

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):

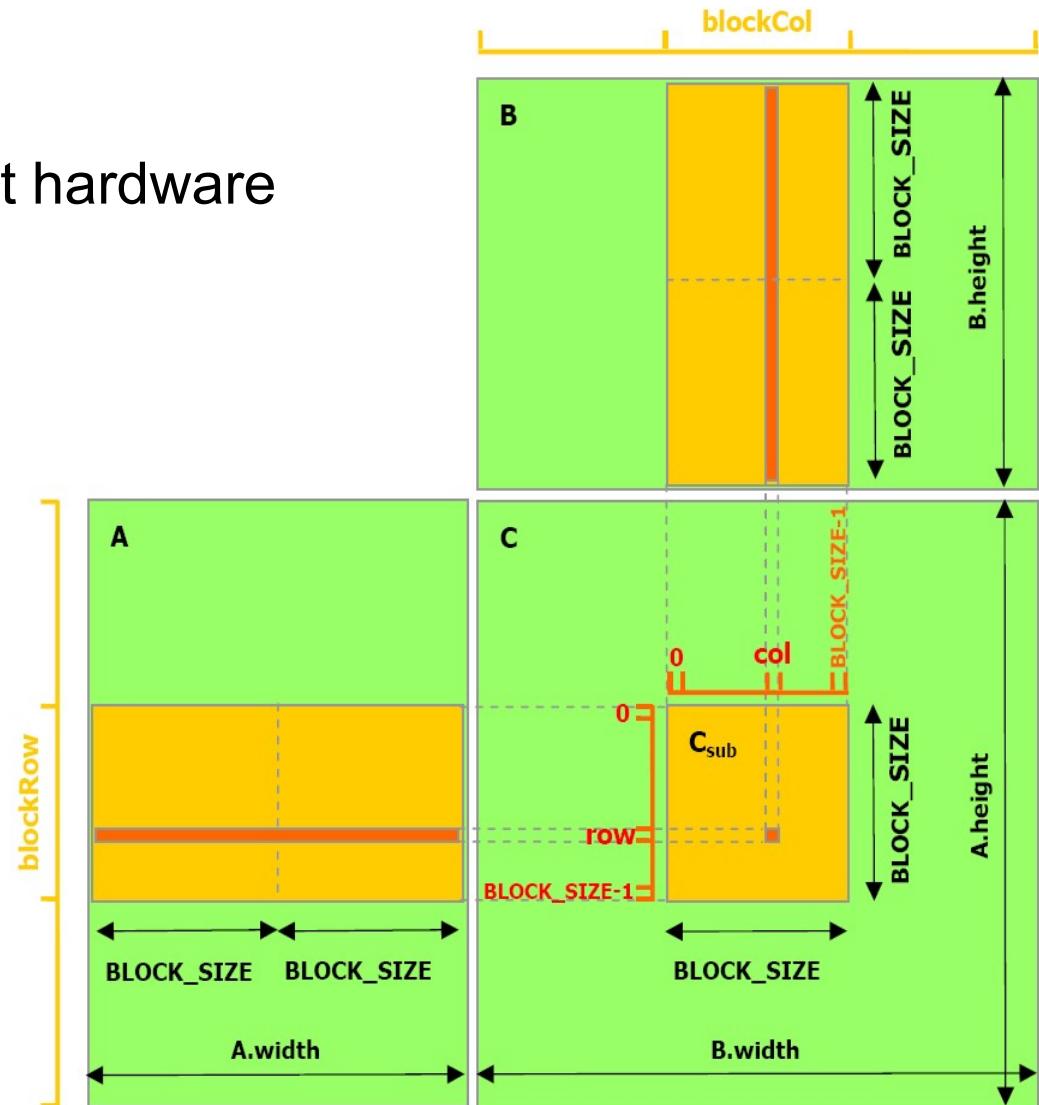
- Programming Massively Parallel Processors book (4th edition),
Chapter 20 (An introduction to CUDA streams)
- Programming Massively Parallel Processors book (4th edition),
Chapter 21 (CUDA dynamic parallelism)

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++ ) {

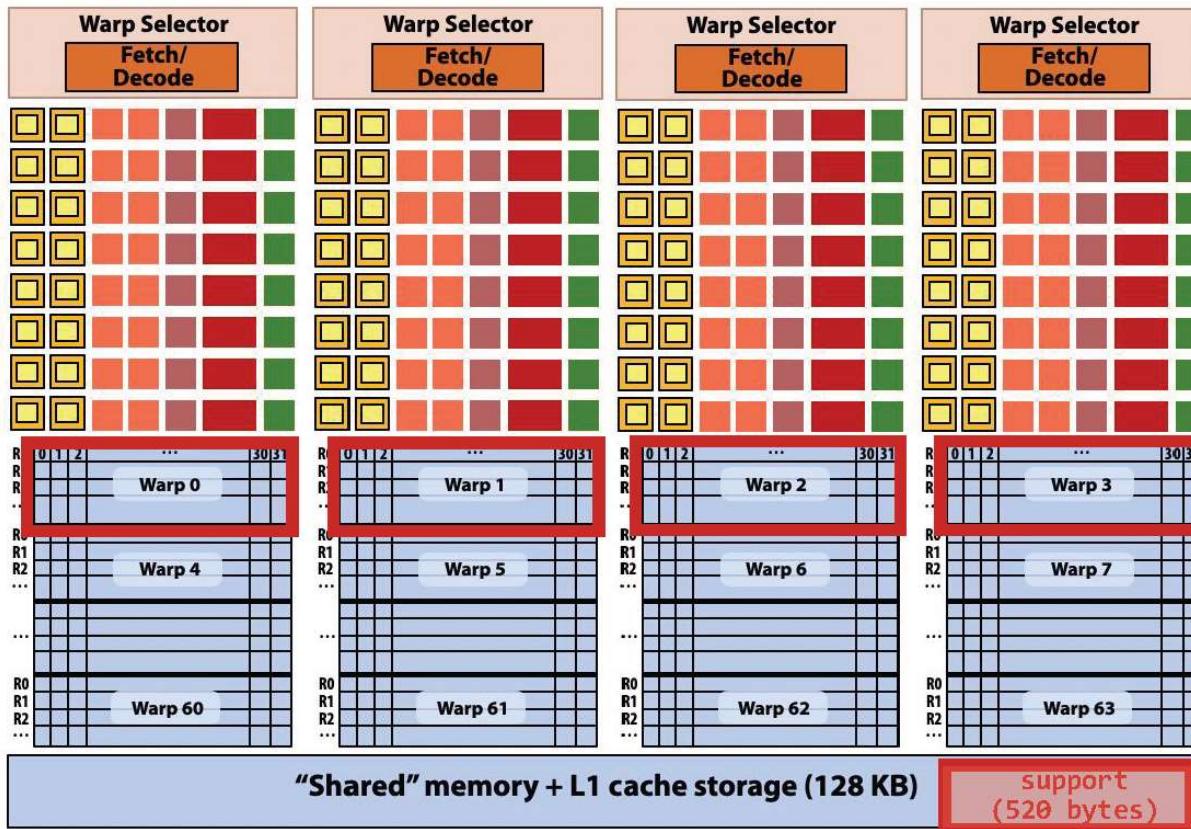
        blockA[ ty ][ tx ] = matA[ row * w + m * BLOCK_SIZE + tx ];
        blockB[ ty ][ tx ] = matB[ col      + ( m * BLOCK_SIZE + ty ) * w ];
        __syncthreads();

        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



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

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)

(sub-core == SM partition)

courtesy Kayvon Fatahalian

Stanford CS149, Fall 2021

```
#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;
}
```



Limits in CUDA Programming Guide

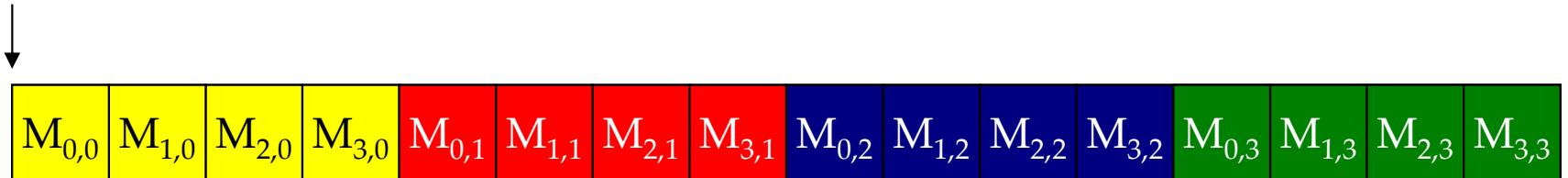
	Compute Capability													
Technical Specifications	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6	8.7	8.9	9.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	32		16	128	32	16	128	16	128					
Maximum dimensionality of grid of thread blocks	3													
Maximum x -dimension of a grid of thread blocks [thread blocks]	$2^{31}-1$													
Maximum y- or z-dimension of a grid of thread blocks	65535													
Maximum dimensionality of thread block	3													
Maximum x- or y-dimensionality of a block	1024													
Maximum z-dimension of a block	64													
Maximum number of threads per block	1024													
Warp size	32													
Maximum number of resident blocks per SM	32								16	32	16	24	32	
Maximum number of resident warps per SM	64								32	64	48		64	
Maximum number of resident threads per SM	2048								1024	2048	1536		2048	

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

Memory Layout of a Matrix in C

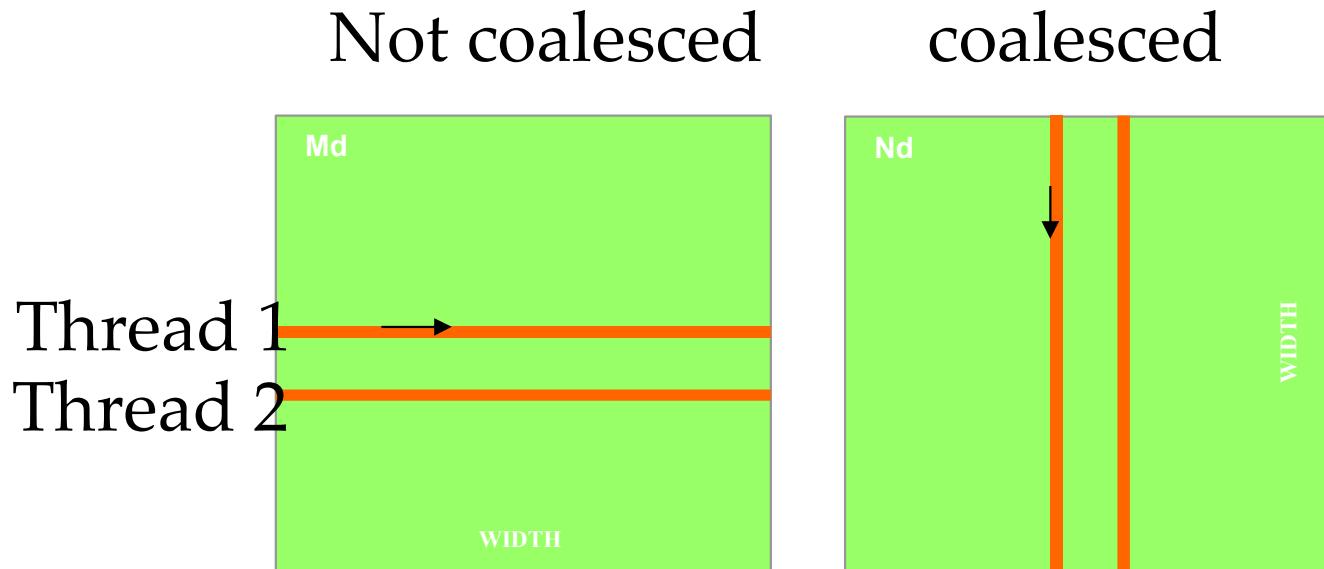
M _{0,0}	M _{1,0}	M _{2,0}	M _{3,0}
M _{0,1}	M _{1,1}	M _{2,1}	M _{3,1}
M _{0,2}	M _{1,2}	M _{2,2}	M _{3,2}
M _{0,3}	M _{1,3}	M _{2,3}	M _{3,3}

M



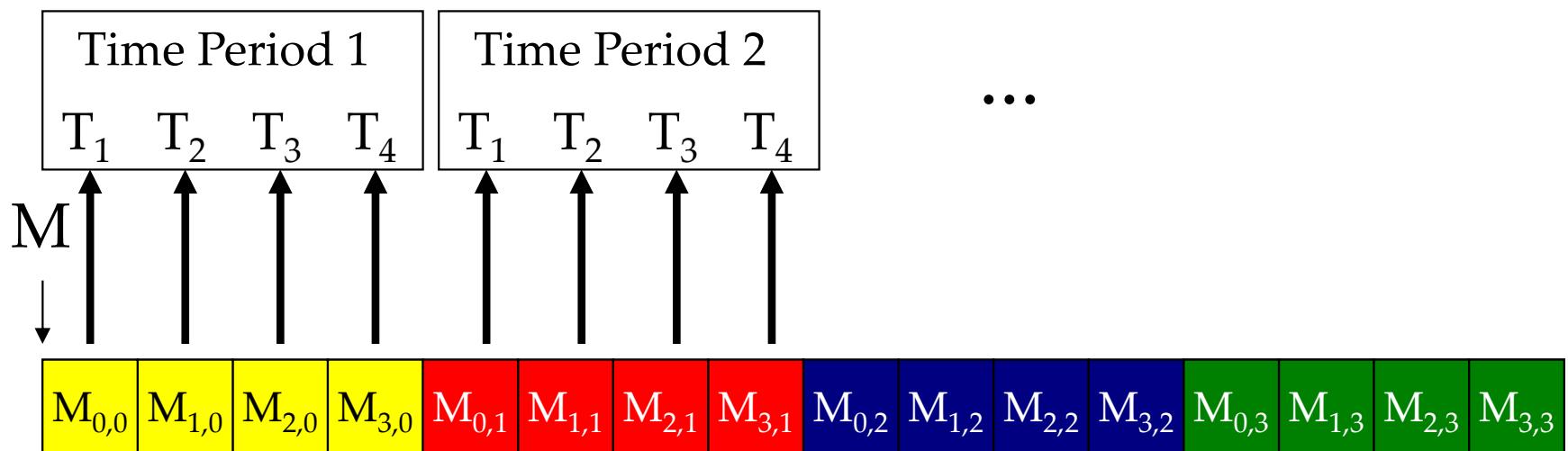
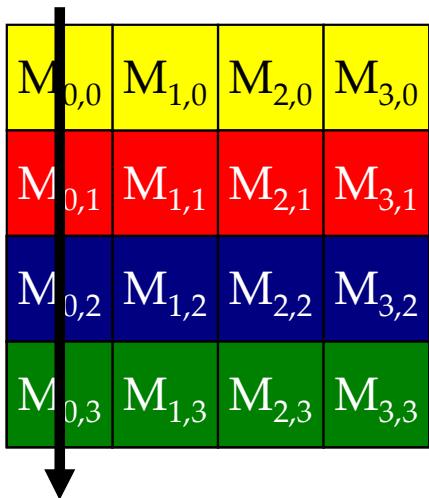
Memory Coalescing

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

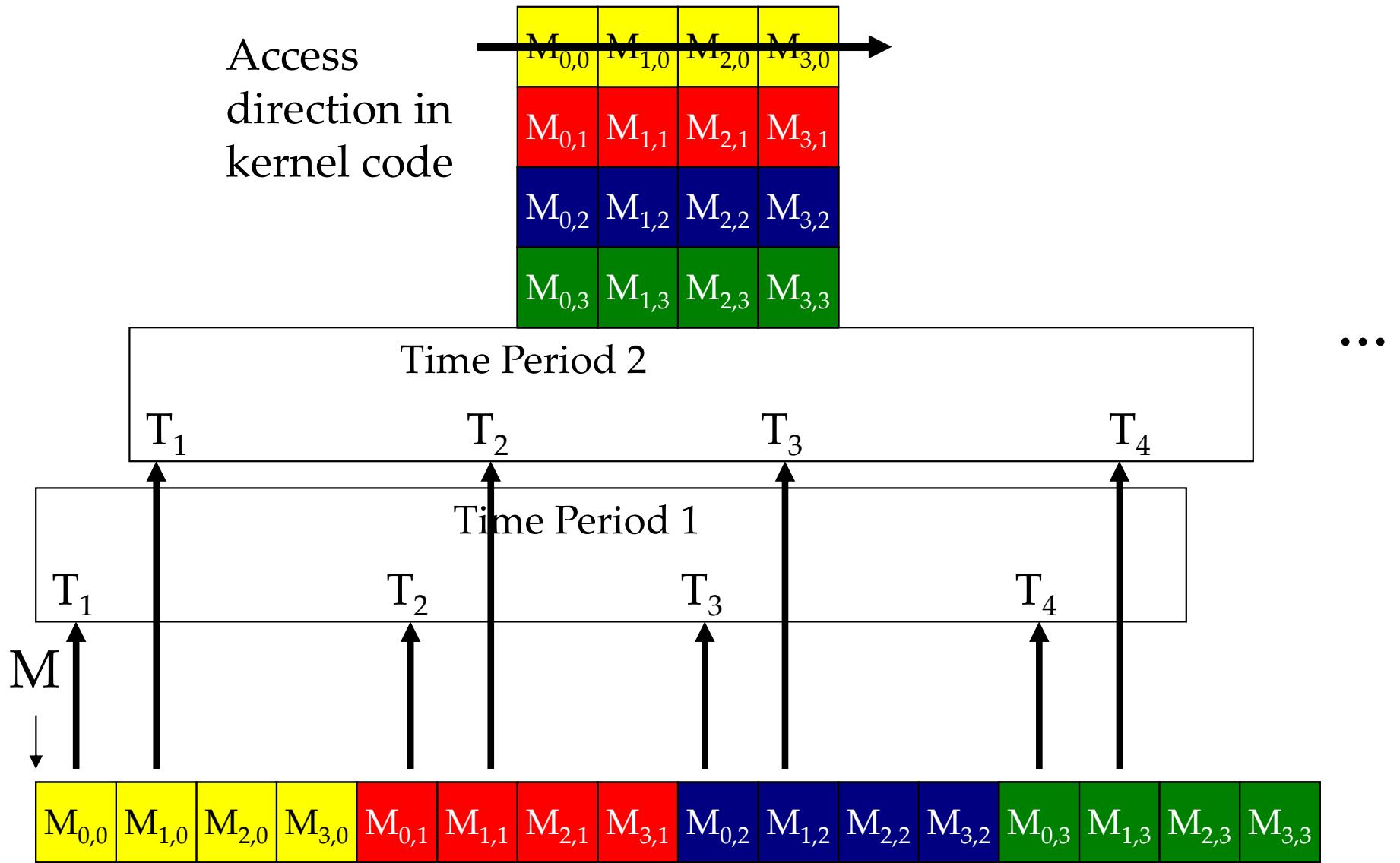


Memory Layout of a Matrix in C

Access
direction in
kernel code



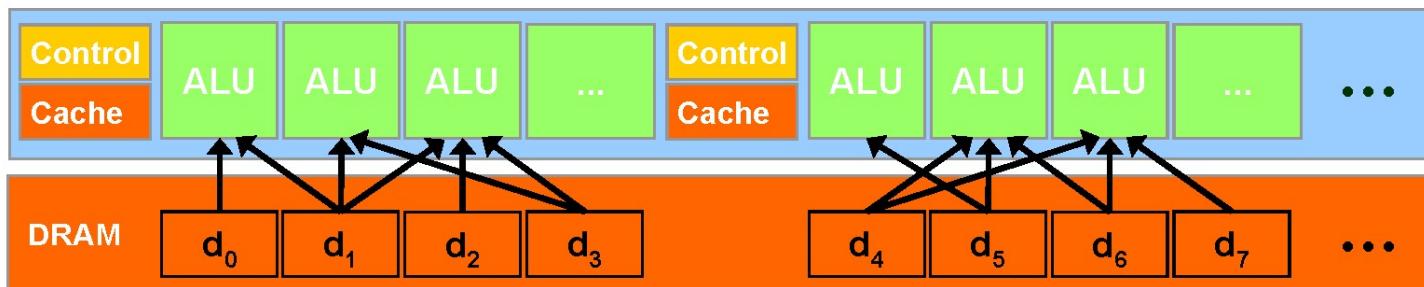
Memory Layout of a Matrix in C



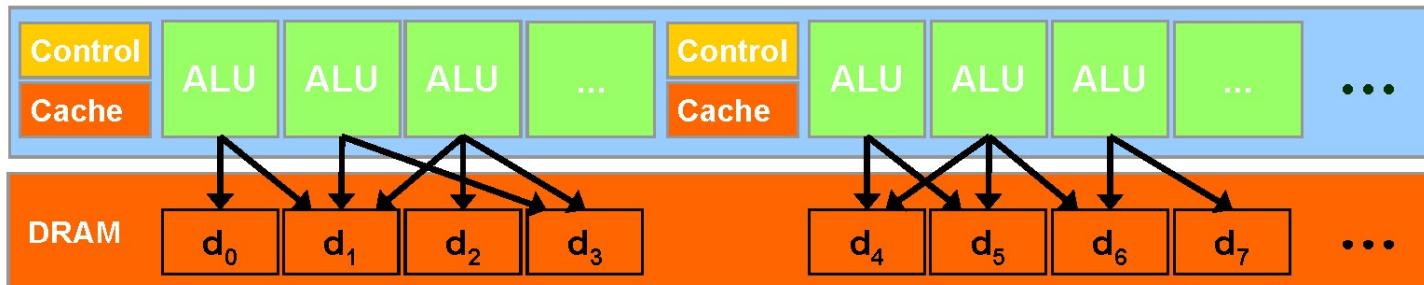
CUDA Memory

CUDA Highlights: Scatter

- CUDA provides generic DRAM memory addressing
 - Gather:



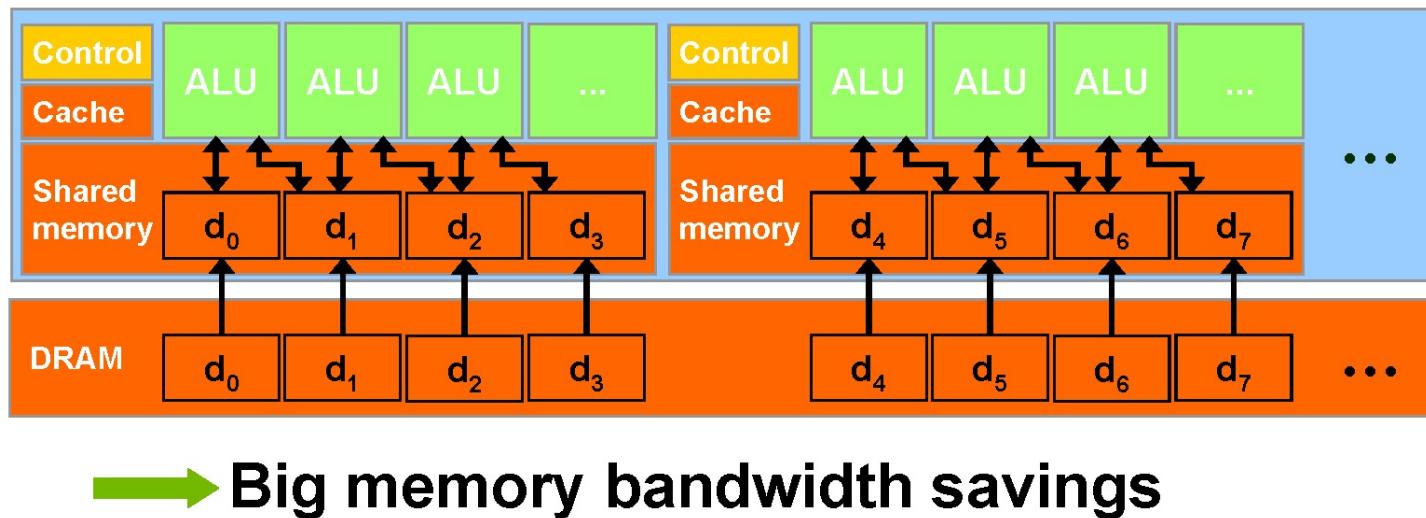
- And scatter: no longer limited to write one pixel



→ More programming flexibility

CUDA Highlights: On-Chip Shared Memory

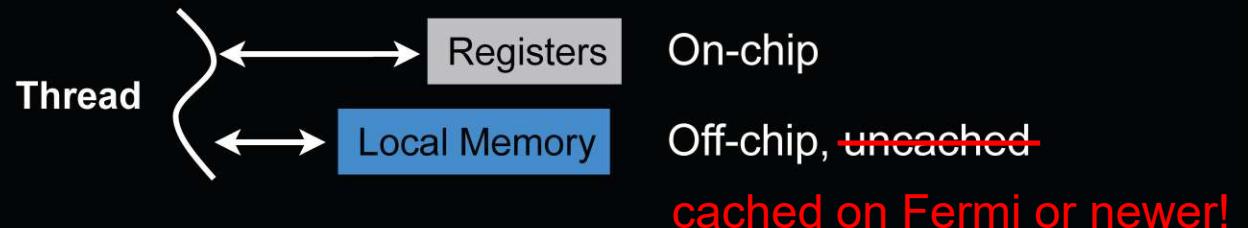
- CUDA enables access to a parallel **on-chip shared memory** for efficient inter-thread data sharing



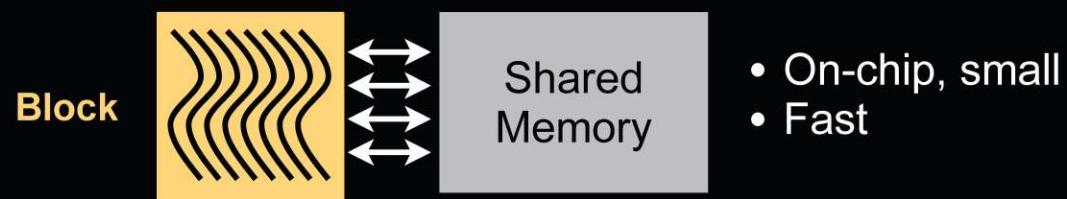
CUDA Memory: Overview

Kernel Memory Access

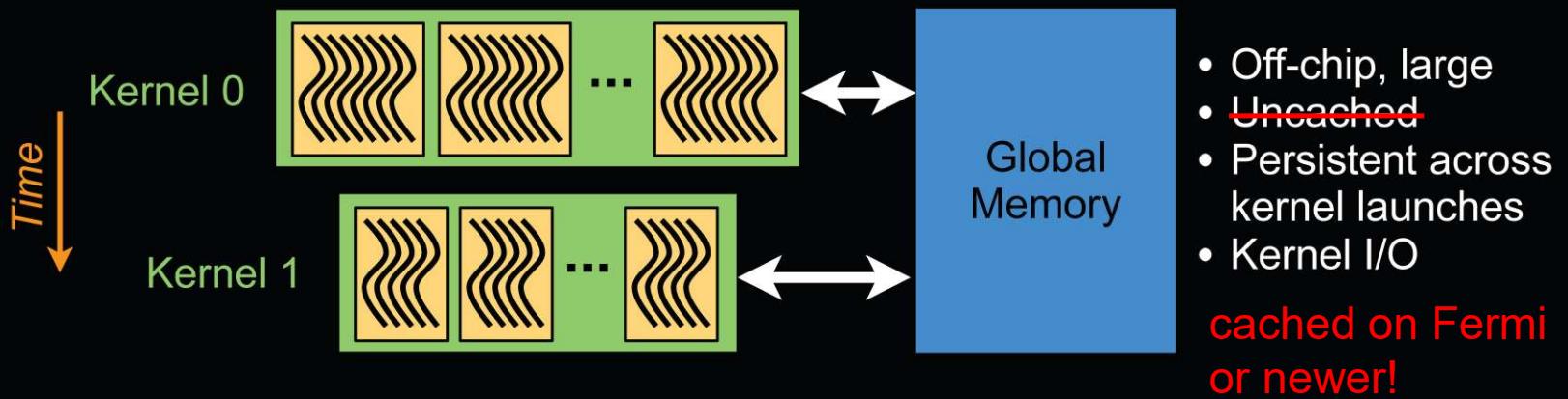
● Per-thread



● Per-block



● Per-device





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

NVIDIA Architectures (since first CUDA GPU)



Tesla [CC 1.x]: 2007-2009

- G80, G9x: 2007 (Geforce 8800, ...)
GT200: 2008/2009 (GTX 280, ...)

Fermi [CC 2.x]: 2010 (2011, 2012, 2013, ...)

- GF100, ... (GTX 480, ...)
- GF104, ... (GTX 460, ...)
- GF110, ... (GTX 580, ...)

Kepler [CC 3.x]: 2012 (2013, 2014, 2016, ...)

- GK104, ... (GTX 680, ...)
- GK110, ... (GTX 780, GTX Titan, ...)

Maxwell [CC 5.x]: 2015

- GM107, ... (GTX 750Ti, ...)
- GM204, ... (GTX 980, Titan X, ...)

Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, ...)

- GP100 (Tesla P100, ...)
- GP10x: x=2,4,6,7,8, ...
(GTX 1060, 1070, 1080, Titan X *Pascal*, Titan Xp, ...)

Volta [CC 7.0, 7.2]: 2017/2018

- GV100, ...
(Tesla V100, Titan V, Quadro GV100, ...)

Turing [CC 7.5]: 2018/2019

- TU102, TU104, TU106, TU116, TU117, ...
(Titan RTX, RTX 2070, 2080 (Ti), GTX 1650, 1660, ...)

Ampere [CC 8.0, 8.6, 8.7]: 2020

- GA100, GA102, GA104, GA106, ...
(A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

Hopper [CC 9.0], Ada Lovelace [CC 8.9]: 2022/23

- GH100, AD102, AD103, AD104, ...
(H100, L40, RTX 4080 (12/16 GB), 4090, RTX 6000, ...)

Blackwell [CC 10.0]: *coming in 2024/25*

- GB200/GB202, GB20x, ...?
(RTX 5080/5090, GB200 NVL72, HGX B100/200, ...?)

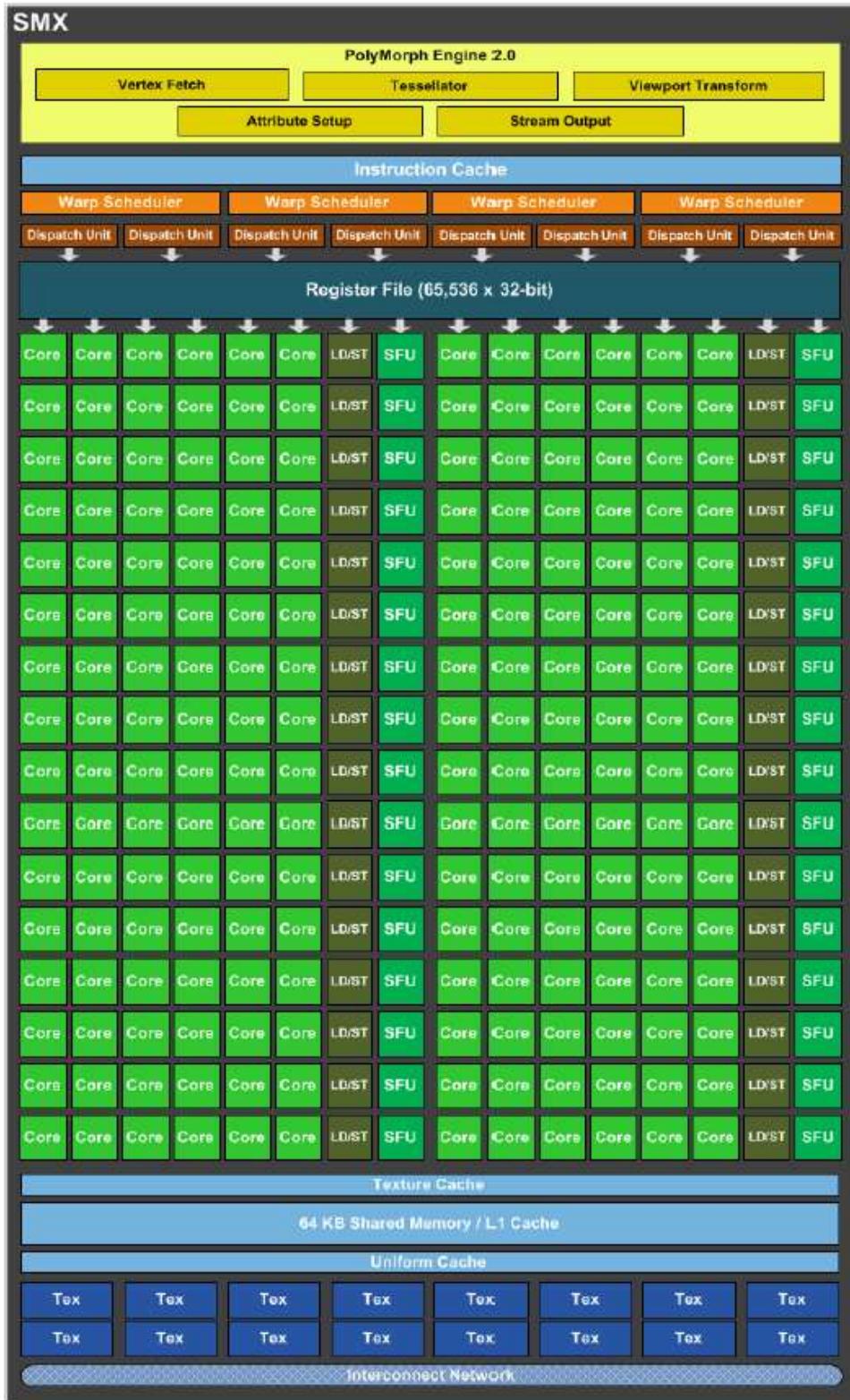
GK104 SMX

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)

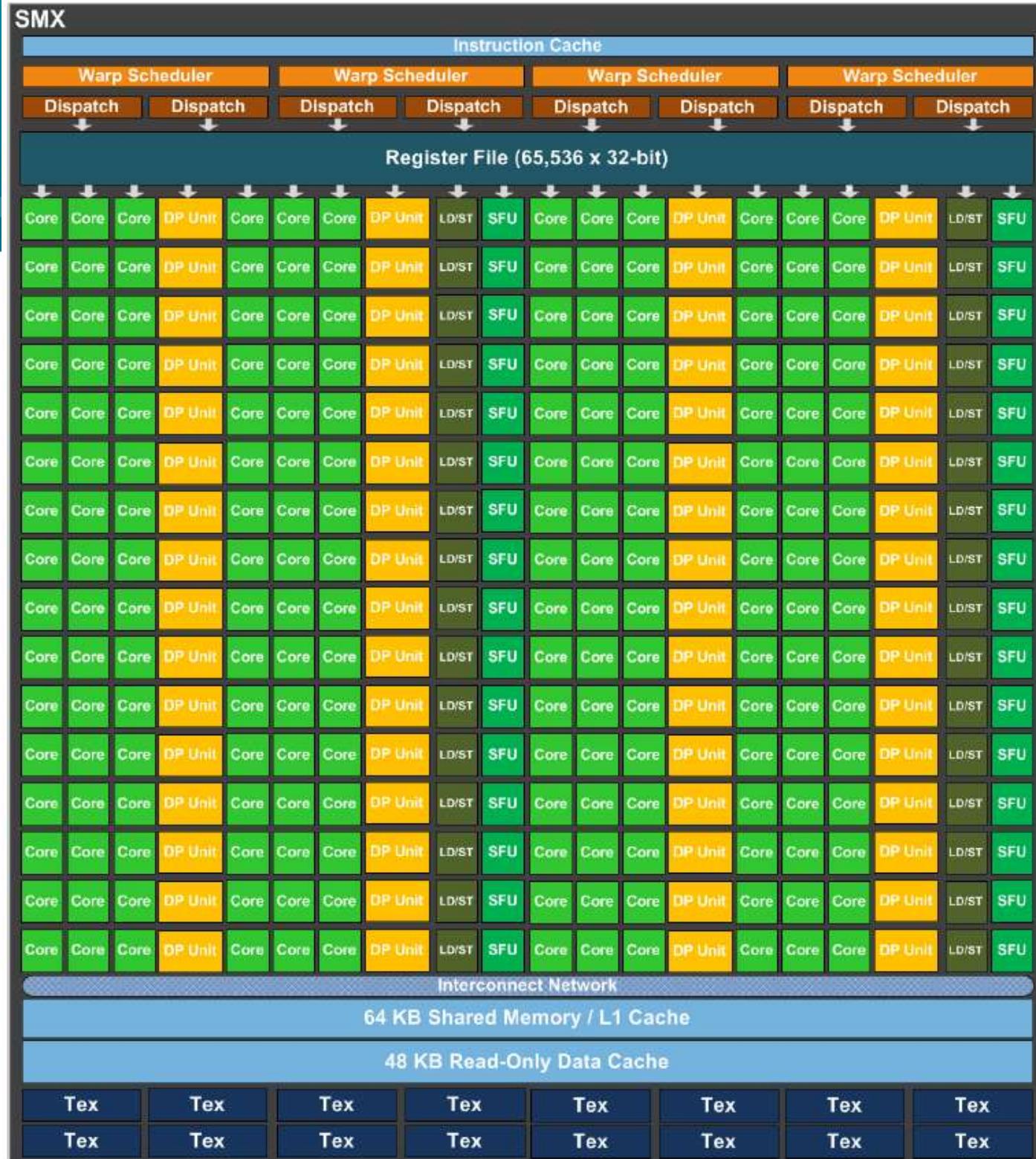
Can dual-issue ALU instructions!
("superscalar")



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





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.



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)

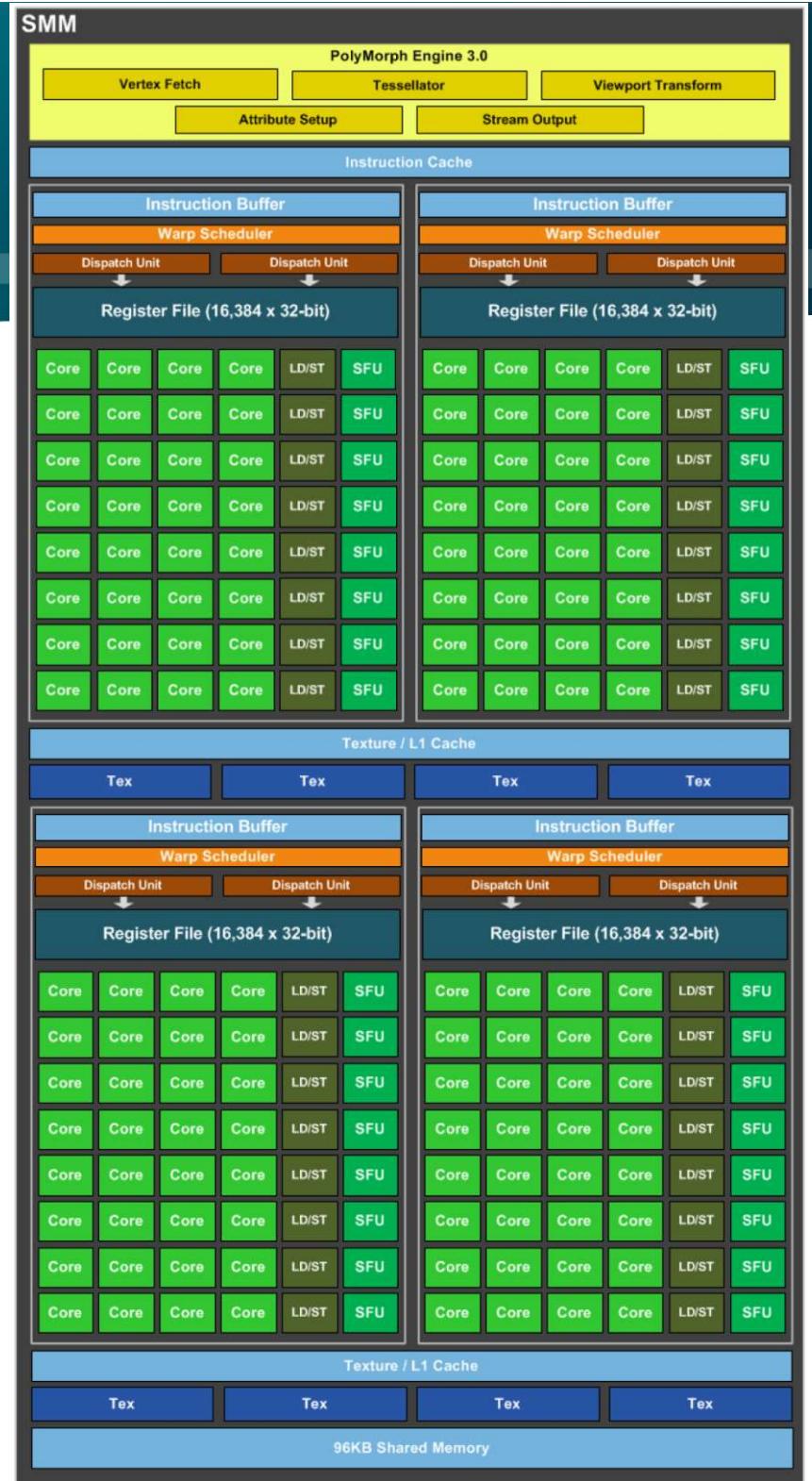
- 128 CUDA cores
- 4 DP units; 32 LD/ST units; 32 SFUs
- 8 texture units

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





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.



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.



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.!)

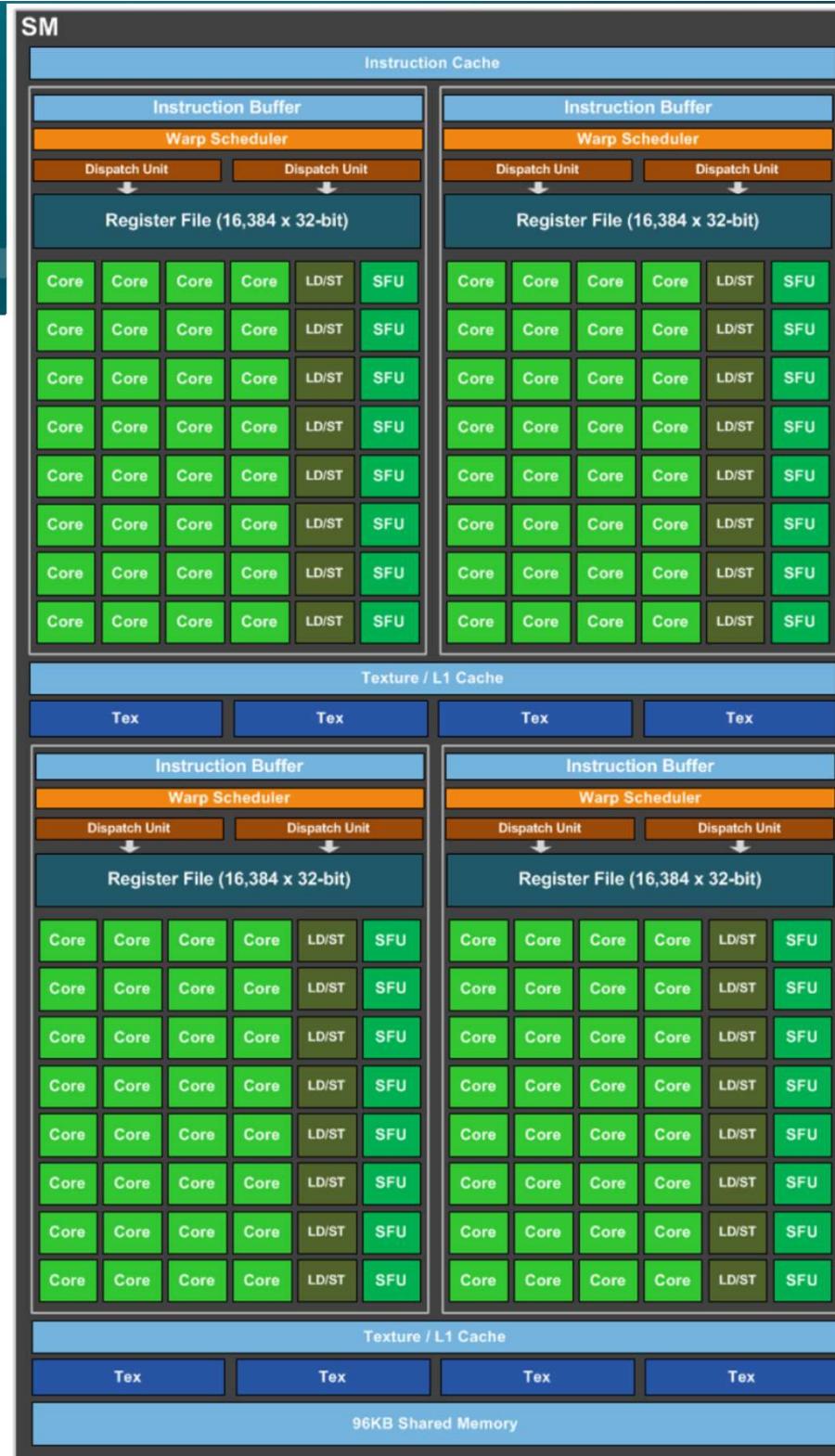
NVIDIA Pascal GP104 SM

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.!)





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.



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



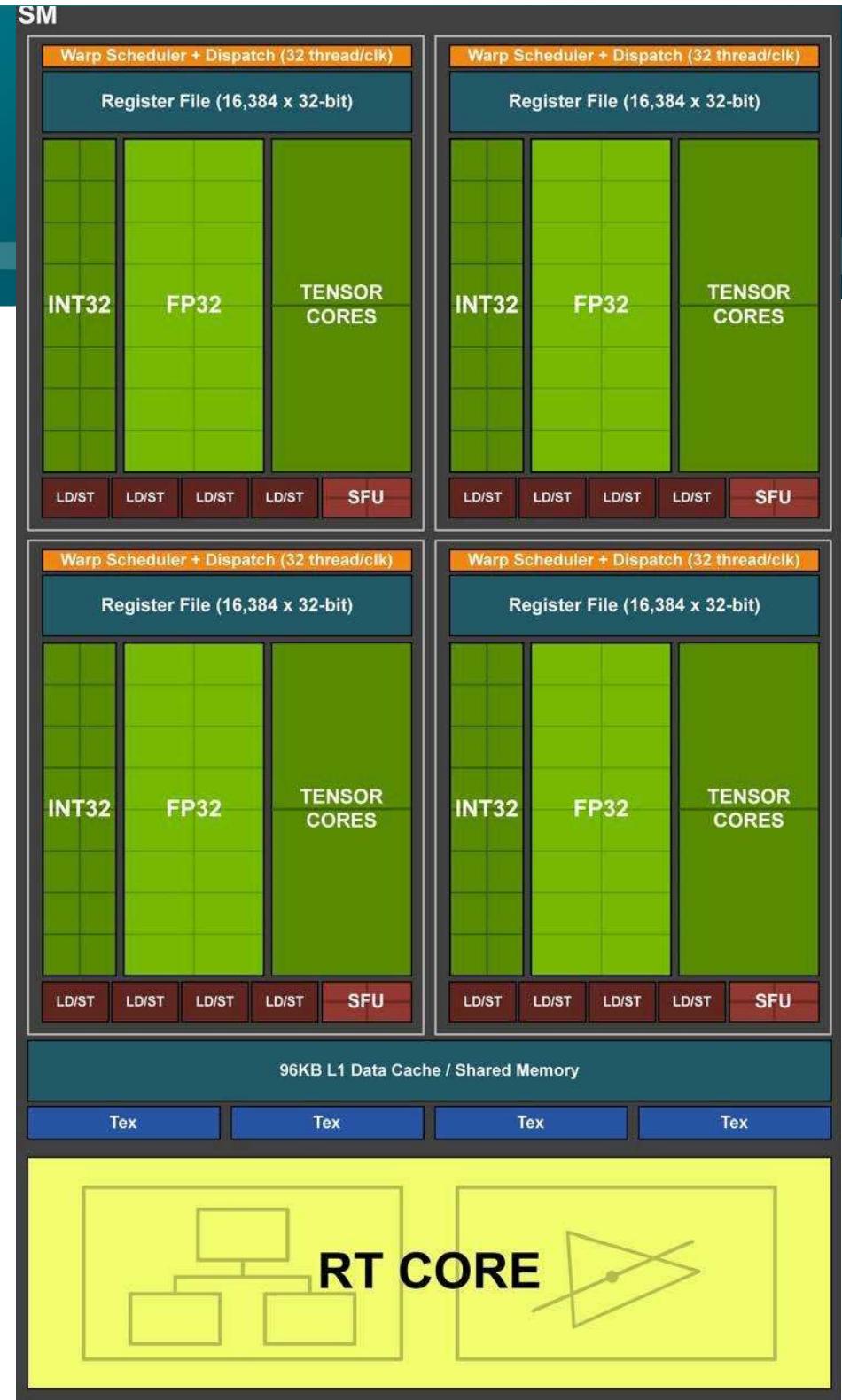
NVIDIA Turing SM

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





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](#).



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





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](#).



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.

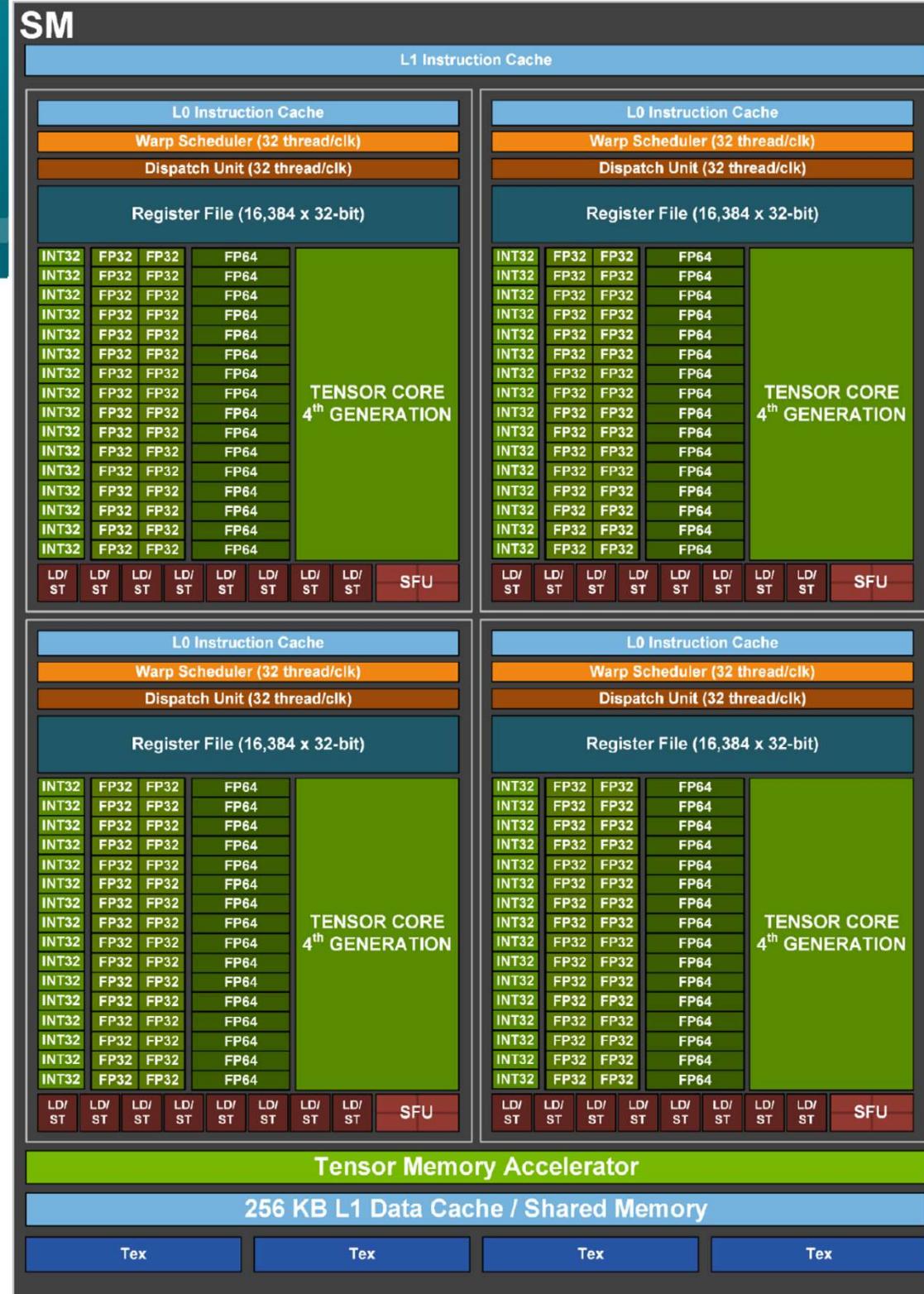
NVIDIA GH100 SM

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





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](#)).



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

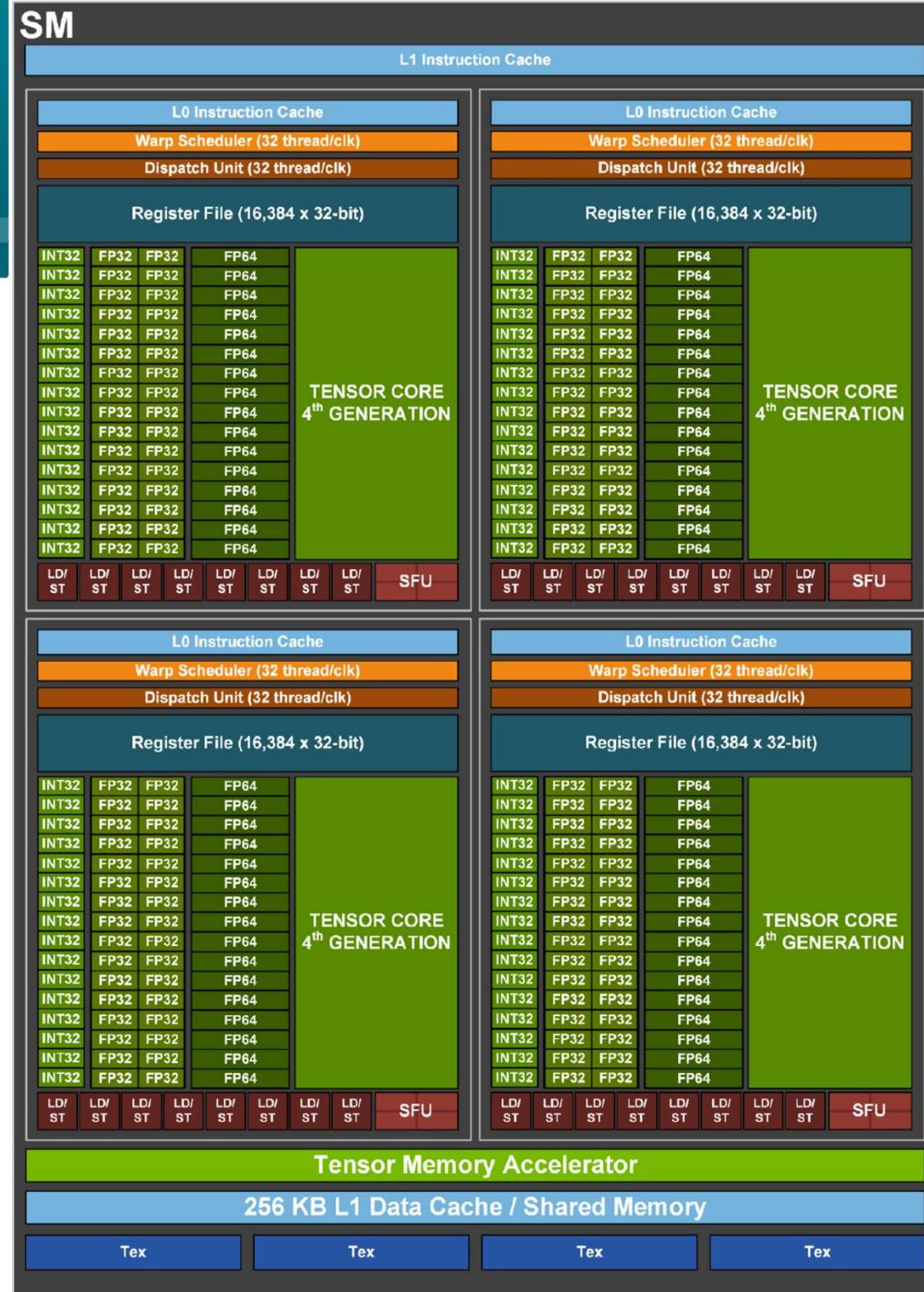
NVIDIA GH100 SM

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





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()~~
- 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

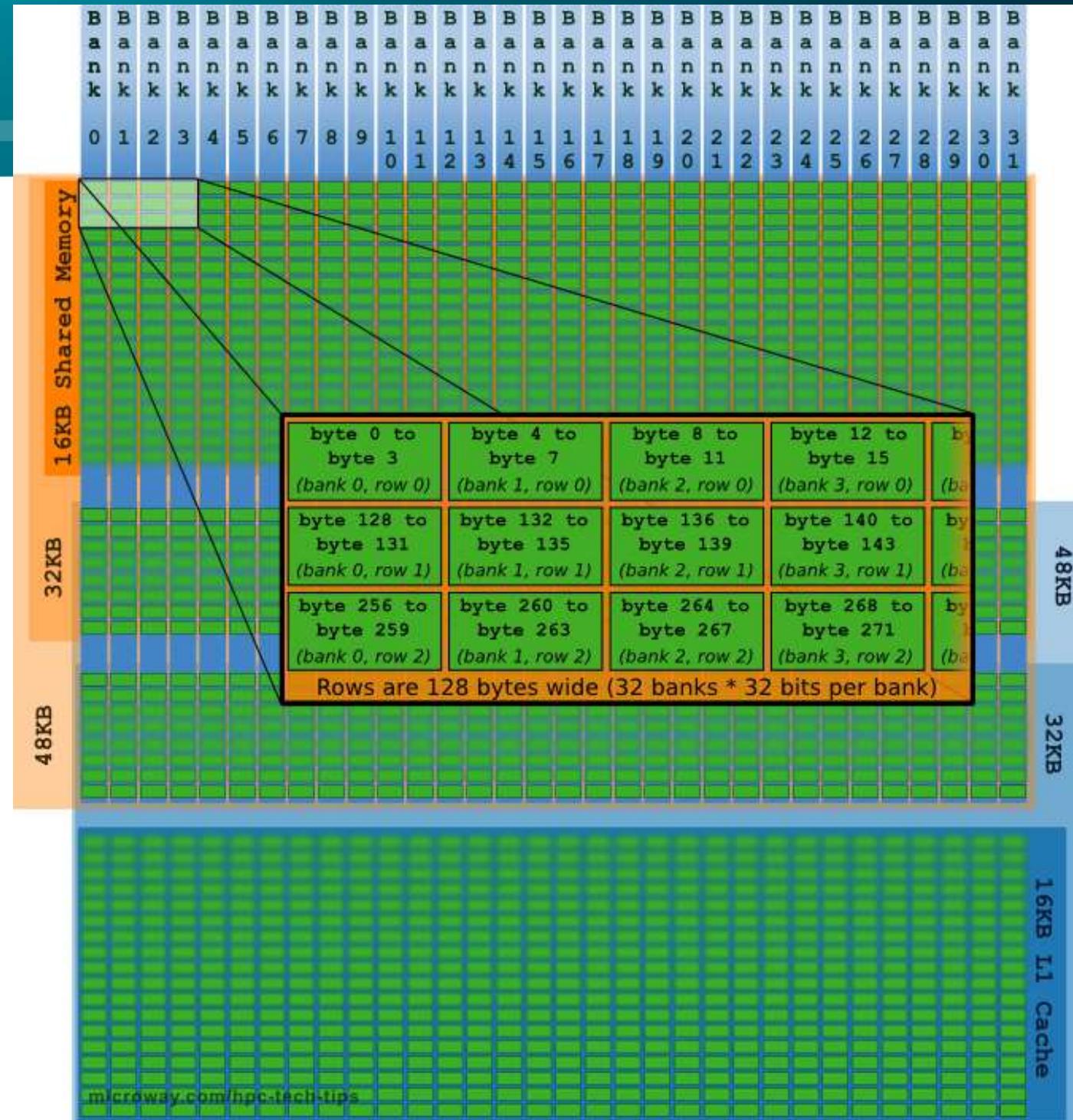
Memory Banks

Fermi/Kepler/Maxwell
and newer:

32 banks

default:
4B / bank

Kepler or newer:
configurable
to 8B / bank

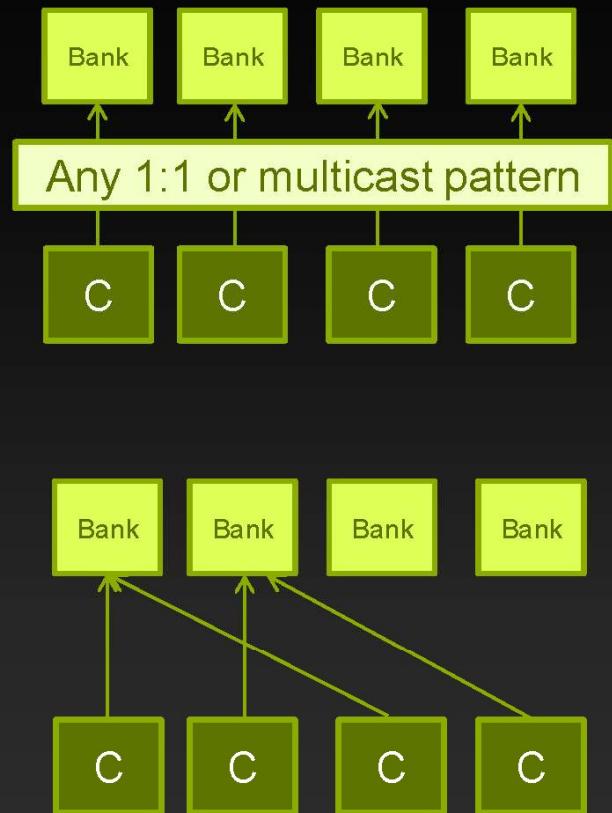


Shared Memory

- **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



Thank you.