CINECA

CUDA: kernel & memory final exercise & advanced content

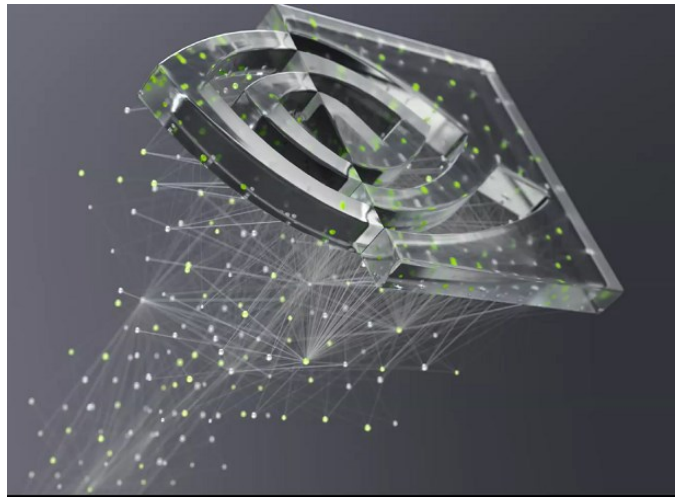
Lara Querciagrossa, Andrew Emerson, Nitin Shukla, Luca Ferraro, Sergio Orlandini

l.querciagrossa@cineca.it

July 12th, 2022

In this lecture…

- CUDA kernels and memory: final exercise
- Advanced content: multidimensional blocks and grids



CUDA kernels and memory: final exercise

Final exercise: 11_vector_add

The starting code contains a CPU vector addition function. Using what you have learned so far, accelerate the `addVectorsInto` function to run as a CUDA kernel on the GPU and to do its work in parallel. Consider the following guidelines.

1. Augment the `addVectorsInto` definition so that it is a **CUDA kernel**.
2. Choose a **working execution configuration** for `addVectorsInto`.
3. **Update memory** allocations and freeing considering that vectors need to be accessed by host and device code.
4. **Refactor the body** of `addVectorsInto` to fit in a single thread. Be sure the thread will never try to access elements outside the range of the input vectors, and take care to note if the thread needs to do work on more than one element of the input vectors.
5. Add **error handling** in locations where CUDA code might otherwise silently fail.

Advanced content: multidimensional blocks and grids

Multidimensional blocks and grids

- So far we have seen monodimensional grids and blocks:

```
someKernel<<<10, 5>>>();
```

- But both can be defined to have up to **3 dimensions**.
- No impact on performance.
- Very helpful **when dealing with multidimensional data**, for example, 2D matrices.

- CUDA's `dim3` type for both 2D and 3D grids and blocks:

```
dim3 threads_per_block(16, 16, 1);
dim3 number_of_blocks(16, 16, 1);
someKernel<<<number_of_blocks, threads_per_block>>>();
```

- CUDA variables: `gridDim.x, gridDim.y, gridDim.z, gridBlock.z,…`

Exercise: 12_2D_matrix_multiplication.cu

The starting point of this exercise contains a working host function, called `matrixMulCPU`. Your task is to build out the `matrixMulGPU` CUDA kernel. The source code will execute the matrix multiplication with both functions, and compare their answers to verify the correctness of your CUDA kernel.

Follow these guidelines.

1. Create an execution configuration whose arguments are both `dim3` values with the `x` and `y` dimensions set to greater than 1.
2. Inside the body of the kernel, establish the running thread's unique index within the grid as usual, but you should define two indices for the thread: one for the `x` axis of the grid, and one for the `y` axis of the grid.

Exercise: 13_heat_conduction.cu

In this exercise, you will accelerate an application that simulates the thermal conduction of silver in 2 dimensional space.

1. Convert the `step_kernel_mod` function to execute on the GPU.
2. Modify the main function to properly allocate data for use on CPU and GPU.

The `step_kernel_ref` function executes on the CPU and is used for error checking. Because this code involves floating point calculations, different processors, or even simply reordering operations on the same processor, can result in slightly different results. For this reason, the **error checking** code uses an error threshold, instead of looking for an exact match.

References

References

- Previous editions of this school at CINECA
- Oakridge National Laboratory's "Introduction to CUDA C++": https://www.olcf.ornl.gov/calendar/introduction-to-cuda-c/
- NVIDIA DL Institute Online Course: **main source of exercises**
- blogs.nvidia.com
- Wikipedia

THANK YOU!

Lara Querciagrossa
l.querciagrossa@cineca.it