TP - Introduction to CUDA Programming

Oguz Kaya
oguz.kaya@universite-paris-saclay.fr

To compile the program `program.cu` with nvcc (CUDA compiler) and generate the executable `program`, type the following command in the terminal:

```
nvcc -O2 -std=c++11 program.cu -o program
```

To connect to the cluster, type

```
ssh psqdcs_??@chome.metz.supelec.fr
```

by replacing ?? with your account number (1-40) then typing your password. Next, to allocate a GPU machine, type

```
srun --reservation=M2QDCS_GPUPROG1 --time=02:00:00 --pty /bin/bash
```

to allocate a reserved machine during the lab sessions. If you would like to use a GPU machine outside the lab sessions, type

```
srun -N 1 -p gpu_inter --time=02:00:00 --pty /bin/bash
```

to allocate a machine. You can find the cluster documentation at

`https://dce.pages.centralesupelec.fr/01_cluster_overview/`

and setup for access using Visual Code over

`https://dce.pages.centralesupelec.fr/03_connection/#using-visual-studio-code`

Part 1

Hello World using OpenMP

Ex. 1

a) Write a program `hello-cuda.cu` (skeleton code given) that launches the provided `cudaHello` CUDA kernel (or function).

- In the CUDA kernel, print the block id and thread id of each thread, as well as the total number of blocks and threads/block.
- Run this kernel using 64 threads. Try different number of blocks for all powers of 2 (1, 2, 4, 8, 16, 32, 64) and set the number of threads proportionally (64, 32, 16, 8, 4, 2, 1).

Part 2

CPU/GPU memory transfer using CUDA

In GPU programming model, the GPU device is used as an *accelerator* for the CPU that executes the main code and holds the data for the computation. Whenever a task for GPU is encountered, the CPU first transfers the data to the GPU, then executes the GPU kernel, finally transfers the results back to the CPU's main memory. The goal of this exercise is to perform such transfers for staticly (as `float A[1000]` for instance) and dynamically allocated arrays (through `malloc()/new()` or `std::vector` for instance).

In the given skeleton codes `cuda-copy-static.cu` and `cuda-copy-dynamic.cu`, the goal is to copy the CPU array `A[N]` into the GPU array `dA[N]` first, then copy the GPU array `dA[N]` into the CPU array `B[N]`. You will need to allocate the GPU array `dA[N]` statically or dynamically in these two codes, respectively.

Ex. 2

a) Allocate a static/dynamic array `dA` of size `N` on the GPU. In the static case, you will do this by adding the keyword `__device__` in front of the array declaration, whereas in the dynamic case you will need to call `cudaMalloc()`.

b) Copy the CPU array `A` into the GPU array `dA` using the corresponding CUDA memcopy function (remember, there are two memcopy functions, one for the static, other for the dynamic case).

c) Copy the GPU array `dA` back to the CPU array `B` using the corresponding CUDA memcopy function.

Part 3

Writing a GPU-GPU memcpy kernel

In this exercise, you will write your own memcpy kernel to copy a GPU array into another GPU array. You will use this to perform the memory transfer `A[N]` -> `dA[N]` -> `dB[N]` -> `B[N]`, where `A[N], B[N]` are CPU arrays and `dA[N], dB[N]` are their GPU counterparts. You should use the given skeleton code `cuda-copy-kernel.cu` for this exercise.

Ex. 3

a) First, complete the provided kernels `cudaCopyByBlocks` and `cudaCopyByBlocksThreads`.

b) Then, perform the necessary memory allocations for `dA[N]` and `dB[N]`, then do the memory transfers using `cudaCopyByBlocks` kernel, as demanded in the skeleton code's comments.

c) Finally, do the same but using the `cudaCopyByBlocksThreads` kernel this time.

Part 4

CUDA saxpy kernel

In this exercise, you will write a saxpy BLAS kernel that performs thte operation $y = ax + y$ for vectors x, y of size N and the scalar a. You will write multiple kernels that perform this operation:

Ex. 4

a) First, a kernel that only launches blocks, and one thread per block, each block working on one vector element

b) Another kernel that uses a certain number of threads (multiple of 32) per block, each thread working on one vector element

c) Finally, a kernel that uses a certain number of threads per block, each thread working on K elements of the vector.

In doing these operations, you should also perform necessary memory copies at appropriate places as indicated in the provided skeleton code **saxpy.cu**.