HE-IG VD | REDS Institut Reconfigurable and Embedded Digital Systems

Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistants : Mehdi Akeddar

Session 2 – CUDA thread hierarchy

lab04 (05.12.2024)

Objectives of the laboratory

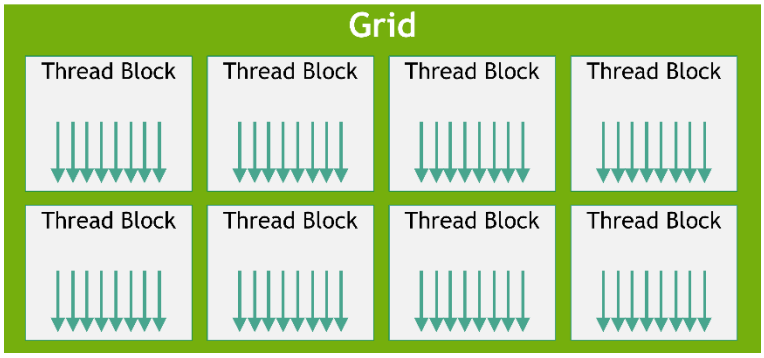
The objectives of Lab04 are to exercise the content regarding acceleration using CUDA. Lab04 is structured in four sessions. This manual refers to "Session 2" and is devoted to understanding CUDA thread hierarchy and some basic concepts.

Laboratory validation and grading

- We will grade this laboratory based on the work we ask to be delivered.
- The parts in *light blue* are what you are expected to reply in the readme file.
- We expect (at least) four different commits to the git repository, one per session, clearly indicating the session submission (for example: git commit -m "submission session 1 lab02")
- This commit should be done at the latest the night before the next lab session.
- Therefore, you have a full week to submit the lab.

Stage 1 – GPU thread hierarchy

CUDA threads are organised into blocks and blocks are organized into grids. Both blocks and grids are organized in either a one-dimension, two-dimension, or three-dimension scheme. Individual threads can be identified by their "thread id" within a block and individual blocks can be identified by their "block id" within the grid.



There is a limit to the number of "threads per block" since all threads are expected to reside on the same streaming multiprocessors core and must share the memory of that core. There is no limitation on the number of blocks in a grid and the number of blocks is usually dictated by the size of the data being processed.

Stage 2 – CUDA thread hierarchy built-in variables

The number of CUDA threads that execute one kernel is specified by each kernel call using the execution configuration syntax. Each thread executing the kernel is given a unique "thread id" that is accessible within the kernel through build-in variables. The built-in variables specify the grid and block dimensions and the block and thread indices. These built-in variables are only valid inside functions that are executed on the device. We can print the content of these variables to better understand the thread hierarchy. Printing from a device-side function requires special consideration since it interacts with the host without the user being aware.

```
# Output example
Thread {thread id}/{block size} in {block id}/{grid size}
```

*We ask you to implement a CUDA kernel that prints **all** the content on the built-in variables that specify the thread hierarchy, in hierarchy.cu and to complete the README.*

Stage 3 – CUDA Scalar multiplication and vector addition

One typical linear algebra operation is scalar multiplication and vector addition. It is defined by $\boldsymbol{z} = \alpha\boldsymbol{x} + \boldsymbol{y}$ where $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ are vectors and α is a scalar. It is very similar to the vector addition examples used in the previous lab and the slides.

Error checking is an important aspect of CUDA programming since all runtime (synchronous) functions return an error code that should be checked in case there was a problem on the device.

We ask you to implement a CUDA kernel that performs scalar multiplication including error checking for any vector size in saxpy.cu and complete the README.

Stage 4 – CUDA matrix multiplication

Matrix multiplication is another typical linear algebra operation that we have seen in this course, and it is a good problem that can be solved with CUDA. This is one fundamental reason deep learning has seen a huge increase in the last ten years. We have seen before that there are different ways to accelerate matrix multiplication. In our case, we can start with a naïve approach where each thread calculates one element on the output matrix (that is each thread calculate the dot product between a row of matrix A and a column of matrix B).

Measuring performance is also relevant for CUDA programming. It is mostly done from host code but we need to understand how to synchronize execution between the host and device. Most runtime functions are synchronous (blocking). This means that a runtime call will not begin until all previously issued CUDA calls have been completed. Kernels are not synchronous, and control is returned to the CPU immediately (it does not wait for the kernel to complete). This is sometimes hidden due to posterior runtime synchronous calls (usually data transfer from device to host) that ensure the kernel completes before the synchronous call begins. If we want to measure kernel performance from the host, we need to use an explicit synchronization barrier to block execution until all previously issued commands on the device have been completed. We can also use CUDA events to measure kernel performance.

We ask you to implement a CUDA kernel that performs matrix multiplication including execution time as a performance metric for any matrix size in gemm.cu and complete the README.
***Optional**: you can implement performance measurement using CUDA events.*

Before you leave…

Make sure you have all the requests *in light blue* of this manual, that you have implemented all the functions required and filled-in the answers in the *README* file in the repo. Then commit all files to your own “cnm23-yourname” repository.

We expect (at least) one commit for this session, clearly indicating the session submission (for example: git commit -m “submission session 2 lab04”)

This commit should be done at the latest the day before the next lab session at 23:59h.