POLITECNICO DI TORINO

Master of science in Energetic and Nuclear Energy

Master Thesis

Profiling of Monte Carlo simulations in particle therapy and code optimization strategy

Advisor:
prof.ssa Sandra Dulla

Co-Advisor:
Faiza Bourhaleb (i-See s.r.l.)

Candidate:
Fabrizio Bello

December 2020

A mia nonna

I've missed more than 9,000 shots in my career. I've lost almost 300 games. Twenty-six times, I've been trusted to take the game winning shot and missed. I've failed over and over and over again in my life. And that is why I succeed

Micheal Jordan

Summary

The objective of this dissertation is profiling Monte Carlo simulations codes used in the field of particle therapy. This allows to analyze most important criticalities during the execution of those simulations. Two major aspects have been taken into account in this study: time of execution and memory allocated. This is mainly for two different reasons: improving performances in terms of speeding up the simulation and to decrease the memory footprint of each of them.

To understand where we have margins for improvements in our codes, simulations have been ran under a profiling toolkit tuning different aspect of each simulation to define dependencies among each of its different properties. In a systematic way it has been possible to move from a simple Bragg Peak simulation to a more complete study for a realistic clinical case, planning a treatment for a real patient. Results have been described and analyzed from the point of view of both the software and the hardware impacts.

This work is concluded with the analysis of possible optimization strategies for execution time reduction through parallel programming using GPUs architecture and also for reduction of memory consumption with the optimization of part of the code.

Sommario

L'obiettivo di questa tesi è il profiling di codici per Simulazioni Monte Carlo nell'ambito della terapia adronica. Questo ci permette di analizzare le più importanti criticità durante l'esecuzione di tali simulazioni. Due sono stati gli aspetti di maggior interesse: il tempo necessario all'esecuzione e la memoria allocata. Tali aspetti sono stati scelti per due ragioni differenti: al fine di migliorare le performance, in termini di velocità e riducendo l'impatto della memoria di ogni simulazioni. Le simulazioni sono state eseguite utilizzando un toolkit per il profiling modificando diversi aspetti della simulazione per definire le dipendenze rispetto a diverse proprietà della simulazione stessa. Mediante lavoro sistematico è stato possibile muoversi dalla semplice simulazione di un Bragg Peak in 1D fino ad un caso clinico più articolato, pianificando il trattamento di un paziente reale. I risultati ottenuti sono stati descritti e analizzati dal punto di vista sia del software che dell'hardware. Questo lavoro si conclude con l'analisi di possibili strategie di ottimizzazione sia per ridurre il tempo di esecuzione attraverso la programmazione in parallelo su architettura GPU e sia per la riduzione del consumo di memoria con possibili interventi all'interno del codice stesso

Contents

List of Figures

List of Tables

Chapter 1

Particle therapy

1.1 History of medical application of radiations

Since x-ray discovery by Roentgen back in 1895, radiations arouse interest on their possible application in medical field, in particular cancer treatment. Just the following year the French physician Victor Despeignes attempted a X-ray treatment for a stomac cancer case and took all the following years in understanding the side effect of such treatment. In the same period scientist Marie Curie discovered two radioactive elements: polonium and radium that ensure her a Nobel Prize initiating several research among possible curative powers ofrRadium. The scant knowladge on radiation exposure results in a fast development of nuclear applications for medicine that where commercialized for pubblic use. From 1915 different public figures stood for public healt concerns till 1941, in which the National Bureau of Standards established the tolerance level for radioactivity to $0.1\mu Ci \sim 3.7kBq$. Even though the commercialization of several products was stopped, the resarch went further on with severral radiation sources studied. A real turning point was the passagge from mass-less photons to heavier particle, such as protons, thanks to the advent of particle accelerators.

In 1946, Robert R. Wilson designed a 150-MeV cyclotron for the Harvard University through which he was able to understand the main properties of high-energy protons beam and outlined the opportunity of using it for cancer treatment[2]. All of his intuitions were included in an article published in the medical journal 'Radiology' entitled *'Radiological Use of Fast Protons'* [3]. The most important acknowledgment was the difference in the depth-dose distribution that, with respect to photons, had an increasing effect with depth.[4]

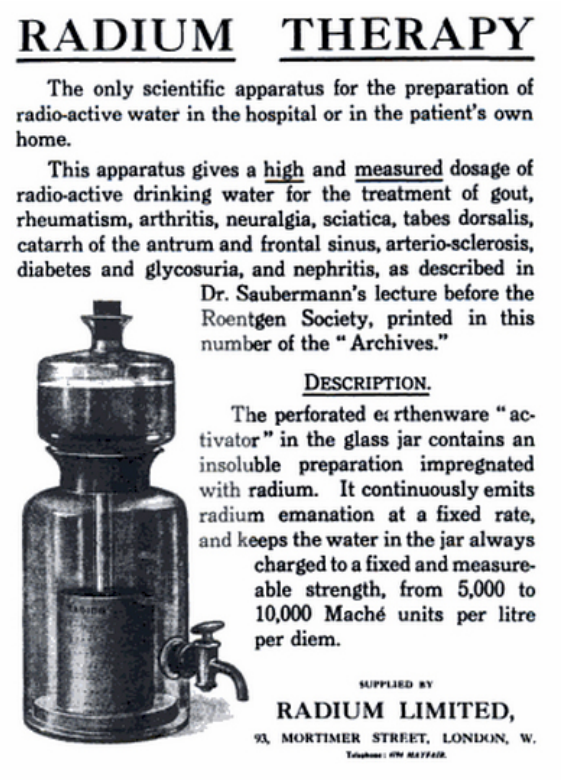


Figure 1.1: Advertisement for a scientifically developed radiation emanation activator. This particular device is suggested for use by Augustus Callé in a textbook on post-graduate medicine. [1]

The results of R.R. Wilson were adopted a few years later in the Lawrence Berkeley Laboratory (LBL) in California. Here, in 1952, Tobias, Anger and Lawrence published their article on biological effect of protons, deuterons and helium beams on mice that opened up the possibility to have human patients.
Between 1954 and 1957, 30 different patients underwent proton radiation therapy starting with large doses and then moving to fractionated delivery. From 1955 there was a flourishing increase in the number of laboratories treating protons beam leading to many technical and treatment planning improvements in proton therapy.[5] [6][7]
Till the 1990 the proton therapy was held in research institutes having only a limited number of patients. In that year, the Loma Linda University Medical Center (LLUMC) in California was the first hospital-based facility to be built. Clinical implementation has dealt with higher facilities costs. Considering capital investment ad operating costs the proton facility results more expensive than photon therapy by a factor of 1.7-2.4 . [8]

According to the data collected by the Particle Therapy Co-Operative Group (PTCOG) the US are steal pioneers of such technology registering the highest number of facilities and the highest number of patients cured till December 2018. Another peculiarity is the number of new facilities that have been built in the early 2000s that better highlights how such technology moved from being a questionable research topic to an essential practice.

Country	First Patient	N° of Facilities	N° of Patients
Austria	2017	1	297
Belgium	1991	1	21
Canada	1995	2	204
Czech Rep.	2012	1	3551
China	2004	2	1567
England	1989	1	3450
France	1991	2	15870
Germany	1998	6	2001
Italy	2002	3	1737
Japan	1979	15	28943
Poland	2011	1	394
Russia	1967	4	7139
South Africa	1993	1	524
South Korea	2007	2	3750
Sweden	1957	3	2185
Switzerland	1984	2	8824
Taiwan	2015	1	1695
USA	1954	31	90922

Table 1.1: Total Years of Operation and Patients Treated with Protons Worldwide (as of 12/2018). [9]

1.2 Basic physics principle in particle therapy

In order to understand the strength of radiotherapy it is necessary to analyze how cancer treatment can benefit from radiations. To start, we need to specify that term radiation refers to ionizing radiations: an high-energy type of radiation that is able to separate an electon from its nucleus generating ions. This kind of radiation can lead DNA chains of tumor cells to break through a Double-strand break (DSB) plus other indirect effects thus reducing its size and proliferation till complete repression.

Effects on DNA is different beween Tumor cells and Healty cells.

In Figure 1.3 is possible to distinguish the effectivness chance, in percentage, of DNA brake

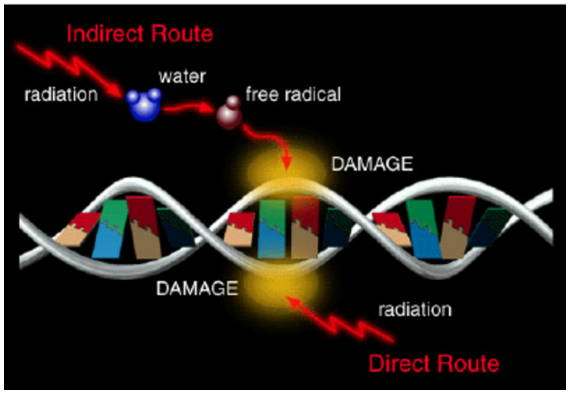


Figure 1.2: Direct and Indirect DNA damages due to radiation [10]

in tumor cells (blue line) and healty cells (red line). This defines a *Therapeutic window* for the Effective dose for high effectivness in tumoral DNA strands rupture keeping low the amont of healty cells involved.

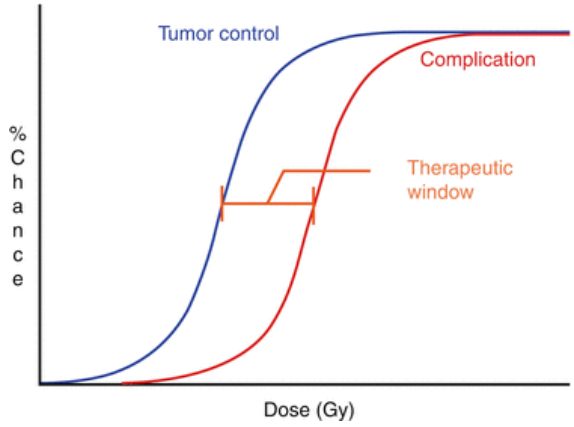


Figure 1.3: Therapeutic window as a function of Dose [11]

This can seem a very straight forward process but it is important to evaluate a very precise treatment planning that present several drawbacks that need to be taken into account. First of all the exposition to ionizing radiation can not be too long in order to encounter several dose limits defined by regulations so different sessions are needed. During the time between sessions, several mechanism of DNA repair are ignitied and cell cycle is stopped. If DSB repair capability is strong enough to complete the procedure before the following radiation delivery, DNA evolves a certain radiation resistance allowing cell

replication that instead is blocked for an effective Radiotherpay treatment.
Another important aspect that needs to be taken into account is the interacion with matter of the radiation. In partiular, in Particle therapy, it refers to an external beam of charged and/or neutral particles at high energy that are injected within the body to reach the specific tumor location. In the path inside the body, particles undergo several interaction with matter and the way they behave is defined by different cross sections that is a measure of likelihood of a specific process to happen after a collision. Even though we refer to different mechanisms, such as collision or Bremsstrahlung, what is important at this stage is the energy lost by the beam from source through the whole body.Iit is possible to introduce the concept of *Stopping Power* that is defined, according to the International Commission on Radiation Units (ICRU), as the average energy dissipated by ionzing radiation in a medium per unith path length of travel of the radiation in the medium. Being E the loss of energy and x the unit path length the linear stopping power can be mathematically defined as:

$$S(E) = \frac{\mathrm{d}E}{\mathrm{d}x}$$

that is divided by the density of the materials in order to obtain mass stopping power that is usually found in $MeV/(mg/cm^2)$. Throguh stopping power definition it is possibile to evaluate the Bragg-curve that helps in understanding the man differences among different Radiotherapy options.

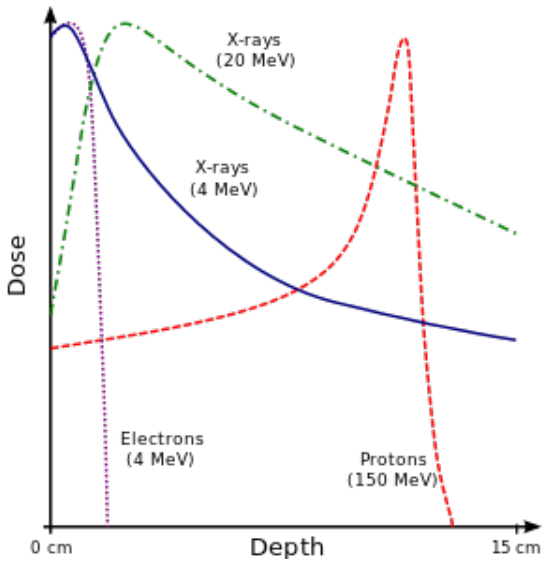


Figure 1.4: Dose deposition for different radiation types[12]

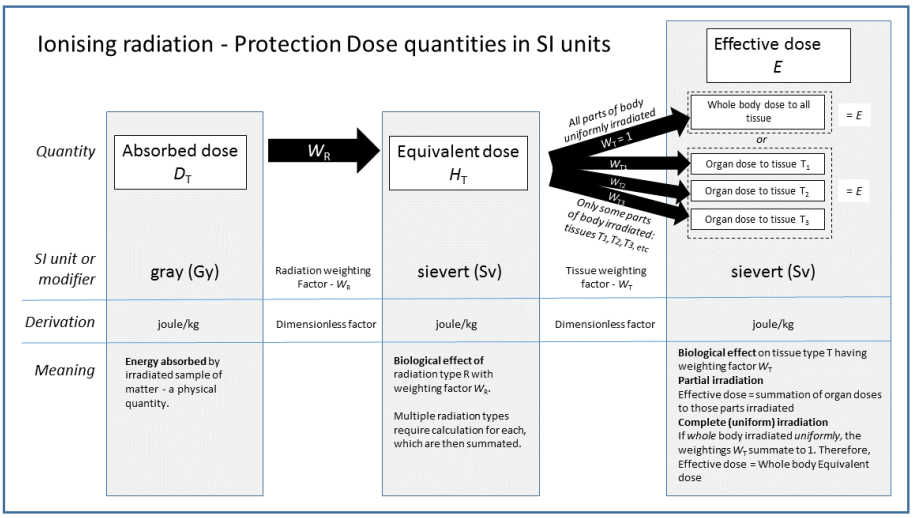
From Figure 1.4 it is possible to understand the difference among different particle

used. The most important aspects that can be visualized are the depth of different beams depending on their energies but also the particular behaviour of protons that present a characteristic peak just before resting. Such peak in energy deposition occurs just before the particles stops definitively due to main factors:

- Cross section increases as the particle energy decreases.
- Energy lost is inversely proportional to the velocity of the particle squared.

To end this general overview among the scientific bases sustaining particle therapy a few definitions in radiobiology must be spelt out:

- *Absorbed Dose*: is the energy released per unit of mass. The effect of dose is not to increase temperature of the body but to break DNA strands. It is measured in milliGray, mGy.
- *Equivalent Dose*: is the dose weighted on the different type of radiation. IN fact it is the Dose times a specific factor that has has a reference value 1 that is related to a standard X-ray radiation of 200 keV.It is measured in milliSievert,mSv.
- *Effective Dose*: is the effective dose weighted on different tissues trough specific weighting factors. It is measured in milliSievert,mSv.



22

Figure 1.5: Different Dose quantities in SI [13]

- *Linear Energy Transfer or LET*: is the amount of energy that an ionizing radiation transfers to the crossed material per unit distance. It is possible to dived radiations in low-LET and high-LET. The latter one is the more distructive to biologcal material , its effectiveness does not depend on the time or the stage in the life cycle but the effects on the healty tissues is less controllable.

- *Relative Biological effect or RBE*: is a comparison measure for the effectivness of two different ionizing radiation given the same amont of absorbed dose. The RBE for radiation R on a tissue is defined as:

$$RBE = \frac{D_X}{D_R}$$

where X is a reference absorbed dose of radiation of a standard type of radiation.

Chapter 2

Monte Carlo Simulations

Monte Carlo methods refer to computational algorithms in the statistic field that are widely used to solve different kind of problem by mean of probabilities. In this chapter a complete overview on Monte Carlo Simulations is presented.

2.1 History of the Monte Carlo method

The first usage of random sampling to solve mathematical and physic problems refers to the *'Buffon's needle problem'* in the 18-th century and it was proposed by Georges-Louis Leclerc, Comte de Buffon.[14]
The statment of this statistic problem was:

> *Suppose we have a floor made of parallel strips of wood, each the same width, and we drop a needle onto the floor. What is the probability that the needle will lie across a line between two strips?*

By mean of integral geometry it turns out that, for a needle of length l that is lower than the strip width t the requested probability is:

$$p = \frac{2}{\pi} \cdot \frac{l}{t}$$

Combining this information with several experiment it is possible to evaluate π with very good accuracy.
In 1901, Italian mathematician Mario Lazarini performed Buffon's needle experiment with 3048 needle tosses tat lead to a consistent approximationaccurate to six significant digits:

$$\pi \approx \frac{355}{113}$$

Another example of a Monte Carlo algorithm performed before its rigorous definition was made refers to the FERMIAC machine.
It was an analog computer developed by *Enrico Fermi* and his group *'I Ragazzi di Via Panisperna'* that was able to model neutron transport giving an initial distribution of neutrons. The model was able to recreate the neutron behavior by mean of a pseudo-random number generation used to determine the following action of the neutron, including scattering and fission.
The instrument was equipped by a series of drums that were adjustable according to the simulated material to be crossed and different settings were available for fast and slow neutrons. The result of the machine was a 2D plot of random walks of slow and fast neutrons in different materials.

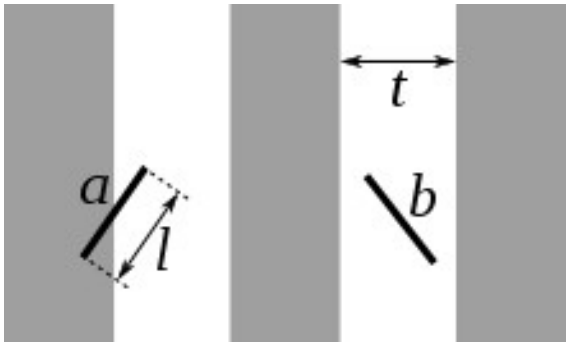


Figure 2.1: Example of two different experiments of Buffon's needle problem [15]

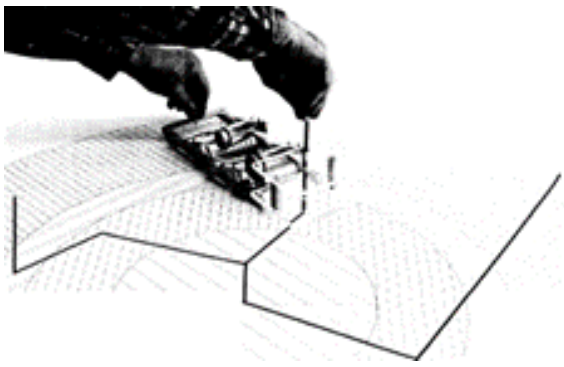


Figure 2.2: Example of FERMIAC utilization [16]

The rigorous implementation of this class of algorithms goes back to the 1945, in the Los Alamos Scientific Laboratory by an intuition of Stanislaw Ulam that, was working on Nuclear Weapons and Shielding. Such intuition come outside of his studies, during an

illness period playing solitaire.
As reported by Roger Eckhardt [17], Ulam described its intuition in this way:

> *The first thoughts and attempts I made to practice [the Monte Carlo Method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later ... [in 1946, I] described the idea to John von Neumann, and we began to plan actual calculations.*

The basic principles of the Monte Carlo methods were formalized by von Neumann and Ulam and the project was marked as secret. The choice of the name, after the suggestion of the physicist Nicholas Metropolis, is a clear reference to the Monte Carlo Casino in Monaco where the game of chance and statistics play a key role.
The strength of this class of stochastic algorithms was enhanced by the usage of the Electronic Numerical Integrator and Computer (ENIAC) [18], developed in the same Laboratory, that opens up the possibility to computerize Monte Carlo Simulations reducing computational time and cost. The first ever Monte Carlo Simulations was ran in April 1948 in Los Alamos Laboratory by John and Klara von Neumann and Nick Metropolis.

2.2 Basic Principle of Monte Carlo Simulations

Monte Carlo simulation is a stochastic type of simulation that takes advantage of random sampling and statistical analysis to evaluate results. The simulation consists in the study of the behaviour of the analysed phenomena by representing all different outcomes in a controllable context.
The main idea is to simulate different experiment and associate to them a random variable that can be defined as a measurable function defined on a probability space that maps from the sample space to the real number.

After several simulation all the possible outcome are stored so that it is possible to map the distribution of the outcomes using definition of 'sample average', 'mean expected value' and 'variance' that distribute the result in a probability density function.

Some definitions:

- ξ_i is the random Variable associated to the i-th experiment;
- $\overline{\xi_i^N} = \frac{1}{N} \cdot \sum_{i=1}^{N} \xi_i$ is the sample average that represents the 'mean value' of the outcomes distribution.
- $\sigma^2[\xi_i^N] = \frac{\sigma^2[N]}{N}$ is the variance of the sample average. It is an informal measure of how far the outcomes spread out from their mean.

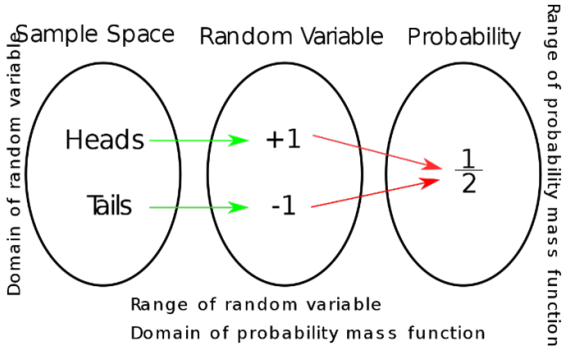


Figure 2.3: Example of random variable definition [19]

A proof of convergence of this method comes from the *Central Limit theorem (CLT)*. It states that a random variable of a statistical phenomenon, described by independent variables according to particular distributions, will be normally distributed for a large number of experiments. The convergence of the sample average is a form of the so-called *weak law of large numbers*. The distribution of the sample average will be:

$$f_{\overline{\xi}^{(N)}} = \frac{1}{\sqrt{2\pi\sigma^2/N}} \cdot e^{\frac{-(x-\mu)^2}{2\sigma^2/N}}$$

Through some algebra it is possible to retrieve important informations on the results. In particular it is possible to evaluate the probability of a relative error smaller or equal to the relative standard deviation for the error bar generation:

$$P\left[\left|\frac{\bar{\xi}^{(N)}-\mu}{\bar{\xi}^{(N)}}\right| \leq \frac{\sigma}{\sqrt{N}\left|\bar{\xi}^{(N)}\right|} \cdot k\right] =$$

• 0,68 for k=1; • 0.95 for k=2; • 0,99 for k=3;

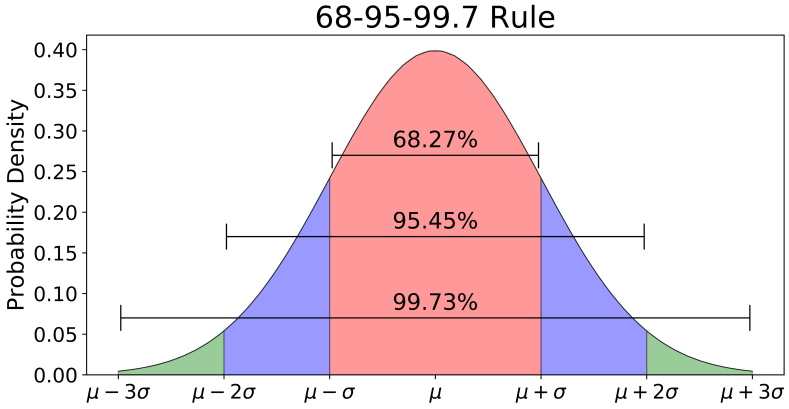


Figure 2.4: Normal distribution of the sample average [19]

According to what has been described so far, Monte Carlo seems very suitable for physics problems even more than deterministic approach because of several strengths:

- Probabilistic results: results are a combination of possible outcome and its likelihood
- Sensitivity analysis that gives a simple correlation between different input parameters and how they relate to the results.
- Graphical representation of the results can be easily retrieved by the simulation for synthetic and faster communication of the outcomes.until

Main drawbacks of this stochastic model refers to computational time and cost of resources for a sufficiently large amount of experiments. The implementation of optimized code for Monte Carlo simulations is an hard challenge.

2.3 Monte Carlo Simulations for Particle Therapy

Going into details of some Monte Carlo method application there are several nuclear-related problem that is possible to take into account mainly due to the stochastic behaviour of

nuclear particles. In particular we will focus on particle therapy that is a form of radiotherapy involving external beam of energetic particles such as neutrons, protons or other heavier positive ions for cancer treatment. The basic principle is to use ionizing radiations to damage DNA of cancerous tissues for cellular death by preserving healthy tissues. What makes particle therapy more attractive than x-rays treatments is the possibility to take advantage of the Bragg peak in dose deposition through the body thus reducing effects on surrounding healthy tissues. To optimize particle energy it is possible to broaden the energy range of the particles or installing different attenuators, such as ripple filters, to spread out the peak.

In order to achieve an accurate clinical dose calculation it is possible to model in a simulation code both the patient anatomy and the radiation source and develop several Monte Carlo simulation of different treatment plans in order to enhance accuracy in dose delivery [20] [21]. In the recent present different Monte Carlo-based treatment planning systems (TPS) have been tested and developed. The main idea is to simulate both radiation transport through the lineac and the dose delivered within the patient in a single simulation that allow a precise description of all the fundamental physics involved by usage of specific library and importing patient data by converting CT numbers into specific materials. [22] [23]

In the following chapter will be presented a detailed analysis of a Monte Carlo based toolkit for particle simulation with some applications.

Chapter 3

Software Packages Used

3.1 Simulation Implementation

In this section will be presented all the toolkit that have been used to develop the code for the presented simulation.

3.1.1 Geant4

Geant4 is a C++ based toolkit for Monte Carlo simulation of particle through matter providing complete functionalities for all simulation level.
Geant4, that states for GEometry And Tracking, is the latest version of this project coming from an international cooperation among CERN (Europe) and KEK (Japan) that merged their independent studies for the improvement of FORTRAN based Geant3.
The fundamental purpose of this joint venture was the development of a functional and flexible program for detector simulations. Earlier achievements opened up the possibility to widen such tool to nuclear, accelerator, space and medical physics community.
This toolkit is based on Object Oriented programming model that organizes software design around defined data field with their unique attributes, called objects that are grouped in classes. Geant4 provides different classes and objects for each step of the simulation to provide the developers a great flexibility in building applications. Another important feature is the great variety of materials and physics implemented within the toolkit for different fields of application and that are constantly kept updated. Geant4 acts as a repository for several data and expertise research projects about particle interactions. The strength of this toolkit relays on the transparency of the code, exploited through object oriented technology. Trough such programming model is possible to handle the complexity of the

different physics involved by defining a uniform framework for physics models recognition or implementation with little modification of existing codes. Nowadays Geant4 is, by far, the most used particle simulation toolkit in research also thanks to his full-free availability and the high number of developments offered among years. [24] [25]

Design Overview

Geant4 software relies on different libraries, each corresponding to a releasable component. All different components are individually managed by a working group of experts. In addition, thereare several working group for testing and quality assurance, software management and documentation management.
The modularity of this software is in the hierarchical structure of domain it presented. Each domain refers to several sub-domains in unilateral connection with no circular dependencies, each made up of different classes that are adjustable according the user needs. [26]
A brief overview among different domains is presented:

- Global is the bottom structure providing system units, mathematical constants, numbers and random number generator.
- Particles, Materials , Geometry refer to the build up phase of the simulation according to robust databases. As the scheme suggests particle and materials structures are fundamental for the geometry structure development.
- Track takes information on particle position at each time step that work as input for the following up domain.
- Processes takes Track information and simulates basic physical processes.
- Digits + Hits regulates particle interaction with sensible volumes.
- Tracking follows the whole particle path at each step that are collected in Event.
- Run refers to a collection of different Event sharing common beam and detector implementation
- At highest level are present different categories for connection outside the toolkit through abstract interfaces.

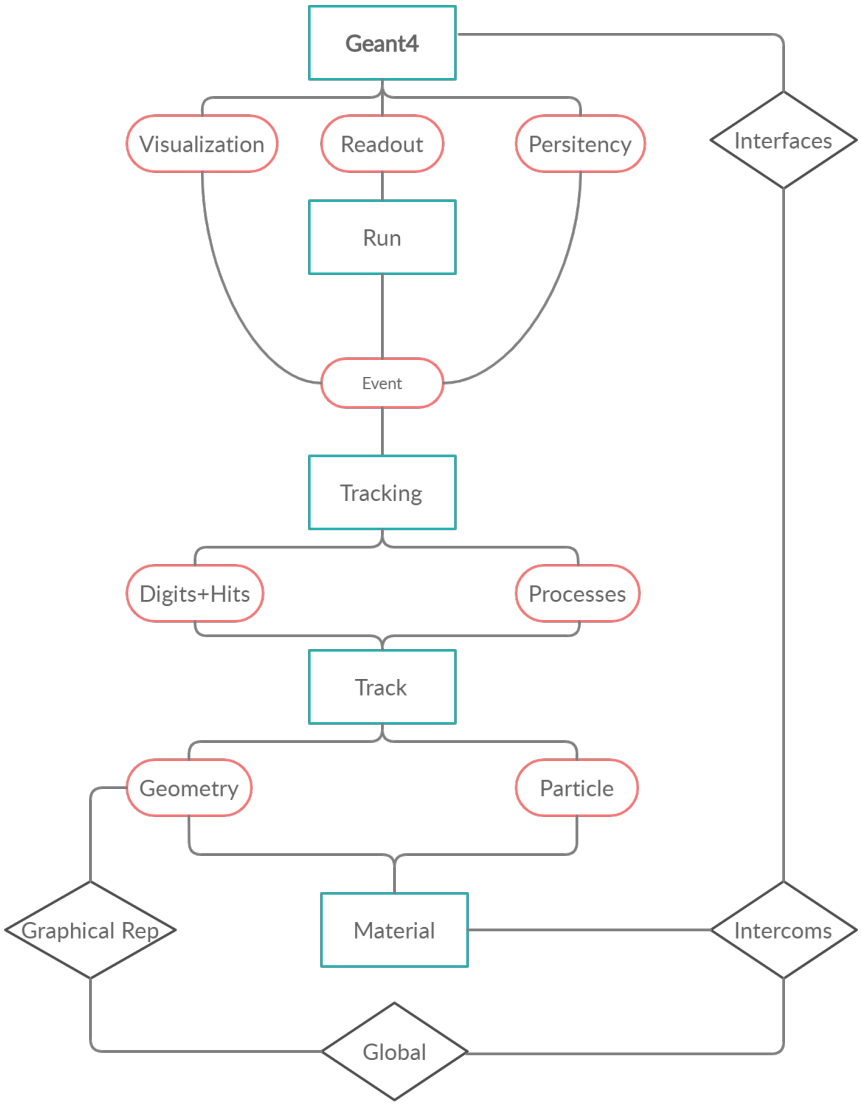


Figure 3.1: Scheme of different domains in Geant4

Simulation Structure

Entering in the detail of a complete simulation it is possible to define a perfectly organized work flow in a hierarchical thus modular structure that exploit implicit classes, directly defined in Geant4, and explicit classes that are user defined.[27]
A simplified scheme for a simulation can be structured as follows:

- Detector Construction: is the class responsible for the setup of the detector in its geometry and materials. At this stage sensitivity and visualization attributes of the detector are defined.
- Physics List: is the class that handles all the physics aspects of the simulation. Geant4 offers different modelling algorithm depending on the energy range of the simulation and with recommended lists for specific physics tasks.
- Primary Generation Action: creates an instance of a primary particle generator by describing initial state of the primary event.

These three classes are the only mandatory ones, also referred to as User Initialization Classes.

- Slice SD: is responsible of build up the sensitive detectors. Such sensitive detectors are specific voxels, logical volumes, from which information are collected.
- Slice Hit: collects information on which a particle/particle beam interacts with matter
- Run Action: is responsible for data initialization and collection of statistic during simulation perform.
- Event Action: is a dynamic class that follows a particle in its entire life to collects results at the end of each particle path.
- Macros: are external files whit respect to the code that can be used for the run of the same simulation with different input data that can change class values

3.1.2 ROOT

ROOT is a software framework for data analysis, developed by CERN for particle physics and afterwards implemented for other applications such as astronomy and data mining. As for Geant4, also ROOT replaces some program libraries writtern in FORTRAN to move

forward to a more efficient and flexible C++ language with all the advantages of an object-oriented program.
For the data analysis ROOT provides:

- Different classes that can be directly implemented within the simulation code for histogramming and graphing, that can be all stored in different file format.
- A graphical interface with different features, such as interfacing Monte Carlo event generators.
- CINT, a C++ interpreter that in the latest versions is back to the original inventor, and replaced by the CLING interpreter.

This software is not necessary to build and complete the simulation but it is very useful for a easier summary of the result with different output files that simplifies the communication of results.

3.2 Code Analysis

Once the simulation have been validated theer is plenty of room for optimization. First of all, a complete analysis on the code is important before getting lost in all the different sources and headers to better understand specifically where we need some hard programming in order to implement better solutions.

In software engineering there have been different attempt to perform a dynamic program analysis of either the program source code or its binary in order to 'profile' the usage of particular instructions and the complexity, in time and space (memory), of a program. [28] [29][30]
To choose the profiler it is important to have a bit of knowledge on how different profilers apply on a code and which is the output required.

- *Tracing Profilers* intercepts calls to functions within the application that are wrapped inside extra code that extracts the needed information and are collected in a trace file that is available for post-processing and displaying. While running the code within the profiler framework it will result in an increase of the runtime for the execution of such extra-instructions. The consistency of the strong deterministic results is strongly affected by several drawbacks such as the code implementation needed and the different in reading the output files that refers to addresses and other machine information that are not easily readable.

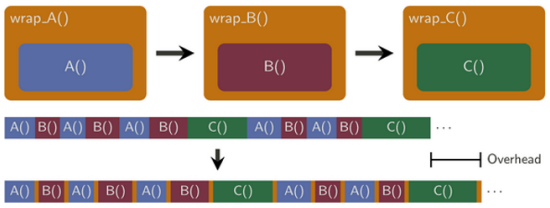


Figure 3.2: Tracing Profiler example [31]

- *Sampling Profilers* do not intervene directly on the code, instead it exploits the operating system periodic inspection on the application profiled. The strength of this kind of application is the simplified intervention without any modification of the source code, instead the code profiler periodically requests the OS kernel to return information on the investigated application. The advantages of this kind of profiling technique is in the collection of information according to functions names, the absence of dependency on the language of the code and the size of the overhead is instead smaller with respect to Tracing profilers and it is dependent on the sampling rate chosen. Drawbacks refer to the output that is a statistical representation of the application rather than the deterministic one that can be provided in the Tracing case.

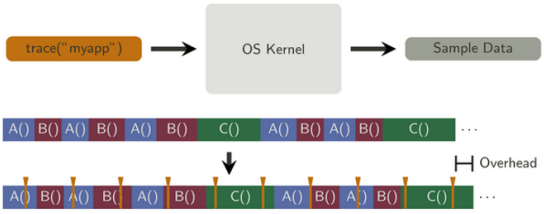


Figure 3.3: Sampling Profiler example [31]

- *Hybrid Profilers* are the most common approach in High Power Computing (HPC) codes that combine both tracing and sampling methods to exploit the main features of both reducing the inefficiencies.

3.2.1 Valgrind

Valgrind is the programmable framework chosen for general supervision of the code in analysis, in particular for the profiling.[32][33] The strength of this meta-tool is the possibility to use dynamic binary translation to control any aspect of the code and acting directing on the executable it has two main advantages:

- Full coverage of all sources and libraries, even if they are not available.
- No recompilation or relinking is needed when Valgrind is called.

The name Valgrind is a clear reference to the Vallahalla main entrance from Norse Mythology. Valgrind original intent was to be a free memory debugger that evolves in a general, broader, framework in dynamic analysis especially for Linux users.
At this stage of the toolkit evolution it is possible to describe it as a virtual machine that benefits from just-in-time (JIT) compilation techniques and dynamic recompilation.
Before entering in the detail of the different analysis and the compilation procedure some technical term must be spelled out:

- Executable or Binary file is the output file of the code compilation and it is the collection of all the task to be performed by the computer at the execution stage.
- JIT: Just-In-Time compiler, also known as dynamic translation, runs after the program is started and compiles the bytecode into machine code instruction of the running machine so that the simple object code can be called, ideally reducing the recompiling inefficiency. In general, a JIT compiler continuously analyses the code executed and identifies the speedup gained.
- Dynamic recompilation (DRC) is a feature of some virtual machines that allows the recompilation during execution to reflect the run-time environment obtaining information on efficiency not available in traditional compilers.

What Valgrind does is straight forward: at first translate the program in a temporary Intermediate Representation (IR) that is a processor-neutral. At this point different actions can occur depending on the chosen tool, among the ones offered by Valgrind. Once the toolkit is called, it performs different analysis on the IR before translating it back in machine code to let host processor run it. Once the code is recompiled on the host also a GNU Debugger (GDB) targets the running program. At the end of the compilation a substantial difference in the execution time with a reduction to 20-25%of the speed, due

to several transformations that are made among the execution.
A short description of the different tools supplied by Valgrind is now presented:

- *Memcheck* is the default and most used tool that works as a tracing profiler especially for memory error detection. This tool repleaces the standard C memory allocator with its own implementation to keep track of error in reading or writing outside allocated blocks.
- *Cachegrind* is a profiler specific for the cache with its own GUI KCacheGrind.
- *Callgrind* generates and analyze a control flow graph for different correlation among subroutines. It has some overlapping with Cachegrind but it has also some extra inforamtions. All the output can be visualized within the same GUI.
- *Helgrind* and *DRD* detect race condition in multithread codes with different analysis techniques leading to possible different results.
- *Massif* is a heap profiler that analyze the memory data structure in a sampling profiler approach with its own GUI massif-visualizer.
- *DHAT* different kind of heap profiler that outputs detailed pattern of memory usage.
- Other experimental tool are implemented but are less used in code profiling.

Chapter 4

Simulations

In this chapter willl be presented different simulations of particle transport with increasing complexity. After a short intriduction of their main features, a profiling analysis has been developed and summarized to circumscribe the part of the codes that offer broader room for improvment and optimization.

4.1 Bragg Peak

As already explicited in chapter 1, proton therapy is gaining increasing importance thanks to the stopping power evolution that presents a peak, the so-called Bragg Peak, for localized energy deposition. At this stage a Geant4-based simulation of a Bragg Peak is presented. The code in analysis describes a particle beam and a water slab as a target. The simulation can be tuned in order to set different aspects through a macro:

- *Number of particle simulated*: This aspect will affect the accuracy of the simulations but also require more computational time and memory.
- *Detector dimension*: In term of depth of the slab, it affects the memory allocation at the Detector Construction stage.
- *Energy of the beam*: It will affect the depth that the beam can reach thus the location of the peak.

This list of features refers to the one that have been tuned to outline a general correlation between them and the simulation performances. Other properties of the simulations have been kept constant such has the physics model involved, distance of the particle source

with respect to the target. In the following subsections the dependencies among the listed attribute is presented in term of memory usage.
The same simulation have been made with dfferrent machines equiped with different CPUs with different techincal specifications, listed in Table 4.2, and the results have been commented in section 4.1.4.

	Intel(R) Core(TM) i7-6700K CPU	Intel(R) Core(TM) i7-8750H CPU
Clock Speed	till 4.20 GHz	till 4.10 GHz
Architecture	× `86_64`	× `86_64`
CPU(s)	8	12
L1d cache	32K	32K
L1i cache	32K	32K
L2 cache	256K	256K
L3 cache	6144K	9216K

Table 4.1: Different CPUs specifications

4.1.1 Number of particle simulated

At first, the number of tested particles has been changed keeping a constant value of the energy of the particle beam set at $150MeV$ with a slab depth of $20cm$. As expected the results become smoother at increasing number of simulated particles, with a substantial noise reduction in the result.
Following Figures 1-3 shows the different curves obtained with values of the simulated particles being 10^4, 10^5 and $2.5 \cdot 10^5$.
The execution of the simulation has been made with the *Massif* option of *Valgrind* toolkit that enriches the result of the simulation with a plentyy of information on code performance, all collected in `.dat` files that have been post processed to make intelligible through a series of graphs and tables.
In Figure 4.4 is presented the execution time as a function of the simulated particles and as expected it increases in a superlinear scalability due to the higher number of secondary particles that are generated increasing the number of simulated particles. It becomes relevant that number of particle number must not be extremely large to have a faster code without loosing accuracy for not enoguh iterations. At this stage of research there are different numerical formula to evaluate the exact number of iterations that has to be done, no one of them is presented due to the absence of validation in the field of particle simulation but are, as well, collected in the bibliography[34][35].

N° of particles	Memory Heap [MB]		Extra Memory Heap[MB]		Percentage	
	8 cores	12 cores	8 cores	12 cores	8 cores	12 cores
10^4	41.36	41.76	3.53	2.89	8.53 %	6.93 %
10^5	157.24	156.96	7.70	5.27	4.9 %	3.36 %
$2.5 \cdot 10^5$	348.96	345.51	14.62	9.09	4.2 %	2.63 %

Table 4.2: Memory informations for a different number of simulated particles

Another aspect analyzed is the the memory allocation. In Table 4.2 the heap memory allocated for each simulation is presented and especially the extra memory heap allocated that rappresented a surplus of memory that can be avoided for better performances. The peculiarity is that the relative weight of the extra memory with respect to the total memory allocated reduces increasing the number of simulated particles. Further more, analyzing the output through the *massif-visualizer*, it is possible to address the increasing amount of extra memory at the stage of *PrimaryGenerator* of the Geant4 toolkit. A wiser analysis of the memory optimization in a code is presented at chapter 6.

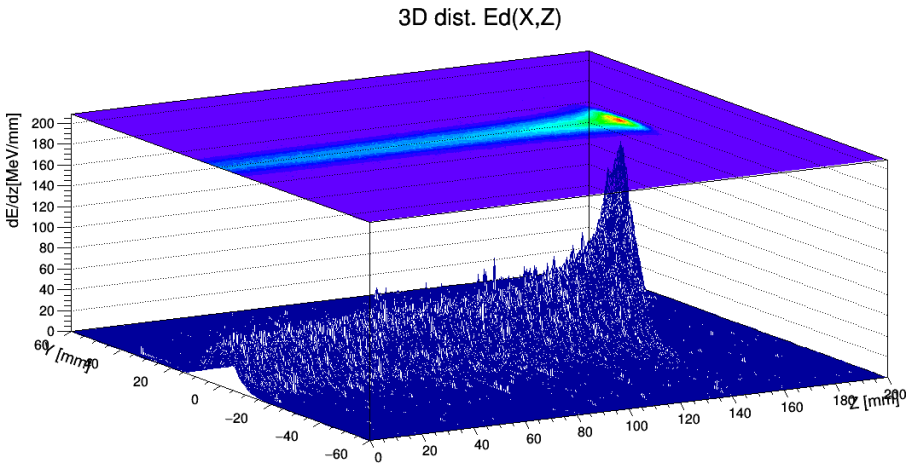


Figure 4.1: Bragg Peak with n= 10^4

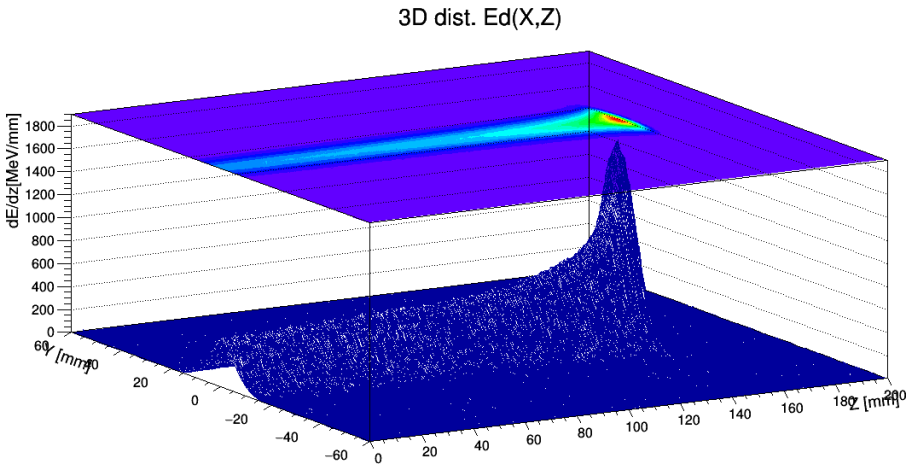


Figure 4.2: Bragg Peak with n= 10^5

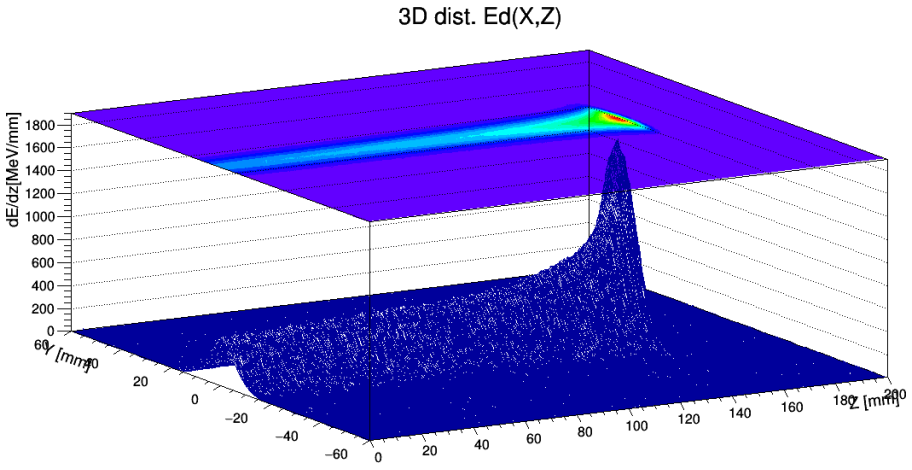


Figure 4.3: Bragg Peak with n=$2.5 \cdot 10^5$

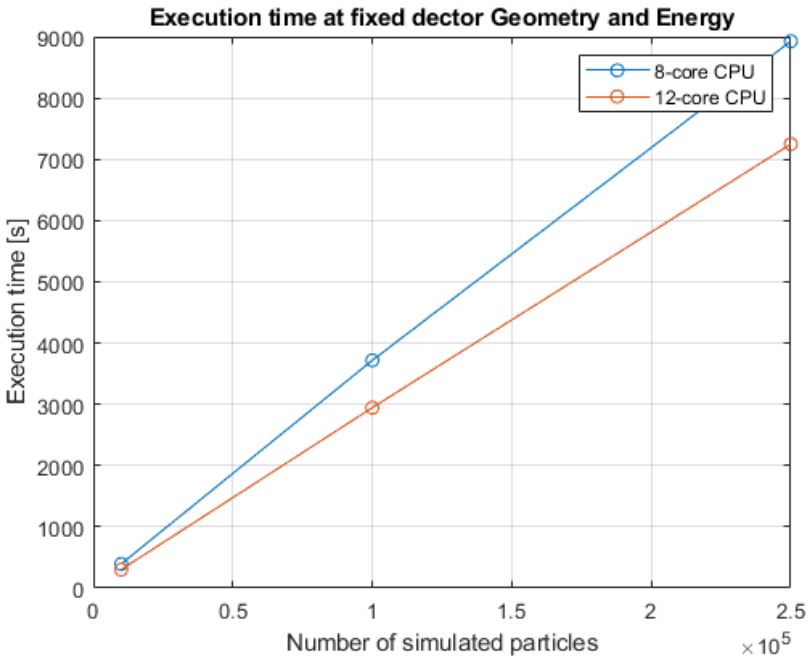


Figure 4.4: Execution time function of the simulated particles

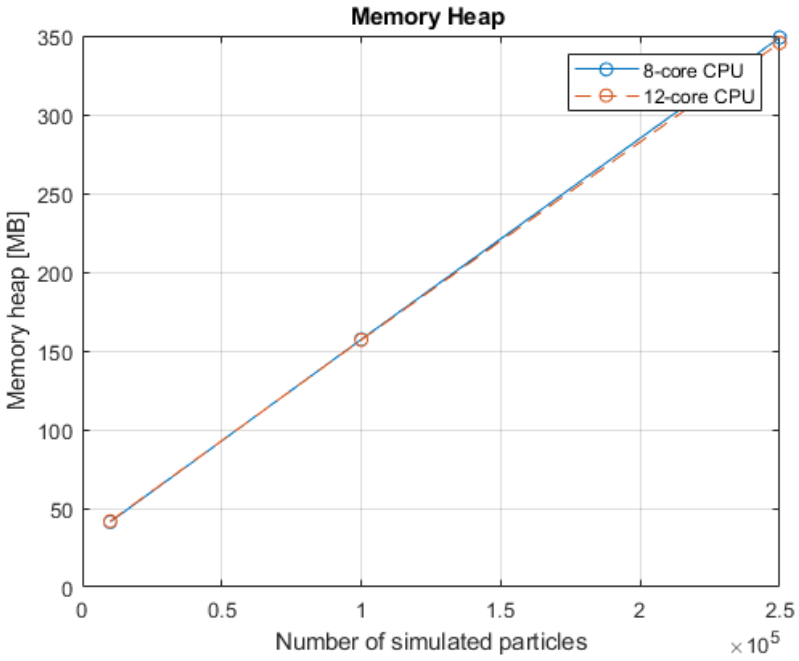


Figure 4.5: Memory heap function of the simulated particles

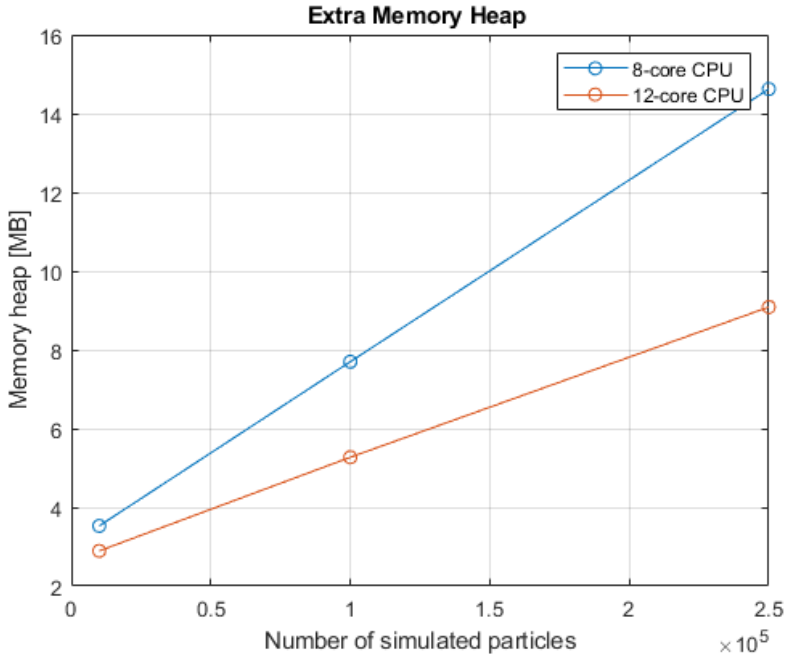


Figure 4.6: Extra memory heap function of the simulated particles

4.1.2 Detector dimension

Fixing the number of simulated particles at 10^5 particles and the energy to $150MeV$ it is possible to evaluate the effect of the allocation of bigger detectors in memory, knowing that part of the detector will rieceve no effect of particles being too far away from the peak. Following Figures 4.7-9 shows the different curves obtained with values of the slab depth $20cm$, $50cm$ and $100cm$. In Fig.4.10 is presented the execution time as a function of the detector sizes and it increases with size of the detector but presenting several seconds of difference that, keeping constant the physics of the problem, can be all addressable to the allocation of the detector within the memory.

Detector size	Memory Heap [MB]		Extra Memory Heap[MB]		Percentage	
	8 cores	12 cores	8 cores	12 cores	8 cores	12 cores
$20cm$	157.24	156.96	7.70	5.27	4.9 %	3.36 %
$50cm$	163.76	163.95	7.84	5.33	4.79%	3.25%
$100cm$	173.40	169.55	7.98	5.43	4.6 %	3.2 %

Table 4.3: Memory informations for a different detector sizes

In table 4.3 the memory information outputed by *Valgrind* have been collected showing a rhougly increment of 5% of the allocated memory doubling the detector size at each simulation. For the Extra memory heap allocated the percentage is almost constant with a negligible (∼- 0.1%) reduction increasing the detector size that results in a negligible influence of the Geant4 function *DetectorConstruction* on the extra memory allocation.

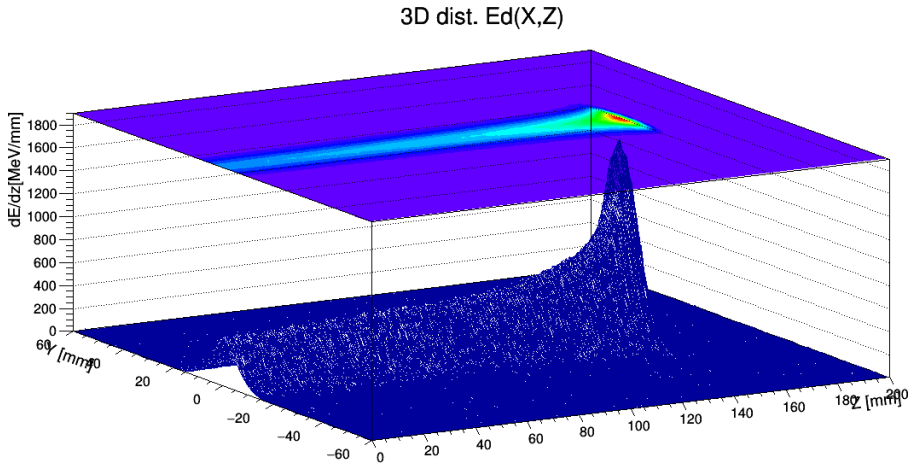


Figure 4.7: Bragg Peak with detector of $20cm$

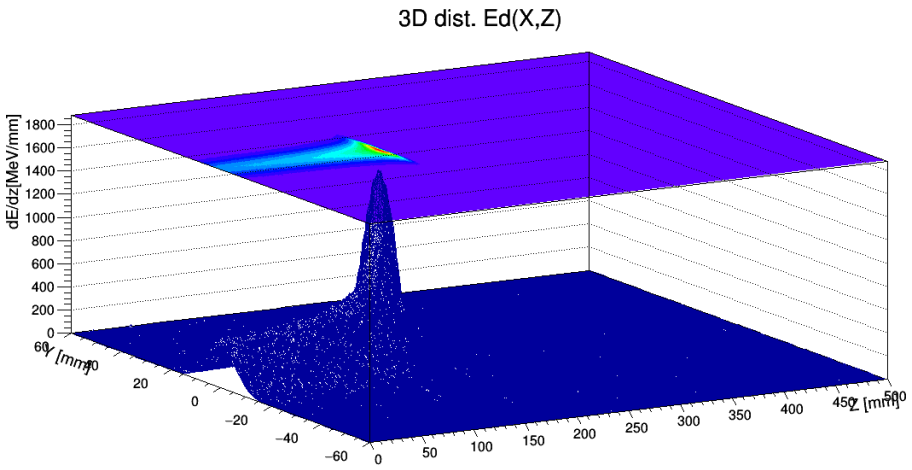


Figure 4.8: Bragg Peak with detector of 50cm

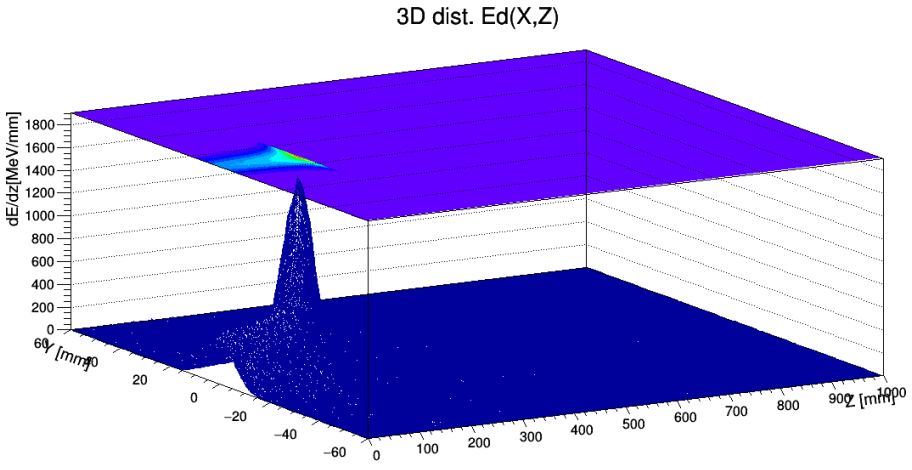


Figure 4.9: Bragg Peak with detector of 100cm

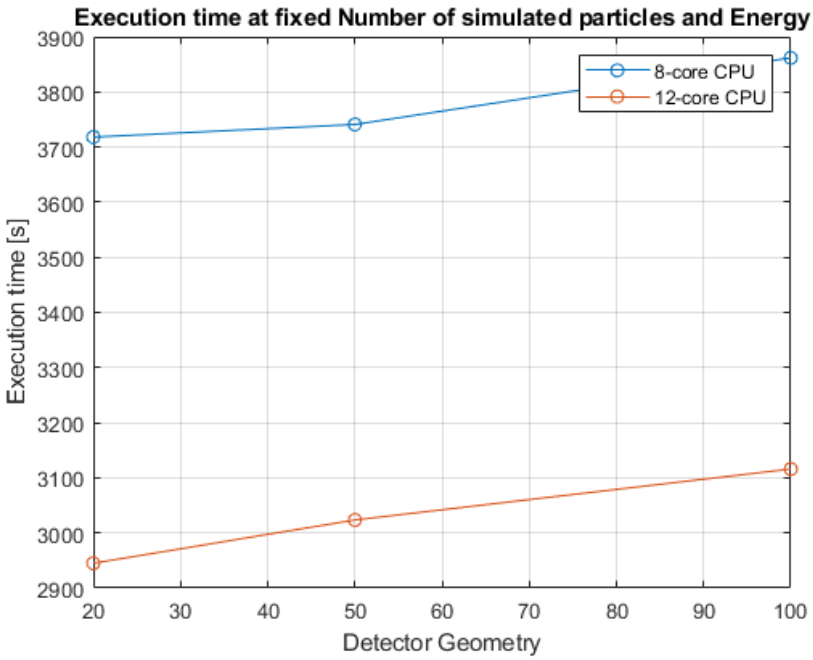


Figure 4.10: Execution time function of the detector size

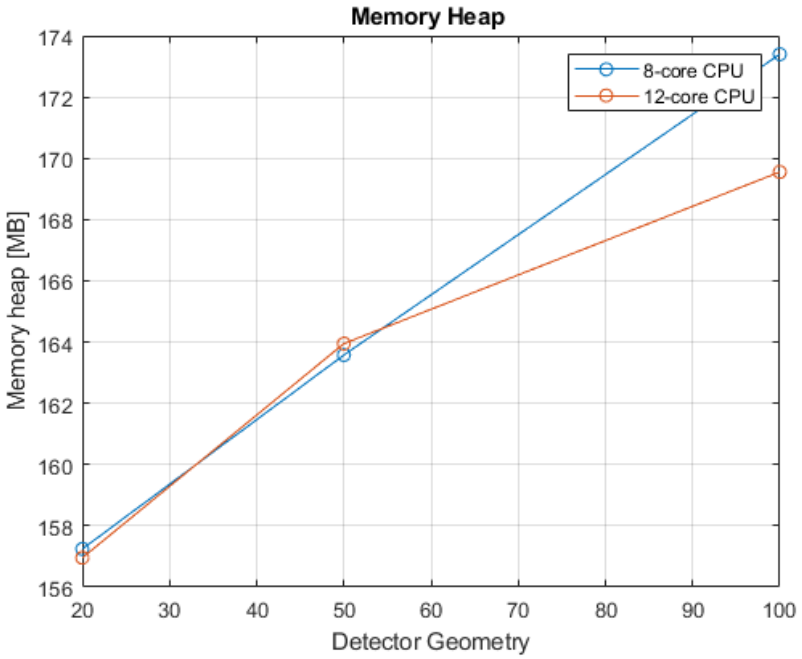


Figure 4.11: Memory heap function of the detector size

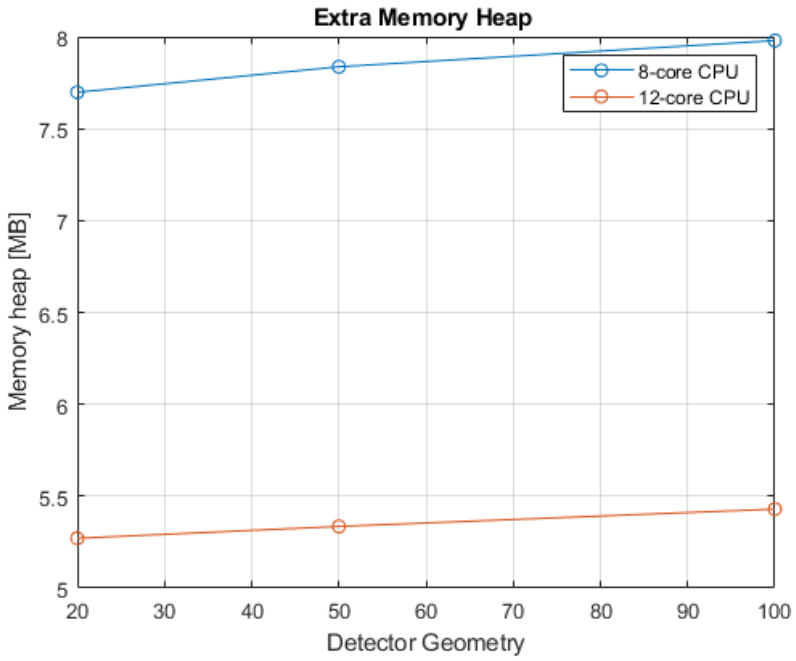


Figure 4.12: Extra memory heap function of the detector size

4.1.3 Energy of the beam

The former dependency addressed is the one on the energy of the proton beam that changes the depth of the Bragg Peak but also increases the number of secondaries generated within the material.
Following Figure 4.13-15 represent different solution obtain with energy $50MeV$, $150MeV$ and $300MeV$.
In this case the execution time has an exponential growth at increasing energy. This results is a direct consequence of the physics of the simulation that follows up a particle from its birth till death of itself and all the secondary generated passing within the matter. This is not a showstopper thus an opportunity to boost our code moving to parallel computing techiniques that find space in Chapter 5.

Beam Energy	Memory Heap [MB]		Extra Memory Heap[MB]		Percentage	
	8 cores	12 cores	8 cores	12 cores	8 cores	12 cores
$50MeV$	172.71	172.97	7.74	5.1	4.50 %	2.95 %
$150MeV$	168.56	169.55	7.98	5.43	4.73%	3.2%
$300MeV$	174.29	174.85	8.32	5.65	4.7 %	3.23 %

Table 4.4: Memory informations for a different detector sizes

As before, the output of the *massif* tool of *Valgrind* are presented and summarized in a table (Table 4.4) .

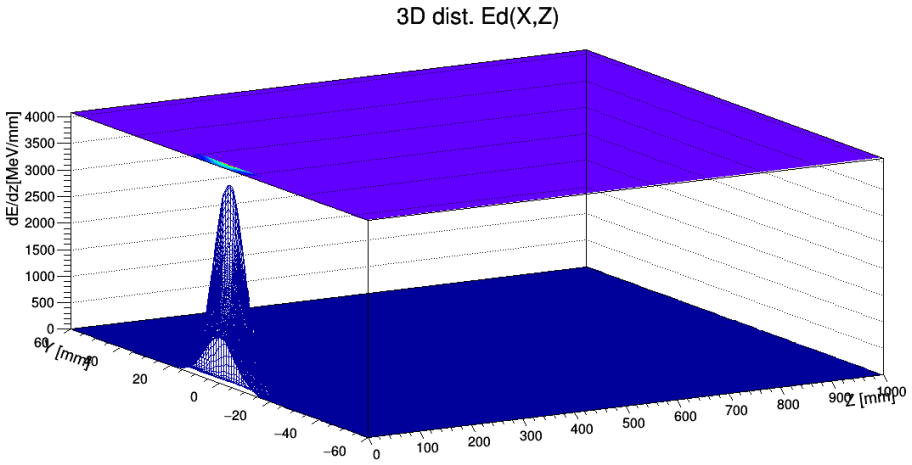


Figure 4.13: Bragg Peak with Beam Energy $50MeV$

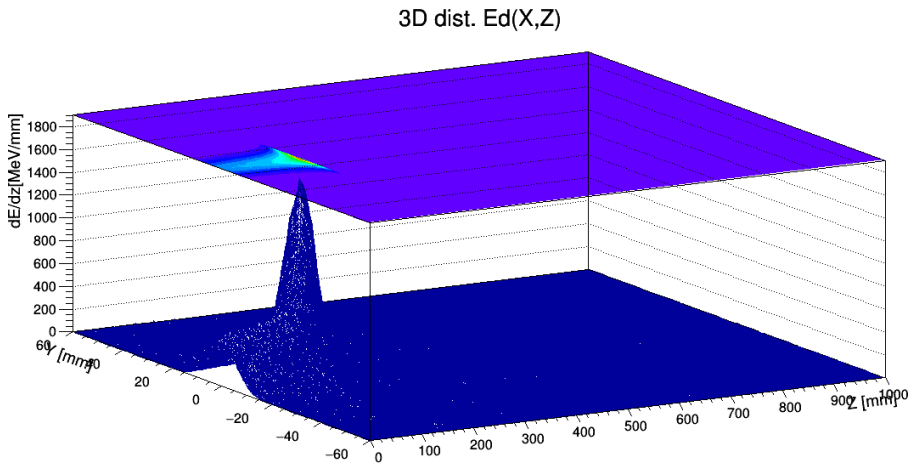


Figure 4.14: Bragg Peak with Beam Energy $150MeV$

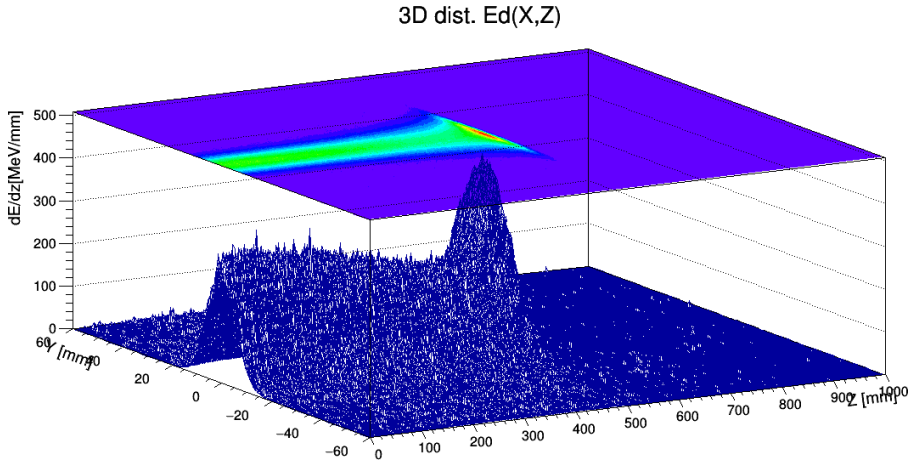


Figure 4.15: Bragg Peak with Beam Energy $300MeV$

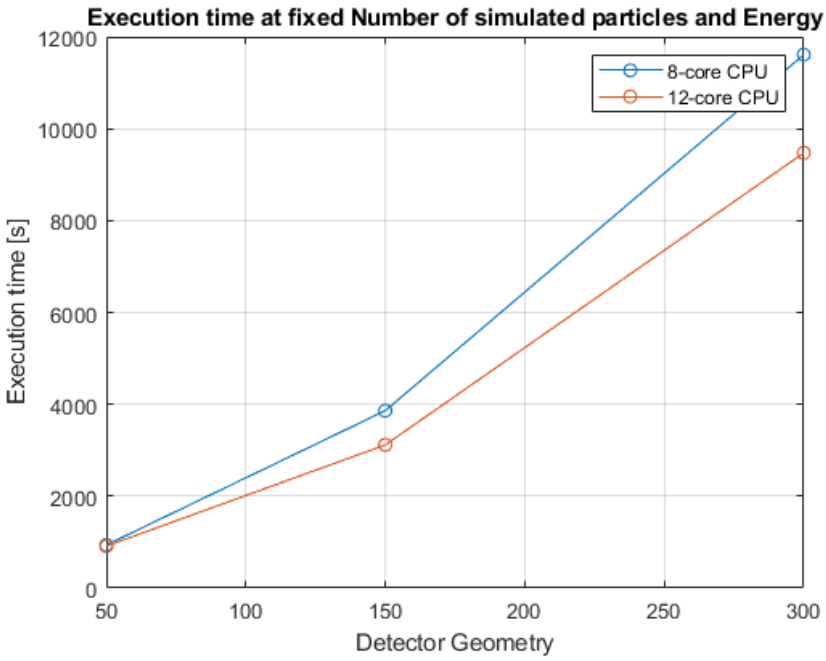


Figure 4.16: Execution time function of the beam energy

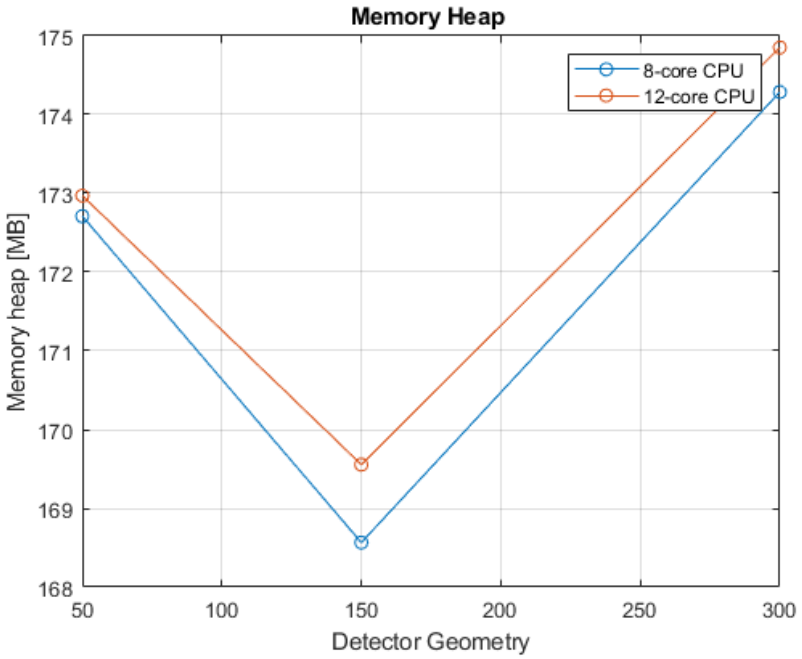


Figure 4.17: Memory heap function of the beam energy

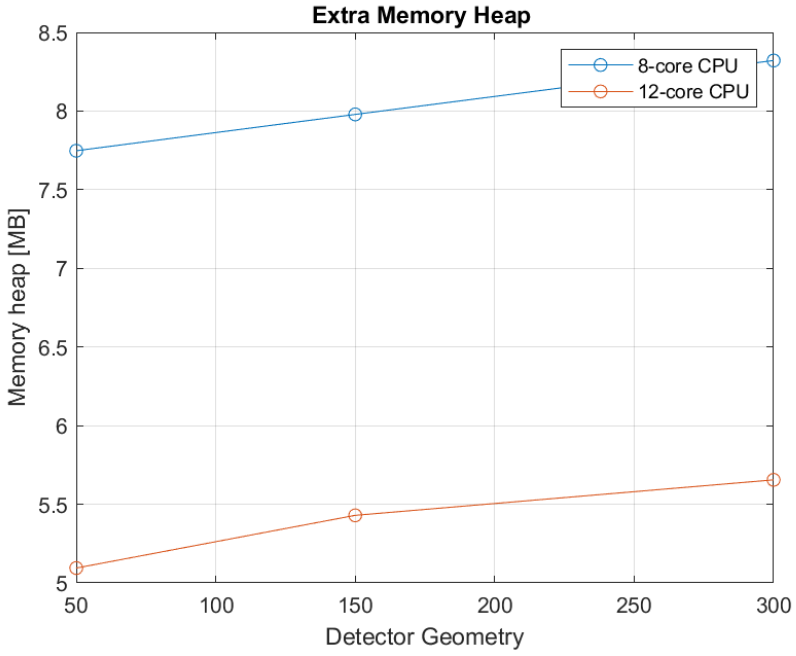


Figure 4.18: Extra memory heap function of the beam energy

4.1.4 Dependency on the Hardware specifications

In general the differences among the CPUs is in the number of cores and the cache memory. Cores difference is not a big deal because the analysed codes do not implement multithreading. What makes the difference in term of time and extra memory heap is in the size of the cache, in particular L3 memory. The CPU cache is related to the average cost, in term of time and energy, to access data from the main memory being located closer to the processor core it can store copies of data from frequently used main memory locations. It works in a hierarchical way and the main difference relies in the L3- cache which is present in medium-high level CPUs. With higher size of L3-cache in the 12-cores CPU the execution time is highly reduced but also the extra heap memory is reduced being able to free and re-allocate in a faster way. The Memory heap needed have no difference among different hardware because the executed codes are the same and also their results.

4.2 Clinical simulation

At this stage it is possible to move forward to a clinical simulation. The main difference with the simpler case presented in the previous section is the construction of the detector that is made reading the structure from the CT of the patient. The file containing the CT is a Digital Imaging and Communications in Medicine (DICOM) file, a standard for medical imaging information, it presented a series of Hounsfield units (HU), dimensionless numbers that are used to express CT numbers in a standardized form. Such HU number are read in the code one by one, they are than calibrated using specific tables for density and materials selection, than they are passed through specific Geant4 functions that implement such data within the Detector construction.
The treatment plan is completed with several particles simulated changing the spot of the beam in order to have a distribution of dose among the prescribed area. The solution presented refers to an already validated treatment planning, courtesy of I-See S.r.l.
The presented simulation refers to particle therapy applied to the prostate of a patient. It is important to understand that the work presented is in no way intended to validate the code provided by I-See S.r.l., instead it tries to exploit margins of improvement in term of memory footprint.
As in Section 4.1 different aspect of the simulation have been tuned in order to understand what influences the code performances. In this case is not possible to change the detector size, that is fixed by the CT and also the energy is fixed due to the requirement on the penetration of the peak.

4.2.1 Tuning the number of particle simulated

The former analysis made is on the number of particle simulated. Three different number of simulated particles has been simulated, increasing each time such number of a factor of ten and several differences arose. No change has been made in the execution straegy: all the executables have been ran under Valgrind Massif toolkit to get a memory profiling. The number of particle simulated refers to a choosen statistic for the analysis of the memory profiling.

A first summary of the reults is listed in table 4.5.

N° of particles	Time [s]	Memory Heap [MB]	Extra Memory Heap[MB]	Percentage
10^3	545.44	2462.7	856.3	34.77%
10^4	645.1	2465.8	856.7	34.74%
10^5	1699.018	2468.3	857.2	34.73%

Table 4.5: Memory informations for a different number of simulated particles

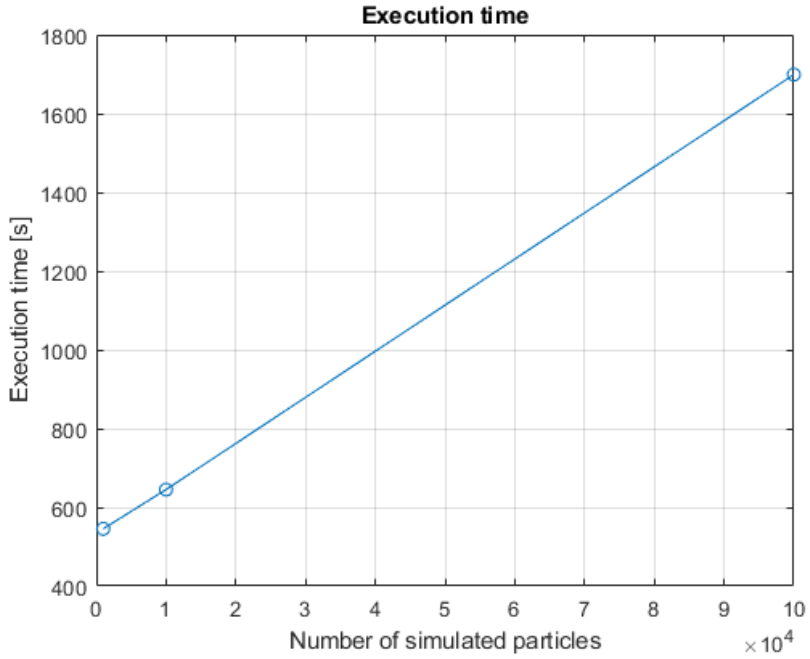


Figure 4.19: Execution time function of the simulated particles

There are two imporant results that need to be highlighted:

- The time execution increasese linearly with the number of particles, as expected and shown in Figure 4.19. On the other side the Memory Heap, among different simulations, is almost constant in all the simulations, as the Extra Memory heap.
- The extra memory heap allocated represents a very high extra memory allocation. This values gives space to a wide set of improvment to reduce the memory footprint that will be discussed in Chapter 6.

For a better analysis of the former result can be usefull to visualize the output file of the massif toolkit presented in Figure 4.20-4.22.

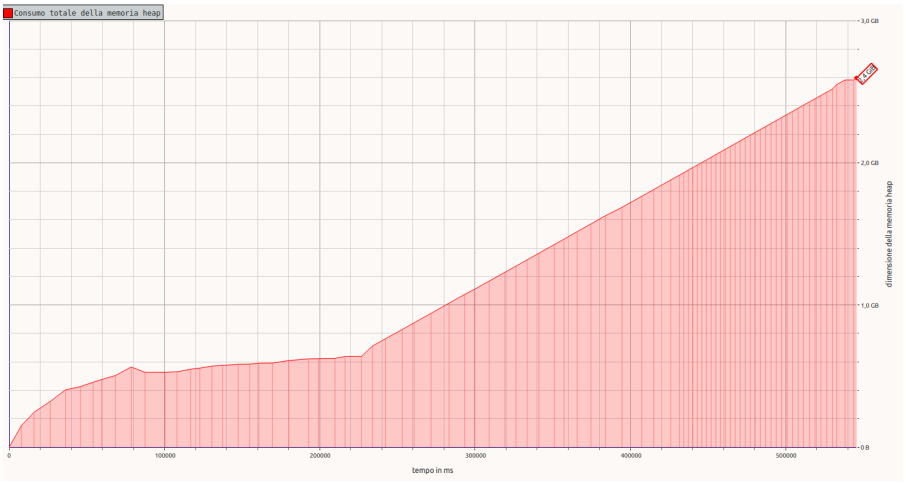


Figure 4.20: Memory Conusmption for n=10^3 particles

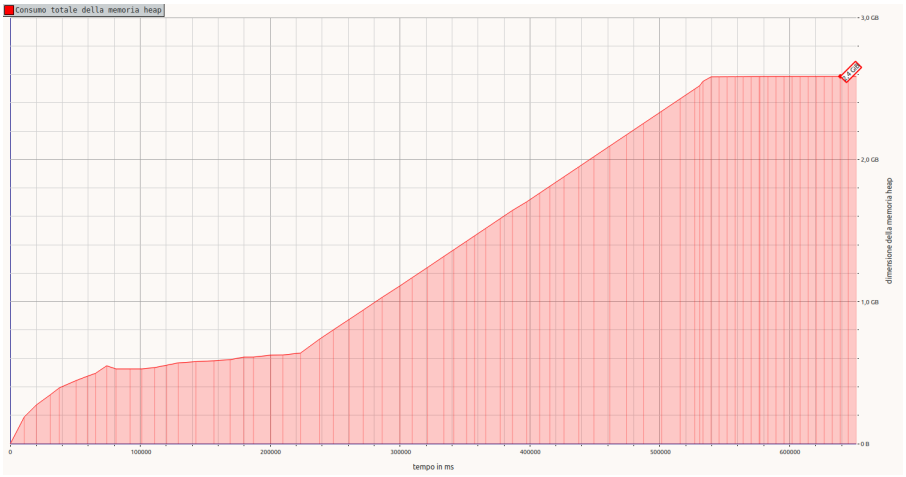


Figure 4.21: Memory Conusmption for n=10^4 particles

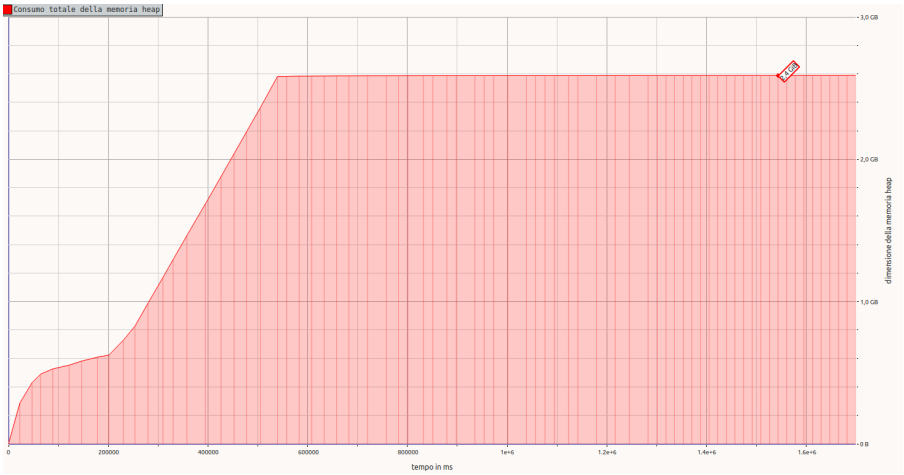


Figure 4.22: Memory Conusmption for n=10^5 particles

This series of graph shows how the heap memory consumption increases with time. The main difference among those three figure lays in the flat part of the curves that increases with the number of particle simulated. Analysing the data provided by the Massif toolkit it is possible to understant the different phases of the simulation. The first ramp up of the curve is associated to the set up of the simulation and some variables that are needed along the whole simulation. The significant increment is associated with the reading of the DICOM file and, in the major part, to the formation of the detector voxels through speicific Geant4 functions. Once the Detector is implemented the beam is inizitialized and than the prescribed particle are simulated. The simulation of the particles is optimized with respect to the Bragg Peak case analyzed in the previous section becasue it does not increase the memory allocation. This explains why the flat part increases with the number of particles but it also enlight the difficulty of having large simulations due to time issues and some possible optimization strategies will be presented in the following chapter.
For a wider analysis of the memory allocation some additional data ara available in the Appendix A.
Before moving on within the analysis is importat to have a graphical visualization of the obtained results. In the following Figure (4.23-4.25) it is possible to understand the result of the single spot simulation. Those figures present a deposition of proton particle overlaid on the CT image. To simplify the interpratation of the deposition a color bar has been presented that indicates the intensity of the proton deposition according to an arbitrary value called 'Intensity'. Such variable refers to the result of the simulation normalized on the particle fluence in order to have comparable results.

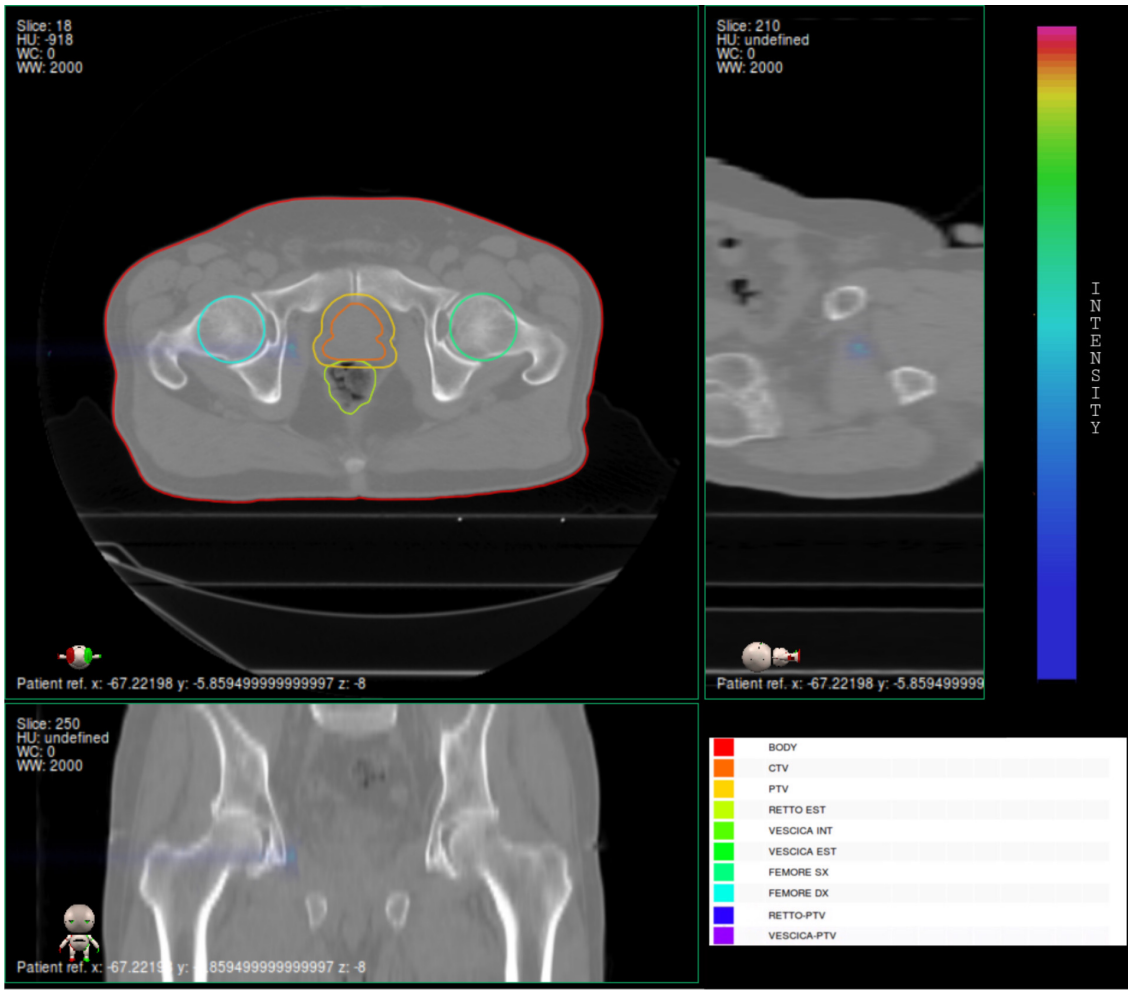


Figure 4.23: Particle deposition for n=10^3 particles

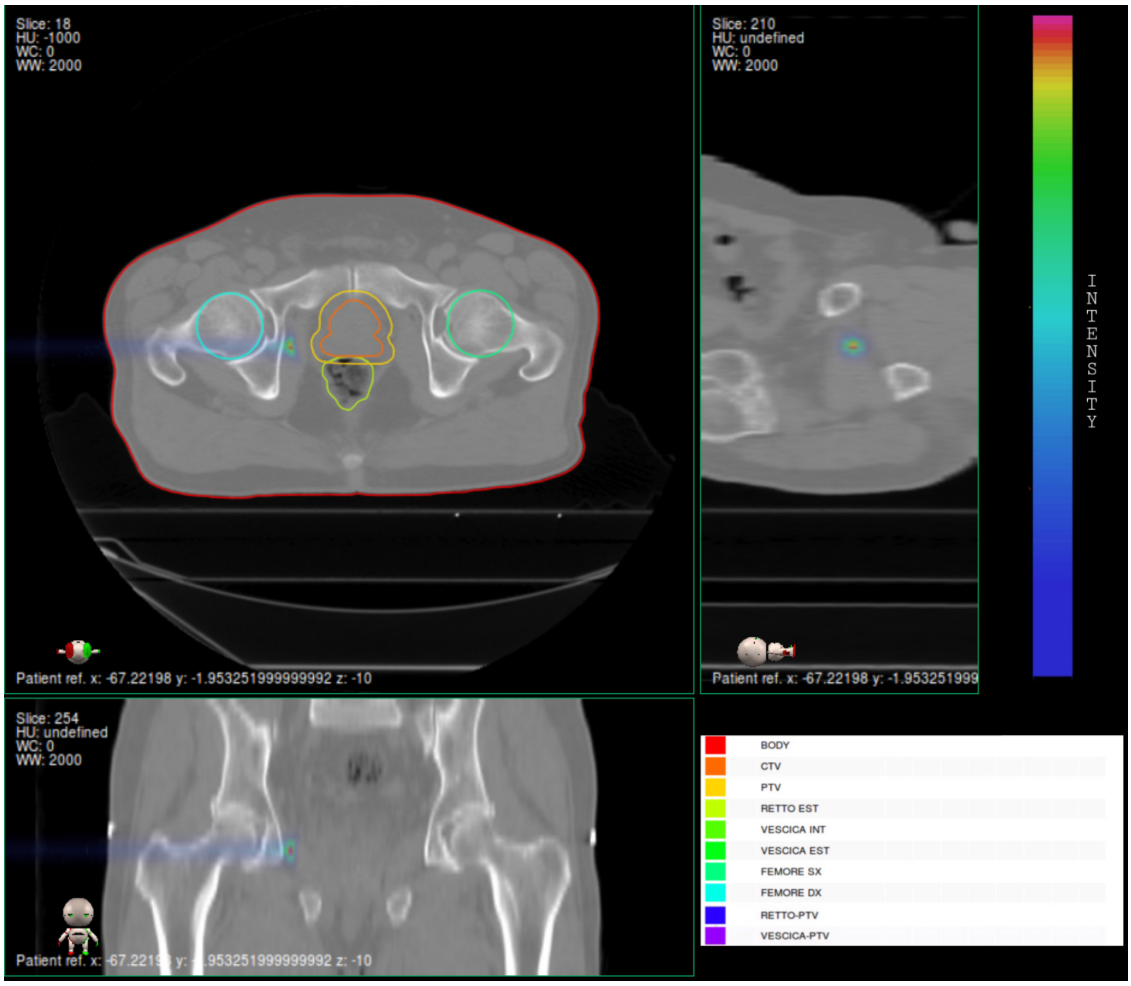


Figure 4.24: Particle deposition for n=10^4 particles

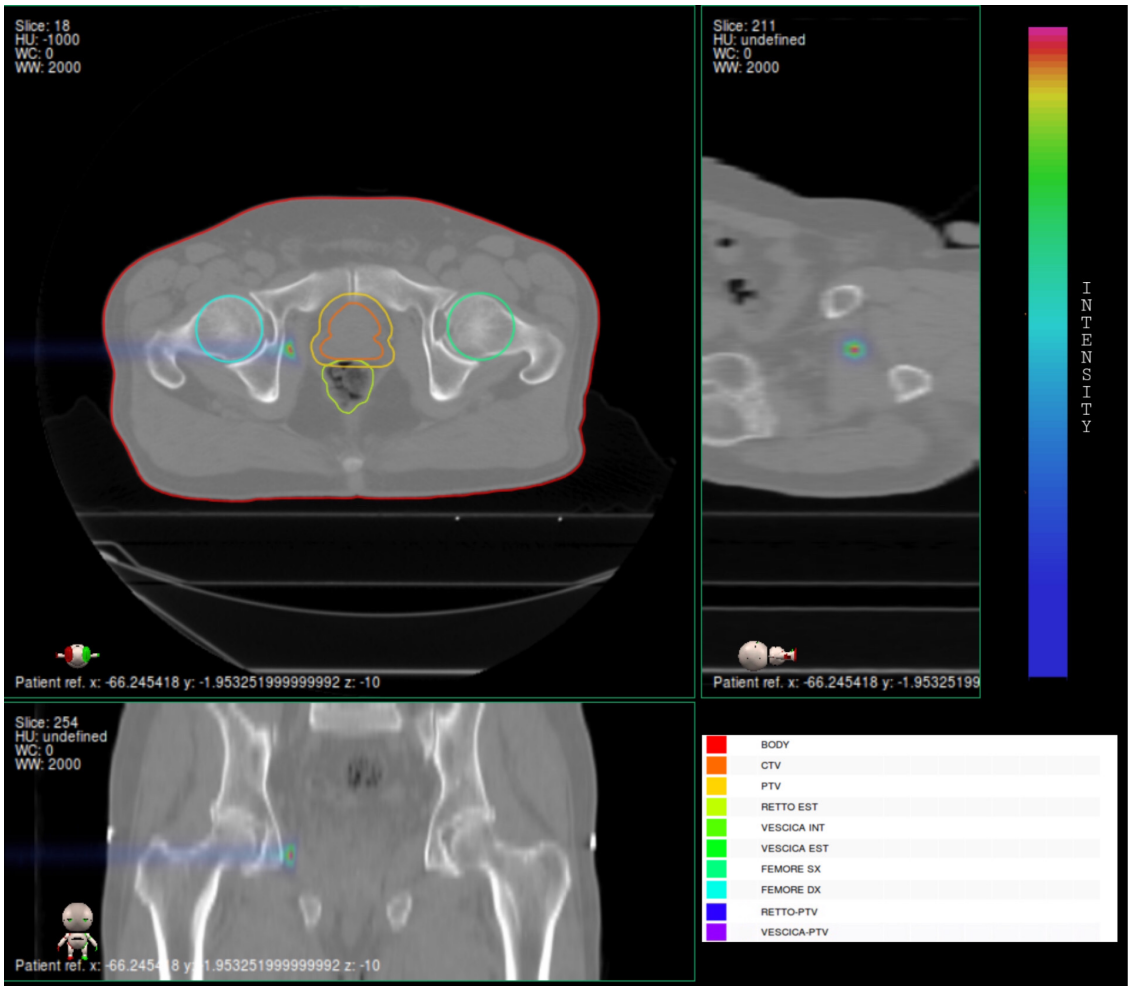


Figure 4.25: Particle deposition for n=10^5 particles

4.2.2 From Single Peak to a Full plan

In the previous subsection has been possibile to understand that increasing the number of particles simulated the peak becomes higher and higher but il is localized in the neighbourhood of a point. In obtaining a full treatment plan it is important to have a deposition of the particle among the prescribed area. This can be obtained chaningn the position of the beam spot and to evalute such positions is necessary to go through different technics. As already stated, the treated plan presented is a result of an inverse planning technique already validated at I-See. The spot prescribed by the plan are more than 1800
For comparison sake, the number of particles that have been simulated are the same (10^5)

for the two simulations thus in reality the number of particles to be simulated for a full plan should be higher.
A summary of the result is available in table 4.6.

Type of simulation	Time [s]	Memory Heap [MB]	Extra Memory Heap[MB]	Percentage
Bragg Peak	1699.018	2468.3	857.2	34.73%
Full plan	1990.9	2477.4	857.4	34.61%

Table 4.6: Comparison of single spot Bragg Peak and Full treatment plan

The execution time has a slight increase mainly due to the time needed to move form one spot to the other within the simulation. As before the heap memory allocated has no change among different simulation, keeping almost the same percentage of Extra Memory Heap allocated. In the following figure is presented the output of the Massif toolkit that can be compared to the one already discussed in Figure 4.22.

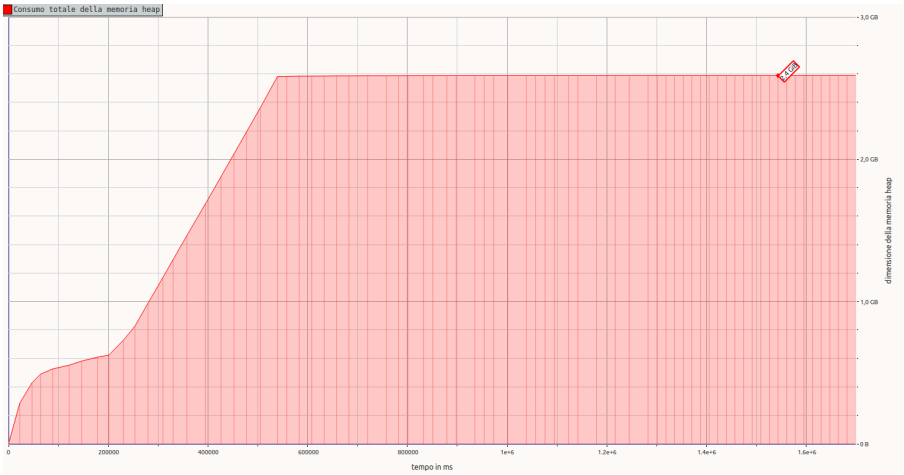


Figure 4.26: Memory Conusmption for a full treatment plan for n=10^5 particles

Comparing this graph with Figure 4.22 there is no major difference if not a slaightly change from a smoother graph for the single Peak case. This is not a crucial difference but it only enlights the difference in changing the position of the beam source. This difference confirm that the Memory footprint does not depends on any input data nor on the settings choosen for the simulation. Detailed memory cost is presented in Appendix A.
Closing the discussion on the memory consumption, it is possible to have a graphical understanding of what a full plan is, with respect to the previous peak, using the same technique previously described. The result are as follows:

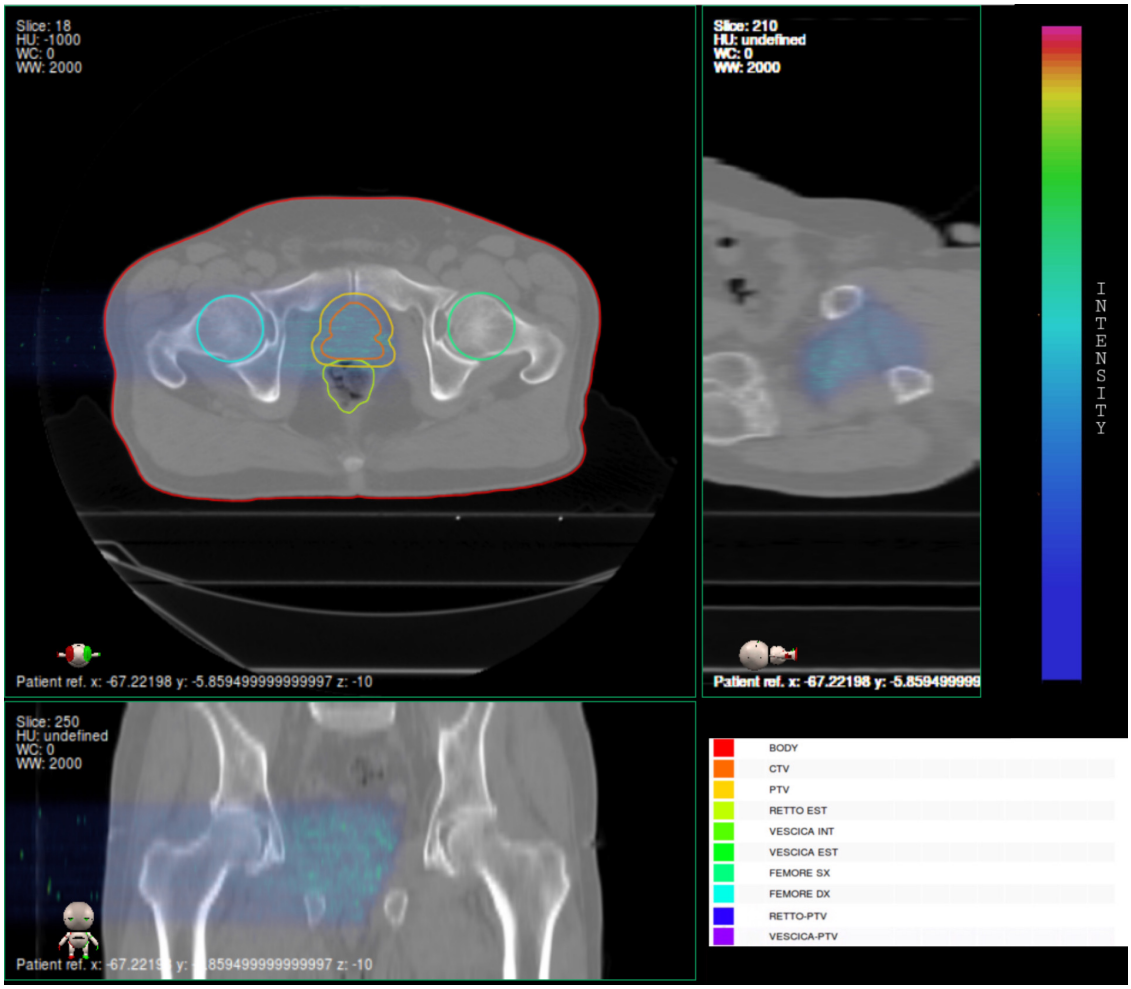


Figure 4.27: Full Plan for n=10^5 particles

Chapter 5

Execution Time optimization

The most important aspect of the simulation to improve is the execution time. The solution that will be discussed in this chapter is the possibility to parallillize the execution of some tasks taking advantage of the Graphical Processing Unit.

5.1 Technical Specification

The main topic of this section is a detailed analysis of the differences between the Graphical Processing Unit (GPU) and the Central Processing Unit (CPU), their strengths and their drawbacks.

5.1.1 Hardware differences

Central Processing Unit, CPU

The Central Processing Unit, sometimes also called just processor, refers to the electronic circuitry within a computer that receives and handles different instruction to make up a computer program. [36]
The History of the CPU begins with the rise of the stored-program computer, that aim to overcome some limitation of the ENIAC [cap 1.2.1] through the storage of the programs in high-speed computer memory preventing the physical wire connection needed to set up a new task. All of this technological enhancements have been summed up by John von Neumann in his paper *'First Draft of a Report on the EDVAC'*[37] on June 30, 1945, that gave birth to a new computer architecture known as *'von Neuman architecture'*.

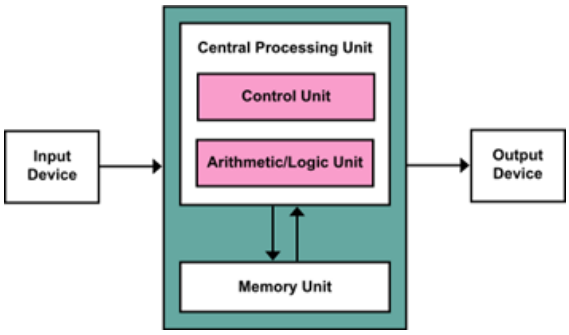


Figure 5.1: von Neumann architecture [36]

The design of early CPUs refers to customed processors for particle application than the production moves to multi-purpose processors till the begin of a standardization process with the arise of transistor mainframes. Nowadays CPUs take advantages of integrated circuit (IC) leading to very complex components with a manufacturing tolerance of the order of nanometers that make them suitable for other electronic devices such as smartphones. The inner structure of the CPU can be subdivided into two main components:

- *Control Unit, CU,* is responsible of directing processor operations by decoding the machine language opcode into command that directs the behavior of the Arithmetic logic unit (ALU). The CU gives directives to other units by providing control signals and timing, moreover it manages data flow between CPU and other devices.
- *Arithmetic logic unit, (ALU),* performs makes arithmetic and logical operations. ALU receives as input operands and a code, from CU, specifying the operation to be performed. Outputs are both data word stored in a register memory and status information that is stored in a specific CPU register.

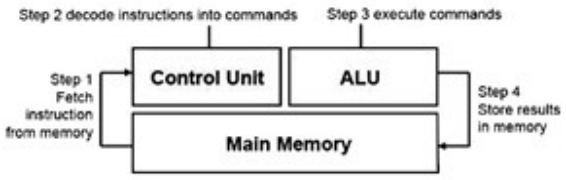


Figure 5.2: CPU structure [36]

To boost CPUs' performance have been developed Multi-core CPUs that are a computer processor integrated circuit with two or more different processing units. The most used

metrics for benchmarking CPUs are:

- *Instructions per second*: it is a combination of the clock rate, measured in multiples of Hertz, and the instructions per clock (IPC).

$$IPS = \frac{Instructions}{Second} = \frac{Insturctions}{Clock - cycle} \cdot \frac{Clock - cycle}{Second} = \frac{Clock - rate}{Instructions - per - clock}$$

- *FLOPS*: that stands for floating point operations per seconds that results more efficient to evaluate CPU's performance in scientific computations involving floating-point operations.

Graphical Processing Unit, GPU

Graphical Processing Unit, GPU, is an electronic circuit designed to work as a processor for parallel image processing and computer graphics manipulation[38]. In origin the term 'GPU' refers to stand-alone programmable graphics processor that works without the CPU support for graphics manipulation. First commercialization of the GPU can refer to the Sony GPU designed by Toshiba to be implemented in the Play Station console. 1999 the American company Nvidia announced the placing on the market of "the world's first GPU" the GeForce 256.
Architecture of the GPUs can be described as an array of independent Compute units, called cores that differ from the CPU because they have a simpler structure due to the different purposes of each component. Initially GPUs were implemented for memory-intensive work of texture mapping and rendering polygons, later different feature related to graphics manipulation have been enhanced.

Recent developments on GPU computation involve matrix and vector operations opening the possibility to take advantage of GPU's parallel computing capability for non-graphical calculation in different field of science suitable for parallelism: the so-called General-Purpose GPU (GPGPU) programming.
For sake of simplicity there will be presented the NVIDIA's Fermi microarchitecture that represent the first development of a Complete GPU computing architecture.
Fermi GPUs feature three billion transistors that build 16 Streaming Multiprocessors composed by 32 CUDA cores, 16 Load/Store Units (LD/ST) and 4 Special Functions Units (SFUs) each. Among the SMs there are 6GB of GDDR5 DRAM memory, a Giga Thread global scheduler that account thread blocks distribution to SM thread schedulers (further explanation in chapter 2.2.2) and Host interface for GPU-CPU connection (via PCI-Express).

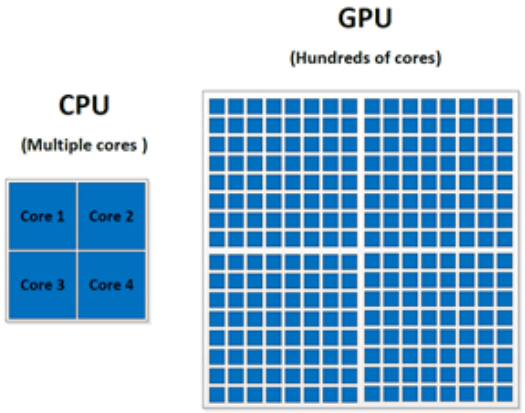


Figure 5.3: N° of cores in CPU and GPU [38]

Basis of the parallel computing are the CUDA cores that present an arithmetic logic unit (ALU) and floating-point unit (FPU) each. Fermi's ALU supports full 32-bit precision for all instruction and a good optimization for 64-bit operations.

CPU and GPU Comparison

Developing a comparison between CPU and GPU is not only a matter of number of cores but also of the type of core composing each unit for a broader view of the components in exam.
Some definitions are needed:

- *Throughput*: refers to the rate of information units processed in a precise number of processor clocks. It can be measured by time needed to complete a specific workload.
- *Latency*: refers to the delay generated by an instruction in a dependency chain. Such delay is measured in processor clocks within data availability for two sequential instructions.
- *Serial Instruction Processing*: refers to Single Instruction, Single Data (SISD) computing that employs a single processor with a single set of input data at each time
- *Parallel Instruction Processing*: refers to Single Instruction, Multiple Data (SIMD) computing among several processors: each processor computes the same instruction with different input data

In detail, we have that GPU requires and consumes less memory than CPU resulting in faster operations with high throughput and high latency that makes it more suitable for

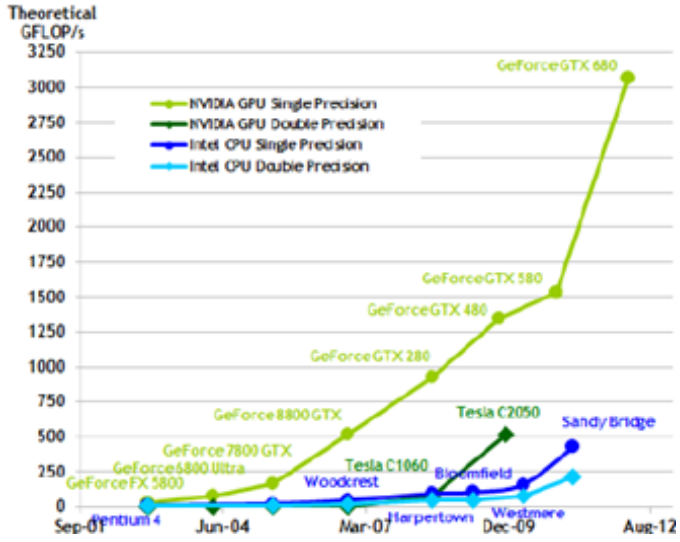


Figure 5.4: Comparison of CPU and GPU single precision floating point performance through the years .[39]

parallel processing as its hardware specifics suggest. Weakness of the GPU is the narrow instruction sets provided that makes the CPU more versatile.
This short dissertation highlights that the only optimal solution is a hybrid computing that benefits from each processor strength. In the following chapters a broader analysis of programming languages for both CPUs and GPUs and how to make both communicate to reach High Performance Computing (HPC).

5.1.2 CUDA, A programming language for GPGPU computing

It has already been introduced the concept of General-Purpose computation on GPU for general computing operations that are able to significantly speed up an algorithm through parallel instruction properties. General Idea is to make CPU and GPU co-operate and it is possible by exploiting specific programming languages that have been developed to share request among Central and Graphical processing units. [38]
The programming languages that allow CPU-GPU interaction for algorithm are mainly two:

- OpenCL: open standard that can be used for multiplatform programming and as no vendors requirement on the GPU type. The portability of such programming language reflects in a lack of performances

- CUDA instead is NVIDIA-GPU programming languages ensuring high efficiency of kernels. The absence of portability to other vendors GPU is partially compensated by the major role that NVIDIA has in the GPU market.

The chosen programming language for this analysis is CUDA for better implementation on the available GPU.[40]
In general, CUDA extends C++ codes by defining kernels that are simple C++ function that can be executed N times in parallel by N different CUDA threads. The definition of kernels is straight forward with just few extensions to classic C++ Language. The kernel is declared with the identifier global that marks this function as callable only by the CPU (host) but executed by the GPU (device). Another modification is presented in the execution configuration syntax that is represented by «<...»> in which the thread mapping among GPU's CUDA core is made.
Each thread is part of a grid/block configuration that can be different for different kernels declaration. The subdivision is as follows:

- Grid: refers to a group of thread running the same kernel. Threads are not synchronized.
- Block: refers to a logical unit containing several coordinating threads and a specific amount of shared memory. Blocks can be 1D or 2D and are identified by a built-in variable *blocIdx* for block identification.
- Thread: refers to the inner most part of the gird to which is associated a single kernel execution to which is associated a certain amount of register memory. As for blocks, also threads have their built.in variable for identification.

Another important aspect of CUDA C programming is the different kind of memory types that can be accessed by each CUDA thread. They can be divided into:

- Global memory: Read and write memory, it is slow and un-cached that can communicate with the host.
- Constant memory: Location in memory for constants and kernel argument storage, it provides a cache, but it is slow. As the Global memory it can communicate with the host.
- Shared memory: refers to single block memory for read and write operations. Its value determines the number of threads per block that can be executed and denotes the block occupancy.

- Registers: Fastest memory of the device and a set memory for each thread.
- Local memory: Refers to what cannot be stored into registers to allow coalesced reads and writes.

Once the program is built the compilation is straight forward taking advantage of a specific compiler developed by NVIDIA named: nvcc. This specific compiler driver accepts a great variety of conventional compiler options such as macros definition and include/libraries paths even though it needs a general C++ host compiler that accounts for all non-CUDA phases (except the run phase).[39]
Developing a CPU-GPU for code acceleration presents different drawbacks that are of fundamental importance and, if not accounted, can result in a slowing down of performances.[41] The main issues are memory and synchronization:

- Different memory location between CPU and GPU need data copying between different location. Even if there are specific functions already implemented in CUDA for memory copy and allocation (cudaMalloc, cudaMemcpy, cudaFree) that can be a bottleneck for program execution thus they should be minimized.
- As already stated, threads are non-synchronized operations and the CUDA kernel are non-blocking functions that let the host code compile without attending the end of kernel execution. Solutions have been implemented through built in CUDA function CudaThreadSynchronize().

For some example of CUDA programming and comparison with code running on CPU see Appendix B.

5.2 Parallel Programming

Parallel programming is becoming more and more relevant in high-performance scientific computing benefit from the production of processors with several power-efficient computing units within one single chip. The general idea is to take parts of the code, called tasks, that run sequentially and make them run in parallel on several cores. But not all the tasks are suitable for parallel programming that explains why is important to integrate the usage of multi-cores within sequential operations. The most suitable operations for parallelization are the ones within loops that do not require results from previous iteration, so that each core can be responsible of the execution of one iteration making code run much faster. All these improvements come along with a series of difficulties that have a high impact in the

decision of implementing existing application in order to run in parallel.
At this stage is necessary to enter in the details of the basic concepts of parallel programming to address all the challenges at any stage of the implementation.
The first step of the design of a parallel algorithm is the decomposition in tasks of the application. This step can be tedious for the different composition that can be made without any general rule to be followed. The number of instructions within a task is defined as granularity and is an important aspect to be defined through a specific analysis of the application. After the definition, tasks must be coded in a parallel programming language and are assigned to processes or threads that are responsible of the execution with an ordered scheduling that can be done within the source code or by the programming environment, at compile time or dynamically at runtime. The assignment of processes or threads onto the cores follows a precise mapping, done by the runtime system, and, if needed, a synchronization and coordination for correct execution. Synchronization and coordination are related to the information exchange between processes or threads that, in turn, depend on the memory organization. [39][42]The memory classification is in:

- Shared memory machines, that connects to the term thread, stores data in global shared memory that is accessible by all processors and the information exchange among threads is done by shared variables
- Distributed memory machines, that connects to the term process and for each of them there is private memory with no need of synchronization for memory access. The information exchange is done by explicit communication operations.

It is possible to place specific barrier operations for coordination which is available for both type of memory machines. This is very important because the execution of some kernels within a standard C++ code is a non-blocking operation, meaning that, without barrier operations the code will continue without the information needed from the kernel. For hybrid codes within CPU and GPU memory is the most crucial aspect because the memory copy among host and device results in a bottle neck for the code thus, if a programmer does multiple copies between memories all the boost given by the parallel execution can be lost.
Moving to Parallel programming is not just a matter of setting up the problem but also to understand how to make the instruct the multiple cores to execute the prescribed instruction. This becomes particularly tedious when the source application presents the usage of some external toolkit that are based on sequential programming languages and to move to parallel programming is necessary to go through a validation procedure that is not so

easy. This is the case of Geant4 that being based on C++ is not so easy to translate in some parallel programming language. A first step forward to possibility if parallelization introducing the possibility to use Multithreads options among different cores of the CPU that is an intrinsic function introduced in the latest versions but that is not perfectly optimized. Another difficulty lays in the impossibility to use virtual function of a toolkit on the device thus needing a complete translation. Other solutions are discussed among different research groups and their details are summarized in the following section
The ideal solution would be to move parallelize different aspect of the simulation:

- The upload of the CT scan within the simulation to ensure faster material definition keeping track of all the possible time lost moving information between CPU and GPU.
- Assign to each CUDA core of the GPU a single particle to be followed from generation till its death. This will strongly reduce the time of the simulation if the data transfer between device and host is reduced at maximum.

Once all this general guidelines are clear is important to have some hard coding to instruct the GPU cores and also make the prescribed compiler, nvcc, to understands which tasks needs to be executed in parallel and which keep sequential execution and it requires high level computing skills.

5.3 State of the art of Particle therapy simulations on GPUs

5.3.1 Geant4 based solutions

There have been increasing interest in moving Geant4 to GPU for faster simulations, that is why different research groups have been involved in different projects with different aim and results. Some of them are than listed here:

- Geant4-Navigation. In Geant4 the Navigation solves geometrical problems on where a specific particle is. In this work the code is firstly translated from C++ to C99 replacing classes with structs and 'forking' is used to implement virtual functions. Results of this simple test seem promising but they have been implemented among simple geometries and further updates are missing [43]
- GGEMS, that states for GPU GEant4-based Monte Carlo Simulation remains one of the most promising project data implements Geant4 properties that are than loaded

to the GPU to run in parallel. This project presents different implementation, mainly for brachytherapy purpose, but it aims to extended GPU usage among all the simulation aspects from physics lists to dose evaluation. The publication of some results was expected for the first quarter of 2017 but at this days, October 2020, only the homepage of their website is available.[44]

- GATE, that states for Geant4 Application for Emission Tomography is part of the OpenGATE project to provide a free software, Geant4-based for emission tomography. The integration of parallel computing on GPU is made within some commands on a macro file. What is interesting is the possibility to have automatic translation, but this does not reflect in a huge gain in time and sometimes can results in time lost if the user is not expert. Another negative aspect is the leak of some output information once moving on GPU computing [45] [46]

- VecGeom. It is the youngest project that tries to incorporate the Geant4-Navigation idea implementing CUDA-friendliness using custom macros adding the possibility for ROOT compatibility. It is the most promising with very good result but again only with simplified geometries implying a long road to commercial solutions. [47]

5.3.2 Other solutions

The solutions presented are not the only one available but are just some of the most promising and recognized among researchers.

- FROG (Fast Recalculation and Optimization on GPU) platform was developed at HIT (Heidelberg) and CNAO (Pavia) in 2017. It was mainly intended for faster plan recalculation and it has shown a maximal variation of -1.5% with respect to the well validated FLUKA model for plan recalculation. It is the only one already working in different centers in Europe including CNAO, HIT but also some other University Hospital. [48]

- Fred (Fast paRticle thErapy Dose) evaluator is a dose engine on GPU to recalculate and optimize ion beam treatment. The main purpose is to have a faster recalculation of a complete treatment plan for many clinical applications strictly related to time optimization. [49]

Chapter 6

Memory consumption optimization

The results of the profiling reflects the need to optimize the memory consumption of the code for improving performance of the code, in general speeding up the execution, and decrease the memory footprint. The memory location does not affect correctness of the program but it can have a strong impact in the performance. Running the code under the profiler is necessary to spot the problem than different solution can be proposed in order to fix the problem. In this chapter will be presented an overview on the memory structure and different solutions tha can be implemented within the code to reduce the memory consumption.

6.1 Memory structure

The memory associated to each processor is divided in 3 main parts:

- *Static/Global memory* is the part of the memory that contains all the instructions implemented within the code.
- *Stack memory* in which are allocated all the variables that are not declared within functions and they are allocated during the whole time of the application. Those variables are allocated in ordered, contiguous block. The access time is fast and the data type structure is linear, the main issue is in the shortage of memory available.

This two parts of the memory do not grow in size during the applications. The limitation of this memory can cause stackoverflow issue if more stack memory is required with respect

to the one allocated a priori. At this level is not possible to manipulate the scope of a varibale.

- *Heap memory* is the greatest part of the memory that can be alloccated and deallocated manually by programmer with a major issue: the allocation is in random order leading to memory fragmentation. Such memory has a hierarchical data structure that makes the acces time very slow.

Heap memory is the one responisble for allocation of big chuncks of data and allows to keep the data allocated for the time needed but all those degrees of freedom can lead to several problems that can significantly affect the code performance. Sometimes the Heap memory is also called dynamic memory to avoid misunderstanding with the heap data structure. Allocation in C++ is obtained through several command (`new/malloc`) and other for the allocation are available (`delete/free`) but there are no key words for placing memory on stack or heap, it is just a matter of the compiler but it is possible to act to optimize such memory allocation process.

6.2 Optimization solutions

The possible solutions presented in this section are just part of good programming techniques that can be used in order to make lighetr programs in term of memory footprint.

6.2.1 Solutions implementable outside the code

In this subsection are presented two different solution that do not require direct action on the source code.

- *Memory Sanitaizer*. It is a dynamic tool that detects uses of uninitialized memor. It relies on shadow-memory, to avoid flase-positive reports, at run time exploiting compile time instumentations. The cost is in a 2.5× in execution time and 2× in memory usage making it, at this tage of development of the avialble toolkits, unsuitbale for large application as the clinicla case analyzed. [50]

- *Garbage Collector* is an automatic memory managment that is indipendentof the programmer. In contrast with other programming languages C++ does not provide a garbage collector but it can be possible to develop one. The main problem is that a non-well validated GC will end up giving a net negative impact on the performance because it could need the programm to stop for check and garbage collection. [51][52]

Those solution are temporary and can take advantage of some previous work that can be extendend on different kind of simulations.

6.2.2 Solutions implementable within the code

The solution presented in this subsection refers to actions that needs to dive into the code and add different lines of code. For well validated codes this actions need to be made by expert programmer to avoid any irreversible complication modifying the code.[53]

- Remove allocation within loops because it can generates the allocation of the same item in different place of the heap memory but with only a small portion needed for the rest of the code. This should be general practice in programming.
- In the framework of a loop it is possible to reuse the memory allocated by allocating the needed bunch of memory outside the code and than have `reserve` and `pushback` functions to force allocation in the same spot. This solution is suitable for object of memory size known a priori.
- Another possible solution is the usage of contiguous/sequential containers. It is allows to have a more ordered memory distributionbut it is applicable to array with fixed size known a priori.
- Using `const` tells the compiler to do some optimization on the memory allocation, especially when the constant variable is called within different functions but it should not be use improperly.
- The most used solution is the one of constructor and destructor. Those are class function that are executed for creating new objects of this particular class or to delete such object when it goes out of scope. Being classes they do not have any return variable. The name of the two classes must be the same for the single exception of a tilde ($\sim$) in front of the deconstructor name. This should be the optimal solution but it is also the one the require the hardest coding.
- Other available solution refers to C++ 17 that is the last revision of the C++ programming language. The main solutions that can be used are non-owing pointers and Polymorphic memory allocators (PMR) that refer that are easily handled by few expert programmer. That is why they have been only cited even if they seem to be the most promising solution for the near future.

Chapter 7

Conclusion

This work constitutes the basis for optimization strategies to be implemented within Monte Carlo codes following the increasing request of faster code. In the field of Particle Therapy it is very important to have fast simulations in order to be able to make several adjustment in the treatment according to the evolution of the clinical situation. This dissertation is intended to be a description of the possible path to be followed to reach the optimizatin objectives cited with a broader understanding on the different, available, solutions.
It is important to understand that no general-purpouse optimization strategy can be prefered among the other, as also stated in the No Free Launch Theorem of Optimization (NLFT) [54] [55], and sometimes the computational cost of finding the best solutions is higher with respect to the maximum possible gain that can be achieved.
The main aim of this dissertation is to trace the most interesting optimization paths to be followed in order to achieve the desired results. As for the example presented in section 5.3, the common aspect is that all different projects aim to shrink their field of interest before moving further but most importantly it requires involving group of perople involved in research for undifined time. The purpouse of the optimization is nobile but a lot of discussion can be made among the worthenss.
This is not a showstopper, instead it can be a very challenging opportunity to keep studying and to work out the most suitable solution for faster simulation and better treatment planning evaluation with the final goal to give contribution for better therapies.

Appendix A

Results of the Simulations

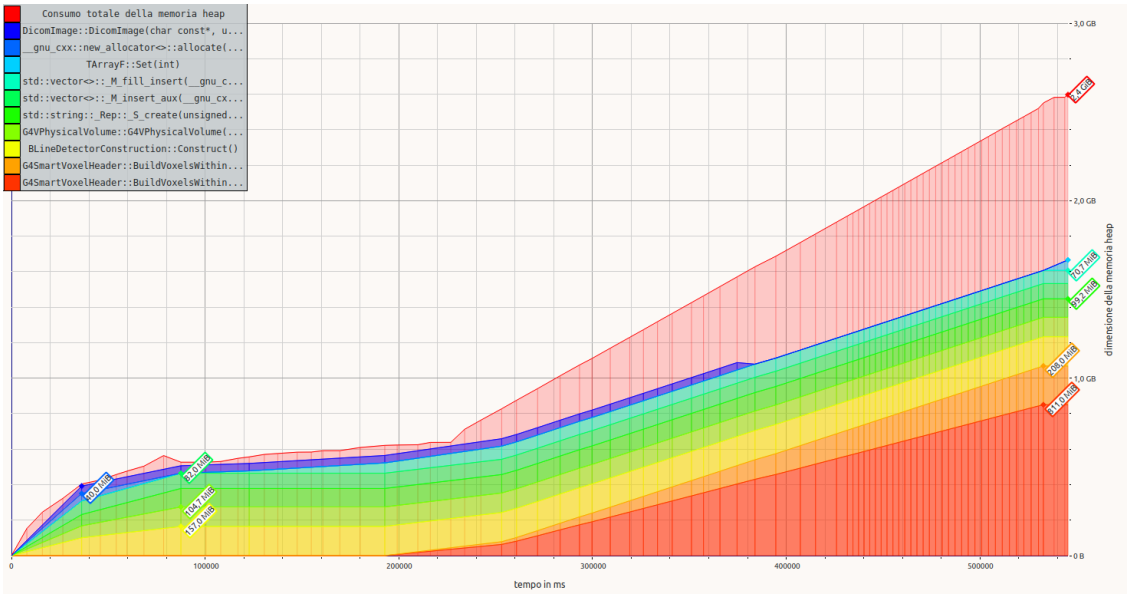


Figure A.1: Detailed memory conusmption for n=10^3 particles

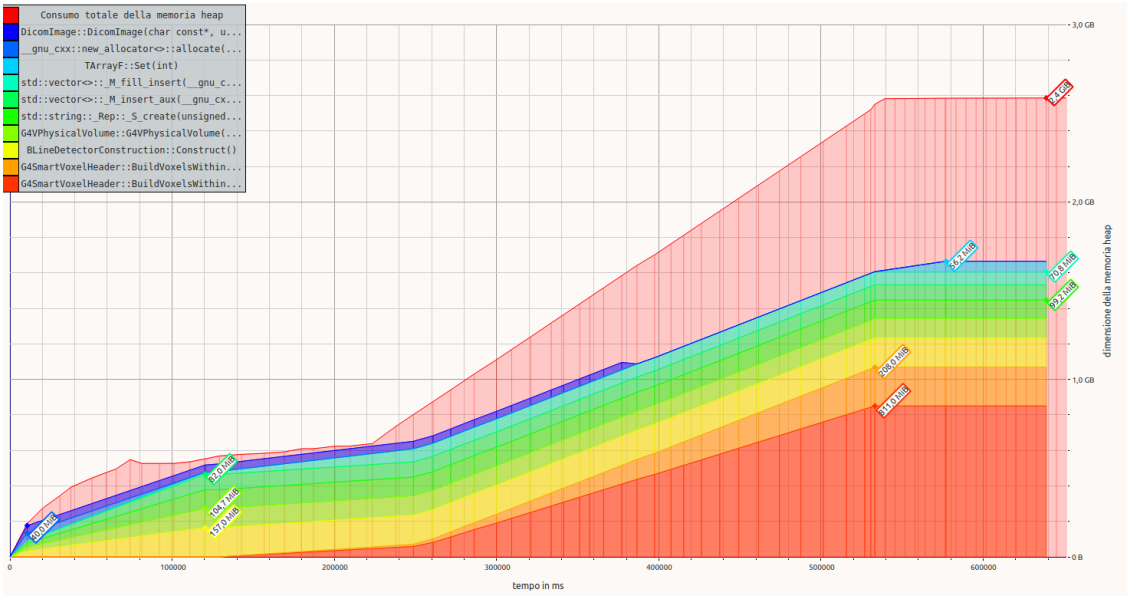


Figure A.2: Detailed memory conusmption for n=10^4 particles

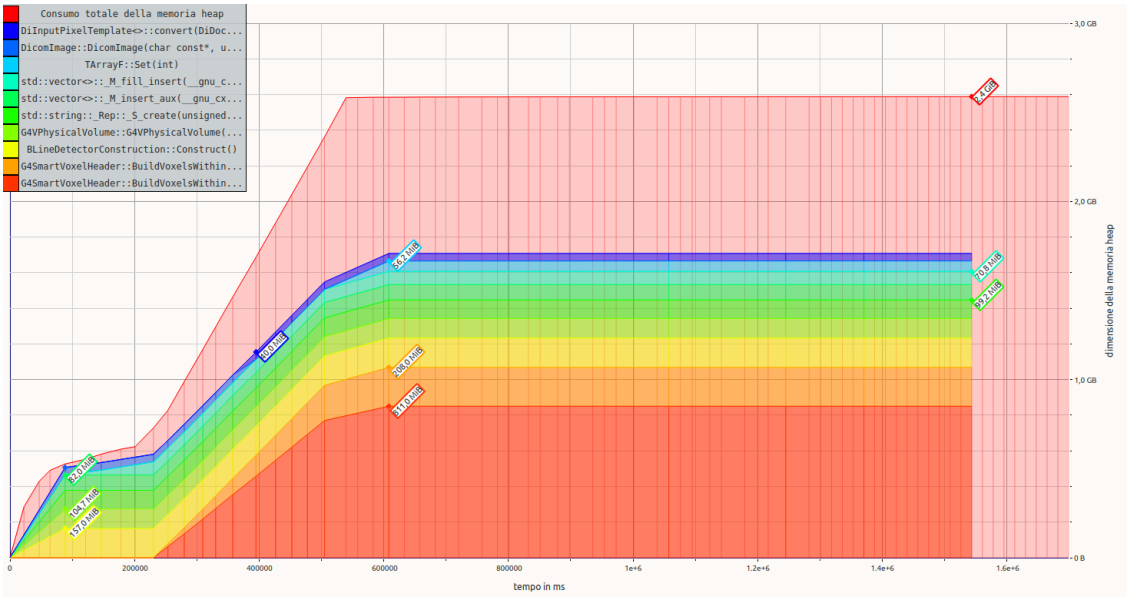


Figure A.3: Detailed memory conusmption for n=10^5 particles

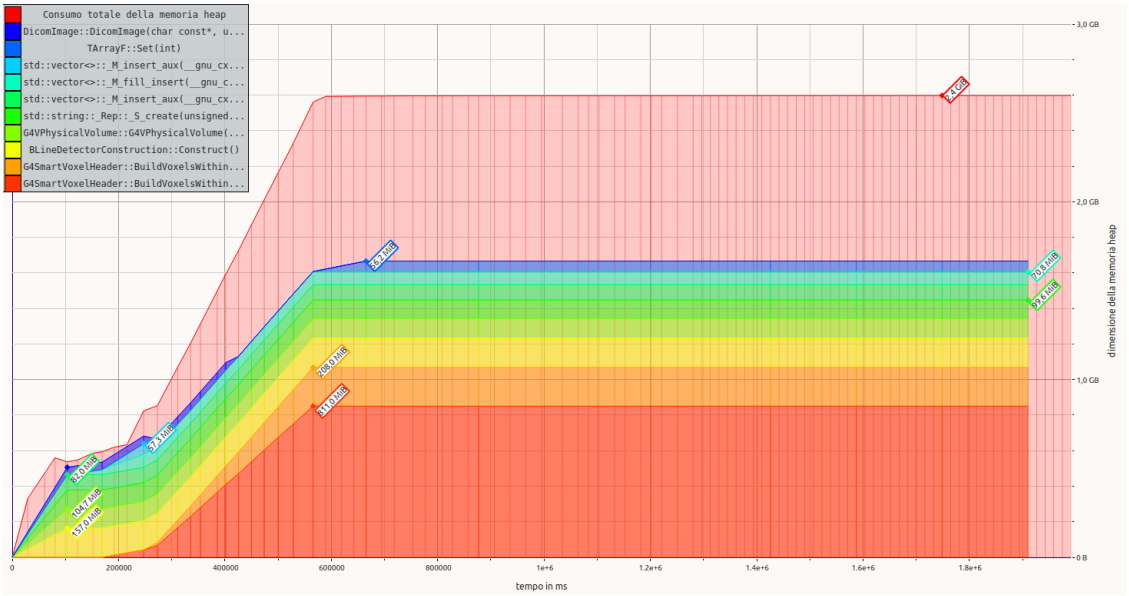


Figure A.4: Detailed memory consuption of a Full Plan for n=10^5 particles.

Appendix B

Example of CUDA applications

Example 1: Monte Carlo code for π estimation

First approach to a Monte Carlo code implementation through CUDA programming has been made by evaluating π.The simulation is very straight forward with an intuitive graphical representation. On a 2D plan is considered a circle, centered in the origin, having radius 1 and enclosed in a 2×2 square.
What the algorithm does is to focus on the first quadrant part of the graph and wants to evaluate the ratio between the area of the circle and the area of the square. Having the possibility to generate an infinite number of random couples of x and y coordinates we would have:

$$\frac{\text{area of the circle}}{\text{area of the square}} = \frac{\text{n° of points generated inside the circle}}{\text{total n° of generated coordinates}}$$

Knowing that the area of a quarter of the circle is $frac\pi 4$ last stage of the algorithm is:

$$\pi \cong 4 \cdot \frac{\text{n° of points generated inside the circle}}{\text{total n° of generated coordinates}}$$

For each (x,y) a check of the position of the point is made through the inequality $x^2+y^2 \leq 1$ and the simulation evolves associating a random variable of value 1 for coordinates within the circle, 0 in the opposite case. At the end of the prescribed iterations the sample average is evaluated and multiplied times 4 for final π estimation.

Two different codes have been written in both CUDA and C++ for time and resolution comparison. In the C++ code the different simulations are done in a sequential way. In CUDA code GPU cores have been mapped into a 1D grid structure of several blocks, of variable numbers depending on the iteration number, and a fixed number of threads that develop a few sequential simulations each.

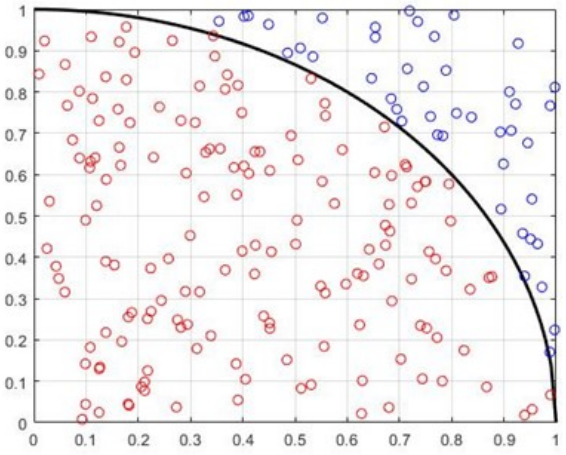


Figure B.1: Example of cooridnates position check

Both codes provide strong results in term of accuracy but with very different computational times.

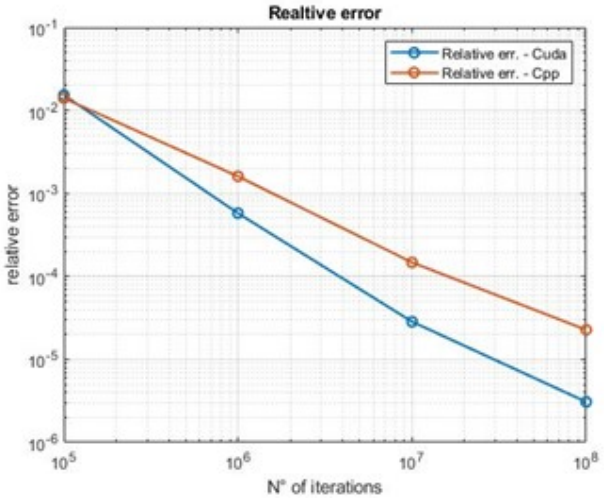


Figure B.2: Relative error comparison

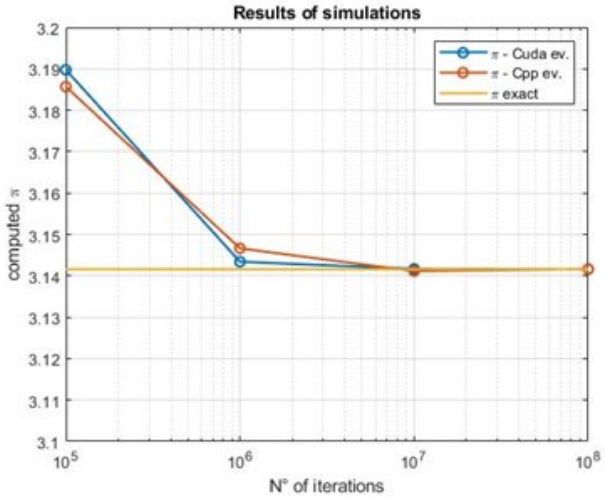


Figure B.3: Simulation Results comparison

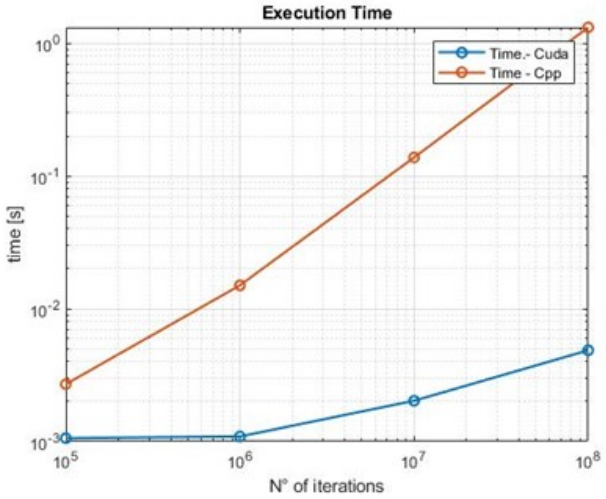


Figure B.4: Execution time comparison

As shown in figures a good accuracy can be achieved in a relatively low number of iterations with same reduction behavior for both the codes. Comparing computational time needed in both code the strength of CUDA code becomes more and more visible for increasing number of iterations confirming the theoretical potential discussed.
Even if the example is a very basic Monte Carlo simulation it is possible to consider a more complex simulation as a rescale of the analyzed case with some bottlenecks for memory copy among host and device.

Example 2: 3D Matrixes Sum

Another important step in the Monte Carlo simulation of a Radiotherapy treatment planning is the sum of large matrixes that in general represent the CT image reconstruction and the Deposited Dose that results from the simulation.

What slows down the operation in current C++ code is the application of three nested loops among the three different directions of the 3D matrix for an element-by-element sum that requires long computational time. Such sequential execution of the same instruction perfectly suits the Single Instruction, Multiple Threads (SIMT) execution model of a CUDA code.The main idea is to map the GPU's CUDA cores to get a 3D structure of the same dimensions of the matrixes involved such that each thread can easily account for a single element sum to be executed in parallel with the other threads.

In the particular, hybrid, code that has been developed generates two 3D matrixes filled by random float numbers that are the input data for the two code. For C++ part the development was very straight forward with the implementation of the already cited nested loops instead in CUDA different difficulties arise.

The first issue was in data copying from host (CPU) to device (GPU) for the complexity of the matrixes structure. A simple solution was to pass the 3D host matrix into a 1D array on the device by an intrinsic flattening of the CudaMemCpy() function. Than in the kernel a simple 1D sum has been made splitting each element for a single thread and at the end of the kernel execution the 1D array has been automatically re-allocated in a 3D matrix double-checking results with the pervious ones. The other issue was the definition of the fixed number of threads per blocks so different runs have been made with different values ranging between 8 and 32 in order to ensure portability of the code among different Graphical Processor Units.

Three different sizes of matrixes have been accounted for a broader view among all the possible causes of differences in code implementation. It is possible to see in figure that the improvement in computational time is more and more visible as dimension of the matrixes increase. Another important aspect is the thread number that ensures better result that is the highest of the three analyzed (32). It was easily predictable because of the synchronization of different threads in a single box and it opens up the possibility to use even more threads per block in GPUs with greater compute capability thus reducing the portability

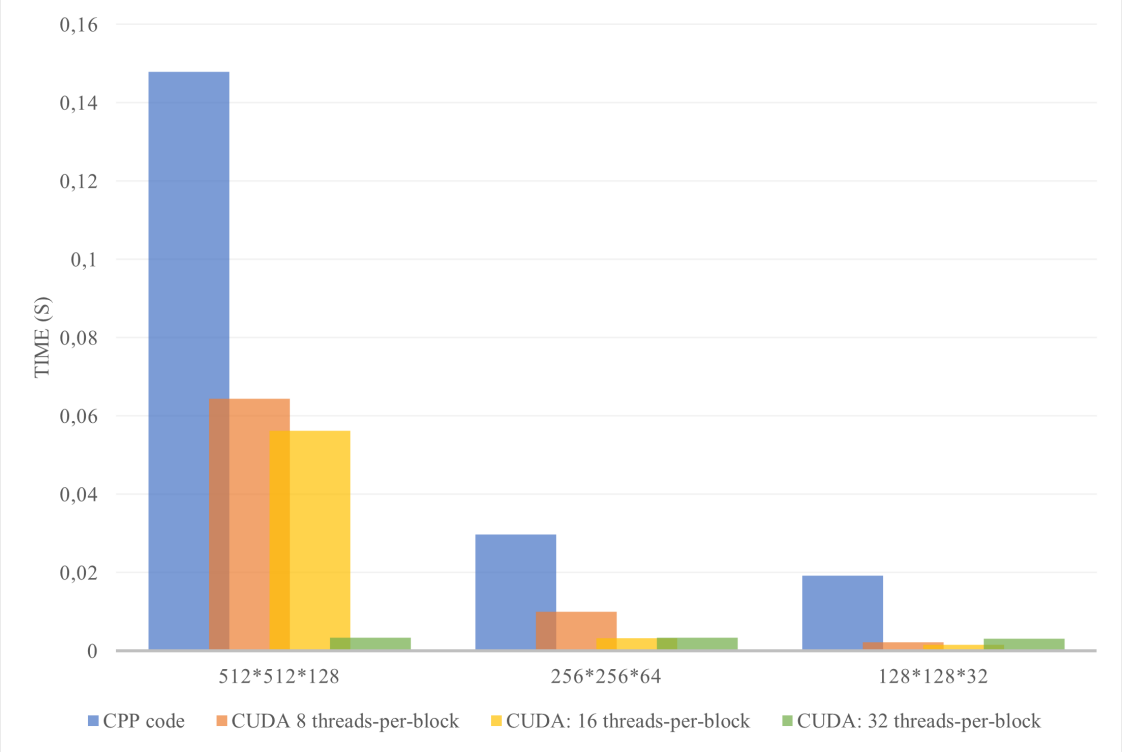


Figure B.5: Execution time comparison

Bibliography

[1] Wikipedia. URL: https://en.wikipedia.org/wiki/History_of_radiation_therapy.

[2] M. Endo. "Robert R. Wilson (1914–2000): the first scientist to propose particle therapy–use of particle beam for cancer treatment". In: *Radiological Physics and Technology* 11 (2017). DOI: https://doi.org/10.1007/s12194-017-0428-z.

[3] R. R. Wilson. "Radiological use of fast proton". In: *Radiology* 47.5 (1946). DOI: https://doi.org/10.1148/47.5.487.

[4] C.A. Tobias et al. *Radiological use of high energy deuterons and alpha particles*. The American Journal of Roentgenology, Radium Therapy, and Nuclear Medicine, 1952.

[5] H. Paganetti. "Proton Therapy Physics". 2012.

[6] M. M. Fitzek M. W. McDonald. *Proton Therapy*. Current Problems in Cancer, 2010. DOI: https://doi.org/10.1016/j.currproblcancer.2010.04.008.

[7] Fondazione CNAO. URL: https://fondazionecnao.it/.

[8] Krengli M. Orecchia R. *Number of Potential patients to be treated with proton therapy in Italy*. PubMed, 1998. DOI: https://doi.org/10.1177/030089169808400218.

[9] The Particle Therapy Co-Operative Group (PTCOG). URL: http://ptcog.web.psi.ch.

[10] Colin K. Hill and Parvesh Kumar. "Radiation Carcinogenesis". In: *Encyclopedia of Cancer*. Ed. by Manfred Schwab. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 3134–3137. ISBN: 978-3-642-16483-5. DOI: 10.1007/978-3-642-16483-5_4898. URL: https://doi.org/10.1007/978-3-642-16483-5_4898.

[11] D. S: Chang et al. "Therapeutic Ration". In: *Basic Radiotherapy Physics and Biolog* (2014), pp. 277–282.

[12] Wikipedia. URL: https://en.wikipedia.org/wiki/Electron_therapy.

[13] Wikipedia. URL: https://en.wikipedia.org/wiki/Absorbed_dose.

[14] S. T. Mugford, E. B. Mallon, and N. R. Franks. "The accuracy of Buffon's needle: a rule of thumb used by ants to estimate area". In: *Behavioral Ecology* 12.6 (Nov. 2001), pp. 655–658. ISSN: 1045-2249. DOI: 10.1093/beheco/12.6.655. eprint: https://academic.oup.com/beheco/article-pdf/12/6/655/9742133/bhec-12-6-655.pdf. URL: https://doi.org/10.1093/beheco/12.6.655.

[15] Wikipedia. URL: https://en.wikipedia.org/wiki/Buffon$%5C%$27s_needle_problem.

[16] Wikipedia. URL: https://en.wikipedia.org/wiki/FERMIAC.

[17] R. Eckhardt. "Stan Ulam, John von Neumann, and the Monte Carlo method". In: *Los Alamos Science* Special Issue 15 (1987), pp. 131–137.

[18] Haigh T. et al. "Los Alamos Bets on ENIAC: Nuclear Monte Carlo Simulations, 1947-1948". IEEE Annals of the History of Computing. 2014.

[19] Wikipedia. URL: https://en.wikipedia.org/wiki/Random_variable.

[20] H. Paganetti. "Range uncertainties in proton therapy and the role of Monte Carlo simulations". In: *Physicsin Medicine & Biology* 57 (2012). DOI: https://doi.org/10.1088/0031-9155/57/11/R99.

[21] Emiliano S. et al. *An Overview of Monte Carlo treatment planning for radiotherapy.* Radiation Protection in Dosimetry, 2008. DOI: https://doi.org/10.1093/rpd/ncn277.

[22] A. Pedro. *Monte Carlo simulations in radiotherapy dosimetry.* Radiation Oncology, 2018. DOI: https://doi.org/10.1186/s13014-018-1065-3.

[23] A.M.Syme et al. *Monte Carlo investigation of single cell beta dosimetry for intraperitoneal radionuclide therapy.* Physics in Medicine & Biology, 2004. DOI: https://doi.org/10.1088/0031-9155/49/10/009.

[24] Geant4 Collaboration. *Introduction to Geant4*. 2020. URL: http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/IntroductionToGeant4/fo/IntroductionToGeant4.pdf.

[25] S. Agostinelli et al. "Geant4 - a simulation toolkit". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506 (). DOI: https://doi.org/10.1016/S0168-9002(03)01368-8.

[26] G. Collaboration. *Book For Application Developers*. URL: http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/fo/BookForApplicationDevelopers.pdf.

[27] B. Walker et al. *A framework for Monte Carlo simulation calculations in Geant4*. Nuclear Instruments and Methods in Physics Research, 2006. DOI: https://doi.org/10.1016/j.nima.2006.06.070.

[28] G. Borrego-Soto et al. *Ionizing radiation-induced DNA injury and damage detection in patients with breast cancer*. Genetics and Molecular Biology, 2015. DOI: https://doi.org/10.1590/s1415-475738420150019.

[29] M. F. L'annunziata. *Handbook of Radioactivity Analysis*. 2nd. Elsevier, 2003.

[30] T. Simunic et al. "Source code optimization and profiling of energy consumption in embedded systems". In: *Proceedings 13th International Symposium on System Synthesis*. 2000. DOI: 10.1109/ISSS.2000.874049.

[31] NAG. URL: https://www.nag.com/blog/application-performance-profiling-part-2-choosing-tools-effective-profiling.

[32] N. Nethercote et al. *Valgrind: A Program Supervision Framework*. Electronic Notes in Theoretical Computer Science, 2003. DOI: 10.1145/1250734.1250746.

[33] Valgrind Developers. "Valgrind User Manual". In: 27 (2020). URL: valgrind.org/docs/manual/manual.html.

[34] A.E. Raftery et al. "The Number of Iterations, Convergence Diagnosticsand Generic Metropolis Algorithms".

[35] M.R. Driels et all. "Determining the Number of Iterations for Monte Carlo Simulations of Weapon Effectiveness". 2004.

[36] "Central Processing Unit". URL: https://en.wikipedia.org/wiki/Central_processing_unit.

[37] J. V. Neumann. "First Draft of a Report on the EDVAC". 1945.

[38] "Graphical Processing Unit". URL: https://en.wikipedia.org/wiki/Graphics_processing_unit.

[39] "CUDA-C programming guide". URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[40] K. Karimi et al. "A perforance comparison of CUDA and OpenCL".

[41] T. Rauber G. Runenger. “Parallel Programming for Multicore and Cluster systems, Springer”. In: 2010 ().

[42] C.Naverro et al. *A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures*. Vol. 15. Sept. 2013, pp. 285–329. DOI: 10.4208/cicp.110113.010813a.

[43] O. Seiskari et al. “GPU in Physics Computation: Case Geant4 Navigation”. 2011.

[44] Lemaréchal Y et al. “GGEMS-Brachy: GPU GEant4-based Monte Carlo simulation for brachytherapy applications”. In: *Physics in Medicine and Biology* 60 (July 2015), pp. 4987–5006.

[45] OpenGATE developers. “GATE user guide”. URL: https://opengate.readthedocs.io/en/latest/index.html.

[46] J. Bert et al. *Hybrid GATE: A GPU/CPU implementation for imaging and therapy applications*. 2012, pp. 2247–2250. DOI: 10.1109/NSSMIC.2012.6551511.

[47] J. Apostolakis et al. *Towards a high performance geometry library for particle-detector simulations*. Journal of Physics: Conference Series, 2015. DOI: 10.1088/1742-6596/608/1/012023.

[48] S. Mein et. al. *EP-1838: FROG: a novel GPU-based approach to the pencil beam algorithm for particle therapy*. Vol. 127. Radiotherapy and Oncology, Apr. 2018, S992–S993. DOI: 10.1016/S0167-8140(18)32147-9.

[49] A. Schiavi et al. “Fred: a GPU-accelerated fast-Monte Carlo code for rapid treatment plan recalculation in ion beam therapy”. In: *Physics in Medicine & Biology* 18 (2017). DOI: 10.1088/1361-6560/aa8134.

[50] et al K. S. Stepanov. “MemorySanitizer: fast detector of uninitialized memory use in C++”. 2015.

[51] A. Sloud H. Mcheick. “Comparison of Garbage Collector prototypes for C++ applications”. In: *2009 IEEE/ACS International Conference on Computer Systems and Applications*. 2009, pp. 668–674. DOI: 10.1109/AICCSA.2009.5069399.

[52] G. Attardi e. al. “A customisable Memory Managment Framework for C++”. 1999.

[53] K. Serebryany et al. “Memory Tagging and how it improves C/C++ memory safety”. 2018.

[54] Y.Ho et al. “Simple explanation of the no free lunch theorem of optimization”. In: vol. 5. Feb. 2001, 4409–4414 vol.5. ISBN: 0-7803-7061-9. DOI: 10.1109/.2001.980896.

[55] D. H. Wolpert and W. G. Macready. “No free lunch theorems for optimization”. In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82. DOI: 10.1109/4235.585893.

Ringraziamenti

Per chiudere progetto e percorso universitario vorrei ringraziare chi ha reso possibile e speciale tutto questo.
Partirei con il ringraziare la prof.ssa Sandra Dulla per la sua disponibilità e professionalità nel suo ruolo di relatrice. Un ringraziamento speciale alla dott.ssa Faiza Bourhaleb per avermi accolto dal primo giorno all'interno di I-See e per aver donato resilienza a questo progetto nei momenti più duri della sua realizzazione. Grazie a Claudia e Mattia, mi avete accolto con affetto e non avete mai fatto mancare il vostro apporto e supporto al mio lavoro.
Un grazie a tutti i miei compagni di corso, è stato bello avere qualcuno con cui condividere gioia e determinazione, rabbia e sconforto, condividere ogni giorno mi ha permesso di crescere e imparare sempre qualcosa in più.
Grazie ai ragazzi del piano zero del Collegio Einaudi, i primi veri compagni di questa avventura, con voi ho mosso i primi passi e non potrò mai dimenticare la forza di un gruppo che si faceva strada in questa nuova realtà. Grazie a tutti i ragazzi transitati sul Quarto Piano, siete stati famiglia a tanti kilometri da casa.
Un grazie particolare a Martina che tra Collegio e Politecnico ha sempre condiviso ogni emozione, anche quando silenziosa e nascosta.
Grazie a Sandrone, per tutte le battute che potevi fare solo tu e per tutte le informazioni che mi hai impartito, grazie a te non dimenticherò mai che una gallina, in rapporto al suo peso, mangia più di un maiale.
Tutti questi ringraziamenti non sarebbero stati possibili senza alcun persone. Perciò un enorme grazie a mio Padre, sei stata la roccia solida alla base dei miei successi, niente sarà mai grande quanto l'amore dei tuoi gesti silenziosi. Sappi, papà, che tra tutti i lavori che potrò mai fare nessuno sarà bella come quei giorni d'estate in cui mi portavi a lavoro con te.
Grazie a te Mamma, sei sempre riuscita a far arrivare l'amore oltre ogni confine. Grazie per esserci stata sempre, è bello poter contare nel calore di casa ogni volta che mi sembra

che il mondo mi stia scivolando sotto i piedi.
Grazie ad Alessandro, il più piccolo e taciturno, la tua ammirazione mi ha sempre dato la forza di dare il massimo. Grazie per aver sopportato un fratello maggiore lontano senza mai smettere di volermi bene, sappi sempre che non importa quanto lontano saremo saprò tenderti la mano fino a lì.
Grazie a nonno Donato e nonna Rosa, la mia seconda casa per sempre. Grazie perché da quando ne ho memoria siete stati presenti con affetto smisurato. Grazie per tutte le lezioni che ho imparato dalle vostre storie di vita che mi raccontavate in veranda le sere d'estate. Siete stati l'esempio più forte di amore e umiltà, spero che chiunque guardi me riconosca quanto voi mi avete dato.
Grazie a nonno Guglielmo, la persona che più manca davanti a questi traguardi. Se mi guardi da lassù sappi che mi riempie di orgoglio ogni singola volta che il mio nome è accostato al tuo.
Grazie ad Angelo, Daniele, Davide, Domenico, Giovanni, Lorenzo e Nicola, i nostri sogni ci hanno portato lontano ma non c'è nessuno con cui vorrei condividere ogni mio traguardo se non voi. Grazie per ogni singolo sorriso, non ve l'ho mai detto ma vi voglio un bene sincero.
Grazie a Sara, hai donato forza ad ogni mia azione affiancando ogni mio passo con l'amore speciale che ti contraddistingue. Grazie per tutta la luce che mi hai donato nei momenti bui, è anche grazie a te sono l'uomo che sono.
Grazie.