

Fundamental CUDA Optimisations

Anthony Morse

Outline

- **Fermi Architecture**
- **Kernel optimisations**
 - Launch configuration
 - Global memory throughput
 - Shared memory access
 - Instruction throughput / control flow
- **Optimisation of CPU-GPU interaction**
 - Maximizing PCIe throughput
 - Overlapping kernel execution with memory copies

20-Series Architecture

- 512 **Scalar Processor (SP) cores** execute parallel thread instructions

- 16 **Streaming Multiprocessors (SMs)** each contains

- 32 scalar processors

- 32 fp32 / int32 ops / clock,

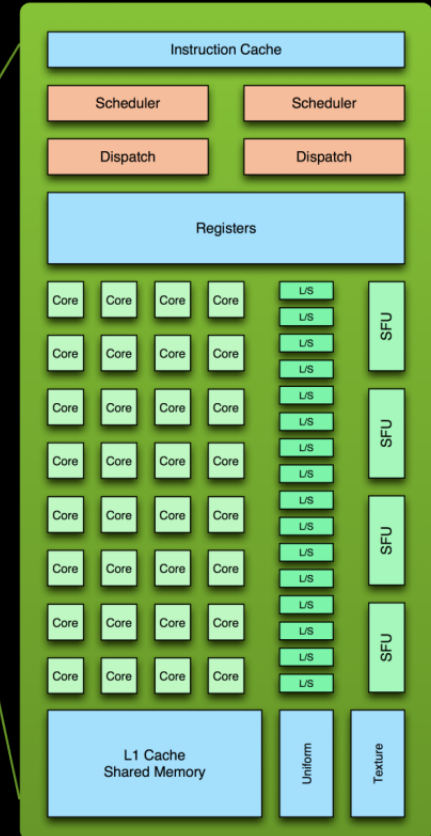
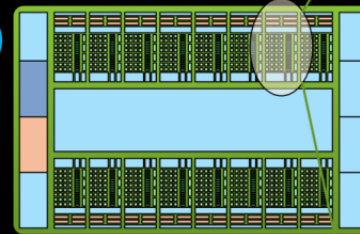
- 16 fp64 ops / clock

- 4 Special Function Units (SFUs)

- Shared register file (128KB)

- 48 KB / 16 KB Shared memory

- 16KB / 48 KB L1 data cache



Execution Model

Software



Thread



Thread
Block

Hardware



Scalar
Processor



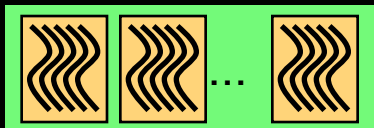
Multiprocessor

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



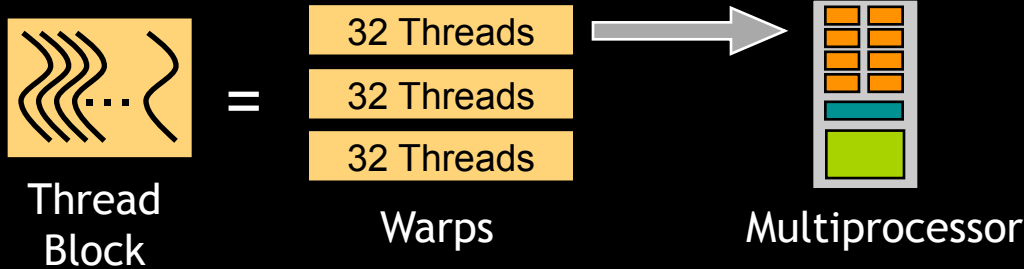
Grid



Device

A kernel is launched as a grid of thread blocks

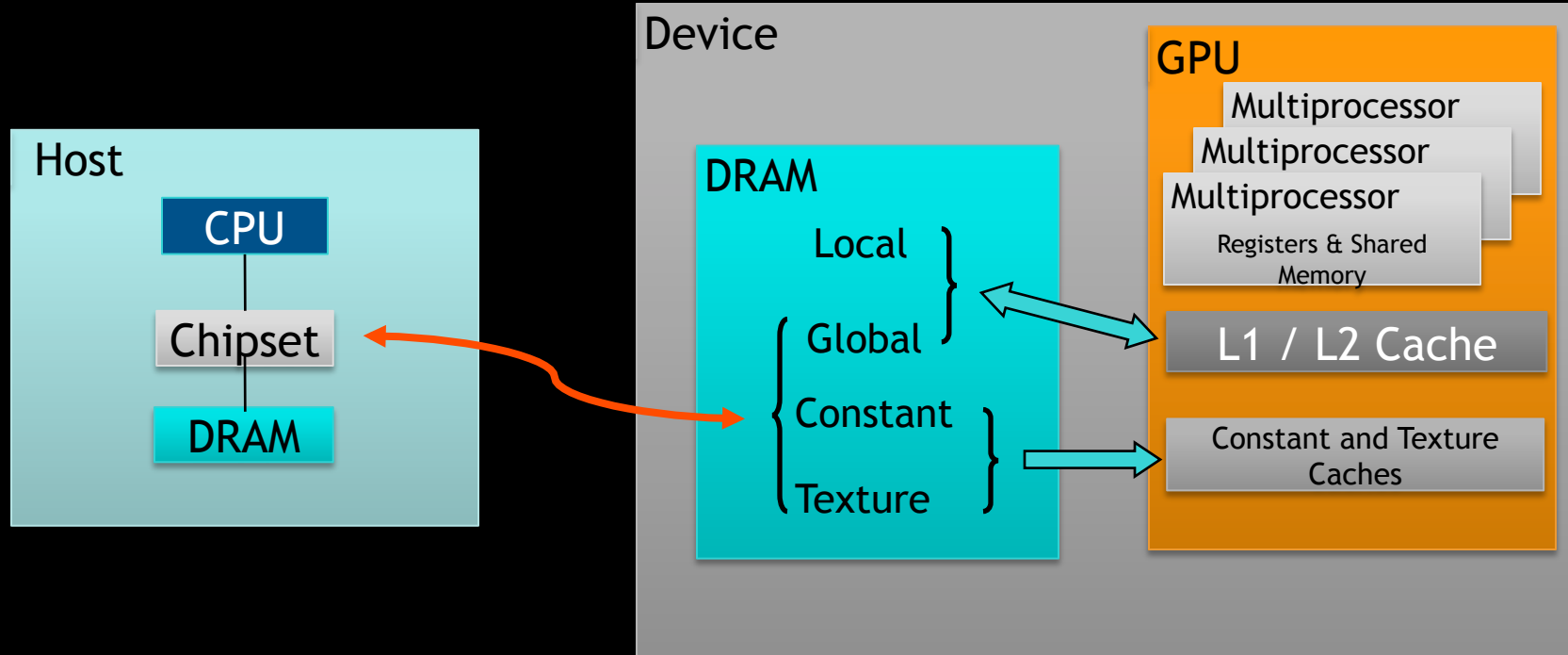
Warps



A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor

Memory Architecture: Tesla T-20 Series



Launch Configuration

Launch Configuration

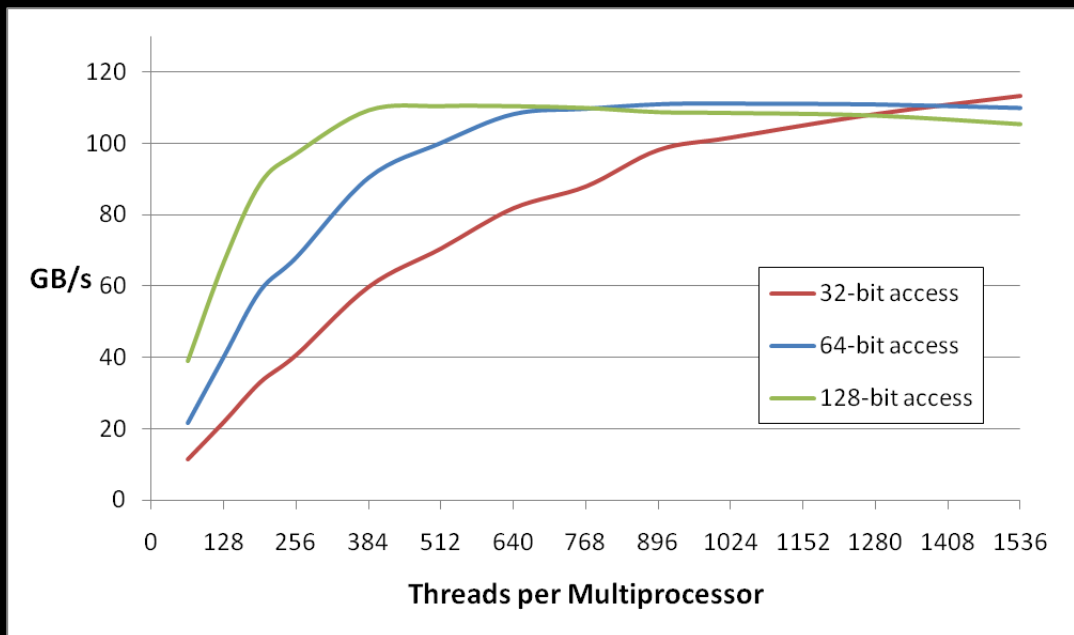
- **Key to understanding:**
 - Instructions are issued in order
 - A thread stalls when one of the operands isn't ready:
 - Memory read by itself doesn't stall execution
 - Latency is hidden by switching threads
 - GMEM latency: 400-800 cycles
 - Arithmetic latency: 18-22 cycles
- How many threads/threadblocks to launch?
- Conclusion:
 - Need enough threads to hide latency

Launch Configuration

- **Hiding arithmetic latency**
 - Need **~18 warps (576)** threads per SM
 - Or, latency can also be hidden with independent instructions from the same warp
 - For example, if instruction never depends on the output of preceding instruction, then only **9** warps are needed, etc.
- **Maximizing global memory throughput**
 - Depends on the access pattern, and word size
 - Need enough memory transactions in flight to saturate the bus
 - Independent loads and stores from the same thread
 - Loads and stores from different threads
 - Larger word sizes can also help (float2 is twice the transactions of float, for example)

Maximizing Memory Throughput

- **Increment of an array of 64M elements**
 - Two accesses per thread (load then store)
 - The two accesses are dependent, so really 1 access per thread at a time
- **Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s**



Several independent smaller accesses have the same effect as one larger one.

For example:

Four 32-bit $\approx$ one 128-bit

Launch Configuration: Summary

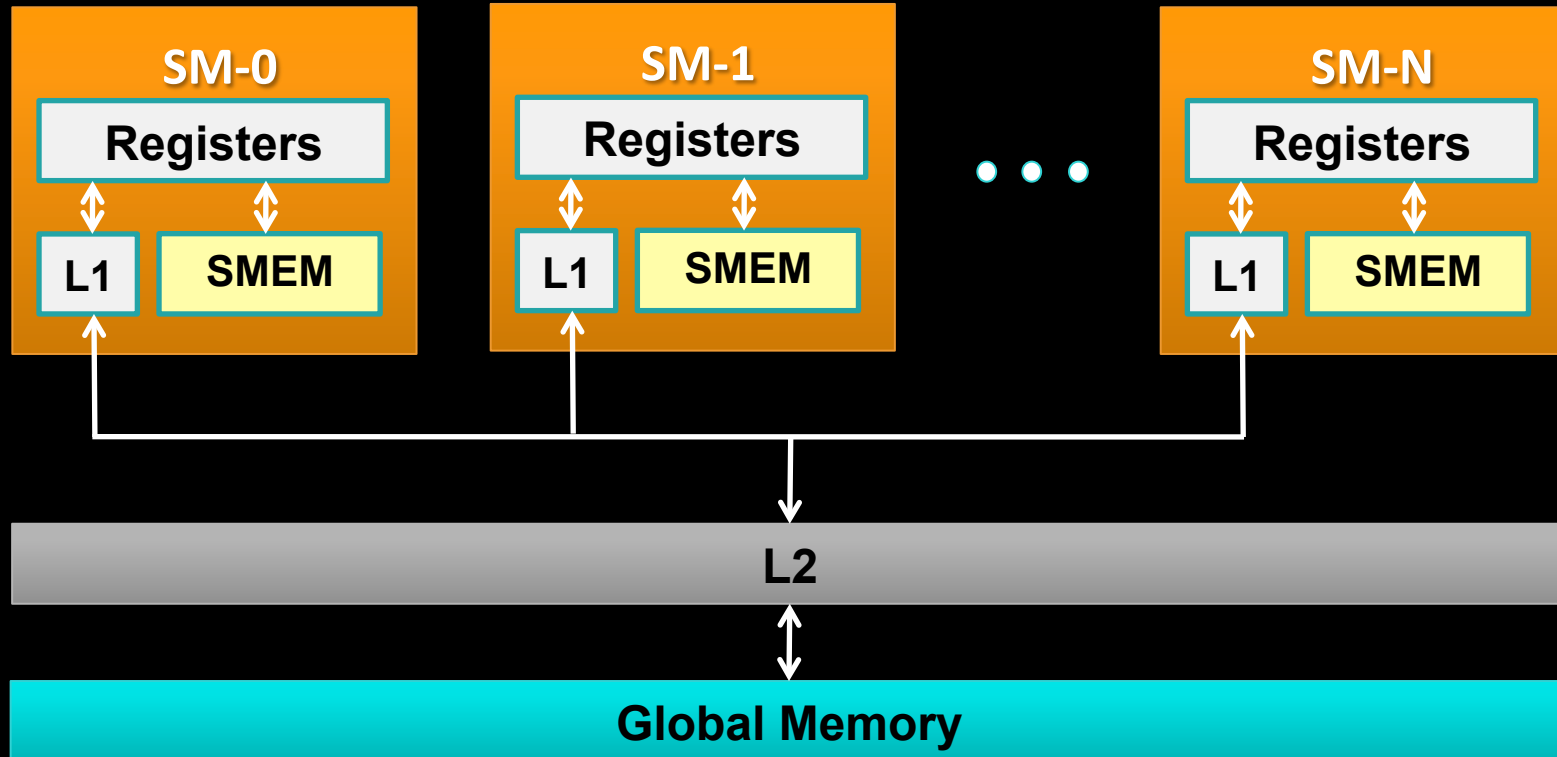
- Need enough total threads to keep GPU busy
 - Typically, you'd like **512+ threads** per SM
 - More if processing **just one** fp32 element per thread
 - Of course, exceptions exist
- Threadblock configuration
 - Threads per block should be a **multiple of warp size (32)**
 - SM can concurrently execute **up to 8** thread blocks
 - Really small thread blocks prevent achieving good occupancy
 - Really large thread blocks are less flexible
 - I generally use **128-256 threads/block**, but use whatever is best for the application
- For more details:
 - Vasily Volkov's GTC2010 talk "Better Performance at Lower Occupancy" (<http://www.gputechconf.com/page/gtc-on-demand.html#session2238>)

Global Memory Throughput

Memory Hierarchy Review

- **Local storage**
 - Each thread has own local storage
 - Mostly registers (managed by the compiler)
- **Shared memory / L1**
 - Program configurable: **16KB shared / 48 KB L1** OR **48KB shared / 16KB L1**
 - Shared memory is accessible by the threads in the same threadblock
 - Very low latency
 - Very high throughput: **1+ TB/s** aggregate
- **L2**
 - All accesses to global memory go through L2, including copies to/from CPU host
- **Global memory**
 - Accessible by all threads as well as host (CPU)
 - High latency (**400-800 cycles**)
 - Throughput: up to **177 GB/s**

Fermi Memory Hierarchy Review



GMEM Operations

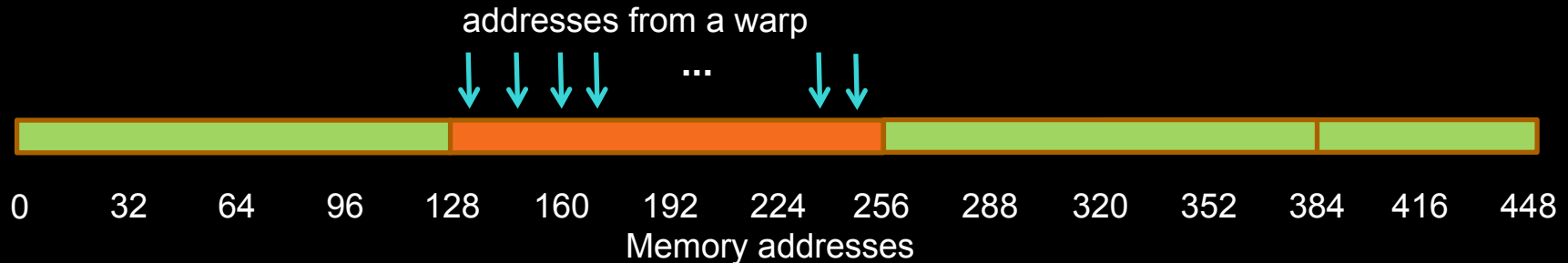
- Two types of loads:
 - Caching
 - Default mode
 - Attempts to hit in L1, then L2, then GMEM
 - Load granularity is 128-byte line
 - Non-caching
 - Compile with `-Xptxas -dlcm=cg` option to nvcc
 - Attempts to hit in L2, then GMEM
 - Do not hit in L1, invalidate the line if it's in L1 already
 - Load granularity is 32-bytes
- Stores:
 - Invalidate L1, write-back for L2

Load Operation

- Memory operations are issued **per warp (32 threads)**
 - Just like all other instructions
- Operation:
 - Threads in a warp provide memory addresses
 - Determine which lines/segments are needed
 - Request the needed lines/segments

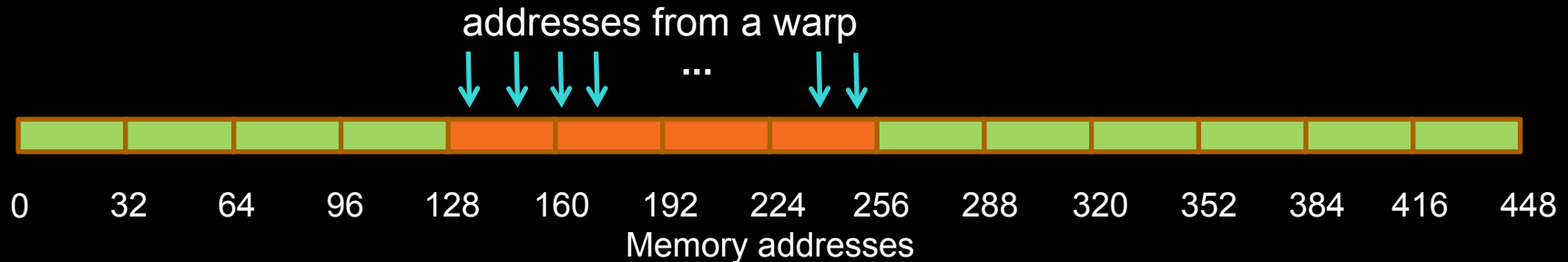
Caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
 - Warp needs 128 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: 100%



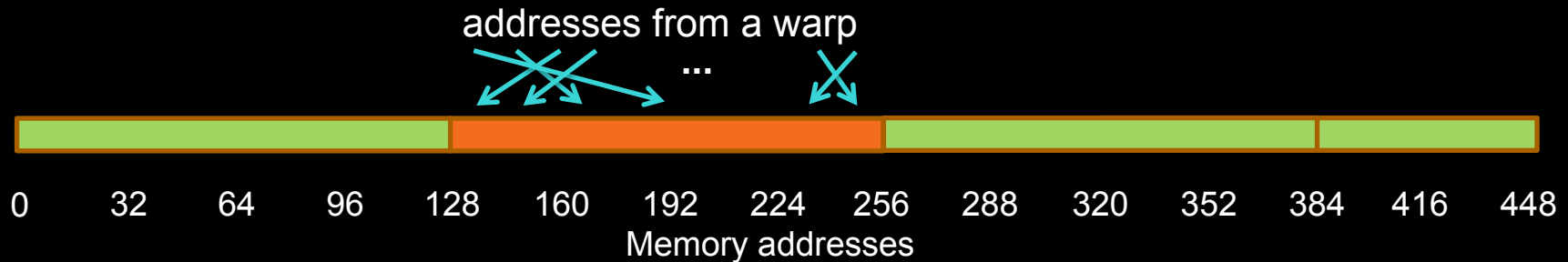
Non-caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
 - Warp needs 128 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: **100%**



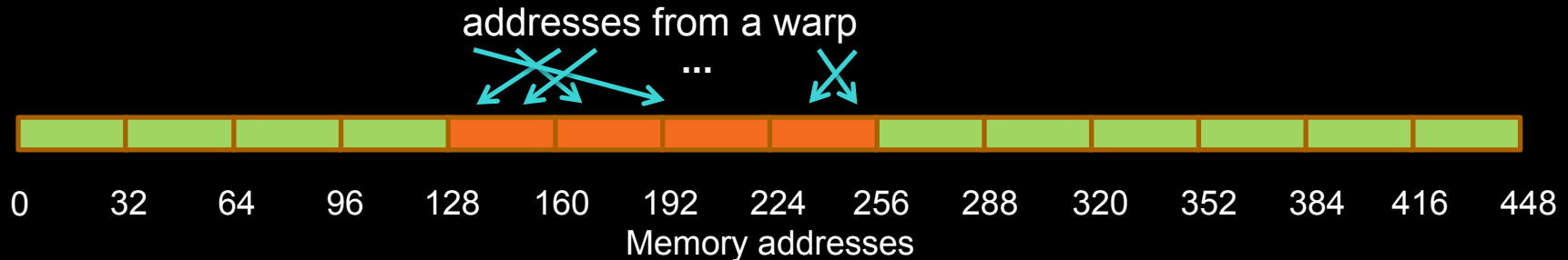
Caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
 - Warp needs 128 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: 100%



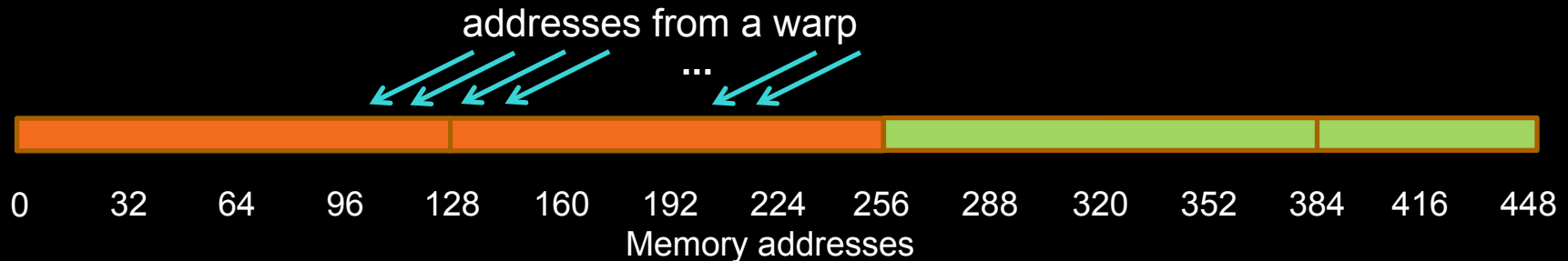
Non-caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
 - Warp needs 128 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: **100%**



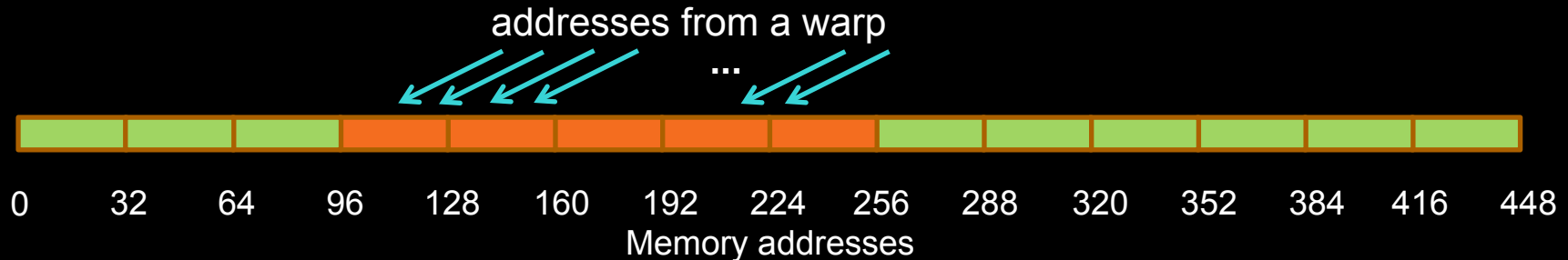
Caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
 - Warp needs 128 bytes
 - 256 bytes move across the bus on misses
 - Bus utilization: 50%



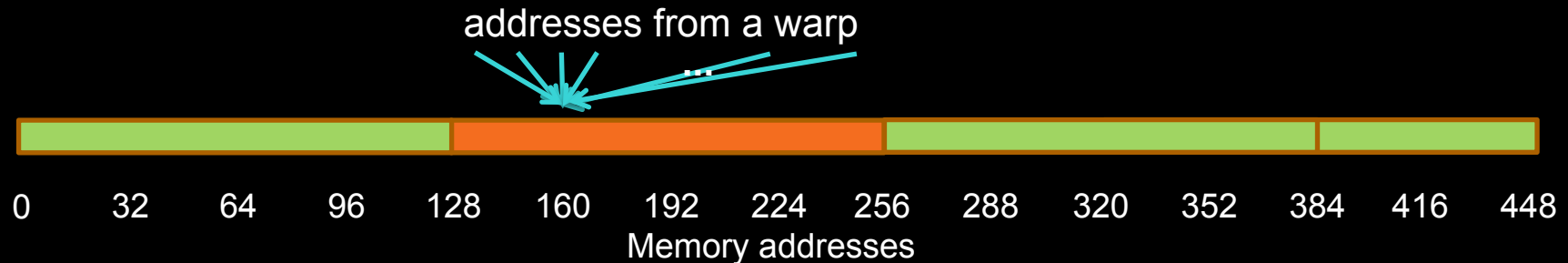
Non-caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
 - Warp needs 128 bytes
 - 160 bytes move across the bus on misses
 - Bus utilization: **at least 80%**
 - Some misaligned patterns will fall within 4 segments, so **100%** utilization



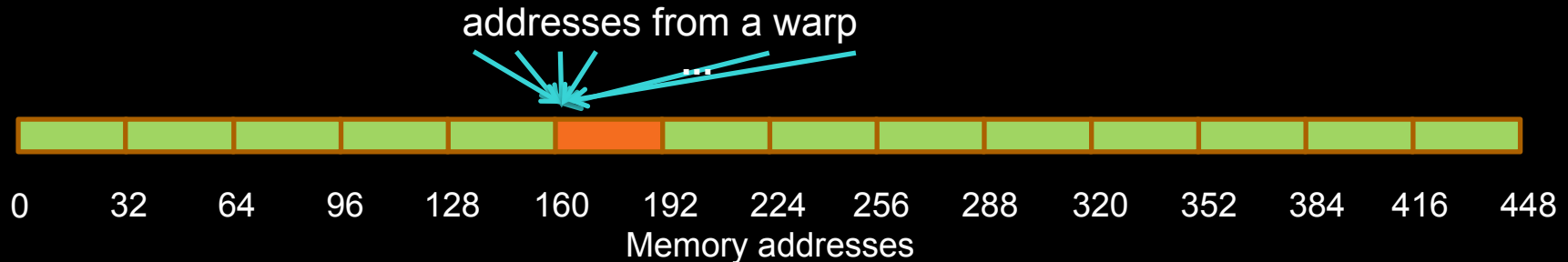
Caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
 - Warp needs 4 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: 3.125%



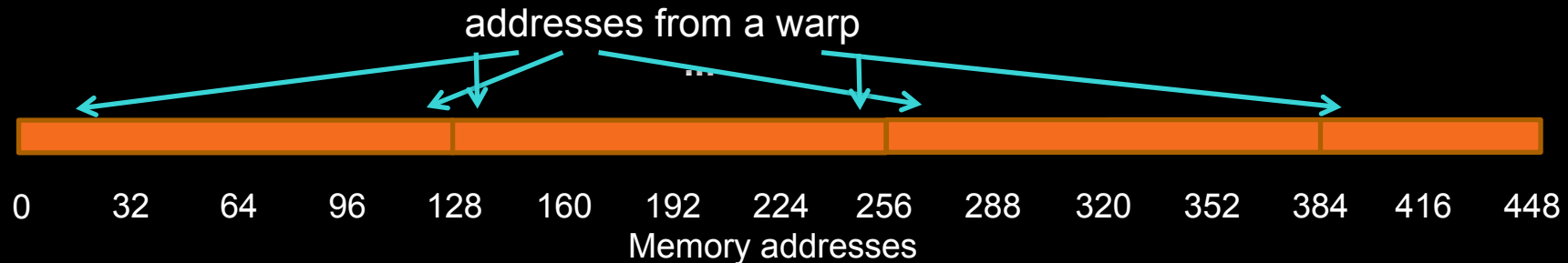
Non-caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
 - Warp needs 4 bytes
 - 32 bytes move across the bus on a miss
 - Bus utilization: 12.5%



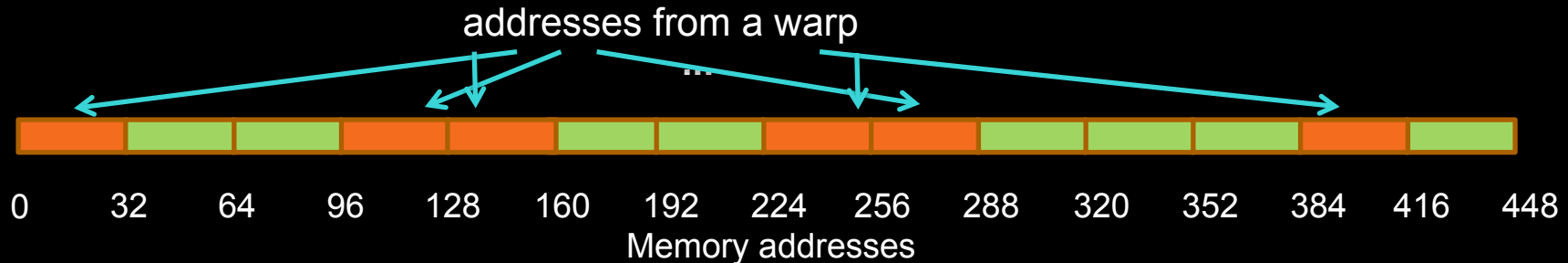
Caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within N cache-lines
 - Warp needs 128 bytes
 - $N \times 128$ bytes move across the bus on a miss
 - Bus utilization: $128 / (N \times 128)$ ← memory required / memory fetched



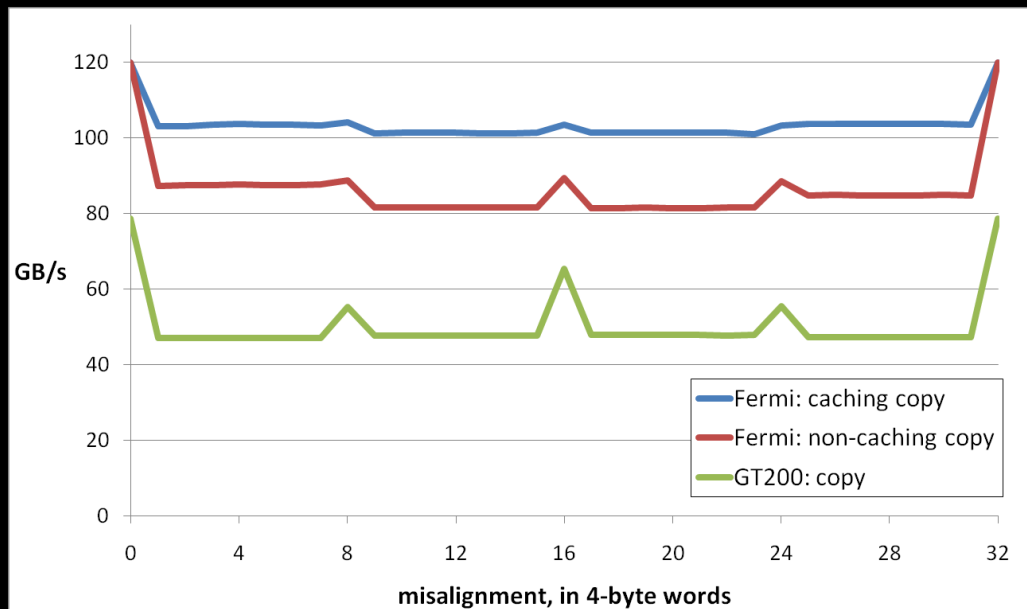
Non-caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within N segments
 - Warp needs 128 bytes
 - $N \times 32$ bytes move across the bus on a miss
 - Bus utilization: $128 / (N \times 32) \leftarrow \text{memory required} / \text{memory fetched}$



Impact of Address Alignment

- Warps should access aligned regions for maximum memory throughput
 - L1 can help for misaligned loads if several warps are accessing a contiguous region
 - ECC further significantly reduces misaligned store throughput



GMEM Optimisation Guidelines

- **Strive for perfect coalescing**
 - Align starting address (may require padding)
 - A warp should access within a contiguous region
- **Have enough concurrent accesses to saturate the bus**
 - Process several elements per thread
 - Multiple loads get pipelined
 - Indexing calculations can often be reused
 - Launch enough threads to maximize throughput
 - Latency is hidden by switching threads (warps)
- **Try L1 and caching configurations to see which one works best**
 - Caching vs non-caching loads (compiler option)
 - 16KB vs 48KB L1 (CUDA call)

Shared Memory

Shared Memory

- **Uses:**

- Inter-thread communication within a block
- Cache data to reduce redundant global memory accesses
- Use it to improve global memory access patterns

- **Organisation:**

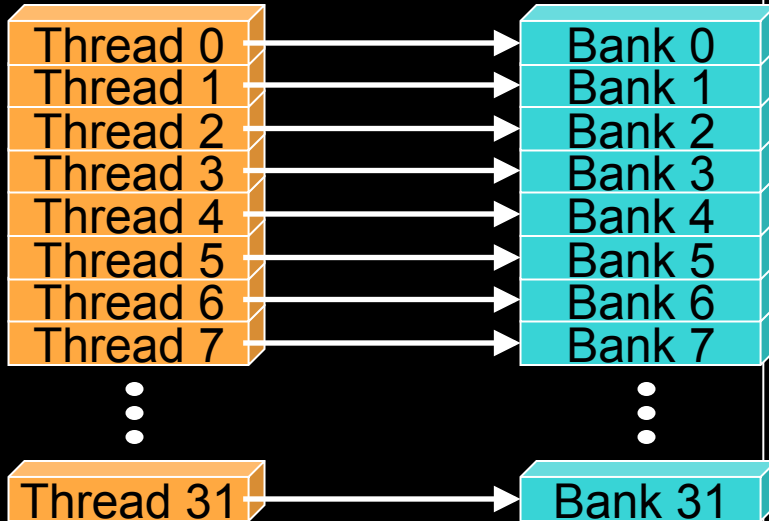
- 32 banks, 4-byte wide banks
- Successive 4-byte words belong to different banks

- **Performance:**

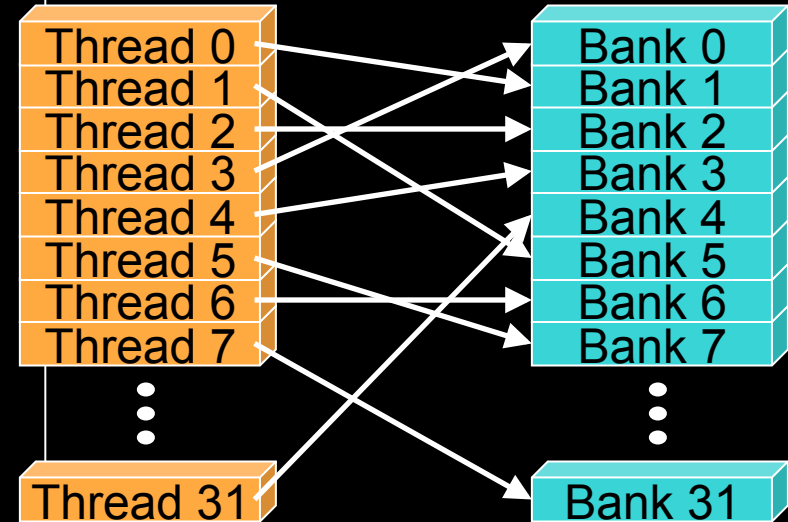
- 4 bytes per bank per 2 clocks per multiprocessor
- smem accesses are issued per 32 threads (warp)
- **serialisation:** if N threads of 32 access different 4-byte words in the same bank, N accesses are executed serially
- **multicast:** N threads access the same word in one fetch
 - Could be different bytes within the same word

Bank Addressing Examples

• No Bank Conflicts

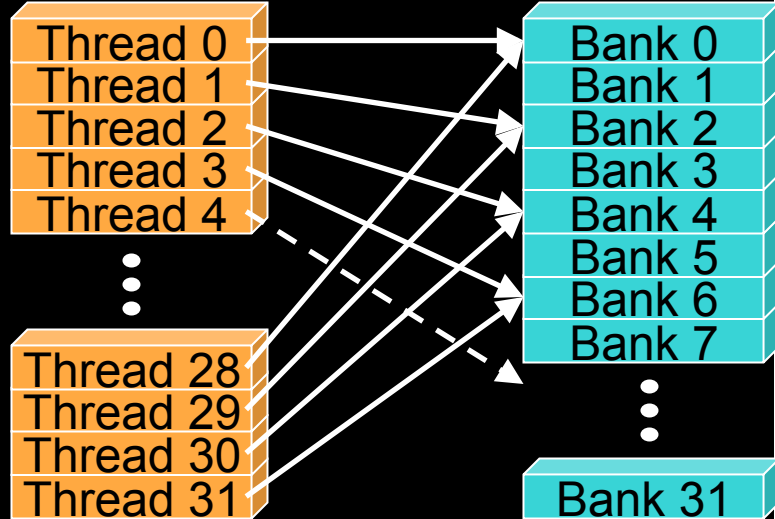


• No Bank Conflicts

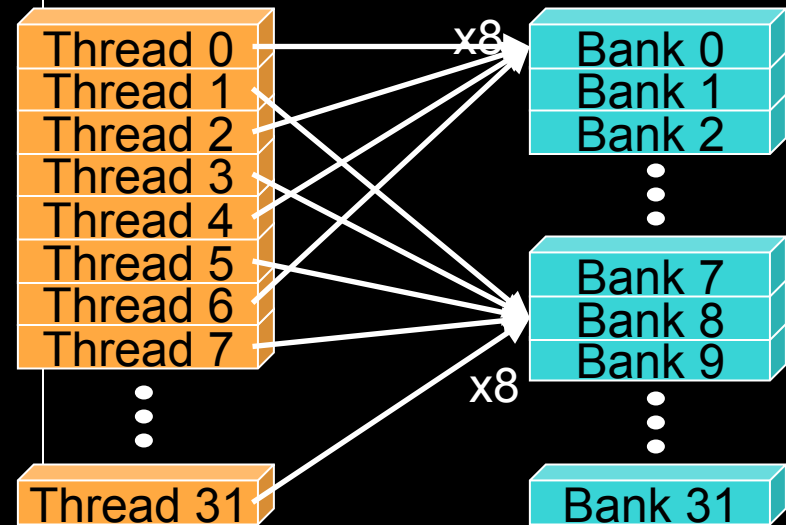


Bank Addressing Examples

2-way Bank Conflicts

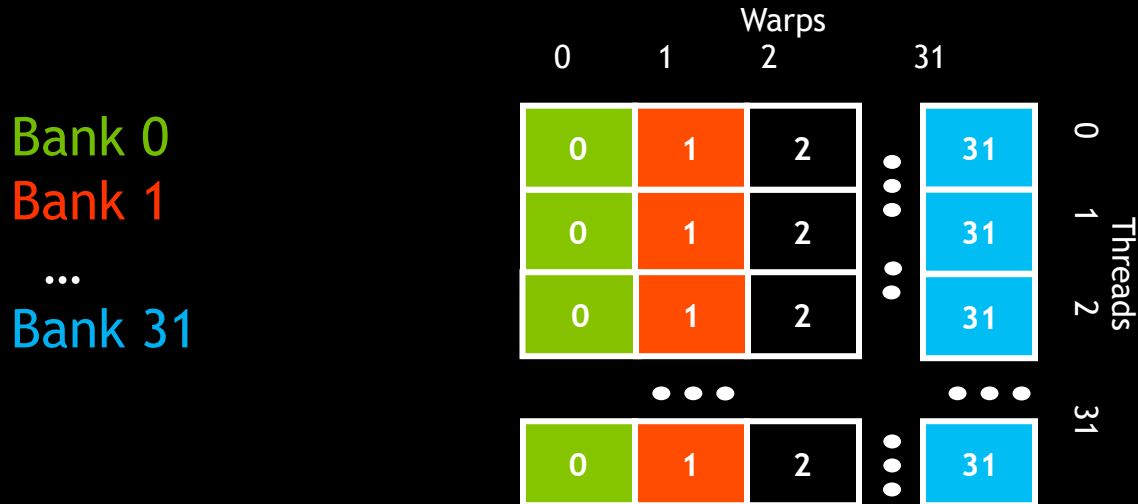


8-way Bank Conflicts



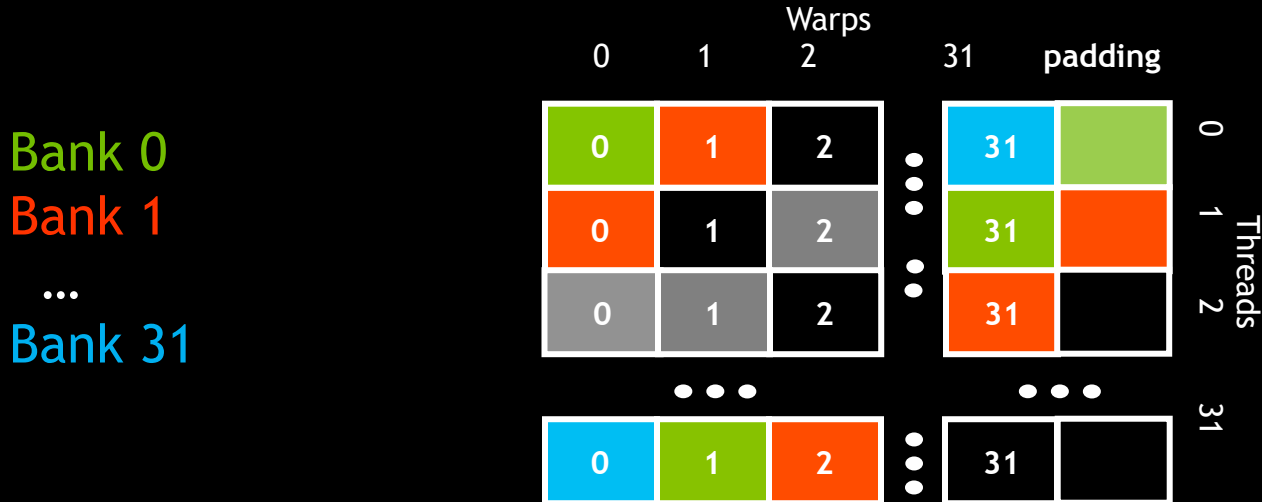
Shared Memory: Avoiding Bank Conflicts

- **32x32** SMEM array
- Warp accesses a column:
 - Different threads access different words within the same bank
 - 32-way bank conflicts (threads in a warp access the same bank)



Shared Memory: Avoiding Bank Conflicts

- Add a column for padding:
 - 32x33 SMEM array
- Warp accesses a column:
 - Different threads access different words within the same bank
 - 32 different banks, no bank conflicts



Texture and Constant Memory

- **Read-only**
- **Data resides in global memory**
- **Read via special-purpose caches**

Texture

- **Separate cache**
- **Dedicated texture cache hardware provides:**
 - **Out-of-bounds index handling**
 - clamp or wrap-around
 - **Optional interpolation**
 - Think: using fp indices for arrays
 - Linear, bilinear, trilinear
 - Interpolation weights are 9-bit
 - **Optional format conversion**
 - {char, short, int} -> float
 - **All of these are “free”**

Instruction Throughput & Control Flow

Runtime Math Library and Ininsics

- Two types of runtime math library functions
 - `__func()`: many map directly to hardware ISA
 - Fast but lower accuracy
 - Examples: `__sinf(x)`, `__expf(x)`, `__powf(x, y)`
 - `func()`: compile to multiple instructions
 - Slower but higher accuracy (5 ulp or less)
 - Examples: `sin(x)`, `exp(x)`, `pow(x, y)`
- A number of additional intrinsics:
 - `__sincosf()`, `__frcp_rz()`, ...
 - Explicit IEEE rounding modes (rz,rn,ru,rd)

Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
 - Threads within a single warp take different paths
 - `if-else, ...`
 - Different execution paths within a warp are serialised
- Different warps can execute different code with no impact on performance
- Avoid diverging within a warp
 - Example with divergence:
 - `if (threadIdx.x > 2) {...} else {...}`
 - Branch granularity < warp size
 - Example without divergence:
 - `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
 - Branch granularity is a whole multiple of warp size

CPU-GPU Interaction

Pinned (non-pageable) memory

- **Pinned memory enables:**
 - faster PCIe copies
 - memcpy asynchronous with CPU
 - memcpy asynchronous with GPU
- **Usage**
 - `cudaHostAlloc` / `cudaFreeHost`
 - instead of `malloc` / `free`
 - `cudaHostRegister` / `cudaHostUnregister`
 - pin regular memory after allocation
- **Implication:**
 - pinned memory is essentially removed from host virtual memory

Streams and Async API

- **Default API:**
 - Kernel launches are asynchronous with CPU
 - Memcopies (D2H, H2D) block CPU thread
 - CUDA calls are serialised by the driver
- **Streams and async functions provide:**
 - Memcopies (D2H, H2D) asynchronous with CPU
 - Ability to concurrently execute a kernel and a memcopy
- **Stream = sequence of operations that execute in issue-order on GPU**
 - Operations from different streams may be interleaved
 - A kernel and memcopy from different streams can be overlapped

Overlap kernel and memory copy

- **Requirements:**

- D2H or H2D memcpy from pinned memory
- Kernel and memcpy in different, non-0 streams

- **Code:**

```
cudaStream_t  stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
  
cudaMemcpyAsync( dst, src, size, dir, stream1 );  
kernel<<<grid, block, 0, stream2>>>(...);
```

} potentially
overlapped

Call Sequencing for Optimal Overlap

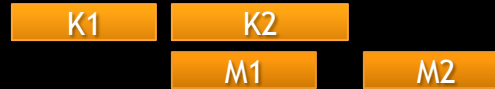
- CUDA calls are dispatched to the hw in the sequence they were issued
- Fermi can concurrently execute:
 - Up to 16 kernels (16SMs)
 - Up to 2 memcpys, as long as they are in different directions (D2H and H2D)
- A call is dispatched if both are true:
 - Resources are available
 - Preceding calls in the same stream have completed
- Scheduling:
 - Kernels are executed in the order in which they were issued
 - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available
- Note that if a call blocks, it blocks all other calls of the same type behind it, even in other streams
 - Type is one of { kernel, memcpy }

Stream Examples (current HW)

K1,M1,K2,M2:



K1,K2,M1,M2:



K1,M1,M2:



K1,M2,M1:



K1,M2,M2:



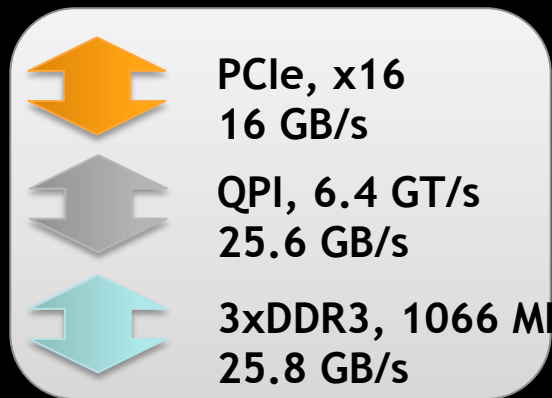
K: Kernel
M: Memcopy
Integer: Stream ID

Time

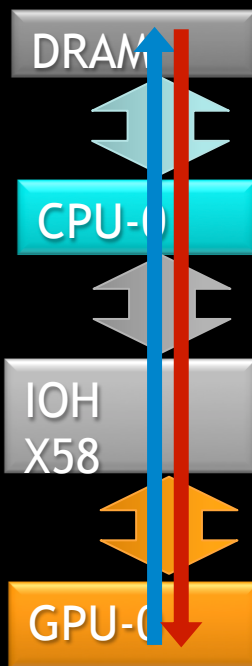
More on Dual Copy

- **Fermi is capable of duplex communication with the host**
 - PCIe bus is duplex
 - The two memcopies must be in different streams, different directions
- **Not all current host systems can saturate duplex PCIe bandwidth:**
 - Likely issues with IOH chips
 - If this is important to you, test your host system

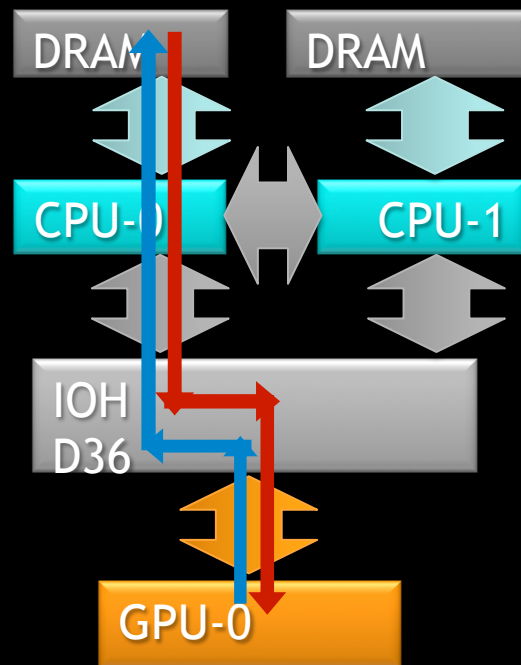
Duplex Copy: Experimental Results



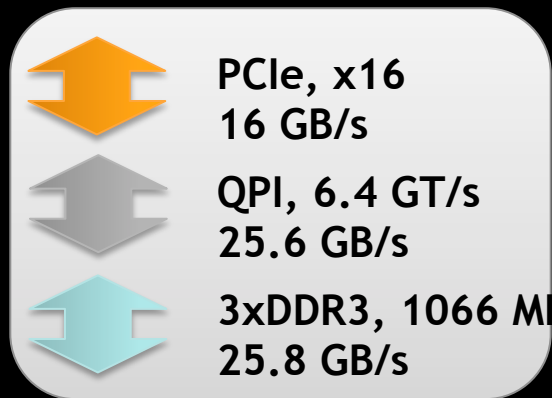
10.8 GB/s



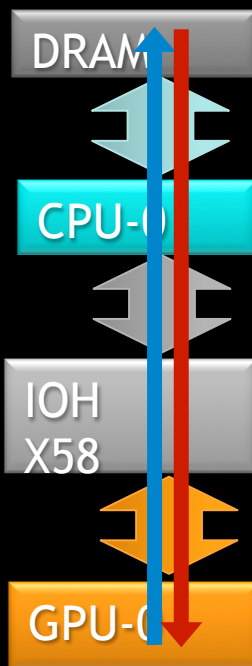
7.5 GB/s



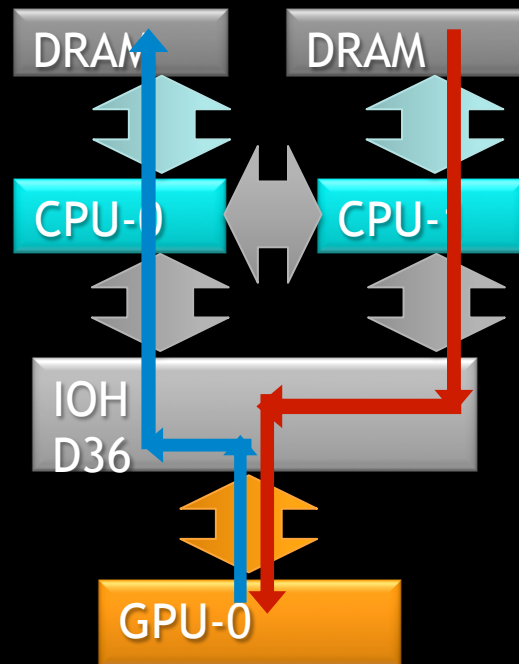
Duplex Copy: Experimental Results



10.8 GB/s

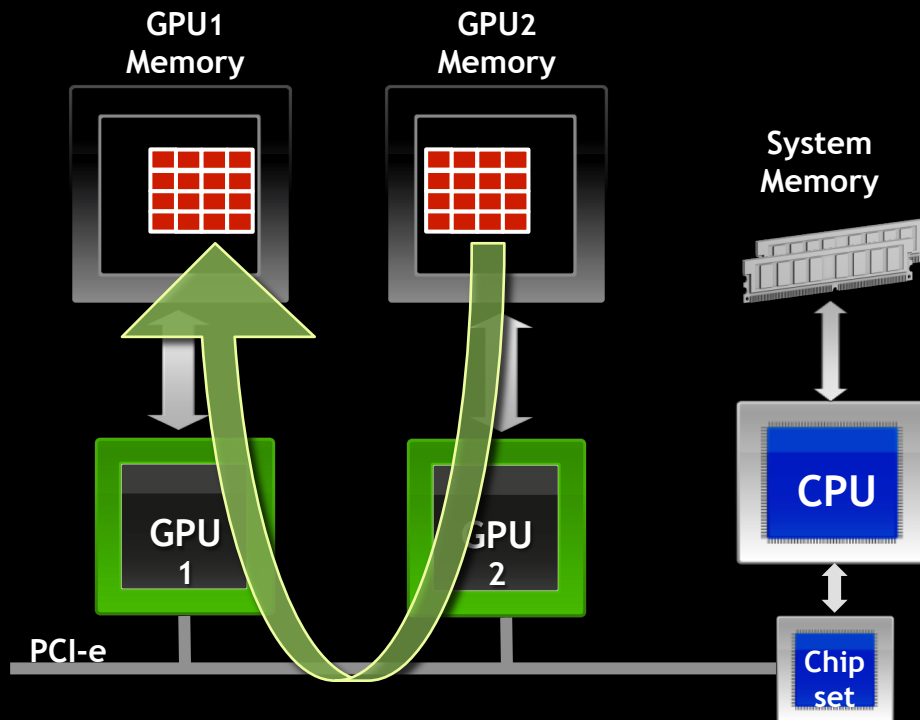


11 GB/s



GPUDirect Peer-to-Peer Communication

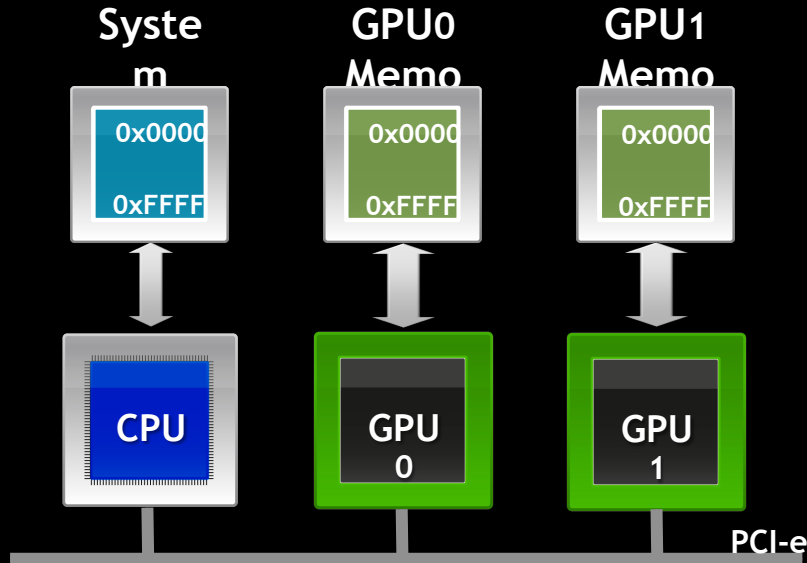
Direct Transfers b/w GPUs



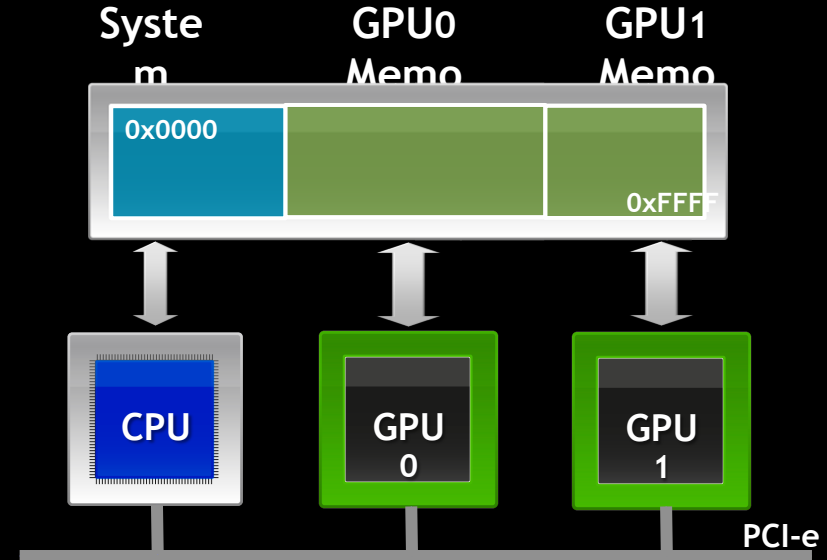
Unified Virtual Addressing

Easier to Program with Single Address Space

No UVA: Multiple Memory Spaces



UVA : Single Address Space



Summary

- **Kernel Launch Configuration:**
 - Launch enough threads per SM to hide latency
 - Launch enough threadblocks to load the GPU
- **Global memory:**
 - Maximize throughput (GPU has lots of bandwidth, use it effectively)
- **Use shared memory when applicable (over 1 TB/s bandwidth)**
- **GPU-CPU interaction:**
 - Minimise CPU/GPU idling, maximize PCIe throughput
- **Use analysis/profiling when optimising:**
 - “Analysis-driven Optimization” part of the tutorial following

Thank you