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 and NVidia GPUs computing units by covering the programming principles that have a relevant impact on performance, such as vectorisation, parallelisation, and scalability. Use a cluster node with an Intel Xeon Phi or a NVidia Tesla GPU (by specifying the keywords `phi` or `tesla` in job submission), use only 4 CPU cores and do not submit interactive jobs (e.g., `qsub -lnodes=1:ppn=4:phi,walltime=...`).

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

To load the compiler in the environment for the Intel Xeon Phi version use the following commands:

Intel Compiler: `source /share/apps/intel/compilers_and_libraries_2016.0.109/linux/bin/compilervars_global.sh intel64`.

GNU Compiler (for several system libraries): `module load gnu/4.7.2`.

To load the compiler in the environment for the NVidia GPU version use the following commands:

GNU Compiler: `module load gnu/4.8.2`.

CUDA Environment: `module load cuda/7.0.28`.

To copy the libraries and binaries to the Intel Xeon Phi see the commands in the `Makefile`.

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 Intel Xeon Phi (on offload mode) 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 Intel Xeon Phi. 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.

Refine your implementation by aligning the data structures allocated in the Xeon Phi memory (see Lab 3). Experiment with different thread affinities.

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.

4.3 Scalability

Goals: to comprehend the concepts behind the scalability on the Intel Xeon Phi and NVidia GPUs.

Lab 4.4 Develop an alternative implementation for both NVidia GPU and Xeon Phi that calls the `cublasSgemm` of cuBLAS and MKL libraries. How do these libraries performance compares to your implementation?

See the attached files for examples in how to use cuBlas and MKL libraries.