Georgia Tech College of Computing
Computational Science and Engineering

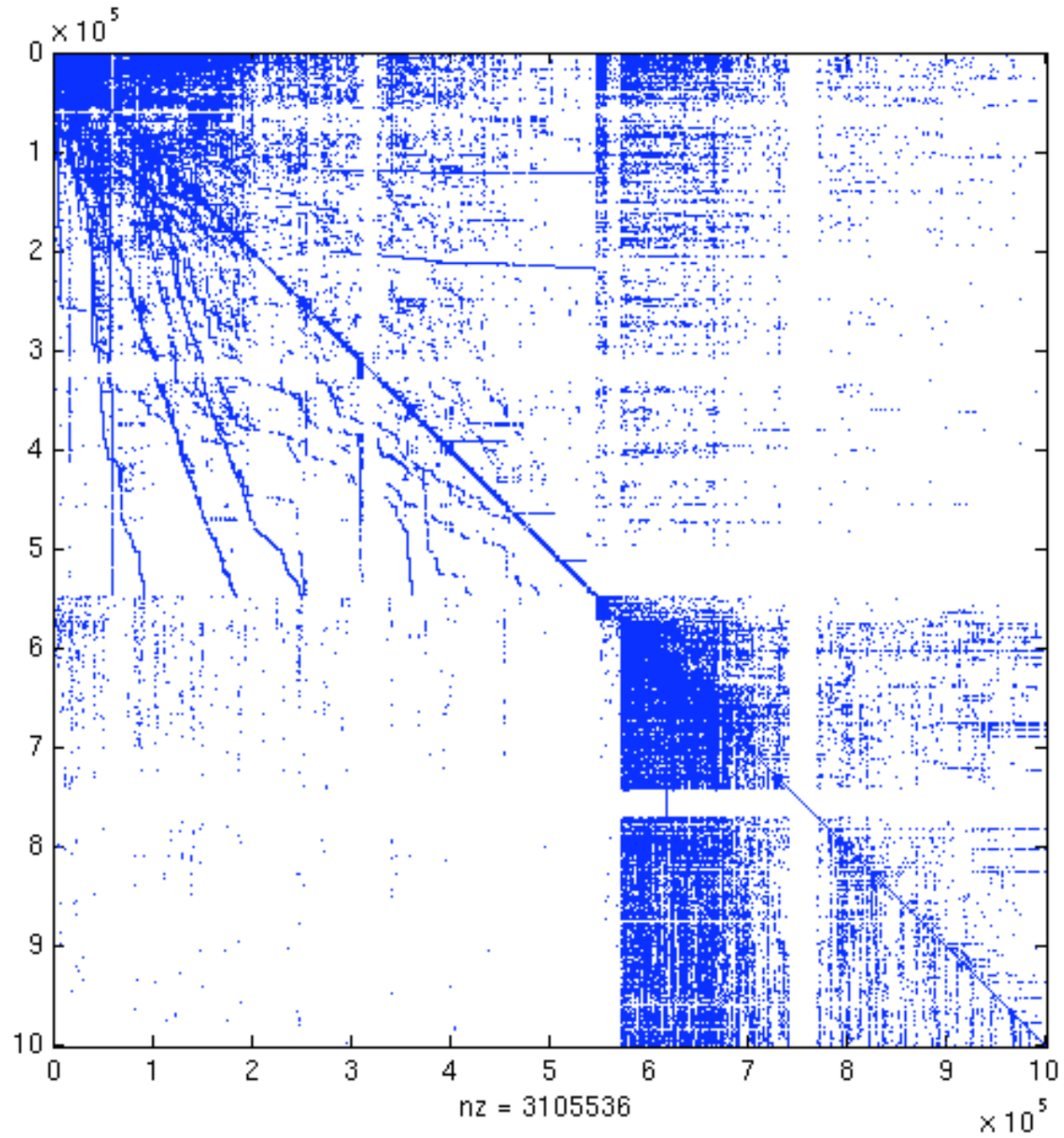
Transforming sparse matrix data structures

Richard Vuduc, Georgia Tech
James Demmel and Katherine Yelick, U.C. Berkeley
2008 Lua Workshop

Context: Part of my research, joint with U.C. Berkeley, explores the idea of building numerical libraries whose performance is tuned automatically for any machine and any user input. This talk focuses on some of our work in developing the Optimized Sparse Kernel Interface (OSKI), an "autotuned library" for sparse linear algebra kernels, such as sparse matrix-vector multiply, sparse triangular solve, and so on.

A major issue in the development of such "autotuned libraries" is how to provide transparency and control to the user; we use Lua to provide users with a high-level interface to the transformation engine. From this audience's technical perspective, the way we use Lua is straightforward--we are basically just providing high-level wrappers to some of OSKI's internal transformation infrastructure. Nevertheless, we feel Lua has been a very effective way for expressing the kinds of complex transformations we need in practice, and we hope this community will advise us on the ways in which we could push our use of Lua further.

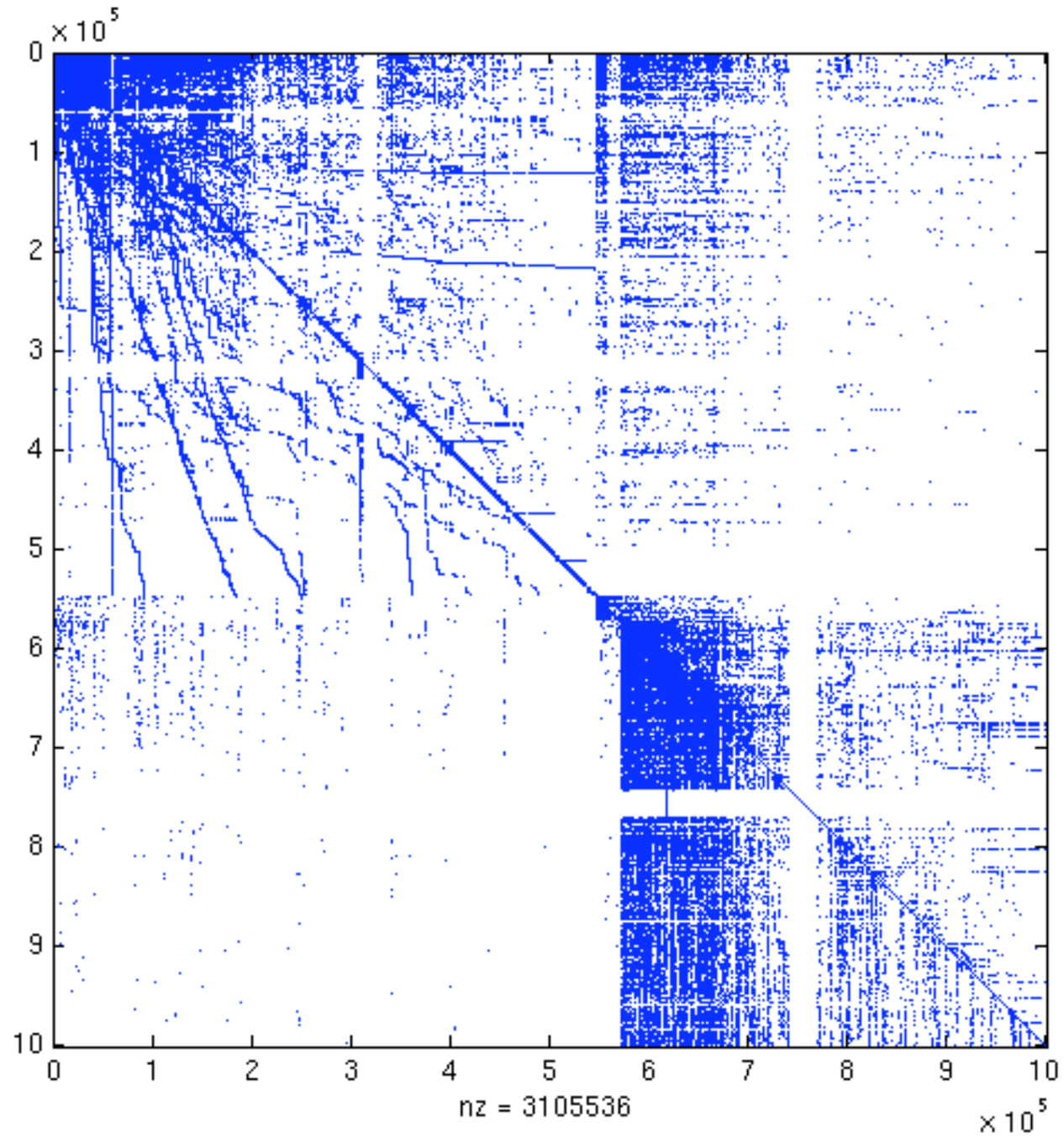
- Pop Quiz: **Who am I?**



This picture shows a piece of a very large sparse matrix. The piece shown is 1 million by 1 million and has ~ 3.1 million non-zeros values, and so is very sparse. Each non-zero is a blue dot; the white space shows regions of purely zero entries, on which we need not perform any operations.

Pop quiz: Can you guess what application gives rise to this particular sparse matrix?

- Answer: **Web connectivity graph** (partial)
- Factoid: Google PageRank is world's largest eigenproblem



Each row (and corresponding column) represents a web page. There is a non-zero (blue dot) at position (i, j) in the matrix if page i links to page j. Thus, the matrix represents the web connectivity graph. Indeed, in Google's PageRank algorithm for computing the order in which to return the list of web pages during a search query, the matrix is a probability transition matrix; PageRank computes a ranking by multiply a perturbation of this matrix by itself over and over again, which it turns out is equivalent to computing the principle eigenvector of a sparse matrix using an algorithm known as the "power method." Given there are O(10-100 billion) web pages, this matrix is quite large, but also quite sparse.

Overview

- Sparse matrix kernels abound
  - Apps: Physics, finance, PageRank, ...
  - Ops: Matrix-vector multiply, tri. solve, …
  - Speed is machine-dependent, hard-to-predict
- Our research: **Automatic tuning**
  - Given: Matrix, machine
  - Goal: Select "best" data structure *at run-time*
  - Implementation: OSKI library (4x)
- **Use Lua to express transformations**

PageRank is one particularly sexy application of sparse linear algebra, but "classical" applications include modeling and simulation in sciences, engineering, and finance. The kinds of sparse matrix operations, or "kernels," in which we are particularly interested include matrix-vector multiply, triangular solve, among others.

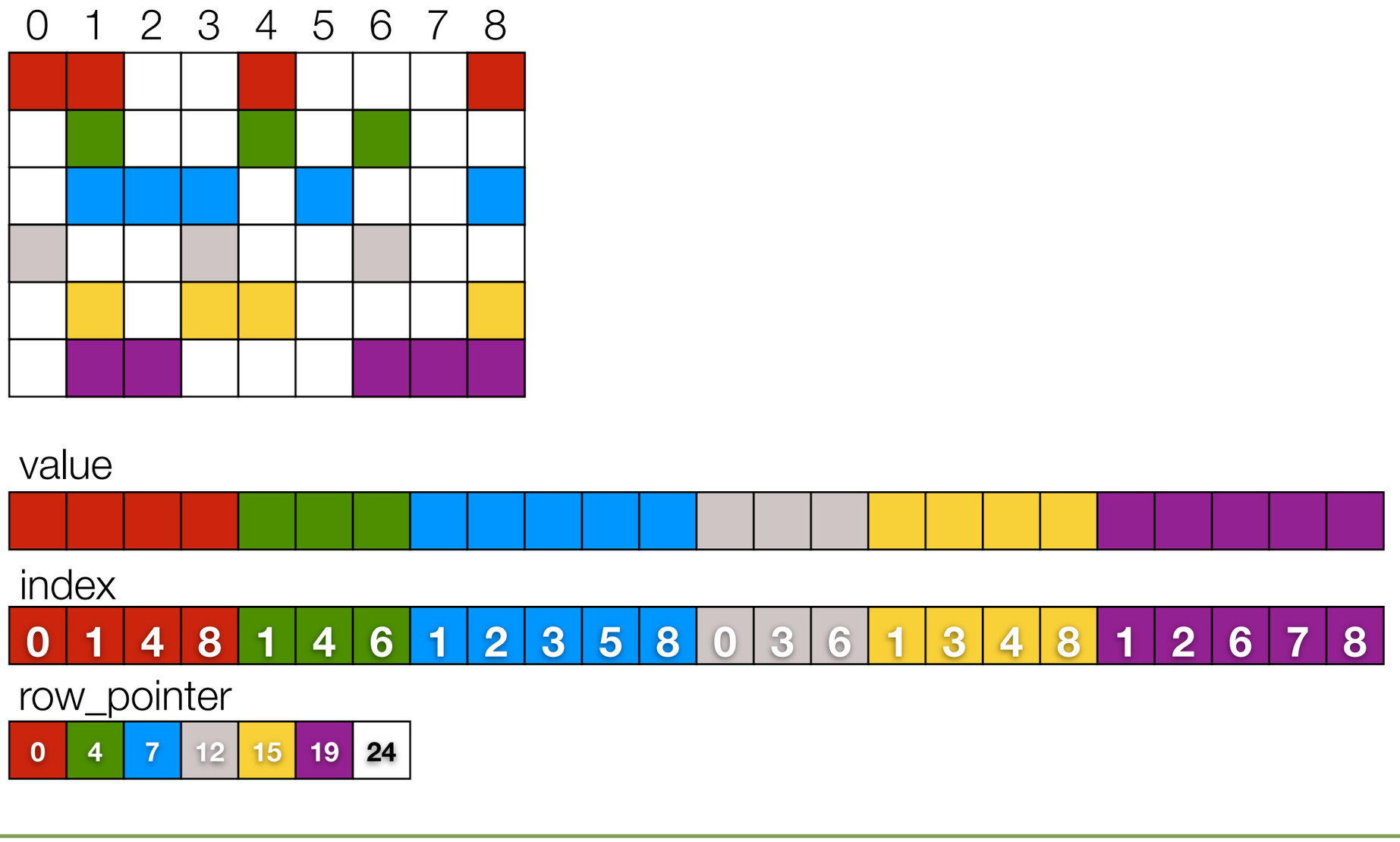
We have found that the performance (speed) of a sparse kernel can be a surprising and hard-to-predict function of both the user's machine and the user's matrix. The goal of our research is to attain high-performance automatically, for any machine and matrix. Achieving high-performance amounts to selecting the right data structure to store the sparse matrix, a task which we may have to carry out at run-time since the matrix may be unknown until then. Over many years, we have been developing data structures and techniques for tuning sparse matrices, and have implemented these ideas in a library called the Optimized Sparse Kernel Interface (OSKI).

Although OSKI transforms the data structure automatically, sometimes the user knows better, or perhaps the user wants to know what OSKI decided, to apply to future matrices. We use an embedded scripting language based on Lua to communicate OSKI's transformations to the user, and to also allow the user to drive the transformations.

1. The need flexible data structure selection

The goal of the first section of this talk is to convince you that there is a pressing need for flexible data structure selection.

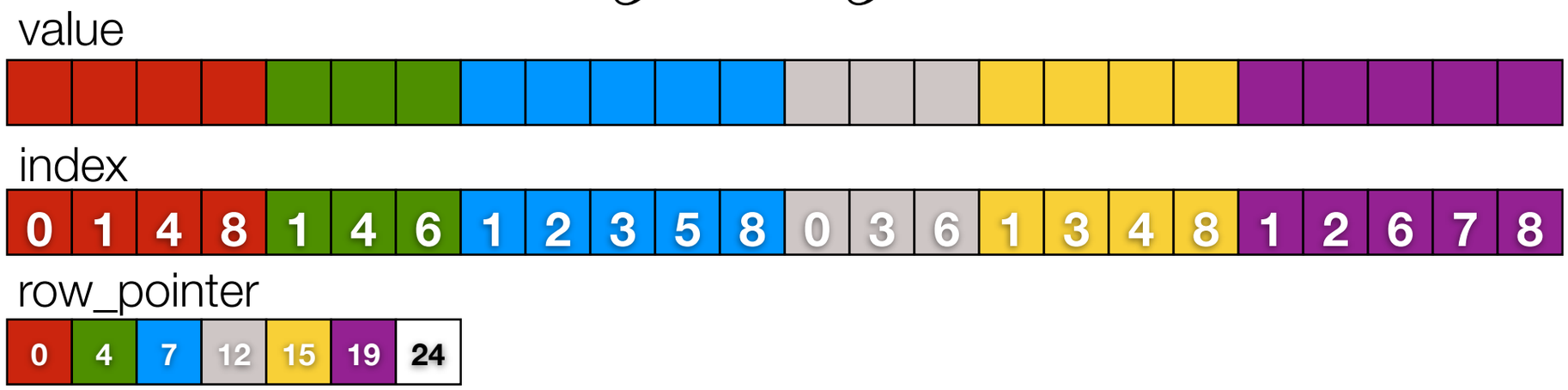
Compressed sparse row (CSR) format



The canonical sparse matrix storage format is known as “compressed sparse row.” The non-zeros in each row are packed together, and laid out row-by-row in a “value” array. For each entry in the value array, we record the corresponding column index in the “index” array. Finally, we maintain a row of pointers, “row_pointer,” to mark the start of each row in the packed index/value arrays.

Sparse matrix-vector multiply (SpMV)

$$y \leftarrow y + A \cdot x$$

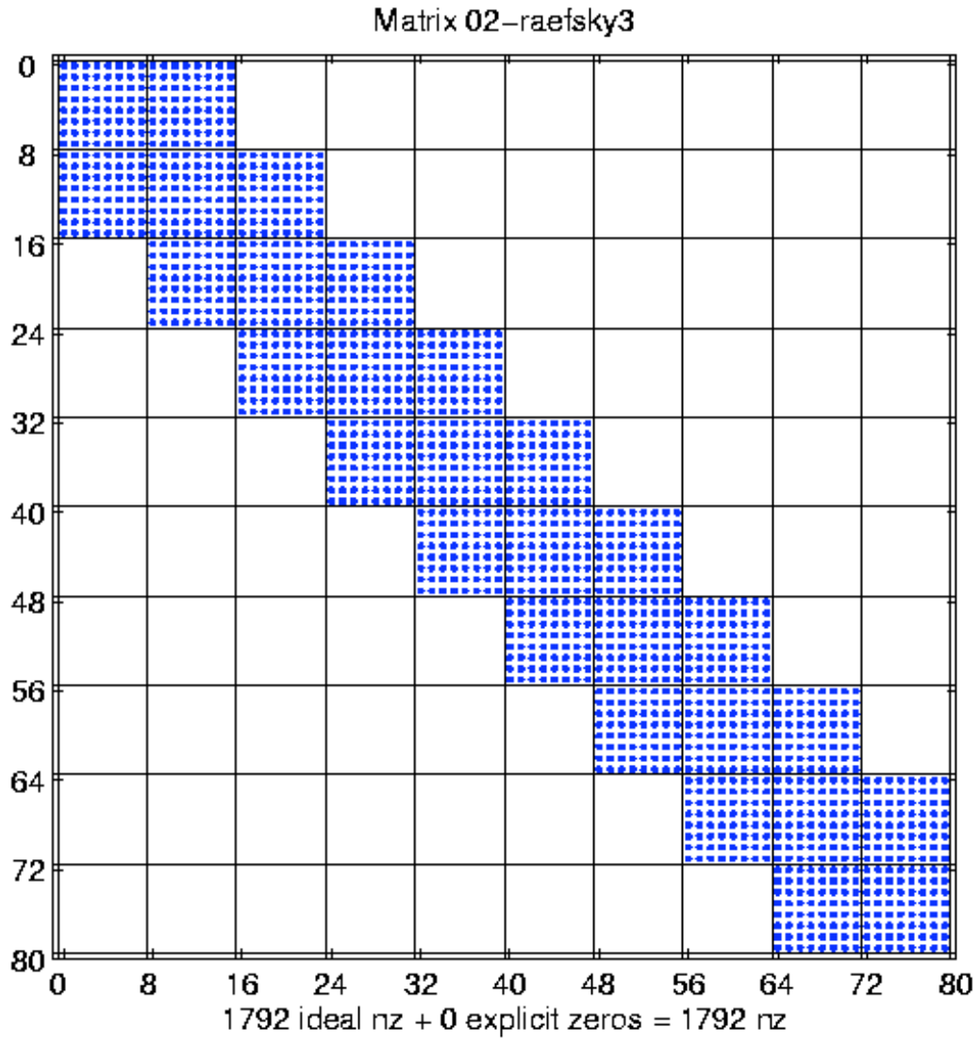


- Low computational intensity, vs. dense matrices
- Serial performance ~ 10% peak or less
- Bandwidth limited → compress
- Eliminate all indices → expect 1.5x speedup (32b ints, 64b vals)

The most commonly used sparse kernel is an operation known as a “sparse matrix-times-dense vector multiply”, or SpMV for short. There are several key facts to note about SpMV.

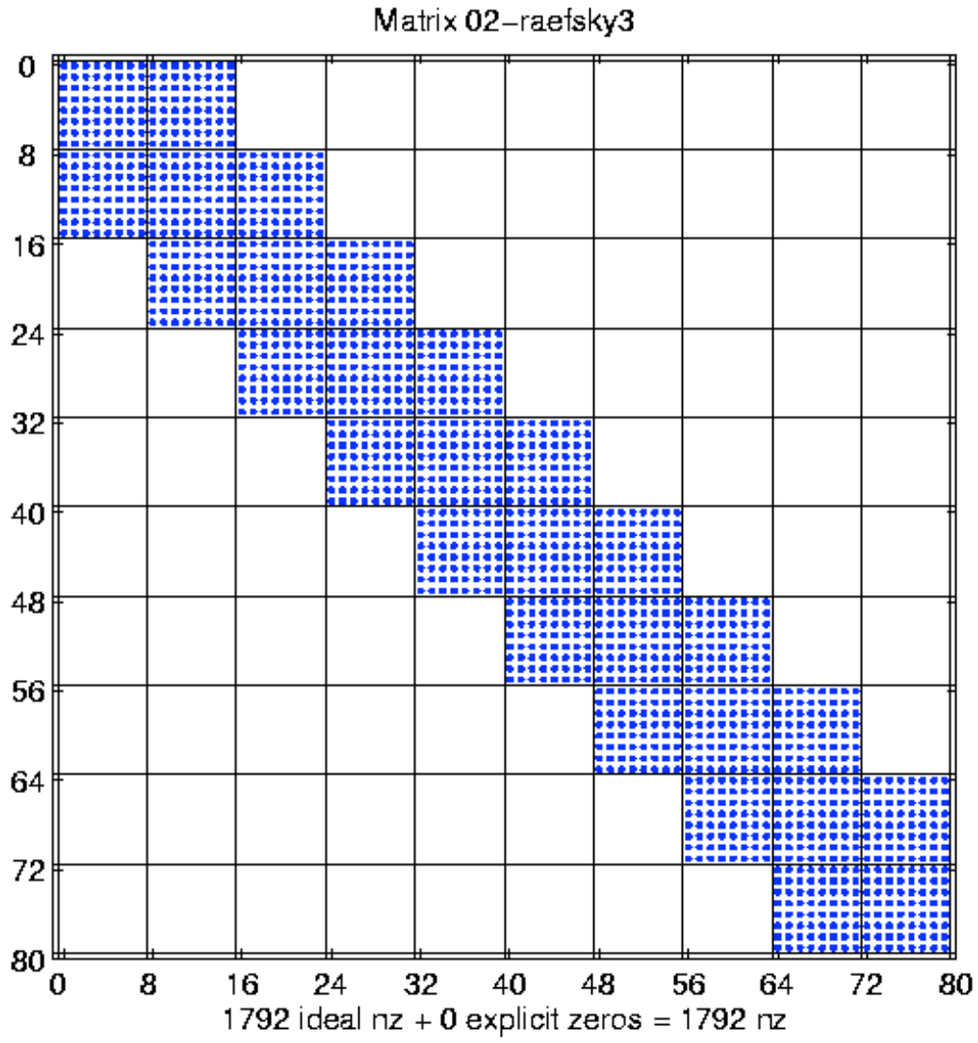
First, SpMV has a relatively low flop-to-memory ratio (particularly compared to dense linear algebra kernels like dense matrix-matrix multiply), yielding sequential performance of 10% of machine peak or less.

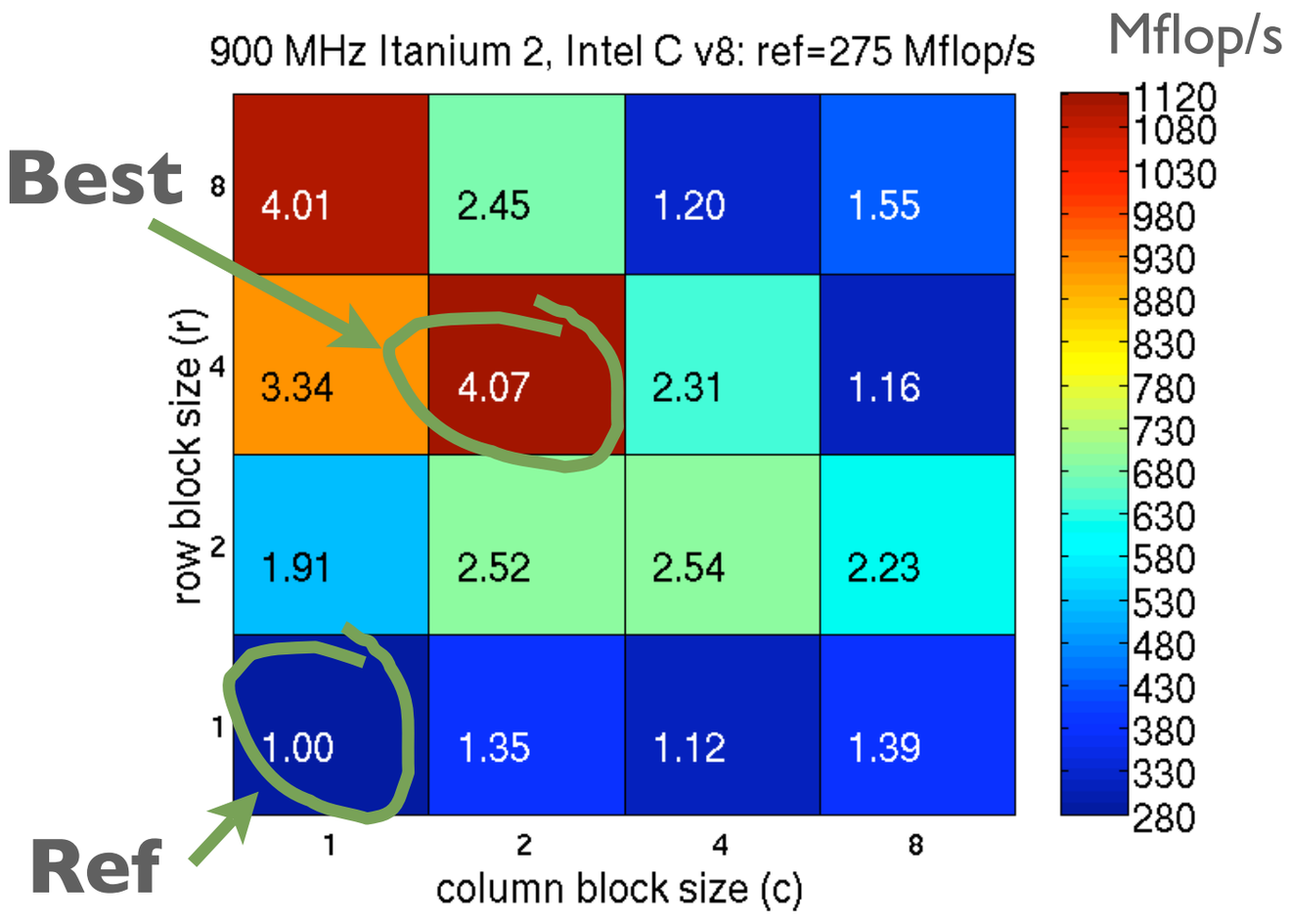
Secondly, its speed is, to first order, limited essentially by the time to read the matrix from memory to the CPU. This fact suggests that one strategy for accelerating SpMV is simply to reduce the size of the data structure, i.e., compress it by, say, recognizing patterns in the matrix and thereby eliminating the need to store some of the indices and/or values. For example, if we could somehow eliminate all the indices in CSR, we might expect at most a 1.5x speedup if we are using 32b ints and 64b values.



Regular structure in real life

In real applications, there is often lots of regular structure to exploit, in order to compress the matrix. For example, this matrix, which comes from a NASA structural engineering application, is full of little dense 8x8 blocks. (This picture only shows a piece of the much larger matrix, but the pattern is regular.) So, rather than store 1 index per non-zero, we could store 1 index per 8x8 block, thereby reducing the index overhead significantly.



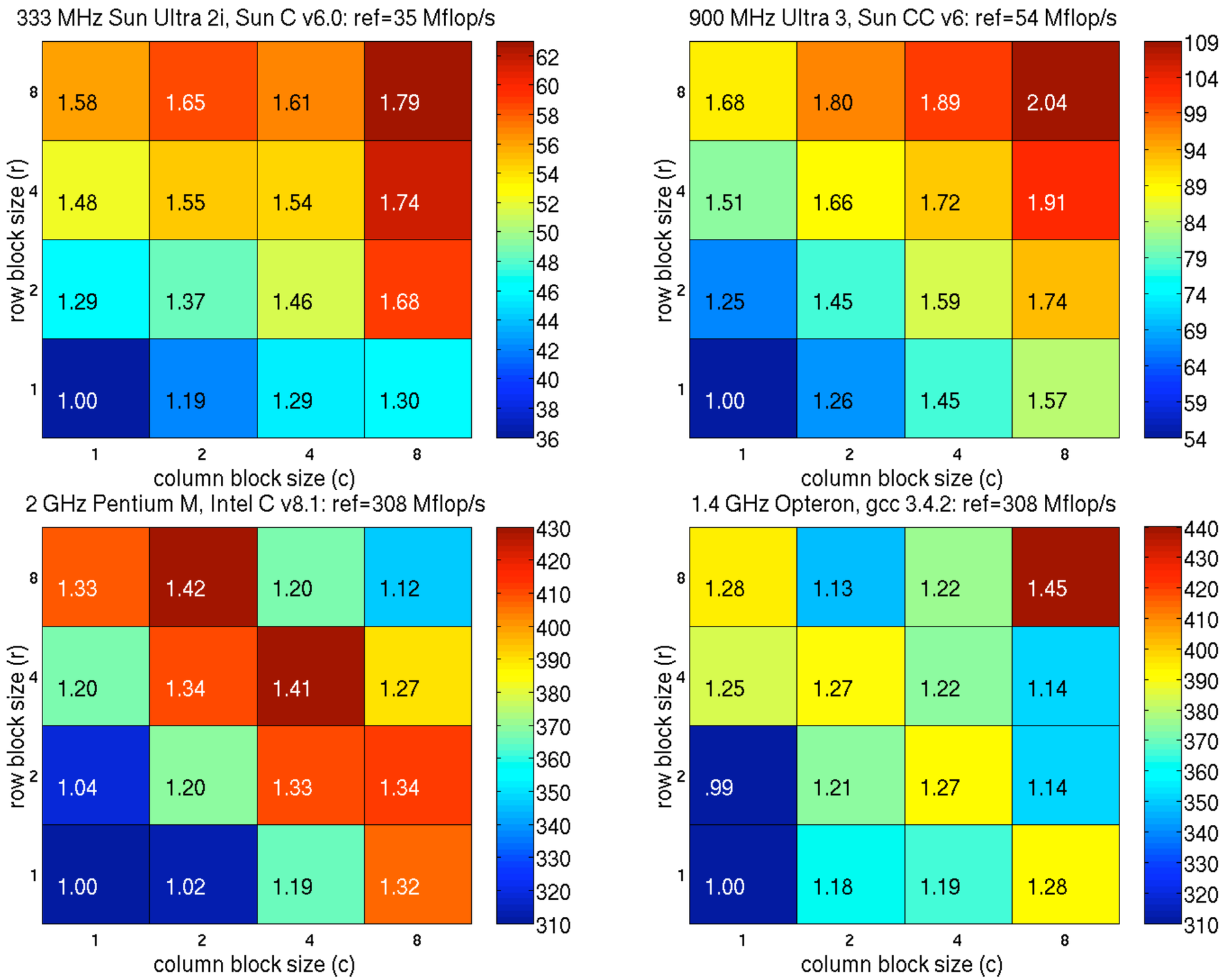


Optimal block size not obvious

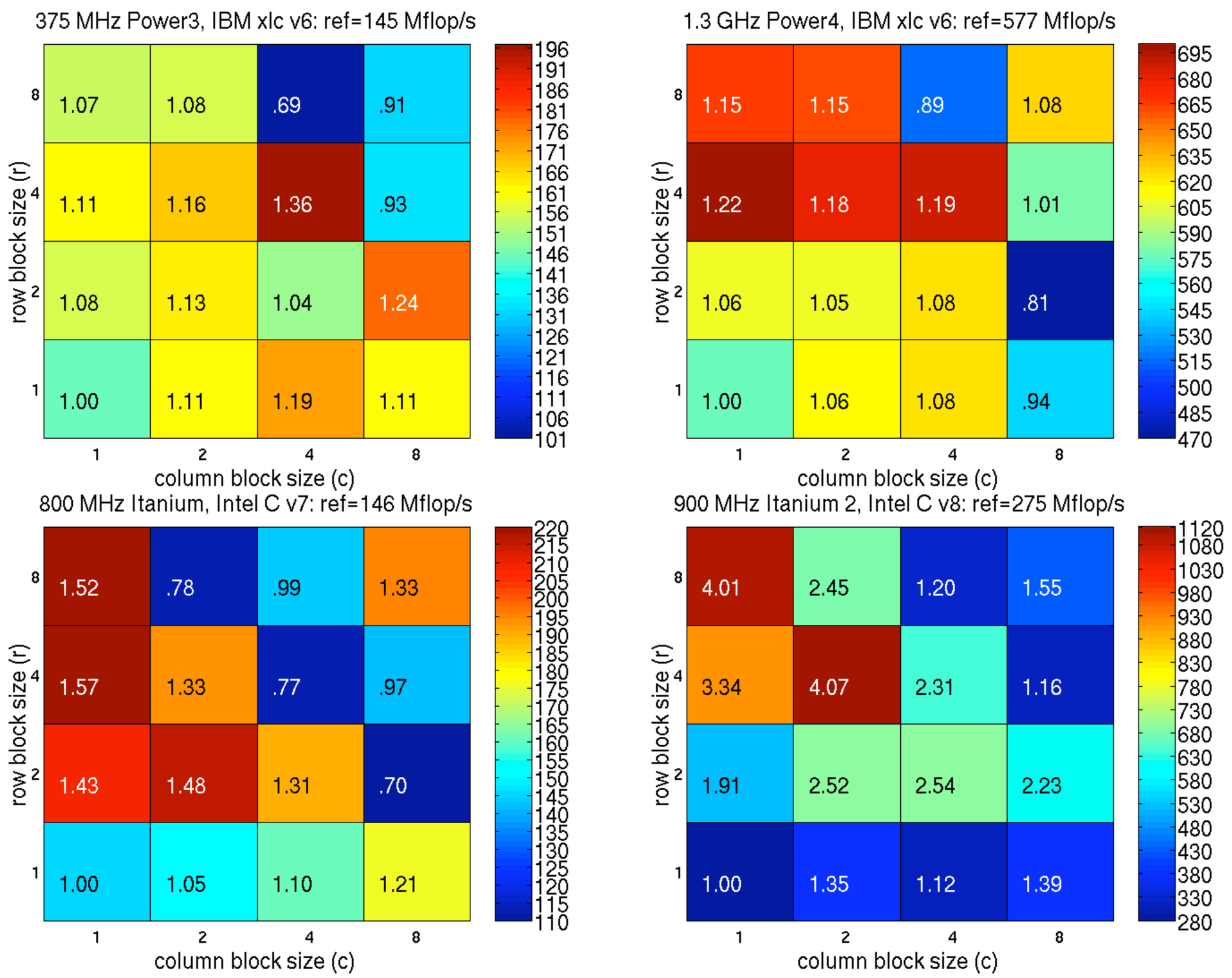
In practice on real machines, however, the best “block size” for such a matrix may not be obvious.

For this matrix, we did an experiment on a 900 MHz Intel Itanium 2-based machine in which we measured the performance of CSR, an 8x8 blocked variant, as well as other block sizes that might make sense, including 2x1, 4x8, etc., or 16 implementations in all. On the right, we show performance as a function of block size. Performance is measured in millions of floating-point operations per second (Mflop/s), and color coded from slow (blue) to fast (red). In addition, I’ve labeled each implementation by its speedup relative to CSR. The CSR (1x1) code runs at ~ 280 Mflop/s (less than 8% of peak on this machine), while the 8x8 variant does indeed deliver an ~ 1.5x speedup as we might expect.

However, the best implementation is actually much faster: the 4x2 version runs at over 1 Gflop/s, which roughly a third of peak on this machine! This is a surprise and would not match the intuition I gave you before.



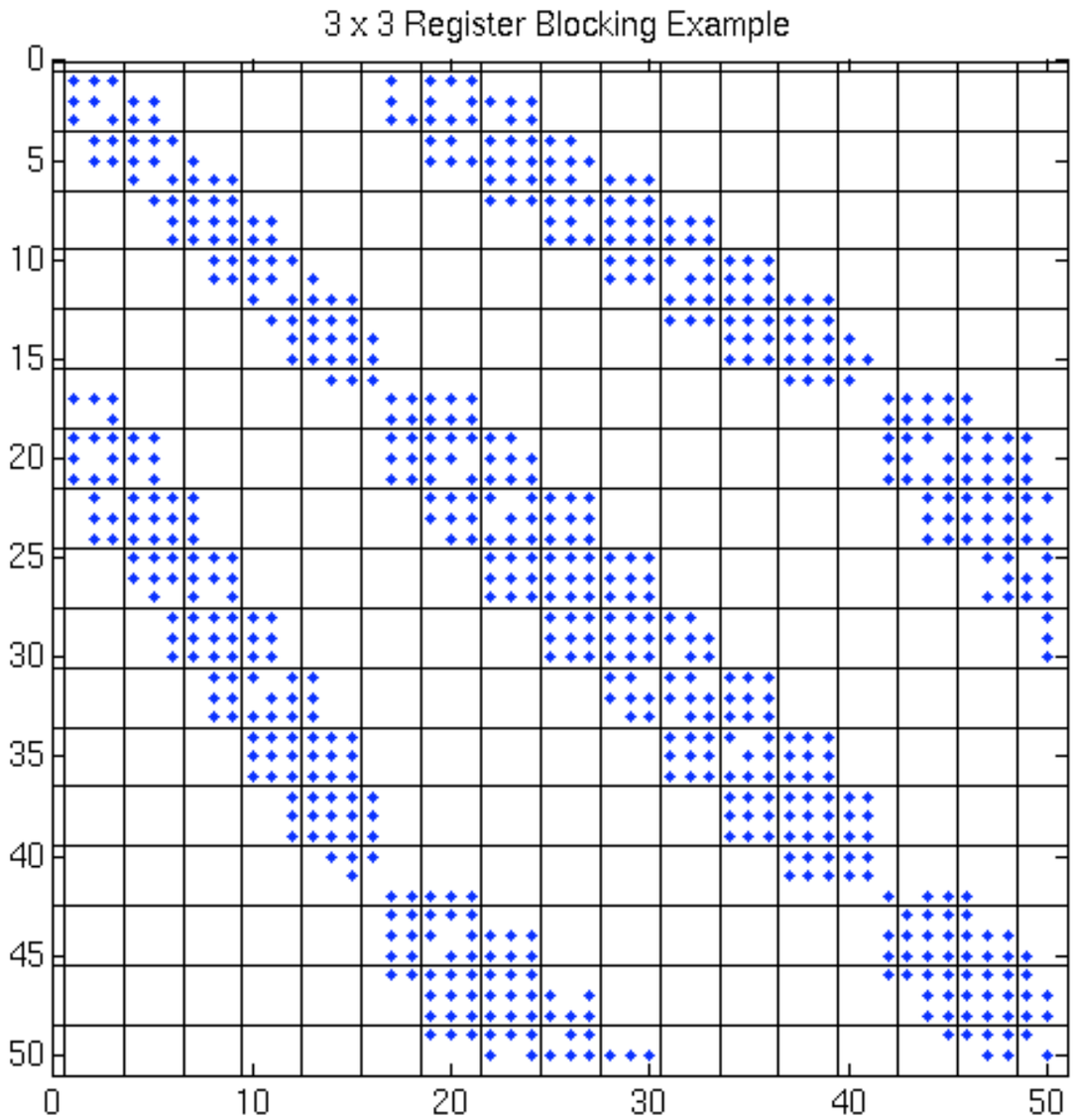
If you think the previous result was a fluke, observe that as we vary machines, the patterns can vary widely across machines.

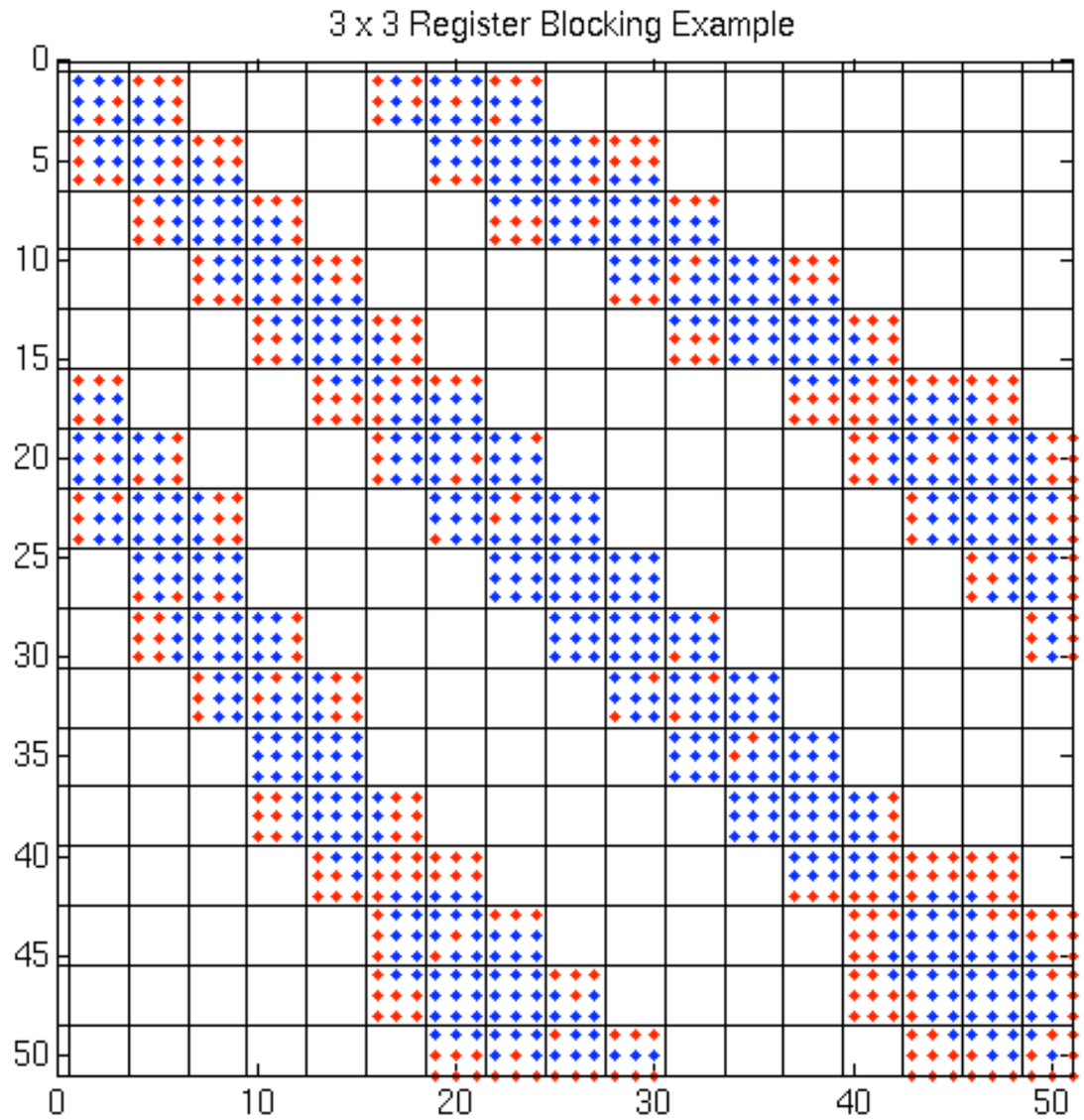


Indeed, sometimes blocking actually _hurts_ performance.

Why does this behavior occur? The answer is not easy and is machine-dependent. The goal of our research is to deliver robust high-performance regardless of the characteristics of the underlying architecture.

50% more flops, 2/3 time (1.5x)



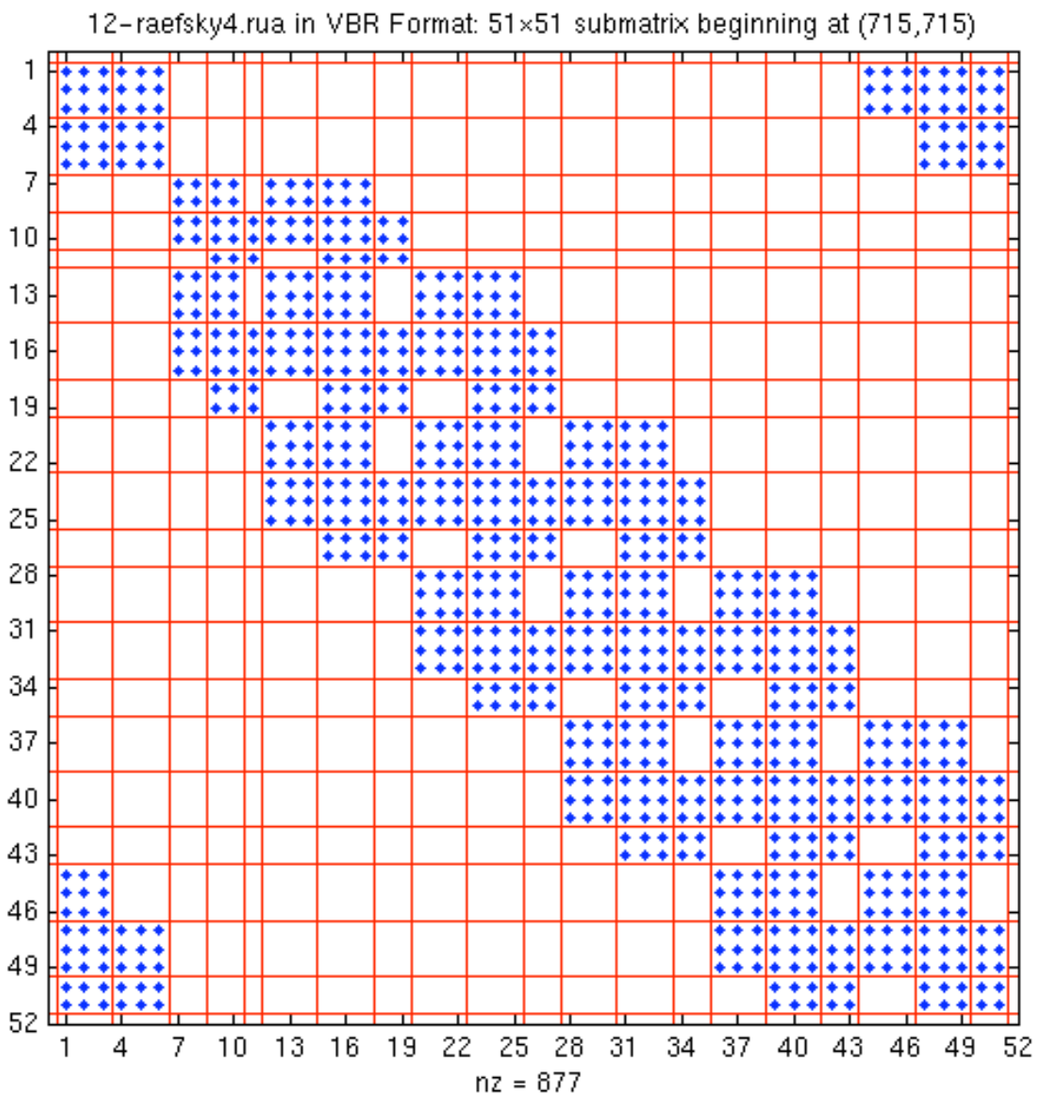


(?) More flops ⇒ Less time (!)

To make things more complicated, consider a matrix with some dense substructure that is not completely uniform. The matrix on the left (again, a piece of a much larger matrix) consists of a mix of block sizes.

However, we did an experiment on an old Pentium III machine in which we took a sparse matrix (left, non-zeros in blue) and forced it to be stored in a 3x3 format. To do so, we had to fill in explicit zeros (right, extra zeros as red dots). This “fill” means we have to perform extra calculations (flops) on the explicitly stored zeroes. However, somewhat to our surprise, in the experiment we still got an SpMV implementation that ran in _less time_.

Splitting for variable blocks



$$A = A_1 + A_2 + \cdots A_s$$

Split into structurally disjoint, separately tuned terms.

Complex tuning problem:

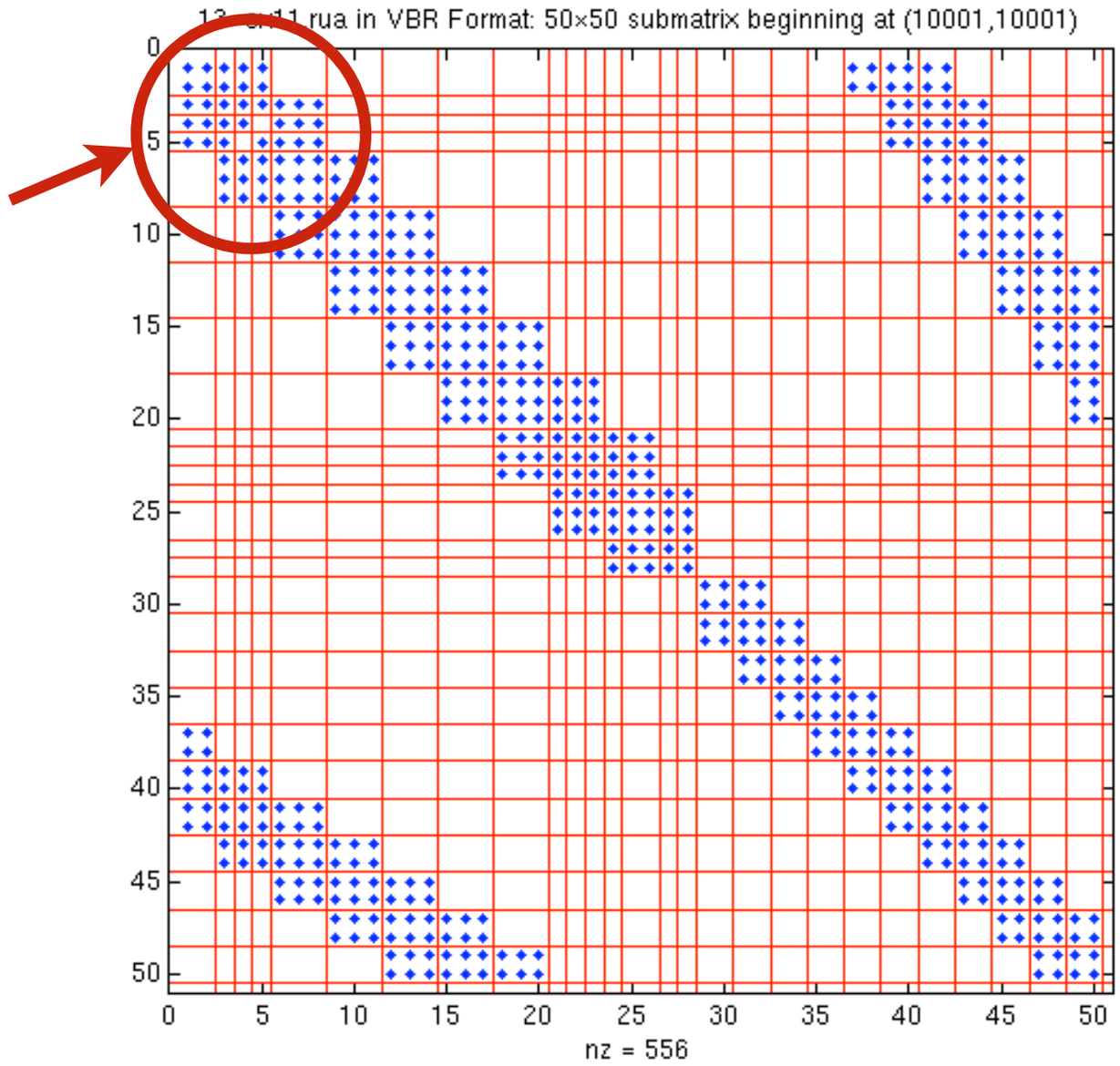
no. of terms, **extraction**, **fill**, plus individual term tuning

2.1x over CSR

1.8x over BCSR

Of course, rather than fill in zeros, we might want to try to exploit exactly the structure we have. One technique is to take the input matrix and split it into structurally disjoint parts, where each part can be tuned separately.

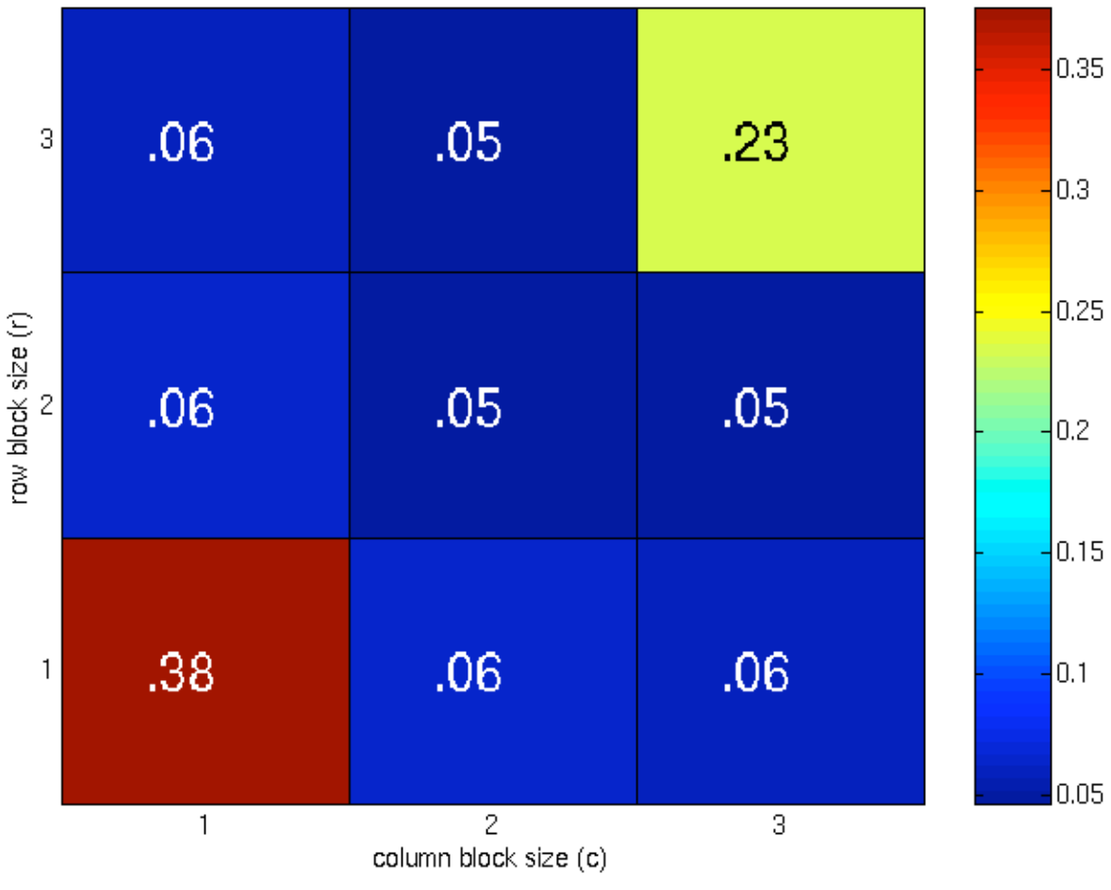
This technique can lead to a big win, but leads to a complex tuning problem. How many times should we split? Should we allow fill? How do we tune each part efficiently _at run-time_?



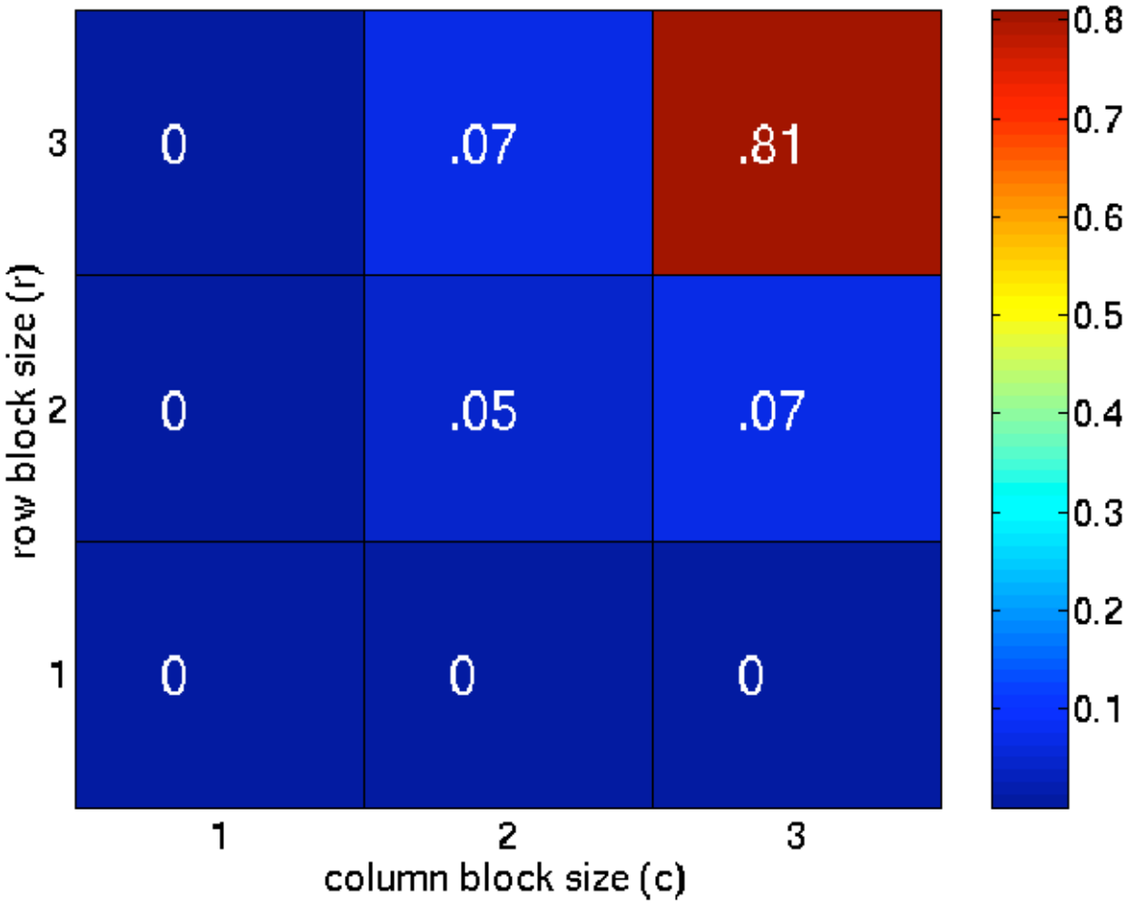
May "relax" partitions

We can easily partition the matrix into groups of consecutive rows and columns with the same pattern. It often makes sense to group by "near similarity," to improve the size of the partitions (i.e., get bigger blocks).

NZ distribution, θ=1.0

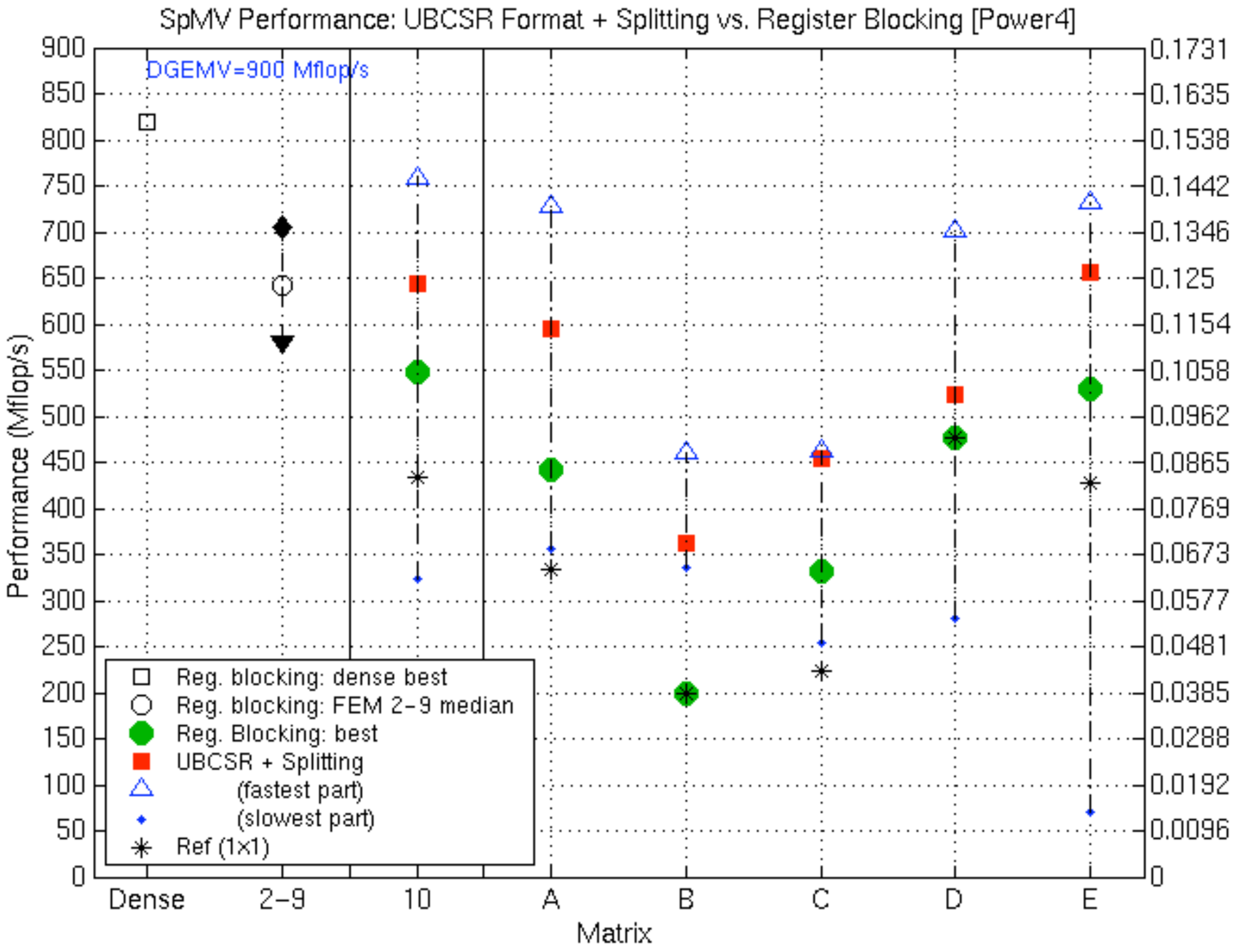


θ=0.7 ⇒ 1% more flops

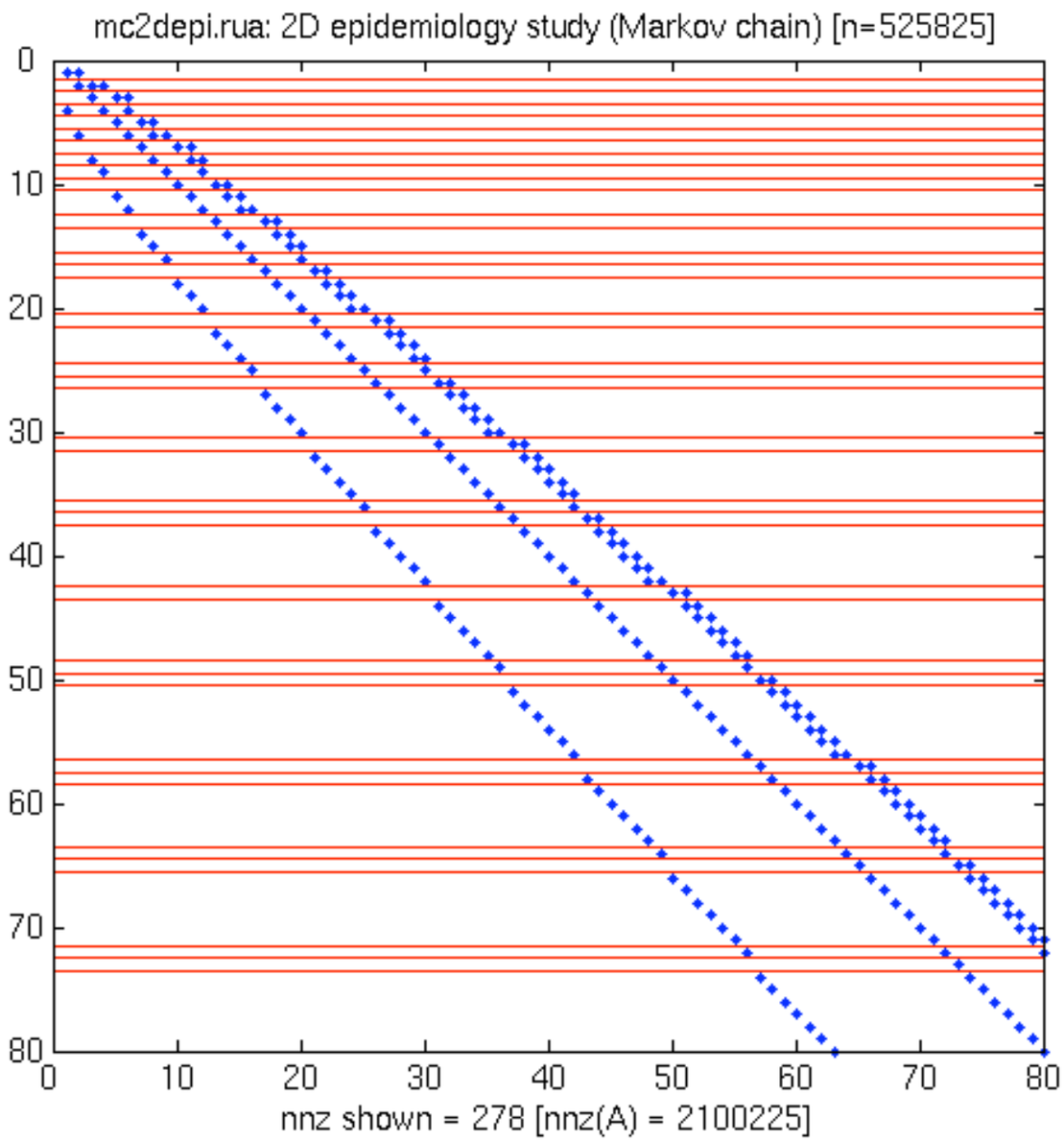


VBR(θ), θ = grouping similarity

For example, in the matrix shown before, partitioning by exactly matching rows/columns leads to a structure in which only 23% of all non-zeros appear in 3x3 blocks. By tolerating 1% fill (i.e., 1% more flops for SpMV), now 81% of all non-zeros appear in 3x3 blocks.



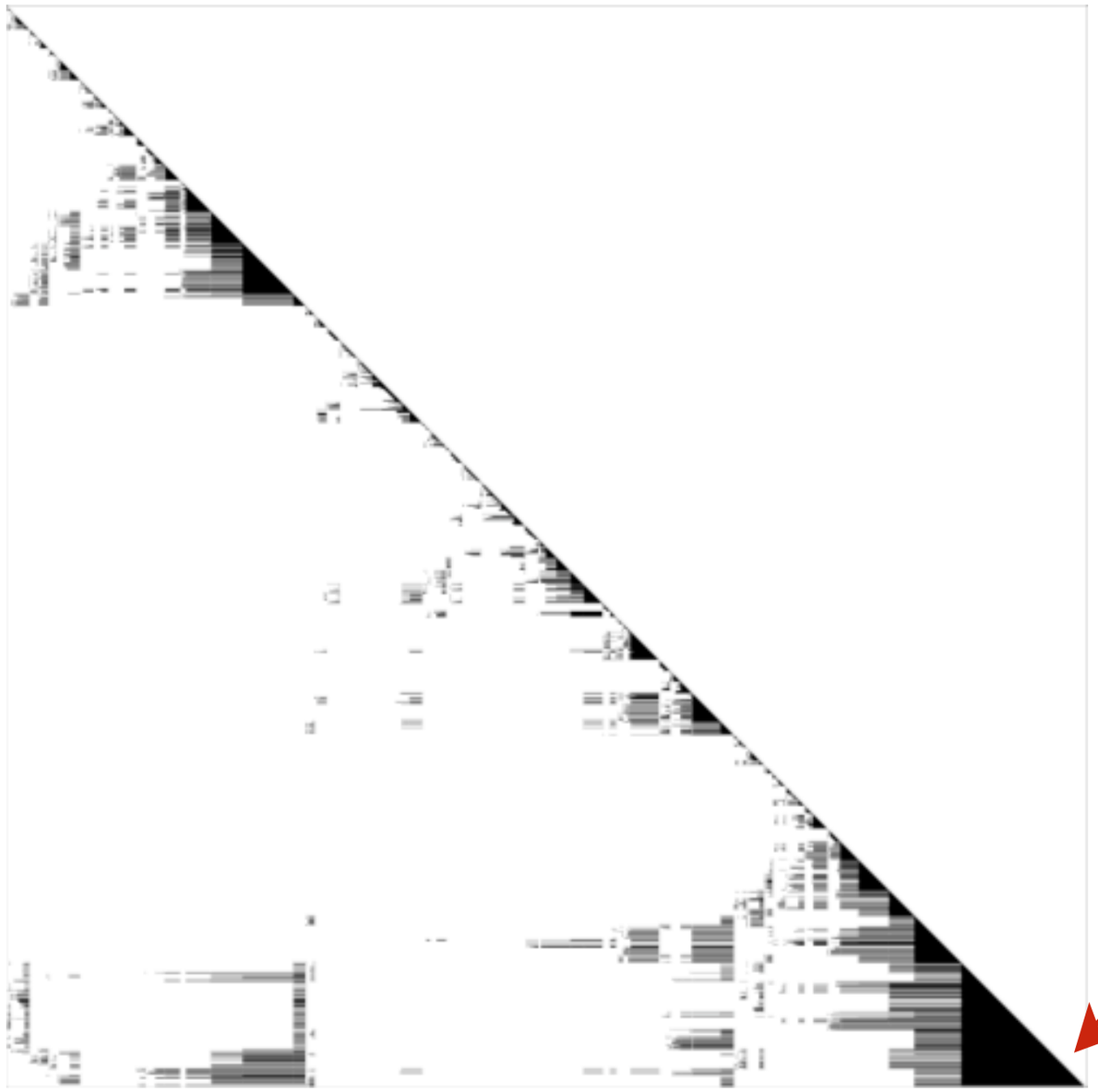
Splitting (shown in red) often beats “regular” blocking (with fill) significantly.



Up to 2x

Row-segmented diagonal

Beyond blocks, there are other canonical structures. For example, this matrix consists of sequences of rows containing only diagonal fragments.



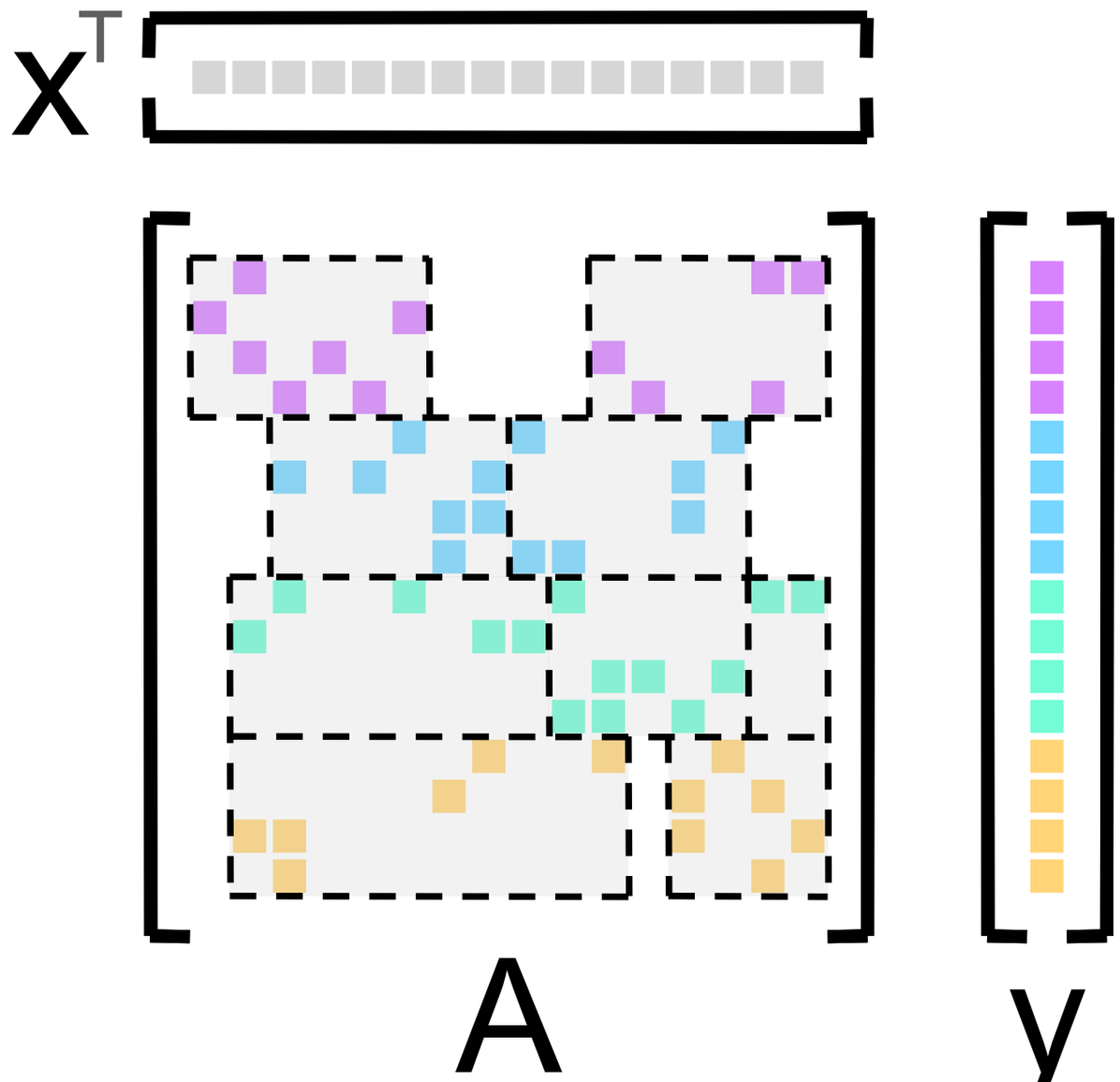
Dense triangular substructure

Triangular factor arising in sparse LU.

Triangular matrices that arise from sparse LU factorization (Gaussian elimination) often have dense triangular substructure. In this example, 90% of all non-zeros live in the trailing triangle in the lower-right corner.

$$y \leftarrow y + A \cdot x$$

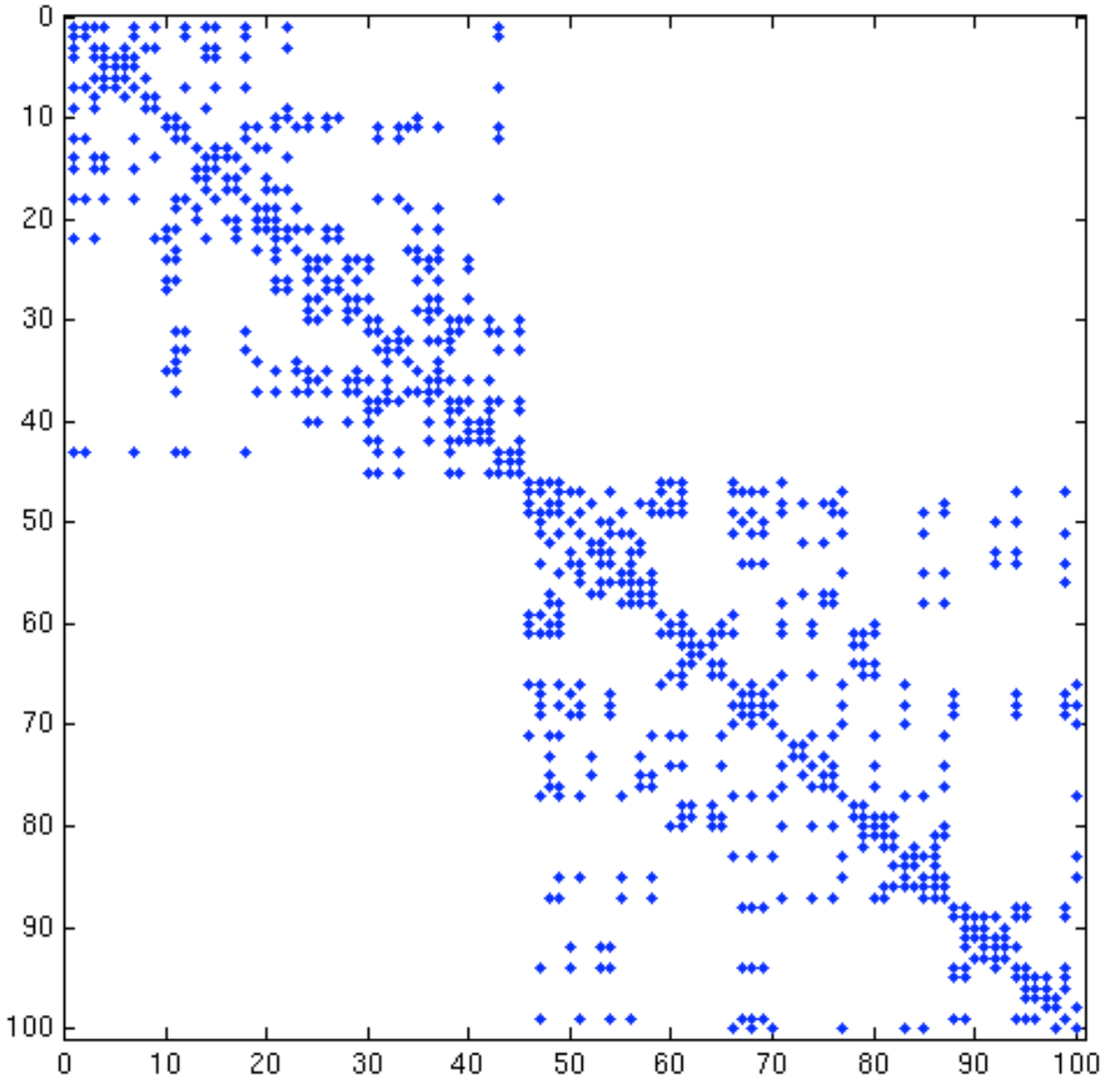
Up to 3x



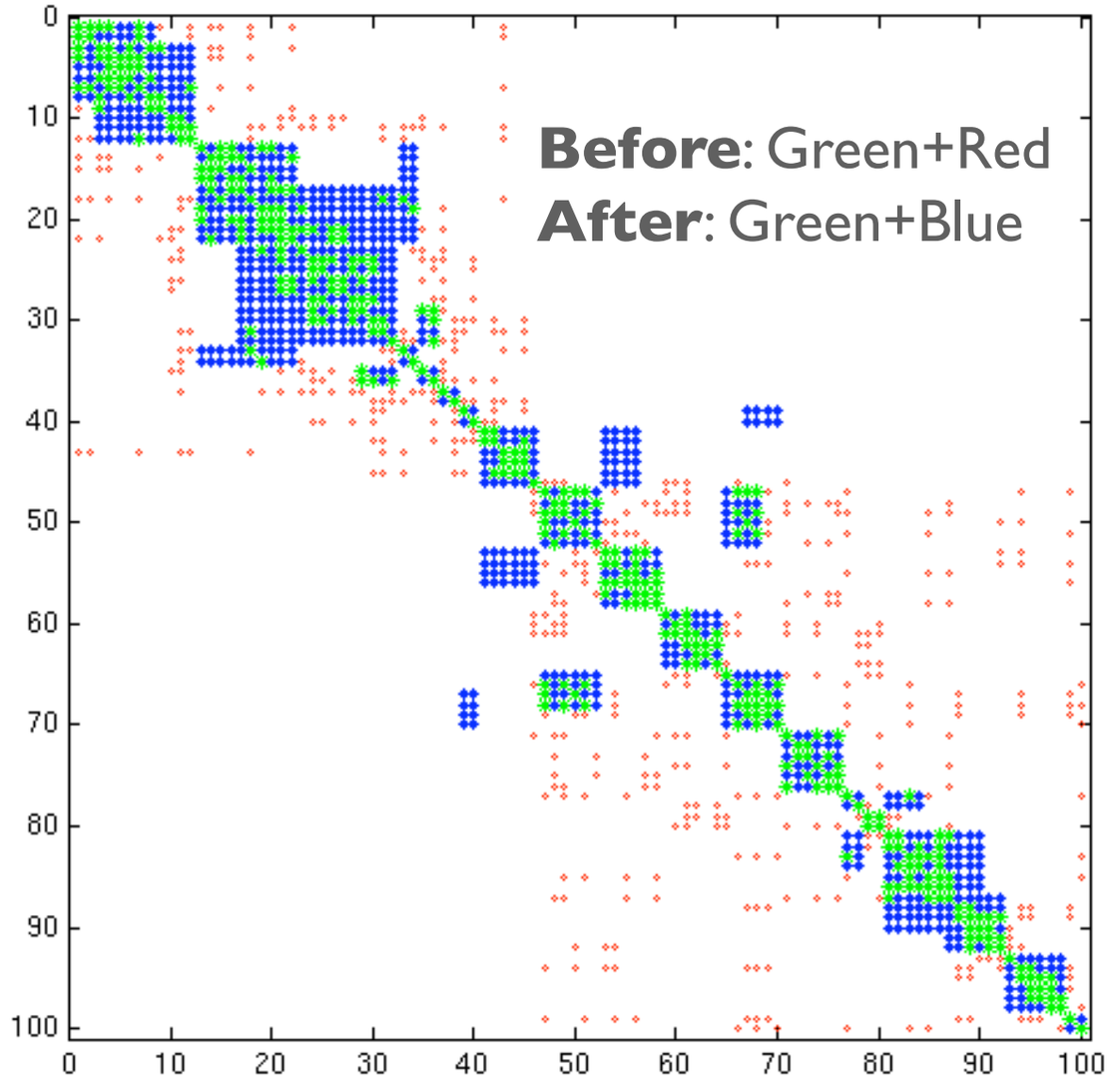
Cache & TLB blocking

For the SpMV operation, y ← y + A*x, all of the potential cache reuse occurs only in access to the vectors x, y. Breaking up the matrix A into a sequence of submatrices is a good technique for improving the locality in x, y accesses.

Source: Accelerator cavity



RCM+TSP-based reordering



1.4 - 1.7x

TSP-based reordering (Pinar)

One of the most interesting kinds of optimizations is actively reordering the rows and columns to discover structure. For example, the matrix on the left, which comes from an accelerator modeling application, can be reordered into the matrix on the right.

(The particular reordering can be found by formulating the reordering problem as a traveling salesman problem, and then applying some TSP heuristics.)

Tuning for workloads

- BiCG, with equal mix of Ax, A^Ty
  - 3x1: **1.05 Gflop/s**, 343 Mflop/s ⇒ 526 Mflop/s
  - 3x3: 806 Mflop/s, 826 Mflop/s ⇒ **816 Mflop/s**
- Fused (Ax, A^Ty) kernel
  - 3x1: 757 Mflop/s
  - 3x3: **1.4 Gflop/s**

Large data structure tuning space

- Optimizations for SpMV
  - Register blocking (RB): up to 4x over CSR
  - Variable block splitting: 2.1x over CSR, 1.8x over RB
  - Diagonals: 2x over CSR
  - Reordering to create dense structure + splitting: 2x over CSR
  - *Symmetry: 2.8x over CSR, 2.6x over RB*
  - Cache blocking: 3x over CSR
  - *Multiple vectors (SpMM): 7x over CSR*
  - And combinations…
- Sparse triangular solve
  - Hybrid sparse/dense data structure: 1.8x over CSR
- Higher-level kernels
  - *AA^T*x, A^TA*x: 4x over CSR, 1.8x over RB*
  - *A^2*x: 2x over CSR, 1.5x over RB*

Related work

- Lots! A sampling follows…

- Bounds modeling: Gropp (1999); V (2002)
- Blocking: Buttari & Eijkout (2005)
- Splitting: Toledo (1997); Geus (1999)
- TSP-based reordering: Pinar (1999; 2006)
- Compression: Willcock (2007)

Oski the Bear (Cal mascot)

Optimized Sparse Kernel Interface (OSKI)

II. Selecting data structures

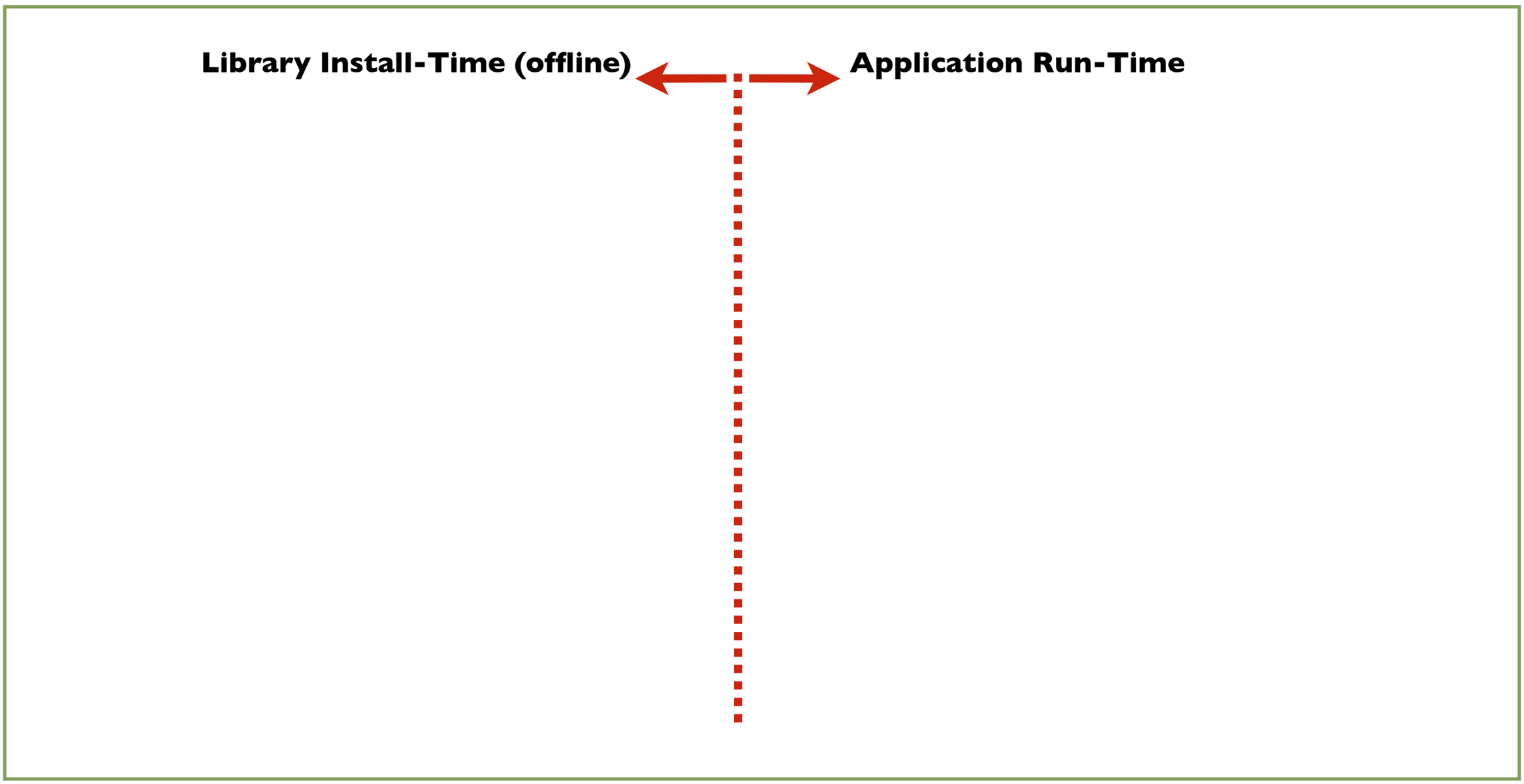
We implemented many techniques for these methods in a library called the Optimized Sparse Kernel Interface (OSKI).

Oski is also the name of the U.C. Berkeley football team mascot, where I did my PhD work leading to the development of the OSKI library. So the “real” Oski is shown above. (Go Bears!)

Optimized Sparse Kernel Interface (OSKI)

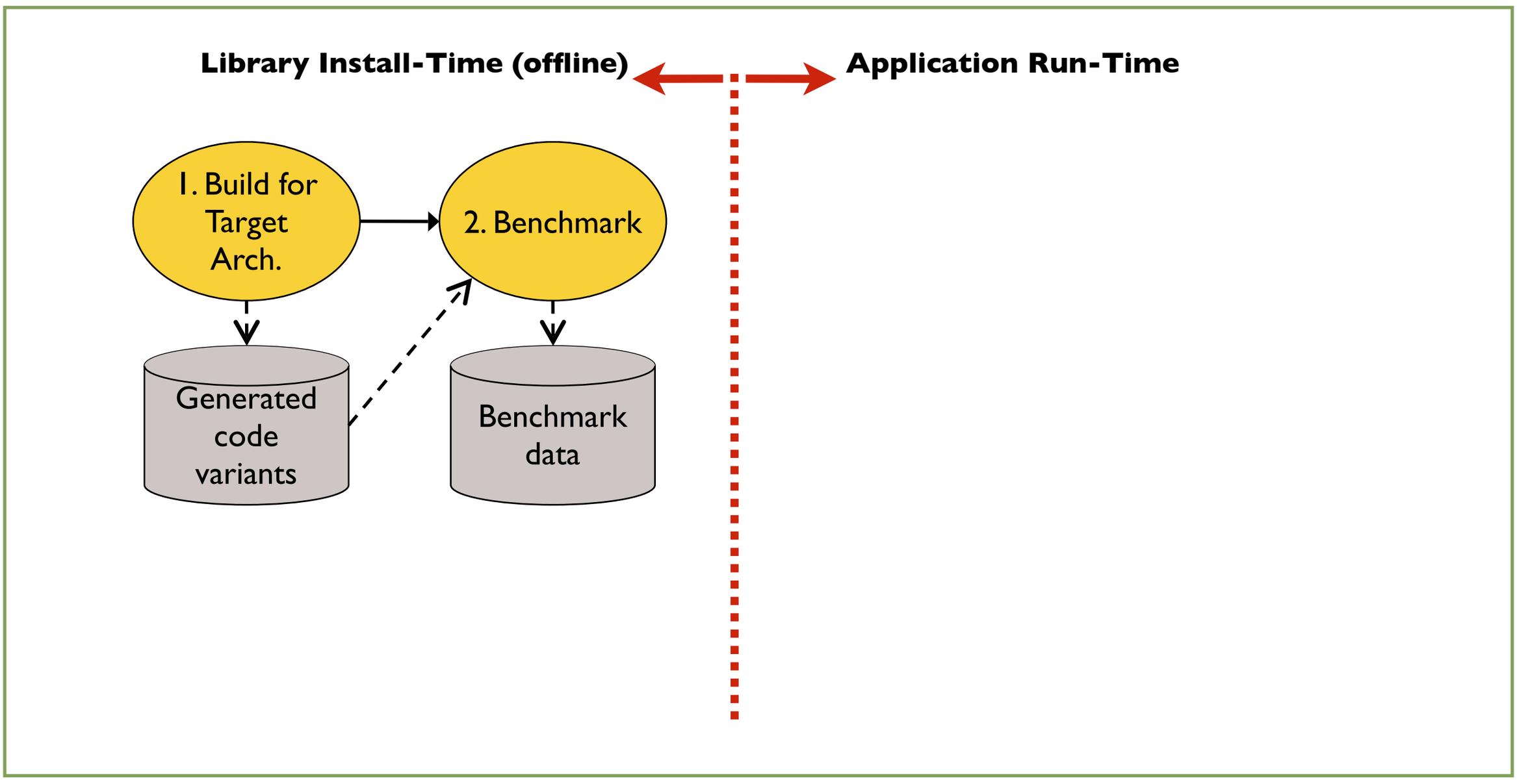
- Autotuned sparse BLAS-like library (C / F77)
  - Kernels: SpMV, tri solve, Ax & A^Ty, A^TAx, A^kx
  - Multivector kernels
  - Hides tuning complexity
- Speed
  - SpMV: ≤10% peak *vs.* up to **31%** with OSKI
  - SpTS: **1.8x**; A^TAx: **4x**
- Prototype integration with PETSc, Trilinos

How OSKI tunes

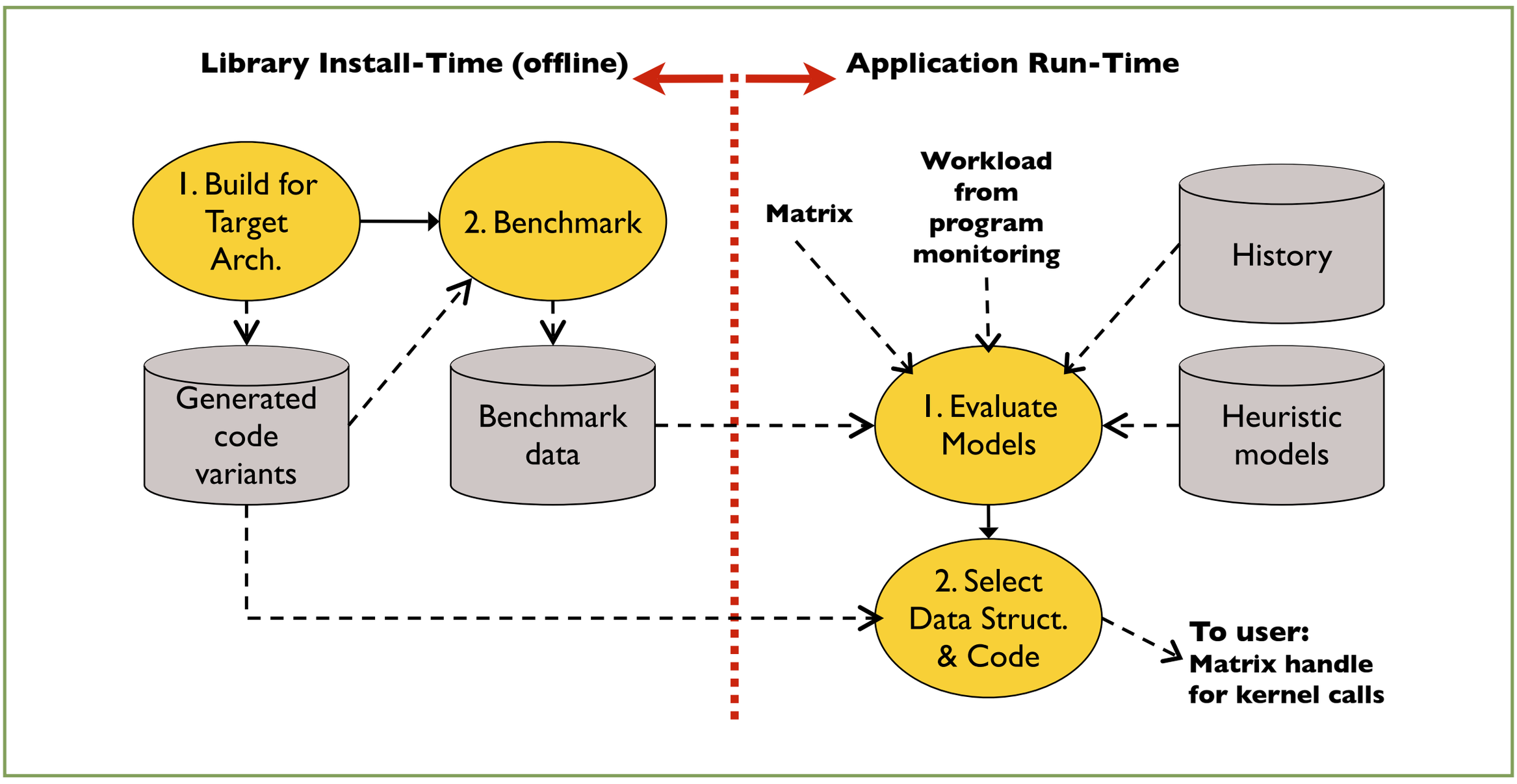


OSKI tunes in two stages: one occurring “off-line” when you compile the library, and the other at run-time when the sparse matrix is known.

How OSKI tunes



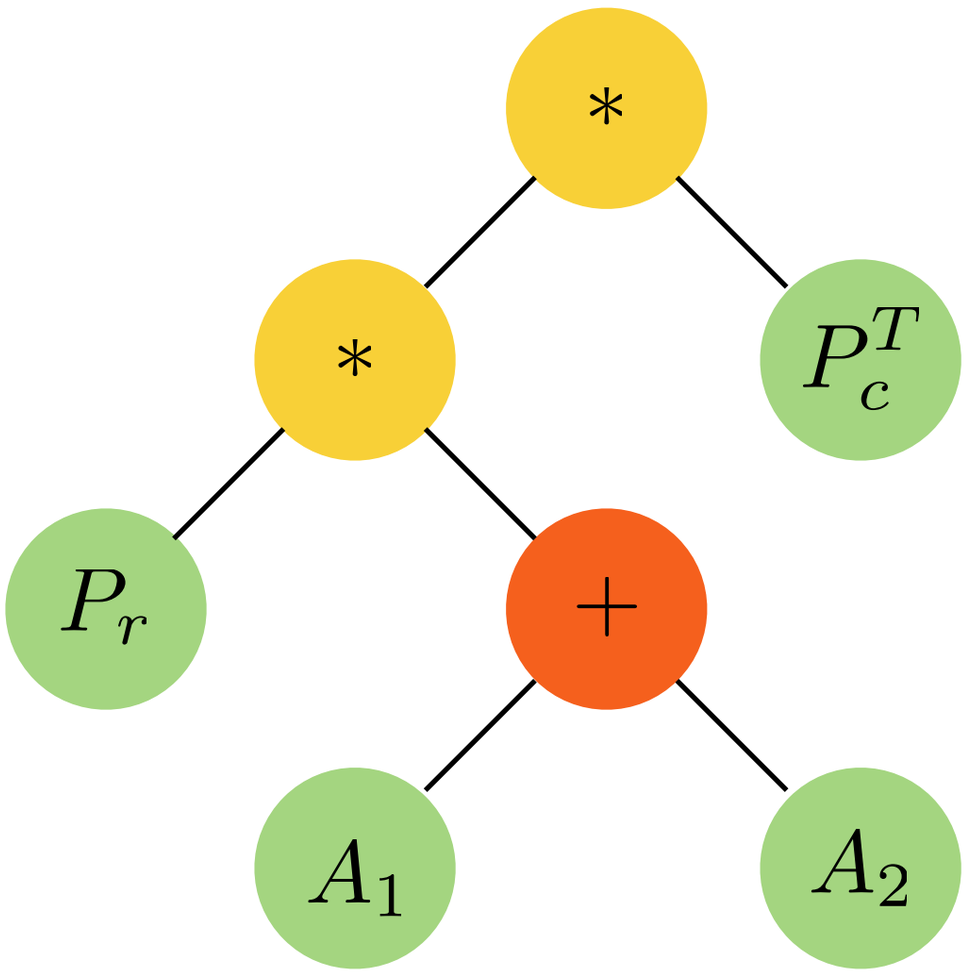
How OSKI tunes



At run-time, we take the user's matrix and any workload information, quickly examine it, and run various kinds of cheap run-time models to make decisions about how to tune. We return an opaque handle to the user, which the user uses to invoke various kinds of sparse kernels.

Expression trees

$$A = P_r \cdot (A_1 + A_2) \cdot P_c^T$$

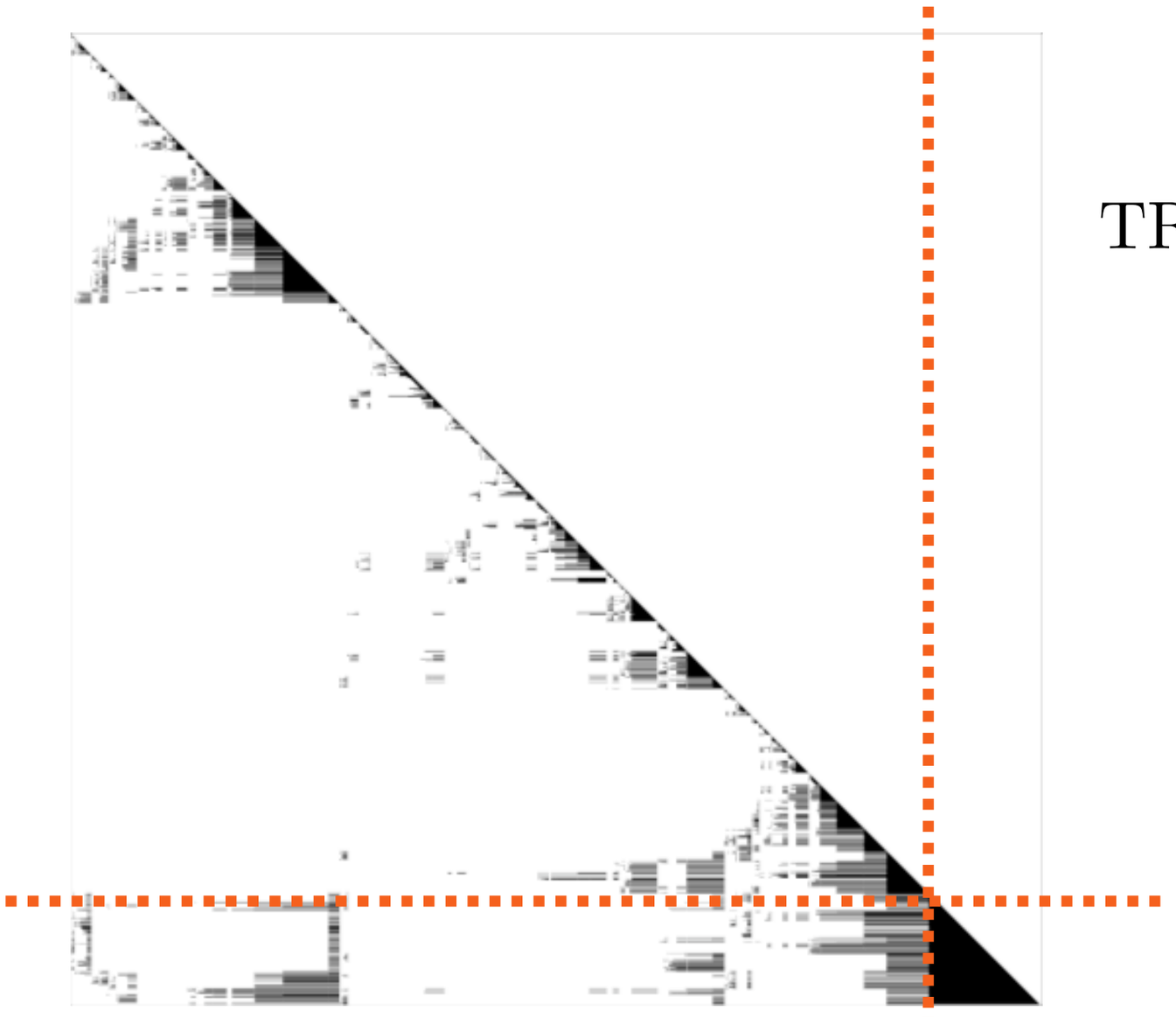


Nodes

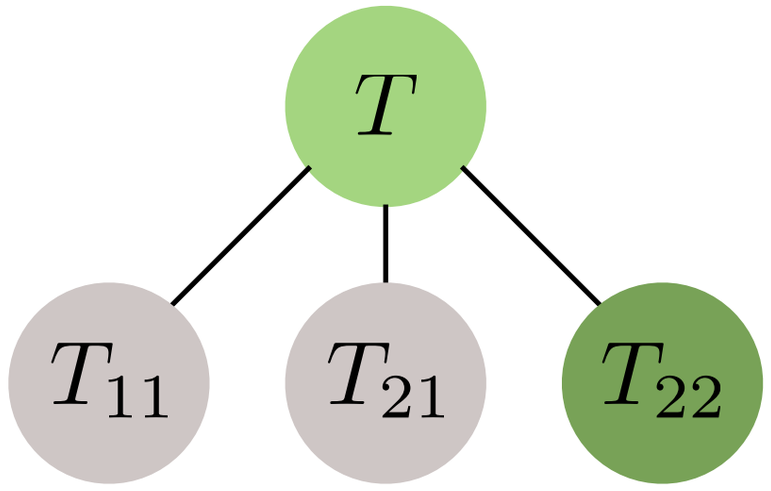
⇒ **Concrete matrix data**,

e.g., CSR, BCSR, PERM, ...

Internally, the data structure is represented by an expression tree. For example, here is the tree corresponding to a matrix that has been split and permuted. The nodes of the tree represent concrete matrix data structures.



$$\text{TRIPART}(T) = \begin{pmatrix} T_{11} & \\ T_{21} & T_{22} \end{pmatrix}$$



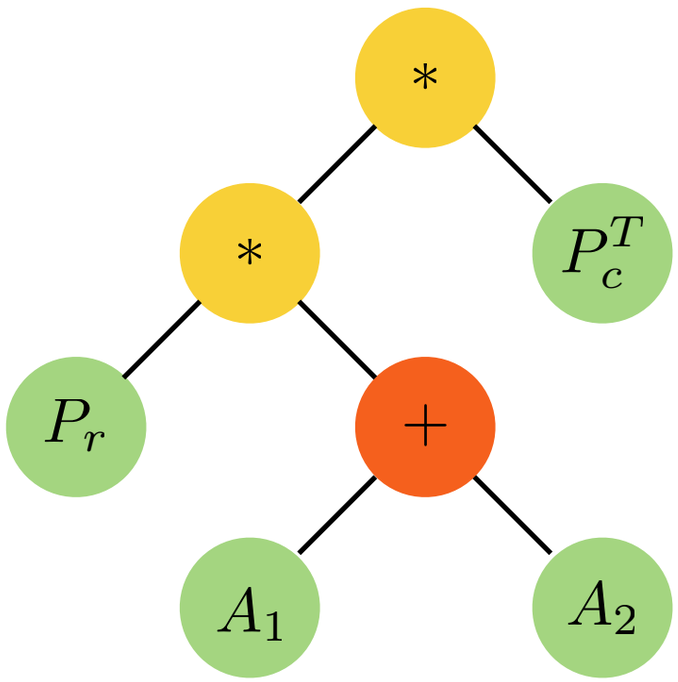
Concrete structures at leaves

For example, there is a type of node called TRIPART, which stores 3 triangular components of a 2-way block partitioning of the matrix. For this triangular matrix, the T22 component might be stored as a dense triangular matrix, and the other components as sparse matrices tuned for different block sizes.

OSKI-Lua, based on Lua (lua.org)

- Interpreted interface for reading or applying data structure transformations

$$A = P_r \cdot (A_1 + A_2) \cdot P_c^T$$




```
oski_ApplyMatTransforms (A_tunable,
                         string_buffer);
oski_MatMult (A_tunable, …);
```

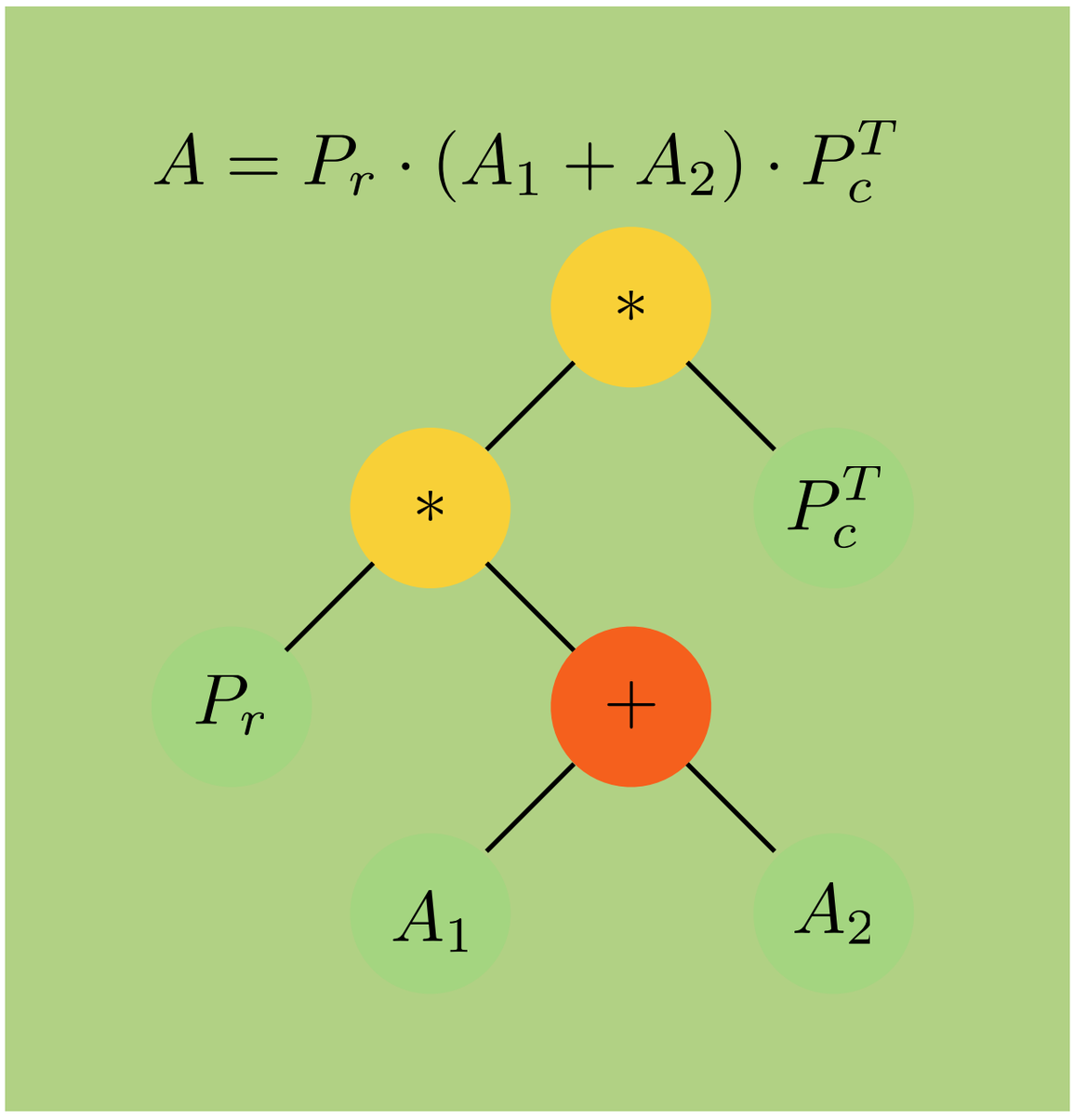
```
A_fast, Pr, Pc =
  reorder_TSP (InputMat)

A1, A2 =
  A_fast.extract_blocks(2,2)

return Pr * (A1 + A2) * Pc
```


Here's where Lua comes into play: we expose these expression tree data structures directly to the user using Lua. If OSKI decides to tune, the user can call a routine to ask for what transformations were applied. OSKI will return a string, which is a Lua program corresponding to the expression tree.

Similarly, the user can construct his/her own transformation as a Lua program, and ask OSKI to apply it to a given input matrix. This facility is particularly handy if the user knows how to tune the matrix or wants to experiment for any reason. For example, not all the techniques I showed you are fully automated, so you might have to ask for a particular transformation.

Status and future work

- OSKI-Lua infrastructure being rewritten
- Multithreaded OSKI "in the works"
  - PThreads-based, with thread & data affinity mapping
  - OpenMP
- Distributed OSKI



As you might imagine, all these transformations target single-core performance, whereas we are obviously much more interested in parallel performance. The need for data structure transformations is similiar, and we expect Lua to continue to play a key role in how we represent and communicate the tuning we do.