Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

TB - GPU optimization - Celeritas
Project report

Computer science and Communication System (ISC), 2022-2023

Student	Simon Barras
Supervisors	Prof. Frédéric Bapst - HEIA-FR Prof. Jean Hennebert - HEIA-FR
Customer	Lawrence Berkeley National Laboratory (LBNL) Paolo Calafiura Julien Esseiva
Expert	Dr. Baptiste Wicht

v1.1
Berkeley, CA, USA, August 10, 2023

Executive Summary

This Bachelor thesis has been made by Simon Barras, a student of the Haute Ecole d'Ingénierie et d'Architecture de Fribourg. The goal is to improve the performance of the Celeritas, a Monte Carlo particle transport library for simulating High Energy Physics (HEP) detectors. tool for high-energy physics. This project is accelerated by GPUs and the focus is on the improvement of the RKDP algorithm. This famous sequential method needs to be analyzed to define the best way to parallelize it. The final goal is to implement a distributed version of the RKDP with multiple GPUs threads that work together to simulate a large number of particles.

Version history

Version	Changes	Date
0.1	Add analysis chapter	01.08.2023
0.2	Add design chapter	02.08.2023
0.3	Fix analysis and design chapter with the comments of my supervisor. The chapter about the `SchedulingTree` has been improved.	04.08.2023
0.4	Add implementation chapter	06.08.2023
0.5	Add result chapter	07.08.2023
1.0	Add introduction and conclusion chapter. Aside chapter like acknowledgments have been made too	08.08.2023
1.1	Correct the report with comments of Julien Esseiva	09.08.2023

Table 1: Version history

Contents

1 Introduction

This Bachelor thesis is done by Simon Barras and supervised by Frederic Bapst and Jean Hennebert. The customer Paolo Calafiura is a physicist and computer scientist at the Lawrence Berkeley National Laboratory (LBNL). To do this project, I am moving to Berkeley, California, for ten weeks. The goal is to explore a way to improve the performance of the project Celeritas, which is a particle physics simulation software accelerated by GPUs.

The two main customers are CMS and ATLAS, two experiments are made at the European Organization for Nuclear Research (CERN) with the Large Hadron Collider (LHC). They are both using Geant4 and they have not committed to using Celeritas beyond an initial evaluation.

Detector simulation is used to validate and calibrate the algorithms used to estimate the properties of the primary particles from the observed detector data. The main goal of the thesis will be to optimize a GPU-accelerated version of the Prince Dormand algorithm [1], a Runge-Kutta solver [2] for the differential equations governing the trajectory of particles in a non-uniform magnetic field. This work will improve the project Celeritas [3] which may replace Geant4 [4] in the future.

The ATLAS experiment tracks the path of particles in the detector and produces coordinates points where particles traverse the sensors. Figure 1.1 represents this experiment.

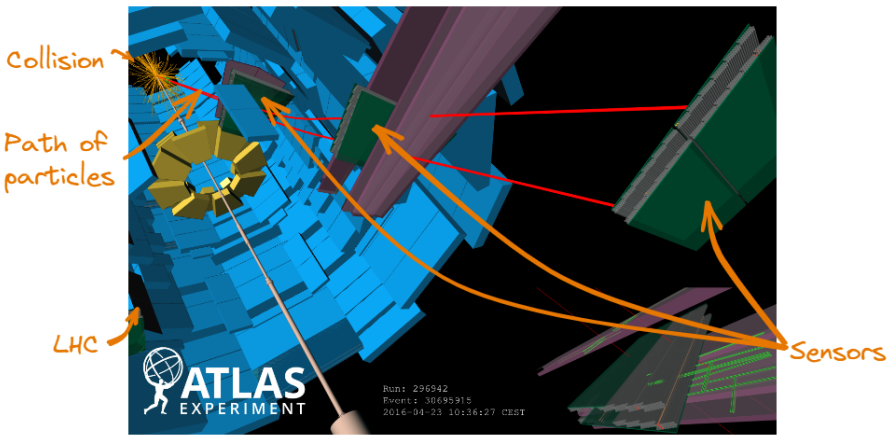

Figure 1.1: ATLAS experiment at CERN [5]

1.1 Lawrence Berkeley National Laboratory

The Lawrence Berkeley National Laboratory (LBNL) is a national laboratory in Berkeley, California. It is managed and operated by the University of California for the Department of Energy (DOE). The lab is situated in the hills of Berkeley and it is composed of many buildings and has a beautiful view of the San Francisco Bay (see Fig.1.2). This laboratory is mentioned in the recently released movie by Christopher Nolan, "Oppenheimer".

Figure 1.2: Lawrence Berkeley National Laboratory

1.2 The objectives

Celeritas is already accelerated by GPUs, however, the team wants to improve the performance. In the current version of the code, each particle track is processed in parallel by one GPU thread, with no collaboration between threads. GPU profiling of the code shows that execution time is dominated by two kernels. The first one is handled by the interaction with the detector geometry to know where, in 3D space, the particle is situated and during the profiling, the library used is vecGeom [6]. The second kernel, which will be the focus of this Bachelor thesis project, is the computation of a differential equation using Dormand-Prince [1].

However, the thesis can be a success even if the project doesn't meet the improvement. This is because we don't yet know whether thread synchronization will be more time-consuming than the original version. In addition, the kernel launch in Celeritas has to be changed, and this could take up a considerable amount of optimization time. For the Bachelor thesis, it is sufficient to have a proof of concept that demonstrates if the enhancement is effective and deserves to be integrated.

1.2.1 Learn GPU programming

Before starting to work on the project, some things need to be learned and the goal here is to learn a new way of programming. To conclude this goal, no code will be produced except for exercises, but the important notions of GPU programming with CUDA will be synthesized using cheat sheets. To take advantage of the delay between the beginning of the Bachelor thesis and the beginning of the LBNL internship, this step will be done during this time.

1.2.2 Understand the project

To be able to improve the performance of the code, the first step is to understand the project and it's always better to understand the background: why it is needed, who will use it and which paradigm and tools are used. This step will be done at the beginning of the project on-site.

1.2.3 Improve the performance

The main goal of the project is to improve the performance of the implementation of Runge Kutta Dormand Prince (RKDP) method [1]. This last mandatory requirement is the core of the thesis and the most important part of the project and it will require the knowledge gained in the first two steps to improve the performance. The realization is done in a test that computes the time of the different implementations to compare them.

1.3 Optional requirements

These optional goals are good additions to the project but they are not required to have a concrete result.

The first one is to have a portable performance. The purpose of Celeritas is to be run on all kinds of GPU and even on machines with just a CPU. During the optimization, the improvement will be checked on the Perlmutter [7] which uses Nvidia A100 with the architecture Ampere [8] and some improvement can be only effective to this kind of GPU. This goal is here to check if the improvement has a positive effect on other architectures and if it is not, to find a way to do that. To begin this step, the main goal needs to be finished. Celeritas is made to be run on AMD GPUs too.

The second optional objective is to do another performance improvement. If the performance of the Runge-Kutta method [2] is improved, another optimization can be done. This part goal will be discussed further in the project with the supervisors and the customers and it will be managed like the last mandatory goal. This step can be done multiple times if there is enough time.

2 Analysis

During this chapter, the project is analyzed to understand the context, the goals and challenges. This project is focused on the optimization of particle tracking using GPUs. GPU's programming is something totally new for a Bachelor's student at the HEIA-FR. Celeritas is a project that is actively developed and it requires time to understand the code and the architecture.

2.1 GPU

The Graphics Processing Unit (GPU) is a processor that is specialized in parallel computing. It is known by the gamer community because it is used to render the graphics of the games and a good graphic card increases the frames per second displayed. However, in the professional world, the GPU are becoming more and more popular because they are more efficient than the CPU for parallel computing and managing a large amount of data without spending much energy.

2.1.1 Use cases

In most cases, CPUs are faster than GPUs to deal with independent tasks which is why they are a mandatory component of a computer. CPUs have a high frequency and they are optimized to execute one task at a time with a high efficiency. On the other hand, GPUs have a lower frequency and they are optimized to execute multiple tasks at the same time.

To understand, imagine John wants to eat dinner with his 5 kids and they have to cook something and dress the table. Cooking is a task that must be done sequentially and require some skills and efficiency to be done quickly. Dressing the table is a task where the skills don't affect the time but the number of hands does. This task can be done by the kids when everyone is putting something on the table and cooking is done by John because he will be faster than the kids. In our case, John is the CPU and the kids are the GPUs.

The GPUs are used when the number of workers is more important than the efficiency of the worker like computing the pixels of an image or the particles of a simulation.

2.1.2 Architecture

Table 2.1 shows the different types of processors. The capabilities to work with one or multiple data and instructions are defined by the architecture of the processor.

	Single data	Multiple data
Single instruction	SISD	**SIMD**
Multiple instructions	MISD	MIMD

Table 2.1: Different types of processors working with one or multiple data and instructions

The GPU is a SIMD processor and the CPU is a SISD processor. The table 2.1 does not show the frequency of the processor is an important factor for performance. This explains why the CPU is faster than the GPU for most of the cases. To be fair, things are a bit more complicated and there is more than just four types of processors and one processor can be classified into multiple categories. Concerning the GPU and according to the CUDA documentation [9], it is possible to program with the Single Instruction Multiple Thread model.

To deal with the SIMD architecture, the GPU executes physically the same instruction on multiple data using multiple threads as shown on Figure 2.1.

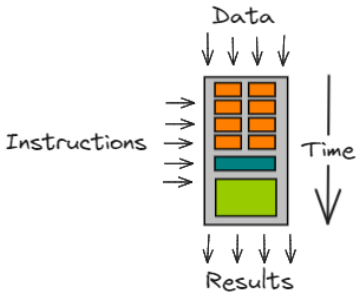

Figure 2.1: SIMD physically executed

The GPU is composed of multiple Streaming Multiprocessor (SM) that are composed of multiple Compute Unified Device Architecture (CUDA) cores. The CUDA cores are the processors that execute the instructions. The SM is the unit that manages the threads and the memory. Figure 2.2 shows the architecture of a SM. Ampere [8] SMs are split into four warps schedulers.

Figure 2.2 introduces the warp that is a group of 32 threads that are executed in parallel. The threads of a warp are linked together and they have to wait for the others to finish their instructions.

Figure 2.2: SM architecture [10]

2.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by Nvidia. It allows developers to use the GPU to execute code written in C++, C, Fortran and Python. The CUDA platform comes with a SDK that provides libraries, debugging and profiling tools.

To develop with CUDA, the SDK must be installed and all the examples in this report are with C++ because it is the language used in the project.

It is recommended to have some basic knowledge to understand in GPU programming because the following sections are not written as a tutorial but as a reminder to have all the most important information in one place. The course [11] which has been dispensed at ORNL is a good way to learn CUDA.

2.2.1 Basis

The code that runs on a GPU is in a function called "kernel" and it is executed by every thread. Those kernels are defined with the keyword `__global__` and they are called with the function `kernel_name<<<number_of_blocks, number_of_threads>>>()`. The code 2.1 shows a basic kernel launch with 2 blocks of 3 threads which means that the kernel code will be executed 6 times.

Listing 2.1: Basic kernel example

```
__global__ void kernel_name() {
    // Code executed on the device
```

```
}

int main() {
    // Code executed on the host
    kernel_name<<<2, 3>>>();
    return 0;
}
```

2.2.2 Memory

As the GPU is a separate processor, it has its own memory space. As a developer, we have to manage three different spaces where data can be stored:

- Host memory (RAM)
- Device memory (DRAM)
- Shared memory

Every space comes with its properties and using the right one is important to have a good performance. Figure 2.3 shows the physical organization of the memory on a GPU.

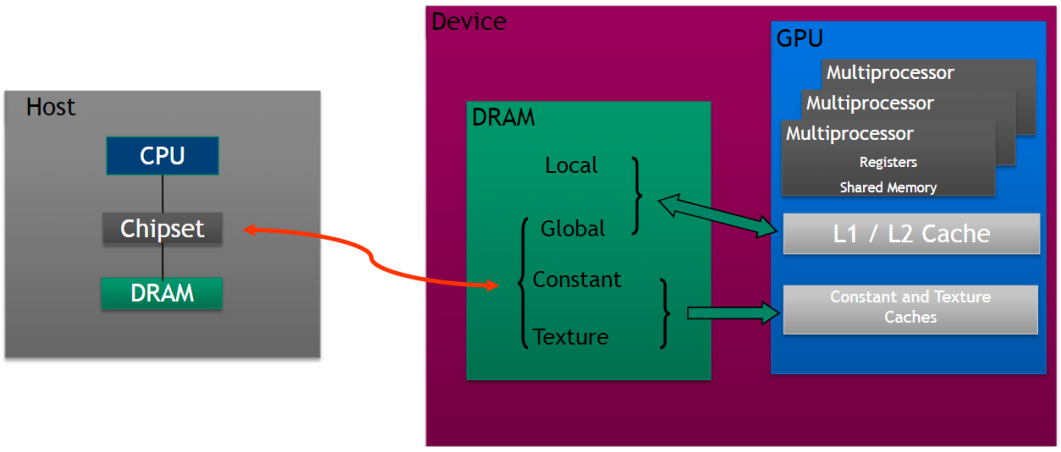

Figure 2.3: Physical memory organization on a GPU [11]

Host memory

Memory allocated on the host is the memory we use every day but it is not accessible by the GPU. Even if a program is using the GPU, it still needs to use the host memory.

Device memory

The device memory is the memory that is used by the GPU to execute the kernel and it is comparable to a heap and a stack memory. This place contains the local variables of a thread, the global variables and the constant memory. Ideally, the local variables stay in the registers which are closer to the core and this is faster. If the kernel needs too much memory, CUDA will store the data in the global memory. Most of the time, the registers are enough. This memory is accessible by all the threads of a grid and the data can be transferred from/to the host memory via specific statements on the host code.

The arguments passed to a kernel can be 32'764 Bytes long but the object cannot be passed by reference. To pass a dynamic object, it must be given with his pointer and the value must be copied into the device's memory. To get the result, the data must be copied back to the host memory. The code 2.2 shows how to launch a kernel with objects and integers as arguments. As the kernel isn't executed on the host, the return keyword can't be used to get the result of the kernel so the result must be copied back to the host using the same mechanism as the arguments.

Listing 2.2: Load and read data from the device memory

```
__global__ void kernel_name(ObjectType1 *input1,
                            int input2,
                            ObjectType2 *output) {
    ObjectType3 local_variable;
    output[threadIdx.x] = ObjectType2(input1[threadIdx.x] + input2);
}

int main() {
    // Instantiate the variables
    ObjectType1 *h_input1, *d_input1;
    ObjectType2 *h_output, *d_output;

    // Setting the values
    h_input1 = new ObjectType1[100];
    for (int i = 0; i < 100; i++) {
        h_input1[i] = ObjectType1(i);
    }
    h_output = new ObjectType2[100];
    int input2 = 5;

    // Allocate the memory on the device
    cudaMalloc(&d_input1, 100 * sizeof(ObjectType1));
    cudaMalloc(&d_output, 100 * sizeof(ObjectType2));

    // Copy the data from the host to the device
    cudaMemcpy(d_input1,
               h_input1,
               100 * sizeof(ObjectType1),
               cudaMemcpyHostToDevice);

    // Launch the kernel
    kernel_name<<<1, 100>>>(d_input1, input2, d_output);

    // Copy the data from the device to the host
    cudaMemcpy(h_output,
               d_output,
               100 * sizeof(ObjectType2),
               cudaMemcpyDeviceToHost);

    // Free the memory
    cudaFree(d_input1);
    cudaFree(d_output);

    // Do something with the output
    print(h_output);

    return 0;
}
```

The instruction `new` should not be used in modern C++, creating a simple stack variable is enough and if the pointer is needed, it should be created with `std::make_unique` or `std::make_shared`.

Expressions marked with `constexpr` and methods that will be used on the device must have the keyword `__device__`. As this keyword indicates that the method will be compiled by `nvcc` and it will not be available anymore on the host. To compile two versions of the method, one for the host and one for the device, the double keyword `__host__ __device__` must be used.

Shared memory

The shared memory is a memory that is shared between the threads of a block. It is quicker than the device memory but it is limited per block. In the GPU used it is 48 KB but the limitations are described in the documentation [12]. The shared memory is used to share data between the threads of a block and to cache data from the global memory or exchange data between the threads.

The shared memory is allocated with the keyword `__shared__` and it can be done statically (see code 2.3) or dynamically (see code 2.4). As the examples are very close to the code 2.2, the details to copy the data from the host to the device and from the device to the host are not shown.

Listing 2.3: Static shared memory allocation

```
__global__ void kernel_name(ObjectType1 *input,
                            int input2,
                            ObjectType2 *output) {
    __shared__ ObjectType3 shared_variable[100];
    shared_variable[threadIdx.x] = ObjectType3(input[threadIdx.x] + input2);
    int index_next_thread = (threadIdx.x + 1) % 100;
    __syncthreads();
    output[threadIdx.x] = ObjectType2(shared_variable[index_next_thread]);
}

int main() {
    // Instantiate the variables
    // Setting the values
    // Allocate the memory on the device
    // Copy the data from the host to the device
    // Launch the kernel
    kernel_name<<<1, 100>>>(d_input1, input2, d_output);

    // Copy the data from the device to the host
    // Free the memory
    // Do something with the output
    return 0;
}
```

Listing 2.4: Dynamic shared memory allocation

```
__global__ void kernel_name(ObjectType1 *input,
                            int input2,
                            ObjectType2 *output) {
    extern __shared__ ObjectType3 shared_variable[];
    shared_variable[threadIdx.x] = ObjectType3(input[threadIdx.x] + input2);
    int index_next_thread = (threadIdx.x + 1) % 100;
    __syncthreads();
```

```
    output[threadIdx.x] = ObjectType2(shared_variable[index_next_thread]);
}

int main() {
    // Instantiate the variables
    // Setting the values
    // Allocate the memory on the device
    // Copy the data from the host to the device
    // Compute the size of the shared memory
    int size_shared_memory = 100 * sizeof(ObjectType3);

    // Launch the kernel
    kernel_name<<<1, 100, size_shared_memory>>>(d_input1, input2, d_output);

    // Copy the data from the device to the host
    // Free the memory
    // Do something with the output
    return 0;
}
```

2.2.3 Synchronization

The easiest way to use GPU is to have a set of data and using one thread to update one data, for instance when you want to compute the sum of two vectors. Next, there are multiple dimension problems with matrix or 3D matrix addition. The most difficult type of problem is the reduction where the threads must communicate between them to get the result. For example, the sum of all the elements of a vector is illustrated in Figure 2.4.

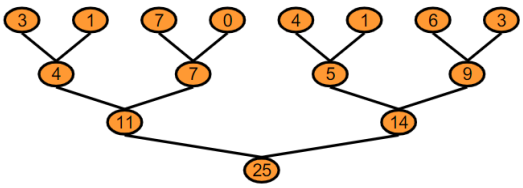

Figure 2.4: Reduction problem [11]

CUDA provides some low-level function to synchronize, preserve the integrity or exchange data.

Atomic operations

Atomic operations are used to preserve the integrity of the data when multiple threads are trying to access the same data. For example, if two threads are trying to increment a shared integer, the result will be wrong because the two threads will read the same value and write the same value so the result will be incremented only once.

Every atomic operation requires a pointer to the data that could eventually be modified and a value. The value could have different roles but the instruction is returning the old value stored there. The different atomic operations are listed in Table 2.2.

Warp shuffle

The warp shuffle is a feature that allows developers to exchange data between the threads of a warp. This mechanism is limiting the time waste to exchange data using shared memory but it is only working for 4 or 8 bytes.

Operation	Description
`atomicAdd/Sub(addr, val)`	Add a value to an integer
`atomicMin/Max(addr, val)`	Set the minimum/maximum value
`atomicInc/Dec(addr, val)`	Increment/Decrement an integer if the new value will be from 0 to val
`atomicCAS(addr, compare, val)`	Swap value to addr if old is equal to compare
`atomicExch(addr, val)`	Swap value to addr
`atomicAnd/Or/Xor(addr, val)`	Bitwise And/Or/Xor

Table 2.2: CUDA atomic operations

Those warp instructions are listed in Table 2.3. They return the value of the variable specified from the thread lane specified. The mask is a 32-bit unsigned int that represents the lane-id in one-enabled bit. Only the threads specified by this mask will wait for the synchronization. Var is the name of the variable that will be pulled from the thread lane.

Operation	Description
`__shfl_sync(mask, var, lane)`	Get the value from the thread lane
`__shfl_up/down_sync(mask, var, delta)`	Get the value from the thread lane ± delta
`__shfl_xor_sync(mask, var, laneMask)`	Get the value from the thread lane ^ laneMask

Table 2.3: CUDA warp shuffle operations

Sync

Warp shuffle offers a synchronization method but CUDA provides functions to synchronize the threads. This could be useful if the threads have to exchange data between them using the shared memory.

Table 2.4 shows the different synchronization functions.

Operation	Description
`__syncthreads()`	Synchronize all the threads of a block
`__threadfence()`	Synchronize all memory accesses of a block
`__synchwarp(mask)`	Synchronize the threads of a warp that match the mask (default: all)

Table 2.4: CUDA synchronization functions

2.2.4 Create a project with CUDA

CUDA programming can be easily integrated in any C++ project. The main constraint is to have a GPU to execute the code and it is easy to build on it to be sure that version is the same. If the working station doesn't have a GPU, it is possible to use a SSH connection to a server with a GPU.

To do that with CLion, the first step is to have a project and then in the settings, add a toolchain with the CUDA compiler (Figure 2.5). Then in the CMake profile set the toolchain set before (Figure 2.6).

Usually, the kernels are defined in a `.cu` file and the host code is in a `.cc` file but everything can be in the same file. To run the code, the most convenient way is to use CMake to compile the code and then run the executable. Code 2.5 shows a basic CMake file to compile a project with CUDA.

Listing 2.5: Basic CMake file to compile a project with CUDA

```
cmake_minimum_required(VERSION 3.16)
project(project_name LANGUAGES CXX CUDA)

set(CMAKE_CXX_STANDARD 14)

# add CUDA
find_package(CUDA REQUIRED)

# add executable
add_executable(executable_name executable_file.cu)
```

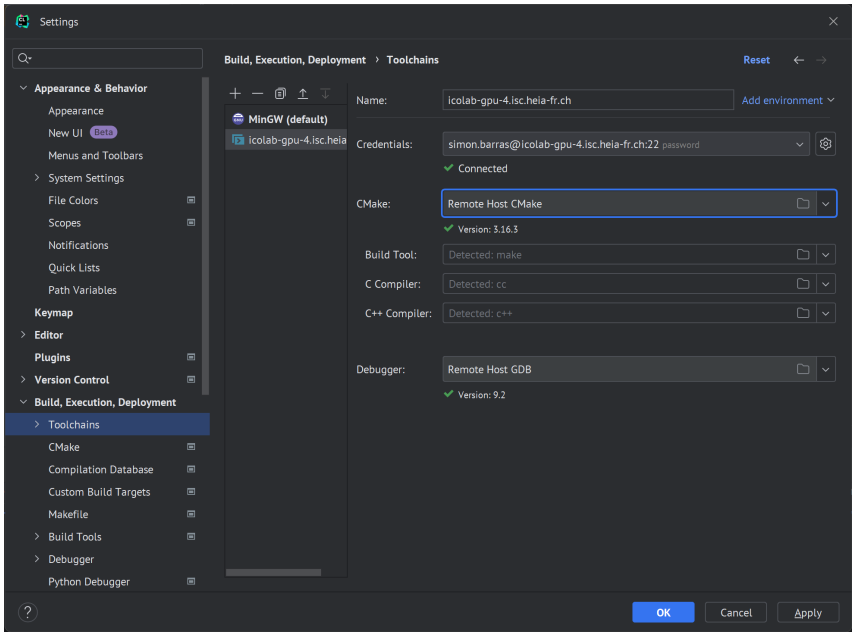

Figure 2.5: Add CUDA toolchain

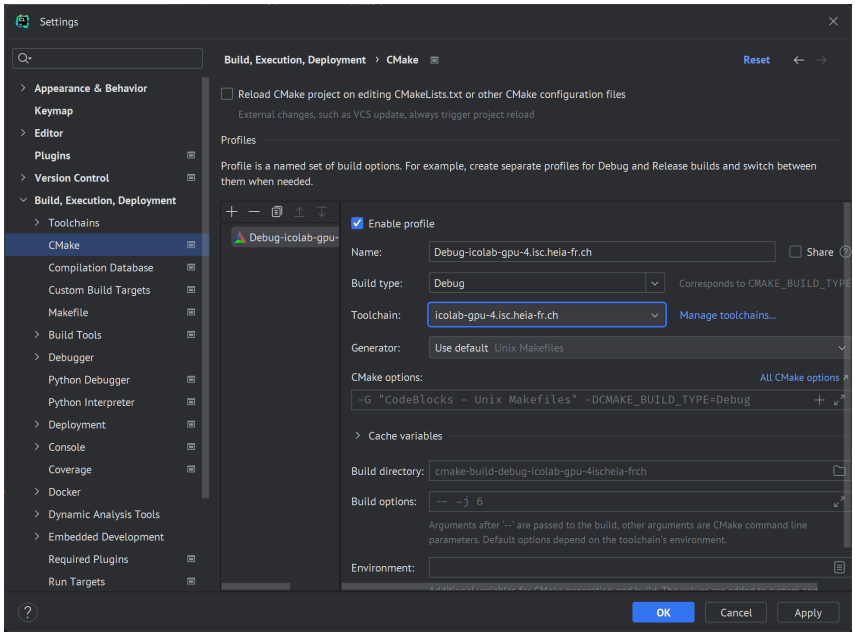

Figure 2.6: Set profile to toolchain

2.3 Celeritas

Celeritas is a High Energy Physics (HEP) detector simulation on GPUs started early 2020. This project purposed to the large amount of data produced by the Large Hadron Collider (LHC) that is not possible to be processed by traditional CPUs. As shown in Figure 2.7 The project is actively developed by five laboratories that are: Lawrence Berkeley National Laboratory (LBNL), Oak Ridge National Laboratory (ORNL), Argonne National Laboratory (ANL), Fermi National Accelerator Laboratory (FNAL) and Brookhaven National Laboratory (BNL). The code leader is Seth Johnson from ORNL and Julien Esseiva is the only participant from the LBNL. The goal is to simulate electromagnetic physics for the LHC detectors with the same precision as Geant4 [4].

2.3.1 Geant4

Celeritas is a library that can be integrated into a project to simulate the path of particles. Figure 2.8 shows the integration diagram of Celeritas into a HEP software.

However, developing all the features for tracking needs a lot of effort to be as complete as Geant4. Celeritas has a library called "Acceleritas" that has been made to be easily integrated with Geant4 [4]. The integration with Geant4 is shown in Figure 2.9.

Geant4 [4] is a toolkit for the simulate the path of particles through matter. It is used in multiple fields, including the experiments made in the accelerator physics. This tool is developed by the European Organization for Nuclear Research (CERN) and is currently used by the ATLAS and CMS experiments.

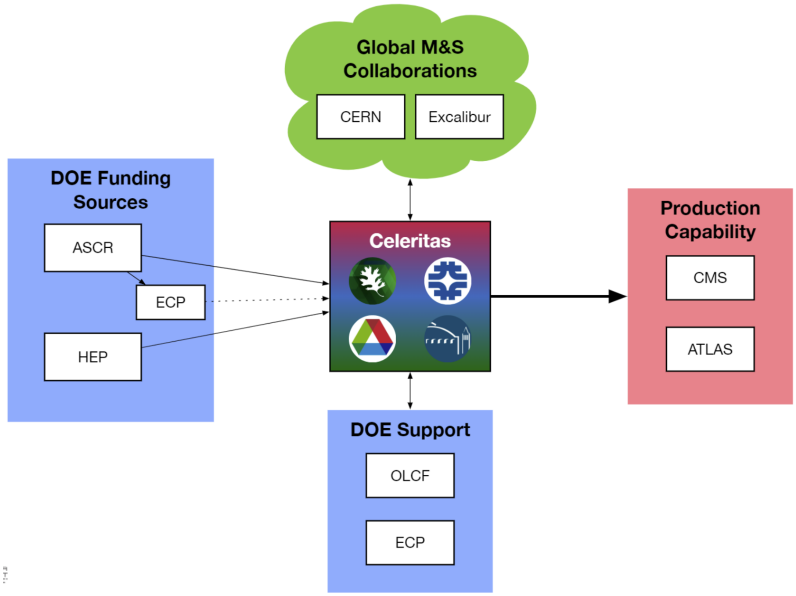

Figure 2.7: Celeritas actors and roles [13]

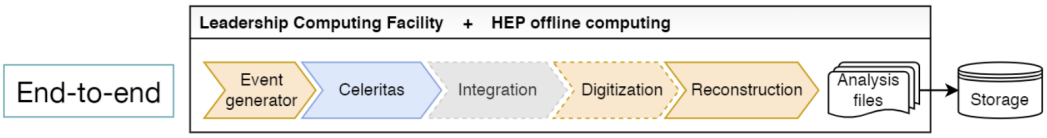

Figure 2.8: Integration diagram of the standalone application [14]

2.3.2 Runge Kutta Dormand Prince

The Runge Kutta Dormand Prince (RKDP) is a numerical method to solve ordinary differential equations. It is defined via the coefficients from the Butcher tableau which is a low triangular matrix. The method is used to simulate the path of a particle through matter and a magnetic field.

This is the Butcher tableau 2.5 for the RKDP used in Celeritas.

0							
1/5	1/5						
3/10	3/40	9/40					
4/5	44/45	-56/15	32/9				
8/9	19372/6561	-25360/2187	64448/6561	-212/729			
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656		
1	35/384	0	500/1113	125/192	-2187/6784	11/84	
	35/384	0	500/1113	125/192	-2187/6784	11/84	0
	5179/57600	0	7571/16695	393/640	-92097/339200	187/2100	1/40

Table 2.5: Butcher tableau for the RKDP

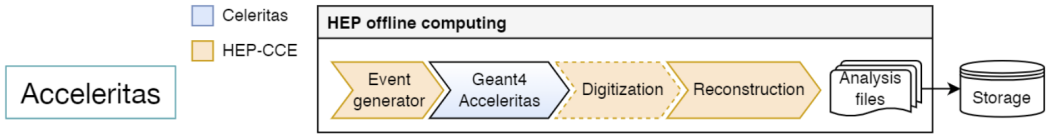

Figure 2.9: Integration diagram for overloading of Geant4 [14]

2.3.3 Particle's path

Celeritas simulates the path of a particle through matter and magnetic fields. The path of a particle is calculated by the RKDP 2.3.2 method.

As Celeritas is actively developed, not all the particles are now implemented. Celeritas is targeting production use for the ATLAS and CMS and the implemented particles are listed in Table 2.6.

Particle	Process	Model(s)
γ	photon conversion	Bethe—Heitler
	Compton scattering	Klein—Nishina
	photoelectric effect	Livermore
	Rayleigh scattering	Livermore
$e^{\pm}$	ionization	Mø11er-Bhabha
	bremsstrahlung	Seltzer—Berger, relativistic
	pair annihilation	EPlusGG
	multiple scattering	Urban, WentzelV1
$\mu^{\pm}$	muon	Muon Bremsstrahlung

Table 2.6: Particles now implemented in Celeritas [15]

The team is working on expanding the project to other experiments and has planned to add the following particles 2.7.

Process defines what is happening, e.g., photon conversion is a photon creating an electron and a positron (e-, e+). The model implements what is happening when that process occurs (change in momentum, direction, energy, ...).

2.3.4 Implementation

Celeritas is implemented in C++ and uses CUDA or HIP to accelerate the simulation. The code is data-oriented and is using composition instead of inheritance. To optimize the kernel launching on the GPU, all the data are loaded one time at the beginning of the simulation.

Celeritas is executing a sequence of actions for a simulation step, as shown in Figure 2.10.

Physics	Process	Particle(s)
EM	Photon conversion	γ
	pair annihilation	$e^{\pm}$
	photoelectric effect	γ
	ionization	charged leptons, hadrons and ions
	bremsstrahlung	charged leptons and hadrons
	Rayleigh scattering	γ
	Compton scattering	γ
	Coulomb scattering	charged leptons and hadrons
	multiple scattering	charged leptons and hadrons
	continuous energy loss	charged leptons, hadrons and ions
Decay	two body decay	$\mu^{\pm}$, $\tau^{\pm}$, hadrons
	three body decay	$\mu^{\pm}$, $\tau^{\pm}$, hadrons
	n-body decay	$\mu^{\pm}$, $\tau^{\pm}$, hadrons
Hadronic	photon-nucleus	γ
	lepto-nucleus	leptons
	nucleon-nucleon	p, n
	hadron-nucleon	hadrons
	hadron-nucleus	hadrons
	nucleus-nucleus	hadrons

Table 2.7: Particles planned in the future version of Celeritas [15]

My own Bachelor project is focusing on optimizing a part of particle simulation which is made in the action called "Along Step". This action is executed once per simulation step and it applies to each particle at the same time by one dedicated GPU's thread per track. Figure 2.12 shows the activity diagram of the particle's track.

The method RKDP (see Section 2.3.2) is used to calculate the chord of the path during the action "Calculate substep from a remaining distance" in Figure 2.12. The chord, with different step sizes, is computed until the error meets the tolerance or the number of tries is reached. A chord is a straight line between two points and the error is the difference between the chord and the real path. If the tolerance is not met, an ultimate iteration will be made and the error is provided.

According to the result of an experiment made by the Compact Muon Solenoid (CMS). Figure 2.11 shows our statistics which say that in 80% of the cases, the tolerance is met at the first iteration.

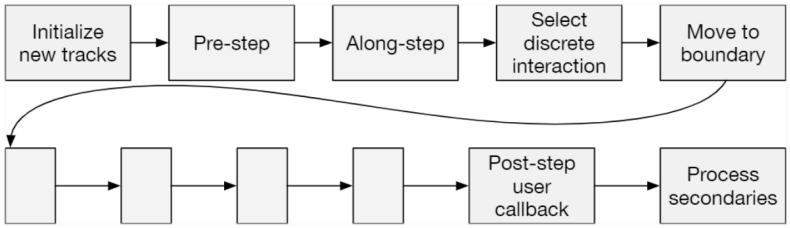

Figure 2.10: Celeritas's actions activity diagram [16]

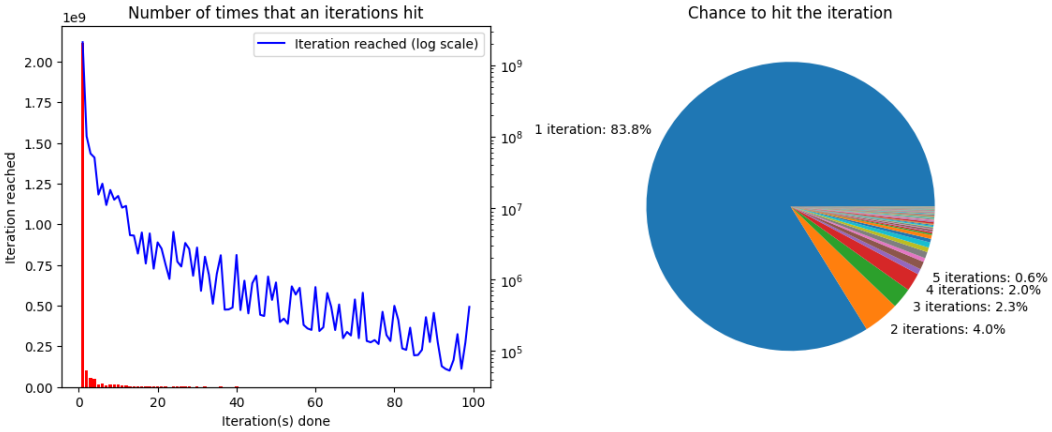

Figure 2.11: Number of iterations needed to meet the tolerance for each computed chord

2.3.5 Optimization

so far Celeritas was focusing mostly on getting a baseline set of features necessary to be useful and hasn't spent much effort on optimization, since it was designed to run on GPU it is by definition "already accelerated by GPUs".

To simulate the particles, Celeritas uses one GPU thread to track one particle. All the chords are simulated one after the other on a kernel and a chord needs one or more RKDP iterations. Figure 2.13 shows the time spent in each action of the simulation.

The team suspects that using more than one thread per track will improve the performance of the simulation. To do that, the threads have to collaborate and this can do the inverse effect and slow down the simulation.

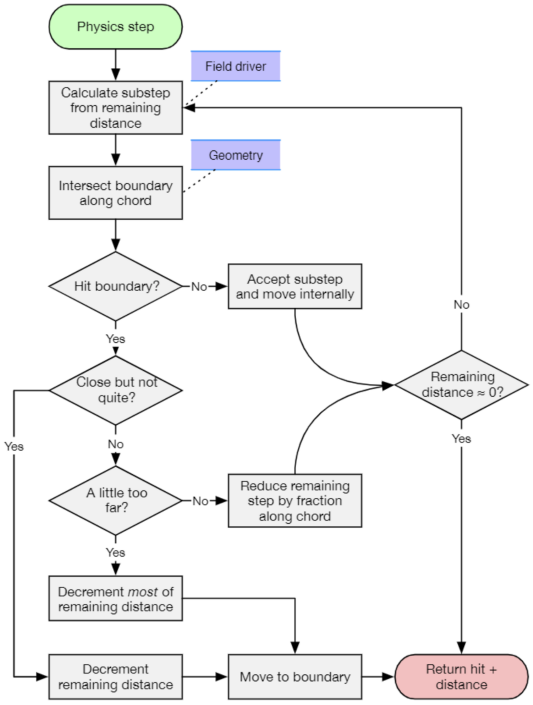

Figure 2.12: Activity diagram of a particle's track [17]

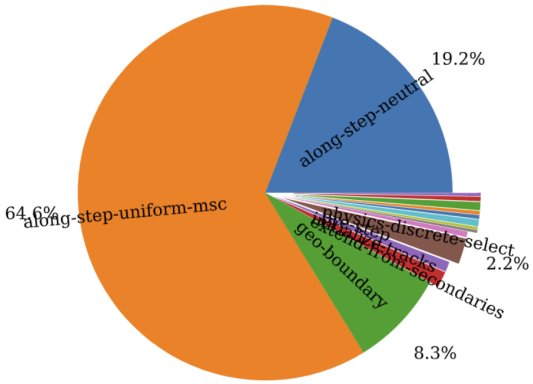

Figure 2.13: Celeritas runtime per action [16]

3 Design

This Bachelor thesis is focusing on optimizing the Runge Kutta Dormand Prince algorithm in a context of GPU computing. The design of the project amounts to finding possible adaptations that could decrease the execution time of the algorithm.

3.1 Simulation

RKDP is a sequential function and as it is described in Chapter 2.1.1, GPUs or parallelization is not so useful in this case. This simulation aims at estimating to what extent using multithreading could potentially improve the running time.

3.1.1 RKDP algorithm

This method has nine stages and is a fifth-order accurate Runge Kutta method. The first six stages are used to compute the end state, the seventh stage is used in the seventh state and the last two stages are used to compute the error and the middle state.

3.1.2 Principle

In the implementation of the RKDP, we can identify different types of code. The first category is the code that must be executed sequentially and cannot be parallelized. The second group of concern is the code that concerns the computation of several expressions that must be assigned before the first reference to the corresponding variables. The last one is a vector multiplication that has a specified place in the code but could be split into parallel substeps.

Figure 3.1 shows how the fourth step is computed and which part of the code is from which type. The red part is the dependent code, that has to be executed sequentially. It is the computation of the intermediate state `kx` using the equation. The blue part is the independent code that just has to be executed before its first use. This is the computation of the coefficients with the step's size. The green part is the vector multiplication that can be executed in parallel. Each line has to be computed sequentially but each involves a vector multiplication which could be multithreaded. Code 3.1 shows the implementation of the method

```
// Fourth step
OdeState k4 = calc_rhs_(state);
state = beg_state;
axpy(a41 * step, k1, &state);
axpy(a42 * step, k2, &state);
axpy(a43 * step, k3, &state);
axpy(a44 * step, k4, &state);
```

Figure 3.1: Example illustrating which part of the code is from which type

Listing 3.1: Implementation of axpy

```
//! \file celeritas/field/Types.hh
/*!
* Perform y <- ax + y for OdeState.
```

```
*/
inline CELER_FUNCTION void axpy(real_type a, OdeState const& x, OdeState* y)
{
    axpy(a, x.pos, &y->pos);
    axpy(a, x.mom, &y->mom);
}
//---------------------------------------------------------------------------//
//! \file celeritas/ext/Convert.geant.hh
/*!
 * Define y += a * x .
 */
inline void axpy(double a, G4ThreeVector const& x, G4ThreeVector* y)
{
    CELER_EXPECT(y);
    for (int i = 0; i < 3; ++i)
    {
        (*y)[i] = a * x[i] + (*y)[i];
    }
}
```

The principle is to record when what type of code is executed to have an idea of the proportion of time spent on each type of code. Code 3.2 shows the same step as Figure 3.1 but with the logger to record the type of code, and written in Python.

Listing 3.2: Fourth step in Python for the simulation

```
# Fourth step
logger(4, TaskType.DEPENDENT)
k4 = equation(*state)
state = beg_state
logger(4, TaskType.INDEPENDENT)
coef_a41 = step * a41
coef_a42 = step * a42
coef_a43 = step * a43
coef_a44 = step * a44
logger(4, TaskType.VECTOR_MULT)
axpy(coef_a41, k1, state)
axpy(coef_a42, k2, state)
axpy(coef_a43, k3, state)
axpy(coef_a44, k4, state)
```

The logger is an object that creates an event with a type and a timestamp. This object is used during the runtime but also after to display the results and to compute the better thread workload using a `SchedulingTree` described in the next section 3.1.3.

3.1.3 SchedulingTree

The `SchedulingTree` has been developed to compute the ideal start time for a given task with the best wanted time.

To have the best scheduling possible, each task has to be executed as soon as possible. To do that, each thread is represented by a `SchedulingTree` and it can return efficiently the best time to start a task. When the best thread is found, the task is added to it.

The implementation is inspired by an interval tree. The tasks are the leaf and every node has the same start as the left child and the same end as the right child. The end is computing with the start time and the duration of the task. This allows the code to know if a task could potentially overlap an existing one. To get the result, the leaves have to be read from the left to the right. The tasks are only stocked in the leaves and the node is only used to know if a new task could potentially overlap an existing one. Figure 3.2 shows an example of a `SchedulingTree` with three tasks. The node help to have a quick overview if a task could overlap and existing or not.

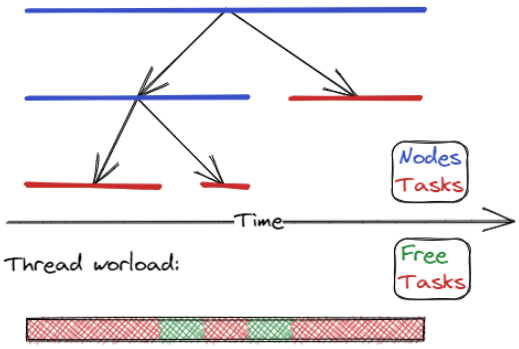

Figure 3.2: Example of a `SchedulingTree` with three tasks

The API of the `SchedulingTree` is shown in Table 3.1. The implementation is not user-bad-usage-proof because indicating an invalid start time in the method `add` will result in the overlapping of some tasks.

Method	Description
`schedule(int wanted_start, Task task) -> int`	Return the earliest start possible after the wanted start given in the parameter
`add(int start, Task task) -> void`	Add a task at the time indicated in the parameter
`get_schedule() -> Task[]`	Return the array of the tasks sorted by start time

Table 3.1: API of the `SchedulingTree`

3.1.4 Results

Those results are obtained by recording the data from a Python script running the RKDP with the chaotic Lorenz attractor equation.

The second graph of Figure 3.4 shows how the time is spent in each RKDP step. This version is without some dead time that appears when recording the events. This is due to the time needed to save the event in the logger. It is important to have this version because when we are simulating the distribution on the threads, we are not taking into account the dead time.

The time of the dependent code is not the same every time but it is always the compute the state using the equation. To explain the first longer time, this is probably due to the first call of the equation that is not in the cache. The last step does not have to create a new state it is just updating the result so it is faster.

Graphs three and four are showing what each thread is doing. We can see that pre-computing the coefficients with the three threads when the first one is computing the equation seems to be a good idea. Also, distributing the vector multiplication could help a lot to decrease the execution time.

The last graph is like the first two ones but with the new repartition of the tasks.

To conclude, this simulation shows that we can expect an improvement in dividing the time up by two. However, this simulation is not taking the time due to the thread synchronization and it is run in a different context than the real implementation.

To implement this solution, the workload has to be divided between the threads. According to the result, the best workload would be something like Figure 3.3. Where a controller is computing the intermediate state using the equation and the other threads are computing the vector multiplication. The role of the threads has been slightly adapted from graph three and four of Figure 3.4.

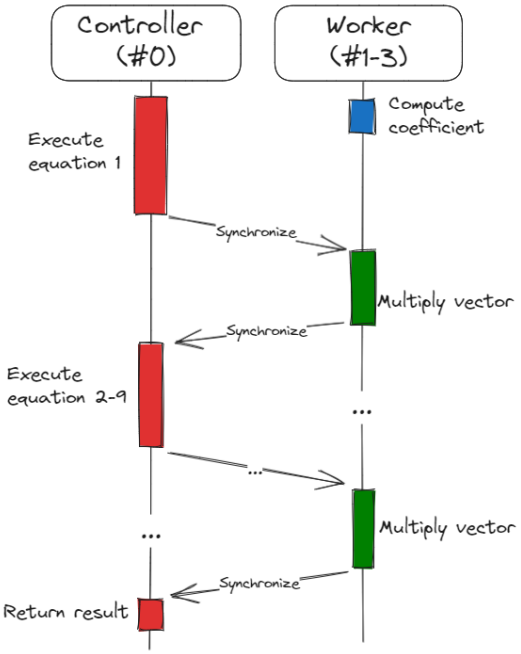

Figure 3.3: Best theoretical distribution of the workload between the threads

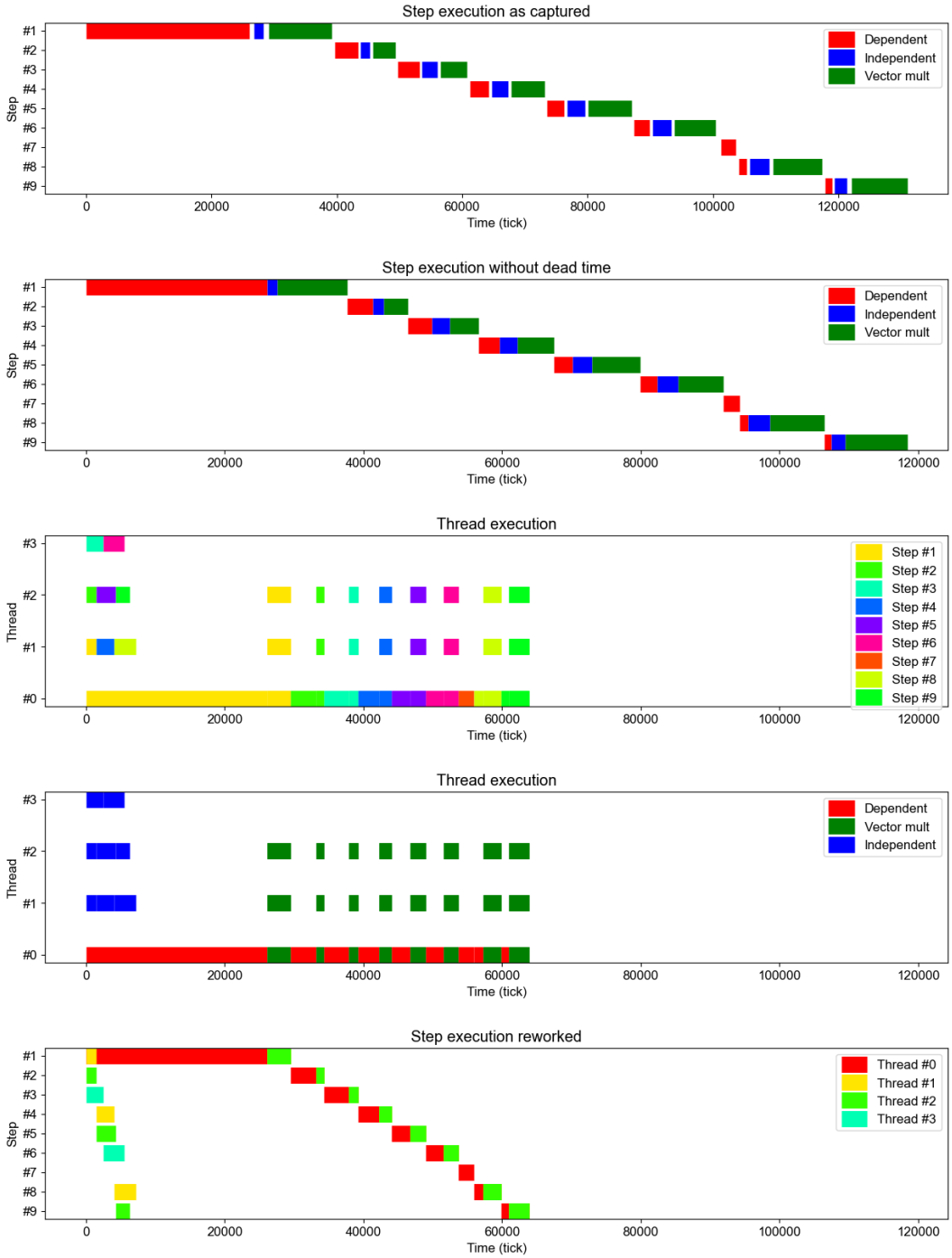

Figure 3.4: Results of the simulation

3.2 Shared memory management

The allocation of the shared memory is a part of some implementations of the new RKDP. To use this memory, the developer must deal with constraints about the size and the alignment of the data.

For each track, the shared memory must contain eight `OdeState` and one `FieldStepperResult`. The `OdeState`s are the seven intermediate states (kx) and the last one is the state that is updated during the whole method (along_state). If the coefficients are pre-computed using multiple threads, they also have to be stored in the shared memory. Table 3.2 shows the size of each object for a track.

Variable	Type	Size (bytes)
ks	OdeState[7]	45 * 7 = 315
along_state	OdeState	45
result	FieldStepperResult	144
(coef)	double[32]	8 * 32 = 256
Total		528 (784)

Table 3.2: Objects stored in the shared memory for each track

According to Table 3.2, the block size has a maximum dimension of 372 threads for the version without precomputed (Equation 3.1) and 248 threads for the version with precomputed coefficients (Equation 3.2).

$$\frac{49152\frac{B}{Block}}{528\frac{B}{Track}} * 4\frac{Thread}{Track} = 93\frac{Track}{Block} * 4\frac{Thread}{Track} = \mathbf{372}\frac{Thread}{Block} \tag{3.1}$$

$$\frac{49152\frac{B}{Block}}{784\frac{B}{Track}} * 4\frac{Thread}{Track} = 62\frac{Track}{Block} * 4\frac{Thread}{Track} = \mathbf{248}\frac{Thread}{Block} \tag{3.2}$$

The limitations of the block size and the shared memory size are dependent on the GPU architecture. The limitations are described in the official documentation [12].

However, the shared memory size is not the only constraint. The shared memory can be allocated statically or dynamically. In this project, the number of tracks is not known at compile time so it is mandatory to use a dynamic allocation. The easiest way to do this is to have an array of different types of objects and use the thread id as an index to access the data as shown in Figure 3.5.

Assuming that the shared memory is a big array of void pointers and knowing the number of tracks of the block, it is possible to create aligned pointers to the different objects. The operator `[x]` on a pointer is returning the pointer to the address `ptr + x * sizeof(type)`. So using `[n]` is giving the first address of the next array.

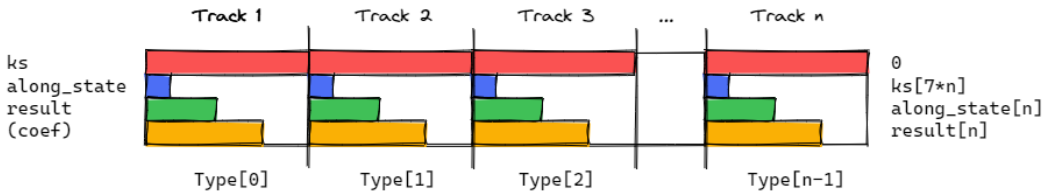

Figure 3.5: Shared memory management

3.3 Number of iterations

As shown in the Chapter 2.3.4 and Figure 2.11, more than 80% of the calls concern just one iteration. However, this does not take into account the time spent to get these results. For example, if a chord needs 100 iterations, it has taken 100 times more time than a chord that needs only one iteration. To have a better view of the effort that takes an iteration, the data have been multiplied by the number of iterations. The result is shown in Figure 3.6.

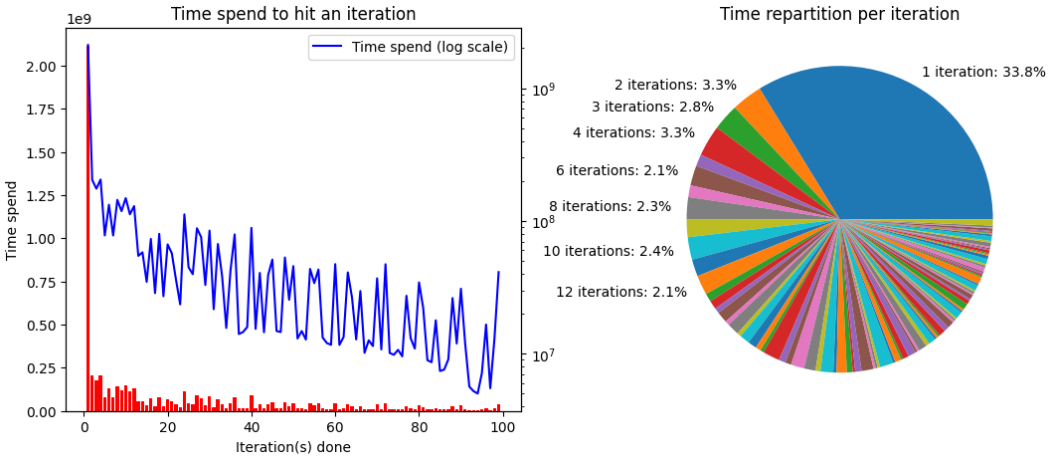

Figure 3.6: Time before an iteration is hit multiplied by the number of iterations to have an idea of the time spent in each iteration.

As we can see, the first iteration is still the most used. However, there is a better distribution and we can't assume that doing only one iteration during the test is representative of the real use of the algorithm.

Based on the heuristic rule of the 80/20, a good number of iterations will be a number that represents 80% of the time. Figure 3.7 shows that this number is 40 iterations. To test RKDP, the method will be called 40 times for every particle.

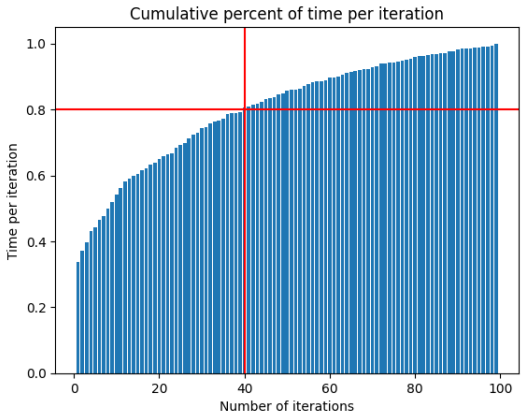

Figure 3.7: Cumulative percentage of the time spent in the iterations with a marker at 80%.

4 Implementation

The implementation of this Bachelor thesis is not done from scratch, it is the improving the Celeritas project. My contribution is detailed in a pull request available here [18].

4.1 Project initialization

Before actively developing the project, it is necessary to set up the environment, compile the project and run it to ensure that everything is working as expected.

4.1.1 Set up the environment

This project is designed to be run on HPC. This means that the tools are slightly different from a regular project. Those systems are designed to be used by multiple users at the same time. Therefore, they have to be adapted to any usage and provide the needed tools or libraries that the users want. To help the users, the system provides a module system or an advanced package manager like `spack` [19].

Module system

Module [20] is a tool that allows the user to load or unload some tools or libraries. It is a simple way to manage the environment and it must be installed and managed by the system administrator.

Table 4.1 shows some commands that can be used with the module system. They are basic but they are sufficient in most cases.

Command	Description
`module avail`	List all available module
`module list`	List all loaded module
`module load <module>`	Load a module
`module unload <module>`	Unload a module

Table 4.1: List of useful modules commands

The command can be added to the profile to load a module when a session is open, for example, `module load git`.

Spack

Spack [19] is an advanced package manager designed for HPC applications. This tool can be installed by the user and it helps to install some projects like Celeritas with their dependencies. Like module, spack allows the user to work on many different projects with the system of the environment.

Install this tool, it requires to clone the repository and source it. As sourcing a script is shell session scoped, it is recommended to add it to the profile. Code 4.1 shows

Listing 4.1: Download spack and source it

```
git clone\
    --depth=100\
```

```
    --branch=releases/v0.20 https://github.com/spack/spack.git\
    ~/spack

echo ". ~/spack/share/spack/setup-env.sh" >> ~/.bashrc
```

To reload the profile, use the command `source /.bashrc`.

Command	Description
`spack env create <name> <path>`	Create a new environment based on a YAML file that describes the environment
`spack env activate <name>`	Activate an environment
`spack env deactivate`	Deactivate the current environment
`spack env list`	List all environments
`spack env remove <name>`	Remove an environment
`spack concretize -f`	Show what will be installed with which option. This command is very useful to check that spack will install Celeritas with the GPU acceleration
`spack install`	Install the environment
`spack find`	List all installed packages
`spack config add package:all:variants:"+/-option"`	Add a variant to all packages

Table 4.2: List of useful spack commands

Celeritas environment

As shown in the previous sections, the environment is something important and it needs a bit of a configuration to have the right one. Code 4.2 is the profile used to develop the project on the host Zeus. It assumes that the environment `celeritas` has already been created with spack.

Listing 4.2: Profile to work with Celeritas

```
# Load spack
. ~/spack/share/spack/setup-env.sh
# Load the right compiler (Zeus)
module unload gcc
source /cvmfs/sft.cern.ch/lcg/contrib/gcc/11.3.0/x86_64-centos7-gcc11-opt/setup.sh
# Activate the environment (Optional)
spack env activate celeritas
```

4.1.2 Compile the project

Celeritas is a complex project with multiple builds possible. The easiest way to compile it is to use the script provided by the team here: `./scripts/build.sh`. This script will automatically read the host to load the right compiler, the right modules and the

right Cmake profile. In the end, this script will build all the applications issued from this project and run the tests. This script is loading the Cmake preset and his name must be given as an argument. Code 4.3 lists all the available presets.

Listing 4.3: Cmake preset for Celeritas

```
Sourcing environment script at scripts/env/zeus.sh
loading cuda/11.8.0
Usage: scripts/build.sh PRESET [config_args...]
Available configure preses:

    "base"        - Zeus default options (GCC, debug)
    "reldeb-novg" - Zeus release with debug symbols and Orange
    "reldeb"      - Zeus release with debug symbols
    "ndebug-novg" - Zeus release with Orange
    "ndebug"      - Zeus release
    "default"     - Automatic options (debug with tests)
    "full"        - Full
    "minimal"     - Minimal
```

If this is the first time the project is built, some pre-steps are required. To start, the right compiler must be loaded. Then, the spack environment must be created. Code 4.4 shows those two parts.

Listing 4.4: Pre-steps to compile Celeritas

```
## Load the right compiler (Zeus) ##
module load cmake
module unload gcc
source /cvmfs/sft.cern.ch/lcg/contrib/gcc/11.3.0/x86_64-centos7-gcc11-opt/setup.sh

## Create the environment ##
# Configure the project to be used with CUDA
spack config add packages:all:variants:"+cuda cuda_arch=80"
spack env create celeritas scripts/spack.yaml
spack activate celeritas
```

After this step, it just needs to install the environment using `spack install` (can be parallelized with `spack install & spack install & ...`) but this step takes a couple of hours. To avoid doing that multiple times, it is recommended to use the command `spack concretize -f` to check what environment will be installed. The important thing is to have `vecgeom` compiled with `gcc 11.3` and have the variant `cuda` enabled with the architecture `80` for Zeus and `70` for Perlmutter. During this Bachelor thesis, the version of Geant4 [4] used was the 11.1 or more. Those requirements are shown in Code 4.5.

Listing 4.5: Check of the environement

```
vecgeom@1.2.2%gcc@11.3.0+cuda+gdml~geant4~ipo~root+shared
    build_system=cmake build_type=Release cuda_arch=80 cxxstd=17
    generator=make arch=linux-centos7-cascadelake

geant4@11.1.1%gcc@11.3.0~ipo~motif~opengl~qt~tbb+threads~vecgeom~vtk~x11
    build_system=cmake build_type=Release cxxstd=17 generator=make
    arch=linux-centos7-cascadelake
```

4.1.3 Run a job on Perlmutter

Perlmutter [7] is the supercomputer of the NERSC. It is a very powerful machine with a lot of GPU and CPU. To run a job on this machine, it is not like a regular computer. The user must submit a job to the scheduler and wait for the job to be executed. To submit a job, the user must create a script that will be executed by the scheduler. Code 4.6 shows an example of a job script. This script is a `sbahs` and it defines the required resources for the job.

Listing 4.6: Job script for Perlmutter

```
#!/bin/bash
#SBATCH -q debug
#SBATCH -N 1
#SBATCH -C cpu
#SBATCH -t 00:10:00
#SBATCH -J my_job

./serial-hello
srun -n 64 -c 4 --cpu-bind=cores ./mpi-hello
```

The arguments are well described in the official documentation (`https://docs.nersc.gov/jobs/#commonly-used-options`).

To launch a job on Perlmutter, use the following checklist to ensure to don't forget anything:

1. Load modules needed with the command `module load <module>`
2. Build the application that will be executed
3. Write the job script as shown in Code 4.6
4. Run the script with the command `sbatch <script>`
5. Monitor the status of the job with the command `squeue -u <username>` Possibility of using `watch` to refresh the status automatically.
6. Prompt the result or the error from the file `slurm-<jobid>.out/.err`

4.1.4 Run and profile the project

The tests are automatically launched after the build. However, we can want to run the application to do a demonstration or to monitor the runtime. In this thesis case, the application has to be profiled to know if the changes improved the performance.

Celeritas needs complex input files to run and as the development has been done on the Zeus, a computer with GPUs in the LBNL. The easiest way to run `celer-sim`, the application to simulate the path of the particles is to use a Python script. This script has been adapted into this pull request [21].

The script `run-problems.py` is a Python script that builds the input files and run the application. To launch this script, there is a bash script to set up the environment.

This step has taken a lot of time at the beginning of the project to have a profile that aims to be used to compare with the final version. Actually, Celeritas is a wide project and it is not easy to integrate some changes when it touches the way the kernels are launched. As the changes are not integrated into the simulation, no profile can be done to compare with the first one. However, a profile of the tests that launch some kernel with the old and the new implementations can be done easily.

4.2 Test framework

As Celeritas is a complex project, all the architecture changes to use multiple threads per track are not in the scope of this Bachelor thesis.

To avoid this problem, the implementation is used and compared in tests. These tests are launched automatically after the build but it is possible to only launch the wanted test file. Code 4.7 shows how to build the project without launching the tests and then launching the file that contains the tests about the performance of RKDP.

Listing 4.7: Launch a test

```
./scripts/build.sh base --no-test
./build/test/celeritas_field_DormandPrinceStepper
```

It is possible to use the test file with `ncu` to profile the kernels.

4.2.1 Launching RKDP

There are three test files, `DormandPrinceStepper.test.hh`, `DormandPrinceStepper.test.cc` and `DormandPrinceStepper.test.cu` (files available in the pull request [18]). The first one is the header file, it contains the helper methods and all the constants used in the tests. The second file is the file that contains the tests and the last one is the implementation of the kernels.

The method `simulate_multi_next_chord` of the file `DormandPrinceStepper.test.cu` is the method to test implementation. This method requires the number of threads per track, the number of states and if it has to use the shared memory with a boolean. These parameters help to choose the implementation to run. Table 4.3 explains how the implementation is chosen.

number_threads	use_shared	Implementation
1	false	Old implementation
4	false	Global memory
4	true	Shared memory

Table 4.3: Implementation chosen in function of the parameters

Figure 4.1 describes the steps of the method `simulate_multi_next_chord`. Points 2, 3 and 10 have more variables to manage if the implementation is the one with global memory. The compute of the properties changes for every implementation because the number of tracks per block is different.

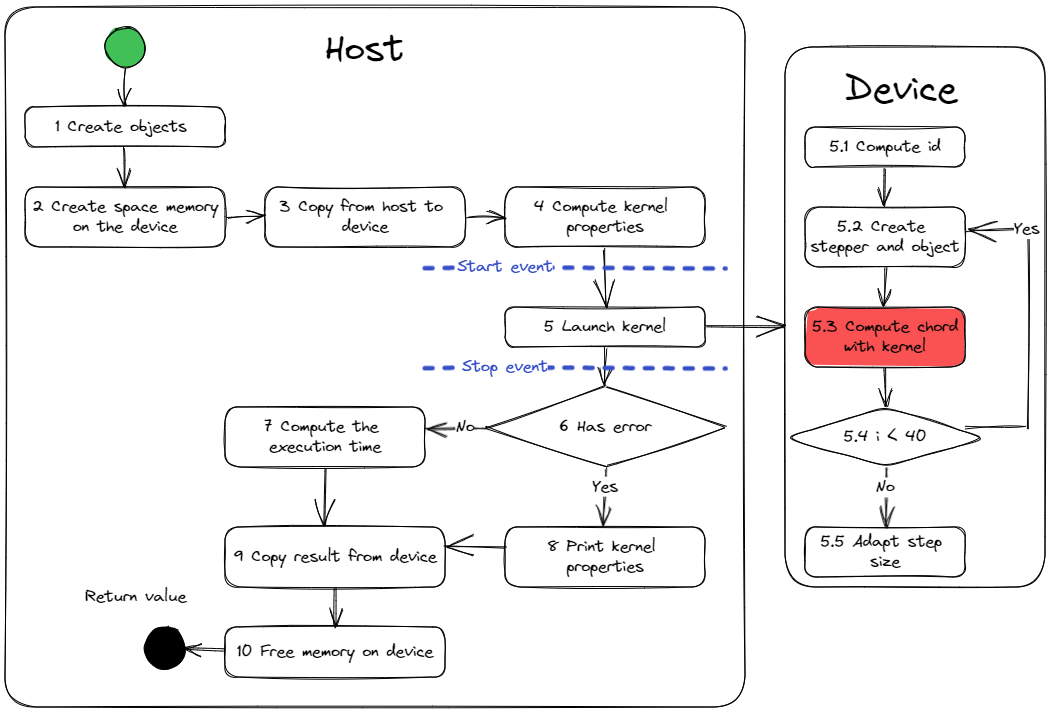

Figure 4.1: Activity diagram to show how a kernel that tests an implementation is launched

To know the execution time of the kernel, CUDA provides an API to create and record events. These objects are used to measure the time between two events. Before recording the end event, the kernel has to be synchronized to ensure that the kernel is finished.

4.2.2 Tests

There are three types of tests in the file `DormandPrinceStepper.test.cc`. The first one is here to check that the results are correct. It launches the old implementation and compares all results with the ones from a new implementation. The name of the tests start with `result_` and the suffix is the name of the implementation. The second type of test is here to check that the new implementation is at least as fast as the old one. The test has `time_` as a prefix and the suffix is the name of the implementation. The last type is here to compare the implementation between them. This kind of test is disabled with the prefix `DISABLED_` because they contain no assertion. It also contains the part `compare` in the name to be easily identified. Code 4.8 shows the result of the tests and the name of the tests.

Listing 4.8: Results of the tests

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from DormandPrinceStepperTest
[ RUN      ] DormandPrinceStepperTest.result_global_memory
[       OK ] DormandPrinceStepperTest.result_global_memory (74 ms)
[ RUN      ] DormandPrinceStepperTest.time_global_memory
[       OK ] DormandPrinceStepperTest.time_global_memory (67 ms)
[ RUN      ] DormandPrinceStepperTest.result_hared_memory
```

```
[       OK ] DormandPrinceStepperTest.result_hared_memory (42 ms)
[ RUN      ] DormandPrinceStepperTest.time_shared_memory
[       OK ] DormandPrinceStepperTest.time_shared_memory (67 ms)
[ DISABLED ] DormandPrinceStepperTest.DISABLED_compare_time_one_by_one
[----------] 4 tests from DormandPrinceStepperTest (253 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (253 ms total)
[  PASSED  ] 4 tests.
```

4.3 Implementation of the RKDP algorithm

All the implementation can be found in the fork [22] of Celeritas on the profile of Simon Barras. The commits related to an implementation have been tagged with the prefix `rkdp-opti-v0.x` and the number of the implementation. This version has been tested with the framework described in the previous section 4.2.

All the tests have been done on the host Zeus and the parameters used are described in Table 4.4. The parameters are the same for all the implementations to have a better comparison. The only difference is the number of threads per track for the old implementation and the new ones. Each implementation is tested with one, forty and one hundred iterations to see the impact of the number of iterations on the performance.

Parameter	**Value**
Number of tracks	5
Number of threads per track	old: 1, new: 4
Initial step size	10000
Delta chord	1e-4
Number of iterations	1, 40 and 100

Table 4.4: Parameter used to run the tests

4.3.1 Global memory and vector multiplication

This first version implemented is a simple implementation that uses the global memory and vector multiplication. This implementation is tagged with `rkdp-opti-v0.1`.

To implement a version with more than one thread per track, the threads must have indicators to know which state they have and which role. These indicators are the `id` which represents the track and the `index` which is an integer from 1 to the number of threads per track. The threads grouped around the same track have to wait for the others so it is important to have a `mask` that is used to synchronize the threads. Code 4.9 shows how the indicators are computed, the `number_threads` is the number of threads per track. How the role of each thread is affected is also shown in the code. The advantage of computing the id with the division and not with the modulo is that the threads assigned to the same track are contiguous in the grid.

Listing 4.9: How the `id`, `index` and `mask` are computed

```
int id = (threadIdx.x + blockIdx.x * blockDim.x) / number_threads;
int index = (threadIdx.x + blockIdx.x * blockDim.x) % number_threads;
constexpr int warp_size = 32;
```

```
int mask = (number_threads * number_threads - 1)
           << ((id * number_threads) % warp_size);
if (index == 0)
{
    run_sequential(step, beg_state, id, mask, ks, along_state, result);
}
else
{
    run_aside(step, beg_state, id, index, mask, ks, along_state, result);
}
return *result;
```

This implementation needs to have the intermediate states (`ks`) and the state that is updated during the function (`along_state`). To have the same pointer for all the threads, these variables must be passed as a parameter to the kernel.

This simple version only needs the to use the instruction `__syncwarp(mask)` to synchronize the threads. This instruction surrounds the vector multiplication and is doubled between each step. Code 4.10 shows the new implementation of Code 4.11.

Listing 4.10: Usage of the instruction `__syncwarp(mask)`

```
// Main thread
// First step
ks[0] = calc_rhs_(beg_state);
*along_state = beg_state;
__syncwarp(mask);
__syncwarp(mask);
//------------------------------------------------------------------------//
// Aside threads
__syncwarp(mask);
along_state->pos[index - 1] = step * axx[coef_counter]
                                * ks[j].pos[index - 1]
                                + along_state->pos[index - 1];
along_state->mom[index - 1] = step * axx[coef_counter]
                                * ks[j].mom[index - 1]
                                + along_state->mom[index - 1];
__syncwarp(mask);
```

Listing 4.11: Legacy implementation of the RKDP algorithm

```
// First step
OdeState k1 = calc_rhs_(beg_state);
OdeState state = beg_state;
for (int i = 0; i < 3; ++i)
{
    state.pos[i] = a11 * step * k1.pos[i] + state.pos[i];
}
for (int i = 0; i < 3; ++i)
{
    state.mom[i] = a11 * step * k1.mom[i] + state.mom[i];
}
```

Figure 4.2 shows the sequence of the communication between the threads.

With the parameters described in Table 4.4, Table 4.5 shows the results of the tests for each number of iterations.

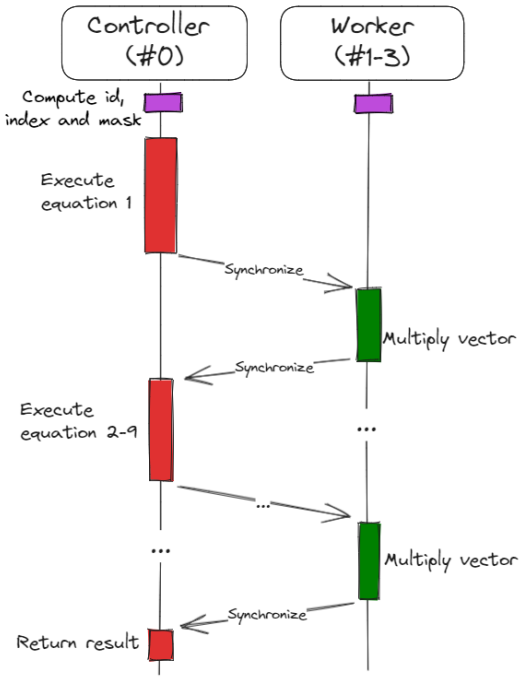

Figure 4.2: Sequence diagram to show the first implementation of RKDP

Number of iterations	Old (ms)	Current (ms)	Speedup
1	1.07725	0.67904	158.64%
40	41.6737	26.071	159.85%
100	104.387	65.8627	158.49%

Table 4.5: Results of the tests for the global memory and vector multiplication

4.3.2 Global memory, vector multiplication and pre-computation

This implementation is based on the previous one 4.3.1. The tag is `rkdp-opti-v0.2` and the difference is the pre-computation of the coefficients.

All the indicators, the synchronization and anything else that is related to the vector multiplication is the same as the previous implementation.

The pre-computation is done by every worker thread for every coefficient. The coefficients are constants but they are scaled by the step size. Figure 4.3 shows this difference with the task in blue.

This version is slightly slower than the previous one as shown in Table 4.6. This version can be improved by distributing the pre-computation between the threads but this version implies more effort and the version with the shared memory has more potential speedup.

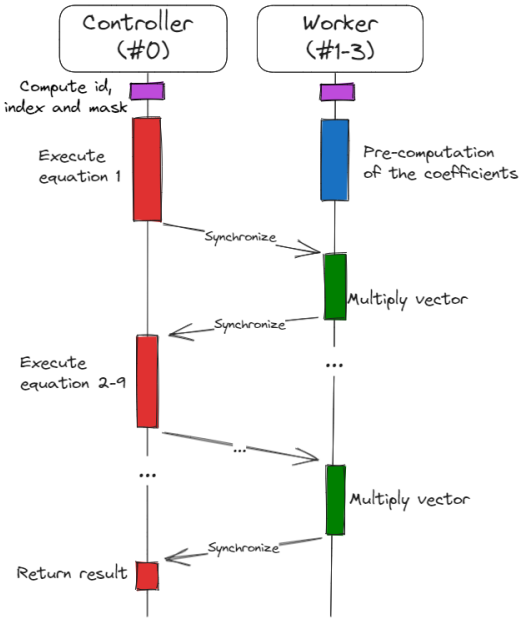

Figure 4.3: Sequence diagram to show the second implementation of RKDP

Number of iterations	Old (ms)	Current (ms)	Speedup
1	1.07725	0.693248	155.39%
40	41.6768	26.3782	158.00%
100	104.389	66.4351	157.13%

Table 4.6: Results of the tests for the global memory, vector multiplication and pre-computation

4.3.3 Shared memory and vector multiplication

This implementation is based on the first one 4.3.1 and it is tagged with `rkdp-opti-v0.3`.

As explained in the design about the shared memory 3.2, it is used as a space to store the objects that will be used by the threads. Code 4.12 shows how the intermediate states, the along state and the result are stored and declared.

Listing 4.12: Declaration of the shared memory

```
extern __shared__ void* shared_memory[];
OdeState* shared_ks = (OdeState*)shared_memory;
OdeState* shared_along_state
    = reinterpret_cast<OdeState*>(&shared_ks[7 * number_states]);
FieldStepperResult* shared_result = reinterpret_cast<FieldStepperResult*>(
    &shared_along_state[number_states]);
```

The implementation of `run_sequential` and `run_aside` is similar to the version with the global memory as long as the call of this method is a bit different as shown in Code 4.13.

Listing 4.13: Call of the method `run_sequential` and `run_aside`

```
if (index == 0)
{
    run_sequential(step,
                   beg_state,
                   id,
                   mask,
                   &shared_ks[7 * id],
                   &shared_along_state[id],
                   &shared_result[id]);
}
else
{
    run_aside(step,
              beg_state,
              id,
              index,
              mask,
              &shared_ks[7 * id],
              &shared_along_state[id],
              &shared_result[id]);
}
```

The call of the kernel changed because the shared memory is dynamically allocated, the size to be given as the third property of the kernel. The size per track is described in Chapter 3.2.

Figure 4.4 shows that all threads have to declare the shared memory with the indicators. This time is probably longer than the version without the shared memory.

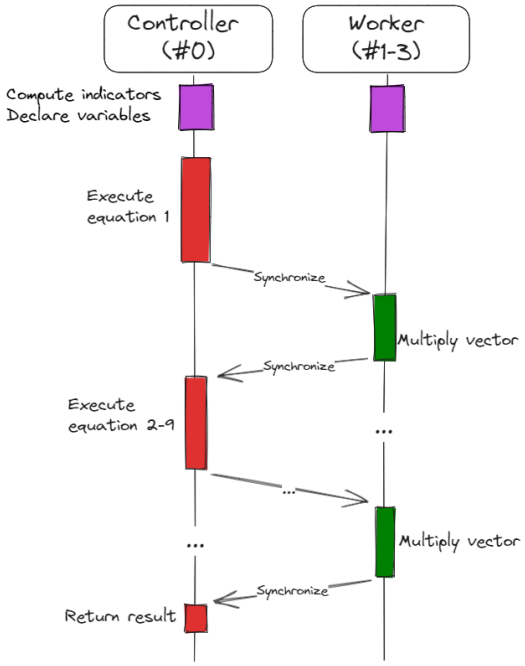

Figure 4.4: Sequence diagram to show the third implementation of RKDP

The result is slightly similar to the first implementation. Table 4.7 shows the results for the three numbers of iterations.

Number of iterations	Old (ms)	Current (ms)	Speedup
1	1.07315	0.677952	158.29%
40	41.5314	26.0813	159.24%
100	104.065	65.8852	157.95%

Table 4.7: Results of the tests for the shared memory and vector multiplication

4.3.4 Shared memory, vector multiplication and pre-computation

As mentioned in the hypothesis of the simulation 3.1, the pre-computation of the coefficients can improve the performance. The implementation number 2 4.3.2, as shown that if all the worker threads precompute all the coefficients, the performance is worse. The goal of this implementation which is tagged with `rkdp-opti-v0.4` is to show that if the pre-computation is distributed will improve the performance.

Figure 4.5 shows that the worker threads have to declare more shared variables to store the precomputed coefficients. Declaring the coefficients in the shared memory increase its size of it and it affects the number of threads per block.

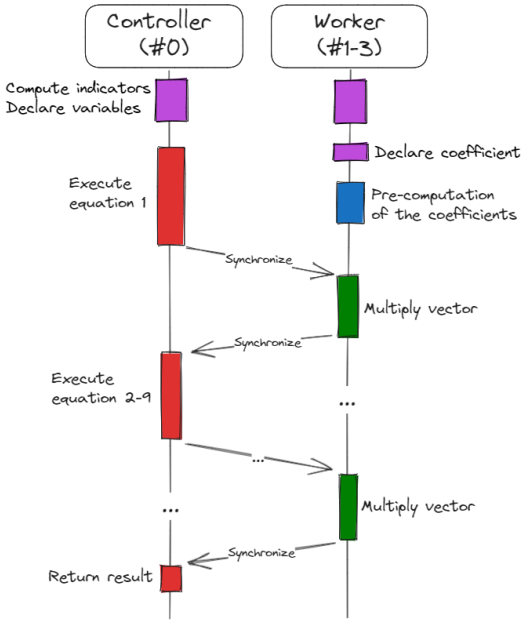

Figure 4.5: Sequence diagram to show the fourth implementation of RKDP

The result in Table 4.8 are similar to the second implementation. The pre-computation seems to take too much time before the first vector multiplication.

4.3.5 Shared memory, vector multiplication and pre-computation optimized

The hypothesis behind this implementation is that the pre-computation of the coefficients takes more time than the first step of the sequential part. This implementation is tagged `rkdp-opti-v0.5` and the pre-computation is distributed between the different vector multiplication.

Number of iterations	Old (ms)	Current (ms)	Speedup
1	1.07213	0.708608	151.30%
40	41.5304	26.5051	156.69%
100	104.02	66.4965	156.43%

Table 4.8: Results of the tests for the shared memory, vector multiplication and pre-computation

As there are three workers threads, the compute three coefficients take the same time as only compute one or two coefficients. Table 4.9 shows that just by completing three coefficients before a vector multiplication is enough to have enough coefficients for the five first steps. Before step 6, the worker threads have to compute another three next coefficients and before the step eight and nine, the worker threads have to compute the last coefficients. Step 7 is not shown here because the coefficients are not used.

Step	Pre-computed	Needed
1	3	1
2	6	3
3	9	6
4	12	10
5	15	15
6	21	21
8	32	32
9	32	32

Table 4.9: Results of the tests for the shared memory and vector multiplication

Figure 4.6 illustrate the sequence of the tasks made by the threads. This is very similar as the previous implementation 4.3.4 but the pre-computation is distributed between the vector multiplication.

Surprisingly, the results in Table 4.10 are not much better than the previous implementation. The hypothesis is not correct and it shows that the pre-computation is not benefic for the performances. This is probably because the time to store the value and to read it after is longer than the time to compute the coefficients and pass them directly to the methods.

4.3.6 Conclusion

In chapter 3.1 about the simulation, the hypothesis is that dispatching the vector multiplication and pre-compute the coefficients improve the performance. This hypothesis is partially correct because the pre-computation is not beneficial for the performance but the vector multiplication is. The Figure 4.7 shows the speedup of the different implementations.

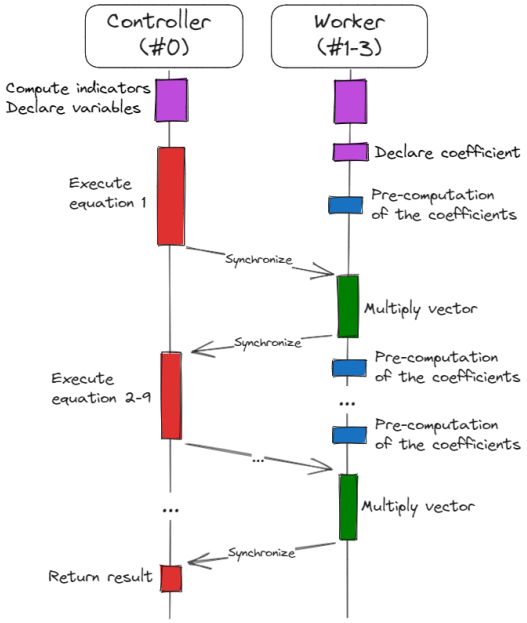

Figure 4.6: Sequence diagram to show the fifth implementation of RKDP

Number of iterations	Old (ms)	Current (ms)	Speedup
1	1.07315	0.712704	150.57%
40	41.5283	26.5257	156.56%
100	104.018	66.646	156.08%

Table 4.10: Results of the tests for the shared memory, vector multiplication and the optimized pre-computation

The conclusion that appears from this graph is that the vector multiplication provides some good results. Now, the pre-computation seems to slow down the performance even if the implementation is optimized. The experiments do not show that the usage of the shared memory is better than the global memory. To know which implementation is better, further experiments are made using the implementation number one (global memory and vector multiplication) and the implementation number 3 (shared memory and vector multiplication).

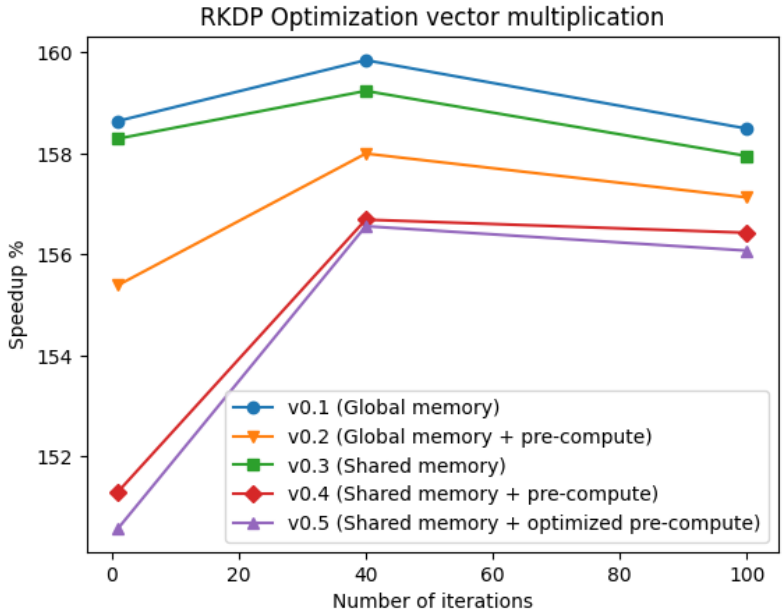

Figure 4.7: Graph to show the speedup of the different implementations

5 Result

The result of this Bachelor thesis is available on the pull request [18]. As explained in the chapter 4, five different implementations have been made and tested but only two of them have been kept. The implementation with the global memory and the shared memory have shown the best results but the tests can be described as "shy" because the number of tracks is low. On the pull request, there are two implementations and the tests have been developed to compare the two implementations.

To compare the three implementations (old, global memory and shared memory), the Table 5.1 shows the parameters used for the comparison. The number of states is not defined because it is the parameters that are changed to compare the three implementations. The number of iterations has been set to 40 according to this graph 3.7.

Parameter	**Value**
Number of states	X
Number of threads per track	old: 1, new: 4
Initial step size	10000
Delta chord	1e-4
Number of iterations	40

Table 5.1: Parameters used for the comparison

5.1 Block limit

The number of threads per block is limited by the GPU architecture. Most of the GPUs have a limit of 1024 threads per block but that is not the only limit. The number of registers used per thread during the execution or the shared memory can get down the limitation of the number of threads per block. This limitation cannot just be applied like this because the implementation requires that the threads that work on the same track are in the same warp. To do that, the number of threads per block must be a multiple of 32. Table 5.2 shows the maximum number of threads per block for the three implementations. The number of threads per block for the shared memory implementation is limited by the shared memory and it is described in Chapter 3.2.

Implementation	**Threads**	**Warps**	**Effective threads**	**Tracks**
Old	772	24	768	768
Global memory	772	24	768	192
Shared memory	372	11	352	88

Table 5.2: Max number of threads per block

The runtime for the three implementations using one block is shown in Figure 5.1. Overall, the performance is good because the speedup is from 135% to 160%.

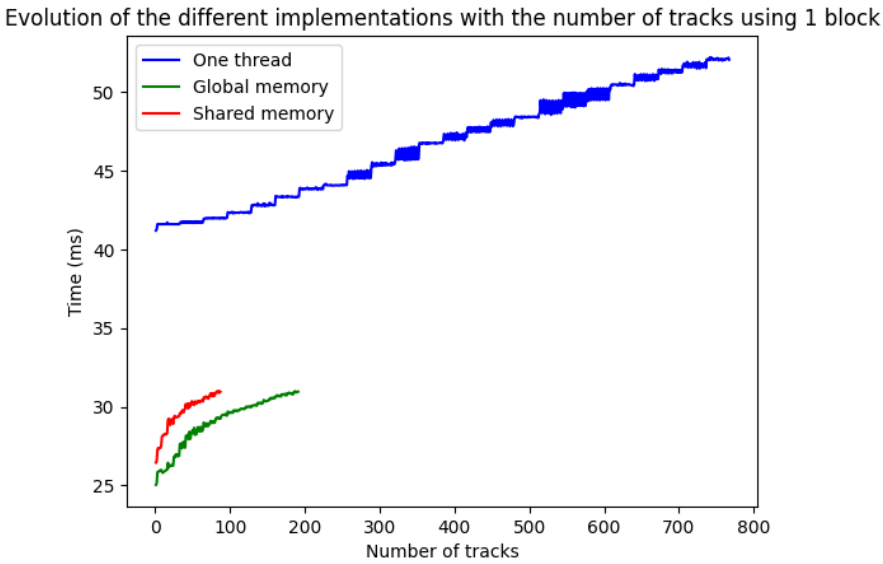

Figure 5.1: Runtime for one block

5.2 Runtime

The goal of this comparison is to prove that the new implementation is faster than the old one for the same context as a classic Celeritas runtime. To do that, the number of tracks will be highly increased to show if the performance at one million tracks is better than the old implementation.

The experiments shown in Figure 5.6 has run the three implementations with a number of tracks from 100 to 276'900 every 100 tracks. It has been made on Zeus with an Nvidia A6000 [23].

5.2.1 Performance

The result for the new implementations starts with a better performance than the old one but the cost to add more tracks is higher than the old implementation. With around 15'000 tracks, the old implementation becomes faster than the new ones.

An interesting observation is that when the time of the new implementations becomes slower, the performance does a "jump". Just by adding 100 tracks, the runtime increase by 25 milliseconds which is around 140% of the runtime at this moment. This jump is also visible on the old implementation with the same proportion but with more tracks. The repairs in Figure 5.6 are showing that this jump appears when the number of threads is around 65'000. Another interesting observation is that this step is continuing with less and less difference.

5.2.2 GPU limits

To explain these jump, Figure 5.2 displays the runtime with the number of threads and not the number of tracks. It shows that the old implementation is slower for a given a number of threads but keep in mind that the old implementation is tracking four times

more particles. The frequency has been set to 1 every 64'512 threads which is the jump at 65'280 minus the size of a block (720 threads) because we want the start of the jump.

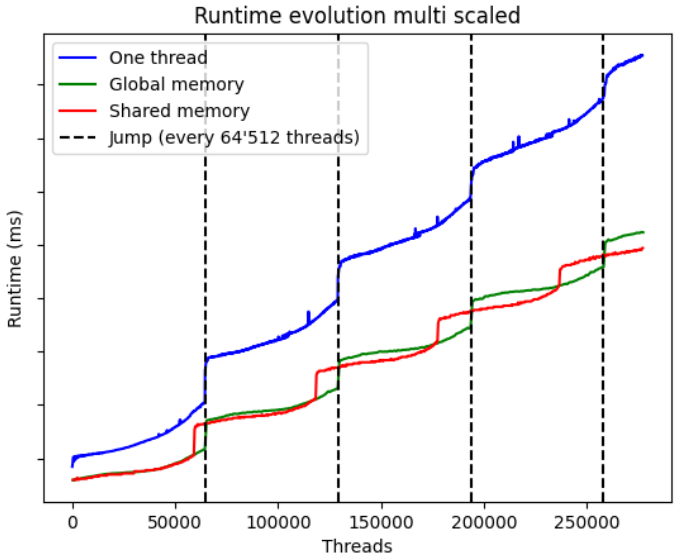

Figure 5.2: Runtime on the number of threads

This number of 64'512 threads seems to fit perfectly into the old implementation with the global memory. However, the new implementation with the shared memory seems to have a higher frequency. This observation can also be made looking Figure 5.6 when the number of threads at the jump was lower. As the number of threads per block is the same for the first two implementations, it is possible that the number of blocks has a higher impact than the number of threads.

Figure 5.3 is basically the same graph as previously but with the number of blocks instead of the number of threads. The frequency has been set to 84 which is the number of SM on the Nvidia A6000 [23] (the graphic card used for the tests).

Now, every jump appears on a marker, but not at every marker for the implementation with the shared memory. This difference is due to the number of blocks per SM which is two for the shared memory implementation and one for the global memory implementation and the old implementation.

As the resources available on a GPU are limited, the number of threads that run at the same time is limited. If this number is exceeded, the GPU will have more rounds to do to complete the work. These rounds are called `waves` and Equation 5.1 shows how to calculate the number of waves.

$$\text{Waves} = \frac{\text{Launched blocks}}{\text{Number of SM}} * \frac{\text{Blocks}}{\text{SM}} \tag{5.1}$$

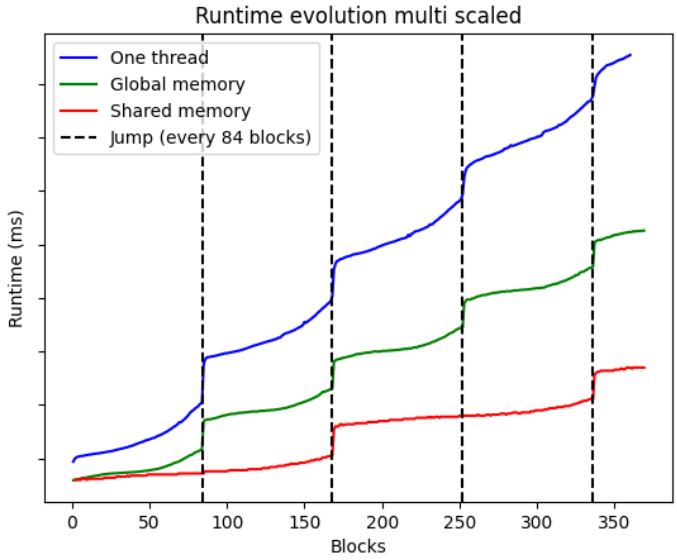

Figure 5.3: Runtime on the number of blocks

To understand the number block per SM, it is important to know how the blocks are executed. The GPU assigns a block to a SM and it manages its own resources to execute the block. Sometimes, the blocks are not too gourmet and more than one block can fit on a SM.

According to the previous section, the problem with the new implementation is that they require too many resources and the number of waves is increasing. The number of waves is the cause of the jump in the runtime that costs a lot of performance.

5.2.3 Profiling

The hypothesis of the GPU limit is confirmed by the profiling of the three implementations. Nvidia Nsight Compute [24] has been used to profile the three implementations using the test `compare_occupancy`.

First of all, the graphs on Figure 5.4, represent how to optimize the kernel occupancy. Graph number 2 shows that the highest number of threads per block is 1024 for the GPU Nvidia A6000 [23] but the limitation of the kernel is 768. The second graph shows the maximum size of the shared memory per block which is 48 KB.

To confirm the hypothesis of the waves per SM, the table on Figure 5.5 show the kernel configuration and the relevant information is the number of waves per SM. As the kernel time is also given by the tool, it confirms that every jump is relating to a number of waves that is not an integer number. This number increase by one every 84 blocks for the old and the global memory version and every two 84 blocks for the shared memory version.

To read the profile [25], use Table 5.3 to know what the context of the kernel launched. The context is the number of tracks and the implementation used.

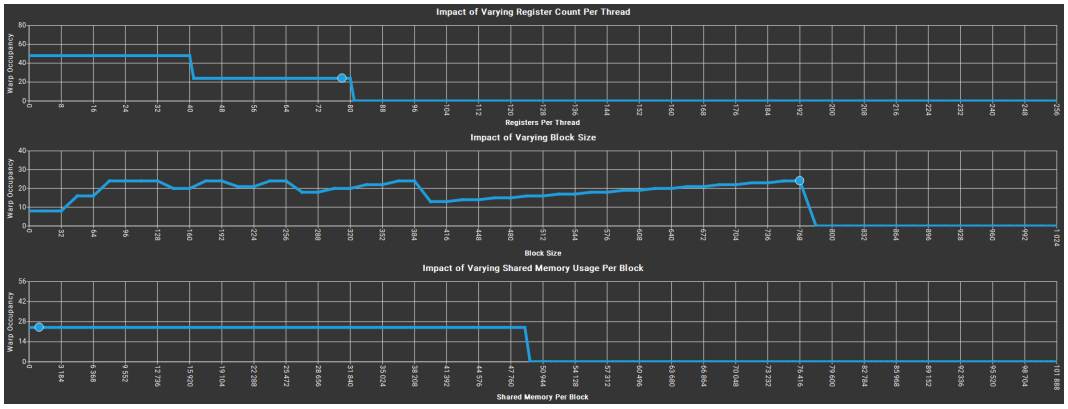

Figure 5.4: Graphs that show the limitation of the GPU for the old version but it is the same for the version with the global memory

Grid Size	84
Registers Per Thread [register/thread]	78
Block Size	768
Threads [thread]	64 512
Waves Per SM	1

Figure 5.5: Table that shows the waves per SM

#	Implementation	Tracks
0	Old	14500
3	Old	14600
6	Old	14700
9	Old	14800
12	Old	14900
15	Old	15000
18	Old	15100
21	Old	15200
24	Old	15300
27	Old	15400
30	Old	15500
33	Old	15600
36	Old	15700
39	Old	15800
42	Old	15900
45	Old	16000
48	Old	16100
51	Old	16200
54	Old	16300
57	Old	16400
60	Old	16500
63	Old	64000
66	Old	64100
69	Old	64200
72	Old	64300
75	Old	64400
78	Old	64500
81	Old	64600
84	Old	64700
87	Old	64800
90	Old	64900
93	Old	65000

#	Implementation	Tracks
1	Global memory	14500
4	Global memory	14600
7	Global memory	14700
10	Global memory	14800
13	Global memory	14900
16	Global memory	15000
19	Global memory	15100
22	Global memory	15200
25	Global memory	15300
28	Global memory	15400
31	Global memory	15500
34	Global memory	15600
37	Global memory	15700
40	Global memory	15800
43	Global memory	15900
46	Global memory	16000
49	Global memory	16100
52	Global memory	16200
55	Global memory	16300
58	Global memory	16400
61	Global memory	16500
64	Global memory	64000
67	Global memory	64100
70	Global memory	64200
73	Global memory	64300
76	Global memory	64400
79	Global memory	64500
82	Global memory	64600
85	Global memory	64700
88	Global memory	64800
91	Global memory	64900
94	Global memory	65000

#	Implementation	Tracks
2	Shared memory	14500
5	Shared memory	14600
8	Shared memory	14700
11	Shared memory	14800
14	Shared memory	14900
17	Shared memory	15000
20	Shared memory	15100
23	Shared memory	15200
26	Shared memory	15300
29	Shared memory	15400
32	Shared memory	15500
35	Shared memory	15600
38	Shared memory	15700
41	Shared memory	15800
44	Shared memory	15900
47	Shared memory	16000
50	Shared memory	16100
53	Shared memory	16200
56	Shared memory	16300
59	Shared memory	16400
62	Shared memory	16500
65	Shared memory	64000
68	Shared memory	64100
71	Shared memory	64200
74	Shared memory	64300
77	Shared memory	64400
80	Shared memory	64500
83	Shared memory	64600
86	Shared memory	64700
89	Shared memory	64800
92	Shared memory	64900
95	Shared memory	65000

Table 5.3: Profile id to context

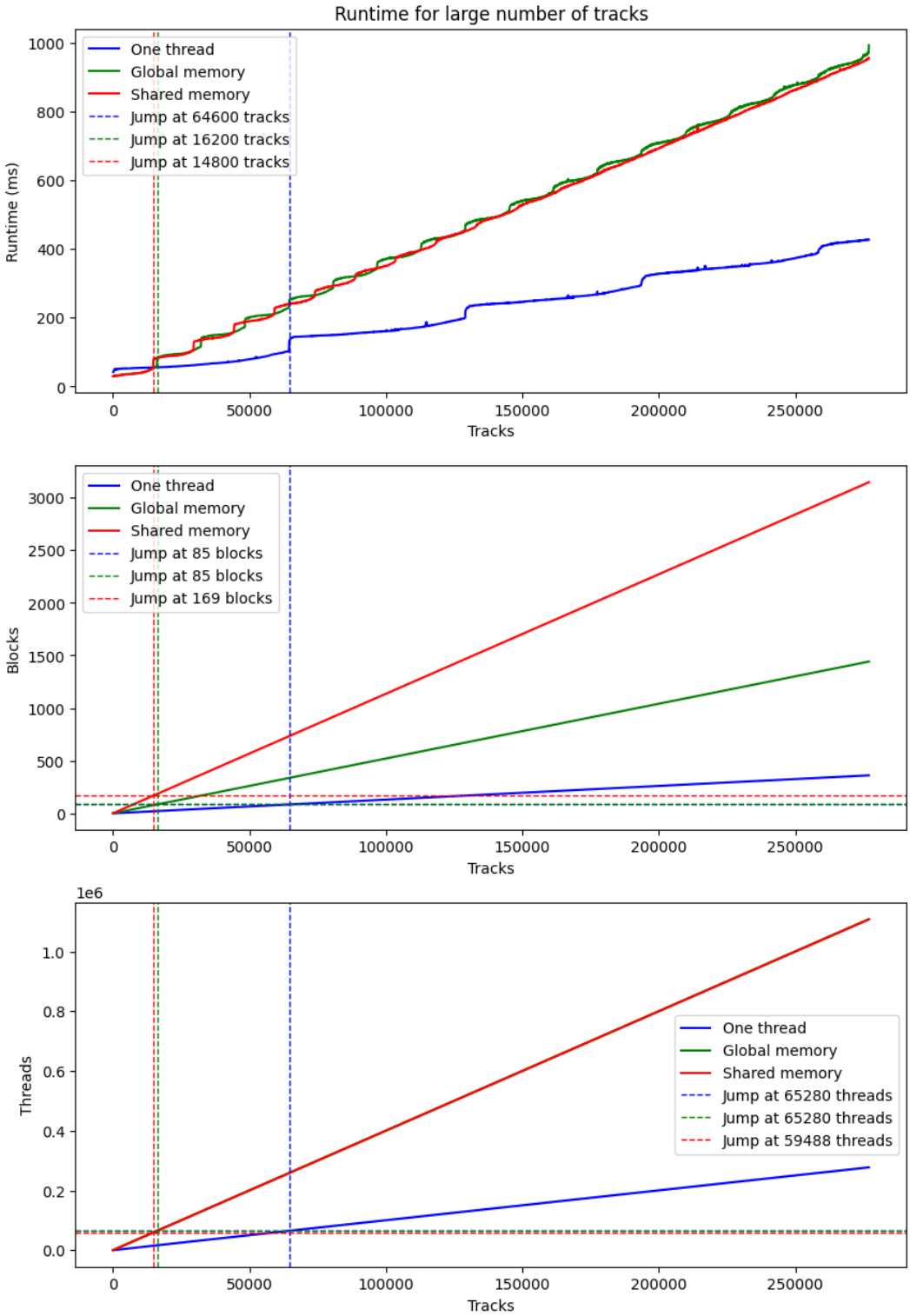

Figure 5.6: Runtime for a large number of tracks

5.3 Validation

The two implementations have been checked using the tool `mem-check` and `compute-sanitizer` from Nvidia. Those tools are respectively here to prevent some memory errors and race condition. In this Bachelor thesis, it is important to test those two things because there is more than one thread that is working with the same data and the shared memory is initialized with managed out-of-bound pointers.

These two tools aren't detecting any error on the two implementations.

The static analysis from a tool is not enough to prove that the implementation is correct. That is why there is a pull request [18] that is reviewed by the Celeritas team.

6 | Conclusion

This chapter is concluding the thesis, my stay at the LBNL and my studies at the HEIA-FR. The two first conclusions are objective and concerning the project and the thesis. The third one is personal and concerns my experiments during the three last years with a focus on the last 3 months.

6.1 Thesis conclusion

This chapter compares the goals defined Section 1.2.

First of all, the goal to learn GPU programming has been achieved. The chapter about that in the documentation was not written as a tutorial but as a reminder for the people that know this type of programming. The knowledge gained during this goal has been used during the whole project and it has been improved too.

Then, to improve the project, a dive into it has been necessary. However, with a bit of distance, I think too much time has been spent here and gained knowledge or facilities has not been used during the third objective.

The study of RKDP has helped a lot to do two implementations of a distributed RKDP. The two implements have been demonstrated that they are not improving the times for a large amount of data but I think this project has been very interesting and some things can be kept.

6.2 Project conclusion

Even if the main goal was to do a Bachelor's thesis to validate the Bachelor of Simon Barras, the project was also a need for the Celeritas project.

The implementation made to improve the runtime of the RKDP have some really good performance for a small number of particles. These implementations do not scale to the number of tracks that Celeritas is processing. The reason for that is the GPUs have not enough SM to run all the threads in parallel. As using four times more threads per track, the threads have to wait to have a SM available to run.

The only way to be able to use these implementations is to have a GPU with more CUDA cores. One of the possible solutions is to wait for the next generation of GPU or to find a way to distribute the vector multiplication with fewer threads.

To resume, the implementations are a Ferrari and a Pagani that we want to use to do some off-road.

6.3 Personal conclusion

This project was a great opportunity for me to learn a lot of things.

I discovered a new country and I practiced my English. It was the first time that I was so far from my home for more than one month. The fact that I was alone was a challenge for me, but I found a new way to enjoy my life. This experience forces me too to be more independent and deal with a very difficult administrative system that is the American one.

Concerning life in the LBNL, I was very lucky to be in a team with people from all over the world. The coffee time was a great moment to share our culture and learn more about the other. This place is very special too because it is situated near the Silicon Valley and there are a lot of people with a high level of education. The events organized in place are very interesting to discover or learn more about a lot of subjects. I have seen that Machine Learning and Quantum Computing are very popular. I would really enjoy coming back to California to work after my career.

About my project, I enjoy discovering a new way of programming. I really love to deal with the constraint and the advantages of the GPU. I personally think that the experiments done will help the Celeritas project to have a better understanding of how RKDP could be improved. I found that my project organization was good and even with the distance, I was able to keep my supervisors and my expert informed about the progress of my work. If I have to do it again, I will probably try to start my documentation earlier but I think it is a good document to trace my work.

To conclude, I would say that I am very happy to graduate (This year I hope) at the HEIA-FR. It was not easy every day, I remember my first year in German and the evaluation of trigonometry where I understand that "Dreieck" means triangle. All the calls that I made with Nicolas Terreaux to finish our projects and all the coffee we drink to stay awake. I have some really good experiences like Eurobot in spring 2022 where we finish at the highest ranking that the school has ever done and, of course, this amazing experience that gives me the wish to discover even more about the world and the computer science. To finish, I would like to thank all the people that I meet during my studies. All the teachers, the staff and, of course, all the students and I hope that I will have the luck to work with some of them in the future.

7 | Generative AI and Tools

To avoid all misunderstandings with the new chart of the HEIA-FR concerning the usage of tools issued from generative AI, I want to clarify.

As these are very powerful tools, I'm totally against the idea of banning them, and I'm looking instead for a way to increase my productivity by using them. However, I do agree that productivity gains must not take precedence over the right to personal property, but I do have some reservations about the HEIA-FR guidelines.

When I code or write reports, I am accompanied by the "GitHub Copilot" tool. This tool constantly generates suggestions that I accept, inspire or ignore. You'll understand that I can't define the exact lines generated. In addition, this raises a question: if Copilot suggests the exact line I wanted to write and I accept its proposal, is it generated or not? That's why you'll never find a mention that says that a line is generated unless I find the information relevant.

I use forums like Stackoverflow or chatbots like chatGPT in two ways. The most common is to solve a bug where I can't find the solution right away. The second is when I don't know how to do it and need an example. In all cases, and whatever the tool, I try to describe my context as best I can and explain my problem. The answers I receive are tested and understood so that they can be perfectly adapted to my situation and integrated. Here again, you won't find any explicit mention in the code, except in relevant cases, as I can't trace all my inspirations.

I should also mention that I do most of my work in English, so I use Deepl to translate texts, sentences and words. For proofreading, I have a premium subscription to Antidote, which improves quality by highlighting errors. To create the graphs, I principally use matplotlib with Python to generate the graphs that show data. The schemas that look hand-drawn are made with excalidraw. Other proofreaders, translators and other tools will probably be used under my supervision. Other sources such as Wikipedia or other sites and articles will be read for inspiration and mentioned if necessary.

Finally, even if there's no annotation mentioning the use of a tool, consider that everything I've done could have been manipulated by an AI or any other tool. However, I remain the only captain of the ship and I assure you that I understand and can explain what is being produced. This includes assuming all responsibility for the work I provide.

8 | Declaration of Honor

I, the undersigned, Simon Barras, declare on my honor that the work rendered is the result of personal work. I certify that I have not resorted to plagiarism or any other form of fraud. All sources of information used and author citations have been clearly mentioned.

9 | Acknowledgements

This Bachelor thesis has been made by Simon Barras, a student of the Haute Ecole d'Ingénierie et d'Architecture de Fribourg. This includes that it is subject to the HES-SO regulations and if it achieves, it will graduate the student. It has been supervised by Prof. Frédéric Bapst and Prof. Jean Hennebert, both are teachers at the institute iCoSys from the HEIA-FR and the expert was Dr. Baptiste Wicht.

This thesis has been made in the Lawrence Berkeley National Laboratory in Berkeley, CA, USA. Simon Barras was in the team of Paolo Calafiura and he works closely with Julien Esseiva. The project was part of the Celeritas project which has its acknowledgements.

9.1 Celeritas

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of High Energy Physics, Scientific Discovery through Advanced Computing (SciDAC) program.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

9.2 Thanks

I would thank all the people that help me to do this project. First, I would thank my supervisors which whom we have been in contact every week. They helped me to keep focus on the objectives and they gave me some inputs to improve my work. I would thank Dr. Baptiste Wicht who was my expert and they have kept an eye on my work. His experience in the field of the GPU helped me to have a better understanding of how the GPU works and how to improve the performance of Cleritas. I would thank Paolo Calafiura that includes me in his team and that gave me the opportunity to work in the LBNL. He was always available to answer my questions and to help me to find my places inside the team. I have to say a big thank you to Julien Esseiva where the person with that I work the most with. I asked a lot of questions and he always answered me with a lot of details. To conclude, I would thank all the other people that helped me to do these experiments like my family for their support but also the staff at the HEIA-FR that helped me to get my visa.

10 | Software Version

These are all the tools with their version used in this project.

Software/tool	Version
googletest	1.10
vecgeom	1.2
Geant4	11.1.1
cmake	3.26.3
root	6.28.04
gcc	11.3.0
cuda	11.8
c++	17

Table 10.1: Software version

List of Figures

List of Tables

List of Listings

References

[1] Wikipedia contributors. *Dormand-Prince method — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-June-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Dormand%E2%80%93Prince_method&oldid=1138052375.

[2] Apr. 2023. URL: https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods.

[3] celeritas-project. *celeritas*. https://github.com/celeritas-project/celeritas; accessed 17-June-2023. [GitHub repository; Commit c8db3fc; Celeritas is a new Monte Carlo transport code designed for high-performance simulation of high-energy physics detectors.] 2023.

[4] S. Agostinelli et al. "Geant4—a simulation toolkit". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3 (2003), pp. 250–303. ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(03)01368-8. URL: https://www.sciencedirect.com/science/article/pii/S0168900203013688.

[5] CERN. *2016 physics season starts at the LHC*. URL: https://home.cern/news/news/accelerators/2016-physics-season-starts-lhc.

[6] *VecGeom/VecGeom: The new geometry library for ROOT*. URL: https://gitlab.cern.ch/VecGeom/VecGeom.

[7] Nersc. *Using perlmutter*. URL: https://docs.nersc.gov/systems/perlmutter/.

[8] May 2023. URL: https://en.wikipedia.org/wiki/Ampere_(microarchitecture).

[9] NVIDIA Corporation. *Programming Model*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-simt-programming-model; accessed 8-August-2023. Asynchronous SIMT Programming Mode. 2023.

[10] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf; accessed 1-August-2023. Page 22. 2020.

[11] Oak Ridge. *CUDA Training Series*. URL: https://www.olcf.ornl.gov/cuda-training-series/.

[12] NVIDIA Corporation. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications-technical-specifications-per-compute-capability; accessed 8-August-2023. Technical Specifications per Compute Capability. 2023.

[13] Seth Johnson. *Celeritas: toward GPU-based particle transport for detector simulations in HEP expirements*. https://github.com/celeritas-project/celeritas-docs/blob/main/presentations/doe-briefing-20201019/pres.pdf; accessed 21-July-2023. Slide 9. 2021.

[14] Stefano Tognini. *An overview of the Celeritas; A novel GPU Monte Carlo detector simulation code*. https://github.com/celeritas-project/celeritas-docs/blob/main/presentations/aps-20230416/aps-seminar.pdf; accessed 21-July-2023. 2023.

[15] Tom Evans. *Exascale Computing at ORNL Past, Current, and Future: Opportunities for High Energy Physics*. https://github.com/celeritas-project/celeritas-docs/blob/main/presentations/cern-2022/evans-hpc.pptx; accessed 21-July-2023. Slide 42. 2022.

[16] Seth Johnson. *Celeritas: EM physics on GPUs; and a path to full-featured accelerated detector simulation*. https://github.com/celeritas-project/celeritas-docs/blob/main/presentations/chep-2023/srj-chep.pdf; accessed 21-July-2023. Slide 8. 2023.

[17] Julien Esseiva. *Integration of Celeritas into Athena*. https://github.com/celeritas-project/celeritas-docs/blob/main/presentations/atlas-sncweek-20230614/atlas-integration-atlas-sc-esseiva.pdf; accessed 21-July-2023. 2023.

[18] Simon Barras. *GitHub Pull Request - First optimization of the RKDP*. https://github.com/celeritas-project/celeritas/pull/864; accessed 5-August-2023. 2023.

[19] *Spack documentation*. URL: https://spack.io/.

[20] *Modules documentation*. URL: https://modules.readthedocs.io/en/latest/.

[21] Simon Barras. *GitHub Pull Request - Zeus profiling*. https://github.com/celeritas-project/regression/pull/1; accessed 5-August-2023. 2023.

[22] Simon Barras. *GitHub Fork - Celeritas*. https://github.com/simbarras/celeritas; accessed 6-August-2023. 2023.

[23] NVIDIA Corporation. *NVIDIA A6000 Tensor Core GPU Architecture*. https://www.nvidia.com/en-us/design-visualization/rtx-a6000/; accessed 7-August-2023. 2021.

[24] NVIDIA Corporation. *NVIDIA Nsight Compute*. https://developer.nvidia.com/nsight-compute; accessed 7-August-2023.

[25] Simon Barras. *Nsight Compute Profile*. https://gitlab.forge.hefr.ch/frederic.bapst/tb23-gpu-opt-celeritas/-/tree/main/doc/05-resources/profile.zip; accessed 7-August-2023. 2023.

Acronyms

ANL Argonne National Laboratory.

API Application Programming Interface.

ATLAS A Toroidal LHC ApparatuS.

BNL Brookhaven National Laboratory.

CERN European Organization for Nuclear Research.

CMS Compact Muon Solenoid.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

DOE Department of Energy.

FNAL Fermi National Accelerator Laboratory.

GPU Graphics Processing Unit.

HEIA-FR Haute Ecole d'Ingénierie et d'Architecture de Fribourg.

HEP High Energy Physics.

HPC High Performance Computing.

LBNL Lawrence Berkeley National Laboratory.

LHC Large Hadron Collider.

NERSC National Energy Research Scientific Computing Center.

ORNL Oak Ridge National Laboratory.

OS Operating System.

RKDP Runge Kutta Dormand Prince.

SDK Software Development Kit.

SIMD Single Instruction Multiple Data.

SIMT Single Instruction Multiple Thread.

SISD Single Instruction Single Data.

SM Streaming Multiprocessor.

SMART Specific, Measurable, Achievable, Relevant, Time-bound.