GPU Profiling and Optimization

Scott Grauer-Gray

Benefits of GPU Programming

- "Free" speedup with new architectures
  - More cores in new architecture
  - Improved features such as L1 and L2 cache
  - Increased shared/local memory space

Benefits of GPU Programming

- GPU program performance likely to improve on new architecture w/ no program adjustment
  - Used to be the case for single-threaded programs on CPUs
  - No longer true there since they moved to multi-core design

GPU Programs Across Generations

Polybench Code	C2050 Time	K20 Time	K20 Speedup
2DCONV	0.0033	0.0026	1.2729
2MM	0.7670	0.5437	1.4107
3DCONV	0.0060	0.0047	1.2840
3MM	0.0162	0.0102	1.5882
ATAX	0.0188	0.0092	2.0364
BICG	0.0190	0.0092	2.0556
CORR	6.8900	5.3480	1.2883
COVAR	6.9000	5.3440	1.2912
FDTD-2D	0.9280	0.6120	1.5163
GEMM	0.0055	0.0037	1.4890
GESUMMV	0.0379	0.0150	2.5267
GRAMSCHM	6.7100	6.2600	1.0719
MVT	0.0187	0.0092	2.0260
SYR2K	16.5000	6.7700	2.4372
SYRK	0.4690	0.2580	1.8178
Average			**1.6742**
Geometric Mean			**1.5211**

GPU Optimization

- New architecture may be a couple years away...
  - Want performance improvement now!
  - Solution: optimize on current architecture
    - Even small speedup can be significant

GPU Optimization

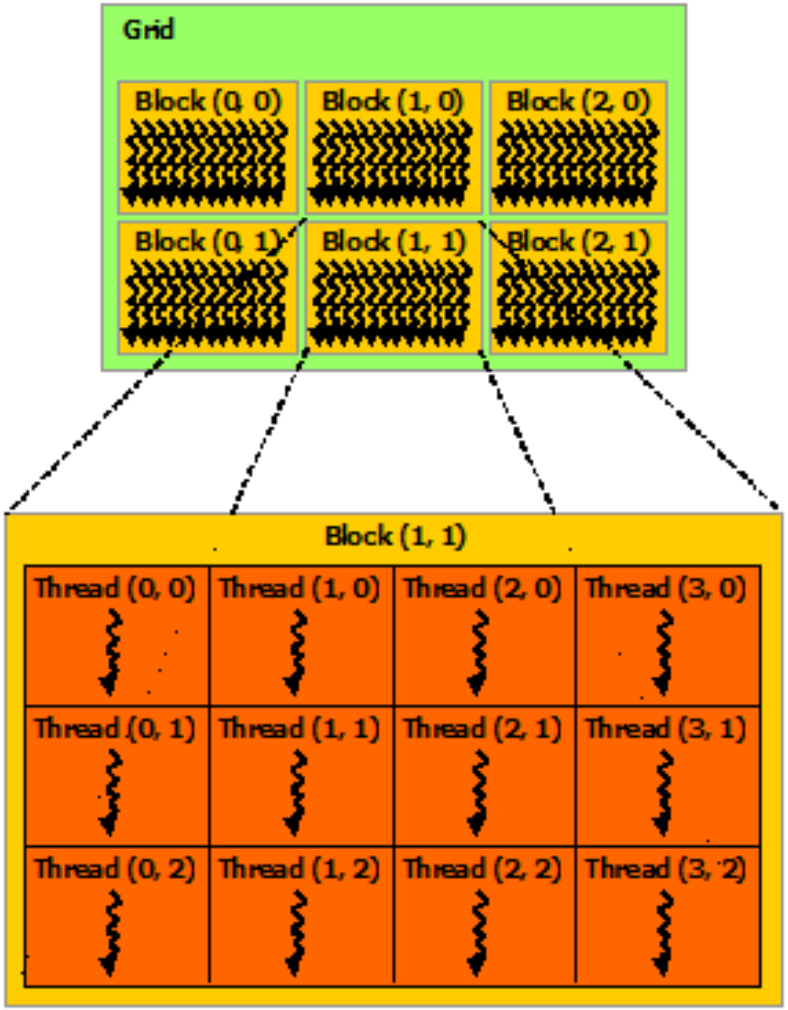
- Challenging problem
  - Particular GPU optimizations may help one program and decrease performance in another
  - Best optimization configuration differs across architectures

NVIDIA GPU Architecture

- CUDA cores grouped into multiprocessors
  - On C1060, there are 8 cores/multiprocessor
  - Multiprocessor contains a limited number of registers
  - Multiprocessor contains fast shared memory (local memory in OpenCL)

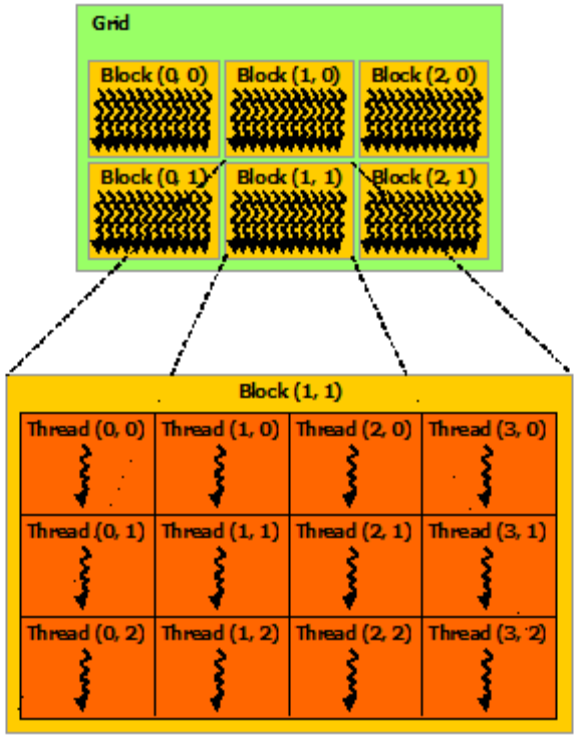
GPU Program Execution Model

- GPU threads grouped into a grid of thread blocks
- Thread block dimensions defined by user
- C1060 GPU allows up to 8 active thread blocks / multiprocessor
- Each active thread block needs to use separate registers/shared memory on multiprocessor



GPU Program Execution Model

- GPU threads (within thread block) execute in groups of 32 on same multiprocessor
  - Each set of 32 threads is called a warp
- All threads in a warp execute same instructions simultaneously
- Different warps are executed independently by scheduler unless there is explicit synchronization call
- Possible for one warp to run until it needs to wait for data, then another warp runs on same



Multiprocessor occupancy

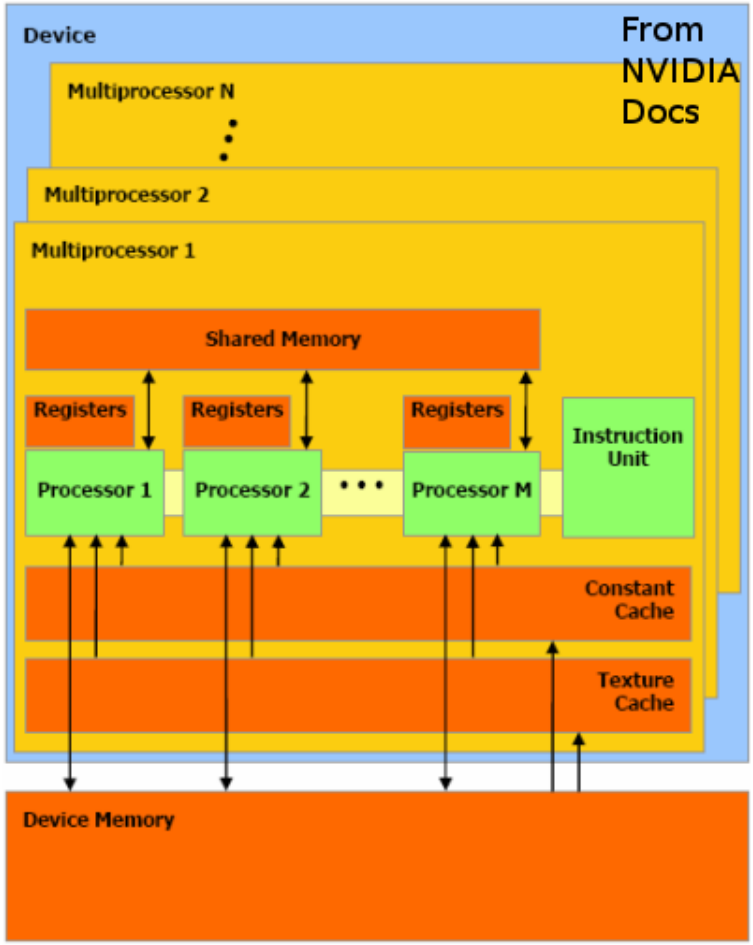
- Defined as number of warps running concurrently on multiprocessor divided by max warps that can run concurrently
  - Max warps differs across architectures
  - On C1060: max of 32 concurrent warps
  - On Kepler K20: max of 64 concurrent warps

Multiprocessor Occupancy

- Can be limited by register and shared memory usage
  - Registers and shared memory are shared among all active warps on multiprocessor
  - Higher register/shared memory usage in each thread limits number of simultaneous warps

Multiprocessor Occupancy

- Higher occupancy is often a goal in GPU optimization
- Greater occupancy can hide instruction latency
  - Read-after-write register latency
  - Latency in reading data from global memory
  - While threads in one warp waiting for data, another warp can run on multiprocessor



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
```

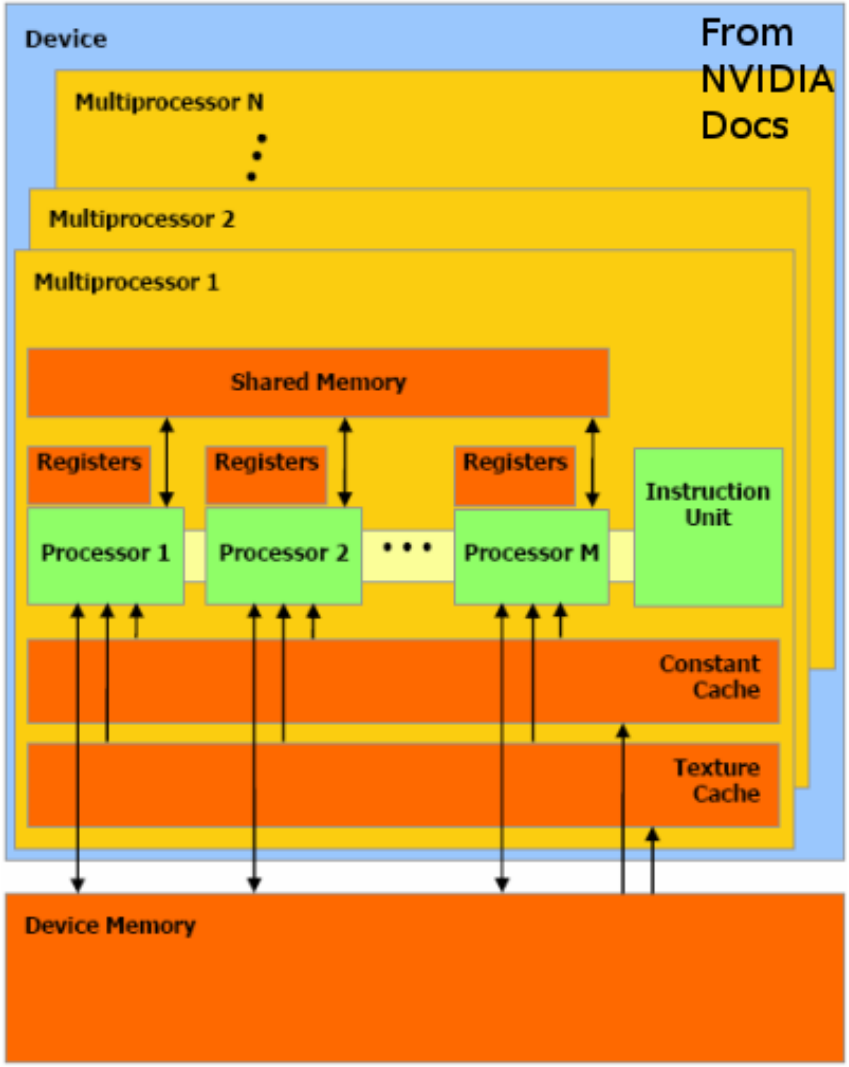
```
s_data[0] += 3;
```

```
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

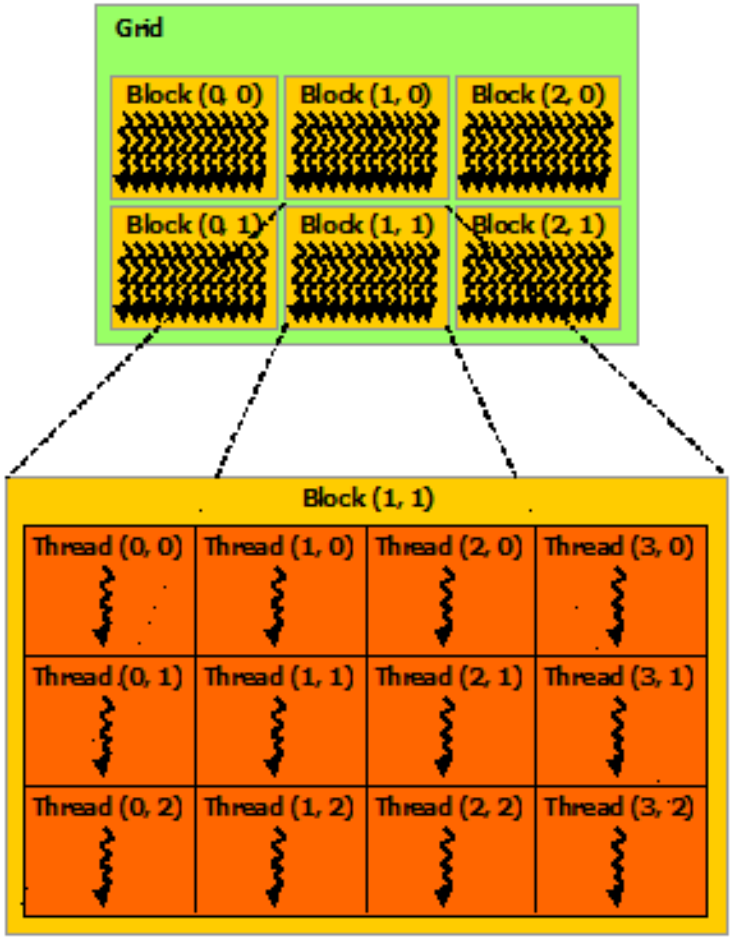
Multiprocessor Occupancy

- However, maximizing occupancy does not always result in best performance
- Increased multiprocessor occupancy can be at expense of faster register/shared memory accesses



Multiprocessor Occupancy Factors

- Thread block dimensions
- Register usage per thread
- Shared memory usage per thread block
- Target GPU architecture



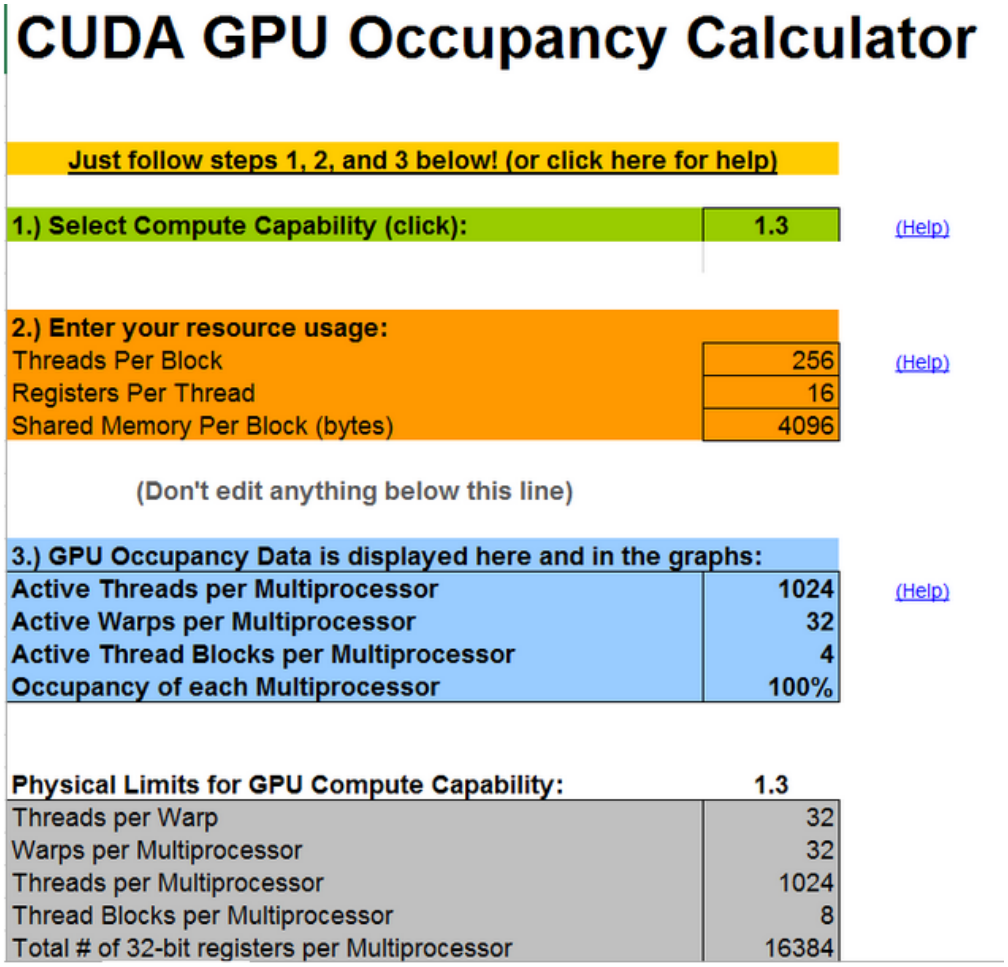
CUDA Occupancy Calculator

- Provided by NVIDIA to compute occupancy of CUDA kernel
- User enters GPU compute capability and resource usage
- Occupancy calculator computes occupancy
- Shows impact of...
  - Adjusting thread block size
  - Adjusting register usage
  - Adjusting shared memory usage

- Can be used as a tool to tweak CUDA program to improve multiprocessor occupancy

CUDA Occupancy Calculator

Available at http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls



CUDA GPU Occupancy Calculator

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

1.) Select Compute Capability (click):	1.3	(Help)

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	16	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	1.3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384

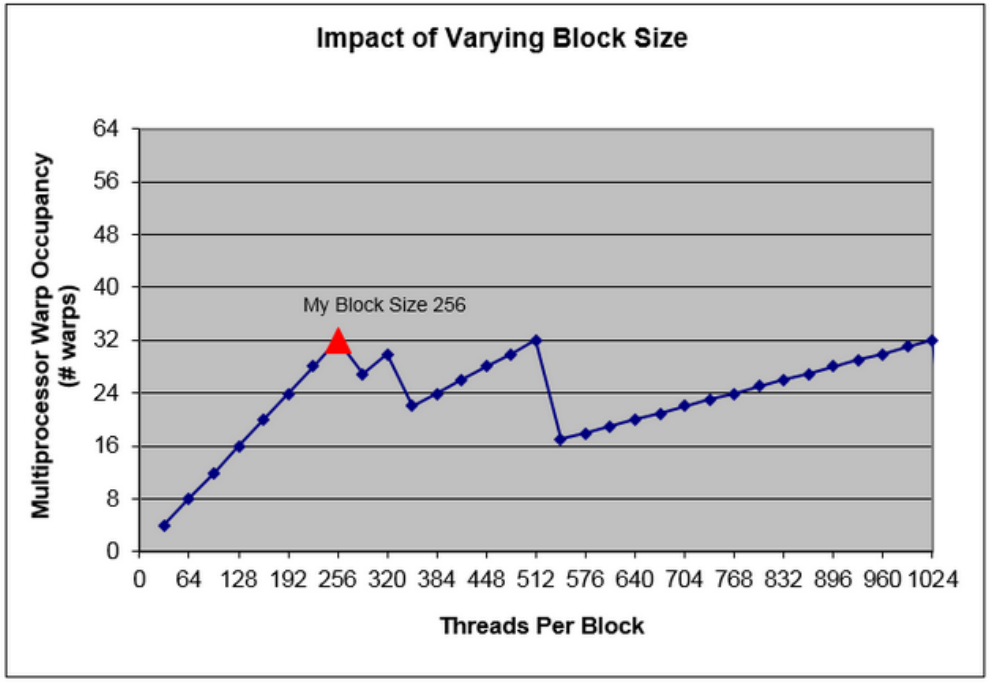
CUDA Occupancy Calculator

Available at http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Click Here for detailed instructions on how to use this occupancy calculator.

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

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.



CUDA Occupancy Considerations

- All things equal, same program with higher occupancy should run faster

- However, may be necessary to sacrifice register / shared memory usage to increase occupancy
  - May increase necessary memory transfers from global memory
  - May slow program more than reduced occupancy

Other Optimization Considerations

- Thread block dimensions
- Global memory load/store pattern
- Register usage
- Local memory usage
- Branches within kernel
- Shared memory
- Constant memory
- Texture memory

Thread Block Dimensions

- CUDA threads grouped together in thread block structure
  - Run on same multiprocessor
  - Have access to common pool of fast shared memory
  - Can synchronize between threads in same thread block
- On C1060, maximum of 8 active thread blocks / multiprocessor
- Max thread block size on C1060 is 512

Optimizing Thread Block Dims (C1060)

- On multiprocessor
  - Up to 1024 threads can execute concurrently
  - Up to 8 thread blocks can be active
- To allow full occupancy, thread block size must be at least 128 and a factor of 1024
- Should use thread block size of at least 128*

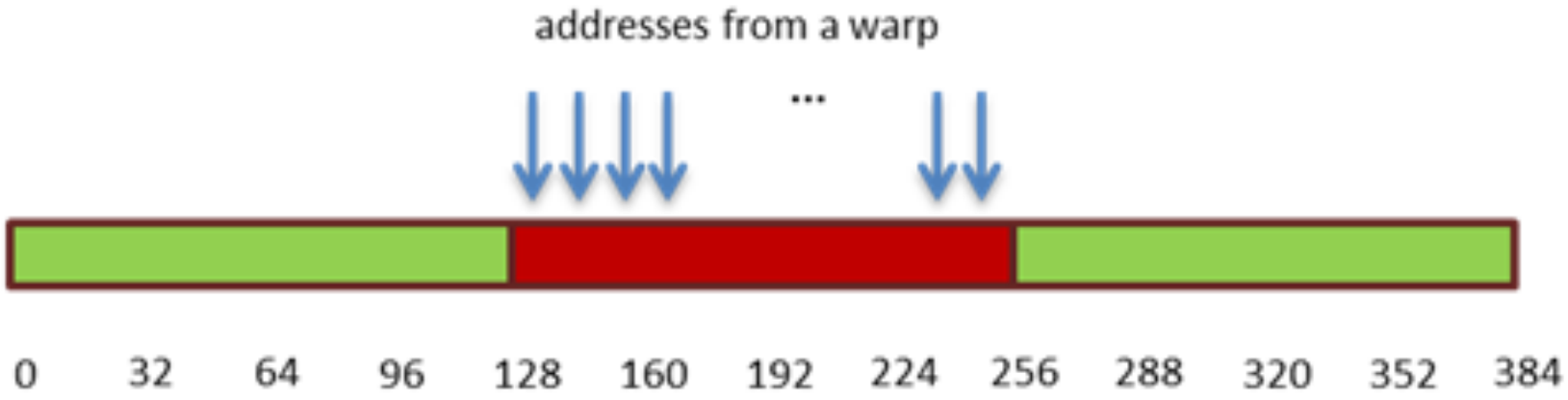
* - unless shared memory requirement forces lower block size

Optimizing Thread Block Dimensions

- Use multiple of warp size (32)
  - Otherwise will be partially full warp --> wasted resources
- Different thread block dimensions work best for different programs
- Experiment and see what works best
  - Common thread block sizes: 128, 192, 256, 384, 512
  - If 2D thread block, common dimensions are 32X4, 32X6, 32X8, 32X12, 32X18

Global memory load/store pattern

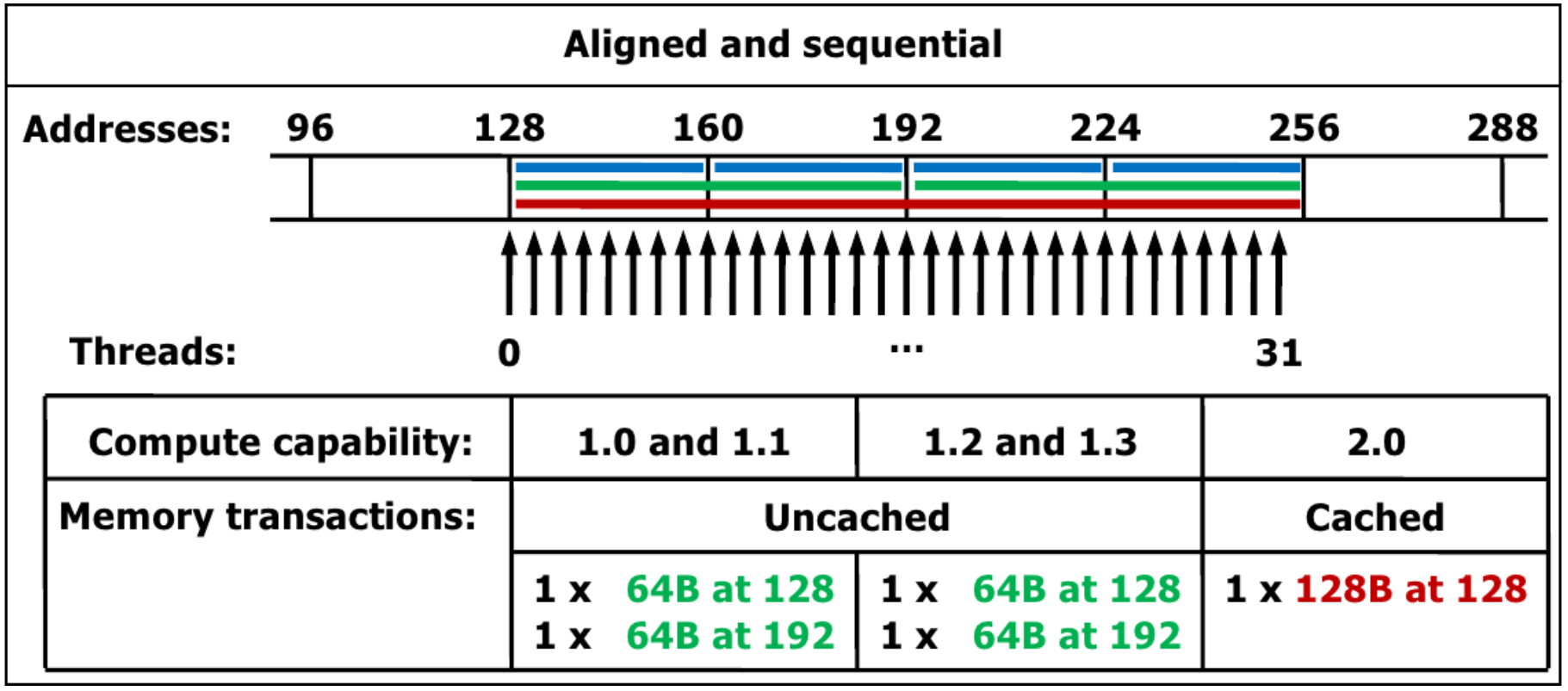
- Global memory latency is over 400 clock cycles
- Minimize accesses as much as possible
- Best to arrange accesses in coalesced manner



Global memory ld/st pattern (C1060)

- Memory accesses in chunks of 32, 64, and 128 bytes
- Global memory accesses are per half-warp (16 threads)
- If data perfectly aligned and thread memory accesses contiguous, data for threads in half-warp retrieved in one chunk
- If data accesses not in same array chunk, need additional memory accesses
  - Anywhere between 1-16 memory accesses necessary to retrieve data from global memory

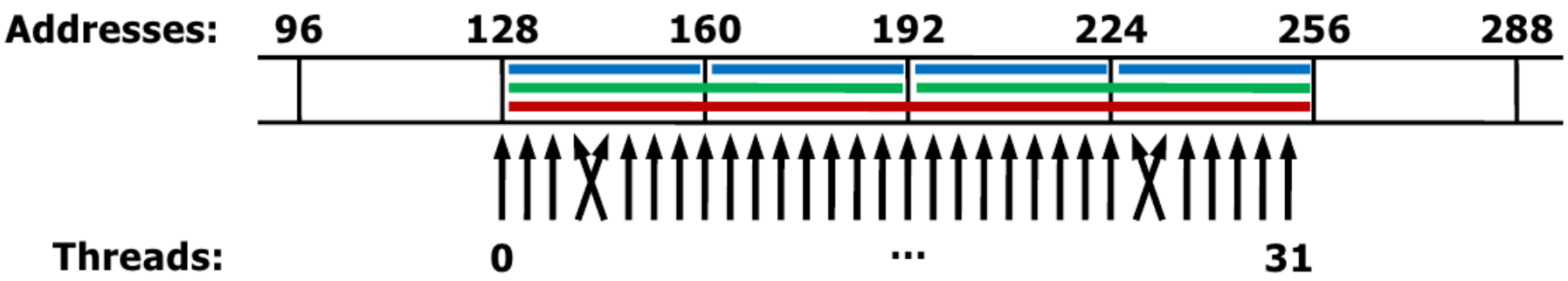
Global Memory Accesses by Architecture



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Global Memory Accesses by Architecture

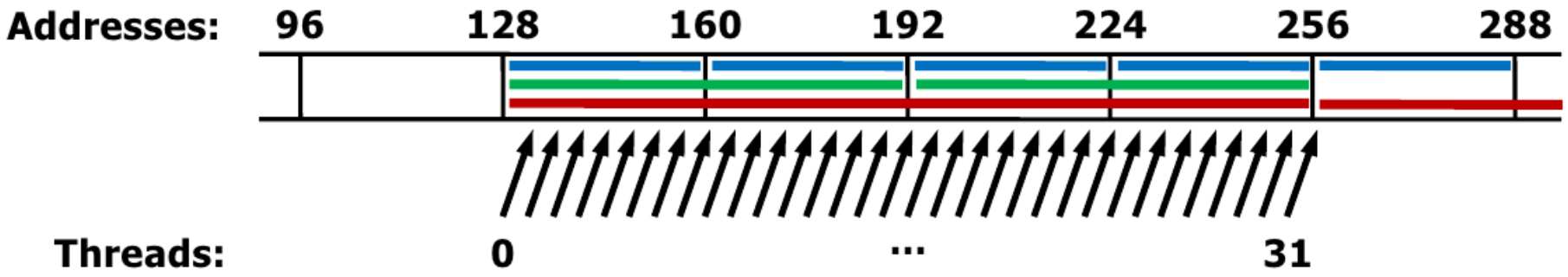
Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Global Memory Accesses by Architecture

Misaligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Register Usage

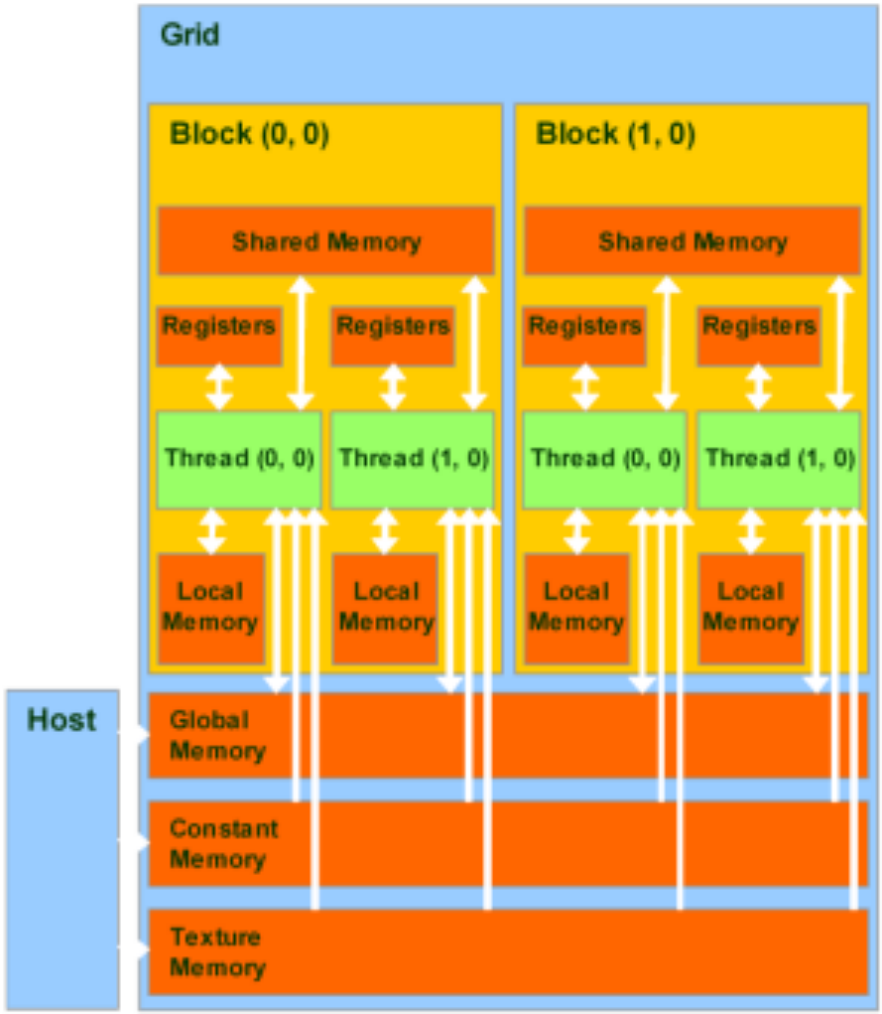
- Registers allow quick access to data on GPU
- Limited number of registers on each GPU multiprocessor
  - C1060: 16,384 registers per multiprocessor
- Limited number of registers / thread allowed
  - C1060: up to 127 registers allowed per thread
  - Fermi / Kepler GK104: up to 63 registers allowed per thread
  - Kepler GK110: up to 255 registers allowed per thread

Register Usage

- Greater register usage limits number of concurrent threads / multiprocessor
  - Decreases multiprocessor occupancy
  - Optimization trade-off
    - Quicker access to data but lower occupancy
- Can limit register use per thread w/ compiler flag
  - –maxrregcount=N flag to NVCC
  - Limiting register use may increase occupancy
  - Can cause spillover to local memory

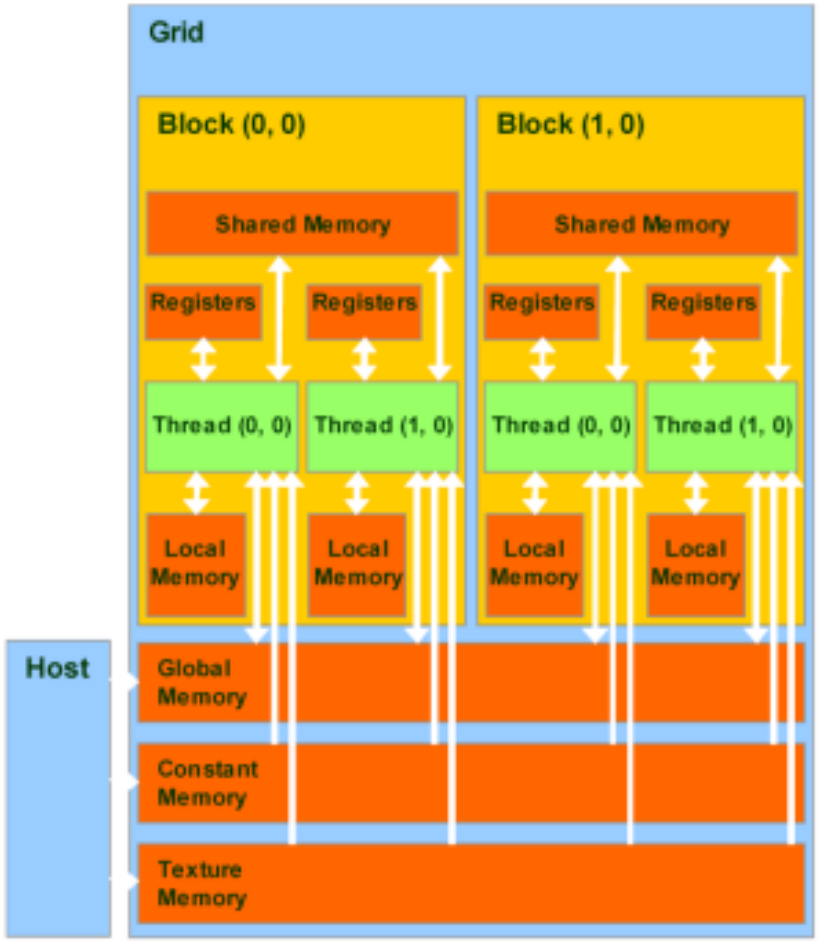
Local Memory Usage

- Local memory on CUDA
  - Data in local memory not explicitly defined
    - No reason to intentionally place data in local memory
  - Corresponds to register spillover and array defined in thread that is accessed in non-unrolled loop
  - Stored in slow global memory space



Local Memory Usage

- Local memory on CUDA
  - Best to avoid as much as possible
    - May be able to place data in registers or shared memory
    - Not as "bad" on Fermi/Kepler due to L1 and L2 cache



Branches Within Kernel

- GPU architecture dedicates most transistors to computation
  - Not much focus on branch prediction / recovery
  - C1060: If divergence in warp, all threads need to step through both branches
  - Thread divergence in warp can hurt performance
  - If branching within thread, want all threads in warp to branch in same direction if possible

Branches Within Kernel

- Example branch within kernel w/ divergence in 32-thread warp
  - **All threads in first warp (corresponding to threads 0-31) need to step through both branches**

```
__global__ void foo(int* data)
{
    if (threadIdx.x < 24){
        data[threadIdx.x] = 5; }
    else { data[threadIdx.x] = 7; }
}
```

Shared Memory

- Can be as fast as registers for data access
  - Can be used as user-managed cache
- Shared between all threads within a thread block
  - Can be used for communication between threads
- C1060: 16KB shared memory/multiprocessor
- Can be limiting factor for multiprocessor occupancy

Shared Memory

- Can be statically declared in CUDA kernel
  - Specified using __shared__ keyword
  - Space for entire thread block given in declaration
  - Example shows shared memory array of size 4*THREAD_BLOCK_SIZE

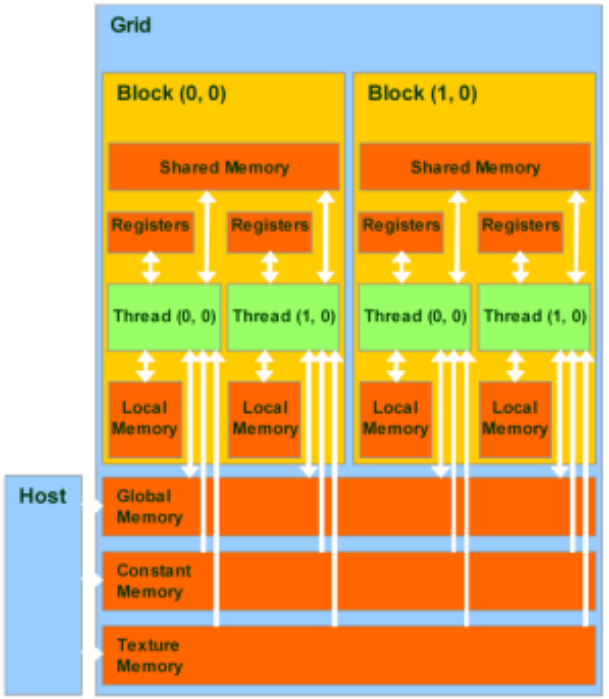
Shared Memory Example

```
__global__ void sharedMemDemo(float* data)
{
    //declare space for shared memory array on GPU
    __shared__ int sharedMemArray
[4*THREAD_BLOCK_SIZE];

    //write thread index to shared memory array at
thread index (no inter-warp conflict since using shared
memory...)
    sharedMemArray[threadIdx.x] = threadIdx.x;
    .....
}
```

Constant Memory

- Potentially faster for read-only data
  - Best when all threads in warp are reading same data simultaneously
  - Allows for single memory read with data then broadcast to remaining threads in half-warp
  - Constant memory also cached
    - Cached constant memory can be as fast as registers



Constant Memory

- Can allocate up to 64KB of constant memory
  - Constant memory cache is 8KB

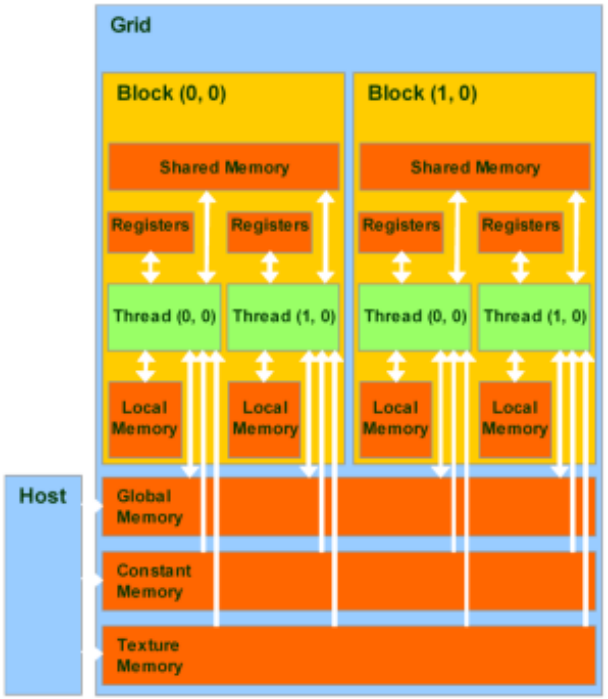
- Constant memory is transferred to GPU from host using cudaMemcpyToSymbol command

```
//declare constant memory
__constant__ float cst_ptr[size];
//copy data from host to constant memory
cudaMemcpyToSymbol(cst_ptr,host_ptr,data_size);
```

Texture Memory

- Also for read-only data
- Uses texture cache on GPU
- Optimized for data with 2D locality

- Can bind linear array or 1D, 2D, or 3D CUDA array to texture
- CUDA array supports linear, bilinear, and trilinear hardware interpolations



Texture Memory Example

- Declare texture to be bound to "float" array
  - texture<float, 1, cudaReadModeElementType> gpuTexture;

- Bind array in global memory to texture:

- Values bound to texture retrieved in kernel using tex1Dfetch (gpuTexture, indexVal)

Texture Memory Example

- Bind array in global memory to texture:

//declare array in global memory on GPU and allocate space

float* gpuGlobMemArray; cudaMalloc(&gpuGlobMemArray, arraySize);

//copy data from host to GPU

cudaMemcpy(gpuGlobMemArray, hostArray, arraySize, cudaMemcpyHostToDevice);

//bind texture to array

cudaBindTexture(0, gpuTexture, gpuGlobMemArray, arraySize);

Performance Measurement

- May need to retrieve run-time characteristics to find possible bottlenecks
  - Often can't determine bottleneck from just inspecting code
  - Solution: use profiler!
  - NVIDIA provides command-line and GUI profiler to retrieve run-time measurements

CUDA Profiler Measurements (C1060)

Info that can be obtained from CUDA Profiler:

- GPU kernel runtime
- Multiprocessor occupancy
- Register use per thread
- Local memory loads/stores
- Number of divergent branches
- Number and type of loads from global memory

Using CUDA Profiler

- Profiler automatically installed with CUDA
- Need to set CUDA_PROFILE (or COMPUTE_PROFILE) environment variable to 1 to enable profiler
- Then run program with profiler enabled
- Profiling output in cuda_profile_0.log
- Default settings give runtime of all GPU kernels and occupancy
  - Can be used for kernel timing

Using CUDA Profiler

- **Need to set flags for more advanced metrics**
  - Create text file profileMetrics.dat
  - Place specific metrics to measure in file
  - Set CUDA_PROFILE_CONFIG to profileMetrics.dat
  - Set CUDA_PROFILE to 1
  - Run program with profiling enabled with given metrics
  - Output in cuda_profile_0.log gives specified run-time metrics
    - Output GPU kernel runtime may be slower when profiling with certain metrics

Using CUDA Profiler

- **Run-time metrics on C1060 (not complete list)**

  gld_32b: 32-byte global memory load transactions
  gld_64b: 64-byte global memory load transactions
  gld_128b: 128-byte global memory load transactions
  gld_request: Global memory loads
  gst_32b: 32-byte global memory store transactions
  gst_64b: 64-byte global memory store transactions
  gst_128b: 128-byte global memory store transactions
  gst_request: Global memory stores

- **Will be more memory transactions if memory accesses non-coalesced than if coalesced**

Using CUDA Profiler

- **Run-time metrics on C1060 (not complete list)**

  local_load: Local memory loads
  local_store: Local memory stores
  branch: Branches taken by threads executing a kernel
  divergent_branch: Divergent branches taken by threads executing a kernel
  instructions: Instructions executed
  warp_serialize: Number of thread warps that serialize on address conflicts to either shared or constant memory
  cta_launched: Number of threads blocks executed

Using CUDA Profiler

- **Run-time metrics on C1060**
  - Can only measure up to 4 metrics at once
  - To measure more than 4 metrics, need to adjust profileMetrics.dat file and re-run program

Sample Output Using Profiler

- Running MonteCarlo in CUDA SDK
- Results with no flags set:

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C1060
# TIMESTAMPFACTOR fffff68f58f9e5c8
method,gputime,cputime,occupancy
method=[ memset32_aligned1D ] gputime=[ 4.288 ] cputime=[ 25.000 ] occupancy=[ 1.000 ]
method=[ memset32_aligned1D ] gputime=[ 2.400 ] cputime=[ 9.000 ] occupancy=[ 1.000 ]
method=[ _Z16inverseCNDKernelPfS_j ] gputime=[ 30.816 ] cputime=[ 8.000 ] occupancy=[ 1.000 ]
method=[ memcpyHtoD ] gputime=[ 5.696 ] cputime=[ 9.000 ]
method=[ _Z27MonteCarloOneBlockPerOptionPfi ] gputime=[ 3209.408 ] cputime=[ 13.000 ] occupancy=[ 0.750 ]
method=[ memcpyDtoH ] gputime=[ 5.440 ] cputime=[ 3232.000 ]
```

Sample Output Using Profiler

- Running MonteCarlo in CUDA SDK
- Results with gld_32b, gst_32b, gld_64b, and gst_64b flags set:

 # CUDA_PROFILE_LOG_VERSION 2.0

 # CUDA_DEVICE 0 Tesla C1060

 # TIMESTAMPFACTOR fffff68fe15c7d58

method,gputime,cputime,occupancy,gld_32b,gst_32b,gld_64b,gst_64b

method=[memset32_aligned1D] gputime=[4.256] cputime=[48.000] occupancy=[1.000] gld_32b=[0] gst_32b=[0] gld_64b=[0] gst_64b=[56]

method=[memset32_aligned1D] gputime=[3.936] cputime=[27.000] occupancy=[1.000] gld_32b=[0] gst_32b=[0] gld_64b=[0] gst_64b=[32]

method=[_Z16inverseCNDKernelPfS_j] gputime=[31.264] cputime=[58.000] occupancy=[1.000] gld_32b=[0] gst_32b=[0] gld_64b=[0] gst_64b=[1664]

method=[memcpyHtoD] gputime=[5.952] cputime=[8.000]

method=[_Z27MonteCarloOneBlockPerOptionPfi] gputime=[3645.568] cputime=[3673.000] occupancy=[0.750] gld_32b=[0] gst_32b=[52] gld_64b=[425984] gst_64b=[0]

method=[memcpyDtoH] gputime=[5.632] cputime=[33.000]

Sample Output Using Profiler

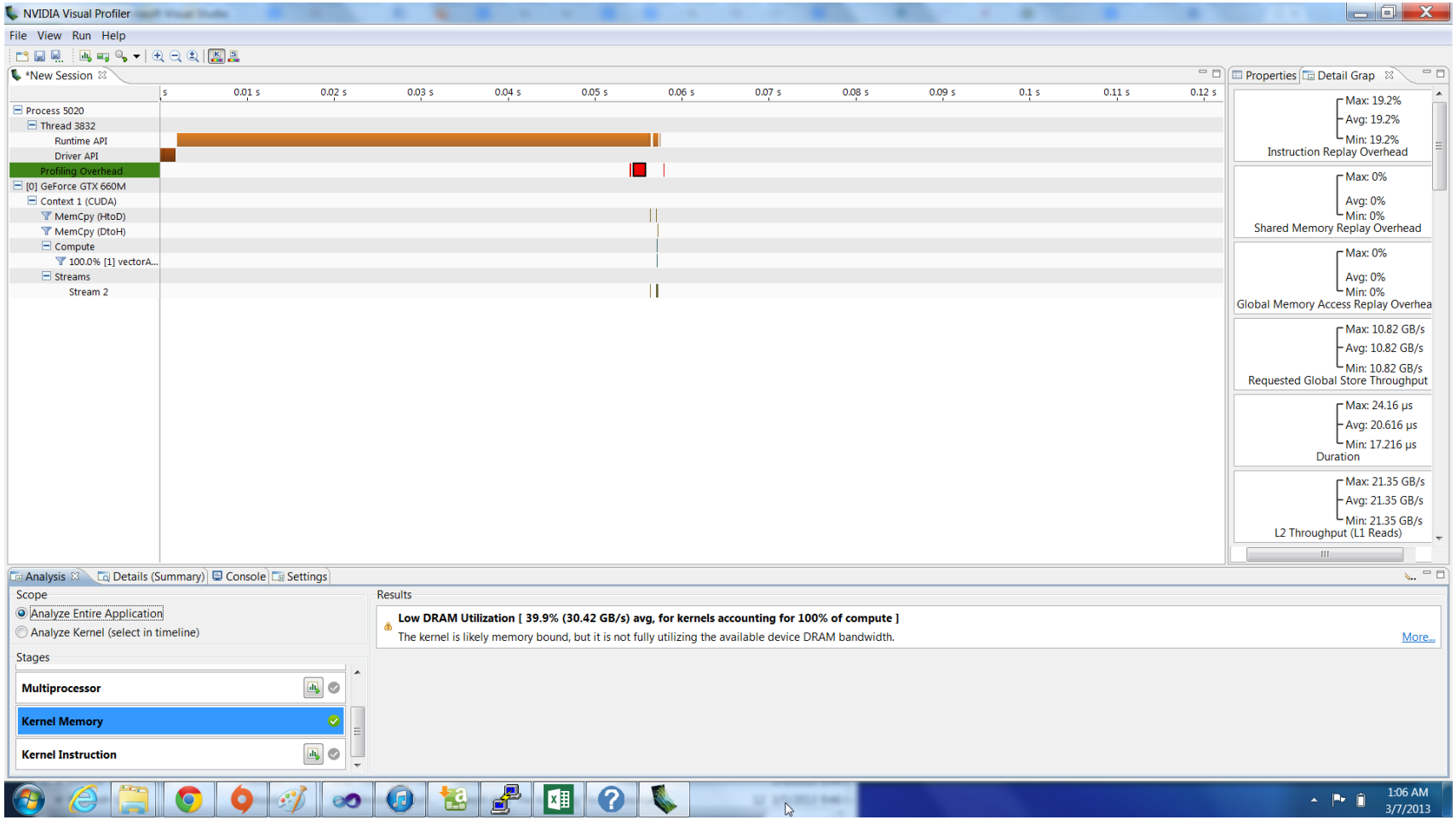
- Running MonteCarlo in CUDA SDK
- Results with local_load, local_store, branch, and divergent_branch flags set:

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C1060
# TIMESTAMPFACTOR fffff68f1cf27808
method,gputime,cputime,occupancy,local_load,local_store,branch,divergent_branch
method=[ memset32_aligned1D ] gputime=[ 3.168 ] cputime=[ 41.000 ] occupancy=[ 1.000 ] local_load=[ 0 ] local_store=[ 0 ] branch=[ 12 ] divergent_branch=[ 0 ]
method=[ memset32_aligned1D ] gputime=[ 3.072 ] cputime=[ 23.000 ] occupancy=[ 1.000 ] local_load=[ 0 ] local_store=[ 0 ] branch=[ 8 ] divergent_branch=[ 0 ]
method=[ _Z16inverseCNDKernelPfS_j ] gputime=[ 30.432 ] cputime=[ 55.000 ] occupancy=[ 1.000 ] local_load=[ 0 ] local_store=[ 0 ] branch=[ 2728 ] divergent_branch=[ 0 ]
method=[ memcpyHtoD ] gputime=[ 5.952 ] cputime=[ 8.000 ]
method=[ _Z27MonteCarloOneBlockPerOptionPfi ] gputime=[ 3626.016 ] cputime=[ 3654.000 ] occupancy=[ 0.750 ] local_load=[ 0 ] local_store=[ 0 ] branch=[ 75528 ] divergent_branch=[ 9 ]
method=[ memcpyDtoH ] gputime=[ 5.792 ] cputime=[ 32.000 ]
```

Using CUDA Profiler

- **Profiler more complicated on Fermi/Kepler**
  - More/different flags
  - Including measurements of L1 and L2 cache hit rate
  - Rules of which metrics can be measured concurrently not given (likely because too complicated)
    - Output will specify if particular metric can't be measured due to conflict
  - GUI Visual profiler allows user to set metrics to measure and runs program multiple times to retrieve data
  - GUI profiler automatically analyzes data / outputs possible bottlenecks

CUDA Visual Profiler



Conclusions

- Many factors in GPU optimizations
- Optimization may help one program and hurt another program
- Necessary to experiment among possible optimizations
- Use CUDA profiler to retrieve run-time measurements than can help with optimization