CUDA Optimization: Memory Bandwidth Limited Kernels

CUDA Webinar

Tim C. Schroeder,
HPC Developer Technology Engineer

Outline

- We'll be focussing on optimizing global memory throughput on Fermi-class GPUs

- Launch Configuration
- Memory Access Patterns
- Using On-Chip Memory

- Summary, Further Reading and Questions

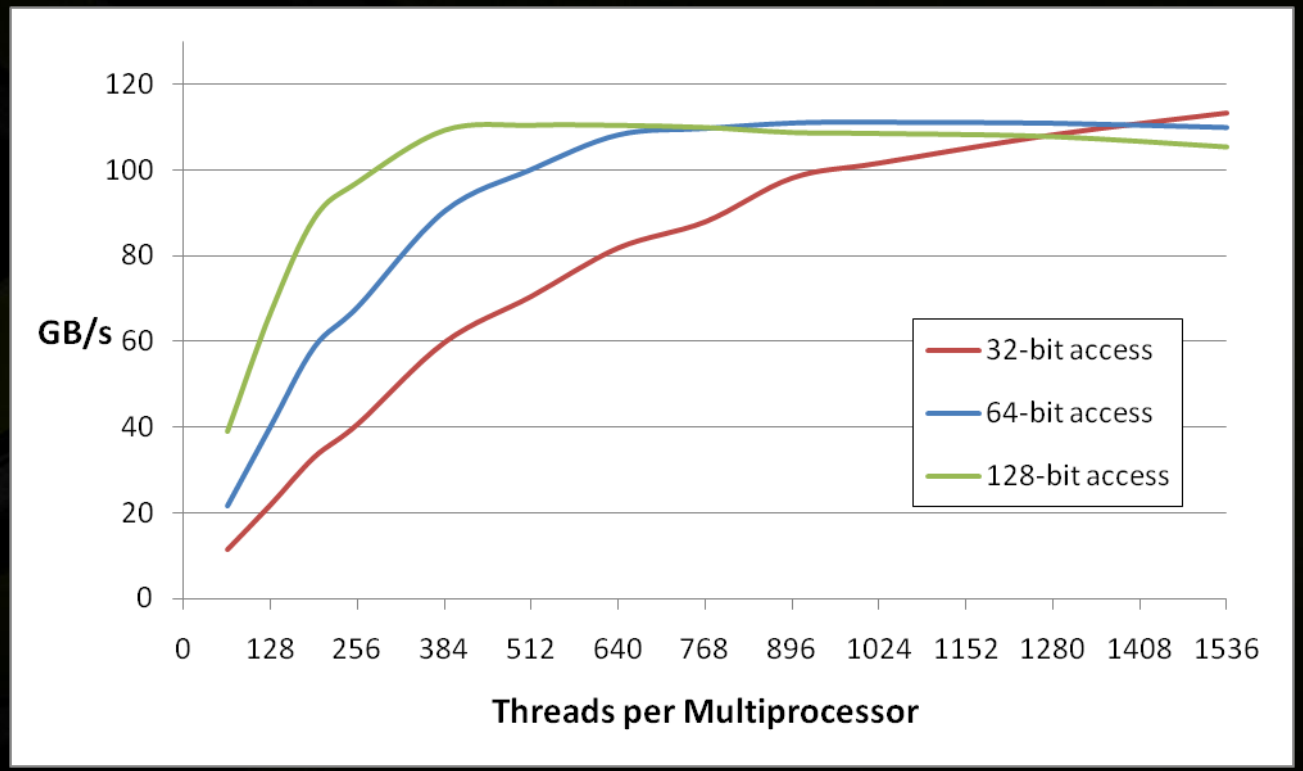
Launch Configuration

Launch Configuration

- **Need enough total threads to keep GPU busy**
  - Typically, you'd like 512+ threads per SM
    - More if processing one fp32 element per thread
    - SM can concurrently execute up to 8 threadblocks
  - Of course, exceptions exist

Example

- **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

- **Conclusion: Have enough loads in flight to saturate the bus, or handle multiple elements per thread with independent loads, same performance. Typically, you'd like 512+ threads per SM**
  - 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)
- **For more details:**
  - Vasily Volkov's GTC2010 talk "Better Performance at Lower Occupancy"

Memory Access Patterns

GMEM Operations

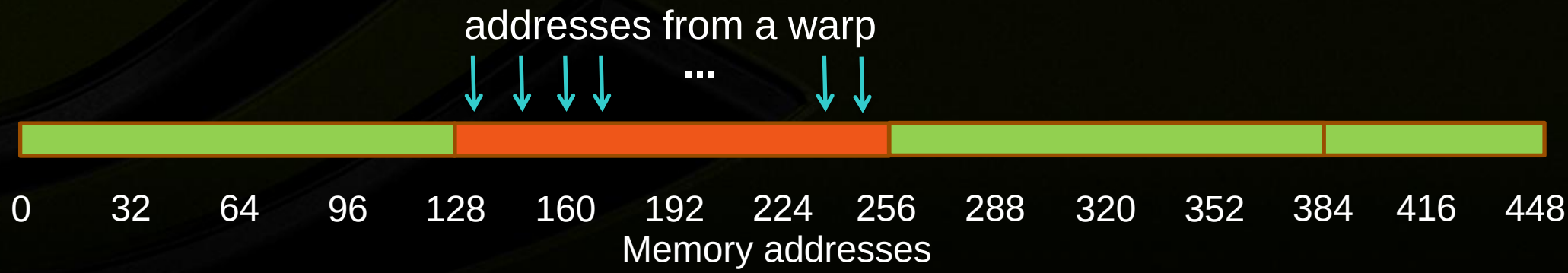
- **Two types of loads:**
  - Caching
    - Default mode
    - Attempts to hit in L1, then L2, then GMEM
    - Load granularity is 128-byte line
    - Program configurable: 16KB shared / 48 KB L1 OR 48KB shared / 16KB L1
  - Non-caching
    - Compile with *–Xptxas –dlcm=cg* option to nvcc
    - Attempts to hit in L2, then GMEM
    - Load granularity is 32-bytes

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

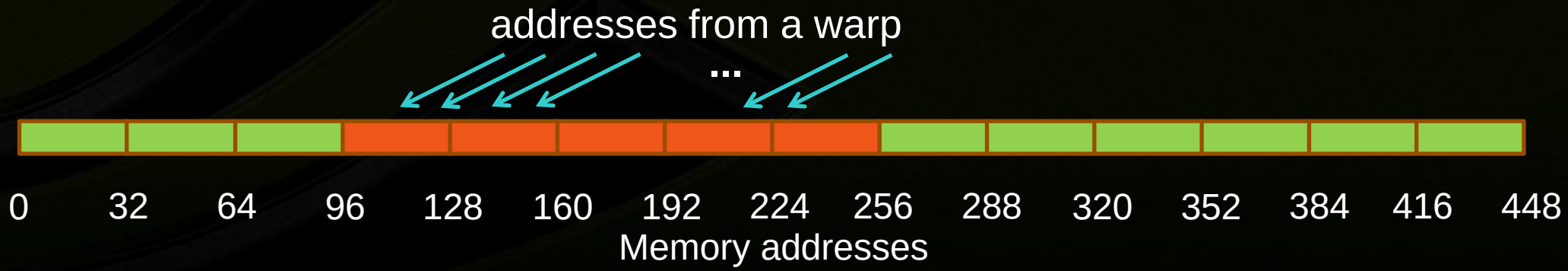
- **Warp requests 32 aligned, permuted 4-byte words**
- **Addresses fall within 4 segments**
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

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%

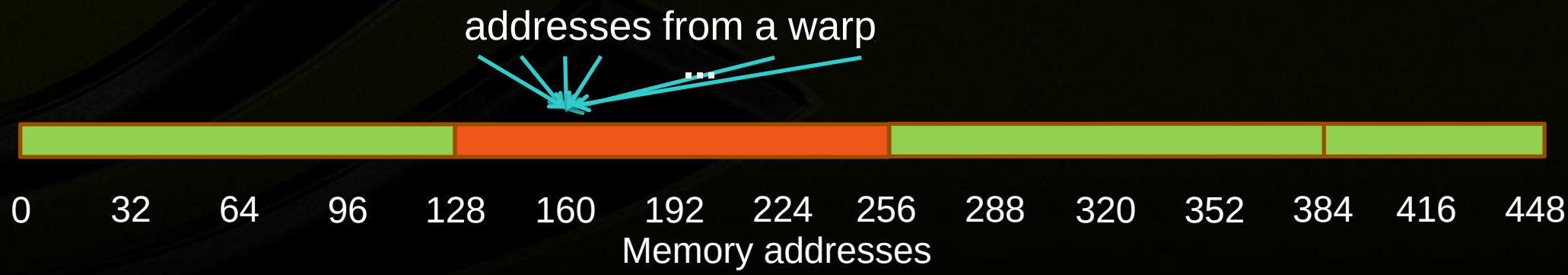
Non-caching Load

- **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%

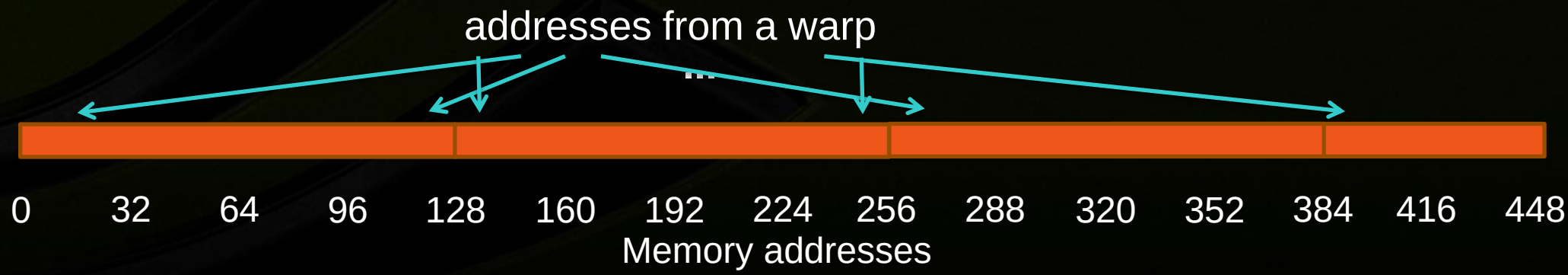


Non-caching Load

- **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%

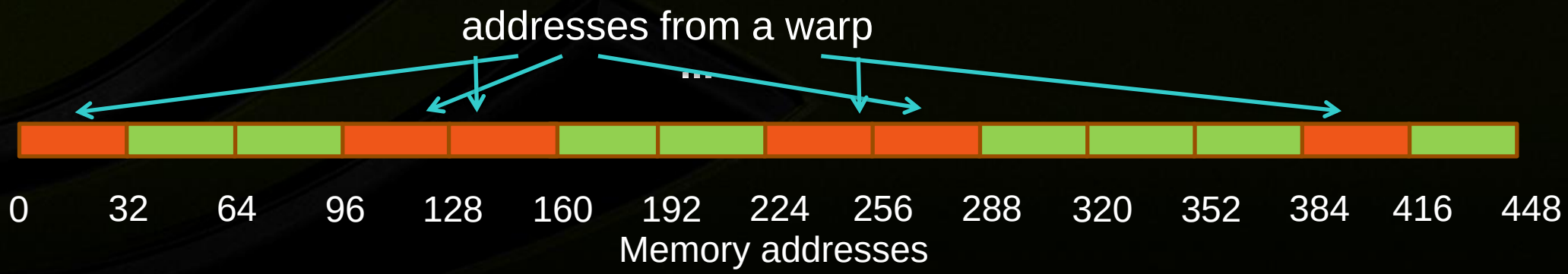
Caching Load

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



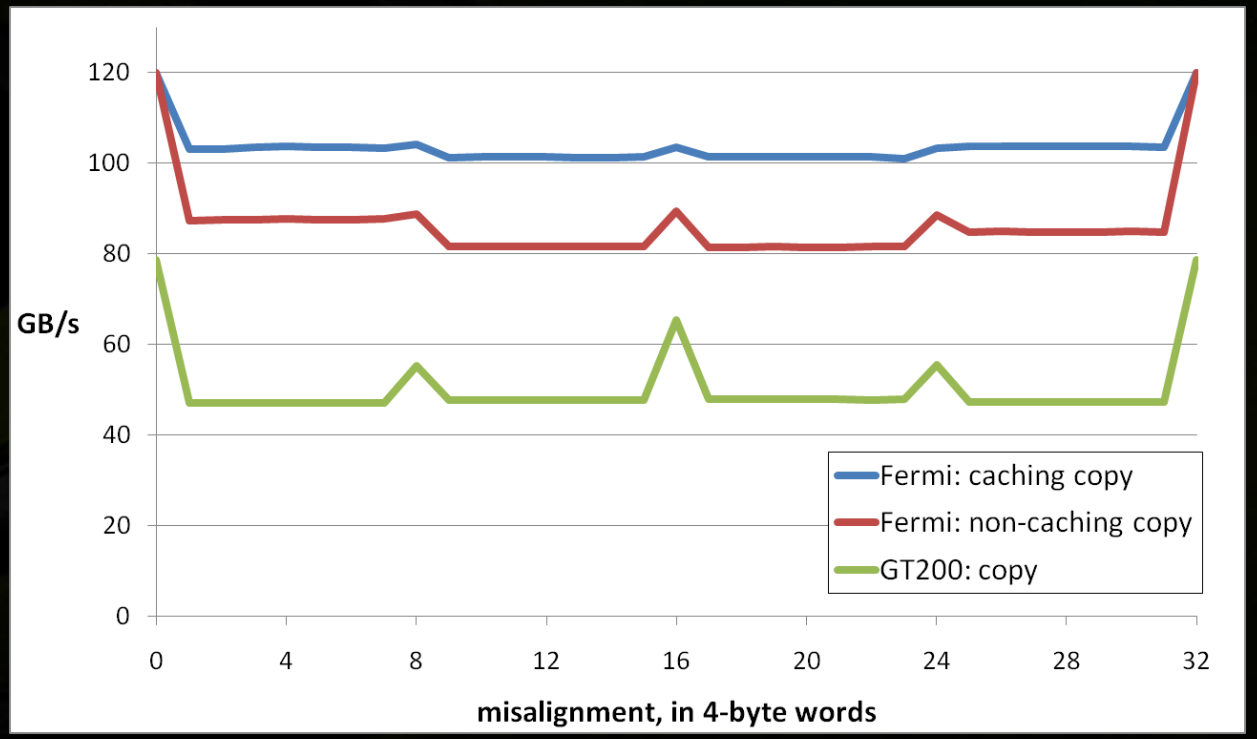
Non-caching Load

- **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**
  - 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:

- CA loads: 15%
- CG loads: 32%

Using On-Chip Memory

Shared Memory

- **Uses:**
  - Use it to improve global memory access patterns
  - Cache data to reduce redundant global memory accesses
  - Inter-thread communication within a block
- **Performance:**
  - Program configurable: 16KB shared / 48 KB L1 OR 48KB shared / 16KB L1
  - Very low latency
  - Very high throughput: 1+ TB/s aggregate

Additional “memories”

- *Texture* and *Constant*
- Read-only
- Data resides in global memory
- Read through different caches
  - Additional fast on-chip memory
  - Avoid polluting L1

Summary

Summary

- **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)
    - Sometimes using shared memory or the texture / constant cache is the best choice

Further reading

- **GTC 2010 Talks:**
  - Fundamental Performance Optimizations for GPUs
    Paulius Micikevicius
  - Analysis-Driven Optimization
    Paulius Micikevicius
  - Better Performance at Lower Occupancy
    Vasily Volkov

http://www.gputechconf.com/page/gtc-on-demand.html


Questions?
NVIDIA