

COL380

Introduction to  
Parallel & Distributed Programming

# CUDA Memory Model Overview

- Global memory

`cudaMalloc`, `cudaFree`, `__device`, Unified Memory

- Host ↔ Device data communication
- Visible to all threads, persistent across kernels
- Long latency
- Through L1 and L2

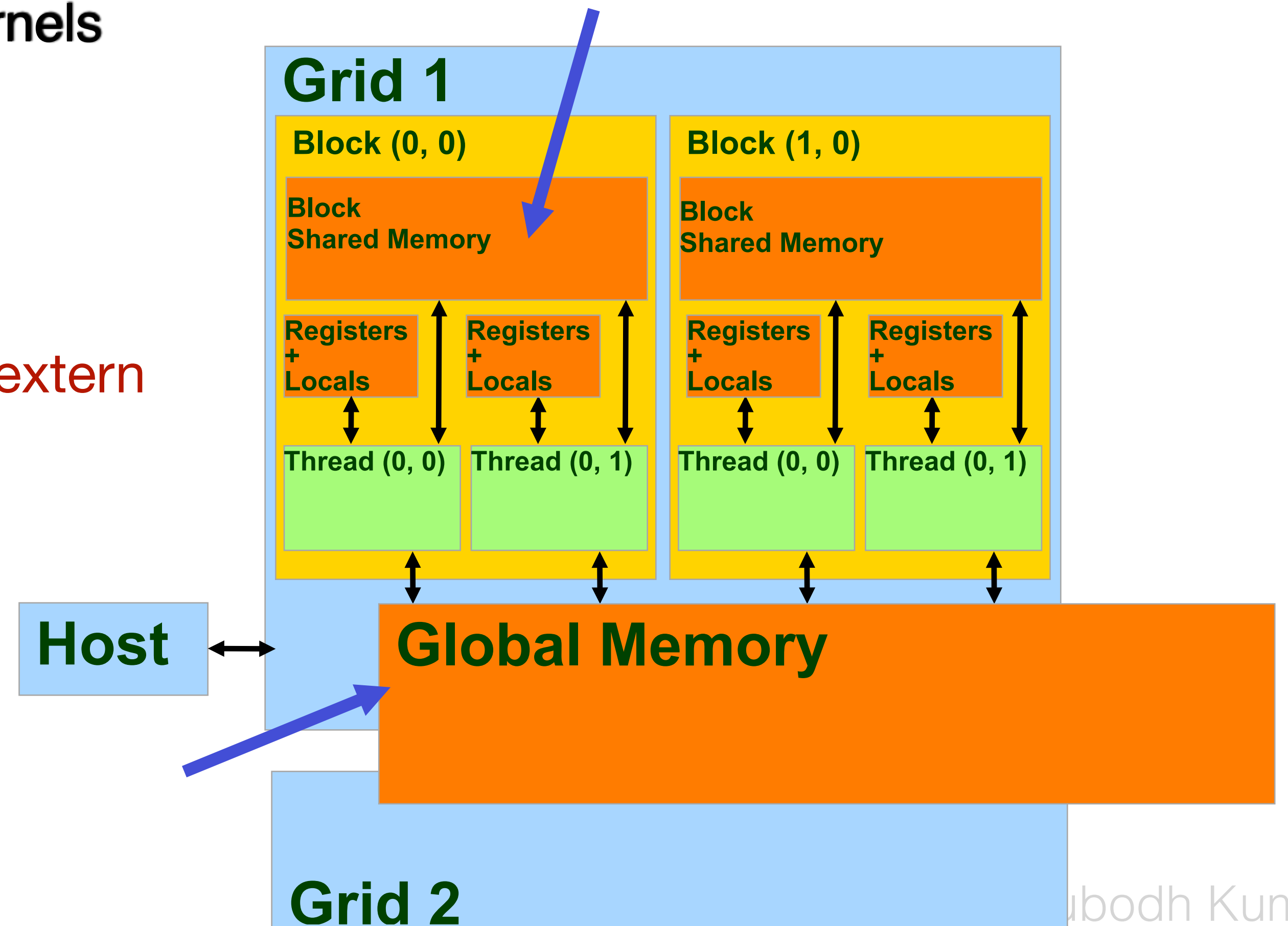
- Shared Memory

`__shared`, Block-extern

- Fast memory (user-managed L1)
- Shared across block, Lifetime of block

- Other memory segments

- Constant and Texture
  - Read-only, cached



- Allocation and Transfer can be both Explicit and Implicit
- Explicit Allocation
  - `cudaMalloc(..)`
- Implicit Allocation
  - declare variables `__managed__`
  - Kernel Launch Arguments
- Explicit Transfer
  - `cudaMemcpy(..)`, `cudaMemcpyToSymbol(..)`
- Implicit Transfer
  - On page-fault
  - Kernel Arguments

# Device Memory Allocation

```
TILE = 64;
float *dM;
int size = TILE * TILE * sizeof(float);
...
cudaMalloc( (void**) &dM, size);
...
cudaFree(dM);
```

Also see:

`cudaHostAlloc(..)`

Page-locked memory allocation  
Can be Mapped on host and device

`cudaMallocManaged(&x, nbytes);`

Unified-memory allocation

`cudaMallocPitch(..), cudaMalloc3D(..)`

2D/3D arrays *\*aligned\** to support  
parallel IO

# Host-Device Data Transfer

```
cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);  
cudaMemcpy(M, dM, size, cudaMemcpyDeviceToHost);
```

Predefined Constants

- Blocking call But see cudaMemcpyAsync().
- M is in host memory allocated regularly
- dM is in device memory

Also see:

```
cudaMemcpy2D(...)  
cudaMemcpy3D(...)  
cudaMemcpyToSymbol(...)
```

## Example (Memcpy)

```
size_t size = N * sizeof(float);  
float* hA = (float*)malloc(size); // Allocate vector @host  
float *dA, *dB, *dC;             // Declare device vectors  
handles  
  
cudaMalloc(&dA, size);             // Allocate vectors @device  
cudaMalloc(&dB, size);             // dA and dB are opaque on CPU  
cudaMemcpy(dA, hA, size, cudaMemcpyDefault); // Copy host->device  
  
// Invoke kernel on GPU  
GPUfunc<<<blocksPerGrid, threadsPerBlock>>>(dA, dB, N);  
  
cudaMemcpy(hB, dB, size, cudaMemcpyDeviceToHost); // device->Host  
  
cudaFree(dA); // Free device memory  
cudaFree(dB);
```



# Managed Memory

```
__device__ __managed__ int N = 65535;    // Managed
float *bothp;
cudaMallocManaged(&bothp, N*sizeof(float)); // Managed

initialize(bothp, N); // initialize bothp on host

Kernel<<<N/1024,1024>>>(bothp); // Launch on GPU
cudaDeviceSynchronize(); // Block until Kernel completes

cpuProcess(bothp) // Use the values computed on GPU
cudaFree(bothp);  // Free memory
```

```
__global__ void Kernel (float *bothp)
{
    int index = blockIdx.x * blockDim.x +
                threadIdx.x;

    if (index < N)
        bothp[index] = f(bothp[index]);
}

__device__ float f(float x) {
    ...
}
```

# CUDA Function Qualifiers


```
__device__ float dSomeName() {}  
__global__ void kSomeName() {}
```

- **\_\_global\_\_** defines a kernel function
  - Must return void
  - called from host, run on device
  - No recursion
- **\_\_device\_\_** are executed and called on device
- **\_\_host\_\_** qualifier also exists
  - **\_\_device\_\_** and **\_\_host\_\_** can be used together



- kernels called with **execution configuration**:

```
__global__ void KernelFunc(...);  
dim3    DimGrid(128, 32);        // Total 4096 thread blocks  
dim3    DimBlock(8, 8, 8);       // 512 threads/block  
size_t  SharedMemBytes = 2048;  // 2KB of shared mem per block  
KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);
```



Allocated on launch

- Calls to a kernel function are **asynchronous**
  - Parameters are passed through shared/constant memory
- Call **cudaDeviceSynchronize()** to block on the kernels in flight

- Use extern

```
extern __shared__ char sharedblock[];
__device__ void func() // or __global__
{
    short* array0 = (short*)sharedblock;
    float* array1 = (float*)&array0[128];
    int* array2 = (int*)&array1[64];
}
```

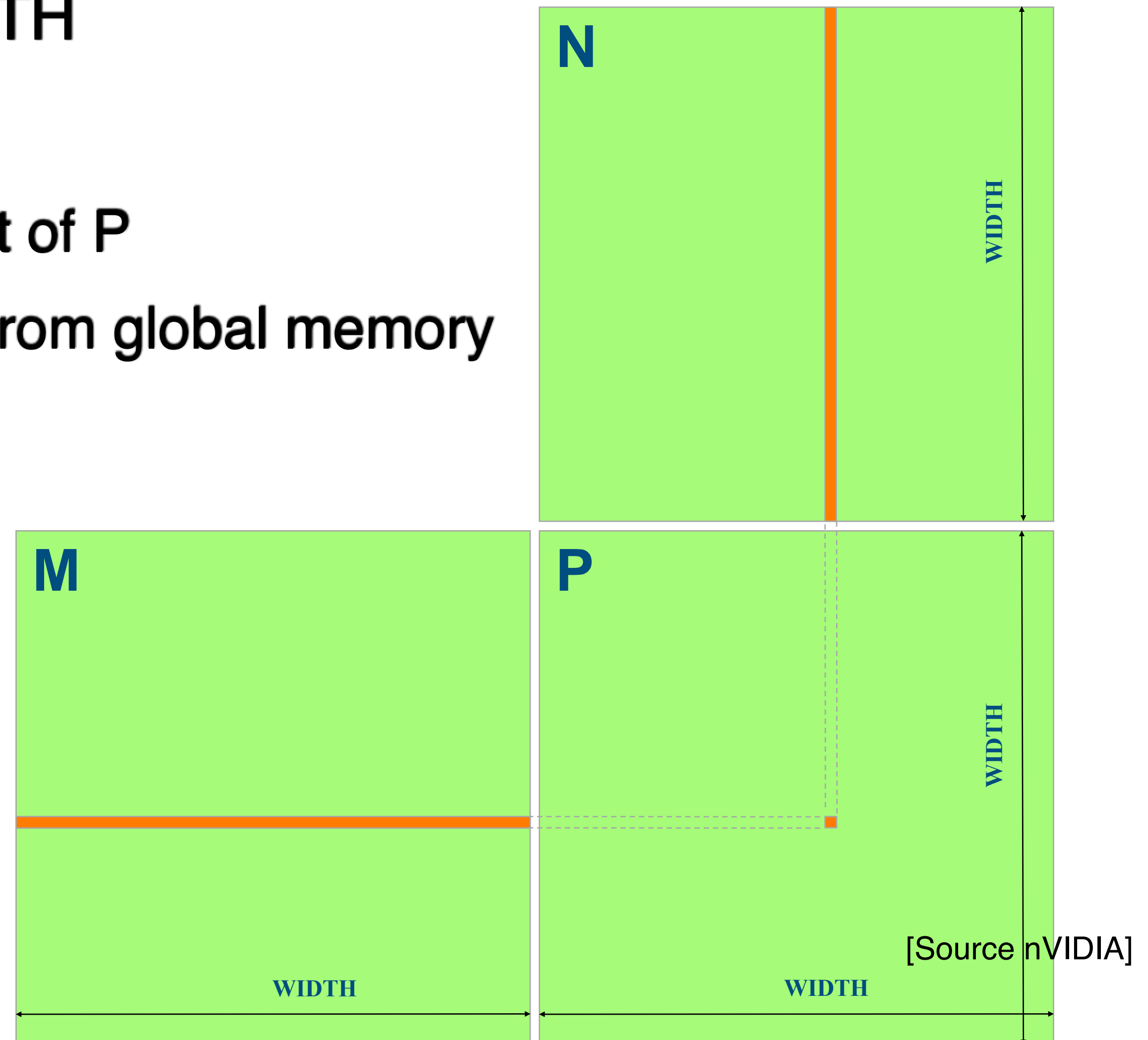
e.g., Matrix  
Multiplication

- Illustrates basic memory/thread management
- Assumes square matrix for simplicity
- No shared memory usage yet
- Local, register usage
- Thread ID usage
- Memory data transfer between host and device

Cuda functions return `cudaError_t`

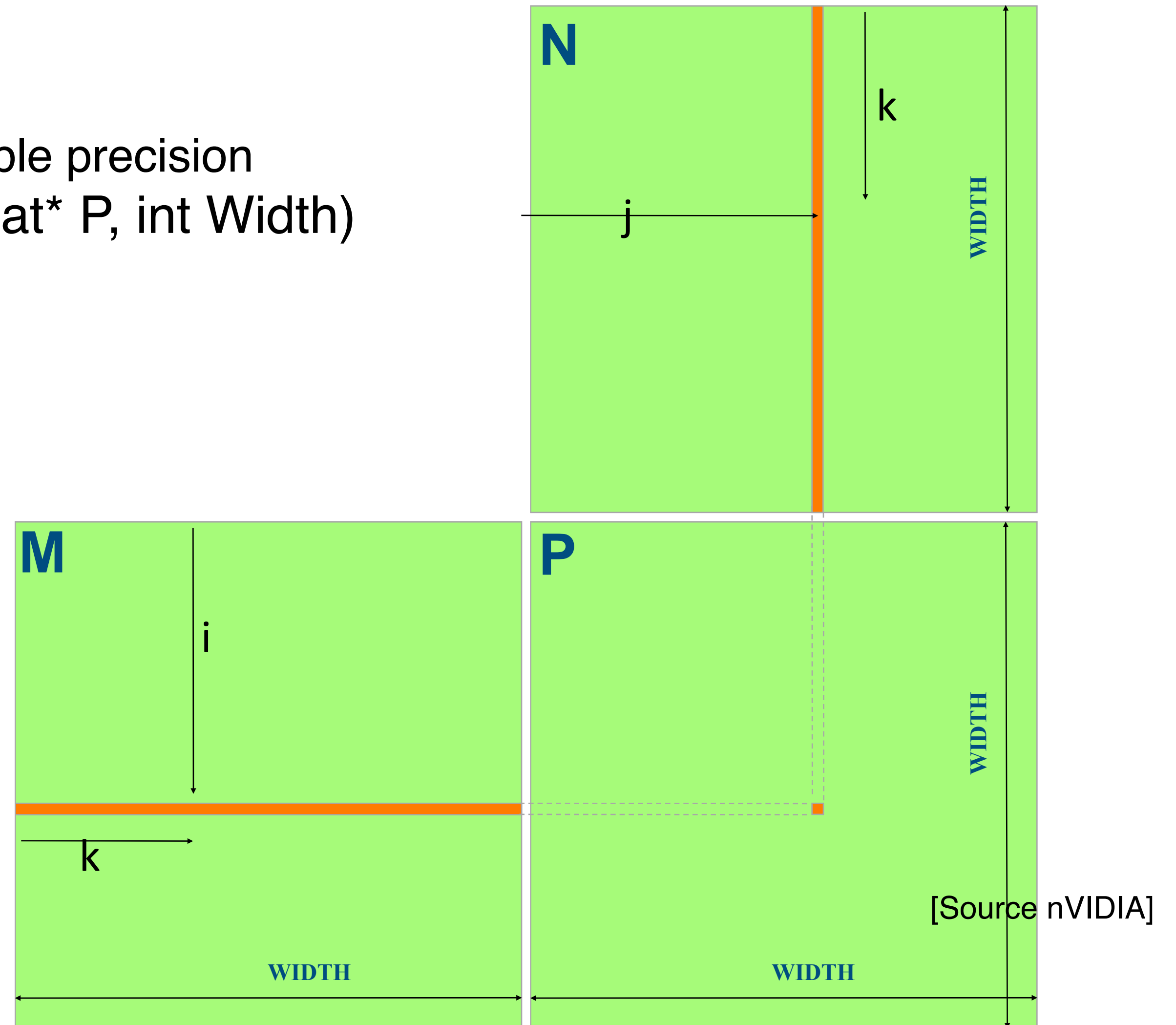
# Square Matrix Multiplication

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



# MATMUL on Host

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



# MATMUL: Matrix Data Transfer

```
void MatrixMulOnDevice(float* M, float* N,
                      float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float *dM, *dN, *dP;
    ...
1. // Allocate and Load M, N to device memory
   cudaMalloc(&dM, size);
   cudaMemcpy(dM, M, size, cudaMemcpyDefault);
   cudaMalloc(&dN, size);
   cudaMemcpy(dN, N, size, cudaMemcpyDefault);
   // Allocate P on the device
   cudaMalloc(&dP, size);
2. // Kernel invocation code (to be shown later)
   ...
3. // Read P from the device
   cudaMemcpy(P, dP, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(dM); cudaFree(dN); cudaFree(dP);
}
```

Blocks until copy is complete

[Example from Kirk & Hwu]



# MATMUL: Matrix Data Transfer

```
void MatrixMulOnDevice(float* M, float* N,
                      float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float *dM, *dN, *dP;

    ...
1. // Allocate and Load M, N to device memory
   cudaMalloc(&dM, size);
   cudaMemcpy(dM, M, size, cudaMemcpyDefault);
   cudaMalloc(&dN, size);
   cudaMemcpy(dN, N, size, cudaMemcpyDefault);
   // Allocate P on the device
   cudaMalloc(&dP, size);
2. // Kernel invocation code (to be shown later)
   ...
3. // Read P from the device
   cudaMemcpy(P, dP, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(dM); cudaFree(dN); cudaFree(dP);
}
```

```
cudaMallocManaged(..);
// invoke Kernel
cudaFree(..);
```

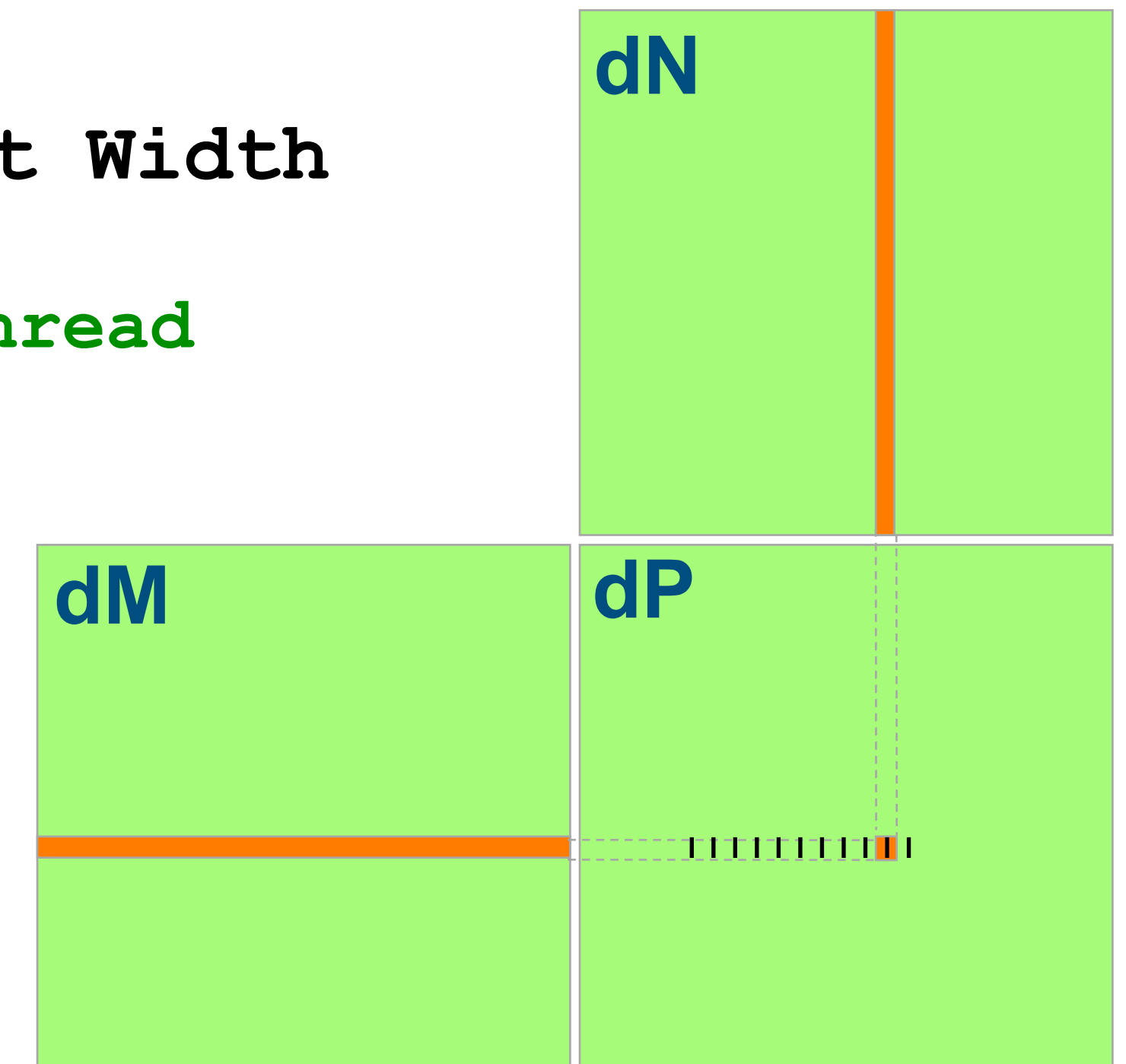
[Example from Kirk & Hwu]

# MATMUL: Kernel Function

// Matrix multiplication kernel – per thread code

```
__global__ void
MatrixMulKernel(float* dM, float* dN, float* dP, int Width
{
    // Pvalue stores the matrix element computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        float Melement = dM[threadIdx.y*Width + k];
        float Nelement = dN[k*Width + threadIdx.x];
        Pvalue += Melement * Nelement;
    }

    dP[threadIdx.y*Width + threadIdx.x] = Pvalue;
}
```



# MATMUL: Matrix Data Transfer

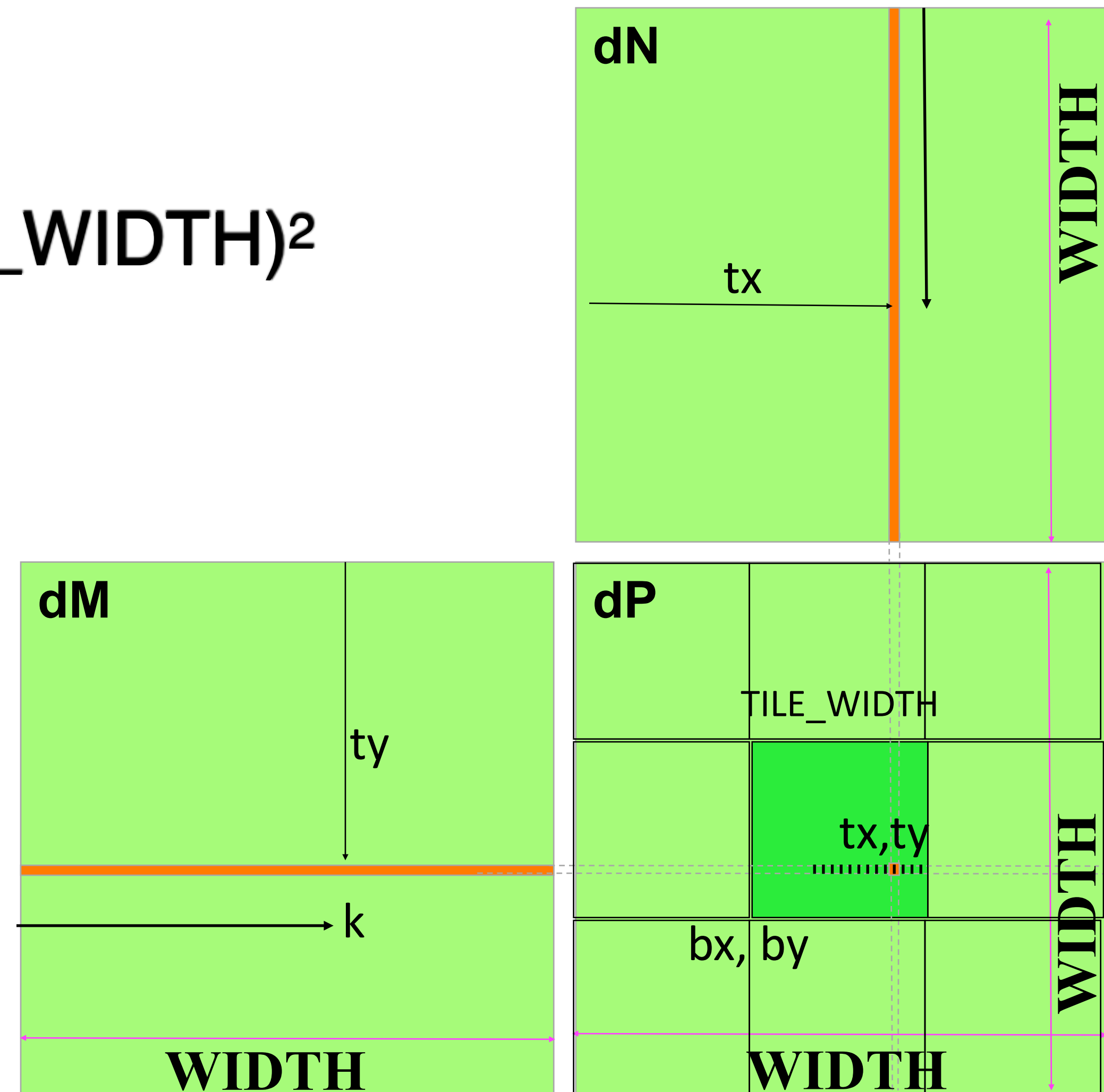
```
void MatrixMulOnDevice(float* M, float* N,
                      float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float *dM, *dN, *dP;
    ...
1. // Allocate and Load M, N to device memory
   cudaMalloc(&dM, size);
   cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);
   cudaMalloc(&dN, size);
   cudaMemcpy(dN, N, size, cudaMemcpyHostToDevice);
   // Allocate P on the device
   cudaMalloc(&dP, size);

2. // Setup the execution configuration
   dim3 dimBlock(Width, Width);
   // Launch the device computation threads!
   MatrixMulKernel<<<1, dimBlock>>>(dM, dN, dP, Width);

3. // Read P from the device
   cudaMemcpy(P, dP, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(dM); cudaFree(dN); cudaFree(dP);
}
```

# MATMUL: Multiple Thread Blocks

- 2D thread block computes a  $(\text{TILE\_WIDTH})^2$  sub-matrix of Pd
  - $(\text{TILE\_WIDTH})^2$  threads/block
- Generate a 2D Grid of  $(\text{WIDTH}/\text{TILE\_WIDTH})^2$  blocks

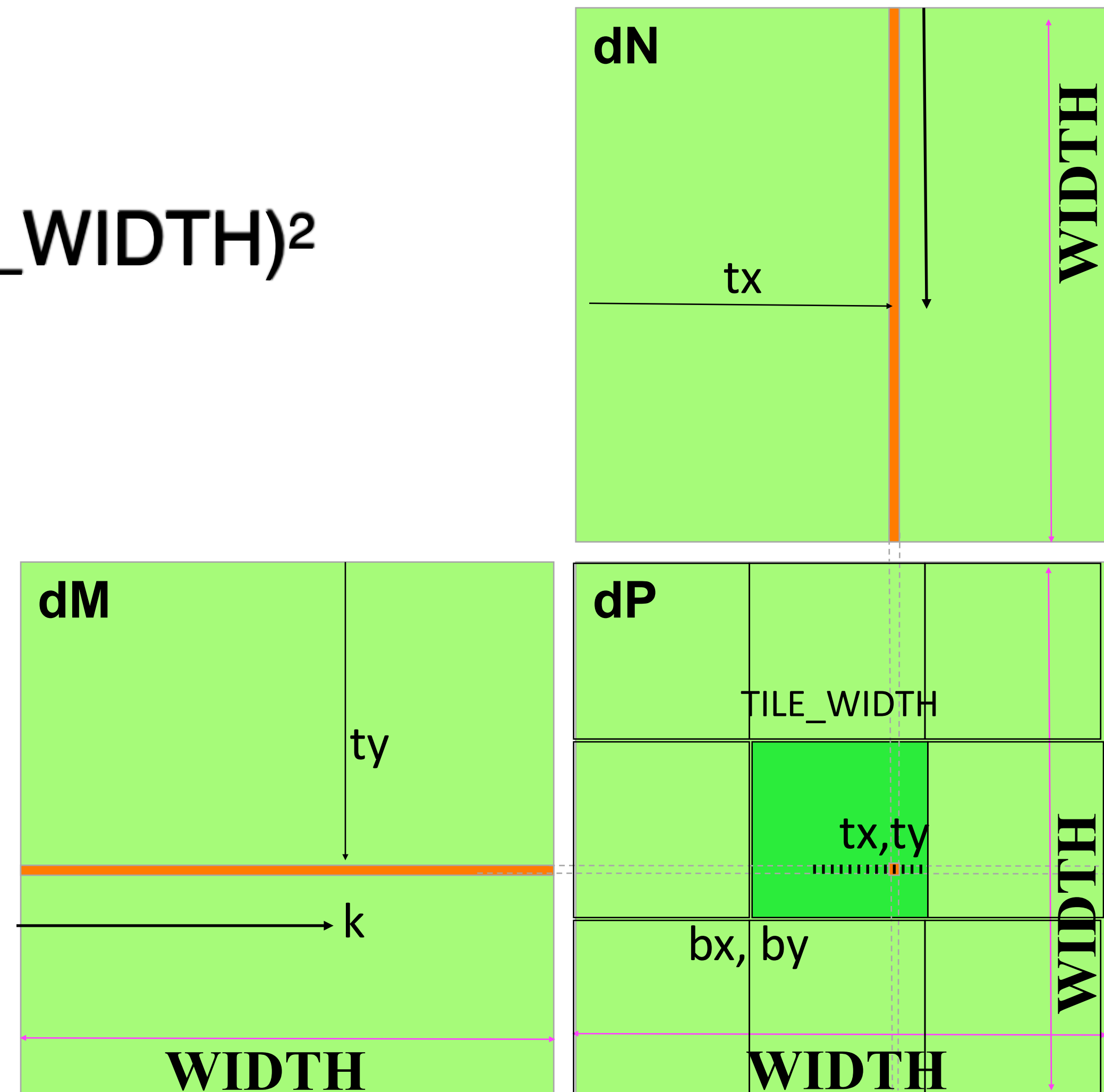


[Source nVIDIA]

# MATMUL: Multiple Thread Blocks

- 2D thread block computes a  $(\text{TILE\_WIDTH})^2$  sub-matrix of Pd
  - $(\text{TILE\_WIDTH})^2$  threads/block
- Generate a 2D Grid of  $(\text{WIDTH}/\text{TILE\_WIDTH})^2$  blocks

If more than  $2^{16} \times 2^{16} \times 2^{16}$ , blocks needed, invoke multiple kernels.



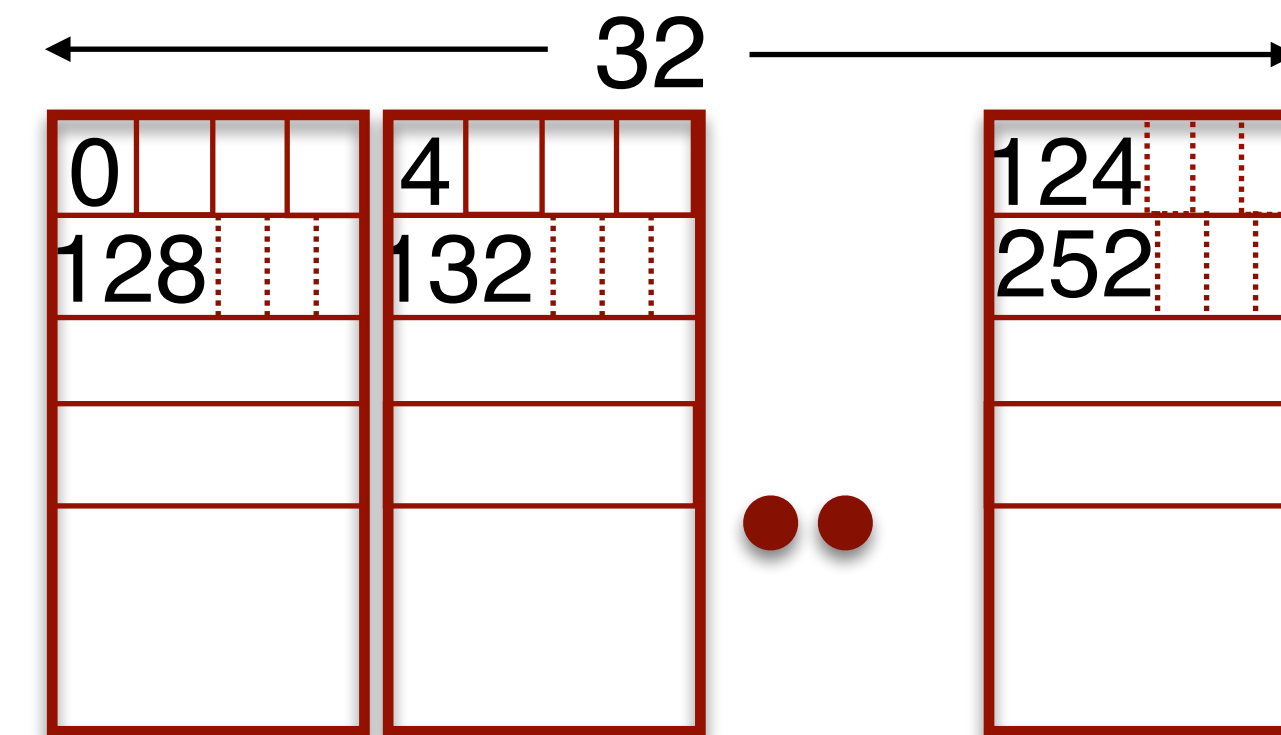
[Source nVIDIA]

- All (active) threads of the warp load/store
- Concurrent accesses by a warp can be coalesced into memory transactions
- Shared memory distributed into banks
  - Non-conflicting access efficient
- Atomic operations are serialized



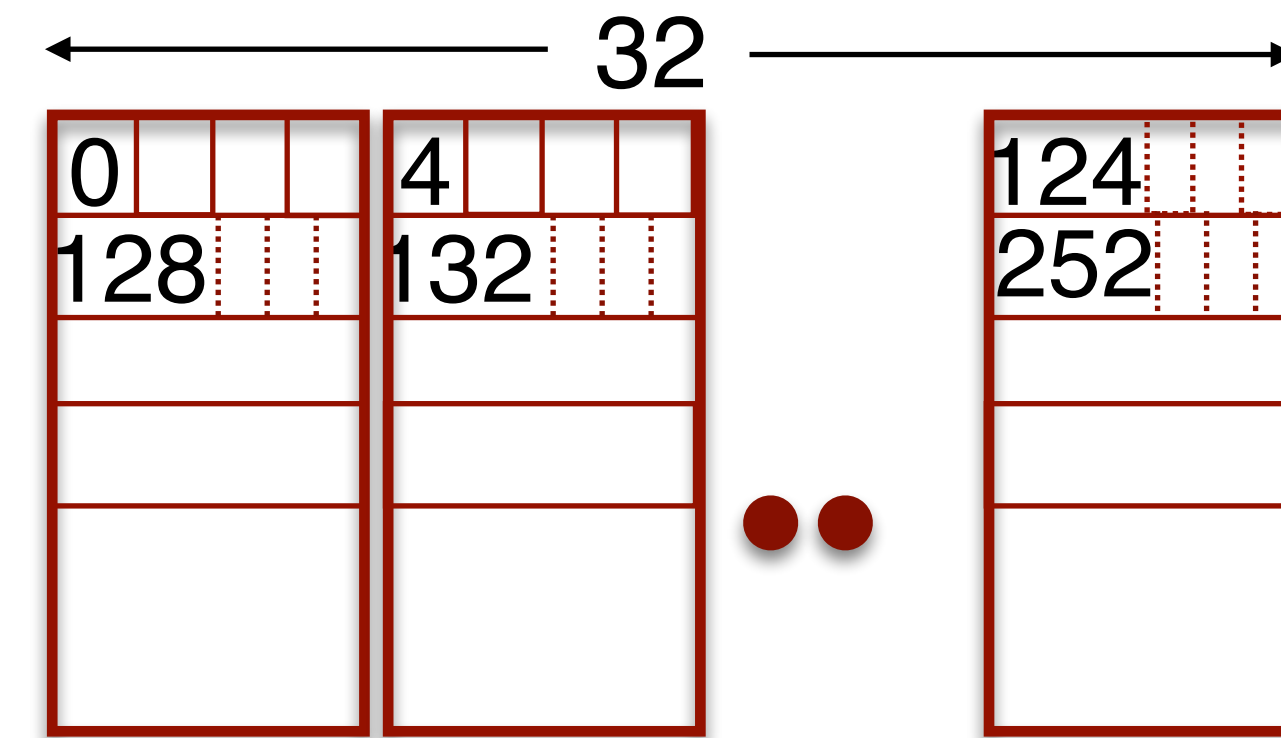
## Shared Memory Coalescing

- Multi-ported, atom size 4 bytes
  - Address space interleaved in banks
  - 32 banks, one word each
- `__shared__ float floats[N];`
  - `float data = floats[step * tid];`
  - No bank conflict for odd values of step



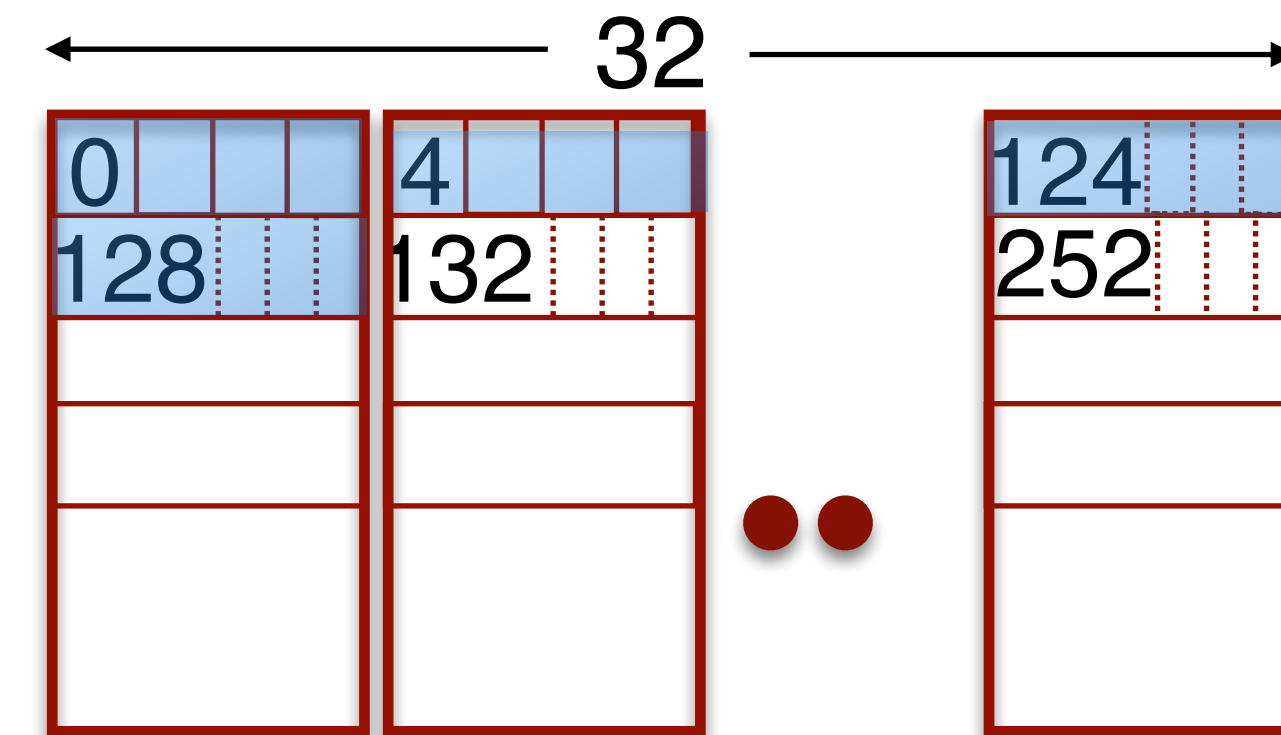
## Shared Memory Coalescing

- Multi-ported, atom size 4 bytes
  - Address space interleaved in banks
  - 32 banks, one word each
- `__shared__ float floats[N];`
  - `float data = floats[step * tid];`
  - No bank conflict for odd values of step
- `__shared__ float M[][33]`
  - Efficient Column-major access



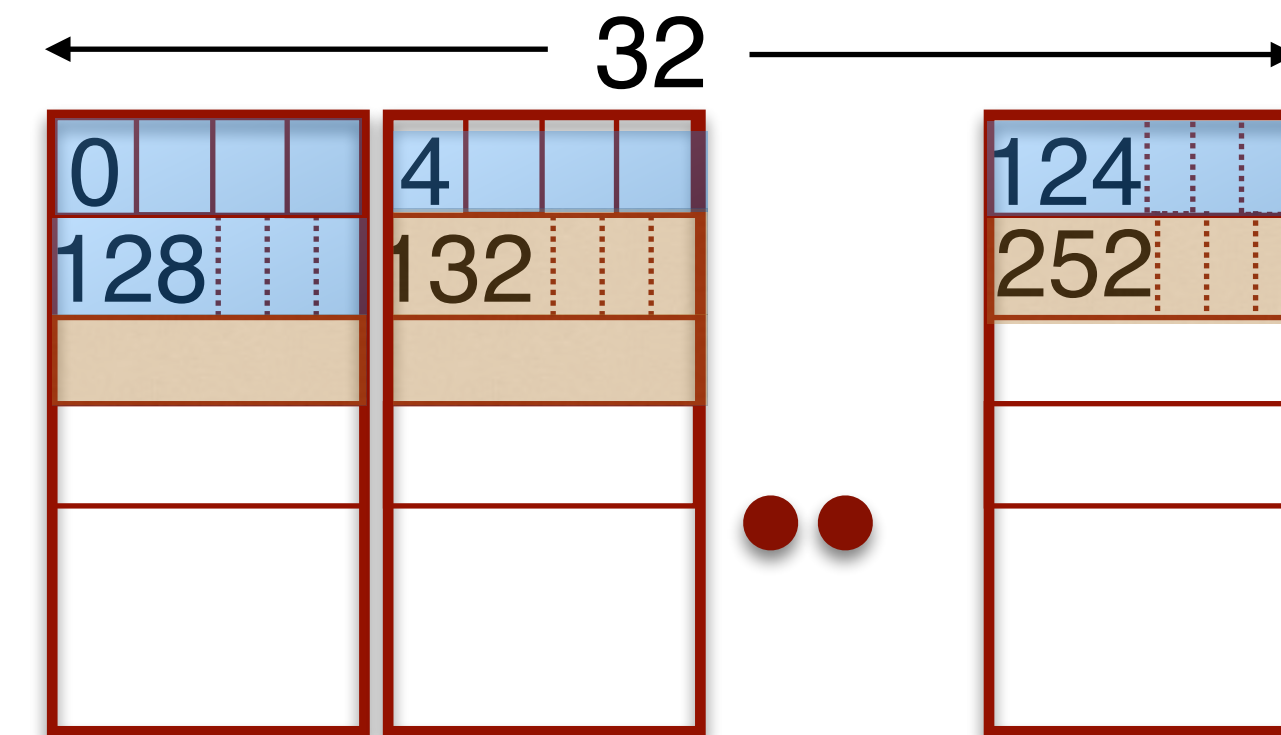
## Shared Memory Coalescing

- Multi-ported, atom size 4 bytes
  - Address space interleaved in banks
  - 32 banks, one word each
- `__shared__ float floats[N];`
  - `float data = floats[step * tid];`
  - No bank conflict for odd values of step
- `__shared__ float M[][33]`
  - Efficient Column-major access



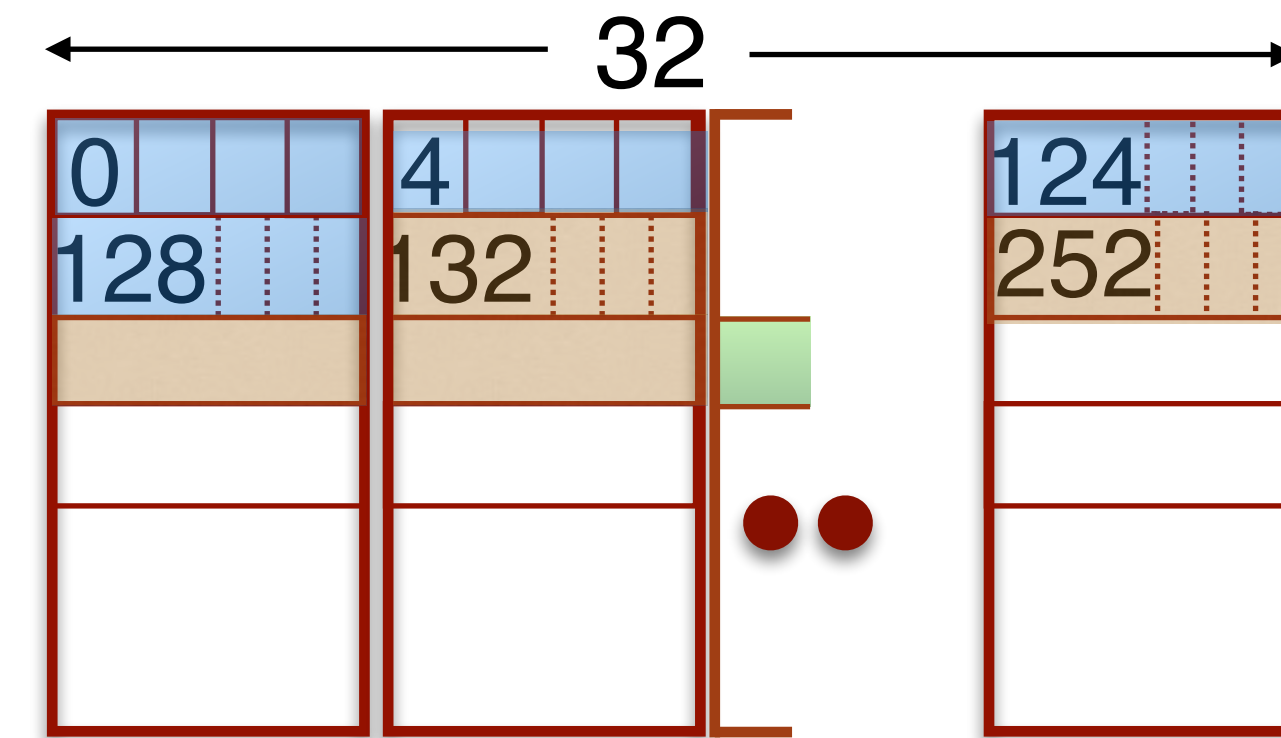
## Shared Memory Coalescing

- Multi-ported, atom size 4 bytes
  - Address space interleaved in banks
  - 32 banks, one word each
- `__shared__ float floats[N];`
  - `float data = floats[step * tid];`
  - No bank conflict for odd values of step
- `__shared__ float M[][33]`
  - Efficient Column-major access



## Shared Memory Coalescing

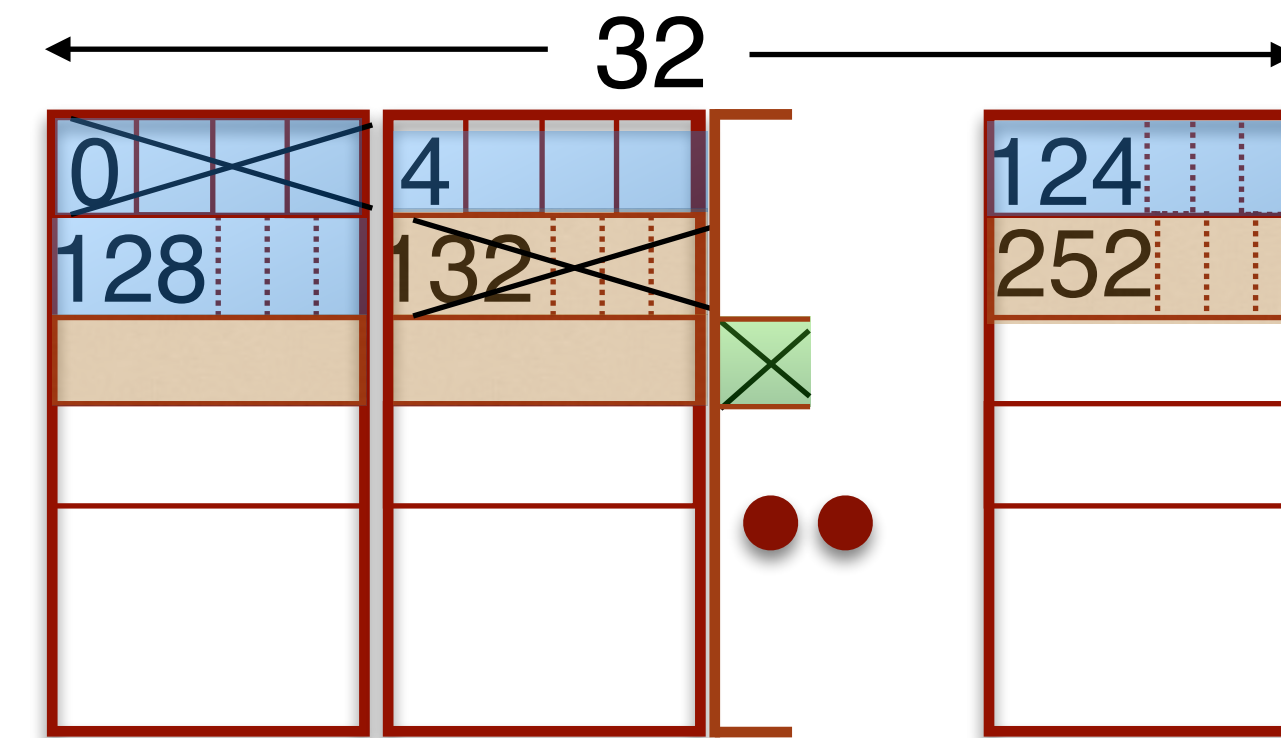
- Multi-ported, atom size 4 bytes
  - Address space interleaved in banks
  - 32 banks, one word each
- `__shared__ float floats[N];`
  - `float data = floats[step * tid];`
  - No bank conflict for odd values of step
- `__shared__ float M[][33]`
  - Efficient Column-major access





## Shared Memory Coalescing

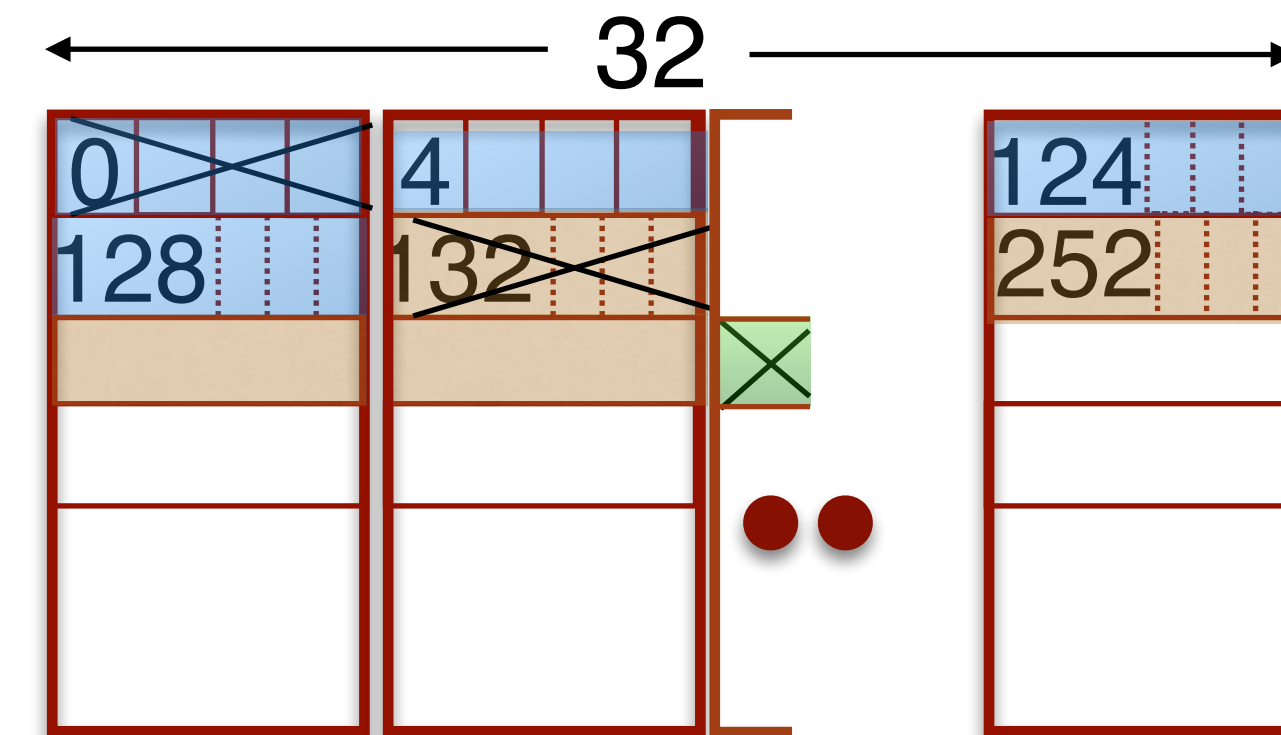
- Multi-ported, atom size 4 bytes
  - Address space interleaved in banks
  - 32 banks, one word each
- `__shared__ float floats[N];`
  - `float data = floats[step * tid];`
  - No bank conflict for odd values of step
- `__shared__ float M[][33]`
  - Efficient Column-major access





# Shared Memory Coalescing

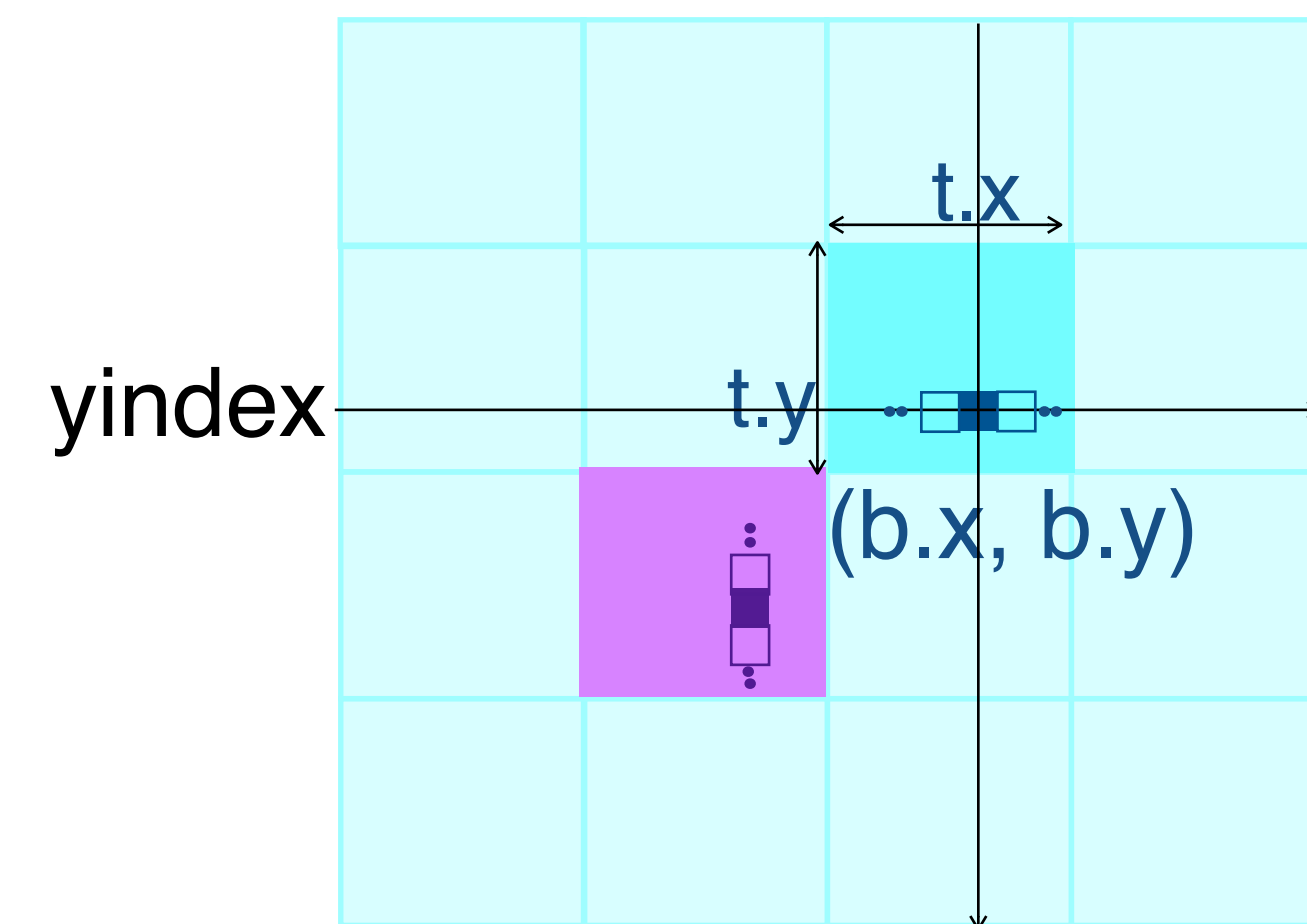
- Multi-ported, atom size 4 bytes
  - Address space interleaved in banks
  - 32 banks, one word each
- `__shared__ float floats[N];`
  - `float data = floats[step * tid];`
  - No bank conflict for odd values of step
- `__shared__ float M[][33]`
  - Efficient Column-major access



L1 and L2 have standard cache behavior  
128 byte Cache line for L1  
32 byte Atom size for L2, Gmem  
Some cache policy control

# Transpose

```
__global__ void transposeGlobal(float *din, float* dout, int width)
{
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index_in = xIndex + width * yIndex;
    int index_out = yIndex + width * xIndex;
    // Loop?: out[index_out+i] = din[index_in+i*width]
    dout[index_out] = din[index_in];
}
```

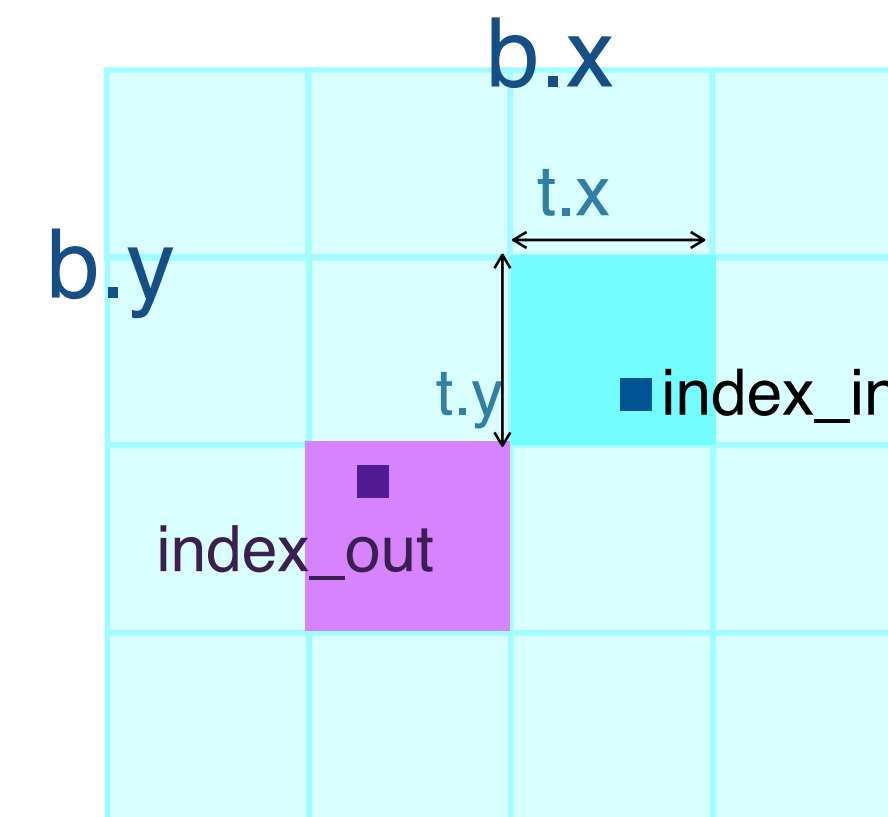


# Transpose

```
__global__ void transposeShared(float *dout, float* din, int width)
{
    __shared__ float stile[TILE_DIM][TILE_DIM];
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + width*yIndex

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + width*yIndex;

    stile[threadIdx.y][threadIdx.x] = din[index_in];
    __syncthreads();
    dout[index_out] = stile[threadIdx.x][threadIdx.y];
}
```

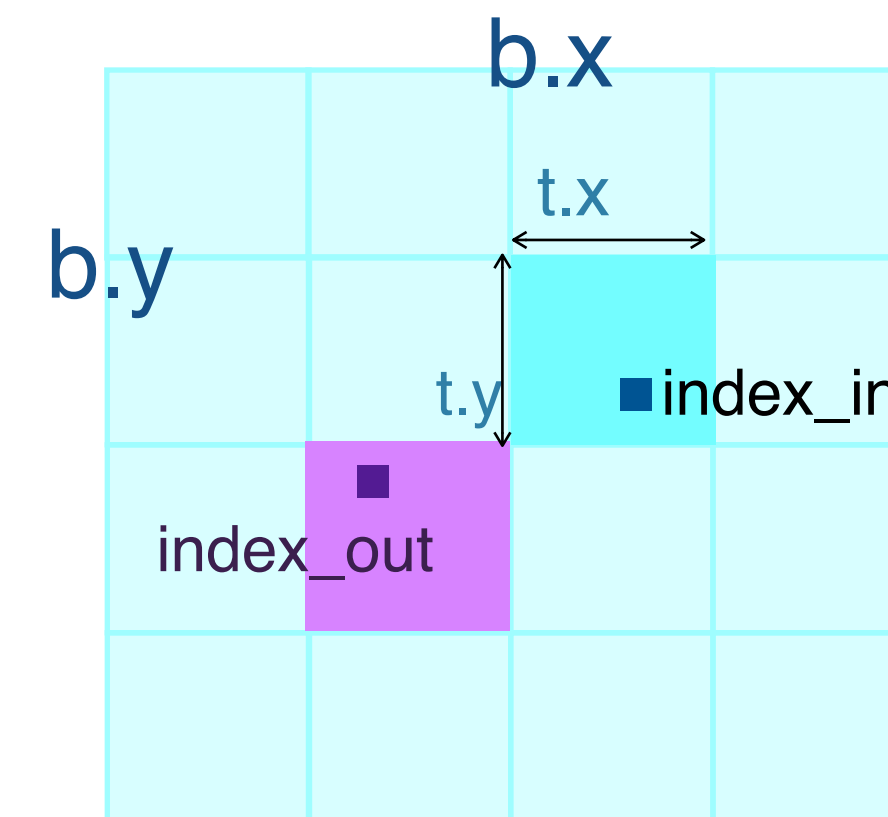


# Transpose

```
__global__ void transposeShared(float *dout, float* din, int width)
{
    __shared__ float stile[TILE_DIM][TILE_DIM+1]
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + width*yIndex

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + width*yIndex;

    stile[threadIdx.y][threadIdx.x] = din[index_in];
    __syncthreads();
    dout[index_out] = stile[threadIdx.x][threadIdx.y];
}
```



Bring entire submatrix to smem  
(Row major)

Read column-wise from smem

- Tasks in a stream execute in sequence
  - Individual tasks in the stream may be parallel
  - Streams may interleave (subject to memory ops)
- To use Cuda streams:
  - create a stream object
  - specify it as a parameter to events
    - kernel launches, host ↔ device memory copies ..
- Default stream == 0-stream
  - Special exclusive stream
    - Does not interleave with other streams
- Also see `cudaStreamAddCallback()`

**Can Query for  
stream's completion**

**Can Query for events  
recorded by streams**



e.g., Streams

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]); // Need two streams
float* hostPtr;
cudaMallocHost(&hostPtr, 2*size); // pagelocked memory
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i*size, hostPtr + i*size,
                    size, cudaMemcpyHostToDevice, stream[i]); // Initiate H->D transfer
    DoKernel<<<100, 512, 0, stream[i]>>> // After that initiate Kernel
        (outputDevPtr + i*size, inputDevPtr + i*size, size);
    cudaMemcpyAsync(hostPtr + i*size, outputDevPtr + i*size,
                    size, cudaMemcpyDeviceToHost, stream[i]); // After that initiate D->H
}
cudaDeviceSynchronize(); // Wait for both streams

// Later cudaStreamDestroy
```



- `cudaDeviceSynchronize()`
  - waits until all preceding commands in all streams have completed
- `cudaStreamSynchronize(stream)`
  - waits until all preceding commands in the given stream have completed
- `cudaStreamQuery(stream)`
  - have all preceding commands in stream completed?
- `cudaStreamWaitEvent(stream, event, flag)` Recall `cudaEventRecord()`
  - Subsequent operations in *stream* may begin only after *event* has happened

- **void \_\_syncthreads()**
  - block barrier AND
  - ensure all global/shared memory accesses by all threads are visible in the block
- **int \_\_syncthreads\_count(int predicate)**
  - returns the count of threads where predicate != 0
- **int \_\_syncthreads\_and(int predicate)**
  - returns non-zero *iff* predicate != 0 for *all* threads
- **int \_\_syncthreads\_or(int predicate)**
  - returns non-zero *iff* predicate != 0 for *any* threads

- **T \_\_shfl\_sync(unsigned mask, T var, int src)**
  - Return var specified by src
- **int \_\_all\_sync(unsigned mask, int predicate)**
  - Return non-0 iff predicate != 0 for all active threads in mask.
- **int \_\_any\_sync(unsigned mask, int predicate)**
  - Return non-0 iff predicate true for *any* thread
- **uint \_\_ballot\_sync(unsigned mask, int predicate)**
  - Return an int with  $i^{\text{th}}$  bit set iff predicate is true for the  $i^{\text{th}}$  thread of the warp
- **uint \_\_activemask()**
  - Return bit mask with 1 for active threads

Warp-collectives  
for active threads  
and common mask

- **Read-modify-write on one 32-bit word**
  - Also 64-bit word in newer architecture
- **Block-wide atomics: All threads within the same block**
  - e.g., `atomicAdd_block`
- **Device-wide atomics: All GPU threads**
  - e.g., `atomicAdd`, `atomicSub`, `atomicMin`, `atomicExchg`
- **System-wide atomics: CPU+GPU threads**
  - e.g., `atomicAdd_system`
- **More generic Compare and Swap**
  - `atomicCAS()`

Need corresponding CPU functions,  
e.g., `__sync_fetch_and_add`



- **\_\_threadfence\_block**

- wait until all global/shared memory accesses by the caller are visible to block

- **\_\_threadfence**

- wait until all memory accesses by the caller are visible to
  - All threads in the thread block for shared memory accesses
  - All threads in the device for global memory accesses

- **\_\_threadfence\_system**

- wait until all memory accesses by the caller are visible to:
  - All threads in the thread block for shared memory accesses,
  - All threads in the device for global memory accesses,
  - Host threads for page-locked host memory accesses