# Competitive and Multithreaded Programming

## Lab4. Degree of Objects Similarity with CUDA

TYTGAT Karel

## 1. Goal

The goal of this laboratory is to implement an implementation of matrix similarity using CUDA (Compute Unified Device Architecture).

## 2. Definition

CUDA is a technology that allows to use the GPU to compute simple tasks instead of computing them on the CPU. Using CUDA is only interesting if the task sent to the GPU can be parallelized.
The matrix similarity represents the distance between each combination of points from the original matrix. the formula to compute the distance matrix D is:

$$D_{i,j} = \sum_{k=0}^{n} (a_{i,k} - a_{j,k})^2$$

With $a$ the origin matrix and $n$ the size of the matrix

## 3. Theory and implementation

To implement this algorithm, we use table data structure instead of matrices. It is a variable that will define the width of the matrix.

**Implementation of the main function :**

```
int main() {
 //define memory size
 int numBytes = N * N * sizeof(float);

 float h_A[N * BLOCK_SIZE];
 float h_D[N * N];

 /*init matrix*/
 int i = 0;
 for (i = 0; i < N * BLOCK_SIZE; i++) {
  h_A[i] = (float) i + 1;
 }
 for (i = 0; i < N * N; i++) {
  h_D[i] = 0.0;
 }

 //assign variable for device
 float * d_A;
 float * d_D;

 // allocate device memory
 cudaMalloc((void ** ) & d_A, numBytes);
 cudaMalloc((void ** ) & d_D, numBytes);

 // set kernel launch configuration
 dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
 dim3 blocks(N / BLOCK_SIZE, N / BLOCK_SIZE);

 //copy data from host to device
 cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

 clock_t begin = clock();
 //kernel launch
 matMult << < blocks, threads >> > (d_A, d_D, N);
 clock_t end = clock();
 double time_spent = (double)(end - begin);
 printf("%d", time_spent);
 //copy data from device to host
 cudaMemcpy(h_D, d_D, numBytes, cudaMemcpyDeviceToHost);

 //memory free
 cudaFree(d_A);
 cudaFree(d_D);

 return 0;
}
```

The main matrix A is initialized. Once it is initialized, the program will allocate the right quantity of memory on the GPU, to be able to transfer this table from the CPU to the GPU with the *cudaMemcpy* function. This allocation of memory allocates the space for the origin matrix, but also for the output matrix where the final result will be written. The program will also define the quantity of blocks and threads to use on the GPU for the computation to paralyze the different tasks. Once the table is copied on the GPU, the program can call the kernel function with the copied matrix and the output matrix.

**CUDA implementation of matrix distances computation :**

```
__global__ void matMult(float * a, float * d, int n) {
 int bx = blockIdx.x;
 int by = blockIdx.y;
 int tx = threadIdx.x; // y index
 int ty = threadIdx.y;

 int ia = n * BLOCK_SIZE * by + n * ty; // index A
 int ib = BLOCK_SIZE * bx + tx; //index B
 int ic = ia + ib;

 int indexLst = ic;
 int i = indexLst / n;
 int j = indexLst % n;

 float sum = 0.0 f;
 __shared__ float ai[BLOCK_SIZE][BLOCK_SIZE + 1];
 __shared__ float aj[BLOCK_SIZE][BLOCK_SIZE + 1];

 for (int k = 0; k < BLOCK_SIZE; k++) {
  ai[i][k] = a[k + i * BLOCK_SIZE];
  aj[j][k] = a[k + j * BLOCK_SIZE];
 }
 __syncthreads();
 for (int l = 0; l < BLOCK_SIZE; l++) {
  sum += (ai[i][l] - aj[j][l]) * (ai[i][l] - aj[j][l]);
 }
 __syncthreads();
 d[indexLst] = sum;
}
```

In the kernel function, each iteration of the function will compute one value inside the final matrix. Two shared matrices are created to store the data that are needed to compute the final matrix. The *__syncthreads()* function will synchronize all  the threads to compute the final sum. Once the final sum computed, all the threads will wait again for all the threads to finish adding their value to the final sum. Then each thread will write their value into the final table.

Once the final table is written on the GPU, the final data will be copied into the CPU memory.

## 4. Results

The use of tables instead of matrix in the data structure can make the program really hard to understand. The different uses and significations of the block indexes and the thread indexes that have multiple dimensions make the execution hard to follow. Also, the complexity of the architecture of CUDA makes it really interesting. It is crucial to understand all the principles of this architecture to be sure to be able to use it efficiently on any hardware. However, a poor use of this architecture can make a program slower and even more complex. CUDA is a powerful tool, but need to be carefully handled to have proper and efficient algorithms.