



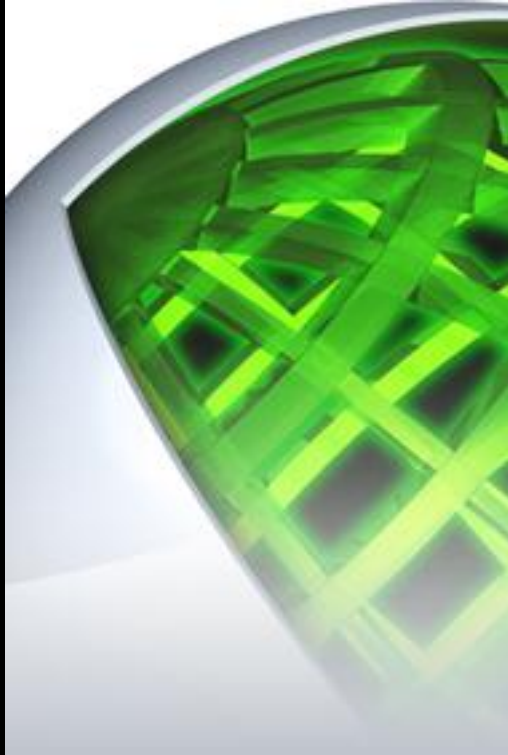
# Performance Optimization Tools And Strategies For CUDA Applications

David Goodwin, NVIDIA

# Performance Opportunities

- Application level opportunities
  - Overall GPU utilization and efficiency
  - Memory copy efficiency
- Kernel level opportunities
  - Instruction and memory latency hiding / reduction
  - Efficient use of memory bandwidth
  - Efficient use of compute resources

# Identifying Performance Opportunities



- NVIDIA® Nsight™ Eclipse Edition (`nsight`)
- NVIDIA® Visual Profiler (`nvvp`)
- `nvprof` command-line profiler

# Optimization Scenarios

- Full application
  - Optimize overall GPU utilization, efficiency, etc.
  - Optimize individual kernels, most likely optimization candidates first
- Algorithm
  - Using a test harness
  - Optimize individual kernels



# Outline: Kernel Optimization

- Discuss an overall application optimization strategy
  - Based on identifying primary performance limiter
- Discuss common optimization opportunities
- How to use CUDA profiling tools
  - Execute optimization strategy
  - Identify optimization opportunities

## 1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels. There are also a number of profiling and optimization technologies that you can leverage to simplify your optimization efforts.

### Examine GPU Utilization

Determine your application's overall GPU utilization. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

### Examine Kernel Performance

Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.


# Identifying Candidate Kernels

- Analysis system estimates which kernels are best candidates for speedup
  - Execution time, achieved occupancy

### 1. CUDA Application Analysis

### 2. Find Performance-Critical Kernels

The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the highest priority kernels, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.

 Perform Kernel Analysis

Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your application will be run once to collect that data for the kernel if it is not already available.

#### **i** Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved speedup:

	Description
1	[ 1 kernel instances ] void miniFE::element_loop_kernel<int, miniFE::SparseMatrix<double>>(int*, double*, double*, double*)
4	[ 51 kernel instances ] void miniFE::spmv_ell_kernel<double, int>(int*, double*, double*, double*)
82	[ 100 kernel instances ] void thrust::detail::backend::cuda::detail::launch_closure_by_value<...>
82	[ 150 kernel instances ] void thrust::detail::backend::cuda::detail::launch_closure_by_value<...>
86	[ 1 kernel instances ] void miniFE::impose_dirichlet_x0_kernel<miniFE::SparseMatrix<double>>(int*, double*, double*, double*)
89	[ 100 kernel instances ] void thrust::detail::backend::cuda::detail::launch_closure_by_value<...>
94	[ 1 kernel instances ] void thrust::detail::backend::cuda::detail::launch_closure_by_value<...>
94	[ 1 kernel instances ] void thrust::detail::backend::cuda::detail::launch_closure_by_value<...>
100	[ 1 kernel instances ] void miniFE::add_to_diagonal_kernel<miniFE::SparseMatrix<double>>(int*, double*, double*, double*)
100	[ 1 kernel instances ] void thrust::detail::backend::cuda::detail::launch_closure_by_value<...>

# Primary Performance Bound

- Most likely limiter to performance for a kernel
  - Memory bandwidth
  - Compute resources
  - Instruction and memory latency
- Primary bound should be addressed first
- Often beneficial to examine secondary bounds as well

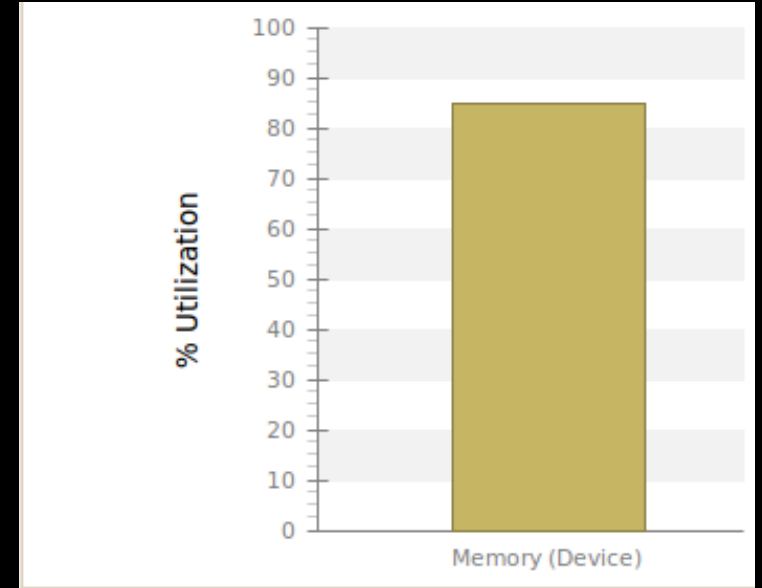
# Calculating Performance Bounds

- Memory bandwidth utilization
- Compute resource utilization
- High utilization value → likely performance limiter
- Low utilization value → likely not performance limiter
- Taken together to determine performance bound



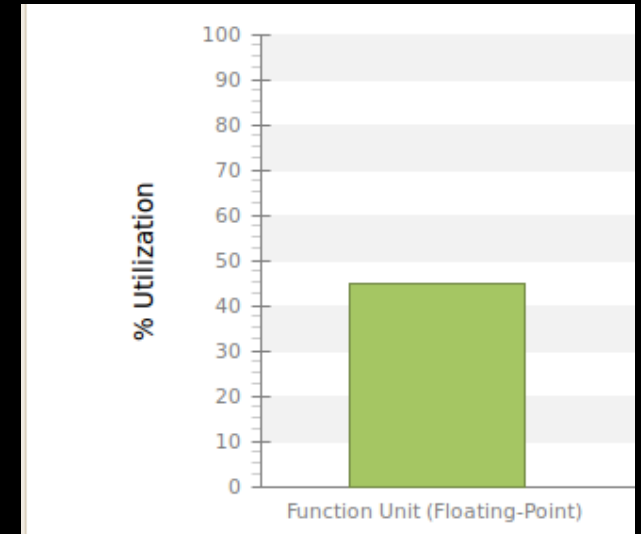
# Memory Bandwidth Utilization

- Traffic to/from each memory subsystem, relative to peak
- Maximum utilization of any memory subsystem
  - L1/Shared Memory
  - L2 Cache
  - Texture Cache
  - Device Memory
  - System Memory (via PCIe)



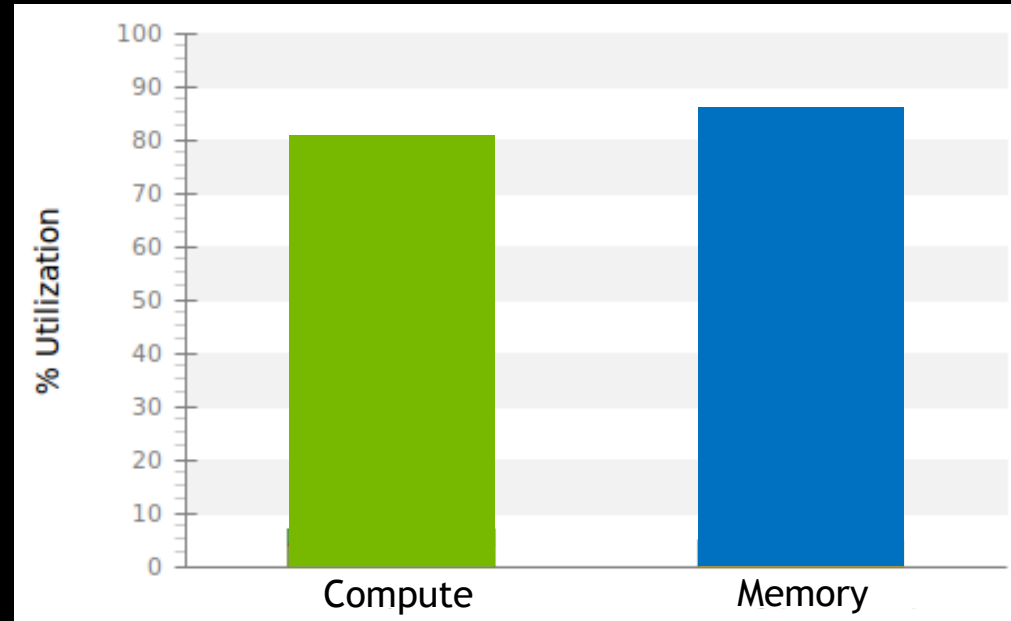
# Compute Resource Utilization

- Number of instructions issued, relative to peak capabilities of GPU
  - Some resources shared across all instructions
  - Some resources specific to instruction “classes”: integer, FP, control-flow, etc.
  - Maximum utilization of any resource



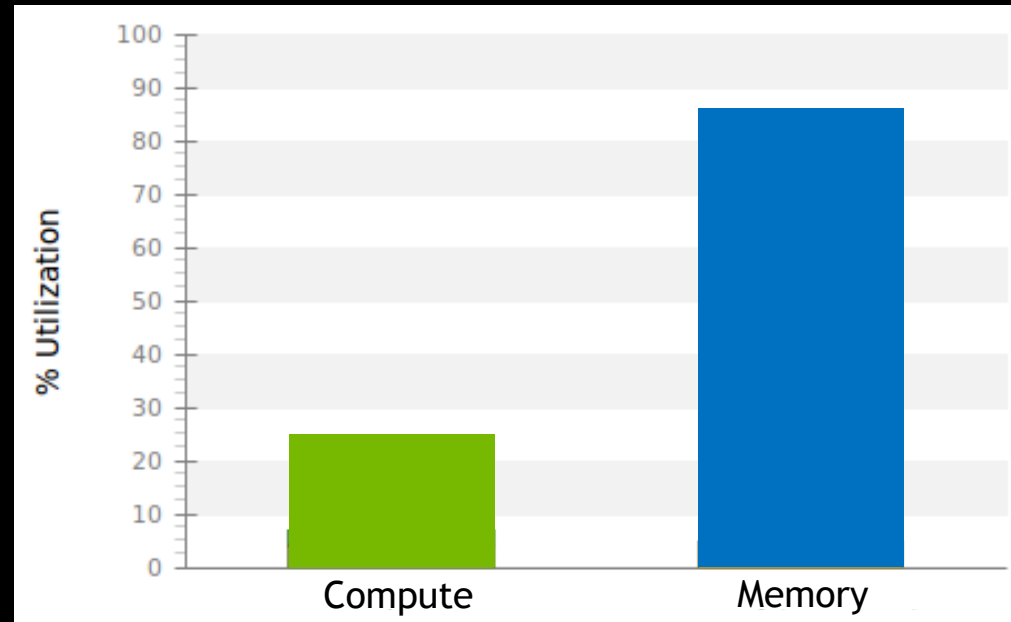
# Calculating Performance Bounds

- Utilizations
  - Both high → compute and memory highly utilized



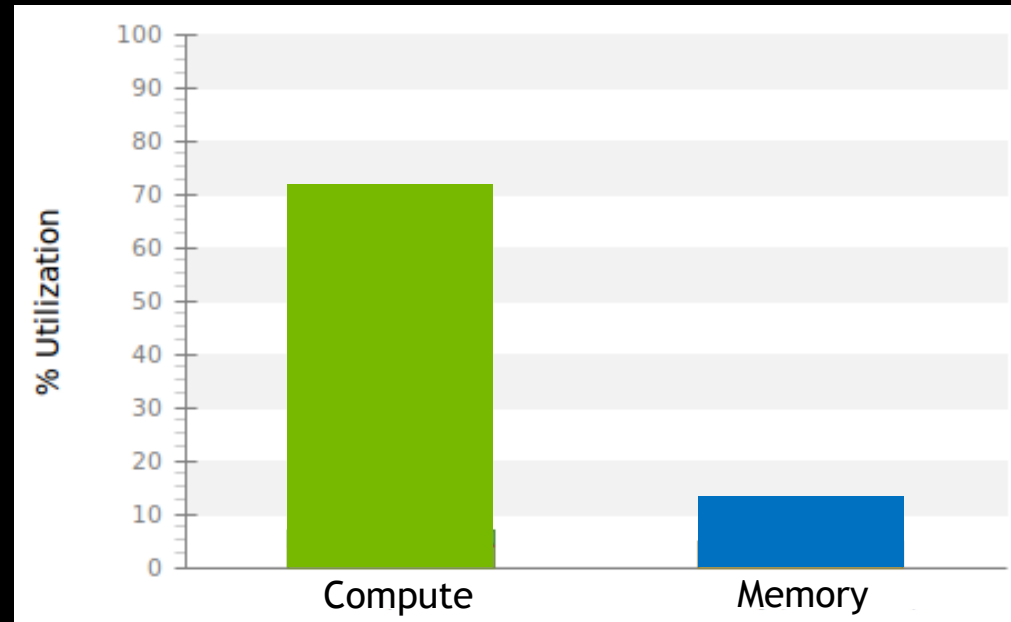
# Calculating Performance Bounds

- Utilizations
  - Memory high, compute low  $\rightarrow$  memory bandwidth bound



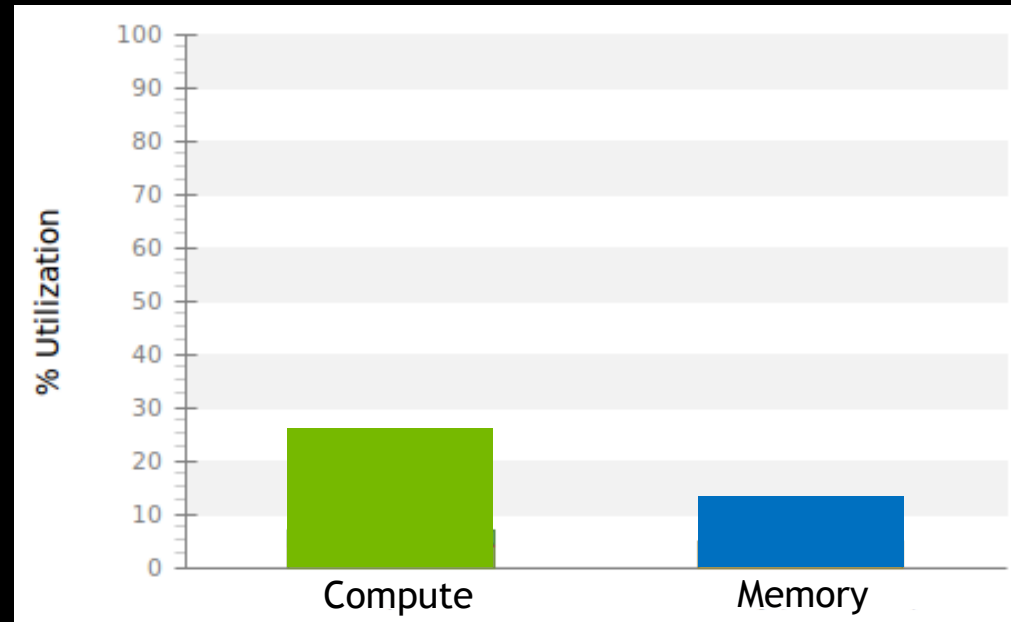
# Calculating Performance Bounds

- Utilizations
  - Compute high, memory low → compute resource bound



# Calculating Performance Bounds

- Utilizations
  - Both low  $\rightarrow$  latency bound




# Visual Profiler: Performance Bound

## 1. CUDA Application Analysis

## 2. Find Performance-Critical Kernels

## 3. Compute, Memory, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory, or latency. The results at right indicate that the performance of kernel "MemoryBoundDeviceMemory" is most likely limited by memory.

 Perform Memory Analysis

The most likely bottleneck to performance for this kernel is memory so you should first perform memory analysis to determine how it is limiting performance.

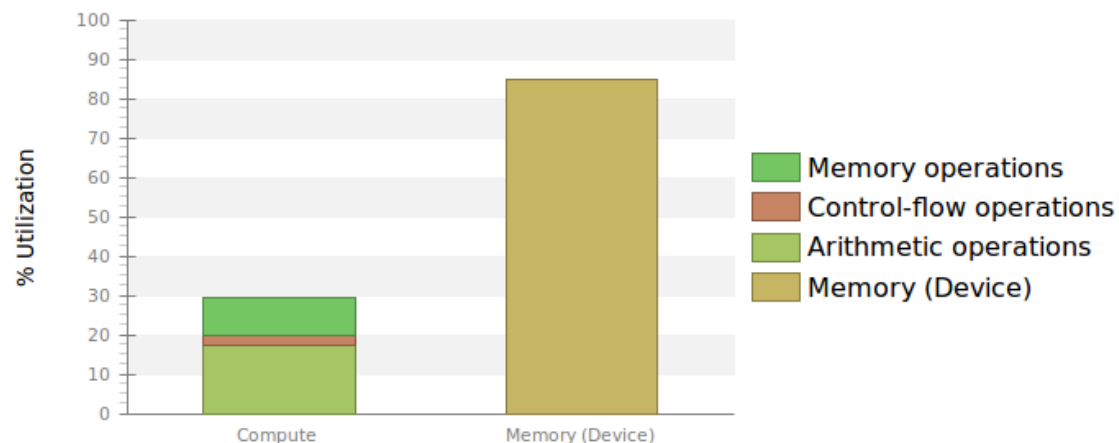
 Perform Compute Analysis

 Perform Latency Analysis

Compute and latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

### i Kernel Performance Is Bound By Memory

For device "Tesla C2050" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by memory.




# Bound-Specific Optimization Analysis

- Memory Bandwidth Bound
- Compute Bound
- Latency Bound


## 3. Compute, Memory, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory, or latency. The results at right indicate that the performance of kernel "MemoryBoundDeviceMemory" is most likely limited by memory.

 Perform Memory Analysis

The most likely bottleneck to performance for this kernel is memory so you should first perform memory analysis to determine how it is limiting performance.

 Perform Compute Analysis

 Perform Latency Analysis

Compute and latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.




# Bound-Specific Optimization Analysis


- Memory Bandwidth Bound
- Compute Bound
- Latency Bound


### 3. Compute, Memory, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory, or latency. The results at right indicate that the performance of kernel "MemoryBoundDeviceMemory" is most likely limited by memory.

 Perform Memory Analysis

The most likely bottleneck to performance for this kernel is memory so you should first perform memory analysis to determine how it is limiting performance.

 Perform Compute Analysis

 Perform Latency Analysis

Compute and latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

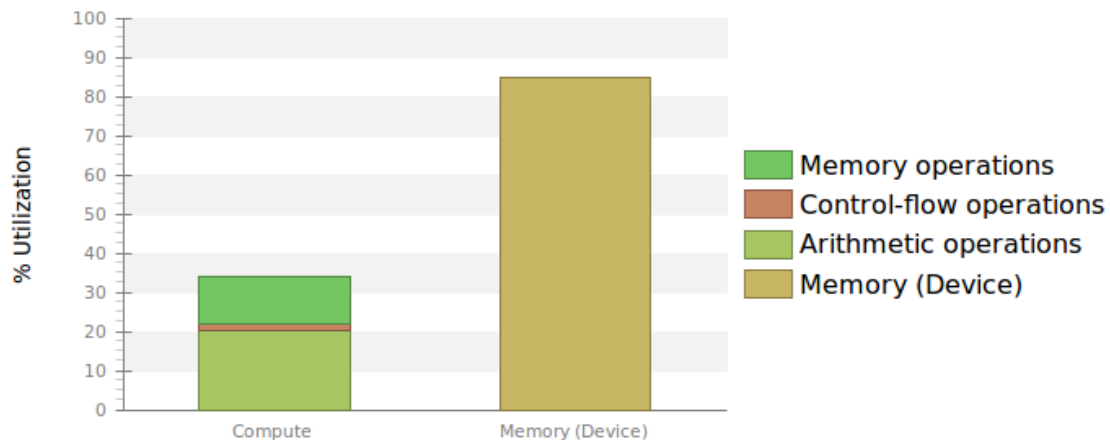
# Memory Bandwidth Bound

```
template<typename Scalar,typename GlobalOrdinal>
__global__ void spmv_ell_kernel(GlobalOrdinal* column_indices,

GlobalOrdinal gridsize=blockDim.x*gridDim.x;
for(GlobalOrdinal row_idx=blockIdx.x*blockDim.x+threadIdx.x;
    row_idx<N;row_idx+=gridsize)
{
    Scalar sum=0;
    GlobalOrdinal offset = row_idx;
    for(int j=0;j<cols;++j)
    {
        GlobalOrdinal c=column_indices[offset];
        if(c!=-1) {
            Scalar A=values[offset];
            Scalar x=__ldg(X+c);
            sum+=A*x;
        }
        offset+=pitch;
    }
    Y[row_idx]=sum;
}
```






## i Kernel Performance Is Bound By Memory

For device "Tesla C2050" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by memory.



# Memory Bandwidths

- Understand memory usage patterns of kernel
  - L1/Shared Memory
  - L2 Cache
  - Texture Cache
  - Device Memory
  - System Memory (via PCIe)

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	1158709	251.08 GB/s	
Global Stores	8624	1.86 GB/s	
L1/Shared Total	1167333	252.94 GB/s	 <div> Idle Low Medium High Max </div>
Texture Cache			
Reads	0	0 B/s	 <div> Idle Low Medium High Max </div>
L2 Cache			
Reads	1635340	91.75 GB/s	
Writes	33163	1.86 GB/s	
Total	1668503	93.61 GB/s	 <div> Idle Low Medium High Max </div>
Device Memory			
Reads	2065469	115.9 GB/s	
Writes	39472	2.23 GB/s	
Total	2104941	118.13 GB/s	 <div> Idle Low Medium High Max </div>
System Memory			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	 <div> Idle Low Medium High Max </div>

# Memory Access Pattern And Alignment

- Access pattern
  - Sequential: 1 memory access for entire warp
  - Strided: up to 32 memory accesses for warp
- Alignment

# Visual Profiler: Memory Efficiency

## ⚠ Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

*Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.*

[More...](#)

Line / File	SparseMatrix_functions.hpp - /minife/src
552	Global Load L2 Transactions/Access = 7.9 [ 878884 L2 transactions for 111942 total executions ]
554	Global Load L2 Transactions/Access = 15.3 [ 1676736 L2 transactions for 109443 total executions ]
555	Global Load L2 Transactions/Access = 17.3 [ 1895720 L2 transactions for 109443 total executions ]

umn\_indices,


```
for(GlobalOrdinal row_idx=blockIdx.x*blockDim.x+threadIdx.x;
    row_idx<N;row_idx+=gridsize)
{
    Scalar sum=0;
    GlobalOrdinal offset = row_idx;
    for(int j=0;j<cols;++j)
    {
        GlobalOrdinal c=column_indices[offset];
        if(c!=-1) {
            Scalar A=values[offset];
            Scalar x=__ldg(X+c);
            sum+=A*x;
        }
        offset+=pitch;
    }
    Y[row_idx]=sum;
}
```

# Bound-Specific Optimization Analysis


- Memory Bandwidth Bound
- Compute Bound
- Latency Bound


### 3. Compute, Memory, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory, or latency. The results at right indicate that the performance of kernel "MemoryBoundDeviceMemory" is most likely limited by memory.

 [Perform Memory Analysis](#)

The most likely bottleneck to performance for this kernel is memory so you should first perform memory analysis to determine how it is limiting performance.

 [Perform Compute Analysis](#)

 [Perform Latency Analysis](#)

Compute and latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

# Warp Execution Efficiency

- All threads in warp may not execute instruction
  - Divergent branch
  - Divergent predication

1 of 32 threads = 3%



32 of 32 threads = 100%





# Visual Profiler: Warp Execution Efficiency

## **Low Warp Execution Efficiency**

Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The kernel's warp execution efficiency of 67% is less than 100% due to divergent branches and predicated instructions.

*Optimization: Reduce the amount of intra-warp divergence and predication in the kernel.*

[More...](#)

# Visual Profiler: Warp Execution Efficiency

## Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

*Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.*

[More...](#)

 File: dct8x8\_kernel\_quantization.cuh - /home/david/depot/davidg-linux-sw/sw/gpgpu/samples/3\_Imaging/dct8x8

103 Divergence = 100.0% [ 8192 divergent executions out of 8192 total executions ]

```
short curCoef = SrcDst[(by * BLOCK_SIZE + ty) * Stride + (bx * BLOCK_SIZE + tx) ];
short curQuant = Q[ty * BLOCK_SIZE + tx];
```

```
//quantize the current coefficient
```

```
if (curCoef < 0)
```

```
{
```

```
    curCoef = -curCoef;
```

```
    curCoef += curQuant>>1;
```

```
    curCoef /= curQuant;
```

```
    curCoef = -curCoef;
```

```
}
```

```
else
```

```
{
```

```
    curCoef += curQuant>>1;
```

```
    curCoef /= curQuant;
```

```
}
```

```
__syncthreads();
```

```
curCoef = curCoef * curQuant;
```

```
//copy quantized coefficient back to the DCT-plane
```

```
SrcDst[(by * BLOCK_SIZE + ty) * Stride + (bx * BLOCK_SIZE + tx) ] = curCoef;
```


```
}
```

# Bound-Specific Optimization Analysis


- Memory Bandwidth Bound
- Compute Bound
- Latency Bound


### 3. Compute, Memory, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory, or latency. The results at right indicate that the performance of kernel "MemoryBoundDeviceMemory" is most likely limited by memory.

 Perform Memory Analysis

The most likely bottleneck to performance for this kernel is memory so you should first perform memory analysis to determine how it is limiting performance.

 Perform Compute Analysis

 Perform Latency Analysis

Compute and latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

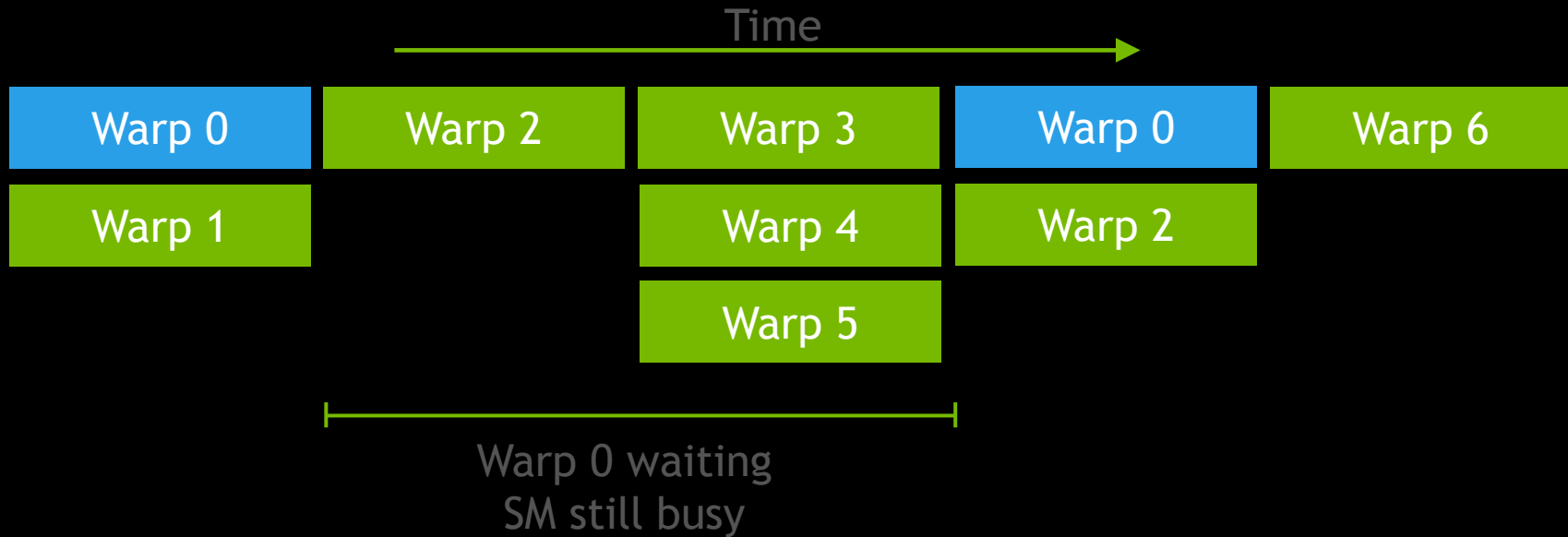
# Latency

- Memory load : delay from when load instruction executed until data returns
- Arithmetic : delay from when instruction starts until result produced



# Hiding Latency

- SM can do many things at once...



- Can “hide” latency as long as there are enough

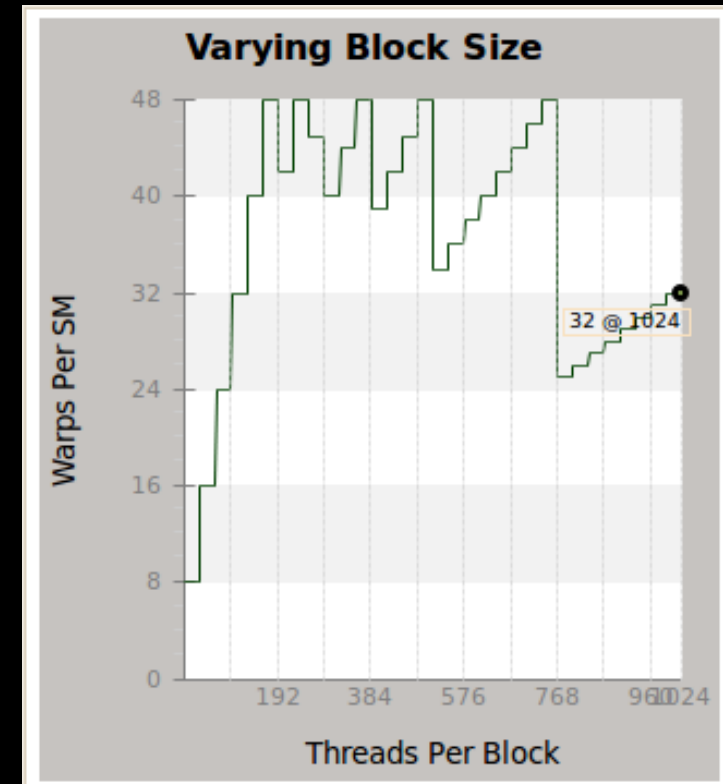
# Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps

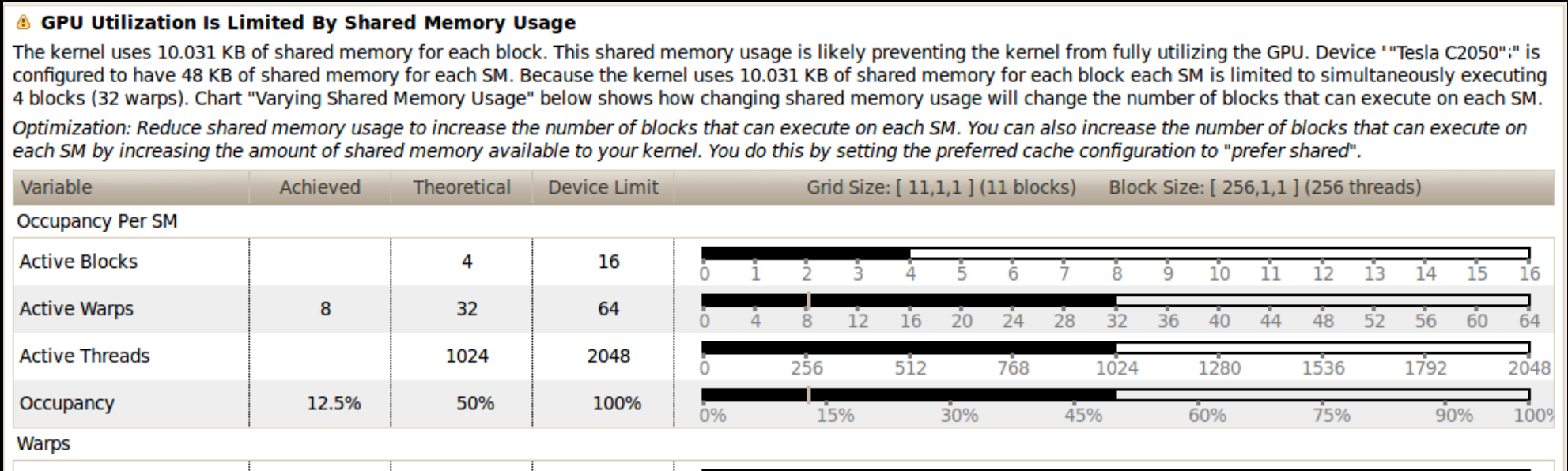
- Theoretical occupancy, upper bound based on:
  - Threads / block
  - Shared memory usage / block
  - Register usage / thread
- Achieved occupancy
  - Actual measured occupancy of running kernel
  - $(\text{active\_warps} / \text{active\_cycles}) / \text{MAX\_WARPS\_PER\_SM}$

# Low Theoretical Occupancy

- # warps on SM = # blocks on SM × # warps per block
- If low...
  - Not enough blocks in kernel
  - # blocks on SM limited by threads, shared memory, registers
- CUDA Best Practices Guide has extensive discussion on improving occupancy

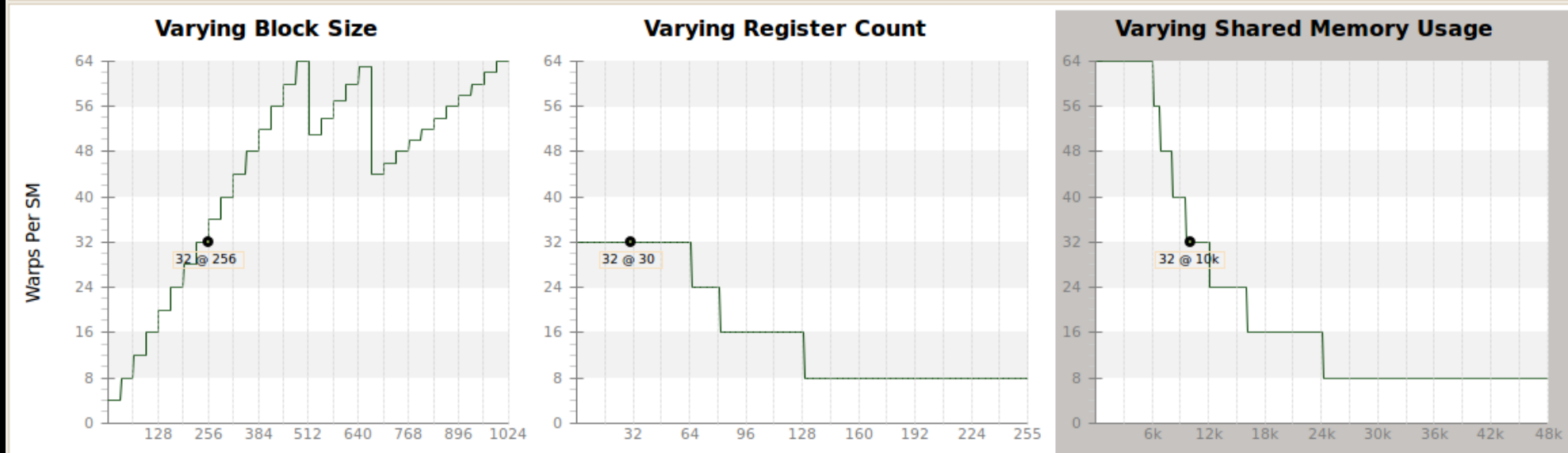
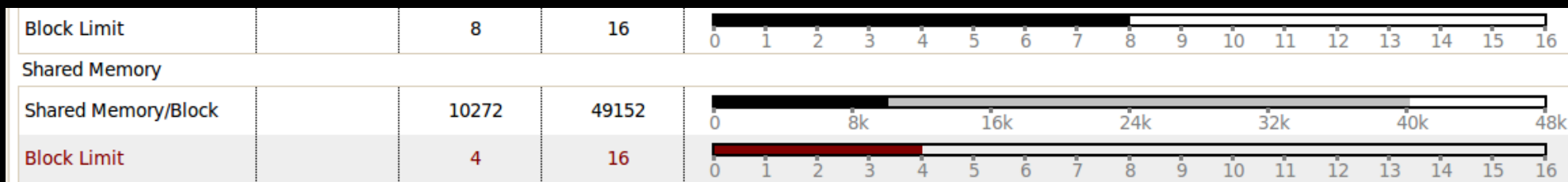


# Visual Profiler: Low Theoretical Occupancy



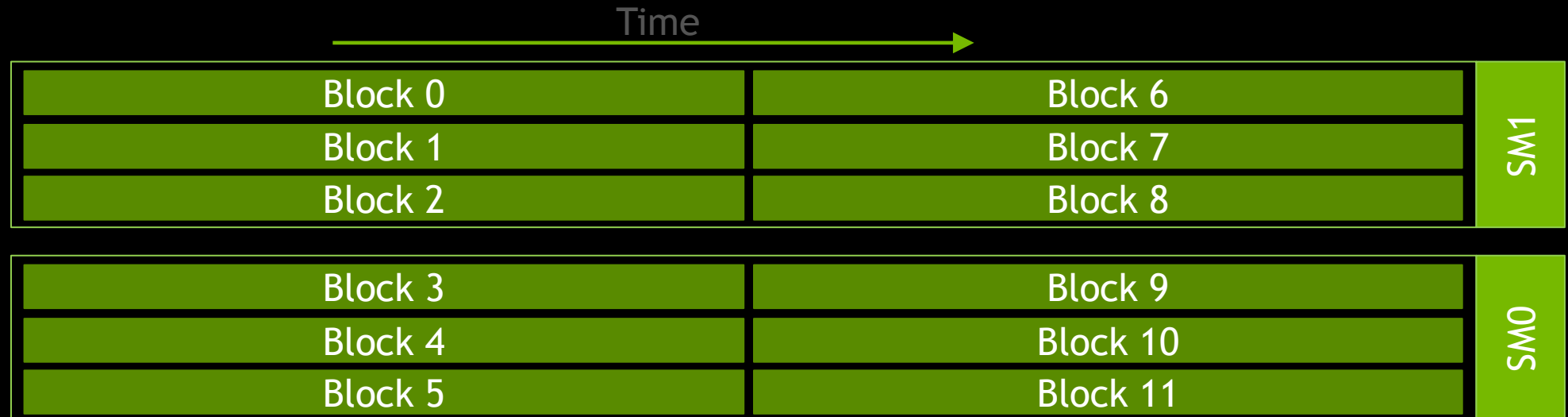


# Visual Profiler: Low Theoretical Occupancy



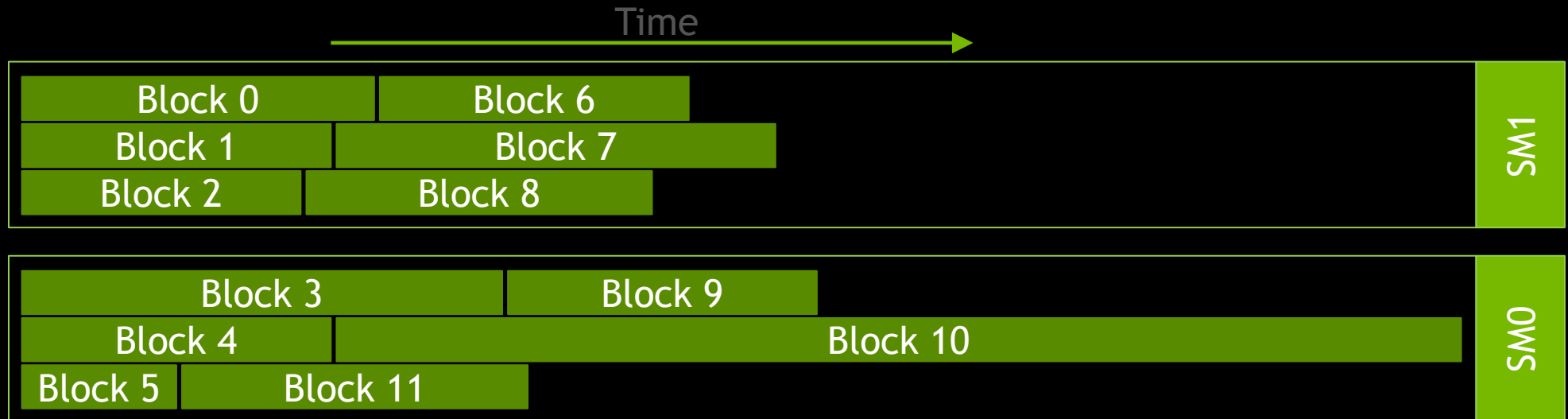
# Low Achieved Occupancy

- In theory... kernel allows enough warps on each SM



# Low Achieved Occupancy

- In theory... kernel allows enough warps on each SM
- In practice... have low achieved occupancy
- Likely cause is that all SMs do not remain equally busy over duration of kernel execution

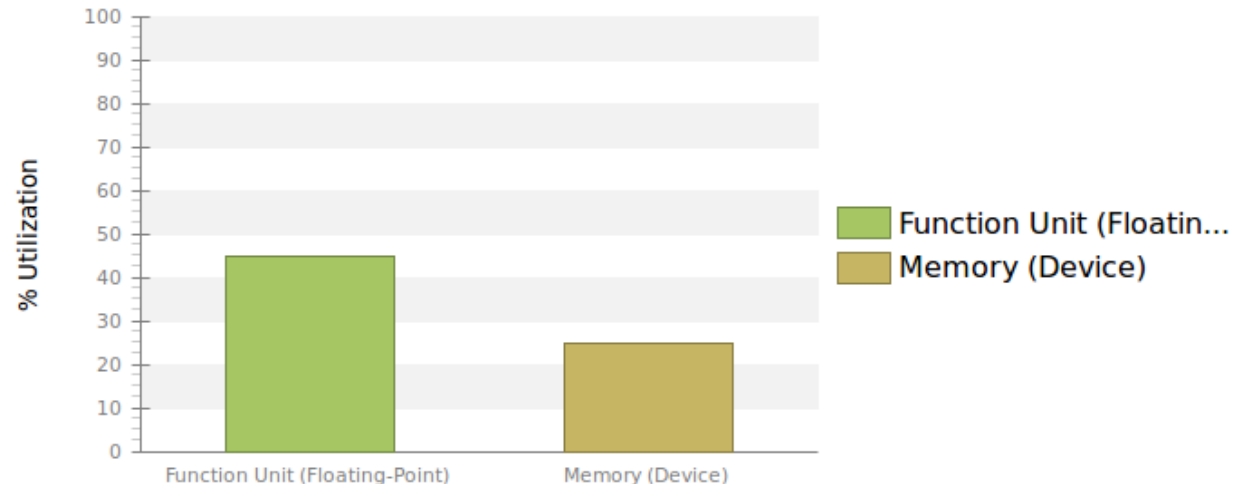


# Latency Bound, Not Occupancy

- Haar wavelet decomposition
  - Excellent occupancy : 100% theoretical, 91% achieved
  - Still latency bound...

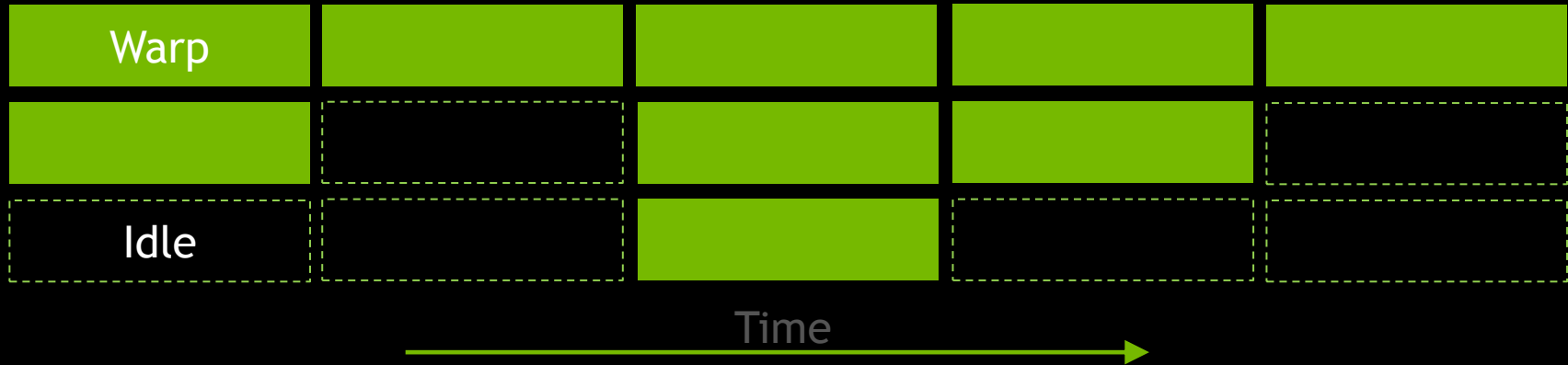
## **i Kernel Performance Is Bound By Latency**

The kernel is utilizing less than 60% of the available compute and memory performance of device "Atlas C-Series". These utilization levels indicate that the performance of the kernel is most likely being limited by instruction and memory latency.

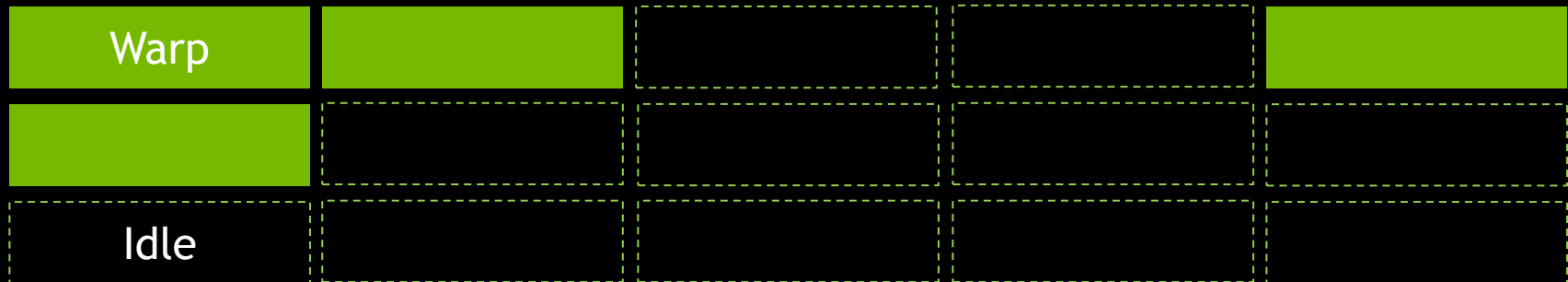


# Lots Of Warps, Not Enough Can Execute

- SM can do many things at once...



- No “ready” warp...

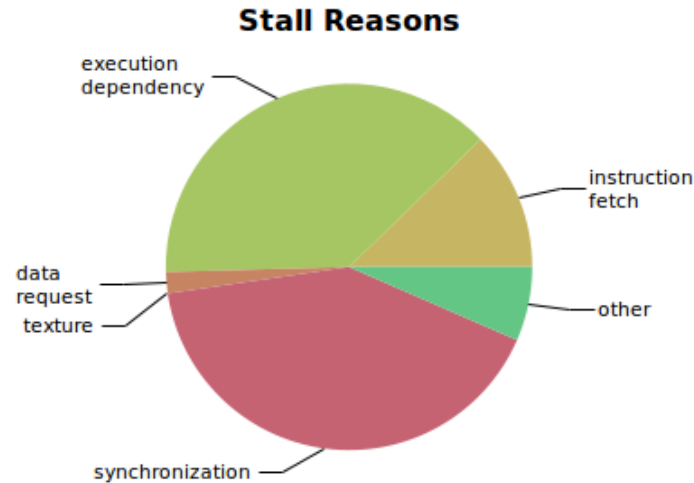


# Latency Bound, Instruction Stalls

## ⚠️ Instruction Latencies May Be Limiting Performance

The kernel has good theoretical and achieved occupancy indicating that there are likely sufficient warps executing on each SM. Since occupancy is not an issue it is likely that performance is limited by the instruction stall reasons described below.

*Optimization: Resolve the primary stall issues; synchronization, execution dependency.*



# \_\_syncthreads Stall

```
// global thread id (w.r.t. to total data set)
const int tid_global = (bid * bdim) + tid;
unsigned int idata = (bid * (2 * bdim)) + tid;

// read data from global memory
shared[tid] = id[idata];
shared[tid + bdim] = id[idata + bdim];
__syncthreads();

// this operation has a two way bank conflicts for all threads, this are two
// additional cycles for each warp -- all alternatives to avoid this bank
// conflict are more expensive than the one cycle introduced by serialization
float data0 = shared[2*tid];
float data1 = shared[(2*tid) + 1];
__syncthreads();

// detail coefficient, not further referenced so directly store in
// global memory
od[tid_global + slength_step_half] = (data0 - data1) * INV_SQRT_2;

// offset to avoid bank conflicts
// see the scan example for a more detailed description
unsigned int atid = tid + (tid >> LOG_NUM_BANKS);
```

# Summary: Kernel Optimization

- Application optimization strategy based on primary performance limiter
  - Memory bandwidth
  - Compute resources
  - Instruction and memory latency
- Common optimization opportunities
- Use CUDA profiling tools to identify optimization opportunities
  - Guided analysis



# Next Steps

- **Download** free CUDA Toolkit: [www.nvidia.com/getcuda](http://www.nvidia.com/getcuda)
- **Join** the community: [developer.nvidia.com/join](http://developer.nvidia.com/join)
- **Post** at Developer Zone Forums: [devtalk.nvidia.com](http://devtalk.nvidia.com)
- GTC On-Demand website for all talks, presentations
- Profiler document at [docs.nvidia.com](http://docs.nvidia.com)