Performance Optimization:
Programming Guidelines and GPU Architecture Reasons Behind Them

Paulius Micikevicius
Developer Technology, NVIDIA
Goals of this Talk

• Two-fold:
  – Describe how hardware operates
  – Show how hw operation translates to optimization advice

• Previous years’ GTC Optimization talks had a different focus:
  – Show how to diagnose performance issues
  – Give optimization advice

• For a full complement of information, check out:
  – GTC 2010, GTC 2012 optimization talks
  – GTC 2013 profiling tool sessions:
    • S3046, S3011
Outline

- Thread (warp) execution
- Kernel execution
- Memory access
- Required parallelism
Requirements to Achieve Good GPU Performance

• In order of importance:
  – Expose Sufficient Parallelism
  – Efficient Memory Access
  – Efficient Instruction Execution
Thread/Warp Execution
SIMT Execution

• Single Instruction Multiple Threads
  – An instruction is issued for an entire warp
    • Warp = 32 consecutive threads
  – Each thread carries out the operation on its own arguments
Warps and Threadblocks

• Threadblocks can be 1D, 2D, 3D
  – Dimensionality of thread IDs is purely a programmer convenience
  – HW “looks” at threads in 1D
• Consecutive 32 threads are grouped into a warp
  – 1D threadblock:
    • Warp 0: threads 0...31
    • Warp 1: threads 32...63
  – 2D/3D threadblocks
    • First, convert thread IDs from 2D/3D to 1D:
      – X is the fastest varying dimension, z is the slowest varying dimension
    • Then, same as for 1D blocks
• HW uses a discrete number of warps per threadblock
  – If block size isn’t a multiple of warp size, some threads in the last warp are inactive
  – A warp is never split between different threadblocks
• **Threadblocks can be 1D, 2D, 3D**
  – Dimensionality of thread IDs is purely a programmer convenience
  – HW “looks” at threads in 1D
• **Consecutive 32 threads are grouped into a warp**
  – 1D threadblock:
    • Warp 0: threads 0...31
    • Warp 1: threads 32...63
  – 2D/3D threadblocks
    • First, convert thread IDs from 2D/3D to 1D:
      – X is the fastest varying dimension
    • Then, same as for 1D blocks
• **HW uses a discrete number of warps per threadblock**
  – If block size isn’t a multiple of warp size, some threads in the last warp are inactive
  – A warp is never split between different threadblocks

Say, 40x2 threadblock (80 “app” threads)
40 threads in x
2 rows of threads in y
Warp block and threadblocks

- **Threadblocks can be 1D, 2D, 3D**
  - Dimensionality of thread IDs is purely a programmer convenience
  - HW "looks" at threads in 1D
- **Consecutive 32 threads are grouped into a warp**
  - 1D threadblock:
    - Warp 0: threads 0...31
    - Warp 1: threads 32...63
  - 2D/3D threadblocks
    - First, convert thread IDs from 2D/3D to 1D:
      - X is the fastest varying dimension, z is the slowest varying dimension
    - Then, same as for 1D blocks
- **HW uses a discrete number of warps**
  - If block size isn’t a multiple of warp size, some threads in the last warp are inactive
  - A warp is never split between different threadblocks

**Say, 40x2 threadblock** (80 “app” threads)
- 40 threads in x
- 2 rows of threads in y

**3 warps** (92 “hw” threads)
- 1st (blue), 2nd (orange), 3rd (green)

Note that half of the “green” warp isn’t used by the app
Control Flow

• **Different warps can execute entirely different code**
  – No performance impact due to different control flow
  – Each warp maintains its own program counter
• **If only a portion of a warp has to execute an operation**
  – Threads that don’t participate are “masked out”
    • Don’t fetch operands, don’t write output
      – Guarantees correctness
    • They still spend time in the instructions (don’t execute something else)
• **Conditional execution within a warp**
  – If at least one thread needs to take a code path, entire warp takes that path
if ( ... )
{
  // then-clause
}
else
{
  // else-clause
}
Different Code Paths in Different Warps

Instructions, time

Warp ("vector" of threads)

Warp ("vector" of threads)
Different Code Paths Within a Warp
Instruction Issue

- Instructions are issued in-order
  - Compiler arranges the instruction sequence
  - If an instruction is not eligible, it stalls the warp

- An instruction is eligible for issue if both are true:
  - A pipeline is available for execution
    - Some pipelines need multiple cycles to issue a warp
  - All the arguments are ready
    - Argument isn’t ready if a previous instruction hasn’t yet produced it
Latency Hiding

- **Instruction latencies:**
  - Roughly 10-20 cycles (replays increase these)
  - DRAM accesses have higher latencies (400-800 cycles)

- **Instruction Level Parallelism (ILP)**
  - Independent instructions between two dependent ones
  - ILP depends on the code, done by the compiler

- **Switching to a different warp**
  - If a warp stalls for $N$ cycles, having $N$ other warps with eligible instructions keeps the SM going
  - Switching between concurrent warps has no overhead
    - State (registers, shared memory) is partitioned, not stored/restored
Latency Hiding

- **Instruction latencies:**
  - Roughly 10-20 cycles (replays increase these)
  - DRAM accesses have higher latencies (400-800 cycles)

- **Instruction Level Parallelism (ILP):**
  - Independent instructions between two dependent ones
  - ILP depends on the code, done by the compiler

- **Switching to a different warp**
  - If a warp stalls for \( N \) cycles, having \( N \) other warps with eligible instructions keeps the SM going
  - Switching between concurrent warps has no overhead
    - State (registers, shared memory) is partitioned, not stored/restored

```
FFMA R0, R43, R0, R4;
FFMA R1, R43, R4, R5;
FMUL R7, R9, R0;
FMUL R8, R9, R1;
ST.E [R2], R7;
```
Latency Hiding

- **Instruction latencies:**
  - Roughly 10-20 cycles (replays increase these)
  - DRAM accesses have higher latencies (400-800 cycles)

- **Instruction Level Parallelism (ILP)**
  - Independent instructions between two dependent ones
  - ILP depends on the code, done by the compiler

- **Switching to a different warp**
  - If a warp stalls for $N$ cycles, having $N$ other warps with eligible instructions keeps the SM going
  - Switching between concurrent warps has no overhead
    - State (registers, shared memory) is partitioned, not stored/restored
Kepler Instruction Issue

- GPU consists of some number of SMs
  - Kepler chips: 1-14 SMs
- Each SM has 4 instruction scheduler units
  - Warps are partitioned among these units
  - Each unit keeps track of its warps and their eligibility to issue
- Each scheduler can dual-issue instructions from a warp
  - Resources and dependencies permitting
  - Thus, a Kepler SM could issue 8 warp-instructions in one cycle
    - 7 is the sustainable peak
    - 4-5 is pretty good for instruction-limited codes
    - Memory- or latency-bound codes by definition will achieve much lower IPC

© 2013, NVIDIA
Kepler Instruction Issue

- **Kepler SM needs at least 4 warps**
  - To occupy the 4 schedulers
  - In practice you need many more to hide instruction latency
    - An SM can have up to 64 warps active
    - Warps can come from different threadblocks and different concurrent kernels
      - HW doesn’t really care: it keeps track of the instruction stream for each warp

- **For instruction limited codes:**
  - No ILP: 40 or more concurrent warps per SM
    - 4 schedulers × 10+ cycles of latency
  - The more ILP, the fewer warps you need

- **Rough rule of thumb:**
  - Start with ~32 warps for SM, adjust from there
    - Most codes have some ILP
CUDA Cores and the Number of Threads

• Note that I haven’t mentioned CUDA cores till now
  – GPU core = fp32 pipeline lane (192 per Kepler SM)
  – GPU core definition predates compute-capable GPUs

• Number of threads needed for good performance:
  – Not really tied to the number of CUDA cores
  – Need enough threads (warps) to hide latencies
GK110 SM Diagram

- **192 fp32 lanes (cores)**
  - fp32 math
  - Simple int32 math (add, min, etc.)
- **64 fp64 lanes**
- **32 SFU lanes**
  - Int32 multiplies, etc.
  - Transcendentals
- **32 LD/ST lanes**
  - GMEM, SMEM, LMEM accesses
- **16 TEX lanes**
  - Texture access
  - Read-only GMEM access

Kepler SM Instruction Throughputs

- **Fp32 instructions**
  - Equivalent of “6 warps worth” of instructions per cycle (192 pipes)
  - Requires some dual-issue to use all pipes:
    - SM can issue instructions from 4 warps per cycle (4 schedulers/SM)
    - Without any ILP one couldn’t use more than $4 \times 32 = 128$ fp32 pipes

- **Fp64 pipelines**
  - Number depends on a chip
  - Require 2 cycles to issue a warp
  - K20 (gk110) chips: 2 warps worth of instructions per cycle (64 pipes)

- **Memory access**
  - Shared/global/local memory instructions
  - 1 warp per cycle

- **See the CUDA Programming Guide for more details** (docs.nvidia.com)
  - Table “Throughput of Native Arithmetic Instructions”
Examining Assembly

- **Two levels of assembly**
  - PTX: virtual assembly
    - Forward-compatible
      - Driver will JIT to machine language
    - Can be inlined in your CUDA C code
    - Not the final, optimized machine code
  - Machine language:
    - Architecture specific (not forward/backward compatible)
    - The sequence of instructions that HW executes

- **Sometimes it’s interesting to examine the assembly**
  - `cuobjdump` utility
    - comes with every CUDA toolkit
    - PTX: `cuobjdump -ptx <executable or object file>`
    - Machine assembly: `cuobjdump -sass <executable or object file>`
  - Docs on inlining PTX and instruction set
    - Look in the docs directory inside the toolkit install for PDFs
Takeaways

• **Have enough warps to hide latency**
  – Rough rule of thumb: initially aim for 32 warps/SM
    • Use profiling tools to tune performance afterwards
  – Don’t think in terms of CUDA cores

• **If your code is instruction throughput limited:**
  – When possible use operations that go to wider pipes
    • Use fp32 math instead of fp64, when feasible
    • Use intrinsics (__sinf(), __sqrtf(), ...)
      – Single HW instruction, rather than SW sequences of instructions
      – Tradeoff: slightly fewer bits of precision
      – For more details: CUDA Programming Guide
  – Minimize different control flow within warps (warp-divergence)
    • Only an issue if large portions of time are spent in divergent code
Kernel Execution
Kernel Execution

• A grid of threadblocks is launched
  – Kernel<<<1024,...>>>(...): grid of 1024 threadblocks

• Threadblocks are assigned to SMs
  – Assignment happens only if an SM has sufficient resources for the entire threadblock
    • Resources: registers, SMEM, warp slots
    • Threadblocks that haven’t been assigned wait for resources to free up
  – The order in which threadblocks are assigned is not defined
    • Can and does vary between architectures

• Warps of a threadblock get partitioned among the 4 schedulers
  – Each scheduling unit keeps track of all its warps
  – In each cycle chooses an eligible warp for issue
    • Aims for fairness and performance
Concurrent Kernel Execution

- General stream rules apply - calls may overlap if both are true:
  - Calls are issued to different, non-null streams
  - There is no synchronization between the two calls

- Kernel launch processing
  - First, assign all threadblocks of the “current” grid to SMs
  - If SM resources are still available, start assigning blocks from the “next” grid
  - “Next”:
    - Compute capability 3.5: any kernel to a different stream that’s not separated with a sync
    - Compute capability <3.5: the next kernel launch in code sequence
  - An SM can concurrently execute threadblocks from different kernels
  - Limits on concurrently execute threadblocks from different kernels per GPU:
    - CC 3.5: 32
    - CC 2.x: 16
Kernel Execution in High Priority Streams

• Priorities require:
  – CC 3.5 or higher
  – CUDA 5.5 or higher

• High-priority kernel threadblocks will be assigned to SMs as soon as possible
  – Do not preempt already executing threadblocks
    • Wait for these to finish and free up SM resources
  – “Pass” the low-priority threadblocks waiting to be assigned

• Concurrent kernel requirements apply
  – Calls in the same stream still execute in sequence
CDP Kernel Execution

- Same as “regular” launches, except cases where a GPU thread waits for its launch to complete
  - GPU thread: kernel launch, device or stream sync call later
  - To prevent deadlock, the parent threadblock:
    - Is swapped out upon reaching the sync call
      - guarantees that child grid will execute
    - Is restored once all child threadblocks complete
  - Context store/restore adds some overhead
    - Register and SMEM contents must be written/read to GMEM
  - In general:
    - We guarantee forward progress for child grids
    - Implementation for the guarantee may change in the future
- A threadblock completes once all its child grids finish

Takeaways

• **Ensure that grids have sufficient threadblocks to occupy the entire chip**
  – Grid threadblocks are assigned to SMs
  – Each SM partitions threadblock warps among its 4 schedulers
  – SM needs sufficient warps to hide latency

• **Concurrent kernels:**
  – Help if individual grids are too small to fully utilize GPU

• **Executing in high-priority streams:**
  – Helps if certain kernels need preferred execution

• **CUDA Dynamic Parallelism:**
  – Be aware that a sync call after launching a kernel may cause a threadblock state store/restore
Memory Access
Memory Optimization

• Many algorithms are memory-limited
  – Most are at least somewhat sensitive to memory bandwidth
  – Reason: not that much arithmetic per byte accessed
    • Not uncommon for code to have ~1 operation per byte
    • Instr:mem bandwidth ratio for most modern processors is 4-10
      – CPUs and GPUs
  – Exceptions exist: DGEMM, Mandelbrot, some Monte Carlo, etc.

• Optimization goal: maximize bandwidth utilization
  – Maximize the use of bytes that travel on the bus
  – Have sufficient concurrent memory accesses
Maximize Byte Use

- Two things to keep in mind:
  - Memory accesses are per warp
  - Memory is accessed in discrete chunks
    - lines/segments
    - want to make sure that bytes that travel from DRAM to SMs get used
      - For that we should understand how memory system works

- Note: not that different from CPUs
  - x86 needs SSE/AVX memory instructions to maximize performance

---

![Diagram showing SMs and DRAM connections](image-url)
GPU Memory System

- All data lives in DRAM
  - Global memory
  - Local memory
  - Textures
  - Constants

DRAM
GPU Memory System

• All DRAM accesses go through L2
• Including copies:
  – P2P
  – CPU-GPU
Once in an SM, data goes into one of 3 caches/buffers

- Programmer’s choice
  - L1 is the “default”
  - Read-only, Const require explicit code
Access Path

- **L1 path**
  - Global memory
    - Memory allocated with cudaMalloc()
    - Mapped CPU memory, peer GPU memory
    - Globally-scoped arrays qualified with __global__
  - Local memory
    - allocation/access managed by compiler so we’ll ignore

- **Read-only/TEX path**
  - Data in texture objects, CUDA arrays
  - CC 3.5 and higher:
    - Global memory accessed via intrinsics (or specially qualified kernel arguments)

- **Constant path**
  - Globally-scoped arrays qualified with __constant__
Access Via L1

• Natively supported word sizes per thread:
  – 1B, 2B, 4B, 8B, 16B
    • Addresses must be aligned on word-size boundary
  – Accessing types of other sizes will require multiple instructions

• Accesses are processed per warp
  – Threads in a warp provide 32 addresses
    • Fewer if some threads are inactive
  – HW converts addresses into memory transactions
    • Address pattern may require multiple transactions for an instruction
    • If $N$ transactions are needed, there will be ($N-1$) replays of the instruction
GMEM Writes

- Not cached in the SM
  - Invalidate the line in L1, go to L2
- Access is at **32 B** segment granularity
- Transaction to memory: **1, 2, or 4** segments
  - Only the required segments will be sent
- If multiple threads in a warp write to the same address
  - One of the threads will “win”
  - Which one is not defined
Some Store Pattern Examples

addresses from a warp

one 4-segment transaction

Memory addresses

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448
Some Store Pattern Examples

addresses from a warp

three 1-segment transactions

Memory addresses

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448
Some Store Pattern Examples

addresses from a warp

one 2-segment transaction

Memory addresses
Some Store Pattern Examples

addresses from a warp

2 1-segment transactions

Memory addresses

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448
GMEM Reads

- Attempt to hit in L1 depends on programmer choice and compute capability
- HW ability to hit in L1:
  - CC 1.x: no L1
  - CC 2.x: can hit in L1
  - CC 3.0, 3.5: cannot hit in L1
    - L1 is used to cache LMEM (register spills, etc.), buffer reads
- Read instruction types
  - Caching:
    - Compiler option: -Xptxas -dlcm=ca
    - On L1 miss go to L2, on L2 miss go to DRAM
    - Transaction: 128 B line
  - Non-caching:
    - Compiler option: -Xptxas -dlcm=cg
    - Go directly to L2 (invalidate line in L1), on L2 miss go to DRAM
    - Transaction: 1, 2, 4 segments, segment = 32 B (same as for writes)
Caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss

addresses from a warp
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words

- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss

Memory addresses:

```
    0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
```

addresses from a warp
Caching Load

**Scenario:**
- Warp requests 32 aligned, permuted 4-byte words

**Addresses fall within 1 cache-line**
- No replays
- Bus utilization: 100%
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss

---

addresses from a warp

Memory addresses
Caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within 2 cache-lines**
  - 1 replay (2 transactions)
  - Bus utilization: 50%
    - Warp needs 128 bytes
    - 256 bytes move across the bus on misses
Non-caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within at most 5 segments**
  - 1 replay (2 transactions)
  - Bus utilization: at least 80%
    - Warp needs 128 bytes
    - At most 160 bytes move across the bus
    - Some misaligned patterns will fall within 4 segments, so 100% utilization

addresses from a warp

Memory addresses
Caching Load

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single cache-line**
  - No replays
  - Bus utilization: 3.125%
    - Warp needs 4 bytes
    - 128 bytes move across the bus on a miss
Non-caching Load

• **Scenario:**
  – All threads in a warp request the same 4-byte word

• **Addresses fall within a single segment**
  – No replays
  – Bus utilization: 12.5%
    • Warp needs 4 bytes
    • 32 bytes move across the bus on a miss

addresses from a warp

Memory addresses

0  32  64  96  128  160  192  224  256  288  320  352  384  416  448

© 2012, NVIDIA
Caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within N cache-lines**
  - (N-1) replays (N transactions)
  - Bus utilization: \(\frac{32 \times 4B}{N \times 128B}\)
    - Warp needs 128 bytes
    - \(N \times 128\) bytes move across the bus on a miss
Non-caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words

- **Addresses fall within N segments**
  - $(N-1)$ replays ($N$ transactions)
    - Could be lower some segments can be arranged into a single transaction
  - Bus utilization: $128 / (N*32)$ (4x higher than caching loads)
    - Warp needs 128 bytes
    - $N*32$ bytes move across the bus on a miss
Caching vs Non-caching Loads

• Compute capabilities that can hit in L1 (CC 2.x)
  – Caching loads are better if you count on hits
  – Non-caching loads are better if:
    • Warp address pattern is scattered
    • When kernel uses lots of LMEM (register spilling)

• Compute capabilities that cannot hit in L1 (CC 1.x, 3.0, 3.5)
  – Does not matter, all loads behave like non-caching

• In general, don’t rely on GPU caches like you would on CPUs:
  – 100s of threads sharing the same L1
  – 1000s of threads sharing the same L2
L1 Sizing

• Fermi and Kepler GPUs split 64 KB RAM between L1 and SMEM
  – Fermi GPUs (CC 2.x): 16:48, 48:16
  – Kepler GPUs (CC 3.x): 16:48, 48:16, 32:32

• Programmer can choose the split:
  – Default: 16 KB L1, 48 KB SMEM
  – Run-time API functions:
    • cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
  – Kernels that require different L1:SMEM sizing cannot run concurrently

• Making the choice:
  – Large L1 can help when using lots of LMEM (spilling registers)
  – Large SMEM can help if occupancy is limited by shared memory
Read-Only Cache

• An alternative to L1 when accessing DRAM
  – Also known as texture cache: all texture accesses use this cache
  – CC 3.5 and higher also enable global memory accesses
    • Should not be used if a kernel reads and writes to the same addresses

• Comparing to L1:
  – Generally better for scattered reads than L1
    • Caching is at 32 B granularity (L1, when caching operates at 128 B granularity)
    • Does not require replay for multiple transactions (L1 does)
  – Higher latency than L1 reads, also tends to increase register use

• Aggregate 48 KB per SM: 4 12-KB caches
  – One 12-KB cache per scheduler
    • Warps assigned to a scheduler refer to only that cache
  – Caches are not coherent – data replication is possible
Read-Only Cache Operation

- Always attempts to hit
- Transaction size: 32 B queries
- Warp addresses are converted to queries 4 threads at a time
  - Thus a minimum of 8 queries per warp
  - If data within a 32-B segment is needed by multiple threads in a warp, segment misses at most once
- Additional functionality for texture objects
  - Interpolation, clamping, type conversion
Read-Only Cache Operation

addresses from a warp

1st Query
Read-Only Cache Operation

addresses from a warp

1\textsuperscript{st} Query

addresses from a warp

2\textsuperscript{nd} Query
Read-Only Cache Operation

addresses from a warp

1st Query
Read-Only Cache Operation

addresses from a warp

1st Query

addresses from a warp

2nd and 3rd Queries
Read-Only Cache Operation

addresses from a warp

1st Query

2nd and 3rd Queries

addresses from a warp

Note this segment was already requested in the 1st query:
cache hit, no redundant requests to L2
Accessing GMEM via Read-Only Cache

• Compiler must know that addresses read are not also written by the same kernel

• Two ways to achieve this
  – Intrinsic: __ldg()
  – Qualify the pointers to the kernel
    • All pointers: __restrict__
    • Pointers you’d like to dereference via read-only cache: const __restrict__
    • May not be sufficient if kernel passes these pointers to functions
Accessing GMEM via Read-Only Cache

• Compiler must know that addresses read are not also written by the same kernel

• Two ways to achieve this
  – Intrinsic: `__ldg()`
  – Qualify the pointers to the kernel:
    • All pointers: `__restrict__`
    • Pointers you’d like to dereference via read-only cache: `const __restrict__`

```c
__global__ void kernel( int *output, int *input )
{
    ... 
    output[idx] = ... + __ldg( &input[idx] );
}
```
Accessing GMEM via Read-Only Cache

• Compiler must know that addresses read are not also written by the same kernel
• Two ways to achieve this
  – Intrinsic: `__ldg()`
  – Qualify the pointers to the kernel
    • All pointers: `__restrict__`
    • Pointers you’d like to dereference via read-only cache: `const __restrict__`
      • May not be sufficient if kernel passes these pointers to functions

```c
__global__ void kernel(int* __restrict__ output, const int* __restrict__ input)
{
    ...
    output[idx] = ... + input[idx];
}
```
Additional Texture Functionality

- All of these are “free”
  - Dedicated hardware
  - Must use CUDA texture objects
    - See CUDA Programming Guide for more details
    - Texture objects can interoperate graphics (OpenGL, DirectX)
- Out-of-bounds index handling: clamp or wrap-around
- Optional interpolation
  - Think: using fp indices for arrays
  - Linear, bilinear, trilinear
    - Interpolation weights are 9-bit
- Optional format conversion
  - {char, short, int, fp16} -> float
Examples of Texture Object Indexing

Index Clamp:

0 1 2 3 4
0 1 2 3 4

0 (5.5, 1.5)
1 (1.0, 1.0)
2
3

Index Wrap:

0 1 2 3 4
0 1 2 3 4

0 (5.5, 1.5)
1
2
3

Integer indices fall between elements
Optional interpolation:
Weights are determined by coordinate distance
Constant Cache

• The 3rd alternative DRAM access path
• Also the most restrictive:
  – Total data for this path is limited to 64 KB
    • Must be copied into an array qualified with `__constant__`
  – Cache throughput: 4 B per clock per SM
    • So, unless the entire warp reads the same address, replays are needed
• Useful when:
  – There is some small subset of data used by all threads
    • But it gets evicted from L1/Read-Only paths by reads of other data
  – Data addressing is not dependent on thread ID
    • Replays are expensive
• Example use: FD coefficients
Constant Cache

• The 3rd alternative DRAM access path
• Also the most restrictive:
  – Total data for this path is limited to 64 KB
    • Must be copied into an array qualified with `__constant__`
  – Cache throughput: 4 B per clock per SM
    • So, unless the entire warp reads the same address, replays are needed
• Useful when:
  – There is some small subset of data used by all threads
    • But it gets evicted from L1/Read
  – Data addressing is not dependent on thread ID
    • Replays are expensive
• Example use: FD coefficients

// global scope:
  `__constant__` float coefficients[16];
...

// in GPU kernel code:
  deriv = coefficients[0] * data[idx] + ...
  ...

// in CPU-code:
  `cudaMemcpyToSymbol` (coefficients, ... )
Address Patterns

• **Coalesced address pattern**
  – Warp utilizes all the bytes that move across the bus

• **Suboptimal address patterns**
  – Throughput from HW point of view is significantly higher than from app point of view
  – Four general categories:
    1) Offset (not line-aligned) warp addresses
    2) Large strides between threads within a warp
    3) Each thread accesses a contiguous region (larger than a word)
    4) Irregular (scattered) addresses

See GTC 2012 “GPU Performance Analysis and Optimization” (session S0514) for details on diagnosing and remedies. Slides and video:

Case Study 1: Contiguous Region per Thread

• Say we are reading a 12-byte structure per thread
  – Non-native word size

```c
struct Position
{
    float x, y, z;
};
...
__global__ void kernel( Position *data, ... )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    Position temp = data[idx];
    ...
}
```
Case Study 1: Non-Native Word Size

• Compiler converts $temp = data[\text{idx}]$ into 3 loads:
  – Each loads 4 bytes
  – Can’t do an 8 and a 4 byte load: 12 bytes per element means that every other element wouldn’t align the 8-byte load on 8-byte boundary

• Addresses per warp for each of the loads:
  – Successive threads read 4 bytes at 12-byte stride
Case Study 1: 1\textsuperscript{st} Load Instruction

addresses from a warp

32 B memory transaction...
Case Study 1: 2\textsuperscript{nd} Load Instruction

addresses from a warp
Case Study 1: 3\textsuperscript{rd} Load Instruction

addresses from a warp
Case Study 1: Performance and Solutions

- Because of the address pattern, SMs end up requesting 3x more bytes than application requests
  - We waste a lot of bandwidth

- Potential solutions:
  - Change data layout from array of structures to structure of arrays
    - In this case: 3 separate arrays of floats
    - The most reliable approach (also ideal for both CPUs and GPUs)
  - Use loads via read-only cache (LDG)
    - As long as lines survive in the cache, performance will be nearly optimal
    - Only available in CC 3.5 and later
  - Stage loads via shared memory (SMEM)
Case Study 1: Speedups for Various Solutions

- **Kernel that just reads that data:**
  - AoS (float3): 1.00
  - LDG: 1.43
  - SMEM: 1.40
  - SoA: 1.51

- **Kernel that just stores the data:**
  - AoS (float3): 1.00
  - LDG: N/A (stores don’t get cached in SM)
  - SMEM: 1.88
  - SoA: 1.88

- **Speedups aren’t 3x because we are hitting in L2**
  - DRAM didn’t see a 3x increase in traffic
Maximize Memory Bandwidth Utilization

• Maximize the use of bytes that travel on the bus
  – Address pattern

• Have sufficient concurrent memory accesses
  – Latency hiding
Optimizing Access Concurrency

• Have enough concurrent accesses to saturate the bus
  – Little’s law: need latency $\times$ bandwidth bytes in flight
Little’s Law for Escalators

• Say the parameters of our escalator are:
  – 1 person fits on each step
  – A step arrives every 2 seconds
    • Bandwidth: 0.5 person/s
  – 20 steps tall
    • Latency: 40 seconds
Little’s Law for Escalators

• Say the parameters of our escalator are:
  – 1 person fits on each step
  – A step arrives every 2 seconds
    • **Bandwidth:** 0.5 person/s
  – 20 steps tall
    • **Latency:** 40 seconds
• 1 person in flight: 0.025 persons/s achieved
Little’s Law for Escalators

- Say the parameters of our escalator are:
  - 1 person fits on each step
  - A step arrives every 2 seconds
    - Bandwidth: 0.5 person/s
  - 20 steps tall
    - Latency: 40 seconds
- 1 person in flight: 0.025 persons/s achieved
- To saturate bandwidth:
  - Need 1 person arriving every 2 s
  - Means we’ll need 20 persons in flight
- The idea: **Bandwidth \times Latency**
  - It takes latency time units for the first person to arrive
  - We need bandwidth persons get on the escalator every time unit
Having Sufficient Concurrent Accesses

• In order to saturate memory bandwidth, SM must issue enough independent memory requests
Optimizing Access Concurrency

- GK104, GK110 GPUs need ~100 lines in flight per SM
  - Each line is 128 bytes
  - Alternatively, ~400 32-byte segments in flight

- Ways to increase concurrent accesses:
  - Increase occupancy (run more warps concurrently)
    - Adjust threadblock dimensions
      - To maximize occupancy at given register and smem requirements
    - If occupancy is limited by registers per thread:
      - Reduce register count (-maxrregcount option, or __launch_bounds__)
  - Modify code to process several elements per thread
    - Doubling elements per thread doubles independent accesses per thread
Case Study 2: Increasing Concurrent Accesses

- **VTI RTM kernel (3D FDTD)**
  - Register and SMEM usage allows to run 42 warps per SM
  - Initial threadblock size choice: 32x16
    - 16 warps per threadblock → 32 concurrent warps per SM
  - Insufficient concurrent accesses limit performance:
    - Achieved mem throughput is only 37%
    - Memory-limited code (low arithmetic intensity)
    - Addresses are coalesced

- **Reduce threadblock size to 32x8**
  - 8 warps per threadblock → 40 concurrent warps per SM
  - 32 → 40 warps per SM: 1.25x more memory accesses in flight
  - 1.28x speedup
Takeaways

• Strive for address patterns that maximize the use of bytes that travel across the bus
  – Use the profiling tools to diagnose address patterns
  – Most recent tools will even point to code with poor address patterns

• Provide sufficient concurrent accesses
Shared memory
Comparing to DRAM:
- 20-30x lower latency
- ~10x higher bandwidth
- Accessed at bank-width granularity
  - Fermi: 4 bytes
  - Kepler: 8 bytes
  - GMEM granularity is either 32 or 128 bytes
Shared Memory Instruction Operation

• 32 threads in a warp provide addresses
  – HW determines into which 8-byte words addresses fall
• Reads: fetch the words, distribute the requested bytes among the threads
  – Multi-cast capable
  – Bank conflicts cause replays
• Writes:
  – Multiple threads writing the same address: one “wins”
  – Bank conflicts cause replays
Kepler Shared Memory Banking

- **32 banks, 8 bytes wide**
  - Bandwidth: 8 bytes per bank per clock per SM
  - 256 bytes per clk per SM
  - K20x: 2.6 TB/s aggregate across 14 SMs

- **Two modes:**
  - 4-byte access (default):
    - Maintains Fermi bank-conflict behavior exactly
    - Provides 8-byte bandwidth for certain access patterns
  - 8-byte access:
    - Some access patterns with Fermi-specific padding may incur bank conflicts
    - Provides 8-byte bandwidth for all patterns (assuming 8-byte words)
  - Selected with `cudaDeviceSetSharedMemConfig()` function
Kepler 8-byte Bank Mode

• Mapping addresses to banks:
  – Successive 8-byte words go to successive banks
  – Bank index:
    • (8B word index) mod 32
    • (4B word index) mod (32*2)
    • (byte address) mod (32*8)
  – Given the 8 least-significant address bits: ...BBBBBxxx
    • xxx selects the byte within an 8-byte word
    • BBBBBB selects the bank
    • Higher bits select a “row” within a bank
### Address Mapping in 8-byte Mode

<table>
<thead>
<tr>
<th>Bank-0</th>
<th>Bank-1</th>
<th>Bank-2</th>
<th>Bank-3</th>
<th>Bank-31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>2 3</td>
<td>4 5</td>
<td>6 7</td>
<td></td>
</tr>
<tr>
<td>64 65</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Byte-address:** 0 4 8 12 16 20 24 28 32 38 40
- **Data:** 0 1 2 3 4 5 6 7 8 9

*(or 4B-word index)*
Kepler 4-byte Bank Mode

• Understanding this mapping details matters **only if you’re trying to get 8-byte throughput in 4-byte mode**
  – For all else just think that you have 32 banks, 4-bytes wide

• **Mapping addresses to banks:**
  – Successive 4-byte words go to successive banks
    • We have to choose between two 4-byte “half-words” for each bank
      – “First” 32 4-byte words go to lower half-words
      – “Next” 32 4-byte words go to upper half-words
  – Given the 8 least-significant address bits: ...HBBBBBxx
    • xx selects the byte with a 4-byte word
    • BBBBB selects the bank
    • H selects the half-word within the bank
    • Higher bits select the “row” within a bank
Address Mapping in 4-byte Mode

Byte-address: 0  4  8  12  16  20  24  28  32  36  40

Data: 0  1  2  3  4  5  6  7  8  9

(or 4B-word index)

Bank-0
Bank-1
Bank-2
Bank-3

Bank-31
Shared Memory Bank Conflicts

• A bank conflict occurs when:
  – 2 or more threads in a warp access different 8-B words in the same bank
    • Think: 2 or more threads access different “rows” in the same bank
  – $N$-way bank conflict: $N$ threads in a warp conflict
    • Instruction gets replayed $(N-1)$ times: increases latency
    • Worst case: 32-way conflict $\rightarrow$ 31 replays, latency comparable to DRAM

• Note there is no bank conflict if:
  – Several threads access the same word
  – Several threads access different bytes of the same word
SMEM Access Examples

Addresses from a warp: no bank conflicts
One address access per bank
SMEM Access Examples

Addresses from a warp: no bank conflicts
One address access per bank

Bank-0  Bank-1  Bank-2  Bank-3

Bank-31
SMEM Access Examples

Addresses from a warp: no bank conflicts
Multiple addresses per bank, but within the same word
SMEM Access Examples

Addresses from a warp: 2-way bank conflict
2 accesses per bank, fall in two different words
SMEM Access Examples

Addresses from a warp: 3-way bank conflict
4 accesses per bank, fall in 3 different words
Case Study 3: Matrix Transpose

• Staged via SMEM to coalesce GMEM addresses
  – 32x32 threadblock, double-precision values
  – 32x32 array in shared memory

• Initial implementation:
  – A warp reads a row of values from GMEM, writes to a row of SMEM
  – Synchronize the threads in a block
  – A warp reads a column of from SMEM, writes to a row in GMEM
Case Study 3: Matrix Transpose

- 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)
Case Study 3: Matrix Transpose

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts
Case Study 3: Matrix Transpose

• **Remedy:**
  – Simply pad each row of SMEM array with an extra element
    • 32x33 array, as opposed to 32x32
    • Effort: 1 character, literally
  – Warp access to SMEM
    • Writes still have no bank conflicts:
      – threads access successive elements
    • Reads also have no bank conflicts:
      – Stride between threads is 17 8-byte words, thus each goes to a different bank

• **Speedup: ~2x**
  – Note that the code has 2 gmem accesses and 2 smem accesses per thread
  – Removing 32-way bank conflicts cut time in half: implies bank conflicts were taking as long as gmem accesses
Summary: Shared Memory

• Shared memory is a tremendous resource
  – Very high bandwidth (terabytes per second)
  – 20-30x lower latency than accessing GMEM
  – Data is programmer-managed, no evictions by hardware

• Performance issues to look out for:
  – Bank conflicts add latency and reduce throughput
    • Many-way bank conflicts can be very expensive
      – Replay latency adds up, can become as long as DRAM latency
      – However, few code patterns have high conflicts, padding is a very simple
        and effective solution
  – Use profiling tools to identify bank conflicts
Exposing sufficient parallelism
Kepler: Level of Parallelism Needed

- **To saturate instruction bandwidth:**
  - Fp32 math: $\sim 1.7K$ independent instructions per SM
  - Lower for other, lower-throughput instructions
  - Keep in mind that Kepler SM can track up to 2048 threads

- **To saturate memory bandwidth:**
  - 100+ independent lines per SM
Exposing Sufficient Parallelism

• What hardware ultimately needs:
  – Arithmetic pipes:
    • sufficient number of independent instructions
      – accommodates multi-issue and latency hiding
  – Memory system:
    • sufficient requests in flight to saturate bandwidth

• Two ways to increase parallelism:
  – More independent work within a thread (warp)
    • ILP for math, independent accesses for memory
  – More concurrent threads (warps)
Occupancy

• **Occupancy: number of concurrent threads per SM**
  – Expressed as either:
    • the number of threads (or warps),
    • percentage of maximum threads

• **Determined by several factors**
  – (refer to Occupancy Calculator, CUDA Programming Guide for full details)
  – Registers per thread
    • SM registers are partitioned among the threads
  – Shared memory per threadblock
    • SM shared memory is partitioned among the blocks
  – Threads per threadblock
    • Threads are allocated at threadblock granularity

---

**Kepler SM resources**

- 64K 32-bit registers
- Up to 48 KB of shared memory
- Up to 2048 concurrent threads
- Up to 16 concurrent threadblocks
Occupancy and Performance

• Note that 100% occupancy isn’t needed to reach maximum performance
  – Once the “needed” occupancy is reached, further increases won’t improve performance

• Needed occupancy depends on the code
  – More independent work per thread -> less occupancy is needed
  – Memory-bound codes tend to need more occupancy
    • Higher latency than for arithmetic, need more work to hide it
Exposing Parallelism: Grid Configuration

- **Grid**: arrangement of threads into threadblocks
- **Two goals:**
  - Expose enough parallelism to an SM
  - Balance work across the SMs
- **Several things to consider when launching kernels:**
  - Number of threads per threadblock
  - Number of threadblocks
  - Amount of work per threadblock
Threadblock Size and Occupancy

• Threadblock size is a multiple of warp size (32)
  – Even if you request fewer threads, HW rounds up

• Threadblocks can be too small
  – Kepler SM can run up to 16 threadblocks concurrently
  – SM may reach the block limit before reaching good occupancy
    • Example: 1-warp blocks -> 16 warps per Kepler SM (probably not enough)

• Threadblocks can be too big
  – Quantization effect:
    • Enough SM resources for more threads, not enough for another large block
    • A threadblock isn’t started until resources are available for all of its threads
Threadblock Sizing

- **SM resources:**
  - Registers
  - Shared memory

Case Study 2
Waves and Tails

• **Wave of threadblocks**
  – A set of threadblocks that run concurrently on GPU
  – Maximum size of the wave is determined by:
    • How many threadblocks can fit on one SM
      – Number of threads per block
      – Resource consumption: registers per thread, SMEM per block
    • Number of SMs

• **Any grid launch will be made up of:**
  – Some number of *full waves*
  – Possibly one *tail*: wave with fewer than possible blocks
    • Last wave by definition
    • Happens if the grid size is not divisible by wave size
Tail Effect

- Tail underutilizes GPU
  - Impacts performance if tail is a significant portion of time

- Example:
  - GPU with 8 SMs
  - Code that can run 1 threadblock per SM at a time
    - Wave size = 8 blocks
  - Grid launch: 12 threadblocks

- 2 waves:
  - 1 full
  - Tail with 4 threadblocks
    - Tail utilizes 50% of GPU, compared to full-wave
    - Overall GPU utilization: 75% of possible
Tail Effect

- **A concern only when:**
  - Launching few threadblocks (no more than a few waves)
  - Tail effect is negligible when launching 10s of waves
    - If that’s your case, you can ignore the following info
- **Tail effect can occur even with perfectly-sized grids**
  - Threadblocks don’t stay in lock-step
- **To combat tail effect:**
  - Spread the work of one thread among several threads
    - Increases the number of blocks -> increases the number of waves
  - Spread the threads of one block among several
    - Improves load balancing during the tail
  - Launch independent kernels into different streams
    - Hardware will execute threadblocks from different kernels to fill the GPU
Tail Effect: Large vs Small Threadblocks

- **2 waves of threadblocks**
  - Tail is running at 25% of possible
  - Tail is 50% of time
    - Could be improved if the tail work could be better balanced across SMs

- **4 waves of threadblocks**
  - Tail is running at 75% of possible
  - Tail is 25% of time
    - Tail work is spread across more threadblocks, better balanced across SMs
    - Estimated speedup: 1.5x (time reduced by 33%)
Tail Effect: Few vs Many Waves of Blocks

80% of time code runs at 100% of its ability, 20% of time it runs at 50% of ability: 90% of possible

95% of time code runs at 100% of its ability, 5% of time it runs at 50% of ability: 97.5% of possible
Takeaways

• **Threadblock size choice:**
  – Start with 128-256 threads per block
    • Adjust up/down by what best matches your function
    • Example: stencil codes prefer larger blocks to minimize halos
  – Multiple of warp size (32 threads)
  – If occupancy is critical to performance:
    • Check that block size isn’t precluding occupancy allowed by register and SMEM resources

• **Grid size:**
  – 1,000 or more threadblocks
    • 10s of waves of threadblocks: no need to think about tail effect
    • Makes your code ready for several generations of future GPUs
Summary

• What you need for good GPU performance
  – Expose sufficient parallelism to keep GPU busy
    • General recommendations:
      – 1000+ threadblocks per GPU
      – 1000+ concurrent threads per SM (32+ warps)
  – Maximize memory bandwidth utilization
    • Pay attention to warp address patterns (Have sufficient independent memory accesses to saturate the bus)
  – Minimize warp divergence
    • Keep in mind that instructions are issued per warp

• Use profiling tools to analyze your code
Additional Resources

- **Previous GTC optimization talks**
  - Have different tips/tricks, case studies
  - GTC 2012: GPU Performance Analysis and Optimization
  - GTC 2010: Analysis-Driven Optimization:

- **GTC 2013 talks on performance analysis tools:**
  - S3011: Case Studies and Optimization Using Nsight Visual Studio Edition
  - S3046: Performance Optimization Strategies for GPU-Accelerated Applications

- **Kepler architecture white paper:**

- **Miscellaneous:**
  - Webinar on register spilling:
    - Video: [http://developer.download.nvidia.com/CUDA/training/CUDA_LocalMemoryOptimization.mp4](http://developer.download.nvidia.com/CUDA/training/CUDA_LocalMemoryOptimization.mp4)