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

Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistants : Mehdi Akeddar

Session 3 – CUDA memory hierarchy

lab04 (19.12.2024)

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 3" and is devoted to understanding CUDA memory hierarchy and related topics.

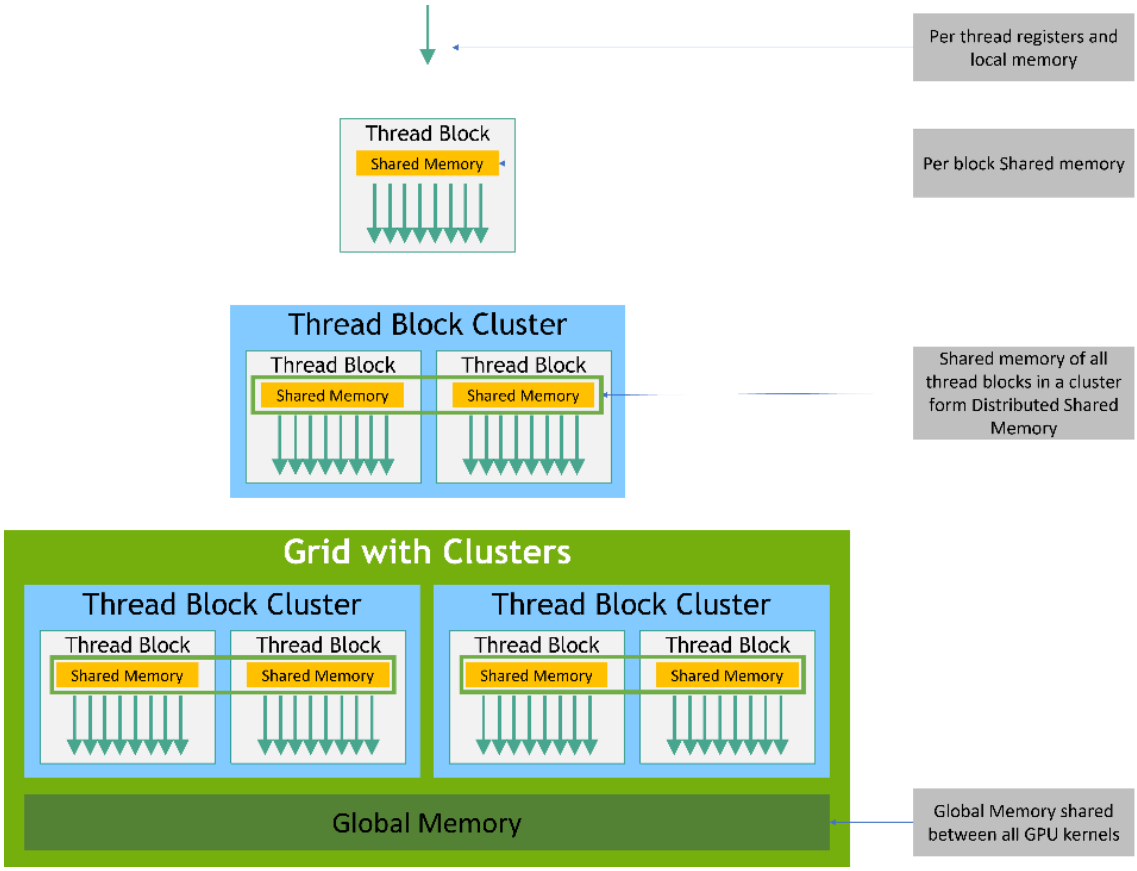
Laboratory validation and grading

- 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 – CUDA memory hierarchy

CUDA threads may access data from multiple memory spaces during their execution, each one with different scope, lifetime and caching behaviour:

- Each thread has a private local memory.
- Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.
- Thread blocks in a thread block cluster can perform read, write and atomics operations on each other's shared memory.
- All threads have access to the same global memory.



The compute capability of a CUDA device determines its general specifications and available features supported by the GPU hardware. For example, thread block clusters were introduced in compute capability 9.0.

You can read more about memory hierarchy and device memory accesses in the CUDA documentation.

Stage 2 – CUDA memory coalescing

Memory coalescing refers to combining multiple memory access into a single transaction. CUDA can combine multiple memory access to global memory. The device *coalesces* global memory reads and writes issued by the threads of a warp (32 threads) into as fewer transactions as possible.

To have global memory coalescing transactions one must follow some conditions:

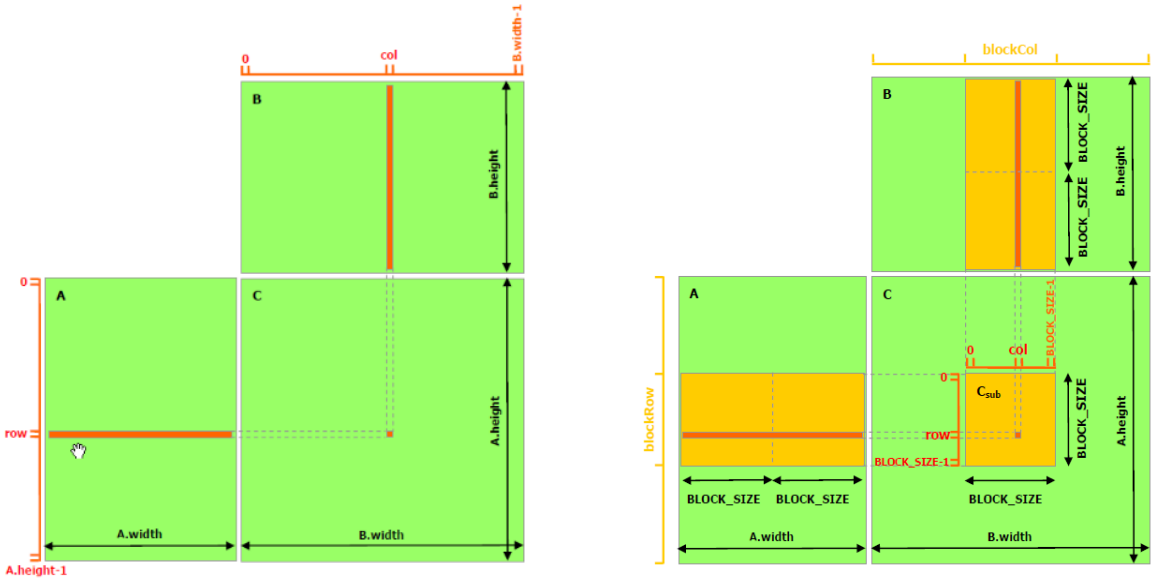
- Memory is sequential.
- Memory access is not sparse.
- Memory access should be aligned.

We ask you to complete the README and then implement the required parts (misaligned and strided memory access) in CUDA.

Stage 3 – CUDA shared memory

Shared memory is much faster than global memory. One way to optimize memory throughput is to minimize global memory access using shared memory. Shared memory is equivalent to a cache managed by the developer because we need to explicitly allocate and access it. Shared memory enables cooperation between threads in a block.

In the last laboratory we implemented matrix multiplication using global memory. Each threads reads one row of A and one column of B and computes the corresponding element of C (left figure). This means global memory access is not optimal because matrix A is read B columns times and B is read A rows times.



To use shared memory, we can use the same method we have already seen in other labs, tiling. In this case each thread block is responsible for computing one square sub-matrix of C (C_{sub}) and each thread in the block is responsible for computing one element in the sub-matrix. This square sub-matrix will have the same size as our block and because it is the product of two rectangular matrices, one with size A columns by block size and another with size B rows by block size. Then we can split these rectangular matrices into as many square matrices with size block size to fit them into shared memory and the result is the sum of the products of these square matrices. To calculate these products, we need to first load the two corresponding square matrices from global memory with one thread loading one element of each matrix and then each thread can compute one element of the product. Each thread accumulates the result of each of these products in private local memory and writes it to global memory when it is finished.

We ask you to implement a CUDA kernel that performs matrix multiplication using shared memory and complete the README. To avoid increasing the complexity assume the shared dimension of the matrices (A width and B height) to be a multiple of the block size.

***Optional**: Implement matrix multiplication with shared memory for matrices with any size.*

Stage 4 – CUDA atomic operations

Atomic functions perform a read-modify-write atomic operation on one word residing in global or shared memory without the interference of any other thread. Atomic operations are used to avoid race conditions in multithreaded applications. If two threads perform an atomic operation at the same time in the same memory address, they will be executed one after the other (serialized) and it will degrade performance.

Atomic functions might not be available depending on the compute capability of the device. For example, double precision floating point atomic add is not available on devices with compute capability lower than 6.0 but any atomic operation can be implemented using atomic compare and swap.

We ask you to implement a CUDA kernel that uses atomic operations to increment the value of an array and complete the README.

Stage 5 – cuBLAS and cuDNN

CUDA accelerated libraries is a collection of libraries, tools and technologies built on top of CUDA that delivers higher performance across multiple application domains.
In this laboratory, we use cuBLAS and cuDNN:

- **cuBLAS** is the GPU accelerated basic linear algebra (BLAS) library. BLAS is a specification of a set of routines that implement common linear algebra operations like matrix multiplication.
- **cuDNN** is the GPU accelerated library of primitives for deep neural networks. It implements standard deep learning routines such as forward and backward convolution, pooling, normalization and activation.

Stage 5.1 – cuBLAS matrix multiplication

Once more, we have the problem of matrix multiplication. General matrix multiplication (*gemm*) is one of the routines specified at the level 3 BLAS operations. You can find the cuBLAS *gemm* signature, operation description and details in the official cuBLAS documentation.

We provide a program that compares a naïve implementation of single precision *gemm* with the cuBLAS implementation for square matrices with predefined size. Then it outputs the execution time for both operations. You can compile it using *nvcc* and linking against the cuBLAS library.

```
nvcc gemmCUBLAS.cpp -lcublas -o gemmCUBLAS
```

We ask you to calculate the speed up achieved by the cuBLAS implementation compared with the naïve implementation (use several matrix sizes). Then we ask you to generalize the code for matrices with any size (not squared) and finally complete the README.

Stage 5.2 – cuDNN convolutional neural network

One of the samples provided by cuDNN implements the forward pass of a convolutional neural network trained to recognize handwritten numbers from the famous dataset mnist called *mnistCUDNN*. The sample is based on a tutorial from Caffe and the network topology is almost the same as the one proposed in the tutorial. The networks weights are also obtained and exported using Caffe.

We provide a slightly modified version of the original sample to ease the compilation.

To run the *mnistCUDNN,* you'll need to install the FreeImage library. *As you'll need an internet connection for this command, please take the ethernet cable connected to the computer next to you and put it back at the end of the session.*

```
sudo apt-get install libfreeimage3 libfreeimage-dev
```

The details to build the program are provided in the mnistCUDNN/readme.txt file.

We ask you to build and execute the provided sample and complete the README.

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 "cnm24-yourname" repository.

We expect (at least) one commit for this session, clearly indicating the session submission (for example: git commit -m "submission session 3 lab04")

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