CUDA learning

Alessio Cimma

April 10, 2025

Contents

1 Project structure

The program will contain a `Bridge` class, which will help call different functions during training. The idea is to have a comfortable envoirement to learn new things. I'll add in near future new tools to make debugging, plotting, benchamrking and comparisons easier. Each exercise will be contained in a different `self.execute_exerciseX()`. At the beginning of the program you will be asked to choose which CUDA kernel and function to load. Change it each time you want test different exercises. Possibly, each kernel will contain only one exercise, if the exercise needs more than one function they will all be contained in a single `.cu` file.

```
# Leggi il file contenente il kernel CUDA
with open(f'{kernel_name}.cu', 'r') as f:
    kernel_code = f.read()
```

Listing 1: Loading CUDA kernel (kernel.cu)

2 CUDA architecture

2.1 Hierarchy of CUDA architecture

We have a grid, which determines the number of blocks, which are a collection of threads. A grid 2x2 contains 4 blocks. If each block is 32 threads, we will have 128 threads total.

To determine the optimal amount of grid and blocks, organize the number of blocks and then derive the size of the grid using the following subsections.

2.2 How to launch a CUDA kernel

Remember that when launching the kernel, the first 2 arguments need to be passed as tuple (grid_size and block_size), even if it's just one number, to do it use the syntax `(x, )`:

```
# 1D version
self.imported_kernel((grid_size,), (block_size,), (x, y, z, n))

# 2D version
self.imported_kernel(grid_size, block_size, (x, y, z, n))
```

Listing 2: Launch CUDA kernel

2.3 Best parameters (block_size)

The following statistics should help choosing the best block-size to use in future projects, remember that you can't exceed the **MaxThreadsPerBlock** value (1024 on my laptop):

- Memory-bound kernels (loads/stores): 256-1024 $\longrightarrow$ *Hide latency via massive parallelism*
- Compute-bound kernels: 128-512 $\longrightarrow$ *More registers per thread may be needed*
- Shared memory usage per block: 128-256 $\longrightarrow$ *Avoid limiting number of resident blocks*
- Register-heavy kernels: 128 or lower $\longrightarrow$ *Prevent spilling and reduce pressure*
- Small data sizes (n $<$ 10k): 64-256 $\longrightarrow$ *Larger blocks may cause underutilization*
- Thread divergence (if-else logic): 32-64 $\longrightarrow$ *Keep warps smaller to minimize divergence waste*

2.4 Best parameters (grid_size)

The best grid size you can use is the following, it maximizes the usage of the GPU by dividing the problem equally across all blocks:

```
# 1D version
grid_size = (n + block_size - 1) // block_size

# 2D version
grid_size = (
  (W_delta + block_size[0] - 1) // block_size[0],
  (W_delta + block_size[1] - 1) // block_size[1]
```

```
8 )
```

Listing 3: Determine grid size