DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Guided research

Solving Linear Systems for Differentiable Physics using cuSPARSE

Marc Gavilán Gil

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Guided research

Solving Linear Systems for Differentiable Physics using cuSPARSE

Lineare Systeme für differenzierbare Physik mit cuSPARSE lösen

Author:	Marc Gavilán Gil
Supervisor:	Prof. Dr. Nils Thuerey
Advisor:	M.Sc. Philipp Holl
Submission Date:	[Submission date]

I confirm that this guided research is my own work and I have documented all sources and material used.

Munich, [Submission date] Marc Gavilán Gil

Acknowledgments

Abstract

Partial differential equation (PDE) solvers embedded within machine learning algorithms are computationally expensive tasks. In particular, we consider sparse linear systems of equations and set out to solve them efficiently on GPUs. In them, the solver for PDEs, the conjugate gradient method (CG), is the bottleneck of our system. Despite the fact that the CG method is one of the fastest solutions given the properties of our equations. At the core of the CG algorithm we find the matrix multiplication. Therefore, we tackle this problem by optimizing the matrix multiplication subroutine. For that, we use sparse matrices and implement a sparse matrix multiplication routine in CUDA. Additionally, we use PyTorch's C++ frontend to implement additional optimizations for CG solver. As a result, these approaches produce significant performance improvements.

Contents

1 Introduction

Partial differential equations (PDEs) have been used to describe hundreds of natural events for ages. A PDE is a mathematical formulation that scientists use to define natural laws. PDEs are commonly used to represent neuronal transmission in biology and predict weather in fluid mechanics, for example. These equations are solved using a linear solver in our project. The conjugate gradient (CG) method is the linear solver of choice for this project.

We can make predictions about the behavior of physical phenomena by using PDE to describe them. Machine learning (ML) is now one of the most powerful techniques for making predictions. By combining machine learning approaches with PDE simulations, we may achieve error reduction in simulations that were not possible before the addition of learning techniques. [A paper with a Solver-in-the-Loop solution]. The cost of integrating a PDE solver into a machine learning system is high. We must solve PDEs at each training iteration, and the conjugate gradient method is at the heart of this process. We will execute the identical actions millions of times across different data instances throughout training. Furthermore, the training is frequently done in batches of data. This means that any inefficiencies in the implementation will slow down the process and make training take an enormous amount of time.

In this project, we'll look into all of the choices for reducing the CG method's computing time. We demonstrate how we can minimize the initial computing time by constructing a low-level PyTorch implementation utilizing C++ and CUDA functions. The outcomes will be assessed by profiling the application as it progresses through the various optimization stages.

2 Methods

In this section we explain how we have carried out the optimization in the implementation. In order to do so, we have added a new data structure to store CSR matrices. Additionally, the cuSPARSE library is used to perform operations on this objects. In order to reuse buffers and enhance performance, we also device a memory optimization approach. Finally, to boost performance, we perform a JIT compilation of the code.

Sparsity pattern.

The systems we are working on are typically contained within two or three dimensional boxes. A matrix describes the particles in this two- or three-dimensional space. This matrix is sparse as only cells in the direct vicinity are modelled as connected. The number of non-zero elements in it is typically 0.2%. In other words, 998 out of 1000 values are zero. Clearly, this property has the potential to give us an advantage in the way we store and manipulate data.

Sparse multiplication options.

The following are the various options in order to implement a sparse representation. PyTorch provides a sparse module. This functionality is still under development and the authors warn that it may not be very stable. For this reason, we decide to use a different approach. We implement our own sparse module that uses CUDA libraries such as cuBLAS and cuSPARSE.

Once decided to use a sparse representation for matrices with a high sparsity factor, the next step is to select the appropriate sparse matrix representation. Researchers have designed multiple sparse data types: Dictionary of keys (DOK), List of lists (LIL), Coordinate list (COO) figure 2.1, Compressed sparse row (CSR) figure 2.2. For performance reasons, the most common representation in sparse libraries such as cuSPARSE are COO and CSR. Therefore, our decision is narrowed down to using either COO or CSR.

COO utilizes three vectors to store the matrix data. Every of the three vectors has the same size. The first vector represents the column number of every non-zero element. The second one encodes the row number. The third vector represents the value. In the following figure we observe an example of this intuitive representation

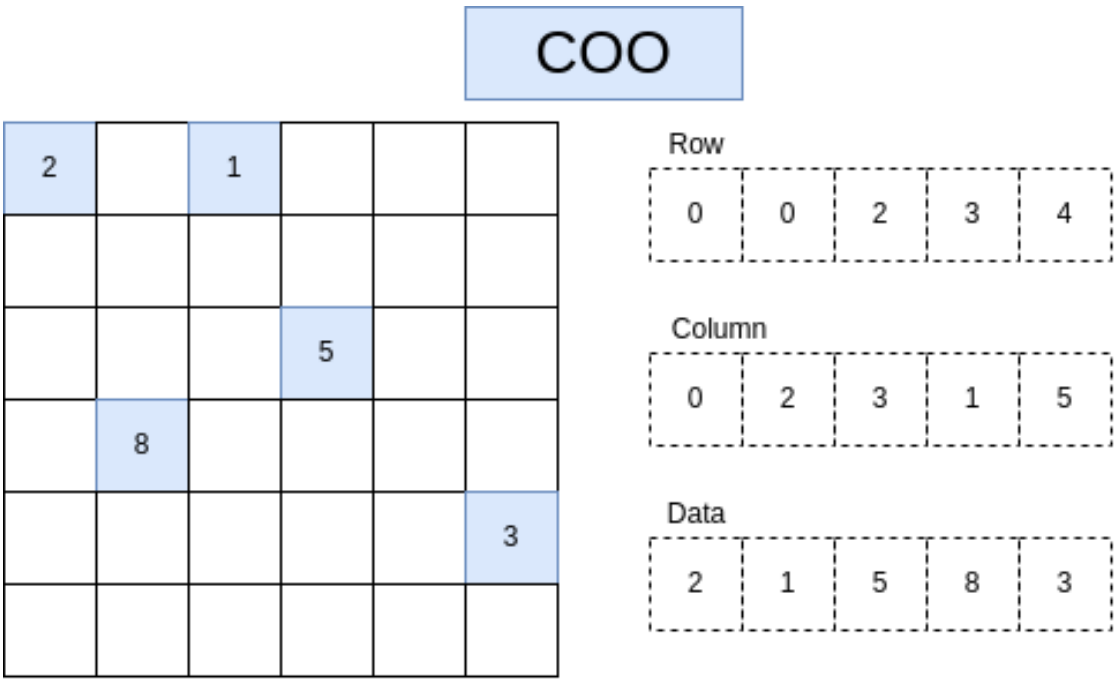


Figure 2.1: Diagram showing how COO works.

CSR also utilizes three vectors to store the data. In this case we find two vectors of the same size, number of non-zero elements. And a third vector that is as long as the number of rows. Likewise to COO, the first vector represents the column number of every non-zero element. The second vector represents the number of elements that are stored up to that row. Lastly, the third vector stores the values for non-zero elements in order. The following figure portraits an example of this representation [Figure].

We decide to use a CSR format because of multiple reasons: Data representation is in general more compact for CSR. COO stores three vectors that asymptotically are $O(3nnz)$ where nnz is the number of non-zero elements. CSR also stores three vectors that asymptotically are $O(2nnz + n)$ where n is the number of rows. The second advantage is that in general CSR provides better performance for matrix multiplications.

Baseline implementation.

The implementation that will serve as a baseline to compare the performance of new optimization is the following. PyTorch's CSR representation is used for sparse tensors. As previously mentioned, this representation is still under development and for reliability reasons we prefer not to rely on it in our further optimizations. The entire

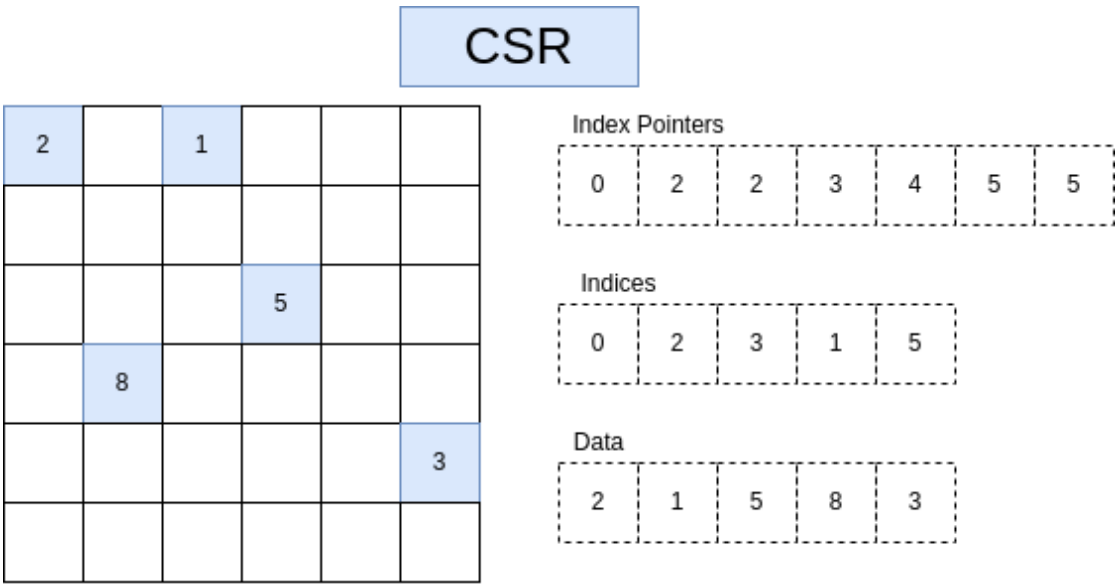
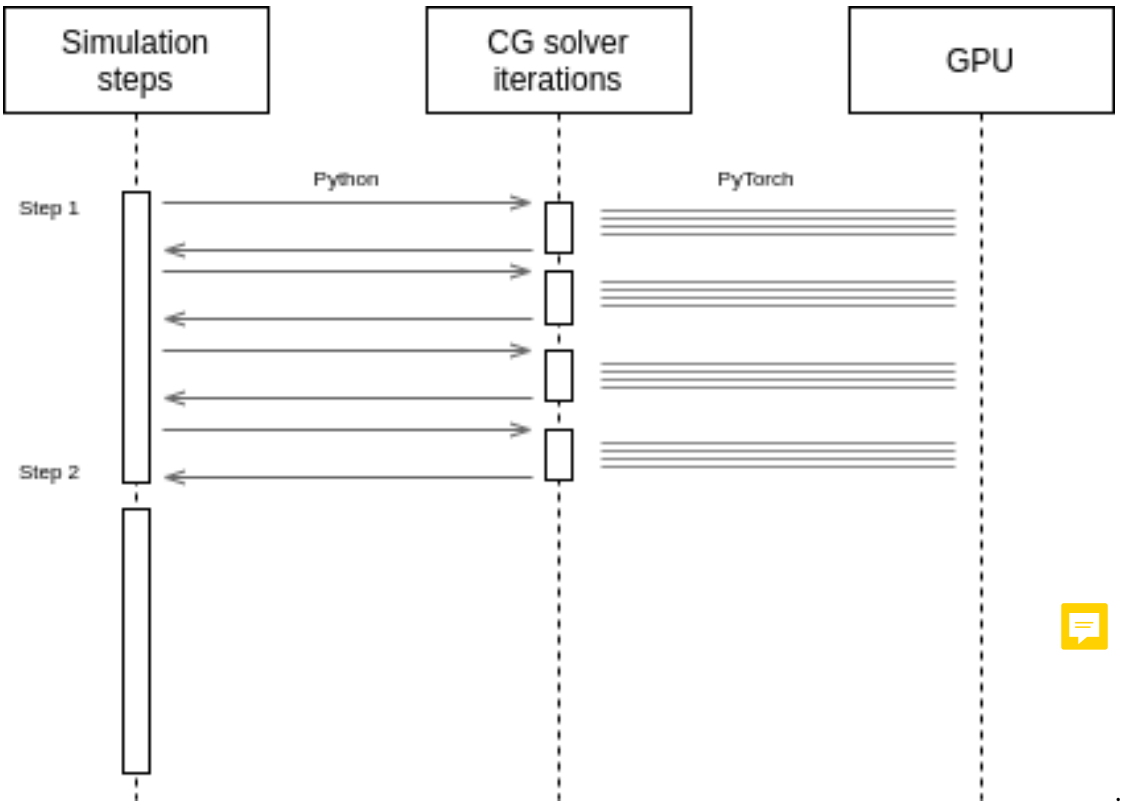


Figure 2.2: Diagram showing how CSR works.

implementation runs in Python using PyTorch's library to instantiate tensors in the GPU. These GPU tensors are also operated in the GPU using PyTorch functions.

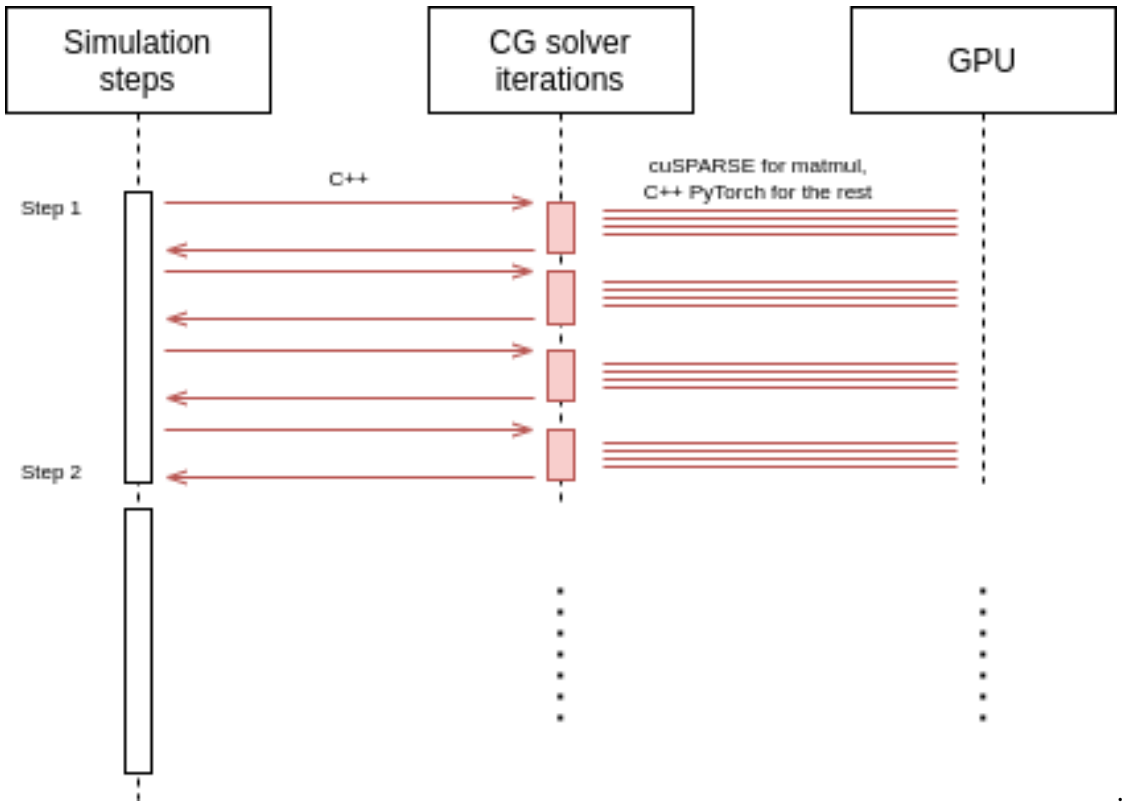
cuSPARSE matmul implementation.

The first optimization is to use the cuSPARSE library to perform the sparse matrix multiplication. This library is a part of the CUDA toolkit and provides optimized implementations for operations that involve sparse representations. The function is SpMV and it multiplies a sparse tensor by a dense vector. The CSR representation that we use is a custom Python class that contains the tensors that describe a CSR matrix: Values, column pointers, row pointers and shape. Calls to cuSPARSE are made using C++. Conveniently, PyTorch offers a C++ frontend that provides a C++ interface to reference objects that are originally created in Python. This interface does not incur large overheads since PyTorch's underlying source code is predominantly C++. Therefore this interface communicates my C++ functions with PyTorch's C++ source code. Additionally, for compilation purposes we use the Python library setuptools. This library provides means to compile a C++ application as a Python module to import it into our Python project. The following is a diagram that shows the current optimization



C++ implementation.

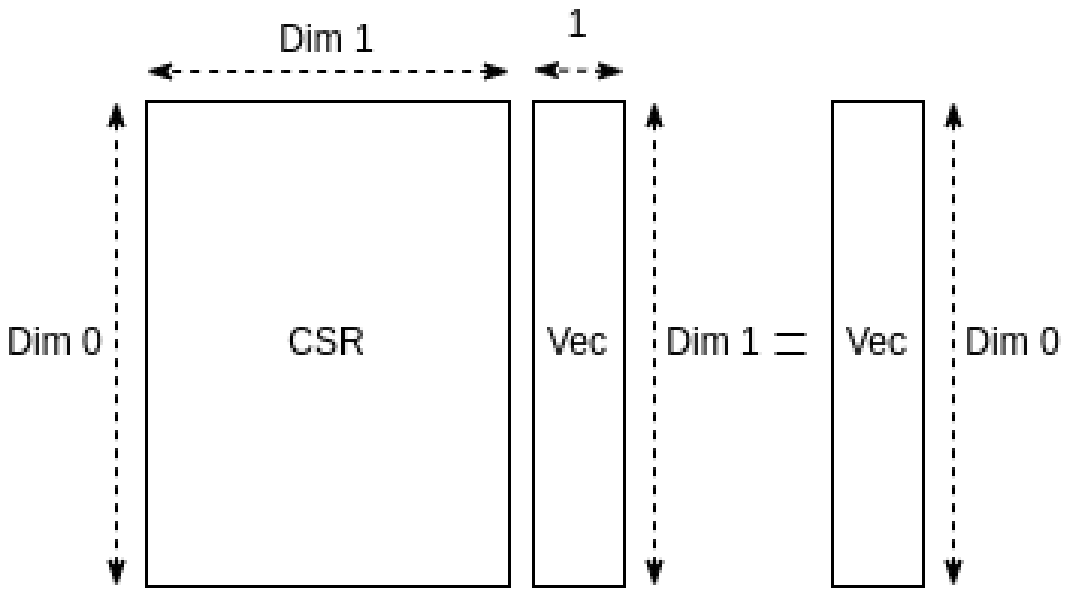
It is known that Python includes an overhead that decreases the performance in comparison with lower-level languages such as C++ [cite]. For our next optimization we decide to implement the CG solver as a C++ function. In this implementation, in order to reference the PyTorch tensors created in Python we use of the PyTorch C++ frontend again. This is a diagram of this optimization.



Memory-optimized C++ implementation.

Once we produce the C++ code, we analyze which parts of the execution of the function that solved the CG method were bottlenecks. We notice that the code is freeing and allocating new memory at each sparse matrix multiplication. Therefore, we decide to focus on memory optimization since a large amount of potentially reusable memory locations are allocated and destroyed inside the CG loop. Therefore, as a first memory optimization we reuse the buffers needed to perform sparse matrix multiplications. The method we use to reuse buffers is as follows. A single pointer to a buffer is created and a value is stored indicating the largest memory buffer that has been required so far. Each time a larger memory size is needed, the memory in the previous buffer is freed and the buffer pointer points to a larger memory area. In the event of needing a smaller size of memory than the one we are buffering we simply use the currently stored buffer. This memory reutilization technique relieves us of the overhead of freeing and reserving memory for each sparse matrix multiplication. Also, since all matrices have a similar size in our system, it is typically only necessary to reserve memory at the beginning of the execution of the CG solver. As a second memory optimization we reuse the

matrix CSR representation needed for sparse matrix vector multiplications. This is a feasible approach because the sparse matrix remains constant for the whole solution of the CG method. In this second optimization we store the sparse matrix representation in a variable that is used for all multiplications of the CG method. Lastly, since the sparse matrix remains constant throughout the solution of the CG method, the vector that results from multiplying that matrix with any other vector has a constant size.



Therefore, the last memory optimization consists of reusing the memory location that reads the matrix-vector multiplication result. For every multiplication, this memory location is completely rewritten.

JIT compilation.

Finally, as mentioned above, one of the weaknesses of high-level languages such as Python is the performance inefficiency compared to lower-level languages. Therefore, the last optimization we carry out is to perform a compilation of our Python code before the simulation run. This is known as just in time compilation (JIT). PyTorch allows JIT compilation by pre-compiling the code to C++ functions. As we will observe in the results, the start of execution with a JIT compilation is always slower than its non-JIT counterpart as it needs an initial compilation time. However, the subsequent execution is noticeably faster.

3 Results

To perform our experiments we simulate a configuration that is representative of a real use case. This simulation is the following: Hot smoke is emitted from a circular region at the bottom of a closed box. We can observe this simulation in figure 3.1. The simulation uses a conjugate gradient method to solve the PDEs that model the airflow. This simulation runs in a two-dimensional setup. The experiments run using PyCharm's profiler. In physical simulations, it is often interesting to vary the numerical precision of the operations to obtain a balance between accuracy and performance. To show our results, we will first explain the legend that we will use in the following figures. Simulations that run using 32-bit precision will be encoded with solid lines; 64-bit precision will be encoded using dotted lines. Simulations that run using JIT compilation will be encoded using a dot mark on every time step.

In figure 3.2 we observe the total cumulative time of each implementation after completing 100 simulation steps. We can see two implementations that are noticeably faster than the rest: C++ and C++ with memory optimization running on 64-bit precision and using JIT compilation. Between them, C++ with memory optimization is slightly faster than the initial C++ implementation. The time per simulation step over the entire simulation can be found in figure 3.3. Both a C++ implementation and a JIT compilation noticeably reduce the computational time.

A value that indicates the relative improvement over the original implementation is the speed-up. In figure 3.4 we can see a comparison of the speed-ups of all the implementations for the baseline: Original implementation in 32-bit precision with JIT compilation. The two fastest implementations, C++ with and without memory optimizations running on 64-bit precision and using JIT compilation, that we observed on the cumulative time graph yield a speed-up of 78.3% and 72.1% respectively.

The main element we have focused on when optimizing the computation is the resolution of the CG method. This method takes In figure 3.5 we show the time taken by the CG method iterations for each simulation step. One element to note is that this graph ignores the first three simulation steps as these values are significantly larger than the rest of the steps. The values for these first three steps are given in figure 3.6. We observe that the JIT compilation adds a significant overhead at the beginning of the simulation.

The resolution of the CG method takes a considerable part of the execution time.

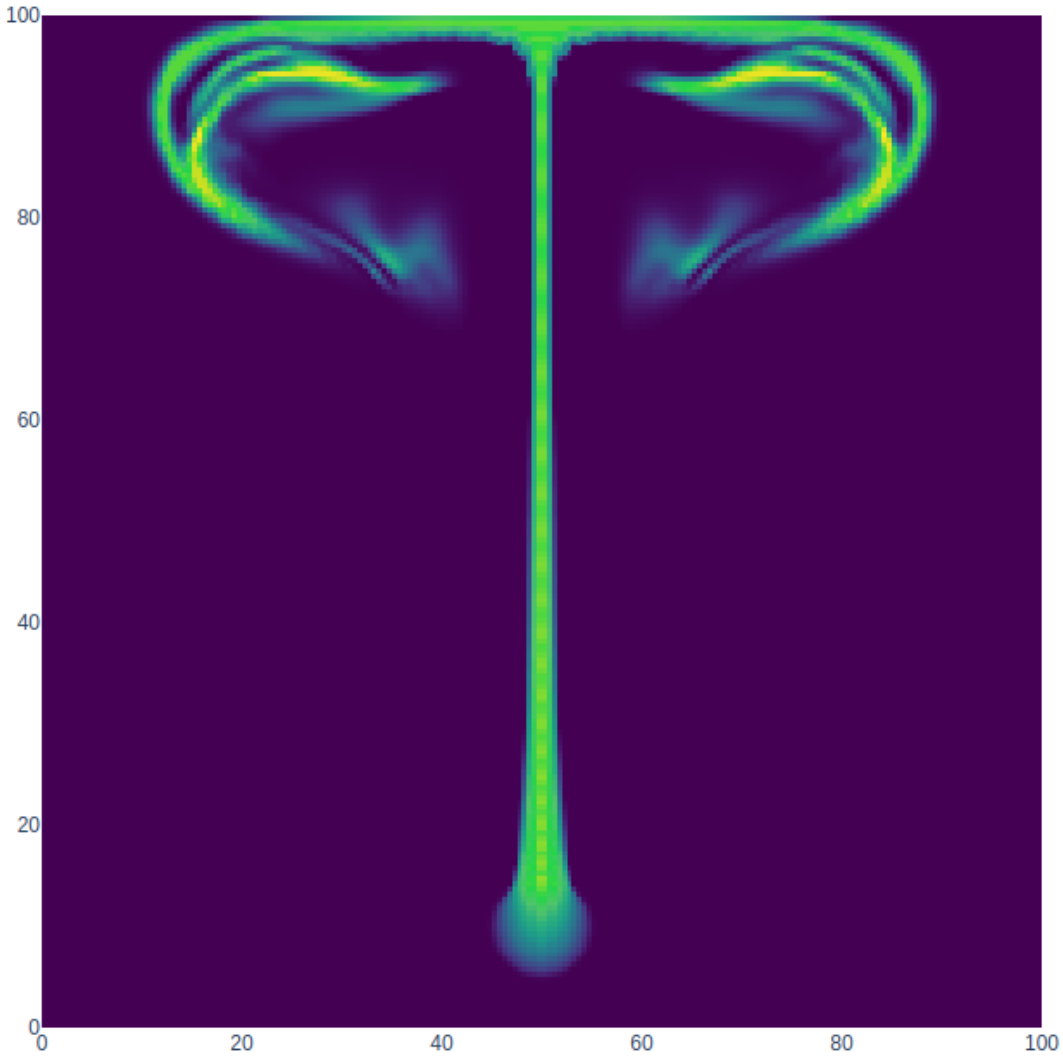


Figure 3.1: Smoke plume simulation state at 100th step.

Figure 3.7 shows these percentages. The value goes down for the first optimization, cuSPARSE matmul implementation. It decreases again for the second and third optimization, C++ with and without memory optimizations implementation. However, despite reducing the computation time, the percentage that the CG method takes increases to a value between 40% and 60% when we apply JIT compilation to these last implementations.

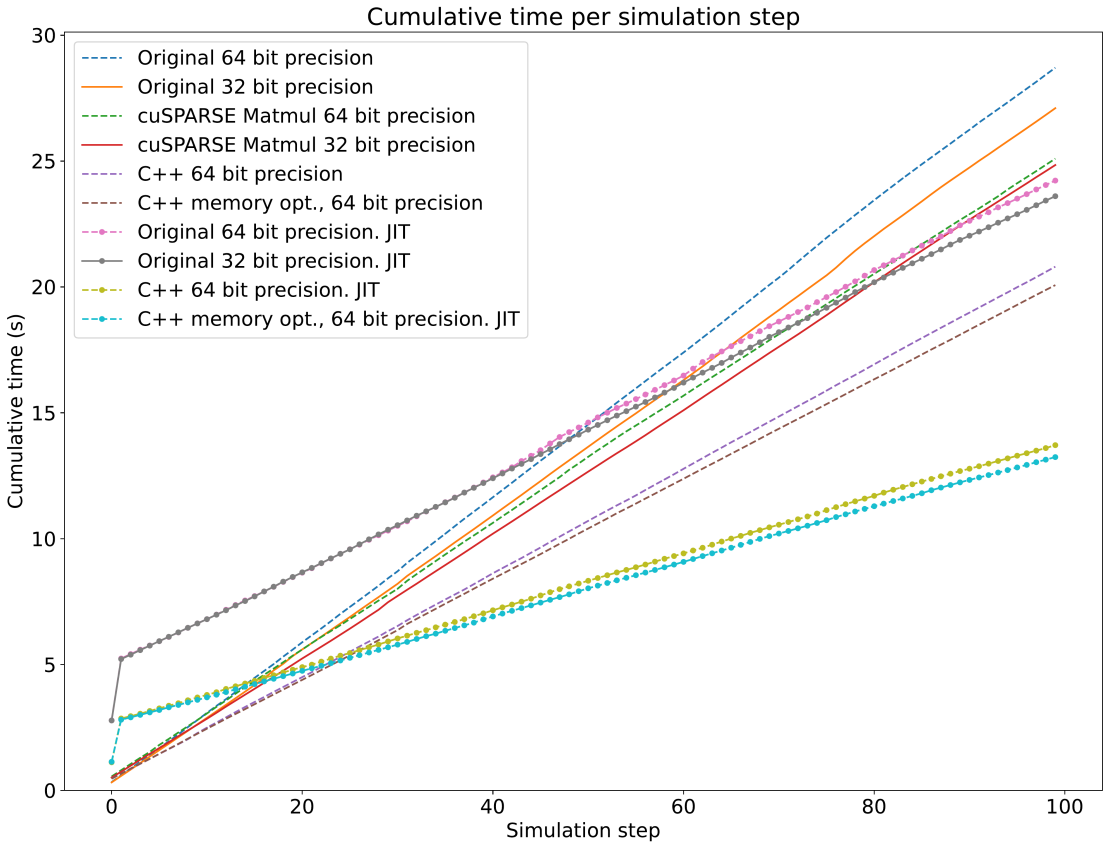


Figure 3.2: Cumulative time per simulation step.

Switching between 32-bit and 64-bit precision can potentially reduce the number of iterations needed for the conjugate gradient algorithm to convert at each simulation step. In figure 3.8 we observe this value for each of the implementations. The number of iterations per simulation step does not change for any of our implementations. In the following, we provide an interpretation of these results and an overview of the progress we have made in this project.

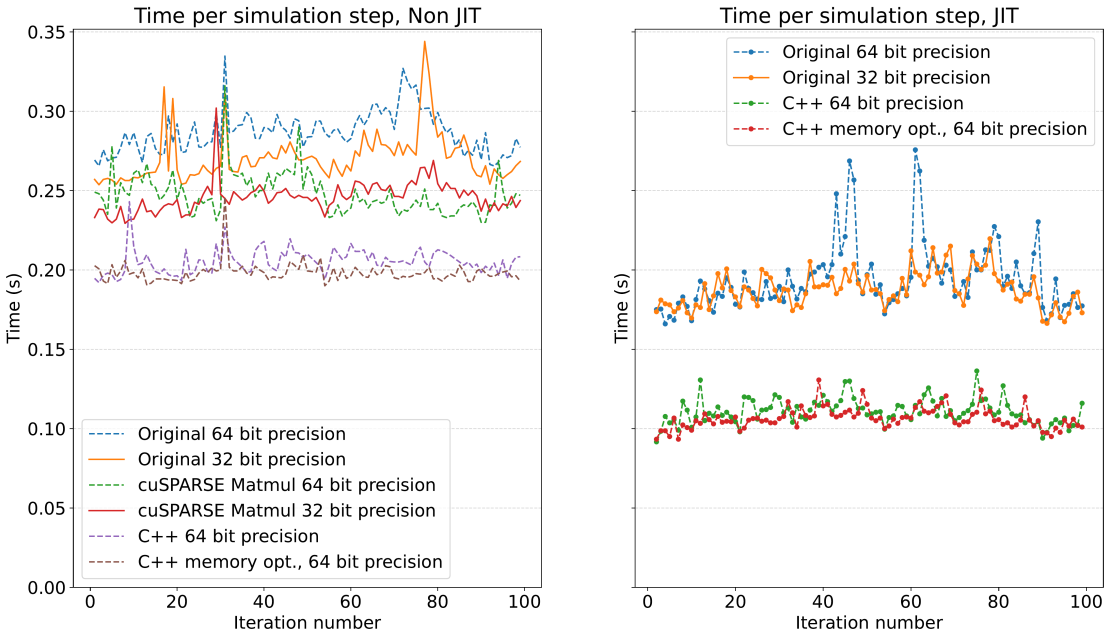


Figure 3.3: Time per simulation step, starting from the fourth step.

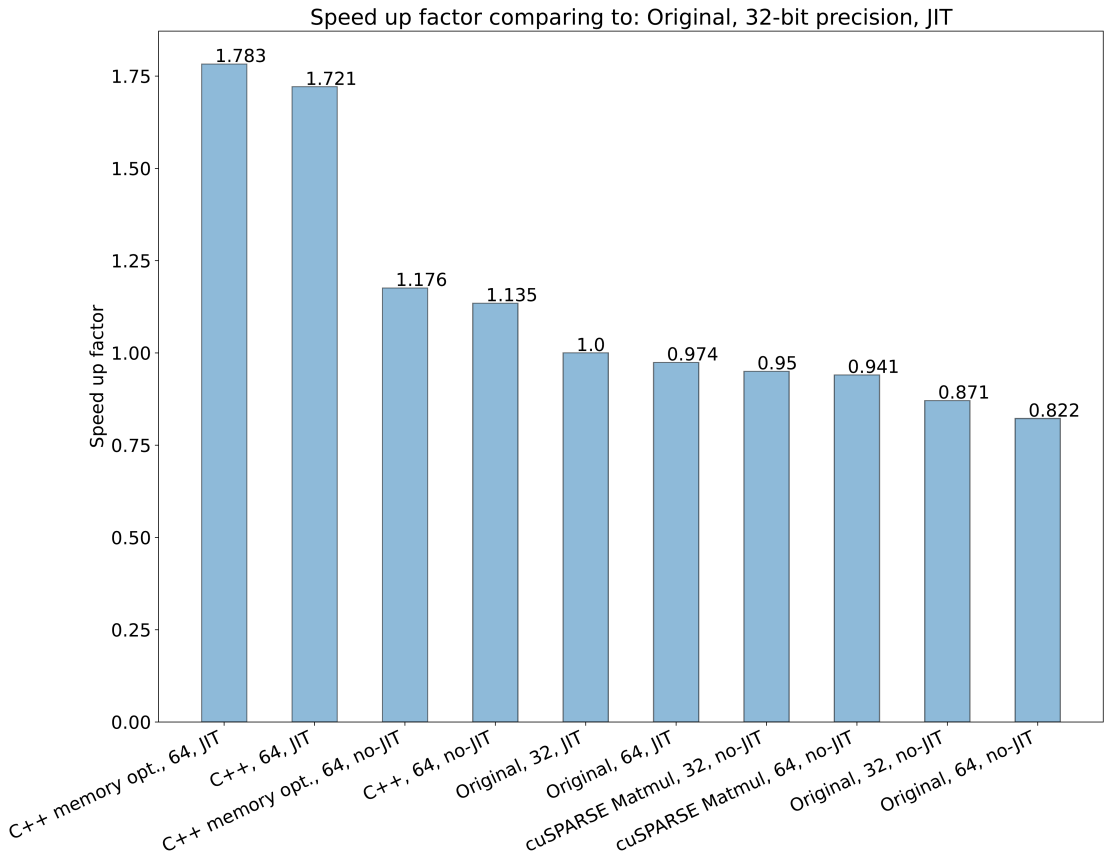


Figure 3.4: Speed-up factor against baseline implementation.

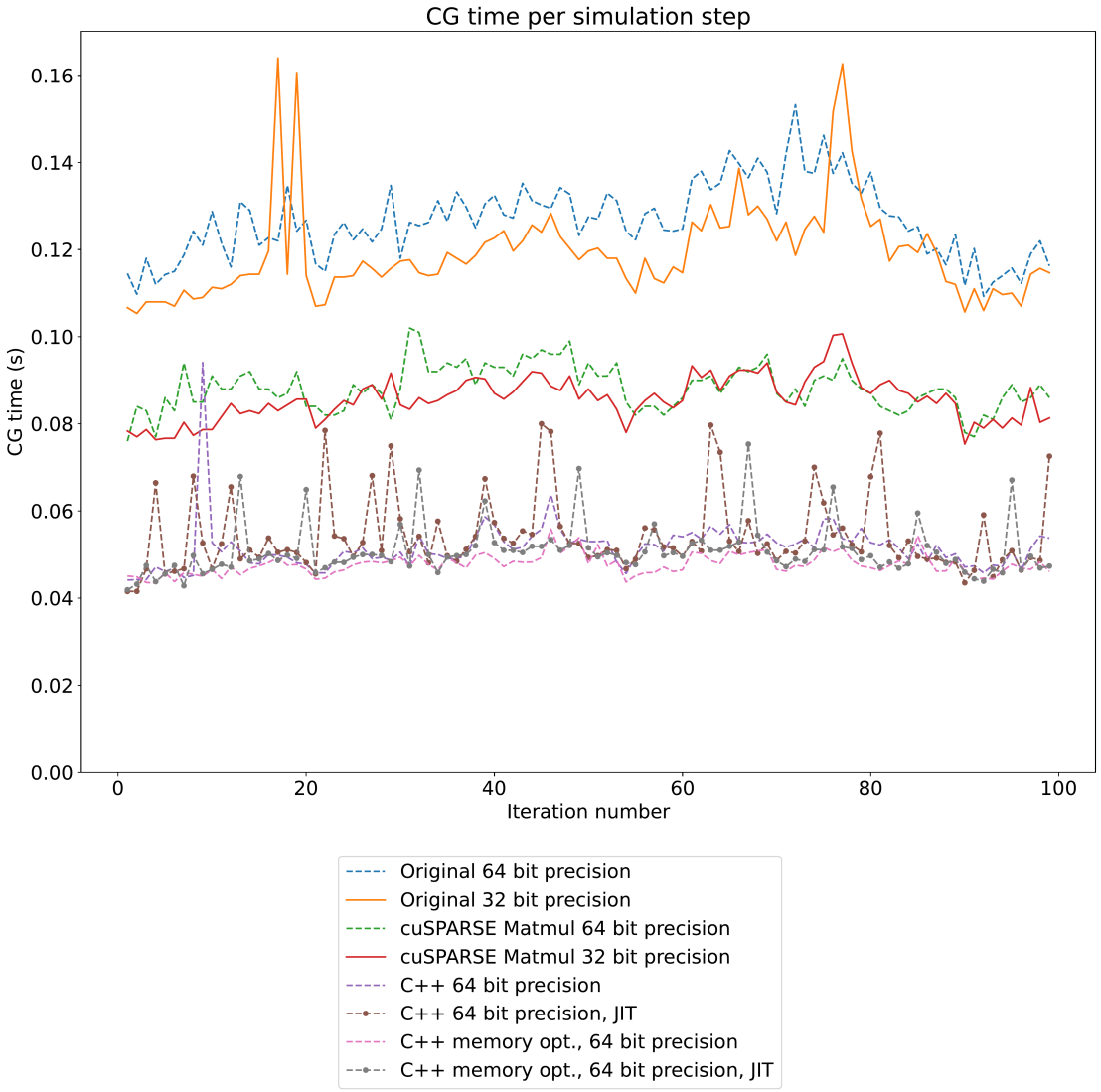


Figure 3.5: Time solving the conjugate gradient method per simulation step.

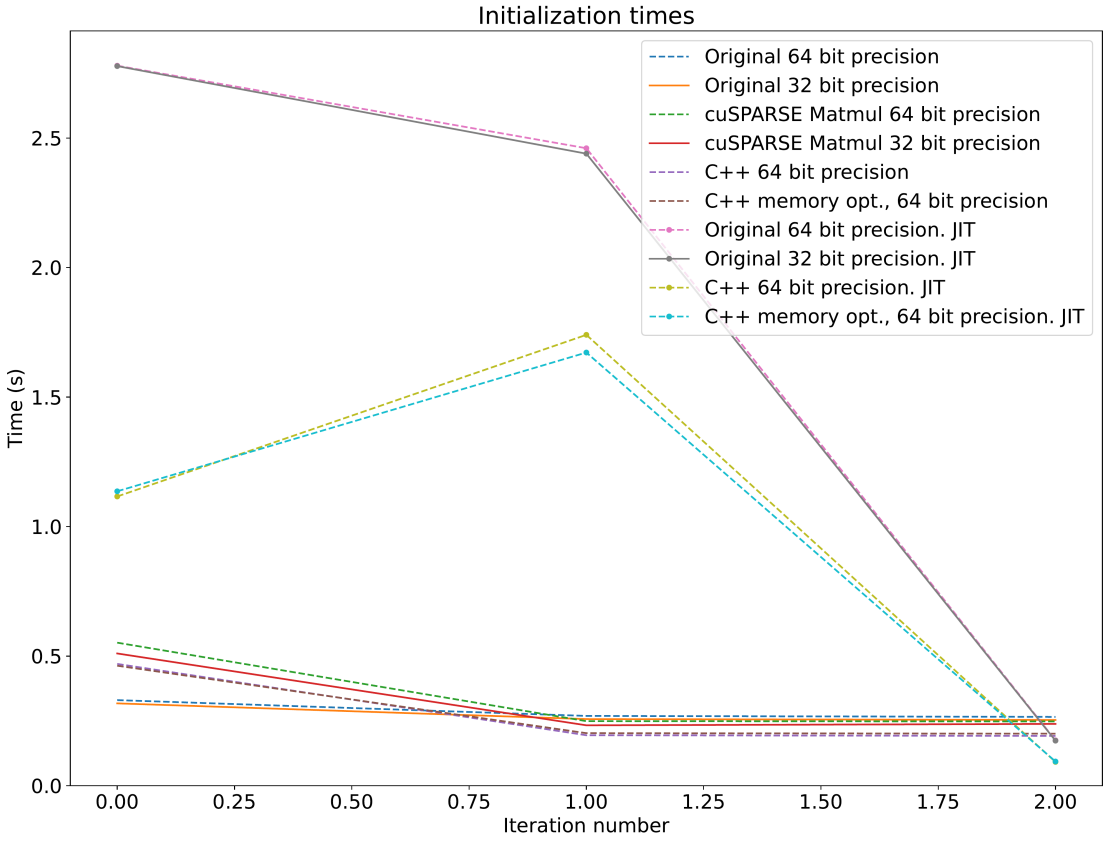


Figure 3.6: Initialization time, simulation's first three steps.

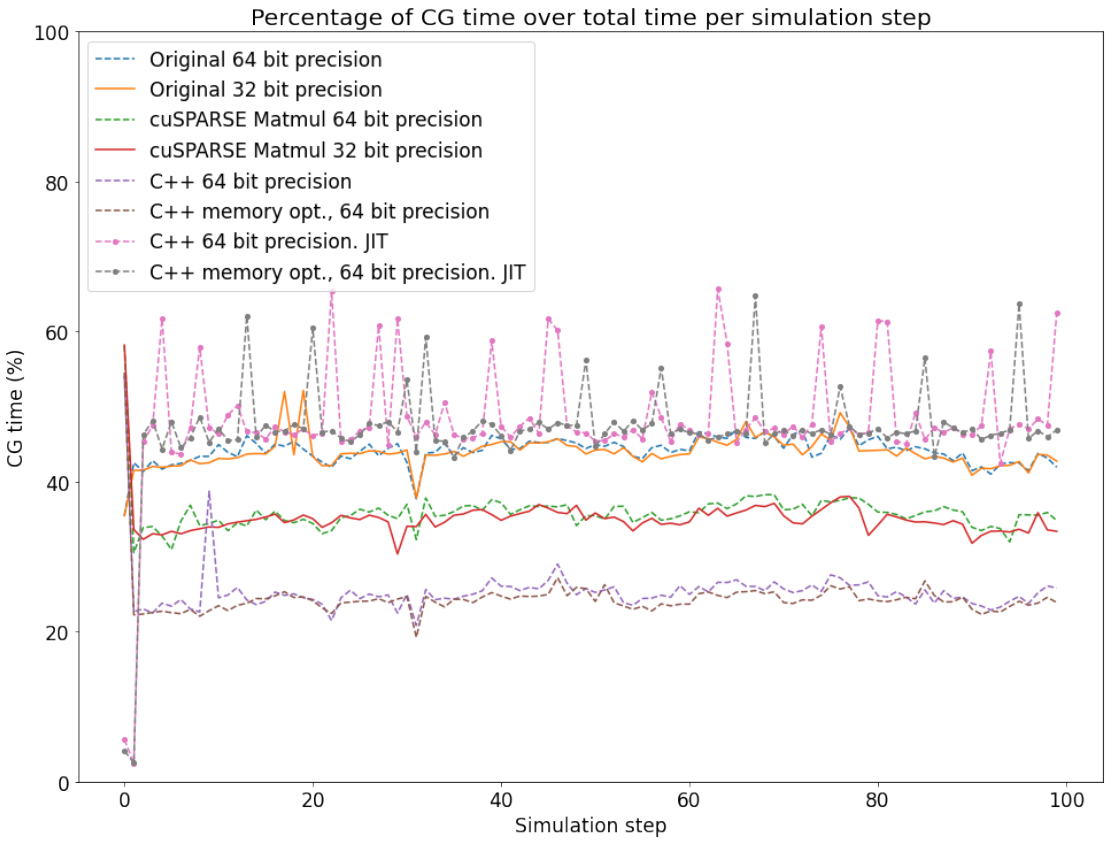


Figure 3.7: Percentage of time that solving the conjugate method takes over the total time for each step.

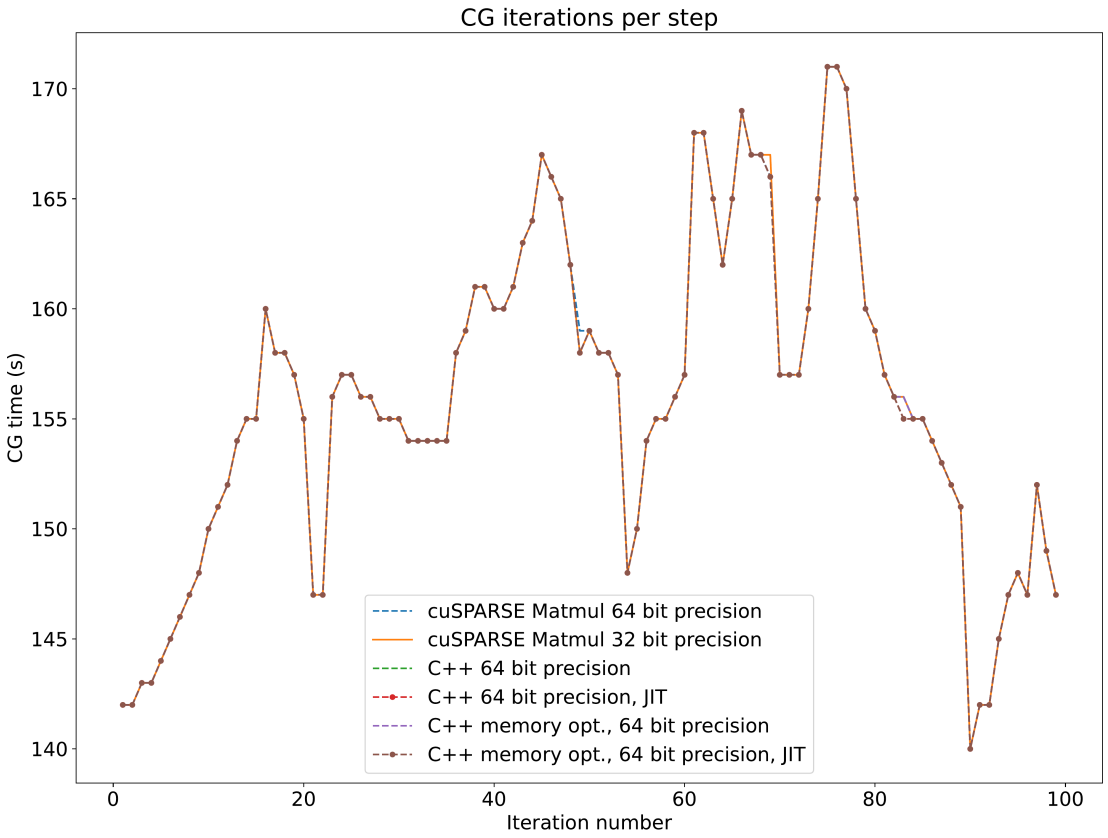


Figure 3.8: Iterations per simulation step.

4 Discussion

The results of our experiments show that we have managed to reduce the computational time of the simulation. The two main factors that helped reducing this computational time have been the encoding of the CG method in C++ and the JIT compilation. The fastest implementation, C++ with memory optimization running on 64-bit and using JIT compilation, improved the baseline execution by 78.3%.

A C++ implementation has proven to be beneficial because of the overhead that a higher level language such as Python introduces to the execution. For similar reasons, a JIT compilation of the simulation has included a significant improvement in the execution time. Figure 3.7 showed us that the CG method is still a bottleneck in the execution. Therefore, there is still room for improvement.

The introduction of JIT compilation in the fastest implementation, C++ with memory optimisation, reduces the execution time of all Python code. Therefore the percentage of time that the execution remains in C++ increases to a value around 40-60%. This means that there is still potential for improvement by focusing on the optimisation of the CG solver.

The results also show that the number of iterations the CG method takes to achieve convergence is constant between 32 and 64-bit precision. This result indicates that running our fastest implementation on 32-bit floating point precision would be a potential run-time improvement. However, the current implementation of the CG solver fails to converge due to numerical accuracy. This is one potential future step that could be of most benefit.