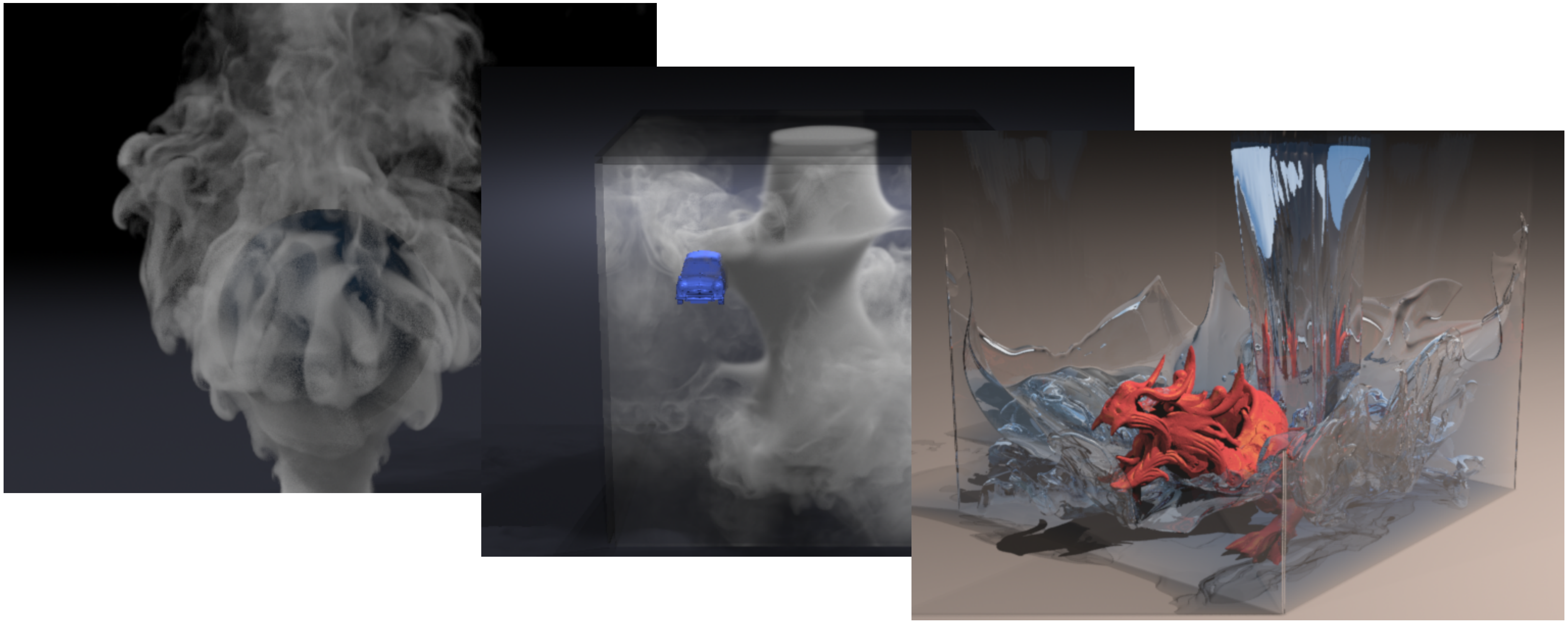


Parallel Sparse Direct Solvers
Performance & design of MKL PARDISO (wrap-up)
+ A few concluding notes on memory prefetching

Sparse Factorizations:
Obstacles to performance & parallelism

*Matrix Density : The number of required operations scale (super-linearly …) with the number of non-zero entries in **L** … thus, ensuring sparser **L** factors has an immediate effect on performance*

Multithreading : Cholesky, similar to Gauss Elimination, is seemingly a very "serial" algorithm (significant dependencies between steps/loops). We must find some way to cope with this apparent limitation.

SparseDirect/LaplacePARDISO_0_0

PARDISO solver (DirectSolver.cpp)

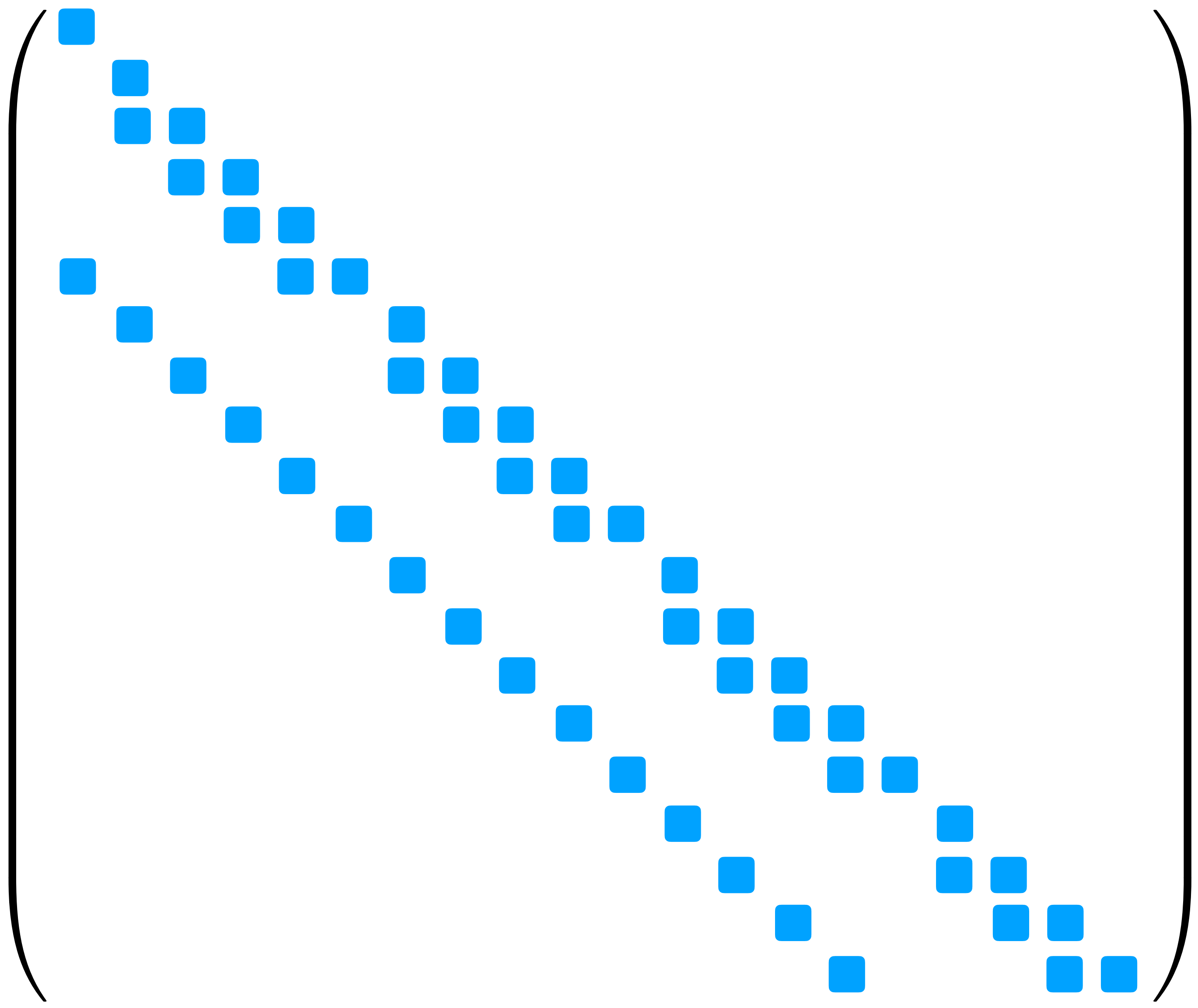
Execution:

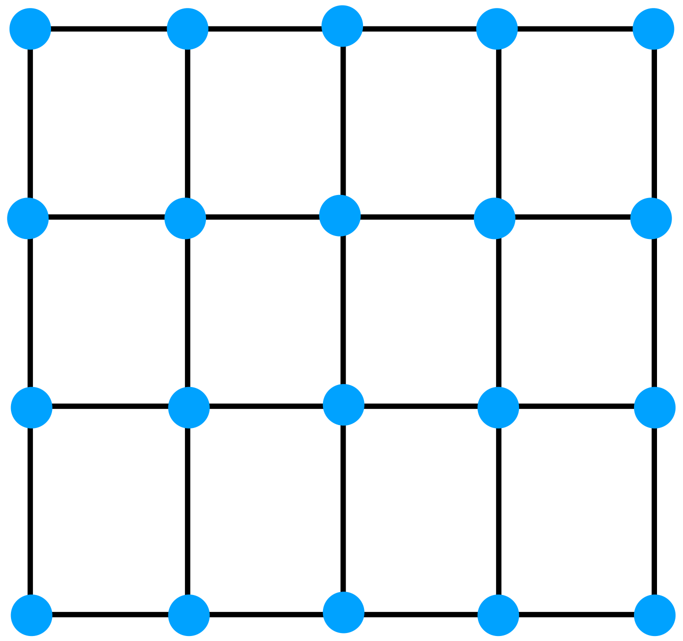
```
Summary: ( factorization phase )
================
Times:
======
Time spent in copying matrix to internal data structure (A to LU): 0.000000 s
Time spent in factorization step (numfct)                        : 44.352600 s
Time spent in allocation of internal data structures (malloc)    : 0.022322 s
Time spent in additional calculations                            : 0.000002 s
Total time spent                                                 : 44.374928 s
Statistics:
===========
Parallel Direct Factorization is running on 20 OpenMP
< Linear system Ax = b >
             number of equations:           2097152
             number of non-zeros in A:      8050652
             number of non-zeros in A (%): 0.000183
             number of right-hand sides:    1
< Factors L and U >
             number of columns for each panel: 96
             number of independent subgraphs:  0
             number of supernodes:                    1410153
             size of largest supernode:               16591
             number of non-zeros in L:                2057589566
             number of non-zeros in U:                1
             number of non-zeros in L+U:              2057589567
             gflop   for the numerical factorization: 22775.748047
             gflop/s for the numerical factorization: 513.515503

Factorization completed ...
 PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```

About 90% of overall runtime (sometimes less)

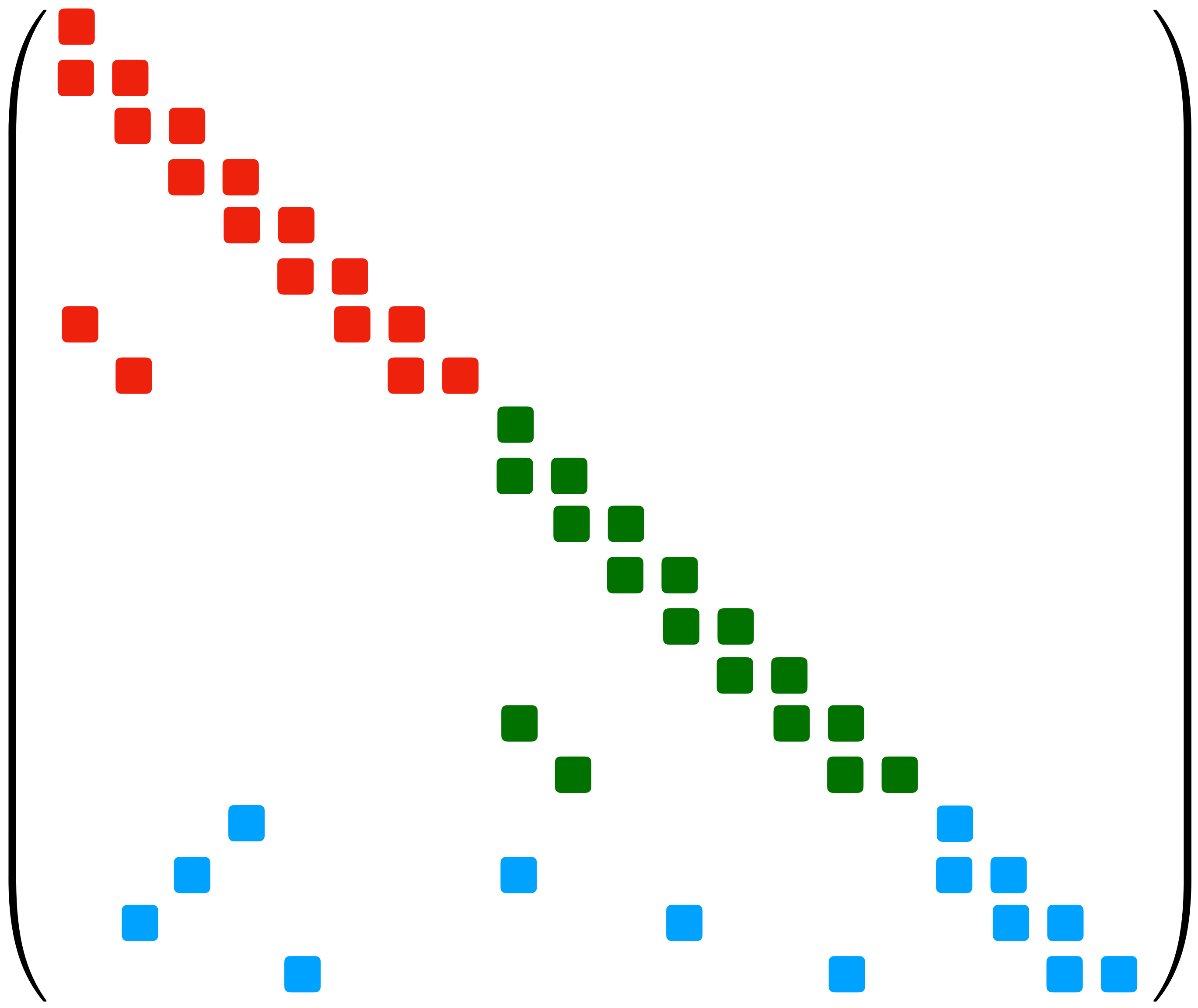
Almost 25% of peak arithmetic utilization

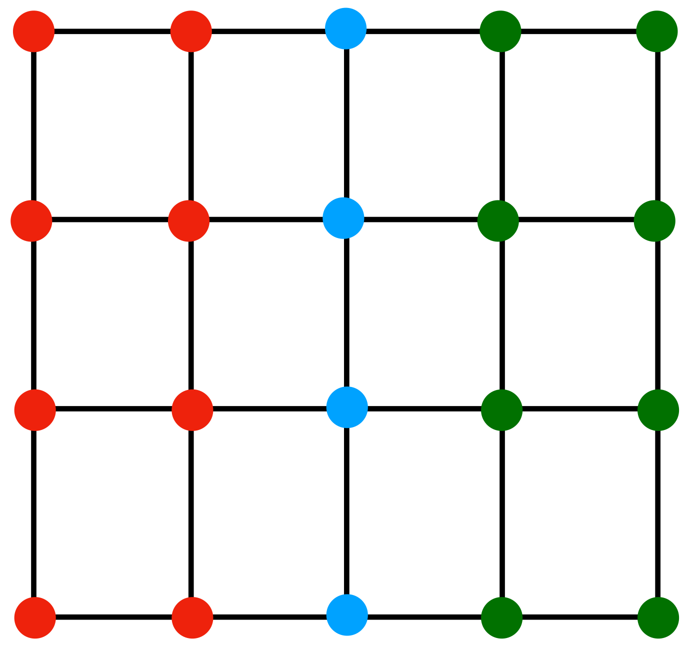
Engineering/Maximizing Sparsity



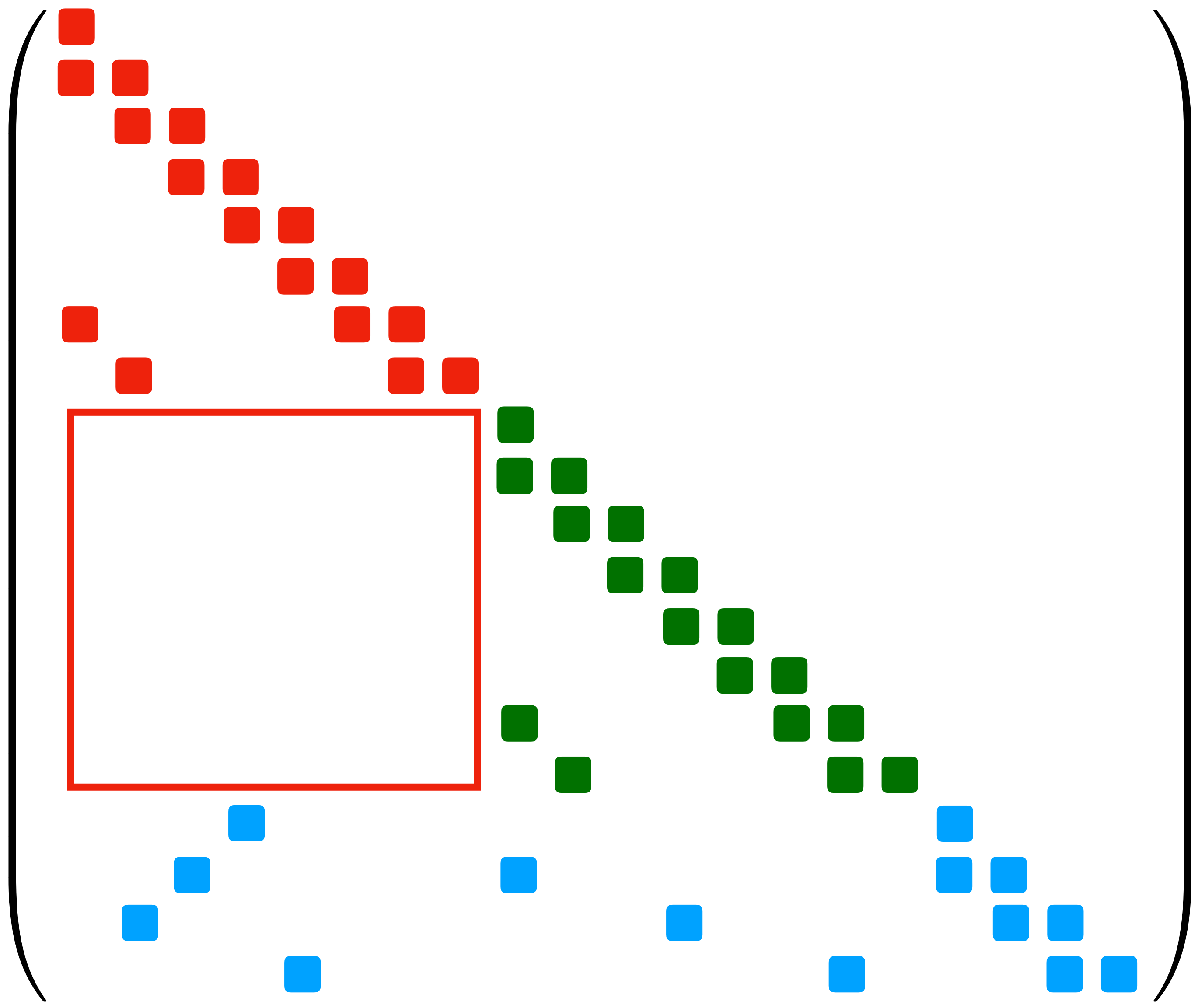


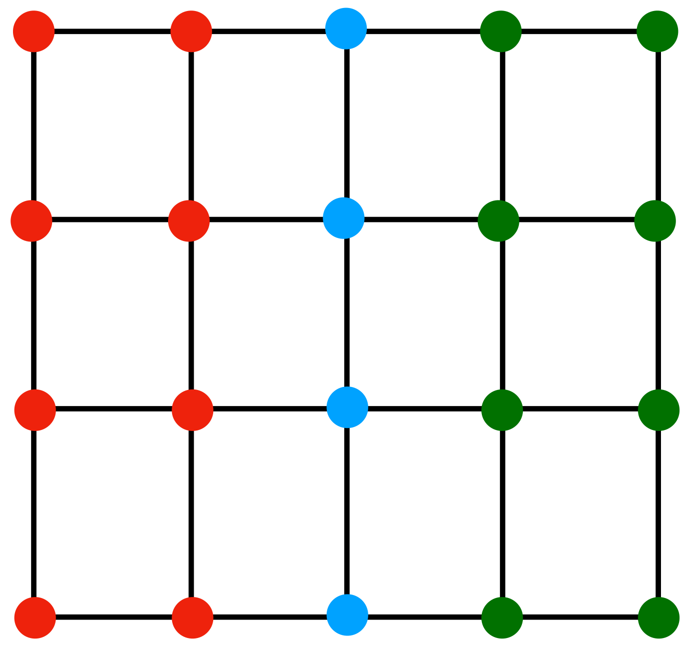
Engineering/Maximizing Sparsity



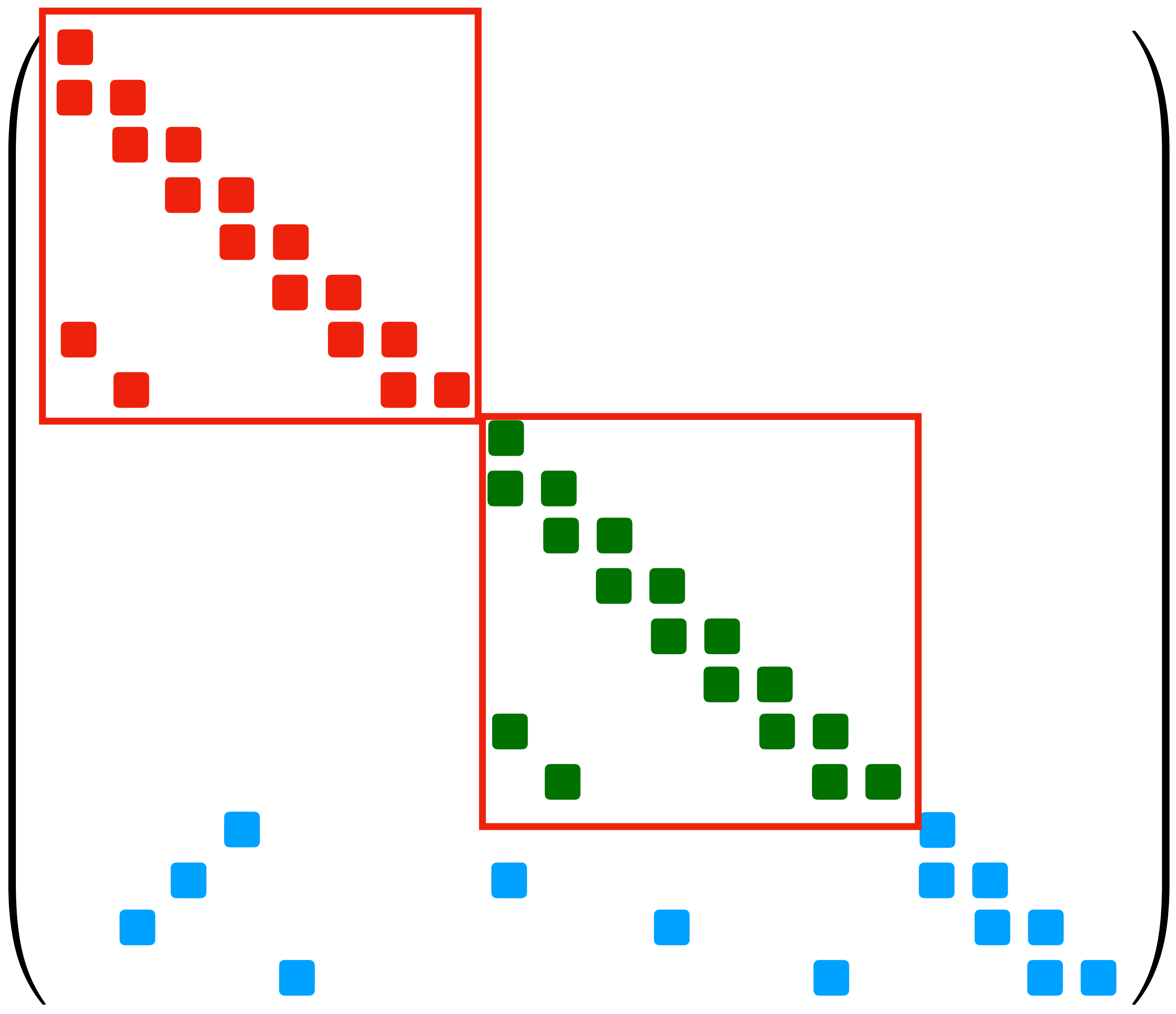


Engineering/Maximizing Sparsity



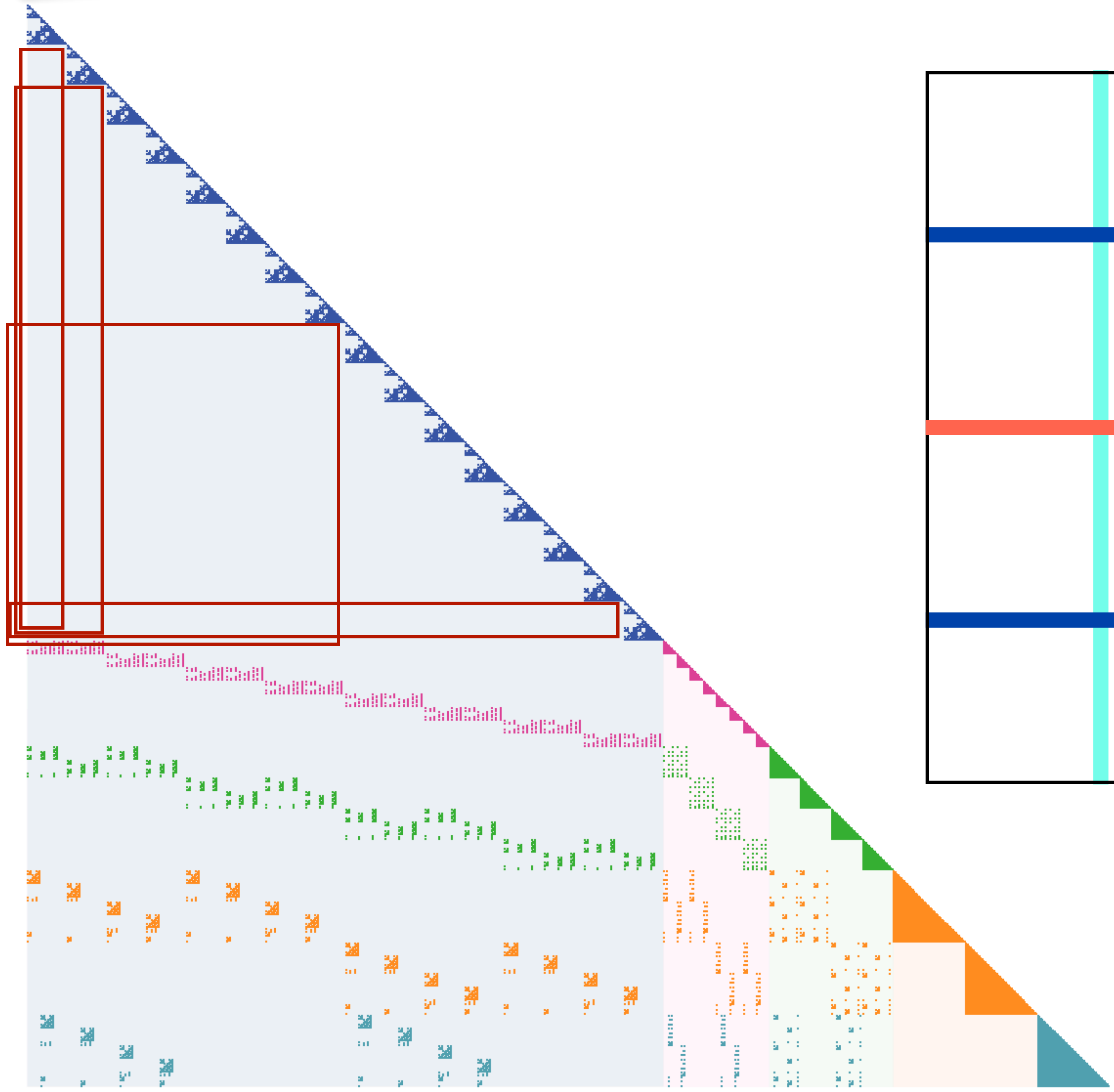


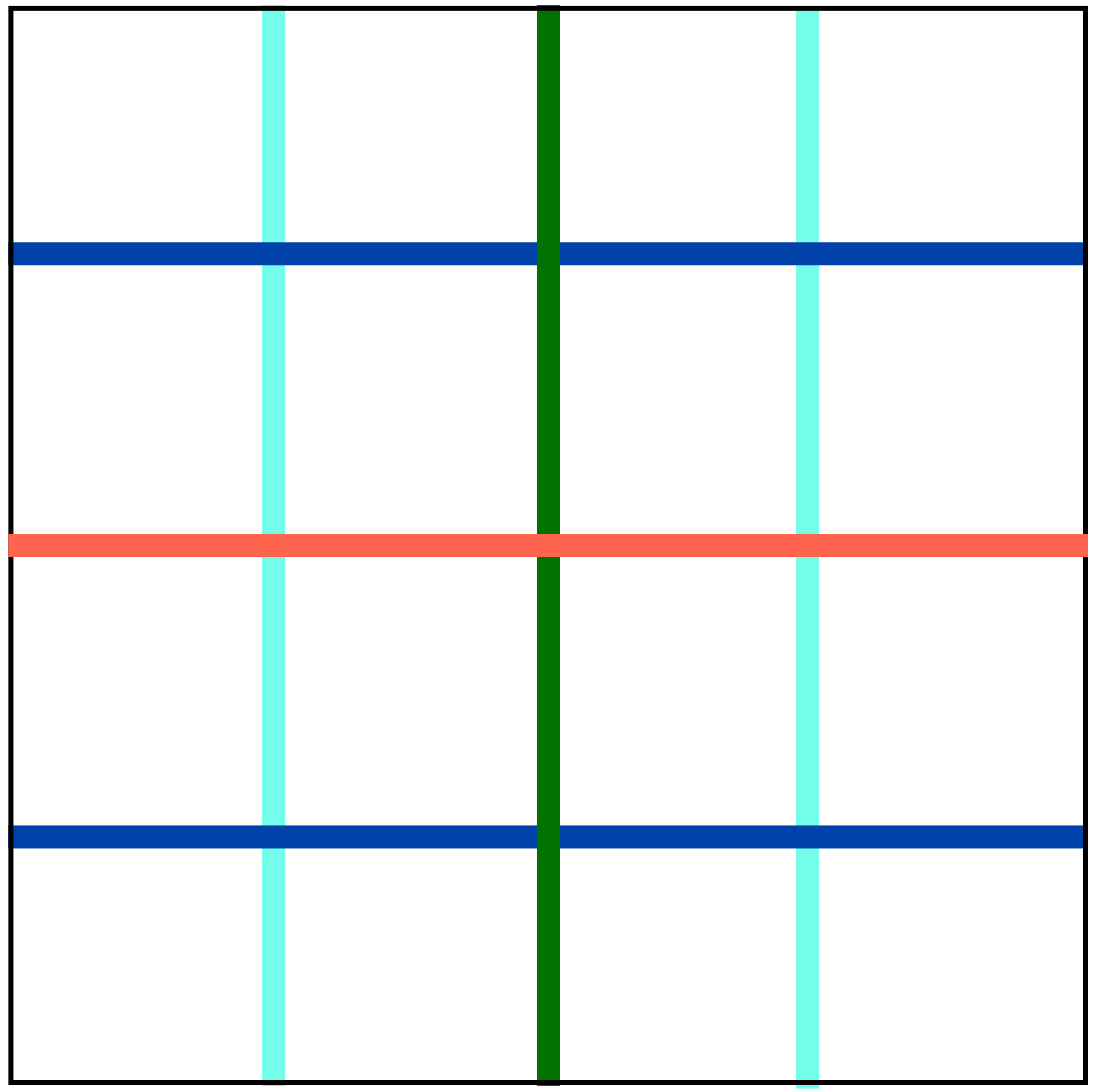
Engineering/Maximizing Sparsity



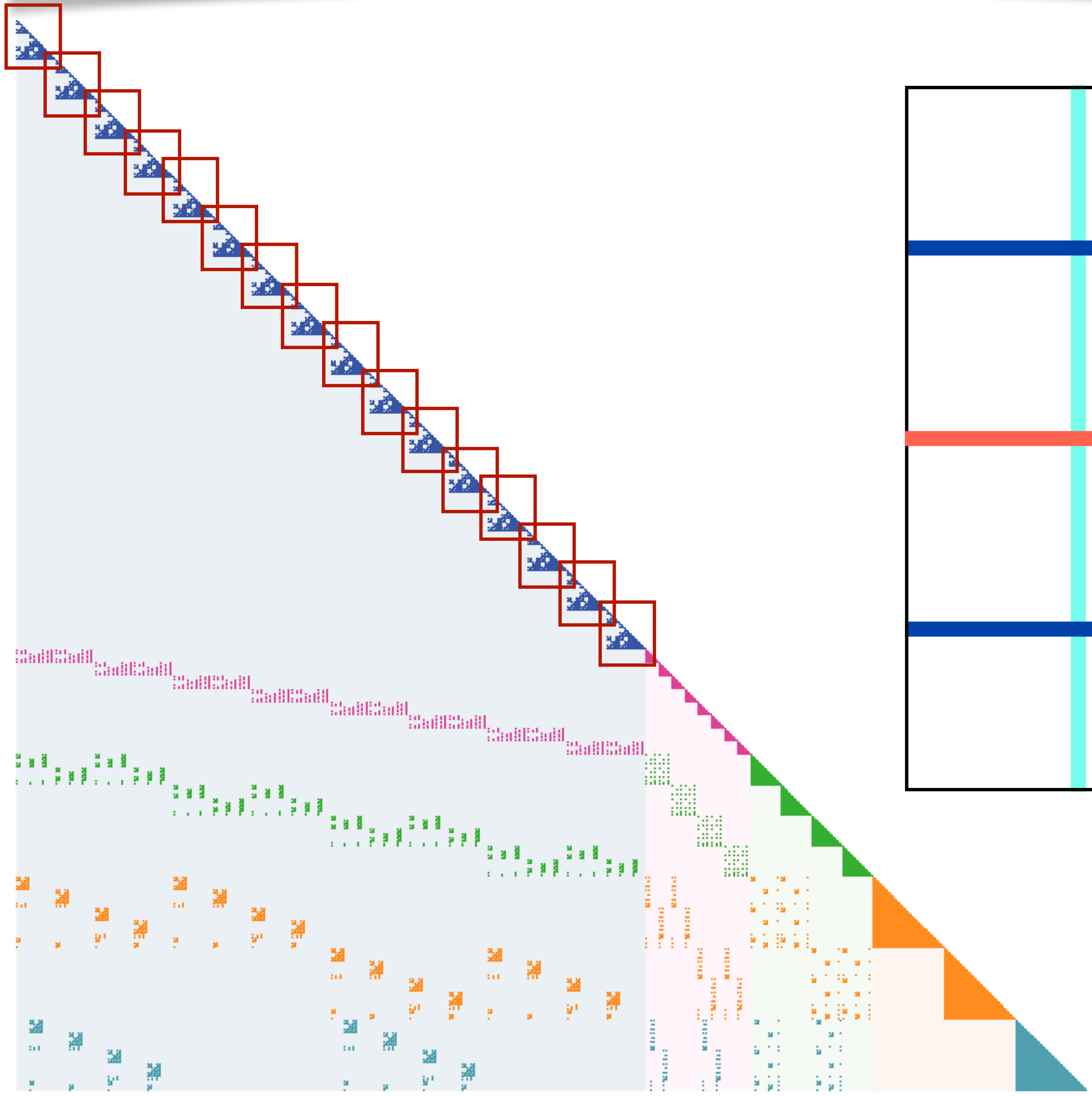
Second benefit:
Cholesky can process each of these two blocks in-parallel!

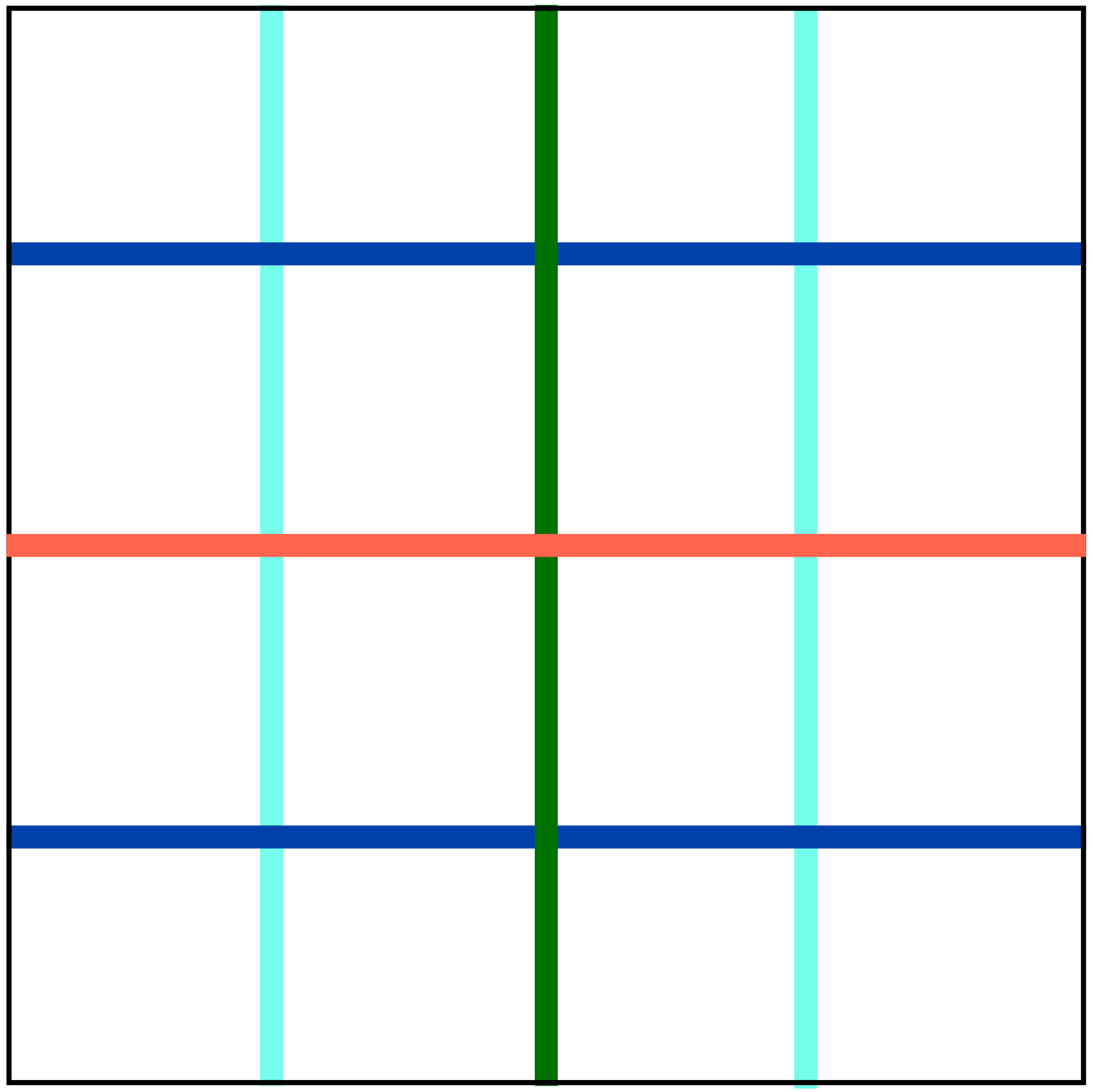
Laplacian - Pattern after a possible reordering





Laplacian - Pattern after a possible reordering





These blocks, too, can be processed in parallel

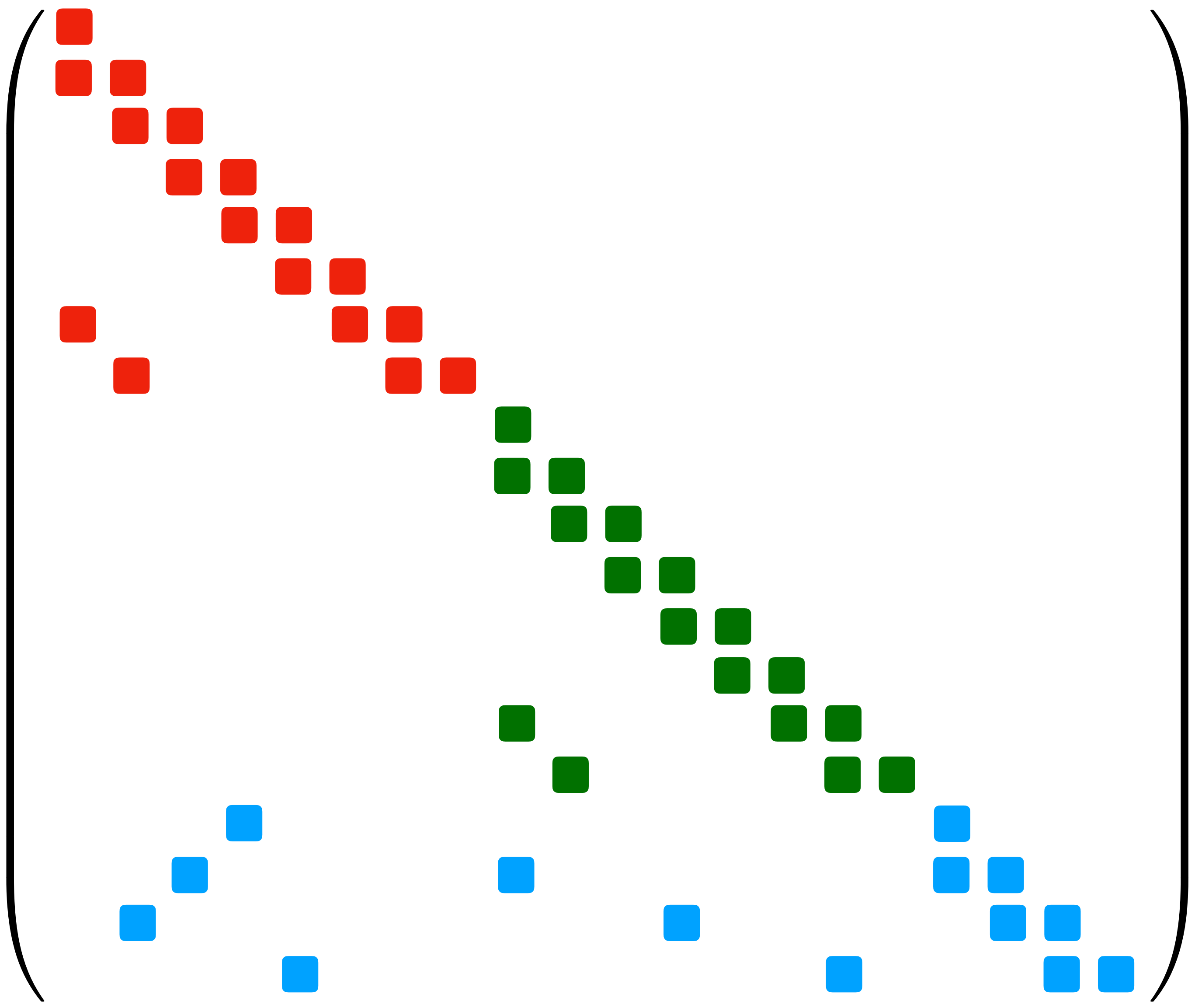
Obstacles to performance & parallelism

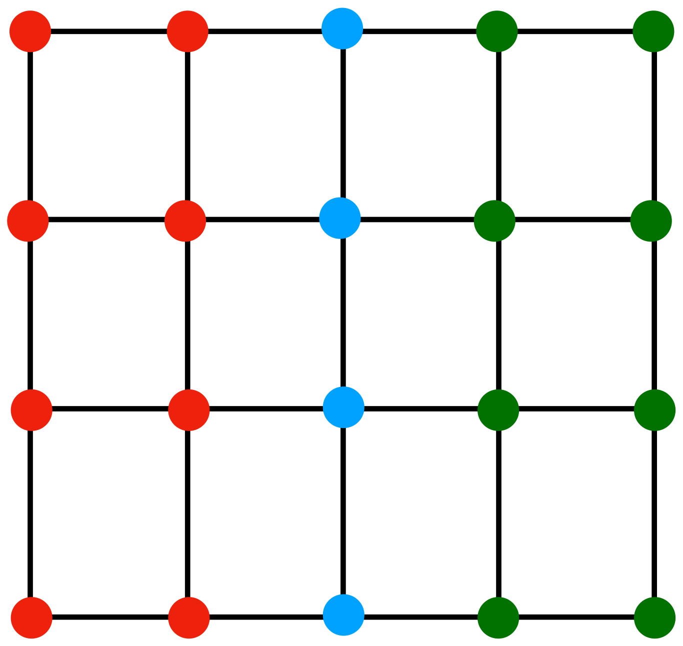
*Matrix Density : The number of required operations scale (super-linearly …) with the number of non-zero entries in **L** … thus, ensuring sparser **L** factors has an immediate effect on performance*

Multithreading : Cholesky, similar to Gauss Elimination, is seemingly a very "serial" algorithm (significant dependencies between steps/loops). We must find some way to cope with this apparent limitation.

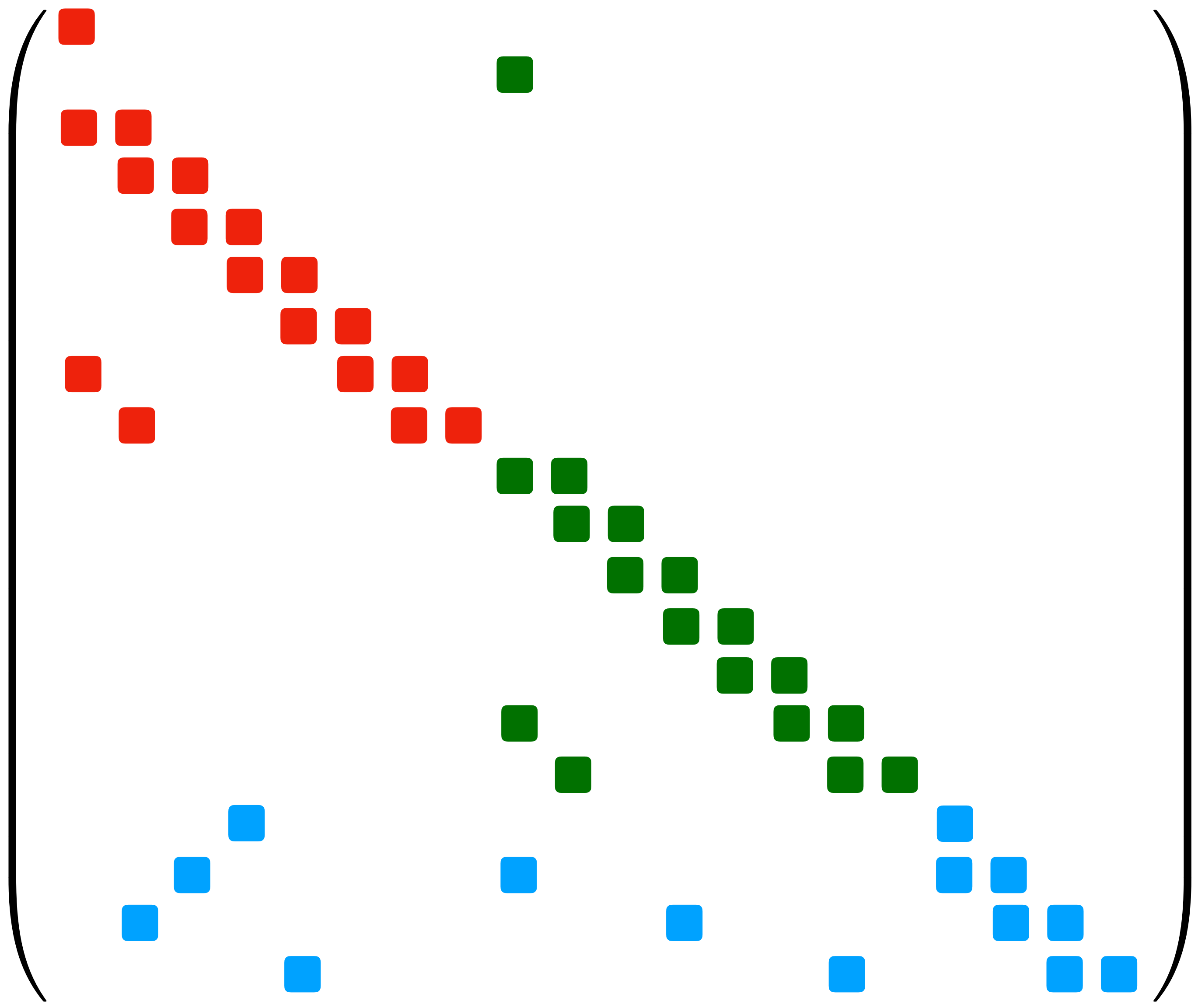
Vectorization/SIMD : Sparse matrices don't have the regularity that SIMD operations require; we need to "engineer" such regularity if possible

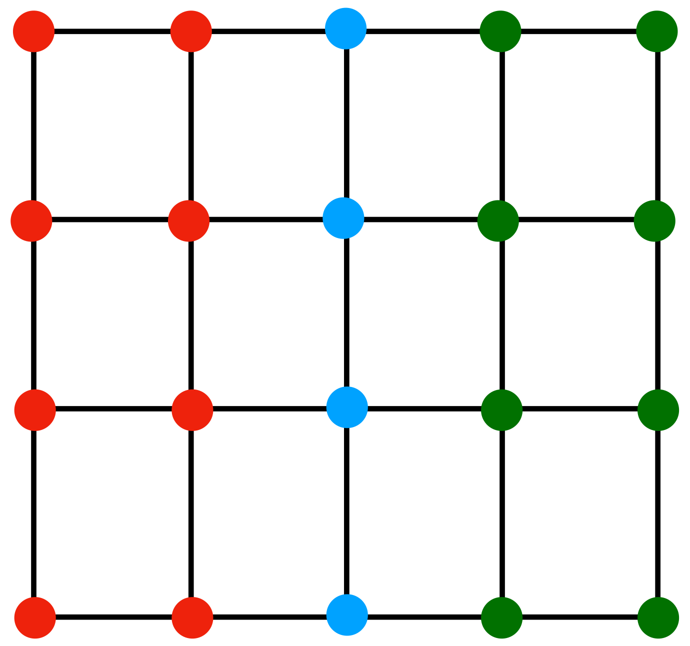
Engineering/Maximizing Sparsity



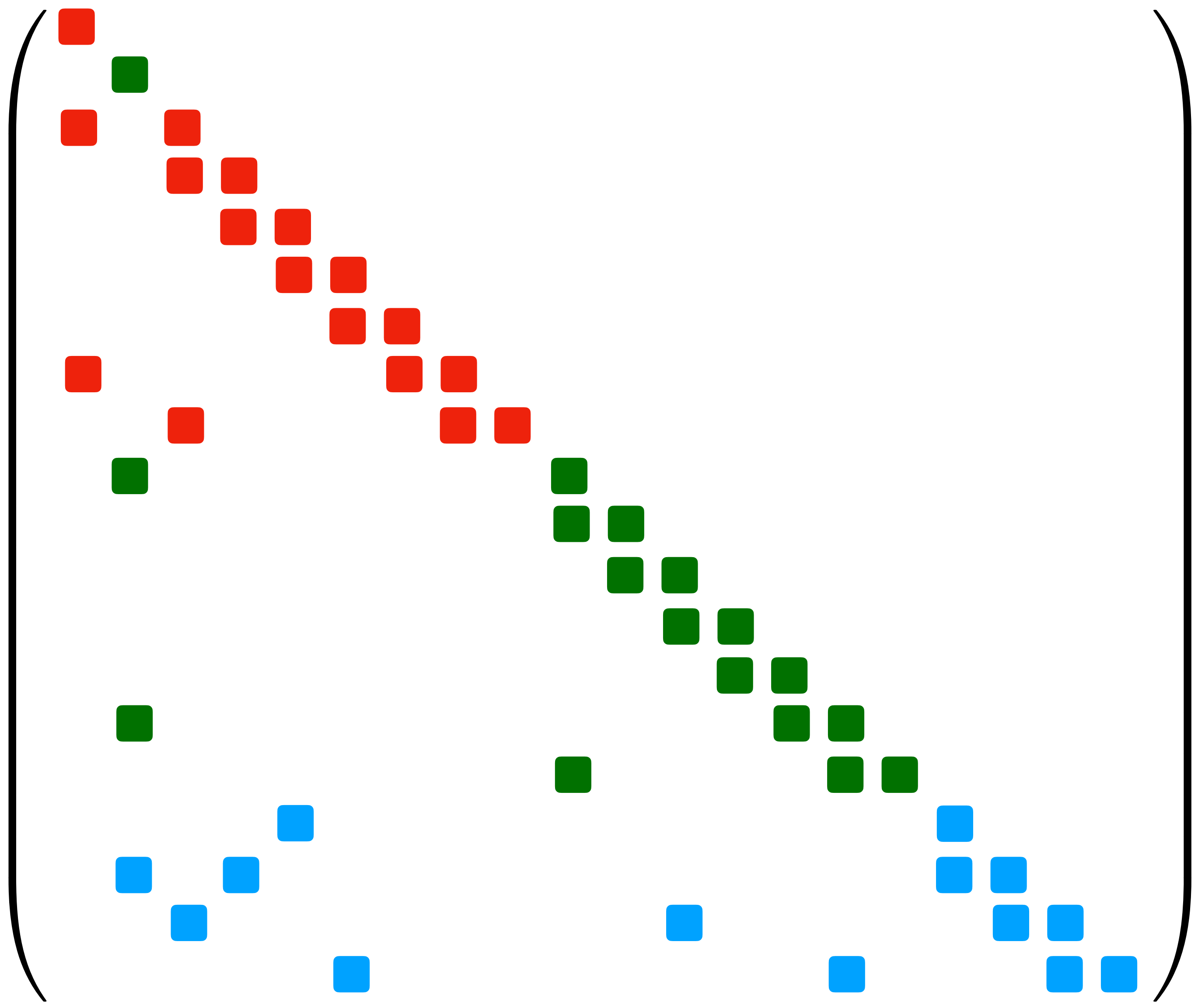


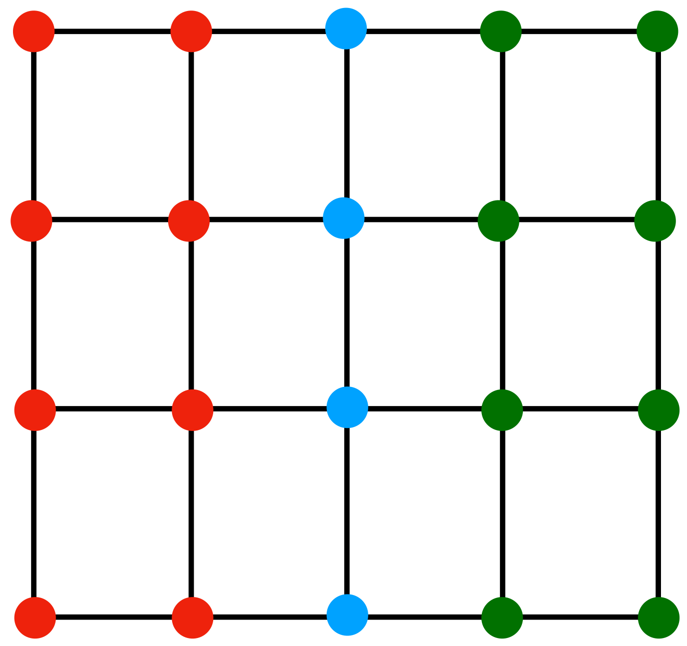
Engineering/Maximizing Sparsity



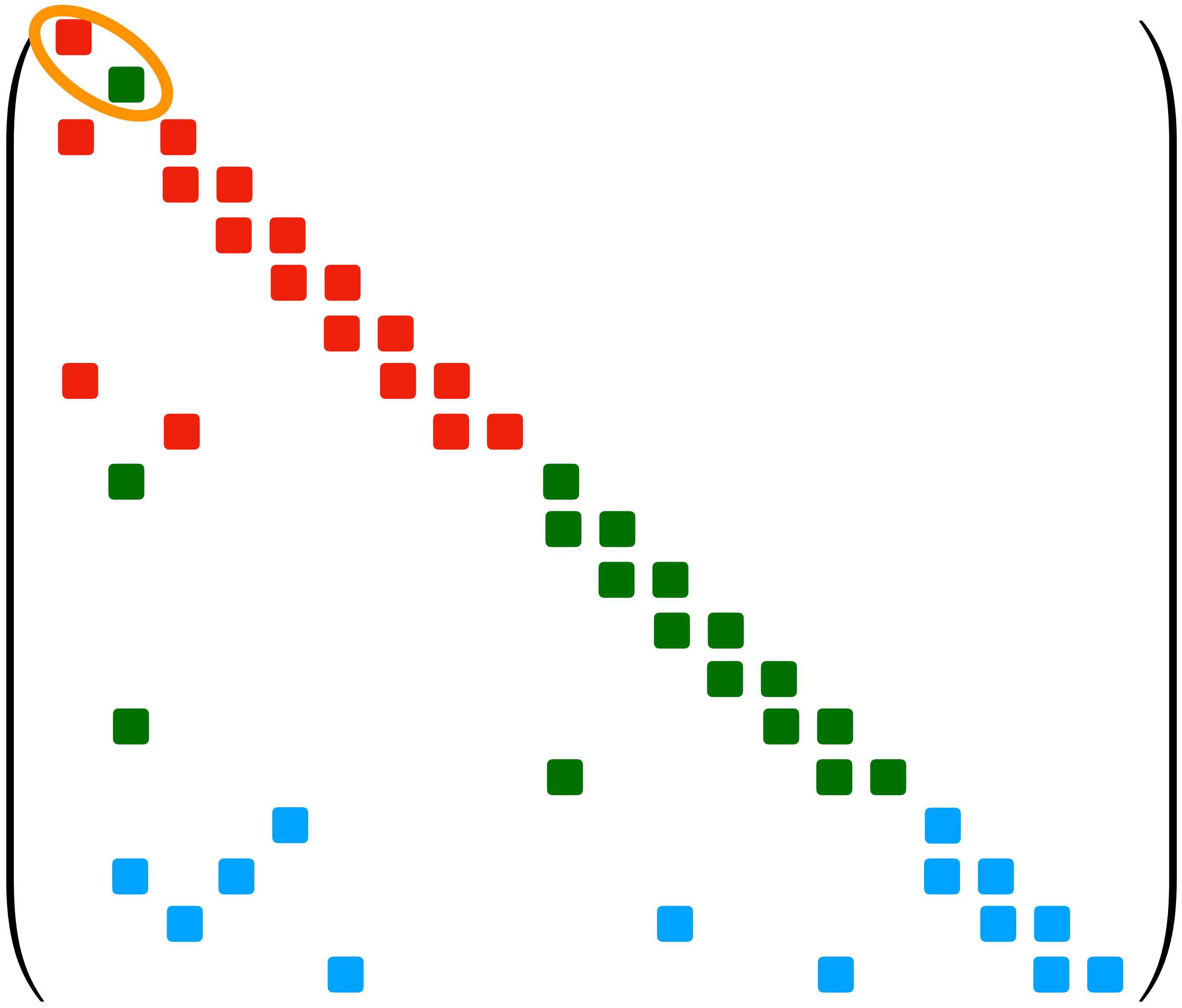


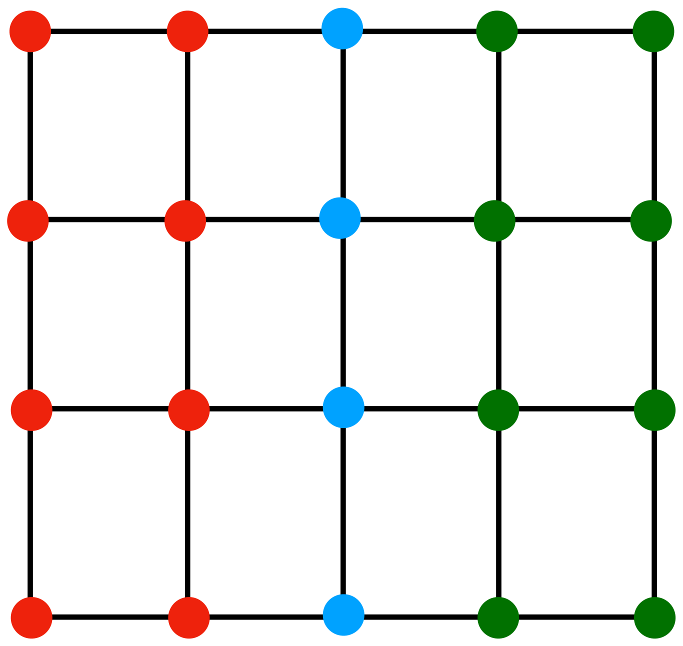
Engineering/Maximizing Sparsity



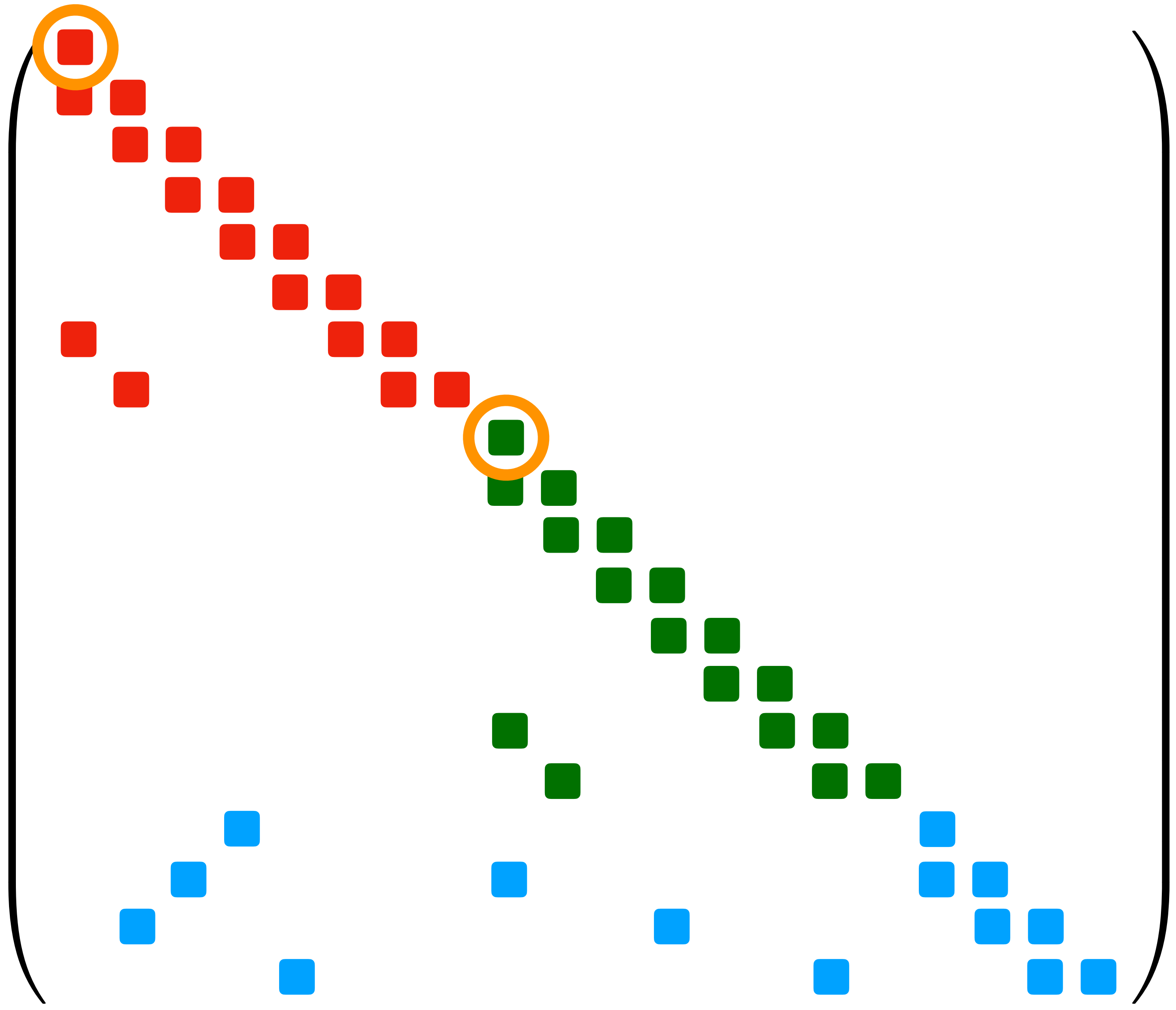


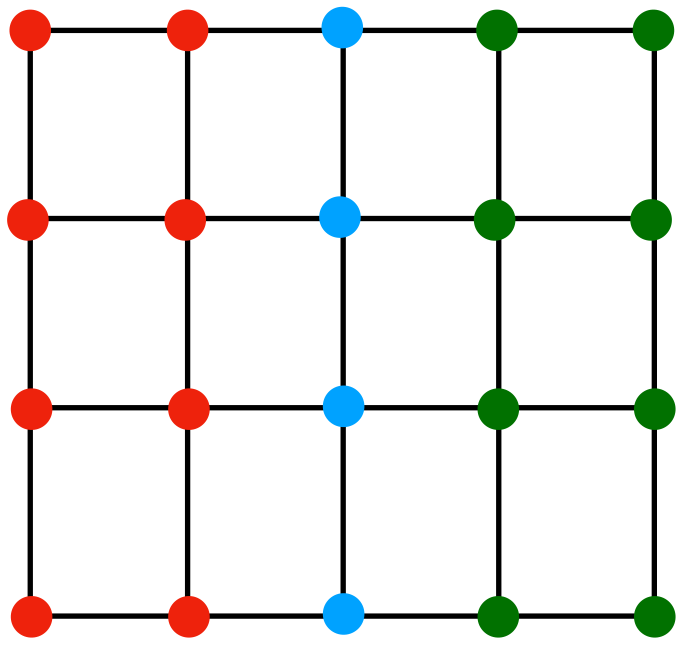
Engineering/Maximizing Sparsity



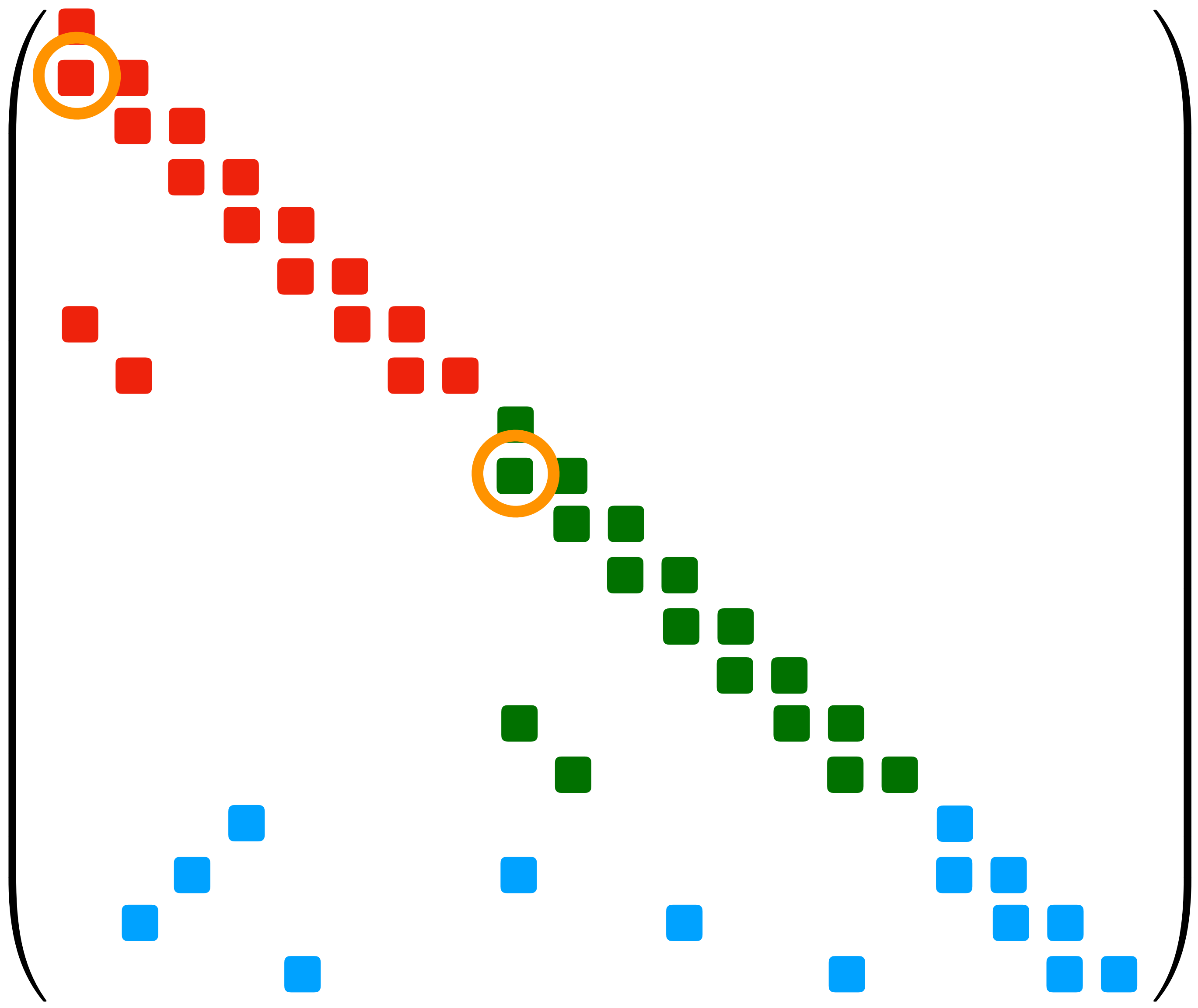


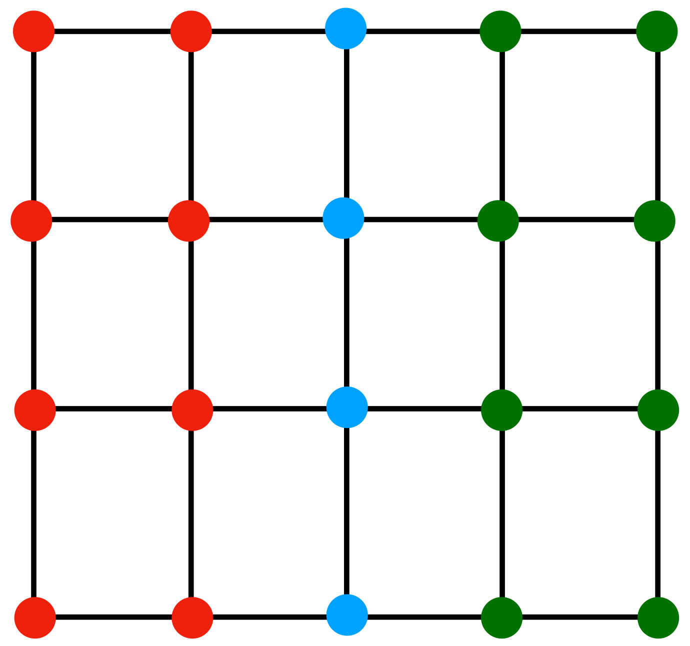
Engineering/Maximizing Sparsity



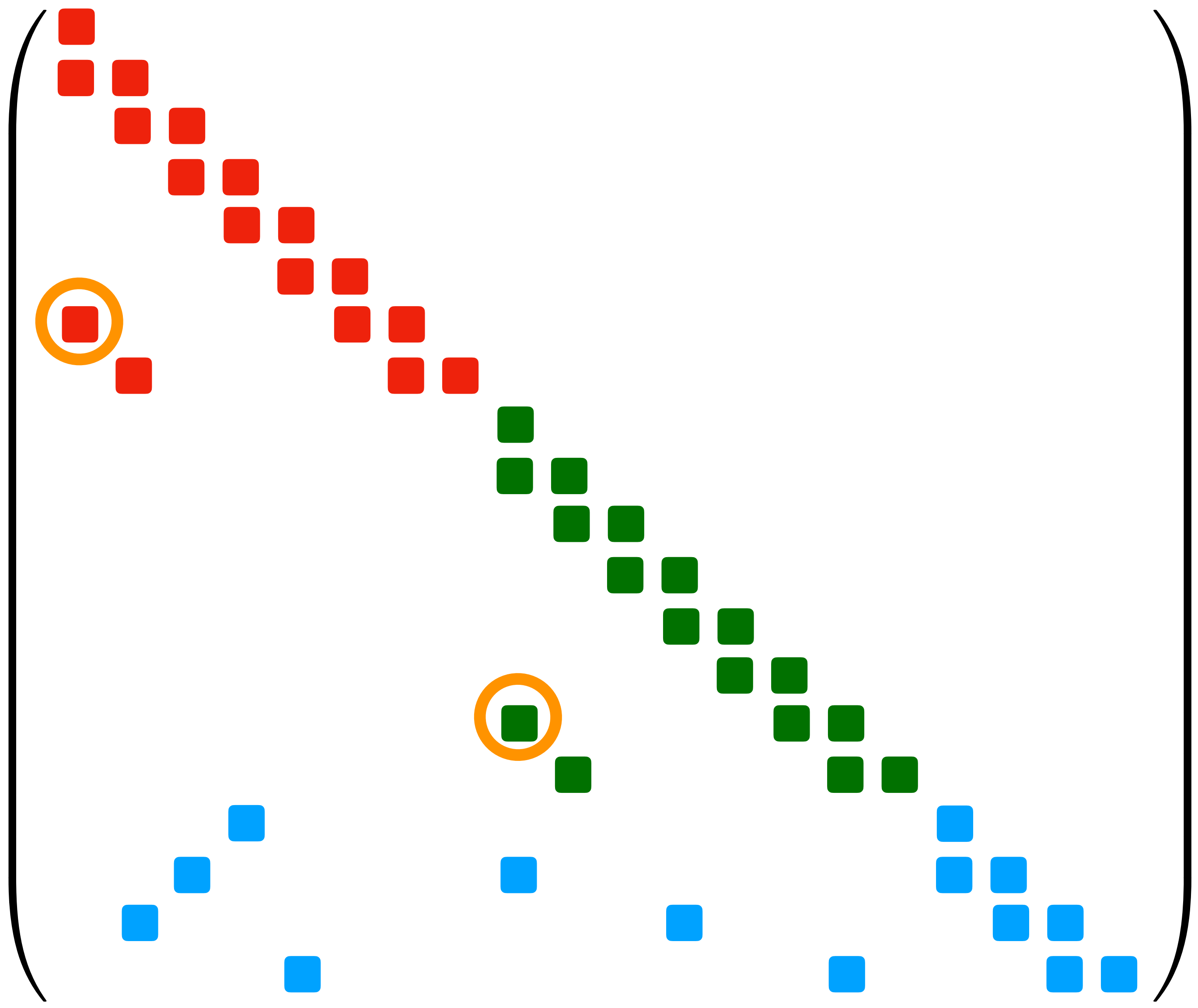


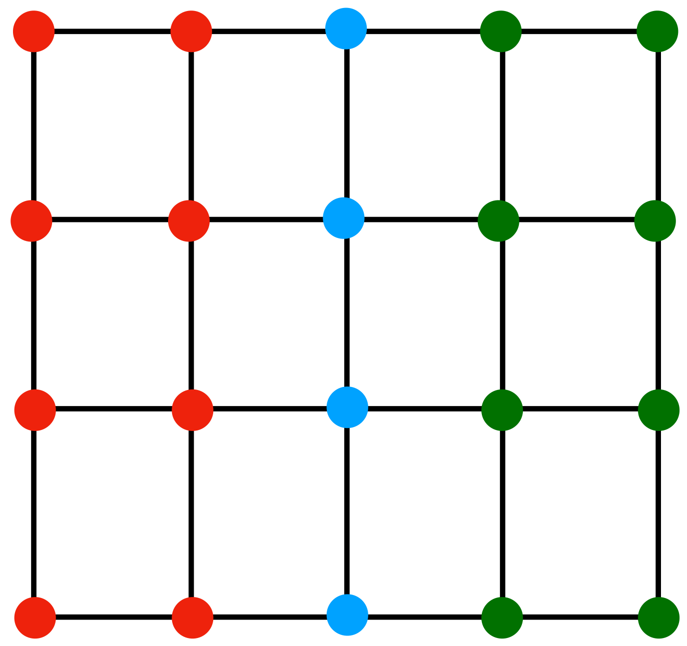
Engineering/Maximizing Sparsity



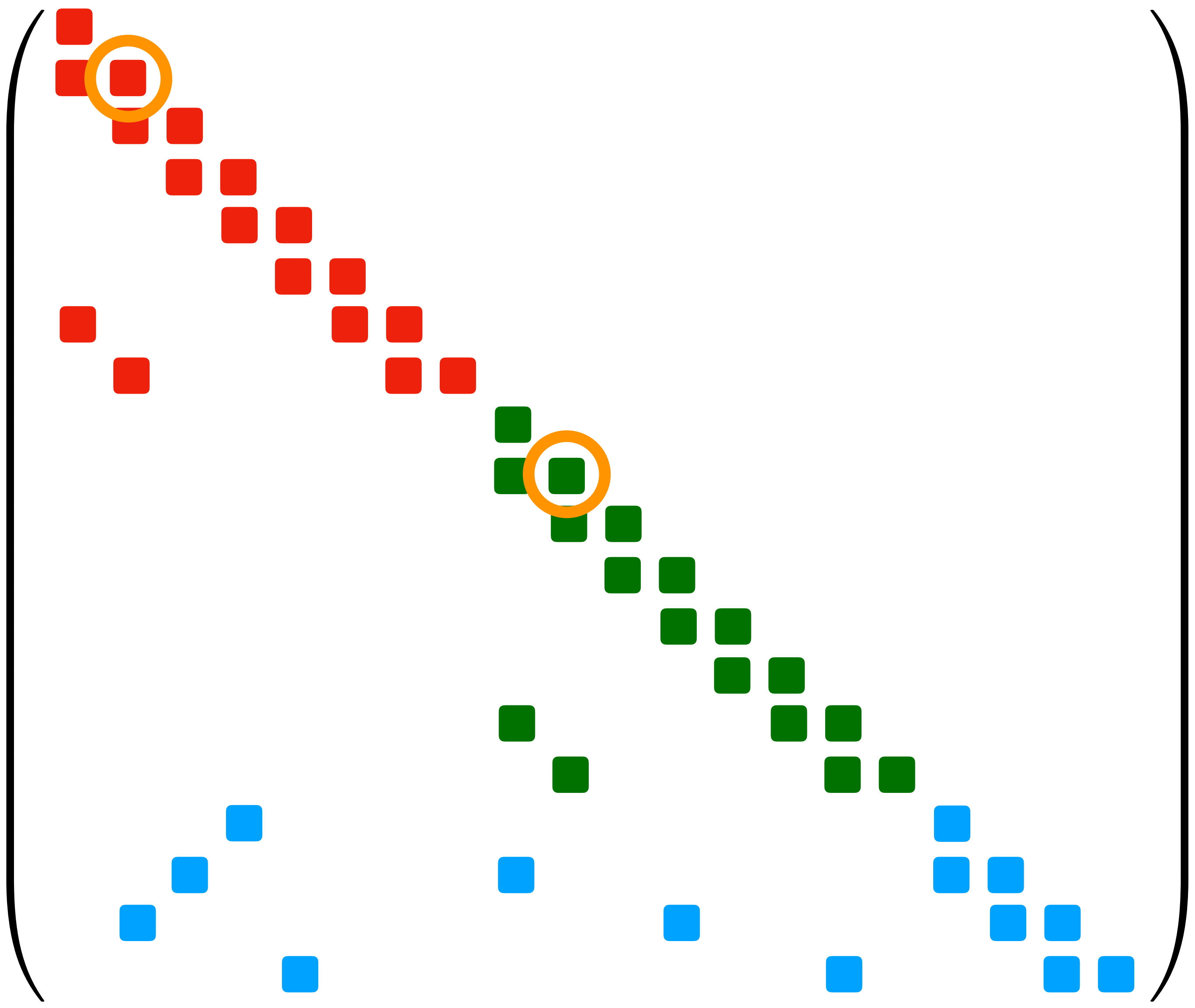


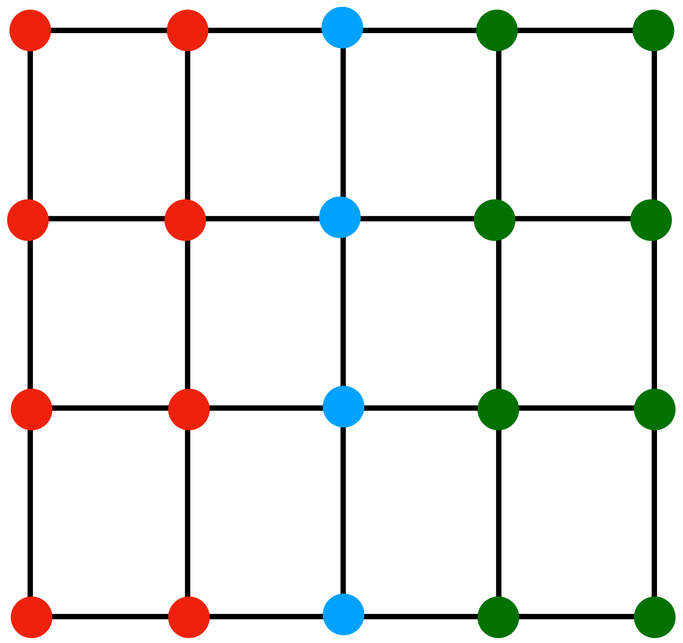
Engineering/Maximizing Sparsity



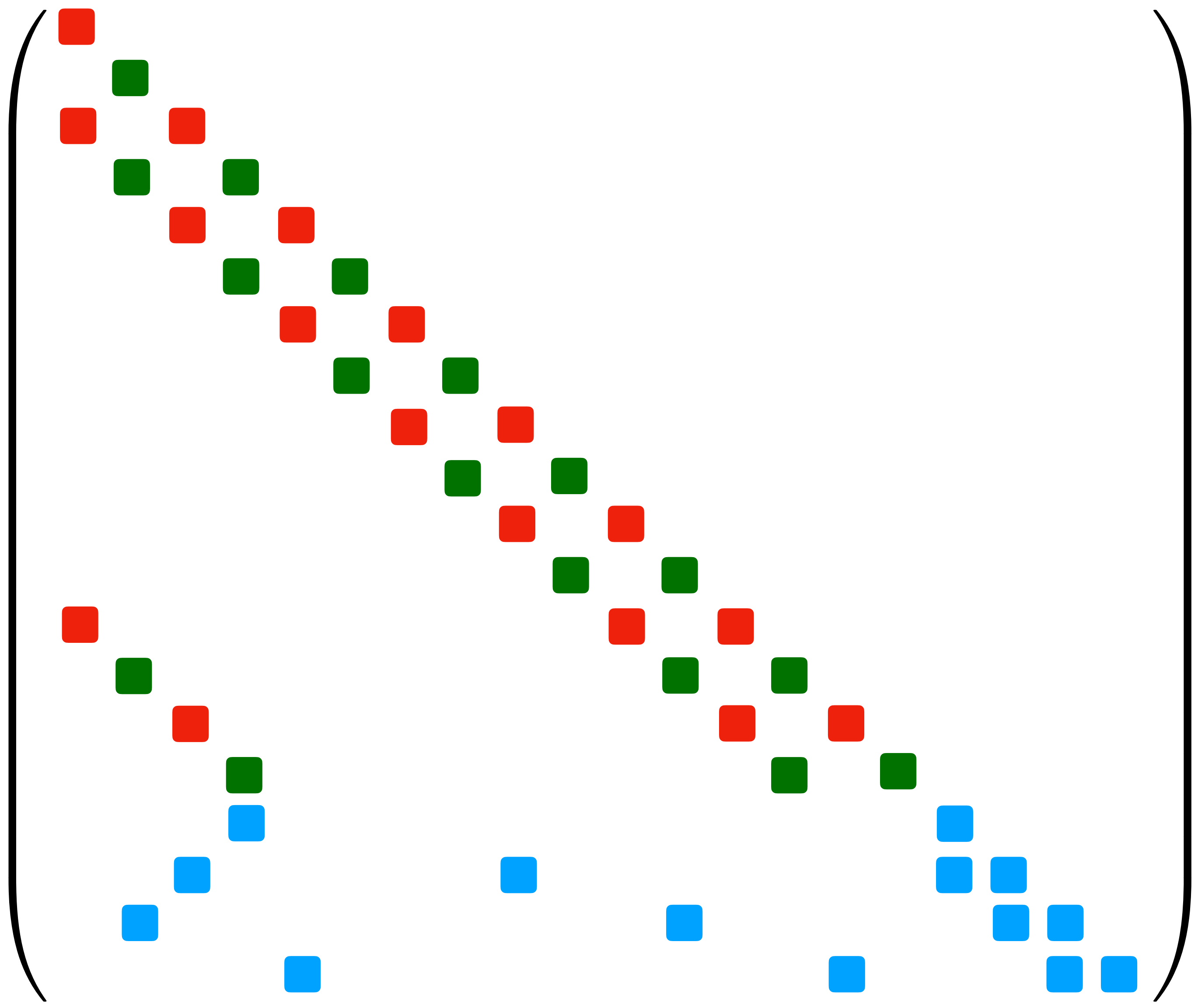


Engineering/Maximizing Sparsity



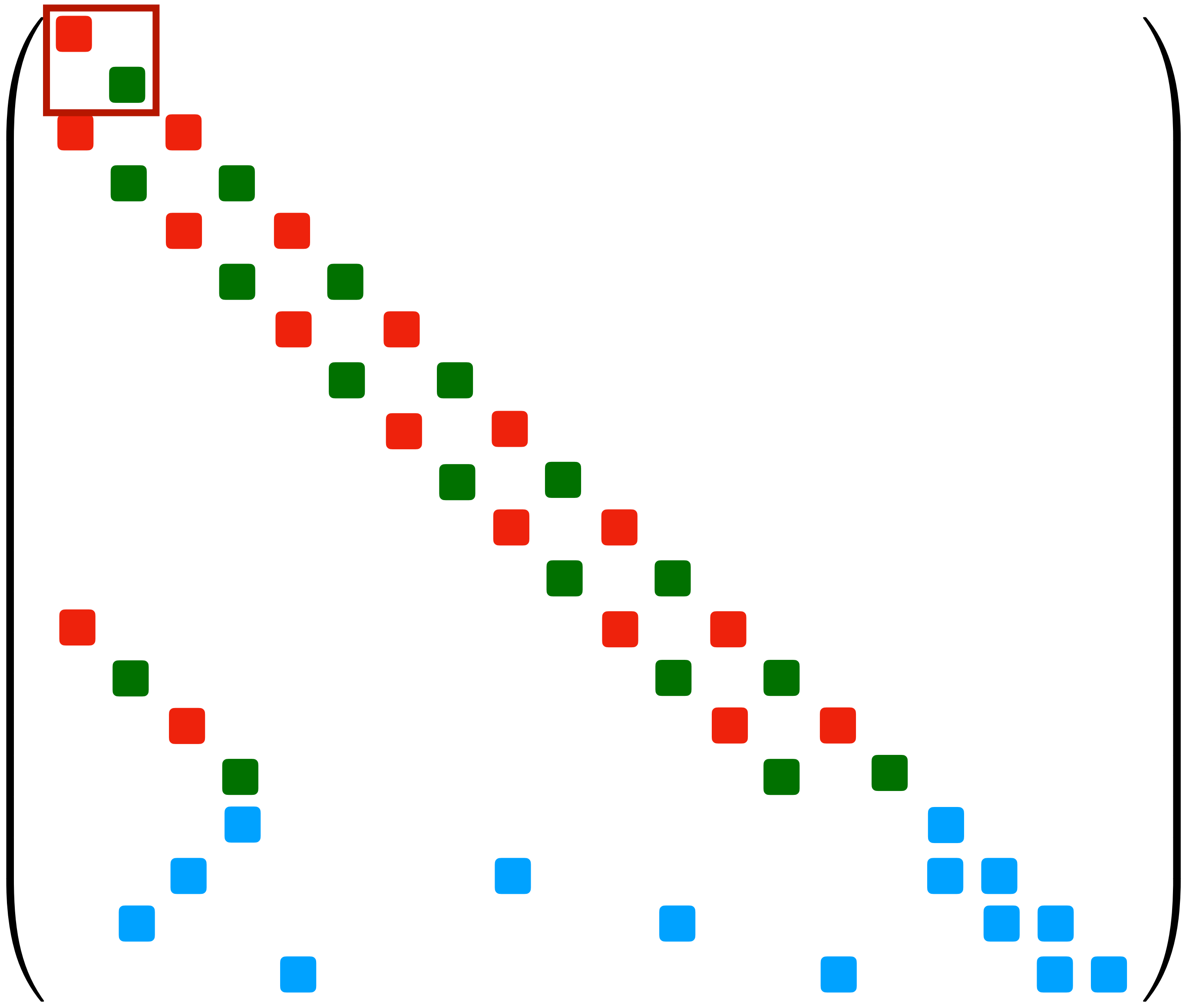


Engineering/Maximizing Sparsity



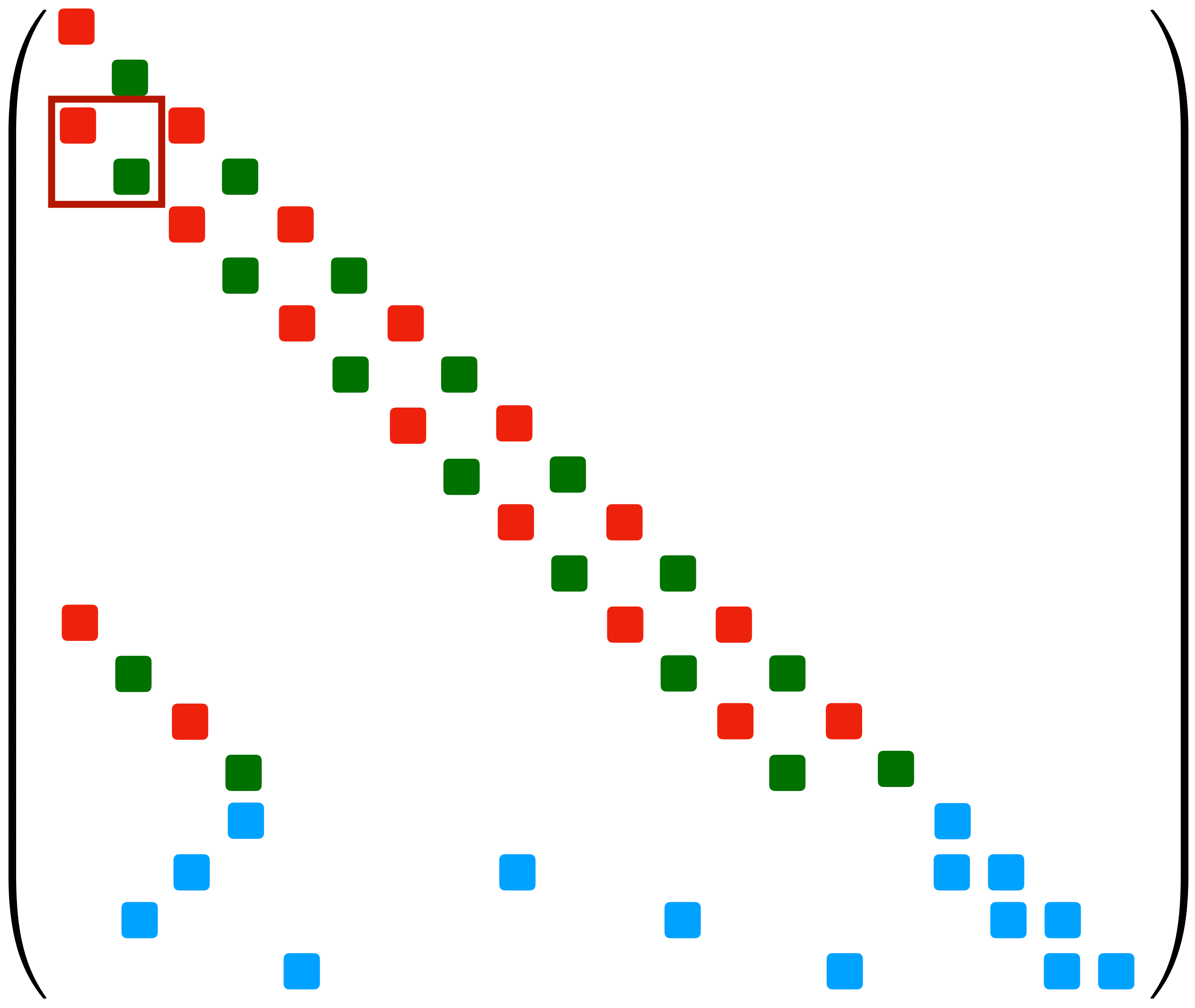
After reordering:
Sparsity pattern becomes a (sparse) collection of diagonal ***blocks***

Engineering/Maximizing Sparsity



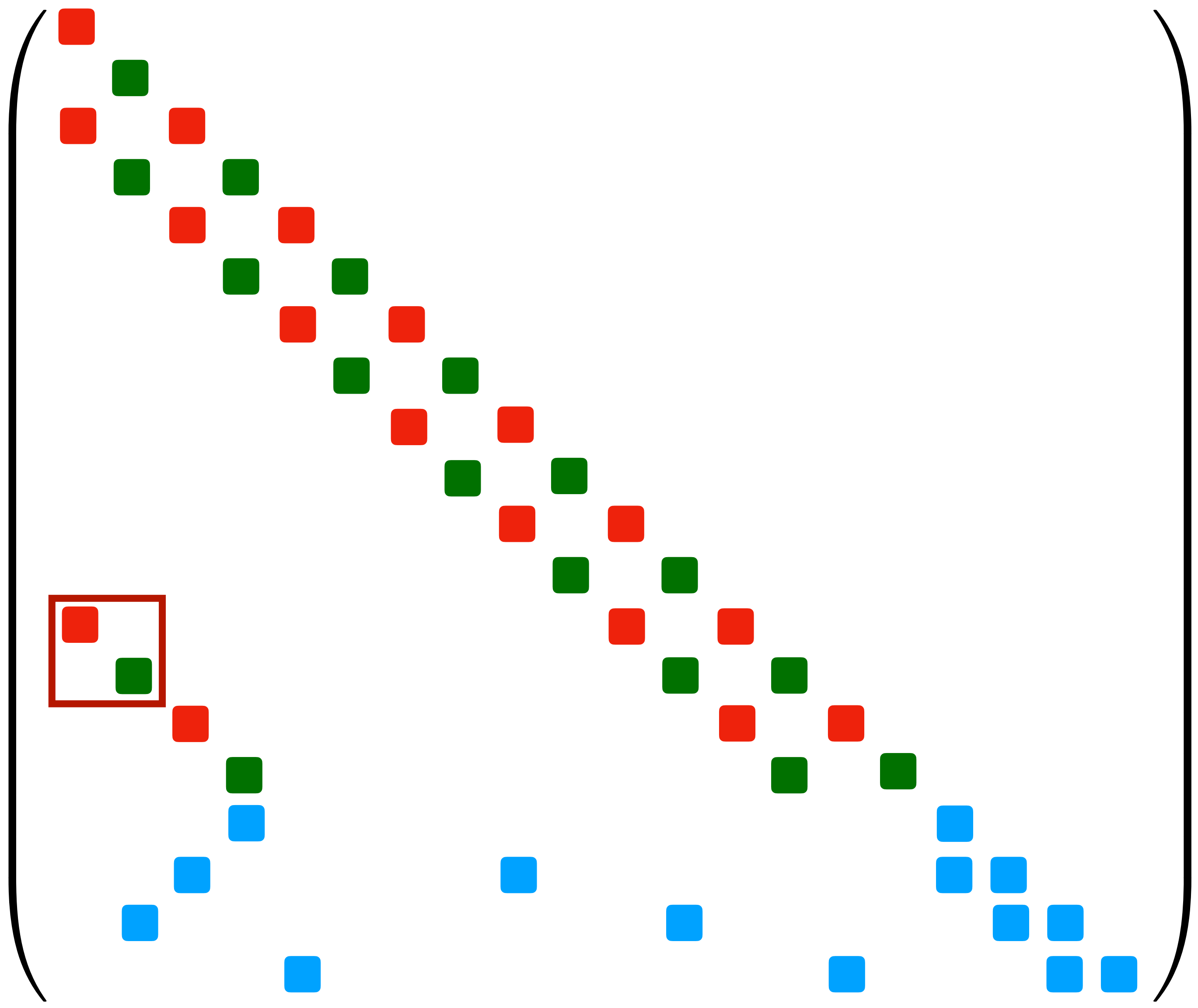
After reordering:
Sparsity pattern becomes a (sparse) collection of diagonal ***blocks***

Engineering/Maximizing Sparsity



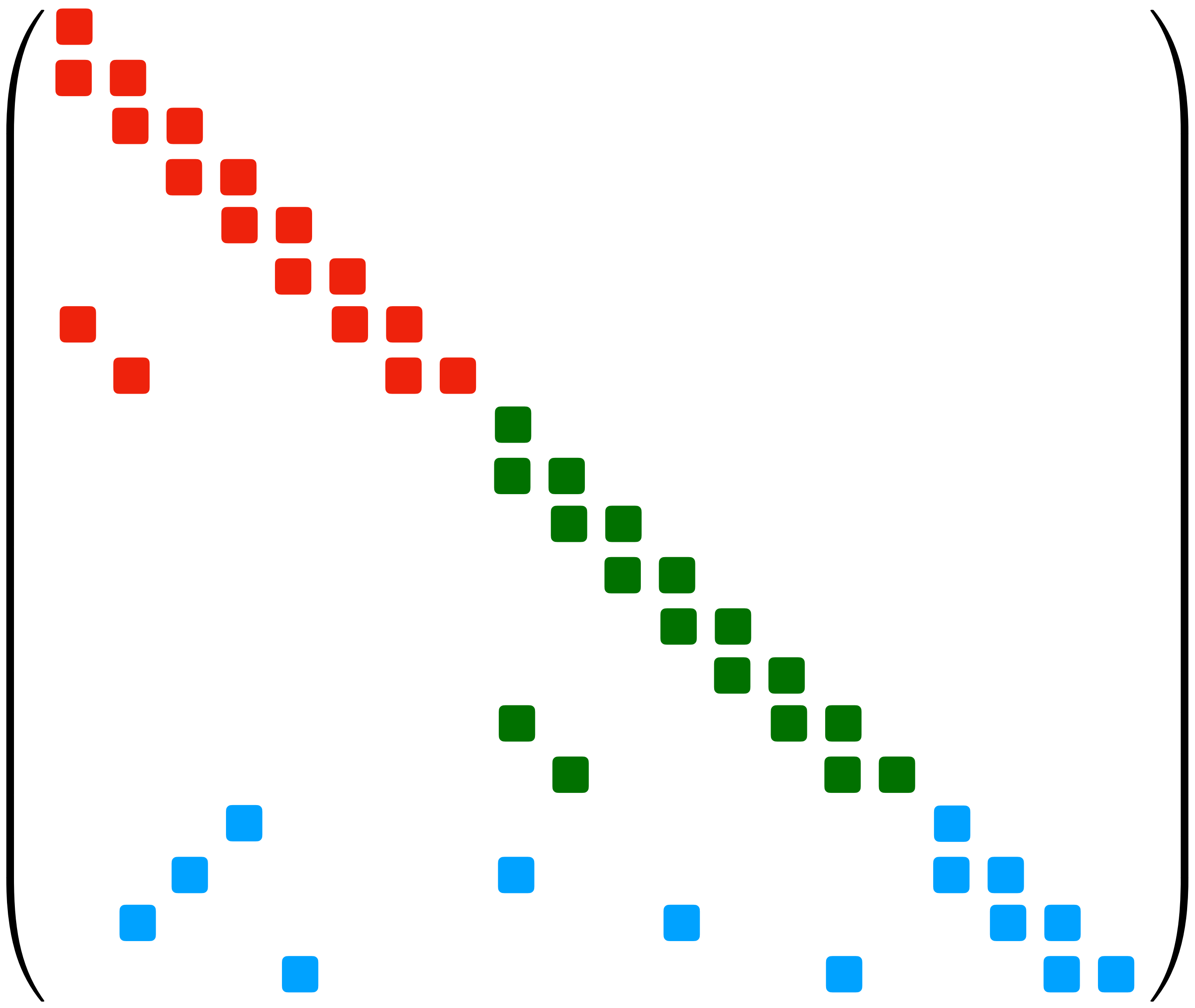
After reordering:
Sparsity pattern becomes a (sparse) collection of diagonal ***blocks***

Engineering/Maximizing Sparsity



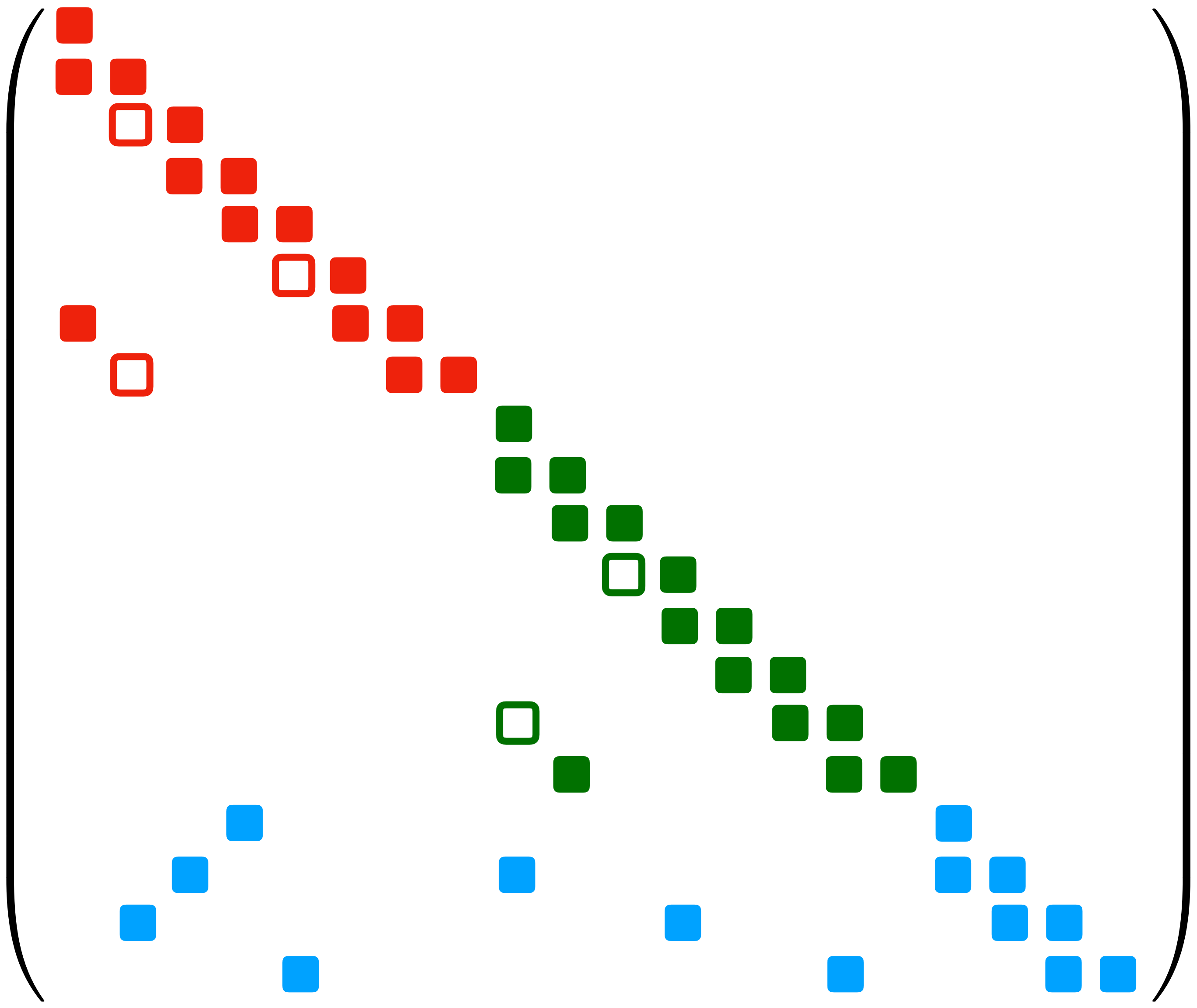
After reordering:
Sparsity pattern becomes a (sparse) collection of diagonal ***blocks***

Engineering/Maximizing Sparsity



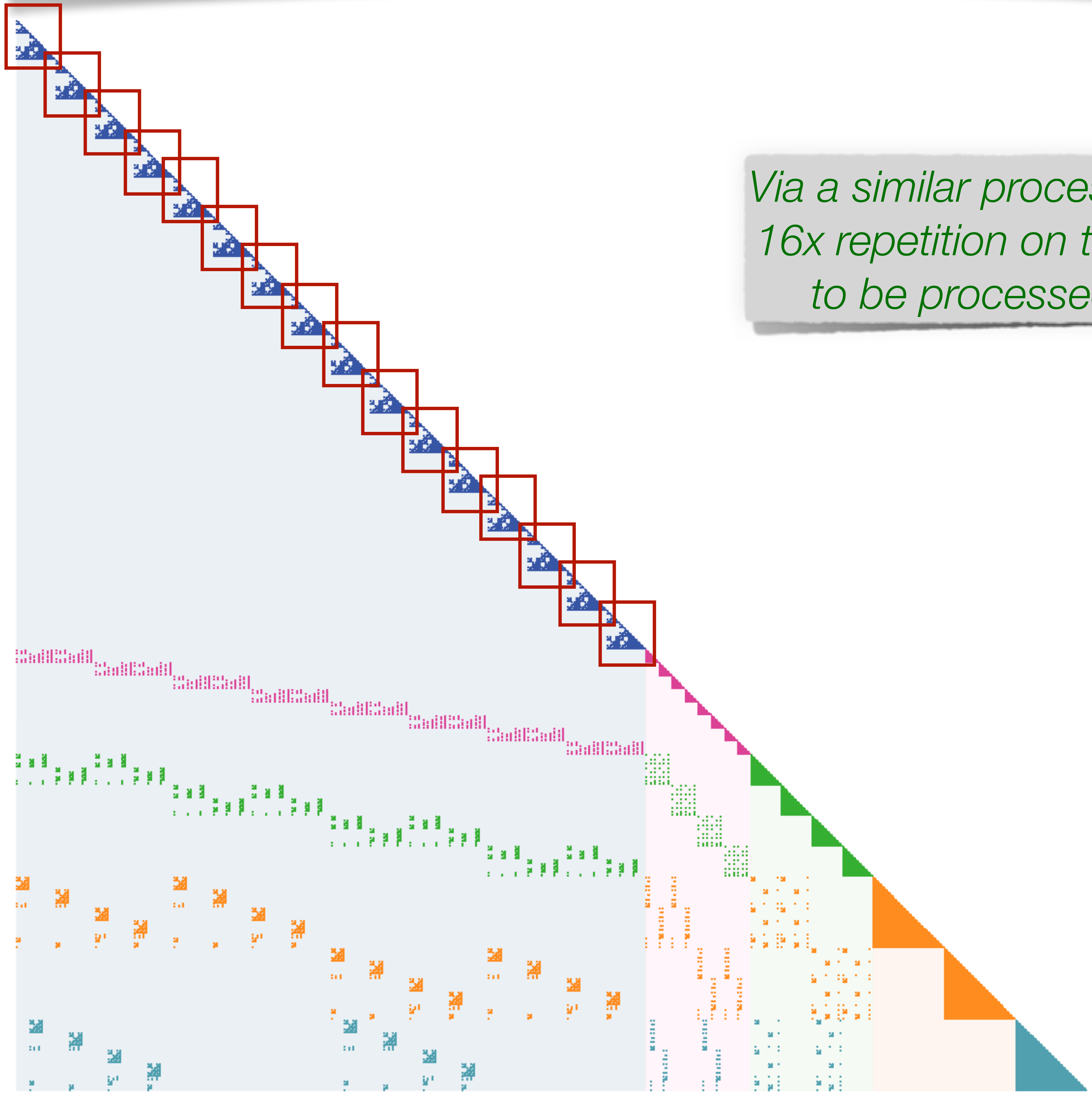
This transformation was predicated on the grid “partitions” having the same sparsity pattern …

Engineering/Maximizing Sparsity



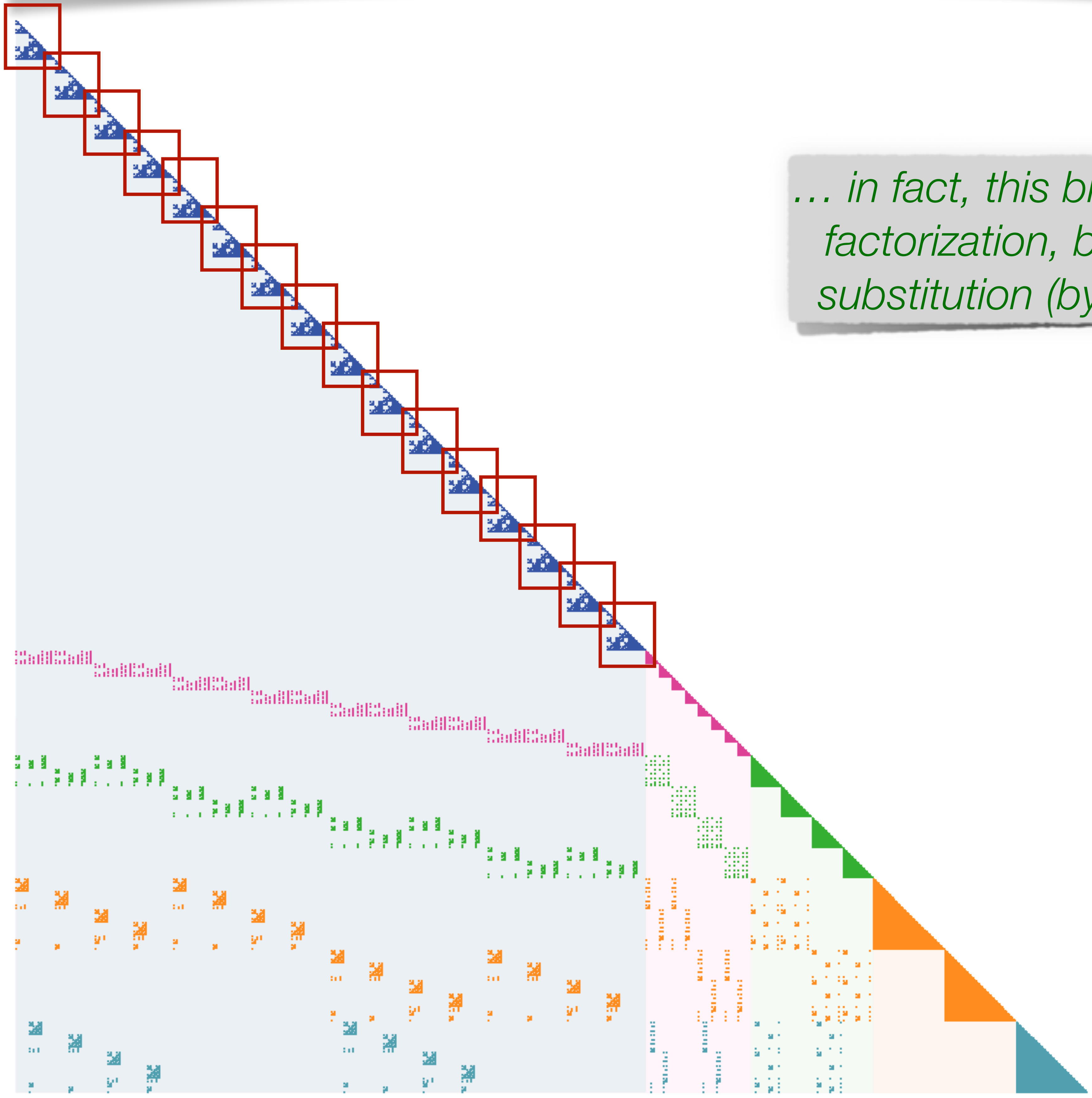
… but can be made to work even if the sparsity patterns are "almost" the same (this near-similarity needs to be discovered…)

Laplacian - Pattern after a possible reordering



Via a similar process … any patterns that exhibit 16x repetition on this layout, can be engineered to be processed in parallel using SIMD …

Laplacian - Pattern after a possible reordering



… in fact, this blocking accelerates not only factorization, but also forward/backward substitution (by allowing SIMD operation)

PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
    // Numerical factorization
    phase = 22;
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during numerical factorization");

    std::cout << "Factorization completed ... " << std::endl;

    // Back substitution and iterative refinement
    phase = 33;
    iparm[7] = 0;          // Max numbers of iterative refinement steps
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, static_cast<void*>(&f[0][0][0]), &x[0][0][0], &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during solution phase");

    std::cout << "Solve completed ... " <<std::endl;

    // Termination and release of memory.
    phase = -1;            // Release internal memory
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        &ddum, matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);

    if (writeOutput) WriteAsImage("x", x, 0, 0, XDIM/2);
}
```

PARDISO solver (DirectSolver.cpp)

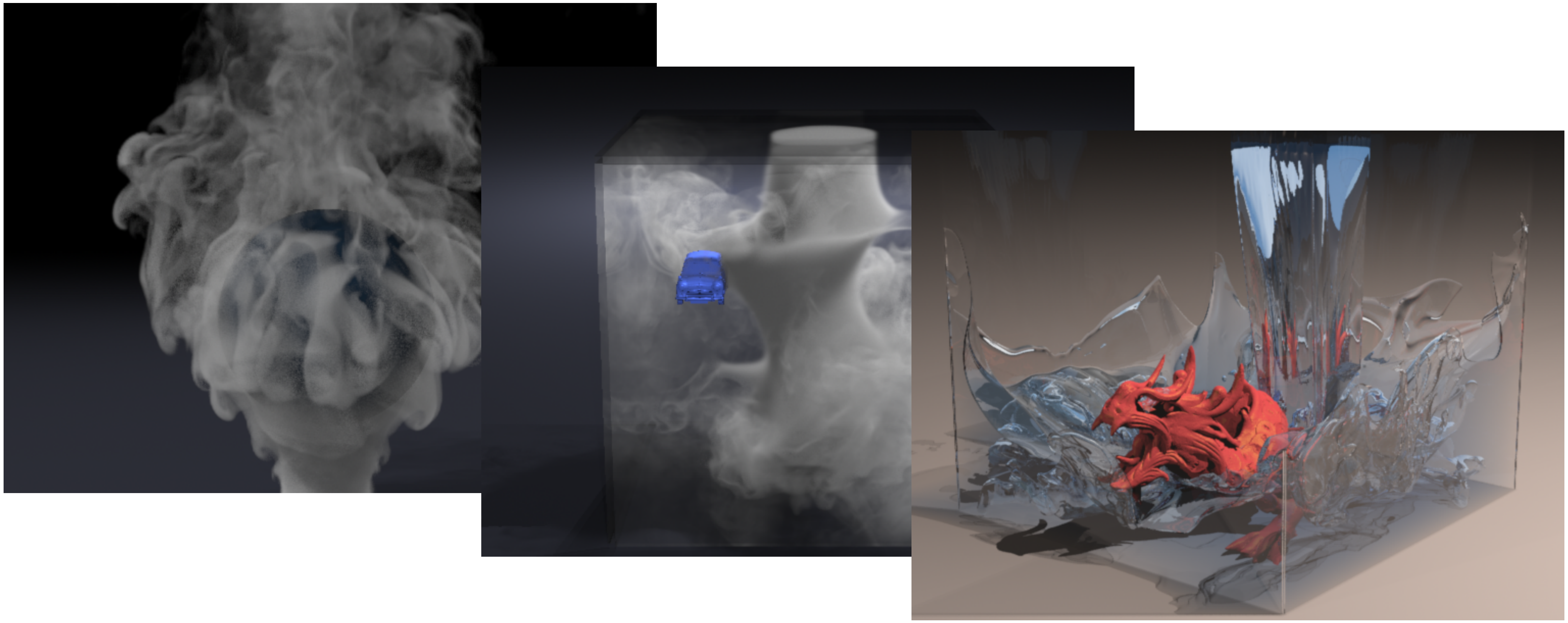
SparseDirect/LaplacePARDISO_0_0

Execution:

```
Summary: ( solution phase )
================
Times:
======
Time spent in direct solver at solve step (solve)                  : 0.463208 s
Time spent in additional calculations                              : 0.021776 s
Total time spent                                                   : 0.484984 s
Statistics:
===========
Parallel Direct Factorization is running on 20 OpenMP
< Linear system Ax = b >
             number of equations:           2097152
             number of non-zeros in A:      8050652
             number of non-zeros in A (%): 0.000183
             number of right-hand sides:    1
< Factors L and U >
             number of columns for each panel: 96
             number of independent subgraphs:  0
             number of supernodes:                       1407769
             size of largest supernode:                  16591
             number of non-zeros in L:                   2080602470
             number of non-zeros in U:                   1
             number of non-zeros in L+U:                 2080602471
             gflop   for the numerical factorization: 23028.583984
             gflop/s for the numerical factorization: 512.041504
```

Almost <1-2% of the factorization cost (which is what we hope!)

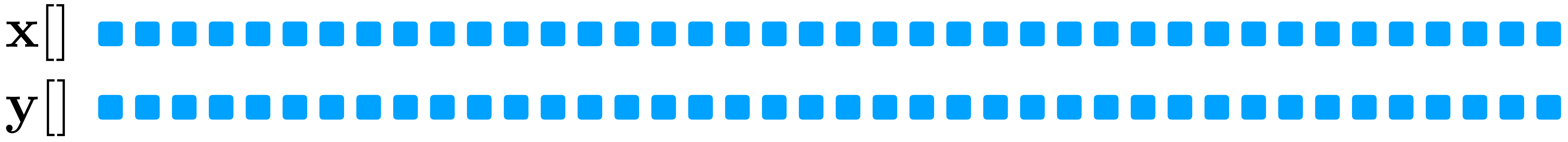
```
iparm[7] = 0;          // Max numbers of iterative refinement steps
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```



Parallel Sparse Direct Solvers
Performance & design of MKL PARDISO (wrap-up)
+ A few concluding notes on memory prefetching

(Dense) Saxpy

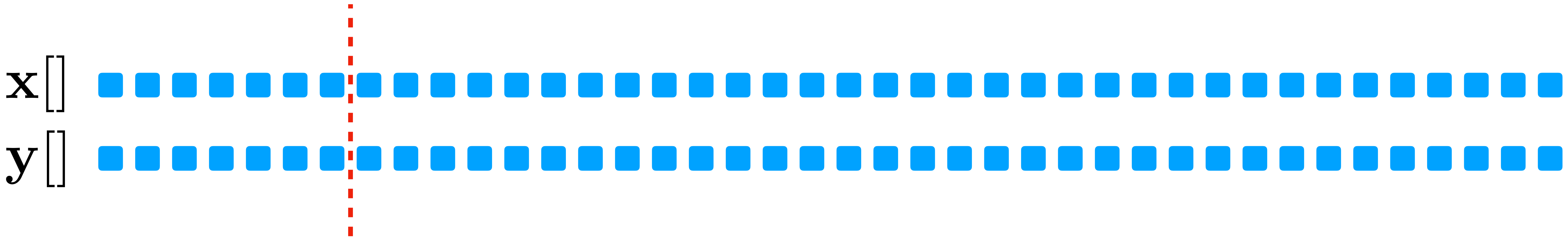
```
for  $i = 0, \ldots, N$
    $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```



(Dense) Saxpy

```
for  $i = 0, \ldots, N$
    $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```

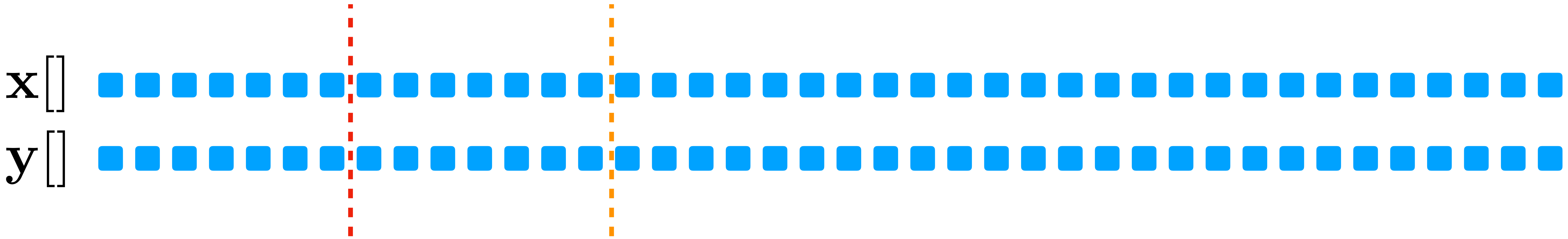
For full-bandwidth use:
If computation is here …



(Dense) Saxpy

```
for $i = 0, \ldots, N$
    $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```

For full-bandwidth use:
If computation is here …



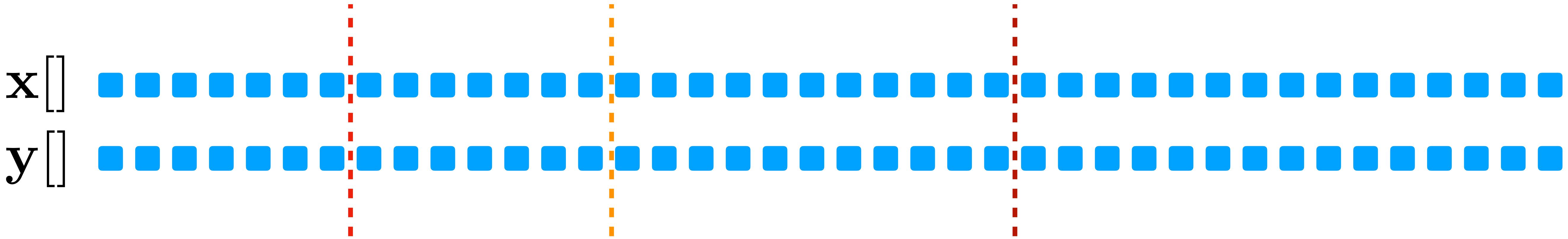
Then, data up to here better be in L1 Cache …

(Dense) Saxpy

```
for $i = 0, \ldots, N$
    $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```

For full-bandwidth use:
If computation is here …

… and everything up to here already in L2 cache

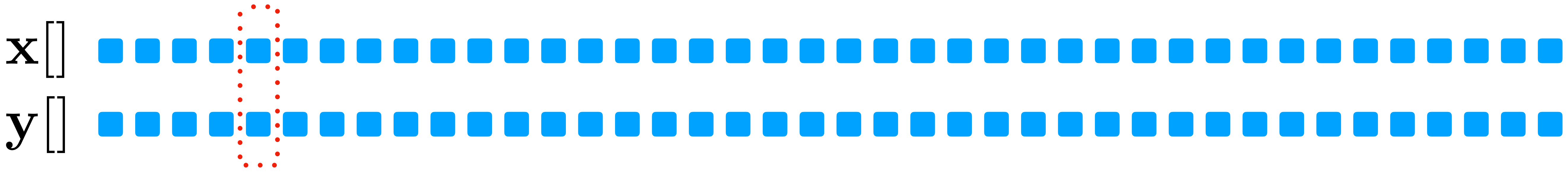


Then, data up to here better be in L1 Cache …

(Dense) Saxpy

```
for  $i = 0, \ldots, N$
    $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```

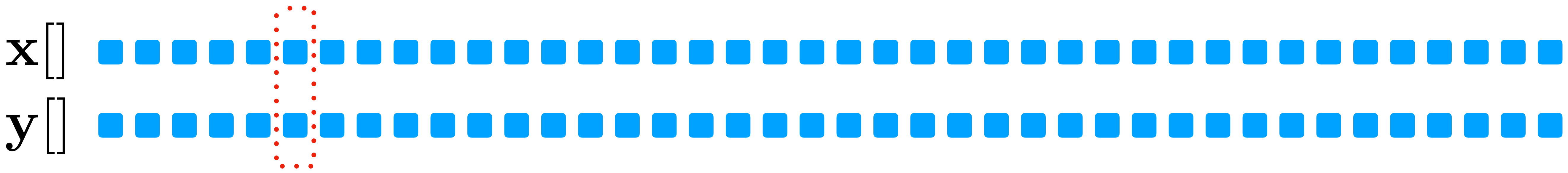
Hardware prefetching:
If following a certain
stride while accessing memory



(Dense) Saxpy

$$\texttt{for} \quad i = 0, \ldots, N$$
$$\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$$

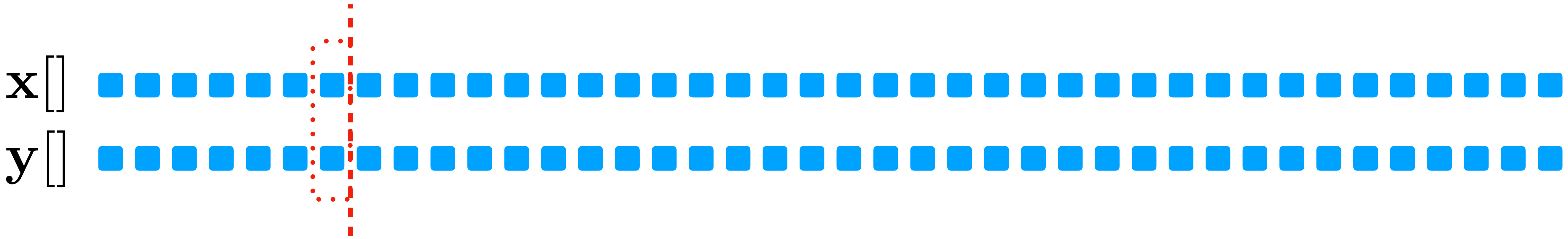
Hardware prefetching:
If following a certain
stride while accessing memory



(Dense) Saxpy

```
for  $i = 0, \dots, N$
        $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```

Hardware prefetching:
If following a certain
stride while accessing memory

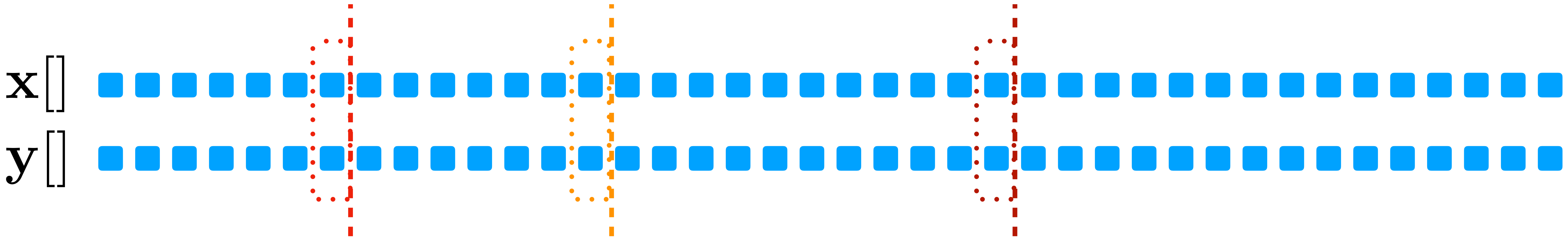


(Dense) Saxpy

```
for $i = 0, \ldots, N$
    $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```

Hardware prefetching:
If following a certain
stride while accessing memory

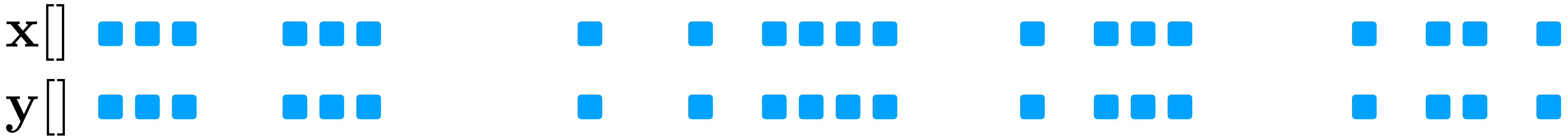
… the CPU automatically "looks ahead"
and prefetches according to the same
("apparent") stride into caches



Sparse Saxpy

```
for $i =$ (some indices)
            $\mathbf{y}[i] \mathrel{+}= \alpha \mathbf{x}[i]$
```

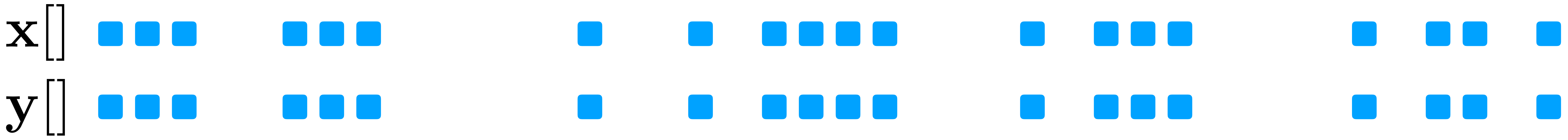
Effective hardware prefetching is hard:
- We don't know what to prefetch
- Even if we guess, good chance what's prefetched will be wasted



Our specific benchmark : Indirectly indexed Saxpy

```
for $i = 0, \ldots, N$
    $\mathbf{y}[\text{offset}[i]] \mathrel{+}= \alpha \mathbf{x}[\text{offset}[i]]$
```

Indices originate from array ***offsets[]***
- There *<u>is</u>* *a logic of where to prefetch from (the offsets array has that information)*
- But the compiler/CPU cannot infer that; the user might have to help



Main routine (main.cpp)

Prefetching/SparseSaxpy_0_0

```
#include "Timer.h"
#include "Utilities.h"
#include "PointwiseOps.h"

int main(int argc, char *argv[])
{
    std::vector<int> blockOffsets;
    float *x;
    float *y;

    InitializeArrays(blockOffsets, x, y);

    // Initialization
    for (int run = 0; run < 30; run++)
    {
        Timer timer;
        timer.Start();
        SparseSaxpy(blockOffsets, x, y, 3.14f);
        timer.Stop("SparseSaxpy time : ");
    }

    return 0;
}
```

Prefetching/SparseSaxpy_0_0

Initialization utilities (Utilities.h/cpp)

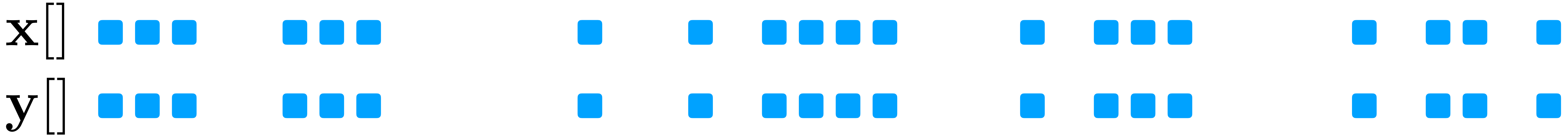
```
#pragma once

#include <vector>

#include "Parameters.h"

void* AlignedAllocate(const std::size_t size, const std::size_t alignment);
void InitializeArrays(std::vector<int>& blockOffsets, float *&x, float *&y);
```

Benchmark Parameters (Parameters.h)

```
#pragma once

#define BLOCK_SIZE 16
#define MAX_CLUSTER_SIZE 4
#define MAX_CLUSTER_DISTANCE 32
#define NUMBER_OF_BLOCKS 4*1024*1024
```

Sparse Saxpy

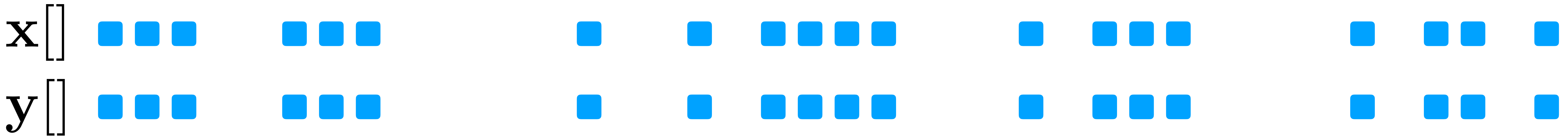
Our test collection of array entries comes in chunks of aligned 16-tuples (for simplicity)
Each "square" in the illustration below corresponds to 16-contiguous entries
(16 = BLOCK_SIZE in Parameters.h)



Sparse Saxpy

*Array **blockOffsets[]** contains the location of where each block-of-16 entries starts*

MAX_CLUSTER_SIZE is the maximum of how many blocks to “bundle/cluster” together (layout is randomly initialized)
while MAX_CLUSTER_DISTANCE is the average distance between block clusters



Prefetching/SparseSaxpy_0_0

Stock saxpy routine (PointwiseOps.cpp)

```
#include "PointwiseOps.h"

void SparseSaxpy(std::vector<int>& blockOffsets, const float *x, float *y, const float scale)
{
#pragma omp parallel for
    for (int b = 0; b < blockOffsets.size(); b++)
        for (int i = 0; i < BLOCK_SIZE; i++)
            y[blockOffsets[b]+i] += scale * x[blockOffsets[b]+i];
}
```

Prefetching/SparseSaxpy_0_0

Stock saxpy routine (PointwiseOps.cpp)

```
#include "

void Spars                                                                      at scale)
{
#pragma om
    for (i
        fo

}
```

Execution:

```
Allocated total of 4194304 blocks (67108864 entries; 256MB of actual data)
   in a span of 1946.55MB
[SparseSaxpy time : 33.5354ms]
[SparseSaxpy time : 25.3708ms]
[SparseSaxpy time : 25.3139ms]
[SparseSaxpy time : 24.32ms]
[SparseSaxpy time : 25.3662ms]
[SparseSaxpy time : 24.3337ms]
[SparseSaxpy time : 24.3135ms]
[SparseSaxpy time : 26.3057ms]
[SparseSaxpy time : 25.3865ms]
[SparseSaxpy time : 24.3556ms]
[SparseSaxpy time : 25.3534ms]
[SparseSaxpy time : 24.1806ms]
[SparseSaxpy time : 24.1684ms]
[SparseSaxpy time : 25.1663ms]
[SparseSaxpy time : 24.1898ms]
[SparseSaxpy time : 25.138ms]
```

Prefetching/SparseSaxpy_0_1

Stock saxpy routine (PointwiseOps.cpp)

```
#include "PointwiseOps.h"
#include "immintrin.h"

void SparseSaxpy(std::vector<int>& blockOffsets, const float *x, float *y, const float scale)
{
    static constexpr int L2_PREFETCH_DISTANCE = 64;
    static constexpr int L1_PREFETCH_DISTANCE = 8;

#pragma omp parallel for
    for (int b = 0; b < blockOffsets.size(); b++) {
        _mm_prefetch ( &x[blockOffsets[b+L2_PREFETCH_DISTANCE]], _MM_HINT_T2 );
        _mm_prefetch ( &x[blockOffsets[b+L1_PREFETCH_DISTANCE]], _MM_HINT_T1 );
        _mm_prefetch ( &y[blockOffsets[b+L2_PREFETCH_DISTANCE]], _MM_HINT_T2 );
        _mm_prefetch ( &y[blockOffsets[b+L1_PREFETCH_DISTANCE]], _MM_HINT_T1 );
#pragma omp simd
        for (int i = 0; i < BLOCK_SIZE; i++)
            y[blockOffsets[b]+i] += scale * x[blockOffsets[b]+i];
    }
}
```

Prefetching/SparseSaxpy_0_1

Stock saxpy routine (PointwiseOps.cpp)

```
#include "PointwiseOps.h"
#include "immintrin.h"

void SparseSaxpy(std::vector<int>& blockOffsets, const float *x, float *y, const float scale)
{
    static constexpr int L2_PREFETCH_DISTANCE = 64;
    static constexpr int L1_PREFETCH_DISTANCE = 8;

#pragma omp parallel for
    for (int b = 0; b < blockOffsets.size(); b++) {
        _mm_prefetch ( &x[blockOffsets[b+L2_PREFETCH_DISTANCE]], _MM_HINT_T2 );
        _mm_prefetch ( &x[blockOffsets[b+L1_PREFETCH_DISTANCE]], _MM_HINT_T1 );
        _mm_prefetch ( &y[blockOffsets[b+L2_PREFETCH_DISTANCE]], _MM_HINT_T2 );
        _mm_prefetch ( &y[blockOffsets[b+L1_PREFETCH_DISTANCE]], _MM_HINT_T1 );
#pragma omp simd
        for (int i = 0; i < BLOCK_SIZE; i++)
            y[blockOffsets[b]+i] += scale * x[blockOffsets[b]+i];
    }
}
```

We provide explicit prefetching hints for both L1 and L2 caches (note: prefetch typically does not fault if given an invalid memory)

Stock saxpy routine (PointwiseOps.cpp)

Prefetching/SparseSaxpy_0_1

```
#include "PointwiseOps.h"
#include "immintrin.h"

void Sparse                                                              scale)
{
    static
    static

#pragma omp
    for (in
        _mm
        _mm
        _mm
        _mm
#pragma omp
        for

    }
}
```

Execution:

```
Allocated total of 4194304 blocks (67108864 entries; 256MB of actual data)
in a span of 1945.54MB
[SparseSaxpy time : 21.6707ms]
[SparseSaxpy time : 12.3966ms]
[SparseSaxpy time : 12.3517ms]
[SparseSaxpy time : 12.3327ms]
[SparseSaxpy time : 12.3688ms]
[SparseSaxpy time : 12.3316ms]
[SparseSaxpy time : 12.333ms]
[SparseSaxpy time : 12.3355ms]
[SparseSaxpy time : 12.3285ms]
[SparseSaxpy time : 12.333ms]
[SparseSaxpy time : 12.3489ms]
[SparseSaxpy time : 12.3211ms]
[SparseSaxpy time : 12.3352ms]
[SparseSaxpy time : 12.3222ms]
[SparseSaxpy time : 12.3475ms]
[SparseSaxpy time : 12.3308ms]
```

Prefetching/SparseSaxpy_0_1

Stock saxpy routine (PointwiseOps.cpp)

```
#include "PointwiseOps.h"
#include "immintrin.h"

void Sparse                                                          scale)
{
    static
    static

#pragma omp
    for (in
        _mm
        _mm
        _mm
        _mm
#pragma omp
        for

    }
}
```

Execution:

```
Allocated total of 4194304 blocks (67108864 entries; 256MB of actual data)
in a span of 1945.54MB
[SparseSaxpy time : 21.6707ms]
[SparseSaxpy time : 12.3966ms]
[SparseSaxpy time : 12.3517ms]
[SparseSaxpy time : 12.3327ms]
[SparseSaxpy time : 12.3688ms]
[SparseSaxpy time : 12.3316ms]
[SparseSaxpy time : 12.333ms]
[SparseSaxpy time : 12.3355ms]
[SparseSaxpy time : 12.3285ms]
[SparseSaxpy time : 12.333ms]
[SparseSaxpy time : 12.3489ms]
[SparseSaxpy time : 12.3211ms]
[SparseSaxpy time : 12.3352ms]
[SparseSaxpy time : 12.3222ms]
[SparseSaxpy time : 12.3475ms]
[SparseSaxpy time : 12.3308ms]
```

Note: Performance boost is highly variable depending on compiler, CPU, optimization level, and context!