Analysis-Driven Optimization

Paulius Micikevicius | NVIDIA

Supercomputing, Tutorial S03
New Orleans, Nov 14, 2010
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
Presentation Outline

• **Identifying performance limiters**
• **Analyzing and optimizing**:
  – Memory-bound kernels
  – Instruction (math) bound kernels
  – Kernels with poor latency hiding
  – Register spilling

• **For each**:
  – Brief background
  – How to analyze
  – How to judge whether particular issue is problematic
  – How to optimize
  – Some cases studies based on “real-life” application kernels

• **Most information is for Fermi GPUs**
Notes on 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
Identifying Performance Limiters
Limited by Bandwidth or Arithmetic?

• Perfect instructions:bytes ratio for Fermi C2050:
  – ~4.5 : 1 with ECC on
  – ~3.6 : 1 with ECC off
  – These assume fp32 instructions, throughput for other instructions varies

• Algorithmic analysis:
  – Rough estimate of arithmetic to bytes ratio

• Code likely uses more instructions and bytes than algorithm analysis suggests:
  – Instructions for loop control, pointer math, etc.
  – Address pattern may result in more memory fetches
  – Two ways to investigate:
    • Use the profiler (quick, but approximate)
    • Use source code modification (more accurate, more work intensive)
Analysis with Profiler

• Profiler counters:
  – instructions_issued, instructions_executed
    • Both incremented by 1 per warp
    • “issued” includes replays, “executed” does not
  – gld_request, gst_request
    • Incremented by 1 per warp for each load/store instruction
    • Instruction may be counted if it is “predicated out”
  – l1_global_load_miss, l1_global_load_hit, global_store_transaction
    • Incremented by 1 per L1 line (line is 128B)
  – uncached_global_load_transaction
    • Incremented by 1 per group of 1, 2, 3, or 4 transactions
    • Better to look at L2_read_request counter (incremented by 1 per 32 bytes, per GPU)

• Compare:
  – 32 * instructions_issued /* 32 = warp size */
  – 128B * (global_store_transaction + l1_global_load_miss)
A Note on Counting Global Memory Accesses

• Load/store instruction count can be lower than the number of actual memory transactions
  – Address pattern, different word sizes

• Counting requests from L1 to the rest of the memory system makes the most sense
  – Caching-loads: count L1 misses
  – Non-caching loads and stores: count L2 read requests
    • Note that L2 counters are for the entire chip, L1 counters are per SM

• Some shortcuts, assuming “coalesced” address patterns:
  – One 32-bit access instruction  -> one 128-byte transaction per warp
  – One 64-bit access instruction  -> two 128-byte transactions per warp
  – One 128-bit access instruction -> four 128-byte transactions per warp
Analysis with Modified Source Code

• **Time memory-only and math-only versions of the kernel**
  – Easier for codes that don’t have data-dependent control-flow or addressing
  – Gives you good estimates for:
    • Time spent accessing memory
    • Time spent in executing instructions

• **Comparing the times for modified kernels**
  – Helps decide whether the kernel is mem or math bound
  – Shows how well memory operations are overlapped with arithmetic
    • Compare the sum of mem-only and math-only times to full-kernel time
Some Example Scenarios

Memory-bound

Good mem-math overlap: latency not a problem (assuming memory throughput is not low compared to HW theory)
Some Example Scenarios

Memory-bound

Good mem-math overlap: latency not a problem
(assuming memory throughput is not low compared to HW theory)

Math-bound

Good mem-math overlap: latency not a problem
(assuming instruction throughput is not low compared to HW theory)
Some Example Scenarios

- **Memory-bound**
  - Good mem-math overlap: latency not a problem
  - (assuming memory throughput is not low compared to HW theory)

- **Math-bound**
  - Good mem-math overlap: latency not a problem
  - (assuming instruction throughput is not low compared to HW theory)

- **Balanced**
  - Good mem-math overlap: latency not a problem
  - (assuming memory/instr throughput is not low compared to HW theory)
Some Example Scenarios

Memory-bound
Good mem-math overlap: latency not a problem
(assuming memory throughput is not low compared to HW theory)

Math-bound
Good mem-math overlap: latency not a problem
(assuming instruction throughput is not low compared to HW theory)

Balanced
Good mem-math overlap: latency not a problem
(assuming memory/instr throughput is not low compared to HW theory)

Memory and latency bound
Poor mem-math overlap: latency is a problem
Source Modification

• Memory-only:
  – Remove as much arithmetic as possible
    • Without changing access pattern
    • Use the profiler to verify that load/store instruction count is the same

• Store-only:
  – Also remove the loads

• Math-only:
  – Remove global memory accesses
  – Need to trick the compiler:
    • Compiler throws away all code that it detects as not contributing to stores
    • Put stores inside conditionals that always evaluate to false
      – Condition should depend on the value about to be stored (prevents other optimizations)
      – Condition outcome should not be known to the compiler
Source Modification for Math-only

```c
__global__ void fwd_3D( ..., int flag)
{
    ...
    value = temp + coeff * vsq;
    if( 1 == value * flag )
        g_output[out_idx] = value;
}
```

If you compare only the flag, the compiler may move the computation into the conditional as well.
Source Modification and Occupancy

• Removing pieces of code is likely to affect register count
  – This could increase occupancy, skewing the results
  – See slide 23 to see how that could affect throughput

• Make sure to keep the same occupancy
  – Check the occupancy with profiler before modifications
  – After modifications, if necessary add shared memory to match the unmodified kernel’s occupancy

  kernel<<< grid, block, smem, ...>>>(...)
Case Study: Limiter Analysis

- 3DFD of the wave equation, fp32
- Time (ms):
  - Full-kernel: 35.39
  - Mem-only: 33.27
  - Math-only: 16.25
- Instructions issued:
  - Full-kernel: 18,194,139
  - Mem-only: 7,497,296
  - Math-only: 16,839,792
- Memory access transactions:
  - Full-kernel: 1,708,032
  - Mem-only: 1,708,032
  - Math-only: 0

- Analysis:
  - Instr:byte ratio = ~2.66
    - $32 \times 18,194,139 / 128 \times 1,708,032$
  - Good overlap between math and mem:
    - 2.12 ms of math-only time (13%) are not overlapped with mem
  - App memory throughput: 62 GB/s
    - HW theory is 114 GB/s, so we’re off

- Conclusion:
  - Code is memory-bound
  - Latency could be an issue too
  - Optimizations should focus on memory throughput first

- Math contributes very little to total time (2.12 out of 35.39 ms)
Case Study: Limiter Analysis

- 3DFD of the wave equation, fp32
- Time (ms):
  - Full-kernel: 35.39
  - Mem-only: 33.27
  - Math-only: 16.25
- Instructions issued:
  - Full-kernel: 18,194,139
  - Mem-only: 7,497,296
  - Math-only: 16,839,792
- Memory access transactions:
  - Full-kernel: 1,708,032
  - Mem-only: 1,708,032
  - Math-only: 0

Analysis:
- Instr:byte ratio = ~2.66
  - $32 \times 18,194,139 / 128 \times 1,708,032$
- Good overlap between math and mem:
  - 2.12 ms of math-only time (13%) are not overlapped with mem
- App memory throughput: 62 GB/s
  - HW theory is 114 GB/s, so we’re off

Conclusion:
- Code is memory-bound
- Latency could be an issue too
- Optimizations should focus on memory throughput first
  - Math contributes very little to total time (2.12 out of 35.39 ms)
Summary: Limiter Analysis

• Rough algorithmic analysis:
  – How many bytes needed, how many instructions

• Profiler analysis:
  – Instruction count, memory request/transaction count

• Analysis with source modification:
  – Memory-only version of the kernel
  – Math-only version of the kernel
  – Examine how these times relate and overlap
Optimizations for Global Memory
Memory Throughput Analysis

• Throughput: from application point of view
  – 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:
  – Addressing pattern
  – Number of concurrent accesses in flight
Memory Throughput Analysis

• Determining that access pattern is problematic:
  – Profiler counters: access instruction count is significantly smaller than transaction count
    • gld_request < ( l1_global_load_miss + l1_global_load_hit) * (word_size / 4B )
    • gst_request < 4 * l2_write_requests * (word_size / 4B )
      • Make sure to adjust the transaction counters for word size (see slide 8)
  – App throughput is much smaller than HW throughput
    • Use profiler to get HW throughput

• Determining that the number of concurrent accesses is insufficient:
  – Throughput from HW point of view is much lower than theoretical
Concurrent Accesses and Performance

• 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 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
Optimization: Address Pattern

• Coalesce the address pattern
  – 128-byte lines for caching loads
  – 32-byte segments for non-caching loads, stores
  – A warp’s address pattern is converted to transactions
    • Coalesce to maximize utilization of bus transactions
    • Refer to CUDA Programming Guide / Best Practices Guide / Fundamental Opt. talk

• 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
Optimizing Access Concurrency

• Have enough concurrent accesses to saturate the bus
  – Need \((\text{mem\_latency}) \times \text{bandwidth}\) bytes in flight (Little’s law)
  – Fermi C2050 global memory:
    • 400-800 cycle latency, 1.15 GHz clock, 144 GB/s bandwidth, 14 SMs
    • Need 30-50 128-byte transactions in flight per SM

• Ways to increase concurrent accesses:
  – Increase occupancy
    • 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
Case Study: Access Pattern 1

• Same 3DFD code as in the previous study
• Using caching loads (compiler default):
  – Memory throughput: 62 / 74 GB/s for app / hw
  – Different enough to be interesting
• Loads are coalesced:
  – gld_request == ( l1_global_load_miss + l1_global_load_hit )
• There are halo loads that use only 4 threads out of 32
  – For these transactions only 16 bytes out of 128 are useful
• Solution: try non-caching loads ( -Xptxas -dlcm=cg compiler option)
  – Performance increase of 7%
    • Not bad for just trying a compiler flag, no code change
  – Memory throughput: 66 / 67 GB/s for app / hw
Case Study: Accesses in Flight

• **Continuing with the FD code**
  – Throughput from both app and hw point of view is 66-67 GB/s
  – Now 30.84 out of 33.71 ms are due to mem
  – 1024 concurrent threads per SM
    • Due to register count (24 per thread)
    • Simple copy kernel reaches ~80% of achievable mem throughput at this thread count

• **Solution: increase accesses per thread**
  – Modified code so that each thread is responsible for 2 output points
    • Doubles the load and store count per thread, saves some indexing math
    • Doubles the tile size -> reduces bandwidth spent on halos
  – Further 25% increase in performance
    • App and HW throughputs are now 82 and 84 GB/s, respectively
Case Study: Access Pattern 2

- **Kernel from climate simulation code**
  - Mostly fp64 (so, at least 2 transactions per mem access)

- **Profiler results:**
  - `gld_request`: 72,704
  - `l1_global_load_hit`: 439,072
  - `l1_global_load_miss`: 724,192

- **Analysis:**
  - L1 hit rate: 37.7%
  - 16 transactions per load instruction
    - Indicates bad access pattern (2 are expected due to 64-bit words)
    - Of the 16, 10 miss in L1 and contribute to mem bus traffic
    - So, we fetch 5x more bytes than needed by the app
Case Study: Access Pattern 2

• Looking closer at the access pattern:
  – Each thread linearly traverses a contiguous memory region
  – Expecting CPU-like L1 caching
    • Remember what I said about coding for L1 and L2
    • (Fundamental Optimizations, slide 11)
  – One of the worst access patterns for GPUs

• Solution:
  – Transposed the code so that each warp accesses a contiguous memory region
  – 2.17 transactions per load instruction
  – This and some other changes improved performance by 3x
Optimizing with Compression

• When all else has been optimized and kernel is limited by the number of bytes needed, consider compression

• Approaches:
  – Int: conversion between 8-, 16-, 32-bit integers is 1 instruction (64-bit requires a couple)
  – FP: conversion between fp16, fp32, fp64 is one instruction
    • fp16 (1s5e10m) is storage only, no math instructions
  – Range-based:
    • Lower and upper limits are kernel arguments
    • Data is an index for interpolation between the limits

• Application in practice:
  – Clark et al. “Solving Lattice QCD systems of equations using mixed precision solvers on GPUs”
Summary: Memory Analysis and Optimization

• Analyze:
  – Access pattern:
    • Compare counts of access instructions and transactions
    • Compare throughput from app and hw point of view
  – Number of accesses in flight
    • Look at occupancy and independent accesses per thread
    • Compare achieved throughput to theoretical throughput
      – Also to simple memcpy throughput at the same occupancy

• Optimizations:
  – Coalesce address patterns per warp (nothing new here), consider texture
  – Process more words per thread (if insufficient accesses in flight to saturate bus)
  – Try the 4 combinations of L1 size and load type (caching and non-caching)
  – Consider compression
Optimizations for Instruction Throughput
Possible Limiting Factors

• **Raw instruction throughput**
  – Know the kernel instruction mix
  – fp32, fp64, int, mem, transcendental, etc. have different throughputs
    • Refer to the CUDA Programming Guide / Best Practices Guide
  – Can examine assembly, if needed:
    • Can look at PTX (virtual assembly), though it’s not the final optimized code
    • Can look at post-optimization machine assembly for GT200 (Fermi version coming later)

• **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

• Profiler counters (both incremented by 1 per warp):
  – instructions executed: counts instructions encountered during execution
  – instructions issued: also includes additional issues due to serialization
  – Difference between the two: issues that happened due to serialization, instr cache misses, etc.
    • Will rarely be 0, cause for concern only if it’s a significant percentage of instructions issued

• Compare achieved throughput to HW capabilities
  – Peak instruction throughput is documented in the Programming Guide
  – Profiler also reports throughput:
    • GT200: as a fraction of theoretical peak for fp32 instructions
    • Fermi: as IPC (instructions per clock)
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 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: Profiler Analysis

• Serialization is significant if
  – instructions\_issued is significantly higher than instructions\_executed

• 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

• SMEM bank conflicts
  – Profiler counters:
    • l1\_shared\_bank\_conflict: incremented by 1 per warp for each replay
      – double counts for 64-bit accesses
    • shared\_load, shared\_store: incremented by 1 per warp per instruction
  – Bank conflicts are significant if both are true:
    • instruction throughput affects performance
    • l1\_shared\_bank\_conflict is significant compared to instructions\_issued
Serialization: Analysis with Modified Code

- **Modify kernel code to assess performance improvement if serialization were removed**
  - Helps decide whether optimizations are worth pursuing

- **Shared memory bank conflicts:**
  - Change indexing to be either broadcasts or just `threadIdx.x`
  - Should also declare smem variables as volatile
    - Prevents compiler from “caching” values in registers

- **Warp divergence:**
  - change the condition to always take the same path
  - Time both paths to see what each costs
Serialization: Optimization

• Shared memory bank conflicts:
  – Pad SMEM arrays
    • For example, when a warp accesses a 2D array’s column
    • See CUDA Best Practices Guide, Transpose SDK whitepaper
  – Rearrange data in SMEM

• 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)
Case Study: SMEM Bank Conflicts

• A different climate simulation code kernel, fp64
• Profiler values:
  – Instructions:
    • Executed / issued: 2,406,426 / 2,756,140
    • Difference: 349,714 (12.7% of instructions issued were “replays”)
  – GMEM:
    • Total load and store transactions: 170,263
    • Instr:byte ratio: 4
      – suggests that instructions are a significant limiter (especially since there is a lot of fp64 math)
  – SMEM:
    • Load / store: 421,785 / 95,172
    • Bank conflict: 674,856 (really 337,428 because of double-counting for fp64)
      – This means a total of 854,385 SMEM access instructions, (421,785 + 95,172 + 337,428), 39% replays
• Solution:
  – Pad shared memory array: performance increased by 15%
    • replayed instructions reduced down to 1%
Instruction Throughput: Summary

• Analyze:
  – Check achieved instruction throughput
  – Compare to HW peak (but must take instruction mix into consideration)
  – Check percentage of instructions due to serialization

• Optimizations:
  – Intrinsics, compiler options for expensive operations
  – Group threads that are likely to follow same execution path
  – Avoid SMEM bank conflicts (pad, rearrange data)
Optimizations for Latency
Latency: Analysis

• Suspect if:
  – Neither memory nor instruction throughput rates are close to HW theoretical rates
  – Poor overlap between mem and math
    • Full-kernel time is significantly larger than max{mem-only, math-only}

• 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

- **Memory-only time**
- **Math-only time**
- **Full-kernel time**, one large threadblock per SM

Kernel where most math cannot be executed until all data is loaded by the threadblock.
Simplified View of Latency and Syncs

- **Memory-only time**
- **Math-only time**

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

- **Full-kernel time, one large threadblock per SM**
- **Full-kernel time, two threadblocks per SM** (each half the size of one large one)
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 “pipelining” blocks
    • 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”
Register Spilling
Register Spilling

- Compiler “spills” registers to local memory when register limit is exceeded
  - Fermi HW limit is 63 registers per thread
  - Spills also possible when register limit is programmer-specified
    - Common when trying to achieve certain occupancy with -maxrregcount compiler flag or __launch_bounds__ in source
  - lmem is like gmem, except that writes are cached in L1
    - lmem load hit in L1 -> no bus traffic
    - lmem load miss in L1 -> bus traffic (128 bytes per miss)
  - Compiler flag –Xptxas –v gives the register and lmem usage per thread

- Potential impact on performance
  - Additional bandwidth pressure if evicted from L1
  - Additional instructions
  - Not always a problem, easy to investigate with quick profiler analysis
Register Spilling: Analysis

• Profiler counters: l1_local_load_hit, l1_local_load_miss

• Impact on instruction count:
  – Compare to total instructions issued

• Impact on memory throughput:
  – Misses add 128 bytes per warp
  – Compare 2*l1_local_load_miss count to gmem access count (stores + loads)
    • Multiply lmem load misses by 2: missed line must have been evicted -> store across bus
    • Comparing with caching loads: count only gmem misses in L1
    • Comparing with non-caching loads: count all loads
Optimization for Register Spilling

• Try increasing the limit of registers per thread
  – Use a higher limit in `--maxrregcount`, or lower thread count for `__launch_bounds__`
  – Will likely decrease occupancy, potentially making gmem accesses less efficient
  – However, may still be an overall win - fewer total bytes being accessed in gmem

• Non-caching loads for gmem
  – potentially fewer contentions with spilled registers in L1

• Increase L1 size to 48KB
  – default is 16KB L1 / 48KB smem
Register Spilling: Case Study

• FD kernel, (3D-cross stencil)
  – fp32, so all gmem accesses are 4-byte words
    • Need higher occupancy to saturate memory bandwidth
  – Coalesced, non-caching loads
    • one gmem request = 128 bytes
    • all gmem loads result in bus traffic
  – Larger threadblocks mean lower gmem pressure
    • Halos (ghost cells) are smaller as a percentage

• Aiming to have 1024 concurrent threads per SM
  – Means no more than 32 registers per thread
  – Compiled with –maxrregcount=32
Case Study: Register Spilling 1

• 10\textsuperscript{th} order in space kernel (31-point stencil)
  – 32 registers per thread: 68 bytes of lmem per thread: upto 1024 threads per SM

• Profiled counters:
  – \texttt{l1\_local\_load\_miss} = 36 \hspace{1cm} \texttt{inst\_issued} = 8,308,582
  – \texttt{l1\_local\_load\_hit} = 70,956 \hspace{1cm} \texttt{gld\_request} = 595,200
  – \texttt{local\_store} = 64,800 \hspace{1cm} \texttt{gst\_request} = 128,000

• Conclusion: spilling is not a problem in this case
  – The ratio of gmem to lmem bus traffic is approx 8,444 : 1 (hardly any bus traffic is due to spills)
    • L1 contains most of the spills (99.9\% hit rate for lmem loads)
  – Only 1.6\% of all instructions are due to spills
Case Study: Register Spilling 2

• 12th order in space kernel (37-point stencil)
  – 32 registers per thread: 80 bytes of lmem per thread: upto 1024 threads per SM

• Profiled counters:
  – l1_local_load_miss = 376,889 inst_issued = 10,154,216
  – l1_local_load_hit = 36,931 gld_request = 550,656
  – local_store = 71,176 gst_request = 115,200

• Conclusion: spilling is a problem in this case
  – The ratio of gmem to lmem bus traffic is approx 7:6 (53% of bus traffic is due to spilling)
    • L1 does not contain the spills (8.9% hit rate for lmem loads)
  – Only 4.1% of all instructions are due to spills

• Solution: increase register limit per thread
  – 42 registers per thread: no spilling: upto 768 threads per SM
  – Single 512-thread block per SM: 13% perf increase
  – Three 256-thread blocks per SM: 37% perf increase
Register Spilling: Summary

• Doesn’t always decrease performance, but when it does it’s due to:
  – Increased pressure on the memory bus
  – Increased instruction count

• Use the profiler to examine the impact by comparing:
  – $2 \times \text{l1\_local\_load\_miss}$ to all gmem accesses that don’t hit in L1
  – Local access count to total instructions issued

• Impact is significant if:
  – Memory-bound code: lmem misses are a significant percentage of total bus traffic
  – Instruction-bound code: lmem accesses are a significant percentage of all instructions
Summary

- **Determining what limits your kernel most:**
  - Arithmetic, memory bandwidth, latency

- **Address the bottlenecks in the order of importance**
  - Analyze for inefficient use of hardware
  - Estimate the impact on overall performance
  - Optimize to most efficiently use hardware

- **More resources:**
  - Fundamental Optimizations talk
  - Prior CUDA tutorials at Supercomputing
    - [http://gpgpu.org]{sc2007,sc2008,sc2009}
  - CUDA webinars
Questions?