King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 18: CUDA Memory Access 2

Markus Hadwiger, KAUST

Quiz #4: Apr. 10

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assigments
- Programming assignments (algorithms, methods)
- Solve short practical examples

Reading Assignment #9 (until Apr. 14)

Read (required):

- Programming Massively Parallel Processors book,
  Chapter 5 (*CUDA Memories*)

- **CUDA C Programming Guide 5.0**
  Appendix F: Compute Capabilities

  Study the different memory access requirements for different compute capabilities

Programming Assignments: Schedule

Assignment #3:

- Image Processing with (a) GLSL, and (b) CUDA — due Apr 7

Assignment #4:

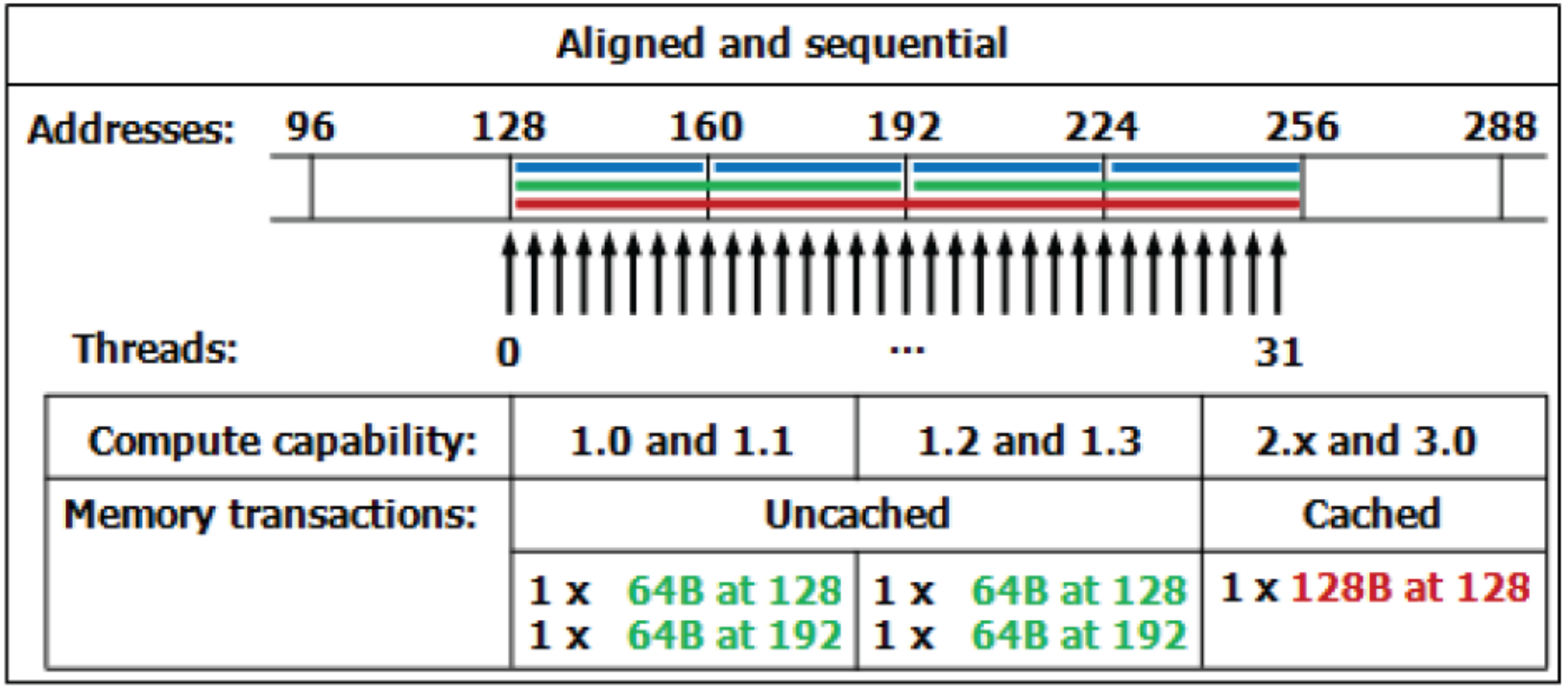
- Conjugate Gradient Linear Systems Solver (CUDA) — due Apr 28

Memory Access Patterns

1. Global Memory Accesses

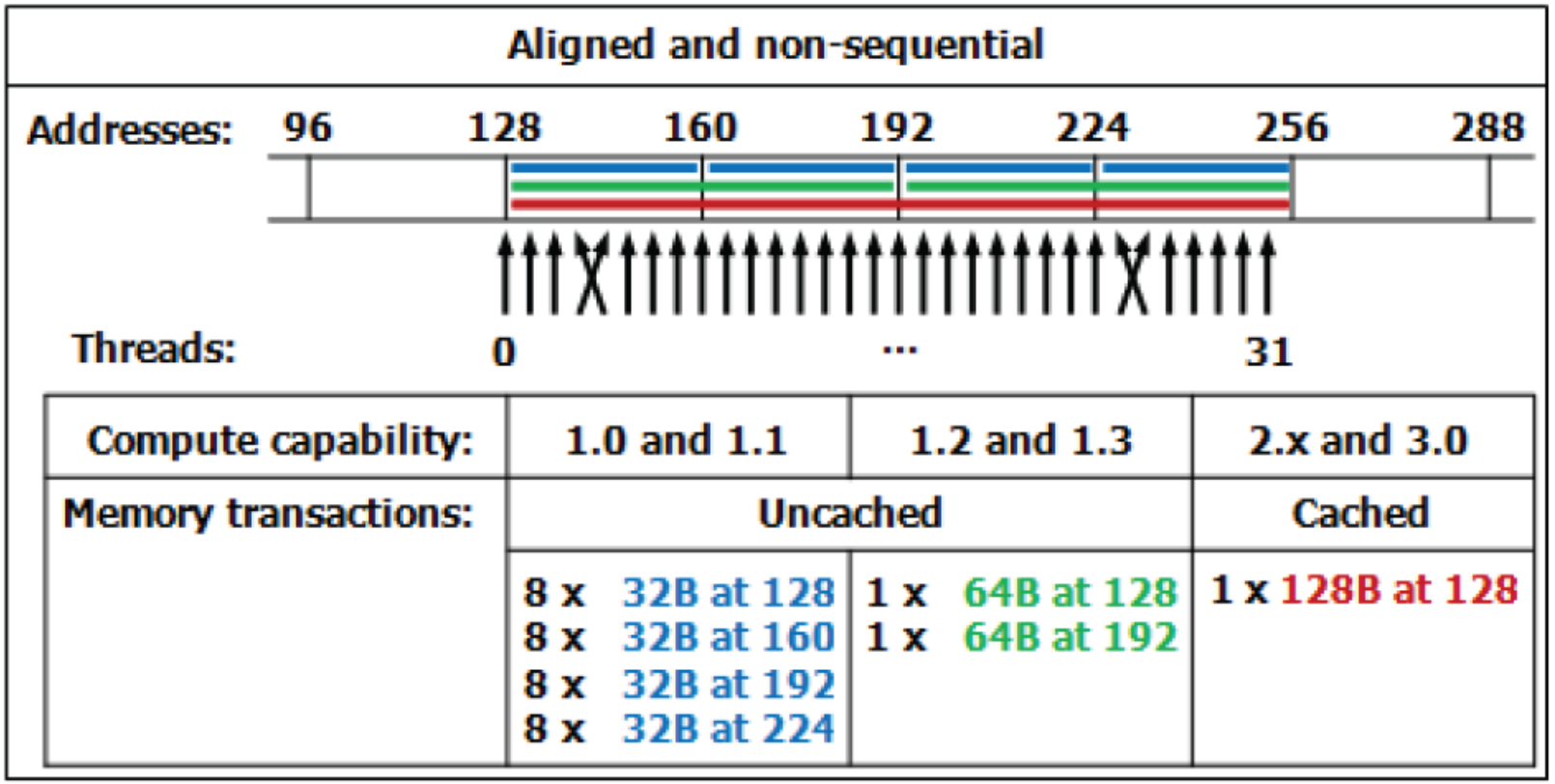
- Memory coalescing
- Cached memory access

Global Memory Access Arch. Differences (1)



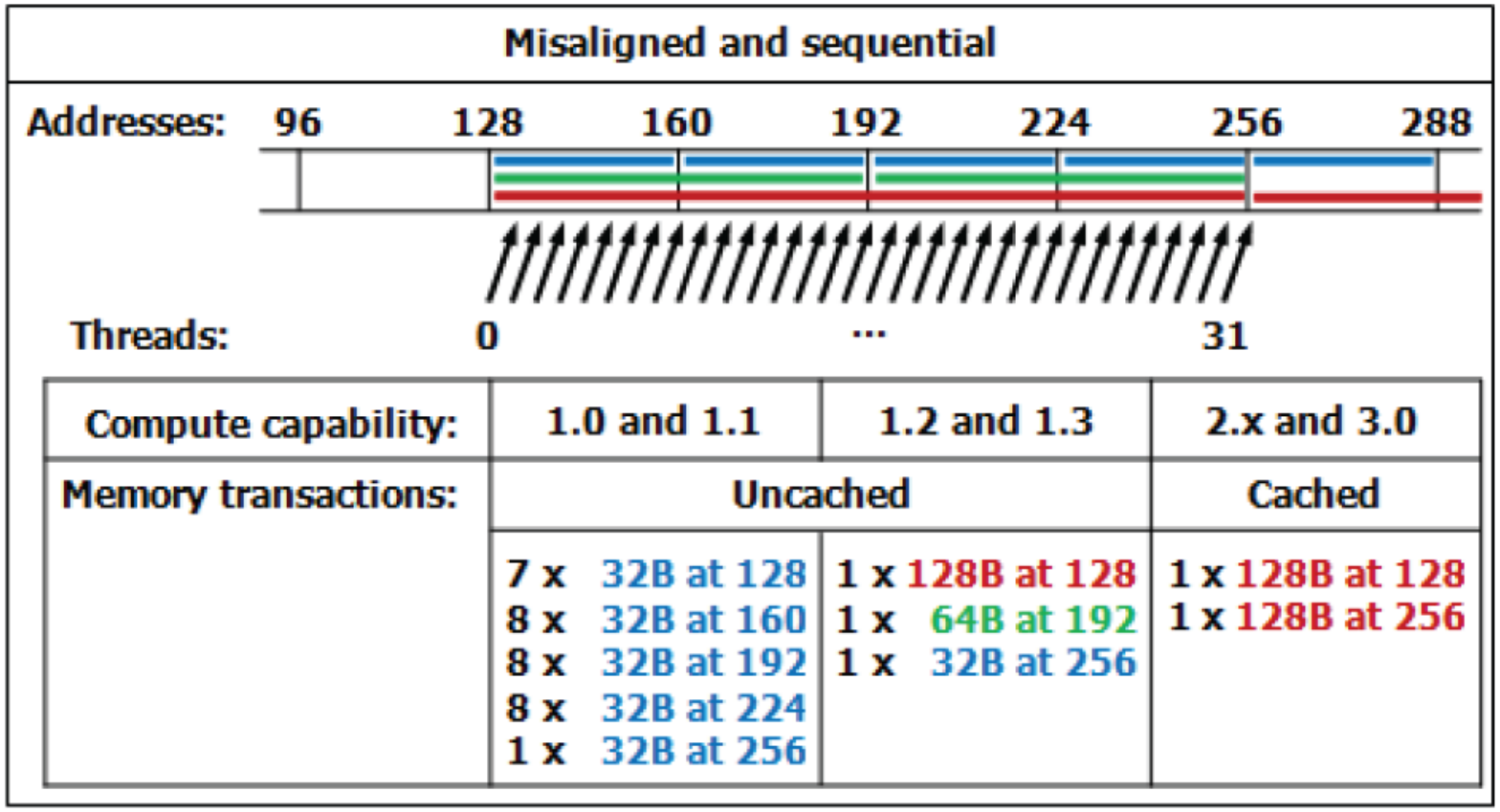
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Global Memory Access Arch. Differences (2)



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Global Memory Access Arch. Differences (3)



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

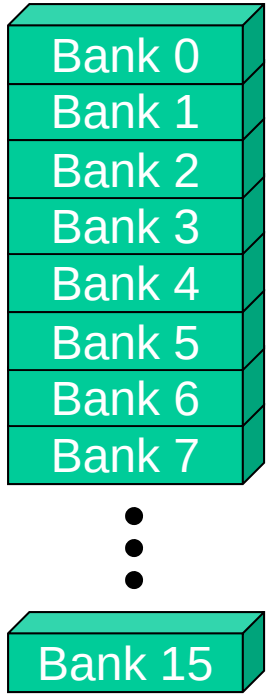
Memory Access Patterns

2. Shared Memory Accesses

- Banked memory access / bank conflicts

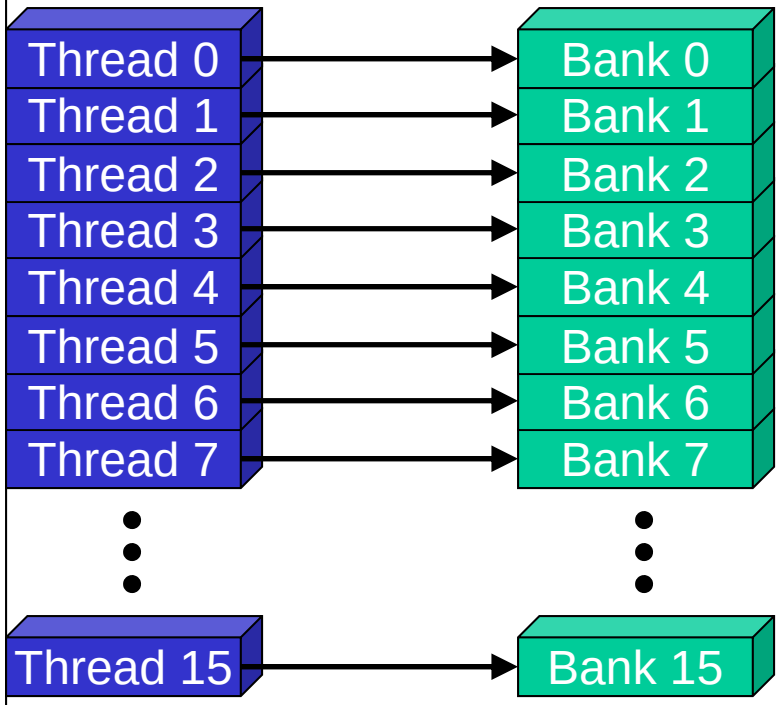
Parallel Memory Architecture

- 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

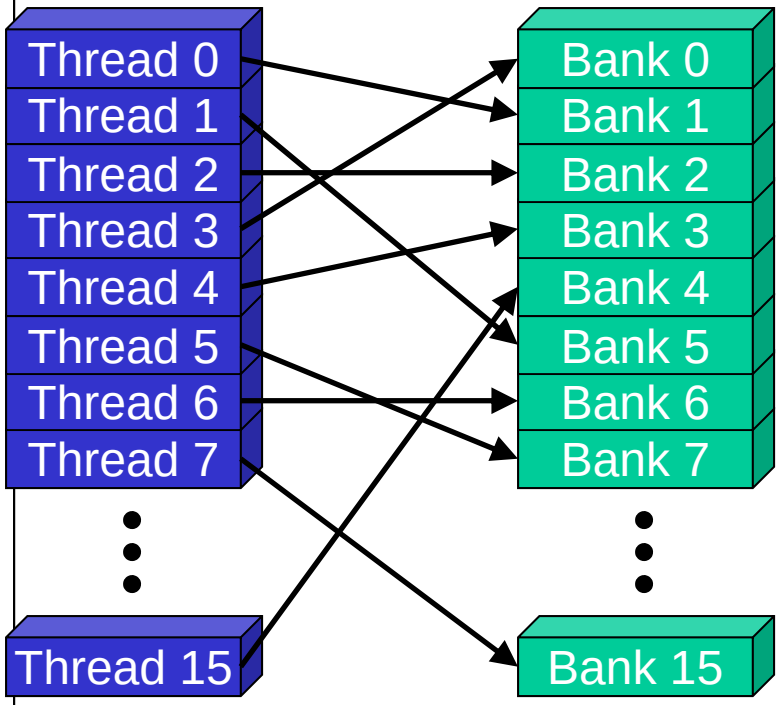


Bank Addressing Examples

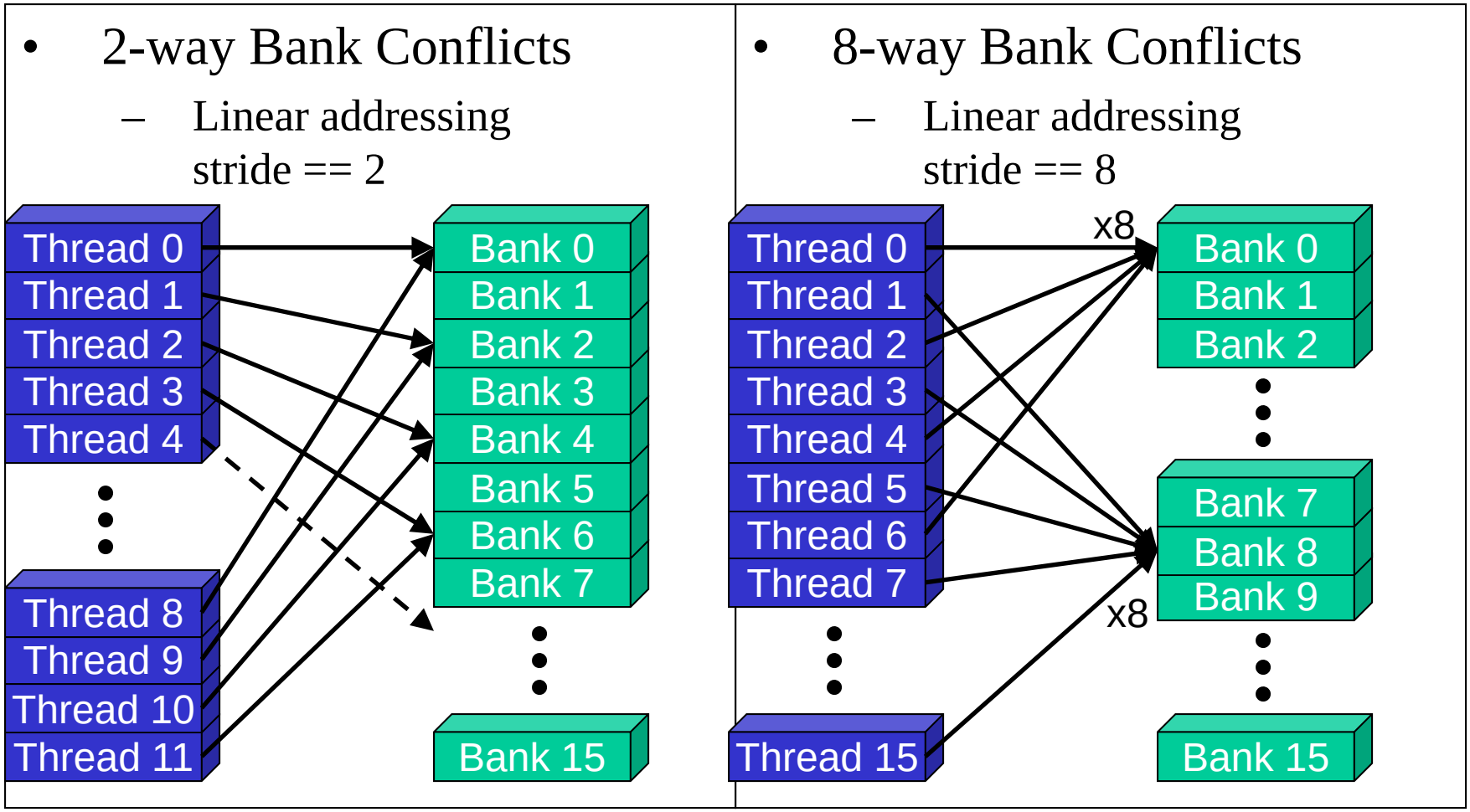
- No Bank Conflicts
  - Linear addressing stride == 1



- No Bank Conflicts
  - Random 1:1 Permutation



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 has 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

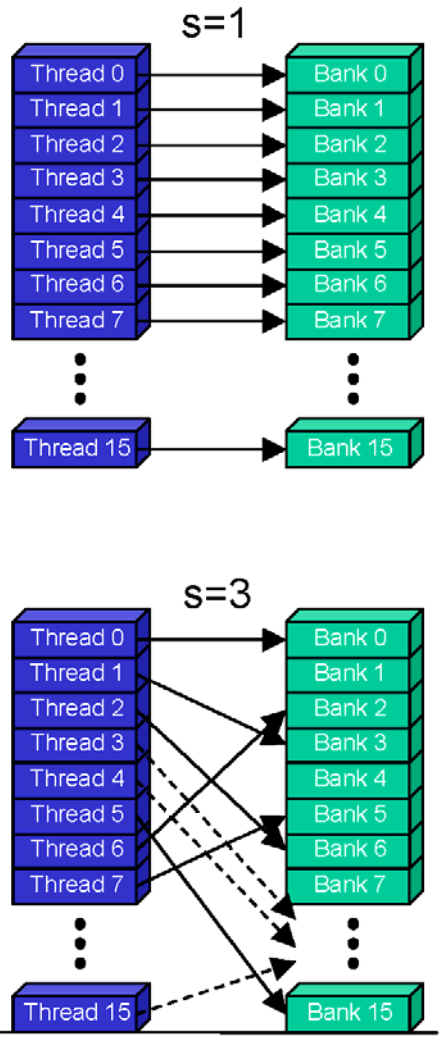
full warps instead of half warps on Fermi!

Linear Addressing

- **Given:**

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

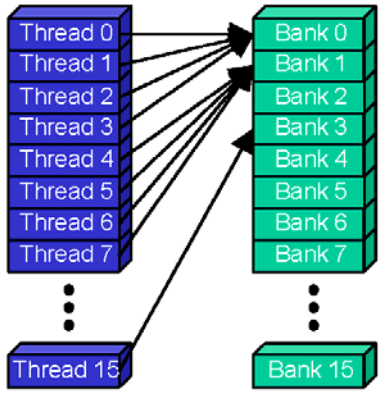
- **This is only bank-conflict-free if s shares no common factors with the number of banks**
  - 16 on G80, so `s` must be odd



Data Types and Bank Conflicts

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

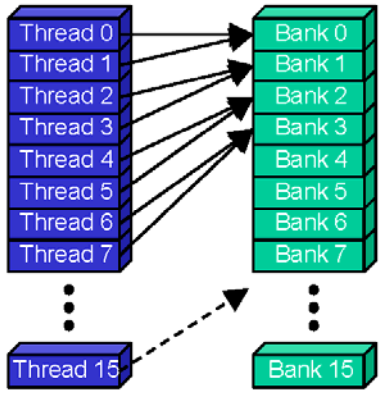
```
foo = shared[baseIndex + threadIdx.x]
```



- **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:

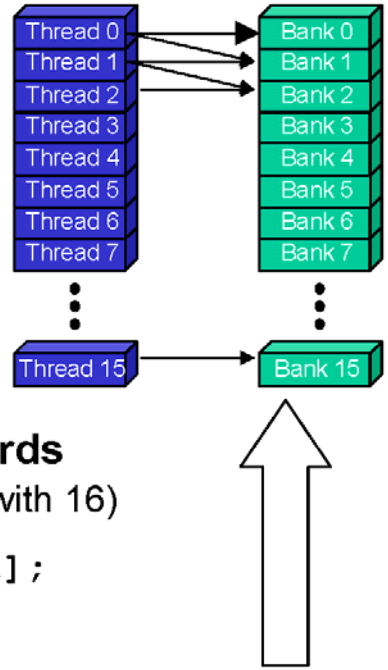
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```

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];
```

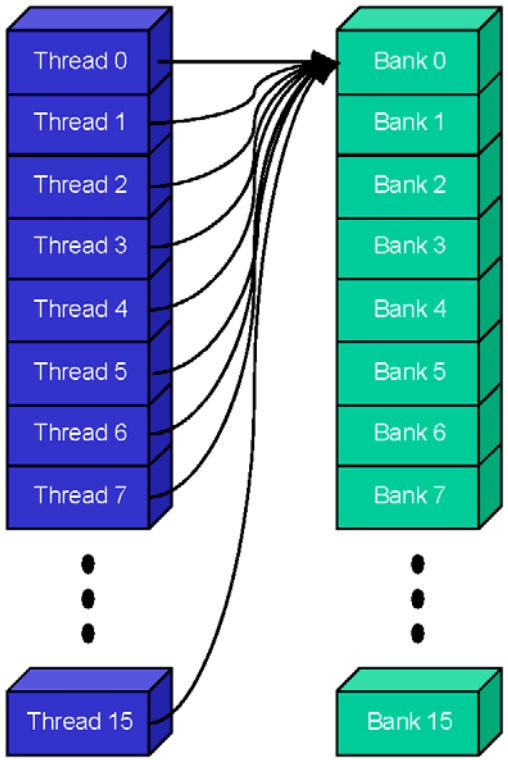
Broadcast on Shared Memory

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

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

- **Will be resolved implicitly**

multi-cast on Fermi!

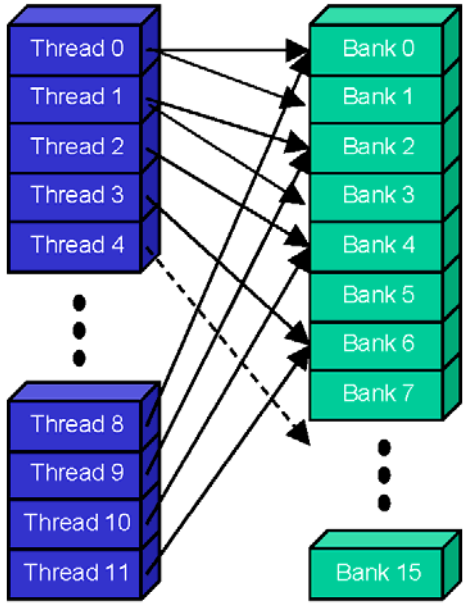


Common Array Bank Conflict Patterns 1D

- **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];
```

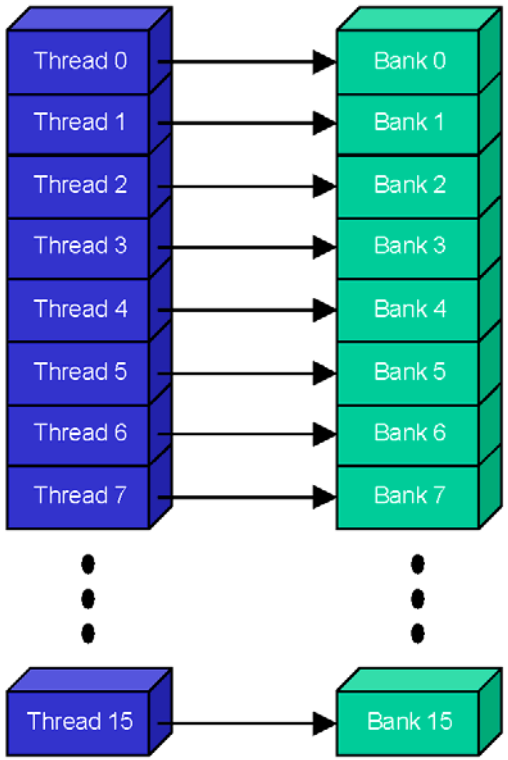
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects



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];
```



Thank you.

- Hendrik Lensch, Robert Strzodka