Preconditioning for Data Locality

Quickly selecting nearest particles in the Friends-of-Friends component of Gadget

Luigi Iapichino
Leibniz-Rechenzentrum (LRZ), Garching b. München, Germany

Collaborators: V. Karakasis, N. Hammer, A. Karmakar (LRZ)
in the framework of the Intel® Parallel Computing Center in Garching (LRZ – TUM)

Partners: M. Petkova, K. Dolag (USM München, Germany)

Intel contributors to this presentation: CJ Newburn, Michael Brown, Ashish Jha, David Kunzman
Managing locality of data that changes over time

Types of problems and examples

- NBody - astrophysics (SWIFT, COSMOS)
- Molecular dynamics - small scale (GROMACS, NAMD, miniMD)
- Particle in cell - sub-atomic scale
- Adaptive mesh problems (FEA, Crash)
Forms of improving locality

- Work with indices, e.g. an array of indices
  - Select to reduce scope of what to evaluate
  - Sort to help prioritize or improve temporal locality

- Work with actual data layout
  - Pared subset of data of interest, e.g. only struct members that get used
  - Arranged data, for better temporal and spatial locality, including for stride 1
Index management

- One or more levels of hierarchy of locality
  - Tree where leaves are at the lowest scope
- Friends of friends component of Gadget (40% of execution time)
  - Find K (295) nearest neighbors (out of 500) to compute interactions with current particle
  - Create an array of indices for neighbors, without moving particle data
  - Principle: condition data only as much as necessary
  - Application: select $\text{dist}_i < \text{dist}_K$, for each $i < K$  vs. sort

- 20-35% on IVB, 10-20% on KNC
Data re-layout

- Space-filling curves or even just by 3D binning
  - Alternative to index management
- Principle: Be selective about what is moved when, do as little as possible
  - May move some data at major time steps, and tweak faster-changing data at minor steps
- Benefits
  - Increased spatial locality in cache and TLB, e.g. eliminate non-needed struct members
  - Manage traversal through data to increase temporal locality, e.g. proceed to neighbors
- While you’re already re-laying out data
  - Rearrange AoS as SoA, perform vector loads vs. gathers
  - Pad for structs if AoS, or dummy elements at infinite distance if SoA
  - Load balancing, so similar numbers of neighbors span nodes
  - Separate halo and core data to help with communication hiding
Factors determining benefits from pre-conditioning

● Costs
  - Rate of particle movement determines # steps to amortize across
  - Ratio of cutoff to preconditioned subset, and preconditions subset to whole

● Benefits
  - Spatial and temporal locality, elimination of gathers

● Impact
  - Diluted with respect to other work - computation intensity
  - Benefits enhanced by repeated usage
  - How the problem is partitioned
Principles

● Think critically about locality
  - What matters, e.g. select (threshold) vs. sort (total order)
  - How it’s maintained, e.g. only parts affected
  - When it’s maintained, e.g. incrementally, as particle data positions updated
  - Increasing work is ok if it significantly benefits locality, e.g. avoid conditionals

● Consider a hierarchical approach
  - Different actions at different granularities of time steps
  - Arrange data to easily manage locality
Supporting materials

- FoF fraction of execution time
- FoF data locality and neighbor search
- FoF bottlenecks and remedies
- FoF performance results
- Benefits of data re-layout
- 3D binning illustration
- Why AoS may remain good
- Transpose AoS to SoA in registers
- miniMD
Why Friends-of-Friends?

- Friends-of-Friends (FoF) computations consume half of the Gadget execution time at late simulation times.
  - Halo finder component using the Subfind algorithm.

- The FoF component is integrated into the Gadget workflow.
  - Tree-walk
  - FoF computations

Simulation time (norm.)
Bottlenecks and remedies

Peculiar features of the current implementation:
- No particle rearrangement needed at sorting stage → data locality already enforced at global time steps.
- Optimization opportunity → the sort information is not needed after the selection/truncation

Resulting strategy for optimisation:
- Replaced complete sorting with partial sorting so that the final neighbor list meets only the following property:
  - \( \text{dist}_i < \text{dist}_K \), for each \( i < K \)
  - O(N) complexity → This optimization becomes more significant for very large neighbor lists
  - 20–35% improvement on IVB, 10–20% on KNC
- Impact on KNC less impressive → KNC-optimized default qsort() implementation?
Performance results

- Complexity really pays off at large scales
- Less impressive impact within Gadget
  - Neighbor list to select from is rather small (approx. 500 particles)
  - Parallelization overhead of the current implementation trims the benefits in higher thread counts
Lessons learnt and future directions

- Our approach to particle selection is effective in cases where:
  - Data locality is guaranteed in the input data layout;
  - Full neighbor sorting is not needed in the algorithm.

- With larger neighbor list, it may be even more convenient.

- What about the performance on the Xeon Phi?
todo_particle_list = particle_list

while not empty(todo_particle_list):
  for p in todo_particle_list:
    ngblist = get_neighbors(p, Hsml)  # Get the nearest neighbors physically
    qsort(ngblist)  # Complexity $O(N\log N)$
    ngblist = ngblist[:K]  # Keep the K nearest elements
    for n in ngblist:
      compute_interactions(p, n)  # Iterate over the neighbors
  todo_particle_list = mark_elements_to_recompute(particle_list)
Benefits of data re-layout (1)

- Increased spatial locality - lower cache and TLB misses
  - Eliminate non-needed struct members (LRZ)

- Increase temporal locality
  - Limit the scope of search for neighbors within cutoff, if arranged properly

- Combine with spatial sort of atom data for better temporal locality
  - Neighbors of next atom in loop likely to overlap with current atom
  - Improve cache efficiency for thread hot-teams
  - Reduce vector inefficiency due to cutoff check in force calc (note that it may be ok to check neighbors beyond the cutoff, if that helps vectorization efficiency by reducing conditional branching)
  - Create the possibility of unit stride accesses
  - Better HW prefetching
  - Replace gathers with vector loads
Benefits of data re-layout (2)

- Create the possibility of padding, for the sake of aligning structs
  - Atom data is not split across cache lines
  - Compiler has more flexibility for repacking as SoA into vector registers
- Load balancing
  - Avoid the situation where some particles have a disproportionately larger fraction of neighbors whose particle data has a longer-latency access, e.g. is in a different node or tile
- Improved vector efficiency
  - “Dummy” atoms placed at infinity will always fail cutoff checks, and can be used to ensure number of neighbours % VLEN == 0
- Communication hiding
  - Separating local/remote neighbors enables remote neighbors to be processed either first (to overlap comms), differently (to avoid force communication) or somewhere else (on the host)
3D binning

- Particles with same color are contiguous in memory
- Bottom picture shows a sort by current \(<x,y,z>\) location
Why AoS may remain good

- Random access to elements in an array of structures, all members in cache line

**Structure of Arrays (SoA)**

All elements potentially on separate cachelines, especially as i, j, k, and m spread out.

*(higher cache pressure per iteration and less efficient use of L1 cache ports)*

**Array of Structures (AoS)**

Elements for the same index located on the same cacheline.

*(lower cache pressure per iteration and more efficient use of L1 cache ports)*
Transpose from AoS to SoA in registers

- Random access to elements in an array of structures, all members in cache line
- Gather vector-length (vlen) elements
- Transpose them explicitly with shuffles, etc., into packed registers, so the corresponding member for vlen elements can be operated on with SIMD
- Justifiable with adequate reuse (NAMD, miniMD), many-body potentials
miniMD Gather/Scatter Optimizations

Ashish Jha
MD: Force Compute

```c
for(int i = 0; i < nlocal; i++) {
    //iterate through all Atom [864000]
    neighs = &neighbor.neighbors[i * neighbor.numneigh];
    const int numneighs = neighbor.numneigh[i];
    const MMD_float xtmp = x[i * PAD + 0];
    const MMD_float ytmp = x[i * PAD + 1];
    const MMD_float ztmp = x[i * PAD + 2];
    MMD_float fix = 0;
    MMD_float fiy = 0;
    MMD_float fiz = 0;
    for(int i = 0; i < numneighs; i++) {
        //iterate over Neighbors of i'th Atom
        const int j = neighs[i];
        const MMD_float delx = xtmp - x[j * PAD + 0];
        const MMD_float dely = ytmp - x[j * PAD + 1];
        const MMD_float delz = ztmp - x[j * PAD + 2];
        const MMD_float rsq = delx * delx + dely * dely + delz * delz;
        if(rsq < cutforcesq) {
            const MMD_float sr2 = 1.0 / rsq;
            const MMD_float sr6 = sr2 * sr2 * sr2;
            const MMD_float force = 48.0 * sr6 * (sr6 - 0.5) * sr2;
            fix += delx * force;
            fiy += dely * force;
            fiz += delz * force;
            if(EVFLAG) {
                t_eng_vdwl += sr6 * (sr6 - 1.0);
                t_virial += delx * delx * force + dely * dely
            }
        }
    }
    //for i
    f[i * PAD + 0] += fix;
    f[i * PAD + 1] += fiy;
    f[i * PAD + 2] += fiz;
} //for i
```

Neighbors “rebuilt” at certain time-step and random
Impl: HW vs. SW Gather

HW G/S codegen for KNC:
-miGLOB_default_function_attrs="gather_scatter_loop_unroll=7;
use_gather_scatter_hint=on"
..L769:
  vgatherdpd (%r15,%zmm18,8), %zmm19{%k3}
  jknzd  ..L768,
  %k3
  vgatherdpd (%r15,%zmm18,8), %zmm19{%k3}
  vgatherdpd (%r15,%zmm18,8), %zmm19{%k3}
  vgatherdpd (%r15,%zmm18,8), %zmm19{%k3}
  vgatherdpd (%r15,%zmm18,8), %zmm19{%k3}
  vgatherdpd (%r15,%zmm18,8), %zmm19{%k3}
  vgatherdpd (%r15,%zmm18,8), %zmm19{%k3}

SW G/S – taking advantage of spatial locality of elements of an Atom

Similar opportunity in “neighbor_build” function as in “Force_Compute”

Full “function” implemented in Intrinsic KNC, HSW

MD Apps are G/S intensive
SW G/S takes advantage of “spatial locality” of Atom elements
Performance App-level

- Older miniMD; results at App-level
- No Data layout changes!!!
  - shows feasibility and impact of improving data locality to the Vector Compute units w/o fundamental code changes
- Latest Intel Compiler now able to incorporate this optimization by identifying AoS patterns
  - some restrictions, not all cases

<table>
<thead>
<tr>
<th>Measurements</th>
<th>Gains: C-Code/Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon 2S IVB</td>
<td>Xeon Phi KNC</td>
</tr>
<tr>
<td>Xeon 2S IVB</td>
<td>Xeon Phi KNC</td>
</tr>
<tr>
<td>Baseline C code: HW G/S; autovectorized by ICC</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Intrinsic SW G/S: Force &amp; Neigh func for KNC, Force for IVB</td>
<td></td>
</tr>
<tr>
<td>1.76</td>
<td>2.12</td>
</tr>
</tbody>
</table>