Optimization of D3Q19 Lattice Boltzmann Kernels for Recent Multi- and Many-cores Intel Based Systems

Ivan Girotto[1235], Sebastiano Fabio Schifano[24] and Federico Toschi[5]

[1] International Centre of Theoretical Physics (ICTP)

[2] University of Ferrara (UNIFE)

[3]University of Modena and Reggio Emilia (UNIMORE)

[4] Italian National Institute of Nuclear Physics (INFN)

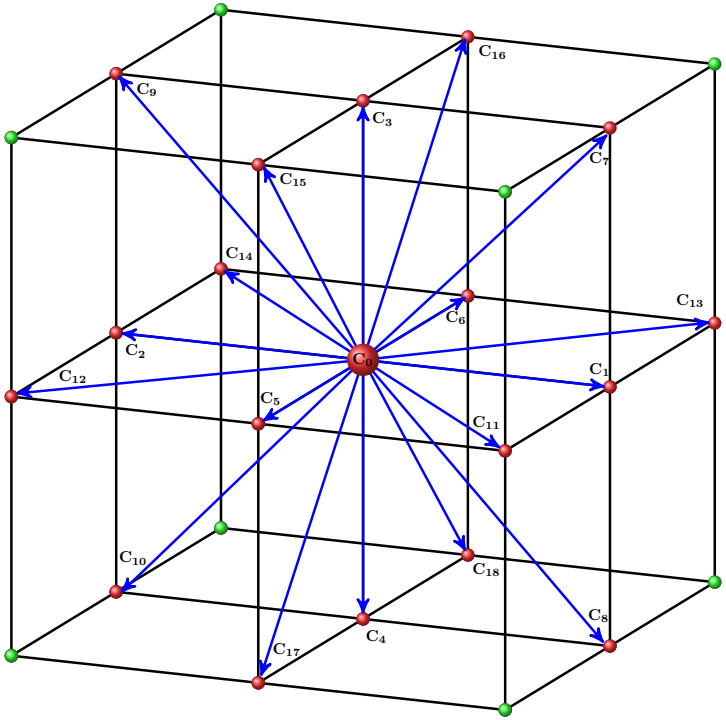
[5] Eindhoven University of Technology (TU/e)

Outline

- Lattice Boltzmann Method (LBM)
- More efficient data layouts
- Reference computer architectures: CINECA Marconi
- Intel compiler vetorization
- Performance analysis and optimization results
- Conclusions

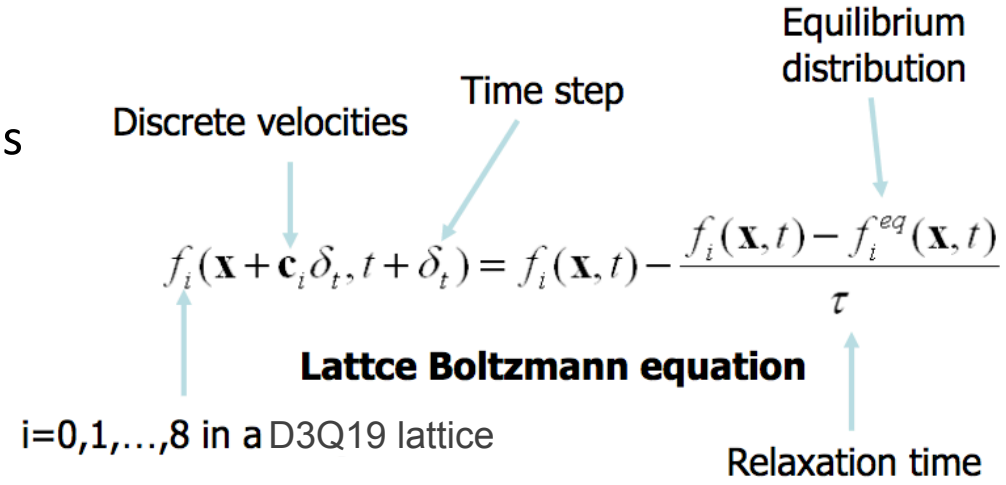
LB3D: Application Description

- Lattice Boltzmann Method: Computational fluid dynamics method for solving complex fluid flows
- D3Q19 LB application, a 3-dimensional model with a set of 19 populations elements corresponding to (pseudo-)particles moving one lattice point away along all 19 possible directions.
- This model is today widely used for carry-on extensively simulations of several types of fluids. In particular, we will present results achieved optimizing the main computational kernels included in a numerical code that is based on the Lattice Boltzmann method (LBM3D)
- The optimization experience was made starting from the original version of a production code developed by the group of Prof. Toschi @ TU/e



Lattice Boltzmann Method (LBM)

- Continuous lattice Boltmann equation describe the probability distribution function in a continuous space phase
- LBM is discretized in time, space, velocity space (directions)




```
1:  for all time step do
2:      < Set boundary conditions >
3:      for all lattice site do
4:              < Move >
5:      for all lattice site do
6:              < Hydrovar >
7:      for all lattice site do
8:              < Equili >
9:      for all lattice site do
10:             < Collis >
11:     end for
12: end for
```

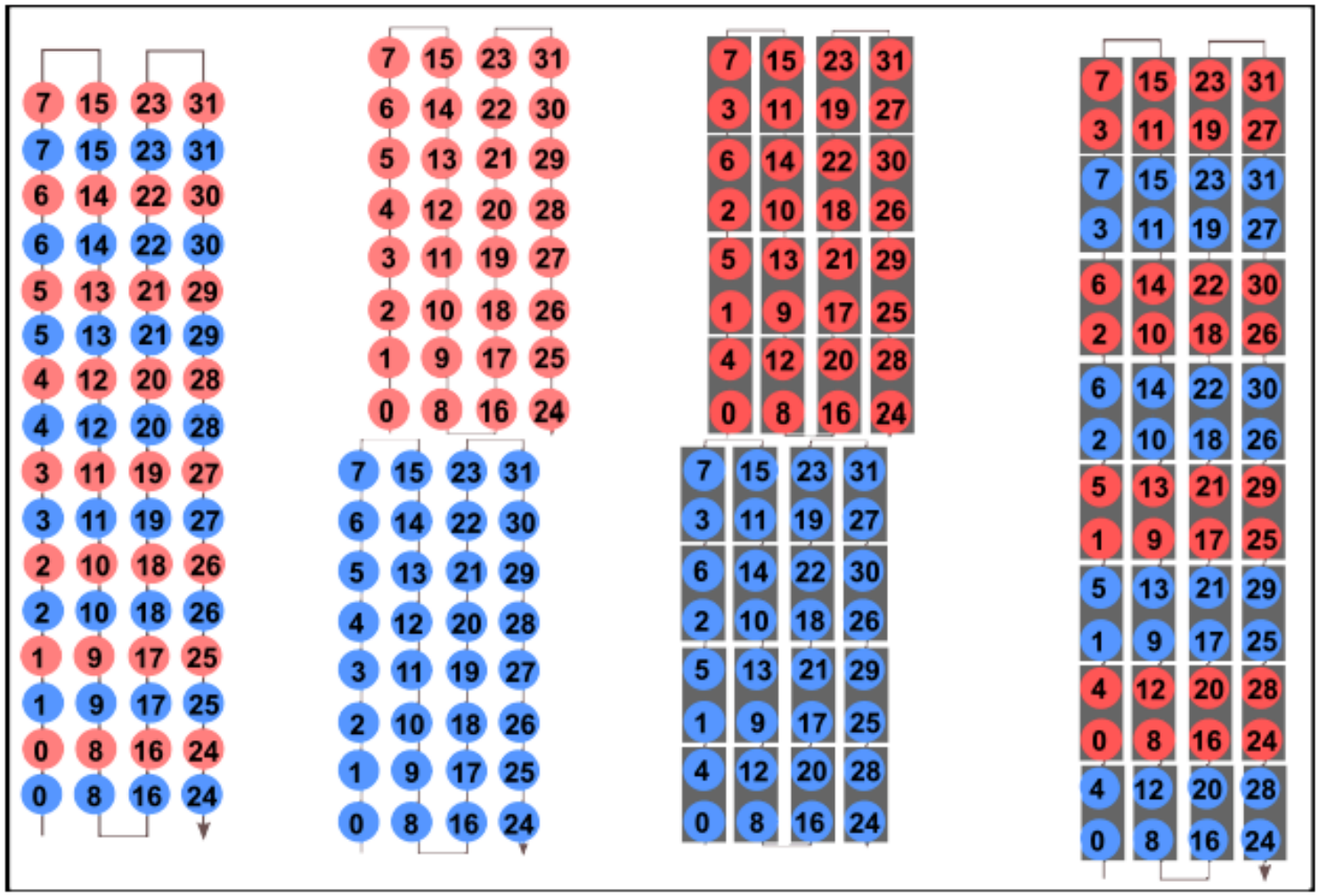
```
1: for all time step do
2:      < Set boundary conditions >
3:      for all lattice site do
4:              < Move >
5:      for all lattice site do
6:              < Collide_Fused >
7:      end for
8: end for
```

```
1: for all time step do
2:      < Set boundary conditions >
3:      for all lattice site do
4:              < Move_Collide_Fused>
6:      end for
7: end for
```

Loop compression and better data locality!!

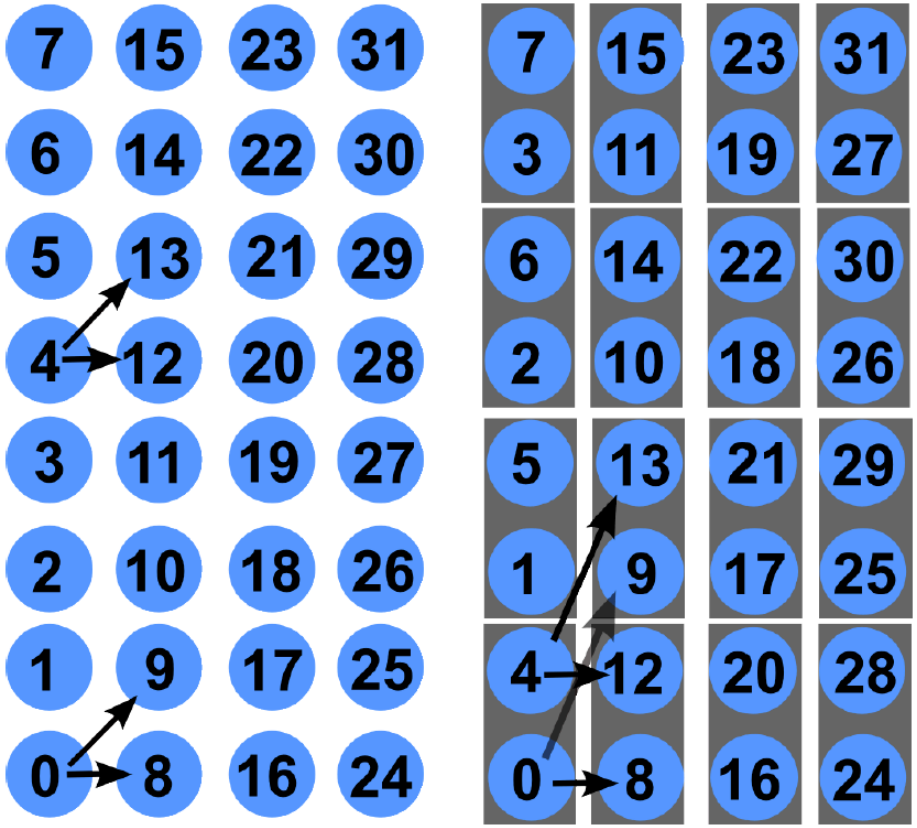
Data Structure

Lattice 4 $\times$ 8 with two (blu and red) population per site.
Left to right: Array of Structures (AoS), Structure of Arrays (SoA),
Clustered Structure of Arrays (CSoA), Clustered Array of Structure of Arrays (CAoSoA).



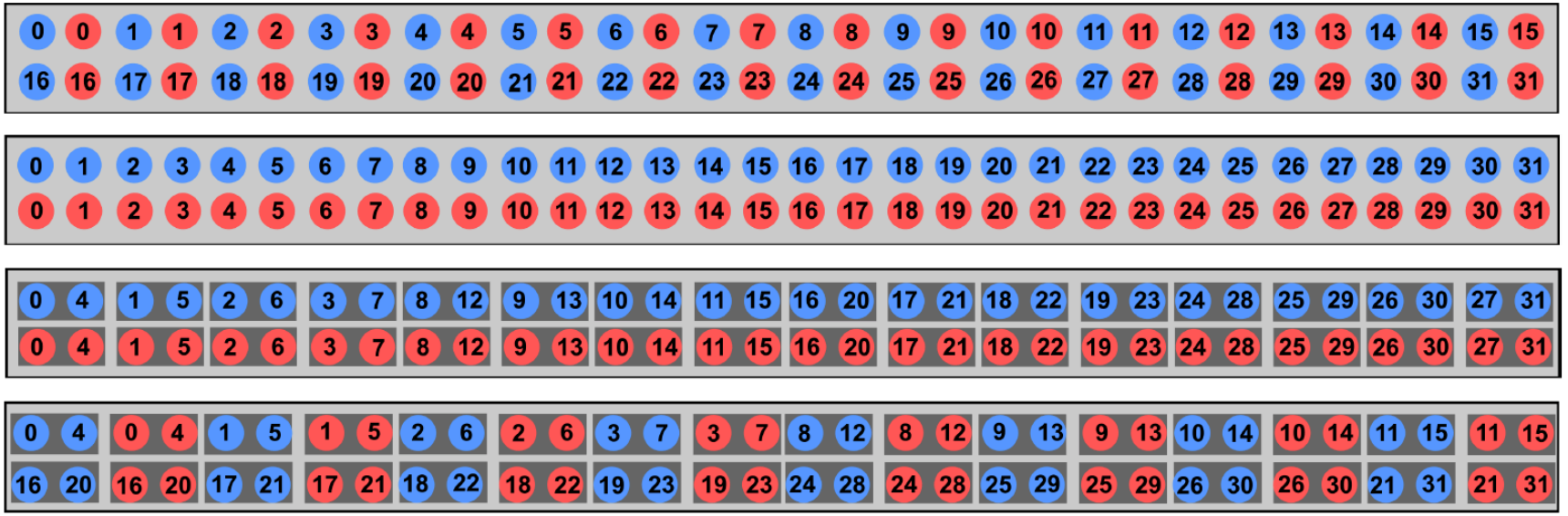
SoA vs CSoA

- Lattice 4×8
- machine vector size of 2-doubles:
- memory alignement is 8 Bytes
- process two sites in parallel
- $0 \rightarrow 8$ has read and write aligned
- $0 \rightarrow 9$ has read aligned and write mis-aligned
- $(0, 4) \rightarrow (8, 12)$ has read and write aligned
- $(0, 4) \rightarrow (9, 13)$ has read and write aligned
- clusters close to borders need special handling



Data Structure

Lattice 4 $\times$ 8 with two (blu and red) population per site.



Top to bottom

- Array of Structures (AoS)
- Structure of Arrays (SoA)
- Clustered Structure of Arrays (CSoA)
- Clustered Array of Structure of Arrays (CAoSoA)

CINECA Marconi: reference computer architectures (socket)

Marconi Sections	Intel CPU Model	Clock Frequency (GHz)	# Cores (Threads)	Vector Extension	Peak Perf (GFLOP/s)	Memory Bandwidth (GB/s)
A1	Xeon E5-2697 v4	2.3	18 (18)	AVX2	662.4	~76.8*
A2	Xeon Phi 7250 CPU	1.4	68 (272)	AVX-512	3046.4	~400+
A3	Xeon 8160 CPU	2.1	24 (24)	AVX-512	1612.8	~119.2**

*Intel source
** wikichip.org

Code analysis AoS Vs CSoA

```
void move_aos ( pop_type *  const __RESTRICT__ nxt,
                const pop_type * const __RESTRICT__ prv )
{
 int i, j, k, pp, idx0, idx0_offset;

 profile_on ( __move_aos__ );

#pragma omp parallel for private( i, j, k, idx0, idx0_offset, pp )
 for ( i = 1; i <= NX; i++ ) {
  for ( j = 1; j <= NY; j++ ) {
   for ( k = 1; k <= NZ; k++ ) {

    idx0 = IDX (i, j, k);
    for ( pp = 0; pp < NPOP; pp++ ) {

     idx0_offset = idx0 + offset_idx[ pp ];
     nxt[ idx0 ].p[ pp ] = prv[ idx0_offset ].p[ pp ];

    }
   }
  }
 }
  profile_off ( __move_aos__ );
}
```

```
# define for_each_element_v(_k)   \
 _Pragma("unroll")      \
 _Pragma("vector aligned")      \
 for(_k = 0; _k < VL; _k++)
```

```
__INLINE__ void vpopcpy_nt ( poptype * const __RESTRICT__ _pp,
                             const poptype * const __RESTRICT__ _qq )
{
  int k_vl;
  for_each_element_v_nontemporal( k_vl )  _pp[ k_vl ] = _qq[ k_vl ];
}

void move_csoa ( pop_type_csoa * const __RESTRICT__ nxt,
                 const pop_type_csoa * const __RESTRICT__ prv )
 {
 int i, j, k, pp, vidx0, vidx0_offset;

 profile_on ( __move_csoa__ );

#pragma omp parallel for private( i, j, k, pp, vidx0, vidx0_offset )
 for( i = 1; i <= NX; i++ ){
    for( j = 1; j <= NY; j++ ){

     for (pp = 0; pp < NPOP; pp++) {

       for( k = 1; k <= NZOVL; k++ ){

         vidx0 = IDX_CLUSTER( i, j, k );
         vidx0_offset = vidx0 + offset_idx[ pp ];
         vpopcpy_nt( nxt->p[ pp ][ vidx0 ].c, prv->p[ pp ][vidx0_offset].c );
        }
      }
    }
 }
}profile_off (__move_csoa__);
```

Vector report analysis AoS

mpiicc -DAOS -DARCH -DARCH_SKL [...] -O3 -fp-model=precise -xCOMMON-AVX512 -D__VEC_WIDTH__=512 -D__INTEL -restrict -qopt-report-routine=move_aos -qopt-report-phase=vec -DH5Tarray_create_vers=2 -DH5Eget_auto_vers=2 -DH5Eset_auto_vers=2 -DH5Acreate_vers=2 -DH5Gcreate_vers=2 -DH5Dcreate_vers=2 -DH5Dopen_vers=2 -DH5Gopen_vers=2 -qopenmp -c lbe_performance.c -qopt-report=2

[...]

Begin optimization report for: move_aos(pop_type *const __restrict__, const pop_type *const __restrict__)

Report from: Vector optimizations [vec]

LOOP BEGIN at lbe_performance.c(56,3)
remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for\
details
remark #15346: vector dependence: assumed OUTPUT dependence between nxt->p[idx0][pp] (64:4) and nxt->p[idx0][pp] (64:4)

LOOP BEGIN at lbe_performance.c(57,5)
remark #15335: loop was not vectorized: vectorization possible but seems inefficient.

LOOP BEGIN at lbe_performance.c(58,7)
remark #15335: loop was not vectorized: vectorization possible but seems inefficient.
LOOP BEGIN at lbe_performance.c(61,2)
remark #15335: loop was not vectorized: vectorization possible but seems inefficient.
LOOP END
LOOP END
LOOP END
LOOP END

Vector report analysis CSoA

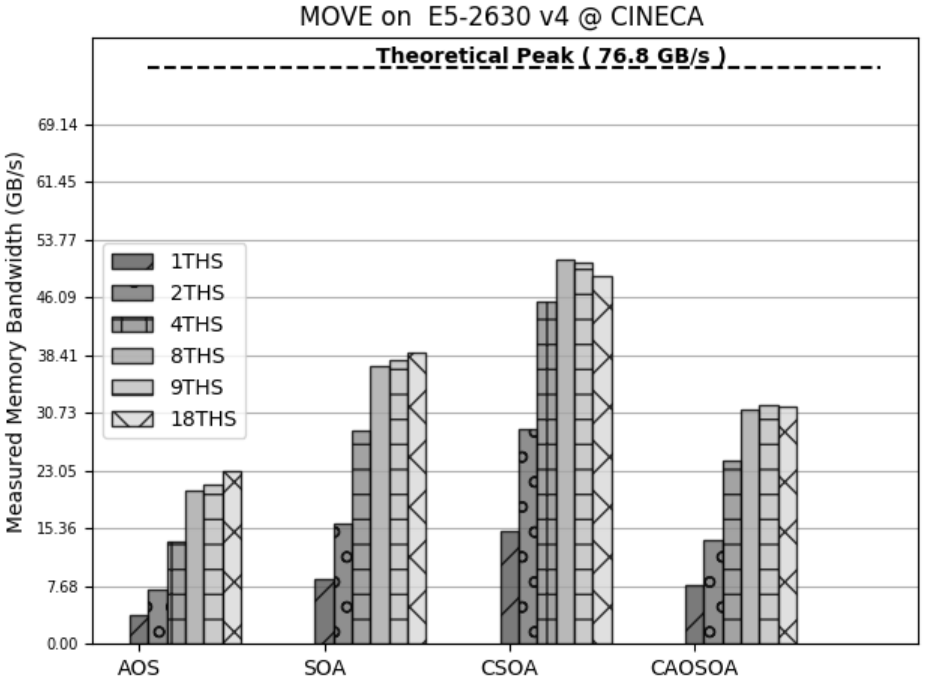
```
Begin optimization report for: move_csoa(pop_type_csoa *const __restrict__, const pop_type_csoa *const __restrict__)
LOOP BEGIN at lbe_performance.c(1023,3)
  remark #15542: loop was not vectorized: inner loop was already vectorized
  LOOP BEGIN at lbe_performance.c(1024,7)
    remark #15542: loop was not vectorized: inner loop was already vectorized
    LOOP BEGIN at lbe_performance.c(1026,2)
      remark #15542: loop was not vectorized: inner loop was already vectorized
      LOOP BEGIN at lbe_performance.c(1028,4)
        remark #15542: loop was not vectorized: inner loop was already vectorized

        LOOP BEGIN at lbe_performance.c(993,3) inlined into lbe_performance.c(1033,6)
          remark #15388: vectorization support: reference _pp[k_vl] has aligned access   [ lbe_performance.c(994,5) ]
          remark #15388: vectorization support: reference _qq[k_vl] has aligned access   [ lbe_performance.c(994,19) ]
          remark #15412: vectorization support: streaming store was generated for _pp[k_vl]   [ lbe_performance.c(994,5) ]
          remark #15305: vectorization support: vector length 8
          remark #15427: loop was completely unrolled
          remark #15300: LOOP WAS VECTORIZED
          remark #15448: unmasked aligned unit stride loads: 1
          remark #15449: unmasked aligned unit stride stores: 1
          remark #15467: unmasked aligned streaming stores: 1
          remark #15475: --- begin vector cost summary ---
          remark #15476: scalar cost: 5
          remark #15477: vector cost: 0.370
          remark #15478: estimated potential speedup: 13.330
          remark #15488: --- end vector cost summary ---
        LOOP END
      LOOP END

      LOOP BEGIN at lbe_performance.c(1028,4)
      <Remainder>
      LOOP END
    LOOP END
  LOOP END
LOOP END
```

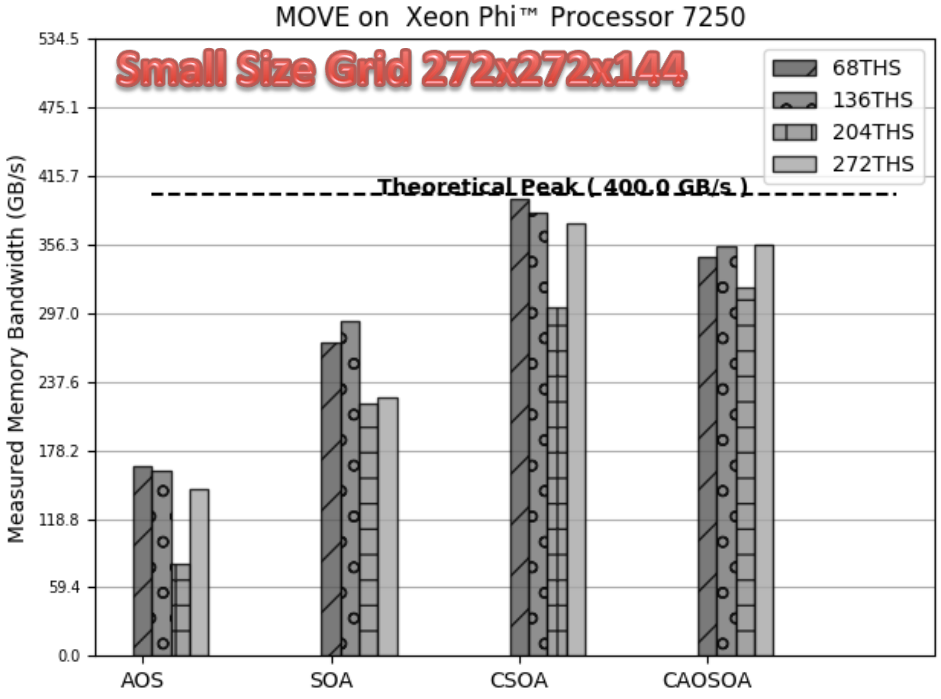
```
void move_collide_fused_csoa ( pop_type_csoa * const __RESTRICT__ nxt,
                                const pop_type_csoa * const __RESTRICT__ prv,  double tau, double omega ) {
 profile_on ( __move_collide_fused_csoa__ );
#pragma omp parallel
 {
  vpoptype vprod, vprod2, vr1, vsq, vux, vuy, vuz, vinvr1, vrho, vfeq, vpop_temp[NPOP];
  vpoptype vfrcex, vfrcey, vfrcez, vu, vv, vw, vamp;
  int i, j, k, p, k_vl;
  size_t vidx0, vidx0_offset;
#pragma omp for
  for (i = 1; i <= NX; i++) {
    for (j = 1; j <= NY; j++) {
       for (k = 1; k <= NZOVL; k++) {
         vidx0 = IDX_CLUSTER (i, j, k);
         for_each_element_v(k_vl){
           vu.c[k_vl] = 0.0;       vv.c[k_vl] = 0.0; vw.c[k_vl] = 0.0;
         }
         for_each_pop(p){
           vidx0_offset = vidx0 + offset_idx[ p ];
           vpopcpy( vpop_temp[ p ].c, prv->p[ p ][ vidx0_offset ].c );
         }
         /* Hydrovar + compute velocity from equili*/
         for_each_element_v(k_vl) vrho.c[ k_vl ] = 0.0;
         }
          vsum( vrho.c, vpop_temp[p].c );
         for_each_element_v(k_vl){
           vinvr1.c[k_vl] = 1.e0 / vrho.c[ k_vl ];
           vr1.c[k_vl] = vrho.c[ k_vl ];
         }
```

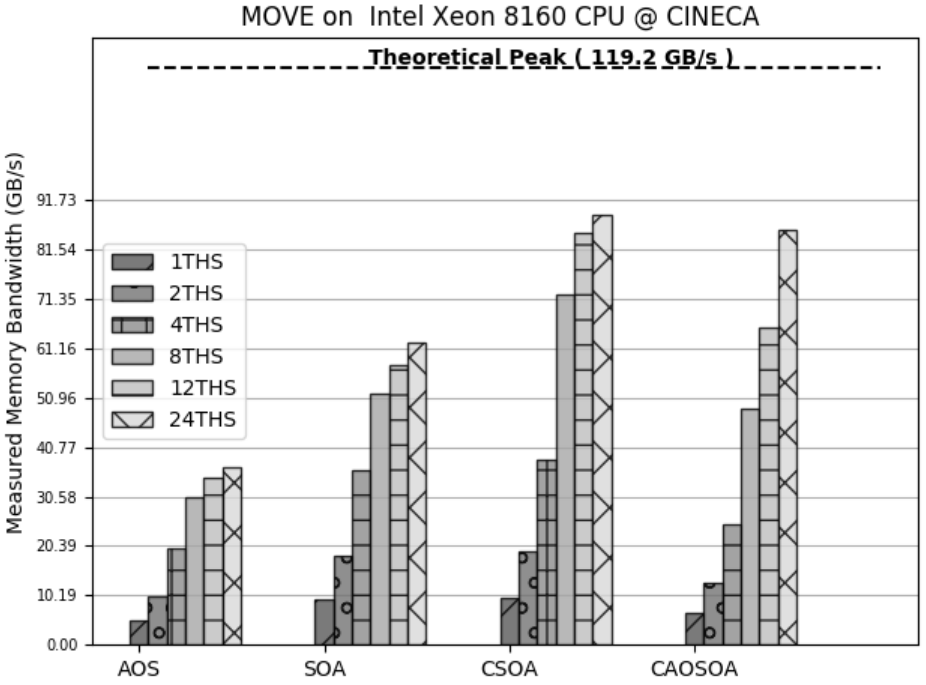
```
          for_each_pop(p){
           for_each_element_v(k_vl){
             vu.c[k_vl] += vpop_temp[p].c[k_vl] * vinvr1.c[k_vl] * vcx[p].c[k_vl];
             vv.c[k_vl] += vpop_temp[p].c[k_vl] * vinvr1.c[k_vl] * vcy[p].c[k_vl];
             vw.c[k_vl] += vpop_temp[p].c[k_vl] * vinvr1.c[k_vl] * vcz[p].c[k_vl];
           }
          }

          for_each_element_v(k_vl){
           vfrcex.c[k_vl] = vrho.c[ k_vl ] * vaccel_gravity_x.c[k_vl];
           vfrcey.c[k_vl] = vrho.c[ k_vl ] * vaccel_gravity_y.c[k_vl];
           vfrcez.c[k_vl] = vrho.c[ k_vl ] * vaccel_gravity_z.c[k_vl];
           vux.c[k_vl] = vu.c[k_vl] + vtau.c[k_vl] * vfrcex.c[k_vl] * vinvr1.c[k_vl];
           vuy.c[k_vl] = vv.c[k_vl] + vtau.c[k_vl] * vfrcey.c[k_vl] * vinvr1.c[k_vl];
           vuz.c[k_vl] = vw.c[k_vl] + vtau.c[k_vl] * vfrcez.c[k_vl] * vinvr1.c[k_vl];
          }

          for_each_pop(p){
           for_each_element_v(k_vl){
            vprod.c[k_vl] = vcx[p].c[k_vl] * vux.c[k_vl] + vcy[p].c[k_vl] * vuy.c[k_vl] + vcz[p].c[k_vl] * vuz.c[k_vl];
             vsq.c[k_vl] = VONEANDHALF.c[k_vl] * (vux.c[k_vl] * vux.c[k_vl] + vuy.c[k_vl] * vuy.c[k_vl] + vuz.c[k_vl] * vuz.c[k_vl]);
             vprod2.c[k_vl] = vprod.c[k_vl] * vprod.c[k_vl];
             vfeq.c[k_vl] = vr1.c[k_vl] * vww[p].c[k_vl] * (VONE.c[k_vl] + VTHREE.c[k_vl] * vprod.c[k_vl]  + vfac.c[k_vl] * vprod2.c[k_vl] - vsq.c[k_vl] );

             vpop_temp[ p ].c[ k_vl ] =  vpop_temp[ p ].c[ k_vl ] * (VONE.c[k_vl] - vomega.c[ k_vl ] ) + ( vomega.c[ k_vl ] * vfeq.c[ k_vl ] );
           } // end k_vl loop
           vpopcpy_nt( nxt->p[ p ][ vidx0 ].c, vpop_temp[ p ].c);
          } // end p loop
         } // end k loop
        } // end j loop
       } // end i loop
      } // end omp parallel loop
```

Measured Memory Bandwidth for *move* (propagate) routine

- CAoS data distribution deliver high speed for the *move* kernel
- On KNL configured in **flat/quadrant** mode almost the peak performance is achieved

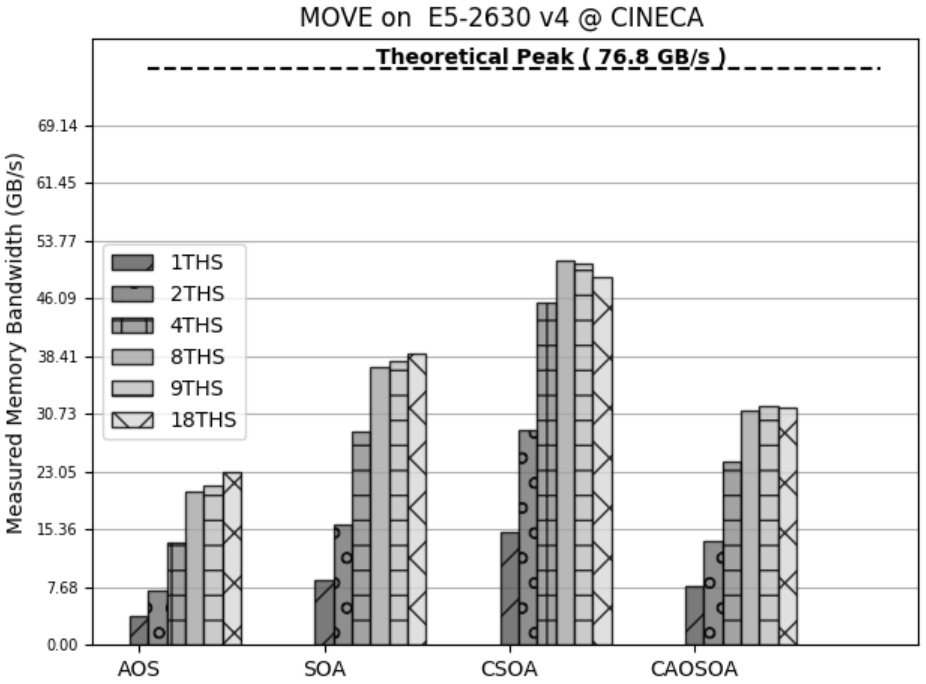


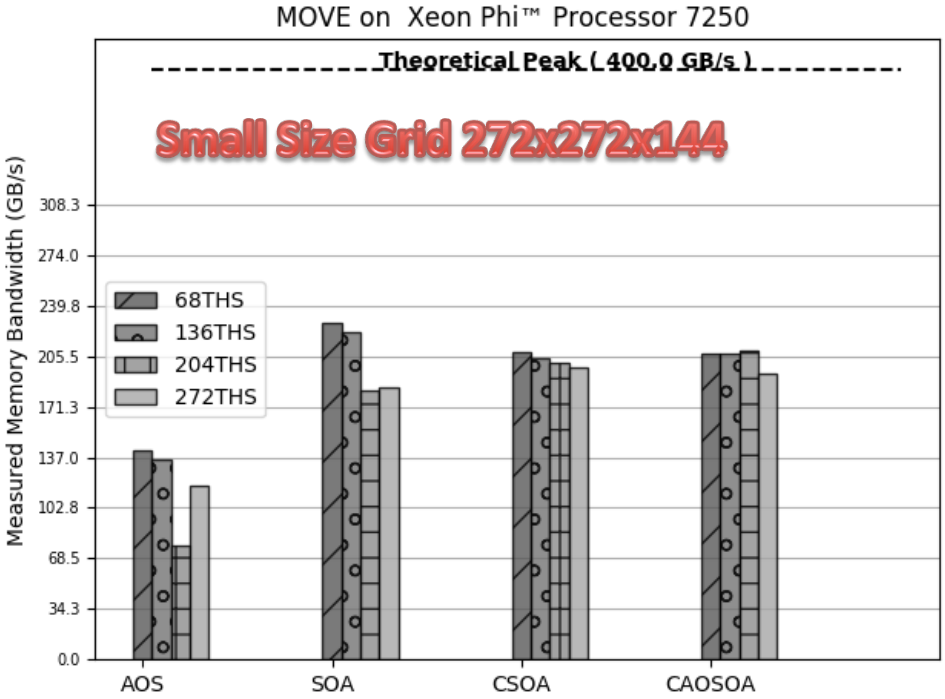


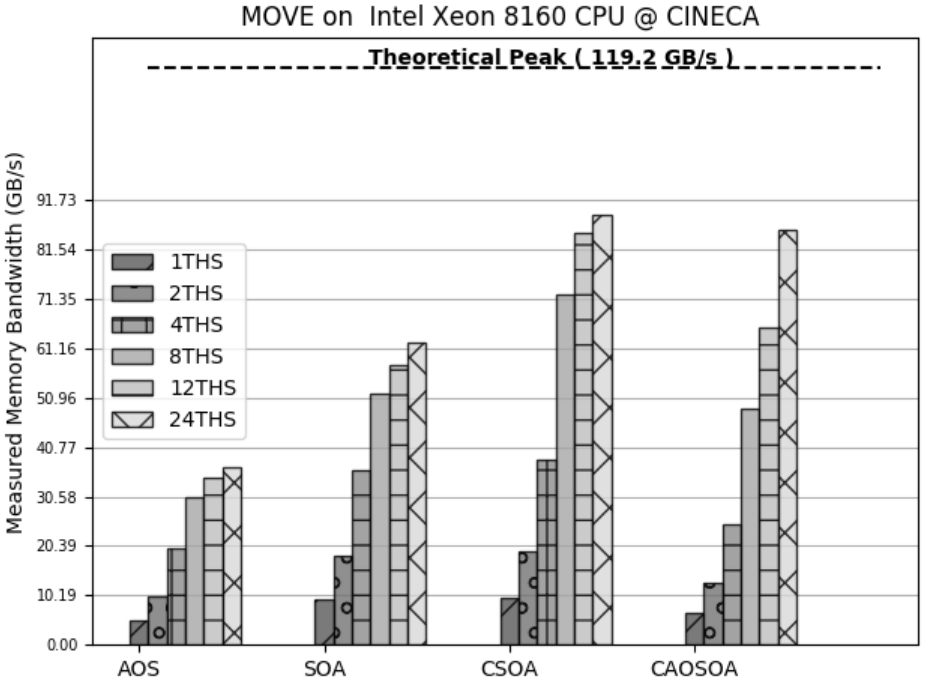


Measured Memory Bandwidth for *move* (propagate) routine

- CAoS data distribution deliver high speed for the *move* kernel
- On KNL configured in **cache/quadrant** mode almost the peak performance is achieved

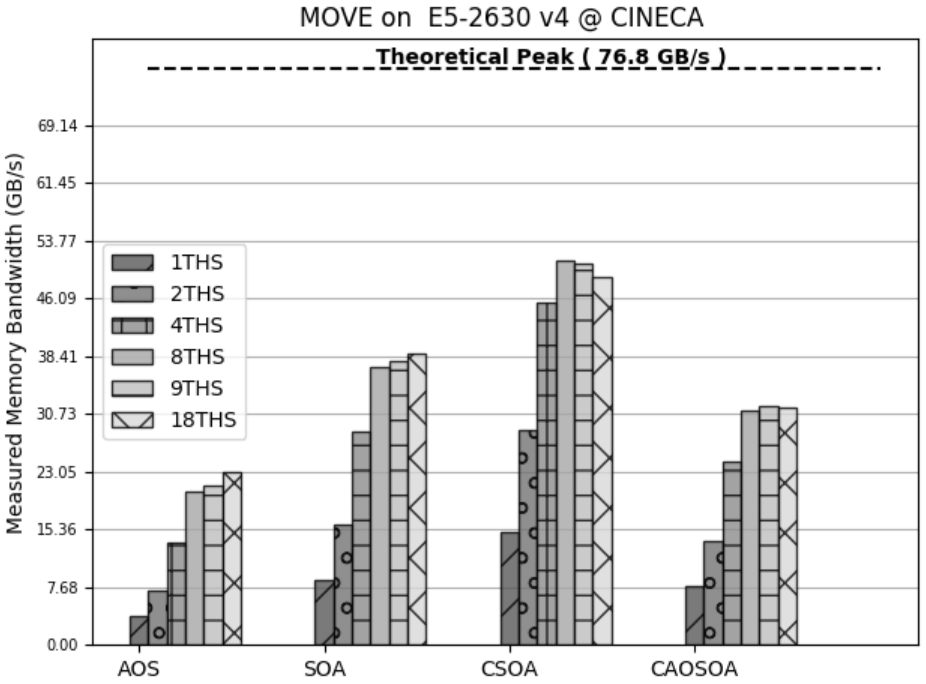


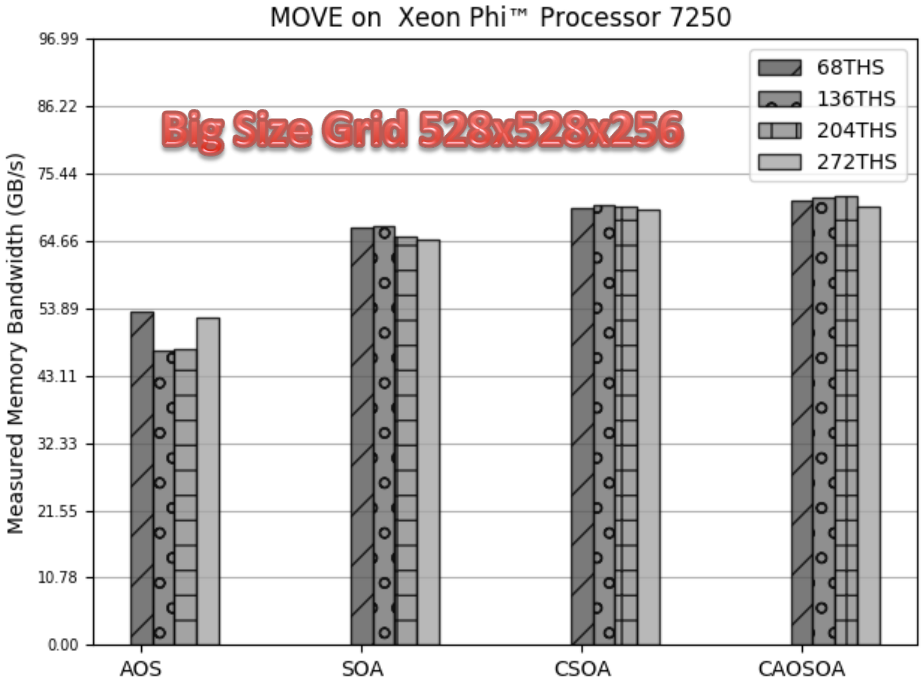


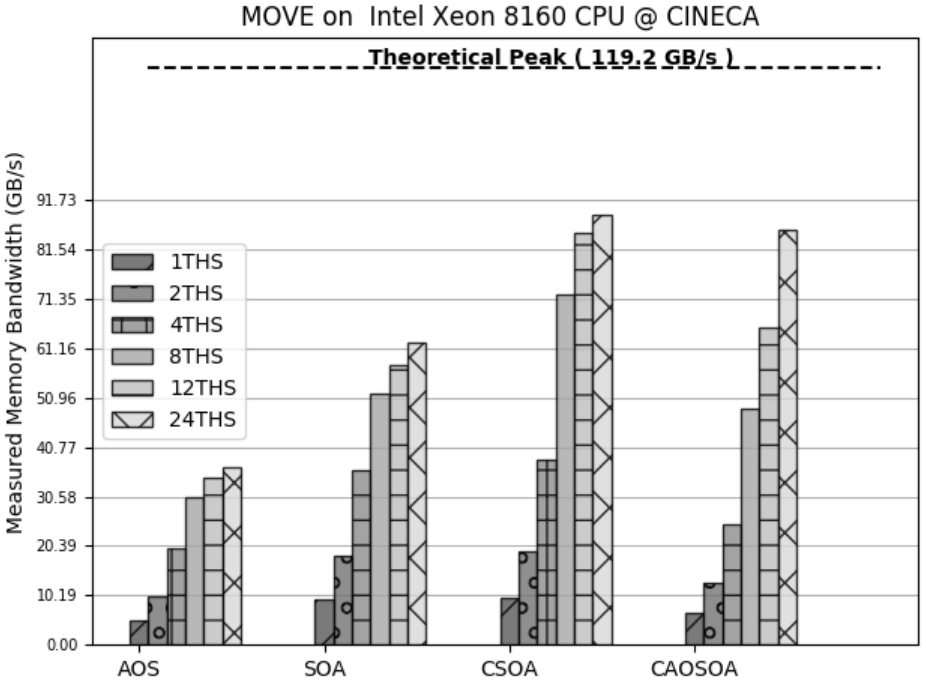


Measured Memory Bandwidth for *move* (propagate) routine

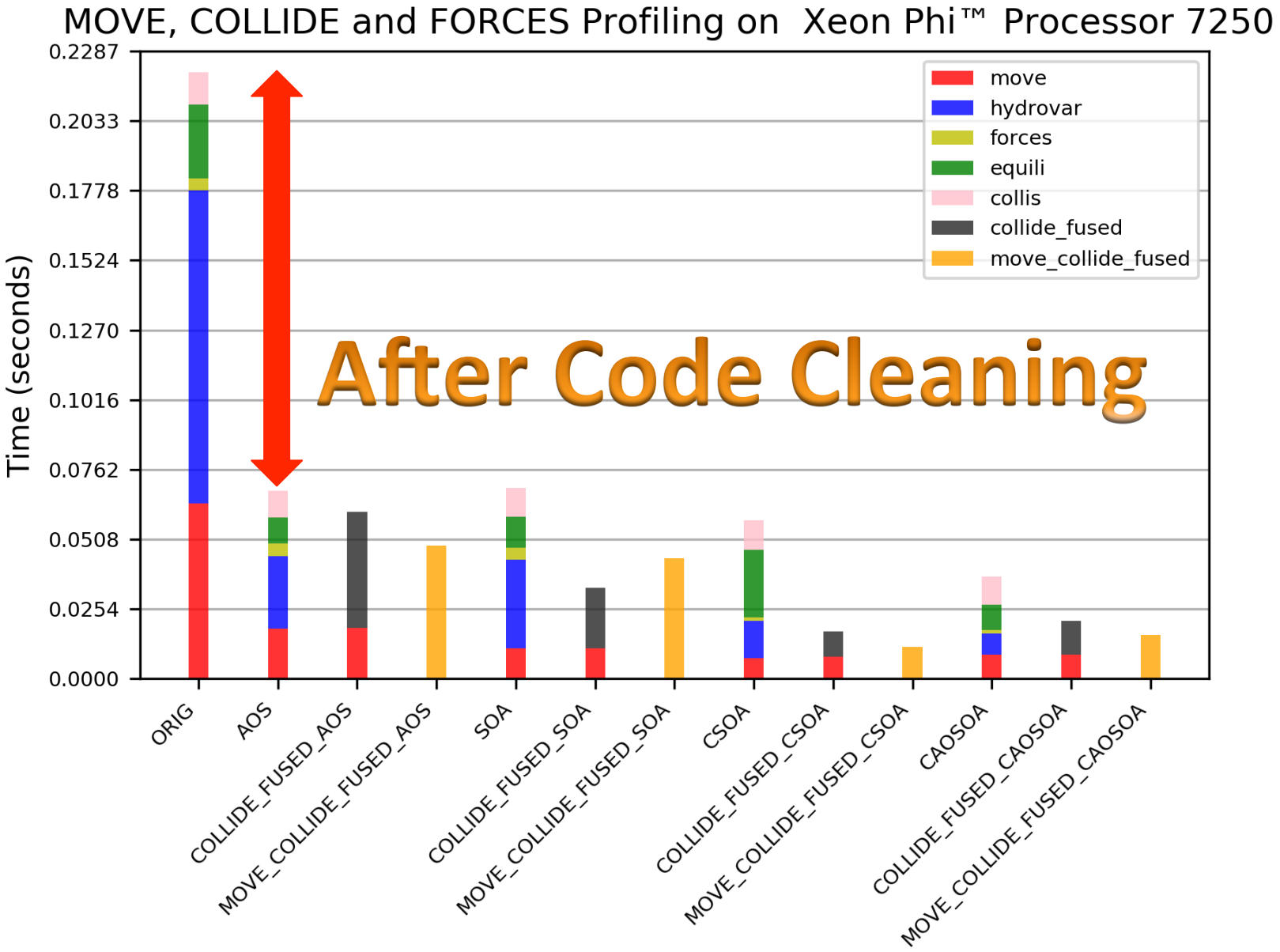
- CAoS data distribution deliver high speed for the *move* kernel
- On KNL configured in **cache/quadrant** mode almost the peak performance is achieved



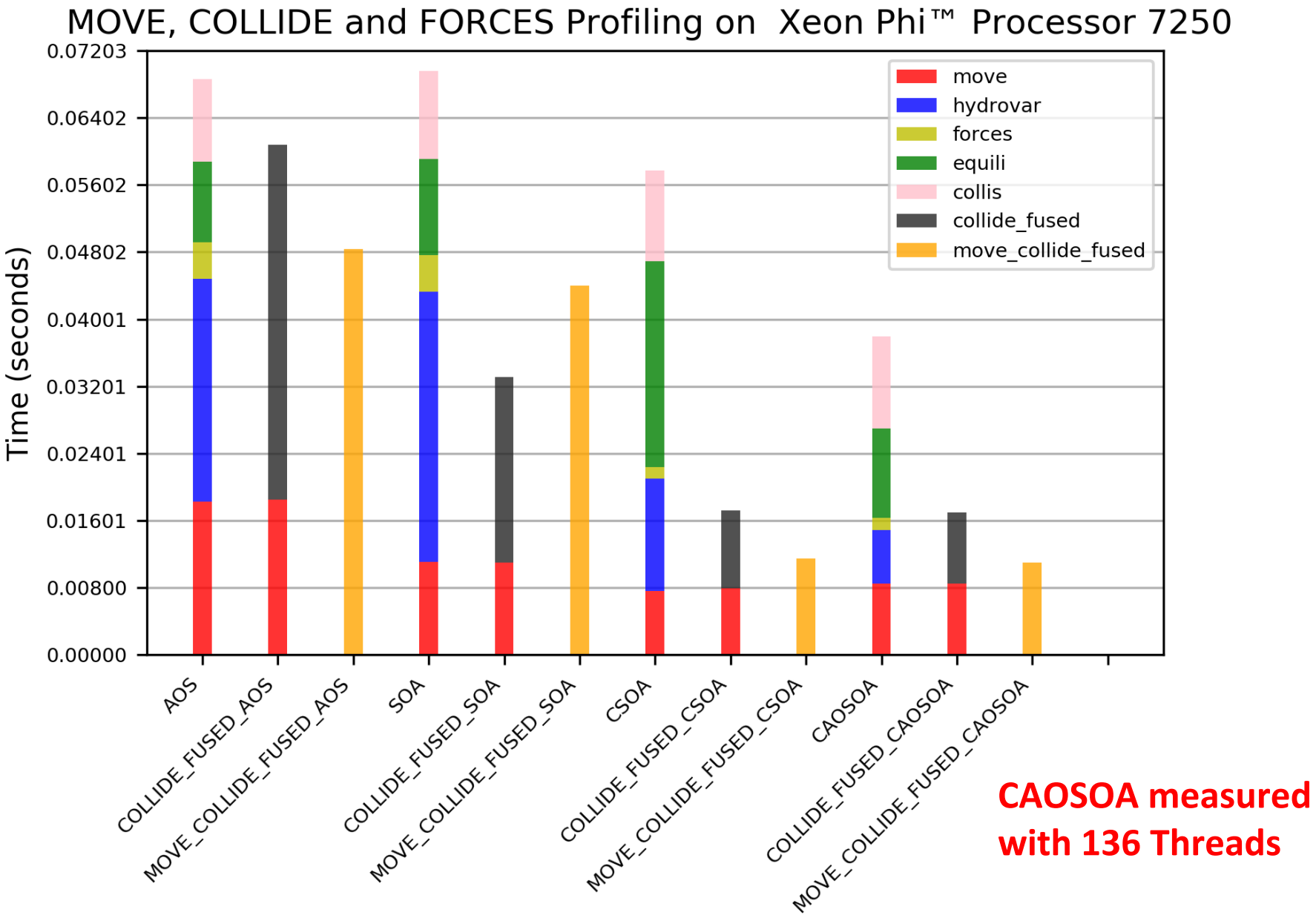




Small Size Grid 272x272x144

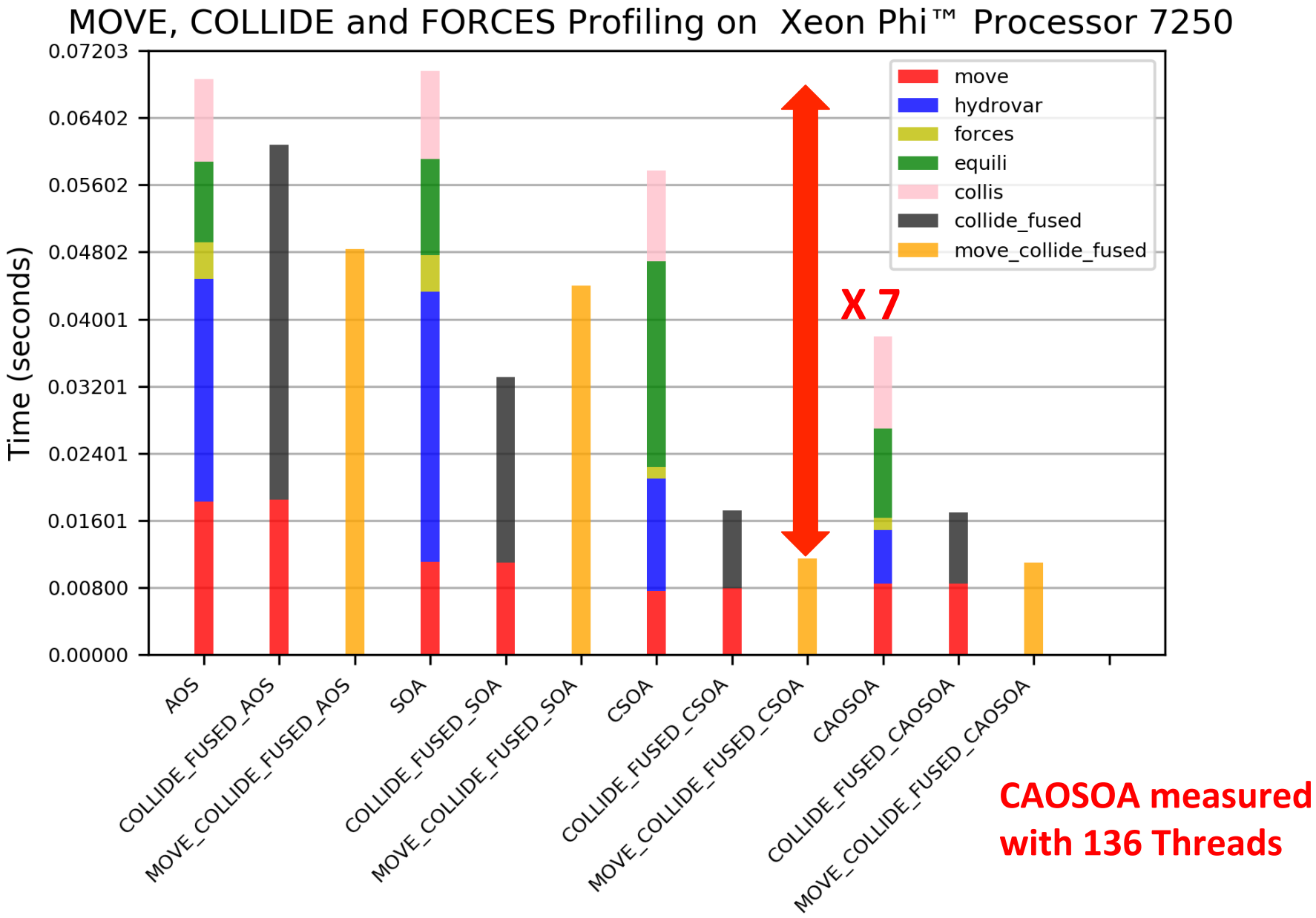


Small Size Grid 272x272x144



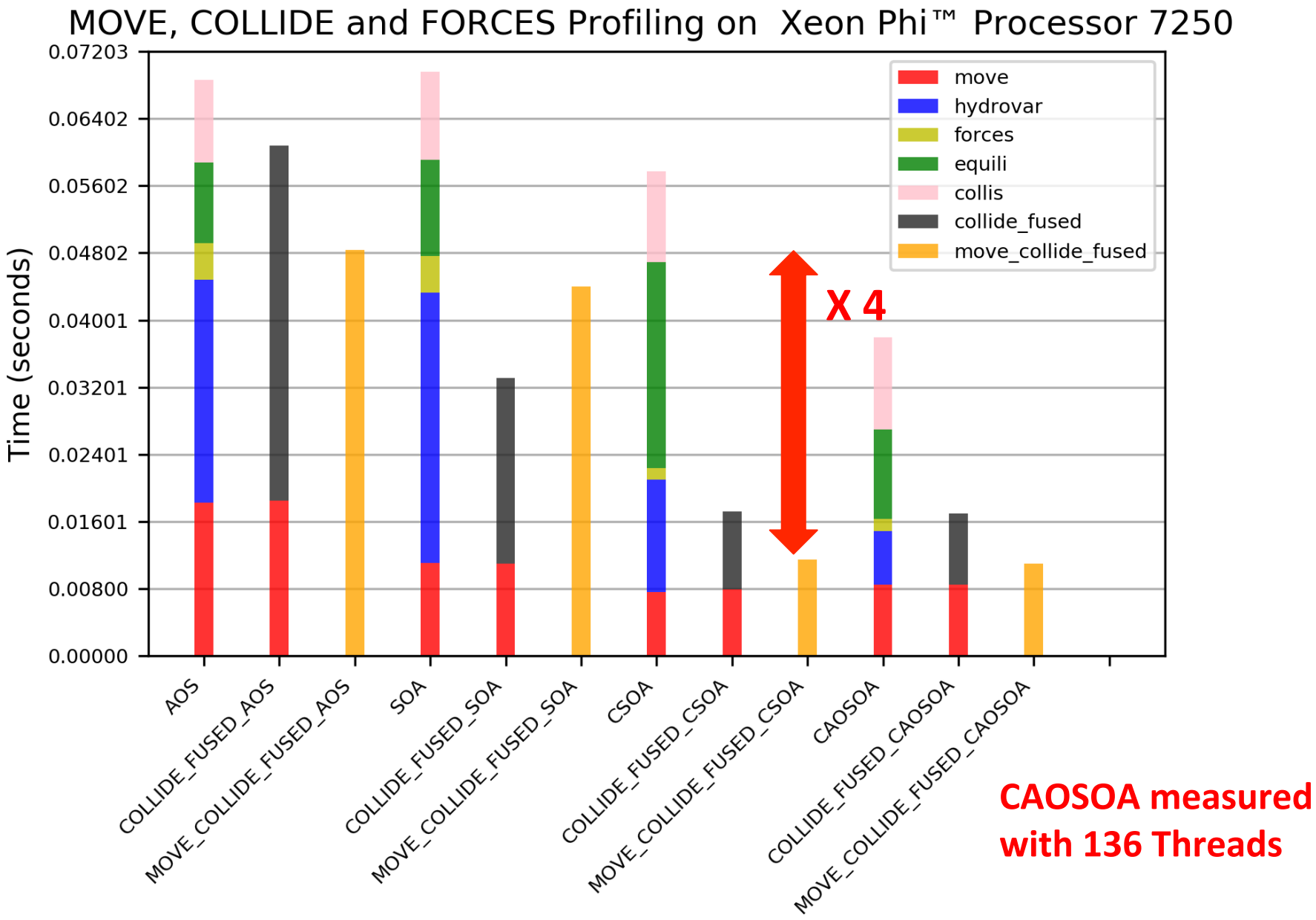
CAOSOA measured with 136 Threads

Small Size Grid 272x272x144



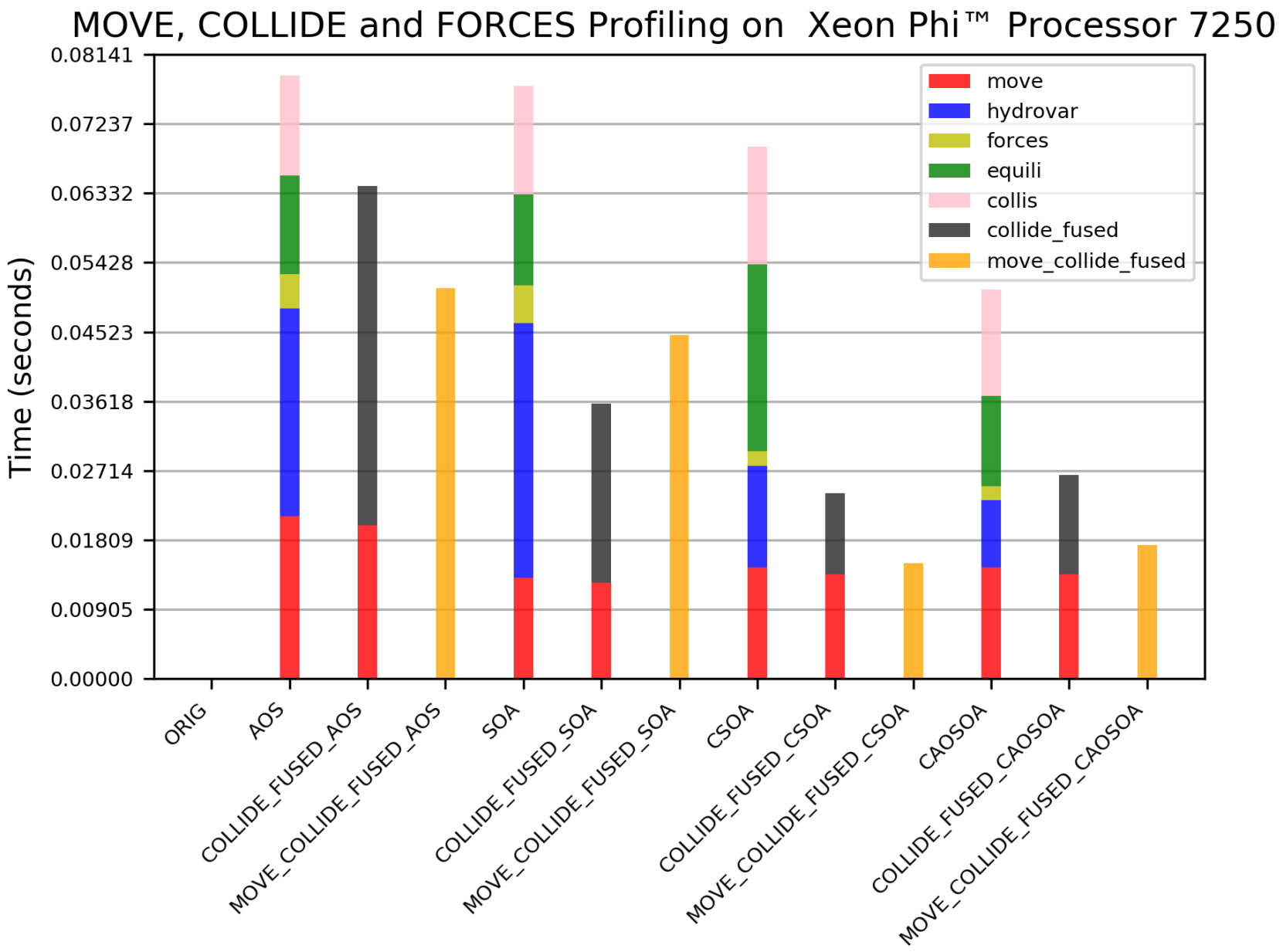
CAOSOA measured with 136 Threads

Small Size Grid 272x272x144

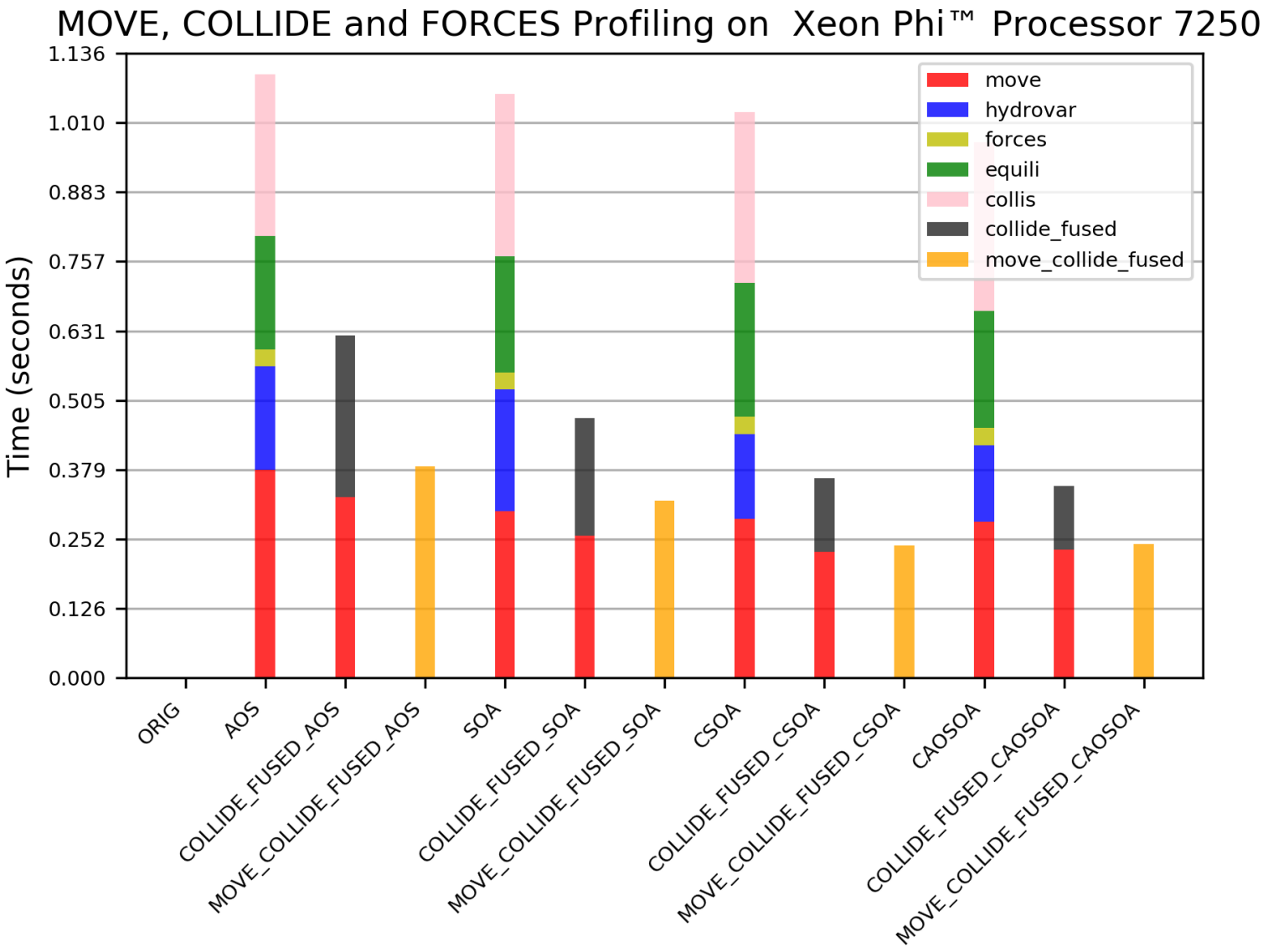


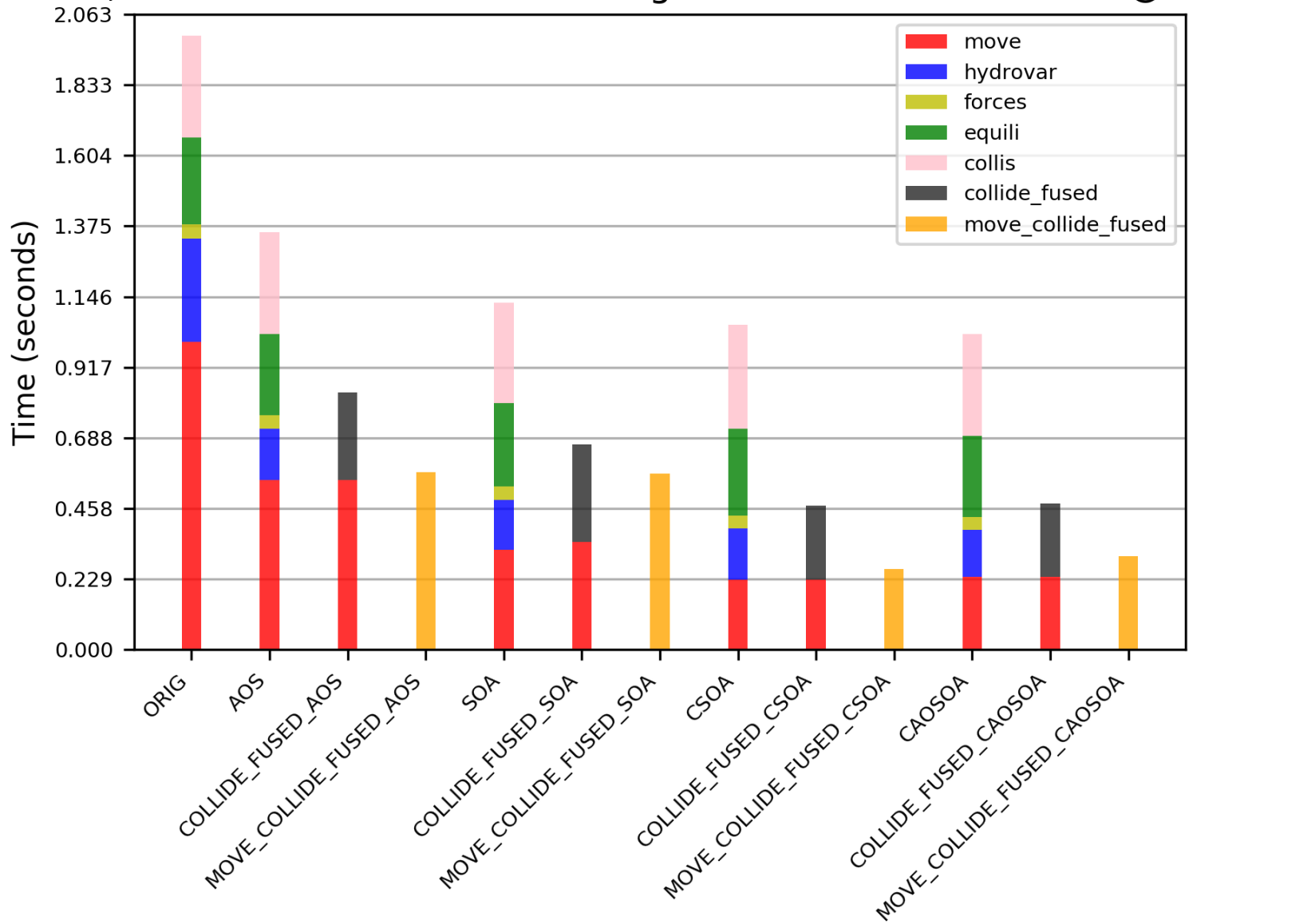
CAOSOA measured with 136 Threads

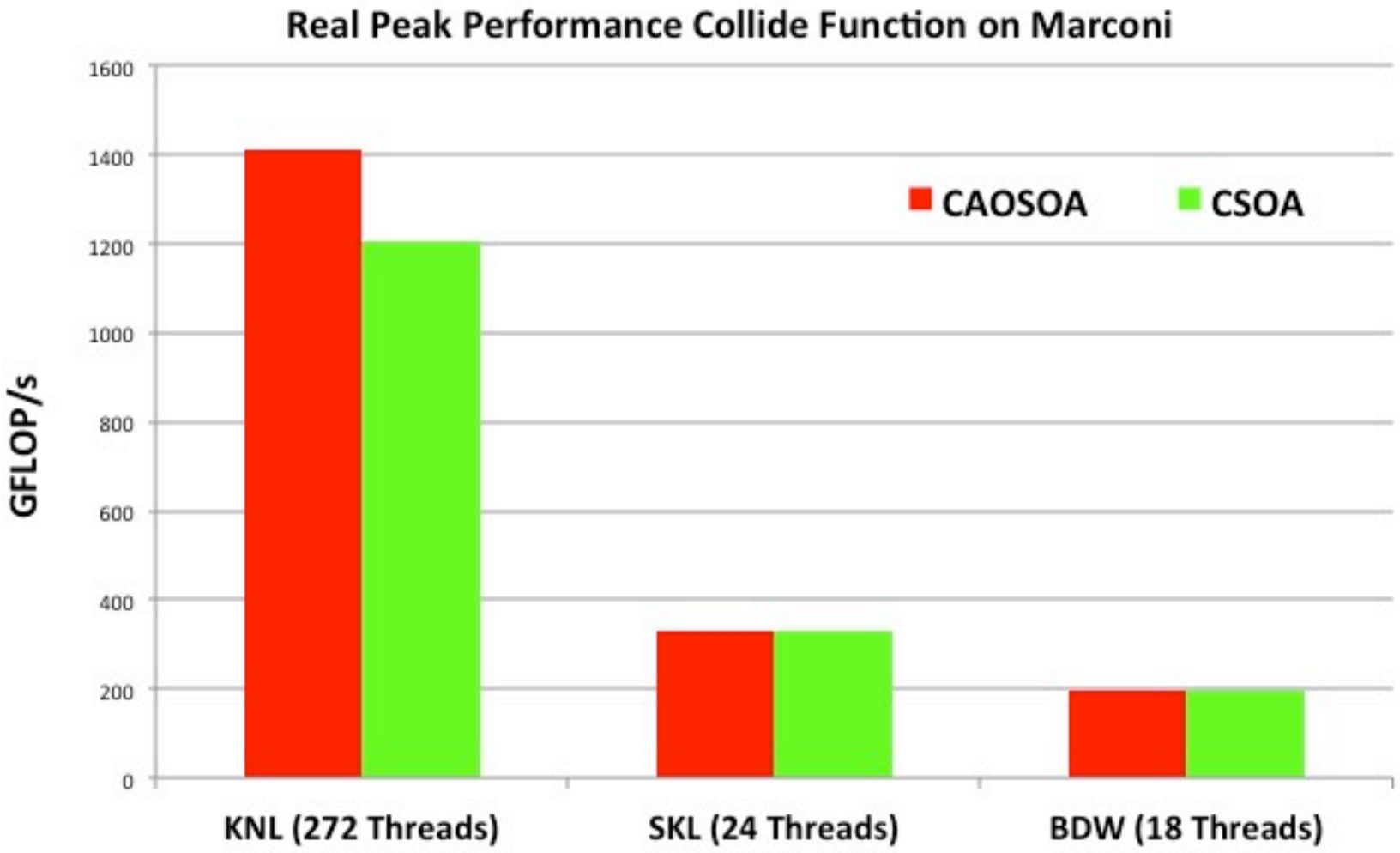
Small Size Grid 272x272x144



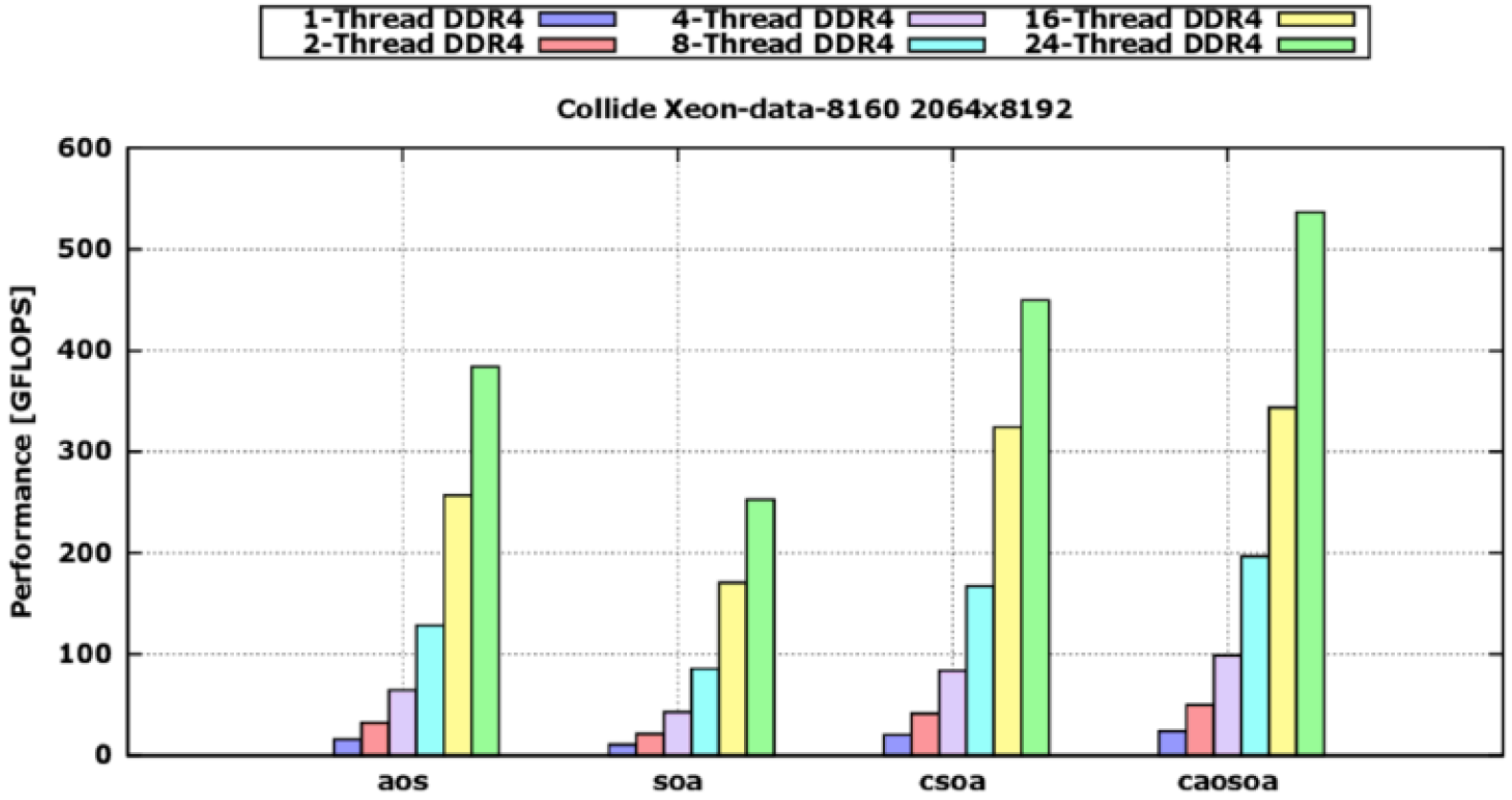
Big Size Grid 528x528x256




MOVE, COLLIDE and FORCES Profiling on Intel Xeon 8160 CPU @ CINECA
Time (seconds)
2.063
1.833
1.604
1.375
1.146
0.917
0.688
0.458
0.229
0.000
move
hydrovar
forces
equili
collis
collide_fused
move_collide_fused
ORIG
AOS
COLLIDE_FUSED_AOS
MOVE_COLLIDE_FUSED_AOS
SOA
COLLIDE_FUSED_SOA
MOVE_COLLIDE_FUSED_SOA
CSOA
COLLIDE_FUSED_CSOA
MOVE_COLLIDE_FUSED_CSOA
CAOSOA
COLLIDE_FUSED_CAOSOA
MOVE_COLLIDE_FUSED_CAOSOA

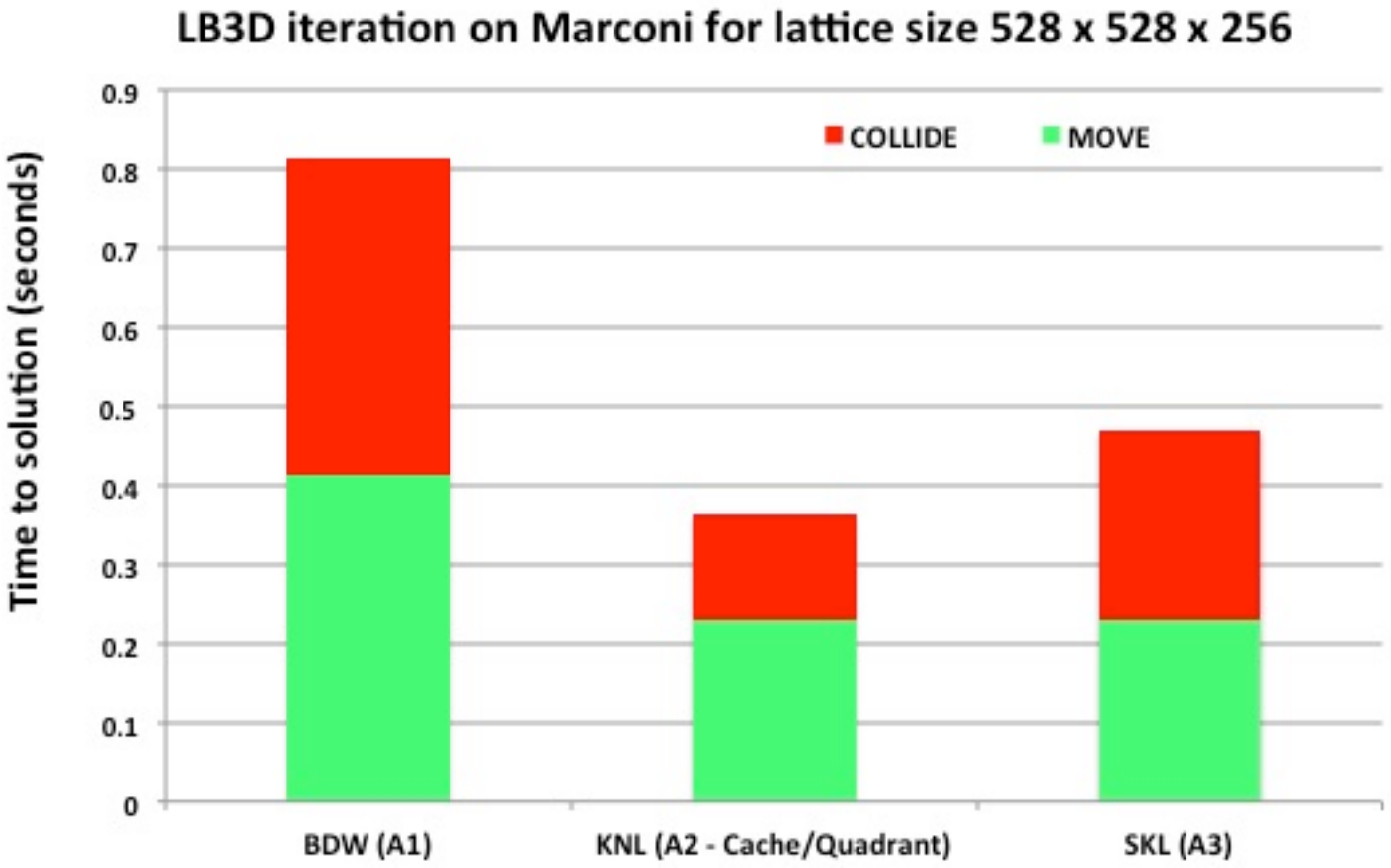

Real Peak Performance Collide Function on Marconi
GFLOP/s
1600
1400
1200
1000
800
600
400
200
0
CAOSOA
CSOA
KNL (272 Threads)
SKL (24 Threads)
BDW (18 Threads)

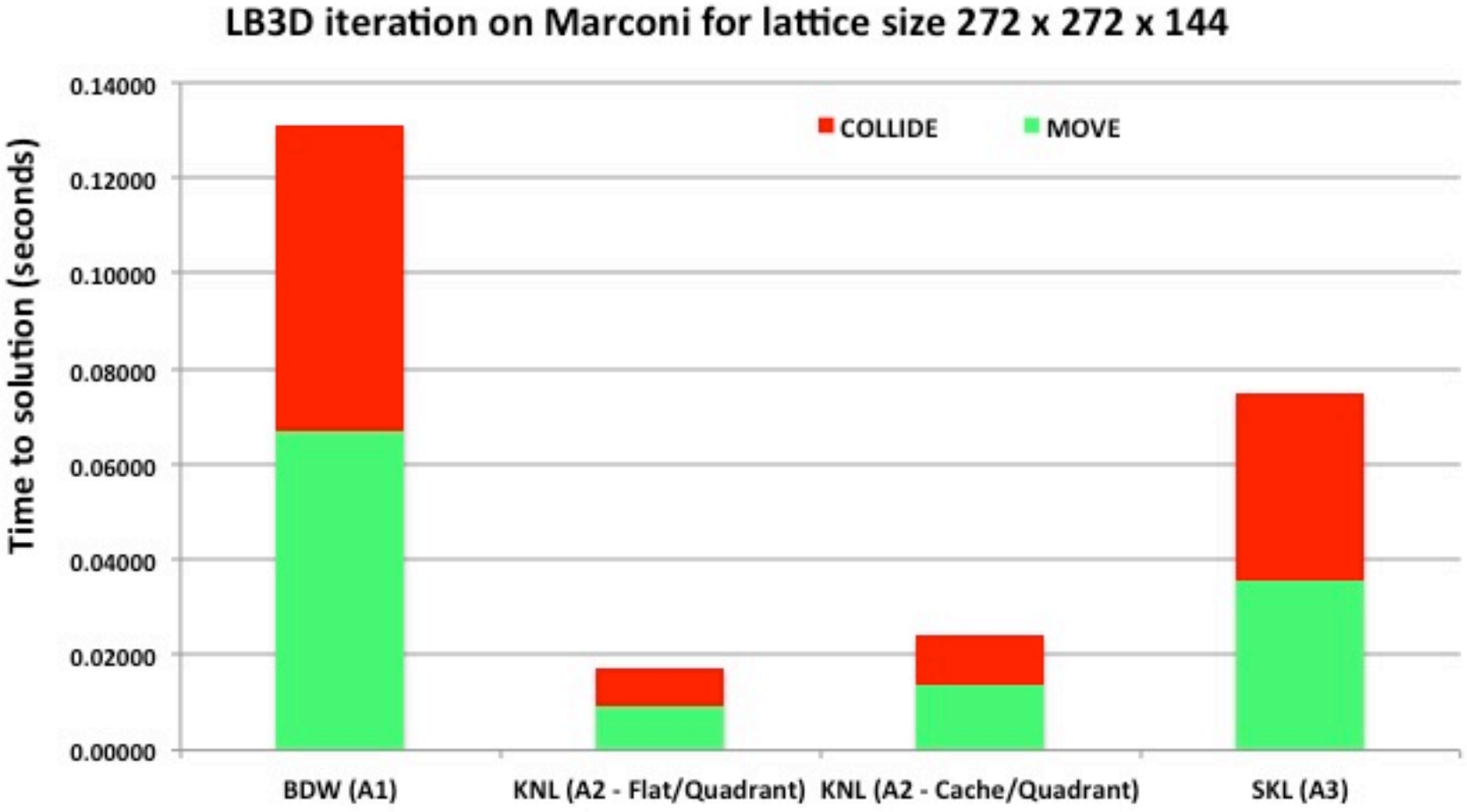
CAOSOA on a D2Q37 LBM



collide ≈ 530 GFlops. ≈ 35% of raw peak.

Optimization of lattice Boltzmann simulations on heterogeneous computers. In INTERNATIONAL JOURNAL OF HIGH PERFORMANCE COMPUTING APPLICATIONS - ISSN:1094-3420 pp.1-16. 2017 doi:10.1177/1094342017703771


LB3D iteration on Marconi for lattice size 528 x 528 x 256
Time to solution (seconds)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
COLLIDE
MOVE
BDW (A1)
KNL (A2 - Cache/Quadrant)
SKL (A3)


LB3D iteration on Marconi for lattice size 272 x 272 x 144
Time to solution (seconds)
0.14000
0.12000
0.10000
0.08000
0.06000
0.04000
0.02000
0.00000
COLLIDE
MOVE
BDW (A1)
KNL (A2 - Flat/Quadrant)
KNL (A2 - Cache/Quadrant)
SKL (A3)

Thread Affinity Setting

KNL Job Script

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH --constraint=flat,quad
#SBATCH --time=00:30:00
#SBATCH --mem=83GB
#SBATCH --output job.out
#SBATCH --error job.err
#SBATCH --partition knl_usr_prod
#SBATCH --account=ICT18_A2
#SBATCH -D .

ulimit -s unlimited
export OMP_NUM_THREADS=68
export MPI_PES=1
export KMP_AFFINITY=compact
export KMP_HW_SUBSET=1t
export I_MPI_PIN_DOMAIN=socket
export I_MPI_DEBUG=4

mpirun -np ${MPI_PES} numactl -m 1 ./lbe3d
```

SKL Job Script

```
!/bin/bash
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:30:00
#SBATCH --mem=177GB
#SBATCH --output job.out
#SBATCH --error job.err
#SBATCH --partition skl_usr_dbg
#SBATCH --account=ICT18_CMSP

ulimit -s unlimited
export OMP_NUM_THREADS=24
export MPI_PES=1
export KMP_AFFINITY=compact
export KMP_HW_SUBSET=1t
export I_MPI_PIN_DOMAIN=socket
export I_MPI_DEBUG=4

mpirun -np ${MPI_PES} ./lbe3d
```

Conclusions

- I presented an experience of code refactoring and optimization on a Lattice Boltzmann based real application
- The optimization had the objective to enhance compiler vectorization
  - implementation of new data structures for the lattice representation
  - code optimization based on a deep analysis of the compiler vectorization report
- On Marconi A2 we achieved very high-memory and good (50%) real peak performance
- On Marconi A2 (KNL) up to a factor of x4 is achieved comparing the most optimized version of the code across different data structures
- On Marconi A2 (KNL) a result of about x20 is achieved comparing the best optimized version with the original version of the code

Community Engagement

- This experience is a real example of best practice for code optimization
- Few rules are inescapable to enhance compiler automatic vectorization:
  - writing clean and simple code
  - data alignment and given patterns of memory access
  - code developers are forced to think as the compiler works
- These rules should be introduce in all educational programs aimed to build tomorrow's computational scientists and scientific codes developers
- There are numbers of scientific codes out there consuming massive amount of computational resources that have not been implemented following the basic rules to achieve high-performance on modern computer architecture