GPU Programming

Performance Considerations

Miaoqing Huang
University of Arkansas

Outline

Outline

Control Flow Divergence

Memory Coalescing

Shared Memory Bank Conflicts

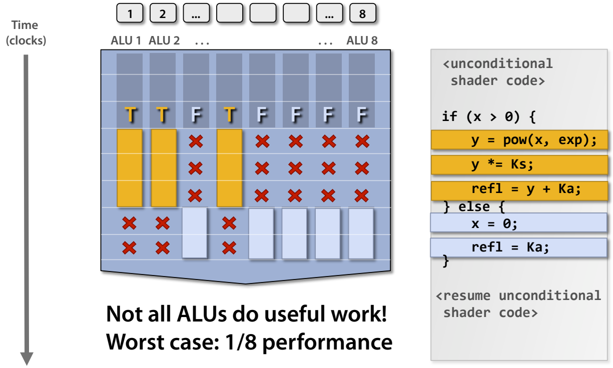
Occupancy

Loop Unrolling

Kernel Launch Overhead

Control Flow

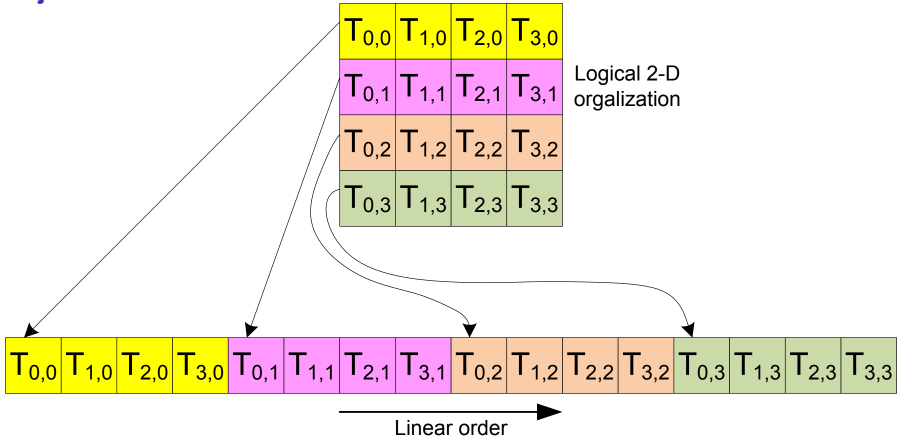
- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - `if-else`, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code



```
<unconditional
 shader code>

if (x > 0) {
      y = pow(x, exp);
      y *= Ks;
      refl = y + Ka;
} else {
      x = 0;
      refl = Ka;
}

<resume unconditional
 shader code>
```

Not all ALUs do useful work!
Worst case: 1/8 performance

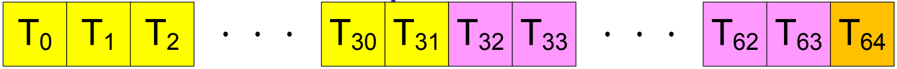
Project the threads into a linear order



$T_{0,0}$ $T_{1,0}$ $T_{2,0}$ $T_{3,0}$
$T_{0,1}$ $T_{1,1}$ $T_{2,1}$ $T_{3,1}$
$T_{0,2}$ $T_{1,2}$ $T_{2,2}$ $T_{3,2}$
$T_{0,3}$ $T_{1,3}$ $T_{2,3}$ $T_{3,3}$
Logical 2-D orgalization
$T_{0,0}$ $T_{1,0}$ $T_{2,0}$ $T_{3,0}$ $T_{0,1}$ $T_{1,1}$ $T_{2,1}$ $T_{3,1}$ $T_{0,2}$ $T_{1,2}$ $T_{2,2}$ $T_{3,2}$ $T_{0,3}$ $T_{1,3}$ $T_{2,3}$ $T_{3,3}$
Linear order

- Line up the row with larger `y` and `z` coordinates after those with lower ones

Partition the threads into warps



Partition the threads into warps

$T_{0,0}$	$T_{1,0}$	$T_{2,0}$	$T_{3,0}$	$T_{4,0}$	$T_{5,0}$	$T_{6,0}$	$T_{7,0}$
$T_{0,1}$	$T_{1,1}$	$T_{2,1}$	$T_{3,1}$	$T_{4,1}$	$T_{5,1}$	$T_{6,1}$	$T_{7,1}$
$T_{0,2}$	$T_{1,2}$	$T_{2,2}$	$T_{3,2}$	$T_{4,2}$	$T_{5,2}$	$T_{6,2}$	$T_{7,2}$
$T_{0,3}$	$T_{1,3}$	$T_{2,3}$	$T_{3,3}$	$T_{4,3}$	$T_{5,3}$	$T_{6,3}$	$T_{7,3}$
$T_{0,4}$	$T_{1,4}$	$T_{2,4}$	$T_{3,4}$	$T_{4,4}$	$T_{5,4}$	$T_{6,4}$	$T_{7,4}$
$T_{0,5}$	$T_{1,5}$	$T_{2,5}$	$T_{3,5}$	$T_{4,5}$	$T_{5,5}$	$T_{6,5}$	$T_{7,5}$
$T_{0,6}$	$T_{1,6}$	$T_{2,6}$	$T_{3,6}$	$T_{4,6}$	$T_{5,6}$	$T_{6,6}$	$T_{7,6}$
$T_{0,7}$	$T_{1,7}$	$T_{2,7}$	$T_{3,7}$	$T_{4,7}$	$T_{5,7}$	$T_{6,7}$	$T_{7,7}$

Avoid diverging within a warp

Example with divergence

```
if (threadIdx.x > 2) {
    ...
}
else {
    ...
}
```

- Branch granularity < warp size

Avoid diverging within a warp

Example with divergence

```
if (threadIdx.x > 2) {
    ...
}
else {
    ...
}
```

- Branch granularity < warp size

Example without divergence

```
if ((threadIdx.x / WARP_SIZE) > 2) {
    ...
}
else {
    ...
}
```

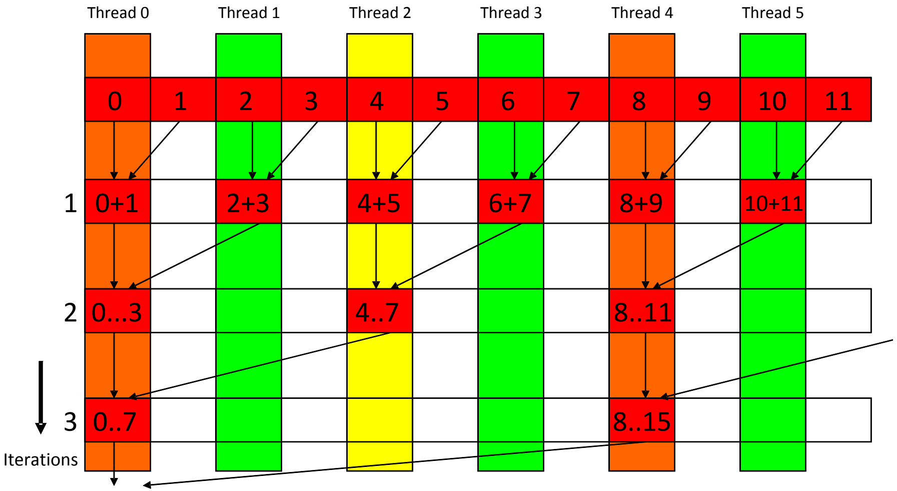
- Branch granularity is a whole multiple of warp size

Example: Divergent Iteration

```
__global__ void per_thread_sum(int *indices,float *data,float *sums)
{
  ...
  // number of loop iterations is data dependent
  for(int j=indices[i];j<indices[i+1]; j++) {
    sum += data[j];
  }
  sums[i] = sum;
}
```

- A single thread can drag a whole warp with it for a long time
- Know your data patterns
  - If data is unpredictable, try to flatten peaks by letting threads work on multiple data items

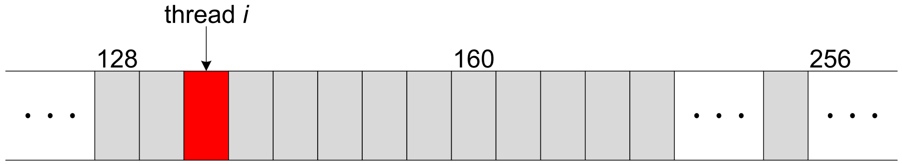
Execution of the Sum Reduction Kernel

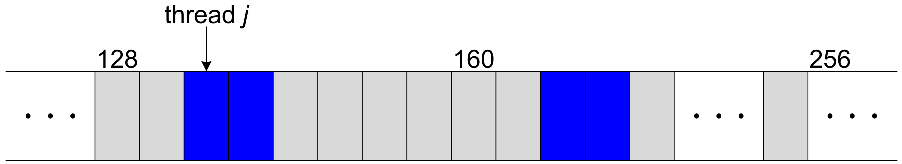


Outline

Memory Coalescing

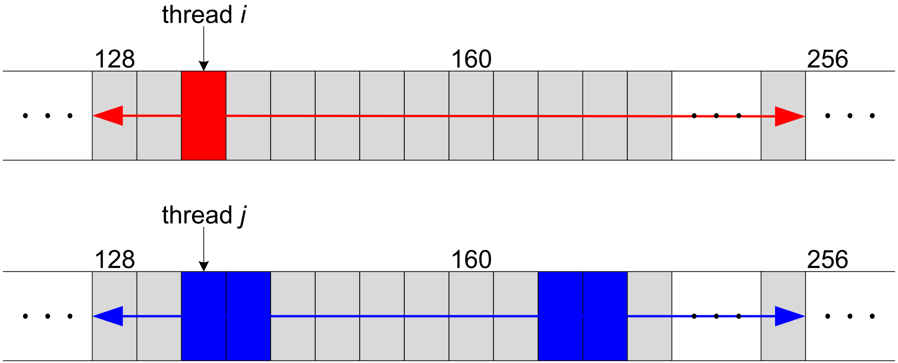
- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don't use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 32, 64, 128 bytes





Memory Coalescing

- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don't use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 32, 64, 128 bytes

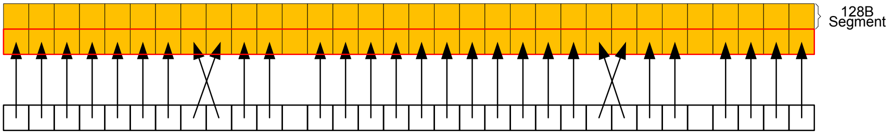


Coalesced Access to Global Memory

- How is the global memory access of the threads in a warp coalesced?
  - On Fermi/Kepler GPUs, global memory loads and stores by threads of a warp (i.e., 32 threads) are coalesced
- How is the coalesced memory access aligned into segments?
  - The segment size is 128 Bytes if the data are cached in both L1 and L2
- Thread blocks are partitioned into warps based on thread indices
  - Each warp contains threads of consecutive and increasing thread IDs with the first warp containing thread 0

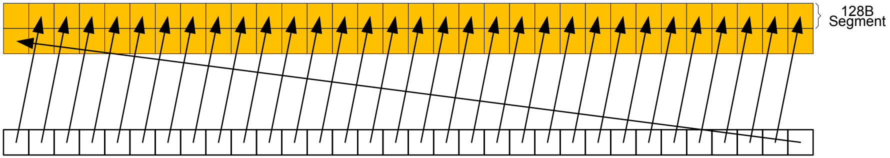
Simple Memory Access Pattern

- Coalesced access in which all threads but a few access the words in a segment
  - Not all threads in a warp need to access the memory
  - The access by threads can be permuted



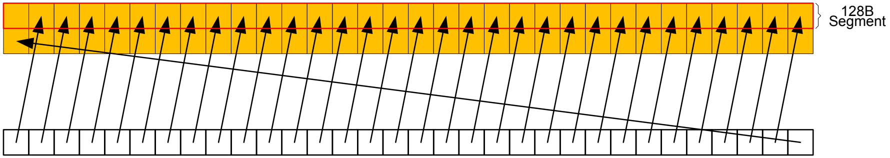
Misaligned Access Pattern

- Misaligned sequential addresses that fall within two 128-byte segments



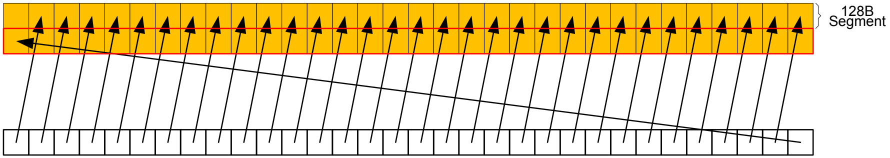
Misaligned Access Pattern

- Misaligned sequential addresses that fall within two 128-byte segments



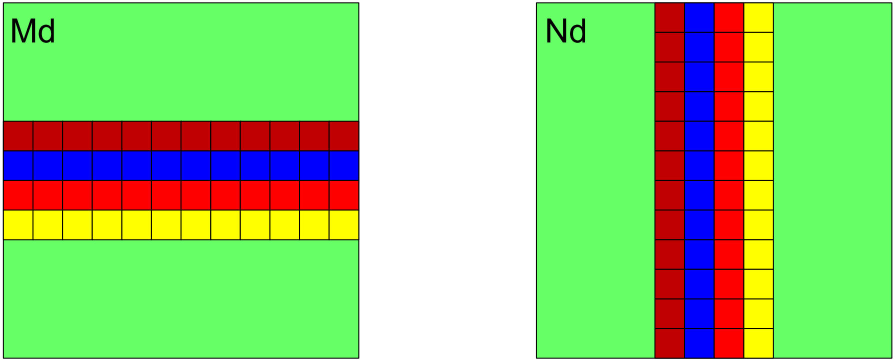
Misaligned Access Pattern

- Misaligned sequential addresses that fall within two 128-byte segments



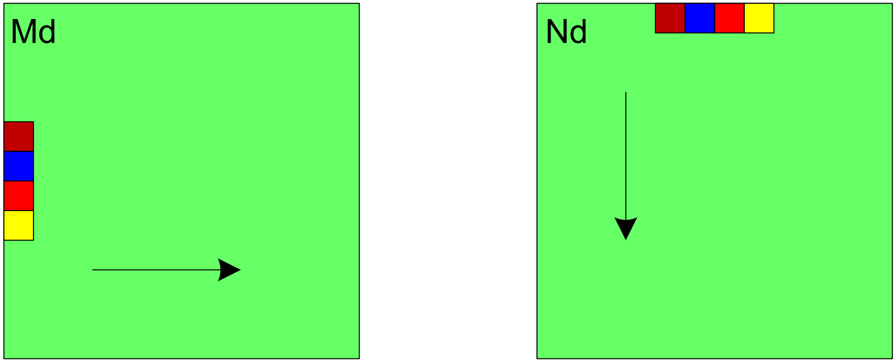
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



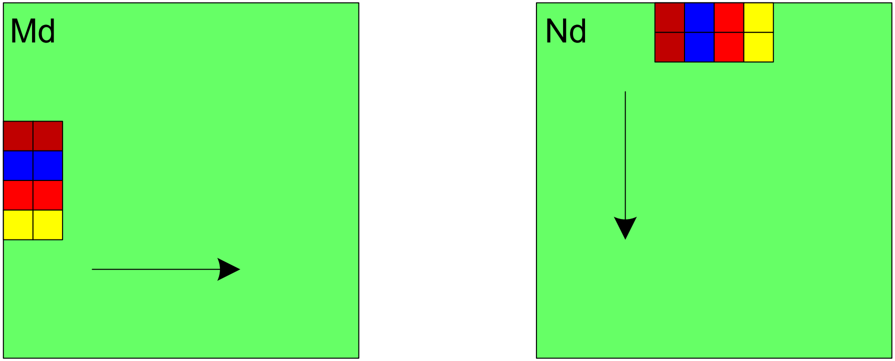
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



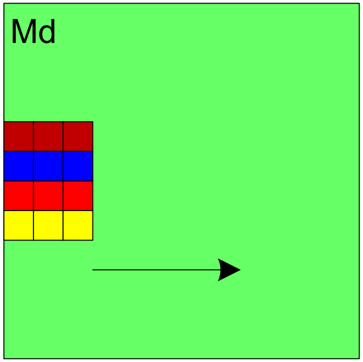
Matrix Data Access Pattern

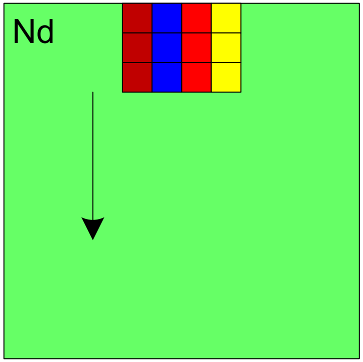
- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



Matrix Data Access Pattern

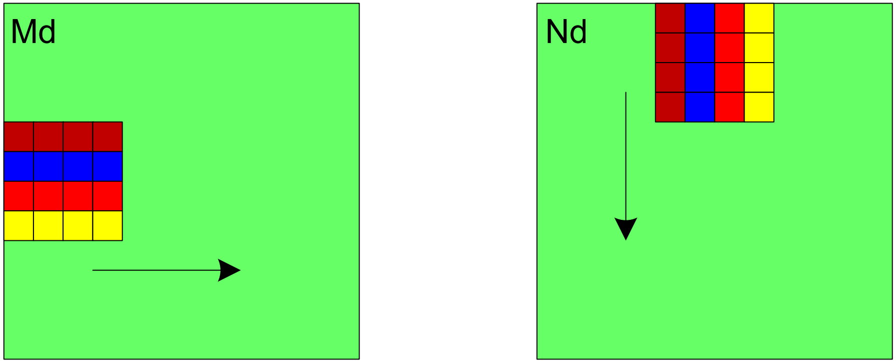
- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`





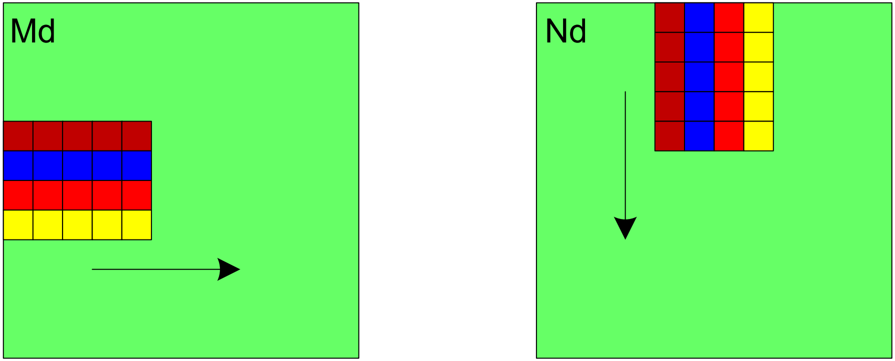
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



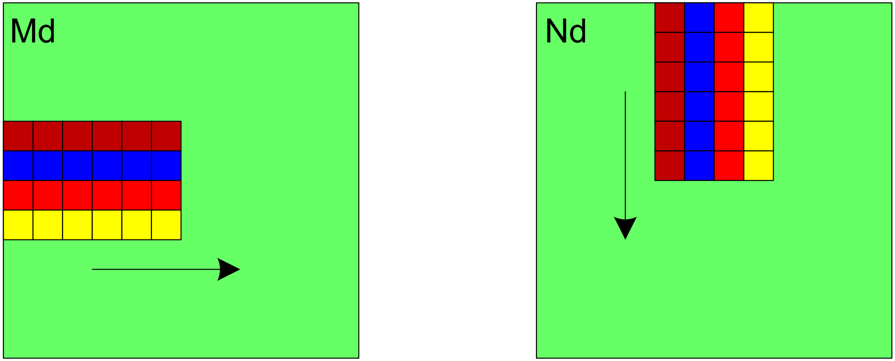
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



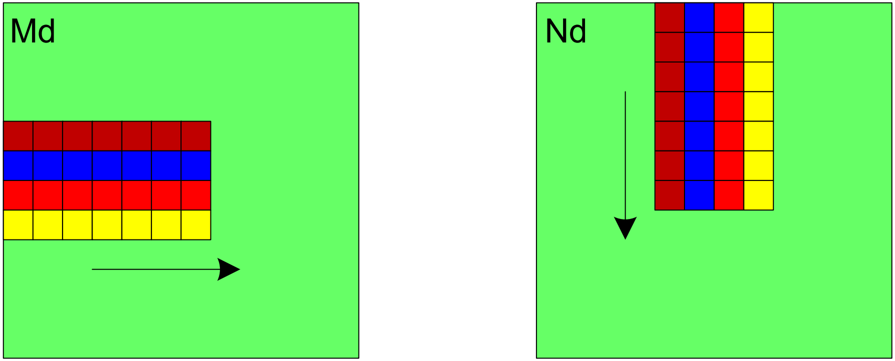
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



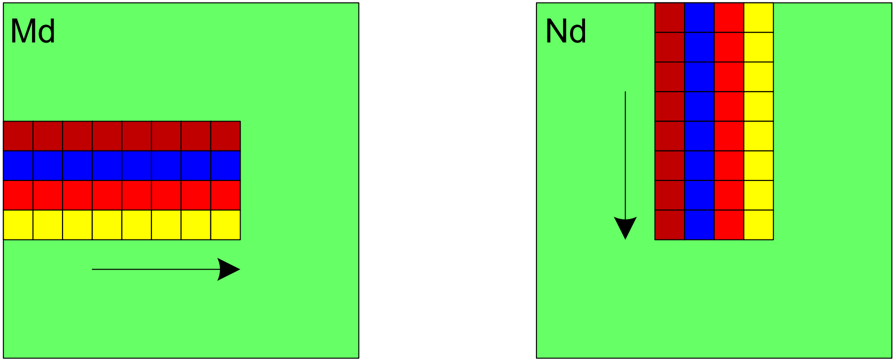
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



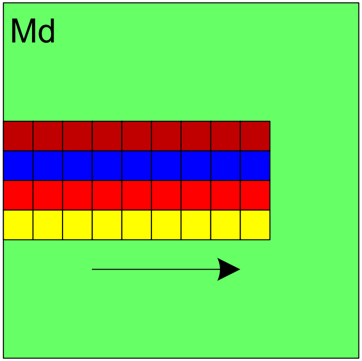
Matrix Data Access Pattern

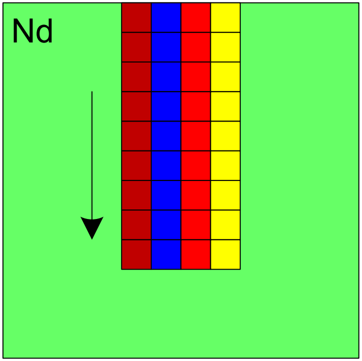
- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



Matrix Data Access Pattern

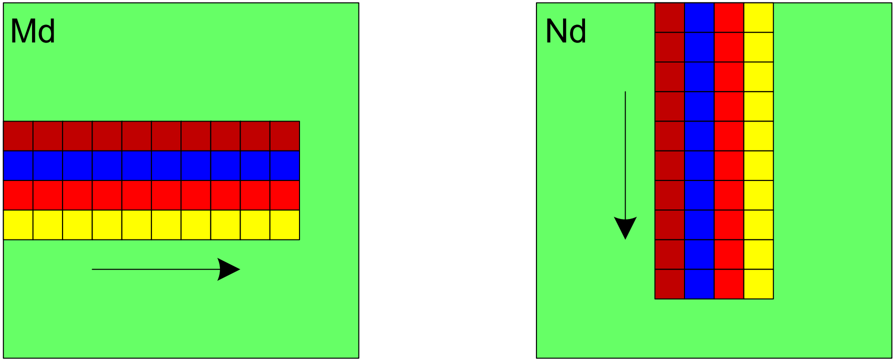
- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`





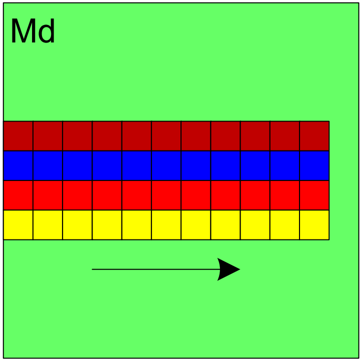
Matrix Data Access Pattern

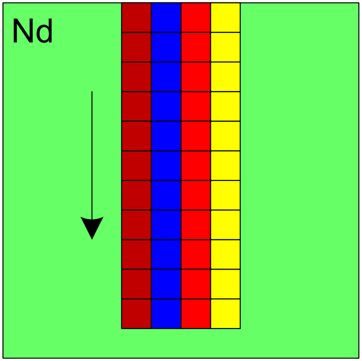
- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



Matrix Data Access Pattern

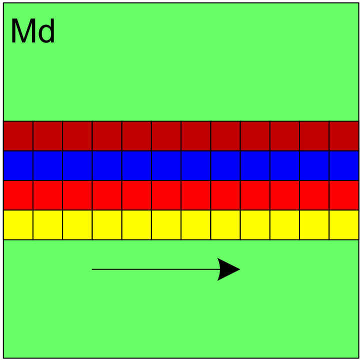
- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`

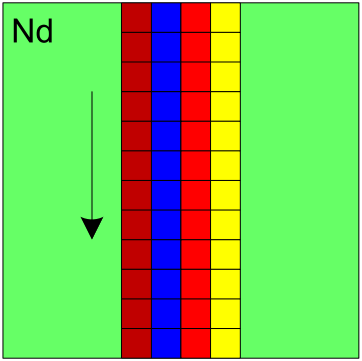




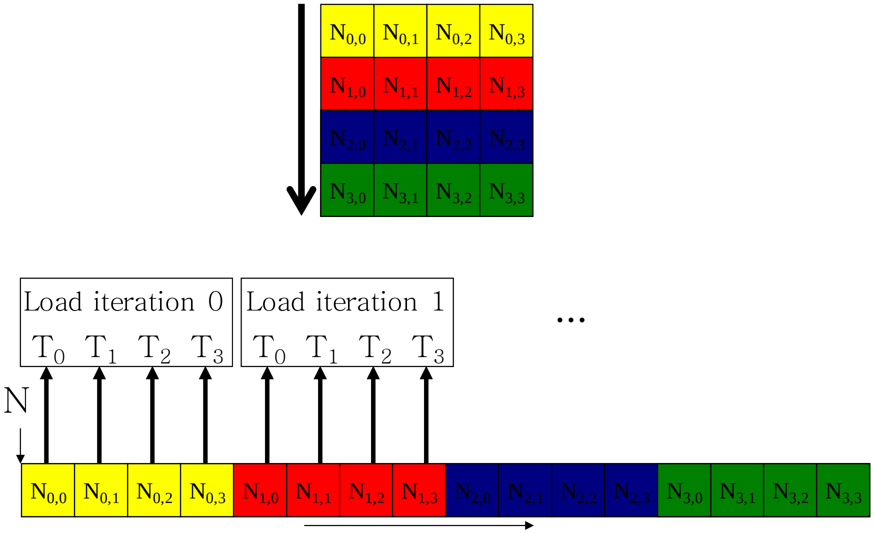
Matrix Data Access Pattern

- Directly access data from global memory
  - Each thread reads one row of `Md` and one column of `Nd`



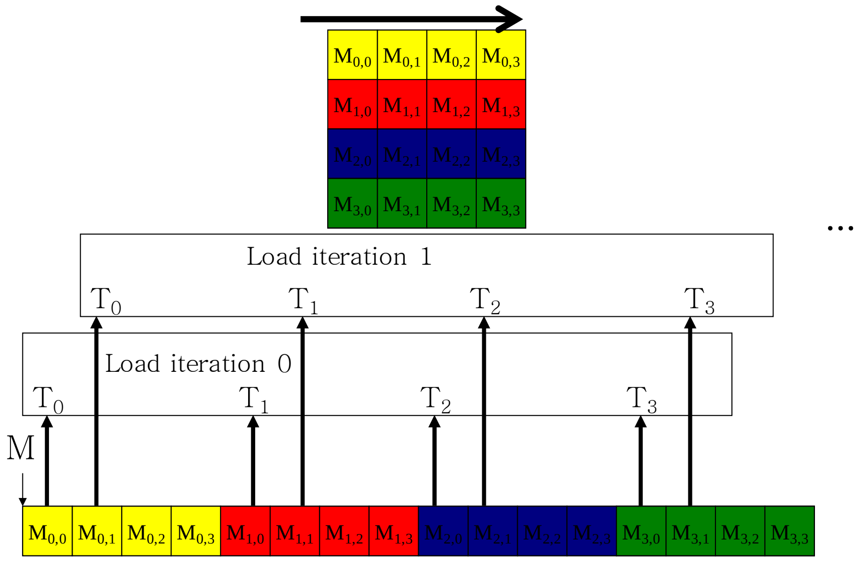


Coalesced access to a matrix



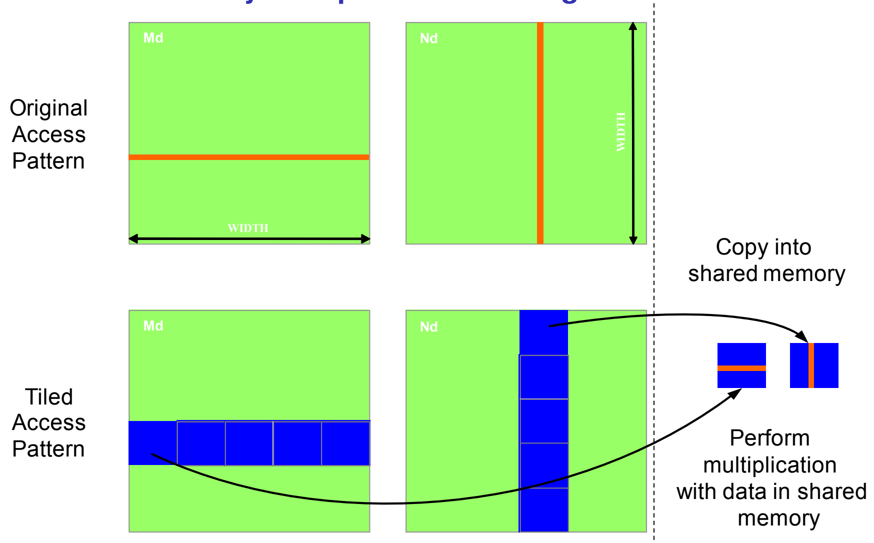
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
...
N

Uncoalesced access to a matrix

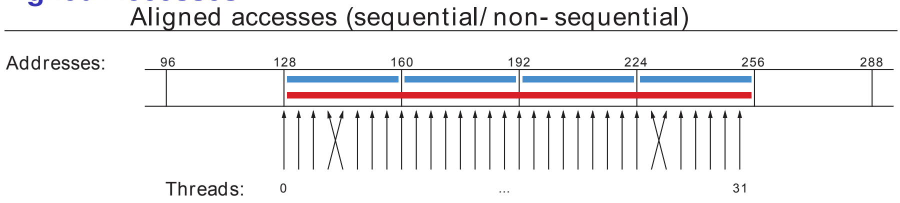


$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
...
Load iteration 1
T_0 T_1 T_2 T_3
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

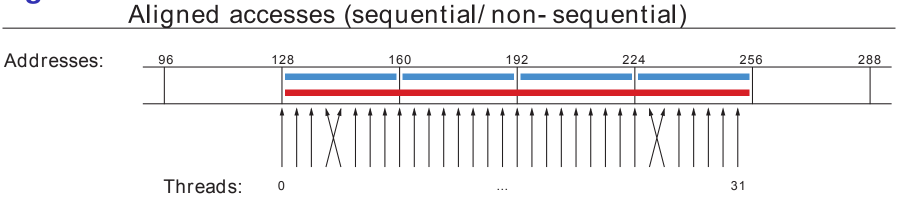
Use Shared Memory to Improve Coalescing



Aligned Accesses



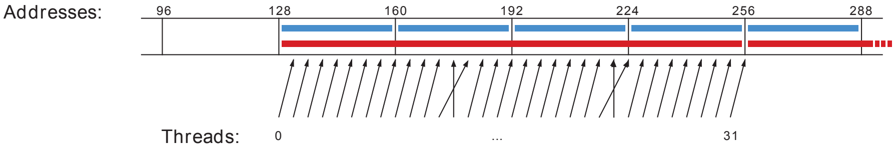
Aligned Accesses



Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

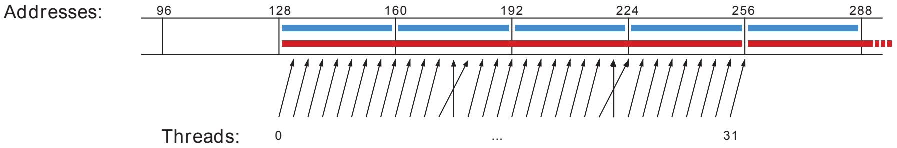
Misaligned Accesses

Mis-aligned accesses (sequential/non-sequential)



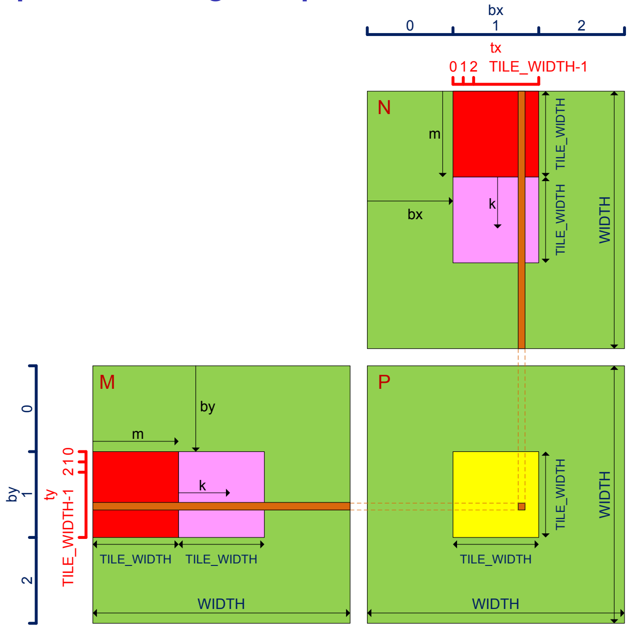
Misaligned Accesses

Mis-aligned accesses (sequential/non-sequential)



Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Matrix Multiplication Using Multiple Blocks with Tile



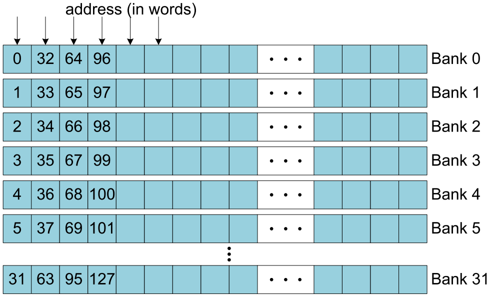
Matrix Multiplication Kernel Using Multiple Blocks with Tile

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

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

    float Pvalue = 0;

// Loop over the Md and Nd tiles required to compute the Pd element
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
       Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
       Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
       __syncthreads();

       for (int k = 0; k < TILE_WIDTH; ++k) {
           Pvalue += Mds[ty][k] * Nds[k][tx];
       }
       __syncthreads();
    }

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

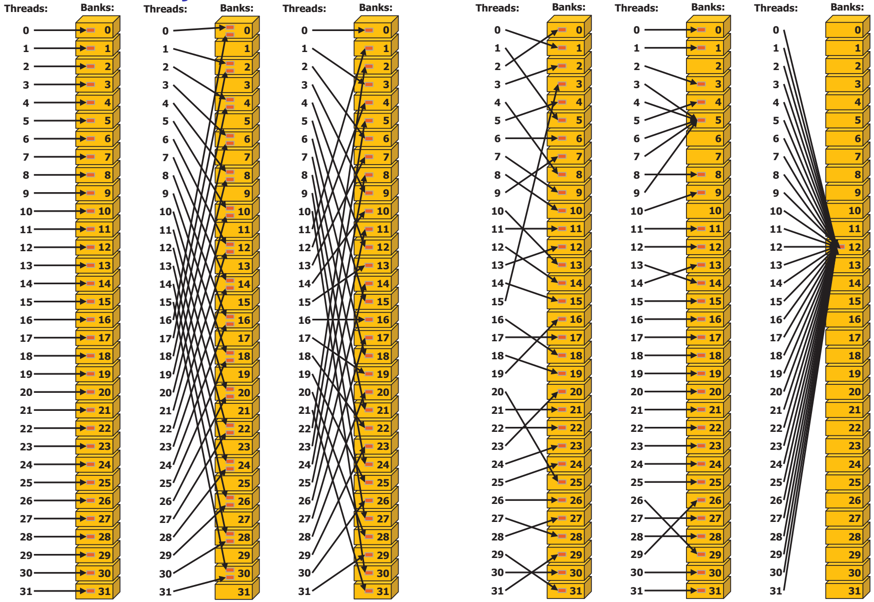
Outline

Shared Memory

- Shared memory is banked (e.g., 32 banks)
  - Consecutive 32-bit words are in different banks
  - Simultaneous & concurrent access
    - All the threads in the same warp share the same request
- If two or more threads access the same bank but different words, get bank conflicts
  - Multiple threads access the same bank for same word $\longrightarrow$ no bank conflict



Shared Memory Access



Matrix Multiplication Kernel Using Multiple Blocks with Tile

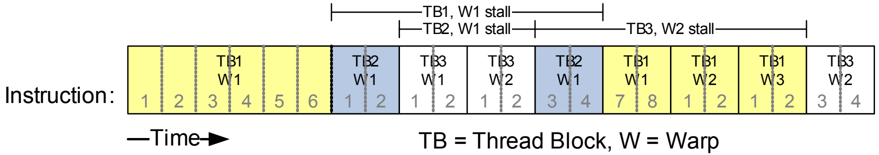
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int width)
{
    __shared __float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared __float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    // Identify the row and column of the Pd element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
            Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
       Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
       __syncthreads();

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

- Do we have a *shared memory bank conflict* problem here?

Outline

Reminder: Thread Scheduling

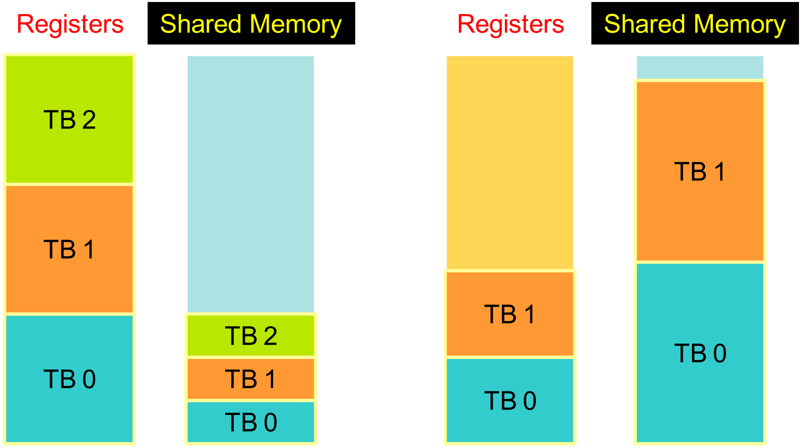
- SM implements zero-overhead warp scheduling
  - At any time, only a few warps are executed by SM
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



Thread Scheduling

- What happens if all warps are stalled?
  - No instruction issued $\longrightarrow$ performance lost
- Most common reason for stalling
  - Waiting on global memory
- If your code reads global memory every couple of instructions
  - Try to maximize occupancy
- What determines occupancy?
  - Register usage per thread
    - Registers are dynamically partitioned across all blocks assigned to the SM
    - Once assigned to a block, a register is NOT accessible by threads in other blocks
    - Each thread in the same block only access registers assigned to itself
  - Shared memory per thread block

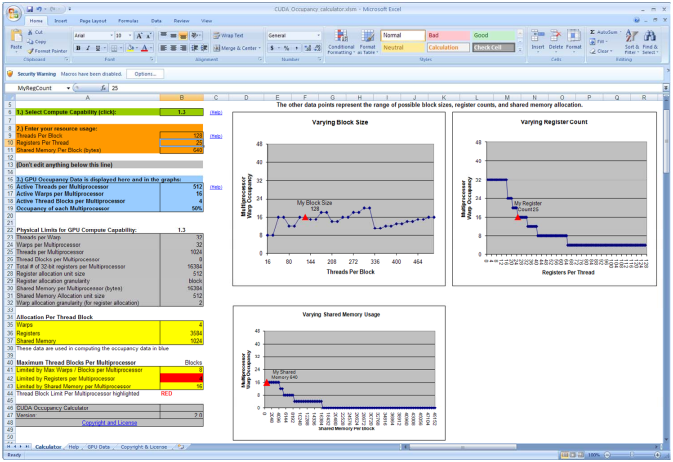
Resource Limits



- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized $\longrightarrow$ no more thread blocks

How do you know what you’re using?

- Use `nvcc main.cu --ptxas-options=-v` to get register & shared memory usage
- Plug those numbers into CUDA Occupancy Calculator
- How to influence how many registers you use?
  - Pass option `-maxrregcount=X` to nvcc


CUDA Occupancy calculator.xlsm - Microsoft Excel
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.
Varying Block Size
Multiprocessor Warp Occupancy
My Block Size 128
Threads Per Block
Varying Register Count
My Register Count 25
Registers Per Thread
Varying Shared Memory Usage
My Shared Memory 640
Shared Memory Per Block

MyRegCount | 25

	A	B	C
6	1.) Select Compute Capability (click):	1.3	(Help)
8	2.) Enter your resource usage:		
9	Threads Per Block	128	(Help)
10	Registers Per Thread	25	
11	Shared Memory Per Block (bytes)	640	
13	(Don't edit anything below this line)		
15	3.) GPU Occupancy Data is displayed here and in the graphs:		
16	Active Threads per Multiprocessor	512	(Help)
17	Active Warps per Multiprocessor	16	
18	Active Thread Blocks per Multiprocessor	4	
19	Occupancy of each Multiprocessor	50%	
22	Physical Limits for GPU Compute Capability:	1.3	
23	Threads per Warp	32	
24	Warps per Multiprocessor	32	
25	Threads per Multiprocessor	1024	
26	Thread Blocks per Multiprocessor	8	
27	Total # of 32-bit registers per Multiprocessor	16384	
28	Register allocation unit size	512	
29	Register allocation granularity	block	
30	Shared Memory per Multiprocessor (bytes)	16384	
31	Shared Memory Allocation unit size	512	
32	Warp allocation granularity (for register allocation)	2	
34	Allocation Per Thread Block		
35	Warps	4	
36	Registers	3584	
37	Shared Memory	1024	
38	These data are used in computing the occupancy data in blue		
40	Maximum Thread Blocks Per Multiprocessor	Blocks	
41	Limited by Max Warps / Blocks per Multiprocessor	8	
42	Limited by Registers per Multiprocessor	4	
43	Limited by Shared Memory per Multiprocessor	16	
44	Thread Block Limit Per Multiprocessor highlighted	RED	
46	CUDA Occupancy Calculator		
47	Version:	2.0	
48	Copyright and License		

Outline

Limited Processing Bandwidth of An SM

```
    for (int k = 0; k < BLOCK_SIZE; ++K) {
        Pvalue += Ms[ty][k] * Ns[k][tx];
    }
```

- How many instructions are required to be carried out in each iteration?

Limited Processing Bandwidth of An SM

```
    for (int k = 0; k < BLOCK_SIZE; ++K) {
        Pvalue += Ms[ty][k] * Ns[k][tx];
    }
```

- How many instructions are required to be carried out in each iteration?
  - Two floating-point arithmetic instructions
  - One loop branch instruction
  - Two address arithmetic instruction
  - One loop counter increment instruction

Limited Processing Bandwidth of An SM

```
    for (int k = 0; k < BLOCK_SIZE; ++K) {
        Pvalue += Ms[ty][k] * Ns[k][tx];
    }
```

- How many instructions are required to be carried out in each iteration?
  - Two floating-point arithmetic instructions
  - One loop branch instruction
  - Two address arithmetic instruction
  - One loop counter increment instruction
  - Only $\frac{1}{3}$ of the instructions executed are for real computation!!!

Loop Unrolling

```
    // Assume BLOCK_SIZE = 16
    Pvalue = Ms[ty][0] * Ns[0][tx] + Ms[ty][1] * Ns[1][tx] + ...
             + Ms[ty][15] * Ns[15][tx];
```

- Loop branch instructions $\longrightarrow$ gone
- Loop counter increment instructions $\longrightarrow$ gone
- Address arithmetic instructions $\longrightarrow$ gone
  - Indices are constants
  - Compiler is able to eliminate address arithmetic instructions
- Only floating-point arithmetic instructions are still there

Loop Unrolling

```
    // Assume BLOCK_SIZE = 16
    Pvalue = Ms[ty][0] * Ns[0][tx] + Ms[ty][1] * Ns[1][tx] + ...
             + Ms[ty][15] * Ns[15][tx];
```

- Loop branch instructions $\longrightarrow$ gone
- Loop counter increment instructions $\longrightarrow$ gone
- Address arithmetic instructions $\longrightarrow$ gone
  - Indices are constants
  - Compiler is able to eliminate address arithmetic instructions
- Only floating-point arithmetic instructions are still there
  - Close to peak performance!!!

Outline

Kernel Launch Overhead

- Kernel launches are not free
  - A null kernel launch will take non-trivial time
  - Actual number changes with HW generations and driver software
  - If you are launching lots of small grids you will lose substantial performance due to this effect
- Independent kernel launches are cheaper than dependent kernel launches
  - Dependent launch: Some readback to the cpu
- If you are reading back data to the cpu for control decisions, consider doing it on the GPU
  - Even though the GPU is slow at serial tasks, can do surprising amounts of work before you used up kernel launch overhead