees. Yellow circle in the bottom left panel shows the ROI for the SNR measurement.*

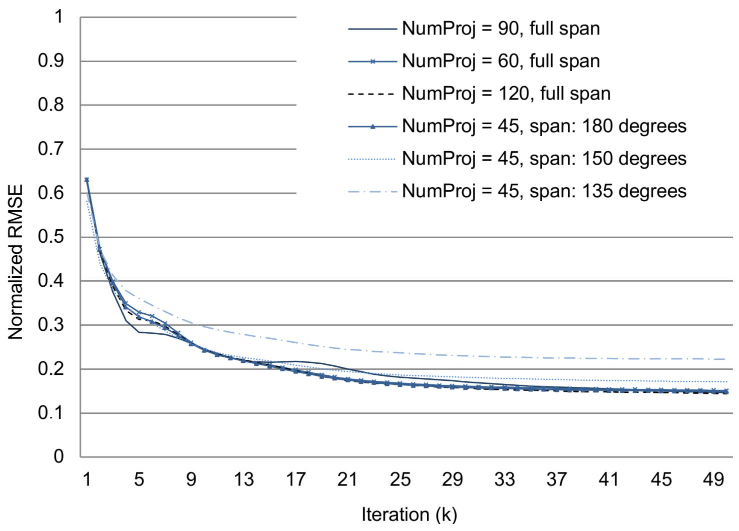


Figure 5.5: *RMSE vs. iterations for: 60, 90 and 120 projections (full span); angular span of 135, 150 and 180 degrees (45 projections)*

span and the number of projections. We can see that the proposed iterative method shows a similar behavior for all limited-data configurations.

Figures 5.6, 5.7, and 5.8 show the breakdown of the reconstruction time of each configuration,

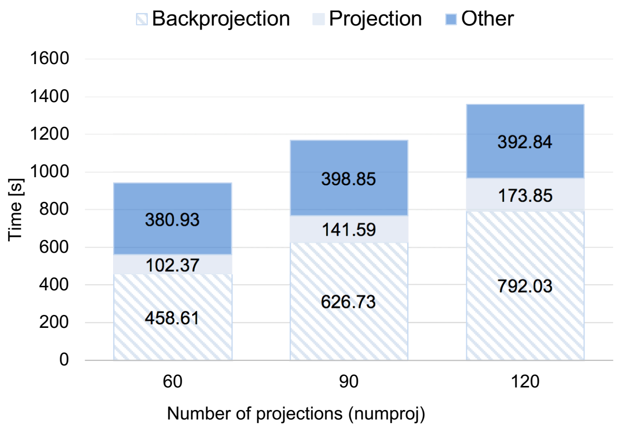


Figure 5.6: *Execution time (in seconds) for different number of projections.*

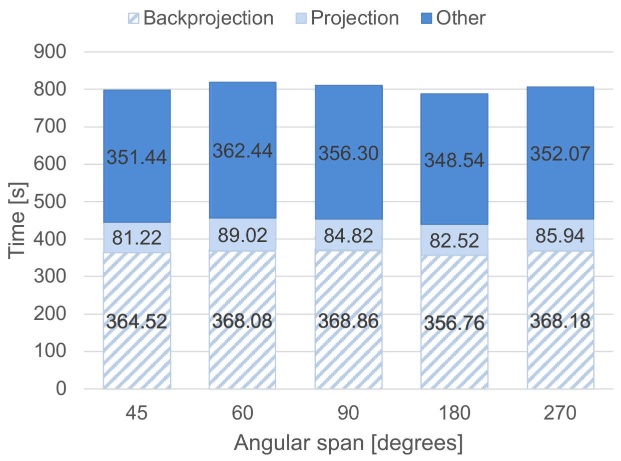


Figure 5.7: *Execution time (in seconds) for different angular span (degrees).*

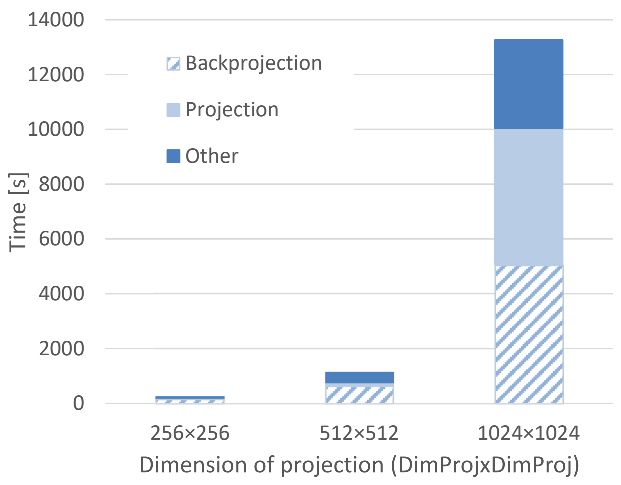


Figure 5.8: *Execution time (in seconds) for different projection sizes (DimProj).*

obtained as the average of three consecutive executions in order to avoid time variability due to operating system operations. Reconstruction time is divided into backprojection, forward projection, and time spent in other operations including I/O operations and CPU computation.

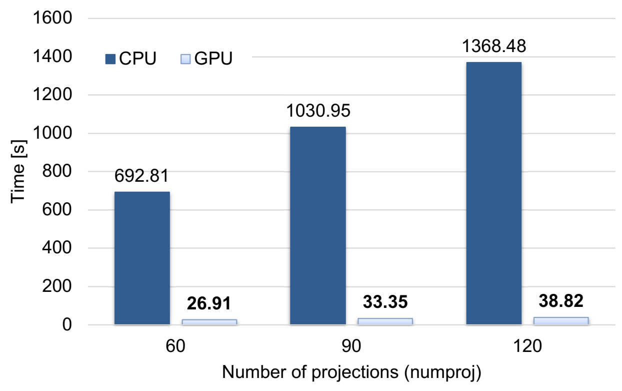


Figure 5.9: *Execution time (in seconds) of the CPU and GPU version of the iterative method in the first iteration for different number of projections (NumProj).*

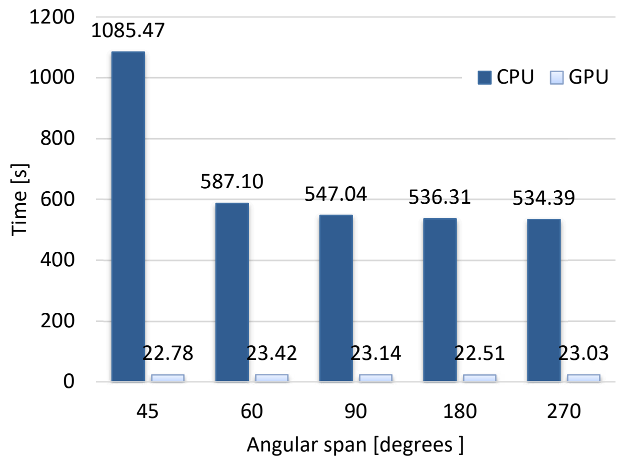


Figure 5.10: *Execution time (in seconds) of the CPU and GPU version of the iterative method in the first iteration for different angular span (degrees).*

Finally, we compared the implementation in GPU of the iterative method with the CPU-only implementation of the same iterative method parallelized using OpenMP to fully exploit multi-core architectures. Figures 5.9 and 5.10 plot the time spent in the first iteration (average of three different executions) reaching a speedup factor of 48× with the GPU implementation with respect with the CPU-only one.

5.3.2 Comparative evaluation of the three iterative reconstruction methods

The previously described data was also employed to evaluate the performance, in terms of total execution time, of the three methods included in the iterative reconstructed platform. All algorithms were configured for a total of 35 iterations although in a real scenario, due to the faster convergence of SCCS and PICCS, not all algorithms should need the same number of Bregman iterations.

For all cases, PICCS performs better in terms of execution time, spending almost half of the time with respect to TV and SCCS.

When exploring the execution time modifying the dimension of the projections (see Figure 5.11), all algorithms scale properly, taking into account that this increment in the size of the

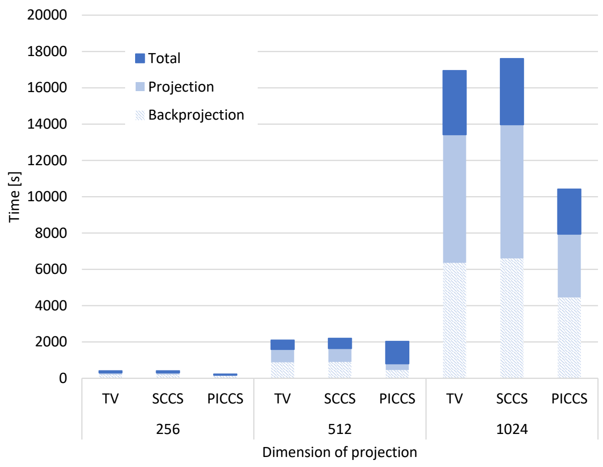


Figure 5.11: *Execution time (in seconds) of the three iterative methods included for different dimensions of the projections.*

projections also affects to the size of the 3D volume, obtaining an increase of the total time of between 6× to 8× for the 512^2 experiment with respect to the small one and of 40× for the largest experiment.

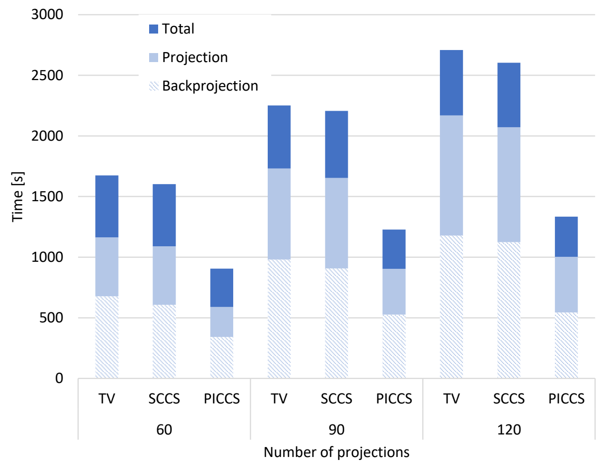


Figure 5.12: *Execution time (in seconds) of the three iterative methods included for different number of projections and a fixed angular span of 360 degrees.*

This scalability is also proved when varying the number of projections for the same angular span (Figure 5.12). The execution time for 90 projections is 1.3× larger than for 60 projections, and for 120 projections the increment is of 1.6× obtaining an increase of 30% of the base execution time for each 30 projections added.

Finally, regarding the angular span (Figure 5.13), times do not differ significantly. TV and SCCS methods perform similarly obtaining SCCS the worst execution times when the angular span is extended.

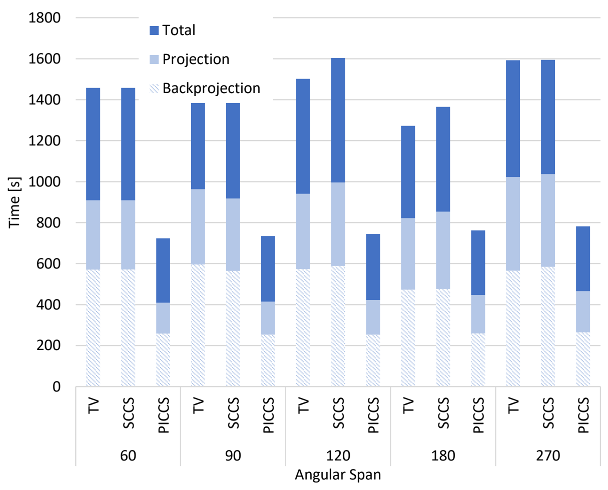


Figure 5.13: *Execution time (in seconds) of the three iterative methods included for 90 projections and a different angular span.*

5.4 Discussion

The evaluation of the proposed framework shows a high reduction of the severe artefacts observed when using the conventional FDK-based method for cases in which a low number of projections is available, with an SNR improvement of more than 20 dB for all cases. The image distortion due to the limited angular span was also reduced with the proposed method.

To evaluate the performance of the implementation according to data size, we fixed a high number of iterations ($iterations = 35$) for all experiments in order to ensure optimum image quality for the worst conditions. Nevertheless, in some cases, the number of iterations could be lowered, resulting in shorter reconstruction times: for example, with 60 projections and an angular span of 360 degrees, 20 iterations were enough for a high quality image. In all experiments, backprojection and forward projection operations represented at least 50% of total execution time, reaching a maximum of 80% of the time when the acquired data set is large. Neither the iterative Krylov space solver nor reading and writing operations significantly increase the total time. The execution time of the proposed implementation varies linearly with the number of projections and does not depend significantly on the angular span. The modifications in the size of the input projections and of the result volume produces a linear increment in the execution time which is consistent with the computational complexity of the backprojection component.

The scalability is also present in the other methods: SCCS and PICCS both in terms of projection size and number of projections. PICCS performs between 60% to 100% better than SCCS and TV. The best performance of PICCS is obtained for larger data sizes, reducing its advantage with smaller projections. This performance enhancement comes from its better convergence, which reduces the number of iterations needed inside the Krylov solver. SCCS and TV do not present differences in execution time.

Reconstructions of large studies (volume of $1024 \times 1024 \times 1024$ pixels) are feasible with this accelerated implementation of the iterative method thanks to the partitioning strategy followed for both backprojection and forward projection operations.

The GPU implementation here presented showed significant time reduction (up to $48\times$) compared with a CPU-only implementation, resulting in a decrease of the total reconstruction time from several hours to few minutes. A fair comparison with other iterative reconstruction implementations proposed in the literature is not feasible due to differences in the specific algorithms and the hardware used. Nevertheless, we note that the work by Matenine et al. [143], which is the most similar to our solution, was limited by the memory capacity of the GPUs and did not address the problem of limited angular span. In contrast, our GPU accelerated algorithm obtains similar results in terms of execution time in spite of the fact that it works with large detector and reconstructed volume sizes with a low number of projections in a limited angular span, which increases significantly the number of iterations needed for convergence.

Regarding the previous implementations of the same algorithm, the implementation we propose substantially reduces reconstruction time and hardware resources. As previously reported [161], a solution combining MATLAB and CUDA kernels resulted in large execution times, which is not feasible when reconstructing 3D volumes in real scenarios. With the solution presented in this chapter the execution times are much lower thanks to the use of native code that allows more efficient optimizations of the algorithm implementation.

This efficient implementation using parallel processing and large-memory management strategies together with GPU kernels enables the use of advanced reconstruction approaches which are needed in limited-data scenarios.

5.5 Summary

An accelerated implementation of a method for 3D limited-data tomography has been presented. It solves the problem of iterative reconstruction in an efficient way by using GPUs for the most time-consuming operations.

The main contributions of this chapter are:

- An iterative reconstruction framework for TV based reconstruction methods with GPU acceleration.
- An evaluation of the efficiency in terms of performance for different iterative reconstruction methods based on TV.
- A matrix-free BiCGStab implementation compatible with GPU callback functions.

The solution here presented is suitable for standard resolution datasets. However, when reconstructing high resolution and large size studies (greater than 1024^3) the large memory requirements for maintaining all the necessary data volume prevents the execution in standard machines. Therefore, for large size studies a distributed memory implementation that can take advantage of the larger resources present in clusters and clouds can be the solution to the problem. The next chapter covers the challenge of reconstructing in distributed environments.

Part of this chapter has already been published in the work: *"GPU-accelerated iterative reconstruction for limited-data tomography in CBCT systems"* [168].

Chapter 6

A distributed iterative reconstructor

The proliferation in the last years of many iterative algorithms for CT is a result of the need of finding new ways for obtaining high quality images using low dose or limited-data acquisitions. These iterative algorithms are, in many cases, computationally much more expensive than traditional analytic methods. The general approximation for iterative reconstruction algorithms is the continuous improvement of the final resulting image, considering the characteristics of the input radiographs. They are also algorithms that require a large amount of memory because they work with large and dense matrices. As the resolution of the available detectors increases, the size of these matrices becomes unmanageable in standard workstations.

The work presented in this chapter, developed during an international stay at IMEC, is dedicated to the adaptation of an iterative reconstruction algorithm to a distributed environment using the PETSc library. With the contribution of Roel Wuyts and Tom Vander Aa, we propose a solution capable of overcoming the memory limits of single node executions by increasing the computational resources. For the construction and development of the distribution of the iterative algorithm, we have extended the FUX-Sim simulator from Chapters 3, 4 and 5. The basis of the distributed implementation is the possibility of, not only increasing the computing power due to the increase in the number of cores or accelerators available, but also, to increase the resources that enable the reconstruction of larger volumes and the use of a higher number of projections. This solution can be applied to other type of iterative algorithms or even to standard analytical methods in the reconstruction of large volumes.

At this point of the thesis, the limitations of single-node shared memory architectures disappear thanks to the use of distributed memory solutions. In this chapter, we will focus on distributed memory architectures inside the hybrid HPC paradigm (see Figure 6.1), with special emphasis in the MPI+OpenMP programming model, mainly used in supercomputers and scientific oriented clusters.

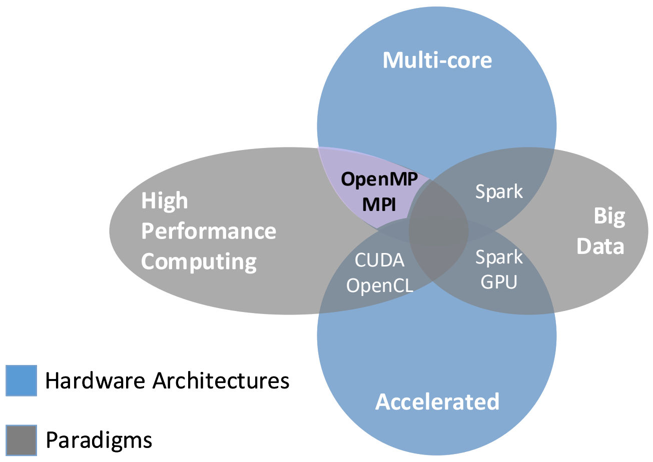


Figure 6.1: *Topics covered in this chapter with respect to the thesis for the distributed iterative reconstructor.*

6.1 Distributed iterative reconstruction design

For the construction and implementation of the iterative algorithm, the proposed simulation and reconstruction framework [153] has been extended to a hybrid distributed-parallel CT simulator using PETSc (Portable, Extensible Toolkit for Scientific Computation), a library [169] that relies on MPI for the communication between different nodes. This library contains multiple methods for matrix and vector computation, as well as optimized and distributed versions of the most popular iterative solvers, including BiCGStab. It also allows the integration with external mathematical libraries and even can take advantage of specialized hardware, like GPUs.

The implementation of the distributed routines is based on the use of parallel structures included in PETSc, the Distributed Memory Distributed Arrays (DMDAs), which describe the parallel structure of an object (a vector or matrix) including the partitioning, ordering, interpolations, and ghost or stencil regions. Designing the distributed functions taking into account this structure results in a partitioning-independent implementation, which provides a more flexible execution of the application in diverse environments. Its compatibility with MPI allows the user to program hybrid algorithms that combine native MPI functions with PETSc structures and methods.

This PETSc-MPI module is the one in charge of distributing the computing load and applying functions that require a coordination between the different processes. Inside each process, the application can have access to the internal FUX-Sim modules, including the backprojection and projection kernels.

In Figure 6.2 the PETSc-MPI module is incorporated in the FUX-Sim architecture. Previously, FUX-Sim was restricted to shared memory architectures. The optimizations made for this type of architectures is still conserved in the work presented in this chapter thanks to the shared memory support inside the distributed iterative reconstructor.

Because of this duality, we can address the parallelism at two different levels: an external coarse-grained parallelism, that takes advantage of distributed resources, and an internal fine-grained parallelism, that is optimized to successfully exploit local resources.

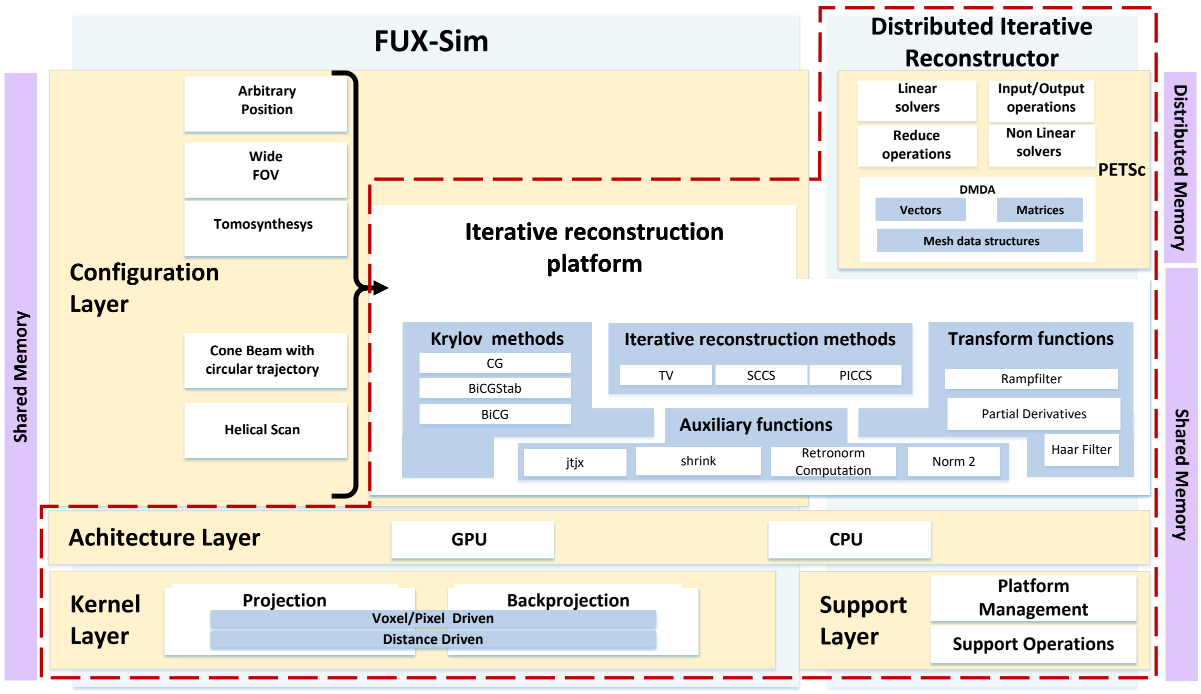


Figure 6.2: *Distributed Iterative reconstruction architecture compared to FUX-Sim architecture. Red dotted line highlights the components evaluated in this chapter.*

6.1.1 Distribution strategy

In distributed memory architectures, data are divided between the different machines or nodes, making it difficult to access the parts of the data that are not locally stored in the machine. Additionally, these data must be partitioned taking into account the communication cost of transferring data from one machine to another, or even, from one process to another.

The distribution of the data also affects the parallelization of the computation. The distribution and parallelization strategy was constructed around the division of the output data (i.e. volume, see Figure 6.3) in different chunks. This division is done in the z axis (as seen in Figure 6.4) to encourage data locality and to avoid unnecessary communications. This division was already proved successful for shared memory architectures, when not enough memory was available to hold the complete 3D volume. It was also employed as a parallelization strategy in the case of detecting the presence of more than one GPU. This is the best distribution strategy due to the reduction of the number of transfers and synchronization points in the algorithm. Although most of the operations are voxel independent (sum of vectors, vector scaling, vector subtraction, etc.) and therefore are not favoured by any partitioning schema, operations such as the backprojection and the gradient operations are dependent in terms of slices ($u-v$ planes). If the partitioning is executed using any of these axes, a communication of halo values or updated values on the limit of the partition would have to be done in order to obtain correct results. The other possibility is to partition the data dividing the projections into different sets to be processed by each machine independently. On the one hand, this last approach would favour the projection operation. On the other hand, partitioning projections would mean to maintain all volumes in all nodes at the same time, an action that would harm the memory gained due to the use of distributed resources. Considering the importance of being able to reconstruct high

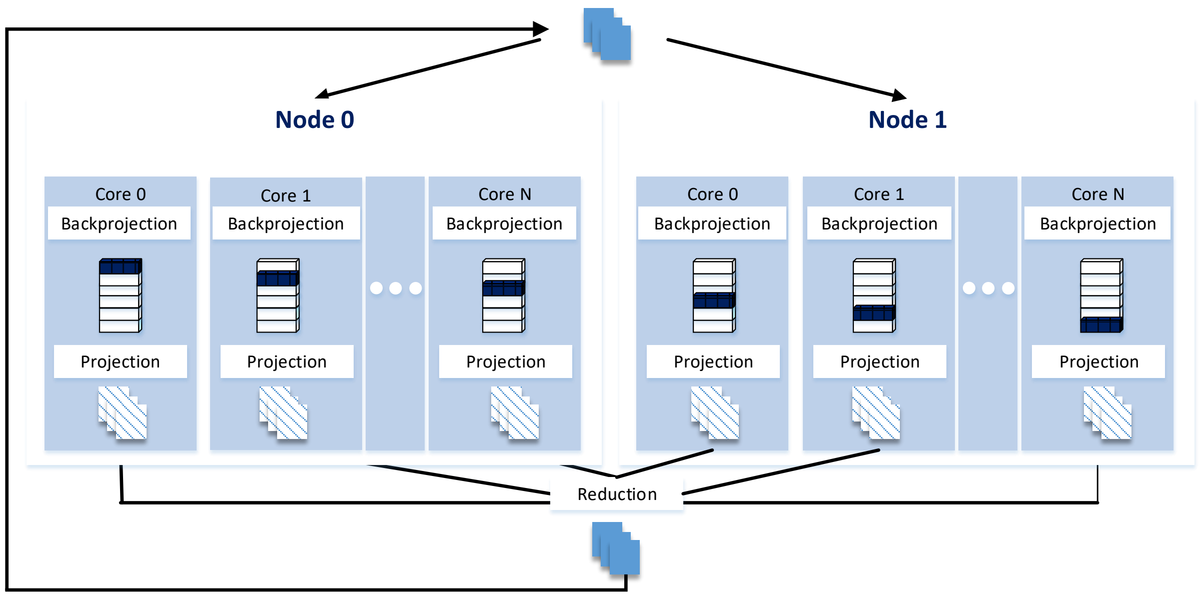


Figure 6.3: *Data partitioning scheme for MPI execution of the iterative reconstruction algorithm.*

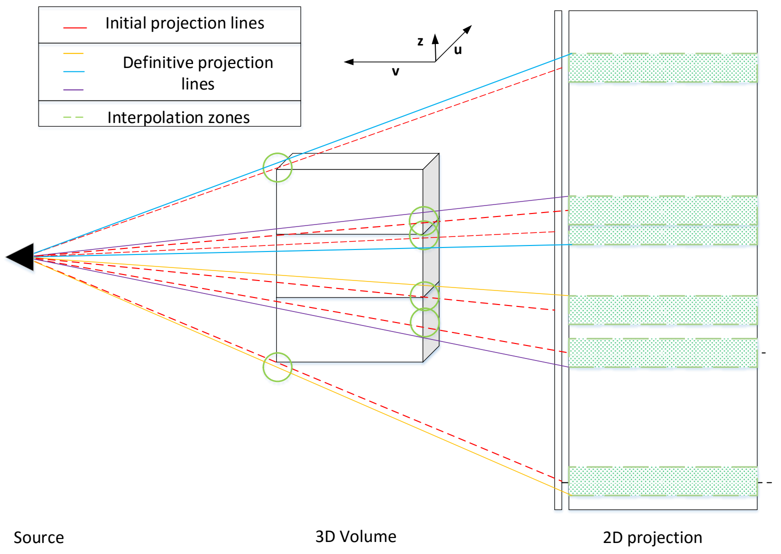


Figure 6.4: *Interpolation zones that must be reduced inside the volume and projection. Each partition of the volume is held by a different node.*

resolution volumes, which can occupy dozens of GB, this second possibility is not viable.

The decision of choosing to divide the volume in different chunks also comes with some disadvantages. There is a dependency in the z plane inside the algorithm: the execution of the backprojection and projection step over a partition of the volume generates incomplete information. As we have seen in previous chapters the backprojection and projection, when separated, constitute embarrassingly parallel problems. Backprojection is an operator in which the generation of each of the voxels is totally independent, and the same happens for projection on the different pixels. However, if we want to obtain a correct volume, the projections must also be complete. As shown in Figure 6.4, the application of the projection operator over independent chunks of the volume generates incomplete projections in the z axis, leading to the computation of incorrect voxel values in the subsequent backprojection step. To overcome this problem, it is necessary to obtain the complete and correct projections in all distributed processes, thus creating a synchronization point. At this synchronization point, the reduction (shown in Figure 6.3) takes place, obtaining the sum of all the interpolation zones of each projection for each process.

This division strategy provides adequate results for the backprojection operator, but it is not expected to provide scalability in the case of projection. This is due to the fact that the projection complexity, $O(DimProj * DimProj * DimVolv)$ does not depend on the partitioned variable, the dimension of the volume in z. To provide scalability, we had to decrease the size of the area of the computed projections for each partitioned volume computing the maximum and minimum projections lines. These projection lines (red dot lines in Figure 6.4) represent the limits of the contribution of the partial volume (the chunk) to the projection. The green zones are sections of the projection that have to be computed with values coming from different chunks, and therefore, from different memory regions.

At the shared memory level, parallelism can be achieved using one of the methods already included in FUX-Sim. For homogeneous architectures, the kernel optimizations with OpenMP are employed, with a parallelization at the voxel/pixel level as previously explained in Chapter 3.

6.2 Experimental study

To evaluate the feasibility of the solution proposed, a preliminary evaluation of the implementation has been executed on a single node and in a distributed environment composed of 12 physical nodes. The purpose was to evaluate the scalability of the application for a varying number of MPI processes. The evaluation was carried out on nodes with an Intel(R) Xeon(R) CPU X5660 @ 2.80GHz with 12 physical cores and 96 GB of memory. The version of MPI used was MPICH 2 with the Hydra manager. The application was compiled with GCC 4.9 and the execution was configured to have the processes bound to the physical cores of the node. The PETSc version employed was 3.7.

The application was tested with real data and two types of studies:

- Small study: 360 projections of 128^2 pixels of a crocodile scapula for the single node execution. The output data consisted on a volume of 128^3 voxels.
- Medium study: 180 projections of 512^2 pixels and the volume output data was 512^3 voxels.

The small study was employed for the first experiment evaluated (MPI over one node). In contrast, for the rest of experiments the medium study was used. Only two iterations of the TV distributed algorithm were executed, considering that the time per iteration is stable.

To determine the effectiveness of the distribution scheme, the execution time was measured in the different phases of the algorithm: reading of the projection data (READ), broadcast of the initial projections to all participating processes (BCINIT), Krylov solver (KRYLOV), and writing of the resulting volume (WRITE). It is important to highlight that backprojection and projection times are obtained from different parts of these phases since they are executed inside and outside the Krylov solver. The phase of projection reduction it is also present inside and outside the solver and therefore shown in a different plot for a better comparison.

6.2.1 Single node execution

The first step is to evaluate the distributed iterative reconstructor in a shared memory environment, to be able to compare the performance in both architectures.

MPI execution

First we evaluate the performance and scalability of the solution with one level of parallelism (coarse-grained). Therefore, for every core in the machine an MPI process is created. Figure 6.5 plots the execution time for each of these phases, as well as the total time employed and the ideal progression time with an ideal scaling. This ideal scaling has been computed as $\frac{time(1)}{num\_processors}$. Krylov solver takes most of the time of the application, representing other phases less than 1% percent of the total time. In Figure 6.6, the time spent in the reduction of the projections is shown compared to the total time to execute the algorithm. This time is almost negligible and does not increase significantly with the number of cores.

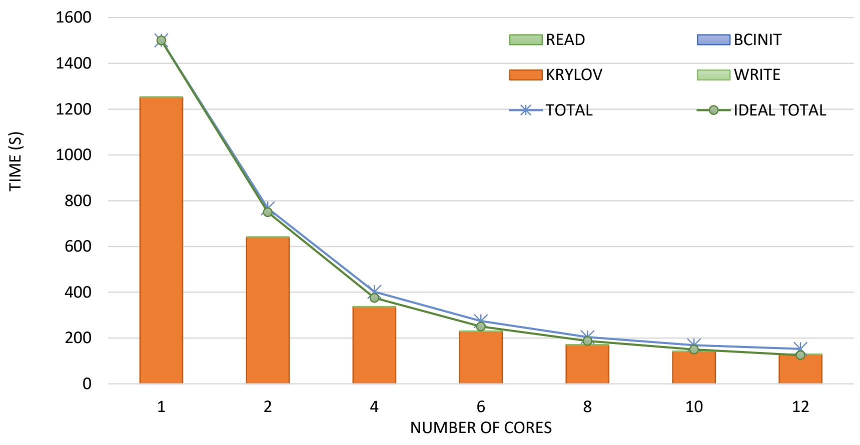


Figure 6.5: *Single node: Execution time (in seconds) of the application for different number of cores and stages.*

In terms of scalability, Figure 6.7 depicts that the backprojection follows almost the same progression as the linear ideal speedup, having with 12 processes a speedup of 11.12× with respect of the execution with just one MPI process. The scalability of the projection kernel is poor compare to the one obtained with the backprojection kernel. This is an expected behavior since, as explained before, the projection does not scale directly with the partitioning variable (the z axis division), but with the projection of these subvolumes over the detector. This means that the scalability of the projection will be worst that than of the backprojection and that also depends of variables such as the distance between source, object and detector, and pixel and voxels sizes. In these first experiments, the influence of the reduction of the projections is not significant because of the low number of processes. However, in further evaluations at larger scales, this reduction phase represents a problem to be tackled.

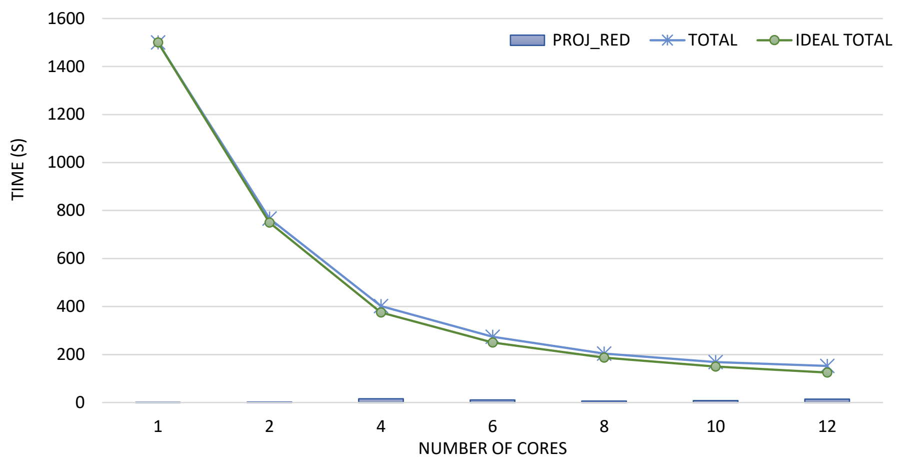


Figure 6.6: *Single node: Projection reduction time (in seconds) of the application for different number of cores.*

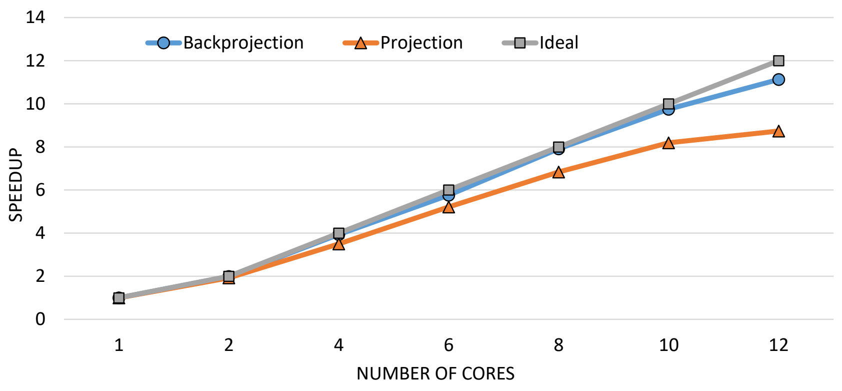


Figure 6.7: *Single node: Speedup for the backprojection and projection phases for different number of cores vs ideal speedup.*

Hybrid MPI+OpenMP evaluation

In the same node, the approach of using OpenMP as fine-grained parallelism inside an MPI process is evaluated. In this case, for the execution on a single node, only one MPI process is created and different number of OpenMP threads are used to parallelize the computation. The number of OpenMP threads employed is equal to the number of cores.

Similar results are obtained with total execution times near the ideal ones (see Figure 6.10), in which most of the time is spent in the Krylov solver. MPI communication does not affect the total execution time. The initial broadcast is negligible and projection reduction time is lower than in the case of using only MPI, since the memory is contained in only one process (Figure 6.8). In fact, this time is almost negligible compared to the total time.

The speedup obtained with this approach is also near the ideal, with positive results for both backprojection and projection stages: projection reaches 11× speedup for the 12 processor case meanwhile with MPI processes the limit was around 9×. However, backprojection speedup do not differ from those obtained using only MPI, since the change in the parallelization approach does not modify the workload of the different computational threads, thus, not obtaining any

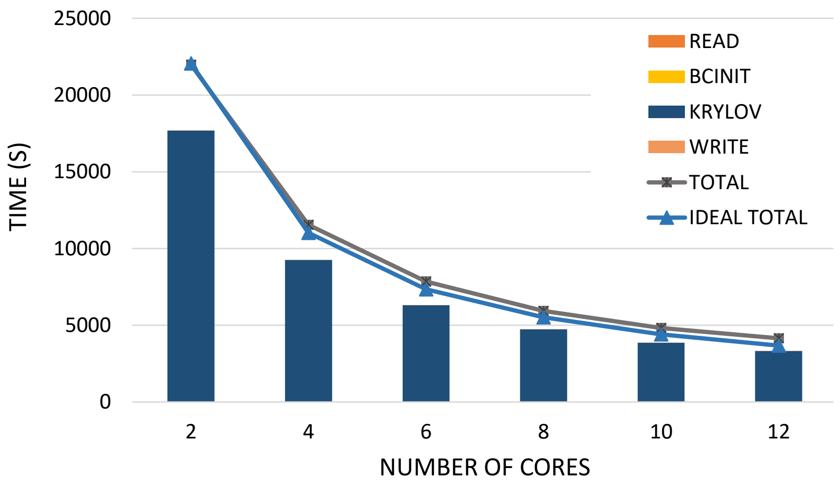


Figure 6.8: *Single node: Execution time (in seconds) for the different phases of the iterative reconstruction algorithm for different number of cores vs ideal speedup with the MPI+OpenMP approach.*

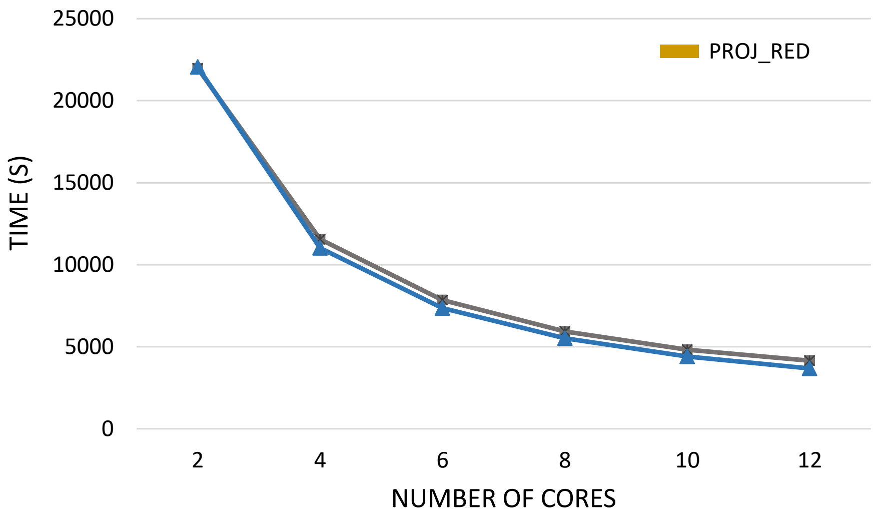


Figure 6.9: *Single node: Execution time (in seconds) for the total execution time and the projection reduction phase of the iterative reconstruction algorithm for different number of cores vs ideal speedup with the MPI+OpenMP approach.*

significant advantage with this option.

6.2.2 Distributed execution

In the distributed solution one-level and two-level parallelism were evaluated. The MPI processes are always distributed equally between nodes.

MPI evaluation

Figure 6.11 shows the total execution time compared with the ideal one for a different number of cores.

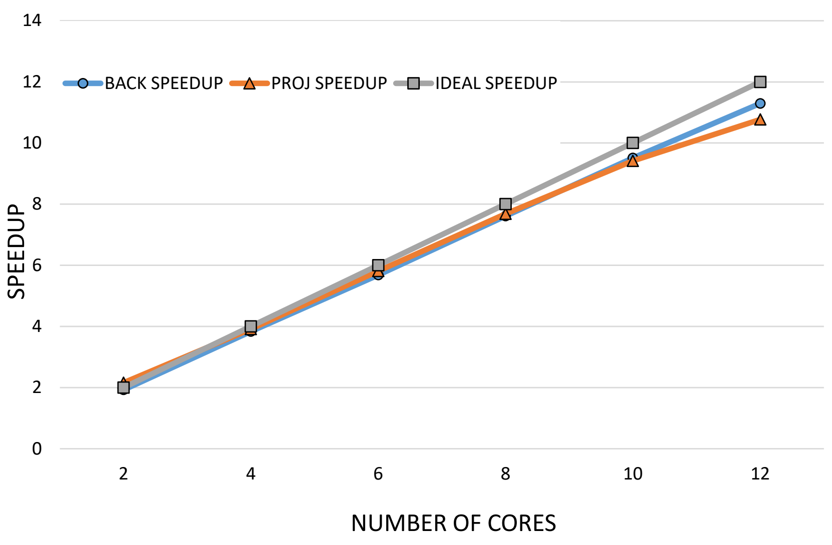


Figure 6.10: *Speedup for the backprojection and projection phases for different number of cores vs ideal speedup with the MPI+OpenMP approach.*

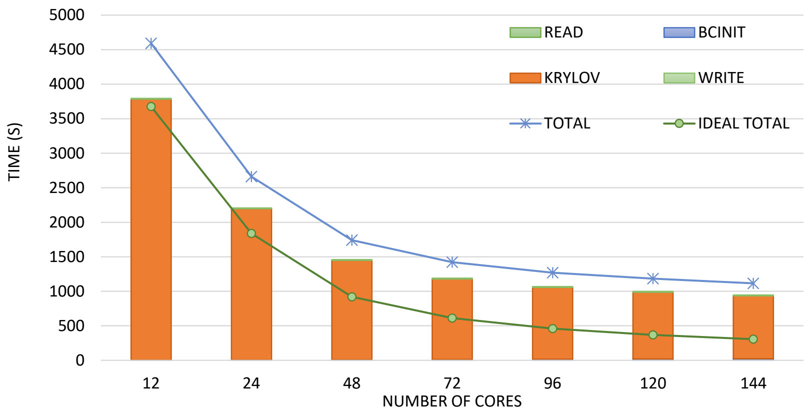


Figure 6.11: *12 nodes: Execution time (in seconds) of the application for different number of cores and stages.*

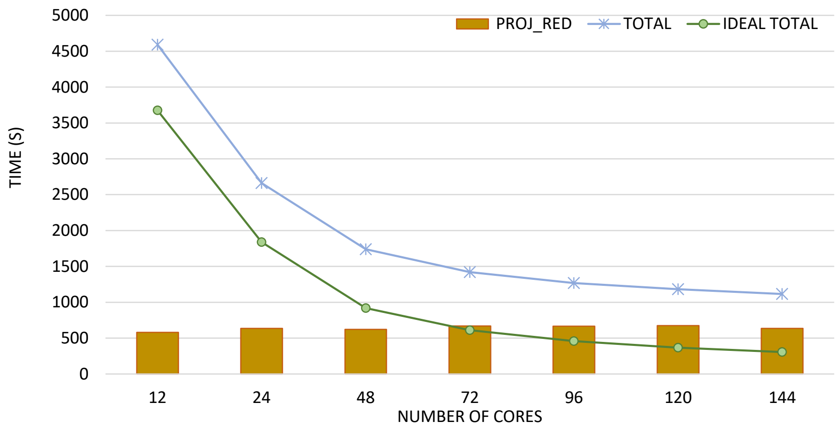


Figure 6.12: *12 nodes: Execution time (in seconds) of the application for different number of cores and stages including time spent in projection reduction.*

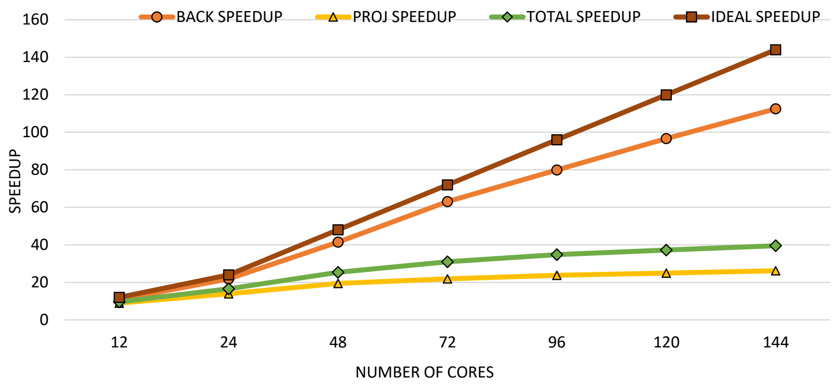


Figure 6.13: *12 nodes: Speedup for the backprojection and projection phases for different number of cores vs ideal speedup.*

As in the evaluation with the single node, Krylov solver represents a large percentage of the execution time of the application, although a larger gap with the total time than in the previous evaluations is observed. This gap includes minor functions, such as matrix subtractions and vector multiplications, as well as the projection reduction. Studying the execution time for all the projection reductions performed during the execution (both included and not included inside the Krylov solver), we can observe in Figure 6.12 that the combination of results to obtain the final projection planes (the reduction) now represents a higher percentage of the total time, being, with 144 cores almost 50% of the total execution time.

This naturally influences the maximum speedup that can be obtained considering parallelization, as we show in Figure 6.13. The backprojection operator scales nearly ideally. However, the inadequate scaling of the projection operator due to the previously explained reasons, as well as the problem introduced with the projection reduction cost reduce significantly the total speedup of the application that reaches a maximum of 40 when using 144 cores.

Hybrid MPI+OpenMP evaluation

For the two-level parallelism evaluation, one MPI process is spawned for each node and then the number of OpenMP threads is increased up to the total number of cores in the cluster.

In Figures 6.14 and 6.15, times for the different phases of the algorithm are shown. Times are significantly better than those obtained only with MPI (3500 seconds vs 2500 seconds for the same number of cores) due to the better performance of the Krylov phase, mainly composed by projection and backprojection kernels. The projection reduction phase is also benefited by the MPI+OpenMP approach reducing the number of processes participating in the reduction operation and taking advantage of the shared memory architecture inside the node.

Regarding the speedup obtained with this approach we can conclude from Figure 6.16 that even though the times are better than those obtained with MPI, scalability is still restricted in the projection algorithm, reaching a maximum speedup of 70× when executing over 144 cores, almost double than the speedup obtained with only MPI.

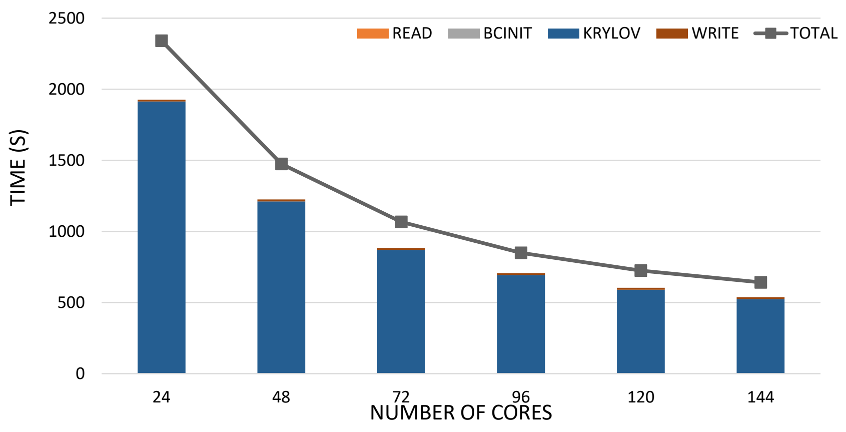


Figure 6.14: *12 nodes: Execution time (in seconds) of the application for different number of cores and stages using MPI+OpenMP.*

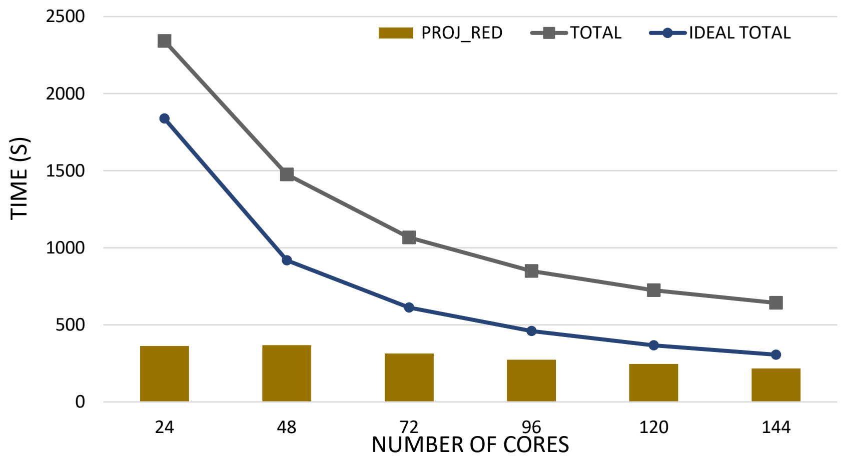


Figure 6.15: *12 nodes: Execution time (in seconds) of the application for different number of cores and projection reduction stage using MPI+OpenMP.*

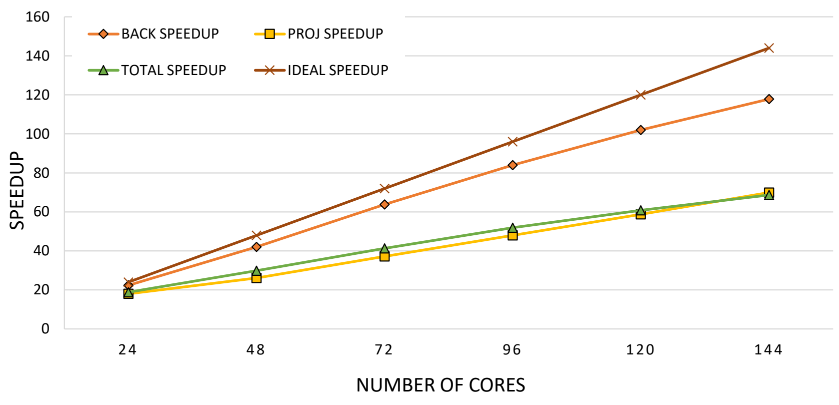


Figure 6.16: *12 nodes: Speedup for the backprojection and projection phases for different number of cores using MPI+OpenMP.*

6.3 Summary

In this chapter, we have presented the implementation and evaluation of an iterative reconstruction method for CT for a distributed environment. For the distribution of the application we have chosen PETSc, a mathematical library on top of MPI that already implemented some of the algorithms needed for our method. We have partitioned the main output dataset to provide scalability as well. In the evaluation, we observed that our implementation scales for the main components (backprojection and projection operators) and it is possible to execute in parallel with little differences in the final result. However, when distributing over several nodes, the specific characteristics of the projection operation along with the increasing cost of the projection reduction impacts negatively on the overall performance providing poor speedup.

The main contributions of this chapter are:

- An iterative reconstruction algorithm adapted to a distributed environment using the PETSc library.
- An study of the scalability in a HPC environment of the backprojection and projection kernels.

Finally, one of the main conclusions is that the reduction of the projection data must be optimized. In the distributed version, this phase represents up to 33% of the total time, reducing the possibilities of scaling to high resolution images. With the distributed platform presented in this chapter, the current solution approach unbalances the load of each node, thus preventing the algorithm from better scaling. MPI does not possess native mechanisms for load balancing and the management of irregular partitioned data is difficult due to the assumption that all hardware is similar and that the load will be balanced by the programmer. Due to these restrictions and the requirements given by the domain, studies over a different type of paradigm, closer to the data and the programmer, are needed. This platform must provide an easier management of irregular data as well as a resource manager capable of distributing the partitioned tasks between the resources.

Part of this chapter has already been published in the work: *"Exploring a Distributed Iterative Reconstructor Based on Split Bregman Using PETSc"* [170].

Chapter 7

Iterative reconstruction framework based on the Big Data paradigm

Programming models for Big Data, like MapReduce, have recently been embraced in many scientific fields. The easiness in which data are managed in massively distributed environments, such as clusters or cloud systems [171], has transformed to the Big Data paradigm many applications of the biomedical field, in which data analysis or generation is fundamental, like DNA analysis, protein discovery, and image processing. The simplification of the programming functions and the abstraction of both data and task management make it accessible to a wider range of developers, not necessarily skilled in low level programming.

The importance of accessibility for domain experts can be appreciated from the many APIs, available in scientifically-focused programming languages for modern programming models and frameworks related to the Big Data paradigm.

This chapter presents a novel approach for adapting low dose CT image reconstruction algorithms to a Big Data framework using accessible programming languages to multi-domain experts without performance degradation. We will focus on a specific programming language (Python) and a specific framework (Apache Spark [77]). Most of the transformations and optimizations applied to the algorithms and presented here can be used in other programming frameworks and languages based on MapReduce models compatible with C libraries and CUDA. The modularization of the solution and the independent intra-scheduler enable an easy integration with other applications or libraries.

We explore the possibilities provided by the Big Data paradigm for both homogeneous and heterogeneous architectures (see Figure 7.1). More concretely, two solutions are proposed: an optimized version for multicore-based environments (homogeneous), and a solution with support for hybrid CPU+GPU environments (heterogeneous).

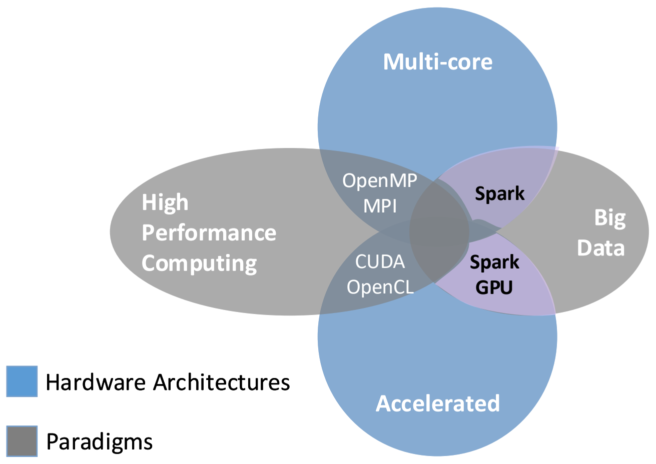


Figure 7.1: *Architectures and paradigms covered in this chapter.*

7.1 Solution for Big Data computing

As described in Figure 7.2, the solution proposed is fundamentally based on the offload the computation of time-expensive components to a distributed environment, being compatible with heterogeneous resources containing CPU and GPU-based compute nodes. This distributed environment can be a private cluster or a public cloud in which heterogeneous resources are available.

7.1.1 Batch processing on Apache Spark

Apache Spark [78] is a general purpose distributed computing framework. It is employed in several fields, although it is mainly used in data analysis and Big Data applications. Spark exposes portable APIs and supports well-known languages (Scala, Java, Python, R) and contains specific libraries focused on machine learning. Spark is based on both an extended MapReduce paradigm and a task-based execution model. This solution is compatible with several resource managers and has connectors for different file systems and distributed databases. One of the main differences with previous frameworks, such as Hadoop [73], is the in-memory data management.

Apache Spark's architecture is composed of two main actors: the *driver* and *workers*. The *driver* is in charge of deploying the application and its management inside the cluster, and communicating with the chosen resource manager. Workers run inside the compute nodes of the cluster and launch the isolated containers in which tasks are executed (*executors*).

Apache Spark follows a lazy execution scheme, having two types of functions depending on whether the real execution of the tasks is triggered or not: transformations and actions. Transformation operations (not to be confused with data transformations) are operations in which new data is created based on previous data and they are not executed immediately. Action operations trigger the execution of the transformation operations returning specific results to the driver. Both types of operations work over an specific data structure called RDDs [78]. These structures hold the data in a distributed and partitioned manner as well as acting as a tool for providing fault tolerance.

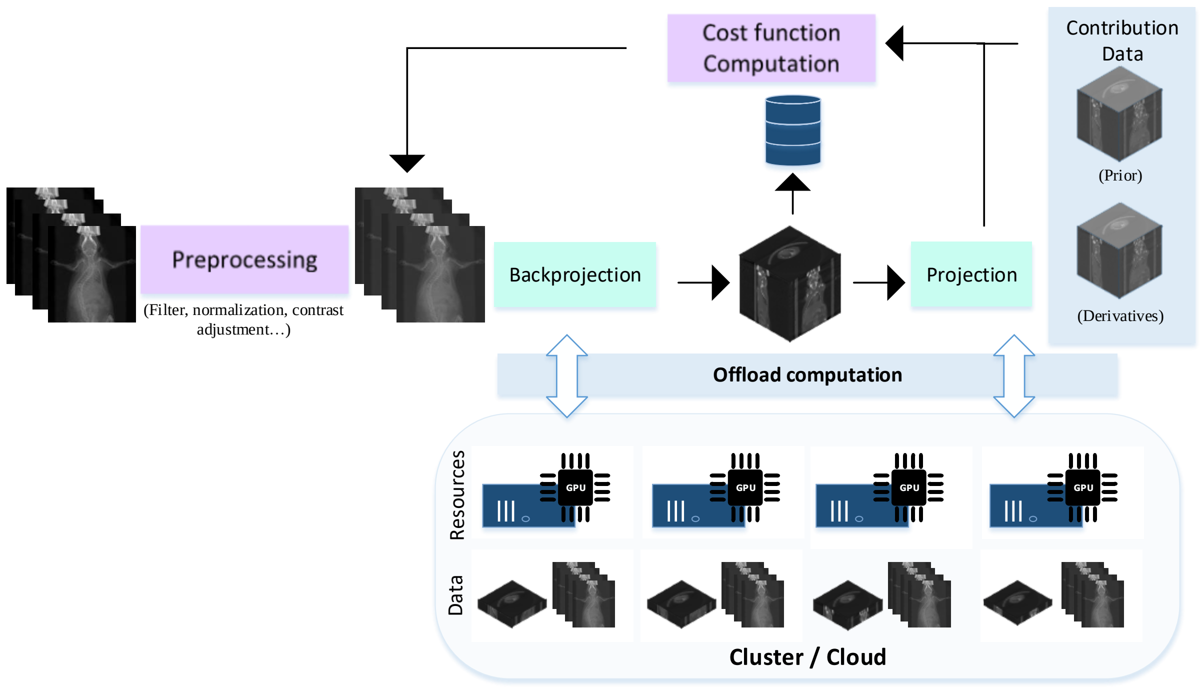


Figure 7.2: *Diagram of a generic iterative reconstruction algorithm with accelerated analytical components: backprojection and projection. Iterative reconstruction algorithms consist of multiple phases in which different contributions are computed. Prior information of the reconstructed object can be included in the form of surface models or previously reconstructed volumes.*

7.1.2 Python and PyCUDA

The selection of Python as programming language is motivated by the fact that it is used as a prototyping language and the quantity and quality of existing scientific modules. These modules provide mathematical functions, highly optimized thanks to their internal development in the C language. However, the process described in this work could have been done with any other compatible language as long as they have bindings for GPUs (either CUDA or OpenCL compatible). In Python there are many alternatives for developing accelerator-based applications. We highlight PyCUDA [172] due to its flexible support of NVidia GPUs and its compatibility with already existing kernels.

7.2 Accelerated architecture for GPU-based Big Data computing

From the application execution point of view, the proposed solution does not modify the components already employed in standard Apache Spark. As it can be seen in Figures 7.3 and 7.4, there are still three main components: the *driver*, the *executor* allocated in the worker nodes, and the *cluster manager*. This architecture is tightly tied with the Apache Spark architecture and the execution model previously presented. In the proposed novel accelerated architecture, a new component is introduced in each of the *worker* nodes, exercising the role of internally managing the available GPU resources, a GPU *intra-scheduler*.

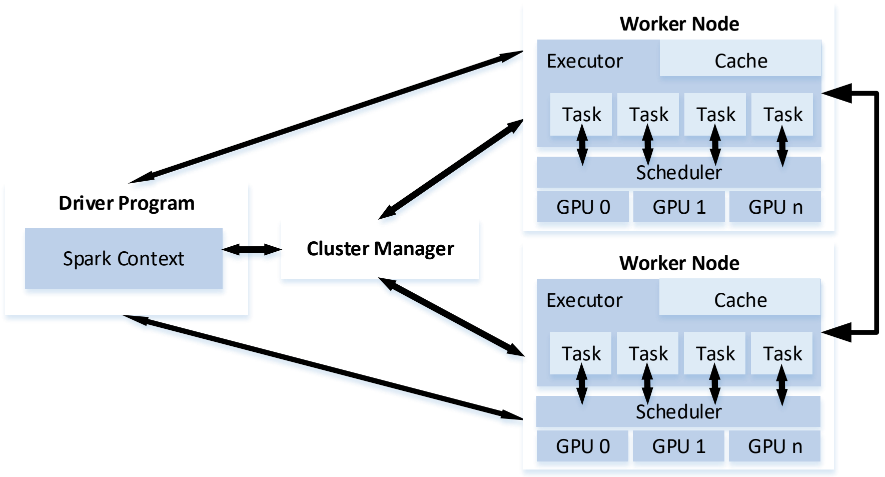


Figure 7.3: *Apache Spark architecture supporting multiple GPU devices.*

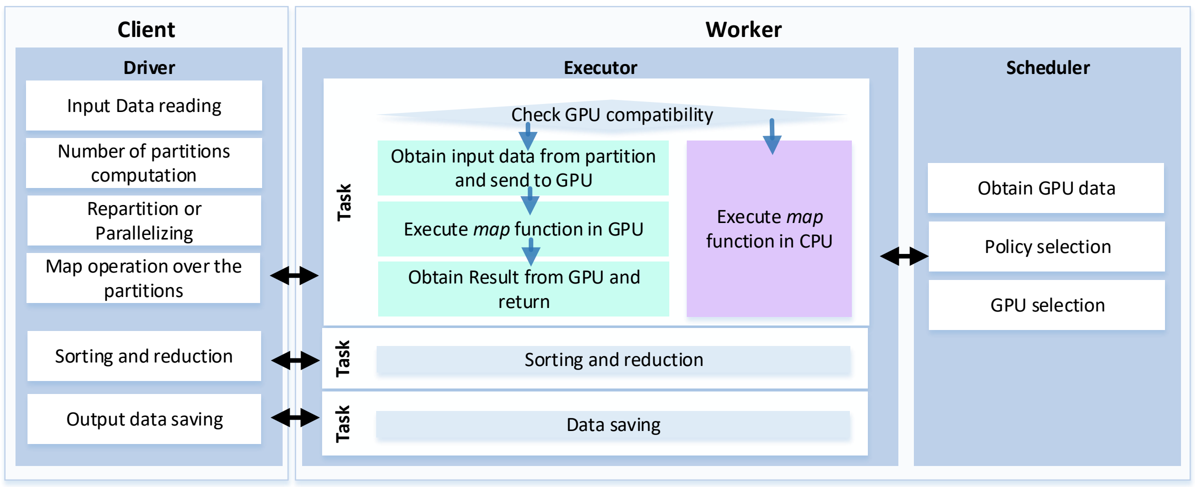

Client
Driver
Input Data reading
Number of partitions computation
Repartition or Parallelizing
Map operation over the partitions
Sorting and reduction
Output data saving
Worker
Executor
Task
Check GPU compatibility
Obtain input data from partition and send to GPU
Execute map function in GPU
Obtain Result from GPU and return
Execute map function in CPU
Task
Sorting and reduction
Task
Data saving
Scheduler
Obtain GPU data
Policy selection
GPU selection

Figure 7.4: *Execution of an application in our accelerated architecture.*

7.2.1 Driver

The *driver* is in charge of the execution of the main code of the application, stating the required transformations and actions. In case of using a local file system, it is also responsible of reading and writing the input and output data. However, if Hadoop Distributed File System (HDFS) is used, each of the worker nodes are in charge of acquiring the nearest data if they are not already available. The node in which the *driver* is allocated is normally independent from the nodes on which tasks are executed (nodes with *workers*). In these cases, GPUs are not required inside the *driver* node, since tasks are executed independently from the main program.

7.2.2 Executor

In the *executor*, tasks are finally computed. These tasks can invoke CUDA-based kernels through the PyCUDA interface. As stated in Figure 7.4, first we need to check out if the node counts with the required accelerators, in our case, NVidia GPUs. This check also includes the presence of the auxiliary tools for GPU programming, either CUDA or OpenCL libraries. After that, it will request a device to the *intra-scheduler*. In case of not counting with enough resources, the scheduler will block the assignment process. Then the task will proceed to the online compilation and preparation of the arguments that will be passed to the GPU kernel. Finally, output data from the GPU will be transferred to main memory and returned as the result of the mapping task.

7.2.3 Many-task GPU intra-scheduler

Nowadays, many clusters and supercomputers include accelerator devices, like GPUs, being also commonly used in many cloud providers (i.e. Amazon EC2). However, Apache Spark lacks of native support for exploiting the accelerators capacity, although some alternatives have appeared [92, 173].

The *GPU intra-scheduler*[1] consists in an independent service, which is responsible of scheduling and selecting the most suitable GPUs in which the kernels will be deployed. In this way, the corresponding tasks that are executed in the same node obtain an available GPU. Due to its characteristic of independent service, the *intra-scheduler* is capable of scheduling the GPUs even between independent applications that are executed inside the same node.

With our proposed approach, Apache Spark can be used without any modification. The main contribution to the architecture is the inclusion of a *intra-scheduler* that will be executed in each compute node. This *intra-scheduler* orchestrates the different *executors* located inside the node, offloading computation into the accelerator devices. Using different policies, the *intra-scheduler* decides which tasks are executed in the GPUs, as well as in which devices are executed considering several variables:number of devices that are installed inside the node, the memory needs of the task to be executed, memory available at each device, number of tasks executed in each devices, etc. If no accelerator is available (due to the limited memory), it stalls the execution of the task or sends it to the CPU if possible, until there is enough memory space available.

GPU assignment policies

This component has different policies that can be applied to the scheduling decisions, depending on factors such as available resources in the device, offered performance, number of tasks in execution, etc. There are already three policies implemented:

- *Round Robin*: Tasks are assigned to GPUs in incremental order.
- *Random*: Tasks are assigned randomly to a GPU without taking into account any specific variable.
- *Least processes*: Considering all the tasks that are already been executed in the different GPUs, the scheduler assigns the one with the least number of tasks being executed.

However, due to the memory limitations of these devices, if the device chosen by the policy does not have enough memory to execute the kernel of a specific task, the next GPU (in numerical order) will be chosen. If there is no GPU with enough memory, then the scheduler stops the

[1] Available at https://github.com/Estefania/py_gpu_scheduler

execution of the task waiting until the required resources are available. Using this mechanism we ensure that all kernels from all tasks can be executed without memory problems in the devices.

The management of the device, which has a separate memory space, is performed inside the Spark *task* and the implementation of the function to be executed can be made using CUDA native kernels or using PyCUDA methods.

Execution of a mapping task

The execution of the mapping in the heterogeneous architecture consists of five phases:

1.- Device acquisition: the *executor* communicates with the scheduler to obtain a device in which the function should be executed. Based on the device obtained, the context for that specific accelerator is created.

2.- Transfer of the input data onto the device memory: using the PyCUDA API, it is possible to transfer the memory containing the input data to the acquired device using the CUDA drivers.

3.- Execution of the CUDA/PyCUDA kernel: the kernel is loaded, on-demand compiled, and executed with the required parameters passed to the function.

4.- Transfer of the output data to the host memory: to be able to obtain the final data, each mapper should return its output stored in the device memory to host memory for Apache Spark to be able to manage it. The output data generated in the device is transferred to the main memory before finishing the task.

5.- Device release: the *executor* requests the scheduler to release the device. The context for that device is destroyed.

The communication with the *intra-scheduler*, provided through RPCs, is implemented with the RPyC package [174].

7.2.4 Multi-core CPU execution architecture

In case no accelerator devices are present in the system, two options are proposed in order to accelerate the execution:

- To exploit the parallelism already provided by the platform. This option is the easiest since it is already incorporated in Apache Spark. However, to fully exploit the resources previous tuning and modification of diverse parameters of the platform is required.
- To incorporate additional parallelism by using native programming models, widely employed in clusters and supercomputers, like OpenMP. This option includes an additional level of parallelism. This approach increases the programming difficulty, having to incorporate HPC programming models into the Big Data platform.

For both alternatives, we use C as native programming language, due to its higher performance over interpreted programming languages (i.e., Java, Scala or Python). Since Apache

Spark lacks of support for native programming languages, we take advantage of the close relation between Python and C, which allows to invoke specialized functions implemented in this language using the provided Python headers.

To better support this decision, we include in Table 7.1 a summary of the execution time of the backprojection component in a single-node configuration using different programming languages and models. Although the difference between using C with OpenMP and Python invoking C with OpenMP represents almost doubling the execution time (due to the use of an interpreted language as a launcher), the main gap is produced when Python is used without any other parallel programming model, obtaining execution times larger than one day. Therefore, the use of non-interpreted programming languages is strongly encouraged.

Table 7.1: *Average execution time in seconds for different versions of the backprojection component.*

Environment	**Time (seconds)**
C (OpenMP-16 cores)	260.79
Python (NumPy)	$>$ 86,400.00
Python (C)	6,124.47
Python (C OpenMP-16 cores)	439.48
PySpark (C OpenMP-16 cores)	479.35

The architecture of the framework (Figure 7.3) as well as the execution flow is the same regardless of the aforementioned alternatives. However, there are differences regarding the execution setup:

- Apache Spark-based parallelism: this alternative is the most widely used in the community. It takes advantage of the possibility of using more *executors* per node (over subscription). In general, the best option to fully exploit the CPU hardware and being able to execute multiple tasks from multiple applications, is to spawn one *executor* per core. However, since *executors* are launched independently with a decoupled memory space, there would be a need of larger resources on *worker* nodes (see Figure 7.3). When configuring the execution, we balance the number of *executors* per node and the cores per *executor*, complying with the trade-off between memory usage and resources exploitation. This is possible thanks to the possibility of deploying multiple cores per *executor*, thus reducing the number of *executors* per node and the memory needed.

- Native-based parallelism: In this case, to parallelize the algorithm, we rely on OpenMP for our backprojection and projection components, in which we parallelize each ray with a different CPU thread. Memory footprint also increases, as in the previous alternative. Nevertheless, in this case the number of *executors* needed is lower, resulting in a reduction of the memory requirements. This approach consists in a two-level parallel strategy managed by two different techniques: coarse-grained parallelism at a distributed level using Apache Spark (external parallelism) and fine-grained parallelism inside the node using OpenMP (internal parallelism).

In both cases, an invocation of a C module is introduced inside each task. This C module contains the proper *map* function in which each element is transformed, returning the corresponding projection or volume. Data are not copied between Python and C, since the reference is valid for both languages, saving memory and execution time. Python tasks are therefore only responsible of setting the parameters of the C module. Inside the C module, Python parameters are transformed into C variables and the algorithm is executed with or without OpenMP depending on what alternative is chosen. After the algorithm is executed, the resulting data is returned to Python with the correct format. In this case, an array containing the corresponding subvolume or projections is returned.

7.3 Redesigning backprojection and projection components to work on a MapReduce programming model

7.3.1 Backprojection

Inside the backprojection component, the *map* stage corresponds to the computation of the geometry of each independent ray. Since there is only one ray per voxel, we have $dimu \times dimv \times dimz$ *map* operations corresponding to the dimensions of the volume in the three different axes. The transformation function is defined as:

$$vol(u,v,z) \leftarrow \sum_{\theta=ini}^{ini+nproj} proj_{\theta}(x,y) \tag{7.1}$$

where each point in $proj$ is obtained using Equation 3.4.

The volume is divided into subvolumes (partitions) along the z axis. As we can see in Figure 7.5, backprojection task is executed per partition. This division is represented inside the framework as an ordered array with the position of each partition inside the complete volume. Thus, partitioning is performed at the output (volume), not at the input (projections). This implies that input data must be transferred to all worker nodes through a *broadcast* operation inside Apache Spark.

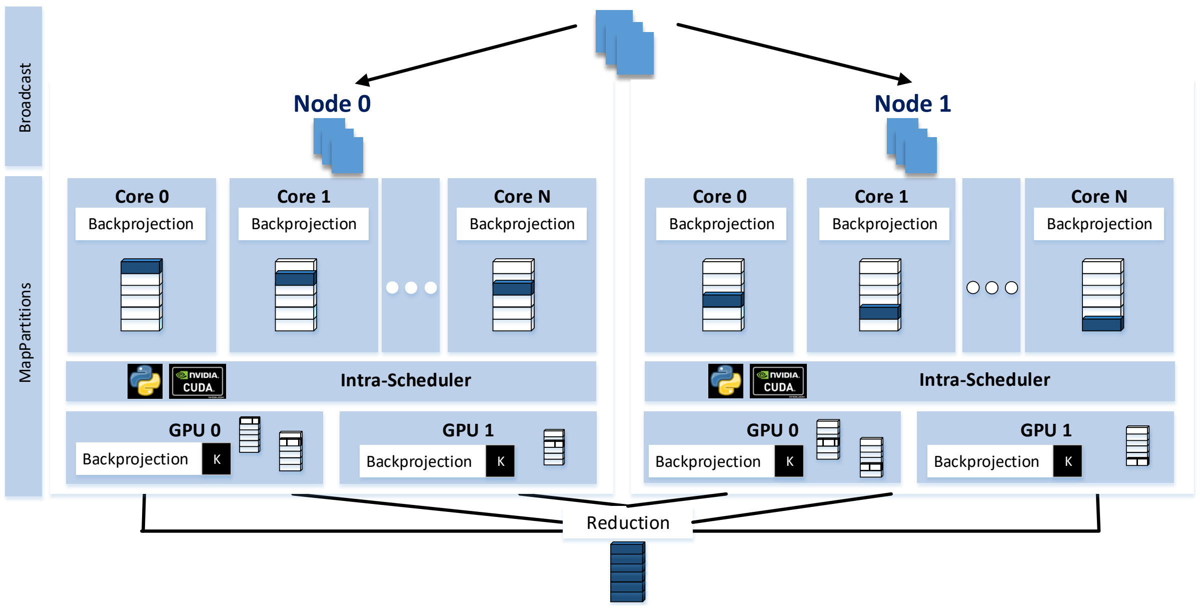


Figure 7.5: *Execution and partitioning of the Backprojection component in our proposed schema.*

7.3.2 Projection

In this case, the partitioning of the computation and data is also based on dividing the volume into smaller subvolumes, which are now the input data of the *map* operations. Therefore, in the projection task there is no need of broadcasting data, since each task already has his partition thanks to the functionality provided by Apache Spark RDDs. The number of *map* operations is equal to $dimx \times dimy \times nproj$ and the transformation function is described as:

$$proj(x, y) \leftarrow \sum_{v=0}^{depth} vol(u, v, z) \tag{7.2}$$

where the points in *vol* that contribute to each point in *proj* are obtained following Equation 3.4.

As shown in Figure 7.6, every core computes all projections from the corresponding subvolume. To reduce the computational load for each subvolume, the projection task only computes the specific rows of the projection data corresponding to each subvolume. Each row is identified by a key computed based on the projection angle and its spatial position inside the complete projection. Due to the cone-beam geometry, multiple subvolumes can contribute to the same row of the projection, thus having tuples containing the same key.

Then, when every projection task has been completed, a *reduction* of all the projection rows is executed over the tuple (key and its corresponding projection row).

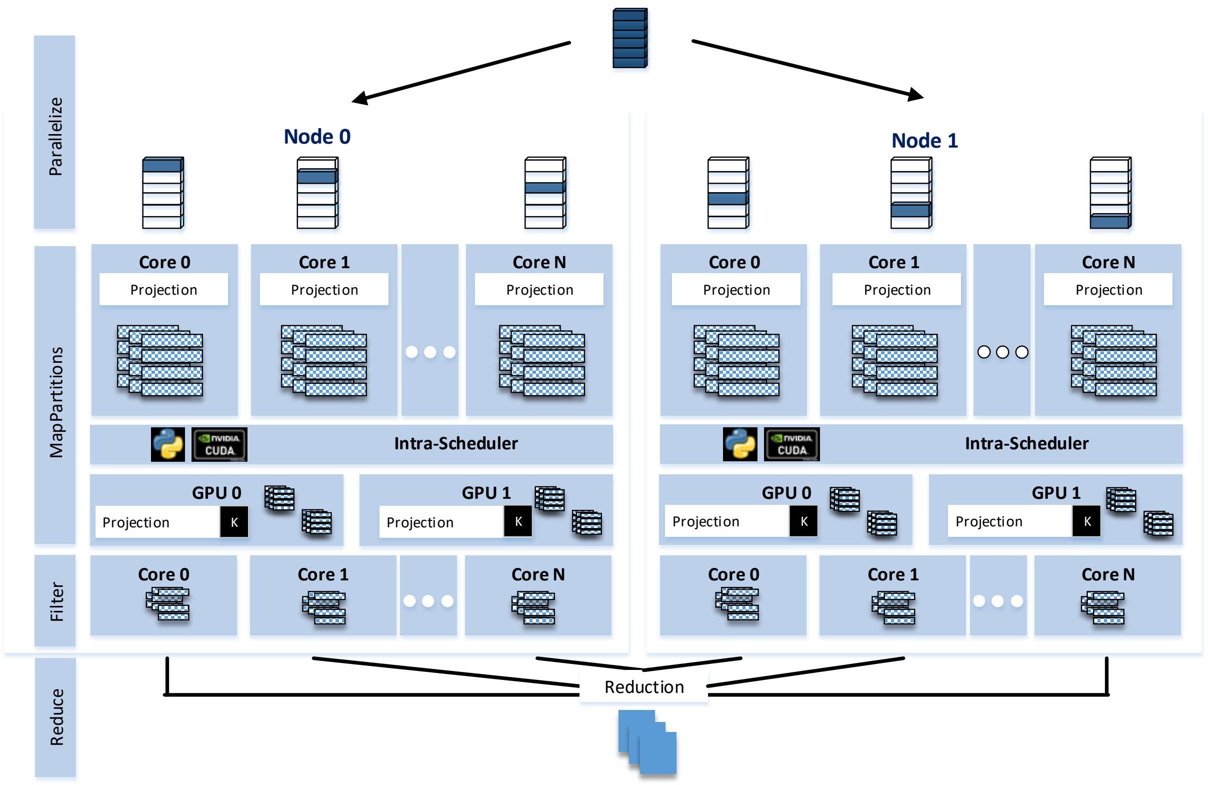


Figure 7.6: *Execution and partitioning of the Projection component in our proposed schema.*

Reduce stage does not require additional functions due to the native implementation of element-wise additions in Numpy arrays. Although this approach requires to have a temporal copy of all projections in each task and it may seem memory expensive, it is important to highlight that low-dose CT techniques normally imply the use of a lower number of projections. Therefore, this replication does not create a high overhead in terms of memory usage. For a large number of projections, this reduction step could represent a bottleneck in the execution of the application, nevertheless, in those cases the usage of other type of algorithms, mainly non-iterative, is advisable.

7.4 Evaluation of the proposed accelerated architecture

The evaluation has been carried out in a compute node composed of two processors Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz and 3 GPUs, 1 Tesla K40c (12 GB of memory GDDR5) and 2 Geforce GTX Titan (6GB GDDR5 memory). In the experiments where the memory is defined to be limited, the memory of each card is reduced to 2 GB each. This limitation we can better determine how each policy work under different hardware.

The complete system is supervised through Cloudera 5.7. The version of Spark employed is 1.6 in *stand-alone* mode. The input files are stored in a local SSD and the result files are saved into an HDFS directory, also in SSDs and with a 10 Gbps Ethernet network. The Python version employed was 2.7 and we used PyCUDA 1.3. The input data for the experiments consisted in 360 projections with 1024^2 pixels (1.2 GB). The size of the output data was 1024^3 voxels (4 GB). In each experiment, we show the average of at least 3 different repetitions as well as the standard error. In the case of the occupancy timelines, data of only one repetition is shown. In this case, only the backprojection component for GPU was evaluated.

7.4.1 Overall execution time

In Figure 7.7, we plot the total execution times for the baseline configuration (labeled as simulator) and the Apache Spark version (labeled as Spark) with 3 threads and 3 partitions.

Due to the overhead that implies the execution of an associated runtime (Apache Spark), the execution times for a standard volume of 1024^3 voxels of the distributed backprojection application in a node are not as competitive compared with the execution time of the simulator in the case of the same node and number of GPUs. In all cases Spark requires more time to produce the result although, when the number of GPUs is increased, the difference is reduced. This is due to the better exploitation of the resources thanks to the scheduler.

In Figure 7.8, we show the execution times of the Apache Spark approach for 3 threads and a variable number of partitions. We evaluate the three policies implemented applying a limited memory scheme on the GPUs. From these results, we can conclude that for the case of only one node, the increment in the number of partitions impacts negatively in the overall execution time due to the increment of the memory usage. It is important to note that incrementing the number of partitions does not increment the level of parallelism if the number of executors and threads does not increase. With respect to the differences between policies, only *Random* obtains a significantly higher execution time than the other policies, being *Round Robin* and *Least Processes* in the same time range. However, if we look at results from Figure 7.9 when we average the execution times for every number of threads and different number of partitions, in general, *Least processes* performs slightly better than *Round Robin.*

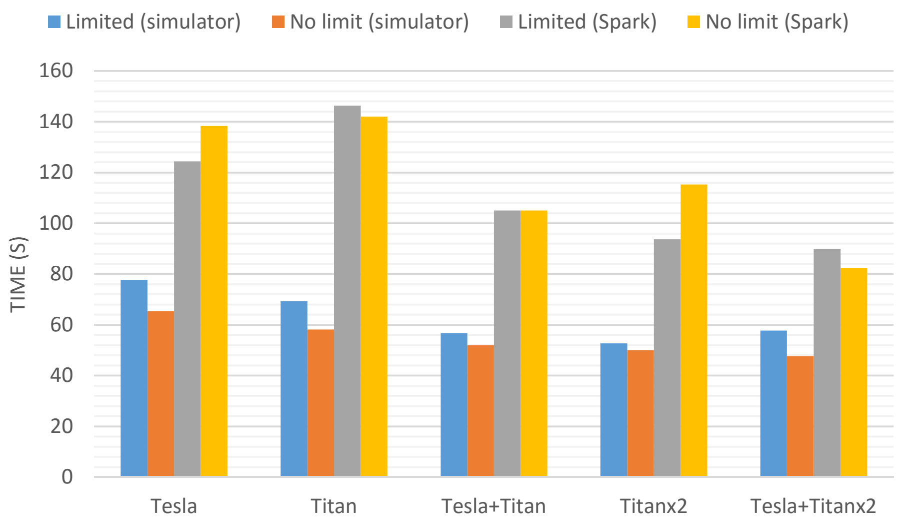


Figure 7.7: *Execution time for the different combinations of GPUs in a node, with memory limitation, no limitation, in the original setup (simulator) and in its Spark version (Spark).*

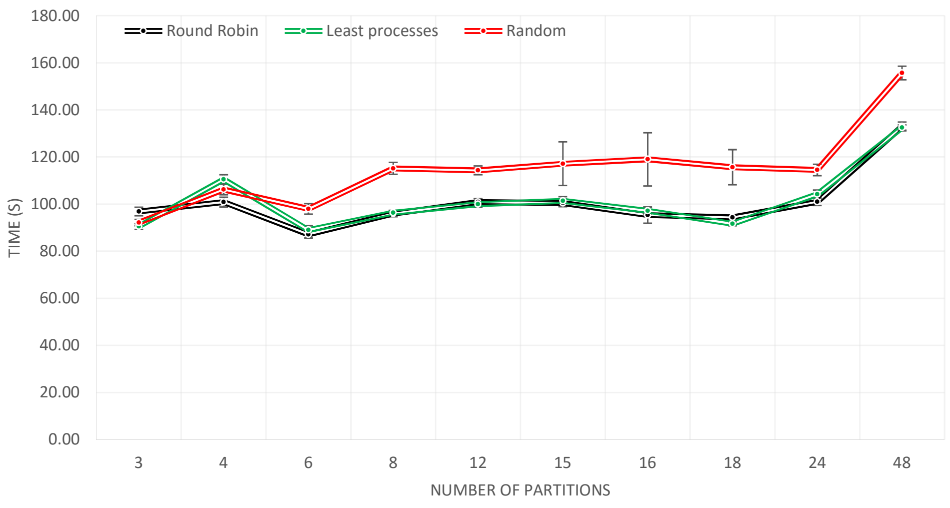


Figure 7.8: *Results of the evaluation of the application with 3 threads and different number of partitions for each of the policies and their corresponding standard error.*

7.4.2 Occupancy of the GPUs with different policies

As described before, the objective of the proposed *scheduler* is the exploitation of multiple GPUs that can be attached to the compute nodes. In Figure 7.10, we plot the execution time of the application based on 12 threads and 48 partitions, on the three available GPUs for the three implemented policies. This configuration maximizes the parallelism in the node. In this experiment, to evaluate the occupancy of the GPUs with the different policies, we have simulated a more homogeneous environment by limiting the memory available on each device to 2 GB.

The policy that takes less time to finish is *Least Processes*, as we also concluded in the previous section. *Least Processes* exploits the GPUs in a more regular manner. Both *Round Robin* and *Random* policies possess several spikes in the occupancy meanwhile with the first policy the occupancy is held around 12 tasks assigned to GPUs, which maximizes the maximum

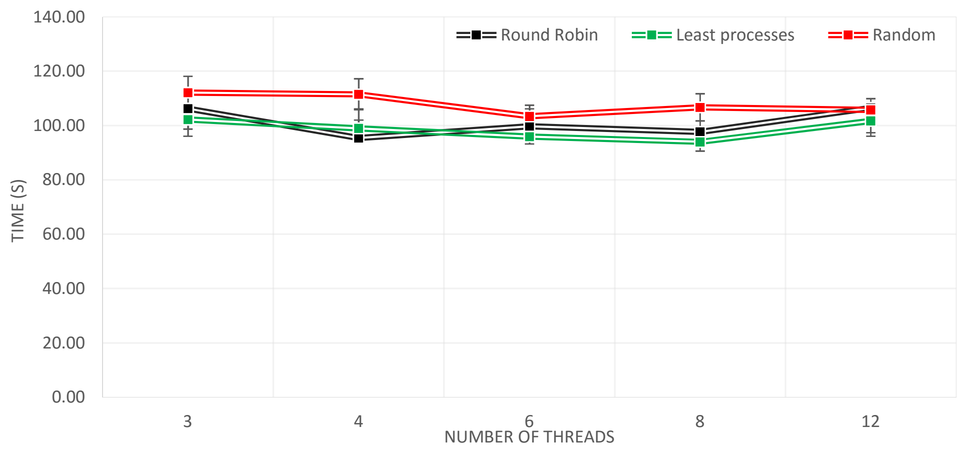


Figure 7.9: *Results of the evaluation of each of the policies with different number of threads. The execution time is the average of the execution times for each number of partitions.*

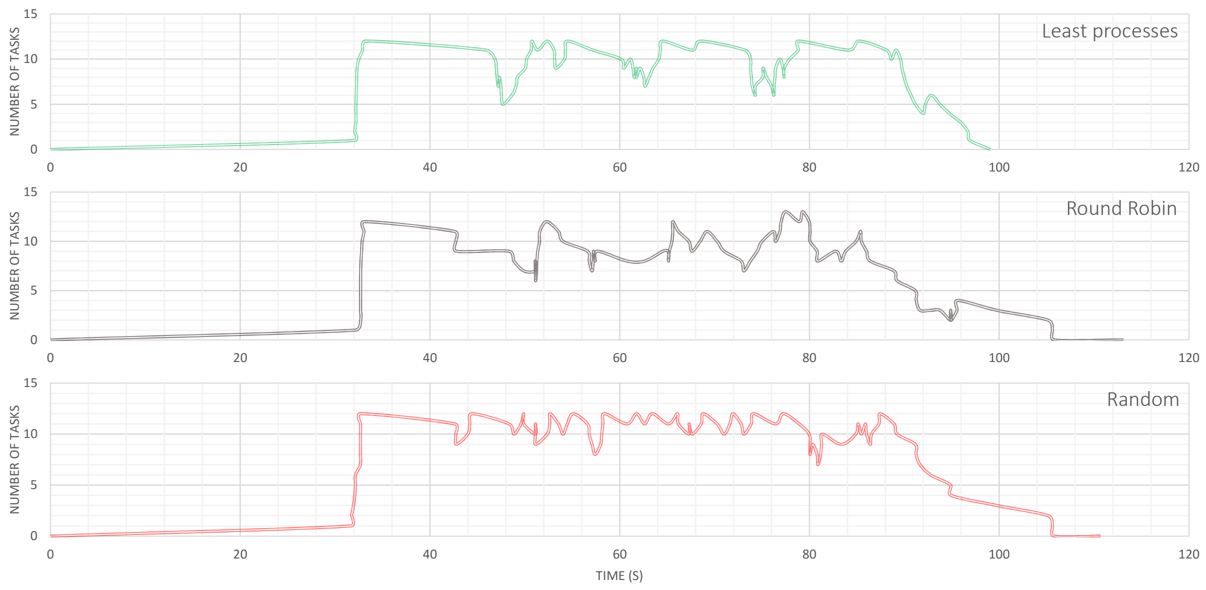


Figure 7.10: *Timeline of the experiment with 12 threads and 48 partitions for each of the policies. We show the number of tasks assigned to the GPUs in each moment.*

number of threads running in parallel. The reason for this is that *Least Processes* is capable of balancing the inequalities in the execution of the tasks. In this case, a more exhaustive analysis discovers that the first GPU (Tesla) takes around 50% more time to process tasks than the others solutions. *Least Processes* detects that the number of tasks in that GPU is higher and sends the tasks to the other ones which are faster and less overloaded. However, in all three policies, we can appreciate regular spikes that are attributed to the finalization of the tasks assigned to the thread. Although the partitioning of the data and therefore of the computational load does not have to be regular, the tasks are similar in size and execution time. Then, it is reasonable that all tasks executed in parallel finish in similar times, reducing the number of tasks executed on the device. Since we have 48 partitions that equal 48 tasks, each thread will be assigned 4 tasks that will be executed sequentially. When a task is finished and before the next tasks is assigned to a device, the GPUs become unused because the mapping functions have not started yet, resulting in the performance spikes.

7.4.3 Evaluation with multiple nodes

For the evaluation of a larger volume we chose to execute the application over a set of nodes ranging from three to five, all of similar characteristics to the one employed in the first experiments. The resource manager used was Apache Yarn. All the nodes possessed at least one GPU. The input data for this evaluation was a set of 360 projections of 2048x2048 pixels (5.7 GB) to obtain a resulting volume of 2048x2048x2048 voxels (32 GB).

Table 7.2 depicts some of the results for a different number of partitions, nodes, threads, and executors. As a reference, the time needed for a backprojection of that size in a Tesla K40c, a high end graphics card, is on average 7m33s. If we compare with the best result in Apache Spark (7m36s), we can observe that both of them require the same execution time.

Table 7.2: *Execution of the Spark application over different nodes for a 2048x2048x2048 volume of result.*

Nodes	Executors	Threads per executor	Partitions	Time
5	5	2	120	**7m36.000s**
3	3	2	60	9m51.756s
3	3	4	60	7m42.368s
3	3	5	40	7m38.091s

7.5 Scalability evaluation

In this section, we show the experiments carried out to evaluate the scalability of the implemented solution for the offloaded components using Python and Apache Spark, in both GPU and multi-core CPU-based architectures.

7.5.1 Experimental setup

We evaluated our solution using two environment setups:

- Distributed multi-core CPU cluster: 8 nodes with Intel(R) Xeon(R) CPU E5-2603 v4 (12 cores), 126 GB of DDR4 RAM, and 256 GB of SSD for scratch storage. The Apache Spark driver was launched in a separated node with two Intel(R) Xeon(R) CPU E5-2630 v3 processors (12 cores) and 252 GB of RAM.
- Distributed GPU-based cluster: 2 nodes with an Intel(R) Xeon Phi(TM) CPU 7210, 148 GB of RAM, 256 GB of SSD for scratch storage. Each node has 2 NVidia GTX 1070 installed, each with 8GB of GDDR5 memory.

The system is supervised through Cloudera 5.13 over Ubuntu 16.04. The version of Apache Spark employed is 2.2 in *stand-alone* mode. For the distributed evaluation, Apache Yarn 2.6, with the *FairScheduler*, was used as resource manager. The input files are stored in a local SSD and the result files are stored into a HDFS, running on top of SSDs disks and a 10 Gbps Ethernet network. The Python version was 2.7, complemented with PyCUDA 1.3 for the GPU-based architecture and a C module compiled with GCC 5.1 and OpenMP for the homogeneous architecture. Taking into account the results from the previous section, the policy for the *GPU intra-scheduler* was set at *Least Processes*.

For both components, we have worked with volume images of 1024^3 voxels and 1024^2x360 projections. This number of projections is higher than the one normally acquired for low doses purposes, therefore this experiment represents a worst case in performance. Input data is read from a SSD in the driver program, requiring first a parallelization of the data.

To better evaluate all possibilities we have executed the components in both architectures with two configurations:

- *Configuration A*: single core per *executor*. Parallelization is done internally either by using OpenMP in the CPU based approach or with the GPU cores.
- *Configuration B*: 5 cores per *executor*. Parallelization in the multi-core CPU approach is done at external level without further usage of other parallel programming models. In the case of GPU-based execution, additional parallelization is introduced with the usage of the multiple GPU cores.

7.5.2 Multi-core CPU architecture evaluation

We executed both backprojection and projection components in a multi-core CPU cluster with both configurations. In *configuration B*, we will be able to execute 5 tasks at the same time per *executor*. Those *executors* can be placed inside the same node if it has enough memory capacity. In *configuration A*, each *executor* will execute one task (coarse-grained) and each task is accelerated by using OpenMP.

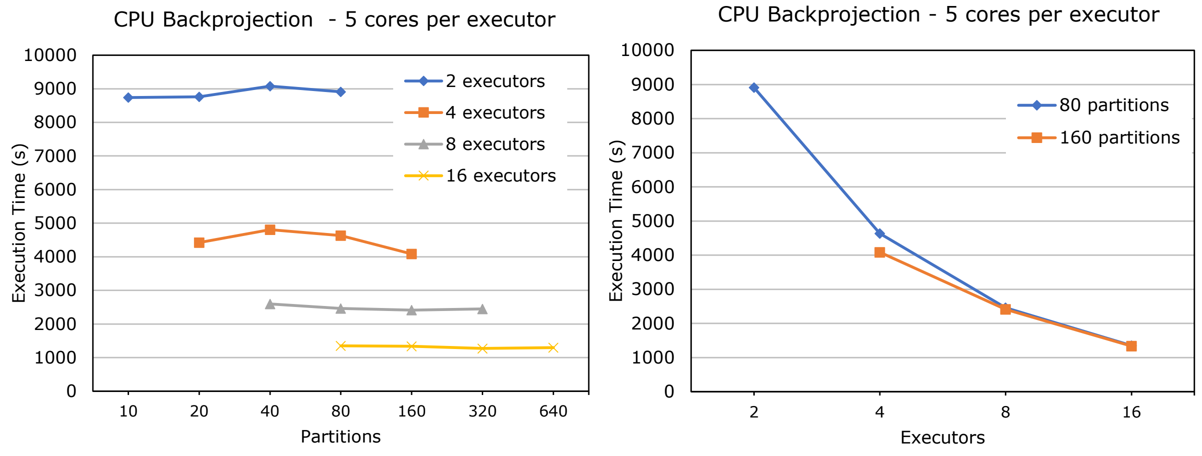


Figure 7.11: *Configuration B: Execution time of the backprojection component on a distributed multi-core CPU cluster for a different number of executors and partitions without using OpenMP.*

In Figure 7.11, we plot the evaluation results employing *configuration B* inside the backprojection component. We explore the results given a varying number of *executors* and partitions.As we increase the number of *executors*, we note the benefits of having a number of tasks larger than the number of *executor* threads. This is due to the better management of the imbalances between nodes when smaller tasks are scheduled. This effect can also be appreciated when employing OpenMP, as shown in Figure 7.12.

In the case of the projection component, the obtained results for *configuration B* depict a significant increase when the number of partitions are incremented (as shown in Figure 7.13).

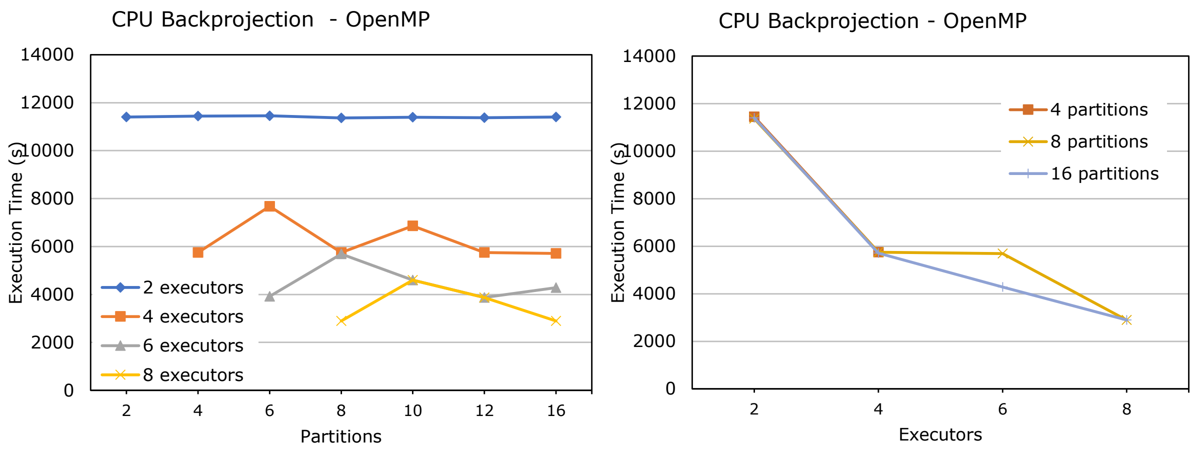


Figure 7.12: *Configuration A: Execution time of the backprojection component on a distributed multi-core CPU cluster for a different number of executors and partitions using OpenMP.*

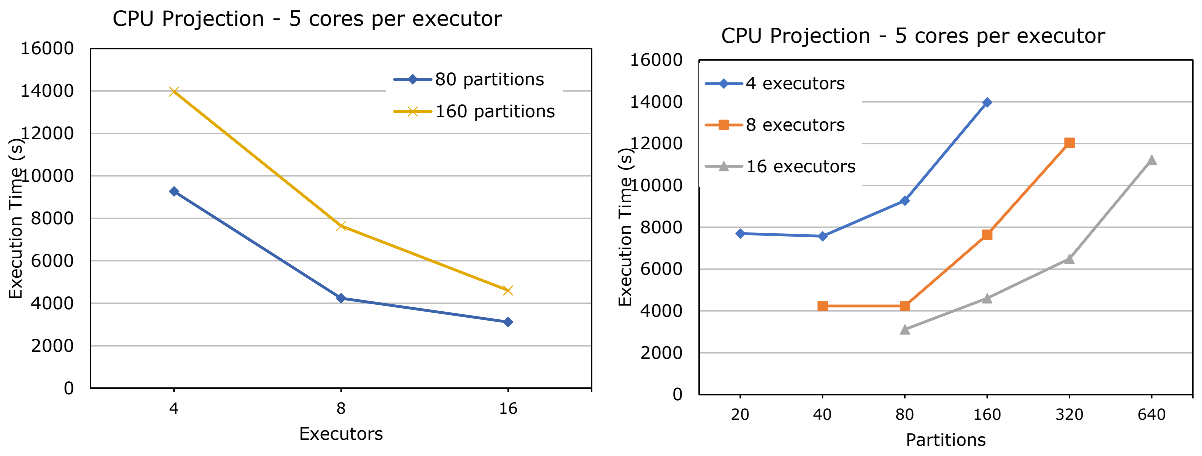


Figure 7.13: *Configuration B: Execution time of the projection component on a distributed multi-core CPU cluster for a different number of executors and partitions using 5 cores per executor.*

In this case, a higher number of partitions does not decrease the computational size of the task due to the discrepancies between the number of slices in a volume and the FOV projected. As shown in Chapter 6, the area computed in projection from a chunk is larger than the number of slices in the chunk due to the cone-beam geomtry. Because of this reason, the increment of parallelism using Apache Spark can harm the performance of this component. Nevertheless, it is important to note that the time does not increase at the same rate as the number of partitions.

The projection performs better in *configuration A* than in *configuration B*, in contrast with what happens with the *backprojection*, as plotted in Figure 7.14 and Figure 7.13. However, in *configuration A* the increment in the number of partitions does not positively affect the overall performance of the application, similarly to what we have seen in the case of the backprojection component. Considering the overall execution time, the number of partitions required when using OpenMP is lower than the one required in *configuration B*.

Considering the overall execution time, the number of partitions required when using OpenMP is lower than the one required in *configuration B*. The reason behind this is that the parallelism is applied at coarse-grained inside the partition and not per partition as in the first configuration,

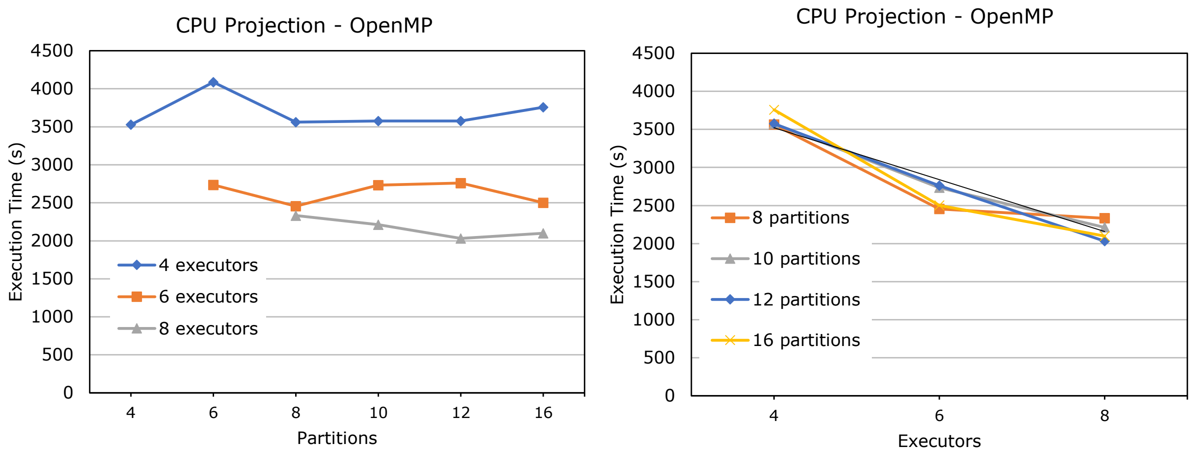


Figure 7.14: *Configuration A: Execution time of the projection component on a distributed multi-core CPU cluster for a different number of executors and partitions using OpenMP in each executor.*

so, to fully exploit the hardware we do not need to increase the external parallelism. Moreover, OpenMP internal parallelism can be favoured by the higher amount of work in bigger partitions, a factor that is penalized in the other configuration.

7.5.3 GPU-based architecture evaluation

We have also evaluated the performance of the Apache Spark/GPU-based architecture based on our proposed *intra-scheduler*, varying the number of *executors* and partitions. Given that the maximum number of GPUs present in the system is four, we evaluated up to four parallel executors over different number of cores. In practice, increasing the number of cores per executor allows to exploit the available concurrency, given that modern NVidia GPUs enables the execution of multiple concurrent kernels in case of having enough resources (mainly, computing units and internal memory).

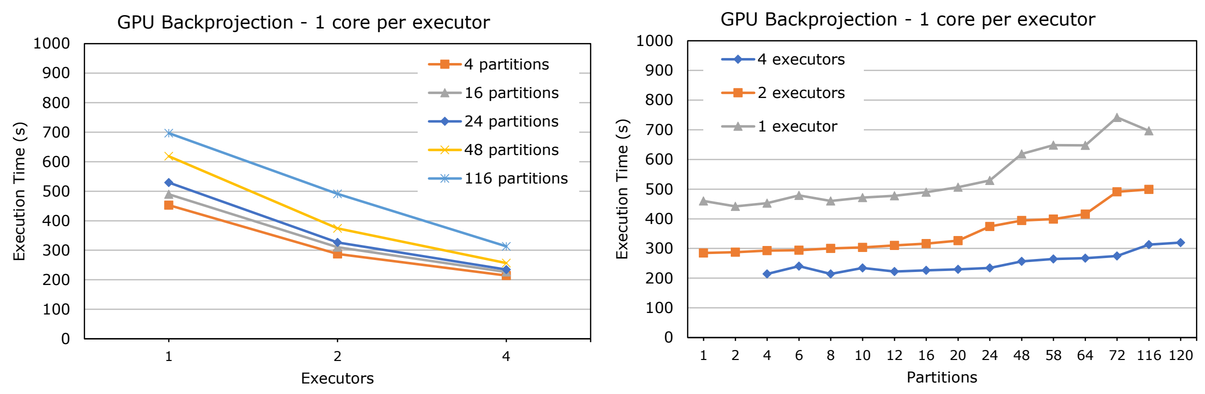


Figure 7.15: *Configuration A: Execution time of the backprojection component on a distributed GPU-based cluster for a different number of executors and partitions.*

Figures 7.15 and 7.16 plot the evaluation of the backprojection component in the GPU-based architecture. For both configurations we observe that from 100 partitions (4 *executors*), the overall execution time increases for both cases. Execution times do not differ between configurations when executing with similar conditions (around 200 seconds for one partition per thread and 4

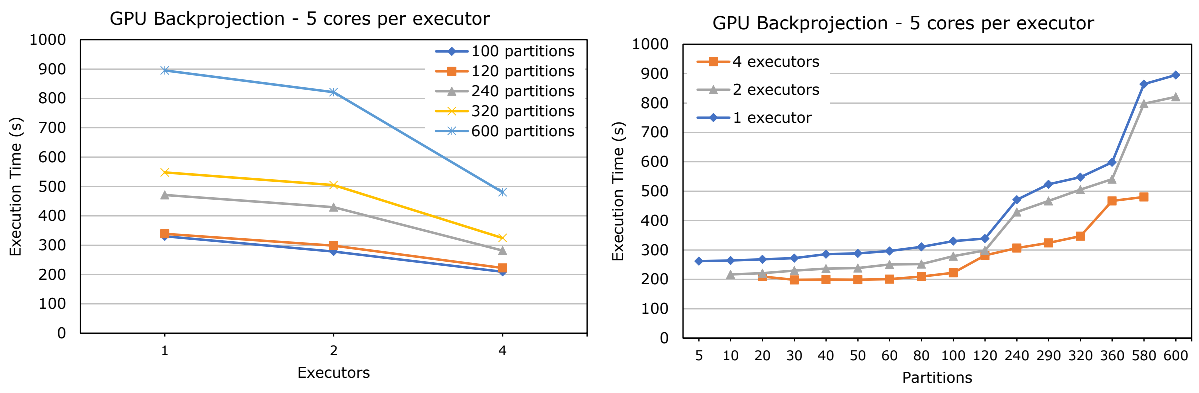


Figure 7.16: *Configuration B: Execution time of the backprojection component on a distributed GPU-based cluster for a different number of executors and partitions with 5 cores per executor.*

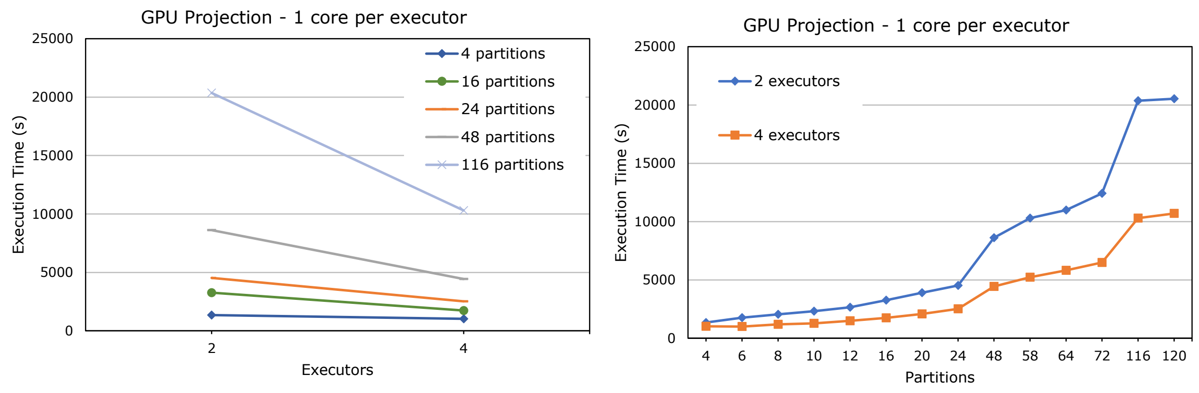


Figure 7.17: *Configuration A: Execution time of the projection component on a distributed GPU-based cluster for a different number of executors and partitions.*

executors and also around 200 seconds for one partition per executor and 4 *executors*), which indicates that the limiting factor is the computing power of the GPU device.

Otherwise, in case of projection, Figures 7.17 and 7.18 show that there is a significant difference between both configurations in the GPU based approach. Executing the algorithm with 4 *executors* employing *configuration A* results in an execution time of around 1000s, meanwhile with 4 *executors* and one partition per core in *configuration B* the algorithm takes around 700s. The number of tasks in the second case is higher, 20 tasks for 4 *executors* vs 4 tasks, which should be counterproductive for the projection algorithm as explained before. However, the number of tasks from which we can perceive a significant increase in time is 40 as seen in Figure 7.13. Being 20 tasks a lower number than this threshold, and considering the benefits of executing smaller tasks for a better load balancing between the GPUs, it is reasonable that *Configuration B* performs better than *Configuration A* in this case.

Figure 7.19 shows the timeline for the execution with 4 *executors* for *Configuration A* and *B*. The larger execution time in task 3 in *Configuration A* causes the whole execution to spend two minutes more. In *Configuration B* however, although there are imbalances, since the tasks are smaller, they only represent less than one minute of increased execution time. The drawback detected in the multi-core CPU approach is also reproduced when employing GPUs with a much higher number of tasks. The decrease in performance starts with a lower number of partitions than in the case of the backprojection, since in projection a greater number of partitions also

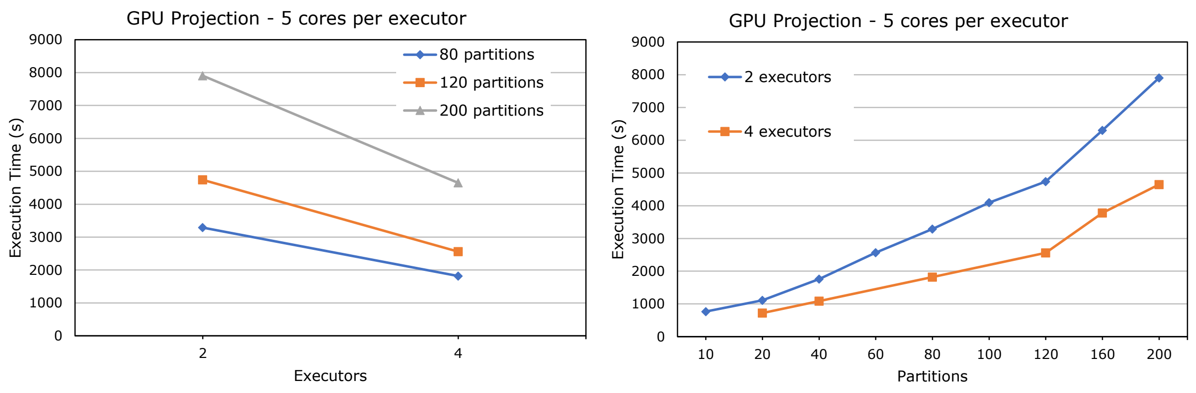


Figure 7.18: *Configuration B: Execution time of the projection component on a distributed GPU-based cluster for a different number of executors and partitions with 5 cores per executor.*

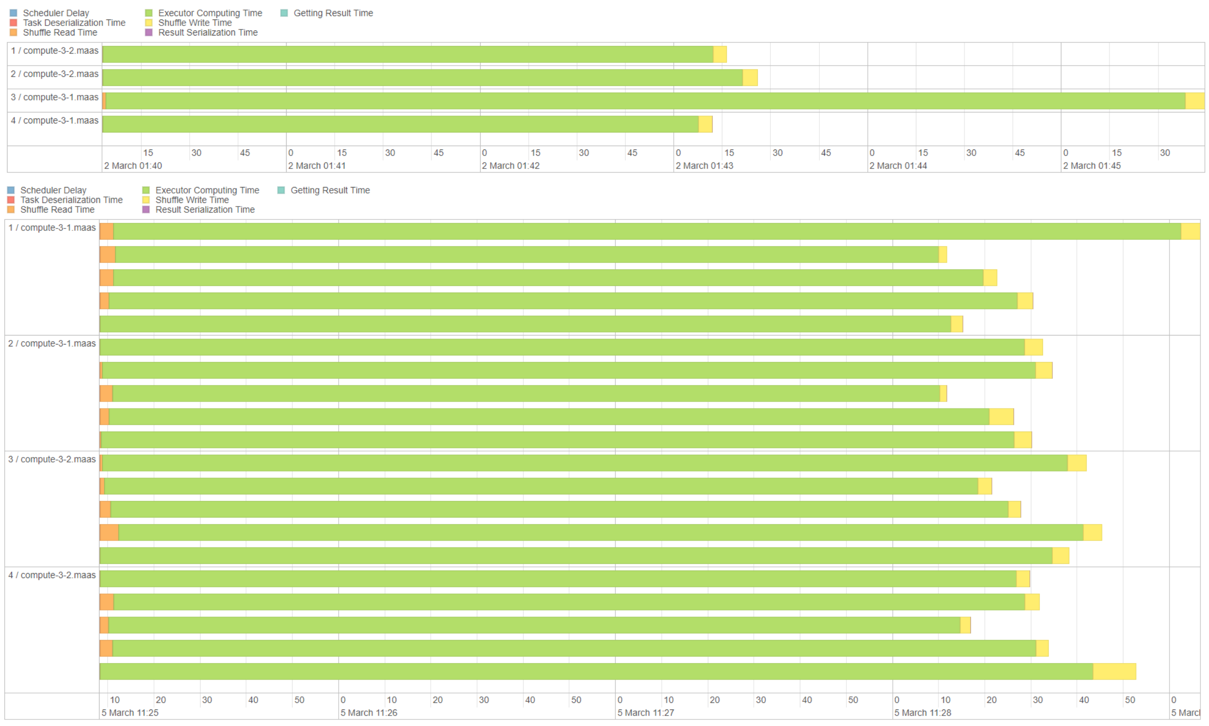


Figure 7.19: *Timeline for the projection component on a distributed GPU-based cluster for Configuration A, 4 executors with one core (top) and Configuration B, 4 executors with 5 cores (down).*

increases the amount of work that is executed. For the same amount of work (same number of partitions) both configurations scale out almost perfectly when increasing the number of *executors*.

7.5.4 Image quality

We have evaluated the effect of the partitioning on the final results using simulations with the Digimouse phantom[2]. We performed a backprojection followed by a projection with 1, 8 24 and

[2]http://neuroimage.usc.edu/neuro/Digimouse

48 partitions, with 360 projections of 512^2 pixels, resulting in a volume of 512^3 voxels. The Root Mean Square Error (RMSE) in the projections with respect to the 1 partition case is significantly low (under $1.0e-4$) for all partition sizes. The small errors, not noticeable with visual inspection, appear in the intersection between the chunks due to the reduction stage in projection, as shown in Figure 7.20 (the difference images are shown with a narrow window so the small errors are noticeable). No effect was observed for the *backprojection* step.

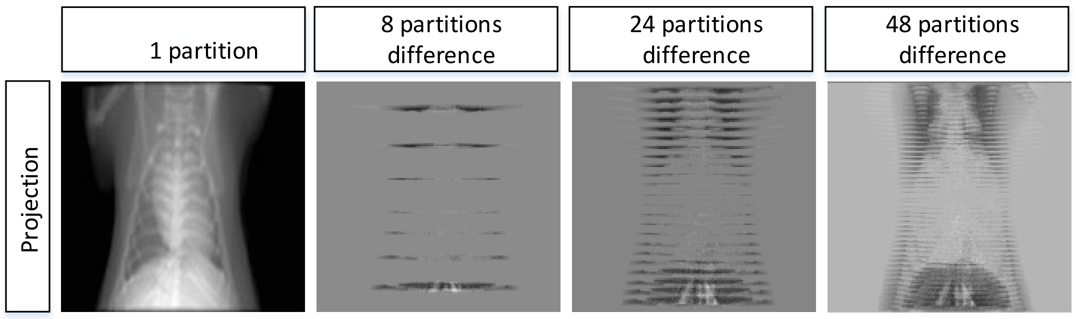


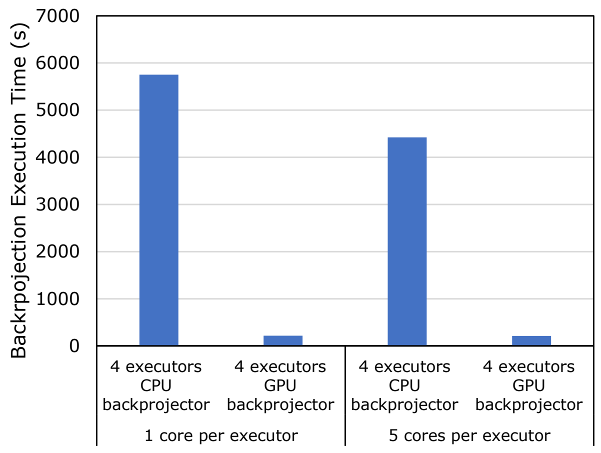
Figure 7.20: *Projection obtained with 1 partition (target) and difference image (the images are shown with a narrow window so the small errors are noticeable) of the projections obtained with different partitioning configurations with respect to the target.*

7.6 Discussion

The GPU and CPU based approaches in conjunction with the different algorithms evaluated have yielded a significant amount of conclusions. First, although both projection and backprojection components are similar in terms of execution and complexity, their inclusion in a Big Data framework exposes different behavior. As seen in Section 7.5, partitioning 3D volumes (which is the most memory consuming data structure) is a suitable approach for *backprojection*, in which partitioning also implies increasing the computational parallelism in the application. However, regarding the projection component, this principle does not hold true. Computational cost is based on the projection data and can even increase when the volume is partitioned due to the increment of the FOV. In Figure 7.21, a comparative between all the evaluated approaches for the same number of *executors* is shown. The best solution combines an increased number of partitions in the backprojection component, aiming at maximizing external/coarse-grained parallelism (even increasing the number of *executors* per node), and a lower number of partitions in case of projection, exploiting the computational resources through a shared-memory mechanism like OpenMP.

Second, in GPU-based architectures, parallelism is provided by using the underlying GPU (internal/fine-grained) and the usage of multiple GPUs devices by different executors (external/ coarse-grained). This approach limits the external parallelism offered by multiple concurrent executors running in the same node. In contrast, our solutions avoid this limitation by orchestrating the execution for a large amount of tasks thanks to our proposed internal scheduler. This approach is specially beneficial in case of the backprojection component.

Finally, it is important to remark the limitations of the framework used, which are more perceptible in the case of the GPU-based architecture. The overhead imposed by Apache Spark can cause not only a problem of performance, but also a problem of memory exhaustion. The



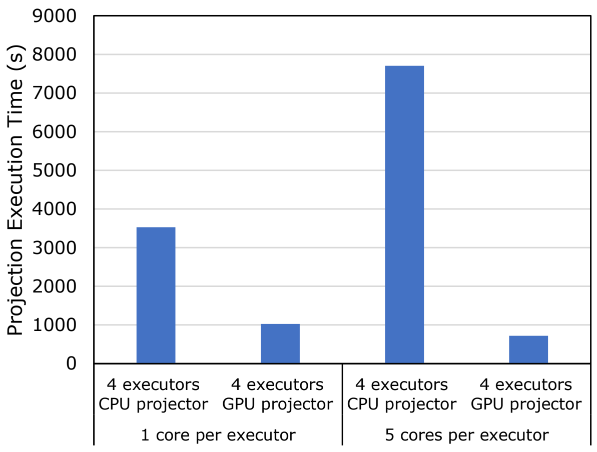


Figure 7.21: *Comparison of all configurations for both algorithms with 4 executors.*

execution of the multiple software layers is translated to a higher memory usage to hold RDDs, Hadoop containers, the buffers for networking, and serialization. This is a problem already noticed in some other works, especially when compared with Apache Hadoop [175, 176, 177]. To overcome these problems, it is necessary to perform a meticulous configuration of the framework execution parameters. This tuning process can take, in some cases, more time than the time needed to implement the application itself.

7.7 Summary

In this chapter we have presented a novel approach based on Python and Apache Spark for the implementation of the backprojection and projection components of an iterative reconstruction method for cone-beam geometry. The solution enables two alternatives for different architectures: a GPU-based architecture, supporting NVidia GPUs, and multi-core CPU architecture, relying on multi-core CPU acceleration and the compatibility with C/C++ native code.

Although the performance results show that a linear speed up is not reached, our approach is an adequate alternative for porting previous HPC applications already implemented in C/C++ or even with CUDA or OpenCL programming models. Furthermore, it is possible to automatically detect the GPU devices and execute CPU and GPU tasks at the same time under the same system, using all the available resources.

This solution is an approach based on Apache Spark, but it is applicable to other Big Data frameworks that support Python, enabling quickly updates for new requirements. Moreover, since the alternatives presented here are based on the union of different components, they can be generalized to other types of architectures or acceleration devices. In the case of the heterogeneous approach, OpenCL could also be used for compatibility with other accelerators or even a better exploitation of the CPU.

The main contributions of this chapter are:

- A novel approach for accelerated iterative reconstruction algorithms based on computation offloading of the most computationally expensive components.

- A complete view of a GPU-based architecture for Apache Spark framework and its evaluation using two main medical imaging algorithms.

- An study of the partitioning problem in the projection and backprojection components in terms of performance.

- An evaluation of the proposed architecture, in both CPU/GPU-based clusters, combining HPC programming models and Big Data frameworks.

Part of this chapter has already been published in the works: *"Architecture for the Execution of Tasks in Apache Spark in Heterogeneous Environments"* [178] and *"Medical Imaging Processing on a Big Data platform using Python: Experiences with Heterogeneous and Homogeneous Architectures"* [179].

Chapter 8

Conclusions

This thesis has covered new advances on the CT research field. As motivated in Section 1.1, we identified the necessity of new solutions for dealing with large scale medical imaging. We have presented a flexible X-Ray simulation framework that is capable of simulating different geometries and system configurations, based on the cone-beam geometry. The proposed simulator architecture is layered to facilitate the introduction of future kernels and configurations. Additionally the solution also offers compatibility with the majority of hardware platforms. The framework for CT reconstruction allows the inclusion of specific optimizations for both heterogeneous and homogeneous architectures. Different programming models have been explored and evaluated, including shared-memory models like OpenMP, CUDA or OpenCL and distributed-memory models such as MPI and MapReduce.

Depending on the availability of the resources, the user experience and the type and size of the input data, the approaches presented in this thesis can provide different advantages or disadvantages:

- Performance: versions with GPU support provide much higher performance than CPU-based versions. This includes FUX-Sim and extended FUX-Sim framework and the distributed Big Data version for PySpark. In all of them the difference between accelerated and non accelerated execution can represent being up to 48× faster. However, the performance is severely affected in low memory systems and GPUs when reconstructing high resolution studies.
- High resolution support: the limiting factor when reconstructing and simulating high resolution images is memory. For this type of studies, distributed resources are needed, especially when working with accelerator-based systems in which GPUs are normally packed with lower memory capacities. However, when performance is not a priority and distributed systems are not available standard FUX-Sim platform executing in an accelerated node can be used, as proved in Chapter 3.
- Programming: except for the Big Data option, all the other alternatives presented in this thesis are constructed around native programming languages. To extend the platform it is necessary to have certain knowledge about the C language. However, domain experts in medical imaging do not normally have expertise in this kind of programming languages. For those cases, the python-based solution for Apache Spark can represent a good option for fast prototyping with the additional advantage of increased performance due to the extended support for GPUs.

In this thesis, we have also studied the unification of traditional HPC-based techniques with modern Big Data methods. The solution presented here combines the performance offered by heterogeneous devices and the leading data management mechanisms offered by the MapReduce programming model.

With the extended FUX-Sim platform, the main objective of this thesis was accomplished obtaining a framework for the simulation and reconstruction of non-regular geometries. We also demonstrated that FUX-Sim provides higher performance than other state-of-the-art platforms with the possibility of reconstructing high resolution studies in near real time.

Additionally, during the development of this thesis, the secondary objectives presented in Section 1.2 have been fulfilled:

O1 To design a flexible simulator/reconstructor for CT medical image processing with support for heterogenous architectures.

In Chapter 2, we described the design and implementation of FUX-Sim, an X-Ray framework for image reconstruction and acquisition simulation. FUX-Sim is compatible with two main hardware architectures: multi-core and GPU accelerated architectures. Additionally, it provides compatibility with different programming models supporting all types of GPUs and devices thanks to CUDA and OpenCL. With CUDA, FUX-SIM is able to obtain full exploitation of NVidia GPUs, meanwhile with OpenCL we enable the compatibility of the application with a larger number of devices.

O2 To improve the performance of simulation/reconstruction algorithms.

Throughout this thesis different solutions have been proposed for optimizing medical imaging algorithms. Chapter 3, more concretely Section 3.3, was dedicated to the optimization of traditional reconstruction algorithms on different architectures and programming models, including GPUs with CUDA and OpenCL support and also multicore architectures with OpenMP. In Chapter 4, additional optimizations were provided taking into account the internal architecture of the GPU. Different configurations and transformation techniques were applied with the objective of exploring the performance of the most time consuming parts over NVidia GPUs.

O3 To design an iterative reconstruction platform for heterogeneous architectures.
Chapter 5 presented the design of a novel iterative reconstruction platform supporting different iterative reconstruction algorithms. It was constructed as an extension of FUX-Sim. Thus, this solution obtains all the advantages of this platform in terms of flexibility of the geometric configurations and the support of heterogeneous architectures with different programming models.

O4 To reconstruct high resolution images.

In Chapters 6 and 7 the need of a larger memory space and computing capabilities for reconstructing high resolution images is solved by translating the iterative reconstruction platform to distributed-memory programming models. In Chapter 6, the HPC paradigm is explored with the use of the MPI programming model. In Chapter 7, the algorithms are transformed to the MapReduce programming model and evaluated in a Big Data platform, Apache Spark, for GPU and CPU-based architectures.

The main contributions of this thesis can be summarized in:

C1 **FUX-Sim**, a novel X-ray simulation/reconstruction framework that was designed to be flexible and fast. It allows the simulation of new geometrical configurations and the reconstruction of the images using different algorithms, which facilitates the task of evaluating novel acquisition systems without designing any physical scanner. The implementation is adapted to two different families of GPUs (with CUDA and OpenCL) and multi-core CPUs using a modularized approach based on a layered architecture and the parallel implementation of the algorithms in both types of devices. Consequently, FUX-Sim can be executed in the majority of the current hardware platforms. FUX-Sim can prove to be valuable for research on new configurations for X-ray systems with non-standard scanning orbits, new acquisition protocols, and advanced reconstruction algorithms.

C2 **Optimized implementation of projection and backprojection algorithms** for different families of GPUs and multi-core CPUs. Kernels for NVidia GPUs are accelerated and evaluated on different architectures, obtaining up to 2× speedup with respect with previous versions. These optimizations enable the possibility of reconstructing in near real-time.

C3 **An iterative reconstruction platform for TV-based reconstruction methods with GPU acceleration.** It solves the problem of iterative reconstruction in an efficient way by using GPUs for the most time-consuming operations. Additionally, it is capable of reconstructing large memory volumes in GPUs thanks to different partitioning strategies devised for these out-of-core problems.

C4 **An iterative reconstruction algorithm adapted to a distributed-memory environment.** Using PETSc and MPI FUX-Sim has been extended to support distributed memory environments, offering new possibilities for the reconstruction of high resolution studies.

C5 **An accelerated architecture for Apache Spark with support for GPUs.** This MPI-based architecture is totally compatible with the standard Apache Spark framework not requiring any additional modification. An *intra-scheduler* for managing GPU resources was including facilitating the execution of the mapping tasks on the GPUs and optimizing the resources used.

8.1 Future directions

The work presented in this thesis can be further extended. On the one hand, it can be extended from the medical imaging point of view, including new algorithms and methods as well as support for other pre and post processing operations. On the other hand, from the point of view of computation additional improvements can be made both in terms of performance and memory consumption:

- Inclusion of other type of pre- and post- processing functions such as image segmentation to complete the framework. These operations could also benefit from the use of optimized components for different architectures and accelerating devices.
- Expansion of the framework to other accelerated programming models and/or graphics API that can also speedup the execution of the projection and backprojection algorithms. The similarities of these kernels with ray casting algorithms used in computer graphics

and their implementation on new software platforms like Vulkan or OpenGL can provide additional flexibility regarding hardware support without losing performance.

- Creation of new mixed projection-backprojection kernels that can accelerate the execution of advanced iterative reconstruction algorithms. Although that approach can reduce the modularity of the platforms presented in this thesis, the combination of both kernels increases the opportunity of applying optimization techniques and therefore, the reduction of the execution time.
- Optimization of the memory management when executing and not executing on accelerators. Study of new storage methods for obtaining better locality of the data and reduce the memory usage. Additional partitioning models with prior information of previous executions in the same hardware could be used to select the best combination of partition size and level of partitioning.
- Introduction of deep learning neural network models to increase the quality and convergence speed of iterative reconstruction models for limited-data and low-dose studies.
- Optimization of the GPU scheduling algorithms in GPU-based architectures for Apache Spark. The addition of a GPU-aware global scheduler would increase the capabilities of offloading computation to less loaded GPU-nodes avoiding possible bottlenecks.
- Inclusion of an advanced physical model for the X-Ray source (heel effect, polychromatic nature, focal spot) or the detector(noise model, intensity response) in order to extend the possibilities of FUX-Sim.

8.2 Research achievements

Apart from the tools presented here derived from this work, we have also produced other research results, including the following publications in International Journals (JCR indexed):

- C. de Molina, E. Serrano, J. Garcia-Blas, J. Carretero, M. Desco, and M. Abella, GPU-accelerated iterative reconstruction for limited-data tomography in CBCT systems, BMC Bioinformatics, vol. 19, issue 1, p. 171, 2018. Impact Factor: 2.448 (Q1).
- M. Abella, E. Serrano, J. Garcia-Blas, I. Garcia, C. de Molina, J. Carretero, and M. Desco, FUX-Sim: Implementation of a fast universal simulation/reconstruction framework for X-ray systems, Plos one, vol. 12, iss. 7, pp. 1-22, 2017. Impact Factor: 2.806 (Q1).
- E. Serrano, J. Garcia-Blas, and J. Carretero, A comparative study of an X-ray tomography reconstruction algorithm in accelerated and cloud computing systems, Concurrency and computation: practice and experience, vol. 27, pp. 5538-5556, 2015. Impact Factor: 0.997 (Q3).

Also related with the development of this thesis we have made several publications in international conferences and workshops:

- E. Serrano, J. Garcia-Blas, J. Carretero, and M. Abella, Medical Imaging Processing on a Big Data platform using Python: Experiences with Heterogeneous and Homogeneous Architectures, in 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017), 2017.

- E. Serrano, T. V. Aa, R. Wuyts, J. Garcia-Blas, J. Carretero, and M. Abella, Exploring a Distributed Iterative Reconstructor Based on Split Bregman Using PETSc, in 16th International conference on Algorithms and Architectures for Parallel Processing, UCER Workshop, 2016.
- E. Serrano, J. Garcia-Blas, J. Carretero, and M. Abella, Architecture for the Execution of Tasks in Apache Spark in Heterogeneous Environments, in 4th International Workshop on Parallelism in Bioinformatics (PBio 2016). Euro-Par 2016, 2016.
- E. Serrano, J. Garcia-Blas, C. Molina, I. Garcia, J. Carretero, M. Desco, and M. Abella, Design and Evaluation of a Parallel and Multi-Platform Cone-Beam X-Ray Simulation Framework, in 4th International Conference on Image Formation in X-Ray Computed Tomography, 2016.
- A. Martinez, A. Garcia-Santos, I. Garcia, E. Serrano, J. Garcia-Blas, C. de Molina, M. Desco, and M. Abella, A software tool for the design and simulation of X-ray acquisition protocols, in 4th International Conference on Image Formation in X-Ray Computed Tomography, 2016.

Additionally, three poster presentations were produced:

- E. Serrano, J. Garcia-Blas, C. Molina, I. Garcia, J. Carretero, M. Desco, and M. Abella, Design and Evaluation of a Parallel and Multi-Platform Cone-Beam X-Ray Simulation Framework, in 4th International Conference on Image Formation in X-Ray Computed Tomography, 2016.
- E. Serrano, J. Garcia-Blas, J. Carretero, Pursuing the HPC and Big Data Convergence: a Medical Imaging use case, in Programming and Tuning Massively Parallel Systems summer School (PUMPS), 2017.
- E. Serrano, J. Garcia-Blas, J. Carretero, Experiences Accelerating a Medical Image Reconstruction Algorithm with HPC and Big Data paradigms, in Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems summer school (ACACES), 2018.

Contributions have also been made in national conferences:

- E. Serrano, J. Garcia-Blas, J. Carretero, and M. Abella, Plataforma flexible y portable de reconstrucción para escáneres de rayos X acelerada con GPUs. Jornadas Sarteco, 2018.
- E. Serrano, J. Garcia-Blas, J. Carretero, and M. Abella, Propuesta arquitectónica para la ejecución de tareas en Apache Spark para entornos heterogéneos. Jornadas de Paralelismo, 2016.

As a result of the work done through the development of this thesis, two software applications have been registered in IPR Office:

- Manuel Desco Menendez, Monica Abella Garcia, Claudia de Molina, Ines Garcia Barquero, Estefania Serrano Lopez, Javier Garcia-Blas, Jesus Carretero. **FUX-Sim**. M-003481/2017, 24/05/2017. University Carlos III of Madrid.

- Manuel Desco, Monica Abella, Claudia de Molina, Estefania Serrano Lopez, Javier Garcia-Blas, Jesus Carretero. **Raptor**. M-003480/2017, 24/05/2017. University Carlos III of Madrid.

During the development of this thesis, I obtained two research internships to collaborate with other research groups:

- IMEC. Leuven, Belgium. Under the direction of PhD. Roel Wuyts. 2 months. From June 2016 to July 2016.
- INESC-ID. Lisbon, Portugal. Under the direction of Professor Leonel Sousa. 1 month. September 2017.

8.3 Funding

This work has been mainly funded thanks to a FPU fellowship (FPU14/03875) from the Spanish Ministry of Education.

It has also been partially supported by other grants:

- DPI2016-79075-R. "Nuevos escenarios de tomografía por rayos X", from the Spanish Ministry of Economy and Competitiveness.
- TIN2016-79637-P *Towards unification of HPC and Big Data Paradigms* from the Spanish Ministry of Economy and Competitiveness.
- Short-term scientific missions (STSM) grant from NESUS COST Action IC1305.
- TIN2013-41350-P, *Scalable Data Management Techniques for High-End Computing Systems* from the Spanish Ministry of Economy and Competitiveness.
- RTC-2014-3028-1 NECRA *Nuevos escenarios clinicos con radiología avanzada* from the Spanish Ministry of Economy and Competitiveness.

Bibliography

[1] Y. Zhang, Y. Chen, H. Huang, J. Sandler, M. Dai, S. Ma, and R. Udelsman, “Diagnostic x-ray exposure increases the risk of thyroid microcarcinoma: a population-based case-control study,” *European journal of cancer prevention: the official journal of the European Cancer Prevention Organisation (ECP)*, vol. 24, no. 5, p. 439, 2015.

[2] M. Law, W.-K. Ma, D. Lau, E. Chan, L. Yip, and W. Lam, “Cumulative radiation exposure and associated cancer risk estimates for scoliosis patients: impact of repetitive full spine radiography,” *European journal of radiology*, vol. 85, no. 3, pp. 625–628, 2016.

[3] T. M. Deserno, “Fundamentals of Medical Image Processing,” in *Springer Handbook of Medical Technology.* Springer, 2011, pp. 1139–1165.

[4] L. Flores, V. Vidal, and G. Verdú, “System matrix analysis for computed tomography imaging,” *PloS one*, vol. 10, no. 11, p. e0143202, 2015.

[5] G. L. Zeng, *Medical Image Reconstruction.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[6] R. L. Siddon, “Fast calculation of the exact radiological path for a three-dimensional CT array,” *Medical physics*, vol. 12, no. 2, pp. 252–255, 1985.

[7] L. Feldkamp, L. Davis, and J. Kress, “Practical cone-beam algorithm,” *JOSA A*, vol. 1, no. 6, pp. 612–619, 1984.

[8] B. Hutton, J. Nuyts, and P. D. H. Zaidi, “Iterative reconstruction methods,” in *Quantitative analysis in nuclear medicine imaging.* Springer, 2006, pp. 107–140.

[9] A. H. Andersen and A. C. Kak, “Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm,” *Ultrasonic imaging*, vol. 6, no. 1, pp. 81–94, 1984.

[10] L. L. Geyer, U. J. Schoepf, F. G. Meinel, J. W. Nance Jr, G. Bastarrika, J. A. Leipsic, N. S. Paul, M. Rengo, A. Laghi, and C. N. De Cecco, “State of the art: iterative CT reconstruction techniques,” *Radiology*, vol. 276, no. 2, pp. 339–357, 2015.

[11] M. Beister, D. Kolditz, and W. A. Kalender, “Iterative reconstruction methods in X-ray CT,” *Physica Medica*, vol. 28, no. 9, p. 4e108, 2012.

[12] H. Hurwitz Jr, “Entropy reduction in Bayesian analysis of measurements,” *Physical Review A*, vol. 12, no. 2, p. 698, 1975.

[13] J. Leipsic, T. M. LaBounty, B. Heilbron, J. K. Min, G. J. Mancini, F. Y. Lin, C. Taylor, A. Dunning, and J. P. Earls, "Adaptive statistical iterative reconstruction: assessment of image noise and image quality in coronary CT angiography," *American Journal of Roentgenology*, vol. 195, no. 3, pp. 649–654, 2010.

[14] C. M. Systems, "AIDR3d | Technology | CT | Canon Medical Systems." [Online]. Available: https://global.medical.canon/products/computed-tomography/aidr3d_technology

[15] C. Ghetti, O. Ortenzia, and G. Serreli, "CT iterative reconstruction in image space: a phantom study," *Physica medica*, vol. 28, no. 2, pp. 161–165, 2012.

[16] K. Grant and R. Raupach, "SAFIRE: Sinogram affirmed iterative reconstruction," *White Paper*, pp. 1–8, 2012.

[17] Philips, "iDose." [Online]. Available: http://incenter.medical.philips.com/doclib/enc/fetch/2000/4504/577242/577249/586938/587315/iDose4_-_Whitepaper_-_Technical_-Low_Res.pdf%3fnodeid%3d8432599%26vernum%3d-2

[18] H. Chen, Y. Zhang, Y. Chen, J. Zhang, W. Zhang, H. Sun, Y. Lv, P. Liao, J. Zhou, and G. Wang, "LEARN: Learned Experts' Assessment-based Reconstruction Network for Sparse-data CT," *IEEE Transactions on Medical Imaging*, 2018.

[19] G. Wang, "A perspective on deep imaging," *arXiv preprint arXiv:1609.04375*, 2016.

[20] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. van der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[21] Q. Yang, M. K. Kalra, A. Padole, J. Li, E. Hilliard, R. Lai, and G. Wang, "Big data from ct scanning," *JSM Biomed. Imag.*, vol. 2, no. 1, pp. 1003–1, 2015.

[22] H. Chen, Y. Zhang, M. K. Kalra, F. Lin, Y. Chen, P. Liao, J. Zhou, and G. Wang, "Low-dose CT with a residual encoder-decoder convolutional neural network," *IEEE transactions on medical imaging*, vol. 36, no. 12, pp. 2524–2535, 2017.

[23] "Low-dose CT via convolutional neural network, author=Chen, Hu and Zhang, Yi and Zhang, Weihua and Liao, Peixi and Li, Ke and Zhou, Jiliu and Wang, Ge," *Biomedical optics express*, vol. 8, no. 2, pp. 679–694, 2017.

[24] D. Nie, X. Cao, Y. Gao, L. Wang, and D. Shen, "Estimating ct image from mri data using 3d fully convolutional networks," in *Deep Learning and Data Labeling for Medical Applications.* Springer, 2016, pp. 170–178.

[25] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[26] K. J. O'Dwyer and D. Malone, "Bitcoin Mining and its Energy Footprint," pp. 280–285, Jan. 2014.

[27] J. A. Dev, "Bitcoin mining acceleration and performance quantification," in *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on.* IEEE, 2014, pp. 1–6.

[28] NVidia, "GP100 Pascal Whitepaper," p. 45. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[29] ——, "GV100 Volta Whitepaper." [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[30] R. T. G. AMD, "Radeon's next-generation vega architecture," *Radeon's next-generation Vega architecture*, 2017.

[31] Intel, "Intel® Xeon Phi™ Coprocessor - the Architecture," Jul. 2015. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[32] A. Sodani, "Knights landing (knl): 2nd generation intel® xeon phi processor," in *Hot Chips 27 Symposium (HCS), 2015 IEEE.* IEEE, 2015, pp. 1–24.

[33] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Foundations and trends in electronic design automation*, vol. 2, no. 2, pp. 135–253, 2008.

[34] M. Blott, "Reconfigurable future for HPC," in *2016 International Conference on High Performance Computing & Simulation (HPCS) 2016.* IEEE, 2016, pp. 130–131.

[35] W. Zhang, L. Shen, T. Page, G. Luo, P. Li, P. Maaß, M. Jiang, and J. Cong, "FPGA Acceleration for Simultaneous Image Reconstruction and Segmentation based on the Mumford-Shah Regularization," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 2015, pp. 261–261.

[36] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[37] U. Farooq, Z. Marrakchi, and H. Mehrez, "Fpga architectures: An overview," in *Tree-based heterogeneous FPGA architectures.* Springer, 2012, pp. 7–48.

[38] "Top 500," Nov 2017. [Online]. Available: https://www.top500.org/lists/2017/11/

[39] P. Van-Roy and S. Haridi, *Concepts, techniques, and models of computer programming.* MIT press, 2004.

[40] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[41] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[42] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming.* MIT press, 2008, vol. 10.

[43] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC? first experiences with real-world applications," in *European Conference on Parallel Processing.* Springer, 2012, pp. 859–870.

[44] J. Garcia, "Repara c++ open specification," *REPARA EU FP7 project, Tech. Rep. ICT-609666-D2*, vol. 1, pp. 2–14.

[45] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.

[46] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[47] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide.* Pearson Education, 2011.

[48] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *International Conference on Parallel Processing (ICPP), 2011.* IEEE, 2011, pp. 216–225.

[49] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition.* Elsevier, 2011, pp. 359–371.

[50] M. Wong, A. Richards, M. Rovatsou, and R. Reyes, "Khronos's opencl sycl to support heterogeneous devices for c++," 2016.

[51] D. del Rio Astorga, M. F. Dolz, L. M. Sanchez, J. G. Blas, and J. D. García, "A C++ generic parallel pattern interface for stream processing," in *International Conference on Algorithms and Architectures for Parallel Processing.* Springer, 2016, pp. 74–87.

[52] H. Scherl, M. Kowarschik, H. G. Hofmann, B. Keck, and J. Hornegger, "Evaluation of state-of-the-art hardware architectures for fast cone-beam CT reconstruction," *Parallel Computing*, vol. 38, no. 3, pp. 111 – 124, 2012.

[53] E. Liria, D. Higuero, M. Abella, C. de Molina, and M. Desco, "Exploiting Parallelism in a X-ray Tomography Reconstruction Algorithm on Hybrid Multi-GPU and Multi-core Platforms," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA),*, 2012, pp. 867–868.

[54] E. Papenhausen and K. Mueller, "Rapid rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction," *hgpu.org*, Apr. 2014.

[55] A. Hart, R. Ansaloni, and A. Gray, "Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 5–16, Aug. 2012.

[56] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC: First Experiences with Real-world Applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870.

[57] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, 2013.

[58] J. Dongarra, H. Jagode, S. Moore, P. Mucci, J. Ralph, D. Terpstra, and V. Weaver, "Performance application programming interface."

[59] NVidia, “NVidia Profiler.” [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html

[60] K.-H. Kim, K. Kim, and Q.-H. Park, “Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model,” *Computer Physics Communications*, vol. 182, no. 6, pp. 1201–1207, 2011.

[61] P. Carvalho, L. M. Drummond, C. Bentes, E. Clua, E. Cataldo, and L. A. Marzulo, “Analysis and Characterization of GPU Benchmarks for Kernel Concurrency Efficiency,” in *Latin American High Performance Computing Conference*. Springer, 2017, pp. 71–86.

[62] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John, “Gpgpu benchmark suites: How well do they sample the performance spectrum?” in *International Conference on Parallel Processing (ICPP), 2015 44th*. IEEE, 2015, pp. 320–329.

[63] S. Che and K. Skadron, “BenchFriend: Correlating the performance of GPU benchmarks,” *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 238–250, 2014.

[64] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, “A survey of performance modeling and simulation techniques for accelerator-based computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 272–281, 2015.

[65] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[66] H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li, “GPURoofline: a model for guiding performance optimizations on GPUs,” in *European Conference on Parallel Processing*. Springer, 2012, pp. 920–932.

[67] E. Konstantinidis and Y. Cotronis, “A practical performance model for compute and memory bound GPU kernels,” in *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2015*. IEEE, 2015, pp. 651–658.

[68] C. Nugteren, G.-J. van den Braak, and H. Corporaal, “Roofline-aware DVFS for GPUs,” in *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*. ACM, 2014, p. 8.

[69] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying GPU microarchitecture through microbenchmarking,” in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010*. IEEE, 2010, pp. 235–246.

[70] X. Mei and X. Chu, “Dissecting GPU memory hierarchy through microbenchmarking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[71] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, “Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017*. IEEE, 2017, pp. 259–268.

[72] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[73] "Hadoop." [Online]. Available: http://hadoop.apache.org/

[74] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, p. 5.

[75] D. Borthakur *et al.*, "HDFS architecture guide," *Hadoop Apache Project*, vol. 53, 2008.

[76] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system.* ACM, 2003, vol. 37, no. 5.

[77] "Apache Spark." [Online]. Available: http://spark.apache.org/docs/latest/index.html

[78] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* USENIX Association, 2012, pp. 2–2.

[79] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM international symposium on high performance distributed computing.* ACM, 2010, pp. 810–818.

[80] D. Lezzi, R. Rafanell, A. Carrión, I. B. Espert, V. Hernández, and R. M. Badia, "Enabling e-Science applications on the Cloud with COMPSs," in *European Conference on Parallel Processing.* Springer, 2011, pp. 25–34.

[81] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Computing*, vol. 37, no. 9, pp. 610–632, 2011.

[82] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards efficient mapreduce using mpi," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting.* Springer, 2009, pp. 240–249.

[83] H. Mohamed and S. Marchand-Maillet, "MRO-MPI: MapReduce overlapping using MPI and an optimized data exchange policy," *Parallel Computing*, vol. 39, no. 12, pp. 851–866, 2013.

[84] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable MapReduce for large supercomputing systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017.* IEEE, 2017, pp. 1098–1108.

[85] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: extending MPI to hadoop-like big data computing," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE, 2014, pp. 829–838.

[86] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[87] L. Sousa, P. Kropf, P. Kuonen, R. Prodan, A. T. Trinh, and J. Carretero, "A roadmap for research in sustainable ultrascale systems," 2017.

[88] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, “Big data, simulations and hpc convergence,” in *Big Data Benchmarking.* Springer, 2015, pp. 3–17.

[89] S. Okur, C. Radoi, and Y. Lin, “Hadoop+ aparapi: Making heterogeneous mapreduce programming easier.”

[90] M. Grossman, M. Breternitz, and V. Sarkar, “Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl,” in *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013.* IEEE, 2013, pp. 1918–1927.

[91] W. He, H. Cui, B. Lu, J. Zhao, S. Li, G. Ruan, J. Xue, X. Feng, W. Yang, and Y. Yan, “Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters,” in *Proceedings of the 29th ACM on International Conference on Supercomputing.* ACM, 2015, pp. 143–153.

[92] P. Li, Y. Luo, N. Zhang, and Y. Cao, “HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms,” in *IEEE International Conference on Networking, Architecture and Storage (NAS), 2015.* IEEE, 2015, pp. 347–348.

[93] “GPUEnabler: Provides GPU awareness to Spark,” Sep. 2017. [Online]. Available: https://github.com/IBMSparkGPU/GPUEnabler

[94] S. Hong, W. Choi, and W.-K. Jeong, “GPU in-memory processing using Spark for iterative computation,” in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.* IEEE Press, 2017, pp. 31–41.

[95] D. Fukutomi, Y. Iida, T. Azumi, S. Kato, and N. Nishio, “GPUhd: Augmenting YARN with GPU Resource Management,” in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region.* ACM, 2018, pp. 127–136.

[96] R. N. Boubela, K. Kalcher, W. Huf, C. Našel, and E. Moser, “Big Data approaches for the analysis of large-scale fMRI data using Apache Spark and GPU processing: A demonstration on resting-state fMRI data from the Human Connectome Project,” *Frontiers in neuroscience*, vol. 9, 2015.

[97] Y. Du, G. Yu, X. Xiang, and X. Wang, “GPU accelerated voxel-driven forward projection for iterative reconstruction of cone-beam CT,” *BioMedical Engineering OnLine*, vol. 16, Dec. 2017.

[98] S. Rit, M. van Herk, and J.-J. Sonke, “Fast distance-driven projection and truncation management for iterative cone-beam CT reconstruction,” in *Proc. International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology Nuclear Medicine*, 2009, pp. 49–52.

[99] J. Hu, X. Zhao, and H. Zhang, “A GPU-based multi-resolution approach to iterative reconstruction algorithms in x-ray 3D dual spectral computed tomography,” *Neurocomputing*, vol. 215, pp. 71–81, 2016.

[100] Jian-Lin, Chen and Lei, Li and Lin-Yuan, Wang and Ai-Long, Cai and Xiao-Qi, Xi and Han-Ming, Zhang and Jian-Xin, Li and Bin, Yan, “Fast parallel algorithm for three-dimensional distance-driven model in iterative computed tomography reconstruction,” *Chinese Physics B*, vol. 24, no. 2, p. 028703, 2015.

[101] X. Zhao, J.-J. Hu, and P. Zhang, “GPU-based 3D cone-beam CT image reconstruction for large data volume,” *Journal of Biomedical Imaging*, vol. 2009, pp. 8:1–8:8, January 2009.

[102] V.-G. Nguyen, J. Jeong, and S.-J. Lee, “GPU-accelerated iterative 3D CT reconstruction using exact ray-tracing method for both projection and backprojection,” in *IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2013*. IEEE, 2013, pp. 1–4.

[103] V. H. Naik and C. S. Kusur, “Analysis of performance enhancement on graphic processor based heterogeneous architecture: A CUDA and MATLAB experiment,” in *National Conference on Parallel Computing Technologies (PARCOMPTECH), 2015*. IEEE, 2015, pp. 1–5.

[104] Y. Lu, F. Ino, and K. Hagihara, “Cache-Aware GPU Optimization for Out-of-Core Cone Beam CT Reconstruction of High-Resolution Volumes,” *IEICE TRANSACTIONS on Information and Systems*, vol. E99-D, no. 12, pp. 3060–3071, Dec. 2016. [Online]. Available: http://search.ieice.org/bin/summary.php?id=e99-d_12_3060&category=D&year=2016&lang=E&abst=

[105] S. Agaian, P. Rad, R. Rajendran, and K. Panetta, “A Novel Technique to Enhance Low Resolution CT and Magnetic Resonance Images in Cloud,” in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, Nov 2016, pp. 73–78.

[106] K. L. Grant, T. G. Flohr, B. Krauss, M. Sedlmair, C. Thomas, and B. Schmidt, “Assessment of an advanced image-based technique to calculate virtual monoenergetic computed tomographic images from a dual-energy examination to improve contrast-to-noise ratio in examinations using iodinated contrast media,” *Investigative Radiology: A journal of Clinical and Laboratory Research*, vol. 49, no. 9, p. 586–592, September 2014.

[107] J. Solomon, A. Mileto, J. C. Ramirez-Giraldo, and E. Samei, “Diagnostic Performance of an Advanced Modeled Iterative Reconstruction Algorithm for Low-Contrast Detectability with a Third-Generation Dual-Source Multidetector CT Scanner: Potential for Radiation Dose Reduction in a Multireader Study,” *Radiology*, vol. 275, no. 3, pp. 735–745, 2015.

[108] R. D. A. Khawaja, S. Singh, M. Blake, M. Harisinghani, G. Choy, A. Karosmanoglu, A. Padole, S. Pourjabbar, S. Do, and M. K. Kalra, “Ultralow-Dose Abdominal Computed Tomography: Comparison of 2 Iterative Reconstruction Techniques in a Prospective Clinical Study,” *Journal of Computer Assisted Tomography*, vol. 39, no. 4, p. 489, Aug. 2015.

[109] X. Wang, A. Sabne, S. Kisner, A. Raghunathan, C. Bouman, and S. Midkiff, “High Performance Model Based Image Reconstruction,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’16. New York, NY, USA: ACM, 2016, pp. 2:1–2:12.

[110] L. Xie, Y. Hu, B. Yan, L. Wang, B. Yang, W. Liu, L. Zhang, L. Luo, H. Shu, and Y. Chen, “An Effective CUDA Parallelization of Projection in Iterative Tomography Reconstruction,” *PLOS ONE*, vol. 10, no. 11, p. e0142184, Nov. 2015.

[111] A. Sabne, X. Wang, S. J. Kisner, C. A. Bouman, A. Raghunathan, and S. P. Midkiff, “Model-based Iterative CT Image Reconstruction on GPUs,” in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’17. New York, NY, USA: ACM, 2017, pp. 207–220.

[112] T. Bai, H. Yan, X. Jia, S. Jiang, G. Wang, and X. Mou, “Z-Index Parameterization for Volumetric CT Image Reconstruction via 3-D Dictionary Learning,” *IEEE Transactions on Medical Imaging*, vol. 36, no. 12, pp. 2466–2478, Dec 2017.

[113] B. Meng, G. Pratx, and L. Xing, “Ultrafast and scalable cone-beam CT reconstruction using MapReduce in a cloud computing environment,” *Medical physics*, vol. 38, no. 12, pp. 6603–6609, 2011.

[114] Palenstijn, W.J., Bédorf, J., Batenburg, K.J., King, M., Glick, S., Mueller, K., and NWO, “A distributed SIRT implementation for the ASTRA Toolbox,” Jun. 2015.

[115] J. M. Rosen, J. Wu, T. F. Wenisch, and J. A. Fessler, “Iterative helical CT reconstruction in the cloud for ten dollars in five minutes,” in *Proc. Intl. Mtg. on Fully 3D Image Recon. in Rad. and Nuc. Med*, 2013, pp. 241–4.

[116] L. Cao, P. Juan, and Y. Zhang, “Real-Time Deconvolution with GPU and Spark for Big Imaging Data Analysis,” in *Algorithms and Architectures for Parallel Processing.* Springer, 2015, pp. 240–250.

[117] S. Bao, B. Landman, and A. Gokhale, “Algorithmic Enhancements to Big Data Computing Frameworks for Medical Image Processing,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2017, pp. 13–16.

[118] X. Yang, T. Jejkal, H. Pasic, R. Stotzka, A. Streit, J. v. Wezel, and T. d. S. Rolo, “Data Intensive Computing of X-Ray Computed Tomography Reconstruction at the LSDF,” in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2013, pp. 86–93.

[119] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, “scikit-image: image processing in Python,” *PeerJ*, vol. 2, p. e453, 2014.

[120] “MedPy — MedPy 0.3.0 documentation.” [Online]. Available: http://loli.github.io/medpy/#reference

[121] M. S. Hansen and T. S. Sørensen, “Gadgetron: An open source framework for medical image reconstruction,” *Magnetic Resonance in Medicine*, vol. 69, no. 6, pp. 1768–1776, 2013.

[122] D. Gürsoy, F. De Carlo, X. Xiao, and C. Jacobsen, “TomoPy: a framework for the analysis of synchrotron tomographic data,” *Journal of synchrotron radiation*, vol. 21, no. 5, pp. 1188–1193, 2014.

[123] W. van Aarle, W. J. Palenstijn, J. De Beenhouwer, T. Altantzis, S. Bals, K. J. Batenburg, and J. Sijbers, “The ASTRA Toolbox: A platform for advanced algorithm development in electron tomography,” *Ultramicroscopy*, vol. 157, pp. 35–47, 2015.

[124] J. Garcia-Blas, M. Abella, F. Isaila, J. Carretero, and M. Desco, “Surfing the optimization space of a multiple-GPU parallel implementation of a X-ray tomography reconstruction algorithm ,” *Journal of Systems and Software*, 2014.

[125] J. G. Blas, F. Isaila, M. Abella, J. Carretero, E. Liria, and M. Desco, "Parallel Implementation of a X-ray Tomography Reconstruction Algorithm Based on MPI and CUDA," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: ACM, 2013, pp. 217–222.

[126] J. Deng, H. Yu, J. Ni, T. He, S. Zhao, L. Wang, and G. Wang, "A parallel implementation of the Katsevich algorithm for 3-D CT image reconstruction," *The Journal of Supercomputing*, vol. 38, no. 1, pp. 35–47, 2006.

[127] L. Domanski, T. Bednarz, T. Gureyev, L. Murray, B. E. Huang, Y. Nesterets, D. Thompson, E. Jones, C. Cavanagh, D. Dadong *et al.*, "Applications of heterogeneous computing in computational and simulation science," *International Journal of Computational Science and Engineering*, vol. 8, no. 3, pp. 240–252, 2013.

[128] Z. Fan and Y. Xie, "A block-wise approximate parallel implementation for ART algorithm on CUDA-enabled GPU," *Bio-medical materials and engineering*, vol. 26, no. s1, pp. S1027–S1035, 2015.

[129] H. Köstler, M. Stürmer, and T. Pohl, "Performance engineering to achieve real-time high dynamic range imaging," *Journal of Real-Time Image Processing*, vol. 11, no. 1, pp. 127–139, 2016.

[130] C. Melvin, M. Xu, and P. Thulasiraman, "HPC for iterative image reconstruction in CT," in *Proceedings of the 2008 C 3 S 2 E conference.* ACM, 2008, pp. 61–68.

[131] A. Tirado-Ramos, P. M. Sloot, A. G. Hoekstra, and M. Bubak, "An integrative approach to high-performance biomedical problem solving environments on the Grid," *Parallel Computing*, vol. 30, no. 9-10, pp. 1037–1055, 2004.

[132] J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein, "Pushing the limits for medical image reconstruction on recent standard multicore processors," *The International Journal of High Performance Computing Applications*, vol. 27, no. 2, pp. 162–177, 2013.

[133] J. Kole and F. J. Beekman, "Evaluation of accelerated iterative x-ray CT image reconstruction using floating point graphics hardware," *Physics in Medicine & Biology*, vol. 51, no. 4, p. 875, 2006.

[134] 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 transactions on medical imaging*, vol. 28, no. 3, pp. 435–445, 2009.

[135] R. Sampson, M. McGaffin, T. Wenisch, and J. Fessler, "Investigating multi-threaded SIMD for helical CT reconstruction on a CPU," in *Proceedings of the 4th International Meeting on image formation in X-ray CT*, 2016, pp. 275–278.

[136] H.-G. Park, Y.-G. Shin, and H. Lee, "A fully GPU-based ray-driven backprojector via a ray-culling scheme with voxel-level parallelization for cone-beam CT reconstruction," *Technology in cancer research & treatment*, vol. 14, no. 6, pp. 709–720, 2015.

[137] S. Mukherjee, N. Moore, J. Brock, and M. Leeser, "CUDA and OpenCL implementations of 3D CT reconstruction for biomedical imaging," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on.* IEEE, 2012, pp. 1–6.

[138] C. Siegl, H. Hofmann, B. Keck, M. Prümmer, and J. Hornegger, "OpenCL: a viable solution for high-performance medical image reconstruction?" in *SPIE Medical Imaging.* International Society for Optics and Photonics, 2011, pp. 79 612Q–79 612Q.

[139] C. B. Mendl, S. Eliuk, M. Noga, and P. Boulanger, "Comprehensive analysis of high-performance computing methods for filtered back-projection," 2013.

[140] F. Xu, "Fast implementation of iterative reconstruction with exact ray-driven projector on GPUs," *Tsinghua Science & Technology*, vol. 15, no. 1, pp. 30–35, 2010.

[141] Y. Zhu, Y. Zhao, and X. Zhao, "A multi-thread scheduling method for 3d ct image reconstruction using multi-gpu," *Journal of X-Ray Science and Technology*, vol. 20, no. 2, pp. 187–197, 01 2012.

[142] A. Biguri, M. Dosanjh, S. Hancock, and M. Soleimani, "TIGRE: a MATLAB-GPU toolbox for CBCT image reconstruction," *Biomedical Physics & Engineering Express*, vol. 2, no. 5, p. 055010, 2016.

[143] D. Matenine, Y. Goussard, and P. Després, "GPU-accelerated regularized iterative reconstruction for few-view cone beam CT," *Medical Physics*, vol. 42, no. 4, pp. 1505–1517, 2015.

[144] "CT Sim," 2002. [Online]. Available: http://files.b9.com/ctsim/ctsim-manual-latest.pdf

[145] J. Fessler, "Michigan image reconstruction toolbox (MIRT)."

[146] "X-Ray Sim," 2015. [Online]. Available: http://xraysim.sourceforge.net/index.htm

[147] F. Verhaegen *et al.*, "ImaSim, a software tool for basic education of medical x-ray imaging in radiotherapy and radiology," *Frontiers in Physics*, vol. 1, p. 22, 2013.

[148] A. Maier, H. G. Hofmann, M. Berger, P. Fischer, C. Schwemmer, H. Wu, K. Müller, J. Hornegger, J.-H. Choi, C. Riess *et al.*, "CONRAD—A software framework for cone-beam imaging in radiology," *Medical physics*, vol. 40, no. 11, 2013.

[149] W. J. Palenstijn, K. J. Batenburg, and J. Sijbers, "The ASTRA tomography toolbox," in *13th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE*, vol. 2013, 2013.

[150] B. De Man and S. Basu, "Distance-driven projection and backprojection," in *Nuclear Science Symposium Conference Record, 2002 IEEE*, vol. 3. IEEE, 2002, pp. 1477–1480.

[151] ——, "Distance-driven projection and backprojection in three dimensions," *Physics in medicine and biology*, vol. 49, no. 11, p. 2463, 2004.

[152] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.

[153] M. Abella, E. Serrano, J. Garcia-Blas, I. García, C. de Molina, J. Carretero, and M. Desco, "FUX-Sim: Implementation of a fast universal simulation/reconstruction framework for X-ray systems," *PLOS ONE*, vol. 12, no. 7, pp. 1–22, 07 2017.

[154] A. Ilic, F. Pratas, and L. Sousa, “Beyond the roofline: cache-aware power and energy-efficiency modeling for multi-cores,” *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 52–58, 2017.

[155] ——, “Cache-aware Roofline model: Upgrading the loft,” *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

[156] NVidia, “Kepler Architecture: White Paper.” [Online]. Available: http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[157] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded gpu using cuda,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming.* ACM, 2008, pp. 73–82.

[158] T. Goldstein and S. Osher, “The Split Bregman Method for L1-Regularized Problems,” *SIAM J. Img. Sci.*, vol. 2, no. 2, pp. 323–343, Apr. 2009.

[159] J. F. P. J. Abascal, M. Abella, A. Sisniega, J. J. Vaquero, and M. Desco, “Investigation of Different Sparsity Transforms for the PICCS Algorithm in Small-Animal Respiratory Gated CT,” *PLOS ONE*, vol. 10, no. 4, pp. 1–18, 04 2015.

[160] J.-F. Cai, S. Osher, and Z. Shen, “Split Bregman methods and frame based image restoration,” *Multiscale modeling & simulation*, vol. 8, no. 2, pp. 337–369, 2009.

[161] C. de Molina, J. F. P. J. Abascal, J. Pascau, M. Desco, and M. Abella, “Evaluation of the possibilities of limited angle reconstruction for the use of digital Radiography system as a tomograph,” in *IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, Nov 2014, pp. 1–4.

[162] T. Goldstein and S. Osher, “The split Bregman method for L1-regularized problems,” *SIAM journal on imaging sciences*, vol. 2, no. 2, pp. 323–343, 2009.

[163] H. A. Van der Vorst, “Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems,” *SIAM Journal on scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[164] L. I. Rudin, S. Osher, and E. Fatemi, “Nonlinear Total Variation Based Noise Removal Algorithms,” *Phys. D*, vol. 60, no. 1-4, pp. 259–268, Nov. 1992.

[165] J. I. Agulleiro and J. J. Fernandez, “Fast tomographic reconstruction on multicore computers,” *Bioinformatics*, vol. 27, no. 4, pp. 582–583, 2011.

[166] H. A. van der Vorst, “BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems,” *SIAM J. Sci. Stat. Comput.*, vol. 13, no. 2, pp. 631–644, Mar. 1992.

[167] M. Abella, J. Vaquero, A. Sisniega, J. Pascau, A. Udías, V. García, I. Vidal, and M. Desco, “Software architecture for multi-bed FDK-based reconstruction in X-ray CT scanners,” *Computer Methods and Programs in Biomedicine*, vol. 107, no. 2, pp. 218 – 232, 2012.

[168] C. de Molina, E. Serrano, J. Garcia-Blas, J. Carretero, M. Desco, and M. Abella, "GPU-accelerated iterative reconstruction for limited-data tomography in CBCT systems," *BMC bioinformatics*, vol. 19, no. 1, p. 171, 2018.

[169] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," 2016. [Online]. Available: http://www.mcs.anl.gov/petsc

[170] E. Serrano, T. Vander Aa, R. Wuyts, J. G. Blas, J. Carretero, and M. Abella, "Exploring a distributed iterative reconstructor based on split bregman using petsc," in *International Conference on Algorithms and Architectures for Parallel Processing.* Springer, 2016, pp. 191–200.

[171] C.-T. Yang, W.-C. Shih, L.-T. Chen, C.-T. Kuo, F.-C. Jiang, and F.-Y. Leu, "Accessing medical image file with co-allocation HDFS in cloud," *Future Generation Computer Systems*, vol. 43, pp. 61–73, 2015.

[172] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[173] A. Abbasi, F. Khunjush, and R. Azimi, "A preliminary study of incorporating GPUs in the Hadoop framework," in *Computer Architecture and Digital Systems (CADS), 2012 16th CSI International Symposium on.* IEEE, 2012, pp. 178–185.

[174] "RPyC - Transparent, Symmetric Distributed Computing — RPyC." [Online]. Available: https://rpyc.readthedocs.io/en/latest/index.html#

[175] A. G. Shoro and T. R. Soomro, "Big data analysis: Apache spark perspective," *Global Journal of Computer Science and Technology*, vol. 15, no. 1, 2015.

[176] L. Gu and H. Li, "Memory or time: Performance evaluation for iterative operation on Hadoop and Spark," in *IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013.* IEEE, 2013, pp. 721–727.

[177] S. Han, W. Choi, R. Muwafiq, and Y. Nah, "Impact of Memory Size on Bigdata Processing based on Hadoop and Spark," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems.* ACM, 2017, pp. 275–280.

[178] E. Serrano, J. G. Blas, J. Carretero, and M. Abella, "Architecture for the execution of tasks in apache spark in heterogeneous environments," in *4th International Workshop on Parallelism in Bioinformatics*, 2016.

[179] E. Serrano, J. Garcia-Blas, J. Carretero, M. Abella, and M. Desco, "Medical Imaging Processing on a Big Data Platform Using Python: Experiences with Heterogeneous and Homogeneous Architectures," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 830–837. [Online]. Available: https://doi.org/10.1109/CCGRID.2017.56