GPU Teaching Kit

Accelerated Computing

Module 4.1 – Memory and Data Locality

CUDA Memories

Objective

- To learn to effectively use the CUDA memory types in a parallel program
  - Importance of memory access efficiency
  - Registers, shared memory, global memory
  - Scope and lifetime

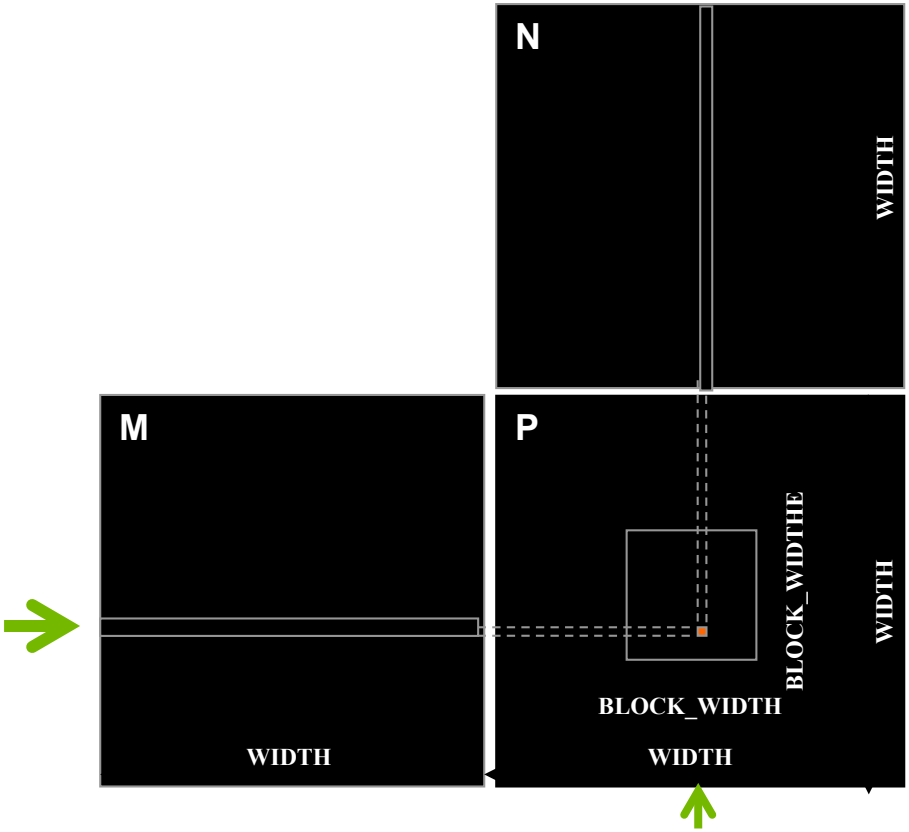
Review: Image Blur Kernel.

```
// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
            pixVal += in[curRow * w + curCol];
            pixels++; // Keep track of number of pixels in the accumulated total
        }
    }
}

// Write our new pixel value out
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```

How about performance on a GPU

- All threads access global memory for their input matrix elements
  - One memory accesses (4 bytes) per floating-point addition
  - 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
  - Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth
  - 4*1,500 = 6,000 GB/s required to achieve peak FLOPS rating
  - The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS
- This limits the execution rate to 3.3% (50/1500) of the peak floating-point execution rate of the device!
- Need to drastically cut down memory accesses to get close to the1,500 GFLOPS

Example – Matrix Multiplication



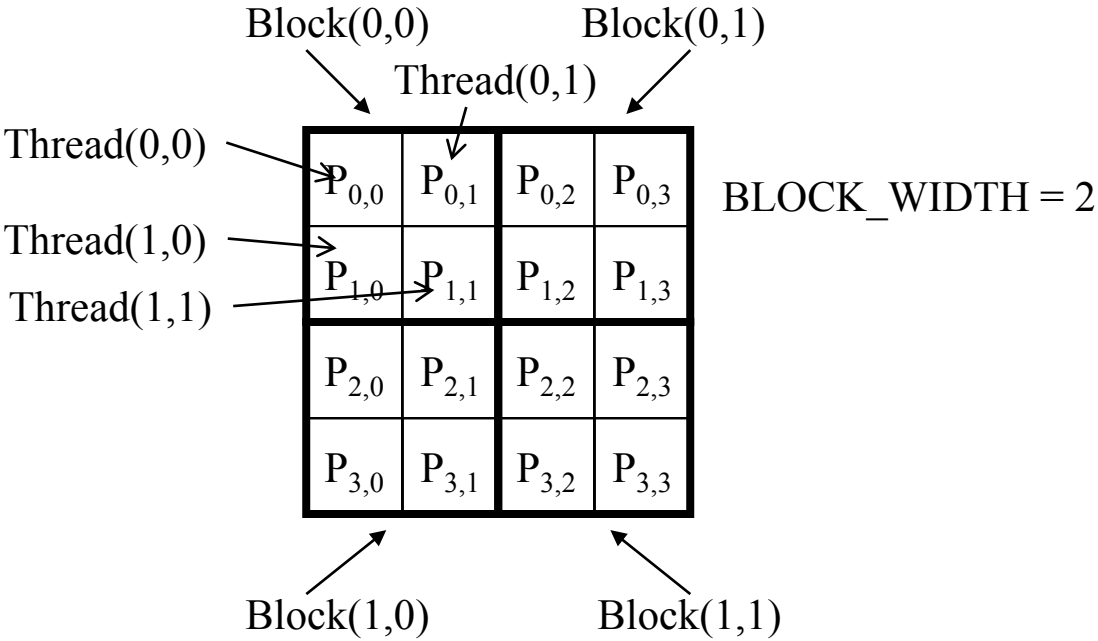
A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {

  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

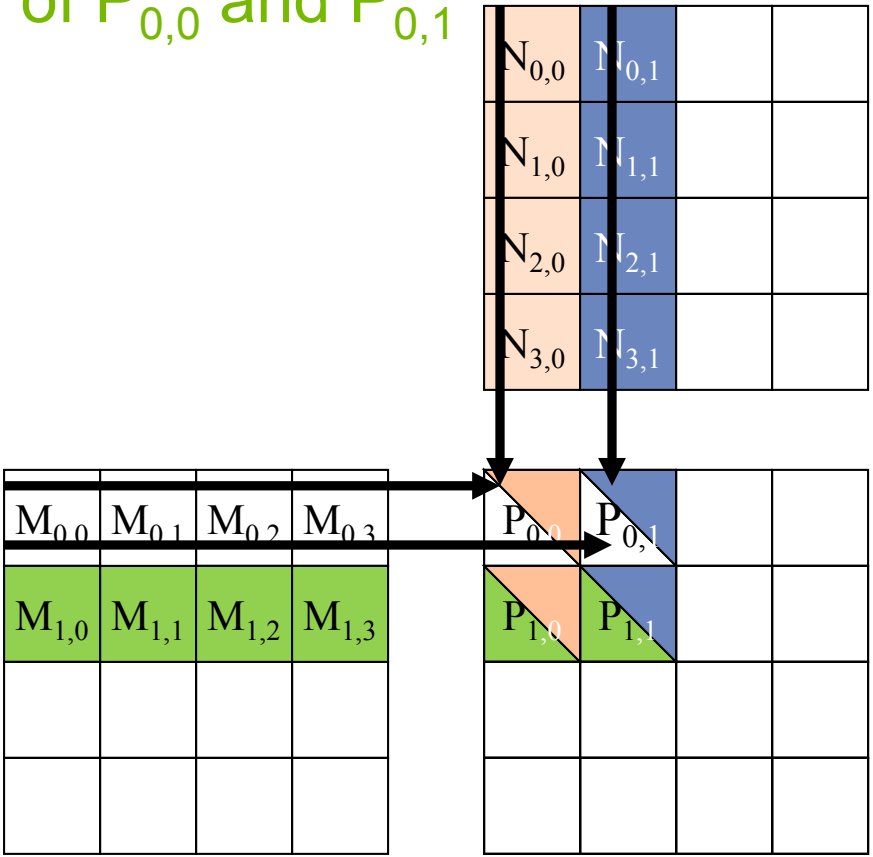
Example – Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {

  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

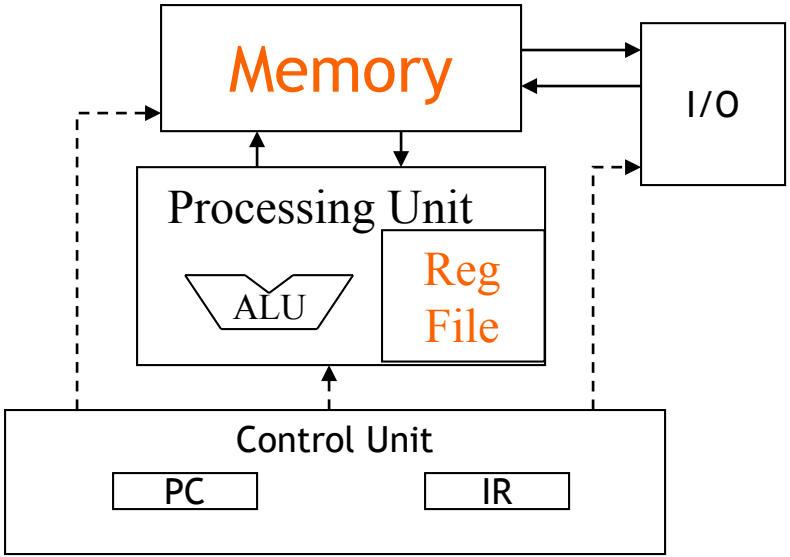
A Toy Example: Thread to P Data Mapping



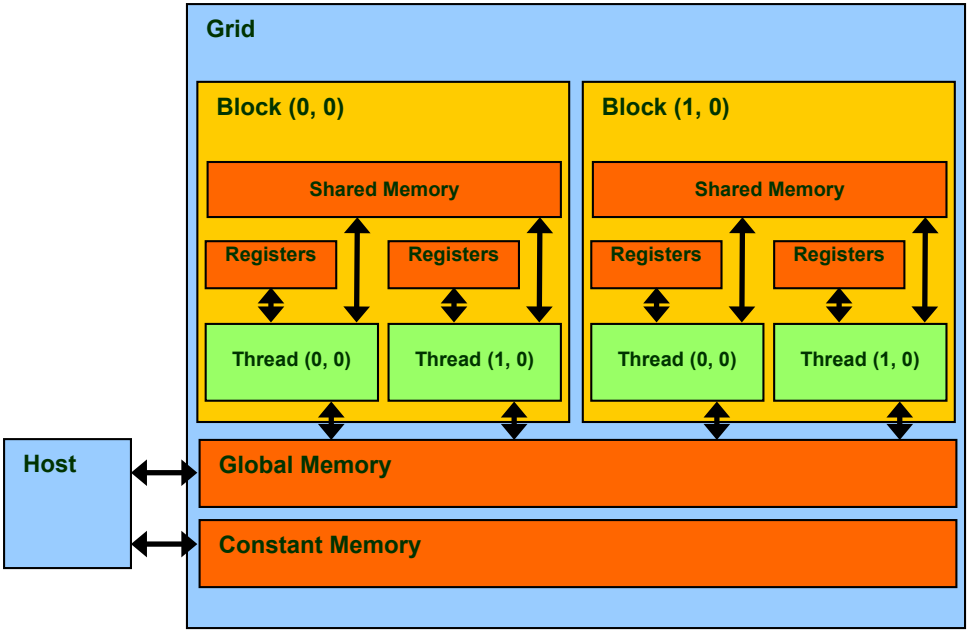
Calculation of $P_{0,0}$ and $P_{0,1}$



Memory and Registers in the Von-Neumann Model



Programmer View of CUDA Memories



Declaring CUDA Variables

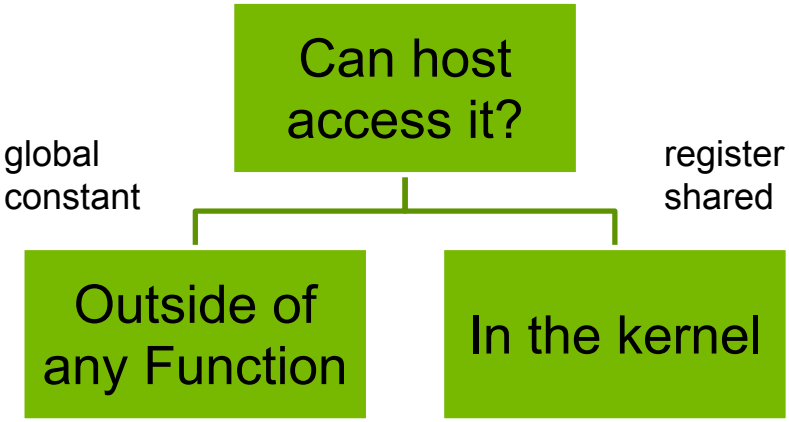
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- **__device__** is optional when used with **__shared__**, or **__constant__**
- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory

Example:
Shared Memory Variable Declaration

```
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{

   __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];

  …
}
```

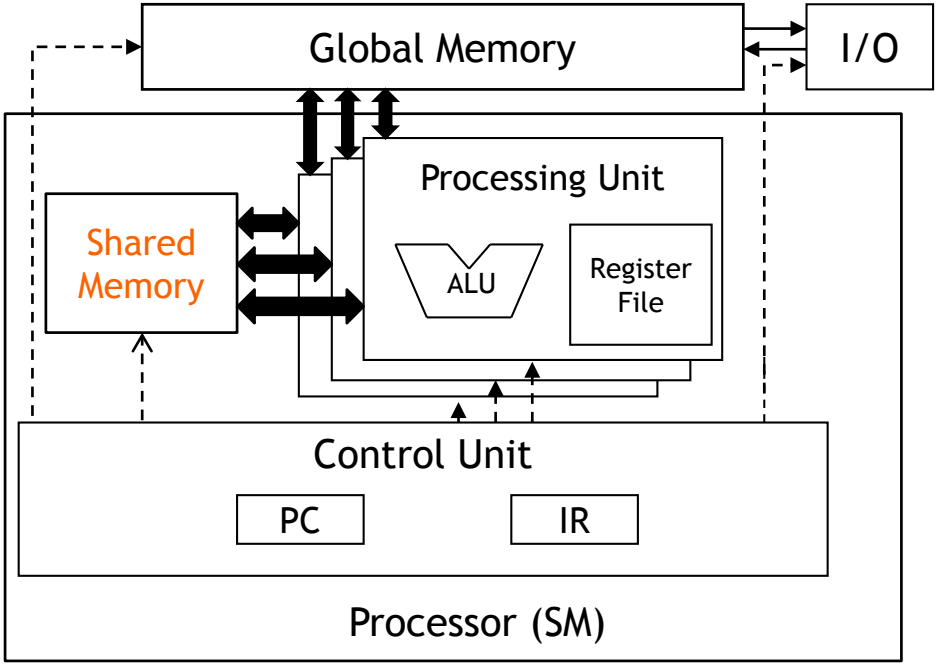
Where to Declare Variables?



Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
  - One in each SM
  - Accessed at much higher speed (in both latency and throughput) than global memory
  - Scope of access and sharing - thread blocks
  - Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
  - Accessed by memory load/store instructions
  - A form of scratchpad memory in computer architecture

Hardware View of CUDA Memories



GPU Teaching Kit

Accelerated Computing

Module 4.2 – Memory and Data Locality

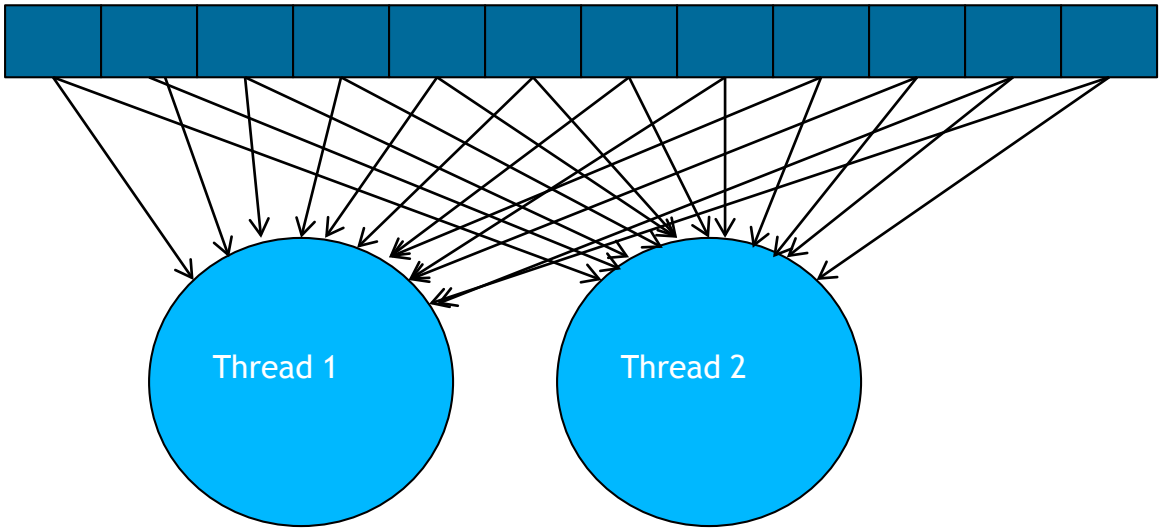
Tiled Parallel Algorithms

Objective

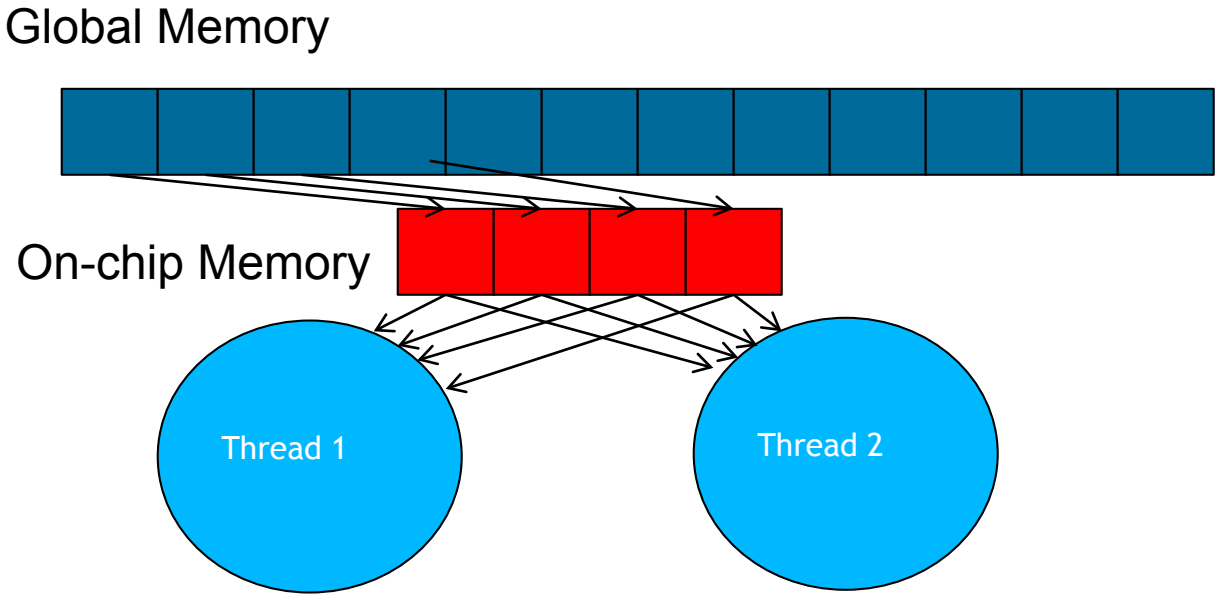
- To understand the motivation and ideas for tiled parallel algorithms
  - Reducing the limiting effect of memory bandwidth on parallel kernel performance
  - Tiled algorithms and barrier synchronization

Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory



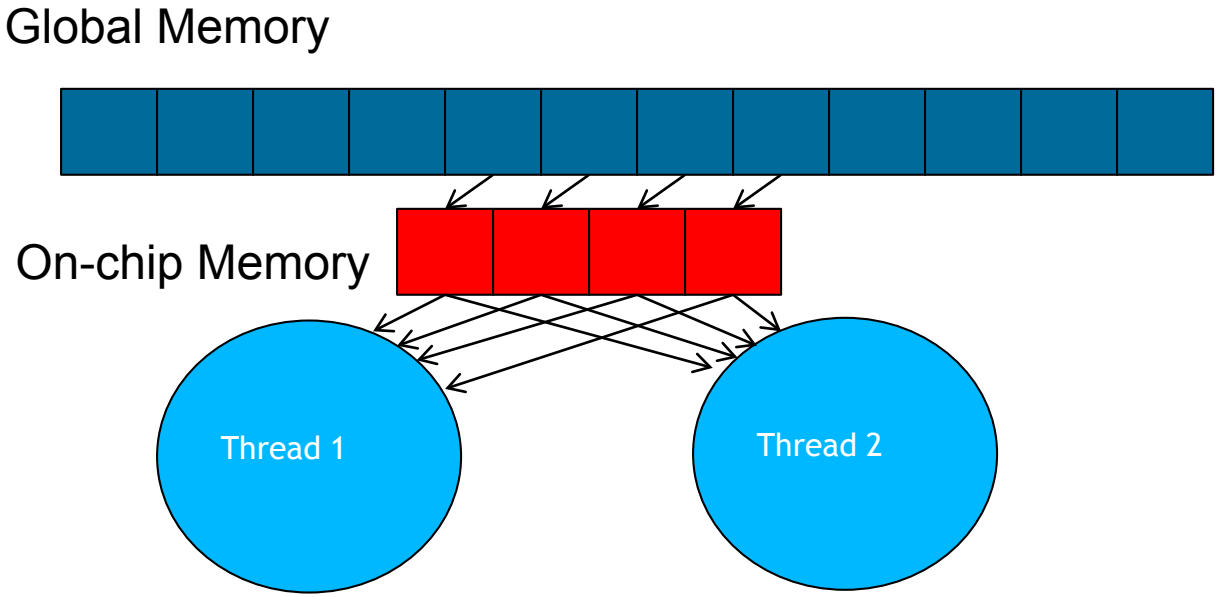
Tiling/Blocking - Basic Idea



Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Tiling/Blocking - Basic Idea



Basic Concept of Tiling

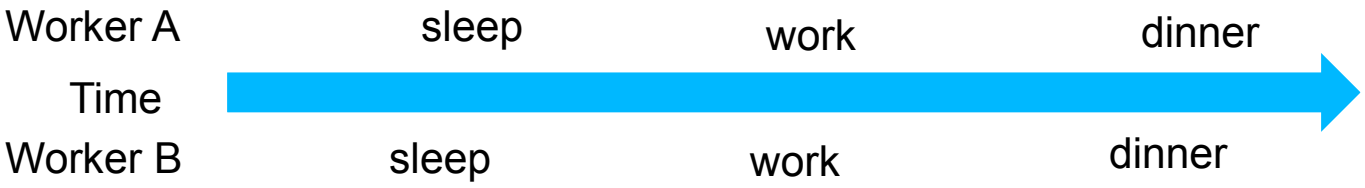
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Tiling for global memory accesses
    - drivers = threads accessing their memory data operands
    - cars = memory access requests

Some Computations are More Challenging to Tile

- Some carpools may be easier than others
  - Car pool participants need to have similar work schedule
  - Some vehicles may be more suitable for carpooling
- Similar challenges exist in tiling

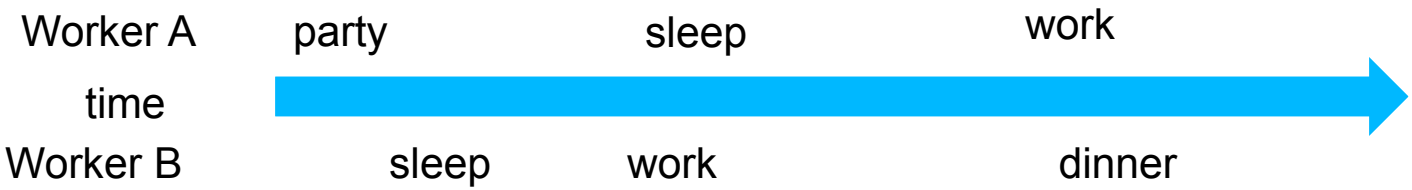
Carpools need synchronization.

– Good: when people have similar schedule



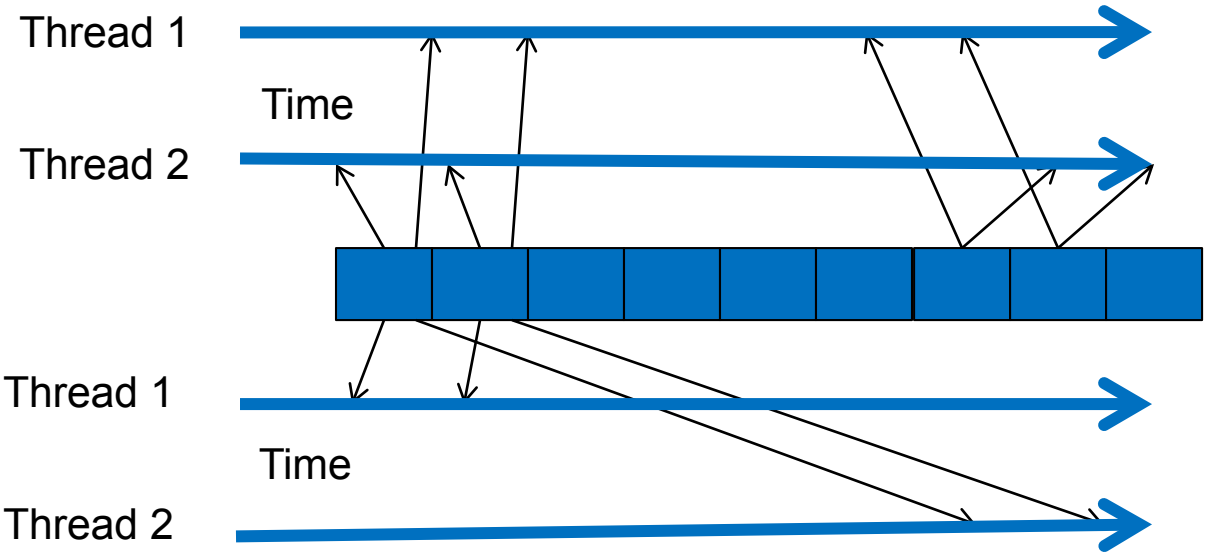
Carpools need synchronization.

– Bad: when people have very different schedule



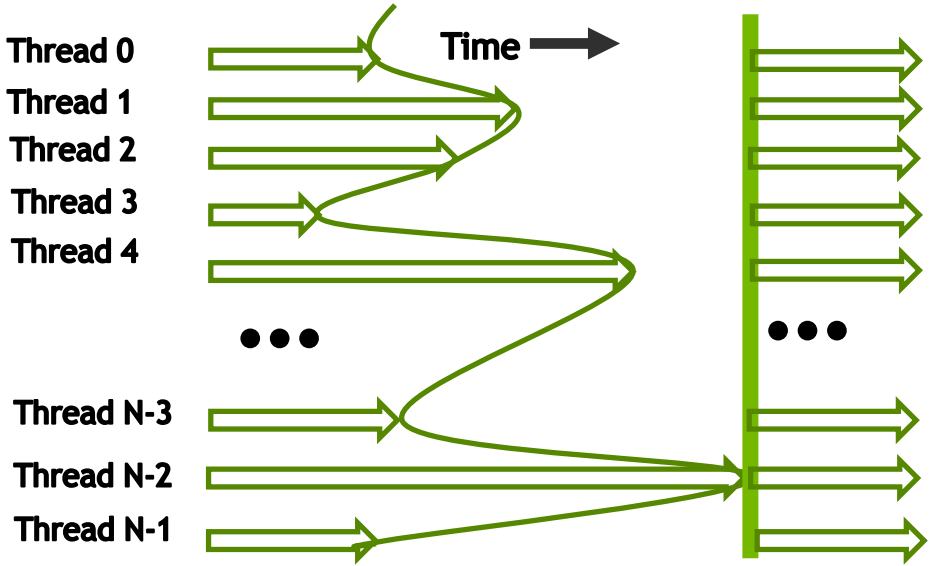
Same with Tiling

– Good: when threads have similar access timing



– Bad: when threads have very different timing

Barrier Synchronization for Tiling



Outline of Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile

GPU Teaching Kit

Accelerated Computing

Module 4.3 - Memory Model and Locality

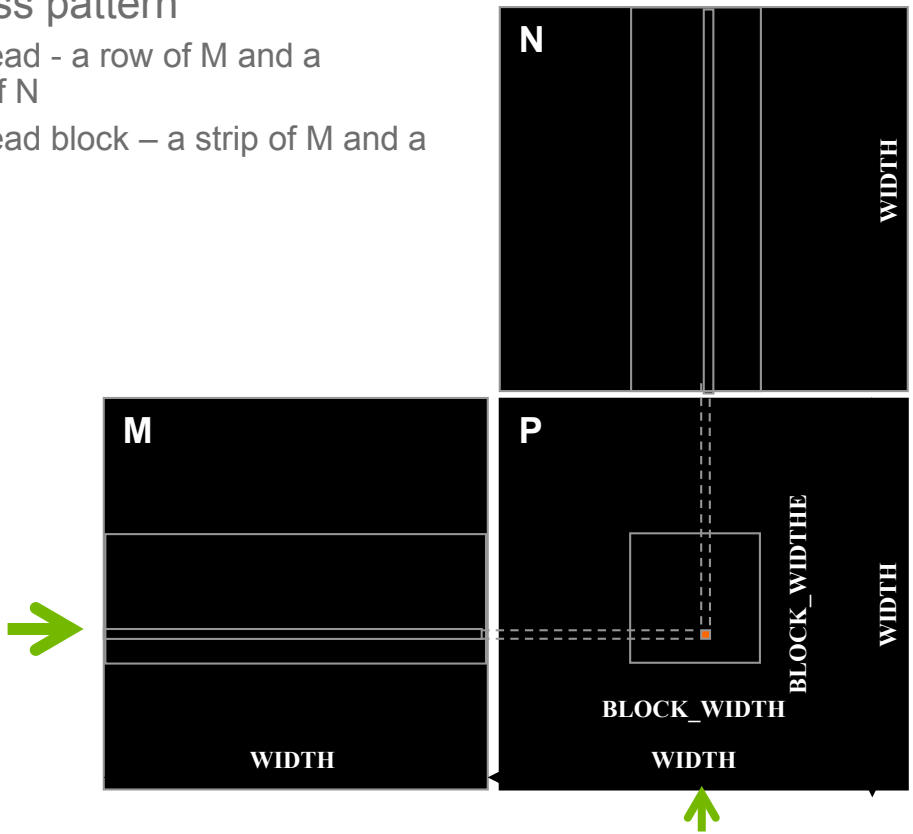
Tiled Matrix Multiplication

Objective

- To understand the design of a tiled parallel algorithm for matrix multiplication
  - Loading a tile
  - Phased execution
  - Barrier Synchronization

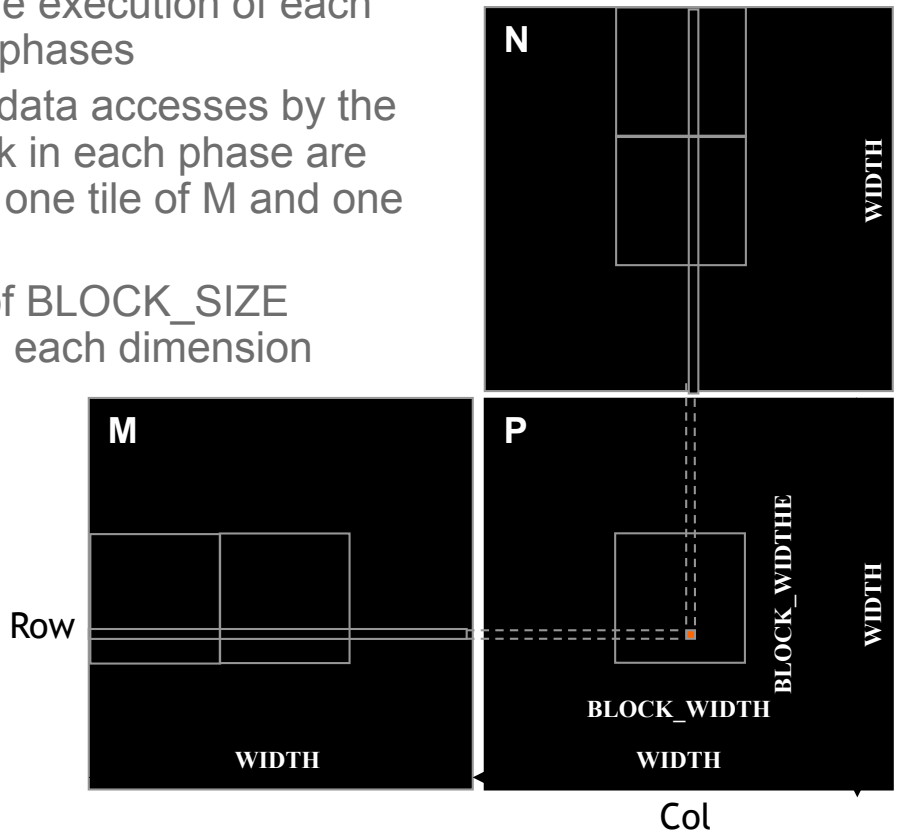
Matrix Multiplication

- Data access pattern
  - Each thread - a row of M and a column of N
  - Each thread block – a strip of M and a strip of N



Tiled Matrix Multiplication

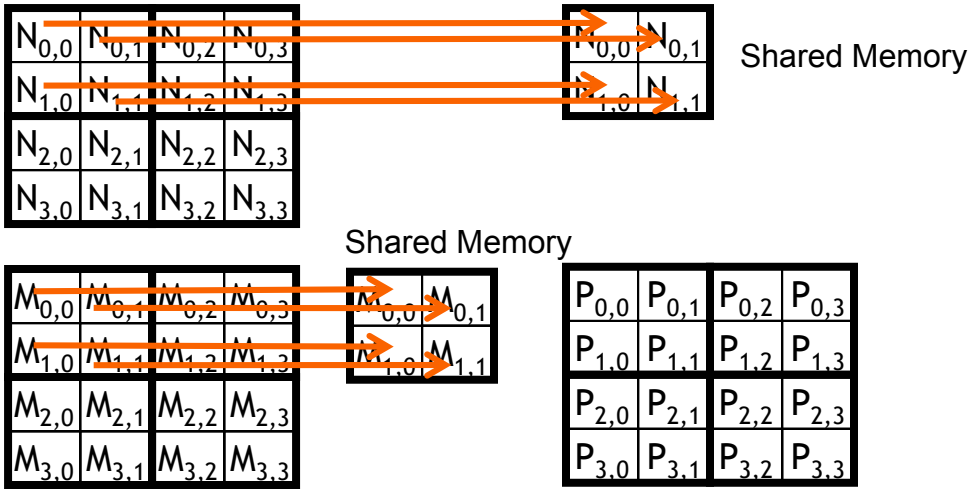
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



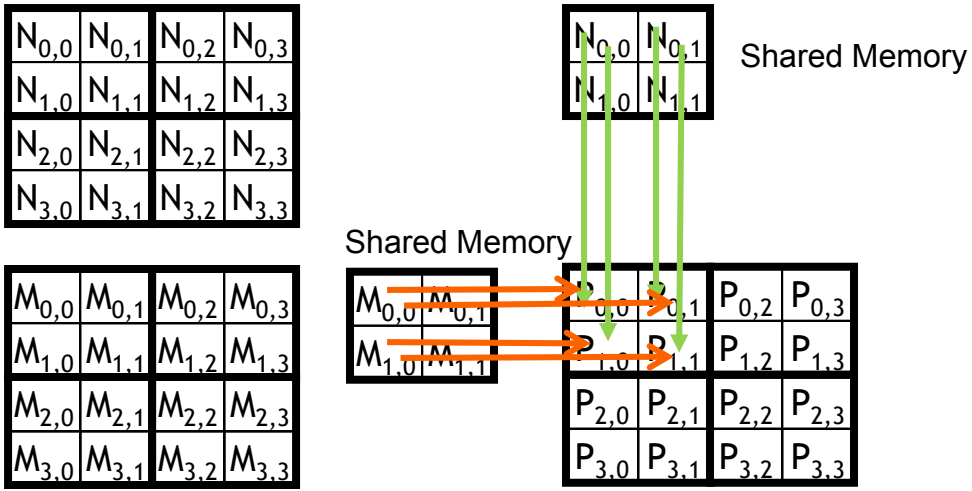
Loading a Tile

- All threads in a block participate
  - Each thread loads one M element and one N element in tiled code

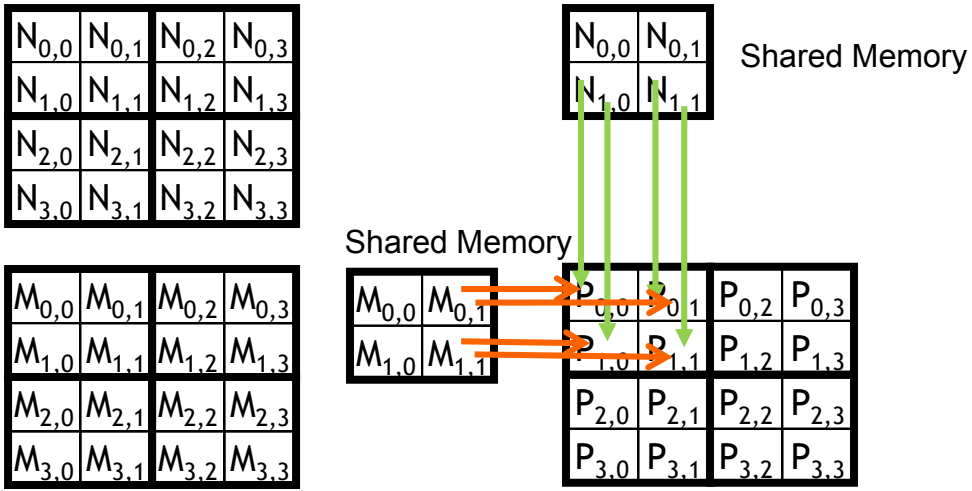
Phase 0 Load for Block (0,0)



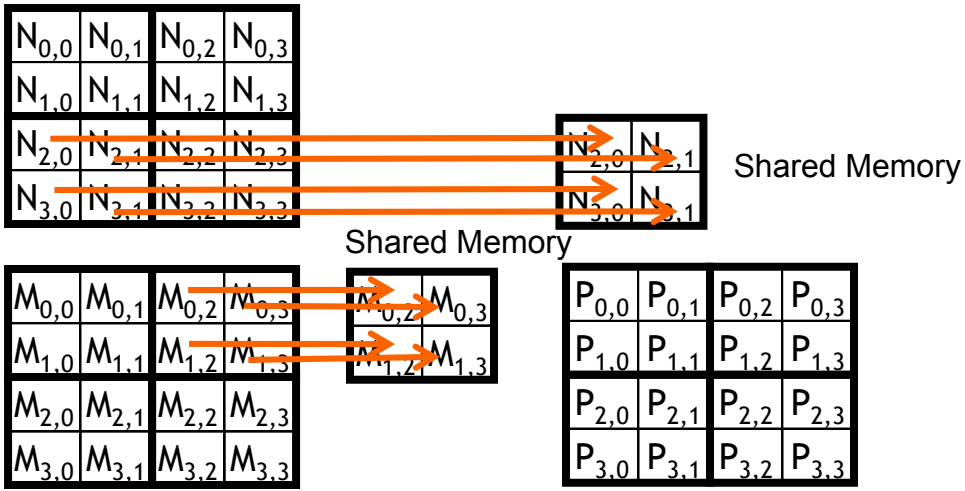
Phase 0 Use for Block (0,0) (iteration 0)



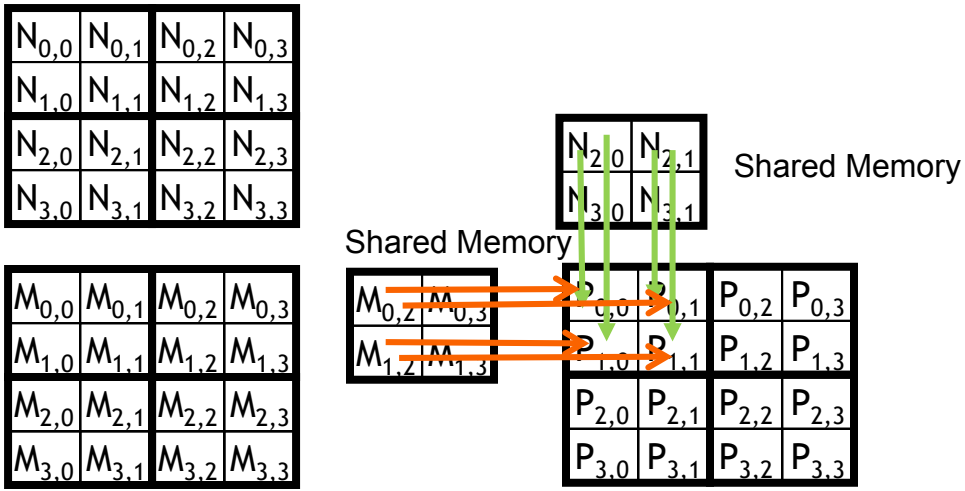
Phase 0 Use for Block (0,0) (iteration 1)



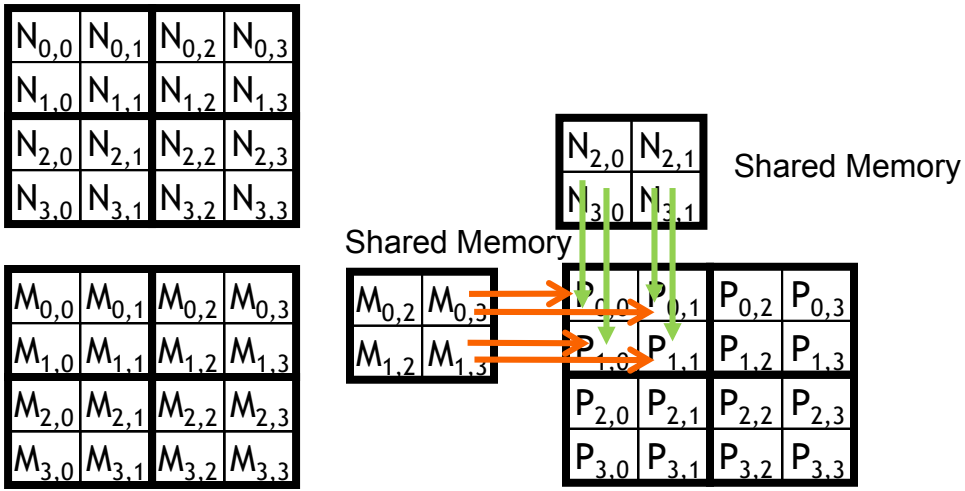
Phase 1 Load for Block (0,0)



Phase 1 Use for Block (0,0) (iteration 0)



Phase 1 Use for Block (0,0) (iteration 1)



Execution Phases of Toy Example

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

Execution Phases of Toy Example (cont.)

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Shared memory allows each value to be accessed by multiple threads

Barrier Synchronization

- Synchronize all threads in a block
  - __syncthreads()
- All threads in the same block must reach the __syncthreads() before any of the them can move on
- Best used to coordinate the phased execution tiled algorithms
  - To ensure that all elements of a tile are loaded at the beginning of a phase
  - To ensure that all elements of a tile are consumed at the end of a phase

GPU Teaching Kit

Accelerated Computing

Module 4.4 - Memory and Data Locality

Tiled Matrix Multiplication Kernel

Objective

- To learn to write a tiled matrix-multiplication kernel
  - Loading and using tiles for matrix multiplication
  - Barrier synchronization, shared memory
  - Resource Considerations
  - Assume that Width is a multiple of tile size for simplicity

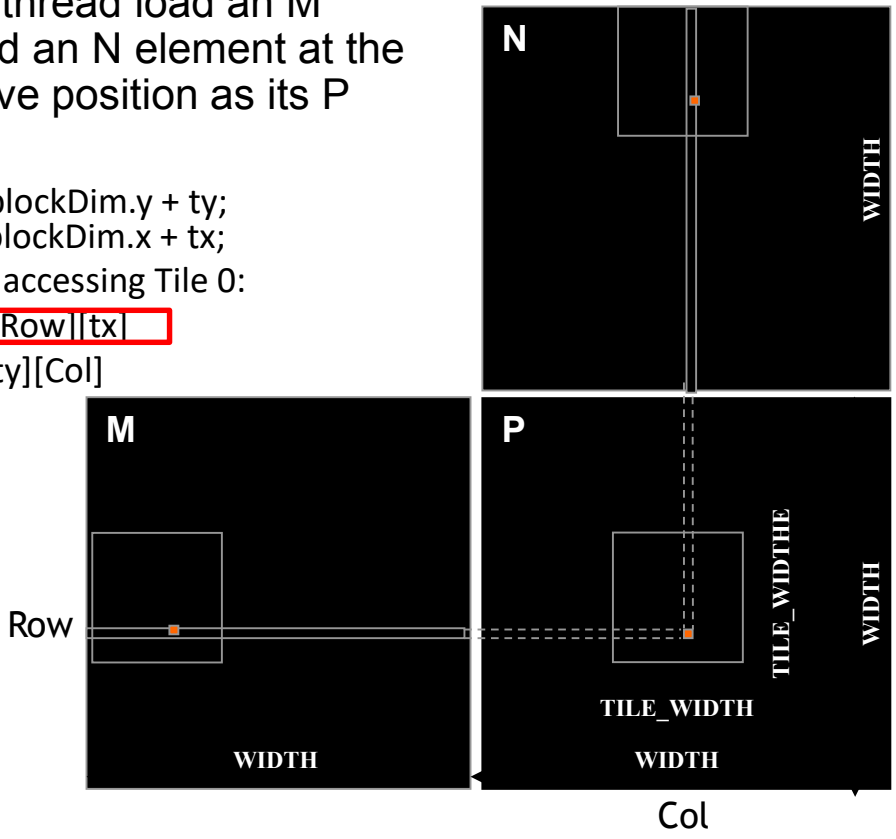
Loading Input Tile 0 of M (Phase 0)

– Have each thread load an M element and an N element at the same relative position as its P element.

```
int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

```
M[Row][tx]
N[ty][Col]
```



Loading Input Tile 0 of N (Phase 0)

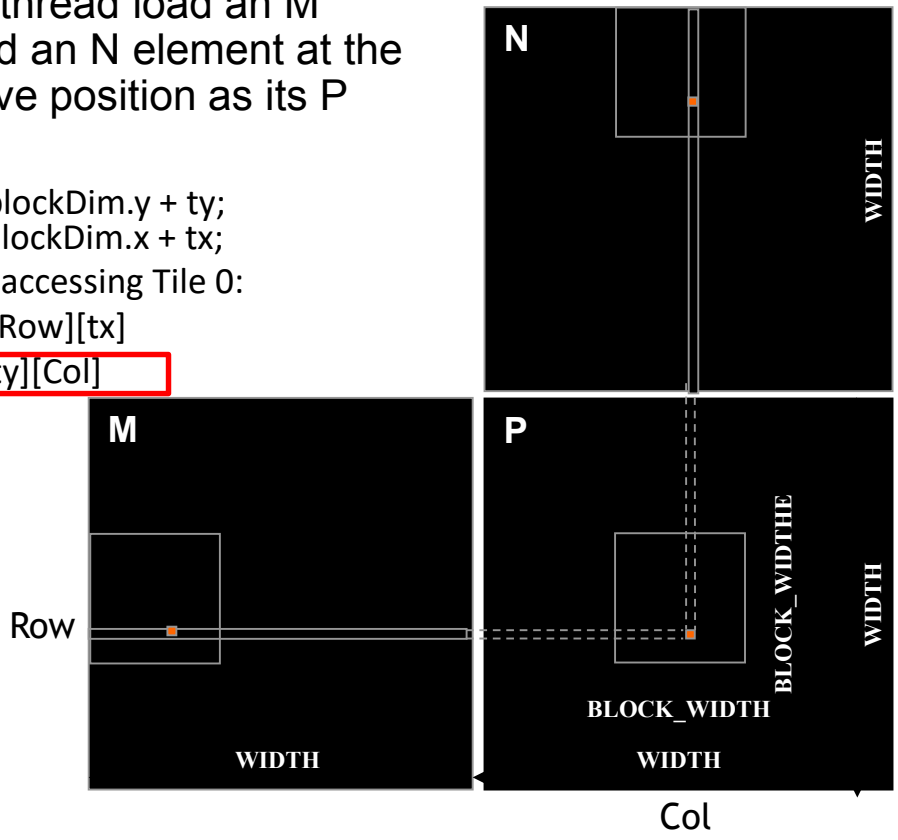
– Have each thread load an M element and an N element at the same relative position as its P element.

```
int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

M[Row][tx]

N[ty][Col]

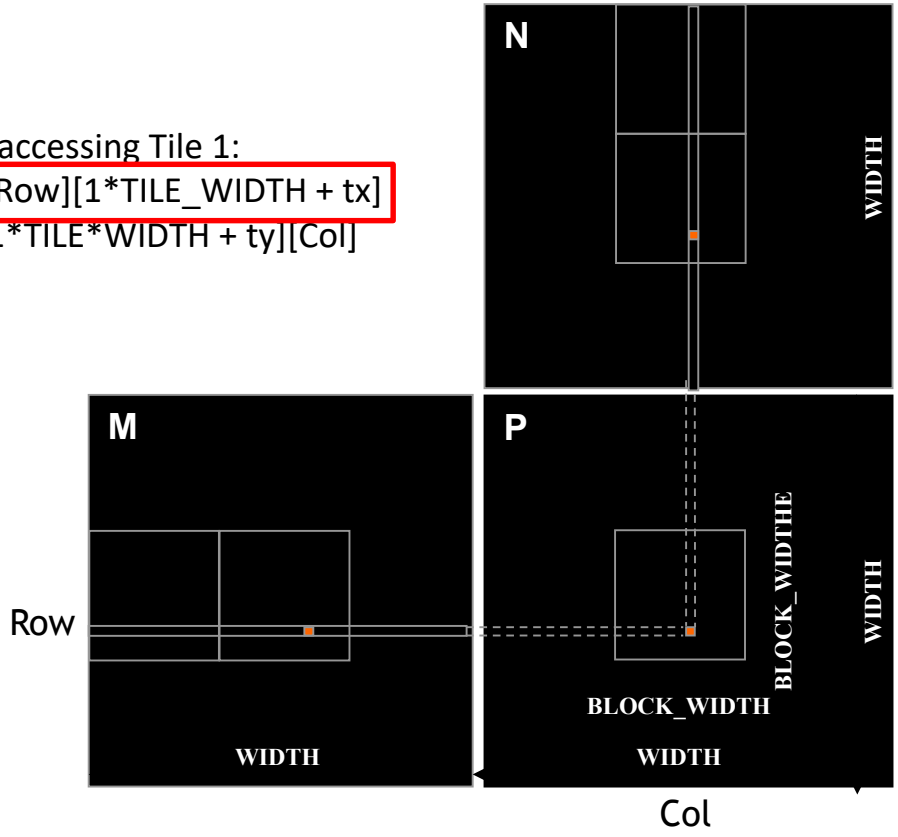


Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE*WIDTH + ty][Col]

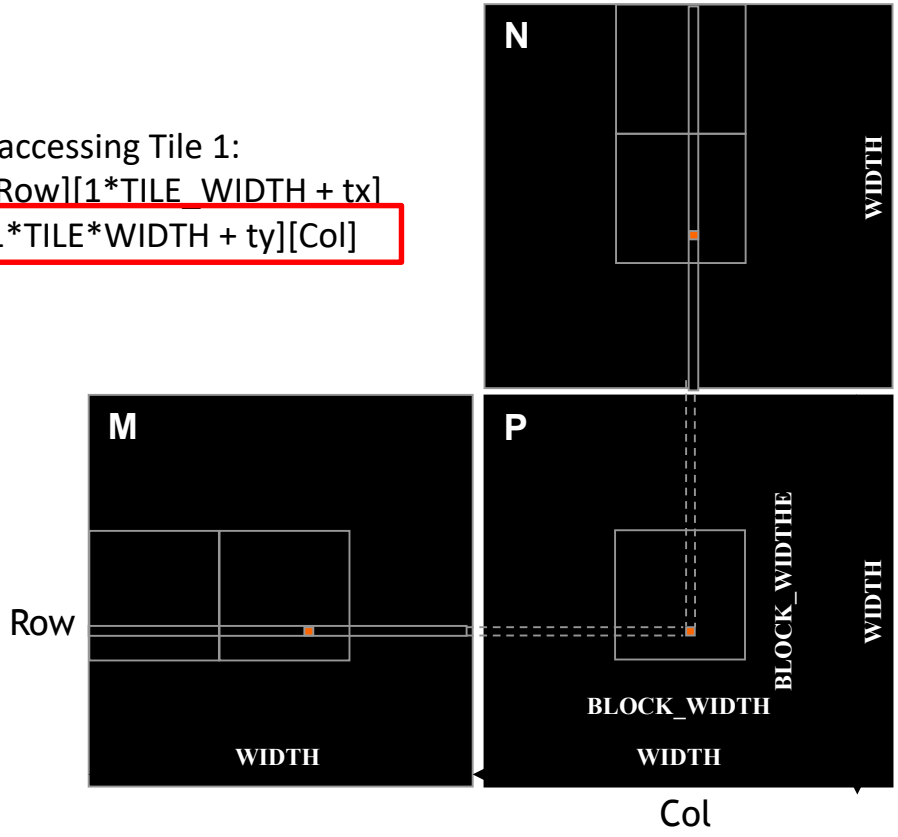


Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE*WIDTH + ty][Col]



M and N are dynamically allocated - use 1D indexing

M[Row][p*TILE_WIDTH+tx]

➡ M[Row*Width + p*TILE_WIDTH + tx]

N[p*TILE_WIDTH+ty][Col]

➡ N[(p*TILE_WIDTH+ty)*Width + Col]

where p is the sequence number of the current phase

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

// Loop over the M and N tiles required to compute the P element
for (int p = 0; p < n/TILE_WIDTH; ++p) {
   // Collaborative loading of M and N tiles into shared memory
   ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
   ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
   __syncthreads();

   for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
   __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Tile (Thread Block) Size Considerations

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - TILE_WIDTH of 32 gives 32*32 = 1024 threads

- For 16, in each phase, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations. (16 floating-point operations for each memory load)

- For 32, in each phase, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations. (32 floating-point operation for each memory load)

Shared Memory and Threading

- For an SM with 16KB shared memory
  - Shared memory size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - For 16KB shared memory, one can potentially have up to 8 thread blocks executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4 Byte= 8K Byte shared memory usage per thread block, allowing 2 thread blocks active at the same time
    - However, in a GPU where the thread count is limited to 1536 threads per SM, the number of blocks per SM is reduced to one!
- Each __syncthread() can reduce the number of active threads for a block
  - More thread blocks can be advantageous

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

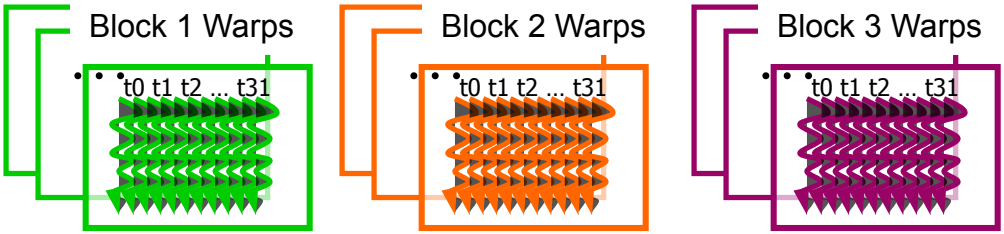
Module 5.1 – Thread Execusion Efficiency

Warps and SIMD Hardware

Objective

- To understand how CUDA threads execute on SIMD Hardware
  - Warp partitioning
  - SIMD Hardware
  - Control divergence

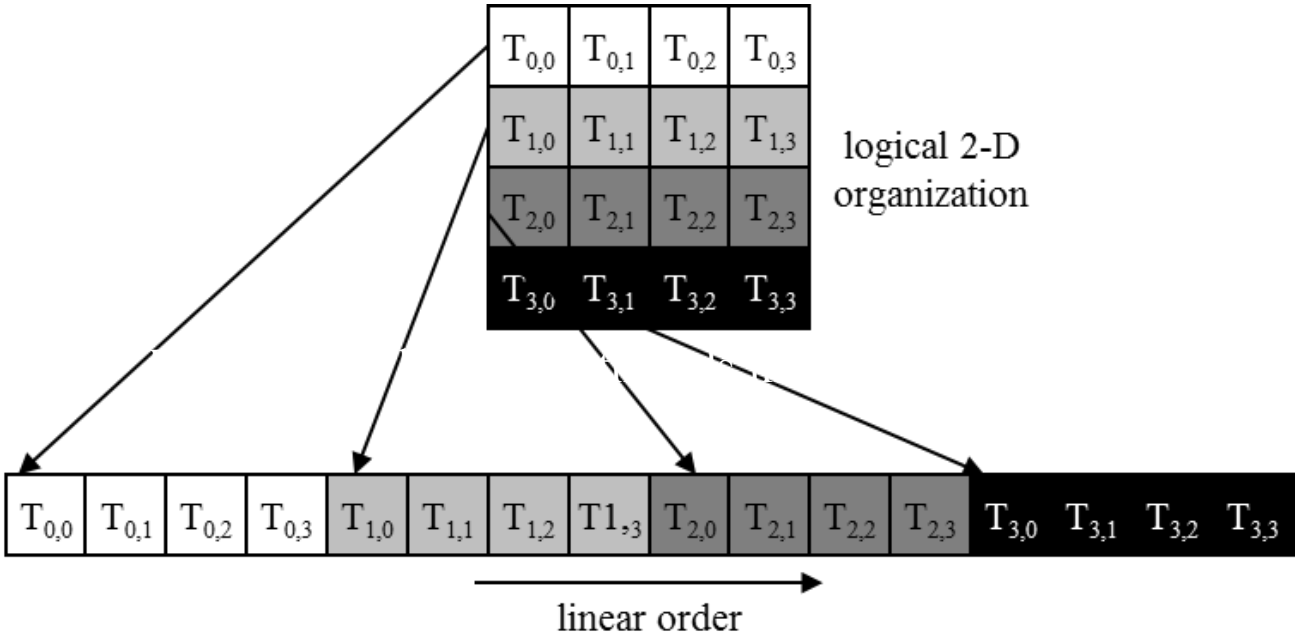
Warps as Scheduling Units



- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations

Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
  - In x-dimension first, y-dimension next, and z-dimension last

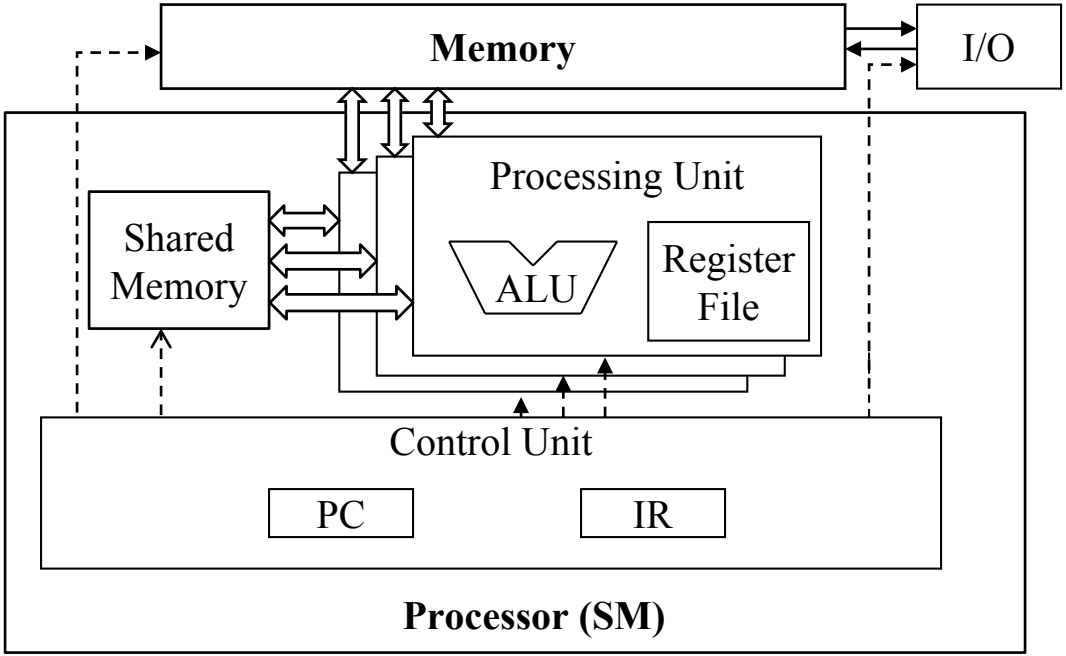


Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).

SMs are SIMD Processors

- Control unit for instruction fetch, decode, and control is shared among multiple processing units
  - Control overhead is minimized (Module 1)



SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time

- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times

Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others

- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

- Divergence can arise when branch or loop condition is a function of thread indices
- Example kernel statement with divergence:
  - if (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - If (blockIdx.x > 2) { }
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C,
  int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
   if(i<n) C[i] = A[i] + B[i];
}
```

Analysis for vector size of 1,000 elements

- Assume that block size is 256 threads
  - 8 warps in each block
- All threads in Blocks 0, 1, and 2 are within valid range
  - i values from 0 to 767
  - There are 24 warps in these three blocks, none will have control divergence
- Most warps in Block 3 will not control divergence
  - Threads in the warps 0-6 are all within valid range, thus no control divergence
- One warp in Block 3 will have control divergence
  - Threads with i values 992-999 will all be within valid range
  - Threads with i values of 1000-1023 will be outside valid range
- Effect of serialization on control divergence will be small
  - 1 out of 32 warps has control divergence
  - The impact on performance will likely be less than 3%

GPU Teaching Kit

Accelerated Computing

Module 5.2 – Thread Execusion Efficiency

Performance Impact of Control Divergence

Objective

- To learn to analyze the performance impact of control divergence
  - Boundary condition checking
  - Control divergence is data-dependent

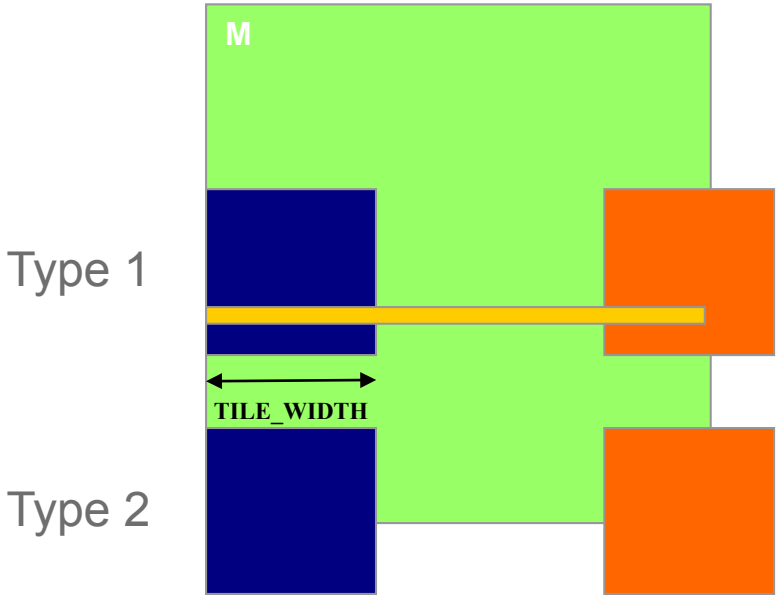
Performance Impact of Control Divergence

- Boundary condition checks are vital for complete functionality and robustness of parallel code
  - The tiled matrix multiplication kernel has many boundary condition checks
  - The concern is that these checks may cause significant performance degradation
  - For example, see the tile loading code below:

```
if(Row < Width && t * TILE_WIDTH+tx < Width) {
   ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
} else {
 ds_M[ty][tx] = 0.0;
}

if (p*TILE_WIDTH+ty < Width && Col < Width) {
   ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
} else {
   ds_N[ty][tx] = 0.0;
}
```

Two types of blocks in loading M Tiles

- 1. Blocks whose tiles are all within valid range until the last phase.
- 2. Blocks whose tiles are partially outside the valid range all the way



Analysis of Control Divergence Impact

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each thread will go through 7 phases (ceiling of 100/16)

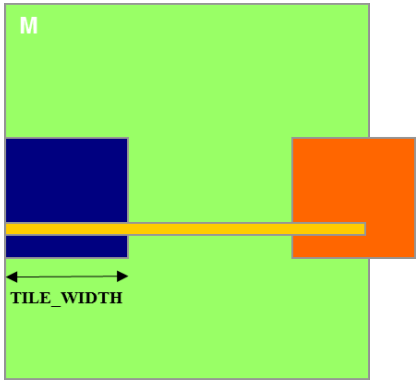
- There are 49 thread blocks (7 in each dimension)

Control Divergence in Loading M Tiles

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of 100/16)

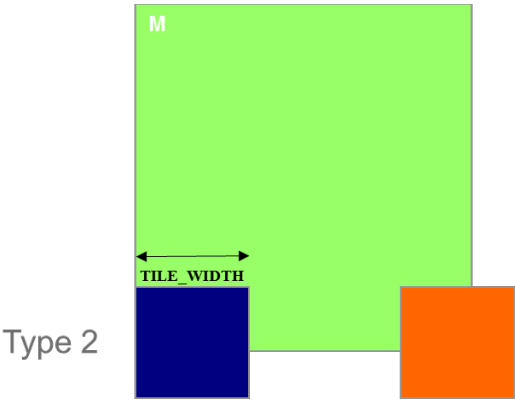
- There are 42 (6*7) Type 1 blocks, with a total of 336 (8*42) warps
- They all have 7 phases, so there are 2,352 (336*7) warp-phases
- The warps have control divergence only in their last phase
- 336 warp-phases have control divergence

Type 1



Control Divergence in Loading M Tiles (Type 2)

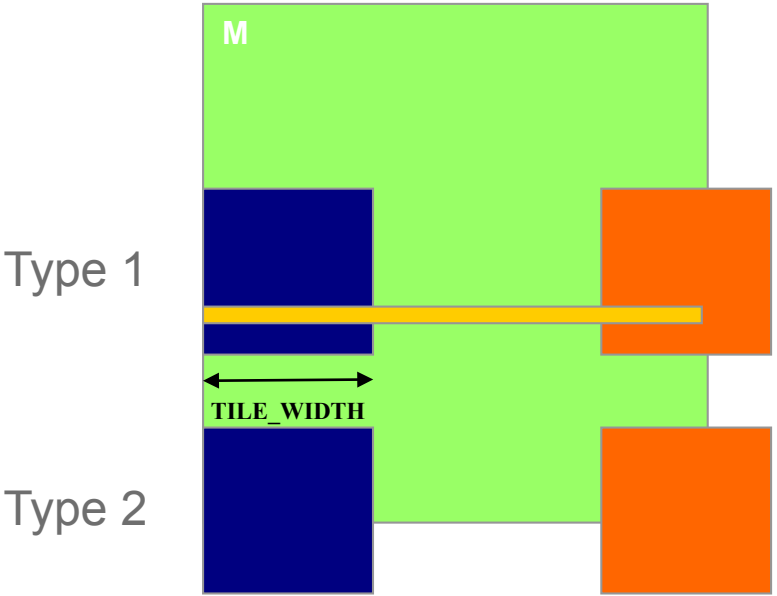
- Type 2: the 7 block assigned to load the bottom tiles, with a total of 56 (8*7) warps
- They all have 7 phases, so there are 392 (56*7) warp-phases
- The first 2 warps in each Type 2 block will stay within the valid range until the last phase
- The 6 remaining warps stay outside the valid range
- So, only 14 (2*7) warp-phases have control divergence



Type 2

Overall Impact of Control Divergence

- Type 1 Blocks: 336 out of 2,352 warp-phases have control divergence
- Type 2 Blocks: 14 out of 392 warp-phases have control divergence
- The performance impact is expected to be less than 12% (350/2,944 or (336+14)/(2352+14))



Additional Comments

- The calculation of impact of control divergence in loading N tiles is somewhat different and is left as an exercise
- The estimated performance impact is data dependent.
  - For larger matrices, the impact will be significantly smaller
- In general, the impact of control divergence for boundary condition checking for large input data sets should be insignificant
  - One should not hesitate to use boundary checks to ensure full functionality
- The fact that a kernel is full of control flow constructs does not mean that there will be heavy occurrence of control divergence
- We will cover some algorithm patterns that naturally incur control divergence (such as parallel reduction) in the Parallel Algorithm Patterns modules

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 6.1 – Memory Access Performance

DRAM Bandwidth

Objective

- To learn that memory bandwidth is a first-order performance factor in a massively parallel processor
  - DRAM bursts, banks, and channels
  - All concepts are also applicable to modern multicore processors

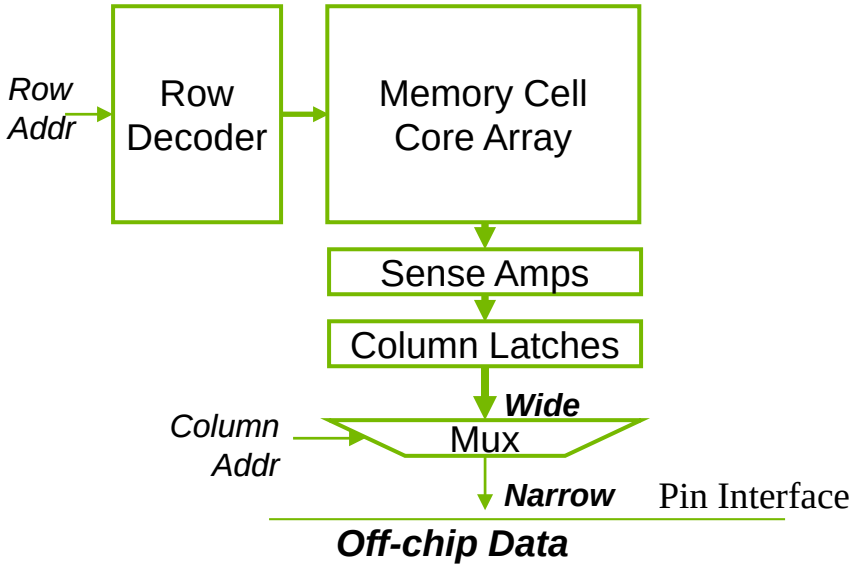
Global Memory (DRAM) Bandwidth

– Ideal

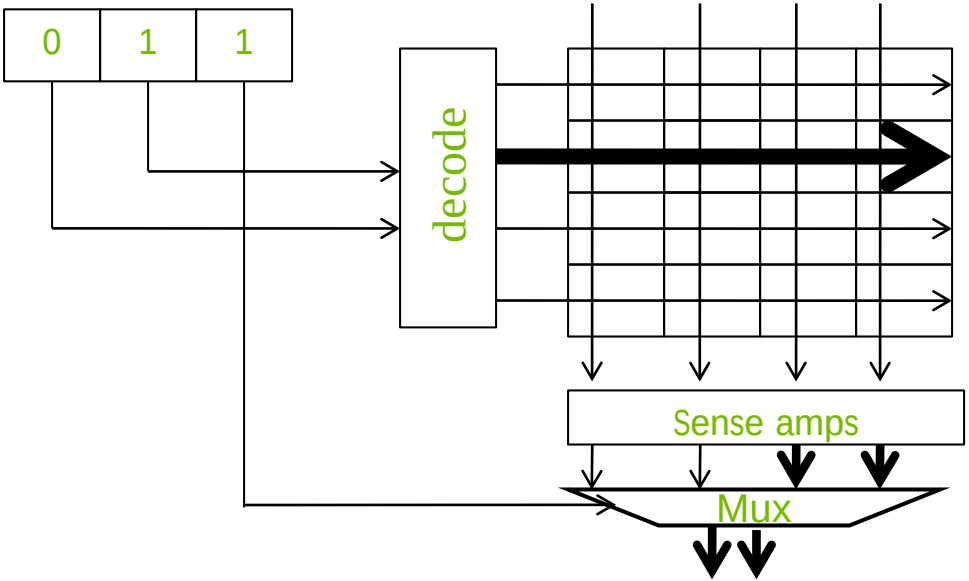
– Reality

DRAM Core Array Organization

- Each DRAM core array has about 16M bits
- Each bit is stored in a tiny capacitor made of one transistor

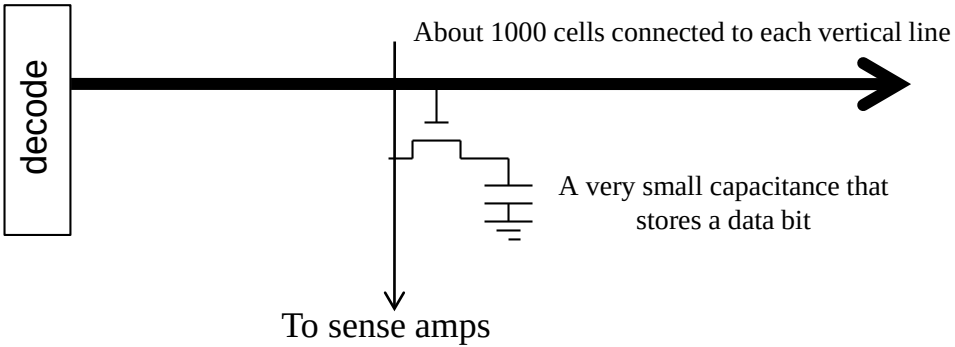


A very small (8x2-bit) DRAM Core Array



DRAM Core Arrays are Slow

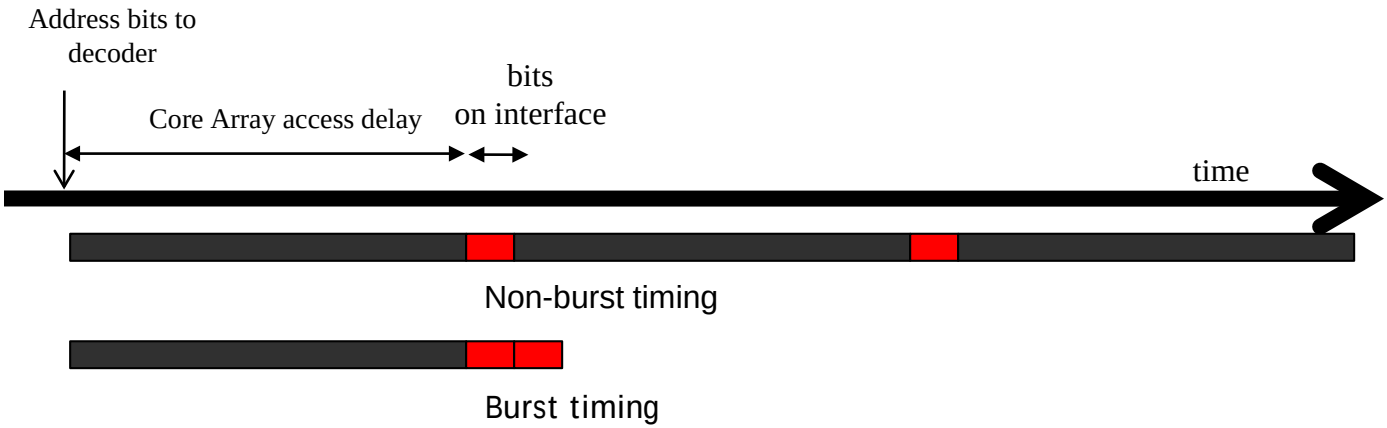
- Reading from a cell in the core array is a very slow process
  - DDR: Core speed = ½ interface speed
  - DDR2/GDDR3: Core speed = ¼ interface speed
  - DDR3/GDDR4: Core speed = ⅛ interface speed
  - … likely to be worse in the future



DRAM Bursting

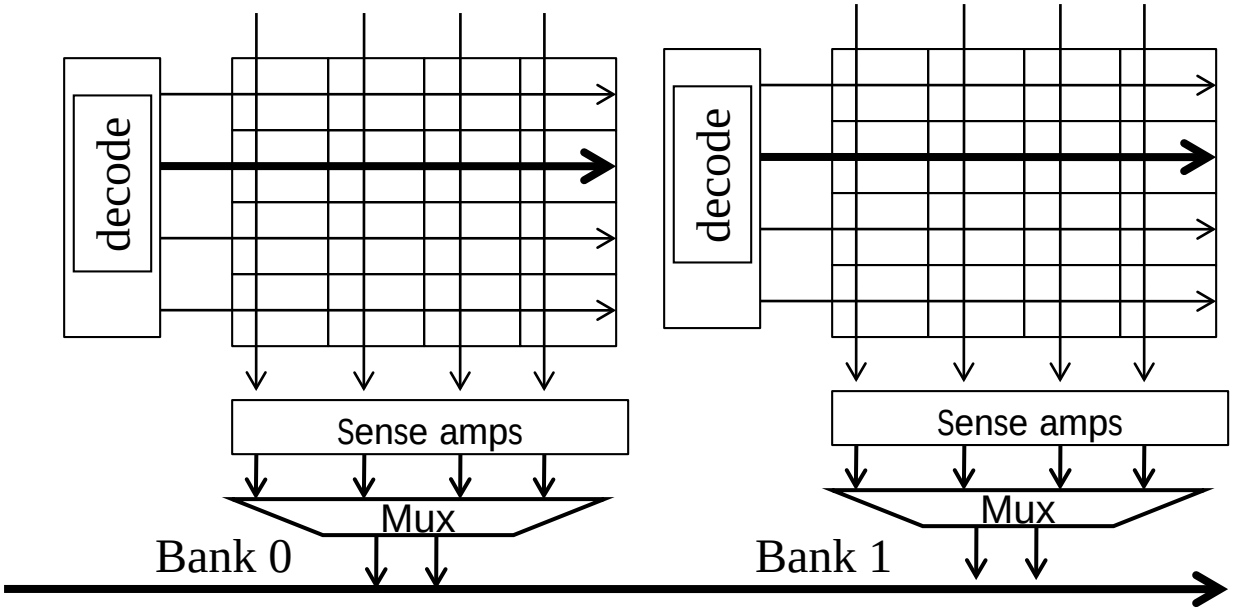
- For DDR{2,3} SDRAM cores clocked at 1/N speed of the interface:
  - Load (N × interface width) of DRAM bits from the same row at once to an internal buffer, then transfer in N steps at interface speed
  - DDR3/GDDR4: buffer width = 8X interface width

DRAM Bursting Timing Example



Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

Multiple DRAM Banks



DRAM Bursting with Banking

Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time

GPU off-chip memory subsystem

- NVIDIA GTX280 GPU:
  - Peak global memory bandwidth = 141.7GB/s
- Global memory (GDDR3) interface @ 1.1GHz
  - (Core speed @ 276Mhz)
  - For a typical 64-bit interface, we can sustain only about 17.6 GB/s (Recall DDR - 2 transfers per clock)
  - We need a lot more bandwidth (141.7 GB/s) – thus 8 memory channels

GPU Teaching Kit

Accelerated Computing

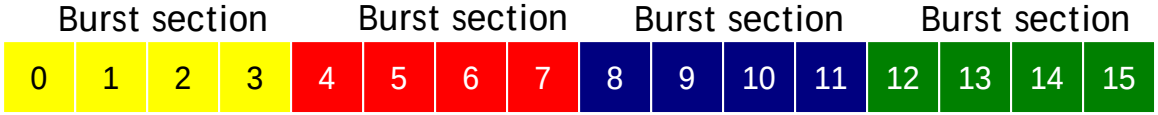
Lecture 6.2 – Performance Considerations

Memory Coalescing in CUDA

Objective

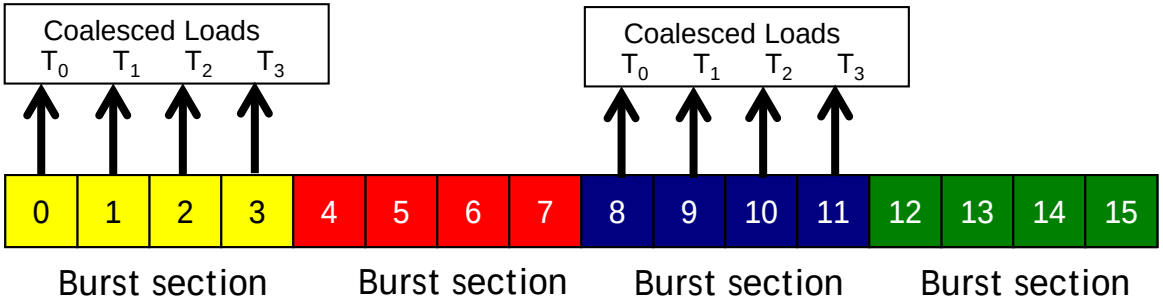
- To learn that memory coalescing is important for effectively utilizing memory bandwidth in CUDA
  - Its origin in DRAM burst
  - Checking if a CUDA memory access is coalesced
  - Techniques for improving memory coalescing in CUDA code

DRAM Burst – A System View



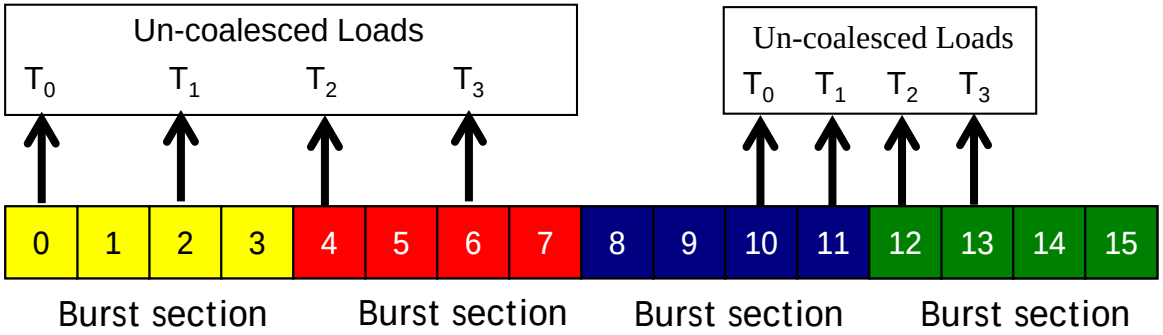
- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

Memory Coalescing



– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

Un-coalesced Accesses

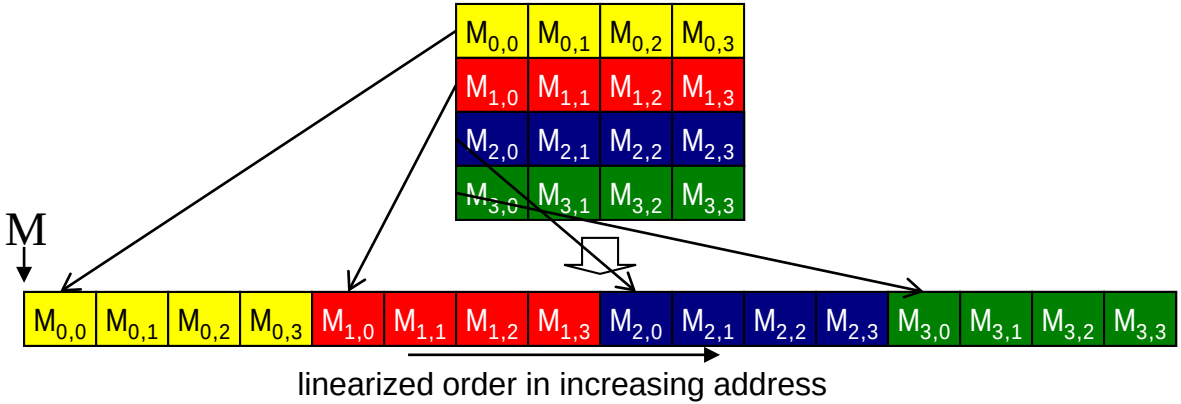


- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

How to judge if an access is coalesced?

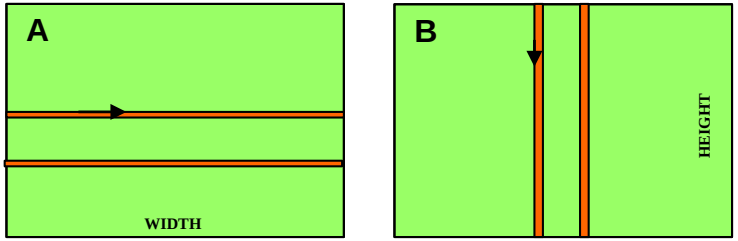
- Accesses in a warp are to consecutive locations if the index in an array access is in the form of
  - A[(expression with terms independent of threadIdx.x) + threadIdx.x];

A 2D C Array in Linear Memory Space



Two Access Patterns of Basic Matrix Multiplication

Thread 1
Thread 2



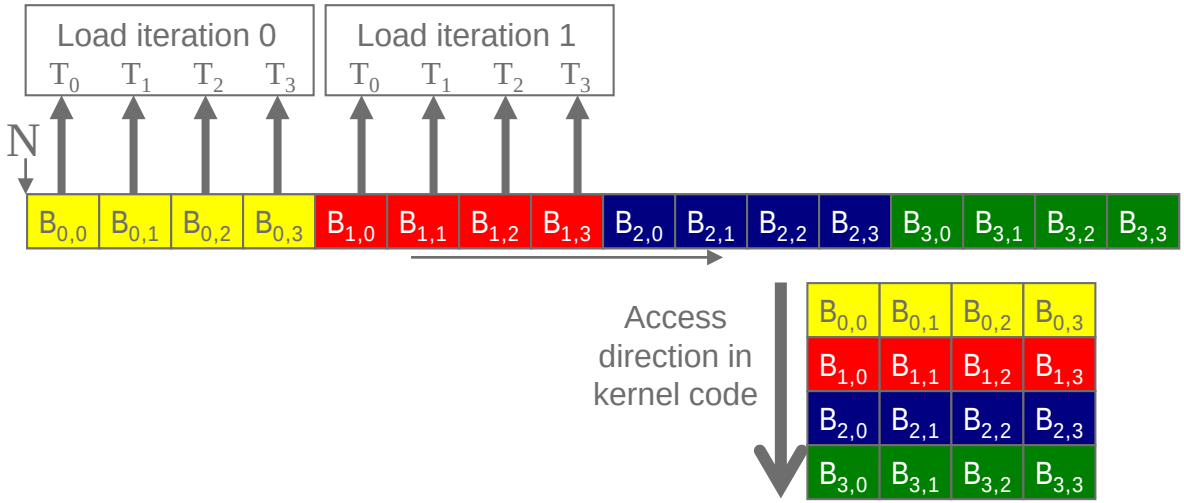
`A[Row*n+i]` `B[i*k+Col]`

i is the loop counter in the inner product loop of the kernel code

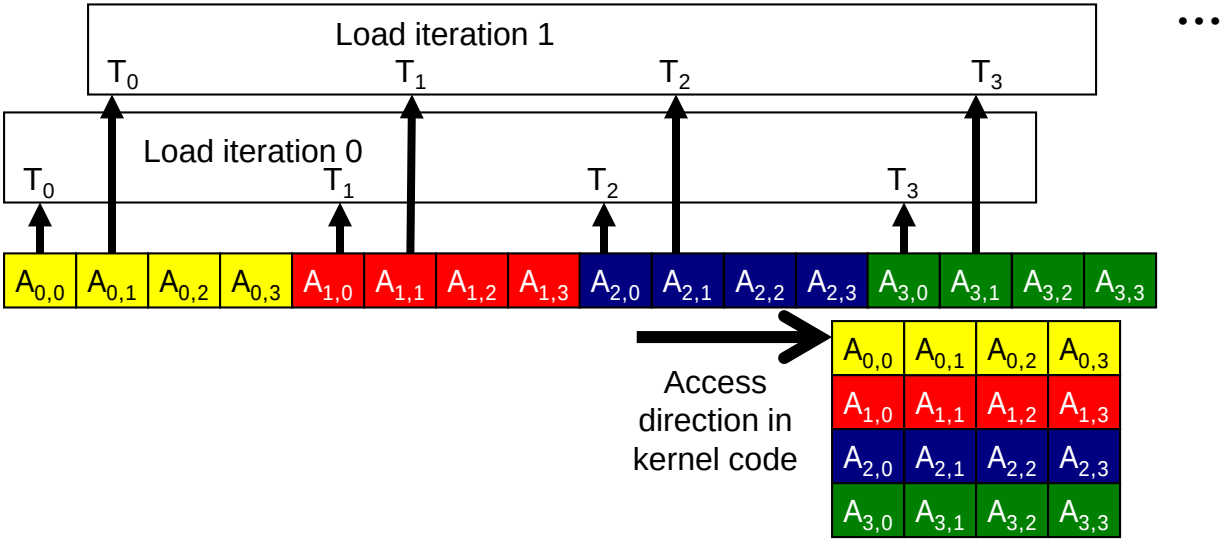
A is m × n, B is n × k

Col = blockIdx.x*blockDim.x + threadIdx.x

B accesses are coalesced



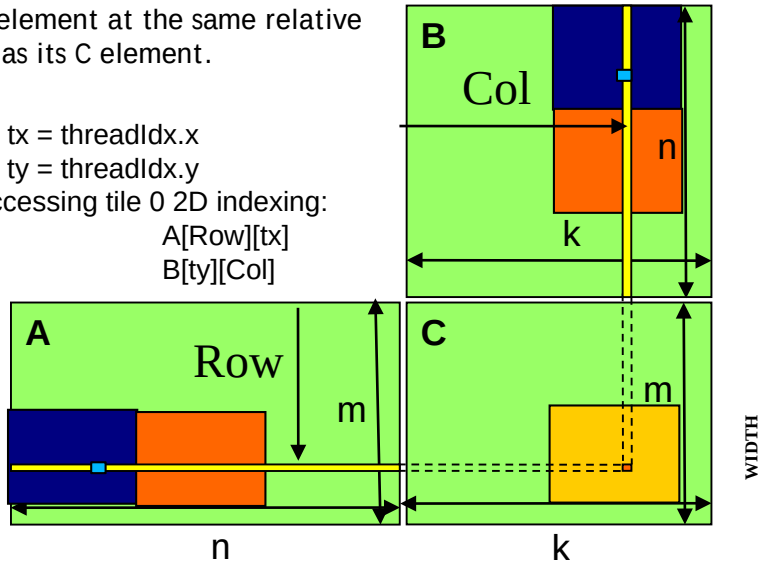
A Accesses are Not Coalesced



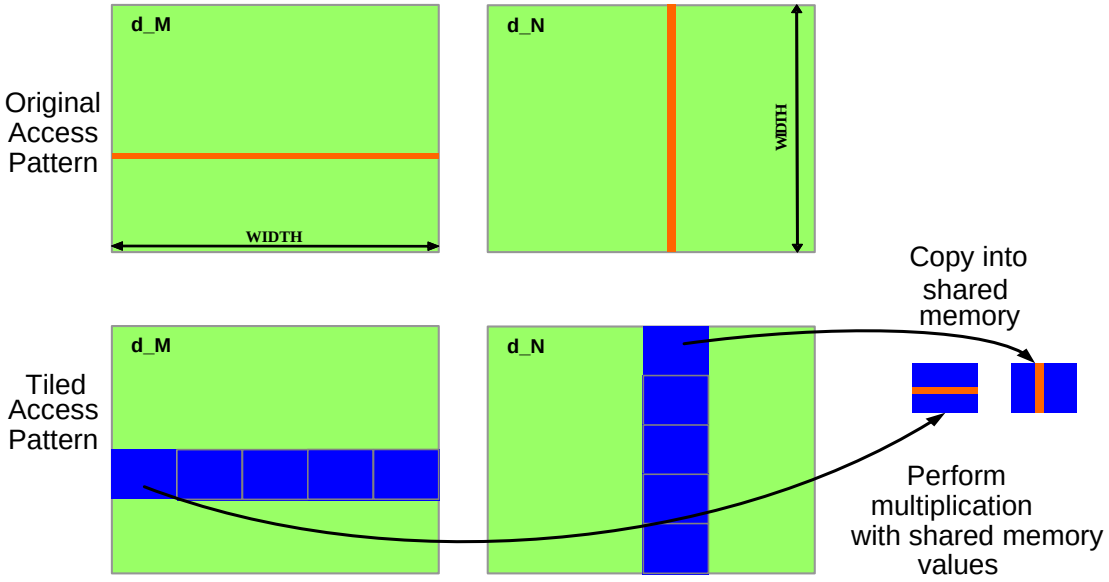
Loading an Input Tile

Have each thread load an A element and a B element at the same relative position as its C element.

int tx = threadIdx.x
int ty = threadIdx.y
Accessing tile 0 2D indexing:
A[Row][tx]
B[ty][Col]



Corner Turning



GPU Teaching Kit

GPU Teaching Kit

Accelerated Computing

Module 14 – Efficient Host-Device Data Transfer

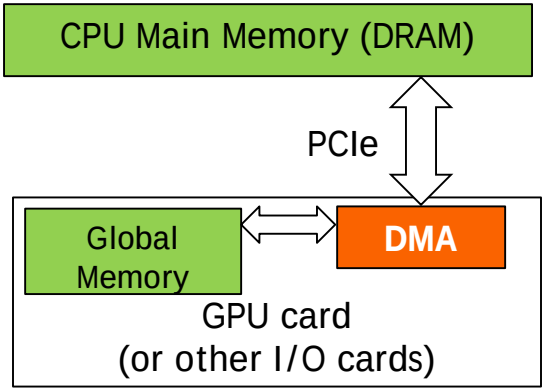
Lecture 14.1 - Pinned Host Memory

Objective

- To learn the important concepts involved in copying (transferring) data between host and device
  - Direct Memory Access
  - Pinned memory

CPU-GPU Data Transfer using DMA

- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()` for better efficiency
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS
  - Between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect, typically PCIe in today's systems



Virtual Memory Management

- Modern computers use virtual memory management
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses (pointer values) are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time

Data Transfer and Virtual Memory

- DMA uses physical addresses
  - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

- The OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

Pinned Memory and DMA Data Transfer

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA data transfer uses pinned memory.

– The DMA used by cudaMemcpy() requires that any source or destination in the host memory is allocated as pinned memory

– If a source or destination of a `cudaMemcpy()` in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead

– `cudaMemcpy()` is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Allocate/Free Pinned Memory

- `cudaHostAlloc()`, three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use `cudaHostAllocDefault` for now
- `cudaFreeHost()`, one parameter
  - Pointer to the memory to be freed

Using Pinned Memory in CUDA

- Use the allocated pinned memory and its pointer the same way as those returned by `malloc();`
- The only difference is that the allocated memory cannot be paged by the OS
- The `cudaMemcpy()` function should be about 2X faster with pinned memory
- Pinned memory is a limited resource
  - over-subscription can have serious consequences

Putting It Together - Vector Addition Host Code Example

```
int main()
{
   float *h_A, *h_B, *h_C;
…
   cudaHostAlloc((void **) &h_A, N* sizeof(float),
      cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_B, N* sizeof(float),
      cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_C, N* sizeof(float),
      cudaHostAllocDefault);
…
   // cudaMemcpy() runs 2X faster
}
```

GPU Teaching Kit

Accelerated Computing

Module 14 – Efficient Host-Device Data Transfer

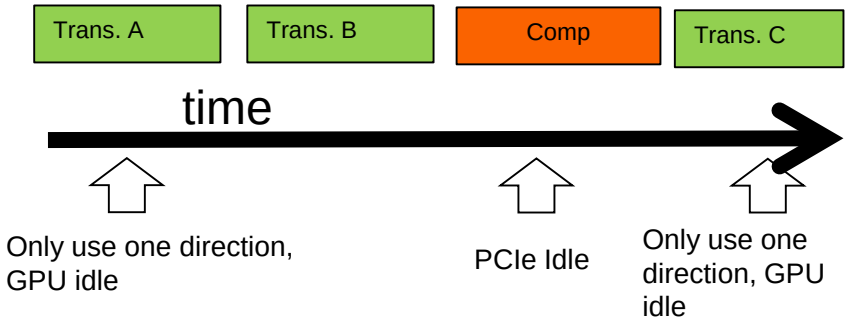
Lecture 14.2 - Task Parallelism in CUDA

Objective

– To learn task parallelism in CUDA
  – CUDA Streams

Serialized Data Transfer and Computation

– So far, the way we use `cudaMemcpy` serializes data transfer and GPU computation for `VecAddKernel()`



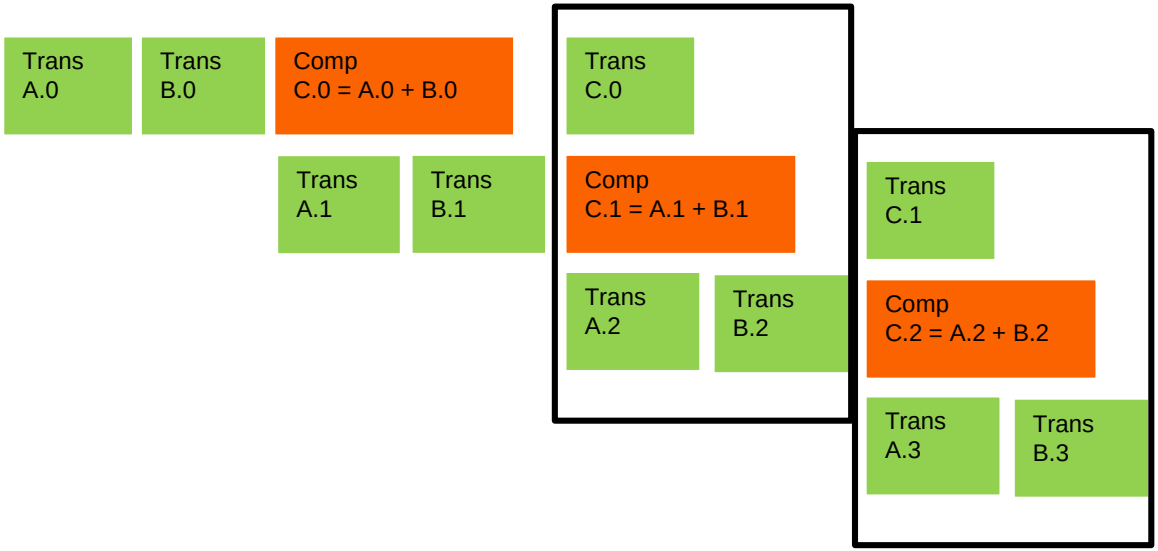
Device Overlap

- Some CUDA devices support device overlap
  - Simultaneously execute a kernel while copying data between device and host memory

```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&prop, i);
  if (prop.deviceOverlap) …
```

Ideal, Pipelined Timing

– Divide large vectors into segments
– Overlap transfer and compute of adjacent segments

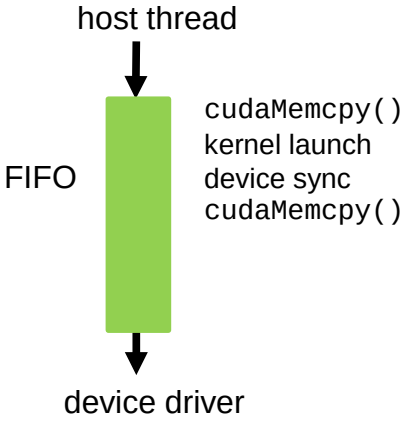


CUDA Streams

- CUDA supports parallel execution of kernels and `cudaMemcpy()` with “Streams”
- Each stream is a queue of operations (kernel launches and `cudaMemcpy()`calls)
- Operations (tasks) in different streams can go in parallel
  - “Task parallelism”

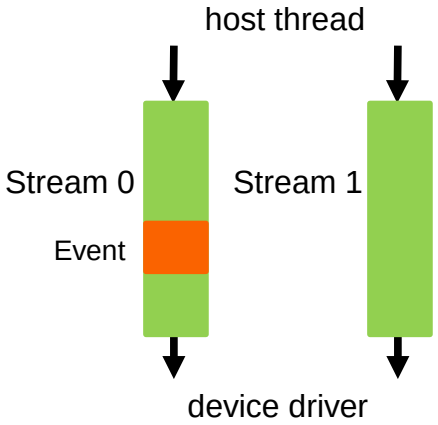
Streams

- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.

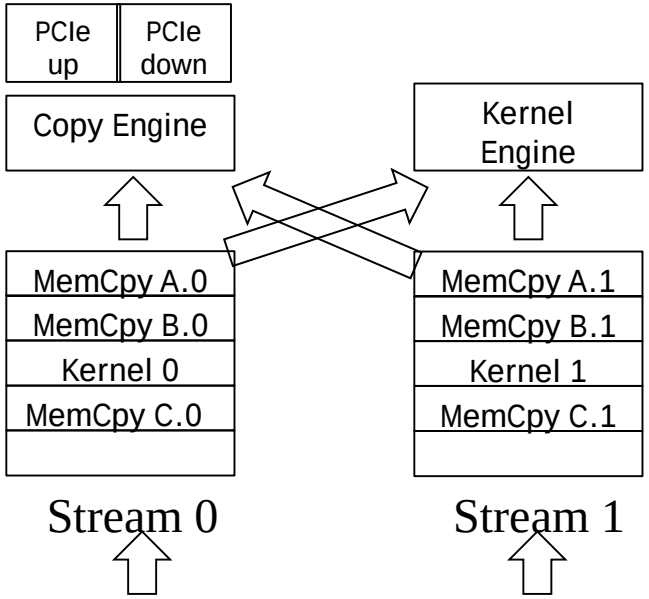


Streams cont.

– To allow concurrent copying and kernel execution, use multiple queues, called “streams”
  – CUDA “events” allow the host thread to query and synchronize with individual queues (i.e. streams).



Conceptual View of Streams



Operations (Kernel launches, `cudaMemcpy()` calls)

GPU Teaching Kit

Accelerated Computing

Module 14 – Efficient Host-Device Data Transfer

Lecture 14.3 - Overlapping Data Transfer with Computation

Objective

- To learn how to overlap data transfer with computation
  - Asynchronous data transfer in CUDA
  - Practical limitations of CUDA streams

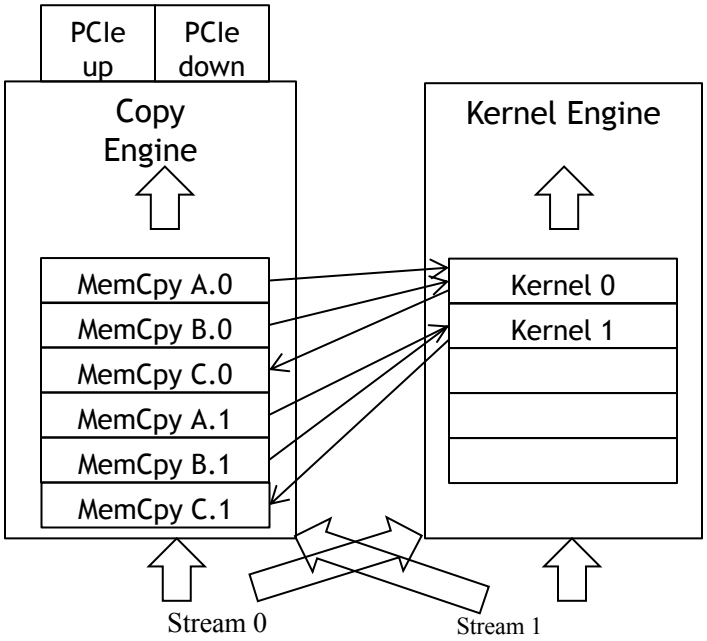
Simple Multi-Stream Host Code

```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go
here
```

Simple Multi-Stream Host Code (Cont.)

```
for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0,…);
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…, stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),…, stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float),…, stream1);
}
```

A View Closer to Reality in Previous GPUs

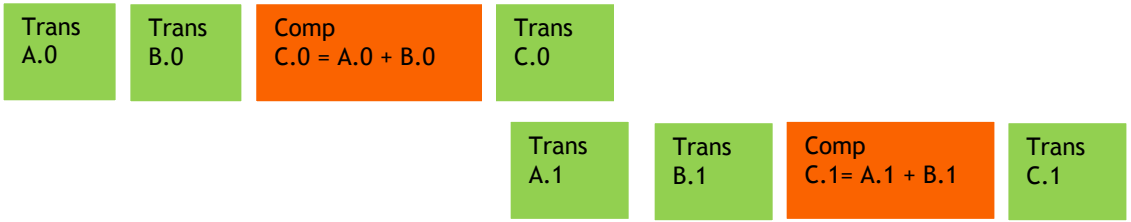


Direction of arrows changed from original slides [SPM]

Operations (Kernel launches, `cudaMemcpy()` calls)

Not quite the overlap we want in some GPUs

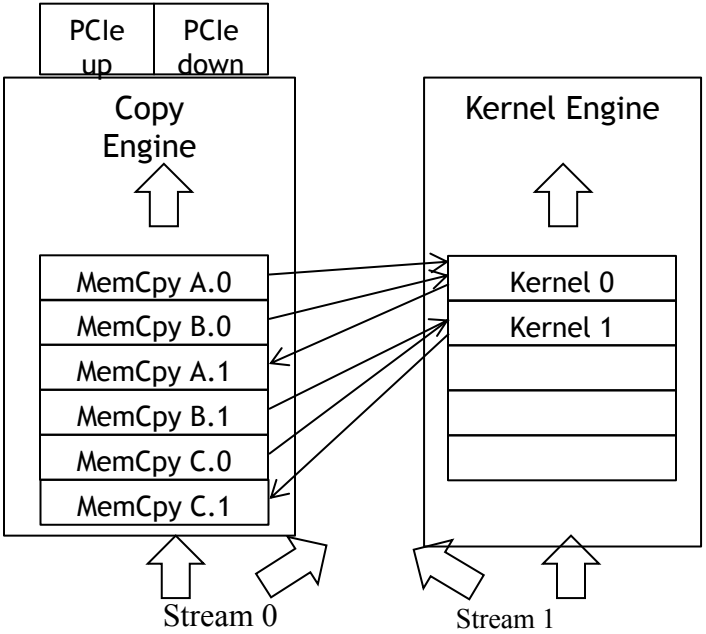
– C.0 blocks A.1 and B.1 in the copy engine queue



Better Multi-Stream Host Code

```
for (int i=0; i<n; i+=SegSize*2) {
  cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…, stream1);
  cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),…, stream1);

  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
  vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);

  cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),…, stream1);
}
```

C.0 no longer blocks A.1 and B.1

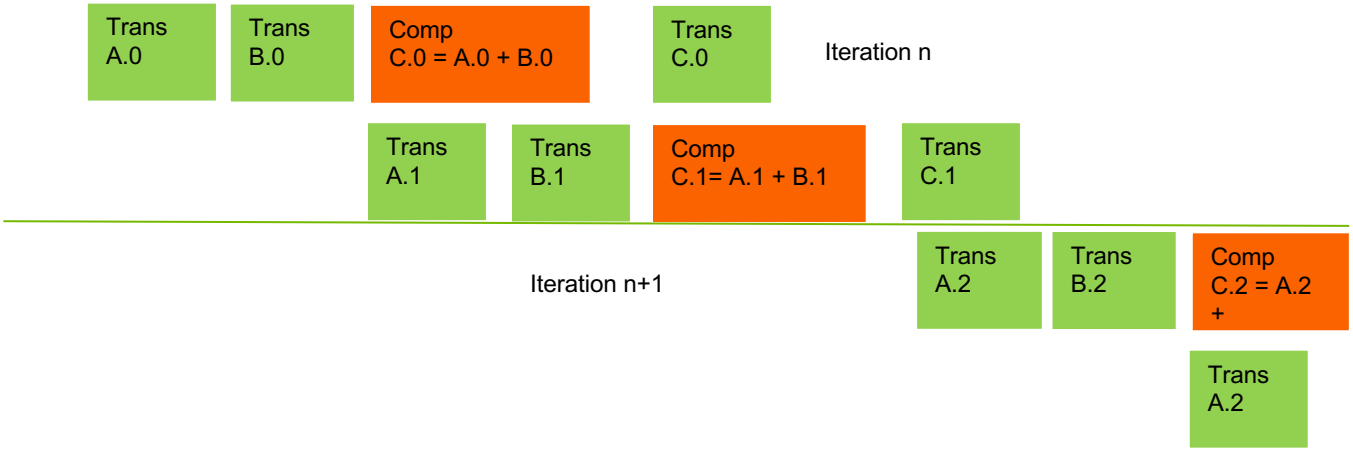


Direction of arrows changed from original slides [SPM]

Operations (Kernel launches, cudaMemcpy() calls)

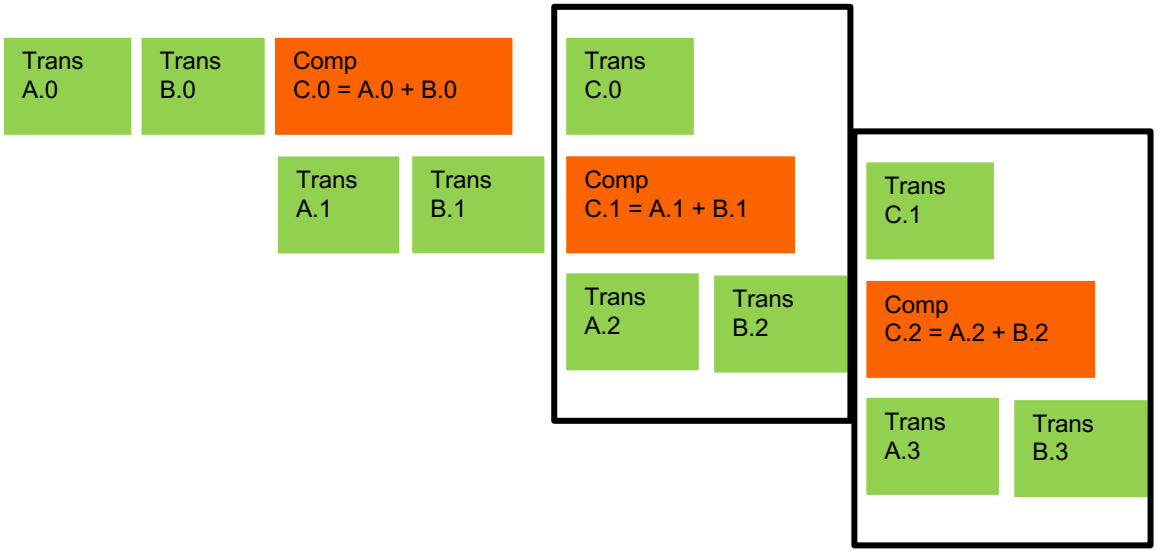
Better, not quite the best overlap

– C.1 blocks next iteration A.2 and B.2 in the copy engine queue



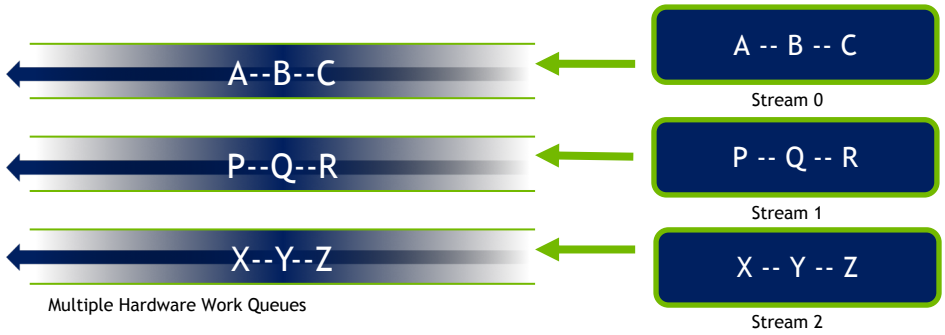
Ideal, Pipelined Timing

– Will need at least three buffers for each original A, B, and C, code is more complicated



Hyper Queues

– Provide multiple queues for each engine
– Allow more concurrency by allowing some streams to make progress for an engine while others are blocked



Wait until all tasks have completed

- `cudaStreamSynchronize(stream_id)`
  - Used in host code
  - Takes one parameter – stream identifier
  - Wait until all tasks in a stream have completed
  - `E.g., cudaStreamSynchronize(stream0)` in host code ensures that all tasks in the queues of stream0 have completed

- This is different from `cudaDeviceSynchronize()`
  - Also used in host code
  - No parameter
  - `cudaDeviceSynchronize()` waits until all tasks in all streams have completed for the current device

GPU Teaching Kit

Accelerated Computing