Matrix product in CUDA

Let $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$. The product $C = A \times B$, $C \in \mathbb{R}^{m \times p}$ is defined by

$$\forall 1 \leq i \leq n, 1 \leq j \leq p, c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

1 Standard tools

1.1 Compilation with `cmake`

In this tutorial, you will be using `CMake` tool to create `Makefile` and compile automatically. To use the proposed `CMake`, you must

- Create a directory call `build` in the lab directory : `mkdir build`.
- Go to the created directory : `cd build`.
- Generate the different `Makfile` : `cmake ..`.
- Compile the code : mshmake.

1.2 Plotting with `gnuplot`

To plot the timing results, you may use `gnuplot`. `gnuplot` is an opensource application that plot data from a text file :

- First, create a file `timing.txt`.
- Open the file.
- On each line, write 2 data. In this lab, it should be the dimension of the problem and the execution time.
- Launch gnuplot : `gnuplot`.
- Plot the file : `plot "timing.txt"`.

2 Matrix multiply on CPU

To get started, we will use the `main_cpu.cxx` that implements a naive matrix multiplication on CPU. All it does is to perform for every element of the output array a scalar product between a row of A and a column of B.

- For different size of matrices, measure the performances of the matrix product. We will focus on square matrix in power of 2. You should plot a graph of the result.

3 Naive multiply on GPU

The second step is to look at `main_gpu.cu` that is a driver for matrix multiplication on GPU. The file `gemm_kernel.cuh` contains the function `gemm_naive` that performs naive multiplication on GPU.

- For different size of matrices, measure the performances of the matrix product. We will focus on square matrix in power of 2. You should plot a graph of the results.

- Try to adapt the block size to identify, for each matrix size, which one is best.
- How many global memory loads are performed ?
- How many arithmetic operations are performed ?

4 Shared memory computation

The next stage is to improve "computation-to-memory ratio". For this purpose, one may apply tiled matrix multiplication . One thread block computes one tile of matrix C. One thread in the thread block computes one element of the tile.

- Create a function that will use shared memory.
- Analyze the performances of the kernel the same way as in the previous section.
- For which tile size the performances are the best ?
- How many global memory loads are performed ?

5 Coalesced memory access

Two dimensional arrays in C/C++ are row-major. In the tiled implementation above, neighboring threads have coalesced access to matrix A, but do not have coalesced access to matrix B. In column-major languages, such as Fortran, the problem is the other way around.

- Implement CPU transposition of matrix B before offloading it to GPU memory.
- Analyze the performances of the kernel the same way as in the previous section.

6 Bank conflict

When loading the tiles of B in memory, memory operations are subject to bank conflicts. To avoid bank conflicts, one should load transposed tile of B.

- Implement bank conflict free operation when loading B in shared memory.
- Analyze the performances.