Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

A HPM based Methodology for SIMD Code Analysis

High Performance Computing

Example: MiniMD proxy app

Proxy app from **MANTEVO**.

`https://mantevo.org`

- **`MiniMD`** – MPI + OpenMP
  - App. 3000 lines, C++
  - Extracted from LAMMPS (`http://lammps.sandia.gov`) Molecular dynamics application code
  - Other implementations: OpenACC, OpenCL, Kokkos
  - Supports Lennard-Jones (default) and eam force calculation
  - Uses neighbour lists

Runtime profile

% time	self seconds	calls	name
83.05	21.44	402	ForceLJ::compute(Atom&, Neighbor&, Comm&, int)
13.29	3.43	21	Neighbor::build(Atom&)
1.63	0.42	1	Integrate::run(Atom&, Force*, Neighbor&, Comm&
0.50	0.13	2280	Atom::pack_comm(int, int*, double*, int*)
0.46	0.12	2412	Atom::unpack_reverse(int, int*, double*)
0.23	0.06	41	Neighbor::binatoms(Atom&, int)
0.19	0.05	2280	Atom::unpack_comm(int, int, double*)
0.19	0.05	21	Comm::borders(Atom&)
0.15	0.04	20	Atom::sort(Neighbor&)
0.12	0.03		intel_avx_rep_memcpy*

Code review

- `ForceLJ::compute` is a proxy routine to jump to different implementations:
  - `compute_halfneigh` – Default for running with 1 thread
  - `compute_halfneigh_threaded` – Default with more than 1 thread
  - `compute_fullneigh` – Optional algorithm

- The optimal algorithm is halfneigh updating both partners for force calculations. But requires atomics for OpenMP parallelisation and only vectorizes if forced with pragma SIMD.

Code review halfneigh compute

131072

78

```
for(int i = 0; i < nlocal; i++) {
    neighs = &neighbor.neighbors[i * neighbor.maxneighs];
    const int numneighs = neighbor.numneigh[i];
    const MMD_float xtmp =
    const MMD_float ytmp =
    const MMD_float ztmp =
    MMD_float fix = 0.0;
    MMD_float fiy = 0.0;
    MMD_float fiz = 0.0;

    for(int k = 0; k < numneighs; k++) {
        const int j = neighs[k];
        const MMD_float delx = xtmp -
        const MMD_float dely = ytmp -
        const MMD_float delz = ztmp -
        const MMD_float rsq = delx * delx + dely * dely + delz * delz;
        // CUTOFF LOOP SEE NEXT SLIDE
    }
    f[i * PAD + 0] += fix;
    f[i * PAD + 1] += fiy;
    f[i * PAD + 2] += fiz;
}
```

Code review cutoff loop `halfneigh` vs `fullneigh`

halfneigh

```
if(rsq < cutforcesq) {
            const MMD_float sr2 = 1.0 / rsq;
            const MMD_float sr6 = sr2 * sr2 * sr2 * sigma6;
            const MMD_float force = 48.0 * sr6 * (sr6 - 0.5) * sr2 * epsilon;

            fix += delx * force;
            fiy += dely * force;
            fiz += delz * force;

            f[j * PAD + 0] -= delx * force;
            f[j * PAD + 0] -= dely * force;
            f[j * PAD + 0] -= delz * force;
}
```

Enter 54 times of 78

fullneigh

```
if(rsq < cutforcesq) {
            const MMD_float sr2 = 1.0 / rsq;
            const MMD_float sr6 = sr2 * sr2 * sr2 * sigma6;
            const MMD_float force = 48.0 * sr6 * (sr6 - 0.5) * sr2 * epsilon;

            fix += delx * force;
            fiy += dely * force;
            fiz += delz * force;
}
```

`mul`: **10**

`add`: **4**

`div`: **1 reciprocal**

Xeon Broadwell system (2 sockets, 18 cores per socket, 2.3 GHz)

Runtime [s]	scalar	sse	avx	avx2
halfneigh	9.00	8.13	6.98	8.19
fullneigh	**15.17** (+69%)	9.15 (+13%)	5.92 (-15%)	**5.68** (-19%)

Scalar vs SIMD	FullNeigh	HalfNeigh
Runtime factor	2.56	1.29
Arithmetic instr. factor	**3.04**	2.66
Total instruction factor	1.80	**0.91**
CPI factor	1.42	1.42
Overall scaling	2.55	1.29

- Halfneigh requires a factor 1.78 less **arithmetic work** which is reflected in a scalar runtime scaling of 1.69
- But because Fullneigh is easier to SIMD vectorize and with AVX it beats the better algorithm by a small margin

Likwid HPM instruction breakdown

Instruction type	Full scalar	Full AVX	Half scalar	Half AVX
CPI	0.98	**0.69**	0.78	**0.55**
Arithmetic Scalar	69.16%	4.83%	51.5%	4.84%
Arithmetic SSE	0%	0.21%	0%	0.14%
Arithmetic AVX	0%	35.82%	0%	12.6%
Arithmetic Total	**69.16%**	40.85%	51.5%	**17.58%**
Load	14.19%	17.03%	16.96%	13.65%
Store	0.23%	1.06%	4.49%	5.64%
Branch	5.27%	3.61%	4.06%	14.34%
Other	10.45%	**37.45%**	22.99%	**48.80%**

MiniMD Assembly AVX

```
const int j = neighs[k];
```

AVX: k in blocks of 4

```
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;
```

```
vmulpd     %ymm9, %ymm9, %ymm12
vfmadd231pd %ymm7, %ymm7, %ymm12
vfmadd231pd %ymm10, %ymm10, %ymm12
```

X[j+x]

0+0	0+1	0+2	1+0	1+1	1+2
2+0	2+1	2+2	3+0	3+1	3+2

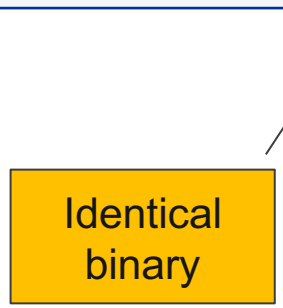
ymm[0-3]

0+0	1+0	2+0	3+0
0+1	1+1	2+1	3+1
0+2	1+2	2+2	3+2

Xeon Skylake system (2 sockets, 20 cores per socket, 2.4 GHz)

Runtime [s]	scalar	sse	avx	avx2
halfneigh	9.00	8.13	6.98	8.19
fullneigh	**15.17** (+69%)	9.15 (+13%)	5.92 (-15%)	**5.68** (-31%)

Runtime [s]	scalar	sse	avx	avx512
halfneigh	13.70	13.62	7.17	3.39
fullneigh	**24.43** (+78%)	24.20 (+78%)	5.53 (-33%)	**2.44** (-38%)



AVX512 version

```
        vpcmpgtd  %ymm23, %ymm22, %k3
        vmovdqu32 (%r13,%r14,4), %ymm14{%k3}{z}
        kmovw     %k3, %r8d
        vpaddd    %ymm14, %ymm14, %ymm15
        vpaddd    %ymm15, %ymm14, %ymm18
..B12.39:
        kmovw     %k3, %k1
        kmovw     %k3, %k2
        vpxord    %zmm19, %zmm19, %zmm19
        vpxord    %zmm16, %zmm16, %zmm16
        vpxord    %zmm14, %zmm14, %zmm14
        vgatherdpd 16(%rdx,%ymm18,8), %zmm19{%k1}
        vgatherdpd 8(%rdx,%ymm18,8), %zmm16{%k2}
        vgatherdpd (%rdx,%ymm18,8), %zmm14{%k3}
..B12.40:
        vsubpd    %zmm19, %zmm24, %zmm19
        vsubpd    %zmm16, %zmm20, %zmm17
        vsubpd    %zmm14, %zmm21, %zmm15
        vpaddd    %ymm18, %ymm9, %ymm14
        vpaddd    %ymm18, %ymm8, %ymm16
        vmulpd    %zmm17, %zmm17, %zmm26
        vfmadd231pd %zmm15, %zmm15, %zmm26
        vfmadd231pd %zmm19, %zmm19, %zmm26
```