

# Lecture 7

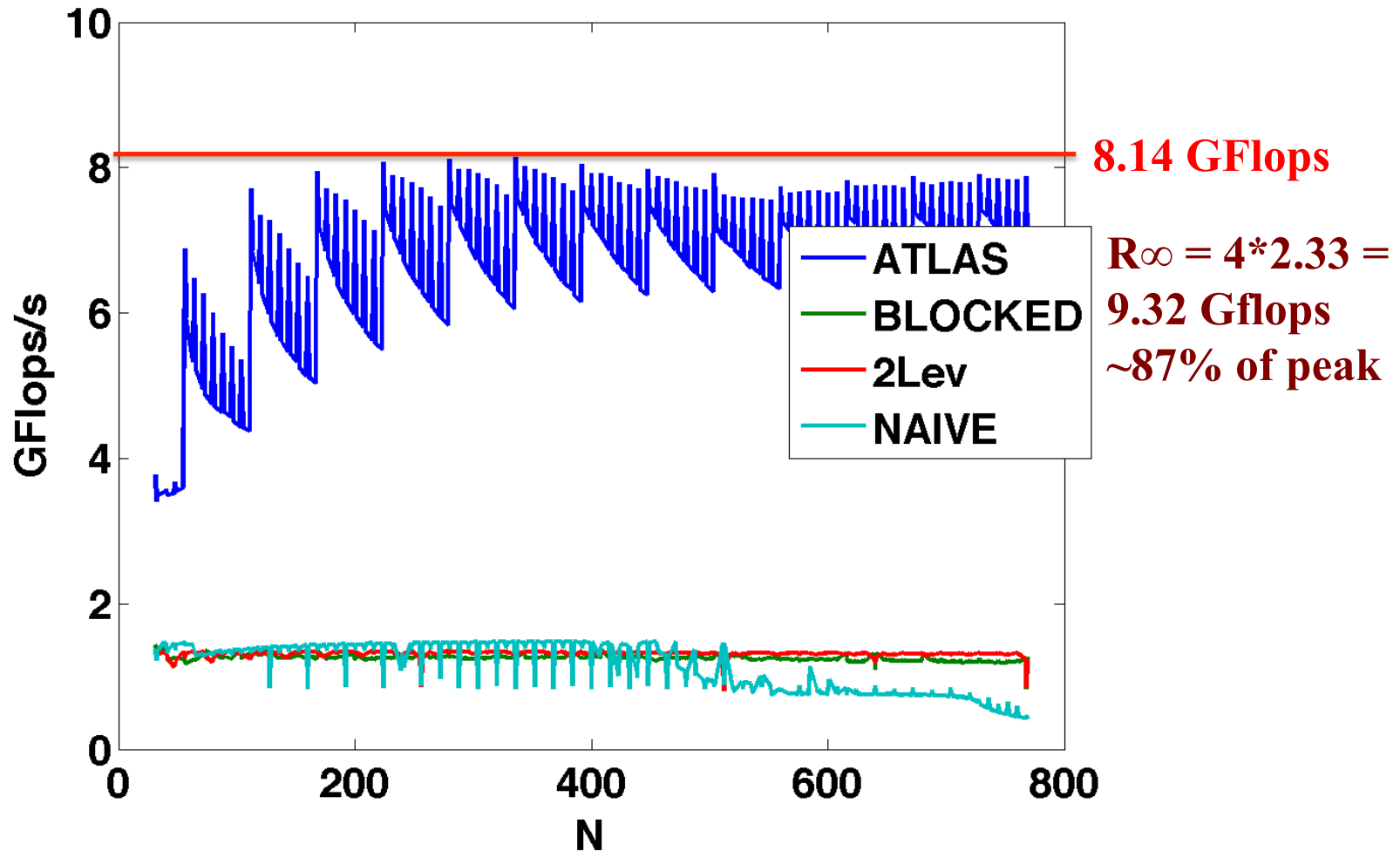
## Using Shared Memory

Performance programming and the memory hierarchy

# Announcements

# Assignment #1

- Blocking for cache will boost performance but a lot more is needed to approach ATLAS' performance

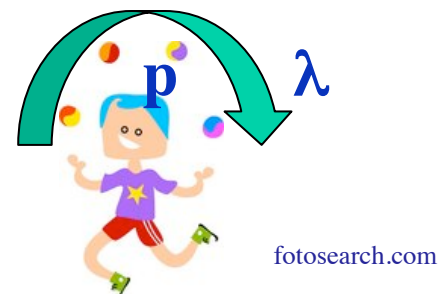


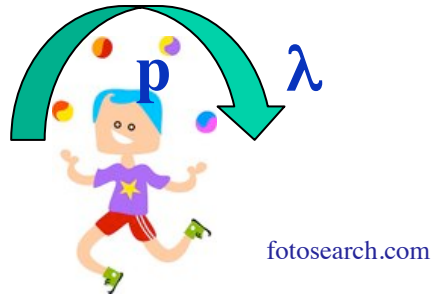
# Today's lecture

- Occupancy
- Improving Matrix Multiplication performance with shared memory
- Memory coalescing
- Avoiding bank conflicts

# Occupancy

- A minimum number of warps needed to hide memory latency
- **Occupancy:  $\# \text{ active warps} \div \text{max } \# \text{ warps supported by vector unit}$**
- Limited by vector unit resources
  - ◆ Amount of shared memory
  - ◆ Number of registers
  - ◆ Maximum number of threads
- Consider a kernel (16x16 block size)
  - ◆ Shared memory/block = 2648 bytes
  - ◆ Reg/thread=38 [38\*256 =9728 < 16k]
  - ◆ # available registers is the limiting factor
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - ◆ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - ◆ Register consumption
- Maximizing the occupancy may not maximize performance





## Occupancy Calculator

[http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)

# Determining occupancy

- Recall the definition for occupancy  
 $\# \text{ active warps} \div \text{max } \# \text{ warps supported by vector unit}$
- NVIDIA provides an occupancy calculator
- Determine resource usage from nvcc for provided A2 code on Dirac (capability 2.0)  
nvcc --ptxas-options=-v  
Used 25 registers, 64 bytes cmem[0]

# Occupancy calculation with 16 x 16 threads

$$\text{Occupancy} = \frac{\# \text{ active warps per SM}}{\text{Maximum possible \# active warps}}$$

## Physical Limits for GPU Compute Capability:

2.0

|                                                |       |
|------------------------------------------------|-------|
| Threads per Warp                               | 32    |
| Warps per Multiprocessor                       | 48    |
| Threads per Multiprocessor                     | 1536  |
| Thread Blocks per Multiprocessor               | 8     |
| Total # of 32-bit registers per Multiprocessor | 32768 |
| Register allocation unit size                  | 64    |
| Register allocation granularity                | warp  |
| Registers per Thread                           | 63    |
| Shared Memory per Multiprocessor (bytes)       | 49152 |
| Shared Memory Allocation unit size             | 128   |
| Warp allocation granularity                    | 2     |
| Maximum Thread Block Size                      | 1024  |

## CUDA GPU Occupancy Calculator

|                                               |       |
|-----------------------------------------------|-------|
| 1.) Select Compute Capability (click):        | 2.0   |
| 1.b) Select Shared Memory Size Config (bytes) | 49152 |

### 2.) Enter your resource usage:

|                                 |     |
|---------------------------------|-----|
| Threads Per Block               | 256 |
| Registers Per Thread            | 25  |
| Shared Memory Per Block (bytes) | 0   |

(Don't edit anything below this line)

### 3.) GPU Occupancy Data

|                                         |      |
|-----------------------------------------|------|
| Active Threads per Multiprocessor       | 1024 |
| Active Warps per Multiprocessor         | 32   |
| Active Thread Blocks per Multiprocessor | 4    |
| Occupancy of each Multiprocessor        | 67%  |

= Allocatable  
Blocks Per  
SM

| Allocated Resources                                     | Per Block | Limit Per SM | SM |
|---------------------------------------------------------|-----------|--------------|----|
| Warps (Threads Per Block / Threads Per Warp)            | 8         | 48           | 6  |
| Registers (Warp limit per SM due to per-warp reg count) | 8         | 38           | 4  |
| Shared Memory (Bytes)                                   | 0         | 49152        | 8  |

Note: SM is an abbreviation for (Streaming) Multiprocessor

### Maximum Thread Blocks Per Multiprocessor

Blocks/SM \* Warps/Block = Warps/SM

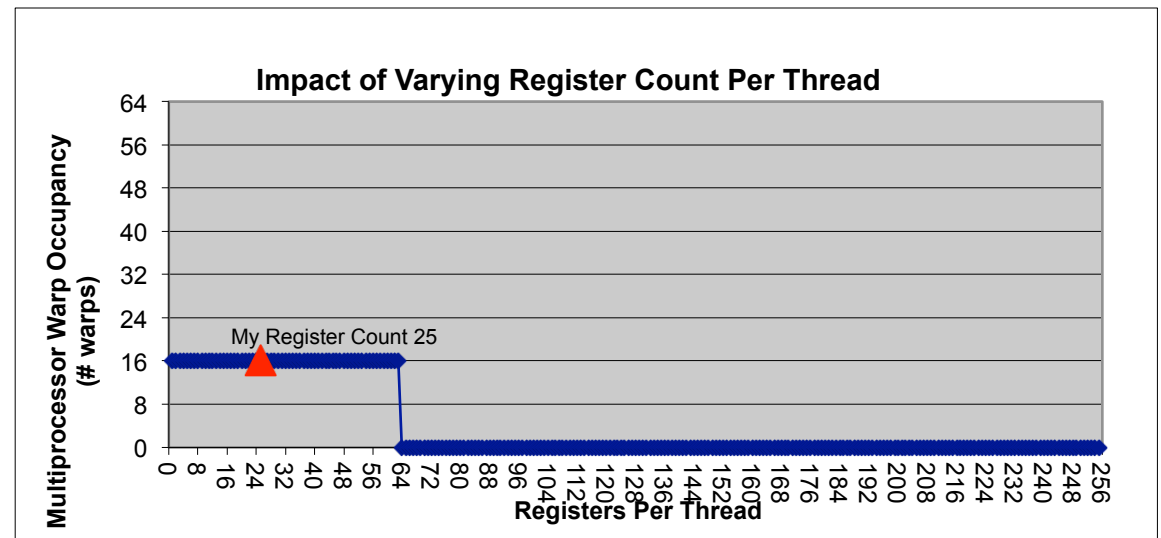
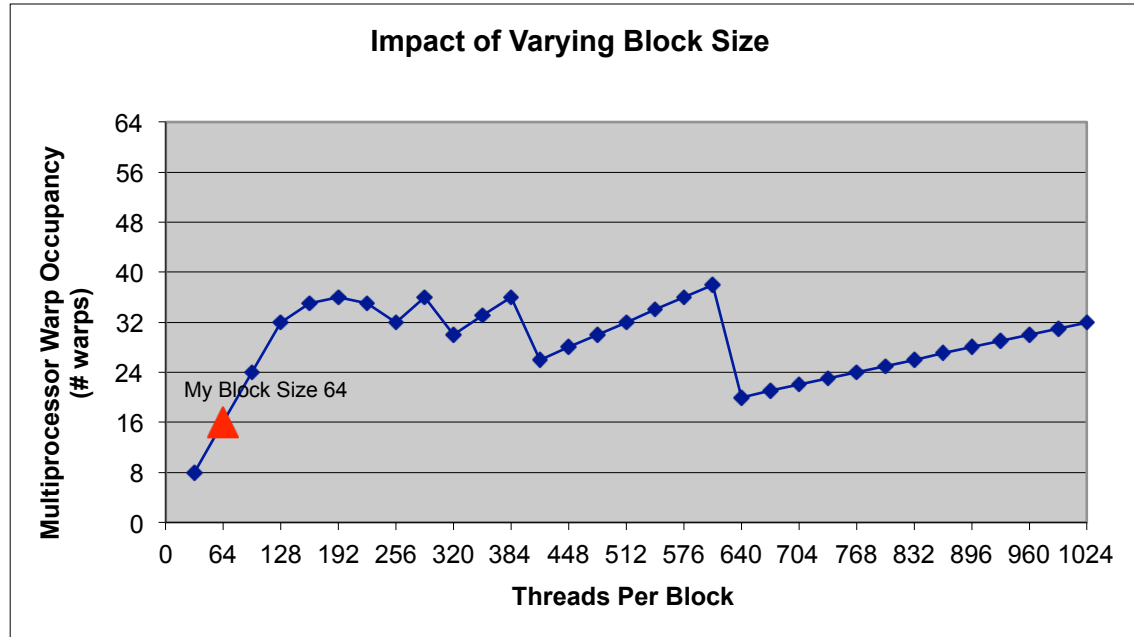
|                                                       |   |   |    |
|-------------------------------------------------------|---|---|----|
| Limited by Max Warps or Max Blocks per Multiprocessor | 6 |   |    |
| Limited by Registers per Multiprocessor               | 4 | 8 | 32 |
| Limited by Shared Memory per Multiprocessor           | 8 |   |    |

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48

Occupancy = 32 / 48 = 67%

# Full occupancy



## Summary - Programming issues

- Branches serialize execution within a warp
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - ♦ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - ♦ Register consumption
  - ♦ Scheduling: hide latency
- Shared memory and registers do not persist across kernel invocations
- Next: using shared memory

# Today's lecture

- Occupancy
- Improving Matrix Multiplication performance with shared memory
- Memory coalescing
- Avoiding bank conflicts

# Speeding up matrix multiply

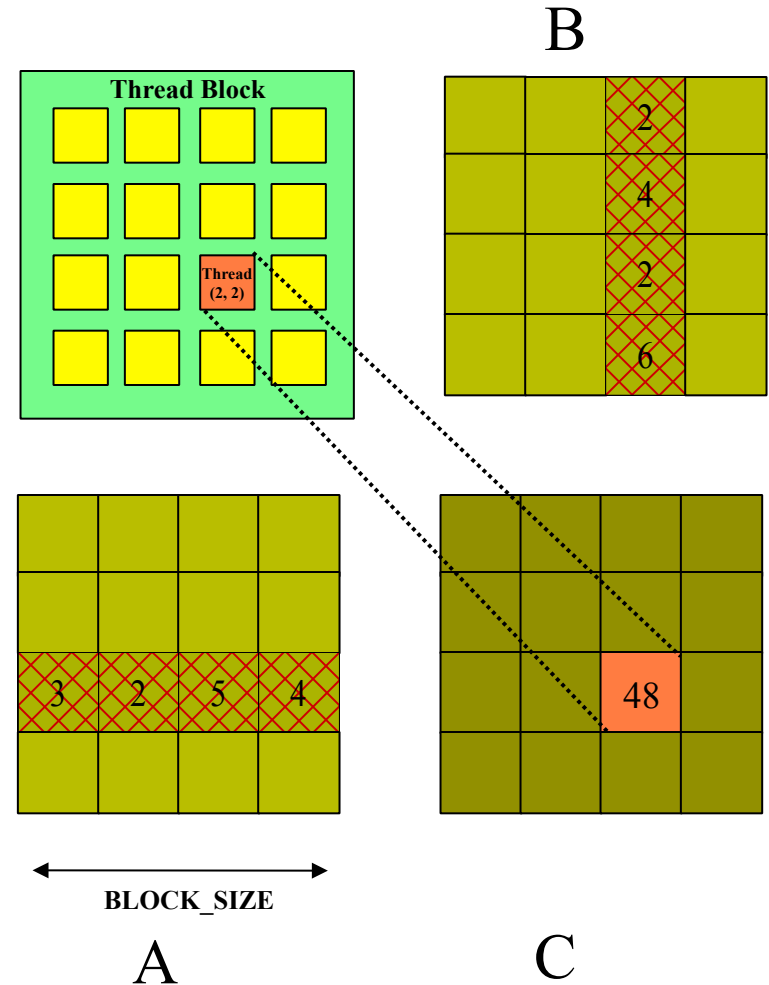
- Use shared memory to increase re-use
- Avoid thread divergence
- Memory Coalescing, avoid Bank Conflicts

## Shared Memory/Cache

- On-chip local store: part shared memory, part L1 \$
  - 16KB shared memory + 48 KB L1 cache
  - 48KB shared memory + 16 KB L1 cache
  - 1 for each vector unit
  - All threads in a block share this on-chip memory
    - A collection of warps share a portion of the local store
- Cache accesses to local or global memory, including temporary register spills
- L2 cache shared by all vector units
- Cache inclusion ( $L1 \subset L2$ ) partially configurable on per-access basis with mem. ref. instruction modifiers
- 128 byte cache line size
- Set the mode using `cudaFuncSetCacheConfig()`  
`cudaFuncSetCacheConfig( boundariesX, PREFERENCE )`  
`PREFERENCE` = {`cudaFuncCachePreferShared`, `cudaFuncCachePreferL1`}

# Naïve kernel implementation

- Each thread computes one element of C
  - ◆ Loads a row of matrix A
  - ◆ Loads a column of matrix B
  - ◆ Computes a dot product
- Every value of A and B is loaded N times from global memory



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

# Naïve Kernel

```
__global__ void
matMul(int N DOUBLE* C, DOUBLE* A, DOUBLE* B) {
    int I = blockIdx.x*blockDim.x + threadIdx.x;
    int J = blockIdx.y*blockDim.y + threadIdx.y;
    int N = blockDim.y*gridDim.y; // Assume a square matrix
    if ((I < N) && (J < N)){
        DOUBLE _c = 0;
        for (unsigned int k = 0; k < N; k++) {
            DOUBLE a = A[I * N + k];
            DOUBLE b = B[k * N + J];
            _c += a * b;
        }
        C[I * N + J] = _c;
    }
}
```

```
for (unsigned int i = 0; i < N; i++)
    for (unsigned int j = 0; j < N; j++) {
        DOUBLE sum = 0;
        for (unsigned int k = 0; k < N; k++)
            sum += A[i * N + k] * B[k * N + j];
        C[i * N + j] = (DOUBLE) sum;
    }
```

# Recall Blocked Matrix Multiplication

**N blocks,  $n \times n$  global matrix,  $b = n/N$**

for  $i = 0$  to  $N-1$

for  $j = 0$  to  $N-1$

// load each block  $C[i,j]$  into cache, once :

$n^2$

//  $b = n/N =$  block size

for  $k = 0$  to  $N-1$

// load each block  $A[i,k]$  and  $B[k,j]$   $N^3$  times

//  $= 2N^3 \times (n/N)^2 =$

$2Nn^2$

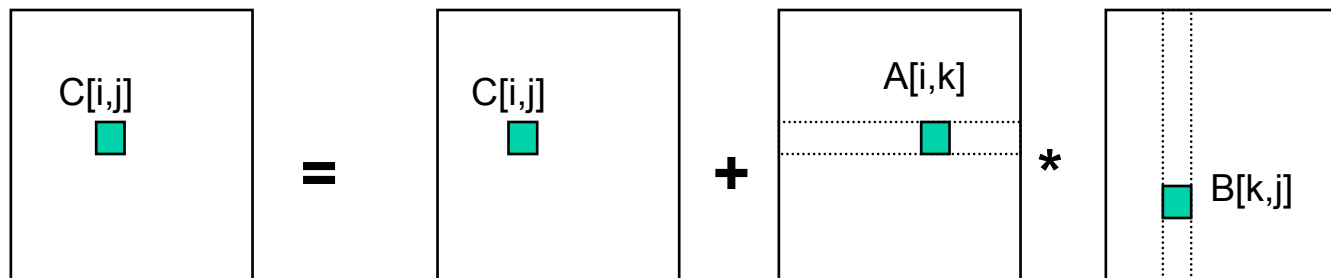
$C[i,j] += A[i,k] * B[k,j]$  // do the matrix multiply

// write each block  $C[i,j]$  once :

$n^2$

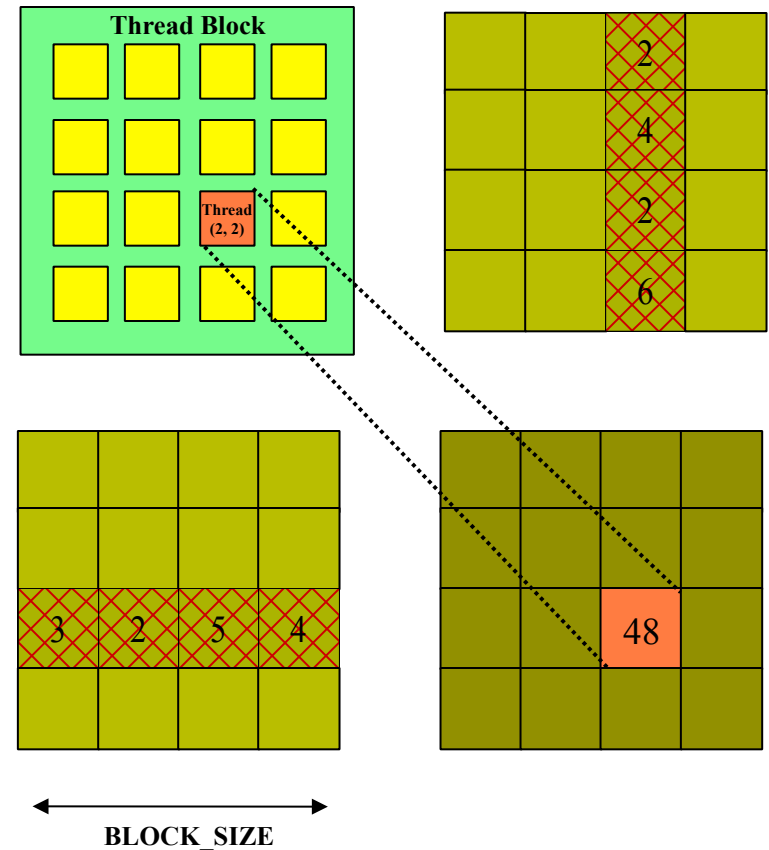
Total:

$(2*N+2)*n^2$



# Improving locality in matrix multiply

- Naïve algorithm
  - ♦ Each thread loads all the data it needs, independently loads a row and column of input
  - ♦ Each input element loaded multiple times
  - ♦ Each thread computes  $1 \text{ MAD} + 2 \text{ loads} + 1 \text{ store}$
- Blocked algorithm with shared memory
  - ♦ Threads cooperate to load a block of A&B into on-chip shared memory
  - ♦ Each thread in the block performs the *ijk* loop within shared memory
  - ♦ Each thread:  $b \text{ mpy-adds} + 1 \text{ load} + 1 \text{ store}$



# Using shared memory (uncoalesced global)

```
__global__ void matMul( float* C, float* A, float* B, int N) {  
    const unsigned int bx = BLOCK_X, by = BLOCK_Y;  
    const unsigned int tx = threadIdx.x, ty = threadIdx.y;  
    const unsigned int I = blockIdx.x*bx + tx, J = blockIdx.y*by + ty;  
    const unsigned int gx = gridDim.x, gy = gridDim.y;  
    __shared__ float a[BLOCK_X][BLOCK_Y], b[BLOCK_X][BLOCK_Y];  
    if ((I < N) && (J < N)){  
        float c = 0.0f;  
        for (unsigned int k=0; k < gy; k++){  
            a[tx][ty] = A[ I*N+k*by+ty];  
            b[ty][tx] = B[J+N*(k*bx+tx)];  
            __syncthreads();      // Synchronizes all threads in a block  
            for (unsigned int kk=0; kk< bx; kk++)  
                c += a[kk][tx]*b[kk][ty];  
            __syncthreads();      // Avoids memory hazards  
        }  
        C[I*N+J] = c;  
    }  
}
```

## Results – shared memory

- N=512, double precision
- **Dirac (C2050)**

| Geometry    | $16 \times 16$ | $8 \times 8$ | $4 \times 4$ |
|-------------|----------------|--------------|--------------|
| Uncoalesced | 40             | 39           | 14           |
| Coalesced   | 93             | 66           | 14           |

Global memory variant

|                          |                |                |                |
|--------------------------|----------------|----------------|----------------|
| <b>Dirac (prefer SM)</b> | <b>57</b>      | <b>50</b>      | <b>33</b>      |
| <b>(prefer L1\$)</b>     | <b>64</b>      | <b>58</b>      | <b>42</b>      |
| Geometries               | $1 \times 512$ | $2 \times 256$ | $4 \times 128$ |

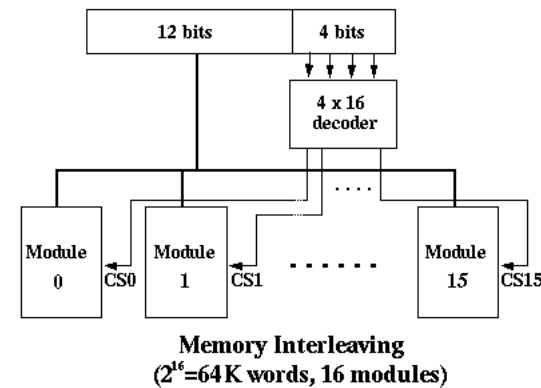
# Today's lecture

- Occupancy
- Improving Matrix Multiplication performance with shared memory
- **Memory coalescing**
- Avoiding bank conflicts

# Memory interleaving

- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?

|    |    |    |    |
|----|----|----|----|
| 0  | 1  | 2  | 3  |
| 4  | 5  | 6  | 7  |
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |



Ruye Wang,  
[fourier.eng.hmc.edu/e85/lectures/memory](http://fourier.eng.hmc.edu/e85/lectures/memory)

# Global memory coalescing

- Global memory accesses in units of 32, 64, 128 B
- Does not require sequential accesses by threads—only need to fall within the same 128B segment
- Accesses organized by warp
- Concurrent accesses by a warp's reads coalesce into  $K$  transactions = # different cache lines (128B) covered by the accesses
- Access not cached in L1: 32B at a time (x4) in L2

`shrd[threadIdx.x] = gbl[blockIdx.x*blockDim.x+threadIdx.x]` **YES!**  
`shrd[threadIdx.x] = gbl[blockDim.x+ blockIdx.x*threadIdx.x]` **NOT!**

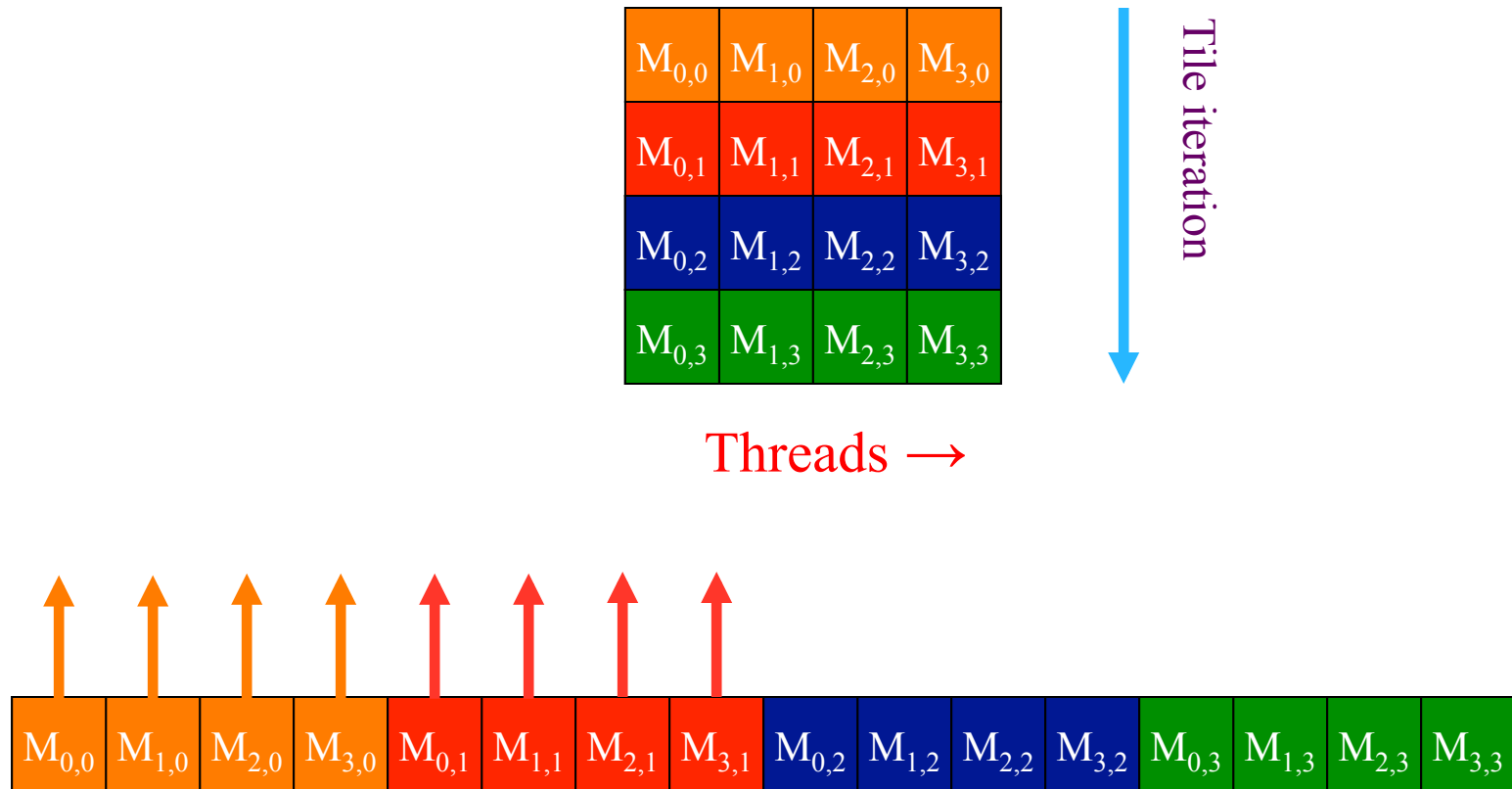


# Global Memory

- If accessed word  $> 4$  bytes, warp's memory request split into separate, independently issued 128-byte memory requests
- Non-atomic, concurrent writes within a warp: writer not defined
- `cudaMalloc()` is guaranteed to be aligned to at least 256 bytes

# Memory coalescing

- Simplest: addresses fall within the same 128B segment
- Accesses organized by half warps (16 threads)



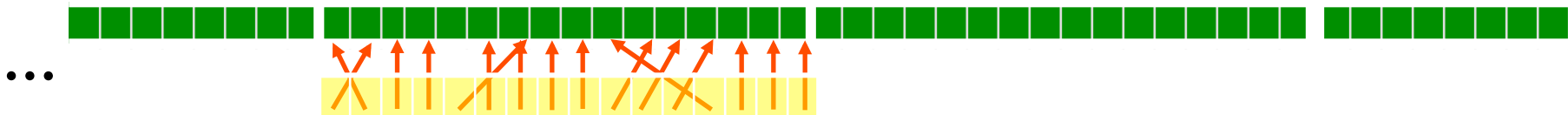
Nvidia Corp.

## Memory coalescing (compute capability $\geq 1.2$ )

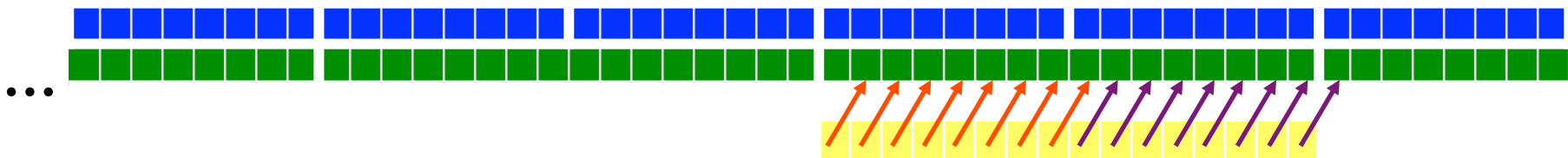
- Find the segment containing the address request of the lowest numbered active thread
- Find all other active threads requesting in same segment
- Reduce transaction size (if possible)
- Mark the serviced threads as inactive
- Repeat until all threads in the warp are complete
- Handles permutations, too

### 1 transaction – 128B segment

128-byte segments



### 2 transactions – 2 x 128B segments



Nvidia Cuda C Programming Guide: Appendix G.4.2

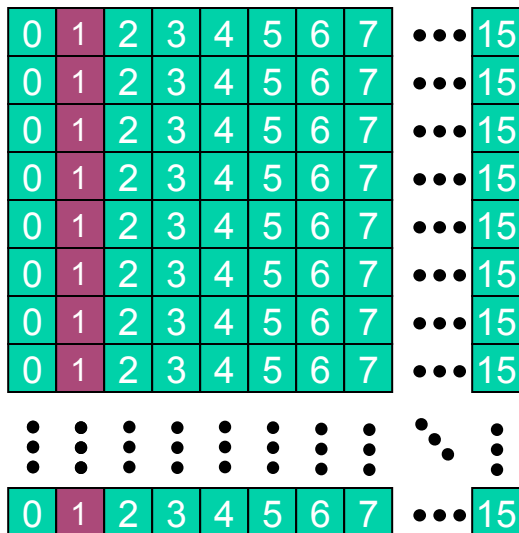
# Coalescing with 2d arrays

- All warps in a block access consecutive elements within a row as they step through neighboring columns

```

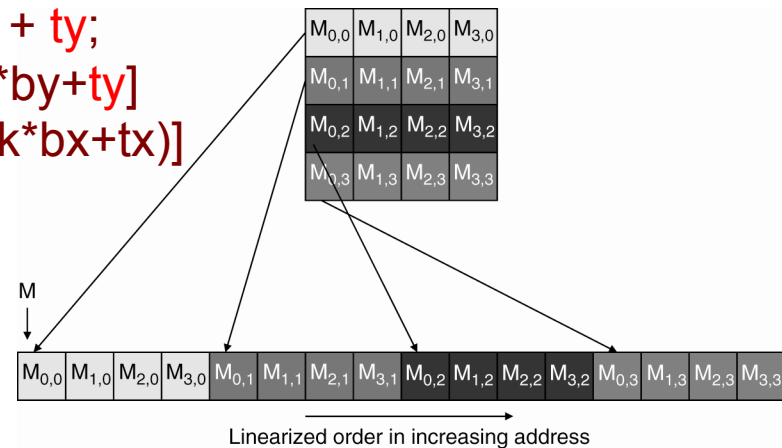
I = blockIdx.y*by + ty;
J = blockIdx.x*bx + tx;
int tx = threadIdx.x
a[ty][tx] = A[I*N+k*by+tx]
b[ty][tx] = B[J+N*(k*bx+ty)]
    
```

- Accesses by threads in a block along a column don't coalesce



```

I = blockIdx.x*bx + tx;
J = blockIdx.y*by + ty;
a[tx][ty] = A[I*N+k*by+ty]
b[ty][tx] = B[J+N*(k*bx+tx)]
    
```



# Coalesced access improve performance

```
I = blockIdx.y*by + ty;  
J = blockIdx.x*bx + tx;
```

Slow:

```
I = blockIdx.x*bx + tx;  
J = blockIdx.y*by + ty;
```

```
__shared__ float  a[BLK][BLK], b[BLK][BLK];  
if ((I < N) && (J < N)){  
    float c = 0.0f;  
    for (k=0; k < gy; k++){  
        a[ty][tx] = A[I*N+k*by+tx];  
        b[ty][tx] = B[J+N*(k*bx+ty)];  
        __syncthreads();  
        for (kk=0; kk< bx; kk++){  
            c += a[ty][kk]*b[kk][tx];  
        }  
        __syncthreads();  
    }  
    C[I*N+J] = c;
```

Slow:

```
a[tx][ty] = A[I*N+k*by+ty];  
b[ty][tx] = B[J+N*(k*bx+tx)];
```

**Fin**