CS 380 - GPU and GPGPU Programming
Lecture 24: CUDA Memory, Pt. 3

Markus Hadwiger, KAUST

Reading Assignment #13 (until Nov 27)

Read (required):

- Optimizing Parallel Reduction in CUDA, Mark Harris,

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

- Programming Massively Parallel Processors book, 4th edition
  Chapter 8: Prefix Sum (Scan) – work efficiency in parallel algorithms

Read (optional):

- Programming Massively Parallel Processors book, 4th edition
  Chapter 7: Reduction
- GPU Gems 3 book, Chapter 39: Parallel Prefix Sum (Scan) with CUDA

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

- Faster Parallel Reductions on Kepler, Justin Luitjens

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

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

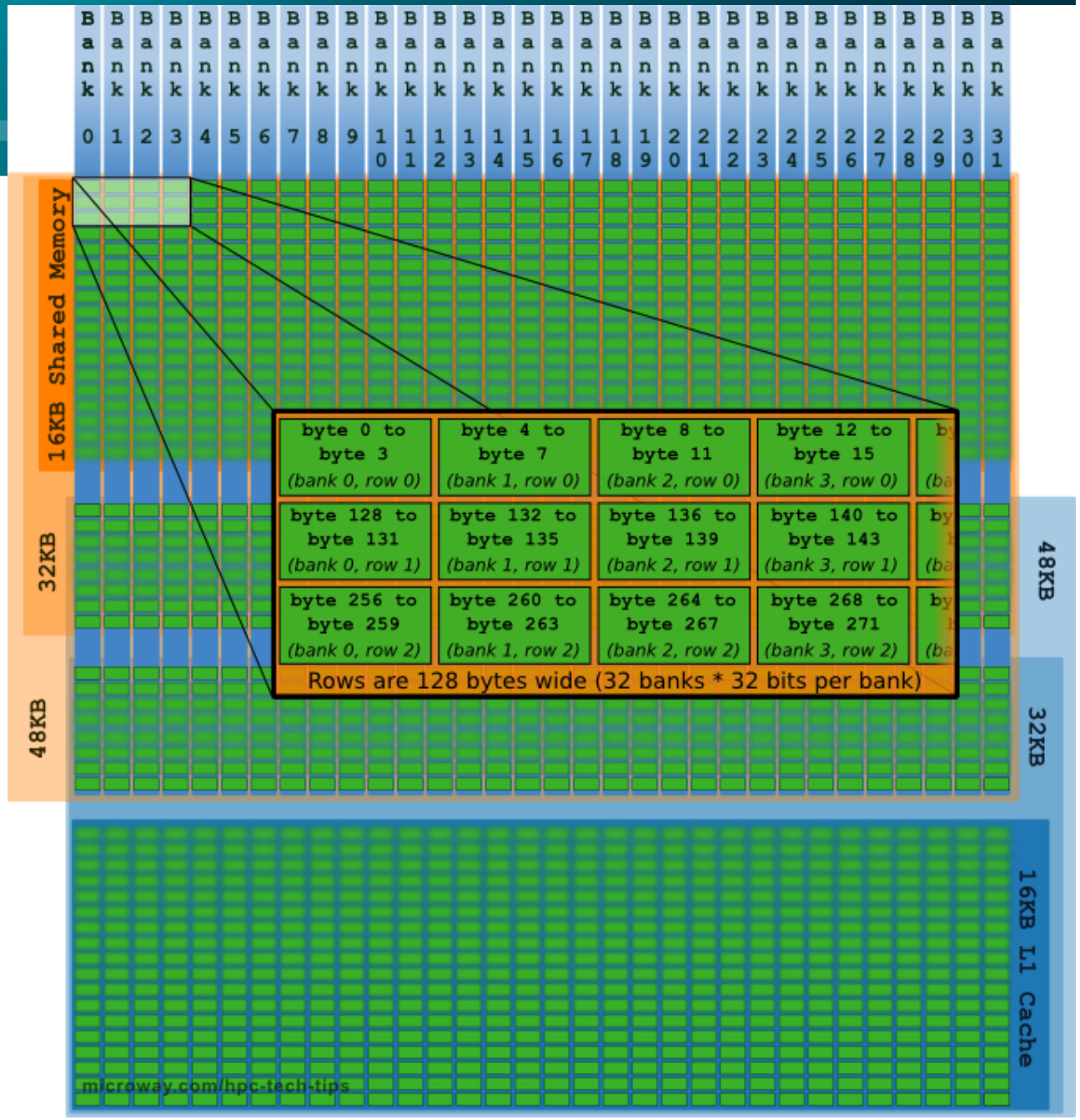
Memory Banks

Fermi/Kepler/Maxwell and newer:

32 banks

default:
4B / bank

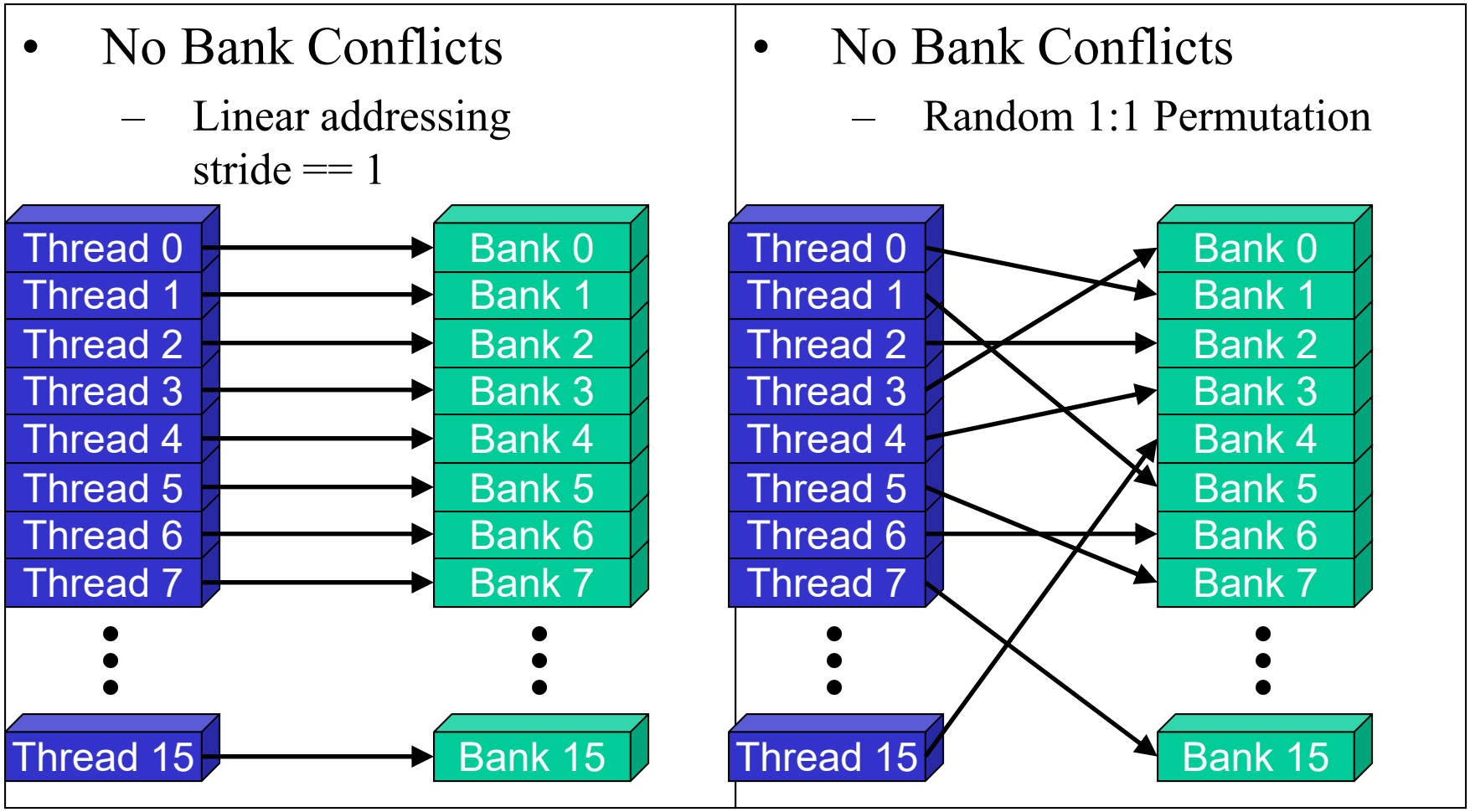
Kepler or newer:
configurable
to 8B / bank



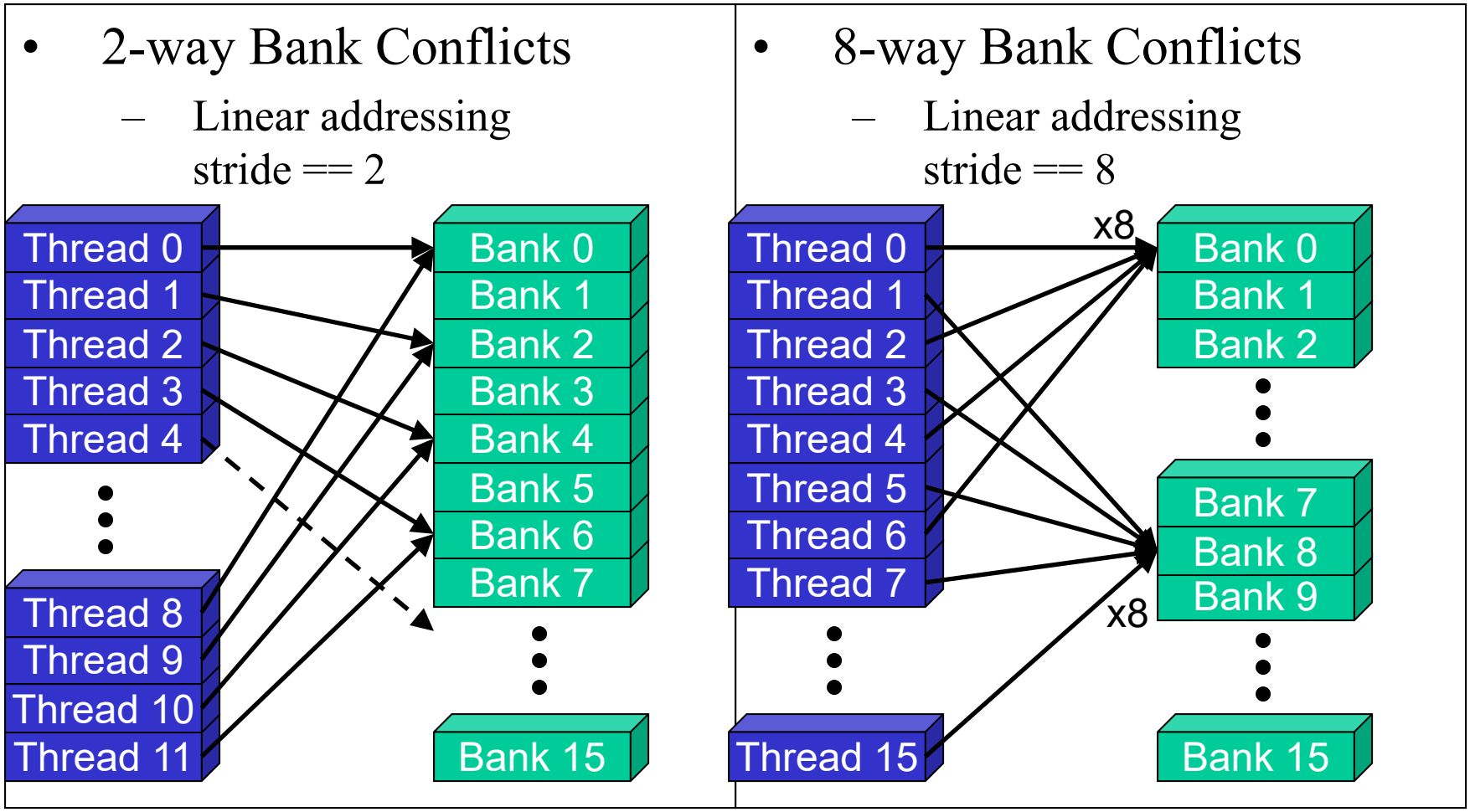
Shared Memory

- 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

Bank Addressing Examples



Bank Addressing Examples



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

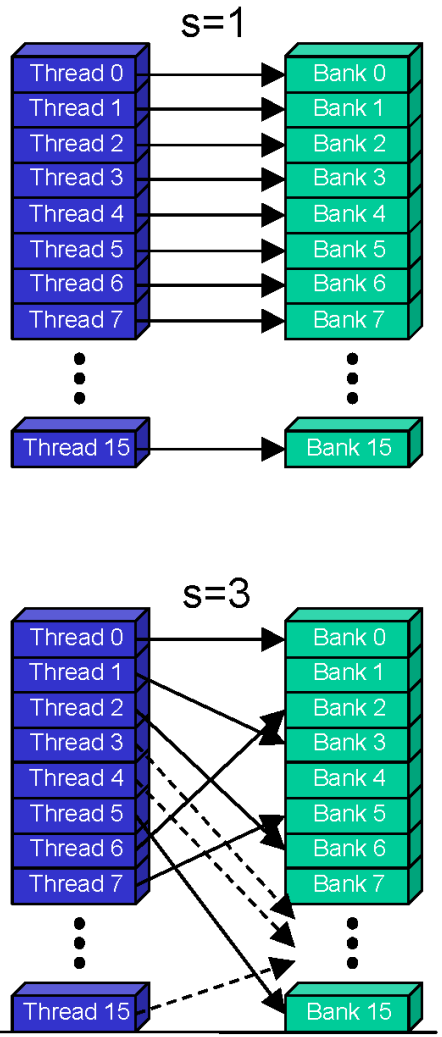
full warps instead of half warps on Fermi and newer!

Linear Addressing

- **Given:**

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

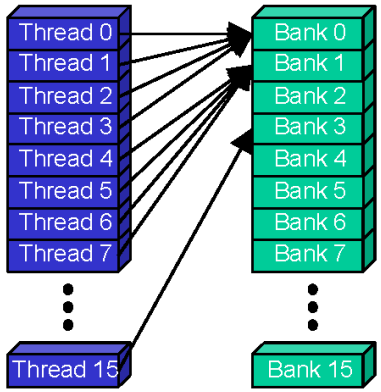
- **This is only bank-conflict-free if s shares no common factors with the number of banks**
  - 16 on G80, so `s` must be odd



Data Types and Bank Conflicts

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

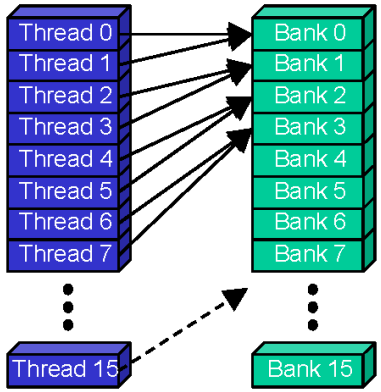
```
foo = shared[baseIndex + threadIdx.x]
```



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

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

but no problem on Fermi or newer: multi-cast!



  - 2-way bank conflicts:

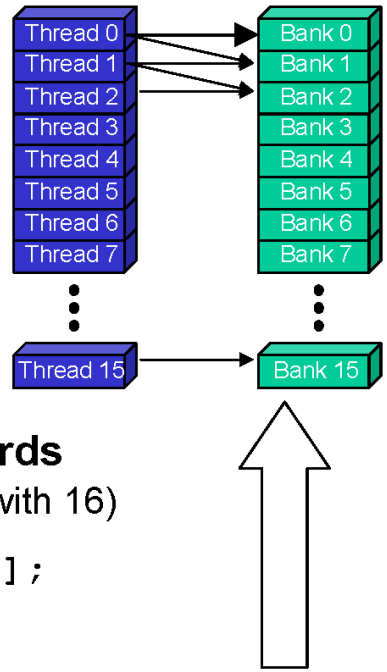
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```

but no problem on Fermi or newer: 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];
```

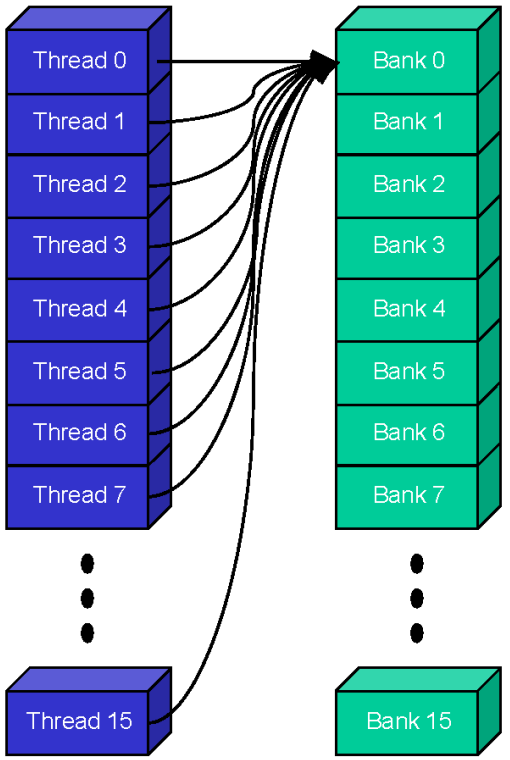
Broadcast on Shared Memory

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

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

- **Will be resolved implicitly**

multi-cast on Fermi and newer!

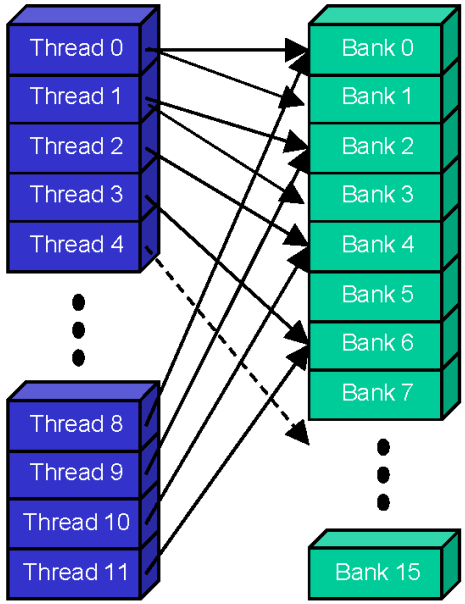


Common Array Bank Conflict Patterns 1D

- **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];
```

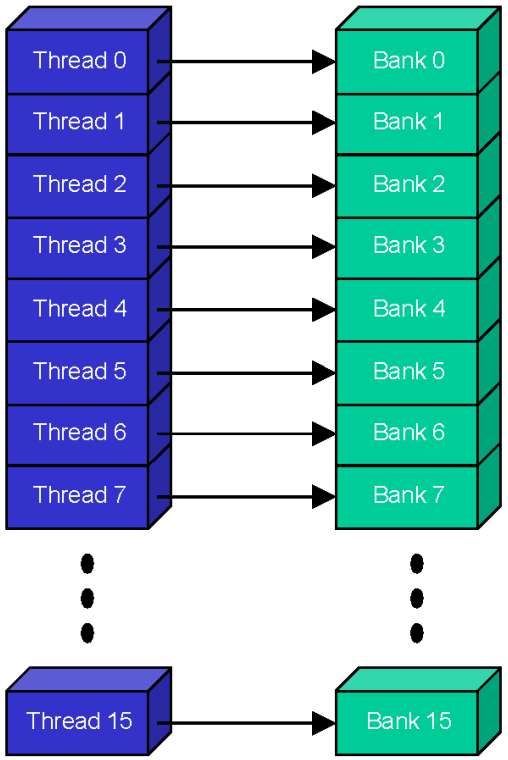
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects



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];

}
```

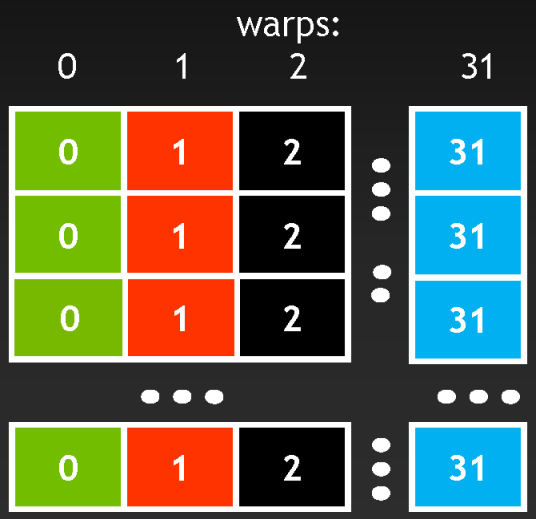
Fixed GMEM coalescing, but introduced SMEM bank conflicts

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

Shared Memory: Avoiding Bank Conflicts

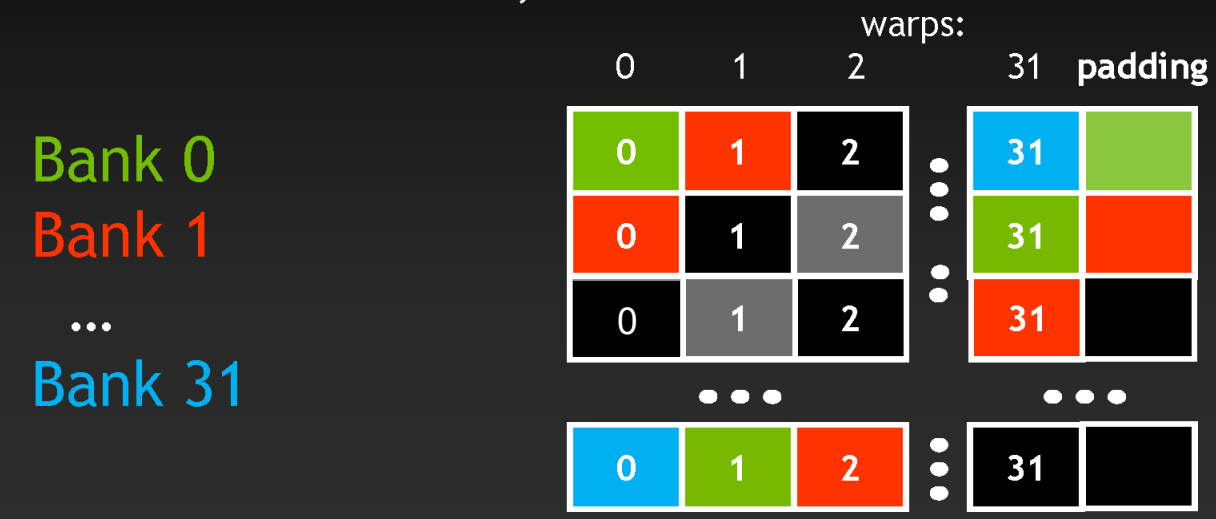
- Example: 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)

Bank 0
Bank 1
...
Bank 31



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.