CS 6023 - GPU Programming

Memory Tiling in CUDA Programs

15/02/2019

Agenda

- CUDA memories
- Optimization on memories

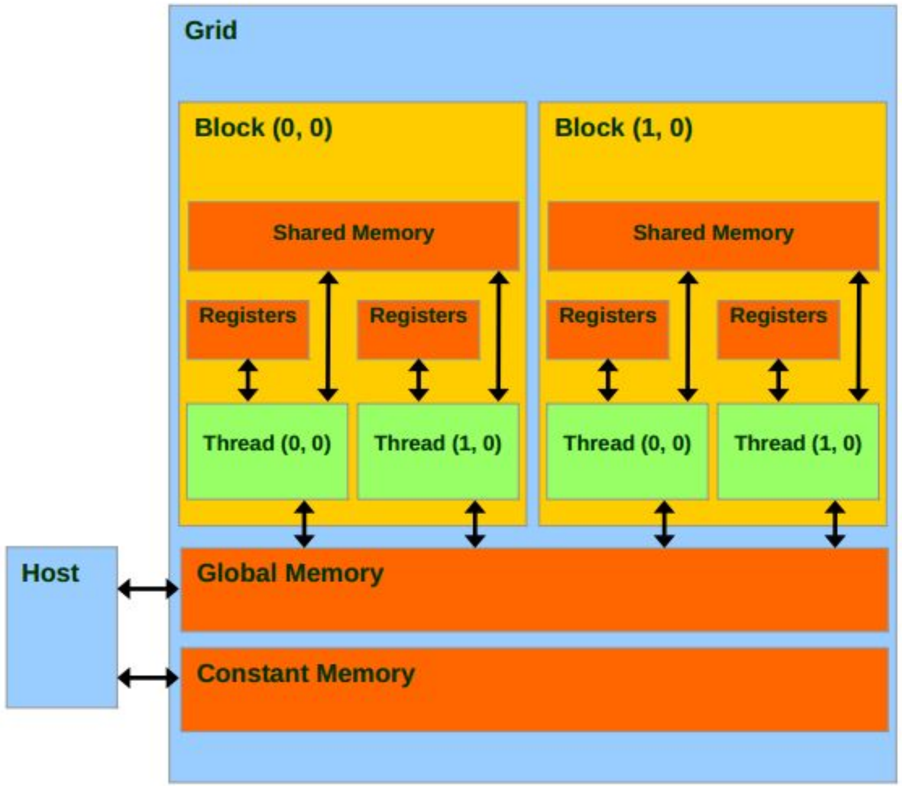
Acknowledgement: Nvidia teaching kit

So far

- We understand CPU and GPU architectures
- We know the different framework choices for parallel programs
- Know how to write a highly parallel vector addition for GPUs in CUDA C
- Know how to trade-off parallelism across blocks and threads

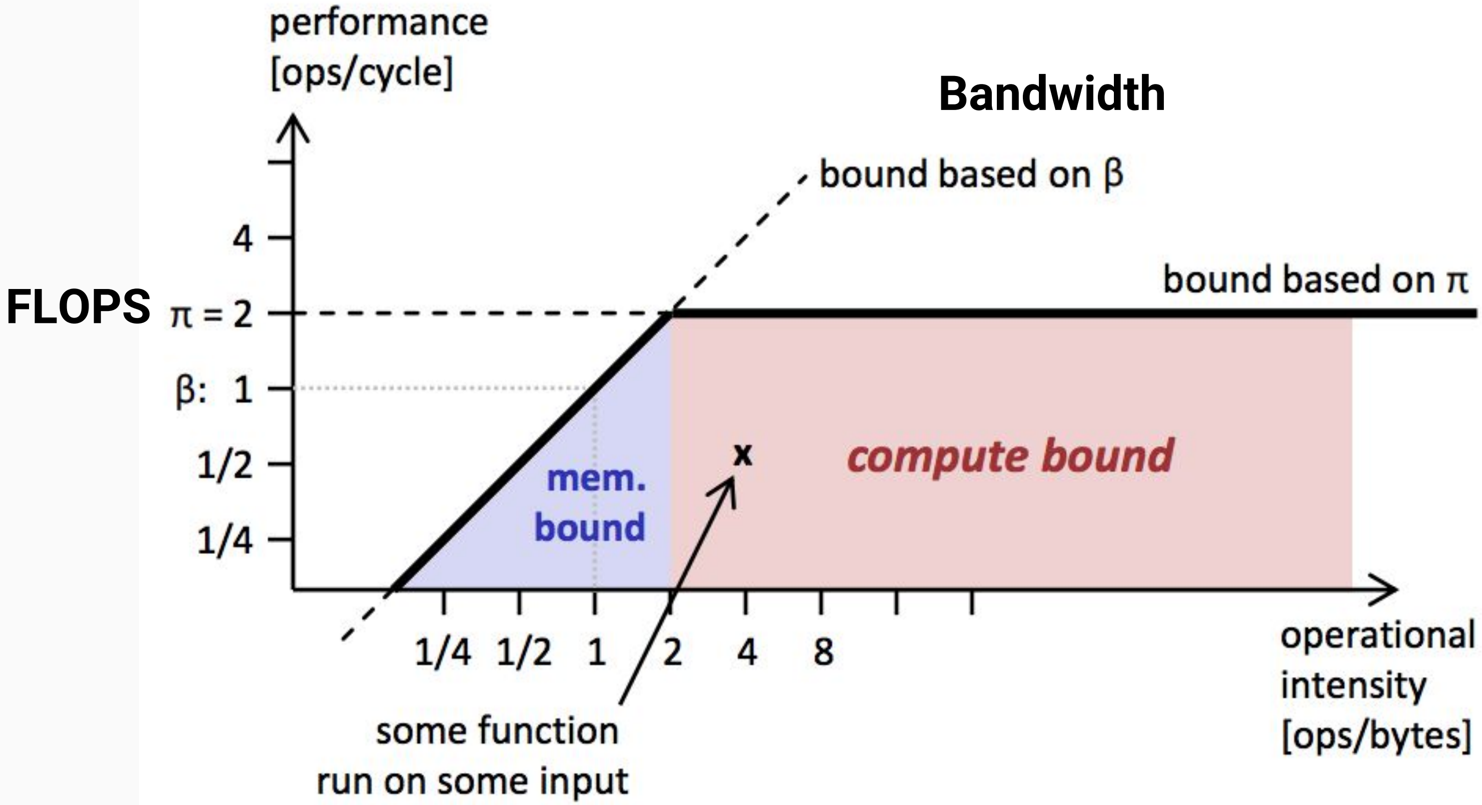
Recap: CUDA memory operations

Device	Thread	R/W registers
		R/W local memory
	Block	R/W shared memory
	Grid	R/W global memory
		Read only constant memory
Host	Grid	R/W global memory
		R/W constant memory



http://courses.engr.illinois.edu/ece498

Roofline model



https://spcl.inf.ethz.ch/Teaching/2017-dphpc/lecture/lecture6-roofline.pdf

Hitting the roof on K40

- Back-of-the-envelope calculation with K40
- Memory bandwidth: 288 GB/sec. Double precision: 1.43TFLOPs
- What is the utilization for vector addition?

Hitting the roof on K40

- Back-of-the-envelope calculation with K40
- Memory bandwidth: 288 GB/sec. Double precision: 1.43TFLOPs
- What is the utilization for vector addition?
    - Every addition (FLOP) requires 24 bytes of memory
    - With 288GB/sec, we can perform 288/24 = 12 GFLOPs
    - Utilization = 12 G / 1.43 T = 0.008 = 0.8%

Memory a huge bottleneck

Efficient memory re-use

- In vector addition, we use each read byte exactly once
    - No scope to increase computational efficiency
- What are some applications with data reuse?

Efficient memory re-use

- In vector addition, we use each read byte exactly once
  - No scope to increase computational efficiency
- However, in other applications, data is used several times
  - Moving average
  - Matrix multiplication
  - Deep Learning - convolution
- Need to ensure effective re-use of data on GPU (SMs to global memory)

Square Matrix Multiplication

Class exercise:

How would you design a kernel for Matrix Multiplication?

Use both block and thread dimensions as 2d

Square 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;
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
}
```

Square 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;
    }
}
```

Output stationary

What is the work done by each block?

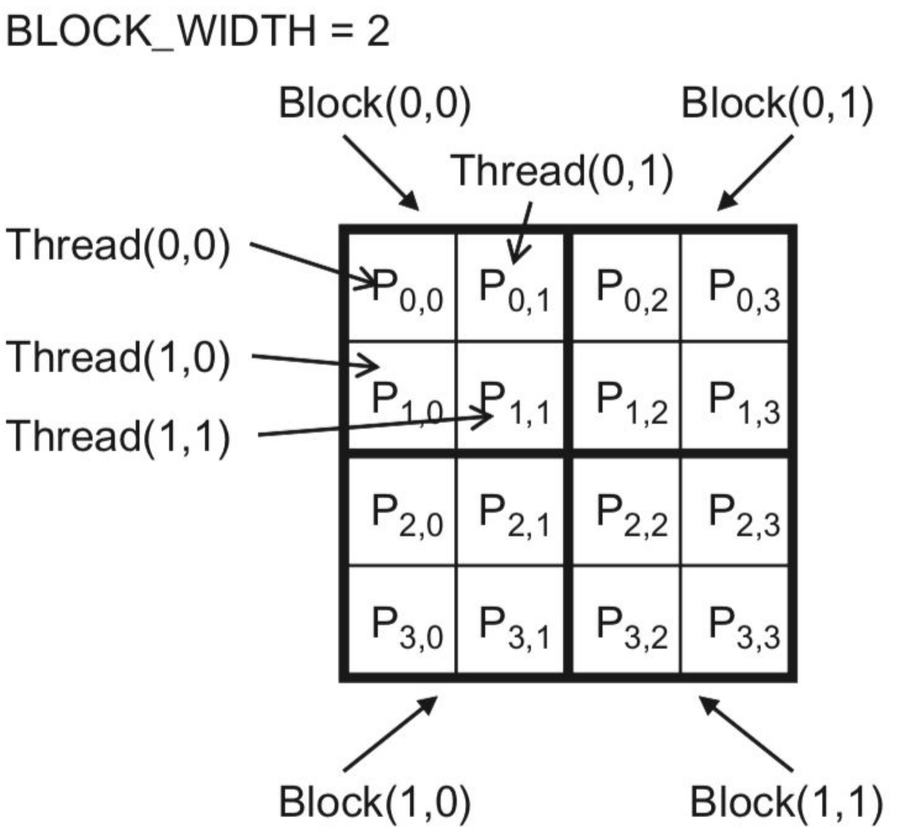
Example

4x4 matrix

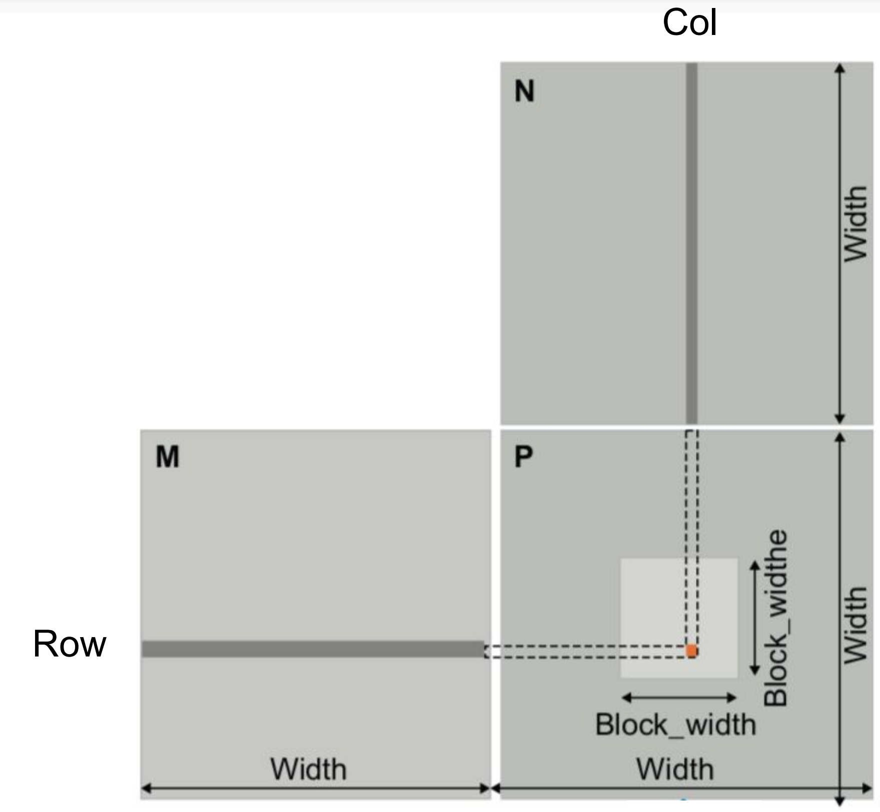
Block_dim (2, 2)

Thread_dim (2,2)

Each block produces the output of a tile of size 2x2



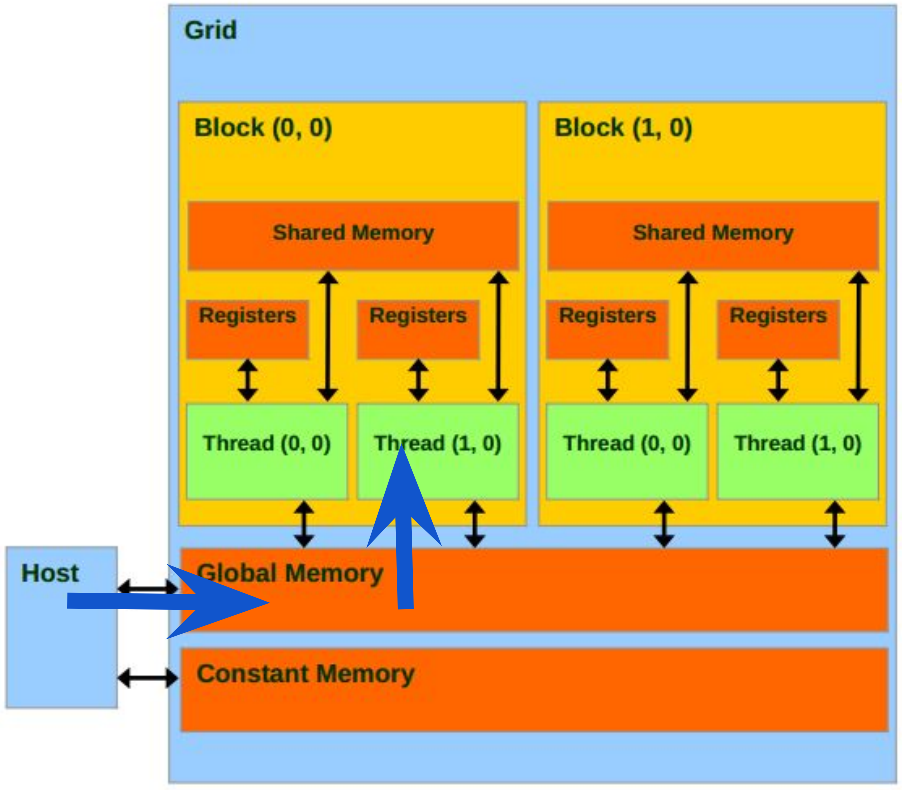
What is the work done by each block?



How are the memory accesses happening?

First the host copies the matrices to global memory

Then each thread reads the matrix entries from the global memory



http://courses.engr.illinois.edu/ece498

What is the sequence of accesses?

M00	M01	M02	M03
M10	M11	M12	M13
M20	M21	M22	M23
M30	M31	M32	M33

Stored in memory in row major order

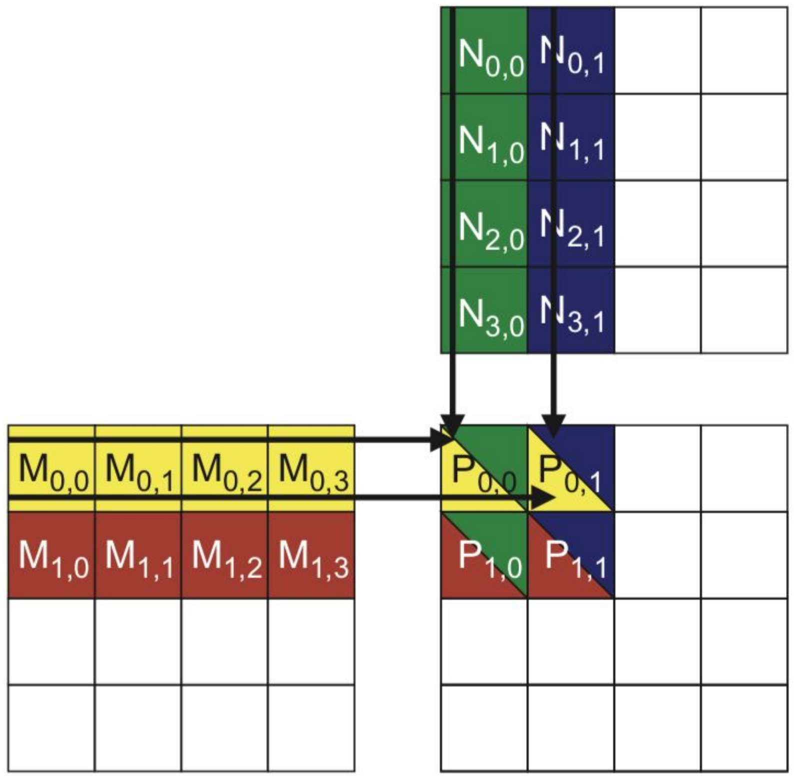
M00	M01	M02	M03	M10	M11	M12	M13	M20	M21	M22	M23	M30	M31	M32	M33

What is the sequence of accesses?

Visualize the memory access patterns of all threads

Are they contiguous accesses in memory?

What about threads of a block?



Optimizing the memory access patterns

We should be able to

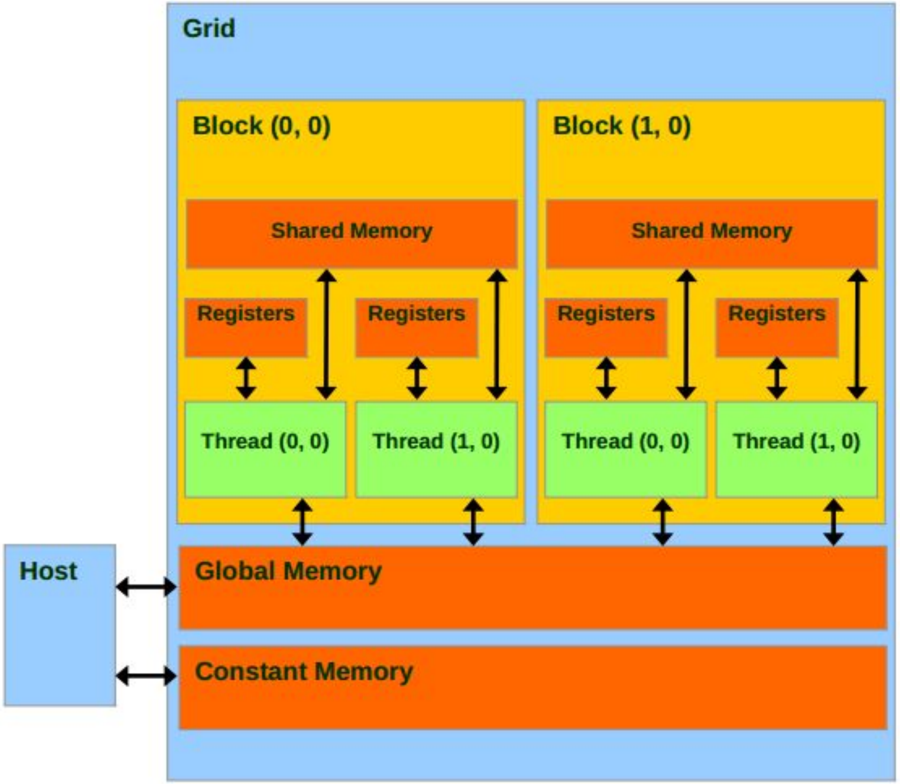
- Reduce the number of global memory accesses
- Order the memory accesses to be contiguous

Need to understand

- Other memory types available in CUDA C
- Sharing memory across threads
- Parallel programming technique of tiling

Other memory types

Device	Thread	R/W registers
	Thread	R/W local memory
	Block	R/W shared memory
	Grid	R/W global memory
	Grid	Read only constant memory
Host	Grid	R/W global memory
	Grid	R/W constant memory



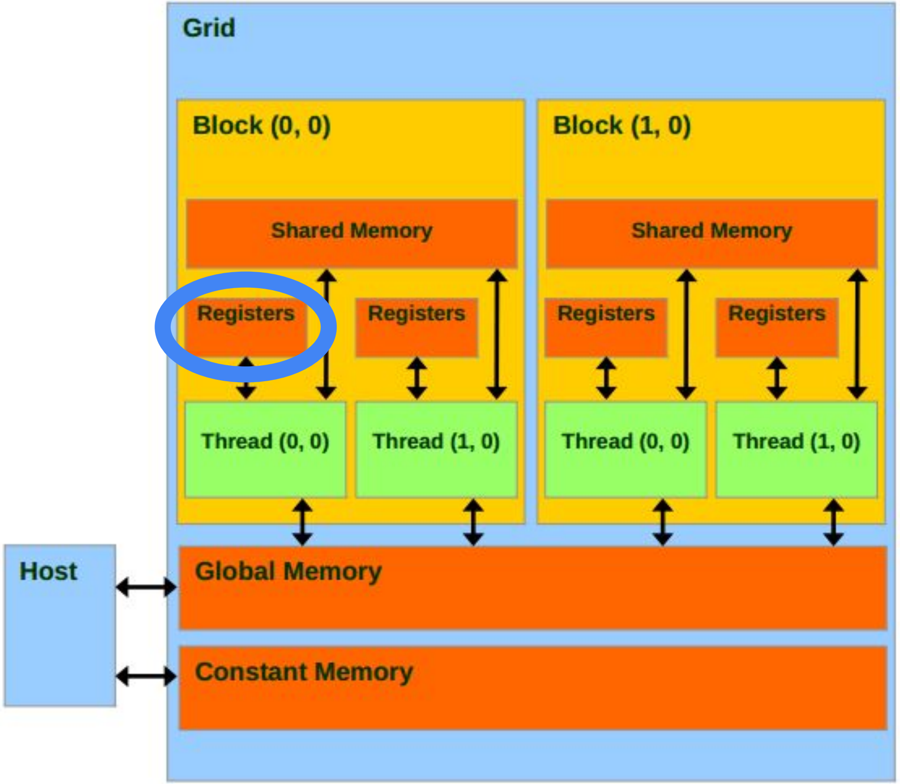
Other memory types

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	register	thread	kernel
Automatic array variables	local	thread	kernel
`__shared__ int sharedVar;`	shared	block	kernel
`__device__ int globalVar;`	global	grid	application
`__constant__ int constantVar;`	constant	grid	application

Registers

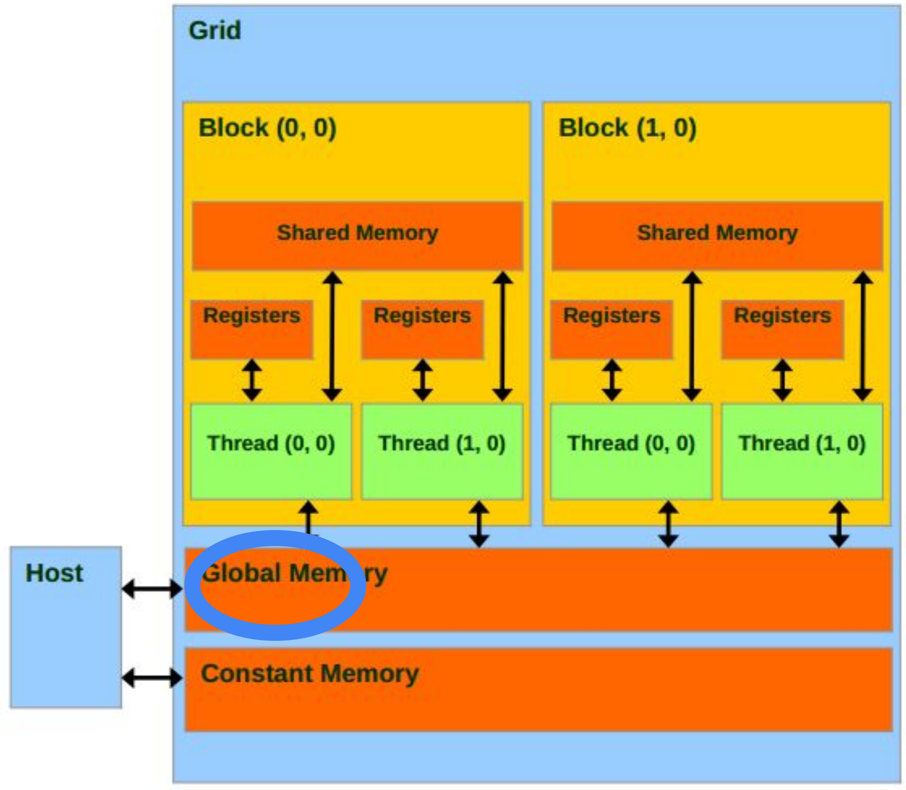
- Fastest read-write speeds
- Local to thread
- Non-array variables declared within kernel go here

- Limits # of threads per SM
- K40: 65,536 registers per block
- At 1,024 max threads per block, we have 64 regs/thread



Local memory

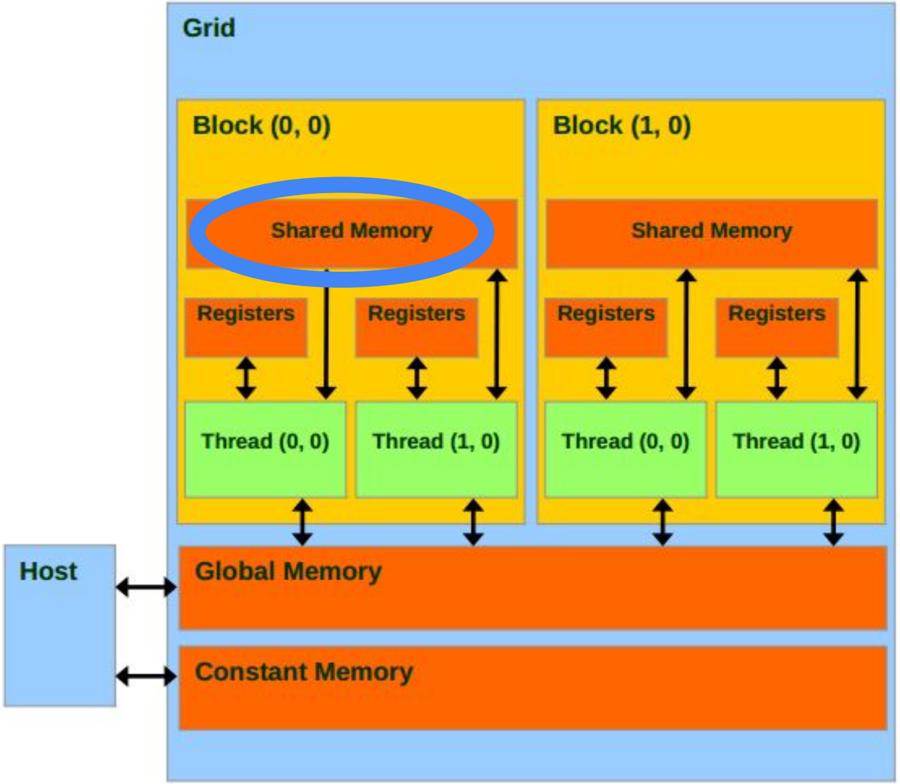
- Physically stored in global memory
- Separated by thread
- Automatic arrays go here



Shared memory

- Fast read-write memory, local to a block
- Random access
- Lifetime - thread block
- A form of scratchpad memory

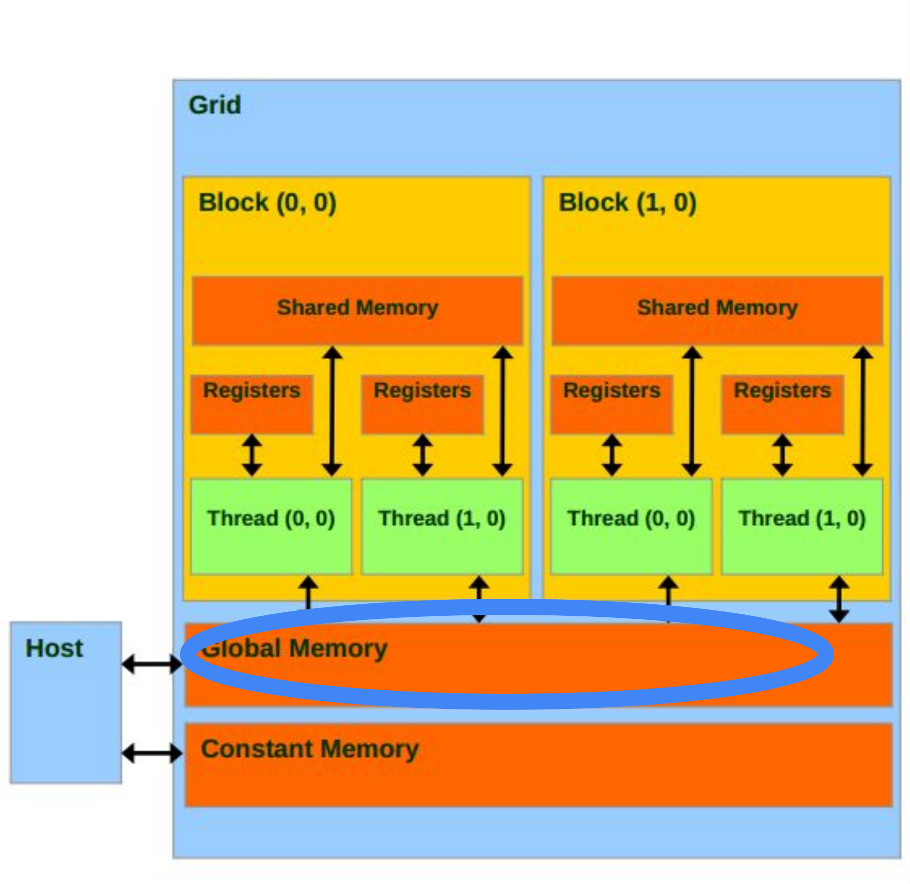
- K40: 48 KB per block
- Max. 8 blocks per SM



Global memory

- Slow (100s of cycles), large memory, shared for all SMs
- Has off-chip access, to/from host
- Performance loss with random access

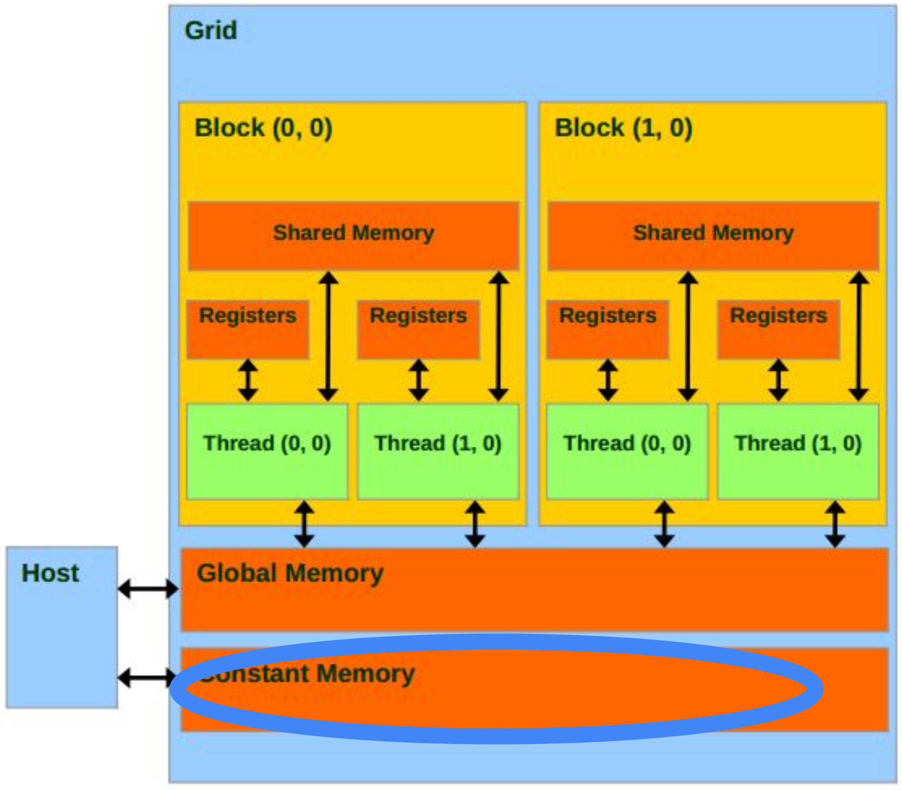
- K40: 12 GB
- 288 GB/s off-chip bandwidth



Constant memory

- Fast, small memory available to all SMs for read-only, host can read/write
- Usually cached

- K40: 64KB



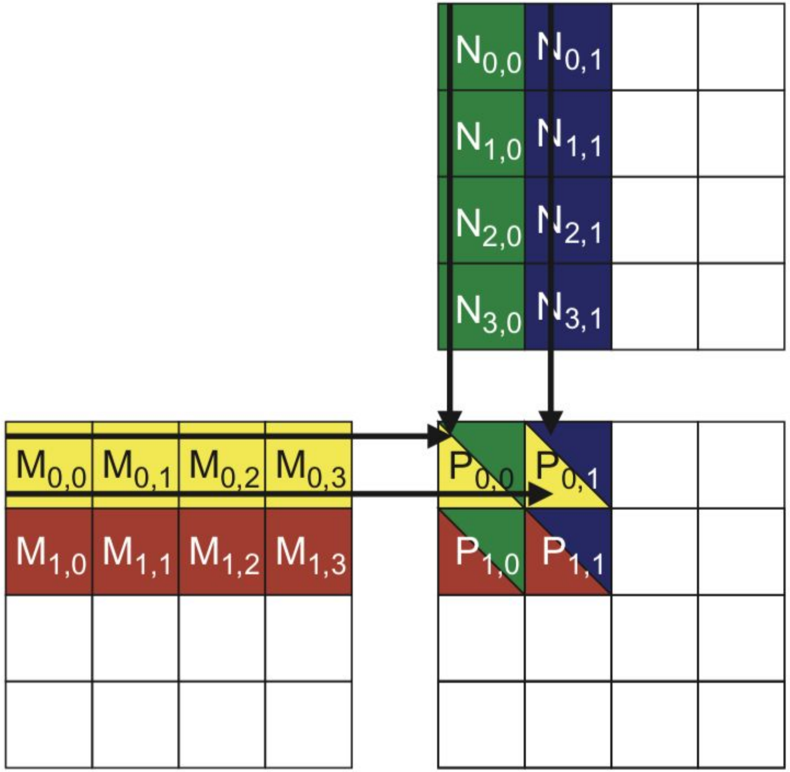
Coming back to the matrix multiplication example

How do we minimize the dependence on the slow global memory?

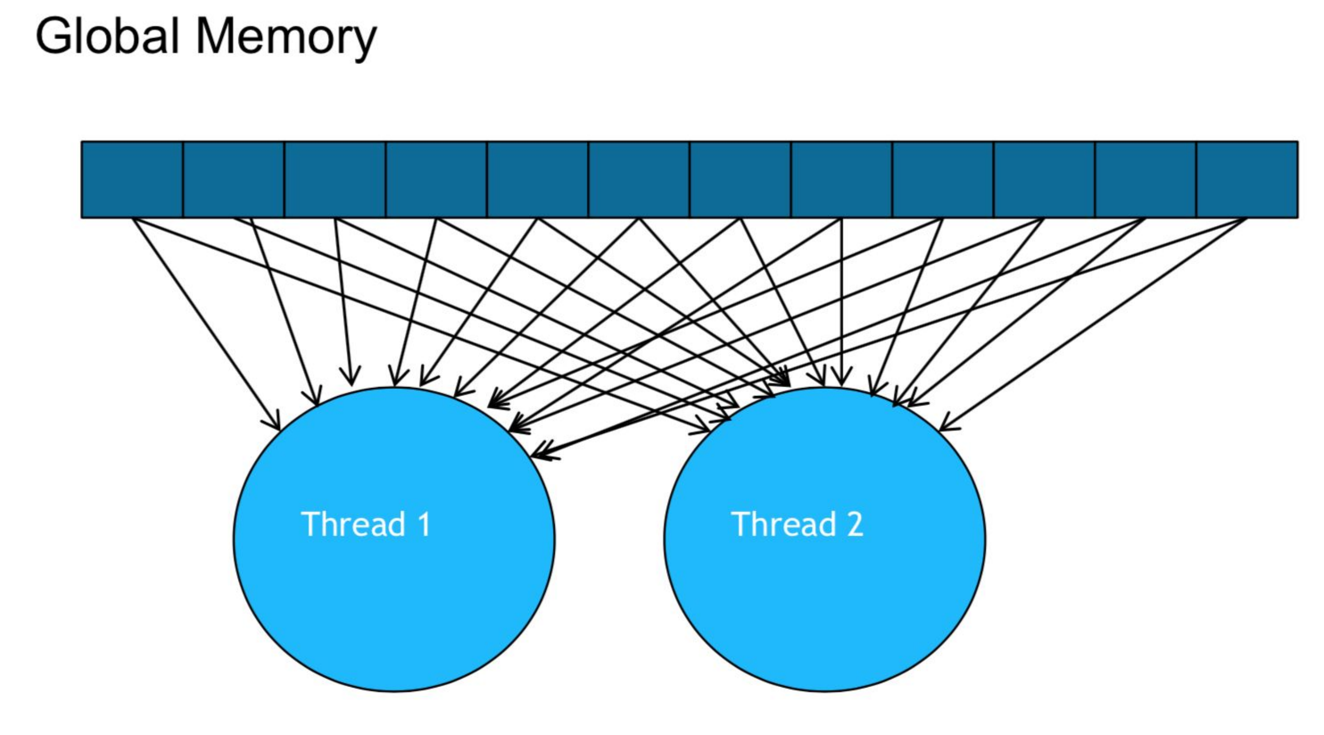
Answer: Use shared memory

What if shared memory is not big enough?

Answer: Tiling

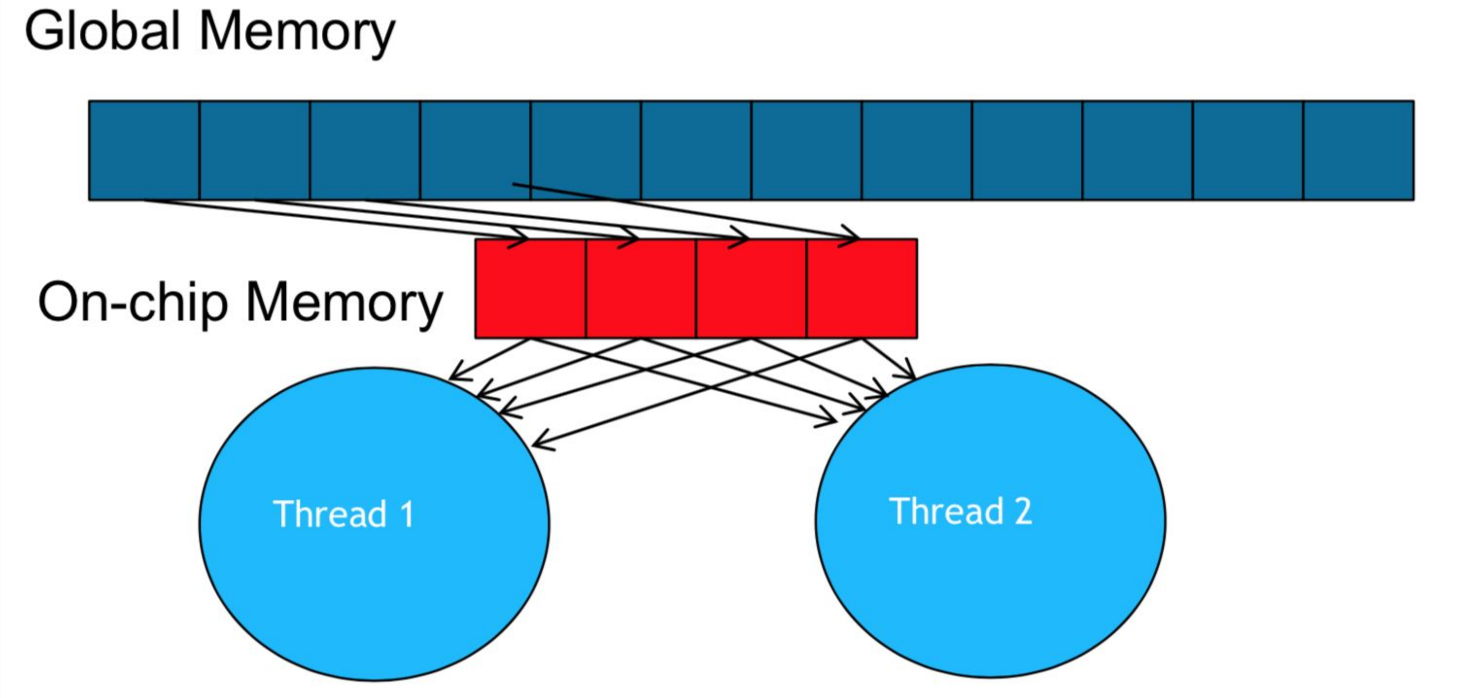


Tiling memory accesses



Global memory accessed in varying order by multiple threads

Tiling memory



Break up the accessed memory into tiles and load them on to shared memory

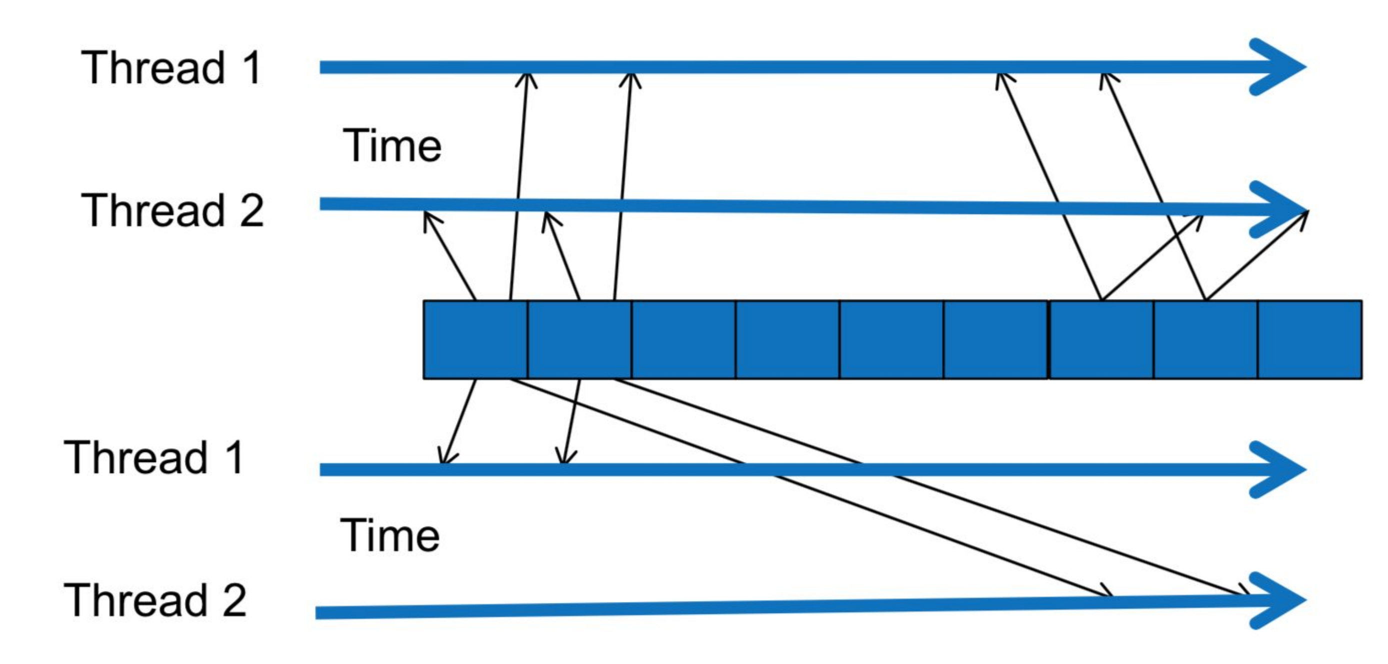
Analogy

- Analogy to carpooling
- When does carpooling work well?

Analogy

- Analogy to carpooling
- When does carpooling work well?
  - When spatial and temporal demands match up

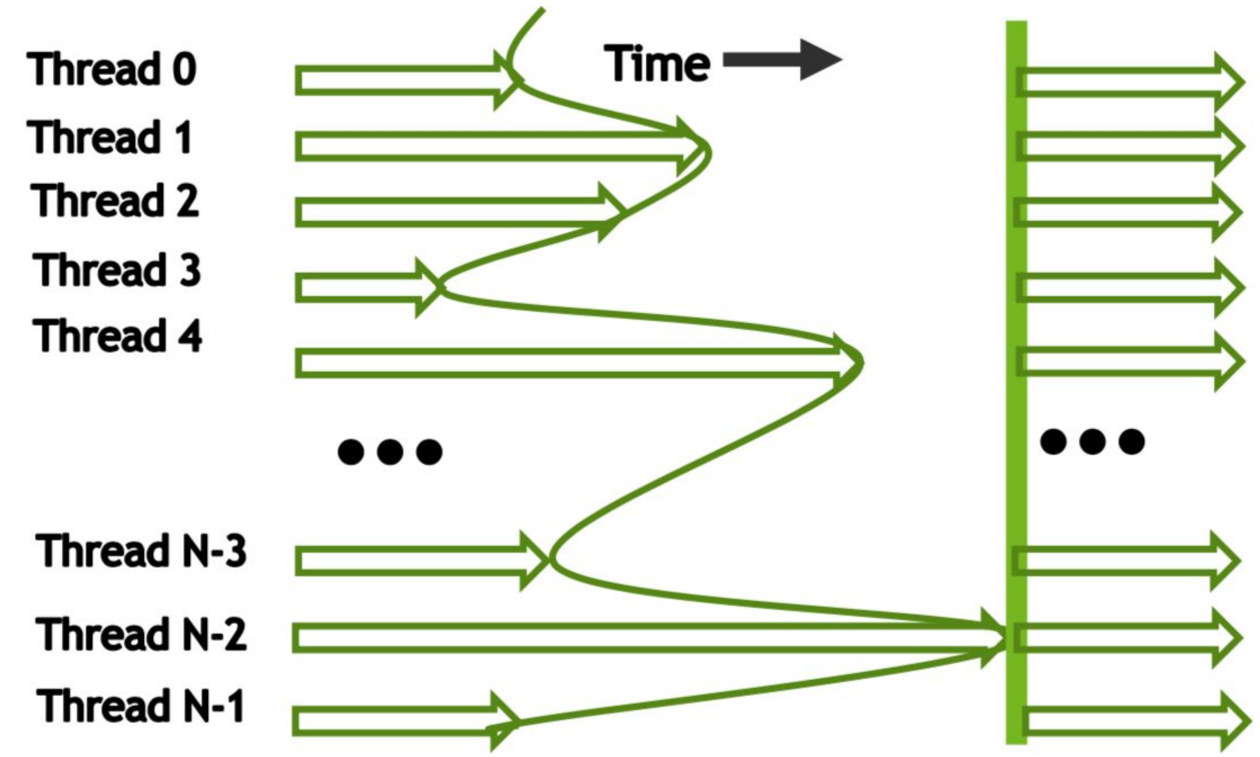
Memory tiling



Memory tiling works when threads require similar regions of the global memory at similar times

We need one more feature to get tiling to work - Synchronize (preview for now)

Synchronization to ensure threads reading memory into shared tiles or operating on them have all completed their work

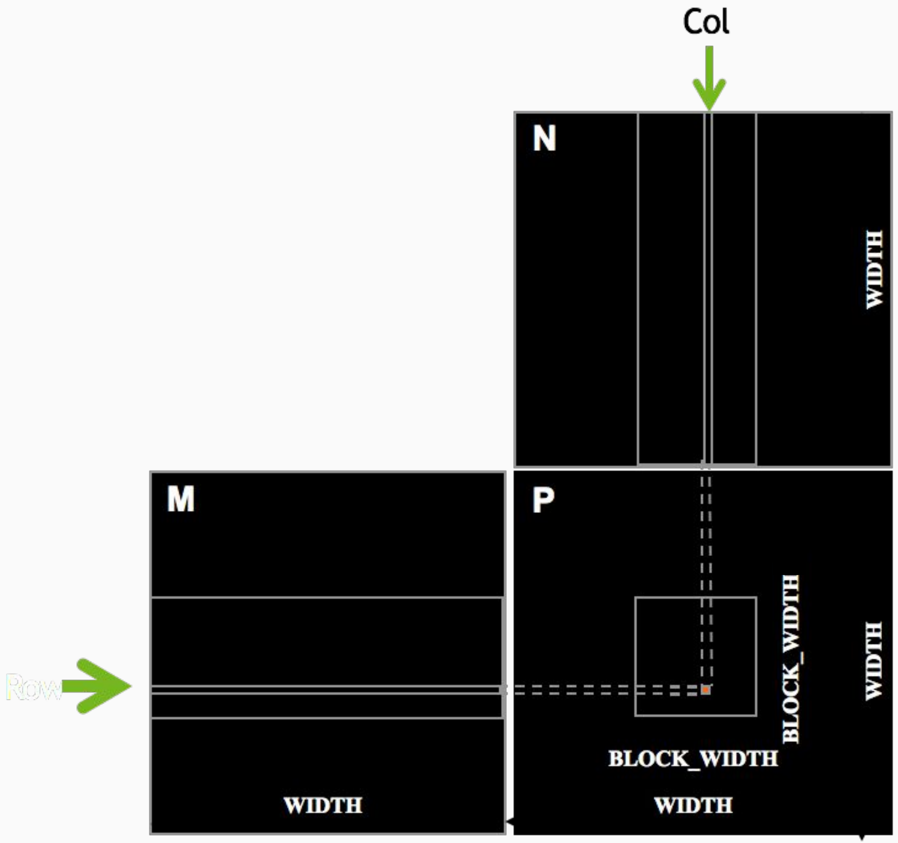


Steps in Tiling

- Analyze memory accesses across threads. If memory access patterns have spatial **and** temporal match, we can use tiling
- Through the different threads, load a tile of memory from the global memory on to the shared memory
- Use synchronization to ensure that all threads have together finished loading the tile
- Through the different threads, operate on the loaded tile in parallel
- Use synchronization to ensure that all threads have together finished all operations
- Move on to the next tile, if any

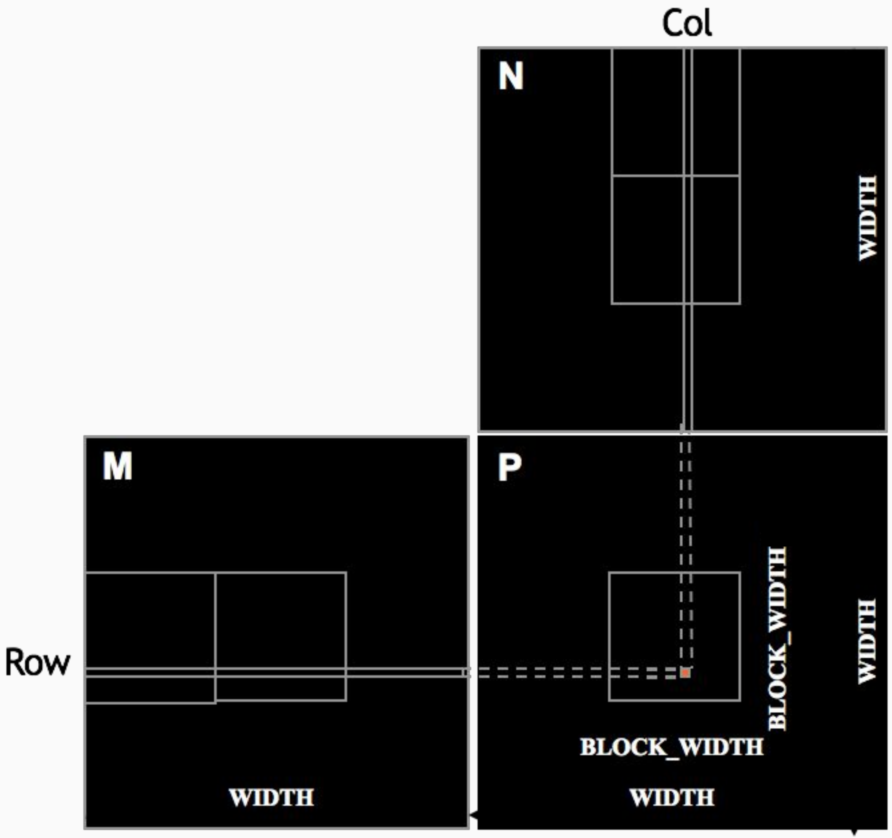
Access pattern of matrix multiplication

- Each thread reads a single row and a single column
- Each block reads a contiguous set of rows and columns

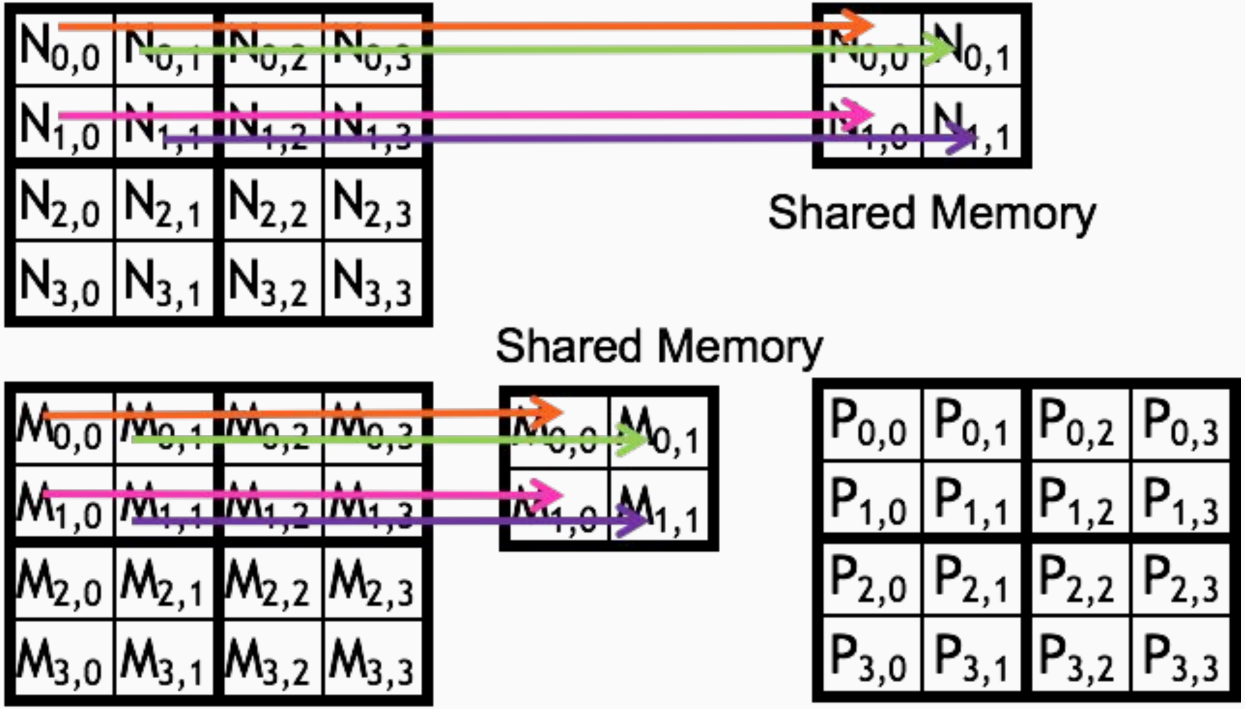


Tiled matrix multiplication

- Split matrix multiplication into phases
- In each phase read a tile of memory on which all threads of a block work
- Tiling procedure:
  - Each thread would then read one item to M and N tiles

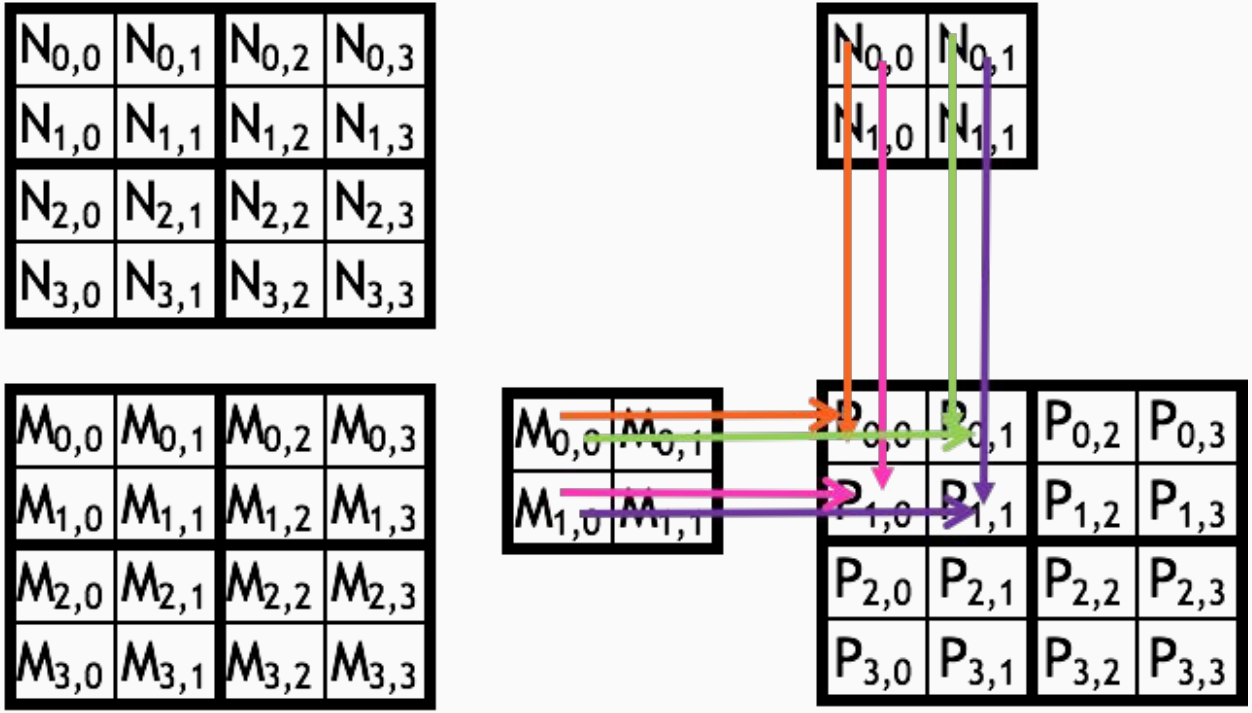


Block (0,0) - Phase 0 - Load



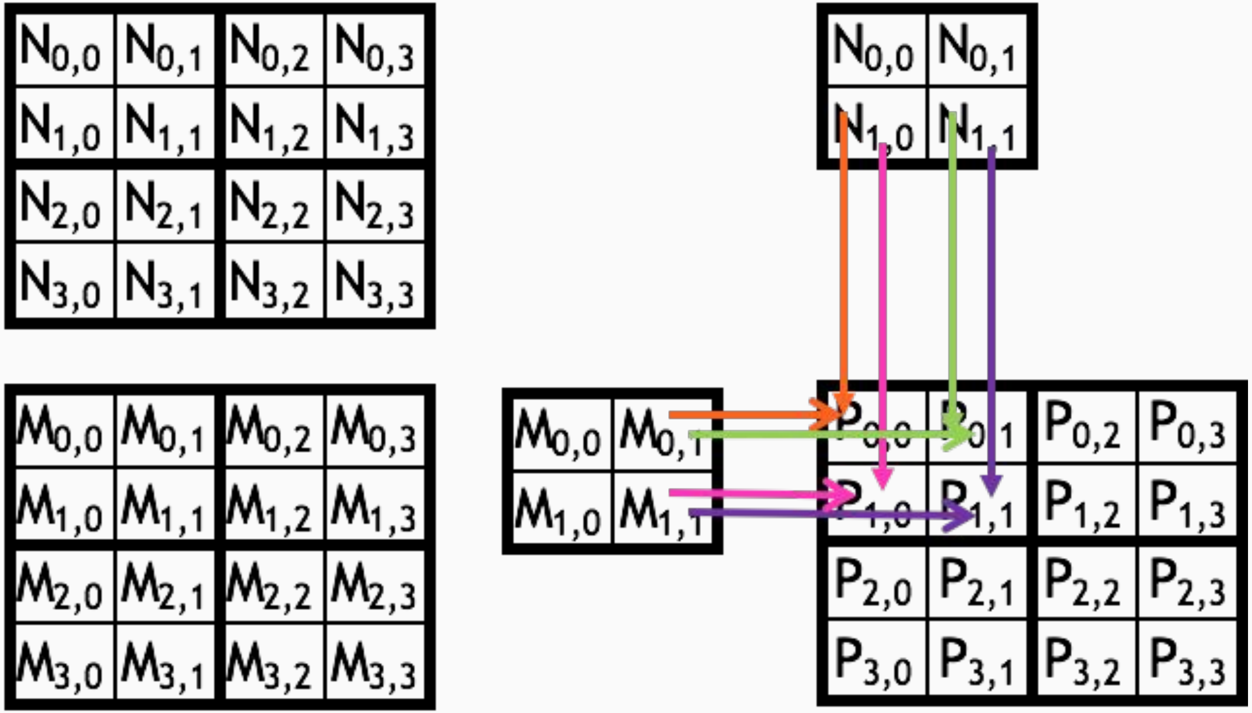
Thread (0,0) - Thread (0,1) - Thread (1,0) - Thread (1,1)

Block (0,0) - Phase 0 - Iteration 0



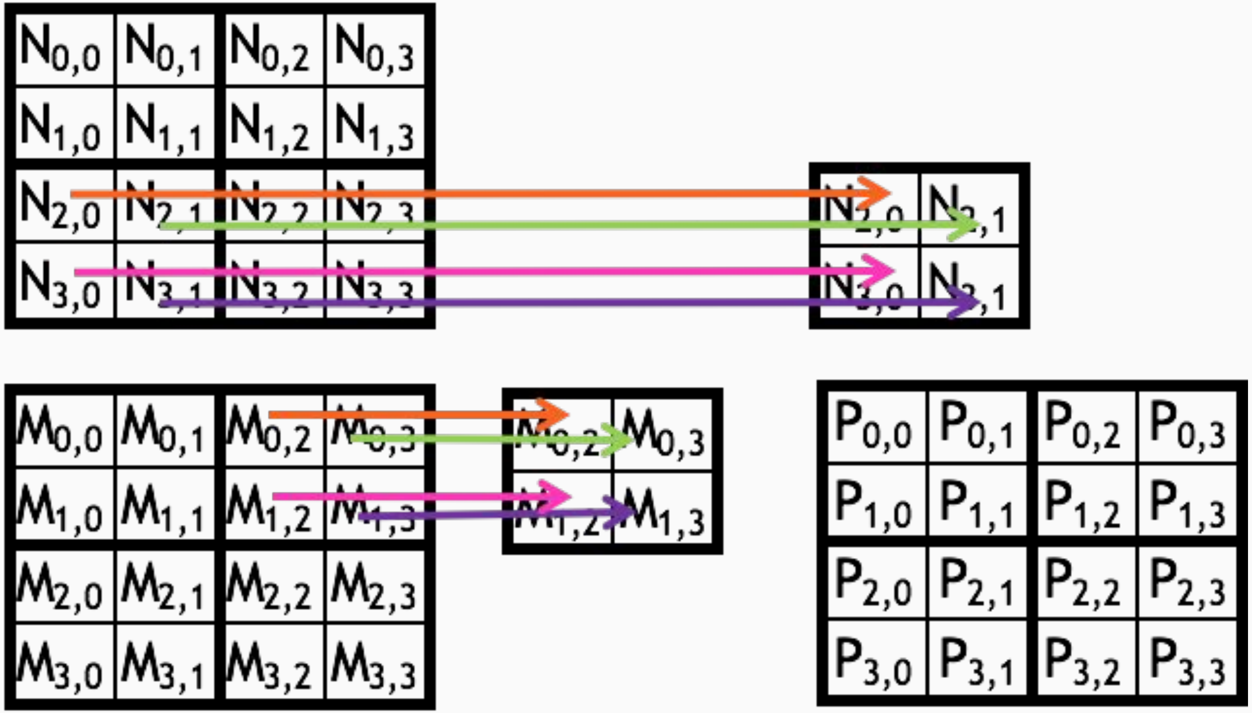
Thread (0,0) - Thread (0,1) - Thread (1,0) - Thread (1,1)

Block (0,0) - Phase 0 - Iteration 1



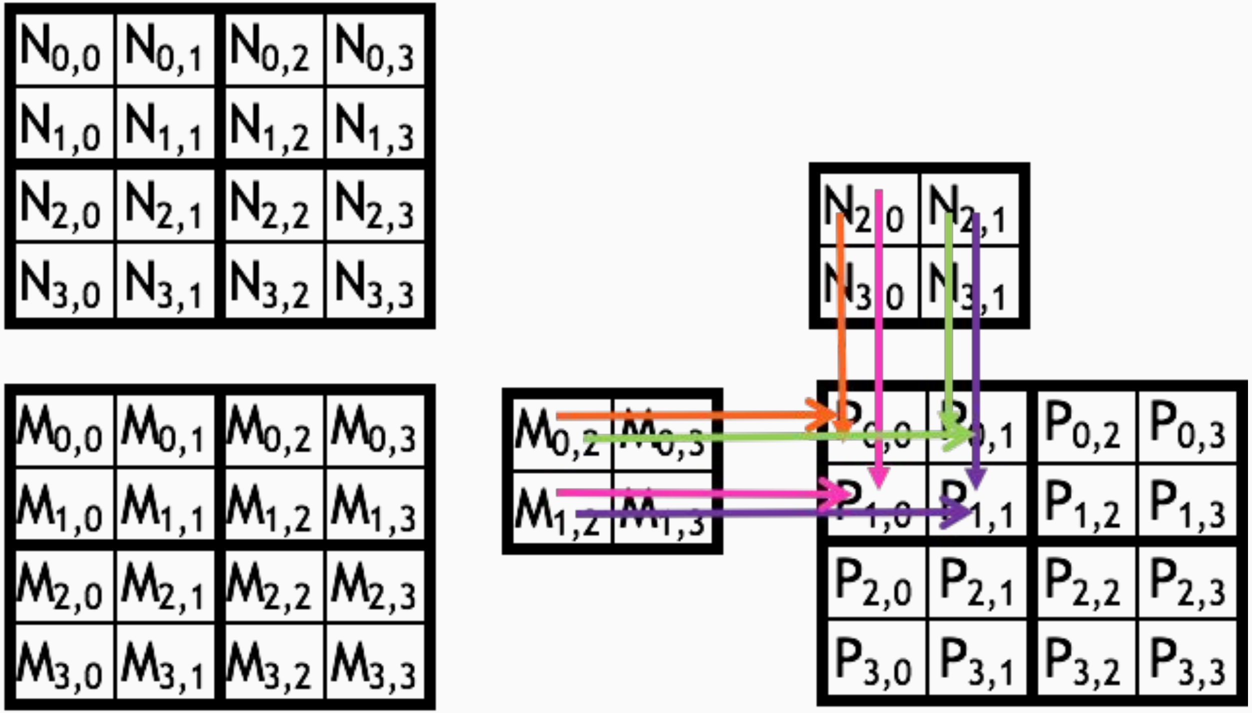
Thread (0,0) - Thread (0,1) - Thread (1,0) - Thread (1,1)

Block (0,0) - Phase 1 - Load



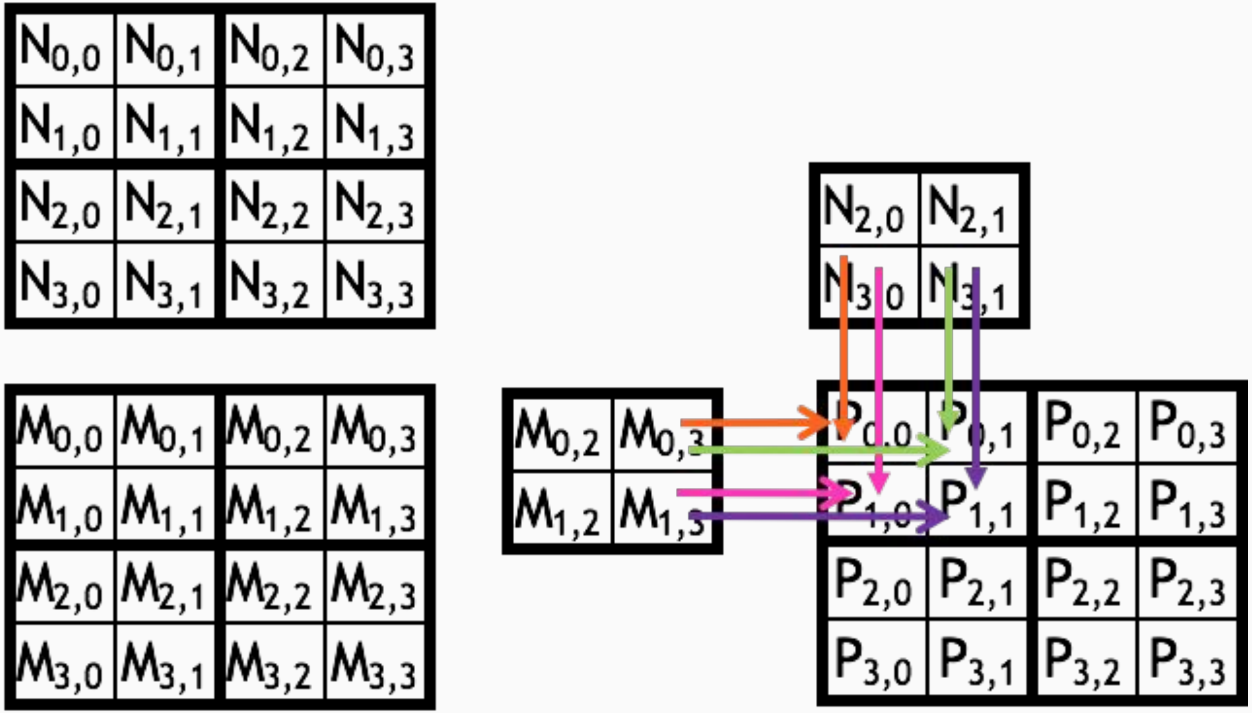
Thread (0,0) - Thread (0,1) - Thread (1,0) - Thread (1,1)

Block (0,0) - Phase 1 - Iteration 0



Thread (0,0) - Thread (0,1) - Thread (1,0) - Thread (1,1)

Block (0,0) - Phase 1 - Iteration 1



Thread (0,0) - Thread (0,1) - Thread (1,0) - Thread (1,1)

Tiling operations in phases

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

time →

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

The CUDA kernel for tiled matrix multiplication

```
__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 < Width/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[(p*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;
}
```

Choosing the tile size

- We have seen how to choose the block, threads based on number of threads / SM, blocks / SM and threads / warp. Now we have an extra parameter tile size
- For tile of size 16 x 16
- For tile of size 32 x 32

Choosing the tile size

- We have seen how to choose the block, threads based on number of threads / SM, blocks / SM and threads / warp. Now we have an extra parameter tile size
- For tile of size 16 x 16

  Number of loads from global memory in each phase = 2 x 16 x 16 = 512 floats

  Number of addition / multiplication operations in each phase = 256 x 16 x 2 = 8,192 ops

  Computational intensity = 8,192 / 512 = 16 ops/float
- For tile of size 32 x 32

Choosing the tile size

- We have seen how to choose the block, threads based on number of threads / SM, blocks / SM and threads / warp. Now we have an extra parameter tile size
- For tile of size 16 x 16

  Number of loads from global memory in each phase = 2 x 16 x 16 = 512 floats

  Number of addition / multiplication operations in each phase = 256 x 16 x 2 = 8,192 ops

  Computational intensity = 8,192 / 512 = 16 ops/float
- For tile of size 32 x 32

  Number of loads from global memory in each phase = 2 x 32 x 32 = 2,048 floats

  Number of addition / multiplication operations in each phase = 1024 x 32 x 2 = 65,536 ops

  Computational intensity = 65,536 / 2048 = 32 ops/float
- Formally argue that for 2^k x 2^k tile, the computational intensity is 2^k ops/float

Choosing the tile size

- But, the full picture is much more complicated

Choosing the tile size

- But, the full picture is much more complicated
- Already seen, that 32x32 tile requires 1,024 threads and if we have 1,536 max. threads per core then we can have only 1 block! For 16x16 tile we can have 6 blocks covering the full 1,536 threads
- Why do we want more threads in an SM?
- But could we want less threads in a block?

Choosing the tile size

- But, the full picture is much more complicated
- Already seen, that 32x32 tile requires 1,024 threads and if we have 1,536 max. threads per core then we can have only 1 block! For 16x16 tile we can have 6 blocks covering the full 1,536 threads
- Why do we want more threads in an SM?
  - In the load stage of any phase, each thread wants to read two floats
  - Thus, for an SM we have 2 * 256 * 6 = 3,072 pending loads => Hide latency
- But could we want less threads in a block?

Choosing the tile size

- But, the full picture is much more complicated
- Already seen, that 32x32 tile requires 1,024 threads and if we have 1,536 max. threads per core then we can have only 1 block! For 16x16 tile we can have 6 blocks covering the full 1,536 threads
- Why do we want more threads in an SM?
  - In the load stage of any phase, each thread wants to read two floats
  - Thus, for an SM we have 2 * 256 * 6 = 3,072 pending loads => Hide latency
- But could we want less threads in a block?
  - `__syncthreads()` adds synchronization which suffers from Amdahl's law => Fewer threads means lesser chance of threads waiting

Next time

- More advanced implications around tiling
  - For underlying DRAM technology, how do we optimize memory (coalescing)
  - What happens when `tile_size` is not a divisor of matrix `width`
    - Warps and control divergence
- Timing CUDA kernels