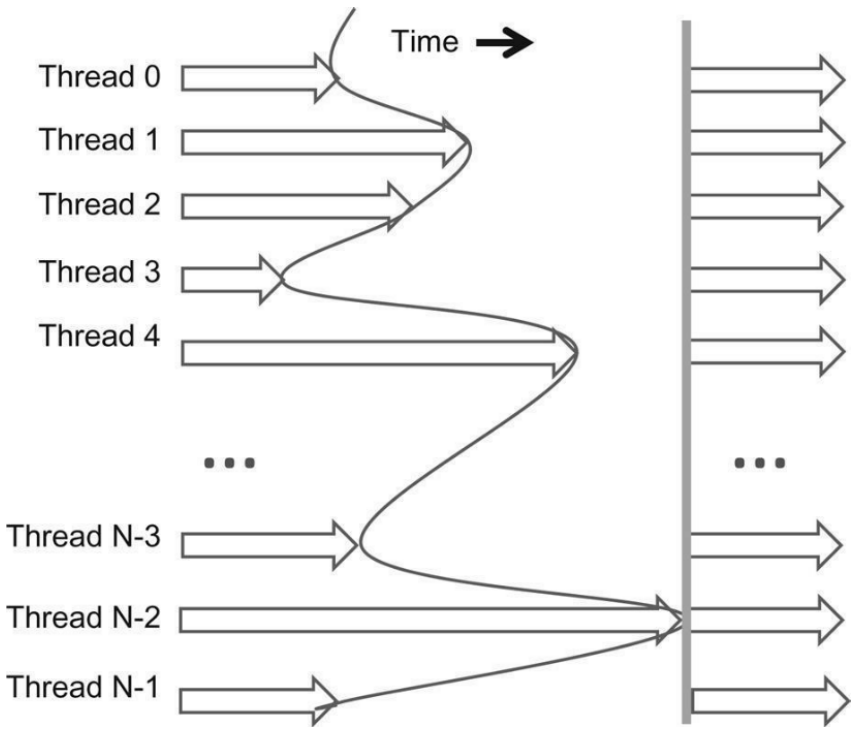
CSc 8530
Parallel Algorithms

Spring 2019

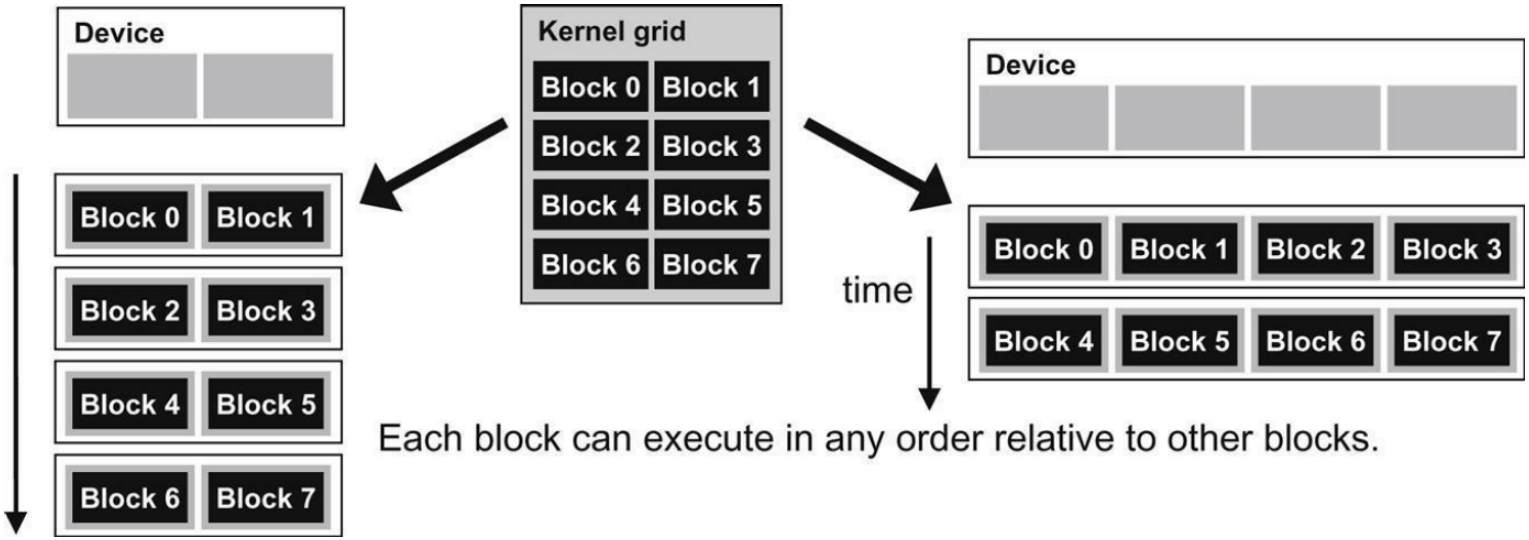
March 14th, 2019

Barrier synchronization



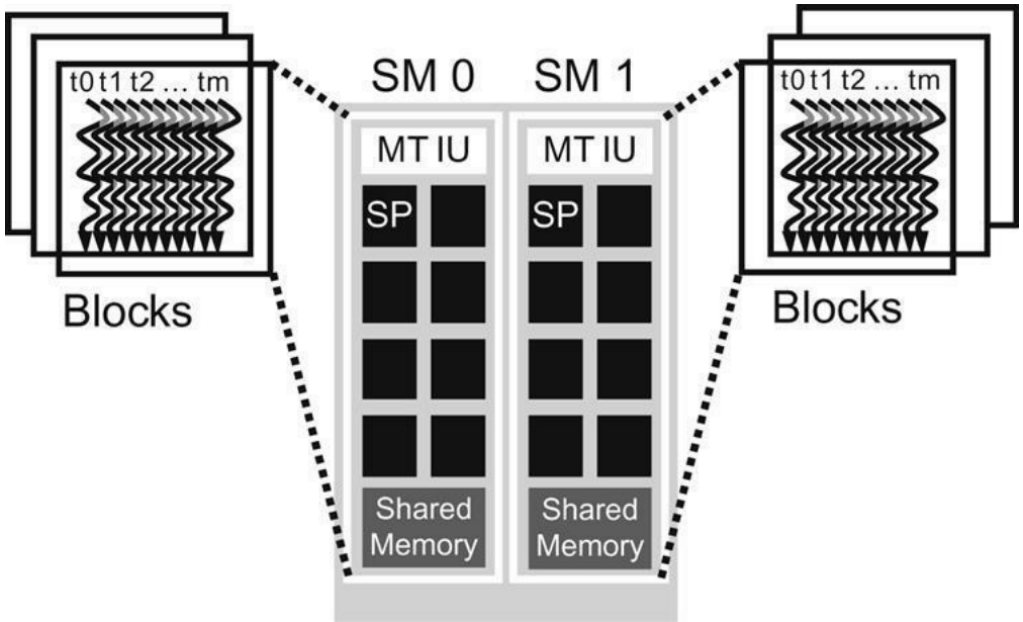
- All synchronized threads must wait until every one of them has reached the barrier location
- Faster threads will have to wait idle while the slower ones catch up

Transparent scalability



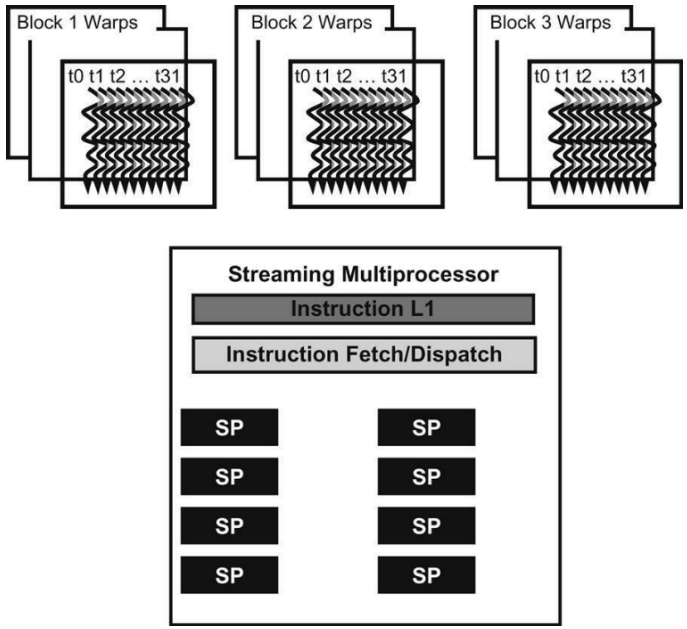
- The effective level of parallelization can be scaled to the available resources
- Above, a GPU with twice as many processors can execute the code twice as fast

Streaming Multiprocessors (SM)



- Above, each SM is assigned 3 blocks
- SMs use built-in registers to track all their assigned threads
  - The number of registers sets a limit on the number of simultaneous threads

Warp scheduling



- Each SM has a number of **Streaming Processors (SP)**
  - These actually execute the threads
- In early GPUs, an SM could only execute one instruction for one warp at a time
- Current GPUs, each SM can handle a few warps at a time

Memory access efficiency

- The CUDA kernels we have looked at so far will only achieve a small, real-world speed-up
  - Long access latencies (hundreds of clock cycles)
  - Access to global memory is a bottleneck
- Traffic congestion in accessing global memory can reduce the number of threads that can execute in parallel
  - Leaving many SMs idle
- CUDA provides additional methods for reducing accesses to global memory

Memory efficiency: example

```
      for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

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

- Above, we have the most executed portion of the image blurring kernel
  - Calculating the mean at each pixel
- We make one global memory access for every floating-point operation
  - The *compute-to-global-memory-access ratio* $= 1.0$

Compute-to-global-memory-access ratio

- This ratio has major implications for the performance of a CUDA program
- Current high-end devices have memory bandwidth of $\approx$ 1000GB/s (1TB/s)
- A single floating-point number has 4 bytes
- So, total number of loadable values is 1000GB/4 = 250GB
- With a ratio of 1.0, this yields 250 GFLOPS (giga floating-point operations per second)
- Only $\approx$ 2% of the 12 TFLOPS or higher that modern GPUs can handle
  - To achieve 12 TFLOPS, we need a compute ratio of 48 or better
- Such a program is **memory bound**

Matrix multiplication

- We will now explore how to reduce memory accesses
  - Using matrix multiplication as an example
- The execution speed of matrix multiplication can vary by orders of magnitude depending on memory accesses
- Let M and N be $i \times j$ and $j \times k$ matrices, resp.
- For $P = MN$ (how big is P?), $P_{Row,Col}$ is the inner product of a row of M with a column of N:

$$P_{Row,Col} = \sum_{q=0}^{k-1} M_{Row,q} \, N_{q,Col}$$

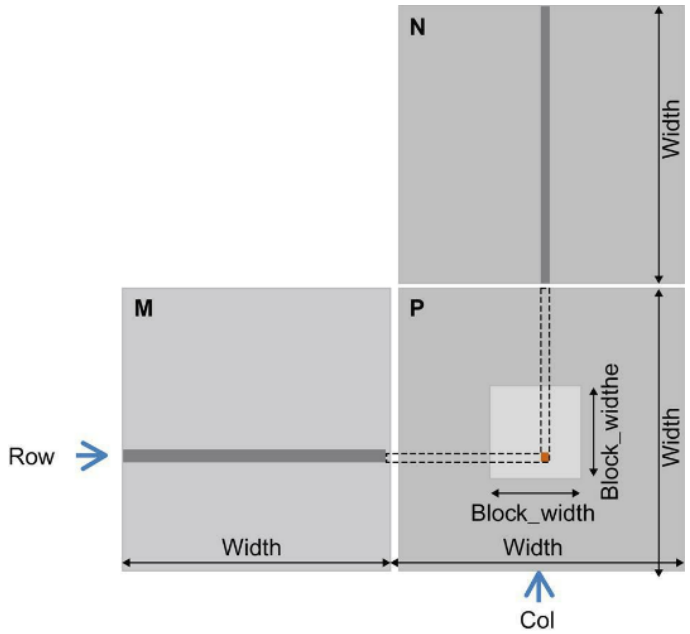
Matrix multiplication

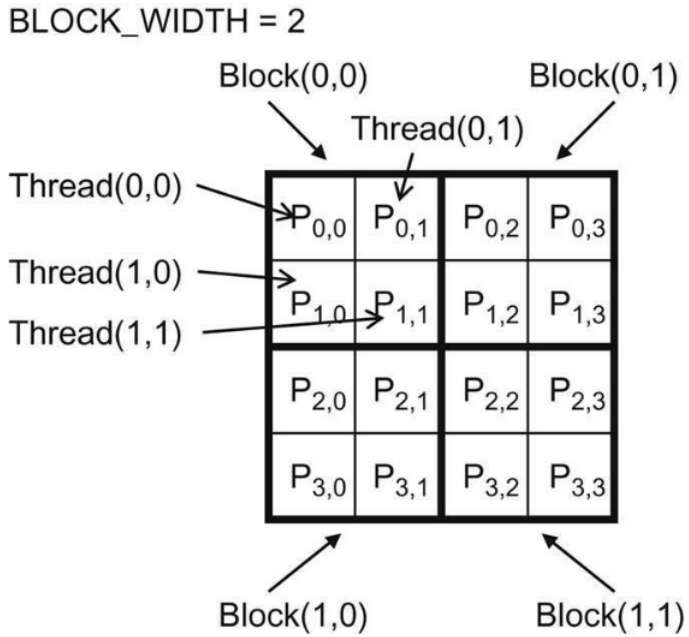
```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    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;
  }

}
```

- Simple matrix multiplication code
- Each threads is responsible for calculating one element of P
  - Code is similar to the color-to-grayscale code

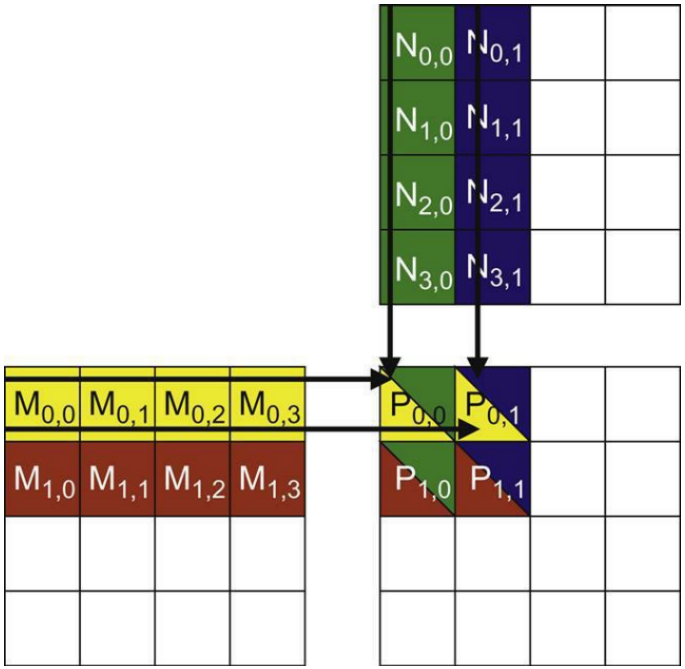
Matrix multiplication: example





- We divide the matrix into blocks of size BLOCK_WIDTH
- Each thread is responsible for one element of P

Matrix multiplication example



- Each processor needs access to a specific row and column (color-coded above)

Matrix multiplication efficiency

```
__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;
  }

}
```

- The memory accesses of this program are dominated by the inner for-loop

Matrix multiplication efficiency

```
__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;
  }

}
```

- The memory accesses of this program are dominated by the inner for-loop
  - Two global memory accesses

Matrix multiplication efficiency

```
__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;
  }

}
```

- The memory accesses of this program are dominated by the inner for-loop
  - Two global memory accesses
  - One addition and one multiplication

Matrix multiplication efficiency

```
__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;
  }

}
```

- The memory accesses of this program are dominated by the inner for-loop
  - Two global memory accesses
  - One addition and one multiplication
- Compute ratio of 1.0

CUDA memory types

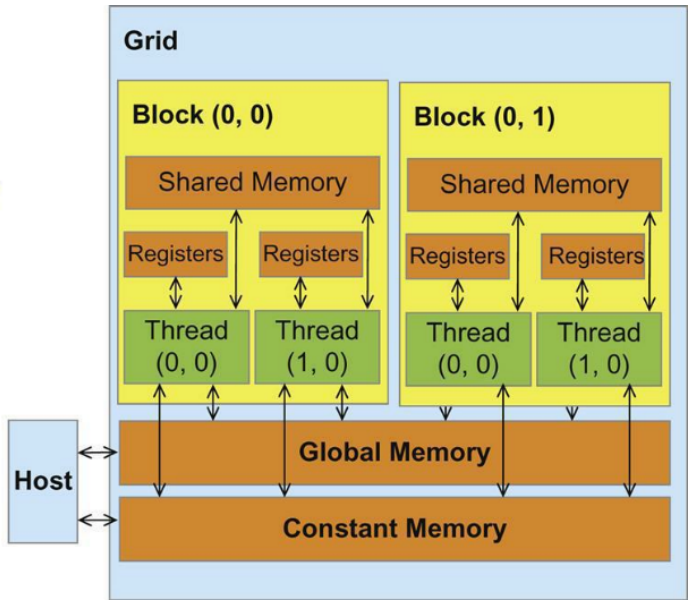
- A GPU contains several types of memory
- Using them effectively is crucial for improving the compute ratio

Device code can:

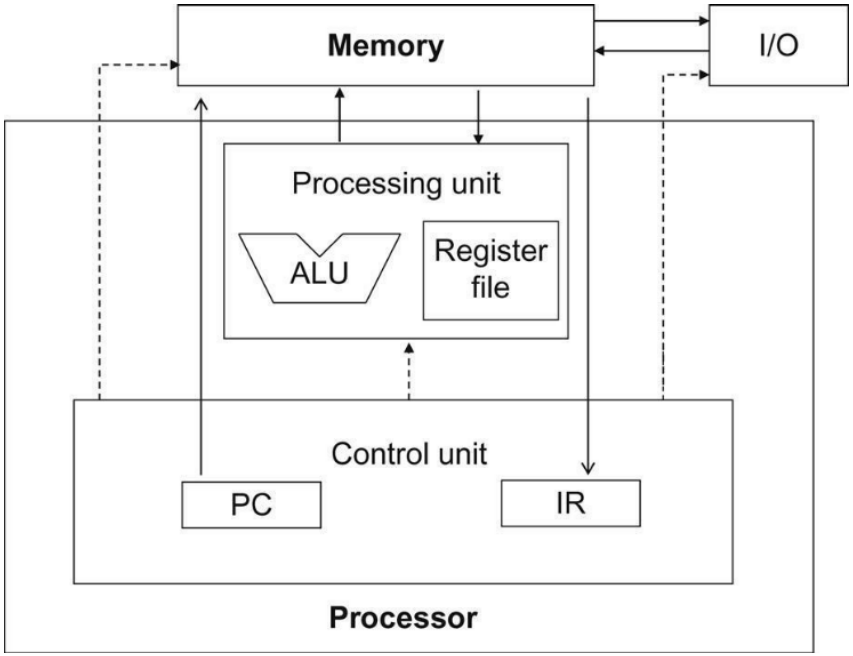
- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories

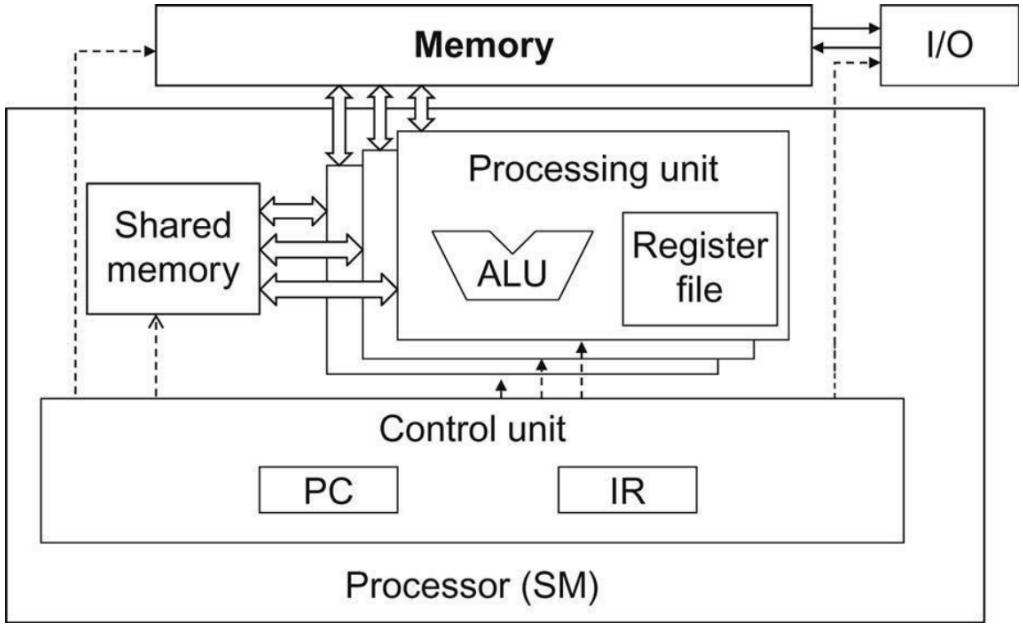


CUDA memory types



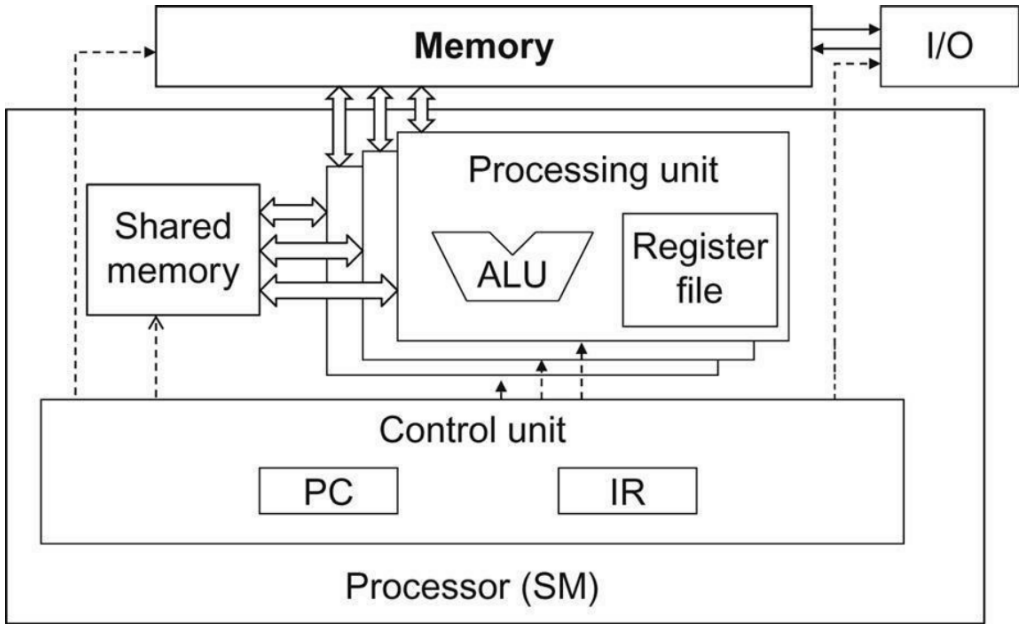
- CUDA devices (i.e., GPUs) still adhere to the Von-Neumann architecture (above)
- Registers are typically two orders of magnitude faster than global memory
  - ALU units can also manipulate register variables directly

CUDA memory types



- A GPU architecture adds shared memory and multiple processors to the Von-Neumann model
- Shared memory has lower latency that global memory
  - But ALUs cannot access it directly
  - Requires one load instruction (same as global memory)

CUDA memory types



- Registers are unique to an individual thread
- Shared memory can be accessed by all the threads in a block
  - Hence, a CREW paradigm

CUDA memory types

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

- In CUDA, we can declare variables so that reside in a specific type of memory
- **Scope** is the set of threads that have access to it
- **Lifetime** is how long the variable is maintained
  - e.g., the top three are initialized every time we call a kernel function

CUDA memory types

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

- Variables other than arrays (or matrices) are *scalar*
- Array variables are stored in *global memory*
  - The name *Local* just means it is visible to a single thread
  - There is no special hardware for "local" memory
  - Thus, you should minimize variables of this type

CUDA memory types

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

- Shared variables are a more efficient alternative for arrays
  - Note that the `__shared__` keyword has two underscores
- Accessing shared variables is extremely fast
  - But they are only visible to a *single block*
  - You might need to redesign your code to minimize accessing data across blocks

CUDA memory types

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

- Constant variables are accessible to *all threads* in *all grids*
  - But they are read only
- They must be declared outside of any function
- They are stored in global memory
  - But cached for efficiency
  - Accessing is typically very fast

CUDA memory types

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

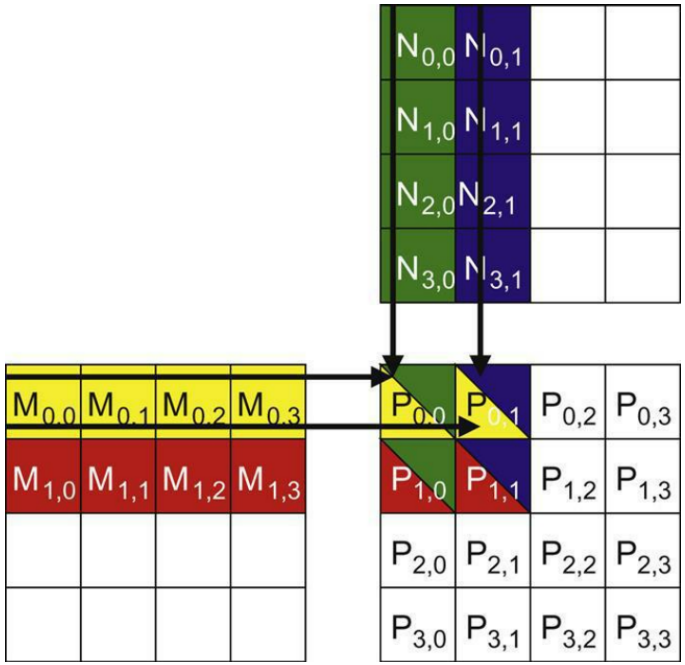
- Global variables are also accessible to all threads in all grids
- But, they can be read and written to
  - Hence, accessing is really slow
- Also, it is impossible to synchronize across different blocks in the same kernel function call
- Used to pass information between different invocations of the same function

Tiling to reduce memory access

- GPU programming has an inherent tradeoff:
  - Global memory is large and slow
  - Shared memory is small and fast
- A common strategy is to partition the data into **tiles**
  - Each tile fits into an SM's shared memory
  - The term comes from wall tiles (like the ones in a bathroom)
- Kernel computations must be *independent* for each tile
  - Not all data or kernel functions can be partitioned in this way

Matrix multiplication revisited

- The colors indicate which threads need which parts of the input matrices
- The black arrows show the memory accesses of the first two threads, $P_{0,0}$ and $P_{0,1}$, from the first block
- They both need the same row, but different columns
- They access each needed element sequentially



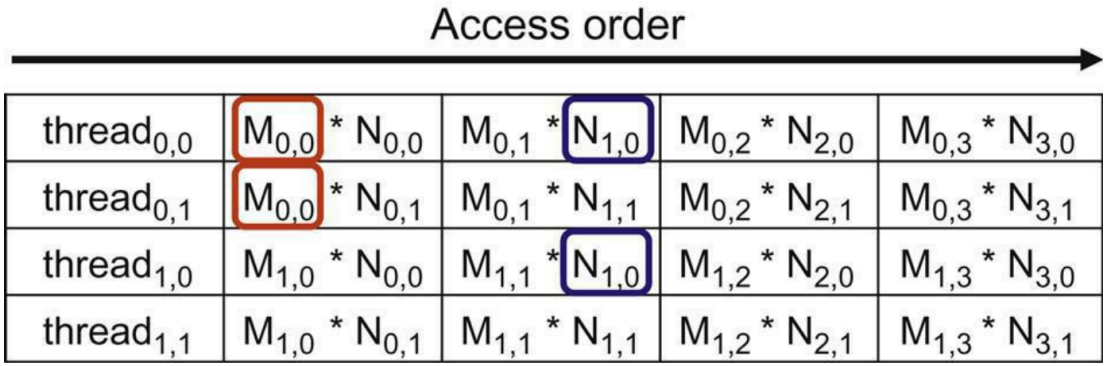
Matrix multiplication revisited

Access order

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

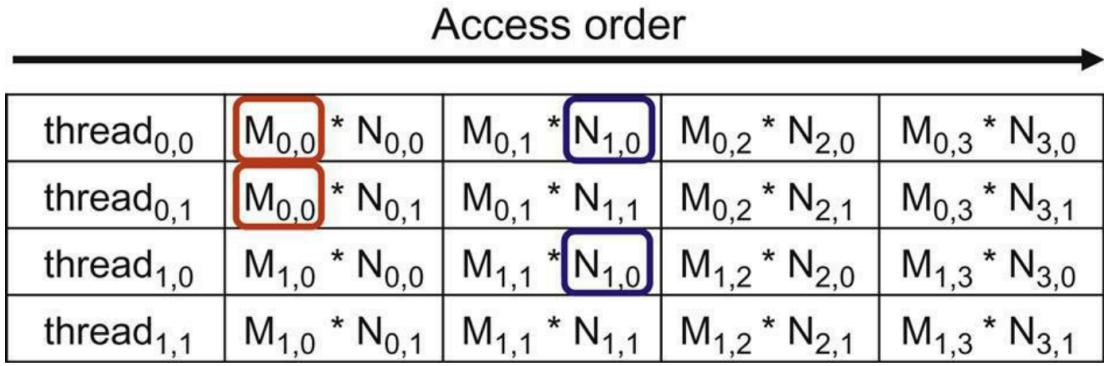
- Above, we have the sequence of memory accesses for all threads, T, in block(0,0)
- e.g. both $T_{0,0}$ and $T_{0,1}$ access $M_{0,0}$
- Note how, in this example each element of M and N is accessed twice
  - If we only retrieve each element once, we reduce global memory accesses by half

Matrix multiplication revisited



- The 1/2 reduction is due to using 2×2 blocks
- **Quick exercise:** what would the factor be for 16×16 blocks?

Matrix multiplication revisited



- The 1/2 reduction is due to using 2×2 blocks
- **Quick exercise:** what would the factor be for 16×16 blocks?
  - 1/16 (see board for details)

Matrix multiplication revisited

Access order →

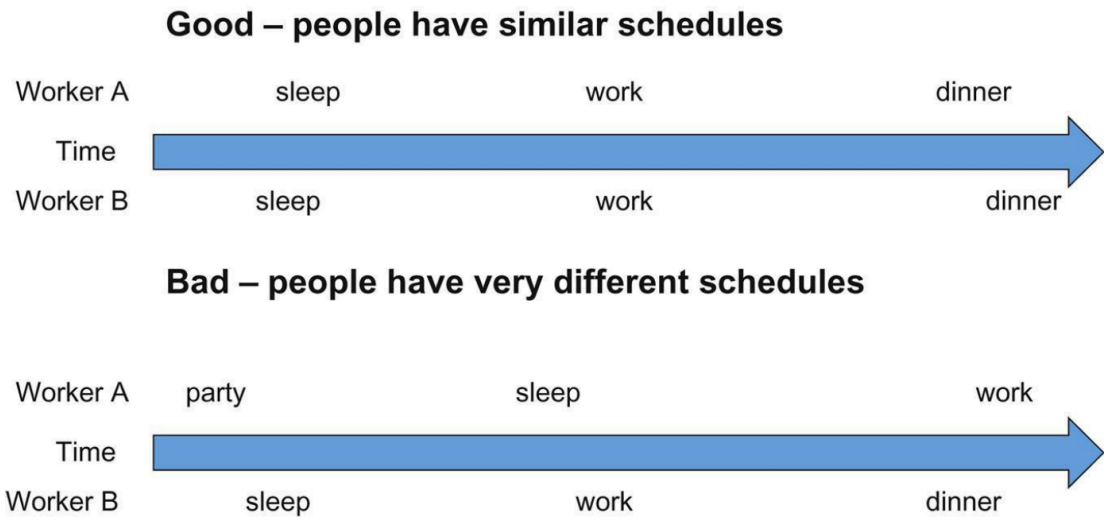
$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0}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

- The 1/2 reduction is due to using 2×2 blocks
- **Quick exercise:** what would the factor be for 16×16 blocks?
  - 1/16 (see board for details)
- Thus, larger blocks are more memory efficient
  - But remember that we have block and shared memory size limitations

Matrix multiplication revisited

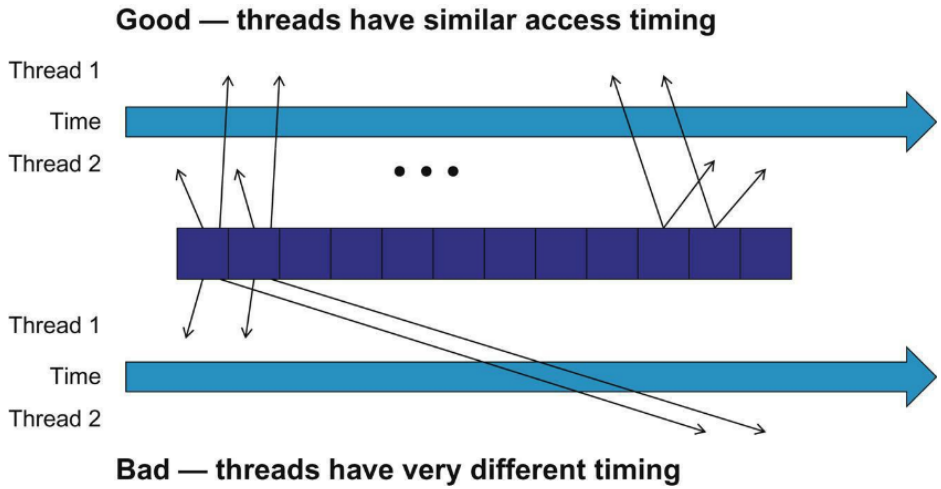
- Memory accesses are similar to highway traffic
- The use of *carpooling* (i.e., sharing resources) reduces overall congestion

Matrix multiplication revisited



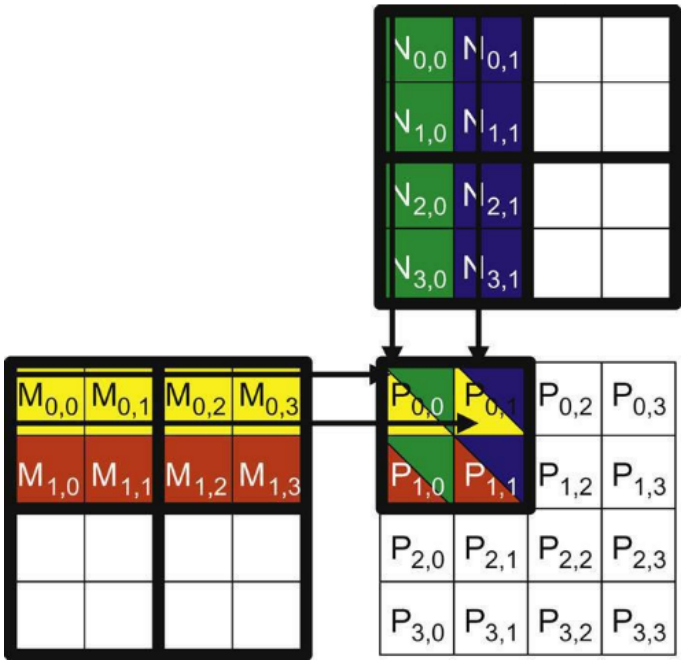
- However, carpooling requires extra effort
- In particular, carpoolers need to synchronize their schedules
- People with very different schedules will be worse off with carpooling than without it

Matrix multiplication revisited



- Tiled algorithms are very similar to carpooling arrangements
  - Threads are commuters
  - Requests to DRAM are vehicles
- If multiple threads need to access the same memory location, they can "carpool" (i.e., make a single DRAM request)
  - Threads must have similar schedules for this scheme to work

Matrix multiplication revisited



- Above, we divide M and N into 2×2 tiles
- We load one tile of M and one of N into the shared memory at a time
- In the simplest case, tile size equals block size
  - But it doesn't have to

Matrix multiplication revisited

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

time →

- All the threads collaborate to load individual elements from the current tile
- Once elements are loaded onto shared memory, they can be accessed by multiple threads

Matrix multiplication revisited

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

time →

- We now have to compute the dot product in **phases**
  - We accumulate the partial dot product after every phase
- We progressively load parts of the rows and columns into smaller arrays Mds and Nds
  - These array live in shared memory
  - We can reuse them every time a new tile is loaded
    - Memory accesses exhibit **locality**