Working with Device Memory

September 30, 2016

Outline

Memory Considerations

There are two primary aspects of memory management when working with accelerators:

- Since the GPU is a physically separate device connected to the host via PCI-e bus, the host and device memories have separate address spaces.
- As a throughput oriented compute architecture, the GPU device itself provides five distinctly different memory types for use in kernel development and execution.

Many features in each CUDA release are centralized around robustification of the host/device memory interaction while mastery of the various device memory types is both an art and a science essential to every GPU accelerated application.

Programable Memory Hierarchy

- Both the CPUs and GPU compute devices use similar principles and models in memory hierarchy design (i.e. progressively lower-latency but lower capacity memories to optimize performance).
- The key difference in the GPU memory model is that CUDA programing exposes more of the device memory hierarchy and provides explicit programable control.

Device Memory Layout

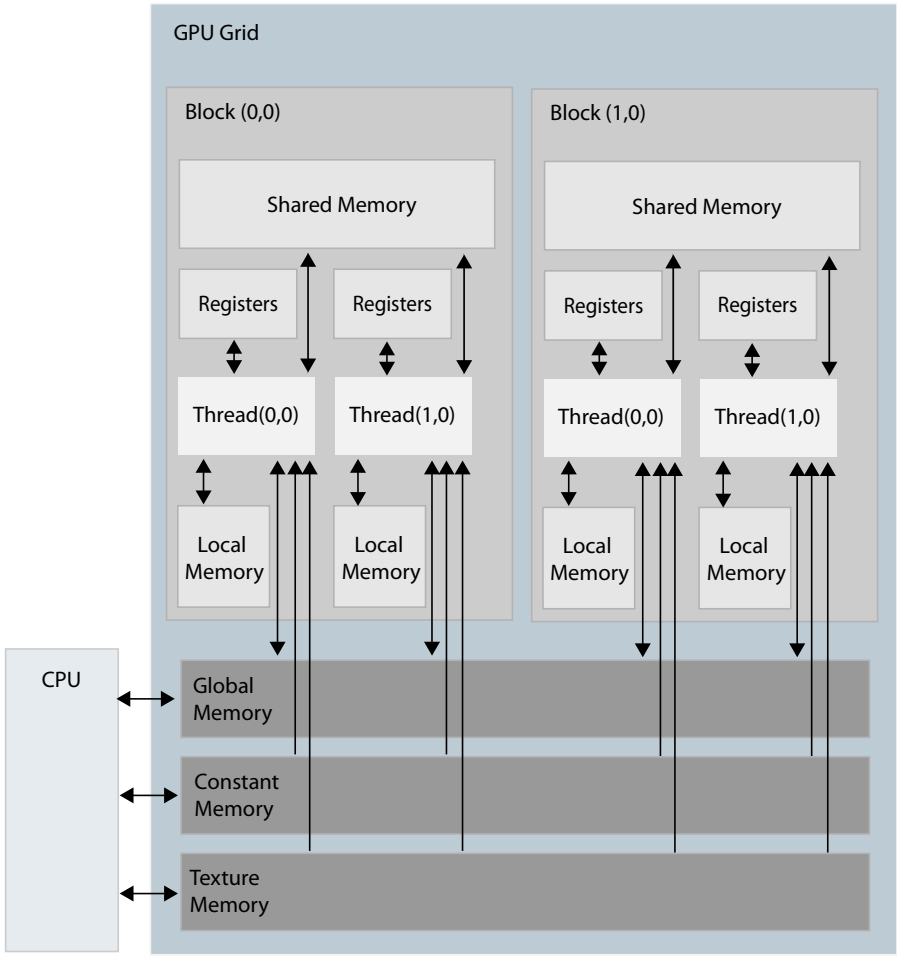

Figure: Various programable device memories. The host can only access global, constant, and texture memories. Each SM has a limited amount of shared memory which is partitioned among resident thread-blocks while registers are thread exclusive.

Device Global Memory

- Global memory is the largest host accessible memory available on the device which is accessible to all threads in any block.
- Global memory is off the processor chip (implies long access latencies) and is implemented using dynamic random access memory (DRAM) technologies such as double data rate type five synchronous graphics memory (GDDR5).
- Global memory is generally referred to as just *device memory* and is usually on the order of 6 to 12 GB in size.
- Since device memory is directly attached and accessed using a dedicated memory controller integrated into the GPU, the peak bandwidth is extremely high (typically 100 to 300 GB/s).
- The next generation Pascal P100 accelerator leverages a manufacturing technique called Chip-on-Wafer-on-Substrate (CoWoS) with second generation high bandwidth memory (HBM2) technology (i.e. stacked DRAM) achieving over 700 GB/s memory bandwidth.

Device Global Memory

- Whenever runtime API calls such as `cudaMalloc`, `cudaFree` and `cudaMemcpy` are invoked, global memory is being referenced.
- Device global memory is accessed in CUDA kernels using *device pointers* which reside in the *device address space*.
- Host code can perform pointer arithmetic on device pointers but can not generally dereference them.
- Keep in mind that mapped pinned pointers are located in host memory but can be accessed by the GPU (i.e. zero-copy). Although, on non-UVA systems, the host and device pointers to this memory are different and `cudaGetDevicePointer()` must be used to map host pointers to corresponding device pointers. But when UVA is in effect, the pointers are the same.

Device Global Memory

```
1 __global__ void GPUmemset(int *array, int value, size_t N)
2 {
3     for (int i = blockIdx.x*blockDim.x + threadIdx.x;
4           i < N;
5           i += gridDim.x*blockDim.x)
6     {
7           array[i] = value;
8     }
9 }
```

Listing 1: Device global memory accessed by CUDA kernel using device pointer. This kernel writes the integer `value` into the address range given by `array` and `N`. The references to `blockIdx`, `blockDim` and `gridDim` enable the kernel to operate correctly for each individual thread with whatever block and grid parameters were used in the kernel launch.

Error Correcting Codes (ECC)

- Tesla GPUs (SM 2.x and later) have the ability to run with error correction. Devices with ECC enabled can silently correct single-bit errors and report double-bit errors.
- ECC can be queried, enabled, and disabled using the `nvidia-smi` command-line tool or the NVIDIA Management Library (NVML) C-based API directly.
- ECC consumes a portion of global memory ($\approx 12\%$), used to store redundancy, and reduces the overall memory bandwidth ($\approx 20\%$) due to the additional traffic for memory checksums. The exact impact of ECC on bandwidth depends on the memory access pattern.
- Note that ECC coverage extends to other memories on the device such as L2/L1 cache and register file space. However, facilities such as hardware queues, thread block scheduler, warp scheduler, instruction dispatch unit, etc are not covered by ECC.

Error Correcting Codes (ECC)

- ECC status can be queried via the runtime API via device property `ECCEnabled` and/or driver API device attribute `CU_DEVICE_ATTRIBUTE_ECC_ENABLED` which are 1 if the device has ECC support turned on, or 0 otherwise.
- When an uncorrectable ECC error is detected, CUDA runtime calls will return `cudaErrorECCUncorrectable` while the driver API returns `CUDA_ERROR_ECC_UNCORRECTABLE`.
- No notification of ECC errors is available within a CUDA kernel. ECC errors are reported, by the driver to the host, once the launch has completed. Any ECC errors which arise during execution of a nested program (i.e. kernel launch within a kernel) will either generate an exception or continue execution (depending upon error and configuration).
- With ECC enabled, the memory error events will be recorded in the InfoROM for retrieval (typically using `nvidia-smi`).

Error Correcting Codes (ECC)

- It is very important to note that typically a single instance of a single-bit-error (SBE) will be silently corrected on-the-fly by the device and thus requiring no user intervention.
- However, once a double-bit-error (DBE) has been detected and flagged a `cudaErrorECCUncorrectable` will be triggered and all further execution on the device will be halted.
- An uncorrectable ECC error will cause the device context to become unusable and, in fact, the device becomes unusable until a reset is performed, or the driver is unloaded/reloaded (which effectively resets the GPU). Therefore all subsequent kernel launch attempts will fail until a reset occurs.
- If ECC is not enabled, then execution will always continue, even in the presence of memory errors.
- The ECC mechanism is not guaranteed to flag or detect errors more complex than a double-bit-error.

Error Correcting Codes (ECC)

- Depending on the memory access pattern, a single flipped bit may cause the error counters to increment more than once.
- It is possible that multiple single-bit-errors could trigger `cudaErrorECCUncorrectable`
- There is currently no way to artificially induce `cudaErrorECCUncorrectable` for testing purposes.
- Note that double-bit ECC errors are additionally reported as XID errors by the driver to the operating system kernel/event log (i.e. `/var/log/messages`).
- The NVIDIA Validation Suite (NVVS) is a health check and stress test tool provided as part of the GPU Deployment Kit. The NVVS tool facilitates checks for basic GPU health, including the presence of ECC errors, PCIe problems, bandwidth issues, and other general problems.

ECC Error Rates at Oak Ridge

- The Oak Ridge TITAN Supercomputer was the first GPU based supercomputer to perform over 10 petaFLOPS.
- TITAN includes 18,688 Tesla GPUs and was ranked number 1 on the TOP500 when it became operational on Oct 29, 2012.
- Total combined GPU memory capacity is roughly 112TB.
- TITAN sees on average one double-bit GPU memory error per week over all devices.
- See full report by NCCS director Jim Rogers: *GPU Errors on HPC Systems: Characterization, Quantification, and Implications for Architects and Operations*.
- See also Tiwari et al, *Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation*.

Dynamic Page Retirement

- The NVIDIA driver supports “retiring” of bad framebuffer memory cells from device global memory. Read the docs.
- The act of marking a memory cell for exclusion is called “retiring”, while the act of actually excluding the associated page from memory allocations is called “blacklisting”.
- The device driver will retire a cell after a single instance of a double-bit ECC error or two instances of a single-bit ECC error. The addresses of memory pages with retired cells are stored in the device InfoROM.
- When each device is initialized, the driver will retrieve retired page addresses from the InfoROM. These retired addresses are then provided to the framebuffer manager for blacklisting such that they cannot be used by the driver.

Dynamic Page Retirement

- This dynamic memory cell retirement strategy can improve the longevity of an otherwise good board and is thus an important resiliency feature on supported products.
- Dynamic page retirement is available on all driver-supported Linux and Windows TCC platforms provided:
  - Driver version R319 and newer
  - K20 and newer Tesla enterprise products
- Dynamic page retirement is not supported on any NVIDIA Quadro, GRID or Geforce products.
- Both XID errors in system logs and `nvidia-smi` provide visibility into page retirement. For example

```
1    nvidia-smi -i <target gpu> -q -d  PAGE_RETIREMENT
```

Dynamic Page Retirement

- Marking cells for retirement will only occur when ECC is enabled. However, once a cell has been retired the associated page will *always* be blacklisted by the driver, even if ECC is later disabled.
- However, page retirement itself can *never* be disabled. All previously retired pages will continue to be excluded in every future allocation.
- The SBE recurrence threshold for cell retirement is not configurable (i.e. always 2).
- The size of each blacklisted page is 4 KB and does reduce total available memory.
- A maximum of 192 retired cell addresses can be stored in the InfoROM but limited to 60 blacklisted pages at any one time.
- The maximum total size of blacklisted memory is $\approx$ 256 KB.

Global Memory: Coalescence

- For best performance when reading and writing global memory, CUDA kernels must perform coalesced[1] memory transactions.
- Recall that instructions are dispatched on a per-warp basis. Therefore, when a warp performs a load/store, the number of memory transactions needed to satisfy the instruction depends on two factors:
  - Distribution of memory addresses across the 32 threads/warp.
  - Alignment of memory addresses per transaction.
- In general, fewer memory transactions needed to fulfill a load/store instruction across a warp means better kernel perf.
- For SM 2.x and later, achieving load/store memory throughput efficiency is more relaxed because memory transactions are cached in L1/L2.

[1] Definition: co·a·lesce: come together and form one mass or whole.

Global Memory: Coalescence

- Memory transactions are coalesced on a per-warp basis.
- There is a (simplified) set of criteria for reads and writes performed by a warp to be coalesced:

    1. The words must be at least 32 bits in size. Reading or writing 8-bit or 16-bit words is always uncoalesced (Pascal: FP16).
    2. The addresses being accessed by the threads of the warp much be contiguous and increasing (e.g. offset by the thread ID).
    3. The base address of the warp (i.e. the address being accessed by the first thread in the warp) must be aligned at 64-byte, 128-byte, 256-byte for word sizes of 32-bit, 64-bit, and 128-bit respectively.

Global Memory: L1 and L2 Cache

- SM 2.x and later include non-programable L1/L2 caches.
- The L2 cache services the entire chip where as L1 caches are per-SM and may be user configured to be 16KB or 48KB.
- The cache lines for both L1 and L2 are 128 bytes and map to 128-byte aligned segments in device memory.
- Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions, whereas L2-only cache hits are serviced with smaller 32-byte transactions.
- The compiler can emit instructions that cache memory accesses in both L1 and L2 using options `-Xptxas -dlcm=ca` or enforce L2-only cache load mechanism using options `-Xptxas -dlcm=cg`.
- Kernels performing scattered memory accesses can improve cache utilization using L2-only load strategy since requests are service with 32-byte transactions.

Global Memory: L1 and L2 Cache

- L1 cache is enabled by default for global memory loads on Fermi devices but *disabled* by default on K40 and later GPUs.
- On Kepler K10, K20, and K20x devices, L1 cache is *not* configurable and is used exclusively to service thread-local register spillover.
- **Q**: So if L1 cache mechanism is not enable for global memory loads, then what is the device doing with L1?
- **A**: L1 is *always* used to support local memory and shared memory.
- Therefore when L1 is enabled for global memory loads, there is potential for considerable L1 cache contention as global memory load operations could inadvertently evict other essential thread-local and shared data out to L2 or even DRAM(!).

LMEM: Application Binary Interface

- Local memory is used to implement what is called the application binary interface (ABI). An ABI determines the calling convention which controls how function arguments are passed and return values retrieved.
- The ABI determines, for example, whether all parameters are passed on the stack or some are passed in registers, which registers are used for which function parameters, and whether the first function parameter passed on the stack is pushed first or last onto the stack. So on and so forth ...
- Adhering to the ABI is usually the job of the compiler.

Local Memory Overview

- Programmatically, local memory contains the stack for every thread in a CUDA kernel: local variables that cannot be held in registers, parameters, and return addresses for subroutines.
- Physically, local memory is backed by the same pool of device DRAM as global memory.
- Logically, local memory differs from global memory in that addressing is resolved by the compiler and load/store are *always* cached in L1.
- The hardware reads from global memory using LD/ST instructions but uses special load and store instructions for local memory: LDL/STL
- Only device threads can access local memory addresses.
- Note, registers are not indexible. Therefore, arrays declared inside kernels are stored in local memory if the compiler cannot resolve indexing.

Local Memory Usage

- The amount of local memory needed by a given kernel with the **nvcc** options `-Xptxas -v,abi=no`.
- The amount of local memory used by a kernel can also be queried at runtime using `cuFuncGetAttribute()` with attribute `CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES`.
- Each kernel launch receives a preallocated local memory buffer. If a kernel launch requires more local memory than the default provisioning, the driver must allocate a new buffer in device DRAM before the kernel can launch which takes extra time and can cause unexpected host/device synchronization.
- If the driver is unable to alloc the larger buffer in device global memory, the kernel launch will fail. This is one of the very few scenarios that can cause a kernel launch failure at runtime.
- The flag `cudaDeviceLmemResizeToMax` instructs the driver to retain and reuse large LMEM buffer allocations. This can prevent thrashing allocations when launching many kernels with high LMEM usage.

Register Spilling

- Local kernel variables in excess of the allotted register capacity overflow or "spill" into local memory.
- Register spilling to local memory can incur two costs: an increased number of instructions and an increase in the amount of memory traffic.
- Spillover impact is limited if LMEM bytes are contained in L1. Furthermore, additional instructions might not matter if code is not instruction-throughput limited.
- The maximum number of 32-bit registers available per thread is architecture dependent: 124 for SM 1.x, 63 for SM 2.x, 63 for SM 3.x, and 254 for SM 3.5 and later.
- The number of registers used by a kernel can be obtained by passing the verbose option to the PTX assembler (i.e. **`nvcc`** options `-Xptxas -v`).

Register Spilling: Performance Strategies

- The maximum number of registers can be lowered by using **nvcc** options `-maxrregcount`.
- A higher value will generally increase the performance of individual GPU threads that execute this function.
- Keep in mind that each SM has a fixed number of available registers. A higher value of this option will likely reduce occupancy and thus execution efficiency may be degraded. Might still be an overall win since reduces total bytes accessed.
- The CUDA ABI requires a minimum of 16 registers per thread. Register counts below the ABI minimum will be modified by `ptxas` to satisfy the ABI specification.
- The `__launch_bounds__` directive (kernel attribute) can be used to tune per-thread register counts when the kernel is being compiled online by the PTX assembler.
- A kernel launch will fail if the attempted thread-block size exceeds a launch bound specification.

Register Spilling: Performance Strategies

- As noted earlier, when L1 is enabled for global memory loads, there is potential for considerable L1 cache contention as global memory load operations could inadvertently evict LMEM cache lines off-chip to L2. If evicted from L2, the LMEM data will be stored off-world in global DRAM.
- Again, use options `-Xptxas -dlcm=cg` to enforce L2-only cache load mechanism for global memory requests (default on K40 and later devices).

Register Spilling: Performance Strategies

- The L1 cache is per-SM and is physically implemented in the same hardware as shared memory.
- On K40 and later devices (i.e. Maxwell architecture) the size of L1 is configurable via `cudaFuncSetCacheConfig()` and `cudaDeviceSetCacheConfig()` with argument:

```
1  enum __device_builtin__ cudaFuncCache
2  {
3      /** Default cache configuration, no preference */
4      cudaFuncCachePreferNone   = 0,
5      /** Prefer 48K shared memory and 16K L1 cache  */
6      cudaFuncCachePreferShared = 1,
7      /** Prefer 16K shared memory and 48K L1 cache  */
8      cudaFuncCachePreferL1     = 2,
9      /** Prefer 32K shared memory and 32K L1 cache  */
10     cudaFuncCachePreferEqual  = 3
11 };
```

- The hardware can change this configuration per kernel launch, but it is expensive and will break concurrency for concurrent kernel launches.

Shared Memory

- Shared memory (SMEM) is an important type of memory in CUDA programing which is used to exchange data between threads within a block. SMEM is declared in a kernel with `__shared__ float sdata[128]` for example.
- Programmatically, shared memory could the thought of as a "scratchpad" that can be used for fast data interchange.
- Physically, SMEM is backed by a per-SM 64KB on-chip cache which is functionally partitioned as 48K, 32 K, or 16K shared using `cudaDeviceSetCacheConfig()`.
- Since SMEM can be explicitly allocated and referenced, it can be thought of as a "manually managed" cache.
- In terms of speed, SMEM is perhaps 10x slower than register access but 10x faster than accessing global DRAM.
- As a result, SMEM is often a critical resource to reduce the external bandwidth needed by CUDA kernels.

Shared Memory

- Any shared memory declared in the kernel itself is automatically allocated for each block at kernel launch.
- Unsized declarations of SMEM in a kernel must be specified when the kernel is launched.

```
1   extern __shared__ int sharedInts[];
```

Listing 2: Unsized declaration of SMEM

```
1   KERNEL<<<gridSize, blockSize, smemSize>>>(args)
```

Listing 3: Kernel launch syntax specifying size of SMEM

- The definition of `smemSize` is often something like `blockSize*sizeof(int)` since `blockSize` specifies number of threads in a block.
- Note that kernels using unsized shared memory must be compiled in separate source code files.

Shared Memory

- Kernels using SMEM typically are written in three phases
  1. Load SMEM and `__syncthreads()`
  2. Process SMEM and `__syncthreads()`
  3. Write results
- The `__syncthreads()` term is a barrier synchronization primitive which can not be passed until all threads in a block have executed the instruction.
- The compiler will report SMEM usage for kernels using the PTX assembler options `-Xptxas -v`.
- The amount of SMEM used by a kernel can be determined at runtime using `cuFuncGetAttribute()` with attribute `CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES`
- SMEM variables used in warp-synchronous programing must be declared `volatile` to prevent the compiler from performing optimization that could render the code incorrect.

Shared Memory Organization

- To achieve high memory bandwidth, the SMEM partition of on-chip memory is organized into 32 equally sized modules called *banks* (32 banks since there are 32 threads in a warp)
- Each independent SMEM bank can be accessed concurrently but only once per transaction.
- When threads in a warp read different addresses within the same bank a *bank conflict* occurs and the reads must be serviced by multiple transactions.
- Therefore, if an SMEM load operation issued by a warp does not access more than one memory location per bank, the operation is serviced by a single memory transaction. That is, all 32 load operations are serviced by a single transaction .

SMEM Banks: Single Address Access

- Note that single address reads are broadcast to all requesting threads. That is, when all threads in a warp read the exact same address within a single bank only a single memory transaction is executed.
- However, single address writes are not defined. That is, when all 32 threads in a warp write to the same address, the word is written by a single thread. Which thread actually performs this write is undefined.

SMEM Bank Access Patterns

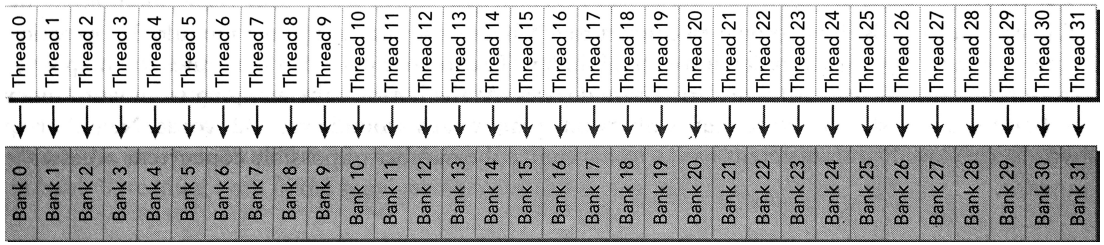

Figure: No bank conflicts, single memory transaction

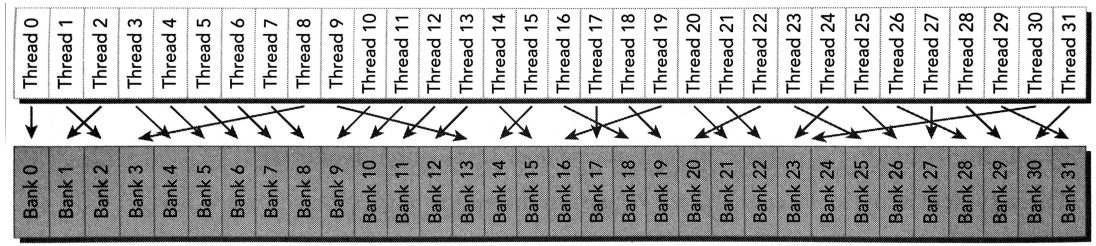

Figure: Random but, no bank conflicts, single transaction.

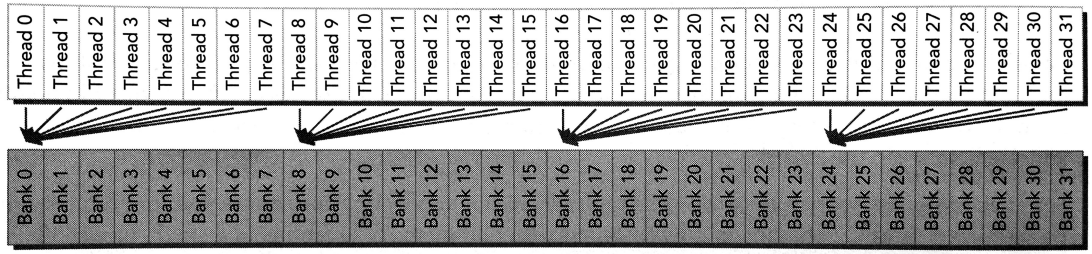

Figure: Potential bank conflict if reading different address in same bank. Keep in mind that store operations are not defined in this situation. Which thread actually performs the write is undefined.

SMEM Bank Width

- With a 4 byte SMEM bank width, successive 32-bit words map to successive banks. In general, the bank index for a byte address is calculated as mod(addr/width,#banks).
- SMEM *bank width* defines which addresses are in which bank.
- The default memory bank width varies by device compute capability: 4 bytes with 2.x and 8 bytes with 3.x.
- Kepler architecture and higher (+3.x) support a configurable bank width via the runtime API function `cudaSetSharedMemConfig()` which accepts the following configurations
  - `cudaSharedMemBankSizeDefault`
  - `cudaSharedMemBankSizeFourByte`
  - `cudaSharedMemBankSizeEightByte`
- The active bank width configuration can be queried at runtime with `cudaGetSharedMemConfig()` returning either a 4 byte or 8 byte configuration.

SMEM Bank Bandwidth

- On Kepler (3.x) architecture, SMEM has 32 banks, where each bank has a bandwidth of 64-bits per clock cycle.
- On Fermi (2.x) architecture, SMEM has 32 banks, but the bandwidth per bank is only 32-bits per clock cycle.
- On Kepler, with SMEM bank width of 32-bits, it is possible to read 64-bits from a bank in a single clock cycle but only pass the 32-bits requested to each thread. For example, in this configuration, threads in a warp reading word 0 and word 32 in the same memory request would not result in a bank conflict.
- In some situations, memory padding can be used as a mitigation strategy to avoid bank conflicts.
- More details on shared memory performance here.

SMEM Organization: 32x32

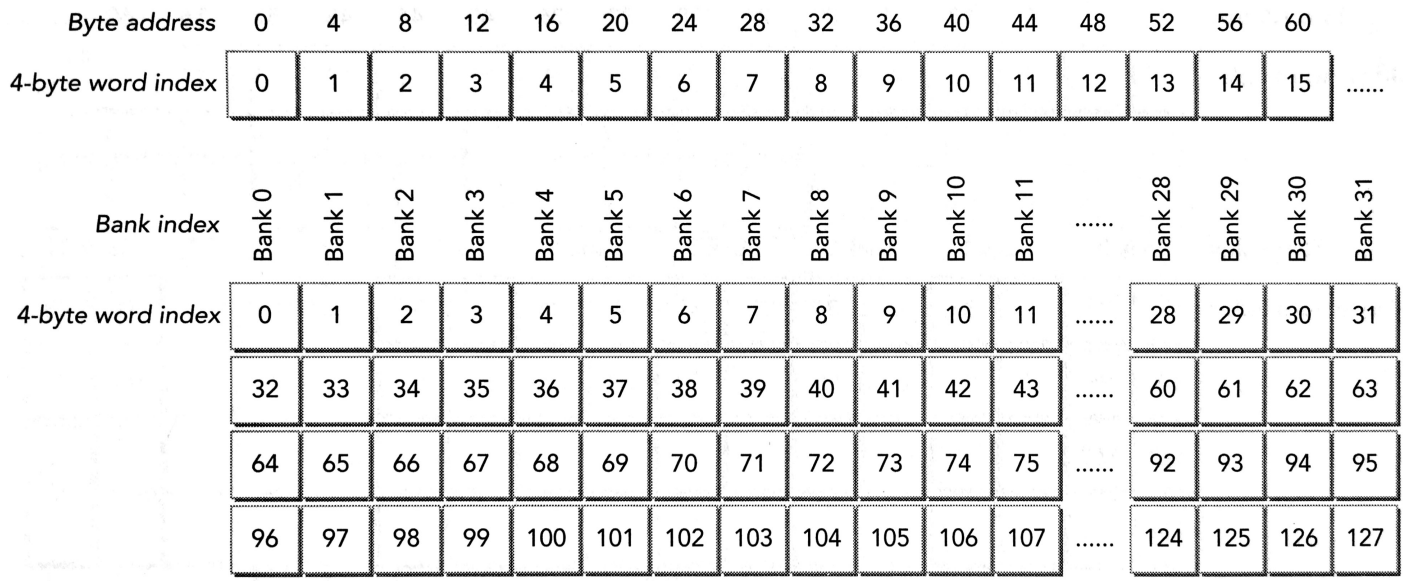

Figure: A bank width of 32-bits with 32-bits per clock cycle (Fermi, 2.x). Therefore, only a single 4 byte word from each bank is retrievable per memory transaction. Notice here that double precision access always incur a bank conflict.

SMEM Organization: 32x64

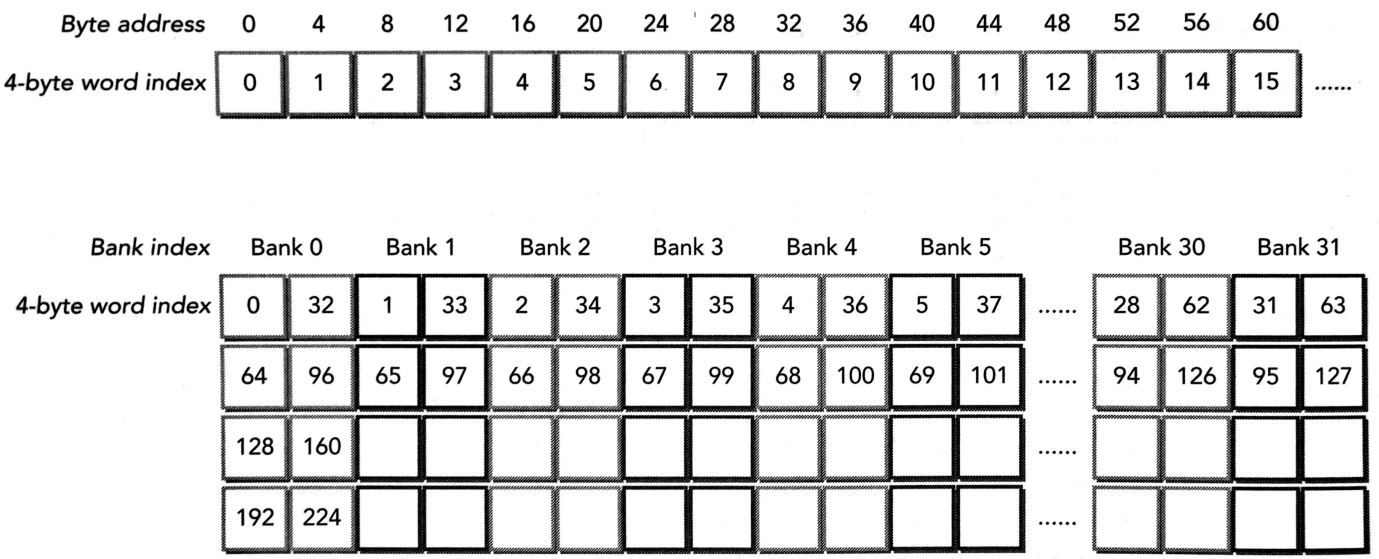

Figure: A bank width of 32-bits with 64-bits per clock cycle bandwidth (Kepler, 3.x). Here two 32-bit words are accessible from each bank per memory transaction. Again, in this configuration, threads in a warp reading word 0 and word 32 in the same memory request would not result in a bank conflict. Although, misaligned access of words 32 and 64 in the same request would indeed be a bank conflict and be serviced with multiple transactions. Depending on access patterns, memory padding can be used as a mitigation strategy to avoid bank conflicts.

Constant Memory

- Constant memory is optimized for read-only broadcast to multiple threads. Although constant memory actually resides in device DRAM, it is accessed using different instructions that cause the device to access it using a dedicated per-SM on-chip "constant cache".
- The compiler uses constant memory to hold constants that could not be easily computed or otherwise compiled directly into the machine code. The compiler has 64KB of memory available to use at its discretion and the developer has another 64KB of memory available.
- CMEM must be declared in global scope using the `__constant__` keyword and is both read and write accessible from the host at runtime using `cudaMemcpyToSymbol()`. CMEM variables exists for the lifetime of the program and are accessible from any thread in any kernel.

Constant Memory?

- One might expect that `__constant__` memory is analogous to the `const` keyword in C/C++ where it connot be changed after initialization. However CMEM can not only be modified by the host at runtime but *also* by querying the pointer to `__constant__` memory using `cudaGetSymbolAddress()` and writing to it with a kernel.
- However, kernels must not write to `__constant__` memory ranges that they are accessing because the constant cache is *not* kept coherent with respect to the rest of the memory hierarchy during kernel execution. That is, threads in a block which modify constant memory while executing on SM(i) are note reflected in the constant caches of other SMs.
- Runtime API call `cudaMemcpyFromSymbol()` can be used to copy data *out of* CMEM.

Constant Memory: Driver API

- Driver API applications can query the device pointer of constant memory using `cuModuleGetGlobal()`.
- The driver API does not include special memory copy functions like `cudaMemcpyToSymbol()` since it does not have the language integration of the CUDA runtime.
- Driver applications must query the address with `cuModuleGetGlobal()` and then call `cuMemcpyHtoD()` or `cuMemcpyDtoH()`.
- The amount of constant memory used by a kernel may be queried using `cuFuncGetAttribute()` with attribute `CU_FUNC_ATTRIBUTE_CONSTANT_SIZE_BYTES`.

Register Shuffle

- With SM 3.0 and higher, the *shuffle* instruction was introduced as a mechanism to allow threads to directly read the registers of another thread in the same warp.
- The shuffle instruction allows threads in a warp to cooperate without having to consume SMEM or travel out to DRAM.
- The shuffle instruction has lower latency than SMEM and does not consume any additional memory to perform the register exchange. Using shuffle rather than SMEM for intra-warp exchange can improve occupancy.
- A thread must know which of the 32 threads it is within the warp (this is called a *lane*) and it is often necessary to know which warp the thread is in (i.e. the warp index). These values are readily computed as:

```
laneIdx = threadIdx.x % 32
warpIdx = threadIdx.x / 32
```

Register Shuffle

- There are two sets of shuffle instructions: ones for integers and another set for floats where each set has four variants:
  - shfl.idx: indexed any-to-any (e.g. read value from lane 2)
  - shfl.up: shift right to n^{th} neighbor (mod 32)
  - shfl.down: shift left to n^{th} neighbor (mod 32)
  - shfl.xor: butterfly exchange (i.e threads 1 and 2 swap values, threads 3 and 4 swap values, and so on ...)

```
1    __global__ void lane_broadcast (int const laneIdx)
2    {
3        int value = threadIdx.x;
4        value = __shfl(value, laneIdx, 32);
5    }
```

Listing 4: Here all threads execute the shuffle instruction to read the register local variable `value` from lane specified by `laneIdx`.

- More details on shuffle instructions and performance here as well as various NVIDIA blogs.

Memory Profiling with **nvprof**

- The **nvprof** profiling tool provides considerable visibility into device memory usage both in the form of summary statistics and timeline traces of device activities in chronological order.
- By default **nvprof** runs in Summary mode
- There are various summary mode *events* and *metrics* which can be enabled with options: `--events <event>` and `--metrics <metric>`
- Events are hardware counters observed during the application execution while metrics are calculated based on the events.
- Use options `--query-events` and `--query-metrics` to list all supported events and metrics
- Where there are multiple devices avaliable, use option `--devices <device IDs>` to restrict the profiling scope to specific devices.

Global Memory Profiling Metrics

- For optimal performance, global memory accesses should be aligned and coalesced. Any other access pattern will result in reply of memory requests. To monitor DRAM load and store performance in a kernel use metrics `gld_efficiency` and `gst_efficiency`
- Additionally, the `g[ld|st]_transactions_per_request` metric can be used to monitor the average number of global memory load/store transactions per memory request
- The total number of memory transactions can be queried using the metric `g[ld|st]_transactions` which provides the total number of global memory load and store transactions per kernel.
- Finally, use metric `g[ld|st]_throughput` to compare global memory throughput with theoretical maximum values.

SMEM Profiling Metrics and Events

- The major performance consideration when using SMEM is bank conflicts. To check if bank conflicts occur, use metric `shared_[load|store]_transactions_per_request` where the optimal value is 1 transaction per SMEM request. Higher values indicate replayed memory requests.
- The total number of bank conflicts can be check directly with the event `l1_shared_bank_conflict`.
- Likewise, use the `shared_[load|store]` events to collect the number of instructions executed for SMEM load/store instructions (does not include replayed transactions).
- While the number of replayed transactions per instruction is not provided as a metric it can be calculated indirectly as

$$\frac{L1\_shared\_bank\_conflict}{shared\_load + shared\_store}$$

Register Profiling Metrics and Events

- When local kernel variables require more memory than is available through a thread's allocated registers, the compiler will spill excess to local memory. And as discussed, register spilling can significantly inhibit kernel performance.
- To monitor register spilling, there are L1 and L2 profile counters which can be used to analyze cache performance via the **`nvprof`** events:
  - `l1_local_[load|store]_hit`
  - `l1_local_[load|store]_miss`
- Similarly, the metric `l1_cache_local_hit_rate` reports the hit rate in L1 cache for local loads and stores. If more local loads and stores are being performed then more spilling must of occurred.