Identifying Performance Limiters
Paulius Micikevicius | NVIDIA
August 23, 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:
  – (number of SMs) * 32 * instructions_issued : 32B * (dram_reads + dram_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:
  - IPC (instructions per clock)
  - 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,
    - If one of the metrics is close to the HW peak,
    - If neither metric is close to the peak,
    - “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 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/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 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
____global____ void add( float *output, float *A, float *B, int flag)
{
    ...
    value = A[idx] + B[idx];
    if( 1 == value * flag )
        output[idx] = value;
}
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

  kernel<<< grid, block, smem, ...>>>(...)

© NVIDIA 2011
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
Questions?