King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 22: CUDA Memory, Pt. 3

Markus Hadwiger, KAUST

Reading Assignment #12 (until Nov 23)

Read (required):

- Optimizing Parallel Reduction in CUDA, Mark Harris,

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

- Programming Massively Parallel Processors book, 3rd edition
  Chapter 8 (Parallel Patterns: Prefix Sum)
- GPU Gems 3 book, Chapter 39: Parallel Prefix Sum (Scan) with CUDA

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

Read (optional):

- Faster Parallel Reductions on Kepler, Justin Luitjens

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

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

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

Memory Configurations and Types for Different Compute Capabilities

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

A multiprocessor 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 multiprocessor and an L2 cache shared by all multiprocessors. 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)

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 multiprocessor, 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 multiprocessor 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)

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.

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

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.

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.

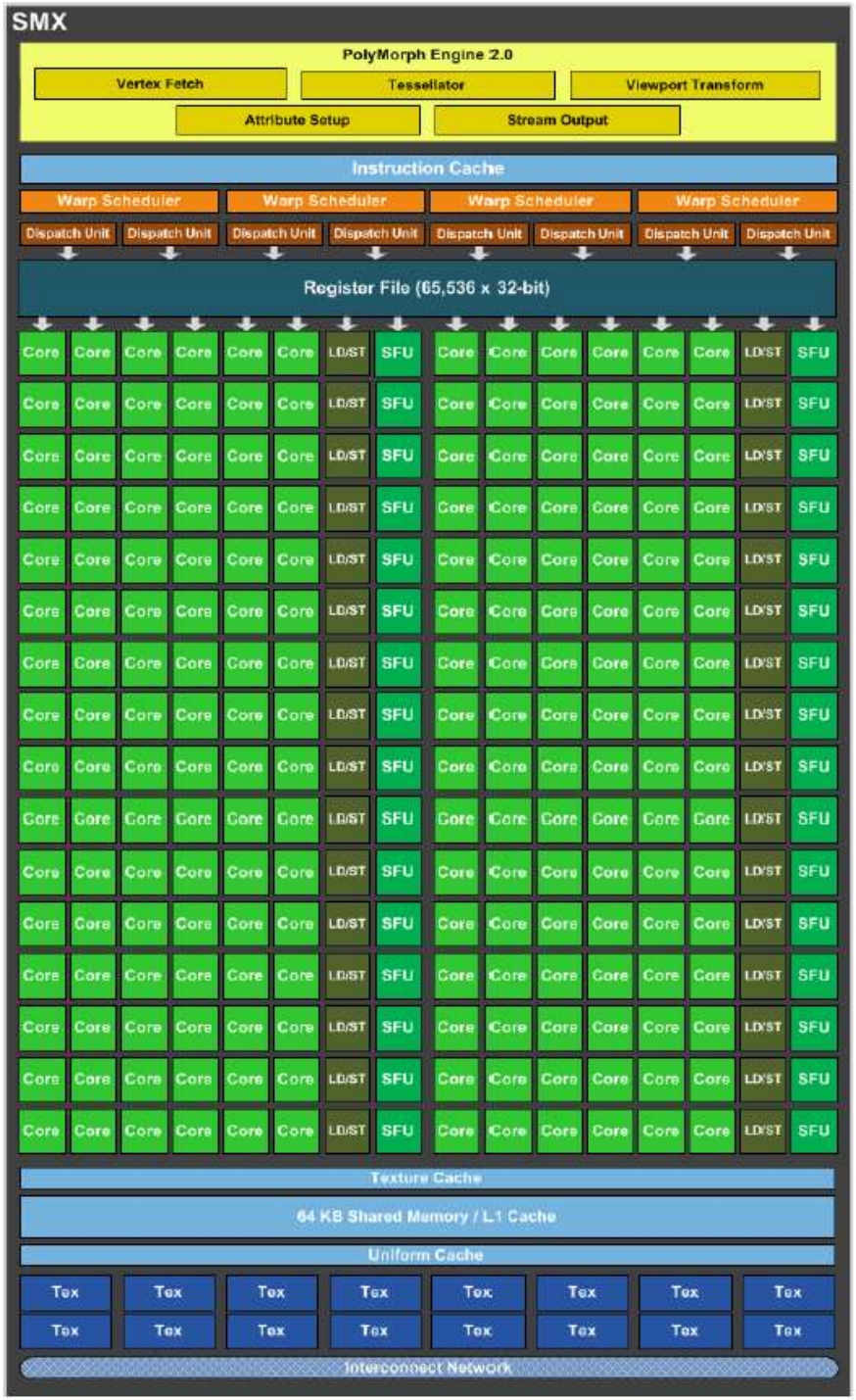
Note that threads can access any words in any order, including the same words.

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.

GK104 SMX

- 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*)



GK110 SMX

- 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. 5.x (Maxwell, Part 1)

A multiprocessor 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 multiprocessors 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)

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

Maxwell (GM) Architecture

Multiprocessor: SMM

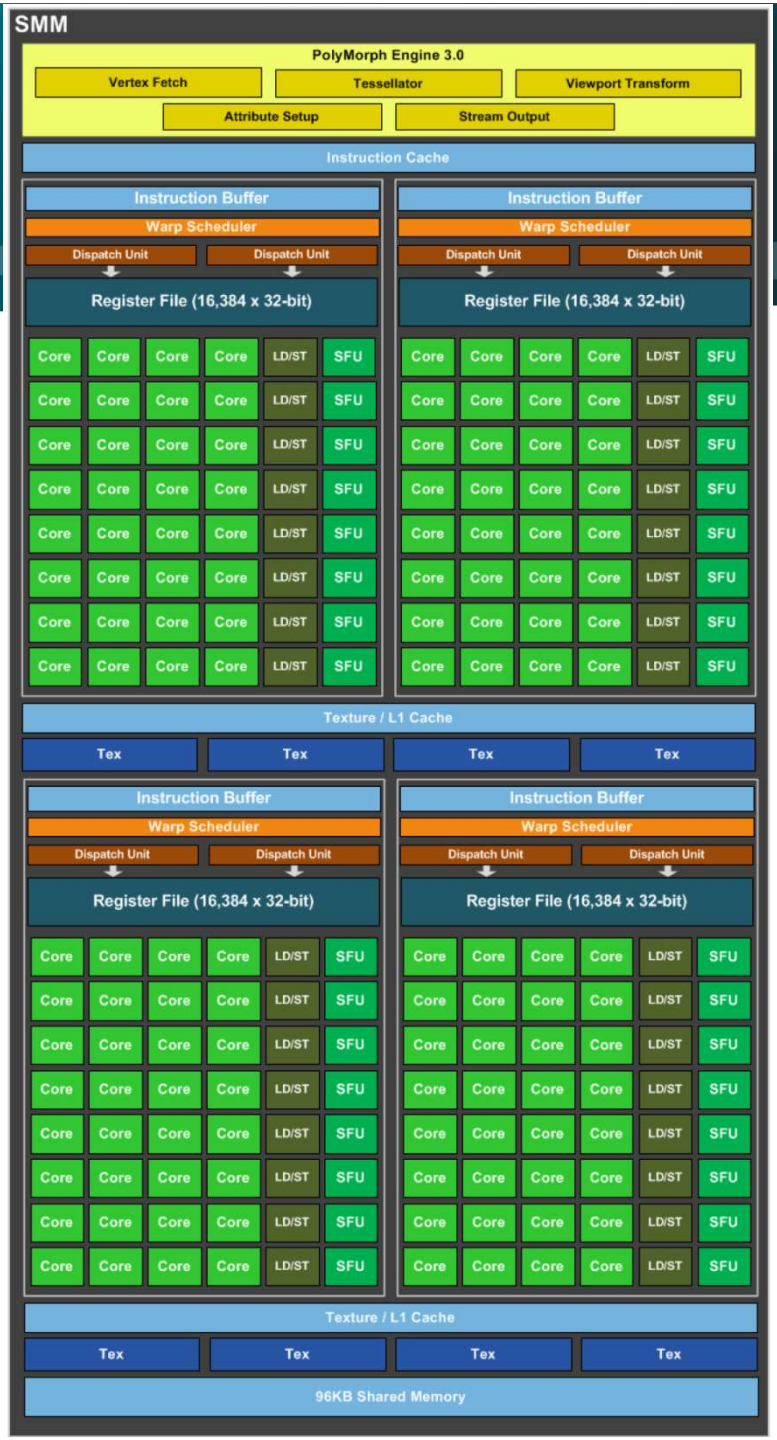
- 128 CUDA cores
- 4 DP units

4 partitions inside SMM

- 32 CUDA cores each
- 8 LD/ST units each
- Each has its own warp scheduler, two dispatch units, register file

Shared memory and L1 cache now separate!

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



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

A multiprocessor 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 multiprocessors 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)

H.5.2. Global Memory

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

H.5.3. Shared Memory

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

NVIDIA Pascal SM

Multiprocessor: SM

- 64 CUDA cores
- 32 DP units

2 partitions inside SM

- 32 CUDA cores each; 16 DP units each; 8 LD/ST units each
- Each has its own warp scheduler, two dispatch units, register file

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

A multiprocessor 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)

H.6.3. Global Memory

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

H.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 Carevout 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(...)
{
    ...
}

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

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

NVIDIA Volta SM

Multiprocessor: SM

- 64 FP32 + INT32 cores
- 32 FP64 cores
- 8 tensor cores (FP16/FP32 mixed-precision)

4 partitions inside SM

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

NVIDIA Turing SM

Multiprocessor: SM

- 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 each
- 2 Turing tensor cores each
- Each has: warp scheduler, dispatch unit, 16K register file

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

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 (1.5x *Volta*'s 128 KB capacity) and 128 KB for devices of compute capability 8.6.

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, Part 2)

I.7.2. Global Memory

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

I.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. 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 capability 8.6.

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, 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 capability 8.6 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 capability 8.6 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 GA100 SM

Multiprocessor: SM

- 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 each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA GA10x SM

Multiprocessor: SM

- 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

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

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.)

Thank you.