PMPP 2015/16

TECHNISCHE UNIVERSITÄT DARMSTADT

Performance Tuning (2)

Course Schedule

TECHNISCHE UNIVERSITÄT DARMSTADT

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Lecture Parallel Basics, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A), S3|19, Room 2.8
2.11.2015	Intro Final Proj., Ex. 1 due, Ex. 2 assigned, Lecture PRAM
3.11.2015	Lecture PRAM (2)
9.11.2015	Final Projects assigned, L. Parallel Sort., Exercise 2 due
10.11.2015	Questions and Answers (Q&A)
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)
30.11.2015	Lecture Design Patterns
1.12.2015	Questions and Answers (Q&A)

(Preliminary) Course Schedule

7.12.2015	Lecture Design Patterns (2), Performance Tuning
8.12.2015	Questions and Answers (Q&A)
14.12.2015	Performance Tuning (2)
15.12.2015	Questions and Answers (Q&A)
11.1.2016	2nd Status Presentation Final Projects
12.1.2016	2nd Status Presentation Final Projects (continued)
18.1.2016	Performance Tuning (3)
19.1.2016	
25.1.2016	
26.1.2016	
1.2.2016	
2.2.2016	
8.2.2016	Final Presentation Final Projects
9.2.2016	Final Presentation Final Projects (continued)

Overview

TECHNISCHE UNIVERSITÄT DARMSTADT

- Instruction Level Parallelism
- Identifying Performance Limiters
  - Instruction Limited Kernels
  - Latency Limited Kernels
  - Bandwidth Limited Kernels
  - Local Memory Optimizations
- Multi GPU Multi Threading
  - Brief review of the scenarios
  - Single CPU process, multiple GPUs
    - GPU selection, UVA, P2P
  - Multiple processes
    - Needs CPU-side message passing
  - Dual-IOH CPU systems and NUMA

Images and slides partially by developer.nvidia.com

Performance Optimization Process

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- Algorithmic
  - Based on algorithm's memory and arithmetic requirements
  - Least accurate: undercounts instructions and potentially memory accesses
- Profiler (nvvp)
  - 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- Theoretical memory throughput
  - For example, Tesla M2090/GTX480 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- Relevant profiler counters:
  - instructions_issued
    - Incremented by 1 per warp, counter is for one SM
  - dram_reads, dram_writes
    - 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

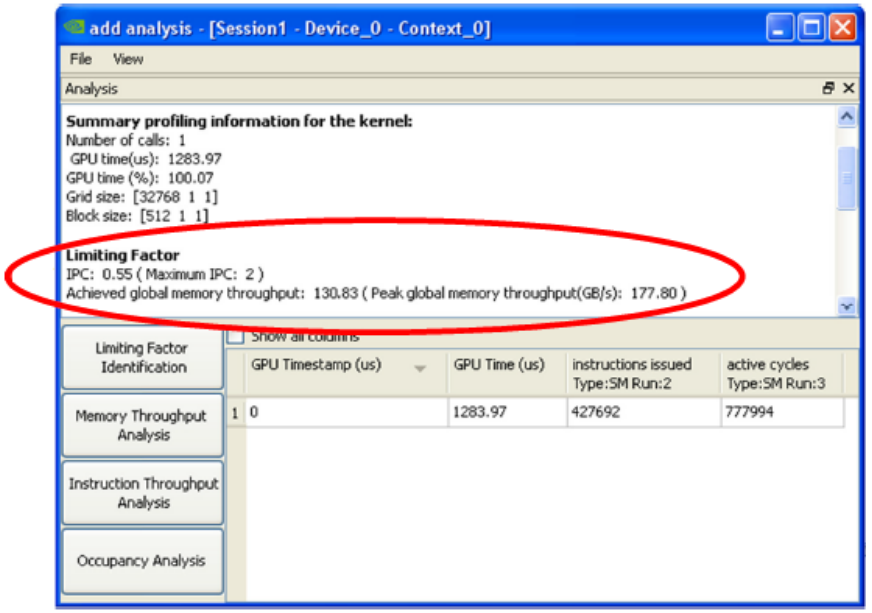
TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



GCC Graphics, Capture and Massively Parallel Computing

Notes on Instruction Counts

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

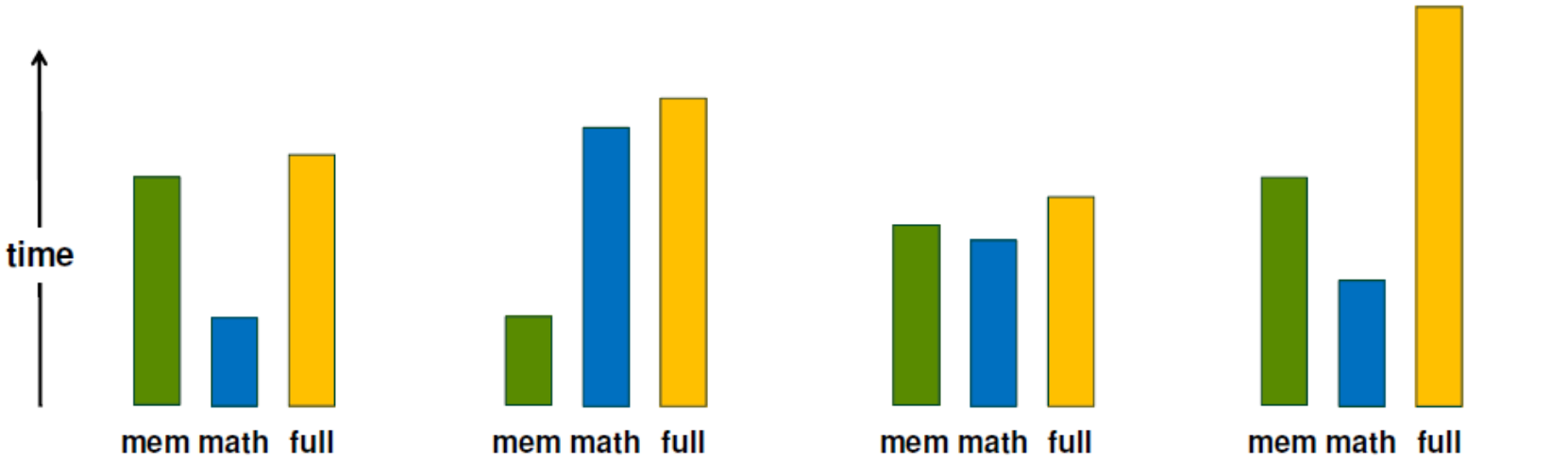
- Most counters are reported per Streaming Multiprocessor (SM)
  - Not entire GPU
  - Exceptions: L2 and DRAM counters
- A single run can collect a few counters
  - Multiple runs are needed when profiling more counters
    - Done automatically by the Visual Profiler
    - Have to be done manually using 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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)

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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

```
__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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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, ...>>>(...)

Another Case Study

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

Review

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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
  - following

Instruction Throughput Limited Kernel

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

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

Instruction Throughput: Optimization

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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)

Serialization: Profiler Analysis

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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)

Serialization: Analysis with Modified Code

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

Serialization: Optimization

TECHNISCHE UNIVERSITÄT DARMSTADT

- Shared memory bank conflicts:
  - Pad SMEM arrays
    - For example, when a warp accesses a 2D arrays 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)

Case Study: SMEM Bank Conflicts

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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%

Instruction Throughput: Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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)

Instruction Latency Limited Kernel

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

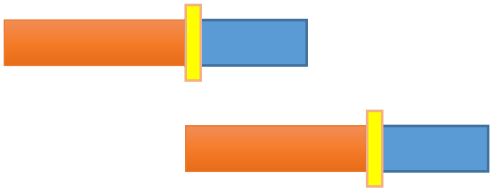
Simplified View of Latency and Syncs

Memory only time

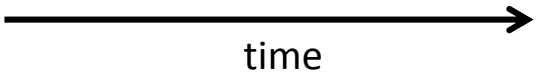
Math only time

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 threadblock per SM
(each half the size of one large one)



Latency: Optimization

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

- Determining what limits your kernel most:
  - Arithmetic
  - Memory bandwidth (next up)
  - Latency
  - Register spilling (still to come)
- 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