Lab 2 - Advanced CUDA

Advanced Architectures

University of Minho

The Lab 2 focus on the development of efficient CUDA code by covering the programming principles that have a relevant impact on performance. Use a cluster node with a NVidia GPU (by specifying the keyword `tesla` in the node characteristics when submitting a job) and do not submit interactive jobs (you should use something like `qsub -qmei -lnodes=1:ppn=??:k20,walltime=60:00 ./your_script.sh`). Compile the code in the frontend.

This lab tutorial includes a homework assignment (HW 2.1), two exercises to be solved during the lab class (Lab 2.x), and an extra exercise to be solved after the session (Ext 2.1).

Several algorithms will be implemented for both multi-core CPUs and CUDA GPUs, to assess the performance benefits of the GPUs, with different problem sizes. When measuring execution times select the best of three measurements.

To load the compiler in the environment use the one of the following commands (do not forget to also use them in `your_script.sh`):

GNU Compiler: `module load gcc/4.9.3`.

CUDA Environment: `module load cuda/7.5`.

You can use other GNU/CUDA combinations at your own risk.

Study the basic CUDA example on the *basic_kernel.cu* file, which contains a simple vector addition kernel, as well as the attached slides to help solve the proposed exercises. Do not forget to read this file header! Study the provided slides before doing the homework exercise, as they provide some insight in how to code for CUDA GPUs. A basic fully functioning vector addition CUDA code is provided as example.

2.1 Shared Memory

Goals: to develop skills in accessing shared data and thread synchronisation.

HW 2.1 Consider the one dimensional stencil algorithm that operates on vectors. Using the CUDA skeleton provided in the attached file develop an efficient OpenMP implementation to run on a 4-core Xeon device. Measure and record the best execution time on a spreadsheet, similarly to Lab 1. Complement this skeleton with the GPU code (kernel) to perform the same task. Remember that the CUDA kernel is coded in such a way that it should be executed by a single thread, using its *id*. The CUDA rutime will assign the kernel to a set of threads when it is called, so no explicit parallelism must be present in the code.

Algorithm 1 Pseudocode for a 1D stencil.

```
for all E in Vector do
    for all Xi in radius R of E do
        OutVector[E] += Xi;
    end for
end for
```

Lab 2.1 Run the GPU code and measure and record the best execution time. Compare the CPU-GPU performance. Listen to the suggestions related to the use of the GPU shared memory to improve performance. Implement the suggested optimisations and measure the performance gains.

2.2 Efficient Access to Data

Goals: to comprehend the coalesced memory access concept and develop skills on data reuse.

Lab 2.2 Consider the one dimensional stencil code from the previous exercise. Create two kernels based on the current implementation, with different ways to access elements on the input vector: specifying **(i)** an offset (i.e., with an offset of 5, thread t_0 will start on the fifth element of the vector and process them sequentially from there on), or **(ii)** a stride (i.e., with a stride of 2, thread t_0 will access the vector at position 0, 2, 4, ...). Handle the memory transfers independently for each different kernel.

Execute the kernels multiple times with different offsets and strides and assess their execution times. How do you explain the results? Listen to the suggestions to avoid excessive data transfers between host and device when calling multiple kernels with read-only inputs. Implement the suggestions and measure their impact on performance.

2.3 Asynchronous/Overlapped Data Transfers

Goals: to develop skills on asynchronous data transfers, Hyper-Q, and data reuse.

Ext 2.1 Consider the code sample provided in the *SimpleHyperQ_tofill.cu* file. Fill the specified sections of the code to initialise the streams, and allocate the data on the GPU. For each stream run both kernels, A and B. After the execution of all kernels, call the *sum* kernel, which operates on the same data as the previous kernels, and copy back the results. Destroy the streams and deallocate the used GPU memory.