Lab 3 - TFlop Performance with Xeon Phi

Advanced Architectures

University of Minho

The Lab 3 focus on the development of efficient code for the Intel Xeon Phi computing unit, covering the programming principles that have a relevant impact on performance, such as vectorisation, parallelisation, and scalability. Use any of the `compute-562` nodes with the device (check it using the `qnodes` command on the frontend) to perform the execution time measurements. It is necessary to compile the code in the compute node and **not** in the frontend. If it gives an error about a lack of license, run the following command:

```
cp /opt/intel/licenses/l_6LPZ4DMP.lic /home/YOUR_USER/intel/licenses/
```

Scripts are provided to execute the time measurements on the cluster node and to plot/visualise the results on your laptop (note that you need to have GNUPlot and Perl). This lab tutorial includes 3 exercises to be solved during the lab classes.

The goal of improving the performance of scientific applications is often to process more data per unit of time rather than to process a given data set faster (see Gustafson's Law). One example is the analysis of molecule docking, where faster applications are capable to process more time steps in a simulation. The first two exercises feature an algorithm which workload increases with the number of threads. Therefore, a throughput metric, such as GFlop/sec, is best suited for this problem instead of considering the execution time.

3.1 The Xeon Phi as a System

Goals: develop skills in the design of parallel and vectorisable code for the Xeon Phi.

Consider the *SAXPY* code provided. For each of the following steps record the floating point performance. Open two sessions on the compute node (not the frontend this time). Use one to compile the code and copy the binary and libraries to the device (see the Makefile), and use the other to access the device, using `ssh user@mic0`, and to run the code as in a normal system.

HW 3.1 The code is highly vectorisable but the Makefile explicitly excludes this compiler option. Remove this restriction, run the code again and comment on the impact that the vectorisation has on the performance of this device. How faster did you expect the code would be run and how fast it was? Can you explain this result?

Lab 3.1 Add an outer loop that iterates through a defined number of threads and parallelise it using OpenMP. Note that the iteration of the inner loops must be replicated through all threads, so make sure that the iteration counters of those loops are private. Also, each thread will only process a subset of the array, which will require a stride to be coded inside the first loop, with a step equal to the `LOOP_COUNT`. Run the code with 2 threads on the same core (note that you need to force a compact thread affinity, otherwise a 2-thread version will run in 2 separate cores). Compare the performance in GFlop/sec between these two versions. Test the code with 60, 120, and 240 threads, with both compact and scatter thread affinity. Justify the obtained results.

OMP_NUM_THREADS: environment variable to set the number of threads.

KMP_AFFINITY: environment variable to set the thread affinity.

3.2 Offloading to the Xeon Phi

Goals: develop skills in offloading parallel code to the Xeon Phi.

Lab 3.2 Consider the code developed in the previous exercise. Consider the suggestions given during the class and adapt the *SAXPY* code to be offloaded to the Xeon Phi. Measure the performance for 61, 122, and 244 threads, and compare the results with the values from the previous exercise.

Note that now the binary must not be copied to the device, as the offloading is similar to the GPU. Run the code from the host and ensure that the Xeon Phi libraries location is properly configured, since they will be accessed from the device runtime system.

```
export MIC_LD_LIBRARY_PATH=/opt/intel/composer_xe_2013.1.117/compiler/lib/mic/
```

3.3 Matrix Multiplication on the Xeon Phi

Goals: comprehend the concepts of the **Lab 3.1** and **Lab 3.2** by implementing an efficient matrix multiplication code for the Xeon Phi.

Lab 3.3 Adapt the parallel matrix multiplication code of the previous lab session to run in both native and offload modes of the Xeon Phi and modify the matrix size to $4096x4096$. Measure and compare the performance for the following 3 cases: 48 threads in the multicore Xeon CPU and 240 threads on the Xeon Phi, both in native and offload mode.