

# GPU Performance Analysis and Optimization

**Paulius Mickevicius**

Developer Technology, NVIDIA



# Goals of This Talk

- **Give insight into how hardware operates**
  - Fermi and Kepler
- **Connect hardware operation to performance**
- **Provide guidelines for diagnosing and optimizing performance limiters**
  - Illustrate the use with brief case studies
    - some Fermi, some Kepler
- **Quick review of things to keep in mind when transitioning from Fermi to Kepler**

# Outline

- **Requirements for GPU performance**
- **Exposing Sufficient parallelism**
- **Optimizing GPU Memory Access**
  - Global memory
  - Shared memory
- **Optimizing GPU instruction execution**
- **Review of Kepler considerations**

# Additional Resources

- **More information on topics for which we don't have time in this session**
- **Kepler architecture:**
  - GTC12 Session S0642: Inside Kepler
  - Kepler whitepapers (<http://www.nvidia.com/object/nvidia-kepler.html>)
- **Assessing performance limiters:**
  - GTC10 Session 2012: Analysis-driven Optimization (slides 5-19):
    - [http://www.nvidia.com/content/GTC-2010/pdfs/2012\\_GTC2010v2.pdf](http://www.nvidia.com/content/GTC-2010/pdfs/2012_GTC2010v2.pdf)
- **Profiling tools:**
  - GTC12 sessions:
    - S0419: Optimizing Application Performance with CUDA Performance Tools
    - S0420: Nsight IDE for Linux and Mac
    - ...
  - CUPTI documentation (describes all the profiler counters)
    - Included in every CUDA toolkit (/cuda/extras/cupti/doc/Cupti\_Users\_Guide.pdf)
- **Register spilling:**
  - Webinar:
    - Slides: [http://developer.download.nvidia.com/CUDA/training/register\\_spilling.pdf](http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf)
    - Video: [http://developer.download.nvidia.com/CUDA/training/CUDA\\_LocalMemoryOptimization.mp4](http://developer.download.nvidia.com/CUDA/training/CUDA_LocalMemoryOptimization.mp4)
- **GPU computing webinars in general:**
  - <http://developer.nvidia.com/gpu-computing-webinars>

# Determining Performance Limiter for a Kernel

- **Kernel performance is limited by one of:**
  - Memory bandwidth
  - Instruction bandwidth
  - Latency
    - Usually the culprit when neither memory nor instruction throughput is a high-enough percentage of theoretical bandwidth
- **Determining which limiter is the most relevant for your kernel**
  - Not really covered in this presentation due to time
  - Covered in more detail in session 2012 of GTC2010:
    - Slides: 5-19, 45-49
    - Link: [http://www.nvidia.com/content/GTC-2010/pdfs/2012\\_GTC2010v2.pdf](http://www.nvidia.com/content/GTC-2010/pdfs/2012_GTC2010v2.pdf)
    - Video: <http://nvidia.fullviewmedia.com/gtc2010/0923-san-jose-2012.html>

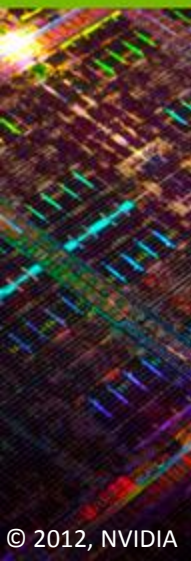
# Topics For Ninjas

- **Topics relevant to 1% (or less) of codes and developers**
  - So, if you're not trying to squeeze out the last few % of performance, you can ignore these
- **Indicated with the following logo:**



# Main Requirements for GPU Performance

- **Expose sufficient parallelism**
- **Coalesce memory access**
- **Have coherent execution within warp**



# EXPOSING SUFFICIENT PARALLELISM

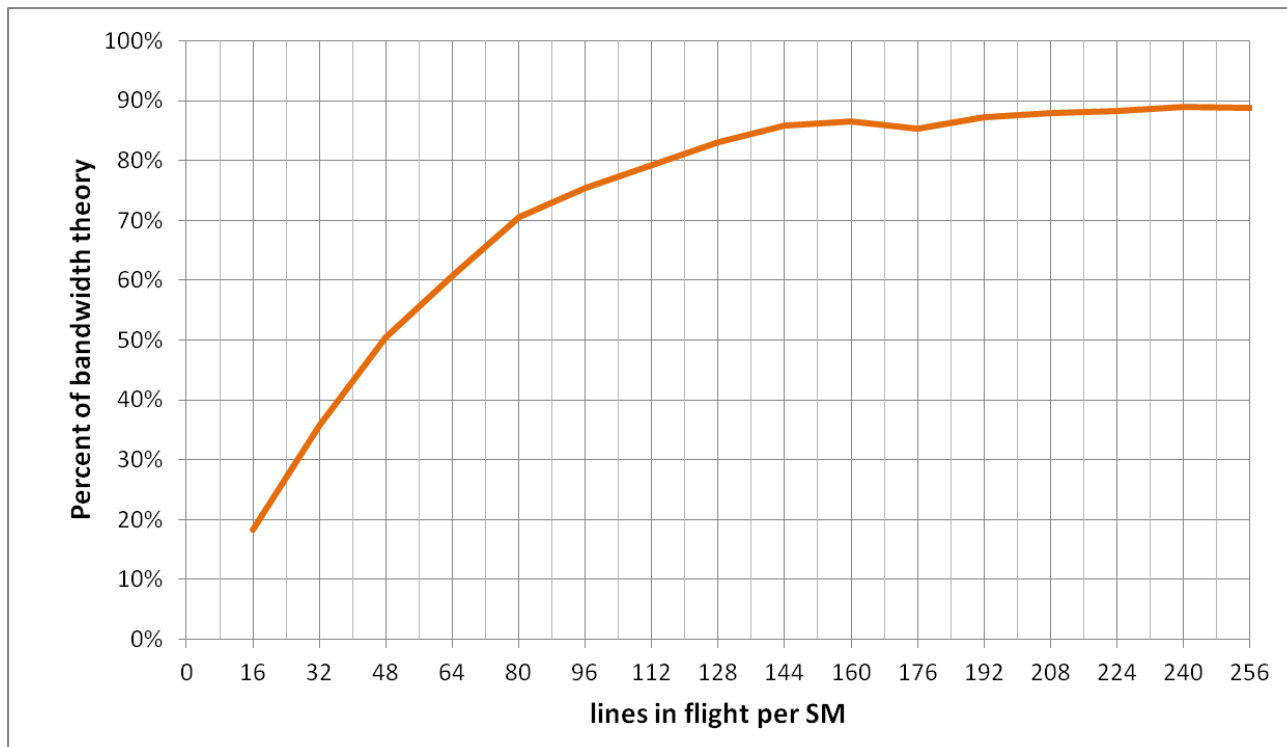


# Kepler: Level of Parallelism Needed

- **To saturate instruction bandwidth:**
  - Fp32 math: ~1.7K independent instructions per SM
  - Lower for other, lower-throughput instructions
  - Keep in mind that Kepler SM can track up to 2048 threads
- **To saturate memory bandwidth:**
  - 100+ independent lines per SM

# Memory Parallelism

- **Achieved Kepler memory throughput**
  - As a function of the number of independent requests per SM
  - Request: 128-byte line



# Exposing Sufficient Parallelism

- **What hardware ultimately needs:**
  - Arithmetic pipes:
    - sufficient number of independent instructions
      - accommodates multi-issue and latency hiding
  - Memory system:
    - sufficient requests in flight to saturate bandwidth
- **Two ways to increase parallelism:**
  - More independent work within a thread (warp)
    - ILP for math, independent accesses for memory
  - More concurrent threads (warps)



# Occupancy

- **Occupancy: number of concurrent threads per SM**
  - Expressed as either:
    - the number of threads (or warps),
    - percentage of maximum threads
- **Determined by several factors**
  - (refer to Occupancy Calculator, CUDA Programming Guide for full details)
  - Registers per thread
    - SM registers are partitioned among the threads
  - Shared memory per threadblock
    - SM shared memory is partitioned among the blocks
  - Threads per threadblock
    - Threads are allocated at threadblock granularity

## Kepler SM resources

- 64K 32-bit registers
- Up to 48 KB of shared memory
- Up to 2048 concurrent threads
- Up to 16 concurrent threadblocks



# Occupancy and Performance

- **Note that 100% occupancy isn't needed to reach maximum performance**
  - Once the “needed” occupancy is reached, further increases won't improve performance
- **Needed occupancy depends on the code**
  - More independent work per thread -> less occupancy is needed
  - Memory-bound codes tend to need more occupancy
    - Higher latency than for arithmetic, need more work to hide it
  - We'll discuss occupancy for memory- and math-bound codes later in the presentation



# Exposing Parallelism: Grid Configuration

- **Grid: arrangement of threads into threadblocks**
- **Two goals:**
  - Expose enough parallelism to an SM
  - Balance work across the SMs
- **Several things to consider when launching kernels:**
  - Number of threads per threadblock
  - Number of threadblocks
  - Amount of work per threadblock

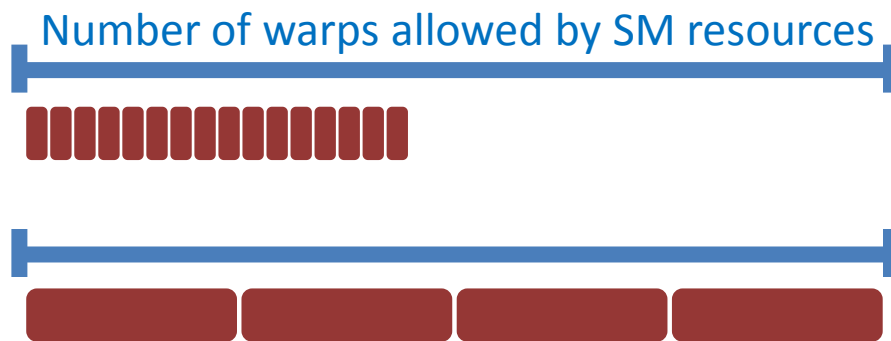
# Threadblock Size and Occupancy

- **Threadblock size is a multiple of warp size (32)**
  - Even if you request fewer threads, HW rounds up
- **Threadblocks can be too small**
  - Kepler SM can run up to 16 threadblocks concurrently
  - SM may reach the block limit before reaching good occupancy
    - Example: 1-warp blocks -> 16 warps per Kepler SM (probably not enough)
- **Threadblocks can be too big**
  - Quantization effect:
    - Enough SM resources for more threads, not enough for another large block
    - A threadblock isn't started until resources are available for all of its threads



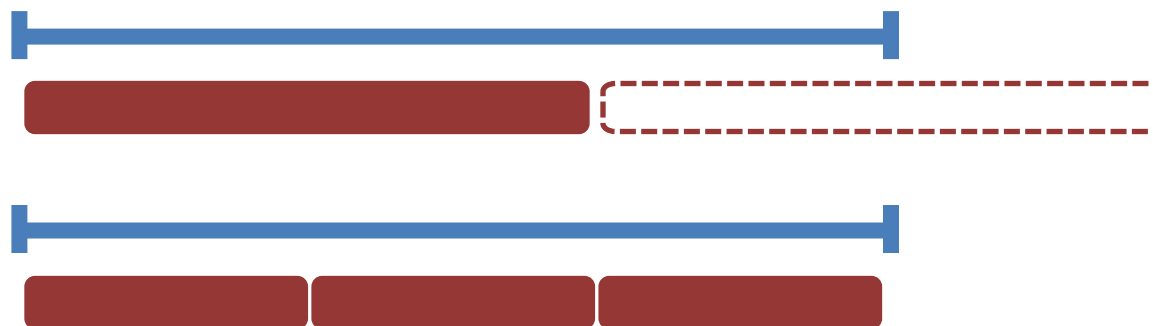
# Threadblock Sizing

Too few  
threads  
per block



- **SM resources:**
  - Registers
  - Shared memory

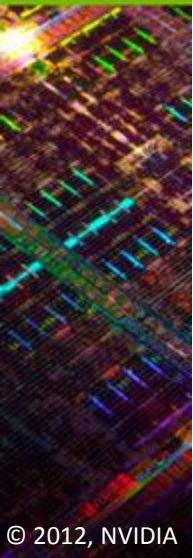
Too many  
threads  
per block





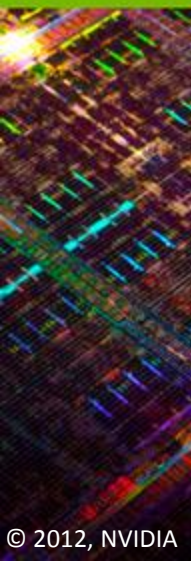
# Case Study 1: Threadblock Sizing

- **Non-hydrostatic Icosahedral Model (NIM)**
  - Global weather simulation code, NOAA
  - vdmintv kernel:
    - 63 registers per thread, 3840 bytes of SMEM per warp
    - At most 12 warps per Fermi SM (limited by SMEM)
- **Initial grid: 32 threads per block, 10,424 blocks**
  - Blocks are too small:
    - 8 warps per SM, limited by number of blocks (Fermi's limit was 8)
    - Code achieves a small percentage (~30%) of both math and memory bandwidth
  - Time: 6.89 ms



# Case Study 1: Threadblock Sizing

- **Optimized config: 64 threads per block, 5,212 blocks**
  - Occupancy: 12 warps per SM, limited by SMEM
  - Time: 5.68 ms (1.21x speedup)
- **Further optimization:**
  - Reduce SMEM consumption by moving variables to registers
    - 63 registers per thread, 1536 bytes of SMEM per warp
  - Occupancy: 16 warps per SM, limited by registers
  - Time: 3.23 ms (2.13x speedup over original)



# Waves and Tails

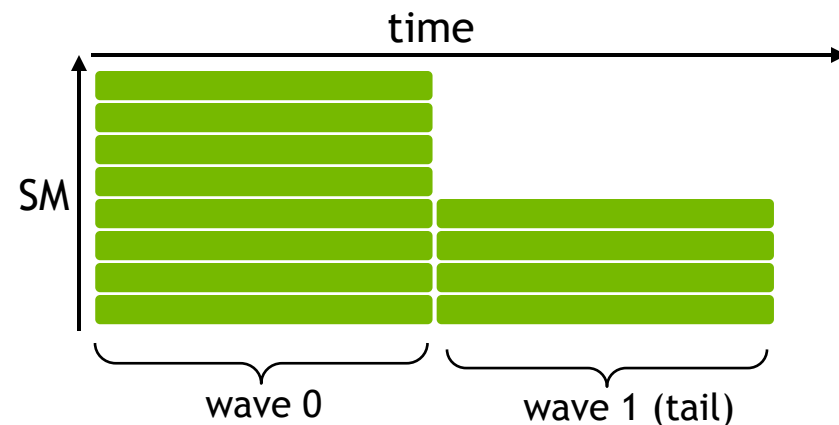


- **Wave of threadblocks**
  - A set of threadblocks that run concurrently on GPU
  - Maximum size of the wave is determined by:
    - How many threadblocks can fit on one SM
      - Number of threads per block
      - Resource consumption: registers per thread, SMEM per block
    - Number of SMs
- **Any grid launch will be made up of:**
  - Some number of *full* waves
  - Possibly one *tail*: wave with fewer than possible blocks
    - Last wave by definition
    - Happens if the grid size is not divisible by wave size

# Tail Effect



- **Tail underutilizes GPU**
  - Impacts performance if tail is a significant portion of time
- **Example:**
  - GPU with 8 SMs
  - Code that can run 1 threadblock per SM at a time
    - Wave size = 8 blocks
  - Grid launch: 12 threadblocks
- **2 waves:**
  - 1 full
  - Tail with 4 threadblocks
    - Tail utilizes 50% of GPU, compared to full-wave
    - Overall GPU utilization: 75% of possible



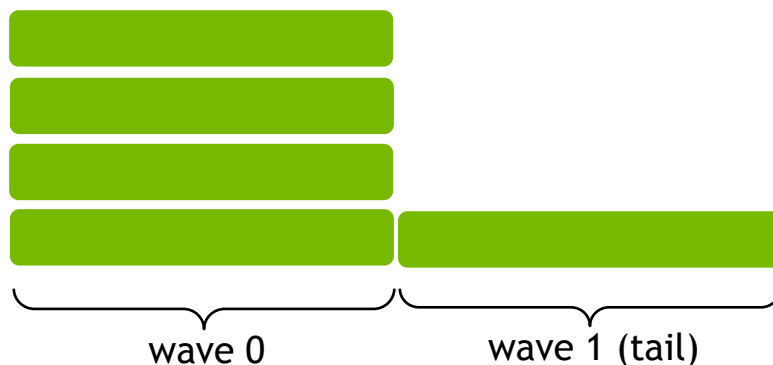


# Tail Effect



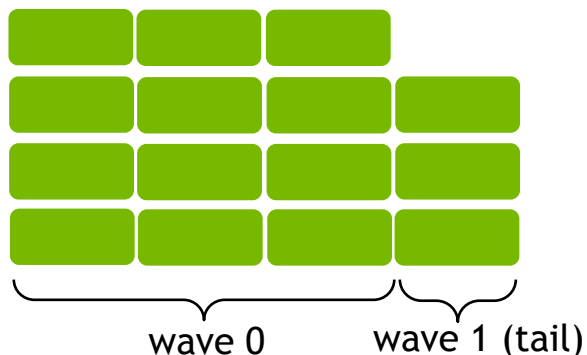
- **A concern only when:**
  - Launching few threadblocks (no more than a few waves)
  - Tail effect is negligible when launching 10s of waves
    - If that's your case, you can ignore the following info
- **Tail effect can occur even with perfectly-sized grids**
  - Threadblocks don't stay in lock-step
- **To combat tail effect:**
  - Spread the work of one thread among several threads
    - Increases the number of blocks -> increases the number of waves
  - Spread the threads of one block among several
    - Improves load balancing during the tail
  - Launch independent kernels into different streams
    - Hardware will execute threadblocks from different kernels to fill the GPU

# Tail Effect: Large vs Small Threadblocks



- **2 waves of threadblocks**

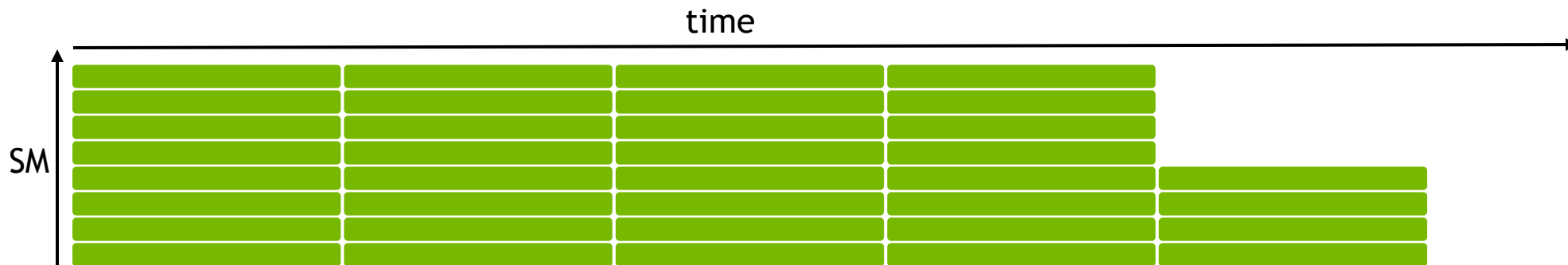
- Tail is running at 25% of possible
- Tail is 50% of time
  - Could be improved if the tail work could be better balanced across SMs



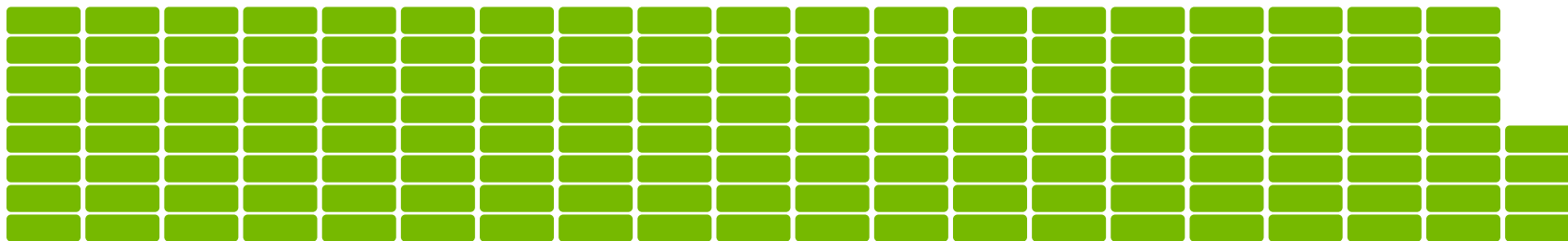
- **2 waves of threadblocks**

- Tail is running at 75% of possible
- Tail is 25% of time
  - Tail work is spread across more threadblocks, better balanced across SMs
- Estimated speedup: 1.5x (time reduced by 33%)

# Tail Effect: Few vs Many Waves of Blocks



80% of time code runs at 100% of its ability, 20% of time it runs at 50% of ability: 90% of possible



95% of time code runs at 100% of its ability, 5% of time it runs at 50% of ability: 97.5% of possible

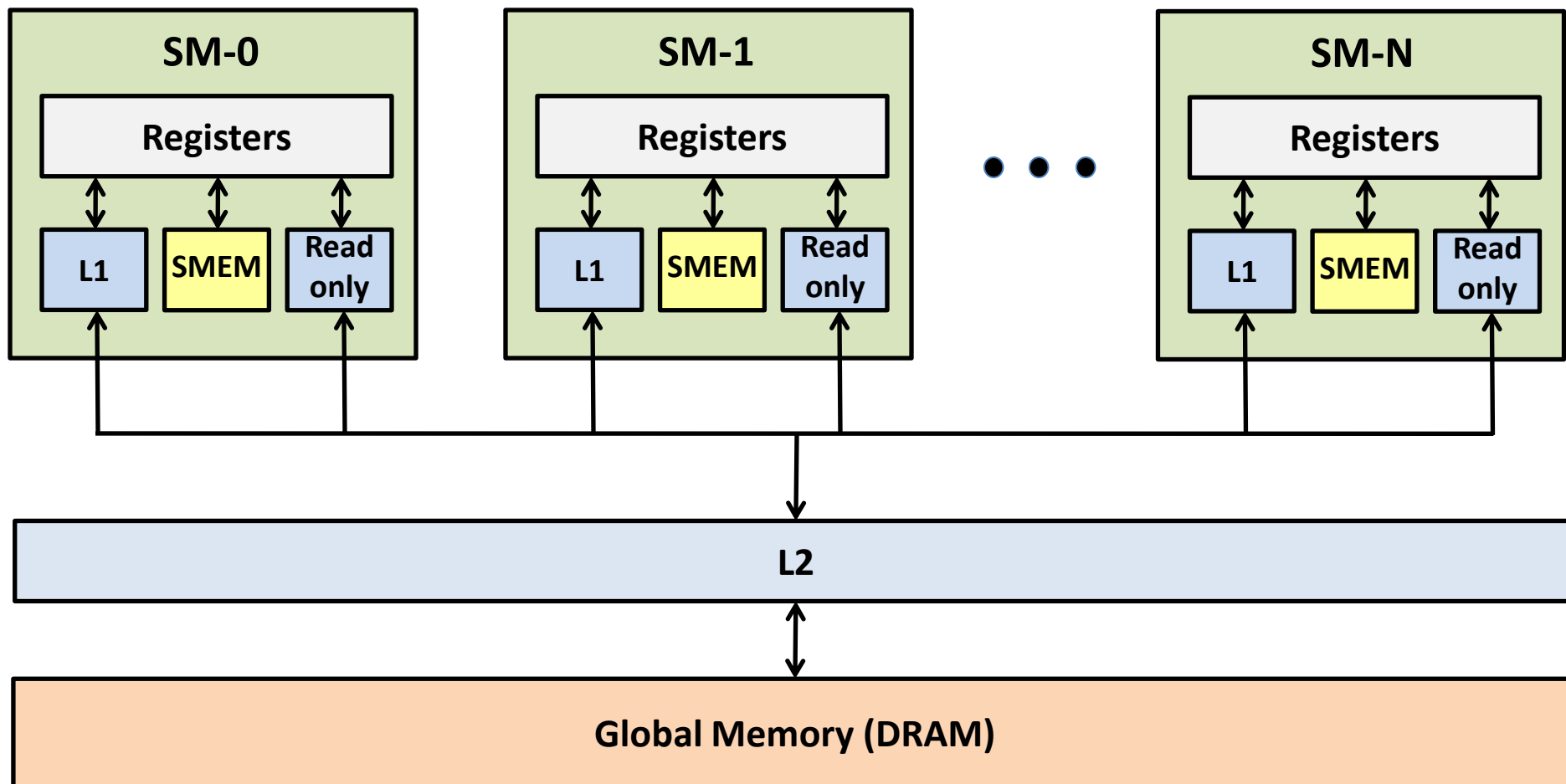
# General Guidelines

- **Threadblock size choice:**
  - Start with 128-256 threads per block
    - Adjust up/down by what best matches your function
    - Example: stencil codes prefer larger blocks to minimize halos
  - Multiple of warp size (32 threads)
  - If occupancy is critical to performance:
    - Check that block size isn't precluding occupancy allowed by register and SMEM resources
- **Grid size:**
  - 1,000 or more threadblocks
    - 10s of waves of threadblocks: no need to think about tail effect
    - Makes your code ready for several generations of future GPUs



# GLOBAL MEMORY

# Kepler Memory Hierarchy



# Memory Hierarchy Review

- **Registers**
  - Storage local to each threads
  - Compiler-managed
- **Shared memory / L1**
  - 64 KB, program-configurable into `shared:L1`
  - Program-managed
  - Accessible by all threads in the same threadblock
  - Low latency, high bandwidth: 1.5-2 TB/s on Kepler GK104
- **Read-only cache**
  - Up to 48 KB per Kepler SM
  - Hardware-managed (also used by texture units)
  - Used for read-only GMEM accesses (not coherent with writes)
- **L2**
  - Up to: 512 KB on Kepler GK104, 1.5 MB on Kepler GK110 (768 KB on Fermi)
  - Hardware-managed: all accesses to global memory go through L2, including CPU and peer GPU
- **Global memory**
  - Accessible by all threads, host (CPU), other GPUs in the same system
  - Higher latency (400-800 cycles)
  - Tesla K10 bandwidth: 2x160 GB/s (2 chips on a board)

# Blocking for L1, Read-only, L2 Caches

- **Short answer: DON'T**
- **GPU caches are not intended for the same use as CPU caches**
  - Smaller size (especially per thread), so not aimed at temporal reuse
  - Intended to smooth out some access patterns, help with spilled registers, etc.
- **Usually not worth trying to cache-block like you would on CPU**
  - 100s to 1,000s of run-time scheduled threads competing for the cache
  - If it is possible to block for L1 then it's possible block for SMEM
    - Same size
    - Same or higher bandwidth
    - Guaranteed locality: hw will not evict behind your back

# L1 Sizing

- **Shared memory and L1 use the same 64KB**
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: `cudaDeviceSetCacheConfig()`, `cudaFuncSetCacheConfig()`
- **Large L1 can improve performance when:**
  - Spilling registers (more lines in the cache -> fewer evictions)
  - Some offset, small-stride access patterns
- **Large SMEM can improve performance when:**
  - Occupancy is limited by SMEM



# Global Memory Operations

- **Memory operations are executed per warp**
  - 32 threads in a warp provide memory addresses
  - Hardware determines into which lines those addresses fall
- **Stores:**
  - Invalidate L1, go at least to L2, 32-byte granularity
- **Three types of loads:**
  - Caching (default)
  - Non-caching
  - Read-only (new option in GK110)



# Load Operation

- **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
    - Does not hit in L1, invalidates the line if it's in L1 already
  - Load granularity is 32 bytes
- **Read-only**
  - Loads via read-only cache:
    - Attempts to hit in Read-only cache, then L2, then GMEM
  - Load granularity is 32 bytes

# Read-only Loads

- **Go through the read-only cache**
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- **Two ways to enable:**
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: `__restrict__ const`
    - All other pointer arguments: `__restrict__`
      - Conveys to compiler that no aliasing will occur
  - Using `__ldg()` intrinsic
    - Requires no pointer decoration
  - Requires GK110 hardware
    - On prior hardware you can get similar functionality with textures

# Read-only Loads

- **Go through the read-only cache**
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- **Two ways to enable:**
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: `__restrict`
    - All other pointer arguments
      - Conveys to compiler that
  - Using `__ldg()` intrinsic
    - Requires no pointer decoration
  - Requires GK110 hardware
    - On prior hardware you

```
__global__ void kernel( __restrict__ int *output,  
                        __restrict__ const int *input )  
{  
    ...  
    output[idx] = ... + input[idx];  
}
```

# Read-only Loads

- **Go through the read-only cache**
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- **Two ways to enable:**
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: `__restrict`
    - All other pointer arguments
      - Conveys to compiler that pointer is not used for writes
  - Using `__ldg()` intrinsic
    - Requires no pointer decoration
  - Requires GK110 hardware
    - On prior hardware you

```
__global__ void kernel( int *output,
                        int *input )
{
    ...
    output[idx] = ... + __ldg( input[idx] );
}
```

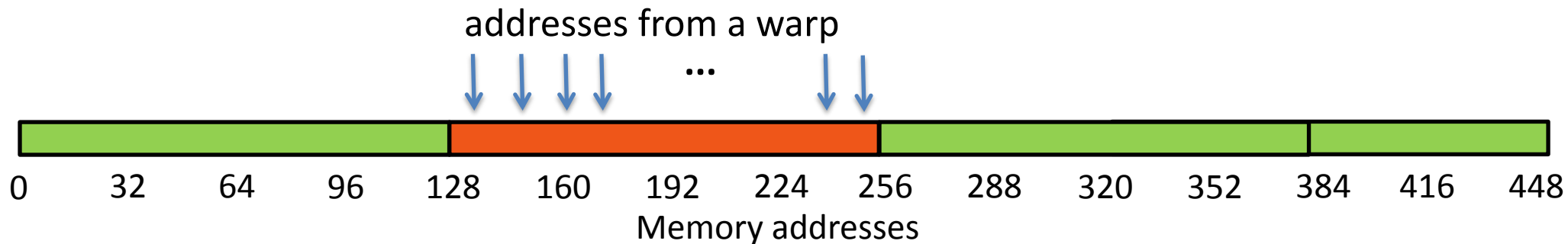


# Load Caching

- **Non-caching loads can improve performance when:**
  - Loading scattered words or only part of a warp issues a load
    - Benefit: memory transaction is smaller, so useful payload is a larger percentage
    - Loading halos, for example
  - Spilling registers (reduce line fighting with spillage)
- **Read-only loads:**
  - Can improve performance for scattered reads
  - Latency is a bit higher than for caching/non-caching loads

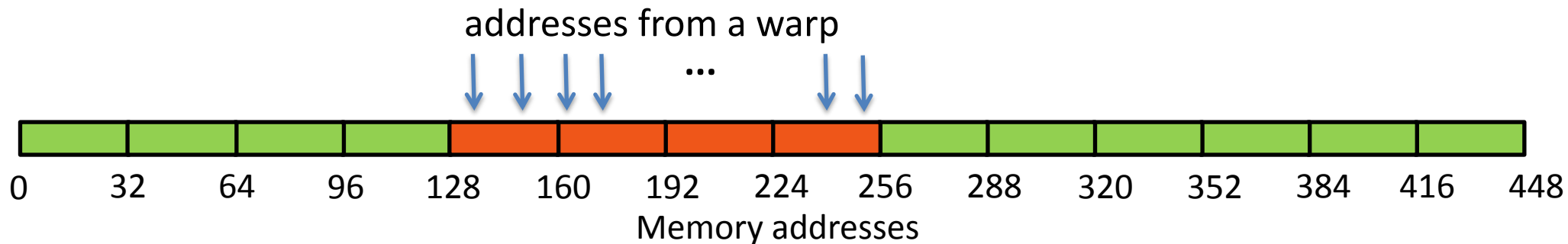
# Caching Load

- **Scenario:**
  - 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%



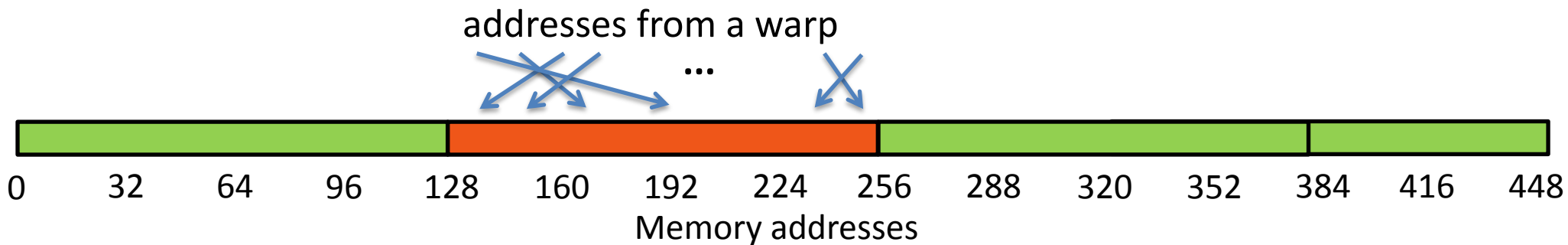
# Non-caching/Read-only Load

- **Scenario:**
  - 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%



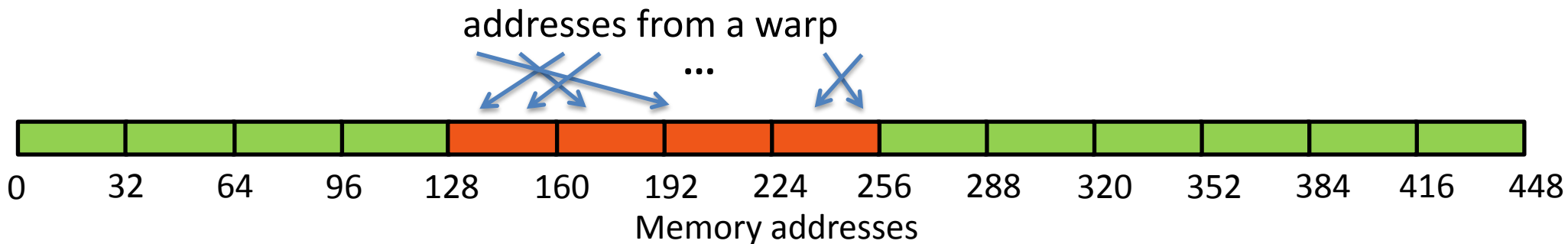
# Caching Load

- **Scenario:**
  - 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/Read-only Load

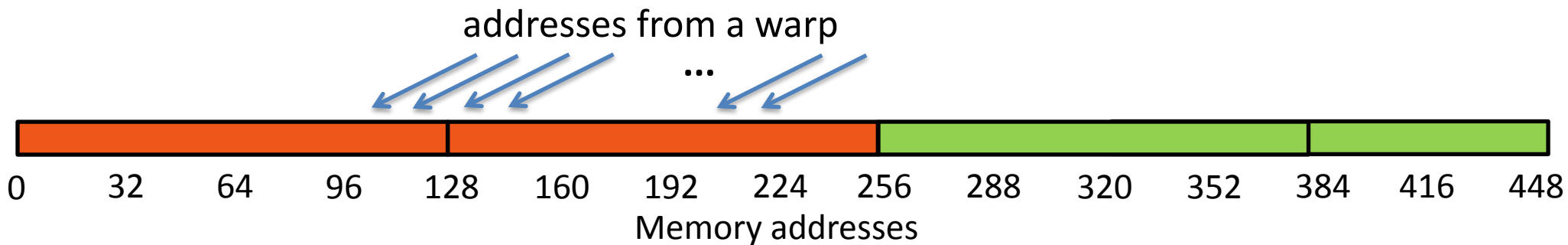
- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 4 segments**
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%





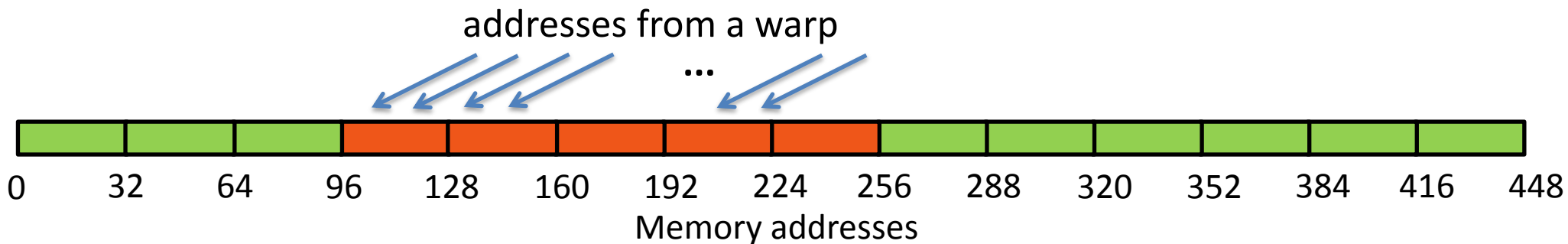
# Caching Load

- **Scenario:**
  - 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%



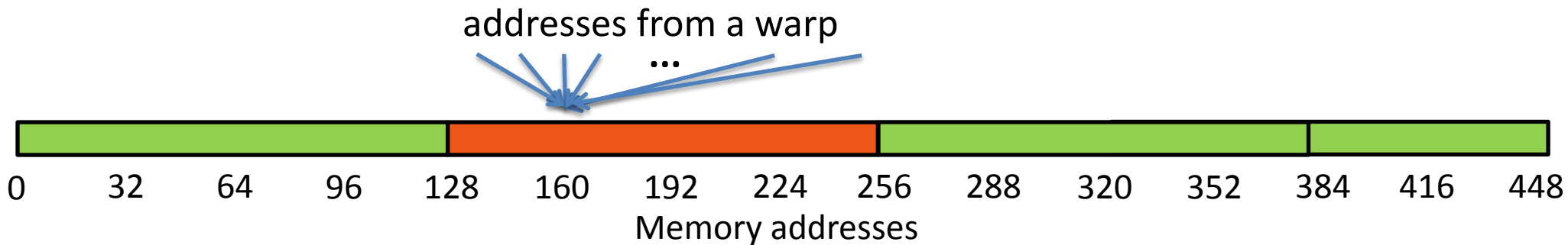
# Non-caching/Read-only Load

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



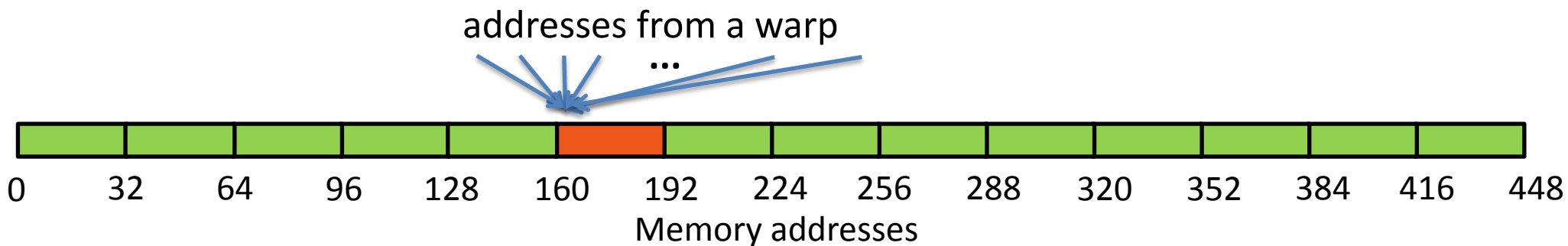
# Caching Load

- **Scenario:**
  - 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%



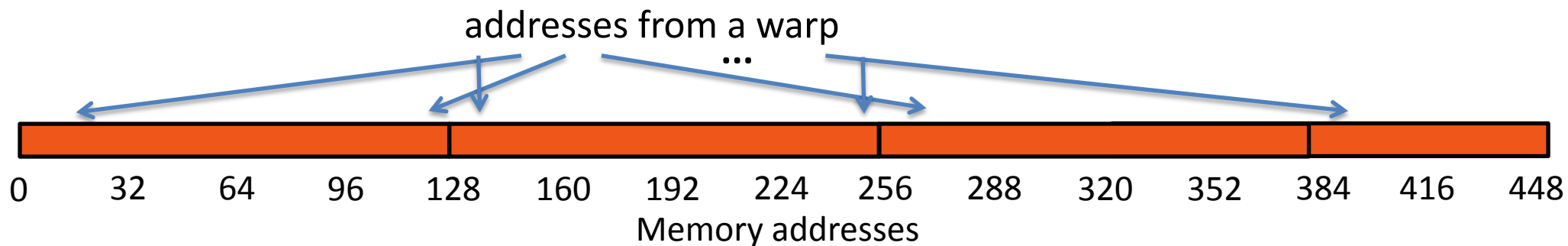
# Non-caching/Read-only Load

- **Scenario:**
  - 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%



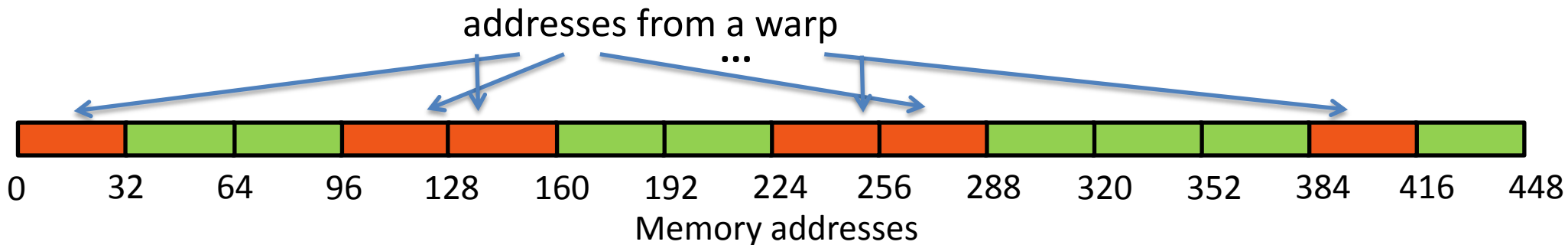
# Caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within  $N$  cache-lines**
  - Warp needs 128 bytes
  - $N*128$  bytes move across the bus on a miss
  - Bus utilization:  $128 / (N*128)$



# Non-caching/Read-only Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within  $N$  segments**
  - Warp needs 128 bytes
  - $N*32$  bytes move across the bus on a miss
  - Bus utilization:  $128 / (N*32)$  (4x higher than caching loads)





# Memory Throughput Analysis

- **Two perspectives on the throughput:**
  - Application's point of view:
    - count only bytes requested by application
  - HW point of view:
    - count all bytes moved by hardware
- **The two views can be different:**
  - Memory is accessed at 32 or 128 byte granularity
    - Scattered/offset pattern: application doesn't use all the hw transaction bytes
  - Broadcast: the same small transaction serves many threads in a warp
- **Two aspects to inspect for performance impact:**
  - Address pattern
  - Number of concurrent accesses in flight



# Causes for Suboptimal Memory Performance

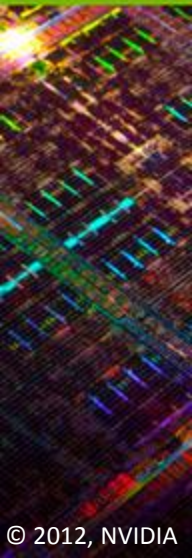
- **Suboptimal address patterns**
  - Throughput from HW point of view is significantly higher than from app point of view
  - Four general categories:
    - 1) **Offset** (not line-aligned) warp addresses
    - 2) **Large strides** between threads within a warp
    - 3) Each thread accesses a **large contiguous region**
    - 4) **Irregular** (scattered) addresses
- **Insufficient concurrent accesses**
  - Arithmetic intensity is low (code should be bandwidth-bound)
  - Throughput from HW point of view is much lower than theory
    - Say, below 60%

# Two Ways to Investigate Address Patterns

- **Profiler-computed load and store efficiency**
  - Efficiency = bytes requested by the app / bytes transferred
  - Accurate, but will slow down code substantially:
    - Bytes-requested is measured by profiler instrumenting code for some load/store instruction
    - Thus, you may want to run for smaller data set
- **Transactions per request:**
  - Fast: requires collecting 5 profiler counters
  - Accurate if all accesses are for the same word-size (4-byte, 8-byte, etc.)
    - Less accurate if a kernel accesses words of varying sizes (still OK if you know statistical distribution)
  - Loads:
    - Make sure to use caching loads for this analysis
    - Compute  $(l1\_global\_load\_hit + l1\_global\_load\_miss)$  to  $gld\_request$  ratio
    - Compare to the ideal ratio:  $32 \text{ threads/warp} * \text{word size in bytes} / 128 \text{ bytes per line}$ 
      - 1.0 for 4-byte words, 2.0 for 8-byte words, 1.5 if 50% accesses are 4-byte and 50% are 8-byte
  - Stores:
    - Compute  $global\_store\_transaction$  to  $gst\_request$  ratio
    - Compare to the ideal ratio:  $32 * \text{word size in bytes} / 128$

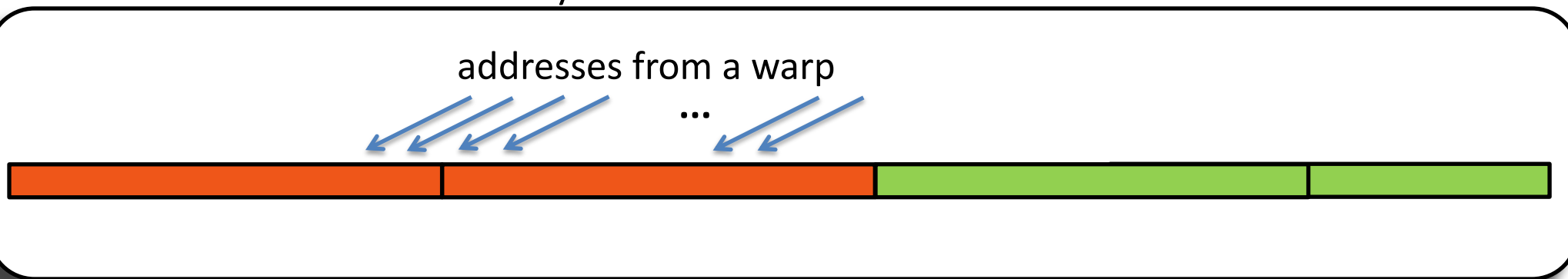
# Pattern Category 1: Offset Access

- **Cause:**
  - Region addressed by a warp is not aligned on cache-line boundary
- **Issue:**
  - Wasted bandwidth: only a fraction of some lines is used
  - Some increase in latency
- **Symptom:**
  - Transactions per request 1.5-2.0x higher than ideal
  - Likely: moderate to medium L1 hit rate
- **Remedies:**
  - Extra padding for data to force alignment
  - Try non-caching loads, read-only loads
    - Reduce overfetched bytes, but don't fully solve the problem



# Pattern Category: Offset Access

- **Cause:**
  - Region addressed by a warp is not aligned on cache-line boundary
- **Issue:**
  - Wasted bandwidth: only a fraction of some lines is used
  - Some increase in latency



- Try non-caching loads, read-only loads
  - Reduce overfetched bytes, but don't fully solve the problem

# Pattern Category 1: Offset Access

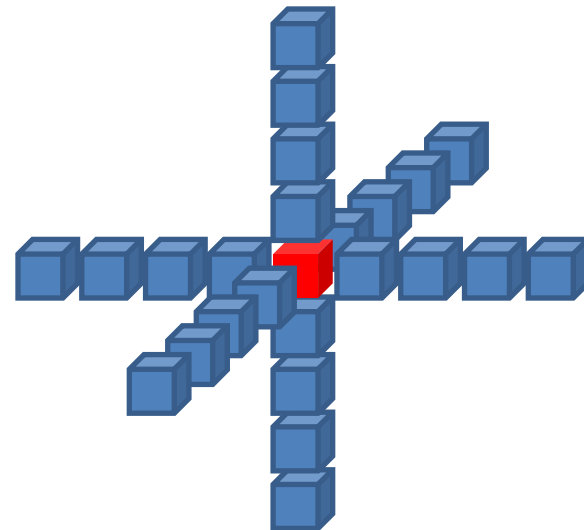
- **Cause:**
  - Region addressed by a warp is not aligned on cache-line boundary
- **Issue:**
  - Wasted bandwidth: only a fraction of some lines is used
  - Some increase in latency
- **Symptom:**
  - Transactions per request 1.5-2.0x higher than ideal
  - Likely: moderate to medium L1 hit rate
- **Remedies:**
  - Full: extra padding for data to force alignment
  - Partial: non-caching loads, read-only loads
    - Reduce overfetched bytes, but don't fully solve the problem





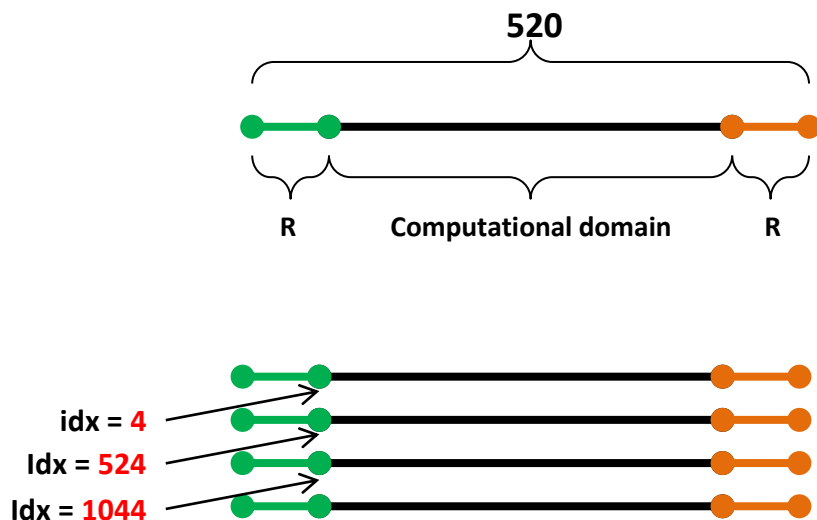
# Case Study 2: Offset Address Pattern

- **Isotropic RTM, 8<sup>th</sup> order in space**
  - Seismic processing: propagate pressure wave
  - Major component: 3DFD computation (Laplacian discretization)
- **Data requires padding to avoid out of bounds accesses**
  - Minimum: pad by 4 elements on all 6 sides of data
  - Minimum padding causes offset access pattern
- **Diagnosing:**
  - Transactions per request:
    - Ideal ratio: 1 (single-precision float code)
    - Loads: 1.78
    - Stores: 2.00
  - L1 hit rate: 15.6%



# Case Study 2: Cause

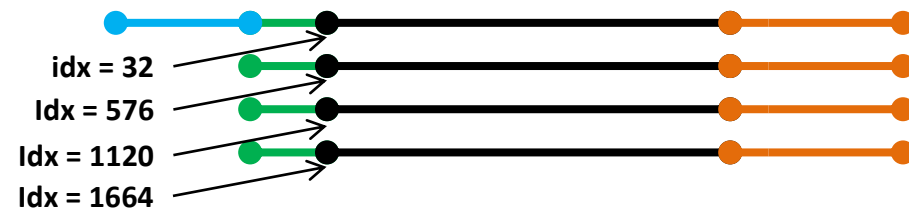
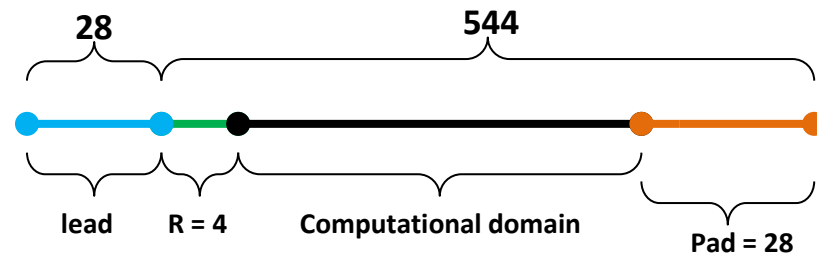
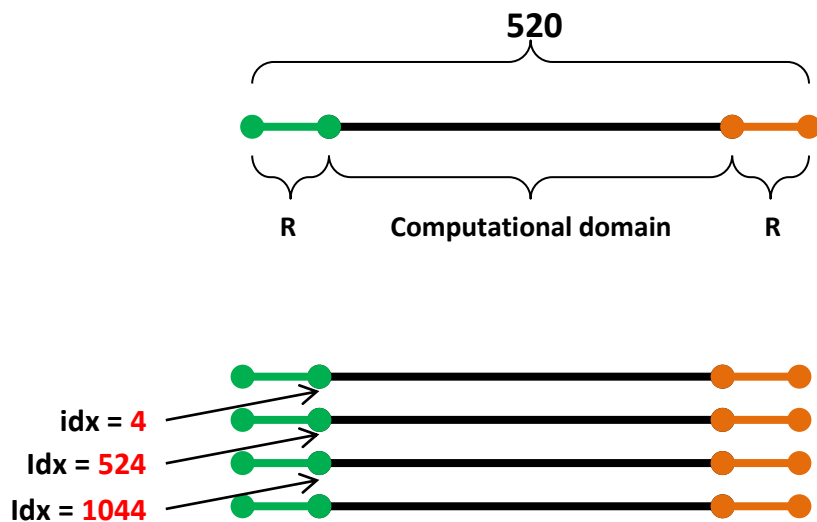
- Looking at the 2 fastest-varying dimensions, 512x512x512 problem
  - Computational domain: 512 cells per row
  - R=4 (stencil “radius”)



- For perfect coalescing we need:
  - Row size, after padding, to be a multiple of 128 bytes (32 floats)
  - The first non-padding element to be at a multiple of 128 bytes

# Case Study 2: Remedy

- Looking at the 2 fastest-varying dimensions, 512x512x512 problem
  - Computational domain: 512 cells per row
  - R=4 (stencil “radius”)



# Case Study 2: Result



- **Programming effort: 3 lines**
  - 2 additional lines for allocation
  - 1 additional line to adjust the pointer to skip past the lead-padding before passing it to the function
  - No changes to the function code
- **Performance impact:**
  - Kepler: 1.20x speedup
  - Fermi: 1.18x speedup
  - 1.0 transactions per request, for both loads and stores

# Pattern Category 2: Large Inter-thread Stride

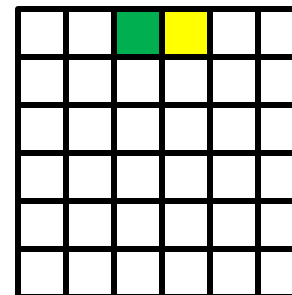
- **Cause:**
  - Successive threads access words at regular distance, distance greater than one word
    - GPU access words: 1, 2, 4, 8, 16 bytes
  - Example cases:
    - Data transpose (warp accessing a column in a row-major data structure)
    - Cases where some data is accessed in transposed fashion, other isn't
- **Issues:**
  - Wasted bandwidth: moves more bytes than needed
  - Substantially increased latency:
    - If a warp address pattern requires N transactions, the instruction is issued N times
- **Symptoms:**
  - Transactions per request much greater than ideal
- **Remedies:**
  - Full: change data layout, stage accesses via SMEM
  - Partial: non-caching loads, read-only loads

# Case Study 3: Matrix Transpose

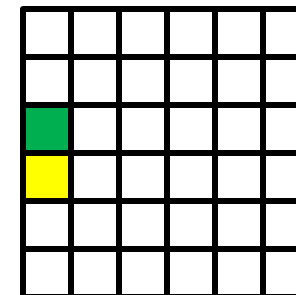
- **Double-precision elements**
- **Row-major storage order**
- **Naïve implementation:**
  - Square threadblocks
  - Each thread:
    - Computes its global x and y coordinates
    - Reads from  $(x,y)$ , writes to  $(y,x)$

 thread X  
 thread (X+1)

READ



WRITE



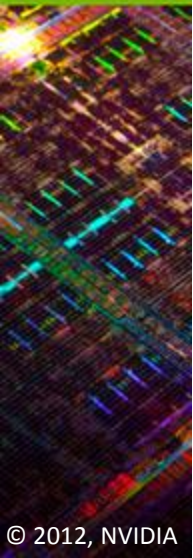


# Case Study 3: Diagnosis

- **Double-precision elements:**
  - ideally 2.0 transactions per request
- **Measured values:**
  - 2.0 lines per load
  - 32 transactions per store
  - 75% of DRAM bandwidth
- **Conclusions:**
  - Performance is bandwidth-limited (75% of theory is very good)
  - Much of the bandwidth is wasted due to store pattern
    - Number of store transactions is 16x higher than ideal

# Case Study 3: Cause and Remedy

- **Cause:**
  - Due to nature of operation, one of the accesses (read or write) will be at large strides
    - 32 doubles (256 bytes) in this case
    - Thus, bandwidth will be wasted as only a portion of a transaction is used by the application
- **Remedy**
  - Stage accesses through shared memory
  - A threadblock:
    - Reads a tile from GMEM to SMEM
    - Transposes the tile in SMEM
    - Write a tile, in a coalesced way, from SMEM to GMEM



# Case Study 3: Result

- **Naïve implementation:**
  - 16.7 ms
  - 32 transactions per store
- **Optimized implementation:**
  - 11.2 ms (1.5x speedup)
  - 2 transactions per store

# Pattern Category 3: Large Contiguous Region Per Thread

- **Cause:**
  - Each thread accesses its own contiguous region of memory, region is several words in size
  - Example: Array of Structures (AoS) data layout
- **Issues:**
  - Wasted bandwidth:
    - Reads: same bytes are fetched redundantly (lines get evicted before all bytes are consumed)
    - Stores: wasted bandwidth since stores happen at 32-byte granularity
  - Substantially increased latency:
    - If a warp address pattern requires N transactions, the instruction is issued N times
- **Symptoms:**
  - Transactions per request much greater than ideal ratio
    - For loads, not a problem if L1 misses per request are equal to the ideal ratio
  - Usually medium to high L1 hit rate
- **Remedies:**
  - Full:
    - Change data layout (Structure of Arrays instead of AoS)
    - Process the region with several threads to get coalescing
  - Partial: read-only loads

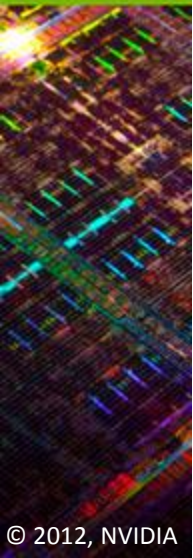
# Case Study 4: SoA vs AoS

- **Global shallow water model**
  - Stencil computation for wave dynamics (height, velocity)
  - Double-precision code
- **Initial implementation:**
  - Array of Structures data layout
    - A structure has 20 fp64 members (160 bytes)
  - Each thread is responsible for one structure:
    - Stencil computation:
      - Read own structure members
      - Read neighbors' structure members
    - Write output structure



# Case Study 4: Diagnosing

- Double-precision code, so ideal ratio is 2.0 for loads and stores
- Measured values:
  - 24.5 L1 lines per load
  - 73% L1 hit rate
  - 6.0 transactions per store
  - Throughputs:
    - 23% of DRAM bandwidth
    - 13% of instruction bandwidth
- Conclusion
  - Performance is latency-limited
    - Both throughputs are small percentages of theory
    - Recall that high reissues of memory instructions increase latency
  - Address pattern wastes bandwidth:
    - transactions per request much higher than 2.0
    - Even with 73% hit rate,  $(1-0.73) * 24.5 = \sim 6.6$  L1 load misses per request





# Case Study 4: Cause

- **Array of Structures data layout:**
  - Threads in a warp access at 160 byte (20 fp64) stride
    - Each thread consumes more than 1 line, but line gets evicted before full use
  - Even after L1 hits, we're reading ~3x more bytes than needed
  - Load and store replays (due to multiple transactions per warp) increase latency, latency is the limiting factor for this code
- **Two possible solutions:**
  - Try reading through read-only cache
    - This is just a partial remedy:
      - Helps reduce wasted bandwidth (smaller granularity for access and caching)
      - Improves, but doesn't resolve the latency increase due to replays
  - Rearrange the data from Array of Structures to Structure of Arrays
    - The ultimate solution, addresses both latency and wasted bandwidth

# Case Study 4: Results

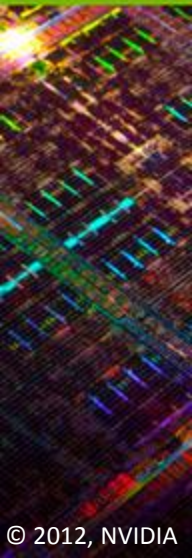
- **Original code (Array of Structures):**
  - Time: 22.8 ms
  - 24.5 transactions per load, 6 transactions per store
- **Code using read-only loads:**
  - Time: 15.7 ms (1.45x speedup over original)
- **Code with Structure of Arrays data layout:**
  - Time: 9.3 ms (2.45x speedup over original)
  - Successive threads access successive words
    - 3 transactions per load request
      - Due to offset halo reads: addressed with non-caching loads: 8.9ms (2.56x speedup)
    - 2 transactions per store request

# Case Study 5: Assigning More Threads per Region

- **CAM HOMME**
  - Climate modeling code, double precision
  - Spectral element code, 4x4x26 elements
  - CUDA Fortran (x26 is the slowest varying dimension)
  - Limiter2d\_zero function:
    - For each element:
      - Read 4x4x26 values from GMEM
      - For each of 26 levels:
        - » Compute the sum over 4x4 values
        - » Adjust the values based on the sum
        - » Write adjusted values to GMEM
- **Initial implementation:**
  - One thread for each of 26 levels

# Case Study 5: Diagnosing

- Ideal transactions per request: 2.0 for loads and stores
- Measured values:
  - Loads: 31.4
  - Stores: 31.3
  - L1 hit rate: 54.9%
  - Achieved throughputs:
    - 21% of DRAM bandwidth
    - 17% of instruction bandwidth
- Conclusions:
  - Performance is latency-limited
  - Address pattern wastes bandwidth:
    - Transactions per request much higher than 2.0
    - Even with 54.9% hit rate,  $(1-0.549) * 31.4 = \sim 14.2$  L1 load misses per request



# Case Study 5: Cause and Remedy

- **Each thread loops through 16 consecutive doubles**
  - Each thread accesses a contiguous region of 128 bytes
    - Threads in a warp address at 128-byte stride
  - Each memory instruction has 32 transactions:
    - Dramatic increase in latency: each instruction is issued 32 times
  - Bandwidth is wasted since lines are fetched redundantly
- **Remedy:**
  - Assign 16 threads per 4x4 level, as opposed to 1
  - No need to rearrange the data



# Case Study 5: Results

- **Initial implementation:**
  - Time: 2.30 ms
  - ~32 transactions per request, both loads and stores
  - Achieves 21% of DRAM bandwidth
- **Optimized implementation:**
  - Time: 0.52 ms (4.45x speedup)
  - ~2 transactions per request, both loads and stores
  - Achieves 63% of DRAM bandwidth



# Pattern Category 4: Irregular Address

- **Cause:**
  - Threads in a warp access many lines, strides are irregular
- **Issues:**
  - Wasted bandwidth: not all the bytes in the lines are used by application

Increased latency if  $N$  transactions are needed per instruction



- **Remedies:**
  - Partial: non-caching loads, read-only loads

# Pattern Category 4: Irregular Address

- **Cause:**
  - Threads in a warp access many lines, strides are irregular
- **Issues:**
  - Wasted bandwidth: not all the bytes in the lines are used by application
  - Increased latency: if **N** transactions are needed per instruction, instruction is issued **N** times
- **Symptoms:**
  - Transactions per request much higher than ideal
  - Low to none L1 hits
- **Remedies:**
  - Partial: non-caching loads, read-only loads

# Summary of Pattern Categories and their Symptoms

Address Pattern Category	Transactions per request	L1 hit rate
<b>Offset</b>	1.5 – 2.0x the ideal	Low – 50%
<b>Large stride between threads</b>	Medium-high	Low or none
<b>Contiguous per thread</b>	High	Medium-high
<b>Scattered-irregular</b>	High	Low or none

# Summary of Pattern Categories and their Symptoms

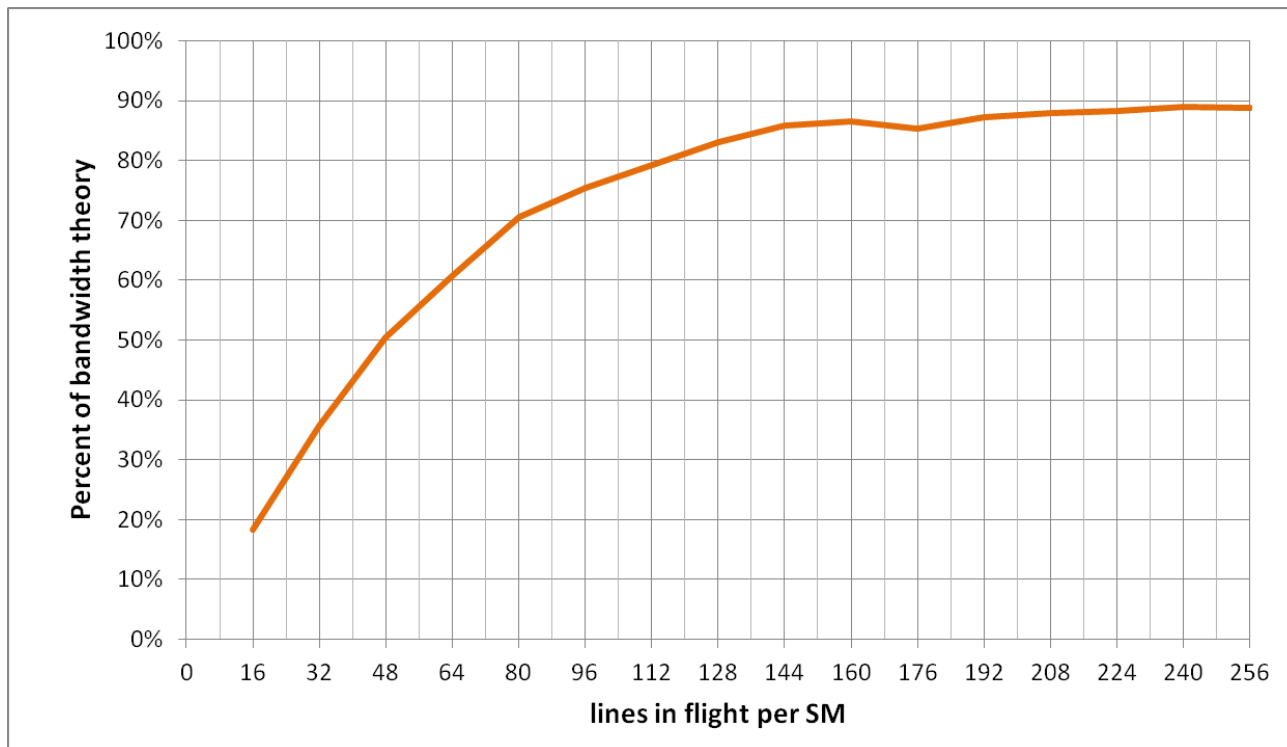
Address Pattern Category	Transactions per request	L1 hit rate
<b>Offset</b>	1.5 – 2.0x the ideal	Low – 50%
<b>Large stride between threads</b>	Medium-high	Low or none
<b>Contiguous per thread</b>	High	Medium-high
<b>Scattered-irregular</b>	High	Low or none

The difference between these patterns is regular (large-stride) vs irregular scatter



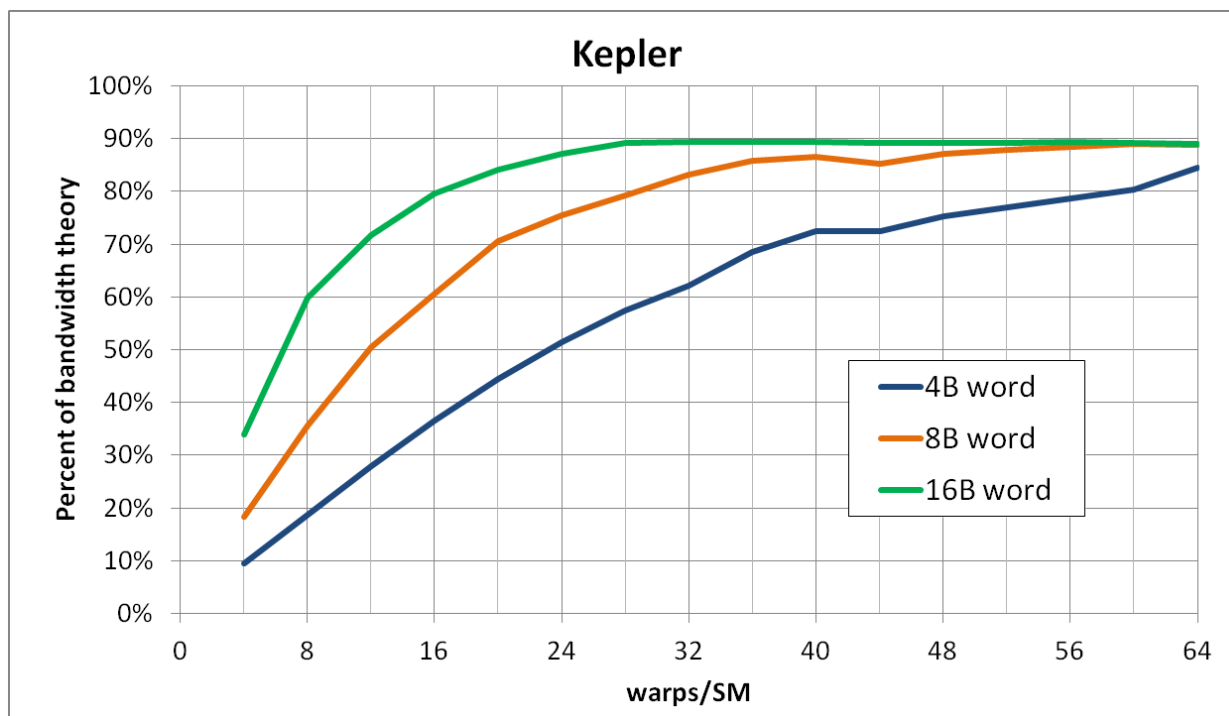
# Having Sufficient Concurrent Accesses

- In order to saturate memory bandwidth, SM must issue enough independent memory requests



# Elements per Thread and Performance

- **Experiment: each warp has 2 concurrent requests (memcpy, one word per thread)**
  - 4B word request: 1 line
  - 8B word request: 2 lines
  - 16B word request: 4 lines



- **To achieve the same throughput at lower occupancy:**
  - Need more independent requests per warp
- **To achieve the same throughput with smaller words:**
  - Need more independent requests per warp

# Optimizing Access Concurrency

- **Have enough concurrent accesses to saturate the bus**
  - Little's law: need  $(\text{mem\_latency}) \times (\text{bandwidth})$  bytes
- **Ways to increase concurrent accesses:**
  - Increase occupancy (run more warps concurrently)
    - Adjust threadblock dimensions
      - To maximize occupancy at given register and smem requirements
    - If occupancy is limited by registers per thread:
      - Reduce register count (`-maxrregcount` option, or `__launch_bounds__`)
  - Modify code to process several elements per thread
    - Doubling elements per thread doubles independent accesses per thread



# Optimizations When Addresses Are Coalesced

- **When looking for more performance and code:**
  - Is memory bandwidth limited
  - Achieves high percentage of bandwidth theory
  - Addresses are coalesced (ideal transaction per request ratio)
- **Consider compression**
  - GPUs provide instructions for converting between fp16, fp32, and fp64 representations:
    - A single instruction, implemented in hw (`__float2half()`, ...)
  - If data has few distinct values, consider lookup tables
    - Store indices into the table
    - Small enough tables will likely survive in caches if used often enough

# Summary: GMEM Optimization

- **Strive for perfect address coalescing per warp**
  - Align starting address (may require padding)
  - A warp will ideally access within a contiguous region
  - Avoid scattered address patterns or patterns with large strides between threads
- **Analyze and optimize:**
  - Use profiling tools (included with CUDA toolkit download)
  - Compare the transactions per request to the ideal ratio
  - Choose appropriate data layout
  - If needed, try read-only, non-caching loads
- **Have enough concurrent accesses to saturate the bus**
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)
  - If needed, process several elements per thread
    - More concurrent loads/stores



# SHARED MEMORY

# Shared Memory

- **On-chip (on each SM) memory**
- **Comparing SMEM to GMEM:**
  - Order of magnitude (20-30x) lower latency
  - Order of magnitude (~10x) higher bandwidth
  - Accessed at bank-width granularity
    - Fermi: 4 bytes
    - Kepler: 8 bytes
    - For comparison: GMEM access granularity is either 32 or 128 bytes
- **SMEM instruction operation:**
  - 32 threads in a warp provide addresses
  - Determine into which 8-byte words (4-byte for Fermi) addresses fall
  - Fetch the words, distribute the requested bytes among the threads
    - Multi-cast capable
    - Bank conflicts cause serialization

# Kepler Shared Memory Banking

- **32 banks, 8 bytes wide**
  - Bandwidth: 8 bytes per bank per clock per SM (256 bytes per clk per SM)
  - 2x the bandwidth compared to Fermi
- **Two modes:**
  - **4-byte access** (default):
    - Maintains Fermi bank-conflict behavior exactly
    - Provides 8-byte bandwidth for certain access patterns
  - **8-byte access:**
    - Some access patterns with Fermi-specific padding may incur bank conflicts
    - Provides 8-byte bandwidth for all patterns (assuming 8-byte words)
  - Selected with `cudaDeviceSetSharedMemConfig()` function arguments:
    - `cudaSharedMemBankSizeFourByte`
    - `cudaSharedMemBankSizeEightByte`

# Kepler 8-byte Bank Mode

- **Mapping addresses to banks:**
  - Successive 8-byte words go to successive banks
  - Bank index:
    - $(8\text{B word index}) \bmod 32$
    - $(4\text{B word index}) \bmod (32 * 2)$
    - $(\text{byte address}) \bmod (32 * 8)$
  - Given the 8 least-significant address bits: ...BBBBBxxx
    - xxx selects the byte within an 8-byte word
    - BBBBB selects the bank
    - Higher bits select a “column” within a bank

# Kepler 4-byte Bank Mode



- Understanding this mapping details matters only if you're trying to get 8-byte throughput in 4-byte mode
  - For all else just think that you have 32 banks, 4-bytes wide
- Mapping addresses to banks:
  - Successive 4-byte words go to successive banks
    - We have to choose between two 4-byte “half-words” for each bank
      - “First” 32 4-byte words go to lower half-words
      - “Next” 32 4-byte words go to upper half-words
  - Given the 8 least-significant address bits: ...HBBBBBxx
    - xx selects the byte with a 4-byte word
    - BBBBB selects the bank
    - H selects the half-word within the bank
    - Higher bits select the “column” within a bank

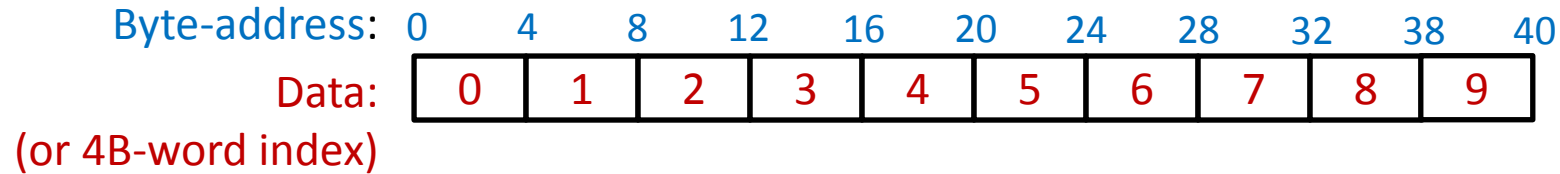




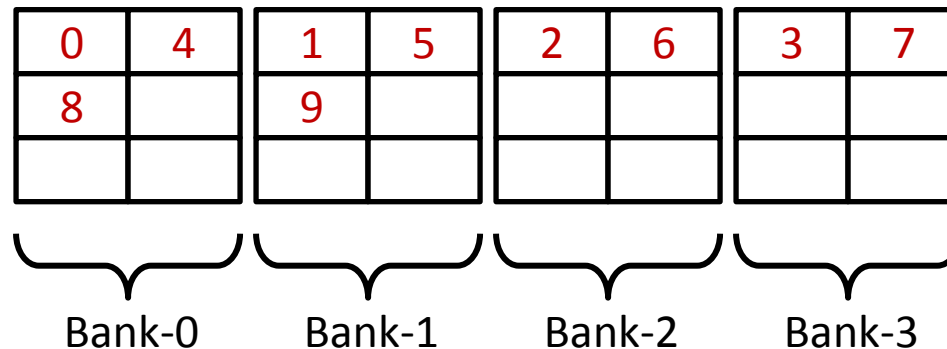
# Kepler 4-byte Bank Mode



- To visualize, let's pretend we have 4 banks, not 32 (easier to draw)
  - Looking at 5 least-significant address bits: ...HBBxx



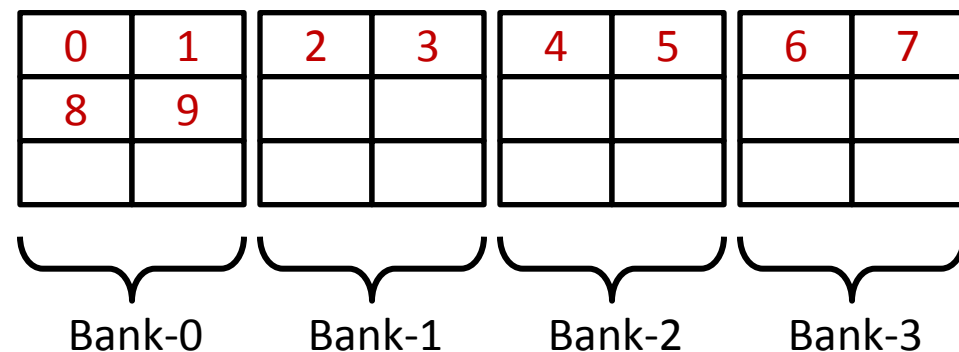
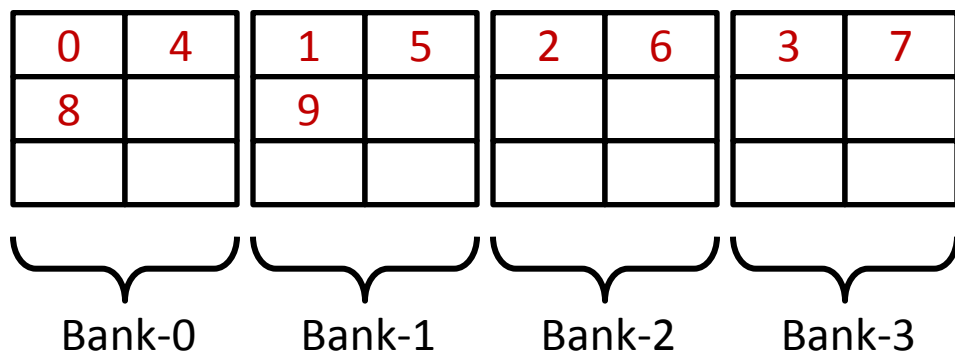
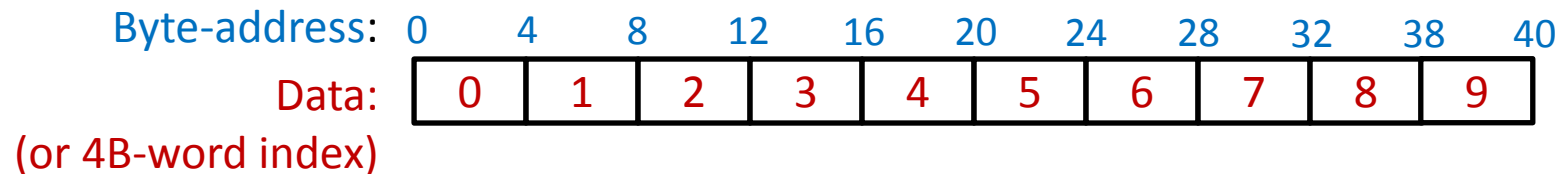
**SMEM:**



# Comparing Bank Modes



- To visualize, let's pretend we have 4 banks, not 32 (easier to draw)
  - Looking at 5 least-significant address bits: ...HBBxx



# Case Study 6: Kepler 8-byte SMEM Access

- **TTI Reverse Time Migration**
  - A seismic processing code, 3DFD
    - fundamental component is applying a 3D stencil to 2 wavefields to compute discrete derivatives
  - Natural to interleave the wavefields in shared memory:
    - store as a float2 structure
    - Also a slight benefit to global memory performance, on both Fermi and Kepler
- **Impact on performance from enabling 8-byte mode:**
  - More SMEM operations as order in space increases
  - 8<sup>th</sup> order in space:
    - 2 kernels, only one uses shared memory
    - 1.14x full code speedup (1.18x kernel speedup)
  - 16<sup>th</sup> order in space:
    - 3 kernels, only one uses shared memory
    - 1.20x full code speedup (1.29x kernel speedup)

# Shared Memory Bank Conflicts

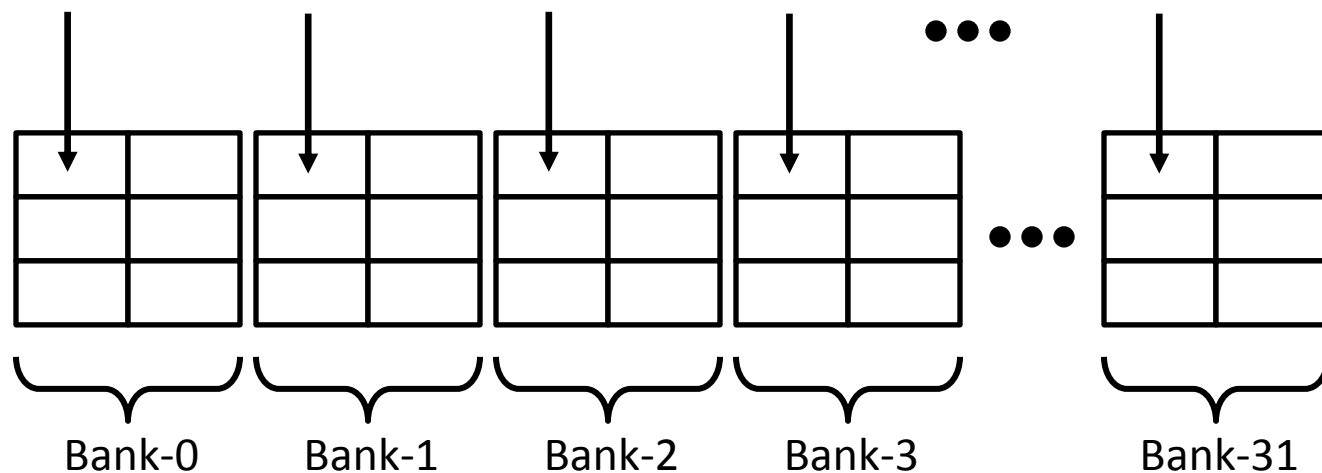
- **A bank conflict occurs when:**
  - 2 or more threads in a warp access different words in the same bank
    - Think: 2 or more threads access different “rows” in the same bank
  - N-way bank conflict: N threads in a warp conflict
    - Instruction gets issued N times: increases latency
- **Note there is no bank conflict if:**
  - Several threads access the same word
  - Several threads access different bytes of the same word



# SMEM Access Examples

## Addresses from a warp: no bank conflicts

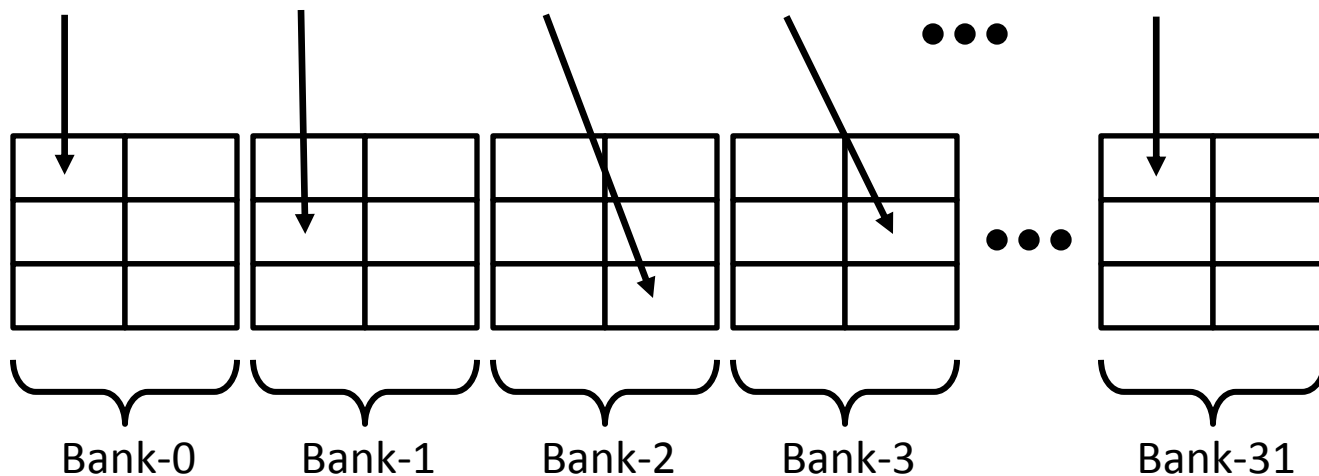
One address access per bank



# SMEM Access Examples

## Addresses from a warp: no bank conflicts

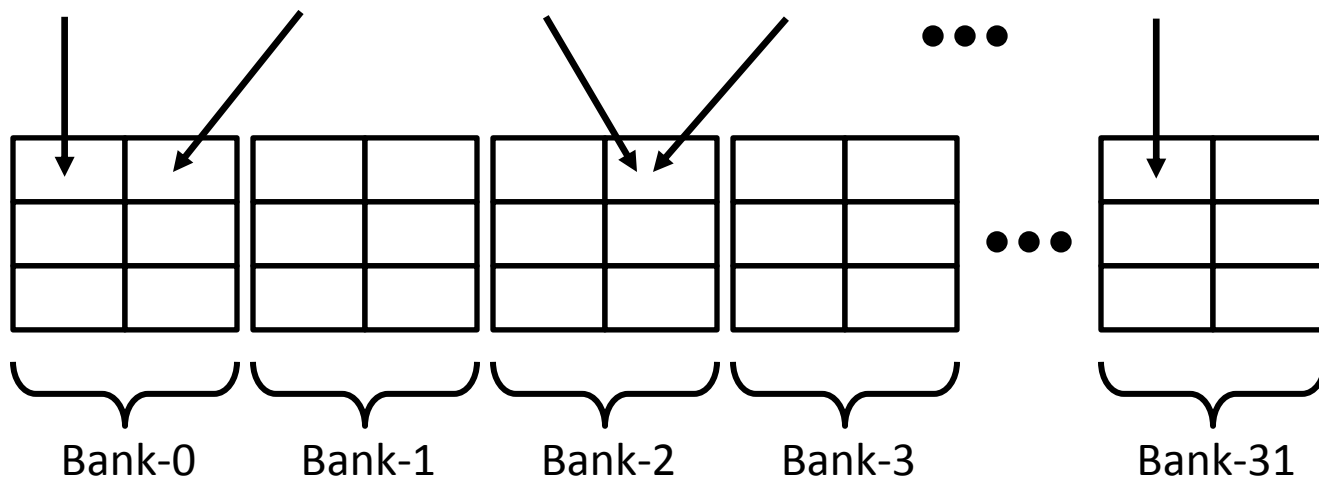
One address access per bank



# SMEM Access Examples

## Addresses from a warp: no bank conflicts

Multiple addresses per bank, but within the same word

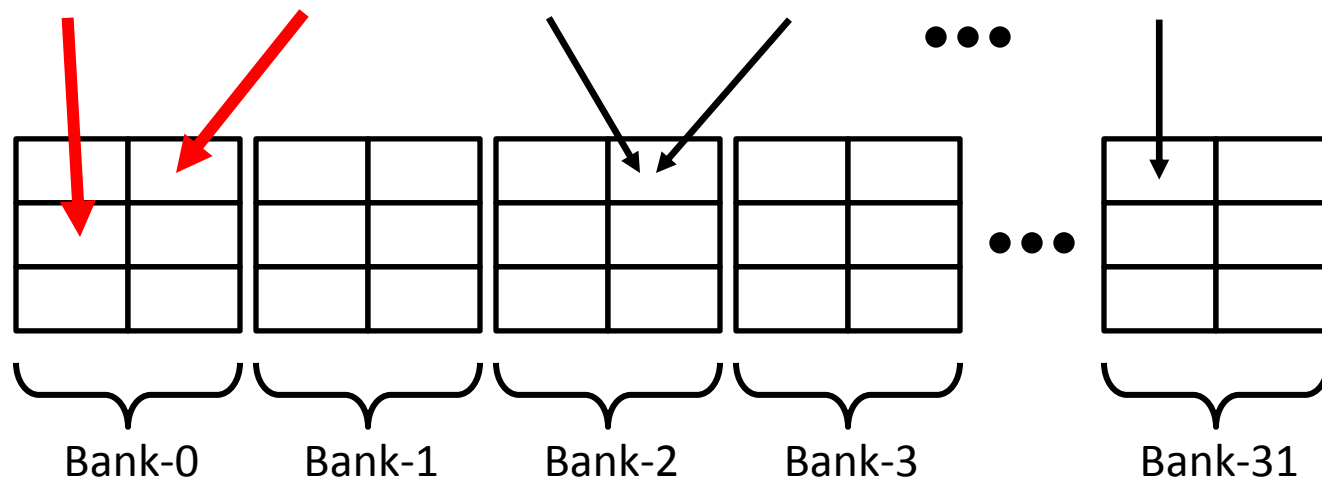




# SMEM Access Examples

## Addresses from a warp: 2-way bank conflict

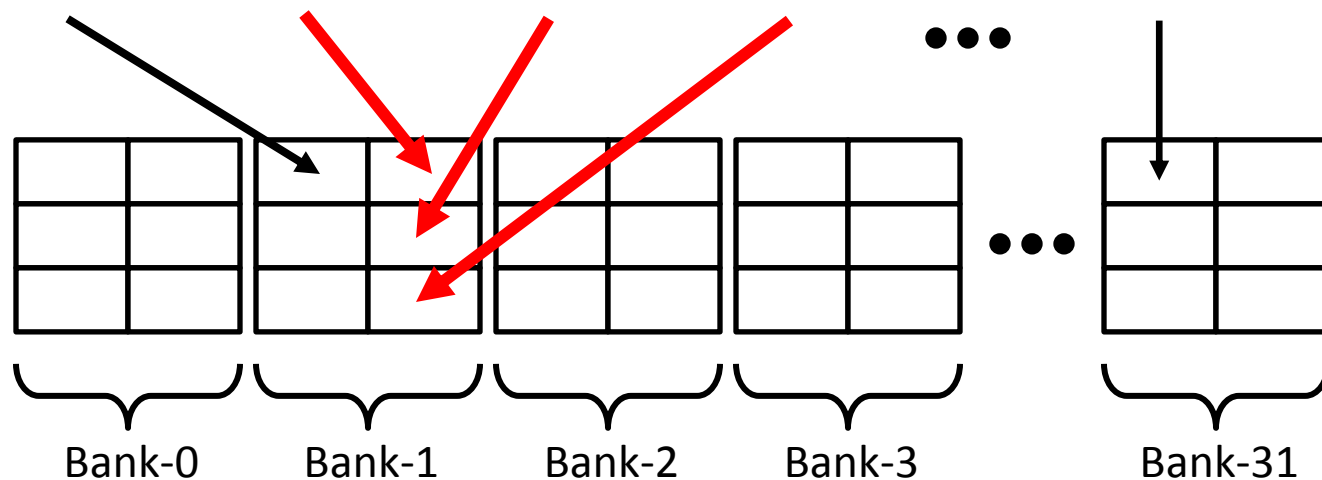
2 accesses per bank, fall in two different words



# SMEM Access Examples

## Addresses from a warp: 3-way bank conflict

4 accesses per bank, fall in 3 different words



# Diagnosing Bank Conflicts

- **Profiler counters:**
  - Number of instructions executed, does not include replays:
    - `shared_load`, `shared_store`
  - Number of replays (number of instruction issues due to bank conflicts)
    - `l1_shared_bank_conflict`
- **Analysis:**
  - Number of replays per instruction
    - $\text{l1\_shared\_bank\_conflict} / (\text{shared\_load} + \text{shared\_store})$
  - Replays are potentially a concern because:
    - Replays add latency
    - Compete for issue cycles with other SMEM and GMEM operations
      - Except for read-only loads, which go to different hardware
- **Remedy:**
  - Usually padding SMEM data structures resolves/reduces bank conflicts

# Case Study 7: Matrix Transpose

- **Staged via SMEM to coalesce GMEM addresses**
  - 32x32 threadblock, double-precision values
  - 32x32 array in shared memory
- **Initial implementation:**
  - A warp writes a row of values to SMEM (read from GMEM)
  - A warp reads a column of values from SMEM (to be written to GMEM)
- **Diagnosing:**
  - 15 replays per shared memory instruction
  - Replays make up 56% of instructions issued
    - Ratio of l1\_shared\_bank\_conflict to inst\_issued
  - Code achieves only 45% of DRAM bandwidth
  - Conclusion: bank conflicts add latency and prevent GMEM instructions from executing efficiently

# Cast Study 7: Remedy and Results

- **Remedy:**
  - Simply pad each row of SMEM array with an extra element
    - 32x33 array, as opposed to 32x32
    - Effort: 1 character, literally
  - Warp access to SMEM
    - Writes still have no bank conflicts:
      - threads access successive elements
    - Reads also have no bank conflicts:
      - Stride between threads is 17 8-byte words, thus each goes to a different bank
- **Results:**
  - Initial: 22.6 ms (worse than naïve with scattered GMEM access)
  - Optimized: 11.2 ms (~2x speedup)
    - 0 bank conflicts, 65% of DRAM theory

# Summary: Shared Memory

- **Shared memory is a tremendous resource**
  - Very high bandwidth (terabytes per second)
  - 20-30x lower latency than accessing GMEM
  - Data is programmer-managed, no evictions by hardware
- **Performance issues to look out for:**
  - Bank conflicts add latency and reduce throughput
  - Many-way bank conflicts can be very expensive
    - Replay latency adds up
    - However, few code patterns have high conflicts, padding is a very simple and effective solution
- **Kepler has 2x SMEM throughput compared to Fermi:**
  - SMEM throughput is doubled by increasing bank width to 8 bytes
  - Kernels with 8-byte words will benefit without changing kernel code
    - Put GPU into 8-byte bank mode with `cudaSetSharedMemConfig()` call
  - Kernels with smaller words will benefit if words are grouped into 8-byte structures



# ARITHMETIC OPTIMIZATIONS



# Execution

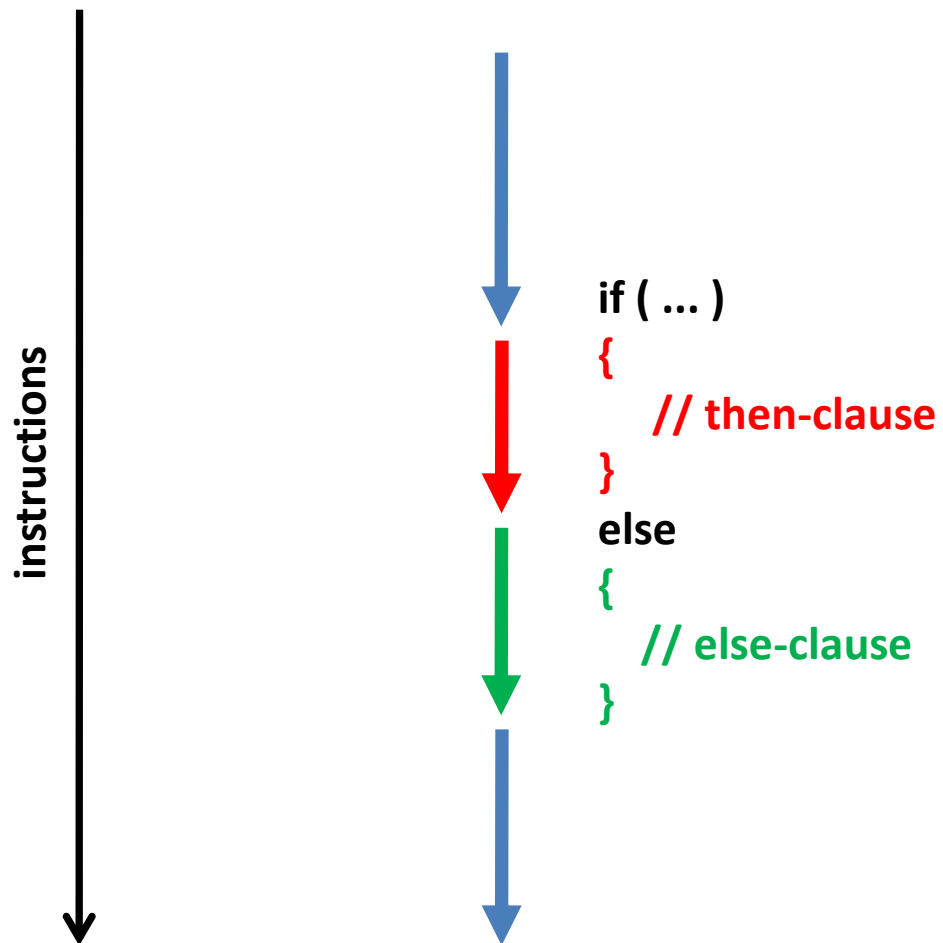
- **Instructions are issued/executed per warp**
  - Warp = 32 consecutive threads
    - Think of it as a “vector” of 32 threads
    - The same instruction is issued to the entire warp
- **Scheduling**
  - Warps are scheduled at run-time
  - Hardware picks from warps that have an instruction ready to execute
    - Ready = all arguments are ready
  - Instruction latency is hidden by executing other warps

# Control Flow

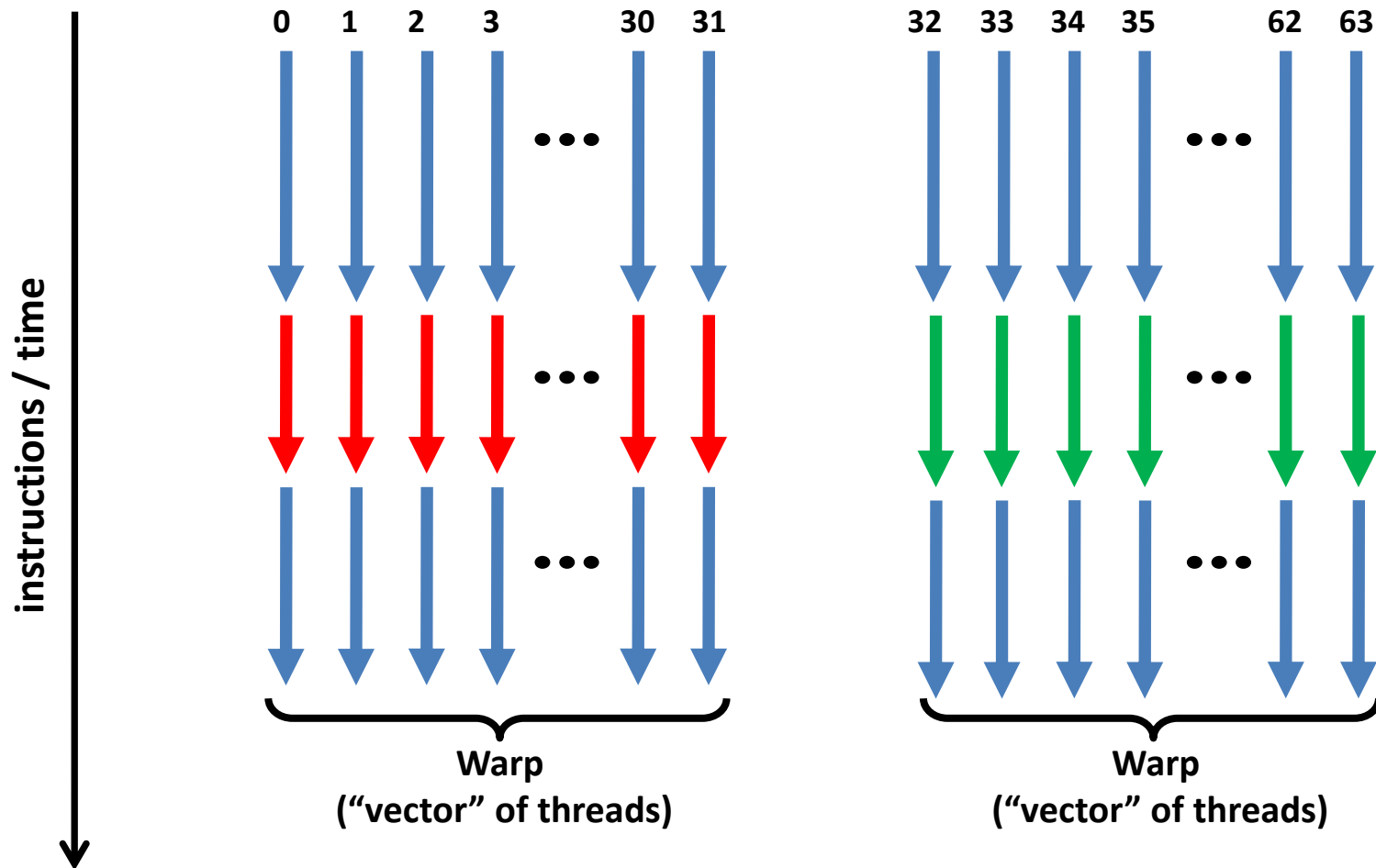
- **Single-Instruction Multiple-Threads (SIMT) model**
  - A single instruction is issued for a warp (thread-vector) at a time
- **SIMT compared to SIMD:**
  - SIMD requires vector code in each thread
  - SIMT allows you to write scalar code per thread
    - Vectorization is handled by hardware
- **Note:**
  - All contemporary processors (CPUs and GPUs) are built by aggregating vector processing units
  - Vectorization is needed to get performance on CPUs and GPUs



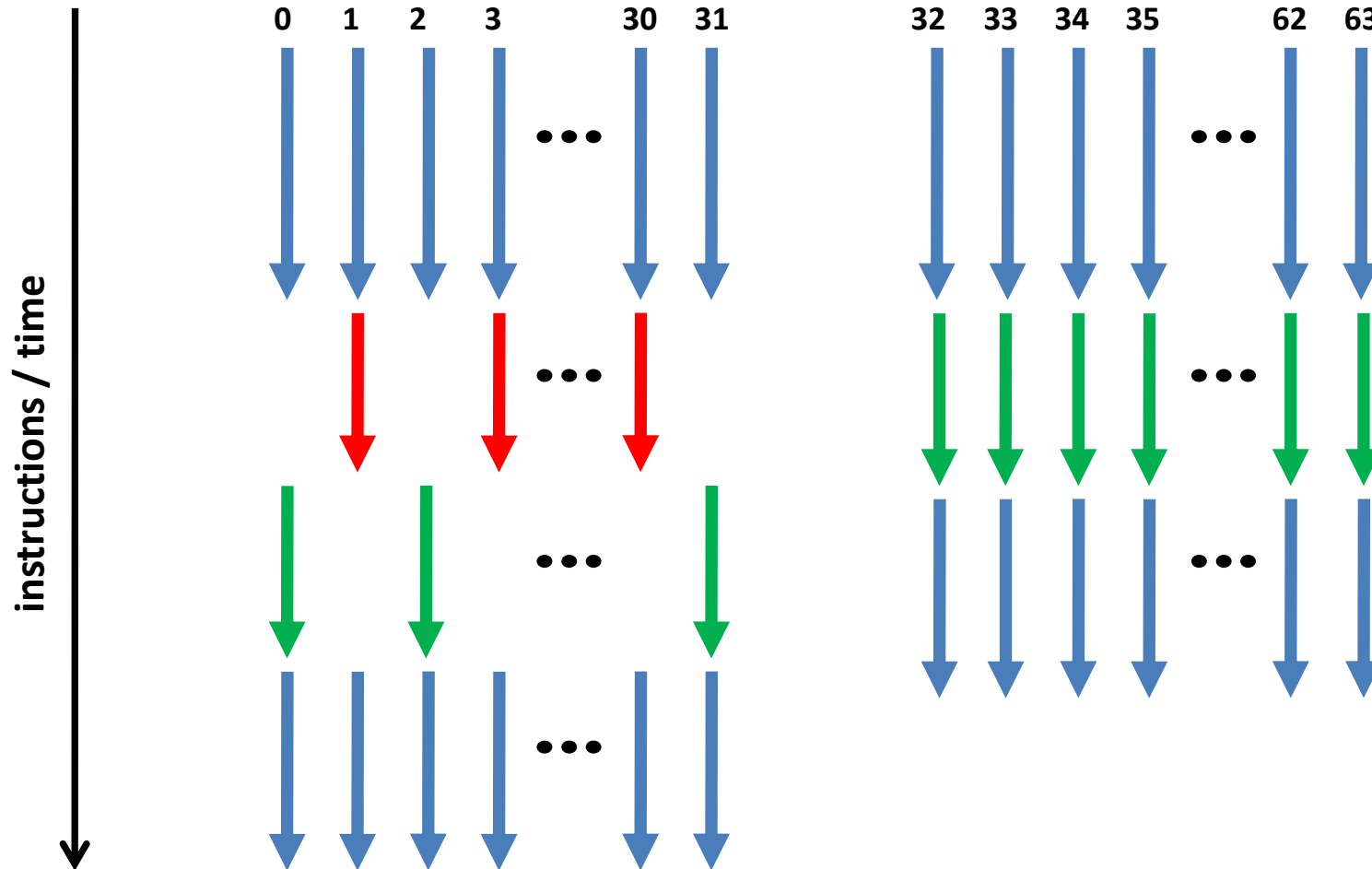
# Control Flow



# Execution within warps is coherent



# Execution diverges within a warp



# Possible Performance Limiting Factors

- **Raw instruction throughput**
  - Know the kernel instruction mix
  - fp32, fp64, int, mem, transcendentals, etc. have different throughputs
    - Refer to the CUDA Programming Guide / Best Practices Guide
    - Can examine assembly: use [cuobjdump](#) tool provided with CUDA toolkit
  - A lot of divergence can “waste” instructions
- **Instruction serialization**
  - Occurs when threads in a warp issue the same instruction in sequence
    - As opposed to the entire warp issuing the instruction at once
    - Think of it as “replaying” the same instruction for different threads in a warp
  - Mostly:
    - Shared memory bank conflicts
    - Memory accesses that result in multiple transactions (scattered address patterns)

# Instruction Throughput: Analysis

- **Compare achieved instruction throughput to HW capabilities**
  - Profiler reports achieved throughput as IPC (instructions per clock)
  - Peak instruction throughput is documented in the Programming Guide
    - Profiler also provides peak fp32 throughput for reference (doesn't take your instruction mix into consideration)
- **Check for serialization**
  - Number of replays due to serialization: `instructions_issued` - `instructions_executed`
  - Profiler reports:
    - **% of serialization** metric (ratio of replays to instructions issued)
    - Kepler: counts replays due to various memory access instructions
  - A concern if: code is instruction or latency-limited, replay percentage is high
- **Warp divergence**
  - Profiler counters: `divergent_branch`, `branch`
  - Compare the two to see what percentage diverges
    - However, this only counts the branches, not the rest of serialized instructions



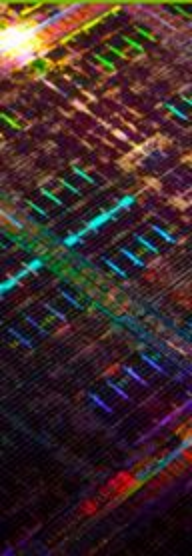


# Instruction Throughput: Optimization

- **Use intrinsics where possible ( `__sin()`, `__sincos()`, `__exp()`, etc.)**
  - Available for a number of math.h functions
  - 2-3 bits lower precision, much higher throughput
    - Refer to the CUDA Programming Guide for details
  - Often a single HW instruction, whereas a non-intrinsic is a SW sequence
- **Additional compiler flags that also help performance:**
  - `-ftz=true` : flush denormals to 0
  - `-prec-div=false` : faster fp division instruction sequence (some precision loss)
  - `-prec-sqrt=false` : faster fp sqrt instruction sequence (some precision loss)
- **Make sure you do fp64 arithmetic only where you mean it:**
  - fp64 throughput is lower than fp32
  - fp literals without an “f” suffix ( `34.7` ) are interpreted as fp64 per C standard

# Instruction Throughput: Summary

- **Analyze:**
  - Check achieved instruction throughput
  - Compare to HW peak (but keep instruction mix in mind)
  - Check percentage of instructions due to serialization
- **Optimizations:**
  - Intrinsics, compiler options for expensive operations
  - Group threads that are likely to follow same execution path (minimize warp divergence)
  - Minimize memory access replays (SMEM and GMEM)





# OPTIMIZING FOR KEPLER

# Kepler Architecture Family

- **Two architectures in the family:**
  - GK104 (Tesla K10, GeForce: GTX690, GTX680, GTX670, ...)
    - Note that K10 is 2 GK104 chips on a single board
  - GK110 (Tesla K20, ...)
- **GK110 has a number of features not in GK104:**
  - Dynamic parallelism, HyperQ
  - More registers per thread, more fp64 throughput
  - For full details refer to:
    - Kepler Whitepaper
    - GTC12 Session 0642: “Inside Kepler”

# Good News About Kepler Optimization

- **The same optimization fundamentals that applied to Fermi, apply to Kepler**
  - There are no new fundamentals
- **Main optimization considerations:**
  - Expose sufficient parallelism
    - SM is more powerful, so will need more work
  - Coalesce memory access
    - Exactly the same as on Fermi
  - Have coherent control flow within warps
    - Exactly the same as on Fermi

# Level of Parallelism

- **Parallelism for memory is most important**
  - Most codes don't achieve peak fp throughput because:
    - Stalls waiting on memory (latency not completely hidden)
    - Execution of non-fp instructions (indexing, control-flow, etc.)
    - NOT because of lack of independent fp math
- **GK104:**
  - Compared to Fermi, needs  $\sim 2x$  concurrent accesses per SM to saturate memory bandwidth
    - Memory bandwidth comparable to Fermi
    - 8 SMs while Fermi had 16 SMs
  - Doesn't necessarily need twice the occupancy of your Fermi code
    - If Fermi code exposed more than sufficient parallelism, increase is less than 2x

# Kepler SM Improvements for Occupancy

- **2x registers**
  - Both GK104 and GK110
  - 64K registers (Fermi had 32K)
  - Code where occupancy is limited by registers will readily achieve higher occupancy (run more concurrent warps)
- **2x threadblocks**
  - Both GK104 and GK110
  - Up to 16 threadblocks (Fermi had 8)
- **1.33x more threads**
  - Both GK104 and GK110
  - Up to 2048 threads (Fermi had 1536)



# Increased Shared Memory Bandwidth

- **Both GK104 and GK110**
- **To benefit, code must access 8-byte words**
  - No changes for double-precision codes
  - Single-precision or integer codes should group accesses into `float2`, `int2` structures to get the benefit
- **Refer to Case Study 6 for a usecase sample**

# SM Improvements Specific to GK110

- **More registers per thread**
  - A thread can use up to 255 registers (Fermi had 63)
  - Improves performance for some codes that spilled a lot of registers on Fermi (or GK104)
    - Note that more registers per thread still has to be weighed against lower occupancy
- **Ability to use read-only cache for accessing global memory**
  - Improves performance for some codes with scattered access patterns, lowers the overhead due to replays
- **Warp-shuffle instruction (tool for ninjas)**
  - Enables threads in the same warp to exchange values without going through shared memory

# Case Study 8: More Registers Per Thread

- **TTI RTM code:**
  - Same as used in Case Study 6
  - Can be implemented in 2 or 3 passes
    - <http://hpcoilgas.citris-uc.org/stencil-computation-gpu-seismic-migration-isotropic-vti-and-tti-rtm-kernels>
    - 2-pass approach has fewer accesses to memory, but consumes many registers
      - Additional benefit: requires less storage than 3-pass
    - 3-pass approach has more accesses to memory, but consumes fewer registers
    - Higher order in space -> more registers needed
- **16<sup>th</sup> order in space:**
  - Fermi: 3-pass is faster than 2-pass
    - 2-pass spills too many registers, which causes extra memory traffic
  - GK110: 2-pass is **1.15x** faster than 3-pass
    - The “large” kernel consumes 96 registers per thread, doesn’t spill
    - Can probably be improved further: literally 5 minutes were spent on optimization

# Considerations for Dynamic Parallelism

- **GPU threads are able to launch work for GPU**
  - GK110-specific feature
- **Same considerations as for launches from CPU**
  - Same exact considerations for exposing sufficient parallelism as for “traditional” launches (CPU launches work for GPU)
  - A single launch doesn’t have to saturate the GPU:
    - GPU can execute up to 32 different kernel launches concurrently

# In Conclusion

- **When programming and optimizing think about:**
  - Exposing sufficient parallelism
  - Coalescing memory accesses
  - Having coherent control flow within warps
- **Use profiling tools when analyzing performance**
  - Determine performance limiters first
  - Diagnose memory access patterns

# Questions

