KNL Performance Comparison: HLBM

March 2017

1. Compilation, Setup and Input

Compilation

The code was compiled using the Intel C++ compiler on both the KNL nodes and the Xeon. Version 17.0.0.098 was used on the KNL while the default version 15.0.2.164 was used on the Xeon. Currently have no resources left on the Xeon to compare the performance of the difference between the two versions of the Intel compiler but it is not expected to have a large effect on the runtime of the code. The compiler option used was -O3 and the compiler directives for the source code where -D_NEW_ALLOC -D_METHOD2 which is the most efficient memory layer currently implemented in the parallel code.

Setup

The code uses MPI for the parallel communication and was run with up to 24 processes per node on the Xeon and 64 processes per node on the KNL. The performance across nodes shows linear speedup from the Xeon but as it is currently only possible to run on two KNL nodes there are not enough data points to draw any conclusions on the performance across nodes for the KNL system. The memory option used on the KNL system was quad_100 where all the MCDRAM is used to cache the main memory.

Input

The benchmark was a three-dimensional counter rotating vortex problem with periodic boundary conditions in all three directions. The lattice size was 121x241x241 and this was split into the number of MPI processes to be used. For example, for the single process run only one block was used where on the case with 16 processes the lattice was split into 16 blocks each of size 61x61x61. Using only a single block per processes give the highest possible performance due to minimizing the halo exchange data.

It should be noted that this problem size requires less than 16GB of memory and so all the data will be cached using the MCDRAM.

2. Performance Data

Figure 1 shows the parallel performance of HLBM while running on ARCHERs Xeon computer nodes. The scaling within a node shows a marked drop off in parallel performance when running on more than 4 cores per node – 8 cores in total. This is because the method is very memory bandwidth intensive and general memory bandwidth has not kept pace with the ever-increasing number of cores on CPUs. However, the performance across nodes shows linear speedup going from 1 node (24 cores) to 64 nodes (1536 cores). This is because the number of lattice points per process dropped from 288,000 when on a single node to just 4500 when on 64. This means a much larger percentage of the data could be stored in the cache which increases the core performance, but about twenty percent. This gain in sequential performance offsets the communication costs.

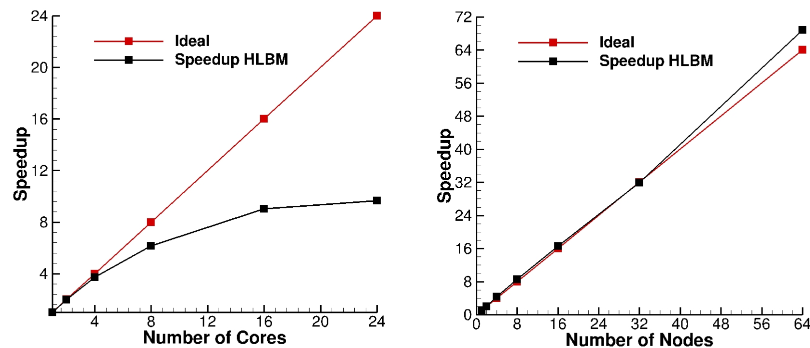

Figure 1: The speed up curve for running HLBM within and across ARCHER Xeon computer nodes – (Two 2.7GHz 12-core E5-2697 v2 Processors)

The code was also evaluated on nodes configured in cache mode with all 16GB of the on-chip Multi-Channel DRAM (MCDRAM) used to cache the system memory, and job sizes were small enough so all the data could fit within the cache. The MCDRAM is a high bandwidth memory which fits well with the needs of a HLBM. The results can be seen in Figure 2 and although the single core performance of a KNL processor was three times slower, mainly due to the lower clock speed, the parallel scaling was much better at high number of cores. Hence 24 processes on an ARCHER Xeon compute node run the same as 32 on a KNL node. This results in the KNL nodes being 80% faster when both nodes are full utilized.

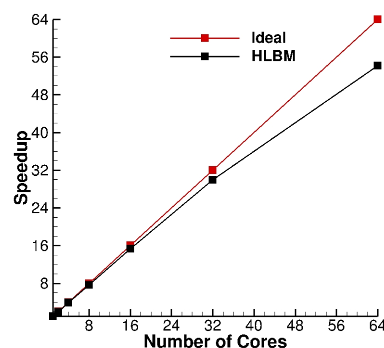

Figure 2: The speedup curve for running HLBM within an ARCHER KNL processor (model 7210) running at 1.3GHz

Table 4 shows the performance difference between the quad_100 configuration and the quad_0 configuration. The effect of having no caching between the cores and the main memory results in just over doubling the runtime of the code and shows the importance of having all the main memory cached for a code with low ratio of memory accesses to floating point operations.

archer

Number of Xeon Cores	CPU time per iteration
1	2.1249s
2	1.0721s
4	0.56952s
8	0.34484s
16	0.23538s
24	0.21953s

Table 1: Performance data for running on different numbers of cores on an ARCHER Xeon compute node.

KNL quad_100 Cores	CPU time per iteration
1	6.772s
2	3.503s
4	1.743s
8	0.880s
16	0.442s
32	0.226s
64	0.126s

Table 2 : Performance data for running on different numbers of cores on an ARCHER KNL compute node in quad_100 configuration.

Number of Nodes	Xeon Nodes (24 Cores)	KNL Nodes (64 Cores)
1	0.21953s	0.12835s
2	0.10560s	0.06350s
4	0.05050s	0.03305s
8	0.02586s	0.01729s

Table 3: Comparison of scaling across nodes for ARCHER Xeon and KNL compute nodes.

KNL Configuration	Quad_100	Quad_0
CPU time per iteration	0.1286s	0.2942s

Table 4: Comparison of performance on a since KNL compute node in two different configurations.

3. Summary and Conclusions

Although the single core performance of the KNL is much slower than that of the Xeon, mainly due to the much slower clock speed, the performance per node was 80% greater. Since HLBM is aimed at the computation of wakes in real time any performance gain is welcome. Due to the smaller lattice sizes used in this type of application the situation where there is not enough cached memory on the KNL nodes will not arise. For larger lattices which cannot be fully cached, the performance advantage of the KNL nodes may well be lost.

A compact OpenMP version of the code is also being tested on the KNL nodes which has been highly optimized to reduce the work load but loop unrolling removing redundant calculation etc. However due all these optimizations, it fails to take advantage of the vector processing units with the KNL node. Different computational kernels are being tested to see if it possible to rework the code so it can make use of the VPUs. The basic code using just the compiler to the optimization is slower in number of lattice updates per second but much faster looking at raw floating point operations. It is thought there could be a version in between these two extremes which is faster than the current optimized version. It has already been determined that a simple padding of the workspace so the inner loop always has memory aligned access is detrimental to the overall performance of the code.