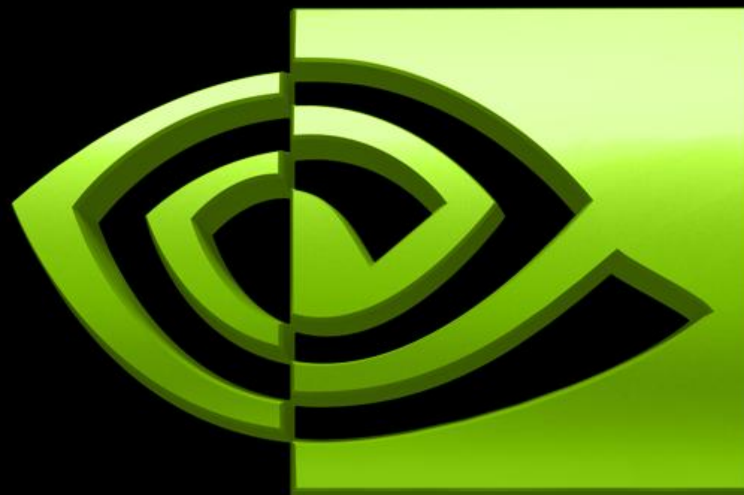


Optimizing CUDA

Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it's better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host

3

Optimize Memory Access

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory
- Optimize for spatial locality in cached texture memory
- In shared memory, avoid high-degree bank conflicts
- Partition camping
  - When global memory access not evenly distributed amongst partitions
  - Problem-size dependent

Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Use one / a few threads to load / compute data shared by all threads
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

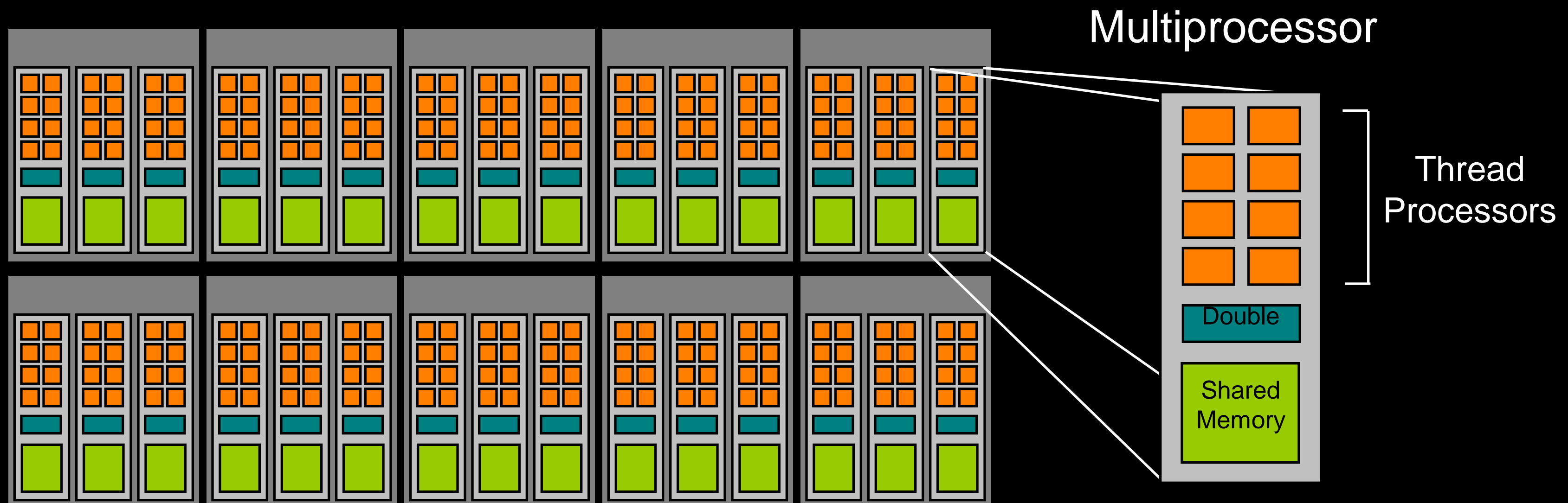
© NVIDIA Corporation 2009

Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

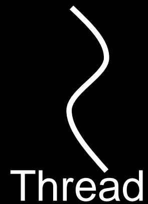
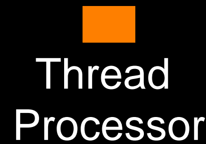
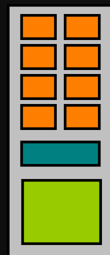
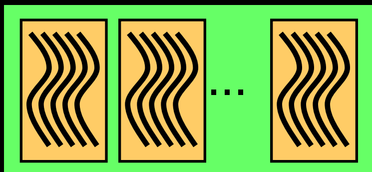
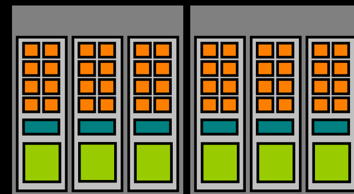
10-Series Architecture

- 240 thread processors execute kernel threads
- 30 multiprocessors, each contains
  - 8 thread processors
  - One double-precision unit
  - Shared memory enables thread cooperation

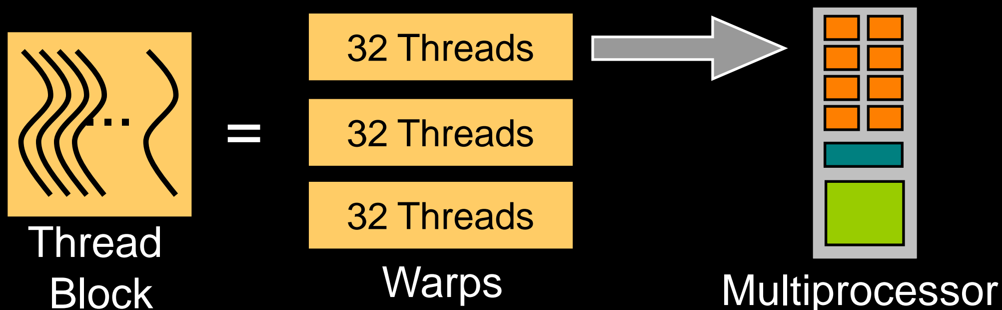


© NVIDIA Corporation 2009

Execution Model

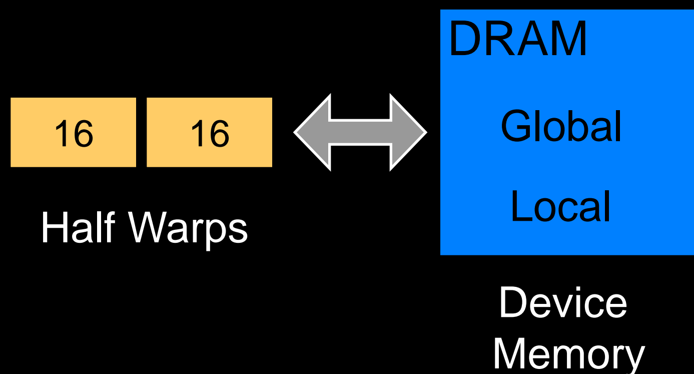
Software	Hardware	
 Thread	 Thread Processor	Threads are executed by thread processors
Thread Block	 Multiprocessor	Thread blocks are executed on multiprocessors Thread blocks do not migrate Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
 Grid	 Device	A kernel is launched as a grid of thread blocks Only one kernel can execute on a device at one time

Warps and Half Warps



A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor



A half-warp of 16 threads can coordinate global memory accesses into a single transaction

Memory Architecture

Memory Architecture

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Host-Device Data Transfers

- Device to host memory bandwidth much lower than device to device bandwidth
  - 4GB/s peak (PCI-e x16 Gen 1) vs. 102 GB/s peak (Tesla C1060)
- Minimize transfers
  - Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory
- Group transfers
  - One large transfer much better than many small ones

Page-Locked Data Transfers

- cudaMallocHost() allows allocation of page-locked ("pinned") host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2
- See the "bandwidthTest" CUDA SDK sample
- Use with caution!!
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits

Overlapping Data Transfers and Computation

- Async and Stream APIs allow overlap of H2D or D2H data transfers with computation
  - CPU computation can overlap data transfers on all CUDA capable devices
  - Kernel computation can overlap data transfers on devices with “Concurrent copy and execution” (roughly compute capability >= 1.1)

- Stream = sequence of operations that execute in order on GPU
  - Operations from different streams can be interleaved
  - Stream ID used as argument to async calls and kernel launches

© NVIDIA Corporation 2009

1

Asynchronous Data Transfers

- Asynchronous host-device memory copy returns control immediately to CPU
  - cudaMemcpyAsync(dst, src, size, dir, stream);
  - requires pinned host memory (allocated with "cudaMallocHost")
- Overlap CPU computation with data transfer
  - 0 = default stream

```
cudaMemcpyAsync(a_d, a_h, size,
        cudaMemcpyHostToDevice, 0);
cpuFunction();
cudaThreadSynchronize();
kernel<<<grid, block>>>(dst);
```

overlapped

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

GPU/CPU Synchronization

- Context based
  - cudaThreadSynchronize()
    - Blocks until all previously issued CUDA calls from a CPU thread complete
- Stream based
  - cudaStreamSynchronize(stream)
    - Blocks until all CUDA calls issued to given stream complete
  - cudaStreamQuery(stream)
    - Indicates whether stream is idle
    - Returns cudaSuccess, cudaErrorNotReady, ...
    - Does not block CPU thread

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

GPU/CPU Synchronization

- Stream based using events
  - Events can be inserted into streams:

    cudaEventRecord(event, stream)
  - Event is recorded then GPU reaches it in a stream
    - Recorded = assigned a timestamp (GPU clocktick)
    - Useful for timing

  - **`cudaEventSynchronize(event)`**
    - Blocks until given event is recorded

  - cudaEventQuery(event)
    - Indicates whether event has recorded
    - Returns cudaSuccess, cudaErrorNotReady, ...
    - Does not block CPU thread

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Overlapping kernel and data transfer

- Requires:
  - "Concurrent copy and execute"
    - deviceOverlap field of a cudaDeviceProp variable
  - Kernel and transfer use different, non-zero streams
    - A CUDA call to stream-0 blocks until all previous calls complete and cannot be overlapped

- Example:

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
cudaStreamSynchronize(stream2);
```

overlapped

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data Transfers between host and device
  - Device memory optimizations
    - Matrix transpose study
      - Measuring performance - effective bandwidth
      - Coalescing
      - Shared memory bank conflicts
      - Partition camping
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Matrix Transpose

- Transpose 2048x2048 matrix of floats
- Performed out-of-place
  - Separate input and output matrices
- Use tile of 32x32 elements, block of 32x8 threads
  - Each thread processes 4 matrix elements
  - In general tile and block size are fair game for optimization
- Process
  - Get the right answer
  - Measure effective bandwidth (relative to theoretical or reference case)
  - Address global memory coalescing, shared memory bank conflicts, and partition camping while repeating above steps

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Theoretical Bandwidth

- **Device Bandwidth of GTX 280**
  - **1107 * 10^6 * (512 / 8) * 2 / 1024^3 = 131.9 GB/s**

    (1107 * 10^6: Memory clock (Hz); (512 / 8): Memory interface (bytes); 2: DDR)
  - **Specs report 141 GB/s**
    - **Use 10^9 B/GB conversion rather than 1024^3**
    - **Whichever you use, be consistent**

Effective Bandwidth

- Transpose Effective Bandwidth
  - 2048^2 * 4 B/element / 1024^3 * 2 / (time in secs) = GB/s

Matrix size (bytes)

Read and write

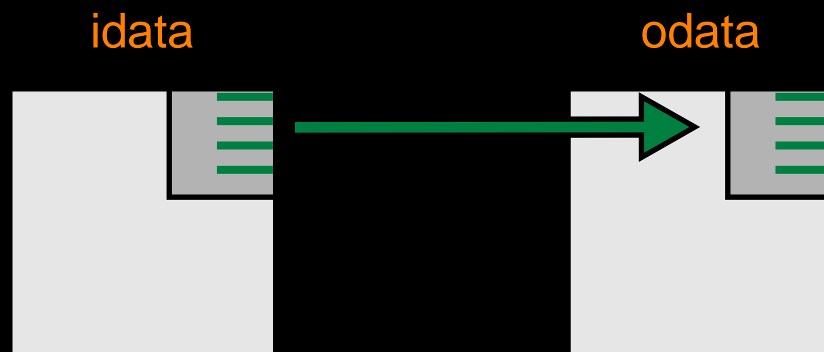
- Reference Case - Matrix Copy
  - Transpose operates on tiles - need better comparison than raw device bandwidth
  - Look at effective bandwidth of copy that uses tiles

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Matrix Copy Kernel

```
__global__ void copy(float *odata, float *idata, int width,
                     int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index  = xIndex + width*yIndex;

  for (int i = 0; i < TILE_DIM;  i += BLOCK_ROWS) {
    odata[index+i*width] = idata[index+i*width];
  }
}
```



Elements copied by a half-warp of threads

TILE_DIM = 32
BLOCK_ROWS = 8

32x32 tile
32x8 thread block

idata and odata in global memory

Matrix Copy Kernel Timing

NVIDIA

- Measure elapsed time over loop
- Looping/timing done in two ways:
  - Over kernel launches (nreps = 1)
    - Includes launch/indexing overhead
  - Within the kernel over loads/stores (nreps > 1)
    - Amortizes launch/indexing overhead

```
__global__ void copy(float *odata, float* idata, int width,
                     int height, int nreps)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index  = xIndex + width*yIndex;

  for (int r = 0; r < nreps; r++) {
    for (int i = 0; i < TILE_DIM;  i += BLOCK_ROWS) {
      odata[index+i*width] = idata[index+i*width];
    }
  }
}
```

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

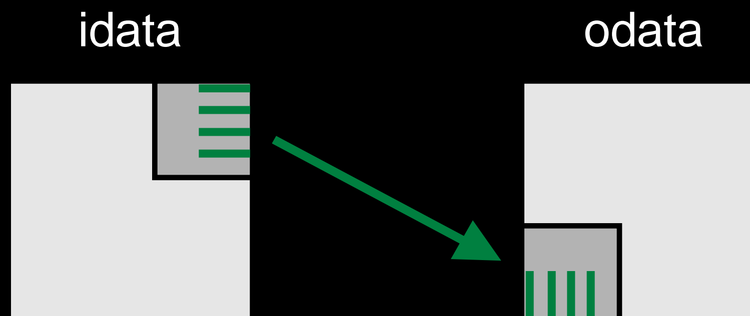
Naïve Transpose

- **Similar to copy**
  - **Input and output matrices have different indices**

```
__global__ void transposeNaive(float *odata, float* idata, int width,
                               int height, int nreps)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i] = idata[index_in+i*width];
    }
  }
}
```



© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Effective Bandwidth

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2

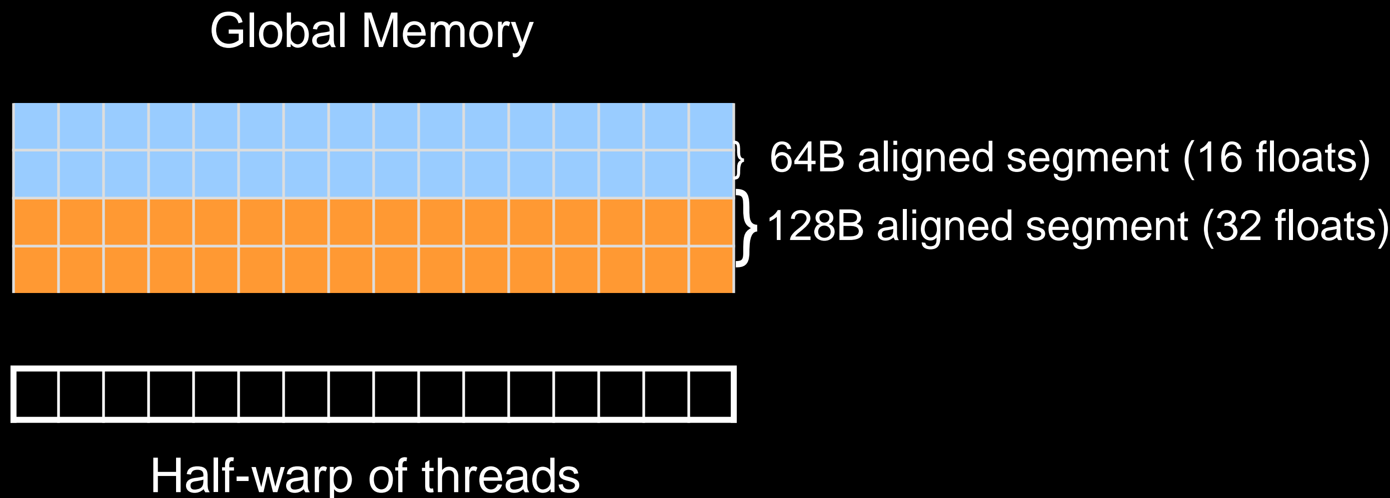
© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data Transfers between host and device
  - Device memory optimizations
    - Matrix transpose study
      – Measuring performance - effective bandwidth
      – Coalescing
      – Shared memory bank conflicts
      – Partition camping
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Coalescing

- **Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met**
- **Depends on compute capability**
  - **1.0 and 1.1 have stricter access requirements**

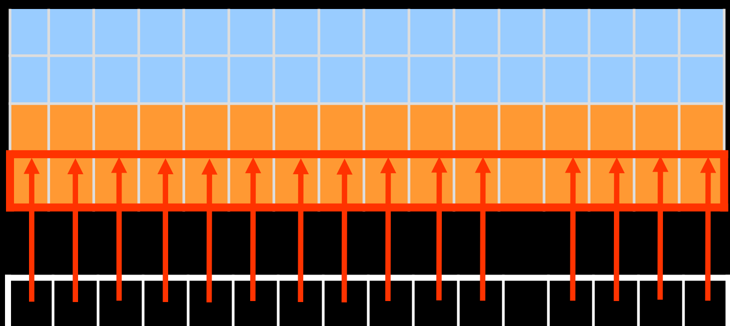


© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

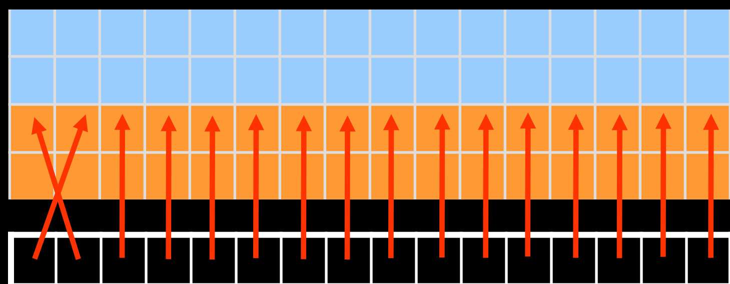
Coalescing
Compute capability 1.0 and 1.1

- **K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate**

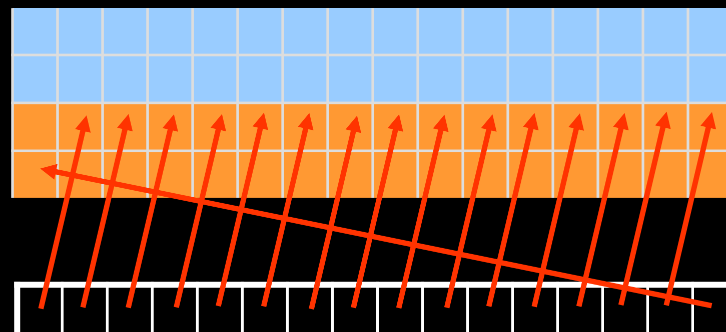
Coalesces – 1 transaction



Out of sequence – 16 transactions



Misaligned – 16 transactions

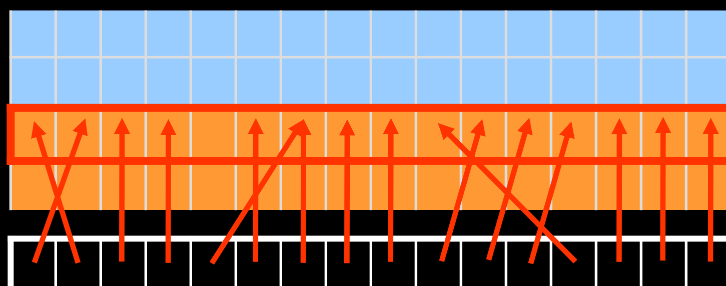


© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Coalescing

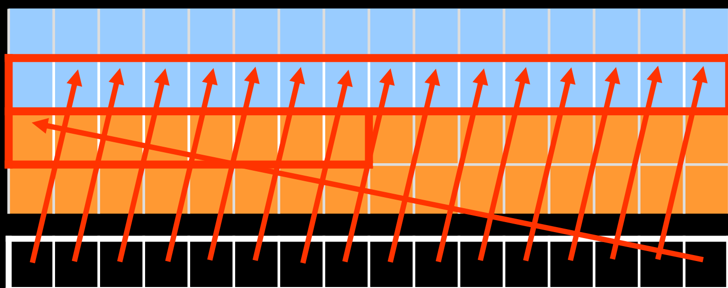
Compute capability 1.2 and higher

- **Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words**
- **Smaller transactions may be issued to avoid wasted bandwidth due to unused words**

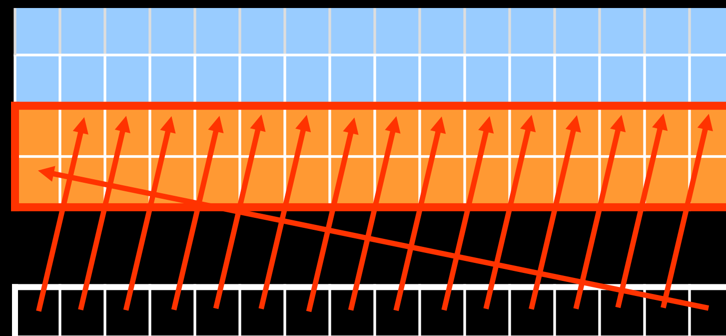


1 transaction - 64B segment

2 transactions - 64B and 32B segments



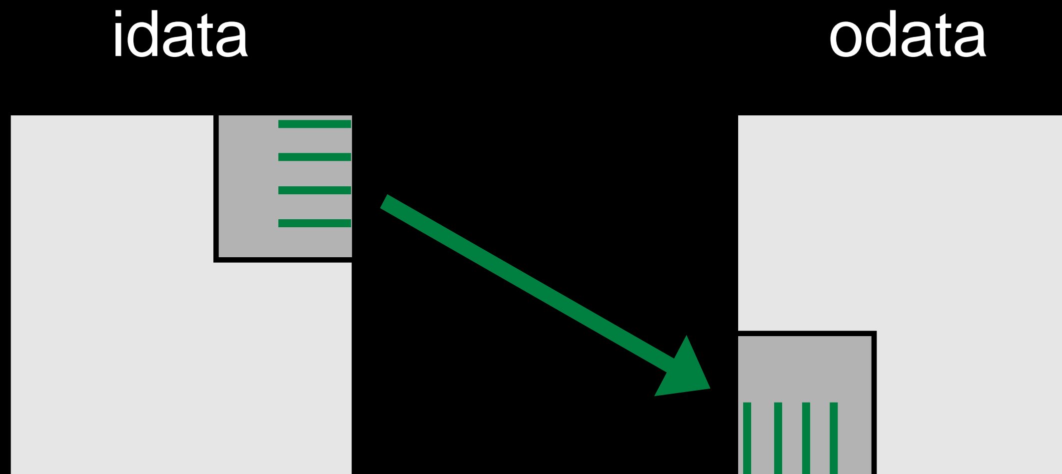
1 transaction - 128B segment



© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Coalescing in Transpose

- **Naïve transpose coalesces reads, but not writes**



Elements transposed by a half-warp of threads

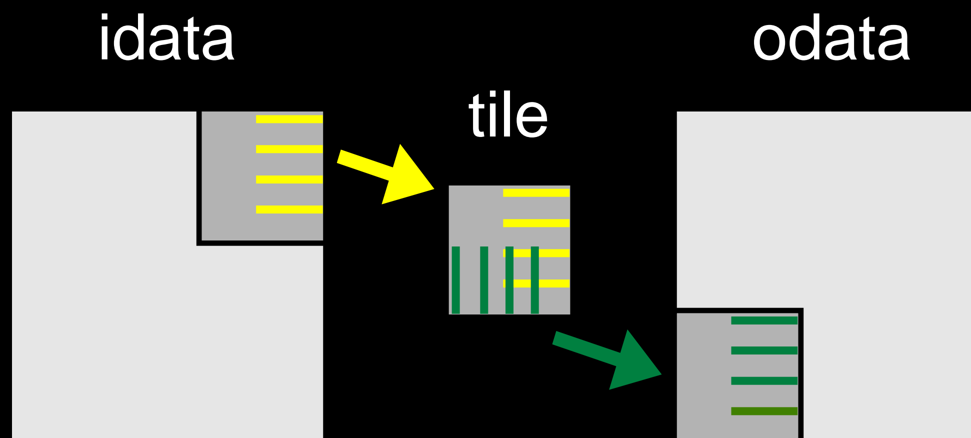
Shared Memory

- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Coalescing through shared memory

- **Access columns of a tile in shared memory to write contiguous data to global memory**
- **Requires __syncthreads() since threads write data read by other threads**



Elements transposed by a half-warp of threads

Coalescing through shared memory

```
__global__ void transposeCoalesced(float *odata, float *idata, int width,
                                   int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
    }
  }
}
```

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Effective Bandwidth

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

Uses shared memory tile and __syncthreads()

Outline

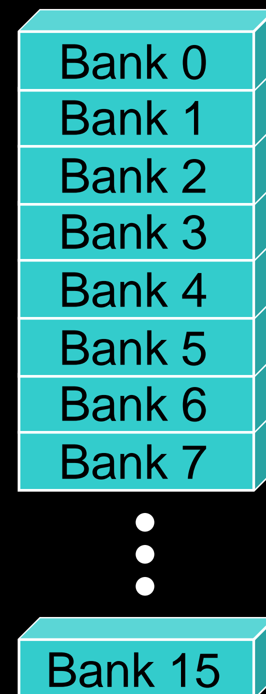
NVIDIA

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Matrix transpose study
      - Measuring performance - effective bandwidth
      - Coalescing
      - Shared memory bank conflicts
      - Partition camping
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Shared Memory Architecture

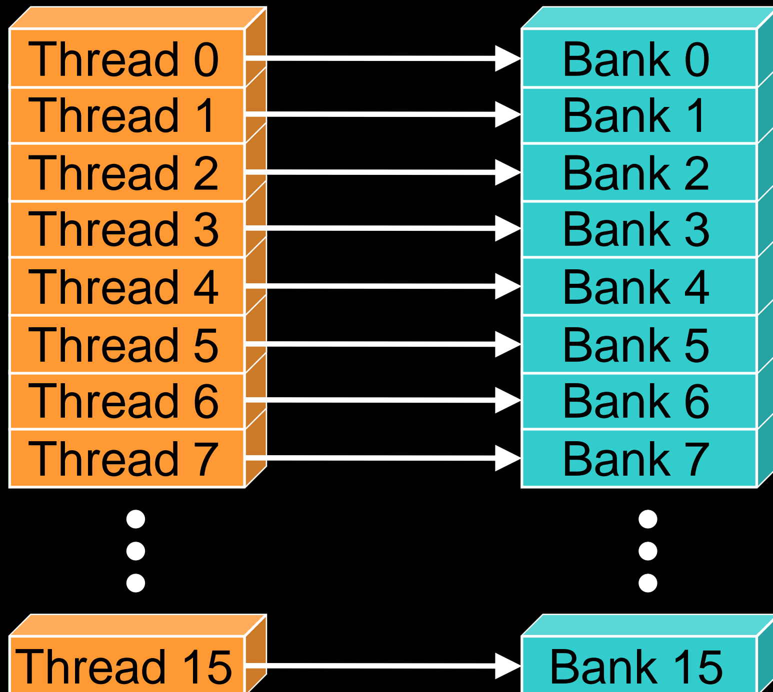
- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



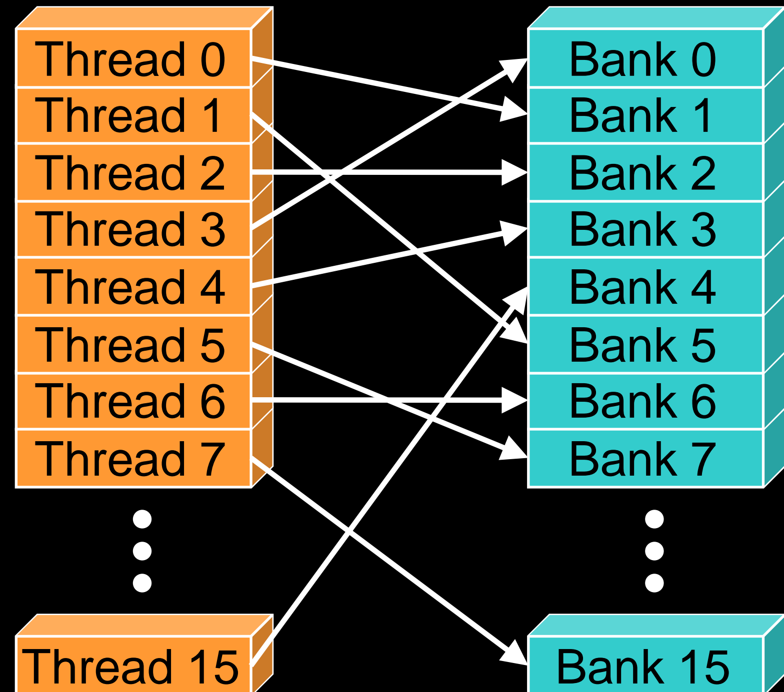
© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1



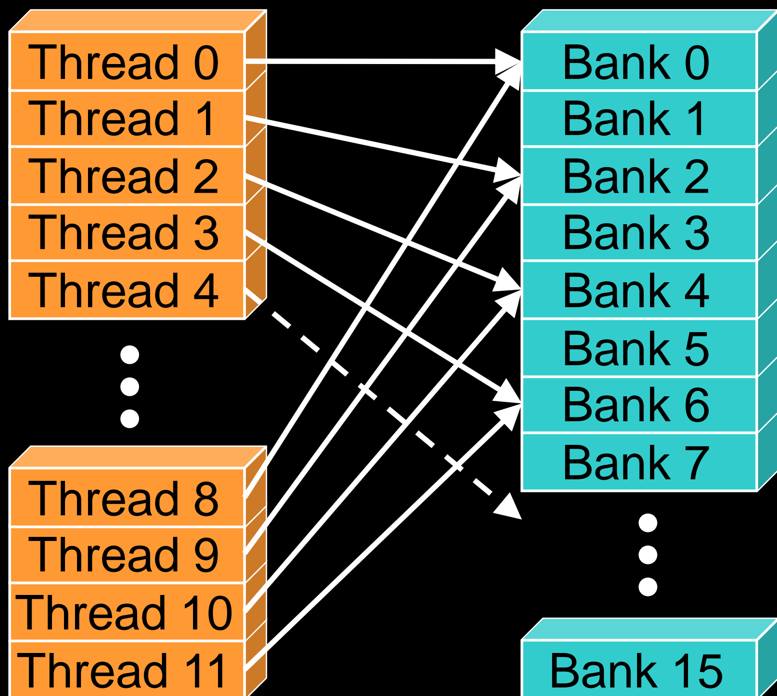
- No Bank Conflicts
  - Random 1:1 Permutation



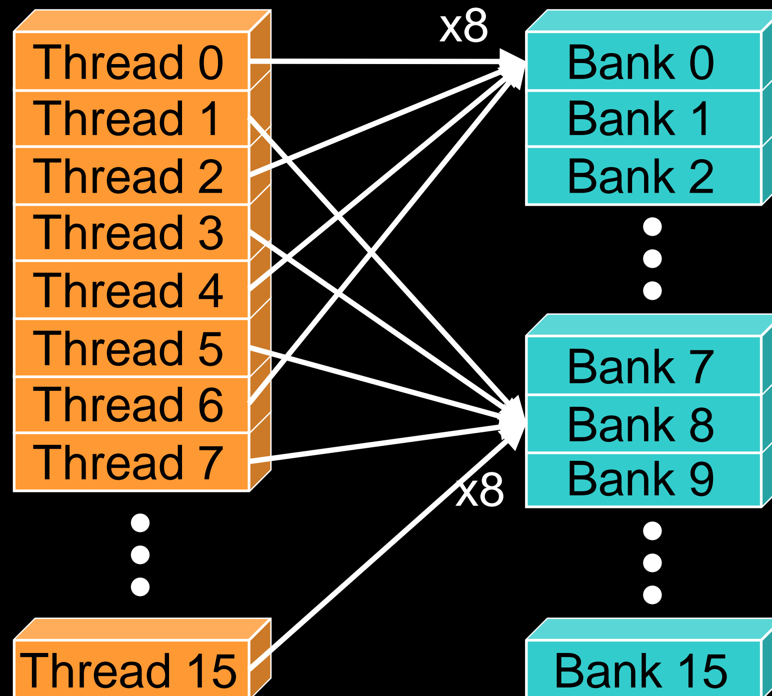
© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

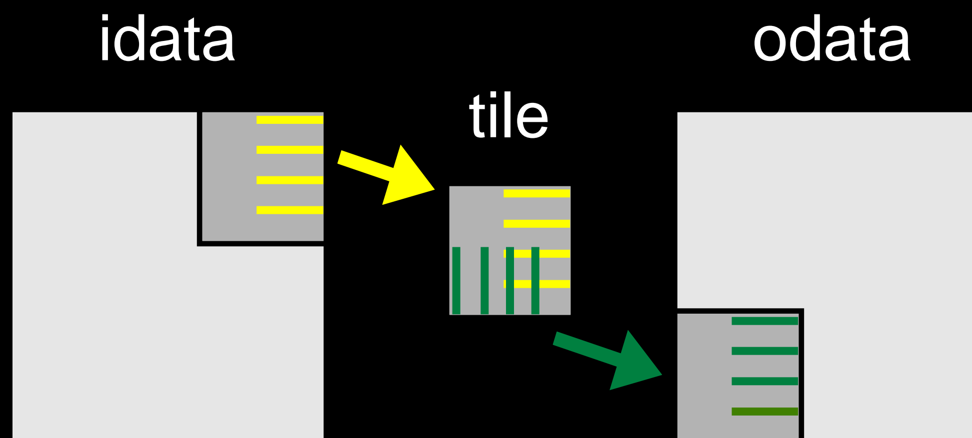
Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts
- warp_serialize profiler signal reflects conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Bank Conflicts in Transpose

- 32x32 shared memory tile of floats
  - Data in columns k and k+16 are in same bank
  - 16-way bank conflict reading half columns in tile
- Solution - pad shared memory array
  - __shared__ float tile[TILE_DIM][TILE_DIM+1];
  - Data in anti-diagonals are in same bank



© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Effective Bandwidth

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Matrix transpose study
      - Measuring performance - effective bandwidth
      - Coalescing
      - Shared memory bank conflicts
      - Partition camping
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

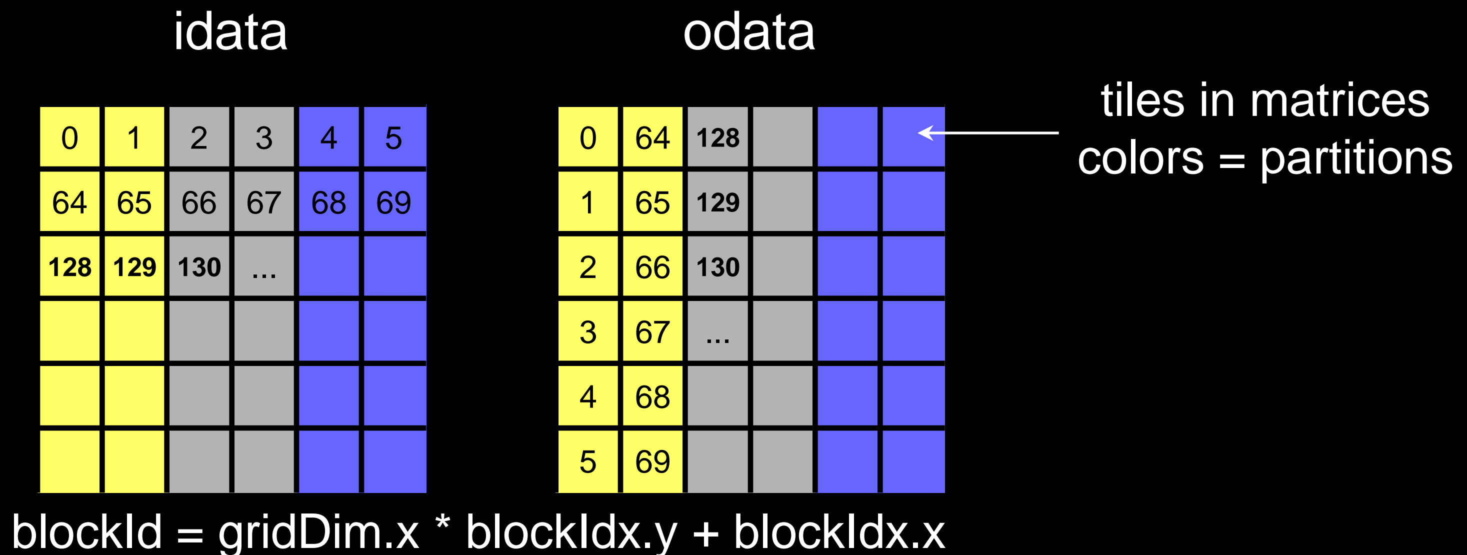
Partition Camping

- Global memory accesses go through partitions
  - 6 partitions on 8-series GPUs, 8 partitions on 10-series GPUs
  - Successive 256-byte regions of global memory are assigned to successive partitions
- For best performance:
  - Simultaneous global memory accesses GPU-wide should be distributed evenly amongst partitions
- Partition Camping occurs when global memory accesses at an instant use a subset of partitions
  - Directly analogous to shared memory bank conflicts, but on a larger scale

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Partition Camping in Transpose

- Partition width = 256 bytes = 64 floats
  - Twice width of tile
- On GTX280 (8 partitions), data 2KB apart map to same partition
  - 2048 floats divides evenly by 2KB => columns of matrices map to same partition



© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Partition Camping Solutions

- Pad matrices (by two tiles)
  - In general might be expensive/prohibitive memory-wise
- Diagonally reorder blocks
  - Interpret blockIdx.y as different diagonal slices and blockIdx.x as distance along a diagonal

idata

0	64	128			
	1	65	129		
		2	66	130	
			3	67	...
				4	68
					5

odata

0					
64	1				
128	65	2			
	129	66	3		
		130	67	4	
			...	68	5

blockId = gridDim.x * blockIdx.y + blockIdx.x

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Diagonal Transpose

NVIDIA

```
__global__ void transposeDiagonal(float *odata, float *idata, int width,
                                  int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int blockIdx_y = blockIdx.x;
  int blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;

  int xIndex = blockIdx_x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx_y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx_y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx_x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }
    __syncthreads();
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
    }
  }
}
```

Add lines to map diagonal to Cartesian coordinates

Replace blockIdx.x with blockIdx_x, blockIdx.y with blockIdx_y

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Diagonal Transpose

- Previous slide for square matrices (width == height)
- More generally:

```
if (width == height) {
  blockIdx_y = blockIdx.x;
  blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
} else {
  int bid = blockIdx.x + gridDim.x*blockIdx.y;
  blockIdx_y = bid%gridDim.y;
  blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
}
```

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Effective Bandwidth

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Diagonal	69.5	78.3

Transpose Summary

- Coalescing and shared memory bank conflicts are small-scale phenomena
  - Deal with memory access within half-warp
  - Problem-size independent

- Partition camping is a large-scale phenomena
  - Deals with simultaneous memory accesses by warps on different multiprocessors
  - Problem size dependent
    - Wouldn’t see in (2048+32)^2 matrix

- Coalescing is generally the most critical

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Outline

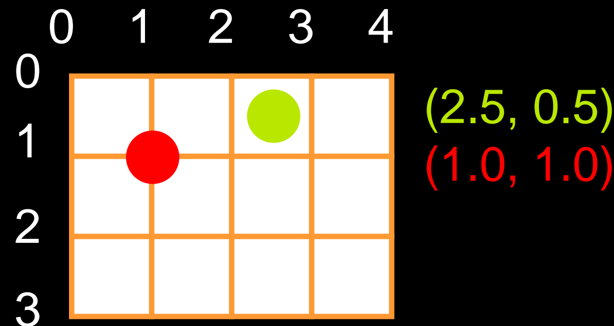
- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Matrix transpose study
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Textures in CUDA

- Texture is an object for reading data
- Benefits:
  - Data is cached (optimized for 2D locality)
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable in 1D, 2D, or 3D
    - Using integer or normalized coordinates
- Usage:
  - CPU code binds data to a texture object
  - Kernel reads data by calling a fetch function

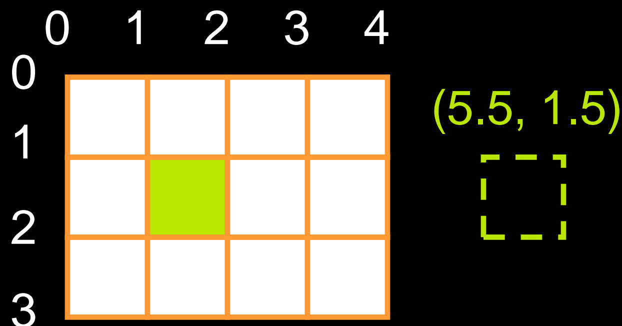
© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Texture Addressing



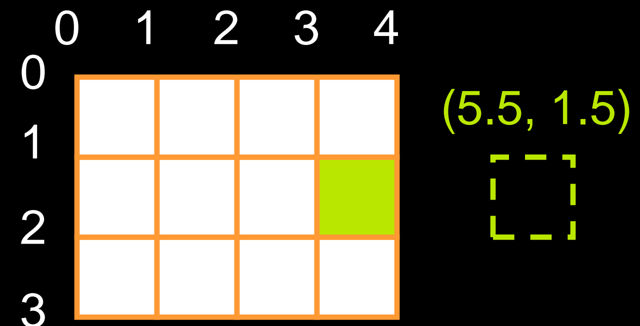
Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



Clamp

- Out-of-bounds coordinate is replaced with the closest boundary



© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Two CUDA Texture Types

NVIDIA

- Bound to linear memory
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes
- Bound to CUDA arrays
  - CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- Both:
  - Return either element type or normalized float

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

CUDA Texturing Steps

- Host (CPU) code:
  - Allocate/obtain memory (global linear, or CUDA array)
  - Create a texture reference object
    - Currently must be at file-scope
  - Bind the texture reference to memory/array
  - When done:
    - Unbind the texture reference, free resources
- Device (kernel) code:
  - Fetch using texture reference
  - Linear memory textures:
    - tex1Dfetch()
  - Array textures:
    - tex1D() or tex2D() or tex3D()

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Occupancy

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy
- Occupancy = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently
- Limited by resource usage:
  - Registers
  - Shared memory

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Grid/Block Size Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute
- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren't waiting at a __syncthreads() keep the hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Register Dependency

- Read-after-write register dependency
  - Instruction's result can be read ~11 cycles later
  - Scenarios: CUDA: PTX:

```
x = y + 5;
z = x + 3;
```

```
add.f32   $f3, $f1, $f2
add.f32   $f5, $f3, $f4
```

```
s_data[0] += 3;
```

```
ld.shared.f32  $f3, [$r31+0]
add.f32        $f3, $f3, $f4
```

- To completely hide the latency:
  - Run at least 192 threads (6 warps) per multiprocessor
    - At least 25% occupancy
  - Threads do not have to belong to the same thread block

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Register Pressure

- Hide latency by using more threads per SM
- Limiting Factors:
  - Number of registers per kernel
    - 8K/16K per SM, partitioned among concurrent threads
  - Amount of shared memory
    - 16KB per SM, partitioned among concurrent threadblocks
- Compile with –ptxas-options=-v flag
- Use –maxrregcount=N flag to NVCC
  - N = desired maximum registers / kernel
  - At some point "spilling" into local memory may occur
    - Reduces performance – local memory is slow

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Occupancy Calculator

Microsoft Excel - CUDA_Occupancy_calculator.xls

File Edit View Insert Format Tools Data Window Help

MyRegCount fx 20

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculato

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select a GPU from the list (click):	G80	(Help)
2.) Enter your resource usage:		
Threads Per Block	192	(Help)
Registers Per Thread	20	
Shared Memory Per Block (bytes)	68	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	384	(Help)
Active Warps per Multiprocessor	12	
Active Thread Blocks per Multiprocessor	2	
Occupancy of each Multiprocessor	50%	
Maximum Simultaneous Blocks per GPU	32	
(Note: This assumes there are at least this many blocks)		
Physical Limits for GPU:	G80	
Multiprocessors per GPU	16	
Threads / Warp	32	
Warps / Multiprocessor	24	
Threads / Multiprocessor	768	
Thread Blocks / Multiprocessor	8	
Total # of 32-bit registers / Multiprocessor	8192	
Shared Memory / Multiprocessor (bytes)	16384	
Allocation Per Thread Block		
Warps	6	
Registers	3840	
Shared Memory	512	
These data are used in computing the occupancy data in blue		
Maximum Thread Blocks Per Multiprocessor	Blocks	
Limited by Max Warps / Multiprocessor	4	
Limited by Registers / Multiprocessor	2	
Limited by Shared Memory / Multiprocessor	32	
Thread Block Limit Per Multiprocessor is the minimum of these 3		
CUDA Occupancy Calculator		
Version:	1.1	
Copyright and License		

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Varying Block Size — Multiprocessor Warp Occupancy vs. Threads Per Block (My Block Size 192)

Varying Register Count — Multiprocessor Warp Occupancy vs. Registers Per Thread (My Register Count 20)

Varying Shared Memory Usage — Multiprocessor Warp Occupancy vs. Registers Per Thread (My Shared Memory 68)

Calculator / Help / GPU Data / Copyright & License

Ready

Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
- More threads per block == better memory latency hiding
- But, more threads per block == fewer registers per thread
  - Kernel invocations can fail if too many registers are used
- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation, so experiment!

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

BUT …

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - (It all comes down to arithmetic intensity and available parallelism)

Parameterize Your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of multiprocessors
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration

Outline

- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

CUDA Instruction Performance

- Instruction cycles (per warp) = sum of
  - Operand read cycles
  - Instruction execution cycles
  - Result update cycles
- Therefore instruction throughput depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth
- "Cycle" refers to the multiprocessor clock rate
  - 1.3 GHz on the Tesla C1060, for example

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Maximizing Instruction Throughput

- Maximize use of high-bandwidth memory
  - Maximize use of shared memory
  - Minimize accesses to global memory
  - Maximize coalescing of global memory accesses

- Optimize performance by overlapping memory accesses with HW computation
  - High arithmetic intensity programs
    - i.e. high ratio of math to memory transactions
  - Many concurrent threads

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Arithmetic Instruction Throughput

- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply

- Integer divide and modulo are more expensive
  - Compiler will convert literal power-of-2 divides to shifts
    - But we have seen it miss some cases
  - Be explicit in cases where compiler can’t tell that divisor is a power of 2!
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Arithmetic Instruction Throughput

- The intrinsics reciprocal, reciprocal square root, sin/cos, log, exp prefixed with "__" 16 cycles per warp
  - Examples: __rcp(), __sin(), __exp()
- Other functions are combinations of the above
  - y / x == rcp(x) * y takes 20 cycles per warp
  - sqrt(x) == x * rsqrt(x) takes 20 cycles per warp

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Runtime Math Library

NVIDIA

- There are two types of runtime math operations
  - __func(): direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x,y)
- The -use_fast_math compiler option forces every func() to compile to __func()

GPU results may not match CPU

- Many variables: hardware, compiler, optimization settings
- CPU operations aren't strictly limited to 0.5 ulp
  - Sequences of operations can be more accurate due to 80-bit extended precision ALUs
- Floating-point arithmetic is not associative!

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

FP Math is Not Associative!

- In symbolic math, (x+y)+z == x+(y+z)
- This is not necessarily true for floating-point addition
  - Try $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ in the above equation
- When you parallelize computations, you potentially change the order of operations
- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution

Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths must be serialized

- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - if (threadIdx.x > 2) { }
    - Branch granularity < warp size
  - Example without divergence:
    - if (threadIdx.x / WARP_SIZE > 2) { }
    - Branch granularity is a whole multiple of warp size

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009

Summary

- GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
  - Use parallelism efficiently
  - Coalesce memory accesses if possible
  - Take advantage of shared memory
  - Explore other memory spaces
    - Texture
    - Constant
  - Reduce bank conflicts
  - Avoid partition camping

© NVIDIA Corporation 2008
© NVIDIA Corporation 2009