Optimizing for CUDA III

Shared Memory, DRAM Organization and Data Layout

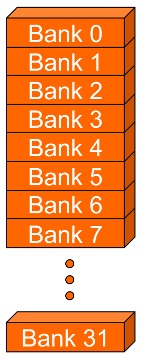
Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green, Mark Haris and many others
Real World Techonologies by David Kanter

1

Shared memory (SMEM)

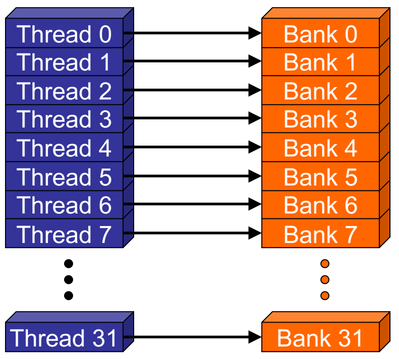
- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



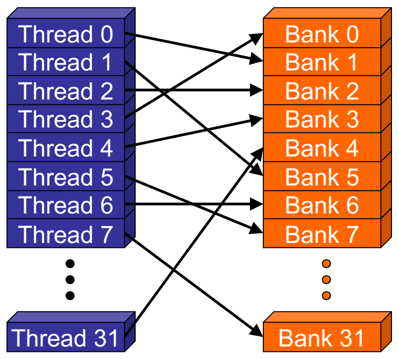
2

Bank addressing examples

No Bank Conflicts
Linear addressing, stride == 1
e.g., `k=array[threadIdx.x]`



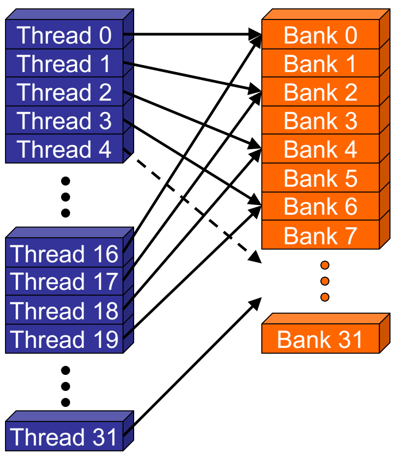
No Bank Conflicts
Random 1:1 Permutation



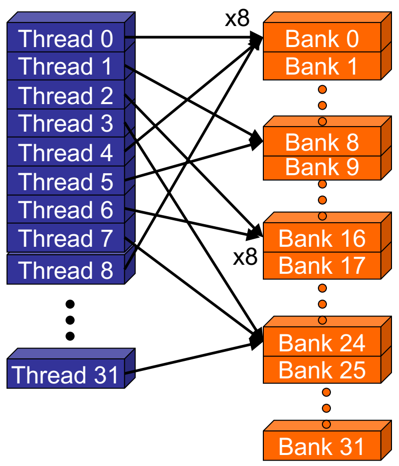
3

Bank addressing examples

2-way Bank Conflicts
Linear addressing, stride == 2
e.g., `k=array[threadIdx.x*2]`



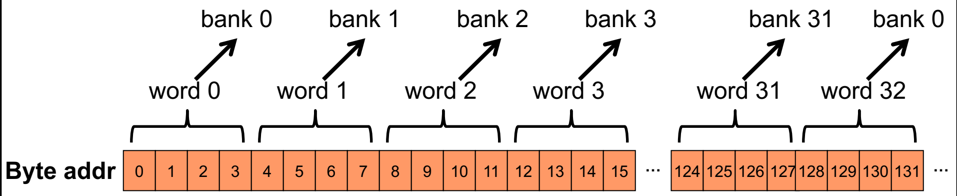
8-way Bank Conflicts
Linear addressing, stride == 8
e.g., `k=array[threadIdx.x*8]`



4

Mapping addresses to SMEM banks (GTX680)

- Each bank has a bandwidth of 64 bits per clock cycle
- Successive 64-bit words are assigned to successive banks (8-byte-wide mode)
- Successive 32-bit words are assigned to successive banks (4-byte-wide mode)



- GTX680 has 32 banks
  - So (for 4-byte mode) bank = word_id % 32 = (address / 4) % 32
    - a.k.a. ***address interleaving at 4-byte granularity***
  - Same as the size of a warp
    - No bank conflicts between different warps, only within a single warp

5

Shared memory bank conflicts

- Shared memory is as fast as registers, provided that:
  1. There are no bank conflicts
  2. There is sufficient shared memory bandwidth
     - if either condition fails, shared memory is much slower than registers

– The fast case:
  – If all threads of a warp access different banks
    → there is no bank conflict
  – If all threads of a warp read an identical address
    → there is no bank conflict (broadcast)

– The slow case:
  – Bank Conflict: multiple threads in the same warp access the same bank for a different row
  – Must serialize the accesses
  – Cost = max # of simultaneous accesses to a single bank

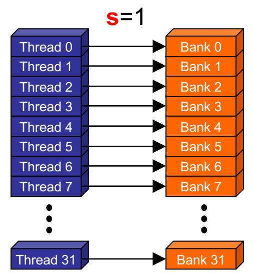
6

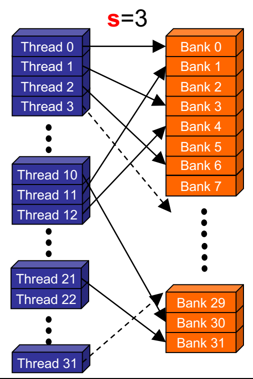
Linear addressing

- Given:

```
__shared__ float shared[256];
float foo;
foo =  shared[baseIndex + s*threadIdx.x];
```

- This is only bank-conflict-free if s shares no common factors with the number of banks
  - e.g., 32 banks on GTX680, **s = 1 or 3**





7

Data types and bank conflicts (capability >2.x)

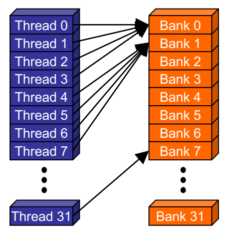
- This has no conflicts if type of shared is 32-bits (e.g., int, float):

```
foo = shared[baseIndex + threadIdx.x]
```

- If **reads** fall within same 32-bit word
  → No conflicts, broadcast !

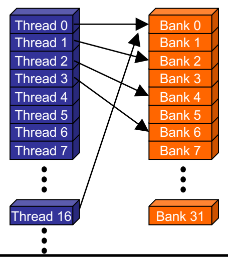
- Assume the array is 4-byte aligned:

```
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```



- 2-way conflicts for larger types:

```
__shared__ double shared[];
foo = shared[baseIndex + threadIdx.x];
```



9

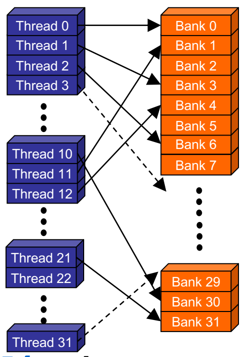
Structs and bank conflicts

- Struct assignments compile into as many memory accesses as struct members:

```
struct vector3 { float x, y, z; };
__shared__ struct vector3 v3[64];
```

- This has no bank conflicts for vector struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 32)

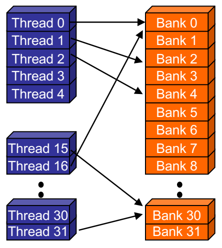
```
struct vector3 v = v3[baseIndex + threadIdx.x];
```



```
struct vector2 { float f; int c; };
__shared__ struct vector2 v2[64];
```

- This has 2-way bank conflicts for `vector2` (2 words)
  - 2 accesses per thread

```
struct vector2 v = v2[baseIndex + threadIdx.x];
```

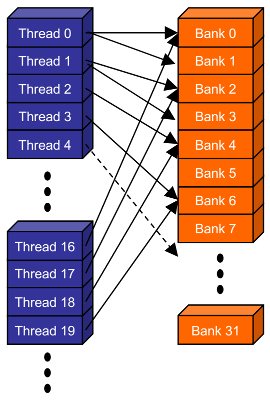


10

Common array bank conflict patterns 1D

- Each thread loads 2 elements into shared mem:
  - 2-way-interleaved loads result in 2-way bank conflicts:

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

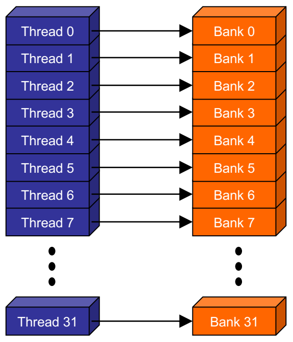


- This makes sense for traditional CPU threads
  - Locality in cache line usage and reduced sharing traffic.
  - But, not in shared memory usage
    - There are no cache line effects but banking effects

11

A better array access pattern

- Each thread loads one element in every consecutive group of blockDim elements.

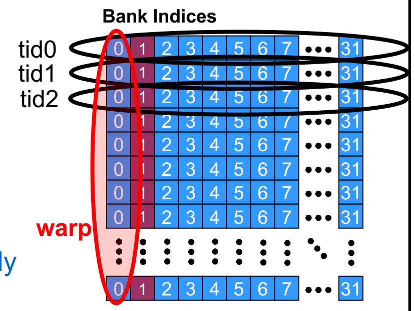


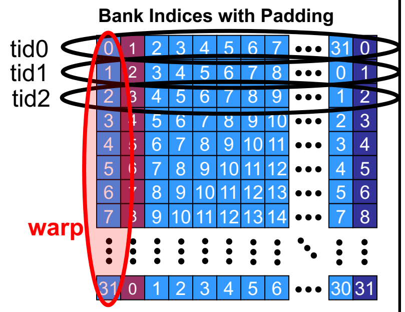
```
shared[tid] = global[tid];
shared[tid + blockDim.x] = global[tid + blockDim.x];
```

12

Common bank conflict patterns (2D)

- Operating on 2D array of floats in shared memory
  - e.g., image processing
- Example: 32x32 block
  - Each thread processes a row
  - So threads in a block access the elements in each column simultaneously (example: row 1 in purple)
  - 32-way bank conflicts: rows all start at bank 0
- Solution 1. Padding
  - Add one float to the end of each row

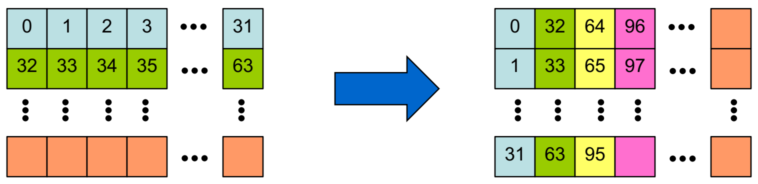




13

Matrix transpose example

- Solution 2. Transpose before processing
  - Suffer bank conflicts during transpose
  - But possibly save them later
  - Illustrates: Coalescing
  - Avoiding shared memory bank conflicts



numbers indicate array element

14

Uncoalesced transpose

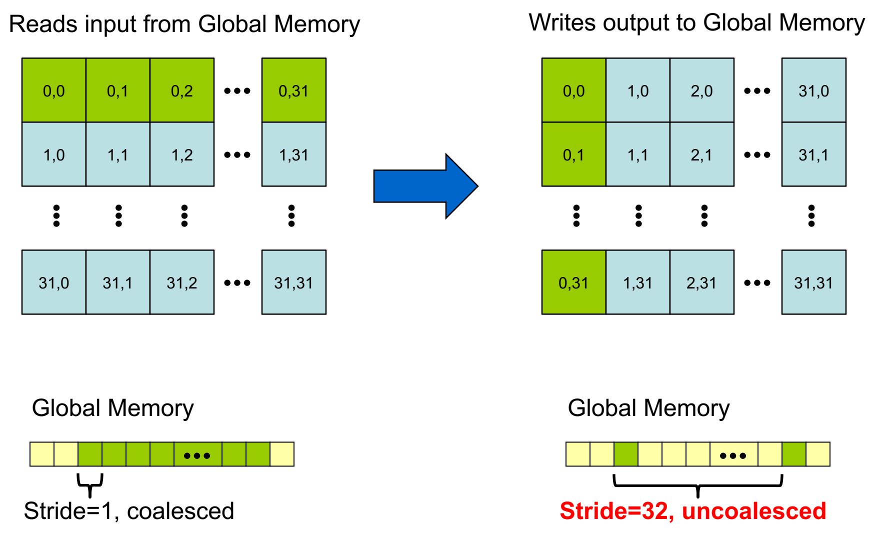
```
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

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

15

Uncoalesced transpose: memory access pattern

numbers of boxes represent array element indexes



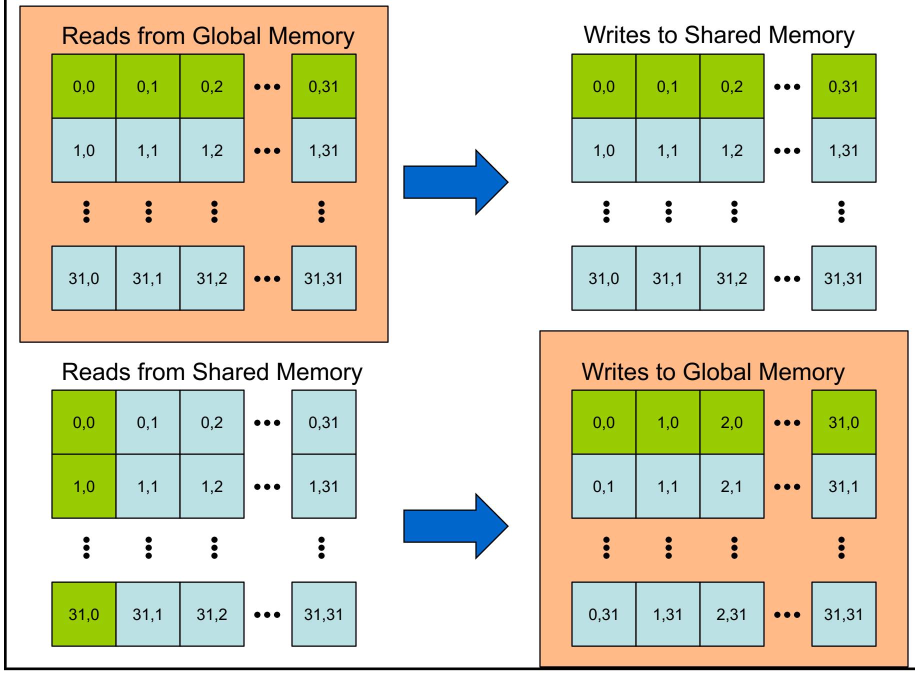
16

Coalesced transpose

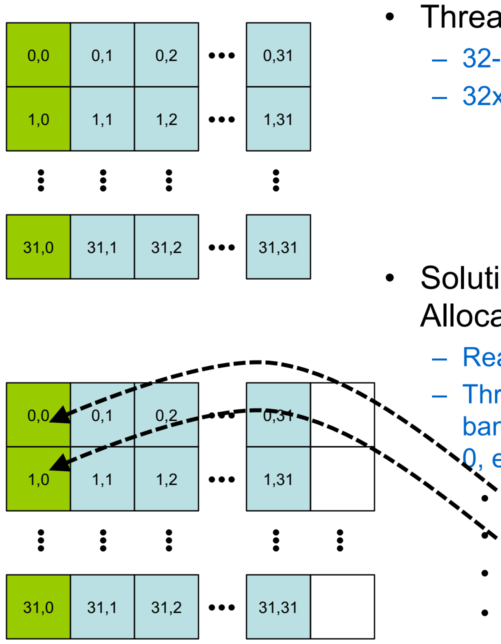
- Conceptually partition the input matrix into square tiles
- Threadblock (bx, by):
  - Read the (bx,by) input tile, store into SMEM
  - Write the SMEM data to (by,bx) output tile
    - Transpose the indexing into SMEM
- Thread (tx,ty):
  - Reads element (tx,ty) from input tile
  - Writes element (tx,ty) into output tile
- Coalescing is achieved if:
  - Block/tile dimensions are multiples of 32

17

Coalesced transpose: access patterns



Avoiding bank conflicts in shared memory



- Threads read SMEM with stride
  - 32-way bank conflicts
  - 32x slower than no conflicts
- Solution: Padding
  Allocate an "extra" column
  - Read stride = 33 elements
  - Threads read from consecutive banks. Assuming starting address is 0, element size is 4 (float):
    - ((33*4)*0)/4 mod 32 → bank 0
    - ((33*4)*1)/4 mod 32 → bank 1
    - ((33*4)*2)/4 mod 32 → bank 2
    - …
    - remember: bank = (addr/4) % 32

Coalesced transpose

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

  unsigned int xBlock = __mul24(blockDim.x, blockIdx.x);
  unsigned int yBlock = __mul24(blockDim.y, blockIdx.y);
  unsigned int xIndex = xBlock + threadIdx.x;
  unsigned int yIndex = yBlock + threadIdx.y;
  unsigned int index_out, index_transpose;

  if (xIndex < width && yIndex < height)
  {
    unsigned int index_in = __mul24(width, yIndex) + xIndex;
    unsigned int index_block = __mul24(threadIdx.y, BLOCK_DIM+1) + threadIdx.x;
    block[index_block] = idata[index_in];
    index_transpose = __mul24(threadIdx.x, BLOCK_DIM+1) + threadIdx.y;
    index_out = __mul24(height, xBlock + threadIdx.y) + yBlock + threadIdx.x;
  }
  __syncthreads();

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

42

20

Transpose measurements

- Average over 10K runs
- 32x32 blocks

- 128x128 array → 1.3x
  - Optimized: 17.5 μs
  - Naïve: 23 μs

- 512x512 array → 8.0x
  - Optimized: 108 μs
  - Naïve: 864.6 μs

- 1024x1024 array → 10x
  - Optimized: 423.2 μs
  - Naïve: 4300.1 μs

21

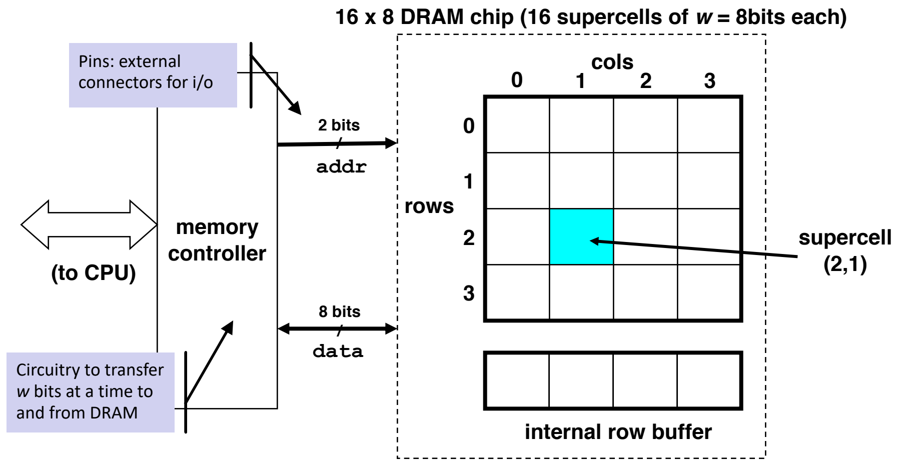
Transpose detail

- 512x512 array, 32x32 blocks

• Naïve:	864.6
• Optimized w/ shared memory:	430.1
• Optimized w/ shared memory & padding:	108

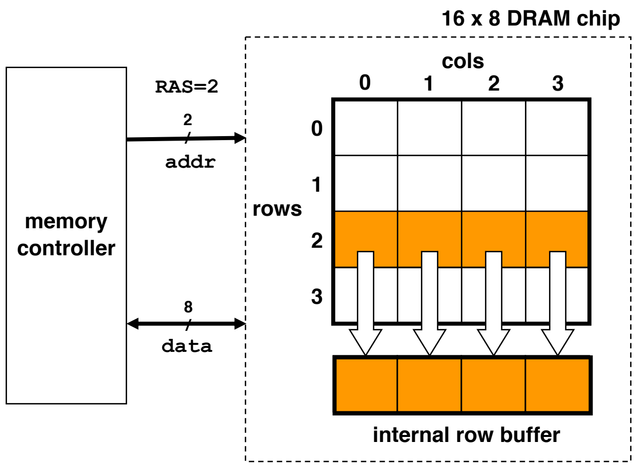
Conventional DRAM Organization

- DRAM chip partitioned into supercells
  - Each supercell consists of *w* DRAM cells
  - Supercells organized as arrays of *r* rows and *c* cols (*r***c*=*d*)
  - A *(d × w)* DRAM stores *(d*w)* bits


16 x 8 DRAM chip (16 supercells of *w* = 8bits each)
Pins: external connectors for i/o
2 bits
addr
cols
0
1
2
3
rows
0
1
2
3
memory controller
(to CPU)
supercell (2,1)
8 bits
data
Circuitry to transfer *w* bits at a time to and from DRAM
internal row buffer

Reading DRAM supercell (2,1)

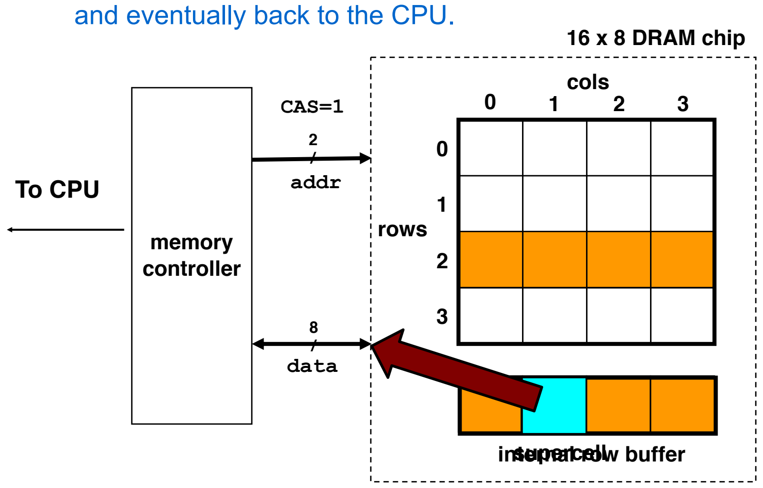
- Access done in two steps
  - Step 1(a): Row access strobe (**RAS**) selects row 2.
  - Step 1(b): Row copied from DRAM array to row buffer.



24

Reading DRAM supercell (2,1)

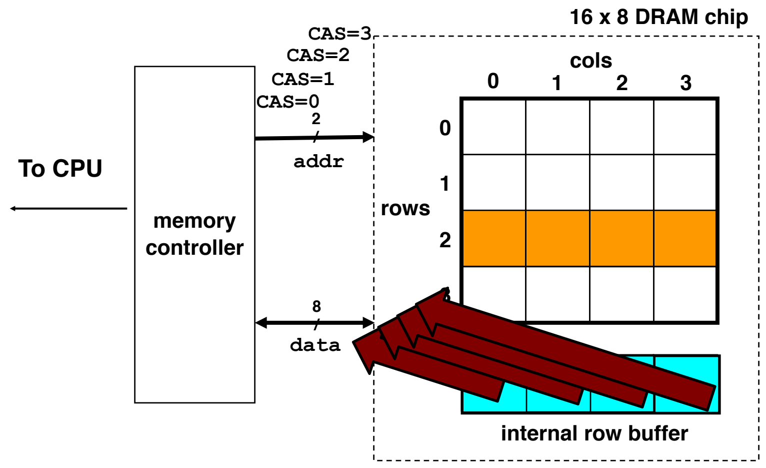
- …
  - Step 2(a): Column access strobe (**CAS**) selects column 1.
  - Step 2(b): Copy supercell (2,1) from row buffer to data lines, and eventually back to the CPU.



25

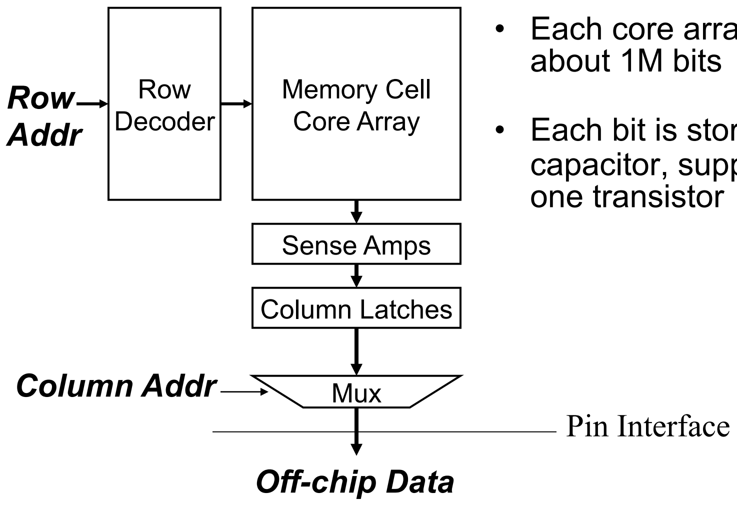
Fast Page Mode: Reading (2,0)(2,1)(2,2)(2,3)

- …
  - Consecutive **CAS** select consecutive columns of the same row
  - Direct row buffer access, no precharge → VERY fast memory access



26

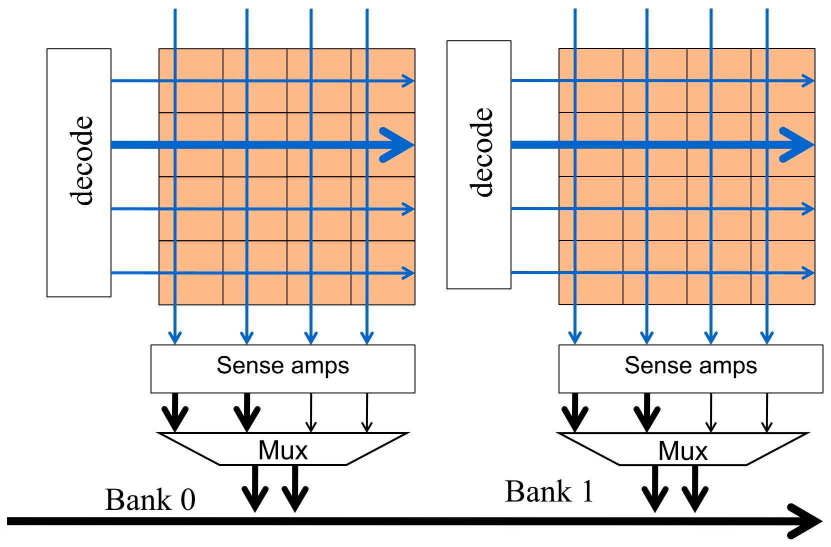
DRAM bank organization



- Each core array has about 1M bits
- Each bit is stored in a tiny capacitor, supported by one transistor

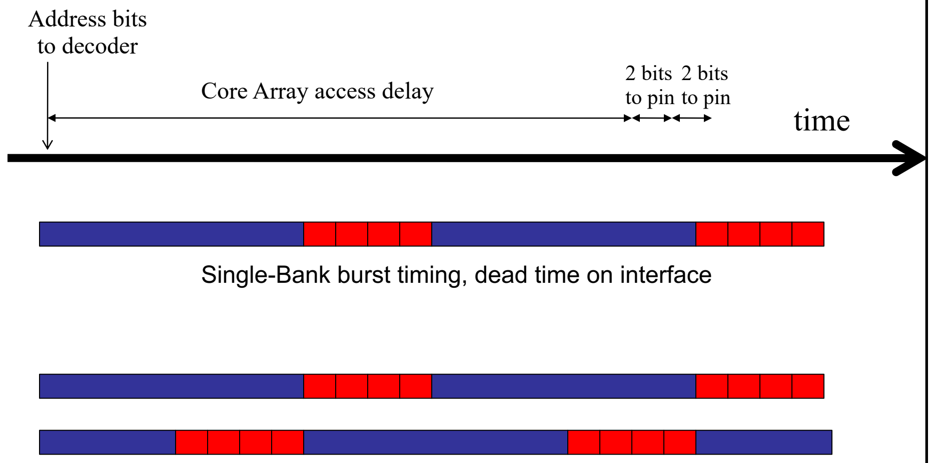
27

Multiple DRAM banks



28

DRAM bursting for the 8x2 bank



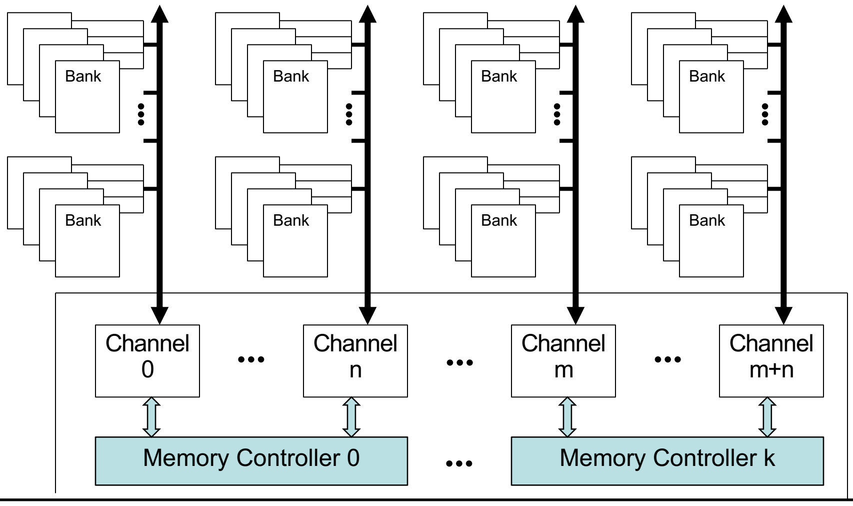
Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time

29

Multiple memory channels

- Divide the memory address space into N parts
  - N is number of memory channels
  - Assign each memory portion to a channel
  - Further subdivide each channel's worth of memory into banks



30

First-order look at the GPU off-chip memory

- nVidia GTX280 GPU:
  - Peak global memory bandwidth = 141.7GB/s
- Global memory (GDDR3) interface @ 1.1GHz
  - (Core speed @ 276Mhz)
  - For a typical 16-bit interface, we can sustain only about 17.6 GB/s
  - We need a lot more bandwidth (141.7 GB/s)
    - Thus, 8 memory channels

31

Memory controller organization

- GTX280: 30 Stream Multiprocessors (SM) connected to 8 DRAM controllers through interconnect
  - DRAM controllers are interleaved
  - Within DRAM controllers (channels), DRAM banks are interleaved for incoming memory requests
  - We approximate the DRAM channel/bank interleaving scheme through micro-benchmarking

32

Back to the big picture

- Each global memory access is made to a memory location with an address
  - Some bits will determine the memory channel used
  - Some bits will determine the DRAM bank used
  - Some bits will determine the position within a burst

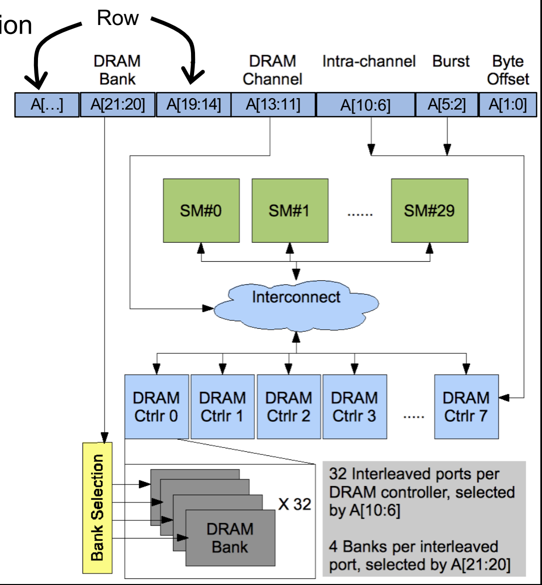
Address:

Other bits	Channel	Bank	Burst

33

The *likely* memory organization of GTX280 *(partial)*

- Address bits A[5:2] together with thread index control **memory coalescing**
- Address bits A[13:6] help steer a request to a memory bank (***steering bits***)
- Micro-benchmarking to reverse-engineer the memory organization
- Likely memory addressing:
  - There are 8 channels
    - A[13:11] select channel
  - There are 32 ports/channel
    - A[10:6] select port
  - There are 4 banks/port
    - A[21:20] select bank
  - There are ??? rows/bank
    - A[???] select row
    - At least 64 rows
  - There are 64 bytes/row
    - A[5:2] Select 32-bit burst
      - 16 bursts per row
    - A[1:0] Select byte



34

Data layout transformations for the GPU

- To exploit parallelism in the memory hierarchy, simultaneous Global Memory accesses from or across warps need to have the *right* address bit patterns at critical bit fields
  - For accesses from the same warp:
    - Patterns at those bit fields that control coalescing should match the thread index
    - e.g., A[5:2] for GTX280
    - It is safe to assume A[6:2] for GTX680
  - For accesses across warps:
    - Patterns at steering bit fields which are used to decode channel/banks should be as distinct as possible
    - e.g., A[13:6] for GTX280

35