

**nVIDIA®**

## 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

# 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

# Outline

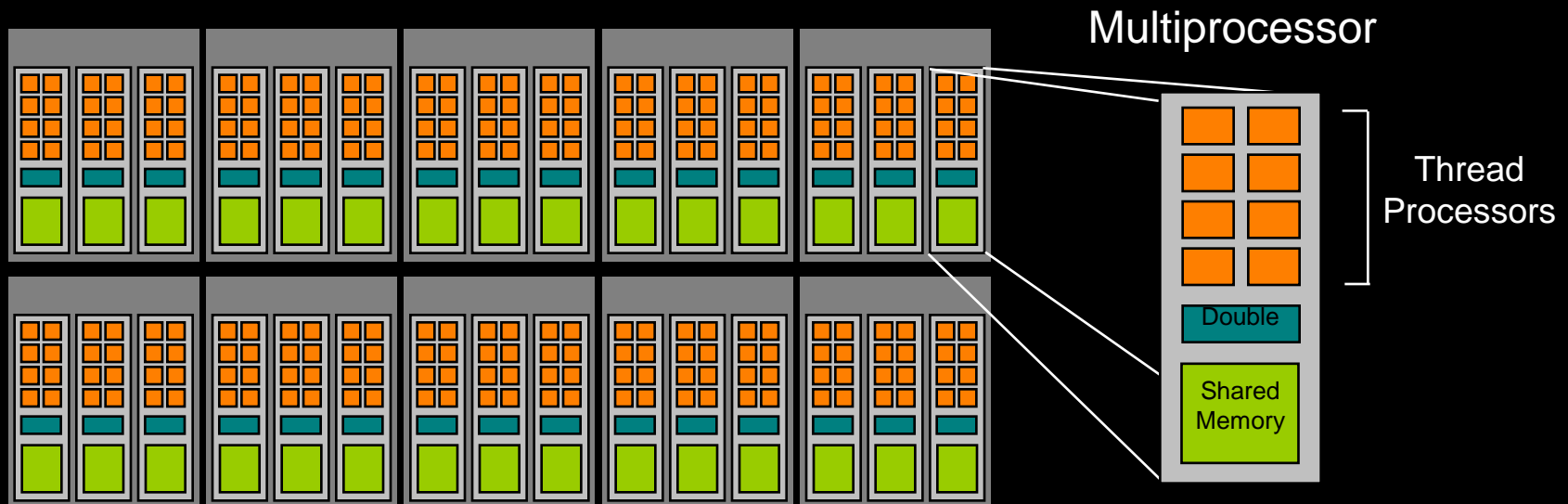


- Overview
- **Hardware**
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

# 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



# Execution Model



Software

Hardware



Threads are executed by thread processors



Thread Block

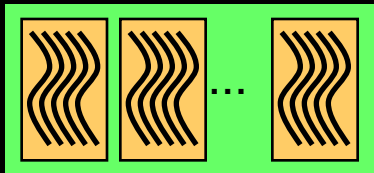


Multiprocessor

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

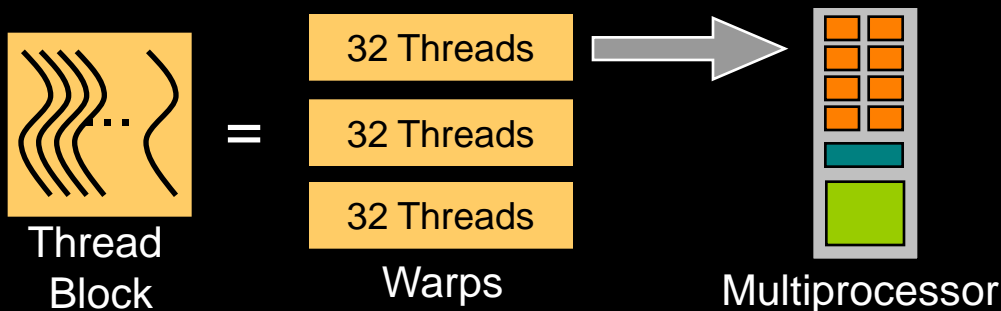


Device

A kernel is launched as a grid of thread blocks

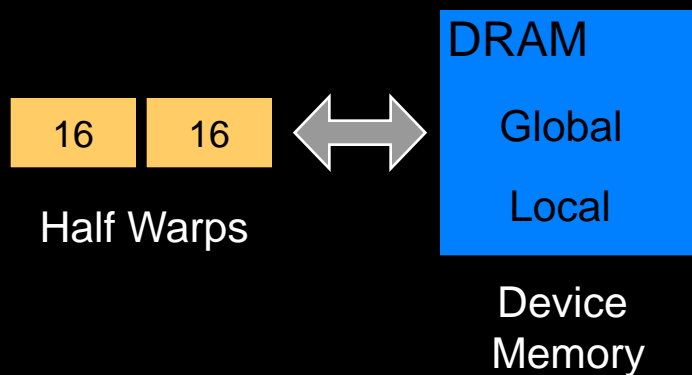
Only one kernel can execute on a device at one time

# Warps and Half Warps



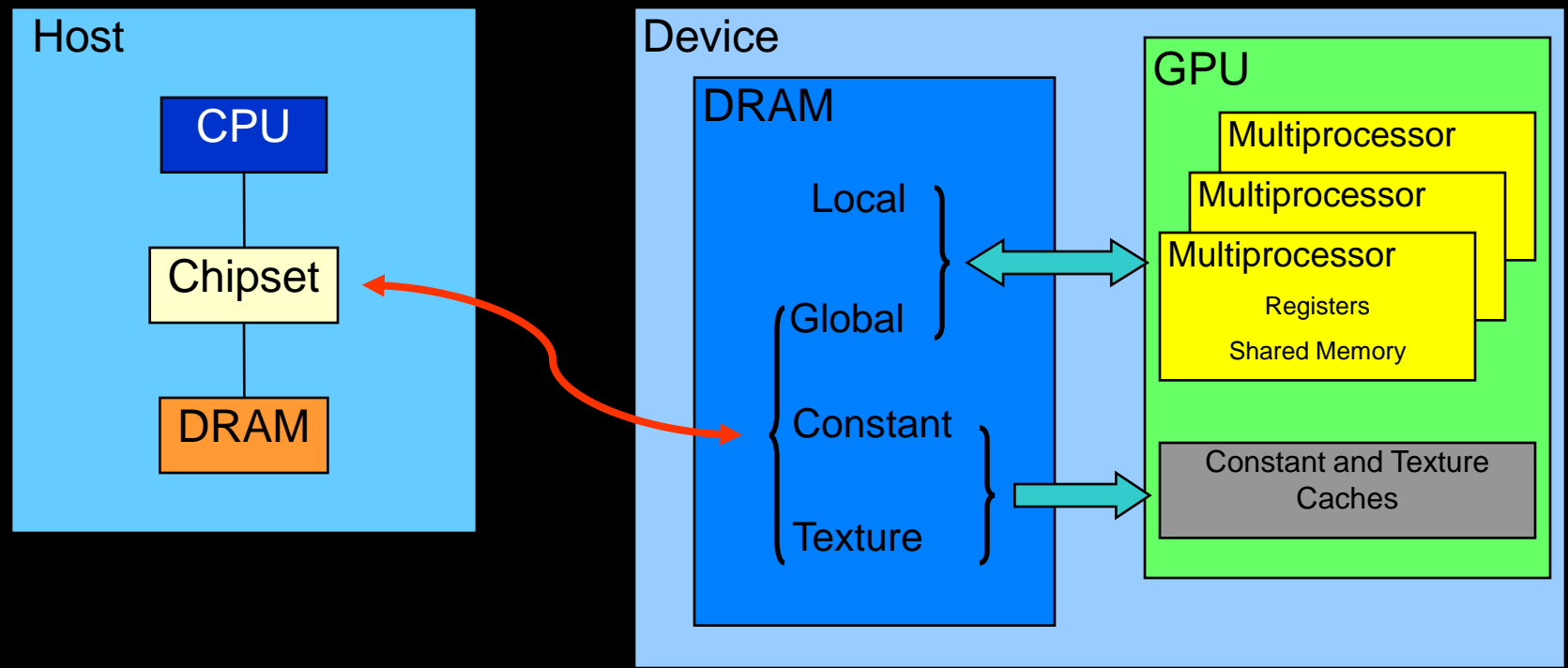
A thread block consists of 32-thread warps

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  $\geq 1.1$ )
- **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

# 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

# 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

# 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

# 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

# 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**

# Theoretical Bandwidth



- **Device Bandwidth of GTX 280**

- $$\underbrace{1107 * 10^6}_{\text{Memory clock (Hz)}} * \underbrace{(512 / 8)}_{\text{Memory interface (bytes)}} * \overset{\text{DDR}}{\downarrow} 2 / 1024^3 = 131.9 \text{ GB/s}$$

- **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**

- $$\underbrace{2048^2 * 4 \text{ B/element}}_{\text{Matrix size (bytes)}} / \underbrace{1024^3 * 2}_{\text{Read and write}} / (\text{time in secs}) = \text{GB/s}$$

- **Reference Case - Matrix Copy**

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

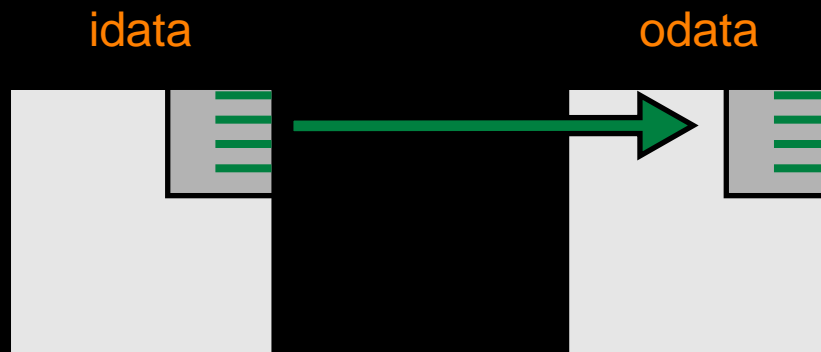
# Matrix Copy Kernel

```

__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

- 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];  
        }  
    }  
}
```

# Naïve Transpose

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

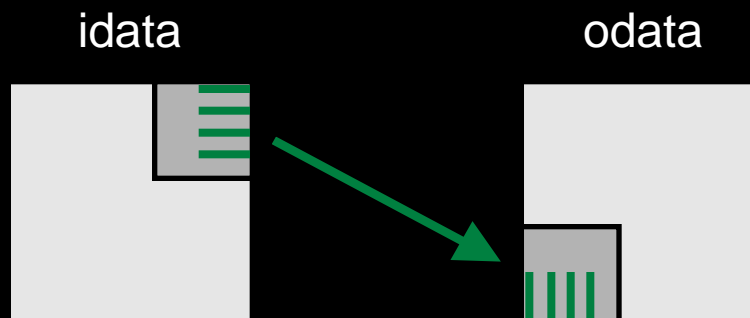
```

__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];
    }
  }
}

```



# Effective Bandwidth



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

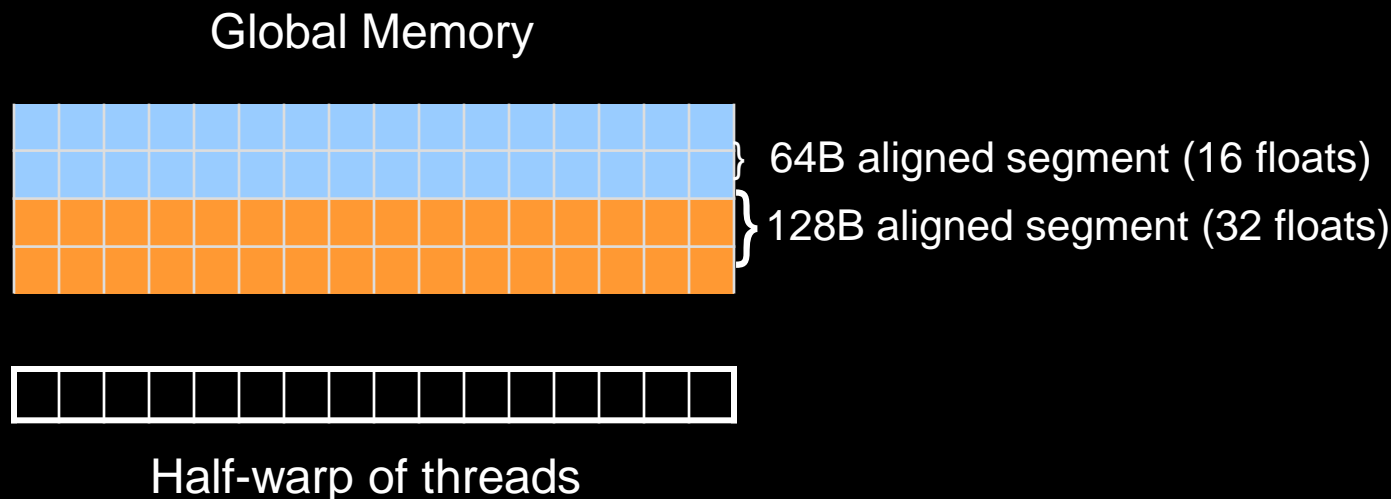
# Outline

- 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

Examples – float (32-bit) data

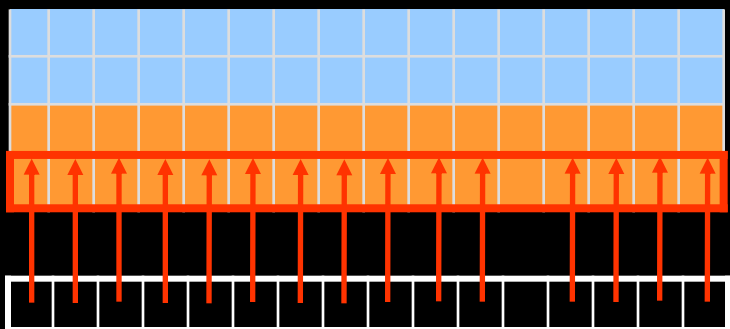


# Coalescing

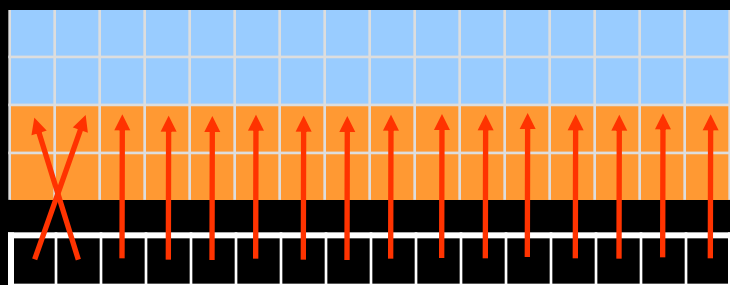
## Compute capability 1.0 and 1.1

- 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

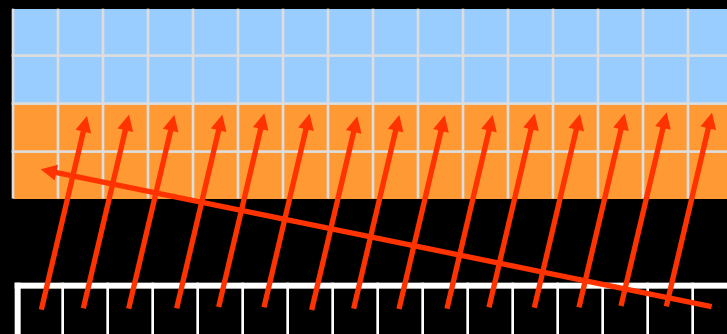
Coalesces – 1 transaction



Out of sequence – 16 transactions



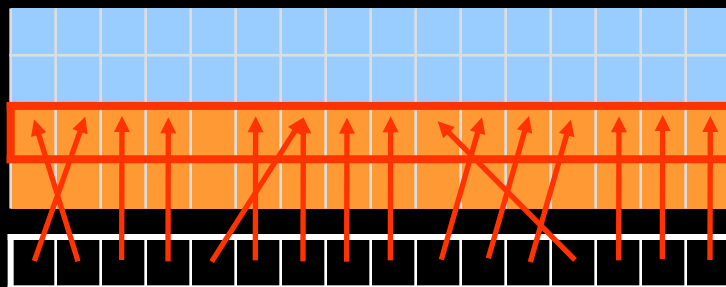
Misaligned – 16 transactions



# Coalescing

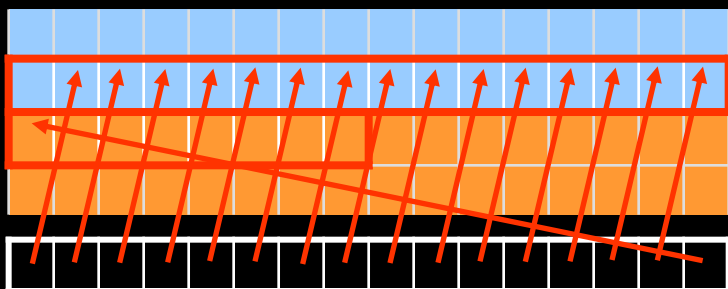
## Compute capability 1.2 and higher

- 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

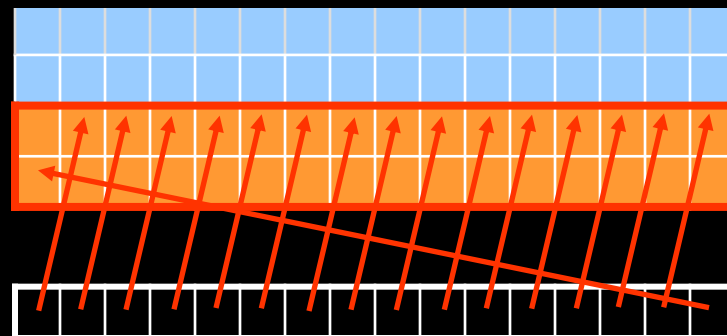


1 transaction - 64B segment

2 transactions - 64B and 32B segments



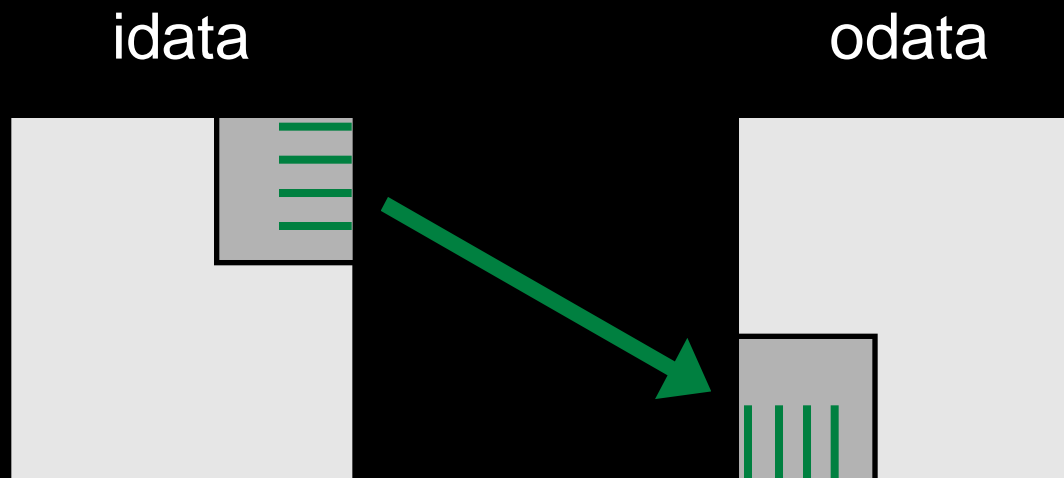
1 transaction - 128B segment



# Coalescing in Transpose



- Naïve transpose coalesces reads, but not writes



Elements transposed by a half-warp of threads

# Shared Memory

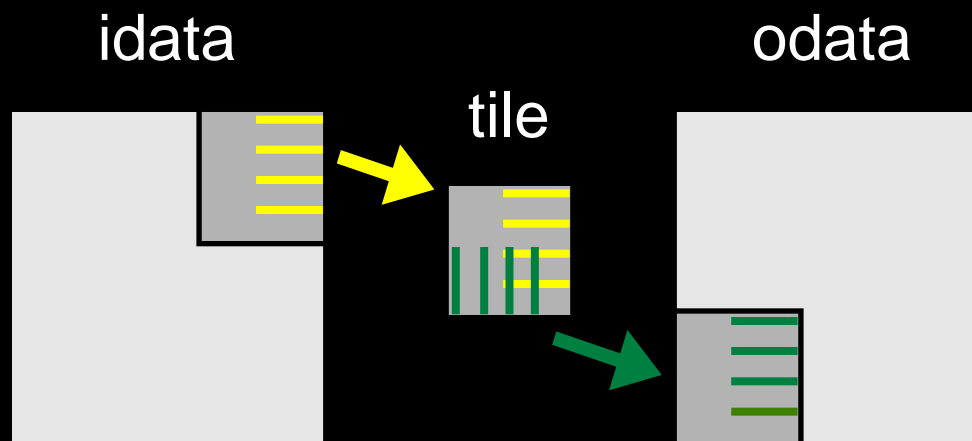


- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

# 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];
        }
    }
}
```

# Effective Bandwidth



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

Uses shared  
memory tile  
and  
\_\_syncthreads()



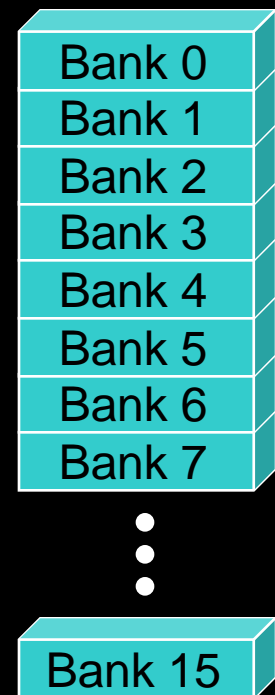
# Outline

- 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

# 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

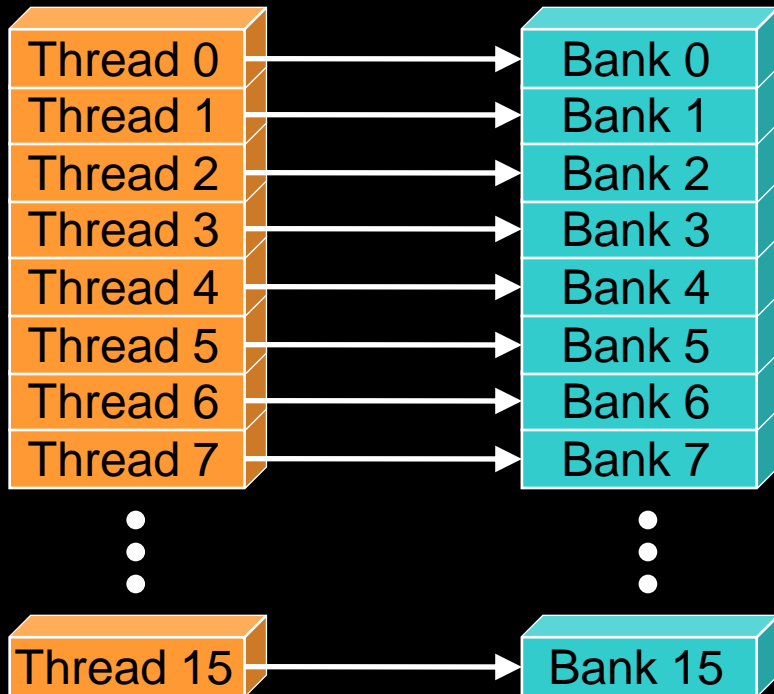


# Bank Addressing Examples



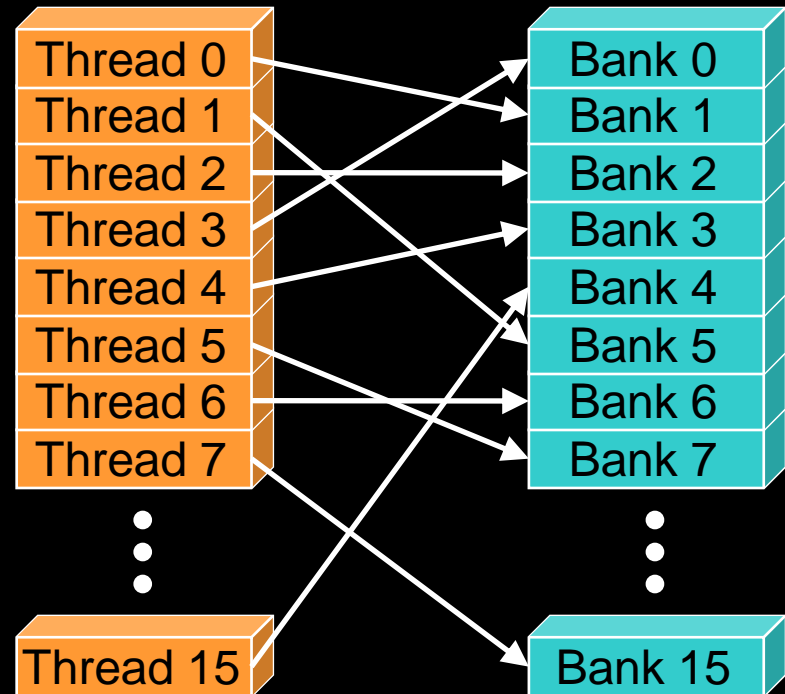
## No Bank Conflicts

- Linear addressing  
stride == 1



## No Bank Conflicts

- Random 1:1 Permutation

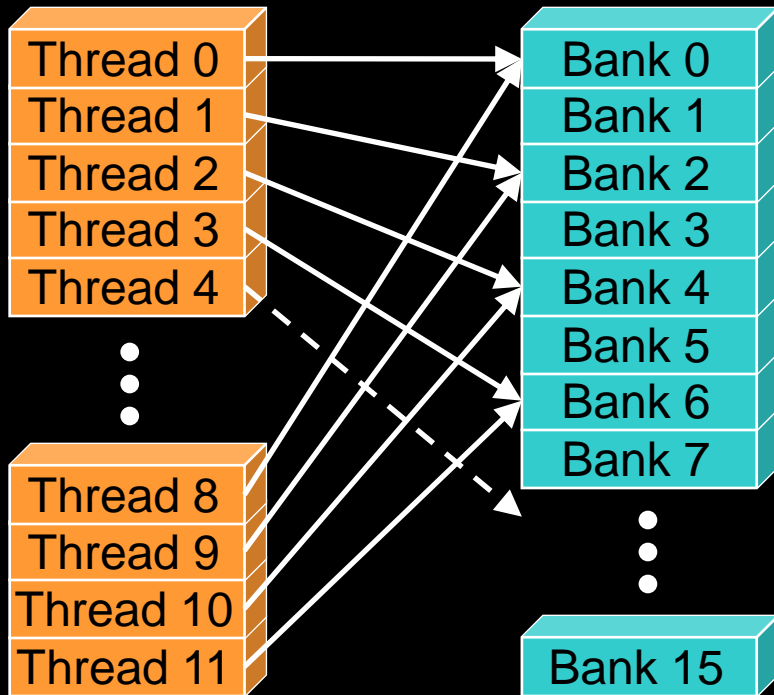


# Bank Addressing Examples



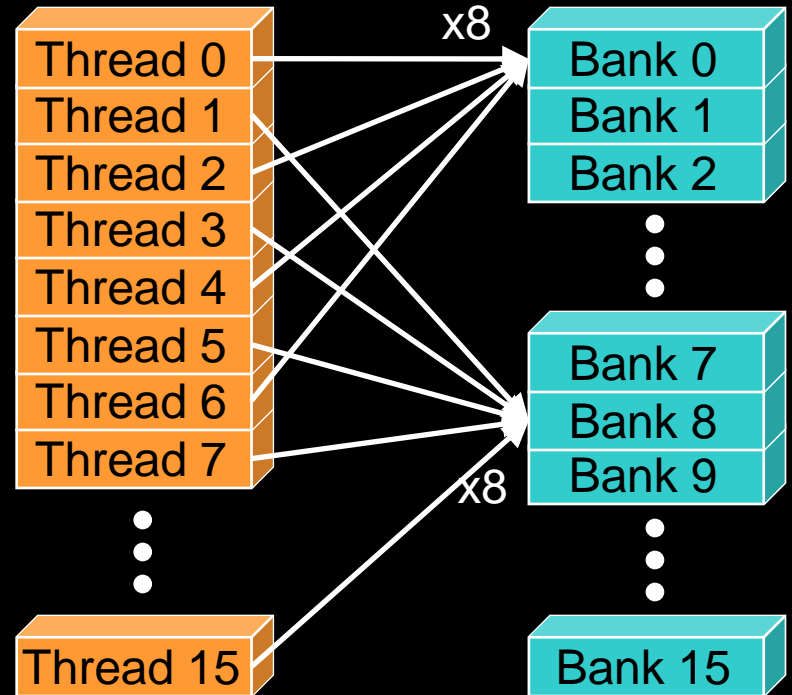
## 2-way Bank Conflicts

- Linear addressing stride == 2



## 8-way Bank Conflicts

- Linear addressing stride == 8



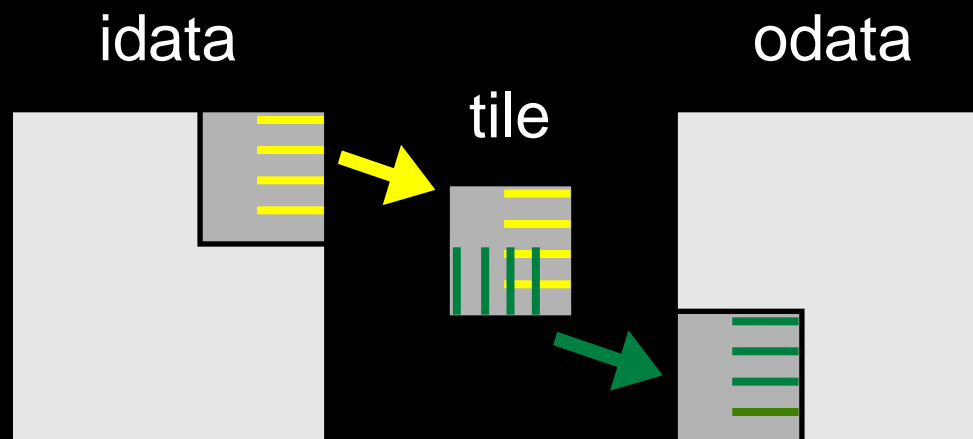
# Shared memory bank conflicts



- 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**

# 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



# Effective Bandwidth



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

# 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

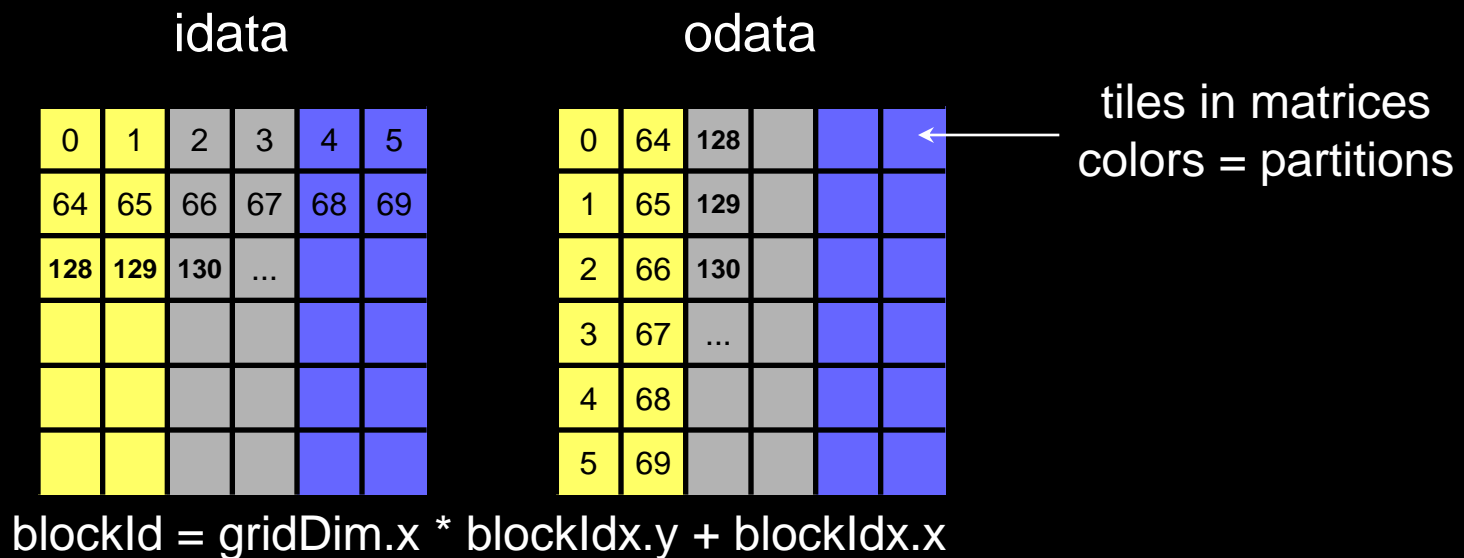


- **Global memory accesses go through partitions**
  - 6 partitions on 8-series GPUs, 8 partitions on 10-series GPUs
  - Successive 256-byte regions of global memory are assigned to successive partitions
- **For best performance:**
  - Simultaneous global memory accesses GPU-wide should be distributed 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

# 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



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

$$\text{blockId} = \text{gridDim.x} * \text{blockIdx.y} + \text{blockIdx.x}$$

# Diagonal Transpose

```
__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

# Diagonal Transpose

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

```
if (width == height) {  
    blockDim_y = blockDim.x;  
    blockDim_x = (blockDim.x+blockDim.y)%gridDim.x;  
} else {  
    int bid = blockDim.x + gridDim.x*blockDim.y;  
    blockDim_y = bid%gridDim.y;  
    blockDim_x = ((bid/gridDim.y)+blockDim_y)%gridDim.x;  
}
```

# Effective Bandwidth



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

# Outline

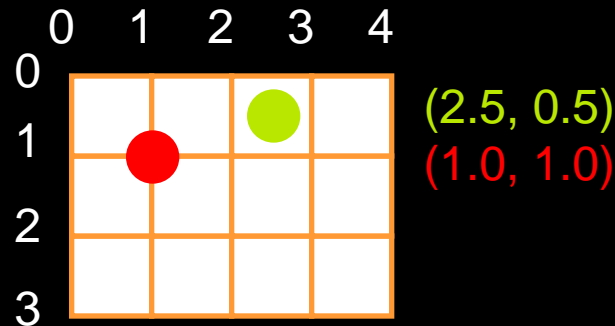
- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Matrix transpose study
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

# Textures in CUDA



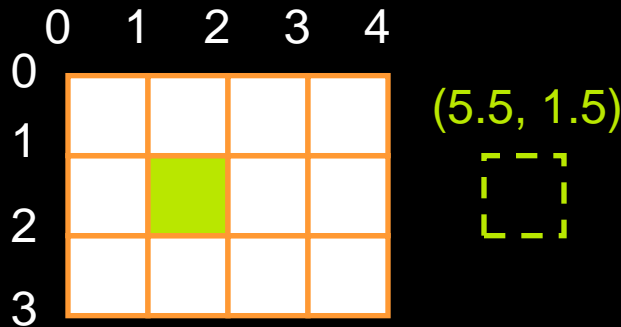
- Texture is an object for reading data
- Benefits:
  - Data is cached (optimized for 2D locality)
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable 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

# Texture Addressing



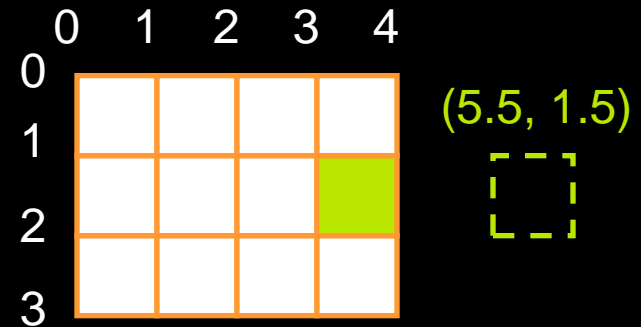
## Wrap

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



## Clamp

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



# Two CUDA Texture Types

- **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

# 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()`

# 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**

# 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

# Register Dependency



- **Read-after-write register dependency**

- Instruction's result can be read ~11 cycles later

- Scenarios:

CUDA:

```
x = y + 5;
```

```
z = x + 3;
```

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

PTX:

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

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

```
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

# 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

# Occupancy Calculator



Microsoft Excel - CUDA\_Occupancy\_calculator.xls

File Edit View Insert Format Tools Data Window Help

MyRegCount 20

## CUDA GPU Occupancy Calculator

click Here for detailed instructions on how to use this occupancy calculator

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)

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.

1.) Select a GPU from the list (click): **G80** (Help)

2.) Enter your resource usage:

Threads Per Block	192
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
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

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

Calculator / Help / GPU Data / Copyright & License

Ready

### Varying Block Size

Threads Per Block	Multiprocessor Warp Occupancy
16	8
32	12
48	12
64	12
80	12
96	12
112	12
128	12
144	12
160	12
176	12
192	12
208	12
224	12
240	12
256	12
272	12
288	12
304	12
320	12
336	12
352	12
368	12
384	12
400	12
416	12
432	12
448	12
464	12
480	12
496	12
512	12
528	12
544	12
560	12
576	12
592	12
608	12
624	12
640	12
656	12
672	12
688	12
704	12
720	12
736	12
752	12
768	12
784	12
800	12
816	12
832	12
848	12
864	12
880	12
896	12
912	12
928	12
944	12
960	12
976	12
992	12
1008	12
1024	12
1040	12
1056	12
1072	12
1088	12
1104	12
1120	12
1136	12
1152	12
1168	12
1184	12
1200	12
1216	12
1232	12
1248	12
1264	12
1280	12
1296	12
1312	12
1328	12
1344	12
1360	12
1376	12
1392	12
1408	12
1424	12
1440	12
1456	12
1472	12
1488	12
1504	12
1520	12
1536	12
1552	12
1568	12
1584	12
1600	12
1616	12
1632	12
1648	12
1664	12
1680	12
1696	12
1712	12
1728	12
1744	12
1760	12
1776	12
1792	12
1808	12
1824	12
1840	12
1856	12
1872	12
1888	12
1904	12
1920	12
1936	12
1952	12
1968	12
1984	12
2000	12
2016	12
2032	12
2048	12
2064	12
2080	12
2096	12
2112	12
2128	12
2144	12
2160	12
2176	12
2192	12
2208	12
2224	12
2240	12
2256	12
2272	12
2288	12
2304	12
2320	12
2336	12
2352	12
2368	12
2384	12
2400	12

### Varying Register Count

Registers Per Thread	Multiprocessor Warp Occupancy
0	24
1	24
2	24
3	24
4	24
5	24
6	24
7	24
8	24
9	24
10	24
11	24
12	24
13	24
14	24
15	24
16	24
17	24
18	24
19	24
20	24
21	18
22	18
23	18
24	18
25	18
26	18
27	18
28	18
29	18
30	18
31	18
32	18
33	12
34	12
35	12
36	12
37	12
38	12
39	12
40	12
41	12
42	12
43	12
44	12
45	12
46	12
47	12
48	12
49	12
50	12
51	12
52	12
53	12
54	12
55	12
56	12
57	12
58	12
59	12
60	12
61	12
62	12
63	12
64	12
65	12
66	12
67	12
68	12
69	12
70	12
71	12
72	12
73	12
74	12
75	12
76	12
77	12
78	12
79	12
80	12
81	12
82	12
83	12
84	12
85	12
86	12
87	12
88	12
89	12
90	12
91	12
92	12
93	12
94	12
95	12
96	12
97	12
98	12
99	12
100	12
101	12
102	12
103	12
104	12
105	12
106	12
107	12
108	12
109	12
110	12
111	12
112	12
113	12
114	12
115	12
116	12
117	12
118	12
119	12
120	12
121	12
122	12
123	12
124	12
125	12
126	12
127	12
128	12
129	12
130	12
131	12
132	12
133	12
134	12
135	12
136	12
137	12
138	12
139	12
140	12
141	12
142	12
143	12
144	12
145	12
146	12
147	12
148	12
149	12
150	12
151	12
152	12
153	12
154	12
155	12
156	12
157	12
158	12
159	12
160	12
161	12
162	12
163	12
164	12
165	12
166	12
167	12
168	12
169	12
170	12
171	12
172	12
173	12
174	12
175	12
176	12
177	12
178	12
179	12
180	12
181	12
182	12
183	12
184	12
185	12
186	12
187	12
188	12
189	12
190	12
191	12
192	12
193	12
194	12
195	12
196	12
197	12
198	12
199	12
200	12
201	12
202	12
203	12
204	12
205	12
206	12
207	12
208	12
209	12
210	12
211	12
212	12
213	12
214	12
215	12
216	12
217	12
218	12
219	12
220	12
221	12
222	12
223	12
224	12
225	12
226	12
227	12
228	12
229	12
230	12
231	12
232	12
233	12
234	12
235	12
236	12
237	12
238	12
239	12
240	12
241	12
242	12
243	12
244	12
245	12
246	12
247	12
248	12
249	12
250	12
251	12
252	12
253	12
254	12
255	12
256	12
257	12
258	12
259	12
260	12
261	12
262	12
263	12
264	12
265	12
266	12
267	12
268	12
269	12
270	12
271	12
272	12
273	12
274	12
275	12
276	12
277	12
278	12
279	12
280	12
281	12
282	12
283	12
284	12
285	12
286	12
287	12
288	12
289	12
290	12
291	12
292	12
293	12
294	12
295	12
296	12
297	12
298	12
299	12
300	12
301	12
302	12
303	12
304	12
305	12
306	12
307	12
308	12
309	12
310	12
311	12
312	12
313	12
314	12
315	12
316	12
317	12
318	12
319	12
320	12
321	12
322	12
323	12
324	12
325	12
326	12
327	12
328	12
329	12
330	12
331	12
332	12
333	12
334	12
335	12
336	12
337	12
338	12
339	12
340	12
341	12
342	12
343	12
344	12
345	12
346	12
347	12
348	12
349	12
350	12
351	12
352	12
353	12
354	12
355	12
356	12
357	12
358	12
359	12
360	12
361	12
362	12
363	12
364	12
365	12
366	12
367	12
368	12
369	12
370	12
371	12
372	12
373	12
374	12
375	12
376	12
377	12
378	12
379	12
380	12
381	12
382	12
383	12
384	12
385	12
386	12
387	12
388	12
389	12
390	12
391	12
392	12
393	12
394	12
395	12
396	12
397	12
398	12
399	12
400	12

### Varying Shared Memory Usage

# 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!

# 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

# 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

# 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

# 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

# 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

- 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!

# 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

# 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