Lab 5 - GPU vs Multicore

Advanced Architectures

University of Minho

The Lab 5 focus on the development of efficient matrix multiplication code for an Intel Xeon multicore 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 a NVidia Tesla GPU (by specifying the keywords `tesla` or `k20` in the job submission, e.g., `qsub -lnodes=1:ppn=1:k20,walltime=...`).

See the previous lab sessions for instructions in how to setup the GPU environment. Copy the lab file to your home:

```
cp -r /share/cpd/lab5 .
```

5.1 Matrix Multiplication Code on Both Devices

Goals: to develop skills in the design of parallel code for Intel multicore devices and NVidia GPU.

Lab 5.1 Consider the matrix multiplication parallel code that you already developed in a previous lab session. Adapt this code to run on the 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, for a problem size of 50 MiBytes (consider square matrices with a width that has to be a power of two). Plot the best measurements.

5.2 Optimize Matrix Multiplication

Goals: to develop skills in tuning code to specific architectures.

Lab 5.2 Consider the initial version of the matrix multiplication for the multicore device. Efficient cache usage can greatly improve performance on this devices, by avoiding accesses to the slower RAM memory. Implement tiling in the matrix multiplication algorithm to promote data reuse. You can later improve by sharing that data only between threads in the same multicore device – note that the cluster node is dual-socket.

Lab 5.3 Consider the initial version of the matrix multiplication in CUDA. The GPU architecture benefits the reuse of data inside each Streaming Multiprocessor (data can be shared among CUDA threads in a 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). Compare the execution time of this implementation with the multicore version using all the cores in one, and then two devices.