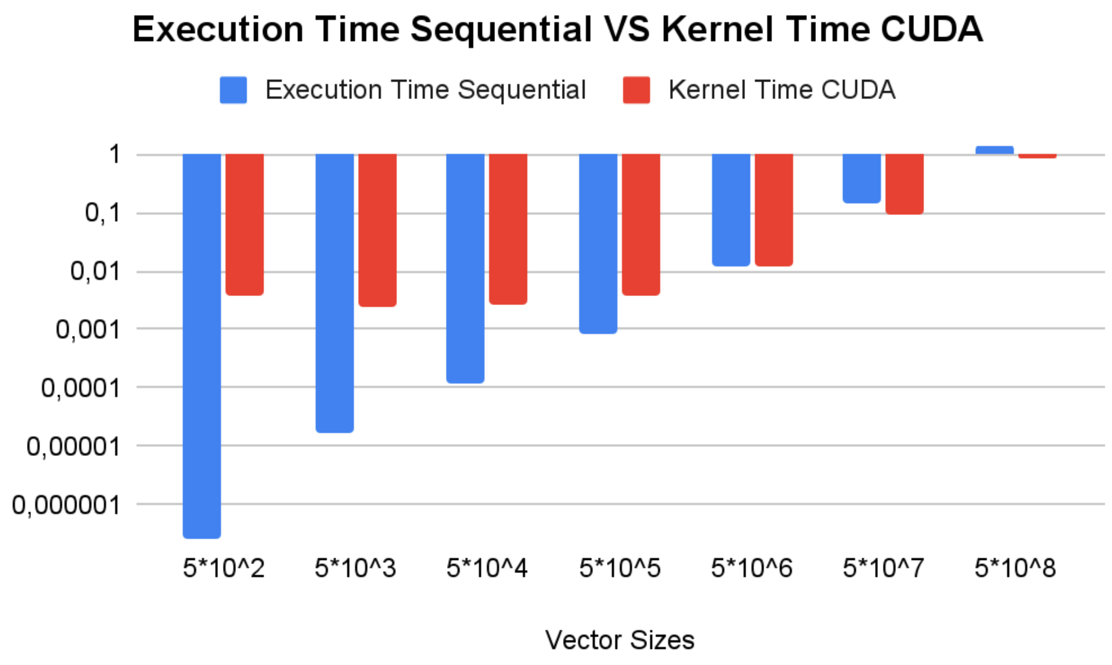
Lab 3: CUDA & OpenACC

Report Questions 1

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 x 10^K, for K = 2, ... , 8. Label the axes clearly.



Vector Sizes	Execution Time Sequential	Kernel Time CUDA
5*10^2	0,00000026	0,003925472
5*10^3	0,00001711	0,002483584
5*10^4	0,000123772	0,002601152
5*10^5	0,000843435	0,003791776
5*10^6	0,012130736	0,011739456
5*10^7	0,143521113	0,092015137
5*10^8	1,399306962	0,819289124

As the vector size increases, the execution time of the sequential version grows rapidly due to its linear processing model.

In contrast, the CUDA version initially performs worse due to GPU Kernel launch overhead and a minimal parallel workload at small sizes. However, from *$N=5\times10^6$ onward*, the CUDA kernel outperforms the sequential version. This intersection point marks when the benefits of massive parallelism outweigh the setup costs of GPU execution. Beyond this size, the GPU handles larger data more efficiently, and the kernel time grows more slowly than the sequential time. *This behavior reflects the expected scaling advantage of parallel architectures.*

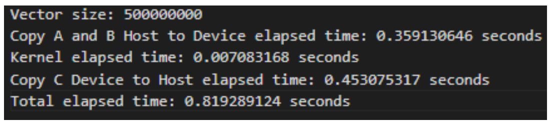
2. Is the GPU always faster? For a fixed vector size N = 5 x 10^8, which version is faster in total execution time? Discuss whether offloading to the GPU always guarantees better performance. What are the main factors that determine whether a computation is suitable for GPU acceleration? Consider aspects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

The GPU is not always faster than the CPU, especially when considering the total execution time. In our experiment, for a fixed vector size of N=5×10^8, the CUDA kernel executes faster than the sequential version, but once we include the time spent transferring data between host and device, the total execution time becomes similar or may even favor the CPU depending on the system.

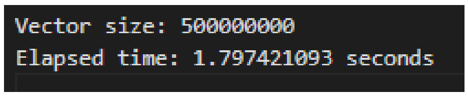
Offloading to the GPU only improves performance when the benefits of parallel processing outweigh the costs of memory transfer. This depends on the arithmetic intensity of the task, which refers to how many operations are performed per unit of data. If the computation involves a small amount of work per element (low arithmetic intensity), then the overhead of moving data to and from the GPU can dominate the execution time. However, when the task involves many operations per data item and the workload can be divided across thousands of threads, the GPU can perform much better.

Other important factors include how well the memory access patterns are optimized and whether the task is easily parallelizable. *Overall, GPU acceleration is most effective for large, parallel tasks with high arithmetic intensity trying to overweigh memory transfer overhead.*

3. CUDA vs OpenACC. Compare the performance of the OpenACC and CUDA implementations for N = 5 x 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.



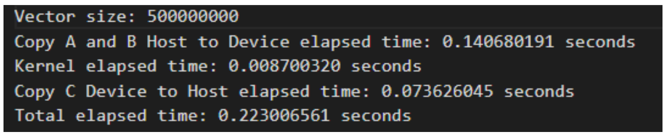
```
Vector size: 500000000
Copy A and B Host to Device elapsed time: 0.359130646 seconds
Kernel elapsed time: 0.007083168 seconds
Copy C Device to Host elapsed time: 0.453075317 seconds
Total elapsed time: 0.819289124 seconds
```



```
Vector size: 500000000
Elapsed time: 1.797421093 seconds
```

For a vector size of N = 5x10^8, *the CUDA implementation outperforms the OpenACC version in terms of total execution time*. In our experiment, the CUDA code was completed in approximately 0.82 seconds, while the OpenACC code required around 1.80 seconds. This performance difference is due to *CUDA's more precise control over execution details such as thread configuration, memory accesses, and kernel optimization*. CUDA has allowed us to tailor the implementation closely to the hardware, resulting in better use of the GPU's capabilities. In contrast, OpenACC abstracts much of this control in favor of simpler syntax and *reduced programming effort*, which is helpful for quick development but can lead to less efficient execution.

One potential optimization in CUDA is to use pinned host memory, allocated via cudaMallocHost() instead of malloc(), for host arrays. Pinned memory prevents the OS from "paging" the memory (which means that the memory can be moved from the physical RAM to the disk, which has a performance cost), enabling the GPU to transfer data more efficiently via direct memory access (DMA). *This reduces memory transfer times*, which is particularly beneficial when transferring large arrays as in our experiment. When host memory is pinned, cudaMemcpy() can achieve higher bandwidth and lower latency, which may further reduce the total execution time.

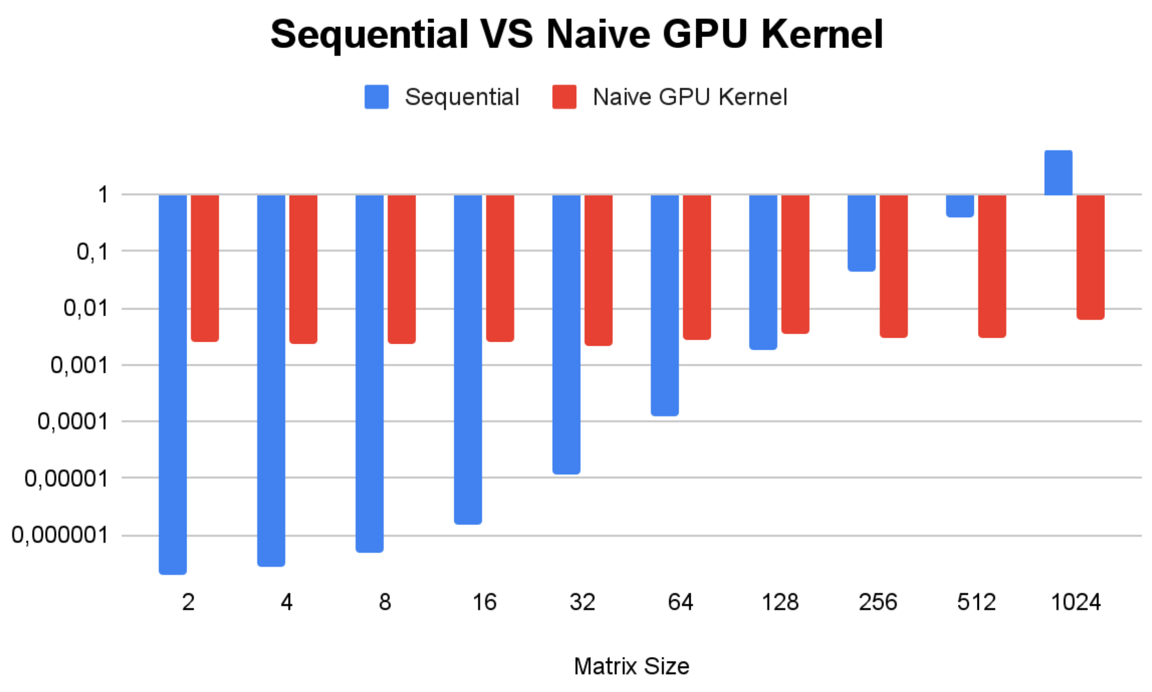

```
Vector size: 500000000
Copy A and B Host to Device elapsed time: 0.140680191 seconds
Kernel elapsed time: 0.008700320 seconds
Copy C Device to Host elapsed time: 0.073626045 seconds
Total elapsed time: 0.223006561 seconds
```

Execution CUDA with pinned memory. We can observe a significant performance improvement in data movement, such as copying A and B to the device and copying C back to the host.

Thereby, the trade-off between CUDA and OpenACC lies in this balance: CUDA offers more performance control but requires a higher programming effort, as developers must manage memory, threads, and synchronization explicitly. OpenACC simplifies development very much through the high-level directives but limits control over lower-level optimizations, which as we saw before can lead to worse performance for complex tasks or large-scale data.

Report Questions 2

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.



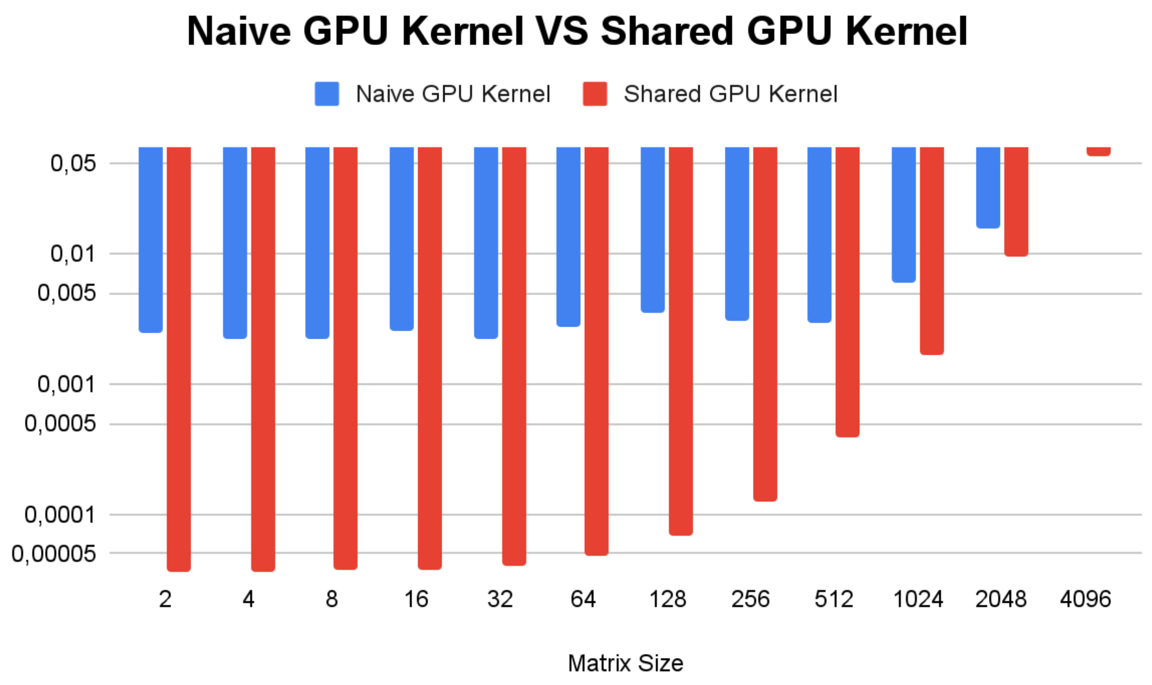
Matrix-matrix multiplication is an excellent candidate for GPU acceleration, as evidenced by the plot.

Matrix multiplication has a high arithmetic intensity. For multiplying two $N \times N$ matrices, the operation performs $O(N^3)$ computations while accessing only $O(N^2)$ memory. This means the number of calculations far exceeds the number of memory accesses, making it ideal for GPUs. Since matrix multiplication minimizes memory overhead relative to computation, it leverages the GPU's computational power efficiently.

Matrix multiplication is inherently parallelizable. Each output element $C[i][j]$ in the resulting matrix can be computed independently, allowing for a natural mapping to GPU threads. Each thread (or group of threads) can compute a single element or a tile of elements in parallel. GPUs, with their thousands of cores, exploit this parallelism far more effectively than CPUs, which are limited to a small number of cores.

For small matrices (N ≤ 128), the sequential CPU implementation is faster due to the overhead of launching GPU kernels and memory transfers. For tiny matrices, this overhead dominates. However, as the matrix size increases, the GPU's advantage becomes overwhelming, achieving speedups of over 1000 times compared to the CPU.

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.



Matrix Size	Naive GPU Kernel	Shared GPU Kernel
2	0,002486848	0,000035584
4	0,002257952	0,000035936
8	0,002258784	0,00003776
16	0,002528864	0,000037952
32	0,002212064	0,000040256
64	0,002742112	0,000048736
128	0,003544512	0,000069344
256	0,003054432	0,000126336
512	0,002974368	0,000386176
1024	0,005983072	0,001675328
2048	0,015589952	0,009620864
4096	0,066522978	0,05661799

The shared memory kernel is significantly faster than the naive kernel for all matrix sizes. The speedup ranges from 70x for small matrices to 1.2x for larger matrices. The improvement is best seen for small matrices, where the naive kernel performs worse due to overheads like thread scheduling and memory latency, as each thread repeatedly loads matrix elements from slow global memory. Shared memory as each thread repeatedly loads matrix elements from slow global memory that helps mitigate these issues by reducing global memory accesses.

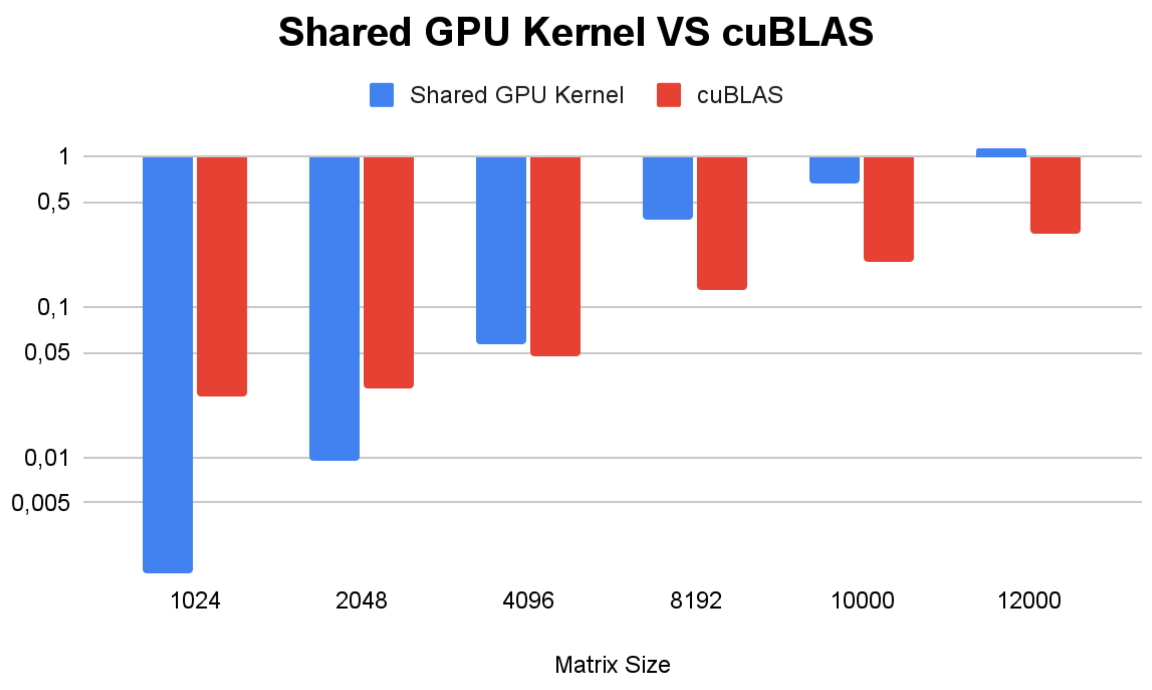
Yes, improvement is consistent in terms of trend (always faster). But not uniform in magnitude, Smaller matrices show huge relative improvements due to overhead masking and memory access efficiencies. Larger matrices benefit less relatively, though the absolute time saved is still significant.

Shared memory has a substantial and positive impact on GPU matrix multiplication performance. The gains are more pronounced for smaller to medium sizes due to efficient reuse of data and lower overhead. For larger matrices, the benefit is still present but diminished by shared memory size limits, synchronization costs, and increased complexity of managing tiles.

Shared memory reduces global memory accesses that instead of reading A and B from global memory for every computation, tiles are loaded into shared memory and reused. It also has better memory coalescing that allows threads to access contiguous memory locations, improving efficiency. Reduces redundant work because threads in the same block collaborate to load data once and reuse it.

For very large matrices, the speedup decreases primarily because the increased computation dominates, as the arithmetic intensity (FLOPs per byte) becomes so high that the GPU shifts from being memory-bound to compute-bound. Additionally, each block has limited shared memory, meaning larger matrices require more blocks, which introduces overhead. Furthermore, while kernel launch overhead is significant for small matrices, making shared memory optimizations more impactful, for large matrices, the kernel execution time overshadows this overhead, reducing the relative benefit of optimizations.

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?



Matrix Size	Shared GPU Kernel	cuBLAS
1024	0,001675328	0,025091616
2048	0,009620864	0,02838368
4096	0,05661799	0,046660479
8192	0,379956871	0,131706268
10000	0,672084391	0,200514108
12000	1,145674229	0,305708915

A custom shared memory GPU kernel significantly outperforms cuBLAS for small matrix sizes (e.g., 1024 and 2048), where cuBLAS exhibits disproportionately high times, likely due to initialization overheads or suboptimal batching. However, as the matrix size increases, cuBLAS rapidly outpaces the custom kernel. For instance, at size 8192, cuBLAS is approximately 3 times faster, and this gap widens as we reach size 12000, where cuBLAS is nearly 4 times faster.

cuBLAS optimizes compute efficiency (FLOPs) by reducing latency and maximizing parallelism, but it doesn't change the GPU's memory bandwidth. The hardware's bandwidth limit (GB/s) stays fixed, cuBLAS just ensures computations get closer to peak FLOPs. The critical trade-off is arithmetic intensity (FLOPs/byte): higher intensity means compute-bound performance. Ultimately, cuBLAS can't overcome bandwidth limits but minimizes wasted cycles.

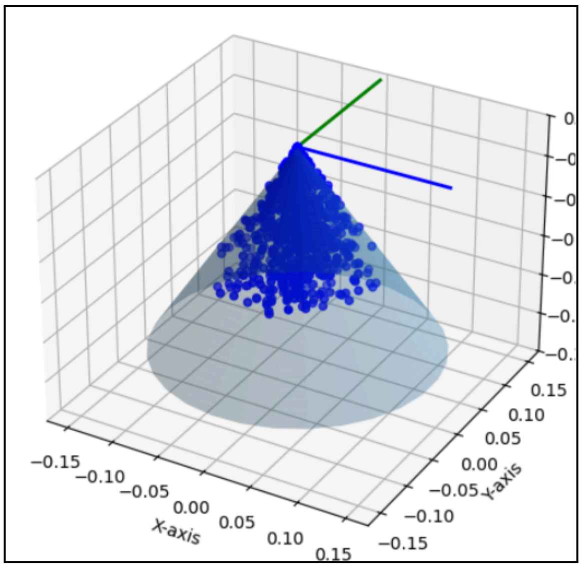
Writing high-performance custom CUDA kernels is challenging due to several key factors. First, optimizing memory usage is complex, properly leveraging shared memory, registers, and cache hierarchies requires careful tiling strategies to minimize slow global memory accesses. Poor memory access patterns lead to issues like uncoalesced reads or shared memory bank conflicts, drastically reducing performance. Second, efficiently utilizing GPU parallelism is difficult. Thread blocks must be sized correctly to avoid underutilized groups of threads, and divergent branching can serialize execution.

Additionally, modern GPUs have specialized hardware like tensor cores that are hard to exploit manually. Finally, achieving peak performance requires architecture-specific tuning for different GPU generations), which libraries like cuBLAS handle automatically but custom implementations often miss. These factors make it extremely difficult to match highly optimized library performance without deep expertise.

It is only worth writing custom CUDA kernels when the problem cannot be expressed with standard library routines or fine-tuned control over every GPU operation is necessary.

Report Questions 3

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. Simulate 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.



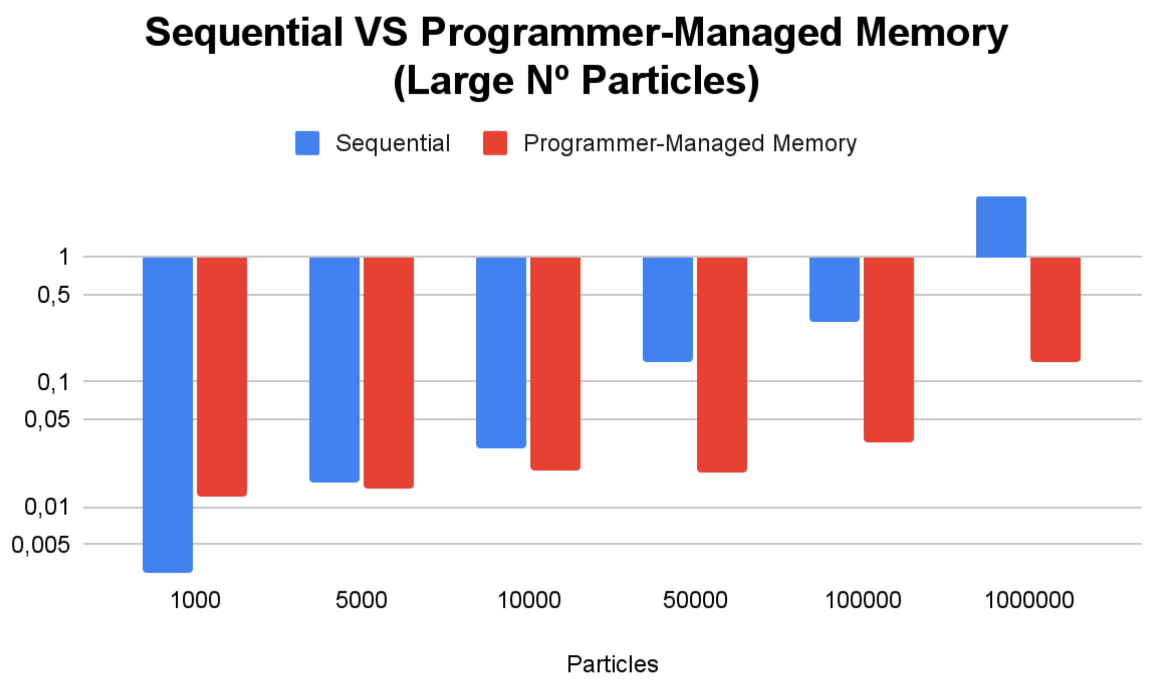
One could implement a GPU acceleration in the particle-update loop. At each iteration, reads and writes only a particle's position and velocity, so there are no particle dependencies. Because every particle does the same arithmetic job, you can map each loop iteration onto its own GPU thread, letting the hardware execute all N updates concurrently while combining memory accesses.

2. OpenACC: Unified vs Programmer-Managed Memory. Use OpenACC directives to parallelize the computation and create two versions of the program:

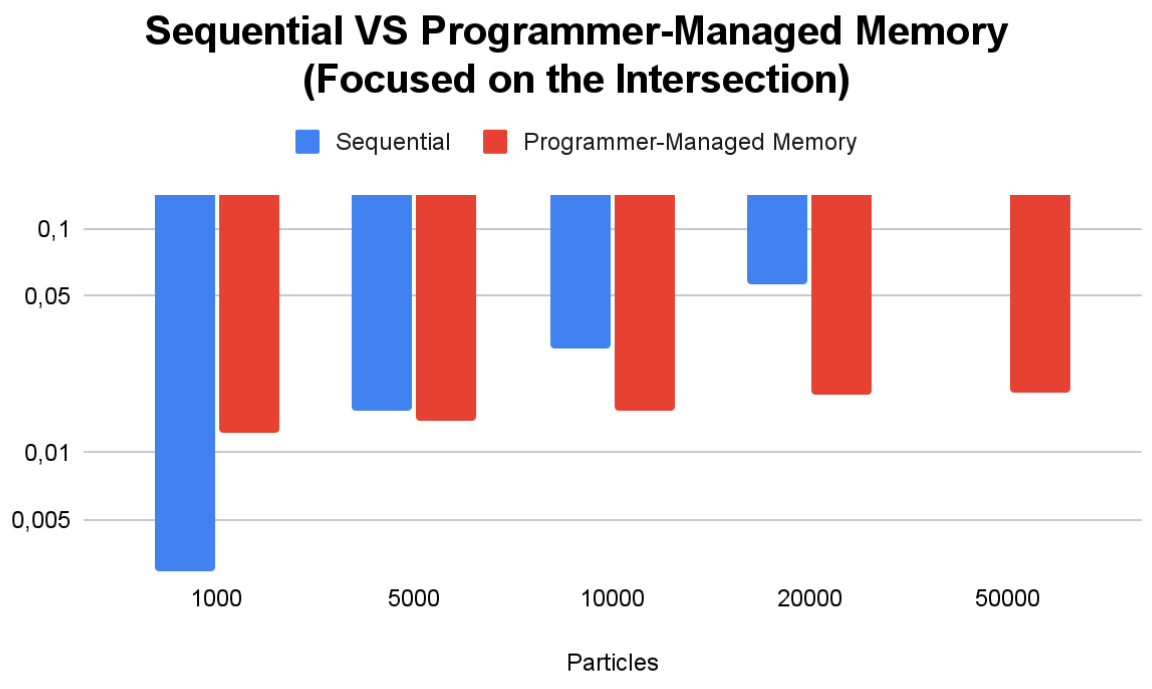
• *In the first version, rely on the compiler to handle memory transfers using CUDA Unified Memory. Compile with the -gpu=managed flag, and use the visual profiler to analyze what happens during execution.*

• *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.



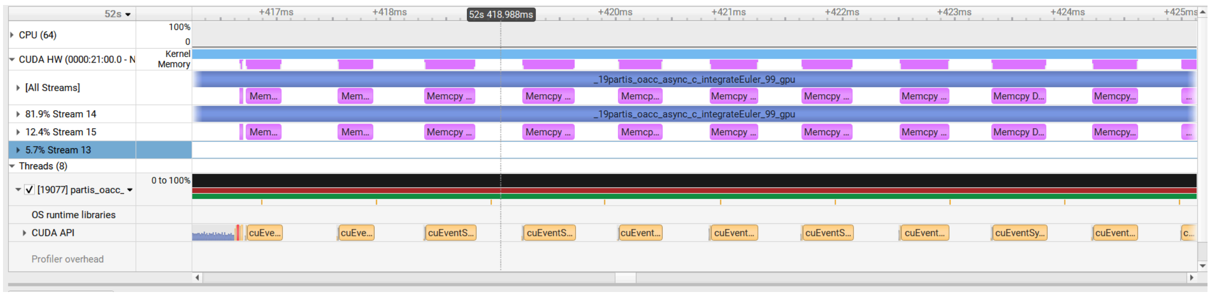
Particles	1000	5000	10000	50000	100000	1000000
Sequential	0,00291	0,015384	0,029038	0,14257	0,302498	3,085193
Programer-Managed Memory	0,012274	0,013754	0,019765	0,018649	0,032613	0,144062

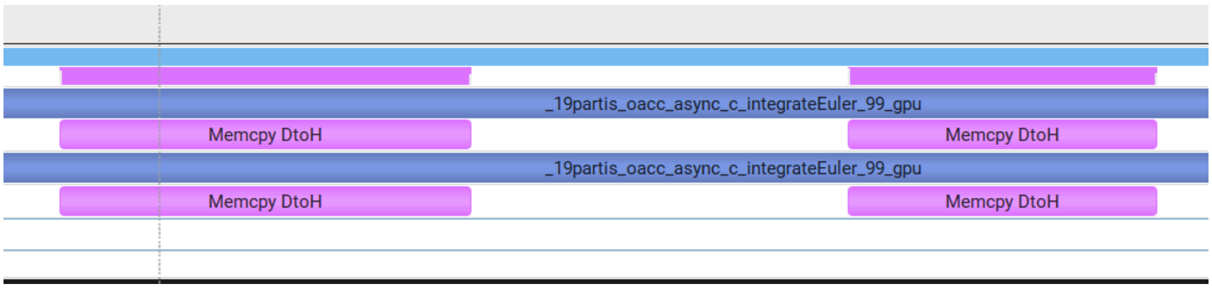


Particles	1000	5000	10000	20000	50000
Sequential	0,00291	0,015384	0,029038	0,056969	0,14257
Programmer-Managed Memory	0,012274	0,013754	0,015384	0,017967	0,018649

Here, one can observe two graphs showing the number of particles over elapsed time in seconds. The first graph presents a large scope of particles, and it is clear that the managed version performs significantly better. In the second graph, one can closely observe the intersection between the two programs.

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.




_19partis_oacc_async_c_integrateEuler_99_gpu
Memcpy DtoH
Memcpy DtoH
_19partis_oacc_async_c_integrateEuler_99_gpu
Memcpy DtoH
Memcpy DtoH

In these pictures, one can see the purple "Memcpy DtoH" bars running beneath the blue integration kernels. This confirms that thanks to Asynchronous Operations, the device-to-host transfer is happening concurrently with the integrateEuler computations.