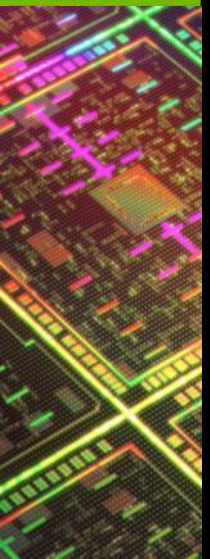
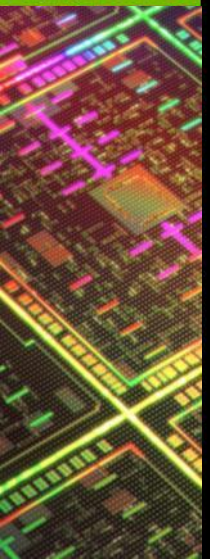
Analysis-Driven Optimizations in CUDA

Peng Wang, Developer Technology, NVIDIA



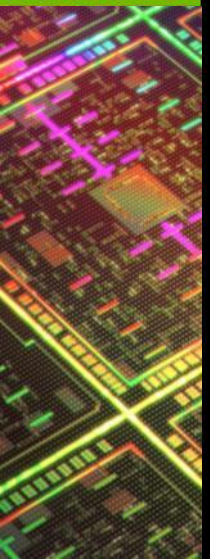
General Optimization Strategies: Measurement

- Find out the limiting factor in kernel performance
  - Memory bandwidth bound
  - Instruction throughput bound
  - Latency bound
  - Combination of the above
- Measure effective memory/instruction throughput
- Address the limiters in order of importance
  - Determine how close to the HW limits the resource is being used
  - Apply optimizations
- Typically an iterative process



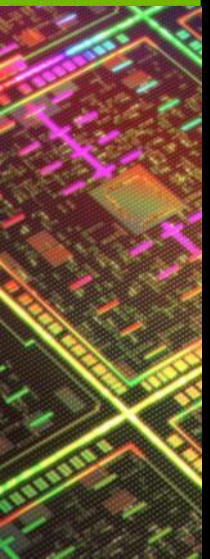
Overview

- Identifying performance limiters
- Memory optimization
- Latency optimization
- Instruction optimization
- Register spilling



Overview

- Identifying performance limiters
- Memory optimization
- Latency optimization
- Instruction optimization
- Register spilling

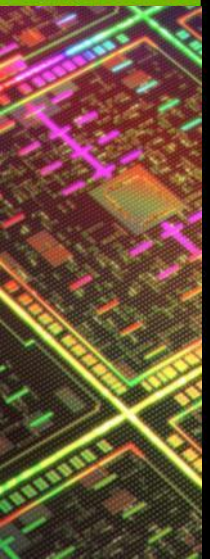


How to Do Measurement

- Automated
  - Visual profiler: use hardware counters
- Manual
  - Source modification

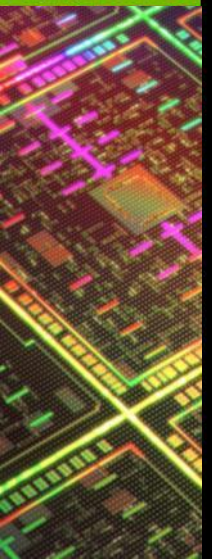
Notes on Visual Profiler

- **We will use Visual Profiler 4.0 for this talk**
- **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**



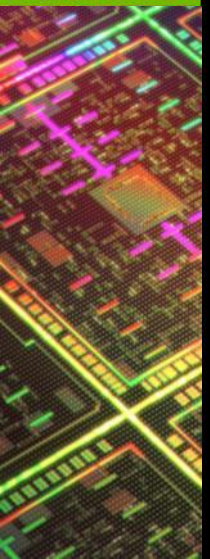
Limited by Bandwidth or Arithmetic?

- **Perfect instructions:bytes ratio for Fermi C2050:**
  - ~3.6 : 1 (515 Ginst/s / 144 GB/s) with ECC on
  - ~4.4 : 1 (515 Ginst/s / 120 GB/s) with ECC off
  - These assume fp32 instructions, throughput for other instructions varies
- **Algorithm 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



Analysis with Profiler

- **Profiler counters:**
  - instructions_issued, instructions_executed
    - Both incremented by 1 per warp
    - "issued" includes replays, "executed" does not
  - l2_read_requests, l2_write_requests, l2_read_texture_requests
    - Incremented by 1 for each 32B access
- **Compare:**
  - 32 * instructions_issued * #SM /* 32 = warp size */
  - 32B * (l2_read_requests+l2_write_requests+l2_read_texture_requests)
  - The ratio gives instruction/byte



Analysis with Modified Source Code

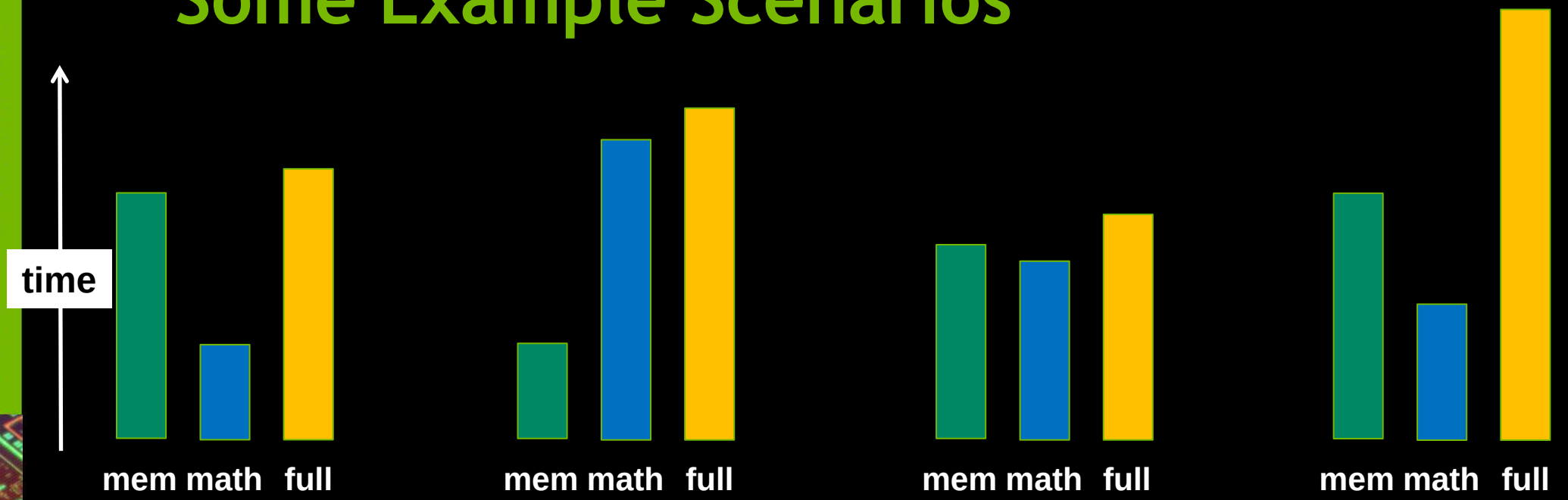
- **Memory-only:**
  - Remove as much arithmetic as possible
    - Without changing access pattern
    - Use the profiler to verify that load/store instruction count is the same
- **Store-only:**
  - Also remove the loads
- **Math-only:**
  - Remove global memory accesses
  - Need to trick the compiler:
    - 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

```
__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

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)

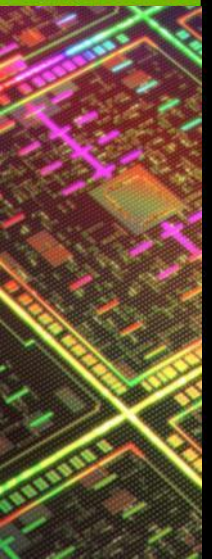
Balanced

Good mem-math overlap: latency not a problem

(assuming memory/instr throughput is not low compared to HW theory)

Memory and latency bound

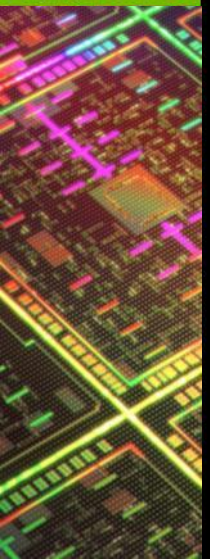
Poor mem-math overlap: latency is a problem



Source Modification and Occupancy

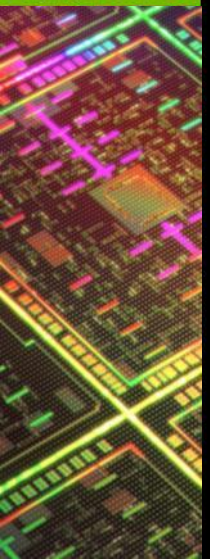
- **Removing pieces of code is likely to affect register count**
  - This could increase occupancy, skewing the results
  - See slide 23 to see how that could affect throughput
- **Make sure to keep the same occupancy**
  - Check the occupancy with profiler before modifications
  - After modifications, if necessary add shared memory to match the unmodified kernel's occupancy

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



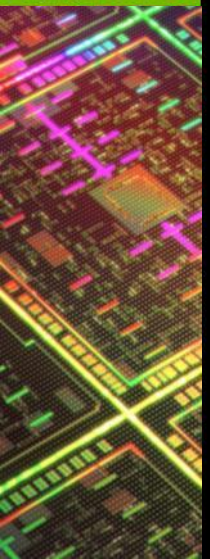
Summary: Limiter Analysis

- **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



Overview

- Identifying performance limiters
- Memory optimization
- Latency optimization
- Instruction optimization
- Register spilling

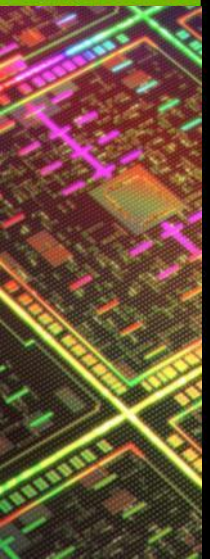


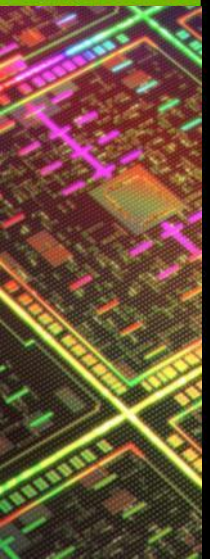
Memory Throughput Analysis

- **Throughput** metrics:
  - From APP point of view
    - Count bytes requested by the application
    - gld_8bit, gld_16bit, …, gld_128bit, gst_8bit, etc. SW counters
    - Kernel requested read throughput = (gld_8bit+…+16B*gld_128bit)/gputime
  - From HW point of view
    - Count bytes moved by the hardware
    - dram_reads, dram_writes
      - Incremented by 1 for each 32B access
    - Glob mem read throughput = (dram_reads)*32B/gputime
  - The two can be different
    - Scattered/misaligned pattern: not all transaction bytes are utilized
    - Broadcast: the same small transaction serves many requests

Access Pattern Analysis

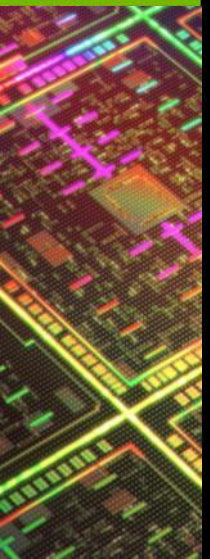
- **Determining that access pattern is problematic**
  - Kernel requested throughput << Glob mem throughput
    - The ratio is the amount of “wasted” memory bandwidth
- Look at source code
  - Array index with threadIdx.x*N (N > 1) would mean uncoalesced access





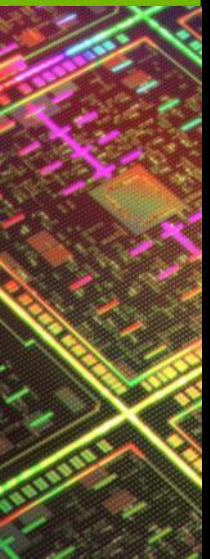
Access Pattern Optimizations

- Transpose the data structure, e.g. AOS to SOA
- 1-thread-per-task to 1-warp-per-task or 1-block-per-task
- Use shared memory to avoid/reduce uncoalesced access
- Non-caching loads
  - Smaller transaction size: 32B instead of 128B
- Bound to texture cache for unpredictable uncoalesced access
  - Small transaction size: 32B
  - Cache not polluted by other gmem loads
- Compression



Overview

- Identifying performance limiters
- Memory optimization
- Latency optimization
- Instruction throughput optimization
- Register spilling



Latency: Analysis

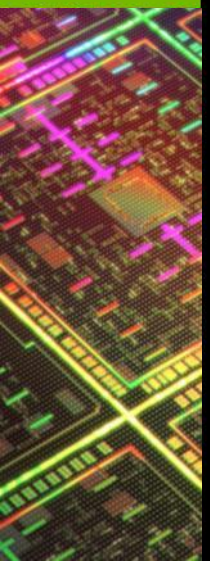
- **Suspect if:**
  - Neither memory nor instruction throughput rates are close to HW theoretical rates
  - Poor overlap between mem and math
    - Full-kernel time is significantly larger than max{mem-only, math-only}
- **Two possible causes:**
  - Insufficient concurrency per multiprocessor to hide latency
    - Two few threads launched by the kernel
    - Occupancy too low
  - 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

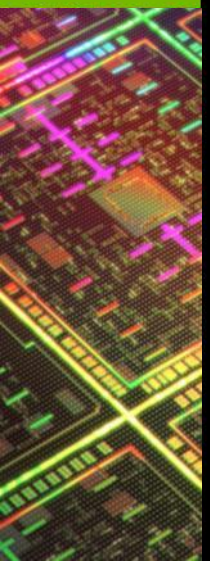
Memory-only time
Math-only time

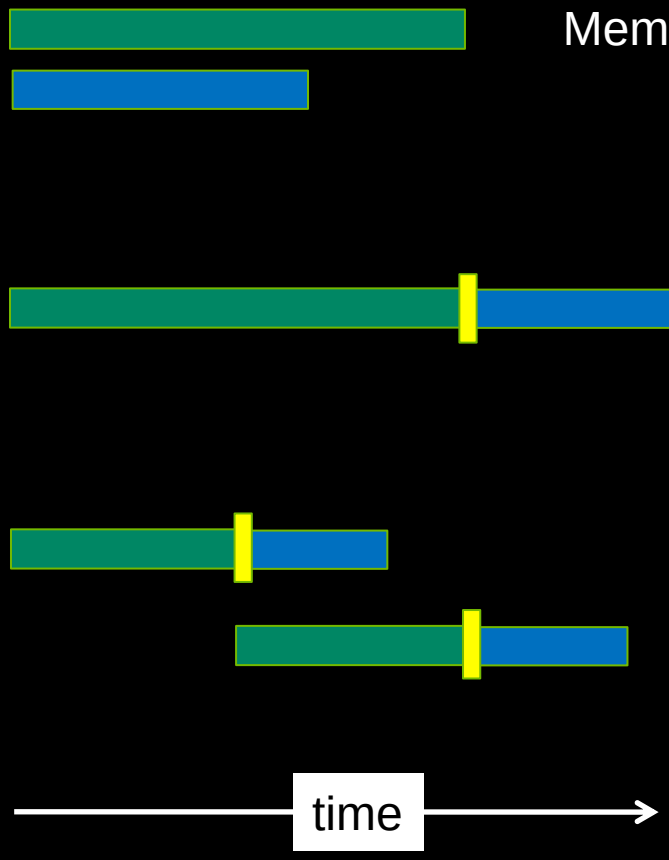
Kernel where most math cannot be executed until all data is loaded by the threadblock

Full-kernel time, one large threadblock per SM



Simplified View of Latency and Syncs

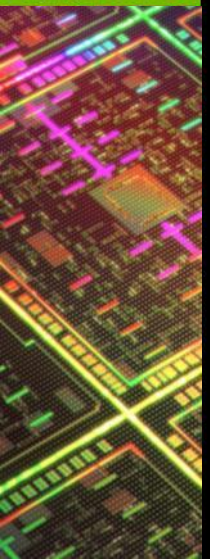




Kernel where most math cannot be executed until all data is loaded by the threadblock

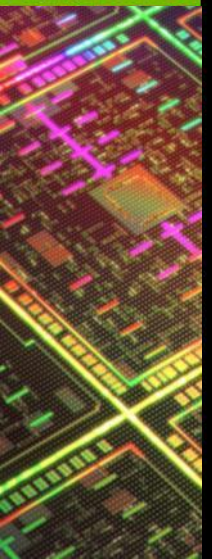
Full-kernel time, one large threadblock per SM

Full-kernel time, two threadblocks per SM
(each half the size of one large one)



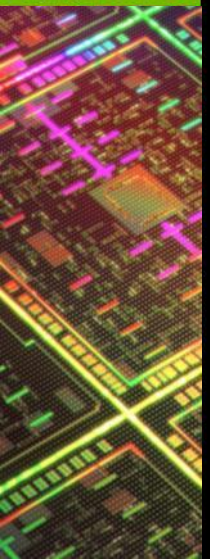
Understanding Latency Bound

- Little's Law
  - Concurrency = Throughput * Latency
    - C2050: 150 GB/s * (600 / 1.15 GHz) / 128 B / 14 SM ~ 44 mem inst per SM on the fly
  - If mem inst on the fly is << 44 per SM, we are not feeding the memory subsystem fast enough, i.e. it will be waiting, i.e. latency-bound
- How to satisfy the requirement
  - Increase occupancy (TLP)
  - Increase mem inst / warp (ILP)



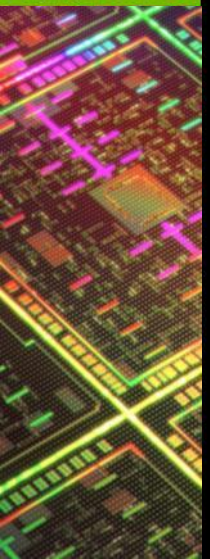
Latency Optimizations

- Adjust resource usage to increase occupancy
- Process several elements per thread
  - Multiple loads get pipelined
  - Indexing calculation may be reused
- Barriers
  - Can assess impact on perf by commenting out __syncthreads()
  - Try running several smaller threadblocks
- Issue global loads as early as possible
  - Group together
  - Prefetch



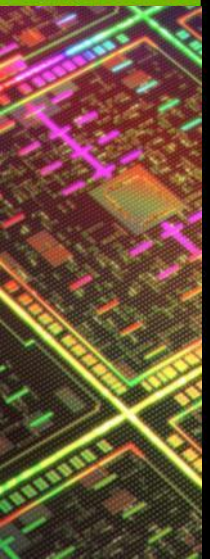
Overview

- Identifying performance limiters
- Memory optimization
- Latency optimization
- Instruction optimization
- Register spilling



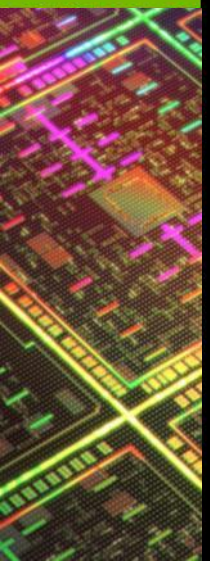
Possible Limiting Factors

- **Raw instruction throughput**
  - Know the kernel instruction mix
  - fp32, fp64, int, mem, transcendentals, etc. have different throughputs
    - Refer to the CUDA Programming Guide / Best Practices Guide
  - Can examine assembly, if needed:
    - Can look at post-optimization machine assembly for Fermi (cuobjdump)
- **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
    - Serialization increases total instruction count
  - Some causes:
    - Shared memory bank conflicts
    - Branch divergence
    - Constant memory bank conflicts



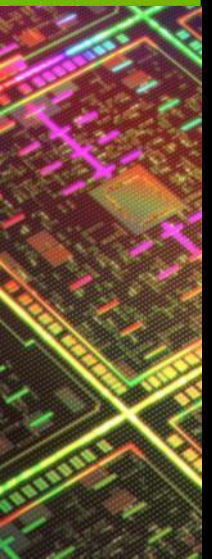
Instruction Throughput: Analysis

- **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 reports achieved throughput:
    - Fermi: as IPC (instructions per clock), max is 2



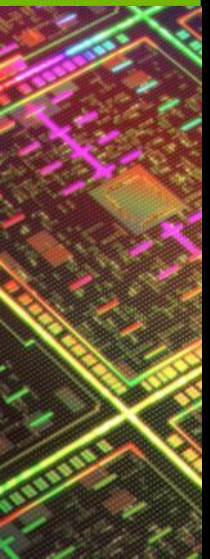
Serialization: Profiler Analysis

- **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:
    - l1_shared_bank_conflict is significant compared to (shared_load+shared_store)
    - l1_shared_bank_conflict is significant compared to instructions_issued



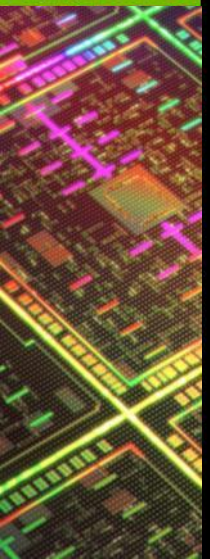
Serialization: Analysis with Modified Code

- **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



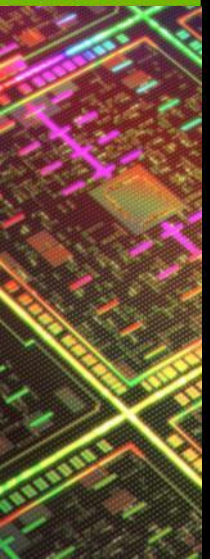
Case Study: SMEM Bank Conflicts

- **A climate simulation code kernel, fp64**
- **Profiler values:**
  - Instr/byte: 4 suggests that
    - instructions are a bigger limiter
  - Instructions:
    - instruction_executed / instruction_issued: 2,406,426 / 2,756,140
    - Difference: 349,714 (**12.7%** of instructions issued were “replays”)
  - SMEM:
    - l1_shared_bank_conflict : 674,856 (really 337,428 because of double-counting for fp64)
      - This means a total of 854,385 SMEM access instructions, **39%** replays
- **Solution:**
  - Pad shared memory array: performance increased by 15%
    - l1_shared_bank_conflit /instruction_issued ~ 12%
    - replayed instructions reduced down to 1%



Overview

- Identifying performance limiters
- Memory optimization
- Latency optimization
- Instruction optimization
- Register spilling

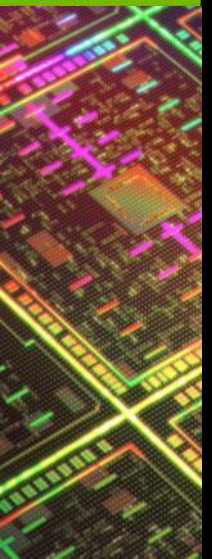


Register Spilling

- Fermi limit is 63 registers per thread
- **Compiler "spills" registers to local memory when register limit is exceeded**
  - Spills also possible when register limit is programmer-specified
    - -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

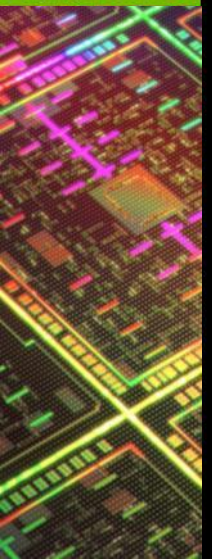
Register Spilling: Analysis

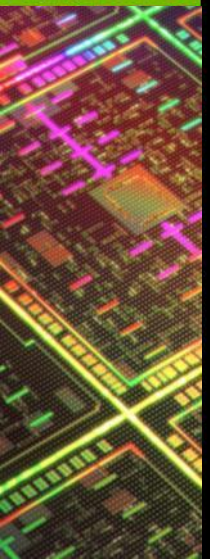
- **Profiler counters:** l1_local_load_hit, l1_local_load_miss
- **Impact on instruction count:**
  - Compare to total instructions issued
- **Impact on memory throughput:**
  - Misses add 128 bytes per warp
  - Compare 2*l1_local_load_miss count to gmem access count (stores + loads)
    - Multiply lmem 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
- Use shared memory as private memory space
- Increase parallelism: 1-thread-per-task to 1-block-per-task or 1-warp-per-task
  - Less work for each thread generally will lead to less registers used



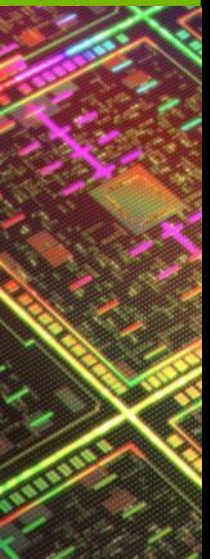


Register Spilling: Case Study

- **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

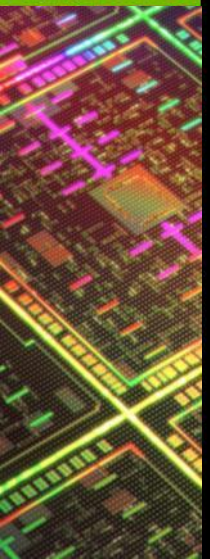
Case Study: Register Spilling 1

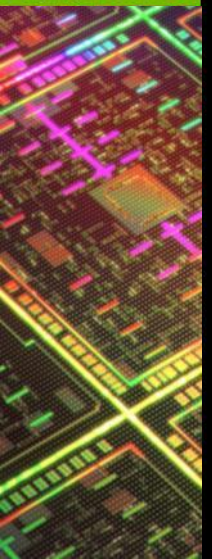
- **10th order in space kernel (31-point stencil)**
  - 32 registers per thread : 68 bytes of lmem per thread : upto 1024 threads per SM
- **Profiled counters:**
  - l1_local_load_miss = 36 inst_issued = 8,308,582
  - l1_local_load_hit = 70,956 gld_request = 595,200
  - local_store = 64,800 gst_request = 128,000
- **Conclusion: spilling is not a problem in this case**
  - The ratio of gmem to lmem bus traffic is approx 10000 : 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
- **Comparison:**
  - 42 registers per thread : no spilling : upto 768 threads per SM
    - Single 512-thread block per SM : 24% perf decrease
    - Three 256-thread blocks per SM : 7% perf decrease
  - You would want spill to keep a high occupancy



Case Study: Register Spilling 2

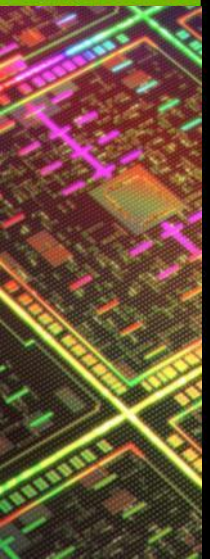
- **12th order in space kernel (37-point stencil)**
  - 32 registers per thread : 80 bytes of lmem per thread : upto 1024 threads per SM
- **Profiled counters:**
  - l1_local_load_miss = 376,889 inst_issued = 10,154,216
  - l1_local_load_hit = 36,931 gld_request = 550,656
  - local_store = 71,176 gst_request = 115,200
- **Conclusion: spilling is a problem in this case**
  - The ratio of gmem to lmem bus traffic is approx 6 : 7 (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
  - You would want to decrease occupancy to avoid spill





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*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 for bandwidth-boun
  - Instruction-bound code: lmem accesses are a significant percentage of 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:**
  - Prior CUDA tutorials video/slides at GTC
    - http://www.gputechconf.com/
  - CUDA Programming Guide, CUDA Best Practices Guide
    - http://developer.nvidia.com/cuda-downloads
  - CUDA webinars covering many introductory to advanced topics
    - http://developer.nvidia.com/gpu-computing-webinars