Identifying Performance Limiters

Paulius Micikevicius | NVIDIA

“Re-presented” by Gregory Kesden, 15-418, Spring 2018

© NVIDIA 2011
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
Algorithmic Analysis

• Approach:
  – Compute the ratio of arithmetic operations to bytes accessed in the algorithm (for example, per output element)
  – Compare to the balanced ratio for your GPU

• Better than nothing, but not very accurate:
  – Undercounts instructions: control flow, address calculation, etc.
  – May undercount memory accesses: ignores cache line sizes

• Example: vector add
  – Read two 4-byte words, add, write one 4-byte word
  – 1 instr : 12 bytes
  – Much lower than 3.76:1, thus memory bound
Analysis with the Profiler

• **Relevant profiler counters:**
  - `instructions_issued`
    - Incremented by 1 per warp, counter is for one SM
  - `dram_reads`, `dramWrites`
    - Incremented by 1 per 32B access to DRAM
    - Note that the VisualProfiler converts each of the above to 2 counters
      - These simply get added together, refer to the Visual Profiler User Guide for details
      - You’ll need to do this yourself if you’re using command-line profiling
  - If your code hits in L2 cache a lot, you may want to look at L2 counters instead (accesses to L2 are still expensive compared to arithmetic)

• **Compute instruction:byte ratio and compare to the balanced one:**
  - \((\text{number of SMs}) \times 32 \times \text{instructions}_\text{issued} : 32\text{B} \times (\text{dram}_\text{reads} + \text{dram}_\text{writes})\)

• **Example: vector add**
  - \(1.49:1\), lower than \(3.76\) so memory-bound
Another Way to Use the Profiler

• **VisualProfiler** will report 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
Another Way to Use the Profiler

- VisualProfiler will report in different ways:
  - IPC (instructions per clock) for instructions
  - GB/s achieved for memory (in B/s)
- Compare those with the theoretical
  - Profiler will also report the theoretical
    - Though for IPC it assumes flat memory access
      - If one of the metrics is close
        - If neither metric is close to
          - “close” is approximate, I’d say
- Example: vector add
  - IPC: 0.55 out of 2.0
  - Memory throughput: 130 G B/s
  - Conclusion: memory bound
Notes on Instruction Counts

• Undercount by algorithmic analysis
  – Algorithmic analysis assumed 1 instruction (add)
  – Actual code contains 17 instructions

• You can actually check the machine-language assembly instructions
  – Compile into a .cubin file
  – Use cuobjdump tool (comes with CUDA toolkit) to get assembly from .cubin
  – Useful for checking instruction counts
  – Actual instruction counts could also be used to somewhat refine the theoretical IPC for the specific code
    • For example, if all instructions were fp64, the theoretical IPC is 1.0, not 2.0
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
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/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)

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

© NVIDIA 2011
Source Modification

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

```c
__global__ void add( float *output, float *A, float *B, int flag )
{
    ...
    value = A[idx] + B[idx];
    if( 1 == value * flag )
        output[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

• 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

  \[
  \text{kernel} \llll \text{grid, block, smem, ...} \rrrr (\ldots)
  \]
Another Case Study

• Time (ms):
  – Full-kernel: 25.82
  – Mem-only: 23.53
  – Math-only: 12.52

• Instructions issued:
  – Full-kernel: 20,388,591
  – Mem-only: 10,034,799
  – Math-only: 14,683,776

• Total DRAM requests
  – Full-kernel: 101,328,372
  – Mem-only: 101,328,372
  – Math-only: 0

• Analysis:
  – Instr:byte ratio = 3.21
  – Good overlap between math and mem:
    • 2.29 ms of math-only time (18%) is not overlapped with mem
  – App memory throughput: 72 GB/s
    • HW throughput is 125 GB/s
    • HW theory is 177 GB/s, so memory is not used efficiently

• Conclusion:
  – Code is more memory- than instruction-limited
    • IPC is 1.2 (60% of theory)
    • Memory throughput is 70%
  – Optimizations should focus on memory throughput first
    • Memory is a larger portion of total time
    • Also note that application and hw throughputs are different
      – More on this in upcoming webinar
Summary

• Rough algorithmic analysis:
  – How many bytes needed, how many instructions
• Profiler analysis:
  – Instruction count, memory access count
  – Check how close instruction and memory throughputs are to hw theory
• Analysis with source modification:
  – Full version of the kernel
  – Memory-only version of the kernel
  – Math-only version of the kernel
  – Examine how these times relate and overlap
• More details on memory- and instruction-optimizations
  – Upcoming webinars
Local Memory

• **Name refers to memory where registers and other thread-data is spilled**
  – Usually when one runs out of SM resources
  – “Local” because each thread has its own private area

• **Details:**
  – Not really a “memory” - bytes are stored in global memory
  – Differences from global memory:
    • Addressing is resolved by the compiler
    • Stores are cached in L1
LMEM Access Operation

• A store writes a line to L1
  – If evicted, that line is written to L2
  – The line could also be evicted from L2, in which case it’s written to DRAM

• A load requests the line from L1
  – If a hit, operation is complete
  – If a miss, then requests the line from L2
    • If a miss, then requests the line from DRAM

• A store always happens before a load
  – Only GPU threads can access LMEM addresses
Fermi Memory Hierarchy

- SM-0
  - Registers
  - L1
  - SMEM

- SM-1
  - Registers
  - L1
  - SMEM

- SM-N
  - Registers
  - L1
  - SMEM

- L2

- Global Memory (DRAM)
When is Local Memory Used?

- **Register spilling**
  - Fermi hardware limit is **63 registers** per thread
  - Programmer can specify lower registers/thread limits:
    - To increase occupancy (number of concurrently running threads)
    - `-maxrregcount` option to `nvcc`, `__launch_bounds`() qualifier in the code
  - LMEM is used if the source code exceeds register limit

- **Arrays declared inside kernels, if compiler can’t resolve indexing**
  - Registers aren’t indexable, so have to be placed in LMEM
How Does LMEM Affect Performance?

• It could hurt performance in two ways:
  – Increased memory traffic
  – Increased instruction count

• Spilling/LMEM usage isn’t always bad
  – LMEM bytes can get contained within L1
    • Avoids memory traffic increase
  – Additional instructions don’t matter much if code is not instruction-throughput limited
General Analysis/Optimization Steps

• **Check for LMEM usage**
  – Compiler output
    • `nvcc` option: `-Xptxas -v,-abi=no`
    • Will print the number of **lmem bytes** for each kernel (only if kernel uses LMEM)
  – Profiler

• **Check the impact of LMEM on performance**
  – Bandwidth-limited code:
    • Check how much of L2 or DRAM traffic is due to LMEM
  – Arithmetic-limited code:
    • Check what fraction of instructions issued is due to LMEM

• **Optimize:**
  – Try: increasing register count, increasing L1 size, using non-caching loads
Register Spilling: Analysis

• **Profiler counters:**
  – l1\_local\_load\_hit, l1\_local\_load\_miss, l1\_local\_store\_hit, l1\_local\_store\_miss
  – Counted for a single SM, incremented by 1 for each 128-byte transaction

• **Impact on memory**
  – Any memory traffic that leaves SMs (goes to L2) is expensive
  – L2 counters of interest: read and write sector queries
    • Actual names are longer, check the profiler documentation
    • Incremented by 1 for each 32-byte transaction
  – Compare:
    • Estimated L2 transactions due to LMEM misses in all the SMs
      – \(2 \times \text{(number of SMs)} \times 4 \times \text{l1\_local\_load\_miss}\)
        • 2: load miss implies a store happened first
        • Number of SMs: l1\_local\_load\_miss counter is for a single SM
        • 4: local memory transaction is 128-bytes = 4 L2-transactions
    • Sum of L2 read and write queries (not misses)

• **Impact on instructions**
  – Compare the sum of all LMEM instructions to total instructions issued
Optimizations When Register Spilling is Problematic

• **Try increasing the limit of registers per thread**
  – Use a higher limit in `--maxrregcount`, or lower thread count for `__launch_bounds__`
  – Likely reduces occupancy, potentially reducing execution efficiency
    • may still be an overall win - fewer total bytes being accessed

• **Try using non-caching loads for global memory**
  – nvcc option: `-Xptxas -dlcm=cg`
  – Potentially fewer contentions with spilled registers in L1

• **Increase L1 size to 48KB**
  – Default is 16KB L1, larger L1 increases the chances for LMEM hits
  – Can be done per kernel or per device:
    • `cudaFuncSetCacheConfig()`, `cudaDeviceSetCacheConfig()`
Case Study

• **Time Domain Finite Difference of the 3D Wave Equation**
  – Simulates seismic wave propagation through Earth subsurface
  – Largely memory bandwidth-bound
  – Running more threads concurrently helps saturate memory bandwidth
    • Thus, to run 1024 threads per Fermi SM we specify 32 register maximum per thread

• **Check for LMEM Use**
  – Spills 44 bytes per thread when compiled down to 32 registers per thread

```bash
$ nvcc -arch=sm_20 -Xptxas -v,-abi=no,-dlcm=cg fwd_o8.cu -maxrregcount=32
ptxas info : Compiling entry function '_Z15fwd_3D_orderX2bILi4ELi9EEvPfS0_S0_iiii' for 'sm_20'
ptxas info : Used 32 registers, 44+0 bytes lmem, 6912+0 bytes smem, 76 bytes cmem[0], ...
```
Case Study: Analyze the Impact on Memory

• Using profiler counters:
  – SM counters:
    • l1_local_load_miss: 564,332
    • l1_local_load_hit: 91,520
    • l1_local_store_miss: 269,215
    • l1_local_store_hit: 13,477
    • inst_issued: 20,412,251
  – L2 query counts: 99,435,608
    • Read: 33,385,908
    • Write: 132,821,516
    • Total:

• This was on a 16-SM GPU

To get the counters use any of:
• Visual Profiler
• Command-line profiler
• NSight
Case Study: Analyze the Impact on Memory

• Using profiler counters:
  – SM counters:
    • l1_local_load_miss: 564,332
    • l1_local_load_hit: 91,520
    • l1_local_store_miss: 269,215
    • l1_local_store_hit: 13,477
    • inst_issued: 20,412,251
  
  – L2 query counts:
    • Read: 33,385,908
    • Write: 132,821,516
    • Total: 165,207,424

  Load L1 hit rate: 13.95%
  Estimated L2 queries per SM due to LMEM:
  \[2*4*564,332 = 4,514,656\]

  Estimated L2 queries due to LMEM of all 16 SMs:
  \[16*4,514,656 = 72,234,496\]
  Percentage of all L2 queries due to LMEM:
  \[72,234,496 / 132,821,516 = 53.38\%

• This was on a 16-SM GPU
Case Study: Analyze the Impact on Memory

• Using profiler counters:
  – SM counters:
    • Using a dual-GPU:
      564,332
      91,520
      269
      21
      5
      13,477
      99,435,608
      33,385,908
      132,821,516
      20,412,251
  – L2 query counts:
    • Read: 33,385,908
    • Write: 132,821,516
    • Total: 99,435,608

53.38% of memory traffic between the SMs and L2/DRAM is due to LMEM (not useful from the application’s point of view).

Since application is bandwidth-limited, reducing spilling could help performance.

Load L1 hit rate: 13.95%
Estimated L2 queries per SM due to LMEM:
\[2 \times 4 \times 564,332 = 4,514,656\]

Estimated L2 queries due to LMEM of all 16 SMs:
\[16 \times 4,514,656 = 72,234,496\]

Percentage of all L2 queries due to LMEM:
\[72,234,496 / 132,821,516 = 53.38\%\]

This was on a 16-SM GPU
Case Study: Analyze the Impact on Instructions

- Using profiler counters:
  - SM counters:
    - l1_local_load_miss: 564,332
    - l1_local_load_hit: 91,520
    - l1_local_store_miss: 269,215
    - l1_local_store_hit: 13,477
    - inst_issued: 20,412,251
  - L2 query counts: 99,435,608
    - Read: 33,385,908
    - Write: 132,821,516
    - Total:

Total instructions due to LMEM: 938,944
Percentage of instructions due to LMEM: 938,944 / 20,412,251 = 4.60%

• This was on a 16-SM GPU
Case Study: Analyze the Impact on Instructions

- Using profiler counters:
  - SM counters:
    - l1_local_load_miss: 564,332
    - l1_local_load_hit: 91,520
    - l1_local_store_miss: 269,215
    - l1_local_store_hit: 13,477
    - inst_issued: 20,412,251
  - L2 query counts: 99,435,608
    - Read: 33,385,908
    - Write: 132,821,516
    - Total:

Total instructions due to LMEM: 938,944
Percentage of instructions due to LMEM: 938,944 / 20,412,251 = 4.60%

- This was on a 16-SM GPU
Case Study: Optimizations

• **Try increasing register count**
  – Remove the `-maxrregcount=32` compiler option
    • 46 registers per thread, no spilling
  – Performance improved by **1.22x**

• **Increase L1 cache size**
  – Keeping the 32 register maximum and spilling 44 bytes
  – Add `cudaDeviceSetCacheConfig( cudaFuncCachePreferL1 );` call
  – L1 LMEM load hit rate improved to **98.32%**
  – Estimated **1.63%** of all requests to L2 were due to LMEM
    • way too small to worry about
    • 1.63 was computed as on slide 12 (not by 100% - 98.32%)
  – performance improved by **1.45x**

• **Application was already using non-caching loads for other reasons**
Register Spilling: Summary

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

• Use the profiler to determine:
  – Bandwidth-limited codes: LMEM L1 miss impact on memory bus (to L2) for
  – Arithmetic-limited codes: LMEM instruction count as percentage of all instructions

• Optimize by
  – Increasing register count per thread
  – Increasing L1 size
  – Using non-caching GMEM loads