Lab 4 - GPU vs Many-core

Advanced Architectures

University of Minho

The Lab 4 focus on the development of efficient matrix multiplication code for the Intel Xeon Phi (Knights Landing architecture) and NVidia GPUs computing units by covering the programming principles that have a relevant impact on performance, such as vectorisation, parallelisation, and scalability. Use the Knights Landing node and a cluster node with a NVidia Tesla GPU (by specifying the keywords `tesla` or `kepler` in the job submission, e.g., `qsub -lnodes=1:ppn=1:tesla,walltime=...`).

This lab tutorial includes one homework assignment (HW 4.1) and two exercises to be solved during the lab class (Lab 4.x).

See the previous lab sessions for instructions in how to setup the GPU/KNL environments.

4.1 Matrix Multiplication Code on Both Devices

Goals: to develop skills in the design of parallel code for the Intel Xeon Phi and NVidia GPU.
HW 4.1 Consider the matrix multiplication code provided in Lab 1. Adapt this code to run on both the Knights Landing and NVidia GPU (consider 1 CUDA thread per result matrix element). Do not optimize the code at this stage. Measure and compare the performance, also considering the original multicore code. Plot the best measurements.

4.2 Optimize Matrix Multiplication

Goals: to develop skills in tuning code to specific architectures.
Lab 4.2 Consider the initial version of the matrix multiplication for the Knights Landing. Efficient cache usage can greatly improve performance on the Xeon Phi, by avoiding accesses to the slower RAM memory. Implement tiling in the matrix multiplication algorithm to promote data reuse, possibly between threads in the same core.

Refine your implementation by aligning the data structures allocated in the device memory (see Lab 3). Measure and plot the execution time for 2048x2048 matrix sizes.
Lab 4.3 Consider the initial version of the matrix multiplication in CUDA. The GPU architecture benefits the reuse of data inside each Streaming Multiprocessor (inside each CUDA thread block). Implement tiling in the matrix multiplication algorithm, so that each tile is processed by the threads in a block.

Refine your implementation by storing the tiles in shared memory per block (using the `__shared__` clause). Measure and plot the execution time for 2048x2048 matrix sizes.