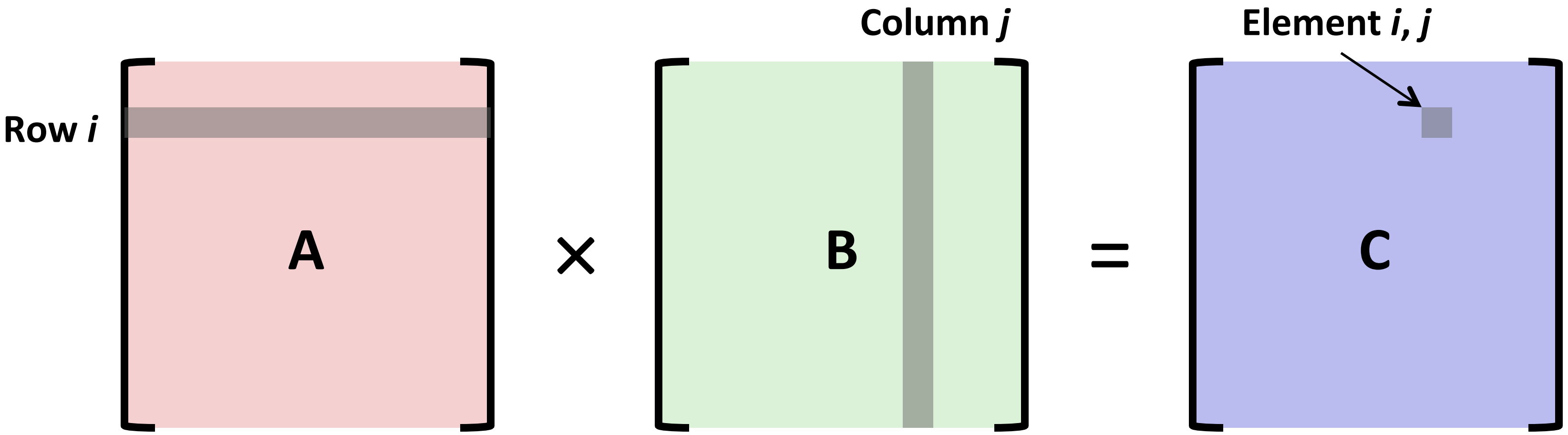
Introduction to CUDA Programming

15-418/618: Parallel Computer Architecture and Programming
Special Presentation. Feb. 8, 2017

Randy Bryant

Application Example: N × N Matrix Multiplication

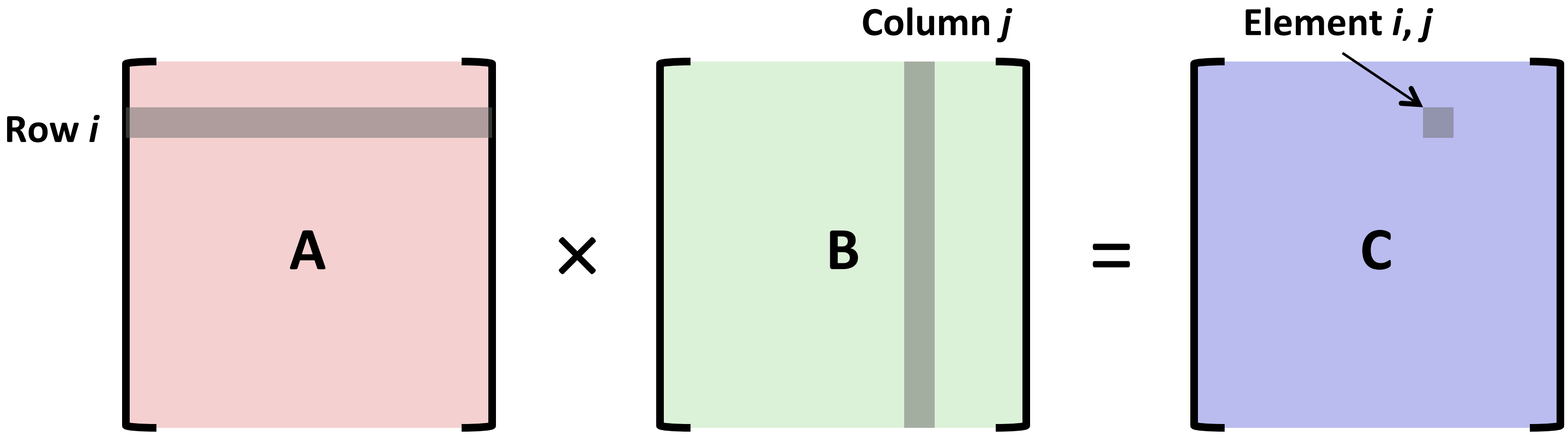


$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b_{k,j}$$

- **Complexity**
  - N^3 multiplications
  - N^3 additions
- **Assume row-major access**

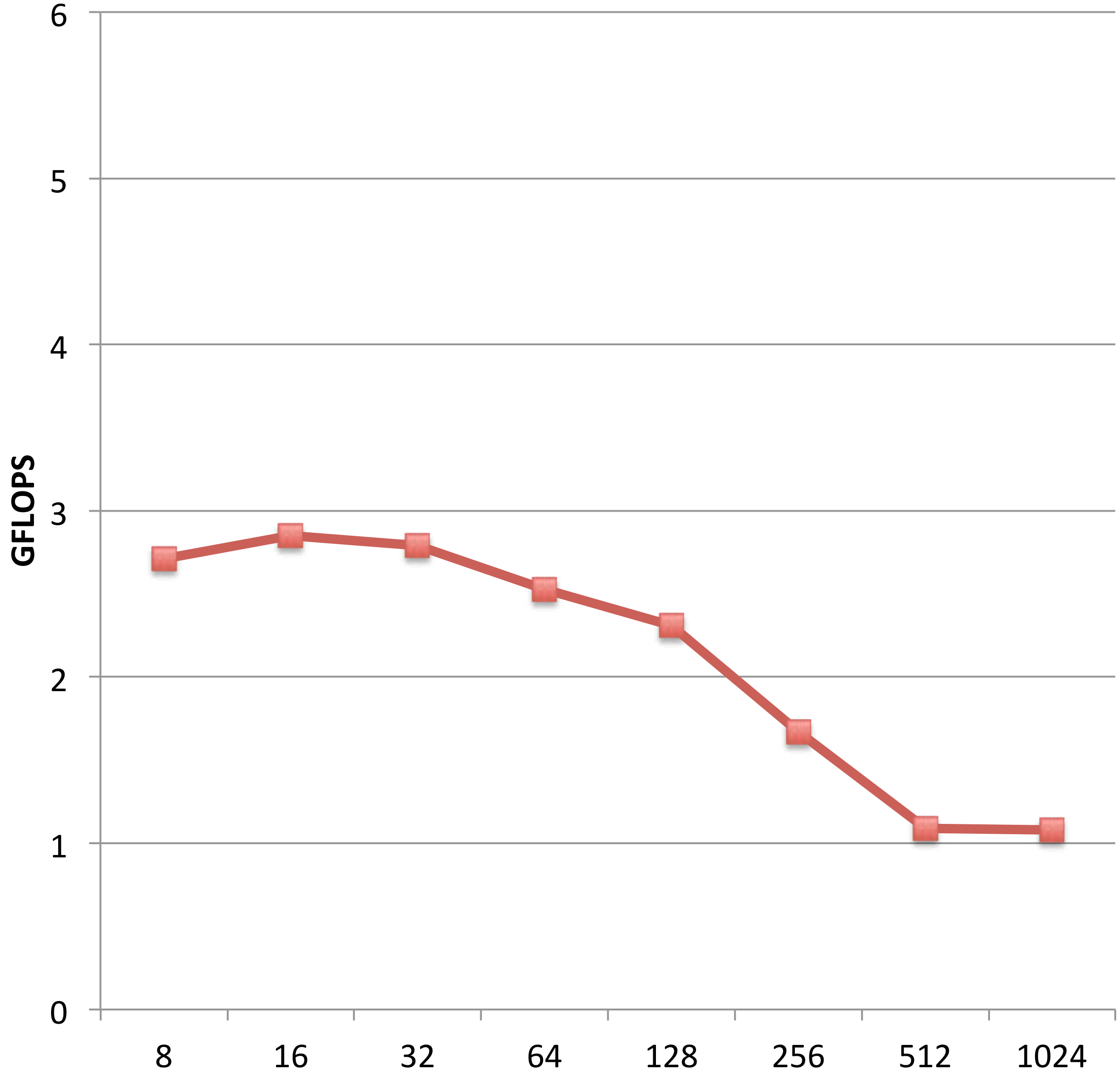
```
#define RM(r, c, width) ((r) * (width) + (c))
```

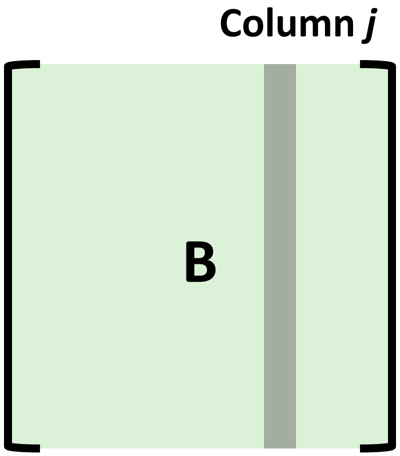
Matrix Multiplication: Simple CPU Implementation



```
void multMatrixSimple(int N, float *matA, float *matB, float *matC) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * matB[RM(k,j,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```

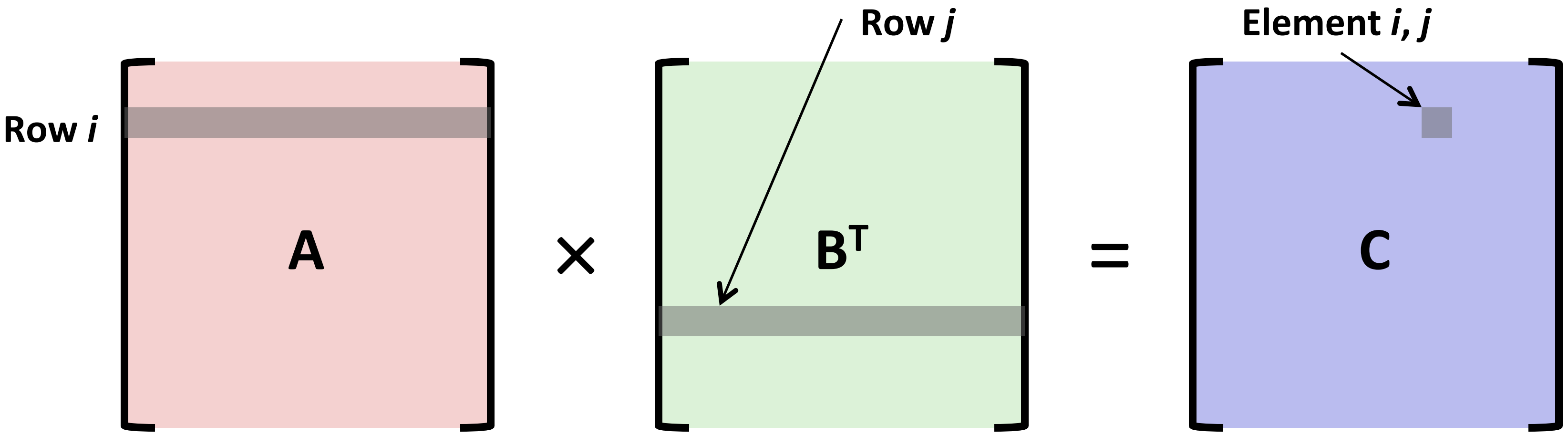
CPU Simple Performance





- Measured in GFLOPS
- Drops off for large values of N
- B has bad access pattern

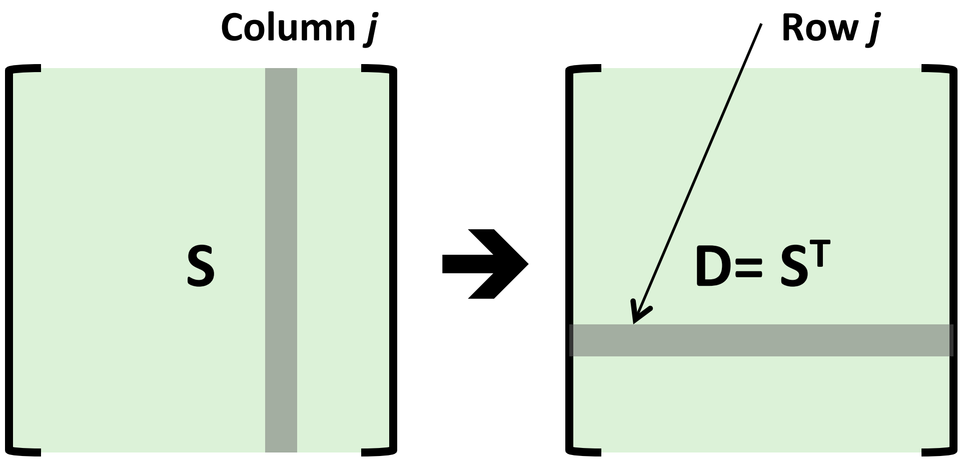
Optimization #1: Pretranspose



- **Transposed version of B has better access pattern**
  - Transpose once
  - Use each element N times

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b_{j,k}^{T}$$

Transposing a Matrix



- Column-major ordering of elements

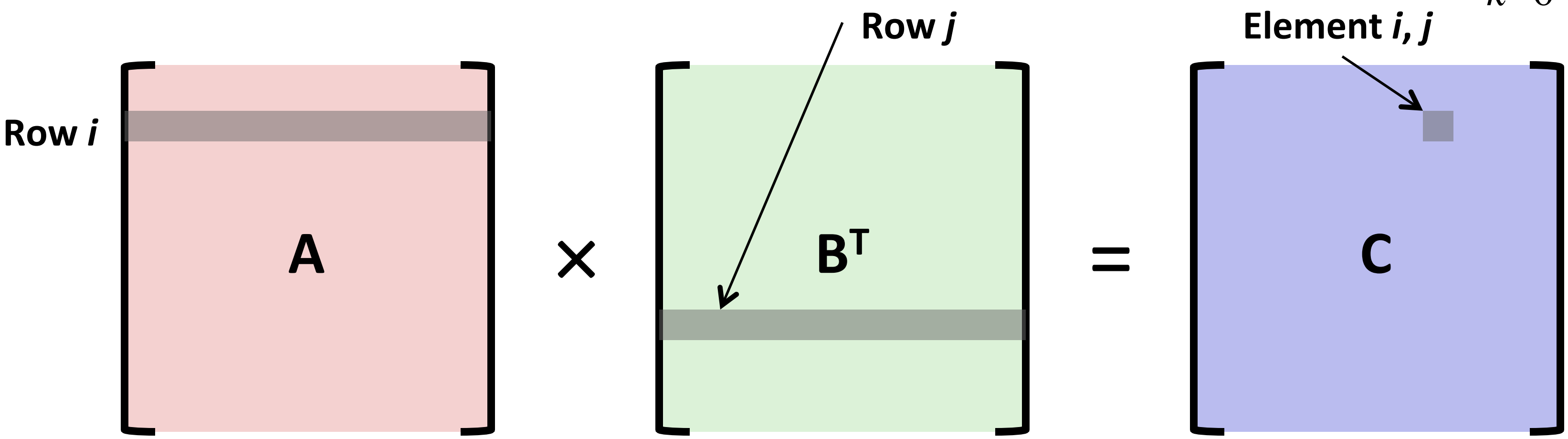
```
#define CM(r, c, height) ((c) * (height) + (r))
```

- Transposing converts from row-major to column-major order

```
void transposeMatrix(int N, float *matS, float *matD) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            matD[CM(i,j,N)] = matS[RM(i,j,N)];
}
```

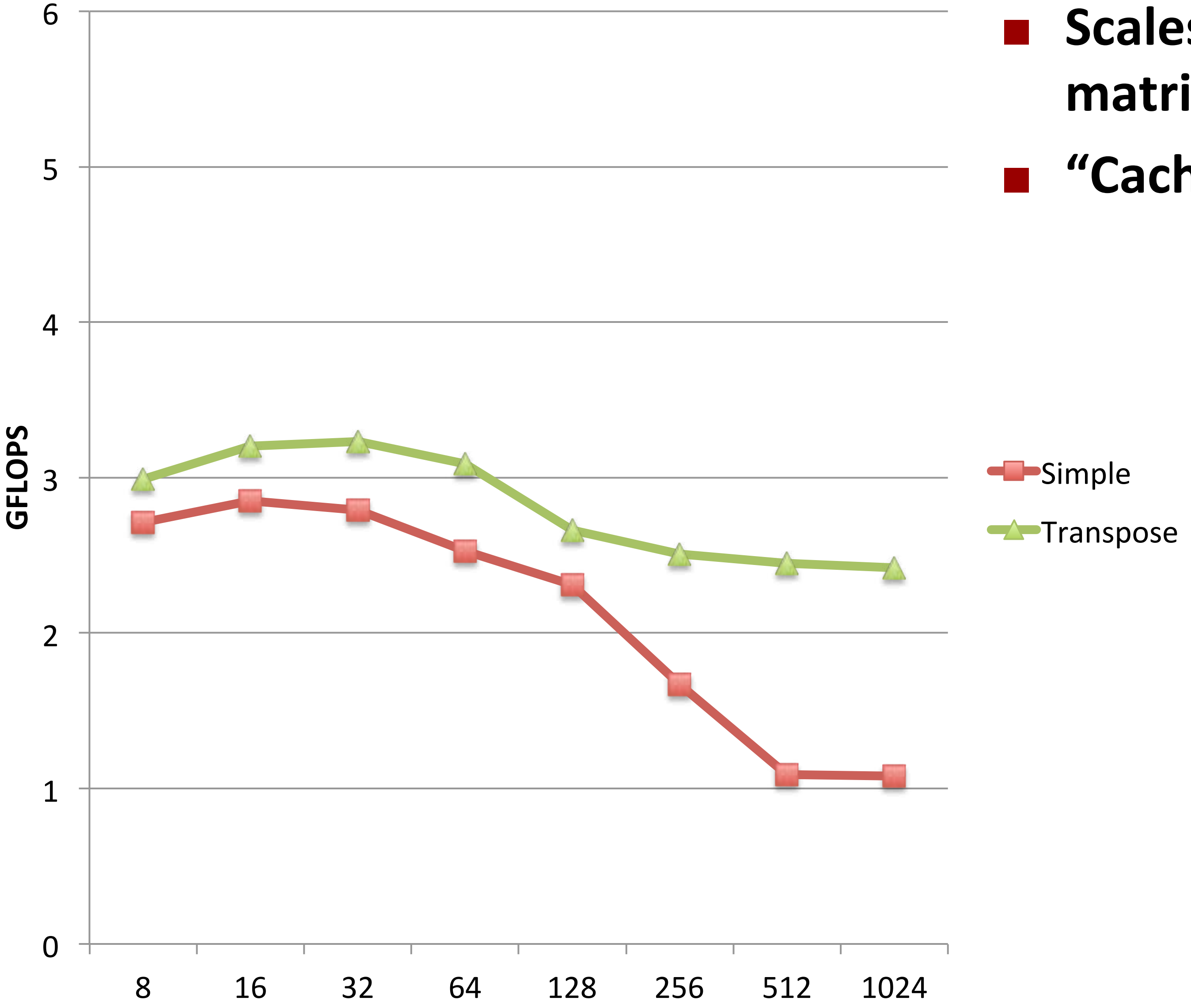
Matrix Multiplication: Pretranspose Implementation

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b^T_{j,k}$$



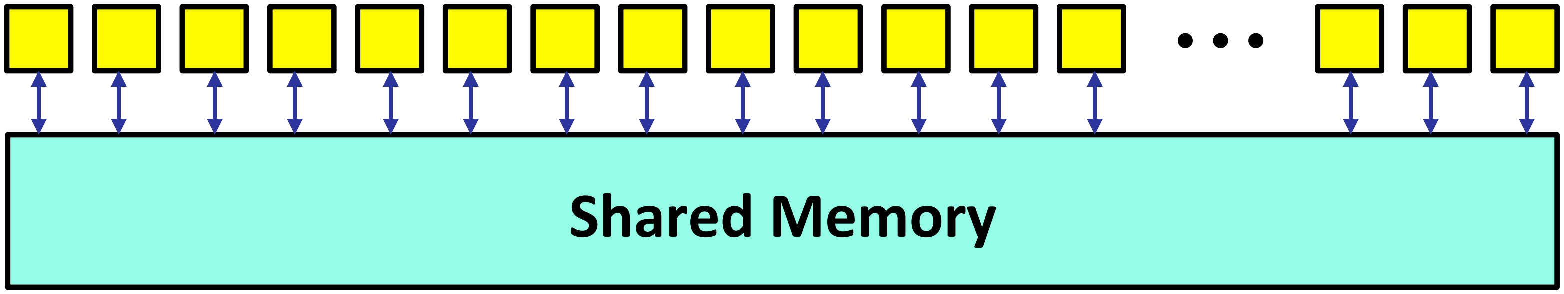
```
void multMatrixTransposed(int N, float *matA, float *matB, float *matC)
{
    float *tranB = scratchMatrix(N);
    transposeMatrix(N, matB, tranB);
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * tranB[RM(j,k,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```

Pretranspose Performance



- Scales to large matrices
- "Cache-friendly" code

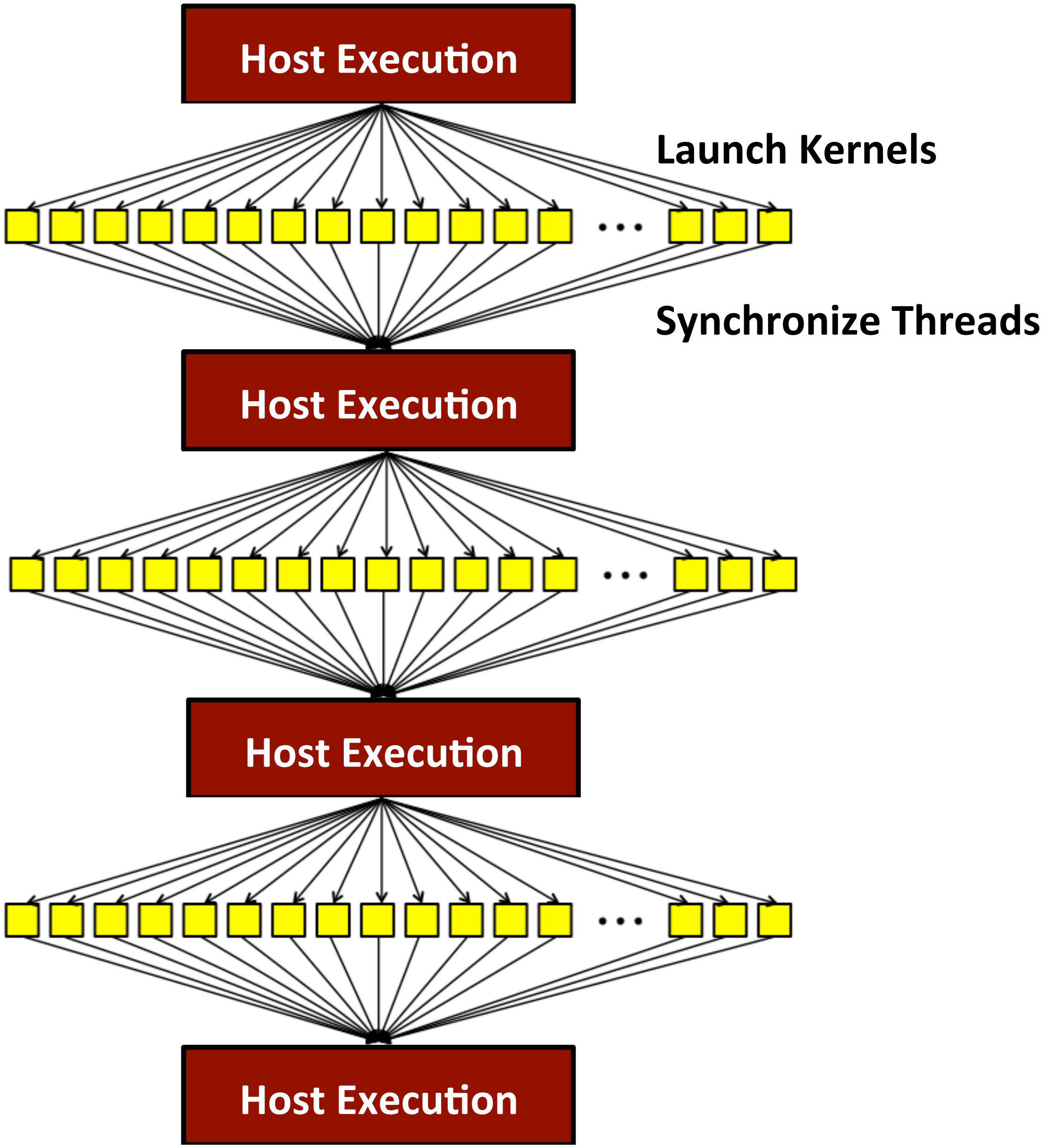
Abstract Single Program Multiple Data (SPMD) Model



- **M Processors, all executing same code**
  - Called “kernels”
  - M based on problem size
- **Share common global memory**
  - And also have private memory for local variables
  - Make no assumptions about effect of memory access conflicts
- **No synchronization primitives**
- **Called *threads*, but not at all like pthreads**
  - Very simple & lightweight
  - All execute the same program

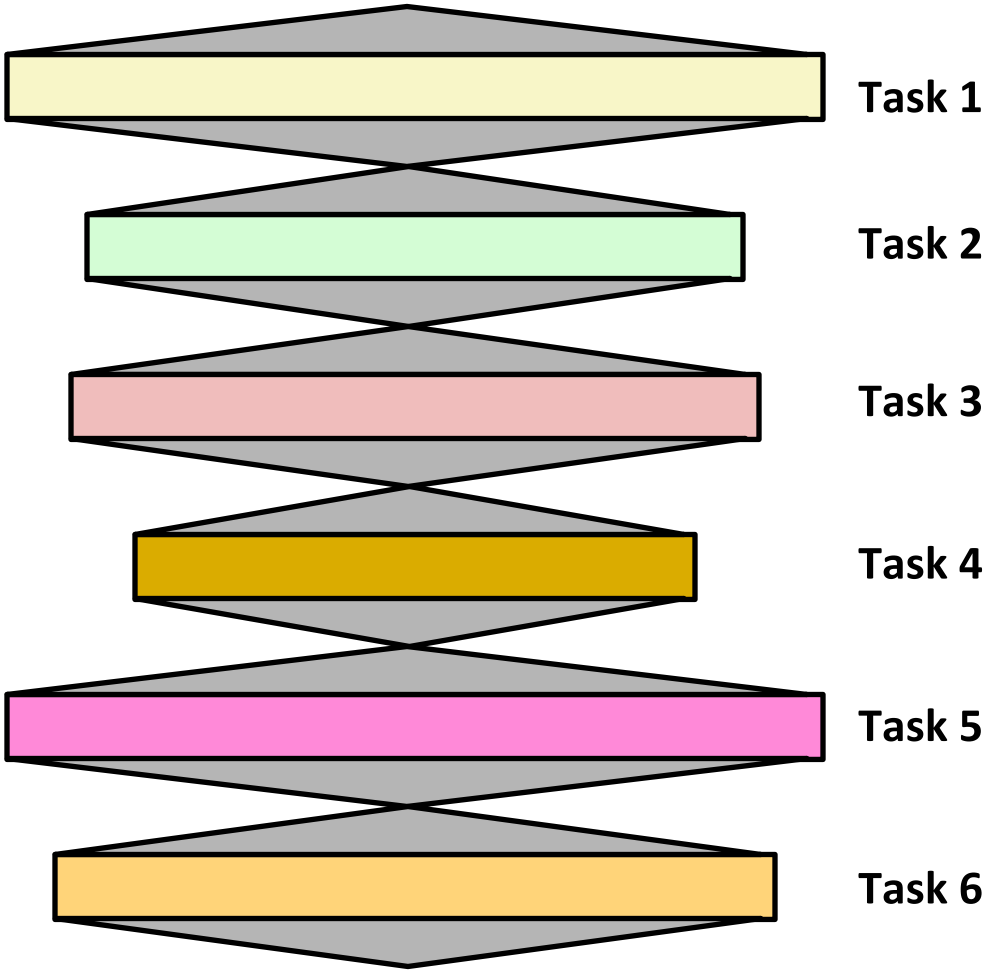
Interacting with SPMD Machine: Control

- Overall execution managed by code executing on host
- Launch set of kernels
  - Number & kernel function can vary with each launch
- Wait until all completed
  - Explicit or implicit synchronization
- Repeat as necessary

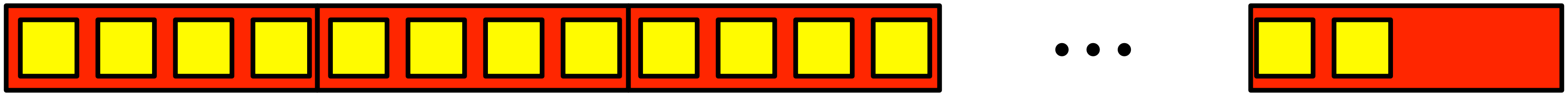


Structure of SPMD Program

- **Concept**
  - Partition computation into sequence of tasks
  - Perform each task over all data with single operation
- **Performance Limitations**
  - Synchronization requires waiting for slowest task
  - No locality of data
  - No locality of synchronization



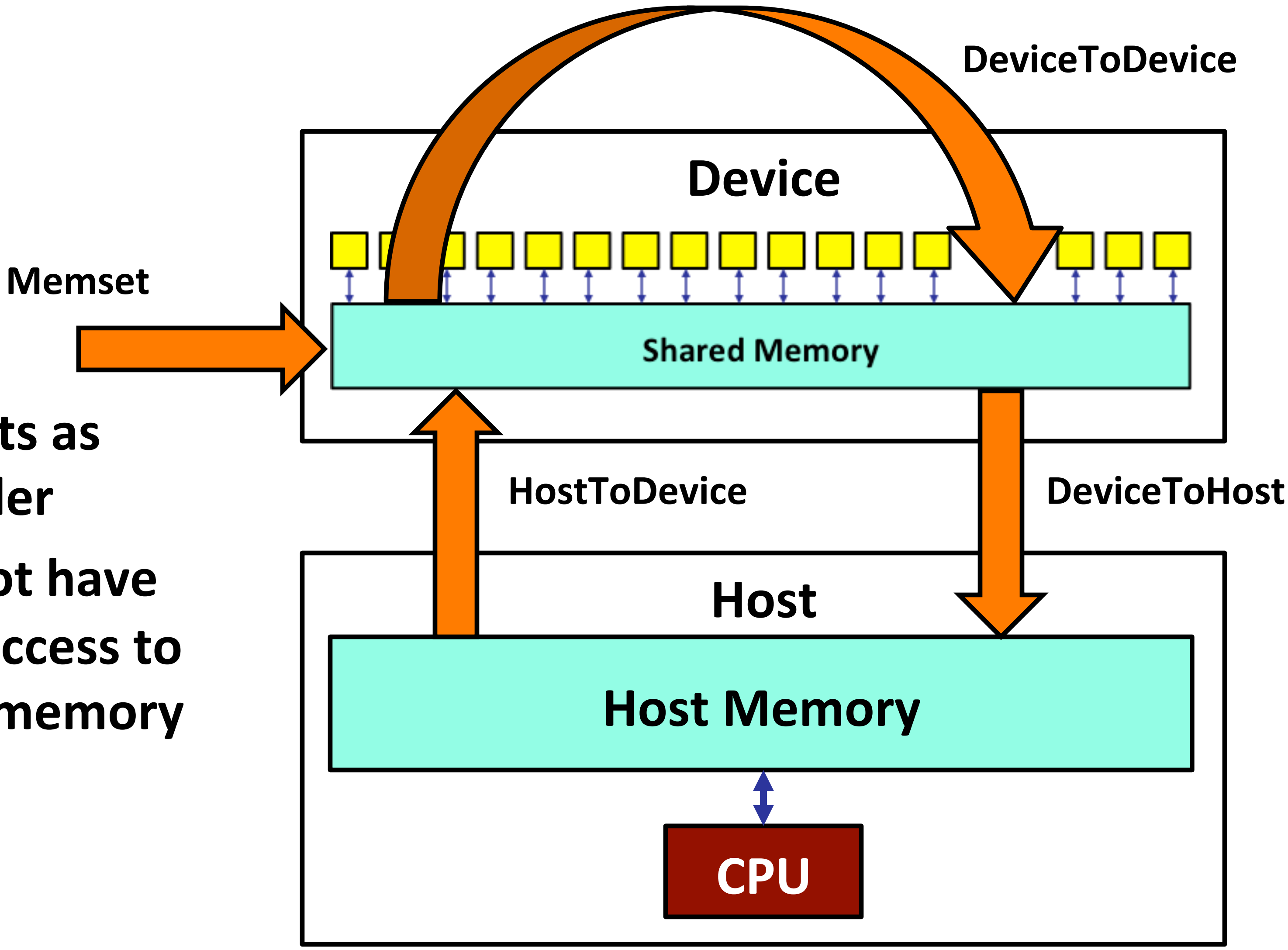
Block/Thread Notation



- **Idea (One-dimensional version)**
  - Executing threads grouped into *blocks*
    - Each contains same number of threads
      - – Host program specifies block size (blockDim.x)
  - Host program makes sure there are enough blocks to generate N threads
  - Some threads in last block should not get used

```
__global__ void
inplaceReduceKernel(int length, int nlength, float *data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < nlength) {
        . . .
    }
}
```

Interacting with SPMD Machine: Data



- Host acts as controller
- Does not have direct access to device memory

Cuda Program

- **Cuda file (.cu) contains mix of device code & host code**
  - It’s up to you to understand which is which!
- **Device Code**
  - Kernels (__global__)
    - Code for threads
    - Must only reference device memory
  - Device functions (__device__)
    - Called by kernels
    - Only reference device memory
    - Do not generate new threads

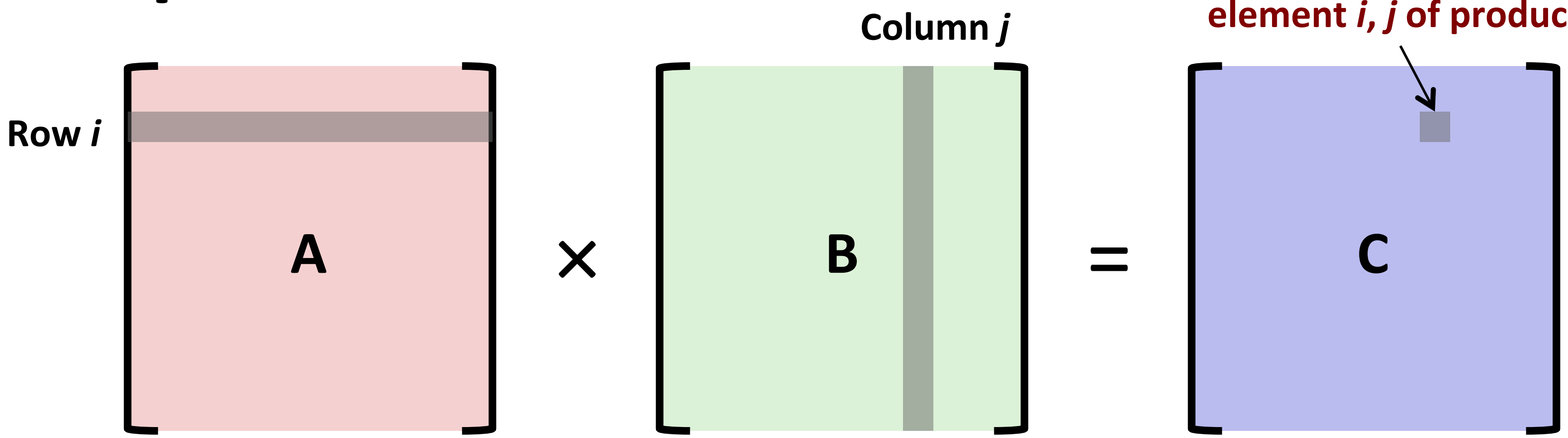
```
__device__ void
deviceMult(float x, float y,
           float *dest)
{
    *dest = x * y;
}
```

```
__global__ void
inplaceReduceKernel(int length, int nlength, float *data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < nlength) {
        . . .
    }
}
```

Cuda Program (cont.)

- **Cuda file (.cu) contains mix of device code & host code**
  - It's up to you to understand which is which!
- **Host Code**
  - Conventional C/C++
  - Can only reference host memory
    - But, can have pointers to device memory
  - Manages the launching of threads
  - Manages movement of data between host & device memories

Matrix Multiplication: Simple Cuda Implementation



Each thread computes element *i*, *j* of product

```
__global__ void
cudaSimpleKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= N || j >= N)
        return;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i,k,N)] * dmatB[RM(k,j,N)];
    }
    dmatC[RM(i,j,N)] = sum;
}
```

Host Code Example

LBLK = 32

32 X 32 = 1024 threads / block

- **Launch kernels to perform vector product**

```
void cudaMultMatrixSimple(int N, float *dmatA, float *dmatB,
                          float *dmatC)
{
  dim3 threadsPerBlock(LBLK, LBLK);
  dim3 blocks(updiv(N, LBLK), updiv(N, LBLK));
  cudaSimpleKernel<<<blocks, threadsPerBlock>>>(N, dmatA, dmatB, dmatC);
}
```

- **Useful stuff**
  - Compute $\lceil n / d \rceil$

```
// Integer division, rounding up
static inline int updiv(int n, int d) {
    return (n+d-1)/d;
}
```

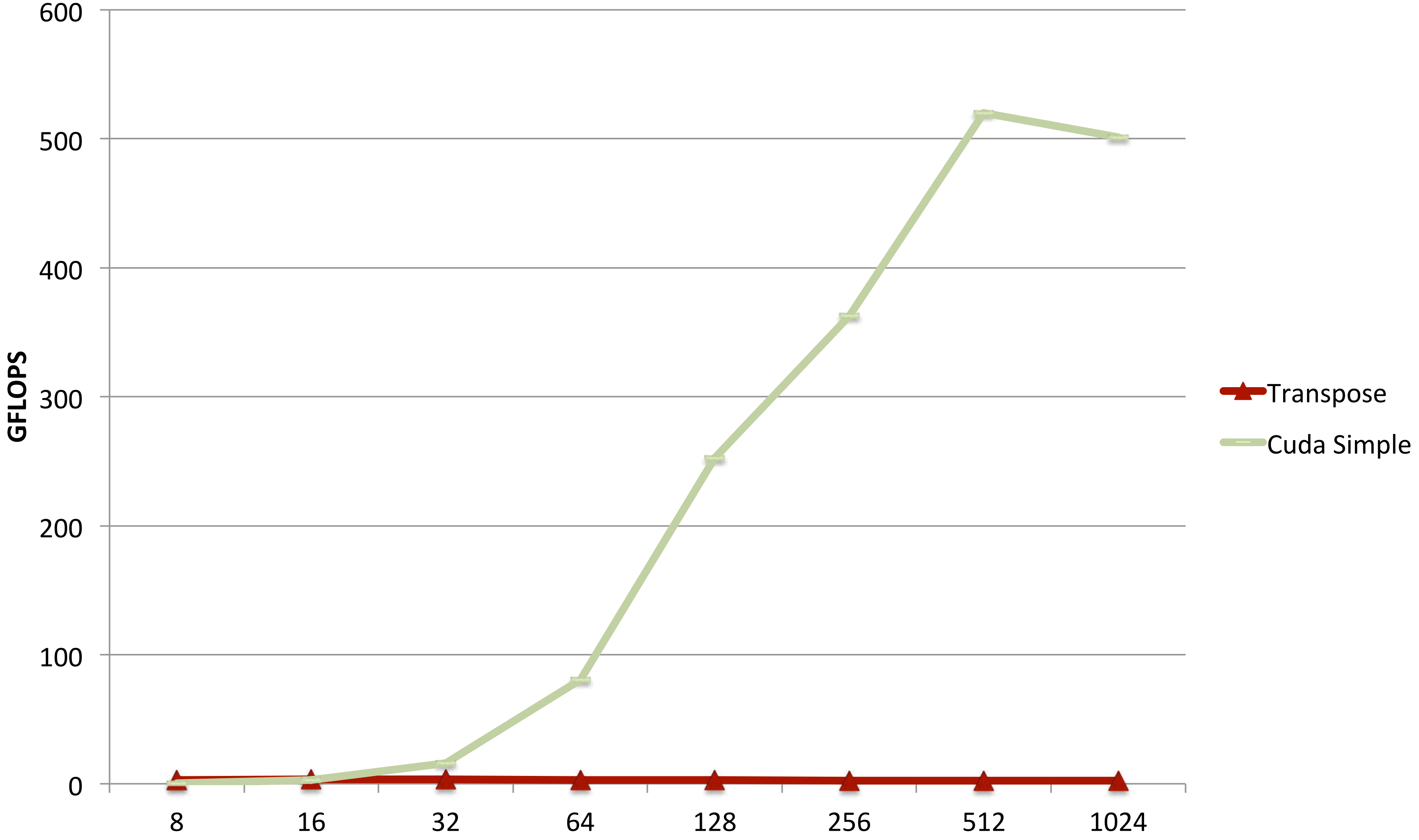
  - Setting number of threads per block:
    - Should be multiple of 32
    - Max value = 1024

Host Code Example (cont).

```
void cudaMultiply(int N, float *aData, float *bData, float *cData) {
    float *aDevData, *bDevData, *cDevData
    cudaMalloc((void **) &aDevData, N*N * sizeof(float));
    cudaMalloc((void **) &bDevData, N*N * sizeof(float));
    cudaMalloc((void **) &cDevData, N*N * sizeof(float));
    cudaMemcpy(aDevData, aData, N*N * sizeof(float),
               cudaMemcpyHostToDevice);
    cudaMemcpy(bDevData, bData, N*N * sizeof(float),
               cudaMemcpyHostToDevice);

    cudaMultMatrixSimple(N, aDevData, bDevData, tDevData);

    cudaMemcpy(cData, cDevData, N*N * sizeof(float),
               cudaMemcpyDeviceToHost);
    cudaFree(aDevData); cudaFree(bDevData); cudaFree(cDevData);
}
```

Observe: **Host can hold pointers to device memory, but cannot read or write device memory locations**

Simple CUDA Performance



Inverted Indexing Accessing Pattern

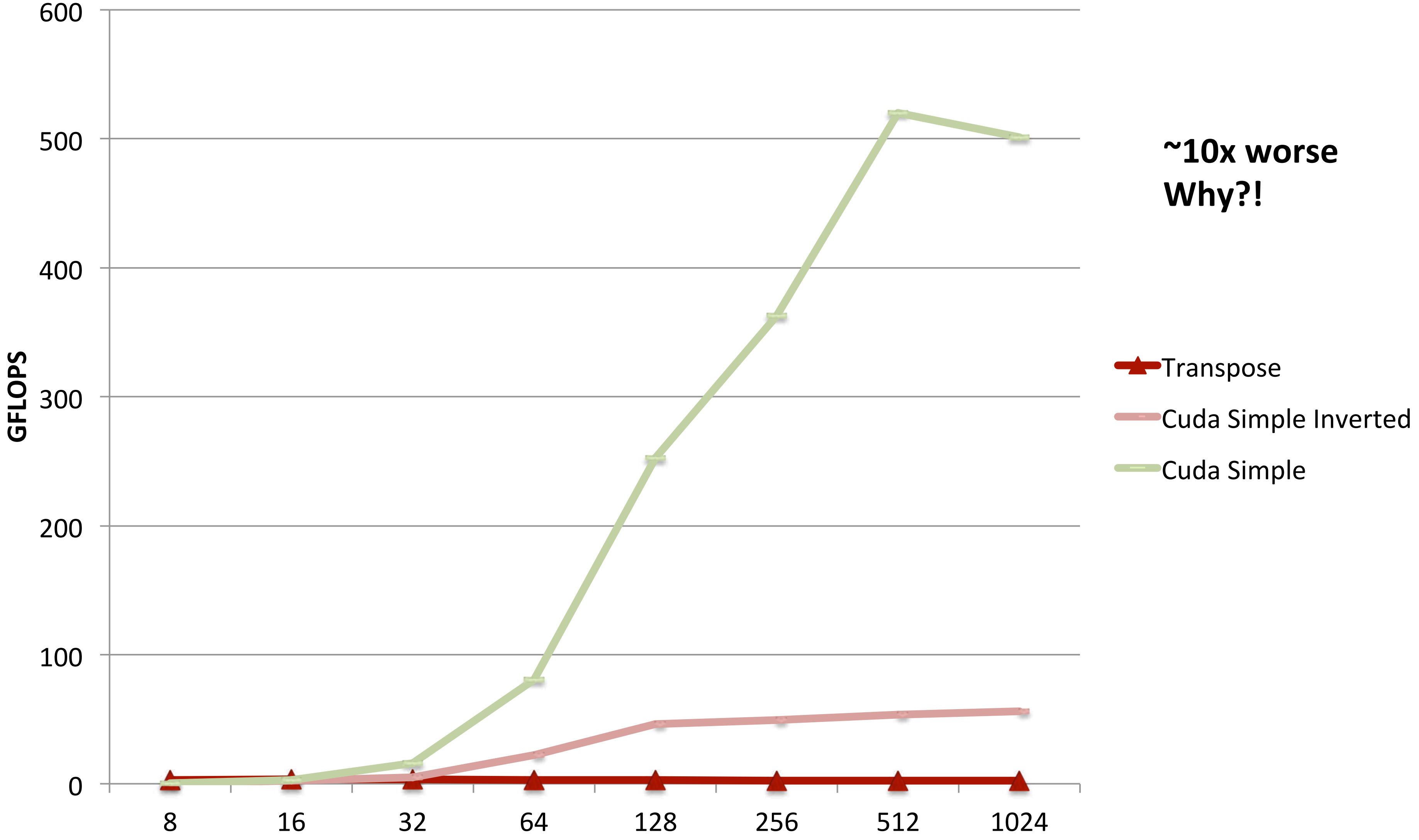
Regular

```
__global__ void
cudaSimpleKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  . . .
}
```

Inverted

```
__global__ void
cudaSimpleKernelOld(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  . . .
}
```

CUDA Inverted Indexing Performance



~10x worse
Why?!

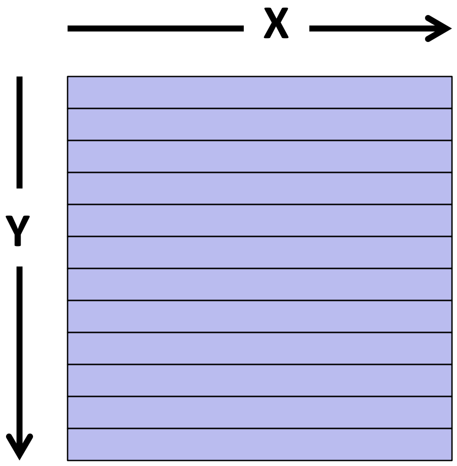
What’s the Difference?

Regular

```
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
```

Inverted

```
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y
```



- **CUDA threads numbered within block in row-major order**
  - X = column number, Y = row number
- **Threads with same value of Y map to single warp.**
- **Threads with same value of Y and consecutive values of X map to consecutive positions in single warp**
- **When single warp accesses consecutive memory locations, do block read or write**
- **When single warp accesses separated memory locations, requires gather (read) or scatter(write)**

Impact on Memory Referencing: Regular

```
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

Read A	`= dmatA[RM(i,k,N)];`	**Threads in warp reference single location**
Read B	`= dmatB[RM(k,j,N)];`	**Threads in warp do block read**
Write B	`dmatC[RM(i,j,N)] =`	**Threads in block do block write**

- **Threads within warp have:**
  - same value of **`k`**
  - same value of **`i`**
  - consecutive values of **`j`**
- **Warp reads & writes match memory organization**

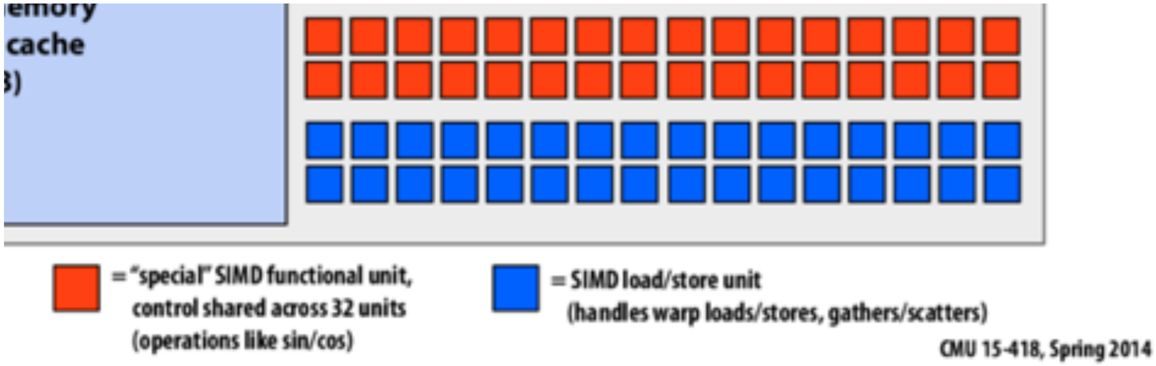
Impact on Memory Referencing: Inverted

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Read A	`= dmatA[RM(i,k,N)];`	Threads in warp do gather
Read B	`= dmatB[RM(k,j,N)];`	Threads in warp reference single location
Write B	`dmatC[RM(i,j,N)] =`	Threads in block do scatter

- **Threads within warp have:**
  - same value of **`k`**
  - consecutive values of **`i`**
  - same value of **`j`**
- **Warp reads/writes does not match memory organization**

Relation to Hardware


emory
cache
3)
🟥 = "special" SIMD functional unit, control shared across 32 units (operations like sin/cos)
🟦 = SIMD load/store unit (handles warp loads/stores, gathers/scatters)

CMU 15-418, Spring 2014

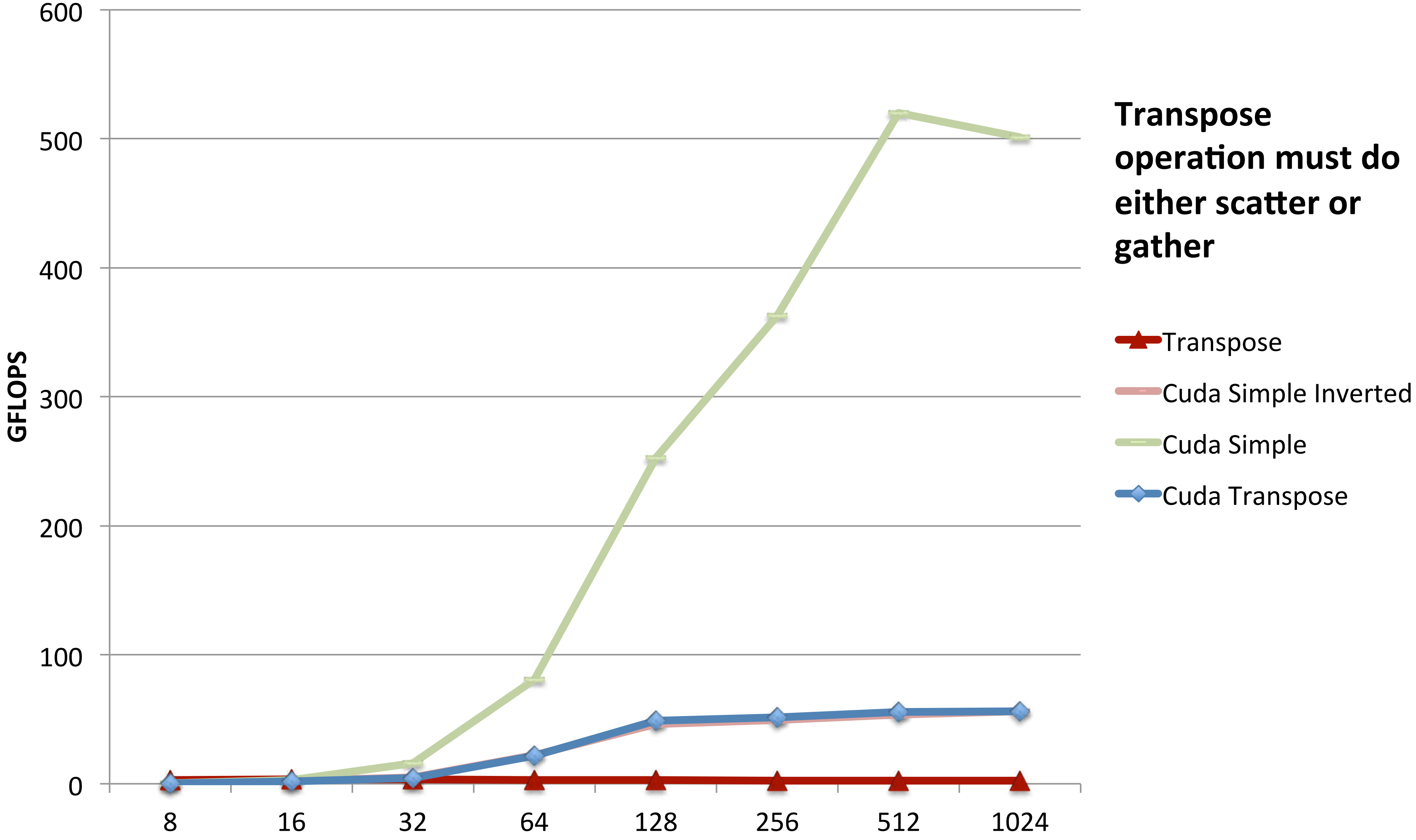
- **Optimizing memory instruction peformance**
  - Load faster than gather
  - Store faster than scatter
- **Avoiding memory conflicts**
  - Inverted code has multiple warps competing for same block of memory

Pretransposing with CUDA

```
/* Transpose matrix */
__global__ void
cudaTransposeKernel(int N, const float  *dmatS, float *dmatD) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N || j >= N)
        return;
    dmatD[CM(i,j,N)] = dmatS[RM(i,j,N)];
}
```

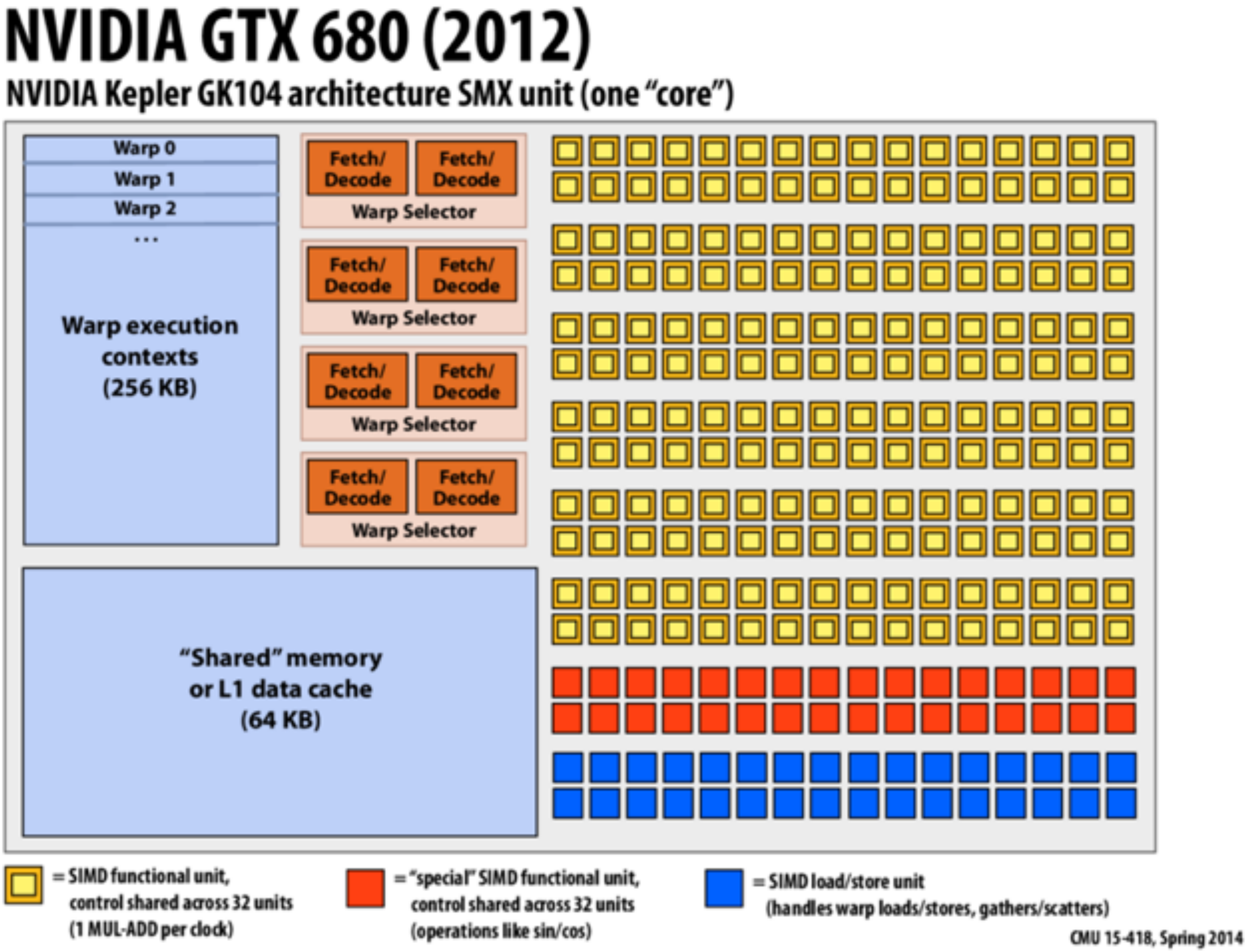
```
__global__ void
cudaTransposedKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= N || j >= N)
        return;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i,k,N)] * dmatB[RM(j,k,N)];
    }
    dmatC[RM(i,j,N)] = sum;
}
```

CUDA Pretranspose Implementations



Transpose operation must do either scatter or gather

Thinking About Cuda



NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture SMX unit (one "core")

Warp 0

Warp 1

Warp 2

...

Warp execution contexts (256 KB)

Fetch/ Decode

Fetch/ Decode

Warp Selector

"Shared" memory or L1 data cache (64 KB)

🟨 = SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

🟥 = "special" SIMD functional unit, control shared across 32 units (operations like sin/cos)

🟦 = SIMD load/store unit (handles warp loads/stores, gathers/scatters)

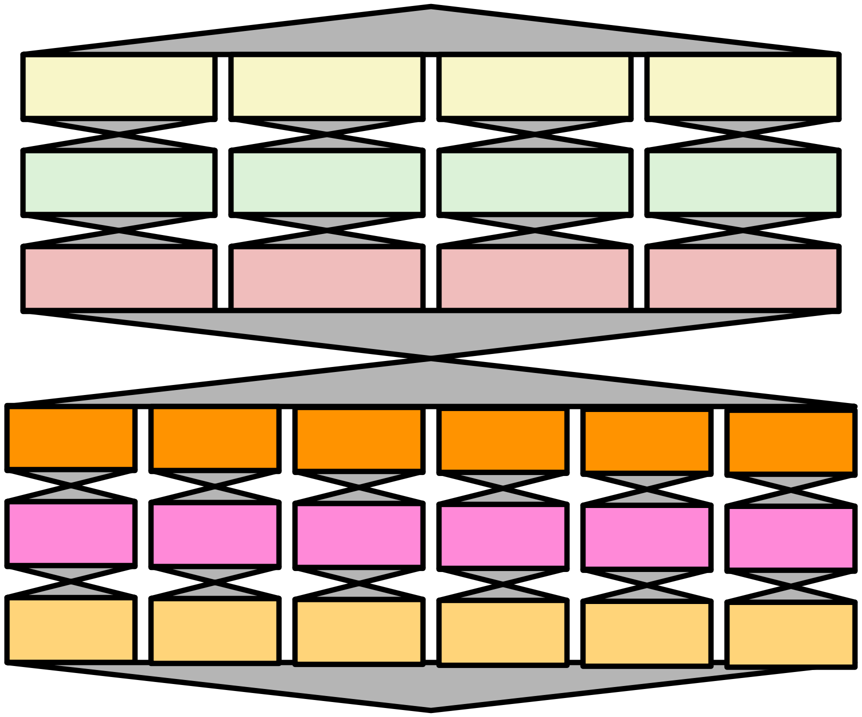
CMU 15-418, Spring 2014

GPU Hierarchy

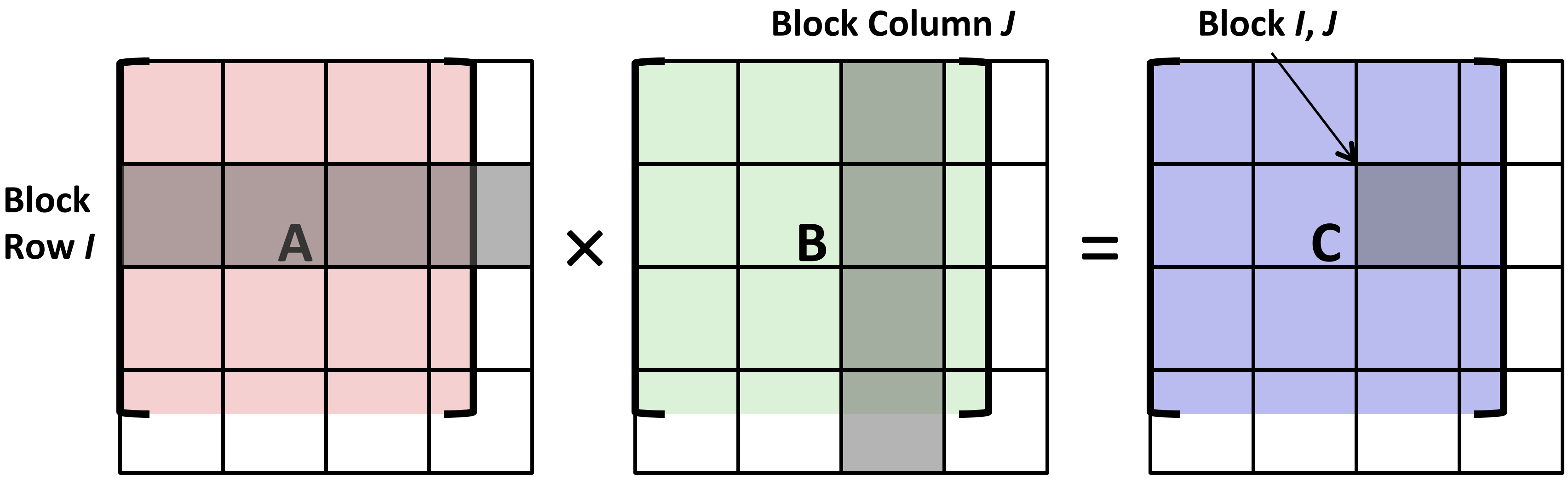
- **Block Level**
  - Programmer partitions problem into blocks of K threads each
    - 32 ≤ K ≤ 1024
    - Multiple of 32
  - Within block, have access to fast shared memory
  - Within block, can synchronize with `__syncthreads()`
- **Warp Level**
  - Each block implemented as set of warps
    - 32 threads each
  - Implemented using "SIMT" processor
    - Single-instruction, multiple threads
    - Guarantees stay synchronized

Programming with Blocks

- Localize computation within blocks
- Each performs sequence of tasks
- Each uses shared memory and local synchronization



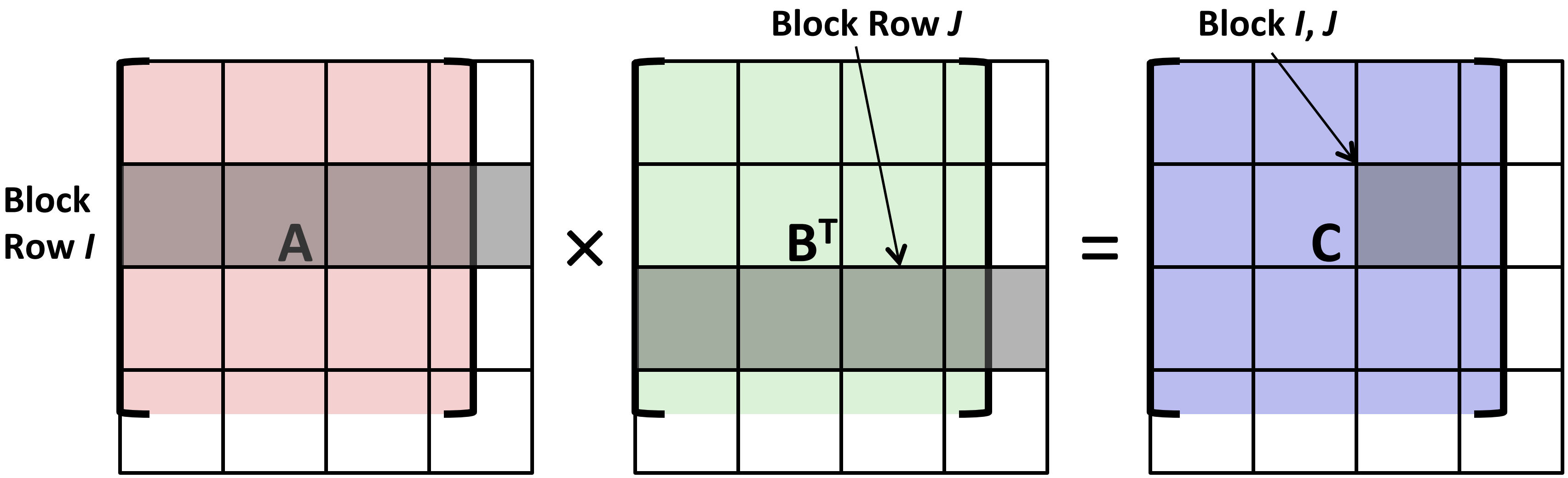
MM Optimization #2: Partitioning into Blocks



$$C_{I,J} = \sum_{K=0}^{N_b-1} A_{I,K} \cdot B_{K,J}$$

- Generate results on block-by-block basis
- Localizes access to A and B
- N need not be multiple of block size

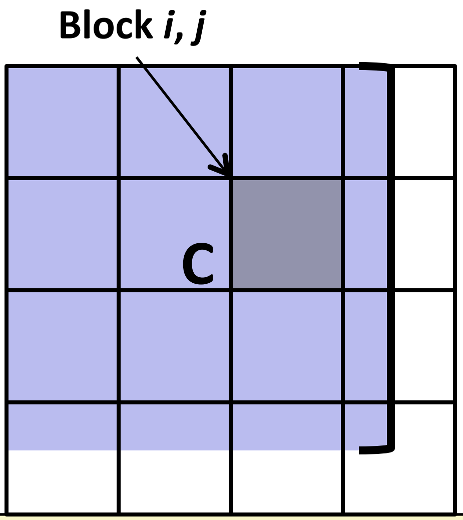
CPU-based Blocked Implementation



$$C_{I,J} = \sum_{K=0}^{N_b-1} A_{I,K} \cdot B_{K,J}$$

- **Use pretranspose**
  - Required for performance
- **Structure**
  - Outer loops index over blocks
  - Inner loops compute product for single block
- **Block size `SBLK` = 8**

Blocked Multiplication Implementation: Outer Loops



$$C_{I,J} = \sum_{K=0}^{N_b-1} A_{I,K} \cdot B_{K,J}$$

- Look at actual code to see how it handles cases where N is not multiple of block size

```
void multMatrixTransposeBlocked(int N,
          float *matA, float *matB, float *matC) {
    float *tranB = scratchMatrix(N);
    transposeMatrix(N, matB, tranB);
    /* Zero out C */
    memset(matC, 0, N * N * sizeof(float));
    int i, j, k;
    for (i = 0; i <= N-SBLK; i+= SBLK)
        for (j = 0; j <= N-SBLK; j+= SBLK)
            for (k = 0; k <= N-SBLK; k+=SBLK)
                Compute contribution to C[i..i+SBLK-1][j..j+SBLK-1]
}
```

Blocked Multiplication Implementation: Inner Loops

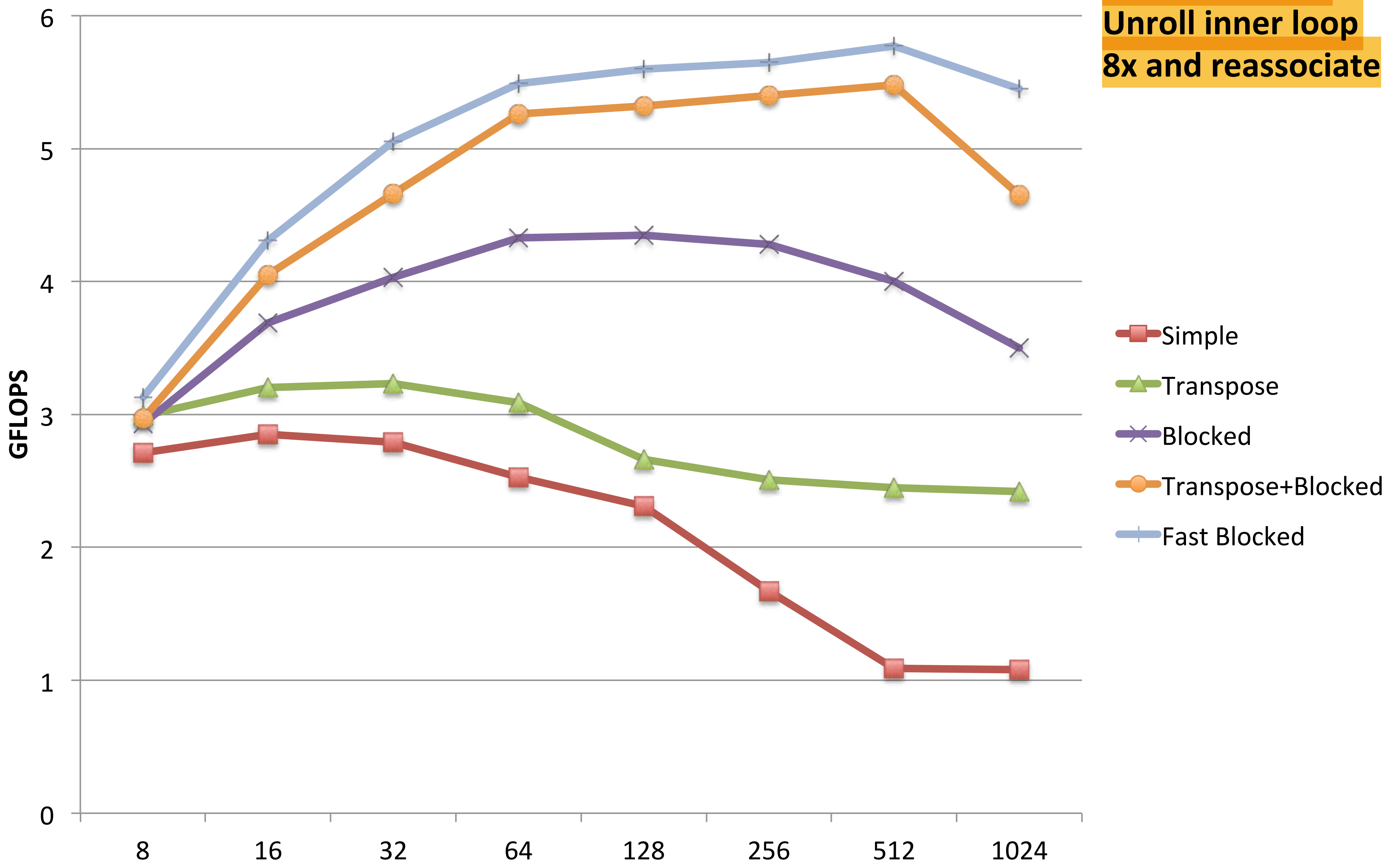
$$c_{i+bi,j+bj} = \sum_{bk=0}^{b-1} a_{i+bi,k+bk} \cdot b^T_{j+bj,k+bk}$$

- `i, j, k` provide starting indices of blocks
- `bi, bj, bk` provide offsets within blocks

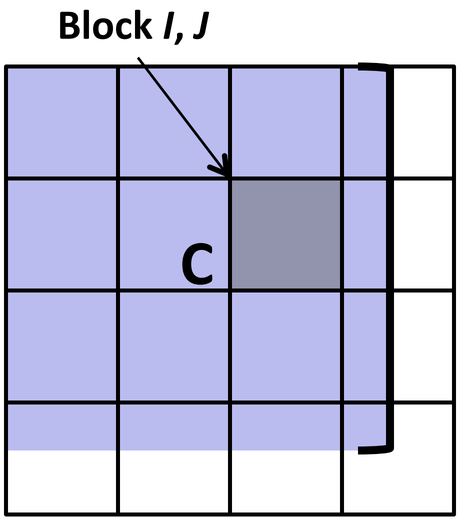
```
for (int bi = 0; bi < SBLK; bi++)
    for (int bj = 0; bj < SBLK; bj++) {
        float sum = 0.0;
        for (int bk =0; bk < SBLK; bk++)
            sum += matA[RM(i+bi,k+bk,N)] * tranB[RM(j+bj,k+bk,N)];
         matC[RM(i+bi,j+bj,N)] += sum;
    }
```

CPU Implementations

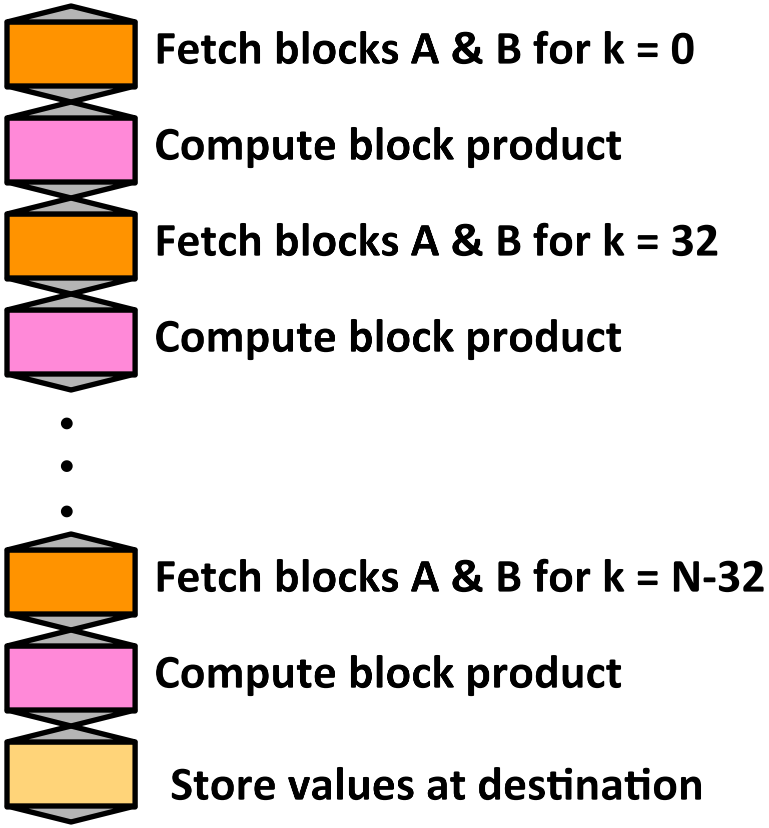
Fast blocked: Pretranspose + Unroll inner loop 8x and reassociate



Blocking with Cuda



- Block size `LBLK` = 32
- Use one Cuda block for each block of destination matrix
- Enough Cuda blocks to cover C
- Each thread in block accumulates single destination value



Fetch blocks A & B for k = 0

Compute block product

Fetch blocks A & B for k = 32

Compute block product

Fetch blocks A & B for k = N-32

Compute block product

Store values at destination

Cuda Block Kernel Structure

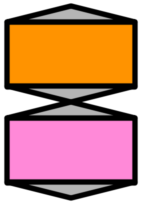
```
__global__ void
cudaBlockKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int bi = threadIdx.x;
    int bj = threadIdx.y;
    float sum = 0.0; // Accumulate result for C[i][j]
    // Shared space for two submatrices of A and B
    __shared__ float subA[LBLK*LBLK];
    __shared__ float subB[LBLK*LBLK];

    Loop over values of k

    if (i < N && j < N)
        dmatC[RM(i,j,N)] = sum;
}
```

- **Block size LBLK = 32**
  - `blockDim.x` = `blockDim.y` = 32
- **`i`, `j` index into source and destination arrays**
- **`bi`, `bj` index local arrays**

Cuda Block Loop Structure

```
    for (int k = 0; k < N; k+= LBLK) {

        Fetch elements bi, bj for local arrays subA and subB

        // Wait until entire block gets filled
        __syncthreads();

        Compute contribution to element i, j of output

        // Wait until all products computed
        syncthreads();
    }
```

- **Within loop, each thread plays two distinct roles**
  - Fetch elements from source arrays into shared memory
  - Compute one element of subblock product



Fetch blocks A & B for next value of k

Compute block product

Fetching Blocks

```
if (i < N && k+bj < N) {
    subA[RM(bi,bj,LBLK)] = dmatA[RM(i,k+bj,N)];
} else {
    subA[RM(bi,bj,LBLK)] = 0.0;
}
if (j < N && k+bi < N) {
    subB[RM(bi,bj,LBLK)] = dmatB[RM(k+bi,j,N)];
} else {
    subB[RM(bi,bj,LBLK)] = 0.0;
}
```

⬢ Fetch blocks A & B for next value of k

- **`k` is multiple of `LBLK`**
  - Coarse-grained
- **Fetch element `i, k+bj` from A to get `subA[bi,bj]`**
- **Fetch element `k+bi, j` from B to get `subB[bi,bj]`**
- **Set to 0 if out of range**

Computing Block Product

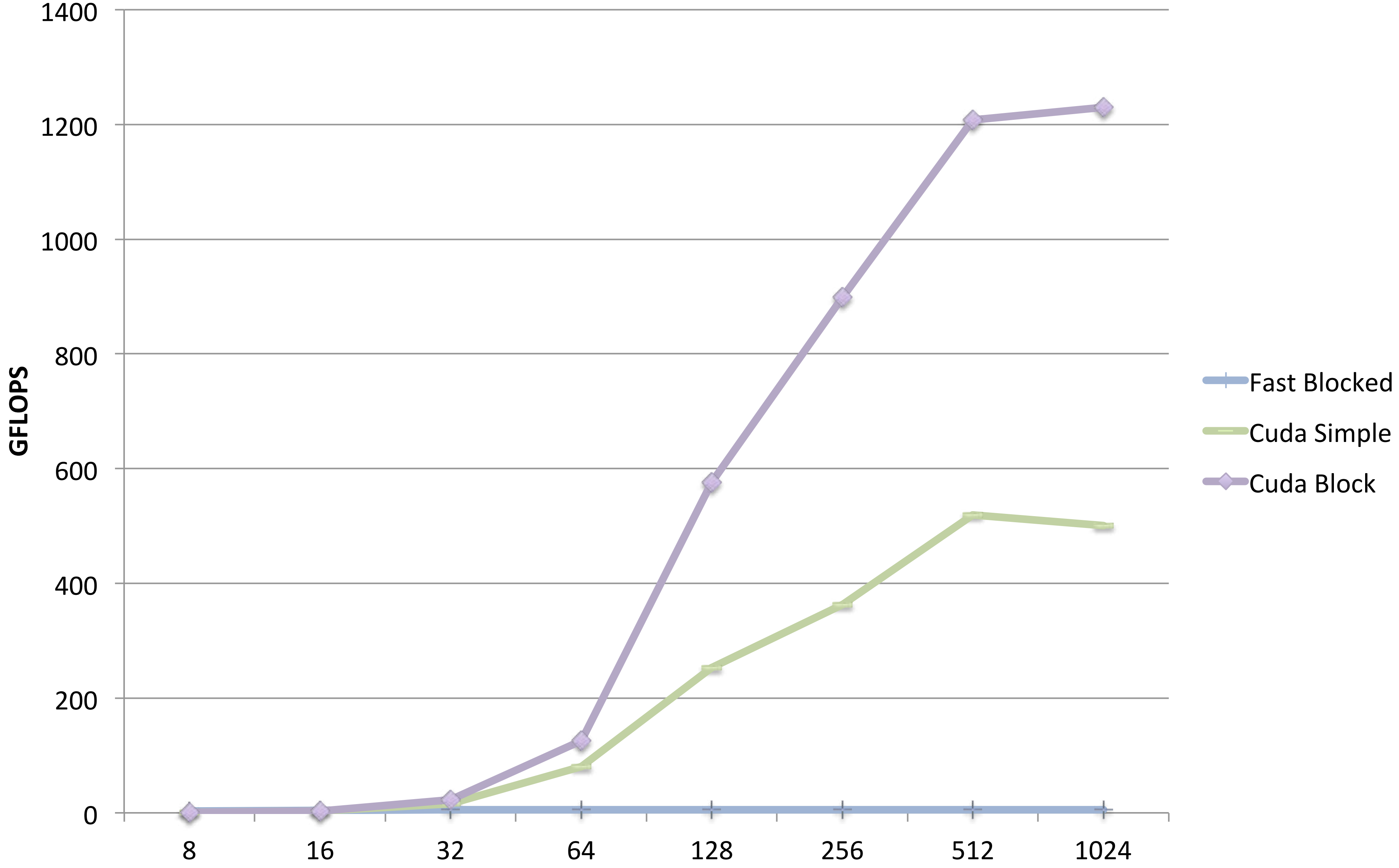
```
    for (int bk = 0; bk < LBLK; bk++)
        sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
```

- **Each thread in block accumulates single destination value**

Compute block product

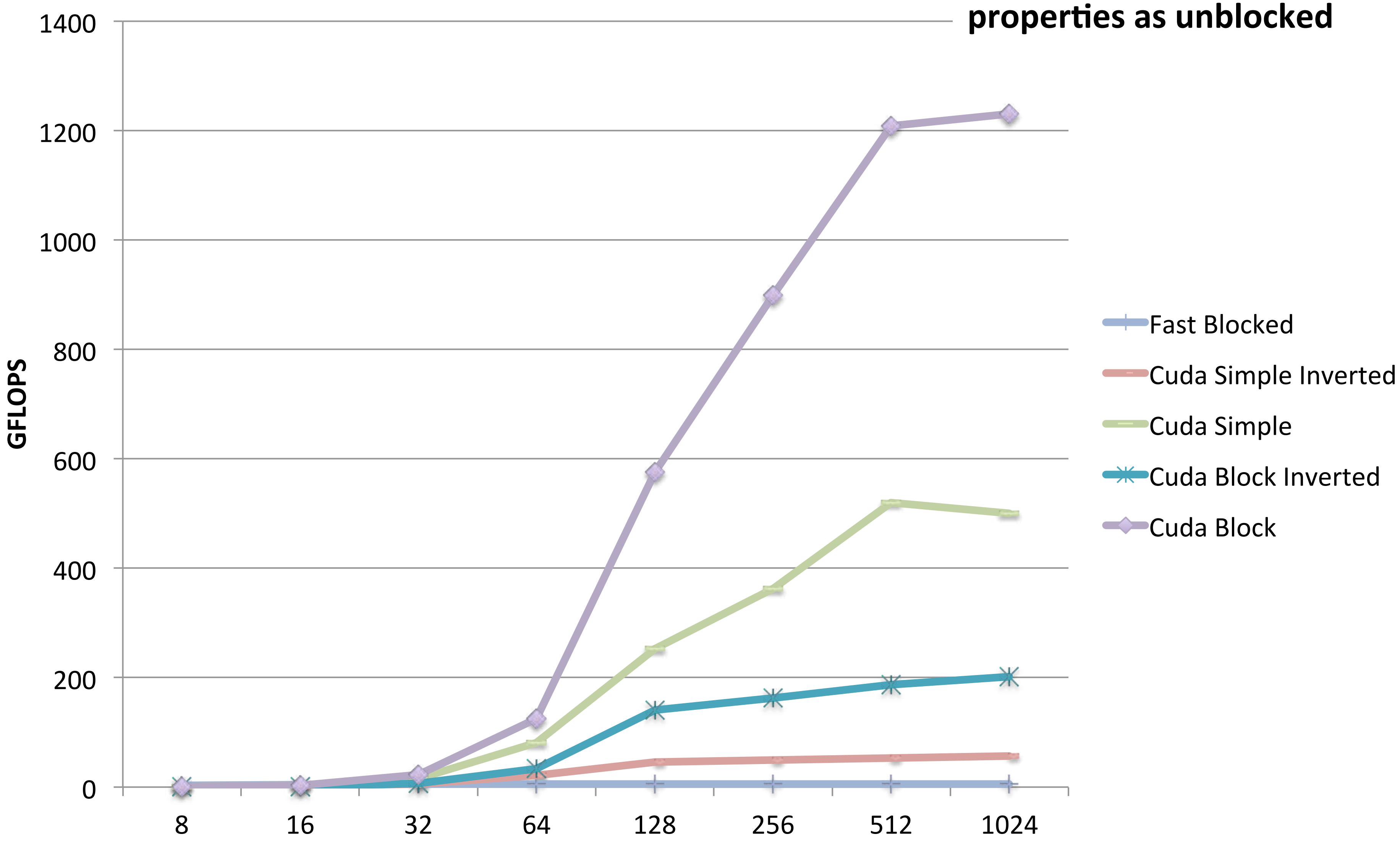
$$c_{bi,bj} = \sum_{bk=0}^{b-1} a_{bi,bk} \cdot b_{bk,bj}$$

CUDA Blocked Implementations



CUDA Inverted Indexing

Blocked version has similar indexing properties as unblocked



Warning!

```
for (int k = 0; k < N; k+= LBLK) {

    if (i >= N || j >= N)
        continue;  // Skip if out of bounds

    Computation when in-bounds

    // Wait until everyone finished
    __syncthreads();

    Compute contribution to element i, j of output
    // Wait until all products computed
    __syncthreads();
}
```

- **What's wrong with this code?**

Observations

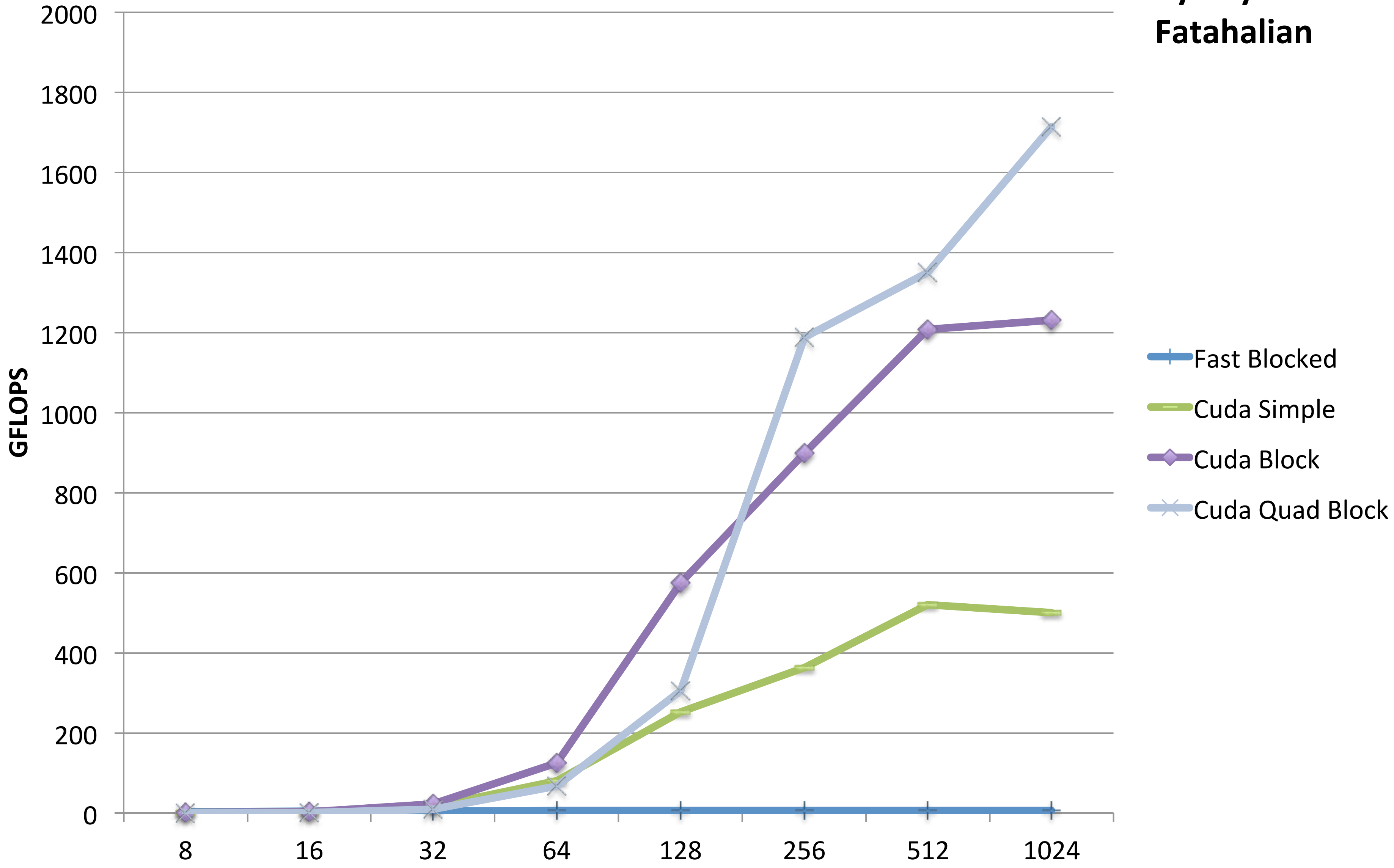
- **Making use of Cuda hierarchy can help**
  - Lighter weight synchronization
  - Shared access to fast memory
  - Different blocks can proceed at different rates
    - (Not shown in this example)
- **Advice**
  - Implement pure data-parallel version first
  - Only exploit hierarchy for performance critical parts
  - Watch out for synchronization bugs
  - Proper memory referencing more important than these low-level optimizations

But Wait...
THERE'S MORE!

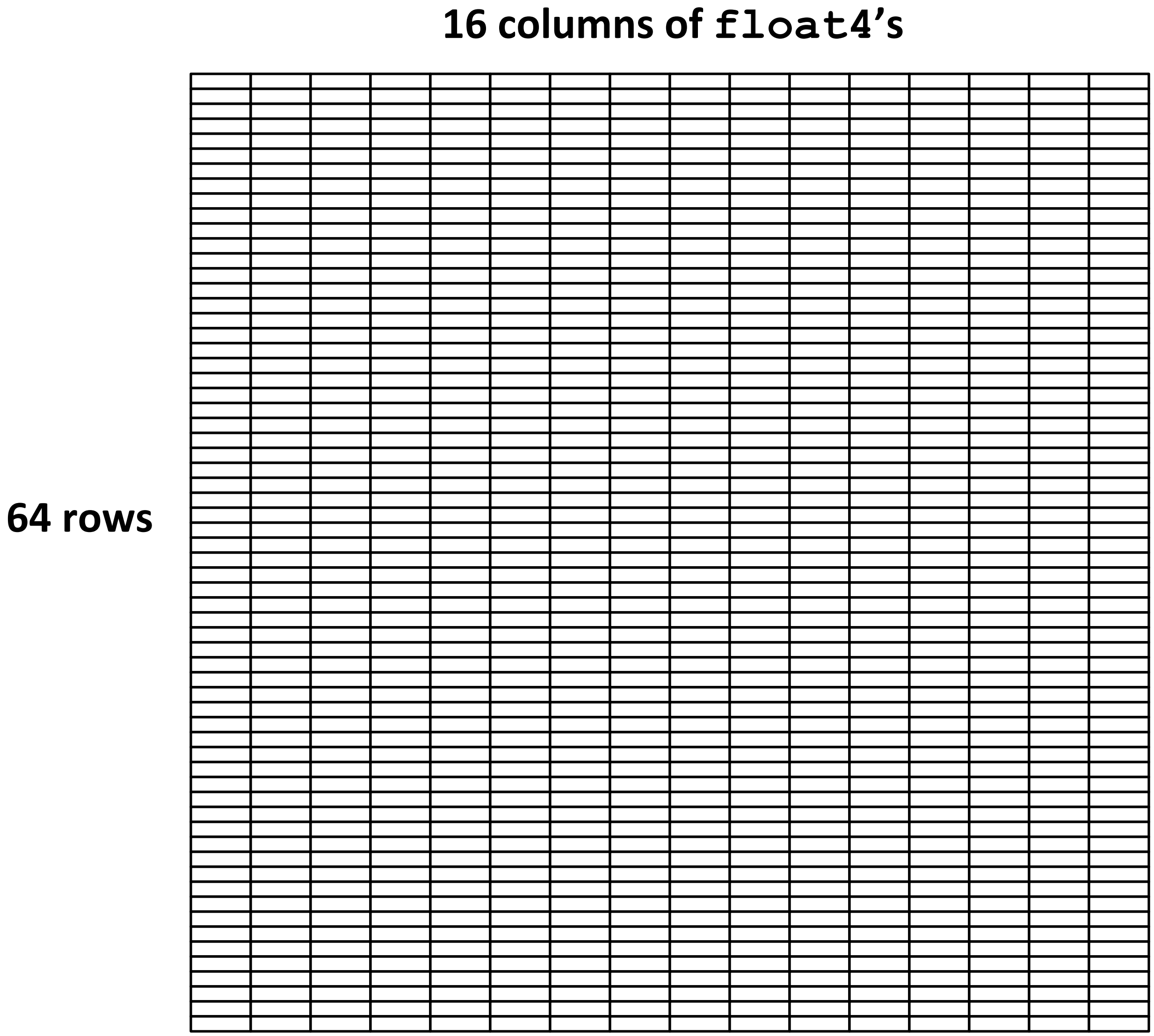
Sugg. Retail
$~~269~~
4 Easy Payments
$39.99
+S&H

Reading Memory with Float4's

Idea suggested by Kayvon Fatahalian

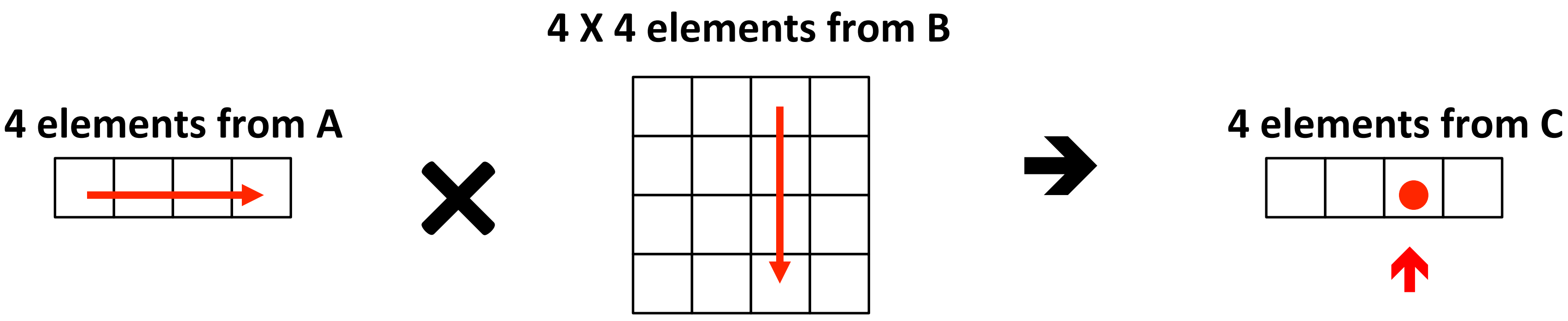


Idea



- **Thread blocks compute products of 64x64 submatrices**
- **1024 threads**
- **Organize as 64 rows X 16 columns**
- **Threads read & write memory in chunks of 16 bytes**
  - 4 `float`’s each

Added Inner Step of Computation



4 X 4 elements from B

4 elements from A

4 elements from C

- **Each thread loops 16 times**
  - Within loop, compute product:
    - 1x4 portion of A
    - 4x4 of B
    - Add sum to 1x4 portion of C
    - 16 multiplies, 16 adds
- **Why so fast?**
  - Makes maximum use of memory bus capability

Some Advice

- **Don't wire down constants**
- **Don't assume special properties of N**
  - Multiple of block size, power of 2, …
- **Use function or macro to do rounding-up division**
- **Write checker code**
  - Overall functionality
  - Individual steps on device
    - Must transfer data back to host to check
- **Avoid printf within kernel functions**
  - Only use on *small* examples
- **Get the algorithm & abstract implementation right before attempting low-level optimizations**
  - Exploiting the various memory categories on device
  - Exploiting properties specific to block level