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 job submission), use only 4 CPU cores and do not submit interactive jobs (e.g., `qsub -lnodes=1:ppn=4:tesla,walltime=...`).

This lab tutorial includes three homework assignments (HW 2.x) and four exercises to be solved during the lab class (Lab 2.x).

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:

GNU Compiler: `module load gnu/4.8.2`.

CUDA Environment: `module load cuda/7.0.28`.

Please do not access the cluster computing nodes with `ssh`.

2.1 Shared Memory

Goals: to 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 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. Complement this CUDA skeleton with the GPU code to perform the same task.

Lab 2.1 Run the GPU code and measure and record the best execution time. Compare the CPU-GPU performance. Listen to the suggestions 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.

HW 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, or **(ii)** a stride.

Lab 2.2 Execute the kernels multiple times with different offsets and strides and assess their execution times. 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.

Lab 2.3 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 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.

2.4 Dynamic Parallelism

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

HW 2.4 Consider the Quicksort algorithm[1] to sort elements in a vector. 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.

Lab 2.4 Listen to suggestions to deal with recursion on CUDA GPUs. Complement this CUDA skeleton with the GPU code to perform the same task, taking advantage of the kernel capability to call itself recursively, known as dynamic parallelism (note that this feature is only available for Kepler GPUs). Measure, record the best execution time and compare the CPU-GPU performance.

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