CUDA learning

Alessio Cimma

April 14, 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.

```python
# 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]
)
```

Listing 3: Determine grid size

3 Exercise 1

Objective: Creating a kernel that sums two 1D arrays element-wise.

What I learned:

- To access the correct index at which I am working, I need to use:

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

Listing 4: Index access

This way, I enter the correct block position multiplied by the size of the block (measured in threads) and then add the thread index of its parent block.

- I should the following command to cover the edge case where i have a potential thread that can do some work, but there's no more work to be found (due to the size of the work not being a power of two):

```
if (idx < n)
```

Listing 5: Index edge-case

4 Exercise 2

Objective: Creating a kernel that performs a convolution (morphological erosion).

What I learned:

- If the grid is 2D I can access the indices X and Y of the block and thread:

```
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

Listing 6: Index access

- Now to check the new boundaries:

```
if (i >= Size_x || j >= Size_y) return;
```

Listing 7: Index edge-case

- I can obtain the value of infinity without importing header files using:

```
float min_val = 1.0f / 0.0f;
```

Listing 8: Infinity

5 Exercise 3

Objective: Trying to speeding up the execution time of the Exercise 2 using shared memory.

Execution Time	NO shared-memory	shared-memory
	75.3s	84.7s

Table 1: Performance comparison

What I learned: No need for shared memory for data that needs to be accessed from ALL threads just once. Useful when each thread needs to access a certain value multiple time during the execution. The best case is when an entire block needs a batch of data to use multiple times, otherwise the memory overhead and transfer time makes it slower than just accessing the data from global memory.

Cooperative-loading: Interesting way of loading memory using all threads:

```
for (int idx = local_thread_idx; idx < total_tile_size; idx += threads_per_block) {
    int ki = idx / TILE_KW;
    int kj = idx % TILE_KW;

    int global_ki = tile_offset_i + ki;
    int global_kj = tile_offset_j + kj;

    float val = (global_ki < KH && global_kj < KW)
        ? ker[global_ki * KW + global_kj] : 0.0f;

    ker_tile[ki * (TILE_KW + 1) + kj] = val;
}
```

Listing 9: Index edge-case

6 Exercise 4 / 5

Objective: Building a simple UV mapping and building a very simple shader. Visualize the result in pygame in real-time.

What I learned:

- To increase performance you can use this to avoid branching:

```
1.0f * (condition) + 0.0f * (1 - condition)
```

Listing 10: Branchless-conditions

- To increase performance you can use this to upgrade memory access pattern (use only if you are sure 2 pointers don't point to the same memory):

```
float* __restrict__ array
```

Listing 11: Restrict access

- In `animation_pygame.py` you can see how to load and animate a shader based on time and mouse position.