

ENGINEERING INTERNSHIP REPORT

03/10/2014 ~ 09/09/2014

Application Development in the Area of Device Management(Home Automation Box)



Author:

FENG Siyao

*A report submitted in fulfilment of the requirements
for the degree of Engineer in Electronics and Computer Science*

at

Polytech Paris-UPMC

August 2015

Supervisors

Industrial Supervisors:

Matthieu ANNE

Ingénieur Développement, Orange/OLPS

Tel : 04.76.76.42.20 E-mail : matthieu.anne@orange.fr

Academic Supervisor:

Andrea PINNA

Directeur de EI-SE

Université Pierre et Marie Curie - LIP6

Tel : 01.44.27.96.35 E-mail : andrea.pinna@lip6.fr

Acknowledgements

The internship opportunity I had with CARE group was a great chance for learning and professional development. It's here at Orange Labs I strengthened my willing to future research careers. Therefore, I consider myself as a very lucky individual as I was provided with an opportunity to be a part of it. I am sincerely grateful for having a chance to meet so many wonderful people and professionals who led me through this six-month internship.

First and foremost, I would like to express my sincere gratitude to my supervisor M. Matthieu ANNE, for the continuous support of my internship, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this report. I could not have imagined having a better superadvisor and mentor for my internship.

My sincere thanks also goes to M. Julien Roland, M. Marc Douet, who provided me their knowledge and experience and precious time, only with their help I can get better work enviroment.

Last but not the least, I would like to thank all the CARE team members. Thank you for the fabulous leisure time we had together!

Contents

Supervisors	iii
Acknowledgements	v
Contents	vi
List of Figures	xi
1 Introduction	1
1.1 Company Presentation	2
1.2 Outline	3
2 Background	5
2.1 Non-Destructive Testing	5
2.2 MODERATO: a Radiography Monte Carlo Simulation	6
2.2.1 Industrial Radiography Principles	6
2.2.2 Monte-Carlo Modeling Process	8
2.2.3 MODERATO Specificities	9
2.3 Parallelization with Various Architectures	10
2.3.1 CPU Clusters	11
2.3.2 Intel MIC	12
2.3.3 GPGPU	12
2.4 Specific Ray Tracing Engines	13
2.4.1 Intel Embree	14
2.4.2 NVIDIA OptiX	15
3 Implementation with OptiX	17
3.1 OptiX Ray Tracing Basics	17
3.2 Programming Model Overview	19
3.2.1 Host Models	19
3.2.2 Device Programs	20
3.2.3 OptiX Ray Type	20
3.2.4 Internally Provided Semantics	21
3.3 Host Object Model	22
3.3.1 Context	22

3.3.2	Program	23
3.3.3	Variable and Buffer	24
3.3.4	Geometry	25
3.3.5	Material	25
3.3.6	GeometryInstance	26
3.3.7	GeometryGroup	26
3.3.8	Acceleration	27
3.4	Device OptiX Programs	27
3.4.1	Ray Generation Program	27
3.4.2	Exception Program	28
3.4.3	Closest Hit Program	28
3.4.4	Any Hit Program	29
3.4.5	Intersection Program	29
3.4.6	Bounding Box Program	31
3.5	CURAND: RNG on GPUs	31
3.5.1	Mersenne Twister: MTGP32	33
3.5.2	Combined Multiple Recursive RNG: MRG32K3A	34
3.5.3	XOR-Shift RNG: XORWOW	34
3.6	Interoperability with CUDA	35
3.7	Performance Guidelines	36
3.8	Programming Caveats	37
4	Implementation	39
4.1	Modeling Process	39
4.1.1	Projection Source	39
4.1.2	Inspected Object	40
4.1.3	Detector	41
4.2	Problems and Solutions	42
4.2.1	Serialized Access to Variables	42
4.2.2	Using CURAND within OptiX	42
4.2.3	Model of Photon Interactions	43
4.2.4	Object-Oriented Programming on GPUs	45
5	Tests and Results	47
5.1	Effectiveness of Fast-Math	47
5.2	Degradation with Atomic Operations	49
5.3	Influence of Kernel Size	49
5.4	Comparison of Acceleration Structures	51
5.5	OptiX Performance	53
5.6	Technical Assessment	55
5.6.1	Advantage of OptiX	55
5.6.2	Difficulties and Inconveniences	55
5.6.3	Following Work	56
6	Conclusion	57
A	Installation of OptiX	59

A.1	Pre-installation Actions	59
A.1.1	Verify the System Has a CUDA-Capable GPU	59
A.1.2	Verify GCC Compiler	60
A.2	Installation of CUDA Toolkit	60
A.2.1	Installation of the NVIDIA Driver	61
A.2.2	Installation of CUDA Toolkit	61
A.2.3	Verify the Installation of CUDA	62
A.3	Installation of OptiX SDK	63
A.3.1	Post-installation Setup	64
B	Acceleration Structures	65
C	Time Management	67
	Bibliography	69
	Abstract	i
	Résumé	ii

List of Figures

2.1	Industrial Radiography Principle	6
2.2	Mechanism of a nuclear power plant	7
2.3	Detector used by EDF	7
3.1	The OptiX ray tracing pipeline	17
3.2	Internal semantics provided by OptiX	21
3.3	CURAND vs. MKL	32
3.4	CURAND high performance RNGs	32
3.5	Initial state with a single bit at 1	33
4.1	OptiX photon interaction model.	44
5.1	Comparison of fast-math functions and standard functions	48
5.2	Effectiveness of atomic operations	50
5.3	Initialization difference between 2 kernels.	51
5.4	Execution time between different kernels.	52
5.5	Simulation results of OptiX and MODERATO.	53
5.6	Tracing performance with different node amounts on ATHOS cluster.	54
A.1	Verify the gcc compiler	60
A.2	CUDA test result	63
B.1	Acceleration Structures (1)	65
B.2	Acceleration Structures (2)	66
C.1	Task list.	67
C.2	Gantt diagram.	68

Chapter 1

Introduction

Monte Carlo methods are a common and accurate way to model particle transport by solving with few approximations the radiation transport equation [1, 2]. However, to ensure this accuracy, the Monte Carlo method sacrifices its computing performance on producing a large number of repeated random samplings [3], because of a very slow convergence rate. Therefore, the acceleration of computational Monte Carlo algorithms turns out to be necessary. Numerical acceleration methods are known to be very efficient, but only computational optimizations will be tackled in this work.

Recently, certain emerging computational accelerators such as General-Purpose Graphics Processing Units (GPGPU), CPU (Central Processing Unit) cluster and Xeon Phi are widely explored for their potentials on large-scale computing. Note that even these accelerators are far less powerful than the top-500 supercomputers; they possess more attractive ratios on performance/price and performance/watt [1]. In addition, Monte Carlo algorithms are ideally suited to parallel processing architectures and so are good candidates for acceleration using these tools [4]. For an item more specific, NVIDIA has published OptiX - a highly optimized ray tracing engine based on NVIDIA's CUDA (Compute Unified Device Architecture) architecture. This engine is created initially to handle graphics problems. However, instead of being a ray tracer itself, OptiX provides developers with a scalable framework for building any kind of ray tracing based applications [5]. By using this low-level ray tracing support, developers may obtain an impressive speedup with few optimizations on their original algorithms.

Électricité de France (EDF) uses radiographic inspection for systematic pipe control so as to ensure security on nuclear power plants [6]. MODERATO code is a software developed at EDF R&D (Research & Development) in order to simulate the inspection process. Using the Monte Carlo method, MODERATO traces photons emitted by gamma source and then simulates images obtained on radiographic films. It is a very useful simulation tool to help finding the best configuration before on-site inspection.

The present report describes the work I have done during my six months internship at EDF R&D. In order to benefit of all advantages of GPU accelerators, my mission was to evaluate the adaptation of MODERATO to the optimized ray tracing frameworks on these devices. The following part of this chapter will at first give a brief introduction about the organization: the structure of the entire R&D center will be presented as well as the research group I worked with. Then, outline of the report will be listed.

1.1 Company Presentation

EDF [7] is one of the world's largest energy producer that specializes in electricity, from engineering to distribution. The company's main operations are electricity generation and distribution; power plant design, construction and dismantling; energy trading; and transport. It is active in such power generation technologies as nuclear power, hydropower, marine energies, wind power, solar energy, biomass, geothermal energy and fossil-fired energy.

Due to a growing energy demand while tackling climate change and managing resource depletion, EDF keeps on innovation and remains the lowest carbon emitter among the major European energy companies, especially electric utilities, adapt fleet and customer offers to promote climate protection and reduce environmental impact [8].

In order to produce more environment-friendly electricity, the EDF Group invests lots of efforts on innovation activities. The seven international research centres (including three in France, one in Germany, one in the UK, one in Poland and one in China) enhance the Group's industrial performance and provide strong foundation of a solid future through medium-and long-range planning.

In particular, among all fifteen departments at R&D, the SINETICS (SIMulation NEu-tronique, Technologies de l'Information et Calcul Scientifique) is specialized in nuclear safety simulation and scientific computing. It consists of six research group [9]:

- ***Computing Infrastructure, Communication and Security Group (I2D):*** *Communications infrastructure on activities of network data and communications between heterogeneous systems, embedded communication systems and the safety of these infrastructures and systems.*
- ***Reactor Neutronic Simulation Group (I27):*** *Chains of reactor neutron calculations and the modeling of nuclear fuel behavior.*
- ***Virtual Reality and Scientific Visualization Group (I2C):*** *Pre and post treatment of numerical analyse, risk prediction and intervention preparation in controlled area.*
- ***Numerical Analysis and Models Group (I23):*** *Numerical methods and HPC strategies for simulations on reactor neutron behavior, non destructive testing (radiographic, ultrasonic), hydraulics and other mechanic activities, etc.*
- ***Cycle Safety and Physics Group (I28):*** *Development and justification of EDF's strategy on the management of fuel cycle, evaluating alternatives for the renewal of nuclear power plants.*
- ***Architecture of Information System and Scientific Computing Group (I2A):*** *Bringing together expertise in information systems architecture and software architecture for scientific computing applied to high performance computing.*

During these six months, I integrated in the I23 group and focused on scientific computing and its optimization with computational accelerators. The group has about 20 members. Generally speaking, they have a formal meeting every two weeks (réunion du groupe). Small technique gatherings are more frequent among the people in the same project. Communication for group members is convenient since all offices are closed. Moreover, EDF provides staff members with internal message and telephone service, which improve working efficiency as well.

1.2 Outline

A very initial investigation on the ray tracing CPU/GPU portability of MODERATO will be introduced in this report. In a first part I will introduce the radiographic inspection and its simulation with MODERATO. Then I will briefly give some details on several popular HPC (High Performance Computing) accelerators: CPU cluster, Xeon

Phi and GPGPU. At the end of this part, two available ray tracing framework for scientific calculations will be described.

Chapter 3 will cover how to build ray tracing softwares with OptiX framework. It first explains programming models defined in OptiX. OptiX *Host Objects* and *Device Programs* will be detailed with code examples. Besides, how to perform CUDA/OptiX communications and the comparison of three RNGs (Random Number Generators) will be presented. At last, OptiX performance guidelines and programming caveats help a more efficient implementation work.

The following chapter focuses on modeling process in OptiX. The procedure of porting the original code to OptiX will be described. Then, it goes over several important points of our OptiX implementation. Optimizations like reductions and selection of RNGs will be discussed. Problems and corresponding solutions will be fully detailed at the end of this chapter.

The final chapter introduces the tests we have done to evaluate and validate the OptiX implementation. Firstly, test configurations such as models of sources, inspected objects and detectors will be presented. Then, in order to verify effectiveness and accuracy of CUDA Fast-Math Intrinsic Functions (Compared to the traditional IEEE 754 standard), tests on this item will be presented with explanations. Moreover, kernel size can have an influence on ray tracing performance as well: even if a 1000×1000 kernel has been proved to be more productive than a 500×500 one on calculations, its poor efficiency during initialization may always be a non-negligible drawback. Therefore, some tests were realized to find the optimal balance between calculation and initialization. Another test concerns the degradation of serialized operations during parallel computing: utilization of atomic operations within OptiX and their accuracy will be discussed. The chapter also evaluates OptiX performance by comparing it with a CPU cluster. In the end, the chapter draws conclusions from the previous test results. Apart from the advantages on using OptiX, difficulties and inconveniences during implementation will be discussed. After these concluding remarks, a future road-map for this OptiX implementation is proposed, and the amount of work needed to make it featured.

Chapter 2

Background

2.1 Non-Destructive Testing

Non-destructive testing (NDT) is a wide group of analysis techniques to examine an object, material or system without impairing its future usefulness [10]. NDT methods may rely upon use of electromagnetic radiation, sound, and inherent properties of materials to examine samples. This includes eddy-current testing, which uses electromagnetic induction to detect flaws in conductive materials. The inside of a sample can be examined with penetrating radiation, such as X-rays or neutrons. Sound waves are utilized in the case of ultrasonic testing. Contrast between a defect and the bulk of the sample may be enhanced for visual examination by the unaided eye by using liquids to penetrate fatigue cracks. Liquid penetrant testing involves using dyes, fluorescent or non-fluorescent, in fluids for non-magnetic materials, usually metals. Another commonly used method for magnetic materials involves using a liquid suspension of fine iron particles applied to a part in an externally applied magnetic field (magnetic-particle testing). Thermoelectric effect uses thermal properties of an alloy to quickly and easily characterize many alloys. The chemical test, or chemical spot test method, utilizes application of sensitive chemicals that can indicate the presence of individual alloying elements. Electrochemical methods, such as electrochemical fatigue crack sensors, utilize the tendency of metal structural material to oxidize readily in order to detect progressive damage [11].

2.2 MODERATO: a Radiography Monte Carlo Simulation

2.2.1 Industrial Radiography Principles

Industrial radiography is a method for inspecting materials with potentially hidden defects by using the ability of high energy X-rays and gamma rays to penetrate various materials [12]. Industrial radiography is similar to medical X-ray technology in that a film records an image of an item placed between it and a radiation source.

The basic principle of the process is fairly simple and common to all radiography applications (Figure 2.1). The radiation from a controlled source penetrates the test item and expose a specially formulated film. As the radiation passes through the item, a portion of it is absorbed by the molecular structure of the material. The amount of radiation absorbed depends on the density and the composition of the material.

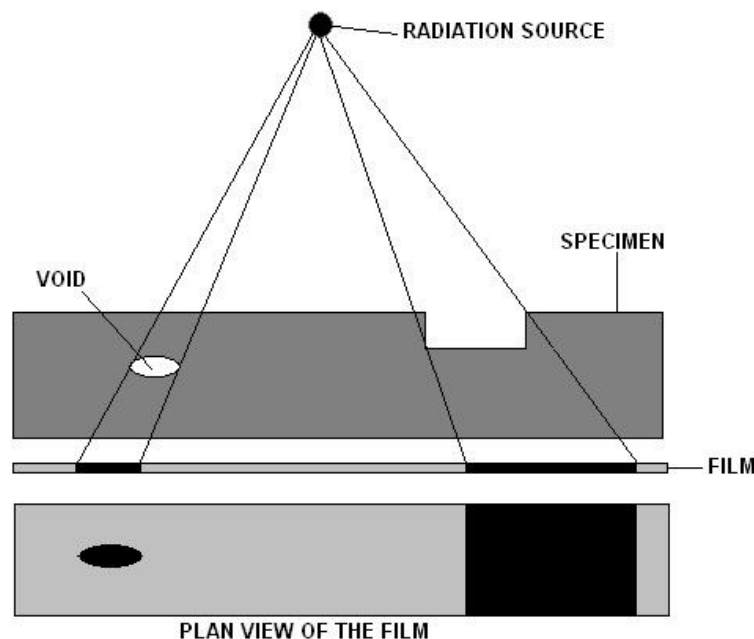


FIGURE 2.1: Industrial Radiography Principle [13].

As cracks, fissures, and any other defects in the material have different densities, they will be characterized by different exposure values as more or less radiation penetrates at those points during exposure. This creates a very accurate image of the internal structure of the item.

At EDF, these techniques are employed to control construction facilities like nuclear power plants, especially metallic components within the primary circuit, the secondary

circuit and the cooling system (Figure 2.2). The sources of radiation for industrial radiography depend on the process used. At present, EDF utilizes sources of cobalt-60 or iridium-192, which generate gamma radiation. Detectors juxtaposed to the specimen consist of a filter and a cassette with lead shields and one or two silver films (Figure 2.3).

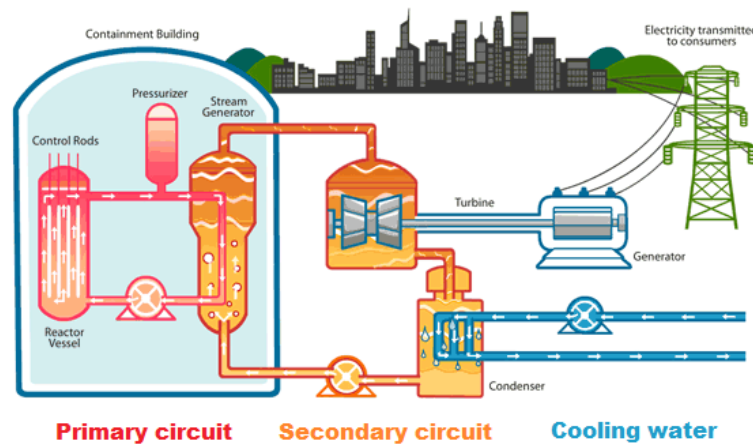


FIGURE 2.2: Mechanism of a nuclear power plant [14].

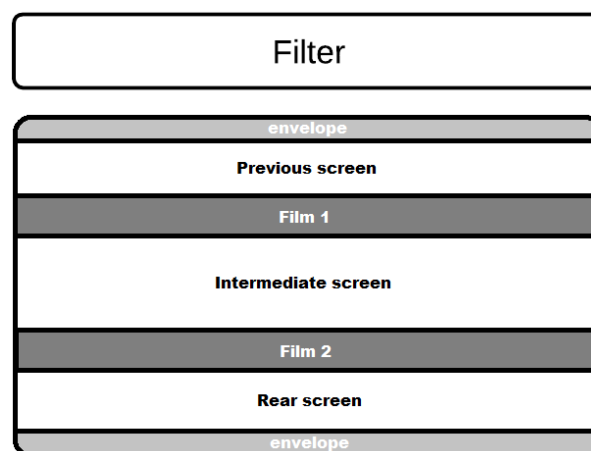


FIGURE 2.3: Detector used by EDF [15].

2.2.2 Monte-Carlo Modeling Process

Monte Carlo uses a statistical computing method for solving complex scientific computing problems. It uses random numbers to simulate the uncertainty of inputs to a problem and processes the repeated sampling of the parameter to obtain a deterministic result and solve problems that would otherwise be impossible. This method was originally pioneered by nuclear physicists involved in the Manhattan Project in late 1940s. It is named in reference to the biggest casino in the principality of Monaco.

Compared to deterministic methods that solve the particle transport equation with closed-form expressions, the Monte Carlo method attempts to simulate what happens in reality. When applied to photon transport in MODERATO, it simulates directly single photon's movement by distributing stochastically its energy, direction, free-path and corresponding interactions, from birth to death. This is a straightforward technique that works particularly well when the underlying probabilities of the process are known but the results are more difficult to determine. For example, calculations of risk in finance and business, analysis of quantitative genetics and nuclear reactor simulations, etc.

The essence of the Monte Carlo method is in using a given data-generating mechanism for the process model you wish to understand, produce new samples of simulated data, and examine the results of those samples. Although the modeling process may vary in each individual case, the following procedure would be common to all problems using this method [16]:

1. *Construct a simulated “universe” of randomizing mechanism whose composition is similar to the universe we wish to describe and investigate. The term “universe” refers to the system that is relevant for a single simple event.*
2. *Specify the procedure that produces a pseudo-sample which simulates the real-life sample in which we are interested. That is, specify the procedural rules by which the sample is drawn from the simulated universe. The simulation procedure must produce simple experimental events with the same probabilities that the simple events have in the real world.*
3. *If several simple events must be combined into a composite event, describe it in the procedure.*

4. *Calculate the probability of interest from outcomes of the re-sampling trials.*

The main drawback in using the Monte Carlo method, however, is that its convergence is governed by the central limit theorem. For many problems, obtaining sufficiently-low statistical error is slow compared to other approaches [2]. This is why any way of accelerating Monte Carlo methods is important for the scientific calculation community and why accelerators like CPU Clusters, MICs, GPGPUs are widely explored in this area. Since the final numerical results are obtained by re-sampling iteratively individual and disjoint events, the Monte Carlo method is well suited to parallel computation in its nature.

2.2.3 MODERATO Specificities

MODERATO produces accurate numerical results and predicted images before real examinations, which allows operators to optimize inspection configurations, verify NDT effectiveness and help training technicians [15]. Areas with heat and radioactivity (inside a reactor for example) can be inaccessible to operators. Therefore, MODERATO will be employed to analyze the test and help determining an optimal configuration. Without a simulation tool like this, the operation can be both dangerous and costly. Another function of MODERATO is to qualify test methods. The code simulates operating process and checks if the method can meet expected needs: can this method find potential defects? Is it capable of locating them all? Since real radiographic tests cost a lot, a qualification before using is very profitable. So as to train inspection controllers, MODERATO plays an important role as well. Although this virtual simulation never replaces real practice, it makes technicians quickly get a preview of their work and have better awareness on the whole operation.

The simulations of a radiographic inspection used by EDF pose particular problems, and none of the existing codes at that time can solve them well. As a result, in order to satisfy all these specificities, MODERATO was created with following features [15]:

- *Handle the thickness of inspected objects and manage to simulate interactions inside.*
- *Model the specific detector with several films and metal screens. Note that characteristics of each film vary a lot, a microscopic modeling is necessary*

instead of a macroscopic one with experimental calibration of a large number of transfer functions. However, this work had never been done before MODERATO. Moreover, the modeling of the rear screen within detectors is rather delicate [15].

- **Deal with complex geometries.** *Analytical parameterized geometries are described precisely by their borders. Using the CADSurface and CADContour system, MODERATO is capable of compositing many complex geometries with several basic ones.*

2.3 Parallelization with Various Architectures

The development of MODERATO has begun since the end of 1990s when parallel computing was not commonly used like nowadays. The original implementation was a scalar one which processed a single pair of operands at a time. Then, parallelism work has been done during the past years, which converts the single-process source code to a multi-process one.

Parallelization of MODERATO was realized with MPI (Message Passing Interface): a major computing process is responsible for centralizing all other processes by distributing them with data and then collecting it back. The whole process from generating photons to simulating their multi-diffusions within inspected objects is treated by sub-processes. Afterwards, all photons which succeed in reaching the detector are sent back to the major process which performs the detector treatments: modeling films, generating result files, etc. In theory, the gain of this parallelization is proportional to the number of sub-units: a certain number of sub-units may bring almost the same amount of speedup.

Although hardware accelerators is a relatively rising technology, there has been much research about porting radiographic Monte Carlo on it. Among all these work, a group at Rensselaer Polytechnic Institute in USA has made significant achievement [17–20]. They developed a fast Monte Carlo code ARCHER (Accelerated Radiation-transport Computation in Heterogeneous EnviRonments) to model medical CT (Computed Tomography) imaging process on heterogeneous computing systems (or hardware accelerators). Simulating the transport of low-energy (1~140keV) photons, ARCHER contains three code variants: ARCHER-CTcpu, ARCHER-CTgpu and ARCHER-CTcop to run in parallel on the multi-core CPU, GPU and coprocessor architectures respectively.

These three versions have been developed correspondingly in MPI-OpenMP, CUDA and offload OpenMP. According to their study, the ARCHER results agreed well with those from the production code Monte Carlo N-Particle eXtended (MCNPX). It was found that all the code variants were significantly faster than the parallel MCNPX running on 12 MPI processes, and that the GPU and coprocessor performed equally well, being 2.89~4.49 and 3.01~3.23 times faster than the parallel ARCHER-CTcpu running with 12 hyper-threads cores [18]. The three computing architectures mentioned above will be presented in the following of this section.

2.3.1 CPU Clusters

A computer cluster is a single logical unit consisting of multiple computers that are linked through a LAN (Local Area Network). The networked computers essentially act as a single, much more powerful machine. The major advantages of using computer clusters are [21]:

- **Cost efficiency:** *the cluster technique is cost effective for the amount of power and processing speed being produced. It is more efficient and much cheaper compared to other solutions like setting up mainframe computers.*
- **Processing speed:** *multiple high speed computers work together to provided unified processing, and thus faster processing overall.*
- **Improved network infrastructure:** *different LAN topologies are implemented to form a computer cluster. These networks create a highly efficient and effective infrastructure that prevents bottlenecks.*
- **Flexibility:** *unlike mainframe computers, computer clusters can be upgraded to enhance the existing specifications or add extra components to the system.*
- **High availability of resources:** *If any single component fails in a computer cluster, the other machines continue to provide uninterrupted processing. This redundancy is lacking in mainframe systems.*

So as to benefit the advantages, parallel MODERATO was recently tested by Matthieu ANNE on a new EDF R&D cluster (ATHOS) with few changes of original source code. Test results show that MODERATO is greatly accelerated with large-scale parallelism, further details about this test will be presented in Section 5.5.

2.3.2 Intel MIC

Intel Many Integrated Core Architecture (or Xeon Phi) is a coprocessor computer architecture running its own operating system. The coprocessor core implements most of the new instructions associated with 64-bit extension. However, the only vector instructions supported are the initial Intel Many Core Instruction set. There is no support for Intel MMX technology, Intel SSE, or Intel AVX in the coprocessor cores, although the scalar math unit (x87) remains integrated and fully functional.

MIC provides an ideal execution vehicle for Monte Carlo-related applications. Building around the x86-64 instruction set architecture, using the very same programming model as the other multi-core processors, Xeon Phi extends the parallel execution infrastructure with little or no modification of source code, according to an Intel developer [22]. The main advantage of using MIC, with respect to other co-processors and accelerators, is the simplicity of the porting [23]. Programmers do not have to learn a new programming language but may compile their source codes specifying MIC as the target architecture. The classic programming languages (HPC-Fortran, C, C++) as well as the parallel paradigms: OpenMP or MPI-may be directly employed regarding MIC as a “classic” x86 based (many-core) architecture.

Aside from ARCHER, there are few studies on performing Monte Carlo particle transport on Xeon Phi. The only research team I found in this area focuses on proton transport in Monte Carlo codes [24, 25]. It is worth noting that because of its totally new vector processing unit, porting work on MIC with numerous vector operations may not be more straightforward than that on GPGPU.

2.3.3 GPGPU

General-purpose computing on graphics processing units is to use the initial rendering graphics GPU for normal computation. Compared to CPUs, they have a higher aggregate memory bandwidth, much higher Floating-point Operations Per Second (FLOPS), and lower energy consumption per FLOP [2]. Because one of the main obstacles in exascale computing is power consumption, many new supercomputing platforms are gaining much of their computational capacity by incorporating GPUs into their compute nodes.

A study led by Martinsen concerns with modeling photon transport in anisotropy media on using CUDA architecture [1]. Their implementation, processing roughly 110 million scattering events per second, was found to run more than 70 times faster than a similar, single-threaded implementation on a 2.67 GHz desktop computer.

Tickner [4] has implemented a general-purpose code that computes the transport of high energy photons through arbitrary 3-D geometry models, simulates their physical interactions and performs tallying and variance reduction. It describes a new algorithm that provides a good match with the underlying GPU multiprocessor hardware design. Benchmarking against an existing CPU-based simulation running on a single-core of a commodity desktop CPU demonstrates that the code can accurately model X-ray transport, with an approximately speed-up factor of 35.

Furthermore, Kalantzis and Tachibana [26] carried out a study of hybrid (OpenMP and CUDA) MC tracking implementation. A maximum speedup of 67.2 (compared to a single threaded CPU version) was achieved for the primary GPU-based MC code, while a further improvement of the speedup up to 20% was achieved for the hybrid approach. The results indicate the encouraging capability of this CPU–GPU implementation for accelerated MC calculations of particle tracks without loss of accuracy.

According to these significant results, we conclude that GPGPU is a powerful accelerator for scientific computing. These speedup factors make it very attractive to use in extremely parallel, computationally-intensive Monte Carlo simulations.

2.4 Specific Ray Tracing Engines

An essential part of MODERATO is its ray tracing package, which is in charge of tracking the entire trajectory of every photon and accelerating with bounding boxes. Given that the photon histories are completely independent, this tracking model fits perfectly parallel accelerators. However, even if parallelization is easy, efficient vectorization is hard since it requires deep knowledge about hardware architecture. The typical ray tracing algorithms can be highly irregular, which poses serious challenges for anyone trying to exploit the full raw computational potential of a computing accelerator [5]. Consequently, several specific ray tracing engines address those challenges and provide

frameworks for harnessing the enormous computational power of the accelerator to incorporate ray tracing into interactive applications. In this manner, interactive ray tracing is finally feasible for developers without a Ph.D. in HPC and a team of ray tracing engineers.

2.4.1 Intel Embree

Embree is an open source ray tracing framework for x86 CPUs [27]. Although it's an API explicitly designed to achieve high performance in professional rendering environments, it's said that Embree doesn't limit within a complete rendering application. In addition, the manual [28] emphasizes that its ray tracing kernel has been optimized for incoherent Monte Carlo ray tracing algorithms.

Embree is a cross-platform API: Windows, Linux and Mac OS X are all supported in both 32bit and 64bit modes. In particular, the Xeon Phi version is only available in 64-bit Linux distribution.

In terms of geometry description, both triangle meshes and analytical methods are supported. Users may choose either a dynamic or a static way to establish geometry primitives. Like OptiX, Embree also provides automatic acceleration structures for the ray tracing process, but this acceleration is relatively simple. Users select a tracing mode (coherent, incoherent, robust, etc.) instead of specifying a particular structure in OptiX. Moreover, bounding box of meshes are automatically built by Embree ray tracing mechanism. Embree supports *Hair Geometry* as a geometry mode, which consist of multiple curves represented as the cubic Bézier curve with varying radius per control point. This new feature may be helpful to describe more accurate *CADContour* in MODERATO.

Ray tracing mechanism in Embree is similar to OptiX, it supports finding the closest hit point of a ray segment with *rtcIntersect* functions, and determining if any intersection point exists by using *rtcOccluded*. The API supports per geometry Embree *Filter* callback functions that are invoked for each intersection found during the *rtcIntersect* or *rtcOccluded* calls. However, the *Filter* function is only supported for triangle mesh

geometry. Since ray reflections or refractions in rendering graphic necessitate such functions to change ray directions and cast new rays, we would like to investigate it for analytical geometries before modeling the photon interaction.

Even if the manual says that the API provides a low-level ray tracing mechanism, further investigation is required to make clear how to combine the ray tracing kernel with general programs. Note that Embree has been proposed since two years ago and as far as I know, until now, there is no paper published concerning Embree for Monte Carlo simulations. This may imply it's not suitable for general-purpose calculations.

2.4.2 NVIDIA OptiX

As a more specific GPGPU extension, OptiX provides a basic and scalable ray tracing framework for graphics as well as general-purpose computing. Wherever possible, OptiX avoids specification of ray tracing behaviors and instead provides mechanisms to execute user-provided CUDA C code [5]. This low-level support focuses exclusively on the fundamental computations required for ray tracing and avoids embedding rendering-specific constructs. The engine presents mechanisms for expressing ray-geometry interactions and does not have built-in concepts of lights, shadows, reflectance, etc [29].

Geometry representation within OptiX is extremely flexible and agrees well to the request of MODERATO: both analytical methods and mesh methods are available. It's users themselves that describe geometries as well as corresponding intersections and interactions. OptiX just provides a ray tracing interface between user-supplied programs and GPGPU architectures. Besides this flexibility, OptiX provides optimizations that may take months for developers to replicate by themselves. Its main advantage is automatically creating high-quality acceleration structures for traversing the primitives in a geometry scene [5].

OptiX Prime was integrated in OptiX since the 3.5 version. Being compared with OptiX, the Prime is a more low-level API for applications which are just designed to find intersections. On removing all features around rendering and shading, OptiX Prime handles meshes and rays and it returns the intersections at over 300 million rays per second on a single GPU [30]. What interests us more is that OptiX Prime also provides an efficient CPU fallback when a suitable GPU is not present. However, analytical

parameterized geometries are no more supported, which would be a main drawback for using it to optimize MODERATO. Another concern is that commercial applications require a commercial license to redistribute OptiX 3.5 and later editions. Since the use of MODERATO is not restricted within EDF, additional cost for licenses will pressure its commercial use.

One can find some researches on using OptiX for scientific Monte Carlo simulations. Bergmann, now post-doc at Nuclear Engineering Department of UC Berkley, just finished his dissertation on developing a framework for continuous energy Monte Carlo neutron transport on GPUs. The framework, which is called WARP, accelerates Monte Carlo simulations while preserving the benefits of Monte Carlo method. Optimized, high-performance GPU libraries such as CUDPP (CUDA Performance Primitives: performing parallel reductions, sorts and sums), CURAND and OptiX are used wherever possible [2]. In an initial testing where 106 source neutrons per criticality batch are used, WARP produces results agreeing well with that of MCNP 6.1 and Serpent 2.1.18 and takes much less computation time. He concludes that WARP's performance on a NVIDIA K20 is equivalent to approximately 45 AMD Opteron 6172 CPU cores.

Other researches draw conclusions similar to that of Bergmann [31, 32]: OptiX presents remarkable tracking performance as well as promising agreements with reference MC codes, for example, MCNP and GEANT4. They emphasized that by using OptiX, GPU implementation problems like keeping all available GPUs busy, minimizing memory transfers and optimizing use of hardware will be automatically solved.

Chapter 3

Implementation with OptiX

3.1 OptiX Ray Tracing Basics

OptiX implementation consists of two major parts: a list of object-based models on the host and eight programs describing ray tracing specifics on the device. Host objects are in charge of establishing test scenes where take place all ray tracing algorithms. They assist in initializing geometries, passing data between host and device, launching OptiX kernels, handling post-treatments and so on. The combination of user-defined programs and built-in tracking algorithms forms the OptiX ray tracing control flow (Figure 3.1).

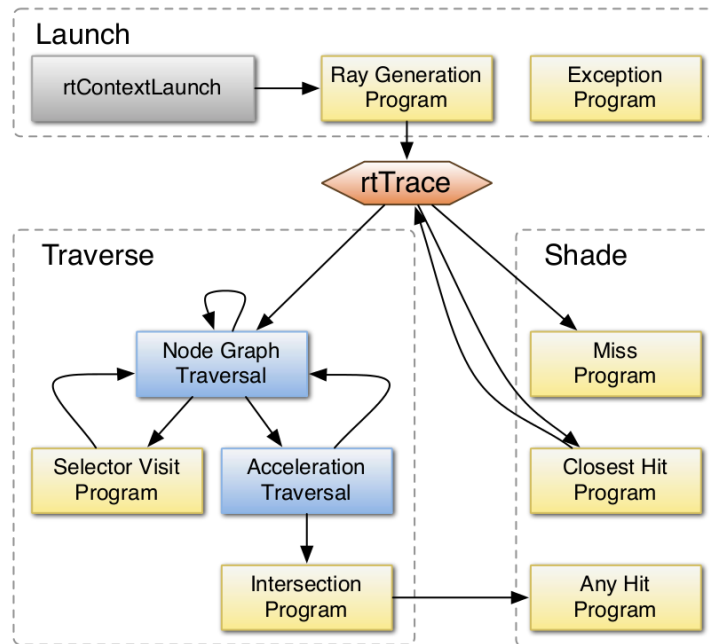


FIGURE 3.1: The OptiX ray tracing pipeline [29].

Launching an OptiX kernel analogous to that of CUDA: after a ray tracing launch call (defined as *rtContextLaunch* in OptiX) from CPU, all operations jump to the GPU side, where the first of the eight device programs, *Ray Generation Program*, is automatically triggered. This first program initializes rays, their starting points and directions, and specifies the data that they carry. Once this is done, the tracking work begins. *Bounding Box Program* accelerates tracking by reporting only potential primitives along a ray's trajectory. *Intersection Program* allows all these geometries to record their crossing points with the ray and send this information to following *Hit Programs*. Any *Hit Program* is called every time a ray-object interface is found. Afterwards, among all the coordinates reported by *Intersection Program*, OptiX will automatically point out the first intersection of each ray and call its corresponding *Closest Hit Program*. It's in *Closest Hit Program* that developers update ray information (direction, position and specific payload data) and continue tracing the new ray. For performance issue, any ray without potential intersections will be immediately killed after *Intersection Program*.

The main idea of ray tracing algorithms is representing ray propagations with a t value. Any 2D or 3D coordinates will be transformed into this key word. For example, movement of a ray in the test scene is described as:

$$\vec{p} = \vec{o} + t \cdot \vec{d} \quad (3.1)$$

Where \vec{o} represents the origin of the ray and \vec{d} represents its direction. A geometry primitive like sphere is the set of points at a given distance r from a center point \vec{c} . It's easily expressed in vector form:

$$(\vec{x} - \vec{c})^2 = r^2 \quad (3.2)$$

The equation of intersection points of the above ray and sphere can be substituted using Equation 3.1 and Equation 3.2:

$$(\vec{o} + t\vec{d} - \vec{c})^2 = r^2 \quad (3.3)$$

$$(\vec{o} - \vec{c})^2 + 2t(\vec{o} - \vec{c}) \cdot \vec{d} + t^2 \cdot \vec{d}^2 = r^2 \quad (3.4)$$

As a result, the t value: $t = \frac{-(2(\vec{o}-\vec{c}) \cdot \vec{d}) \pm \sqrt{(2(\vec{o}-\vec{c}) \cdot \vec{d})^2 - 4((\vec{o}-\vec{c})^2 - r^2)}}{2}$.

3.2 Programming Model Overview

As stated in the last section, OptiX programming model is constituted by the host code and the GPU device programs. This section gives an overview of these objects, programs and variables defined on CPU and executed on GPU. Note that models related to rendering concerns were not employed in our MC simulation, only the used ones will be presented and detailed in this report.

3.2.1 Host Models

- **Context:** The top object includes all other models in OptiX. Each *Context* is an instance of the ray tracer.
- **Program:** An interface between user-defined codes and the eight OptiX device programs.
- **Variable:** A label used in communications between CPU and GPU. It must be bound to a *Buffer*.
- **Buffer:** A multidimensional (host \Leftrightarrow device, CUDA \Leftrightarrow OptiX, OpenGL \Leftrightarrow OptiX, Direct3D \Leftrightarrow OptiX) array that transforms a *Variable*.
- **Geometry:** Meshes or user-defined primitive that can be attached to multiple *Materials* to handle multiple ray types (or interactions in MODERATO).
- **Material:** A container of *Any Hit Program* and *Closest Hit Program* attached to a specified OptiX ray.
- **GeometryInstance:** A combination of *Geometry* and *Material* objects.
- **GeometryGroup:** A set of *GeometryInstance* objects.
- **Acceleration:** A structure which is bound to a *GeometryGroup* and provides automatic accelerations.

Further information on these objects will be detailed in Section 3.3.

3.2.2 Device Programs

- **Ray Generation Program:** The entry point into the device-side ray tracing pipeline; where the radiation source in MODERATO is modeled.
- **Exception Program:** Program triggered while stack overflow or other errors, employed only for debug.
- **Closest Hit Program:** Handle calculations at the nearest intersection point, invoked by our MODERATO implementation to update cross-sections, create and trace new photons, etc.
- **Any Hit Program:** Called whenever a traced ray finds a new potential intersection point, unapplied in our implementation.
- **Intersection Program:** Where intersection detections take place, passing new free-path to Closest Hit Program in our implementation.
- **Bounding Box Program:** User-programmed bounding box, invoked during acceleration.
- **Miss Program:** Killing a traced ray when it misses all geometry primitives, unapplied in our implementation for performance concern.

PTX is a virtual assembly language that can be easily interpreted by hardware architectures and requires no further compilation [2]. All OptiX *Programs* must be compiled to *.ptx* files in order to be combined with their host objects. These programs are further detailed in Section 3.4.

3.2.3 OptiX Ray Type

OptiX *Ray* is an encapsulation of a ray mathematical entity which is contained within the *optix::* namespace. Except for some ray properties pre-defined within this object, every *Ray* carries an user-defined *rtRayPayload*. This payload contains variables we need for the following data processing. In our implementation of MODERATO, for example, it includes cross-section ratios, current free-path, photon energy, interaction counter, etc. Note that only an OptiX *Ray* object can be accepted as an argument of *rtTrace* function.

In order to take full advantages of pre-defined properties and avoid reloading them in a ray payload (which may degrade ray tracing performance), public attributes of OptiX *Ray* are listed as follows:

```

1  /* The origin of the ray; note that float3 is a CUDA      *
2  *  variable that corresponds to Vec3d in MODERATO.      */
3  float3 origin;
4
5  float3 direction; // The direction of the ray
6
7  unsigned int ray_type; // The ray type associated with this ray
8
9  // The min and max extents associated with this ray
10 float tmin;
11 float tmax;
```

3.2.4 Internally Provided Semantics

Name	rtLaunchIndex	rtCurrentRay	rtPayload	rtIntersectionDistance
Access	read only	read only	read/write	read only
Description	The unique index identifying each thread launched by <code>rtContextLaunch {1 2 3}D</code> .	The state of the current ray.	The state of the current ray's payload of user-defined data.	The parametric distance from the current ray's origin to the closest intersection point yet discovered.
Ray Generation	Yes	No	No	No
Exception	Yes	No	No	No
Closest Hit	Yes	Yes	Yes	Yes
Any Hit	Yes	Yes	Yes	Yes
Miss	Yes	Yes	Yes	No
Intersection	Yes	Yes	No	Yes
Bounding Box	No	No	No	No
Visit	Yes	Yes	Yes	Yes

FIGURE 3.2: Internal semantics provided by OptiX [5].

Certain variables can be frequently invoked by all eight OptiX programs. For example, a *Ray* generated by *Ray Generation Program* would be processed everywhere in the ray tracing pipeline: in *Closest Hit Program* for changing direction, in *Intersection Program* for finding potential interfaces, etc. Under the condition of large-scale parallelism, a global access to such variables without thread conflicts will be highly appreciated. To make it simple, OptiX manages several internal semantics for this program-variable binding. Figure 3.2 summarizes in which types of program these semantics are available, along with their access rules from device programs [33].

3.3 Host Object Model

Source code examples in either *OptiX Programming Guide* or *OptiX Quickstart Guide* only concern on C API of OptiX. As a result, in order to be familiar with the C++ wrapper and integrate OptiX in an existing object-oriented implementation, it's highly recommended to consult OptiX SDK and *OptiX API Reference*. OptiX SDK provides various source code samples and is installed in the root directory by default.

3.3.1 Context

Context is the top node among the whole OptiX hierarchy. It encapsulates all resources for subsequent ray tracing control flows. Creation and destruction of this object are simple:

```

1  optix::Context context = Context::create(); // Setup context
2
3  /* In MODERATO, we define that there are two types of rays. The first is normal ray, which
   models photons outside inspected objects. The second, called interaction ray, is responsible
   for representing photons that may change their energies, directions, free paths, etc.*/
4  context->setRayTypeCount( 2 );
5
6  /* Indicate camera numbers in traditional ray tracing algorithms, here it figures the projection
   source number. We could therefore handle multi-source situation. */
7  context->setEntryPointCount( 1 );
8  ..... // a lot of codes ...
9
10 context->destroy(); // Clean up and invalidate all resources

```

OptiX kernel launch is realized by using *Context*'s public member *launch*. This host instruction starts the device ray tracing pipeline. The maximum kernel size varies according to GPU card's [CUDA Compute Capability](#) as well as their memory size. Quadro FX 1800, a card with Capability 1.1, may afford 500×500 launch indices. A 2.0 card like Quadro 5000, may increase up to 1000×1000. The ceiling kernel size of each card is indicated nowhere in OptiX manual. Developers should explore it based on their own implementations.

```

1  .....    // Previous code, initializations ...
2
3  // Check the given context and all of its associated OptiX objects for a valid state and create
   final computation kernel from the given programs
4  context->validate();
5  context->compile();
6
7  /* Launch a width*height OptiX kernel for source "0" (only one source). */
8  context->launch( 0, width, height );
9
10 .....    // Collecting GPU data, post process...
```

Needless to say, even 1000×1000 kernel is not big enough to simulate trillions of photons at the same time. We can use *for* loop to achieve this purpose. Note that GPU computation has time-out concerns, simulations with a great number (billions or more, depends on the card) of photons require loops on both CPU and GPU side:

```

1  /* Launch a 1000*1000 OptiX kernel for 1000 iterations: a trillion photons created */
2  for(i=0;i<1000;i++){
3      context->launch( 0, 1000, 1000 );
```

3.3.2 Program

Program object is created from the *.ptx* file. For instance, a *Ray Generation Program* is defined in a CUDA file named *ray.cu*.

```

1  RT_PROGRAM void ray_gen(){
2      for(i=0;i<N;i++){
3          ... // cast photons....
4      }
5  }
```

After compiling *ray.cu* into *ray.ptx*, in the C++ main function, we establish the combination between host *Program* object and device program *ray-gen* by calling *createProgramFromPTXFile*.

```

1 // Set PTX file path
2 std::string ptx_path( ptxPath( "ray" ) );
3
4 // create host object "ray_gen_program" with device program "ray_gen"
5 optix::Program ray_gen_program = context>createProgramFromPTXFile( ptx_path, "ray_gen" );
6
7 // void optix::ContextObj::setRayGenerationProgram ( unsigned int entry_point_index, Program
   program )
8 context->setRayGenerationProgram( 0, ray_gen_program );

```

3.3.3 Variable and Buffer

OptiX *Variable* and *Buffer* pass data between the host and the device. Communication of general variable types has been encapsulated within context's `[]` operator. In particular, transmission of tables or other special variable types will be implemented in another way:

```

1 // General types transfered by using [] operator
2 context["ThetaMax"]->setFloat( 4.f * 3.14159f/180 );
3 context["O_source"]->setFloat( 0.0f, 0.0f, 55.f );
4 context["sphere"]->setFloat( 0.f, 0.f, 0.f, 50.f );
5
6 optix::Variable output = context["output"]; // a 1D table
7 // A 1D table with 15 user format elements
8 optix::Buffer output_buffer = context->createBuffer( RT_BUFFER_INPUT_OUTPUT,
   RT_FORMAT_USER, 15 );
9 // User format: unsigned long long int
10 output_buffer->setElementSize(sizeof(unsigned long long int));
11 // Warning: must be bound to a Buffer.
12 output->set(output_buffer);
13
14 ..... // Kernel launch, device calculation, etc ...
15
16 // Then, we use map to collect data from GPU
17 void* o_data = output_buffer->map();
18 unsigned long long int* result = (unsigned long long int*) o_data;

```

Within CUDA files, these global variables are declared just after header files:

```

1  #include ....
2
3  // Accessible to all Programs within the file
4  rtDeclareVariable(float ,           ThetaMax, , );
5  rtDeclareVariable(float3 ,          O_source, , );
6  rtDeclareVariable(float4 ,          sphere, , );
7  rtBuffer<unsigned long long int, 1> output_buffer;
```

3.3.4 Geometry

A *Geometry* offers an interface to combine geometry representation with the ray tracing mechanism. This host object just passes setup parameters to its corresponding device *Intersection Program* and *Bounding Box Program*. It's only on the device that we describe geometry primitives and determine potential intersection points.

```

1  // Search device programs in the "object.ptx" (compiled from "object.cu") file
2  std::string ptx_path( ptxPath( "object" ) );
3  Geometry sphere = context->createGeometry();
4  // There is only one sphere in our case
5  sphere->setPrimitiveCount( 1u );
6
7  // Set bounding box with the program named "bounds" in the "object.ptx"
8  sphere->setBoundingBoxProgram(context->createProgramFromPTXFile(ptx_path,"bounds"));
9  // "intersect" program establishes sphere description and intersection
10 sphere->setIntersectionProgram(context->createProgramFromPTXFile(ptx_path,"intersect"));
11
12 // Sphere parameter: origin (0,0,0) and radius=50
13 sphere["sphere"]->setFloat( 0.f, 0.f, 0.f, 50.f );
```

3.3.5 Material

A *Material* encapsulates the actions when a photon intersects a primitive or reaches the end of its current free-path. Two types of *Hit Programs* can be assigned to a *Material*, but only *Closest Hit Program* is employed in our implementation. It is important to note that *Hit Programs* are bound to *Materials* per ray type, which means each *Material* can actually hold more than one *Closest Hit Program* [5].

```

1 // Hit Program is situated in "object.ptx"
2 std::string ptx_path( ptxPath( "object" ) );
3 Material matl = context->createMaterial();
4 Program ch = context->createProgramFromPTXFile( ptx_path, "closest_hit" );
5 /* The Hit Program corresponds to ray type 0 (normal ray), more programs can be held:
6  * matl->setClosestHitProgram( 1, ch1 );
7  * matl->setClosestHitProgram( 2, ch2 ); */
8 matl->setClosestHitProgram( 0, ch );

```

3.3.6 GeometryInstance

A *GeometryInstance* represents a coupling of a single *Geometry* with a set of *Materials*. Note that multiple geometry instances are allowed to refer to a single geometry object, enabling a geometric object with different materials. Likewise, materials can be reused between different geometry instances.

```

1 // Create object with 1 geometry and 2 materials
2 GeometryInstance object = context->createGeometryInstance();
3 object->setGeometry( sphere );
4
5 object->setMaterialCount( 2 );
6 // One material corresponds to the normal ray
7 object->setMaterial( 0, sphere_matl );
8 // Another for interaction ray
9 object->setMaterial( 1, interaction_matl );

```

3.3.7 GeometryGroup

A *GeometryGroup* contains all *GeometryInstances* in a test scene. In our first test MODERATO, for example, it represents a collection of an inspected sphere-object and a detector in parallelogram:

```

1 // Place all in geometry group
2 GeometryGroup geometrygroup = context->createGeometryGroup();
3 geometrygroup->setChildCount( 2 );
4 // object and detector are GeometryInstances defined before
5 geometrygroup->setChild( 0, object );
6 geometrygroup->setChild( 1, detector );

```

3.3.8 Acceleration

Acceleration structures are important for speeding up the ray tracing process. Generally speaking, they decompose the hierarchical scene geometries into several basic boxes and cull non-intersection regions of space at the very beginning of the trace. The traversal is therefore accelerated by querying less geometries.

There are many different types of acceleration structures, each with their own advantages and drawbacks (Appendix B). Note that no single type of acceleration structure is optimal for all scenes, it's developers themselves who find the best choice for their own implementations. Comparison among different structures for MODERATO test will be discussed in Section 5.4.

```

1  /* Set acceleration with a "Bvh" builder and a "BvhCompact" traverser.
2  * More details on introduction of OptiX acceleration structures see AppendixB */
3  geometrygroup->setAcceleration( context->createAcceleration("Bvh","BvhCompact") );

```

3.4 Device OptiX Programs

3.4.1 Ray Generation Program

The following example implements a simple ray generation model. Each ray is generated by host-defined origin and direction with `tmin=scene.epsilon`, `tmax=RT_DEFAULT_MAX`. The nature of these rays is "normal_ray". Then, the user-defined payload will be charged with several photon characteristics: energy, diffusion counter, cross-section, etc. `rtTrace` initialize the trace with a coupling of one ray and one payload.

```

1  // A user defined payload
2  struct Photon{
3      float3 S;           //S1, S2, S3
4      float  energy;      // Photon energy
5      int    nDiffusions;  // Diffusion counter
6      ...
7  };
8
9  // Variables passed from the host side
10 rtDeclareVariable(unsigned int,    normal_ray, , );
11 rtDeclareVariable(float ,          scene_epsilon, , );

```

```

12 rtDeclareVariable(float3, ray_origin, , );
13 rtDeclareVariable(float3, ray_direction, , );
14 rtDeclareVariable(rtObject, top_object, , );
15
16 RT_PROGRAM void ray_gen(){
17     optix::Ray ray = optix::make_Ray(ray_origin, ray_direction, normal_ray, scene_epsilon,
        RT_DEFAULT_MAX);
18
19     Photon payload;
20     payload.energy = 1.f;
21     payload.nDiffusions = 0;
22     ...
23     rtTrace(top_object, ray, payload);
24 }

```

3.4.2 Exception Program

Exception Program is invoked when OptiX kernel encounters errors. The information provided by this program would be useful for debug. So as to have better tracing performance, however, it can be removed after application development.

```

1 rtDeclareVariable(uint2, launch_index, rtLaunchIndex, );
2
3 RT_PROGRAM void exception()
4 {
5     const unsigned int code = rtGetExceptionCode();
6     rtPrintf( "Caught exception 0x%X at launch index (%d,%d)\n", code, launch_index.x,
        launch_index.y );
7 }

```

3.4.3 Closest Hit Program

Closest Hit Program is in charge of updating photon properties in our MODERATO implementation. Only in this program can we launch new rays (which means update randomly free-path, direction, interaction types, energy, etc.) and trace it.

```

1 rtDeclareVariable(float, t_hit, rtIntersectionDistance, );
2 rtDeclareVariable(Photon, pd, rtPayload, );
3

```



```

4  RT_PROGRAM void closest_hit(){
5      // Determine intersection point and change direction
6      float3 hit_point = ray.origin + t_hit * ray.direction;
7      new_direction = make_float3...
8
9      pd.energy ... // Other updates
10
11     // Launch a new ray at hit point and trace it
12     optix::Ray new_ray(hit_point, new_direction, normal_type, scene_epsilon, RT_DEFAULT_MAX
        );
13     rtTrace(top_object, new_ray, pd);
14 }

```

3.4.4 Any Hit Program

An *Any Hit Program* is called wherever ray tracer finds an intersection point. It's necessary to point out that for all intersections along a cast ray, *Any Hit Program* corresponding to each intersection point won't be invoked in order. In MODERATO, it's expected to find nearest intersection point, take update actions and launch the new ray. Unfortunately, *Any Hit Program* matches none of these features because *rtTrace* is only available in *Ray Generation Program* and *Closest Hit Program*. Furthermore, disabling useless programs contributes to better computing performance. Consequently, this program was not applied in our implementation.

However, *Any Hit Program* could be very practical for the 2D1D neutronic simulation, where neutrons traversing geometries without changing direction or other properties in each intersection point are expected to be recorded. The problem of intersection sort will come into focus in this situation.

3.4.5 Intersection Program

Code examples of a sphere *Intersection Program* are presented in this sub-section. They demonstrate how to build a simple analytic geometry and query its ray-object intersections.

Once the intersection program has determined the t value of a ray-primitive intersection, it must report the result by calling a pair of OptiX functions, *rtPotentialIntersection* and

rtReportIntersection. Note that another semantic, *attribute*, is employed in this program. It's used to pass variables from *Intersection Program* to *Hit Programs*. Attribute variables can only be written in *Intersection Program* and only be placed between these two functions. In the example below, *attribute distance* will be sent to *Closest Hit Programs* for signaling photons' traverse distance within sphere:

```

1  rtDeclareVariable( float4,          sphere, , );
2  rtDeclareVariable( optix::Ray, ray, rtCurrentRay, );
3  rtDeclareVariable( float, distance, attribute distance, );
4
5  // Intersection for sphere geometry
6  RT_PROGRAM void intersect(int primIdx){
7      float3 center = make_float3(sphere);
8      float3 O = ray.origin - center;
9      float3 D = ray.direction;
10     float radius = sphere.w;
11
12     // analytic sphere solution
13     // float a = dot(D, D); useless since direction is normalized, a=1
14     float b = 2*dot(O, D);
15     float c = dot(O, O)-radius*radius;
16     float disc = b*b-4*c;
17
18     // If there are two different intersections , we ignore the case with only one intersection
19     if( disc > 0 ){
20         float sdisc = sqrtf(disc);
21         float tmin = (-b - sdisc)/2;
22         float tmax = (-b + sdisc)/2;
23
24         // tmin will be surely nearest potential point, so we report it
25         if( rtPotentialIntersection (tmin) ) {
26             // Attribute data between rtPotentialIntersection and rtReportIntersection
27             distance = (tmax - tmin);
28             // Call closest hit
29             rtReportIntersection(0);
30         }
31     } // if( disc >= 0 )
32 } // intersection program

```

3.4.6 Bounding Box Program

A *Bounding Box Program* describes the minimal three dimensional axis-aligned cube that contains the geometry. The idea is to find out a primitive's minimum and maximum value in each dimension and create two points with three maximums and three minimums respectively. A cube formed by these two vertices is therefore the geometry's bounding box. We take an example of a sphere primitive:

```

1  rtDeclareVariable( float4, sphere, , );
2
3  // Bounding box for sphere geometry
4  RT_PROGRAM void bounds (int, float result[6])
5  {
6      const float3 cen = make_float3( sphere );
7      const float3 rad = make_float3( sphere.w );
8
9      // Axis aligned bounding box
10     optix::Aabb* aabb = (optix::Aabb*)result;
11
12     if( rad.x > 0.0f && !isinf(rad.x) ) {
13         // first vertex with three minimums
14         aabb->m_min = cen - rad;
15         // second vertex with three maximums
16         aabb->m_max = cen + rad;
17     } else {
18         aabb->invalidate();
19     }
20 }
```

3.5 CURAND: RNG on GPUs

The CURAND library provides efficient pseudo-random and quasi-random number generators on GPUs. It's a RNG (Random Number Generator) API extremely optimized for GPU environment. According to the development group, the latest CURAND 6.0 performs up to 75x more powerful than Intel MKL [34] (Figure 3.3).

CURAND encapsulates several popular PRNGs. Regardless of their difference on statistical properties, the standard XORWOW has the best performance (Figure 3.4). Another

cuRAND: Up to 75x Faster vs. Intel MKL

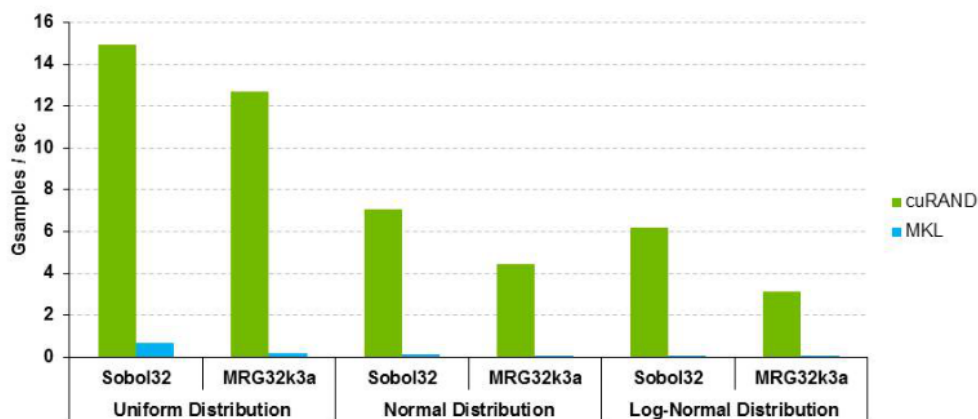


FIGURE 3.3: CURAND vs. MKL [34].

PRNG highly recommended by developers is MRG32K3A. This generator preserves good statistic quality as well as satisfied performance. The most famous Mersenne Twister MTGP32 is widely used for scientific computing at present. Even if it maintains an impressively long period, its relatively poor efficiency on GPUs should always be taken into considerations for performance concerns. These three generators will be evaluated for our MODERATO implementation in the following of this section.

cuRAND: High Performance RNGs

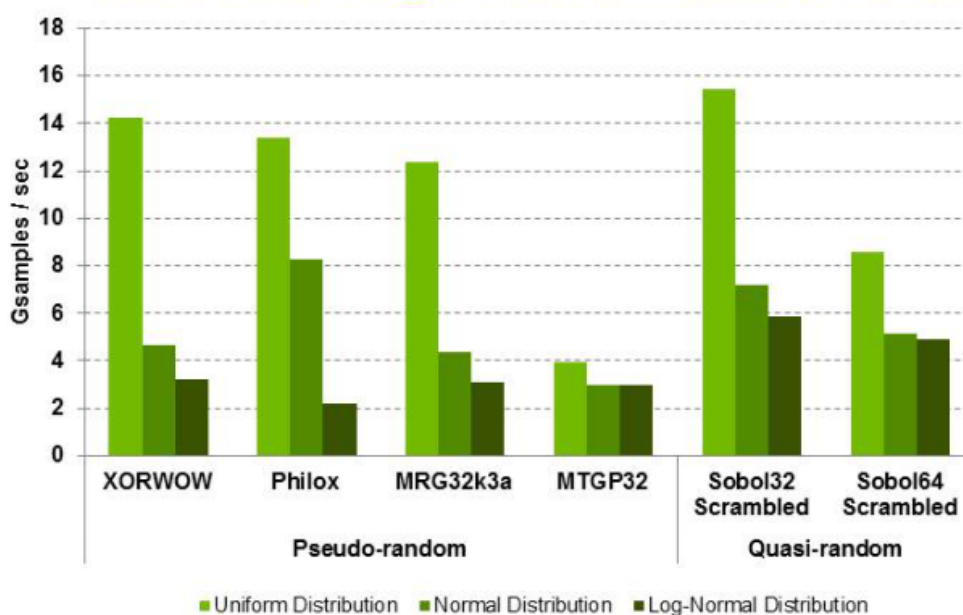


FIGURE 3.4: CURAND high performance RNGs [34].

3.5.1 Mersenne Twister: MTGP32

Mersenne Twister (MT) is one of the most widely used PRNG at the moment [35]. MTGP32 is its variant optimized for GPUs. Just like the standard MT19937, MTGP32 has a very long period of $2^{19937} - 1$, but it appears to be its only advantage.

The major drawback is that MTGP32 is fairly slow compared to other PRNGs: nearly three times more sluggish than other generators (Figure 3.4). Moreover, it requires a very large state (624 words), which makes it memory-expensive if you want a random number sequence for each thread where you have tens of thousands of threads. Furthermore, these states heavily stress the GPU cache as well. MT's statistical property bring it grand reputation, however, it doesn't manage to pass all tests of TestU01 (Empirical statistical testing of uniform random number generators by L'Ecuyer) [36].

As for an initial state with many zeros, MTGP32 will spend a long time on turning the state into output that passes randomness tests. Figure 3.5 shows a test presented by L'Ecuyer: he initializes the state with a single bit at 1. The mean of uniform random outputs is badly deviated until 700000 iterations. Any developers who try to use MT in their implementations should pay attention to this issue.

WELL19937 vs MT19937; moving average over 1000 iterations.

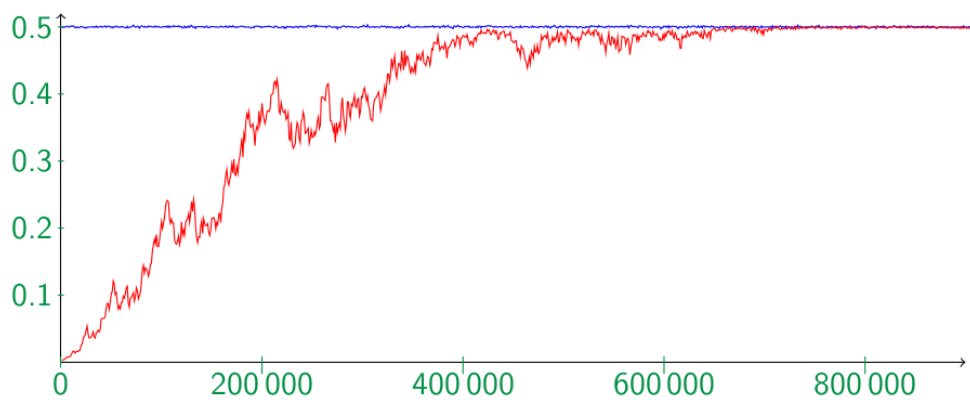


FIGURE 3.5: Initial state with a single bit at 1 [36].

Another point mentioned by CURAND tutorial is that MTGP32 is closely tied to the thread and block count and the most efficient use of MTGP32 is to generate a multiple of 16384 samples [37]. Unfortunately, OptiX has no thread control; the number of block is automatically distributed and optimized by the API. Hence MTGP32 might be neither

safe nor efficient in OptiX. In conclusion, MTGP32 is not an efficient choice for our OptiX implementation.

3.5.2 Combined Multiple Recursive RNG: MRG32K3A

MRG32K3A is a combined multiple-recursive generator with a period of 2^{191} [38]. It performs slightly slower than the fastest XORWOW (Figure 3.4). However, it's one of the few generators recommended by L'Ecuyer since it passes all TestU01 randomness tests [36].

An initial OptiX implementation was tested with this RNG. The result was not conclusive: output number sequences from different threads would produce repeatedly a certain value. This may signalize that different threads share a same state and MRG32K3A is sensitive to thread-safety. Debug of OptiX can be rather difficult for any problems related to thread control and the bug hasn't been solved yet.

3.5.3 XOR-Shift RNG: XORWOW

XORWOW is the default generator type in CURAND with a period of $2^{192} - 2^{32}$. It rapidly produces random numbers with efficient xor-shift operations, which contributes to its competitive performance against other RNGs. L'Ecuyer figured out that XORWOW is rejected by TestU01 [38]. Saito and Matsumoto conducted a deeper investigation to explain this rejection [39] and found that XORWOW has a significant deviation on 6-dimensional distribution of the most significant 5 bits. It means that one may find a particular regularity between a sixth random number and its previous five ones and therefore it's not strictly stochastic. At last, they concluded that this RNG is not ideal for serious Monte Carlo simulation.

The current OptiX implementation employs this RNG. Test results demonstrate that XORWOW preserve satisfied statistical characteristics since the difference between OptiX and the original code is negligible ($\approx \pm 0.2\%$, see Section 5.5). Nevertheless, explorations of MRG32K3A would be taken into account in future work for serious simulations.

3.6 Interoperability with CUDA

It is often advisable to combine CUDA kernels with OptiX programs so that we can take full advantage of GPU performance. In our implementation, for instance, a CUDA kernel launched before OptiX content is responsible for initializing and seeding the CURAND RNG. Furthermore, other applications may request post-treatment of OptiX's output with CUDA. Both of these two cases require a CUDA-OptiX communication.

There are two possible solutions for this interoperability. One is to ask OptiX to allocate device memory and get a pointer to data. Another is to create OptiX buffers in CUDA kernels. The first method can be used for either feeding OptiX with data or postprocessing outputs. The second one, on the contrary, can be only used for pre-calculations since OptiX will neither allocate memory nor upload data for these buffers [5]. But this solution is similar to transferring data with *Variable* and *Buffer* that are used before and therefore easier to implement. In addition, its one-way communication capability has already satisfied initialization need. So we choose the second solution in MODERATO.

The use of CUDA buffers is nearly the same as normal host-device buffers:

```

1  optix::Variable states_buffer = context["states.buffer"];
2
3  // Create an OptiX buffer for CUDA kernel
4  optix::Buffer states = context->createBufferForCUDA( RT_BUFFER_INPUT,
               RT_FORMAT_USER, N );
5  states->setElementSize( sizeof(curandState) );
6
7  // Allocate space for prng states on device
8  cudaMalloc((void **)&devStates, N * sizeof(curandState));
9
10 // State setup is complete
11 // Initialize one state per thread
12 setup_kernel<<<h, w>>>>(devStates, w);
13
14 // Providing a device pointer for the CUDA-OptiX buffer
15 states->setDevicePointer( 0, reinterpret_cast<CUdeviceptr>(devStates) );
16
17 // Attach buffer with the corresponding variable in OptiX Program
18 states_buffer->set(states);

```

Consult *OptiX Programming Guide* for more details on getting CUDA device pointers from OptiX.

3.7 Performance Guidelines

In terms of OptiX performance, several precautions should be taken in order to improve ray tracing process. Even small changes in the code can dramatically alter performance. Here is a list of some essential points based on OptiX manual, developers may complete it according to their own investigations [5]:

- Use floats instead of doubles since float are already sufficient for MODERATO. This also extends to the use of literals and math functions. For example, use *0.5f* instead of *0.5* and *sinf* instead of *sin* to prevent automatic type promotion.
- Compile PTX with *-use-fast-math* option, it can reduce code size and increase the performance for most OptiX programs with a small decrease on accuracy. Replace standard functions with intrinsic functions when small accuracy errors are acceptable.
- Each new *Program* object can introduce execution divergence. Try to reuse the same program with different variable values. For example, interaction rays and normal rays in MODERATO share the same sphere *Intersection Program*.
- Minimize payload size and try to use a structure of arrays instead of an array of structures data. Any uninitialized variables can increase register pressure and degrade performance.
- Disable useless *Programs* wherever possible: *Miss*, *Exception* and *Any Hit Program* are disabled in the MODERATO implementation.
- Multiple access to global memory can be replaced by declaring local variables with much lower communication latency.
- No recursion on GPUs: tremendous complexity, try to replace it by iterations.

3.8 Programming Caveats

OptiX hasn't been fully developed at the moment. Hence certain errors or bugs might appear during implementation. Even if this is a CUDA based API, it's not completely compatible with all CUDA features. Following caveats should be always kept in mind to avoid unnecessary mistakes [5]:

- Setting a large kernel size will consume GPU device memory. Try to minimize the stack as much as possible. Start with a small stack and with the use of an exception program that indicates you have exceeded your memory, increase the stack size until the stack is sufficiently large. This is why a 1000×1000 launch was available on Quadro 5000 but not on Quadro FX 1800, since the latter has smaller memory space.
- `__shared__` memory within OptiX *Programs* is forbidden.
- NEVER call CUDA `syncthreads()` in OptiX Programs.
- CUDA `threadIdx` can map to multiple launch indices in OptiX. Use of OptiX *rtLaunchIndex* semantic is mandatory.
- Currently, OptiX is not guaranteed to be thread-safe. OptiX should therefore be used only from a single host thread.
- Dynamic allocations are allowed in CUDA with 2.0 cards or greater [40], but not in OptiX Programs. Attempts to use these functions will result in an illegal symbol error.

Chapter 4

Implementation

4.1 Modeling Process

Thanks to the flexible tracking utility employed by MODERATO, transplants of this simulation code to other ray tracing algorithms are feasible and relatively easy. Our OptiX implementation was realized without many changes of the original code.

Moreover, our implementation doesn't cover all functionalities of MODERATO. The work aims to familiarize OptiX mechanism and explore OptiX's ray tracing potential for scientific computing, but not to perform the entire MODERATO code on GPUs. In the source model, for example, one can only generate mono-energetic photon with sphere shape.

In order to make the work more comprehensible, porting process of a normal MODERATO test scene will be presented in this section.

4.1.1 Projection Source

The source model of test scenes is described with *Ray Generation Program*, which corresponds to `CADSource::generatePhoton()` in MODERATO. It's in this program that we create and cast photons by drawing random numbers to determine their characteristics. Note that the device side implementation is currently non-object oriented, some work has been done during the adaptation to remove the object layer. Fortunately, this part

of codes is not so structured as complex geometry objects, turning it to C-like program did not need too many efforts.

In spite of these "Copy-Paste" adaptations, it should be noted that some small changes have been made in order to adapt OptiX pipeline. Cross section ratios $S1$, $S2$, $S3$ were integrated by a single *float3* type. $\theta1$, $\theta2$ weren't encapsulated within payload in order to minimize payload size. Attributes like *direction*, $p0$ (previous location), $p1$ (current location) and L (current free-path) were removed since there are similar attributes already defined in the *optix::Ray* object. Attempts to make full use of OptiX objects are good for performance.

4.1.2 Inspected Object

Simulation of inspected object is much more complicated than the other two models. This is the essential part of the implementation: potential intersection points are no longer restricted on geometry borders due to interactions within materials. Furthermore, scattered photonss change direction and energy properties during the traversals. Different materials can greatly vary diffusion behavior, which leads to considerable thread divergence on GPUs. Besides, the hierarchical geometry objects make it difficult to rebuild with other programming languages or ray tracing mechanisms, especially when no oriented-object language is available. Building UML diagrams seems to be very helpful for further development.

As for simple geometry types like sphere, cube or plane, there is no need to rewrite the original *CADObject* into OptiX. On the contrary, complex geometries corresponding to *CADComplexBREP* should preserve their combination of *CADSurface* and *CADContour* and make some necessary adaptations. Note that *cudaMalloc()* and *cudaFree()* are temporarily not supported in OptiX *Program*, allocating dynamically objects inside the entire geometry hierarchy is no longer possible, which leads to considerable work for rewriting. In this situation, OptiX *Intersection Program* corresponds to the same hierarchy level as *CADBREPObject::getIntersections()*:

```

1 // Intersection Program for CADRod geometry
2 RT_PROGRAM void intersect_rod(int primIdx)
3 {
4     PCADComplexBREP section[3];

```

```

5
6  // three patches restrict CADRod borders
7  section [0] = CADCylinderPatch;
8  section [1] = CADDiscPatch;
9  section [2] = CADDiscPatch;
10
11  tmin = ... // Look over all patched and find out the minimum t value
12
13  ... // Report tmin value and call Hit Programs

```

Management of complex geometries has not been completed in OptiX implementation since rewriting dynamic allocation to static takes long. Meanwhile, there is no limit to use *cudaMalloc()* and *cudaFree()* on the host side. In other words, we could allocate dynamically on CPUs instead of on GPUs. Therefore, such allocation problem is not so serious as we imagined before.

Another porting difficulty is associated with the tracking utility. Although the tracer used by MODERATO is open and flexible, it always exists small differences with OptiX. For instance, OptiX developers only need t value for tracking things. Updates of position coordinates or intersection numbers are not included in their concern. But in MODERATO, these ray tracing operations are usually combined with some simulation-specialized functions. Owing that OptiX doesn't provide interface to modify its basic ray tracing mechanism, it's not easy to distinguish which operations are already encapsulated within OptiX mechanism and which calculations need to be picked out and reprogram, not to mention that there is still oriented-object adaptation to treat at the same time. A lot of rewriting work on *Moderato::TracePhoton()* and *Photon::BilanPhoton()* has been done during this procedure. In addition, ensuring that the update of various photon characteristics is in right order. Confusing updates would cause simulation errors and such mistakes are really hard to debug.

4.1.3 Detector

Detector model in OptiX is highly simplified because fully featured MODERATO on GPUs is not the aim of this internship. The atomic addition is employed to count photons with different diffusion numbers. At last, all simulation results will be sent back to the host by *Buffers*. Generating simulation images is supported as well.

4.2 Problems and Solutions

4.2.1 Serialized Access to Variables

In parallel computing, some serializations are inevitable. In MODERATO, for example, the detector needs to count the number of photons arrived on each pixel and therefore generates simulation images. On parallelism, it's possible that several photons arrive simultaneously at the same pixel. Such thread mutual exclusion will surely lead to calculation errors. To solve this problem, CUDA provides developers with built-in atomic operation. This operation ensures a serialized access to global variable. In other words, no other thread can access this address until the operation is complete.

In our implementation, we use `atomicAdd()` to count different interactions occurred during simulation process. Since these numbers may up to trillions, precision of `int` or `float` can not meet the request. Thus, we employ:

```
1 unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);
```

Note that the 64-bit integer version of `atomicAdd()` is only available for devices of Compute Capability 1.2 and higher. The simple precision floating-point requires 2.0 or higher and the double precision floating-point is temporarily unavailable [40].

4.2.2 Using CURAND within OptiX

CURAND provides highly optimized RNGs for general-purposed GPU computation with a satisfying long period ($> 2^{190}$). Thanks to the CUDA/OptiX interop, we could call CURAND functions inside OptiX *Programs*.

The essential idea is to have a CUDA kernel make an array of RNG states on using CURAND device API, and then OptiX creates a *CUDABuffer* around it. Device API allows random numbers to be generated and immediately consumed by kernels without requiring the random numbers to be written to and then read from global memory. With host API, however, we can only generate a fixed number of random variables before using them. `curand_uniform(&state)` is used to do an uniform random number selection in the device *Programs*:

```

1  // In state setup kernel (CUDA kernel) under "random.cu" file compile to .o
2  __global__ void setup_kernel(curandState *state, unsigned int w)
3  {
4      int id = threadIdx.x + blockIdx.x * w;
5      // Each thread gets same seed, a different sequence number, no offset
6      curand_init(1234, id, 0, &state[id]);
7  }
8
9  // In OptiX Program under "ray.cu" file compile to .ptx
10 RT_PROGRAM void ray_gen(){
11     for(int i = 0; i < iteration; i++){
12         unsigned int idx = launch_index.x*launch_dim.y+launch_index.y;
13         /* Copy state to local memory for efficiency */
14         curandState localState = states_buffer[idx];
15
16         Phi = curand_uniform(&localState)*2*3.14159f;
17         ...
18
19         // Copy state back to global memory
20         states_buffer[idx] = localState;
21         ...
22     }
23 }

```

See Section 3.6 for host side code examples. Note that CURAND doesn't support the CUDA driver API launch, only CUDA runtime for the setup kernel is accepted [37].

4.2.3 Model of Photon Interactions

As an API initially targeted for rendering graphics, ray refraction and reflection can be certainly handled by OptiX. However, these changing direction movements all take place on geometry borders instead of inside a geometry primitive. One of our initial questions with the use of OptiX was then concentrated on its capability to model photon interactions inside an inspected object.

After learning the intersection mechanism of OptiX, we managed to solve this modeling problem. Figure 4.1 points out the five essential procedures of this process:

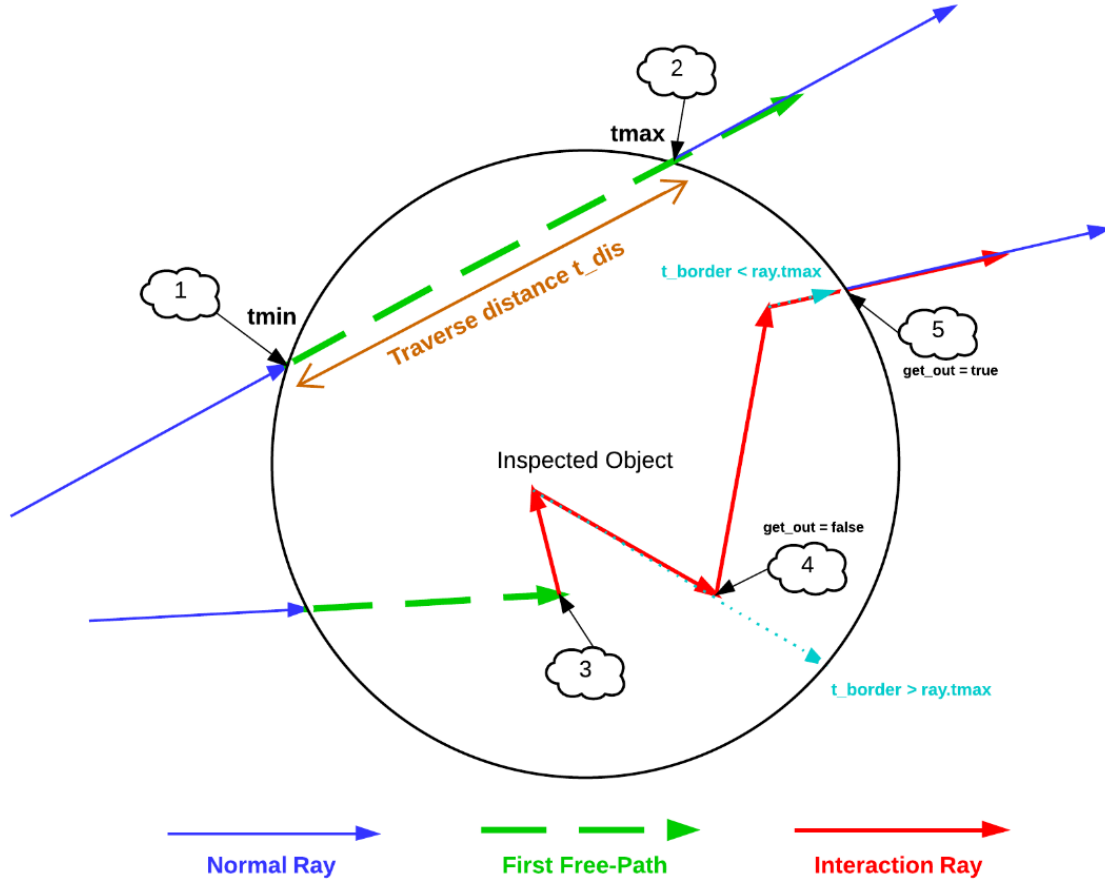


FIGURE 4.1: OptiX photon interaction model.

1. In *Intersection Program*, a normal ray obtains its two t values for intersecting a sphere object: t_{min} and t_{max} . Then *Intersection Program* reports t_{min} as the nearest intersection point and transfers the traversal distance t_{dis} to *Normal Ray Closest Hit Program* by *Attribute* semantic.
2. In *Normal Ray Closest Hit Program*, CURAND is employed to generate the photon's first free-path $pd.L$. If this path distance is greater than t_{dis} , it symbolizes that the photon has no interaction in the object. In this situation, a new normal ray will be emitted at the initial exit point.

```

1 // Cast a new normal ray at exit with the same direction
2 Ray interaction_ray(exit, ray.direction, normal_type, scene_epsilon, RT_DEFAULT_MAX);

```

3. Otherwise ($t_{dis} < pd.L$), we update particle properties and produce a new free-path to replace the former one in *Normal Ray Closest Hit Program*. Then, a new ray of type *interaction_type* will be launched with an up-to-date direction at the

end of the first free-path. The trick in this process is to define the ray property *tmax* with the new free-path.

```
1 // Cast new ray with interaction_type and tmax=pd.L
2 Ray interaction_ray( hit_point, new_direction, interaction_type, scene_epsilon, pd.L );
```

4. An interaction ray calls *Intersection Program* to determine potential crossing points as well. It will at first calculate a *t_border*, which represents a ray's minimum *t* value to reach the sphere border. Afterwards, it will compare this *t_border* with the ray property *tmax*. If *tmax* is no bigger than *t_border*, which means the photon will stay in the sphere and continue to have other interactions, *Intersection Program* will call *Interaction Ray Closest Hit Program* at the end of *tmax* and will signalize a flag *get_out=false* with *Attribute*. The flag informs *Interaction Ray Closest Hit Program* to launch a new ray that won't exit the object and the ray type is always *interaction_type*.
5. If *tmax* exceeds *t_border*, the photon will arrive at the sphere border before it runs through the current free-path. At this time, *Intersection Program* will invoke *Interaction Ray Closest Hit Program* with the crossing point corresponding to *t_border*. The flag *get_out=true* informs *Interaction Ray Closest Hit Program* that the next ray will escape from the sphere and there won't be interactions any more. So it needs to launch the following ray in *normal_type* and right on the border.

4.2.4 Object-Oriented Programming on GPUs

As it has been indicated by *CUDA Programming Guide* [40], CUDA supports full C++ for the host code but not for the device code, the same for OptiX. Consult <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#restrictions> for more restriction details. Generally speaking, most of these restrictions are related to the CUDA streaming-processing architecture.

Actually, the main difficulty of object-oriented programming is that we should remove all dynamic allocations since it's not available on GPUs. Furthermore, this part of the code is relatively more hierarchical than other objects in MODERATO. All geometry objects locating in a file with 20000 lines of programs pressure the work as well.

In a word, creating user-defined objects in OptiX is feasible. Object-oriented programming allows us to adapt more easily complex geometry objects to OptiX.

Chapter 5

Tests and Results

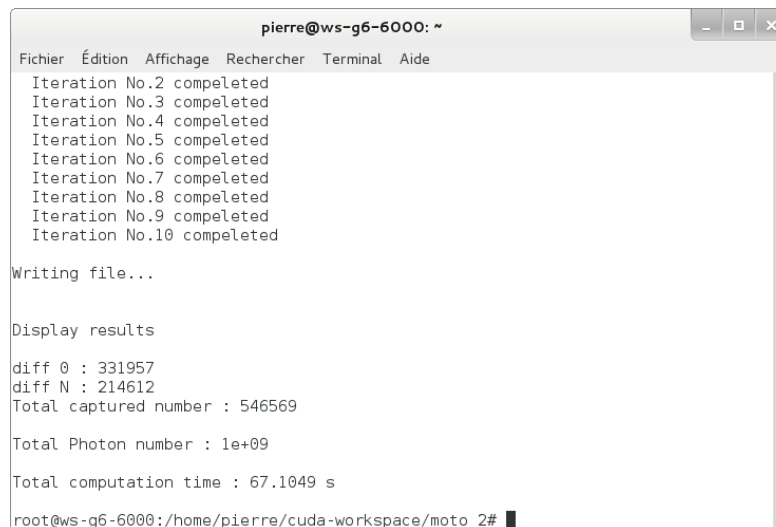
This chapter presents the tests that we have taken to evaluate OptiX's characteristics from different points of view. Test results will be analysed in order to conclude OptiX's potential for Monte Carlo simulations.

Note that all tests and implementations presented in this report is based on a working environment with:

- **CPU:** Xeon E5504 (4M Cache, 2.00 GHz)
- **GPU:** Quadro 5000 (Compute Capability: 2.0, 352 CUDA cores, 120 GB/s)
- **OS:** 64-bit Debian Wheezy
- **APIs:** OptiX 3.5 and CUDA 5.5.

5.1 Effectiveness of Fast-Math

Fast-math functions are built in CUDA architecture by using the "special function unit" in each multiprocessor, taking one instruction, whereas the CPU implementations can take dozens of instructions. The *CUDA Programming Guide* does not list the speed difference, but according to the slides from Hwu's presentation [41], fast-math operations take 16 to 32 cycles per warp while standard operations are done in hundreds of cycles. Of course, here we are talking about the expensive functions (exp, sin, sqrt etc.), not simple multiplications or additions - those take about 20 cycles per warp.



```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Iteration No.2 compeleted
Iteration No.3 compeleted
Iteration No.4 compeleted
Iteration No.5 compeleted
Iteration No.6 compeleted
Iteration No.7 compeleted
Iteration No.8 compeleted
Iteration No.9 compeleted
Iteration No.10 compeleted

Writing file...

Display results

diff 0 : 331957
diff N : 214612
Total captured number : 546569

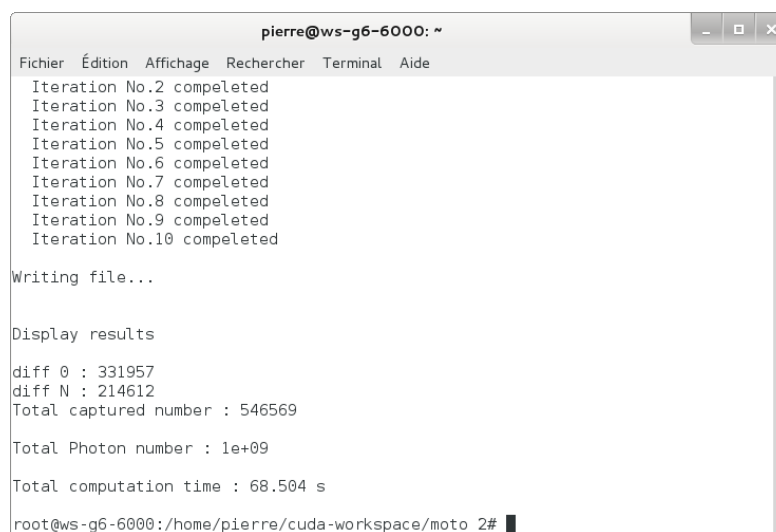
Total Photon number : 1e+09

Total computation time : 67.1049 s

root@ws-g6-6000:/home/pierre/cuda-workspace/moto_2#

```

(a) Fast-math case



```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Iteration No.2 compeleted
Iteration No.3 compeleted
Iteration No.4 compeleted
Iteration No.5 compeleted
Iteration No.6 compeleted
Iteration No.7 compeleted
Iteration No.8 compeleted
Iteration No.9 compeleted
Iteration No.10 compeleted

Writing file...

Display results

diff 0 : 331957
diff N : 214612
Total captured number : 546569

Total Photon number : 1e+09

Total computation time : 68.504 s

root@ws-g6-6000:/home/pierre/cuda-workspace/moto_2#

```

(b) Normal case

FIGURE 5.1: Comparison of fast-math functions and standard functions

Fast-math functions get this speed improvement at the price of sacrificing their accuracy privilege. Note that few functions in CUDA follow the IEEE 754 specification [40] and they only produce valid results for a given range of input values. The programming guide lists their maximum ULP (Unit in the Last Place) error information in [Appendix D](#).

Developers can use the `-use-fast-math` flag to force usage of intrinsics for the fast-math functions without changing source codes, so this would quickly give an idea of potential performance effect on the code.

Figure 5.1 shows test results on this effect. We find that fast-math has a slight speedup

compared to normal operations. It indicates that among the entire calculation in OptiX kernel, arithmetic functions occupy a little part and therefore this acceleration is too hard to distinguish. Furthermore, since MODERATO doesn't require a high-standard accuracy, the amount of photons arrived on the detector is exactly the same in these two tests. In conclusion, fast-math functions doesn't effect our implementation's outputs. They can satisfy our arithmetic precision requirements and accelerate calculation performance. Note that the current implementation doesn't include many computationally intensive operations, the acceleration of fast-math will be more evident for a more complicated test. Likewise, fast-math might be not accurate enough for other serious simulations.

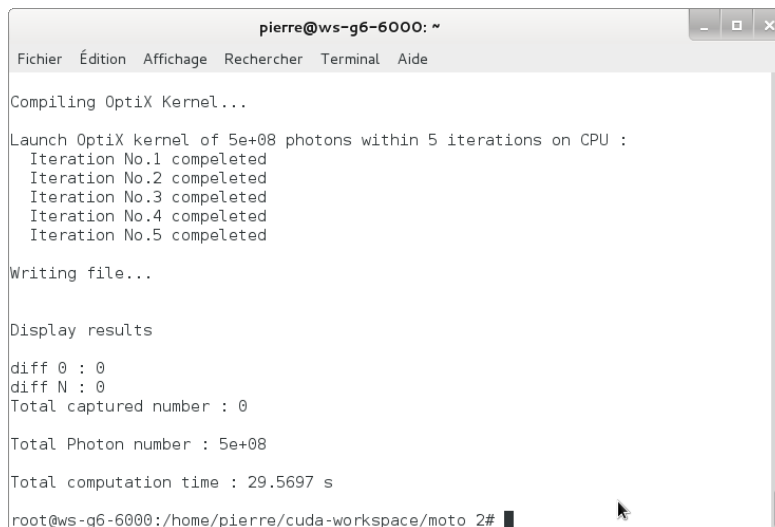
5.2 Degradation with Atomic Operations

Atomic operations serialize the execution of parallel programs and therefore pull down computing performance. Some algorithms can be optimized to avoid this serialization, others may not. We did a test to know how serious the atomic will effect our implementation's performance (Figure 5.2).

Fifteen counter variables require atomic addition. For the first test, we comment all these additions so the counter works no more. Then a second test deactivates the comment and gets a delay for about 3.1 seconds ($\approx 10\%$). This degradation is considerable because we haven't employed too many atomics. For further implementation, changing algorithms to avoid this serialization could be a potential solution. However, since *Programs* are fixed in OptiX ray tracing mechanism and take no arguments as input, the reduction with thread's local variable could be intractable.

5.3 Influence of Kernel Size

The kernel size influences the synchronic computing scale on parallel. Generally speaking, cards with higher Compute Capability could afford larger kernels. But it doesn't mean that a larger kernel is more efficient than a smaller one in any case. With the increment of kernel dimension, latency of block/thread or local/global communication will be extended. For example, two calculations are launched respectively for tracing



```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Recherche  Terminal  Aide

Compiling OptiX Kernel...

Launch OptiX kernel of 5e+08 photons within 5 iterations on CPU :
  Iteration No.1 completed
  Iteration No.2 completed
  Iteration No.3 completed
  Iteration No.4 completed
  Iteration No.5 completed

Writing file...

Display results

diff 0 : 0
diff N : 0
Total captured number : 0

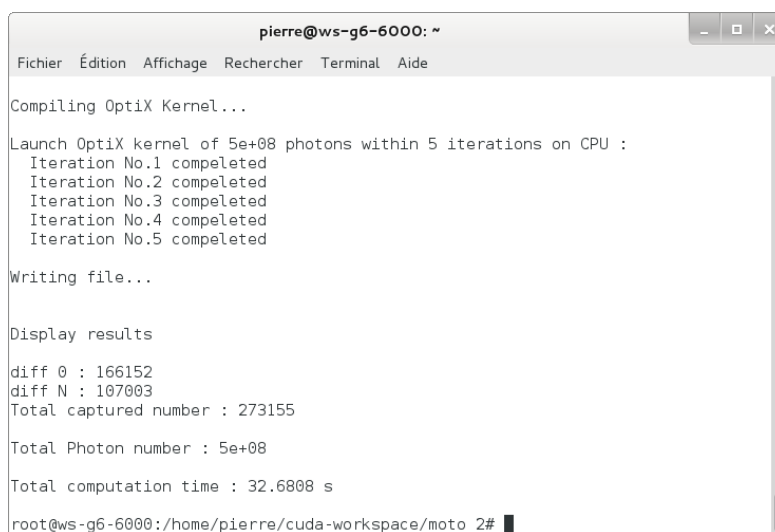
Total Photon number : 5e+08

Total computation time : 29.5697 s

root@ws-g6-6000:/home/pierre/cuda-workspace/moto_2#

```

(a) No counters



```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Recherche  Terminal  Aide

Compiling OptiX Kernel...

Launch OptiX kernel of 5e+08 photons within 5 iterations on CPU :
  Iteration No.1 completed
  Iteration No.2 completed
  Iteration No.3 completed
  Iteration No.4 completed
  Iteration No.5 completed

Writing file...

Display results

diff 0 : 166152
diff N : 107003
Total captured number : 273155

Total Photon number : 5e+08

Total computation time : 32.6808 s

root@ws-g6-6000:/home/pierre/cuda-workspace/moto_2#

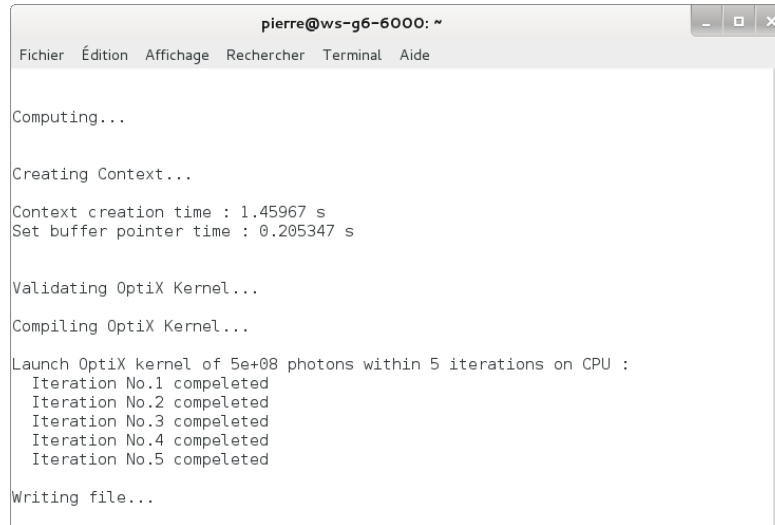
```

(b) Atomic counters

FIGURE 5.2: Effectiveness of atomic operations

5×10^8 photons with the same configuration (Figure 5.3): a first one with a 500×500 kernel and a second one with a 1000×1000 kernel. Though the difference between kernel executions is negligible, we find that the setup of the OptiX device buffer for the 1000×1000 kernel spends almost 44 times longer (8.84 s vs. 0.20 s) than the 500×500 one. There is no formal explanation for the huge variance, but we think this relates to device architecture.

In another test with 1×10^{10} photons, however, we find that even the initialization time is huge. The larger kernel performs more efficiently than the smaller one (Figure 5.4). This result indicates that a bigger kernel size is more suitable for larger datasets of



```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide

Computing...

Creating Context...

Context creation time : 1.45967 s
Set buffer pointer time : 0.205347 s

Validating OptiX Kernel...

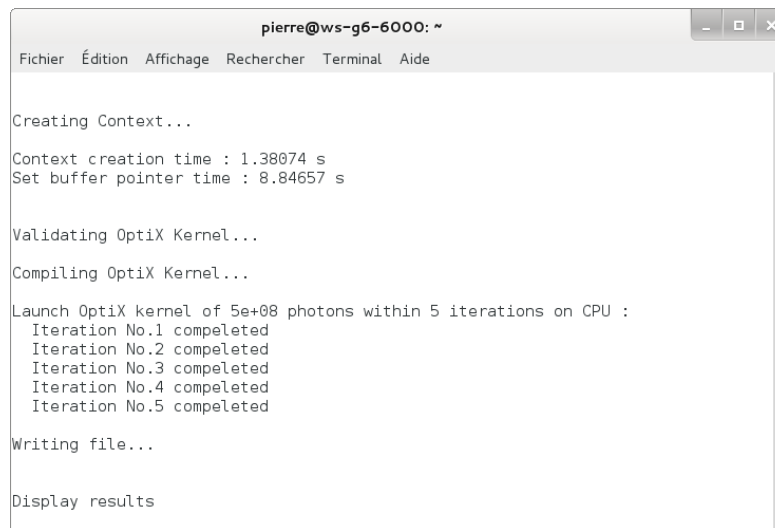
Compiling OptiX Kernel...

Launch OptiX kernel of 5e+08 photons within 5 iterations on CPU :
  Iteration No.1 completed
  Iteration No.2 completed
  Iteration No.3 completed
  Iteration No.4 completed
  Iteration No.5 completed

Writing file...

```

(a) 500×500 kernel



```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide

Creating Context...

Context creation time : 1.38074 s
Set buffer pointer time : 8.84657 s

Validating OptiX Kernel...

Compiling OptiX Kernel...

Launch OptiX kernel of 5e+08 photons within 5 iterations on CPU :
  Iteration No.1 completed
  Iteration No.2 completed
  Iteration No.3 completed
  Iteration No.4 completed
  Iteration No.5 completed

Writing file...

Display results

```

(b) 1000×1000 kernel

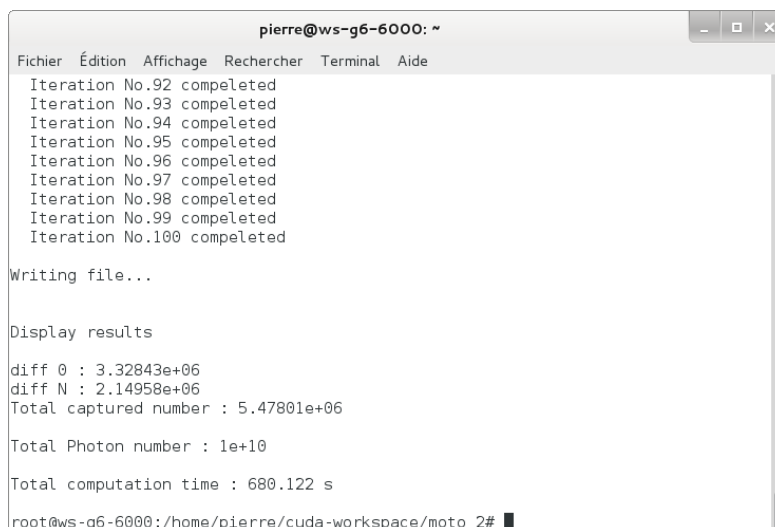
FIGURE 5.3: Initialization difference between 2 kernels.

particle histories. Profiting a maximum kernel dimension and keeping all threads busy will contribute to a better GPU performance.

5.4 Comparison of Acceleration Structures

OptiX encapsulates various acceleration algorithms for ray tracing process. This design liberates users from days of work and knowledges of ray tracing basics.

Different acceleration structures are evaluated with the same geometry configuration, but it's rather hard to tell the difference. Actually, there are only one sphere and one



```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Iteration No.92 compeleted
Iteration No.93 compeleted
Iteration No.94 compeleted
Iteration No.95 compeleted
Iteration No.96 compeleted
Iteration No.97 compeleted
Iteration No.98 compeleted
Iteration No.99 compeleted
Iteration No.100 compeleted

Writing file...

Display results

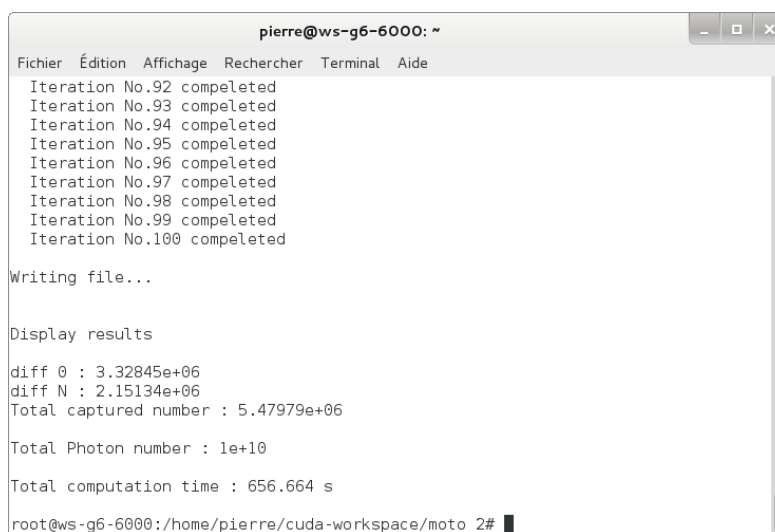
diff 0 : 3.32843e+06
diff N : 2.14958e+06
Total captured number : 5.47801e+06

Total Photon number : 1e+10

Total computation time : 680.122 s

root@ws-g6-6000:/home/pierre/cuda-workspace/moto_2#

```

(a) 500×500 kernel for 1×10^{10} photons


```

pierre@ws-g6-6000: ~
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Iteration No.92 compeleted
Iteration No.93 compeleted
Iteration No.94 compeleted
Iteration No.95 compeleted
Iteration No.96 compeleted
Iteration No.97 compeleted
Iteration No.98 compeleted
Iteration No.99 compeleted
Iteration No.100 compeleted

Writing file...

Display results

diff 0 : 3.32845e+06
diff N : 2.15134e+06
Total captured number : 5.47979e+06

Total Photon number : 1e+10

Total computation time : 656.664 s

root@ws-g6-6000:/home/pierre/cuda-workspace/moto_2#

```

(b) 1000×1000 kernel for 1×10^{10} photons

FIGURE 5.4: Execution time between different kernels.

parallelogram in the test scene and different structures don't vary much in such simple scenes. To be more precise, *NoAccel* (no acceleration) is even better than other structures in our case. This consequence is confirmed by *OptiX Programming Guide* as well: it suggests that for static geometry groups, use *Sbvh* or *TriangleKdTree*. For dynamic geometry groups or large number of elements, experiment with *Lbvh*. For Group nodes, *Bvh* is the best choice in many cases, but if there are few enough children *NoAccel* can be useful [5].

5.5 OptiX Performance

OptiX ray tracing performance is evaluated by comparing it with a CPU cluster. Test configurations are:

- *Photon*: number = 1×10^{10} , mono-energy = 400 keV, direction = (0,0,-1);
- *Source*: sphere, position = (0,0,0), radius = 1, open angle = 4° ;
- *Object*: sphere, iron, position = (0,0,-55), radius = 50;
- *Detector*: parallelogram, position = (0,0,-110), side length: 10×10 , image resolution: 100×100 pixels.

The cluster ATHOS uses to link the computing nodes a FDR14 InfiniBand Cable with a bandwidth of 56 Gb/s. It consists of 776 normal nodes and 27 "Bigmem" (up to 2T memory space) nodes. Each normal node contains two Intel Xeon E5-2697 V2 processors (2.7GHz, 12 cores, 30M cache) and a shared memory of 64 GB.

Firstly, we tested OptiX accuracy by comparing it with the original MODERATO simulation result. One hundred million photons were launched with previous configurations.

•Résultat MODERATO

```
100000000 generated 55214 on film 0 split
diff 0 : 33675
diff 1 : 10386
diff 2 : 5093
diff 3 : 2744
diff 4 : 1651
diff 5 : 878
diff 6 : 415
diff 7 : 201
diff 8 : 95
diff 9 : 45
diff 10+: 31
33675 direct 21539 diff
Compton 299989300 Rayleigh 24358058 PE 76049818 PP 0
```

•Résultat OptiX

```
1e+08 generated 54903 on film
diff 0 : 33299
diff 1 : 10313
diff 2 : 5025
diff 3 : 2877
diff 4 : 1688
diff 5 : 887
diff 6 : 441
diff 7 : 211
diff 8 : 94
diff 9 : 40
diff 10+: 28
33299 direct 21604 diff
Compton 300029327 Rayleigh 24366377 PE 76043856 PP 0
```

FIGURE 5.5: Simulation results of OptiX and MODERATO.

From the above figure, we observe that the difference between these two simulations is reasonable. There is only a small deviation (about $\approx 0.2\%$) on the final total diffusion number. Moreover, if we reseed the RNG, the result will vary with the same type of deviation. This corresponds well to the Monte Carlo's intrinsic variance since it's based

on stochastic processes. This test proves that our implementation works correctly and OptiX can provide enough accuracy for the MODERATO simulation.

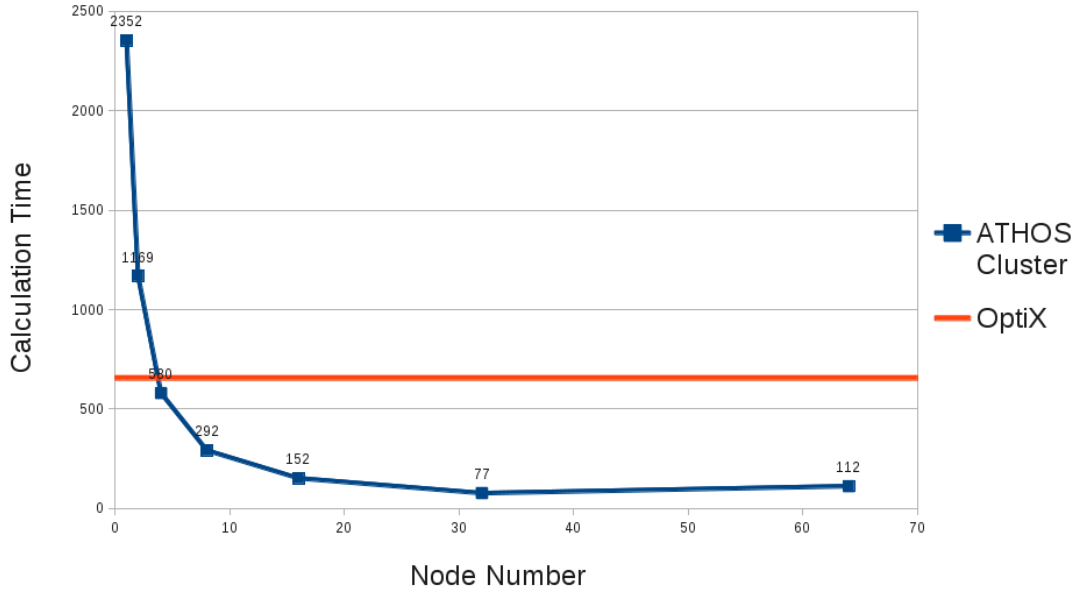


FIGURE 5.6: Tracing performance with different node amounts on ATHOS cluster.

In the second test, calculation time is figured on changing the number of execution nodes on ATHOS. According to Figure 5.6, we observe that calculation time reduces proportionally with an increment of node numbers. A double nodes brings almost a two-time speedup, but the test with 64 nodes is however longer than that of 32. This decreasing speed-up may come from the increasing data exchange between different nodes, frequent cable latency neutralizes the advantage of large-scale parallelization.

Comparing these results with the OptiX implementation (Figure 5.4(b)), we remark that our OptiX program spend approximately the same time as a 4-node CPU cluster. Note that the GPU we used is just a Quadro (initial for rendering graphics) 2.0 card, such result is rather remarkable and encouraging. Without changing the current code, its performance would be still improved by using a more powerful Tesla card (capability 3.0 or more).

Moreover, eight Xeon E5-2697 processors within the 4-node cluster cost about 20000 dollars. The unit price of Quadro 5000 is only 2499 dollars. In terms of power consumption, a Quadro 5000 consumes ($\approx 150\text{ W}$) no much more than a single Xeon processor ($\approx 130\text{ W}$). Those contrasts strongly confirm GPU's giant advantage in terms of performance/price and performance/watt ratios.

5.6 Technical Assessment

OptiX's ray tracing potential for general Monte Carlo simulations will be concluded in this section. Advantage and drawbacks will be listed respectively.

5.6.1 Advantage of OptiX

- OptiX provides an open ray tracing framework, developers would find it easy to integrate to their own programs. It's also an user-friendly API because programming with OptiX doesn't need too much GPU programming skills.
- It's highly optimized for CUDA architecture so as to get full use of the material, traditional programming difficulties like keeping all threads busy or use of shared memory are automatically managed.
- Geometries are totally user-defined, which allows OptiX implementation to preserve the essential analytical description of the original code. Complex geometry with patches is feasible since OptiX supports object-oriented programming on the device.
- Built-in simplified mathematical functions reduce calculation time of expensive operations and provide sufficient accuracy for MODERATO.
- Good cost-performance ratio economizes the cost and save energy. More calculations can be operated by cheaper device within shorter time.

5.6.2 Difficulties and Inconveniences

- OptiX manuals are sometimes difficult to understand, users should consult SDK code examples to understand some specific explanations. Few details about OptiX C++ wrapper are included in the programming guide. Several object methods are not even mentioned in the API reference.
- Available device cards are confined to NVIDIA's products. Meanwhile, OptiX ray tracing pipeline is not available on the host (no execution on CPU). Thus sustainability of the code is not guaranteed, who knows if NVIDIA still exists after twenty years.

- Since the device-side ray tracing pipeline is not object-oriented, several MODERATO models need to be restructured for this mechanism.
- Since OptiX 3.5, the API is no longer open-source. Commercial uses are required to pay additional fees, which result in extra cost for selling MODERATO on the market.
- Debug of device codes is rather difficult since *printf* is not practical in such large-scale parallel computing.

5.6.3 Following Work

Future work of our OptiX implementation would take into account the following aspects:

- Replace the current XORWOW RNG with MRG32K3A in order to have better statistical characteristics and be more accurate for industrial simulations.
- Continue rewriting complex geometry objects on deactivating dynamic allocations. Only the top node of geometry hierarchy can be initialized dynamically on the host. This procedure may take a developer a few weeks.
- Implement pair production, electron models in OptiX by using new ray types and creating their own *Materials* bounded to the geometry object. In theory, these modeling processes have no difference with other interaction models and therefore are easy to realize.
- Note that material description was highly simplified in the current OptiX implementation: full physic mode, interpolation operations, cross-sections etc, were not full-featured.
- Complete the detector model with OptiX. This part of modeling was not managed during my practice period, so it's hard to estimate the corresponding workload.

Chapter 6

Conclusion

The past six months of my internship have been very instructive for me. It's the first time that I participated in an industrial research project. This opportunity allows me to learn and develop myself in many areas. I gained a lot of experience, especially in scientific programming and parallel computing. A lot of activities are familiar with what I have learned at school. They enhanced my competence in HPC and inspired my interests in industrial research work.

On the other hand, I also encountered problems that I had never come across before. It was good to find out what my weaknesses are. This helped me to define which skills and knowledges I have to improve in the coming study time.

One thing I appreciate a lot is the HPC coffee organized by group members. They invite researchers specialized in one particular area to share their opinions and ideas with other people. All participants exchange their study issues comfortably and freely. Even if I didn't understand all discussions, I did get a global picture for my unknown topics from these discussions. Besides, If I hadn't had the chance to participate the L'Ecuyer's presentation at CEA, I might never know that generating random numbers is a very interesting topic. Such experience makes me eager to learn more and explore more; it really broadened my horizons.

In addition, every day's coffee time offered me a great chance to get involved in a French life. Traveling, sports, technique or children, all daily life came into conversation. These enriched my sight in a "French" way and I found myself greatly improved on my french.

From this internship I also learned the importance of time management. Once I realized what I had to do, I organized my day and work so that I did not waste my hours. A Gantt diagram of my internship helps with the time management (see Appendix C).

This six-month internship passed too quickly. I am grateful for meeting so many wonderful staff members. I will try my best to apply all I learned here into my future careers and sincerely hope that I have the chance to come back.

Appendix A

Installation of OptiX

This appendix will introduce how to install and verify the correct operation of the OptiX Ray Tracing Engine. Since OptiX is a CUDA-based framework, this installation is composed of 2 steps: Installation of CUDA Toolkit and installation of OptiX SDK. Note that OptiX is not listed in the *Logithèque of Calibre 7* (a Debian based distribution developed at EDF); a root account during installation is required.

A.1 Pre-installation Actions

Some verifications must be taken before the entire process of installation:

A.1.1 Verify the System Has a CUDA-Capable GPU

First of all, as a GPU framework developed by Nvidia, OptiX requires a CUDA-supported device. To verify if your GPU is CUDA-capable, open your *Terminal* and enter:

```
$ lspci | grep -i nvidia
```

If the shown GPU can be found in the following list: <http://developer.nvidia.com/cuda-gpus>, it's CUDA-capable. Otherwise, you should look for another GPU.

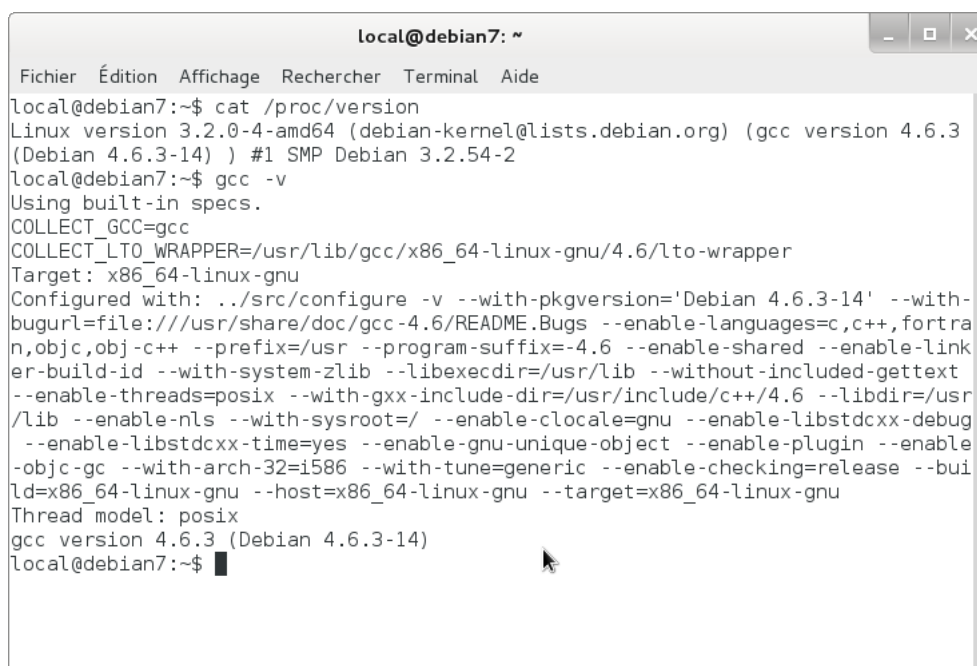
A.1.2 Verify GCC Compiler

The gcc compiler is generally installed as part of the Linux distribution. Either it was installed or should be installed manually by yourself, make sure the installed gcc is the same version which was used to build your Linux kernel (Figure A.1), or you will surely get errors during next step. Confirm your Linux kernel information:

```
$ cat /proc/version
```

Using the following command in order to verify the version of gcc installed on your system:

```
$ gcc -v
```



```
local@debian7: ~
Fichier Édition Affichage Rechercher Terminal Aide
local@debian7:~$ cat /proc/version
Linux version 3.2.0-4-amd64 (debian-kernel@lists.debian.org) (gcc version 4.6.3
(Debian 4.6.3-14) ) #1 SMP Debian 3.2.54-2
local@debian7:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.6/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Debian 4.6.3-14' --with-
bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortra
n,objc,obj-c++ --prefix=/usr --program-suffix=-4.6 --enable-shared --enable-link
er-build-id --with-system-zlib --libexecdir=/usr/lib --without-included-gettext
--enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.6 --libdir=/usr
/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug
--enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin --enable
-objc-gc --with-arch-32=i586 --with-tune=generic --enable-checking=release --bui
ld=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 4.6.3 (Debian 4.6.3-14)
local@debian7:~$
```

FIGURE A.1: Verify the gcc compiler

After all these actions done, the installation can be continued to the next step.

A.2 Installation of CUDA Toolkit

The NVIDIA CUDA Toolkit is available at <https://developer.nvidia.com/cuda-toolkit-archive>. Choose and download the distribution-independent package ("RUN" package)

for Ubuntu distribution, which corresponds to the *Calibre* system. Notice that any architecture of CUDA is running based on a supported driver, for which an update is highly recommended. Although a driver update module is integrated in the CUDA installation package, it turns out not to be convenient for users: updates crash without any warning information. Therefore, one practical choice is to install respectively these two parts. Here we take a Nvidia Quadro FX 1800 as an example and install CUDA 5.5.22.

A.2.1 Installation of the NVIDIA Driver

With the GPU information checked before, download the latest driver installation package: <http://www.nvidia.com/Download/index.aspx?>. In order to update a GPU driver, one should drop graphics interface on pressing *Ctrl+Alt+F1* and stop *X* server:

```
$ service gdm3 stop
```

To make the package executable, run the following:

```
$ sudo chmod x NVIDIA-Linux-x86_64-331.49.run
```

Install the NVIDIA driver on performing:

```
$ sudo ./NVIDIA-Linux-x86_64-331.49.run  
$ sudo reboot
```

Verify the driver has been successfully installed:

```
$ cat /proc/driver/nvidia/version
```

A.2.2 Installation of CUDA Toolkit

Quit GUI on pressing *Ctrl-Alt-F1* and begin installation by executing:

```
$ cat service gdm3 stop
$ sudo chmod a+x cuda_5.5.22_linux_64.run
$ sudo ./cuda_5.5.22_linux_64.run
```

Then the environment variables for our operating system need to be changed on editing the *.bashrc* file:

```
$ gedit .bashrc
```

Add these two lines on the top of the current file and save:

```
1 export PATH=/usr/local/cuda5.5/bin:$PATH
2 export LD_LIBRARY_PATH=/usr/local/cuda5.5/lib64:$LD_LIBRARY_PATH
```

A.2.3 Verify the Installation of CUDA

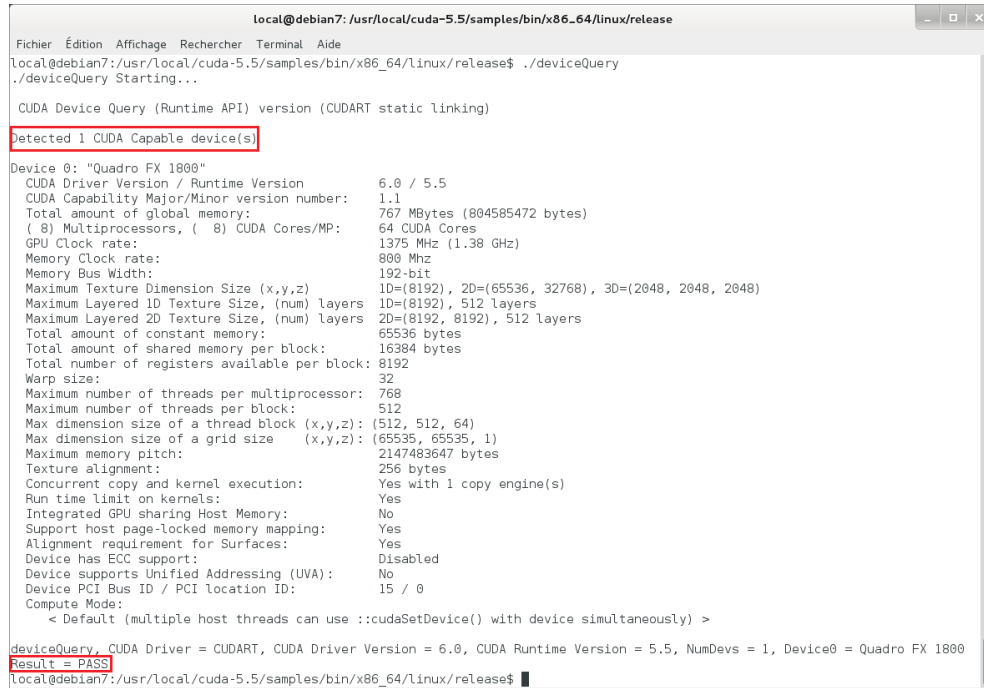
Before continuing, it is important to verify that the CUDA Toolkit can find and communicate correctly with the CUDA-capable hardware. To do this, you need to compile and run some sample programs. Changing to the sample directory and compiling the examples:

```
$ cd ~/NVIDIA_CUDA-5.5_Samples
$ make
```

After compilation, run *deviceQuery* and observe the result:

```
$ cd ~/NVIDIA_CUDA-5.5_Samples/bin/linux/release
$ ./deviceQuery
```

The output of this executable should look similar to the Figure A.2. Note that the important outcomes are that a device was found (the first highlighted line), that the device matches the one on your system, and that the test passed successfully (the second highlighted line).



```

local@debian7: /usr/local/cuda-5.5/samples/bin/x86_64/linux/release
Fichier  Edition  Affichage  Recherche  Terminal  Aide
local@debian7: /usr/local/cuda-5.5/samples/bin/x86_64/linux/release$ ./deviceQuery
./deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro FX 1800"
  CUDA Driver Version / Runtime Version      6.0 / 5.5
  CUDA Capability Major/Minor version number: 1.1
  Total amount of global memory:             767 MBytes (804585472 bytes)
  ( 8) Multiprocessors, ( 8) CUDA Cores/MP:  64 CUDA Cores
  GPU Clock rate:                           1375 MHz (1.38 GHz)
  Memory Clock rate:                        800 Mhz
  Memory Bus Width:                         192-bit
  Maximum Texture Dimension Size (x,y,z)     1D=(8192), 2D=(65536, 32768), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers 1D=(8192), 512 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(8192, 8192), 512 layers
  Total amount of constant memory:           65536 bytes
  Total amount of shared memory per block:    16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                 32
  Maximum number of threads per multiprocessor: 768
  Maximum number of threads per block:       512
  Max dimension size of a thread block (x,y,z): (512, 512, 64)
  Max dimension size of a grid size (x,y,z): (65535, 65535, 1)
  Maximum memory pitch:                      2147483647 bytes
  Texture alignment:                         256 bytes
  Concurrent copy and kernel execution:      Yes with 1 copy engine(s)
  Run time limit on kernels:                  Yes
  Integrated GPU sharing Host Memory:         No
  Support host page-locked memory mapping:    Yes
  Alignment requirement for Surfaces:         Yes
  Device has ECC support:                     Disabled
  Device supports Unified Addressing (UVA):   No
  Device PCI Bus ID / PCI location ID:       15 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Version = 5.5, NumDevs = 1, Device0 = Quadro FX 1800
Result = PASS
local@debian7: /usr/local/cuda-5.5/samples/bin/x86_64/linux/release$

```

FIGURE A.2: CUDA test result

A.3 Installation of OptiX SDK

Unlike CUDA, there is no direct link to download an OptiX installation package. All developers need to create a special account: <https://www.surveymonkey.com/s/3D9FLR6>, two or three days later you will receive an email with your username and password inside. Then go to: <https://ftpservices.nvidia.com> and download corresponding installation package. Since the registration may take long, you can get a direct access with following coordinates:

- **Username:** OptiX-Beta-137
- **Password:** OptiX-Prime-Raytracing

To fully explore all features of OptiX, we choose to install the latest release at present: OptiX 3.5.1. Run the installation package:

```

$ sudo chmod a+x NVIDIA-OptiX-SDK-3.5.1-linux64.run
$ sudo ./NVIDIA-OptiX-SDK-3.5.1-linux64.run

```

A.3.1 Post-installation Setup

Establish links to OptiX library files:

```
$ sudo ln -s /home/local/NVIDIA-OptiX-SDK-3.5.1-PRO-linux64/lib64/liboptix.so.1 /usr/lib/  
liboptix.so.1
```

```
$ sudo ln -s /home/local/NVIDIA-OptiX-SDK-3.5.1-PRO-linux64/lib64/liboptixu.so.1 /usr/lib/  
liboptixu.so.1
```

```
$ sudo ln -s /home/local/NVIDIA-OptiX-SDK-3.5.1-PRO-linux64/lib64/liboptix_prime.so.1 /usr/lib/  
liboptix_prime.so.1
```

Appendix B

Acceleration Structures

An *Acceleration* consists of a builder and a traverser. The builder is responsible for collecting input geometry (in most cases, this geometry is the bounding boxes created by geometry nodes' bounding box programs) and computing a data structure that allows a traverser to accelerate a ray-scene intersection query. Builders and traversers are not application-defined programs. Instead, the application chooses an appropriate builder and its corresponding traverser from the table below [5]:

Builder / Traverser	Description
Lbvh / Bvh or BvhCompact	The Lbvh builder uses the HLBVH2 algorithm ⁴ to perform a very fast GPU-based bounding volume hierarchy build. It is good for many applications including very large or animated scenes where acceleration structure construction time dominates run time. But Trbvh will often be superior even for these use cases.
TriangleKdTree / KdTree	This builder constructs a high quality kd-tree, but is rarely comparable to the SBVH in ray tracing performance. Build times and memory footprint are usually higher for the kd-tree. This builder is specialized for triangle geometry and thus needs to be configured using certain properties (see Table 4). It is rarely the best choice and is no longer recommended.
NoAccel / NoAccel	This is a dummy builder which does not construct an actual acceleration structure. Traversal loops over all elements and intersects each one with the ray. This is very inefficient for anything but very simple cases, but can sometimes outperform real acceleration structures, e.g. on a group with very few child nodes.

FIGURE B.1: Acceleration Structures (1) [5].

Builder / Traverser	Description
Trbvh / Bvh	<p>The Trbvh¹ builder performs a very fast GPU-based BVH build. Its ray tracing performance is usually within a few percent of SBVH, yet its build time is generally the fastest. This new builder should be strongly considered for all datasets that fit. Temporarily, the Trbvh builder uses about three times the size of the final BVH for scratch space. OptiX Commercial includes a CPU-based Trbvh builder that does not have the memory constraints, and an optional automatic fallback to the CPU version when out of GPU memory.</p>
Sbvh / Bvh or BvhCompact	<p>The Split-BVH (SBVH) is a high quality bounding volume hierarchy. While build times are highest, it was traditionally the method of choice for static geometry due to its high ray tracing performance, but may be superseded by Trbvh. Improvements over regular BVHs are especially visible if the geometry is non-uniform (e.g. triangles of different sizes). This builder can be used for any type of geometry, but for optimal performance with triangle geometry, specialized properties should be set (see Table 4)².</p>
Bvh / Bvh or BvhCompact ³	<p>The Bvh builder constructs a classic bounding volume hierarchy. It focuses on quality over construction performance and delivers a good middle ground between the Sbvh and MedianBvh. It also supports refitting for fast incremental updates (Table 4).</p> <p>Bvh is often the best choice for acceleration structures built over groups.</p>
MedianBvh / Bvh or BvhCompact	<p>The MedianBvh builder uses a fast construction scheme to produce a medium-quality bounding volume hierarchy. It is typically useful for dynamic and semi-dynamic content, as well as for acceleration structures on groups.</p>

FIGURE B.2: Acceleration Structures (2) [5].

Appendix C

Time Management

This appendix presents a task list through the practice period and its corresponding Gantt diagram.

Generally speaking, most tasks were finished as they had had been scheduled. Especially in the first two months, time spent on bibliography work was much less than it had been estimated.

Major delay came from the implementation of object-oriented programming on GPUs and dynamic allocations. This part of work should have been done within one month. Unfortunately, it hasn't been finished during the internship. To a certain extent, other reasons like no available working computer in the first week or contacting researchers in the USA for technical supports postponed the advancement as well. Furthermore, the report redaction lasted for about one month.

Tasks List			
Name	Begin date	Resources	
		End date	
Bibliography: MODERATO, Ray tracing	3/10/14	4/30/14	
Reading OptiX Manuals	3/12/14	5/30/14	
Installation of OptiX	3/20/14	3/24/14	
Presentation about Ray Tracing Theory	3/24/14	3/26/14	
Compile OptiX Tutorial Codes	3/27/14	3/31/14	
1st Implementation	3/31/14	4/11/14	
Bibliography: Random Generator	4/7/14	4/18/14	
CURAND in OptiX	4/14/14	5/2/14	
2nd Implementation	5/5/14	5/9/14	
First Little MODERATO	5/12/14	5/16/14	
Debug of 1st Little Moto	5/16/14	6/6/14	
Prepare Second Presentation	6/6/14	6/16/14	
Change RNG	6/17/14	6/27/14	
Object-Oriented on CPU	6/27/14	7/10/14	
Dynamic Allocation	7/11/14	7/28/14	
Redaction of Report	7/25/14	8/27/14	
Correction and Last Presentation	8/27/14	9/3/14	
Ending	9/4/14	9/9/14	

FIGURE C.1: Task list.

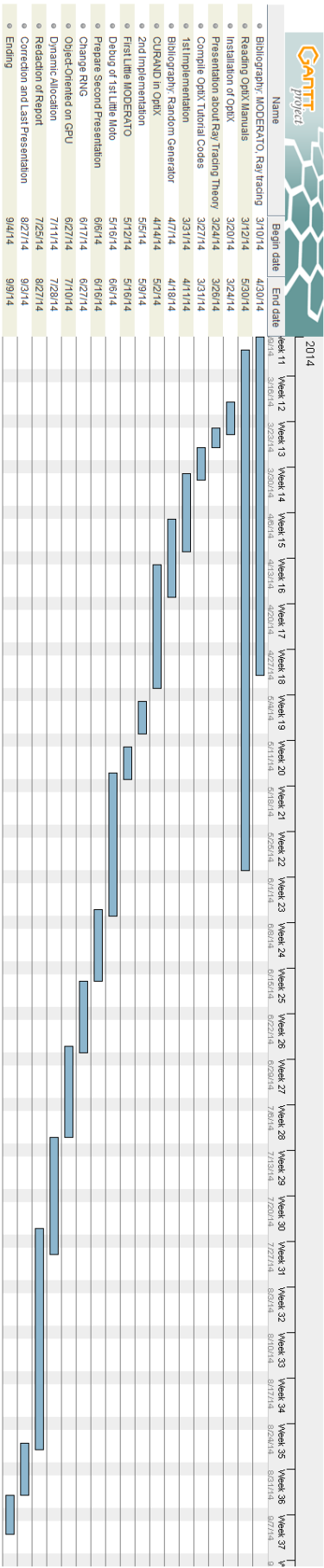


FIGURE C.2: Gantt diagram.

Bibliography

- [1] Paul Martinsen, Johannes Blaschke, Rainer Künnemeyer, and Robert Jordan. Accelerating monte carlo simulations with an nvidia graphics processor. *Computer Physics Communications*, 180(10):1983 – 1989, 2009. ISSN 0010-4655. URL <http://www.sciencedirect.com/science/article/pii/S0010465509001593>.
- [2] Ryan M. Bergmann. *The Development of WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs*. PhD thesis, University of California, Berkeley, Berkeley, California, USA, Spring 2014. URL <http://rmbergmann.com/RyanBergmann-dissertation-FINAL.pdf>.
- [3] *Monte Carlo method*. Wikipedia, 2014. URL http://en.wikipedia.org/wiki/Monte_Carlo_method.
- [4] James Tickner. Monte carlo simulation of x-ray and gamma-ray photon transport on a graphics-processing unit. *Computer Physics Communications*, 181(11):1821–1832, 2010.
- [5] *NVIDIA OptiX Ray Tracing Programming Guide*. NVIDIA Corporation, Santa Clara, California, 3.5.1 edition, February 2014.
- [6] A. Bonin and B. Lavayssière. New advances in the development of moderato. In *Conference Proceedings*, Roma, October 2000. 15th World Conf. on Non Destructive Testing.
- [7] *Électricité de France*. Wikipedia, 2014. URL http://en.wikipedia.org/wiki/%C3%89lectricit%C3%A9_de_France.
- [8] *R&D A CRUCIAL ELEMENT IN INDUSTRIAL PERFORMANCE*. The EDF Group, 2014. URL <http://businesses.edf.com/research/missions-94278.html>.

- [9] *Introduction of SINETICS*. The EDF Group, 2014. URL <https://intranet.edf.fr/web/rd-sinetics/lire-detail/-/>.
- [10] Louis Cartz. *Non-destructive Testing*, volume ISBN of 978-0-87170-517-4. A S M International, 1995.
- [11] *Nondestructive testing*. Wikipedia, 2014. URL http://en.wikipedia.org/wiki/Nondestructive_testing.
- [12] *Industrial radiography*. Wikipedia, 2014. URL http://en.wikipedia.org/wiki/Industrial_radiography.
- [13] *RT Film Making a Radiograph*. Bernoullies, 2014. URL http://en.wikipedia.org/wiki/Radiographic_testing#mediaviewer/File:RT_Film_Making_a_Radiograph.jpg.
- [14] *Three Cooling Circuits of Pressurised Water Reactor*. The Government of the Hong Kong Special Administrative Region, 2012. URL http://www.dbcp.gov.hk/eng/safety/plants_clip_image001_0000.gif.
- [15] Alice Bonin-Girardi. *Simulation de Monte Carlo d'un Système Radiographique et Expérimentations*. PhD thesis, École Normale Supérieure de Cachan, Chatou, France, March 2001.
- [16] Julian L. Simon. *Resampling: The New Statistics*, volume ISBN-13 of 978-0534217204. Resampling Stats (1995), 1st edition, 1995.
- [17] X George Xu, Tianyu Liu, Lin Su, Xining Du, Matthew Riblett, Wei Ji, and Forrest B Brown. An update of archer, a monte carlo radiation transport software testbed for emerging hardware such as gpus. *Transactions of the American Nuclear Society*, 108:16–20, 2013.
- [18] Tianyu Liu, X George Xu, and Christopher D Carothers. Comparison of two accelerators for monte carlo radiation transport calculations, nvidia tesla m2090 gpu and intel xeon phi 5110p coprocessor: A case study for x-ray ct imaging dose calculation. In *Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo, held 27-31 October, 2013 in Paris, France. Edited by D. Caruge, C. Calvin, CM Diop, F. Malvagi, and J.-C. Trama. EDP Sciences, 2014, id. 04205, 7 pp.*, volume 1, page 04205, 2014.

- [19] Lin Su, Youming Yang, Bryan Bednarz, Edmond Sterpin, Xining Du, Tianyu Liu, Wei Ji, and X George Xu. Archer-rt—a gpu-based and photon-electron coupled monte carlo dose computing engine for radiation therapy: Software development and application to helical tomotherapy. *Medical physics*, 41(7):071709, 2014.
- [20] T Liu, X Du, and X Xu. Tu-g-103-05: Affordable supercomputer-based monte carlo ct dose calculations: A hardware comparison between nvidia m2090 gpu and intel xeon phi 5110p coprocessor. *Medical Physics*, 40(6):459–459, 2013.
- [21] *Computer Cluster*. Janalta Interactive Inc., 2014. URL <http://www.techopedia.com/definition/6581/computer-cluster>.
- [22] *Case Study: Achieving High Performance on Monte Carlo European Option Using Stepwise Optimization Framework*. Shuo Li (Intel), September 6th, 2013. URL <https://software.intel.com/en-us/articles/case-study-achieving-high-performance-on-monte-carlo-european-option/using-stepwise>.
- [23] M. Bernaschi, M. Bisson, and F. Salvatore. Multi-kepler {GPU} vs. multi-intel {MIC} for spin systems simulations. *Computer Physics Communications*, 185(10):2495 – 2503, 2014. ISSN 0010-4655. URL <http://www.sciencedirect.com/science/article/pii/S0010465514002008>.
- [24] K Souris, J Lee, and E Sterpin. Th-a-19a-08: Intel xeon phi implementation of a fast multi-purpose monte carlo simulation for proton therapy. *Medical Physics*, 41(6):535–535, 2014.
- [25] E Sterpin, J Sorriaux, J Schuemann, K Souris, J Lee, S Vynckier, X Jia, S Jiang, and H Paganetti. Su-et-180: Fano cavity test of proton transport in monte carlo codes running on gpu and xeon phi. *Medical Physics*, 41(6):264–264, 2014.
- [26] Hidenobu Tachibana Georgios Kalantzis. Accelerated event-by-event monte carlo microdosimetric calculations of electrons and protons tracks on a multi-core cpu and a cuda-enabled gpu. *Computer methods and programs in biomedicine*, 113(1):116–125, 2014.
- [27] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. 2014.

-
- [28] *Embree: High Performance Ray Tracing Kernels*. Embree Group, Santa Clara, CA, USA, 2.3.2 edition, July 23 2014.
- [29] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.
- [30] *NVIDIA OptiX ray tracing engine*. OptiX Group, 2014. URL <https://developer.nvidia.com/optix>.
- [31] Jakub Pietrzak and Krzysztof Kacperski. Ultra fast ray-tracer for medical imaging with nvidia optix engine. In *Conference Presentations*, San Jose, CA, March 2013. GPU Technology Conference.
- [32] M.J. Safari, H. Afarideh, N. Ghal-Eh, and F. Abbasi Davani. Optix: A monte carlo scintillation light transport code. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 737(0):101 – 106, 2014. ISSN 0168-9002. URL <http://www.sciencedirect.com/science/article/pii/S0168900213015921>.
- [33] *OPTIX API REFERENCE*. Doxygen, NVIDIA Corporation, Santa Clara, California, 3.5 edition, February 2014.
- [34] *CUDA 6.0 Performance Report*. CUDA group, NVIDIA Corporation, Santa Clara, California, April 2014. URL developer.download.nvidia.com/compute/cuda/6_0/rel/docs/CUDA_6_Performance_Report.pdf.
- [35] Stephen Marsland. *Machine learning: an algorithmic perspective*. CRC Press, 2011.
- [36] Pierre L’Ecuyer. *Multiple Streams of Random Numbers for Parallel Computers: Design and Implementation*. CEA, Saclay, June 2014.
- [37] *CURAND LIBRARY: Programming Guide*. CUDA group, NVIDIA Corporation, Santa Clara, California, 5.5 edition, July 2013. URL <http://docs.nvidia.com/cuda/curand/index.html>.
- [38] Pierre L’ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

-
- [39] Mutsuo Saito and Makoto Matsumoto. *A deviation of CURAND: standard pseudo-random number generator in CUDA for GPGPU*. Hiroshima University, University of Tokyo, February 13 2012.
 - [40] *CUDA Programming Guide*. CUDA group, NVIDIA Corporation, Santa Clara, California, 5.5 edition, July 2013. URL <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
 - [41] Wen-mei Hwu and David Kirk. Programming massively parallel processors. *Special Edition*, 92, 2009.

Abstract

NVIDIA OptiX provides developers with a highly optimized ray tracing platform on GPU. It manages automatically traditional GPU implementation problems like keeping thread blocks busy, minimizing data transfers, optimizing use of hardware and so on. With OptiX ray tracing engine, users may fully concentrate on the content itself without worry about specialized GPGPU architecture. Furthermore, OptiX allows users to assemble easily their own programs with the built-in tracking mechanism.

EDF R&D developed MODERATO to simulate the industrial radiography process. This Monte Carlo code employs ray tracing method to track gamma ray propagations and generates simulation images, but such calculations are extremely time-consuming. As a result, acceleration of this Monte Carlo simulation appears to be very important.

During this six-month internship, I explored OptiX's ray tracing potentials for general-purposed scientific computing. Major features of MODERATO implemented with this ray tracer. Test results indicate that OptiX implementation is really powerful since it attained the same computing performance as a CPU cluster with 8 Intel Xeon E5-2697 processors (96 cores). In conclusion, OptiX is a well suited ray tracing platform for scientific Monte Carlo simulations.

Résumé

Le code MODERATO développé par EDF simule le procédé de contrôle non destructif par radiographie X et gamma. Il s'agit d'une simulation Monte Carlo qui permet de reproduire et de prévoir les phénomènes physiques intervenant dans le cadre de l'inspection de centrales nucléaires. En particulier, cette simulation emprunte une modélisation de lancer de rayons afin de simuler le déplacement des photons.

Depuis récemment, Nvidia propose une bibliothèque de lancer de rayons, OptiX, optimisée pour l'architecture CUDA. En utilisant cette API pour un calcul scientifique, les développeurs peuvent se concentrer sur les problèmes de simulation et laisser OptiX prendre en charge le matériel et le parallélisme.

Pendant ces six mois de stage, ma mission principale a été d'étudier le portage de l'algorithme Monte Carlo sur l'accélérateur GPU, en explorant la possibilité de s'appuyer sur OptiX. Selon des tests qu'on a réalisé, OptiX fournit une performance au même niveau qu'un CPU cluster avec 8 Xeon E5-2697 processeurs (96 coeurs). On peut conclure qu'OptiX est un outil de lancer de rayons flexible et ouvert pour le calcul scientifique. Il nous permet d'avoir une accélération impressionnante en changeant peu l'algorithme initial.