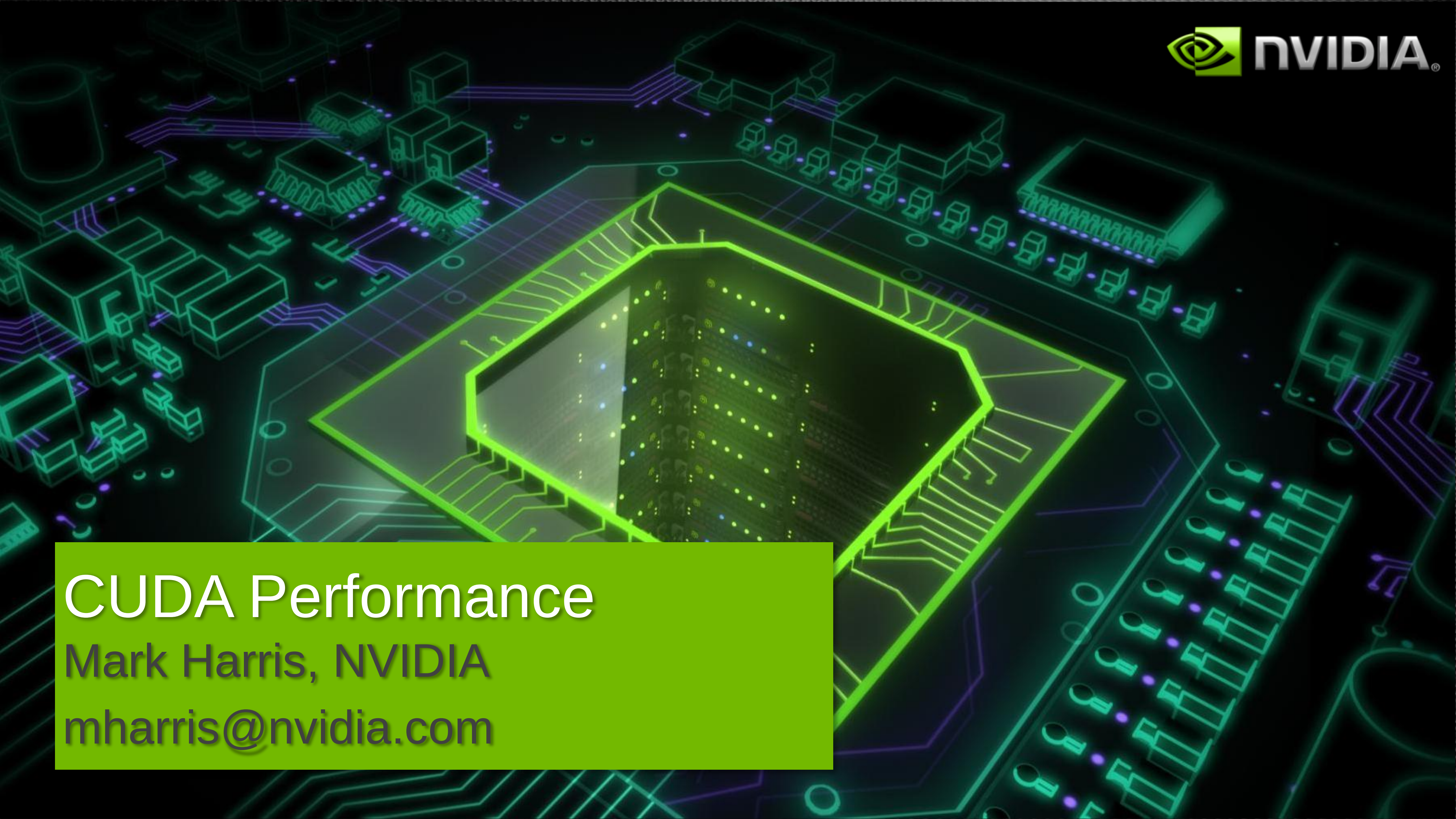


CUDA Performance

Mark Harris, NVIDIA
mharris@nvidia.com

Outline

- Overview
- Hardware Architecture
- Memory Optimizations
    - Data transfers between host and device
    - Device memory optimizations
        - Measuring performance – effective bandwidth
        - Coalescing
        - Shared Memory
        - Textures (if we have time)
- Fermi update

Optimize Algorithms for GPU

- Algorithm selection
  - Understand the problem, consider alternate algorithms
  - Maximize independent parallelism
  - Maximize arithmetic intensity (math/bandwidth)
- Recompute?
  - GPU allocates transistors to arithmetic, not memory
  - Sometimes better to recompute rather than cache
- Serial computation on GPU?
  - Low parallelism computation may be faster on GPU vs copy to/from host

Optimize Memory Access

- Coalesce global memory access
  - Maximise DRAM efficiency
  - Order of magnitude impact on performance
- Avoid serialization
  - Minimize shared memory bank conflicts
  - Understand constant cache semantics
- Understand spatial locality
  - Optimize use of textures to ensure spatial locality

Exploit Shared Memory

- Hundreds of times faster than global memory
- Inter-thread cooperation via shared memory and synchronization
- Cache data that is reused by multiple threads
- Stage loads/stores to allow reordering
  - Avoid non-coalesced global memory accesses

Use Resources Efficiently

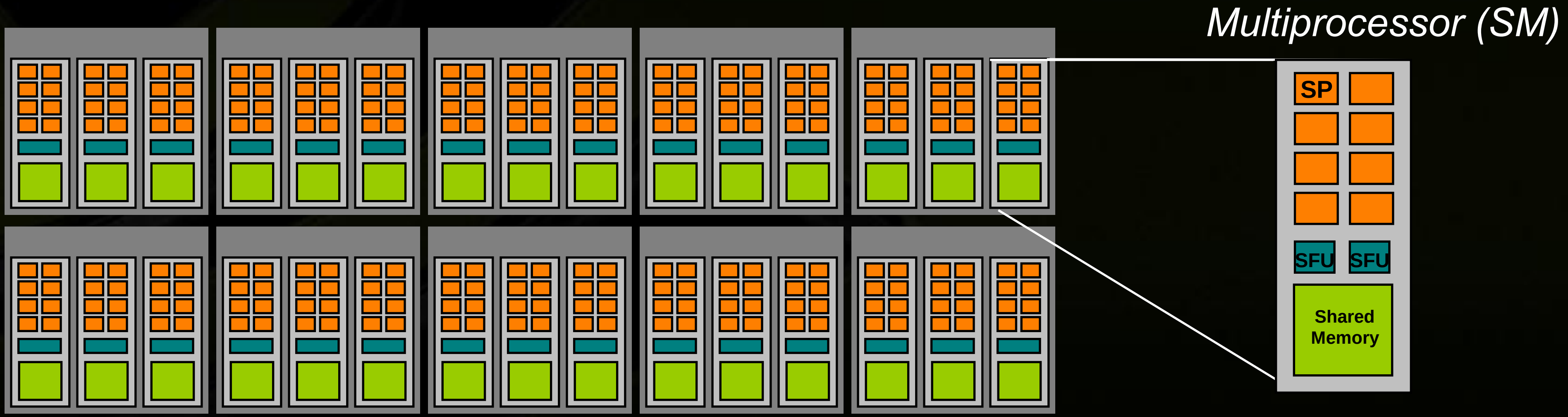
- Partition the computation to keep multiprocessors busy
  - Many threads, many thread blocks
  - Multiple GPUs
- Monitor per-multiprocessor resource utilization
  - Registers and shared memory
  - Low utilization per thread block permits multiple active blocks per multiprocessor
- Overlap computation with I/O
  - Use asynchronous memory transfers

Outline

- Overview
- Hardware Architecture
- Memory Optimizations
    - Data transfers between host and device
    - Device memory optimizations
        - Measuring performance – effective bandwidth
        - Coalescing
        - Shared Memory
        - Textures (if we have time)
- Fermi update

10-Series Architecture

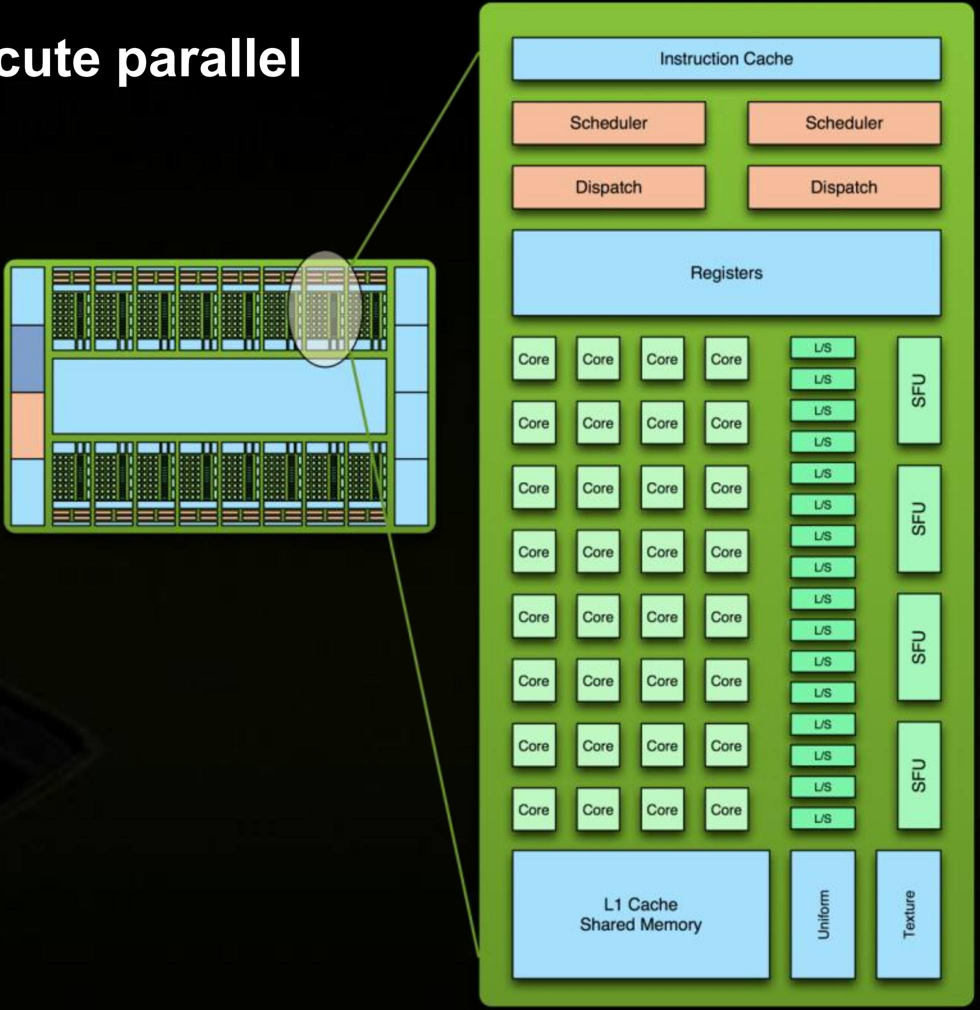
- 240 *Scalar Processor (SP) cores* execute parallel thread instructions
- 30 Streaming *Multiprocessors (SMs)* each contain
  - 8 scalar processors: 8 fp32/int ops / clock, 1 fp64 op / clk
  - 2 Special Function Units (SFUs)
  - *Shared memory* enables thread cooperation



20-Series Architecture

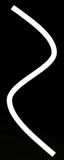
- 448 *Scalar Processor (SP) cores* execute parallel thread instructions

- 14 Streaming *Multiprocessors (SMs)* each contains
  - 32 scalar processors
    - 32 fp32 / int32 ops / clock,
    - 16 fp64 ops / clock
  - 4 Special Function Units (SFUs)
  - Shared register file (128KB)
  - *48 KB / 16 KB Shared memory*
  - 16KB / 48 KB L1 data cache



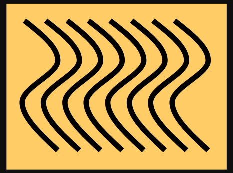
Execution Model

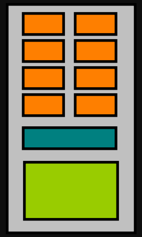
Software **Hardware**



Scalar Processor

Threads are executed by scalar processors

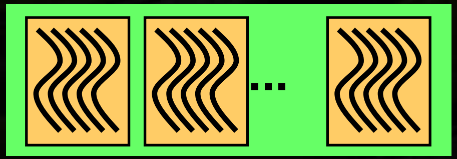


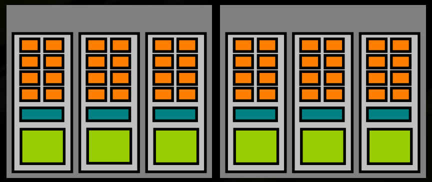


Thread blocks are executed on multiprocessors

Thread blocks do not migrate

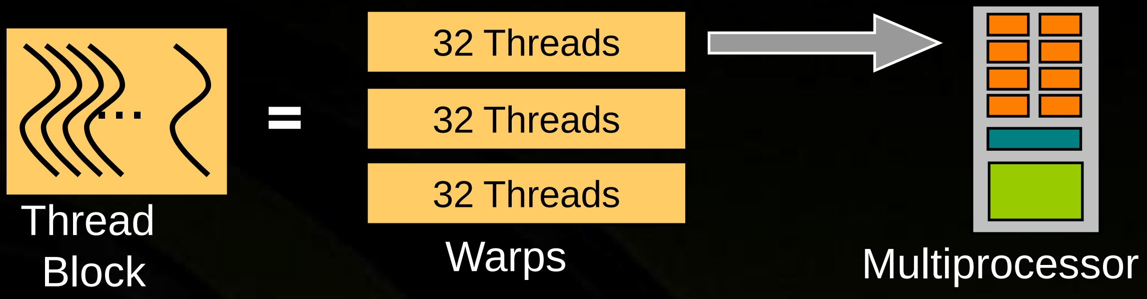
Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)





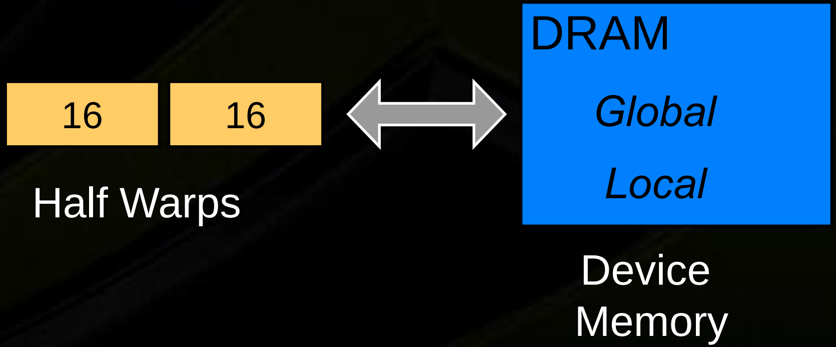
A kernel is launched as a grid of thread blocks

Warps and Half Warps



A thread block consists of 32-thread warps

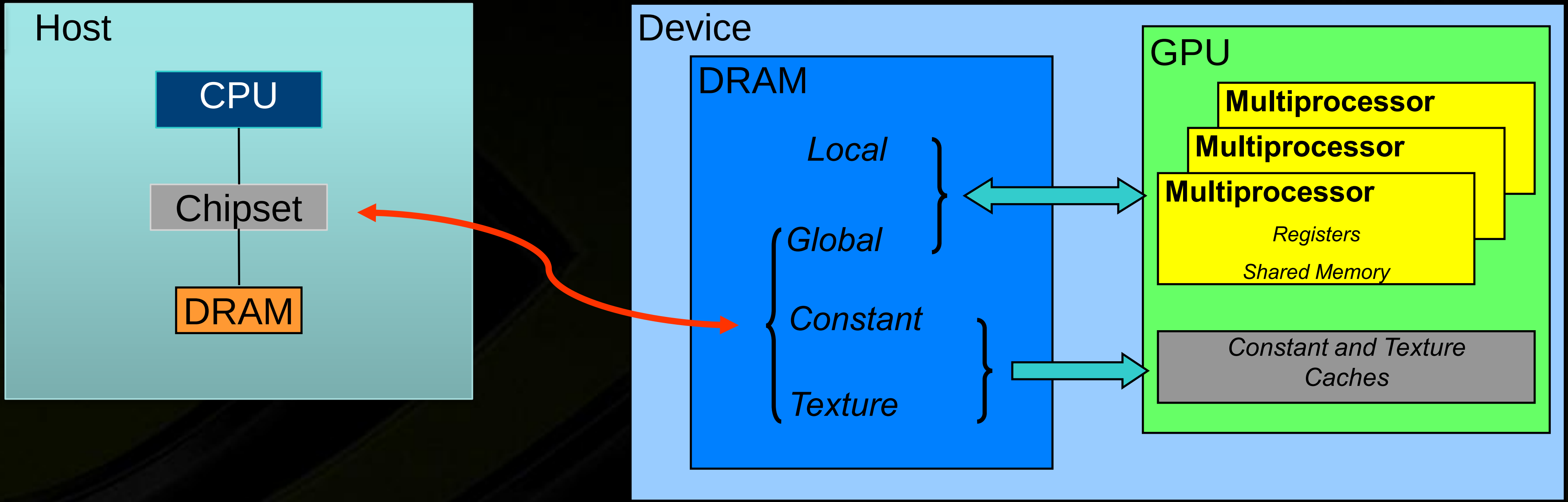
A warp is executed physically in parallel (SIMD) on a multiprocessor



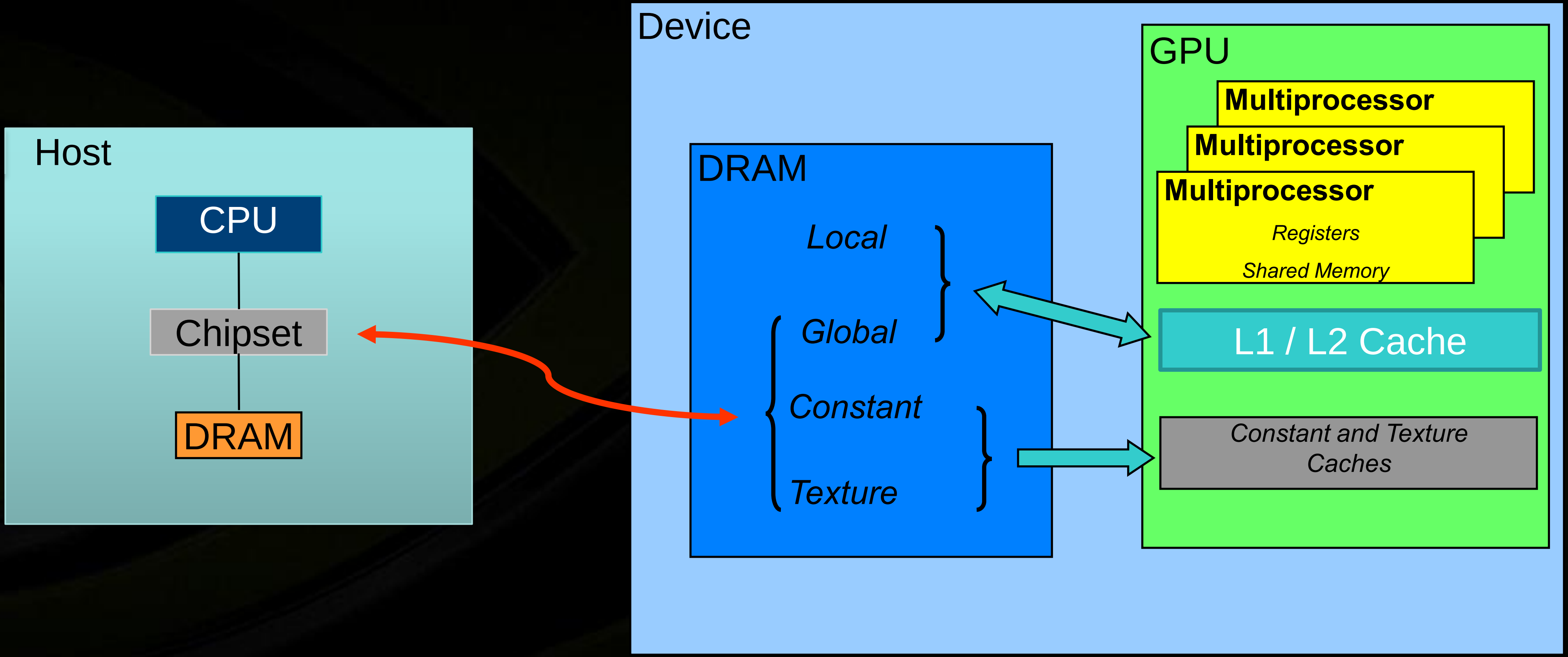
A half-warp of 16 threads can coordinate global memory accesses into a single transaction (compute capability < 2.0)

On compute capability >= 2.0, a warp of 32 threads can coordinate global memory accesses into a single transaction

Memory Architecture: Tesla T-10 Series



Memory Architecture: Tesla T-20 Series



Memory Architecture: Tesla T-10 Series

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

Memory Architecture: Tesla T-20 Series

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	Yes	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	Yes	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 Architecture
- Memory Optimizations
    - Data transfers between host and device
    - Device memory optimizations
        - Measuring performance – effective bandwidth
        - Coalescing
        - Shared Memory
        - Textures (if we have time)
- Fermi update

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. 144 GB/s peak (C2050)
- Minimize transfers
  - Intermediate data can be allocated, operated on, and deallocated without ever copying 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
  - Page-locking too much 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)
  - Tesla T-20 Series can overlap bidirectional transfers with device computation and host computation
- 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 *page-locked* 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();
```

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

GPU/CPU Synchronization

- Context based
  - `cudaThreadSynchronize()`
    - Blocks until all outstanding CUDA calls from 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

- Always a win for integrated devices that utilize CPU memory
  - Can check using the integrated field in cudaDeviceProp
- Faster if data only read/written from/to global memory once, e.g.
  - Copy input data to GPU memory
  - Run one kernel
  - Copy output data back to CPU memory
- Coalescing is even more important with zero copy!
- Potentially easier and faster alternative to cudaMemcpyAsync
  - For example, can both read and write CPU memory from within one kernel
- Note: current devices use 32-bit addressing so limited to *4GB per context*

Outline

- Overview
- Hardware Architecture
- Memory Optimizations
    - Data transfers between host and device
    - Device memory optimizations
        - Measuring performance – effective bandwidth
        - Coalescing
        - Shared Memory
        - Textures (if we have time)
- Fermi update

Theoretical Bandwidth

- Device Bandwidth of GTX 280

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

  - 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)**
  - **N * 4 B/element / 1024³ * 2 / (time in secs) = GB/s**

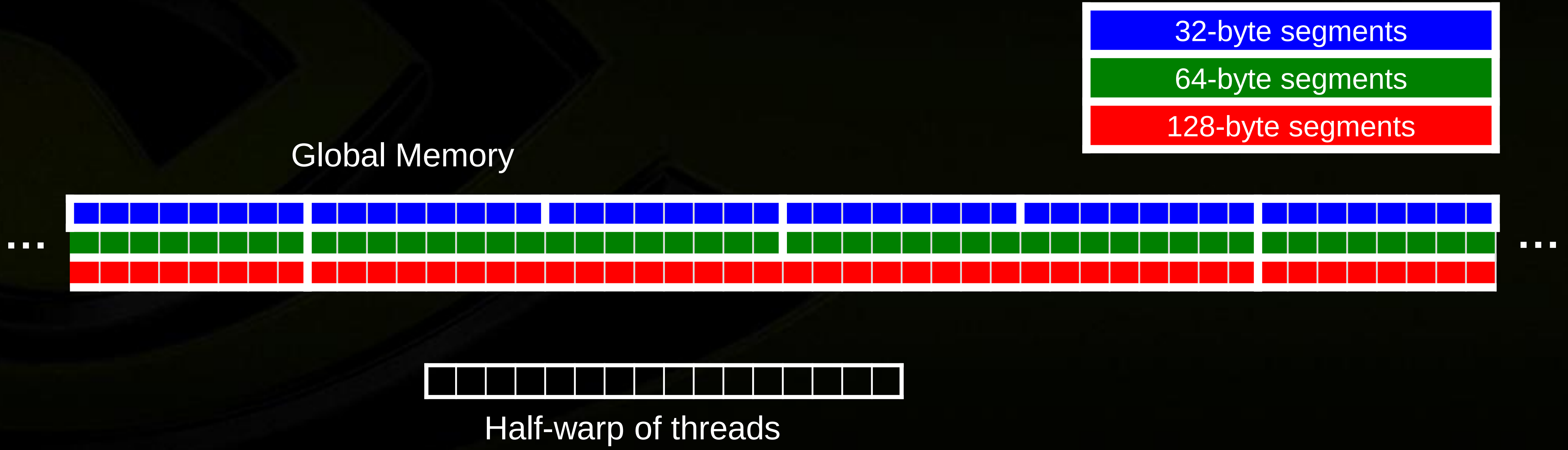
$$\underbrace{N * 4\ \text{B/element}}_{\text{Array size (bytes)}} \;/\; \underset{\uparrow\,\text{B/GB (or } 10^9)}{1024^3} * \underset{\uparrow\,\text{Read and write}}{2} \;/\; (\text{time in secs}) = \text{GB/s}$$

Outline

- Overview
- Hardware Architecture
- Memory Optimizations
    - Data transfers between host and device
    - Device memory optimizations
        - Measuring performance – effective bandwidth
        - Coalescing
        - Shared Memory
        - Textures (if we have time)
- Fermi update

Coalescing

- Global memory access of aligned 32, 64, or 128-bit segment by a half-warp
  - Results in as few as one (or two) transaction(s) if access requirements are met
- Depends on compute capability (CC)
  - Devices with CC < 1.2 have stricter access requirements
- Float (32-bit) data example:

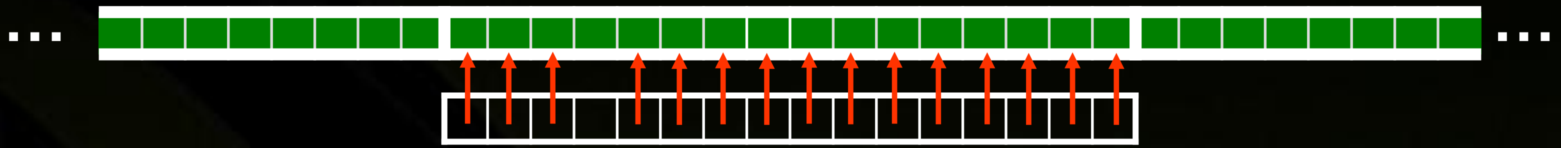


Coalescing

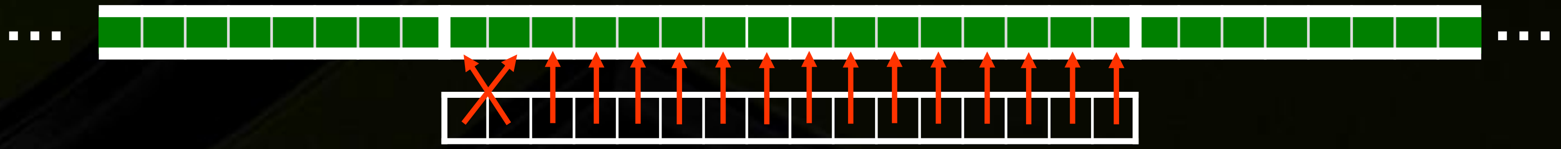
Compute capability 1.0 and 1.1

- **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**

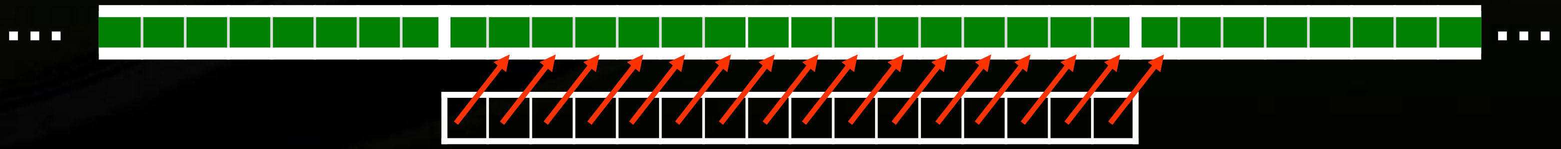
Coalesces – 1 transaction



Out of sequence – 16 transactions



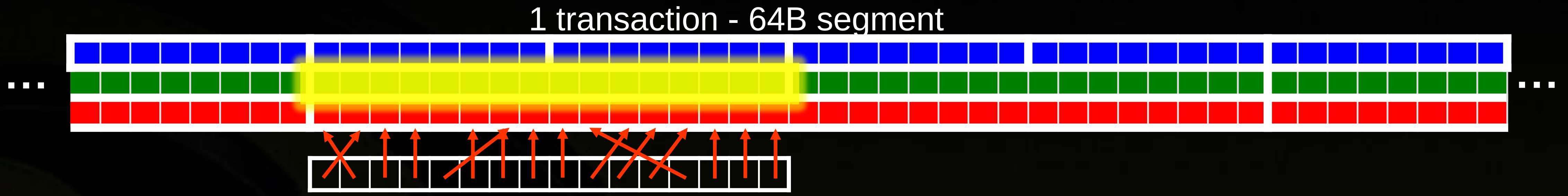
Misaligned – 16 transactions

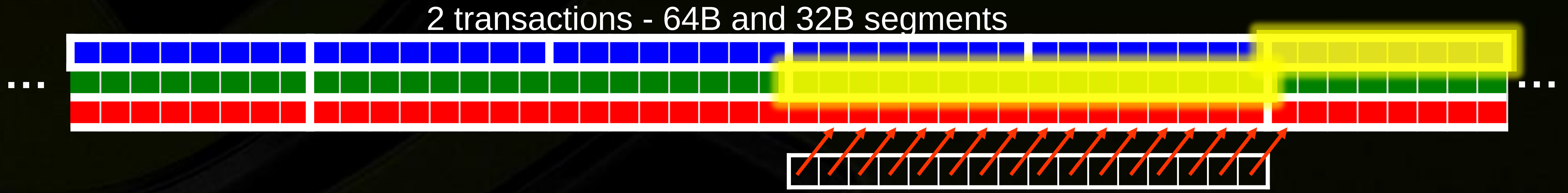


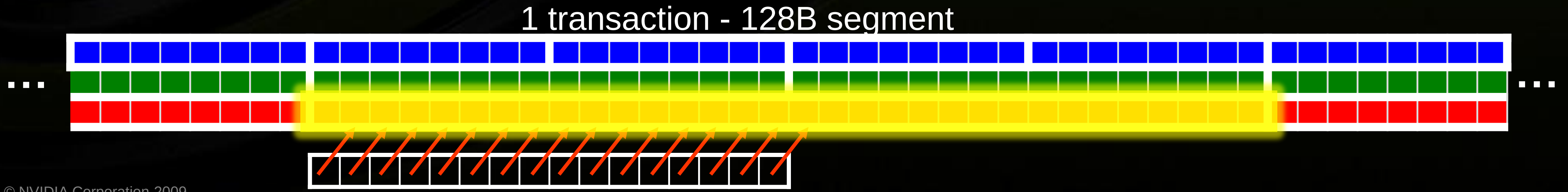
Coalescing

Compute capability 1.2 and higher

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





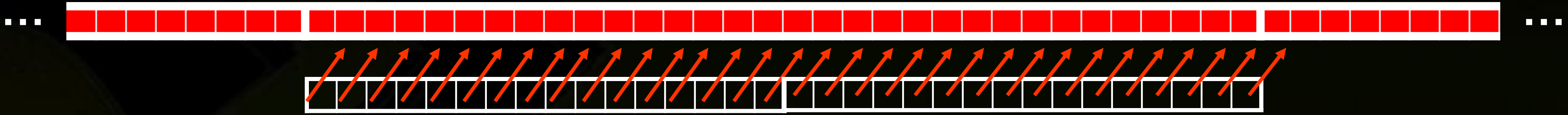


Coalescing

Compute capability 2.0 (Fermi, Tesla C2050)

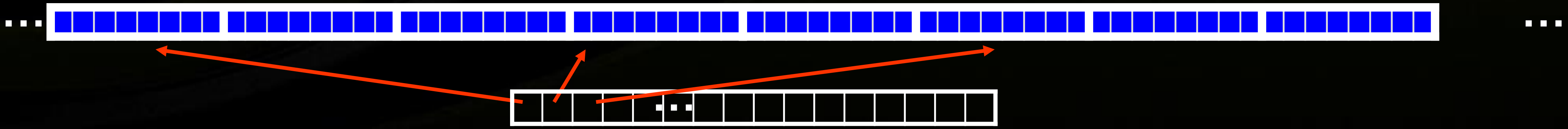
- Memory transactions handled per warp (32 threads)
- L1 cache ON:
  Issues always 128B segment transactions
  caches them in 16kB or 48kB L1 cache per multiprocessor

2 transactions - 2 x 128B segment - but next warp probably only 1 extra transaction, due to L1 cache.



- L1 cache OFF (configure with compiler switch – see programming guide):
  Issues always 32B segment transactions
- E.g. advantage for widely scattered thread accesses

32 transactions - 32 x 32B segments, instead of 32 x 128B segments.



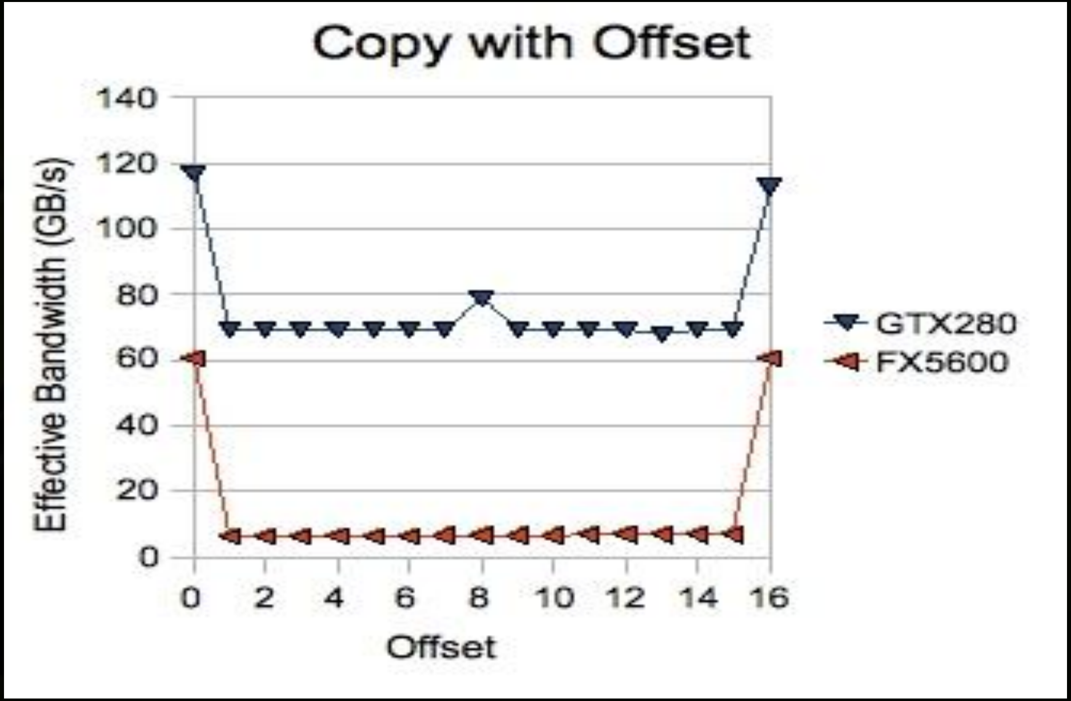
Coalescing Examples

- 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

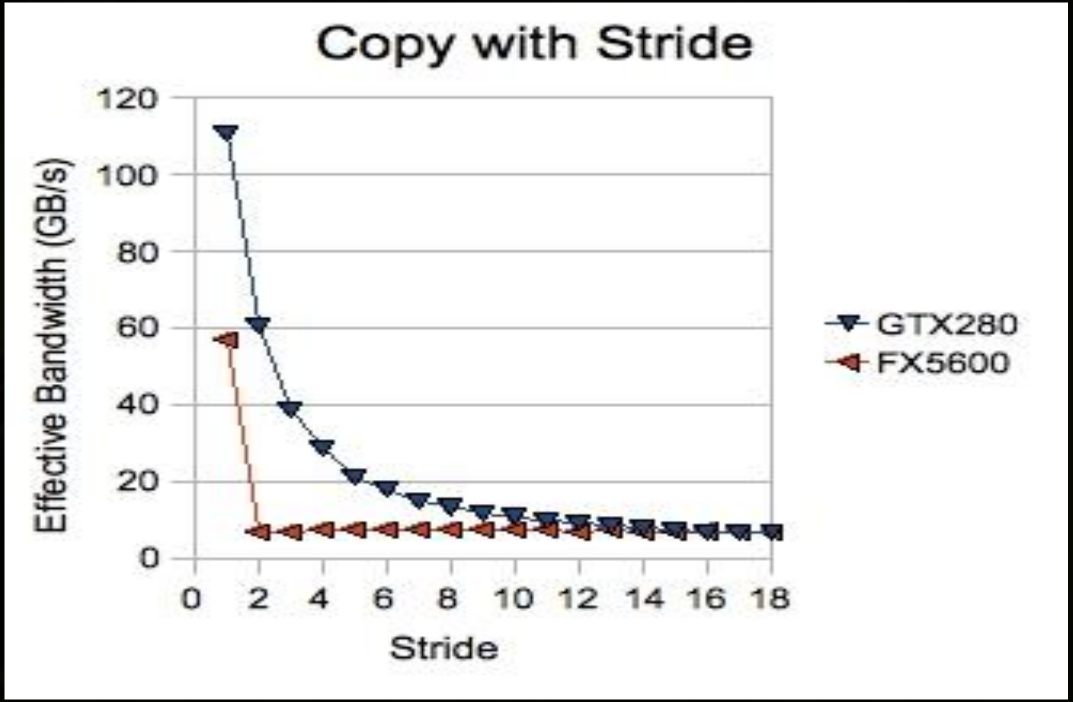
Coalescing Examples

```
__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];
}
```

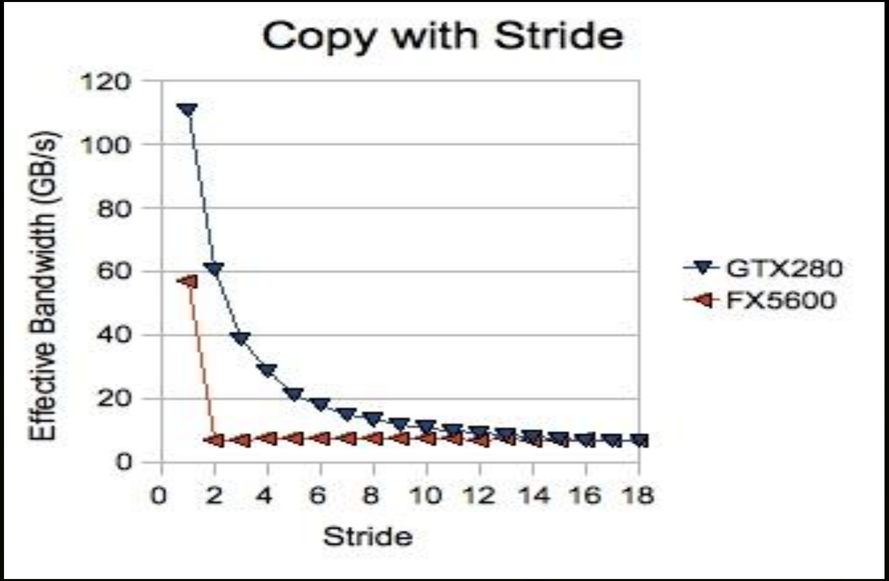


Coalescing Examples

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

However …

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



Outline

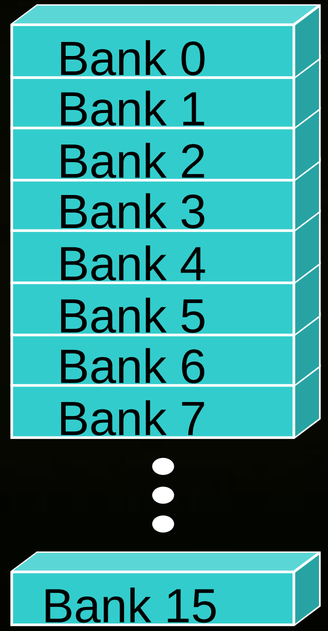
- Overview
- Hardware Architecture
- Memory Optimizations
    - Data transfers between host and device
    - Device memory optimizations
        - Measuring performance – effective bandwidth
        - Coalescing
        - Shared Memory
        - Textures (if we have time)
- Fermi update

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

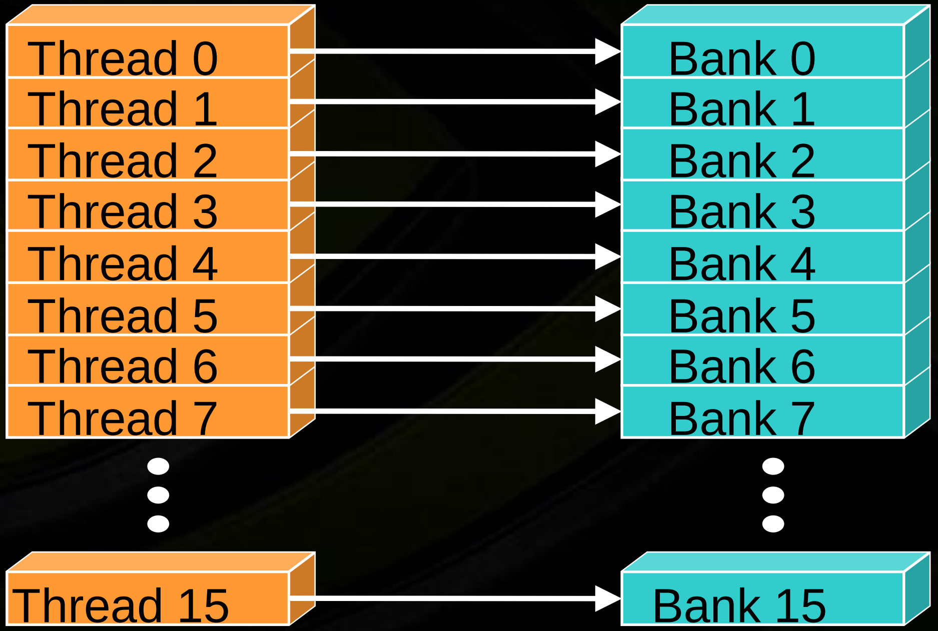
Shared Memory Architecture

- 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

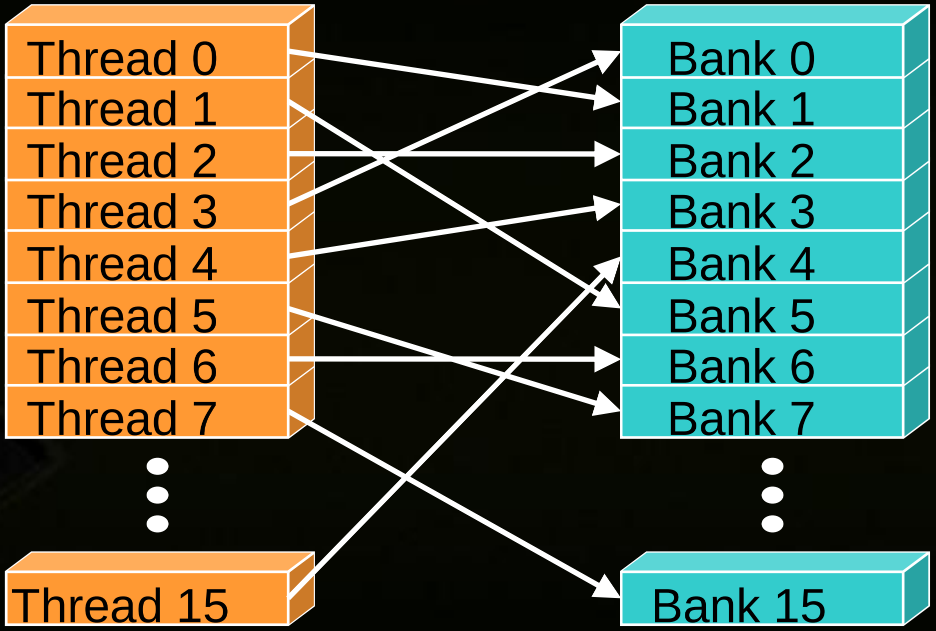


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

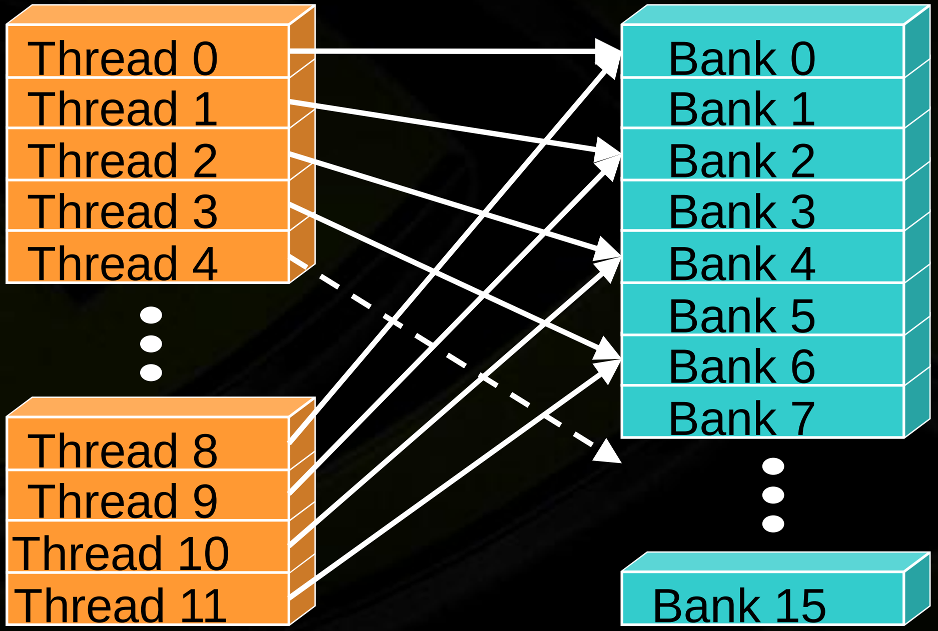


- No Bank Conflicts
  - Random 1:1 Permutation

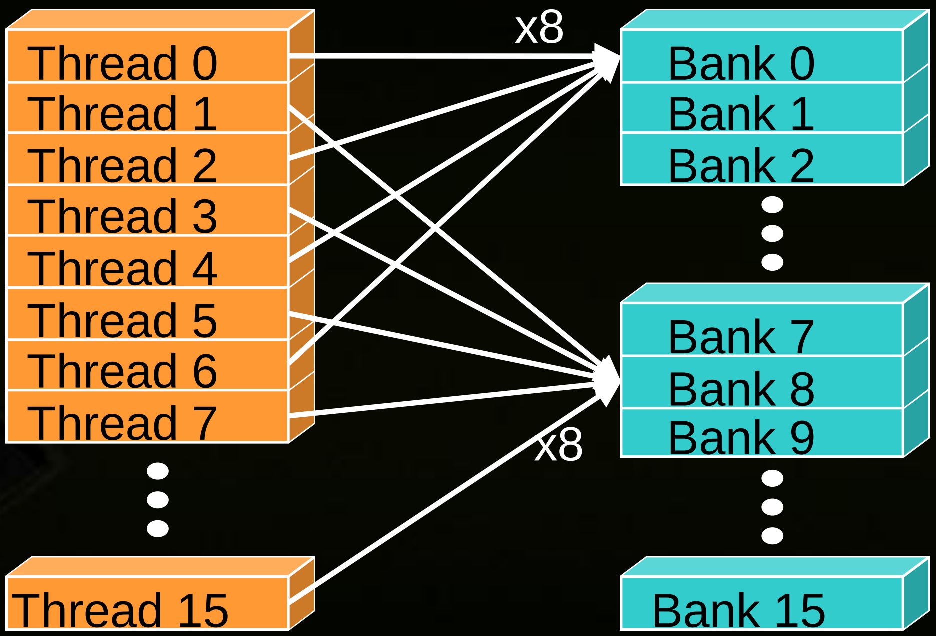


Bank Addressing Examples

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



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

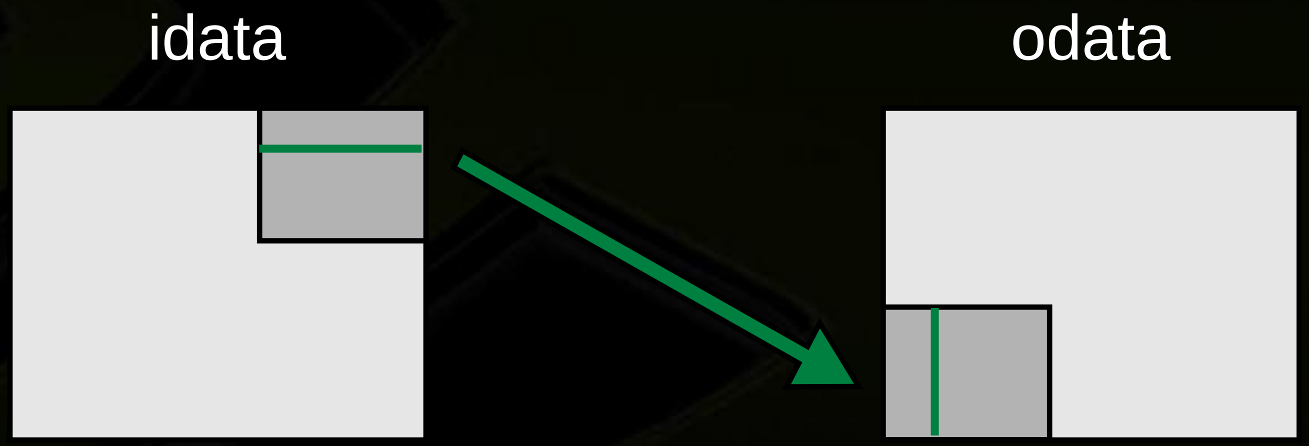


Shared memory bank conflicts

- 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 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

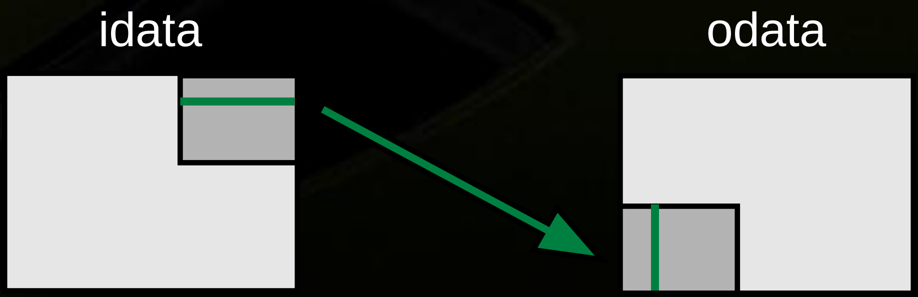


Elements transposed by a half-warp of threads

Naïve Transpose

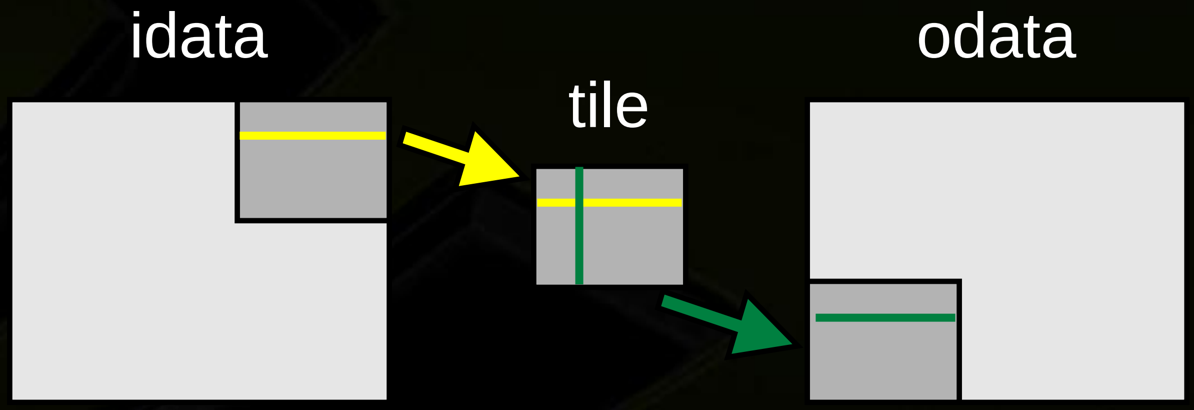
- Loads are coalesced, stores are not (strided by height)

```
__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



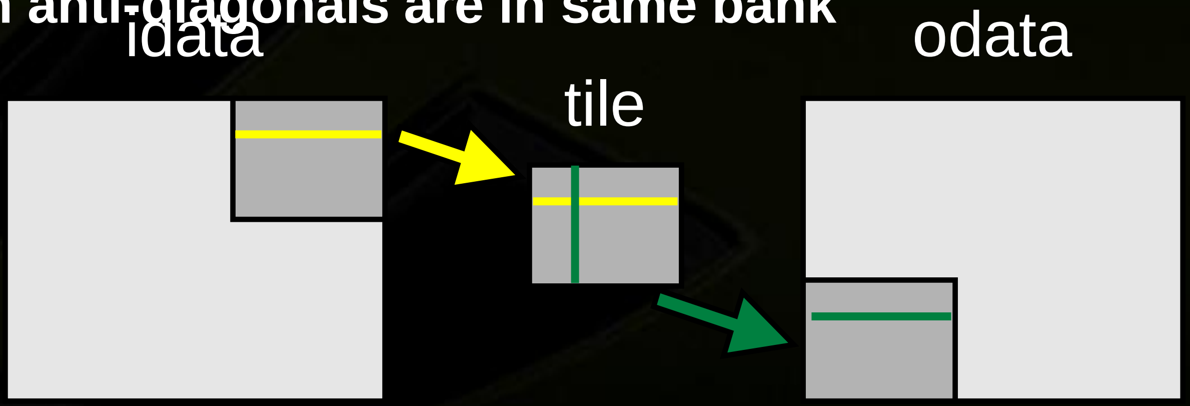
Elements transposed by a half-warp of threads

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 Architecture
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - Textures (if we have time)
- Fermi update

Fermi

- Up to 1536 resident threads per multiprocessor (48 warps)
- Memory operations are done per warp (32 threads) instead of half-warp
  - Global memory, Shared memory, constant memory
- Shared memory:
  - 16 or 48KB
  - Now 32 banks, 32-bit wide each
  - No bank-conflicts when accessing 8-byte words
- L1 cache per multiprocessor
  - Should help with misaligned access, strides access, some register spilling
- Much improved dual-issue:
  - Can dual issue fp32 pairs, fp32-mem, fp64-mem, etc.
- IEEE-conformant rounding
- ECC option, 64bit address space, generic

Fermi: L1 cache

- L1 cache designed for parallel (spatial) re-use, not temporal
  - Similar to coalescing
- Partitions 64kb with shared memory:
  - Switch size between 16KB and 48KB (CUDA API call)
- Caches gmem reads only
  - It benefits if compiler detects that all threads load same value (LDU, load uniform)
- L1 cache can be deactivated: Smaller granularity of memory transactions
- L1 cache per multiprocessor
  - Not coherent!
- Caches local memory reads <u>and</u> writes
  - To improve spilling behaviour
  - (Coherence no problem as local memory SM-private)

Fermi: Constant (uniform) Loads

- Before Fermi, uniform data for all threads should be kept in __constant__
  - Reads go through constant cache, so only first warp that reads pays for global load
- Fermi has a new "load uniform" instruction that can perform similar constant caching for any global memory location that is uniformly accessed.

- <u>Condition</u>:
  a) prefix pointer with `const` keyword
  b) Memory access must be uniform across all threads in the block, e.g.

```
__global__ void kernel( float *g_dst, const float *g_src )
{
    g_dst = g_src[0] + g_src[blockIdx.x];
}
```

Fermi: Additional capabilities

- Fermi can execute several kernels concurrently
  - Thread blocks from one kernel are launched first
  - If there are resources available, thread blocks from a kernel in another stream are launched
    - Streams are used to define kernel independence
- Fermi has 2 copy-engines
  - Can concurrently copy CPU-GPU and GPU-CPU across PCIe
    - PCIe is duplex, so aggregate bandwidth is doubled in such cases
  - Previous generation could only do one copy at a time
  - Copies must be from separate streams

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

CUDA Performance Optimization

Questions?

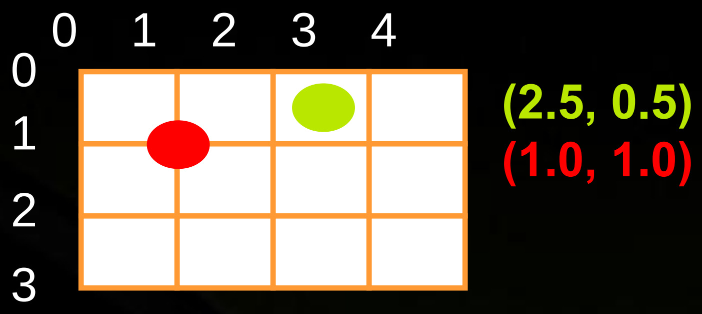
Outline

- Overview
- Hardware
- Memory Optimizations
    - Data transfers between host and device
    - Device memory optimizations
        - Measuring performance – effective bandwidth
        - Coalescing
        - Shared Memory
        - Textures
- Summary

Textures in CUDA

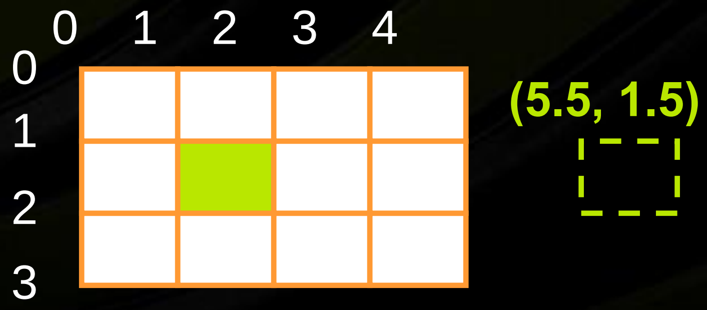
- Texture is an object for ***reading*** data
- Benefits:
  - Data is cached
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear interpolation
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable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

Texture Addressing



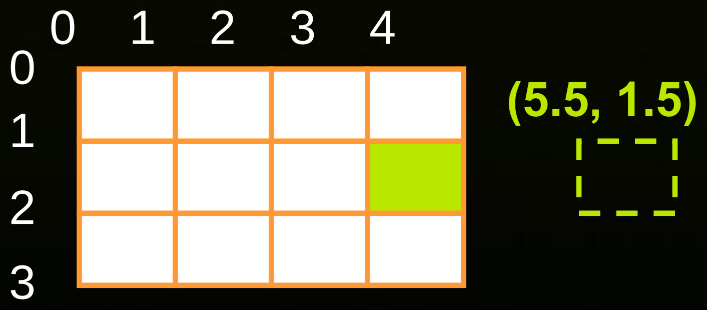
Wrap

- 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

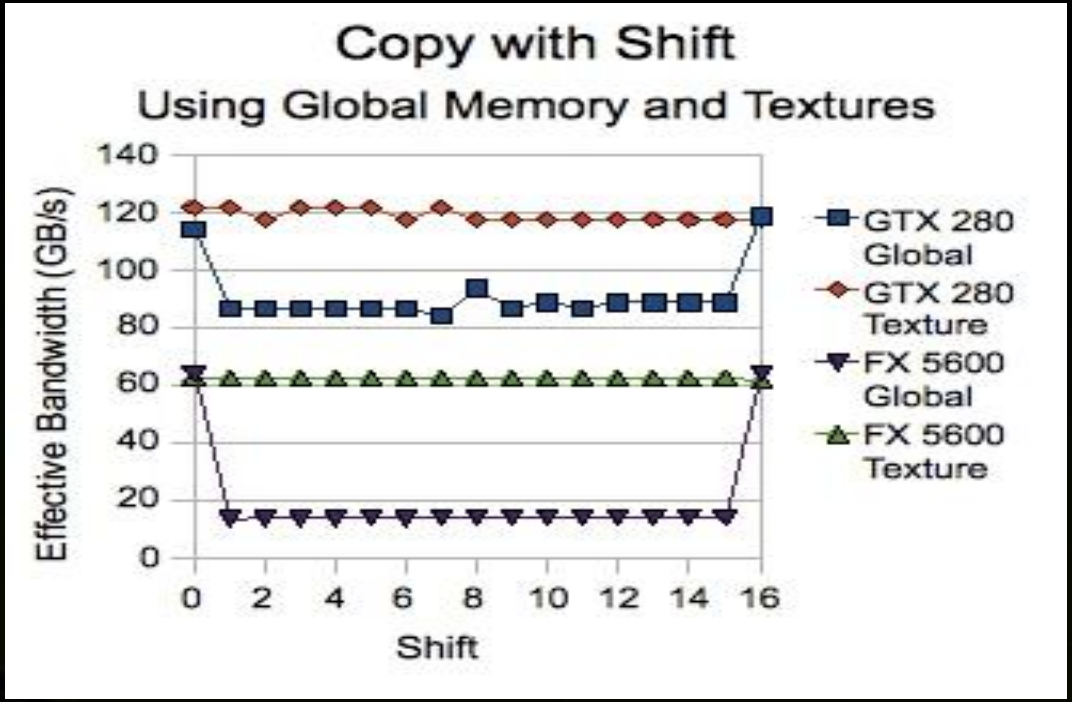
CUDA Texturing Steps

- 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);
}
```



Constant Memory

- Data stored in global memory, read through constant cache
    - __constant__ qualifier in declarations
    - Can only be read by GPU kernels
    - Limited to 64KB
- To be used when all threads in a warp read the same address
    - Serializes otherwise
- Throughput:
    - 32 bits per warp per clock per multiprocessor

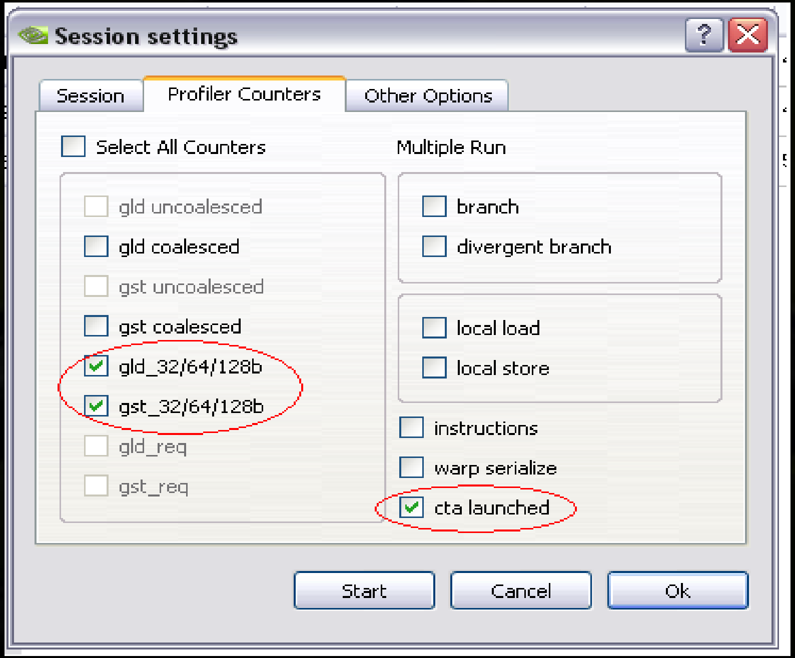
Memory Throughput as Performance Metric

Global Memory Throughput Metric

- Many applications are memory throughput bound
- When coding from scratch:
    - Start with memory operations first, achieve good throughput
    - Add the arithmetic, measuring perf as you go
- When optimizing:
    - Measure effective memory throughput
    - Compare to the theoretical bandwidth
        - 70-80% is very good, ~50% is good if arithmetic is nontrivial
- Measuring throughput
    - From the app point of view (“useful” bytes)
    - From the hw point of view (actual bytes moved across the bus)
    - The two are likely to be different
        - Due to coalescing, discrete bus transaction sizes

Measuring Memory Throughput

- Latest Visual Profiler reports memory throughput
  - From HW point of view
  - Based on counters for one TPC (3 multiprocessors)
  - Need compute capability 1.2 or higher GPU



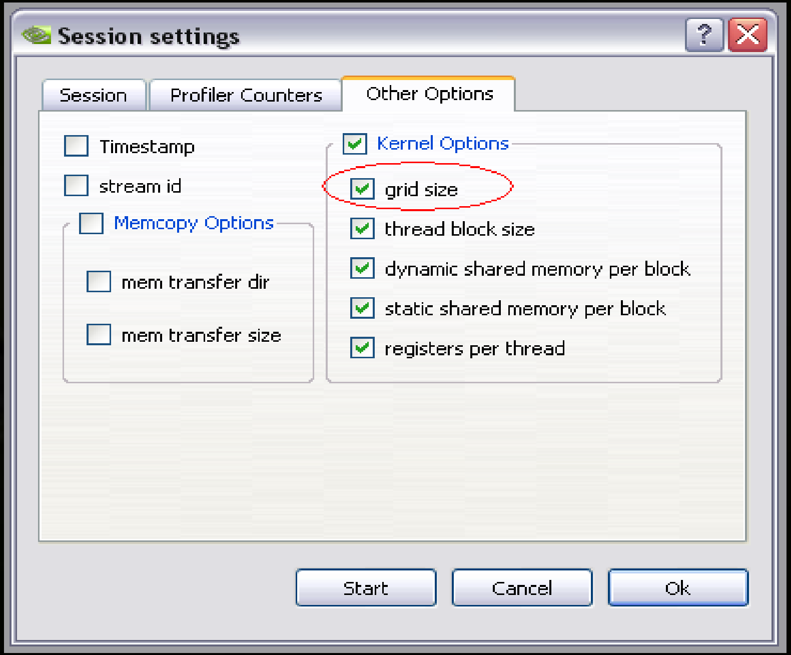
Session settings

Session | Profiler Counters | Other Options

- [ ] Select All Counters
- [ ] gld uncoalesced
- [ ] gld coalesced
- [ ] gst uncoalesced
- [ ] gst coalesced
- [✓] gld_32/64/128b
- [✓] gst_32/64/128b
- [ ] gld_req
- [ ] gst_req

Multiple Run
- [ ] branch
- [ ] divergent branch
- [ ] local load
- [ ] local store
- [ ] instructions
- [ ] warp serialize
- [✓] cta launched

Start | Cancel | Ok



Session settings

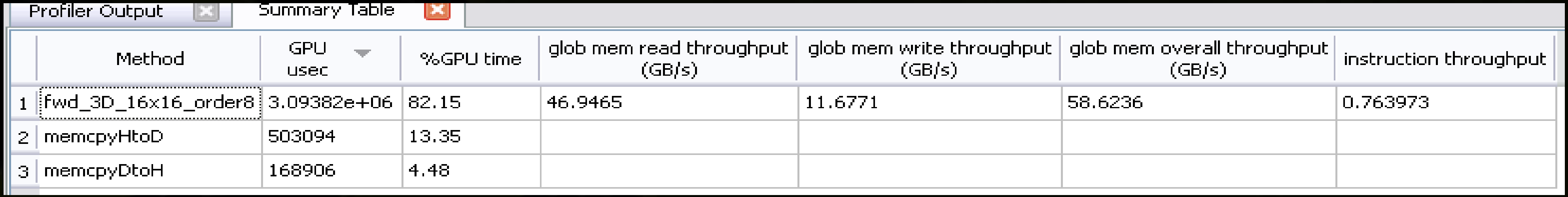
Session | Profiler Counters | Other Options

- [ ] Timestamp
- [ ] stream id
- [ ] Memcopy Options
  - [ ] mem transfer dir
  - [ ] mem transfer size
- [✓] Kernel Options
  - [✓] grid size
  - [✓] thread block size
  - [✓] dynamic shared memory per block
  - [✓] static shared memory per block
  - [✓] registers per thread

Start | Cancel | Ok

Measuring Memory Throughput

- Latest Visual Profiler reports memory throughput
  - From HW point of view
  - Based on counters for one TPC (3 multiprocessors)
  - Need compute capability 1.2 or higher GPU

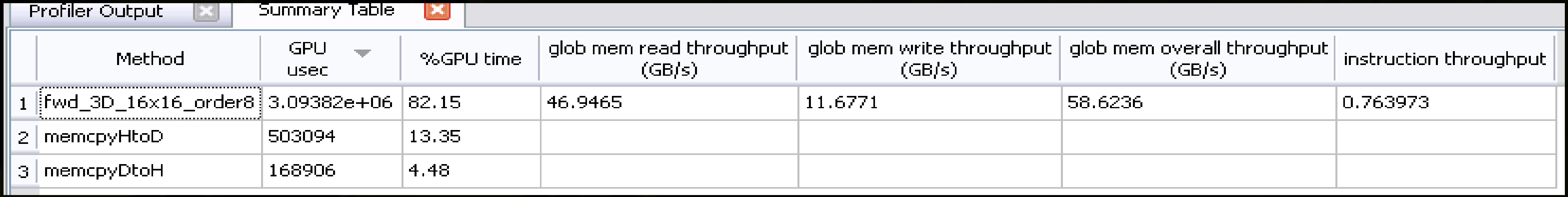


Profiler Output | Summary Table

	Method	GPU usec	%GPU time	glob mem read throughput (GB/s)	glob mem write throughput (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	fwd_3D_16x16_order8	3.09382e+06	82.15	46.9465	11.6771	58.6236	0.763973
2	memcpyHtoD	503094	13.35				
3	memcpyDtoH	168906	4.48				

Measuring Memory Throughput

- Latest Visual Profiler reports memory throughput
  - From HW point of view
  - Based on counters for one TPC (3 multiprocessors)
  - Need compute capability 1.2 or higher GPU



Profiler Output | Summary Table

	Method	GPU usec	%GPU time	glob mem read throughput (GB/s)	glob mem write throughput (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	fwd_3D_16x16_order8	3.09382e+06	82.15	46.9465	11.6771	58.6236	0.763973
2	memcpyHtoD	503094	13.35				
3	memcpyDtoH	168906	4.48				

- How throughput is computed:
  - Count load/store bus transactions of each size (32, 64, 128B) on the TPC
  - Extrapolate from one TPC to the entire GPU
    - Multiply by (total threadblocks / threadblocks on TPC)

(grid size / cta launched)