TP3: Heterogenous Gemm

1 Goals

On modern computing architectures we have different kind of processing units, most commonly CPU (central processing unit) and GPU (graphics processing unit). They both have different strengths and weaknesses regarding computation, meaning they can overperform or underperform depending on the kind of computation we ask them to do. That is why when we want to use all our hardware computing potential we are not just only interested in using all the processing units available but we are also willing to choose wisely how to share the computation to use at best all the hardwares depending on their properties. This kind of computation mixing different processing units is called heterogenous computation.

In this TP you will propose a parallel effective algorithm mixing both GPU and CPU for the computation of a matrix-matrix product (i.e. GEMM) using the OpenMP interface. First you will be asked to implement a full CPU multi-threaded GEMM, second a full GPU version without CPU, third an heterogenous version using both at the same time the CPU and the GPU, and finally you will develop a little performance study to conclude.

2 Run the code

For the three exercises you will need to complete the functions contains in the `gemm_collection.c` file. In this file there are four versions of the GEMM : full sequential CPU, mutli-threaded CPU, full GPU, heterogenous CPU and GPU.

To compile your code you need to launch the environment by typing in a terminal `source env.sh` in the TP directory, and then type `make`. This will create three executables : `testing`, `gpu_vs_cpu`, and `gpu_with_cpu`. `testing` can be used to test one by one your implementations of every function, `gpu_vs_cpu` will create a graphic comparing GPU and CPU performances, `gpu_with_cpu` will create a graphic comparing the performances for different amount of workload on GPU of the heterogenous `gemm`.

To launch a test you can write `testing func N (%)` in a terminal. `func` as the values 1, 2, or 3 referring respectively to a call to `gemm_cpu`, `gemm_gpu`, or `gemm_gpu_cpu`. `N` is the dimension of the square matrices A and B which will be generated. `(%)` is the percentage of workload on GPU which has to be specified when `func` is set to 3.

3 Exercise 1 : Multi-threaded GEMM on CPU

We consider a matrix-matrix product of the form $A \times B = C$ where $A \in \mathbb{K}^{M \times N}$, $B \in \mathbb{K}^{N \times K}$, and $C \in \mathbb{K}^{M \times K}$. An easy approach to parallelize this operation on n CPU threads is to distribute on every thread the product of A with a sub-matrix column of B such that the i^{th} thread will compute $A \times B_i = C_i$ where $B_i \in \mathbb{K}^{N \times K/n}$ and $C_i \in \mathbb{K}^{M \times K/n}$. We illustrate this approach in Figure 1.

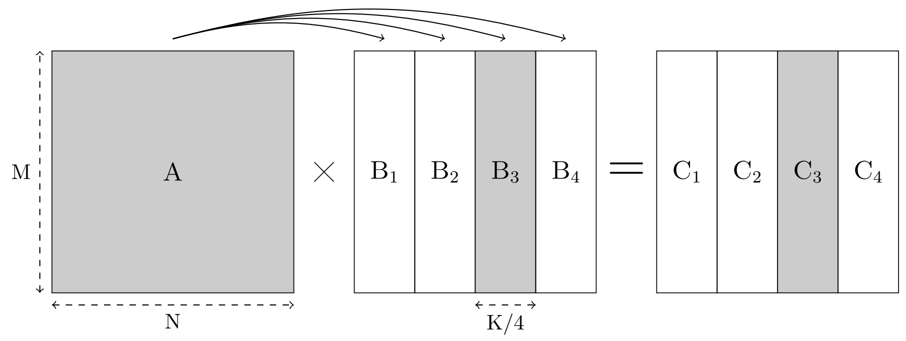

Figure 1 – CPU 4-threaded gemm

In this TP, our matrices are actually stored contiguously in an array as represented in Figure 2. This mean that if $A = (a_{i,j})$, recovering $a(i,j)$ from the pointer `A` representing the matrix A is done with `A[i*N+j]`.

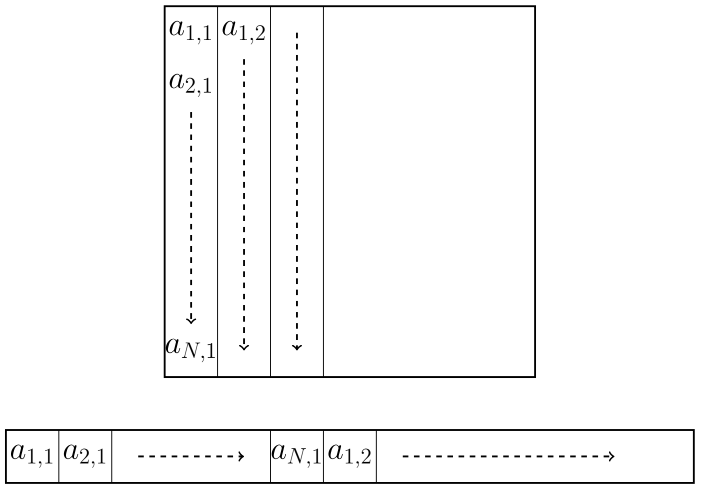

Figure 2 – Columnwise representation of a matrix

We are building our different parallel version of `gemm` on top of an implemented sequential version CBLAS for CPU and CuBLAS for GPU. The interface is as follow `gemm(Order, TransA, TransB, M, N, K,` α`, A, lda, B, ldb,` β`, C, ldc)` and it computes $C = \alpha AB + \beta C$. You will not need to change the arguments `Order`, `TransA`, `TransB`, `lda`, `ldb`, `ldc`, α and β. `A`, `B` and `C` are pointers pointing to the first component of the arrays, then equivalently we can write `&A[0]`, `&B[0]`, and `&C[0]`.

Your work : Adapt the function `gemm_cpu` in the `gemm_collection.c` file to run on multiple thread (ideally the number of available thread on your computer). Test your code with the `testing` executable.

4 Exercise 2 : GPU offload

CPU and GPU do not share memory, then this is essential to allocate and migrate the data to the GPU before being able to compute with this data using CuBLAS functions for example. Inversly, the result of a computation on GPU is stored on GPU and is not automatically accessible by the CPU. The data exchanges between CPU and GPU is done in OpenMP using the `data map` interface.

With the GPU's models present in the student computer park of ENSEEIHT, we can not implement directly an efficient GEMM fully in OpenMP, then we are forced to use the CuBLAS functions which embeds an optimized version of the GEMM.

Your work : Adapt the function `gemm_gpu` in the `gemm_collection.c` file to run the matrix-matrix product on GPU. Test your code with the `testing` executable and then compare the performances with `gemm_cpu` with the executable `gpu_vs_cpu`.

5 Exercise 3 : Heterogenous GEMM

A simple way of sharing the workload of the GEMM between the GPU and the CPU is to separate the matrix B into two matrix column. A certain percentage of the matrix B is given to the GPU and the rest to the CPU in multi-thread. This strategy is represented by Figure 3 where the product $A \times B_1 = C_1$ is done on GPU, and the products $A \times B_2 = C_2$, $A \times B_3 = C_3$, and $A \times B_4 = C_4$ are done respectively on the remaining threads of the CPU.

The offload of the GEMM on the GPU will block one thread on the CPU, then if the CPU has 4 available threads there will be only three threads remaining for computing the CPU part of the GEMM.

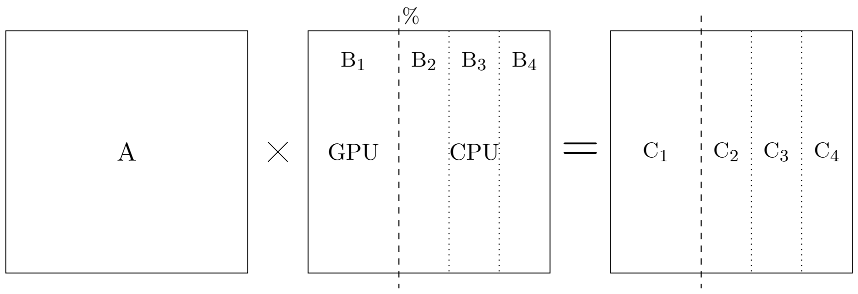

Figure 3 – Heterogenous GEMM

Your work : Adapt the function `gemm_gpu_cpu` in the `gemm_collection.c` file to run the matrix-matrix product on both CPU and GPU. Test your code with the `testing` executable (do not forget to add the `(%)` argument) and then compare the performances for different percentages of offload with the executable `gpu_with_cpu`.