Instruction Limited Kernels

CUDA Optimization Webinar

Gernot Ziegler,
Developer Technology (Compute)

© NVIDIA Corporation 2011

Analysis-Driven Optimization

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

© NVIDIA Corporation 2011

Presentation Outline

- **Identifying performance limiters**
- **Analyzing and optimizing :**
  - Memory-bound kernels (Oct 11th, Tim Schroeder)
  - Instruction (math) bound kernels
  - Kernels with poor latency hiding
  - Register spilling (Oct 4th, Paulius Micikevicius)
- **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**

© NVIDIA Corporation 2011

Notes on profiler

- Most counters are reported per Streaming Multiprocessor (SM)
  - Not entire GPU
  - Exceptions: L2 and DRAM counters
- A single run can only 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
    - Use CUPTI API to have your application collect signals on its own
- 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

© NVIDIA Corporation 2011

Limited by Bandwidth or Arithmetic?

NVIDIA

- **Perfect fp32 instructions:bytes ratio for Fermi C2050:**
  - ~4.5 : 1 instructions/byte with ECC on
  - ~3.6 : 1 instructions/byte with ECC off
  - These assume fp32 instructions, throughput for other instructions varies
- **Algorithmic analysis:**
  - Rough estimate of arithmetic to bytes ratio
- **Actual Code likely uses more instructions and bytes than algorithmic analysis suggests:**
  - Instructions for loop control, pointer math, etc.
  - Address pattern may result in more memory transactions/bandwidth
  - Two ways to investigate:
    - **Use the profiler (quick, but approximate)**
    - Use source code modification (more accurate, more work intensive)

© NVIDIA Corporation 2011

CUDA 4.0: Visual Profiler Optimization Hints

- Visual Profiler gathers hardware signals via command-line profiler
- Profiler computes derived values:
  - Instruction throughput
  - Memory throughput
  - GPU Occupancy

  (Computation detailed in documentation)
- Using these derived values, profiler *hints* at limiting factors
  This talk shows the thoughts behind Profiler hints, but also how to interpret GPU profiler signals while doing your own experiments, e.g. through source-code modifications

convolutionColumnsKernel analysis - [Session4 - Device_0 - Context_0]

File View

Analysis

Instruction Throughput Analysis for kernel convolutionColumnsKernel on device GeForce GTX 480

- IPC: 1.56
- Maximum IPC: 2
- Divergent branches(%): 0.00
- Control flow divergence(%): 0.03
- Replayed Instructions(%): 29.65
  - Global memory replay(%): 0.00
  - Local memory replays(%): 0.00
  - Shared bank conflict replay(%): 26.38
- Shared memory bank conflict per shared memory instruction(%): 99.90

Hint(s)

- **The kernel is compute bound**, to reduce instruction count
  - Understand the instruction mix, as single precision floating point, double precision floating point, int, mem, transcendentals, etc. have different throughputs. Use double precision arithmetic only when required (E.g. floating point literals without an f suffix (34.7) are interpreted as double precision as per C standard);
  - Try using arithmetic intrinsic functions.
  - Try using compiler flags(-ftz=true, -prec-div=false, -prec-sqrt=false etc) to get higher performance, but may result in some precision loss;

  Refer to the "Arithmetic Instructions" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.
- **Shared memory bank conflicts are high** which causes serialization of threads within a warp. Shared memory bank conflicts can be reduced by
  - Using appropriate padding for data stored in shared memory so that each thread in a warp accesses data from a different bank;
  - Rearranging data in shared memory, thus changing access pattern;

  Refer to the "Shared Memory" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.

Factors that may affect analysis

Limiting Factor Identification | Memory Throughput Analysis | Instruction throughput Analysis | Occupancy Analysis

Show all columns

	GPU Timestamp (us)	GPU Time (us)	shared load Type:SM Run:4	shared store Type:SM Run:4
1	38718	1652.96	334560	24600
2	41989.6	1652.86	334560	24600
3	44507.4	1652.93	334560	24600
4	47024.9	1652.96	334560	24600
5	49541.9	1653.09	334560	24600

© NVIDIA Corporation 2011

Optimizations for Instruction Throughput

© NVIDIA Corporation 2011

Possible Limiting Factors

- **Raw instruction throughput**
  - Know the kernel instruction mix
  - fp32, fp64, int, mem, transcendentals 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 (--dump-sass, via cuobjdump)
- **Instruction serialization ("instruction replays" for warp's threads)**
  - Occurs when threads in a warp execute/issue the same instruction *after* each other instead of in parallel
    - Think of it as "replaying" the same instruction for different threads in a warp
  - Some causes:
    - Shared memory bank conflicts
    - Constant memory bank conflicts

© NVIDIA Corporation 2011

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: instruction issues that happened due to serialization, instruction cache misses, etc.
    - Will rarely be 0, 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).
      Note that theoretical maximum depends on instruction mix:
      e.g. fp32 only: IPC of 2.0 possible; fp64 instructions: IPC of 1.0 possible

© NVIDIA Corporation 2011

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

© NVIDIA Corporation 2011

Serialization: Profiler Analysis

- **Serialization is significant if**
  - instructions_issued is significantly higher than instructions_executed
  - CUDA 4.0 Profiler: Instructions replayed %
- **Warp divergence (Warp has to execute both branch of if())**
  - Profiler counters: divergent_branch, branch
    Profiler derived: Divergent branches (%)).
  - However, only counts the branch instructions, not the rest of divergent instructions.
  - Better: 'threads instruction executed' counter:
    Increments for every instruction by number of threads that executed the instruction.
  - If there is no divergence, then for every instruction it should increment by 32
    (and threads_instruction_executed = 32* instruction_executed)
  - Thus: Control_flow_divergence% =
    100 * ((32 * instructions executed) – threads instruction executed)/(32* instructions executed)

© NVIDIA Corporation 2011

Serialization: Profiler Analysis

- **SMEM bank conflicts**
  - Profiler counters:
    - l1_shared_bank_conflict
      - incremented by 1 per warp for each replay
        (or: each n-way shared bank conflict increments by n-1)
      - double increment 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:
      - Shared bank conflict replay (%) =
        100 * (l1_shared_bank_conflict)/instructions_issued
      - Shared memory bank conflict per shared memory instruction (%) =
        100 * (l1 shared bank conflict)/(shared load + shared store)

© NVIDIA Corporation 2011

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 if-condition to have all threads take the same path
  - Time both paths to see what each costs

© NVIDIA Corporation 2011

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 into same warp
    - Rearrange the data, pre-process the data
    - Rearrange how threads index data (may affect memory perf)

© NVIDIA Corporation 2011

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) , of which 39% replays
- **Solution: Pad shared memory array**

  Performance increased by 15%
    - replayed instructions reduced down to 1%

© NVIDIA Corporation 2011

Instruction Throughput: Summary

- **Analyze:**
  - Check achieved instruction throughput
  - Compare to HW peak (note: 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)

© NVIDIA Corporation 2011

Optimizations for Latency

© NVIDIA Corporation 2011

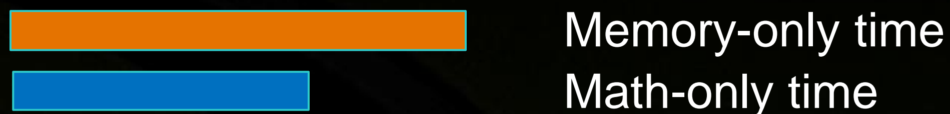
Latency: Analysis

- **Suspect latency issues 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
      - Indicator: 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

© NVIDIA Corporation 2011

Simplified View of Latency and Syncs

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



Memory-only time
Math-only time

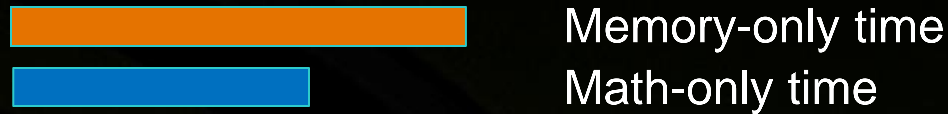
Full-kernel time, one large threadblock per SM

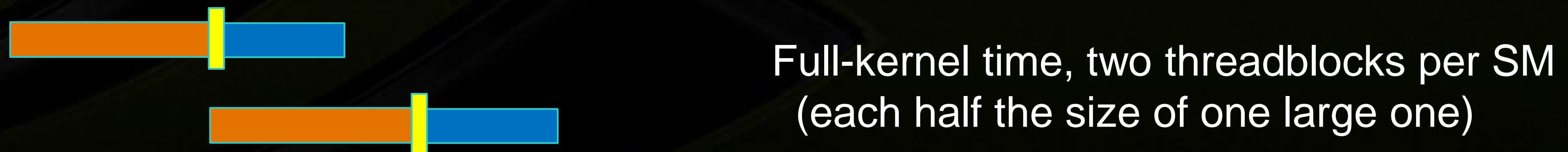
time →

© NVIDIA Corporation 2011

Simplified View of Latency and Syncs

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





time

© NVIDIA Corporation 2011

Latency: Optimization

- **Insufficient threads or workload:**
  - Best: Increase the level of parallelism (more threads)
  - Alternative: Process several output elements per thread – gives more independent memory and arithmetic instructions (which get pipelined) - downside: code complexity
- **Synchronization Barriers:**
  - Can assess impact on perf by commenting out __syncthreads()
    - Incorrect result, but gives upper bound on improvement
  - Try running several smaller threadblocks
    - Less hogging of SMs; think of it as SM "pipelining" blocks
    - In some cases that costs extra bandwidth due to more halos
- **More information and tricks:**
  - Vasily Volkov, GTC2010: "Better Performance at Lower Occupancy" http://www.gputechconf.com/page/gtc-on-demand.html#session2238

© NVIDIA Corporation 2011

Summary

NVIDIA

- **Determining what limits your kernel most:**
  - Arithmetic, memory bandwidth (Oct 11th), latency, register spilling (Oct 4th)
- **Address the bottlenecks in the order of importance**
  - *Analyze* for inefficient use of hardware
  - *Estimate* the impact on overall performance
  - *Optimize* to use hardware most efficiently
- **More resources:**
  - Profiler documentation! (Especially on derived profiler values)
  - Prior CUDA tutorials at Supercomputing
    - http://gpgpu.org/{sc2007,sc2008,sc2009,sc2010}
  - GTC2010 talks: http://www.nvidia.com/gtc2010-content
  - CUDA Programming Guide, CUDA Best Practices Guide

© NVIDIA Corporation 2011