



nVIDIA®

Optimizing CUDA

Outline



- **Overview**
- **Hardware**
- **Memory Optimizations**
- **Execution Configuration Optimizations**
- **Instruction Optimizations**
- **Multi-GPU**
- **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**

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

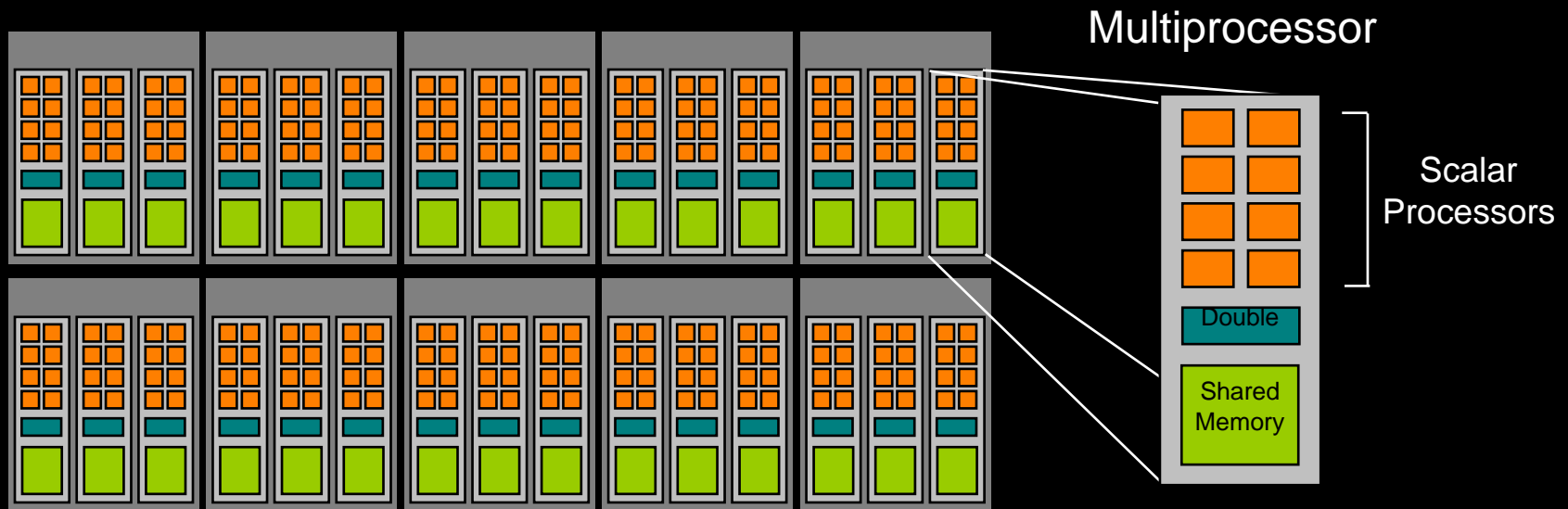


- Overview
- **Hardware**
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Summary

10-Series Architecture



- 240 **Scalar Processor (SP) cores** execute kernel threads
- 30 Streaming Multiprocessors (SMs) each contain
 - 8 scalar processors
 - 2 Special Function Units (SFUs)
 - 1 double precision unit
 - **Shared memory** enables thread cooperation



Execution Model



Software

Hardware



Threads are executed by scalar processors



Thread
Block

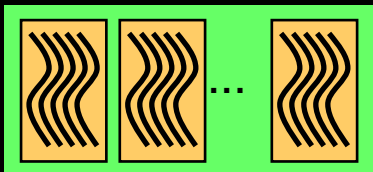


Multiprocessor

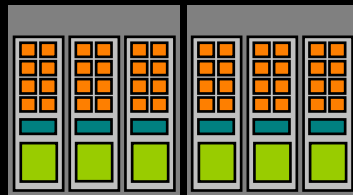
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

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



Grid

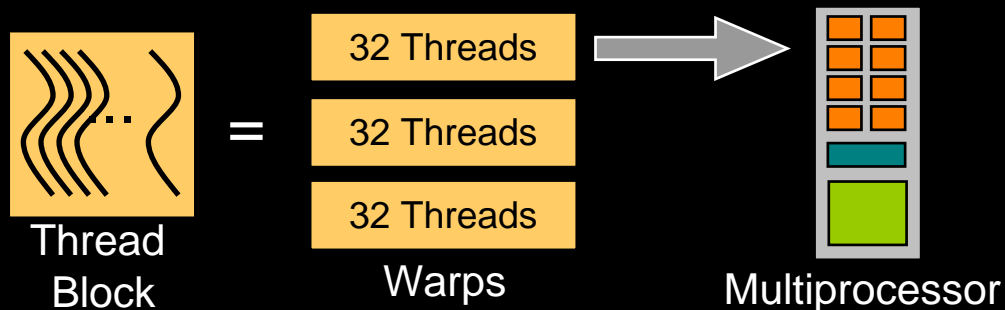


Device

A kernel is launched as a grid of thread blocks

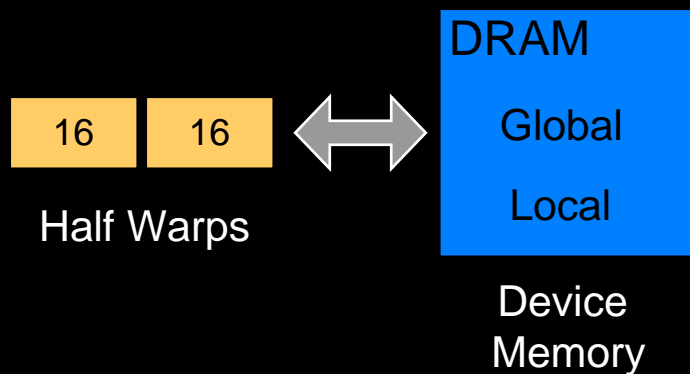
Only one kernel can execute on a device at one time

Warps and Half Warps



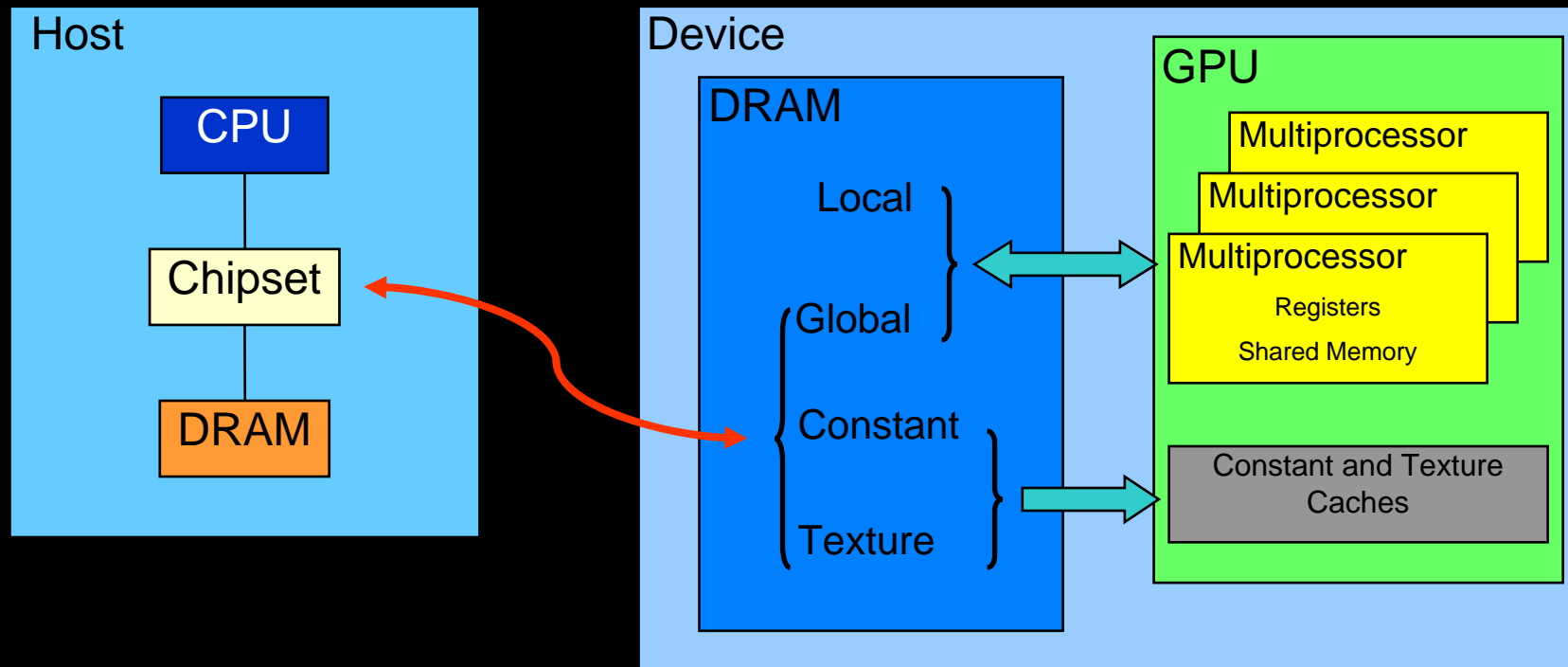
A thread block consists of 32-thread warps

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
 - Measuring performance – effective bandwidth
 - Coalescing
 - Shared Memory
 - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Summary

Host-Device Data Transfers



- **Device to host memory bandwidth much lower than device to device bandwidth**
 - 8 GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX 280)
- **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

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);  
kernel<<grid, block>>>(a_d);  
cpuFunction();
```

A diagram illustrating the overlap of data transfer and CPU computation. A large right-facing curly bracket groups the first three lines of code. A light blue arrow points from the right side of this bracket to the text "overlapped". Another light blue arrow points from the "overlapped" text to the `cpuFunction();` line, indicating that the CPU computation can begin while the data transfer is still in progress.

overlapped

Overlapping kernel and data transfer

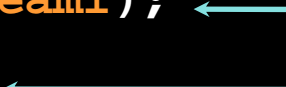


Requires:

- “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>>>(...);
```



overlapped

The diagram shows a blue line connecting the `stream1` argument in the `cudaMemcpyAsync` call to the `stream2` argument in the `kernel` call. A blue arrow points from the word "overlapped" to this connection line, indicating that the two operations can be executed concurrently.

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 when 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

Zero copy



- Access host memory directly from device code
 - Transfers implicitly performed as needed by device code
 - Introduced in CUDA 2.2
 - Check `canMapHostMemory` field of `cudaDeviceProp` variable
- All set-up is done on host using mapped memory

```
cudaSetDeviceFlags(cudaDeviceMapHost);  
...  
cudaHostAlloc((void **)&a_h, nBytes,  
              cudaHostAllocMapped);  
cudaHostGetDevicePointer((void **)&a_d, (void *)a_h, 0);  
for (i=0; i<N; i++) a_h[i] = i;  
increment<<grid, block>>>(a_d, N);
```

Zero copy considerations



- Zero copy will always be a win for integrated devices that utilize CPU memory (you can check this using the **integrated** field in **cudaDeviceProp**)
- Zero copy will be faster if data is only read/written from/to global memory once, for example:
 - Copy input data to GPU memory
 - Run one kernel
 - Copy output data back to CPU memory
- Potentially easier and faster alternative to using **cudaMemcpyAsync**
 - For example, can both read and write CPU memory from within one kernel
- Note that current devices use pointers that are 32-bit so there is a limit of **4GB per context**

Outline



- Overview
- Hardware
- **Memory Optimizations**
 - Data transfers between host and device
 - **Device memory optimizations**
 - **Measuring performance – effective bandwidth**
 - Coalescing
 - Shared Memory
 - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Summary

Theoretical Bandwidth



● Device Bandwidth of GTX 280

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

● Specs report 141 GB/s

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

Effective Bandwidth



● Effective Bandwidth (for copying array of N floats)

●
$$\underbrace{N * 4 \text{ B/element}}_{\text{Array size (bytes)}} / \underbrace{1024^3}_{\text{B/GB (or } 10^9)} * \underbrace{2}_{\text{Read and write}} / (\text{time in secs}) = \text{GB/s}$$

Outline

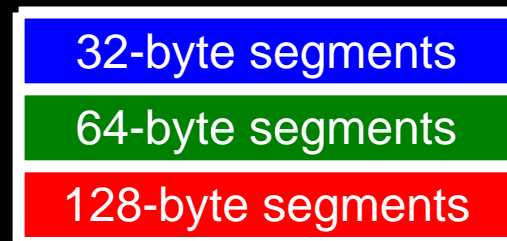


- Overview
- Hardware
- **Memory Optimizations**
 - Data transfers between host and device
 - **Device memory optimizations**
 - Measuring performance – effective bandwidth
 - **Coalescing**
 - Shared Memory
 - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Summary

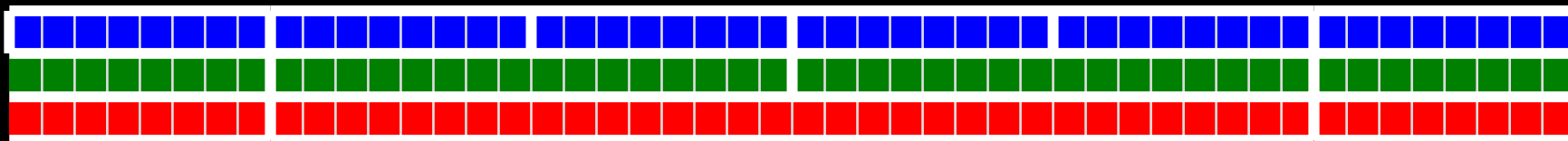
Coalescing



- 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
- Float (32-bit) data example:



Global Memory

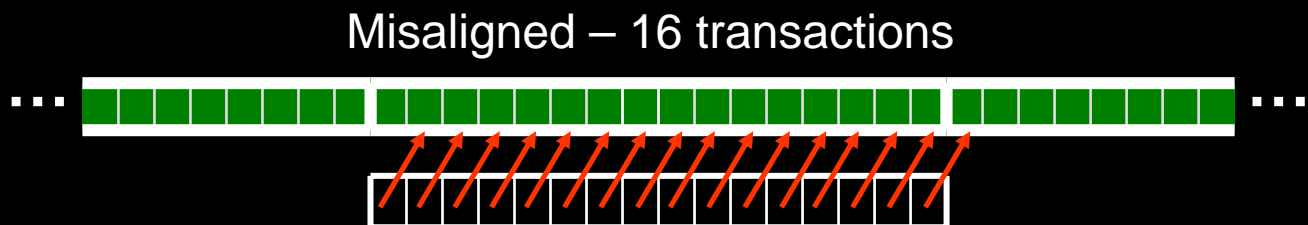
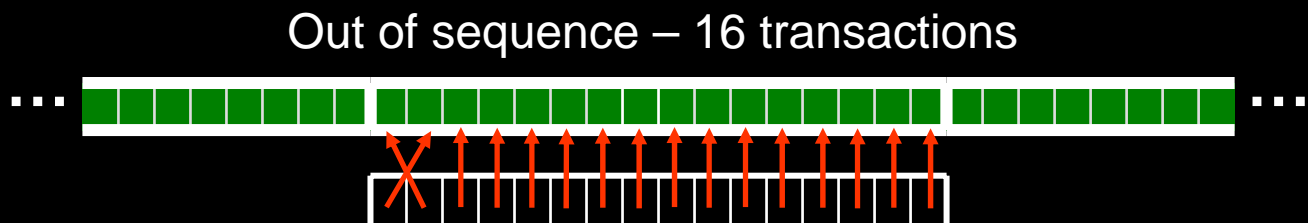
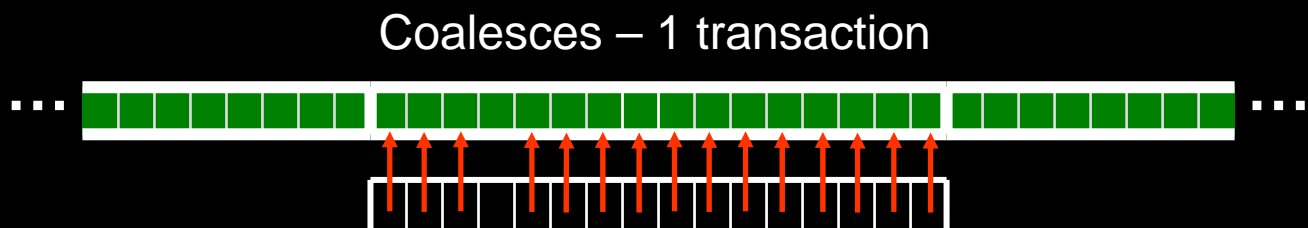


Half-warp of threads

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

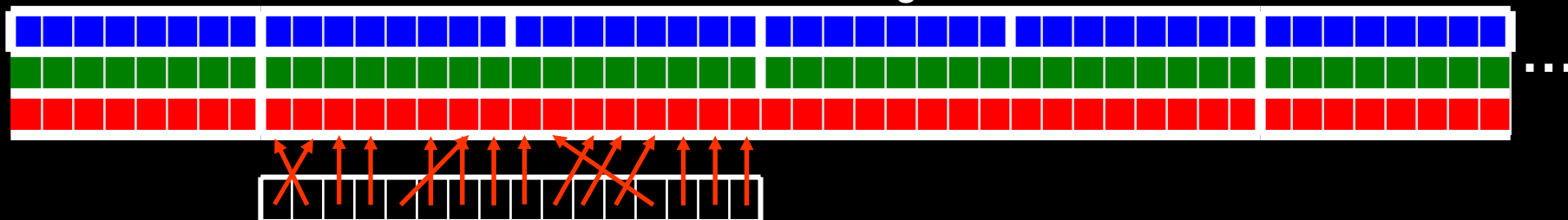


Coalescing

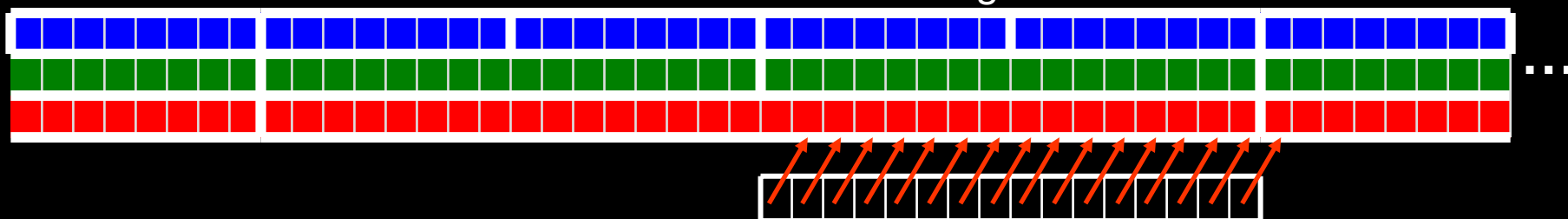
Compute capability 1.2 and higher

- Issues transactions for segments of 32B, 64B, and 128B
- Smaller transactions used to avoid wasted bandwidth

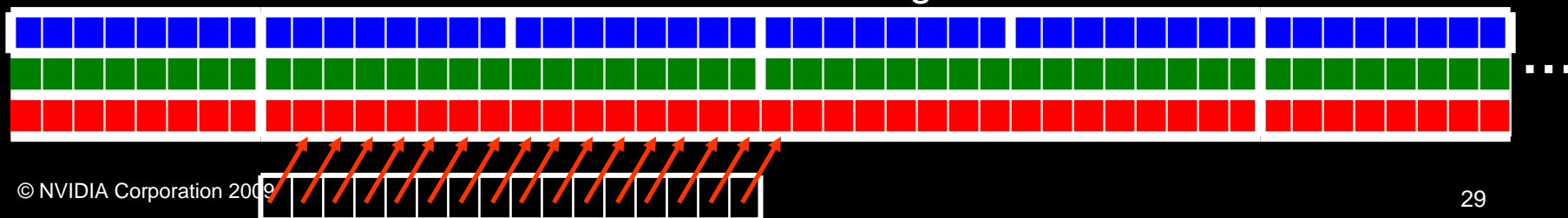
1 transaction - 64B segment



2 transactions - 64B and 32B segments



1 transaction - 128B segment



Coalescing Examples

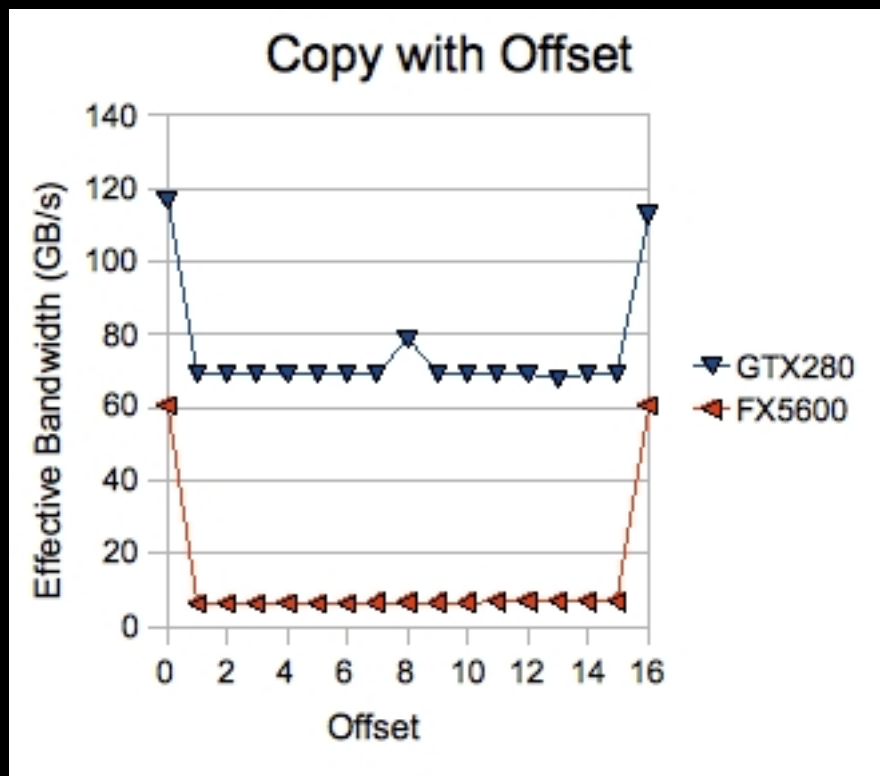


- **Effective bandwidth of small kernels that copy data**
 - **Effects of offset and stride on performance**
- **Two GPUs**
 - **GTX 280**
 - **Compute capability 1.3**
 - **Peak bandwidth of 141 GB/s**
 - **FX 5600**
 - **Compute capability 1.0**
 - **Peak bandwidth of 77 GB/s**

Coalescing Examples



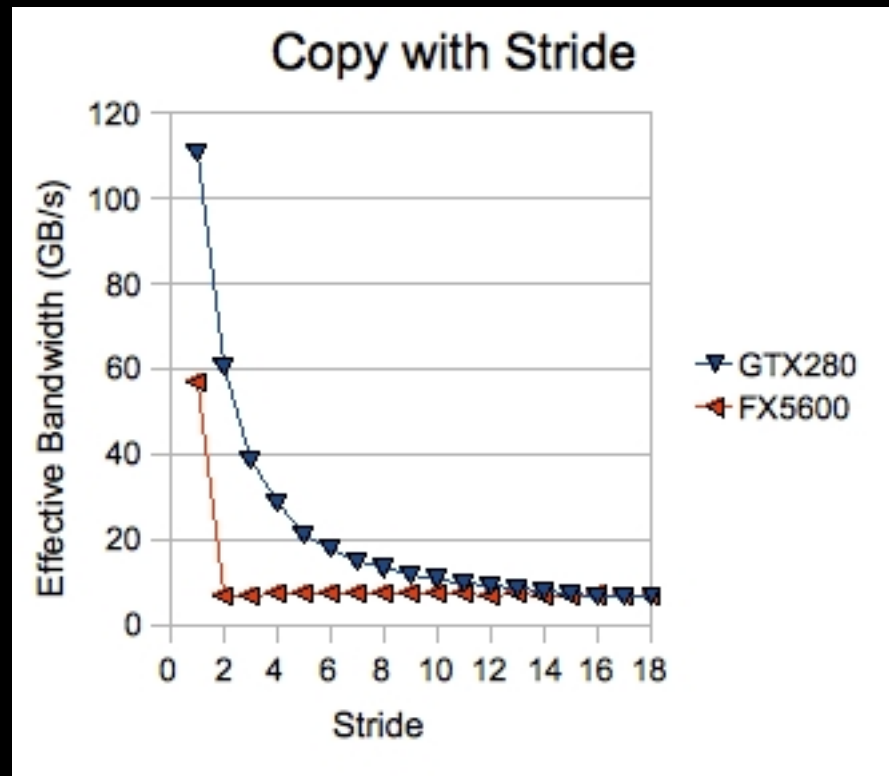
```
__global__ void offsetCopy(float *odata, float *idata,  
                           int offset)  
{  
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;  
    odata[xid] = idata[xid];  
}
```



Coalescing Examples



```
__global__ void strideCopy(float *odata, float *idata,  
                           int stride)  
{  
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;  
    odata[xid] = idata[xid];  
}
```



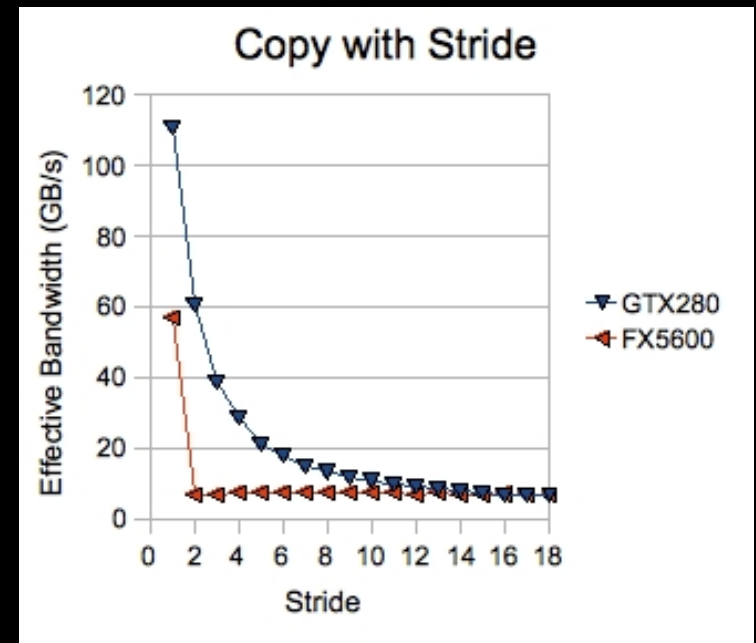
Coalescing Examples



- Strided memory access is inherent in many multidimensional problems
 - Stride is generally large ($\gg 18$)

However ...

- Strided access to global memory can be avoided using *shared memory*



Outline



- Overview
- Hardware
- **Memory Optimizations**
 - Data transfers between host and device
 - **Device memory optimizations**
 - Measuring performance – effective bandwidth
 - Coalescing
 - **Shared Memory**
 - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Summary

Shared Memory

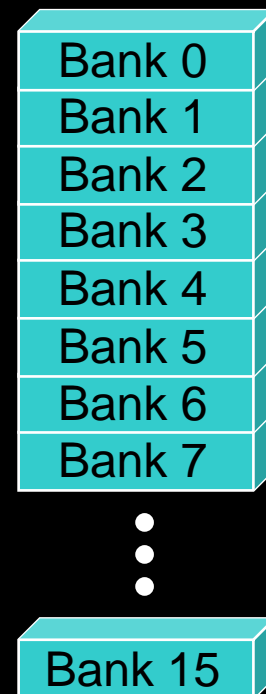


- **~Hundred times faster than global memory**
- **Cache data to reduce global memory accesses**
- **Threads can cooperate via shared memory**
- **Use it to avoid non-coalesced access**
 - **Stage loads and stores in shared memory to re-order non-coalesceable addressing**

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

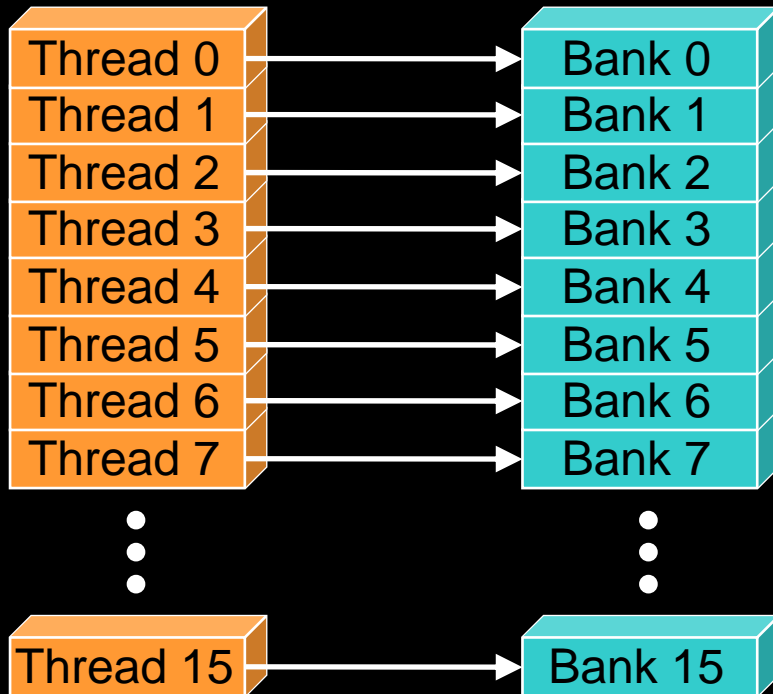


Bank Addressing Examples



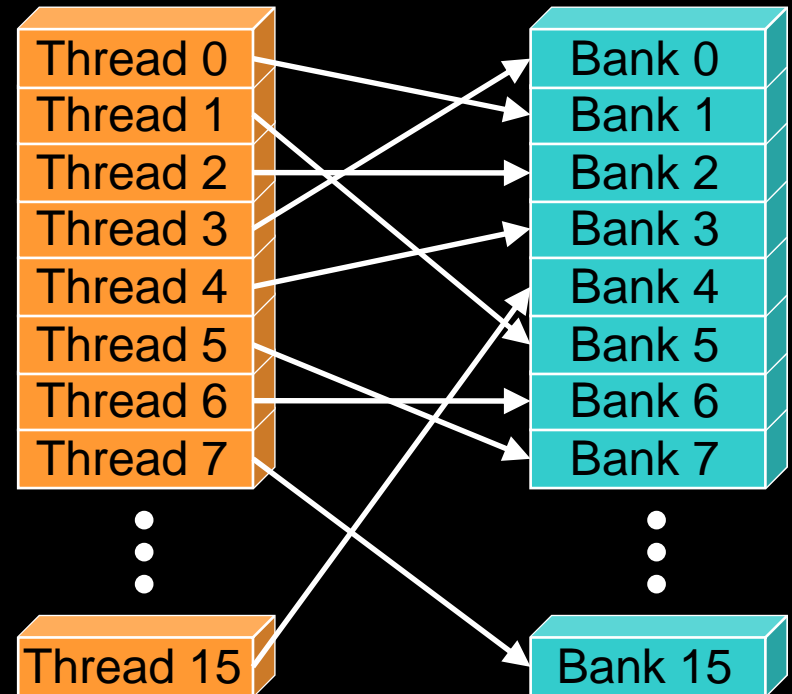
● No Bank Conflicts

- Linear addressing
stride == 1



● No Bank Conflicts

- Random 1:1 Permutation

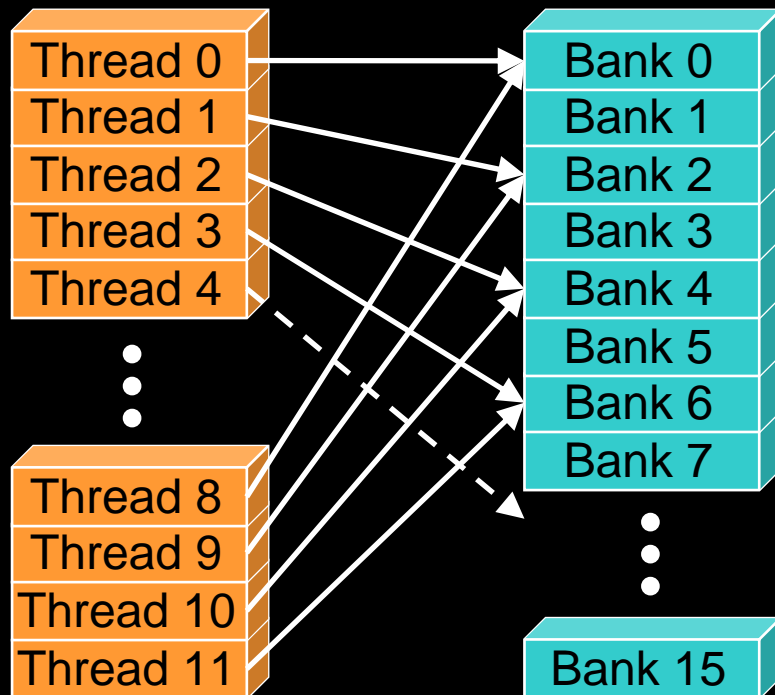


Bank Addressing Examples



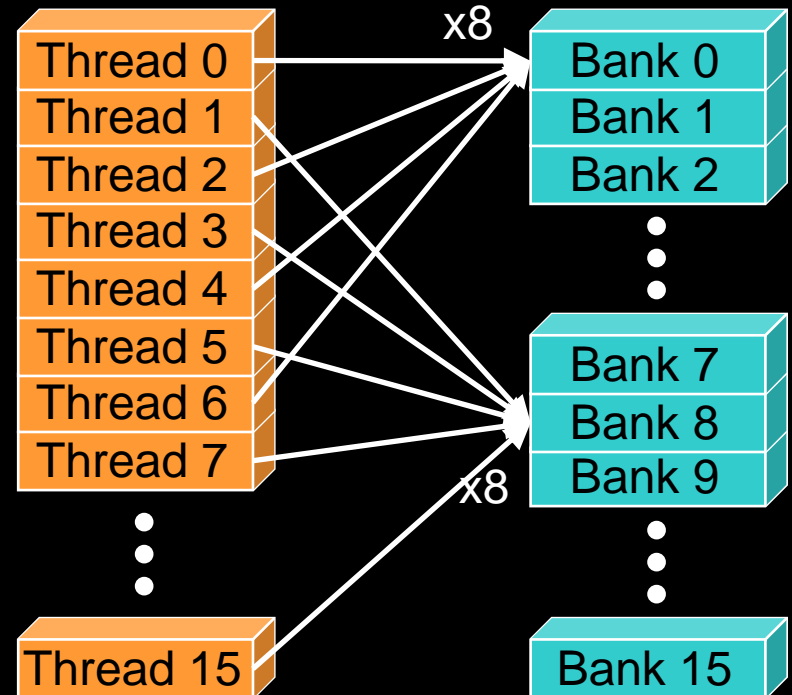
2-way Bank Conflicts

- Linear addressing stride == 2



8-way Bank Conflicts

- Linear addressing stride == 8



Shared memory bank conflicts

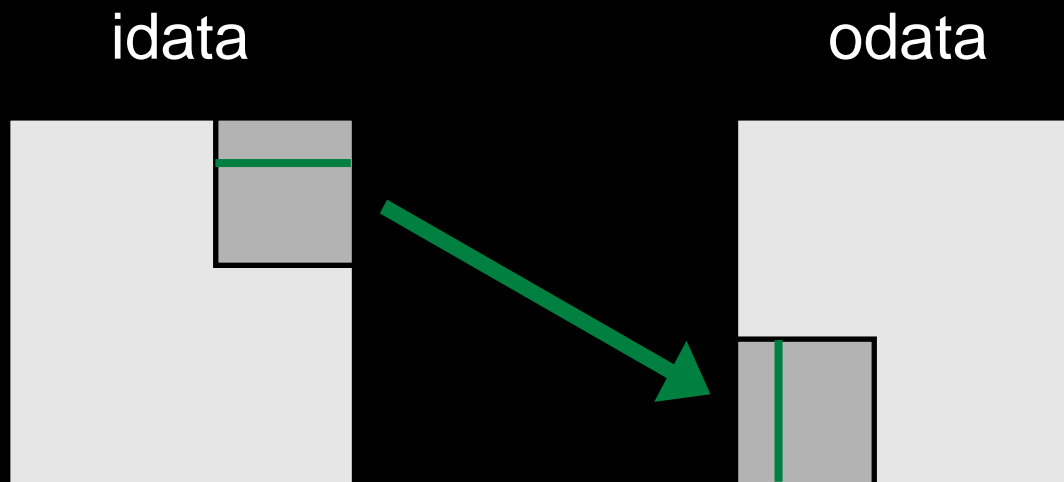


- **Shared memory is ~ as fast as registers if there are no bank conflicts**
- **warp_serialize** profiler signal reflects conflicts
- **The fast case:**
 - If all threads of a half-warp access **different banks**, there is no bank conflict
 - If all threads of a half-warp read the **identical address**,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

Shared Memory Example: Transpose



- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory



Elements transposed by a half-warp of threads

Naïve Transpose

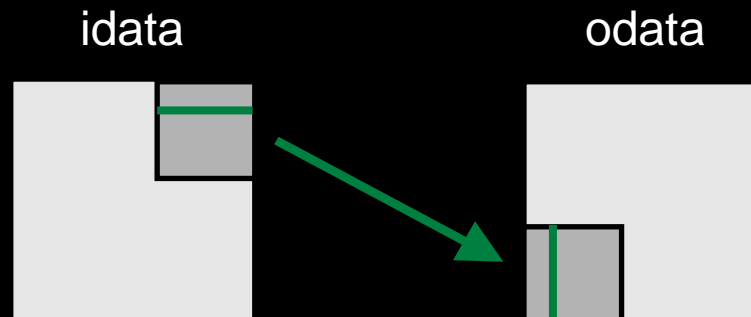


- Loads are coalesced, stores are not (strided by height)

```
__global__ void transposeNaive(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_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;

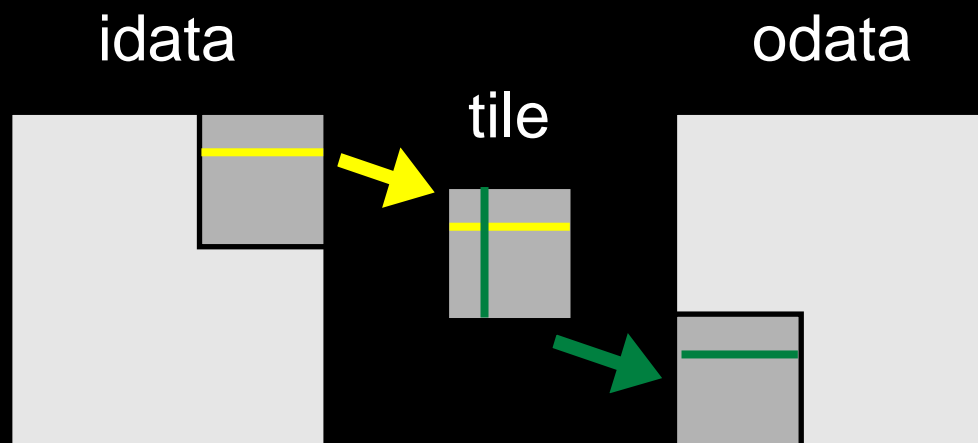
    odata[index_out] = idata[index_in];
}
```



Coalescing through shared memory



- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored 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)
{
    __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;

    tile[threadIdx.y][threadIdx.x] = idata[index_in];

    __syncthreads();

    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Bank Conflicts in Transpose

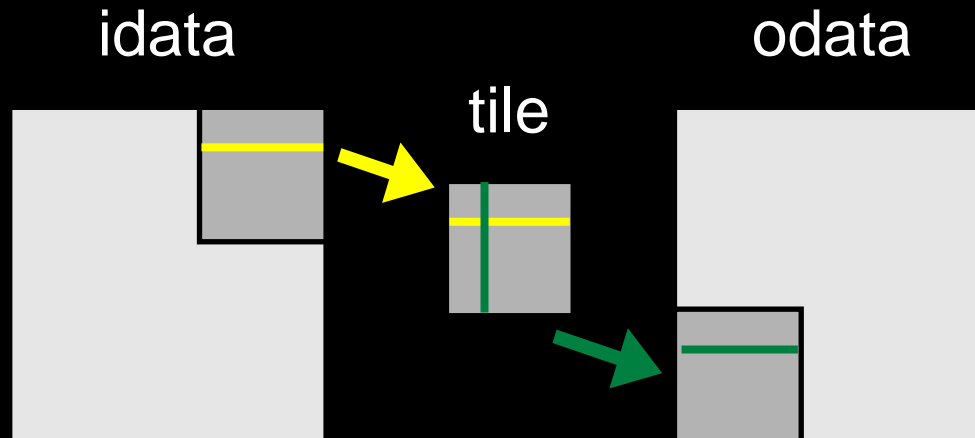


- **16x16 shared memory tile of floats**

- Data in columns are in the same bank
- 16-way bank conflict reading 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

Outline



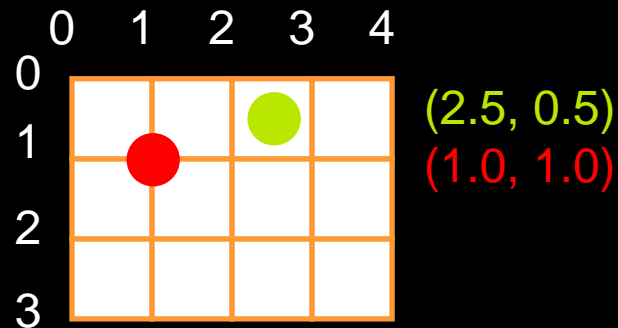
- Overview
- Hardware
- **Memory Optimizations**
 - Data transfers between host and device
 - **Device memory optimizations**
 - Measuring performance – effective bandwidth
 - Coalescing
 - Shared Memory
 - **Textures**
- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Summary

Textures in CUDA



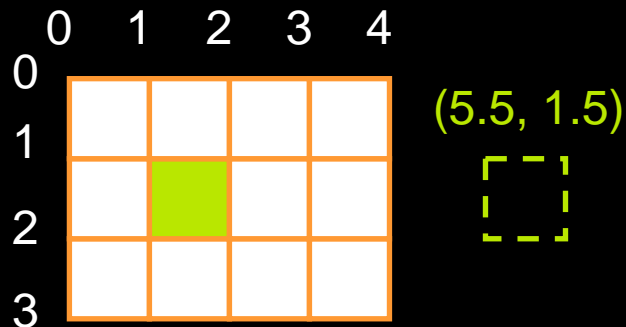
- **Texture is an object for reading data**
- **Benefits:**
 - Data is cached
 - Helpful when coalescing is a problem
 - Filtering
 - Linear / bilinear / trilinear interpolation
 - Dedicated hardware
 - Wrap modes (for “out-of-bounds” addresses)
 - Clamp to edge / repeat
 - Addressable in 1D, 2D, or 3D
 - Using integer or normalized coordinates

Texture Addressing



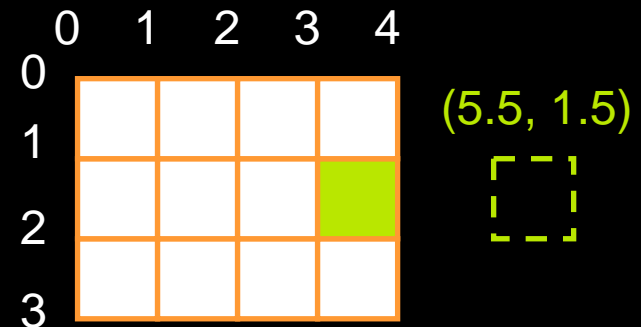
Wrap

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



Clamp

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



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**

- Block linear CUDA array is bound to a texture
- 1D, 2D, or 3D
- Float addressing (size-based or normalized)
- Filtering
- Addressing modes (clamping, repeat)

● **Bound to pitch linear (CUDA 2.2)**

- Global memory address is bound to a texture
- 2D
- Float/integer addressing, filtering, and clamp/repeat addressing modes similar to CUDA arrays

CUDA Texturing Steps



● Host (CPU) code:

- Allocate/obtain memory (global linear/pitch 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()`
- Pitch linear textures: `tex2D()`

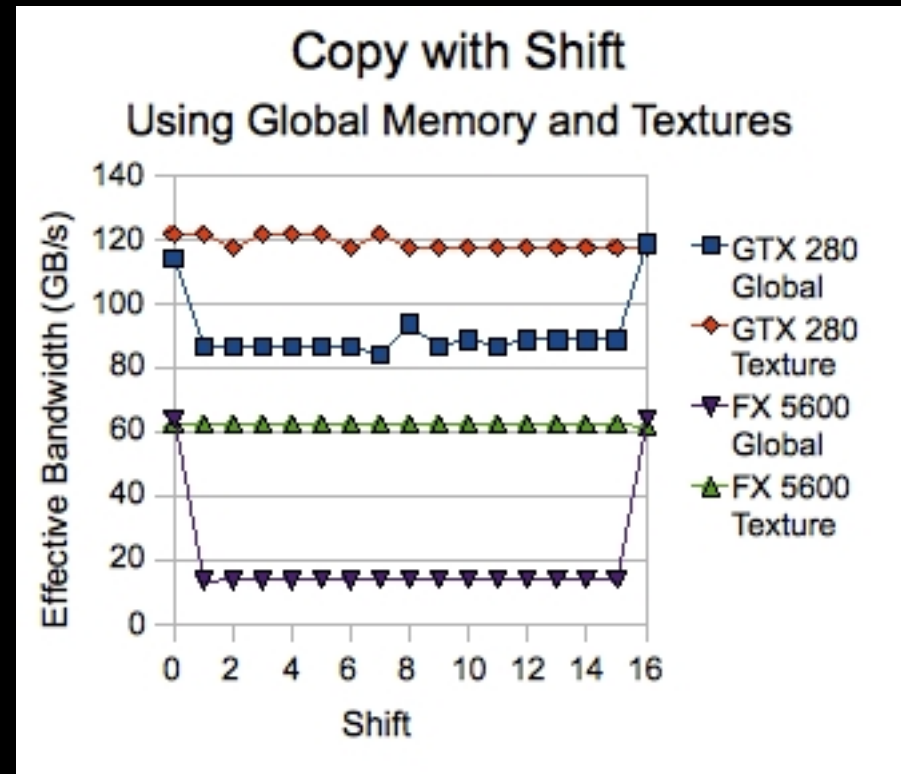
Texture Example



```
__global__ void
shiftCopy(float *odata,
          float *idata,
          int shift)
{
    int xid = blockIdx.x * blockDim.x
            + threadIdx.x;
    odata[xid] = idata[xid+shift];
}

texture <float> texRef;

__global__ void
textureShiftCopy(float *odata,
                 float *idata,
                 int shift)
{
    int xid = blockIdx.x * blockDim.x
            + threadIdx.x;
    odata[xid] = tex1Dfetch(texRef, xid+shift);
}
```



Outline



- Overview
- Hardware
- Memory Optimizations
- **Execution Configuration Optimizations**
- Instruction Optimizations
- Multi-GPU
- 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**

Blocks per Grid 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 ~24 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:

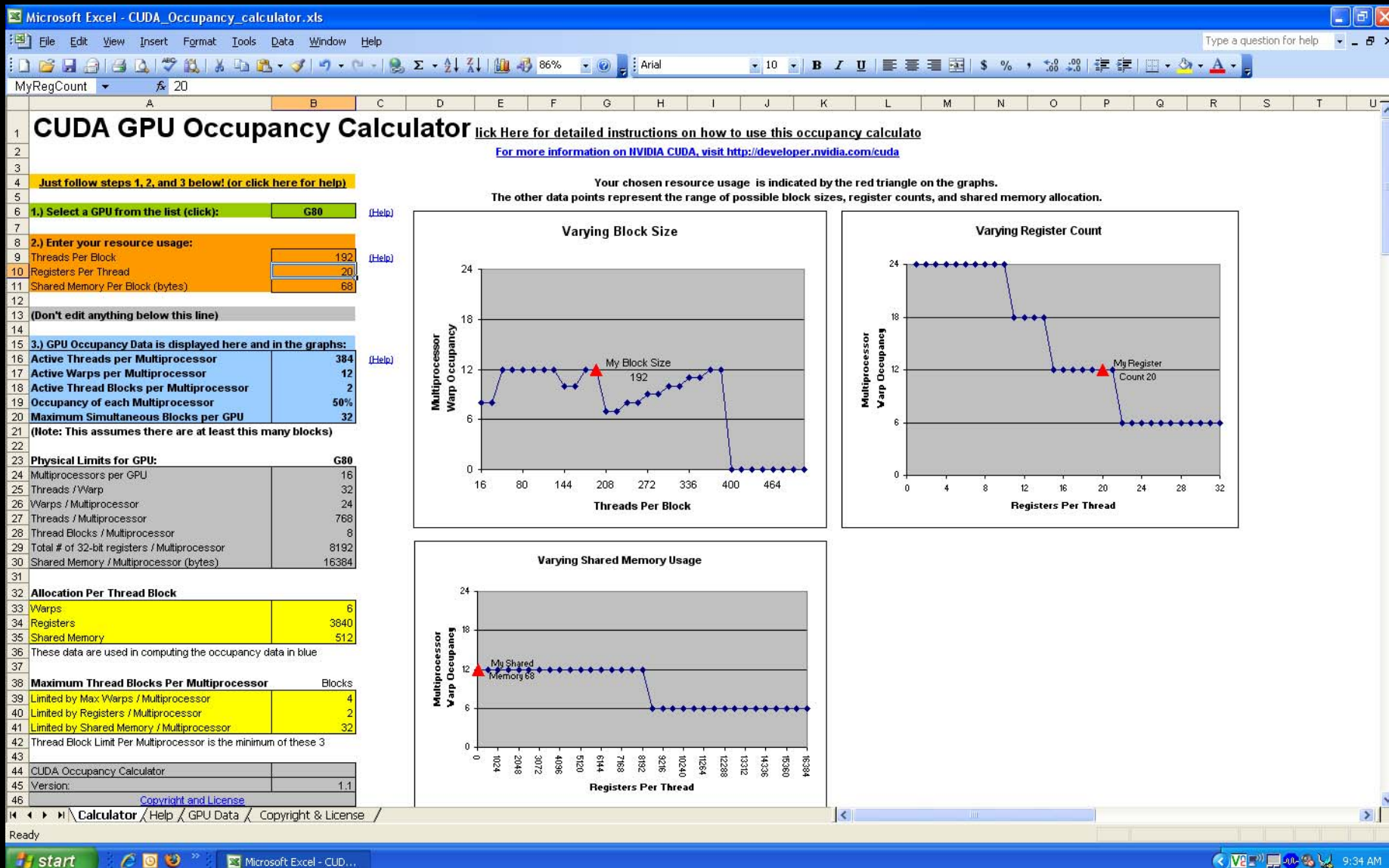
- Run at least **192** threads (6 warps) per multiprocessor
 - At least **25%** occupancy (1.0/1.1), **18.75%** (1.2/1.3)
- Threads do not have to belong to the same thread block

Register Pressure



- Hide latency by using more threads per multiprocessor
- Limiting Factors:
 - Number of registers per kernel
 - 8K/16K per multiprocessor, partitioned among concurrent threads
 - Amount of shared memory
 - 16KB per multiprocessor, 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



Optimizing threads per block



- **Choose threads per block as a multiple of warp size**

- Avoid wasting computation on under-populated warps
- Facilitates coalescing

- **Want to run as many warps as possible per multiprocessor (hide latency)**

- **Multiprocessor can run up to 8 blocks at a time**

- **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**
- Multi-GPU

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

- **There are two types of runtime math operations in single precision**
 - `__funcf()`: direct mapping to hardware ISA
 - Fast but lower accuracy (see prog. guide for details)
 - Examples: `__sinf(x)`, `__expf(x)`, `__powf(x,y)`
 - `funcf()` : compile to multiple instructions
 - Slower but higher accuracy (5 ulp or less)
 - Examples: `sinf(x)`, `expf(x)`, `powf(x,y)`
- **The `-use_fast_math` compiler option forces every `funcf()` to compile to `__funcf()`**

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

Outline



- Overview
- Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- **Multi-GPU**

Why Multi-GPU Programming?

- **Many systems contain multiple GPUs:**
 - Servers (Tesla/Quadro servers and desksides)
 - Desktops (2- and 3-way SLI desktops, GX2 boards)
 - Laptops (hybrid SLI)
- **Additional processing power**
 - Increasing processing throughput
- **Additional memory**
 - Some problems do not fit within a single GPU memory

Multi-GPU Memory

- **GPUs do not share global memory**
 - One GPU cannot access another GPU's memory directly
- **Inter-GPU communication**
 - Application code is responsible for moving data between GPUs
 - Data travels across the PCIe bus
 - Even when GPUs are connected to the same PCIe switch

CPU-GPU Context

- A CPU-GPU context must be established before calls are issued to the GPU
- CUDA resources are allocated per context
- A context is established by the first CUDA call that changes state
 - `cudaMalloc`, `cudaMemcpy`, `cudaFree`, kernel launch, ...
- A context is destroyed by one of:
 - Explicit `cudaThreadExit()` call
 - Host thread terminating

Run-Time API Device Management:

- **A host thread can maintain one context at a time**
 - GPU is part of the context and cannot be changed once a context is established
 - Need as many host threads as GPUs
 - Note that multiple host threads can establish contexts with the same GPU
 - Driver handles time-sharing and resource partitioning
- **GPGUs have consecutive integer IDs, starting with 0**
- **Device management calls:**
 - `cudaGetDeviceCount(int *num_devices)`
 - `cudaSetDevice(int device_id)`
 - `cudaGetDevice(int *current_device_id)`
 - `cudaThreadExit()`

Choosing a Device

- **Properties for a given device can be queried**
 - No context is necessary or is created
 - `cudaGetDeviceProperties(cudaDeviceProp *properties, int device_id)`
 - This is useful when a system contains different GPUs
- **Explicit device set:**
 - Select the device for the context by calling `cudaSetDevice()` with the chosen device ID
 - Must be called prior to context creation
 - Fails if a context has already been established
 - One can force context creation with `cudaFree(0)`
- **Default behavior:**
 - Device 0 is chosen when no explicit `cudaSetDevice` is called
 - Note this will cause multiple contexts with the same GPU
 - Except when driver is in the *exclusive mode* (details later)

Ensuring One Context Per GPU

- Two ways to achieve:
 - Application-control
 - Driver-control
- Application-control:
 - Host threads negotiate which GPUs to use
 - For example, OpenMP threads set device based on OpenMPI thread ID
 - Pitfall: different applications are not aware of each other's GPU usage
 - Call `cudaSetDevice()` with the chosen device ID

Driver-control (Exclusive Mode)

- **To use exclusive mode:**
 - Administrator sets the GPU to exclusive mode using `nvidia-smi` (System Management Tool) is provided with Linux drivers
 - Application: do not explicitly set the GPU in the application
- **Behavior:**
 - Driver will implicitly set a GPU with no contexts
 - Implicit context creation will fail if all GPUs have contexts
 - The first state-changing CUDA call will fail and return an error
- **Device mode can be checked by querying its properties**

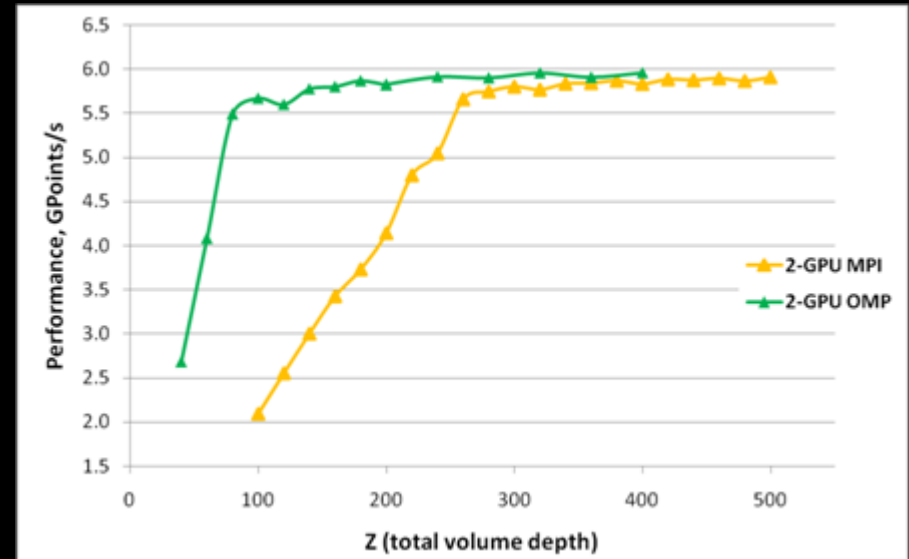
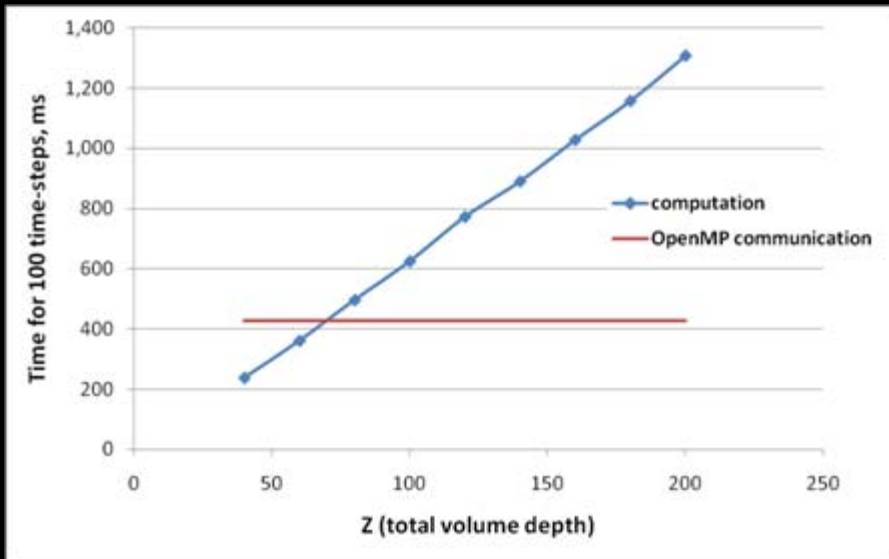
Inter-GPU Communication

- Application is responsible for moving data between GPUs:
 - Copy data from GPU to host thread A
 - Copy data from host thread A to host thread B
 - Use any CPU library (MPI, ...)
 - Copy data from host thread B to its GPU
- Use asynchronous memcopies to overlap kernel execution with data copies
- Lightweight host threads (OpenMP, pthreads) can reduce host-side copies by sharing pinned memory
 - Allocate with `cudaHostAlloc(...)`

Example: Multi-GPU 3DFD

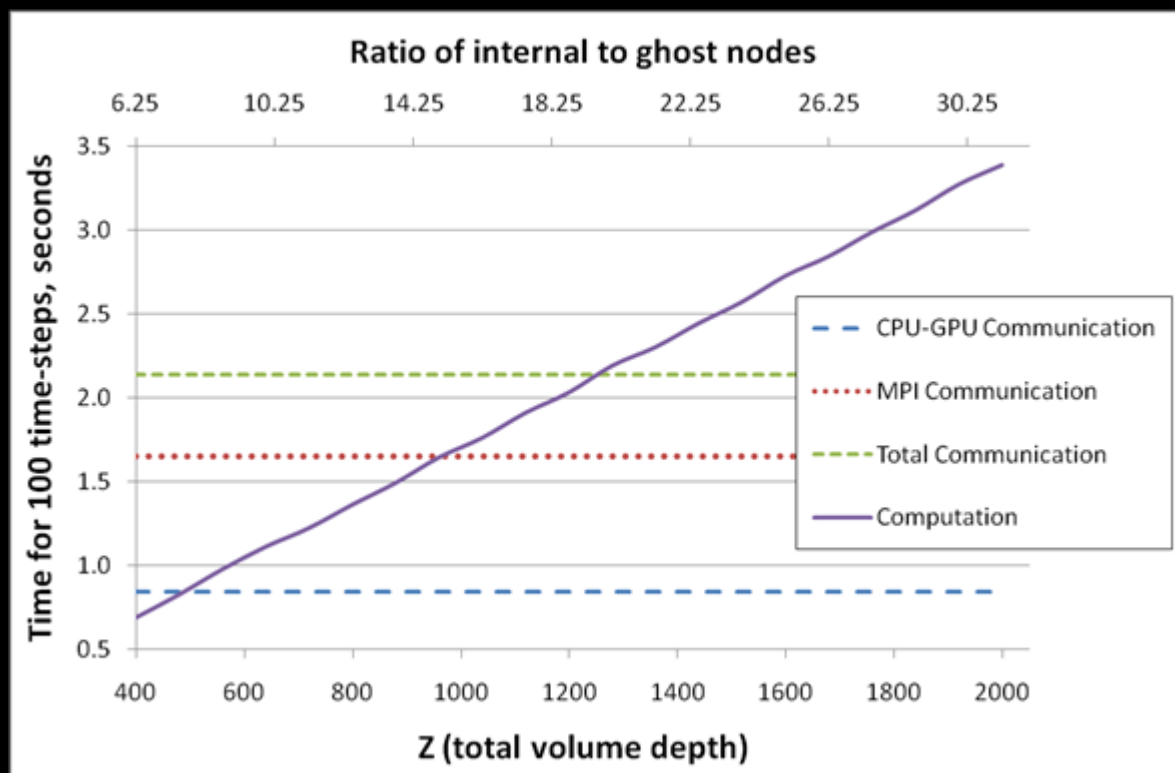
- **3DFD Discretization of the Seismic Wave Equation**
 - 8th order in space, 2nd order in time, regular grid
- **Fixed x and y dimensions, varying z**
- **Data is partitioned among GPUs along z**
 - Computation increases with z , communication (per node) stays constant
 - A GPU has to exchange 4 xy -planes (ghost nodes) with each of its neighbors
- **Cluster:**
 - 2 GPUs per node
 - Infiniband SDR network

2-GPU Performance



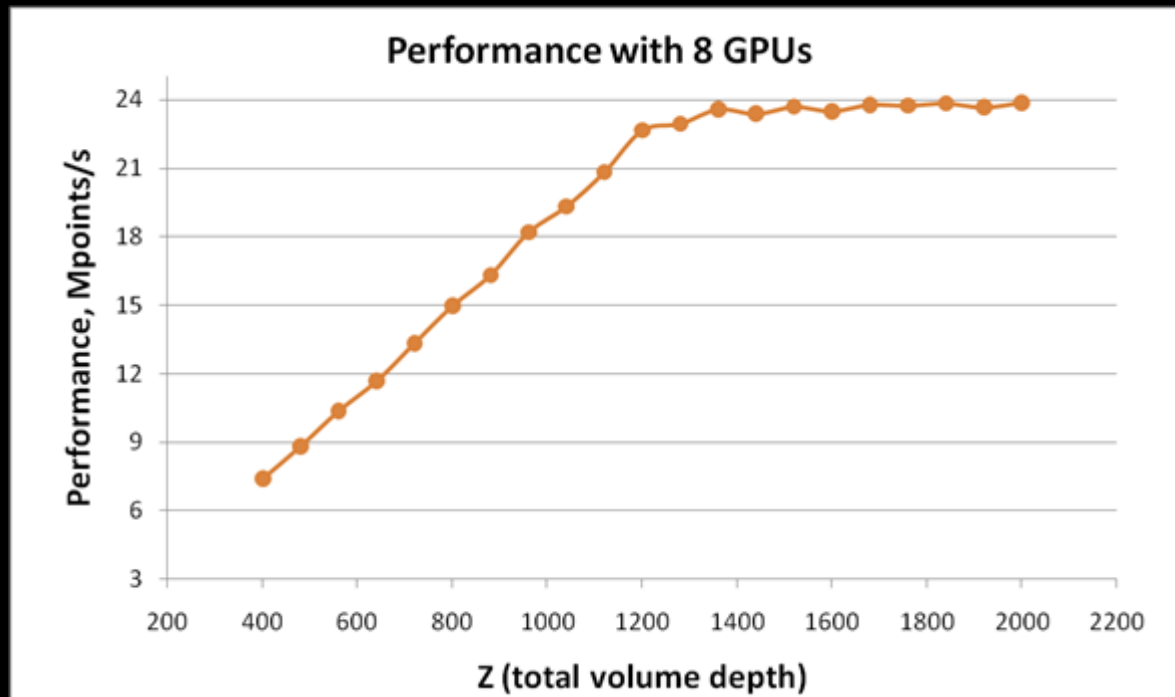
- Linear scaling is achieved when computation time exceeds communication time
 - Single GPU performance is
- OpenMP case requires no copies on the host side (shared pinned memory)
 - Communication time includes only PCIe transactions
- MPI version uses MPI_Sendrecv, which invokes copies on the host side
 - Communication time includes PCIe transactions and host memcopies

3 or more cluster nodes



- Times are per cluster node
- At least one cluster node needs two MPI communications, one with each of the neighbors

Performance Example: 3DFD



- Single GPU performance is ~3,000 MPoints/s
- Note that 8x scaling is sustained at $z > 1,300$
 - Exactly where computation exceeds communication

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**