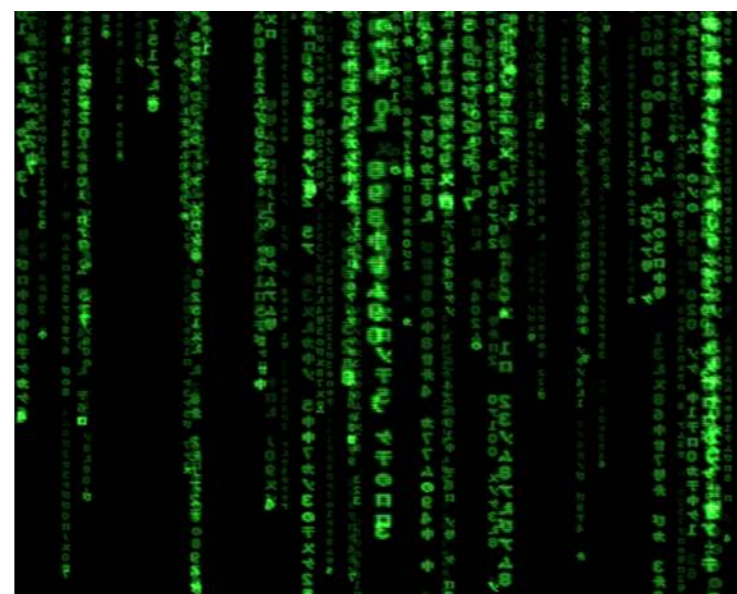




# Massively Parallel Algorithms

## Dense Matrix Algorithms



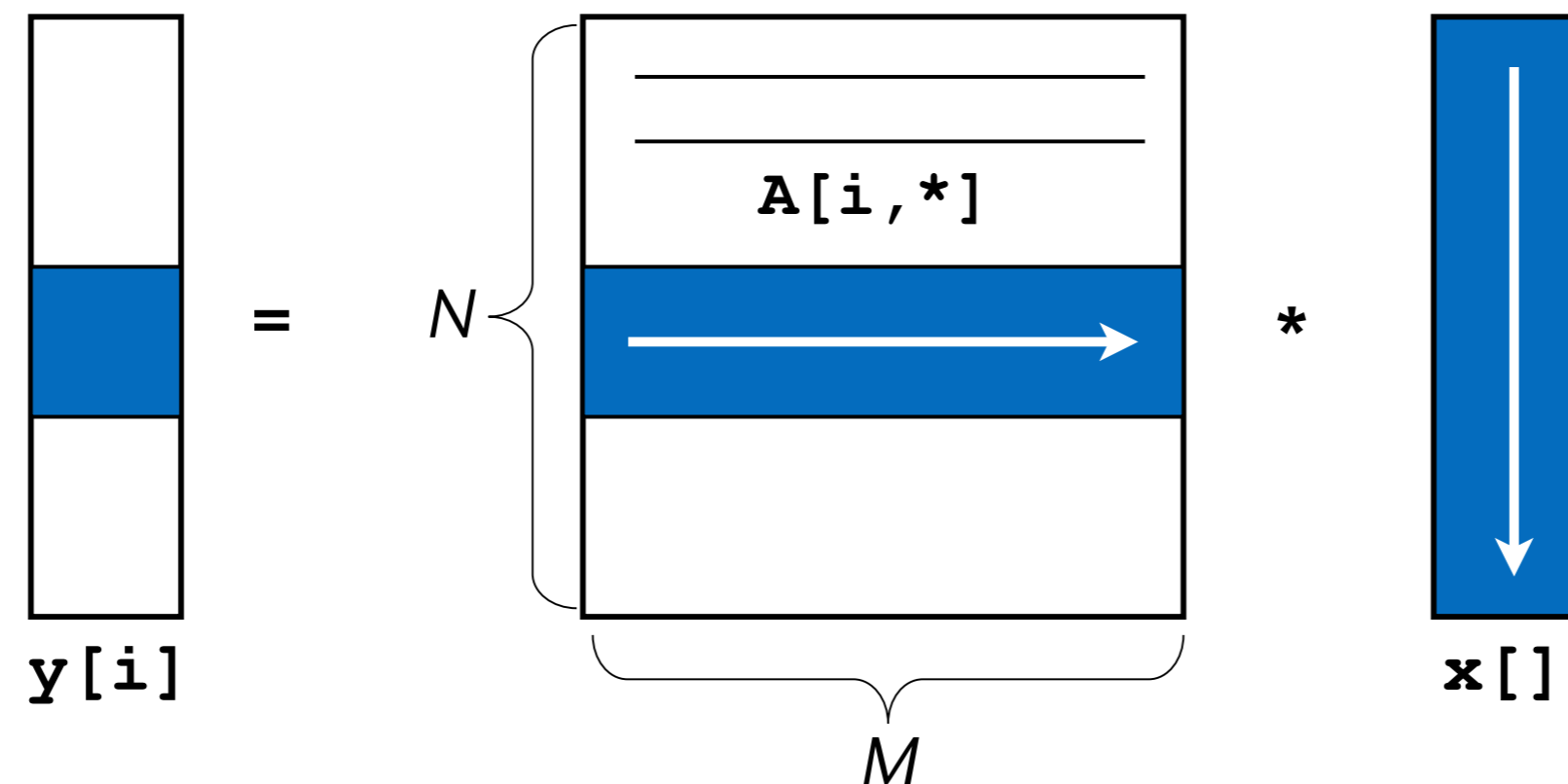
G. Zachmann  
University of Bremen, Germany  
[cgvr.cs.uni-bremen.de](http://cgvr.cs.uni-bremen.de)

# Warming Up: Matrix-Vector Product

- Given matrix  $A$ , and vector  $\mathbf{x}$ , 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



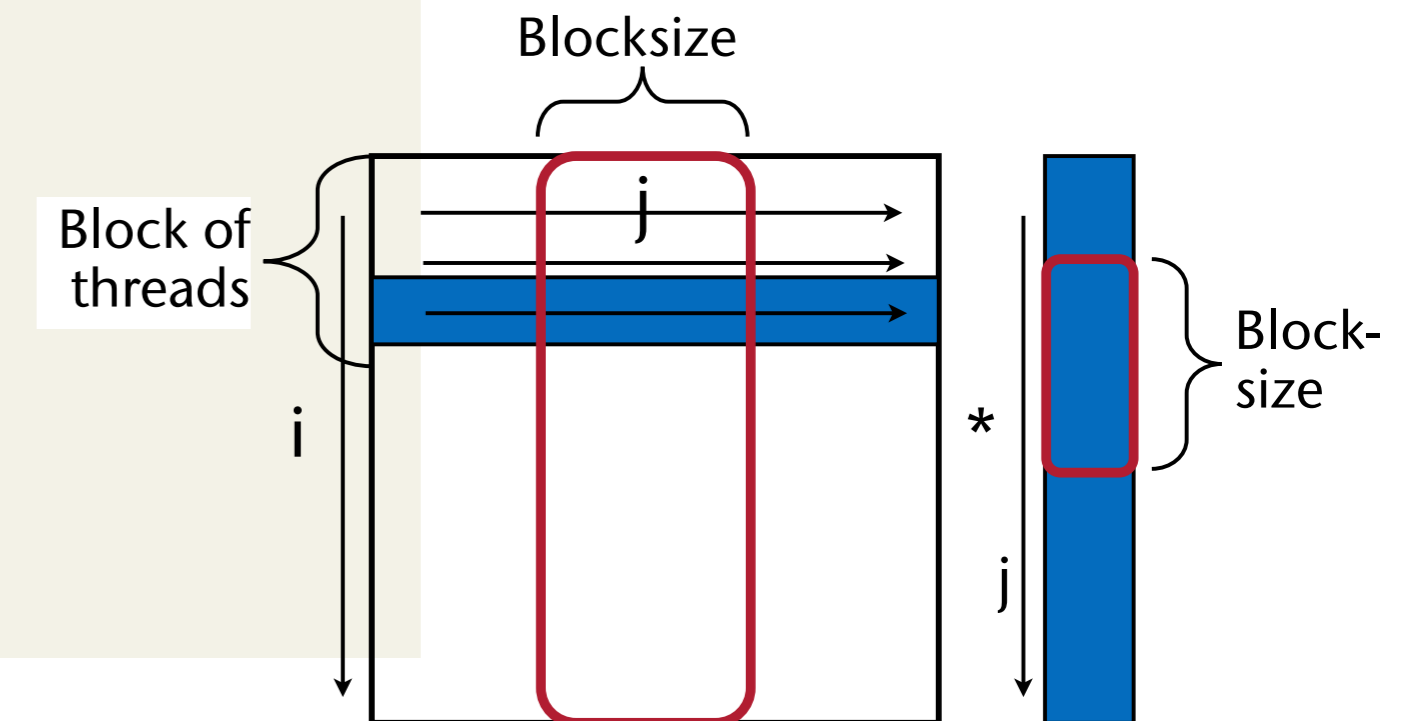
- Observation: all threads use the same data from  $\mathbf{x}$   $\rightarrow$  shared memory

# Algorithm for First Attempt (One Thread per Row)

```

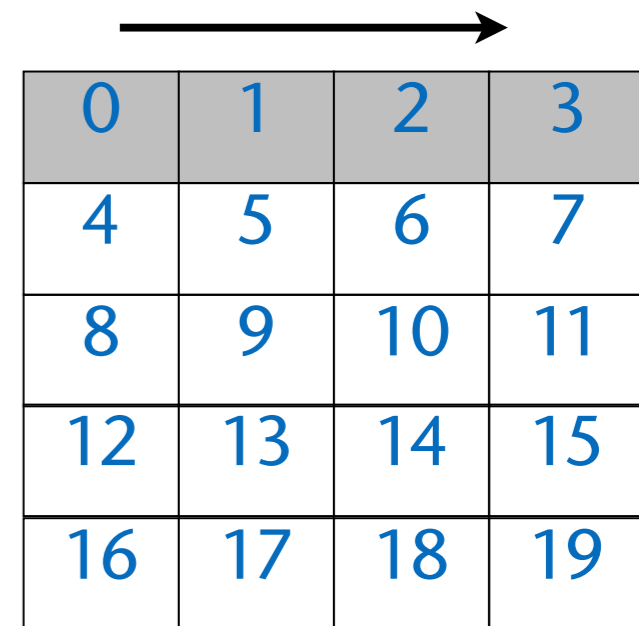
multMatrixVector( const float * A, const float * x,
                  const int n_columns, float * y )
{
    __shared__ x_cache[ THREADS_PER_BLOCK ];
    float yi = 0.0; // output of each thread
    int i = threadIdx.x + blockIdx.x * blockDim.x; // row index
    for ( int j = 0; j < n_columns; j += THREADS_PER_BLOCK )
    {
        // 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[k];
        }
    }
    y[i] = yi;
}

```



- For sake of clarity, we assume  $M, N = \text{multiple of block-size}$

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

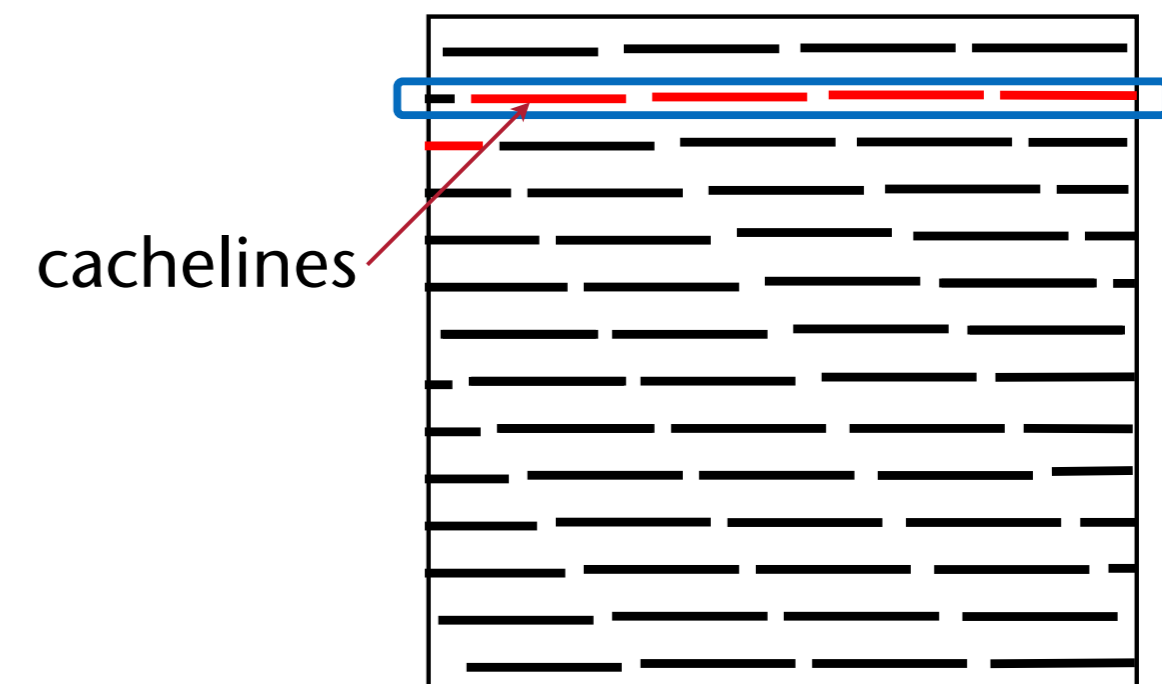


0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

```

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

```



# Coalesced Memory Access

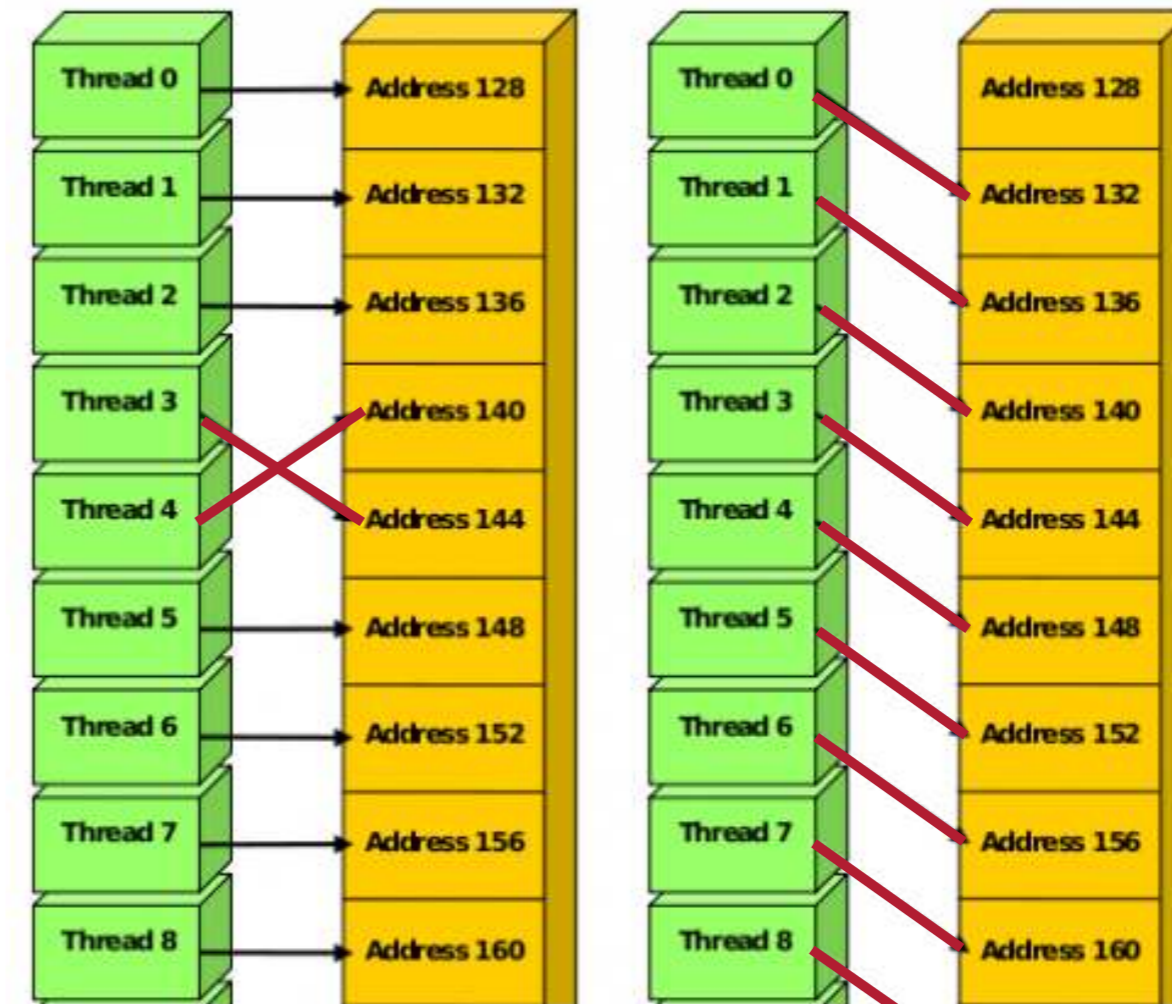
- One of the most important optimization techniques for massively parallel algorithm design on GPUs and – to some degree – CPUs!

Coalesced memory accesses



Aligned and sequential memory access (a few gaps are OK)

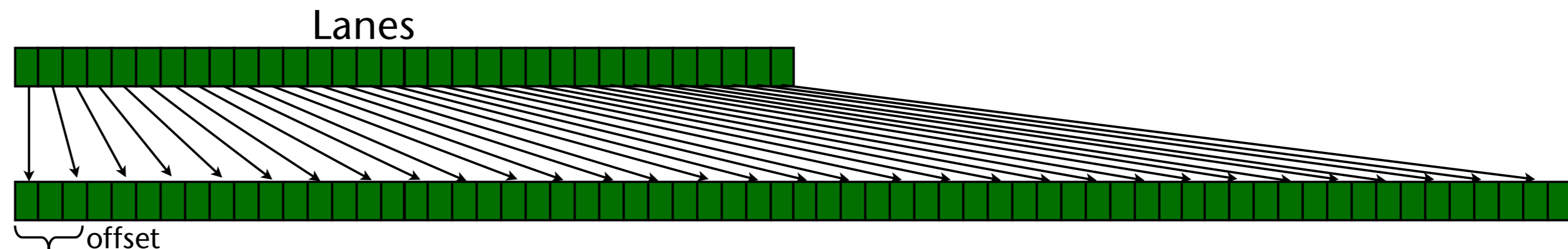
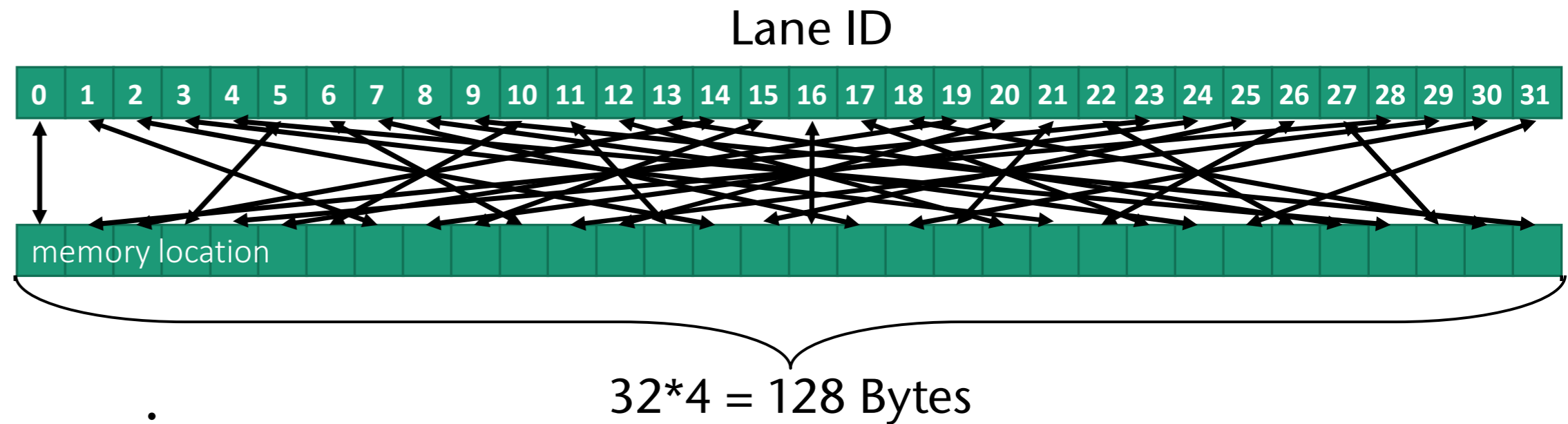
Uncoalesced memory accesses



Aligned but not sequential

Sequential but not aligned

- In more detail: so long as memory access stays within a warp bound, everything is fine
- As fast as sequential memory access (i.e., counts as coalesced, too)
- The following access pattern gives only  $\frac{1}{n}$ -th of the transfer bandwidth, where  $n = \text{offset}$



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

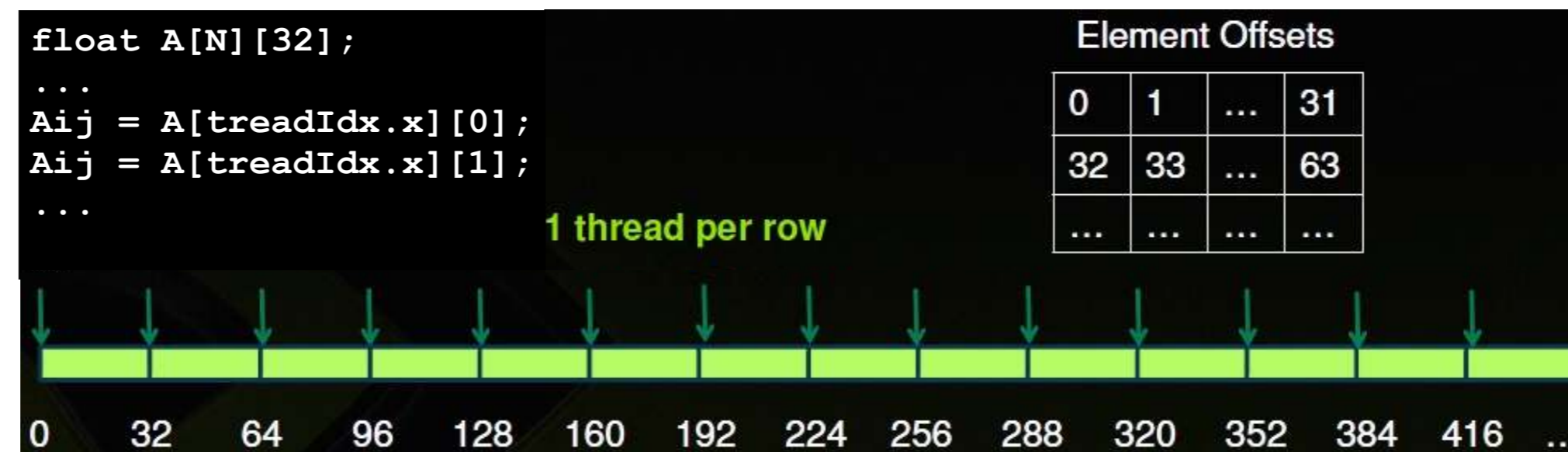
- Consider the following piece in a kernel (e.g., matrix  $\times$  vector):

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

- Generally, most natural access pattern for direct port of C/C++ code

➤ Problem: **uncoalesced** access pattern

- Elements read on 1<sup>st</sup> SIMT access: 0, 32, 64, ...
- Elements read on 2<sup>nd</sup> SIMT access: 1, 33, 65, ...
- Also, extra data will be transferred in order to fill the cache line size

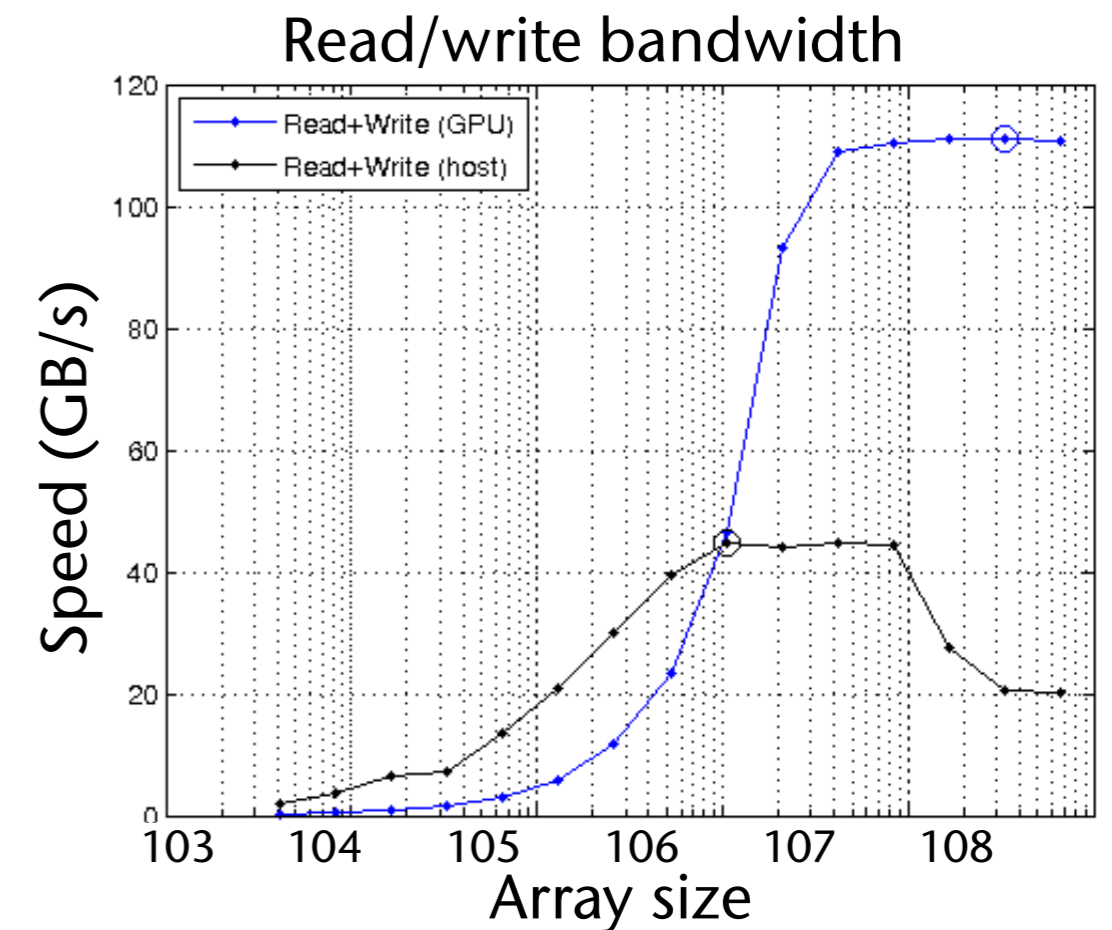


# How to Achieve Coalesced Access

- Addresses from a warp are converted into memory line requests
  - Line sizes: 32B (= 32x `char`) and 128B (= 32x `float`)



- Goal is to maximally utilize the bytes in these lines
- GPU wins over CPU at memory access, if it is "streamed" = coalesced
  - Hence, "stream programming architecture"





# Column Major (Transposed) 2D Array Access Pattern

- **Column major** := store a *logical row* in a *physical column*

0	5	10	15
1	6	11	24
2	7	12	17
3	8	13	18
4	9	14	19

- I.e.,  $A_{00} \rightarrow A[0][0]$ ,  $A_{01} \rightarrow A[1][0]$ ,  $A_{02} \rightarrow A[2][0]$ , ...  
 $A_{10} \rightarrow A[0][1]$ ,  $A_{11} \rightarrow A[1][1]$ ,  $A_{12} \rightarrow A[2][1]$ , ...  
 $A_{20} \rightarrow A[0][2]$ , ...

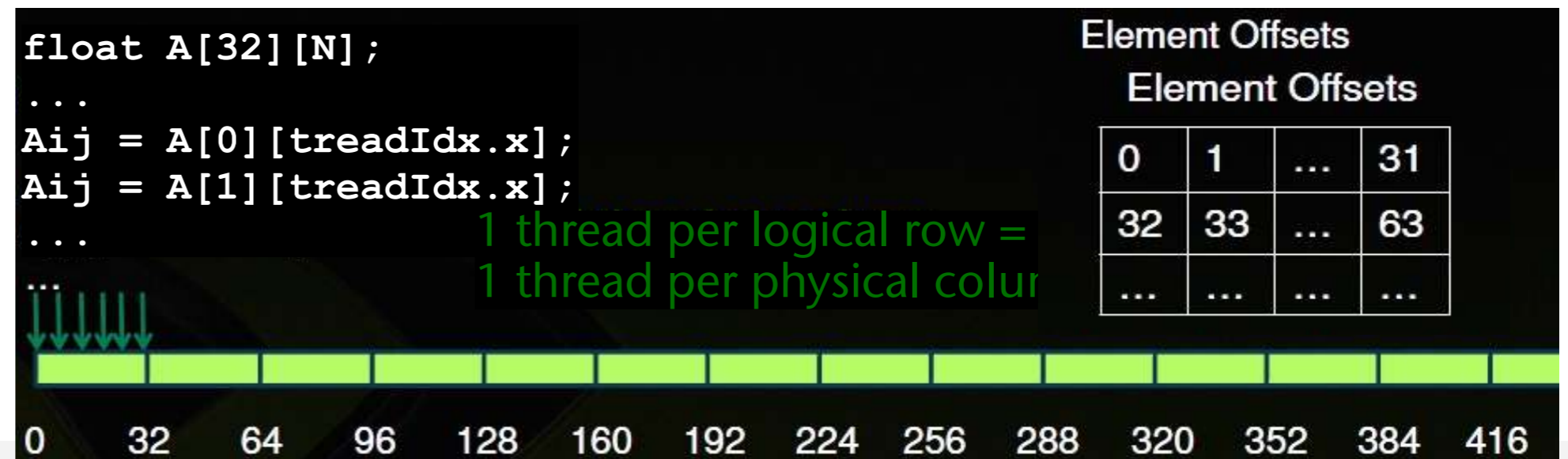
- In general:  $A_{ij}$  is stored at  $A + j * n\_columns + i$

- 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 1<sup>st</sup>  
SIMT access: 0, 1, 2, ..., 31
- Elements read on 2<sup>nd</sup>  
SIMT access: 32, ..., 63



# Array of Struct's or Struct of Arrays?

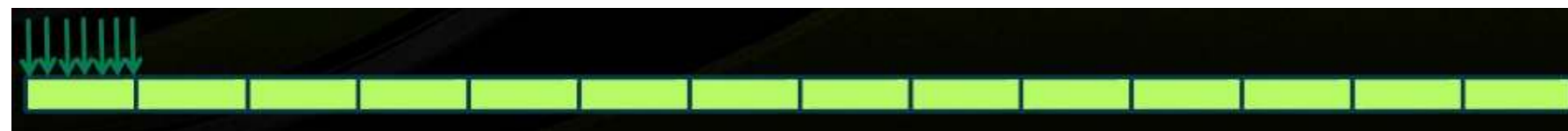
- An **array of structs (AoS)** yields memory accesses like *row major*:



```

struct Point {
    float x, y, z;
};
Point PointList[N];
...
PointList[threadIdx].x = ...
    
```

- A **struct of arrays (SoA)** yields memory accesses like *column major*:



```

struct PointList {
    float x[N];
    float y[N];
    float z[N];
};
...
PointList.x[threadIdx] = ...
    
```

# Modified Matrix\*Vector Algorithm for Column-Major Matrix Storage

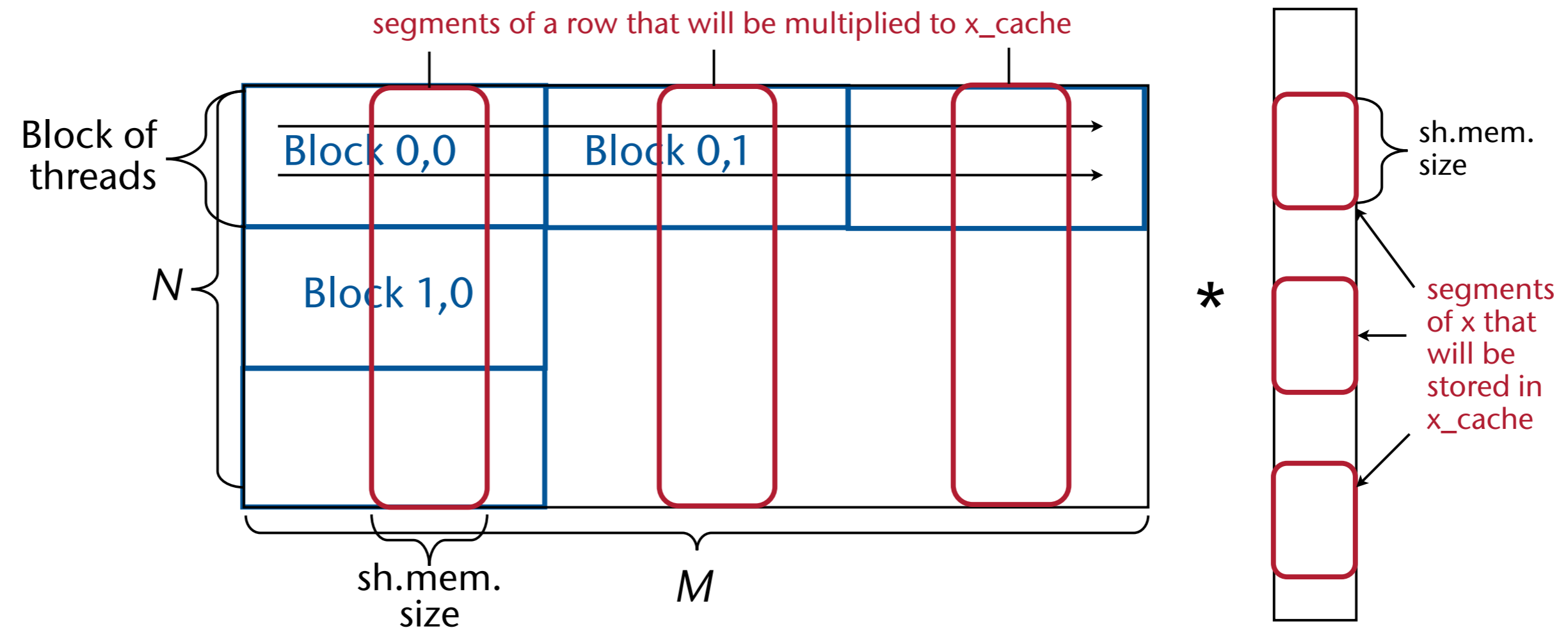
```
multMatrixVector( const float * A, const float * x,
                  const int n_columns, float * y )
{
    __shared__ x_cache[ THREADS_PER_BLOCK ];
    float yi = 0.0; // output of each thread
    int i = threadIdx.x + blockIdx.x * blockDim.x; // row index
    for ( int j = 0; j < n_columns; j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

Note: `n_columns` is still the number of columns of the *logical* matrix, *not* the number of columns of the *physical* matrix!

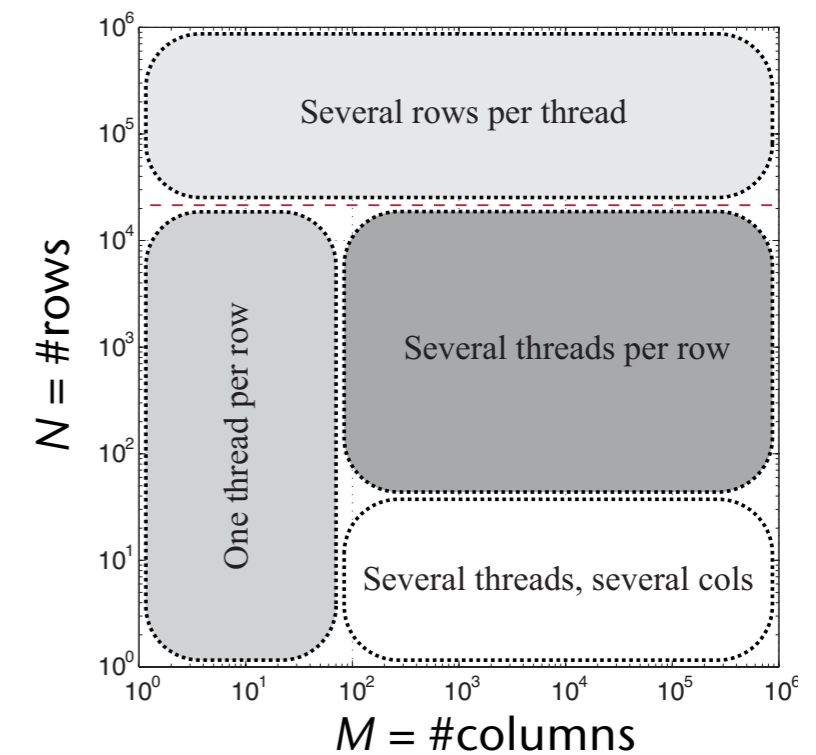
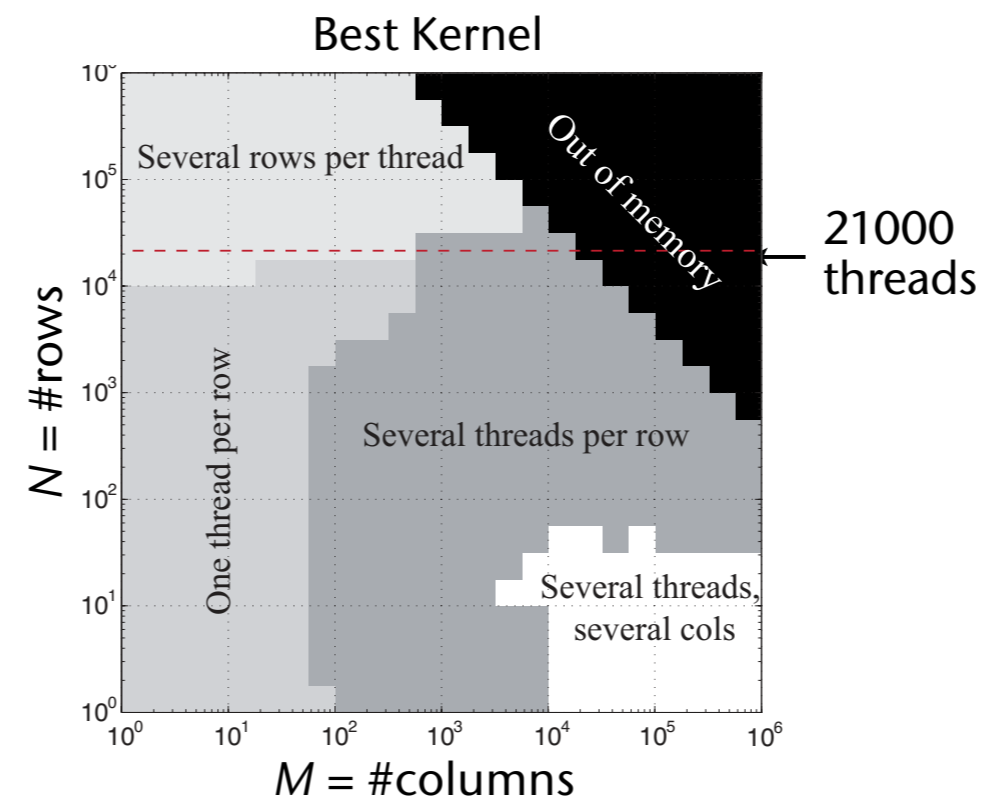
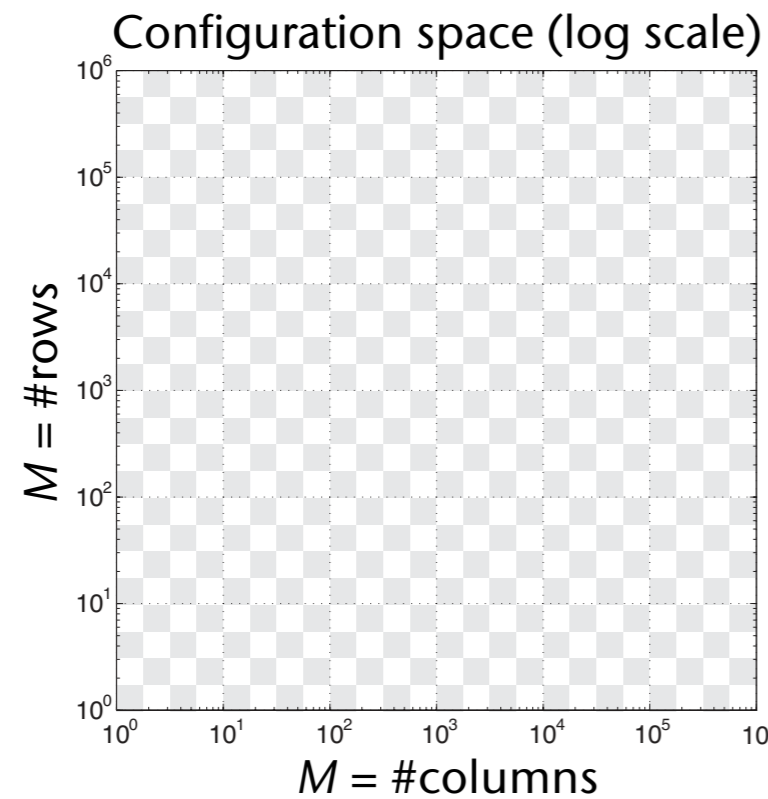
- Note: from now on, we will use **row-major notation** (just for sake of clarity)!
  - But we will assume that an **actual implementation uses column-major!**
  - We expect you to transform everything to column-major
  - Start with small matrices that you can check "by hand"
  - Or implement your code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

# Auto-Tuning

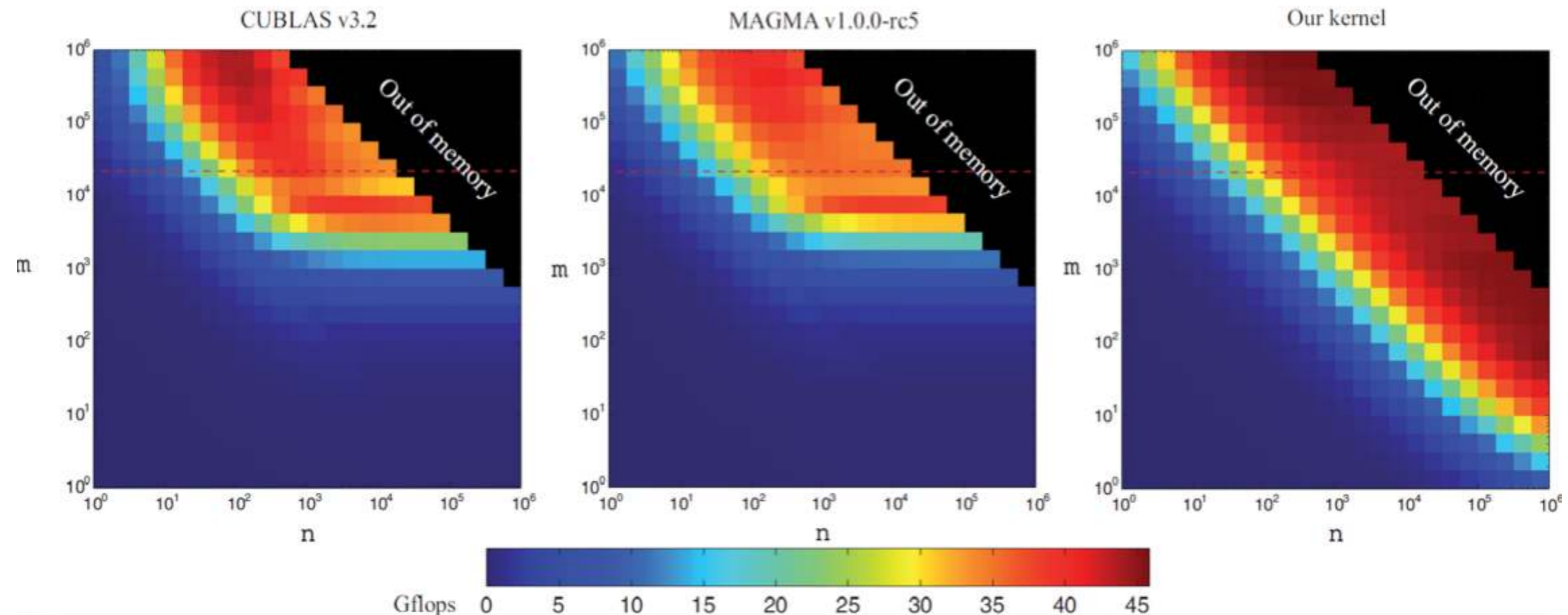
- Do we keep all hardware resources of the GPU busy?
- Example: 14 SMs, each supports 1536 active threads
  - If  $N < 14 \times 1536 = 21504 \rightarrow$  some SMs are idle!
- Idea for the case  $N < 21504$  and  $M$  "not too small": use 2D partitioning of our problem/domain



- All possible domain decomposition 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 per row, each thread handles several rows (2 & 3)
- 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  $n \times m$

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

# Arithmetic Intensity

- **Arithmetic intensity** of an algorithm :=

$$\frac{\text{number of arithmetic operations}}{\text{amount of transferred bytes}}$$

- Sometimes also called **computational intensity**
- Unfortunately, many (most?) algorithms have a low arithmetic intensity → they are **bandwidth limited**



# Complexities of Matrix-Vector Multiplication

- Sequential version:  $O(n^2)$  (assuming  $n=m$ )
- Parallel version:  $O(n)$  parallel time
  - Assuming  $n$  parallel threads, one thread per row (ideal case)

- Arithmetic intensity:

- Assume following simplified (sequential) version:

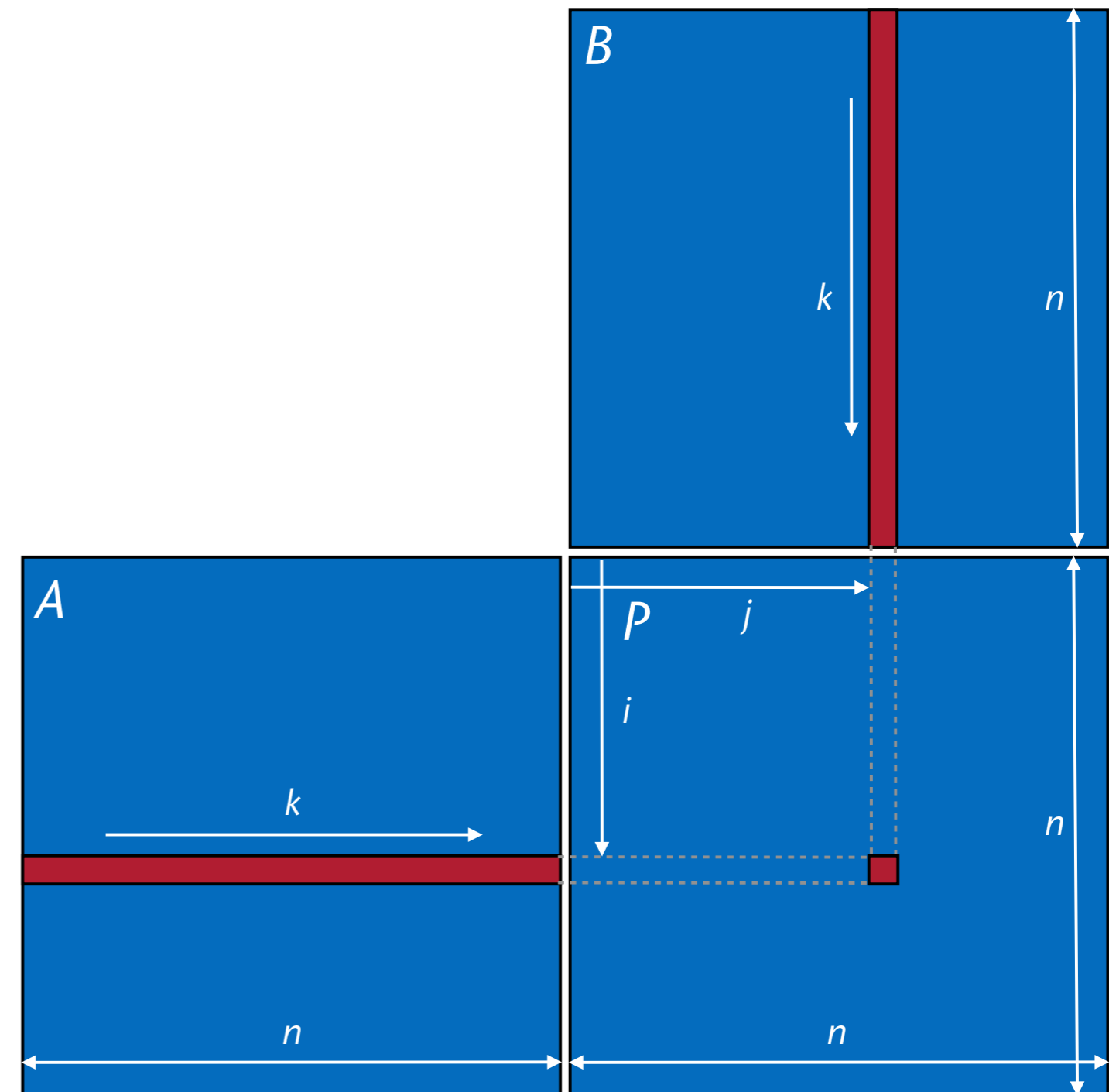
```
load vector x completely into fast memory
for i = 1 ... n:           // assuming n = m
    load row i of A into fast memory
    for j = 1 ... m:
        yi += A[i][j] * x[j]
    store yi in y[i]
```

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

# 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  $n \times n$
- 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
```



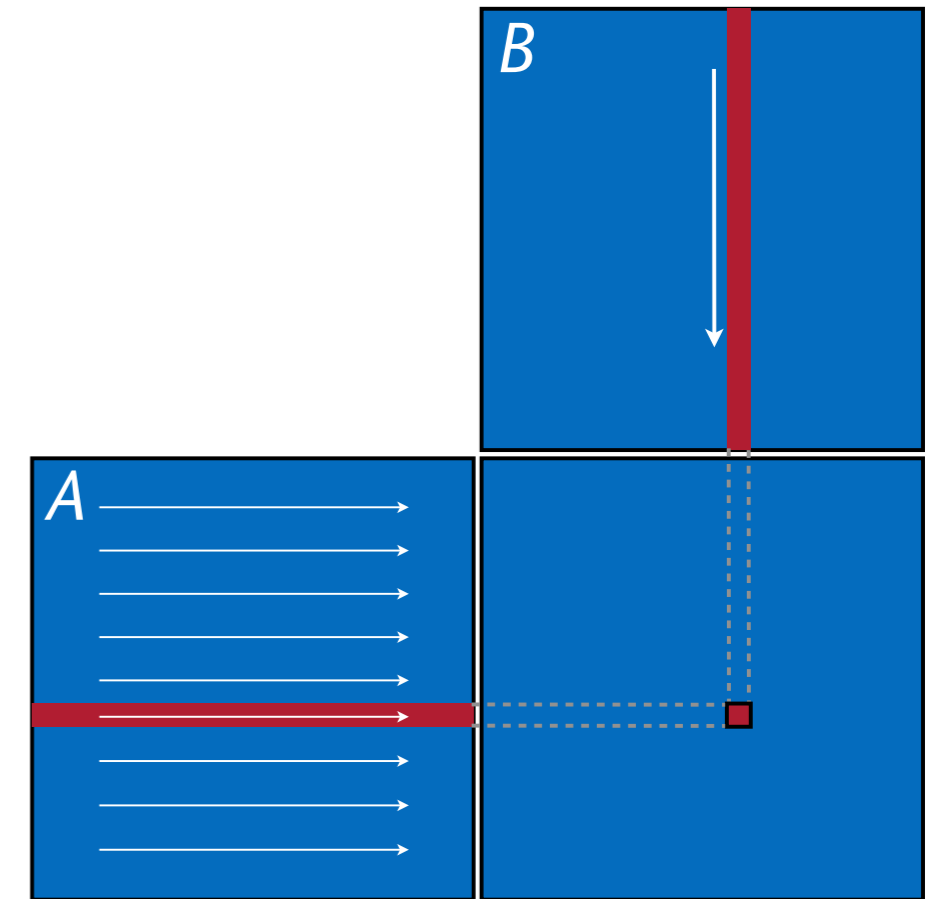
- Complexity:  $O(n^3)$
- Arithmetic intensity:  $a = \frac{2n^3}{2n^3 + n^2} \approx 1$ 
  - Even worse than matrix-vector multiplication!
- Problem: no data re-use!
- Theorem (w/o proof):  
For all iterative (= non-recursive) matrix-matrix multiplication algorithms,  
the upper bound on arithmetic intensity is

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

# 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 in parallel:  
    s = 0.0  
    for k = 1 ... n:  
      s += A[i][k] * B_cache[k]  
    P[i][j] = s
```

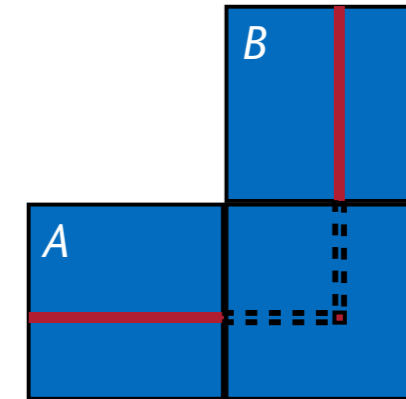


- Arithmetic intensity:  $a = \frac{2n^3}{n^3 + 2n^2} \approx 2$
- Not much better 😞

# Blocked (Tiled) Matrix Multiplication

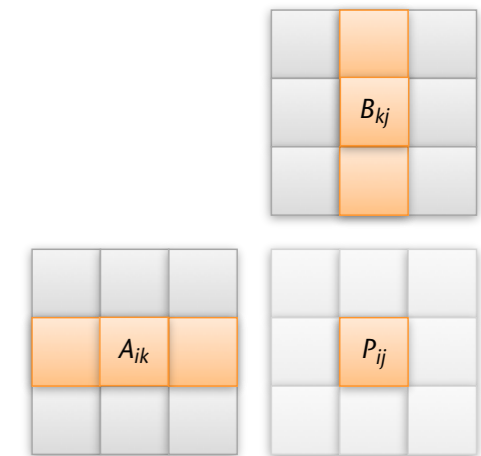
- Remember linear algebra class: the procedure

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



works also for **sub-blocks** of the matrices

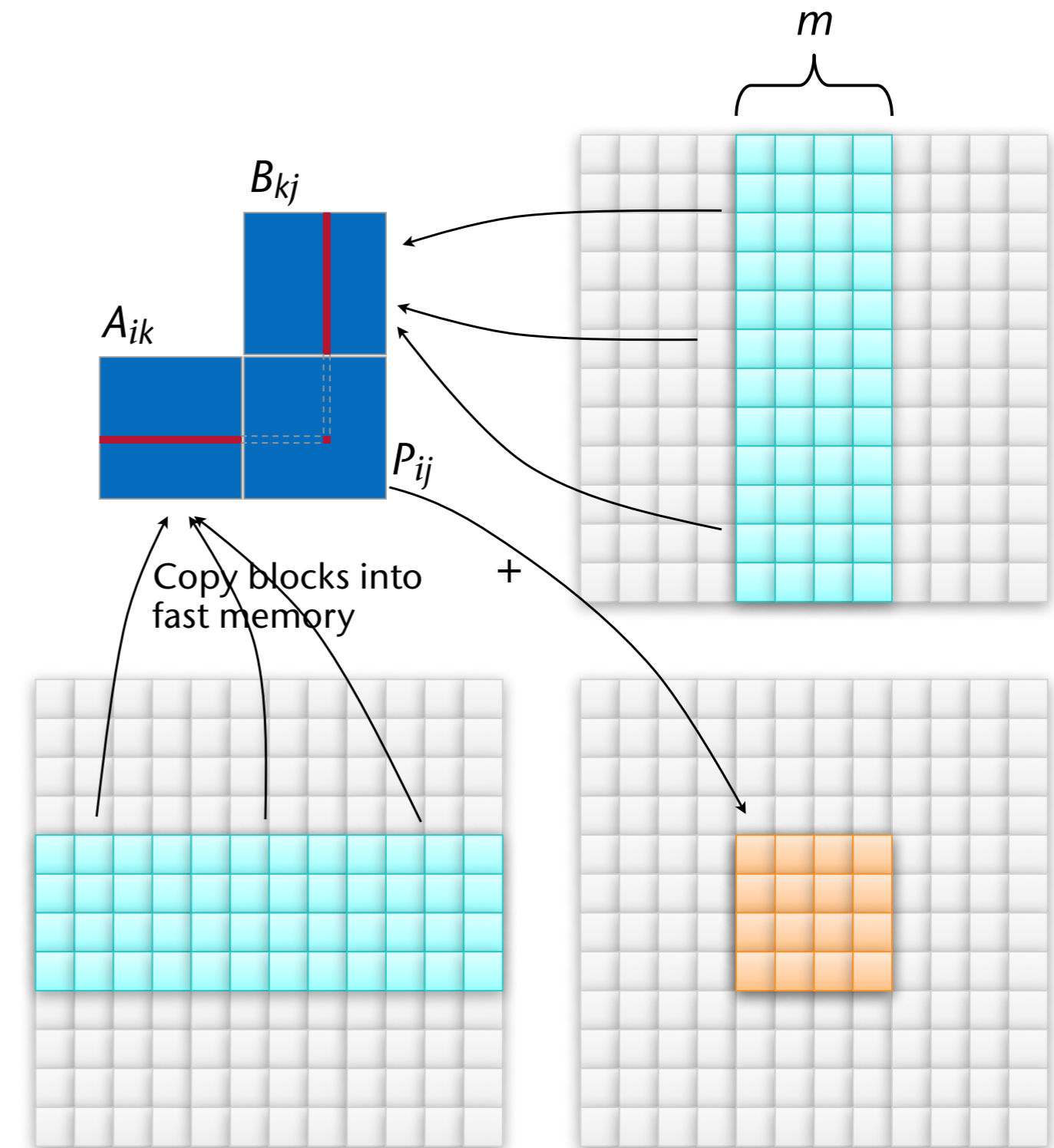
$$P_{ij} = \sum_{k=1}^{n/m} A_{ik} B_{kj}$$



where  $A_{ik}, B_{kj}, P_{ij} \in \mathbb{R}^{m \times m}$  are block matrices of size  $m$

- Assumption:  $n = \text{multiple of } m$ 
  - 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:
  - Load blocks  $A_{ik}$ ,  $B_{kj}$  into shared memory
    - Each thread loads one  $a_{ij}$ , one  $b_{ij}$
  - Perform "row  $\times$  column" over block
  - Accumulate partial results



Actual kernel!

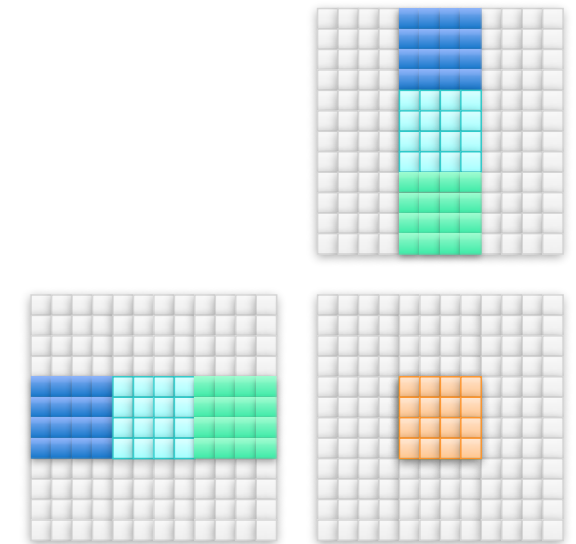
```

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
  
```

Kernel launch

```

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



- 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	14
5	7	13	15



- 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,  $a = \frac{2n^3}{2\frac{n^3}{m}} = m$
- Consequence: make  $m$  large
- Bound on  $m$ : all three blocks  $P_{ij}, A_{ik}, B_{kj}$ , must fit in shared memory

- Calculating  $m$ :

- Assume:  $\sim 2 \text{ TFlops/sec} = 2 \cdot 10^{12} \text{ Flops/sec}$ , and  
 $\sim 200 \text{ GB/sec} = 200 \cdot 10^9 \text{ B/sec}$

- Try to choose  $m$  such that we achieve peak bandwidth & peak Flops/sec

$$m = a = \frac{\# \text{ Flops}}{\# \text{ Loads}} = \frac{\# \text{ Flops/sec}}{\# \text{ Loads/sec}} = \frac{2 \cdot 10^{12} \text{ Flops/sec}}{\frac{200}{4} \cdot 10^9 \text{ B/sec}} = 40$$

$\swarrow$  1 Load = 4 Bytes

- Note: these are very crude estimations, but good for a starting point for the search for the sweet spot
- Consequence: size of shared memory should be at least

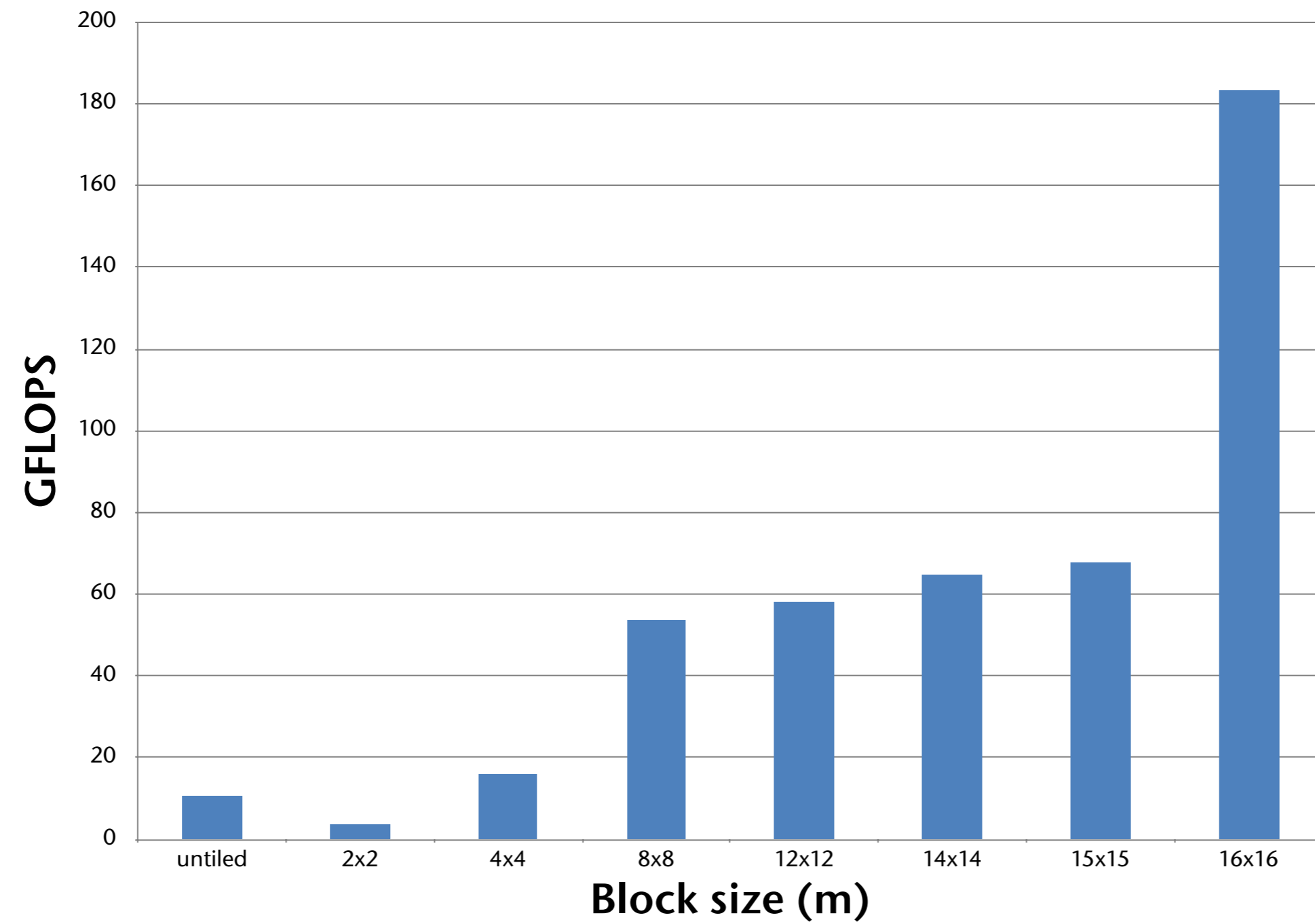
$$3 \cdot 40^2 \cdot 4 \text{ Bytes} = 19.2 \text{ kB}$$

- Otherwise, we would be bandwidth limited

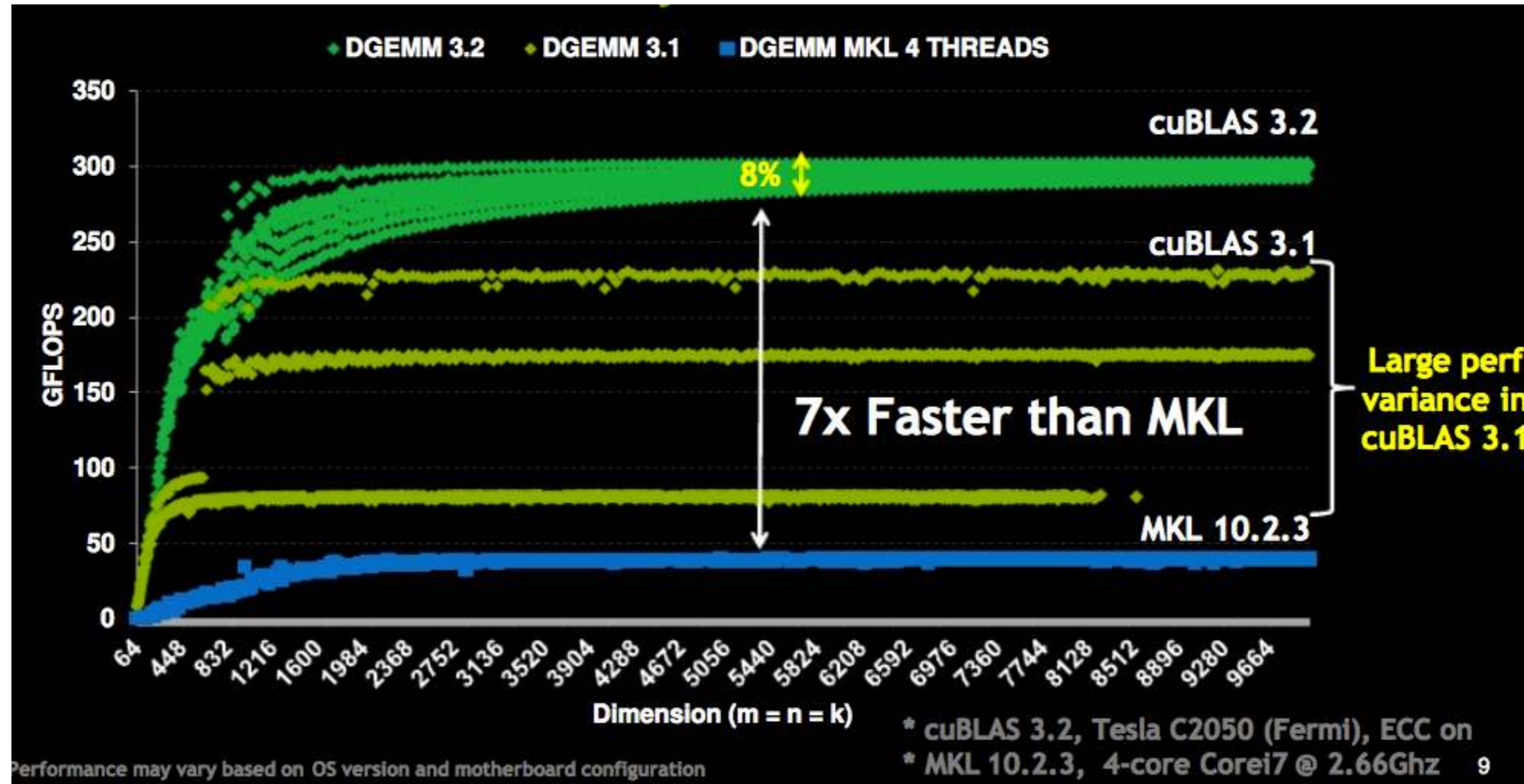
# Summary

- 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 =  $\frac{\text{Tflops/sec}}{\text{GB/sec}}$

# Effects of Block Size



# Comparison with MKL (Intel)



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

# Limitations / Optimality

- Tiling/blocking only works, if the arithmetic 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 load 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*

# Digression: Strassen's Algorithm

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

# Recap: Strassen's Algorithm

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

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

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

- 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:

$$c_{11} = Q_1 + Q_4 - Q_5 + Q_7$$

$$c_{12} = Q_2 + Q_4$$

$$c_{21} = Q_3 + Q_5$$

$$c_{22} = Q_1 + Q_3 - Q_2 + Q_6$$

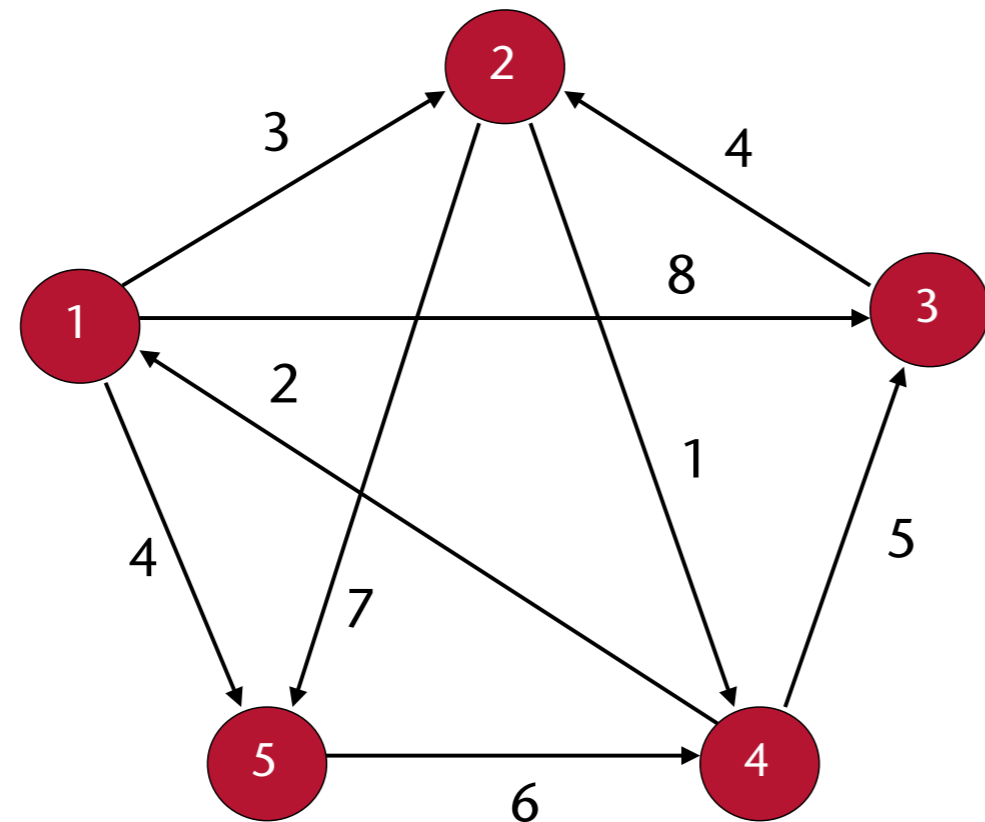
- Computational complexity:

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

- Assumption here: multiplications are the expensive operation
  - However, it needs more addition operations
- 
- How would this perform on a GPU?

# Application: All Pairs Shortest Paths (APSP)

- Given: directed graph  $G = (V, E)$  and a distance function  $\text{dist} : E \rightarrow \mathbb{R}$  where  $V = \text{set of all vertices (nodes)}$ ,  $|V| = n$ , and  $E = \text{set of edges}$
- Goal: compute  $n \times n$  matrix  $D = d_{ij}$  that stores for each pair  $(v_i, v_j)$  the length of the shortest path from  $v_i$  to  $v_j$  in graph  $G$
- Example:



	1	2	3	4	5
1	0	3	8	4	4
2	3	0	6	1	7
3	7	4	0	5	11
4	2	5	5	0	6
5	8	11	11	6	0

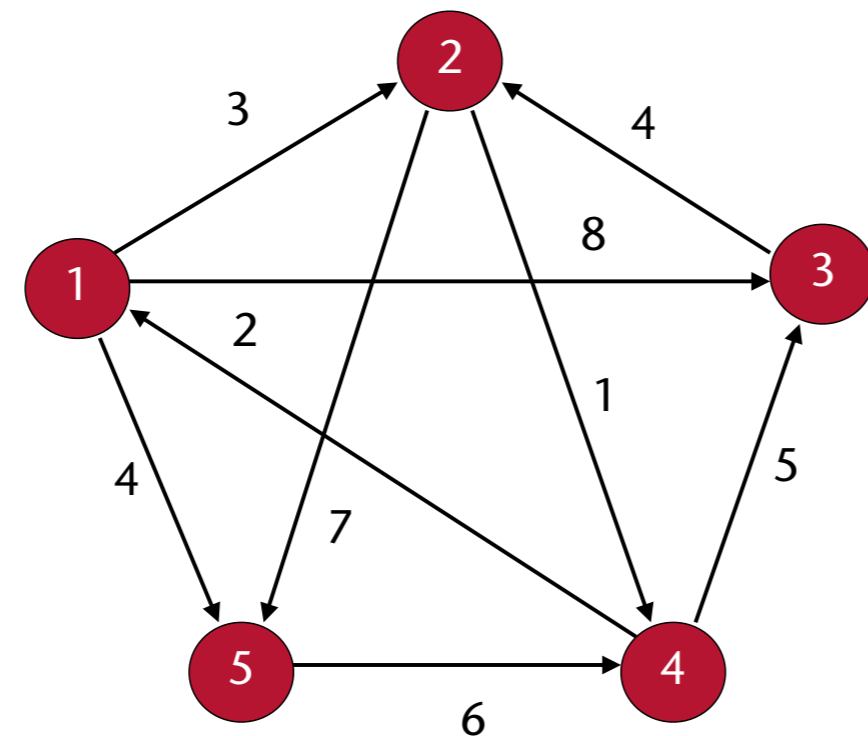
Shortest path matrix D

# The Adjacency Matrix Representation of Directed Graphs

- The **adjacency matrix**  $A$  represents the distance function  $\text{dist}$
- $A$  is an  $n \times n$  matrix  $A = (\delta_{ij})$  where

$$\delta_{ij} = \begin{cases} \text{dist}(v_i, v_j), & \text{if } (v_i, v_j) \in E \\ \infty, & \text{if } (v_i, v_j) \notin E \\ 0, & \text{if } i = j \end{cases}$$

- Example:



	1	2	3	4	5
1	0	3	8	$\infty$	4
2	$\infty$	0	$\infty$	1	7
3	$\infty$	4	0	$\infty$	$\infty$
4	2	$\infty$	5	0	$\infty$
5	$\infty$	$\infty$	$\infty$	6	0

Adjacency matrix

# The Shortest Paths Property

- We will now extend the simple, edge-based distance function to a distance function  $\text{dist}'$  on paths

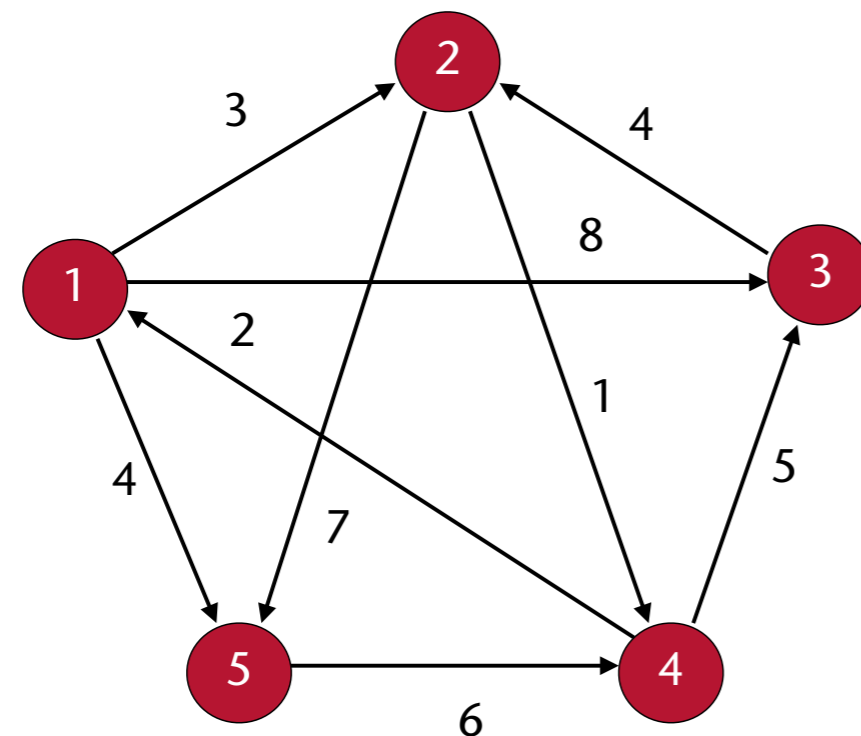
- Define

$$\text{dist}'(p_{ij}^1) = \begin{cases} 0, & i = j \\ \delta_{ij}, & i \neq j \end{cases}$$

- Consider a shortest path  $p_{ij}^k$  from  $v_i$  to  $v_j$  such that  $|p_{ij}^k| \leq k$ , i.e.,  $p_{ij}^k$  can have most  $k$  edges
  - Let  $(v_l, v_j)$  be the last edge of path  $p_{ij}^k$
  - Then, there must be a shortest path  $p_{il}^{k-1}$  from  $v_i$  to  $v_l$  (optimal substructure!)
- Therefore,  $\exists l : \text{dist}'(p_{ij}^k) = \text{dist}'(p_{il}^{k-1}) + \delta_{lj}$

# A Simple Algorithm for APSP

- Given the adjacency matrix  $A$ , compute a series of matrices  $D^1=A, D^2, \dots, D^{n-2}, D^{n-1}$  where matrix  $D^k = \text{dist}'(p_{ij}^k)$  contains lengths of shortest paths in  $G$  with at most  $k$  edges
- Final matrix  $D^{n-1}$  contains the actual shortest paths in  $G$
- Example:



	1	2	3	4	5
1	0	3	8	$\infty$	4
2	$\infty$	0	$\infty$	1	7
3	$\infty$	4	0	$\infty$	$\infty$
4	2	$\infty$	5	0	$\infty$
5	$\infty$	$\infty$	$\infty$	6	0

Adjacency matrix

	1	2	3	4	5
1	0	3	8	4	4
2	3	0	6	1	7
3	$\infty$	4	0	5	11
4	2	5	5	0	6
5	8	$\infty$	11	6	0

Matrix  $D^2$

# The Algorithm

```
A = adjacency matrix
D1 = A
for k = 2 to n-1:
    Dk = ExtendPaths (Dk-1, A)
return Dk
```

```
ExtendPaths ( D, A )
```

```
In: A (with  $\delta_{ij}$ ) =  $n \times n$  adj. matrix
```

```
Out: E (with  $e_{ij}$ ) =  $n \times n$  dist. matrix
```

```
for i = 1 to n:
```

```
    for j = 1 to n:
```

```
         $e_{ij} = d_{ij}$ 
```

```
        for l = 1 to n:
```

```
             $e_{ij} = \min\{e_{ij}, d_{il} + \delta_{lj}\}$ 
```

```
return E
```

```
MatrixMultiply ( B, A )
```

```
In: A = ( $\delta_{ij}$ ) =  $n \times n$  input matrix
```

```
Out: C = ( $c_{ij}$ ) =  $n \times n$  matrix product
```

```
for i = 1 to n:
```

```
    for j = 1 to n:
```

```
         $c_{ij} = 0$ 
```

```
        for l = 1 to n:
```

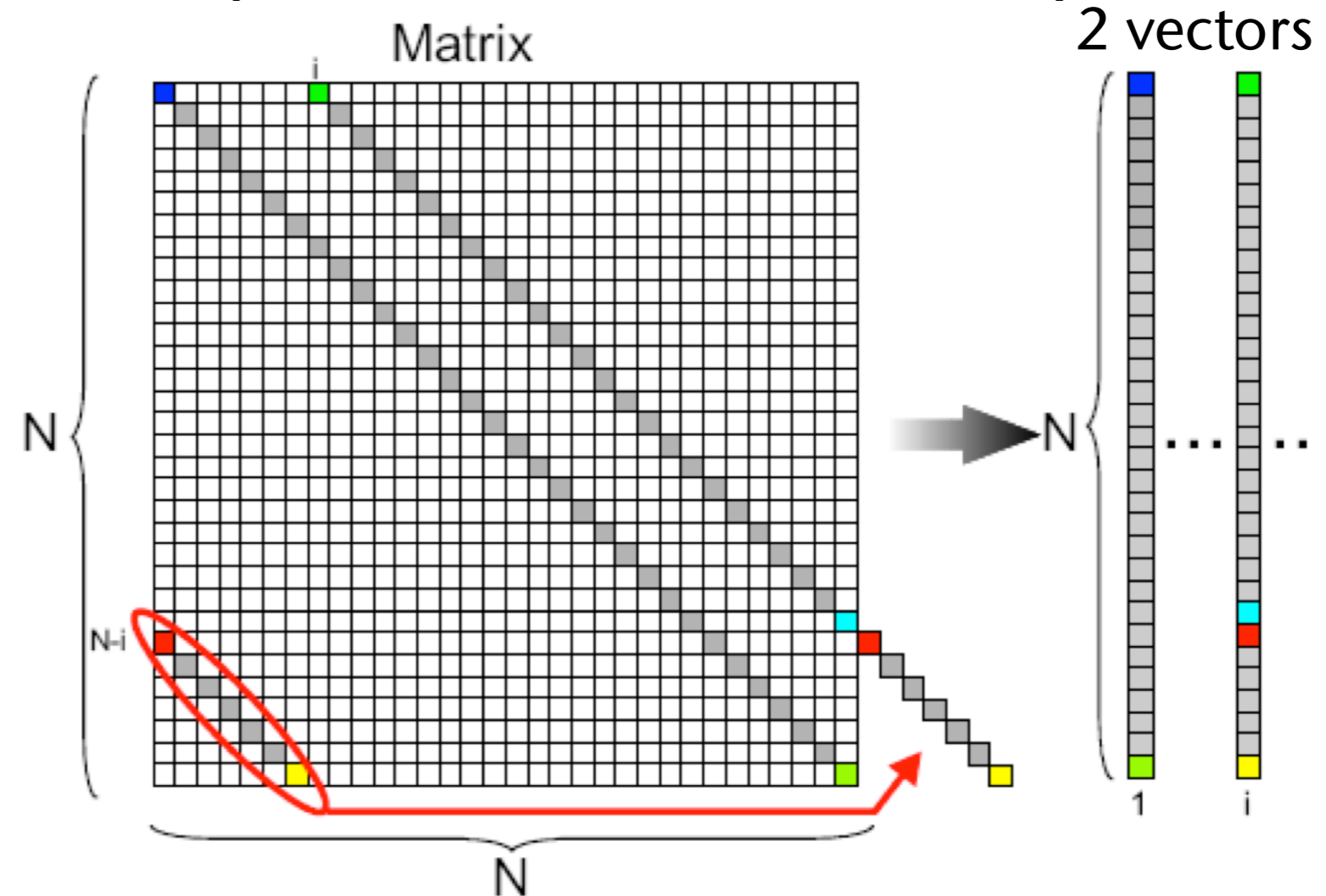
```
             $c_{ij} = c_{ij} + a_{il} \cdot b_{lj}$  (*)
```

```
return C
```

- Notice the similarity with matrix multiplication
  - We can adapt our fast GPU-based matrix multiplication code to solve the APSP problem quite easily (just replace the operators in line (\*))

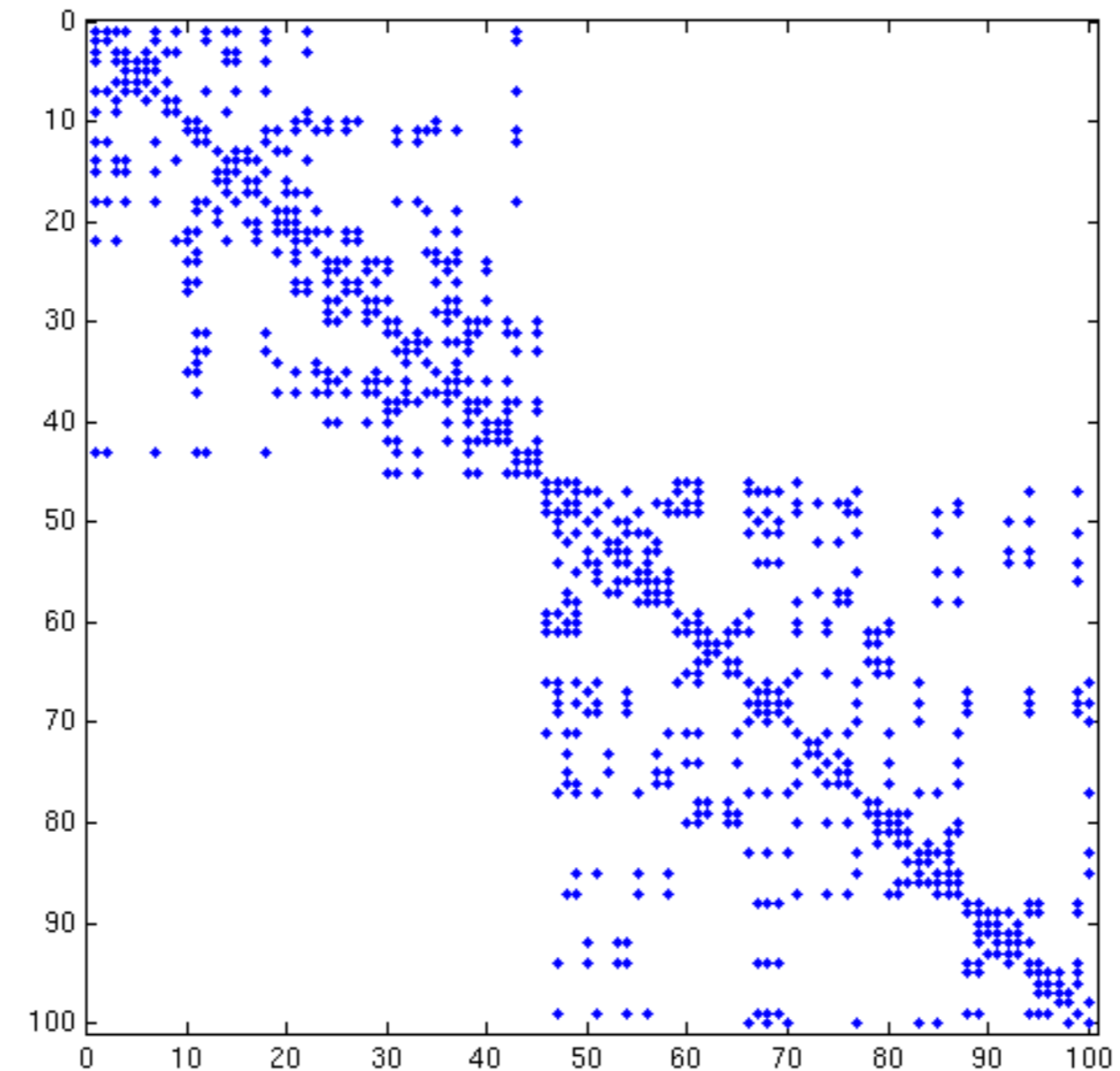
# A Word on Sparse Matrices

- Just some remarks
- Frequent case: sparse band matrices
  - Represent matrix as a number of vectors
  - Devise specialized parallel algorithm (similar to vector addition)





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



- One of the biggest increments in the GPU's architecture
- On Volta architecture, each SM has:
  - 64 FP32 cores
  - 64 Int32 cores
  - 32 FP64 cores
  - 8 tensor cores
- Numbers vary a lot from generation to generation!
- Specifically integrated to speed up machine learning



# The GA100 Architecture, Just FYI



GA100 architecture

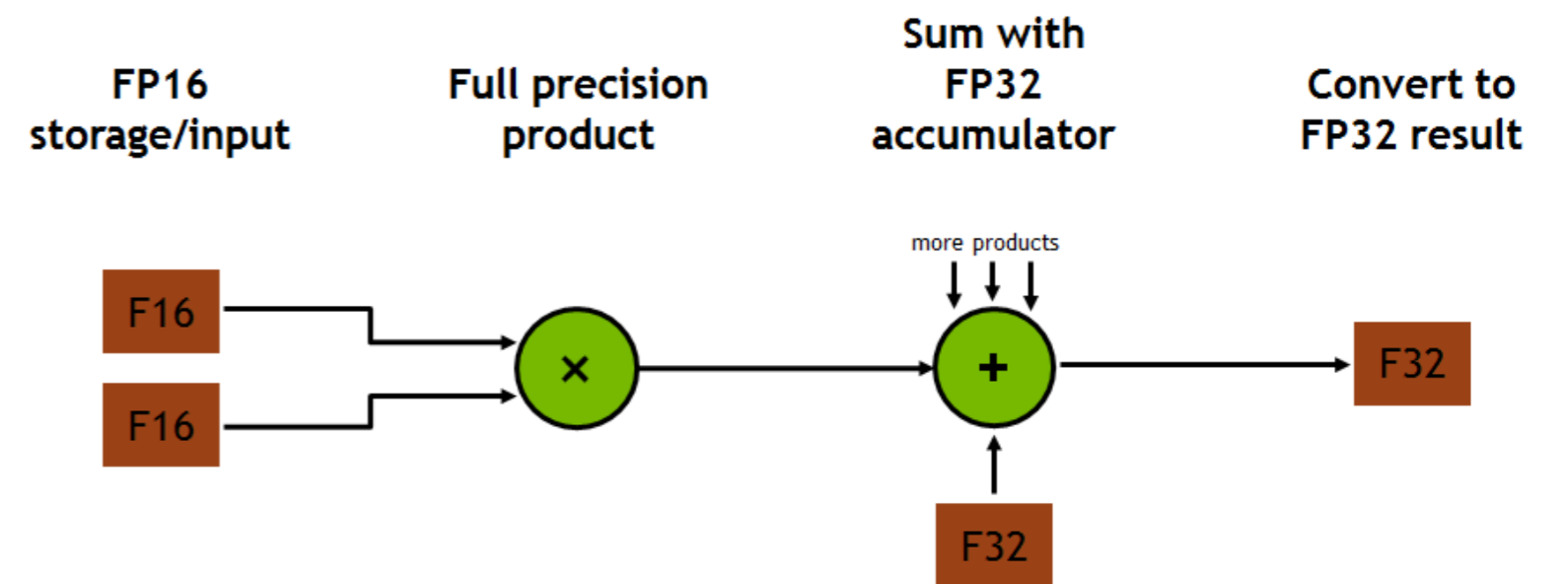
# The Basic Operation of Tensor Cores

- Matrix-Multiply-and-Accumulate (MMA):  $D = A \cdot B + C$

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

where  $C$  and  $D$  could be the same register,  
 $A$  is  $M \times K$ ,  $B$  is  $K \times N$ ,  $C$  and  $D$  are  $M \times N$  matrices

- Usually (often):
  - $A, B$  are  $4 \times 4$  of type FP16 (`__half`)
  - $C, D$  are  $4 \times 4$  of FP32 (`float`)
- One MMA = 64 FLOPs in 1 cycle!



- All CUDA libraries use them (cuBLAS, CUB, CUTLASS, cuDNN, ...)
- You can use them in your own kernels, iff all threads within a warp collaborate, i.e., *execute the same* MMA instructions
- Idea:
  - Each warp computes an MMA for bigger matrices
  - All warps together compute big matrix multiplication in tiled fashion
- Example tiling:
  - You kernel partitions the big matrix into  $16 \times 16$  tiles
  - Each warp works on one  $16 \times 16$  tile
  - Distribution of one tile into  $4 \times 4$  tensor core operations is done by GPU scheduler

# Minimal Example: 16x16 Matrix Multiplication

```

#include <mma.h>
using namespace nvcuda::wmma;

__global__ void wmma_example( __half* a, __half* b, float* c )
{
    // Declare the fragments
    fragment<matrix_a, 16, 16, 16, half, col_major> frags_of_a;
    fragment<matrix_b, 16, 16, 16, half, col_major> frags_of_b;
    fragment<accumulator, 16, 16, 16, float> frags_of_acc;

    fill_fragment( frags_of_acc, 0.0f );

    // Load the inputs
    load_matrix_sync( frags_of_a, a, 16 );
    load_matrix_sync( frags_of_b, b, 16 );

    // Perform the matrix multiplication
    mma_sync( acc_frag, frags_of_a, frags_of_b, frags_of_acc );

    // Store the output
    store_matrix_sync( c, frags_of_acc, 16, mem_col_major );
}

```

All data types and functions are provided by `mma.h`

← A warp will work on 16x16 matrices, each thread in the warp will work on a "fragment" of those matrices

← Clear the accumulator

← All threads load "their" fragments of matrix a/b, resp., into the registers ("sync" says they work in sync)

← Here, the actual multiplication happens, using all the tensor cores of the SM in collaboration

# Declarations of Some of the Functions/Types in `mma.h` (Just FYI)

```
template< typename Use, int m, int n, int k,  
          typename T, typename Layout=void > class fragment;  
  
void load_matrix_sync( fragment<...> &a,  
                      const T* mptr, unsigned ldm );  
  
void store_matrix_sync( T* mptr, const fragment<...> &a,  
                      unsigned ldm, layout_t layout );  
  
void fill_fragment( fragment<...> &a, const T& v );  
  
void mma_sync( fragment<...> &d, const fragment<...> &a,  
              const fragment<...> &b, const fragment<...> &c );
```

All threads together will declare their fragments, which together will form a tile/block of the matrix

Waits until all threads in a warp are at this load instruction, then loads the tile/block from memory

Same as `load_matrix`

Performs warp-synchronous matrix multiply-accumulate

# High-Level Procedure for Matrix-Matrix Multiplication Using Tensor Cores

each block of threads works on one tile of the output  $P$

each warp loads a  $16 \times 16$  tile of  $A$  and  $B$  into shared memory:

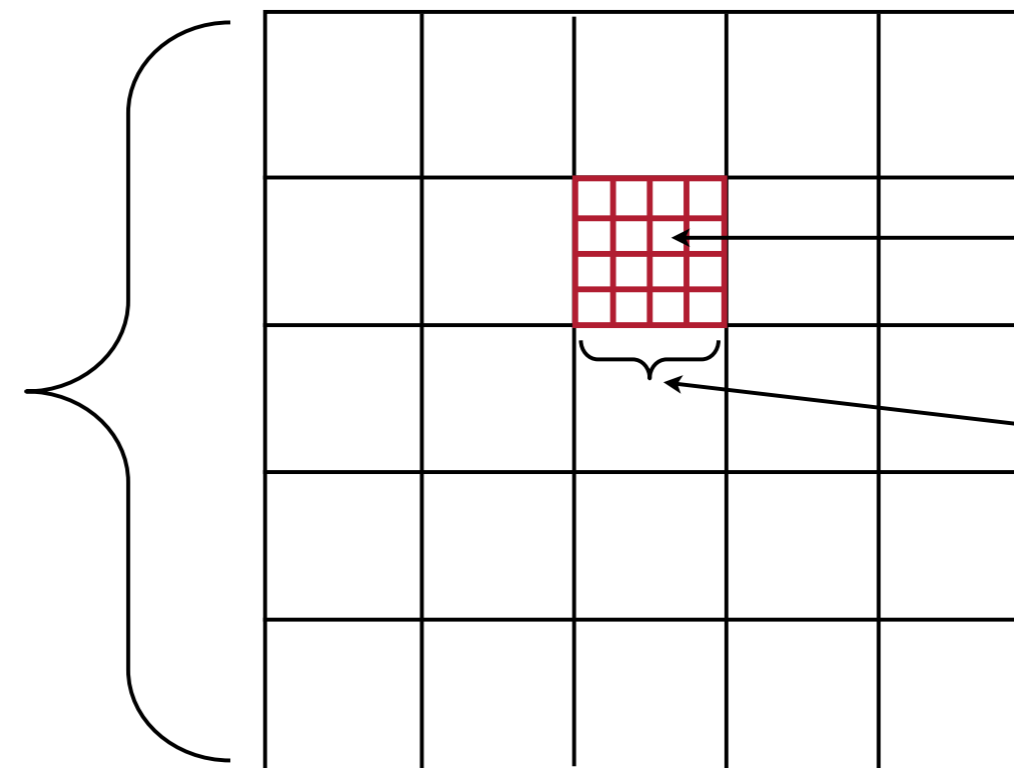
$A, B$  are usually stored in row or column major, so threads need to do some offset calculations and re-arrangements

each warp multiplies the tiles and accumulates results

(the GPU partitions the work into  $4 \times 4$  matrix multiplications automagically)

each warp stores the result in  $P$

Partitioning of the big matrices into tiles (e.g., tiles of size  $16 \times 16$ ) that you must do yourself



$4 \times 4$  matrices

Partitioning of a tile into fragments that is done by CUDA's MMA types, e.g., `fragment<>`, and MMA functions, e.g., `load_matrix_sync()`



## Matrix-matrix multiplication (GEMM)

cuBLAS Mixed Precision (FP16 Input, FP32 compute)

