INTRODUCTION TO GPU ARCHITECTURE & PROGRAMMING

COMP4300/8300 PARALLEL SYSTEMS

PROF. JOHN TAYLOR

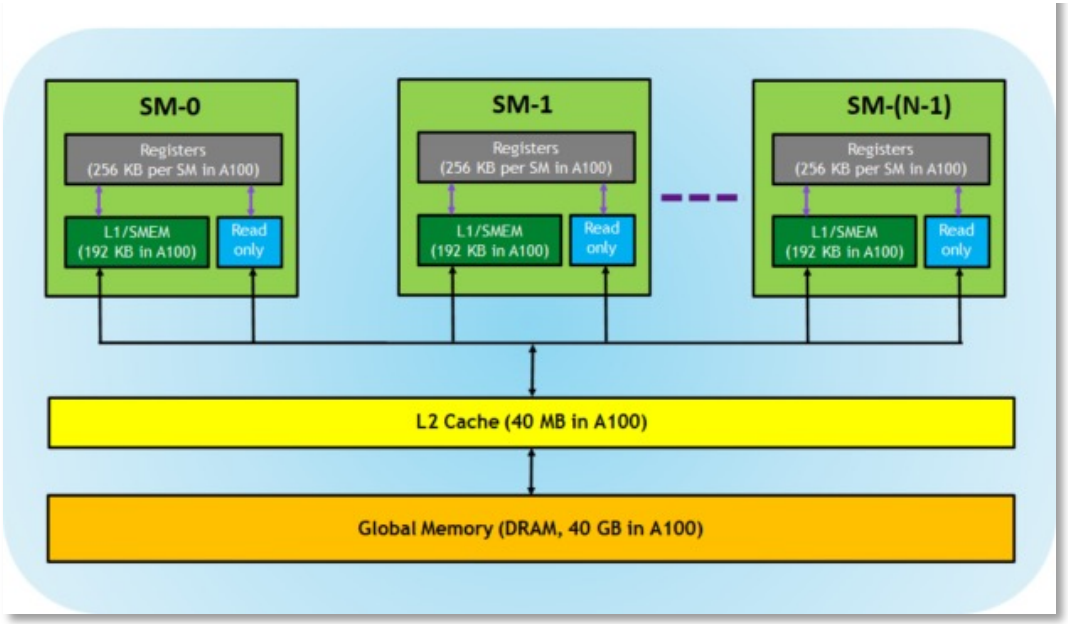
APRIL 2024

Australian National University

Logistics

- Attendance to the Lab sessions is highly encouraged. Most of the practical aspects of the programming models are covered in the Labs.

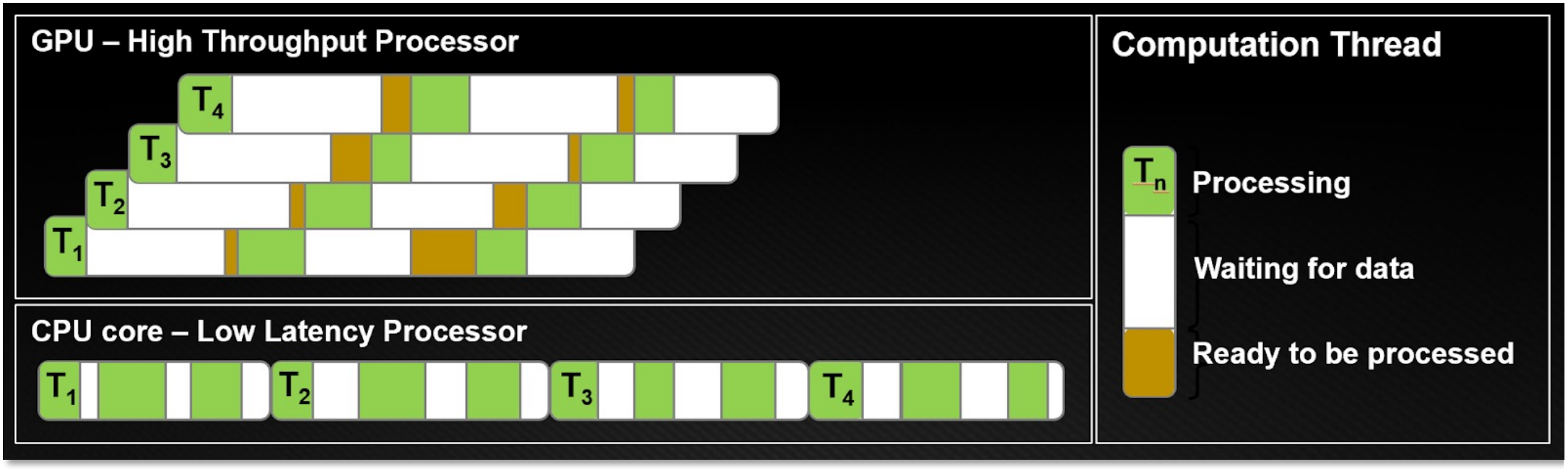
GPU Memory Model & Management



"The NVIDIA CUDA compiler does a good job in optimizing memory resources, but an expert CUDA developer can choose to use this memory hierarchy efficiently to optimize the CUDA programs as needed."

The following memories are exposed by the GPU architecture:

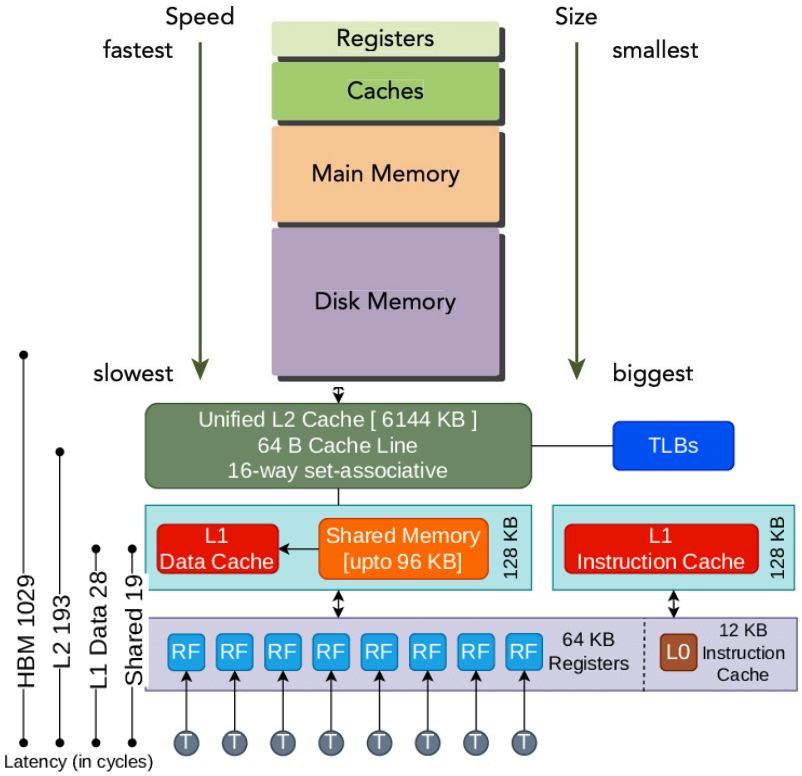
- **Registers**—These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.
- **L1/Shared memory (SMEM)**—Every SM has a fast, on-chip scratchpad memory that can be used as L1 cache and shared memory. All threads in a CUDA block can access shared memory, and all CUDA blocks running on a given SM can share the physical memory resource provided by the SM.
- **Read-only memory**—Each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.
- **L2 cache**—The L2 cache is shared across all SMs, so every thread in every CUDA block can access this memory. The NVIDIA A100 GPU has increased the L2 cache size to 40 MB as compared to 6 MB in V100 GPUs.
- **Global memory**—This is the framebuffer size of the GPU and DRAM sitting in the GPU.



- **CPU architectures** must minimize latency within each thread.
  - On CPUs, every thread minimizes the data access time (white bars). During a single time slice, thread gets work done as much as possible (green bar).
  - To achieve this, CPUs require low latency, which requires large caches and complex control logic.
  - Caches work best with only a few threads per core, as context switching between threads is expensive.

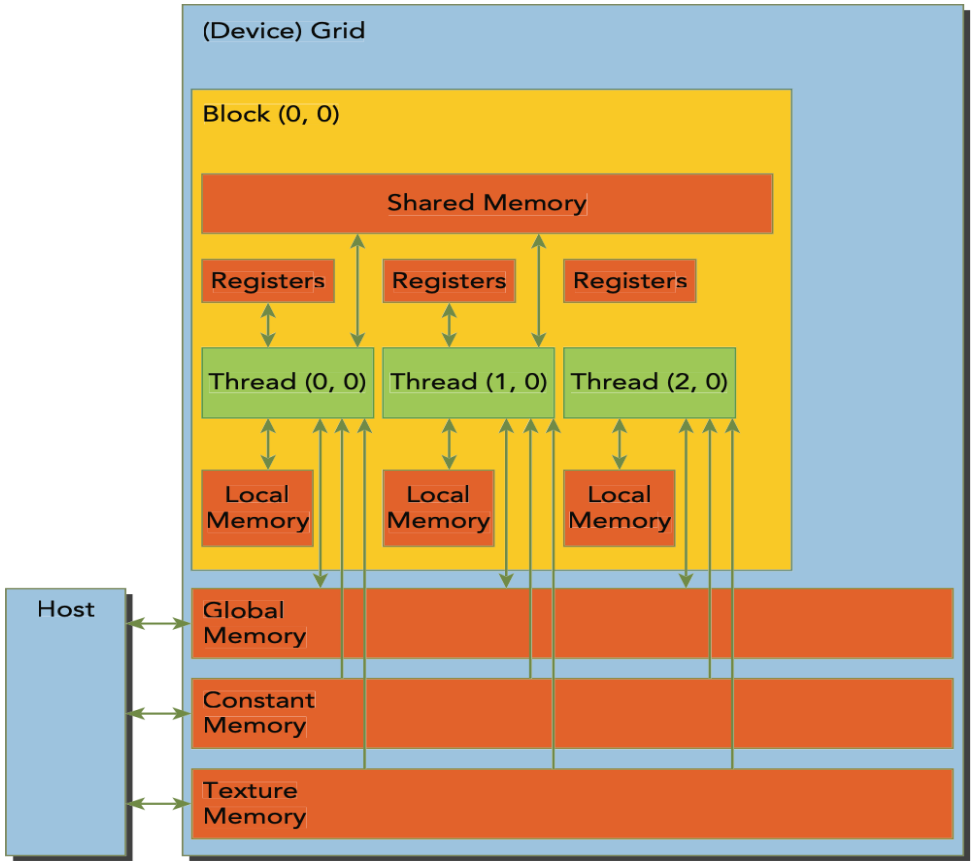
- **GPU architecture** hides instruction and memory latency with computation. In GPUs, threads are lightweight, so a GPU can switch from stalled threads to other threads at no cost as often as every clock cycle.

The GPU Memory Hierarchy



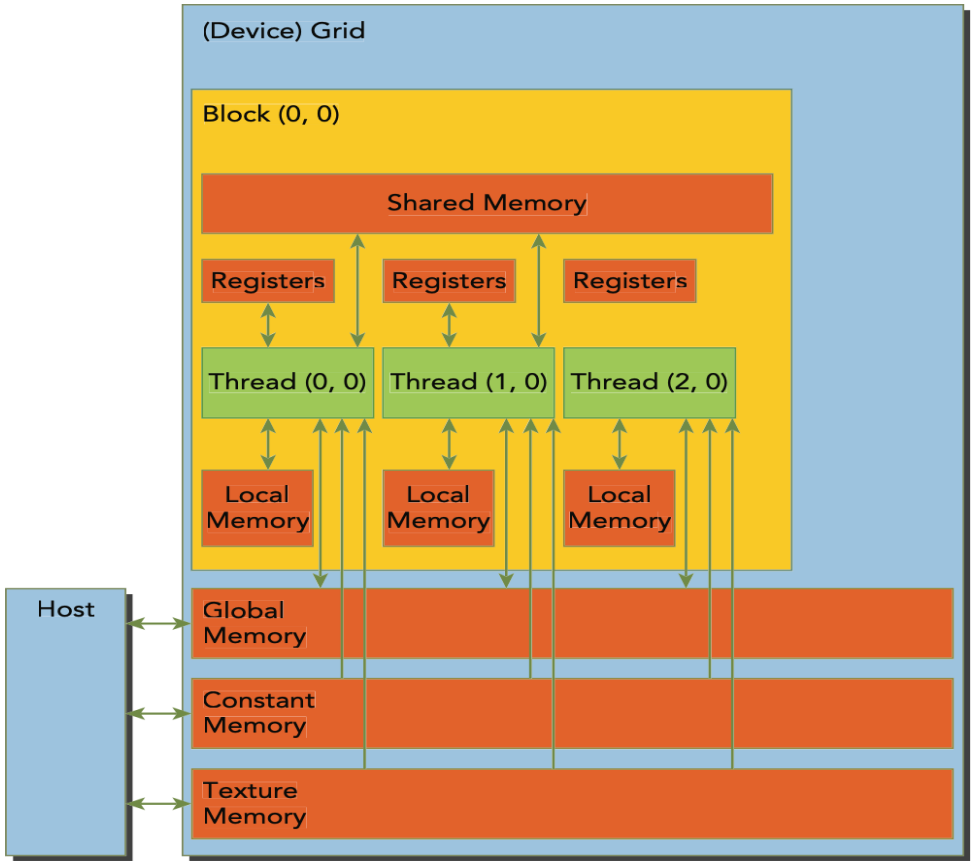
- Typical memory hierarchy of progressively lower-latency but lower-capacity memories to optimize performance
- Main memory on modern GPUs is made of High-Bandwidth Memory (HBM, better BW than DDR)
- V100 has a 16-32 GB of HBM with a BW of 900 GB/s, latency up to 1029 cycles
- Unified L2 Cache, on V100 6 MB, with BW of 900 GB/s, latency 193 cycles
- Two types lower-level caches → L1 and Shared Memory
- V100 has 64 KB of L1 **per SM** with hit latency of 28 cycles
- V100 has 64 KB of Shared Memory **per SM** with a peak hit latency (no-conflict) of 19 cycles, and BW of 14.8 TB/s
- Each SM has a 256 KB register file

The GPU Memory Model



- While *L1 and L2 cache* remain non-programmable, the CUDA memory model exposes many additional types of programmable memory:
  - Registers, shared memory, local memory, constant memory, texture memory and global memory.
- Each memory type has a different scope, lifetime, and caching behavior
- A *thread* in a kernel has its own private local memory
- A *thread block* has its own shared memory, visible to all threads in the same thread block
- All threads can access *global memory*
- The *constant* and *texture memory* spaces are read-only

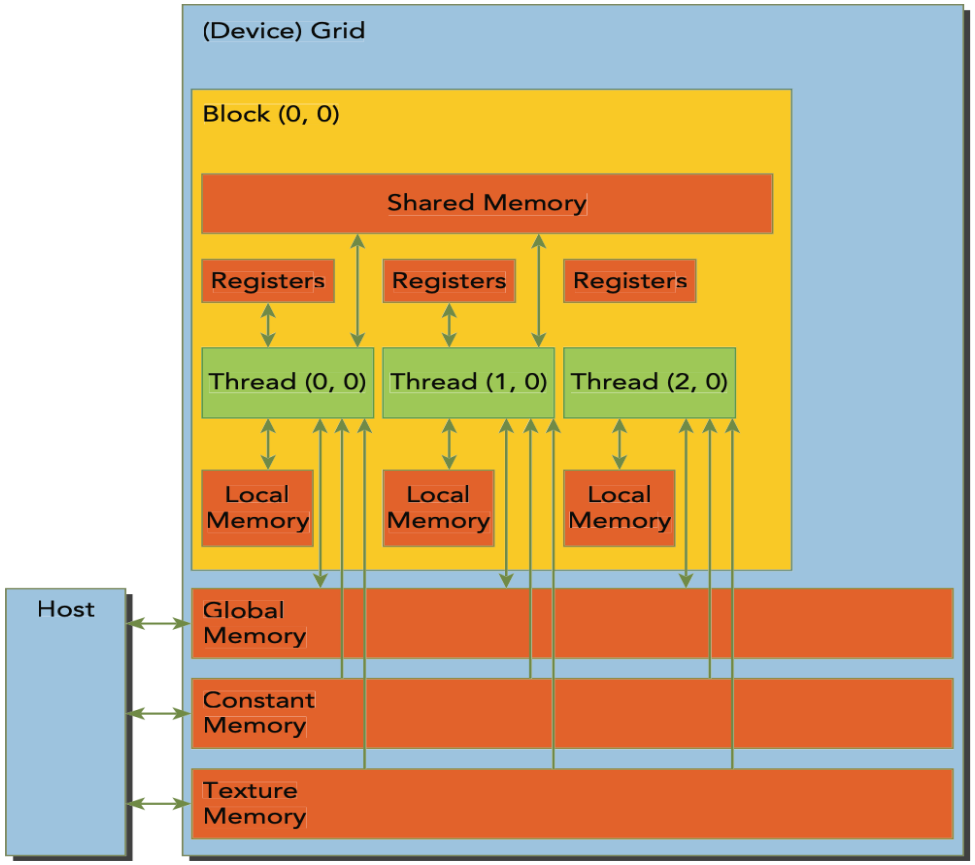
The GPU Memory Model



- **Registers**
  - fastest memory space
  - Automatic variable declared in a kernel is generally stored in a register
  - Register variables are thread-private
  - On V100 one thread can use maximum 255 registers
  - If a kernel uses more registers than this limit the excess registers will spill over to local memory, significantly impinging on performance

- **Local Memory**
  - Stores all thread-private variables that cannot fit into the register space Local arrays with indices whose values cannot be determined at compile-time
  - **Warning:** values spilled to local memory reside in the same physical location as global memory!

The GPU Memory Model



- **Shared Memory (ShM)**
  - Stores variables decorated with the `__shared__` attribute
  - High-bandwidth and lower-latency memory (basically a programmable cache)
  - Threads within same block can cooperate by sharing data stored in shared memory
  - Access to shared memory must be synchronized using `-syncthreads()`
- **Constant Memory**
  - Stores variables decorated with the `-constant-` attribute
  - Resides in device memory and is cached in a dedicated, per-SM constant cache.
  - It is read-only, must be statically declared and must initialized by the host using `cudaMemcpyToSymbol`
  - Good for constants because of the dedicated caching

Summary of Memory Type Mappings and Features

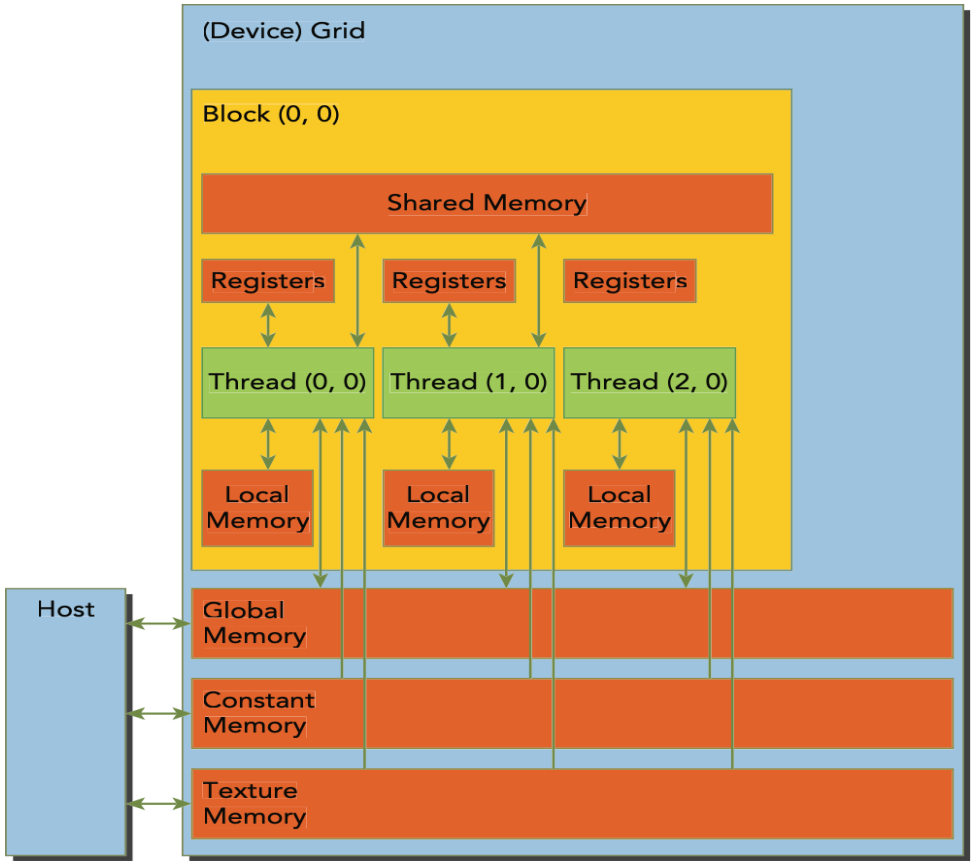
QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	`float var`	Register	Thread	Thread
	`float var[100]`	Local	Thread	Thread
`__shared__`	`float var` †	Shared	Block	Block
`__device__`	`float var` †	Global	Global	Application
`__constant__`	`float var` †	Constant	Global	Application

† Can be either scalar variable or array variable

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x

The GPU Memory Model



Global Memory

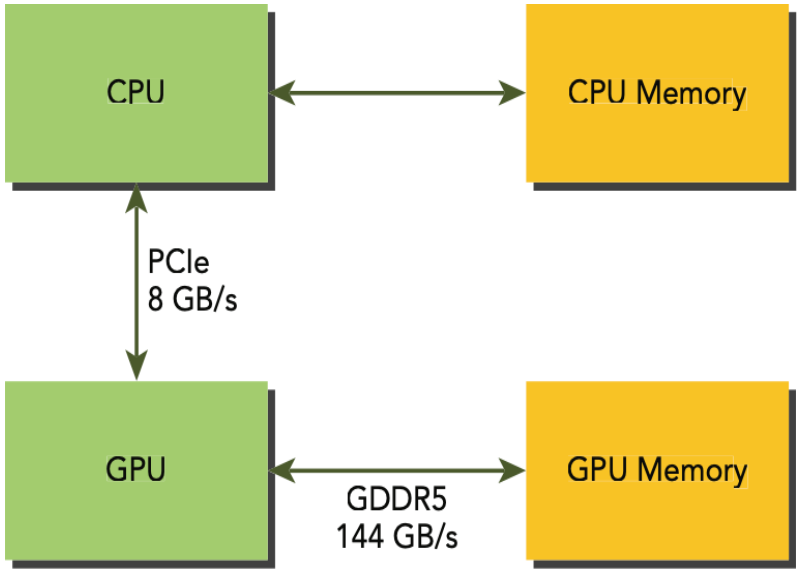
- Largest, highest-latency memory
- Global memory variables can be declared statically, using `__device__` Declared dynamically using `cudaMalloc` and released using `cudaFree`
- Danger of data hazards when multiple threads access it, as for CPUs except that threads cannot sync across blocks!
- Optimizing global memory access on GPU device is crucial for performance (see later slides!)

DEMO:

CUDA Memory Model

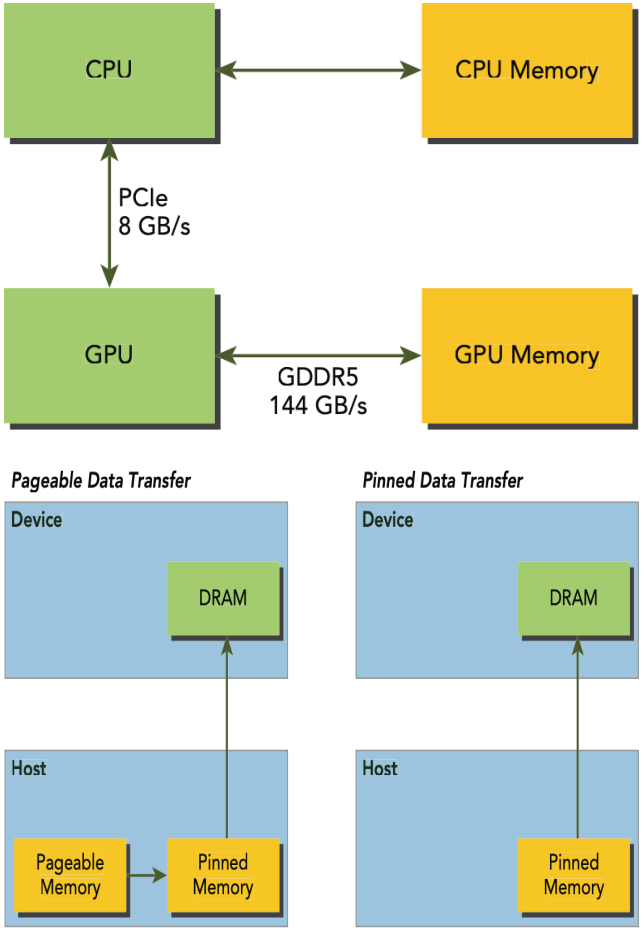
CUDA Memory Management

Host ↔ Device Data Transfers



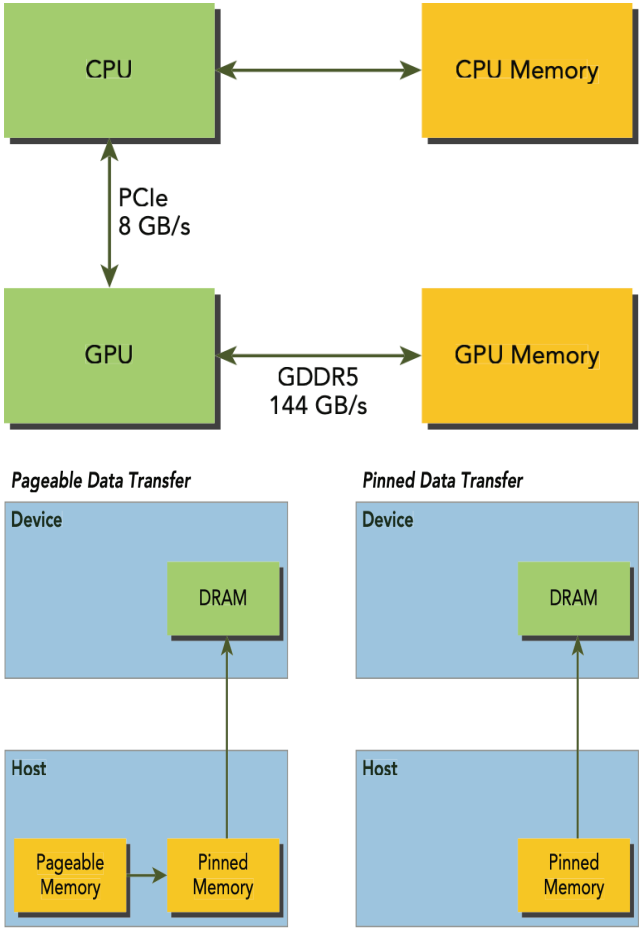
- Once global memory is allocated (`cudaMalloc`), you can transfer data between host and device using `cudaMemcpy`
- Due to the low relative bandwidth of the CPU↔GPU interconnect, host↔device transfers can throttle overall application performance
- Nvlink provides higher performance
- What actually happens when we call `cudaMemcpy`?

Host ↔ Device Data Transfers



- What actually happens when we call `cudaMemcpy`?
- The allocated host memory (`Malloc`) is by default *pageable*, that is, subject to page fault operations that move data in host virtual memory to different physical locations as directed by the operating system.
- The GPU cannot safely access data in pageable memory
- In a transfer: -
  - The CUDA driver allocates temporary *pinned* (page-locked) host memory
  - Copies the source host data to pinned memory
  - Transfers the data from pinned memory to device memory

Host ↔ Device Data Transfers

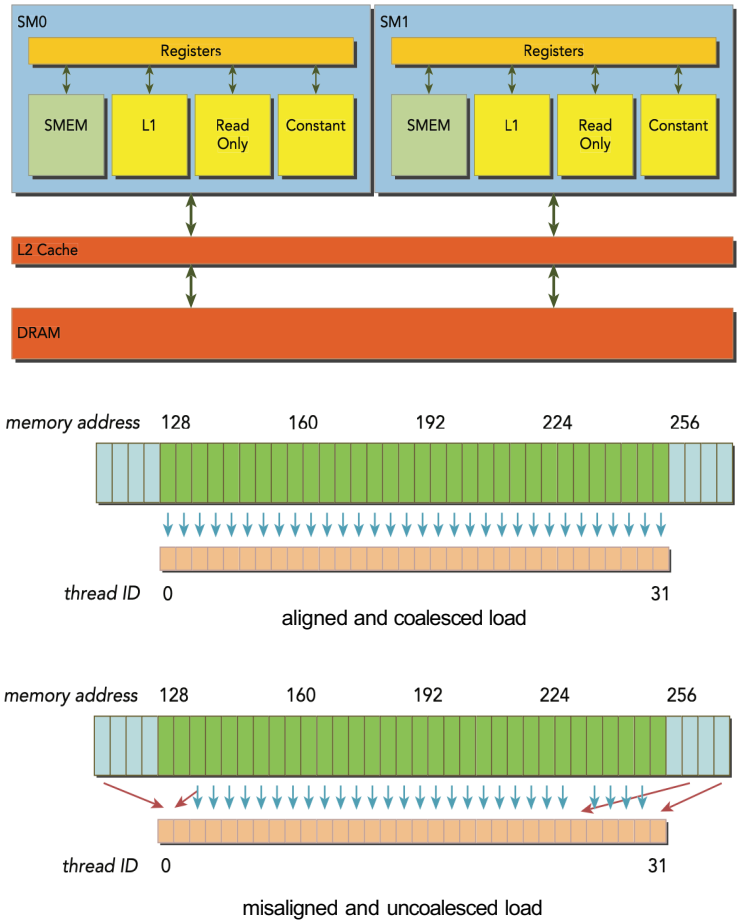


- In a transfer
  - The CUDA driver allocates temporary *pinned* (page-locked) host memory
  - Copies the source host data to pinned memory
  - Transfers the data from pinned memory to device memory
- You can avoid the extra copy (and improve both latency and bandwidth) by directly allocating host-pinned memory using `cudaError_t cudaMallocHost(void**hPtr, size t count)`
- Pinned host memory must be freed with `cudaError_t cudaFreeHost(void *ptr)`

DEMO:

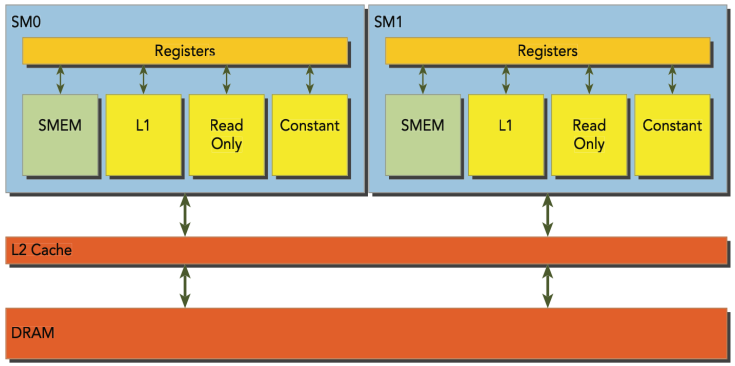
Host Device Transfers

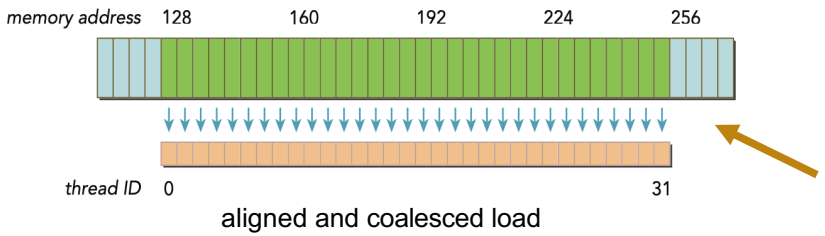
Global Memory Access Patterns



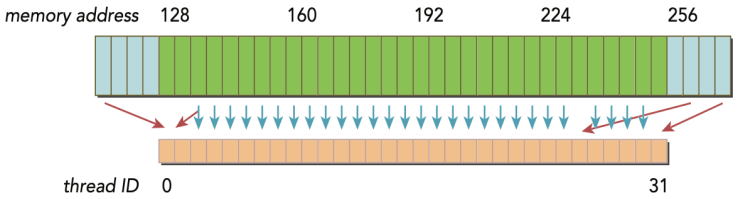
- Kernel memory requests issued `per warp` and are served between the device memory and SM on-chip memory using either 128-byte or 32-byte transactions
- All accesses to global memory go through the L2 cache.
- Accesses also pass through the L1 cache, depending on the type of access and your GPU's architecture
- On devices of compute capability 6.0 or higher, L1-caching is the default
- L1 cache lines have a size of 128 bytes, and it maps to four 32-byte aligned segments in device memory
- If each thread in a warp requests one4-byte value, that results in 128 bytes of data per request

Global Memory Access Patterns





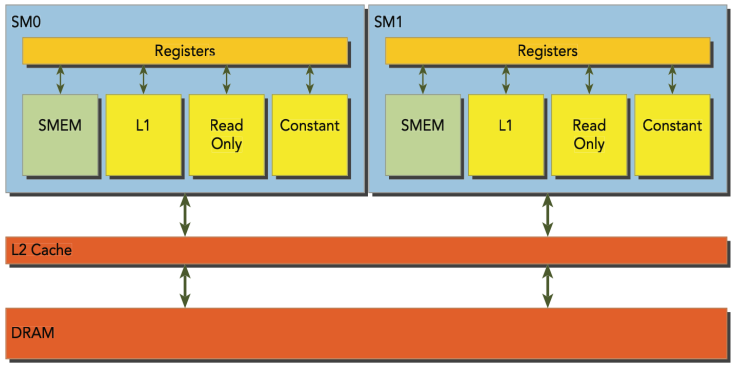
aligned and coalesced load

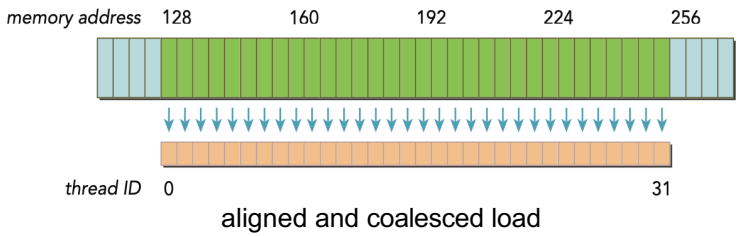


misaligned and uncoalesced load

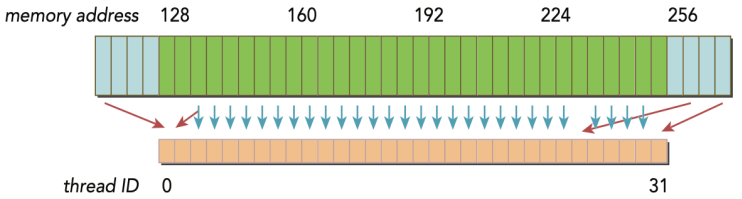
- L1 cache lines have a size of 128 bytes, and it maps to four 32-byte aligned segments in device memory
- If each thread in a warp requests one 4-byte value, that results in 128 bytes of data per request
- There are two types of device memory accesses
  - *Aligned memory accesses*: The first address of a device memory transaction is a multiple of 32 bytes
  - *Coalesced memory accesses*: All 32 threads in a warp access a contiguous chunk of memory
- Aligned coalesced memory accesses are ideal
- Misaligned and/or uncoalesced memory transactions will cause a loss of bandwidth efficiency!

Global Memory Access Patterns: Cached Loads





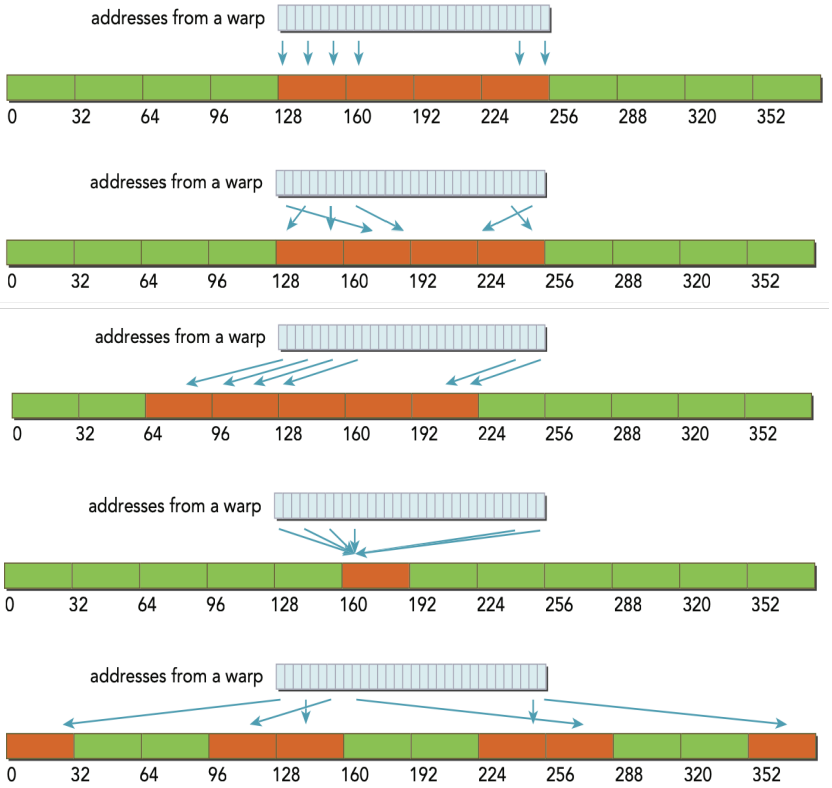
aligned and coalesced load



misaligned and uncoalesced load

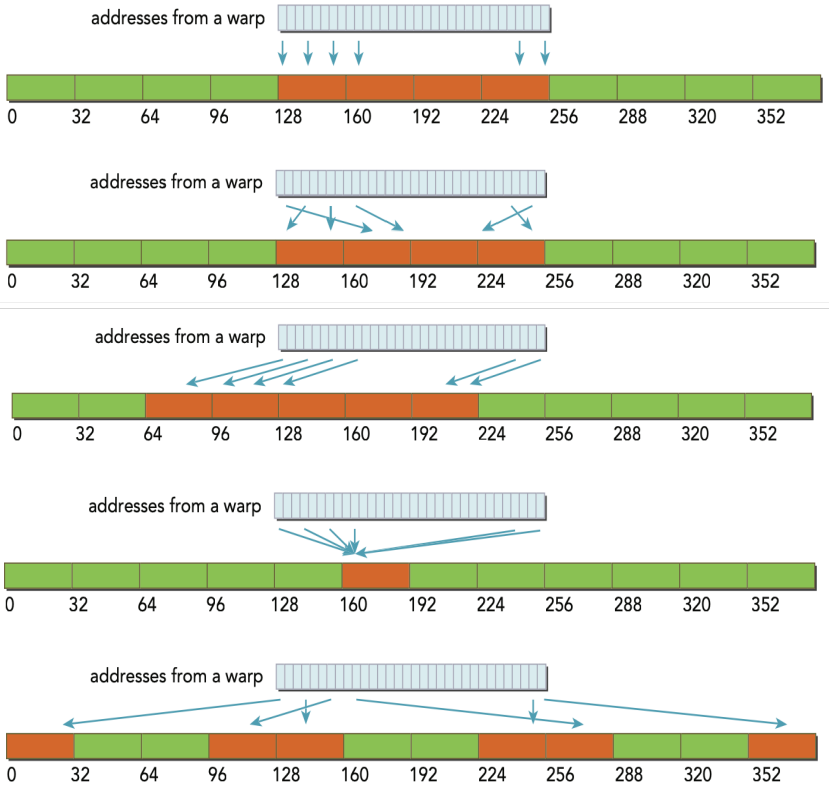
- On modern GPUs all loads go through L1-cache (cached loads)
- Even if an L1 cache line size is 128 bytes, the memory transaction from global memory to cache are performed at a granularity of 32-byte segments
- In aligned and coalesced loads a warp accesses 32 consecutive words (e.g. 4 byte words) with the starting address in global memory being a multiple of 32 bytes
- This will be performed using 4 parallel 32-byte memory transactions → cache-global-memory bus utilization (efficiency) will be 100 %.
- The cache controller will assemble the 4 blocks of data into a single cache line
- Subsequent accesses to the same cache line can be satisfied with a single cache read transaction

Global Memory Access Patterns: Cached Loads



- If memory access is aligned and thread accesses are not sequential, but randomized within a 128-byte range → load efficiency 100%
- If sequential threads in a warp access memory that is sequential but not aligned with a 32-byte segment, five 32-byte segments will be requested → load efficiency 80%
- If all threads in the warp request the same data → efficiency 4 bytes requested / 32 bytes loaded = 12.5%
- If warp requests 32 4-byte words scattered across global memory → efficiency 128 bytes requested / 1024 bytes loaded = 12.5%

Global Memory Access Patterns: Stores



- Before the Volta architecture stores were not L1-cached, they were only cached in the L2 cache before being sent to device memory
- Since Volta stores are cached but the L1 cache is write-through. In the context of an L1 cache, a “write-through” policy means that when data is written to the L1 cache, it is also immediately written to the next level of cache - L2
- They are also performed at a 32-byte segment granularity
- Store efficiency considerations are analogous to reads

Memory Performance Tuning

There are two goals to strive for when optimizing device memory bandwidth utilization:

- Aligned and coalesced memory accesses that reduce wasted bandwidth
- Sufficient concurrent memory operations to hide memory latency

Maximization of concurrent memory accesses is also necessary and achievable by:

- Increasing the number of independent memory operations performed within each thread.
- Experimenting with the execution configuration of a kernel launch to expose sufficient parallelism to each SM.

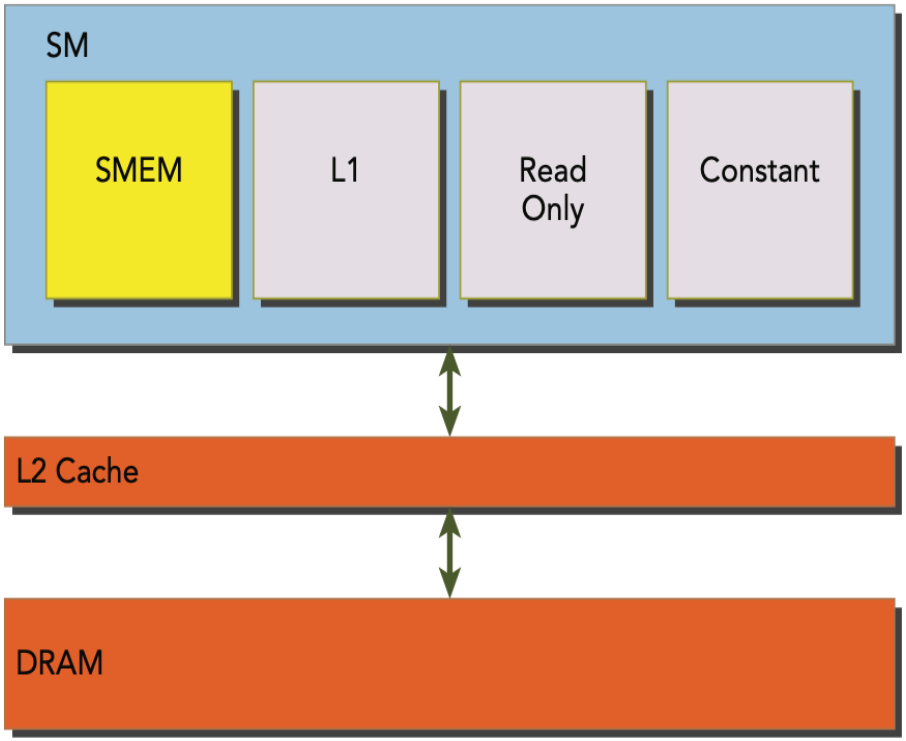
Unrolling loops that contain memory operations adds more independent, memory operations to the pipeline.

- You can unroll loops by using `#pragma unroll X`
- By default, the compiler unrolls small loops with a known trip count

If you use the `-O3` flag the compiler will attempt loop unrolling more aggressively.
The impact of loop unrolling on GPU code performance is very large

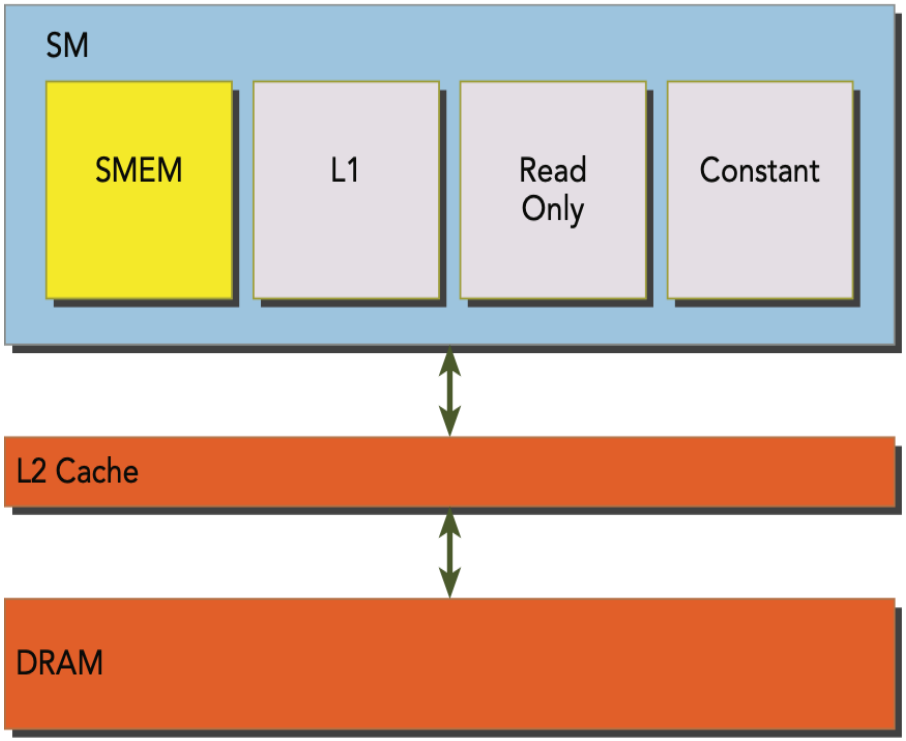
- `arrayReadKernel <<< 32768, 512 >>>` → 0.001825 s
- `arrayReadKernelUnroll4 <<< 32768, 512 >>>` → 0.000599 s

Shared Memory (ShM)



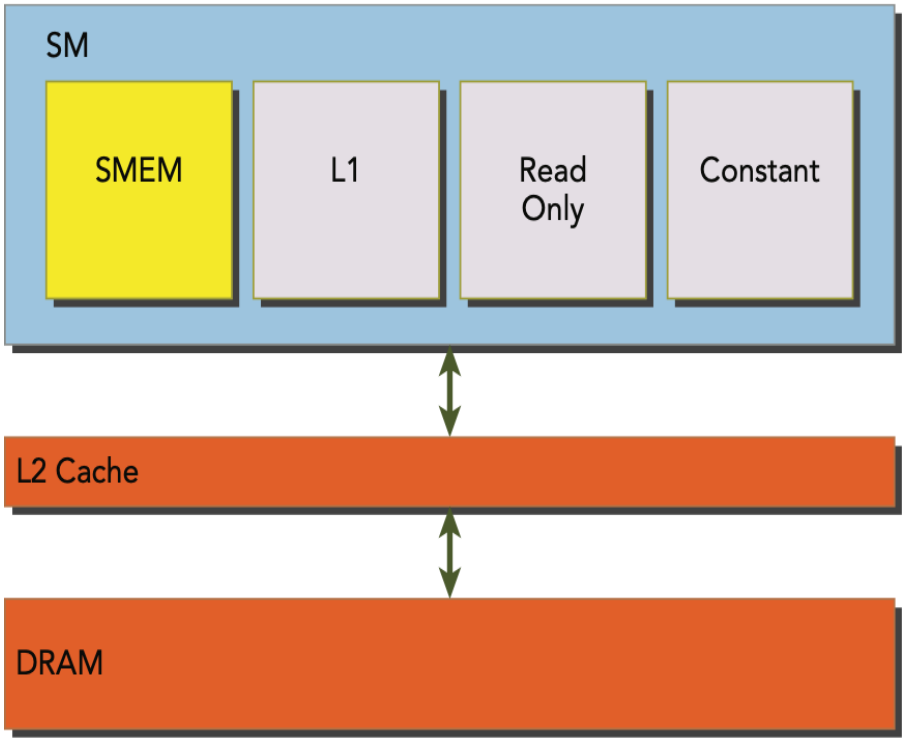
- Because it is on-chip, shared memory is much faster than local and global memory.
- In fact, shared memory latency is roughly 100x lower than uncached global memory latency, provided that there are no bank conflicts between the threads.
- Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory.
- Threads can access data in shared memory loaded from global memory by other threads within the same thread block.
- This capability (combined with thread synchronization) has a number of uses, such as user-managed data caches, high-performance cooperative parallel algorithms (parallel reductions, for example), and to facilitate global memory coalescing in cases where it would otherwise not be possible.

Shared Memory (ShM)



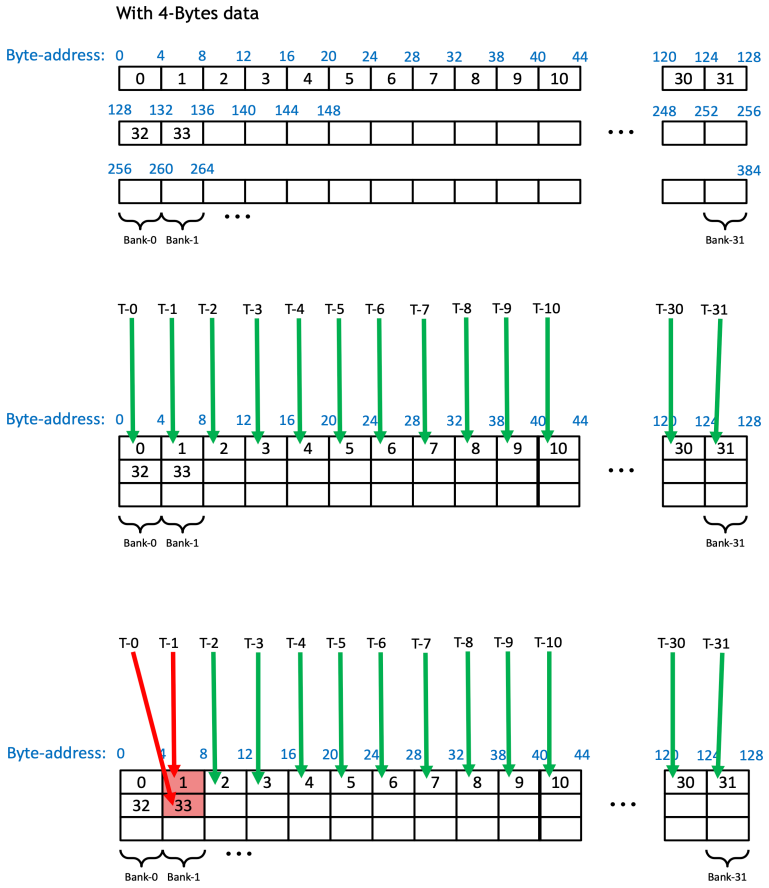
- Shared memory is smaller, low-latency on-chip memory that offers much higher bandwidth than global memory
- On V100, each SM has by default 64 KB of ShM, configurable at compile time for up to 96 KB/SM
- Shared memory latency is roughly 20 to 30 times lower than global memory, and bandwidth is nearly 10 times higher.
- You can think of it as a program-managed cache, partitioned among all SM-resident thread blocks
- It is generally useful as:
  - An intra-block thread communication channel
  - A program-managed cache for global memory data (you have control over eviction)
  - Scratch pad memory for transforming data to improve global memory access patterns

Allocating Shared Memory



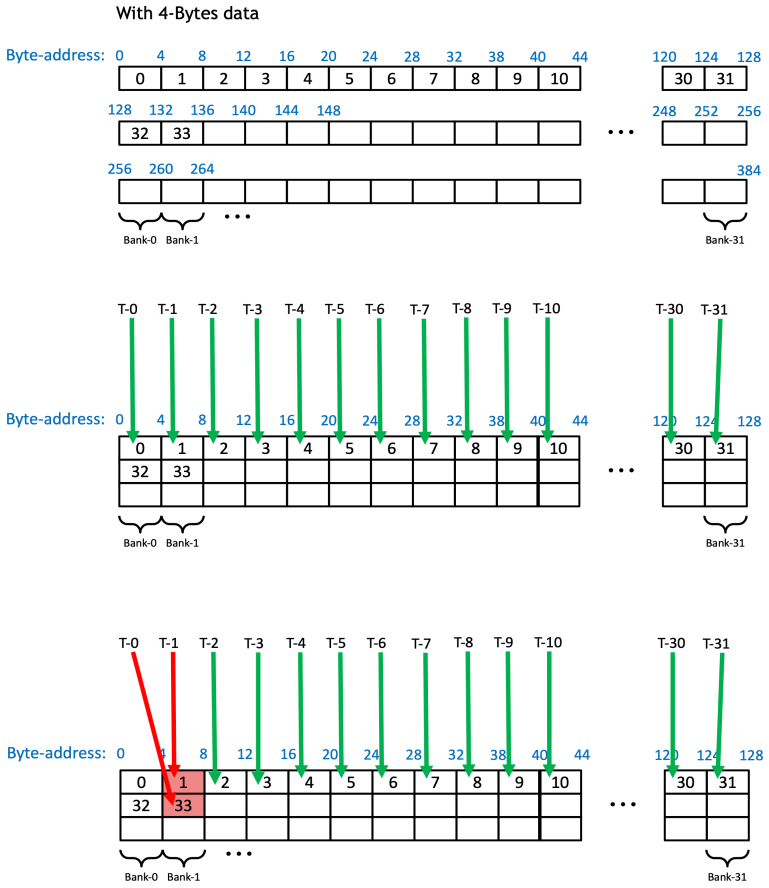
- Shared memory variables are declared with the `shared` qualifier and can be either statically or dynamically allocated
- Static declaration
  - `__Shared__float tile[size y][size x];`
  - Declared inside a kernel function, the variable scope is local to that kernel.
  - Declared outside of any kernels in a file, the variable scope is global to all kernels.
- Dynamic declaration
  - `extern shared float tile[];`
  - The size of this array is unknown at compile-time → need to dynamically allocate shared memory at each kernel invocation
  - `kernel<<<grid, block, isize * sizeof(float)>>>(...)`
  - You can only declare 1D arrays dynamically.

Shared Memory Banks: Access Mode and Bank Conflicts



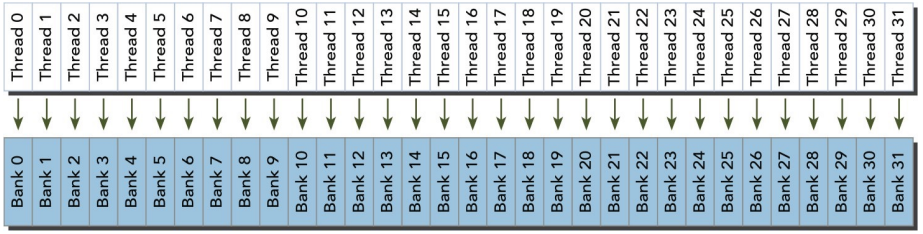
- Shared memory is divided into 32 equally-sized memory modules, called *banks*, which can be accessed simultaneously
- The addresses of shared memory are mapped to the 32 banks. Bank indices:
  - (4B word index) % 32
  - ((1B word index) / 4) % 32
  - 8B word spans two successive banks
- When multiple addresses requested fall into the same bank, a bank conflict occurs, causing the request to be replayed
- **Bank conflict**: 2 or more threads access within different 4B words in the same bank Think: 2 or more threads access different “rows” in the same bank
- The hardware splits a request with a bank conflict into as many separate conflict-free transactions as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory transactions required

Shared Memory Banks: Access Mode and Bank Conflicts

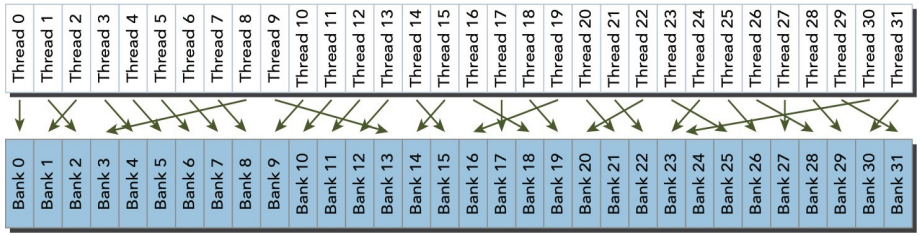


- **Bank conflict**: 2 or more threads access within different 4B words in the same bank Think: 2 or more threads access different "rows" in the same bank
- **N-way bank conflict**: N threads in a warp conflict. Worst case: 32-way conflict → 31 replays
- **No bank conflict** if:
  - Several threads access the same 4-byte word
  - Several threads access different bytes of the same 4-byte word

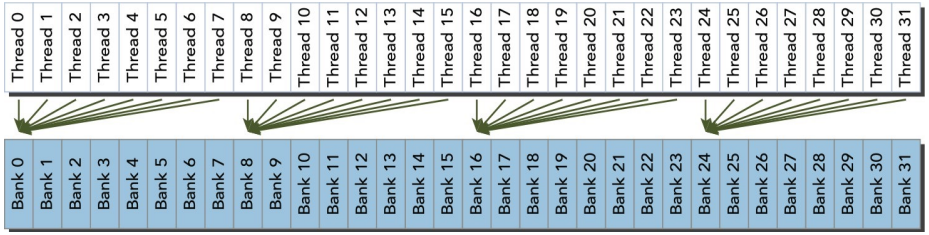
Shared Memory Banks: Access Mode and Bank Conflicts



optimal parallel access pattern



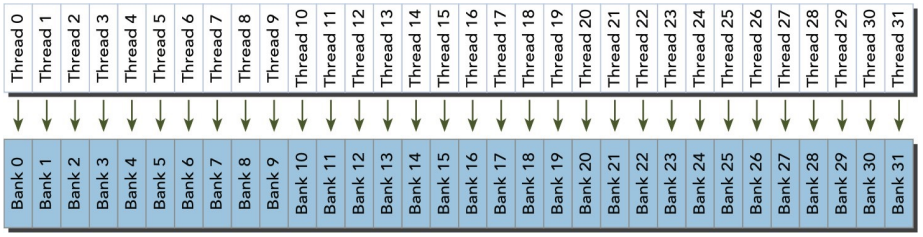
irregular but conflict-free access pattern



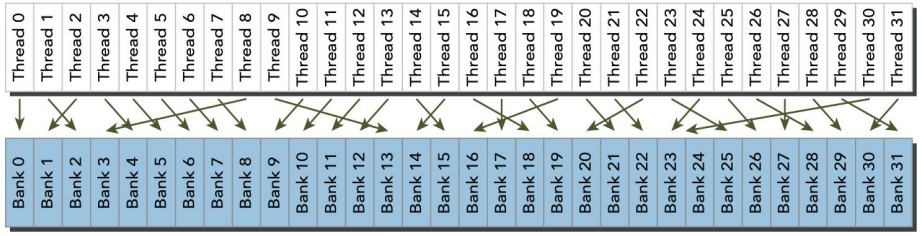
irregular access pattern with potential bank conflicts

- **Bank conflict**: 2 or more threads access within <u>different</u> 4B words in the same bank Think: 2 or more threads access different "rows" in the same bank
- **N-way bank conflict**: N threads in a warp conflict. Worst case: 32-way conflict → 31 replays
- **No bank conflict** if:
  - Several threads access the same 4-byte word
  - Several threads access different bytes of the same 4-byte word

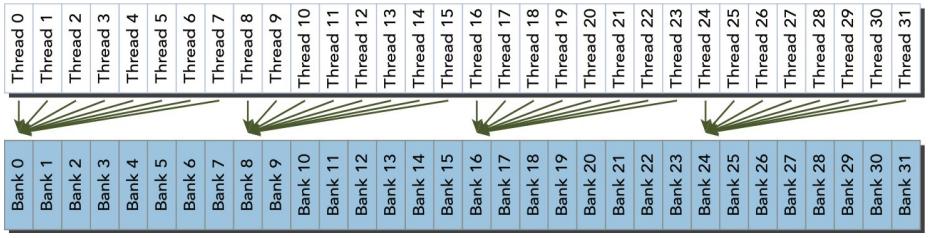
Shared Memory Banks: Access Mode and Bank Conflicts



optimal parallel access pattern



irregular but conflict-free access pattern



irregular access pattern with potential bank conflicts

There are three typical shared memory access modes

Parallel access: multiple addresses accessed by a warp that fall into multiple banks.

- Optimally, a conflict-free shared memory parallel access is performed when every address maps to a separate bank (top 2 cases on LHS). In this case all addresses are serviced in a single memory transaction.

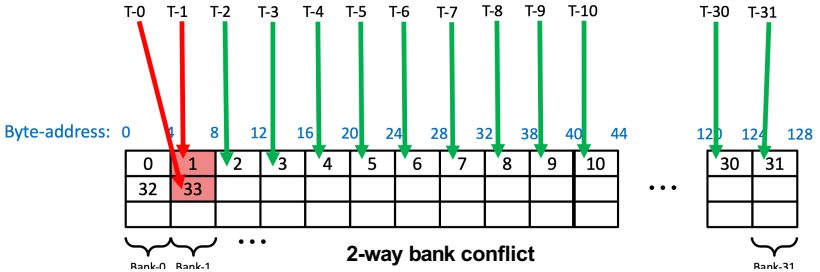
Serial access (worst pattern): When multiple addresses fall into the same bank, the request must be serialized.

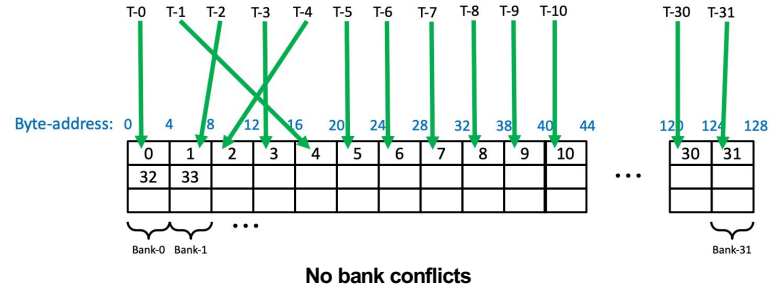
- If all 32 threads in a warp access different memory locations in a single bank, 32 memory transactions will be required!

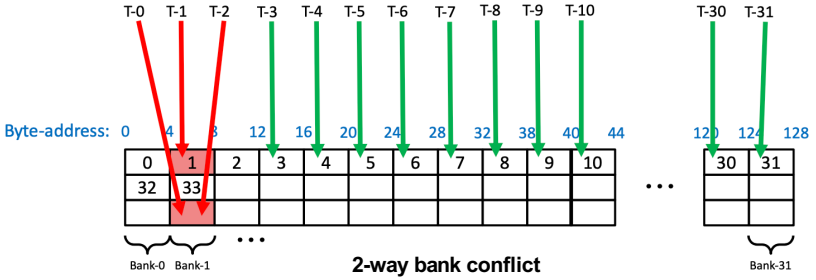
Broadcast access: All threads in a warp read the same address within a single bank.

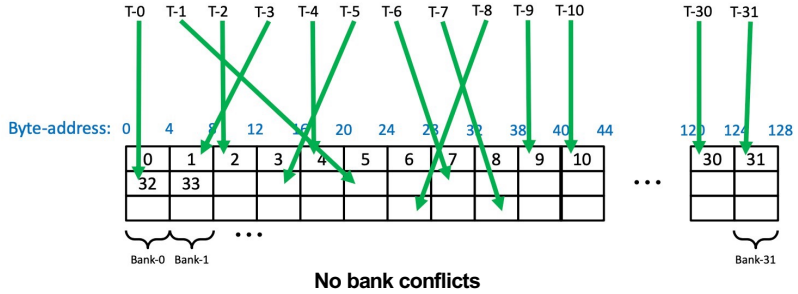
- One memory transaction is executed, and the accessed word is broadcast to all requesting threads. Bandwidth utilization is low because only a small number of bytes are read.

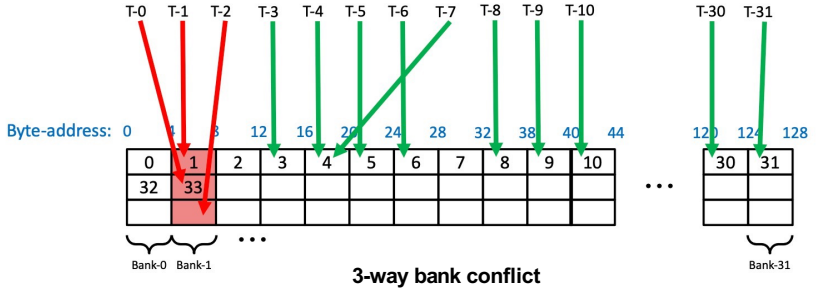
Shared Memory Banks: Access Mode and Bank Conflicts

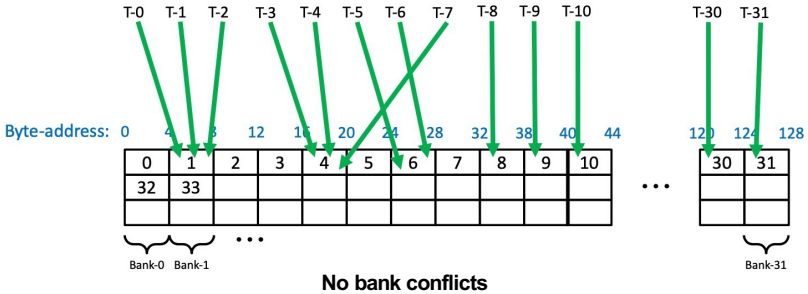




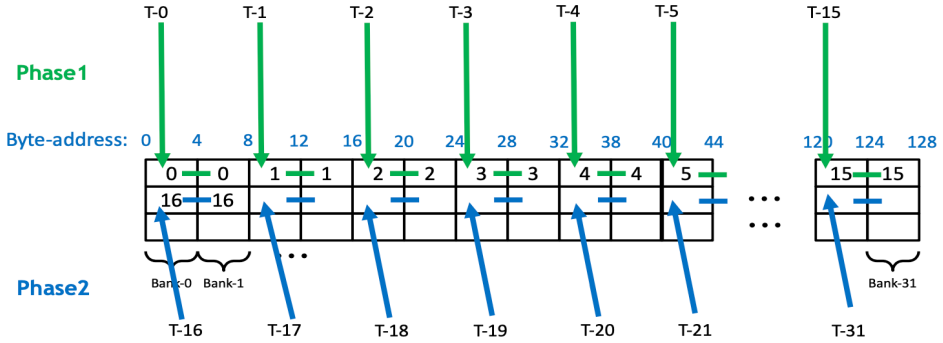




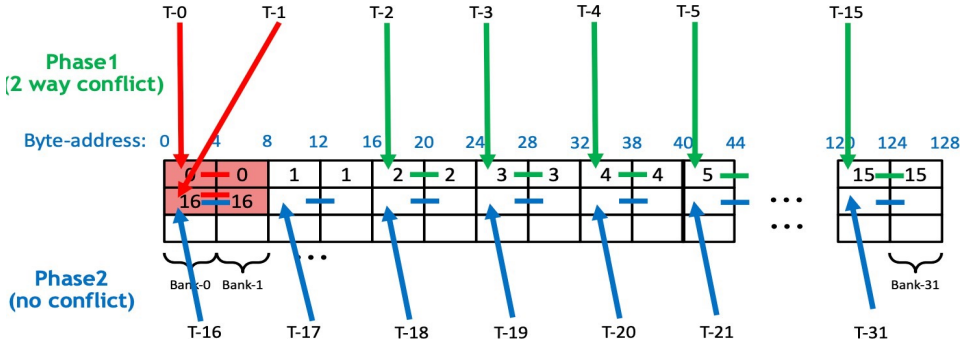




Bank Conflicts: Word Size & Phases



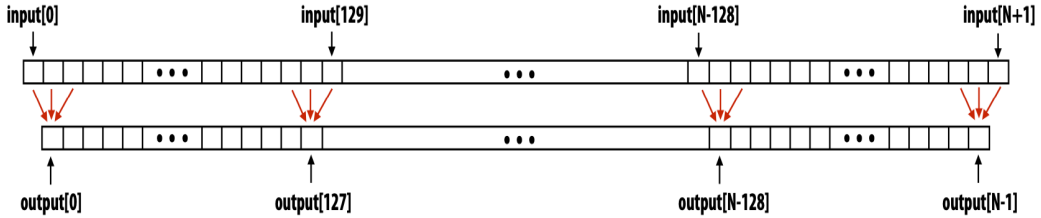
8-byte words, no conflict



8-byte words, 2-way conflict

- **4B or smaller words** → hardware processes addresses of all threads in a warp in a single phase
- **8B words are accessed in 2 phases:**
  - addresses of the first 16 threads in a warp
  - addresses of the second 16 threads in a warp
- Bank conflicts occur only between threads in the **same phase**

Code Example: 1D Convolution with Global Memory




```
// DEVICE CODE
#define THREADS_PER_BLK 128
__global__ void convolve(int N, float* input,
    float* output) {
  int index = blockIdx.x * blockDim.x +
      threadIdx.x;  // thread-local variable
  float result = 0.0f;  // thread-local variable
  for (int i=0; i<3; i++)
   result += input[index + i];
  output[index] = result / 3.f;
}
```

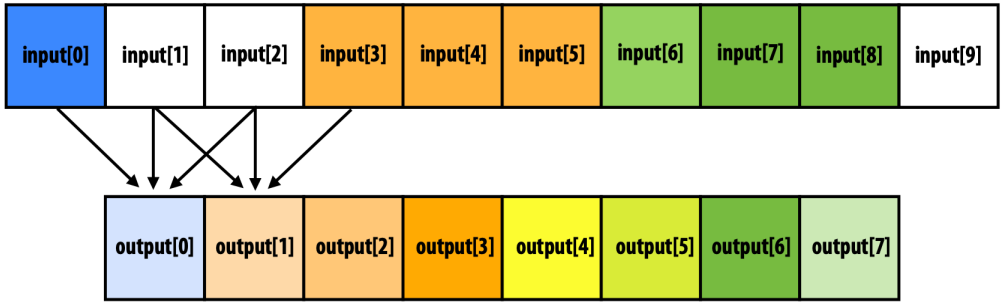
```
// HOST CODE
int N = 1024 * 1024;
// allocate arrays in device memory
cudaMalloc(&devInput, sizeof(float) * (N+2) );
cudaMalloc(&devOutput, sizeof(float) * N);
// Initialize contents of devInput here ...
convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(
    N, devInput, devOutput);
```

- Simple 1D convolution of input array into output array
- Both input and output are global memory arrays
- Each thread computes result for one element out the output array

Code Example: 1D Convolution with Global Memory

```
// DEVICE CODE
#define THREADS_PER_BLK 128
__global__ void convolve (int N, float * input
    , float * output ) {
  // per-block ShM allocation
  __shared__ float support [THREADS_PER_BLK +2];
  int index = blockIdx.x * blockDim.x +
      threadIdx.x;  // thread local variable
  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 variable
  for (int i =0; i <3; i ++)
    result += support [threadIdx.x + i];
  output [index ] = result / 3.f;
```

```
// HOST CODE
int N = 1024 * 1024;
// allocate arrays in device memory        cudaMalloc
(&devInput , sizeof (float) * (N +2) ); cudaMalloc (&
devOutput , sizeof (float) * N);
// Initialize contents of devInput here ...
convolve <<<N/THREADS_PER_BLK , THREADS_PER_BLK >>> (
      N, devInput , devOutput );
```

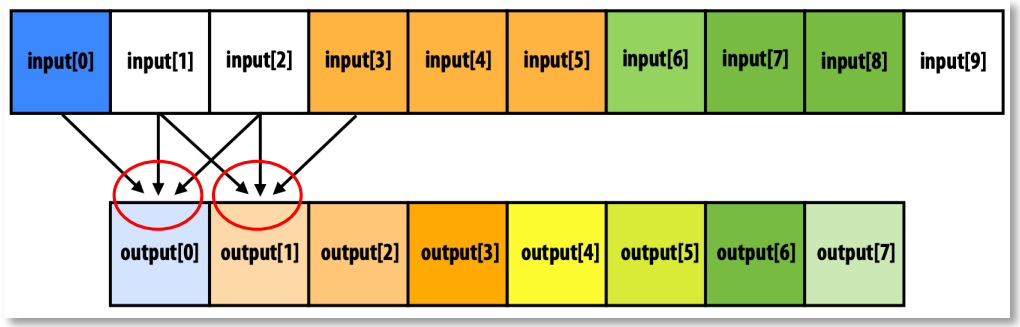


- Both `input` and `output` are global memory arrays
- All threads cooperatively load a block region from global memory into the `support` shared memory array
- All threads in the block synchronize (`__syncthreads()`) to ensure all threads have access to updated `support` values
- Each thread computes result for one element using `support` values and writes results to `output`
- What is the advantage compared to previous version?

Code Example: 1D Convolution with Global Memory

```
// DEVICE CODE
#define THREADS_PER_BLK 128
__global__ void convolve (int N, float * input
    , float * output ) {
  // per-block ShM allocation
  __shared__ float support [THREADS_PER_BLK +2];
  int index = blockIdx.x * blockDim.x +
      threadIdx.x;  // thread local variable
  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 variable
  for (int i =0; i <3; i ++)
    result += support [threadIdx.x + i];
  output [index ] = result / 3.f;
```

```
// HOST CODE
int N = 1024 * 1024;
// allocate arrays in device memory        cudaMalloc
(&devInput , sizeof (float) * (N +2) ); cudaMalloc (&
devOutput , sizeof (float) * N);
// Initialize contents of devInput here ...
convolve <<<N/THREADS_PER_BLK , THREADS_PER_BLK >>> (
      N, devInput , devOutput );
```



- Both `input` and `output` are global memory arrays
- All threads cooperatively load a block region from global memory into the `support` shared memory array
- All threads in the block synchronize (`__syncthreads()`) to ensure all threads have access to updated `support` values
- Each thread computes result for one element using `support` values and writes results to `output`
- Total of 130 load instructions from global mem instead of 3×128 load instructions