King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming Lecture 16: CUDA Memories, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #6 (until Oct 14)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 5** (*Memory architecture and data locality*)

Read (optional):

- Programming Massively Parallel Processors book (4th edition), **Chapter 20** (*An introduction to CUDA streams*)
- Programming Massively Parallel Processors book (4th edition), **Chapter 21** (*CUDA dynamic parallelism*)

Reading Assignment #7 (until Oct 21)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 6** (*Performance considerations*)

Read (optional):

- Inline PTX Assembly in CUDA: `Inline_PTX_Assembly.pdf`
- Dissecting GPU Architectures through Microbenchmarking:

  Volta: `https://arxiv.org/abs/1804.06826`
  Turing: `https://arxiv.org/abs/1903.07486`

  `https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9839-discovering-the-turing-t4-gpu-architecture-with-microbenchmarks.pdf`

  Ampere: `https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322/`

Next Lectures

no lecture on Oct 14 ! (fall semester break)

Lecture 17: Tue, Oct 15: Vulkan tutorial (room 3128, 14:30-15:45)

Lecture 18: Thu, Oct 17: Quiz #2 (only quiz; room 3128, 10:00)

Lecture 19: Mon, Oct 21

Lecture 20: Tue, Oct 22 (make-up lecture; 14:30 – 15:45)

Lecture 21: Thu, Oct 24

Example: Matrix Multiplication

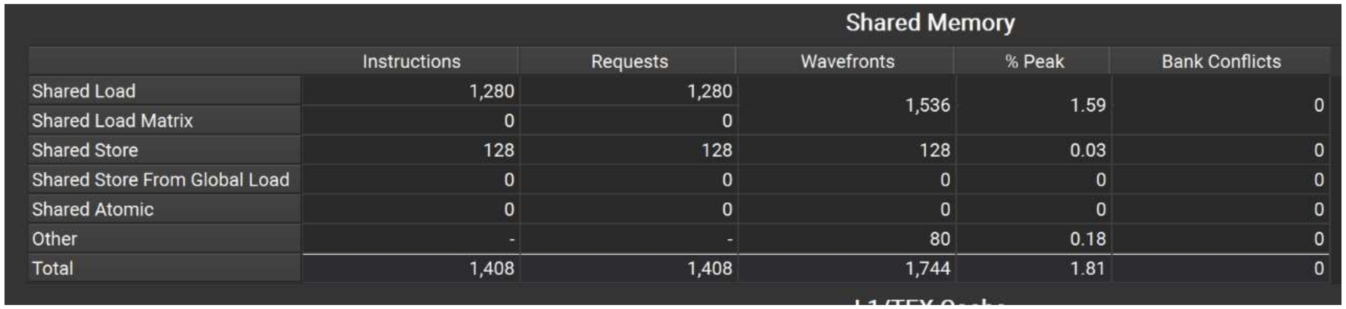
```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
      __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
      __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

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

      int col = bx * BLOCK_SIZE + tx;
      int row = by * BLOCK_SIZE + ty;

      float out = 0.0f;
      for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {

          blockA[ ty ][ tx ] = matA[ row * w +   m * BLOCK_SIZE + tx       ];
          blockB[ ty ][ tx ] = matB[ col     + ( m * BLOCK_SIZE + ty ) * w ];
          __syncthreads();

          for ( int k = 0; k < BLOCK_SIZE; k++ ) {
              out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
          }
          __syncthreads();
      }

      matC[ row * w + col ] = out;
}
```

Caveat: for brevity, this code assumes matrix sizes are a multiple of the block size (either because they really are, or because padding is used; otherwise guard code would need to be added)

Example: Matrix Multiplication

```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
    __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
    __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

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

    int col = bx * BLOCK_SIZE + tx;
    int row = by * BLOCK_SIZE + ty;

    float out = 0.0f;
    for ( int m = 0; m < w

        blockA[ ty ][ tx ]
        blockB[ ty ][ tx ]
        __syncthreads();

        for ( int k = 0; k < BLOCK_SIZE; k++ ) {
            out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
        }
        __syncthreads();
    }

    matC[ row * w + col ] = out;
}
```

here: *no bank conflicts* (using row-major order);
but in general be careful with block sizes!



Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	1,280	1,280	1,536	1.59	0
Shared Load Matrix	0	0			
Shared Store	128	128	128	0.03	0
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	80	0.18	0
Total	1,408	1,408	1,744	1.81	0

Caveat: for brevity, this code assumes matrix sizes are a multiple of the block size (either because they really are, or because padding is used; otherwise guard code would need to be added)

CUDA Memory: Shared Memory

Memory and Cache Types

Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

L1 Cache vs. Shared Memory

Different configs on Fermi and Kepler; carveout on Maxwell and newer

- More shared memory on newer GPUs (64KB, 96KB, 100KB, 164KB, ...)

  Carveout from unified L1/read-only data cache

  (See CUDA C Programming Guide!)

```
// Device code
__global__ void MyKernel(...)
{
    __shared__ float buffer[BLOCK_DIM];
    ...
}

// Host code
int carveout = 50; // prefer shared memory capacity 50% of maximum
// Named Carveout Values:
// carveout = cudaSharedmemCarveoutDefault;   //  (-1)
// carveout = cudaSharedmemCarveoutMaxL1;     //   (0)
// carveout = cudaSharedmemCarveoutMaxShared; // (100)
cudaFuncSetAttribute(MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout,
 carveout);
MyKernel <<<gridDim, BLOCK_DIM>>>(...);
```

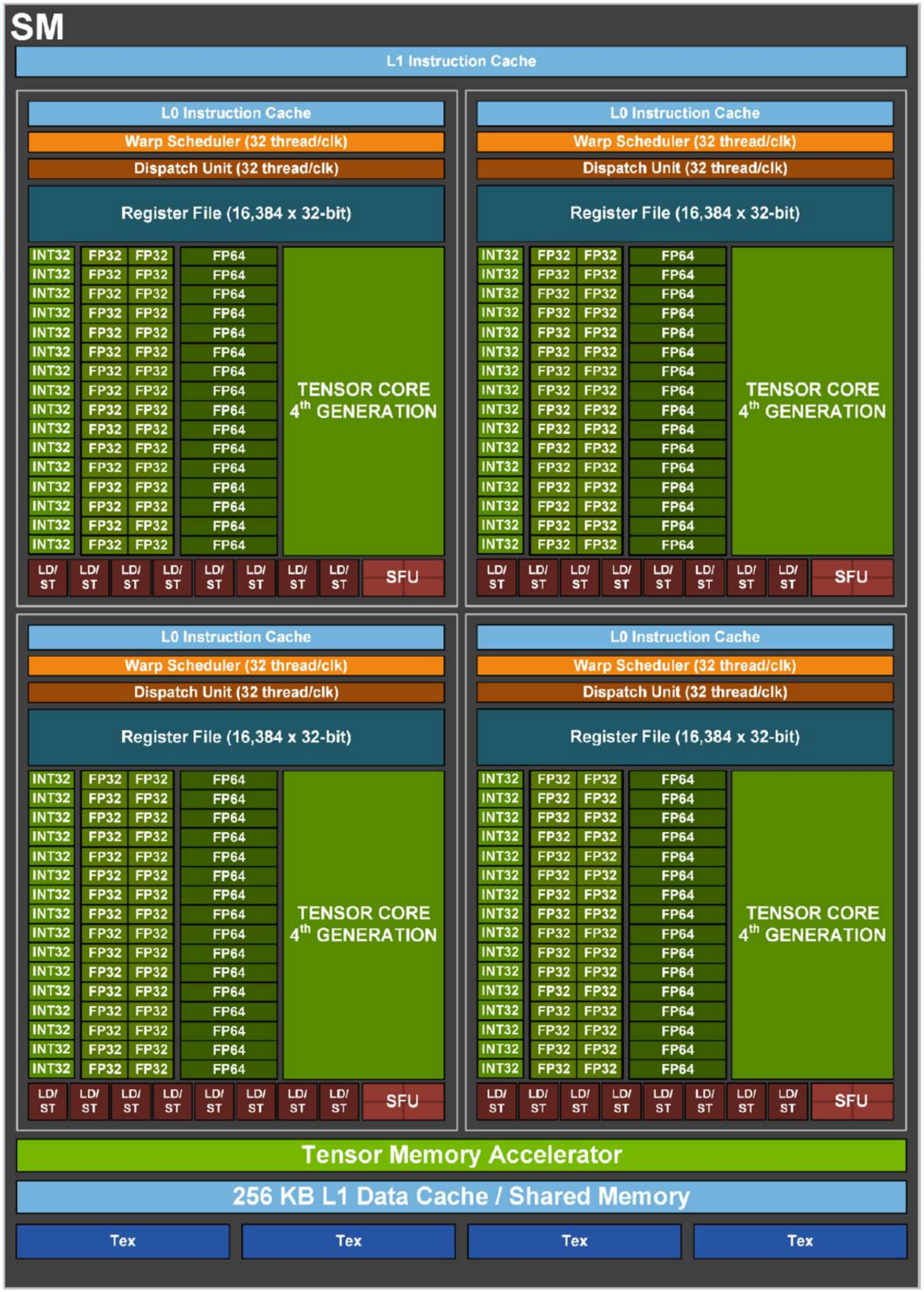
NVIDIA GH100 SM

Multiprocessor: SM (CC 9.0)

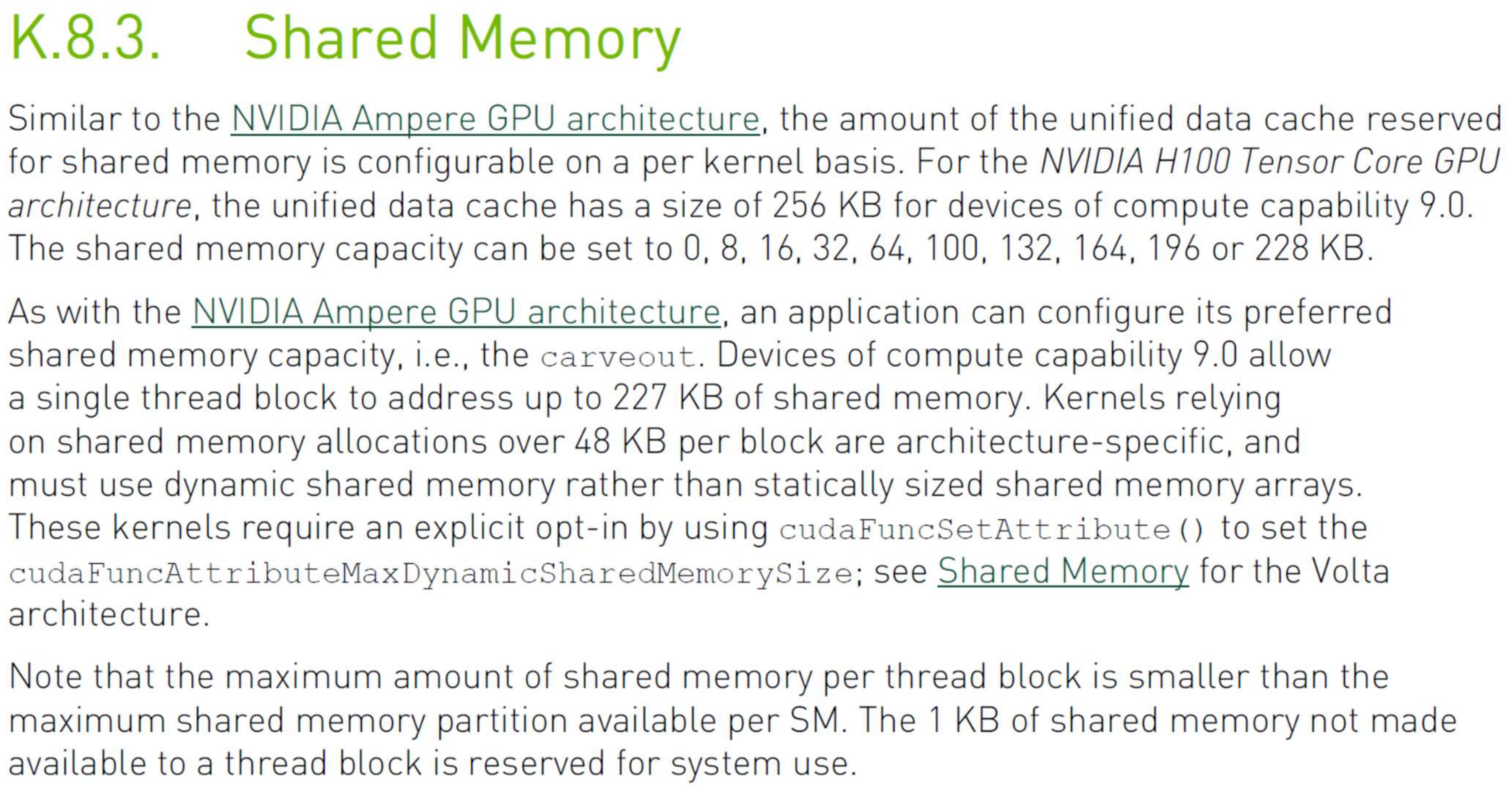
- 128 FP32 + 64 INT32 cores
- 64 FP64 cores
- 4x 4th gen tensor cores
- ++ thread block clusters, DPX insts., FP8, TMA

4 partitions inside SM

- 32 FP32 + 16 INT32 cores
- 16 FP64 cores
- 8x LD/ST units; 4 SFUs each
- 1x 4th gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

Compute Capab. 9.x (Hopper, Part 2)



K.8.3. Shared Memory

Similar to the NVIDIA Ampere GPU architecture, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *NVIDIA H100 Tensor Core GPU architecture*, the unified data cache has a size of 256 KB for devices of compute capability 9.0. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132, 164, 196 or 228 KB.

As with the NVIDIA Ampere GPU architecture, an application can configure its preferred shared memory capacity, i.e., the `carveout`. Devices of compute capability 9.0 allow a single thread block to address up to 227 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see Shared Memory for the Volta architecture.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

Shared Memory Allocation

- **2 modes**
- **Static size within kernel**

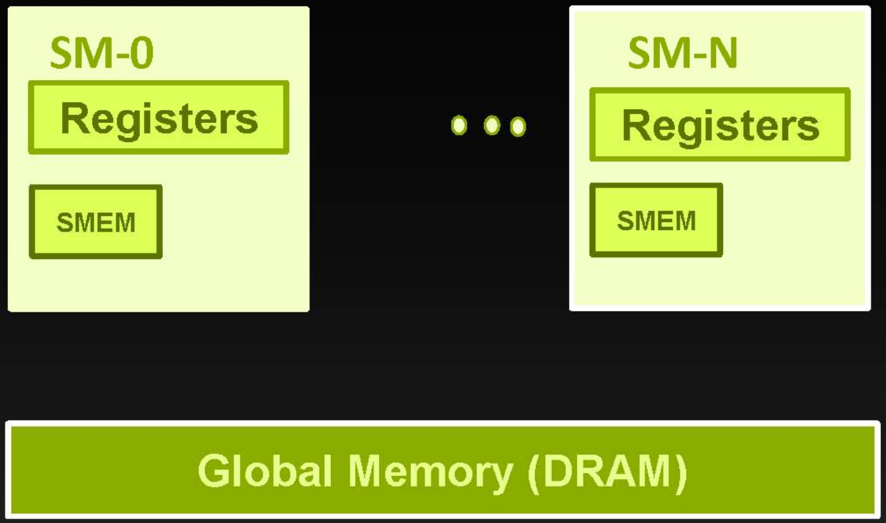
```
__shared__ float vec[256];
```

- **Dynamic size when calling the kernel**

```
// in main
int VecSize = MAX_THREADS * sizeof(float4);
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, …);

// declare as extern within kernel
extern __shared__ float vec[];
```

Shared Memory

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion



Shared Memory/L1 Sizing

- Shared memory and L1 use the same 64KB
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: ~~cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()~~
- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)
- Large SMEM can improve performance when:
  - Occupancy is limited by SMEM

later: use *carveout*

use cudaFuncSetAttribute()

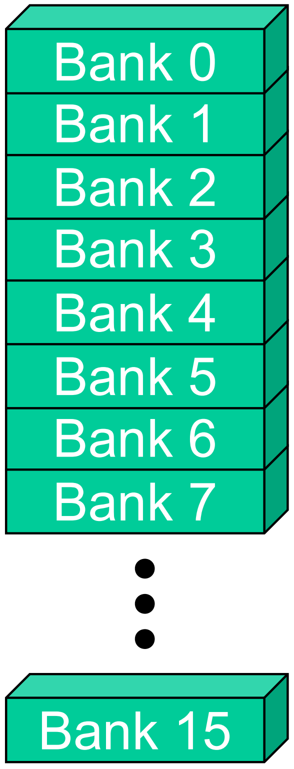
Shared Memory

- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns

- Organization:
  - 32 banks, 4-byte (or 8-byte) banks
  - Successive words accessed through different banks

Parallel Memory Architecture

- 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



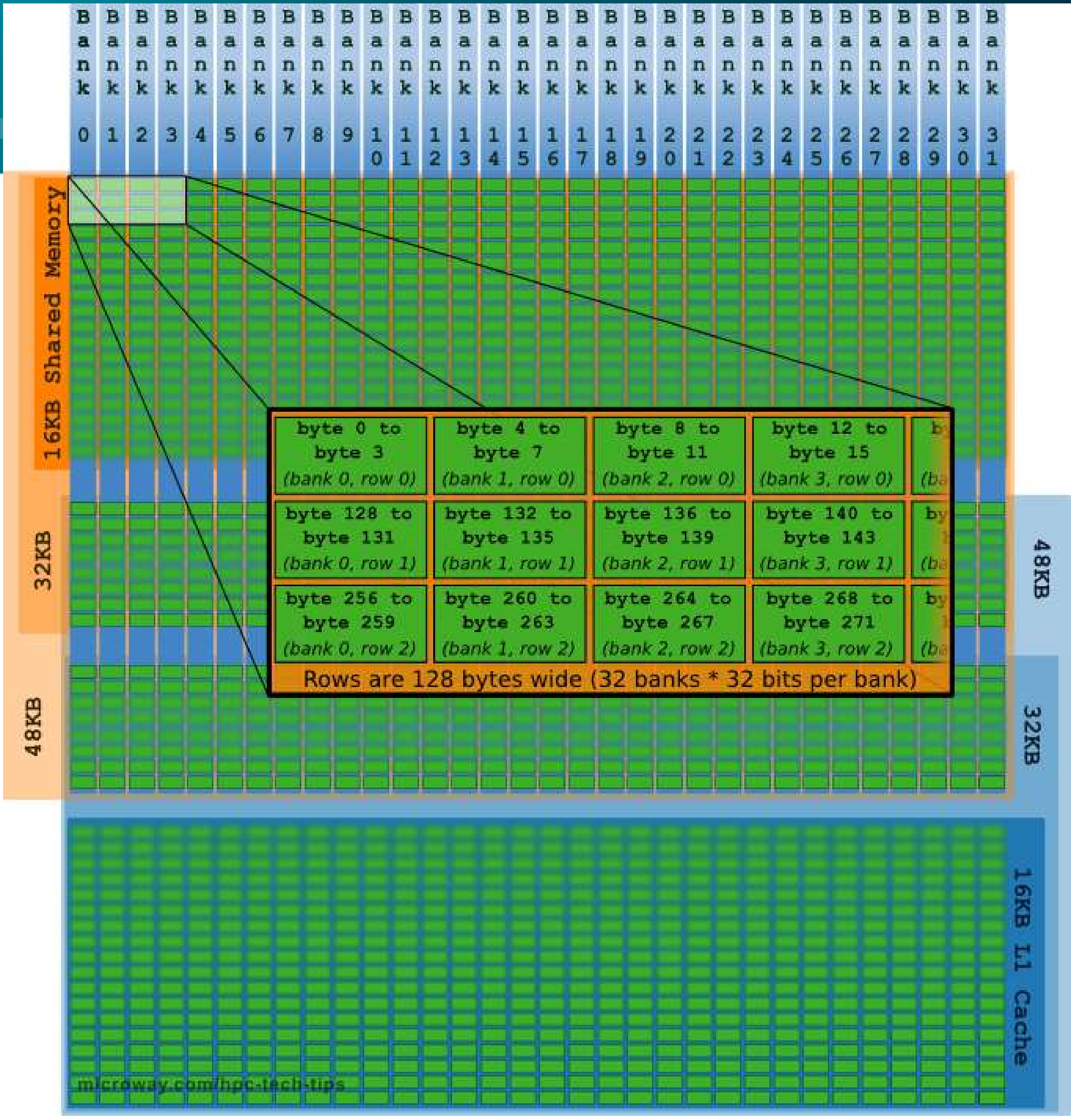
Memory Banks

Fermi/Kepler/Maxwell
 and newer:

32 banks

default:
 4B / bank

Kepler or newer:
 configurable
 to 8B / bank

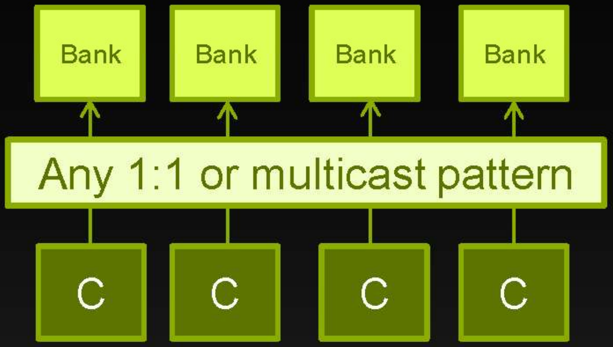

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Bank 12
Bank 13
Bank 14
Bank 15
Bank 16
Bank 17
Bank 18
Bank 19
Bank 20
Bank 21
Bank 22
Bank 23
Bank 24
Bank 25
Bank 26
Bank 27
Bank 28
Bank 29
Bank 30
Bank 31
16KB Shared Memory
32KB
48KB
48KB
32KB
16KB L1 Cache
byte 0 to byte 3 (bank 0, row 0)
byte 4 to byte 7 (bank 1, row 0)
byte 8 to byte 11 (bank 2, row 0)
byte 12 to byte 15 (bank 3, row 0)
byte 128 to byte 131 (bank 0, row 1)
byte 132 to byte 135 (bank 1, row 1)
byte 136 to byte 139 (bank 2, row 1)
byte 140 to byte 143 (bank 3, row 1)
byte 256 to byte 259 (bank 0, row 2)
byte 260 to byte 263 (bank 1, row 2)
byte 264 to byte 267 (bank 2, row 2)
byte 268 to byte 271 (bank 3, row 2)
Rows are 128 bytes wide (32 banks * 32 bits per bank)
microway.com/hpc-tech-tips

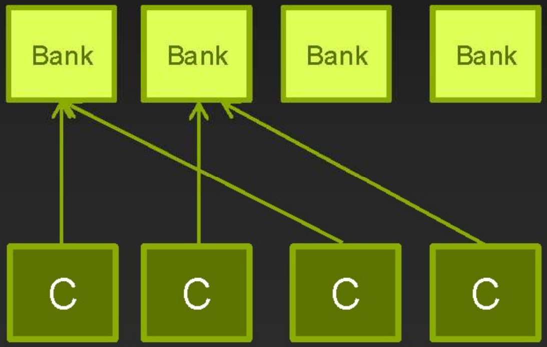
Shared Memory

- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Performance:
  - smem accesses are issued per warp
  - Throughput is 4 (or 8) bytes per bank per clock per multiprocessor
  - serialization: if *N* threads of 32 access different words in the same bank, *N* accesses are executed serially
  - multicast: *N* threads access the same word in one fetch
    - Could be different bytes within the same word

Shared Memory Organization

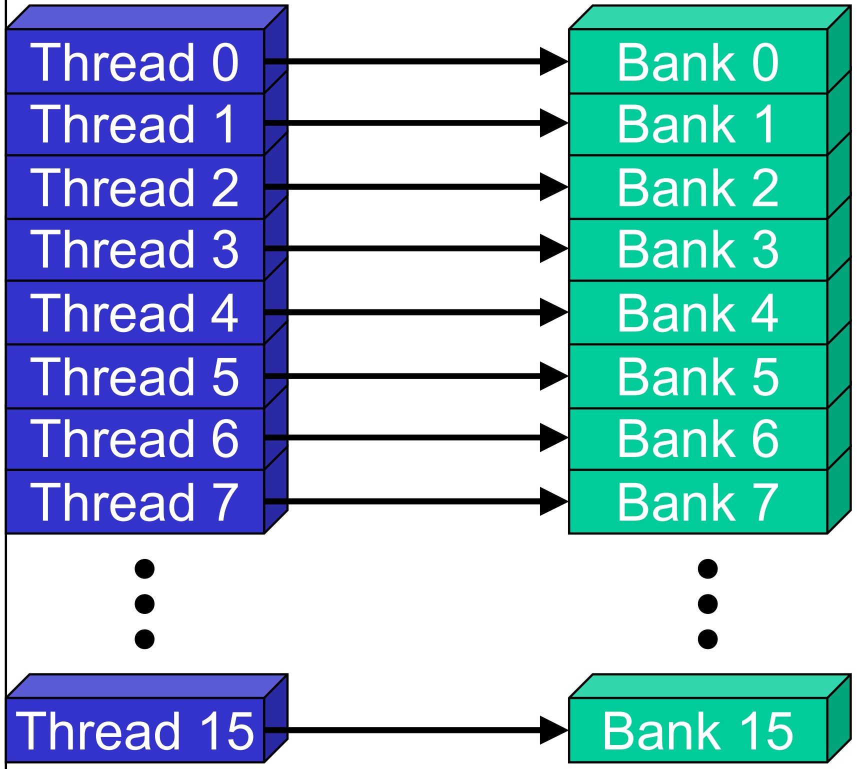
- Organized in 32 independent banks
- Optimal access: no two words from same bank
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize



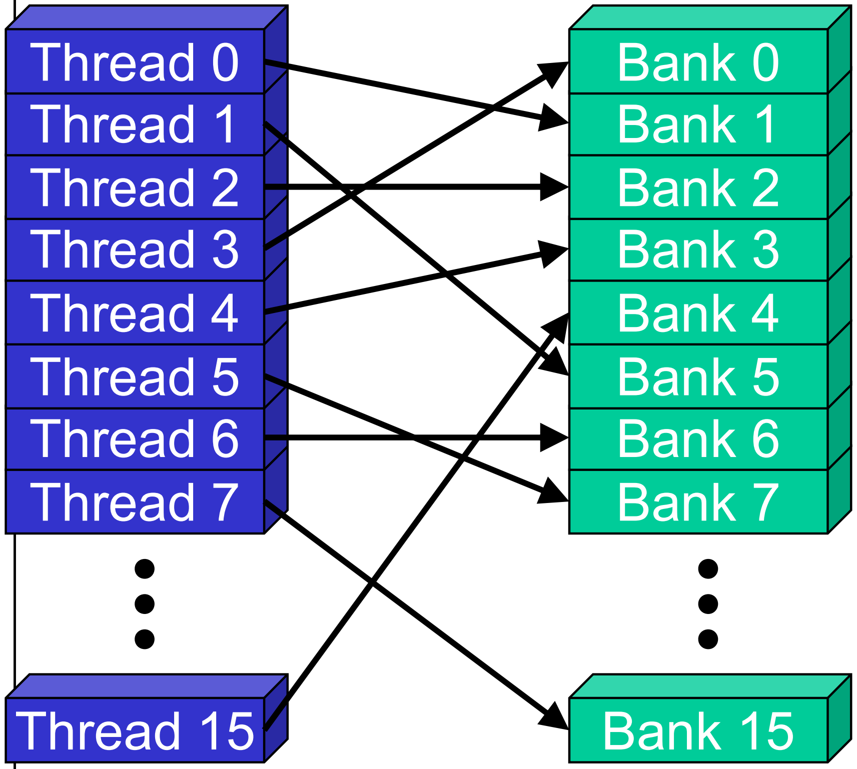


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

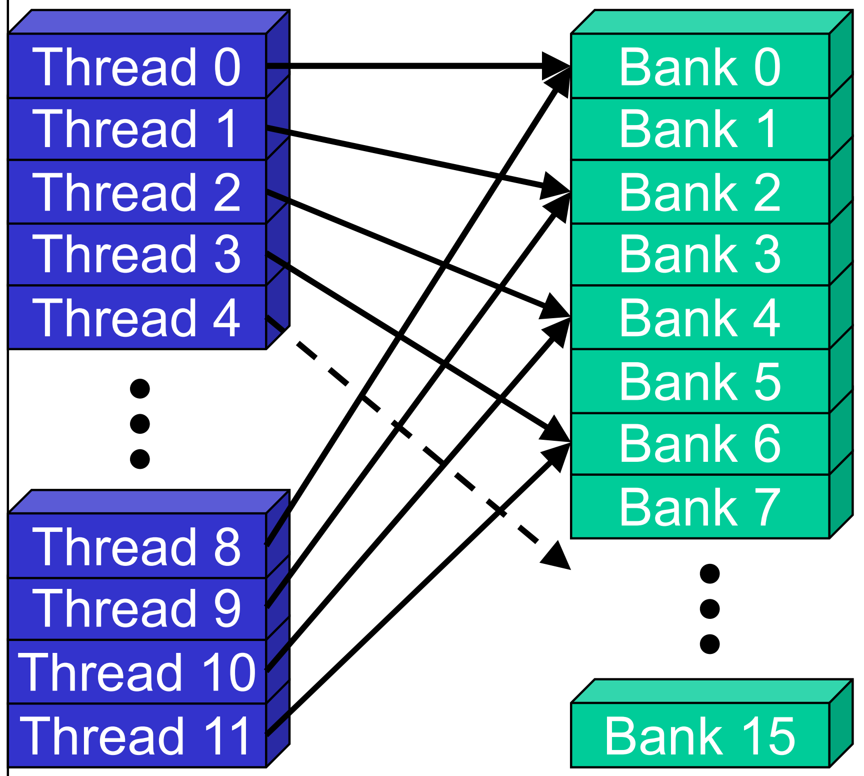


- No Bank Conflicts
  - Random 1:1 Permutation

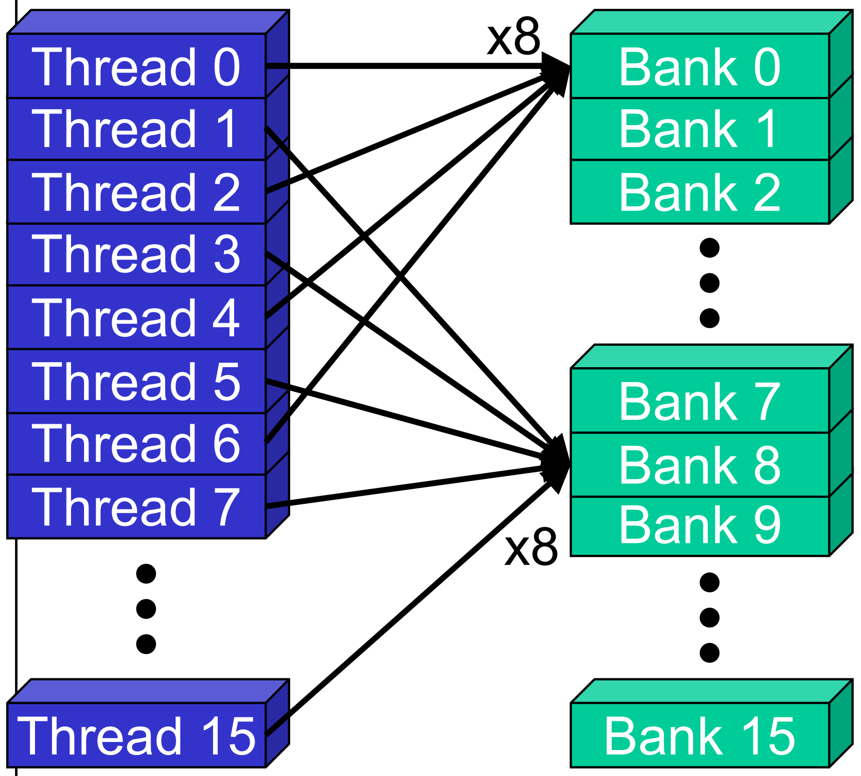


Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



How addresses map to banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Fermi and newer have 32 banks,
considers full warps instead of half warps!

Shared Memory Bank Conflicts

- **Shared memory is as fast as registers if there are no bank conflicts**

- **The fast case:**
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- **The slow case:**
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

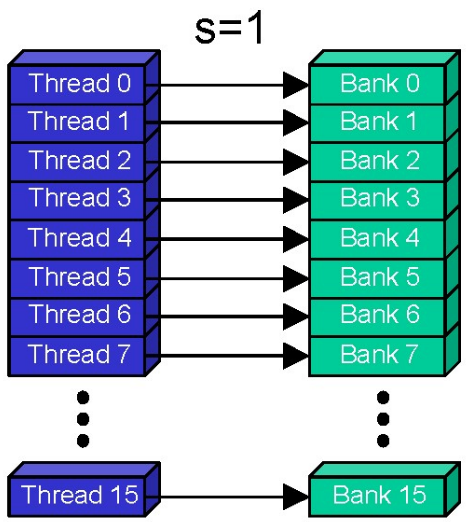
full warps instead of half warps on Fermi and newer!

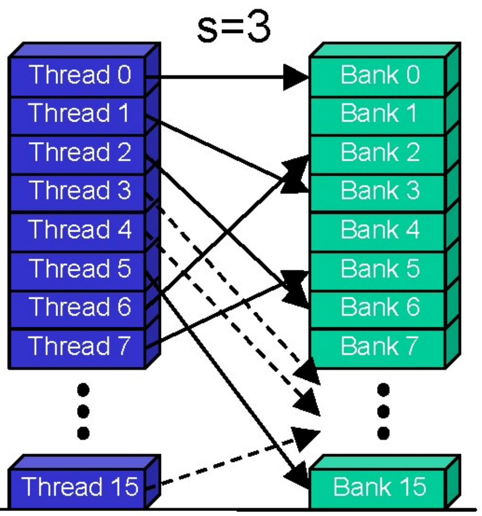
Linear Addressing

- Given:

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

- This is only bank-conflict-free if s shares no common factors with the number of banks
  - 16 on G80, so s must be odd

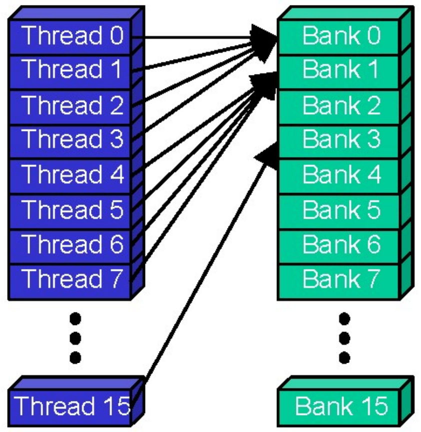




Data Types and Bank Conflicts

- **This has no conflicts if type of `shared` is 32-bits:**

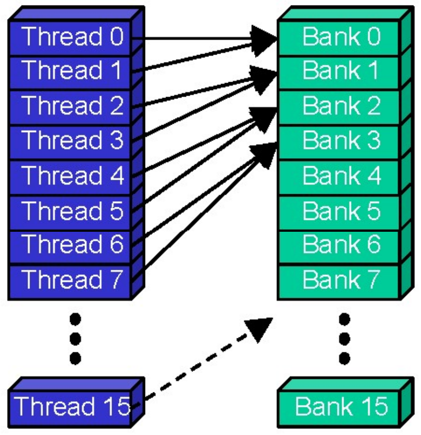
```
foo = shared[baseIndex + threadIdx.x]
```



- **But not if the data type is smaller**
  - 4-way bank conflicts:

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

not true on Fermi, because of multi-cast!



  - 2-way bank conflicts:

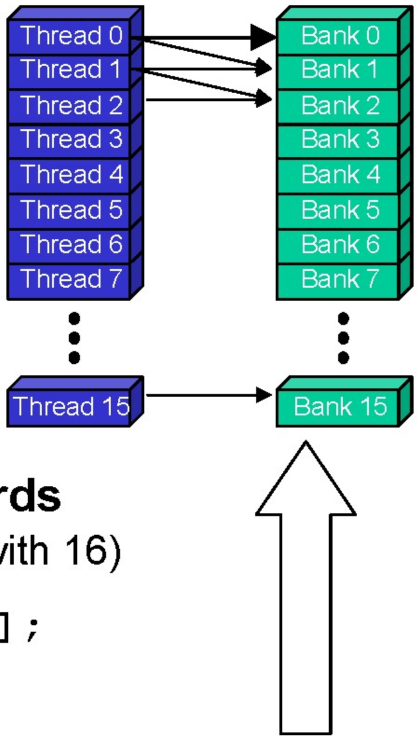
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```

not true on Fermi, because of multi-cast!

Structs and Bank Conflicts

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

```
struct vector { float x, y, z; };
struct myType {
    float f;
    int c;
};
__shared__ struct vector vectors[64];
__shared__ struct myType myTypes[64];
```



- **This has no bank conflicts for vector; struct size is 3 words**
  - 3 accesses per thread, contiguous banks (no common factor with 16)

```
struct vector v = vectors[baseIndex + threadIdx.x];
```

- **This has 2-way bank conflicts for myType;**
  (each bank will be accessed by 2 threads simultaneously)

```
struct myType m = myTypes[baseIndex + threadIdx.x];
```

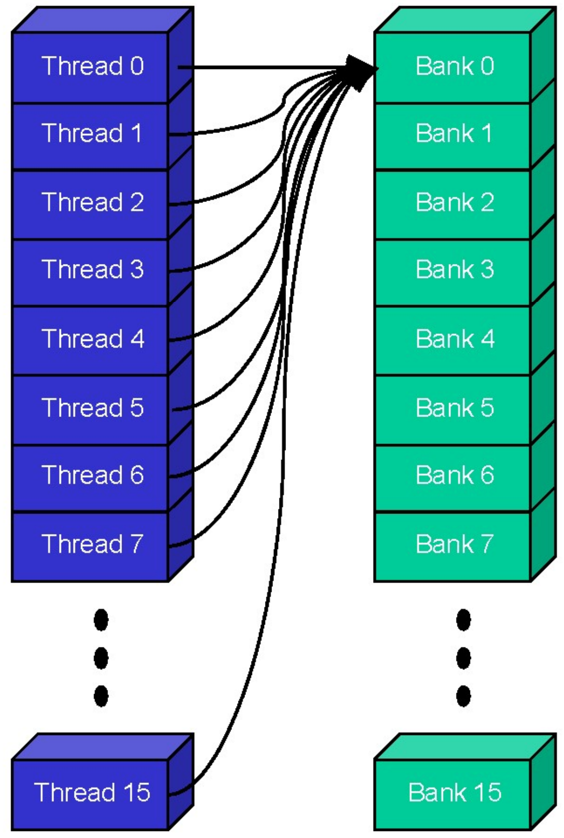
Broadcast on Shared Memory

- **Each thread loads the same element – no bank conlict**

```
x = shared[0];
```

- **Will be resolved implicitly**

multi-cast on Fermi and newer!

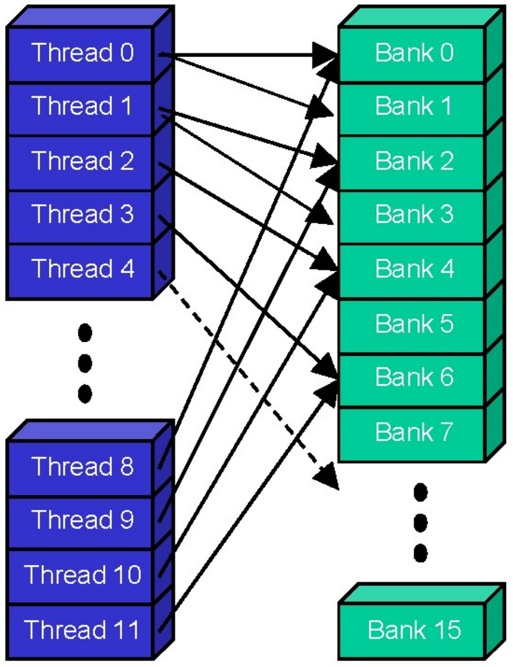


Common Array Bank Conflict Patterns 1D

- **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];
```

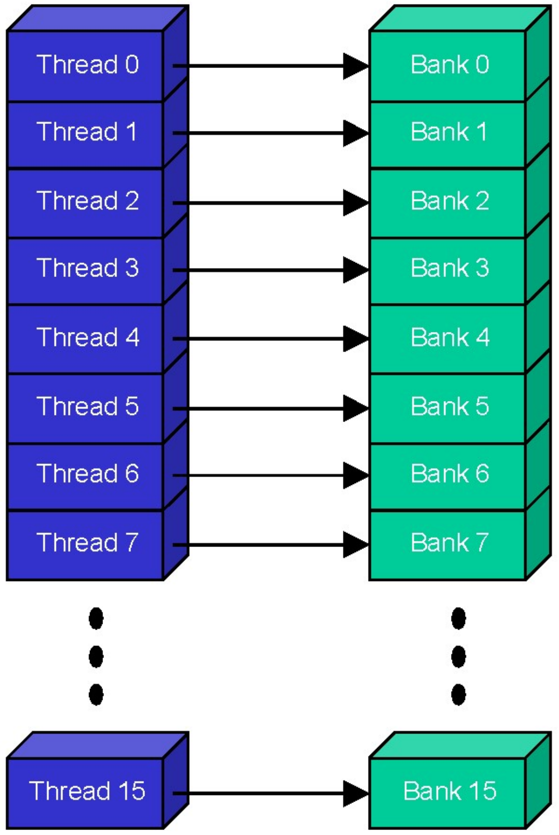
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects



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];
```



OPTIMIZE

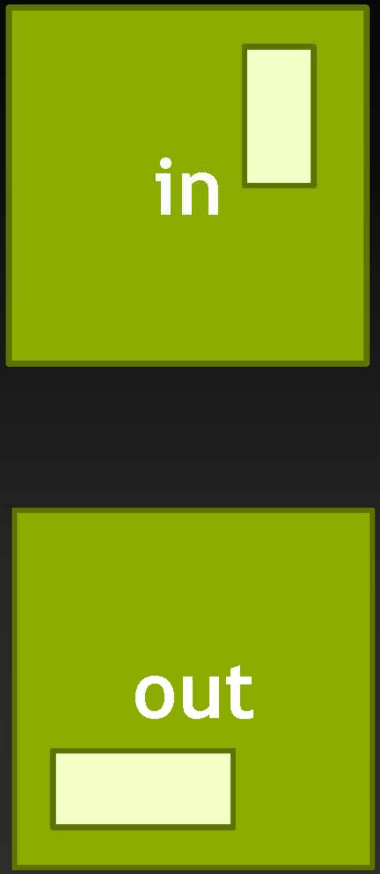
Kernel Optimizations: *Shared Memory Accesses*

Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)

⇒ Process matrix tile, not single row/column, per block

⇒ Transpose matrix tile within block

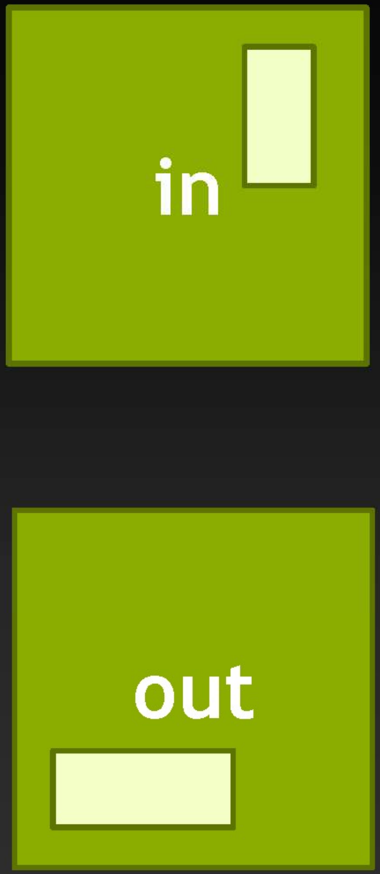


Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)

- Transpose matrix tile within block

⇒ Need threads in a block to cooperate: use shared memory



Transpose with coalesced read/write

```
__global__ transpose(float in[], float out[])
{

  __shared__ float tile[TILE][TILE];

  int glob_in = xIndex + (yIndex)*N;
  int glob_out = xIndex + (yIndex)*N;

  tile[threadIdx.y][threadIdx.x] = in[glob_in];

  __syncthreads();

  out[glob_out] = tile[threadIdx.x][threadIdx.y];

}

[illegible]
transpose<<<grid, threads>>>(in, out);
```

Fixed GMEM coalescing, but introduced SMEM bank conflicts

Transpose with coalesced read/write

```
__global__ transpose(float in[], float out[])
{

  __shared__ float tile[TILE][TILE];

  int glob_in = xIndex + (yIndex)*N;
  int glob_out = xIndex + (yIndex)*N;

  tile[threadIdx.y][threadIdx.x] = in[glob_in];

  __syncthreads();

  out[glob_out] = tile[threadIdx.x][threadIdx.y];

}
```

Fixed GMEM coalescing, but introduced SMEM bank conflicts

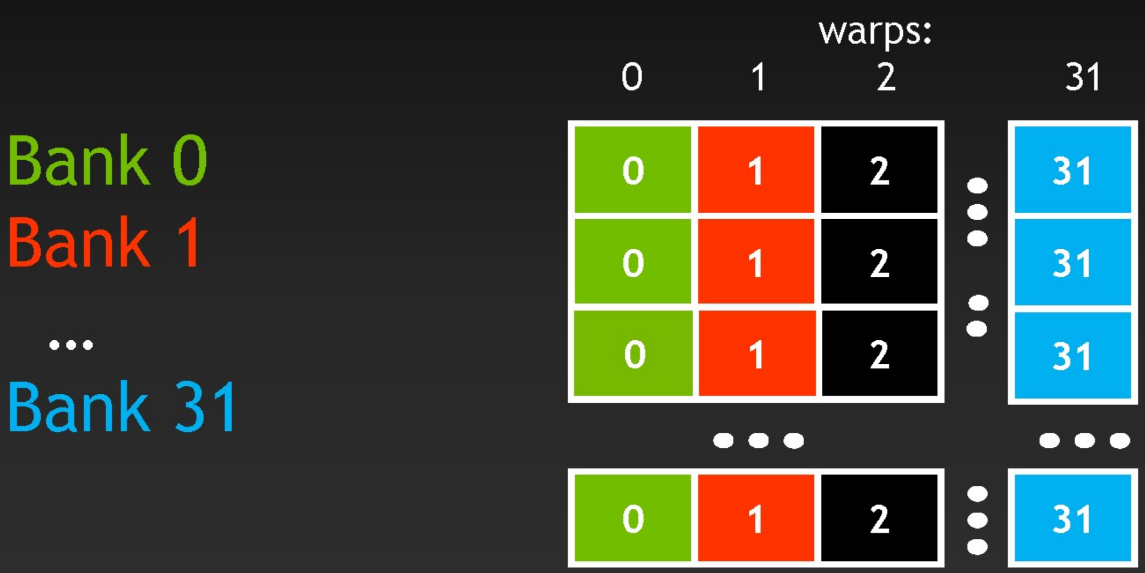
```
threads(TILE, TILE, 1);
transpose<<<grid, threads>>>(in, out);
```

Shared Memory: Avoiding Bank Conflicts

- **Example: 32x32 SMEM array**
- **Warp accesses a column:**
  - **32-way bank conflicts (threads in a warp access the same bank)**

read (LD) from shared memory

```
out[glob_out] =
    tile[threadIdx.x][threadIdx.y];
```



stride for read (LD) = **32**

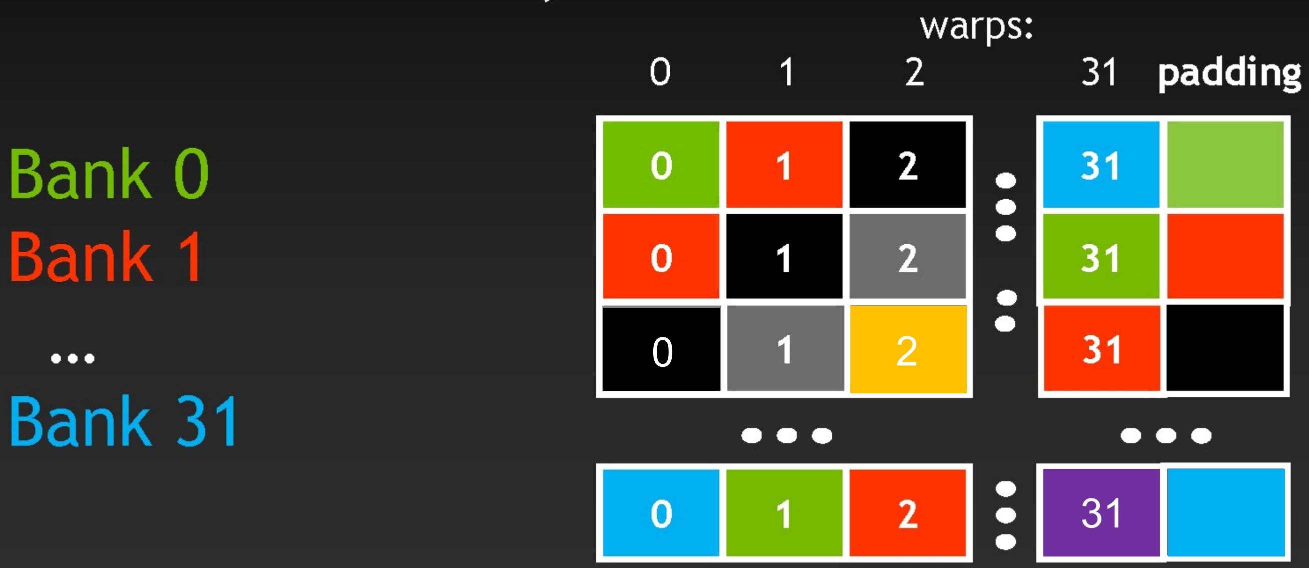
each of the 32 warps has 32-way bank conflicts!

Shared Memory: Avoiding Bank Conflicts

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts

read (LD) from shared memory

```
out[glob_out] =
    tile[threadIdx.x][threadIdx.y];
```



stride for read (LD) = **33**

none of the 32 warps has bank conflicts!

No bank conflicts anymore

```
__global__ transpose(float in[], float out[])
{

  __shared__ float tile[TILE][TILE+1];

  int glob_in = xIndex + (yIndex)*N;
  int glob_out = xIndex + (yIndex)*N;

  tile[threadIdx.y][threadIdx.x] = in[glob_in];

  __syncthreads();

  out[glob_out] = tile[threadIdx.x][threadIdx.y];

}
```

Thank you.