Lab 1 - Advanced CUDA

Advanced Architectures

University of Minho

The Lab 1 focus on the development of efficient CUDA code by covering the programming principles that have a relevant impact on performance. Considering an efficient implementation of the algorithms on multicore CPUs, the performance benefits of using GPUs will be assessed, comparing various optimisations. Start by developing an efficient parallel implementation of the algorithms on CPU using OpenMP. Use the Intel Xeon E5-2695 v2 @ 2.40GHz CPU, available in any of the `compute-562` nodes, to perform the execution time measurements. Select the best of three measurements. Test the code with various problem sizes to fit the different cache levels and one size that forces the access to the RAM. This lab tutorial includes 4 exercises, part of them to be solved as homework (**HW**) and some during the lab classes (**Lab**).

1.1 Shared Memory

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

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
```

HW 1.1 Consider the one dimensional stencil algorithm (that operates on vectors). Develop an efficient implementation on CPU and measure and record its execution time. Port the code to the GPU using the CUDA environment.

Lab 1.1 Run the GPU code and measure its execution time. Compare the CPU-GPU performance. Listen to the suggestions to improve performance. Implement the suggested optimisations and measure the performance gains.

1.2 Efficient Access to Data

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

HW 1.2 Consider the one dimensional stencil code from the previous exercise. Create two kernels based on the current implementation, in such a way that it is possible to specify (i) an offset and (ii) a stride to access the elements on the input vector.

Lab 1.2 Execute the kernels multiple times with different offsets and strides and assess their execution time. 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.

1.3 Asynchronous Operation

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

Lab 1.3 Consider the code sample provided during the lab class. Fill the specified sections of the code to initialise the streams, and allocate the data on the GPU. For each stream, to run both kernel 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 free the allocated memory on the host.

1.4 Dynamic Parallelism

Goals: develop advanced skills in CUDA streams and dynamic parallelism.

HW 1.4 Consider the Quicksort algorithm[1] for sorting elements in a vector. Develop an efficient implementation of the recursive Quicksort algorithm on CPU and assess its performance on the given system. Note that it is possible to take advantage of the recursion to create nested parallelism.

Lab 1.4 Listen to suggestions to deal with recursion on CUDA GPUs. Port the code to the CUDA environment, taking advantage of the kernel capability to call itself recursively, known as dynamic parallelism. Measure the execution time and compare the CPU-GPU performance.

[1] Available at http://en.wikipedia.org/wiki/Quicksort