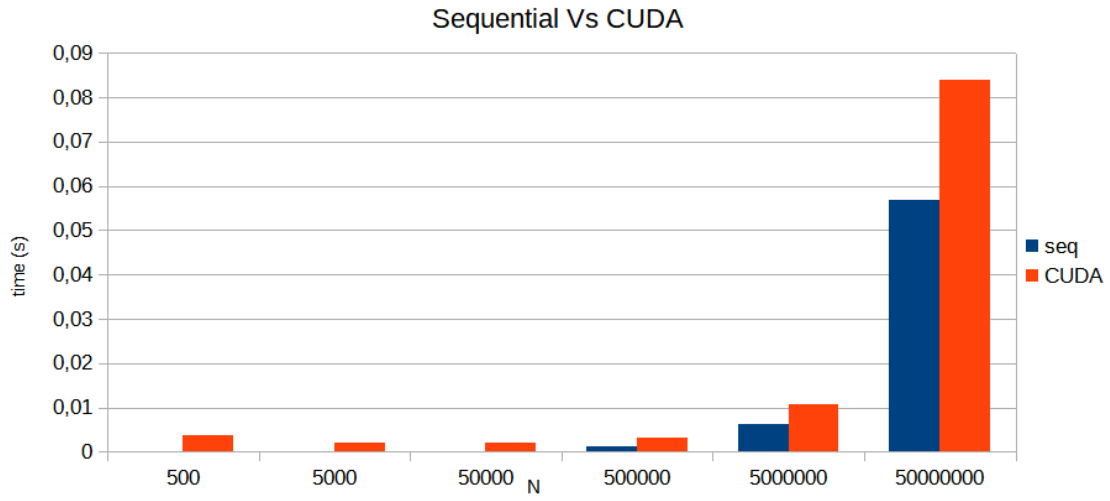
Exercise 1: Vector Addition

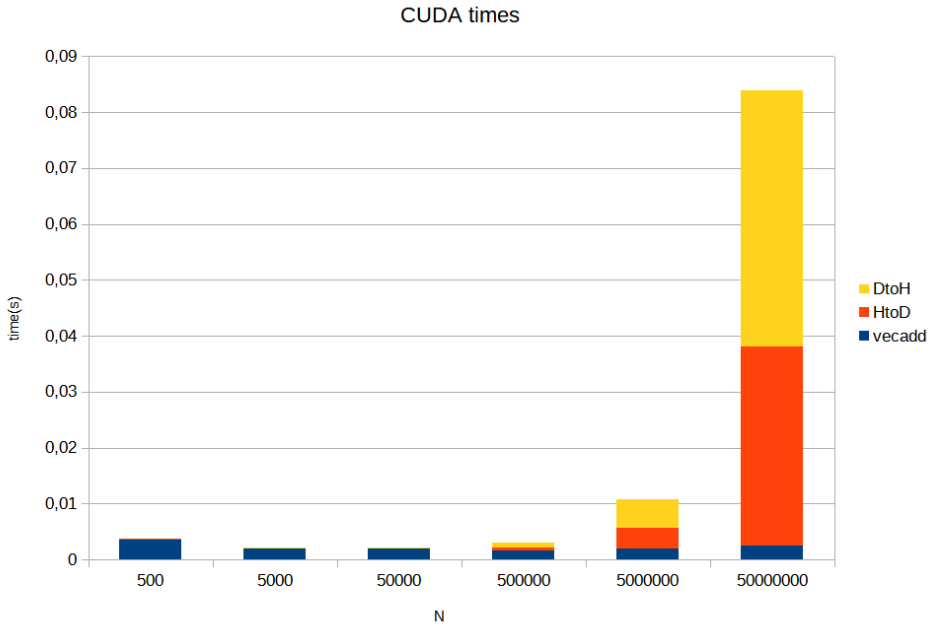
DISCLAIMER: In the following exercises, measures for vector size $5*10^8$ could not be taken due to memory limitations leading to errors. This would only happen in the CUDA versions. Instead, measures were made with vectors of size $5*10^7$.

1. **Compare the sequential version with the CUDA version. At what vector size does the GPU kernel become faster than the sequential one? Create a bar plot showing the execution time of the sequential code, along with the kernel time of the CUDA code. (including memory transfers) for increasing vector sizes N = $5 \cdot 10^k$, for k = 2, . . . , 8. Label the axes clearly.**

We observe the CUDA kernel to be faster than the sequential for vectors of size N = $5*10^6$ and N = $5*10^7$.

Sadly, every time, the time added by the memory transfers would slow down the overall execution time and make it always worse than the sequential one.
The plots below show the total time taken for both processes to perform the vecadd, followed by a plot of the time division in the CUDA version.





We can appreciate how the sequential version is always faster overall while in the CUDA version, memory transfers (DtoH & HtoD) eventually take up most of the time while the vecadd operation remains mostly unaltered.

2. **Is the GPU always faster? For a fixed vector size N = $5 \cdot 10^8$, which version is faster in total execution time? Discuss whether offloading to the GPU always guarantees better performance. What are the main factors that determine whether a computation is suitable for GPU acceleration? Consider aspects like memory transfer cost, arithmetic intensity, etc.**

When it comes to vector addition, the OpenACC version is the fastest, but including data transfers it is by far the slowest. Results with CUDA have already been discussed, again, data transfers make it slower.

We can conclude that the cost of memory transfer through the narrow bus that connects the CPU and the GPU is so punishing to performance that it must be avoided whenever possible. Moreover, the high capacity of the GPU to perform parallel processing proves efficient with high arithmetic intensity (AI) operations (the vector addition has very low AI). So, perhaps, the acceleration provided by the device is not worth using for a single vector addition, instead it should be used for operations of high AI like the following matMul or other kind of operations that allow for data to live longer on the device and minimize data transfers.

Concluding, the GPU is not always faster, its limitations require an understanding of the problem to decide whether or not the GPU is suitable for such tasks, having in mind the amount of data that has to be moved and the operations that have to be made.

3. **CUDA vs OpenACC. Compare the performance of the OpenACC and CUDA implementations for N = $5 \cdot 10^8$. Which one is faster? What happens if you use pinned host memory in the CUDA version (HINT: cudaMallocHost(...))? Reflect**

on the trade-off between programming effort and performance control when using CUDA versus OpenACC.

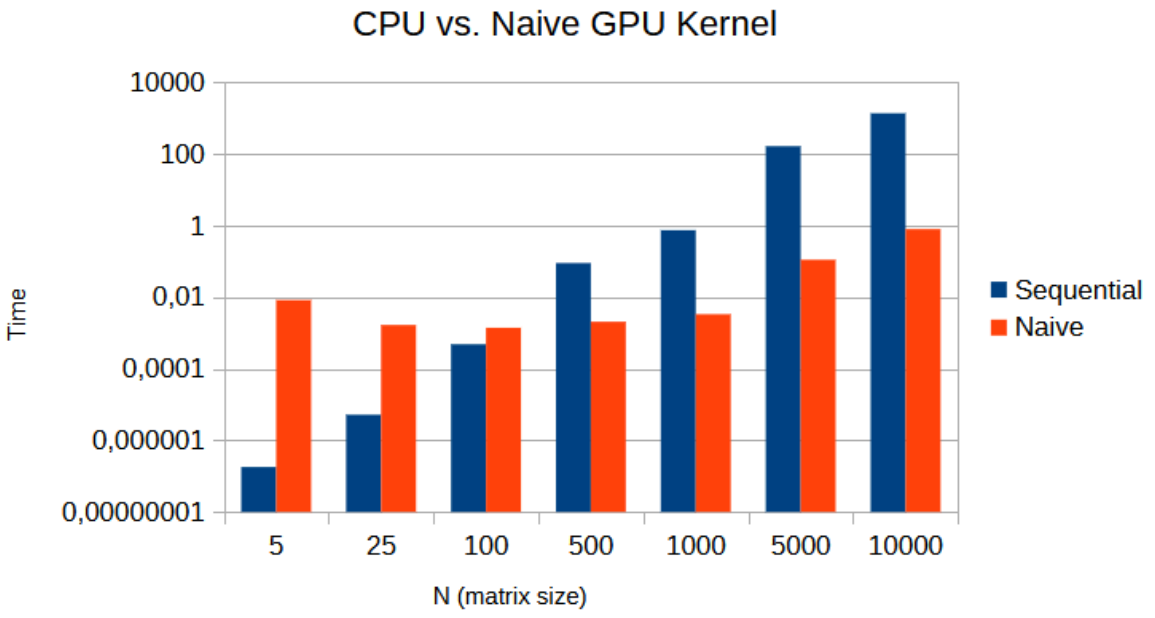
	CUDA	CUDA (pinned)	OpenACC
time (s)	0.087058	0.049328	0.370210428

As discussed previously, OpenACC is always slower than CUDA due to the pile up of transfer times. When using pinned memory for the C vector, the transfer times DtoH are reduced, almost halving the total time of execution.

Benefits of CUDA are obvious, as it gives full control of the memory transfers in many ways, this produces a code that is self explanatory and predictable when read by the programmer. Of course that carries an extra load of work when writing CUDA code as it requires the use of its specific functions and syntax. Meanwhile OpenACC presents an easier implementation, as the compiler directives can be applied directly to the sequential code and require much less typing, but reduces control and readability (sometimes, it's harder to predict). Overall, the election between CUDA and OpenACC is highly dependent on the context. When it comes to implementing GPU acceleration to code that's currently sequential, perhaps OpenACC can be a good choice to begin with, while projects that start from scratch or need thorough control of data movement might want to go for CUDA.

Exercise 2: Matrix Multiplication

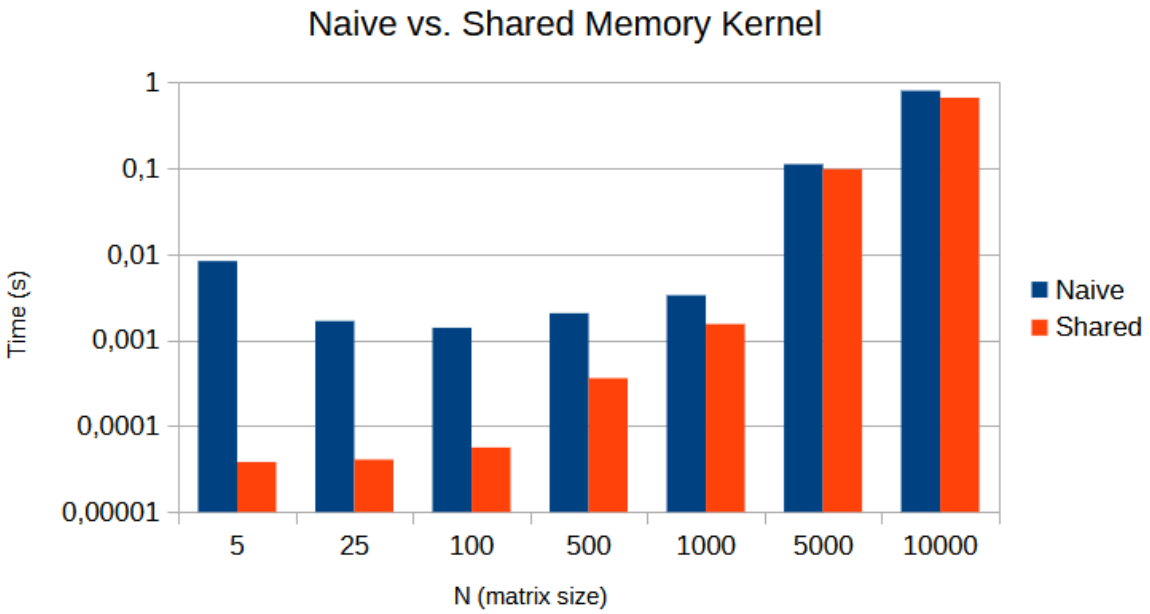
1. **CPU vs. Naive GPU Kernel. Compare the performance of the sequential CPU implementation with the naive CUDA kernel. Run experiments with increasing matrix sizes. Is matrix-matrix multiplication a good candidate for GPU acceleration? Justify your answer based on the computation-to-memory ratio, parallelism, and observed results.**



Based on these observations, we concluded that matrix-matrix multiplication is an excellent candidate for GPU acceleration.
Matrix multiplications have high arithmetic intensity (O(N^3) operations (2 for-loops to traverse the matrix, and one more to calculate each value) for O(N^2) data values)), making them a compute-bound operation. GPUs allow us to parallelize the heavy part of this process, these computations, decreasing massively the time required to perform it, because as each output element is independent, we can assign one thread to each and compute all, or at least a significant chunk, at the same time.
From this data, we observe that at first, a sequential implementation is by far faster than the GPU kernel, which may be caused by the data transfer overhead, from host to device and vice-versa. However, somewhere between N = 100 and N = 500 there's a threshold where this tendency is inverted. We see how the sequential implementation's computation time keeps growing fast, while the GPU, although growing too, is slower. We tested a wide range of matrix sizes to be able to see this difference. We calculated here how much faster is the GPU naive implementation with respect to the Sequential, and we believe that the fact that for N = 10000 matrices a naive implementation on a GPU runs almost 1720 times faster is a strong indicator that matrix-matrix multiplication is an excellent candidate for GPU acceleration .

N	**Sequential**	**Naive**	**Speedup**
5	0,00000018	0,008334978	0,000
25	0,0000052	0,00166992	0,003
100	0,000484537	0,001392128	0,348
500	0,090030009	0,00205696	43,768
1000	0,746424541	0,003335552	223,778
5000	166,316465985	0,111696512	1.489,003
10000	1378,259252307	0,801430941	1.719,748

2. **Naive vs. Shared Memory Kernel. Evaluate the performance difference between the naive kernel and the shared memory version. How much impact does shared memory have on performance? Is the improvement consistent across different matrix sizes? Explain the reasons behind the observed behavior**



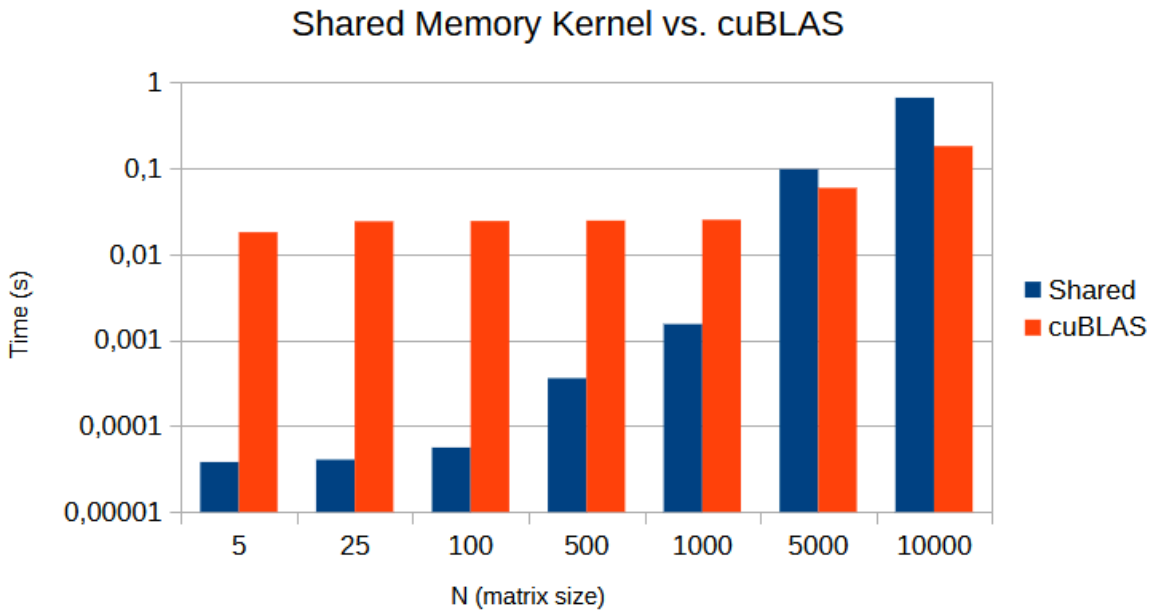
N	Naive GPU (s)	Shared GPU (s)	Speedup
5	0.00833	0.000038	218,70
25	0.00167	0.000041	41,09
100	0.00139	0.000056	24,76
500	0.00206	0.00036	5,71
1000	0.00334	0.00154	2,17
5000	0.1117	0.09755	1,14
10000	0.8014	0.6638	1,21

In this case we observe that the most dramatic improvements are achieved on really small matrices, because shared memory eliminates most global memory accesses, which is what usually causes the overheating that spikes runtime for small matrices where compute time is negligible. For larger matrices, we observe the time difference is smaller overall, which could happen because the whole process becomes more compute-bound, and products take far more time than memory transfers. We believe we could improve this by increasing the block size, as long as the required data still fits in the L1 cache.

As a conclusion we can say the improvement on small matrices is huge because the total time is bound mainly by the memory access time, the computation time is almost negligible, and the Shared GPU is excellent to solve that. But as N becomes large enough, the computation time begins to take over the total time, and the memory access time, which grows at a much steadier rate, doesn't quite keep up, so the whole operation is bound by

computations again. We'd like to mention too that the shared GPU version has a lot of operation overheat, because it has to calculate a lot of indices and evaluate conditionals (for example, to keep the operations within bounds when N isn't a multiple of BLOCKSIZE), it has to create and fill the tiles, and we have a couple of barriers, and all this also affects its performance.

3. **Shared Memory Kernel vs. cuBLAS. Compare your best custom implementation to the highly optimized cuBLAS version. How close does your implementation get to cuBLAS performance? What are the main difficulties in achieving high performance with custom CUDA code? When is it worth writing custom kernels versus using GPU libraries?**



From this data we observe that our custom Shared Memory implementation outperforms significatively the cuBLAS one up to a size 5000, where cuBLAS starts to take over. We see that the time baseline for the cuBLAS implementation is a little over 25ms for any matrix size, so it won't outperform the Shared until it takes more than 25ms, and this happens in our case, for N = 5000. Here, although cuBLAS's time increases slightly, Shared memory's time increases much more. From here onwards, the time taken by Shared increases significatively with N, while cuBLAS grows slow and steady.

This phenomenon is quite logical, knowing that our Shared implementation is great for relatively small matrix sizes, as explained in the previous question, and that cuBLAS is prepared and optimized for large matrices and compute-bound operations, it uses advanced techniques to achieve that.

We can encounter major difficulties when trying to achieve high performance with custom CUDA code, for example the complexity to manage all the different types of memory from the device to the host, shared memory and so on. For a custom implementation we also have to manage threads, blocks and grids, which can also become confusing. Another problem may be architecture, or the ability of our implementation to run efficiently across different GPUs, which is challenging because all have different specifications, and we risk not using certain features, or underutilizing resources on newer GPUs.

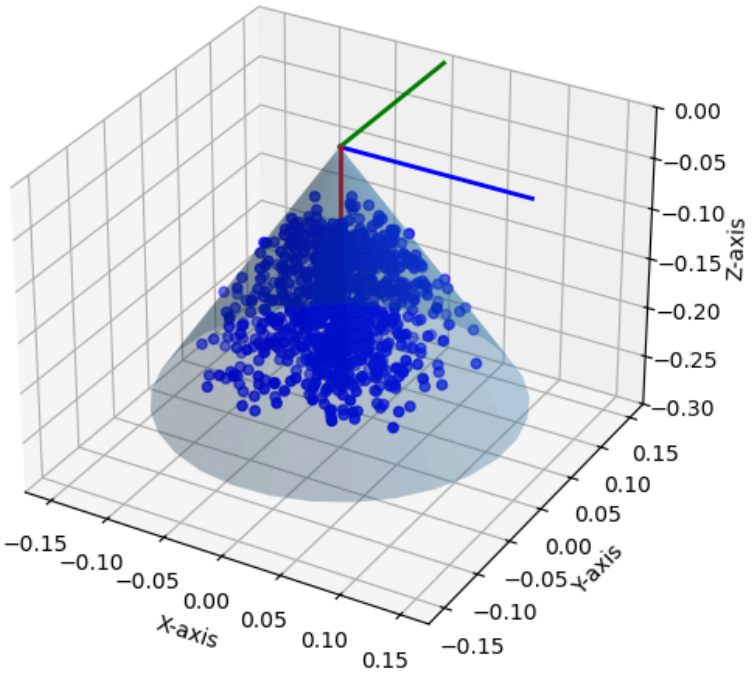
With all this, we concluded that in this case, it'd be worth writing a custom kernel like, for example, our Shared Memory, if we were certain our matrix's size is below a certain threshold, somewhere between 1000 and 5000, we could say 2500 for example. If we know our N will be greater, it's better to use a library implementation, as we saw, for large matrices it's x1.5-3.5 times faster.
In any case, we could use a hybrid implementation, where we check if N is less than our threshold, if it is, we use the custom, and if it isn't, we use cuBLAS.

	Shared	**cuBLAS**	**Speedup**
5	0,000038112	0,018065119	0,00
25	0,00004064	0,024182431	0,00
100	0,000056224	0,024384961	0,00
500	0,000360032	0,024674017	0,01
1000	0,001536896	0,025226817	0,06
5000	0,097553983	0,059256382	1,65
10000	0,663757384	0,18154183	3,66

Exercise 3: Lagrangian Particle Tracking

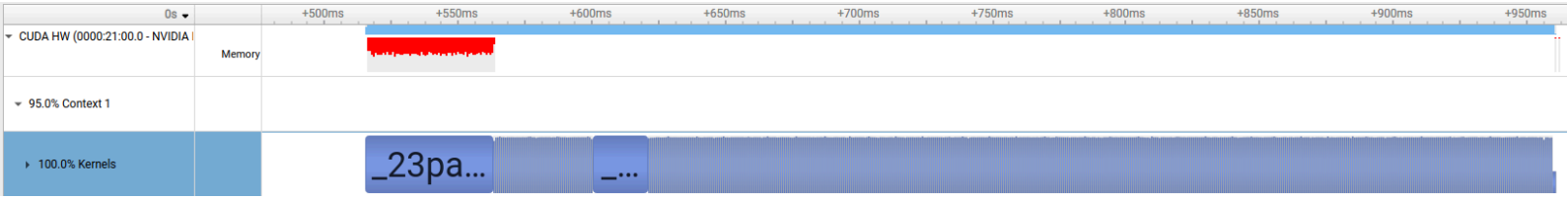
1. **Sequential program. Implement the sequential version of the program and ensure that the validation test passes successfully. Identify the GPU acceleration opportunity inside the while loop—explain why and how the computation can be parallelized. Run the simulation with 1000 particles, saving the solution to disk. Use the provided Python script plot.py to create an animated video of the droplets (note: it requires the .csv files generated in the out folder). Add your favorite snapshot from the animation to the assignment. Before running the script, you will need to create a Conda environment and install the required dependencies:**

```
$ module load conda
$ conda create -n <my_conda_env>
$ conda activate <my_conda_env>
$ conda install matplotlib opencv
$ python plot.py
```



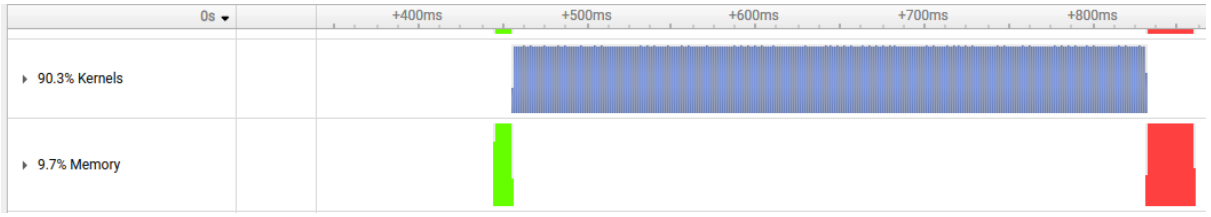
2. **OpenACC: Unified vs Programmer-Managed Memory. Use OpenACC directives to parallelize the computation and create two versions of the program:**
   • In the first version, rely on the compiler to handle memory transfers using CUDA Unified Memory. Compile with the -gpu=managed flag, and use the visual profiler to analyze what happens during execution.
   • In the second version, manually manage memory transfers. Compile with the -gpu=cc90 flag, and use OpenACC data directives and clauses carefully to minimize data movement between host and device. Use the profiler to identify opportunities for optimization.

   Compare the performance of the second OpenACC version against the sequential implementation. Create a bar plot showing execution time versus number of particles, as you did in the first question of the Vector Addition problem. For this performance study, disable output writing by setting the write flag to 0.



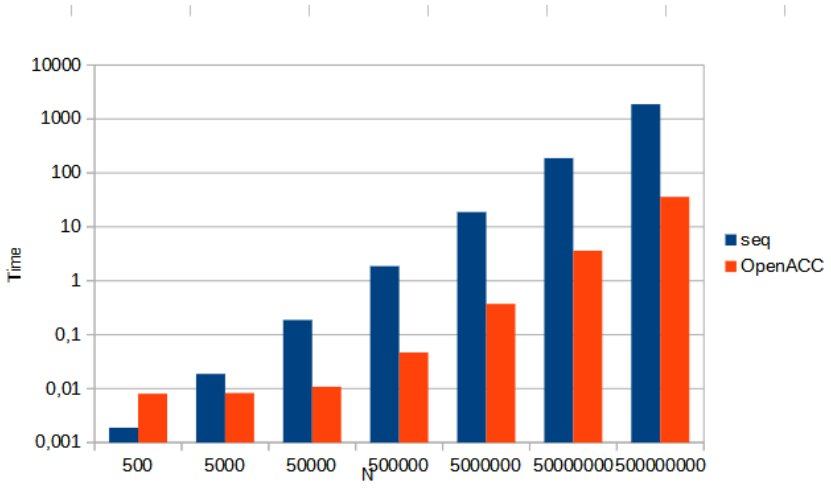
This image of the profiler displays how the first execution of the integratedEuler as well as the copyFrame take significantly longer than the rest,s as the data they use is not allocated yet in the device. You can observe a red region over the first integratedEuler, this indicates page faults due to the data not being available at the moment of the request.

This situation can be avoided loading the data before starting the work, i.e. putting the loop inside a data region:



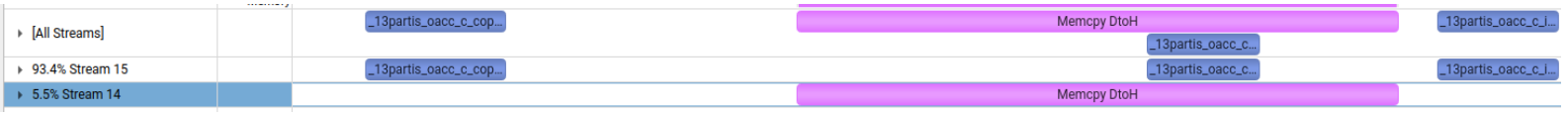
Now we observe how OpenAcc takes care of uploading (green segment) the particles at the beginning of the data region as well as allocating the frame array. This way data is already present when the integratedEuler and copyFrame execute. After the while loop ends, the pFrame array is downloaded back to host (red region). A *present* clause appears in the parallelized loops of integratedEuler and copyFrame to ensure that no data is collected twice.

When comparing this implementation with the sequential one, we get a huge range of values, thus, the following plot is expressed in a logarithmic scale:



We clearly see how the OpenACC implementation is clearly superior beyond 500 particles. At $5*10^8$, the sequential version took over 30' to execute, while the OpenACC version took 35 seconds.

3. **Asynchronous Operations. As you may have noticed, the copyFrame function is used to save snapshots of the transient solution at a fixed frequency—specifically, every 100 iterations. This creates an opportunity to overlap device-to-host memory transfers with ongoing computations during those 100 simulation steps. Use the async and wait clauses to enable concurrent execution of data transfers and computation. For this task, set the write flag to 0 to avoid writing output to disk. Use the visual profiler to verify and provide evidence of the achieved overlap.**



With this image of the profiler you can see how after the execution of *copyFrame(leftmost blue box)* the transfer of data from the Device to the host begins. Shortly after the transfer begins, integratedEuler gets called and starts execution before the *Memcpy* is done.
The actual call to the function is visible in the profiler but has been left out of frame for the picture would look too small and unreadable.

For this implementation, there is also an update call before the loop, that is because on the first call to update, OpenACC allocated pinned memory automatically (cuMemHostAlloc) which took a lot of time, so we trigger that allocation early before the while loop begins so every iteration after can run back to back.