CUDA Shared Memory

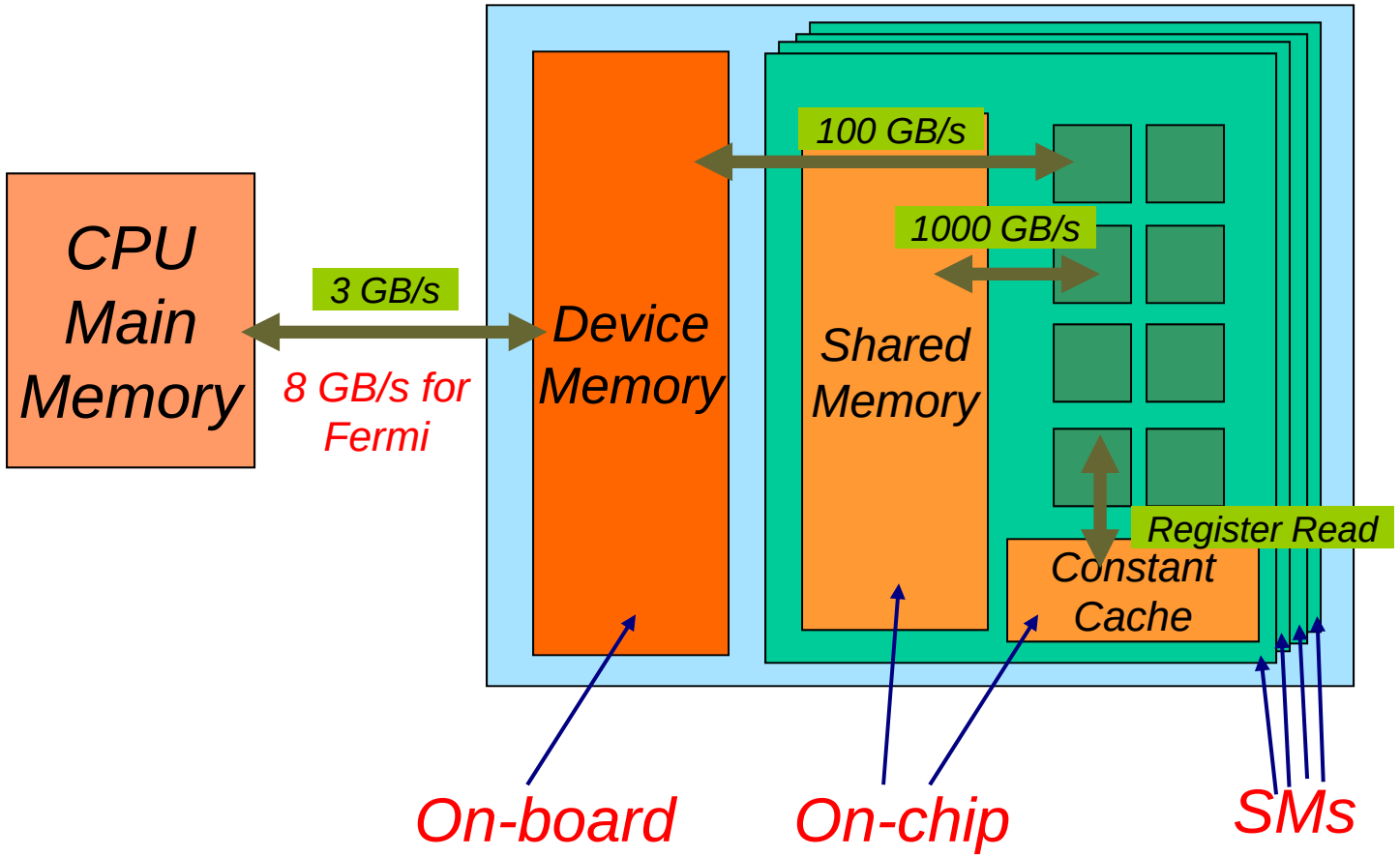
Outline

1. CUDA -- Shared Memory

2. Continue Example -- MMM

3. Continue Example -- Averaging Filter

Memory Hierarchy



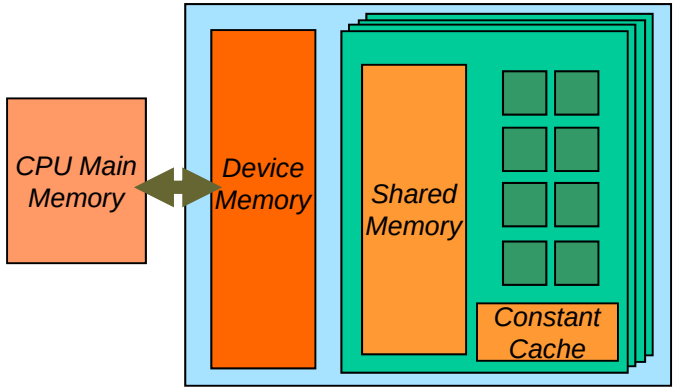
- On-chip shared memory is per SM.
- Shared memories on different SMs are completely independent of one-another.
- In fact, shared memory is partitioned by block and blocks can only access their own partitions of the shared memories on their own SMs

Memory Hierarchy (contd.)

Device memory (Global) is not cached

Constant and Texture caches are read-only

Data must be brought into the shared memory by the threads



Shared memory on each SM is divided into 16 equally-sized banks that can be accessed simultaneously

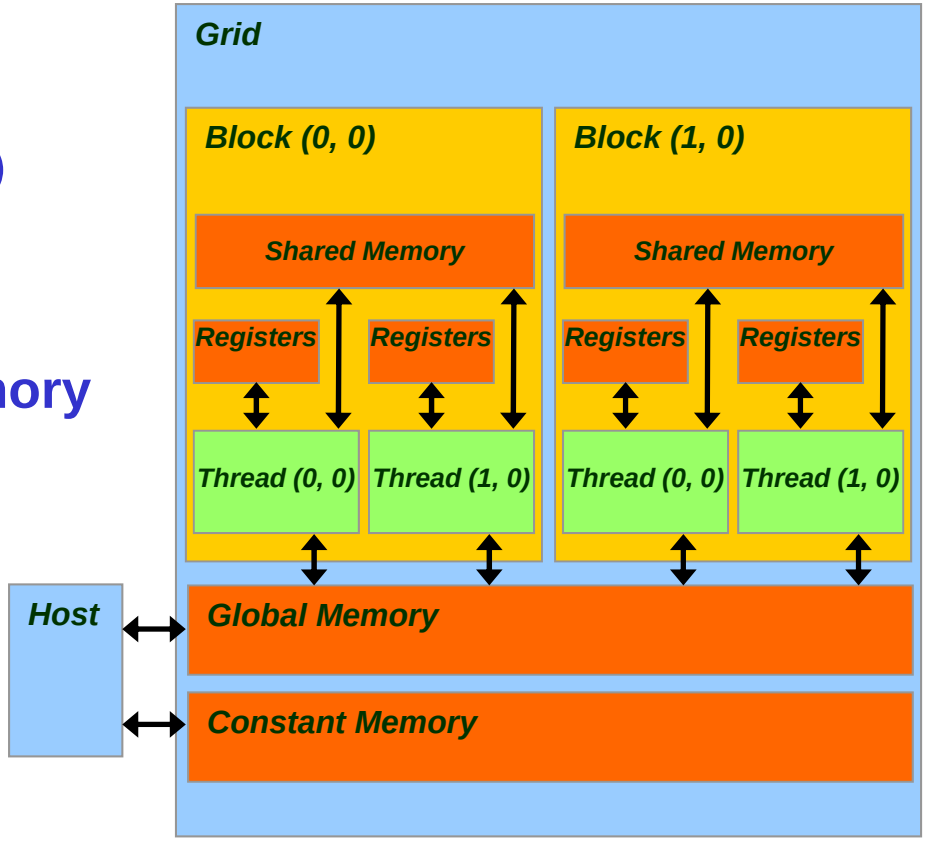
Shared memory is not persistent across "kernel" calls

Global, constant, and texture memory spaces are persistent across "kernel" calls

Local memory is in Device Memory. It is called local because it is the per-thread "spill" storage

G80 Implementation of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread *"local memory"* (sits in global memory)
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read (only) per-grid **constant memory**
- The host can R/W global, constant, and texture memories



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- **`__device__`** is optional when used with **`__local__`**, **`__shared__`**, or **`__constant__`**
- "automatic" scalar variables without qualifier reside in a register
  - compiler will spill to thread local memory
- "automatic" array variables without qualifier reside in thread-local memory

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

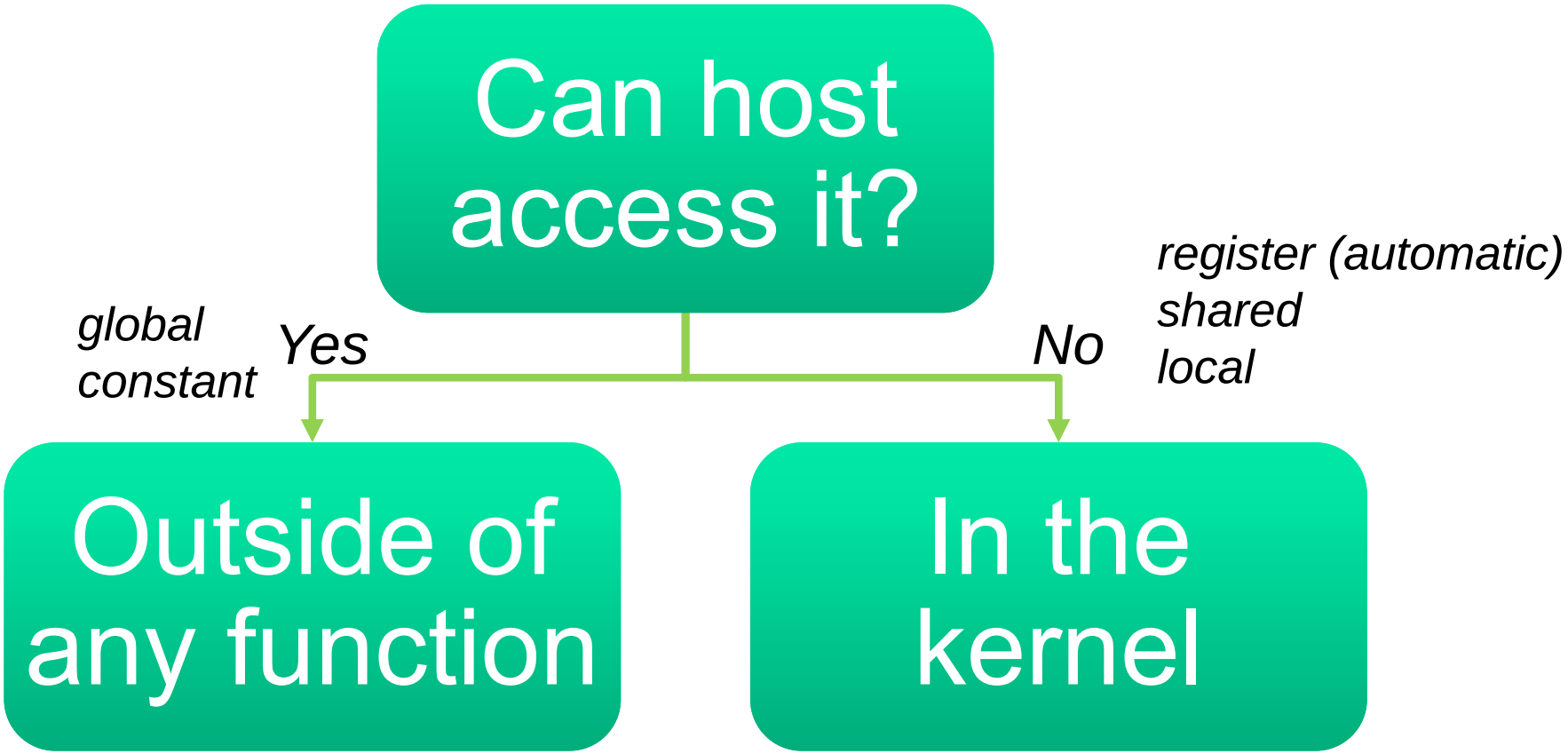
- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

CUDA Variable Type Scale

Variable declaration	Instances	Visibility
`int var;`	100,000s	1
`int array_var[10];`	100,000s	1
`__shared__ int shared_var;`	100s	100s
`__device__ int global_var;`	1	100,000s
`__constant__ int constant_var;`	1	100,000s

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 100s of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads

Where to declare variables?




```
__constant__ int constant_var;
__device__   int global_var;
```

```
int var;
int array_var[10];
__shared__   int shared_var;
```

Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    ```
    __global__ void KernelFunc(float* ptr)
    ```
  - Obtained as the address of a global variable:
    ```
    float* ptr = &GlobalVar;
    ```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Programming Strategy

- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data

- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)
  - *R/W shared **not** within a Block → global memory (very slow)*

For texture memory usage, see NVIDIA document.

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Programming Strategy, cont.

- A profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory
  - *Can be viewed as explicitly managed cache*

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Shared Memory -- Summary
(from text)

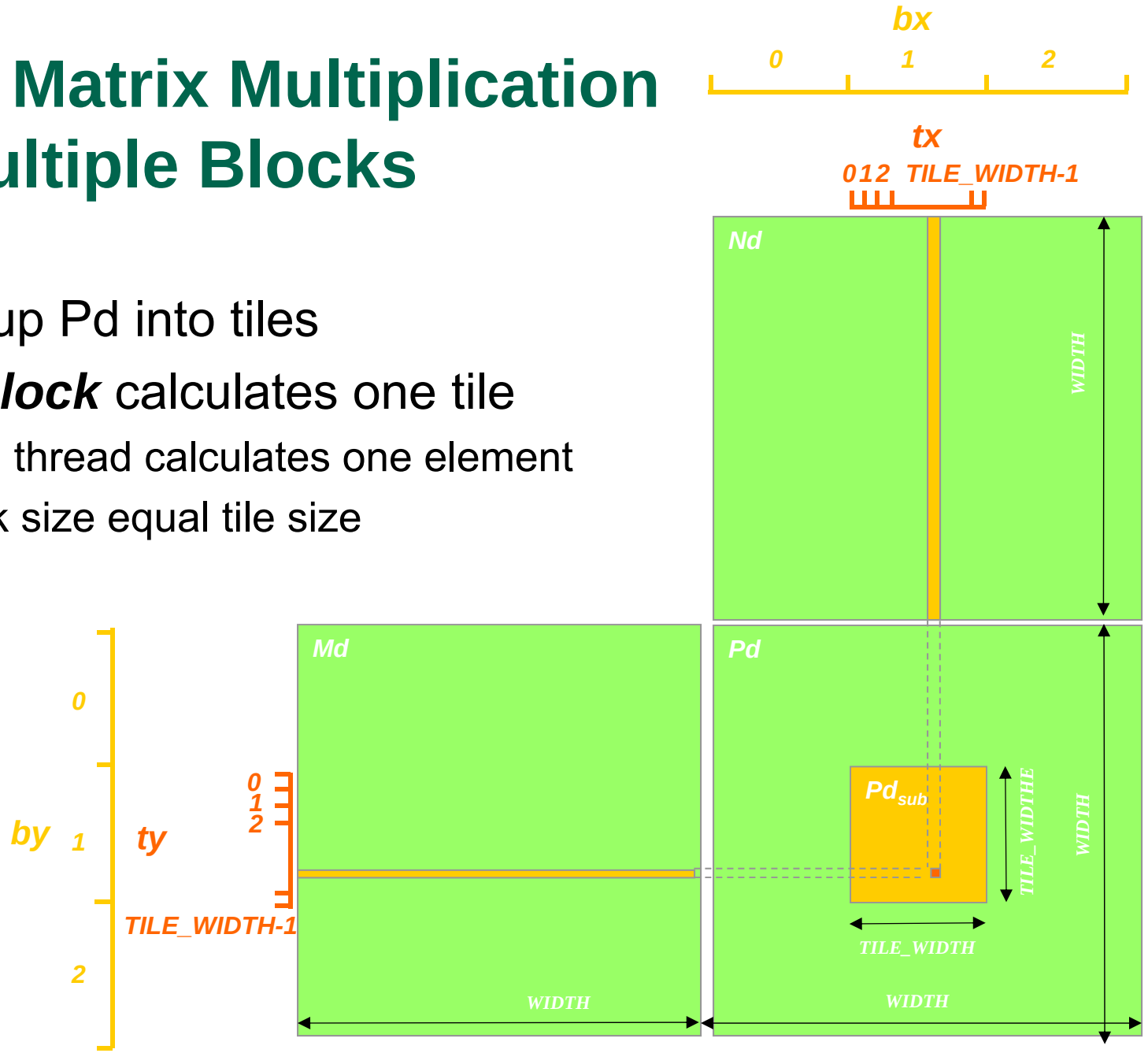
- The scope of a shared variable is within a thread BLOCK
  - That is, all threads in a block see the same version of a shared variable.
- A private version of the shared variable is created for, and used by, each thread BLOCK during kernel execution
- The lifetime of a shared variable is the duration of the kernel
  - When a kernel terminates, its shared variables cease to exist.
- Shared variables are an efficient means for threads ***within a block*** to collaborate with each other
- Accessing shared memory is extremely fast and highly parallel
  - CUDA programmers often use shared memory to hold the portion of global memory data that is heavily used.
  - One may need to adjust the algorithms used in order to create execution phases that heavily focus on small portions of the global memory data.
    - *see matrix multiplication (next)*

Matrix Multiplication using Shared Memory

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Review: Matrix Multiplication Using Multiple Blocks

- Break-up Pd into tiles
- Each ***block*** calculates one tile
  - Each thread calculates one element
  - Block size equal tile size



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

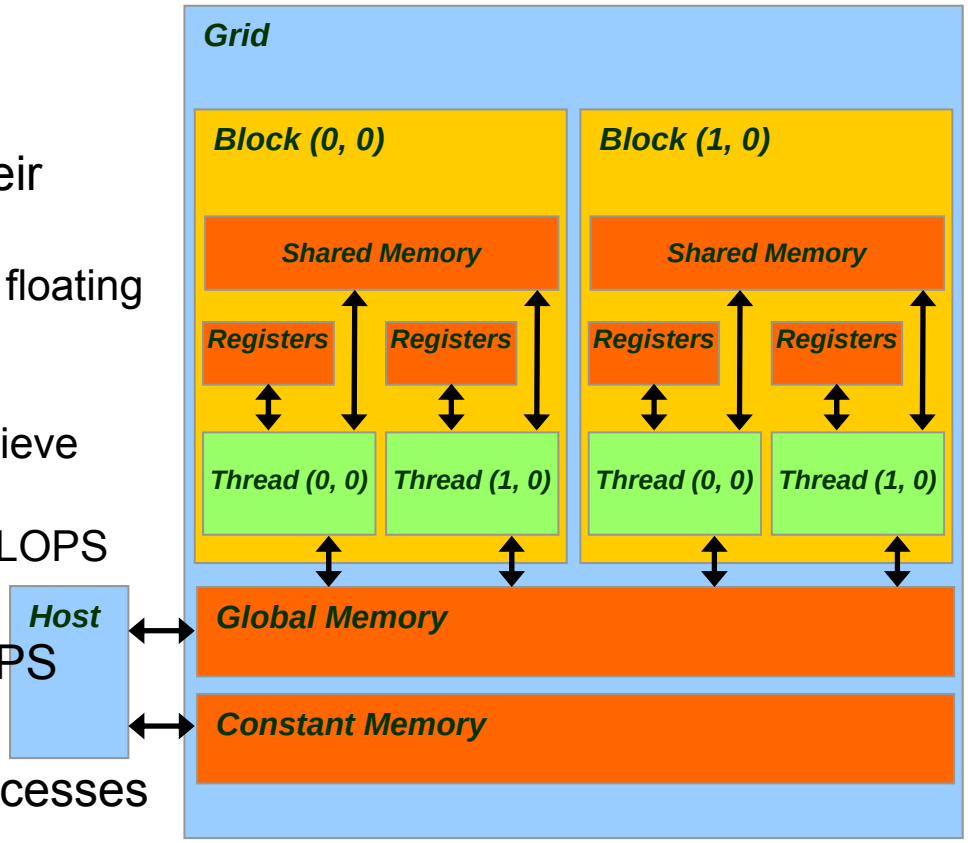
Review: Matrix Multiplication Kernel using Multiple Blocks

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;   // register
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

How about performance on G80?

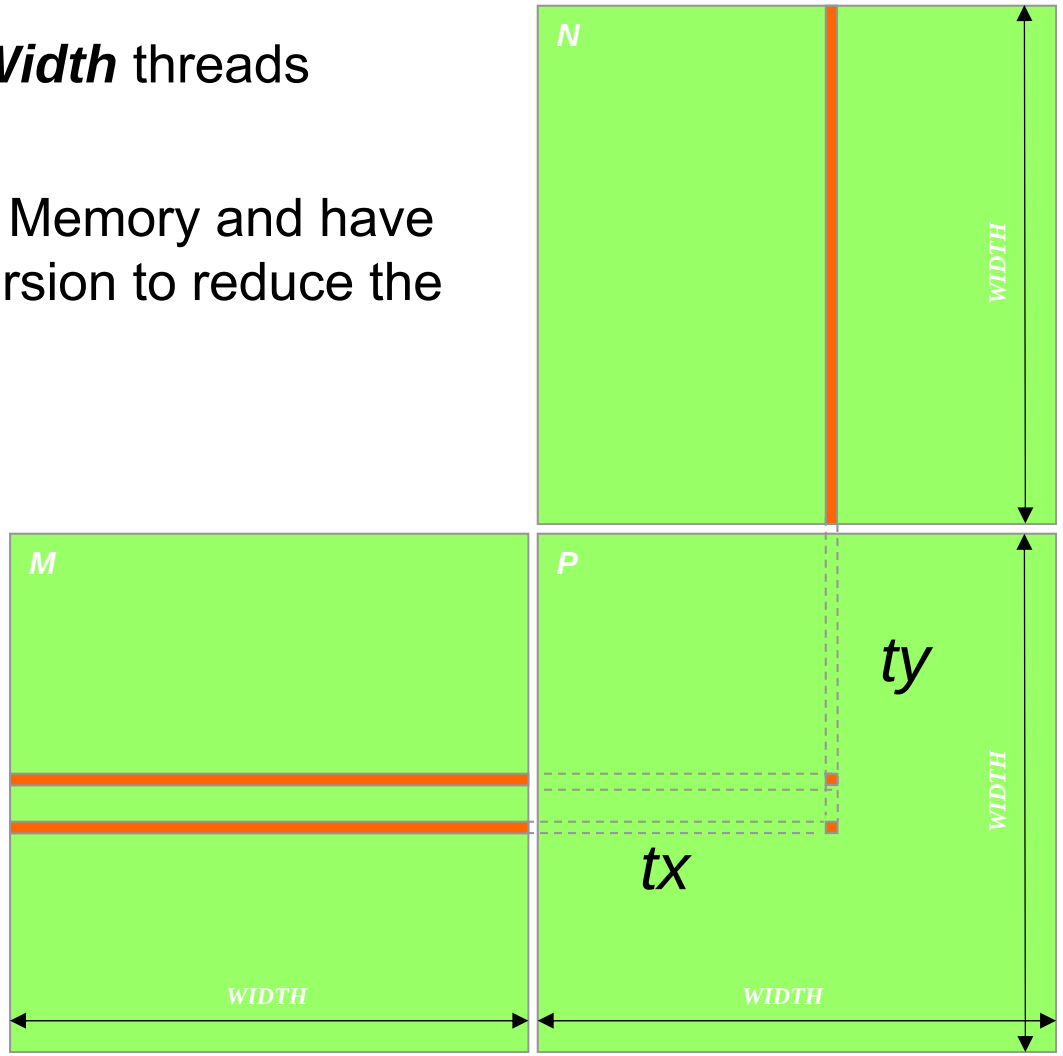
- How close to peak performance can we get with Global-Memory-only algorithm?
  - Peak = 346.5 GFLOPS
- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4Bs of memory bandwidth/FLOP
  - 4*346.5 = 1386 GB/s required to achieve peak FLOP rating
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code runs at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Idea: Use Shared Memory to reuse global memory data

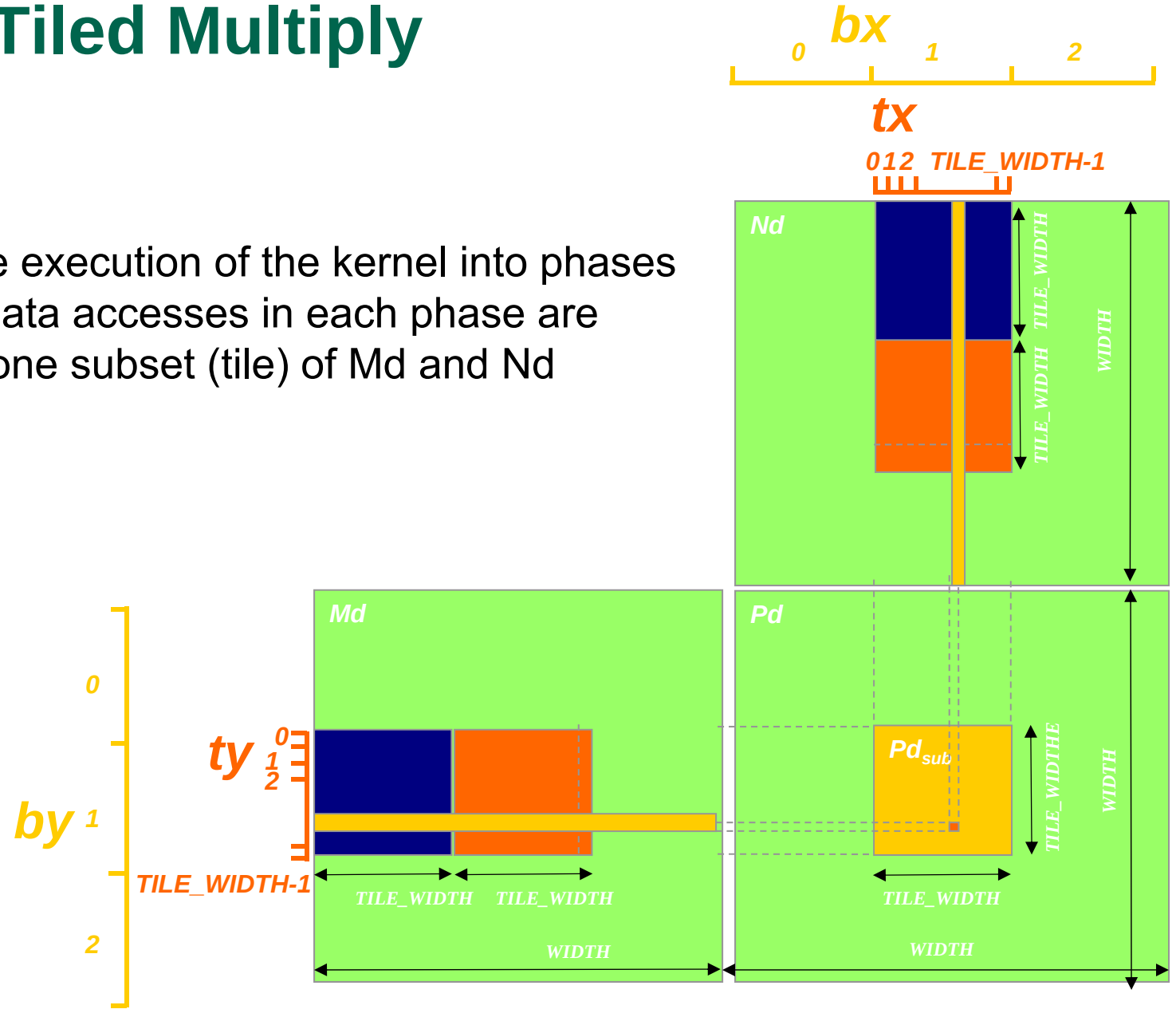
- Each input element is read by ***Width*** threads
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - Tiled algorithms



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana Champaign

Tiled Multiply

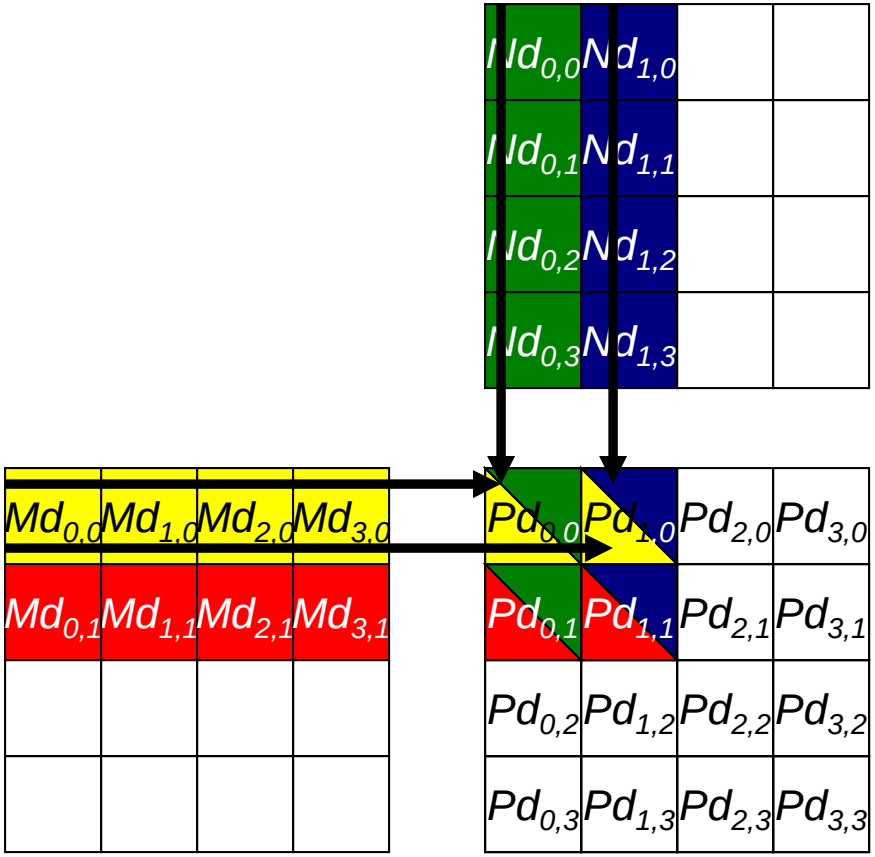
- Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one subset (tile) of Md and Nd



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana Champaign

A Small Example

```
// Let 1 BLOCK compute one 2x2 output tile
// Unroll loop to make reuse obvious

  P00 += M00 * N00
  P00 += M10 * N01
  P00 += M20 * N02
  P00 += M30 * N03

  P10 += M00 * N10
  P10 += M10 * N11
  P10 += M20 * N12
  P10 += M30 * N13

  P01 += M01 * N00
  P01 += M11 * N01
  P01 += M21 * N02
  P01 += M31 * N03

  P11 += M01 * N10
  P11 += M11 * N11
  P11 += M21 * N12
  P11 += M31 * N13

// How much reuse is there?
// What can prevent reuse from being useful?
```



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

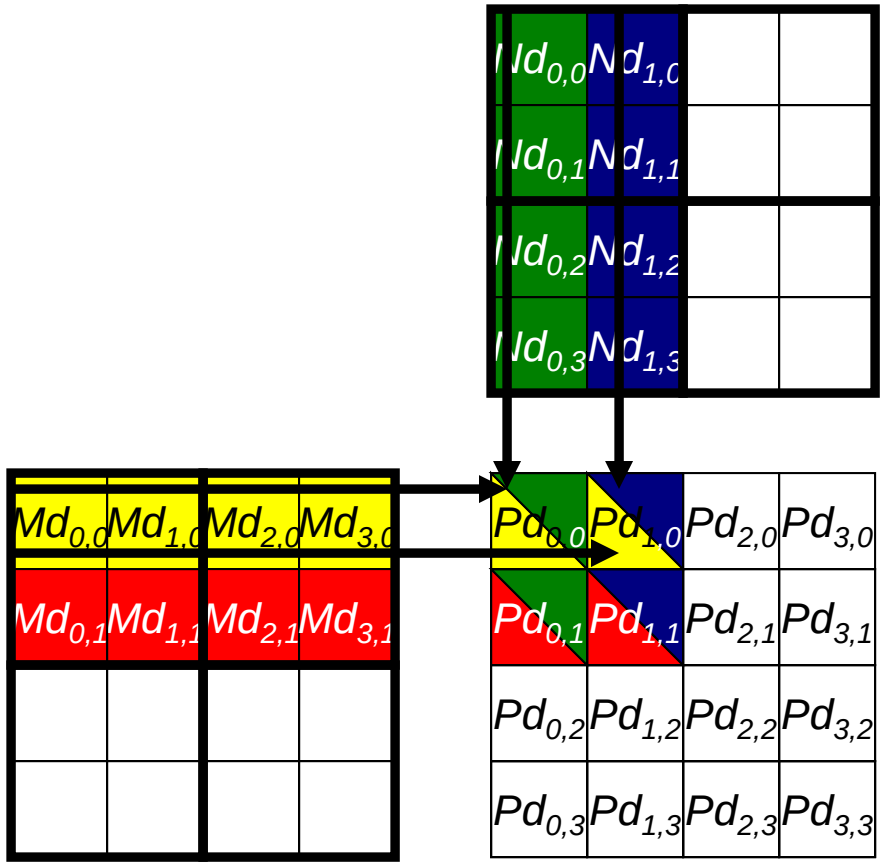
Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P

Access order

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Breaking Md and Nd into Tiles

```
// Let 1 BLOCK compute one 2x2 output tile
// This time, group by input tile
// Unroll loop to make reuse obvious

  P00 += M00 * N00
  P00 += M10 * N01
  P10 += M00 * N10
  P10 += M10 * N11
  P01 += M01 * N00
  P01 += M11 * N01
  P11 += M01 * N10
  P11 += M11 * N11

  P00 += M20 * N02
  P00 += M30 * N03
  P10 += M20 * N12
  P10 += M30 * N13
  P01 += M21 * N02
  P01 += M31 * N03
  P11 += M21 * N12
  P11 += M31 * N13

// How much reuse is there?
// Which thread executes which instructions?
```



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

```
// Let 1 BLOCK compute one 2x2 output tile

// How do you load the data?
// There are 4 threads
// For 1 block, each “phase” (tile) use
//    4 M elements (input)    M00, M01, M10, M11
//    4 N elements (input)    N00, N01, N10, N11
//    4 P elements (output)   P00, P01, P10, P11

// Which thread loads which elements?
```

Each phase of a Thread Block uses one tile from Md and one from Nd

	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

done in parallel

time

First-order Size Considerations in G80

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
- There should be many thread blocks
  - A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
- Each thread block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - For each Tile, each thread computes a length TILE_WIDTH (16) dot product for 2*16 FLOPs.
- Memory bandwidth no longer a limiting factor

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

CUDA Code – Kernel Execution Config

```
// Set up the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH,
             Width /  TILE_WIDTH);
```

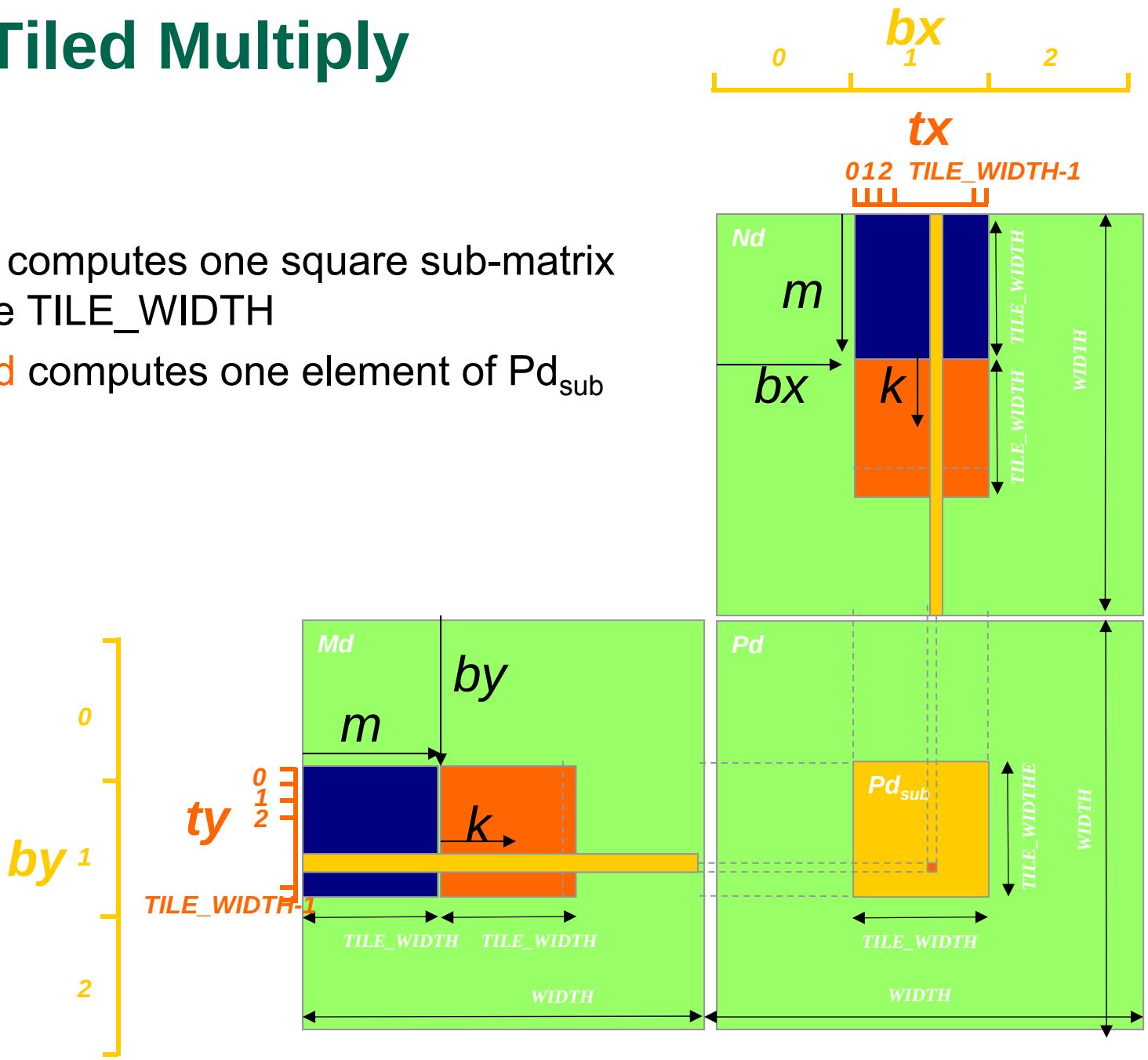
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  __shared__float Mds[TILE_WIDTH][TILE_WIDTH];  // Shared memory
  __shared__float Nds[TILE_WIDTH][TILE_WIDTH];  //   declarations

  int bx = blockIdx.x;  int by = blockIdx.y;    // ID thread
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0; // REGISTER!
  // Loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

   for (int k = 0; k < TILE_WIDTH; ++k)
     Pvalue += Mds[ty][k] * Nds[k][tx];
   __syncthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



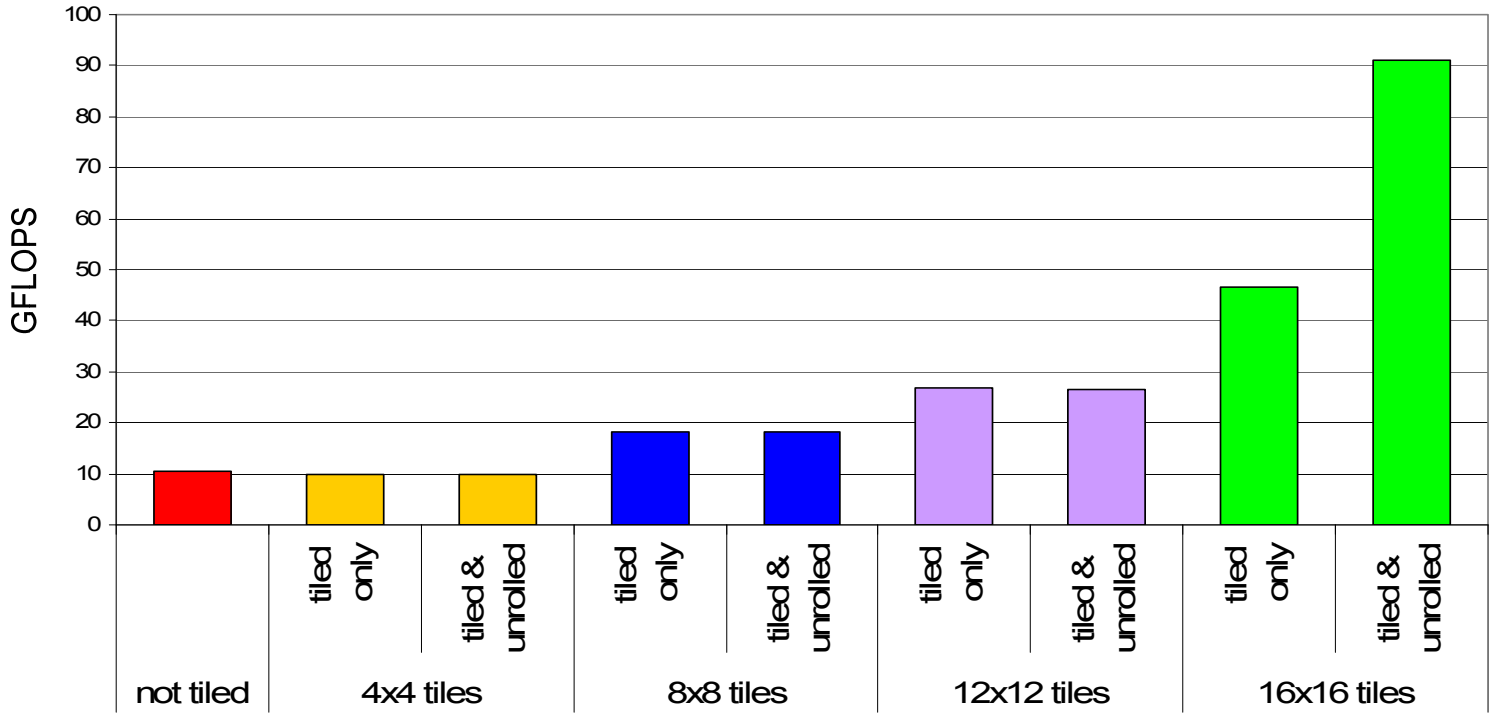
G80 Shared Memory and Threading

- Each SM in G80 has 16KB shared memory
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread BLOCK uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread BLOCKs actively executing
    - This allows up to 8*512 = 4,096 pending loads.
    - (8 blocks, 2 per thread, 256 threads per block)
  - The next TILE_WIDTH (32) would lead to 2*32*32*4B = 8KB shared memory usage per thread BLOCK
    - would allow only up to two thread blocks active at the same time

- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4GB/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Tiling Size Effects



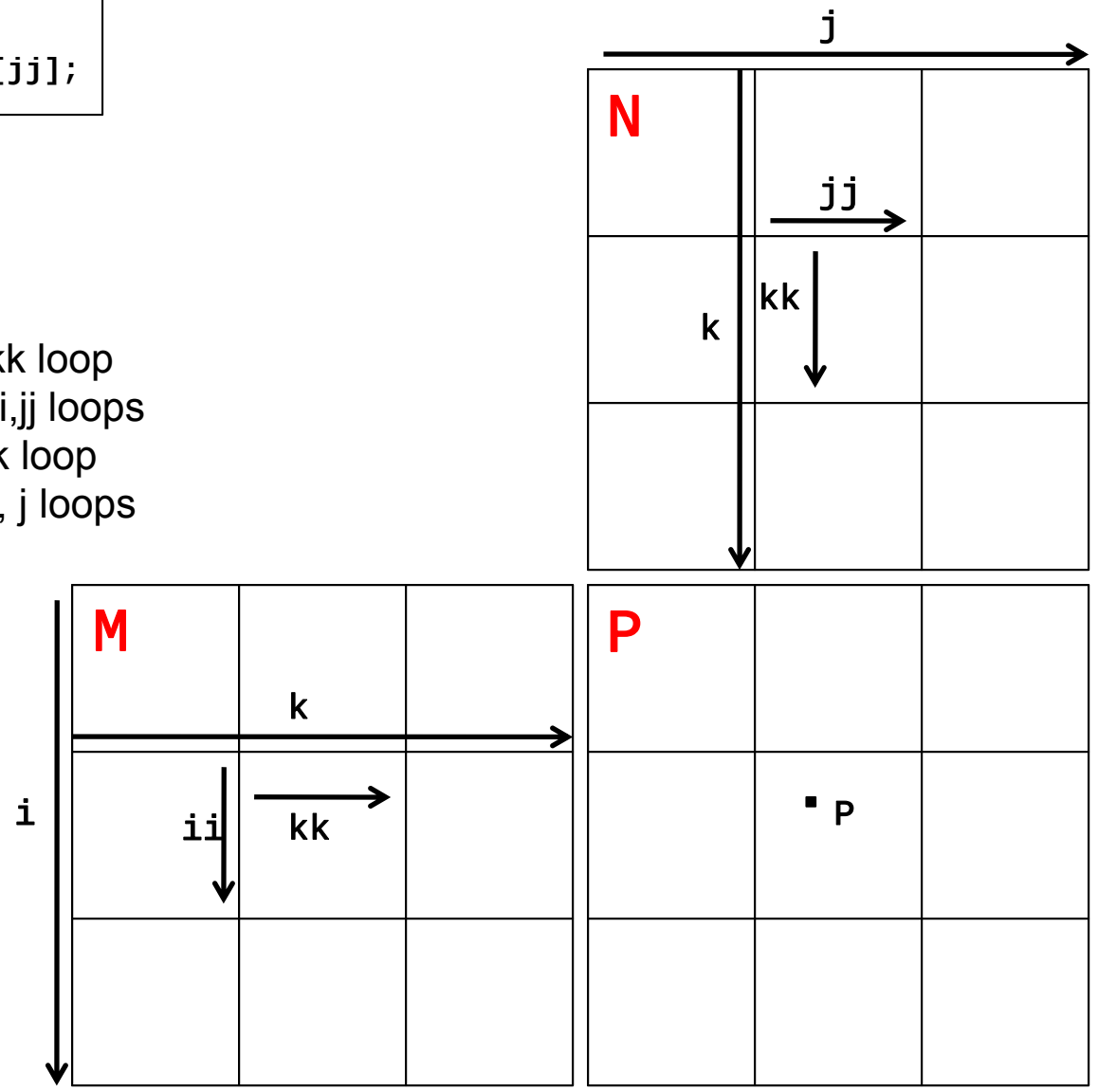
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana Champaign

```
for (i=0; i<N; i=i+B)
 for (j=0; j<N; j=i+B)
  for (k=0; k<N; k=k+B)
   for (ii=i; ii<i+B; ii=ii+1)
    for (jj=j; jj<j+B; jj=jj+1)
     for (kk=k; kk<k+B; kk=kk+1)
      x[ii][jj] = y[ii][kk] * z[kk][jj];
```

What is each THREAD computing?
What is each BLOCK computing?

NxN THREADs
3 x 3 = 9 BLOCKs
NxN/9 = THREADs/BLOCK

Inner loop of each THREAD = kk loop
THREADs within BLOCK = ii,jj loops
Different phases within THREAD = k loop
Different BLOCKs = i, j loops



Continue example: Simple THREAD/BLOCK interaction

Continue Example

The purpose of this example is to look at simple THREAD interaction under various scenarios. Assume →

- Some number of SMs
- Possibly multiple BLOCKs
- ***Global Memory and SM Shared Memory***

Averaging filter

- Input = 1D array with N elements
- Output = 1D array with N elements
- Operation – Each output array element is the average of its corresponding element in the input array and its neighbors.

Example Parameters

Averaging filter

- Input = 1D array with N elements
- Output = 1D array with N elements
- Operation – Each output array element is the average of its corresponding element in the input array and its neighbors.

0. Read and Write Different/Same arrays.

N → B[i] = (A[i-1]+A[i]+A[i+1])/3

Y → A[i] = (A[i-1]+A[i]+A[i+1])/3

1. Many iterations or One iteration?

N → One iteration

Y → Many iterations

2. Many Blocks or One Block?

N → One BLOCK

Y → Many BLOCKs

3. Shared Memory or Global Memory Only?

N → Global Memory Only

Y → Shared Memory and Global Memory

16 Cases

CASES 8-15 -- Y(N/Y)(N/Y)(N/Y)

Use Shared Memory as well as Global Memory

For Shared Memory, ***single iteration makes no sense*** – this is just a useless copy operation. So no CASES 8,9,12,13

CASES 10,11,14,15 -- Y(N/Y)Y(N/Y)

3. Shared Memory and Global Memory

2. Single **OR** Many Blocks

1. Many iterations

0. Same **OR** Different arrays

CASE 10 (CASE 11 is similar)

3. Shared and Global Memory
2. Single Block
1. Multiple Iterations
0. Different Arrays (same array)

Note: Transfer data from global memory to shared and then back again when done.

Note: Shared memory arrays need to have a fixed size known at kernel launch time.

(check this -- maybe not necessarily)

```
#define threadN 256

__global__ void Average(float* A, float* B, int N)
{ // start by getting the number of threads in BLOCK
  int i = threadIdx.x;      // who am I?
  __shared__float Ads[threadN];
  __shared__float Bds[threadN];
// transfer data from global to shared memory
  Ads[i] = A[i];  // each thread, coalesced
  Bds[i] = B[i];
  __syncthreads();
// do averaging
  for (int j = 0; j < 50; j++) {
    if (i > 0 && i < threadN-1) {
      Bds[i] = (Ads[i-1] + Ads[i] + Ads[i+1])/3;
      __syncthreads();
      Ads[i] = (Bds[i-1] + Bds[i] + Bds[i+1])/3;
      __syncthreads();
    }
  }
// transfer data from shared to global memory
 __syncthreads();
  A[i] = Ads[i];
  B[i] = Bds[i];
}

int main()
{ …
 // Kernel invocation
  dim3 dimBlock(256,1,1);
  Average<<<1, dimBlock>>>(A,B,256);
}
```

CASE 14 (CASE 15 is similar)

3. Shared and Global Memory
2. Multiple Blocks
1. Multiple Iterations
0. Different Arrays

First recall the Global-Memory-Only ***CASE 5*** (at right →). Kernels are spawned only for a single operation.

This is obviously no good for Shared Memory:

Shared Memory is LOST across kernel calls.

```
// CASE 5 -- Different Arrays, Multiple iterations
__global__ void Average1(float* A, float* B, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x
  if (i > 0 && i < N-1)
    B[i] = (A[i-1] + A[i] + A[i+1])/3;
}

__global__ void Average2(float A[N], float B[N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x
  if (i > 0 && i < N-1)
    A[i] = (B[i-1] + B[i] + B[i+1])/3;
}

int main()
{
// Kernel invocations -- Since these are
// asynchronous, must force to launch all at once.
  dim3 dimGrid(16,1,1);
  dim3 dimBlock(256,1,1);
  for (int i = 0; i < 50; i++) {
    Average1<<<dimGrid, dimBlock>>>(A, B, N);
    cudaThreadSynchronize();
    Average2<<<dimGrid, dimBlock>>>(A, B, N);
    cudaThreadSynchronize();
  }
}
// cudaThreadSynchronize() → Blocks until the device has completed
//  all preceding requested tasks
```

CASE 14 (CASE 15 is similar)

3. Shared and Global Memory

2. Multiple Blocks

1. Multiple Iterations

0. Different Arrays

PROBLEM: Each THREAD must communicate with its neighbors.

With ONE BLOCK, that's OK, use normal SYNC mechanisms.

With MULTIPLE BLOCKs, not OK:

→ Cannot SYNC across multiple BLOCKs

→ Data in Shared Memory is not persistent across kernel launches

Possible **SOLUTION: EXTRA WORK per BLOCK**

Possible **SOLUTION: EXTRA WORK per BLOCK**

For this example, let each BLOCK compute extra elements on the borders with its neighboring BLOCKs. (see example on next slide)

For each additional iteration computed by the BLOCK, there is more REUSE of the elements copied from GLOBAL into SHARED memory.

However,

For each iteration, more border elements need to be loaded and computed which requires more transfers and more threads.

Break-even point depends on the work per BLOCK.

Four iterations with a single BLOCK

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IN	25	6	34	91	10	62	55	5	80	20	10	40	6	99	26	2
1	25	22	44	45	54	42	41	47	35	37	23	19	48	44	42	2
2	25	30	37	48	47	46	43	41	39	32	26	30	37	45	29	2
3	25	31	38	44	47	45	43	41	37	32	29	31	37	37	25	2
4	25	31	38	43	45	45	43	41	40	33	31	33	35	33	21	2

One iteration with multiple BLOCKs requires ONE “border” element per neighbor

	0	1	2	3	4	5	6	7	8
In	25	6	34	91	10	62	55	5	80
1	25	22	44	45	54	42	41	47	--

	7	8	9	10	11	12	13	14	15
In	5	80	20	10	40	6	99	26	2
1	--	35	37	23	19	48	44	42	2

Two iteration with multiple BLOCKs requires TWO “border” elements per neighbor

	0	1	2	3	4	5	6	7	8	9
IN	25	6	34	91	10	62	55	5	80	20
1	25	22	44	45	54	42	41	47	35	--
2	25	30	37	48	47	46	43	41	--	--

	6	7	8	9	10	11	12	13	14	15
IN	55	5	80	20	10	40	6	99	26	2
1	--	47	35	37	23	19	48	44	42	2
2	--	--	39	32	26	30	37	45	29	2

CODE SKETCH -- DOES NOT FULLY IMPLEMENT ALGORITHM

```
__global__ void kernel_avg_filter(float* x) {
    const int tid = IMUL(blockDim.x, blockIdx.x) + threadIdx.x; // Assuming 1D block & 1D grid
    const int local_tid = threadIdx.x;                          // Thread Id within the block
    const int threadN = IMUL(blockDim.x, gridDim.x);            // Assuming 1D block & 1D grid

    float temp; int i;

    // Number of threads = N / number of thread blocks; N = array length
    __shared__ float xsm[THREADS_PER_BLOCK + 2*NUM_ITERS];

    // Copy from the global memory to the shared memory
    // Center
    xsm[local_tid + NUM_ITERS] = x[tid];

    //Left
    if(local_tid < NUM_ITERS) {
        xsm[local_tid] = x[NUM_ITERS + local_tid];
    }

    //Right
    if(local_tid >= NUM_ITERS) {
        xsm[local_tid + THREADS_PER_BLOCK] = x[NUM_ITERS + tid];
    }
    __syncthreads();

    for(i = 0; i < NUM_ITERS; i++) {
        // Compute the center elements
        if (tid > 0 && tid < threadN-1) {  // compute the average
            temp = (xsm[local_tid + (NUM_ITERS-1)] + xsm[local_tid + (NUM_ITERS-1) + 1] +
                    xsm[local_tid + (NUM_ITERS-1) + 2]) / 3;
        }
        // Compute the left elements
        // Compute the right elements

        __syncthreads();

        // Update the elements (center, left, right) in the shared memory
        xsm[local_tid] = temp;

        __syncthreads();
    }
    __syncthreads();

    // Copy from the shared memory to the global memory
    if (tid > 0 && tid < threadN-1) {
        x[tid] = xsm[local_tid];
    }
}
```

```
//EXECUTABLE VERSION
// Define some constants
#define ARR_WIDTH           16
#define NUM_ITERS            4
#define NUM_BLOCKS           2
#define THREADS_PER_BLOCK  (ARR_WIDTH/NUM_BLOCKS) + 2*NUM_ITERS
// This kernel assumes that each thread copies one array element from the global memory to
// the shared memory. This also includes copying the overlapping corner elements. That is
// why the number of threads = (work per block + 2*number of iterations). Then only some of
// the threads perform the actual computation and even fewer update the global memory.
__global__ void kernel_avg_filter(float* x, int arrLen) {
    const int tid = IMUL(blockDim.x, blockIdx.x) + threadIdx.x;// Assume1D block and 1D grid
    const int bid = blockIdx.x;
    const int local_tid = threadIdx.x;                      // Thread Id within the block
    const int threadN = IMUL(blockDim.x, gridDim.x);        // Assuming 1D block and 1D grid

    float temp;
    int i;
    int arr_id;   //Index of the input array element for which this thread is responsible
    int work_per_block;
    // Threads per block = work per block + 2*NUM_ITERS
    __shared__ float xsm[THREADS_PER_BLOCK];
    work_per_block = (arrLen / NUM_BLOCKS);
    arr_id = bid * work_per_block + (local_tid - NUM_ITERS);

    // Copy from the global memory to the shared memory
    // Leave out the first NUM_ITERS elements and the last NUM_ITERS elements
    if(tid >= NUM_ITERS && tid < (threadN-NUM_ITERS)) {
      xsm[local_tid] = x[arr_id];
    }
    __syncthreads();

    for(i = 0; i < NUM_ITERS; i++) {
        if (local_tid > 0 && local_tid < THREADS_PER_BLOCK-1) {
            // Compute the average
            temp = (xsm[local_tid - 1] + xsm[local_tid] + xsm[local_tid + 1]) / 3;
       }
        __syncthreads();
        // Update only if computed
        if (local_tid > 0 && local_tid < THREADS_PER_BLOCK-1) {
            // and it is not the corner element of the original array
            if (arr_id > 0 && arr_id < (arrLen-1)) {
              xsm[local_tid] = temp;
            }
        }
        __syncthreads();
    }
    __syncthreads();
```

```
// Copy back from the shared memory to the global memory -- Only if the current thread
// was responsible to compute something and was also actually responsible to update
// the memory.  (since some threads do repeat the computation for their local use but are
// not responsible to update the global memory).
    if ((local_tid >= NUM_ITERS) && local_tid < (THREADS_PER_BLOCK-NUM_ITERS)) {
        if (arr_id > 0 && arr_id < (arrLen-1)) {
            x[arr_id] = xsm[local_tid];
        }
    }
}
// ---- Launch the kernel ---
kernel_avg_filter<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(d_my_x, ARR_WIDTH);
```