Bremen

CG
VR

Massively Parallel Algorithms
Dense Matrix Algorithms

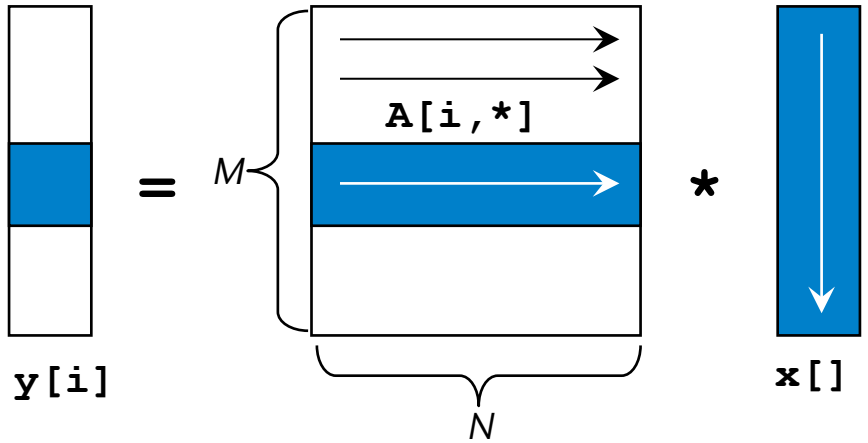
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

Warming Up: Matrix-Vector Product

- Given matrix *A*, and vector **x**, compute

$$\mathbf{y} = A\mathbf{x}$$

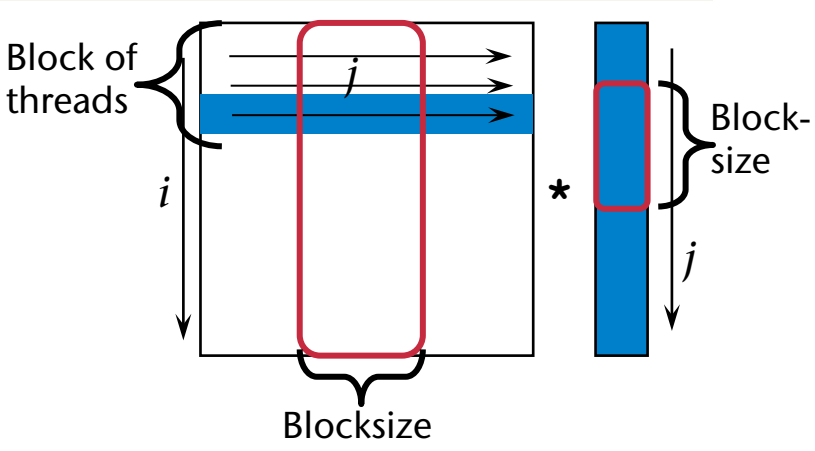
- One of the most important operations in linear algebra algorithms
  - Called SGEMV in BLAS (Basic Linear Algebra Subroutines)
- First approach: *one thread per row*



- Observation: all threads use the same data from **x** → shared memory

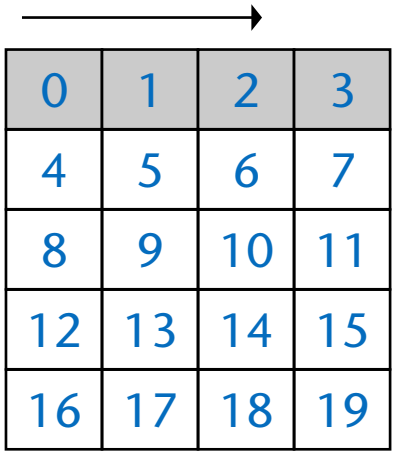
First Attempt

```
multMatrixVector( const float * A, const float * x,
                  const int n_columns,   float * y )
{
   __shared__ x_cache[ THREADS_PER_BLOCK ];
   yi = 0.0;                                  // output of each thread
   int i = threadIdx.x + blockIdx.x * blockDim.x; // row index
   for ( int j = 0; j < n_columns; j += THREADS_PER_BLOCK )
   {
      // new segment of columns → fill cache
      x_cache[threadIdx.x] = x[ j + threadIdx.x ];
      // now process this segment of columns
      for ( int k = 0; k < THREADS_PER_BLOCK; k ++ ) {
         Aij = A[ i*n_columns + (j+k) ];
         yi += Aij*x_cache[j+k];
      }
   }
   y[i] = yi;
}
```

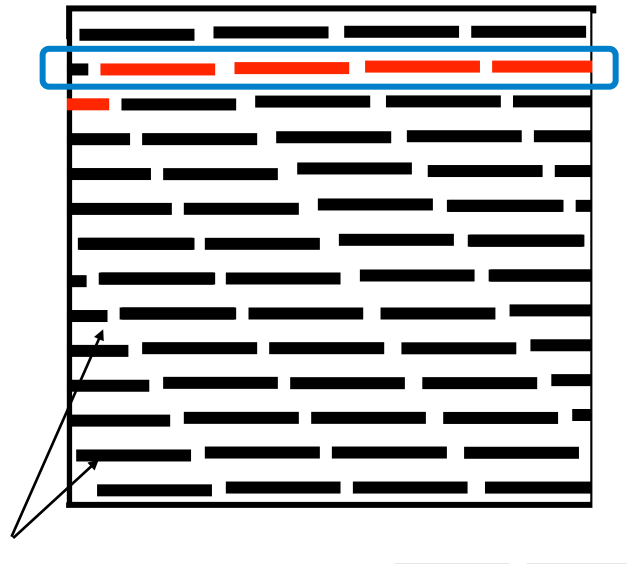


- For sake of clarity, we assume
  M, N = multiple of block-size

- The "natural" (C) way to store matrices is called row major order
  - A_{ij} is stored at memory address `A + i*n + j`
- For a conventional (sequential) matrix-vector-multiplication algorithm, this is good:



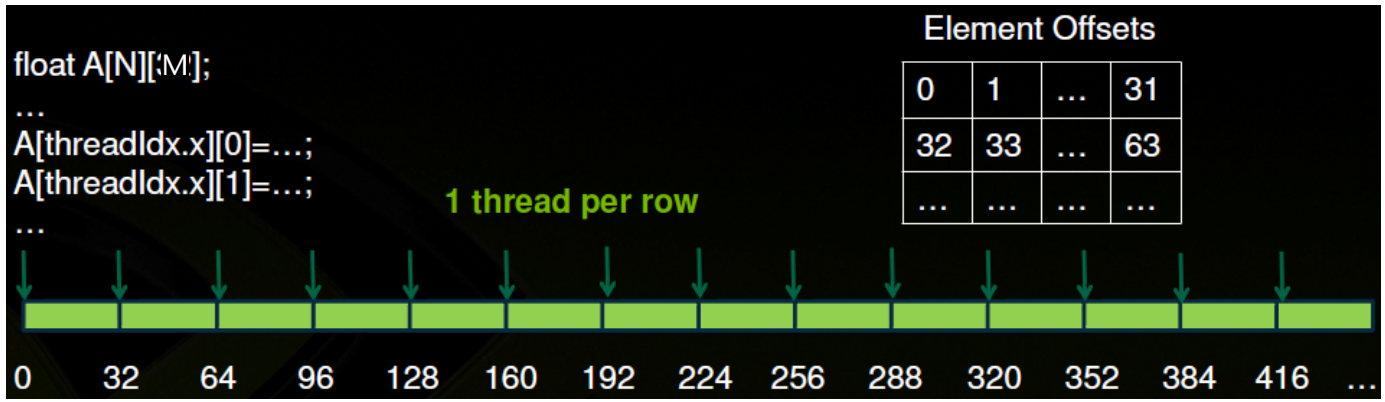

```
for ( int i = 0; i < M; i ++ ) {
   float yi = 0.0;
   for ( int j = 0; j < N; j ++ )
      yi += A[i][j] * x[j];
   y[i] = yi;
}
```



2D Array Access Patterns (row major vs column major)

- Consider the following piece in a kernel (e.g., matrix × vector):

```
for ( int j = 0; j < blockDim.x; j ++ ) {
   float Aij = A[treadIdx.x][j];
   ... do something with it ...
```



Memory layout of a matrix in C !

- Problem: uncoalesced access pattern
  - Elements read on 1st SIMT access: 0, 32, 64, ...
  - Elements read on 2nd SIMT access: 1, 33, 65, ...
  - Also, extra data will be transferred in order to fill the cache line size
- Generally, most natural access pattern for direct port of a C/C++ code!

Transposed 2D Array Access Pattern

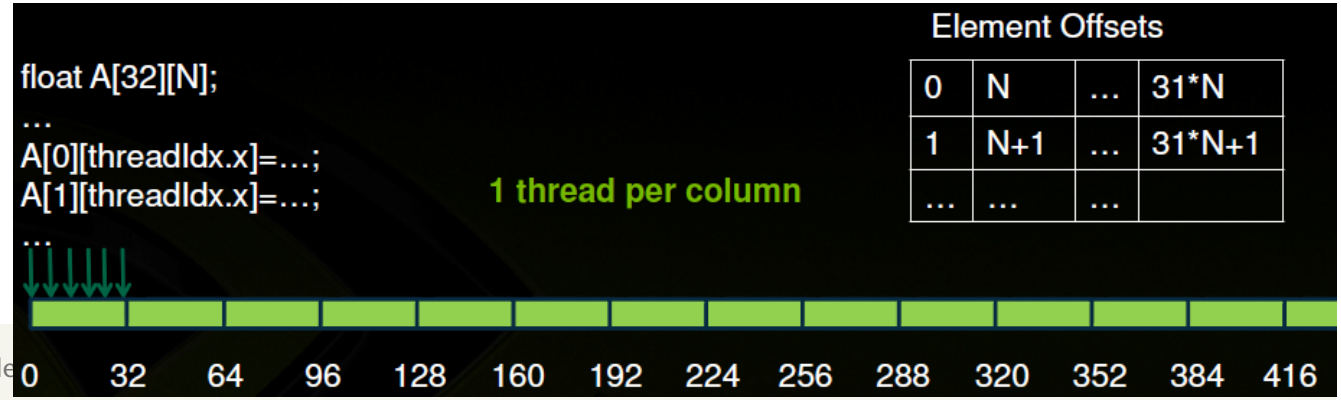
- Column major := store a *logical* row in a *physical* column
  - I.e., $A_{00} \rightarrow$ A[0][0] , $A_{01} \rightarrow$ A[1][0] , $A_{02} \rightarrow$ A[2][0] , ...
    $A_{10} \rightarrow$ A[0][1] , $A_{11} \rightarrow$ A[1][1] , $A_{12} \rightarrow$ A[2][1] , ...
    $A_{20} \rightarrow$ A[0][2] , ...
  - In general: A_{ij} is stored at **`A + i + j*n`**

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

- Transform the code to column major:

```
for ( int j = 0; j < blockDim.x; j ++ ){
   float Aij = A[j][treadIdx.x];
   ... do something with it ...
```

- Now, we have coalesced accesses:
  - Elements read on 1st SIMT access: 0, 1, 2, ..., 31
  - Elements read on 2nd SIMT access: 32, 33, ..., 63

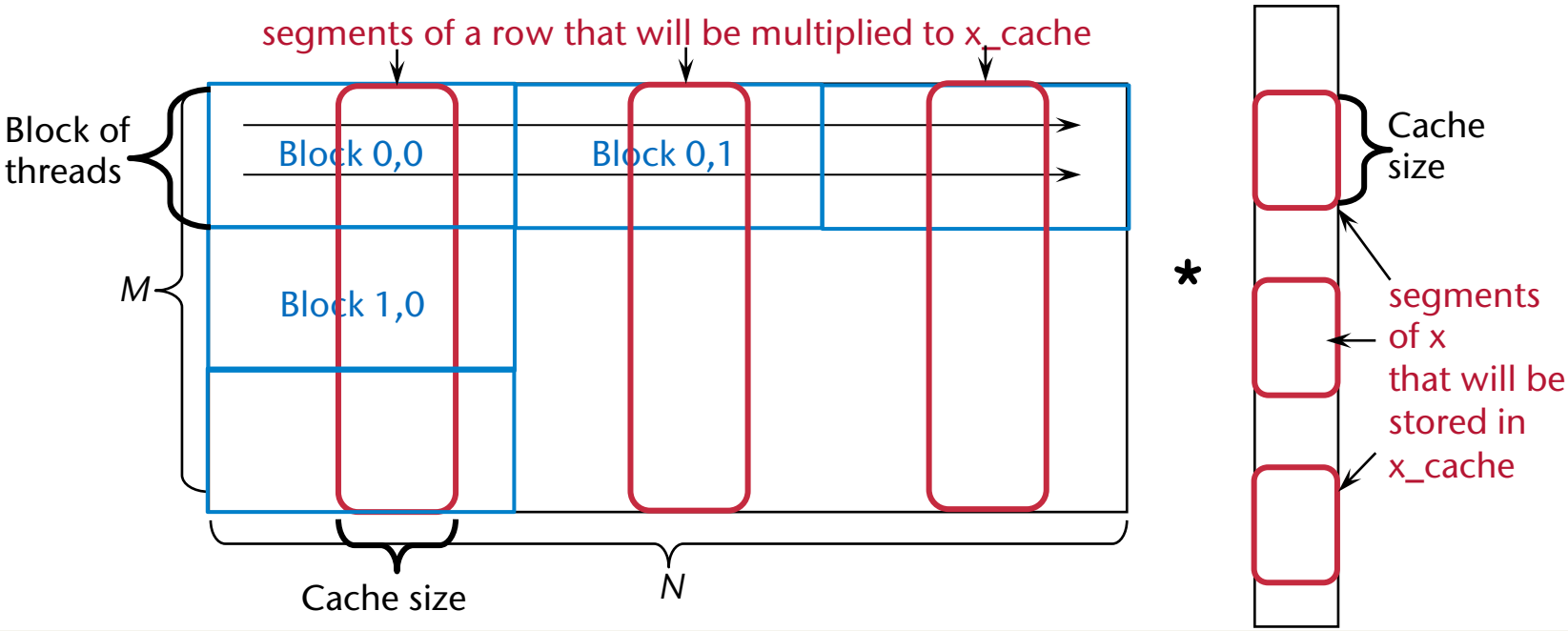


Modified Matrix*Vector Algorithm for Column-Major Matrix Storage

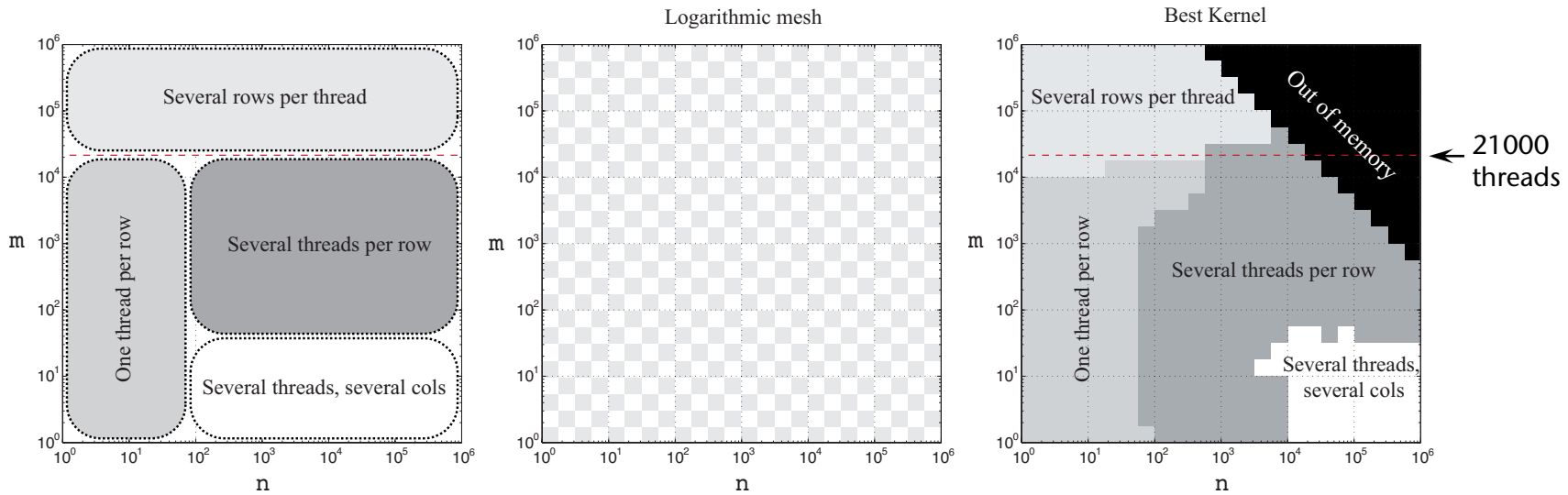
```
multMatrixVector( const float * A, const float * x,
                  const int n_columns ,   float * y )
{
   __shared__ x_cache[ THREADS_PER_BLOCK ];
   yi = 0.0;                                  // output of each thread
   int i = threadIdx.x + blockIdx.x * blockDim.x; // row index
   for ( int j = 0; j < n_columns; j += THREADS_PER_BLOCK )
   {
      // new segment of columns → fill cache
      x_cache[threadIdx.x] = x[ j + threadIdx.x ];
      // now process this segment of columns
      for ( int k = 0; k < THREADS_PER_BLOCK; k ++ ) {
         Aij = A[ i + (j+k)*n_columns ];
         yi += Aij * x_cache[j+k];
      }
   }
   y[i] = yi;
}
```

- Note: from now on, we will use row-major notation (for sake of clarity)!
  - But we will assume that an actual implementation uses column-major!
  - We expect you to transform everything to column-major
  - Start with small matrices that you can check "by hand"
  - Or implement your code first on the CPU and test it there

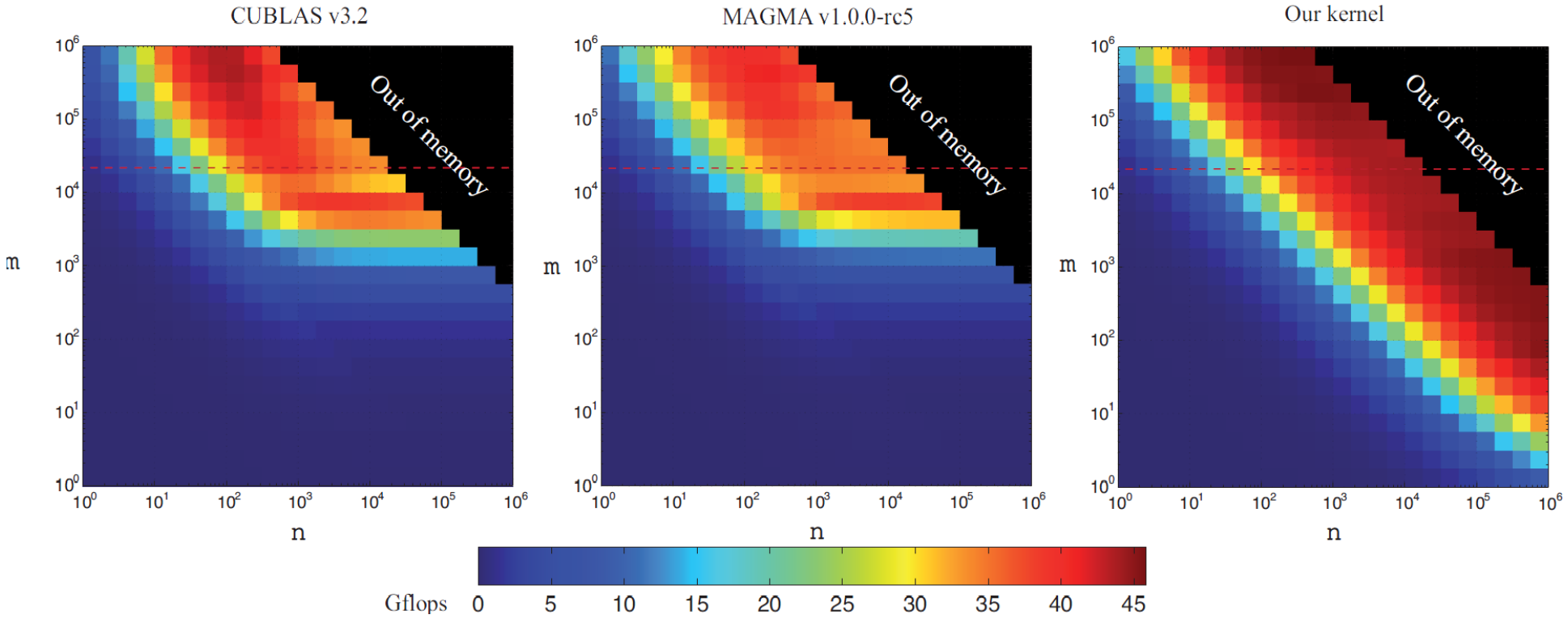
- Do we keep all hardware resources of the GPU busy?
- Assume Fermi [2011] hardware:
  - 14 SMs, each supports 1536 active threads
  - If $M < 21504 = 14 \times 1536 \rightarrow$ some SMs are idle!
- Idea for the case $M < 21504$ and N "not too small":
  - Use 2D partitioning of our problem/domain



- All possible variants:
  1. One thread per row
  2. Several threads per row (previous slide)
  3. Several rows per thread (one thread computes several y[i]'s at the same time)
  4. Several threads, several rows (version 2 & 3 combined)
- Which version is best in which case? (YMMV)



- Computational performance that can be achieved [2011]:



Performance of matrix-vector multiplication (SGEMV) over matrices of size *m*×*n*

["Fast High-performance Modeling Tools for Many-core Architectures ", Glimberg et al., 2011]

Complexities

- Sequential version: $O(n^2)$ (assuming $n=m$)
- Parallel version: $O(n)$ parallel time
  - Assuming $O(n)$ parallel threads
- Arithmetic intensity:
  - Assume following simplified version:

```
load vector x completely into fast memory
for i = 1 ... n:                    // assuming m = n
   load row i of A into fast memory
   for j = 1 ... n:
      yi += A[i][j] * x[j]
   store yi in y[i]
```

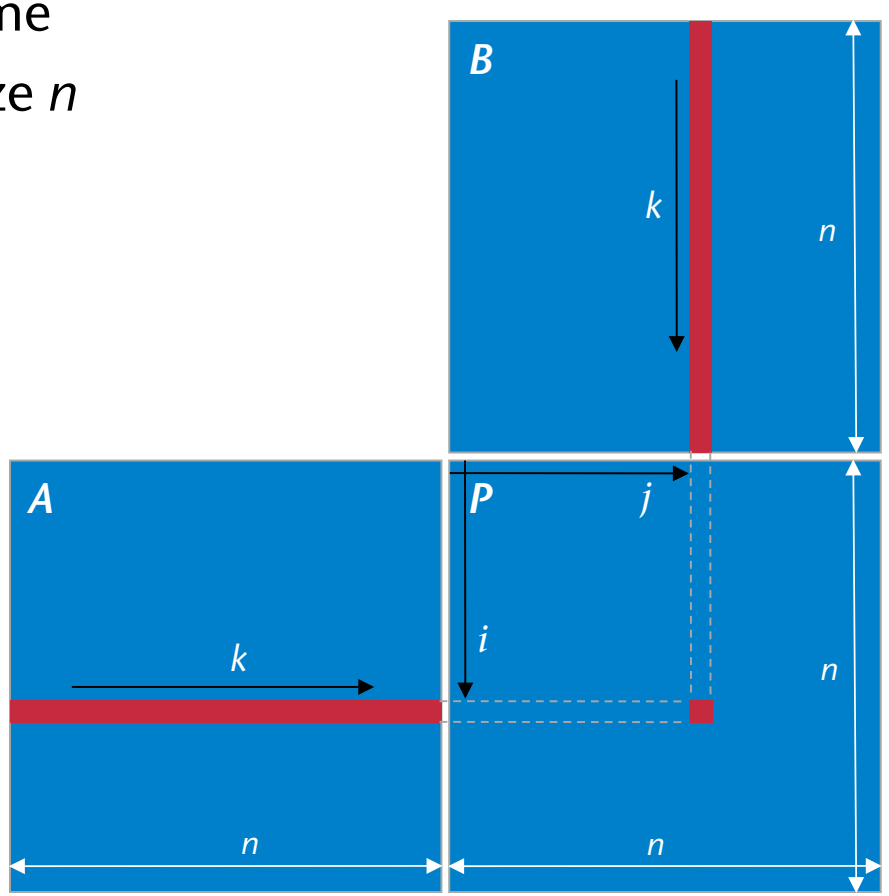
  - Number of slow memory references = $f = 2n + n^2$
  - Number of arithmetic operations = $o = 2n^2$
  - Arithmetic intensity $a = \frac{o}{f} \approx 2$ → memory limited

- Remark: actually, SGEMV in BLAS computes $\mathbf{y} = \alpha A\mathbf{x} + \beta\mathbf{y}$
  - Should be fairly straight-forward to modify our kernels

Matrix-Matrix Multiplication

- Called SGEMM in BLAS
- Given matrices A and B, compute $P = A \cdot B$
- For sake of simplicity, we'll assume
  A and B are square matrices of size n
- Sequential algorithm:

```
for i = 1 ... n:
  for j = 1 ... n:
    s = 0.0
    for k = 1 ... n:
      s += A[i][k] * B[k][j]
    P[i][j] = s
```



- Complexity: $O(n^3)$
- Arithmetic intensity:

$$a = \frac{2n^3}{2n^3 + n^2} \approx 1$$

```
for i = 1 ... n:
  for j = 1 ... n:
    s = 0
    for k = 1 ... n:
      s += A[i][k] * B[k][j]
    P[i][j] = s
```

  - Even worse than matrix-vector mult.
  - Upper bound (at least with iterative = non-recursive algorithms):

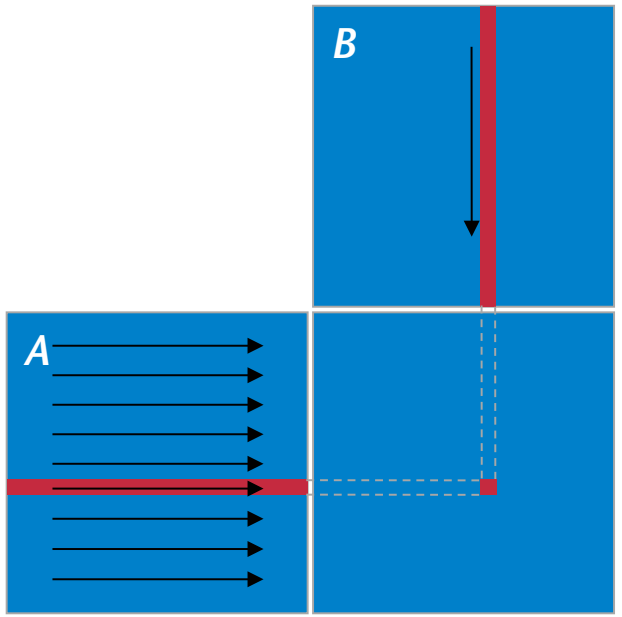
$$\hat{a} = \frac{2n^3}{3n^2} \in O(n)$$

- Problem: no data re-use!

Naïve Parallel Matrix-Multiplication

- Approach:
  - Use matrix-vector-multiplication idea
  - Run one thread per row of A:

```
for j = 1 ... n:
   read column j of B into fast memory (B_cache)
   foreach i = 1 ... n run one thread in parallel:
      s = 0.0
      for k = 1 ... n:
         s += A[i][k] * B_cache[k][j]
      P[i][j] = s
```



- Arithmetic intensity:

$$a = \frac{2n^3}{n^2 + n^3} \approx 2$$

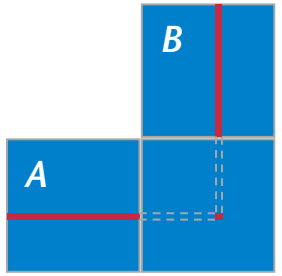
- Not much better ☹

Blocked (Tiled) Matrix-Multiplication

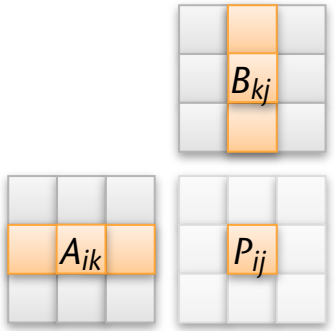
- Remember linear algebra class: the procedure

$$p_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

works also for sub-blocks of the matrices



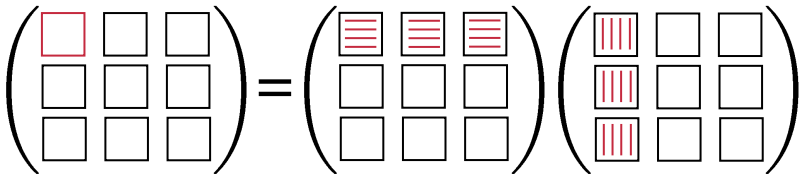
$$P_{ij} = \sum_{k=1}^{n/m} A_{ik} B_{kj}$$



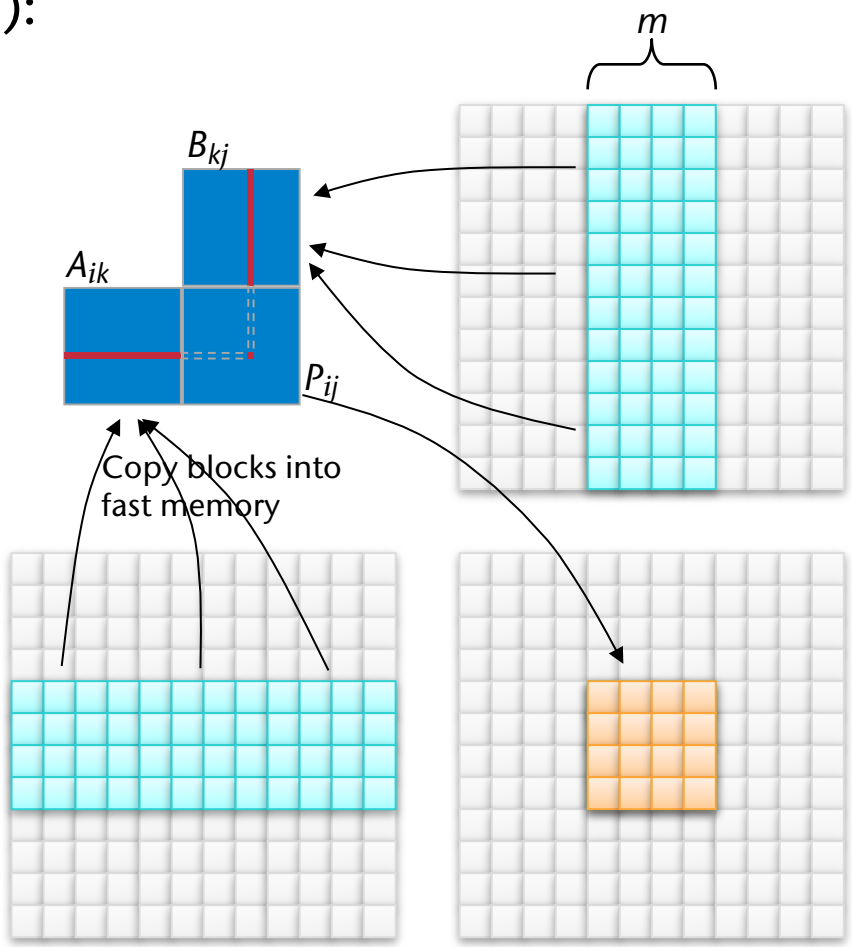
where $A_{ik}, B_{kj} \in \mathbb{R}^{m \times m}$

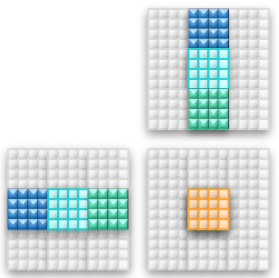
are *block matrices* of size m

- Assumption: n = multiple of m
  - In production code, you'd have to cope with any matrix size!
    - Lots of nitty-gritty details ...



- New approach (2D partitioning):
  - For each sub-matrix P_{ij}, run one block of m^2 threads
  - Each thread in the block computes one p_{ij}
  - The kernel runs in phases
- Each phase consists of:
  1. Load blocks A_{ik}, B_{kj} into shared memory
     - Each thread loads one a_{ij}, one b_{ij}
  2. Perform "row × column" over block
  3. Accumulate partial result





- Pseudo-Code:

```
let b = n/m          // = number of blocks in each dimension
foreach i = 1...b, j = 1...b  run one block in parallel:
  let p = 0.0        // = thread-local accumulator
  for k = 1 ... b:
    load sub-matrices A(i,k) and B(k,j) into shared memory
    → Asub , Bsub
    for l = 1...m:
      p += Asub[tid.x][l] * Bsub[l][tid.y]
  P[I,J] := p        // I,J = per-thread global indices into P
```

```
dim3 threadsPerBlock(m,m);
dim3 n_blocks( n/m, n/m );
multMatrices<<< n_blocks, threadsPerBlock >>>( A, B, P, n );
```

- Arithmetic intensity:
  - P consists of b^2 blocks
  - For each block P_{ij}, we load b blocks of A and b blocks of B
  - Overall, our algorithm loads $2b^3$ many blocks
  - One block load = m^2 float loads
  - $b = \frac{n}{m}$
  - Overall, our algorithm loads $2\left(\frac{n}{m}\right)^3 m^2 = 2\frac{n^3}{m}$ many floats
  - Therefore, $a = \dfrac{2n^3}{2\frac{n^3}{m}} = m$
- Consequence: make m large
- Limit: all three blocks P_{ij}, A_{ik}, B_{kj}, must fit in shared memory

- Calculating *m*:
  - Assume Kepler-GPU: ~ 2 TFlops/sec = $2 \cdot 10^{12}$ FLOPs/sec ,
    ~ 200 GB/sec = $200 \cdot 10^{9}$ B/sec
  - Choose *m* such that we achieve peak bandwidth & peak FLOPs/sec
  - $$m = a = \frac{\#\text{ FLops}}{\#\text{ Loads}} = \frac{\#\text{ Flops/sec}}{\#\text{ Loads/sec}} = \frac{2 \cdot 10^{12}\ \text{Flops/sec}}{\frac{200}{4} \cdot 10^{9}\ \text{B/sec}} = 40$$

    1 Load = 4 Bytes
  - Note: these are very crude estimations, but good for a starting point where to search for the sweet spot
- Consequence: size of shared memory should be at least
  $$3 \cdot 40^2 \cdot 4\ \text{Bytes} = 19.2\ \text{kB}$$
  - Otherwise, we would be bandwidth limited

Limitations / Optimality

- Tiling/blocking only works, if the arithm. operation is associative
- Arithmetic intensity, a, is bounded by size of shared memory, S:

$$a \approx m \leq \sqrt{\frac{S}{3}}$$

- Our algorithm performs $O\left(\frac{n^3}{\sqrt{S}}\right)$ many loads operations
- Note: in a sense, our blocked matrix multiplication algorithm is a way to schedule memory transfers and floating point operations
- Theorem (Hong & Kung, 1981; w/o proof):
  Any schedule of conventional matrix multiplication must transfer $O\left(\frac{n^3}{\sqrt{S}}\right)$ many floats between slow and fast memory.
- In this sense, blocked matrix multiplication is *optimal*

- Previous optimization is called blocking/tiling (copy optimization)
- How should matrices *A* and *B* be stored?
  - Remember: at the beginning of each phase: each thread loads one a_{ij} & one b_{ij}
- Store matrices in blocked form, in order to achieve coalesced memory access:

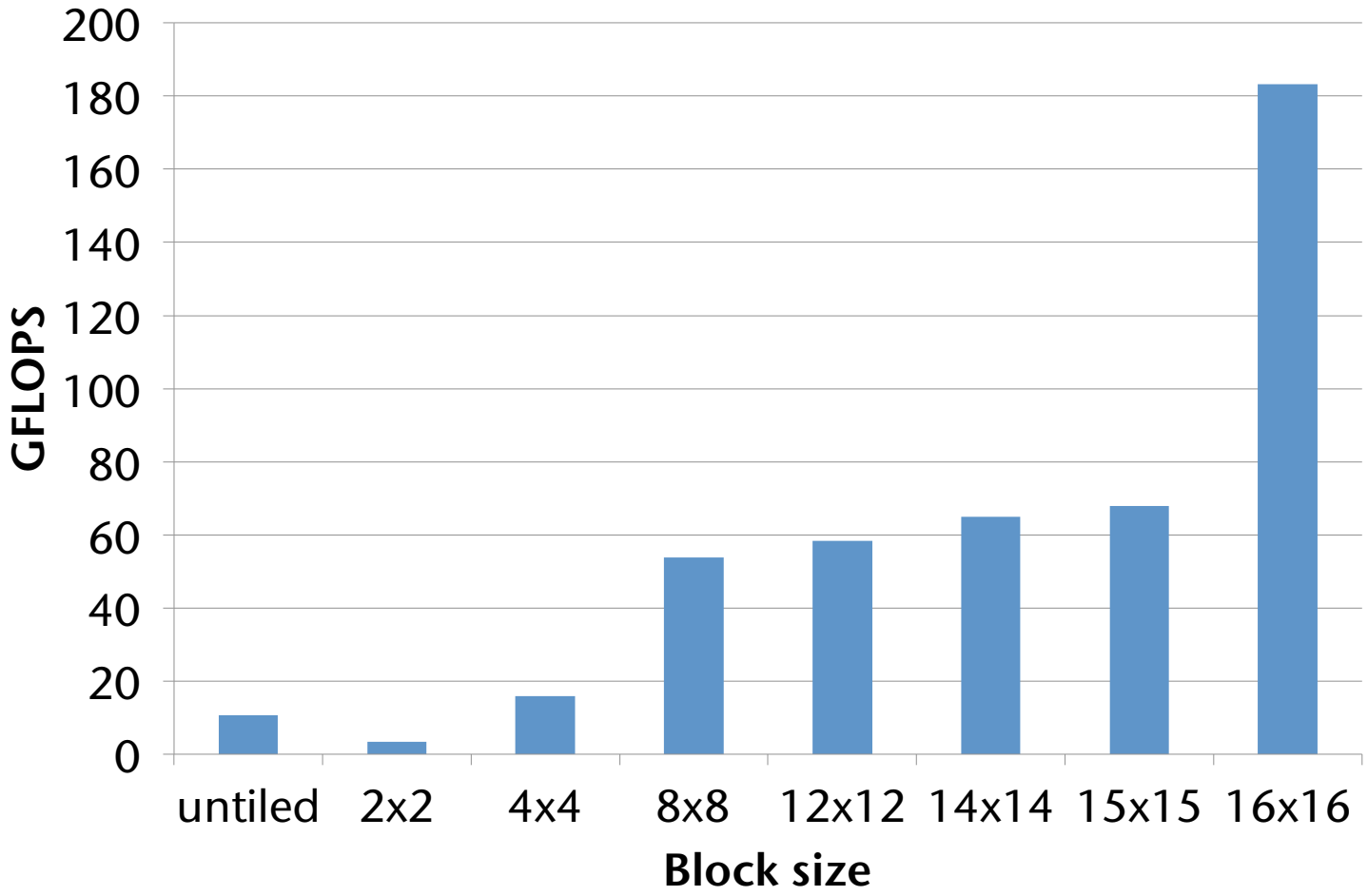
Original matrix
(numbers are addresses)

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

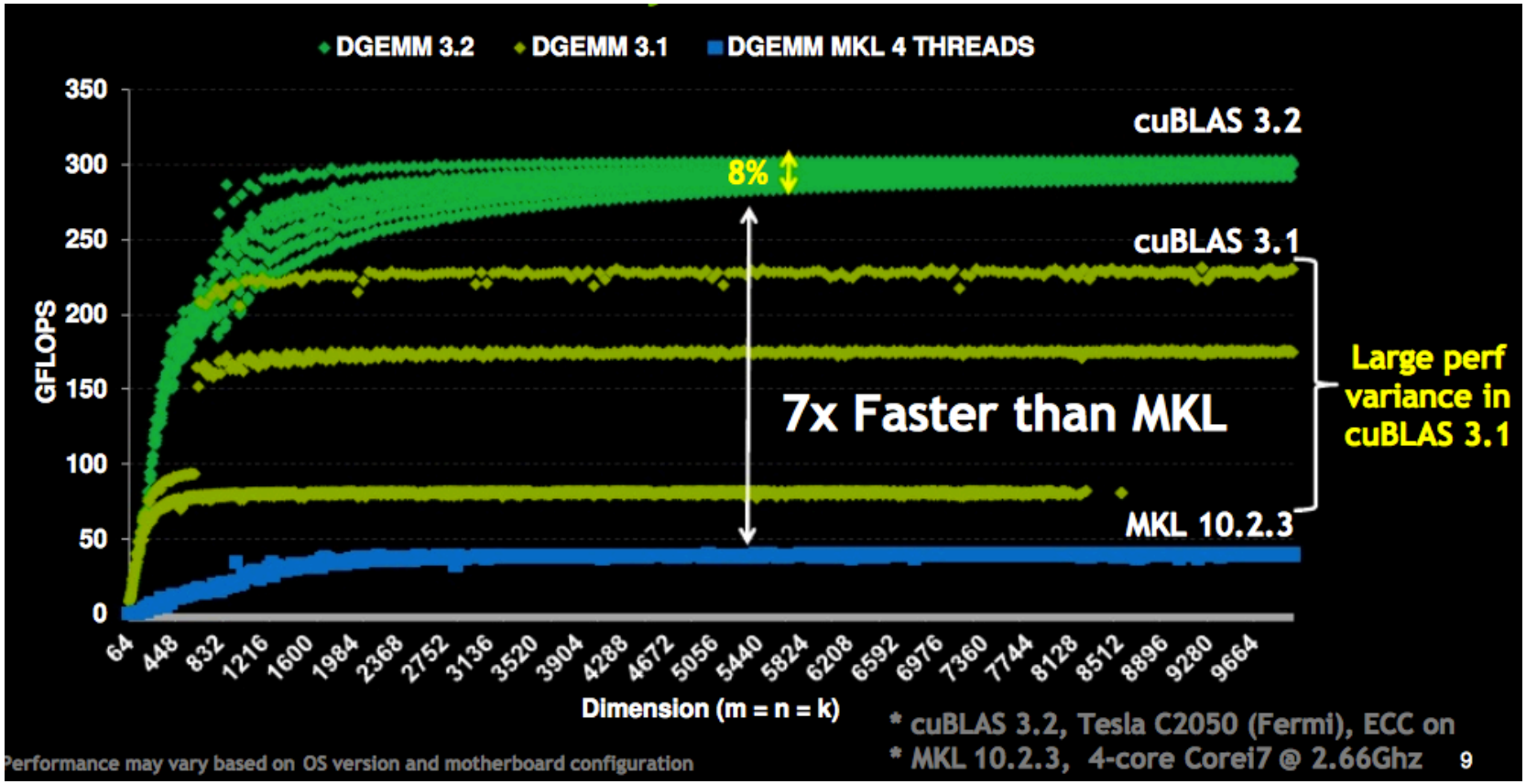
Reorganized
into blocks

0	2	8	10
1	3	9	11
4	6	12	13
5	7	14	15

Effects of Block Size



Comparison with MKL (Intel)



[http://www.scribd.com/doc/47501296/CUDA-3-2-Math-Libraries-Performance]

Strassen's Algorithm

- All "traditional" algos need $O(n^3)$ FLOPs
- Strassen's algorithm: $O(n^{2.81})$
  - Recursive algorithm!
  - See 2nd semester's course "algorithms and data structures"
- Current world record: $O(n^{2.376})$
- Strassen on GPU?
  - Probably not worth it (recursion / complex control flow)

Recap: Strassen's Algorithm

- Task: compute $C = A \cdot B, \quad A, B \in \mathbb{R}^{n \times n}$
- Idea : divide-and-conquer
  - Partition A, B, C in 2x2 block matrices

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

mit $a_{ij}, b_{ij}, c_{ij} \in \mathbb{R}^{\frac{n}{2} \times \frac{n}{2}}$

- Multiplication gives:

$$c_{11} = a_{11} b_{11} + a_{12} b_{21}$$
$$\vdots$$
$$c_{22} = a_{21} b_{11} + a_{22} b_{22}$$

- Which amounts to 8 matrix multiplications of size $\frac{n}{2} \times \frac{n}{2}$

- The trick: compute some (seemingly tedious) intermediate products

$$Q_1 \equiv (a_{11} + a_{22})(b_{11} + b_{22})$$
$$Q_2 \equiv (a_{21} + a_{22})b_{11}$$
$$Q_3 \equiv a_{11}(b_{12} - b_{22})$$
$$Q_4 \equiv a_{22}(-b_{11} + b_{21})$$
$$Q_5 \equiv (a_{11} + a_{12})b_{22}$$
$$Q_6 \equiv (-a_{11} + a_{21})(b_{11} + b_{12})$$
$$Q_7 \equiv (a_{12} - a_{22})(b_{21} + b_{22})$$

- Now we can compute the c_{ij}'s like so:

$$c_{11} = Q_1 + Q_4 - Q_5 + Q_7$$
$$c_{12} = Q_2 + Q_4$$
$$c_{21} = Q_3 + Q_5$$
$$c_{22} = Q_1 + Q_3 - Q_2 + Q_6$$

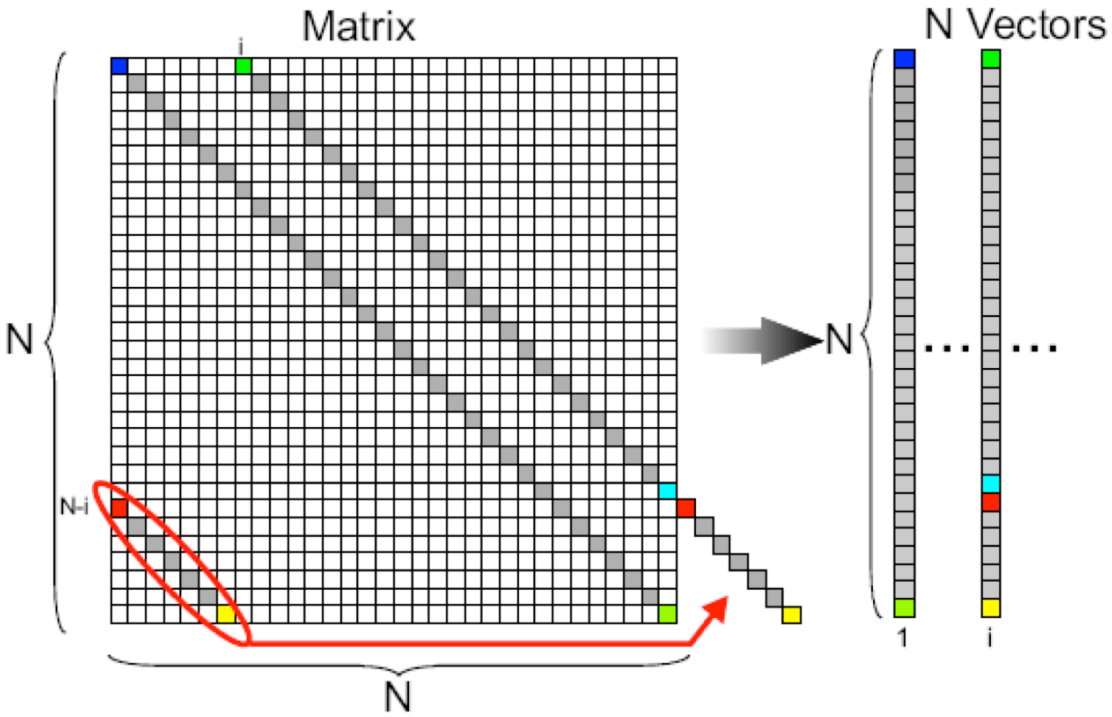
- Computational complexity:

$$T(n) = 7T\left(\frac{n}{2}\right) + cn^2 \in O\left(n^{2.8\ldots}\right)$$

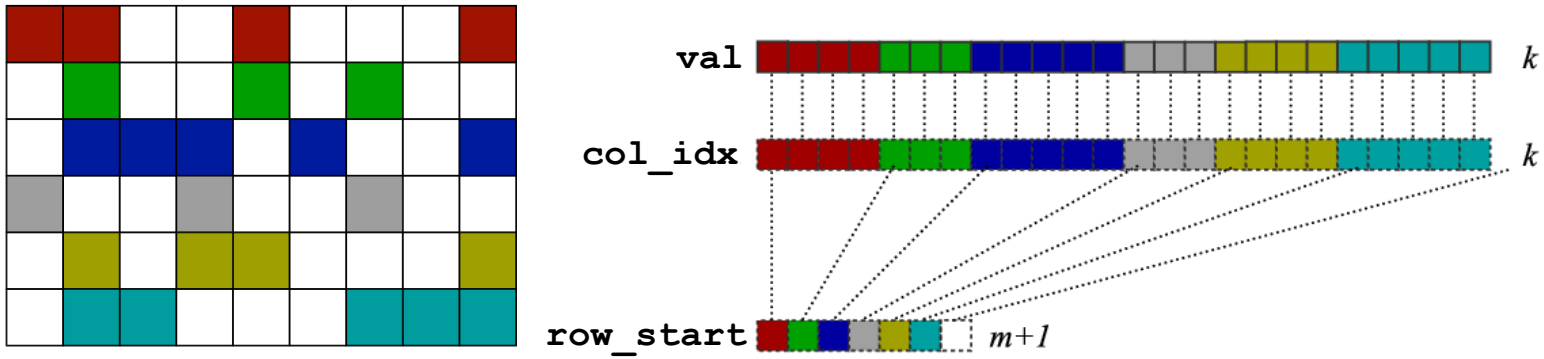
  - Assumption here: multiplications are the expensive operation

Sparse Matrices

- Just some remarks
- Frequent case: sparse band matrices
  - Represent matrix as a number of vectors
  - Devise new parallel algorithm (one thread per row is inefficient)

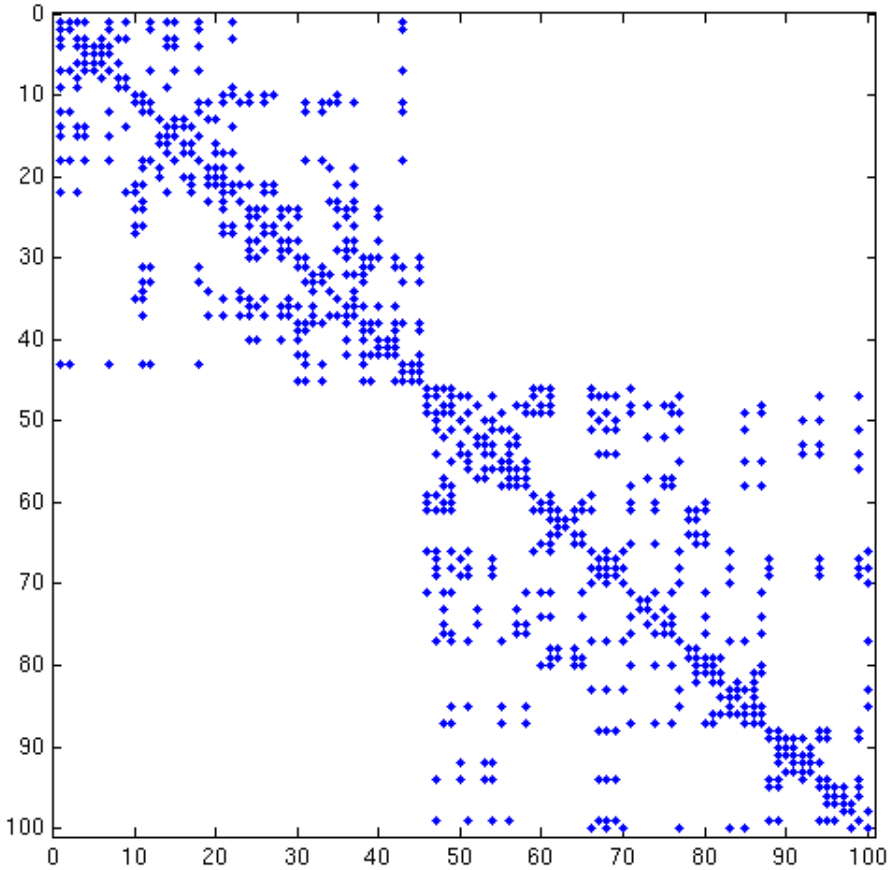


- "Unstructured" sparse matrices:
  - Most common storage format is Compressed Sparse Row (CSR)




```
struct {
  int n_rows;              // number of rows
  int nnz;                 // total number of non-zero elements
  int row_start[n_rows+1];
  int col_idx[nnz];
  double val[NNZ];
}
```

- Many more kinds of sparse matrices
  - Specialized representation / algorithms for each of them?



Summary

- Simple performance models can aid in understanding
- Two ratios are key:
  - Arithmetic (computational) intensity = $\frac{\#\ \text{flops}}{\#\ \text{mops}}$
    - "flops" = floating point operations, "mops" = memory operations
  - Machine balance = $\frac{\text{Tflops/sec}}{\text{GB/sec}}$