



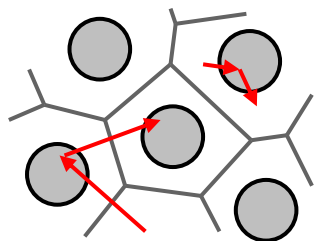
# A MEASURE FOR DATA LOCALITY

How re-ordering of loops might or might not help for an improved performance of block-sparse matrix-matrix multiplications on multi-level cache processor architectures

2018-04-24 | PAUL F. BAUMEISTER

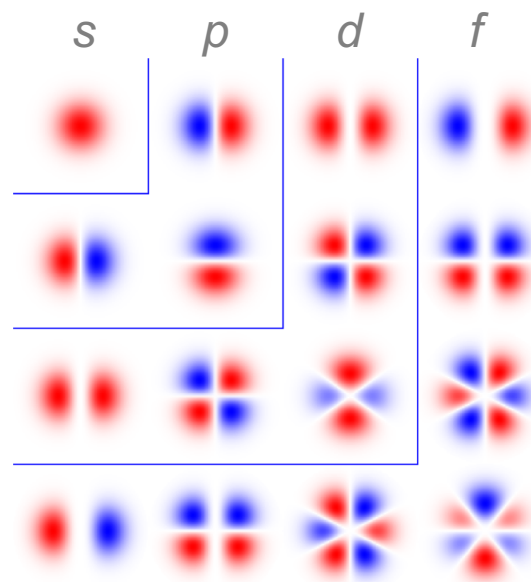
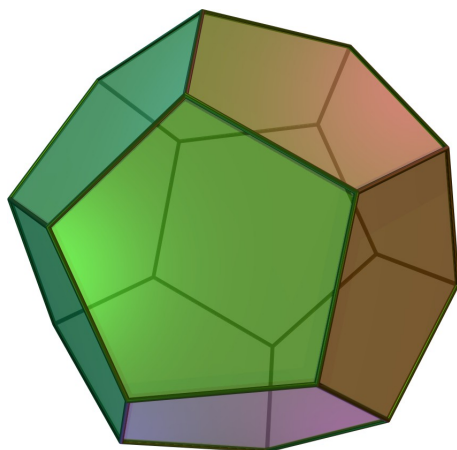
# PROBLEM: MULTIPLE SCATTERING

Core solver of KKRnano inverts large block-sparse matrices by tfQMR iterations

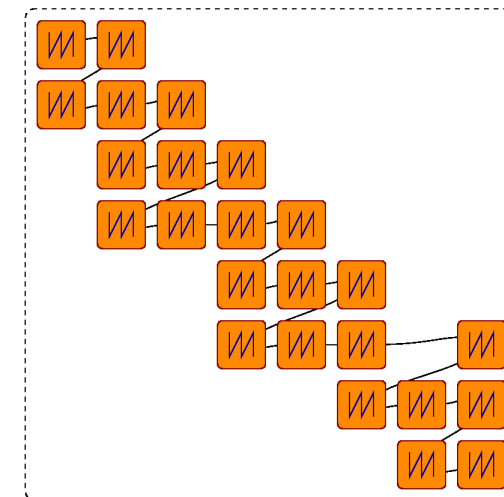


KKR formalism  
leads to multiple  
scattering path  
operator

All quantities are  
described on  
radial grids  
inside Voronoi  
polyhedra



Radial exponential grids  
plus an expansion in  $b=16$   
spherical harmonics

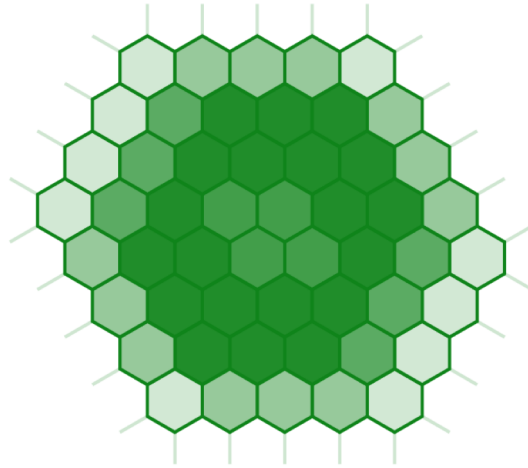
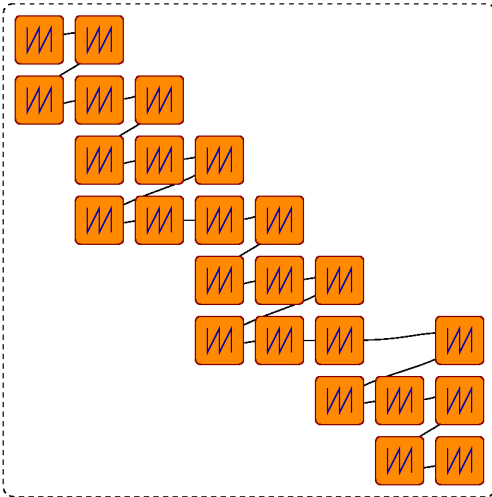


Selected columns of  
the inverse of a block-  
sparse complex  
matrix are sought for

# PROBLEM: SPARSE MATRIX INVERSION

Solve  $A \cdot X == 1$  for large block-sparse matrices with small block sizes

The block-sparse matrix  $A$  has 13 to 15 non-zero bands  
→ neighbor atoms



The solution is a block-sparse matrix  $X$  that is only non-zero inside a given truncation radius around the sources (red)

The performance depends crucially on the efficiency of *ZGEMM* for small block sizes and the amount of data re-use, as the Arithmetic Intensity is low

$$AI = \frac{\text{\#Flop}}{\text{\#Byte}} = \frac{8 b^3}{4 \cdot 16 b^2}$$

So,  $AI = 2$  Flop/Byte for block size  $b=16$   
The critical AI for current many-core processors is about 8 Flop/Byte

# PROBLEM: REDUCE BANDWIDTH REQUIREMENTS

Maximize the amount of data that can be re-used once fetched from main/device memory

Regarding dense matrix-matrix multiplications,  
the AI grows linear with the dimension  $n$   
→ cache-blocking / tiling solves the problem

$$Y'_{ij} = Y_{ij} + A_{ik} \times X_{kj}$$

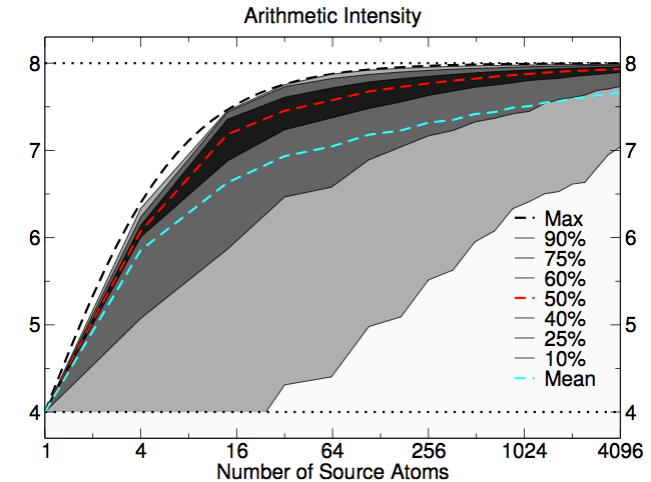
Here, the number of non-zero blocks per row is limited (13 to 15) so we can keep the accumulator matrix (4 kiByte) in cache (CPU) or registers (GPU) so that there is only a final write to memory

→ AI = 3.85 to 3.87 for  $b=16$

Can we re-use elements of  $A$ ?

Yes, if index  $j$  runs over more than 1 element

→ Always solve for several source atoms at a time



Can we also re-use elements of  $X$ ?

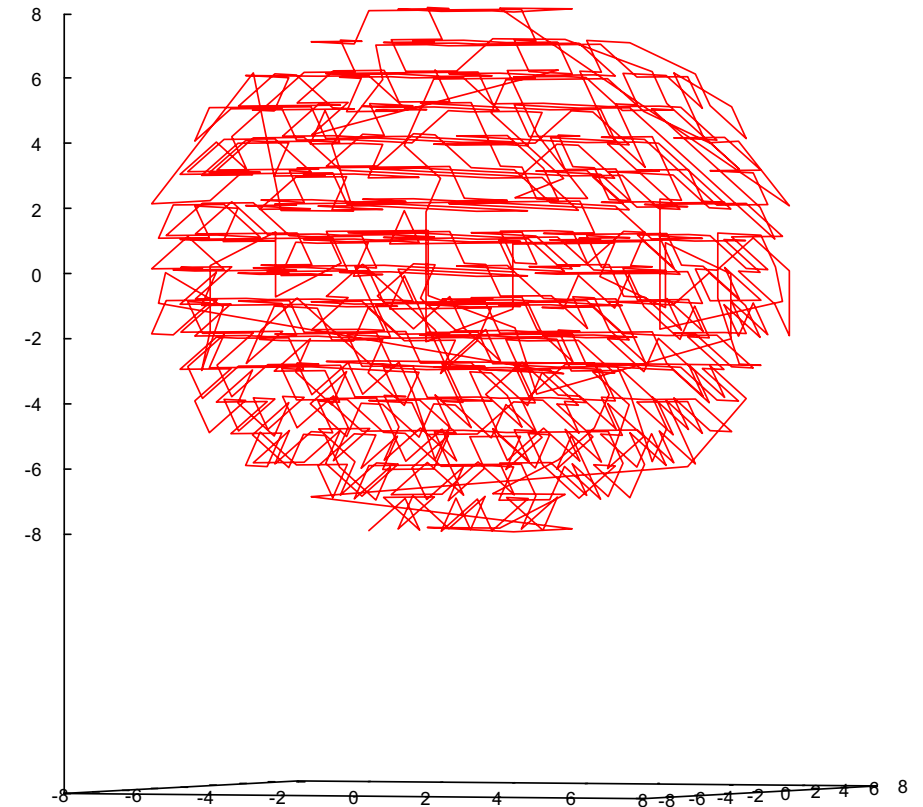
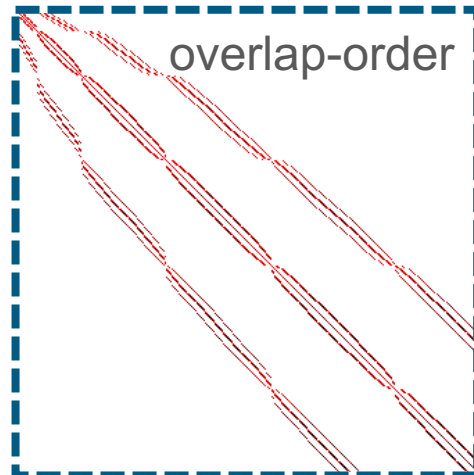
Yes, with a smart loop order for the elements  $ij$

→ Find a smart order with high temporal data locality

# OVERLAP ORDERING

## A simple definition for 1 source atom per MPI process

Distance ordering only decides after the spatial distance between source and target cell. Works only for 1 source atom. Overlap ordering maximizes the overlap of the target entries  $X_{kj}$



Overlap ordered path through the truncation zone visualised by gnuplot

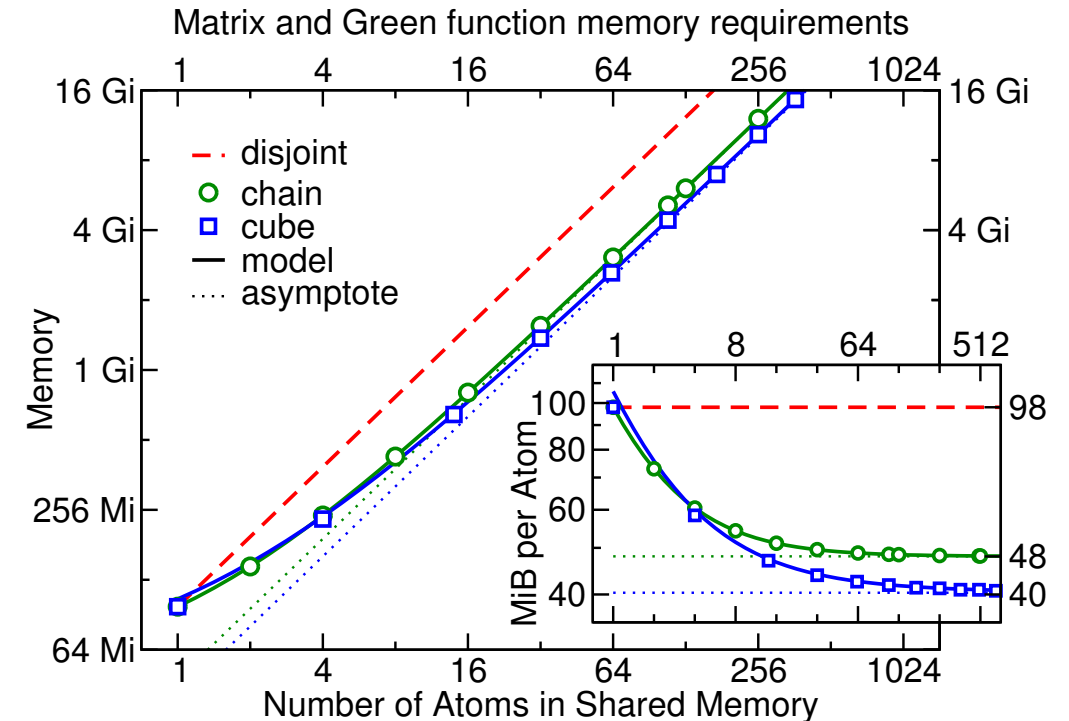
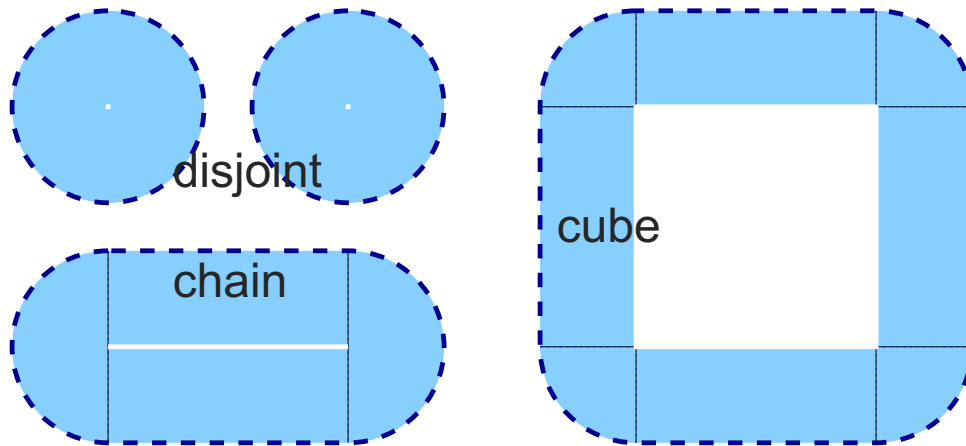
# THE MEMORY ISSUE

## Why we should aim for using more than 1 source atom per MPI process

Reason 1: We can re-use elements of  $A$  if we treat more than 1 source atom: median AI at 7.0 for 16 atoms

Reason 2: Memory consumption for 10 instances of  $X$  is given by  $160 \text{ Byte } b^2 n_{\text{sources}} 4\pi R_{\text{truncation}}^3$

Memory consumption for elements of  $A$  is approximately  $16 \text{ Byte } b^2 4\pi [R_{\text{truncation}} + (n_{\text{sources}}/4\pi)^{1/3}]^3$   
Assuming a sphere-shaped cluster of source atoms



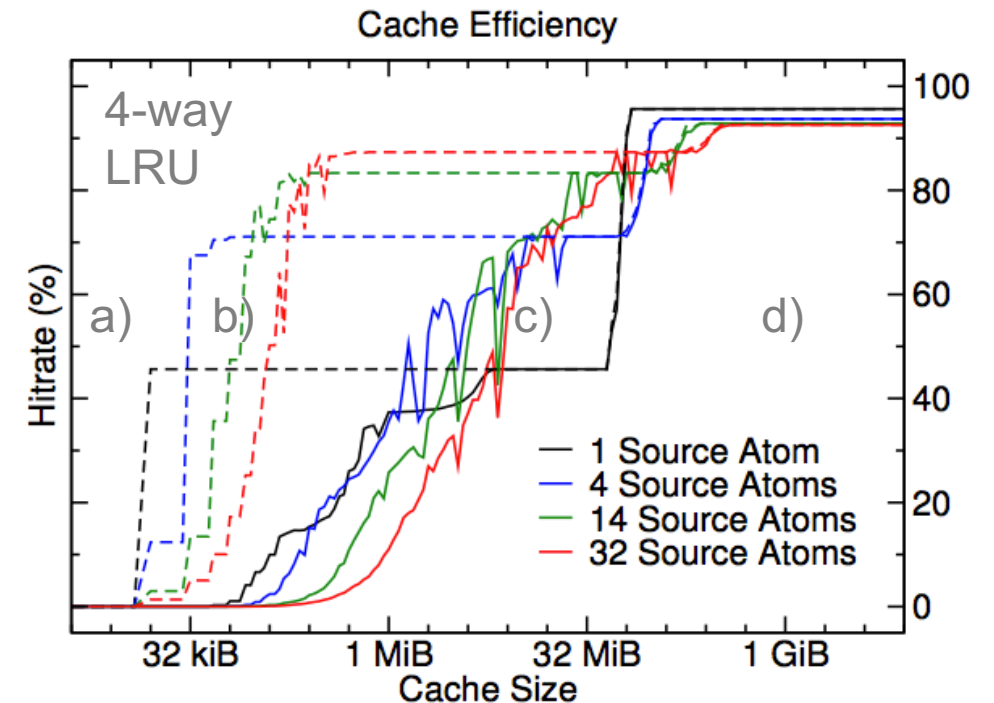
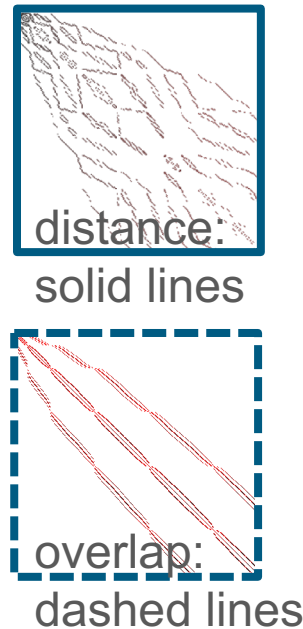
# CACHE SIMULATIONS

## Overlap ordering for many source atoms

Assume a 4-way associative cache with least recently used (LRU) eviction policy

We can identify four regimes:

- a) cache too small, low efficiency
- b) transition region for overlap
- c) transition region for distance
- d) entire problem fits into cache



# SPARSE MATRIX TIMES SPARSE MATRIX IN CODE

## The straightforward solution

$$Y_{ij} += A_{ik} \times X_{kj}$$

```
for iRow = 0 ... y_nRows
    for y = y_RowStart[iRow] ... y_RowStart[iRow + 1]
        jCol = y_ColIndex[y]
        Y[y][:][:] = 0

        for a = a_RowStart[iRow] ... a_RowStart[iRow + 1]
            kCol = a_ColIndex[a]
            kRow = kCol

            if (x = BSR_entry_exists(kRow, jCol))
                ZGEMM( A[a], X[x], Y[y] )
```

Pseudo-code

# TASK SCHEDULING

## The straightforward schedule for one thread

```
tasks = 0
for iRow = 0 ... y_nRows
    for y = y_RowStart[iRow] ... y_RowStart[iRow + 1]
        jCol = y_ColIndex[y]
        scalar = 0

        for a = a_RowStart[iRow] ... a_RowStart[iRow + 1]
            kCol = a_ColIndex[a]
            kRow = kCol
            if (x = BSR_entry_exists(kRow, jCol))
                tasks[0].append( {a, x, y, scalar} )
                scalar = 1
```

## Pseudo-code

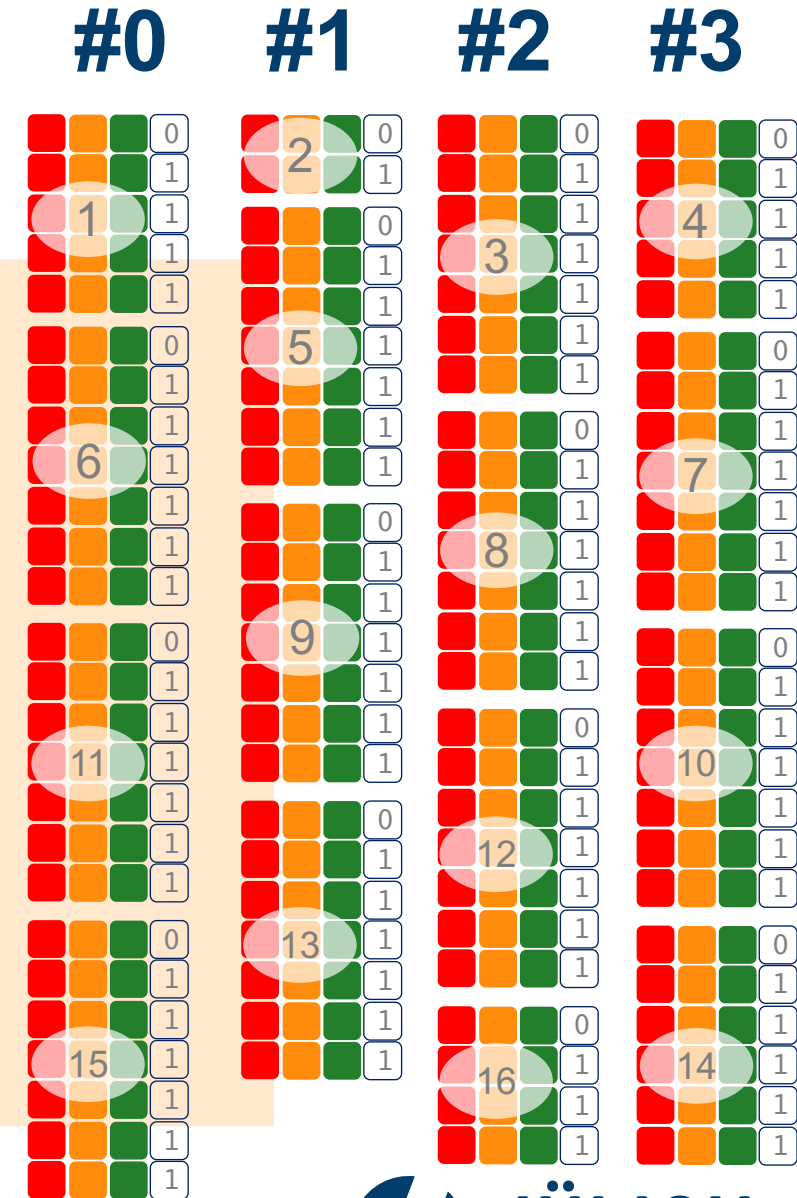


# TASK SCHEDULING FOR THREADS

A simple scheduling policy for many threads

```
tasks = 0
for iRow = 0 ... y_nRows
  for y = y_RowStart[iRow] ... y_RowStart[iRow + 1]
    jCol = y_ColIndex[y]
    scalar = 0
    t = minloc(tasks) // choose the thread
    for a = a_RowStart[iRow] ... a_RowStart[iRow + 1]
      kCol = a_ColIndex[a]
      kRow = kCol
      if (x = BSR_entry_exists(kRow, jCol))
        tasks[t].append( {a, x, y, scalar} )
        scalar = 1
```

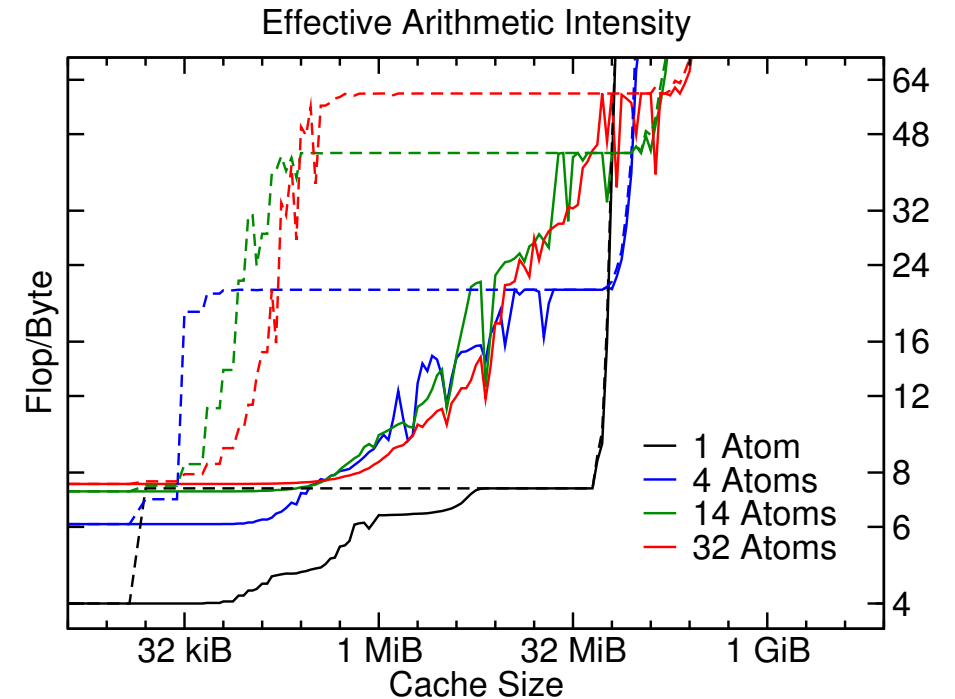
Pseudo-code



# REAL WORLD EXPERIMENT FAILURE

## Using the sparse matrix structures of a real world problem

- Setup with 16 source atoms, up to 289 target atoms
- Matrix block size  $b=32$  from a non-collinear magnetic system
- No measureable effect due to reordering for a single CPU thread
- No effect even with reduced block sizes, e.g.  $b=4$ , or with artificially reduced arithmetic intensity (e.g. element-wise multiplication)



Hitrates translated into an expected effective arithmetic intensity behind the cache, i.e.  $\text{total\#Flop}/\text{total\#Byte}$