Fundamental Optimizations
Global Memory

Stanford 2011

Steve Rennich, NVIDIA Corporation

NVIDIA

Schedule

- Introduction and CUDA C Basics 1
  - Memory allocation/copy, programming model, kernel code, kernel configuration, error-reporting, coordinating CPU and GPU execution
- CUDA C Basics 2
  - GPU memory system, shared memory basics, relationship between the hardware and programming models, requirements for performance.
- Fundamental Optimizations 1 – Global Memory
  - Address pattern and performance, number of accesses and performance, caching, using the profiler
- Fundamental Optimizations 2 – Shared Memory
  - Shared memory banking, shared memory multicast, using the profiler
- Finite Difference Stencils on Regular Grids
- Fundamental Optimizations 3 – Warp Execution, Synchronizing CPU and GPU
  - Control flow divergence, intrinsic functions, Events, streams, and call synchronicity
- Determining Kernel Performance Limiters
  - Description of the three limiting factors (memory, arithmetic, latency) and how to use the profiler and code modifications to investigate them

Outline

- Launch configuration
- Global memory throughput
- Using the profiler

Launch Configuration

Launch Configuration

- **How many threads/threadblocks to launch?**
- **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
- **Conclusion:**
  - Need enough threads to hide latency

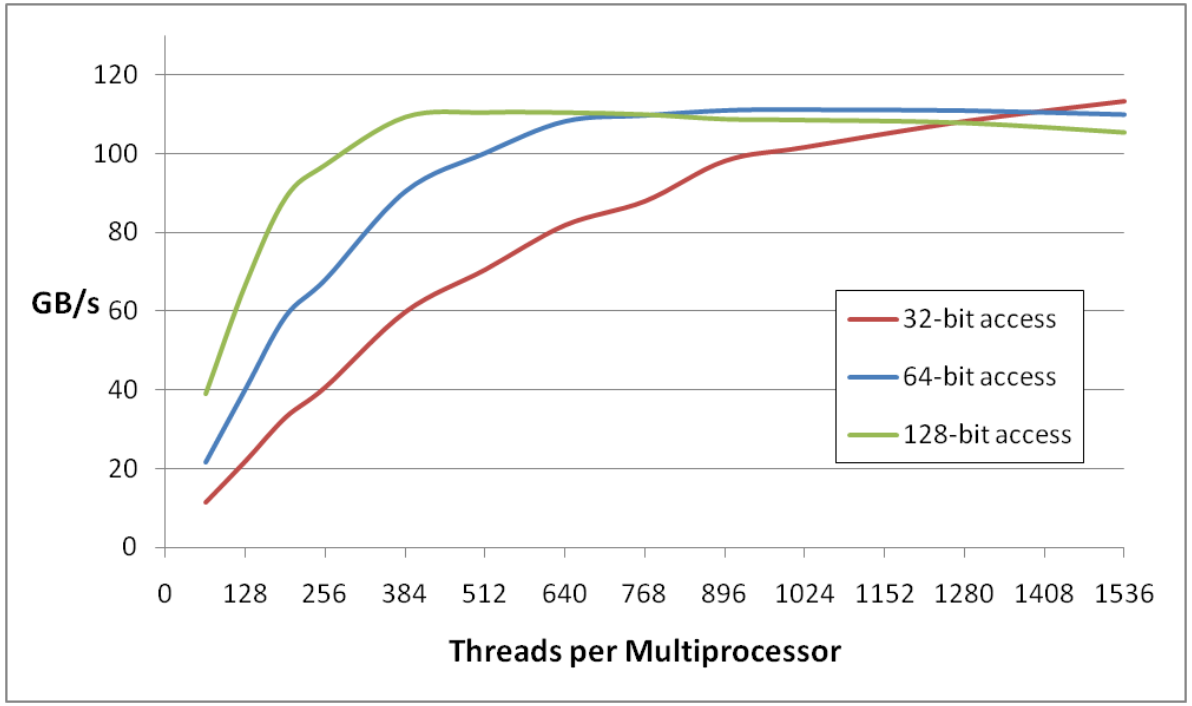
Launch Configuration

- **Hiding arithmetic latency:**
  - Need ~18 warps (576) threads per Fermi 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:

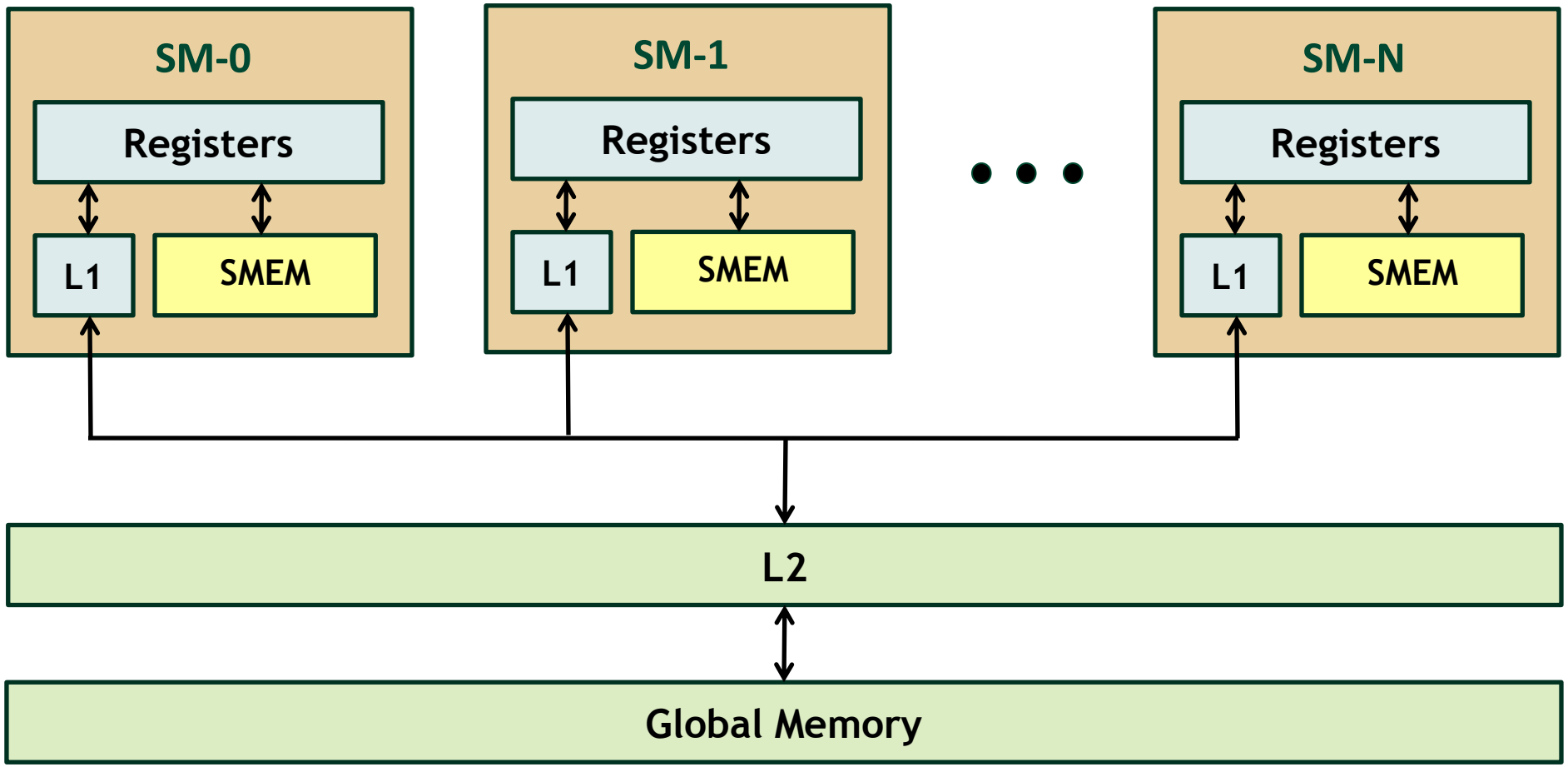
Four 32-bit ~= one 128-bit

Launch Configuration: Summary

- **Need enough total threads to keep GPU busy**
  - Typically, you'd like 512+ threads per SM
    - More if processing 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 threadblocks
    - Really small threadblocks prevent achieving good occupancy
    - Really large threadblocks are less flexible
    - I generally use 128-256 threads/block, but use whatever is best for the application

Global Memory Throughput

Fermi Memory Hierarchy Review



Fermi 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)
  - Higher latency (400-800 cycles)
  - Throughput: up to 177 GB/s

Programming for L1 and L2

- **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.
  - Don't try to block for L1/L2 like you would on CPU
    - You have 100s to 1,000s of run-time scheduled threads hitting the caches
    - If it is possible to block for L1 then block for SMEM
      - Same size, same bandwidth, hw will not evict behind your back
- **Optimize as if no caches were there**
  - Some cases will just run faster

Fermi 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 Caching and L1 Size

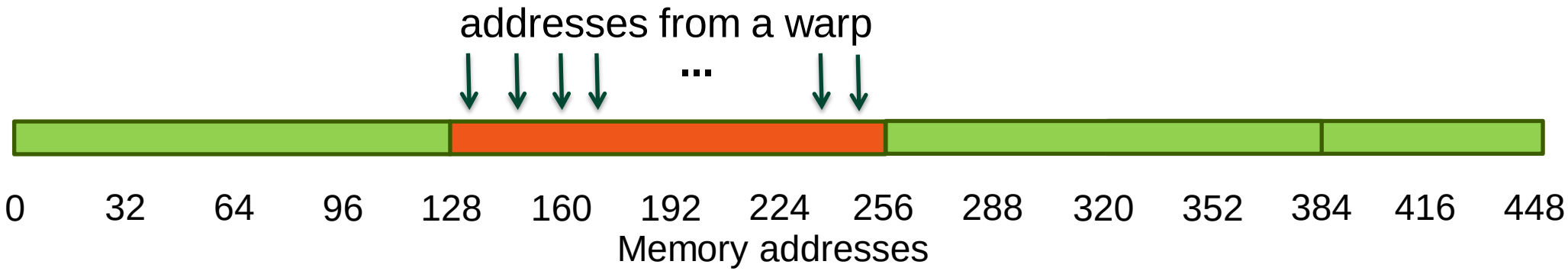
- **Non-caching loads can improve perf when:**
  - Loading scattered words or only a part of a warp issues a load
    - Benefit: transaction is smaller, so useful payload is a larger percentage
    - Loading halos, for example
  - Spilling registers (reduce line fighting with spillage)
- **Large L1 can improve perf when:**
  - Spilling registers (more lines so fewer evictions)
  - Some misaligned, strided access patterns
  - 16-KB L1 / 48-KB smem **OR** 48-KB L1 / 16-KB smem
    - CUDA call, can be set for the app or per-kernel
- **How to use:**
  - Just try a 2x2 experiment matrix: {CA,CG} x {48-L1, 16-L1}
    - Keep the best combination - same as you would with any HW managed cache, including CPUs

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

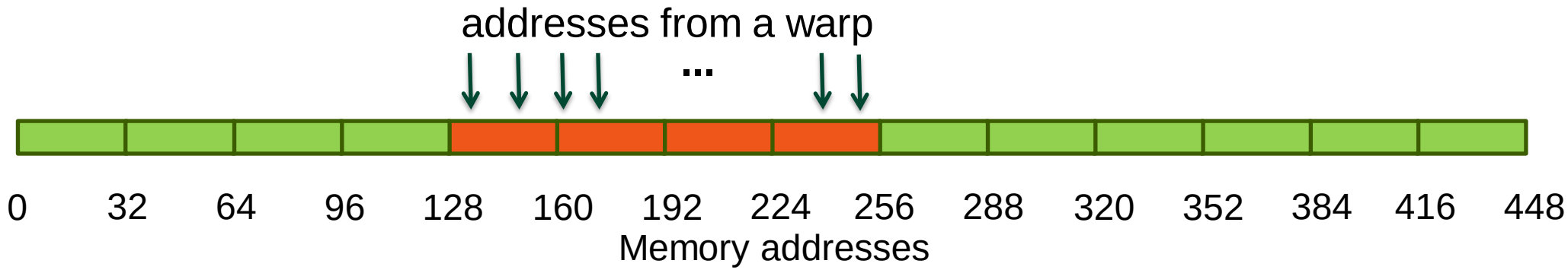
Caching Load

- **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%



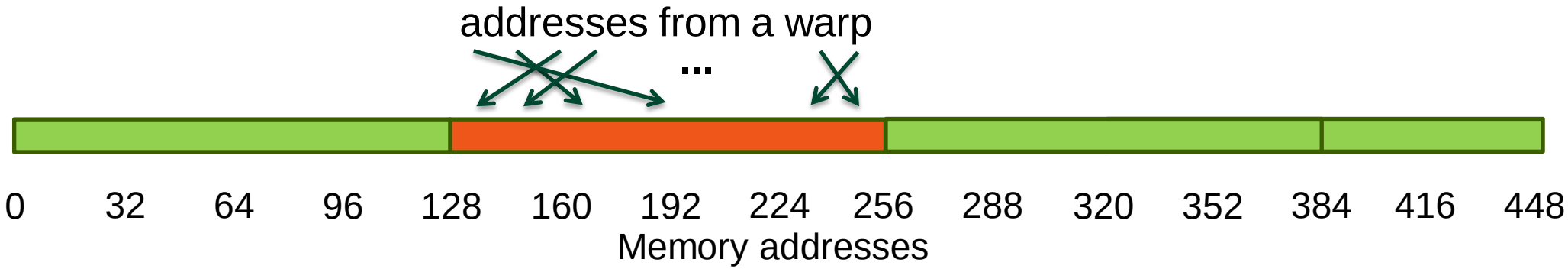
Non-caching Load

- 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%



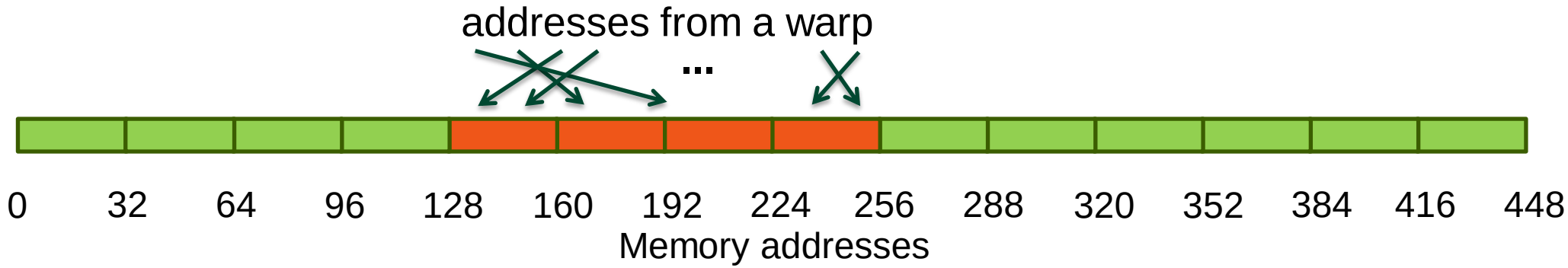
Caching Load

- **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%



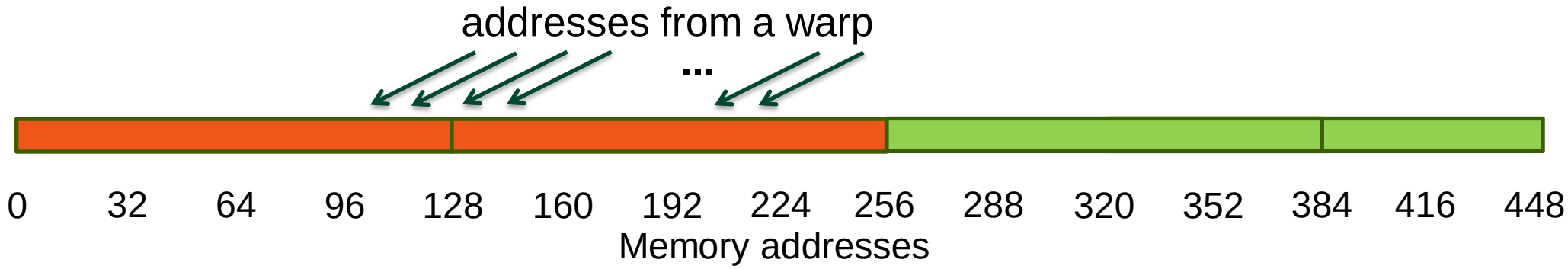
Non-caching Load

- 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%



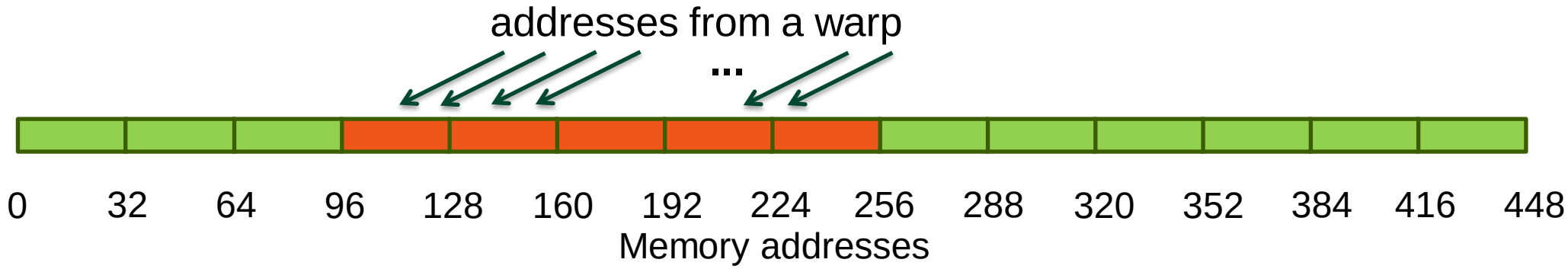
Caching Load

- **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%



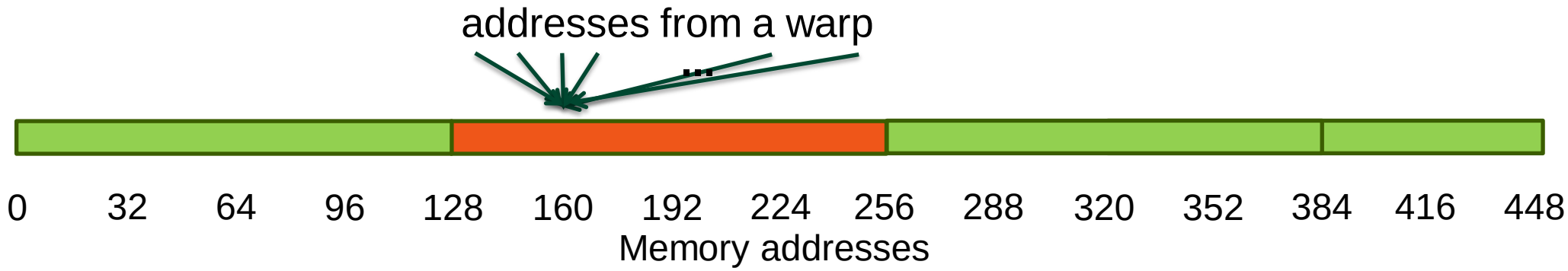
Non-caching Load

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



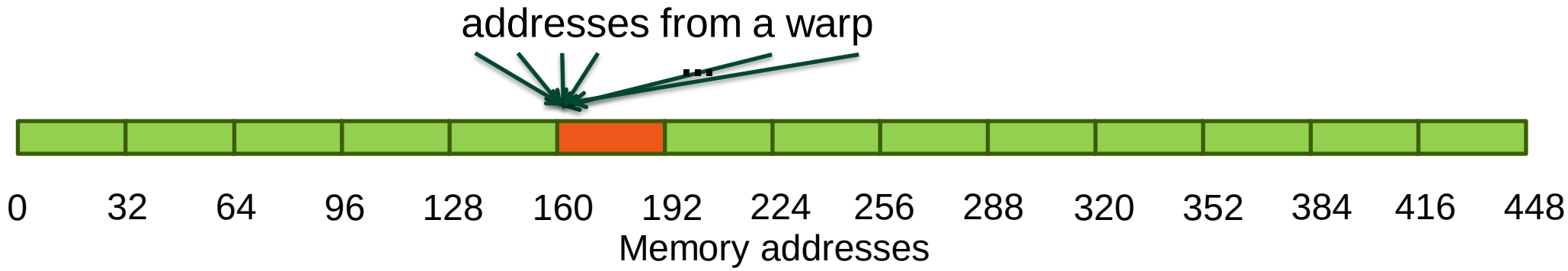
Caching Load

- 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%



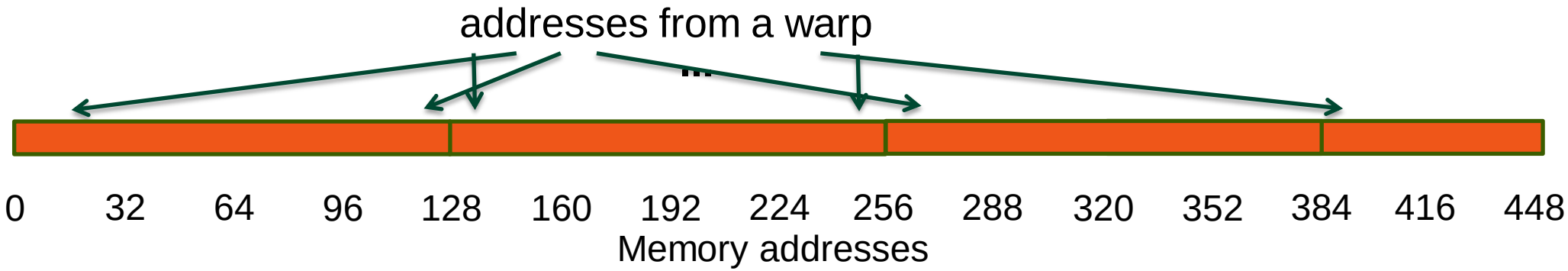
Non-caching Load

- 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%



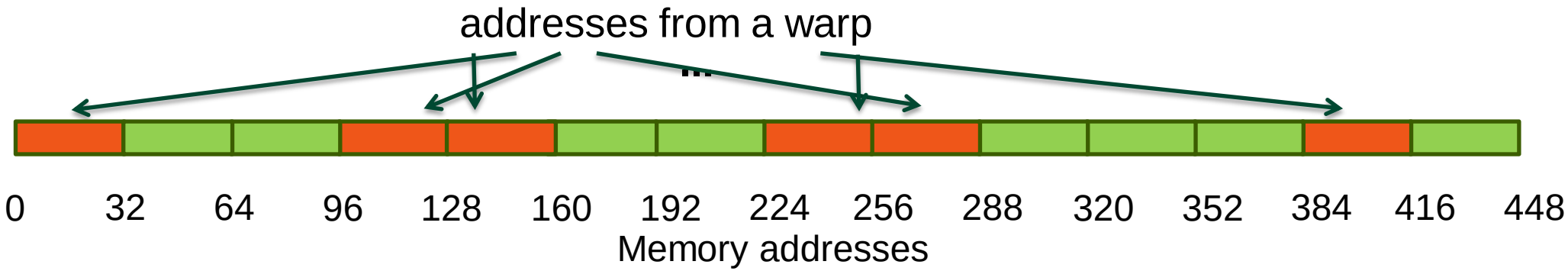
Caching Load

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



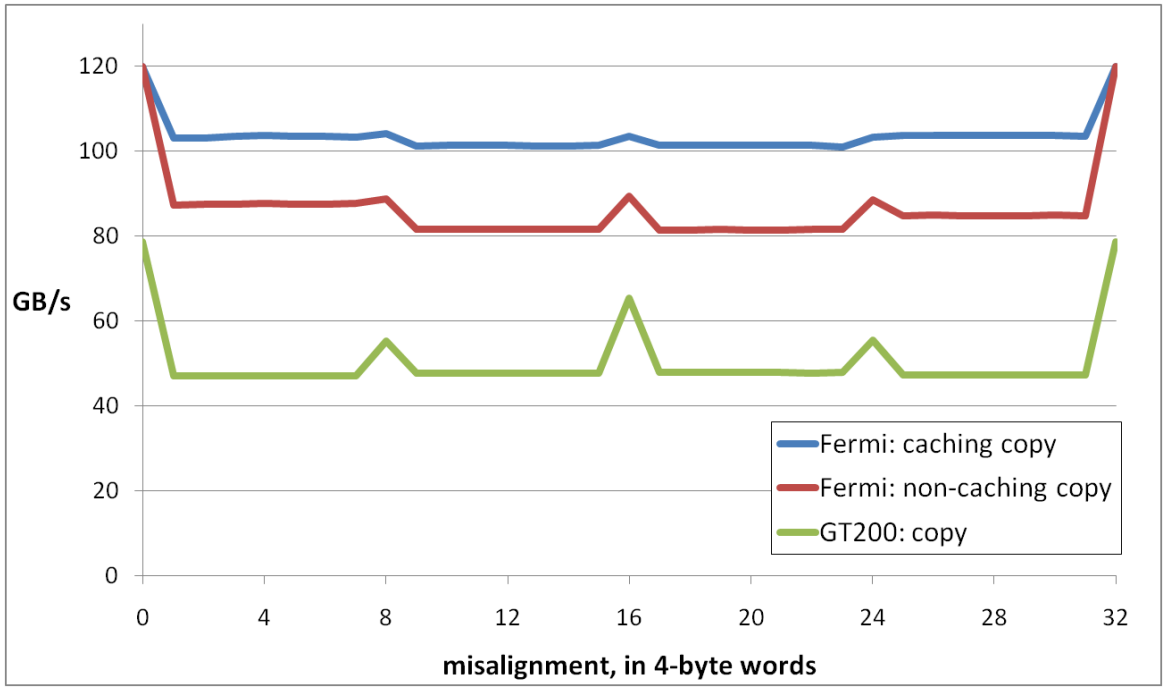
Non-caching Load

- 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)



Impact of Address Alignment

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



Experiment:

- Copy 16MB of floats
- 256 threads/block

Greatest throughput drop:

- GT200: 40%
- Fermi:
  - CA loads: 15%
  - CG loads: 32%

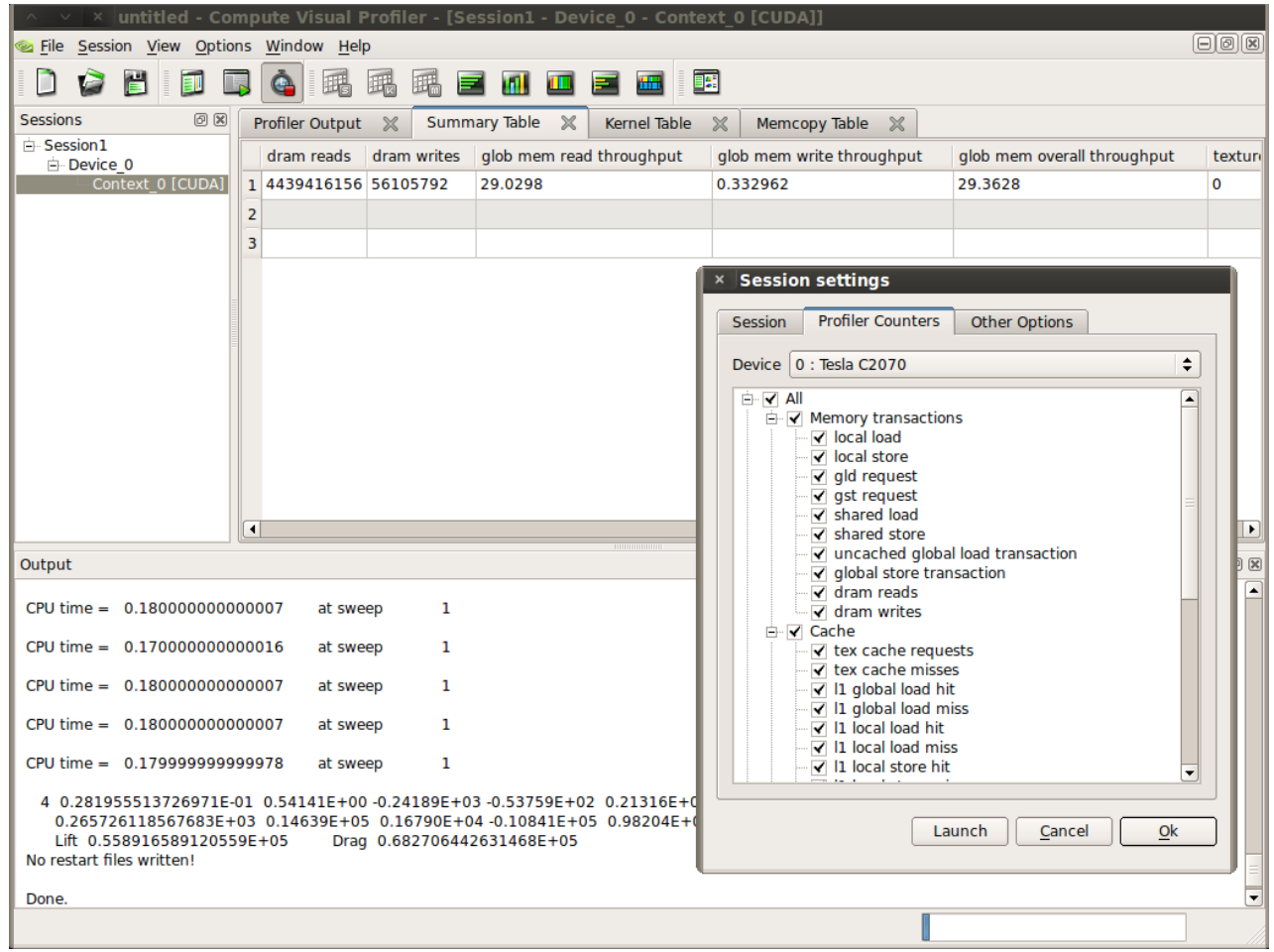
GMEM Optimization Guidelines

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

Using the Profiler

Compute Visual Profiler

- computeprof



Notes on using the profiler

- **Most counters are reported per Streaming Multiprocessor (SM)**
  - Not entire GPU
  - Exceptions: L2 and DRAM counters
- **A single run can collect a few counters**
  - Multiple runs are needed when profiling more counters
    - Done automatically by the Visual Profiler
    - Have to be done manually using command-line profiler
- **Counter values may not be exactly the same for repeated runs**
  - Threadblocks and warps are scheduled at run-time
  - So, “two counters being equal” usually means “two counters within a small delta”
- **See the profiler documentation for more information**

Questions?

Schedule

- Introduction and CUDA C Basics 1
  - Memory allocation/copy, programming model, kernel code, kernel configuration, error-reporting, coordinating CPU and GPU execution
- CUDA C Basics 2
  - GPU memory system, shared memory basics, relationship between the hardware and programming models, requirements for performance.
- Fundamental Optimizations 1 – Global Memory
  - Address pattern and performance, number of accesses and performance, caching, using the profiler
- Fundamental Optimizations 2 – Shared Memory
  - Shared memory banking, shared memory multicast, using the profiler
- Finite Difference Stencils on Regular Grids
- Fundamental Optimizations 3 – Warp Execution, Synchronizing CPU and GPU
  - Control flow divergence, intrinsic functions, Events, streams, and call synchronicity
- Determining Kernel Performance Limiters
  - Description of the three limiting factors (memory, arithmetic, latency) and how to use the profiler and code modifications to investigate them