- Atomic single-precision floating-point adds on shared memory always operate with denormal support, i.e., behave equivalent to **`FADD.F32.RN`**.

In accordance to the IEEE-754R standard, if one of the input parameters to **`fminf()`**, **`fmin()`**, **`fmaxf()`**, or **`fmax()`** is NaN, but not the other, the result is the non-NaN parameter.

The conversion of a floating-point value to an integer value in the case where the floating-point value falls outside the range of the integer format is left undefined by IEEE-754. For compute devices, the behavior is to clamp to the end of the supported range. This is unlike the x86 architecture behavior.

The behavior of integer division by zero and integer overflow is left undefined by IEEE-754. For compute devices, there is no mechanism for detecting that such integer operation exceptions have occurred. Integer division by zero yields an unspecified, machine-specific value.

http://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus includes more information on the floating point accuracy and compliance of NVIDIA GPUs.

G.3. Compute Capability 2.x

G.3.1. Architecture

For devices of compute capability 2.x, a multiprocessor consists of:

- For devices of compute capability 2.0:
  - 32 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 4 special function units for single-precision floating-point transcendental functions,
- For devices of compute capability 2.1:
  - 48 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 8 special function units for single-precision floating-point transcendental functions,
- 2 warp schedulers.

At every instruction issue time, each scheduler issues:

- One instruction for devices of compute capability 2.0,
- Two independent instructions for devices of compute capability 2.1,

for some warp that is ready to execute, if any. The first scheduler is in charge of the warps with an odd ID and the second scheduler is in charge of the warps with an even ID. Note that when a scheduler issues a double-precision floating-point instruction, the other scheduler cannot issue any instruction.

A warp scheduler can issue an instruction to only half of the CUDA cores. To execute an instruction for all threads of a warp, a warp scheduler must therefore issue the instruction over two clock cycles for an integer or floating-point arithmetic instruction.

A multiprocessor also 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, both of which are used to cache accesses to local or global memory, including temporary register spills. 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.

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, using **`cudaFuncSetCacheConfig()`**/**`cuFuncSetCacheConfig()`**:

```
// Device code
__global__ void MyKernel(int* foo, int* bar, int a)
{
...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)

// Or via a function pointer:
void (*funcPtr)(int*, int*, int);
funcPtr = MyKernel;
cudaFuncSetCacheConfig(*funcPtr, cudaFuncCachePreferShared);
```

The default cache configuration is "prefer none," meaning "no preference." If a kernel is configured to have no preference, then it will default to the preference of the current thread/context, which is set using **`cudaDeviceSetCacheConfig()`**/**`cuCtxSetCacheConfig()`** (see the reference manual for details). If the current thread/context also has no preference (which is again the default setting), then whichever cache configuration was most recently used for any kernel will be the one that is used, unless a different cache configuration is required to launch the kernel (e.g., due to shared memory requirements). The initial configuration is 48 KB of shared memory and 16 KB of L1 cache.

Applications may query the L2 cache size by checking the **`l2CacheSize`** device property (see Device Enumeration). The maximum L2 cache size is 768 KB.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes four multiprocessors.

Each multiprocessor has a read-only texture cache to speed up reads from the texture memory space, which resides in device memory. It accesses the texture cache via a

texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory.

G.3.2. Global Memory

Global memory accesses are cached. Using the **-dlcm** compilation flag, they can be configured at compile time to be cached in both L1 and L2 (**-Xptxas -dlcm=ca**) (this is the default setting) or in L2 only (**-Xptxas -dlcm=cg**).

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.

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.

Figure 16 shows some examples of global memory accesses and corresponding memory transactions.

G.3.3. 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 two clock cycles.

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 (multiple words can be broadcast in a single transaction) and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).

This means, in particular, that there are no bank conflicts if an array of **char** is accessed as follows, for example:

```
extern __shared__ char shared[];
char data = shared[BaseIndex + tid];
```

Figure 18 shows some examples of memory read accesses that involve the broadcast mechanism for devices of compute capability 3.x. The same examples apply for devices of compute capability 2.x.

32-Bit Strided Access

A common access pattern is for each thread to access a 32-bit word from an array indexed by the thread ID **tid** and with some stride s:

```
extern __shared__ float shared[];
float data = shared[BaseIndex + s * tid];
```

In this case, threads **tid** and **tid+n** access the same bank whenever **s*n** is a multiple of the number of banks (i.e., 32) or, equivalently, whenever **n** is a multiple of **32/d** where **d** is the greatest common divisor of **32** and **s**. As a consequence, there will be no bank conflict only if the warp size (i.e., 32) is less than or equal to **32/d**, i.e., only if **d** is equal to 1, i.e., **s** is odd.

Figure 17 shows some examples of strided access for devices of compute capability 3.x. The same examples apply for devices of compute capability 2.x. However, the access pattern for the example in the middle generates 2-way bank conflicts for devices of compute capability 2.x.

Larger Than 32-Bit Access

64-bit and 128-bit accesses are specifically handled to minimize bank conflicts as described below.

Other accesses larger than 32-bit are split into 32-bit, 64-bit, or 128-bit accesses. The following code, for example:

```
struct type {
       float x, y, z;
};

extern __shared__ struct type shared[];
struct type data = shared[BaseIndex + tid];
```

results in three separate 32-bit reads without bank conflicts since each member is accessed with a stride of three 32-bit words.

64-Bit Accesses: For 64-bit accesses, a bank conflict only occurs if two threads in either of the half-warps access different addresses belonging to the same bank.

128-Bit Accesses: The majority of 128-bit accesses will cause 2-way bank conflicts, even if no two threads in a quarter-warp access different addresses belonging to the same bank. Therefore, to determine the ways of bank conflicts, one must add 1 to the maximum number of threads in a quarter-warp that access different addresses belonging to the same bank.

G.3.4. Constant Memory

In addition to the constant memory space supported by devices of all compute capabilities (where **__constant__ variables** reside), devices of compute capability 2.x support the **LDU** (LoaD Uniform) instruction that the compiler uses to load any variable that is:

- pointing to global memory,
- read-only in the kernel (programmer can enforce this using the const keyword),
- not dependent on thread ID.

G.4. Compute Capability 3.x

G.4.1. Architecture

A multiprocessor consists of:

- 192 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
- 32 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

When a multiprocessor is given warps to execute, it first distributes them among the four schedulers. Then, at every instruction issue time, each scheduler issues two independent instructions for one of its assigned warps that is ready to execute, if any.

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()**:

```
// Device code
__global__ void MyKernel()
{
    ...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferEqual: shared memory is 32 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
```

The default cache configuration is "prefer none," meaning "no preference." If a kernel is configured to have no preference, then it will default to the preference of the current thread/context, which is set using **cudaDeviceSetCacheConfig()/cuCtxSetCacheConfig()** (see the reference manual for details). If the current thread/context also has no preference (which is again the default setting), then whichever cache configuration was most recently used for any kernel will be the one that is used, unless a different cache configuration is required to launch the kernel (e.g., due to shared memory requirements). The initial configuration is 48 KB of shared memory and 16 KB of L1 cache.

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.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes three multiprocessors.

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.

G.4.2. 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**.

Caching in L2 behaves in the same way as for devices of compute capability 2.x (see Global Memory).

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 16 shows some examples of global memory accesses and corresponding memory transactions.

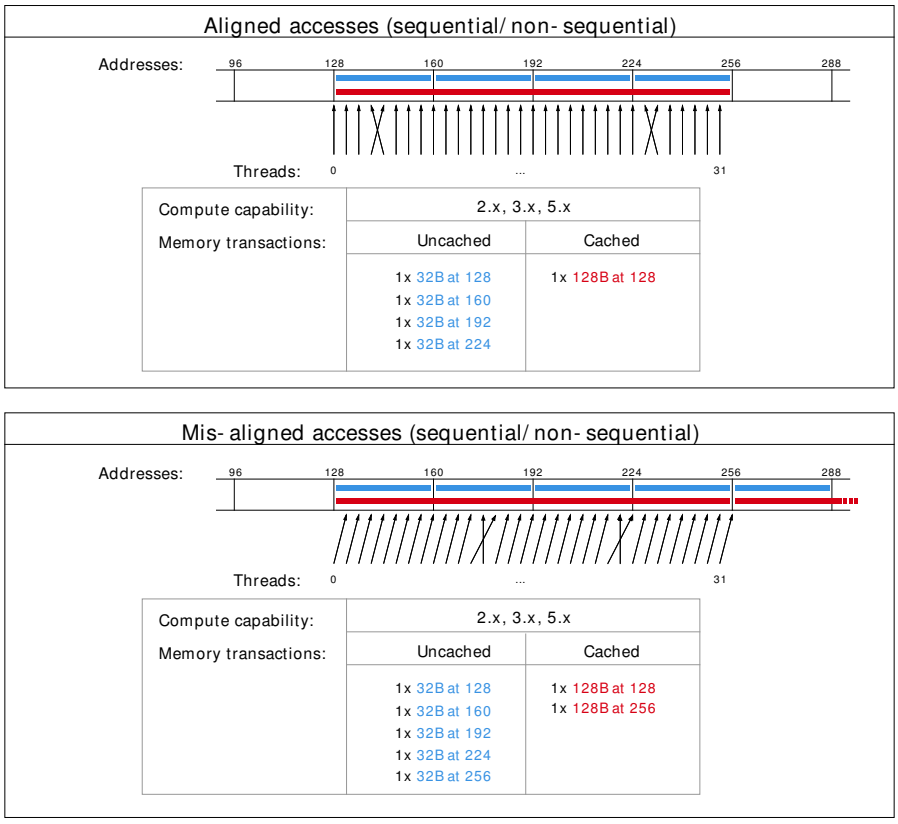

Figure 16 Examples of Global Memory Accesses

Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions for Compute Capabilities 2.x and Beyond

G.4.3. Shared Memory

Shared memory has 32 banks with two addressing modes that are described below.

The addressing mode can be queried using **`cudaDeviceGetSharedMemConfig()`** and set using **`cudaDeviceSetSharedMemConfig()`** (see reference manual for more details). Each bank has a bandwidth of 64 bits per clock cycle.

Figure 17 shows some examples of strided access.

Figure 18 shows some examples of memory read accesses that involve the broadcast mechanism.

64-Bit Mode

Successive 64-bit words map to successive banks.

A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 64-bit word (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 64-bit word is broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined).

In this mode, the same access pattern generates fewer bank conflicts than on devices of compute capability 2.x for 64-bit accesses and as many or fewer for 32-bit accesses.

32-Bit Mode

Successive 32-bit words map to successive banks.

A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 32-bit word or within two 32-bit words whose indices i and j are in the same 64-word aligned segment (i.e., a segment whose first index is a multiple of 64) and such that $j=i+32$ (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 32-bit words are broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined).

In this mode, the same access pattern generates as many or fewer bank conflicts than on devices of compute capability 2.x.

G.5. Compute Capability 5.x

G.5.1. Architecture

A multiprocessor consists of:

- 128 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
- 32 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

When a multiprocessor is given warps to execute, it first distributes them among the four schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

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.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes four multiprocessors.

G.5.2. 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 2.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:

- 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 some 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 multiprocessor's resources. These exceptions are reported by the profiler.

G.5.3. 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 17 shows some examples of strided access.

Figure 18 shows some examples of memory read accesses that involve the broadcast mechanism.