King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 15: GPU Compute APIs, Pt. 5

Markus Hadwiger, KAUST

Reading Assignment #6 (until Oct 21)

Read (required):

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

Read (optional):

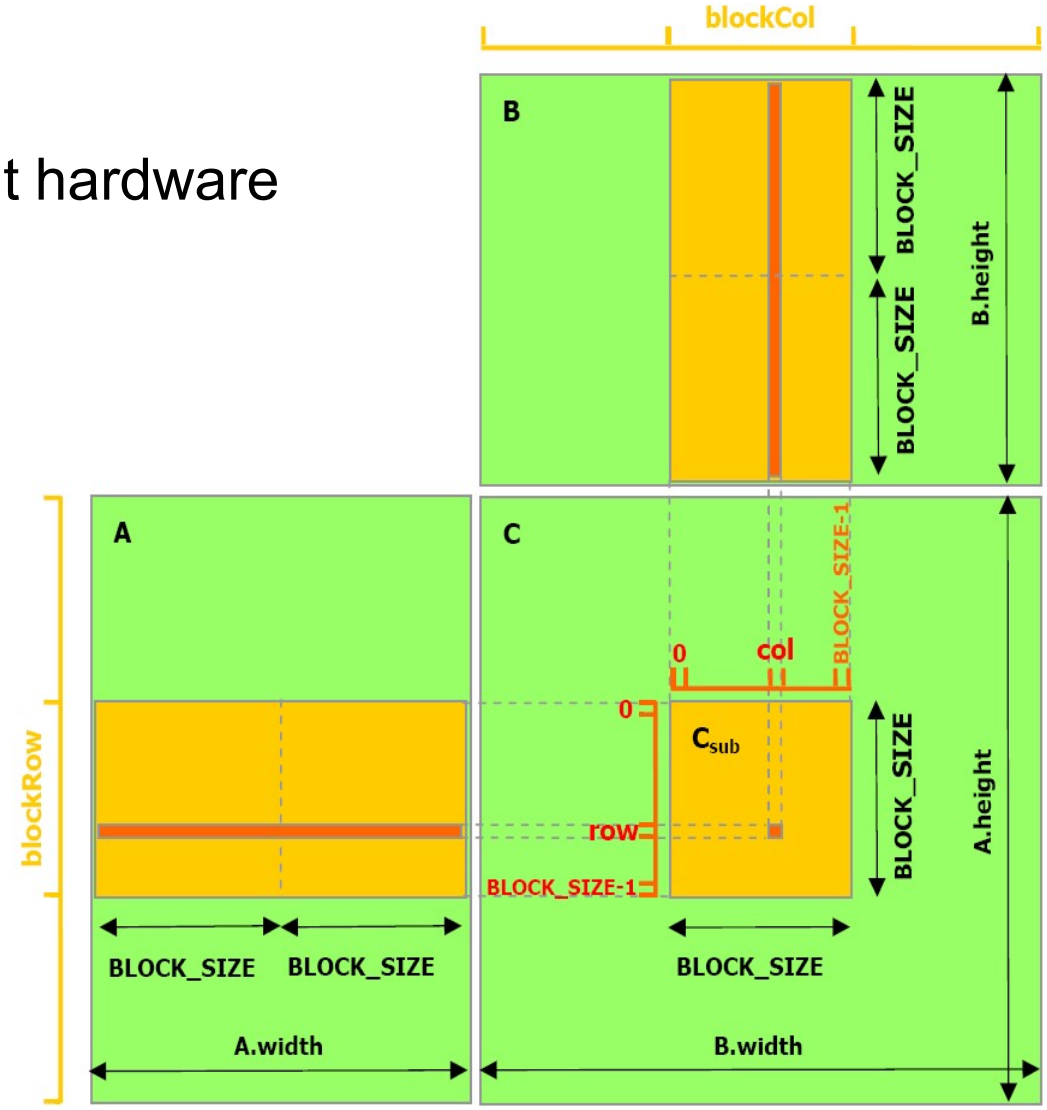
- Programming Massively Parallel Processors book (4th edition), **Chapter 20** (*An introduction to CUDA streams*)
- Programming Massively Parallel Processors book (4th edition), **Chapter 21** (*CUDA dynamic parallelism*)

Code Example #2: Matrix Multiply

Example: Matrix Multiplication (3)

- Multiply matrix block-wise
- Set BLOCK_SIZE for efficient hardware use, e.g., to 16 on cc. 1.x or 16 or 32 on cc. 2.x +

- Maximize parallelism
  - Launch as many threads per block as block elements
  - Each thread fetches one element per block
  - Perform row * column dot products in parallel

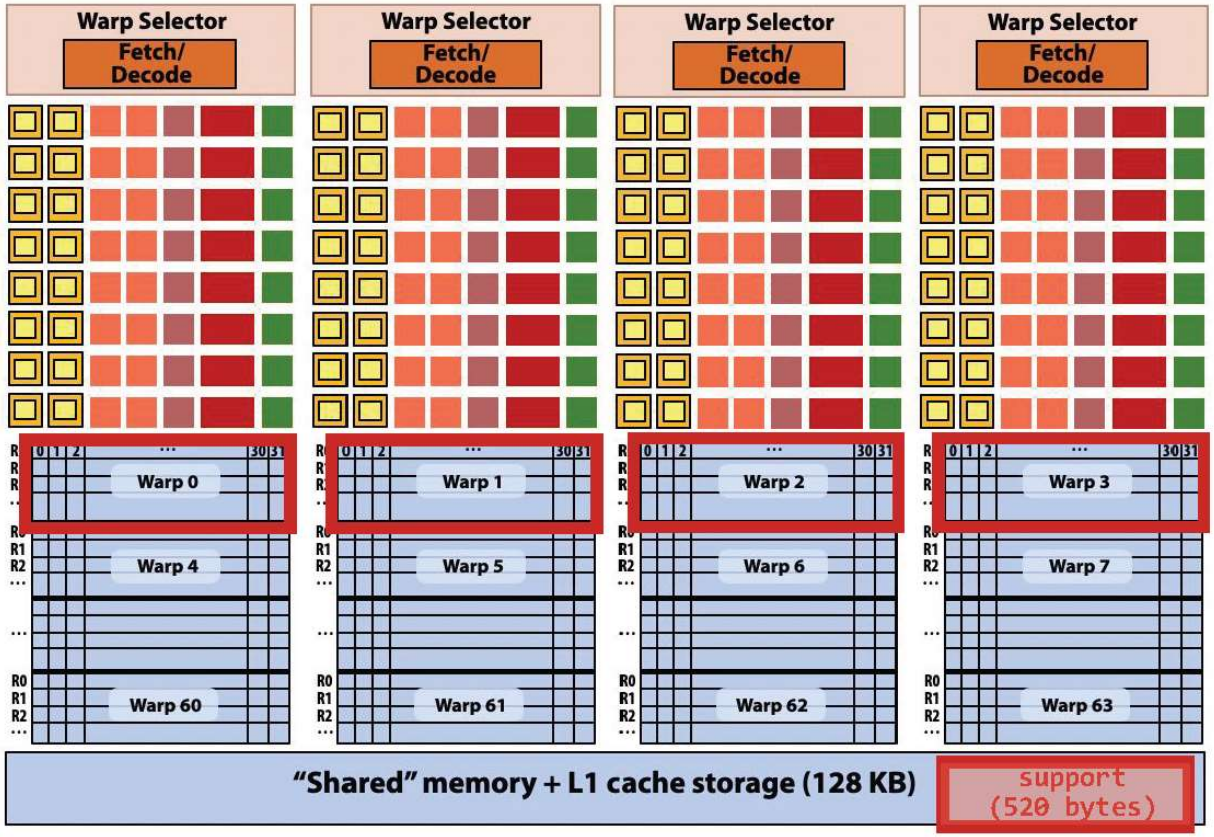


Example: Matrix Multiplication (4)

```
__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)

Running on a V100 (Volta) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

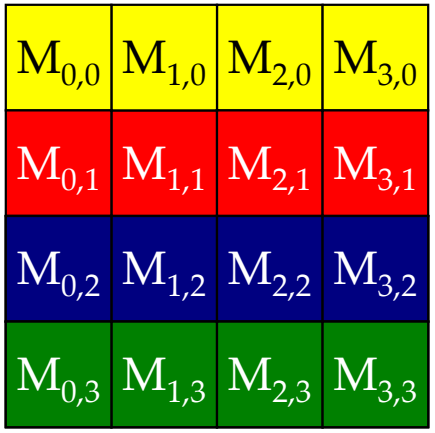
SM core operation each clock:

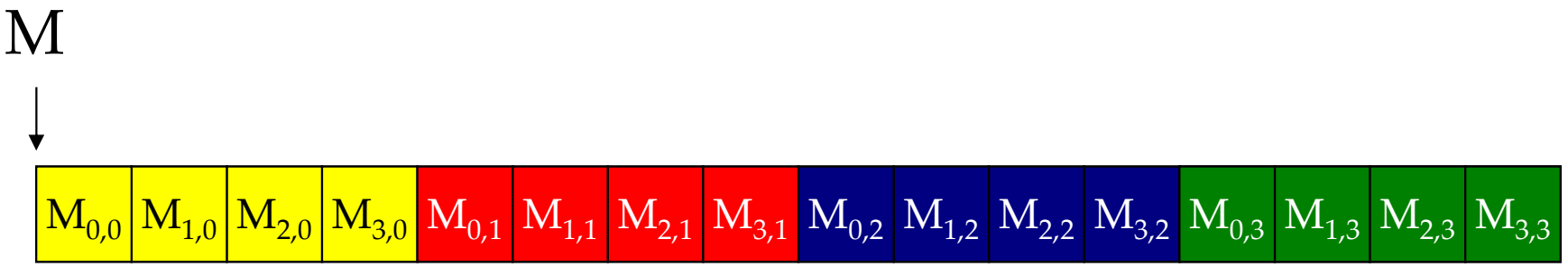
- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

What About Memory Performance? (more to come later…)

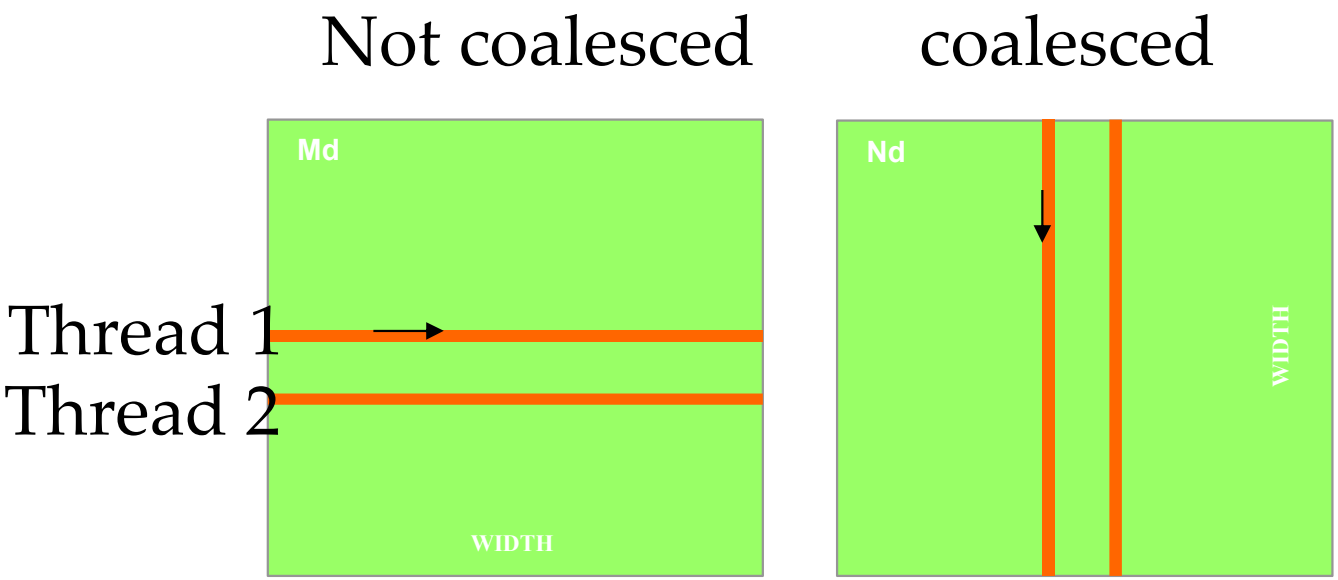
Memory Layout of a Matrix in C





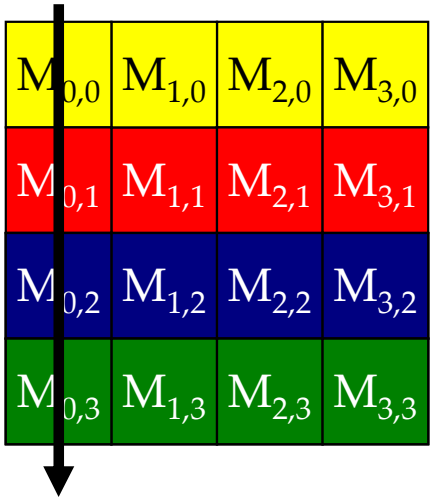
Memory Coalescing

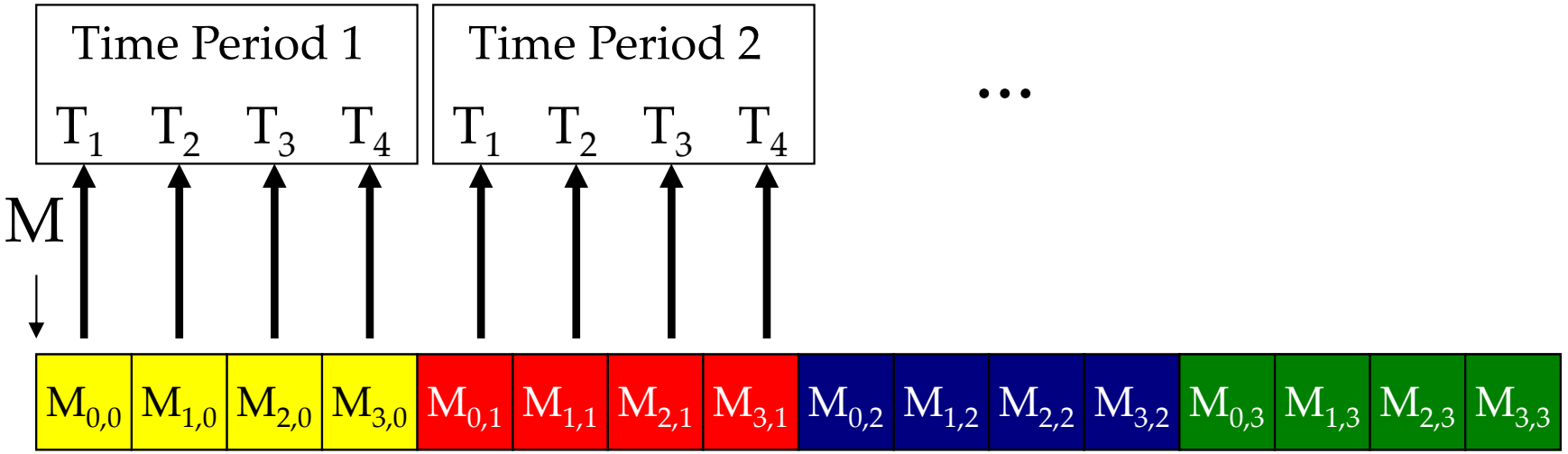
- When accessing global memory, peak performance utilization occurs when all threads in a half warp (full warp on Fermi+) access continuous memory locations.
- Requirements relaxed on >=1.2 devices; L1 cache on Fermi!



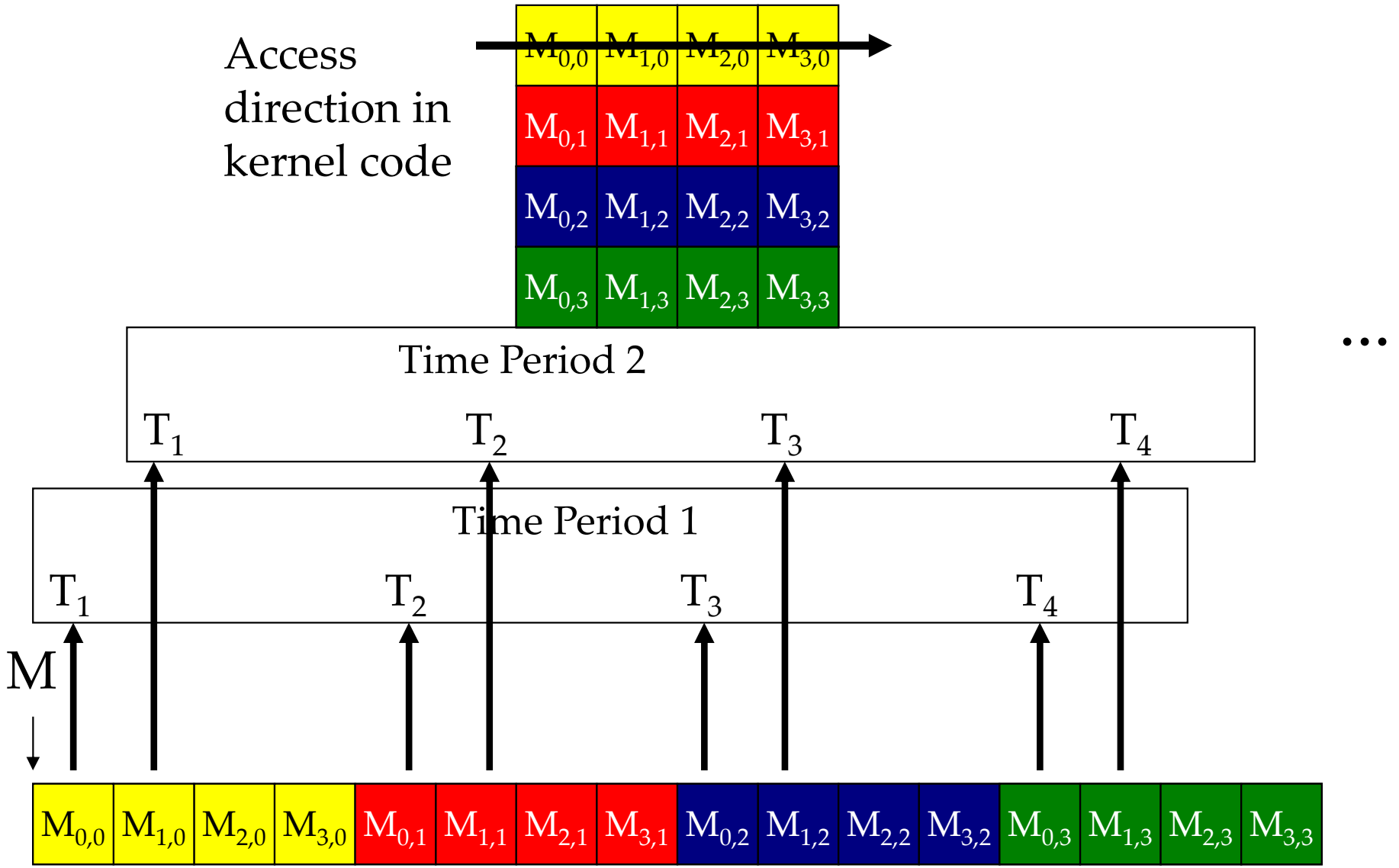
Memory Layout of a Matrix in C

Access direction in kernel code





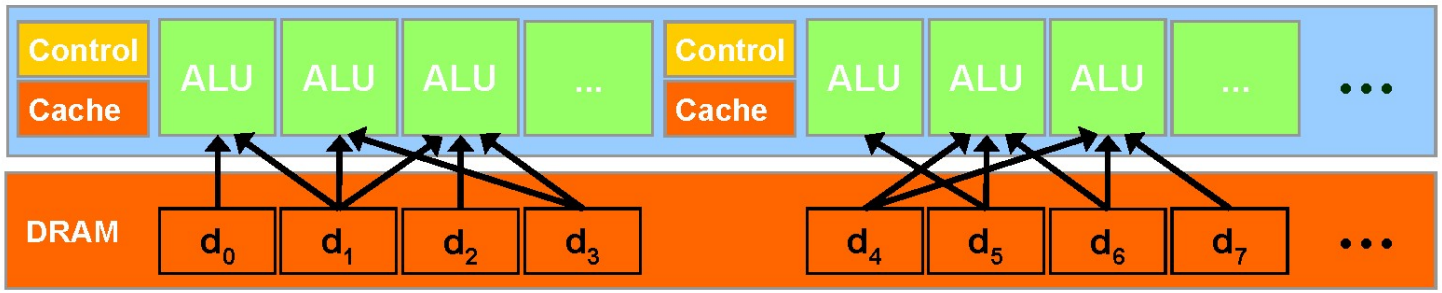
Memory Layout of a Matrix in C



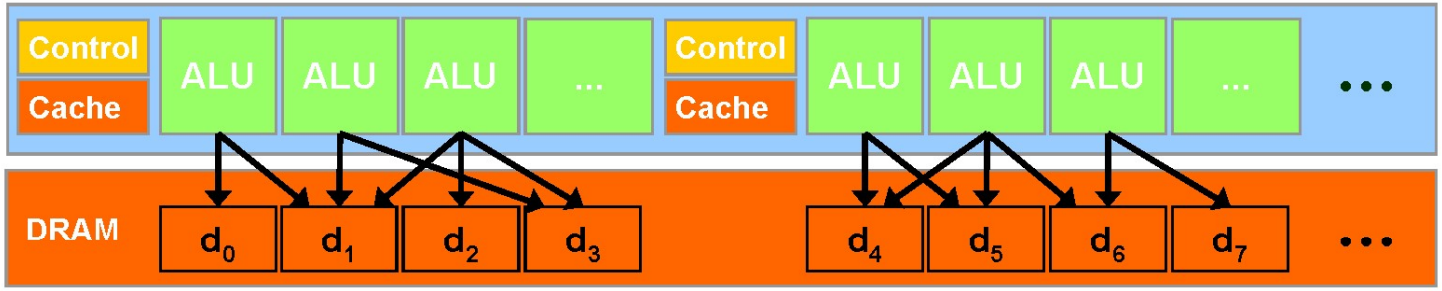
CUDA Memory

CUDA Highlights: Scatter

- **CUDA provides generic DRAM memory addressing**
  - Gather:



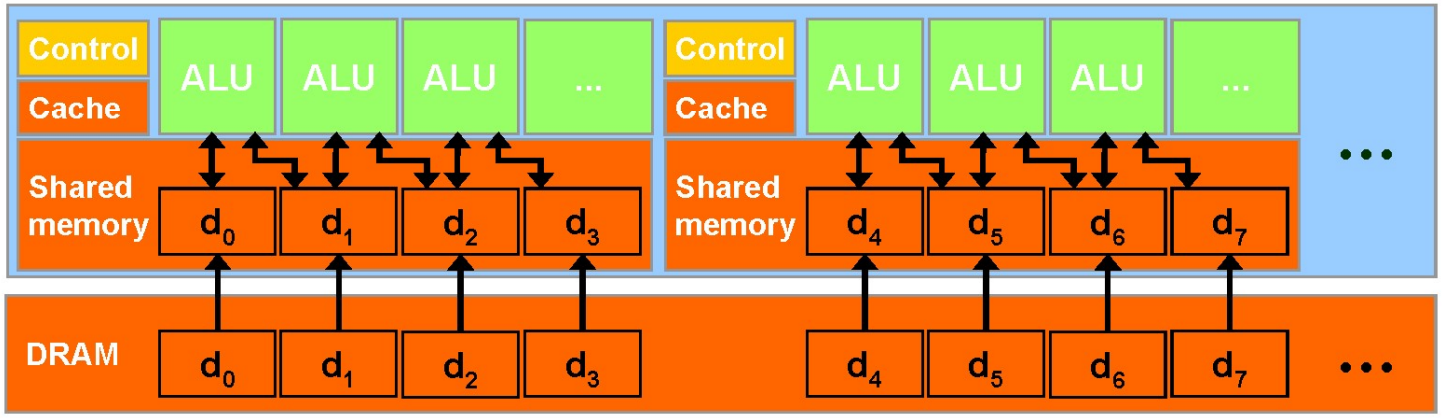
  - And scatter: no longer limited to write one pixel



→ **More programming flexibility**

CUDA Highlights: On-Chip Shared Memory

- **CUDA enables access to a parallel on-chip shared memory for efficient inter-thread data sharing**

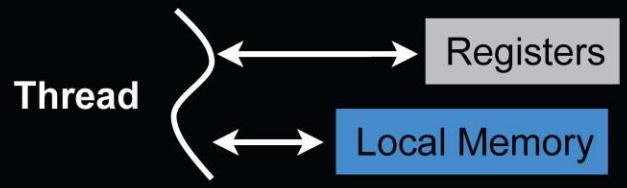


Big memory bandwidth savings

CUDA Memory: Overview

Kernel Memory Access

- **Per-thread**

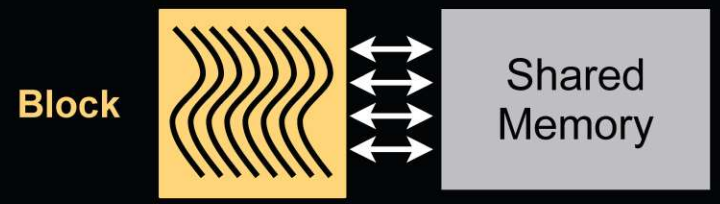


On-chip

Off-chip, ~~uncached~~

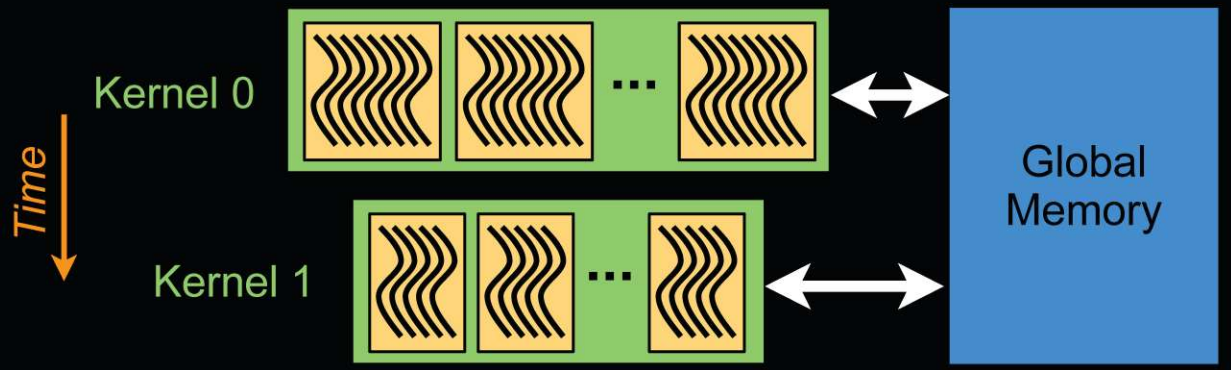
cached on Fermi or newer!

- **Per-block**



- On-chip, small
- Fast

- **Per-device**



- Off-chip, large
- ~~Uncached~~
- Persistent across kernel launches
- Kernel I/O

cached on Fermi or newer!

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 Configurations and Types for Different Compute Capabilities

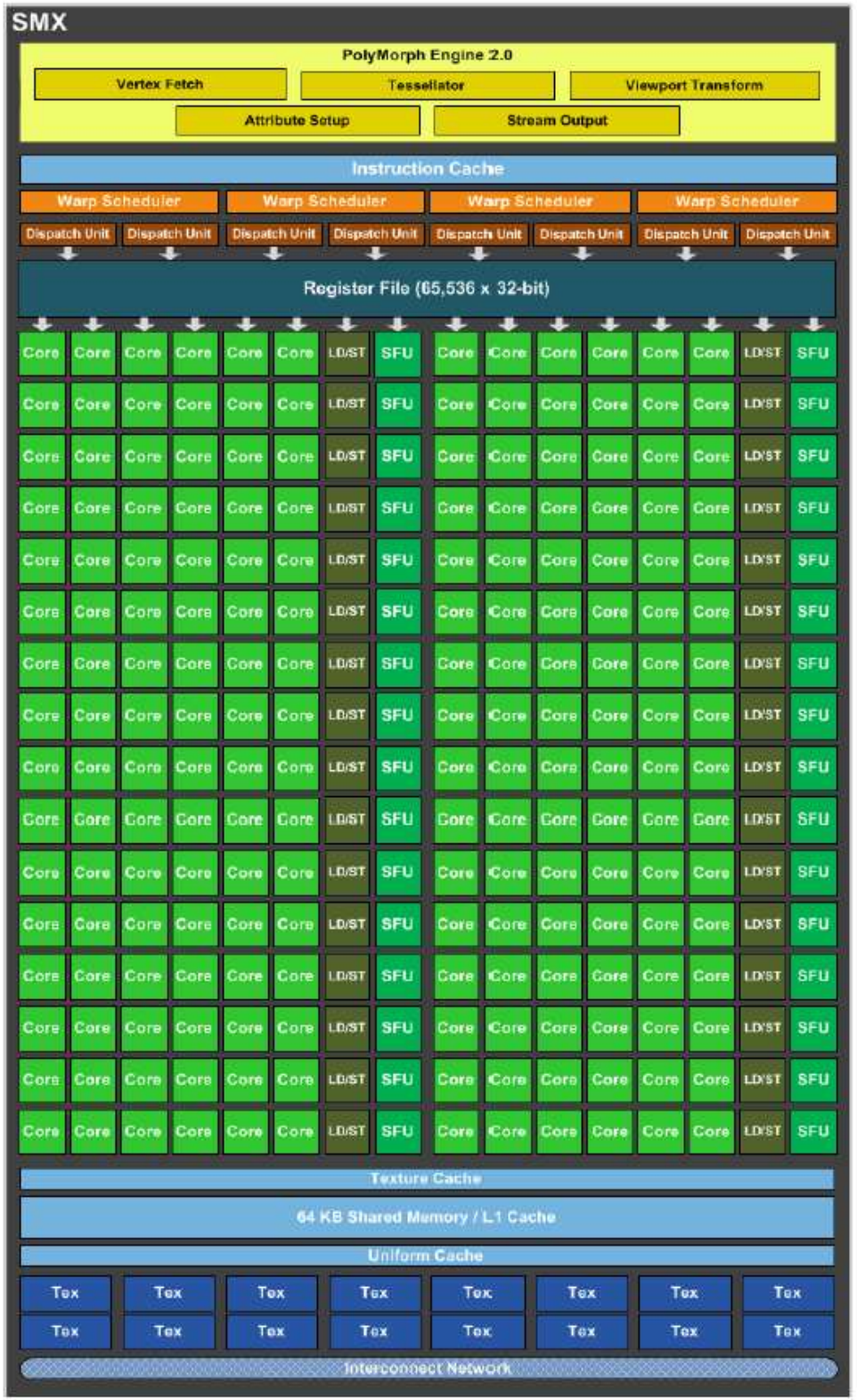
GK104 SMX

Multiprocessor: SMX (CC 3.0)

- 192 CUDA cores
  (192 = 6 * 32)
- 32 LD/ST units
- 32 SFUs
- 16 texture units

Two dispatch units per warp scheduler exploit ILP (*instruction-level parallelism*)

Can dual-issue ALU instructions! ("superscalar")



GK110 SMX

Multiprocessor: SMX (CC 3.5)

- 192 CUDA cores (192 = 6 * 32)
- 64 DP units
- 32 LD/ST units
- 32 SFUs
- 16 texture units

New read-only data cache (48KB)

Compute Capab. 3.x (Kepler, Part 1)

K.3.1. Architecture

An SM has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory.

There is an L1 cache for each SM and an L2 cache shared by all SMs. The L1 cache is used to cache accesses to local memory, including temporary register spills. The L2 cache is used to cache accesses to local and global memory. The cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load or store instruction. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory in both L1 and L2 via compiler options.

The same on-chip memory is used for both L1 and shared memory: It can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache or as 32 KB of shared memory and 32 KB of L1 cache, using `cudaFuncSetCacheConfig()/cuFuncSetCacheConfig()`:

Compute Capab. 3.x (Kepler, Part 2)

Note: Devices of compute capability 3.7 add an additional 64 KB of shared memory to each of the above configurations, yielding 112 KB, 96 KB, and 80 KB shared memory per SM, respectively. However, the maximum shared memory per thread block remains 48 KB.

Applications may query the L2 cache size by checking the `l2CacheSize` device property (see Device Enumeration). The maximum L2 cache size is 1.5 MB.

Each SM has a read-only data cache of 48 KB to speed up reads from device memory. It accesses this cache either directly (for devices of compute capability 3.5 or 3.7), or via a texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory. When accessed via the texture unit, the read-only data cache is also referred to as texture cache.

Compute Capab. 3.x (Kepler, Part 3)

K.3.2. Global Memory

Global memory accesses for devices of compute capability 3.x are cached in L2 and for devices of compute capability 3.5 or 3.7, may also be cached in the read-only data cache described in the previous section; they are normally not cached in L1. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory accesses in L1 via the `-Xptxas -dlcm=ca` option to `nvcc`.

A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions, whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses.

If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently:

- Two memory requests, one for each half-warp, if the size is 8 bytes,
- Four memory requests, one for each quarter-warp, if the size is 16 bytes.

Compute Capab. 3.x (Kepler, Part 4)

Each memory request is then broken down into cache line requests that are issued independently. A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Note that threads can access any words in any order, including the same words.

If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined.

Data that is read-only for the entire lifetime of the kernel can also be cached in the read-only data cache described in the previous section by reading it using the `__ldg()` function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Figure 21 shows some examples of global memory accesses and corresponding memory transactions.

Maxwell (GM) Architecture

Multiprocessor: SMM (CC 5.x)

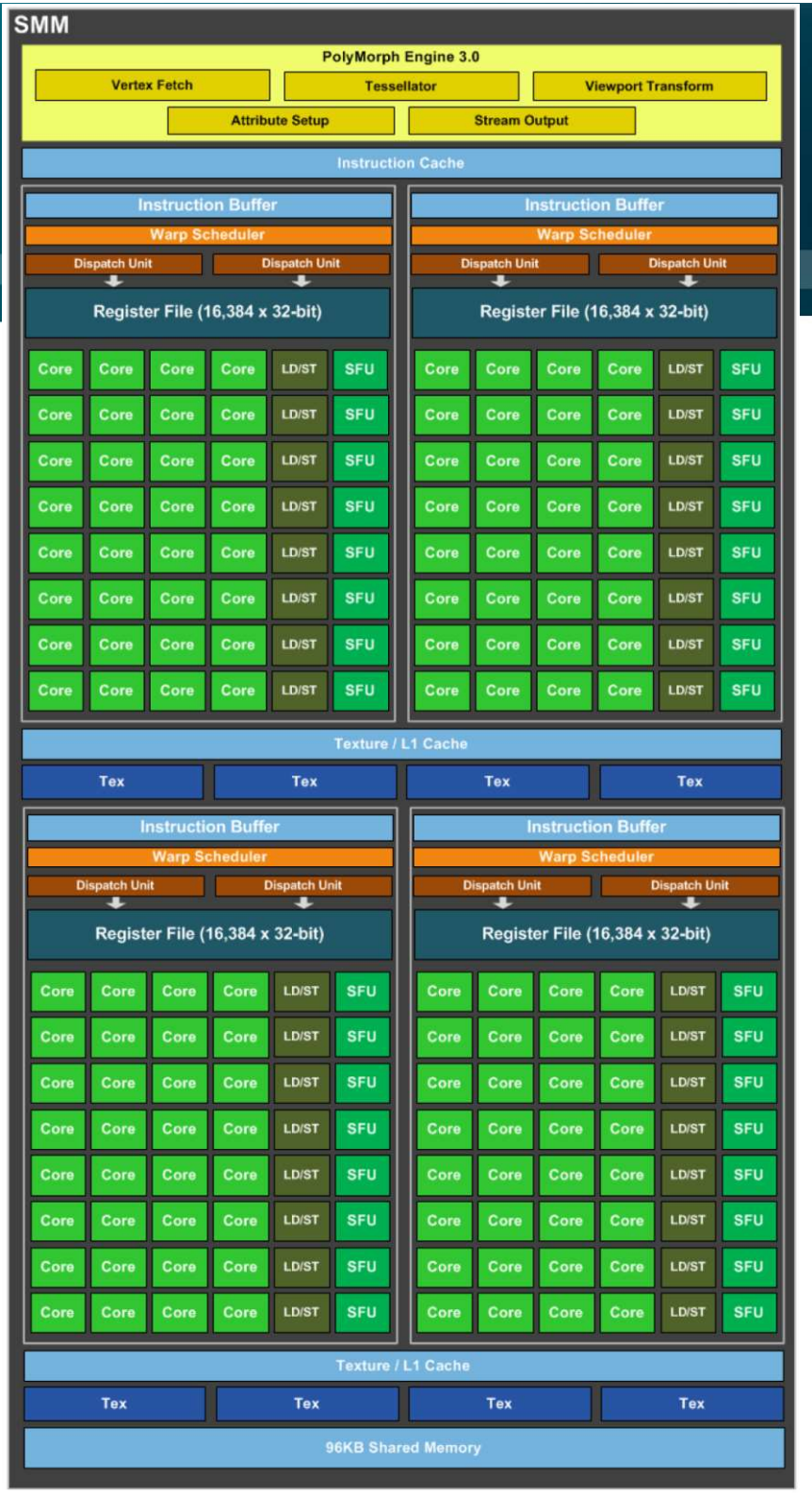
- 128 CUDA cores
- 4 DP units; 32 LD/ST units; 32 SFUs
- 8 texture units

4 partitions inside SMM

- 32 CUDA cores each
- 8 LD/ST units; 8 SFUs each
- Each has its own register file, warp scheduler, two dispatch units *(but cannot dual-issue ALU insts.!)*

Shared memory and L1 cache now separate!

- L1 cache shares with texture cache
- Shared memory is its own space



Compute Capab. 5.x (Maxwell, Part 1)

K.4.1. Architecture

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified L1/texture cache of 24 KB used to cache reads from global memory,
- 64 KB of shared memory for devices of compute capability 5.0 or 96 KB of shared memory for devices of compute capability 5.2.

The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory.

There is also an L2 cache shared by all SMs that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the `l2CacheSize` device property (see Device Enumeration).

The cache behavior (e.g., whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction.

Compute Capab. 5.x (Maxwell, Part 2)

K.4.2. Global Memory

Global memory accesses are always cached in L2 and caching in L2 behaves in the same way as for devices of compute capability 3.x (see Global Memory).

Data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache described in the previous section by reading it using the `__ldg()` function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

Compute Capab. 5.x (Maxwell, Part 3)

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

- Perform the read using inline assembly with the appropriate modifier as described in the PTX reference manual;
- Compile with the `-Xptxas -dlcm=ca` compilation flag, in which case all reads are cached, except reads that are performed using inline assembly with a modifier that disables caching;
- Compile with the `-Xptxas -fscm=ca` compilation flag, in which case all reads are cached, including reads that are performed using inline assembly regardless of the modifier used.

When caching is enabled using one of the three mechanisms listed above, devices of compute capability 5.2 will cache global memory reads in the unified L1/texture cache for all kernel launches except for the kernel launches for which thread blocks consume too much of the SM's register file. These exceptions are reported by the profiler.

Compute Capab. 5.x (Maxwell, Part 4)

K.4.3. Shared Memory

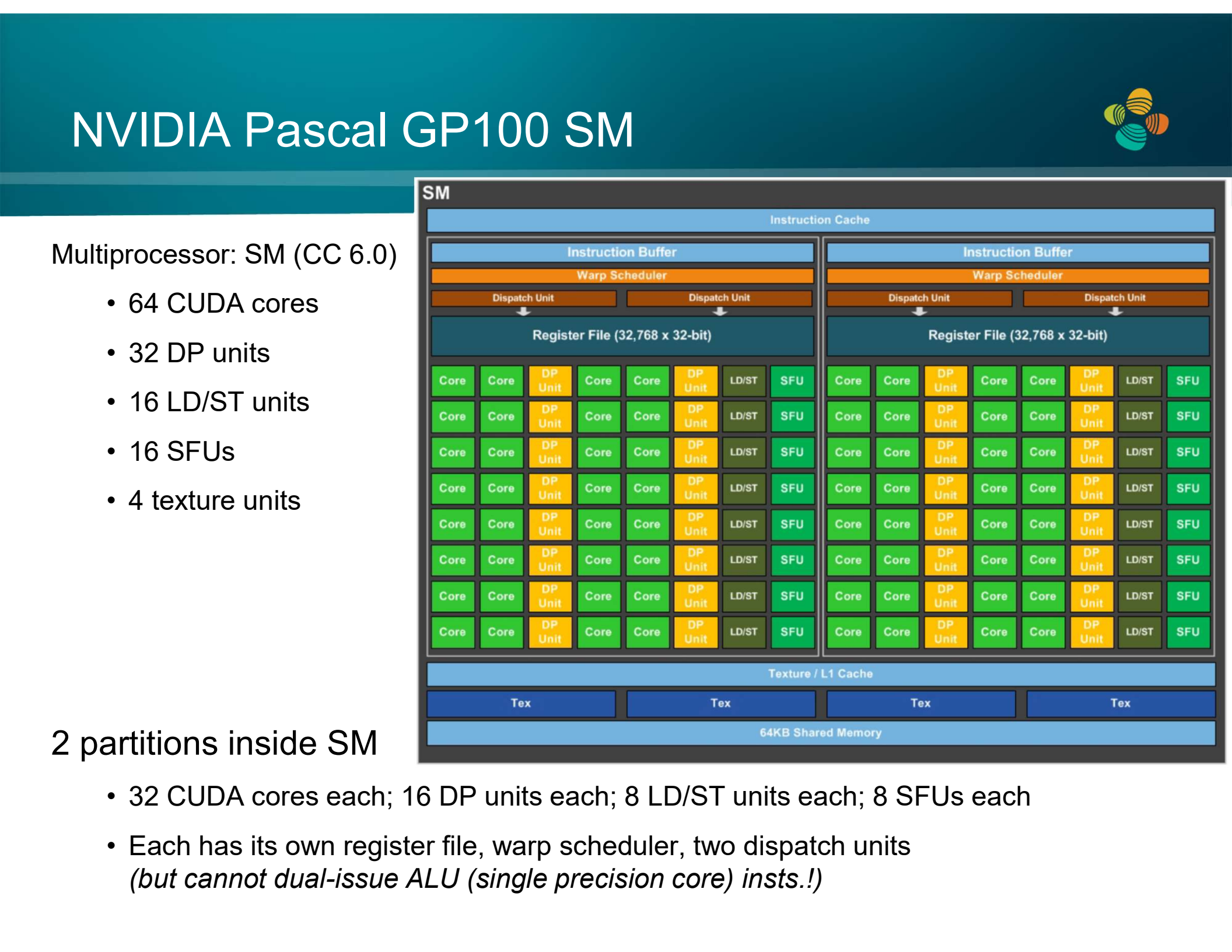
Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle.

A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank). In that case, for read accesses, the word is broadcast to the requesting threads and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).

Figure 22 shows some examples of strided access.

Figure 23 shows some examples of memory read accesses that involve the broadcast mechanism.

NVIDIA Pascal GP100 SM



Multiprocessor: SM (CC 6.0)

- 64 CUDA cores
- 32 DP units
- 16 LD/ST units
- 16 SFUs
- 4 texture units

2 partitions inside SM

- 32 CUDA cores each; 16 DP units each; 8 LD/ST units each; 8 SFUs each
- Each has its own register file, warp scheduler, two dispatch units
  (but cannot dual-issue ALU (single precision core) insts.!)

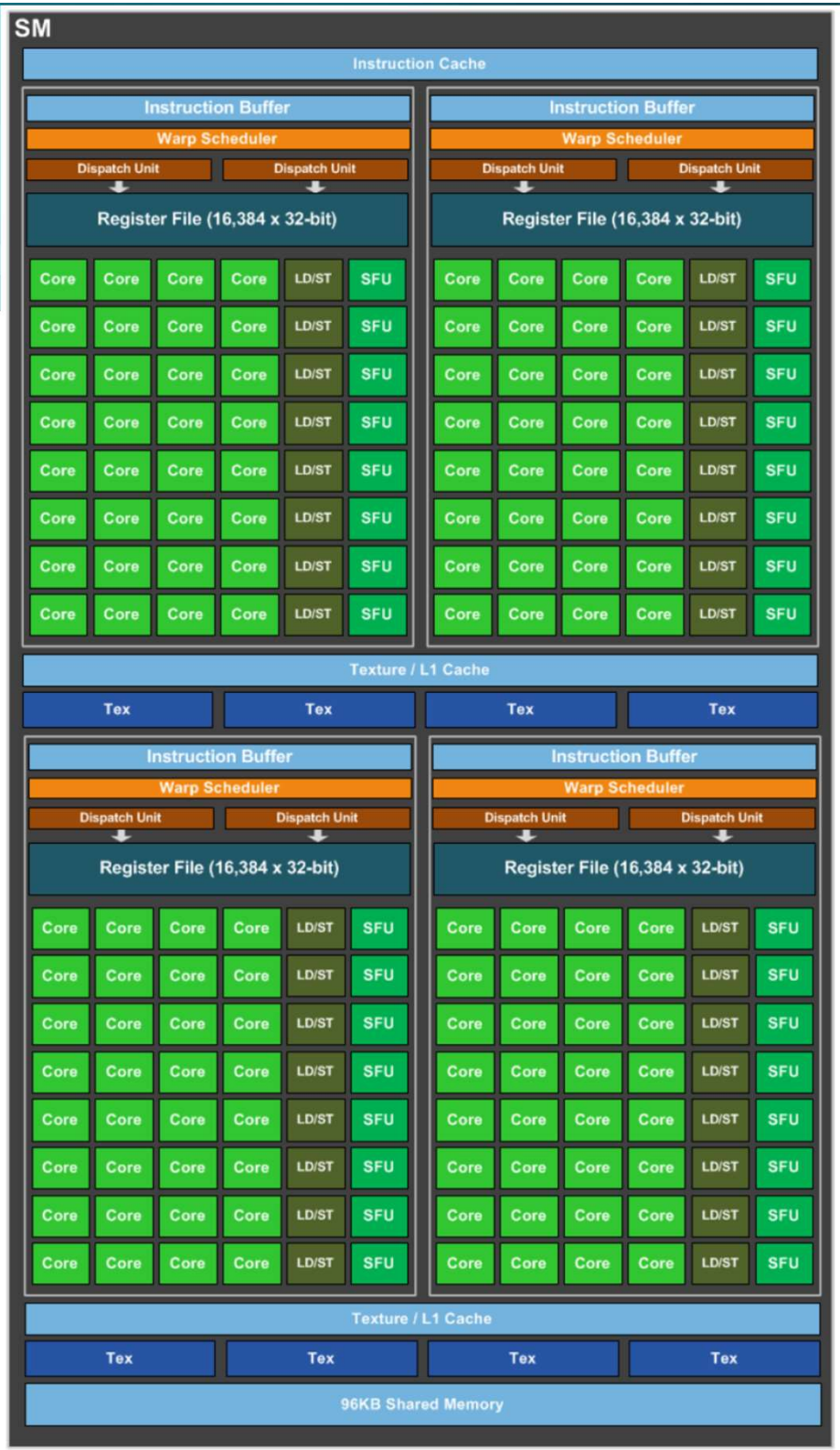
NVIDIA Pascal GP104 SM

Multiprocessor: SM (CC 6.1/6.2)

- 128 CUDA cores
- 32 LD/ST units
- 32 SFUs
- 8 texture units

4 partitions inside SM

- 32 CUDA cores; 8 LD/ST units; 8 SFUs
- Each has its own register file, warp scheduler, two dispatch units *(but cannot dual-issue ALU insts.!)*



Compute Capab. 6.x (Pascal, Part 1)

K.5.1. Architecture

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified L1/texture cache for reads from global memory of size 24 KB (6.0 and 6.2) or 48 KB (6.1),
- a shared memory of size 64 KB (6.0 and 6.2) or 96 KB (6.1).

The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory.

There is also an L2 cache shared by all SMs that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the `l2CacheSize` device property (see Device Enumeration).

The cache behavior (e.g., whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction.

Compute Capab. 6.x (Pascal, Part 2)

K.5.2. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See Global Memory).

K.5.3. Shared Memory

Shared memory behaves the same way as in devices of compute capability 5.x (See Shared Memory).

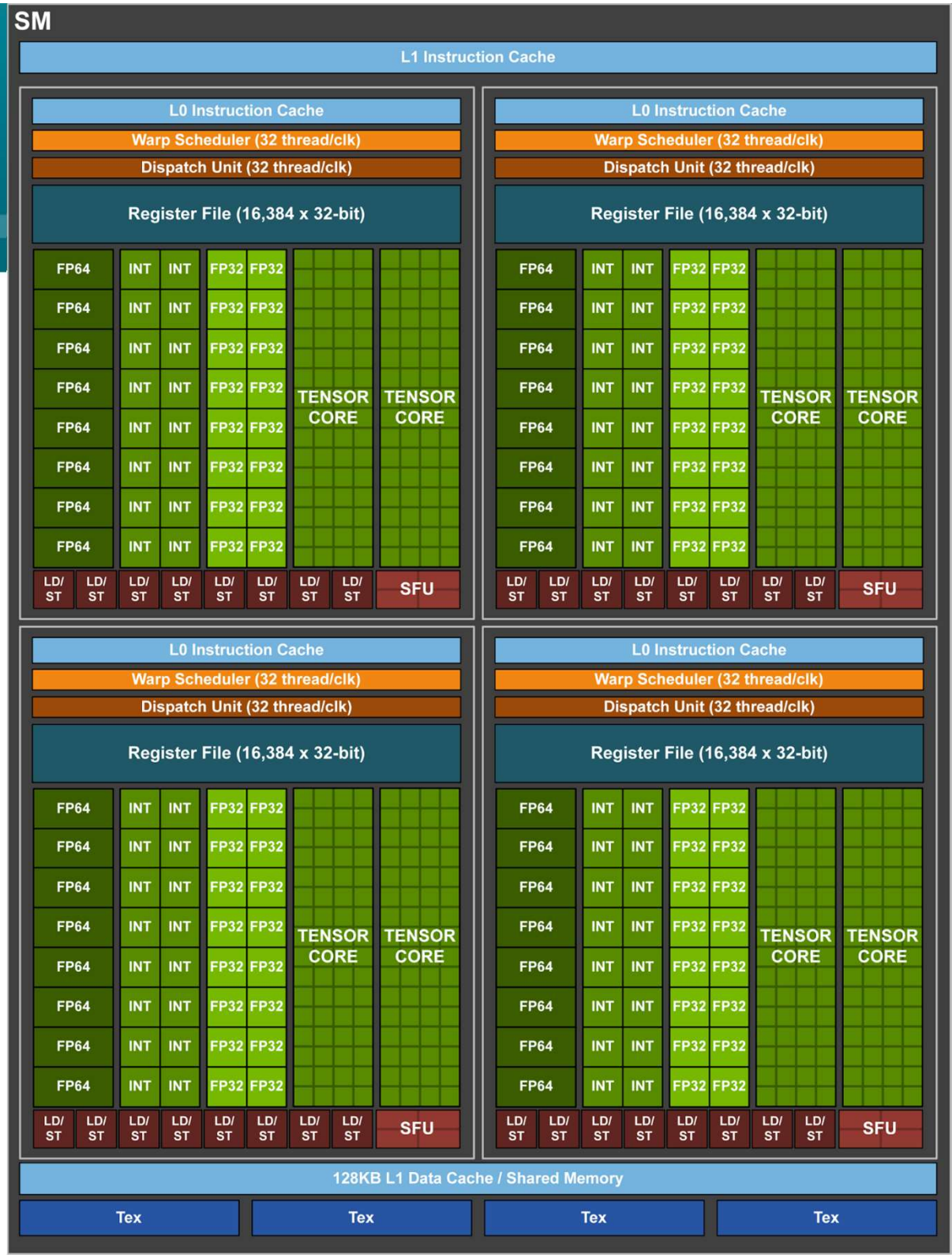
NVIDIA Volta SM

Multiprocessor: SM (CC 7.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 32 LD/ST units; 16 SFUs
- 8 tensor cores (FP16/FP32 mixed-precision)

4 partitions inside SM

- 16 FP32 + 16 INT32 cores each
- 8 FP64 cores each
- 8 LD/ST units; 4 SFUs each
- 2 tensor cores each
- Each has: warp scheduler, dispatch unit, register file



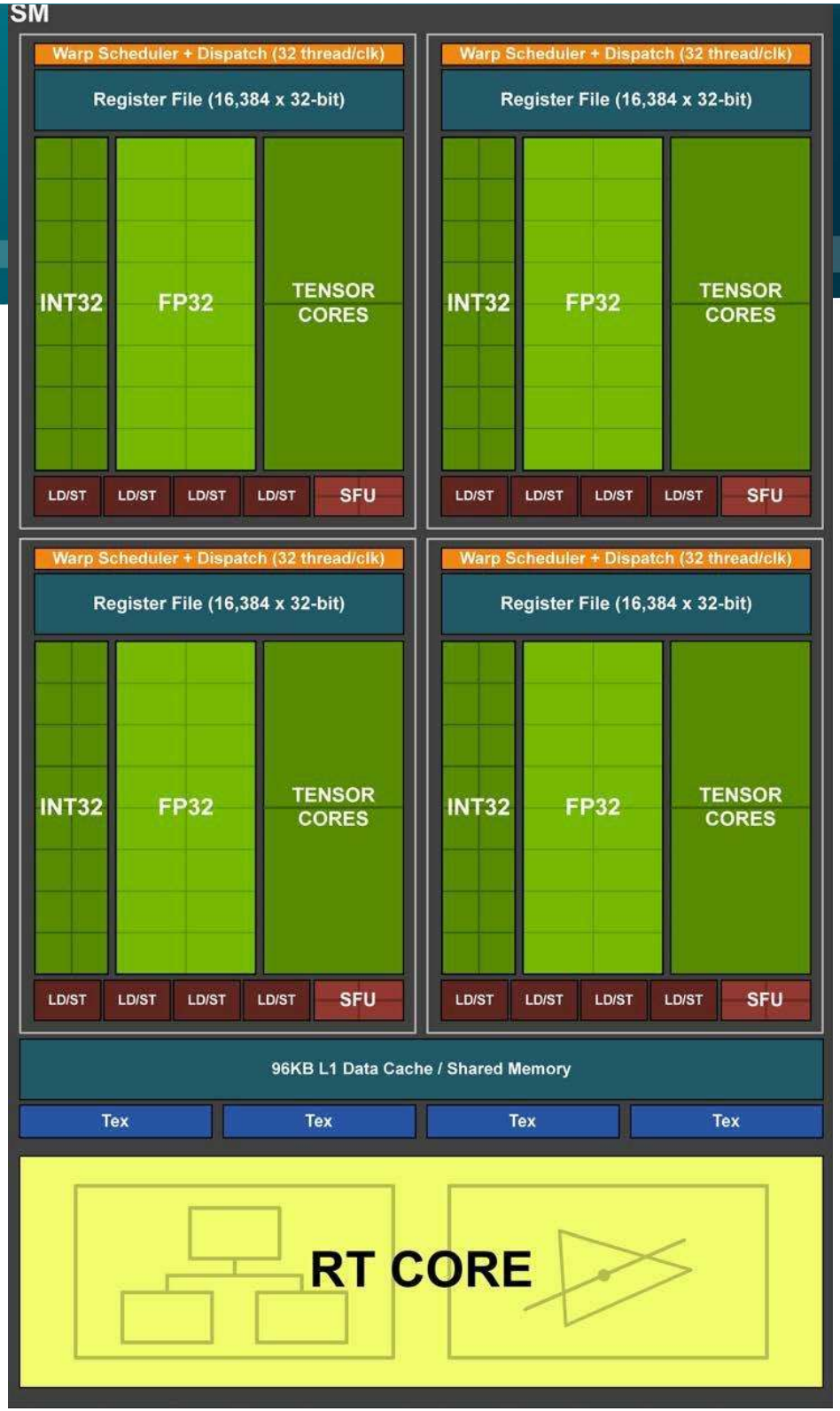
NVIDIA Turing SM

Multiprocessor: SM (CC 7.5)

- 64 FP32 + INT32 cores
- 2 (!) FP64 cores
- 8 Turing tensor cores (FP16/32, INT4/8 mixed-precision)
- 1 RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + INT32 cores each
- 4 LD/ST units; 4 SFUs each
- 2 Turing tensor cores each
- Each has: warp scheduler, dispatch unit, 16K register file



Compute Capab. 7.x (Volta/Turing, Part 1)

K.6.1. Architecture

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 128 KB (*Volta*) or 96 KB (*Turing*).

Shared memory is partitioned out of unified data cache, and can be configured to various sizes (See Shared Memory.) The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in Texture and Surface Memory.

Compute Capab. 7.x (Volta/Turing, Part 2)

K.6.3. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See Global Memory).

K.6.4. Shared Memory

Similar to the Kepler architecture, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *Volta* architecture (compute capability 7.0), the unified data cache has a size of 128 KB, and the shared memory capacity can be set to 0, 8, 16, 32, 64 or 96 KB. For the *Turing* architecture (compute capability 7.5), the unified data cache has a size of 96 KB, and the shared memory capacity can be set to either 32 KB or 64 KB. Unlike *Kepler*, the driver automatically configures the shared memory capacity for each kernel to avoid shared memory occupancy bottlenecks while also allowing concurrent execution with already launched kernels where possible. In most cases, the driver's default behavior should provide optimal performance.

Compute Capab. 7.x (Volta/Turing, Part 3)

Because the driver is not always aware of the full workload, it is sometimes useful for applications to provide additional hints regarding the desired shared memory configuration. For example, a kernel with little or no shared memory use may request a larger carveout in order to encourage concurrent execution with later kernels that require more shared memory. The new `cudaFuncSetAttribute()` API allows applications to set a preferred shared memory capacity, or `carveout`, as a percentage of the maximum supported shared memory capacity (96 KB for *Volta*, and 64 KB for *Turing*).

`cudaFuncSetAttribute()` relaxes enforcement of the preferred shared capacity compared to the legacy `cudaFuncSetCacheConfig()` API introduced with Kepler. The legacy API treated shared memory capacities as hard requirements for kernel launch. As a result, interleaving kernels with different shared memory configurations would needlessly serialize launches behind shared memory reconfigurations. With the new API, the carveout is treated as a hint. The driver may choose a different configuration if required to execute the function or to avoid thrashing.

Compute Capab. 7.x (Volta/Turing, Part 4)

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

In addition to an integer percentage, several convenience enums are provided as listed in the code comments above. Where a chosen integer percentage does not map exactly to a supported capacity (SM 7.0 devices support shared capacities of 0, 8, 16, 32, 64, or 96 KB), the next larger capacity is used. For instance, in the example above, 50% of the 96 KB maximum is 48 KB, which is not a supported shared memory capacity. Thus, the preference is rounded up to 64 KB.

Compute Capab. 7.x (Volta/Turing, Part 5)

Compute capability 7.x devices allow a single thread block to address the full capacity of shared memory: 96 KB on *Volta*, 64 KB on *Turing*. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, as such they must use dynamic shared memory (rather than statically sized arrays) and require an explicit opt-in using `cudaFuncSetAttribute()` as follows.

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

// Host code
int maxbytes = 98304; // 96 KB
cudaFuncSetAttribute(MyKernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
 maxbytes);
MyKernel <<<gridDim, blockDim, maxbytes>>>(...);
```

Otherwise, shared memory behaves the same way as for devices of compute capability 5.x (See Shared Memory).

NVIDIA GA100 SM

Multiprocessor: SM (CC 8.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + 16 INT32 cores
- 8 FP64 cores
- 8 LD/ST units; 4 SFUs each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA GA10x SM

Multiprocessor: SM (CC 8.6)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 32 (16+16) FP32 + 16 INT32 cores
- 4 LD/ST units; 4 SFUs each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

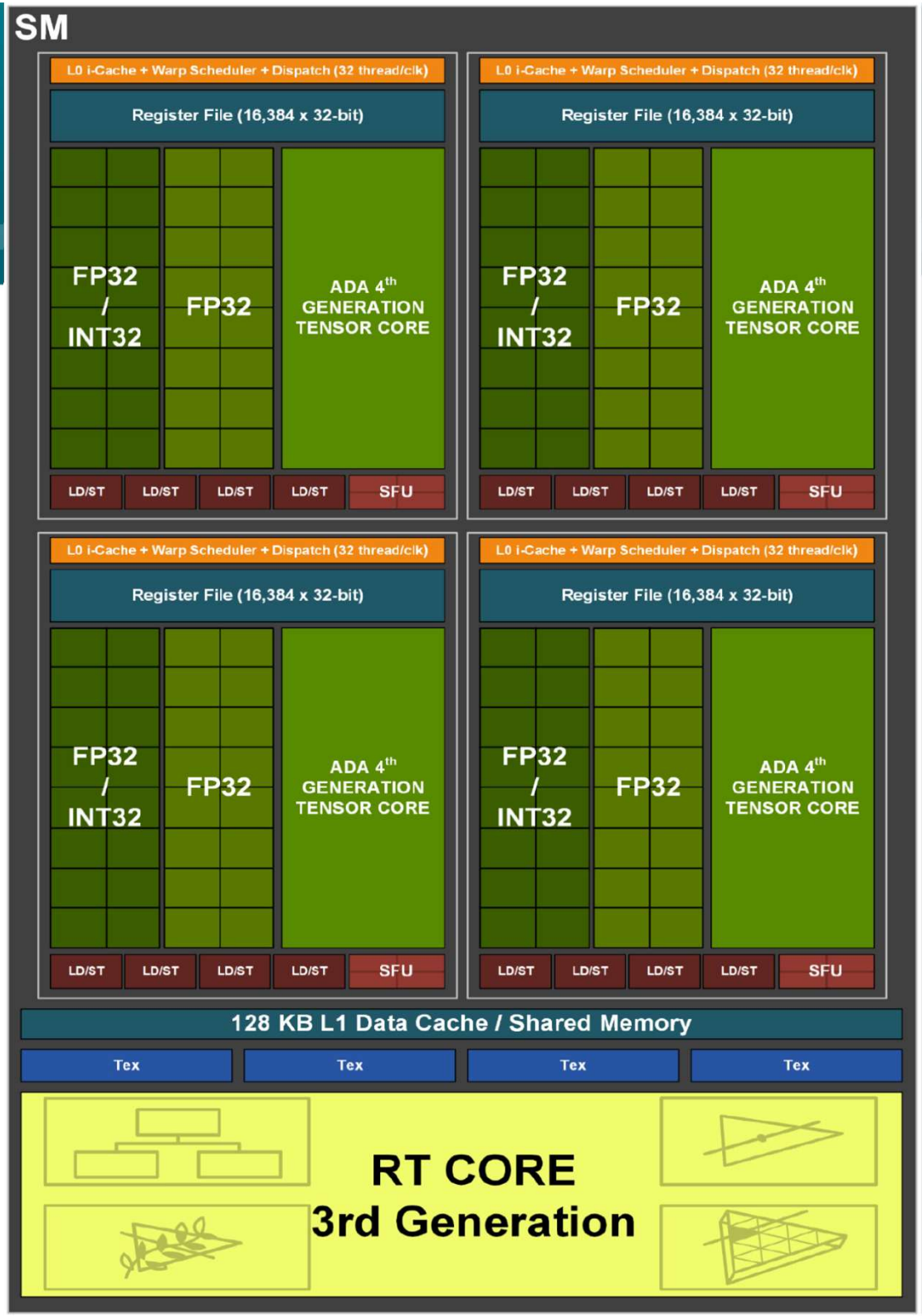
NVIDIA AD102 SM

Multiprocessor: SM (CC 8.9)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores (not in diagram)
- 4x 4th gen tensor cores
- 1x 3rd gen RT (ray tracing) core
- ++ thread block clusters, FP8, … (?)

4 partitions inside SM

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



Compute Capab. 8.x (Ampere/Ada, Part 1)

K.7.1. Architecture

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 192 KB for devices of compute capability 8.0 and 8.7 (1.5x *Volta*'s 128 KB capacity) and 128 KB for devices of compute capabilities 8.6 and 8.9.

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see Shared Memory section). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in Texture and Surface Memory.

Compute Capab. 8.x (Ampere/Ada, Part 2)

K.7.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See Global Memory).

K.7.3. Shared Memory

Similar to the Volta architecture, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *NVIDIA Ampere GPU architecture*, the unified data cache has a size of 192 KB for devices of compute capability 8.0 and 128 KB for devices of compute capability 8.6 and 8.9. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132 or 164 KB for devices of compute capability 8.0, and to 0, 8, 16, 32, 64 or 100 KB for devices of compute capabilities 8.6 and 8.9.

An application can set the `carveout`, i.e., the preferred shared memory capacity, with the `cudaFuncSetAttribute()`.

```
cudaFuncSetAttribute(kernel_name, cudaFuncAttributePreferredSharedMemoryCarveout,
 carveout);
```

Compute Capab. 8.x (Ampere/Ada, Part 3)

The API can specify the carveout either as an integer percentage of the maximum supported shared memory capacity of 164 KB for devices of compute capability 8.0 and 100 KB for devices of compute capabilities 8.6 and 8.9 respectively, or as one of the following values: `{cudaSharedmemCarveoutDefault`, `cudaSharedmemCarveoutMaxL1`, or `cudaSharedmemCarveoutMaxShared`. When using a percentage, the carveout is rounded up to the nearest supported shared memory capacity. For example, for devices of compute capability 8.0, 50% will map to a 100 KB carveout instead of an 82 KB one. Setting the `cudaFuncAttributePreferredSharedMemoryCarveout` is considered a hint by the driver; the driver may choose a different configuration, if needed.

Devices of compute capability 8.0 allow a single thread block to address up to 163 KB of shared memory, while devices of compute capabilities 8.6 and 8.9 allow up to 99 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.

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

Compute Capab. 9.x (Hopper, Part 1)

K.8.1. Architecture

An SM has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a unified data cache and shared memory with a total size of 256 KB for devices of compute capability 9.0 (1.33x *NVIDIA Ampere GPU Architecture*'s 192 KB capacity).

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see Shared Memory section). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in Texture and Surface Memory.

K.8.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See Global Memory).

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.

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

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**

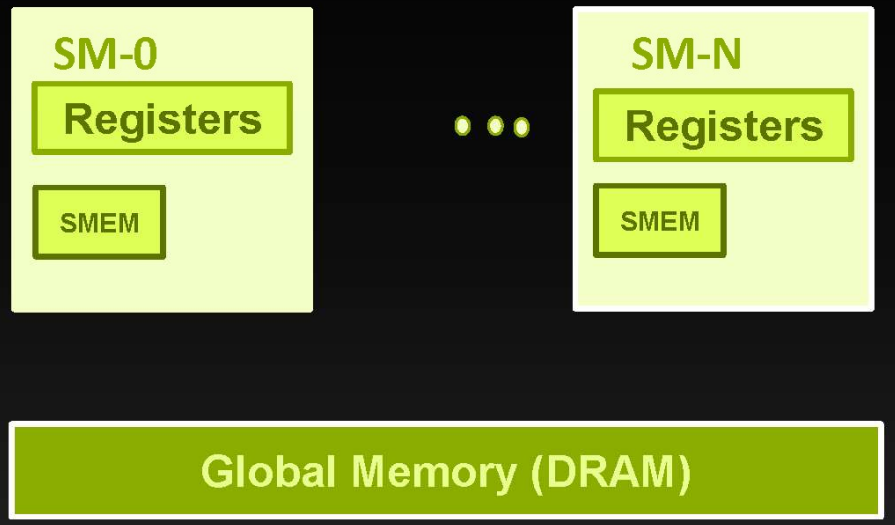
```
__shared__ float vec[256];
```

- **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()~~

later: use *carveout*

use cudaFuncSetAttribute()

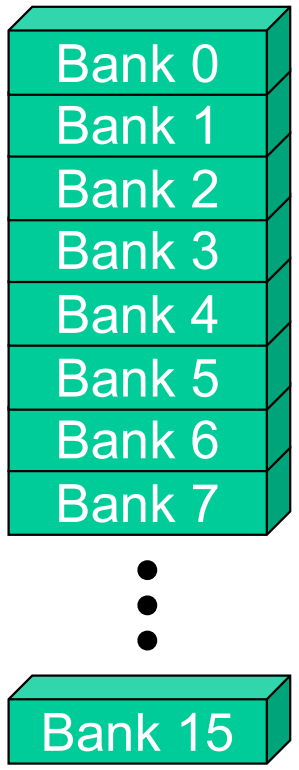
- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)
- Large SMEM can improve performance when:
  - Occupancy is limited by SMEM

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



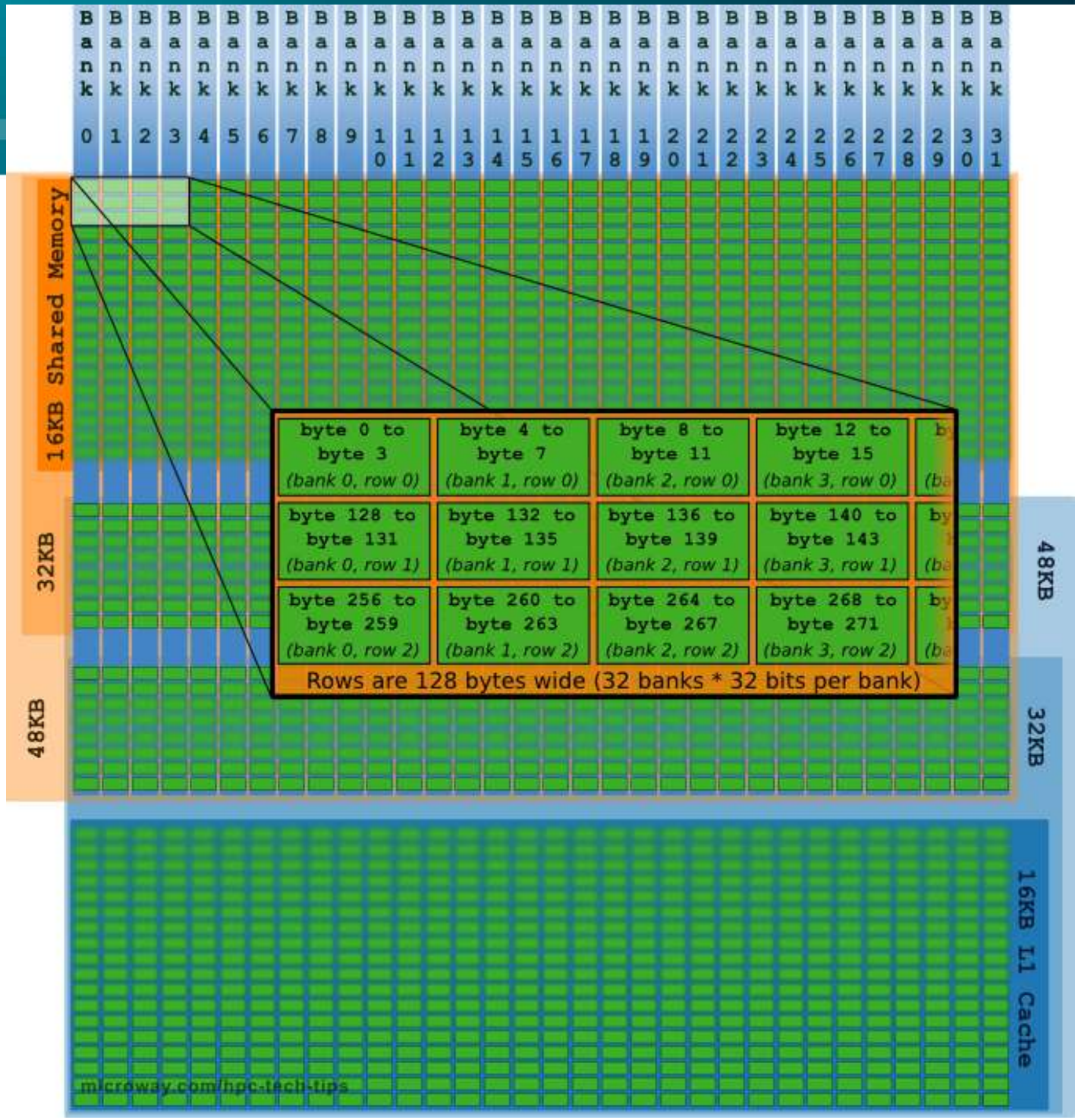
Memory Banks

Fermi/Kepler/Maxwell and newer:

32 banks

default:
4B / bank

Kepler or newer:
configurable
to 8B / bank

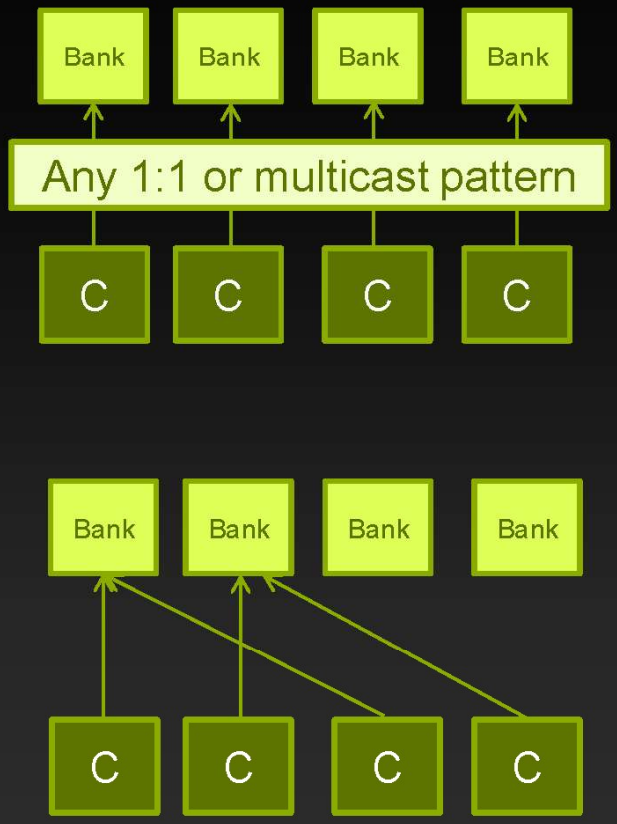


Shared Memory

- 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

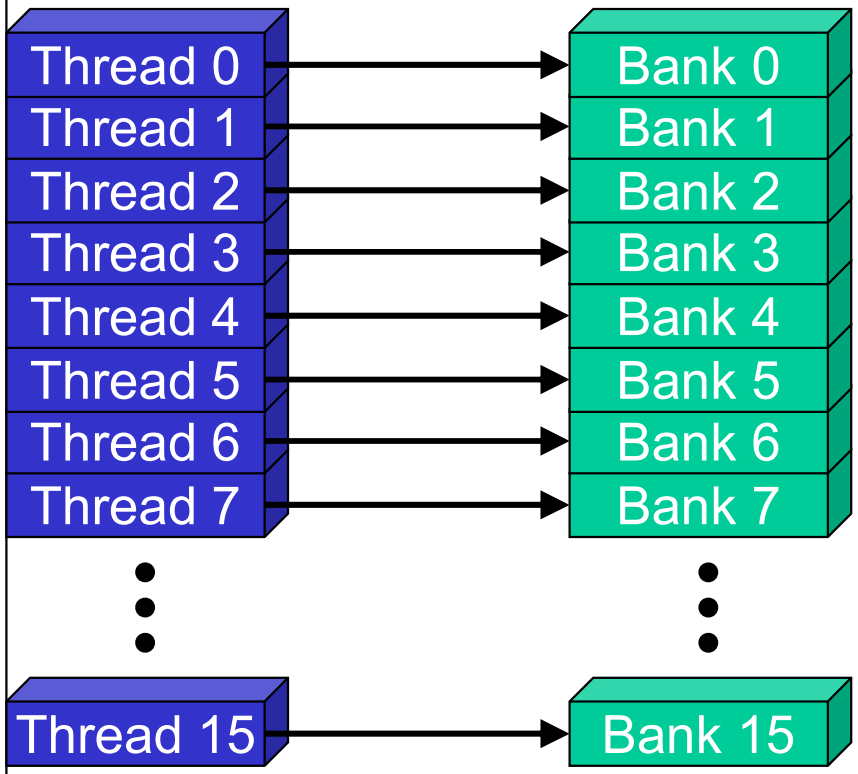
Shared Memory Organization

- 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

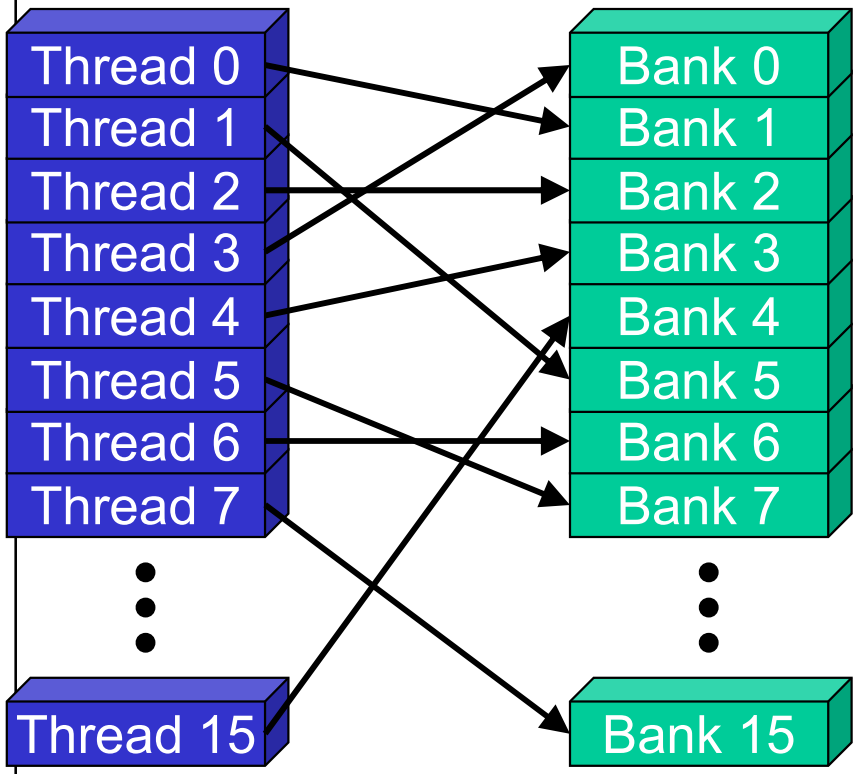


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

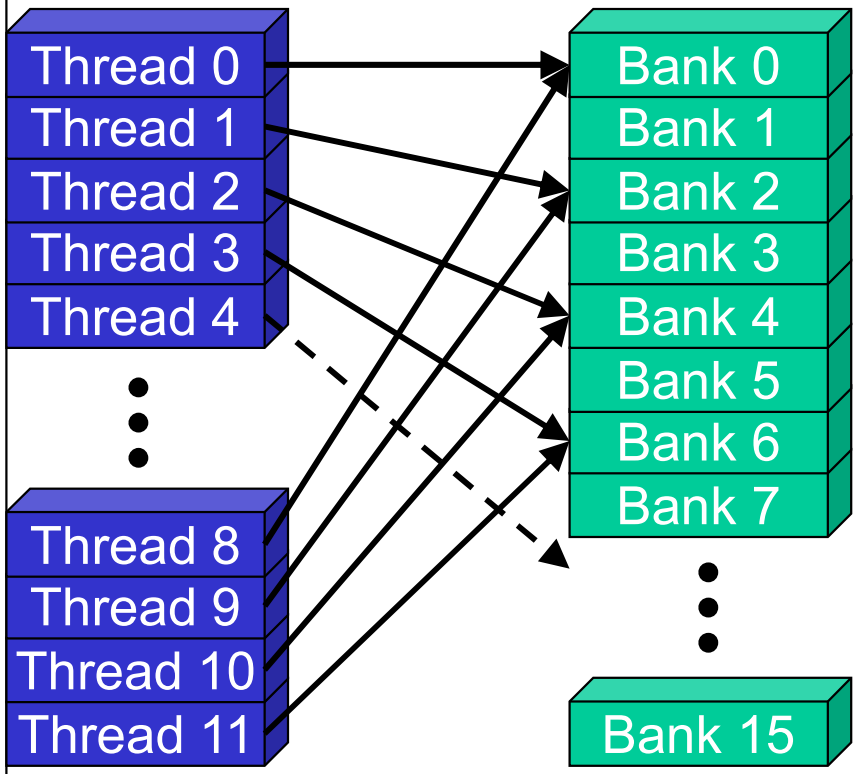


- No Bank Conflicts
  - Random 1:1 Permutation

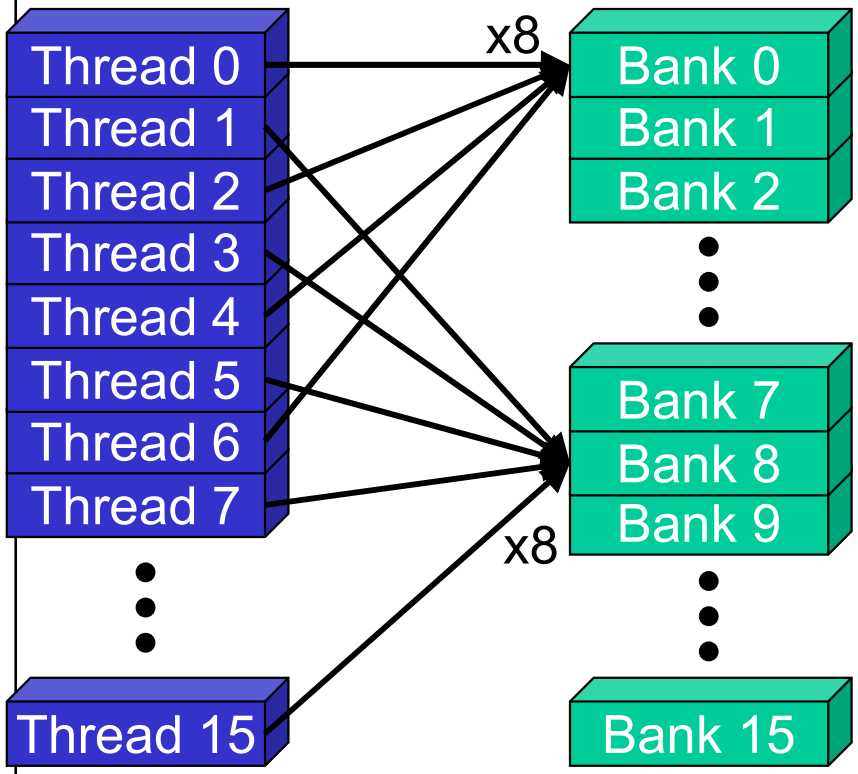


Bank Addressing Examples

- 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**

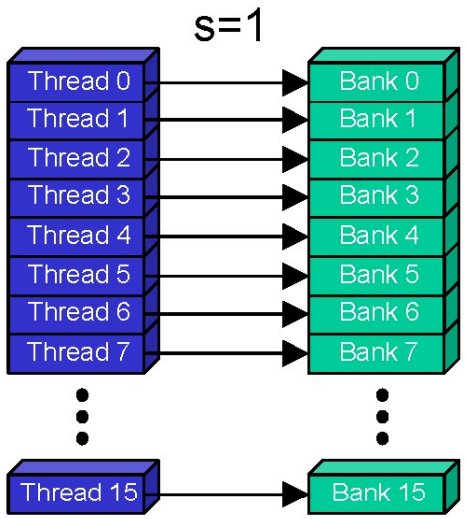
- **The fast case:**
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- **The slow case:**
  - Bank Conflict: multiple threads in the same half-warp access the same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

full warps instead of half warps on Fermi and newer!

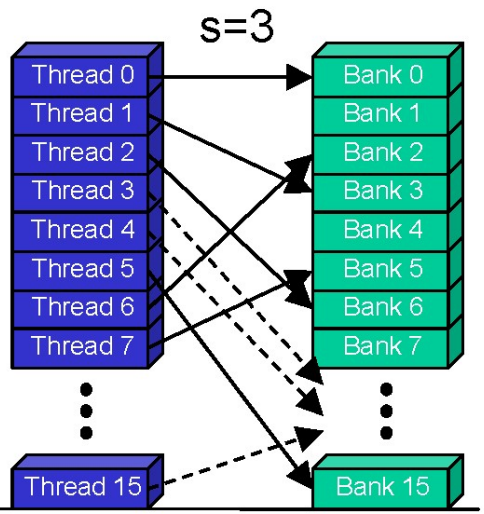
Linear Addressing

- **Given:**

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



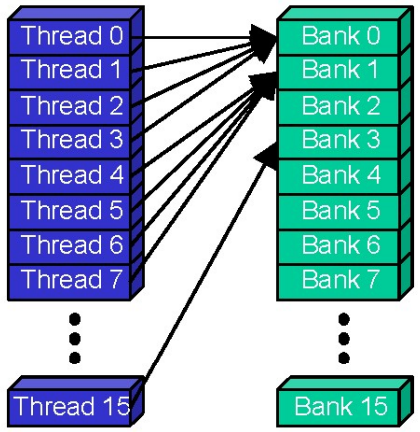
- **This is only bank-conflict-free if s shares no common factors with the number of banks**
  - 16 on G80, so `s` must be odd



Data Types and Bank Conflicts

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

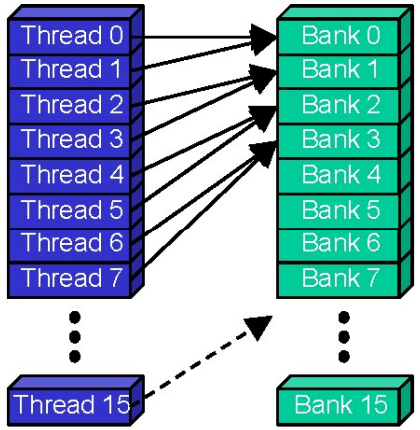
```
foo = shared[baseIndex + threadIdx.x]
```



- **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:

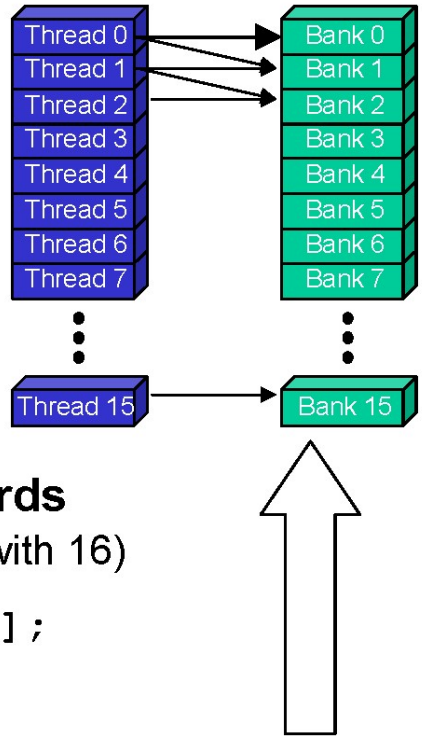
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```

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];
```

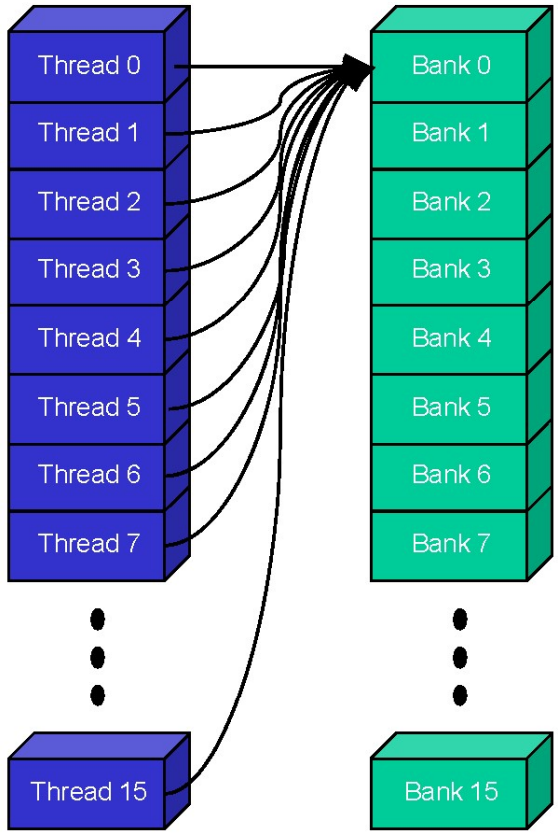
Broadcast on Shared Memory

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

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

- **Will be resolved implicitly**

multi-cast on Fermi and newer!

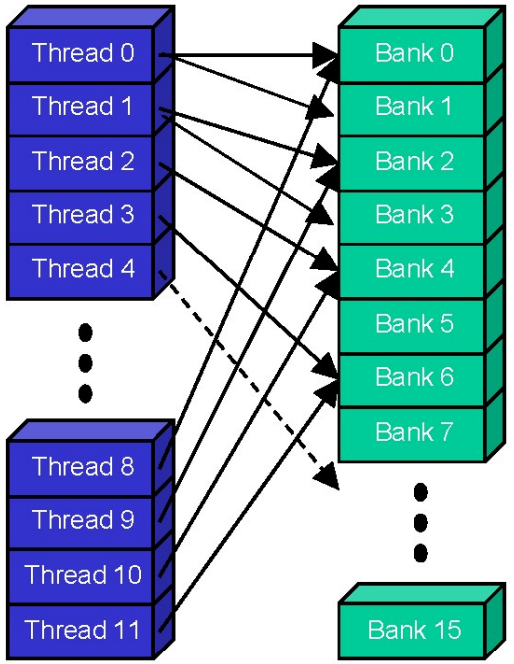


Common Array Bank Conflict Patterns 1D

- **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];
```

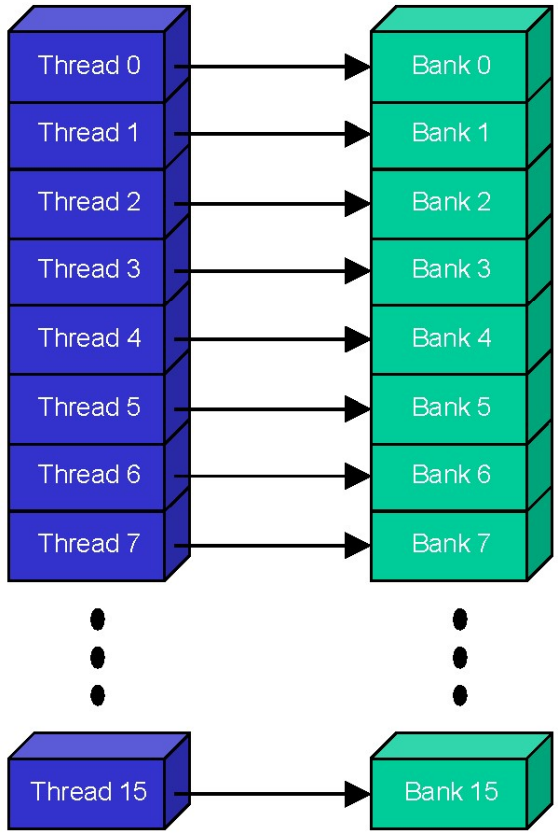
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects



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

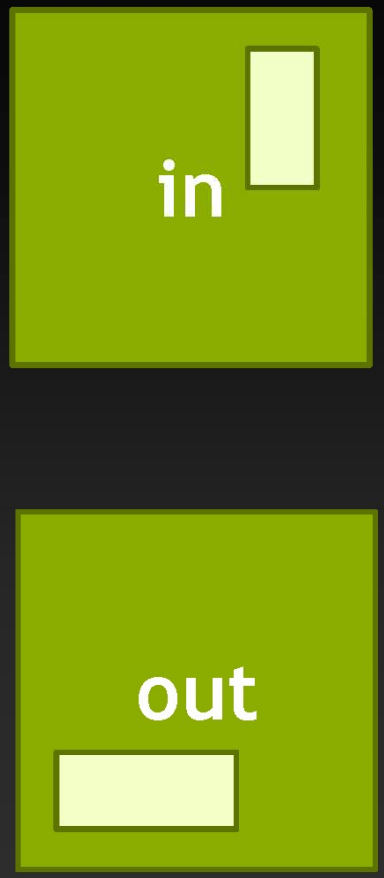
Kernel Optimizations: *Shared Memory Accesses*

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

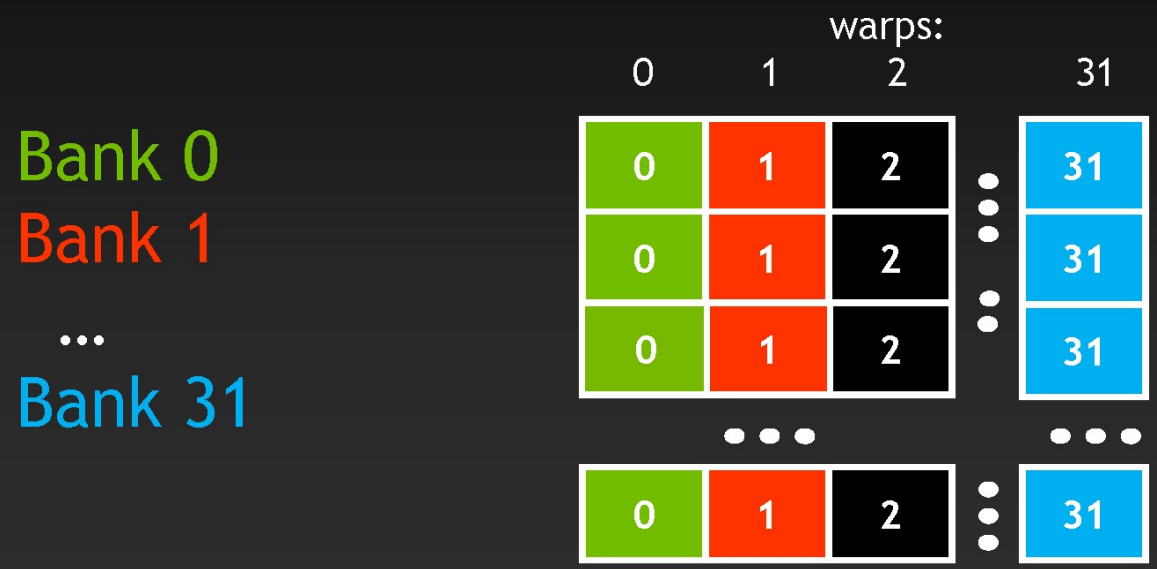
```
__global__ transpose(float in[], float out[])
{

  __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];

}
```

Fixed GMEM coalescing, but introduced SMEM bank conflicts

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

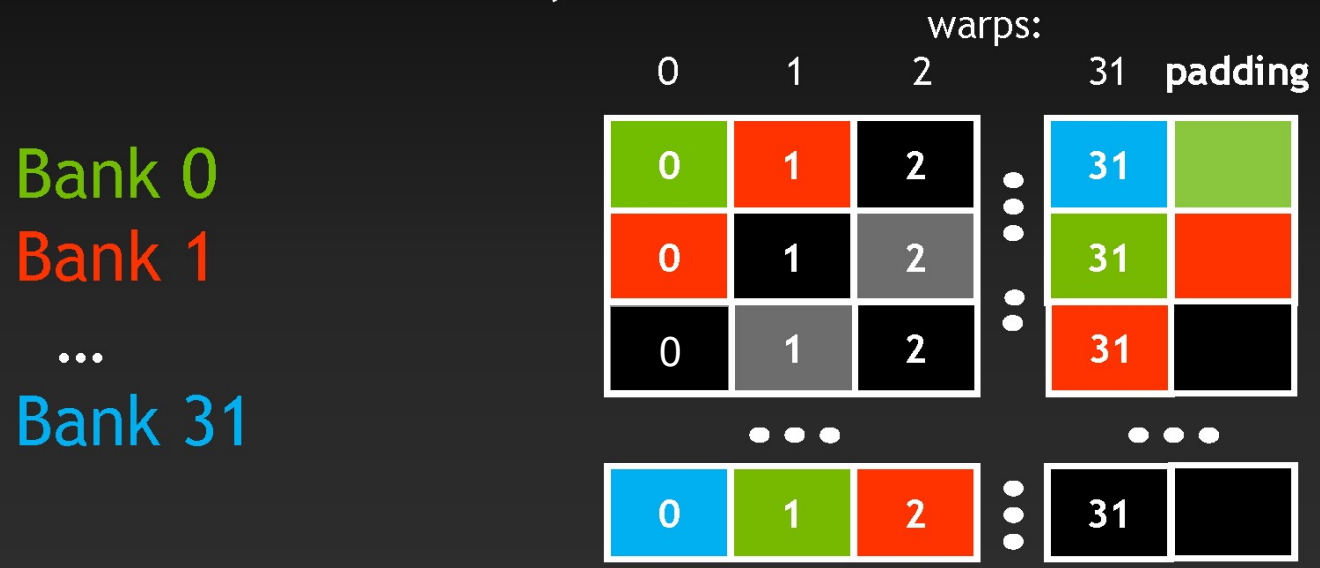
Shared Memory: Avoiding Bank Conflicts

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



Shared Memory: Avoiding Bank Conflicts

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



CUDA Memory: Uniforms & Textures

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

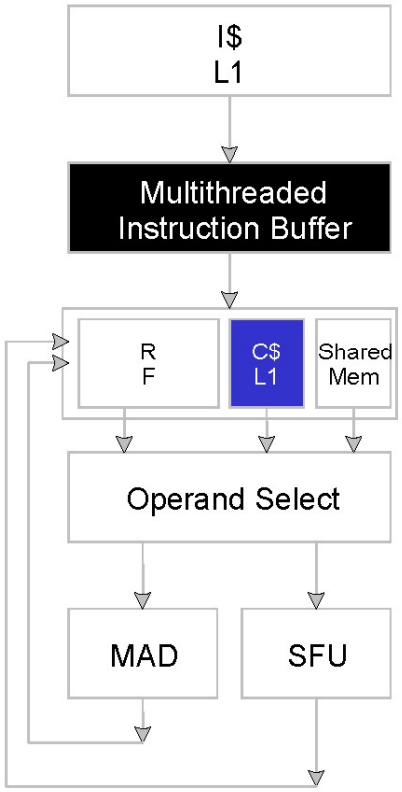
- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

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!




```
// specify as global variable
__device__ __constant__ float gpuGamma[2];
…
// copy gamma value to constant device memory
cudaMemcpyToSymbol(gpuGamma, &gamma, sizeof(float));
// access as global variable in kernel
res = gpuGamma[0] * threadIdx.x;
```

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

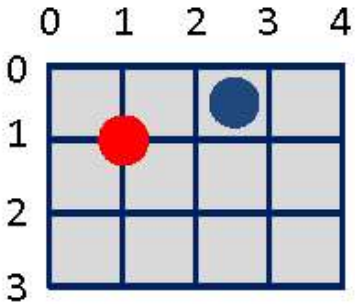
Texture Memory

- ***Cached***, potentially exhibiting higher bandwidth if there is locality in the texture fetches;
- They are not subject to the constraints on memory access patterns that global or constant memory reads must respect to get good performance
- The latency of addressing calculations is hidden better, possibly improving performance for applications that perform random accesses to the data
- No penalty when accessing float4
- Optional
  - 8-bit and 16-bit integer input data may be optionally converted to 32-bit floatingpoint
  - Packed data may be broadcast to separate variables in a single operation;
  - values in the range [0.0, 1.0] or [-1.0, 1.0]
  - texture filtering
  - address modes, e.g. wrapping / texture borders

Additional Texture Functionality

- **All of these are "free"**
  - Dedicated hardware
  - Must use CUDA texture objects
    - See CUDA Programming Guide for more details
    - Texture objects can interoperate graphics (OpenGL, DirectX)
- **Out-of-bounds index handling: clamp or wrap-around**
- **Optional interpolation**
  - Think: using fp indices for arrays
  - Linear, bilinear, trilinear
    - Interpolation weights are 9-bit
- **Optional format conversion**
  - {char, short, int, fp16} -> float

Examples of Texture Object Indexing



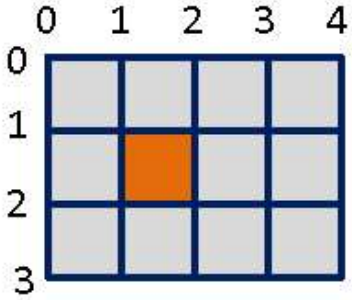
(2.5, 0.5)
(1.0, 1.0)

Integer indices fall between elements

Optional interpolation:

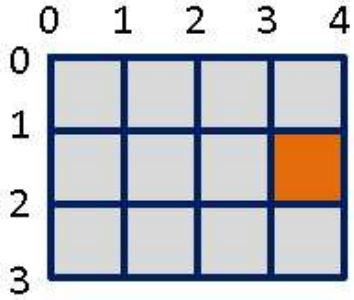
Weights are determined by coordinate distance

Index Wrap:



(5.5, 1.5)

Index Clamp:

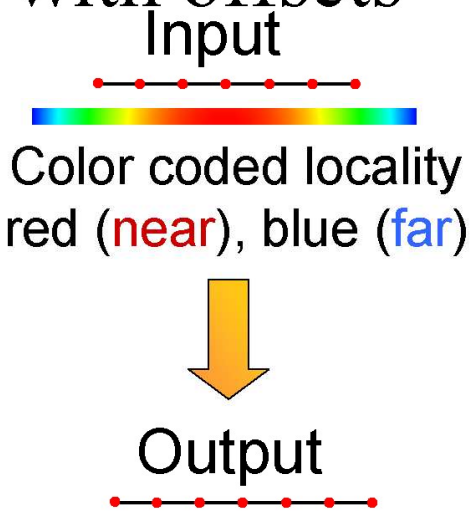


(5.5, 1.5)

Native Memory Layout – Data Locality

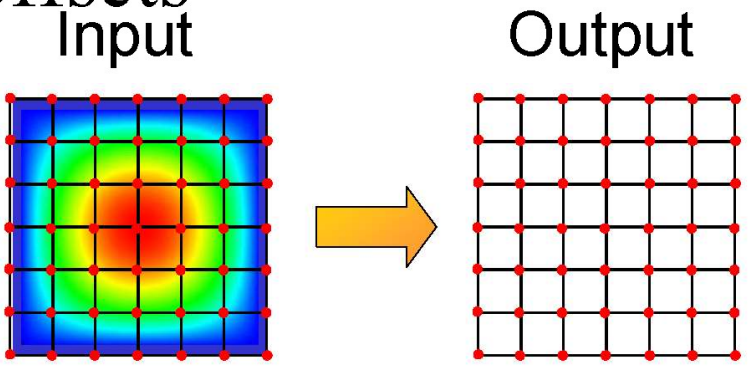
CPU

- 1D input
- 1D output
- Other dimensions with offsets



GPU

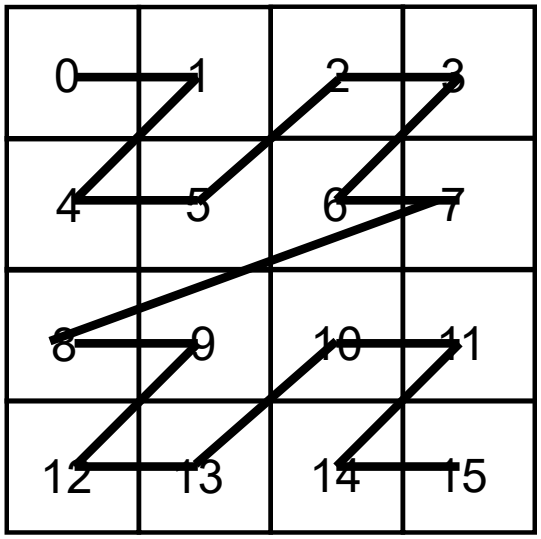
- 2D input
- 2D output
- Other dimensions with offsets



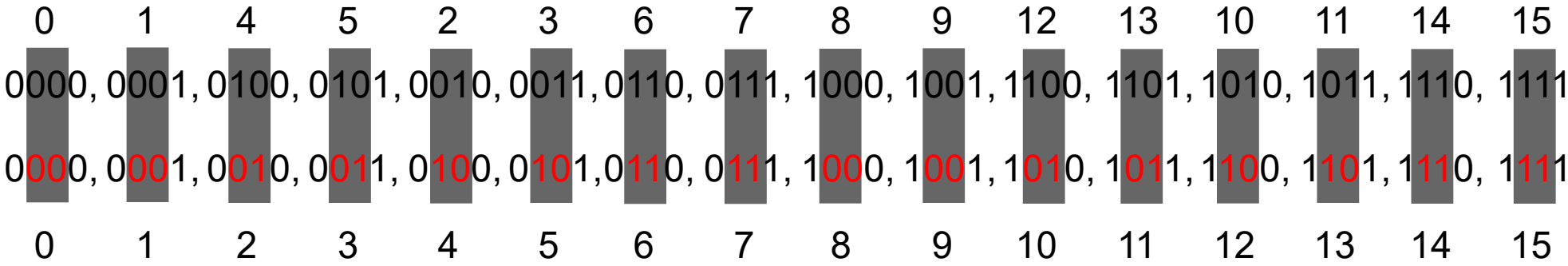
Space-Filling Curves: Morton Order (Z Order)

Map higher-dimensional space to 1D

- Z-order: Equivalent to quadtree (octree in 3D) depth-first traversal order



0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111



1D Access

- **Access to linear Cuda memory**

```
float4* pos; cudaMalloc( (void**)&pos, x*sizeof(float4) );
```

- **Texture reference**
  - type
  - access/filtering mode

```
// global texture reference
texture< float4, 1, cudaReadModeElementType> texPos;
```

- **Bind to linear array**

```
cudaBindTexture(0, texPos, pos, x*sizeof(float4)));
cudaUnbindTexture(texPos);
```

- **Within kernel**

```
float4 pa1 = tex1Dfetch( texPos, threadIdx.x);
```

- **Writing to a texture that is currently read by some threads is undefined!!!**

2D Access

- **Optimized for 2D / 3D locality**

```
texture< float4, 2, cudaReadModeElementType> texImg;
```

- **Requires binding to special *Array* memory – special memory layout**

```
cudaChannelFormatDesc floatTex =
cudaCreateChannelDesc<float4>();

float4* src;

cudaArray* img;

cudaMallocArray( &img, &floatTex, w, h);

cudaMemcpyToArray(img, 0, 0, src, w*h*sizeof(float4),
 cudaMemcpyHostToDevice);

cudaBindTextureToArray( texImg, img, floatTex) );

cudaUnbindTexture(texImg);
```

2D Access

- **Within kernel**

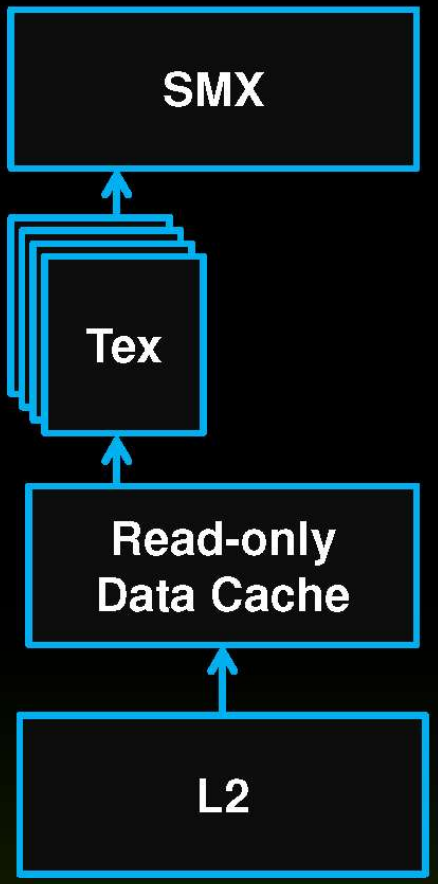
```
float4 r = tex2D( texImg, x +xoff, y+yoff);
```

- **Pros**
  - optimized for 2D locality (optimized memory layout / spacefilling curve)
- **Cons**
  - If the result of some kernel should be used as 2D texture `cudaMemcpyToArray` is required
  - You cannot write to a texture which is currently read from

- CUDA “surfaces” are writeable textures!

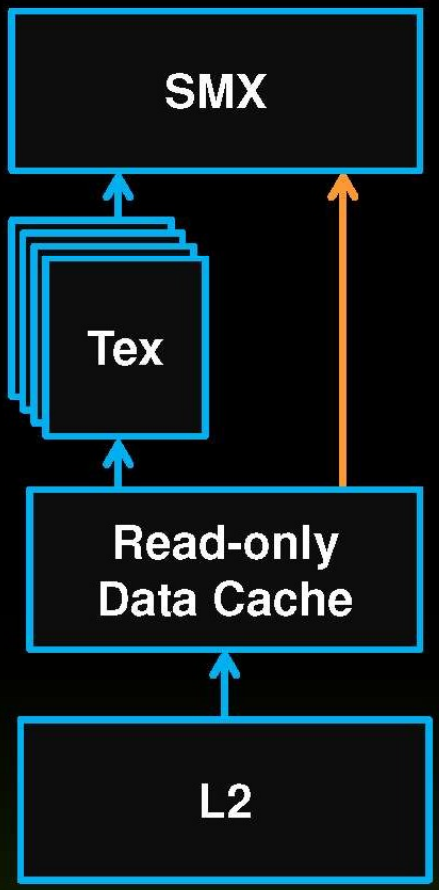
Texture performance

- Texture :
  - Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
  - Read-only data cache holds fetched samples
  - Backed up by the L2 cache
- SMX vs Fermi SM :
  - 4x filter ops per clock
  - 4x cache capacity



Texture Cache Unlocked

- Added a new path for compute
  - Avoids the texture unit
  - Allows a global address to be fetched and cached
  - Eliminates texture setup
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
- Managed automatically by compiler
  - “const __restrict” indicates eligibility



CUDA Memory: Global Memory

- Memory coalescing
- Cached memory access (L2 / L1)

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)

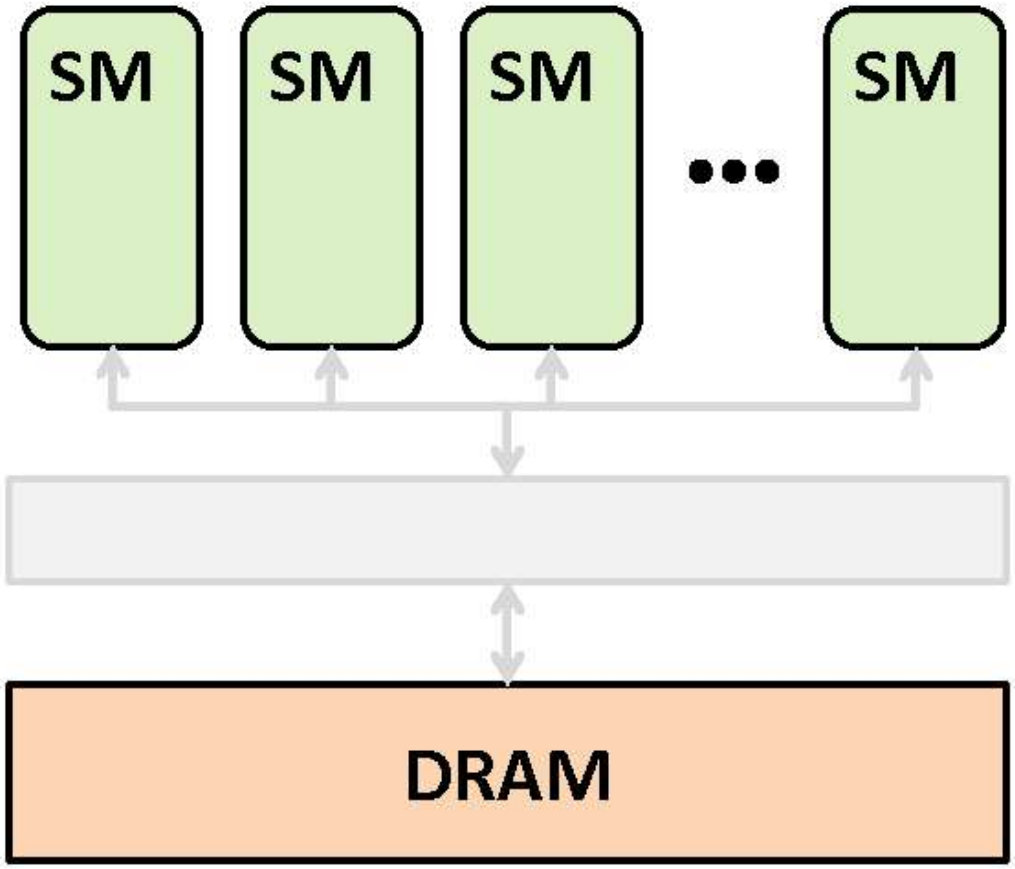
Shared memory

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

Constant memory: Constant (uniform) cache

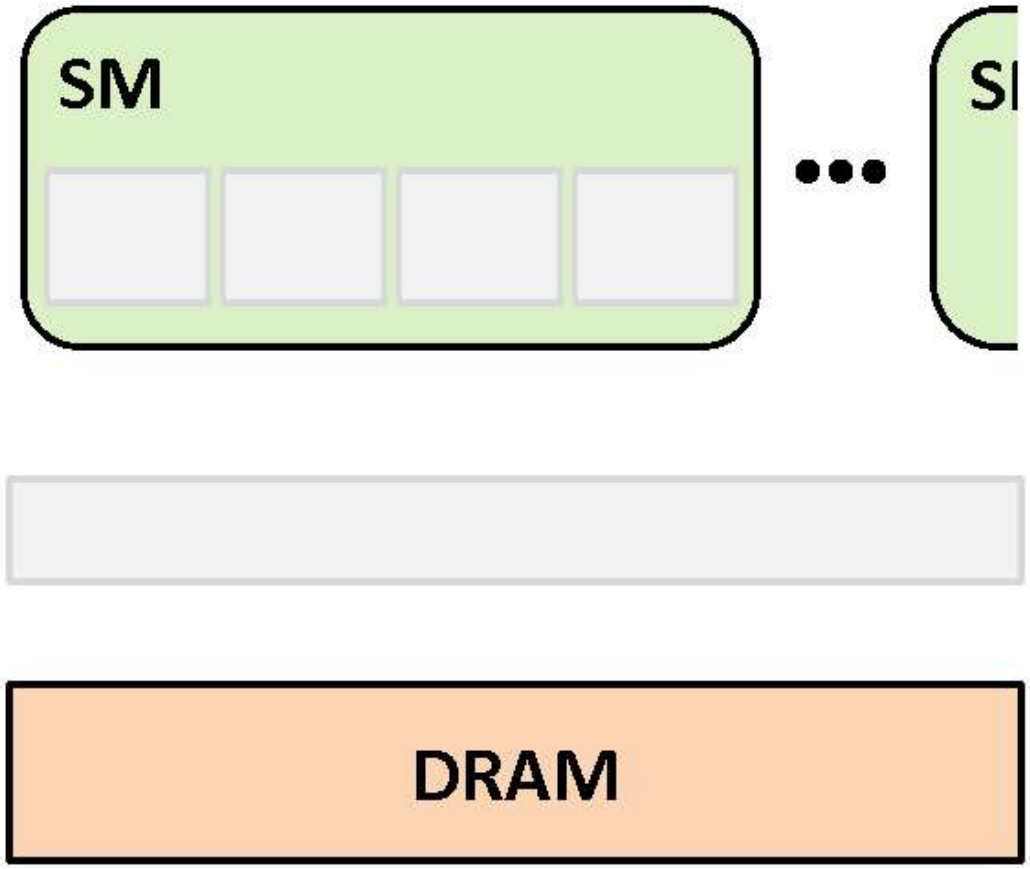
Unified memory programming: Device/host memory sharing

Maximize Byte Use



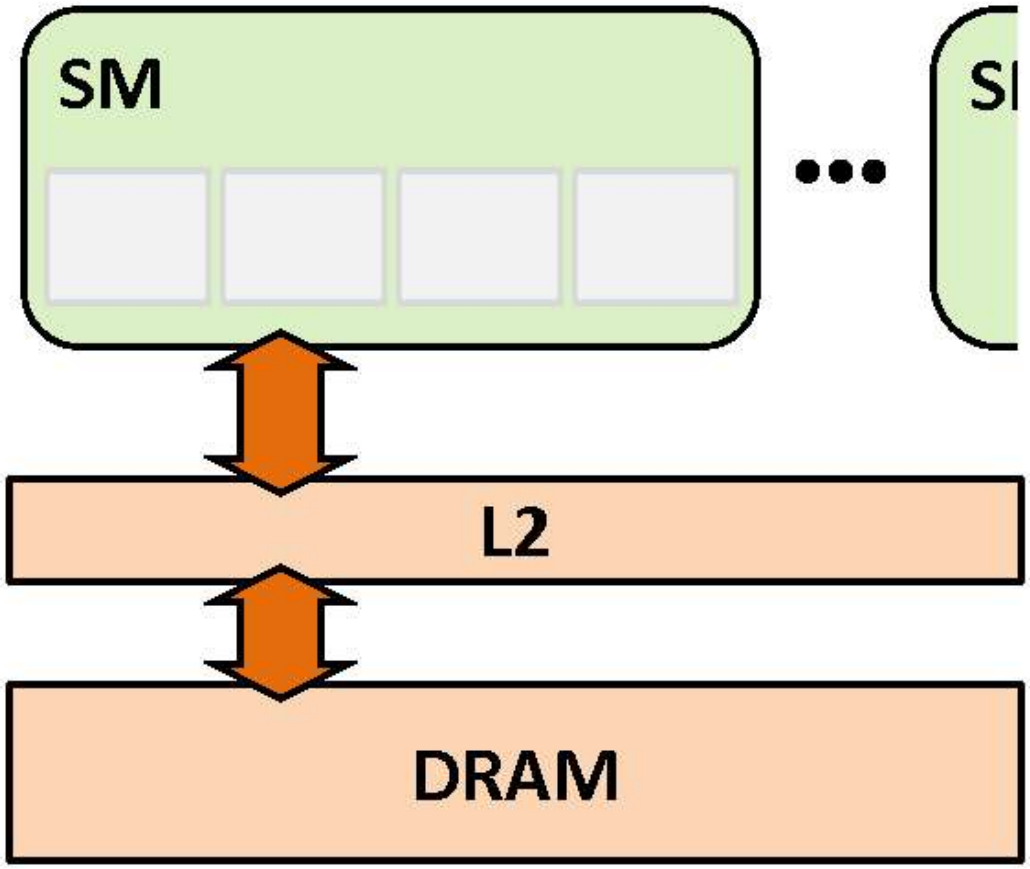
- **Two things to keep in mind:**
  - Memory accesses are per warp
  - Memory is accessed in discrete chunks
    - lines/segments
    - want to make sure that bytes that travel from DRAM to SMs get used
      - For that we should understand how memory system works
- **Note: not that different from CPUs**
  - x86 needs SSE/AVX memory instructions to maximize performance

GPU Memory System



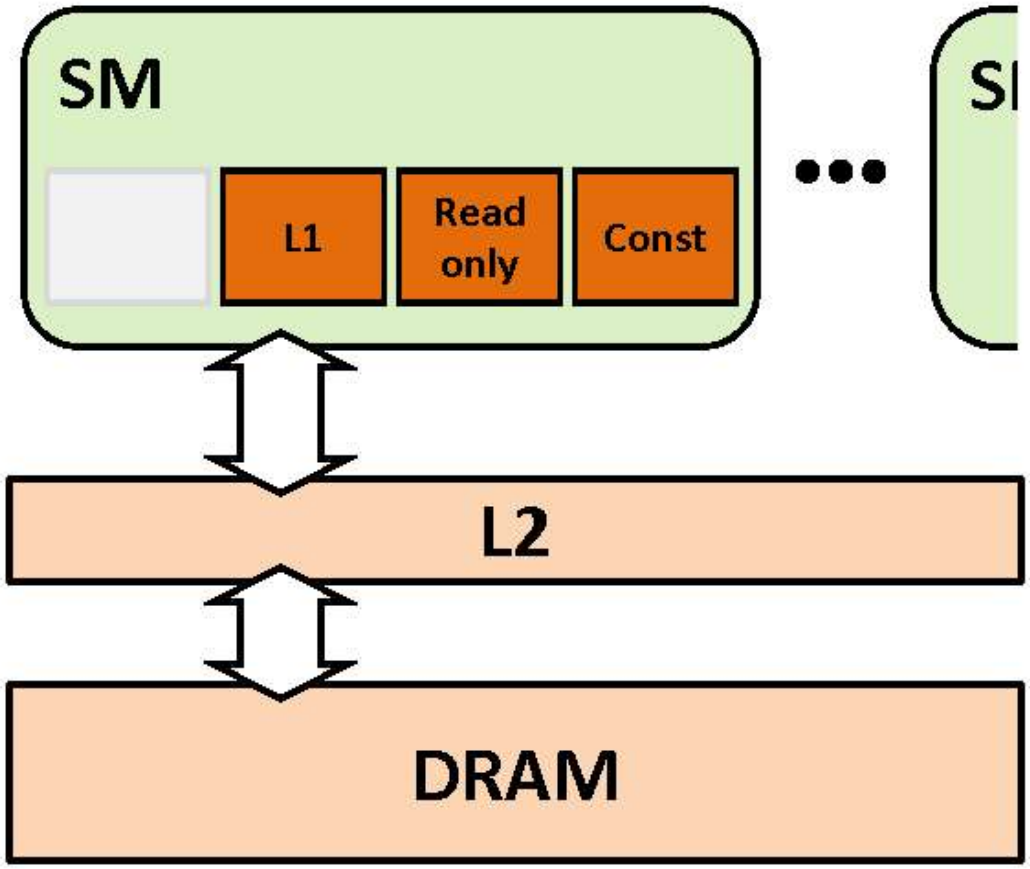
- **All data lives in DRAM**
  - Global memory
  - Local memory
  - Textures
  - Constants

GPU Memory System



- All DRAM accesses go through L2
- Including copies:
  - P2P
  - CPU-GPU

GPU Memory System



- Once in an SM, data goes into one of 3 caches/buffers
- Programmer's choice
  - ~~L1 is the "default"~~
  - Read-only, Const require explicit code

Access Path

- **L1 path**
  - Global memory
    - Memory allocated with cudaMalloc()
    - Mapped CPU memory, peer GPU memory
    - Globally-scoped arrays qualified with __global__
  - Local memory
    - allocation/access managed by compiler so we'll ignore
- **Read-only/TEX path**
  - Data in texture objects, CUDA arrays
  - CC 3.5 and higher:
    - Global memory accessed via intrinsics (or specially qualified kernel arguments)
- **Constant path**
  - Globally-scoped arrays qualified with __constant__

Access Via L1

- **Natively supported word sizes per thread:**
  - 1B, 2B, 4B, 8B, 16B
    - Addresses must be aligned on word-size boundary
  - Accessing types of other sizes will require multiple instructions
- **Accesses are processed per warp**
  - Threads in a warp provide 32 addresses
    - Fewer if some threads are inactive
  - HW converts addresses into memory transactions
    - Address pattern may require multiple transactions for an instruction
    - If ***N*** transactions are needed, there will be (***N-1***) replays of the instruction

Global Memory Access

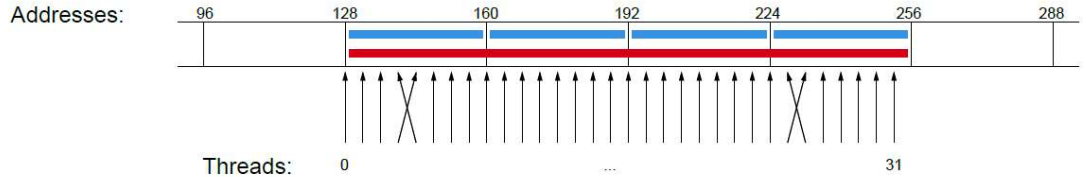
all recent
compute capabilities
(- 9.x)

Beware:

Uncached here means
not cached in L1

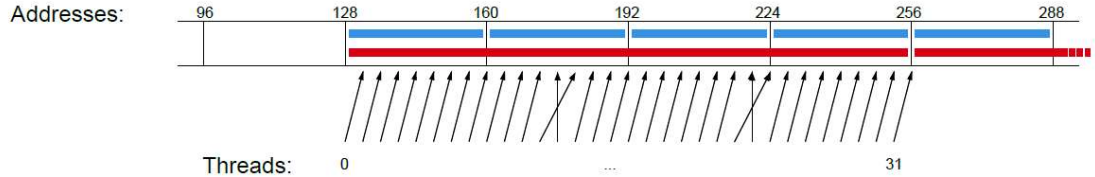
the L2 cache is
***always* used!**

Aligned accesses (sequential/non-sequential)



Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis-aligned accesses (sequential/non-sequential)



Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Compute Capab. 3.x (Kepler, Part 1)

K.3.2. Global Memory

Global memory accesses for devices of compute capability 3.x are cached in L2 and for devices of compute capability 3.5 or 3.7, may also be cached in the read-only data cache described in the previous section; they are normally not cached in L1. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory accesses in L1 via the `-Xptxas -dlcm=ca` option to `nvcc`.

A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions, whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses.

If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently:

- Two memory requests, one for each half-warp, if the size is 8 bytes,
- Four memory requests, one for each quarter-warp, if the size is 16 bytes.

Compute Capab. 3.x (Kepler, Part 2)

Each memory request is then broken down into cache line requests that are issued independently. A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Note that threads can access any words in any order, including the same words.

If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined.

Data that is read-only for the entire lifetime of the kernel can also be cached in the read-only data cache described in the previous section by reading it using the `__ldg()` function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Figure 21 shows some examples of global memory accesses and corresponding memory transactions.

Compute Capab. 5.x (Maxwell, Part 1)

K.4.2. Global Memory

Global memory accesses are always cached in L2 and caching in L2 behaves in the same way as for devices of compute capability 3.x (see Global Memory).

Data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache described in the previous section by reading it using the `__ldg()` function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

Compute Capab. 5.x (Maxwell, Part 2)

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

- Perform the read using inline assembly with the appropriate modifier as described in the PTX reference manual;
- Compile with the `-Xptxas -dlcm=ca` compilation flag, in which case all reads are cached, except reads that are performed using inline assembly with a modifier that disables caching;
- Compile with the `-Xptxas -fscm=ca` compilation flag, in which case all reads are cached, including reads that are performed using inline assembly regardless of the modifier used.

When caching is enabled using one of the three mechanisms listed above, devices of compute capability 5.2 will cache global memory reads in the unified L1/texture cache for all kernel launches except for the kernel launches for which thread blocks consume too much of the SM's register file. These exceptions are reported by the profiler.

PTX State Spaces (1)

Memory type/access etc. organized using notion of *state spaces*

Table 6 State Spaces

Name	Description
`.reg`	Registers, fast.
`.sreg`	Special registers. Read-only; pre-defined; platform-specific.
`.const`	Shared, read-only memory.
`.global`	Global memory, shared by all threads.
`.local`	Local memory, private to each thread.
`.param`	Kernel parameters, defined per-grid; or Function or local parameters, defined per-thread.
`.shared`	Addressable memory shared between threads in 1 CTA.
`.tex`	Global texture memory (deprecated).

PTX State Spaces (2)

Table 7 Properties of State Spaces

Name	Addressable	Initializable	Access	Sharing
`.reg`	No	No	R/W	per-thread
`.sreg`	No	No	RO	per-CTA
`.const`	Yes	Yes[1]	RO	per-grid
`.global`	Yes	Yes[1]	R/W	Context
`.local`	Yes	No	R/W	per-thread
`.param` (as input to kernel)	Yes[2]	No	RO	per-grid
`.param` (used in functions)	Restricted[3]	No	R/W	per-thread
`.shared`	Yes	No	R/W	per-CTA
`.tex`	No[4]	Yes, via driver	RO	Context

Notes:

[1] Variables in `.const` and `.global` state spaces are initialized to zero by default.

[2] Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

[3] Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

[4] Accessible only via the `tex` instruction.

PTX Cache Operators

Table 27 Cache Operators for Memory Load Instructions

Operator	Meaning
`.ca`	Cache at all levels, likely to be accessed again. The default load instruction cache operation is ld.ca, which allocates cache lines in all levels (L1 and L2) with normal eviction policy. Global data is coherent at the L2 level, but multiple L1 caches are not coherent for global data. If one thread stores to global memory via one L1 cache, and a second thread loads that address via a second L1 cache with `ld.ca`, the second thread may get stale L1 cache data, rather than the data stored by the first thread. The driver must invalidate global L1 cache lines between dependent grids of parallel threads. Stores by the first grid program are then correctly fetched by the second grid program issuing default `ld.ca` loads cached in L1.
`.cg`	Cache at global level (cache in L2 and below, not L1). Use `ld.cg` to cache loads only globally, bypassing the L1 cache, and cache only in the L2 cache.
`.cs`	Cache streaming, likely to be accessed once. The `ld.cs` load cached streaming operation allocates global lines with evict-first policy in L1 and L2 to limit cache pollution by temporary streaming data that may be accessed once or twice. When `ld.cs` is applied to a Local window address, it performs the `ld.lu` operation.
`.lu`	Last use. The compiler/programmer may use `ld.lu` when restoring spilled registers and popping function stack frames to avoid needless write-backs of lines that will not be used again. The `ld.lu` instruction performs a load cached streaming operation (`ld.cs`) on global addresses.
`.cv`	Don't cache and fetch again (consider cached system memory lines stale, fetch again). The ld.cv load operation applied to a global System Memory address invalidates (discards) a matching L2 line and re-fetches the line on each new load.

SASS LD/ST Instructions

Architecture-dep.

Kepler:

Compute Load/Store Instructions	
LDC	Load from Constant
LD	Load from Memory
LDG	Non-coherent Global Memory Load
LDL	Load from Local Memory
LDS	Load from Shared Memory
LDSLK	Load from Shared Memory and Lock
ST	Store to Memory
STL	Store to Local Memory
STS	Store to Shared Memory
STSCUL	Store to Shared Memory Conditionally and Unlock
ATOM	Atomic Memory Operation
RED	Atomic Memory Reduction Operation
CCTL	Cache Control
CCTLL	Cache Control (Local)
MEMBAR	Memory Barrier

(see also LDG.CI etc.)

Compute Capab. 6.x (Pascal)

K.5.2. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See Global Memory).

Compute Capab. 7.x (Volta/Turing)

K.6.3. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See Global Memory).

Compute Capab. 8.x (Ampere/Ada)

K.7.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See Global Memory).

Compute Capab. 9.x (Hopper)

K.8.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See Global Memory).

Vectorized Memory Access

See **https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/**

```
__global__ void device_copy_vector2_kernel(int* d_in, int* d_out, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i = idx; i < N/2; i += blockDim.x * gridDim.x) {
    reinterpret_cast<int2*>(d_out)[i] = reinterpret_cast<int2*>(d_in)[i];
  }

  // in only one thread, process final element (if there is one)
  if (idx==N/2 && N%2==1)
    d_out[N-1] = d_in[N-1];
}

void device_copy_vector2(int* d_in, int* d_out, int n) {
  threads = 128;
  blocks = min((N/2 + threads-1) / threads, MAX_BLOCKS);

  device_copy_vector2_kernel<<<blocks, threads>>>(d_in, d_out, N);
}
```

```
/*0088*/                   IMAD R10.CC, R3, R5, c[0x0][0x140]
/*0090*/                   IMAD.HI.X R11, R3, R5, c[0x0][0x144]
/*0098*/                   IMAD R8.CC, R3, R5, c[0x0][0x148]
/*00a0*/                   LD.E.64 R6, [R10]
/*00a8*/                   IMAD.HI.X R9, R3, R5, c[0x0][0x14c]
/*00c8*/                   ST.E.64 [R8], R6
```

SASS

LD.E.64, LD.E.128,
ST.E.64, ST.E.128

Vectorized Memory Access

See **https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/**

```
__global__ void device_copy_vector4_kernel(int* d_in, int* d_out, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for(int i = idx; i < N/4; i += blockDim.x * gridDim.x) {
    reinterpret_cast<int4*>(d_out)[i] = reinterpret_cast<int4*>(d_in)[i];
  }

  // in only one thread, process final elements (if there are any)
  int remainder = N%4;
  if (idx==N/4 && remainder!=0) {
    while(remainder) {
      int idx = N - remainder--;
      d_out[idx] = d_in[idx];
    }
  }
}

void device_copy_vector4(int* d_in, int* d_out, int N) {
  int threads = 128;
  int blocks = min((N/4 + threads-1) / threads, MAX_BLOCKS);

  device_copy_vector4_kernel<<<blocks, threads>>>(d_in, d_out, N);
}
```

```
/*0090*/                   IMAD R10.CC, R3, R13, c[0x0][0x140]
/*0098*/                   IMAD.HI.X R11, R3, R13, c[0x0][0x144]
/*00a0*/                   IMAD R8.CC, R3, R13, c[0x0][0x148]
/*00a8*/                   LD.E.128 R4, [R10]
/*00b0*/                   IMAD.HI.X R9, R3, R13, c[0x0][0x14c]
/*00d0*/                   ST.E.128 [R8], R4
```

SASS

LD.E.64, LD.E.128,
ST.E.64, ST.E.128

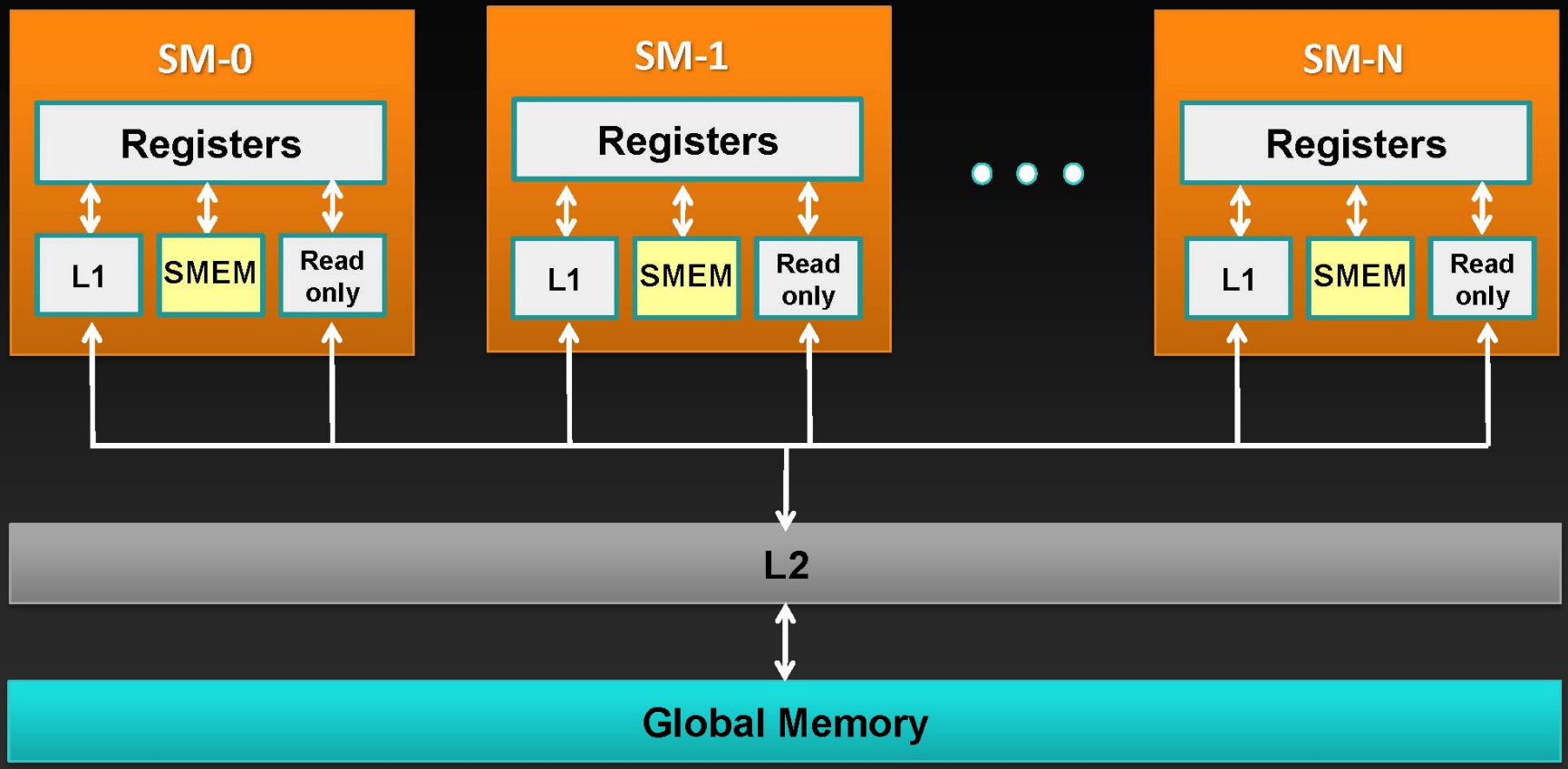
GMEM Writes

- **Not cached in the SM**
  - Invalidate the line in L1, go to L2
- **Access is at 32 B segment granularity**
- **Transaction to memory: 1, 2, or 4 segments**
  - Only the required segments will be sent
- **If multiple threads in a warp write to the same address**
  - One of the threads will "win"
  - Which one is not defined

OPTIMIZE

Kernel Optimizations: *Global Memory Throughput*

Kepler Memory Hierarchy



Load Operation

- Memory operations are issued per warp (32 threads)
  - Just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

Memory Throughput Analysis

- Two perspectives on the throughput:
  - Application's point of view:
    - count only bytes requested by application
  - HW point of view:
    - count all bytes moved by hardware
- The two views can be different:
  - Memory is accessed at 32 byte granularity
    - Scattered/offset pattern: application doesn't use all the hw transaction bytes
  - Broadcast: the same small transaction serves many threads in a warp
- Two aspects to inspect for performance impact:
  - Address pattern
  - Number of concurrent accesses in flight

Global Memory Operation

- Memory operations are executed per warp
  - 32 threads in a warp provide memory addresses
  - Hardware determines into which lines those addresses fall
    - Memory transaction granularity is 32 bytes
    - There are benefits to a warp accessing a contiguous aligned region of 128 or 256 bytes
- Access word size
  - Natively supported sizes (per thread): 1, 2, 4, 8, 16 bytes
    - Assumes that each thread's address is aligned on the word size boundary
  - If you are accessing a data type that's of non-native size, compiler will generate several load or store instructions with native sizes

Access Patterns vs. Memory Throughput

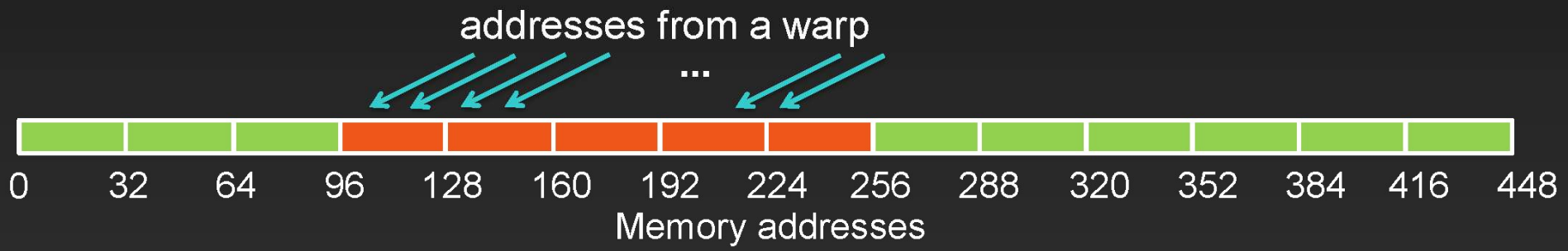
- Scenario:
  - Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus
  - Bus utilization: 100%

Access Patterns vs. Memory Throughput

- Scenario:
  - Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus
  - Bus utilization: 100%

Access Patterns vs. Memory Throughput

- Scenario:
  - Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization

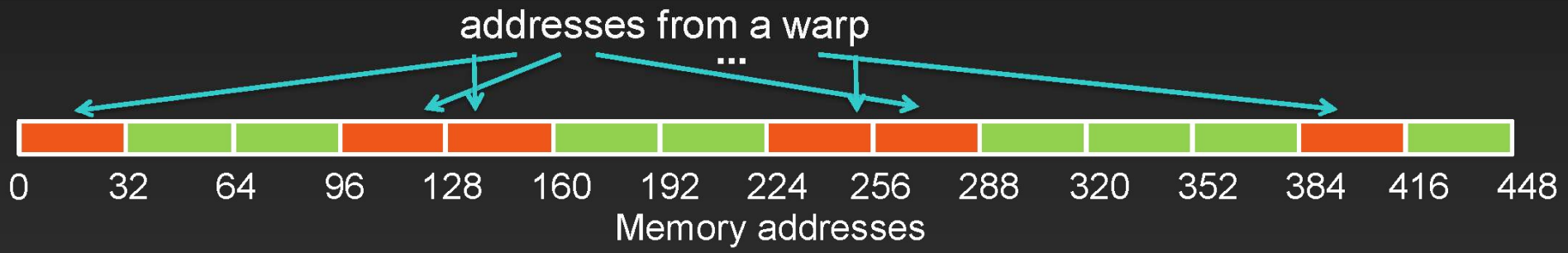


Access Patterns vs. Memory Throughput

- Scenario:
  - All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus
  - Bus utilization: 12.5%

Access Patterns vs. Memory Throughput

- Scenario:
  - Warp requests 32 scattered 4-byte words
- Addresses fall within *N* segments
  - Warp needs 128 bytes
  - *N**32 bytes move across the bus
  - Bus utilization: 128 / (*N**32)



Structures of Non-Native Size

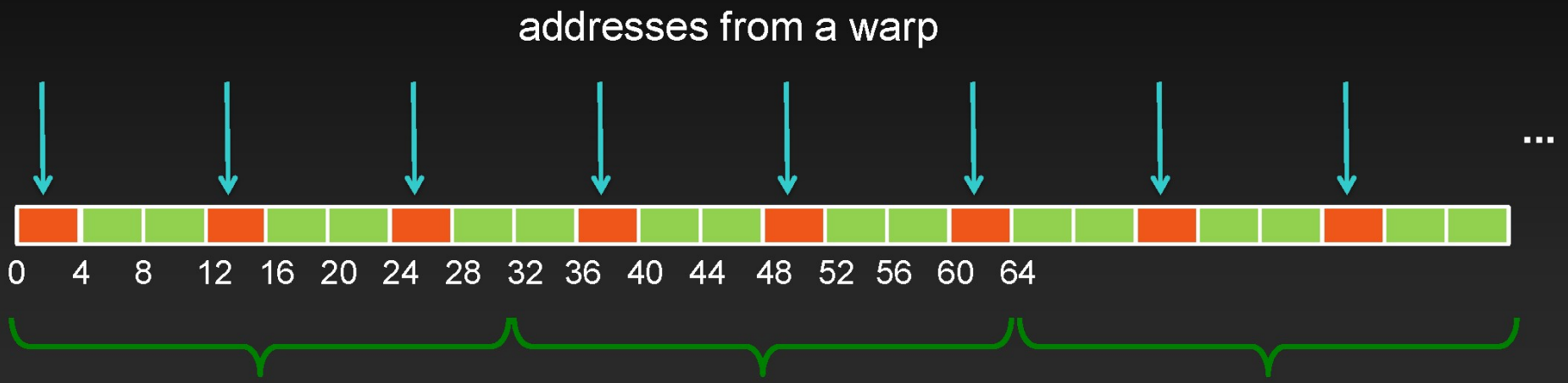
- Say we are reading a 12-byte structure per thread

```
struct Position
{
  float x, y, z;
};
...
__global__ void kernel( Position *data, ... )
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  Position temp = data[idx];
  ...
}
```

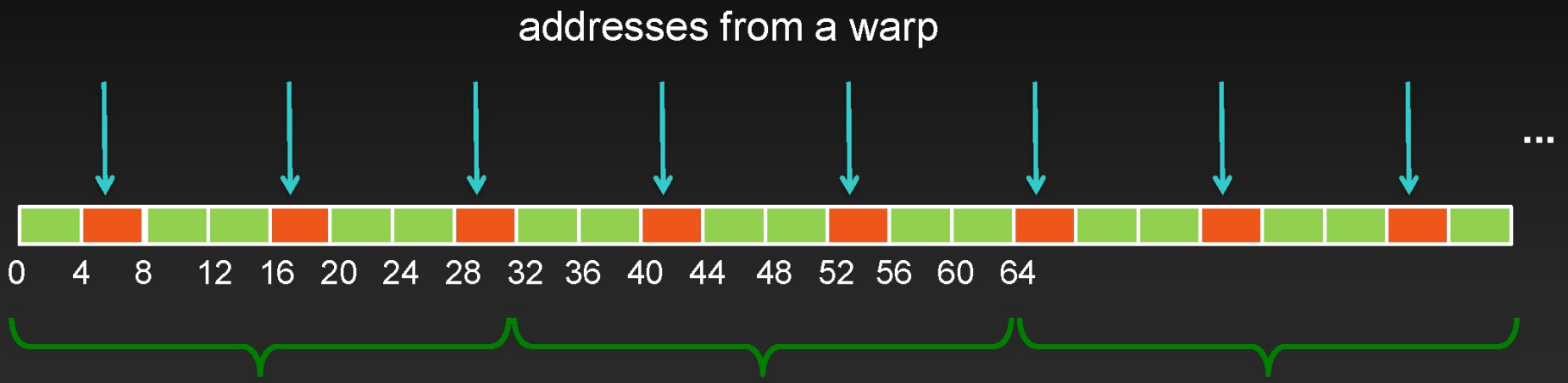
Structure of Non-Native Size

- Compiler converts `temp = data[idx]` into 3 loads:
  - Each loads 4 bytes
  - Can't do an 8 and a 4 byte load: 12 bytes per element means that every other element wouldn't align the 8-byte load on 8-byte boundary
- Addresses per warp for each of the loads:
  - Successive threads read 4 bytes at 12-byte stride

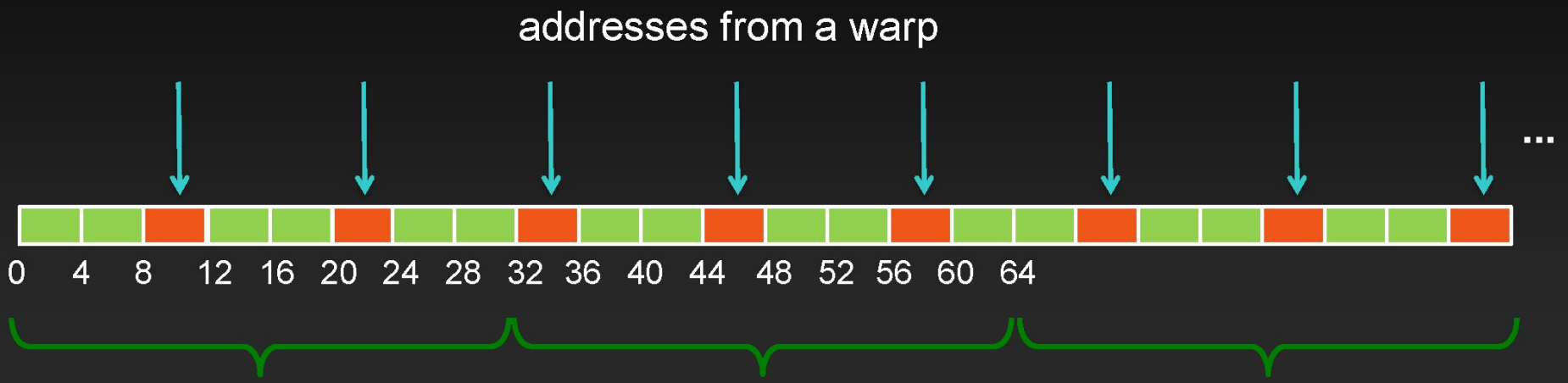
First Load Instruction



Second Load Instruction



Third Load Instruction

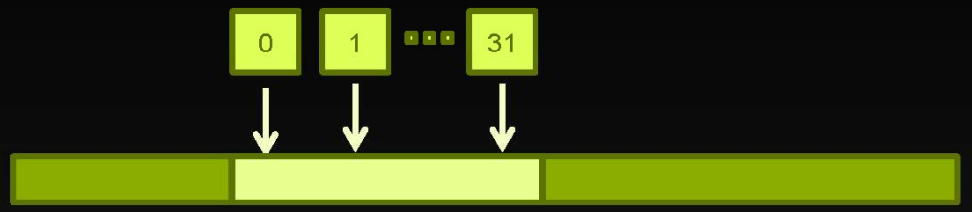


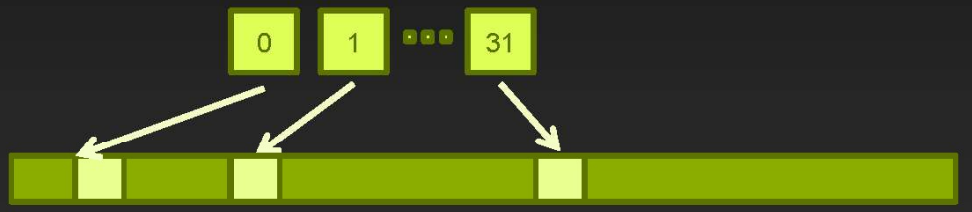
Performance and Solutions

- Because of the address pattern, we end up moving 3x more bytes than application requests
  - We waste a lot of bandwidth, leaving performance on the table
- Potential solutions:
  - Change data layout from array of structures to structure of arrays
    - In this case: 3 separate arrays of floats
    - The most reliable approach (also ideal for both CPUs and GPUs)
  - Use loads via read-only cache
    - As long as lines survive in the cache, performance will be nearly optimal
  - Stage loads via shared memory

Global Memory Access Patterns

- SoA vs AoS:
  - Good: point.x[i]
  - Not so good: point[i].x
- Strided array access:
  - ~OK: x[i] = a[i+1] – a[i]
  - Slower: x[i] = a[64*i] – a[i]
- Random array access:
  - Slower: a[rand(i)]



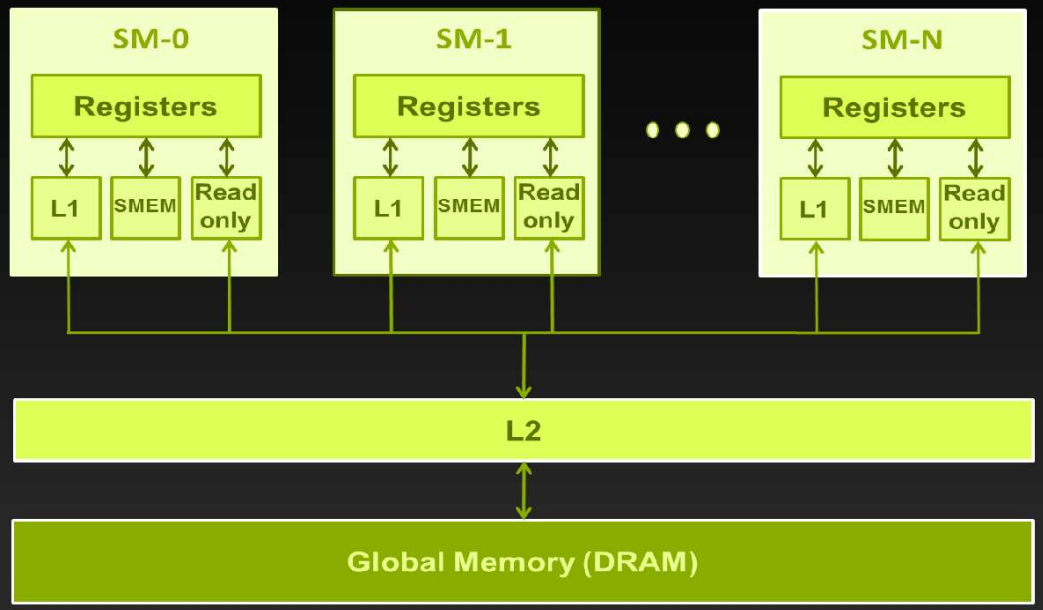


Summary: GMEM Optimization

- Strive for perfect address coalescing per warp
  - Align starting address (may require padding)
  - A warp will ideally access within a contiguous region
  - Avoid scattered address patterns or patterns with large strides between threads
- Analyze and optimize address patterns:
  - Use profiling tools (included with CUDA toolkit download)
  - Compare the transactions per request to the ideal ratio
  - Choose appropriate data layout (prefer SoA)
  - If needed, try read-only loads, staging accesses via SMEM

A note about caches

- L1 and L2 caches
  - Ignore in software design
  - Thousands of concurrent threads – cache blocking difficult at best
- Read-only Data Cache
  - Shared with texture pipeline
  - Useful for uncoalesced reads
  - Handled by compiler when `const __restrict__` is used, or use `__ldg()` primitive



Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: const __restrict__
    - All other pointer arguments: __restrict__
      - Conveys to compiler that no aliasing will occur
  - Using __ldg() intrinsic
    - Requires no pointer decoration

Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer argumen
    - Pointer of interest: const
    - All other pointer argumen
      - Conveys to compiler t
  - Using __ldg() intrinsic
    - Requires no pointer deco

```
__global__ void kernel(
                int* __restrict__ output,
          const int* __restrict__ input )
{
      ...
      output[idx] = input[idx];
}
```

Read-only Data Cache

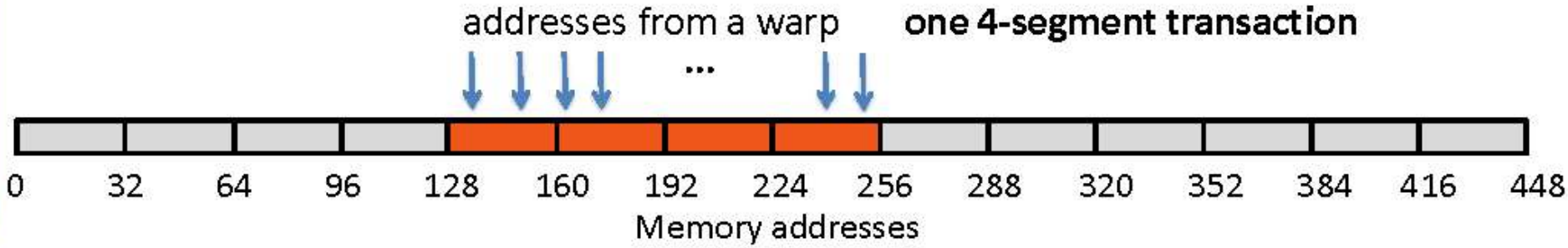
- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer argumen
    - Pointer of interest: const
    - All other pointer argument
      - Conveys to compiler t
  - Using __ldg() intrinsic
    - Requires no pointer deco

```
__global__ void kernel( int *output,
                        int *input )
{
    ...
    output[idx] = __ldg( &input[idx] );
}
```

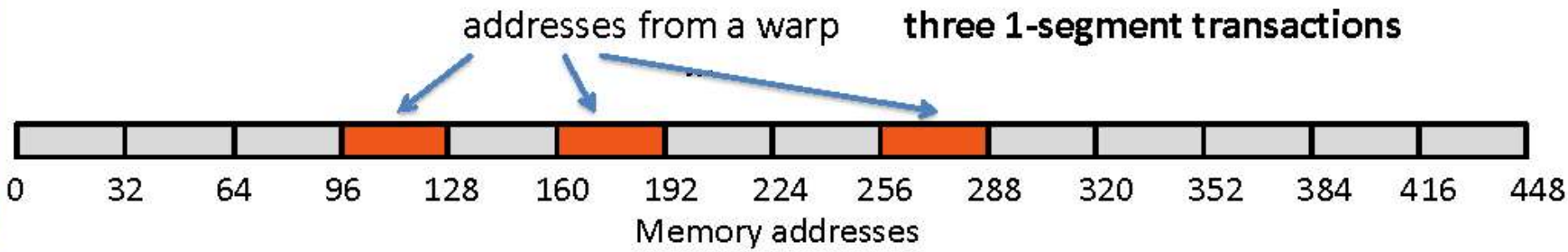
Blocking for L1, Read-only, L2 Caches

- Short answer: DON'T
- GPU caches are not intended for the same use as CPU caches
  - Smaller size (especially per thread), so not aimed at temporal reuse
  - Intended to smooth out some access patterns, help with spilled registers, etc.
- Usually not worth trying to cache-block like you would on CPU
  - 100s to 1,000s of run-time scheduled threads competing for the cache
  - If it is possible to block for L1 then it's possible block for SMEM
    - Same size
    - Same or higher bandwidth
    - Guaranteed locality: hw will not evict behind your back

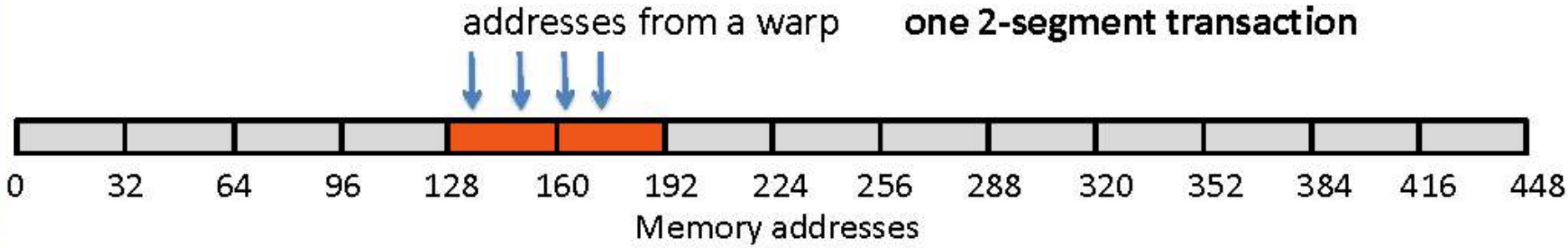
Some Store Pattern Examples



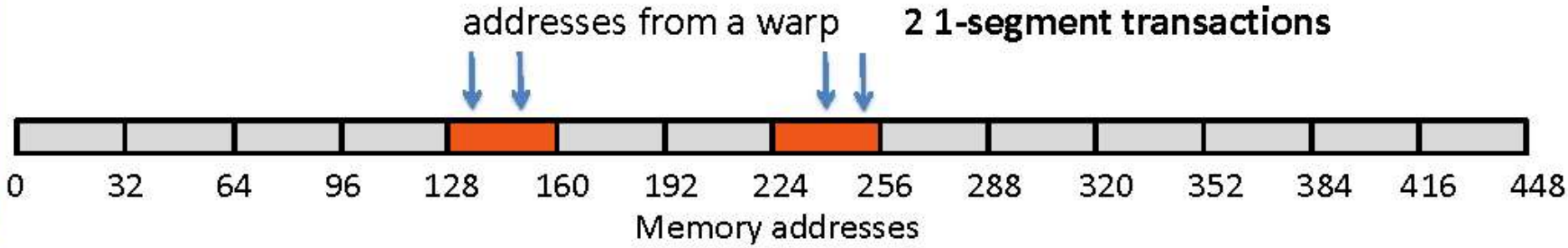
Some Store Pattern Examples



Some Store Pattern Examples



Some Store Pattern Examples

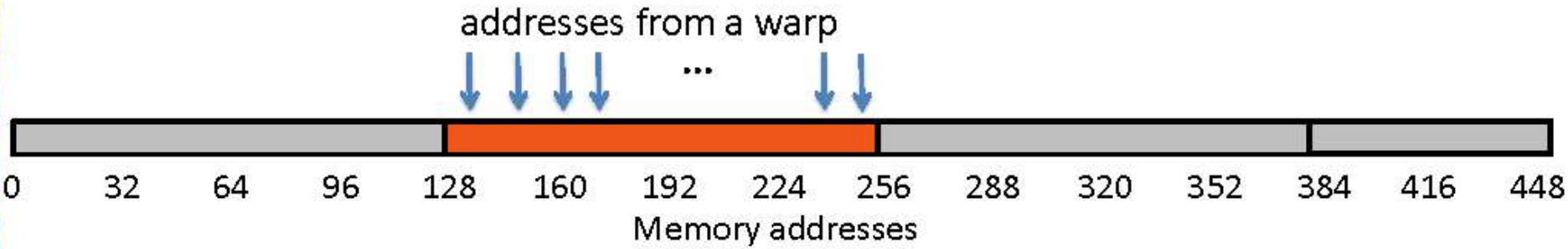


GMEM Reads

- **Attempt to hit in L1 depends on programmer choice and compute capability**
- **HW ability to hit in L1:**
  - CC 1.x: no L1
  - CC 2.x: can hit in L1
  - CC 3.0, 3.5: cannot hit in L1
    - L1 is used to cache LMEM (register spills, etc.), buffer reads
- **Read instruction types**
  - Caching:
    - Compiler option: **-Xptxas -dlcm=ca**
    - On L1 miss go to L2, on L2 miss go to DRAM
    - Transaction: 128 B line
  - Non-caching:
    - Compiler option: **-Xptxas -dlcm=cg**
    - Go directly to L2 (invalidate line in L1), on L2 miss go to DRAM
    - Transaction: 1, 2, 4 segments, segment = 32 B (same as for writes)

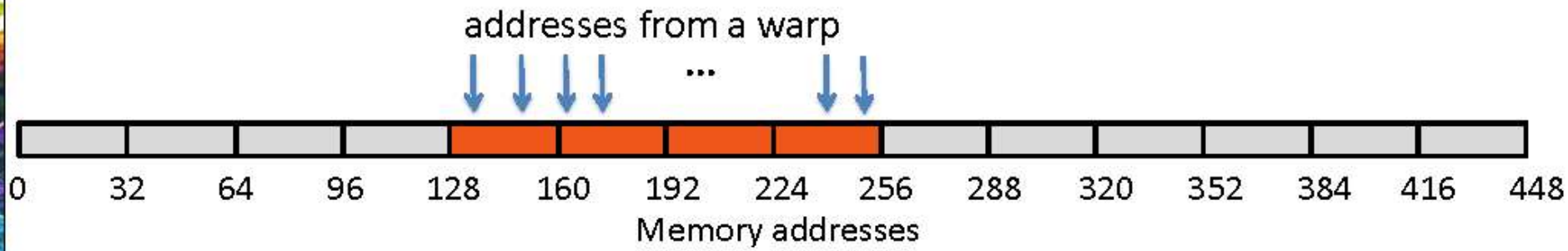
Caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



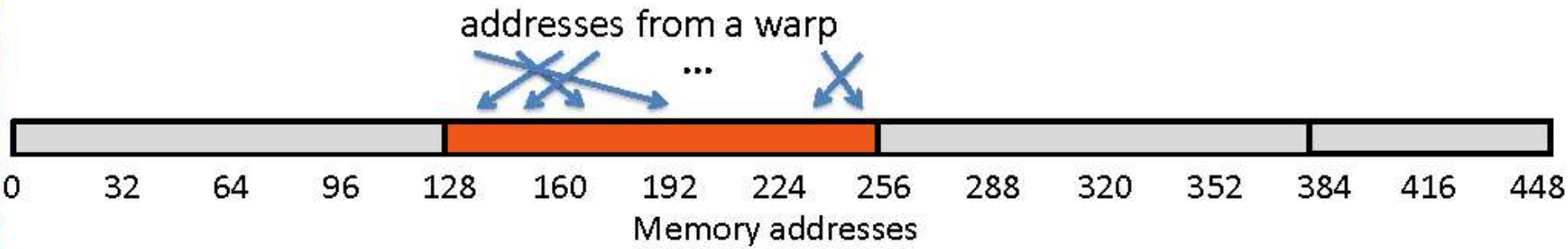
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



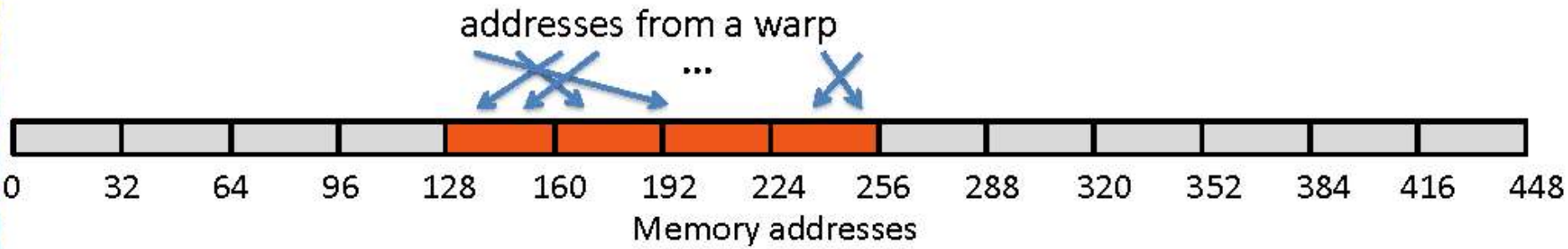
Caching Load

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



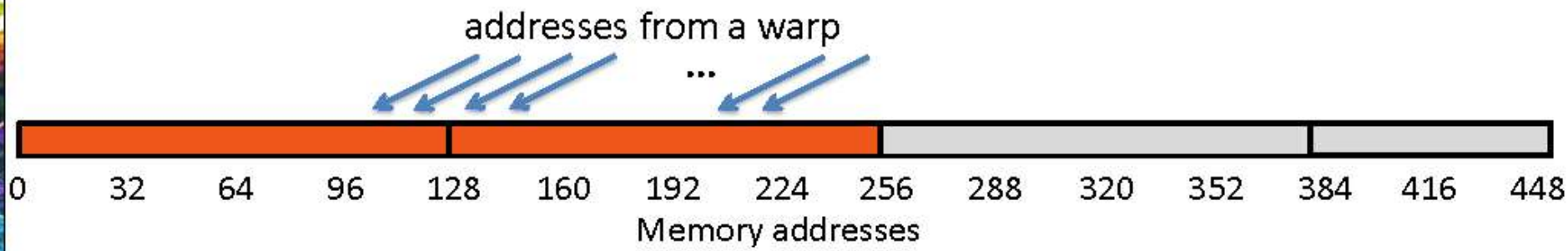
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



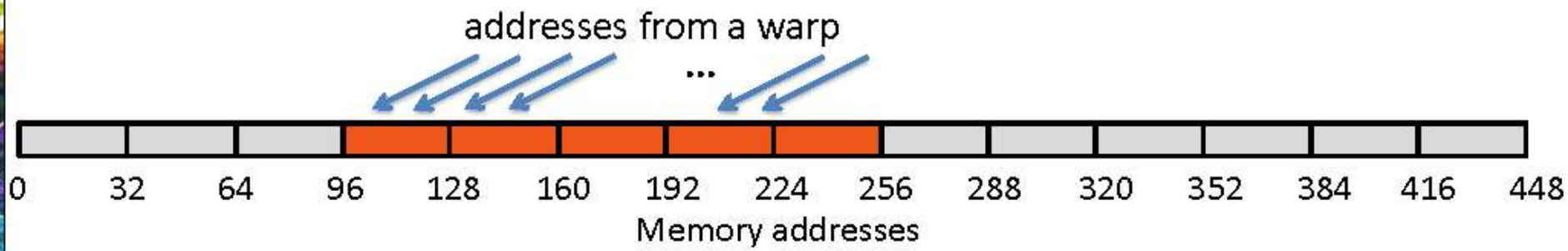
Caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within 2 cache-lines**
  - 1 replay (2 transactions)
  - Bus utilization: 50%
    - Warp needs 128 bytes
    - 256 bytes move across the bus on misses



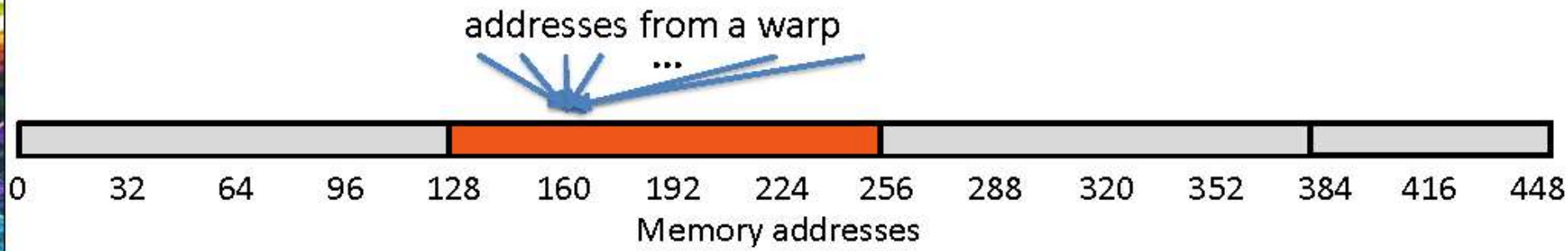
Non-caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within at most 5 segments**
  - 1 replay (2 transactions)
  - Bus utilization: at least 80%
    - Warp needs 128 bytes
    - At most 160 bytes move across the bus
    - Some misaligned patterns will fall within 4 segments, so 100% utilization



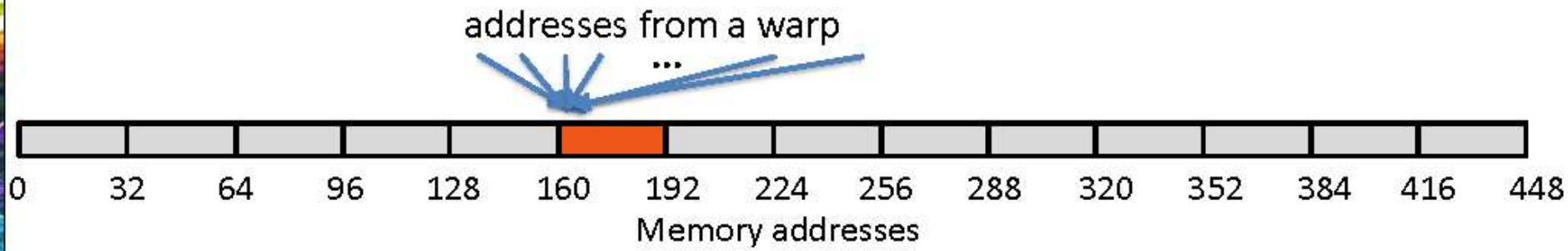
Caching Load

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single cache-line**
  - No replays
  - Bus utilization: 3.125%
    - Warp needs 4 bytes
    - 128 bytes move across the bus on a miss



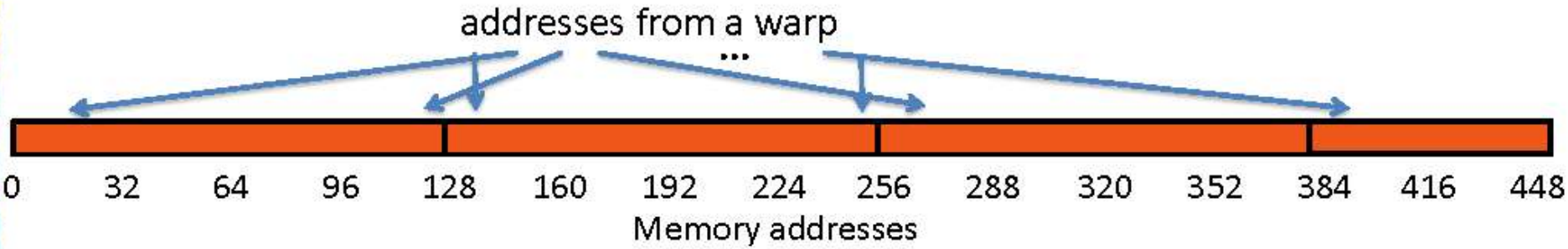
Non-caching Load

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single segment**
  - No replays
  - Bus utilization: 12.5%
    - Warp needs 4 bytes
    - 32 bytes move across the bus on a miss



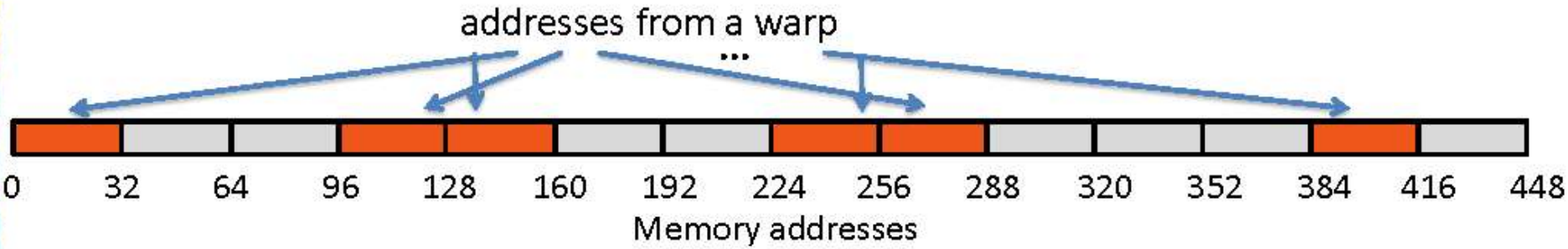
Caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within *N* cache-lines**
  - (N-1) replays (N transactions)
  - Bus utilization: 32*4B / (*N*128B)
    - Warp needs 128 bytes
    - *N*128 bytes move across the bus on a miss



Non-caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within *N* segments**
  - (N-1) replays (N transactions)
    - Could be lower some segments can be arranged into a single transaction
  - Bus utilization: 128 / (*N**32) (4x higher than caching loads)
    - Warp needs 128 bytes
    - *N**32 bytes move across the bus on a miss



Caching vs Non-caching Loads

- **Compute capabilities that can hit in L1 (CC 2.x)**
  - Caching loads are better if you count on hits
  - Non-caching loads are better if:
    - Warp address pattern is scattered
    - When kernel uses lots of LMEM (register spilling)
- **Compute capabilities that cannot hit in L1 (CC 1.x, 3.0, 3.5)**
  - Does not matter, all loads behave like non-caching
- **In general, don't rely on GPU caches like you would on CPUs:**
  - 100s of threads sharing the same L1
  - 1000s of threads sharing the same L2

L1 Sizing

- **Fermi and Kepler GPUs split 64 KB RAM between L1 and SMEM**
  - Fermi GPUs (**CC 2.x**): 16:48, 48:16
  - Kepler GPUs (**CC 3.x**):16:48, 48:16, 32:32
- **Programmer can choose the split:**
  - Default: 16 KB L1, 48 KB SMEM
  - Run-time API functions:
    - cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
  - Kernels that require different L1:SMEM sizing cannot run concurrently
- **Making the choice:**
  - Large L1 can help when using lots of LMEM (spilling registers)
  - Large SMEM can help if occupancy is limited by shared memory

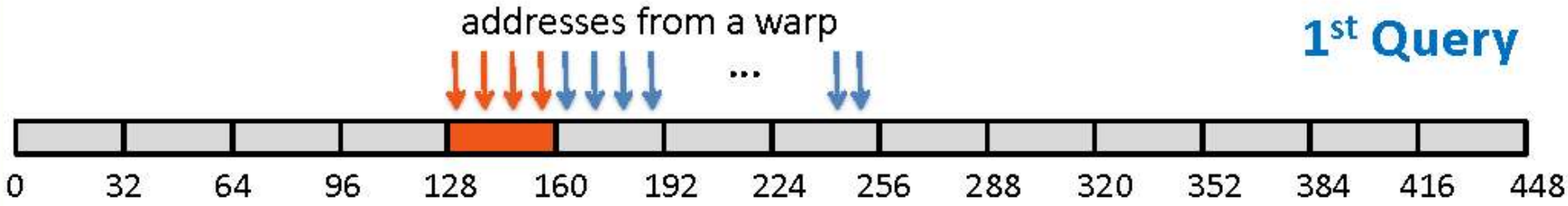
Read-Only Cache

- **An alternative to L1 when accessing DRAM**
  - Also known as *texture* cache: all texture accesses use this cache
  - CC 3.5 and higher also enable global memory accesses
    - Should not be used if a kernel reads and writes to the same addresses
- **Comparing to L1:**
  - Generally better for scattered reads than L1
    - Caching is at 32 B granularity (L1, when caching operates at 128 B granularity)
    - Does not require replay for multiple transactions (L1 does)
  - Higher latency than L1 reads, also tends to increase register use
- **Aggregate 48 KB per SM: 4 12-KB caches**
  - One 12-KB cache per scheduler
    - Warps assigned to a scheduler refer to only that cache
  - Caches are not coherent – data replication is possible

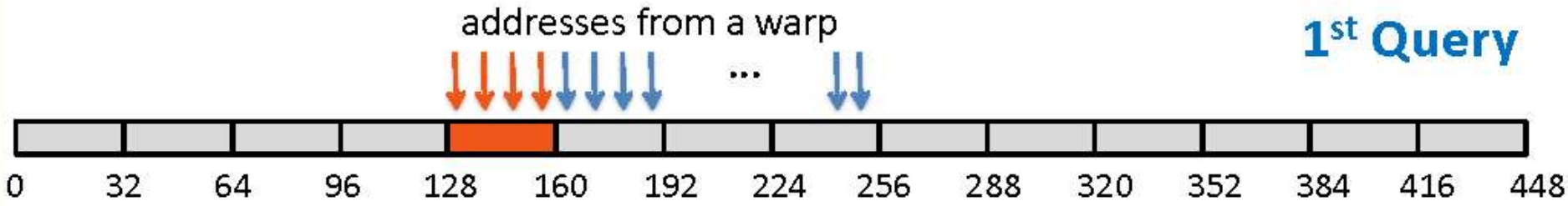
Read-Only Cache Operation

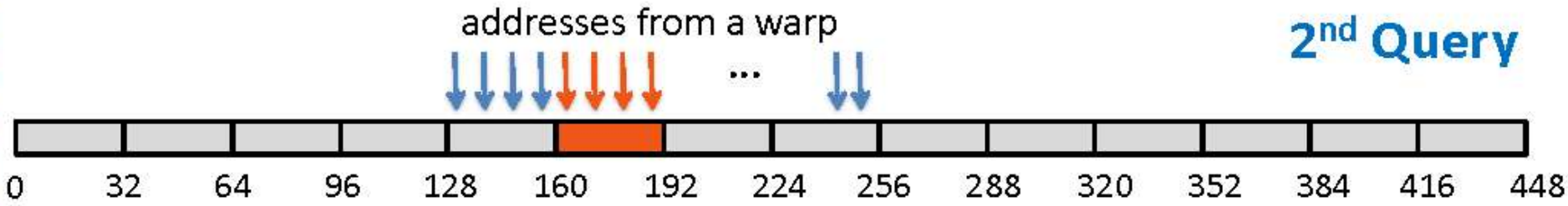
- **Always attempts to hit**
- **Transaction size: 32 B queries**
- **Warp addresses are converted to queries 4 threads at a time**
  - Thus a minimum of 8 queries per warp
  - If data within a 32-B segment is needed by multiple threads in a warp, segment misses at most once
- **Additional functionality for texture objects**
  - Interpolation, clamping, type conversion

Read-Only Cache Operation

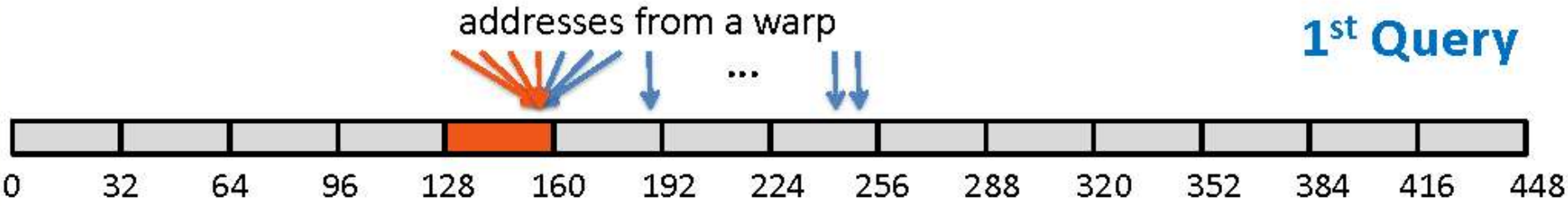


Read-Only Cache Operation

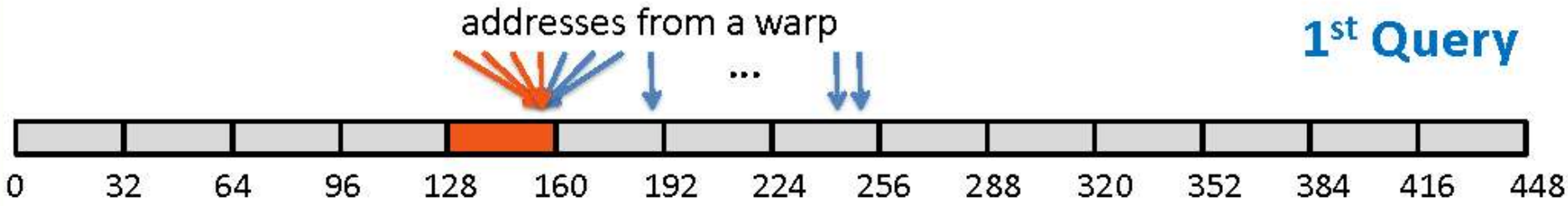


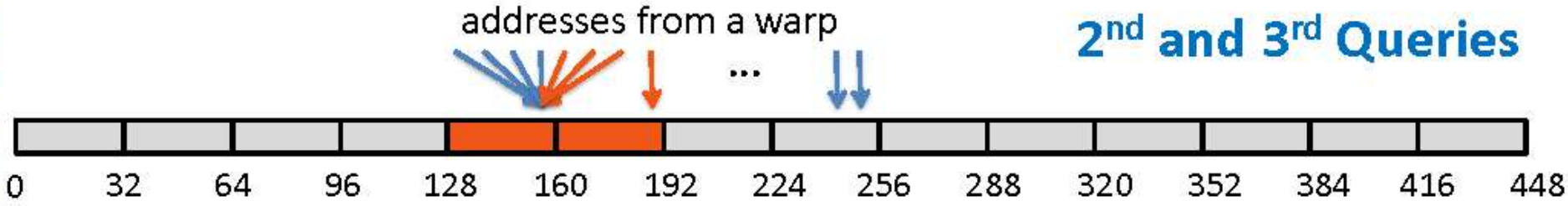


Read-Only Cache Operation

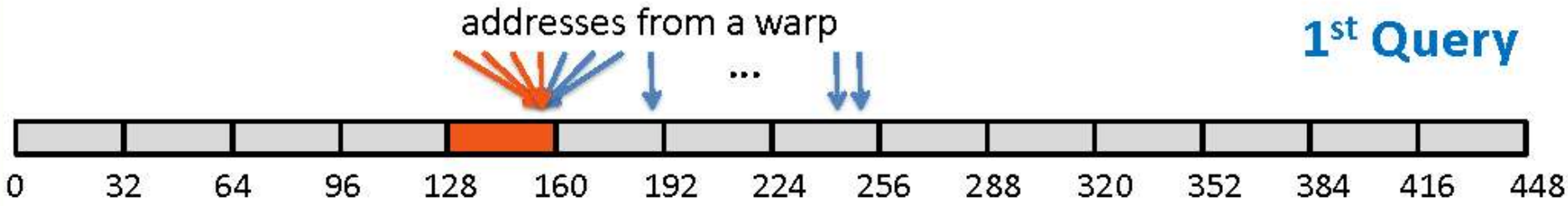


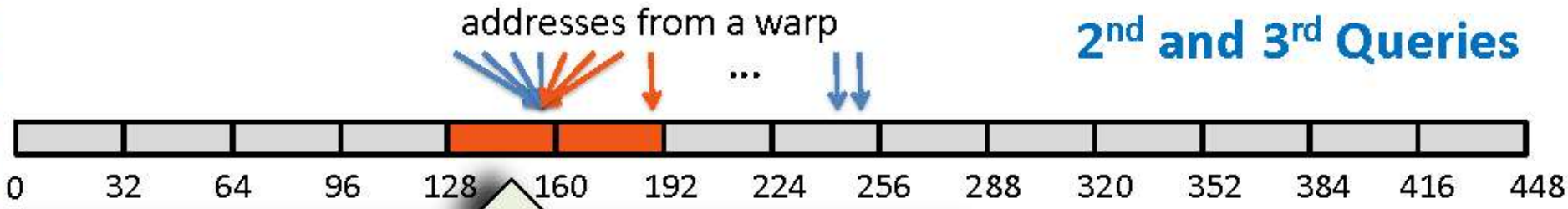
Read-Only Cache Operation





Read-Only Cache Operation





Note this segment was already requested in the 1st query: cache hit, no redundant requests to L2

Thank you.