Abstract
This poster presents the optimization work carried out on a Smoothed Particle Hydrodynamics (SPH) code Phantom on Haswell and KNL. With 8 steps of optimization, the performance results show 3x speedup on Haswell and 4x on KNL. A few remarks are provided to highlight the differences on KNL in terms of architecture and optimization strategy. This work will also serve as an example of optimizing other codes, such as those in molecular dynamics, computer graphics and neural networks that also deal with id-tree, neighbour search, irregular computation and random memory accesses.

Codebase and Platforms
Phantom \cite{1} is a smoothed particle hydrodynamics and magnetohydrodynamics code for Astrophysics. It is parallelized with OpenMP and widely used for studies of accretion discs, turbulence and star formation. There are two main data structures: maketree, getneigh, density-iterate and force.maketree builds a three dimension tree for all the particles (Fig.2), and densitize and force calculate the density and force of a particular particle based on the neighbour list provided by getneigh through tree traversal.

Employ nested OpenMP parallel regions. Two serial code regions in maketree are optimized by master thread in the breadth-first build. Nested OMP regions are used to improve thread concurrency so that all threads work on node 1, 2 teams of half the threads work on node 2-3 simultaneously, and so on (Fig.3).

Step 3: Remove linked lists in maketree to reduce random memory access and LLC miss. This also helps reduce the serial code since part of it can now be parallelized with the 3 new arrays representing the tree.

Move particles as the tree is built. Particles positions are copied at each level. The memory access pattern has proved to have enough benefit to overcome this through vectorization (Fig.7).

Step 4: Change data layout from AoS to SoA for getneigh in maketree to improve vectorization efficiency in the calculation of centre of mass, node size, quadruple moments and other quantities.

Step 5: Vectorize the inner loop of getneigh. There is dependency in the outer loop since child nodes cannot be determined to be relevant until the parent node is known. The new tree representation from Step 3. With branches moved outside, it is efficiently vectorized.

Data structure: cache arrays replace cache and detach are changed from AoS to SoA to make sure summation loops get dens_sums and compute_force more efficient. Self particle is excluded from the neighbour list to help remove the loop statement in get_dens_sums and compute_force to make them pure compute, vectorized.

Step 6: Vectorize density and force summation loops in get_dens_sums and compute_force. Branches are brought outside, functions are inlined, and loop fission is applied based on the sparsity of data.

Step 8: Improve vectorization efficiency in get_dens_sums and compute_force. Add CONTIGUOUS attribute to assumed shape arrays to avoid multidimensional arrays. Adjust size of cache arrays to make sure that column length is a multiple of vector length as well as columns are 32byte aligned on Haswell for multidimensional arrays and 64byte on KNL. Use %fomp simd declare annotation, %DIR$ FORCEINLINE directive, and %fomp compile flag to ensure function calls (land calls within calls) are properly inlined.

After these 8 steps, loops have been moved rightwards and upwards on the roofline model for both Haswell and KNL (Fig.6 for KNL). Walltime has been reduced by 3x on Haswell and 4x on KNL (Fig.7), and this means months of time is saved for a typical simulation in real life with low-to-medium resolution and millions of timesteps.

Fig.1: Improved roofline efficiency for Phantom on KNL.

CITATION

Optimizing Smoothed Particle Hydrodynamics Code Phantom on Haswell and KNL

Charlene Yang
Chris Burdige
Pawsey Supercomputing Centre, Perth, Australia

Daniel Price
Rebecca Heelan
Murdoch University, Melbourne, Australia

Step 1:
Adjust OpenMP scheduling. Since the total number of calls in the 100,000's and 70% of them are to getneigh, heuristics need to be studied. The Phantom has been optimized with 32 threads to achieve the best load balance and cache.

Step 2:
Adjust number of threads to be a power of 2. Even though there are 32 cores on Haswell, it is usually given to have a better load balance since the tree is based on a brick build and every level in the breadth-first stage has a 2^n to a 2^n number of nodes (Fig.3). Number of threads on KNL stays the same as the baseline code was written to utilise the other cores on Haswell.

References
[3] Pawsey Supercomputing Centre, Perth, Australia
[4] Monash University, Melbourne, Australia

Optimization Path and Results

Step 1:
Adjust OpenMP scheduling. Since the total number of calls in the 100,000's and 70% of them are to getneigh, heuristics need to be studied. The Phantom has been optimized with 32 threads to achieve the best load balance and cache.

Step 2:
Adjust number of threads to be a power of 2. Even though there are 32 cores on Haswell, it is usually given to have a better load balance since the tree is based on a brick build and every level in the breadth-first stage has a 2^n to a 2^n number of nodes (Fig.3). Number of threads on KNL stays the same as the baseline code was written to utilise the other cores on Haswell.

Remarks on Optimization Strategies

• Thread parallelism: KNL has more cores than Haswell but all are running at a much lower frequency. Well balanced load is critical for codes to achieve good thread concurrency and thus performance.

• Data parallelism: KNL has wider vectors and improved instruction sets. Data parallelism should be exploited as much as possible to auto-vectorize and improve performance.

• Memory awareness: Data layout, alignment and spacial/temporal locality is more important on KNL than Haswell due to its wider vectors and lack of L2 cache. Vector-friendly and cache-friendly data arrangement is greatly encouraged. MCDRAM also favors cache-friendly codes because even though it has higher bandwidth, its latency still stays close to DDR's.

Fig.3: Different parallelisms in the d-tree build.

Fig.6: Cache-aware roofline model on KNL.

Fig.7: Optimization results on KNL.

Fig.6: Cache-aware roofline model on KNL.

Fig.7: Optimization results on KNL.

Fig.8: Top right: execute busy streamlines, bottom left: vectorization efficiency.