M2-BIG DATA

GPGPU - Chapter 4

Exercice 1

Réalisé par:

Gaby Maroun

Encadré par:

Dr. Etancelin JM

December 13, 2020

Objectives

Implement a basic dense matrix multiplication kernel. This is the first version of several optimizations to follow (see next exercises). The program computes :

$$C = AB$$

where A, B and C are general rectangular matrices.

Instructions

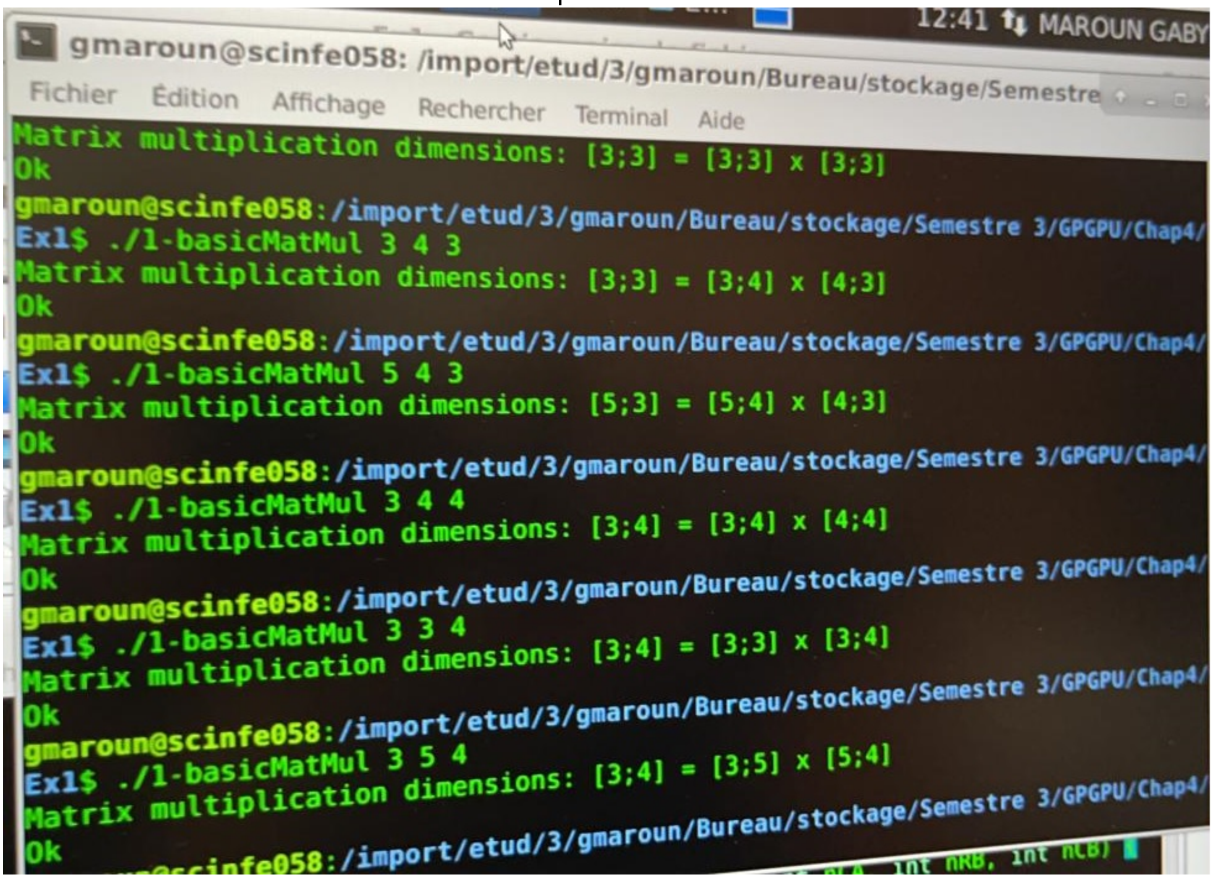
Edit the given code to perform the following :

```
#include <stdio.h>
#include <cuda.h>
#include <time.h>

#include "matmul_utils.hpp"


// Cuda kernel
__global__ void dgemm(float *A, float *B, float *C,
                        int numARows, int numAColumns, int numBRows, int numBColumns) {
  // @TODO@ : Complete here the kernel code

      // Calculate the row index of the C element and A
    int Row= blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate the column index of C and B
    int Col= blockIdx.x * blockDim.x + threadIdx.x;

    if ( (Row< numARows ) && (Col < numBColumns )) {
      float Cvalue =0;
      // each thread computes one element of the block sub-matrix
      for(int k=0; k< numAColumns ; k++) {
         Cvalue += A[Row*numAColumns +k]* B[k*numBColumns +Col];
       }
     C[Row*numBColumns + Col ] = Cvalue;
    }
}

int main(int argc, char **argv)
{
  if(argc!=4) {printf("Usage : %s [nb of rows for A] [nb of cols for A] [nb of cols for B]\n", argv[0]);exit(2);}
  //initilize a pseudo-random number generator
  srand(time(0));

```

```
 int dimension = 32;
 int numARows, numAColumns,numBRows, numBColumns,numCRows, numCColumns;
 // Read given dimensions
 numARows = atoi(argv[1]);
 numAColumns = atoi(argv[2]);
 numBColumns  = atoi(argv[3]);
 // Compute the remaining dimensions for given ones
 //@TODO@
 numBRows = numAColumns;
 //@TODO@
 numCRows =  numARows;
 //@TODO@
 numCColumns = numBColumns ;

 printf("Matrix multiplication dimensions: [%d;%d] = [%d;%d] x [%d;%d]\n",
        numCRows, numCColumns, numARows, numAColumns, numBRows, numBColumns);
 // host pointers
 float *host_a, *host_b, *host_c;
 // Device pointers
 float *dev_a, *dev_b, *dev_c;

 // Allocations on host
 host_a = (float *)calloc(numARows*numAColumns, sizeof(float));
 host_b = (float *)calloc(numBRows*numBColumns, sizeof(float));
 host_c = (float *)calloc(numCRows*numCColumns, sizeof(float));

 // Initialize vectors
 init(host_a,host_b,numARows, numAColumns, numBRows, numBColumns);

 // Allocations on device
 // @TODO@ : complete device allocations
  cudaMalloc(&dev_a, numARows*numAColumns *sizeof(float));
  cudaMalloc(&dev_b, numBRows*numBColumns *sizeof(float));
  cudaMalloc(&dev_c, numCRows*numCColumns *sizeof(float));

 // Copy from host to device
 // @TODO@ : complete copy from host to device
  cudaMemcpy(dev_a,host_a, numARows*numAColumns*sizeof(float),
   cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, host_b, numBRows*numBColumns*sizeof(float),
   cudaMemcpyHostToDevice);


 // Invoke kernel
 // @TODO@ : complete compute grid and block dim
  dim3 DimGrid((numARows-1)/dimension+1 , (numBColumns-1)/dimension+1,1);
  dim3 DimBlock (dimension,dimension,1);

 // Initialize C device data
 cudaMemset(dev_c, 0, numARows * numBColumns * sizeof(float));

```

```
  // Call the kernel
  // @TODO@ : complete to call the kernel
   dgemm<<< DimGrid, DimBlock >>>(dev_a, dev_b, dev_c, numARows, numAColumns,
    numBRows, numBColumns);

  // Copy result from device to host
  // @TODO@ : complete copy from device to host
  cudaMemcpy(host_c, dev_c, numCRows*numCColumns*sizeof(float),
    cudaMemcpyDeviceToHost);

  // Check result
  check(host_a,host_b,host_c,numARows, numAColumns, numBRows, numBColumns);

  // Free device memory
  // @TODO@ : complete to deallocate memory
    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
    cudaFree(host_a); cudaFree(host_b); cudaFree(host_c);


  return 0;
}
```

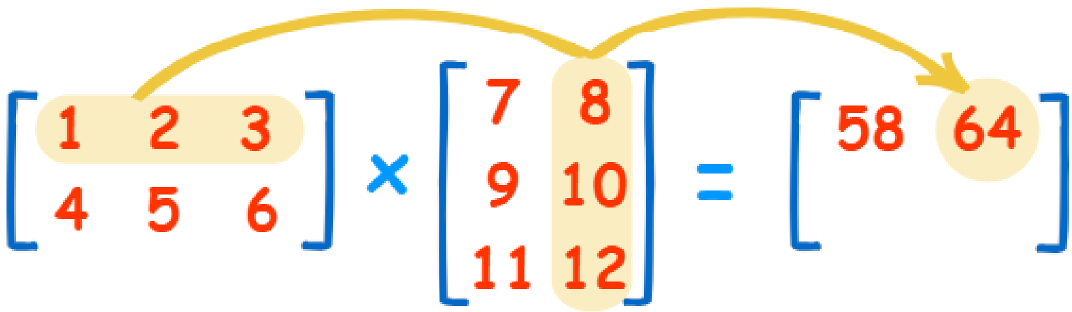
I went on and tried some different examples and the results came as follow:



Questions

1. **Before coding : What are the relations between the three matrices dimensions to have a well defined multiplication ?**

   To multiply a matrix by another matrix we need to do the "dot product" of rows and columns. For example :



   We apply the dot product on our matrix A of 2 rows and 3 columns and B of 3 rows and 2 columns and insert the resulted number in the matrix C

2. **. How many floating operations are being performed in your matrix multiply kernel ? EXPLAIN.**
   There are $nARows*nBCols(2*nACols)$ *floating operations being performed. By having to read each row in the matrix A and each column in the matrix B which makes the* $nARows*nBCols$ *then multiplying the rows of A with the cols of B before adding it to the Cvalue , which the kernel do it* $2*nACols$ *times*

3. **How many global memory reads are being performed by your kernel ? EXPLAIN.**
   There are $nARows*nBCols(2*nACols)$ *reads from A and B that are being performed by the kernel as explained in quest 2*

4. **How many global memory writes are being performed by your kernel ? EXPLAIN.**
   There are $nARows*nBCols$ *writes in C that are being performed by the kernel represented by the number of times the CValue was written into the C matrix.*

5. **Compute the arithmetic intensity of your kernel. The arithmetic intensity is a FLOP/Byte number standing for the number of floating point operations performed per byte of global memory accessed.**

$$\frac{nARows*nBCols(2*nACols)}{(nARows*nBCols(2*nACols)+nARows*nBCols)}$$

$$=> \frac{nARows*nBCols(2*nACols)}{nARows*nBCols((2*nACols)+1)} => \frac{2*nACols}{2*nACols+1}(FLOP/Byte)$$

La fin.