TÉCNICO LISBOA

Assignment 3B

- Tutorial -

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2024/2025

Overview

In this assignment, you will analyze a couple of parallel programs in CUDA. You will not need to develop a single line of code, but you will learn about efficient parallelization and code tuning on GPUs. Yes, we are talking about the mat mul, what else – is there any other algorithm nowadays ;)

Deadlines and Submission

NO SUBMISSIONS! This work serves as a tutorial and preparation – **NO EVALUATION**. It is intended for your practice and learning via examples and tutorials. This assignment provides a set of exercises for you to practice! You can resolve them at our lab session or at home. Surely, you can discuss your solutions and doubts with the teacher of your labs in our next lab session. Again, there is no submission for this part of the work and your solutions will not be graded!

You can consider these examples as a representative set of problems to get you prepped for the work to be developed in the class (which will be evaluated). Those assignments are not yet provided to you!

Environment Setup

For this assignment, you will need to run (the final version of) your codes on `LSD2` machines, which contain NVIDIA RTX 4070 Ti GPUs with the following characteristics.

Before compiling and executing any of the codes, ensure that you enabled the Intel oneMKL by executing the following command in your terminal window:

```
source /opt/intel/oneapi/mkl/latest/env/vars.sh
```

Part 1: Warm up! Remembering our CPU …

The starter code for this part of the assignment is located in the `/mat_mul` directory of the assignment tarball. In the `matrix.cpp` file you can find various CPU implementations of the matrix multiplication (some of them you should already know since we discussed them in the previous assignment). You are encouraged to take a close look at them!

As a reminder, we are dealing with the operation `C=AxB`, where A, B, and C are squared single-precision floating-point matrices of a size `NxN`. For curiosity, this type of operation is referred to as `sgemm` (*s* stands for single precision and *gemm* for general matrix multiply), which is formally defined as `D=alpha*A*B+beta*C`. There are three hand-tuned implementations provided in the following functions:

- `multMatrix`**`Simple`**: simple three-for-loop implementation;
- `multMatrix`**`Transposed`**: implementation with transposed B matrix;
- `multMatrixTranspose`**`Blocked`**: implementation with blocking, where the processing is conducted in the square blocks of an `SBLKxSBLK` size.

Hit `make` to compile the program and run the "`simple`" implementation for a range of matrix sizes from 32x32 to 1024x1024 by executing the following command:

```
./matrix -m simple -n 32 -N 1024
```

The program will output the matrix sizes it ran (see the line in the output denoted with N), as well as the attained GFLOPS/s (see line denoted with GF).

Repeat the same procedure to invoke the other implementations. For this, you will need to substitute the keyword `simple`, with `transpose` and `tblock`, to run transposed and blocked implementation, respectively. Fill the table below with the sizes and GFlops these implementations achieved:

Sizes (N)					
simple [GFLOPS]					
transpose [GFLOPS]					
tblock [GFLOPS]					
MKL [GFLOPS]					

To finish off the matrix multiplication optimization on CPU, we can call the best implementation provided in the Intel Math Kernel Library (MKL) by running the following command:

```
./matrix -m mkl -n 32 -N 1024
```

How do you explain the obtained results?

Some interesting questions that you should think about: What happens with the "`simple`" performance as the matrix size increases? How does the performance of the "`transpose`" version compare with the "`simple`"? Why? What about its performance scalability for different matrix sizes? How about the performance and scalability of "`tblock`" version? Finally, what do you observe for the "`mkl`" version? Can you explain the amount of GFLOPs reported? What do you think would be the magic that the Intel code developers had put in this code?

Don't forget, all the codes are run on a single-core! How would the performance change if we would run the codes on all available cores?

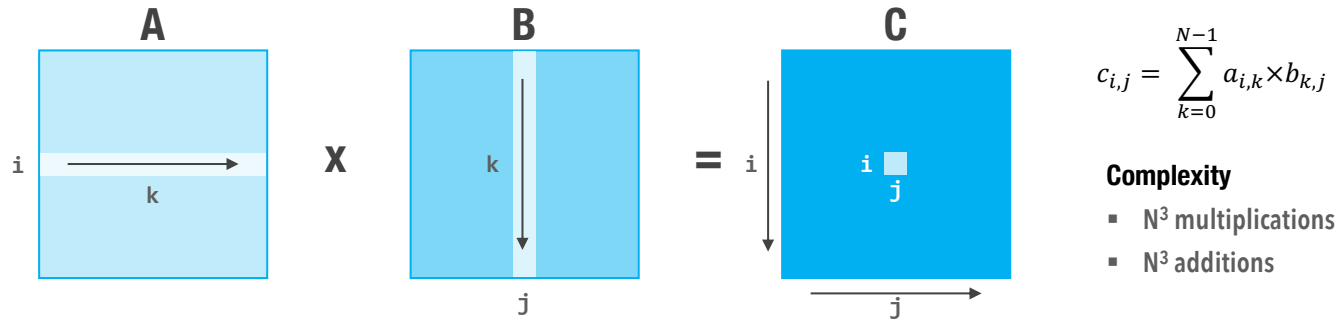
Part 2: GPU Matrix-multiplication with CUDA

The starter code for this part of the assignment is located in the (same) `/mat_mul` directory of the assignment tarball. In the `cudaMatrix.cu` file you can find four hand-tuned GPU implementations of the matrix multiplication, provided in the following CUDA kernels:

- `cudaSimpleKernel`: the `cudaMultMatrixSimple` provides the wrapper where this kernel is called at the host;
- `cudaSimpleKernelInverted`: with the `cudaMultMatrixSimpleInverted` host wrapper;
- `cudaTransposedKernel`: with the `cudaMultMatrixTransposed` host wrapper;
- `cudaBlockKernel`: with the `cudaMultMatrixBlocked` host wrapper.

You are encouraged to take a close look at them! You don't need to worry about allocating and transferring the matrices between the host and the device, that is already handled for you within the `cudaBenchMM` function.

Let us start with the **`cudaMultMatrixSimple`**, which implements a naïve version of CPU matrix multiplication depicted below.



$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j}$$

Complexity

- N^3 multiplications
- N^3 additions

Locate the implementation of this function, where you can observe the following piece of code:

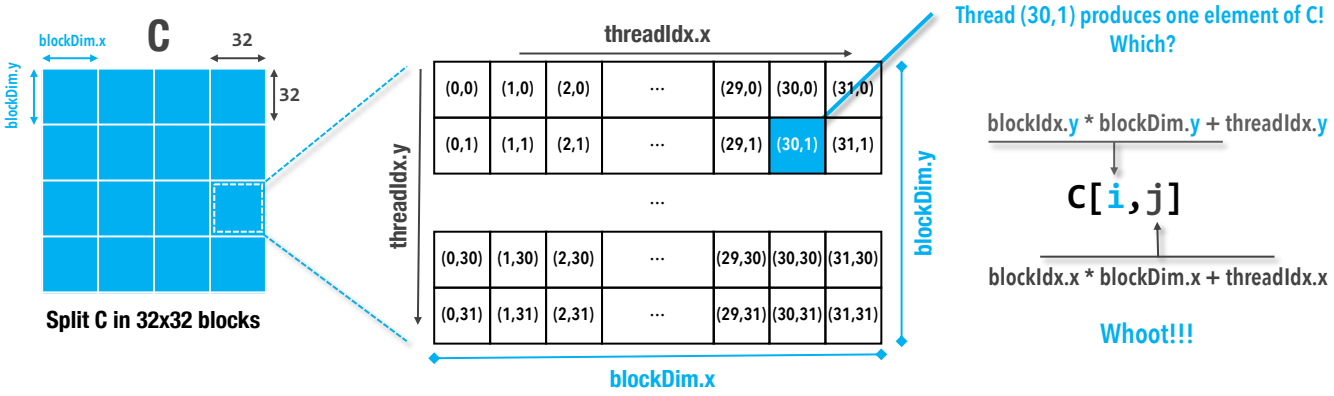
```
void multMatCUDASimple(int N, float *dA, float *dB, float *dC)
{
  dim3 threadsPerBlock(LBLK, LBLK); // LBLK=32, 32x32=1024 threads per block
  dim3 blocks( updiv(N, LBLK), updiv(N, LBLK));
  cudaSimpleKernel<<<blocks, threadsPerBlock>>>(N, dA, dB, dC);
}
```

As you know from the theoretical classes, this code will instruct the GPU to launch the `cudaSimpleKernel` with 32x32 (1024) threads per block (LBLK is fixed to 32 in our code) organized in a 2D space, within a 2D grid of (N/32 x N/32) thread blocks (N being the matrix dimension).

If you take a look at the `cudaSimpleKernel` implementation, you can notice that each "CUDA thread" is responsible for computing one element of the C matrix, i.e., `C[i,j]`. For this purpose, each thread will load one row from matrix A (i.e., row `A[i,*]`) and multiply it with a corresponding column from matrix B (i.e., column `B[*,j]`), where * denotes that all elements from 0 to k-1 are considered for this operation.

An interesting observation can be made on how the `C[i,j]` element is indexed (see code above and picture below).

```
__global__ void cudaSimpleKernel(int N, float *dA, float *dB, float *dC)
{
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  . . .
}
```



As you can notice, the CUDA built-in variables for `blockIdx`, `blockDim` and `threadIdx` refer to the **y** dimension when defining the **`i-th`** row, while the built-ins in **x** dimension are used to denote the **`j-th`** column of the C element to produce.

As we already know, we can index the elements as we like in CUDA. So, we provide an additional implementation referred to as **`cudaSimpleKernelInverted`** with the `cudaMultMatrixSimpleInverted` host wrapper. The only deference between the `cudaSimpleKernel` and `cudaSimpleKernelInverted` is in the way how the C element is indexed. In the `cudaSimpleKernelInverted` kernel, we adopted inverted indexing, where the built-in variables in the **x** dimension are used when defining the **`i-th`** row, while the built-ins in **y** dimension are used to denote the **`j-th`** column of the C element to produce (see the code). Which code version do you expect to be faster? Why?

Hit `make` to compile the program and run the "`csimple`" (cuda simple) implementation for a range of matrix sizes from 32x32 to 1024x1024 by executing the following command:

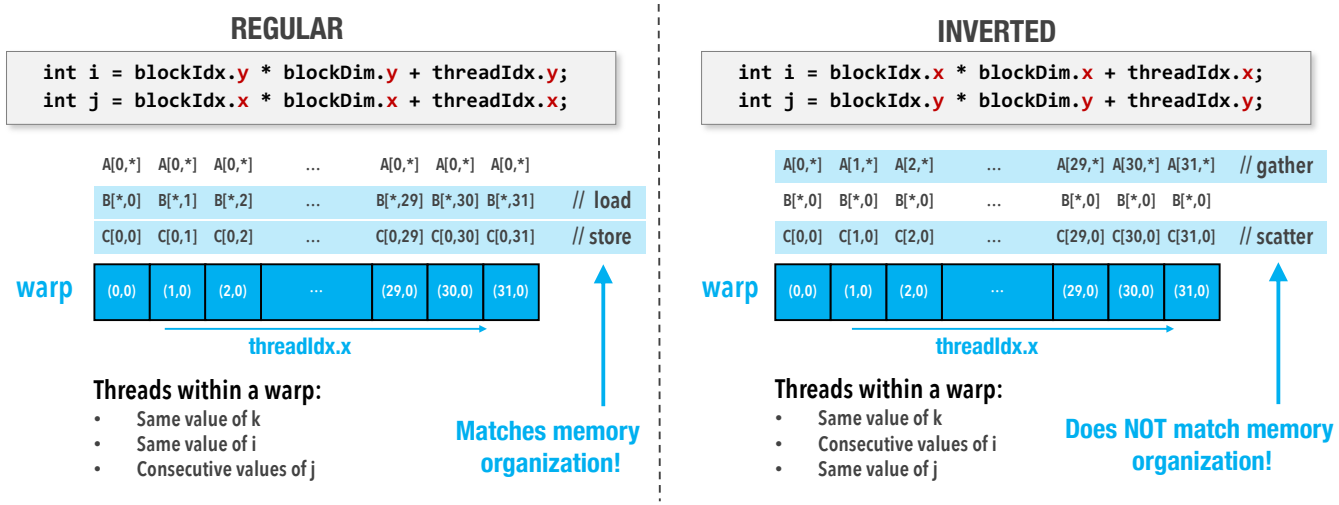
```
./matrix -m csimple -n 32 -N 1024
```

Repeat the same procedure to invoke the inverted implementation by substituting `csimple` with `csimpleinv`. The program will output the matrix sizes it ran (see the line in the output denoted with N), as well as the attained GFLOPS (see line denoted with GF). Fill the table below with the sizes and GFLOPS these implementations achieved:

Sizes (N)					
csimple [GFLOPS]					
csimpleinv [GFLOPS]					
ctranspose [GFLOPS]					
cblock [GFLOPS]					
cublas [GFLOPS					

What do you observe? Were you expecting this result? Why (or why not)? You should think about how the warps (and threads within a warp) are organized in these two implementations. What do the threads

in these warps do in the same clock cycle? Do they efficiently perform their operations? To help you discover the answer, we provide a small sketch below of how the access to A, B and C matrices are conducted with both “regular” and “inverted” indexing.



Moving onto our **third optimization**: **transposition**! If you recall, this is the trick that provided good performance improvements on the CPU, due to better cache utilization! So, let’s try to benefit from it on the GPU! The host wrapper of this code is provided in the `cudaMultMatrixTransposed` function, presented below:

```
void cudaMultMatrixTransposed(int N, float *dmatA, float *dmatB, float *dmatC)
{
    dim3 threadsPerBlock(LBLK, LBLK);
    dim3 blocks(updiv(N, LBLK), updiv(N, LBLK));
    . . .
    cudaMatTransposeKernel<<<blocks, threadsPerBlock>>>(N, dmatB, tranB);
    cudaTransposedKernel<<<blocks, threadsPerBlock>>>(N, dmatA, tranB, dmatC);
}
```

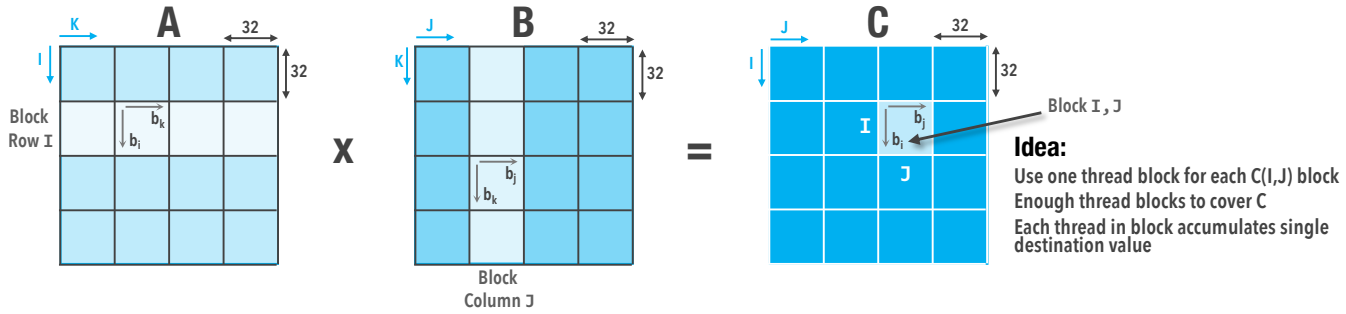
As you can see, we now invoke two different CUDA kernels: `cudaMatTransposeKernel` and `cudaTransposedKernel`. The first one is responsible for matrix B transposition, while the second computes the matrix multiplication using the transposed B matrix. You should recall that multiple kernel launches are important and very common in CUDA programming since they are the ONLY WAY to sync across different thread blocks. Basically, when the kernel returns, it is guaranteed that all threads (in all thread blocks) finished their work)!

Test this implementation by executing the following command (and fill the table above):

```
./matrix -m ctranspose -n 32 -N 1024
```

What do you observe? Does this code version provide significant speedups versus the previous two versions? Why (or why not)? Again, think about warps!!!

The **fourth hand-tuned CUDA implementation** is provided in the **`cudaBlockKernel`**, with the `cudaMultMatrixBlocked` host wrapper. This version of the code exploits the use of shared memory and matrix partitioning into blocks (similar to the CPU version), as depicted below.



In a nutshell, there are two main execution steps in this code: *1)* the threads in a thread block (i.e., C partition) first cooperatively fetch blocks of A and B in the shared memory (see `subA` and `subB` arrays), and then we need to place the `__syncthreads()` to ensure that all threads finished their loading into the shared memory arrays; then *2)* each thread reads one "small" row from `subA` and a "small" column from the `subB` array in the shared memory) to compute its own C element (using regular indexing). It is worth noting that the fetching to shared memory is done Bk times (Bk=N/LBLK), i.e., jumps in tiles (partitions), which are preceded by the `__syncthreads()` primitive to ensure that the computation on the previously fetched partitions has been completed. Test this implementation by executing the following command (and fill the table above):

```
./matrix -m cblock -n 32 -N 1024
```

What do you observe? Does this code version provide significant speedups versus the previous two versions? Why (or why not)? Again, think about warps and the amount of load/store operations that we now perform.

Finally, to finish off the matrix multiplication optimization on the GPU, we can call the best implementation provided in the NVIDIA CUBLAS library by running the following command:

```
./matrix -m cublas -n 32 -N 1024
```

What do you observe? Is this code version significantly faster than the previous versions? How does it compare with the best code version on the CPU? How many CPU cores will you need to reach the GPU performance? How many "SMs" (GPU cores, as we call them in the classes) will you need to match the CPU single-core performance?