Improving the performance of a Gadget kernel on many-core systems – from KNC to KNL

Luigi Iapichino and Fabio Baruffa, Leibniz-Rechenzentrum (LRZ), Garching b. München, Germany, in the framework of the Intel® Parallel Computing Center in Garching (LRZ – TUM)

Collaborator: N. Hammer (LRZ)

Project partners: M. Petkova, K. Dolag (USM München, Germany)
Gadget: numerical simulations of cosmological structure formation

- Publicly available, cosmological TreePM N-body + SPH code for the simulation of the build-up of the cosmic large-scale structure.
- Good scaling performance up to $O(100k)$ Xeon cores (SuperMUC @ LRZ).
- Most of the work performed on a stand-alone, pure Open-MP representative code kernel.
Node-level performance comparison between HSW, KNC and KNL

Features of the KNL tests:

- native runs on Xeon Phi™ 7210 @ 1.30GHz (KNL), 64 cores
- Intel® compiler 2016, -xmic-avx512
- KMP Affinity: scatter; Memory mode: Flat; Cluster mode: Quadrant.

Results:

- Previous optimisations (data layout, vectorisation) improved the speedup on all systems, by different factors.
- KNL scalability slightly better than HSW and KNC up to 128 threads.
- Necessity of using hyperthreading can be different between KNC and KNL.
Performance comparison: first results including KNL

- Initial version vs. vectorised including all optimisations.
- IVB, HSW: 1 socket w/o hyperthreading. KNC: 1 MIC, 240 threads. KNL: 1 node, 128 threads.
- Performance gain for Xeon Phi™ larger than for Xeon.
- Single-core execution time on KNL: 3.3x faster than on KNC.
Vectorisation: improvements from HSW to KNL

- Vectorisation of the kernel main "compute" loop (red bar) through better localized masking.
- On KNL: measured loop speed-up 6.6x. A vector efficiency of 83% is reached without using intrinsics.
- Both on HSW and KNL, vectorisation provides some performance improvement also in other parts of the kernel.
Summary and outlook

Analysis of a representative kernel of Gadget3.

Optimisation of a prototypical vector loop.

Vectorised workload in the kernel is quite small, but important lessons learnt in view of backporting to Gadget.

**Impressive vectorisation efficiency with little effort on KNL.**

Encouraging performance reached within a short time on KNL, looking forward to further tuning on full code.

**Full out-of-the-box Gadget runs on one KNL node:** the code works, execution time 1.6x slower than on one HSW node, but obviously lots of room for optimisation.
Backup I: obstacles to vectorization efficiency - pseudocode

for (n = 0, n < neighbouring particles (selected)) {
    j = ngblist[n];  // getting the index from the particle data structure (SoA)

    if (particle n within smoothing length) {
        // Problem 1: if statement
        inlined_function1(.....);
        inlined_function2(.....);
    }

    vx += NewPart.Vel[0][j];  // Problem 2: indirect (strided) access to the data

    ... 

    v2 += NewPart.Vel[0][j] * NewPart.Vel[0][j] + ... ;  // additional load
    // (unnecessary): why does the compiler not reuse it from the register?
}
for (n = 0, n < neighbouring particles (selected)) {
    j = ngblist[n];  // getting the index from the particle data structure (SoA)

    inlined_function1(.....);  // the if condition is moved inside the function
    inlined_function2(.....);

    vel1 = NewPart.Vel[0][j];  // still strided data access
    ...
    vx += vel1;  // optimised data load
    ...
    v2 += vel1 * vel1 + ... ;
}