# Lab 6: Getting started with GPU & CUDA

authors: Arias Rodrigo, Burca Horia
date: 12/11/2018

## Task 6.1

*A program that performs a matrix product (A * B = C) parallelizing the computation with CUDA*

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define N 256
#define BLOCK_SIZE_DIM 16
#define MAX_CELL 1000

#define err(format, ...) do { fprintf(stderr, format, ##__VA_ARGS__); exit(1); }
while(0)

inline void checkCuda(cudaError_t e)
{
    if (e != cudaSuccess)
    {
        err("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
    }
}

__global__ void matrixProduct(double *matrix_a, double *matrix_b, double *matrix_c,
int width)
{
    int sum = 0;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    int col = threadIdx.x + blockDim.x * blockIdx.x;

    if (col < width && row < width)
    {
        for (int k=0; k<width; k++)
        {
            sum += matrix_a[row * width + k] * matrix_b[k * width + col];
        }
        matrix_c[row * width + col] = sum;
    }
}


void initializeMatrices(double matrix_a[N][N], double matrix_b[N][N])
{
    srand(time(NULL));
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
        {
            // Perform a module here to ensure that the multiplication doesn't
```

```
overflow double
            matrix_a[i][j] = rand() % MAX_CELL;
            matrix_b[i][j] = rand() % MAX_CELL;
        }
    }
}


void showResults(double matrix_a[N][N], double matrix_b[N][N], double
matrix_c[N][N])
{
    printf("***** MATRIX A ***** \n");
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
        {
            (j % N == N-1) ? printf("%.0f \n", matrix_a[i][j]) : printf("%.0f,",
matrix_a[i][j]);
        }
    }
    printf("***** MATRIX B ***** \n");
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
        {
            (j % N == N-1) ? printf("%.0f \n", matrix_b[i][j]) : printf("%.0f,",
matrix_b[i][j]);
        }
    }
    printf("***** MATRIX C ***** \n");
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
        {
            (j % N == N-1) ? printf("%.0f \n", matrix_c[i][j]) : printf("%.0f,",
matrix_c[i][j]);
        }
    }
}

int main(int argc, char *argv[])
{
    double h_a[N][N], h_b[N][N], h_c[N][N];
    double *d_a, *d_b, *d_c;
    int size = N * N * sizeof(double);

    initializeMatrices(h_a, h_b);

    // Allocate memory in the device
    checkCuda(cudaMalloc((void **) &d_a, size));
    checkCuda(cudaMalloc((void **) &d_b, size));
    checkCuda(cudaMalloc((void **) &d_c, size));
    // Copy the information in the device
    checkCuda(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice));

    // CUDA threads structure definition
    dim3 dimGrid((N + BLOCK_SIZE_DIM -1) / BLOCK_SIZE_DIM,
```

```
            (N + BLOCK_SIZE_DIM -1) / BLOCK_SIZE_DIM);
    dim3 dimBlock(BLOCK_SIZE_DIM, BLOCK_SIZE_DIM);
    matrixProduct<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, N);

    checkCuda(cudaDeviceSynchronize());
    checkCuda(cudaGetLastError());
    checkCuda(cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost));
    checkCuda(cudaFree(d_a));
    checkCuda(cudaFree(d_b));
    checkCuda(cudaFree(d_c));
    showResults(h_a, h_b, h_c);
    cudaDeviceReset();

    return 0;
}
```

# Task 6.2

*Timing the Kernel*

With the `nvprof` utility, a detailed log can be used to time the `dot` program.

```
% nvprof ./dot
==140328== NVPROF is profiling process 140328, command: ./dot
==140328== Profiling application: ./dot
==140328== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   68.74%  45.186us         1  45.186us  45.186us  45.186us
matrixProduct(double*, double*, double*, int)
                  25.95%  17.056us         2  8.5280us  8.5120us  8.5440us  [CUDA memcpy
HtoD]
                   5.31%  3.4880us         1  3.4880us  3.4880us  3.4880us  [CUDA memcpy
DtoH]
      API calls:   83.15%  462.93ms         3  154.31ms  5.7710us  462.92ms
cudaMalloc
                  15.06%  83.830ms         1  83.830ms  83.830ms  83.830ms
cudaDeviceReset
                   1.26%  7.0001ms       376  18.617us     391ns  726.01us
cuDeviceGetAttribute
                   0.33%  1.8094ms         4  452.35us  449.94us  455.77us
cuDeviceTotalMem
                   0.10%  574.59us         4  143.65us  140.49us  146.74us
cuDeviceGetName
                   0.06%  360.29us         3  120.10us  12.913us  330.92us  cudaFree
                   0.02%  98.145us         3  32.715us  26.097us  36.813us  cudaMemcpy
                   0.01%  49.897us         1  49.897us  49.897us  49.897us
cudaDeviceSynchronize
                   0.01%  46.403us         1  46.403us  46.403us  46.403us  cudaLaunch
                   0.00%  4.8940us         3  1.6310us     598ns  3.6600us
cuDeviceGetCount
                   0.00%  4.5690us         8     571ns     469ns     864ns  cuDeviceGet
                   0.00%  2.6040us         4     651ns     215ns  1.7850us
cudaSetupArgument
                   0.00%     750ns         1     750ns     750ns     750ns
cudaConfigureCall
```

```
            0.00%     404ns         1     404ns     404ns     404ns
cudaGetLastError
```

We can see in the "GPU activities" section, how long the computation took (45.186us) and how long the transfer time was. From the host to device, we sent 2 matrices, and it took 17.056us. While from the device to the host, only the result matrix, took 3.4880us.

We also see that the CUDA directive with more time was cudaMalloc, with a long delay of 462.93ms.