1. Vector Addition

Vector addition is a fundamental operation in linear algebra, where each element of the result vector is obtained by summing the corresponding elements of two input vectors:

$$c_i = a_i + b_i, \quad i = 0, 1, \ldots, n-1,$$

where $a = [a_0, \ldots, a_{n-1}]$, $b = [b_0, \ldots, b_{n-1}]$ and $c = [c_0, \ldots, c_{n-1}]$ are vectors of double precision floating point numbers.

Your task is to implement three version of the vector addition:

- A **sequential** CPU version written in standard C.
- A parallel GPU version using the **CUDA** programming model.
- A parallel GPU version using **OpenACC** compiler directives.

1.1. Guidelines for coding the sequential version:

- **Command-Line Argument:**
  The program must accept a single command-line argument `N`, which defines the size of the vectors.
  If the argument is not provided, the program must print the following usage message and terminate:

  ```
  Usage: ./vecadd_seq <vector size N>
  ```

- **Vector Initialization:**
  Use the following formulas to initialize the vectors:

  $$A_i = i, \quad B_i = 2 \cdot (N - i)$$

  This pattern makes it easier to catch indexing errors when the computation is later parallelized on the GPU.

- **Function Prototype:**
  Implement the sequential vector addition in a function with the following signature:

  ```
  void vecadd_seq(double *A, double *B, double *C, const int N);
  ```

  Here, `A` and `B` are the input vectors, `C` is the result vector, and `N` is the number of elements.

- **Timing:**
  Use the `clock_gettime` function from `<time.h>` with the `CLOCK_MONOTONIC` clock to measure execution time. For nanosecond precision, combine the `tv_sec` and `tv_nsec` fields from the `struct timespec`.

- **Validation:**
  After computation, validate the result by checking that each element in `C` satisfies:

$$C_i = 2 \cdot N - i$$

  with an absolute error tolerance of less than 10^{-6}.

- **Program Output:**
  The program must print the following summary to standard output:

```
Vector size: 500000000
Elapsed time: 0.425861260 seconds
```

1.2. Guidelines for the CUDA Version

In order to work with GPUs on the `pirineus3.csuc.cat` cluster:

- To access the NVIDIA compilers and CUDA development tools, load the required module with:

```
module load nvhpc/24.9
```

- When submitting a job that requires a GPU, make sure to request one by including the following directive in your submission script:

```
#SBATCH --gres=gpu:1
```

In this version, the goal is to port the sequential vector addition code to run in parallel on the GPU using CUDA. Each CUDA thread will compute one element of the result vector `C`. The structure of the code should follow these steps:

- **Memory Allocation:**
  Allocate memory on both the host and the device.

- **Host Initialization:**
  Same as the sequential version.

- **Host to Device Transfers:**
  Copy vectors from host to device. **HINT:** You can reduce data transfer overhead by initializing the device memory for `C` directly to zero using the `cudaMemset(...)` function, instead of initializing it on the host and copying it to the device.

- **Grid Configuration:**
  Calculate threads per block and grid dimension (number of thread blocks). **HINT:** Fix the number of threads per block using the macro:

```
#define BLOCKSIZE 128
```

- **Kernel Launch:**
  Call the kernel that performs vector addition. The kernel signature must be:

```
__global__ void vecadd_cuda(double *A, double *B, double *C, const int N);
```

- **Device to Host Transfer:**
  Copy back to the host vectors you think are needed.
- **Validation:**
  Same as the sequential version.
- **Timing:**
  Time the following separately using CUDA events (`cudaEvent_t`):
  - Host to Device copy
  - Kernel execution
  - Device to Host copy
- **Error Handling:**
  For clean and maintainable code, use a macro to check CUDA error codes:

```
#define CUDA_CHECK(call)                                                    \
    do {                                                                    \
        cudaError_t err = (call);                                           \
        if (err != cudaSuccess) {                                           \
            fprintf(stderr, "CUDA error at %s:%d: %s\n",                    \
                    __FILE__, __LINE__, cudaGetErrorString(err));           \
            return 1;                                                       \
        }                                                                   \
    } while (0)
```

  Example usage:

```
CUDA_CHECK(cudaMalloc(...));
```

- **Program Output:**

```
Vector size: 10000
Copy A and B Host to Device elapsed time: 0.000037280 seconds
Kernel elapsed time: 0.000071968 seconds
Copy C Device to Host elapsed time: 0.000052736 seconds
Total elapsed time: 0.000161984 seconds
```

1.3. Guidelines for the OpenACC Version

- Copy the sequential version of the code into the OpenACC folder and update the vector addition function signature to:

```
void vecadd_oacc(double *A, double *B, double *C, const int N);
```

- Modify the `Makefile` to use the `nvc` compiler with the appropriate flags to enable OpenACC support on the GPU. Ensure you disable implicit memory management and control data transfers manually.
- Use OpenACC directives to parallelize the computation. Explicitly specify all required `data` clauses to manage memory movement between host and device.
- The program output format must be the same as in the sequential version:

```
Vector size: 10000
Elapsed time: 0.056185851 seconds
```

Report Questions 1 — Vector Addition (20%)

1. Compare the sequential version with the CUDA version. At what vector size does the GPU **kernel** become faster then 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 $N = 5 \times 10^k$, for $k = 2, \ldots, 8$. Label the axes clearly.
2. Is the GPU always faster? For a fixed vector size $N = 5 \times 10^8$, 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.
3. CUDA vs OpenACC. Compare the performance of the OpenACC and CUDA implementations for $N = 5 \times 10^8$. 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.

2. Matrix Multiplication

Matrix-matrix multiplication is a core operation in numerical linear algebra. Given two square matrices A and B of size $N \times N$, the product matrix C is defined as:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}, \quad i, j = 0, 1, \ldots, N-1,$$

where $A = [A_{ij}]$, $B = [B_{ij}]$, and $C = [C_{ij}]$ are matrices of double-precision floating-point numbers. Each element C_{ij} is calculated as the dot product of the i-th row of A and the j-th column of B.

The goal of this assignment is to implement and compare four versions of square matrix-matrix multiplication, progressively exploring the optimization opportunities available with CUDA:

1. Sequential CPU implementation.
2. Naive CUDA kernel.
3. Tiled CUDA kernel using shared memory.
4. Use the highly optimized matrix multiplication routine provided by NVIDIA's `cuBLAS` library.

You will measure the performance of each implementation and validate the results against the sequential version.

Instructions

You are provided with a CUDA template file `matmul.cu` with utility functions for memory management, timing, and result validation. Your task is to complete the sections marked with `// TODO`.

The resulting executable `matmul` accepts two command-line arguments:

1. `<matrix size NxN>`: The dimension N of the square matrices to be multiplied.
2. `<check>`: A validation flag. Set to `1` to:
   - Compute the reference result using the sequential CPU implementation.
   - Compare each GPU-based result (naive kernel, shared memory kernel, and cuBLAS) against the reference for correctness.

   Set this flag to `0` to skip the sequential computation and validation checks (especially when measuring GPU performance of big matrices).

1. Sequential Version (CPU)

Complete the CPU matrix multiplication function:

```
void matmul_seq(double *A, double *B, double *C, const int N);
```

2. Naive CUDA Kernel

You are now asked to implement a CUDA kernel to parallelize the matrix multiplication. In this version, each CUDA thread should compute one element of the output matrix C. This means that you will need a 2D grid of threads, where each thread identifies a unique pair (i, j), corresponding to row i of matrix A and column j of matrix B (see Figure 1). Complete the simple matrix multiplication CUDA kernel:

```
__global__ void matmat_naive_kernel(double *A, double *B, double *C, const int N);
```

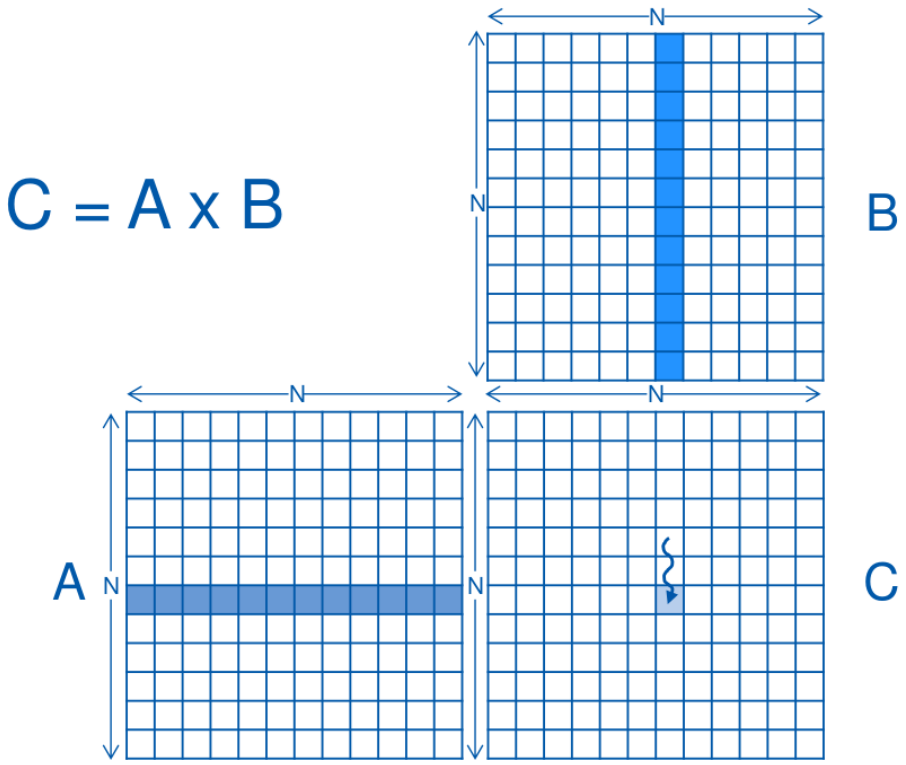

Figure 1: Simple parallelization approach, assign one thread to each element in the output matrix.

To achieve this, you will launch a 2D grid of 2D thread blocks. As in the *Vector Addition* exercise, to determine the number of thread blocks, we fix the number of threads using the following macro: `#define BLOCKSIZE 16`.

Guidelines for the kernel:

- Compute the global row and column indices i and j that correspond to the position in matrix C for each thread.
- Make sure to include bounds checking: not all threads will correspond to valid matrix indices if N is not divisible by the block size.
- Implement the loop that computes $C_{ij} = \sum_k A_{ik} B_{kj}$.

3. Tiled Shared Memory CUDA Kernel

To improve performance over the naive implementation, you will now implement a **tiled** matrix multiplication kernel using **shared memory**. This technique reduces the number of global memory accesses by loading sub-matrices (tiles) of A and B into fast shared memory, allowing data reuse among threads in the same block. Figure 2 might help to have a more graphical overview of the kernel algorithm.

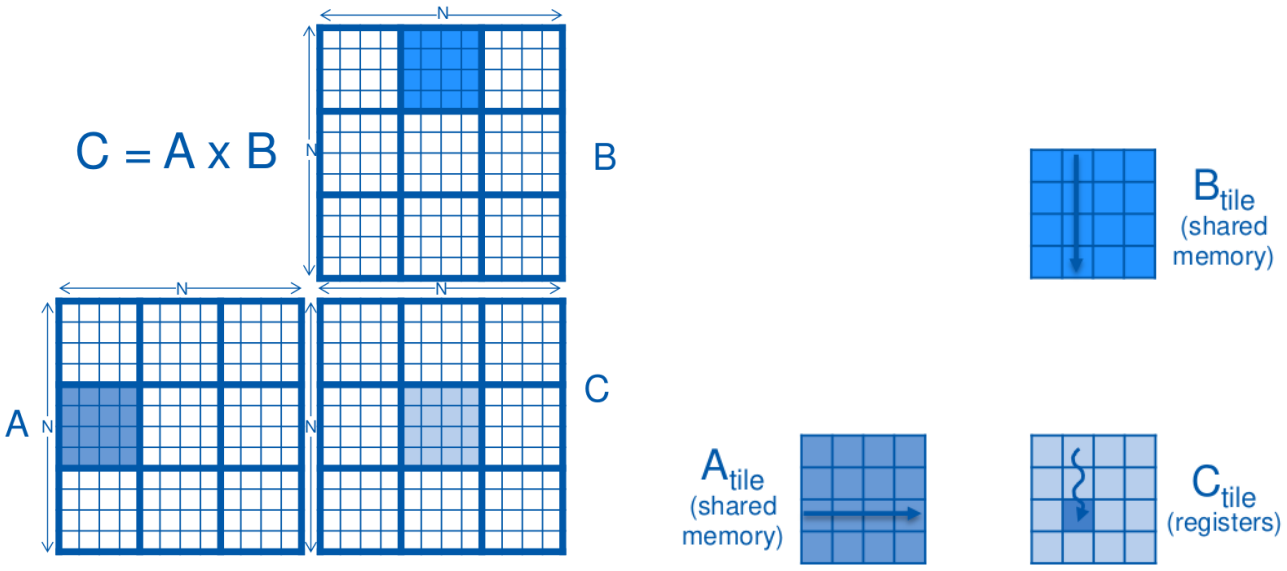

Figure 2: Tiled Matrix-Matrix Multiplication.

Each thread block computes a `BLOCKSIZE` × `BLOCKSIZE` tile of the output matrix C. Threads cooperate to load the corresponding tiles of A and B into shared memory and then perform a partial computation. The final result is accumulated over all the required tiles.

Guidelines for the kernel:

- Declare two shared memory tiles, for A and B matrices:

```
__shared__ double A_tile[BLOCKSIZE][BLOCKSIZE];
__shared__ double B_tile[BLOCKSIZE][BLOCKSIZE];
```

  Each thread will help load one element of A and B into these shared memory tiles.

- Compute the global row index i and column index j of the output matrix C for which the thread is responsible.

- Loop over all the tiles of A and B that are needed to compute the output tile. For each tile step:
  - Load a submatrix (tile) of A and B from global memory into shared memory.

  - **Synchronize threads with `__syncthreads()`** to ensure all loads are complete.
  - Perform the partial dot product computation using data in shared memory.
  - **Synchronize again with `__syncthreads()`** before loading the next tile.
- Store the final result to $C[i \times N + j]$, if within bounds.

Hint: Start by designing the kernel for matrices whose dimensions are multiples of the thread block size (e.g., 128, 256, 512, ...). In a second step, generalize the implementation to support arbitrary square matrix sizes by carefully handling boundary conditions when loading data into shared memory and storing the total result in C.

4. Matrix Multiplication with cuBLAS

In this final version, you will use cuBLAS, NVIDIA's GPU-accelerated implementation of the Basic Linear Algebra Subprograms (BLAS) library. BLAS is a standardized set of low-level routines for performing common linear algebra operations such as vector addition, dot products, and matrix multiplication. cuBLAS provides highly optimized versions of these operations that run on NVIDIA GPUs.

Linking cuBLAS: To compile a program that uses cuBLAS, make sure to link with the cuBLAS library by adding the `-lcublas` flag.

Double Precision Matrix Multiplication: cuBLAS provides the function `cublasDgemm()` to perform matrix multiplication in double precision. Refer the official cuBLAS documentation for detailed information about this function here.

Row-Major vs. Column-Major Storage

cuBLAS assumes that all matrices are stored in **column-major** format (as in Fortran), while C and CUDA code typically use **row-major** format. To handle this mismatch, we can use the mathematical property of transposes in matrix multiplication:

$$C = A \cdot B \quad \Rightarrow \quad C^\top = (A \cdot B)^\top = B^\top \cdot A^\top$$

We exploit this identity by computing $C^\top = B^\top \cdot A^\top$ in cuBLAS. Because our matrices are actually stored in row-major format, this computation will yield a result equivalent to $C = A \cdot B$ in row-major order.

Report Questions 2 — Matrix Multiplication (40%)

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.

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.

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?

3. Lagrangian Particle Trancking

The Propulsion Technologies Group of the Barcelona Supercomputing Center is dedicated to the generation of large-scale simulation software for propulsion and power applications. One relevant area of work is modeling liquid fuel injection in the combustion chamber of aeroengines. A Lagrangian description of liquid droplets allows one to determine their trajectory in time. A simplified model considers a droplet as a perfect sphere where the only acting force is the drag force; see Figure 3.

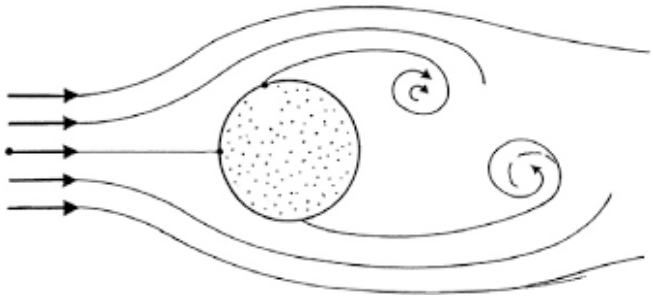

Figure 3: Aerodynamic drag force on a sphere.

To determine the position of the droplet, Newton's second law is applied:

$$\vec{F} = m\vec{a}, \tag{1}$$

$$\vec{F}_D = -\frac{1}{2}\rho A C_D |\vec{v}|\vec{v}, \tag{2}$$

where it can be assumed that air density (ρ), drag coefficient (C_D) and front area (A) of every particle are equal and constant in time, $k = \frac{1}{2}\rho A C_D$. Then, decomposing vectors in the Cartesian coordinates:

$$-k|\vec{v}|\,(v_x, v_y, v_z) = m\left(\frac{dv_x}{dt}, \frac{dv_y}{dt}, \frac{dv_z}{dt}\right) \tag{3}$$

In this simplified model, the position of the particle, as a function of time, has analytical solution. However, for more complex scenarios, where more forces are acting, such as fluid-particle interaction (two-way coupling), evaporation, and/or chemical reactions, the equation can only be solved numerically. Knowing the initial position and the initial velocity, we can discretize the equation in time with the explicit Euler method. The solution of the new particle position and velocity is given by:

$$\begin{aligned} x^{n+1} &= x^n + v_x^n \Delta t \\ y^{n+1} &= y^n + v_y^n \Delta t \\ z^{n+1} &= z^n + v_z^n \Delta t \end{aligned} \tag{4}$$

$$
\begin{aligned}
v_x^{n+1} &= v_x^n - \frac{k|\vec{v}|}{m} v_x^n \Delta t \\
v_y^{n+1} &= v_y^n - \frac{k|\vec{v}|}{m} v_y^n \Delta t \\
v_z^{n+1} &= v_z^n - \frac{k|\vec{v}|}{m} v_z^n \Delta t
\end{aligned} \tag{5}
$$

In order to recreate the initial conditions (position and velocity) of a fuel injection, see Figure 4, all the particles are spawned from the same position, $(x, y, z) = (0, 0, 0)$, with a random initial velocity where its direction is bounded by a cone as it is ilustrated by Figure 5.

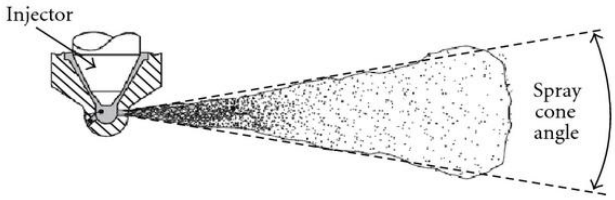

Figure 4: Gasoline Direct Injection (GDI) spray cone angle.

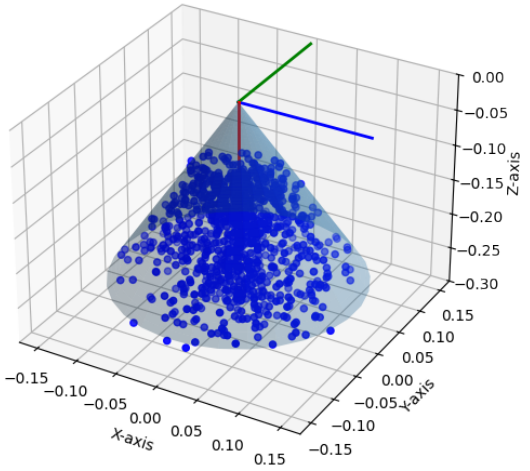

Figure 5: Cone model of a fuel injection spray.

The initial velocity vector is determined by the following formulas:

$$
\begin{aligned}
v_{0x} &= |\vec{v_0}| \sin(\theta) \cos(\phi), \\
v_{0y} &= |\vec{v_0}| \sin(\theta) \sin(\phi), \\
v_{0z} &= -|\vec{v_0}| \cos(\theta),
\end{aligned} \tag{6}
$$

where

- $100 \leqslant |\vec{v_0}| \leqslant 300$.
- $0 \leqslant \theta \leqslant \frac{\pi}{6}$.
- $0 \leqslant \phi \leqslant 2\pi$.

Instructions

Implement the functions `setInitialConditions`, `integrateEuler` and `copyFrame`. Compile the code with the GNU compiler, linking with the standard library for math functions, `-lm`. Run the program with a number of particles, $N = 1000$, and make sure the validation test passes successfully. You are provided with a C source template file `partis_seq.c` with the skeleton of the program.

The resulting executable `partis_seq` accepts two command-line arguments:

1. `<particles N>`: Number of particles to spawn in the simulation.
2. `<write>`: Set to `1` to write the particle solutions on disk for post-processing purposes. Set this flag to `0` when running big simulations.

Profiling with NVIDIA Nsight Systems

To better understand and analyze the performance of your OpenACC programs, you will use Nsight Systems, NVIDIA's visual profiler. This tool provides a timeline-based view of CPU-GPU activity, which is especially useful to inspect memory transfers and kernel executions. You need to install the Nsight Systems GUI on your local machine to visualize the profiling results: https://developer.nvidia.com/nsight-systems/get-started

Profiling is performed in two steps:

1. On the cluster, submit the program using the Nsight Systems CLI profiler:

   ```
   nsys profile ./your_program
   ```

   This will generate a file such as `report1.nsys-rep`.

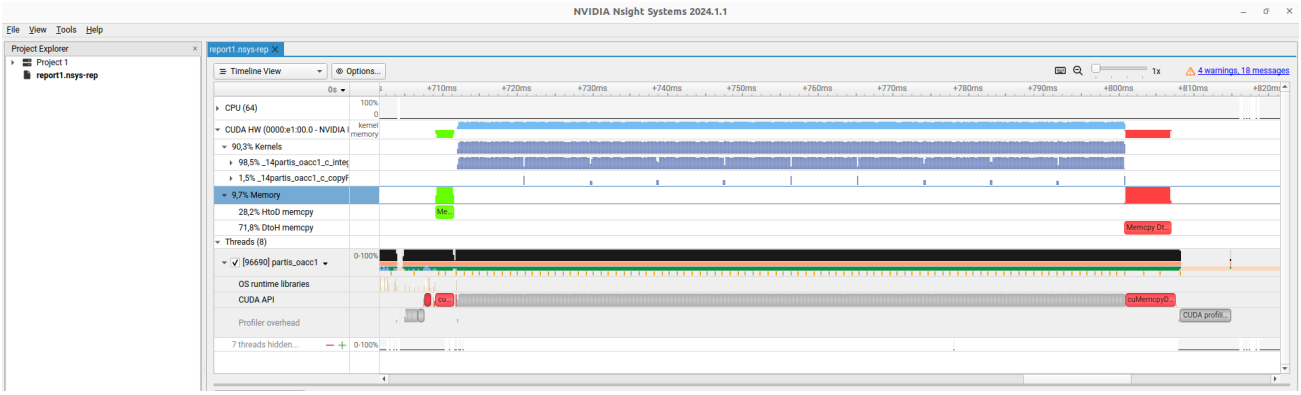

Figure 6: Example of visual profiling of an OpenACC execution of the particle simulation program.

2. Copy the generated profiler output file (`.nsys-rep`) to your local machine and open it using the Nsight Systems GUI. Figure 6 illustrates an example of what you should expect.

Report Questions 3 — Particle Tracking Algorithm (40%)

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.

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.