

## Lab Guide 9

### GPU Programming with CUDA

#### Objective:

- get acquainted with CUDA programming by implementing a 1D stencil
- improve input data locality on GPU by using shared memory

#### Introduction

This lab session aims to introduce the basic concepts of programming GPU devices using CUDA through the parallelisation of a one-dimensional stencil.

Copy the `/share/cpar/PL09_Codigo` folder to your home directory in the SeARCH cluster.

Setup the CUDA environment with a compatible GNU compiler using the following commands on the cluster:

```
module load gcc/5.3.0
module load cuda/11.3.1
```

Compile the program in the cluster frontend using the provided `Makefile`, by executing the `make` command.


Use the `sbatch run.sh` command to submit a job to the cluster to run the application.




Exercise 1 should be done in the lab session, while the optional exercise 2 should be completed afterwards.

#### Exercise 1 – 1D stencil

Consider the following pseudo-code of a 1D stencil:

```
for all Ei in inVector do
    for all Xi in radius R of Ei do
        outVector[Ei] += Xi;
    end for
end for
```



- a) The `launchStencilKernel` function in the provided code is responsible for (i) allocating space in the GPU memory for the inputs and outputs, (ii) copying the inputs from the host memory (CPU) to the device (GPU), (iii) launching the stencil kernel, and (iv) copying the outputs from the device memory to the host. Fill the missing code for the correct implementation of steps (i) to (iv). 
- b) Complete the `stencilKernel` function to implement the 1D stencil described above. The stencil CUDA code should calculate the sum of 4 neighbours (2 on each side) for a single position of the `outVector`. This kernel is automatically replicated through the CUDA threads launched in `launchStencilKernel`, where it is already guaranteed that the number of threads is equal to the size of the vectors. Run the code and identify how much time is spent on memory transfers and on the kernel execution. 
- c) The `launchStencilKernel` function launches one CUDA thread per element of `in/outVector` to process, which means that the increasing the size of `in/outVector` will cause more threads to be spawned. Increase the value of `NUM_BLOCKS`, in the `stencil.cu` file, to 256, which in turn doubles the size of the input, recompile the code and submit a new job for execution. Repeat the process for 512. How does the time spent on memory transfers and kernel execution vary with the increase in input size? 

#### Exercise 2 (optional) – Sharing memory among CUDA threads

Consider the implementation of `stencilKernel` developed in 1.b). Extend the code so that all input data required for the CUDA threads of a given block is stored in shared memory. Pay close attention to the comments on the code that may hint to the steps required for this implementation.