

Lab 1: First steps in CUDA

A GPU runs thousands of threads at once, but only if you give it the work in the right shape. This lab takes you from the smallest possible CUDA program to a fractal image computed a few hundred times faster than on the CPU. Along the way, you will write your first kernels, learn to launch, check and time them, and meet the two questions that every GPU program raises sooner or later: is the work evenly spread over the threads, and can memory keep up with it?

In this lab you will

- ▶ know your GPU: its number of SMs, the limits on blocks and grids, its peak performance and memory bandwidth;
- ▶ write a kernel, compute the global index of a thread in 1D and 2D, and choose the block and grid sizes;
- ▶ move data between host and device, check every CUDA call, and time a kernel with CUDA events;
- ▶ compare a measured bandwidth with the peak, and observe the effect of coalescing;
- ▶ port a CPU code to CUDA, and discuss precision and load balancing.

How to work. Work in pairs, and answer each question **Q** in `REPORT.md`, where the tables are already laid out. Some questions sit in a *Predict first* box: write down your prediction *before* running anything, and keep it in the report even if it turns out wrong, since understanding why it was wrong is where you learn the most. In the source files, the code you have to write is marked **TODO (n)**.

Part	Topic	Indicative time
0	Setup and your GPU	20 min
1	First kernels: <code>simple</code> , <code>sqrt</code>	40 min
2	Matrix addition: indexing, timing, bandwidth, coalescing	80 min
3	Mandelbrot: porting, precision, load balancing, gallery	remaining time

What you hand in. The three source files you completed in `src/`, your report `REPORT.md`, your benchmark file `results/<team>.csv` with its plot, and your fractal for the gallery.

0 Setup and your GPU

The GPUs are not on your workstation: you reach them through `ssh`, and share them with the other students through the Slurm scheduler. You compile on the login node, then run on a GPU node by prefixing the command with `srun`. Start by building everything and asking the GPU to describe itself:

```
ssh -K login@nash.ensimag.fr
tar xzf CGPU-Fall2026-Lab1.tar.gz && cd CGPU-Fall2026-Lab1
cmake -S . -B build && cmake --build build -j
srun --gres=shard:1 --cpus-per-task=4 --mem=4GB ./build/device_query
```

From now on, rebuild with `cmake --build build -j` after each change. Every program describes its options with `--help`, and `README.md` lists all the commands of the lab.

Q0.1 Fill in the identity card of your GPU in the report. You will need these numbers throughout the lab, to judge whether a measurement is good or poor.

Q0.2 How many threads can be resident on the whole GPU at the same time? This is the amount of parallelism a kernel must expose to keep the GPU busy.

Tip. Since the GPU is shared (shard), timings fluctuate: repeat a measurement if it looks odd. For long runs, `sbatch scripts/run.sbatch <command>` frees your terminal and writes the output to `lab1-<jobid>.out`.

1 First kernels

1.1 The smallest CUDA program

Open `src/simple.cu` and run it:

```
srunch --gres=shard:1 --cpus-per-task=4 --mem=4GB ./build/simple
```

Each thread writes its own index, so the program prints the numbers 0 to 15. Short as it is, it already follows the three steps of every CUDA program: allocate memory on the device, launch the kernel, and copy the result back to the host. Notice also that every call to the CUDA runtime is wrapped in `CUDA_CHECK`; you will soon see why.

Q1.1 How many blocks, threads and warps does `simple` use, and how many SMs at most? Compare with the identity card of your GPU: how much of it is busy?

1.2 Square root of an array

Your first kernel computes the square root of every element of an array, one thread per element. With B threads per block, $\lceil N/B \rceil$ blocks are needed to cover the N elements, and each thread finds the element it owns from its position in the grid:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

Complete `src/sqrt.cu`, from **TODO (1)** (the kernel) to **TODO (5)** (copying the result back). The program then compares your results with the CPU, bit for bit:

```
srunch --gres=shard:1 --cpus-per-task=4 --mem=4GB ./build/gpu_sqrt -n 16
srunch --gres=shard:1 --cpus-per-task=4 --mem=4GB ./build/gpu_sqrt -n 1000000 -b 256
```

Q1.2 When N is not a multiple of B , the last block is only partly used. What is the `if` of your kernel for? Remove it and run the memory checker:

```
srunch --gres=shard:1 --cpus-per-task=4 --mem=4GB \
  compute-sanitizer ./build/gpu_sqrt -n 1000 -b 256
```

What does it report? Did the program crash without the `if`? Put it back.

Predict first.

Q1.3 Will the square roots computed by the GPU be identical to those of the CPU? Check with `gpu_sqrt`, then with `gpu_sqrt_fast`, which is built from the same source with `--use_fast_math`. Can we assume that the GPU and the CPU always give identical results?

Checkpoint. `gpu_sqrt -n 1000000` runs, and reports how many results differ from the CPU.

2 Matrix addition

A one-dimensional array was a gentle start; most data, however, come in two dimensions. We now add two $N \times N$ matrices, $C = A + B$, again with one thread per element. The kernel performs almost no arithmetic, so its speed is set by the memory alone. This makes it ideal to learn how to launch, check and time a kernel, and to compare a measurement with the peak bandwidth of the GPU. The matrices are stored row-major: element (i, j) sits at index $iN + j$.

2.1 2D indexing

With two-dimensional blocks and grids, each thread has two coordinates. Figure 1 shows how they map onto the matrix: x runs along a row, giving the column index j , and y runs down a column, giving the row index i .

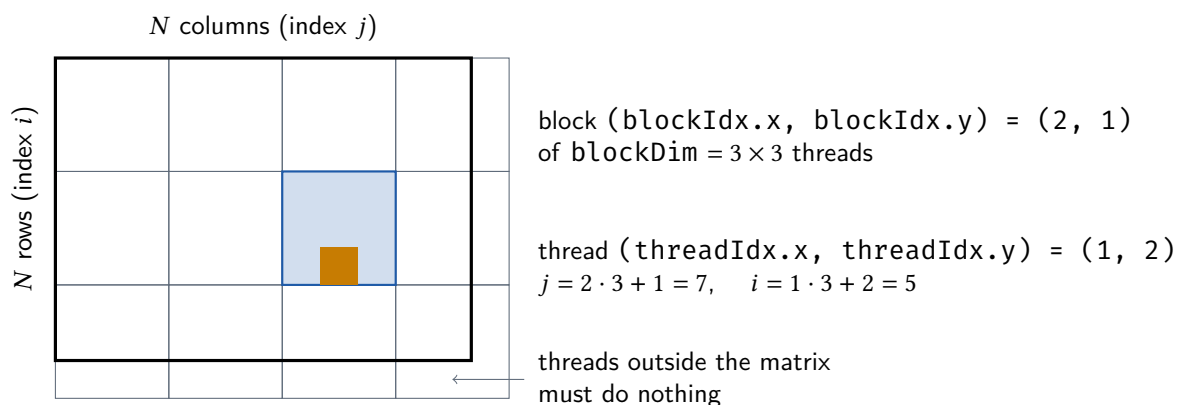


Figure 1: A 4×3 grid of 3×3 blocks covering an 8×11 matrix.

Complete the kernel `add` in `src/matrix_add.cu` (**TODO (1)**). For now, the grid is left at a single block. Is that enough for small matrices? Try two block sizes:

```
srunk --gres=shard:1 --cpus-per-task=4 --mem=4GB ./build/matrix_add -n 16 -b 16
srunk --gres=shard:1 --cpus-per-task=4 --mem=4GB ./build/matrix_add -n 64 -b 64
```

Q2.1 The first run passes, the second does not. What does it say, and why? Which line of the program detected the problem? A kernel launch never returns an error by itself: what would have happened without this check?

Q2.2 A block cannot grow beyond 1024 threads, so large matrices need more blocks. To see it, run `-n 1024 -b 16`: the result is `FAILED`, yet no error is reported. Why? Complete **TODO (2)** so that the grid covers the matrix for any N , and check it with `-n 1000 -b 16`.

2.2 Timing with CUDA events

Now that the kernel is correct, how fast is it? Kernel launches are asynchronous: the host queues the work and moves on immediately, so a CPU timer around the launch measures almost nothing. A CUDA event is a timestamp recorded by the GPU itself, in order with the kernels:

```
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start);           // enqueued, returns immediately
kernel<<<grid, threads>>>(...);  // enqueued, returns immediately
cudaEventRecord(stop);
cudaEventSynchronize(stop);       // wait until the GPU reaches `stop`
```

```
float ms;
cudaEventElapsedTime(&ms, start, stop);
```

The program already uses this pattern to measure the total time, transfers included (`ms_total`). Complete **TODO (3)** to measure the kernel alone: the average time of `R` launches, stored in `ms_kernel`. From it, the program derives the *effective bandwidth*, that is, the bytes read and written by the kernel divided by its time.

Q2.3 Why is `cudaEventSynchronize` needed before `cudaEventElapsedTime`?

2.3 How fast is fast?

A time in milliseconds means little on its own; it becomes meaningful once compared with what the hardware allows.

Predict first.

Q2.4 How many bytes does the kernel move, and how many floating-point operations does it perform, per element? Given the peak bandwidth of your GPU, what is the shortest possible time of the kernel for $N = 8192$?

The benchmark script runs sizes from 256 to 8192 with blocks of 8^2 , 16^2 and 32^2 threads, together with the CPU version `matrix_cpu`. Run it, then plot the results:

```
srn --gres=shard:1 --cpus-per-task=4 --mem=8GB \
  scripts/bench.sh -k add -o results/<team>.csv
python3 scripts/plot.py results/<team>.csv
```

Q2.5 Fill in the table of the report. What fraction of the peak bandwidth do you reach, and from which size on? From which size is the GPU faster than the CPU, with and without the transfers? What do you conclude about sending a single addition to the GPU?

Checkpoint. For large matrices, `add` reaches a large fraction of the peak bandwidth (typically 70 to 90 %). If you are far below, check your time measurement first.

2.4 Coalescing

Reaching such a bandwidth is no accident: it depends on how the threads access memory. The threads of a block are grouped into *warps* of 32 threads with consecutive `threadIdx.x`, which execute the same instruction together. When the 32 threads of a warp read 32 consecutive floats, the hardware serves them all with a few 32-byte memory transactions: the access is *coalesced*. To see what happens otherwise, complete the kernel `add_swapped` (in the same file). It performs the same computation, but with `x` running down a *column* and `y` along a row.

Predict first.

Q2.6 In `add_swapped`, how far apart in memory are the elements accessed by two consecutive threads of a warp? Predict the slowdown, then measure it: run the benchmark again, this time with all kernels (no `-k` option), and plot the new results.

Checkpoint. `swapped` is PASSED and several times slower than `add`.

3 Mandelbrot

The matrix addition moved a lot of data for very little computation. The Mandelbrot set is the opposite: almost no data, and a lot of computation. It is the set of points c of the complex plane for which the

sequence $z_0 = c$, $z_{n+1} = z_n^2 + c$ stays bounded. For each pixel, the program iterates until $|z_n| > 2$ or until a maximum number of iterations is reached, and colours the pixel according to the number of iterations. Since every pixel is independent, the problem is *embarrassingly parallel*. Yet the work per pixel ranges from a single iteration to `maxiter`, and this will matter.

3.1 The CPU version

Start from the sequential program, and look at the image it produces:

```
./build/mandelbrot_cpu && python3 scripts/ppm2png.py mandelbrot_cpu.ppm
./build/mandelbrot_cpu -x -0.745 -y 0.11 -s 0.01 -i 1000 -o zoom.ppm
```

The options `-x` and `-y` set the centre of the view and `-s` its half-width, `-i` sets the maximum number of iterations, and `--double` switches to double precision. Read `src/mandelbrot_cpu.cpp` before moving on: the GPU version starts from the same code.

3.2 Port to CUDA

In `src/mandelbrot.cu`, begin with the kernel (**TODO (2)**), one thread per pixel, then build: the compiler refuses the code. Once you have understood why, fix it (**TODO (1)**), and complete the launch, the copy back and the timing (**TODO (3)** and **TODO (4)**).

Q3.1 What did the compiler say, and which modifiers did you add, and where?

Q3.2 What became of the two loops of the CPU version? Which block and grid sizes do you use for a 1024×768 image?

Q3.3 Compare your image with the CPU one, byte for byte:

```
cmp mandelbrot_cpu.ppm mandelbrot_gpu.ppm
```

If a few pixels differ, find out why. (Hint: look up the `nvcc` option `-fmad`.)

Checkpoint. `mandelbrot` writes the same fractal as `mandelbrot_cpu`, possibly up to a few pixels.

3.3 Performance and load balancing

Q3.4 Fill in the performance table of the report, in single and double precision, on the CPU and on the GPU. What is the speedup of the GPU over one CPU core? Look up the FP64 throughput of your GPU: why does double precision cost so much (or so little) here?

The speedup you just measured is an average over the whole image. The work, however, is far from uniform, as the three views listed in the report show: one lies inside the set, one outside, and one on its boundary.

Predict first.

Q3.5 Which of the three views is the slowest *per pixel*? In which one is the work the least evenly distributed? Measure them.

Q3.6 On the boundary view, try blocks of 32×1 , 8×8 , 16×16 and 32×32 threads (options `--bx` and `--by`). Is load balancing an issue here? Discuss it at two levels: among the 32 threads of a warp, which execute together, and among the blocks, which the GPU distributes over its SMs.

3.4 Gallery

Challenge. To finish, find the most beautiful view of the set: explore with $-x$, $-y$, $-s$ and $-i$, in a larger image ($-W$ 1920 $-H$ 1080). For a bolder entry, modify the kernel to draw a Julia set (z_0 is the pixel and c is fixed), the Burning Ship ($z_{n+1} = (|\operatorname{Re} z_n| + i|\operatorname{Im} z_n|)^2 + c$), $z^3 + c$, or to use your own palette. Convert your image to PNG, name it `<team>-<title>.png`, and hand it in with the options you used: `scripts/gallery.py` will gather all the entries into the class gallery.

A Command cheat sheet

```
cmake --build build -j                                     # rebuild after a change
srun --gres=shard:1 --cpus-per-task=4 --mem=4GB ./build/<program> [options]
sbatch scripts/run.sbatch ./build/<program> [options]      # output: lab1-<jobid>.out
srun --gres=shard:1 --cpus-per-task=4 --mem=4GB compute-sanitizer ./build/<program>
./build/matrix_add -n 4096 -b 16 -k swapped --csv         # one CSV line
scripts/bench.sh -k "add swapped" -b "16 32" -o results/<team>.csv
python3 scripts/plot.py results/<team>.csv
python3 scripts/ppm2png.py image.ppm
```

Further reading

- NVIDIA, *CUDA C++ Programming Guide*, chapter “Programming Model”. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- NVIDIA, *CUDA C++ Best Practices Guide*, sections on memory optimisations and on precision. <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>