Performance Optimization

Supercomputing 2011

Paulius Micikevicius| NVIDIA

November 14, 2011

Requirements for Maximum Performance

Requirements for Maximum Performance

- Have sufficient parallelism
  - At least a few 1,000 threads per function
- Coalesced memory access
  - By threads in the same “thread-vector”
- Coherent execution
  - By threads in the same “thread-vector”

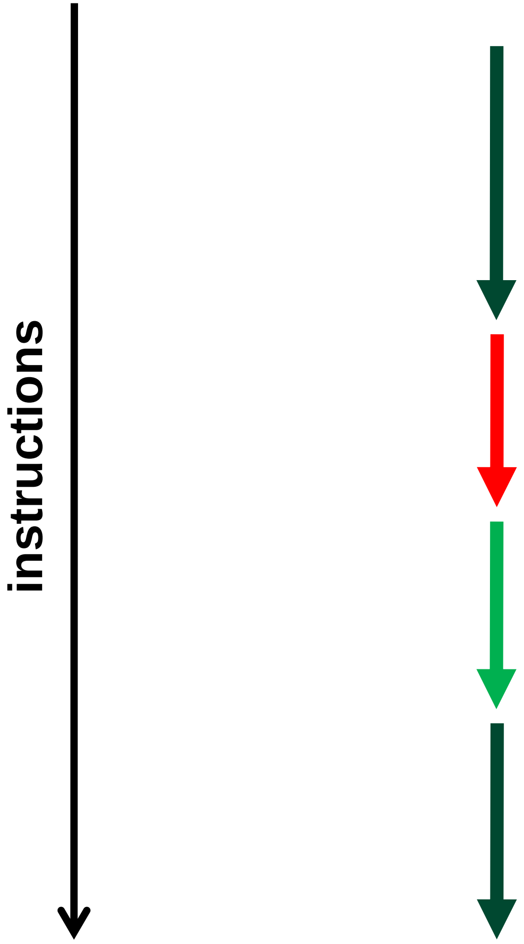
Amount of Parallelism

- GPUs issue instructions in order
  - Issue stalls when instruction arguments are not ready
- GPUs switch between threads to hide latency
  - Context switch is free: thread state is partitioned (large register file), not stored/restored
- Conclusion: need enough threads to hide math latency and to saturate the memory bus
  - Independent instructions (ILP) within a thread also help
- Very rough rule of thumb:
  - Need ~512 threads per SM
  - So, at least a few 1,000 threads per GPU

Control Flow

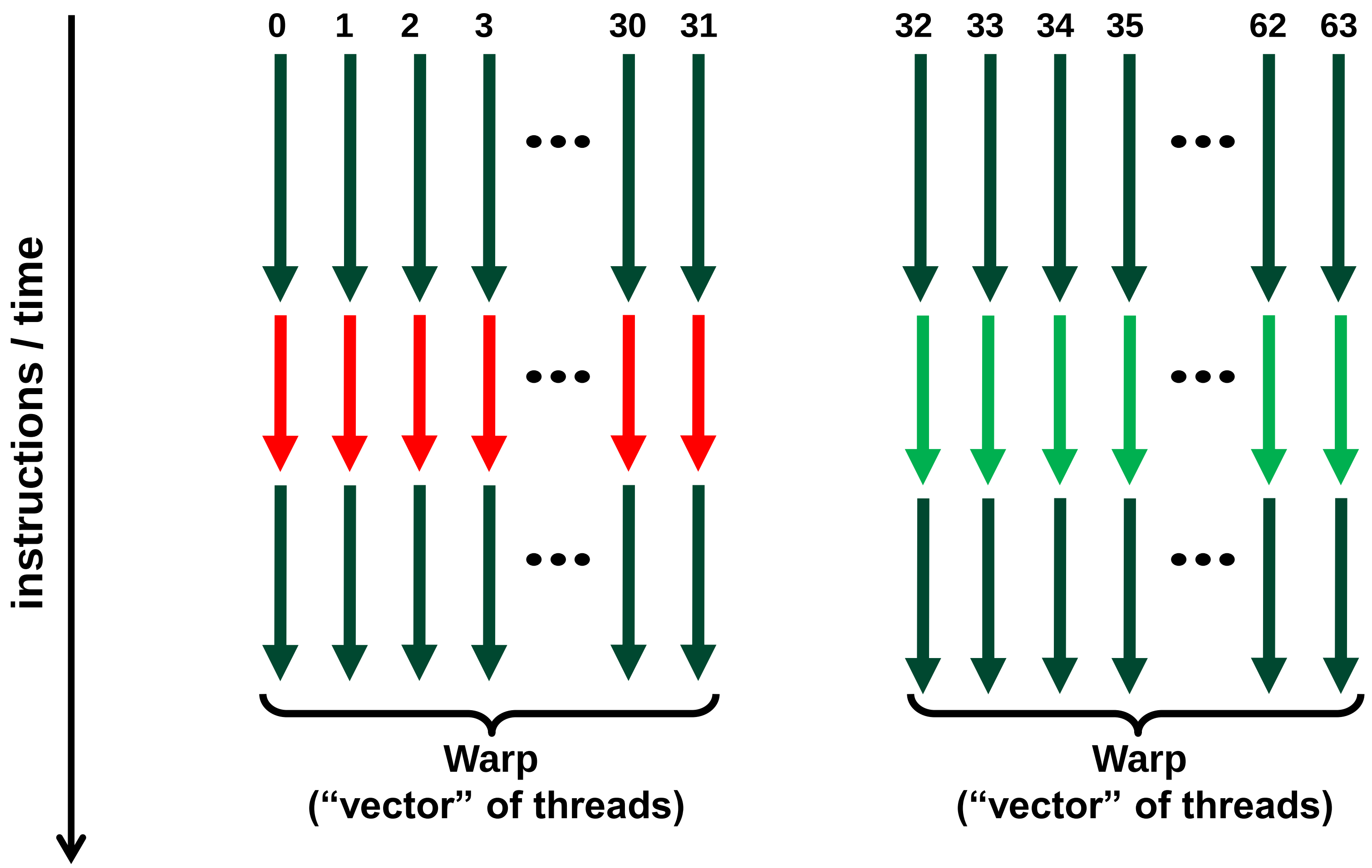
- **Single-Instruction Multiple-Threads (SIMT) model**
  - A single instruction is issued for a warp (thread-vector) at a time
  - NVIDIA GPU: warp = a vector of 32 threads
- **Compare to SIMD:**
  - SIMD requires vector code in each thread
  - SIMT allows you to write scalar code per thread
    - Vectorization is guaranteed by hardware
- **Note:**
  - All contemporary processors (CPUs and GPUs) are built by aggregating vector processing unit

Control Flow

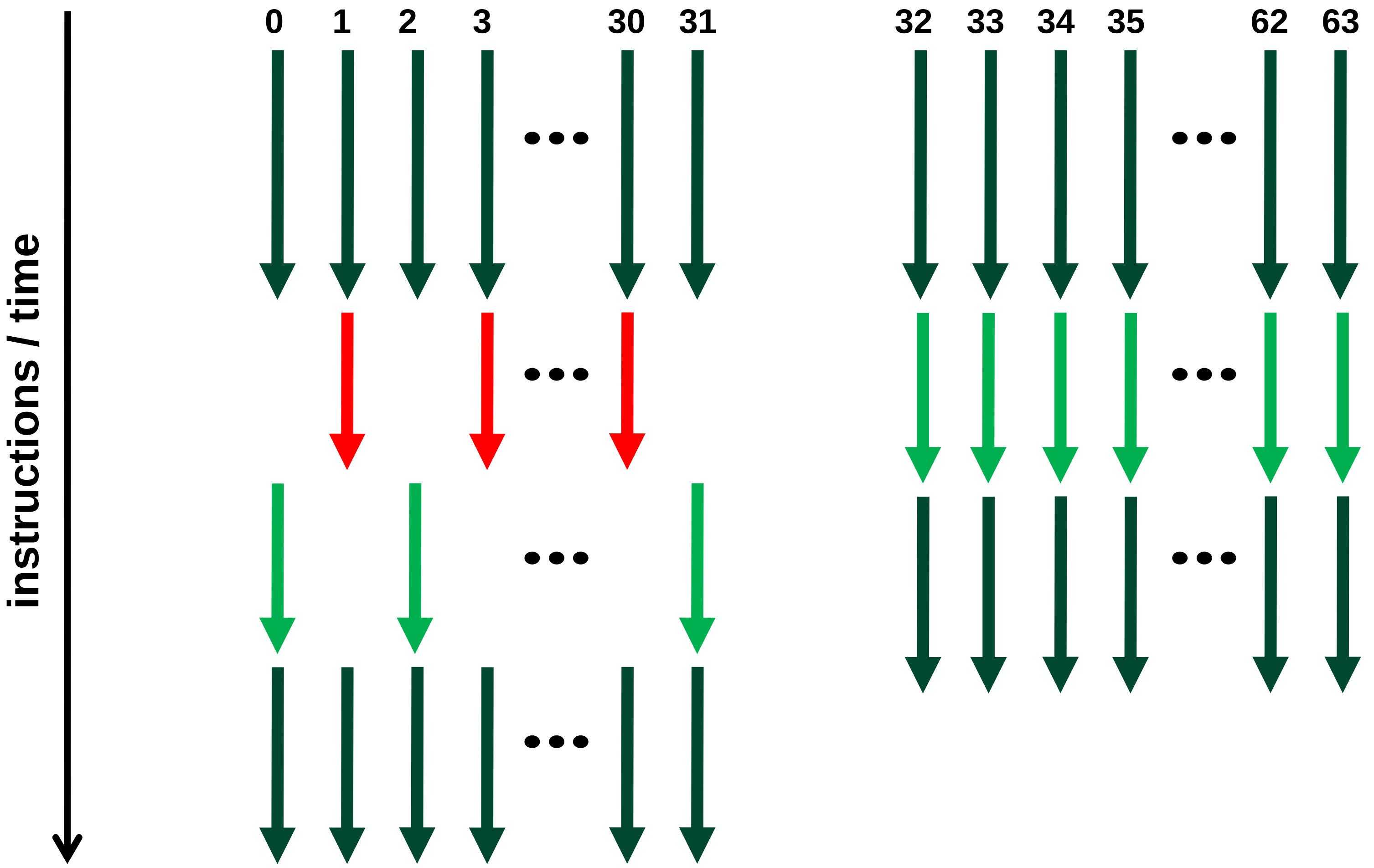



```
if ( ... )
{
    // then-clause
}
else
{
    // else-clause
}
```

Execution within warps is coherent

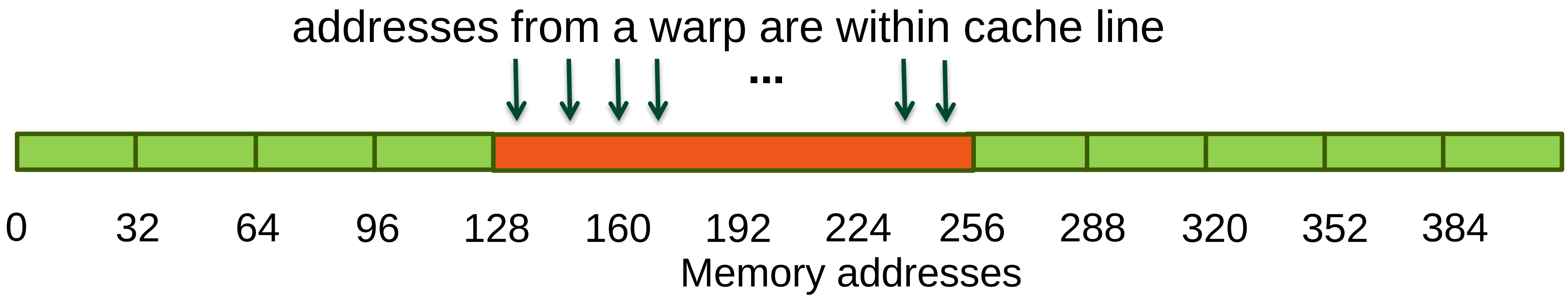


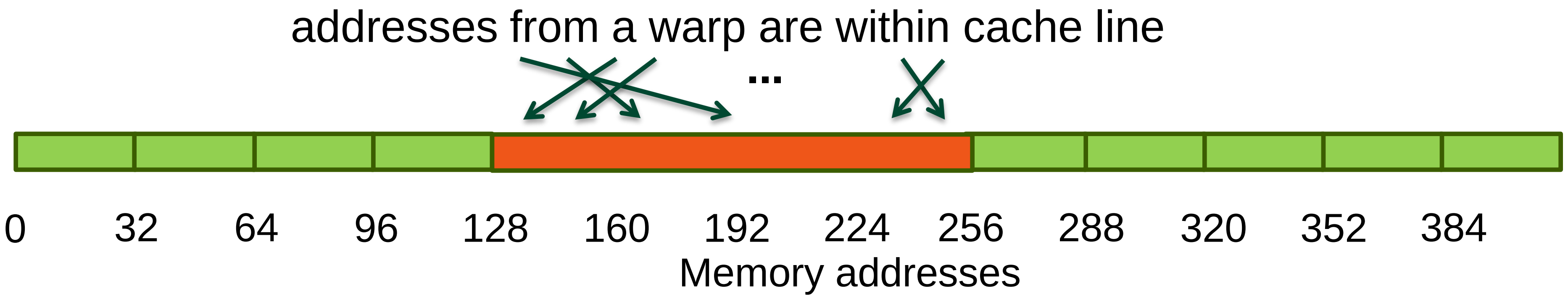
Execution diverges within a warp

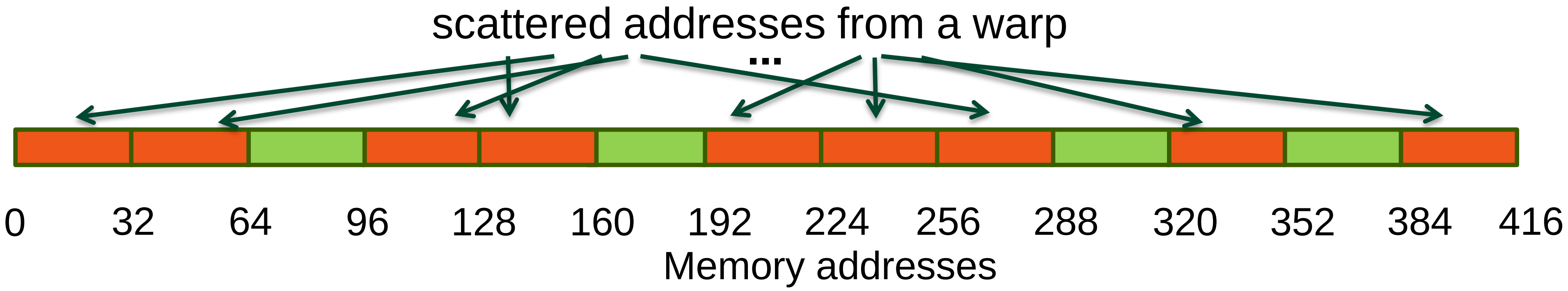


Memory Access

- **Addresses from a warp ("thread-vector") are converted into line requests**
  - line sizes: 32B and 128B
  - Goal is to maximally utilize the bytes in these lines




addresses from a warp are within cache line
...
0
32
64
96
128
160
192
224
256
288
320
352
384
Memory addresses


scattered addresses from a warp
...
0
32
64
96
128
160
192
224
256
288
320
352
384
416
Memory addresses

Performance Optimization

Performance Optimization Process

- Use appropriate performance metric for each kernel
  - For example, Gflops/s don't make sense for a bandwidth-bound kernel
- Determine what limits kernel performance
  - Memory throughput
  - Instruction throughput
  - Latency
  - Combination of the above
- Address the limiters in the order of importance
  - Determine how close to the HW limits the resource is being used
  - Analyze for possible inefficiencies
  - Apply optimizations
    - Often these will just fall out from how HW operates

3 Ways to Assess Performance Limiters

- **Algorithmic**
  - Based on algorithm's memory and arithmetic requirements
  - Least accurate: undercounts instructions and potentially memory accesses
- **Profiler**
  - Based on profiler-collected memory and instruction counters
  - More accurate, but doesn't account well for overlapped memory and arithmetic
- **Code modification**
  - Based on source modified to measure memory-only and arithmetic-only times
  - Most accurate, however cannot be applied to all codes

Things to Know About Your GPU

- **Theoretical memory throughput**
  - For example, Tesla M2090 theory is **177 GB/s**
- **Theoretical instruction throughput**
  - ***Varies by instruction type***
    - refer to the CUDA Programming Guide (Section 5.4.1) for details
  - Tesla M2090 theory is **665 GInstr/s** for fp32 instructions
    - Half that for fp64
    - I’m counting instructions per thread
- **Rough “balanced” instruction:byte ratio**
  - For example, **3.76:1** from above (fp32 instr : bytes)
    - Higher than this will usually mean instruction-bound code
    - Lower than this will usually mean memory-bound code

Another Way to Use the Profiler

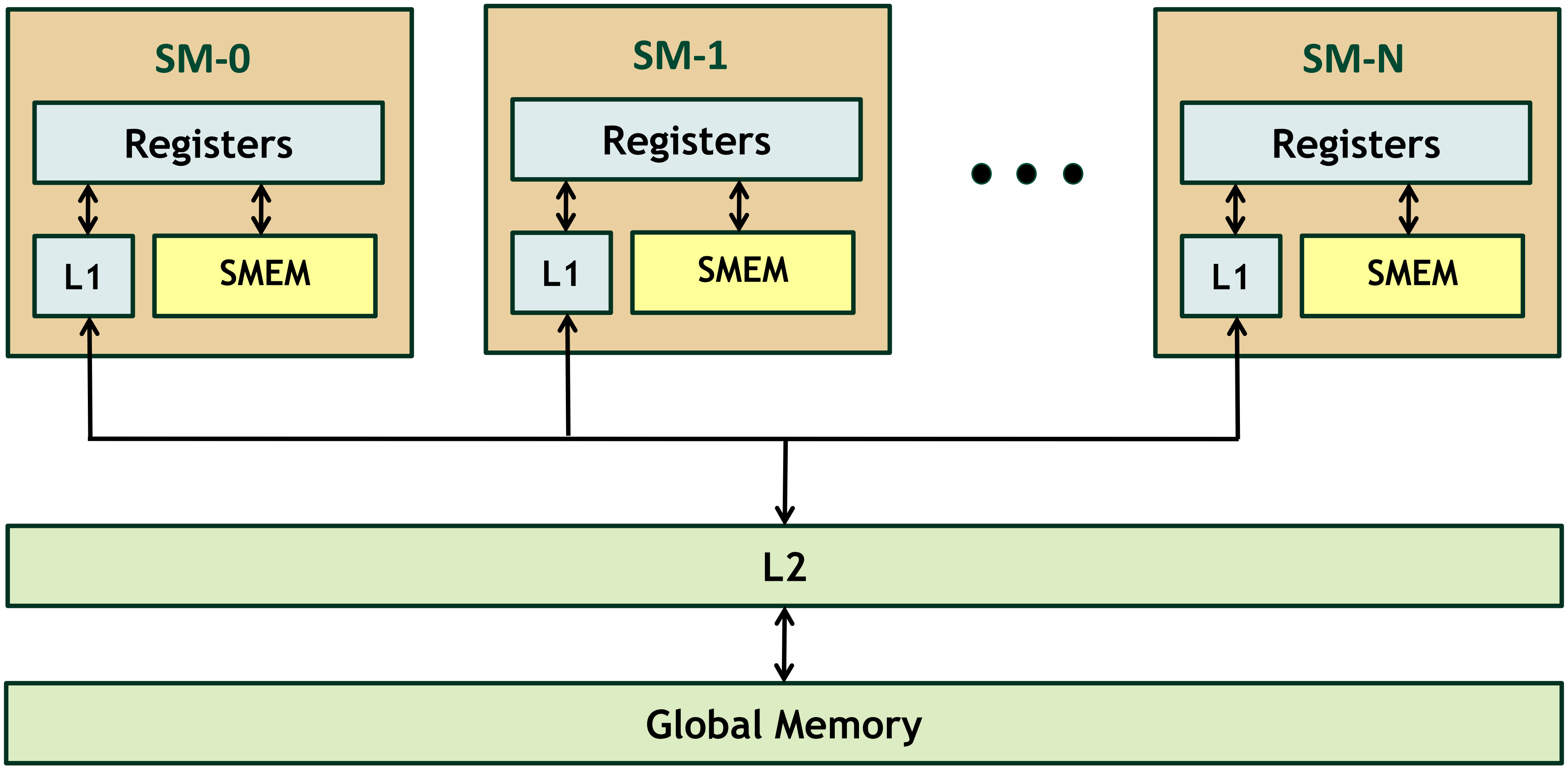
- **VisualProfiler reports instruction and memory throughputs**
  - IPC (instructions per clock) for instructions
  - GB/s achieved for memory (and L2)
- **Compare those with the theory for the HW**
  - Profiler will also report the theoretical best
    - Though for IPC it assumes fp32 instructions, it DOES NOT take instruction mix into consideration
  - If one of the metrics is close to the hw peak, you're likely limited by it
  - If neither metric is close to the peak, then unhidden latency is likely an issue
  - "close" is approximate, I'd say 70% of theory or better
- **Example: vector add**
  - IPC: 0.55 out of 2.0
  - Memory throughput: 130 GB/s out of 177 GB/s
  - Conclusion: memory bound

Notes on 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”
- Refer to the profiler documentation for more information

Global Memory Optimization

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
  - Low latency
  - Very high throughput (1.33 TB/s aggregate on Tesla M2090)
- L2
  - All accesses to global memory go through L2, including copies to/from CPU host
  - 768 KB on Tesla M2090
- Global memory
  - Accessible by all threads as well as host (CPU)
  - Higher latency (400-800 cycles)
  - Throughput: 177 GB/s on Tesla M2090

Programming for L1 and L2

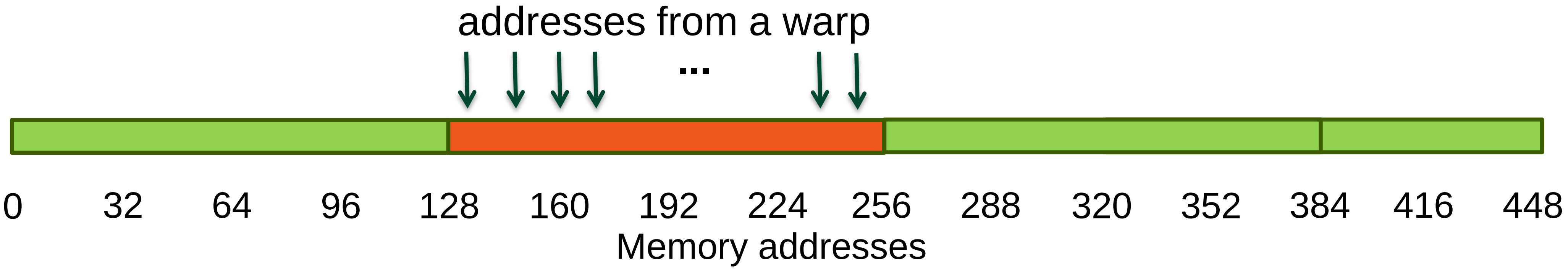
- Short answer: DON'T
- GPU caches are not intended for the same use as CPU caches
  - Smaller size (especially per thread), so not aimed at temporal reuse
  - Intended to smooth out some access patterns, help with spilled registers, etc.
- Don't try to block for L1/L2 like you would on CPU
  - You have 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

Fermi 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
- 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
      - Does not hit in L1, invalidates the line if it's in L1 already
    - Load granularity is 32-bytes
- Stores:
  - Invalidate L1, go at least to L2, 32-byte granularity

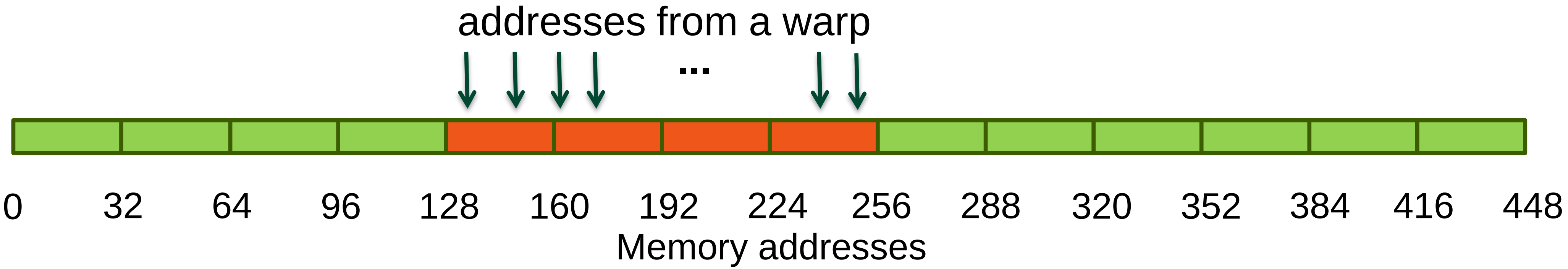
Caching Load

- **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%



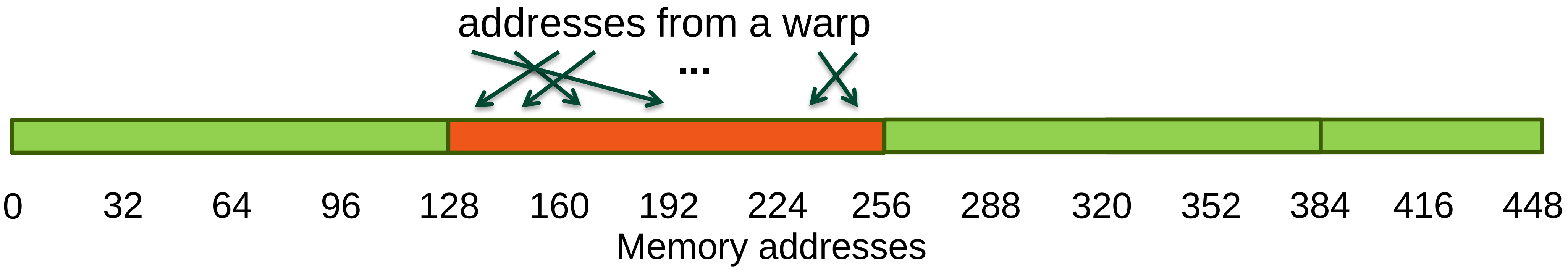
Non-caching Load

- **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%



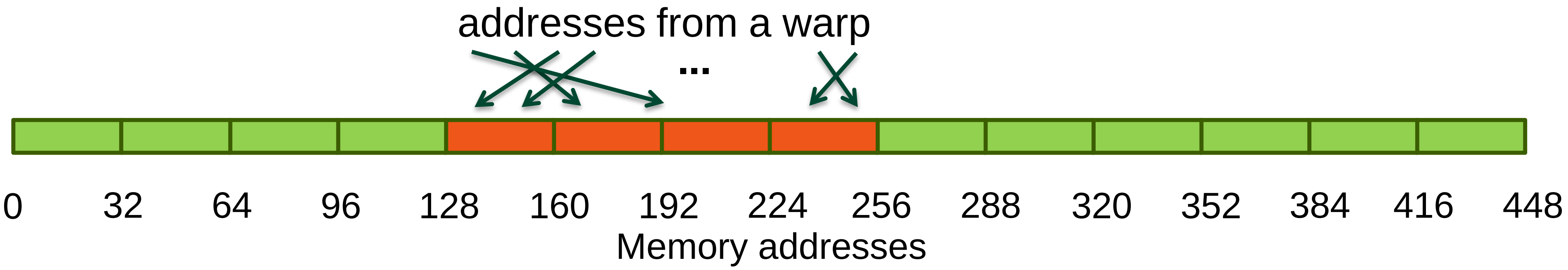
Caching Load

- **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%



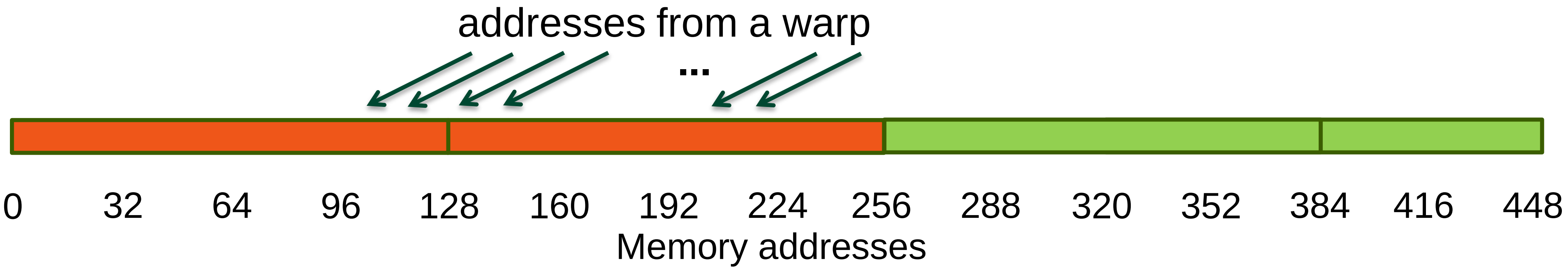
Non-caching Load

- **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%



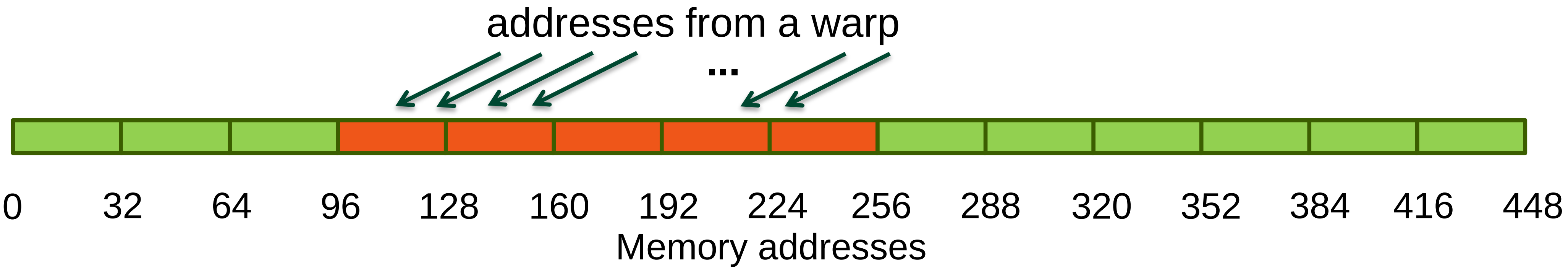
Caching Load

- **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%



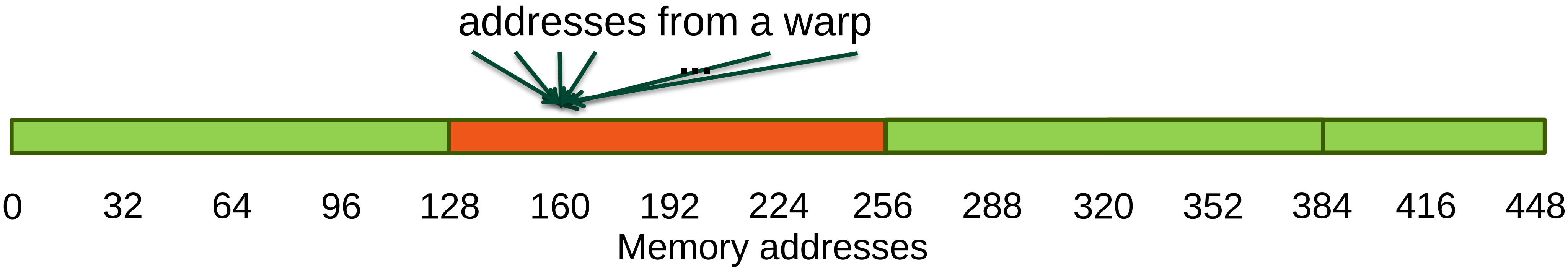
Non-caching Load

- **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



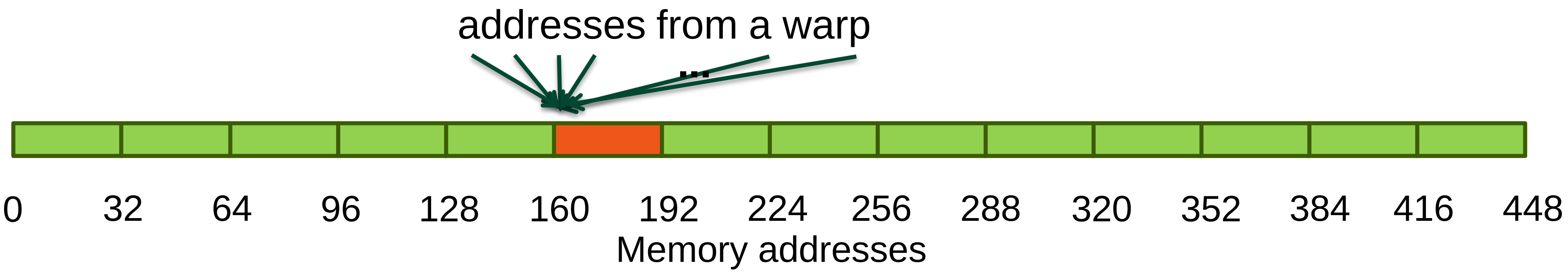
Caching Load

- **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%



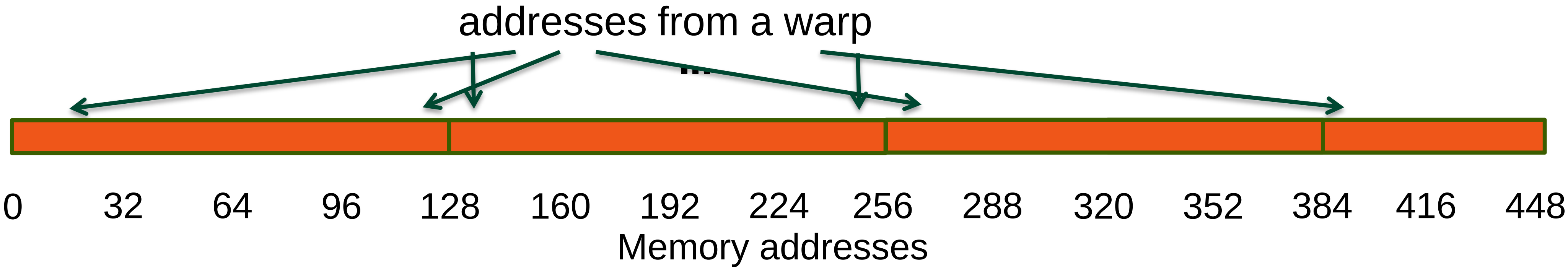
Non-caching Load

- **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%



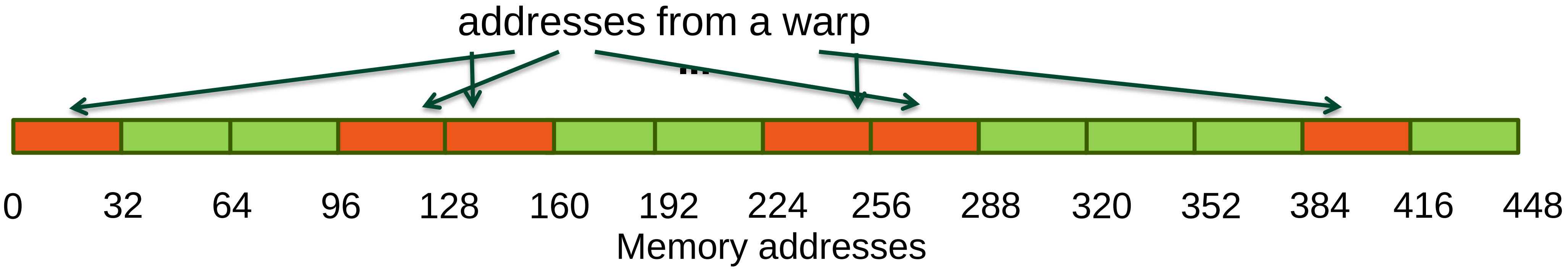
Caching Load

- **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 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)



Load Caching and L1 Size

- **Non-caching loads can improve performance when:**
  - Loading scattered words or only a 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)
- **Large L1 can improve perf when:**
  - Spilling registers (more lines in the cache -> 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: {caching, non-caching} x {48-L1, 16-L1}
    - Keep the best combination - same as you would with any HW managed cache, including CPUs

Memory Throughput Analysis

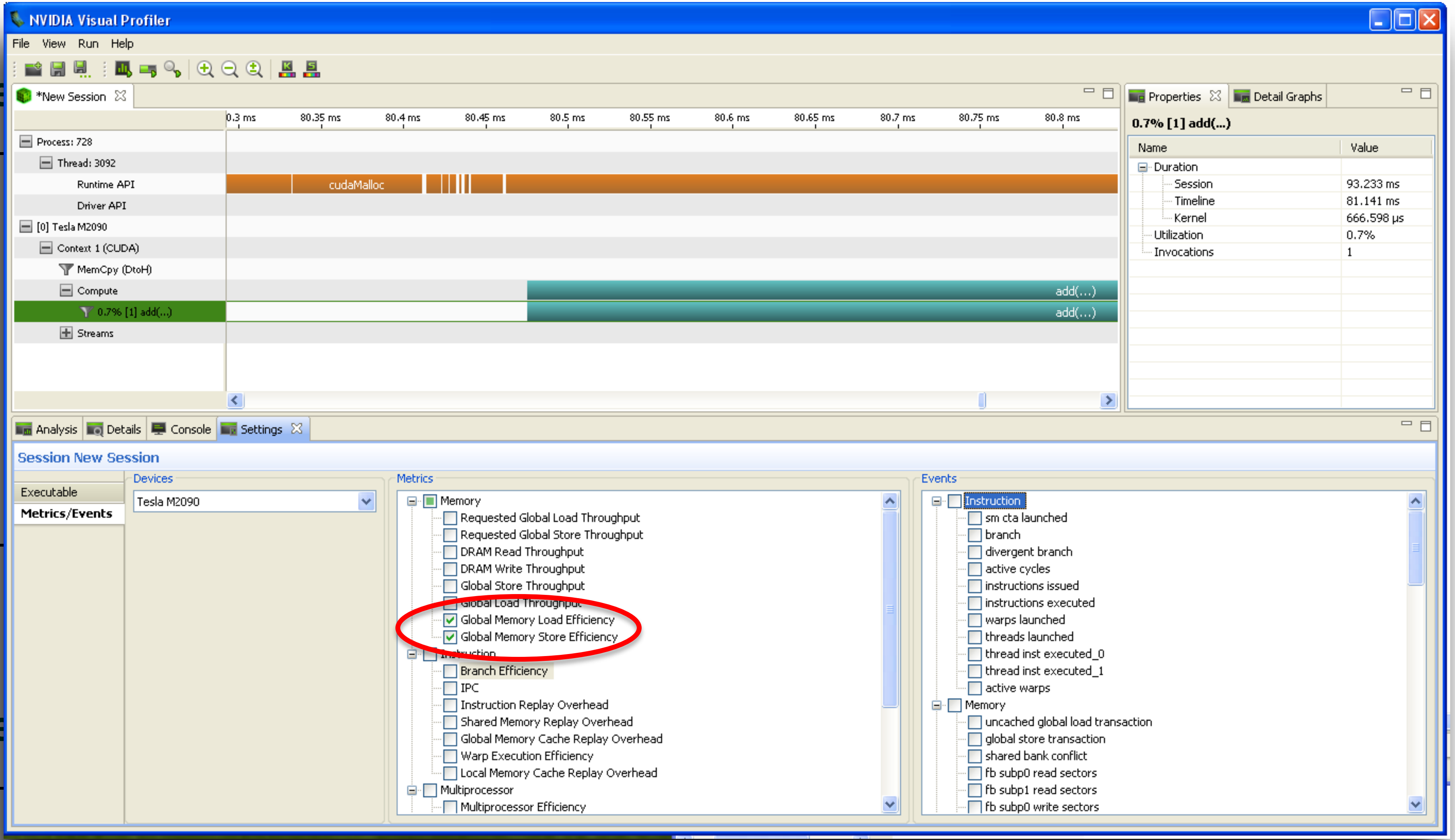
- **Throughput:**
  - From app point of view: count bytes requested by the application
  - From HW point of view: count bytes moved by the hardware
  - The two can be different
    - Scattered/misaligned pattern: not all transaction bytes are utilized
    - Broadcast: the same small transaction serves many requests
- **Two aspects to analyze for performance impact:**
  - Address pattern
  - Number of concurrent accesses in flight

Memory Throughput Analysis

- **Determining that access pattern is problematic:**
  - Use the profiler to check load and store efficiency
    - Efficiency = bytes requested by the app / bytes transferred
    - Will slow down code substantially:
      - Bytes-requested is measured by adding code for every load/store
      - So, you may want to run for smaller data set
    - If efficiency isn't 100%, then bandwidth is being wasted
      - Below 50% certainly means scattered accesses
      - Above 50% could be scattered or misaligned
  - Derive app-requested bytes yourself
    - Still use profiler to get HW throughput (fast, no sw modification)
- **Determining that the number of concurrent accesses is insufficient:**
  - Throughput from HW point of view is much lower than theoretical

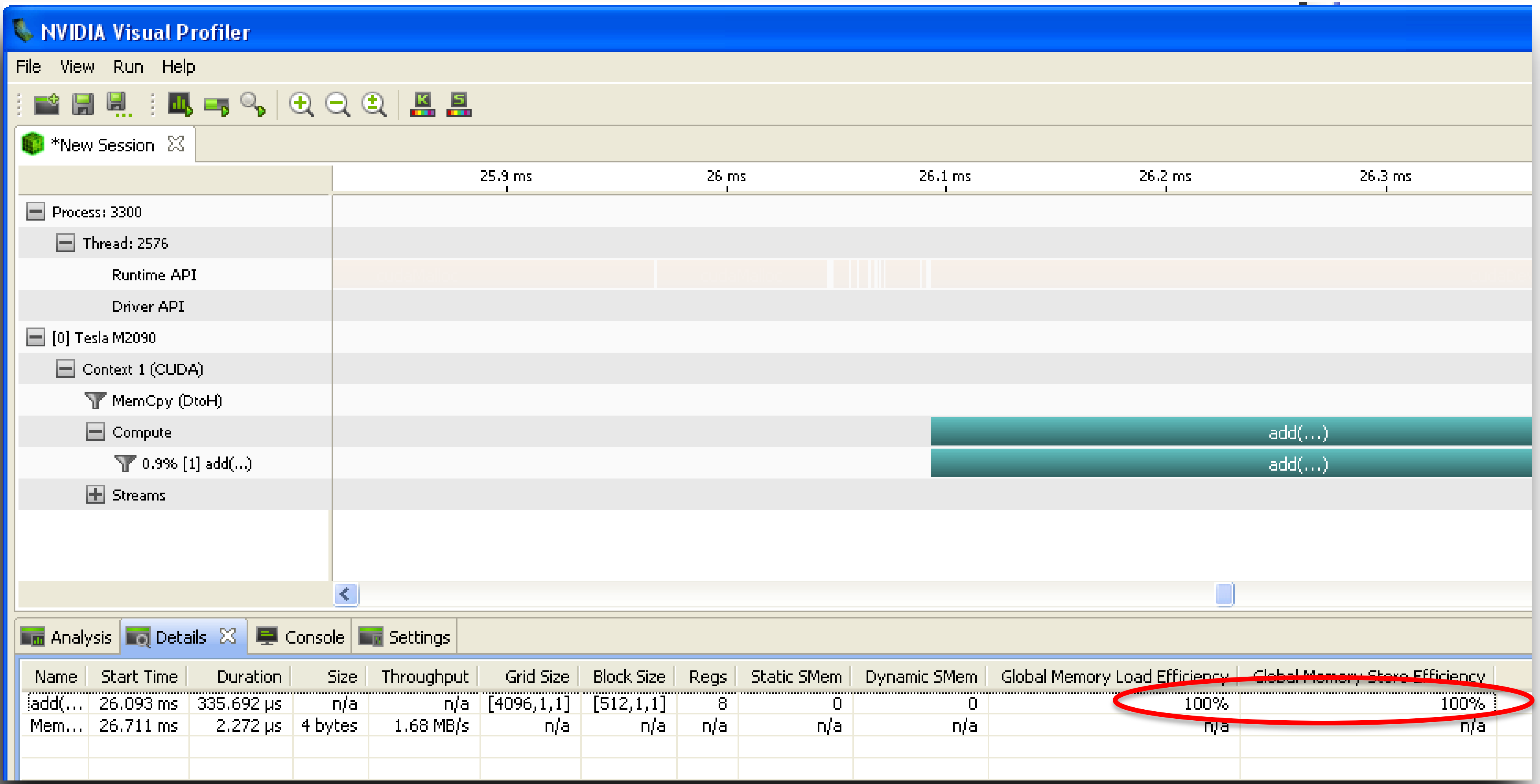
Memory Throughput Analysis

- De



- De

Memory Throughput Analysis



Optimization: Address Pattern

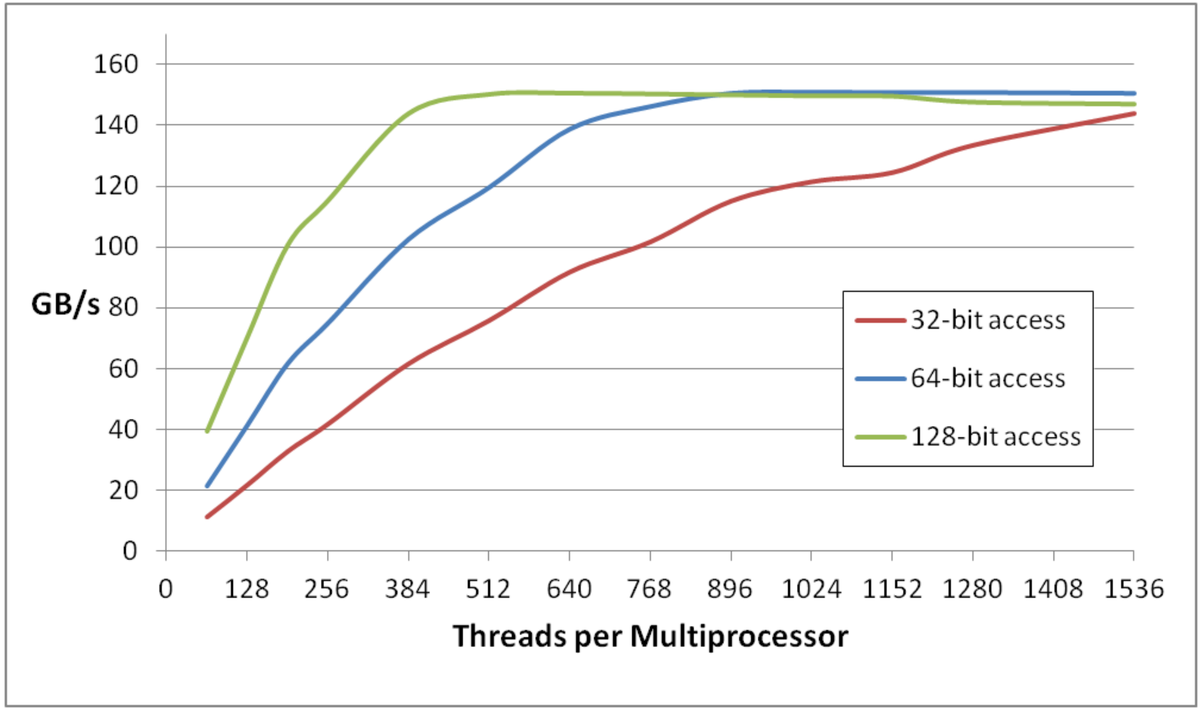
- Coalesce the address pattern
  - Minimize the lines that a warp addresses in a given access
    - 128-byte lines for caching loads, 32-byte segments for non-caching loads, stores
  - Use structure-of-arrays storage (as opposed to array of structures)
    - You have to do this for any architecture, including CPUs
  - Pad multi-dimensional structures so that accesses by warps are aligned on line boundaries
- Try using non-caching loads
  - Smaller transactions (32B instead of 128B)
    - more efficient for scattered or partially-filled patterns
- Try fetching data from texture
  - Smaller transactions and different caching
  - Cache not polluted by other gmem loads

Optimization: Access Concurrency

- **Have enough concurrent accesses to saturate the bus**
  - Need (mem_latency)x(bandwidth) bytes in flight (Little’s law)
- **Ways to increase concurrent accesses:**
  - Increase occupancy (run more threads concurrently)
    - Adjust threadblock dimensions
      - To maximize occupancy at given register and smem requirements
    - Reduce register count (-maxrregcount option, or __launch_bounds__)
  - Modify code to process several elements per thread

Some Experimental Data

- Increment a 64M element array
  - Two accesses per thread (load then store, but they are dependent)
    - Thus, each warp (32 threads) has one outstanding transaction at a time
- Tesla M2090, ECC off, theoretical bandwidth: 177 GB/s



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

For example:

Four 32-bit ~= one 128-bit

Summary: GMEM Optimization

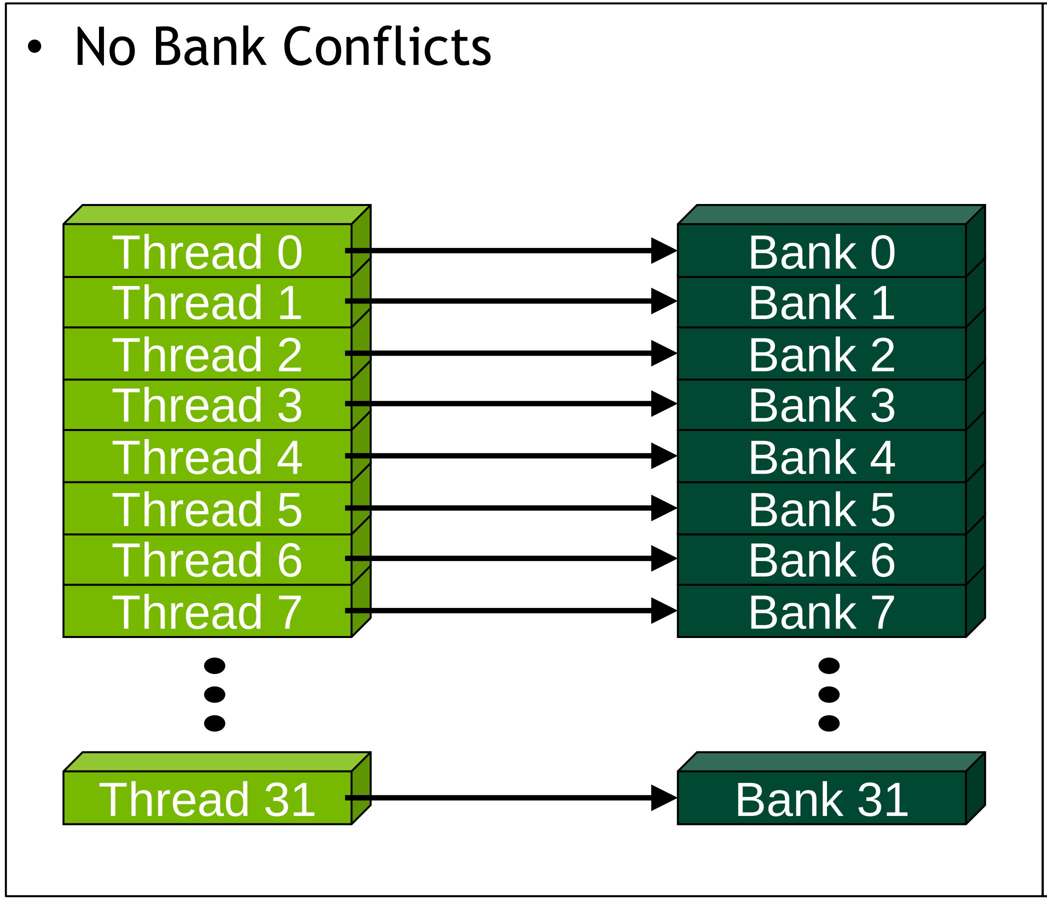
- Strive for perfect coalescing per warp
  - Align starting address (may require padding)
  - A warp should access within a contiguous region
  - Structure of Arrays is better than Array of Structures
- Have enough concurrent accesses to saturate the bus
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)
  - If needed, process several elements per thread
    - More concurrent loads/stores
- Try L1 and caching configurations to see which one works best
  - Caching vs non-caching loads (compiler option)
  - 16KB vs 48KB L1 (CUDA call)

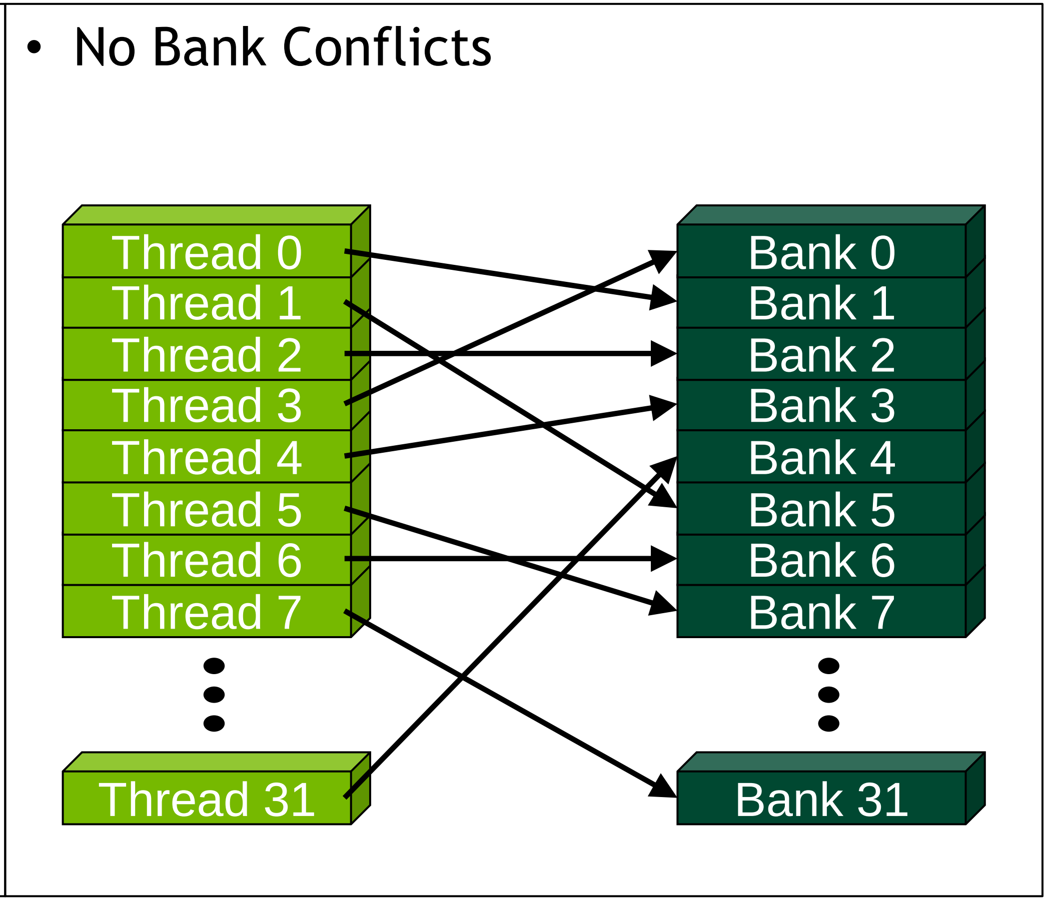
Shared Memory Optimization

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
- **Fermi organization:**
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks
- **Performance:**
  - 4 bytes per bank per 2 clocks per multiprocessor: **1.3 TB/s on M2090**
  - smem accesses are issued per 32 threads (warp)
  - serialization: if *n* threads in a warp 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

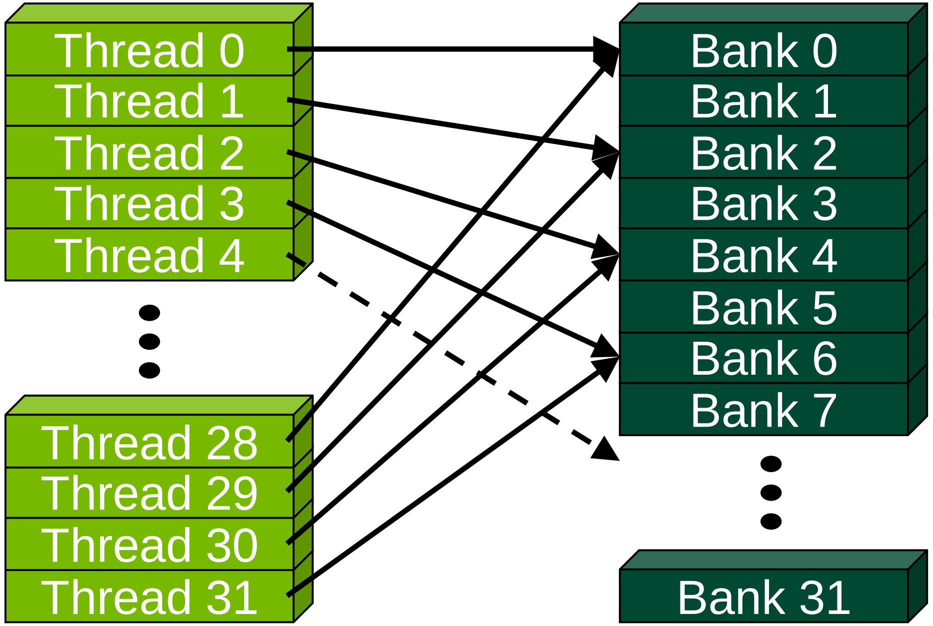
Bank Addressing Examples



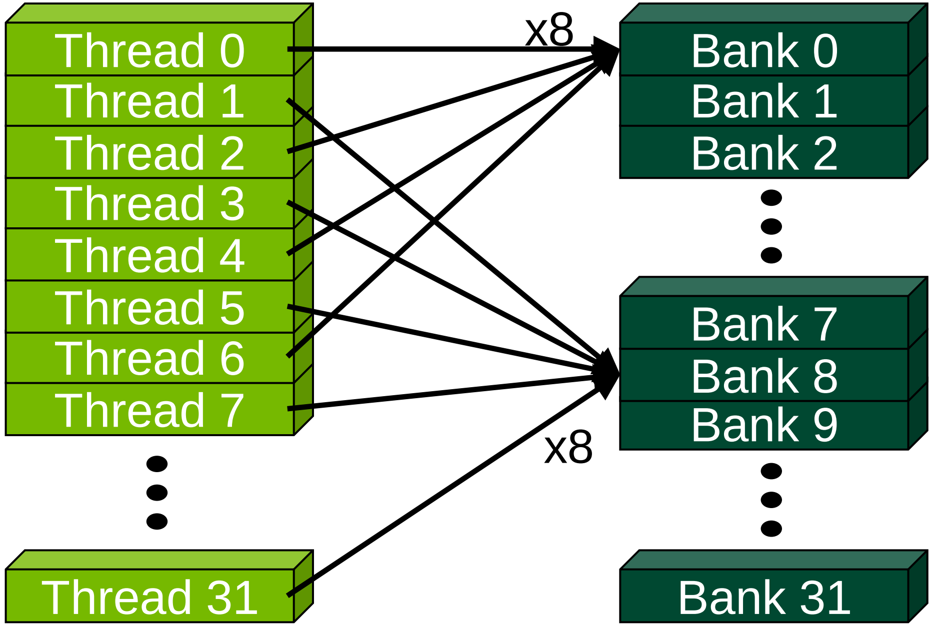


Bank Addressing Examples

- 2-way Bank Conflicts



- 8-way Bank Conflicts

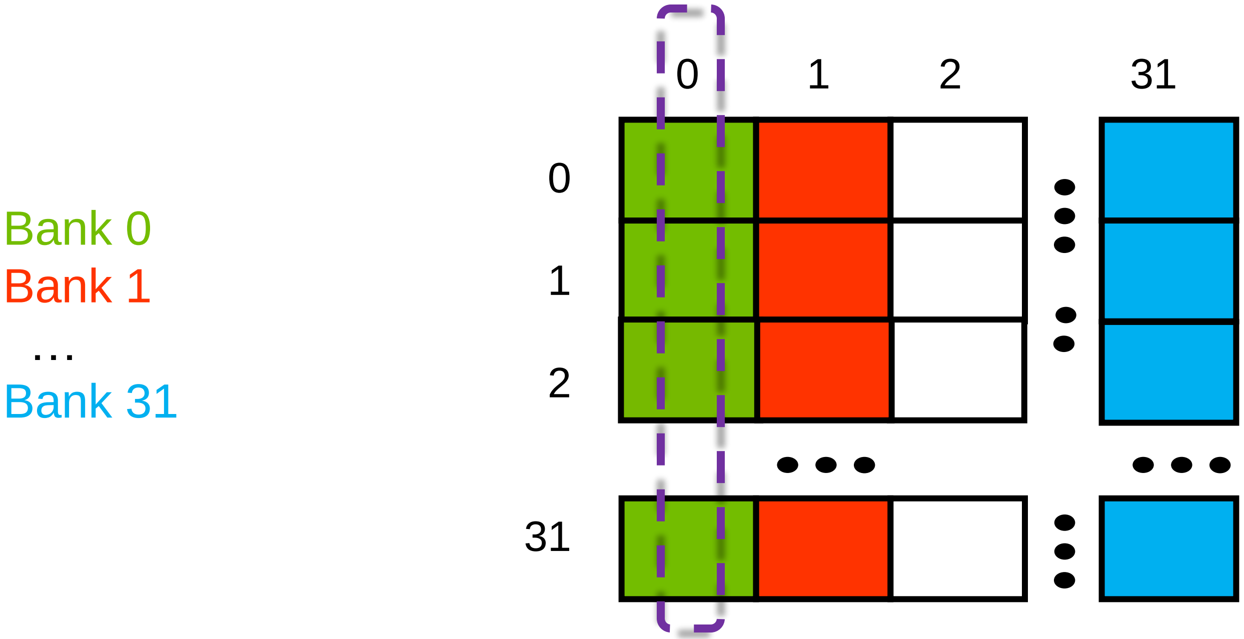


Profiling SMEM Bank Conflicts

- **Find out whether:**
  - Bank conflicts are occuring
  - Bank conflicts significantly impact performance
    - No need to optimize if they don’t
- **Impact on performance is significant if:**
  - Kernel is limited by instruction throughput
  - Shared memory bank conflicts are a significant percentage of instructions issued
- **Use the profiler to get:**
  - Bank conflict count, instructions-issued count
    - Currently bank-conflicts get overcounted for accesses greater than 32-bit words:
      - Divide by 2 for 64-bit accesses (double, float2, etc.)
      - Divide by 4 for 128-bit accesses (float4, etc.)

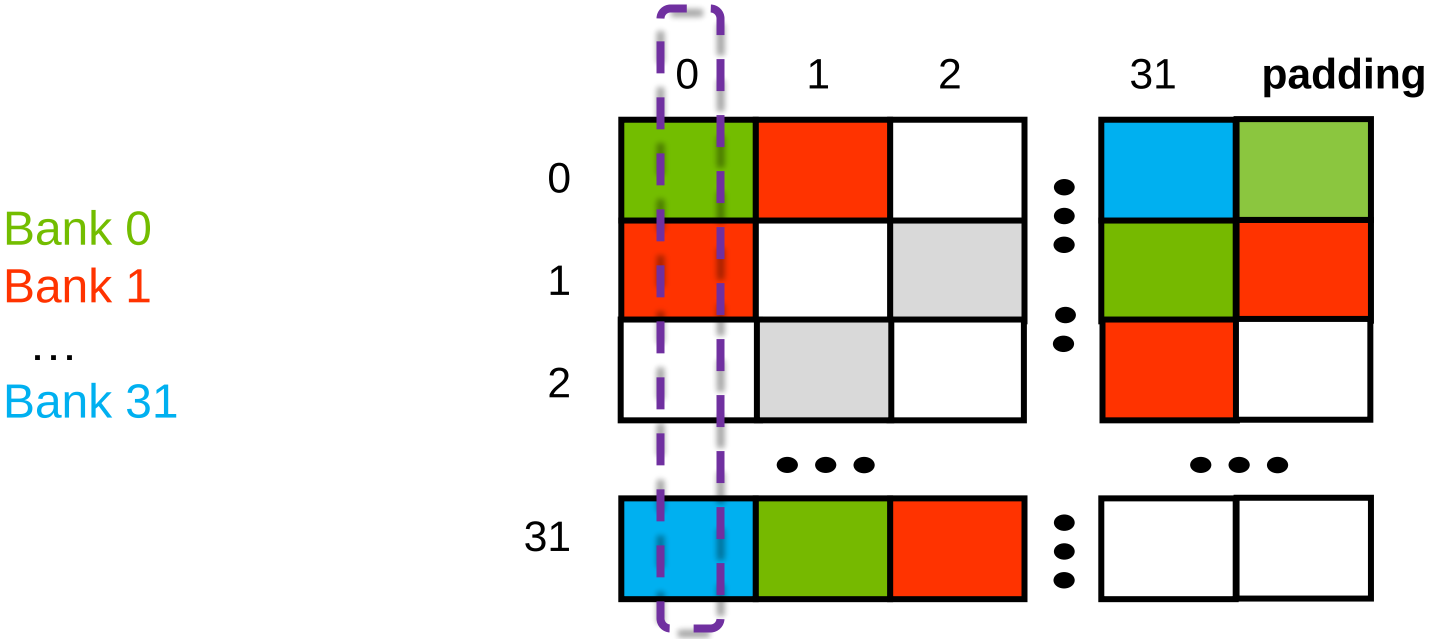
Shared Memory: Avoiding Bank Conflicts

- 32x32 SMEM array
- Warp accesses a column:
  - 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:
  - 32 different banks, no bank conflicts



Case Study: SMEM Bank Conflicts

- One of CAM-HOMME kernels (climate simulation), fp64
- Profiler values:
  - Instructions:Byte ratio, reported by profiler: 4
    - Suggests kernel is instruction limited (even before adjusting for fp64 throughput)
  - Instruction counts
    - Executed / issued: 2,406,426 / 2,756,140
    - Difference: 349,714 (**12.7%** of instructions issued were “replays”)
  - SMEM instructions:
    - Load + store: 421,785 + 95,172 = 516,957
    - Bank conflicts: 674,856 (really 337,428 because of double-counting for fp64)
    - So, SMEM bank conflicts make up 12.2% of all instructions (337,428 / 2,756,140)
- Solution: pad shared memory
  - Performance increased by ~15%

Texture and Constant Data

Constant and Texture Data

- Constants:
  - __constant__ qualifier in declarations
  - Up to 64KB
  - Ideal when the same address is read by all threads in a warp (FD coefficients, etc.)
    - Throughput is 4B per SM per clock
- Textures:
  - Dedicated hardware for:
    - Out-of-bounds index handling (clamp or wrap-around)
    - Optional interpolation (think: using fp indices for arrays)
      - Linear, bilinear, trilinear
    - Optional format conversion
      - {char, short, int} -> float
- Operation:
  - Both textures and constants reside in global memory
  - Both are read via dedicated caches

Instruction Throughput and Optimization

Kernel Execution

- Threadblocks are assigned to SMs
  - Done at run-time, so don’t assume any particular order
  - Once a threadblock is assigned to an SM, it stays resident until all its threads complete
    - It’s not migrated to another SM
    - It’s not swapped out for another threadblock
- 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

- **Divergent branches:**
  - Threads within a single warp take different paths
    - `if-else`, ...
  - Different execution paths within a warp are serialized
- **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

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
  - Some causes:
    - Shared memory bank conflicts
    - Constant memory bank conflicts

Instruction Throughput: Analysis

- Compare achieved instruction throughput to HW capabilities
  - Profiler reports achieved throughput as IPC (instructions per clock)
    - As percentage of theoretical peak for pre-Fermi GPUs
  - 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 = difference between instructions_issued and instructions_executed counters
  - Profiler reports % of serialization metric, additional counters for smem bank conflicts
  - A concern only if code is instruction-bound and serialization 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 (select GT200-level precision):
  - -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

Serialization: Optimization

- **Shared memory bank conflicts:**
  - Covered earlier in this presentation
- **Constant memory bank conflicts:**
  - Ensure that all threads in a warp access the same __constant__ value
  - If many different values will be needed per warp:
    - Use gmem or smem instead
- **Warp serialization:**
  - Try grouping threads that take the same path
    - Rearrange the data, pre-process the data
    - Rearrange how threads index data (may affect memory perf)

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)
  - Avoid SMEM bank conflicts (pad, rearrange data)

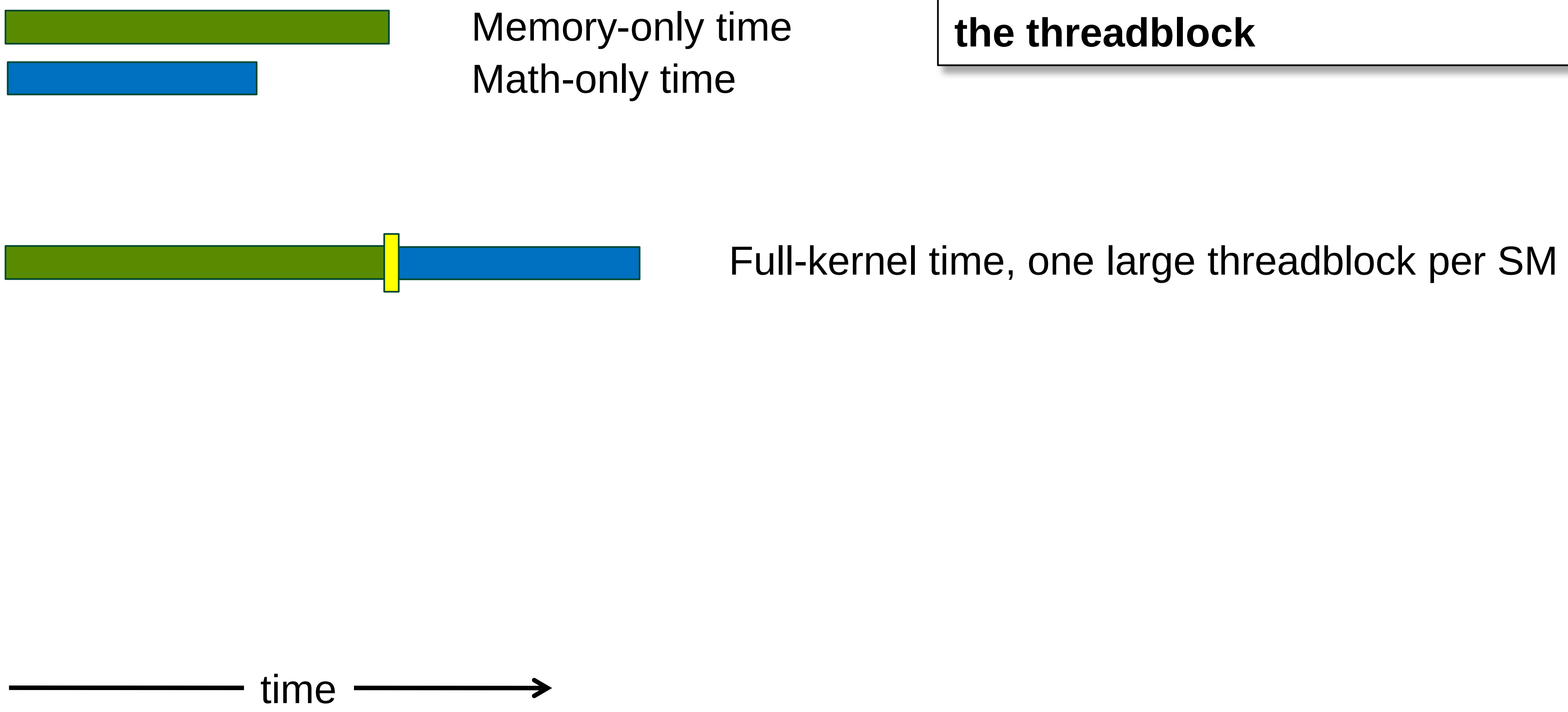
Latency Hiding

Latency: Analysis

- Suspect unhidden latency if:
  - Neither memory nor instruction throughput is close to HW theoretical rates
  - Poor overlap between mem and math
    - Full-kernel time is significantly larger than max{mem-only, math-only}
      - Refer to SC10 or GTC10 Analysis-Driven Optimization slides for details
- Two possible causes:
  - Insufficient concurrent threads per multiprocessor to hide latency
    - Occupancy too low
    - Too few threads in kernel launch to load the GPU
      - elapsed time doesn't change if problem size is increased (and with it the number of blocks/threads)
  - Too few concurrent threadblocks per SM when using __syncthreads()
    - __syncthreads() can prevent overlap between math and mem within the same threadblock

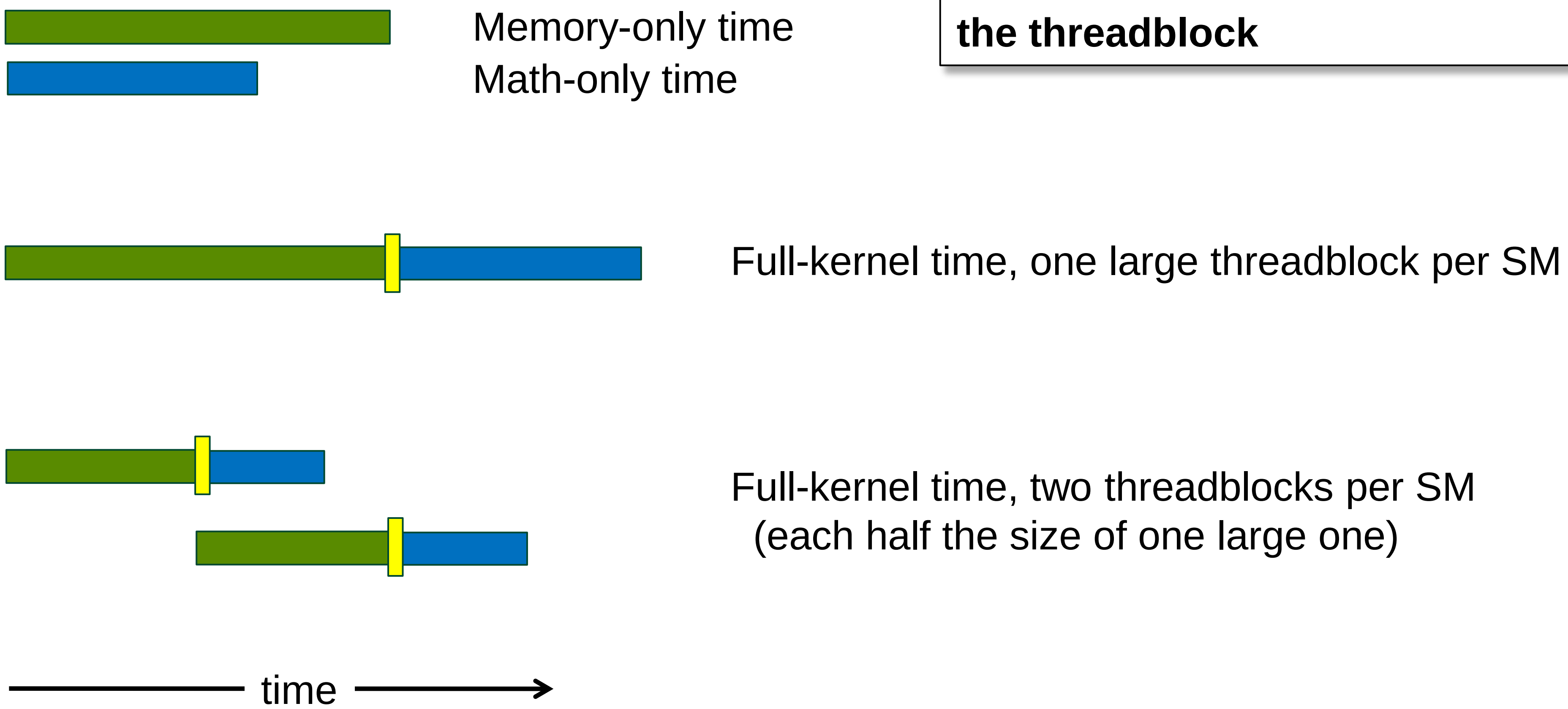
Simplified View of Latency and Syncs

Kernel where most math cannot be executed until all data is loaded by the threadblock



Simplified View of Latency and Syncs

Kernel where most math cannot be executed until all data is loaded by the threadblock



Latency: Optimization

- **Insufficient threads or workload:**
  - Increase the level of parallelism (more threads)
  - If occupancy is already high but latency is not being hidden:
    - Process several output elements per thread – gives more independent memory and arithmetic instructions (which get pipelined)
- **Barriers:**
  - Can assess impact on perf by commenting out __syncthreads()
    - Incorrect result, but gives upper bound on improvement
  - Try running several smaller threadblocks
    - Think of it as “pipeled” threadblock execution
    - In some cases that costs extra bandwidth due to halos
- **Check out Vasily Volkov’s talk 2238 at GTC 2010 for a detailed treatment:**
  - “Better Performance at Lower Latency”

Summary

- **Keep the 3 requirements for max performance in mind:**
  - Sufficient parallelism
  - Coalesced memory access
  - Coherent (vector) execution within warps
- **Determine what limits kernel performance**
  - Memory, arithmetic, latency
- **Optimize in the order of limiter severity**
  - Use the profiler to determine performance impact first
    - Some code modifications help here too

Additional Resources

- **Fundamental Optimizations / Analysis-Driven Optimization**
  - More detailed treatment of this information, more cases studies
  - SC10: http://www.nvidia.com/object/sc10_cuda_tutorial.html
  - GTC10 (includes video recordings):
    - http://www.gputechconf.com/page/gtc-on-demand.html#2011
    - http://www.gputechconf.com/page/gtc-on-demand.html#2012
- **CUDA Best Practices Guide / CUDA Programming Guide**
  - Included in the docs of any CUDA toolkit
  - All optimization materials apply to OpenCL and other programming models
- **CUDA Webinars:**
  - http://developer.nvidia.com/gpu-computing-webinars
  - Shorter, more focused presentations (recorded video of past talks)
    - Memory optimization, local memory and register spilling, etc.

Questions?