King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 20: CUDA Memory, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #12 (until Nov 22)

Read (required):

- Optimizing Parallel Reduction in CUDA, Mark Harris,

`https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`

- Programming Massively Parallel Processors book, 3rd edition
  Chapter 8 (Parallel Patterns: Prefix Sum)
- GPU Gems 3 book, Chapter 39: Parallel Prefix Sum (Scan) with CUDA

`https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html`

Read (optional):

- Faster Parallel Reductions on Kepler, Justin Luitjens

`https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/`

CUDA Memory:

Shared Memory

L1 Cache vs. Shared Memory

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

- 64KB total
  - 16KB shared, 48KB L1 cache
  - 48KB shared, 16KB L1 cache
  - 32KB shared, 32KB L1 cache (Kepler only)
- Set per kernel

```
// Device code
__global__ void MyKernel()
{
    ...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
```

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 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>>>(...);
```

Shared Memory Allocation

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

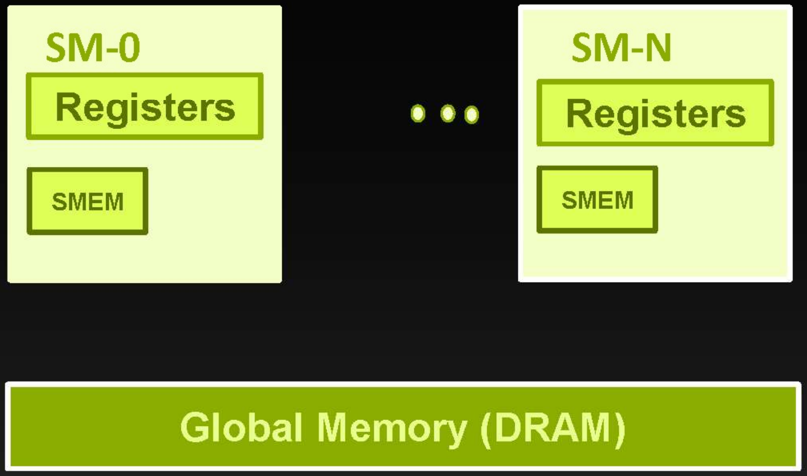
```
__shared__ float vec[256];
```

- **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

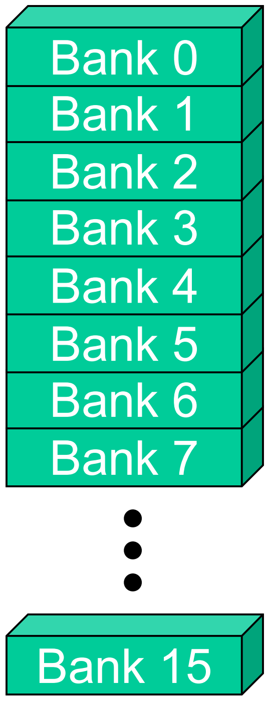
- Shared memory and L1 use the same 64KB
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)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

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



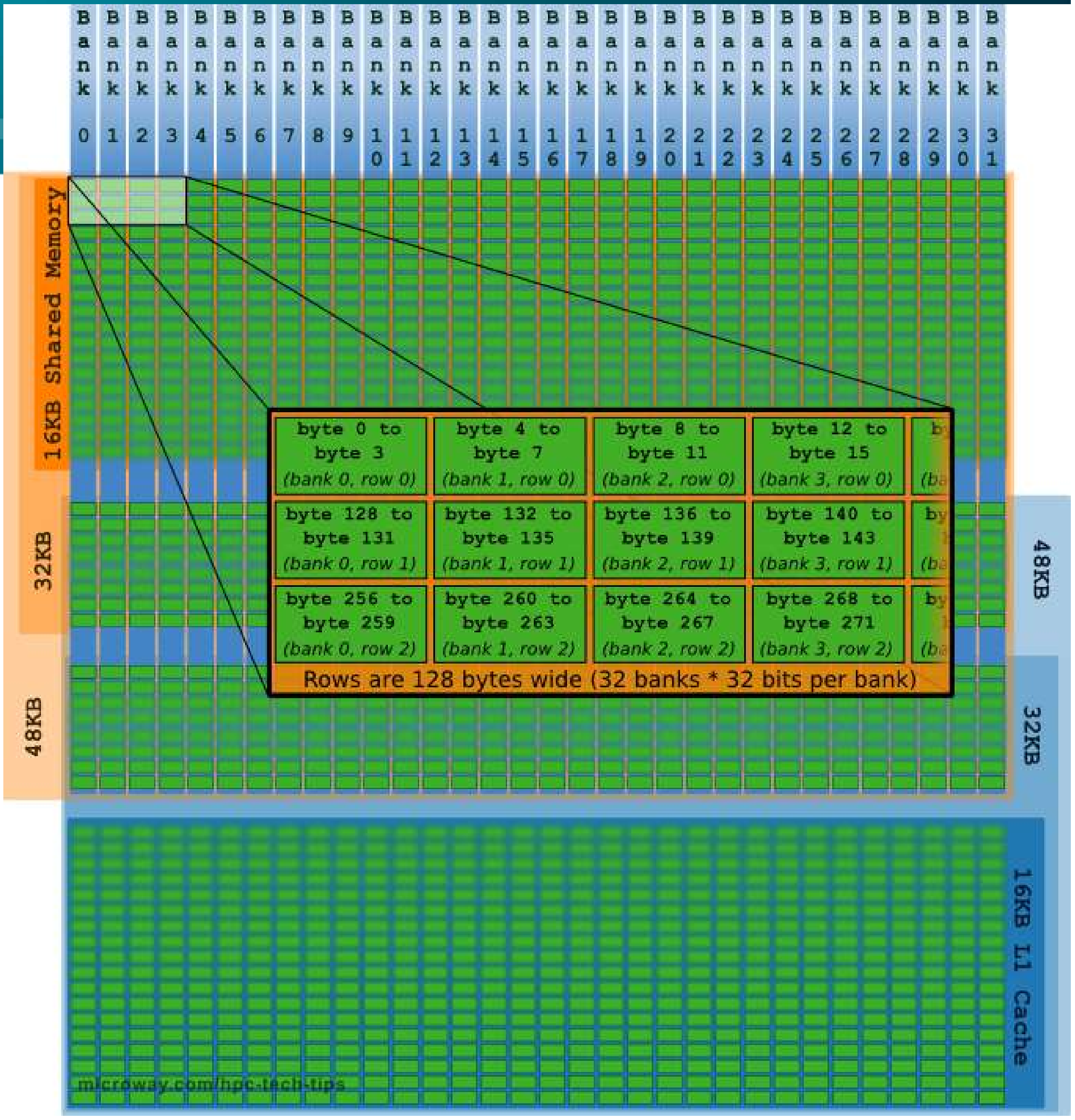
Memory Banks

Fermi/Kepler/Maxwell
and newer:

32 banks

default:
4B / bank

Kepler or newer:
configurable
to 8B / bank

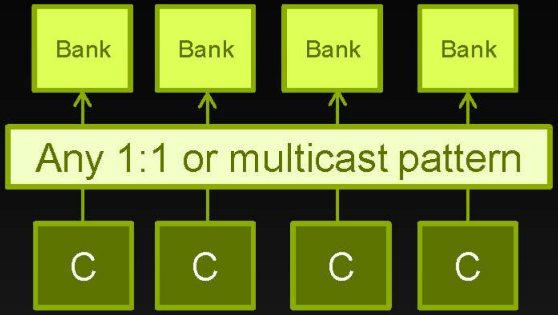

Bank 0 … Bank 31
16KB Shared Memory
32KB
48KB
48KB
32KB
16KB L1 Cache
byte 0 to byte 3 (bank 0, row 0)
byte 4 to byte 7 (bank 1, row 0)
byte 8 to byte 11 (bank 2, row 0)
byte 12 to byte 15 (bank 3, row 0)
byte 128 to byte 131 (bank 0, row 1)
byte 132 to byte 135 (bank 1, row 1)
byte 136 to byte 139 (bank 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

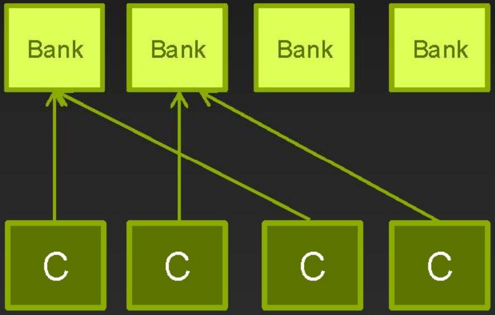
Shared Memory

- 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

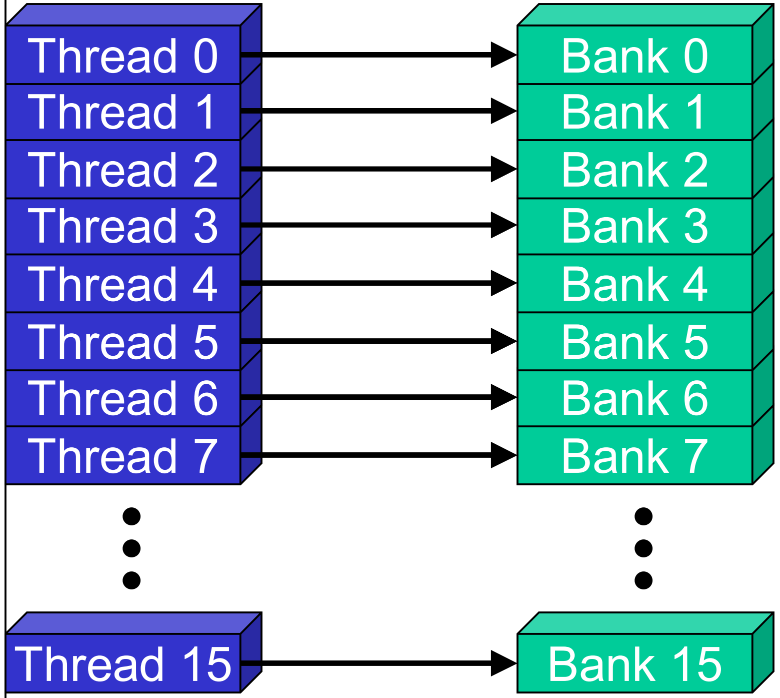
- Organized in 32 independent banks
- Optimal access: no two words from same bank
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize



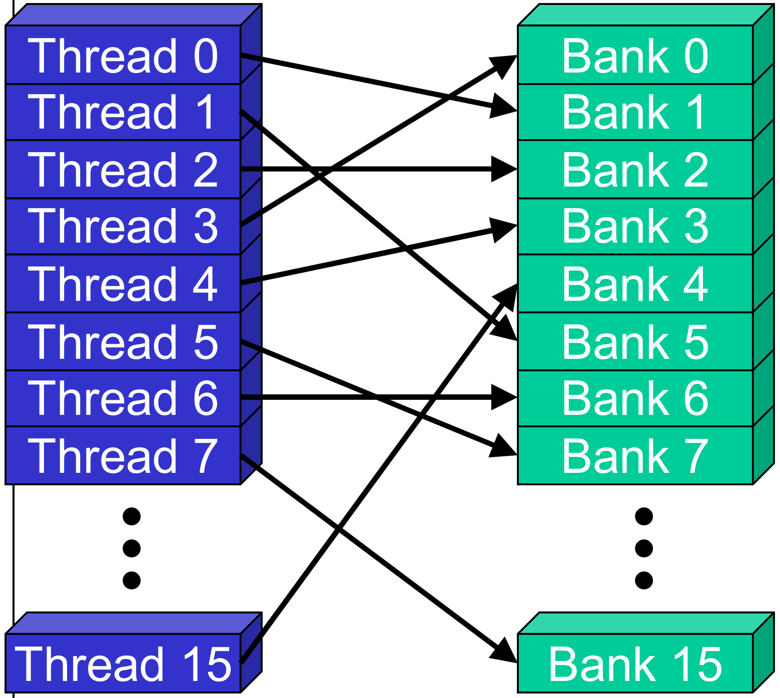


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

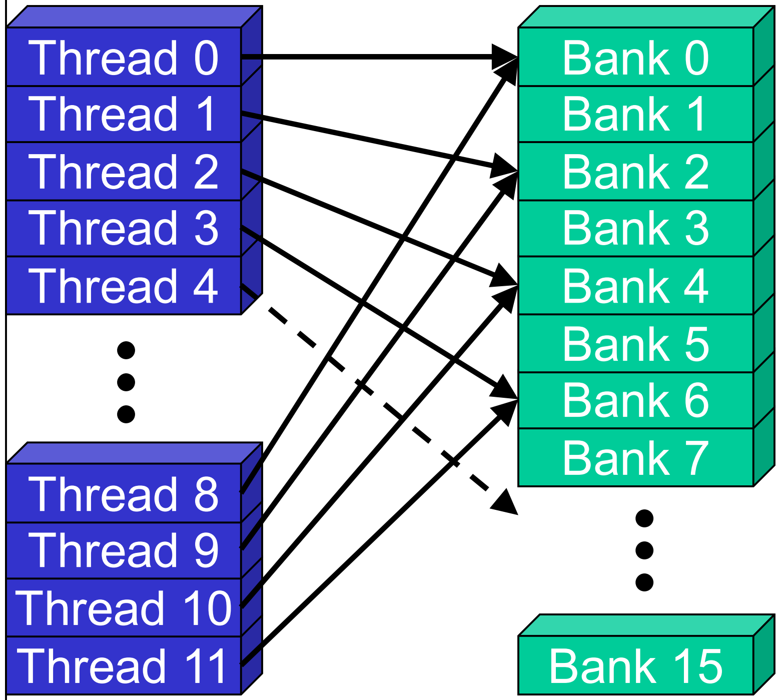


- No Bank Conflicts
  - Random 1:1 Permutation

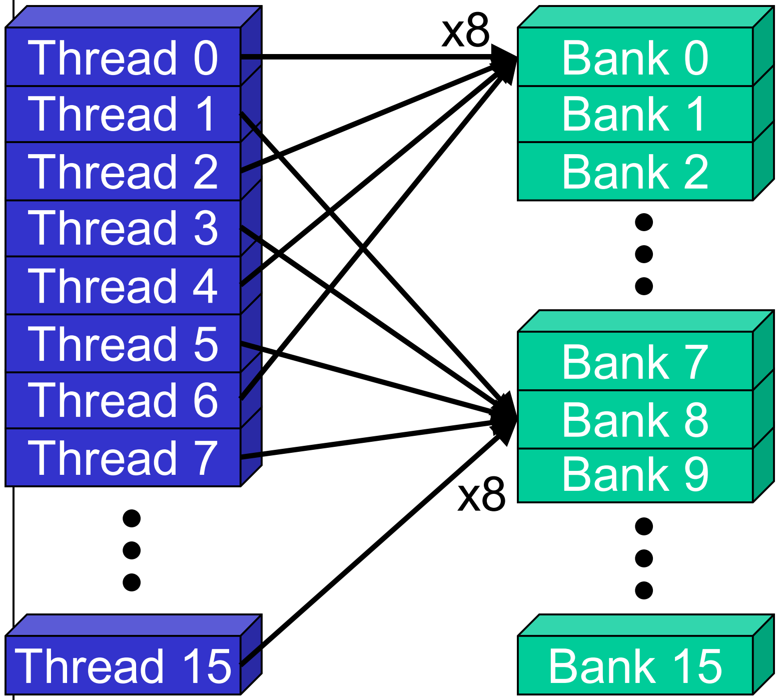


Bank Addressing Examples

- 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

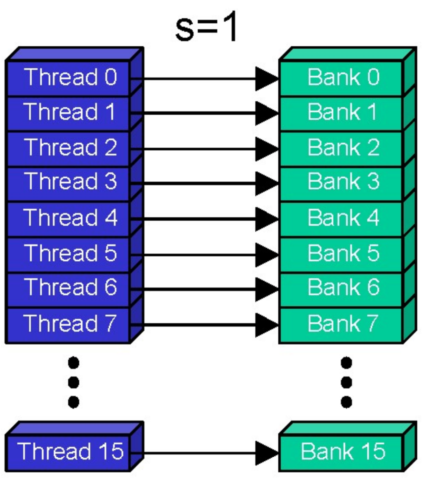
full warps instead of half warps on Fermi and newer!

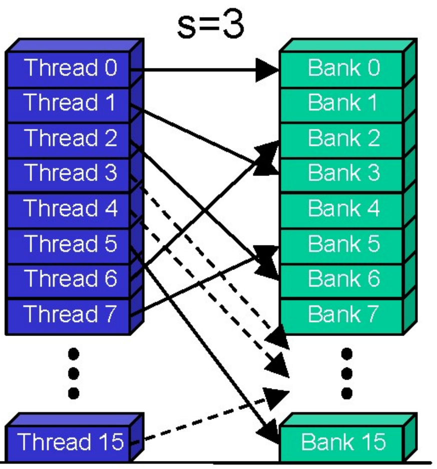
Linear Addressing

- **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

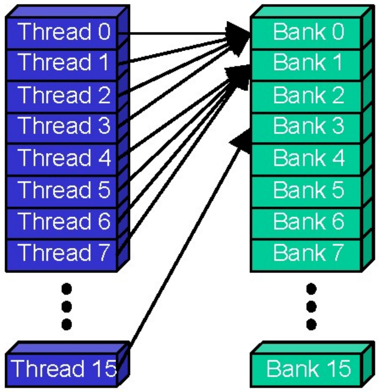
- **This has no conflicts if type of `shared` is 32-bits:**

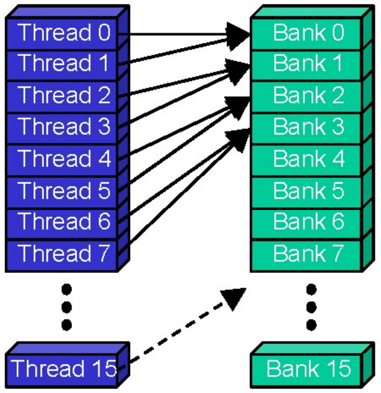
```
foo = shared[baseIndex + threadIdx.x]
```

- **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:

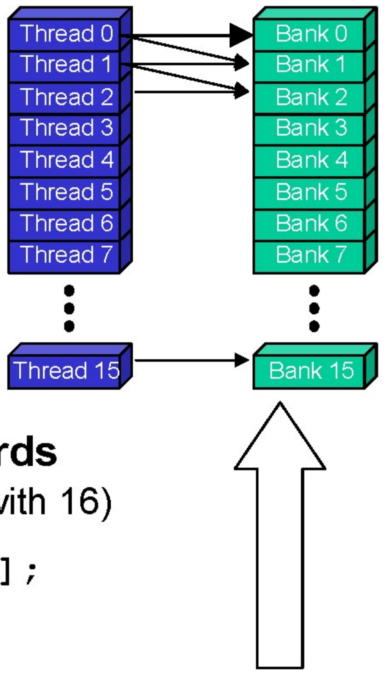
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```

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];
```

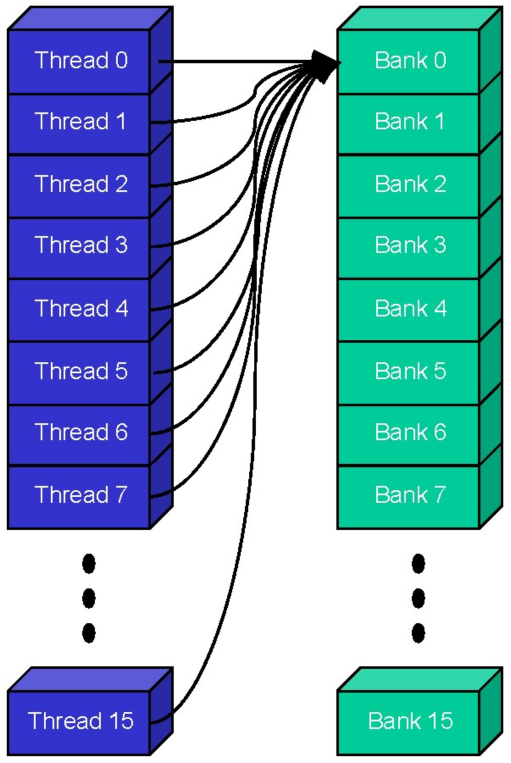
Broadcast on Shared Memory

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

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

- **Will be resolved implicitly**

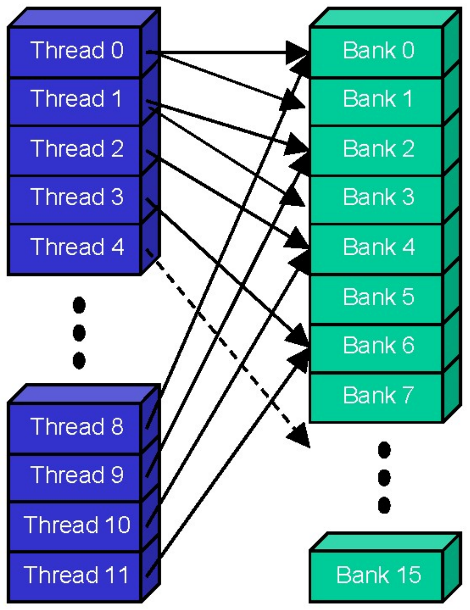
multi-cast on Fermi and newer!



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];
```

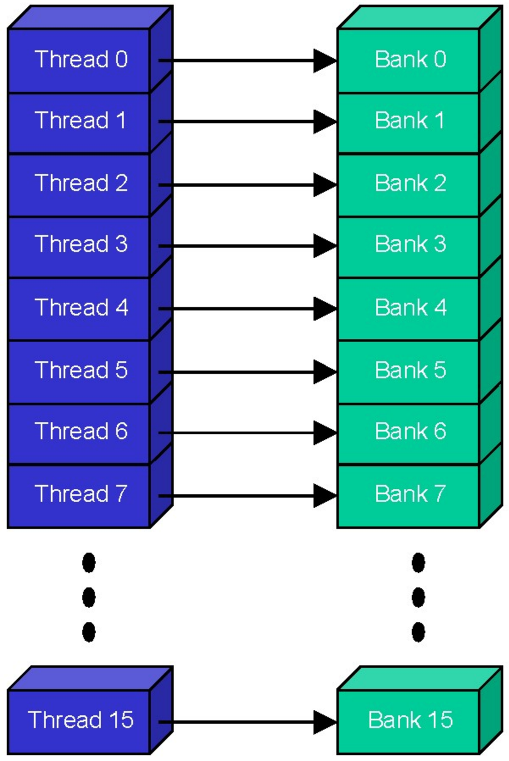


- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects

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

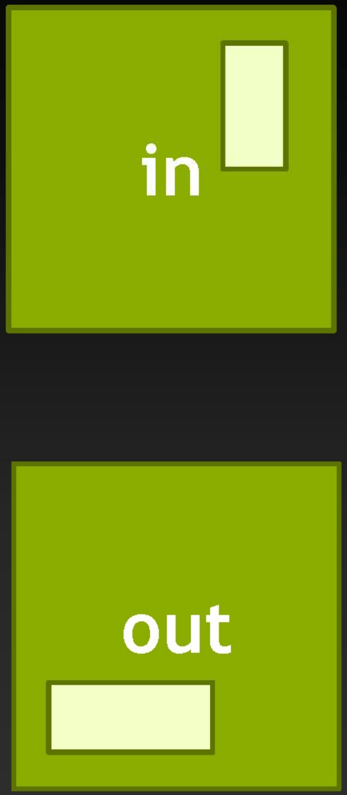
Kernel Optimizations: *Shared Memory Accesses*

Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)

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

⇒ Transpose matrix tile within block

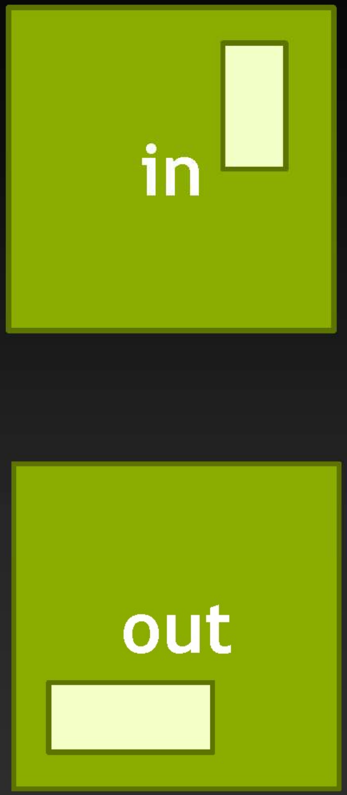


Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)

- Transpose matrix tile within block

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



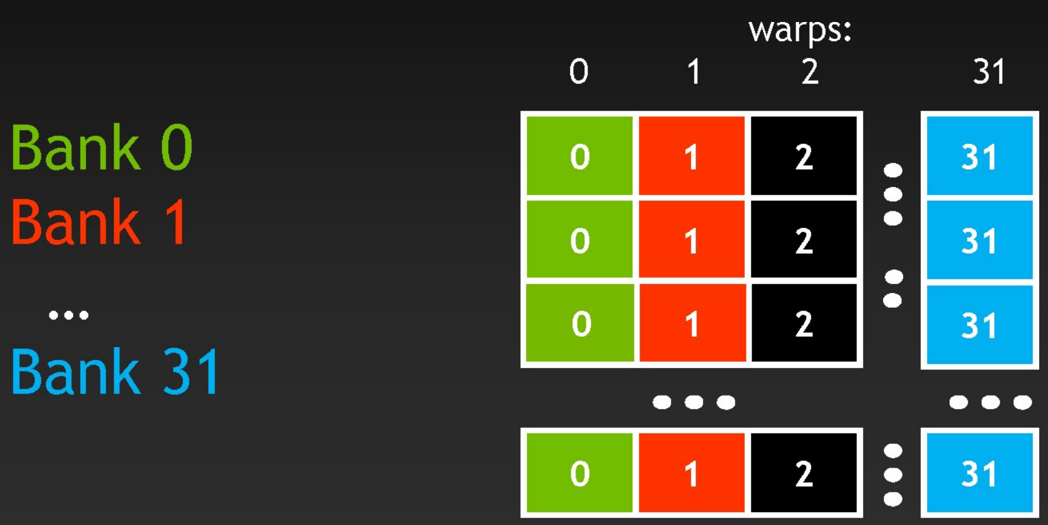
Transpose with coalesced read/write

```
__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

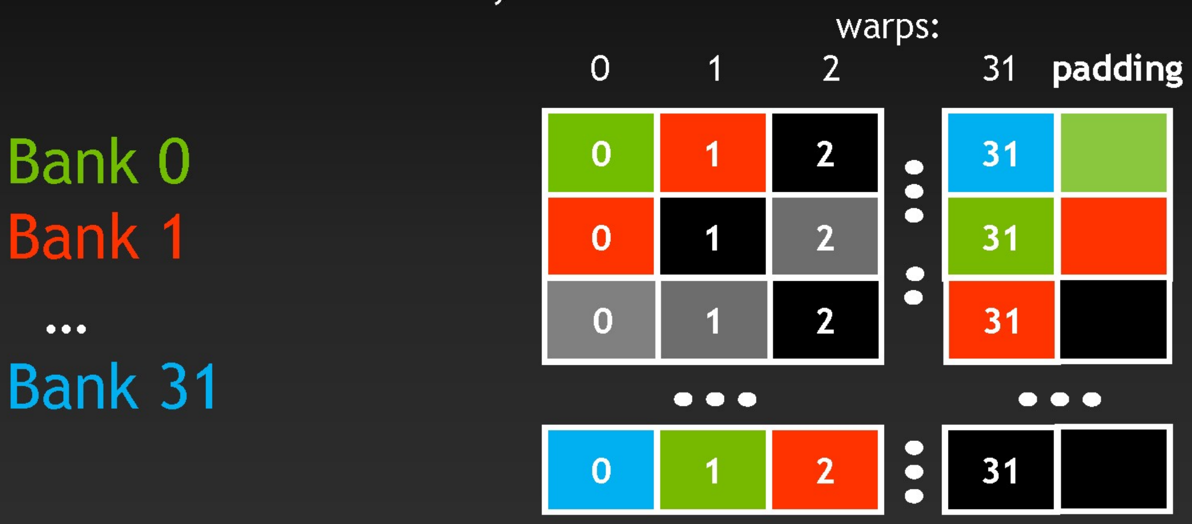
Shared Memory: Avoiding Bank Conflicts

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



Shared Memory: Avoiding Bank Conflicts

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



Thank you.

- Hendrik Lensch, Robert Strzodka
- NVIDIA