

GPU Computing with CUDA

Part 3: CUDA Performance Tips and Tricks

Dortmund, June 4, 2009

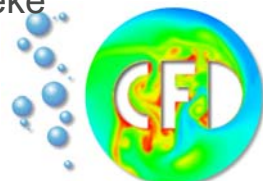
SFB 708, AK "Modellierung und Simulation"

Dominik Göddeke

Angewandte Mathematik und Numerik

TU Dortmund

dominik.goeddeke@math.tu-dortmund.de // <http://www.mathematik.tu-dortmund.de/~goeddeke>



- Slides based on previous courses by
 - Mark Harris, Simon Green, Gregory Ruetsch (NVIDIA)
 - Robert Strzodka (MPI Informatik)
 - Dominik Göddeke (TU Dortmund)
- ARCS 2008 GPGPU and CUDA Tutorials
<http://www.mathematik.tu-dortmund.de/~goeddeke/arcs2008/>
- University of New South Wales Workshop on GPU Computing with CUDA
<http://www.cse.unsw.edu.au/~pls/cuda-workshop09/>

- Overview
- Hardware
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Summary

- Maximize independent parallelism
- Maximize arithmetic intensity
 - Math per 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

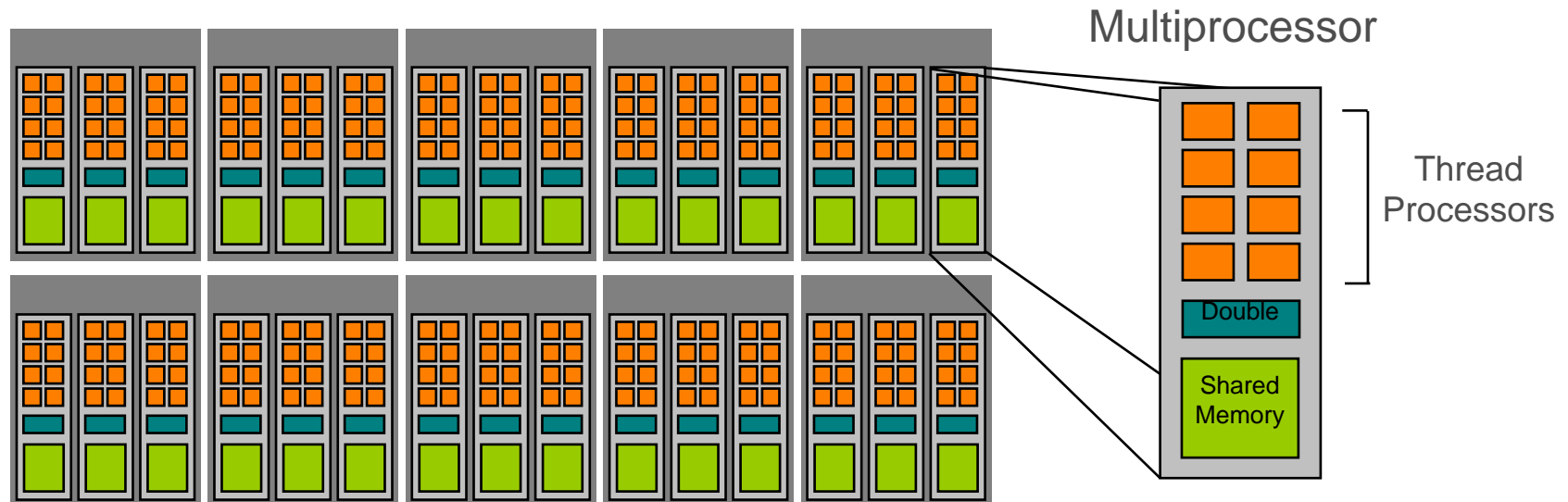
- Coalesced vs. non-coalesced = order of magnitude
 - Global / local device memory
- Optimize for spatial locality in cached texture memory
- In shared memory, avoid high-degree bank conflicts
- Partition camping
 - When global memory access not evenly distributed among partitions
 - Problem-size dependent

- Hundreds of times faster than global memory
 - Sometimes as fast as registers
- Threads can cooperate via shared memory
 - Per thread block
- Use one (a few) threads to load or 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

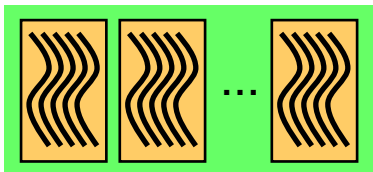
- Partition computation to keep the multiprocessors equally busy
 - Many threads, many thread blocks
 - Scalability on future devices
- Keep resource usage low enough to support multiple active threads blocks per multiprocessor
 - Registers, shared memory
 - Occupancy

- Overview
- **Hardware**
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Summary

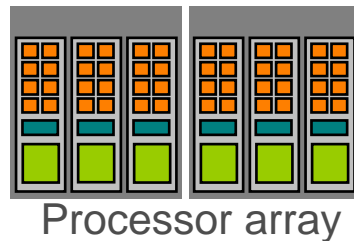
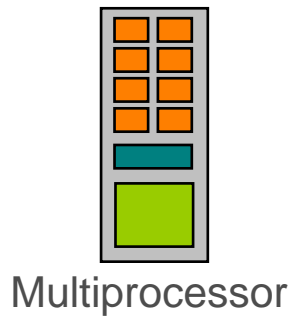
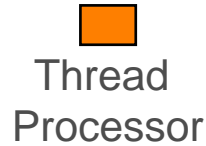
- 240 thread processors execute kernel threads
- 30 multiprocessors, each contains
 - 8 (single precision and integer) thread processors
 - 1 double precision unit
 - Shared memory enables thread cooperation



Software



Hardware



Threads are executed by thread processors

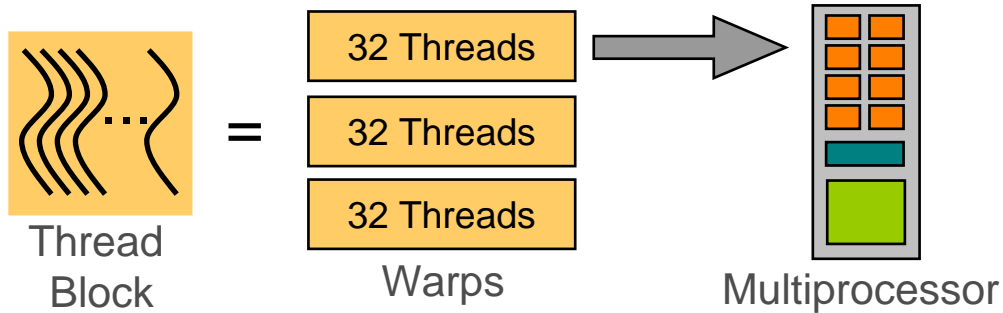
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

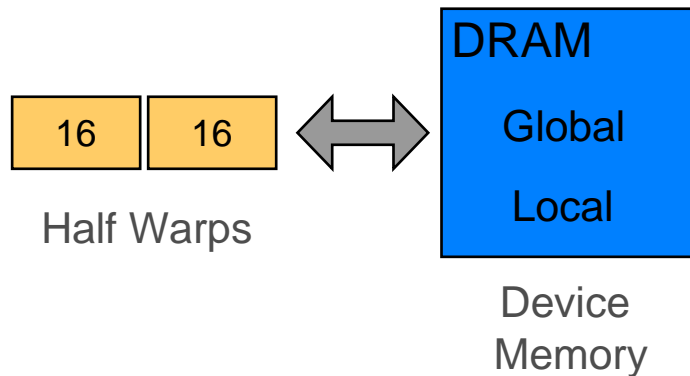
A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

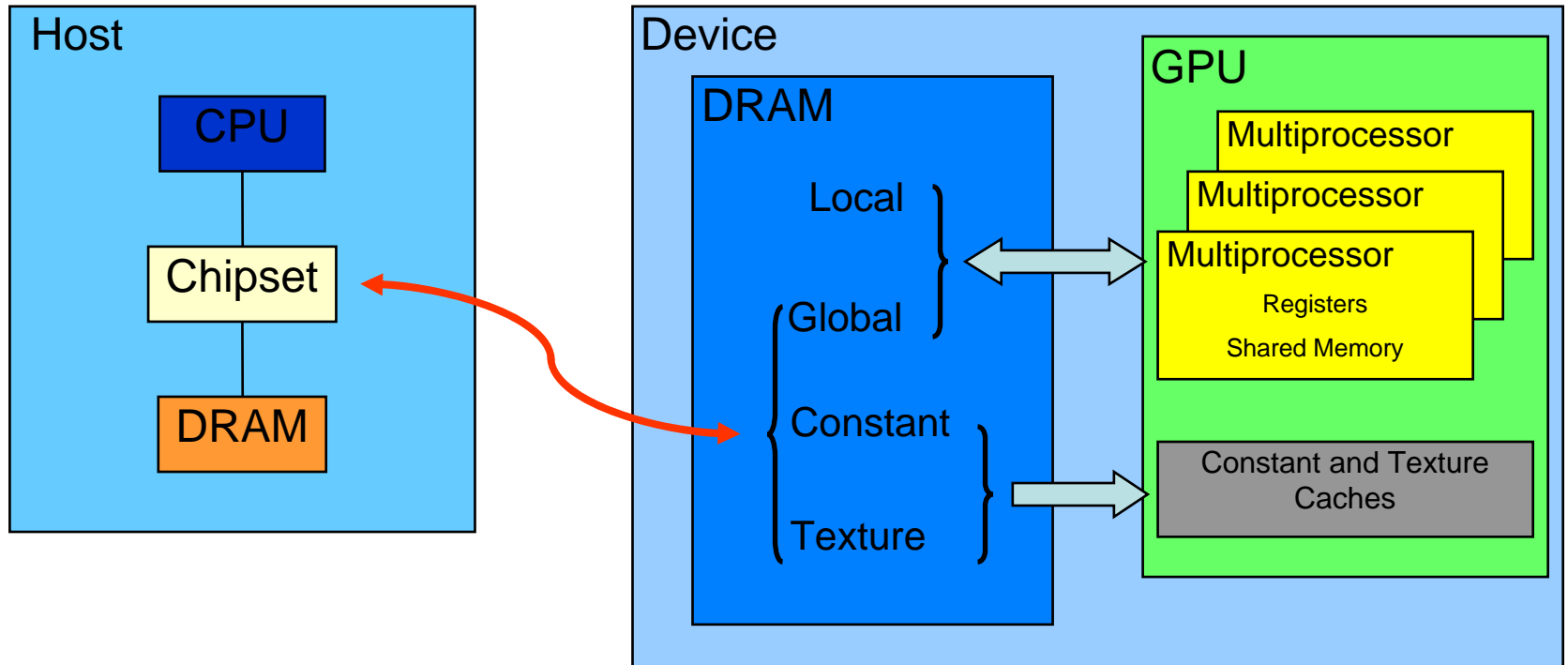


A thread block consists of 32-thread warps

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 called coalescing




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

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

- **Device to host bandwidth much lower than device to device bandwidth**
 - 8 GB/s peak (PCIe x16 Gen 2) vs. 160 GB/s (GTX 285)
- **Minimize transfers**
 - Intermediate data can be allocated, operated on, and deallocated without even copying them to host memory
- **Group transfers**
 - One large transfer is better than many small ones

- `cudaMallocHost()` allows allocation of page-locked („pinned“) host memory
- Enables highest `cudaMemcpy()` performance
 - 3.2 GB/s on PCIe x16 Gen 1
 - 5.2 GB/s on PCIe x16 Gen 2
- Use with caution!!
 - Allocating too much page-locked memory can reduce overall system performance and stability
 - Test systems and learn their limits
- Live demo
 - BandwidthTest CUDA SDK example

- Async and stream APIs allow overlap of H2D or D2H data transfer 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
 - If not used, everything happens in stream 0

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

overlapped
- Live demo
 - `streamTest`

- **Kernel based**
 - Implicit barrier between kernel invocations in the same stream
- **Context based**
 - `cudaThreadSynchronize();`
 - Blocks until all previously issued CUDA calls from a CPU thread complete
- **Stream based**
 - `cudaStreamSynchronize(streamID);`
 - Blocks until all CUDA calls issued to given stream complete
 - `cudaStreamQuery(streamID);`
 - Indicates whether stream is idle
 - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
 - Does not block CPU thread

- **Stream based using events**
 - Event = simple label created by `cudaEventCreate(&(cudaEvent_t e));`
 - Events can be inserted into streams
 - `cudaEventRecord(event, streamID);`
 - Event is recorded then GPU reaches it in a stream
 - Recorded = assigned a timestamp (GPU clocktick)
 - Useful for fine-granular timing
 - `cudaEventSynchronize(event);`
 - Blocks until given event is recorded
 - `cudaEventQuery(event);`
 - Indicates whether event has recorded
 - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
 - Does not block CPU thread

- 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

- 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


- Device bandwidth of GTX 280

- $1107 * 10^6 * (512 / 8) * 2 / 1024^3 = 131.9 \text{ GB/s}$

The diagram shows the calculation $1107 * 10^6 * (512 / 8) * 2 / 1024^3 = 131.9 \text{ GB/s}$. Brackets are drawn under the terms $1107 * 10^6$ and $(512 / 8)$. Below the first bracket is the text "Memory clock (Hz)". Below the second bracket is the text "Memory interface (bytes)". To the right of the equation, a downward arrow points to the text "DDR".

- Specs report 141 GB/s
 - Use 10^9 B/GB conversion rather than 1024^3
 - Whichever you use, be consistent

- Transpose effective bandwidth

- $2048^2 * 4 \text{ B/element} / 1024^3 * 2 / (\text{time in secs}) = \text{GB/s}$


Matrix size
(bytes)

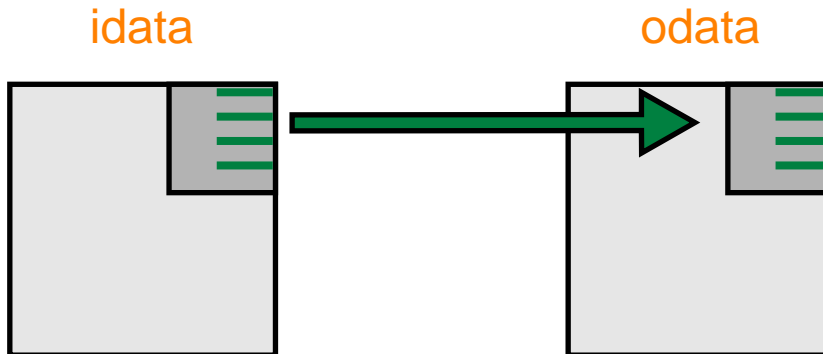
Read and
write

- Reference case - matrix copy

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


```
__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

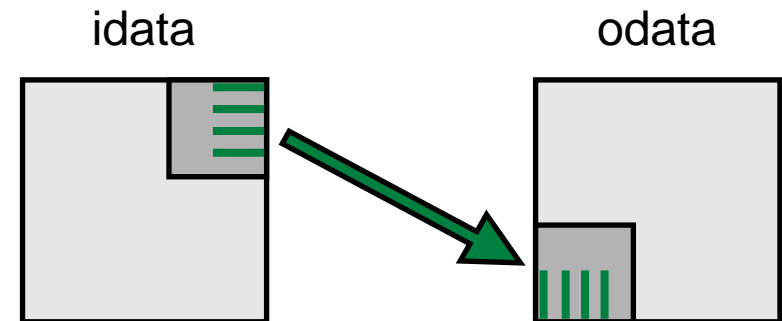
idata and odata
in global memory

- 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];  
        }  
    }  
}
```

- 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 (GB/s)
2048x2048, GTX 280

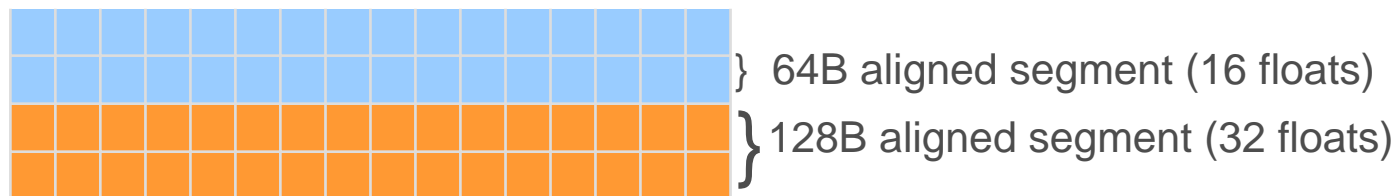
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2

- 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

- 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

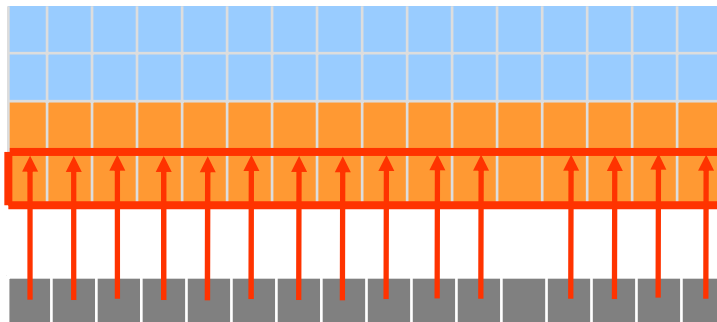
Examples – float (32-bit) data

Global Memory



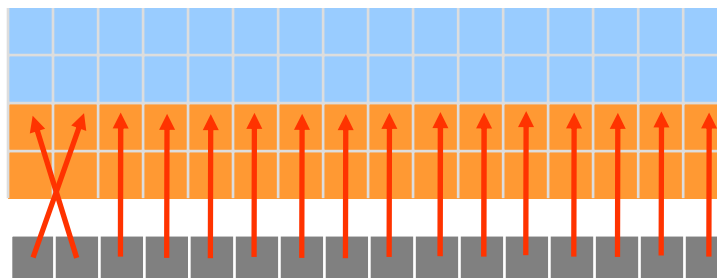
Half-warp of threads

- Compute capability 1.0 and 1.1
 - K-th thread must access k-th word in the segment (or k-th word in two contiguous 128B segments for 128-bit words)
 - Not all threads need to participate

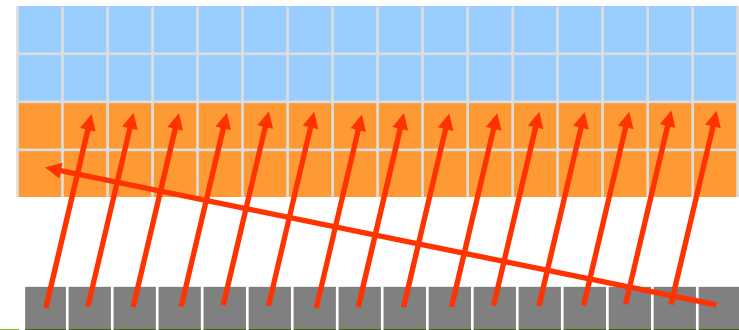


Coalesces – 1 transaction

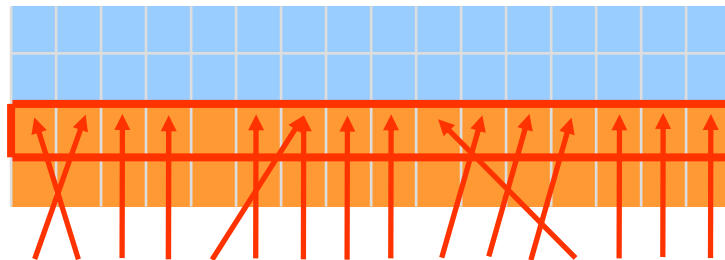
Out of sequence – 16 transactions



Misaligned – 16 transactions

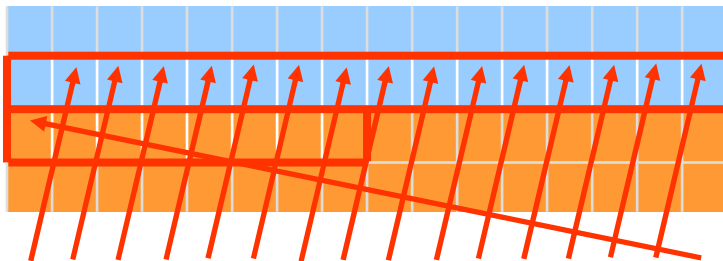


- Compute capability 1.2 and higher
 - Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words
 - Smaller transactions may be issued to avoid wasted bandwidth due to unused words

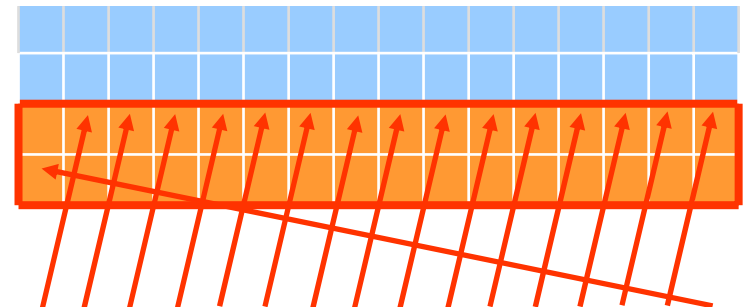


1 transaction - 64B segment

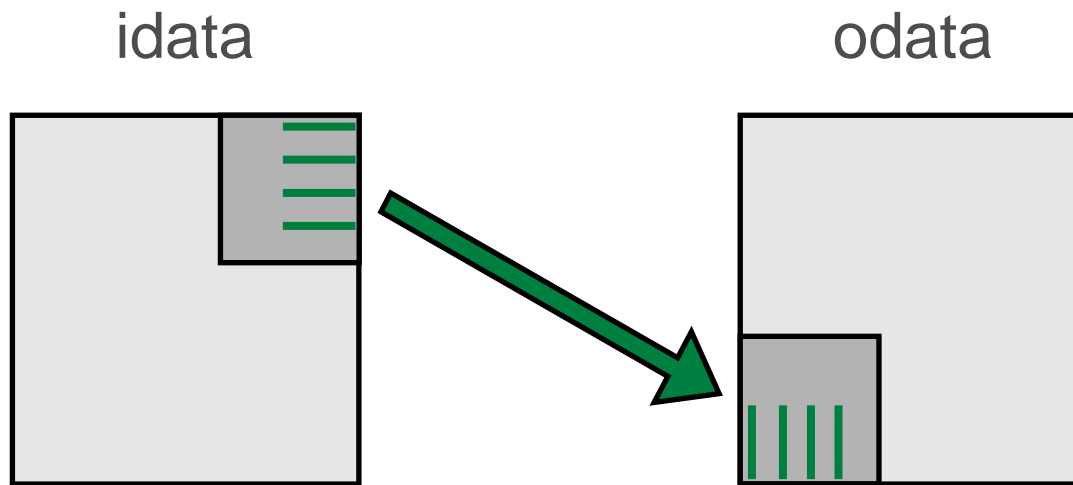
2 transactions - 64B and 32B segments



1 transaction - 128B segment



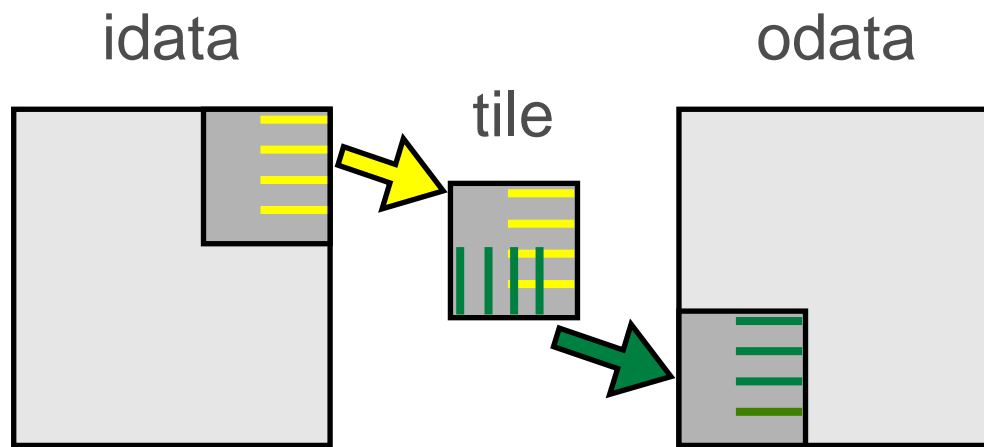
- Naïve transpose coalesces reads, but not writes



Elements transposed by a half-warp of threads

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Use one (a few) threads to load or 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

- 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

```
__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();

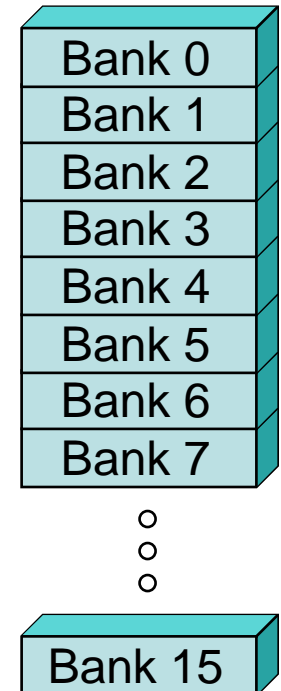
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
        }
    }
}
```

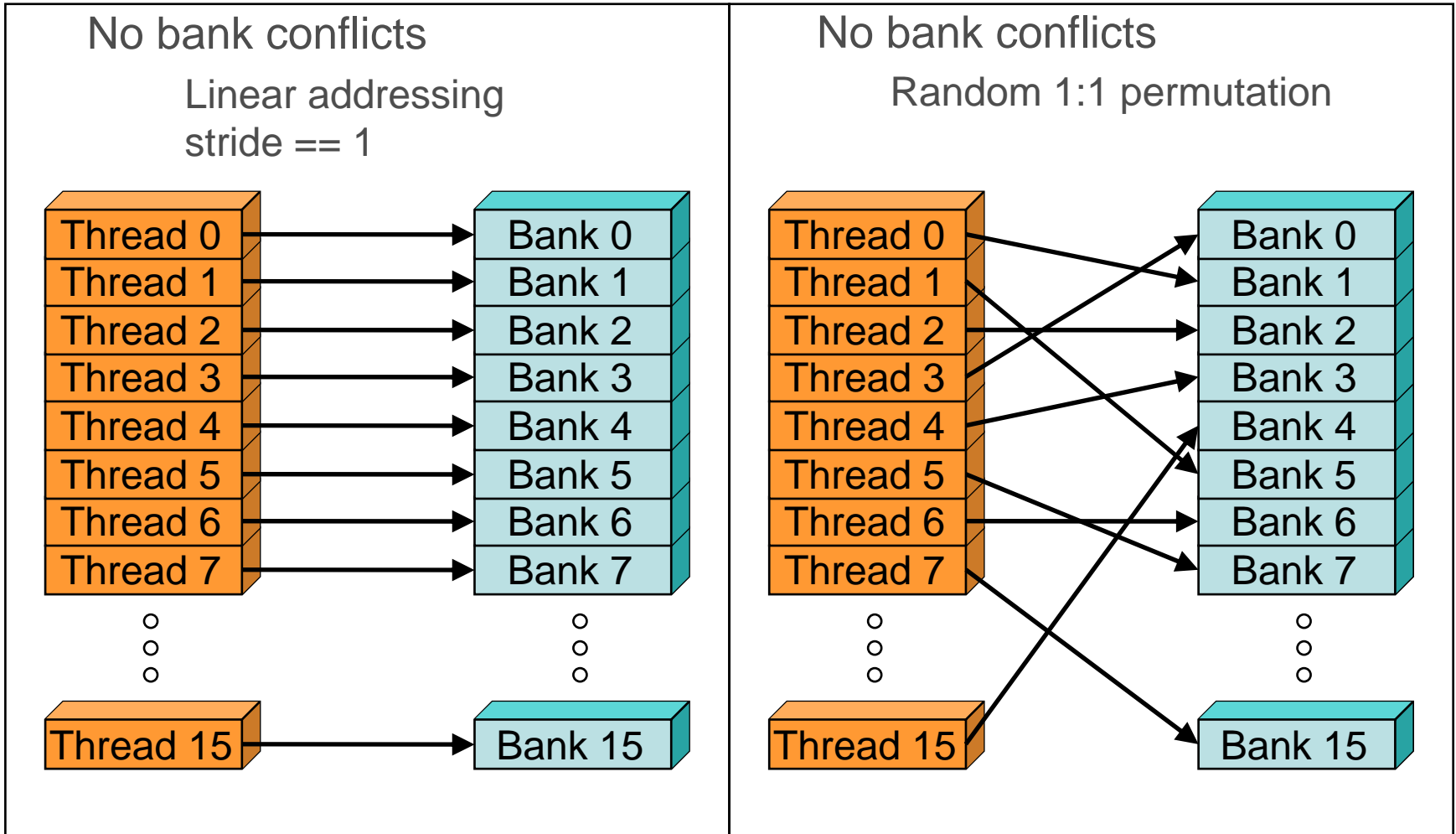
Effective Bandwidth (GB/s)
2048x2048, GTX 280

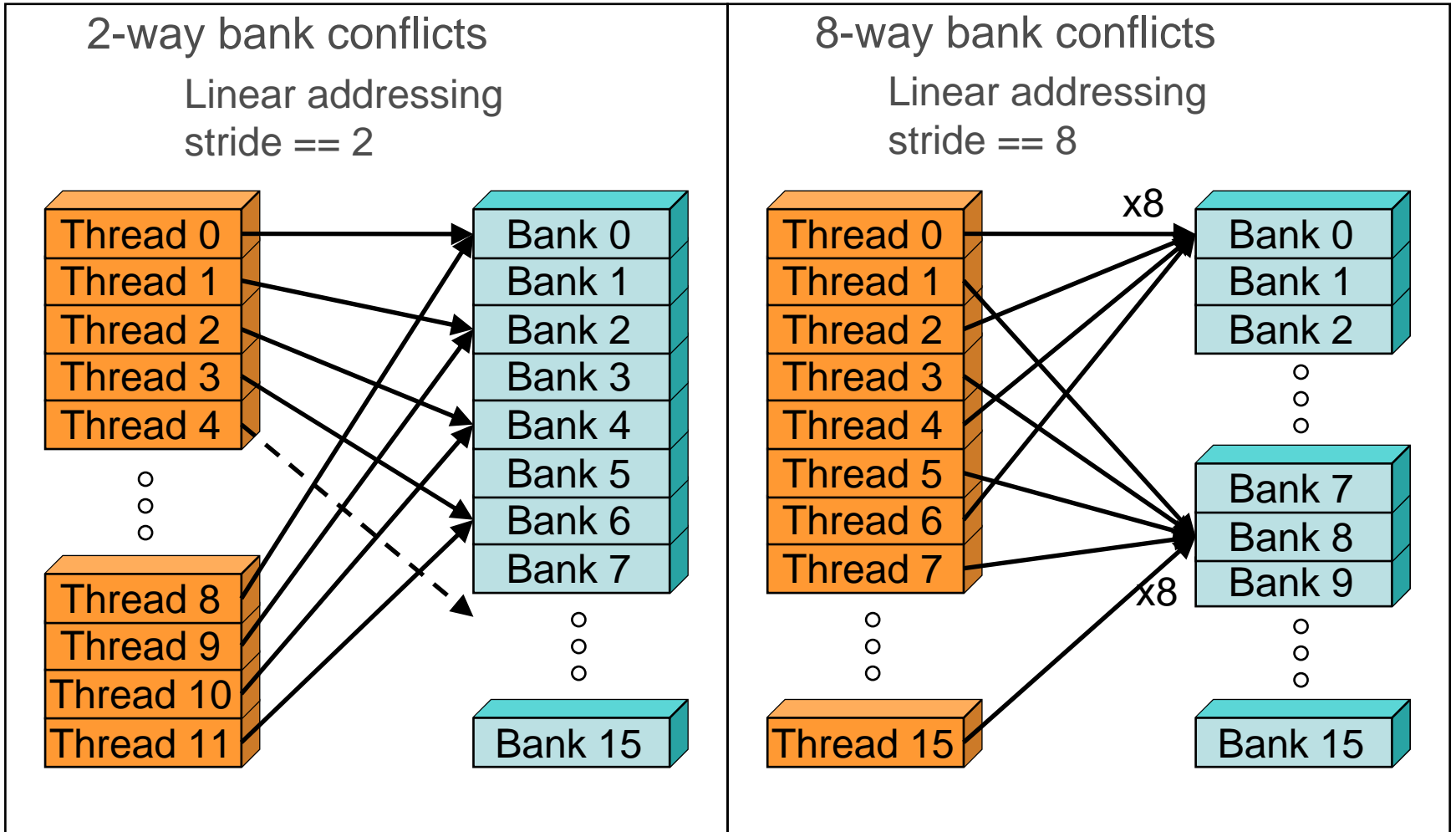
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

- 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

- Many threads accessing memory
 - Therefore, memory is divided in banks
 - Successive 32-bit words assigned to successive banks
- Each bank can serve 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 addresses are serialized

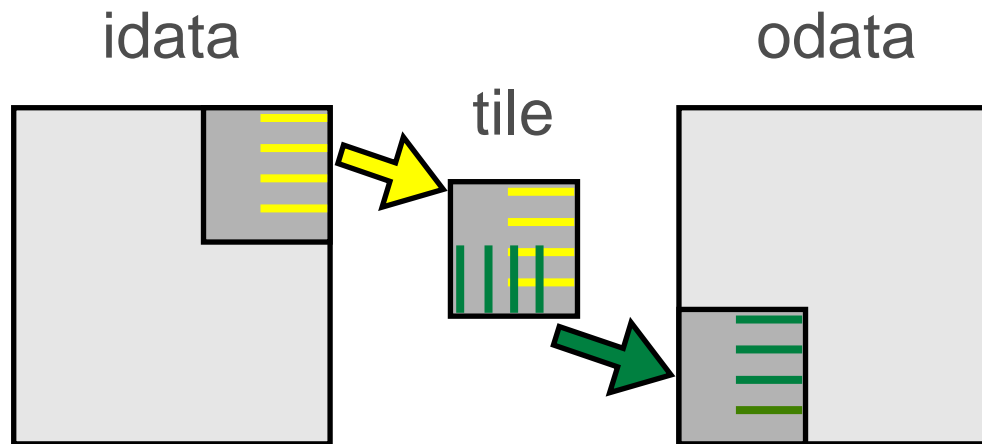






- **Shared memory is ~ as fast as registers**
 - If there are no bank conflicts
 - warp_serialize profiler signal
- **The fast case**
 - If all threads of a half-warp access different banks, there are no bank conflicts
 - If all threads of a half-warp read the identical address, there is no bank conflict (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

- 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



Elements transposed by a half-warp of threads

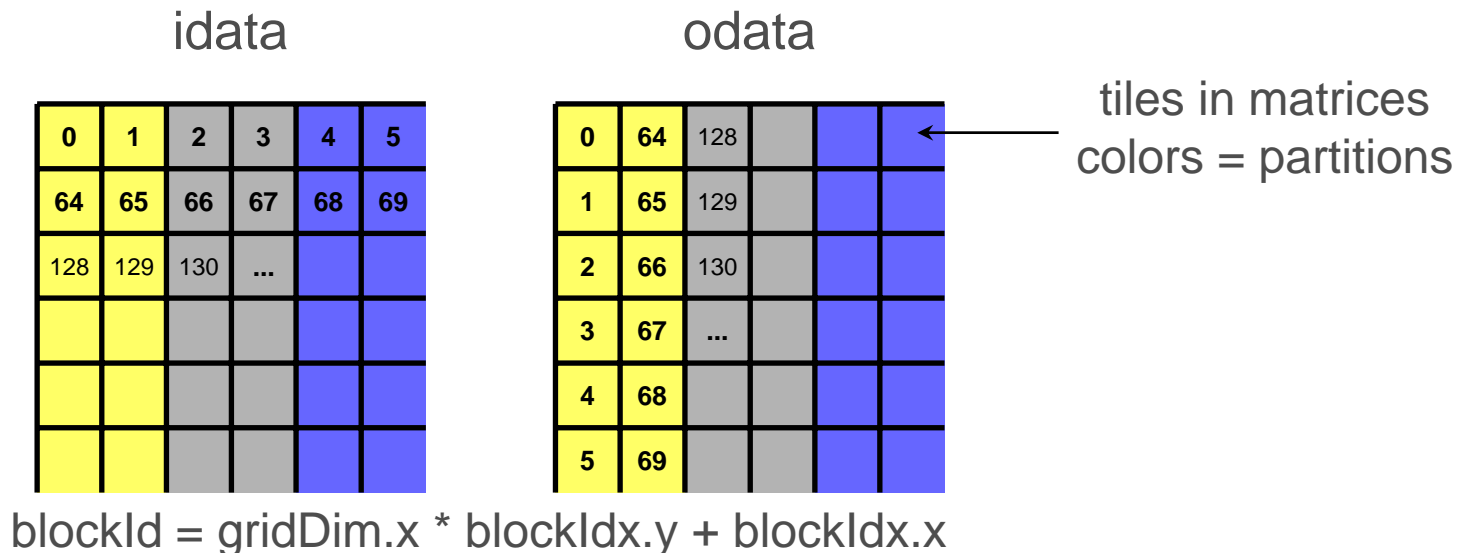
Effective Bandwidth (GB/s)
2048x2048, GTX 280

	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

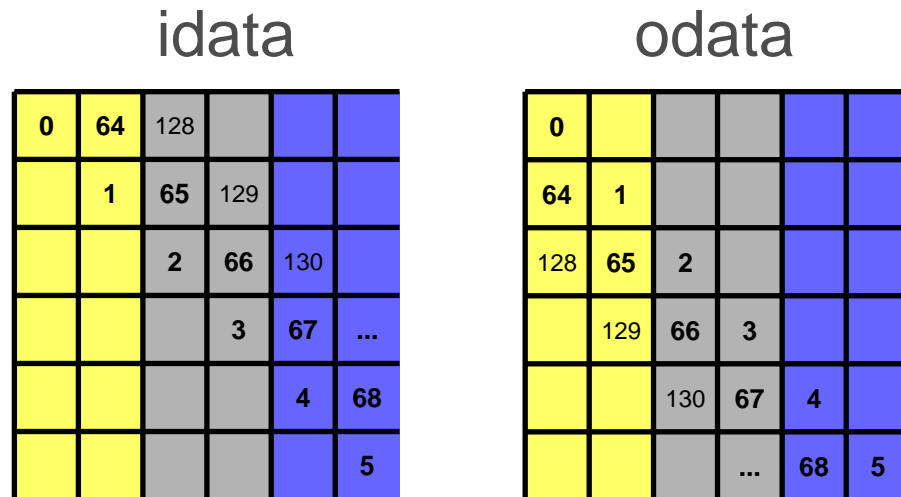
- Overview
- Hardware
- Memory optimizations
 - Data transfers between host and device
 - Device memory optimizations
 - Matrix transpose study
 - Measuring performance - effective bandwidth
 - Coalescing
 - Shared memory bank conflicts
 - Partition camping
- Execution configuration optimizations
- Instruction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

- **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 width = 256 bytes = 64 floats
 - Twice size of tile
- On GTX 280 (8 partitions), data 2K apart map to same partition
 - 2048 floats divides evenly by 2kB => columns of matrices map to same partition



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



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


```
__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;
```

Add lines to map diagonal
to Cartesian coordinates

```
    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;
```

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

```
    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];  
        }  
    }  
}
```

- Previous slide for square matrices
- More generally

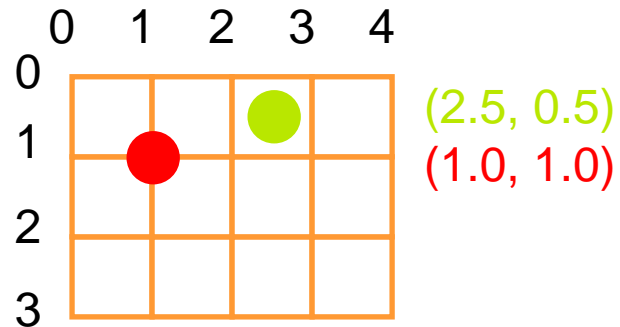
```
if (width == height) {
    blockIdx_y = blockIdx.x;
    blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
} else {
    int bid = blockIdx.x + gridDim.x*blockIdx.y;
    blockIdx_y = bid%gridDim.y;
    blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
}
```

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

- 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

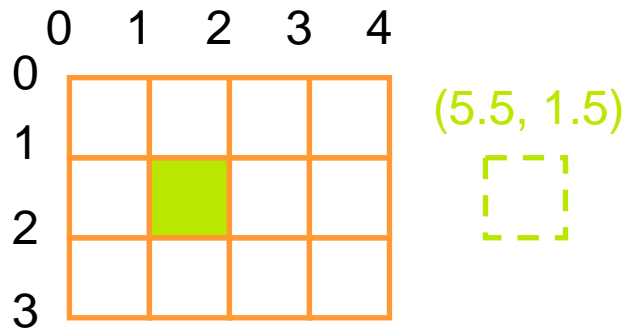
- Overview
- Hardware
- Memory optimizations
 - Data transfers between host and device
 - Device memory optimizations
 - Matrix transpose study
 - Textures
- Execution configuration optimizations
- Instruction optimizations
- Summary



Read-only access!

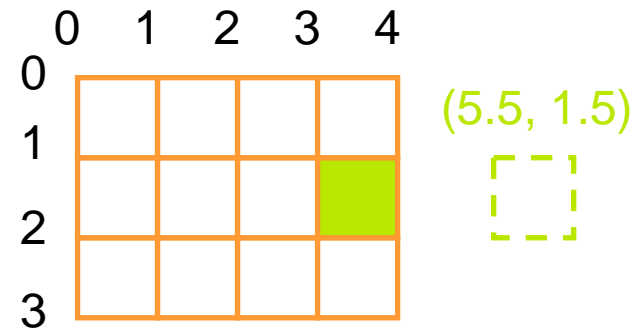
Wrap

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



Clamp

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



- **Bound to linear memory**
 - Standard way to get cached access to 1D arrays
 - Global memory address is bound to a texture
 - Only 1D
 - Integer addressing
 - No filtering, no addressing modes
- **Bound to CUDA arrays**
 - Full graphics functionality
 - CUDA array is bound to a texture
 - 1D, 2D, 3D
 - Float addressing (size-based or normalized)
 - Filtering
 - Address modes (clamping, repeat)

- **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
 - Compute
 - Unbind the texture reference, free resources
- **Device (kernel) code**
 - Fetch using texture reference
 - Linear memory textures
 - `tex1Dfetch()`
 - Array textures
 - `tex1D()` or `tex2D()` or `tex3D()`

- Overview
- Hardware
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Summary

- Thread instructions are executed sequentially (in order)
 - 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

- # 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()` barrier 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

- Hide latencies by using more threads per multiprocessor
- Limiting factors
 - Number of registers per kernel
 - 16K (8K on G8x) per SM, partitioned among concurrent threads
 - Amount of shared memory
 - 16kB per SM, partitioned among concurrent thread blocks
- Compile with `-ptxas-options=v` flag
 - Verbose mode, study carefully
- Use `-maxregcount=N` flag
 - N = desired maximum registers per kernel
 - At some point spilling into local memory may occur
 - Reduces performance, local memory is slow (implemented in global memory)

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)

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
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	3640
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

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

Varying Register Count

Varying Shared Memory Usage

- Choose threads per block as a multiple of warp size
 - Avoid wasting computation on under-populated warps (SIMD)
- More threads per block = better memory latency hiding
 - But: 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 registers to compile and invoke successfully
 - This all depends on your computation, so experiment

- Increasing occupancy does not necessarily increase performance

BUT ...

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

- 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 or ATLAS
 - Experiment mode discovers and saves optimal configuration
 - Recall transpose example

- Overview
- Hardware
- Memory optimizations
- Execution configuration optimizations
- **Instruction optimizations**
- Summary

- **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 GTX 280

- **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 hardware computations**
 - High arithmetic intensity programs
 - High ratio of math to memory transactions
 - Many concurrent threads

- **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 per warp
 - Use `__mul24()`, `__umul24()` intrinsics for 4-cycle 24-bit int multiply
- **Integer divide and modulo are more expensive**
 - Compiler tries to convert literal power-of-two divides to shifts
 - Be explicit in cases where compiler can't tell that divisor is power of 2
 - Useful trick: `foo % n == foo & (n-1)` if `n` is a power of two

- Intrinsic reciprocal, reciprocal square root, sin/cos, log, exp prefixed with „__“
 - 16 cycles per warp
 - Example: __rcp()
- Other functions are combinations of the above
 - $y/x == \text{rcp}(x) * y$ takes 20 cycles per warp
 - $\text{Sqrt}(x) == x * \text{rsqrt}(x)$ takes 20 cycles per warp

- There are two types of runtime math operations
 - `__func()`: direct mapping to hardware ISA
 - Fast
 - But lower accuracy (see progguide)
 - Example: `__sin(x)`
 - `func()`: compiles to multiple instructions
 - Slower but higher accuracy (5 ULP or less)
 - Example: `sin(x)`
- `-use-fast-math` compiler flag
 - Forces every `func()` to compile to `__func()`
- Double precision always IEEE-754 compliant

- Many, many variables
 - Hardware, compiler, optimization flags...
- CPU operations aren't strictly limited to 0.5 ulp
 - Sequences of operations can be more accurate due to 80-bit extended precision ALUs
 - CPU-SSE code usually closest to GPU code
- Floating point arithmetic is not associative and commutative!

- **Symbolic**
 - $(x+y)+z = x+(y+z)$
- **Not necessarily true for floating-point addition**
 - Try $x=10^{30}$, $y = -10^{30}$ and $z=1$ in the above equation
- **Parallelizing computations**
 - Potentially changes the order of operations
 - Results may not exactly match sequential results
 - This is not specific to CUDA or GPU
 - Inherent part of parallel computation

- **Main performance concern with branching is divergence**
 - Overhead of simple branch: ~4 cycles per warp
 - Divergence: Threads within a single warp take different paths
 - Different execution paths must be serialized
- **Avoid divergence when branch condition is a function of the 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

- 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