Memory management for performance

Ramani Duraiswami

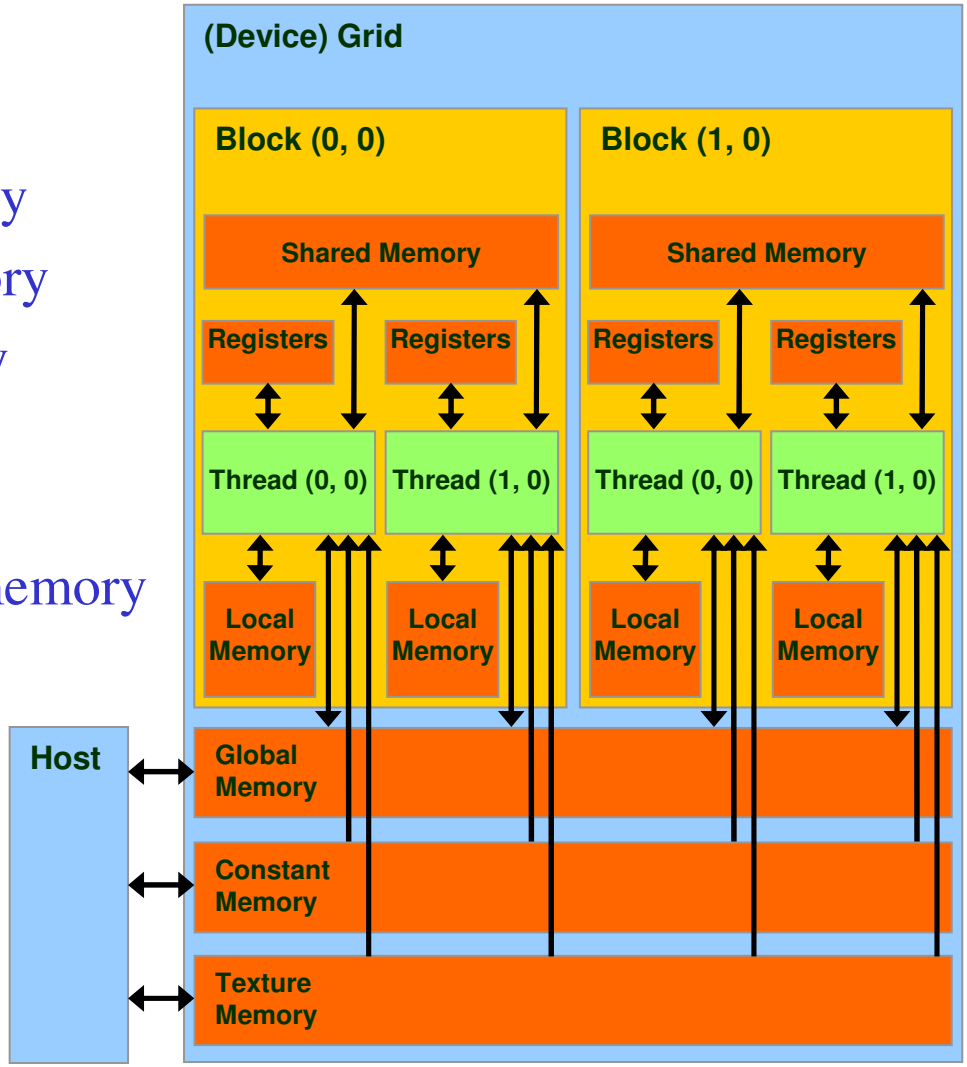
Several slides from Wen-Mei Hwu and David Kirk's course

Hierarchical Organization

- GPU -> Grids
  - Multiprocessors -> Blocks, Warps
    - Thread Processor -> Threads
- Global Memory
  - Shared Memory
    - Registers
  - Cached global memory (Texture and Constant)
- Parallel programming
  - dividing job in to pieces correctly and optimize the utilizing of available processing and memory resources
  - Do it correctly so that
    - Use results only when computations are complete
    - No simultaneous writes, etc.
- Avoid serialization
  - If compiler cannot determine if things are correct it will serialize
  - Can also happen because of hardware limitations
- Two techniques
  - Coalescing -> global memory reads and writes
  - Bank Conflicts -> multiprocessors accesses to shared memory

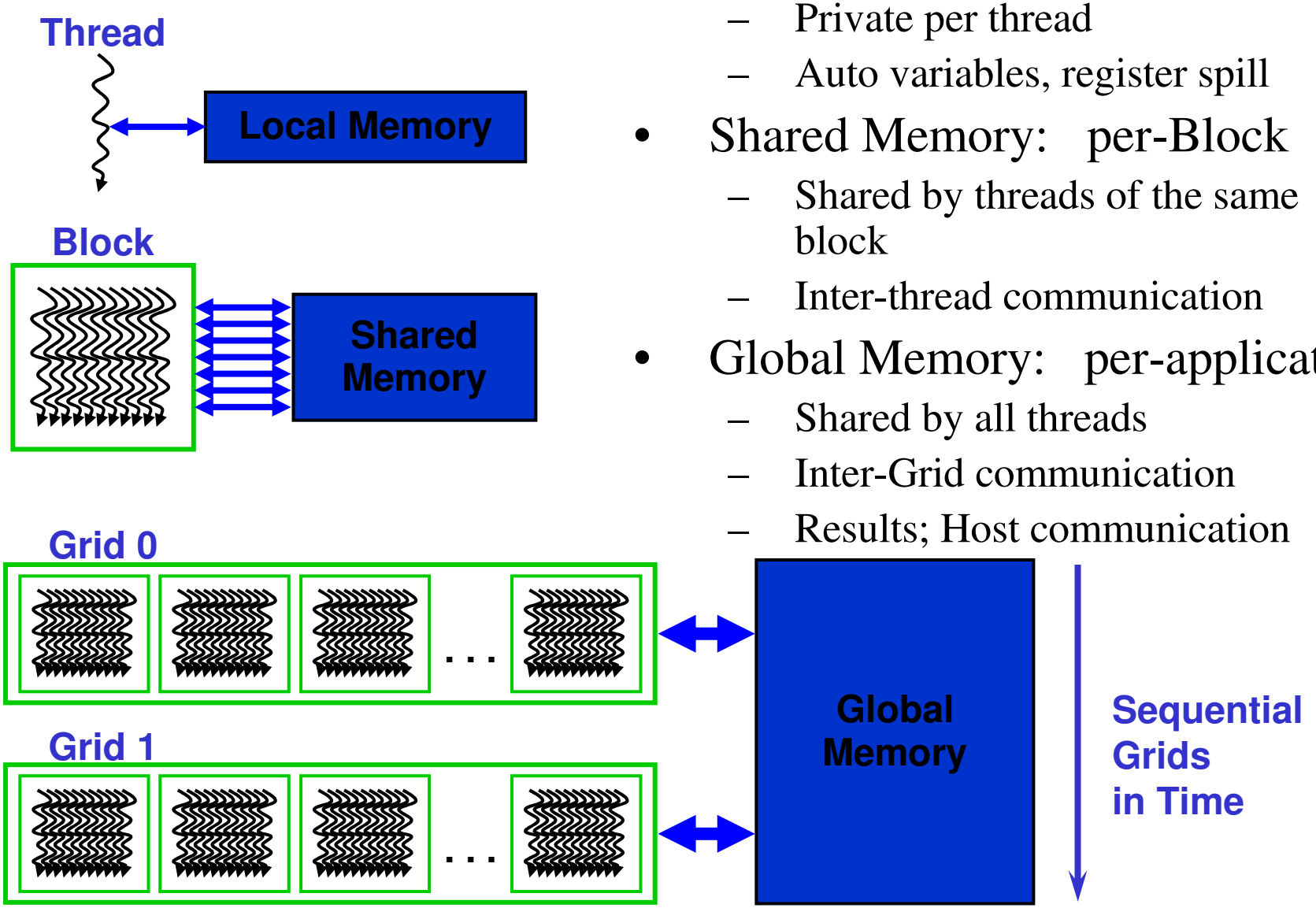
CUDA Device Memory

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories

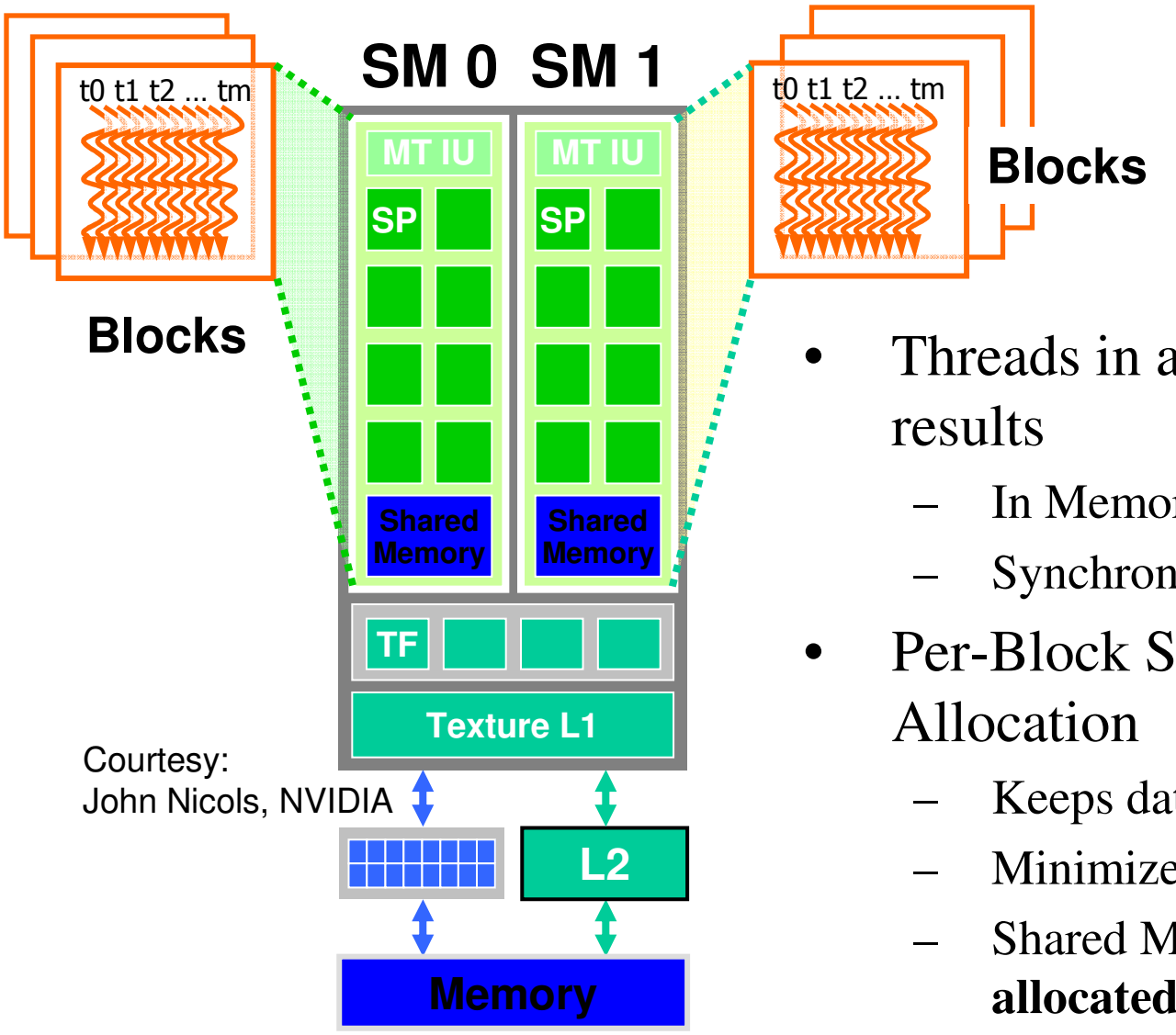


Parallel Memory Sharing

- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication
  - Results; Host communication



SM Memory Architecture

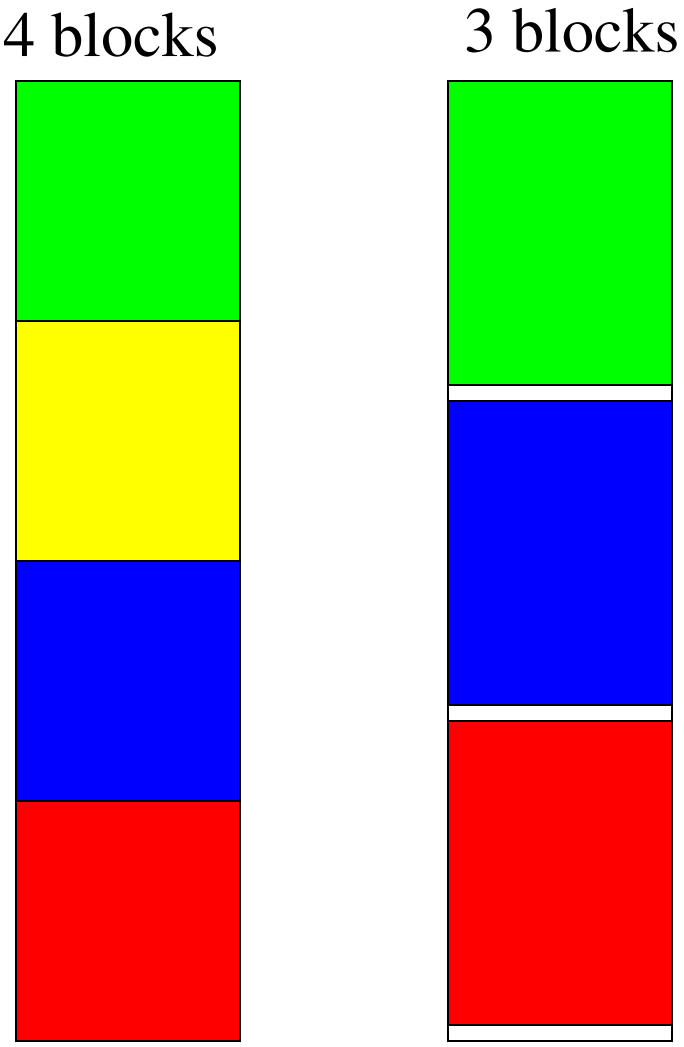


Courtesy:
John Nicols, NVIDIA

- Threads in a block share data & results
  - In Memory and Shared Memory
  - Synchronize at barrier instruction
- Per-Block Shared Memory Allocation
  - Keeps data close to processor
  - Minimize trips to global Memory
  - Shared Memory is **dynamically allocated** to blocks, one of the limiting resources

Programmer View of Register File

- Register File (RF)
  - 32 KB (8K entries) for each SM in G80 (Compute Capability 1.0-1.1)
  - (Compute Capability 1.2 has 16384 registers)
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all blocks assigned to the multiprocessor
  - Once assigned to a block, the register is NOT accessible by threads in other blocks
  - Each thread in the same block only accesses registers assigned to itself



Matrix Multiplication Example

- If each Block has 16×16 threads and each thread uses 10 registers, how many threads can run on each SM?
  - Each block requires 10*256 = 2560 registers
  - 8192 = **3** * 2560 + change
  - So, three blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 1?
  - Each Block now requires 11*256 = 2816 registers
  - 8192 < 2816 *3
  - Only two Blocks can now fit on the SM,
  - **1/3 reduction of parallelism**!!!

More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between improving performance via instruction-level parallelism or via thread level parallelism

ILP vs. TLP Example

- Assume that a kernel has
  - 256-thread Blocks,
  - 4 independent instructions for each global memory load in the thread
  - each thread uses 10 registers,
  - global loads take 200 cycles
  - Then => 3 Blocks can run on each SM
- If the compiler can use one more register to change the dependence pattern so that
  - 8 independent instructions can be performed per global memory load
  - Then => Only two blocks can run on each SM
- However, one only needs 200/(8*4) = 7 Warps to tolerate the memory latency
  - Two blocks have 16 Warps. The performance can be actually higher!

Global Memory

- global memory space is not cached
- important to follow the right access pattern to get maximum memory bandwidth
- access is costly – 400-600 clock cycles
- For efficiency the access must result in fewer instructions, and fewer address computations
- Device can read 32-bit, 64-bit, or 128-bit words from global memory into registers in a single instruction.
- Structures and arrays must align on these byte boundaries to minimize number of instructions.

Example

- To ensure that it generates the minimum number of instructions, such structures should be defined with **`__align__(16)`** , such as

```
struct __align__(16) {
float a;
float b;
float c;
float d;
float e;
};
```

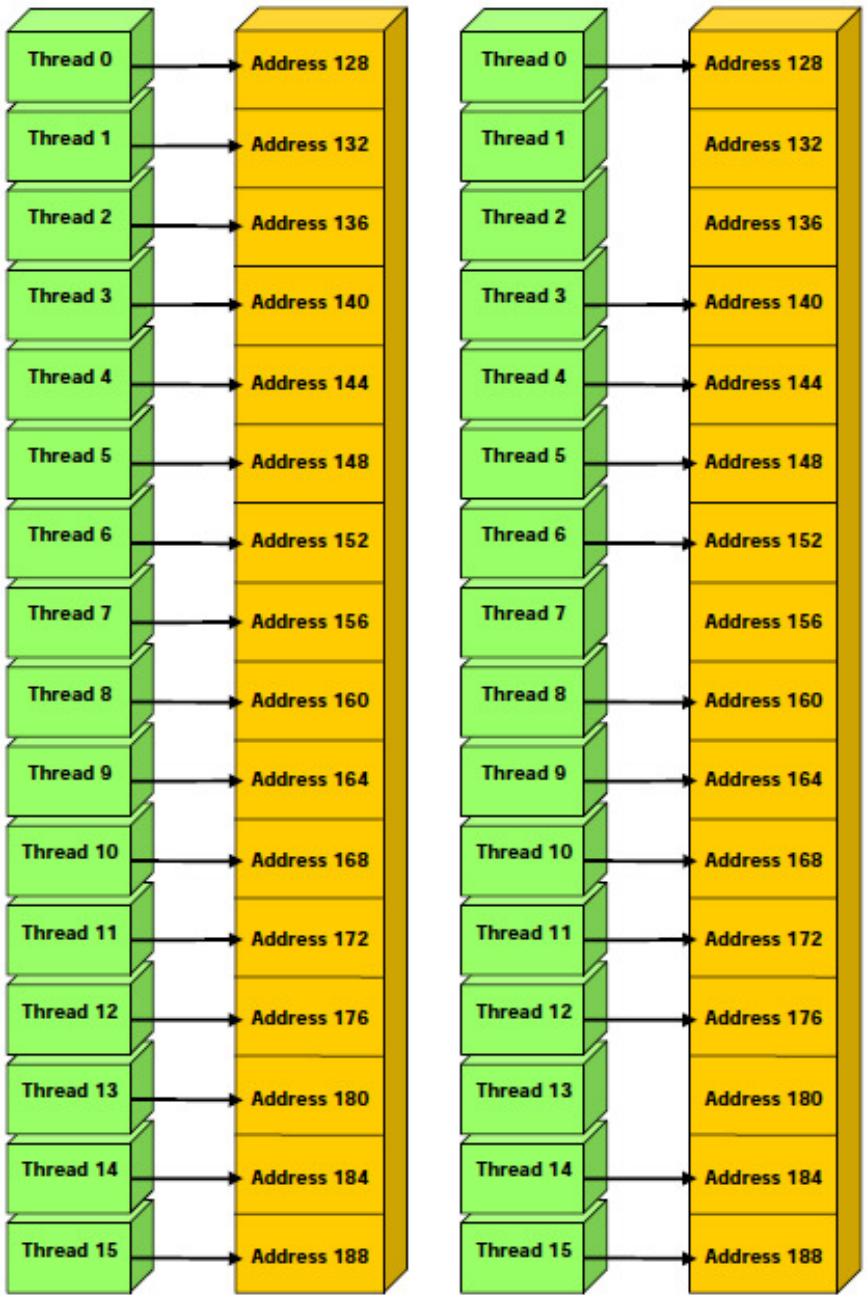
- compiles into two 128-bit load instructions instead of five 32-bit load instructions.

Coalescing accesses

- global memory bandwidth is most efficient when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be *coalesced* into a single memory transaction of 32, 64, or 128 bytes.
- coalescing is achieved even if the warp is divergent and some threads of the half-warp do not actually access memory.
- global memory access by all threads of a half-warp is coalesced into one or two memory transactions if :
  - Either 32-bit words, resulting in one 64-byte memory transaction,
  - Or 64-bit words, resulting in one 128-byte memory transaction,
  - Or 128-bit words, resulting in two 128-byte memory transactions;
  - All 16 words must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words);
  - Threads must access the words in sequence: The *k*th thread in the half-warp must access the *k*th word.
- If a half-warp does not fulfill all the requirements above, a separate memory transaction is issued for each thread and throughput is significantly reduced.

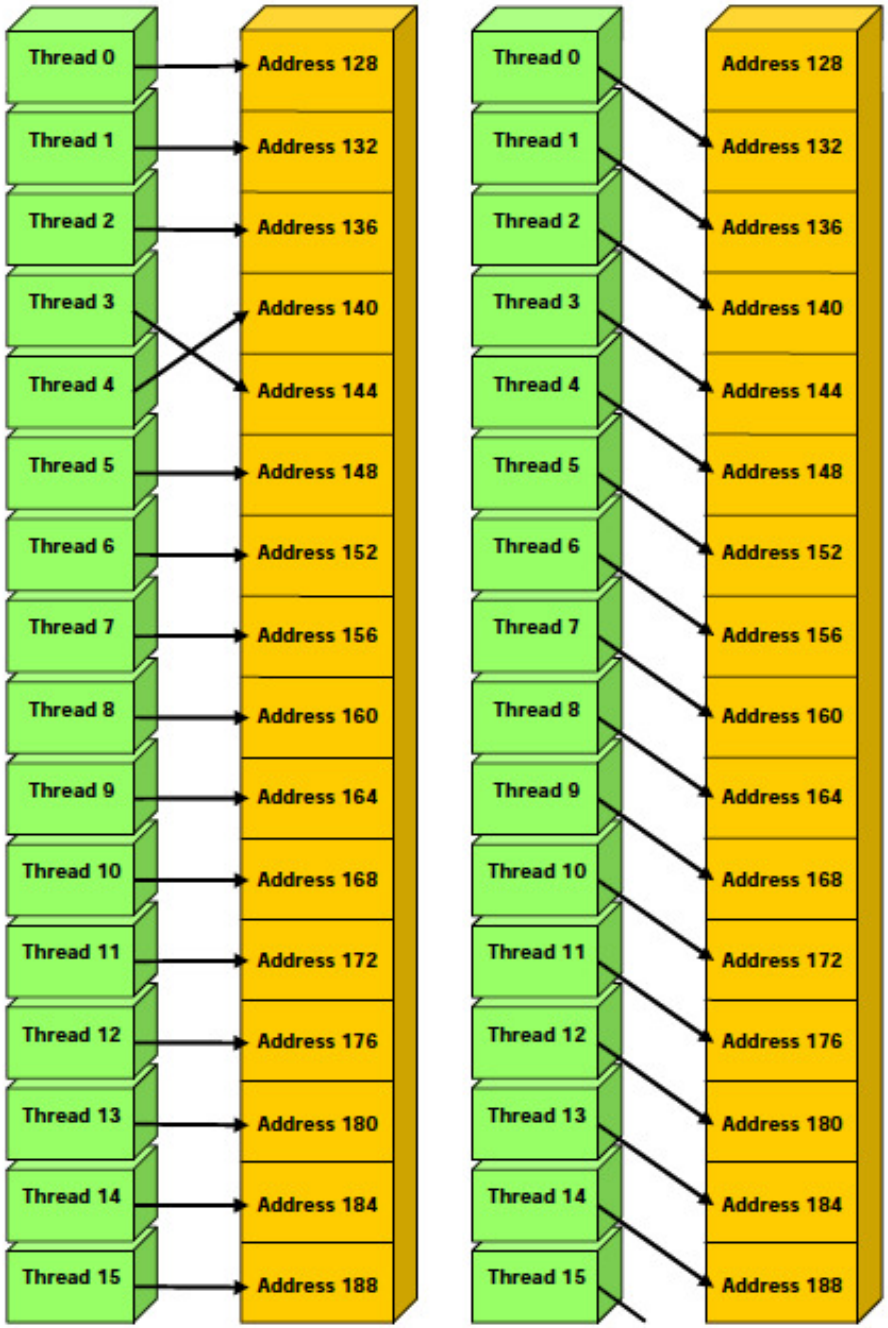
Examples of Coalesced Global Memory Access

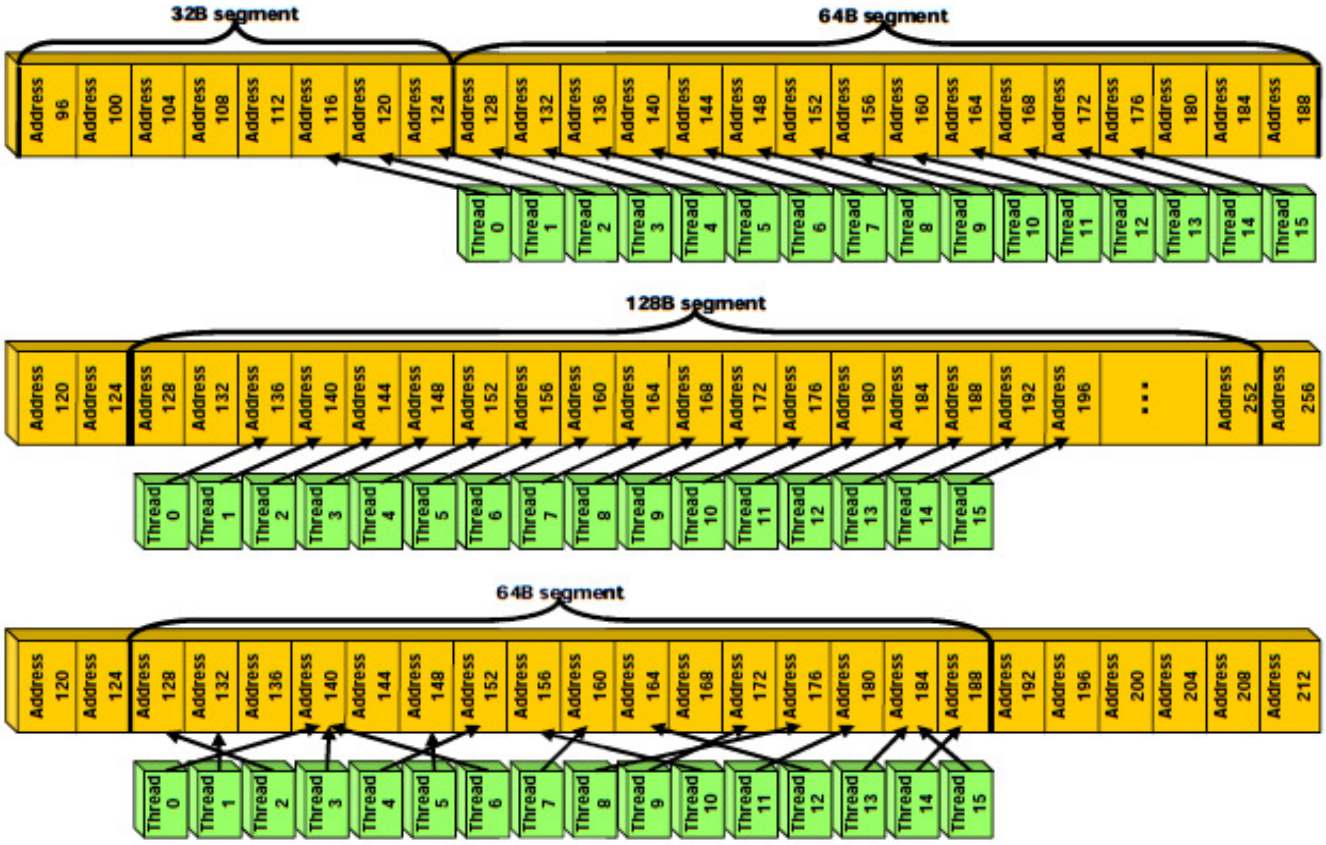
- Left: coalesced **float** memory access, resulting in a single memory transaction.
- Right: coalesced **float** memory access (divergent warp), resulting in a single memory transaction.



Noncoalesced access

- Left: non-sequential **float** memory access, resulting in 16 memory transactions.
- Right: access with a misaligned starting address, resulting in 16 memory transactions.
- In newer machines this is fixed




32B segment
64B segment
Address 96
Address 100
Address 104
Address 108
Address 112
Address 116
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
128B segment
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
...
Address 252
Address 256
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
64B segment
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 208
Address 212
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15

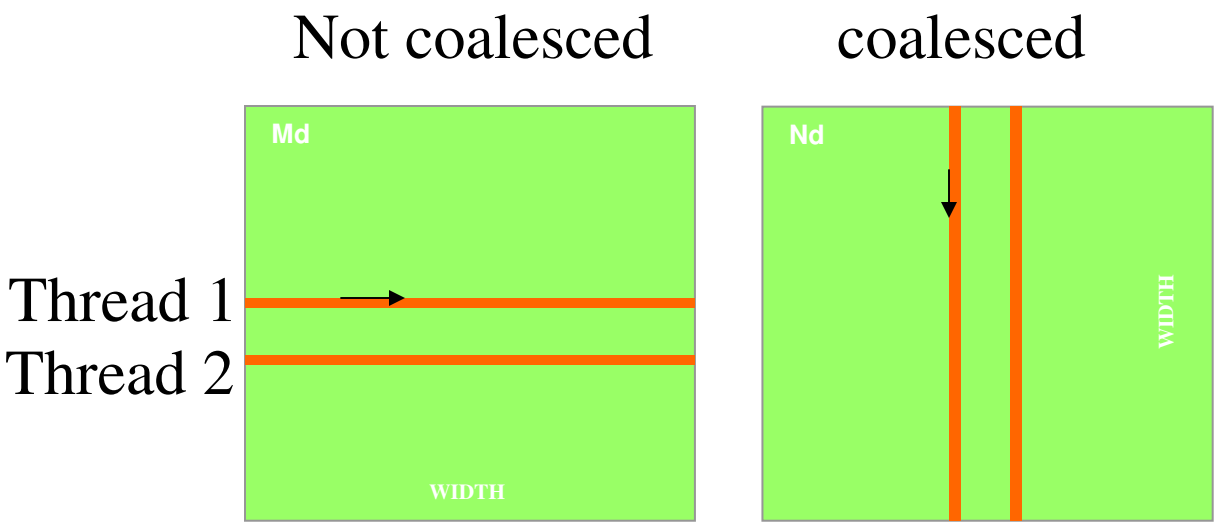
Memory Layout of a Matrix in C

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

M

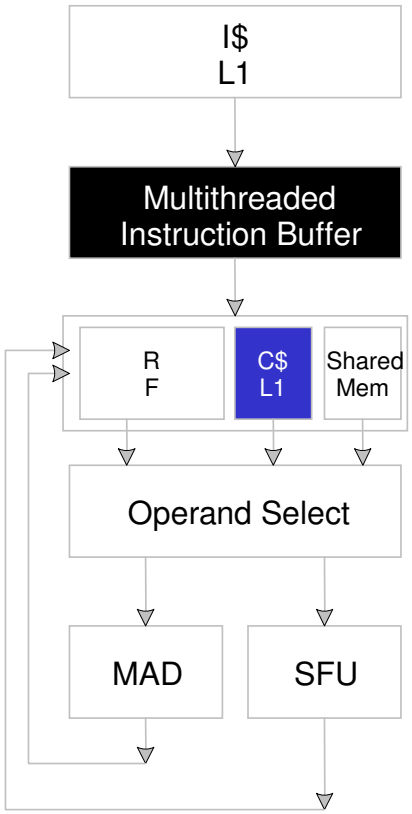
Memory Coalescing

- When accessing global memory, peak performance utilization occurs when all threads in a half warp access continuous memory locations.



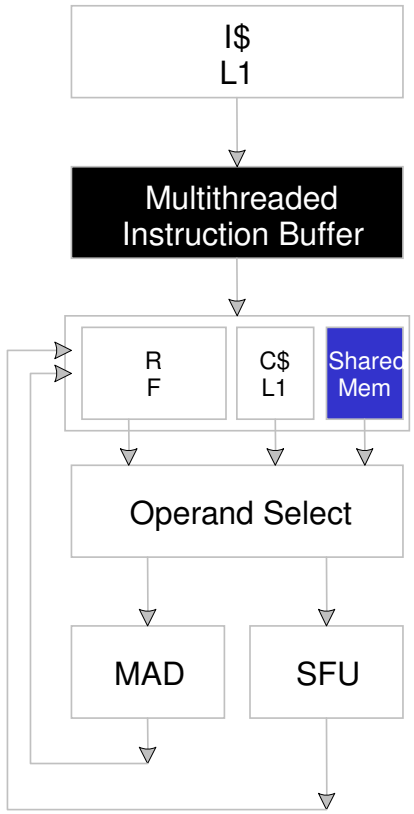
Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a block!



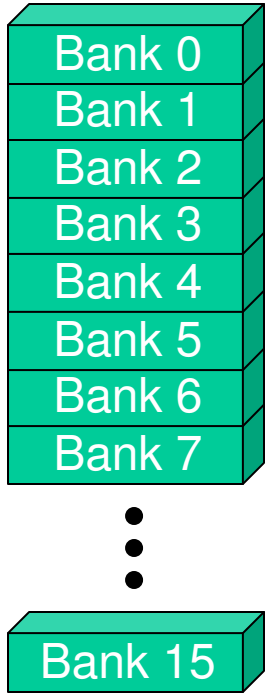
Shared Memory

- Each SM has 16 kB of Shared Memory
  - 16 **banks** of 32 bit words
- Shared Memory is visible to all threads in a thread block
  - read and write access



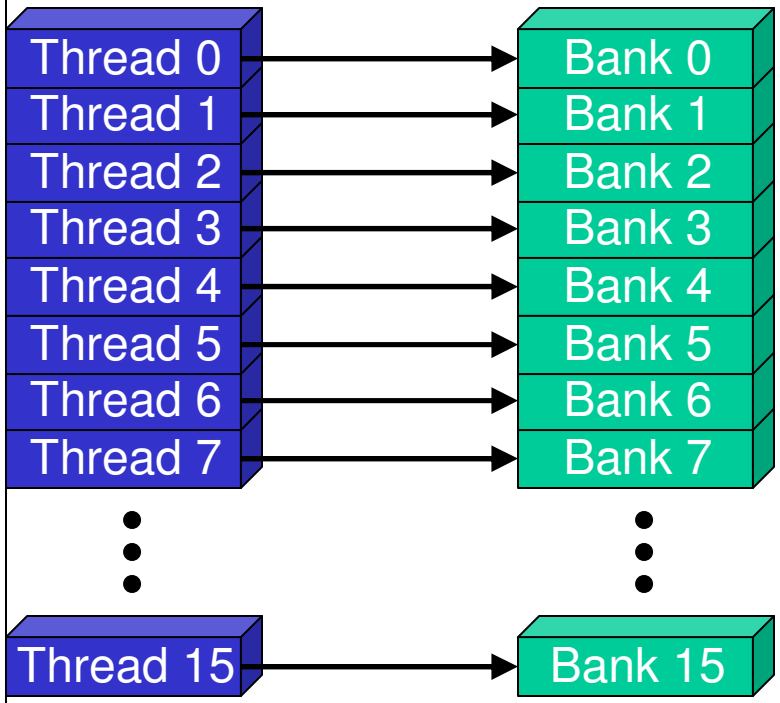
Parallel Memory Architecture

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth since the bandwidth per "wire" is limited
- 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

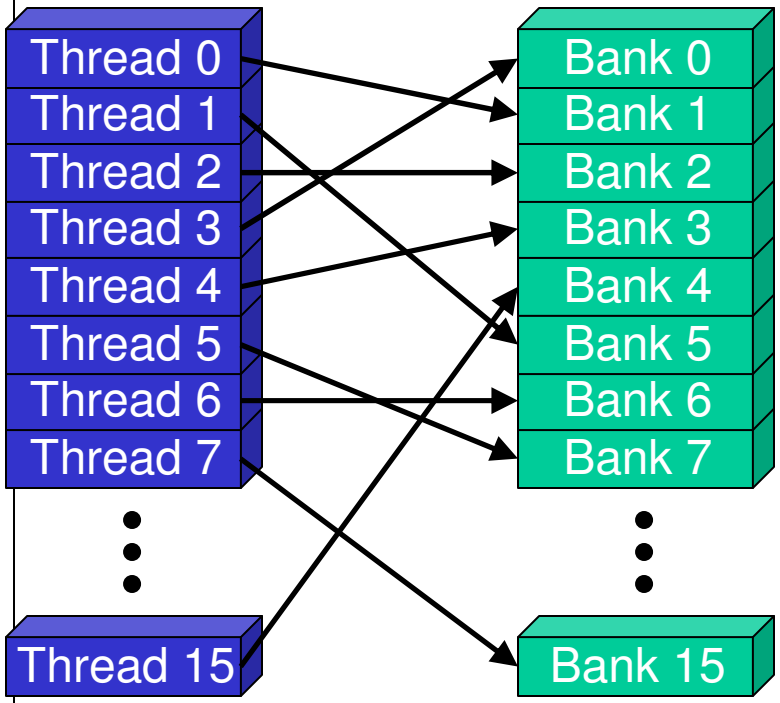


Bank Addressing Examples

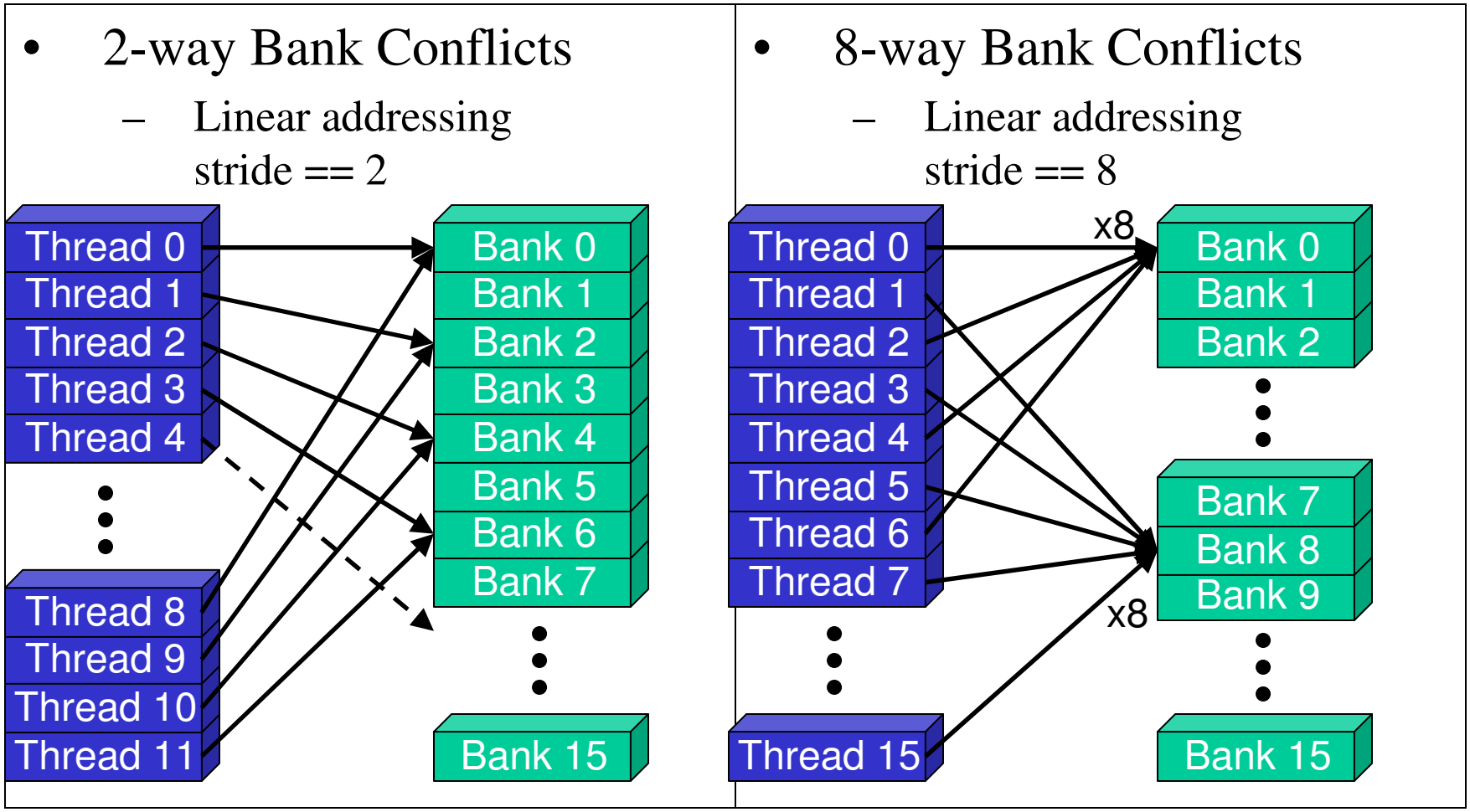
- No Bank Conflicts
  - Linear addressing stride == 1



- 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

Shared memory bank conflicts

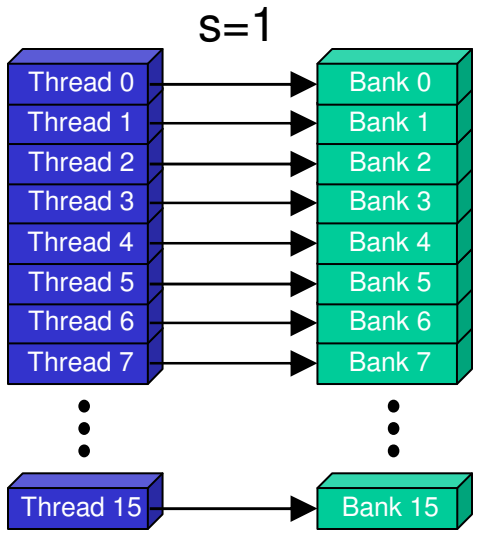
- Shared memory is as fast as registers if there are no bank conflicts

- 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

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