
Massively Parallel Computing with Cuda

- Memory Access -

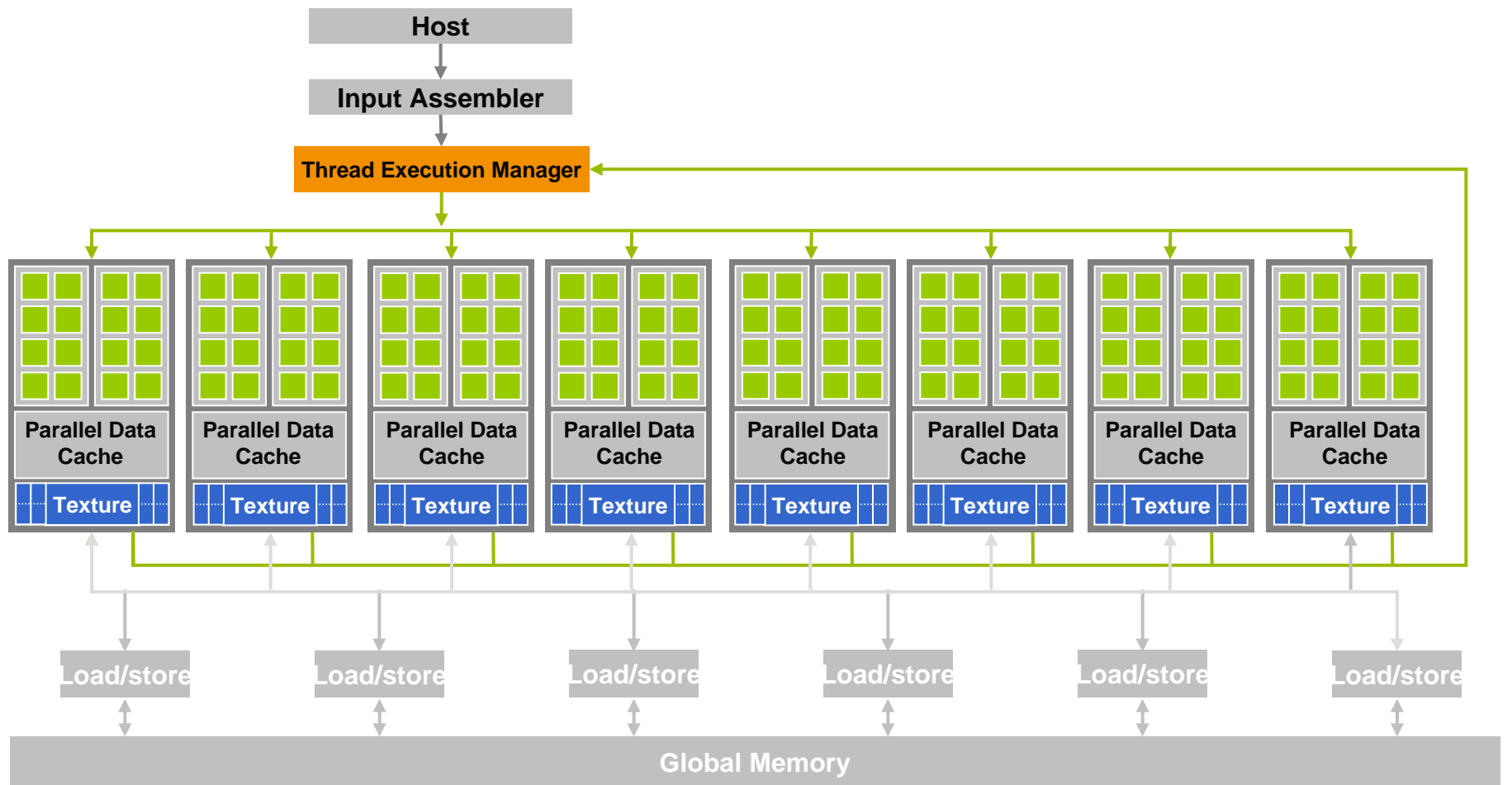
Hendrik Lensch
Robert Strzodka

Today

- **Memory**
 - Latency
 - Bandwidth
 - Hierarchy
- **Algorithms**
 - Vector-matrix multiplication
 - Matrix-matrix multiplication
 - Reduction

GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU

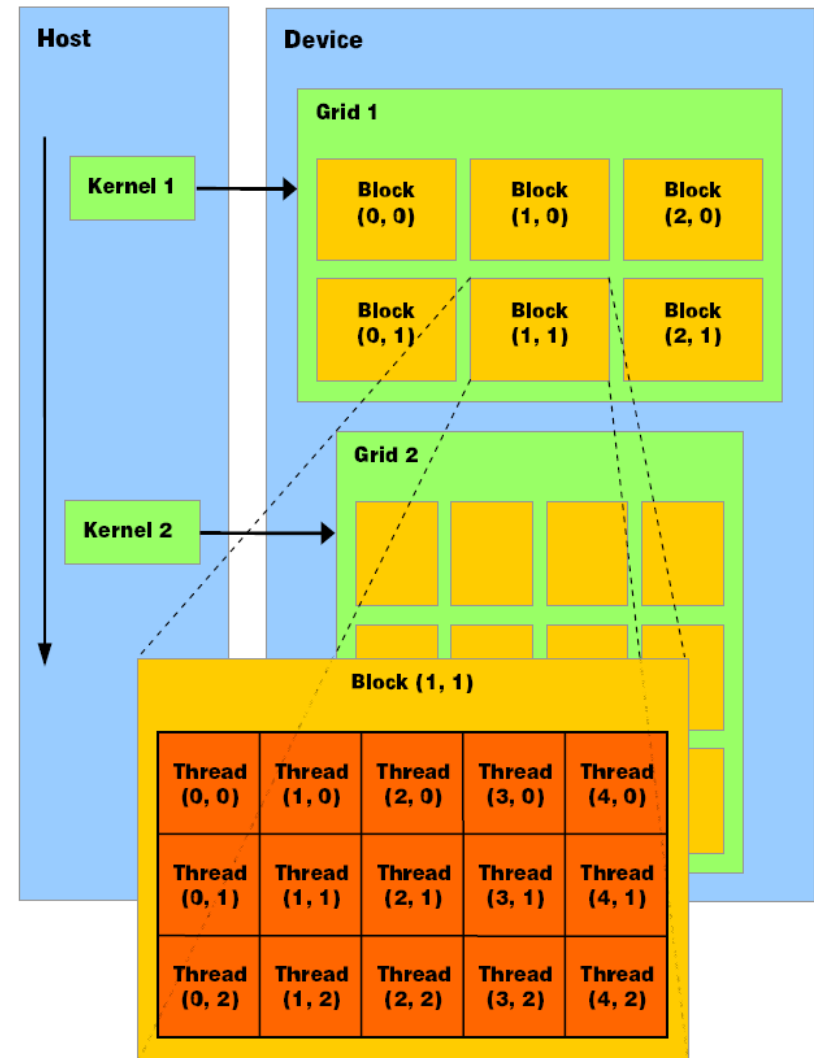


Programming Model

- **Programming Model**

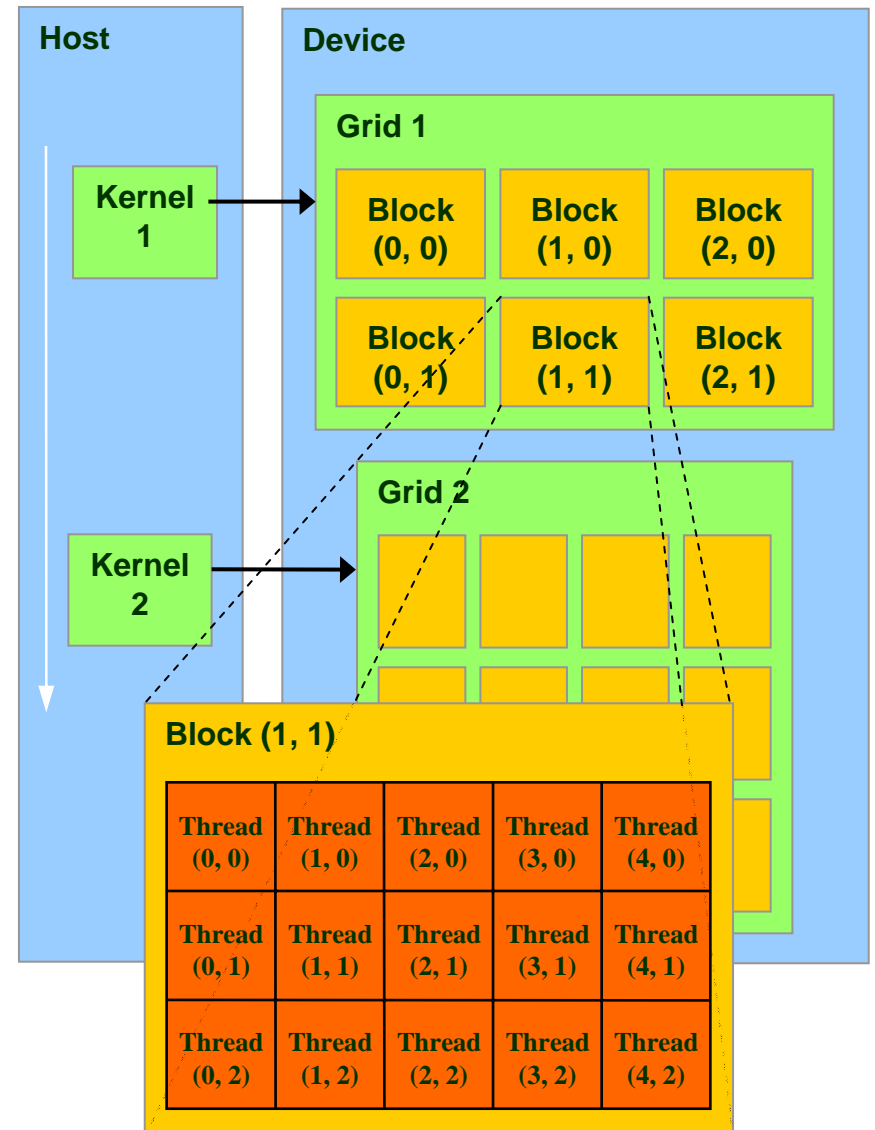
- The programmer writes a kernel (in C) for each task he or she wishes to perform
- The application splits the data to be processed into grids of thread blocks
- When a kernel is launched, each block is allocated to a single TP
- Threads of a given block are time sliced onto SPs contained within that block's TP

Many problems have natural grid structure, but decomposing data into threads can be difficult in general



Thread Batching: Grids and Blocks

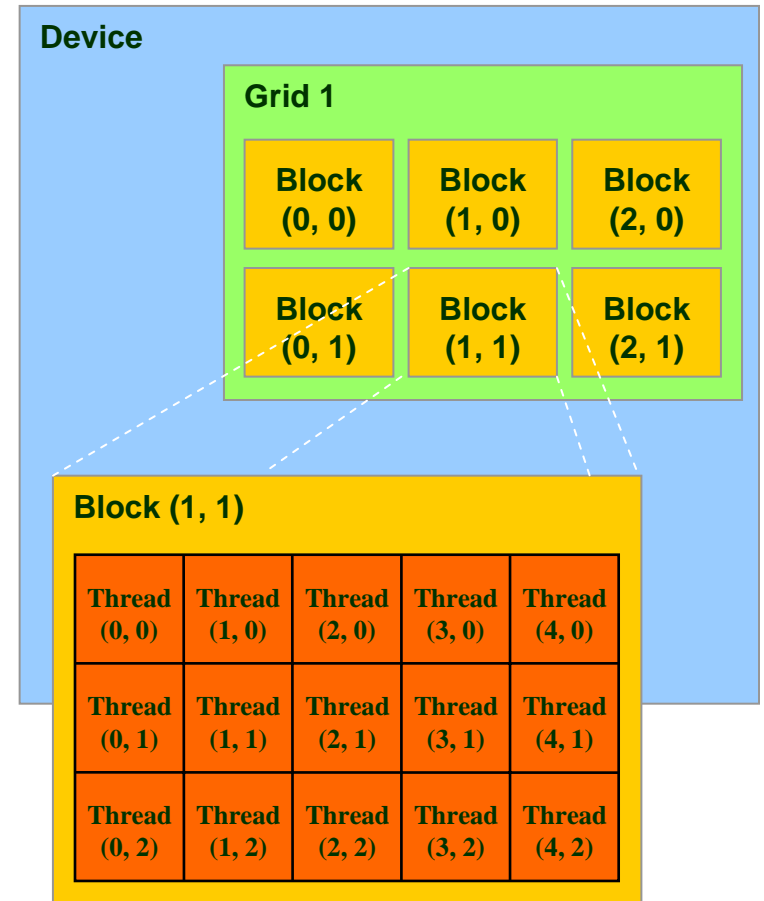
- A kernel is executed as a **grid of thread blocks**
 - All threads share data memory space
- A **thread block** is a batch of threads that can **cooperate** with each other by:
 - Synchronizing their execution
 - For hazard-free shared memory accesses
 - Efficiently sharing data through a low latency **shared memory**
- **Two threads from two different blocks cannot cooperate**



Courtesy: NVIDIA

Block and Thread IDs

- **Threads and blocks have IDs**
 - So each thread can decide what data to work on
 - Block ID: 1D or 2D
 - Thread ID: 1D, 2D, or 3D
- **Simplifies memory addressing when processing multidimensional data**
 - Image processing
 - Solving PDEs on volumes
 - ...



Courtesy: NVIDIA

Quick Terminology Review

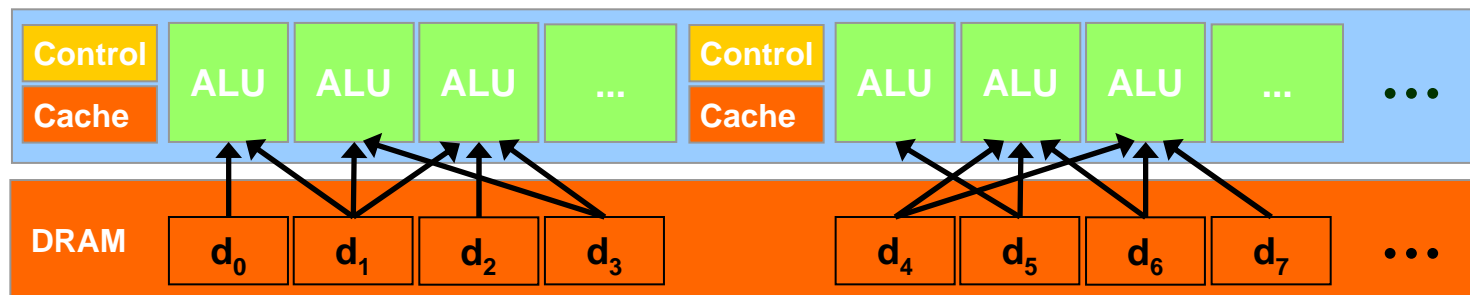
- ***Thread***: concurrent code and associated state executed on the **CUDA device** (in parallel with other threads)
 - The unit of parallelism in CUDA
- ***Warp***: a group of threads executed *physically* in parallel in G80
- ***Block***: a group of threads that are executed together and form the unit of resource assignment
- ***Grid***: a group of thread blocks that must all complete before the next phase of the program can begin

How Thread Blocks are Partitioned

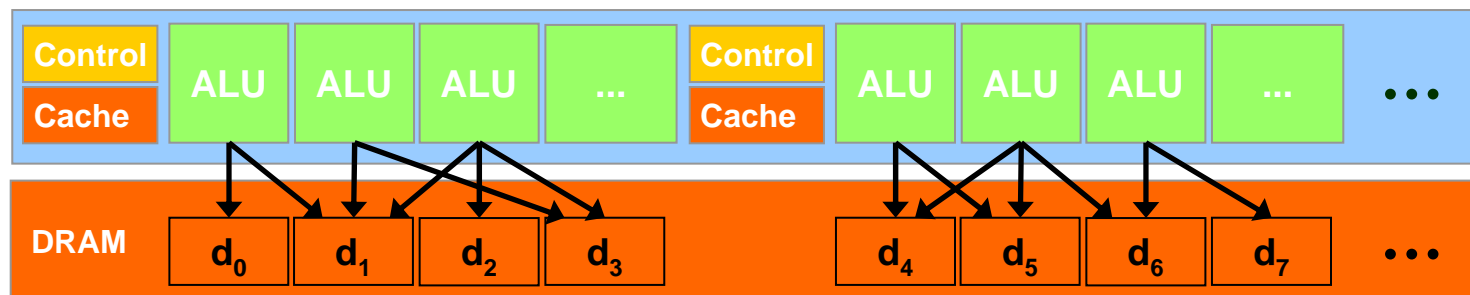
- **Thread blocks are partitioned into warps**
 - Thread IDs within a warp are consecutive and increasing
 - Warp 0 starts with Thread ID 0
- **Partitioning is always the same**
 - Thus you can use this knowledge in control flow
 - However, the exact size of warps may change from generation to generation
 - (Covered next)
- **However, DO NOT rely on any ordering between warps**
 - If there are any dependencies between threads, you must `__syncthreads()` to get correct results

CUDA Highlights: Scatter

- **CUDA provides generic DRAM memory addressing**
 - Gather:



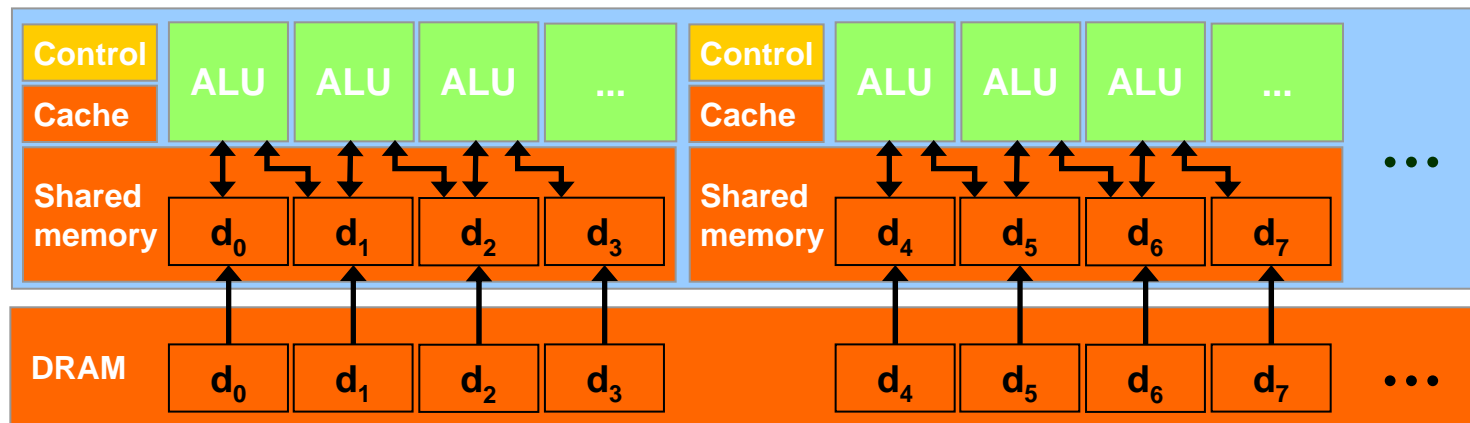
- And **scatter**: no longer limited to write one pixel



➔ **More programming flexibility**

CUDA Highlights: On-Chip Shared Memory

- **CUDA enables access to a parallel **on-chip shared memory** for efficient inter-thread data sharing**

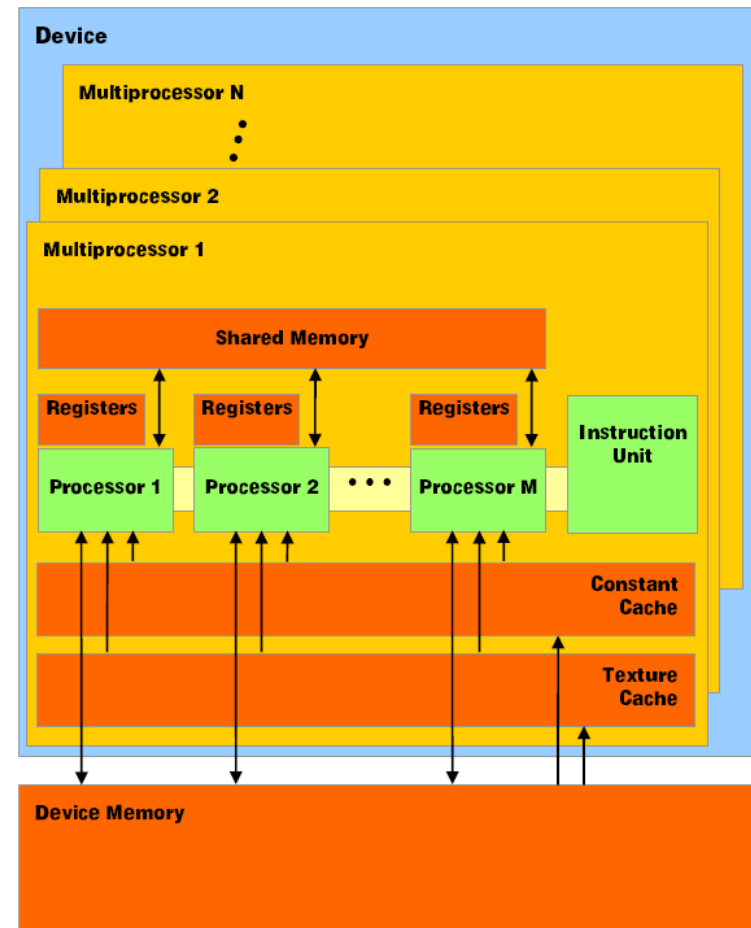


➔ **Big memory bandwidth savings**

Memory Hierarchy

Programming Model: Memory Spaces

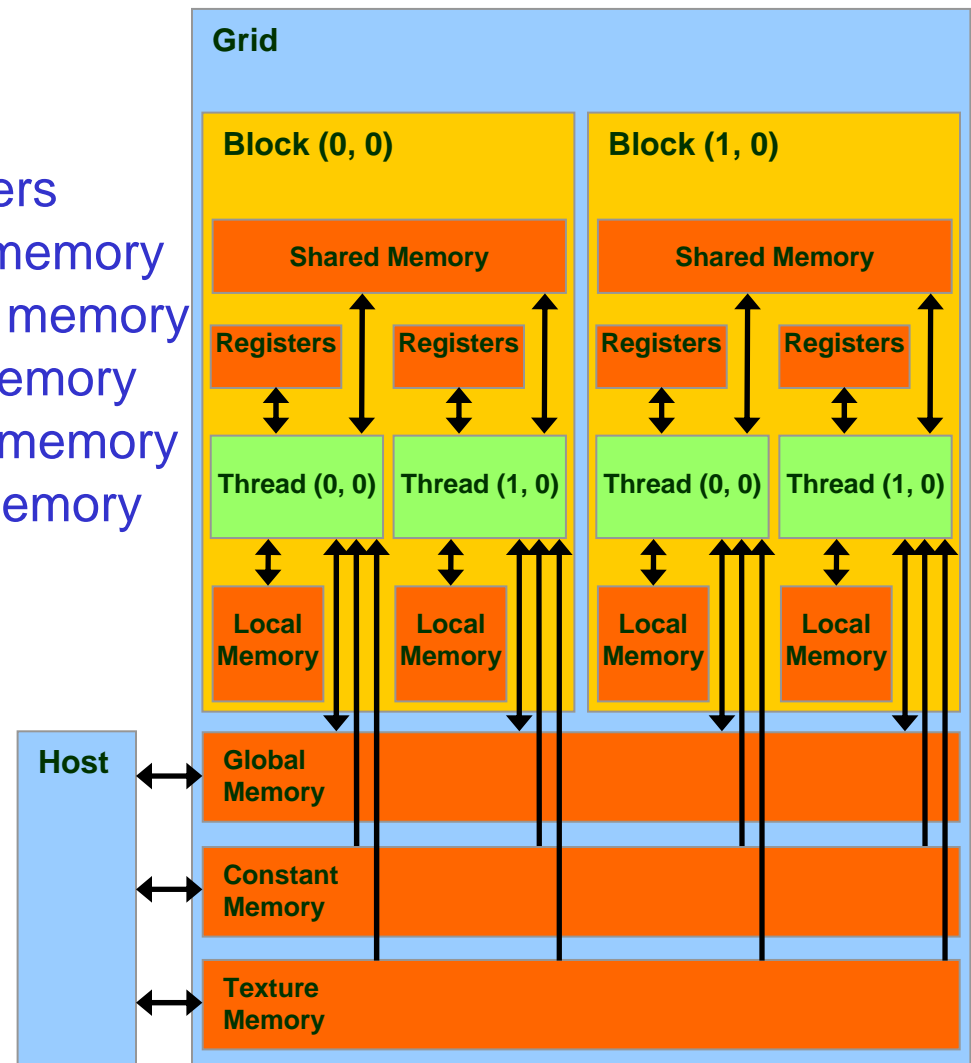
- **Global Memory**
 - Read-write per-grid
 - Hundreds of MBs
 - Very slow (600 clocks)
- **Texture Memory**
 - Read-only per-grid
 - Hundreds of MBs
 - Slow first access, but cached
 - Built-in filtering, clamping
- **Constant Memory**
- **Shared! Memory**
 - Read-write per-block
 - 16 KB per block
 - Very fast (4 clocks)
- **Registers**
 - Unique per thread



Programming Model: Memory Spaces

- **Each thread can:**
 - Read/write per-thread registers
 - Read/write per-thread local memory
 - Read/write per-block shared memory
 - Read/write per-grid global memory
 - Read only per-grid constant memory
 - Read only per-grid texture memory

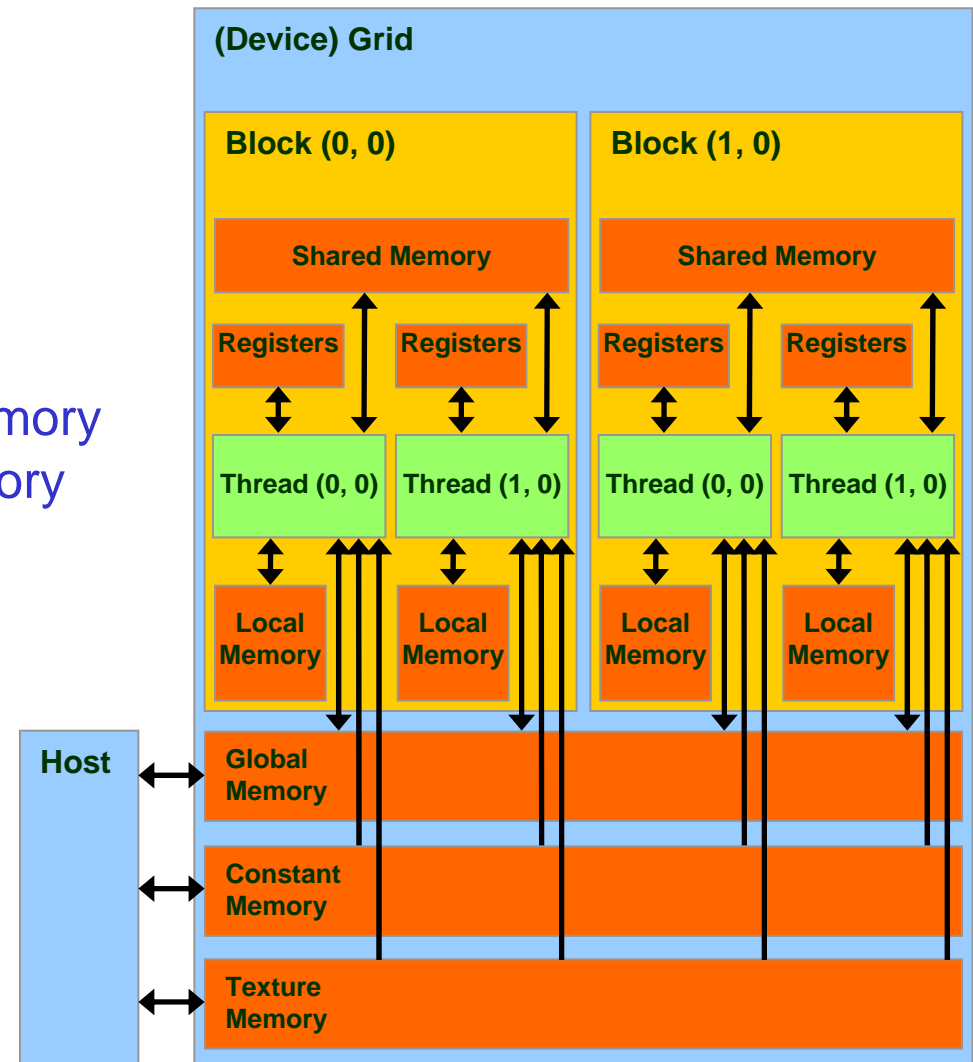
- **The host can read/write global, constant, and texture memory**



CUDA Device Memory Space

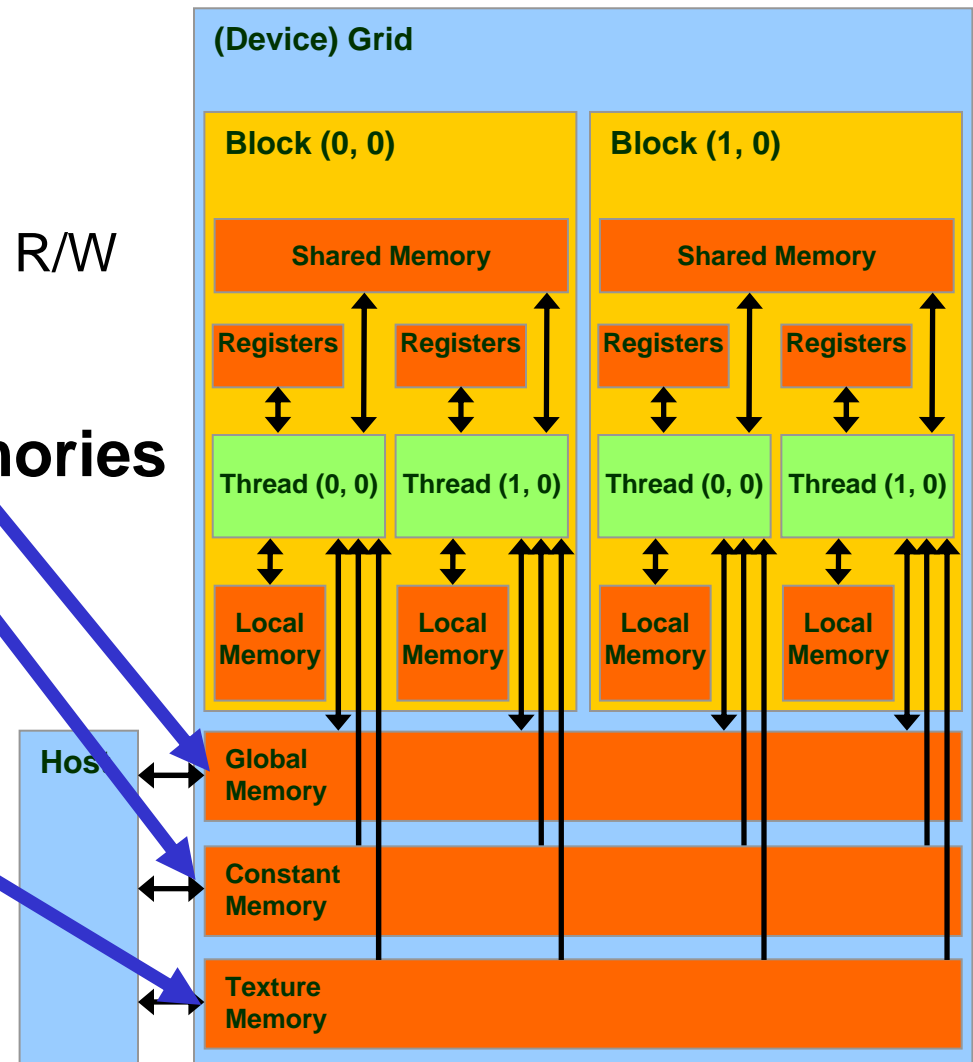
- **Each thread can:**
 - R/W per-thread registers
 - R/W per-thread local memory
 - R/W per-block shared memory
 - R/W per-grid global memory
 - Read only per-grid constant memory
 - Read only per-grid texture memory

- **The host can R/W global, constant, and texture memories**



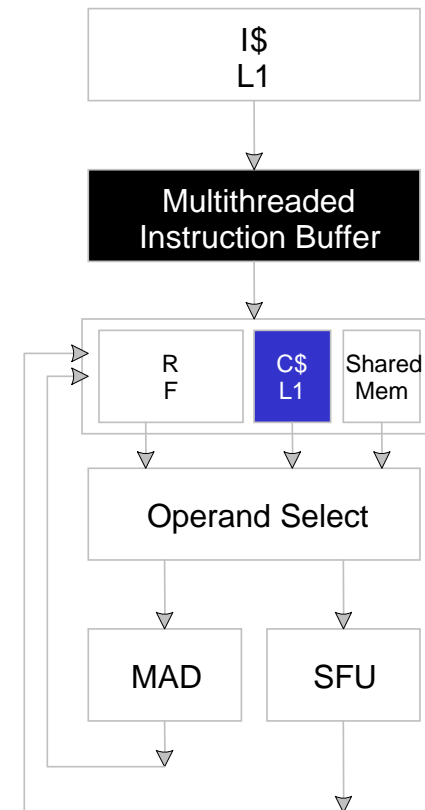
Global, Constant, and Texture Memories (Long Latency Accesses)

- **Global memory**
 - Main means of communicating R/W Data between **host** and **device**
 - Contents visible to all threads
- **Texture and Constant Memories**
 - Constants initialized by host
 - Contents visible to all threads



Constants

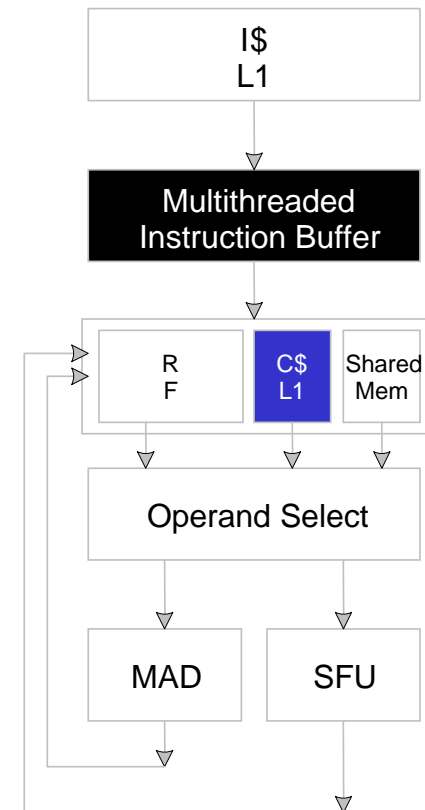
- **Immediate address constants**
- **Indexed address constants**
- **Constants stored in DRAM, and cached on chip**
 - L1 per SM
- **A constant value can be broadcast to all threads in a Warp**
 - Extremely efficient way of accessing a value that is common for all threads in a block!



Constants

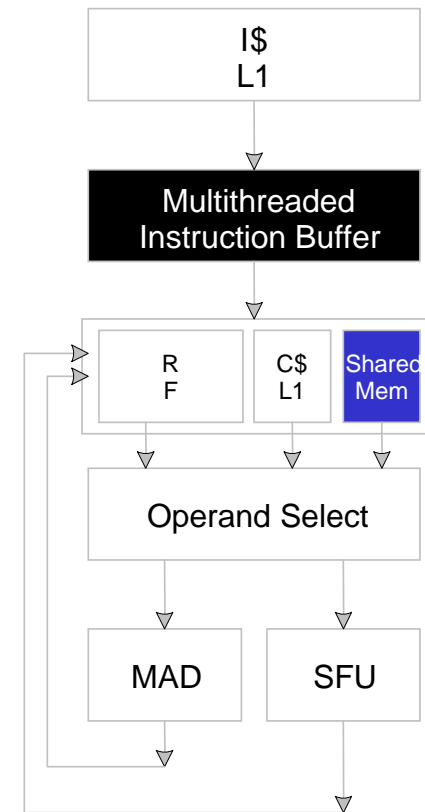
- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
 - L1 per SM
- A constant value can be broadcast to all threads in a Warp
 - Extremely efficient way of accessing a value that is common for all threads in a block!

```
// specify as global variable  
__device__ __constant__ float gpuGamma[2];  
...  
// copy gamma value to constant device memory  
cudaMemcpyToSymbol(gpuGamma, &gamma, sizeof(float));  
// access as global variable in kernel  
res = gpuGamma[0] * threadIdx.x;
```



Shared Memory

- **Each SM has 16 KB of Shared Memory**
 - 16 banks of 32bit words
- **CUDA uses Shared Memory as shared storage visible to all threads in a thread block**
 - read and write access
- **Not used explicitly for pixel shader programs**
 - we dislike pixels talking to each other 😊



Shared Memory Allocation

- 2 modes
- **Static size within kernel**

```
__shared__ float vec[256];
```

- **Dynamic size when calling the kernel**

```
// in main
```

```
int VecSize = MAX_THREADS * sizeof(float4);
```

```
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, ...);
```

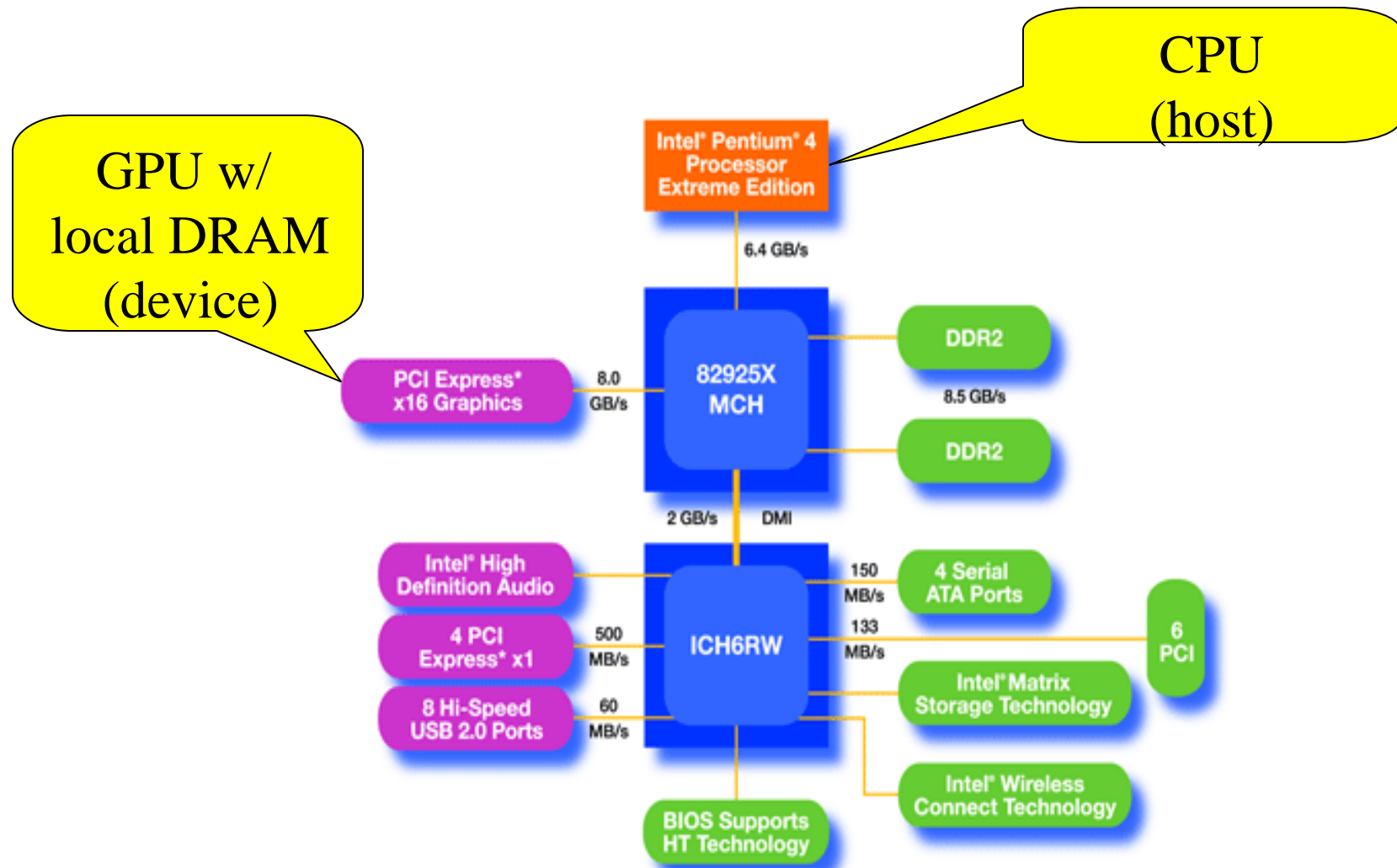
```
// declare as extern within kernel
```

```
extern __shared__ float vec[];
```

Access Times

- **Register – dedicated HW - single cycle**
- **Shared Memory – dedicated HW – 2-4 cycles**
- **Local Memory – DRAM, no cache - *slow***
- **Global Memory – DRAM, no cache - *slow***
- **Constant Memory – DRAM, cached, 1...10s...100s of cycles, depending on cache locality**
- **Texture Memory – DRAM, cached, 1...10s...100s of cycles, depending on cache locality**
- **Instruction Memory (invisible) – DRAM, cached**

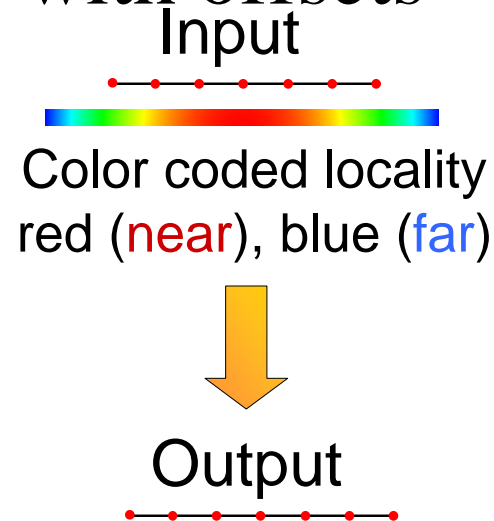
An Example of Physical Reality Behind CUDA



Native Memory Layout – Data Locality

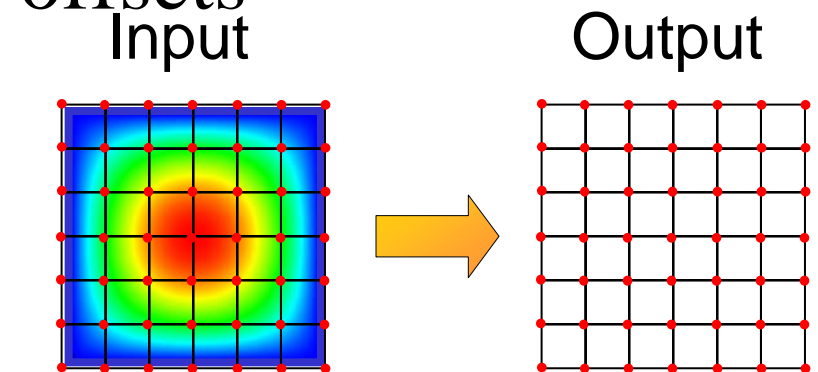
CPU

- 1D input
- 1D output
- Other dimensions with offsets



GPU

- 2D input
- 2D output
- Other dimensions with offsets

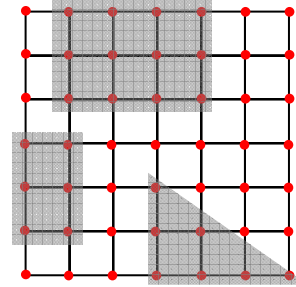


Primitive Index Regions in Output Arrays

- **Quads and Triangles**

- Fastest option

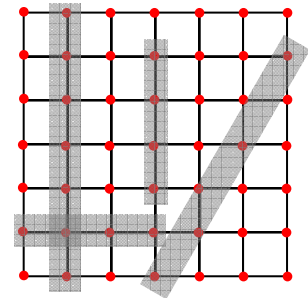
Output region



- **Line segments**

- Slower, try to pair lines to 2xh, wx2 quads

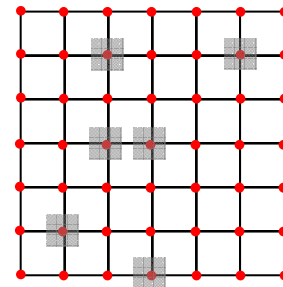
Output region



- **Point Clouds**

- Slowest, try to gather points into larger forms

Output region



GPUs are Optimized for Local Data Access

Memory access types: **Cache**, **Sequential**, **Random**

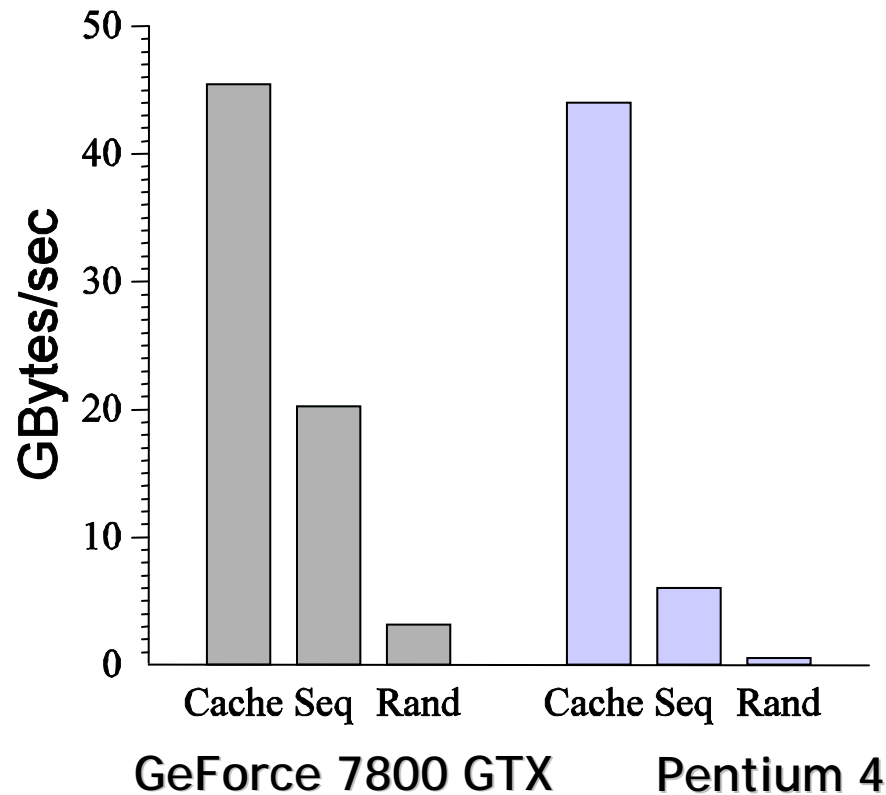


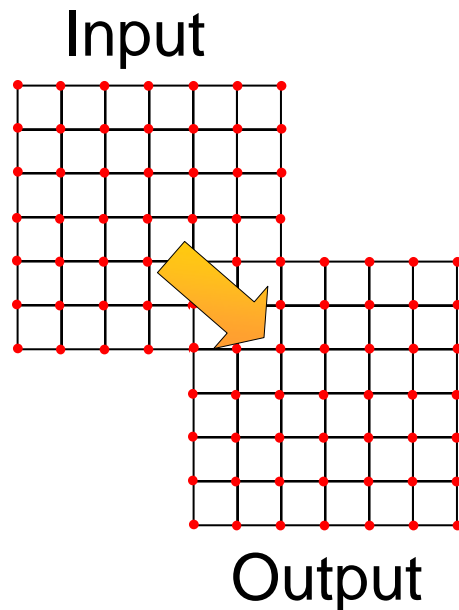
chart courtesy
of Ian Buck

- **CPU**
 - **Large** cache
 - **Few** processing elements
 - Optimized for **spatial** and **temporal** data reuse
- **GPU**
 - **Small** cache
 - **Many** processing elements
 - Optimized for **sequential** (streaming) data access

Input and Output Arrays

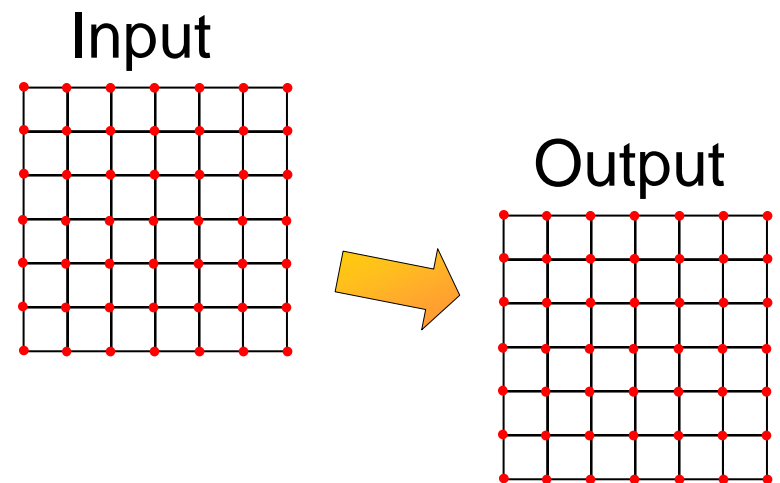
CPU

- Input and output arrays **may** overlap



GPU

- Input and output arrays **must not** overlap



Configuration Overhead

- Look for problems with a high computational load per data item!

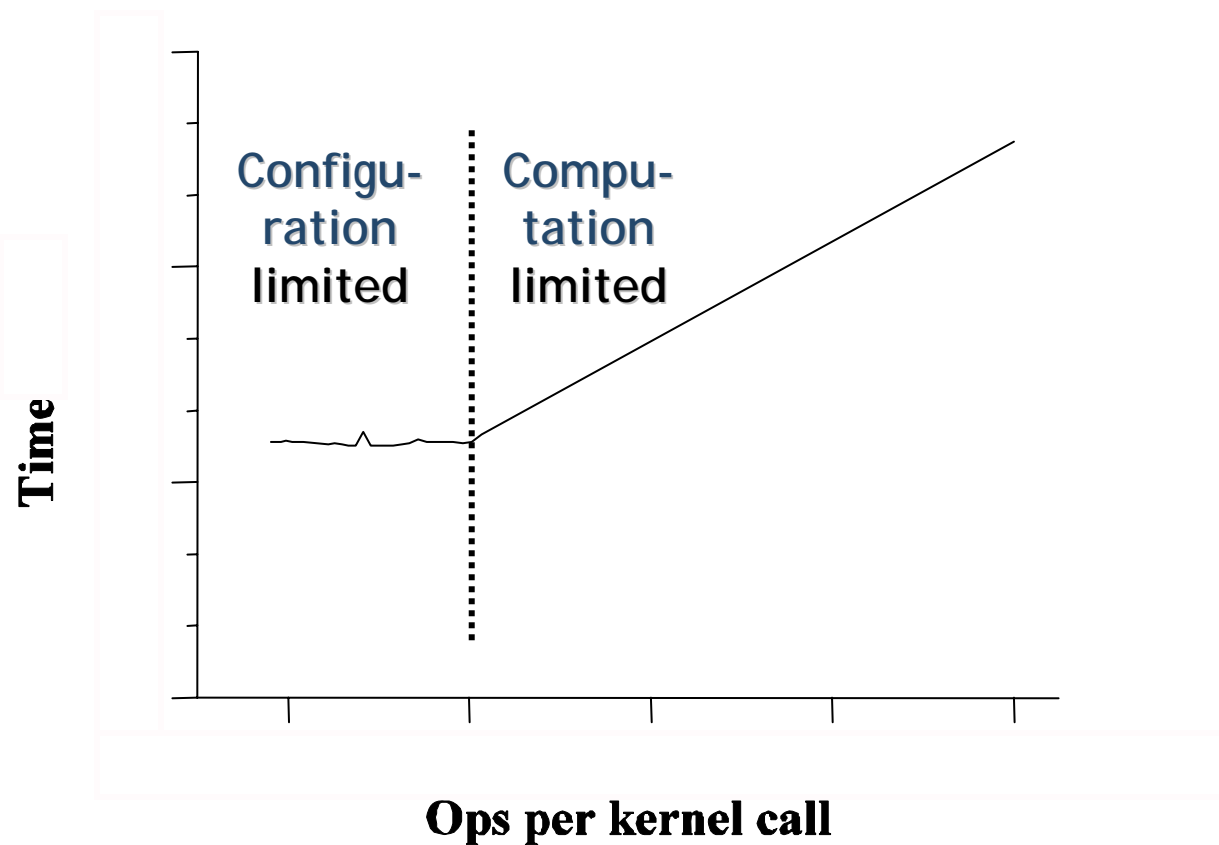


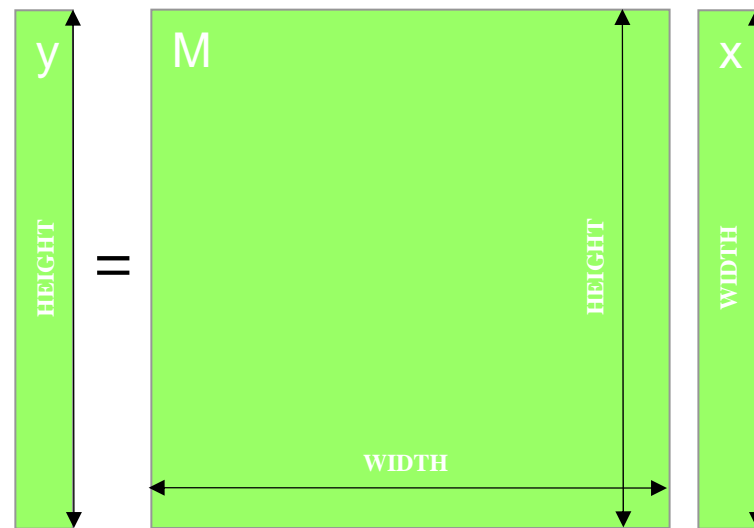
chart courtesy
of Ian Buck

Vector-Matrix Multiplication - data parallelism -

$$y=mx$$

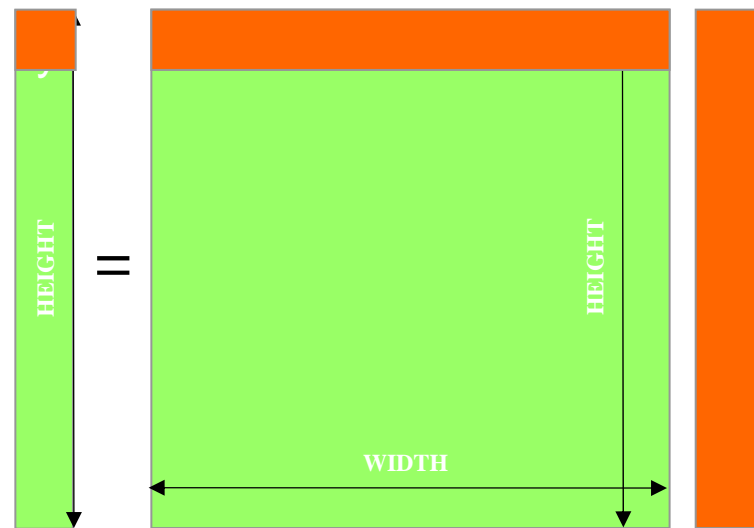
Vector-Matrix Multiplication V1

- Every thread computes a single output value in y
- Every thread computes the dot product between one line of M and x



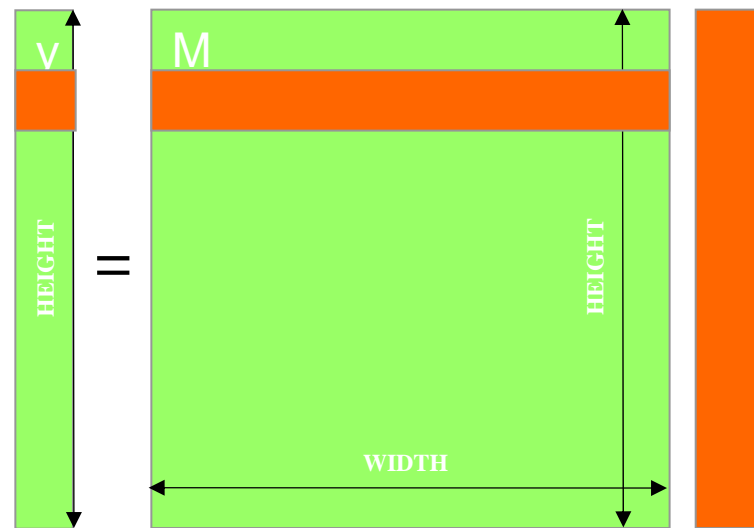
Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of M and x



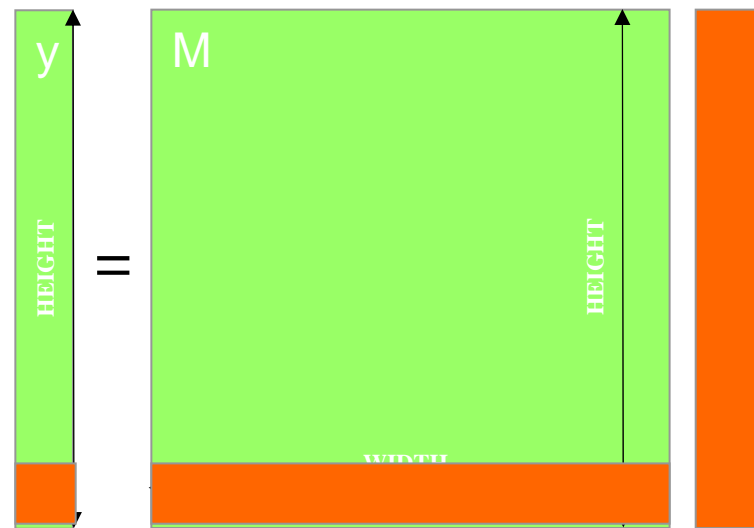
Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of M and x



Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of M and x
- Computations totally independent



Setup

```
... // allocate memory
float* gpuMat, gpuVec, gpuResVec;
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuMat, w*h* sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuVec, w * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuResVec, h * sizeof(float) )
);
CUT_CHECK_ERROR("allocation failed\n");
// upload M and x
CUDA_SAFE_CALL( cudaMemcpy( gpuMat, hostMat, w*h * sizeof(float),
    cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL( cudaMemcpy( gpuVec, hostVec, w * sizeof(float),
    cudaMemcpyHostToDevice) );
// compute the block and grid dimensions
dim3 threadBlock( MAX_THREADS, 1 );
dim3 blockGrid( h / MAX_THREADS + 1, 1, 1);
vecMat1<<< blockGrid, threadBlock >>>( gpuResVec, gpuMat, gpuVec,
    w,h);
CUT_CHECK_ERROR("vecMat filter failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );
// download result y
CUDA_SAFE_CALL( cudaMemcpy( hostResVec, gpuResVec, h * sizeof(float),
    cudaMemcpyDeviceToHost) );
cudaFree( gpuMat ); cudaFree( gpuVec ); cudaFree( gpuResVec );
```

VecMat Kernel – Version 1

```
__global__ void vecMat1(float *_dst, const float* _mat,
    const float* _v, int _w, int _h ) {

    // row index the thread is operating on
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < _h) {
        float res = 0.;

        // dot product of one line
        for (int j = 0; j < _w; ++j) {
            res += _mat[i*_w + j] * _v[j];
        }
        // write result to global memory
        _dst[i] = res;
    }
}
```

Why is this slow?

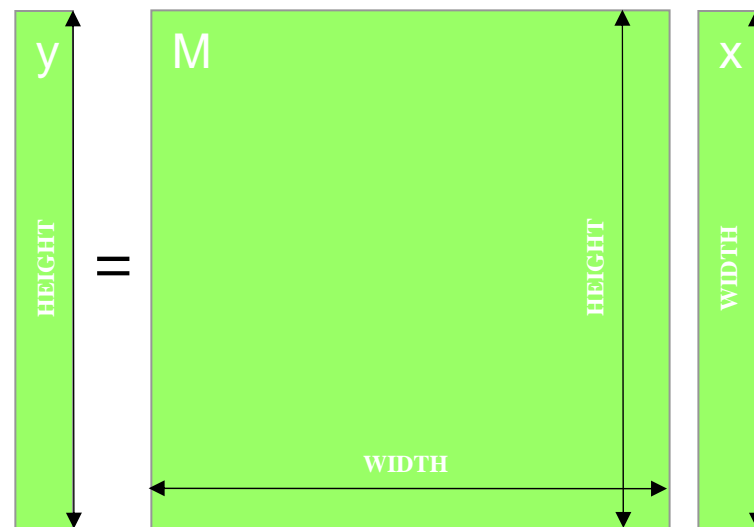
- **Problem is bandwidth limited (read)**
- **Each thread is accessing**
 - w elements of M
 - w elements of x**from global memory**
- **Total bandwidth: $2 * w * h$**
- **But all threads are accessing the same elements of x**
- **Load x into shared memory and reuse!**

Vector-Matrix Multiplication - using shared memory -

$$y=mx$$

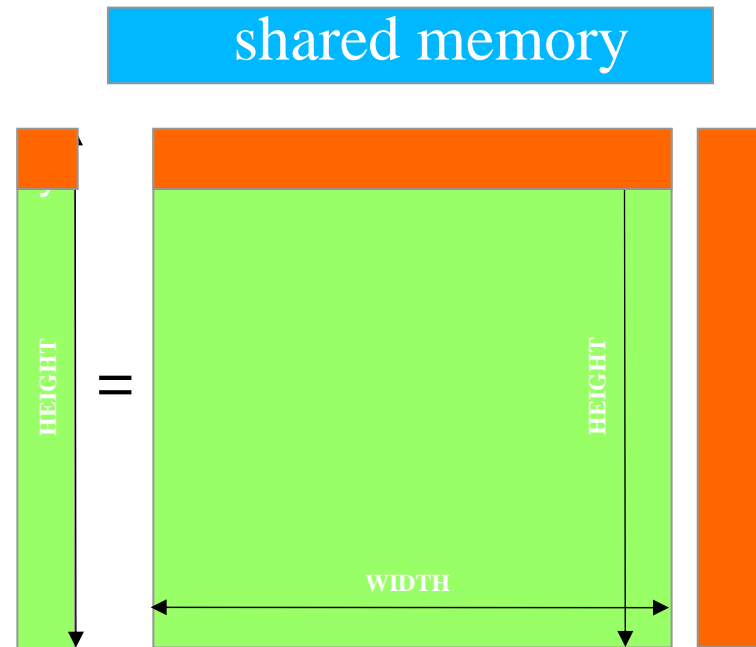
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



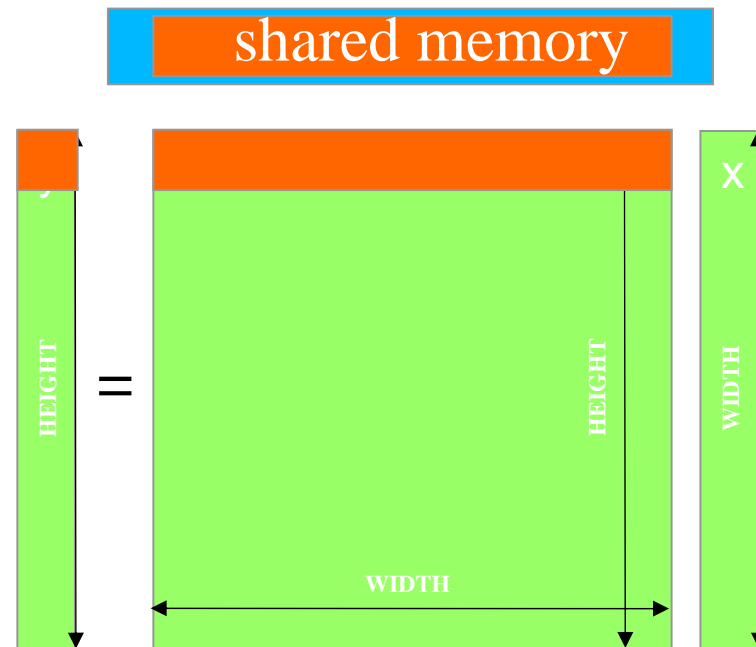
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



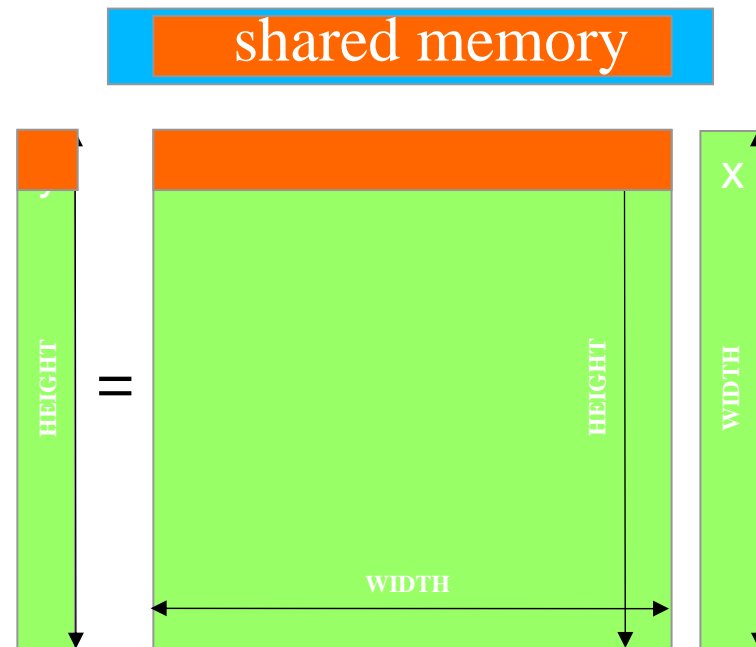
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



Setup – Version 2

```
... // allocate memory
float* gpuMat, gpuVec, gpuResVec;
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuMat, w*h* sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuVec, w * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuResVec, h * sizeof(float) )
);
CUT_CHECK_ERROR("allocation failed\n");
// upload M and x
CUDA_SAFE_CALL( cudaMemcpy( gpuMat, hostMat, w*h * sizeof(float),
    cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL( cudaMemcpy( gpuVec, hostVec, w * sizeof(float),
    cudaMemcpyHostToDevice) );
// compute the block and grid dimensions
dim3 threadBlock( MAX_THREADS, 1 );
dim3 blockGrid( h / MAX_THREADS + 1, 1, 1);
vecMat2<<< blockGrid, threadBlock, w * sizeof(float) >>>( gpuResVec,
    gpuMat, gpuVec, w,h, w / MAX_THREADS);
CUT_CHECK_ERROR("vecMat filter failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );
// download result y
CUDA_SAFE_CALL( cudaMemcpy( hostResVec, gpuResVec, h * sizeof(float),
    cudaMemcpyDeviceToHost) );
cudaFree( gpuMat ); cudaFree( gpuVec ); cudaFree( gpuResVec );
```

VecMat Kernel – Version 2

```
__global__ void vecMat2(float *_dst, const float* _mat, const float*
    _v, int _w, int _h, int nIter ) {
extern __shared__ float vec[];

int i = blockIdx.x * blockDim.x + threadIdx.x;
float res = 0.; int vOffs = 0;

// load x into shared memory
for (int iter = 0; iter < nIter; ++iter, vOffs += blockDim.x) {
vec[vOffs + threadIdx.x] = _v[vOffs + threadIdx.x];
}
// make sure all threads have written their parts
__syncthreads();

// now compute the dot product again
// use elements of x loaded by other threads!
if (i < _h) {
    for (int j = 0; j < _w; ++j) {
        res += _mat[offs + j ] * vec[j];
    }
    _dst[i] = res;
}}
```

VecMat Kernel – Version 2

```
__global__ void vecMat2(float *_dst, const float* _mat, const float*  
_v, int _w, int _h, int nIter ) {
```

```
extern __shared__ float vec[];
```

```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
float res = 0.; int vOffs = 0;
```

```
// load x into shared memory  
for (int iter = 0; iter < nIter; ++iter, vOffs += blockDim.x) {  
vec[vOffs + threadIdx.x] = _v[vOffs + threadIdx.x];  
}
```

```
// make sure all threads have written their parts  
__syncthreads();
```

```
// now compute the dot product again  
// use elements of x loaded by other threads!  
if (i < _h) {  
for (int j = 0; j < _w; ++j) {  
res += _mat[offs + j ] * vec[j];  
}  
_dst[i] = res;  
}}
```

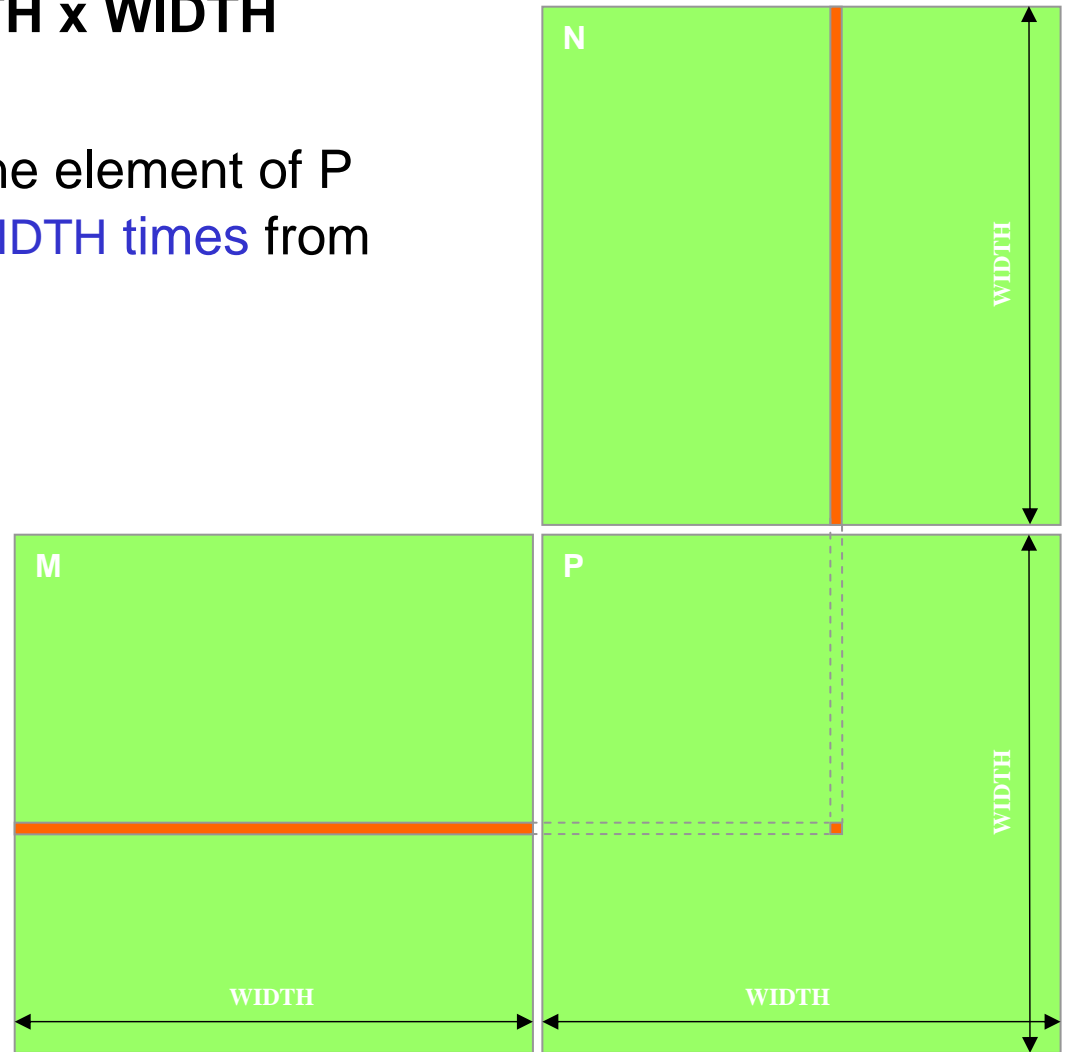
up to 4 times speedup!

Matrix-Matrix Multiplication

$$P=MN$$

Programming Model: Square Matrix Multiplication

- $P = M * N$ of size WIDTH x WIDTH
- Without tiling:
 - One **thread** handles one element of P
 - M and N are loaded WIDTH times from global memory



Step 1: Matrix Data Transfers

```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size  = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

Step 2: Matrix Multiplication

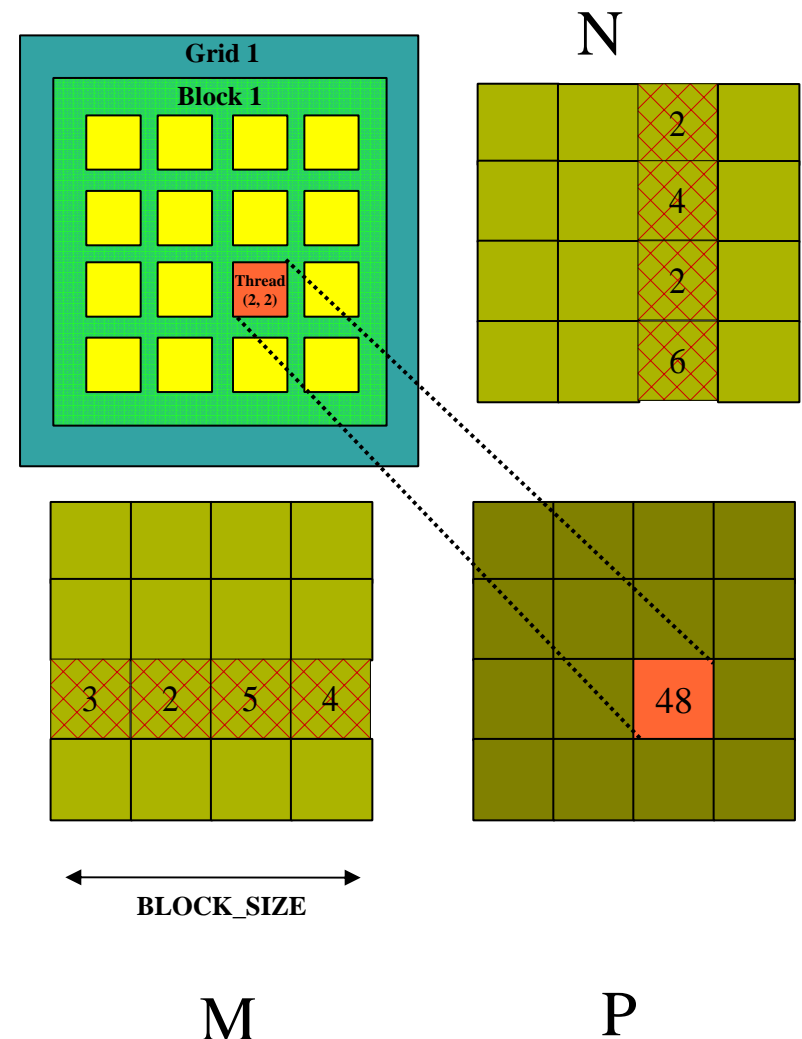
A Simple Host Code in C

```
// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal
```

```
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
```

Multiply Using One Thread Block

- **One block of threads computes matrix P**
 - Each thread computes one element of P
- **Each thread**
 - Loads a row of matrix M
 - Loads a column of matrix N
 - Perform one multiply and addition for each pair of M and N elements
 - Compute to off-chip memory access ratio close to 1:1 (not very high)
- **Size of matrix limited by the number of threads allowed in a thread block**



Step 3: Matrix Multiplication

Host-side Main Program Code

```
int main(void) {  
    // Allocate and initialize the matrices  
    Matrix M = AllocateMatrix(WIDTH, WIDTH, 1);  
    Matrix N = AllocateMatrix(WIDTH, WIDTH, 1);  
    Matrix P = AllocateMatrix(WIDTH, WIDTH, 0);  
  
    // M * N on the device  
    MatrixMulOnDevice(M, N, P);  
  
    // Free matrices  
    FreeMatrix(M);  
    FreeMatrix(N);  
    FreeMatrix(P);  
    return 0;  
}
```

Step 3: Matrix Multiplication

Host-side code

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
}
```

Step 3: Matrix Multiplication

Host-side Code (cont.)

```
// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

// Read P from the device
CopyFromDeviceMatrix(P, Pd);

// Free device matrices
FreeDeviceMatrix(Md);
FreeDeviceMatrix(Nd);
FreeDeviceMatrix(Pd);
}
```

Step 4: Matrix Multiplication

Device-side Kernel Function

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

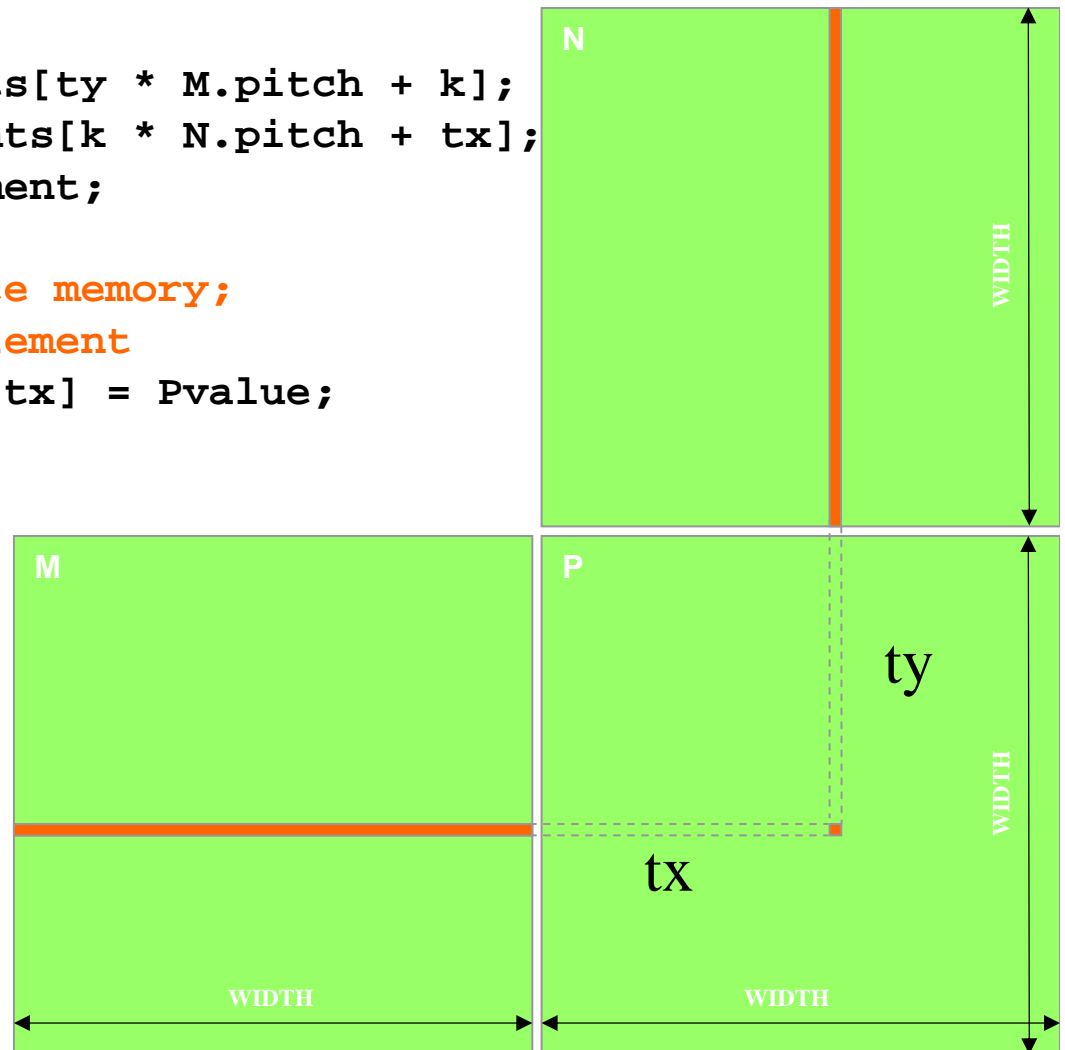
    ...
}
```

Step 4: Matrix Multiplication

Device-Side Kernel Function (cont.)

...

```
for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * blockDim.x + tx] = Pvalue;
}
```



Step 5: Some Loose Ends

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void*)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Step 5: Some Loose Ends (cont.)

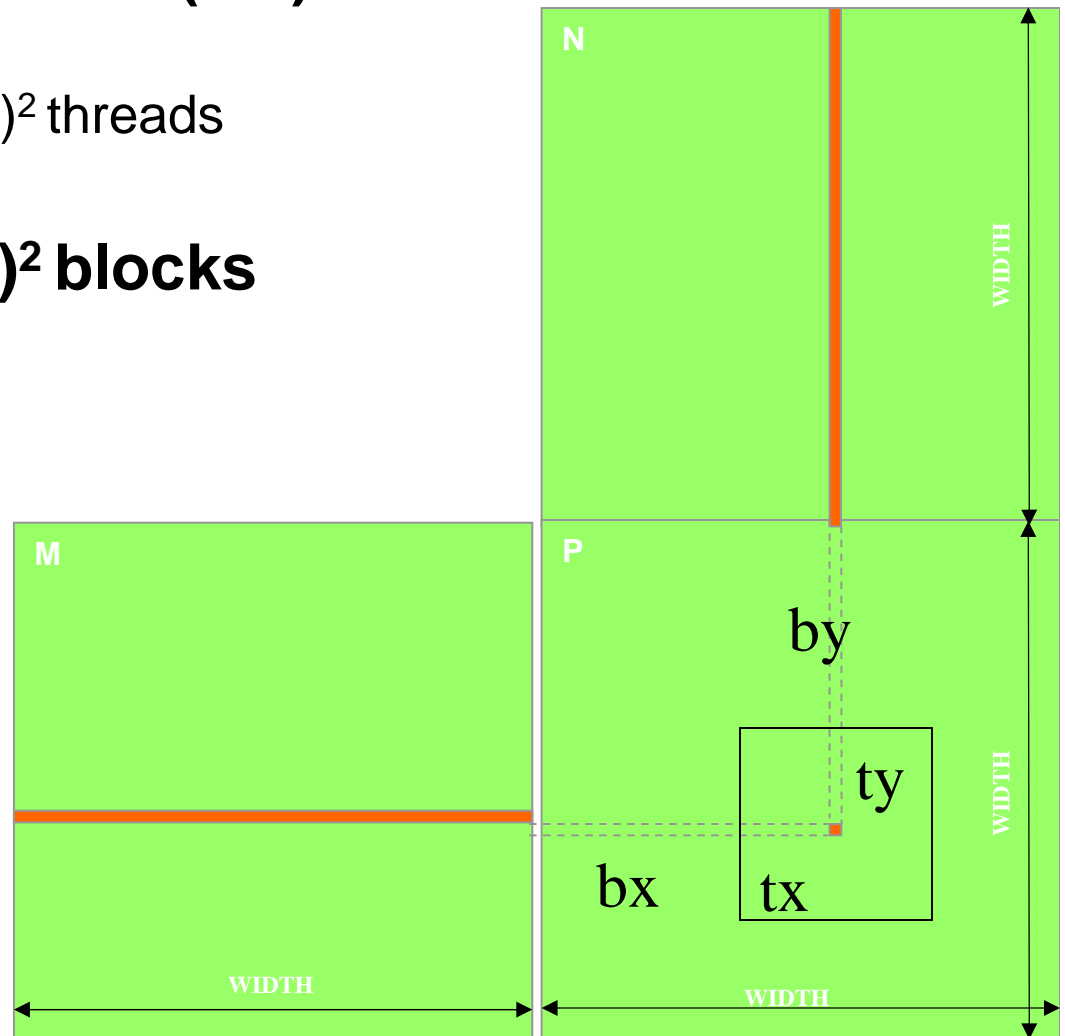
```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size,
               cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size,
               cudaMemcpyDeviceToHost);
}
```

Step 6: Handling Arbitrary Sized Square Matrices

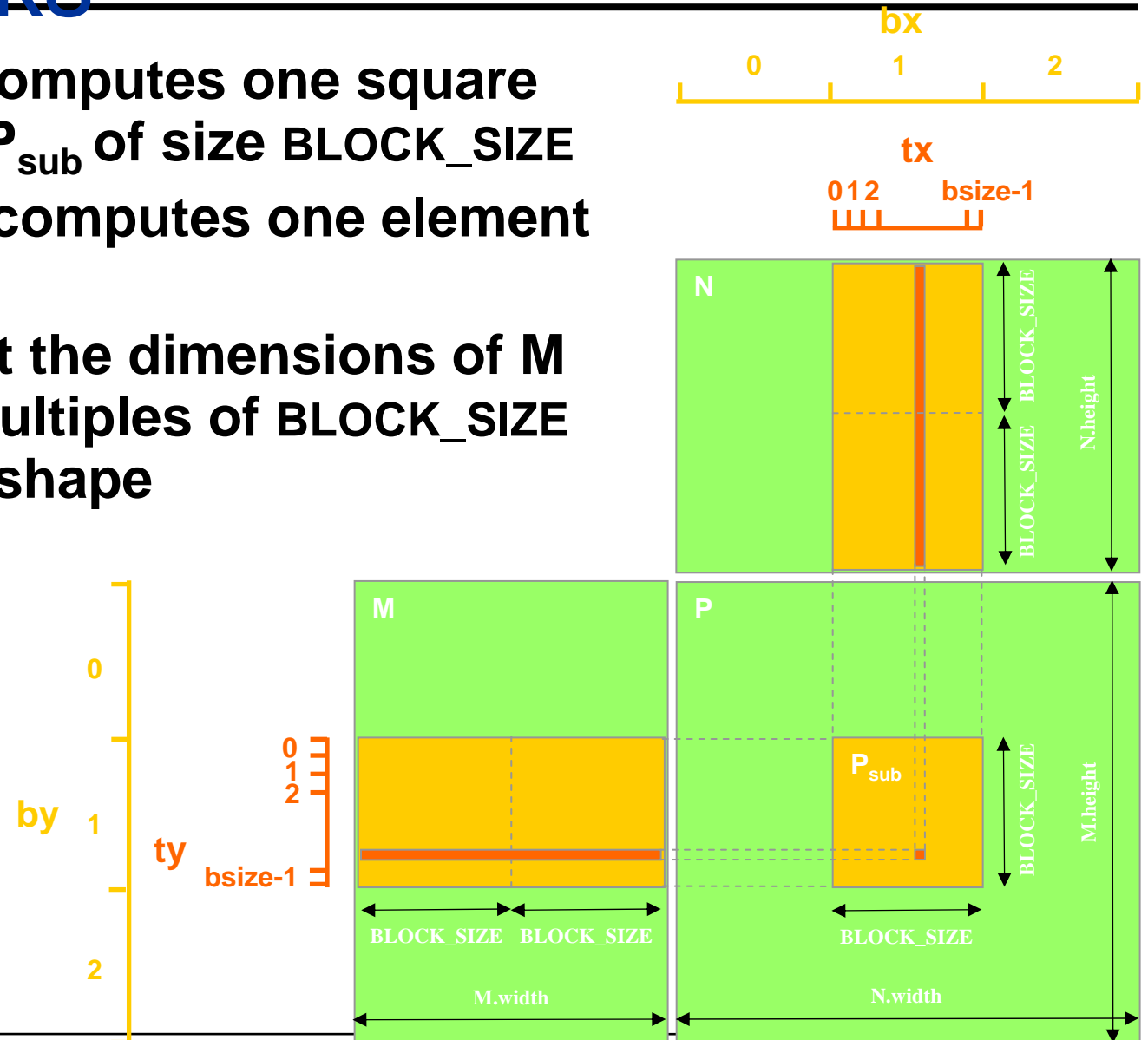
- Have each 2D thread block to compute a $(\text{BLOCK_WIDTH})^2$ sub-matrix (tile) of the result matrix
 - Each has $(\text{BLOCK_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH}/\text{BLOCK_WIDTH})^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!



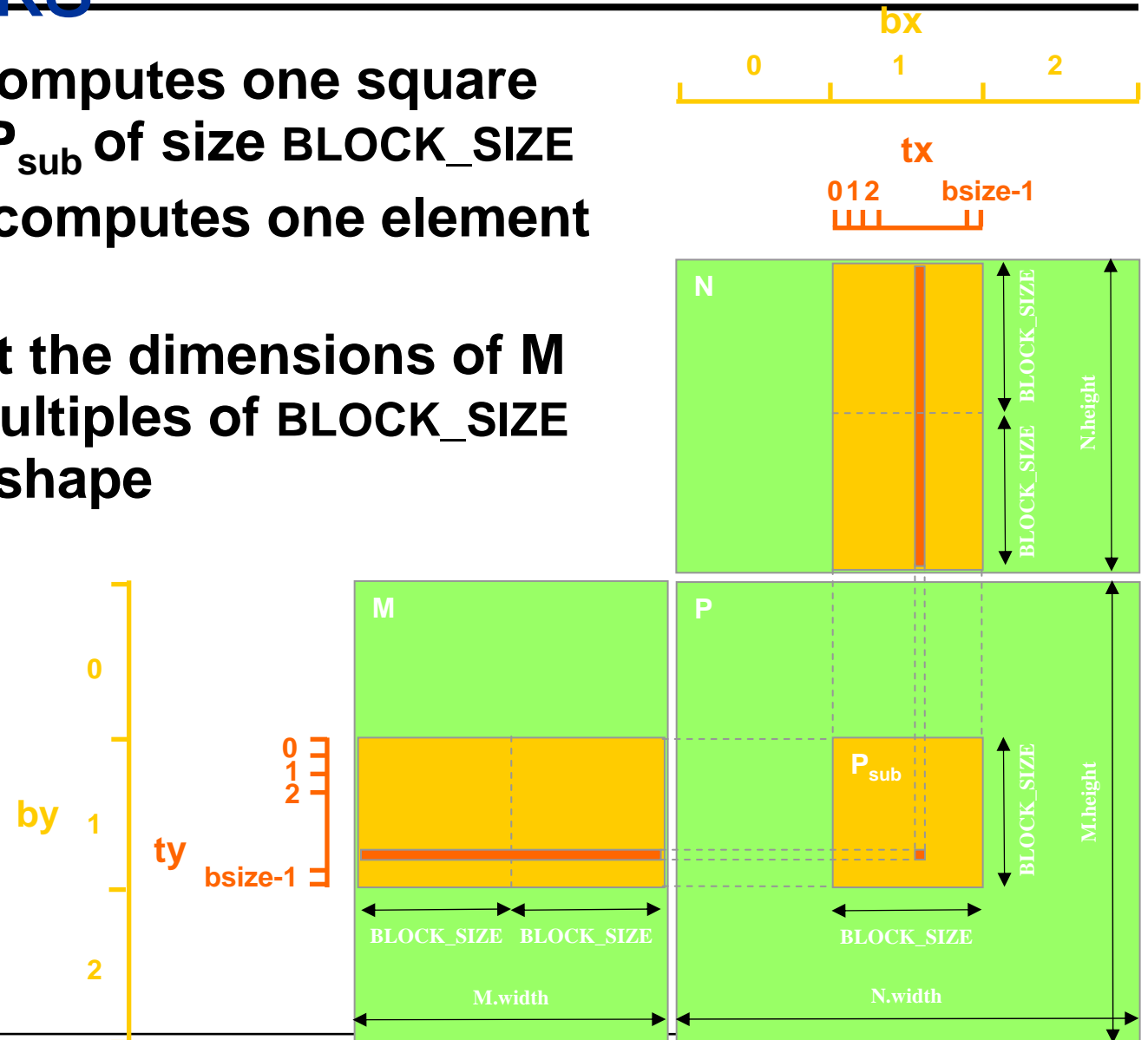
Multiply Using Several Blocks

- One **block** computes one square sub-matrix P_{sub} of size `BLOCK_SIZE`
- One **thread** computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of `BLOCK_SIZE and square shape`



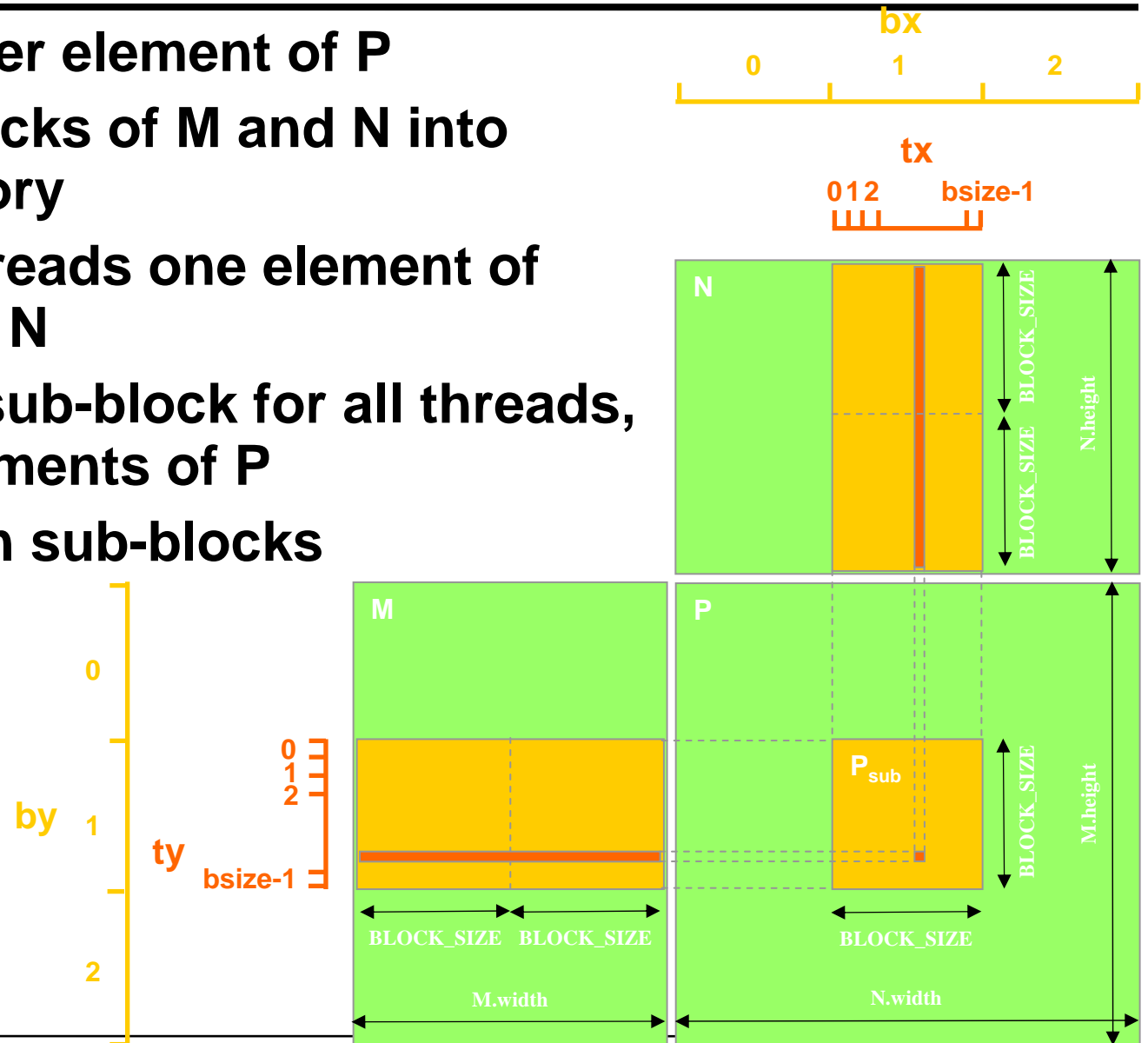
Multiply Using Several Blocks

- One **block** computes one square sub-matrix P_{sub} of size `BLOCK_SIZE`
- One **thread** computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of `BLOCK_SIZE and square shape`



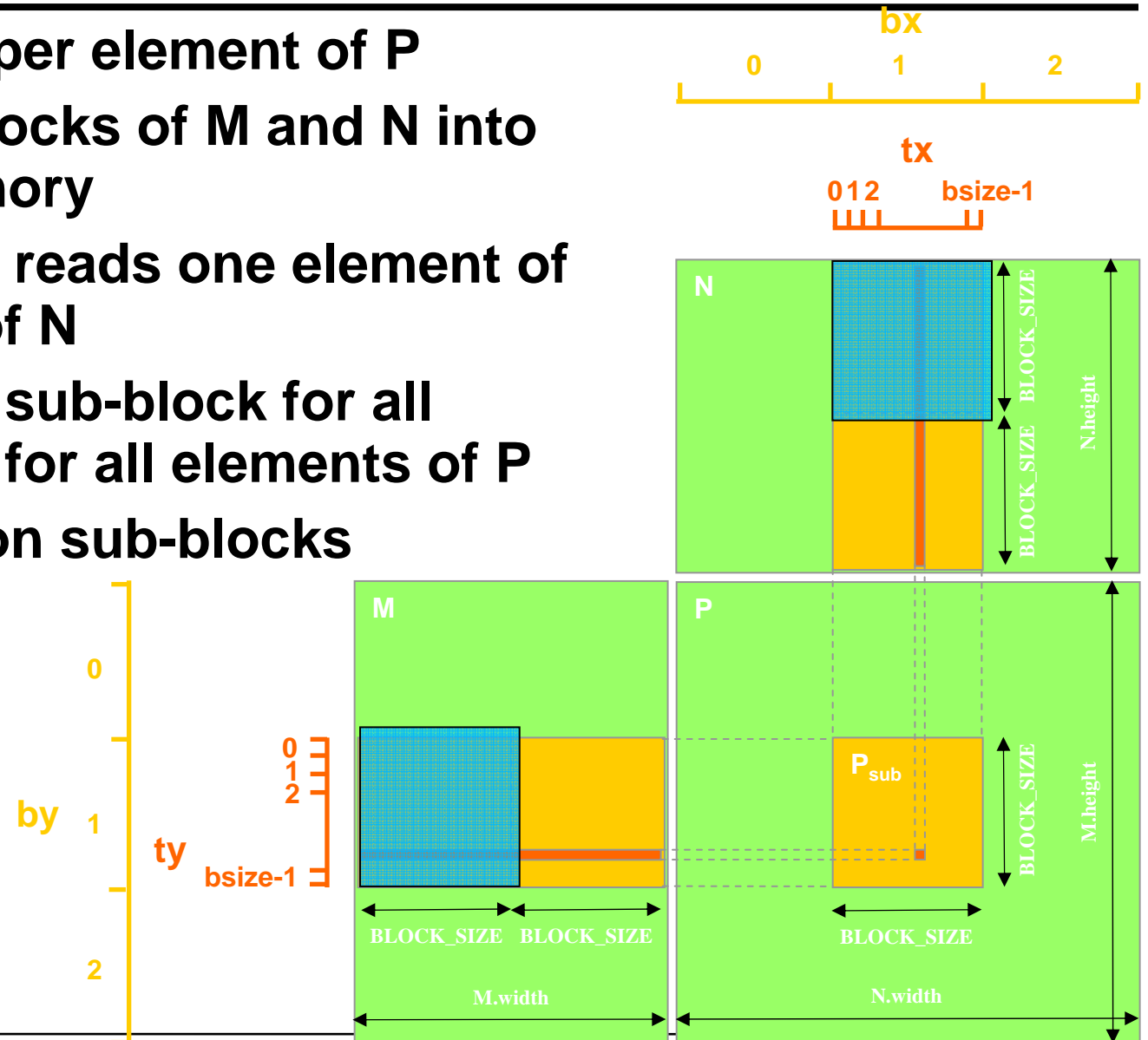
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



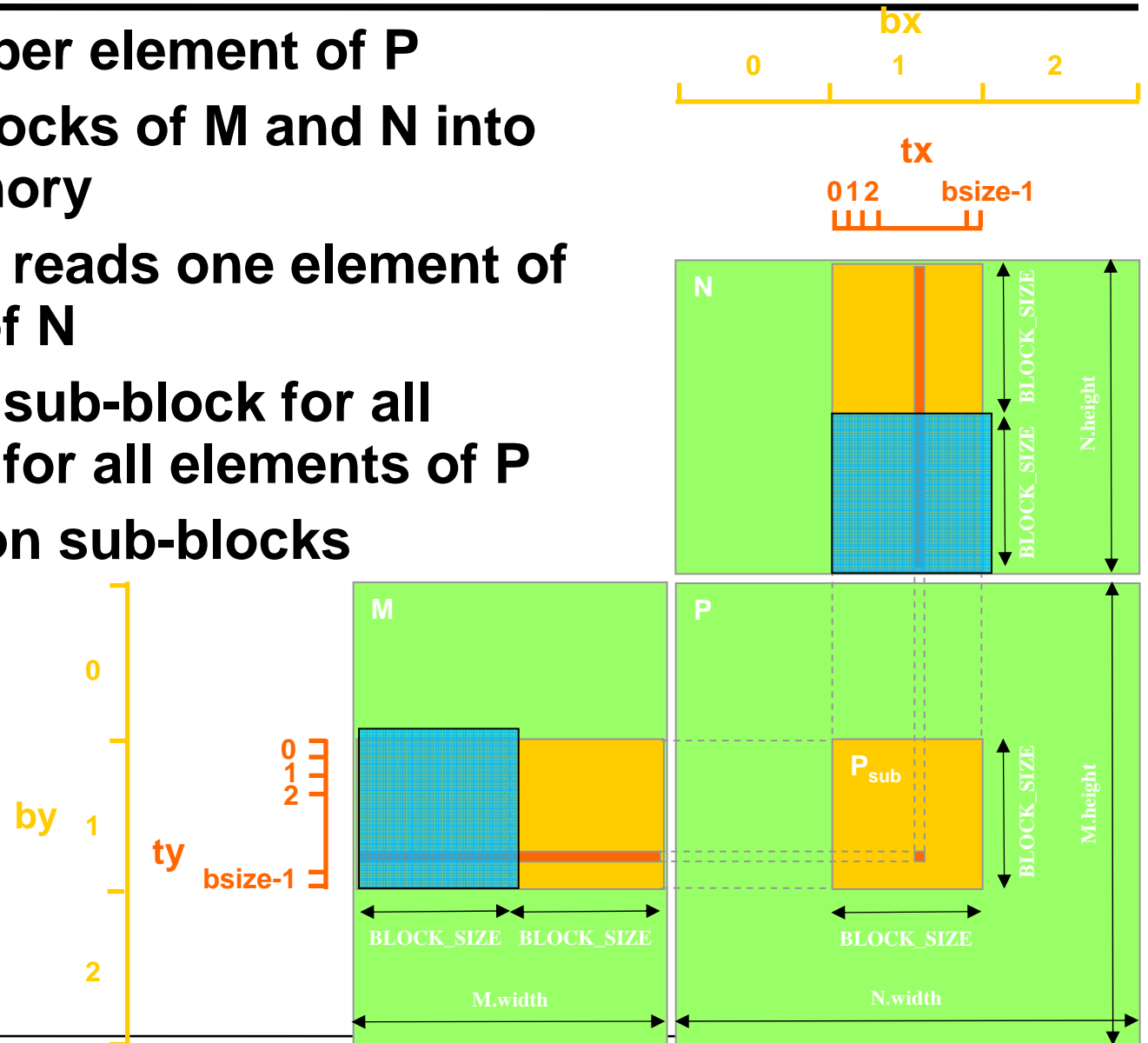
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



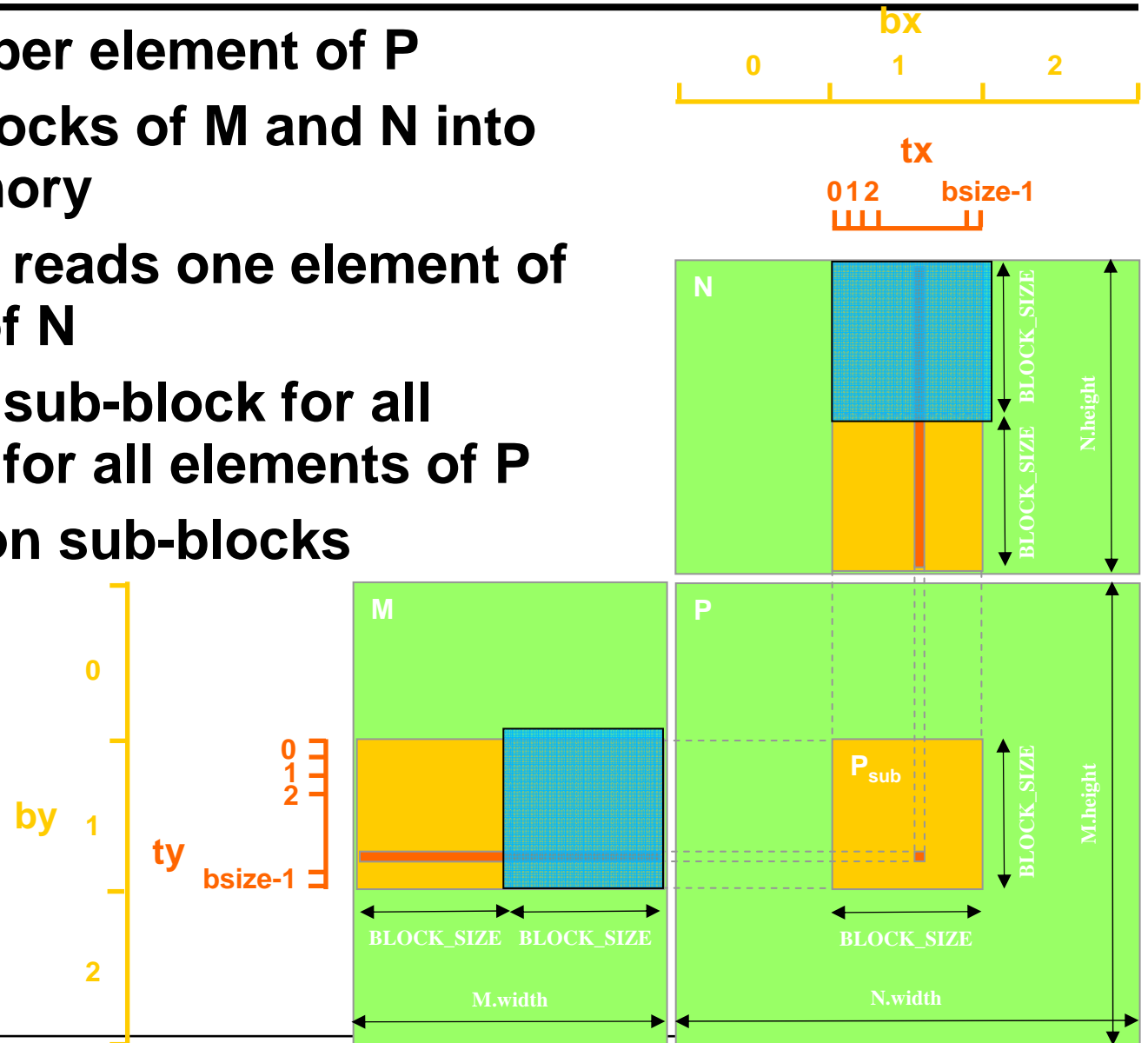
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



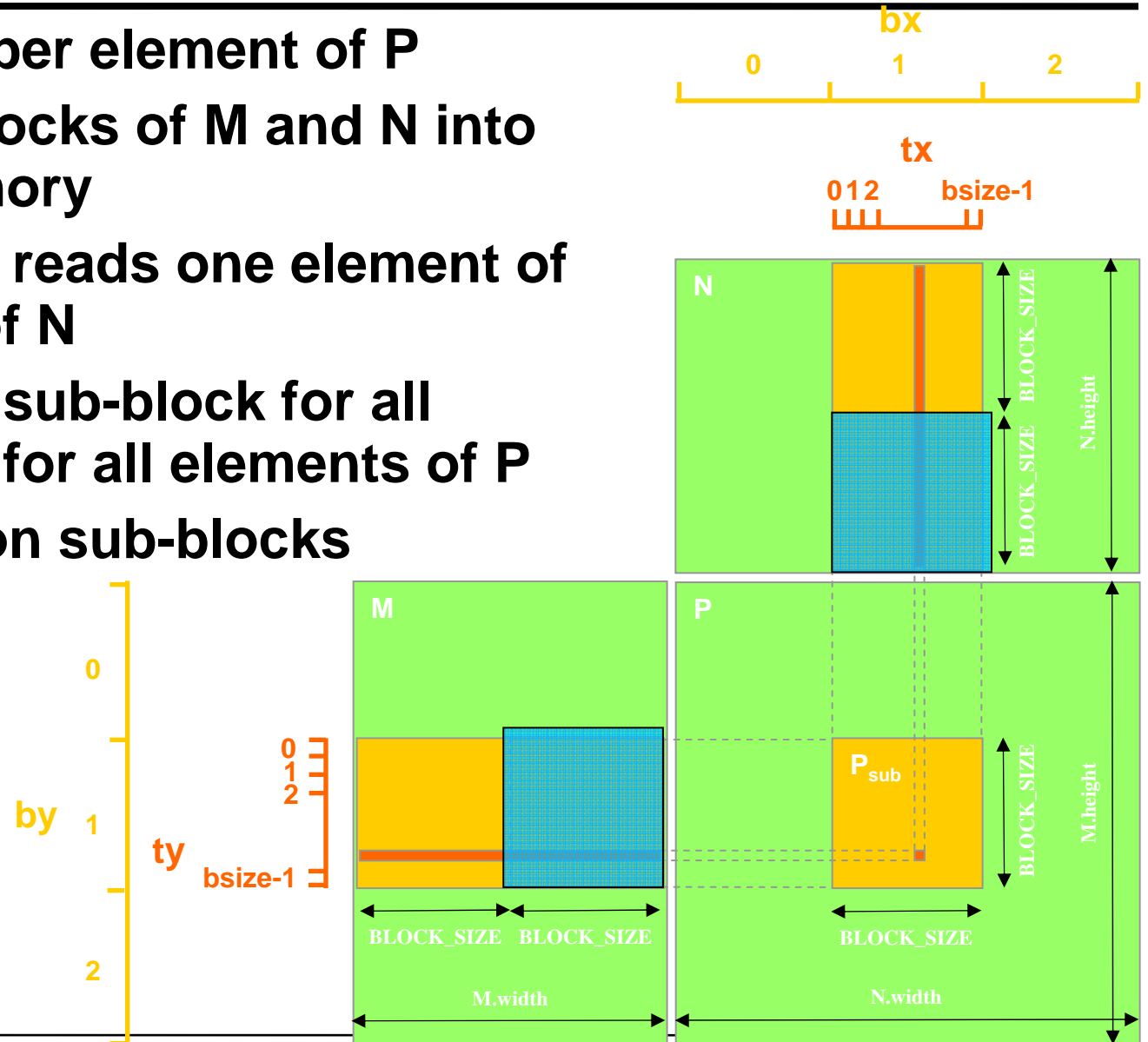
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Matrix Multiplication Kernel with Shared Mem

```
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x; int by = blockIdx.y; //Block index
    int tx = threadIdx.x; int ty = threadIdx.y; // Thread index

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
```

```
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory to shared
        // memory; each thread loads one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        __syncthreads(); // to make sure the matrices are loaded

        // Multiply the two matrices together; each thread
        // computes one element of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Make sure that the preceding computation is done
        // before loading two new sub-matrices of A and B
        __syncthreads();
    }
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Matrix Multiplication Kernel with Shared Mem

```
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
```

```
{
    int bx = blockIdx.x; int by = blockIdx.y; //Block index
    int tx = threadIdx.x; int ty = threadIdx.y; // Thread index

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
```

**address
calculations**

```
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory to shared
        // memory; each thread loads one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        __syncthreads(); // to make sure the matrices are loaded

        // Multiply the two matrices together; each thread
        // computes one element of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Make sure that the preceding computation is done
        // before loading two new sub-matrices of A and B
        __syncthreads();
    }
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Matrix Multiplication Kernel with Shared Mem

```

__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x; int by = blockIdx.y; //Block index
    int tx = threadIdx.x; int ty = threadIdx.y; // Thread index

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd  = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix

```

```

for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {

```

```

// Declaration of the shared memory array As used to
// store the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

// Declaration of the shared memory array Bs used to
// store the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

// Load the matrices from device memory to shared
// memory; each thread loads one element of each matrix
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

__syncthreads(); // to make sure the matrices are loaded

```

**shared memory
upload**

```

// Multiply the two matrices together; each thread
// computes one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

```

```

// Make sure that the preceding computation is done
// before loading two new sub-matrices of A and B
__syncthreads();

```

```

}
// Write the block sub-matrix to device memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}

```

Matrix Multiplication Kernel with Shared Mem

```
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x; int by = blockIdx.y; //Block index
    int tx = threadIdx.x; int ty = threadIdx.y; // Thread index

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
```

```
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory to shared
        // memory; each thread loads one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        __syncthreads(); // to make sure the matrices are loaded

        // Multiply the two matrices together; each thread
        // computes one element of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Make sure that the actual computation is done
        // before loading two new sub-matrices of A and B
        __syncthreads(); computation
    }
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Matrix Multiplication Kernel with Shared Mem

```
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x; int by = blockIdx.y; //Block index
    int tx = threadIdx.x; int ty = threadIdx.y; // Thread index

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
```

```
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory to shared
        // memory; each thread loads one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        __syncthreads(); // to make sure the matrices are loaded

        // Multiply the two matrices together; each thread
        // computes one element of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Make sure that the preceding computation is done
        // before loading two new sub-matrices of A and B
        __syncthreads();
    }
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Control Flow Instructions

- **Main performance concern with branching is divergence**
 - Threads within a single warp take different paths
 - Different execution paths are serialized in G80
 - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- **A common case: avoid divergence when branch condition is a function of thread ID**
 - Example with divergence:
 - `If (threadIdx.x > 2) { }`
 - This creates two different control paths for threads in a block
 - Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp
 - Example without divergence:
 - `If (threadIdx.x / WARP_SIZE > 2) { }`
 - Also creates two different control paths for threads in a block
 - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Memory Access Patterns

Shared Memory Bank Conflicts

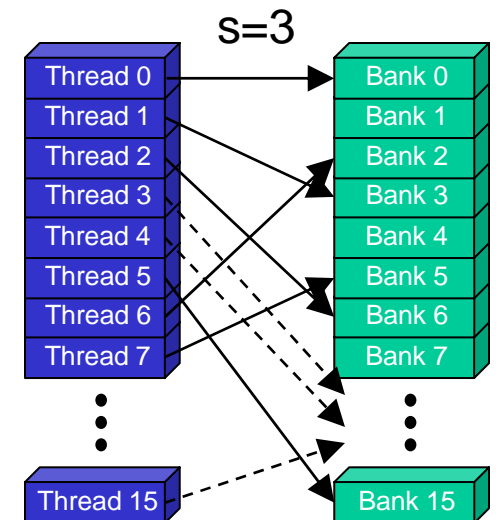
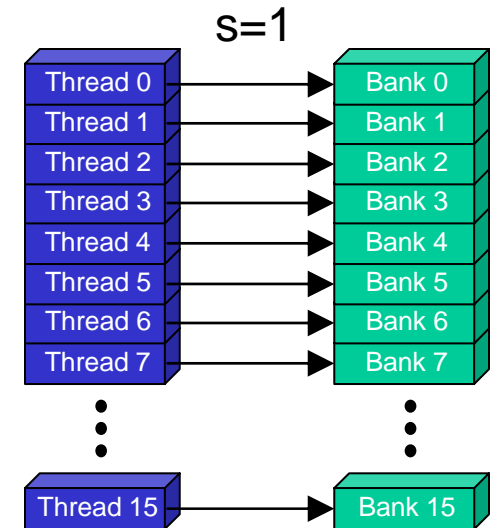
- **Shared memory is as fast as registers if there are no bank conflicts**
- **The fast case:**
 - If all threads of a half-warp access different banks, there is no bank conflict
 - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- **The slow case:**
 - Bank Conflict: multiple threads in the same half-warp access the same bank
 - Must serialize the accesses
 - Cost = max # of simultaneous accesses to a single bank

Linear Addressing

- **Given:**

```
__shared__ float shared[256];  
float foo =  
    shared[baseIndex + s * threadIdx.x];
```

- **This is only bank-conflict-free if s shares no common factors with the number of banks**
 - 16 on G80, so s must be **odd**



Data Types and Bank Conflicts

- This has no conflicts if type of shared is 32-bits:

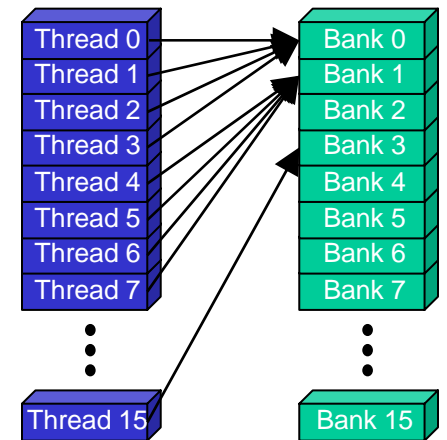
```
foo = shared[baseIndex + threadIdx.x]
```

- But not if the data type is smaller

- 4-way bank conflicts:

```
__shared__ char shared[];
```

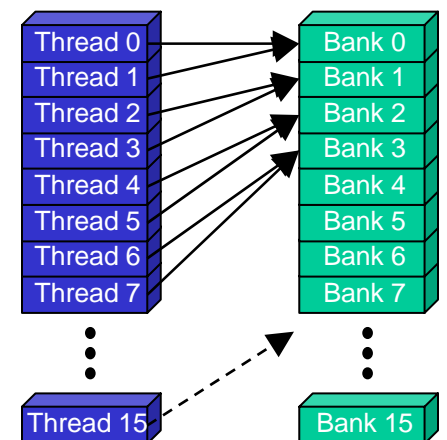
```
foo = shared[baseIndex + threadIdx.x];
```



- 2-way bank conflicts:

```
__shared__ short shared[];
```

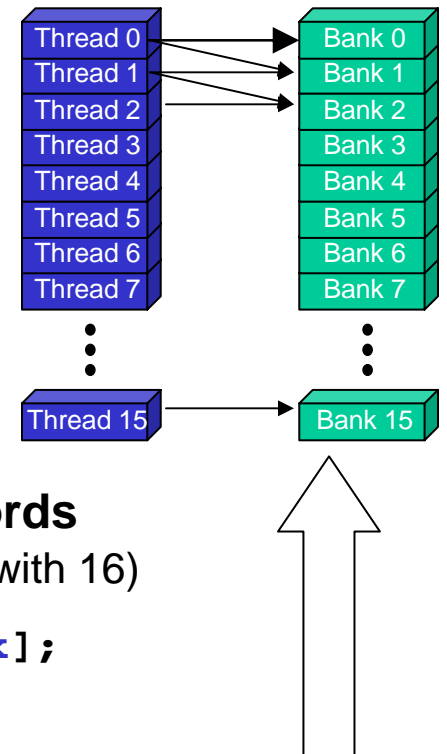
```
foo = shared[baseIndex + threadIdx.x];
```



Structs and Bank Conflicts

- **Struct assignments compile into as many memory accesses as there are struct members:**

```
struct vector { float x, y, z; };  
struct myType {  
    float f;  
    int c;  
};  
__shared__ struct vector vectors[64];  
__shared__ struct myType myTypes[64];
```



- **This has no bank conflicts for vector; struct size is 3 words**
 - 3 accesses per thread, contiguous banks (no common factor with 16)

```
struct vector v = vectors[baseIndex + threadIdx.x];
```

- **This has 2-way bank conflicts for myType;**
(each bank will be accessed by 2 threads simultaneously)

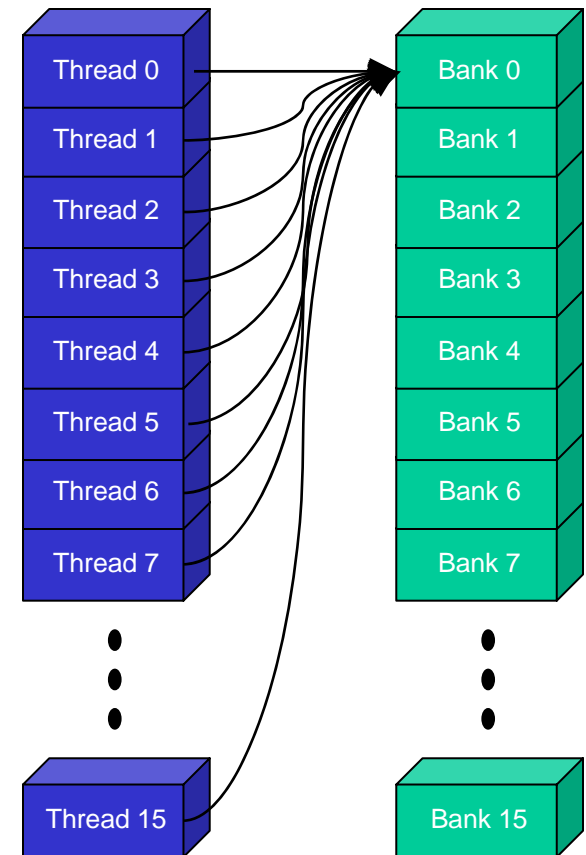
```
struct myType m = myTypes[baseIndex + threadIdx.x];
```

Broadcast on Shared Memory

- Each thread loads the same element – no bank conflict

```
x = shared[0];
```

- Will be resolved implicitly



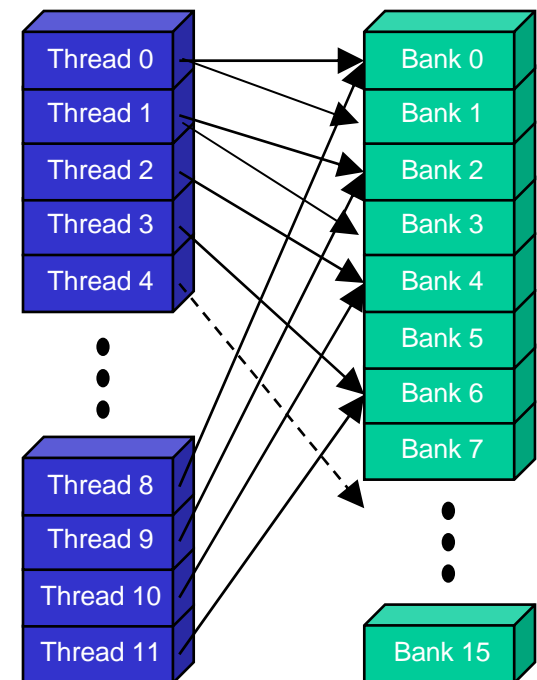
Common Array Bank Conflict Patterns

1D

- Each thread loads 2 elements into shared mem:
 - 2-way-interleaved loads result in 2-way bank conflicts:

```
int tid = threadIdx.x;  
shared[2*tid] = global[2*tid];  
shared[2*tid+1] = global[2*tid+1];
```

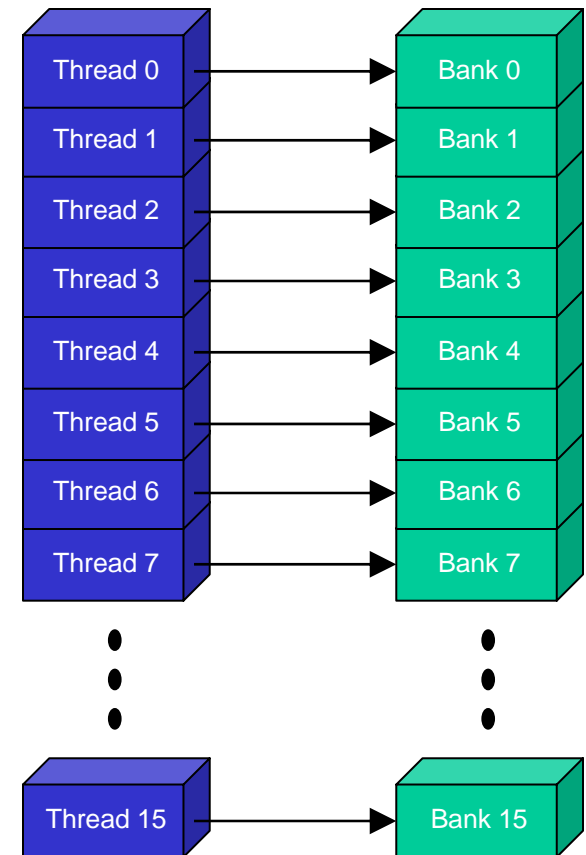
- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.
 - Not in shared memory usage where there is no cache line effects but banking effects



A Better Array Access Pattern

- Each thread loads one element in every consecutive group of `blockDim.x` elements.

```
shared[tid] = global[tid];  
shared[tid + blockDim.x] =  
    global[tid + blockDim.x];
```



Global Memory Access

- **Pretty much the same as for the shared memory**
- **no broadcast though**
- **always try to load continuous chunks of 32bit**
 - hides latencies
- **continuous loading of**
 - `float[]` is faster than loading `float2[]`
 - `float2[]` is faster than loading `float4[]`
 - (performance drops down close to random access!!!!)
- **don't! (4-way bank conflict)**

```
__shared__ float4 vector[64];
vector[threadIdx.x] = globMem[threadIdx.x];
```
- **but**

```
__shared__ float vector[4*64];
for (int i = 0; i < 4; ++i)
    vector[i*blockDim.x+threadIdx.x] =
        (float*)globMem[i*blockDim.x+threadIdx.x];
```

Shared Memory Upload

- **The role of a thread might be different for**
 - Loading into shared memory
 - And accessing shared memory

Texture Memory

Texture Memory

- **Cached**, potentially exhibiting higher bandwidth if there is locality in the texture fetches;
- They are not subject to the constraints on memory access patterns that global or constant memory reads must respect to get good performance
- The latency of addressing calculations is hidden better, possibly improving performance for applications that perform random accesses to the data
- No penalty when accessing float4
- Optional
 - 8-bit and 16-bit integer input data may be optionally converted to 32-bit floatingpoint
 - Packed data may be broadcast to separate variables in a single operation;
 - values in the range [0.0, 1.0] or [-1.0, 1.0]
 - texture filtering
 - address modes, e.g. wrapping / texture borders

1D Access

- **Access to linear Cuda memory**

```
float4* pos; cudaMalloc( (void**)&pos, x*sizeof(float4) );
```

- **Texture reference**

- type
- access/filtering mode

```
// global texture reference
```

```
texture< float4, 1, cudaReadModeElementType> texPos;
```

- **Bind to linear array**

```
cudaBindTexture(0, texPos, pos, x*sizeof(float4));  
cudaUnbindTexture(texPos);
```

- **Within kernel**

```
float4 pa1 = tex1Dfetch( texPos, threadIdx.x);
```

- **Writing to a texture that is currently read by some threads is undefined!!!**

2D Access

- **Optimized for 2D / 3D locality**

```
texture< float4, 2, cudaReadModeElementType> texImg;
```

- **Requires binding to special *Array* memory – special memory layout**

```
cudaChannelFormatDesc floatTex =  
cudaCreateChannelDesc<float4>();  
float4* src;  
cudaArray* img;  
cudaMallocArray( &img, &floatTex, w, h);  
cudaMemcpyToArray(img, 0, 0, src, w*h*sizeof(float4),  
    cudaMemcpyHostToDevice);  
cudaBindTextureToArray( texImg, img, floatTex) );  
cudaUnbindTexture(texImg);
```

2D Access

- **Within kernel**

```
float4 r = tex2D( texImg, x +xoff, y+yoff);
```

- **Pros**

- optimized for 2D locality (optimized memory layout / spacefilling curve)

- **Cons**

- If the result of some kernel should be used as 2D texture `cudaMemcpyToArray` is required
- You cannot write to a texture which is currently read from

Accelerating Upload/Download

Up/Download of Data

- **standard memcpy**

```
float* hostData = new float[100];  
cudaMemcpy(devData, hostData, size,  
           cudaMemcpyHostToDevice);
```

- **memcpy from pagelocked memory
(faster DMA transfer)**

```
float* hostData;  
cudaMallocHost(hostData, 100);  
cudaMemcpy(devData, hostData, size, cudaMemcpyHostToDevice);
```

upload

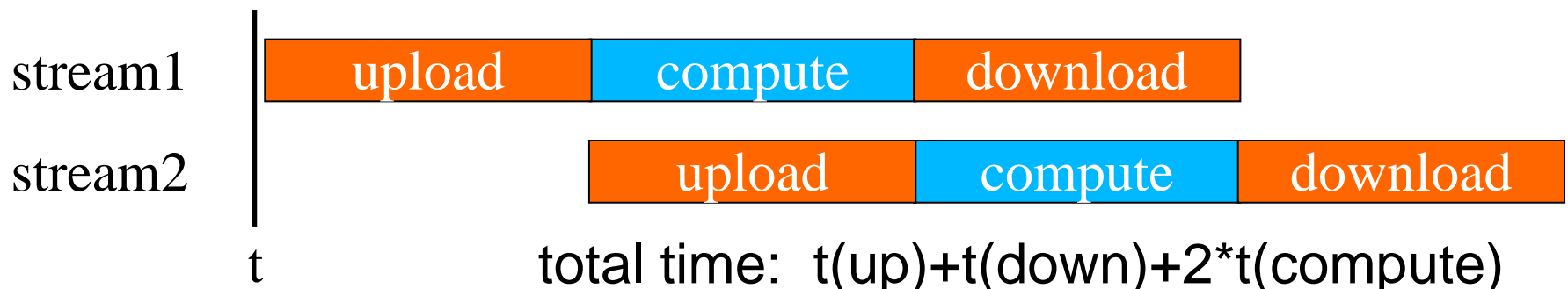
compute

download

Up/Download of Data - Streams

- asynchronous transfer interlinked with kernel execution
- streams (from pagelocked memory)

```
cudaStream_t stream[2];
cudaStreamCreate(&(stream[0])); cudaStreamCreate(&(stream[1]));
cudaMemcpyAsync(devData, hostData, size/2,
    cudaMemcpyHostToDevice, stream[0]);
mykernel<<< block, thread, 0, stream[0] >>>(params);
cudaMemcpyAsync(devData +size/2, hostData + size/2, size/2,
    cudaMemcpyHostToDevice, stream[1]);
mykernel<<< block, thread, 0, stream[1] >>>(params);
cudaStreamSynchronize(stream[0]);
cudaMemcpyAsync(hostData, devData, size/2, cudaMemcpyDeviceToHost,
    stream[0]);
cudaStreamSynchronize(stream[1]);
cudaMemcpyAsync(hostData, devData, size/2, cudaMemcpyDeviceToHost,
    stream[1]);
```



Reduction – Version1

Example: Parallel Reduction

- **Given an array of values, “reduce” them to a single value in parallel**
- **Examples**
 - sum reduction: sum of all values in the array
 - Max reduction: maximum of all values in the array
- **Typically parallel implementation:**
 - Recursively halve # threads, add two values per thread
 - Takes $\log(n)$ steps for n elements, requires $n/2$ threads

A Vector Reduction Example

- **Assume an in-place reduction using shared memory**
 - The original vector is in device global memory
 - The shared memory used to hold a partial sum vector
 - Each iteration brings the partial sum vector closer to the final sum
 - The final solution will be in element 0

A Simple Implementation

- Assume we have already loaded array into

```
__shared__ float partialSum[];
```

```
unsigned int t = threadIdx.x;
```

```
// loop log(n) times
```

```
for (unsigned int stride = 1;
```

```
    stride < blockDim.x; stride *= 2)
```

```
{
```

```
    // make sure the sum of the previous iteration
```

```
    // is available
```

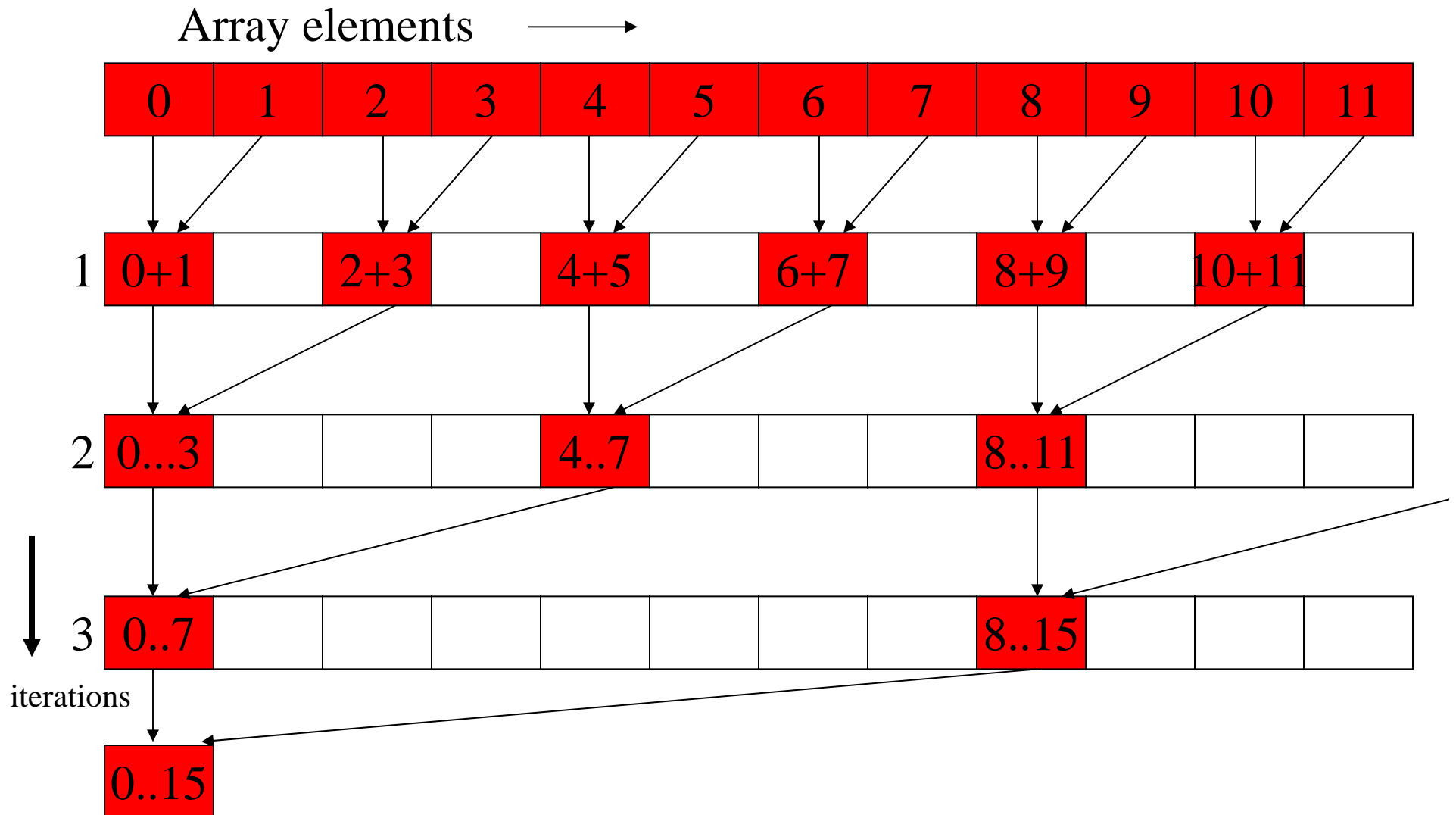
```
    __syncthreads();
```

```
    if (t % (2*stride) == 0)
```

```
        partialSum[t] += partialSum[t+stride];
```

```
}
```

Vector Reduction



Typical Parallel Programming Pattern

- $\log(n)$ steps

