Advanced CUDA Programming

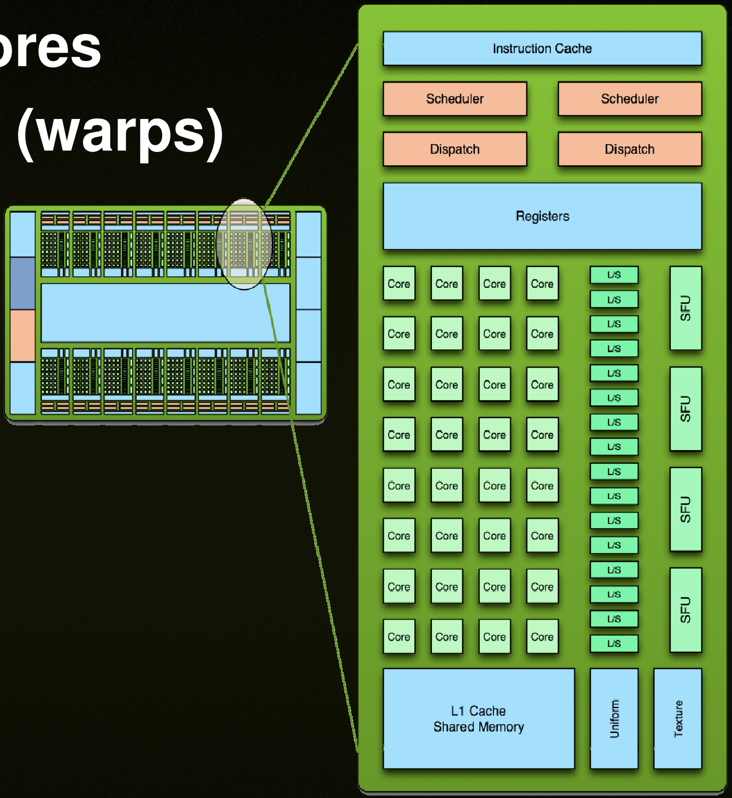
Dr. Timo Stich (tstich@nvidia.com)

Outline

- SIMT Architecture, Warps
- Kernel optimizations
  - Global memory throughput
  - Launch configuration
  - Shared memory access
  - Instruction throughput / control flow
- Optimization of CPU-GPU interaction
  - Maximizing PCIe throughput
  - Overlapping kernel execution with memory copies
- Next-Gen GPUs – The Fermi Architecture

SIMT Architecture

- GPU is structured into MPs of multiple cores
- Threads execute as groups of 32 threads (warps)
  - Threads share same PC
- Threads inside warp may diverge
  - Divergent threads within one warp serialize
- Best Performance when warp executes same code path
  - Note: Threads of warp execute synchronous



Global Memory Throughput

Memory Hierarchy Review

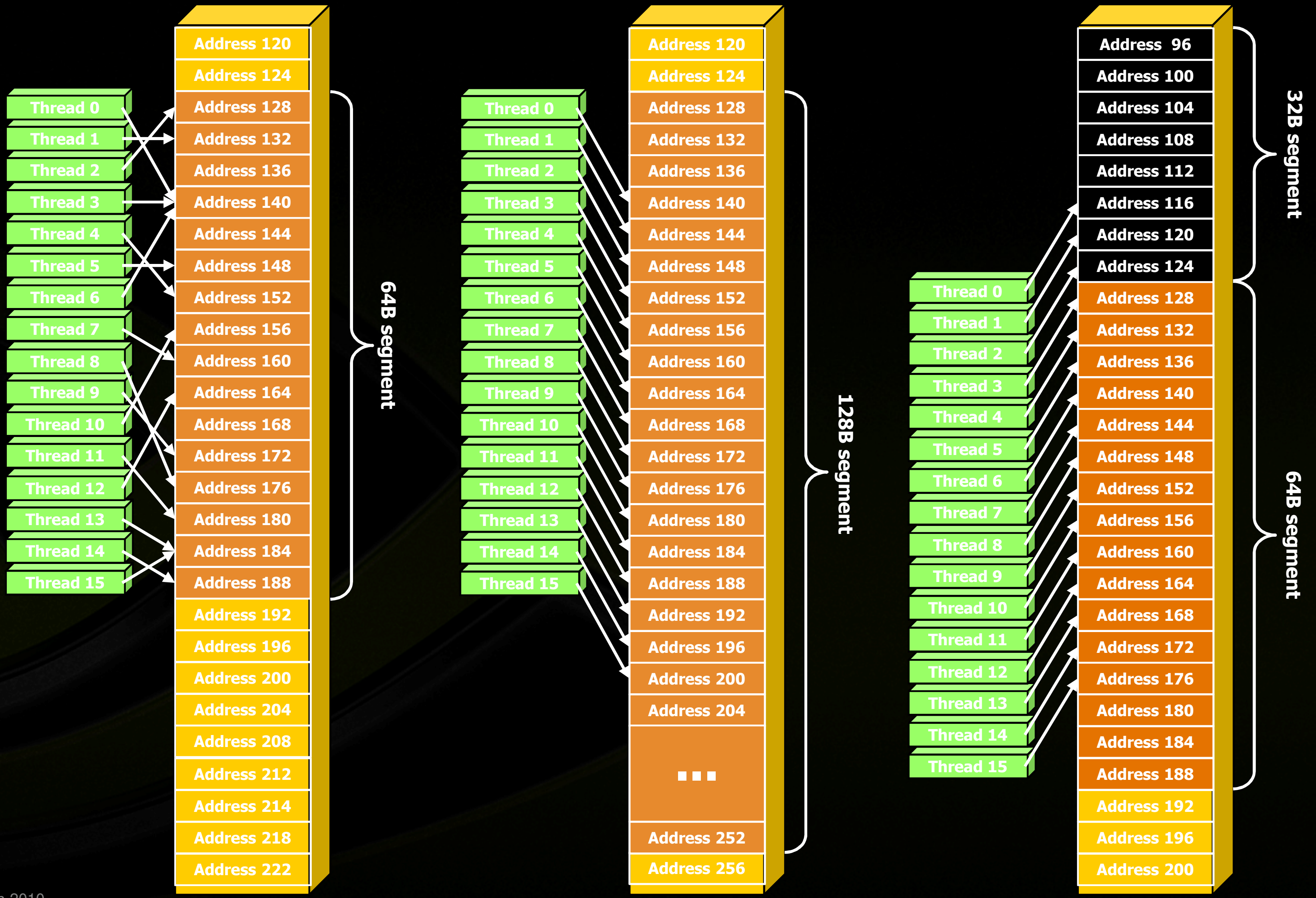
- Local storage
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
- Shared memory
  - Each thread block has its own shared memory
  - Very low latency (a few cycles)
  - Very high throughput: 38-44 GB/s per multiprocessor
    - 30 multiprocessors per GPU -> over 1.1-1.4 TB/s
- Global memory
  - Accessible by all threads as well as host (CPU)
  - High latency (400-800 cycles)
  - Throughput: 140 GB/s (1GB boards), 102 GB/s (4GB boards)

GMEM Coalescing: Compute Capability 1.2, 1.3

- **Possible GPU memory bus transaction sizes:**
  - 32B, 64B, or 128B
  - Transaction segment must be aligned
    - First address = multiple of segment size
- **Hardware coalescing for each half-warp (16 threads):**
  - Memory accesses are handled per half-warps
  - Carry out the smallest possible number of transactions
  - Reduce transaction size when possible

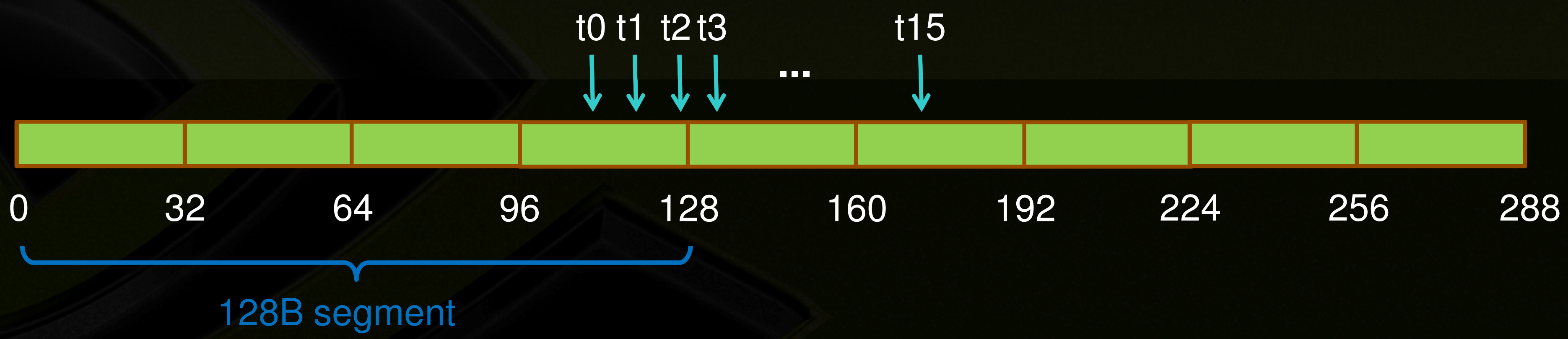
HW Steps when Coalescing for half-warp

- Find the memory segment that contains the address requested by the lowest numbered active thread:
  - 32B segment for 8-bit data
  - 64B segment for 16-bit data
  - 128B segment for 32, 64 and 128-bit data.
- Find all other active threads whose requested address lies in the same segment
- Reduce the transaction size, if possible:
  - If size == 128B and only the lower or upper half is used, reduce transaction to 64B
  - If size == 64B and only the lower or upper half is used, reduce transaction to 32B
    - **Applied even if 64B was a reduction from 128B**
- Carry out the transaction, mark serviced threads as inactive
- Repeat until all threads in the half-warp are serviced


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 96
Address 100
Address 104
Address 108
Address 112
Address 116
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 208
Address 212
Address 214
Address 218
Address 222
Address 252
Address 256
...
64B segment
128B segment
32B segment
64B segment

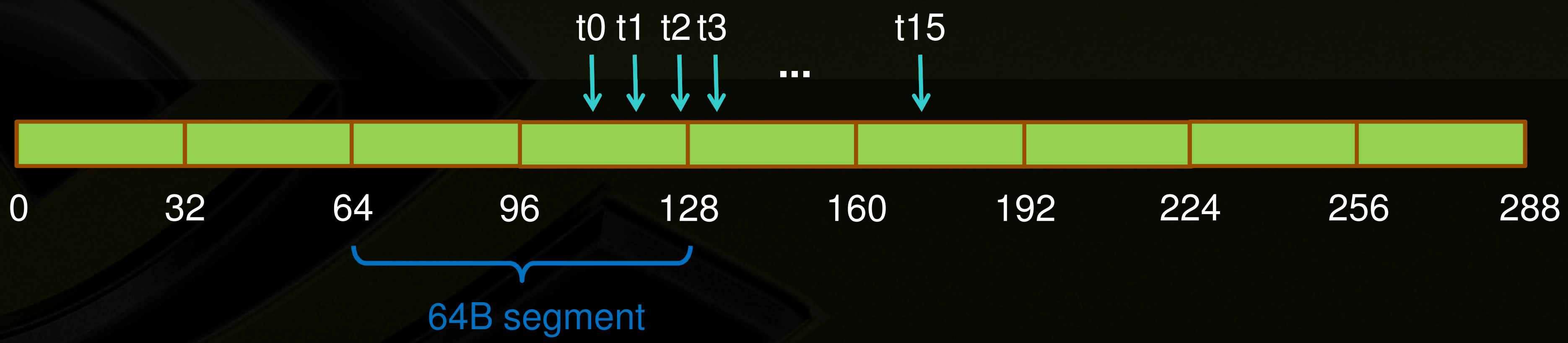
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127



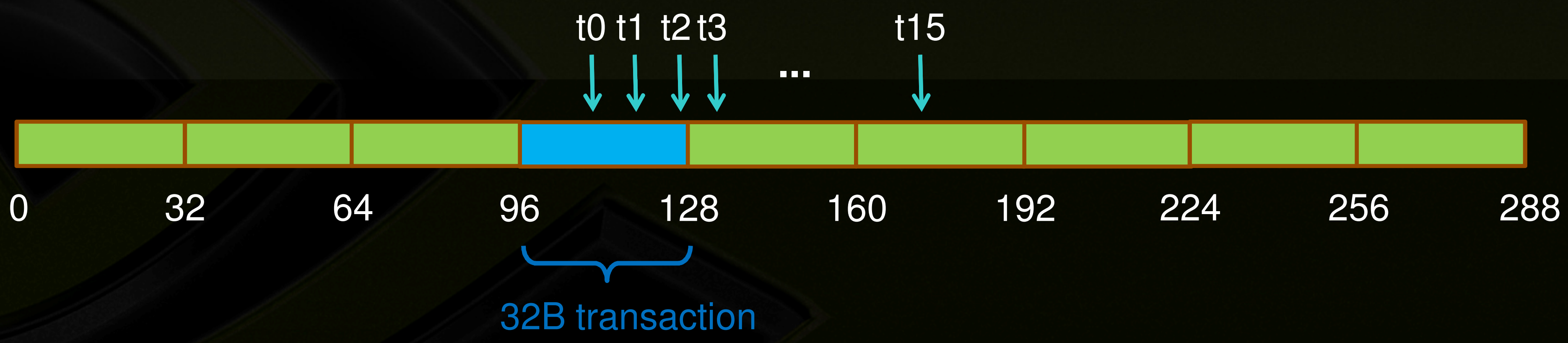
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 64B)



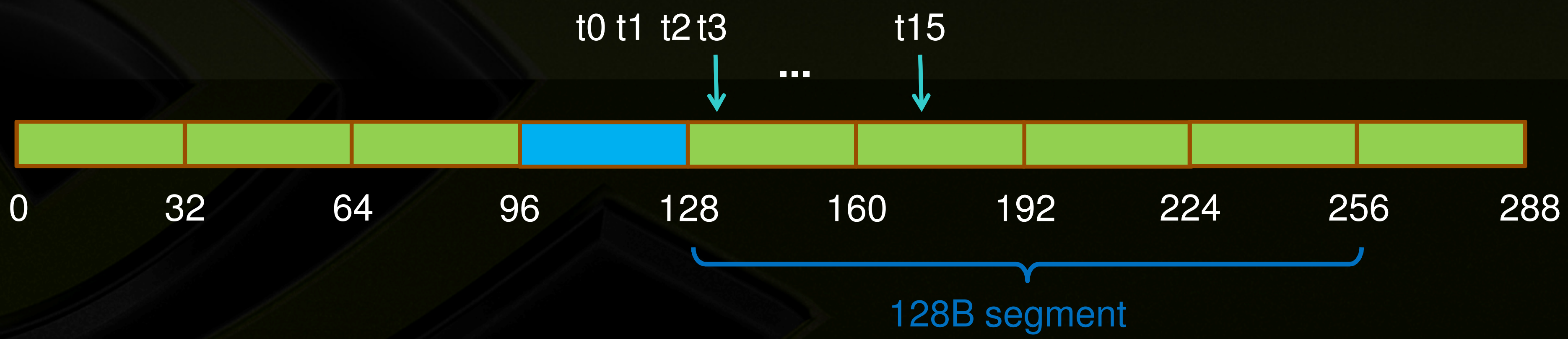
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 32B)



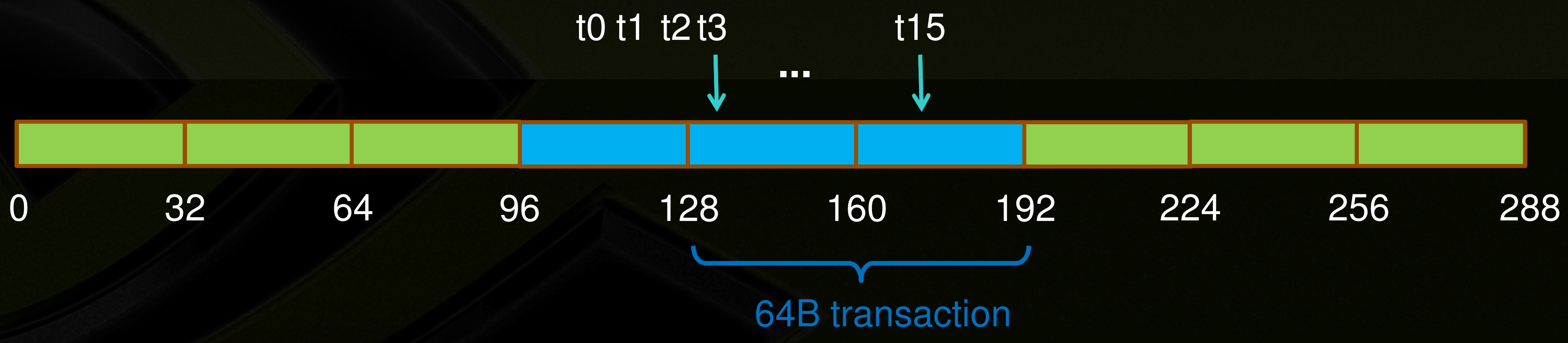
Threads 0-15 access 4-byte words at addresses 116-176

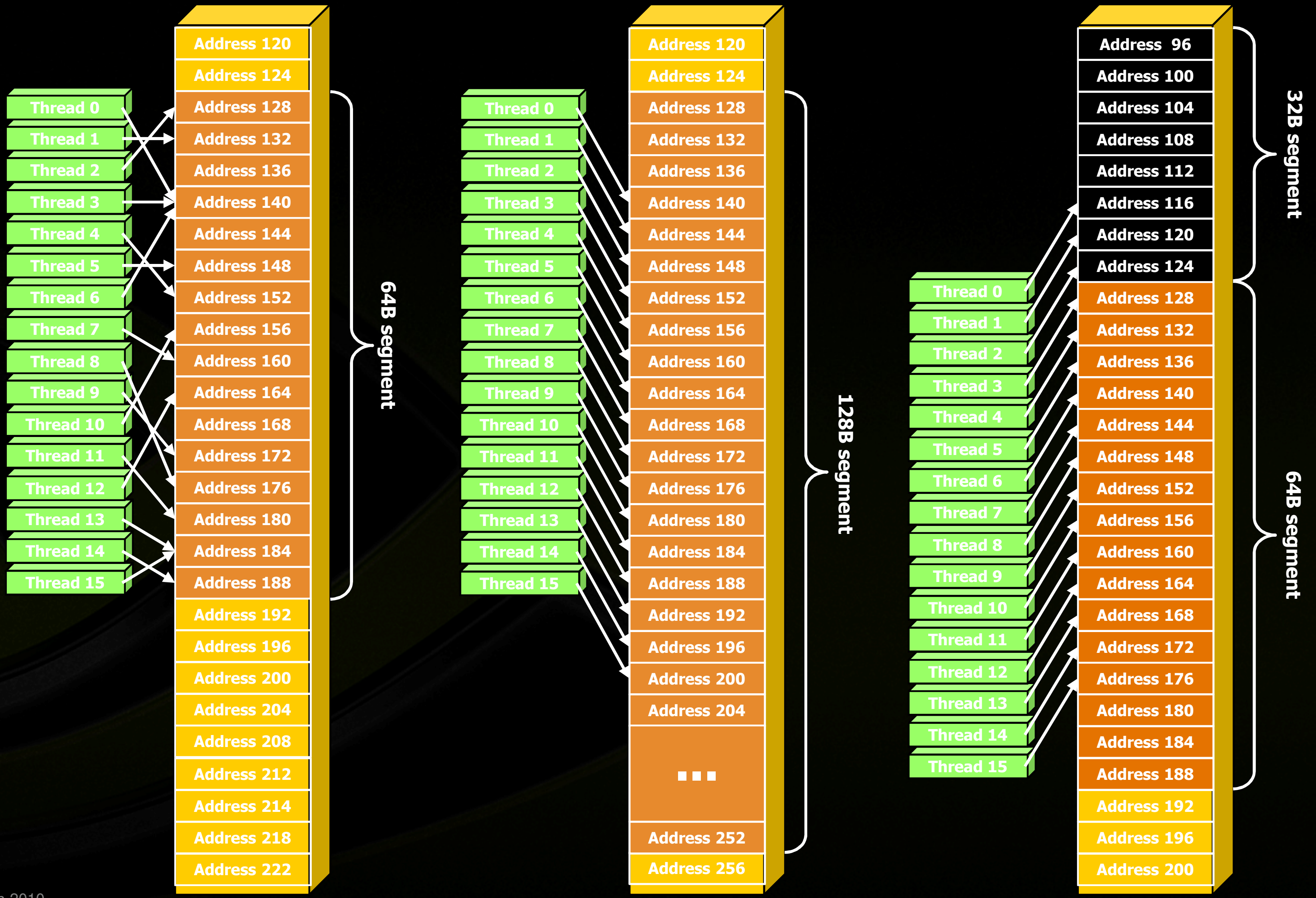
- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255



Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255 (reduce to 64B)




Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 96
Address 100
Address 104
Address 108
Address 112
Address 116
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 208
Address 212
Address 214
Address 218
Address 222
...
Address 252
Address 256
64B segment
128B segment
32B segment
64B segment

Comparing Compute Capabilities

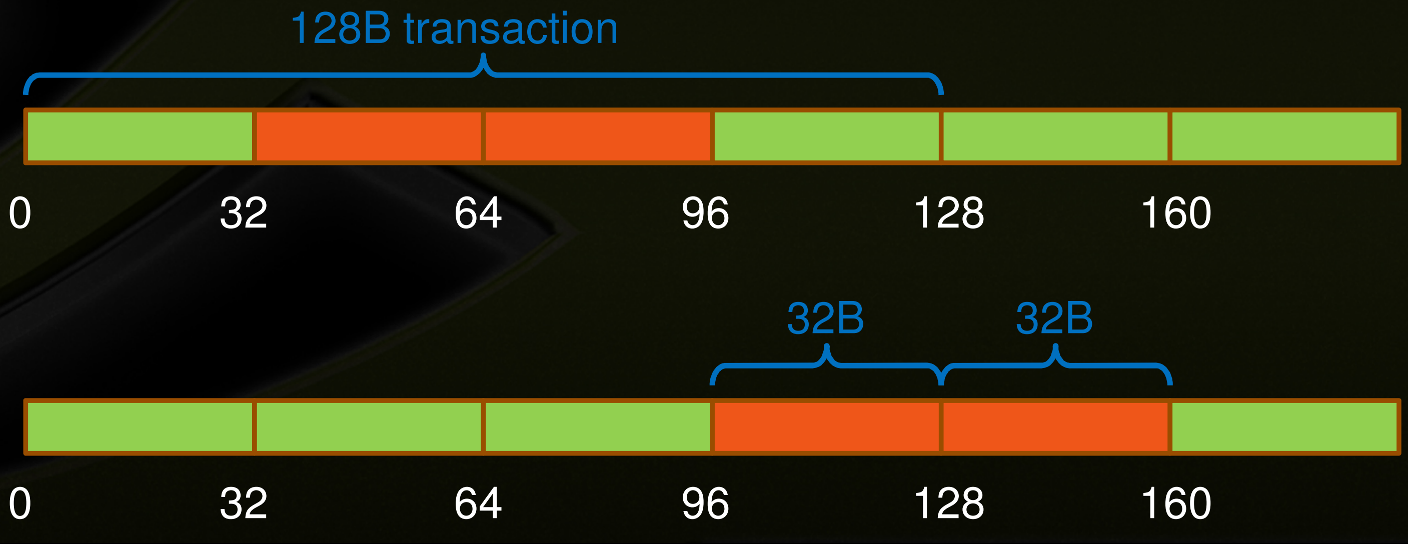
- Compute capability < 1.2
  - Requires threads in a half-warp to:
    - Access a single aligned 64B, 128B, or 256B segment
    - Threads must issue addresses in sequence
  - If requirements are not satisfied:
    - Separate 32B transaction for each thread
- Compute capability 1.2 and 1.3
  - Does not require sequential addressing by threads
  - Perf degrades gracefully when a half-warp addresses multiple segments

Experiment: Impact of Address Alignment

- Assume half-warp accesses a contiguous region
- Throughput is maximized when region is aligned on its size boundary
  - 100% of bytes in a bus transaction are useful
- Impact of misaligned addressing:
  - 32-bit words, streaming code, Quadro FX5800 (102 GB/s)
  - 0 word offset: 76 GB/s (perfect alignment, typical perf)
  - 8 word offset: 57 GB/s (75% of aligned case)
  - All others: 46 GB/s (61% of aligned case)

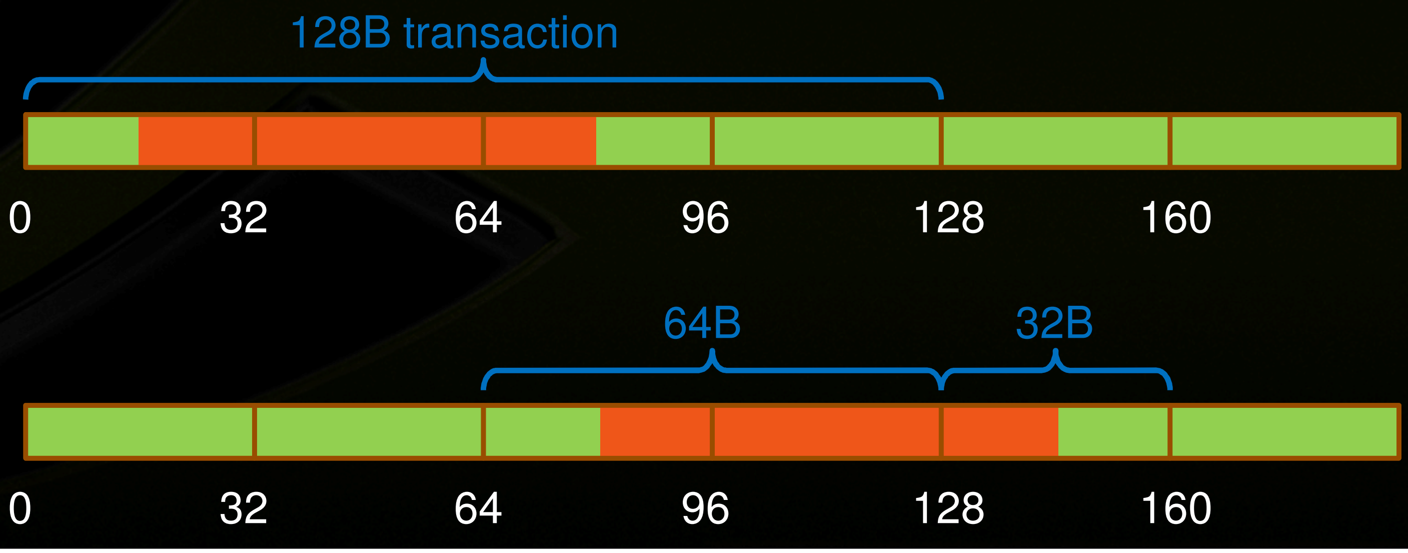
Address Alignment, 32-bit words

- 8-word (32B) offset from perfect alignment:
  - Observed 75% of the perfectly aligned perf
  - Segments starting at multiple of 32B
    - One 128B transaction (50% efficiency)
  - Segments starting at multiple of 96B
    - Two 32B transactions (100% efficiency)



Address Alignment, 32-bit words

- 4-word (16B) offset (other offsets have the same perf):
  - Observed 61% of the perfectly aligned perf
  - Two types of segments, based on starting address
    - One 128B transaction (50% efficiency)
    - One 64B and one 32B transaction (67% efficiency)



Address Alignment, 64-bit words

- Can be analyzed similarly to 32-bit case:
  - 0B offset: 80 GB/s (perfectly aligned)
  - 8B offset: 62 GB/s (78% of perfectly aligned)
  - 16B offset: 62 GB/s (78% of perfectly aligned)
  - 32B offset: 68 GB/s (85% of perfectly aligned)
  - 64B offset: 76 GB/s (95% of perfectly aligned)
- Compare 0 and 64B offset performance:
  - Both consume 100% of the bytes
    - 64B: two 64B transactions
    - 0B: a single 128B transaction, slightly faster

GMEM Optimization Guidelines

- Strive for perfect coalescing
  - Align starting address (may require padding)
  - Warp should access within contiguous region
- Process several elements per thread
  - Multiple loads get pipelined
  - Indexing calculations can often be reused
- Launch enough threads to cover access latency
  - GMEM accesses are not cached
  - Latency is hidden by switching threads (warps)

Launch Configuration

Launch Configuration

- How many threads/threadblocks to launch?
- Key to understanding:
  - Instructions are issued in order
  - A thread blocks when one of the operands isn't ready:
    - Memory read doesn't block
  - Latency is hidden by switching threads
    - GMEM latency is 400-800 cycles
- Conclusion:
  - Need enough threads to hide latency

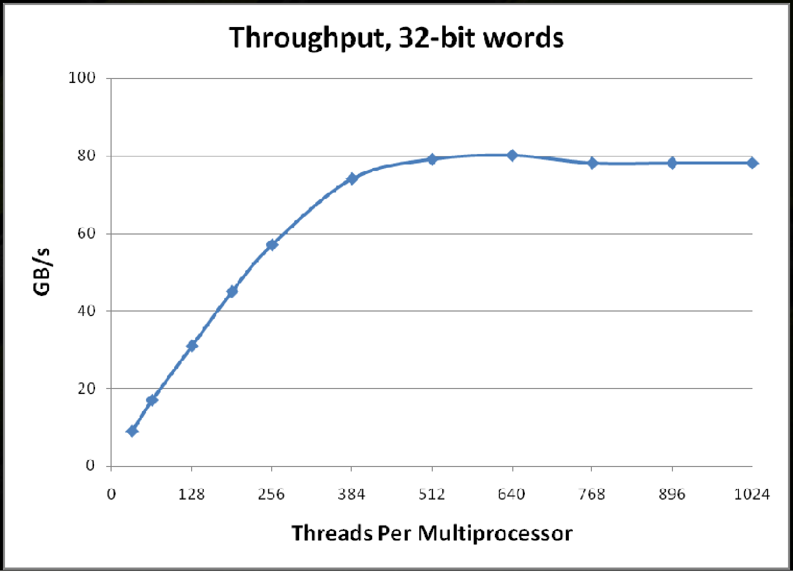
Hiding Latency

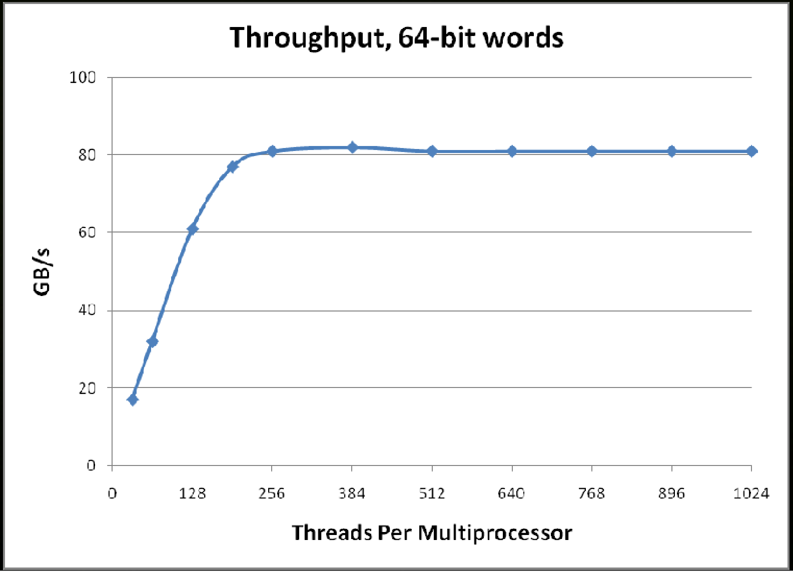
- **Arithmetic:**
  - Need at least 6 warps (192) threads per SM
- **Memory:**
  - Depends on the access pattern
  - For GT200, 50% occupancy (512 threads per SM) is often sufficient
    - Occupancy = fraction of the maximum number of threads per multiprocessor

Hiding Latency

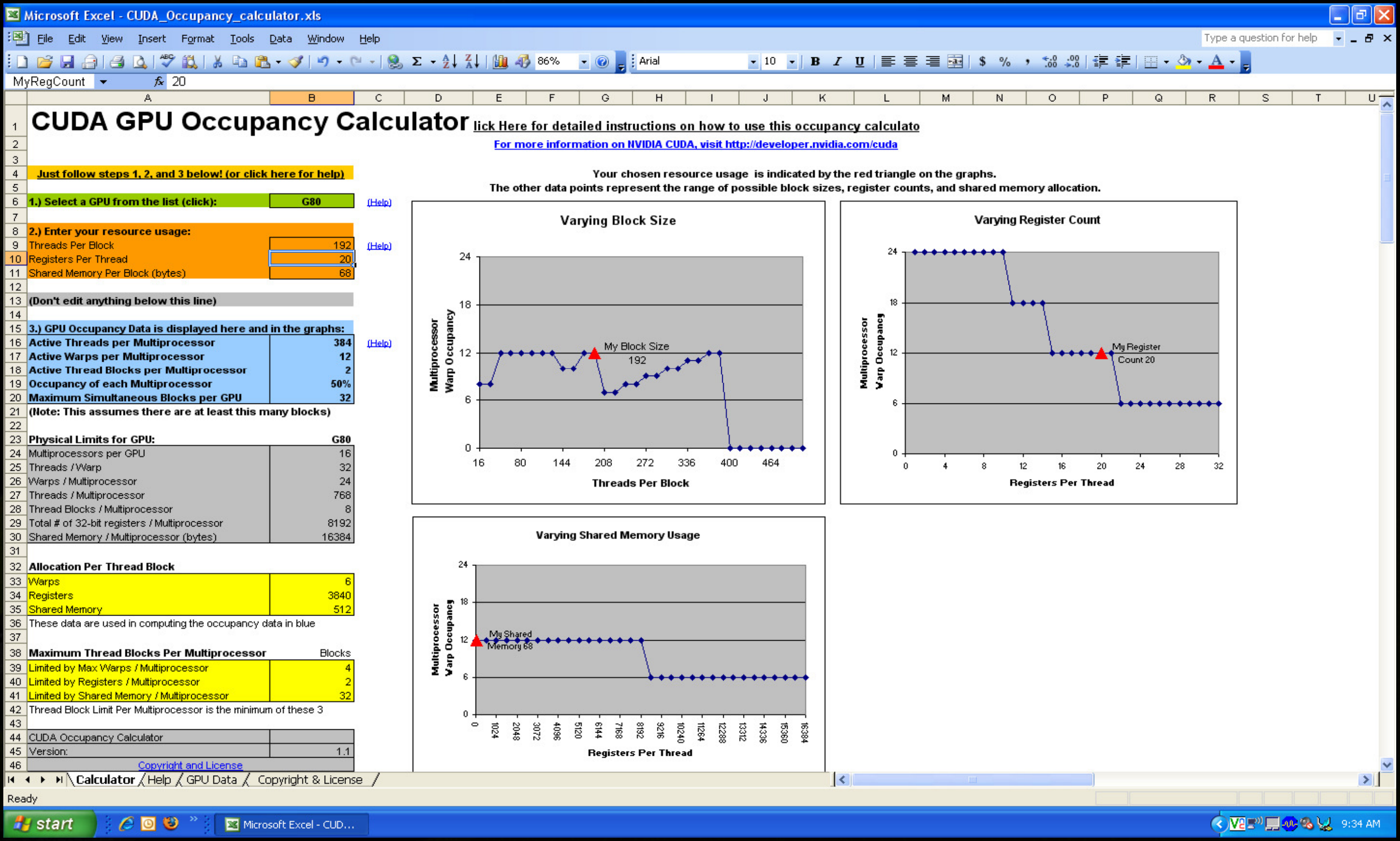
- **Arithmetic:**
  - Need at least 6 warps (192) threads per SM
- **Memory:**
  - Depends on the access pattern
  - For GT200, 50% occupancy (512 threads per SM) is often sufficient
    - Occupancy = fraction of the maximum number of threads per multiprocessor

Streaming 16M words: each thread reads, increments, writes 1 element





Occupancy Calculator



Microsoft Excel - CUDA_Occupancy_calculator.xls

MyRegCount | 20

CUDA GPU Occupancy Calculator lick Here for detailed instructions on how to use this occupancy calculato

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

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

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

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

Launch Configuration: Summary

- Need enough total threads to keep GPU busy
  - Currently (GT200), 512+ threads per SM is ideal
  - Fewer than 192 threads per SM WILL NOT hide arithmetic latency
- Threadblock configuration
  - Threads per block should be a multiple of warp size (32)
  - SM can concurrently execute up to 8 threadblocks
    - Really small threadblocks prevent achieving good occupancy
    - Really large threadblocks are less flexible
    - I generally use 128-256 threads/block, but use whatever is best for the application

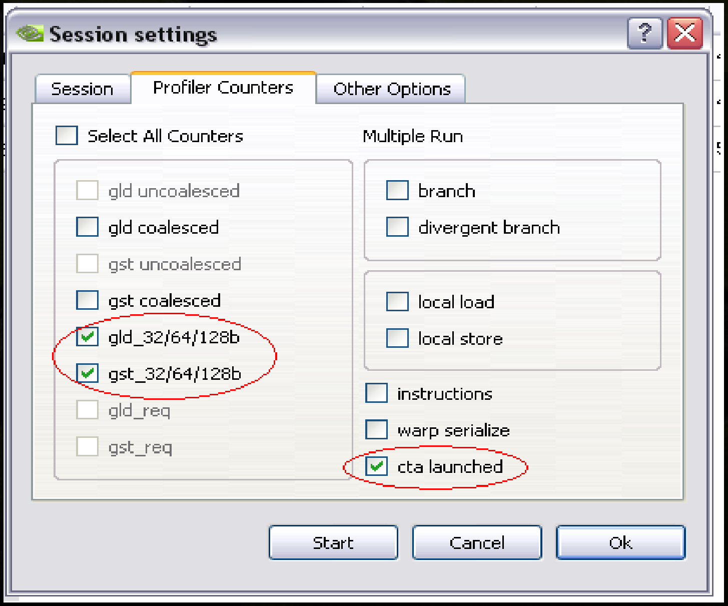
Memory Throughput as Performance Metric

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

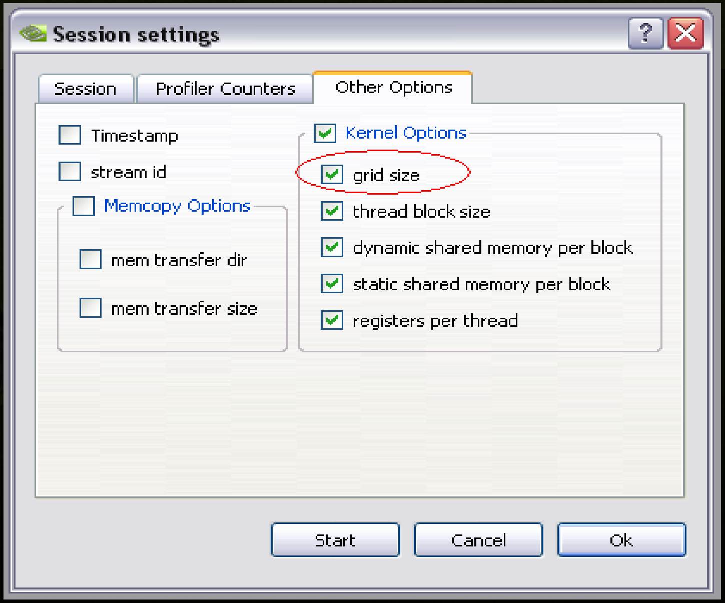
Session | Profiler Counters | Other Options

- [ ] 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

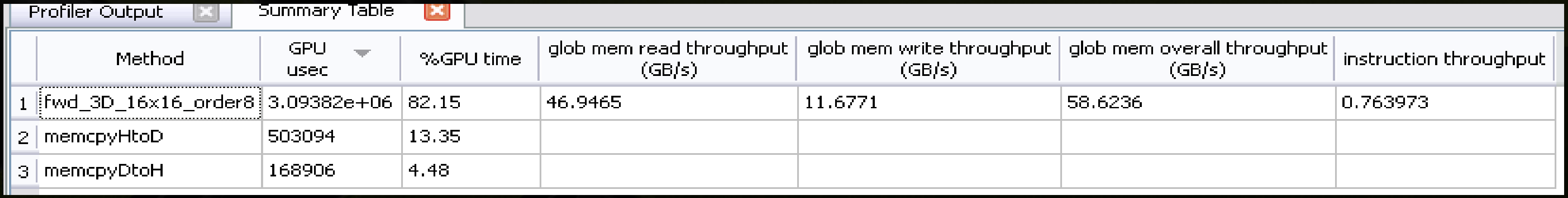
Session | Profiler Counters | Other Options

- [ ] 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

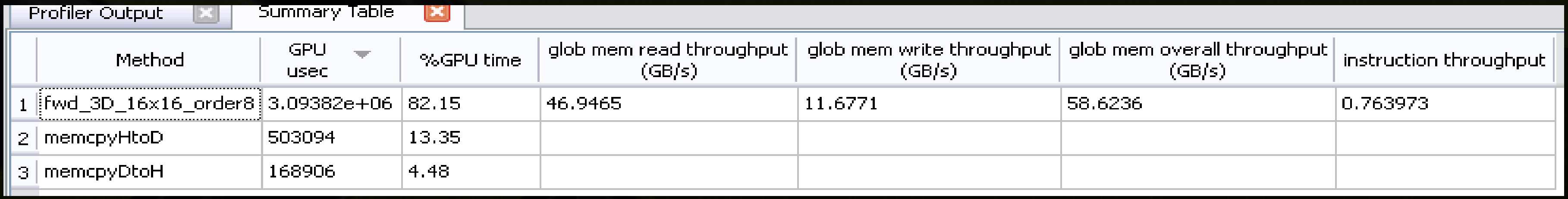
- Latest Visual Profiler reports memory throughput
  - From HW point of view
  - Based on counters for one TPC (3 multiprocessors)
  - Need compute capability 1.2 or higher GPU



	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



	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)

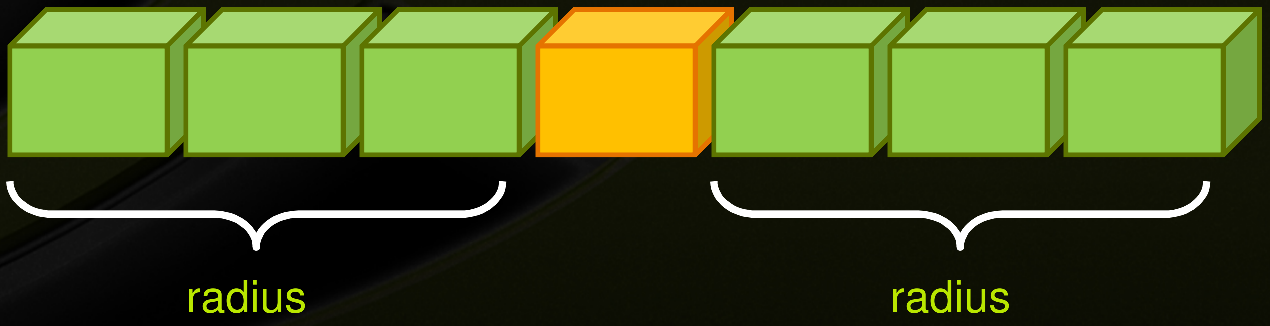
Shared Memory

Shared Memory

- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce global memory accesses
  - Use it to avoid non-coalesced access
- Organization:
  - 16 banks, 32-bit wide banks
  - Successive 32-bit words belong to different banks
- Performance:
  - 32 bits per bank per 2 clocks per multiprocessor
  - smem accesses are per 16-threads (half-warp)
  - serialization: if *n* threads (out of 16) access the same bank, *n* accesses are executed serially
  - broadcast: *n* threads access the same word in one fetch

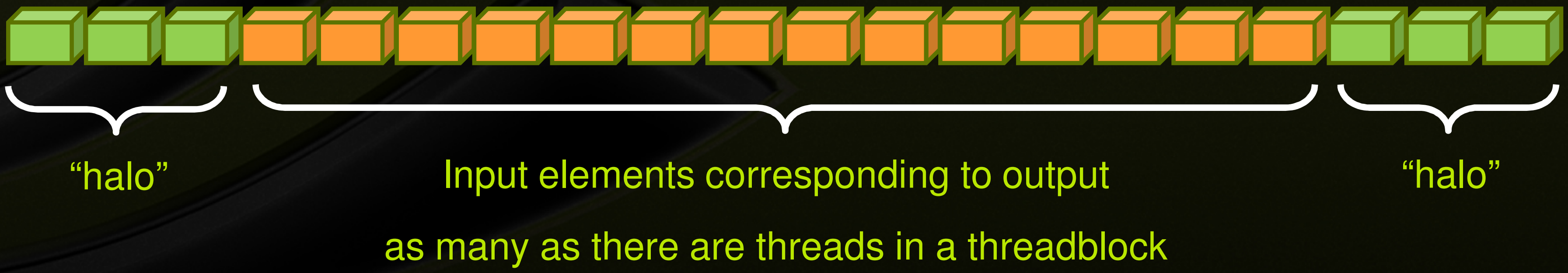
Example of Using Shared Memory

- Applying a 1D stencil:
  - 1D data
  - For each output element, sum all elements within a radius
- For example, radius = 3
  - Add 7 input elements

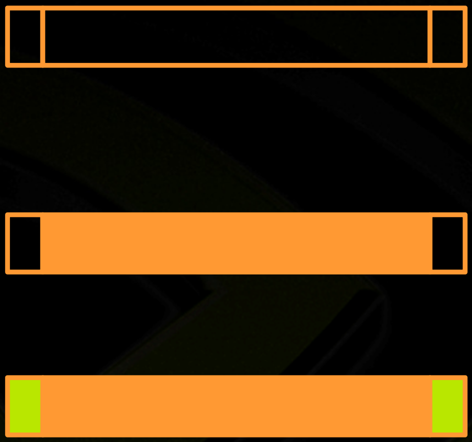


Implementation with Shared Memory

- 1D threadblocks (partition the output)
- Each threadblock outputs BLOCK_DIMX elements
  - Read input from gmem to smem
    - Needs BLOCK_DIMX + 2*RADIUS input elements
  - Compute
  - Write output to gmem


"halo"
Input elements corresponding to output
as many as there are threads in a threadblock
"halo"

Kernel code



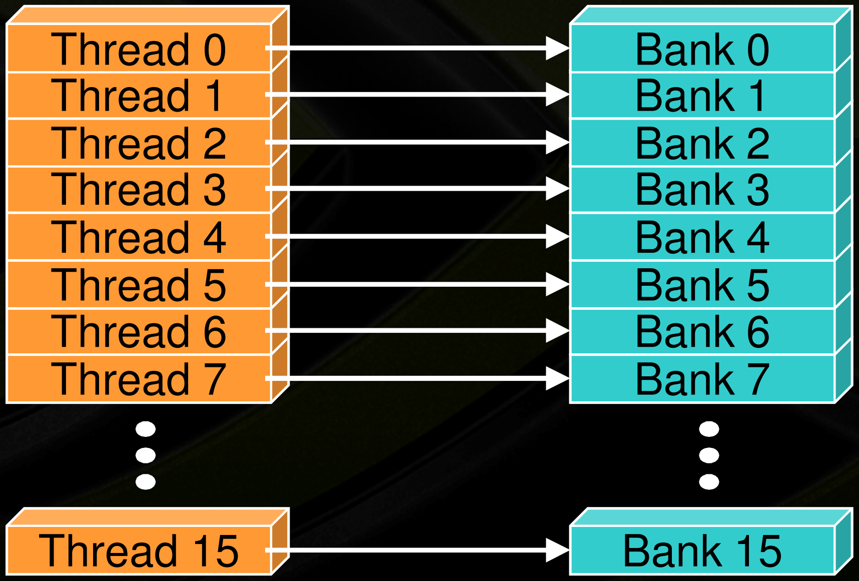
```
__global__ void stencil( int *output, int *input, int dimx, int dimy )
{
    __shared__ int s_a[BLOCK_DIMX+2*RADIUS];

    int global_ix = blockIdx.x*blockDim.x + threadIdx.x;
    int local_ix   = threadIdx.x + RADIUS;

    s_a[local_ix] = input[global_ix];

    if ( threadIdx.x < RADIUS ) {
        s_a[local_ix – RADIUS]                         = input[global_ix – RADIUS];
        s_a[local_ix + BLOCK_DIMX + RADIUS] = input[global_ix + RADIUS];
    }
    __syncthreads();

    int value = 0;
    for( offset = -RADIUS; offset<=RADIUS; offset++ )
        value += s_a[ local_ix + offset ];

     output[global_ix] = value;
}
```

Thread Synchronization Function

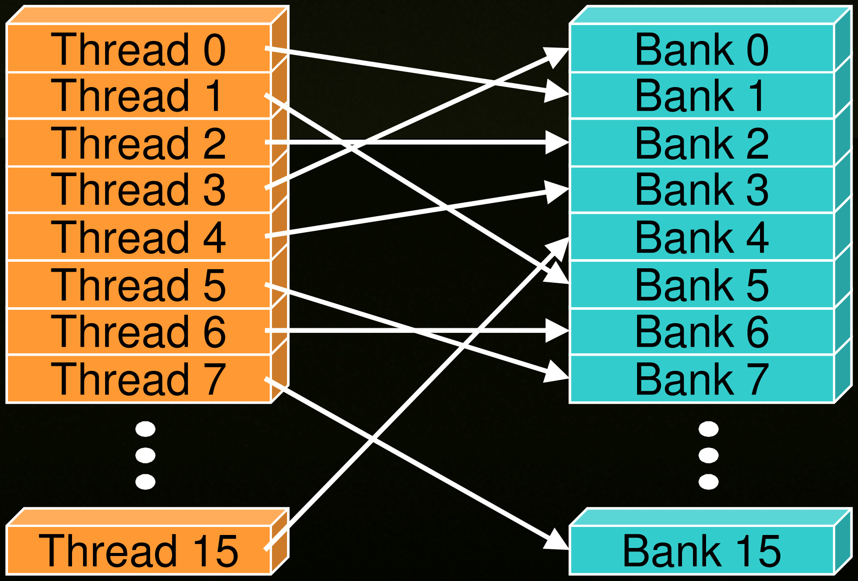
- void __syncthreads(void)
- Synchronizes all threads in a threadblock
  - Since threads are scheduled at run-time
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block

Bank Addressing Examples

- No Bank Conflicts

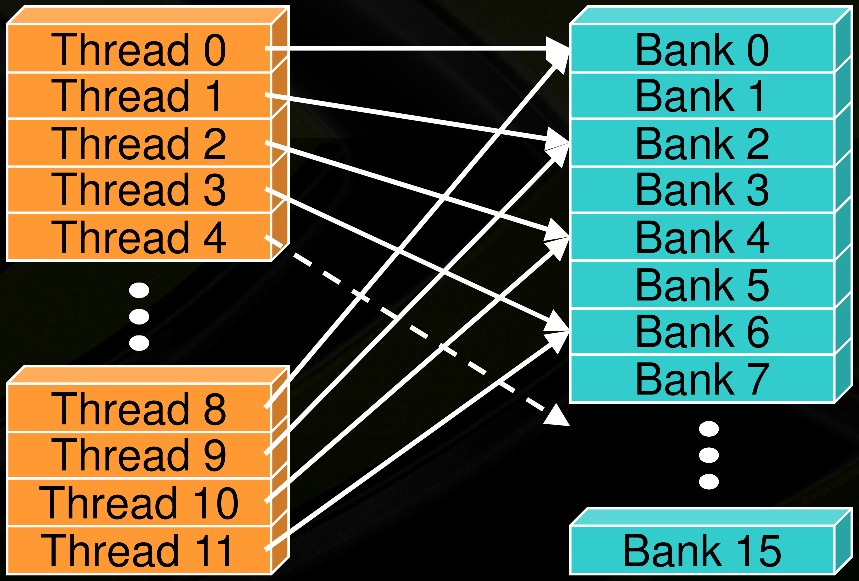


- No Bank Conflicts

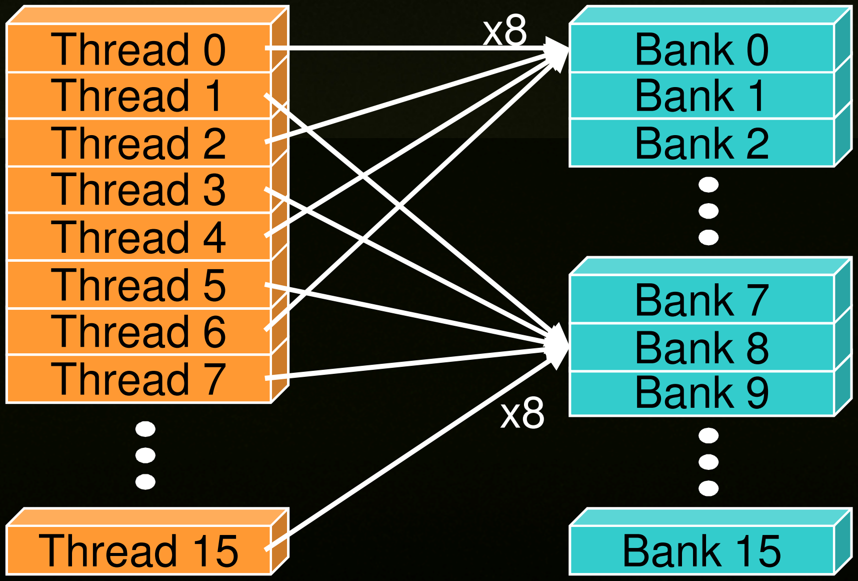


Bank Addressing Examples

- **2-way Bank Conflicts**



- **8-way Bank Conflicts**



Trick to Assess Impact On Performance

- Change all SMEM reads to the same value
    - All broadcasts = no conflicts
    - Will show how much performance could be improved by eliminating bank conflicts
- The same doesn’t work for SMEM writes
    - So, replace SMEM array indices with `threadIdx.x`
    - Can also be done to the reads

Additional “memories”

- *Texture* and *constant*
- Read-only
- Data resides in global memory
- Different read path:
  - includes caches (unlike current global memory access)

Constant Memory

- Data stored in global memory, read through a constant-cache path
  - __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

Instruction Throughput / Control Flow

Runtime Math Library and Intrinsics

- Two types of runtime math library functions
  - __func(): many map directly to hardware ISA
    - Fast but lower accuracy (see CUDA Programming Guide for full details)
    - Examples: __sinf(x), __expf(x), __powf(x, y)
  - func(): compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x, y)
- A number of additional intrinsics:
  - __sincosf(), __frcp_rz(), ...
  - Explicit IEEE rounding modes (rz,rn,ru,rd)

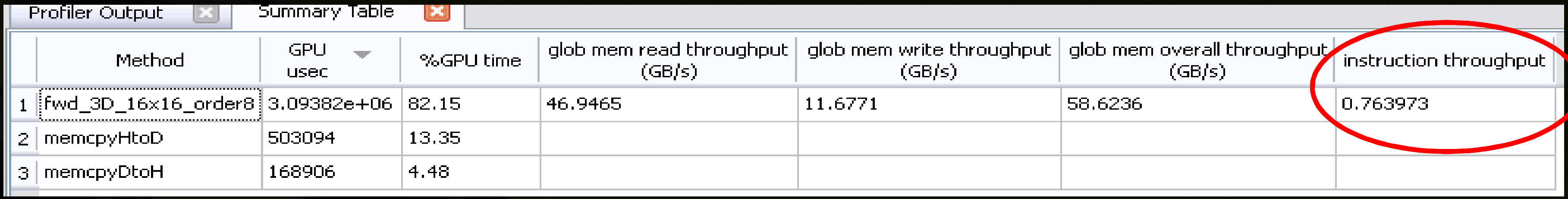
Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - `if-else`, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance
- Avoid diverging within a warp
  - Example with divergence:
    - `if (threadIdx.x > 2) {...} else {...}`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

Profiler and Instruction Throughput

- Profiler counts per multiprocessor:
  - Divergent branches
  - Warp serialization
  - Instructions issues
- Visual Profiler derives:
  - Instruction throughput
    - Fraction of fp32 arithmetic instructions that could have been issued in the same amount of time
      - So, not a good metric for code with fp64 arithmetic or transcendentals
  - Extrapolated from one multiprocessor to GPU

Profiler and Instruction Throughput



	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				

- **Visual Profiler derives:**
  - **Instruction throughput**
    - **Fraction of fp32 arithmetic instructions that could have been issued in the same amount of time**
      - So, not a good metric for code with fp64 arithmetic or transcendentals
  - **Extrapolated from one multiprocessor to GPU**

Tricks with Code Comments

- Comment out arithmetic
  - To assess memory-only performance
  - Fine as long as memory access is not data-dependent
- Comment out gmem accesses
  - To assess arithmetic-only performance
  - Fine as long as computation is not data dependent
  - Eliminating reads is straightforward
  - Eliminating writes is trickier
    - Compiler will throw away all code it deems as not contributing to output
    - Workaround: precede writes with an if-statement that always fails
      - For example: `if( threadIdx.x == -2 )`

CPU-GPU Interaction

Pinned (non-pageable) memory

- Pinned memory enables:
  - faster PCIe copies (~6 GB/s with PCIe 2.0)
  - memcopies asynchronous with CPU
  - memcopies asynchronous with GPU
- Usage
  - cudaHostAlloc / cudaFreeHost
    - instead of malloc / free
- Implication:
  - pinned memory is essentially removed from host virtual memory

Streams and Async API

- Default API:
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- Streams and async functions provide:
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
- Stream = sequence of operations that execute in issue-order on GPU
  - Operations from different streams can be interleaved
  - A kernel and memcopy from different streams can be overlapped

Overlap kernel and memory copy

- **Requirements:**
  - **D2H or H2D memcopy from pinned memory**
  - **Device with compute capability ≥ 1.1 (G84 and later)**
  - **Kernel and memcopy in different, non-0 streams**
- **Code:**

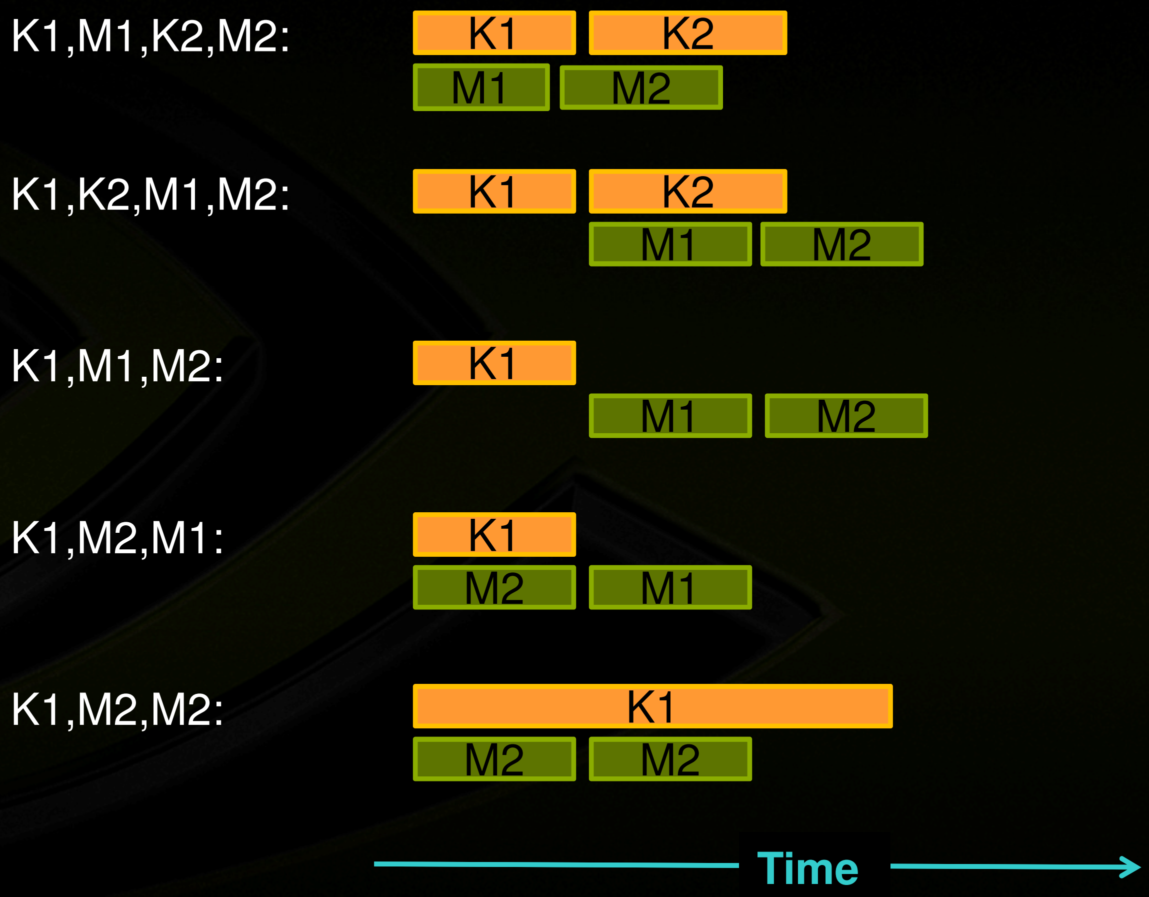
```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

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

$\}$ **potentially overlapped**

Call Sequencing for Optimal Overlap

- CUDA calls are dispatched to the hw in the sequence they were issued
- One kernel and one memcopy can be executed concurrently
- A call is dispatched if both are true:
  - Resources are available
  - Preceding calls in the same stream have completed
- Note that if a call blocks, it blocks all other calls of the same type behind it, even in other streams
  - Type is one of { kernel, memcopy}

Stream Examples (current HW)



Copy-Compute Overlap with Zero-Copy

- Pinned host memory can be mapped into GPU memory space
  - Read/Write via GPU translates to PCIe traffic
  - Increased latency can still be hidden if enough threads available
- Finer grained overlap
  - Overlap on the warp level rather than kernel level
  - Coalesced memory access is even more critical!
- Suited to transfer small amounts of memory
  - Use-case: Iterative solver, 1 kernel/iteration, 1 kernel to check convergence and write back result via mapped host memory

Summary

- GPU-CPU interaction:
  - Minimize CPU/GPU idling, maximize PCIe throughput
- Global memory:
  - Maximize throughput (GPU has lots of bandwidth, use it effectively)
- Kernel Launch Configuration:
  - Launch enough threads per SM to hide latency
  - Launch enough threadblocks to load the GPU

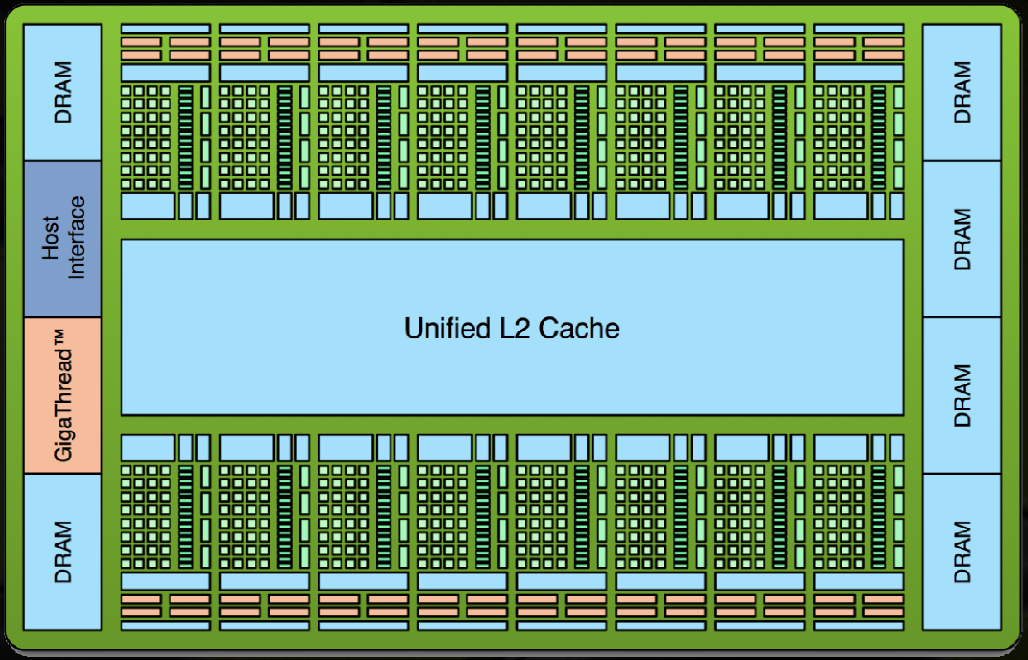
Summary

- GPU-CPU interaction:
  - Minimize CPU/GPU idling, maximize PCIe throughput
- Global memory:
  - Maximize throughput (GPU has lots of bandwidth, use it effectively)
- Kernel Launch Configuration:
  - Launch enough threads per SM to hide latency
  - Launch enough threadblocks to load the GPU
- Measure!
  - Use the Visual Profiler or Parallel Nsight, simple code modifications
  - Compare to theoretical peaks

Fermi

NEXT GENERATION ARCHITECTURE

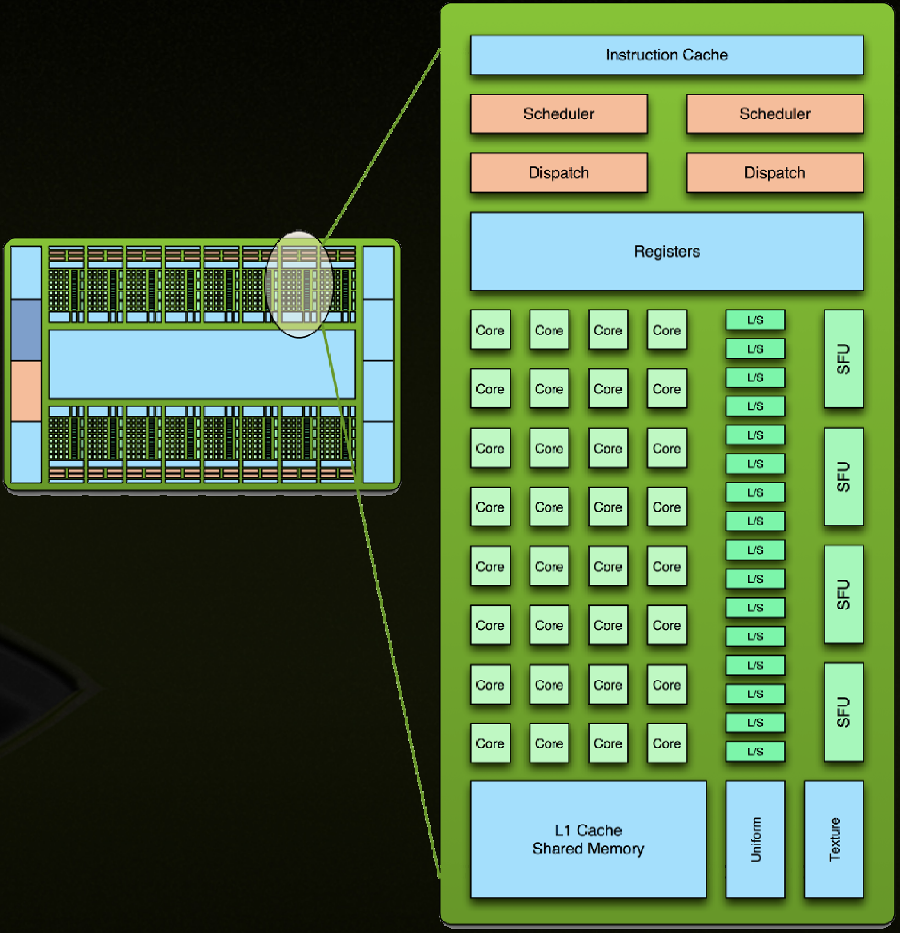
Introducing the Fermi Architecture



- 3 billion transistors
- 512 cores
- DP performance 50% of SP
- ECC
- L1 and L2 Caches
- GDDR5 Memory
- Up to 1 Terabyte of GPU Memory
- Concurrent Kernels, C++

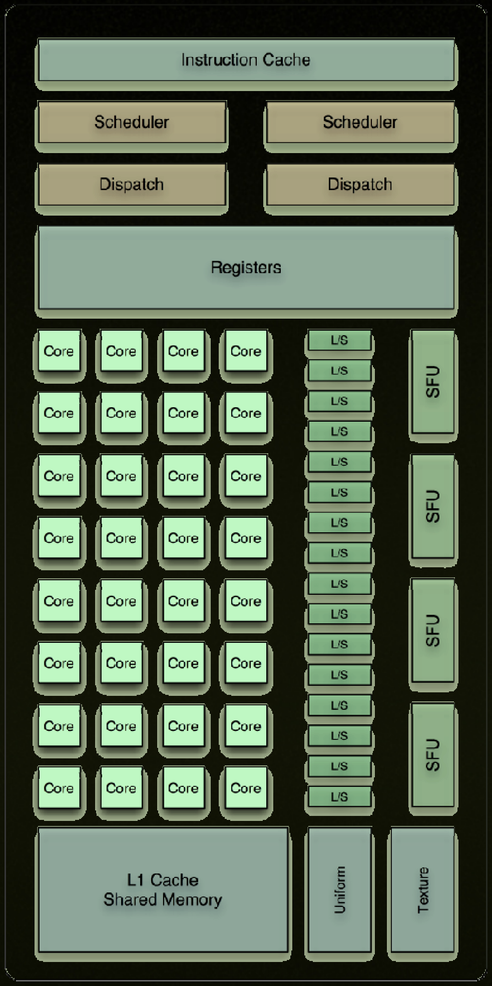
Fermi SM Architecture

- 32 CUDA cores per SM
- Double precision 50% of single precision
  - 8x over GT200
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)



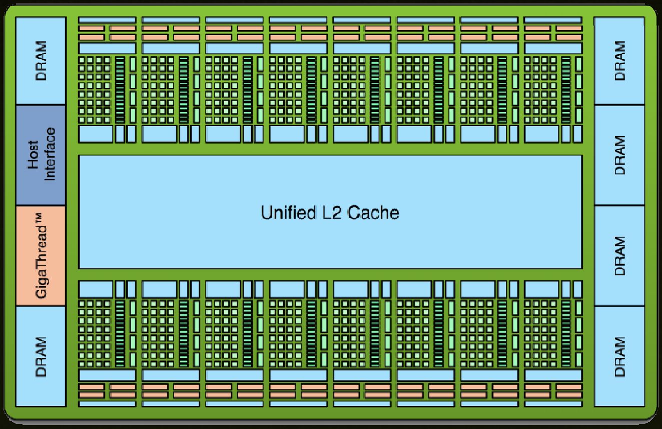
CUDA Core Architecture

- New IEEE 754-2008 floating-point standard, surpassing even the most advanced CPUs
- Fused multiply-add (FMA) instruction for both single and double precision
- Newly designed integer ALU optimized for 64-bit and extended precision operations



Cached Memory Hierarchy

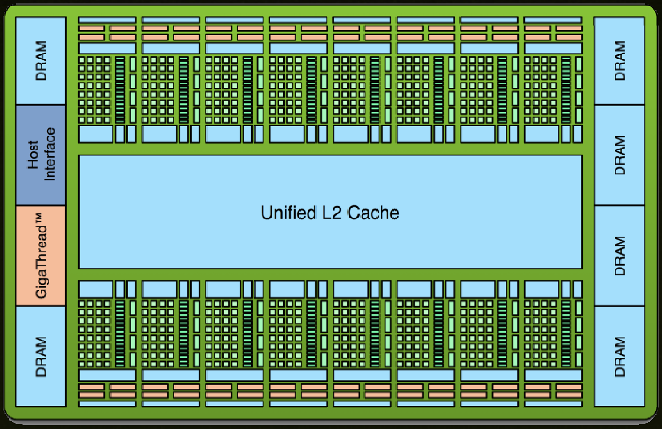
- First GPU architecture to support a true cache hierarchy in combination with on-chip shared memory
- L1 Cache per SM (per 32 cores)
  - Improves bandwidth and reduces latency
- Unified L2 Cache (768 KB)
  - Fast, coherent data sharing across all cores in the GPU



Parallel DataCache™ Memory Hierarchy

Larger, Faster Memory Interface

- GDDR5 memory interface
  - 2x speed of GDDR3
- Up to 1 Terabyte of memory attached to GPU
  - Operate on large data sets

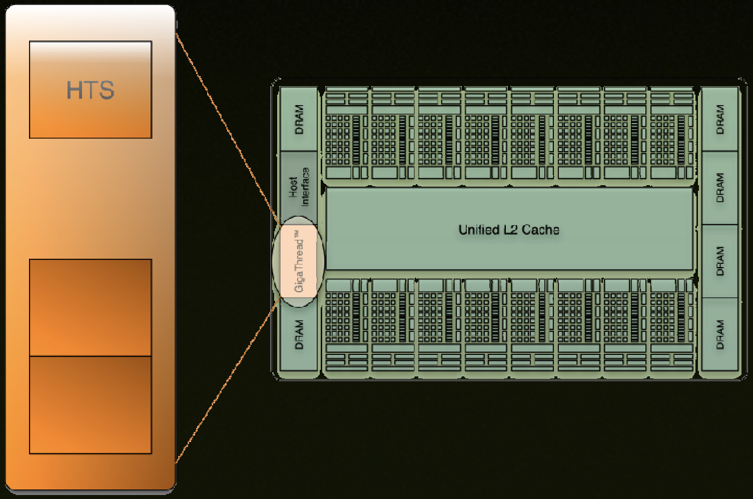


ECC

- ECC protection for
  - DRAM
    - ECC supported for GDDR5 memory
  - All major internal memories
    - Register file, L1 cache, L2 cache
- Detect 2-bit errors, correct 1-bit errors (per word)

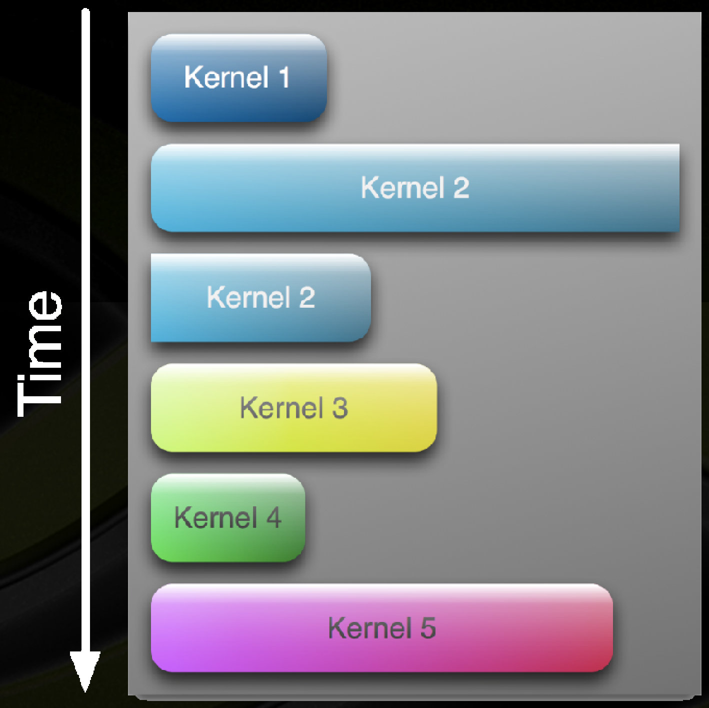
GigaThread™ Hardware Thread Scheduler

- Hierarchically manages thousands of simultaneously active threads
- 10x faster application context switching
- Concurrent kernel execution

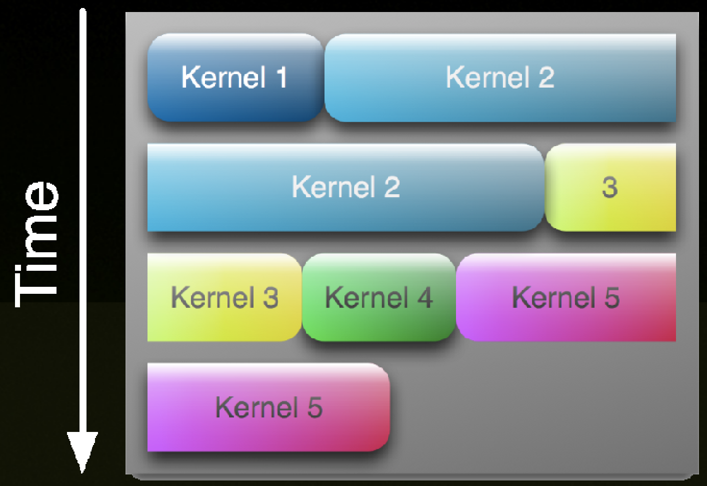


GigaThread™ Hardware Thread Scheduler

Concurrent Kernel Execution + Faster Context Switch



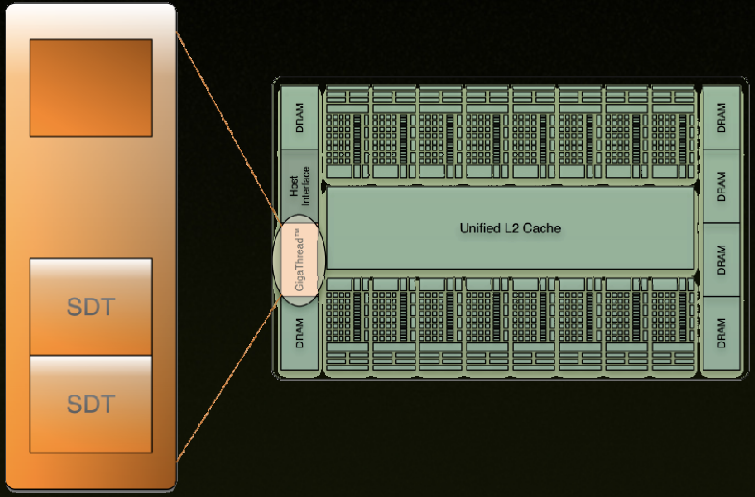
Serial Kernel Execution



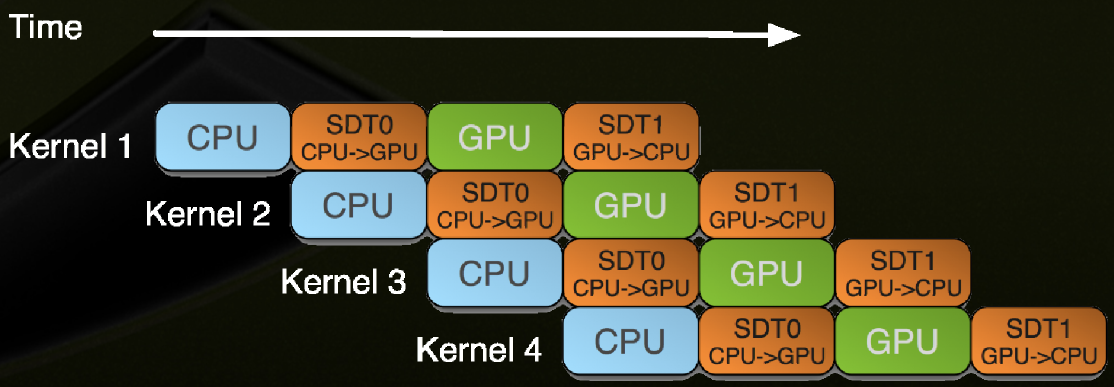
Parallel Kernel Execution

GigaThread Streaming Data Transfer Engine

- Dual DMA engines
  - Simultaneous CPU→GPU and GPU→CPU data transfer
  - Fully overlapped with CPU and GPU processing time



- Activity Snapshot:



Enhanced Software Support

- Full C++ Support
  - Virtual functions
  - Try/Catch hardware support
- System call support
  - Support for pipes, semaphores, printf, etc
- Unified 64-bit memory addressing

Changes for Fermi from optimization POV

- Memory operations are done per warp (32 threads)
  - Global memory, Shared memory
- Shared memory:
  - 16 or 48KB
  - Now 32 banks, 32-bit wide each
- L1 cache per multiprocessor
  - Helps with misaligned access, strides access, some register spilling
- Much improved dual-issue:
  - Can dual issue fp32 pairs, fp32-mem, fp64-mem, etc.
- Concurrent kernel execution
  - Multiple Kernels can execute simultaneously