CMPSCI 691AD – General Purpose Computation on the GPU

Spring 2009

Lecture 9: CUDA Memory Models II

Rui Wang

Typical Data Movement

Host Memory

Device (global) Memory

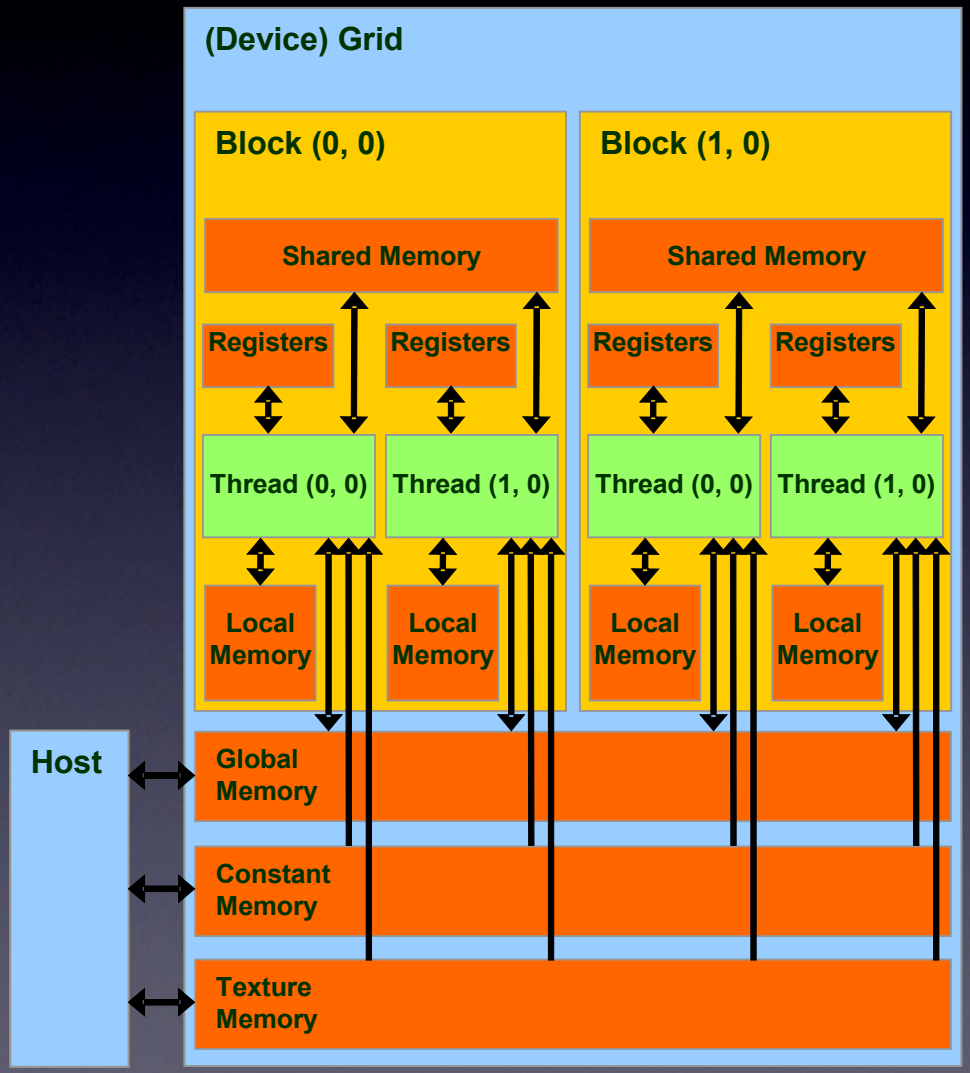
[Shared Memory]

Computation

[Shared Memory]

Device (global) Memory

Host Memory



Global Memory Access

- Global memory not cached, so important to follow the right access pattern to get maximum bandwidth.
- Device is capable reading 4, 8, 16-Byte word from global memory in a single instruction.
  - Data address should be properly aligned to 4, 8, or 16
  - Alignement guaranteed for built-in types (float2 etc.)
- For structures, can force alignment using __align__

```
struct __align__(16)
{   float a;
    float b;
    float c;   };
```

Coalescing of Global Memory Access

- A coordinated access by a warp (16 threads) to a contiguous region of global memory, resulting in a single memory transaction:
  - 64 bytes: each thread accesses 4-byte word (int, float...)
  - 128 bytes: each thread accesses 8-byte word (int2,float2...)
  - 2x128 bytes: each thread access 16-byte word (int4, float4...)

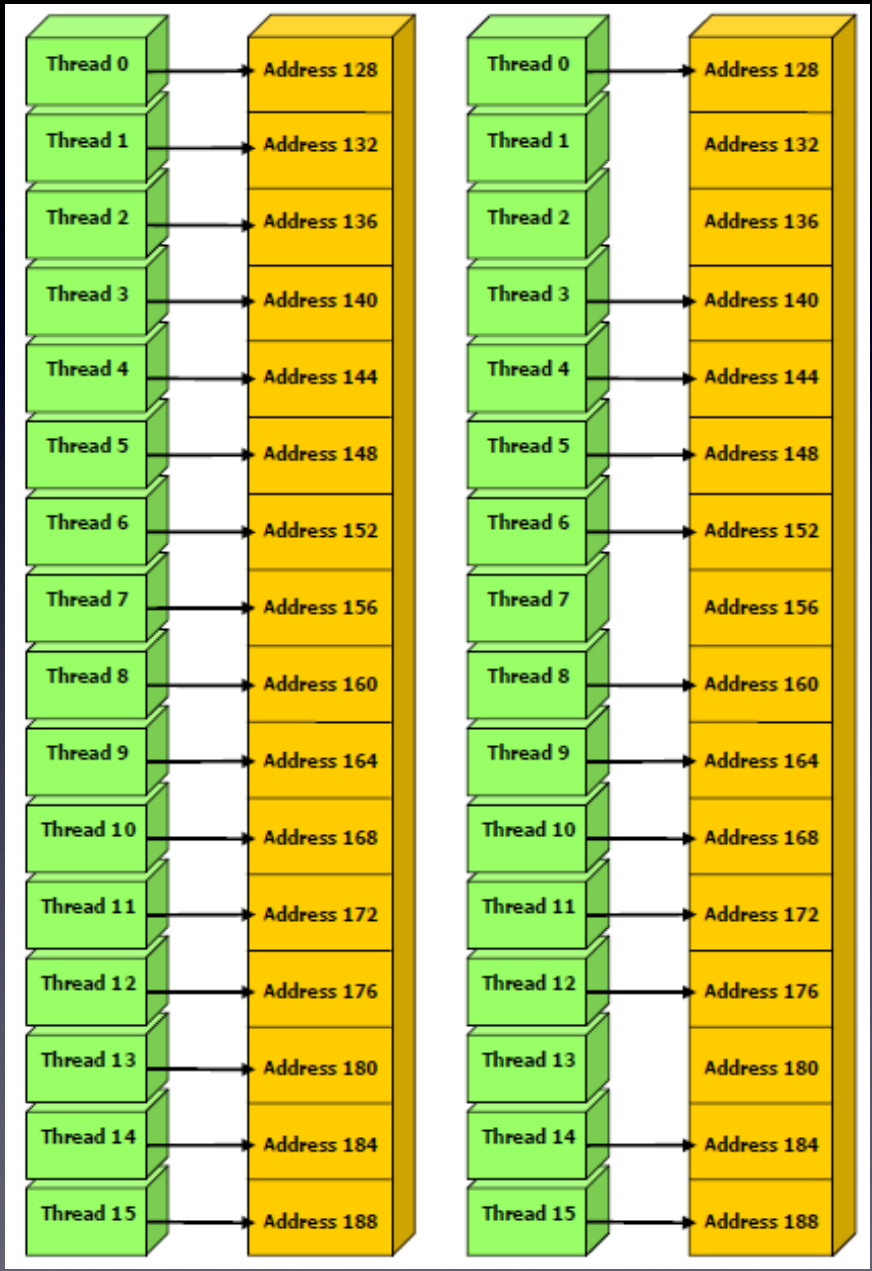
Coalescing of Global Memory Access

- All 16 accesses in the warp must lie in the same segment of size equal to the above transaction sizes; otherwise additional memory transactions will be issued.
- The data being accessed must be properly aligned to the corresponding transaction size.
- Not all threads must participate.

Coalescing of Global Memory Access

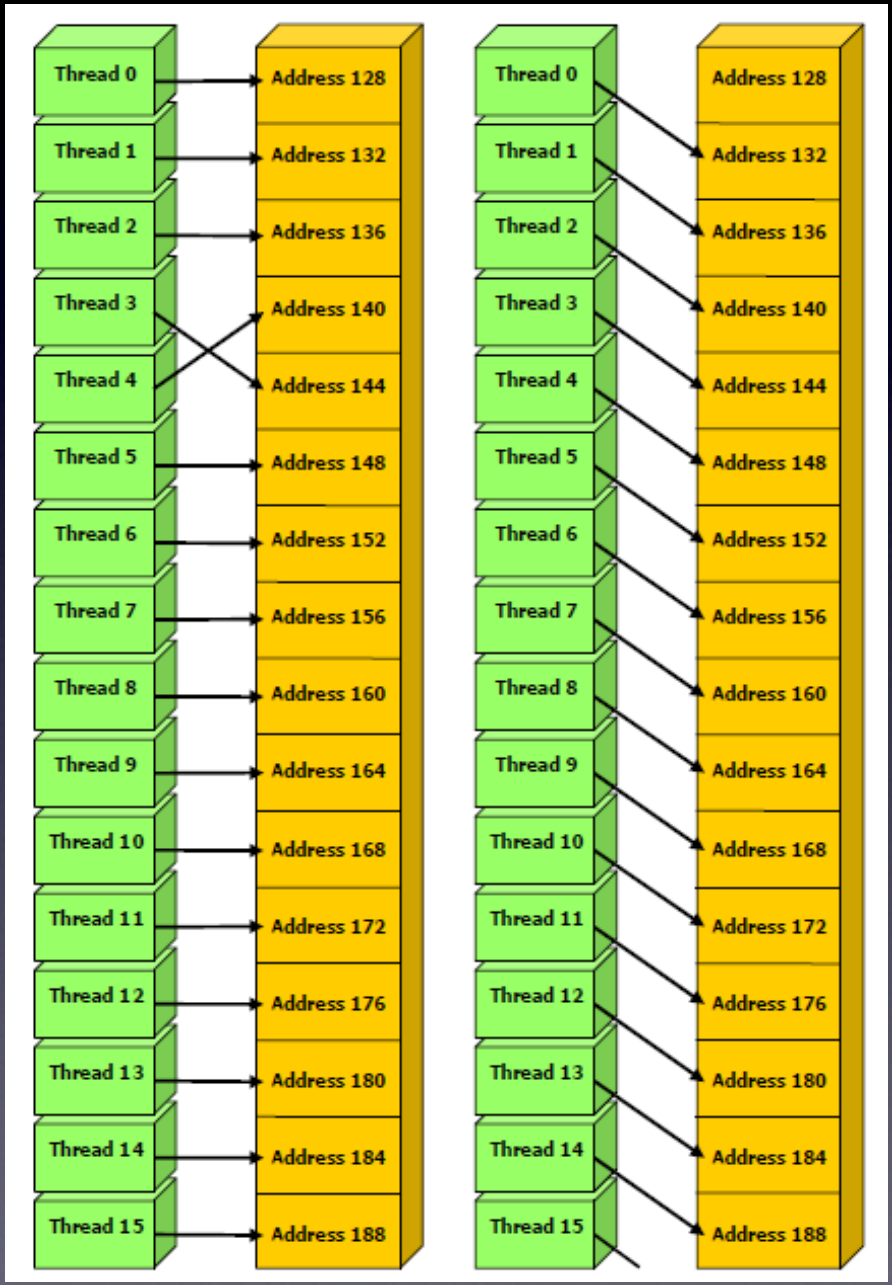
- Additional constraints for CC1.0/1.1 (G80/90)
  - Threads must access the words in sequence, i.e. thrtead k in the half-warp must access word k
  - If coalescing is not fulfilled, a separate memory transaction is issued for every thread
- Delivering one coalesced 32B is slightly faster than 64B, and a lot faster than 128B; and an order of magnitude faster than non-coalesced.

Coalesced Access: reading floats



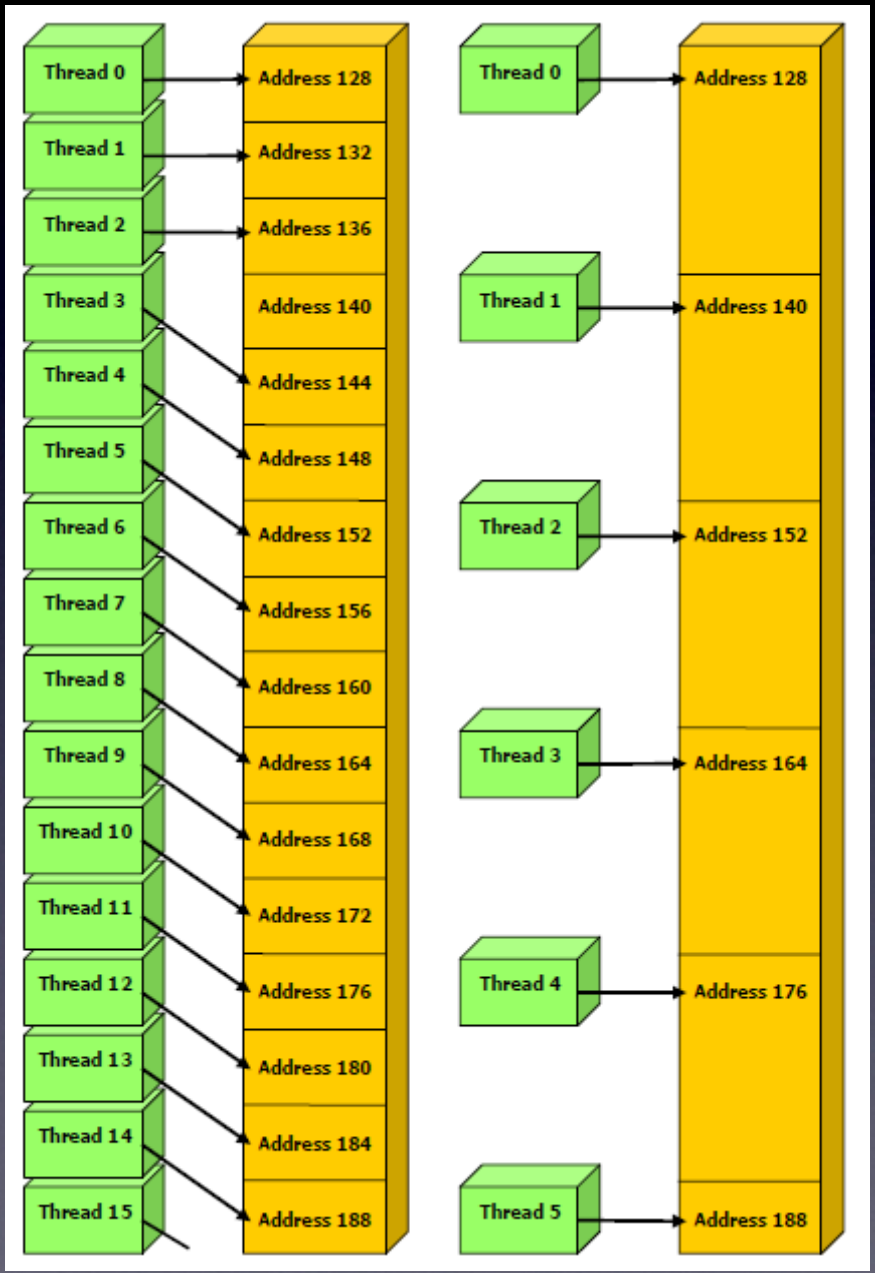
Coalesced, resulting in
1 memory transaction

Non-Coalesced Access (CC1.0/1.1)



Permutated access, or misaligned starting address: non-coalesced, resulting in 16 transactions.

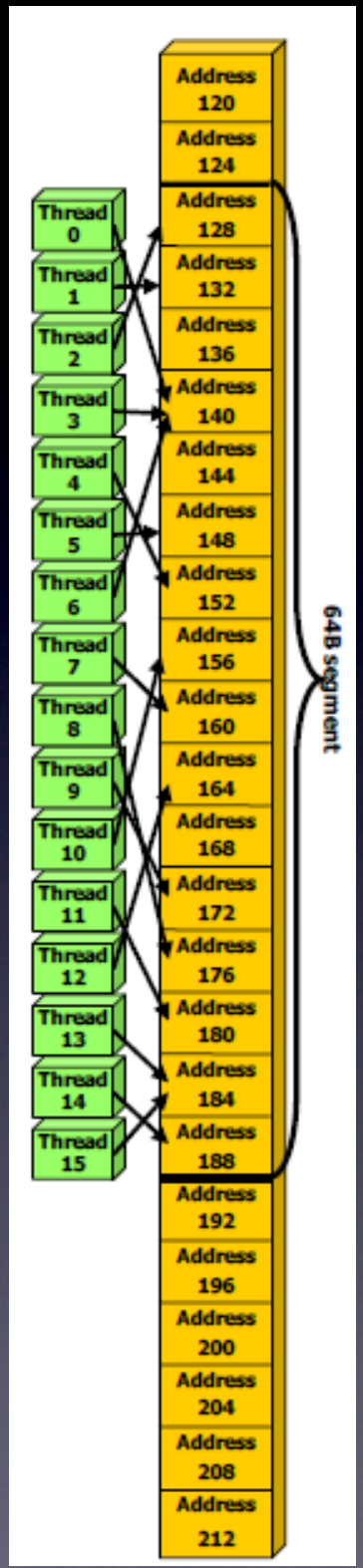
Non-Coalesced Access (CC1.0/1.1)



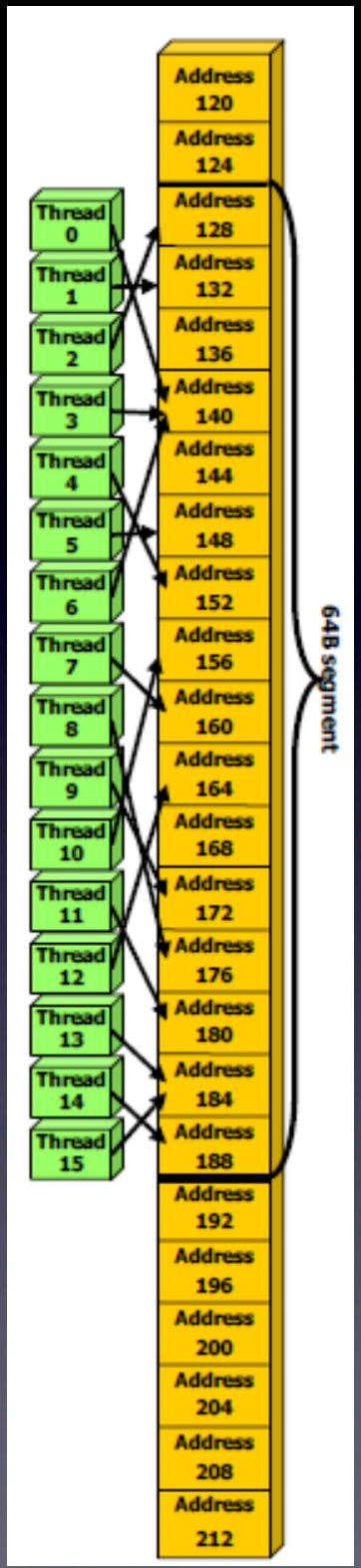
misaligned access,
or non-contiguous read:
non-coalesced, resulting
in 16 transactions.

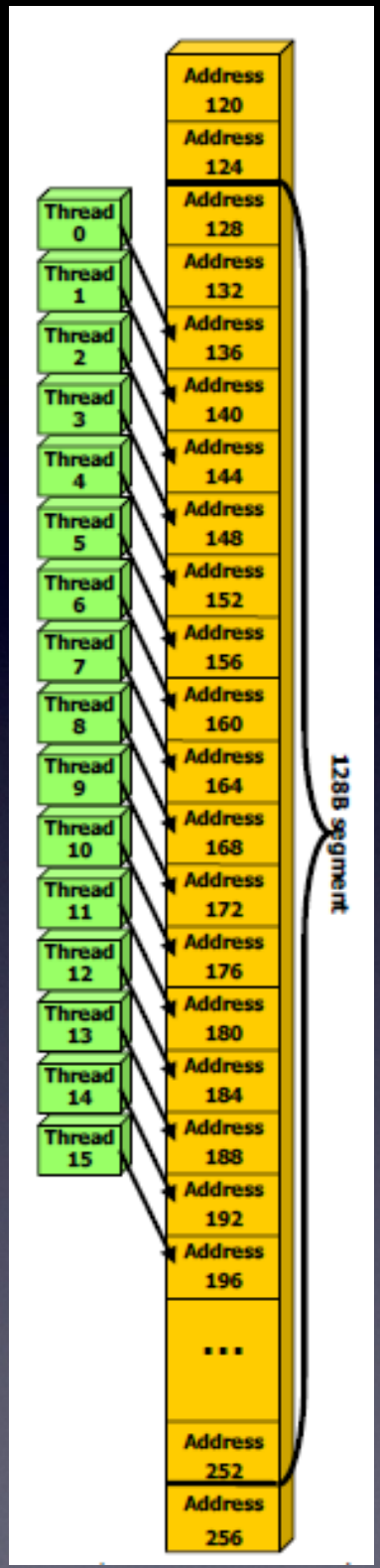
Coalescing of Global Memory Access

- For CC1.2 or higher (GT200):
  - Coalescing achieved for any pattern of access as long as all accesses are within the same transaction size. (vs. 1.1, where access must be sequential)
  - K memory transactions are issued for k segments (vs. 1.1, where 16 transactions are issued if k>1)
  - Also, device tries to minimize the transaction size if possible.

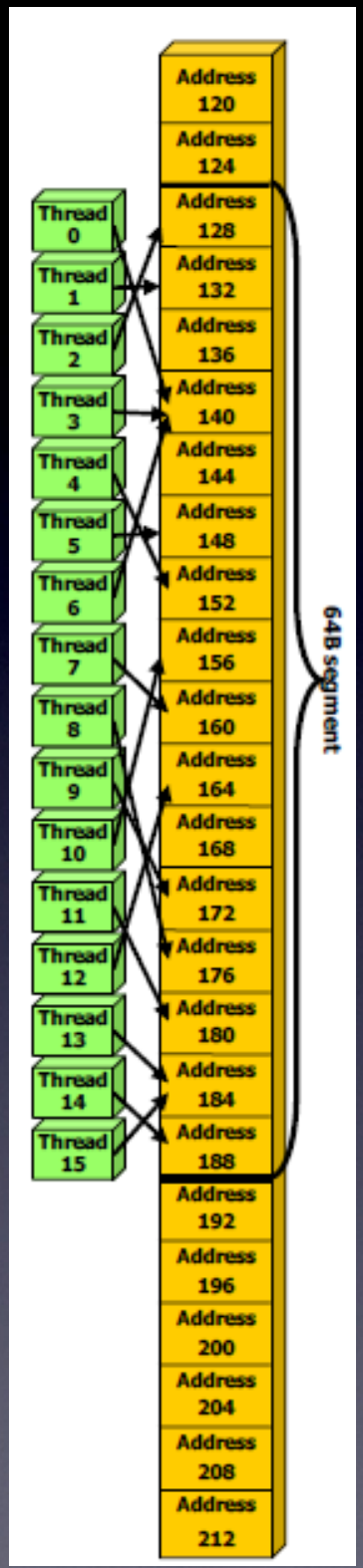


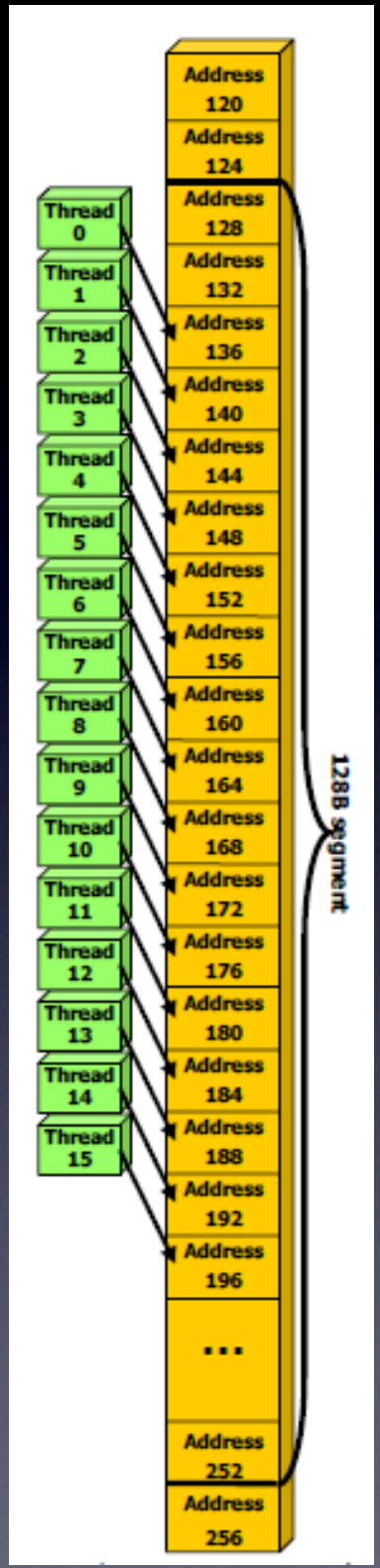
CC1.2/1.3:
coalesced into one
64B transaction.

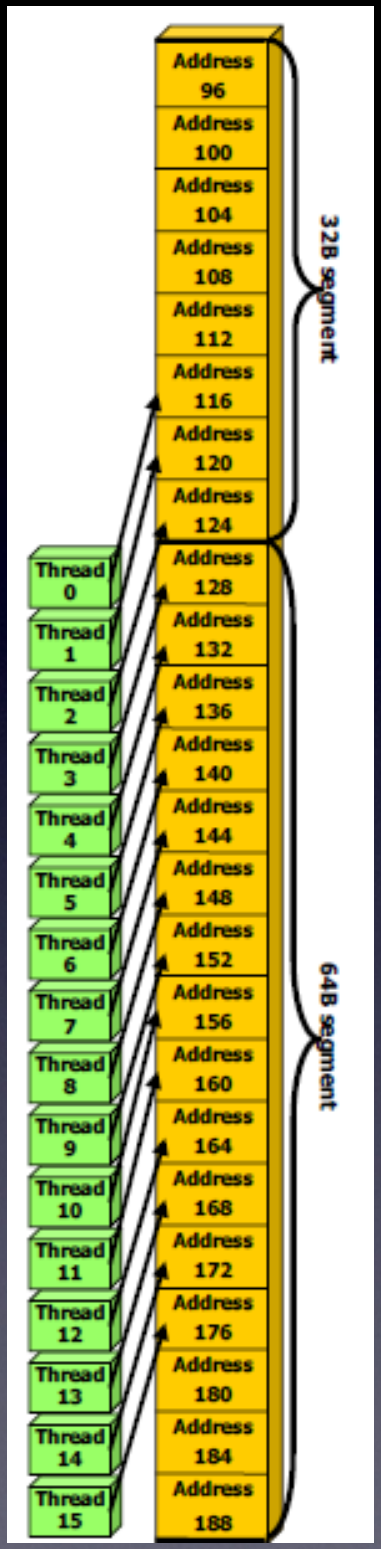




CC1.2/1.3:
coalesced into one
128B transaction.







CC1.2/1.3:
coalesced into two
transactions: one
32B + one 64B

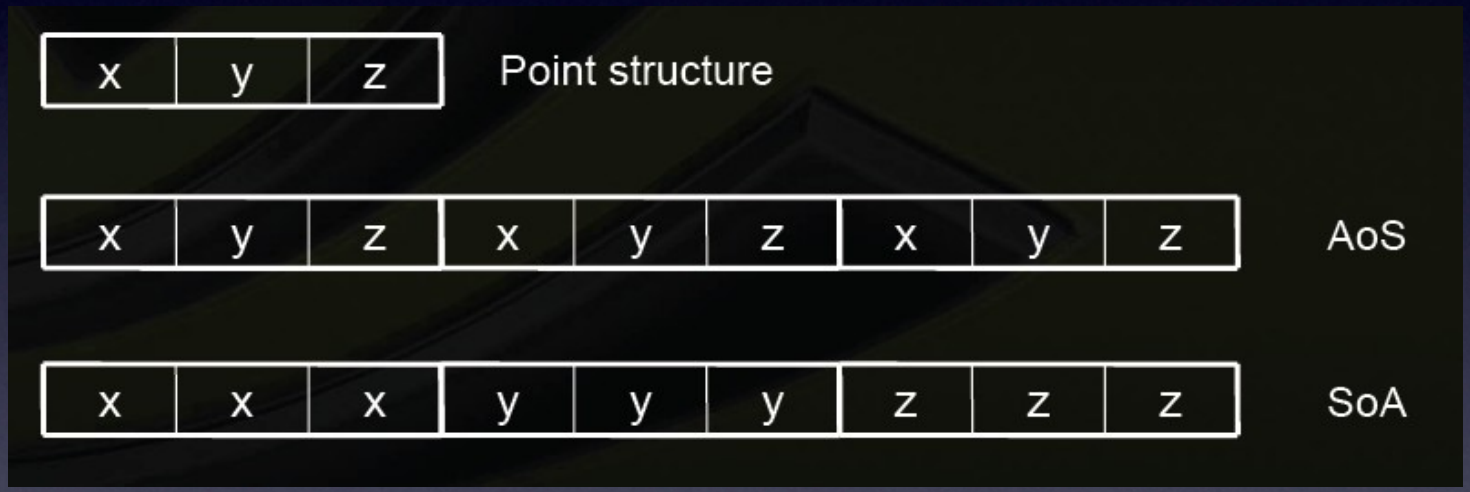
Coalescing: timing results

- Experiments on G80:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
- 12K blocks x 256 threads / block:
  - 356 us → coalesced
  - 3494 us → permutated/misaligned thread access

http://dl.getdropbox.com/u/484203/Lectures/NicolasPinto/

Coalescing: Structures

- For structures that are not of size 4, 8, or 16 bytes:
  - Use a Structure of Arrays (SoA) if possible



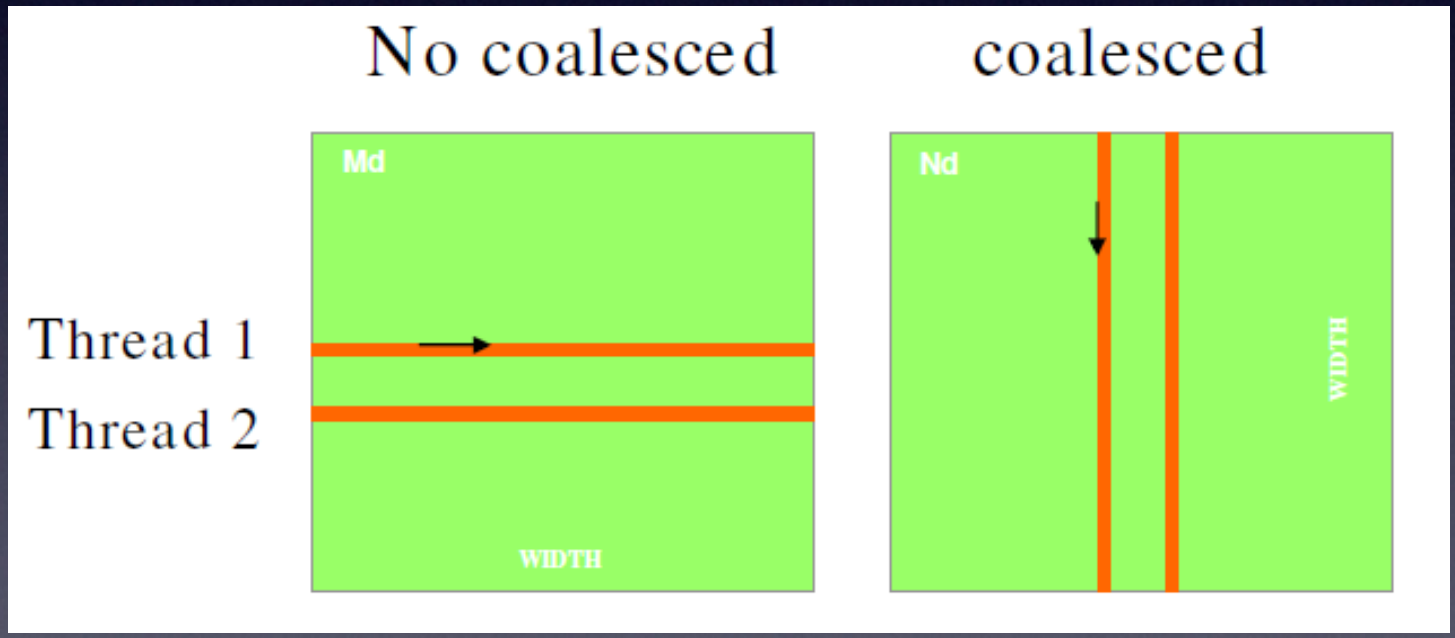
  - If SoA not possible, force structure alignment
  - Use shared memory to achieve coalescing.

Local Memory

- Stored in device memory space, not cached, accesses are as expensive as to global memory
- Local memory accesses are always coalesced because they are per-thread by definition.

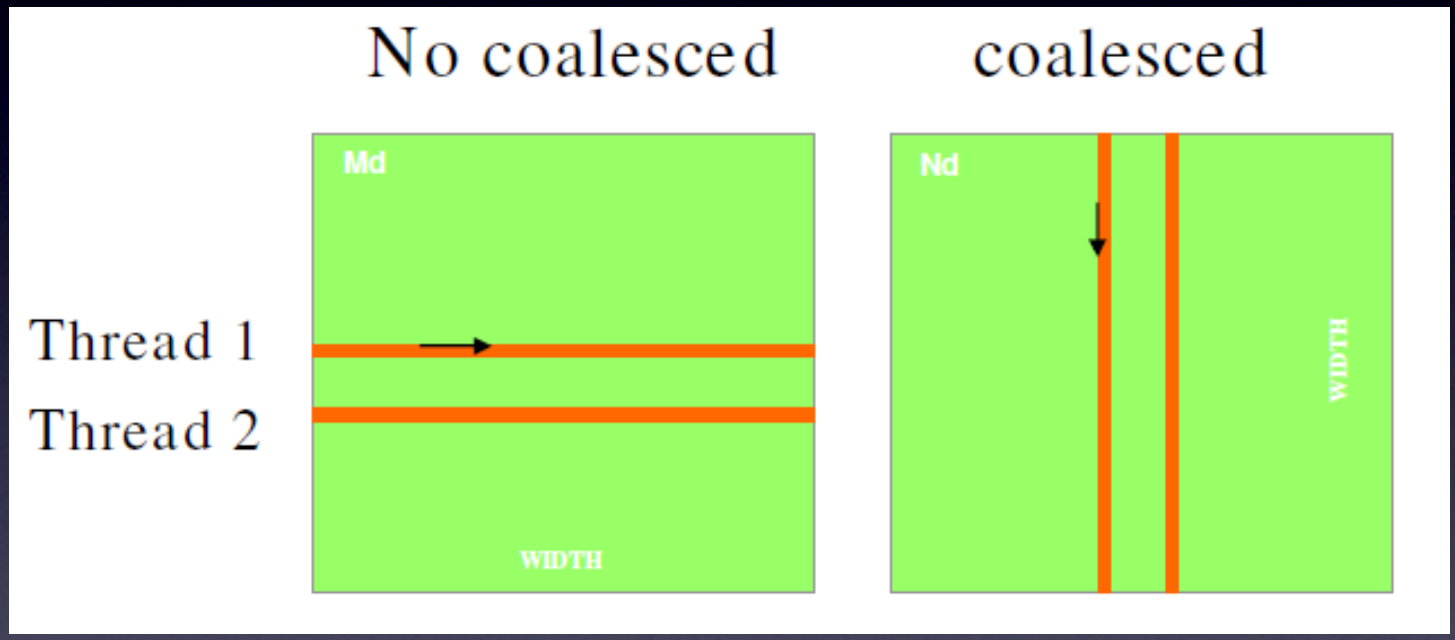
Examining Global Memory Coalescing

- Accessing a 2D matrix:
  - Device organizes threads in a 2D block into warps in row-major order, similar to the layout of a 2D matrix in memory.



Examining Global Memory Coalescing

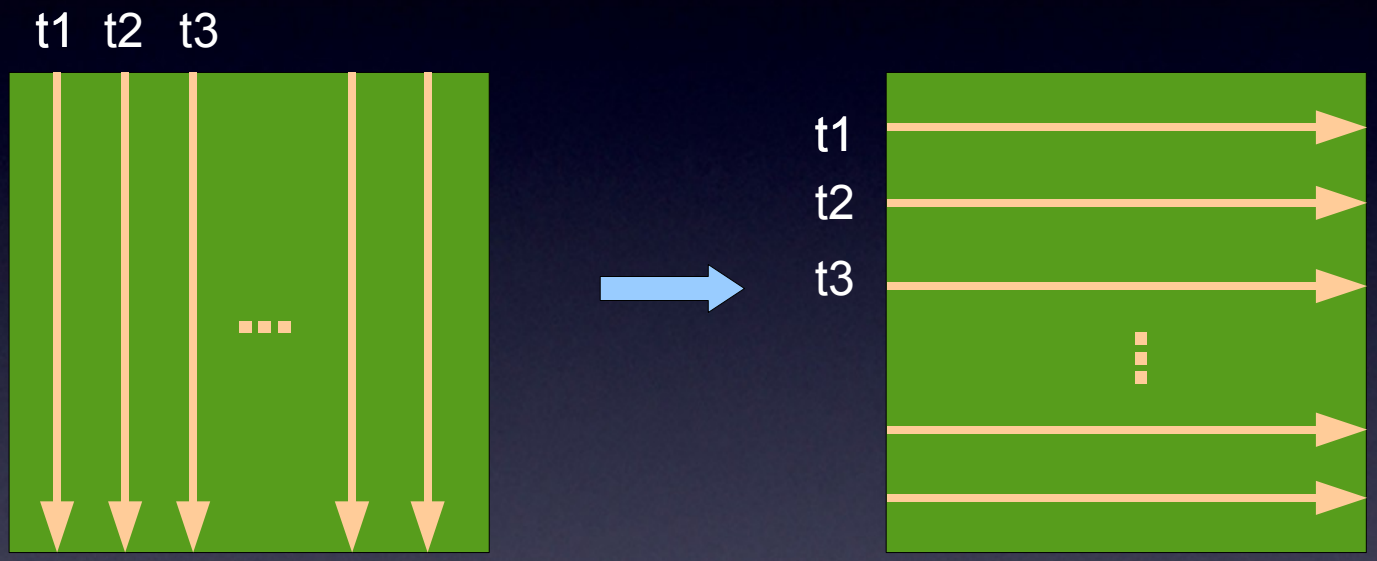
- Accessing a 2D matrix:



- One solution: use shared memory to achieve memory coalescing.

Examining Global Memory Coalescing

- Use Shared Memory to Achieve GMEM Coalescing:



Step 1: Load data into shared memory

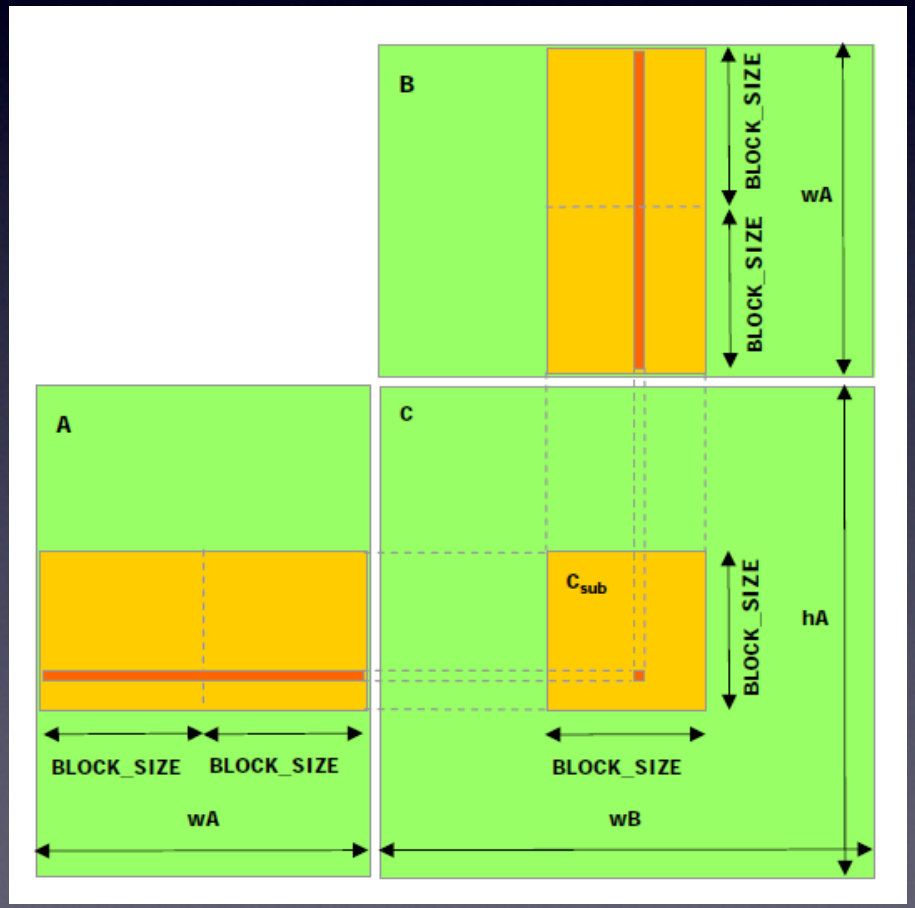
Step 2: Read data from SMEM and compute in the order desired

Coalesced GMEM access

SMEM access, coalescing not required

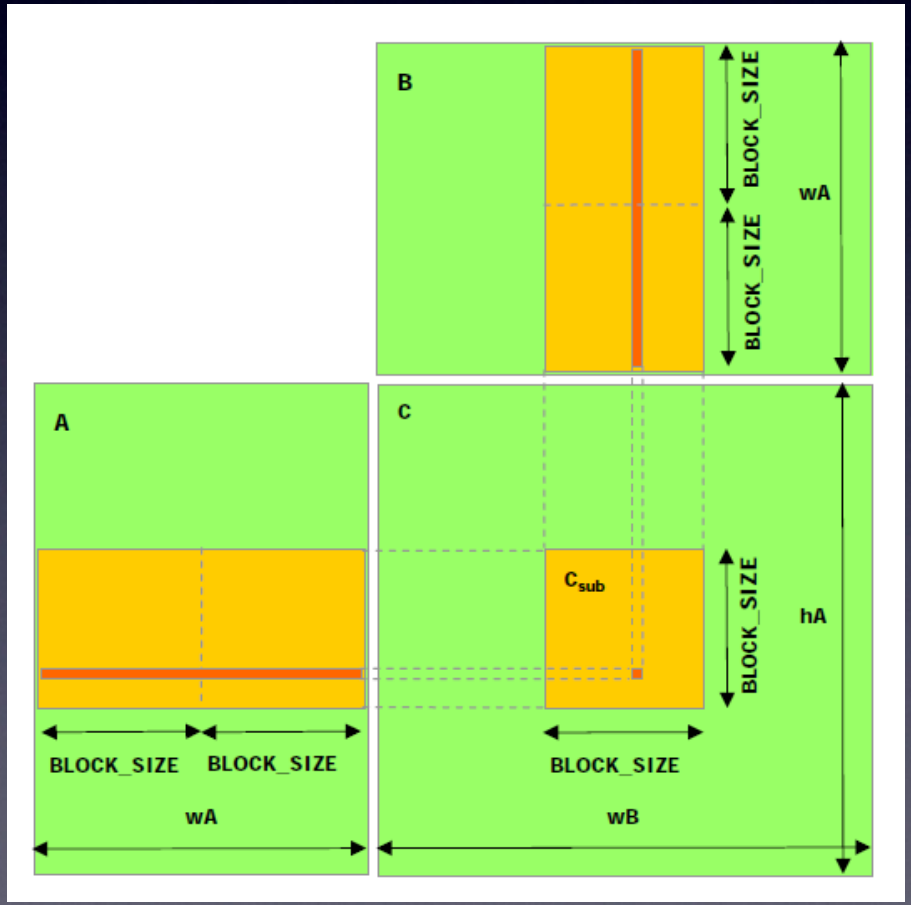
Examining Global Memory Coalescing

- Recall the matrix multiplication we discussed last time:



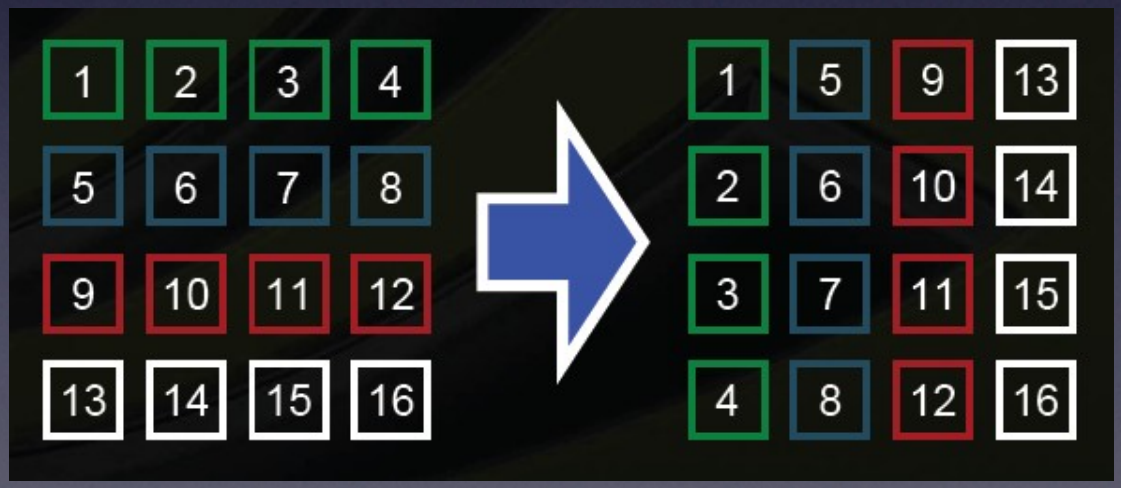
Examining Global Memory Coalescing

- Recall the matrix multiplication we discussed last time:
  - Memory read coalesced into a single transaction as long as the width of each submatrix is a multiple of 16
  - If not, then multiple transactions are needed.
    - e.g., 8x8 block



Examining Global Memory Coalescing

- Therefore, the use of shared memory can serve two purposes: 1) reduce the amount of GMEM accesses; 2) achieve GMEM coalescing even if the natural order of the computation does not.
- Another example: Matrix Transpose



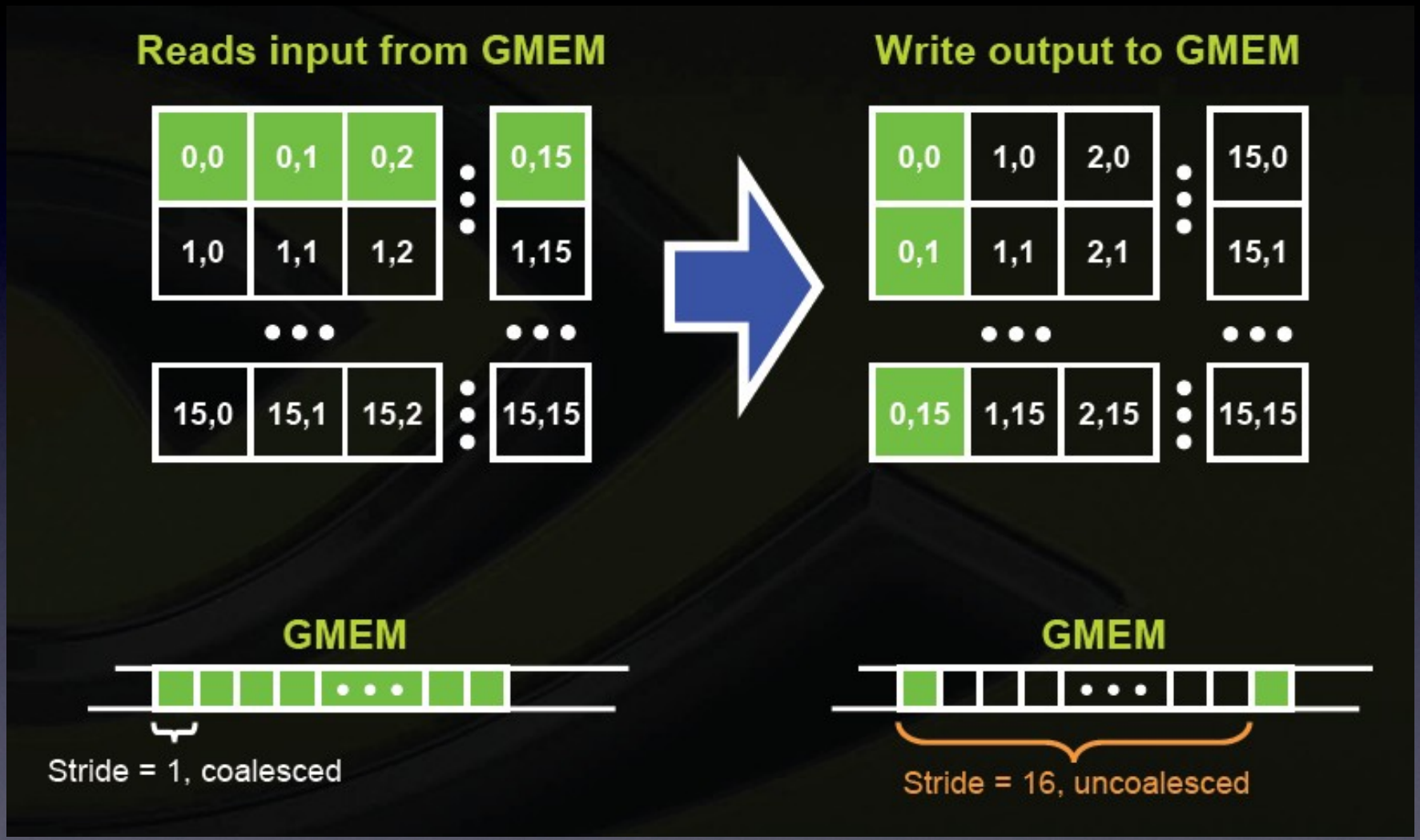
Uncoalesced Matrix Transpose

```
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
1.  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
2.  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

3.  if (xIndex < width && yIndex < height)
    {
4.      unsigned int index_in  = xIndex + width * yIndex;
5.      unsigned int index_out = yIndex + height * xIndex;
6.      odata[index_out] = idata[index_in];
    }
}
```

http://dl.getdropbox.com/u/484203/Lectures/NicolasPinto/

Uncoalesced Matrix Transpose

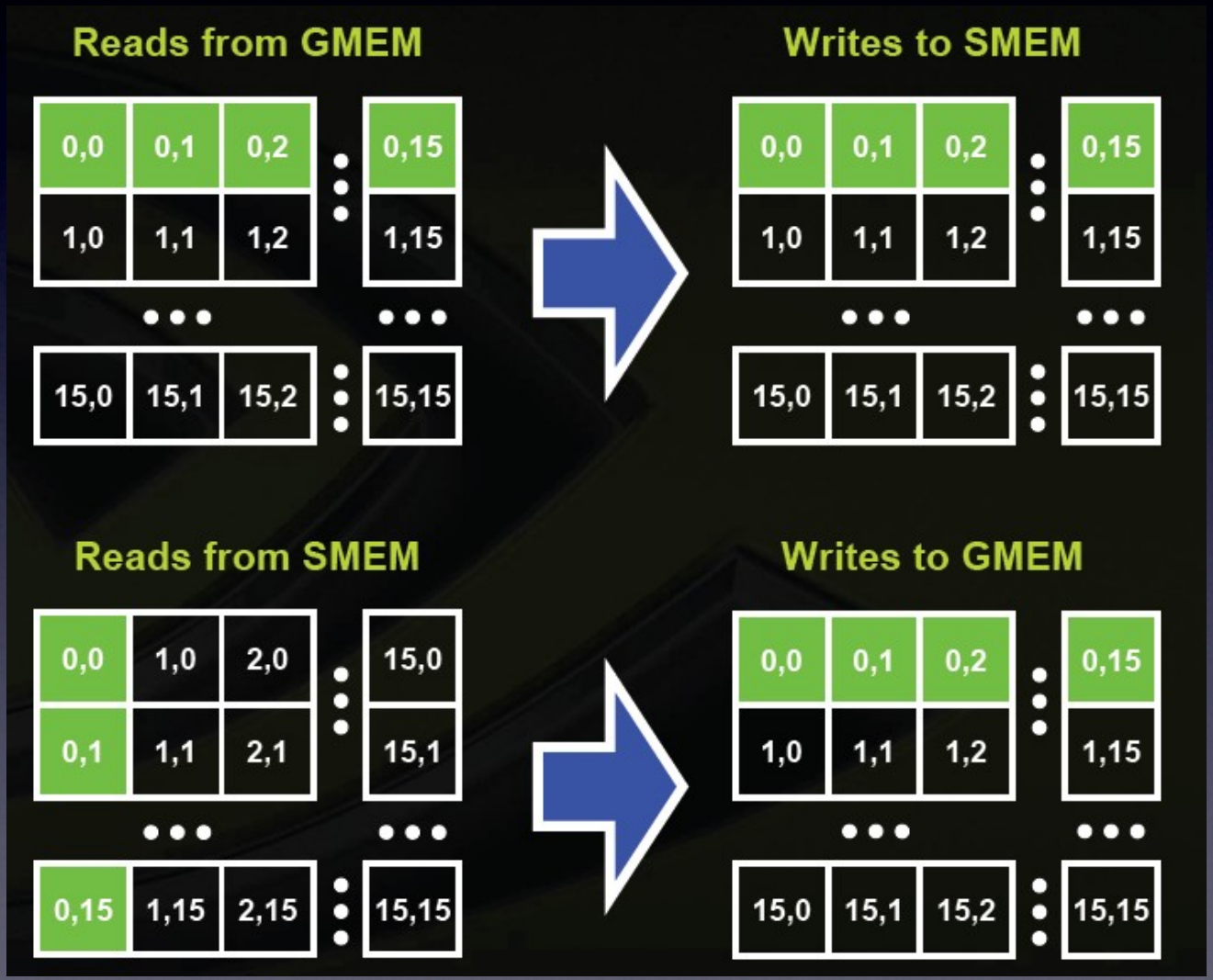


http://dl.getdropbox.com/u/484203/Lectures/NicolasPinto/

Coalesced Matrix Transpose

- Idea: partition matrix into square blocks (16x16)
- Threads in each block coorperate to read data from GMEM (coalesced); store in SMEM; then read from SMEM in different order and write to GMEM (coalesced).

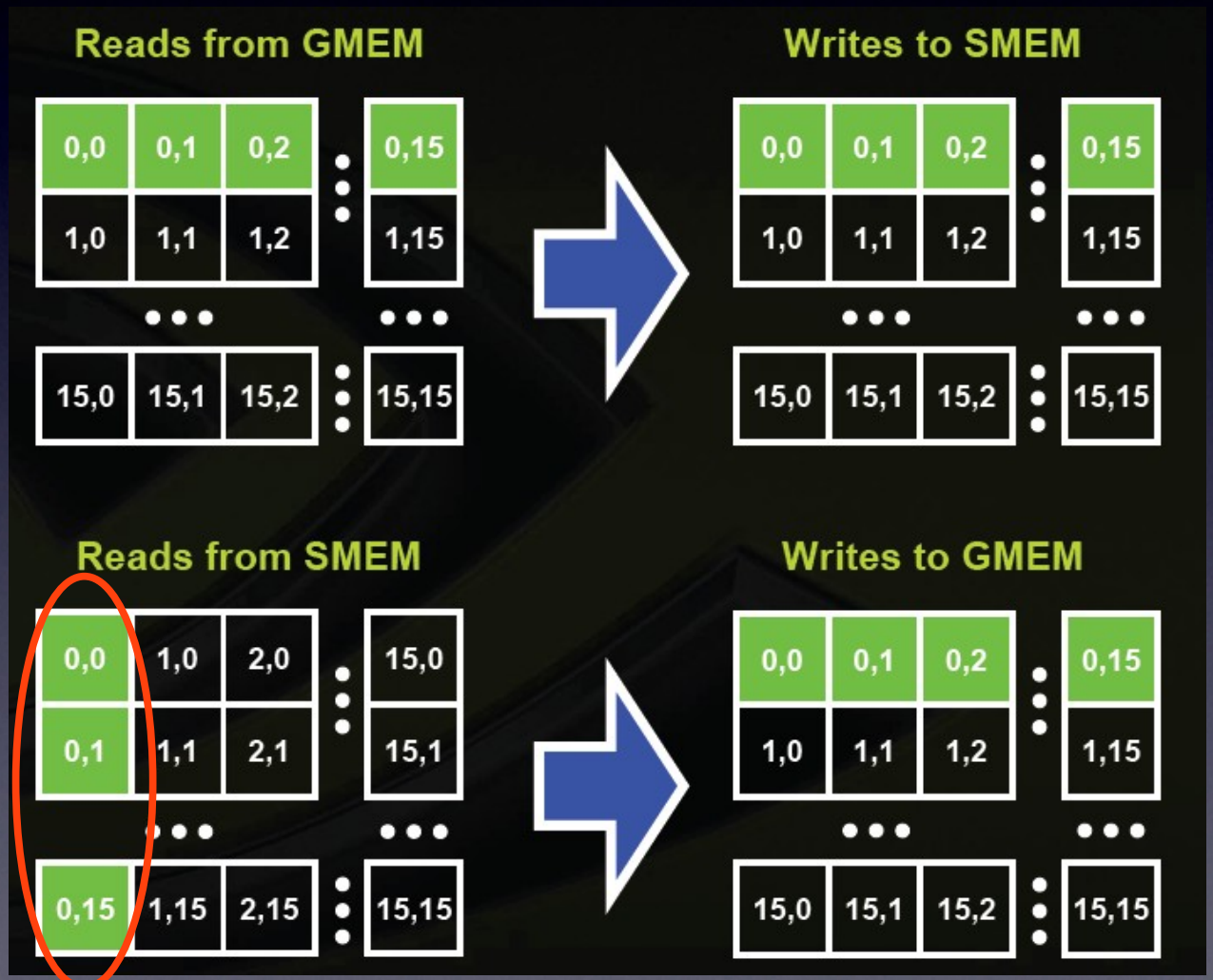
Coalesced Matrix Transpose



http://dl.getdropbox.com/u/484203/Lectures/NicolasPinto/

Solving SMEM Bank Conflicts

- But now we have SMEM bank conflicts!!



http://dl.getdropbox.com/u/484203/Lectures/NicolasPinto/

Solving SMEM Bank Conflicts

- Problem: read stride = 16
- Solution: Allocate one extra column (padding) to force elements in the same row stored in different banks

http://dl.getdropbox.com/u/484203/Lectures/NicolasPinto/

Coalesced Matrix Transpose

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
1.  __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];

2.  unsigned int xBlock = blockDim.x * blockIdx.x;
3.  unsigned int yBlock = blockDim.y * blockIdx.y;
4.  unsigned int xIndex = xBlock + threadIdx.x;
5.  unsigned int yIndex = yBlock + threadIdx.y;
6.  unsigned int index_out, index_transpose;

7.  if (xIndex < width && yIndex < height)
    {
8.      unsigned int index_in = width * yIndex + xIndex;
9.      unsigned int index_block = threadIdx.y * (BLOCK_DIM+1) + threadIdx.x;
10.     block[index_block] = idata[index_in];
11.     index_transpose = threadIdx.x * (BLOCK_DIM+1) + threadIdx.y;
12.     index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
13. __syncthreads();

14. if (xIndex < width && yIndex < height)
15.     odata[index_out] = block[index_transpose];
}
```

Coalesced Matrix Transpose

- Was it worth the trouble?

Grid Size	Coalesced	Non-coalesced	Speed up
128 × 128	0.011 ms	0.022 ms	2.0×
512 × 512	0.07 ms	0.33 ms	4.5×
1024 × 1024	0.30 ms	1.92 ms	6.4×
1024 × 2048	0.79 ms	6.6 ms	8.4×

- Is matrix multiplication algorithm we discussed is also subject to SMEM bank conflicts?