Technical University of Denmark

May, 2024

Dynamic Particle Simulations using GPUs

Bachelor Project

Magnus Emil Mouritzen
s214931

Johannes Christian Poulsen
s204399

Abstract

This project explores the implementation and optimisation of a particle simulation on a GPU using CUDA. The simulation builds on a previously implemented CPU-based simulation developed by a research group at DTU Space, transitioning to GPU implementations to handle more particles with higher efficiency. Various scheduling methods were implemented and tested thoroughly to identify the most effective strategies for GPU utilization. Optimisations were made to fully utilise the GPU architecture, emphasizing memory accessing patterns and thread execution. Through analysis of these tests, the project proposes an optimal scheduling technique based on dynamic allocation of thread workload. The final simulation is ready to be used as the foundation for a fully-fledged PIC MCC particle simulation.

Contents

1 Introduction

In the field of physics, computer simulations are often utilised to study various physical phenomena. Many such phenomena involve the movement and interactions of molecules or smaller particles. For such simulations, a very large amount of particles must be processed for a high number of iterations to achieve sufficiently large and accurate results.

Traditionally, these simulations would be implemented utilising only a CPU. CPUs are very versatile, but must to a large extent to all work sequentially. A GPU, however, is designed to process large amounts of data in parallel where the operations performed on each data point are similar. Therefore, a GPU can be utilised to simulate thousands of particles at the same time. This can however be difficult to utilise if not all particles behave the same, especially if particles are added and removed throughout the course of the simulation.

A research group at DTU Space is developing a CPU-based simulation to study the phenomena of thermal runaway electrons. Individual electrons are simulated to move in the atmosphere and collide in various ways with different molecules. The motivation for this project is to study various approaches to an efficient GPU-based implementation of such a simulation.

The project is divided into three parts or simulations, each documented and tested. The first part provides a simplistic simulation where many different approaches to kernel design for particle simulation are explored. In the second part, the most interesting methods are selected to be further optimised and studied in a slightly more complicated simulation. Finally, in the third part, a much more comprehensive simulation is implemented to utilise so-called Monte-Carlo collisions and a mock-version of the Particle-in-Cell technique. The best scheduling method from the second part is developed further and used for this simulation. This final simulation provides a solid foundation to implement a physically correct particle simulation for a GPU.

The code is developed for an NVIDIA GPU and is written in CUDA, which is an extension of C++ that greatly simplifies GPU programming.

2 Background

2.1 GPU Architecture - Johannes

Architectures of CPUs and GPUs are designed and optimised to handle very distinct tasks. The CPU is designed with a small number of versatile cores with the purpose of executing few varying tasks as fast as possible.
The GPU on the other hand consists of many more lightweight cores that are organised into streaming multiprocessors (SMs). The SMs execute tasks simultaneously in parallel, resulting in massive performance increases when working on larger datasets.
A rough picturing of the distribution of the chip resources in the architecture of a GPU compared to the one of a CPU can be seen in figure (2.1)[1].

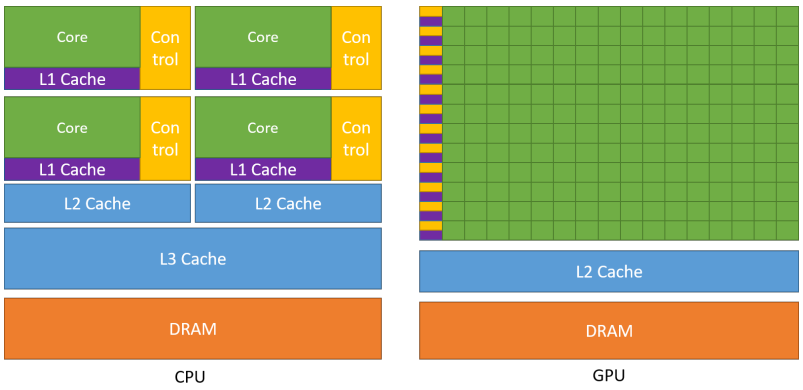

Figure 2.1: CPU- vs. GPU architecture

SMs are the primary computational units within a GPU. For Nvidia GPUs the specific components of an SM can depend on the architecture. The GPU used throughout this project is a Tesla V100 meaning that the SMs follow the volta architecture. The SM contains multiple different components that all are optimised for different purpose. To break it down, each SM consists of an L1-cache and 4 processing blocks (see appendix figure (9.1)). Each processing block consists of numerous different cores and units (see table(2.1)) [2].

Type	Amount	Purpose
FP32 Core	16	Single-precision arithmetic operations [3]
FP64 Core	8	Double-precision arithmetic operations
INT32 Core	16	For integer math
Tensor Core	2	Specialised for deep learning and matrix operations
L0 Instruction Cache	1	Stores instructions
Warp Scheduler	1	Handles which warp to execute next. Can handle 16 warps at a time [4]
Dispatch Unit	1	Works with the warp scheduler to dispatch instructions to execution units
64 KB Register File	1	Data Storing

Table 2.1: Content of processing block

When GPU programs (known as a kernels) are run, the threads are firstly divided into thread-blocks, where the size of the blocks is user defined. A collection of blocks is called a grid. The thread-blocks in the grid are run by the SMs. In the hardware of a GPU, thread-blocks are composed of warps. A warp is a collection of 32-threads that all are contained in the same block and run the same instruction at the same time. With each warp-scheduler being able to handle 16 warps each, the total number of warps that can be handled by each SM is 64 corresponding to a total of 2048 threads. Initialising a new block on the SM can not be done until there are sufficient warps to cover all threads in the block. For example, a thread-block with a block size of 1024, would need an SM to have 32 warps available to cover the entire block. At the same time, SMs do not free warps until all threads inside the thread-block have been executed. It is therefor important to find the right block size for the kernels to avoid extra wait time for blocks to be executed. The L1-cache of the SMs, along side caching, is used for storing shared memory. Shared memory enables fast memory accessing for threads being executed in the same thread-block. This means, using more shared memory causes less memory for caching and can limit the amount of blocks processable by each SM.
When executing warp-instructions, the GPU adopts a Single Instruction, Multiple Threads (SIMT) model to efficiently run data on very large data sets. This is an abstraction of the Single Instruction, Multiple Data (SIMD) model used in CPUs. In SIMD a single processor core may perform the same computation on multiple data points by using vector registers. This allows it to handle all the data with one vector instruction. Where the SIMT differs is that it instead runs the same instructions on multiple threads with the data contained in the specific thread.
Considering how a warp is executed is important when writing efficient code for GPUs. If threads within a warp were to get desynchronised due to branching, the increased parallelism as a result of the SIMT-model would decrease greatly. This is because both branches of the code would need different instructions, resulting in the warp having to execute both branches separately.

2.1.1 Memory accessing

A GPU contains its own memory, separate from the CPU. Memory transfers are necessary to communicate between CPU and GPU, which can take a considerable amount of time. Optimising the amount and size of memory transfers can greatly increase performance.

Accessing global memory on the GPU is done using dynamic random access memories (DRAMs) which is relatively slow for reading small amounts of data. DRAMs usually work by loading chunks of data at once to be used for caching. If warps are organised such that all threads in the warp access memory close to each other, the hardware can detect multiple DRAM-requests and coalesce these into one DRAM operation, potentially saving significant time. [5]

2.2 Physics - Magnus

The motivation behind the particle simulation that this project is inspired by is to study the behaviour of electrons in the atmosphere, particularly during the phenomenon known as thermal runaway where electrons reach incredible speeds. The program simulates individual electrons in discrete timesteps according to forces applied to them by the other electrons, as well as collisions with molecules in the atmosphere.[6]

2.2.1 Monte-Carlo Collisions

A particle simulation based on Monte-Carlo Collisions utilises probabilities to simulate extremely complicated particle interactions in a very simple way. Specifically, it considers accurately simulating every particle collision to be unimportant when there is a high

enough amount of particles. Instead of knowing exactly which conditions generate a certain collision, it is enough to know the average probability of said collision for a particle with a given speed. These probabilities are gathered through experiments and compiled into so-called cross section data. To determine if an electron has a collision in a given timestep, its current energy is turned into an index for the cross section data, and from that probabilities for different kinds of collisions are gathered. A random number is then generated, and the electron is simulated according to what happens. [6]

2.2.2 Random timestep and Null-collision

We are aware that there exists an optimisation to MCC-simulations utilising the fact that it is wasteful to perform iterations where no collision occurs. Instead of generating a random number to determine if a collision occurs in the current timestep, the basic idea is to generate a random number to determine how much time goes by before the next collision occurs, and then determining the nature of that collision. This will of course reduce the number of iterations and increase the number of collisions per iteration. We have unfortunately not had the time to implement or research this feature further, and it is thus not used in the project.

2.2.3 Particle-in-Cell

A Particle-in-Cell (PIC) simulation arranges electrons into groups according to their positions. Each group is considered a cell in a grid and is treated as one entity when calculating the electrons' forces upon each other. This way, instead of every electron considering every other electron when determining its acceleration, it must only consider every cell, or perhaps only the nearby cells, depending on the solver used. Additionally, the cells do not necessarily need to be updated every time the particles move. Instead, a number of iterations (called mobility steps) are performed within a so-called poisson step. For each poisson step, the grid of cells is recalculated with a poisson-solver. [7] This type of simulation is the inspiration for the implemented simulation described in the PIC section (3.4).

3 Method

3.1 Scheduler Tests - Johannes

There are many ways to implement a kernel to achieve the same simulation result. These can vary between the amount of particles each thread handles, which order they are simulated in, how many threads are launched, how many times the kernel is invoked, etc. An implementation of a kernel, along with the CPU's method of invoking it, is known as a scheduler.
Various different schedulers are implemented to test and study the performance differences when handling a simple 2D particle simulation. This gives a basic understanding of the advantages and disadvantages of each when implementing more complicated simulations.

The program simulates multiple particles bouncing in a box, with each particle splitting into two when hitting the bottom of the box. The schedulers therefore have to handle an increasing number of particles throughout the simulation.

In total 10 scheduler are implemented, all varying in different levels from one another.
All schedulers update the particles using the same *updateParticle*-function. This function updates the particle one step at a time and uses an atomicAdd-operation to update the total number of particles, should a new be spawned. atomicAdd adds to a number and returns the old value, while ensuring that no atomic operation from another thread can happen to the same variable at the same time. This is great for avoiding race conditions occurring but might cause a slow down if performed by many threads, as only one can succeed at a time. It also ensures that the operation is not performed on a local, cached version of the variable.

For some schedulers in this section, the implementation of the CPU side for launching the kernel as well as the kernel itself are shown in pseudo code to present the idea of the scheduler. In these the variables specified in the following table (3.1) are used.

Variable	Description
n	The current amount of particles (GPU memory)
init_n	The initial amount of particles
n_host	The current amount of particles (CPU memory)
max_t	The amount of iterations run
capacity	The maximum amount of particles allowed
block_size	The size of the thread-blocks
num_blocks	The amount of blocks in the grid

Table 3.1: Code Variables

3.1.1 CPU Sync Iterate

The first scheduler implemented is called CPU Sync Iterate. This very simple scheduler updates all particles once every kernel invocation and runs the kernel once per time step. This scheduler only runs the kernel with the exact number of blocks needed to simulate the current number of particles with the pre-specified block size. It therefore has to synchronise the new total number of particles back to the CPU after each time step before it can launch the next kernel, which forces the GPU to stop its work and wait after each

kernel call. Another likely cause of significant performance cost could be the repeated starting of a kernel. It should also be noted that some threads in the last block might not correspond to an actual particle. To let them know that they should just immediately return, the parameter start_n is passed to the kernel.

CPU:

```
for (int t = 1; t <= max_t; t++) {
    int num_blocks = (min(n_host, capacity) + block_size - 1) / block_size;
    int start_n = n_host;
    runKernel<<<num_blocks, block_size>>>(start_n, ...);
    cudaMemcpy(n_host, n);
}
```

If a thread is started that has an index corresponding to a number larger than the current amount of particles, the thread returns.

Kernel:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i >= start_n) return;
updateParticle(i);
```

3.1.2 Huge Iterate

Huge Iterate works much like CPU Sync Iterate in that it invokes the kernel once per time step. However, in this version, instead of starting just enough blocks to handle the current number of particles, the kernel is started with enough blocks to handle the maximum number of particles the simulation can reach (capacity). However, as new particles can spawn during the kernel execution, and the kernel can't know many particles were present when it was started, there is a risk of accidentally simulating particles that have just been created. To avoid this, the kernel must check a variable on the particle that tells which timestep it was spawned in. Additionally, the kernel needs to know if there even is a particle at its index. The same variable can be used to mark if there is no electron present. These two checks are abstracted to ”ready” in the code.

A benefit of this is that the scheduler now avoids having to synchronise back to the CPU every timestep, since it's guaranteed to have sufficient blocks to handle the current number of particles. Therefore the CPU can invoke all the kernels at once, and the GPU can execute them without pausing to wait for the memory transfer and the next kernel invocation. There is however still some synchronisation within the GPU itself, as each SM has to wait for the other SMs to be done executing the current kernel before they all can start the next one. Additionally, having to execute numerous blocks that immediately return due to being past the current amount of particles will most certainly cost significant time.

CPU:

```
int num_blocks = (capacity + block_size - 1) / block_size;
for (int t = 1; t <= max_t; t++) {
    runKernel<<<num_blocks, block_size>>>(...);
}
```

Kernel:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i >= min(n, capacity) || !particles[i].ready) return;
updateParticle(i);
```

3.1.3 Static Simple Iterate

Like the previously mentioned schedulers, Static Simple Iterate invokes the kernel once per time step. However, this scheduler does not simulate one particle per thread like the formerly described. Instead, this makes use of so-called persistent static scheduling. In static scheduling a thread that gets initiated in the kernel call can potentially simulate multiple particles.

Instead of defining the number of blocks initiated in the kernel based on the number of particles, this scheduler defines the block count according to the amount of SMs on the GPU. This should give a more optimal number of blocks to be initiated in the kernel instead of starting more blocks than the GPU is capable of handling at once.

The distribution of work among the threads is predetermined. This means that specific particles are assigned to each threads based on their id's. With each particles potentially taking dissimilar time to simulate, the workload among the threads could get skewed, which could result in a bottleneck where slower threads slow down the whole kernel.

CPU:

```
int sm_count = cudaGetMultiProcessorCount();
int num_blocks = sm_count;
for (int t = 1; t <= max_t; t++) {
    runKernel<<<num_blocks, block_size>>>(...);
}
```

Kernel:

```
int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
int num_blocks = gridDim.x;
int block_size = blockDim.x;
for (int i = thread_id; i < min(n, capacity); i += num_blocks * block_size) {
    if (!particles[i].ready) return;
    updateParticle(i);
}
```

3.1.4 CPU Sync Full

To avoid the performance cost of starting a new kernel for each time step, this scheduler runs all the time steps of the current particles in one kernel call. When all the current particles have been simulated, it copies n from the GPU and makes a new kernel call for all the newly added particles. The initial call of the kernel is made with just enough blocks to cover all particles, like CPU Sync Iterate. However, after the initial particles are simulated, it only launches enough blocks to simulate the newly added particles. It uses a variable called last_n to inform the kernel where the new particles start appearing in the array. So it repeats until no new particles have been added.

When simulating new particles, it is important to only simulate them from the point when they were spawned. A timestamp variable on the particle tells when it was spawned and is used for this purpose.

There will be a performance cost as a result of having to synchronise with the CPU after each iteration and starting the kernel multiple times. The cost will likely be drastically smaller than in the previous schedulers, since the synchronisations and kernel calls happen much less frequently.

CPU:

```
int last_n = 0;
while(min(*n_host, capacity) != last_n){
    int start_n = min(n_host, capacity)
    int num_blocks = (min(n_host, capacity) - last_n + block_size - 1) /
        block_size;
    runKernel<<<num_blocks, block_size>>>(start_n, last_n, ...);
    int last_n = min(n_host, capacity);  // Update last_n to the amount just
        run
    cudaMemcpy(n_host, n);
}
```

Kernel:

```
int i = blockIdx.x * blockDim.x + threadIdx.x + last_n;
if (i >= start_n) return;
for(int t = max(1, particle[i].t + 1); t <= max_t; t++){
    updateParticle(...);
}
```

3.1.5 GPU Iterate Global Memory

In GPU Iterate Global Memory the scheduler utilises static scheduling whilst avoiding CPU synchronisation and having to start the kernel multiple times. This means that everything is persistently handled directly on the GPU.
Each thread updates every particle it has been assigned one time step at a time. Afterwards the threads wait for all particles to have been updated before loading the new total number of particles.
To ensure that all threads are always working on the same time step, two barriers have been implemented. Firstly, one that makes sure that all threads are done simulating before they read the new number of particles. And secondly, one that make sure that all threads are ready to simulate the next time step.

CPU:

```
int sm_count = cudaGetMultiProcessorCount();
int num_blocks = sm_count;
runKernel<<<num_blocks, block_size>>>(...);
```

Kernel:

```
for(int t=1; t<=max_t; t++) {
    for (int i = thread_id; i < capacity; i += num_blocks * block_size) {
        if (i >= start_n) break;
        updateParticle(i);
    }
    barrier();
    start_n = atomicAdd(n, 0);
    barrier();
}
```

The barrier used in this scheduler uses busy waiting with an atmoicAdd function call on a counter.

```
wait_target = block_size * num_blocks;
barrier() {
    atomicAdd(wait_counter, 1);
    while (wait_counter != wait_target) {
        continue;
    }
}
```

3.1.6 GPU Iterate Global Memory Organised

The scheduler is almost the exact same as described in GPU Iterate Global Memory. The only change is that this scheduler uses a slightly different barrier. To limit the amount of atomicAdds performed, only one thread in each block performs an atomic operation. It then utilises an implemented function called __syncthreads(), which causes each thread to wait until all threads of the block have reached it.

```
wait_target = num_blocks;
barrier() {
    if (threadIdx.x == 0) {
        atomicAdd(wait_counter, 1);
        while (wait_counter != wait_target) {
            continue;
        }
    }
    __syncthreads();
}
```

3.1.7 GPU Iterate Multi Block Sync

GPU Iterate Multi Block Sync works just like GPU Iterate Global Memory Organised and GPU Iterate Global Memory. What differentiates this scheduler is that is uses the Cooperative Groups API to launch the kernel. This API can for example be used to have multiple grids. Here it is primarily used for its grid.sync-function. This barrier works on grid-level, waiting for all threads within the grid to reach the sync-call.

Overhead using Cooperative Groups API

To test if overhead is produced by using the Cooperative Groups API, the Static Simple Iterate scheduler and GPU Iterate Global Memory scheduler were re-implemented with the only difference being them now using the API to launch the kernels.

3.1.8 Static GPU Full

Much like GPU Iterate Global Memory, this static scheduler only invokes the kernel once. However, this scheduler fully updates each particle for all time steps at a time before moving on to the next. With this approach, there is no need to synchronise the threads.
When no new particles are assigned, threads that aren't working on a particle are busy-waiting until a new particle is spawned. If all threads are waiting, it means that all particles have been simulated and the thread terminates.
This scheduler does however feature a warp-level synchronisation to limit potential branching. This synchronisation only applies to the particles that are not currently waiting for work. This need was discovered during testing.
Furthermore, when deciding the amount of blocks to use in the grid, the amount of SMs is multiplied with the amount of blocks each SM can handle at once. The reason we didn't do this previously will be explained in the results section (5.1.1).

CPU:

```
int sm_count = cudaGetMultiProcessorCount();
int blocks_per_sm = cudaGetMaxActiveBlocksPerMultiprocessor();
int num_blocks = sm_count * blocks_per_sm;
runKernel<<<num_blocks, block_size>>>(...);
```

Kernel:

```
int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
for (int i = thread_id; i < capacity; i += num_blocks * block_size) {
    while(!particles[i].exists) {
        if (n == n_done) return;
    }

    unsigned working_threads = __activemask();
    for(int t = max(1, particle[i].t + 1); t <= max_t; t++){
        updateParticle(i);
    }
    __syncwarp(working_threads); //wait for threads in warp

    atomicAdd(n_done, 1);
}
```

To keep track of the number of particles simulated, each thread uses a global counter (*n_done*) that is incremented with an atomicAdd-operation every time a particle has fully finished.

3.1.9 Dynamic With Threads

Instead of using static scheduling that may result in a skewed distribution of the workload, this approach uses dynamic scheduling. In dynamic scheduling, each particle isn't assigned to a specific thread. This means that when a thread finishes simulating a particle, the thread - regardless of its id and the count of particles - moves on to the next available particle. It does this using global indexing to keep track of the next particle to be simulated. It is otherwise identical to the previous scheduler.

CPU:

```
int sm_count = cudaGetMultiProcessorCount();
int blocks_per_sm = cudaGetMaxActiveBlocksPerMultiprocessor();
int num_blocks = sm_count * blocks_per_sm;
int i_global = 0;  // One global variable for all threads to share
runKernel<<<num_blocks, block_size>>>(i_global, ...);
```

Kernel:

```
int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
for (int i = atomicAdd(i_global, 1); i < capacity; i = atomicAdd(i_global, 1))
     {
    while(!particles[i].exists) {
        if (n == n_done) return;
    }

    unsigned working_threads = __activemask();
    for(int t = max(1, particle[i].t + 1); t <= max_t; t++){
        updateParticle(i);
    }
    __syncwarp(working_threads); //wait for threads in warp

    atomicAdd(n_done, 1);
}
```

3.1.10 Dynamic With Blocks

To limit the amount of atomicAdd operations being used, this scheduler updates the global index on block-level instead of on thread-level. By doing so each block is assigned a chunk (equal to the block size) of the total number of particles, which is stored in a global array. Each element in the array corresponds to the current starting index for corresponding block. Additionally, block-level synchronisation is introduced.

CPU:

```
int sm_count = cudaGetMultiProcessorCount();
int blocks_per_sm = cudaGetMaxActiveBlocksPerMultiprocessor();
int num_blocks = sm_count * blocks_per_sm;
int i_global = 0;  // One global variable for all threads to share
int i_blocks[num_blocks];
runKernel<<<num_blocks, block_size>>>(i_global, i_blocks, ...);
```

Kernel:

```
while (true) {
    __syncthreads(); //sync all threads in the same block
    if (threadIdx.x==0) {
        i_blocks[blockIdx.x] = atomicAdd(i_global, blockDim.x);
    }
    __syncthreads();

    int i = i_blocks[blockIdx.x] + threadIdx.x;

    if (i >= capacity) break;

    while(!particles[i].exists) {
        if (n == n_done) return;
    }

    for (int t=max(1,electrons[i].t + 1); t<=max_t; t++) {
        updateParticle(i);
    }

    atomicAdd(n_done,1);
}
```

Testing of this scheduler will help give an understanding of how the running time is affected by having a block-level synchronisation instead of using atomicAdd operations.

3.2 Random Numbers - Magnus

The following two simulations developed in this project require random numbers. CUDA includes the cuRAND library, which implements random number generation on GPUs. To use it, a number of states (curandState) must be initialised. Each state is used to make a random number, after which the state is altered, so a new number will be generated the next time. If multiple threads make random numbers at the same time, they should all use different states.

When initialising a state, a seed, sequence, and offset must be provided. The seed is the basis for the state and how it will change when numbers are made. Offset initialises it at a point further along in the process. So if a state is initialised with an offset of 5, it is identical to if it had been initialised with an offset of 0, and 5 random numbers had been generated with it. 0 is always used for the offset. Sequence is essentially also an offset, though multiplied by a very large number. As that amount of numbers will never be made, it in practice works like a modification to the seed. The theoretically proper way to use

seed and sequence is to let all states use the same seed and different sequences. In the opposite case - where the sequence is 0 and seed is varied - some of the theoretical guarantees of the randomness break down. The benefit to this, however, is that initialisation is much faster, as initialising a state with a high seed is costly.[8] [9] We had read that problems caused by this are incredibly rare, and therefore decided to use a sequence of 0 and varying seeds for PIC. At the very end of the project, we unfortunately realised that this indeed does cause problems for us, as can be seen in one of the results (5.3.1). We did not have time to fix this and redo the tests.

3.3 MVP - Johannes

This simulation constitutes the declared Minimum Viable Product (MVP) of the project. Its purpose is to showcase a few schedulers selected from the previous simulation to develop and study them further in a slightly simulated that is slightly closer to the one made by DTU Space . In Scheduler Tests (3.1), many of the schedulers were very similar. Therefore, the three most interesting or promising schedulers were selected for further work. These were CPU Sync Full (now renamed to CPU Sync), Static GPU Full (now Static), Dynamic With Blocks (now Dynamic). CPU Sync Iterate (now Naive) was also chosen for performance comparison due to its simple and intuitive design.

Each scheduler will be running the same deterministic simulation where an initial number of particles (*init_n*) are simulated for a number of time steps (*max_t*). The particles are spawned in random locations spread out in a 2D rectangle. In each step, each particle moves, and it has a predefined chance to spawn a new particle (*split_chance*). Beyond this, the updateParticle for each scheduler is the same as described in 3.1.

When analysing the performance of the schedulers, different versions of the MVP are tested; one that only simulates the initial particles without simulating any newly added particles, and one that also simulates the newly added particles. The former is done to provide a more stable environment to get an idea how long each scheduler takes to just move and add particles. With this, the performances of each scheduler will later be measured and compared to analyse how each performs relative to each other when adding more complexity to the simulations.

During testing, it was observed that each thread spent a lot of time having to access the global memory multiple times when updating a particle. To optimise this, the schedulers that fully simulate each particle at once now load the particle into the cache and only uploads it to global memory at the end. This is easily implemented as they all use the same function to simulate the particles.

3.3.1 Naive

Naive takes base in the CPU Sync Iterate-scheduler (3.1.1) but includes some optimisations. To make use of the speed increase from reading or writing to adjacent memory (2.1.1), each thread adds newly spawned particles to shared memory. Then, only one thread in each block has to update the global memory counter, after which the threads of the block each upload one of the spawned particles. With each block having a local counter, time should also be saved from having fewer atomicAdd-operations waiting for each other in the updateParticle function.

3.3.2 CPU Sync

CPU Sync works the same as CPU Sync Full (3.1.4)

3.3.3 Static

Static is equivalent to Static GPU Full (3.1.8) with the exception of a few small optimisations. To reduce the number of atomicAdd-calls done by each thread, this scheduler now only updates the counter for finished simulated particle (*n_done*) once it has finished all the particles that are assigned to it and ready to be simulated.
It was noticed that the warp synchronisation was causing a massive increase in the running time of the scheduler. A test was performed with the following configuration:

Static Test Configuration
- Block Size: 512
- Initial N: 10^4
- Time Steps: $3 \cdot 10^4$
- Split Chance (per mobility step): 0.02%
- Capacity: 10^8

With the warp sync, the total running time was about $5 \cdot 10^3$ ms. However, without the warp sync, the running time was only $1.4 \cdot 10^3$ ms. Finding the exact reason for this is difficult but it could be a result of each thread now being forced to wait for slower ones in the same warp where it before was able to continue.

3.3.4 Dynamic

Dynamic works the same as Dynamic With Blocks (3.1.10) with the only optimisation being that it now uses uses shared memory instead of global memory to keep track of the block index.
Indirectly, the optimisation to the *updateParticle*-function optimise the Dynamic even further. Since the Dynamic synchronises each block, all threads in a block work on consecutively placed particles in global memory. As described in section (2.1.1), this should result in more effective DRAM operations.

3.4 PIC - Magnus

The next major evolution of the project is the Particle-in-Cell (PIC) simulation. The goal of this stage is to have a more complicated simulation, introducing new restrictions and problems to solve. In general, the design of this simulation has been modeled to closely resemble a true Particle-in-Cell simulation (2.2.3), guided heavily by DTU Space 's code and the Monte-Carlo Collisions paper[6]. Besides this change towards a much more realistic simulation, the purpose of the PIC was also to dive deep into optimising a single scheduler from MVP, as will be explained later. The simulation features the following prominent additions.

- Separation of simulation steps (mobility steps) into larger chunks (poisson stepts).
- Organisation of the simulation space into a grid, each with local forces to affect the particles, updated each poisson step.
- Removal of particles by chance and out-of-bounds checks.
- Introduction of cross sections, meaning look-ups in a large array to find split and removal chances for particles according to their kinetic energy.

3.4.1 Structure

The implementation is structured such that the CPU runs a loop where each iteration corresponds to one poisson step. In each such step, a number of operations are executed on the GPU.

- **Reset grid.** Reset particle count in each grid cell in preparation for the next step. Each thread is responsible for one grid cell.
- **Particles to grid.** Count the number of electrons present in each grid cell. Each cell is responsible for one electron, and must calculate the appropriate cell according to its position. An atomic operation is used to increment the counter in the cell. The electrons processed by a given warp will likely all be in different cells. As atomic operations work best if they go to the same address (assuming they are first merged within the warp and done as a single operation) or adjacent addresses (2.1.1) there might be potential to speed up the simulation in this step. Further work could be directed at exploring this, perhaps by sorting the electrons according to their position.
- **Poisson solver.** Calculate the electric field across the grid according to the charge of each cell. A mock-version of this was made where the electric field in each direction is calculated by the immediate neighbours to the cell. Each thread is responsible for one cell.
- **Grid to particles.** Each particle reads the field present in the cell it is in, and stores an appropriate acceleration to be used throughout the mobility steps to come. Each thread is responsible for one particle. This step might also benefit from sorting of electrons according to position, as that would mean better cache utilisation.
- **Mobility steps.** The mobility steps of the poisson step are executed, not dissimilar to the execution of an entire simulation in the MVP.
- **Remove dead particles.** If, during the mobility steps, a particle disappears by random chance or by going out of the bounds of the simulation, it is marked as dead and ignored for the remainder of the mobility steps. In this step, the electron array is compressed to remove all the dead electrons.

3.4.2 Remove dead particles

The removal of dead particles is a relatively simple process. There are two arrays of electrons, one to read from (old), and one to write to (new). This could be cut down to one array to save memory, but there was not enough time to do so. Each thread is responsible for one electron in the old array. If it is not marked as dead, a spot for it is acquired in the new array through an atomic increment of a counter, and its values are copied over. At the same time, its timestamp is reset.

A basic implementation would look something like this in pseudocode:

```
1    int i = thread_id + block_id * block_size;
2    bool alive = electrons_old[i].timestamp != dead;
3    if (!alive) return;
4    int new_i = atomicAdd(n, 1);
5    electrons_new[new_i] = electrons_old[i];
```

This could be improved by adding block-level cooperation to reduce the amount of atomic operations.

```
    int i = thread_id + block_id * block_size;
    bool alive = electrons_old[i].timestamp != dead;

    shared int n_block;
    if (thread_id == 0) n_block = 0;
    syncThreadsBlock();

    int i_local;
    if (alive) i_local = atomicAdd(n_block, 1);
    syncThreadsBlock();

    shared int i_block;
    if (thread_id == 0) i_block = atomicAdd(n, n_block) ;
    syncThreadsBlock();

    if (!alive) return;
    electrons_new[i_block + i_local] = electrons_old[i];
```

However, for the actual implementation of this function, warp-level coordination was used. It is a set of techniques that lets the threads of a warp synchronise and exchange information. Once the threads in a warp have checked the status of their electron, they cooperate to build a bitmask to indicate which threads of the warp have an alive electron. One thread in the warp is designated as ”leader” and performs an atomic addition to the shared block counter equal to the amount of threads in the warp with a living electron. Each other thread in the warp with a living electron then copies the index given to the leader. Once the block's counter has been added to the global counter, each thread can calculate its own global index by checking how many threads ”before” it in the warp has a living electron (its rank) and adding that to the index the warp leader got. That number is then added to the block-level shared index for a final global index. See the actual implementation in pic.cu for further details.

This optimisation of course introduces additional operations and a potential minor slow down due to warp-level synchronisation, but it does potentially reduce the amount of atomic operations 32-fold. Quick tests were run to compare the performance of a simple implementation, a block level implementation, and a warp + block level implementation (the one actually used) with the following configuration.

Remove Dead Particles Configuration

- Block Size: $128, 1024$
- Initial N: 10^7
- Poisson Timesteps: 10
- Mobility Timesteps: 0
- Capacity: $2 \cdot 10^7$

The results are shown in the table below (3.2). Interestingly, the additional complexity makes the process slower for a high block size. A low block size, however, is overall faster, and in that case small improvements can be seen. This test also informs us that a block size of 128 likely will be the best no matter the block size used for other kernels in the program. This was unfortunately discovered too late in the process, so the same block size has been used for this and the other kernels in all the tests presented later.

	Block size 128	Block size 1024
Simple	$71ms$	$108ms$
Block	$66ms$	$111ms$
Actual	$64ms$	$118ms$

Table 3.2: Remove Dead Particles Results

3.4.3 Mobility steps schedulers

From the work on the MVP, it became clear that the persistent dynamic scheduler had the most potential for this project. Therefore it was chosen as the focus of attention for the development of PIC. It went through further optimisations as will be discussed here. Eventually, the Naive and CPU Sync schedulers were taken from MVP and implemented in PIC to serve as comparisons against the Dynamic scheduler. However, even though the new insights gained during the optimisation of the Dynamic scheduler without doubt could have benefited Naive and CPU Sync, they were not modified further due to time constraints. The persistent static scheduler was left out completely, as its performance was so close to the Dynamic one that it didn't seem to make much sense to study the difference further.

An interesting difference between PIC and MVP is that each run of a simulation scheduler constitutes only a small part of the entire simulation in PIC. This means that the scheduler might not be the bottleneck anymore. It is however still the focus of our work. As described later (7.2), future work can be directed at investigating other bottlenecks.

3.4.4 Dynamic scheduler optimisations

In MVP, the Naive scheduler had been optimised by the implementation of a simple shared buffer for new electrons. This did not seem feasible to implement in the more advanced schedulers without significant restructure. Additionally, a shared buffer with enough space for an entire block to upload to seemed too big to be practical.

With the focus for this simulation narrowed down to a single scheduler, and with the new-found knowledge of warp-level synchronisation, a major rework of the Dynamic scheduler began. The key difference lies in the ”inverted” structure of the simulation loop. In the old version, the structure was, in rough terms, as follows.

- All threads in a block get indices of new electrons to simulate.
- Each thread waits for the new electron to be ready to process. If some threads in a warp are ready while others are not, they will either be forced to wait, or a desync happens.
- Each thread loops simulates the particle fully through a loop. If a new electron is made, the other threads in the warp might have to wait for it to be uploaded. Additionally, if one electron only needs one iteration while another needs 100, the former thread must wait for 99 iterations without performing any work.
- Once all threads in the block have completed the cycle, it can start again.

After the rework, the structure is as follows.

- Each thread that doesn't already have an electron gets the index of a new electron to simulate. The threads within a warp cooperate to reduce the amount of atomic operations.

- If a thread got a new electron, it will be check if it is ready for work. If it is, or if it is not a newly acquired electron, it is simulated one single step.
- If a new electron is produced, the thread remembers this new electron.
- If the electron is done, it is uploaded, and the thread remembers that it wants a new electron.
- If the produced a new electron, the warp gains control of a block-level lock and adds any new electrons from that warp to a shared buffer with a size of 32. If it would overflow, the warp flushes the buffer to global memory and adds the remaining new particles.
- The cycle starts again.

With this system, a thread is only left to do nothing for a single simulation step if it doesn't have work to do anyway. Additionally, it gets the benefit of a buffer for better performance when adding new particles. For the implementation it should also be noted that a thread will check if the simulation is done if it doesn't have a ready particle, but it will not quit immediately upon realising there will be no more work for it. This is because its presence will be required to perform a variety of warp-synchronised operations, such as flushing the buffer. Once every thread in a warp considers itself done, the entire warp can exit the simulation.

There are, however, some drawbacks to this solution.

- **Complexity.** It contains significantly many more operations for each simulation step than the previous version, making it harder to understand, verify, and debug, and it also simply slows it down.
- **Desynchronised loading.** Though it is undesirable to leave threads spinning for many iterations while a few other threads in the warp simulate, it is also undesirable for the threads to load their new particles at different times. One might imagine a worst-case scenario where each thread in a warp must simulate its particle for one iteration more than the previous thread. Ie. one must do one step, another must do two, a third must do three, etc. For every single step, one thread will need to load a new particle, which will slow down the other threads. In this case, it would likely be better for each thread to simply wait until at least a few other threads are ready to get a new particle as well.

  Further work could be invested to investigate the potential performance boost of caching particles when loading. For example, a warp could cooperate to load more particles than it needs, using it as a shared pool for threads to load from if they finish their work.
- **Desynchronised uploading.** In the same vein as the previous issue, if particles finish their simulation at different times, they will have to wait for each other to upload the finished electron. It should be noted that this also is a weakness in the previous design.

  For further work, perhaps it would be beneficial to make a buffer system for the uploading of these finished particles as well.
- **Critical section in buffer.** As only one warp can interact with the buffer for new electrons at a time, there will naturally be some congestion. This issue will be relatively small, if the spawning of a new particle is rare enough that most warps don't

do it in a single iteration, but otherwise it might be a serious problem, as each warp in a block that wants to upload a new particle must wait for the other warps to do the same.

Even with these drawbacks, and even though there has been very little time to iron out the problems mentioned above, the new version performs significantly better than the old one, as will be shown in the results later (5.3).

3.4.5 Cross Section

The cross section is not based on real data, but is manually designed to pose a desirable challenge to the program. For simplicity, we also only consider two types of collision, and only one way for each of them to happen: the splitting of a particle to two, or the removal of the particle. When a particle splits, one of them has the same properties as the original one, while the other has its velocity reversed. Furthermore, the chance for split and removal are always the same to make the simulation more stable and easier to analyse. This equality means that the collision chance has no impact on the total amount of particle updates performed during a simulation.

Similarly to how it would be done in a ”real” physics simulation, when we must check if an electron has a collision, its energy is converted into an energy-level, which is an integer index to the cross section data. Our energy calculation is based on classical kinetic energy, and is simply the magnitude of its velocity, squared. The level is found with the following function. N_STEPS is the amount of energy levels, which we copied from the physicists code. It is 10^4. The minimum energy is 10^{-6}, and the maximum is 10^{16}. A logarithmic function maps it to the range of $[0; N\_STEPS[$.

```
__device__ int energyToIndex(double energy){
    int energy_index = trunc((log10(energy)+6)*N_STEPS/22);
    return (energy_index < 0) ? 0 : ((energy_index >= N_STEPS) ? N_STEPS - 1 :
         energy_index);
}
```

The chance for each of the two collision types are generated by the following formula and then stored to a file. The program reads the file at the start of the simulation. The chances in the formula and program are percentages.

$$chance = (sin(\frac{x^{2.6}}{5 \cdot 10^7}) + 1.01) \cdot x^{0.1}$$

The following two graphs show a visual representation of the cross section. The x-axis is energy level and the y-axis is chance of each of the collisions, again in percent. It peaks at about 5% near level 10^4, and the minimum is about 0.02%, not counting energy level 0, where the chance is 0.

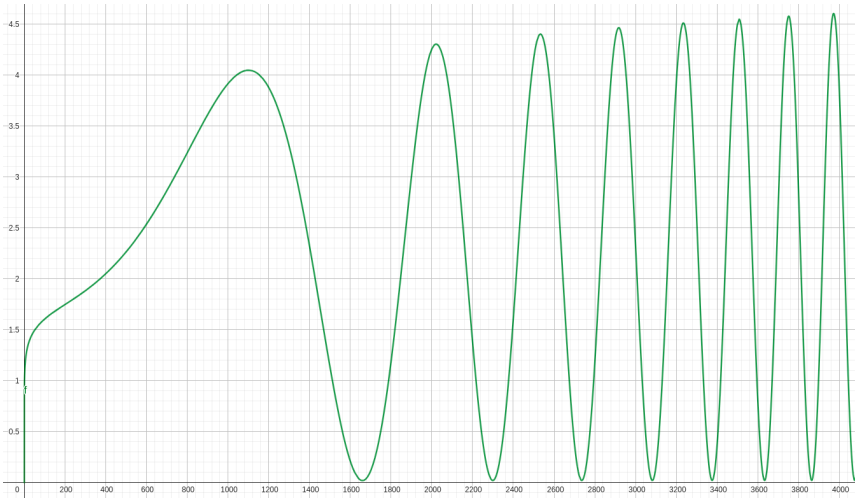

Figure 3.1: Cross Section data up to 4000

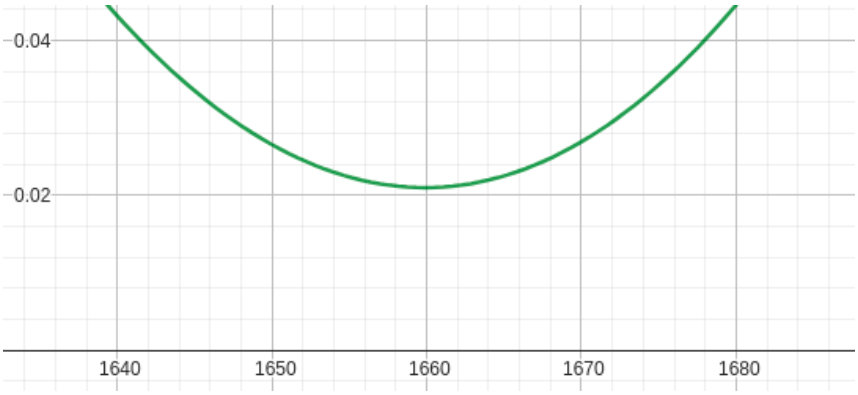

Figure 3.2: Cross Section data at minimum

4 Test - Magnus

During the project, several tools have been used to aid debugging, verification, and benchmarking of the simulations. The most significant ones are described here.

4.1 Visualisation

For initial testing, as well as for debugging the implementation of Scheduler Tests (3.1) it was useful to visualise the particles by drawing them to PNGs and animating them into GIFs. The code for this can be found in utility.cu and the python script to_gif.py. We did not make use of this for the two other simulations as the positions of particles in MVP mattered much less, and the number of particles in PIC would have to be way too high to draw them individually.

For the PIC specifically, it would make sense to create a visualisation which draws the density of particles in each cell, represented as a single pixel. This would show the development of the particles over time, and would allow for visualisation of any number of particles. Such a tool would be crucial for comparing the code with a trusted CPU-based implementation of the particle simulation to verify the correctness of the GPU-based simulation.

4.2 Jobs and Benchmarks

The programs are run on shared computers. This introduces noise when testing the performance of the code, as disturbances might occur due to others usage. To avoid this, all benchmarks have been performed by submitting jobs to the system. These jobs specify the required resources, including the need for exclusive access to the GPU.

4.3 Report

It is possible to generate reports from the execution of CUDA code, which can be viewed in the NVIDIA Nsight Systems program. This gives a timeline of the execution of the kernels as well as memory operations. Early in development it was a useful tool to understand the execution order of schedulers as well as the impact of memory copying and synchronisations. It did also occasionally prove useful for debugging. The program provides much further information that could be analysed, but we have not delved into that for this project.

4.4 Unit Tests

When developing new kernels or making changes to existing ones, it is of course important to verify that they behave correctly. One reason for this is the simple desire for correct behaviour, but the other reason - which possibly is more relevant to the majority of this project - is to ensure that the kernels are comparable in bench marks. That is, if one kernel has a bug causing it to produce more particles than another kernel, it will bias towards looking slower than the other one.

Manual verification of correct behaviour is useful during particular stages of development, but this method falls short for two important reasons. One is that it simply is bothersome to do, and one might decide to not do it at times when it should be done. Another reason is that manual verification only can be done on relatively low amounts of particles. While

it might be easy enough to check if the total number of particles produced by two kernels are the same, checking their positions is much more bothersome.

A unit test function was therefore made, and can be found in the respective test.cu file. It works on the premise that there is a ”trusted” kernel which the others are compared to. All the kernels are run, and their particle data is compared by comparing the number of them, and then comparing each particle at a time. To do that, they are first sorted according to all their values.

4.5 Determinism

For this unit test to work, the kernels must be deterministic. In the early stages of the project, the program would often be run such that the maximum number of particles was hit. However, we realised that hitting the capacity would break determinism, as there would be no other way to determine which thread gets to produce the final particle before the limit. Since then, the capacity has always been treated as something that may not be hit.

5 Results

5.1 Scheduler Tests - Johannes

To analyse the performance of each scheduler, a number of different tests are run. Each of these tests focuses on understanding the effect of various variables on the running time.
The data gathered will contribute to a better knowledge of how the properties of a GPU are best taken advantage of when further developing the particle simulation.

5.1.1 Effect of Block Size

As mentioned in section (2.1), it is important to find the right block size for the kernels. To test the effect of different block sizes, the schedulers described in (3.1) are all run with the following configuration, for which only the defined block size varies:

Test Configuration
- Block Size: $128, 256, 512, 1024$
- Initial N: $2 \cdot 10^5$
- Time Steps: 1900
- Capacity: 10^8

In this configuration, the initial number of particles are all spawned in the middle of the x-axis, but evenly spread out on the y-axis. This is done to ensure varying split frequencies of each particle, resulting in varying workloads for each thread.
With 80 SMs that each can run 2048 threads, the total number of threads that can be run simultaneously is ~164.000 if the SMs are full. For schedulers that define the number of blocks depending on the GPU, more blocks are need to be initiated to fill the SMs. For example, if a block size of 128 is defined, 16 thread-blocks need to be initiated to fill an SM with 2048 threads, while for a block size of 1024, only 2 thread-blocks are needed.
As a result of this, running with $2 \cdot 10^5$ initial particles should make sure that the full capabilities of the GPU are used.
As seen in figure (5.1), the scheduler affected the most is Huge Iterate. This is expected, considering that Huge Iterate allocates the amount of thread-blocks based on the maximum number of particles (capacity). With lower block sizes, more thread-blocks are waiting, that all need to be scheduled. Despite most threads in these thread-blocks not doing anything, they still need to be executed by an SM to terminate. This all takes time and could explain the drastic increase in running time Huge Iterate experiences with lower block sizes, compared to higher block sizes, where the scheduling work would be less.

To get a clearer view of the effect of block size on the other schedulers, Huge Iterate is removed in figure (5.2). Here, it appears that the running time of the schedulers vary a bit.

CPU Sync Iterate and the statically implemented Iterate-schedulers all appear to have running times increasing with block size.
The increase for CPU Sync Iterate is the most dramatic. This scheduler allocates the number of blocks based on the current number of particles in simulation. By doing so, more thread-blocks will be initiated than the SMs can handle. With a higher amount of threads per block, more warps in each SM need to be executed before leaving room for a new block. This causes the bottleneck created by a slow thread to be more severe as it would hold up more threads.

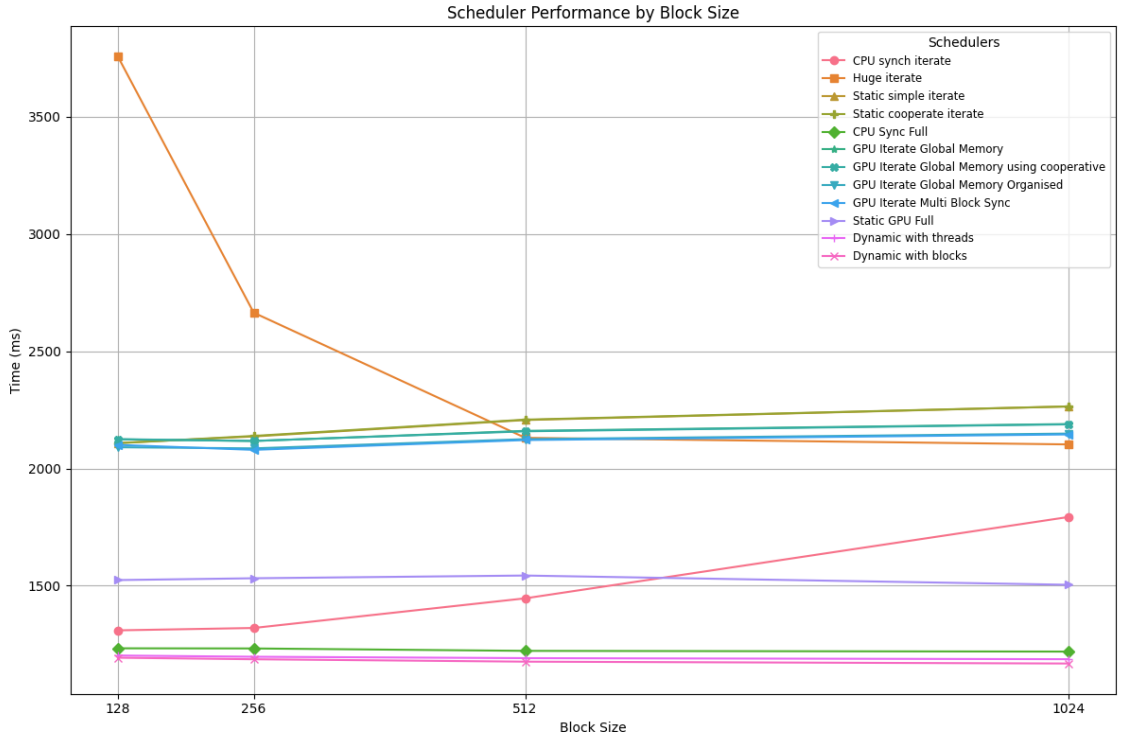

Figure 5.1: Running time of test schedulers as a result of block size

Determining why the the statically implemented Iterate-schedulers take longer with more thread-blocks is a bit more tricky. When defining the number of blocks to be used, these schedulers fully fill all the SMs on the GPU. To closer study the effect of this, the same test is re-conducted to examine the effect of filling the SMs fully. In figure (5.3) the schedulers are run with only one block per SM.

Here it can be seen that filling the SMs with as many threads as possible, with the purpose of increasing parallelism, isn't always guaranteed to improve the performance.

A contrast can be seen on how the different schedulers are affected by filling the SMs or not. For the schedulers that fully simulate the particles for all time steps, especially for lower block sizes, a performance gain can be observed with the SMs filled in figure (5.2). However, for the statically implemented Iterate-schedulers, the performance is improved by not filling the SMs and thereby lowering the total amount of threads in figure (5.3). A reason for this could be that these schedulers have to synchronise. Having to synchronise all threads can lead to some threads stalling when waiting for slower ones to finish. Despite granting a greater level of parallelism, the cost of having to wait for these extra threads appears outweigh the potential reward.
For this simple particle simulation, the best overall performing block size appears to be 512. While the block size definitely has a direct effect on the performance, it is important to remember that this is in correlation to the number of blocks, and that the balance of these might be even more individual for each scheduler as more complexities are added to the simulation.

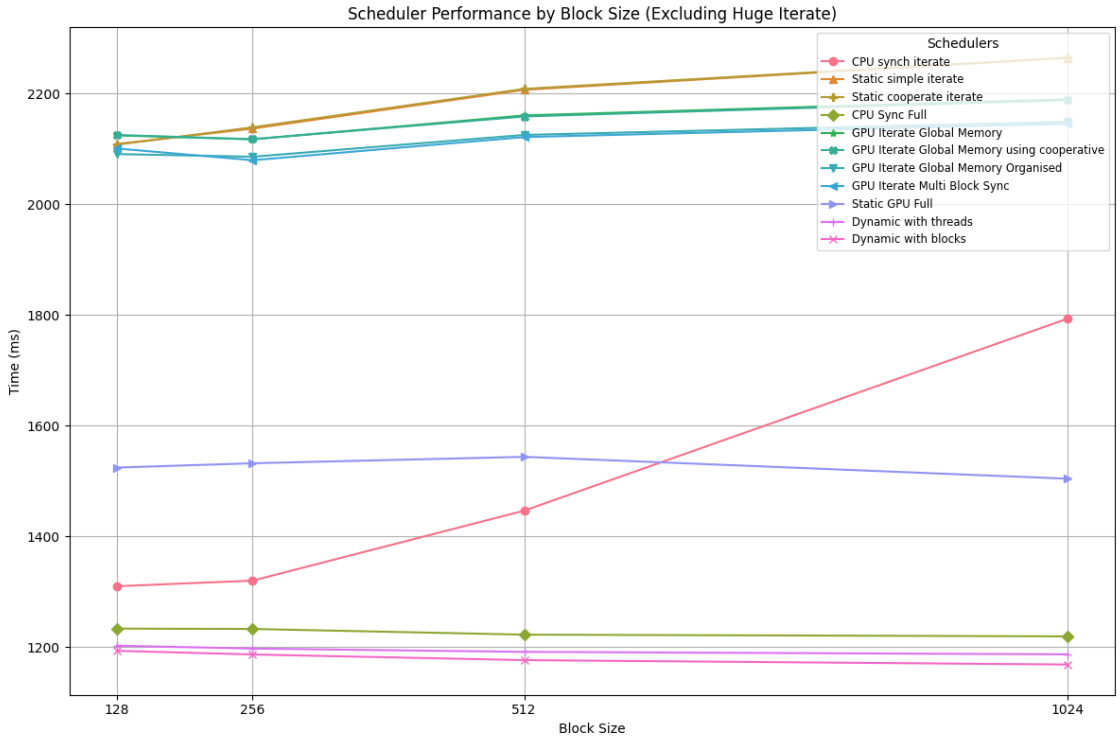

Figure 5.2: Running time of test schedulers as a result of block size (without Huge Iterate)

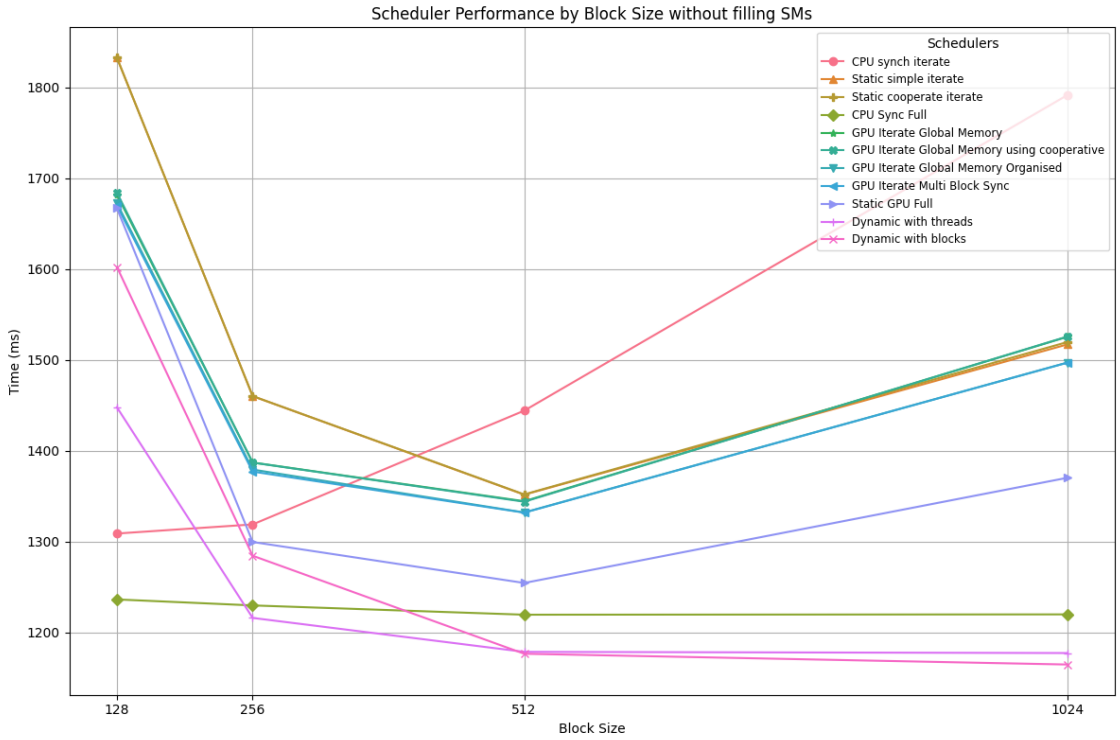

Figure 5.3: Running time as a result of block size without filling the SMs

5.1.2 Launching kernel using Cooperative Groups API

As seen in figures (5.1) and (5.2), for Static Cooperate Iterate and GPU Iterate Global Memory, using the Cooperative Groups API to launch kernels produces no significant overhead when comparing them to their regular launched counterparts: Static Simple Iterate and GPU Iterate Global Memory. Static Cooperate Iterate and GPU Iterate Global Memory will therefore be excluded from future tests.

5.1.3 Increasing Initial Number of Particle

In order to analyse the effect on the performance for each of the schedulers when increasing the initial number of particles, a new test is run. Each of the schedulers run the simulation multiple times with an *init_n* starting from 100. *init_n* is then increased by 100 until 1000 is reached where after *init_n* is increased by 1000. The simulation is run in this fashion until an *init_n* of 10^7 is reached. The particles are all spawn in the middle of the 2D box.
For the rest of the variables, the following test configuration is used:

Test Configuration

- Block Size: 512
- Time Steps: 5000
- Capacity: 10^8

Based on the observations done for figure (5.3), the statically implemented Iterate-schedulers are from here and onwards run with a number of blocks equal to the amount of SMs in the GPU. The rest of the schedulers, that use a GPU specified number of blocks, fill the SMs.

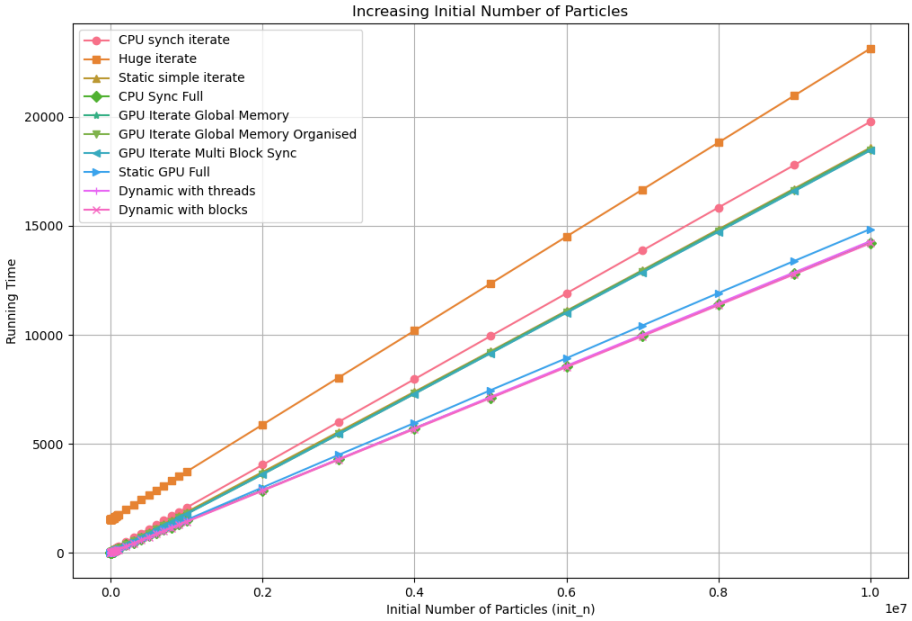

Figure 5.4: Running time as a result of increasing initial number of particles.

By looking at figure (5.4), it appears that the running time increases linearly with the amount of initial particles. This is to be expected, as all particles are spawned in the same coordinates with the same total time steps. For example, if the initial number of

particles is 10^6 and after 5000 time steps there is a total of $8 \cdot 10^6$ particles, then if the initial number of particles is doubled so are the resulting number of total particles. This means that $2 \cdot 10^6$ initial particles would produce $16 \cdot 10^6$ total particles that need to be simulated.
This is valid for higher values of *init_n*, where the GPU is fully used. However for lower values, the GPU shouldn't be fully utilized before there are several thousand of particles. To further analyse how the running time corresponds to fewer initial particles, figure (5.4) is re-plotted with a logarithmic scale in figure (5.5).

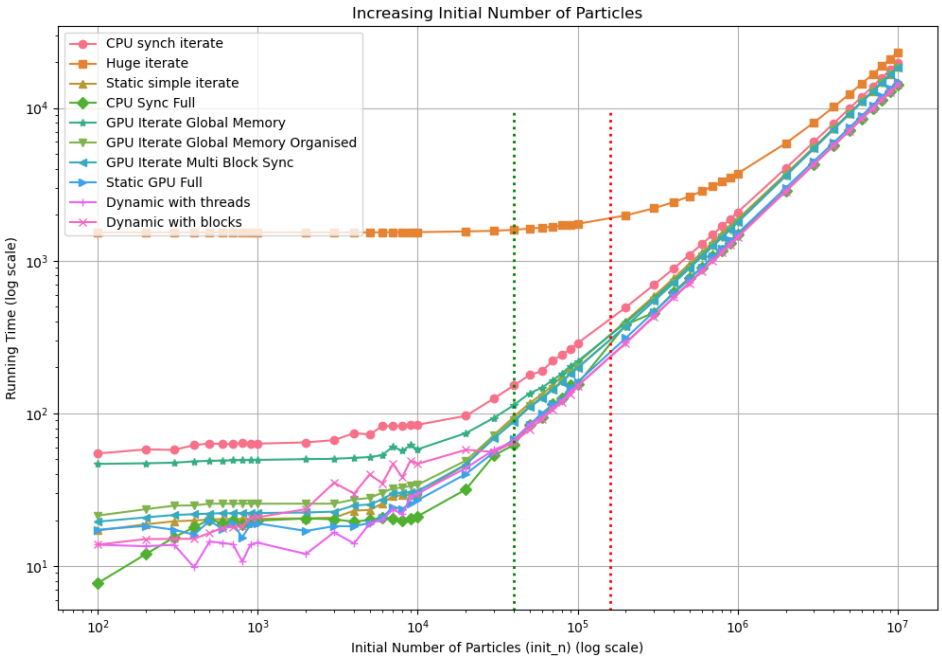

Figure 5.5: Running time as a result of increasing initial number of particles. (logarithmically scaled). The green line marks the 40.000 tick while the red line marks the 160.000 tick of initial particles

To use all 80 of the SMs, $80 \cdot 512 \approx 40.000$ particles need to be handled at the same time. While ≈ 160.000 need to be handled to ensure that all SMs are fully used. Two lines are inserted in the graph to mark the 40.000 (green) and 160.000 (red) initial particle ticks. Due to more particles being spawned, 40.000 and 160.000 particles are with all likelihood handled before these marks, but can be guaranteed after.
In figure (5.5) it can be seen that the running times are much more constant until all SMs are used. Here, it can be observed that the best schedulers for more particles aren't necessarily the best for fewer. As seen in (5.4), the best performing schedulers for more particles are the dynamically implemented and CPU Sync Full with Static GPU Full close behind. Taking a closer look at the Dynamic with Threads-scheduler, it, when compared to the other best performing schedulers, performs worse with fewer particles. There could be many reasons for this, but one that quickly stands out when looking at section (3.1.9) is the many atomicAdd operations used. Every thread that is started in this scheduler, regardless of it having a particle to simulate, is calling atomicAdd. With few particles the effect of the overhead is very significant, but quickly diminishes when the all threads have work.

5.1.4 Increasing Number of Time Steps

The relationship between the running time and increasing number of time steps is now analysed. In this test, the simulations are run with the following configuration for non-changing variables:

Test Configuration

- Block Size: 512
- Initial N: 10^3
- Capacity: 10^8

The total amount of time steps are incremented by 100 until 20.000 is reached. The particles in this simulation again are all spawned with the same coordinates in the middle of the box.

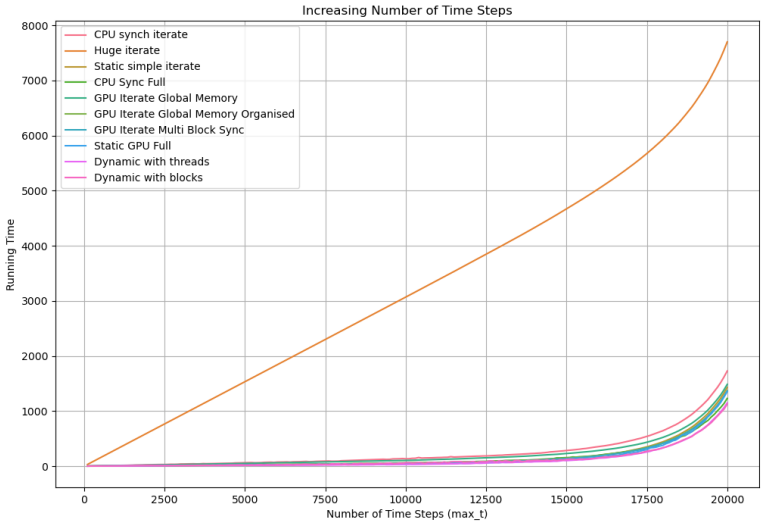

Figure 5.6: Running time as a result of number of time steps

In figure (5.6), the running times of the schedulers increase exponentially with the amount of time steps (see appendix 9.2 for zoomed in excluding Huge Iterate). The horrific performance of Huge Iterate can easily be explained by the capacity of the simulations. As explained earlier, this causes numerous blocks to be initiated that don't perform any work. Figure (5.7) logarithmically scales the running time and the number of time steps. Here, it can be seen that the running time for the most part actually grows linearly with *max_t* and only with higher amounts of time steps start growing exponentially.

Figure (5.8) takes a closer look at the final number of particles plotted against the number of time steps. It can be observed that figure (5.6) follows the same trend. The exponential increase with more time steps could therefore caused by a high number of particles needed to be simulated, added to linear effect of time steps on running time seen in figure (5.7). All in all, the rapid increase in running time could be explained by these steps:

1. Particles have to do more time steps, meaning each thread need to work more per particle.
2. The more each particle moves, the more particles will be spawned.
3. Newly spawned particles then repeat this, resulting in even more work.

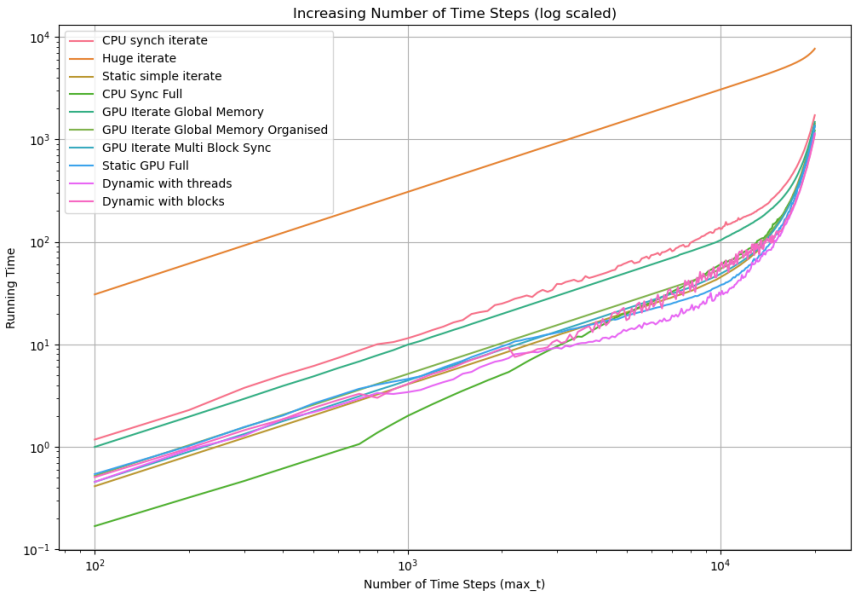

Figure 5.7: Running time as a result of number of time steps - logarithmically scaled

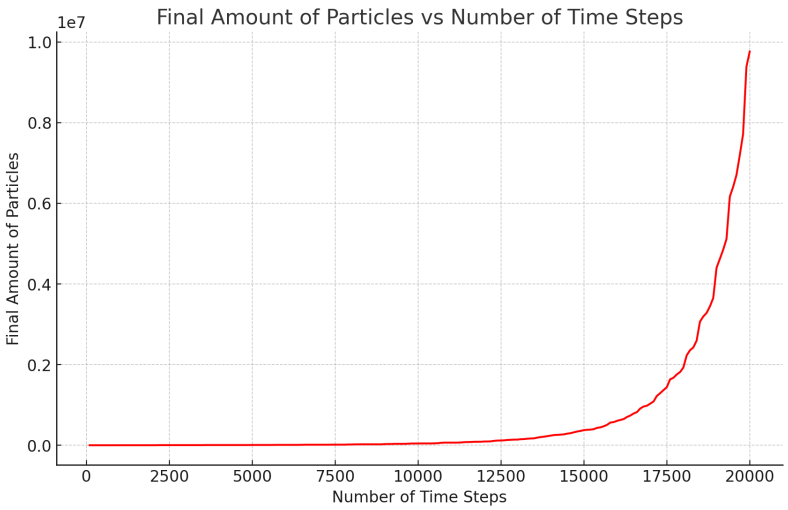

Figure 5.8: Final number of particles as a result of number of time steps

5.1.5 Overall Performance Assessment

In the ”real” physical simulation that is being worked towards in this project, the number of particles having to be simulated is assumed to be very large. When determining which of the schedulers are the best performing, it is therefore with this in mind.

Looking over the previous results it's clear that fully simulating a particle saves significant amounts of time. As a likely result of this, Static GPU Full is the best of the statically implemented schedulers.

The statically implemented Iterate-schedulers all appear to have worse running times with some variations among them. In figure (5.4) it can be observed they have an almost indistinguishable running time for more particles. However, by taking a look at figure (5.3), a slight performance increase can be seen when avoiding synchronising on the CPU and using the barrier included in the Cooperative Launch API. This results in GPU Iterate Multi Block Sync having a slight edge over the rest of the statically implemented Iterate-scheduler.

The performance of CPU Sync Full may initially appear surprising considering it's synchronising back to the CPU multiple times. However, this synchronisation only happens relatively few times during a simulation. For example, in section (5.1.4) the scheduler only has to synchronise back to the GPU 21 times to fully simulate more than $9 \cdot 10^6$ particles for 20.000 time steps. In contrast, CPU Sync Iterate would have to synchronise 20.000 times to fully simulate the same amount. Furthermore, while Static GPU Full uses busy waiting to wait for particles, there are no wait loops in CPU Sync Full. Lastly, among the few synchronisations and lack of busy waiting, the kernel itself is as simple as possible with minimal risk of warp-branching, that can cause inefficient execution patterns. All of this results in CPU Sync Full being the only scheduler able to keep up with the dynamically implemented.

Nevertheless, the dynamic schedulers have the best performances. Not having a predetermined set of particles to simulate and simply proceeding to the next available particles, shows to be very effective when comparing to CPU Static Full. Comparing Dynamic with Threads and Dynamic with Blocks shows that Dynamic with Blocks is the best. By that, the overhead added by synchronising threads on block-level must be smaller than the one created by having each thread using atomicAdd on a global variable.

Based on the results presented in this section, and as mentioned in 3.3, the schedulers chosen to make up the MVP are optimised versions of CPU Sync Iterate, CPU Sync Full, Static GPU Full and Dynamic with Blocks. For simplicity, are now referred to as Naive, CPU Sync, Static and Dynamic, respectively.

5.2 MVP - Johannes

To get at clearer picture of the performance of the selected schedulers for the MVP, similar tests as in section (5.1) are run on the updated simulation, with the included optimisations.

5.2.1 Effect of Block Size

To test the performance of varying block sizes for the MVP the following configuration was used:

Test Configuration

- Initial N: 10^6
- Time Steps: 10^4
- Split Chance (per mobility step): 0.02%
- Capacity: 10^8

From figure (5.9) is it clear that the Naive scheduler is the overwhelmingly worst. This does not make it uninteresting though. Of the four schedulers in the MVP, this is the only one that doesn't run with a GPU-specified number of blocks, but instead defines it based on number of particles. This results in more thread-blocks needed to be scheduled than the other schedulers. When comparing this scheduler to its less optimised counterpart CPU Sync Iterate, it appears that the optimisations implemented has led to a shift in optimal block size. Where in figure (5.9) the optimal block size was 128, it now is closer to 256/512. A reason for this might be the use of the shared memory counter described in section (3.3.1). The bigger the block size is, the fewer threads need to access the global memory, but like CPU Sync Iterate this can delay other threads.

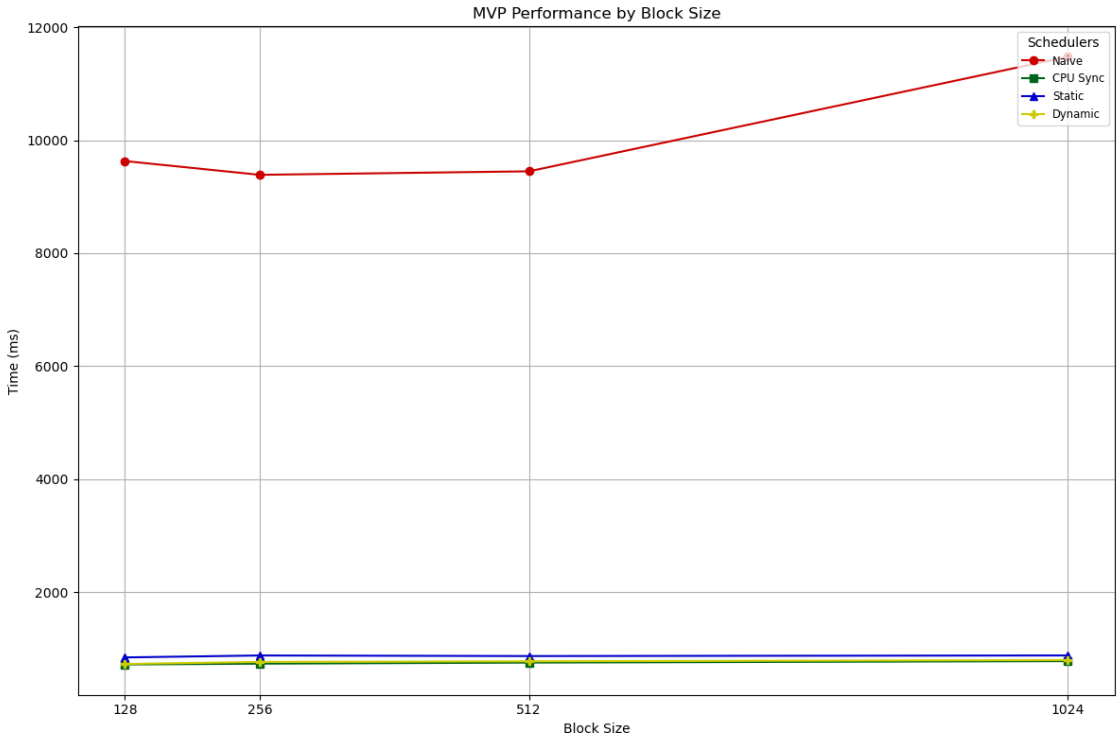

Figure 5.9: Running time as a result of block size

To analyse the effect of block size on the other MVP-schedulers, the Naive-scheduler is excluded in figure (5.10). For the remaining schedulers, the best block size is 128. Considering the Dynamic and Static schedulers fill out the SMs completely, this is to be

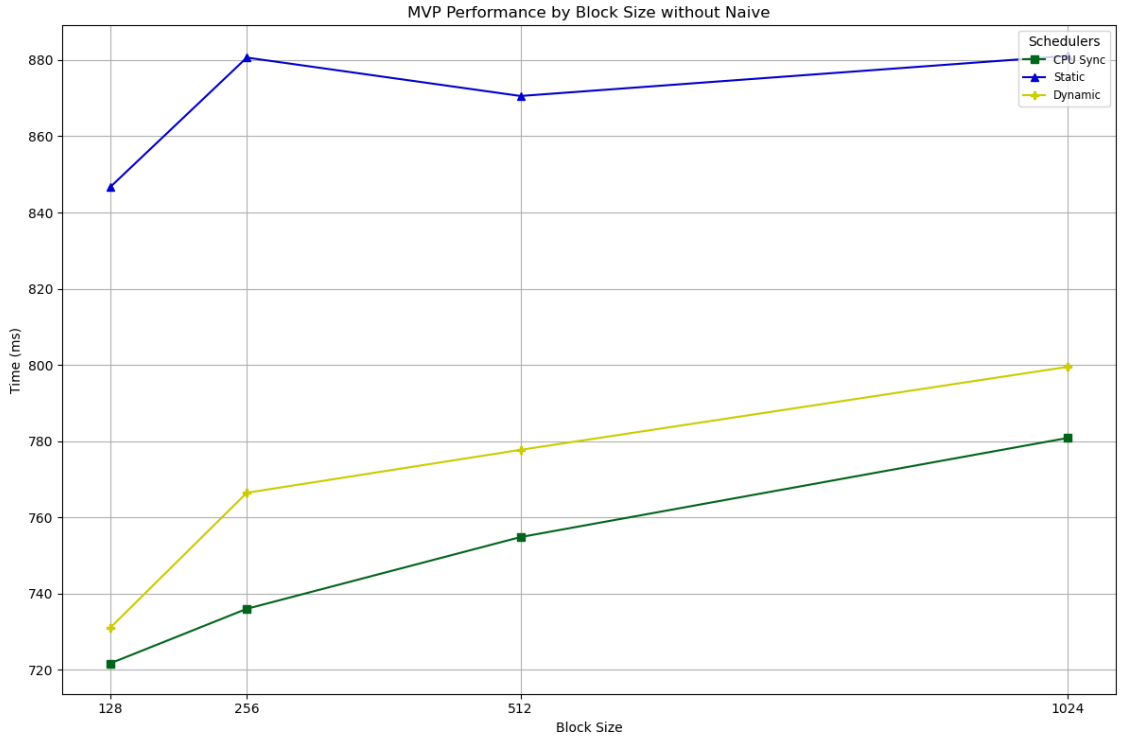

Figure 5.10: Running time as a result of block size (excluding Naive)

expected. By using all of the SMs, more thread-blocks can be executed at the same time. With lower block sizes, a slow thread slows down fewer threads with it.
For CPU Sync the optimal block size can be explained by the same reasons as for CPU Sync Iterate. No block-level cooperation is used while the entire GPU is utilised. This means that bigger block sizes make the impact of a slow thread more significant.

Based on these results, the following tests will be conducted using a block size of 128.

5.2.2 Increasing Initial Number of Particles

Testing the performance of the MVP-schedulers as a result of initial number of particles was done with the following configuration:

Test Configuration
- Block Size: 128
- Time Steps: 10^4
- Split Chance (per mobility step): 0.02%
- Capacity: 10^8

Again, the initial number of particles is starting from 100. *init_n* is then increased by 100 until 1000 is reached where after *init_n* is increased by 1000. The simulation is run like this until an *init_n* of 10^6 is reached.

As would be expected, the running times of CPU Sync, Static, and Dynamic schedulers are virtually constant until the SMs are filled where after a linear increase can be observed. Naive presents more unexpected results. This scheduler appears to be have a running time increasing linearly with the initial number of particles before the SMs are filled. This can be explained by the results in the following test.

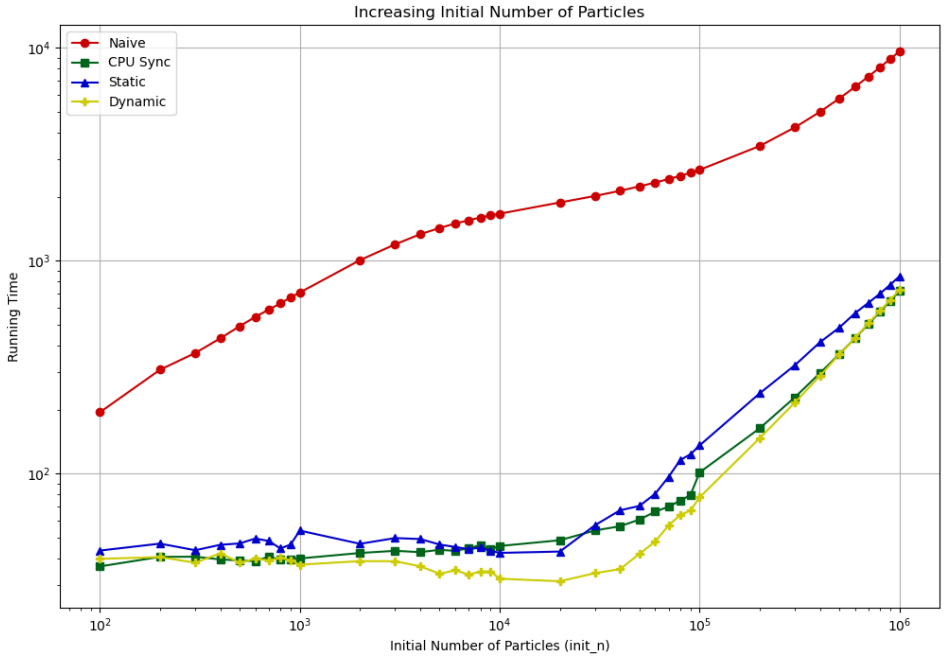

Figure 5.11: Running time as a result of number of initial particles

Without Simulating New Particles

To help understand in what part of the simulation most time can be saved, a test is conducted where only the initial particles are fully simulated. New particles are still spawned but these won't be handled. The test is run with the following configuration

Test Configuration
- Block Size: 128
- Time Steps: 10^4
- Split Chance (per mobility step): 0.05%
- Capacity: 10^8

Where *init_n* is increased the same way as in section (5.11).

In figure (5.12) the linear growth as a result of full SMs is even clearer. It is also clear that CPU Sync is the best scheduler here. When keeping in mind that CPU Sync only has to run multiple iterations if new particles have been spawned, this makes sense. This means, that for this test, CPU Sync only has to synchronise back to the GPU once after the initial particles have been simulated. The pure simplicity of CPU Sync makes it hard for Static and Dynamic to keep up.
It can be observed that Naive has a linear growth up until around there are 1000 initial particles, due to this scheduler updating each particle once per time step. The average amount of particles that split in each iteration can be calculated with:

$$average\_splits\_per\_time\_step = init\_n \cdot split\_chance$$

Before simulating the next time step, all threads synchronise on the CPU. Should a split therefore occur, all threads that didn't split need to wait for the slower thread handling the splitting particle. Theoretically, for $init\_n > 1000$, there should be a bigger chance for a

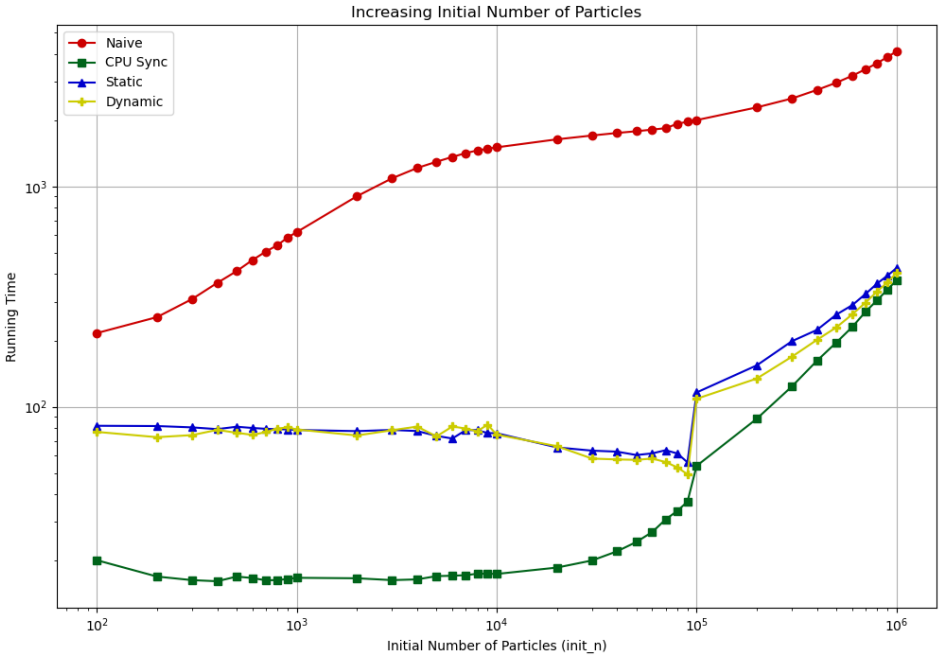

Figure 5.12: Running time as a result of number of initial particles without simulating new particles

split to occur per time step than not, because $init\_n \cdot 0.0005 > 0.5$ in this case. This would explain the linear growth in figure (5.12) until 1000 particles are being simulated and why the growth declines after.

5.2.3 Increasing Number of Time Steps

The running time as a result of number of time steps was tested with the following configuration:

Test Configuration
- Block Size: 128
- Initial N: 10^4
- Split Chance (per mobility step): 0.02%
- Capacity: 10^8

Starting from 500, the number of time steps was increased by 500 until $3 \cdot 10^4$ was reached. By logarithmically scaling the running time and number of time steps in figure (5.13), it is observed that the MVP follows the same trends as explained in section (5.1.4).

Without Simulating New Particles

Testing the effect of time steps when not simulating the newly spawned particles was done with the following configuration:

Test Configuration
- Block Size: 128
- Time Steps: 10^4
- Split Chance (per mobility step): 0.05%
- Capacity: 10^8

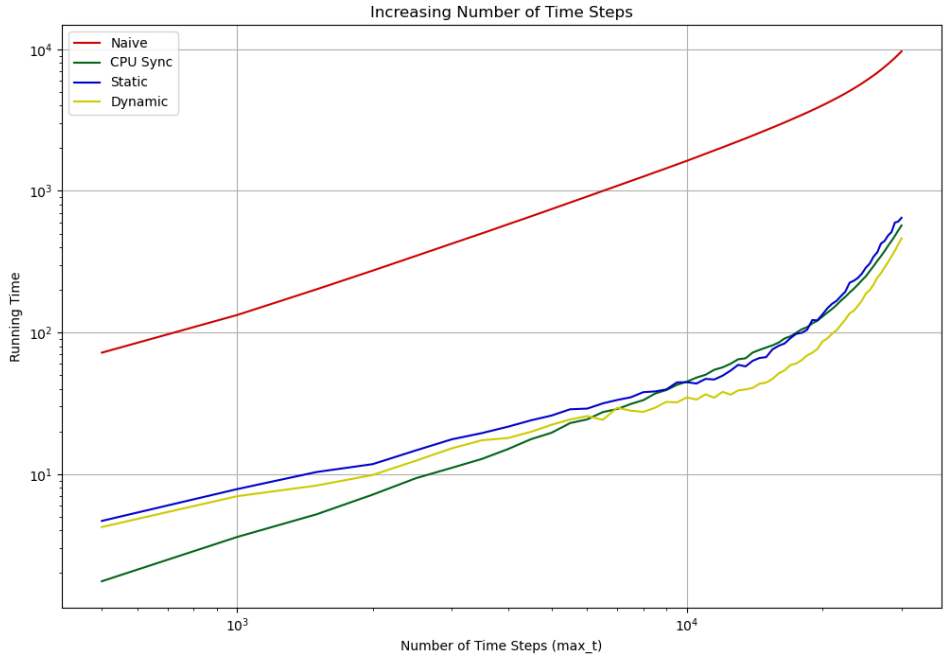

Figure 5.13: Running time as a result of number of time steps

From figure (5.14) it can be seen that the running time here only increases linearly with the number of time steps. This confirms that the exponential growth seen in figure (5.13) is a result of more and more particles being spawned.

5.2.4 Overall Assessment

To summarise the results for the schedulers in the MVP, it appears that the Dynamic and CPU Sync schedulers outperform the others. However, for further development and optimisation, the potential for the two schedulers differ. With the CPU Sync having a kernel as simple as possible, it leaves little room for refining and optimisations. This leaves back the Dynamic scheduler, being the most attractive for continuous improvements when implementing more physical complexities in the PIC.

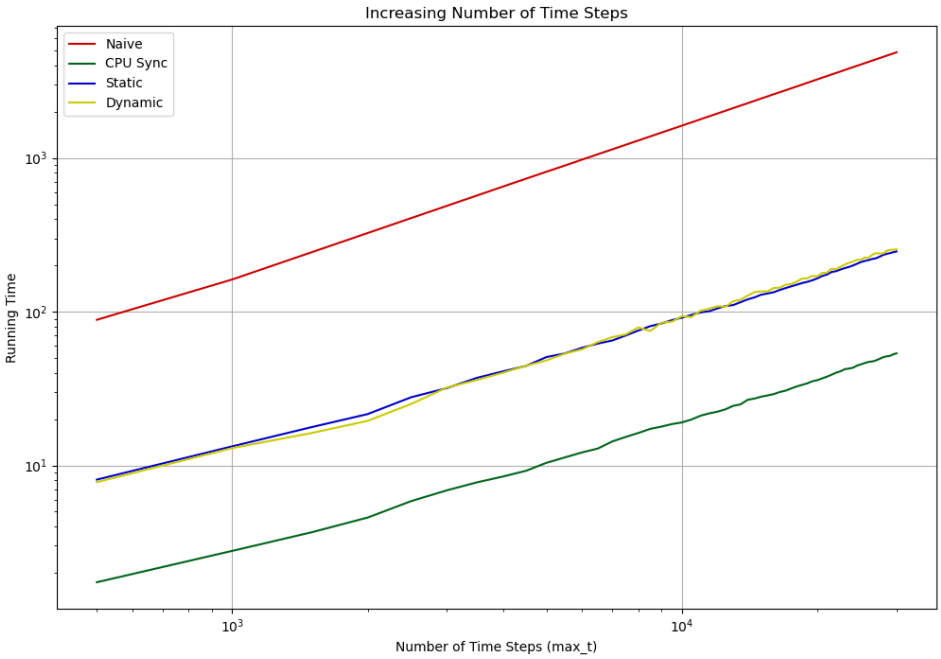

Figure 5.14: Running time as a result of number of time steps without simulation new particles

5.3 PIC - Magnus

For the final simulation, it is desirable to analyse the performance of the implemented solution in a variety of scenarios. This will provide insights to how various conditions influence the performance of the solution, hopefully guiding future work to improve the program. This is especially relevant in the context of adapting the code to be used in a ”real” physical simulation. A comparison to the three other schedulers from MVP (Naive, CPU Sync, and Dynamic (now referred to as Dynamic Old)) is included to help understand how the changes to the scheduler impacts its performance. This might also reveal situations where the older and more simple schedulers are better.

For all the tests done in this section, the values in the following box - as well as the cross section data described in (3.4.5) - have been used unless otherwise stated. It would of course be ideal to have as realistic values as possible, in regards to a real physical simulation, such as the one DTU Space is doing. Unfortunately there was not enough time to do the necessary research to achieve that. Furthermore, a real simulation would likely implement variable time steps (2.2.2), drastically changing the mobility steps value, as well as the chance for a collision in each step. This choice of cross section data (3.4.5) and mobility steps are loosely guided by discussions with our advisor. The hope is that they are roughly at the correct size, though mobility steps likely will be lower for most particles in a simulation with variable time steps. 1000 should however be a fitting value for some of the highest energy particles. This was chosen as the standard for the tests, as a high mobility steps value puts the highest pressure on the schedulers that this project has focused on refining.

Default Values
- Initial N: 10^6
- Poisson Timesteps: 10
- Mobility Timesteps: 10^3
- Capacity: 10^8
- Time Step Length: 10^{-12}

5.3.1 Cross Section Tests

As mentioned above, the cross section data used for these tests are likely quite different from those used in a real simulation. Therefore it is interesting to investigate how the schedulers behave with various collision chances, some of which might be better aligned with real values. For this purpose, a constant function was used for the cross sections, meaning the energy level of the electron has no influence on its collision chance. These tests examine the change in running time as the collision chance increases. The x-axis on all the following graphs show the combined chance for split and removal collisions (each of equal size as always) in percentage.

As it was not possible to have both 10^6 initial particles and 10^3 mobility steps when using collision chances above 10%, two versions of the test were run; one using 10^3 time steps (referred to as ”long”), the other using 10^2 (”short”). They used the following configuration values.

Cross Section Test Values (long)
- Initial N: 10^5
- Poisson Timesteps: 1
- Mobility Timesteps: 10^3

Cross Section Test Values (short)

- Initial N: 10^6
- Poisson Timesteps: 1
- Mobility Timesteps: 10^2

It turns out that the short version generally shows the same tendencies as the long version, but to a less extreme extent. Therefore the focus will be on the long version, but the same graphs can be found for the short version in the appendix (9.3).

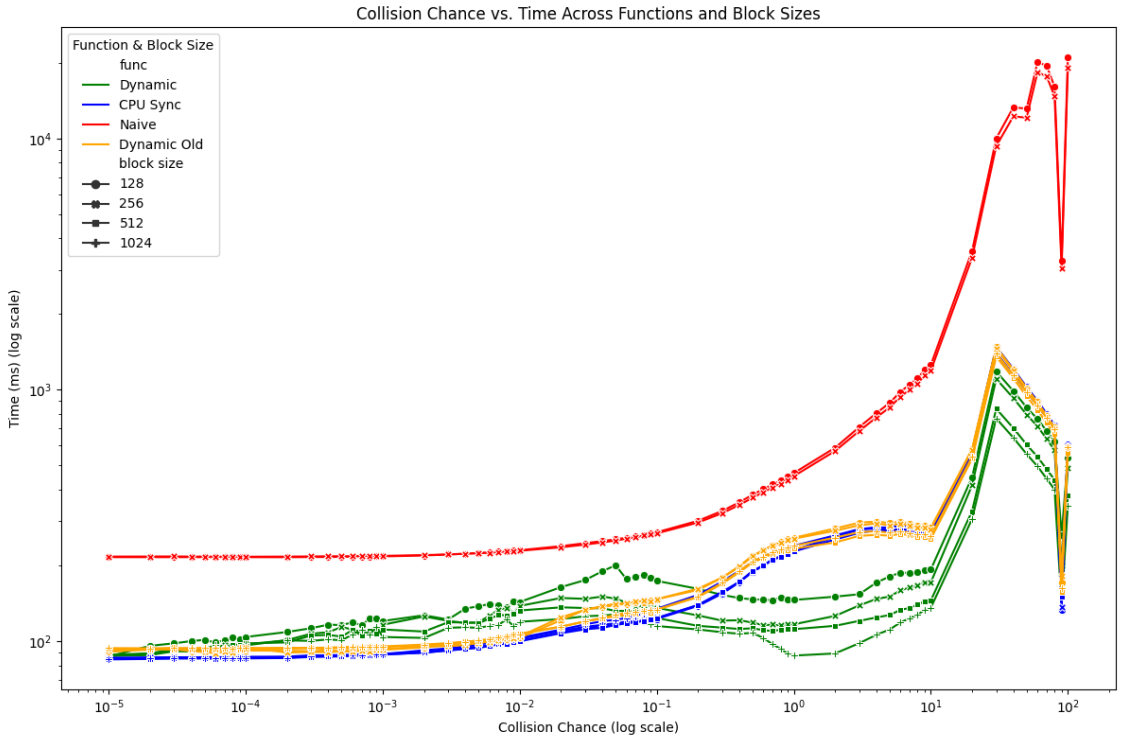

Figure 5.15: Cross Section Test Long

A weird phenomena occurs for all schedulers; after a collision chance of about 30%, the time decreases. They additionally all have an incredible dip at 90% collision chance. It was eventually discovered that this is due to a bug caused by improper initialisation of random states, as explained earlier (3.2). The good news is that the graphs make sense until that 30% mark, and we therefore believe that this hasn't significantly influenced any other tests or analysis, as they all are significantly below 30% collision chance (3.4.5).

As expected, the Naive scheduler performs consistently worse than the three others, as seen in figure (5.15). Though all schedulers get worse with increased collision chances, Naive explodes compared to the others. This is likely because a particle that has been removed still needs to be checked by naive every single iteration, as one thread is started for every particle every time, whether it is dead or alive. The same phenomena occurs in the mobility steps test later on (5.3.4).

To better analyse the more interesting schedulers, Naive are omitted in figure (5.16). Additionally, to get a better view of the performance at the lower collision chances, a figure is included that only shows chances below 0.1% (5.17).

In general, it can be clearly seen that a high block chance always is preferable for the new Dynamic scheduler, no matter the collision chance. This mostly makes sense, as

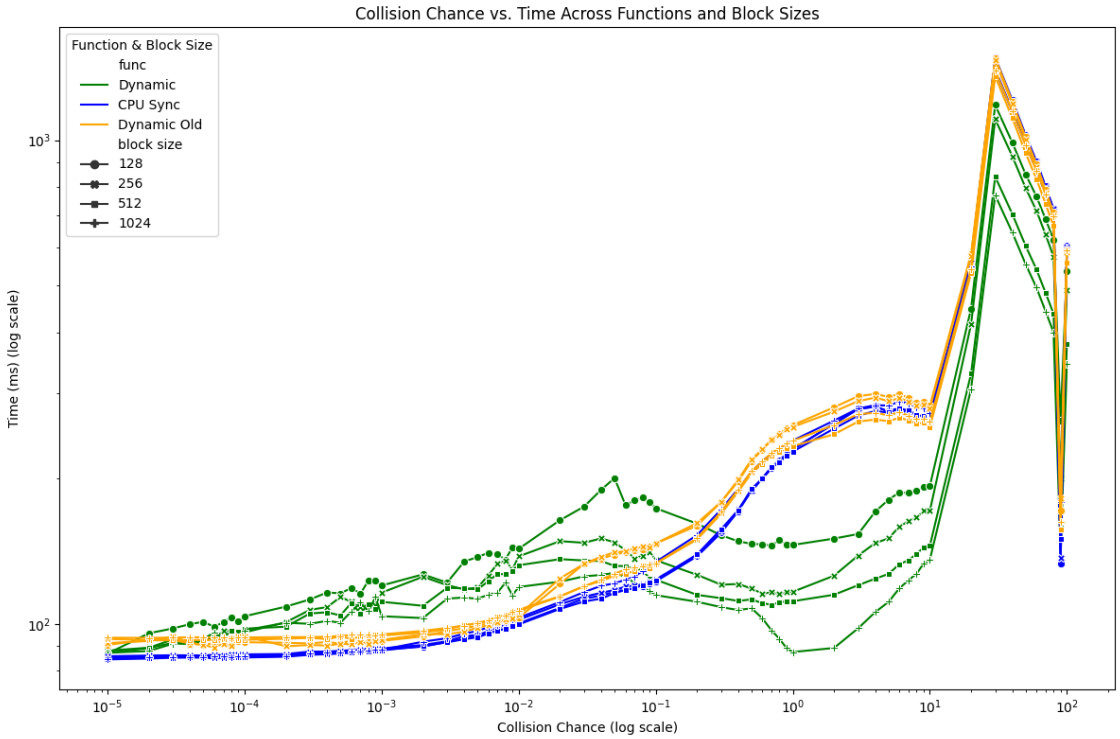

Figure 5.16: Cross Section Test Long, Without Naive

bigger blocks means more cooperation on uploading new electrons. An advantage of lower block sizes with high collision chance is that there will be fewer warps competing for access to the buffer, but this might be outweighed by the benefit of using relatively less shared memory on big block sizes. It can also be seen that CPU Sync and Dynamic Old behave almost identically, though CPU Sync does have an advantage in the lower collision chances, especially on the long version. This makes sense, as low collision chances means the kernel has to be run very few times to get through all the new particles.

When the collision chance is above 0.1%, the new Dynamic scheduler outperforms the other two, but this is reversed on the lower collision chances. This was surprising, as the opposite was expected. As explained in (3.4.3), the Dynamic scheduler risks a bottle neck on acquiring the lock to the shared buffer, which should favour lower collision chances. A hypothesis for this behaviour is that the performance cost of the additional computational operations required by warp synchronisation, buffers, locks, etc. in Dynamic is not worth it when collisions are so rare. When the collision chance is less than 0.1%, a given particle might not experience a single collision in 1000 iterations. Maybe only a few collisions will happen for an entire block. In such a case, CPU Sync only has to run a few times to do all its work, and Dynamic Old wastes very little time on desynchronised particle uploads and the like. It will also be rare for both of them to have particles in a warp that only need a few iterations before they are done; most of them will need to be simulated all 1000 steps. When the collision chance gets high enough, however, particles will frequently disappear, and they will require varying amounts of simulation steps to be done, as they will have been spawned at different times. These conditions allows Dynamic to shine. The combination of all these hypothesis could also explain why the schedulers become more similar double-digit collision chances - at this point, acquiring shared locks becomes a significant bottle neck, slowing down Dynamic considerably.

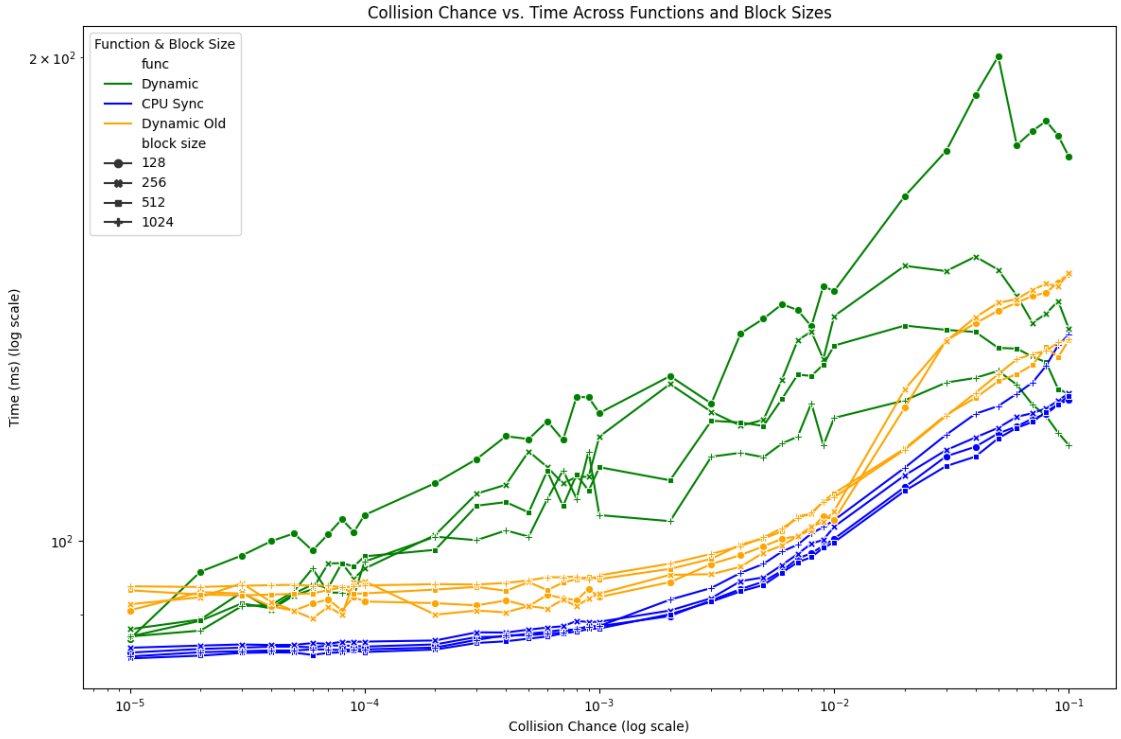

Figure 5.17: Cross Section Test Long, Low Chances

5.3.2 Block Size Test

For the remainder of the results analyzed in this section, each function has a clearly superior block size, and the varying performance of block sizes is not particularly interesting. To make the graphs less cluttered, only those superior block sizes will be used. A comparison of block sizes with the previously mentioned default values is presented here (5.18). It shows that using a big a block size as possible is ideal for all functions.

Note that Naive only goes up to a block size of 256. This is because electrons use more memory in this simulation, so the shared buffer that Naive uses is bigger. It now also has to include the random states. The bigger the block size, the bigger the buffer, and going to 512 or above will use more than is available.

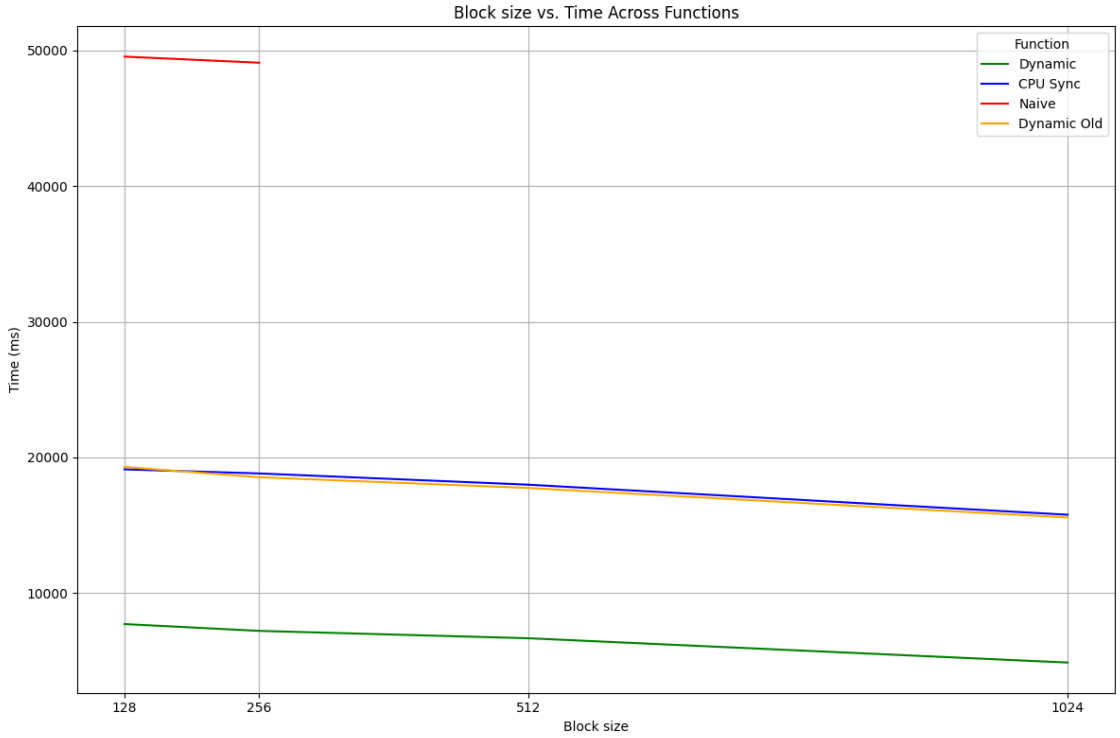

Figure 5.18: PIC Block Size Test

5.3.3 Poisson Steps Test

Varying the amount of poisson steps should have a mostly trivial impact on the running time of the simulation, as it essentially just increases the amount of times the same code has to be run. This is because split and removal chances always are equal, so the amount of particles present at the start of each poisson step is quite similar.

However, as more poisson steps are added, each particle has more time to accelerate to higher energy levels, changing their collision chances. Therefore, as context for analysing the performance cost of increasing poisson steps, average collision chance change from increasing poisson steps should be considered. (5.19) shows the amount of split collisions that occurs during a simulation with the given amount of poisson steps. Keep in mind that the amount of removal collisions will be almost identical. Had this graph been linear, it would have meant a constant collision chance, no matter the amount of poisson steps. Rather, the figure shows that the collision chance decreases as poisson steps goes up. This is likely because enough particles will pass the first peak on the cross section graph (3.1) if given more than one poisson iteration.

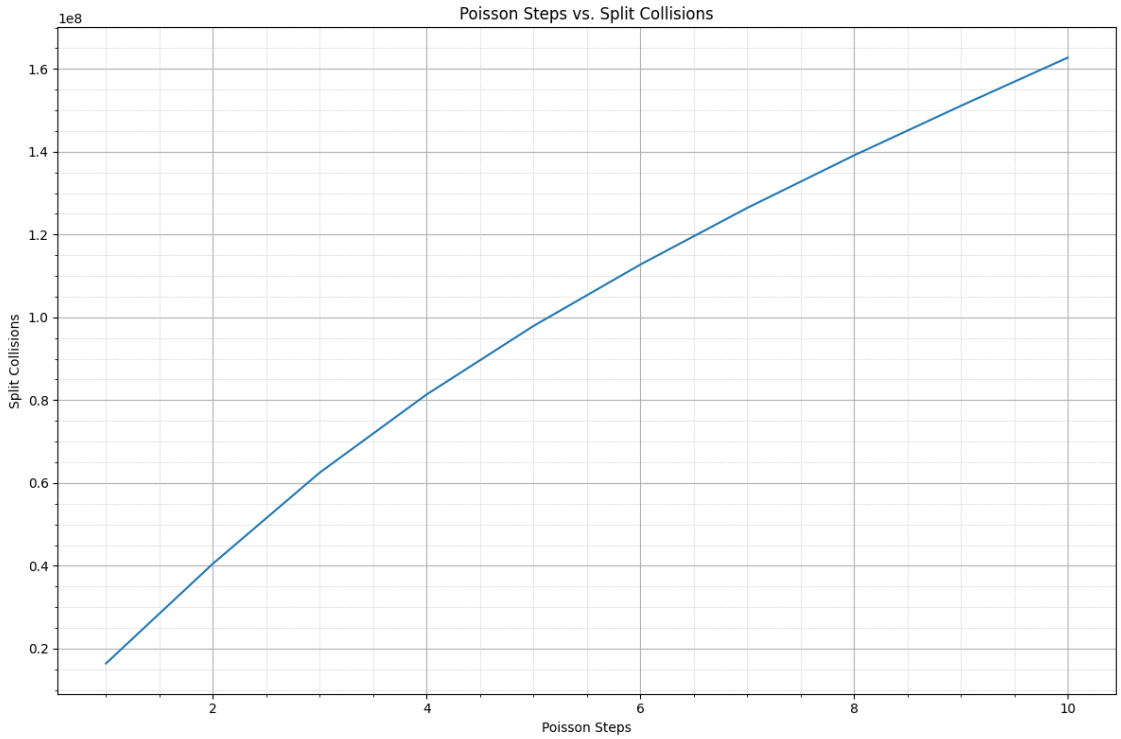

Figure 5.19: Split Collisions According to Poisson Steps

As the collision chance decreases with poisson steps, it should be expected to see a less-than-linear increase in the simulation time as the poisson step count increases, due to decreased time spent loading particles. As can be seen 5.20, this is exactly what happens for all schedulers. See appendix (9.4) for Dynamic only.

The downwards curvature could additionally be explained by the particles being spread out more in the grid as more poisson steps are run, lowering the bottle-neck of performing atomic operations on the grid cells when mapping particles to the grid. This should be a very minor factor, though.

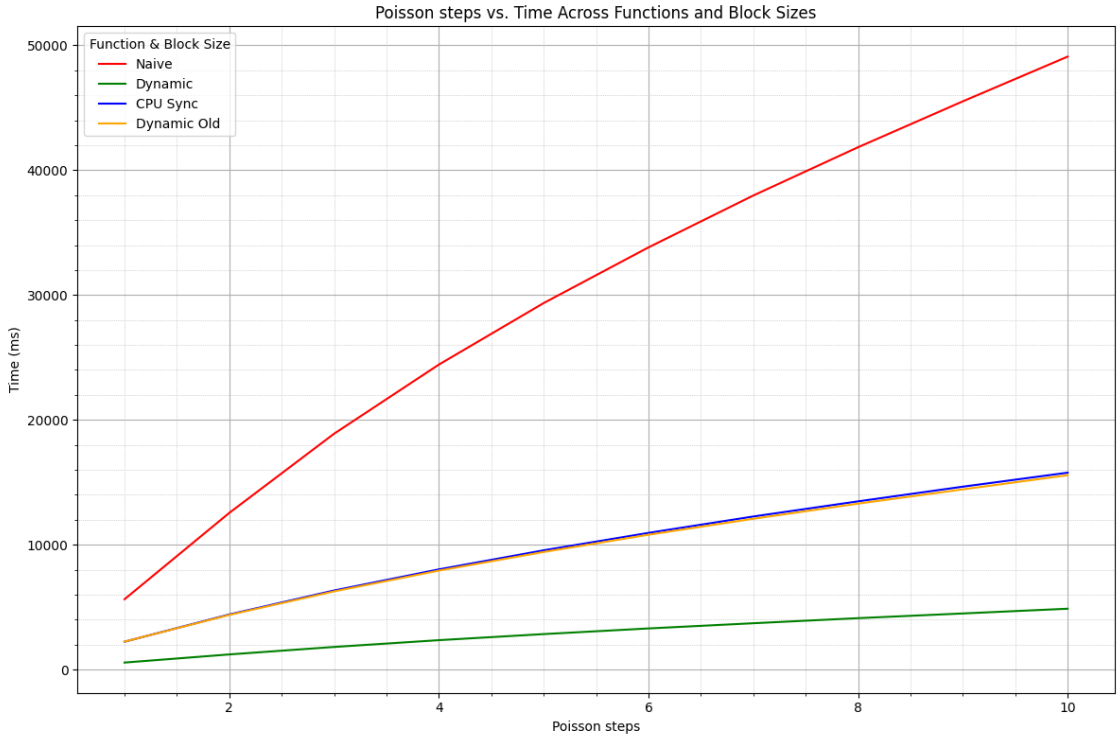

Figure 5.20: Performance According to Poisson Steps

5.3.4 Mobility Steps Test

The other factor contributing to total iterations is mobility steps, which in this test was varied from 10 to 10^3. For clarity they are split into two figures with linear axis scales, one showing mobility steps 10-100 (5.21) and the other showing mobility steps 100-10^3(5.23). See appendix (9.5) for Dynamic only. As before, they are respectively supplemented by graphs showing collision count increases (5.22) and (5.23). The latter of the figures displaying collision count is almost identical to its poisson step counterpart (5.19). This is because they display the same range of total iterations performed.

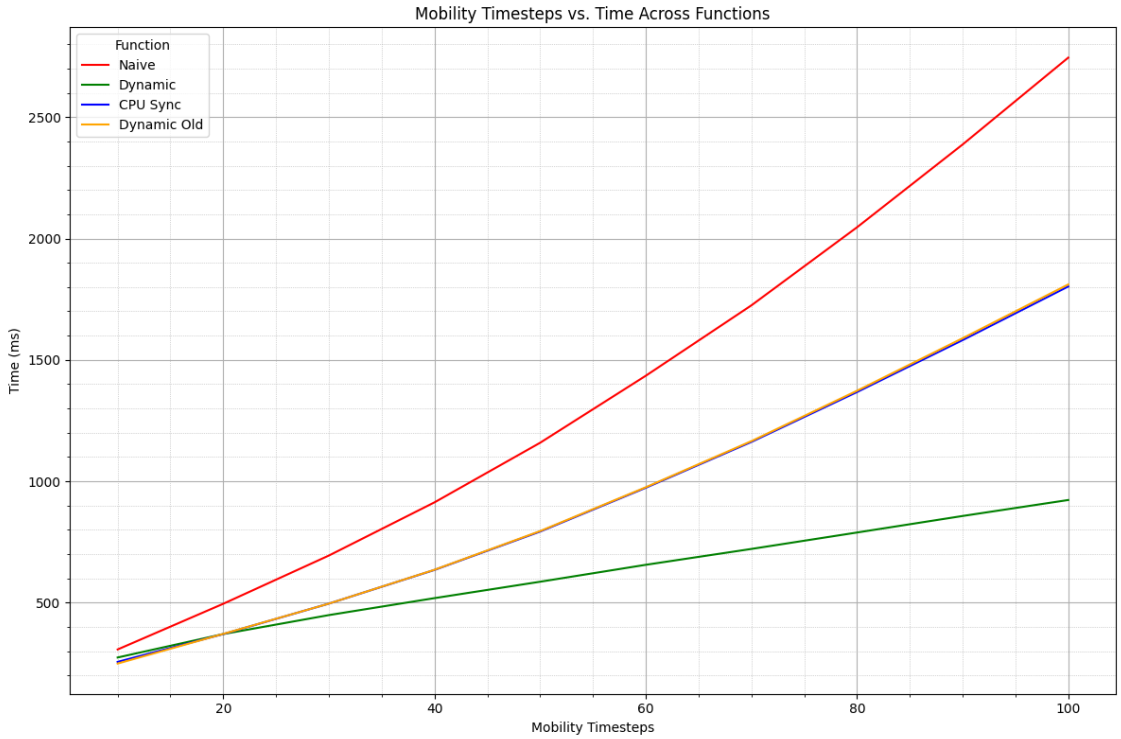

Figure 5.21: Performance According to Mobility Steps (Low)

The graph displaying collision counts for mobility steps 10-100 instead shows an increasing curvature. This likely represents the majority particles moving from energy level 0 towards the first peak on the cross section graph (3.1).

By the same logic as described in the poisson steps analysis, the expection is for the simulation time graphs to display roughly the same curves as the corresponding collision count graphs. This is indeed the case for CPU Sync and Dynamic Old, which as usual have nearly identical performance. Naive, on the other hand, curves slightly upwards in the latter part, even though it should benefit greatly from the lowered collision chance, as alluded to in the cross section tests (5.15) and poisson step tests (5.20). The reason that increasing mobility steps is much worse for Naive than increasing poisson steps might be that there is no cleanup between mobility steps; in every iteration, a thread must be assigned to every particle, whether it is dead or not. Therefore, for Naive, the total number of threads needed for each iteration increases as more iterations are performed. Dynamic is nearly linear in the lower range. This might be due to very low collision chances not being very favourable for it (5.3.1), so the increase does not hurt it much.

From this test and the one varying poisson timesteps (5.20), it can be concluded that if the other variables - including cross section data - are similar to what was used in these

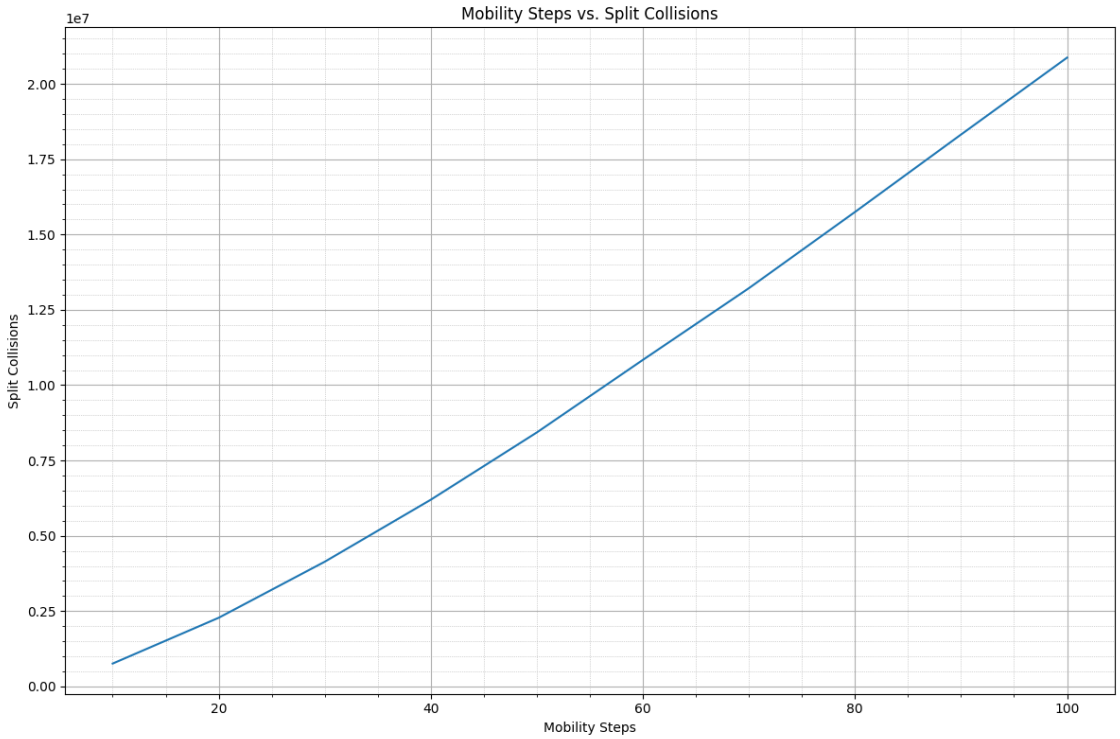

Figure 5.22: Split Collisions According to Mobility Steps (Low)

tests, Dynamic behaves very similarly to the other schedulers when varying the length of the simulation in either way, and it is always superior.

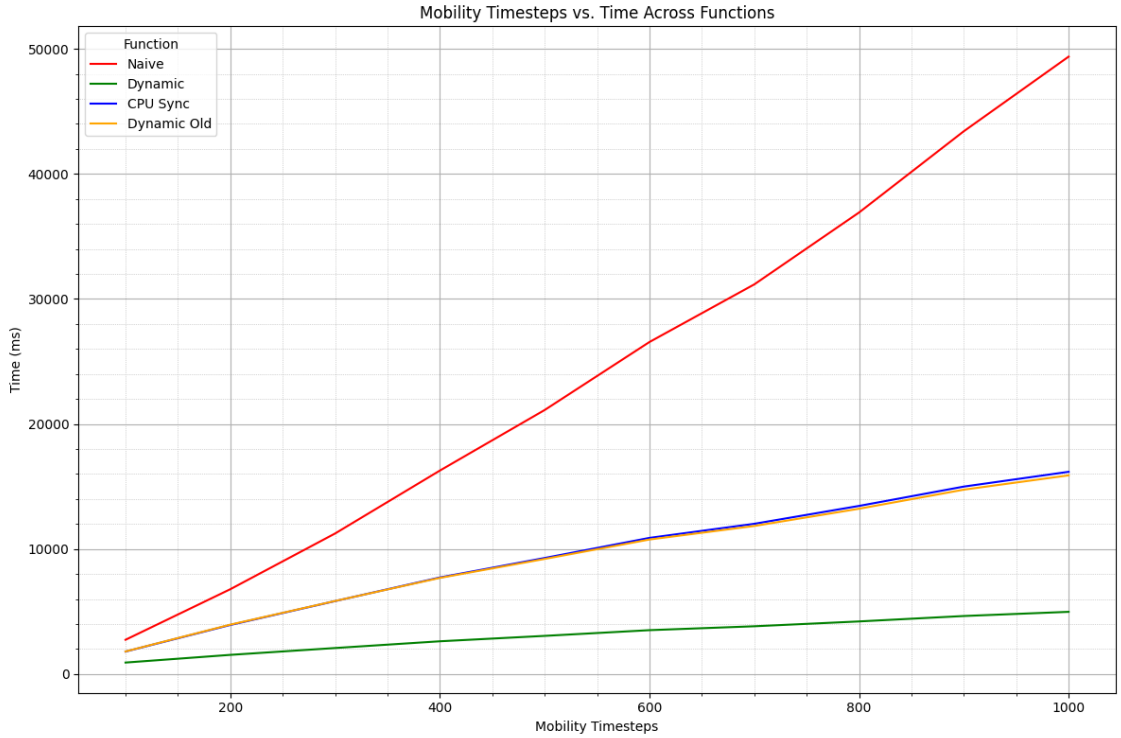

Figure 5.23: Performance According to Mobility Steps (High)

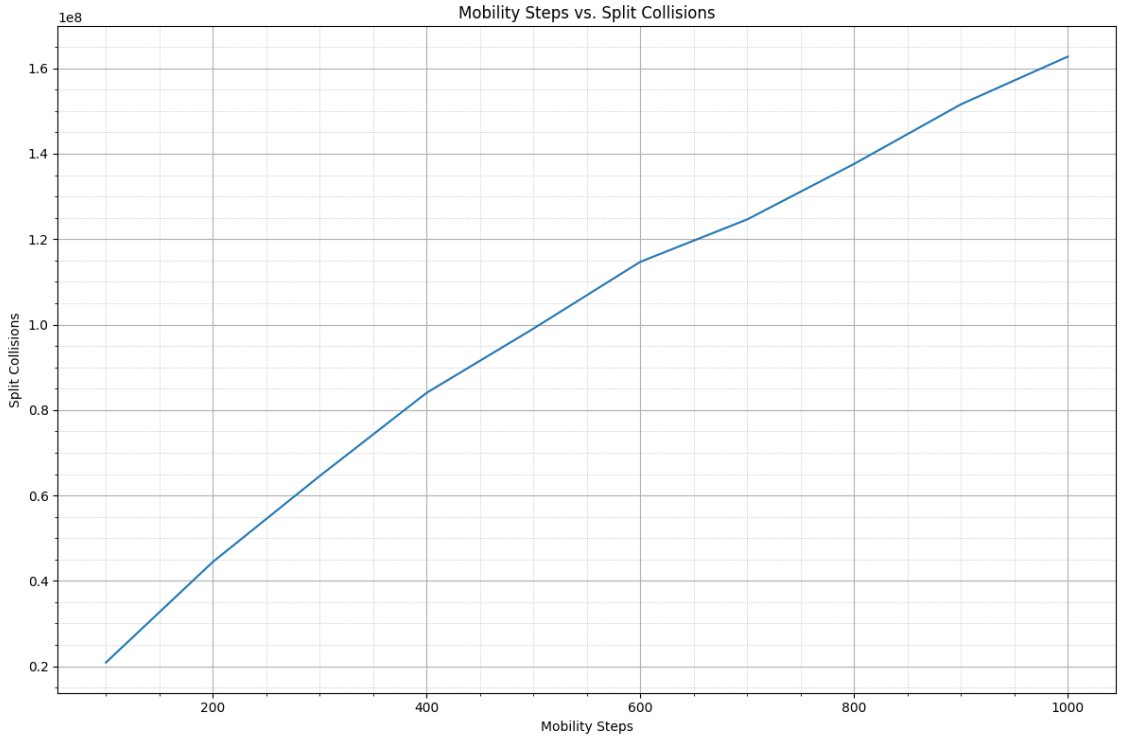

Figure 5.24: Split Collisions According to Mobility Steps (High)

5.3.5 Initial N Test

For the final test, the initial amount of particles was varied between 10^4 and 10^6. As before, they were split by 10^5 (5.25) (5.27), each with a corresponding collision count graph (5.26) (5.28). See appendix (9.6) for graphs showing only Dynamic. The same logic as stated in the previous sections regarding collision chance development applies here. When the collision chance increases due to an increase in particles, it is because of the increased density of particles, leading to higher acceleration.

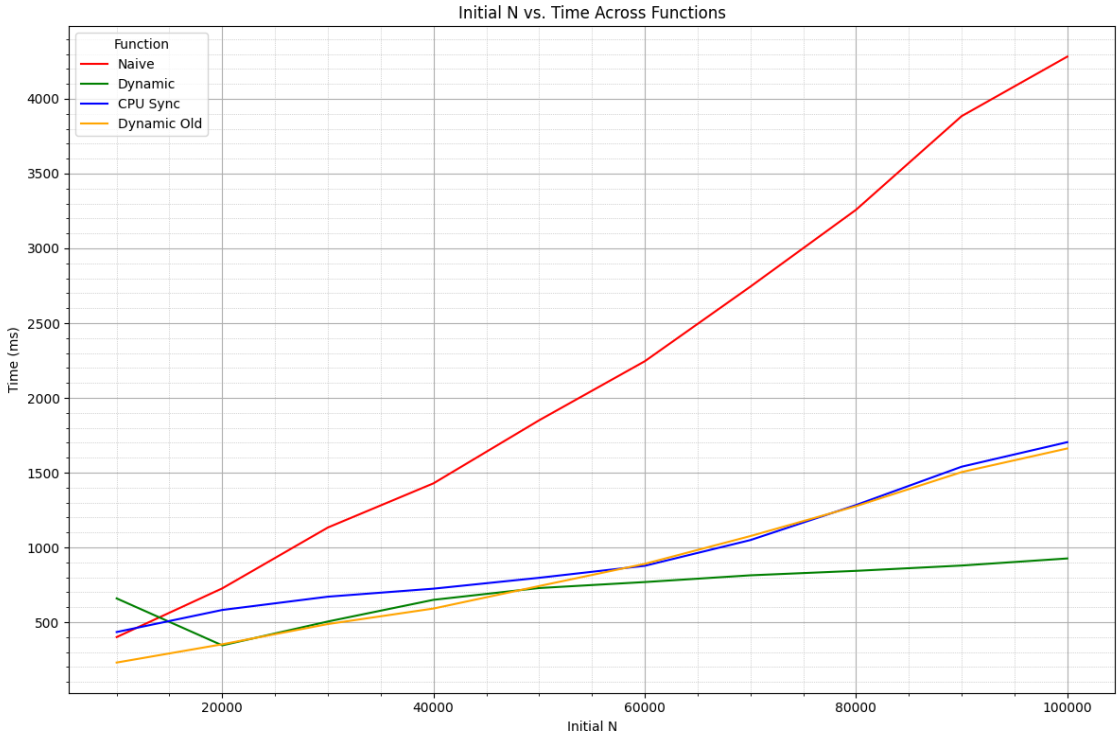

Figure 5.25: Performance According to Initial N (Low)

The interesting difference is to be found in the lowest end of the range of initial particles, where CPU Sync is different from Dynamic Old, Naive beats everything but Dynamic Old, and Dynamic has a spike. The spike of Dynamic is likely an overhead cost of the complexity applied to a problem with very few particles and collisions. Many threads will be started only to spend many operations not achieving anything. It is hard to say why exactly this would be more expensive than simulating more particles, but it might be related to congestion on atomic operations caused by such a high amount of threads spinning without doing anything at the same time. In any case, such a low amount of particles is not particularly interesting for this application.

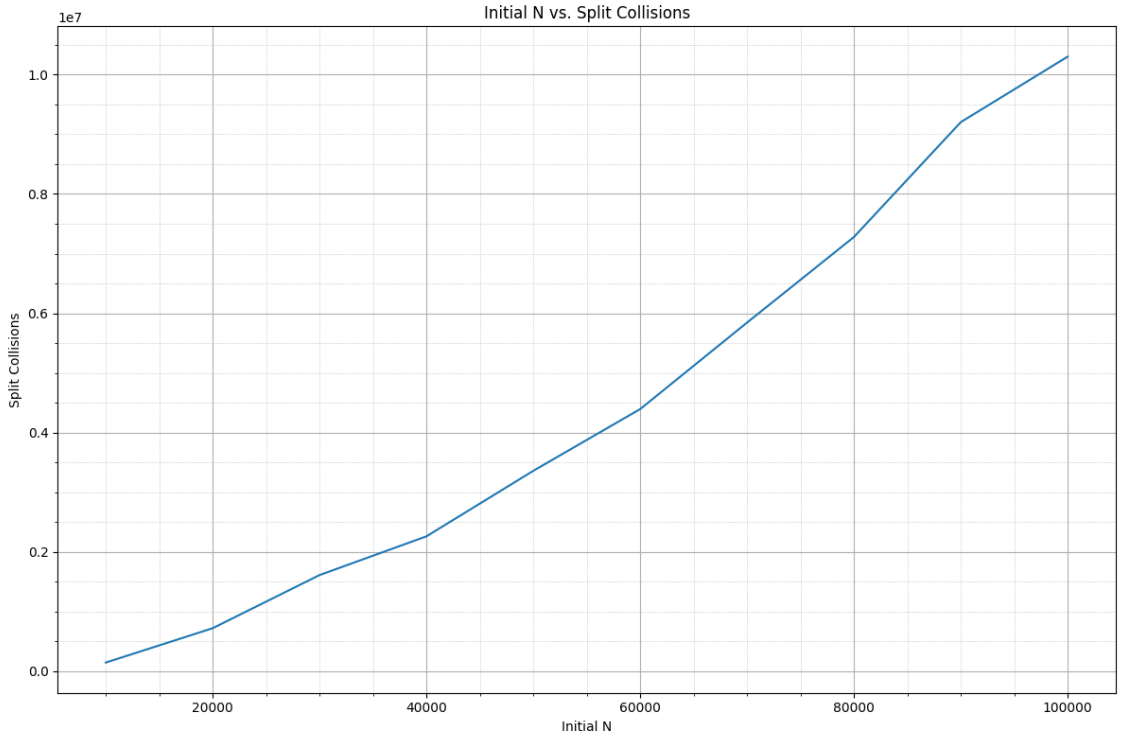

Figure 5.26: Split Collisions According to Initial N (Low)

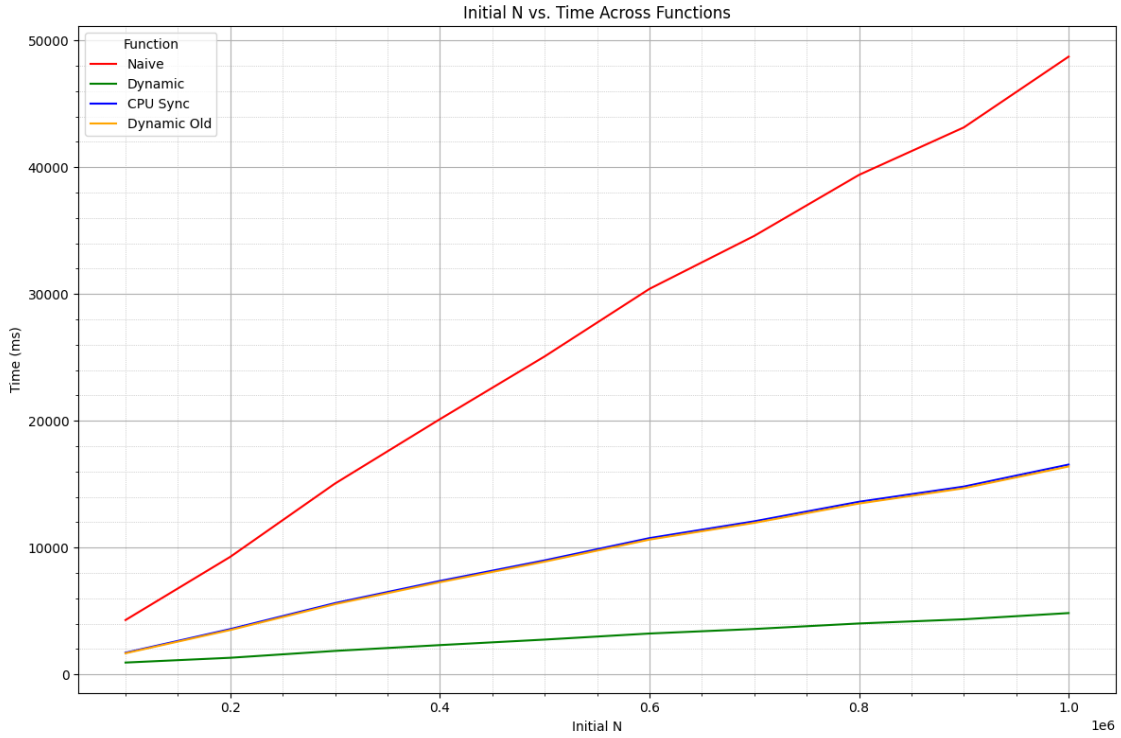

Figure 5.27: Performance According to Initial N (High)

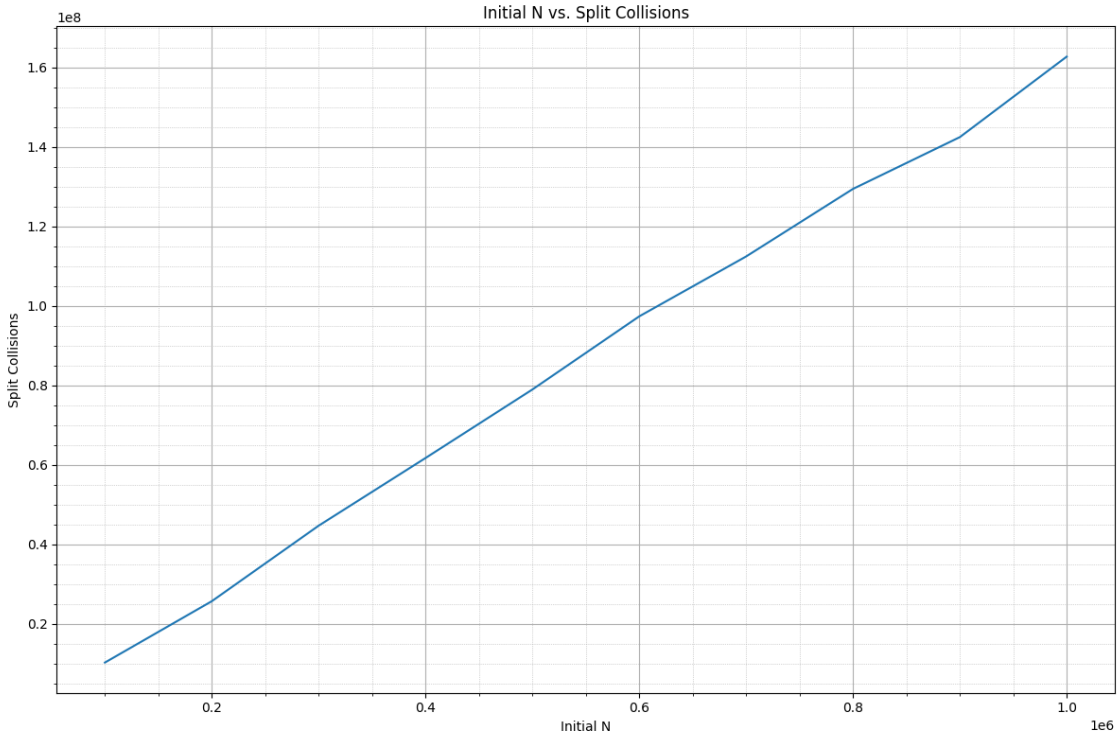

Figure 5.28: Split Collisions According to Initial N (High)

6 Conclusion

A variety of approaches to kernel design for particle simulation were implemented and tested, and their distinct performances, advantages, and disadvantages were analysed. A few of them were chosen to be refined and analysed further. It was shown that the choice of design and usage of various optimisations to utilise the GPU fully have an enormous effect on performance. In the end it can be concluded that for a particle simulation similar to the one being designed by the research group at DTU Space, a kernel utilising persistent dynamic scheduling to distribute the particles among the threads works best. This allows some threads to take their time on particles that demand a lot of processing power, while letting other threads run through many of the other electrons at the same time. It also leaves room for extensive optimisations to maximise the efficiency of each thread and the memory of the GPU at various levels, despite the complexities introduced by addition and removal of particles. The final simulation provides a solid foundation for the implementation of a fully-fledged simulation.

7 Future Work - Magnus

This chapter covers the additions, changes, and research we would do, if we had more time to work on the project. Should anyone else wish to continue the project or use it for something else, this covers our recommendations for where to start and take it. It specifically concerns the PIC simulation (3.4).

7.1 More Physically Accurate Simulation

To make the program useful for actual physical particle simulations, such as the one developed by DTU Space , the following changes should be made.

1. **Poisson solver.** Implement a real poisson solver to calculate the forces generated by each cell in the grid.
2. **Variable mobility timestep.** Though technically an optimisation, it would be advisable to implement the variable time steps and null collision checking (2.2.2) early, as this might change the behaviour of the schedulers significantly.
3. **Electron acceleration.** Consider making electron acceleration more precise by updating the acceleration multiple times during a poisson step, likely by reading a new acceleration value from the grid if the electron moves into a new cell during the poisson step. Additional precision can be gained by interpolating the particle's acceleration from its current cell and the nearest neighbouring cell.
4. **Expand collisions.** Split and removal correspond to somewhat to real collision types, but they do for example not have correct scattering angles of the particles, which is necessary for a correct simulation. There are also many other types of collisions that should be implemented. Additionally, they all need cross sections with real data. The code for checking cross section data should also be modified to support many more types of collisions.
5. **Proper constant values.** Many constants used in the project are real values, but others were just chosen to fit this implementation. This is for example the cell size used for the grid and the grid size itself. These should of course be modified.
6. **Choose a scheduler** With these changes done, run a simulation with realistic values for all of the schedulers, and choose the one that performs best. According to our tests and analysis (5.3), this is very likely to be the new Dynamic scheduler, but it will depend on the final implementation.
7. **Remove determinism.** Finally, consider letting go of determinism in favour of verifying the correctness of the simulation through more qualitative comparisons to implementations that are known to be correct, for example by studying a density map of particles during the simulation. This will increase performance by reducing the amount of random states allocated and not needing to move them around and initialise new ones during the simulation. At the start of the simulation, simply initialise one state for each thread, each with the same seed and with the thread id as the sequence.

7.2 Optimisations

There is plenty of room for optimisations to be made to the code, especially outside the realm of the mobility step simulation kernels. This list represents areas where we suspect

there are big performance gains to be made, though we can not know exactly how much and how to implement it, as we of course haven't tried and tested it ourselves yet.

- **Avoid two electron arrays.** The kernel that removes dead particles makes use of two arrays, one to read from and one to write to (3.4.2). It should be possible to reduce this to one array, which will save a considerable amount of memory.
- **Optimise cross section lookups.** This involves many potential improvements. One is to cache the collision chances for a particle for as long as its energy level stays the same. Another is to sort the particles by energy when removing the dead particles, allowing a warp or block to coordinate a lookup of adjacent - or possibly even just one - cross section values. This might not be possible if the electrons have very different accelerations, but perhaps that could be accounted for somehow.
- **Optimise simulation syncs.** As was shown in some schedulers in Scheduler Tests (5.1), it is sometimes advantageous to force the warps to sync after branching code. One could experiment with syncing at various times throughout the simulation to see if it improves performance.
- **Coordinate loading and uploading of particles in Dynamic.** As explained previously (3.4.4), there is potential to avoid desynchronised loading and uploading of particles in the new dynamic scheduler.
- **Change block sizes.** For simplicity of testing, we have used the same block size for all kernels, but this is not necessarily ideal. It is for example likely that remove dead particles will benefit from a small block size, while the simulation schedulers benefit from big block sizes.

8 Project Management

8.1 Old Project Plan

The approach to the plan of this project is based on the idea of a continuous cycle of implementation, testing, and learning. Each cycle builds on the results of the previous one and introduces more complicated code which allows us to test more complicated concepts. The testing stage involves developing multiple approaches to solving the same problem and bench marking them.

The plan is divided into 5 overall phases, each containing various activities. There is also the additional ”Project Plan” part, which simply is to create this plan. The plan can be seen at the end of the document specifying how many weeks are approximated for each task. Some tasks are marked as milestones, which we consider ”final simulations” that could constitute a product. They build on each other, getting progressively more complicated. The features they introduce have been ordered in terms of how important they are to us to implement. Below is an elaboration of the Gantt-diagram in appendix (9.11), which shows the time span planned for each task.

Intro to CUDA
The first phase is about getting set up and familiar with CUDA and in general programming on GPUs, which is completely new to us. To do this, the focus will be on:

- **HPC Setup:** Connect to DTU’s VPN, the HPC system, set up environment and repository, and execute some code on the HPC system.
- **Simple Simulaiton:** We plan to implement a simple simulation to help us learn how to program in CUDA. While implementing this, we will study the architecture of GPUs and the advantages/disadvantages GPU programming brings when compared to traditional CPU programming. The simulation will feature particles (represented by coordinates) moving around in a simple gravity field.

Kernel Experiments
There are many ways to go about running kernels and synchronising them. In this phase we plan to learn about, implement, test, and evaluate a host of different techniques such that we can utilise them in later phases and choose the right tool for those simulations. This phase includes the code that will constitute the minimum viable product for the project. Once this is made, and we also have done thorough testing and analysis and written a report, we consider the project succeeded, as the learning objectives have been met. Everything else is ”nice to have”.

- **Adding particles: Milestone 1 (MVP):** We plan to implement a simple simulation based on the previous one that can add new particles over time. More importantly, we will implement many different ways of scheduling the kernels and running the simulation. We then bench mark them, evaluate, and compare the results. This is the minimum viable product for this project.
- **Removing Particles:** The next step is to also be able to remove particles dynamically. This will likely require rethinking the data structures such that it works with multiple kernel scheduling paradigms.

Simulation Development
Based on our kernel execution results, we will begin to work towards a more physically correct simulation. These simulations will build on each other and gradually resemble

actual PIC-MCC simulation more and more, though many parts will be mock-functions to avoid the complicated physical details. As these all come after the MVP, none of them are crucial to get done. They will be skipped if need be, so we can get the steps of the next phase done.

- **Static PIC: Milestone 2:** The first step is to restructure the simulation to be more akin to a PIC simulation. This involves two nested loops and structuring the space into a grid. Each cell in the grid puts force on the particles in it, but the cells are static and do not change according to the particles.
- **Variable Time Steps:** In a physically correct simulation, the inner nested loop runs at varying, random time steps. This will be implemented here.
- **Dynamic PIC: Milestone 3:** Now the cells will change according to the particles within them. This is a mock simulation and will not be physically correct.
- **Dynamic PIC: Evaluation:** A larger step back to evaluate how our kernel techniques perform in the current simulation. It will involve rigorous bench marking. Even if we do not have time to do all the simulations of this phase, this evaluation will still be done of the last simulation we make.

Final Implementation
The last stretch of the project where we finalise the simulation.

- **Full Physics Implementation:** A simulation as close to being physically correct as we have time for, based on the lessons from all previous phases. This will be almost exactly the same simulation as the previous one, but where the mock-parts are replaced with physically correct calculations. These might just utilise third party code. This is also not part of the MVP for the project.
- **Final Touches:** Final optimisations, code tidying, etc.

Report
Though we will work on the report throughout the project and document our results, this is the part where we dive deep on getting it done.

8.1.1 Our Learning Objectives

If we only hit the MVP, as explained above, our learning objectives will still be fulfilled as follows.

- **Gain experience in working with GPUs in the context of high-performance computing.** Especially the two first phases involve gaining HPC experience for GPUs.
- **Understand and examine programming patterns suitable for GPU hardware.** The second phase is dedicated to exploring kernel scheduling paradigms.
- **Be able to explain the overall architecture of GPUs.** In our research of kernel scheduling techniques, and in writing the report, we will dive into GPU architecture.
- **Be able to explain the overall concept of how parallel code executes on GPUs.** After implementing, testing, evaluating, and documenting our code, we will have a solid understanding of parallel code.
- **Profile and analyze the performance of critical areas within an application.** Bench marking and analysing algorithm performance will fulfil this.

8.1.2 Official Learning Objectives

- **Independent planning and time management** is achieved through this project plan as well as the revisions to it throughout the project.
- **Solving technical problems and understanding information.** We are facing some challenges with utilising the full potential of GPU hardware when trying to achieve particular goals.
- **Working with all phases of the project and documenting our work** We will attempt different programming techniques, test them rigorously, and document our findings.
- **Learning new and thinking independently** GPU programming is completely new to us, and will thus involve learning a lot of new things.
- **Written and oral documentation** The entire process will be documented and evaluated in a report. The project will furthermore be defended at an exam.

8.2 Revised Project Plan

The steps of the plan were followed as well as possible. As a result of everything taking longer than expected, we very quickly realised that the time set for each step would not fit. This was not a bad thing though.
The Kernel Experiments, that were supposed to be finished in week 6, along side the MVP, ended up lasting until week 8. This extra time was spent fixing and optimising the kernels based and benchmarks performed during this period. Despite the frustrations feeling a bit behind might have caused during the time, looking back at it, it was time well spent as it resulted in a deeper and better understanding of how to write efficient CUDA programs. The phases after were followed more or less as planned. This lead to us having less time to get as close to full physically accurate simulation as we wanted.

8.2.1 Learning Objectives

As a result of us, for the most part, following our plan, we managed to complete the majority of the goals we set for ourselves and by that fulfill the learning objectives presented.

8.3 Work Distribution

To develop most of the schedulers presented in this project, we used the agile software development technique known as Pair Programming. Day to day we would take turns acting as driver and navigator. This ensured that we both kept a good understanding of the functionality and ideas behind each scheduler and the way they were implemented. The final responsibility for the code is divided in the same way as the report - Scheduler Tests and MVP for Johannes, PIC for Magnus.

9 Appendix

9.1 Background

Figure 9.1: Volta SM

9.2 Scheduler Tests

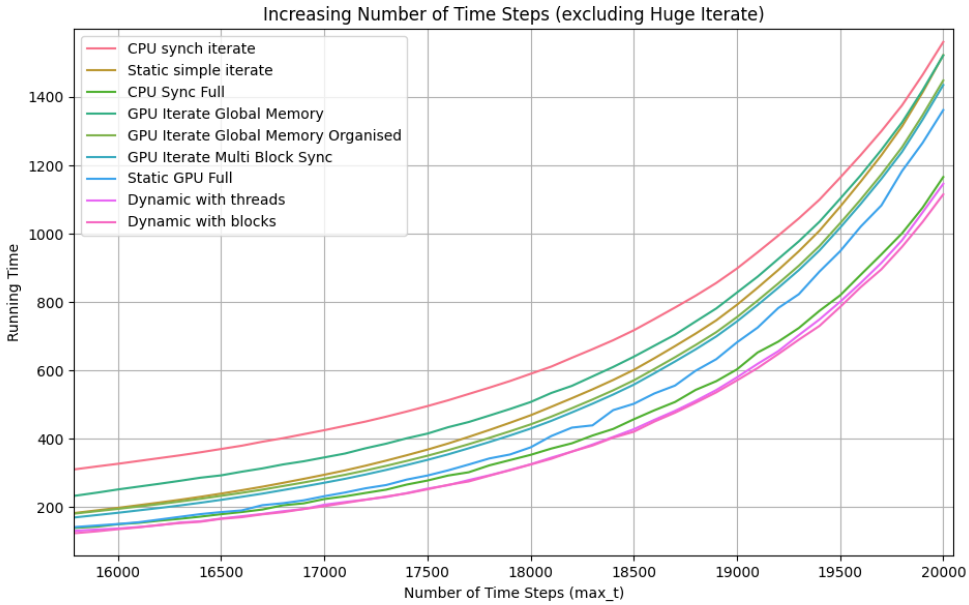

Figure 9.2: Running time as a result of time steps (zoomed)

9.3 Cross Section Test Short

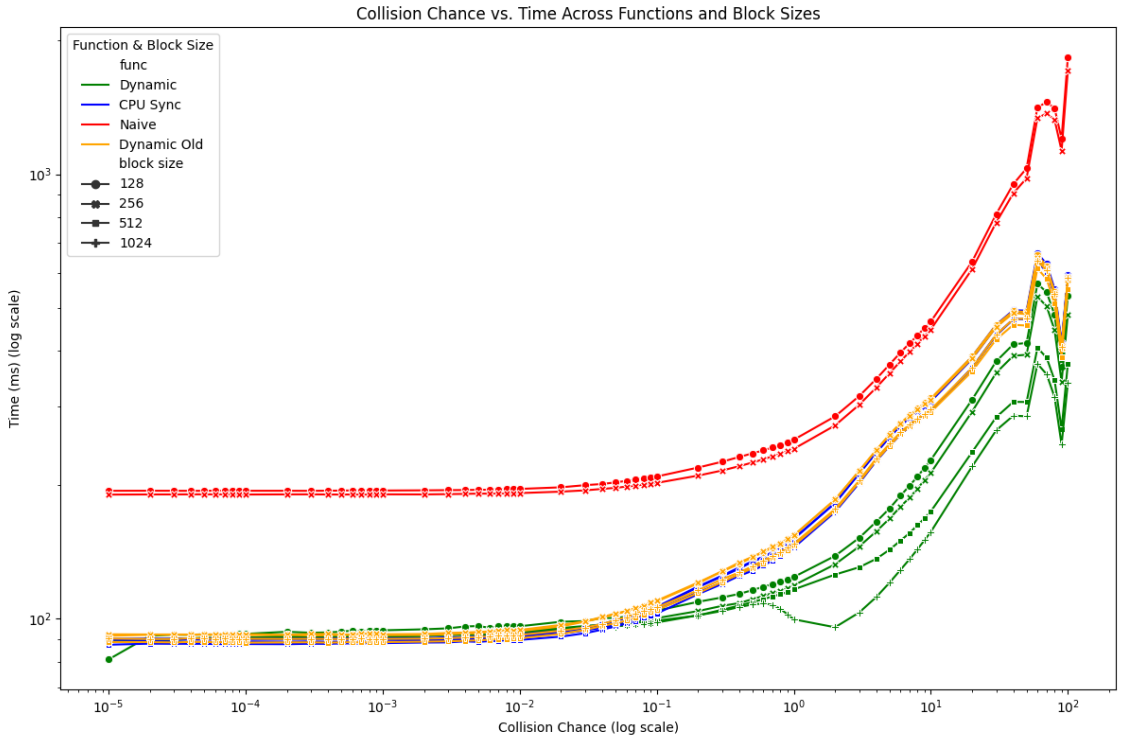

Figure 9.3: Cross Section Test Short

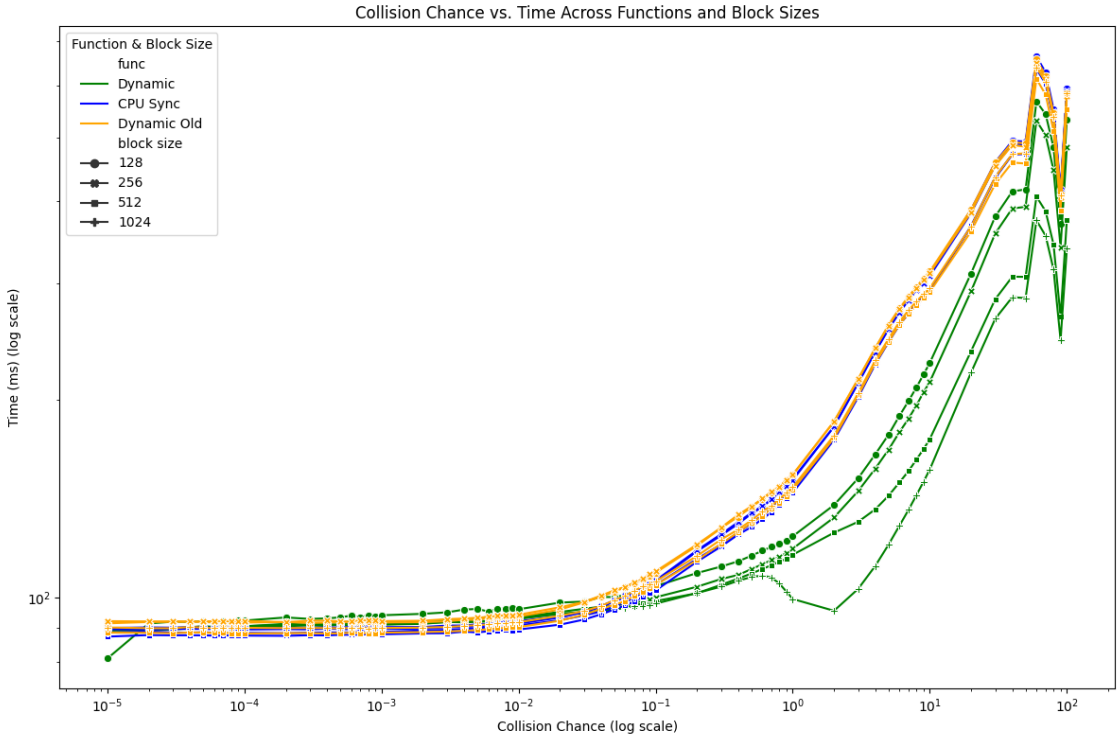

Figure 9.4: Cross Section Test Short, Without Naive

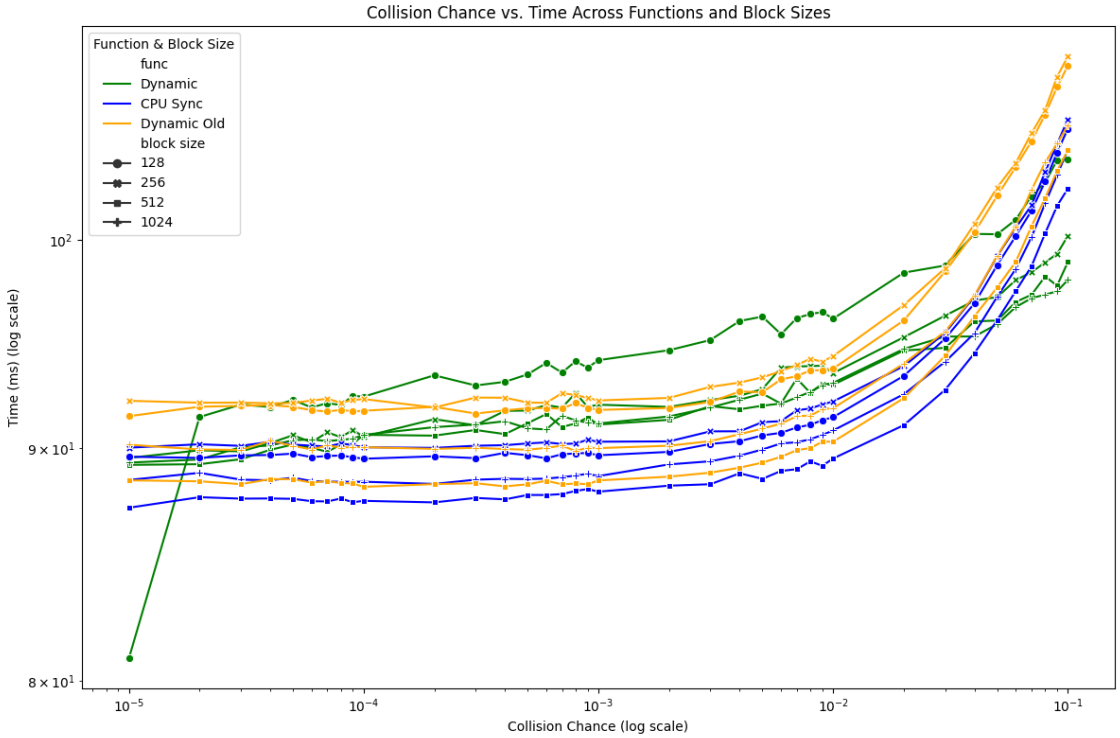

Figure 9.5: Cross Section Test Short, Low Chances

9.4 PIC Poisson Steps Test Dynamic

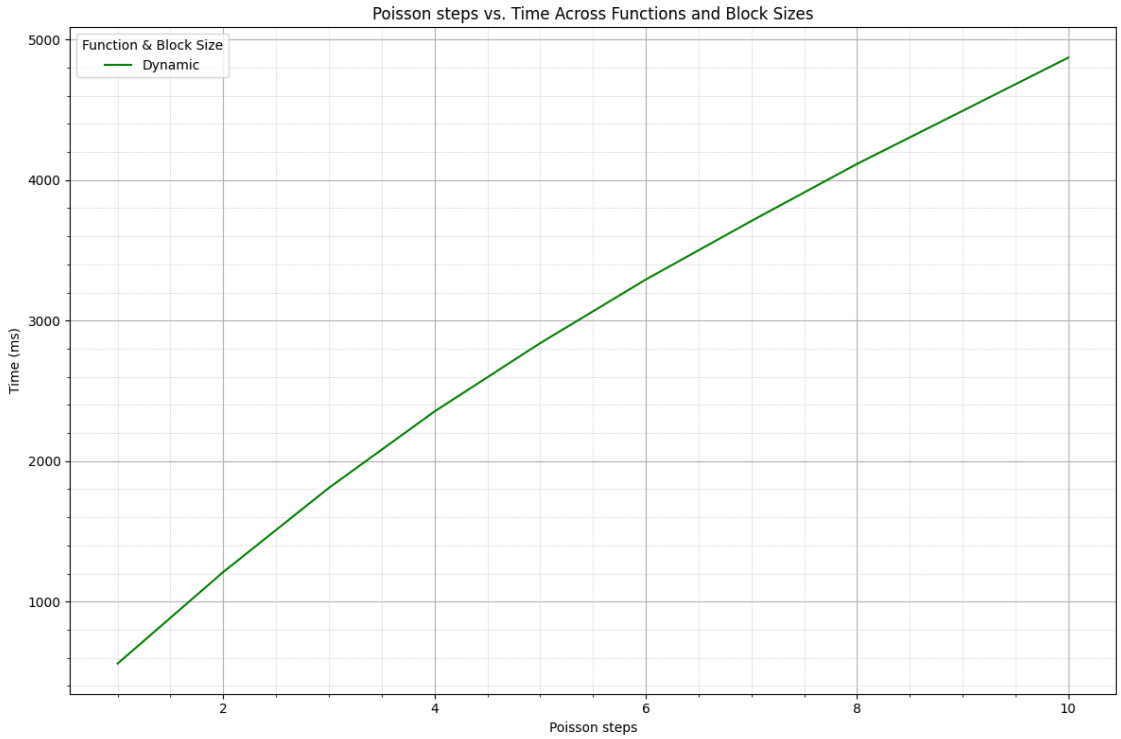

Figure 9.6: Performance According to Poisson Steps, Dynamic Only

9.5 PIC Mobility Steps Test Dynamic

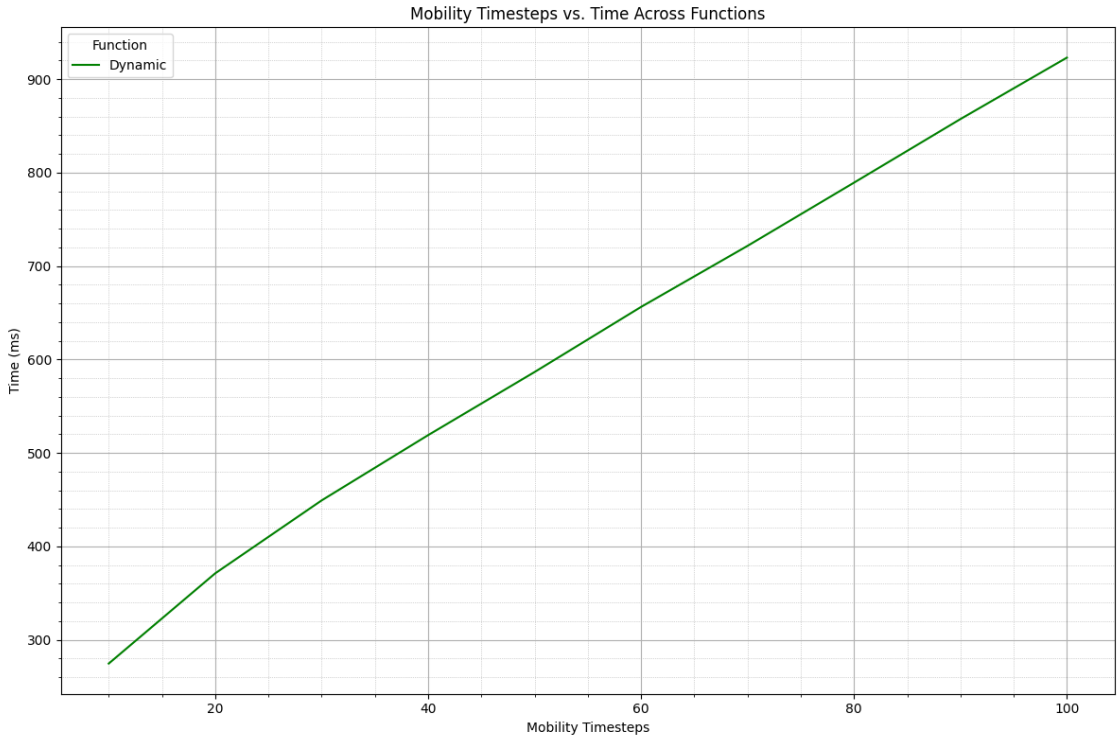

Figure 9.7: Performance According to Mobility Steps (Low), Dynamic Only

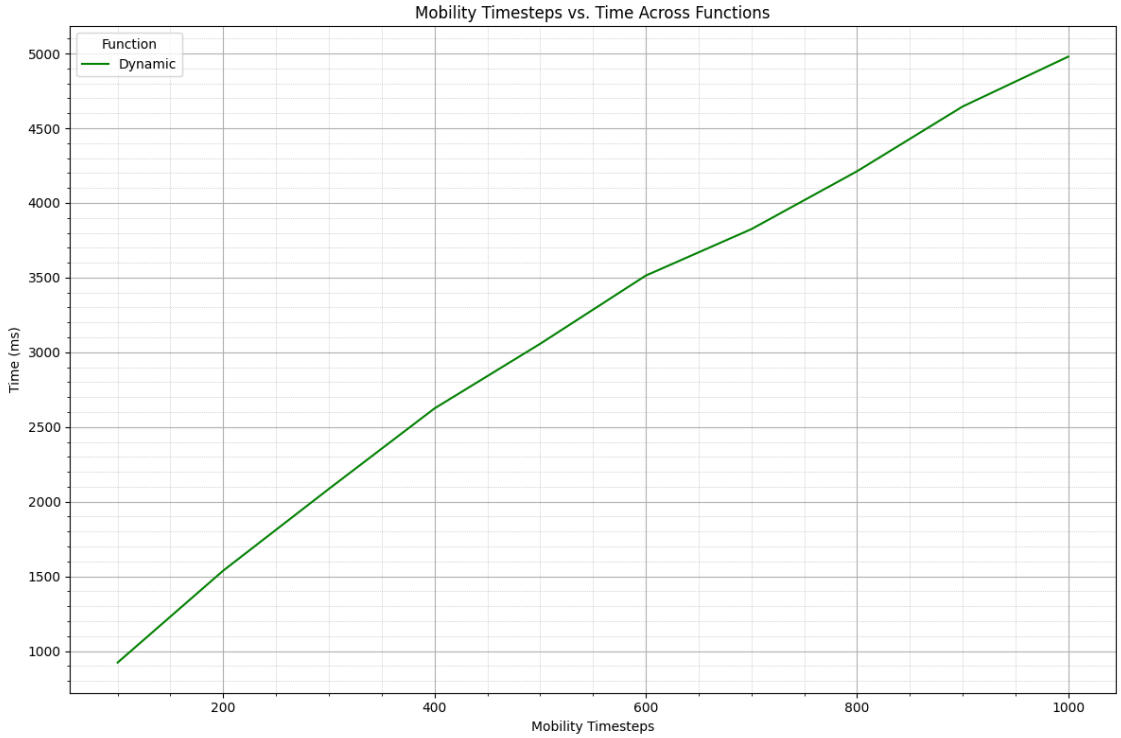

Figure 9.8: Performance According to Mobility Steps (High), Dynamic Only

9.6 PIC Initial N Test Dynamic

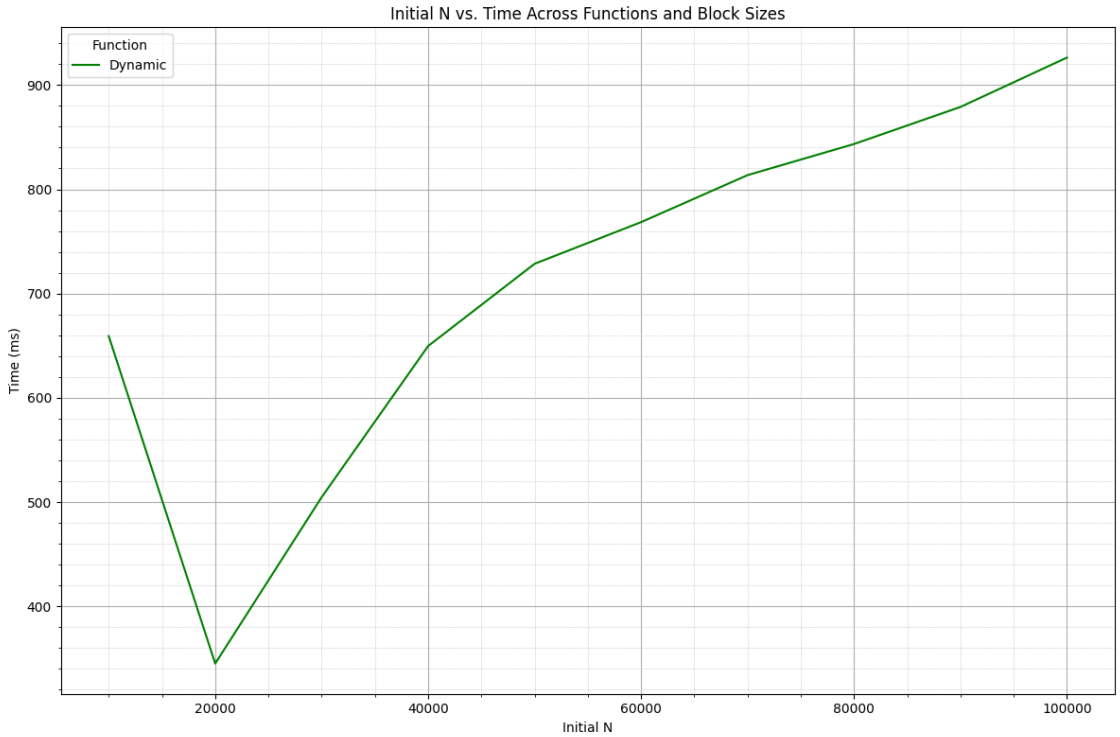

Figure 9.9: Performance According to Mobility Steps (Low), Dynamic Only

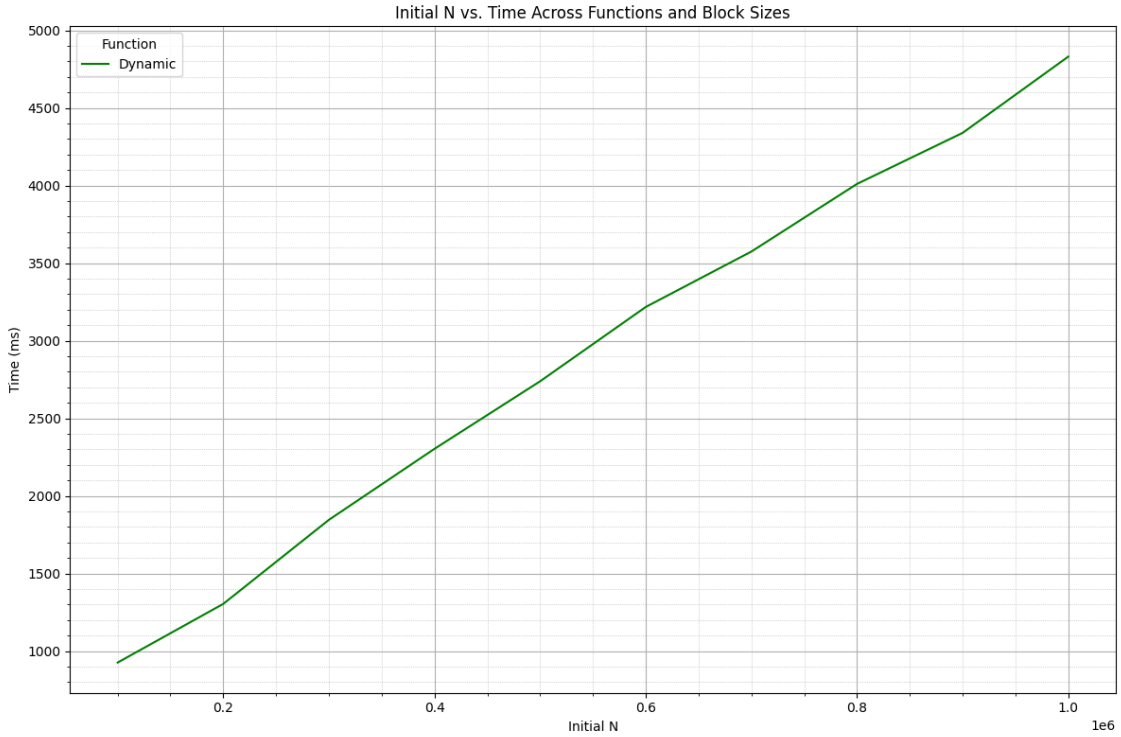

Figure 9.10: Performance According to Mobility Steps (High), Dynamic Only

9.7 Project Management

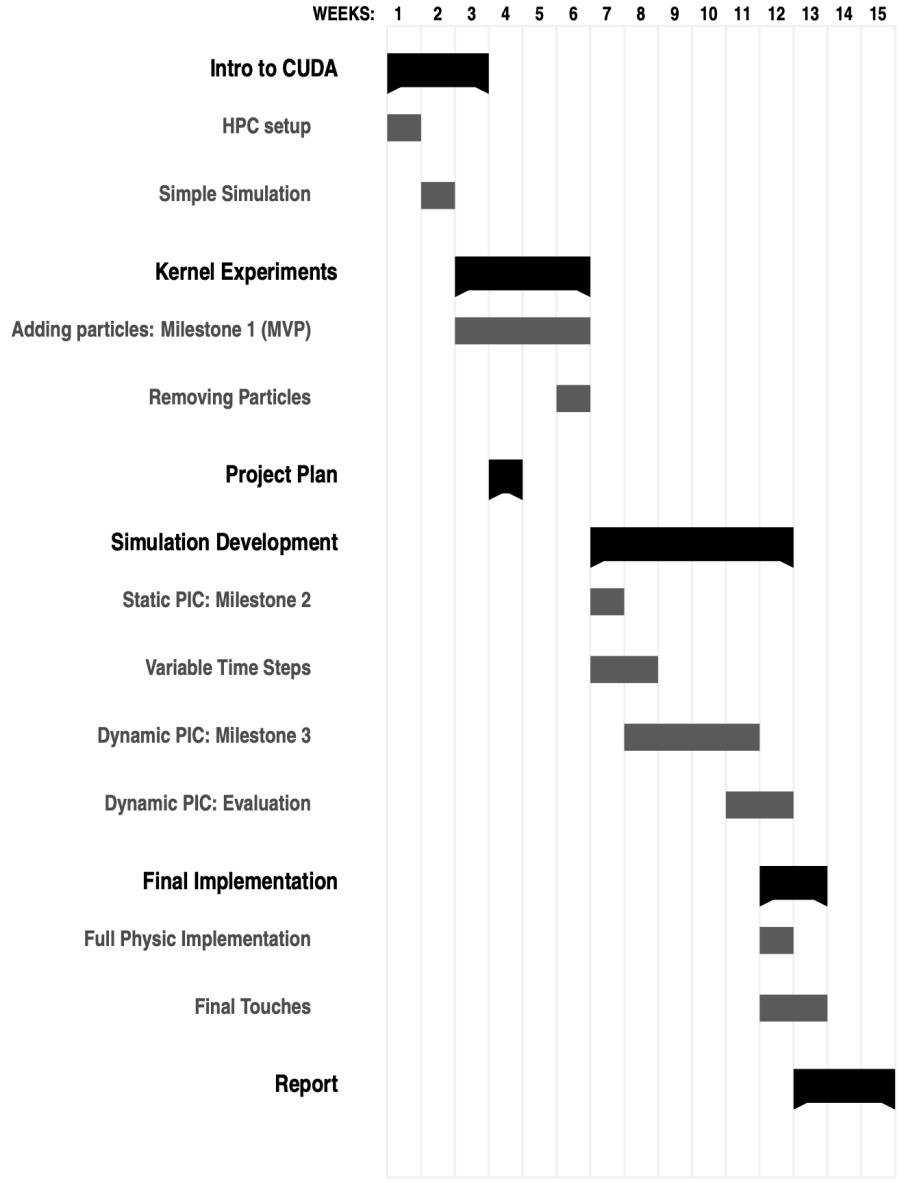

Figure 9.11: Gant diagram of old project plan

Bibliography

[1] NVIDIA. *CUDA C++ Programming Guide: The Benefits of Using GPUs*. 2024. url: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#the-benefits-of-using-gpus.

[2] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE*. 2017. url: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[3] NVIDIA. *CUDA C++ Programming Guide: SM Architecture*. 2024. url: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#architecture-8-x.

[4] NVIDIA. *CUDA C++ Programming Guide: Multiprocessor Level*. 2024. url: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#multiprocessor-level.

[5] Jan Verschelde. *Accessing Global and Shared Memory*. 2016. url: http://homepages.math.uic.edu/~jan/mcs572f16/mcs572notes/lec35.html#accessing-global-and-shared-memory.

[6] Gregory D. Moss et al. "Monte Carlo model for analysis of thermal runaway electrons in streamer tips in transient luminous events and streamer zones of lightning leaders". In: *Journal of Geophysical Research: Space Physics* 111.A2 (2006). doi: https://doi.org/10.1029/2005JA011350. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2005JA011350. url: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2005JA011350.

[7] *The Electrostatic Particle In Cell (ES-PIC) Method*. 2010. url: https://www.particleincell.com/2010/es-pic-method/.

[8] *cuRAND Library*. 2024. url: https://docs.nvidia.com/cuda/curand/group__DEVICE.html#group__DEVICE_1g50ffb42b31744f27be26d98c1f30d51d.

[9] Greg Gutmann. *cuRAND Usage: CUDA Feature Testing*. 2020. url: https://codingbyexample.com/2020/09/15/curand/.