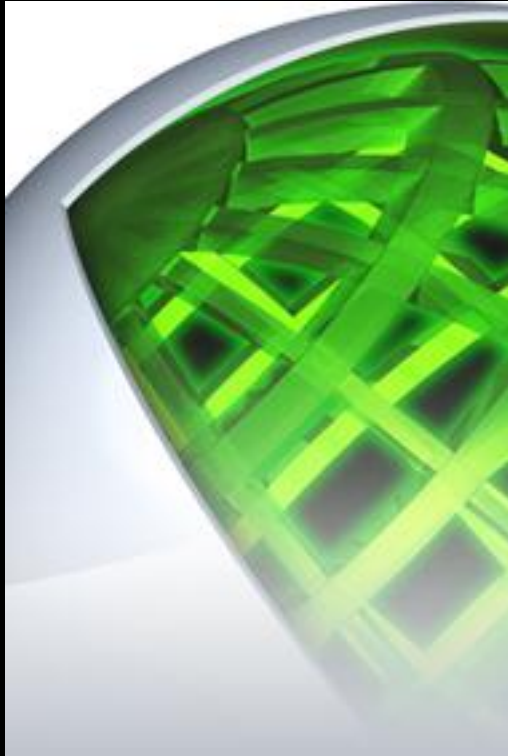




# Guided Performance Analysis with the NVIDIA Visual Profiler

# Identifying Performance Opportunities



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

# Guided Performance Analysis

- NEW in 5.5 Step-by-step optimization guidance

## 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.

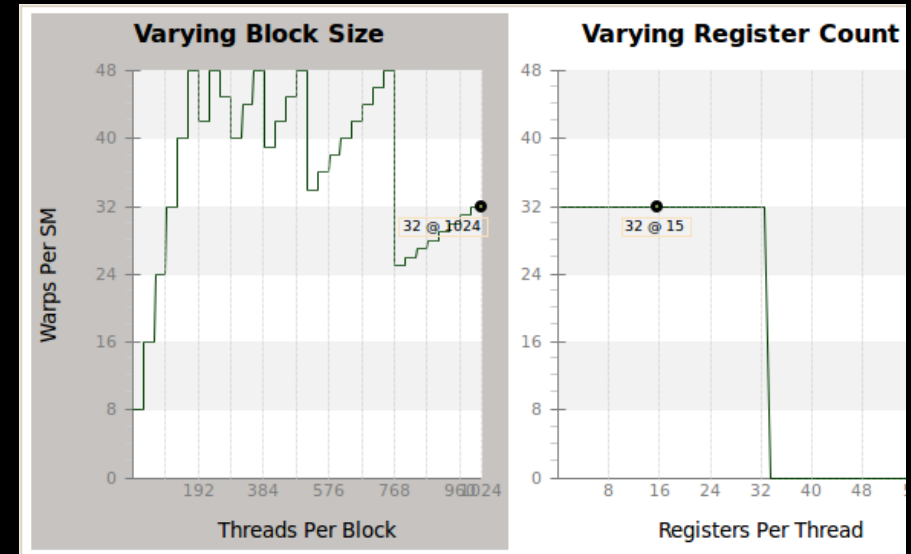
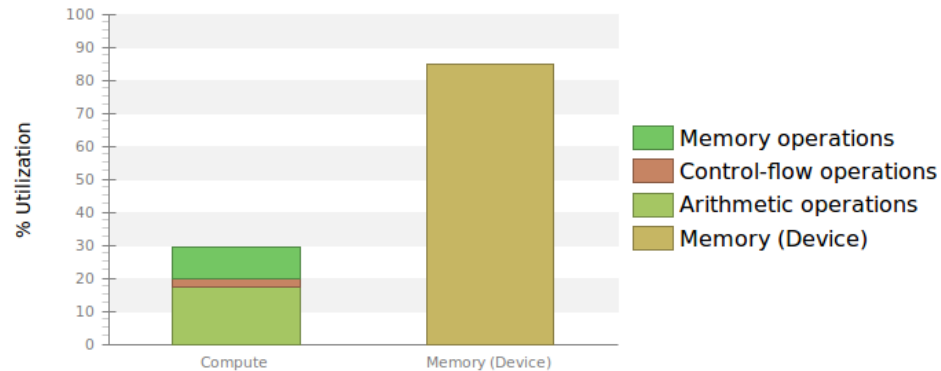
### Examine Kernel Performance

# Improved Analysis Visualization

- NEW in 5.5 More clearly present optimization opportunities

## 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.



# Demo - Analysis Introduction

# Guided Analysis: Kernel Optimization

- Overall application optimization strategy
  - Based on identifying *primary performance limiter*
- 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.



# Primary Performance Limiter

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

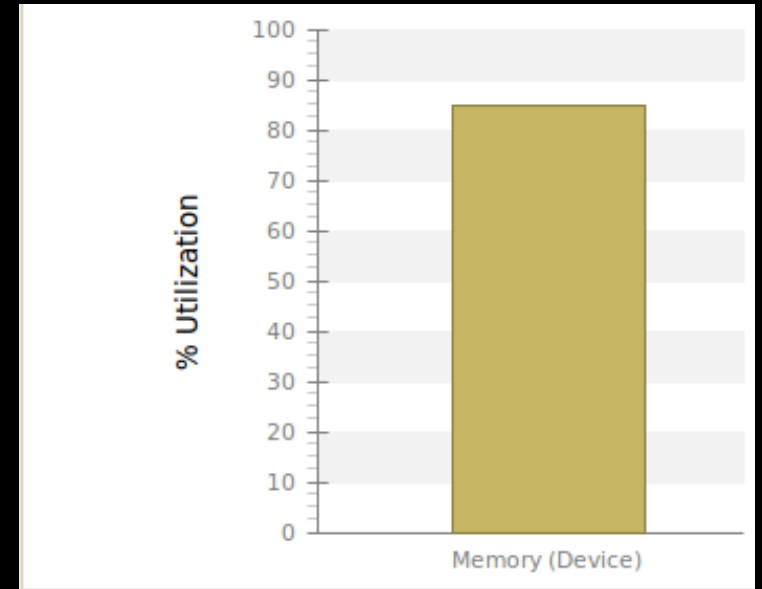


# Calculating Performance Limiter

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

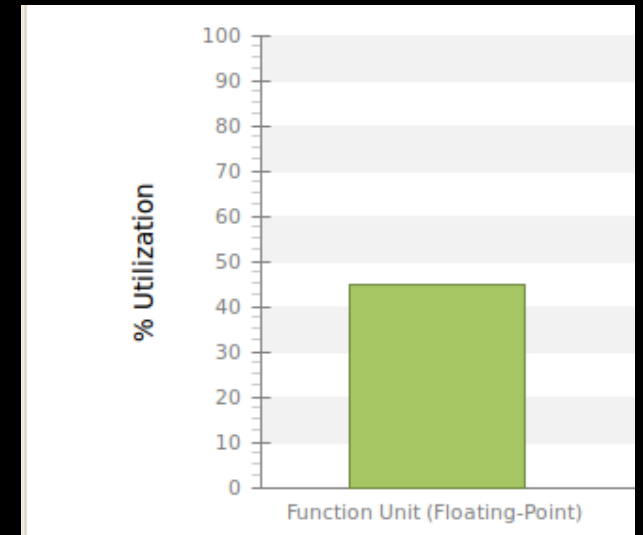
# Memory Bandwidth Utilization

- 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

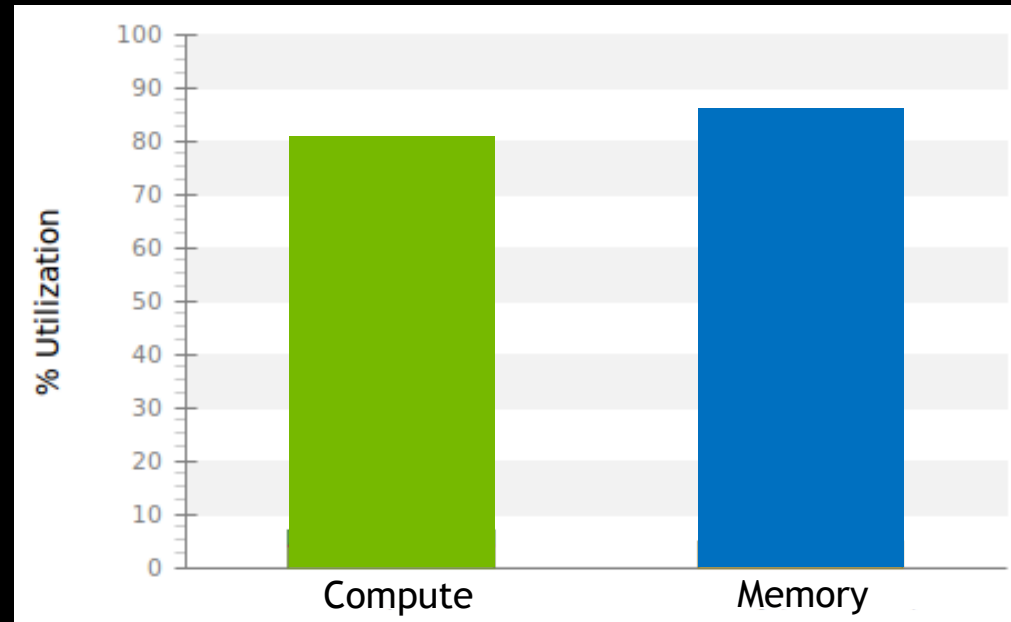
- Number of instructions issued, relative to peak capabilities of GPU
  - Some resources shared across all instructions
  - Some resources specific to instruction “classes”: integer, FP, control-flow, etc.
  - Maximum utilization of any resource



# Calculating Performance Limiter

- Utilizations

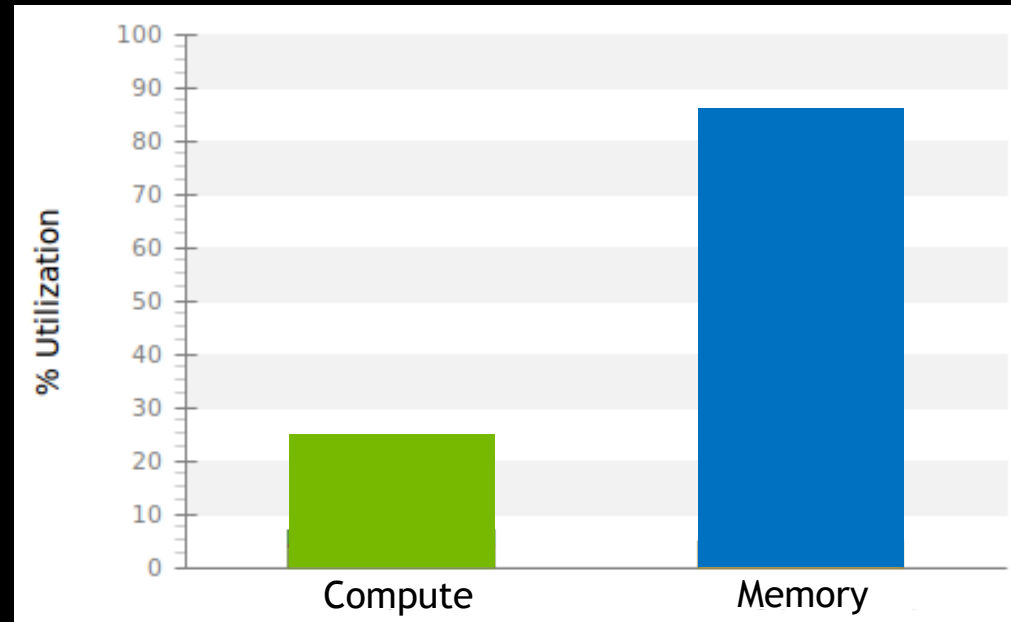
- Both high → compute and memory highly utilized



# Calculating Performance Limiter

- Utilizations

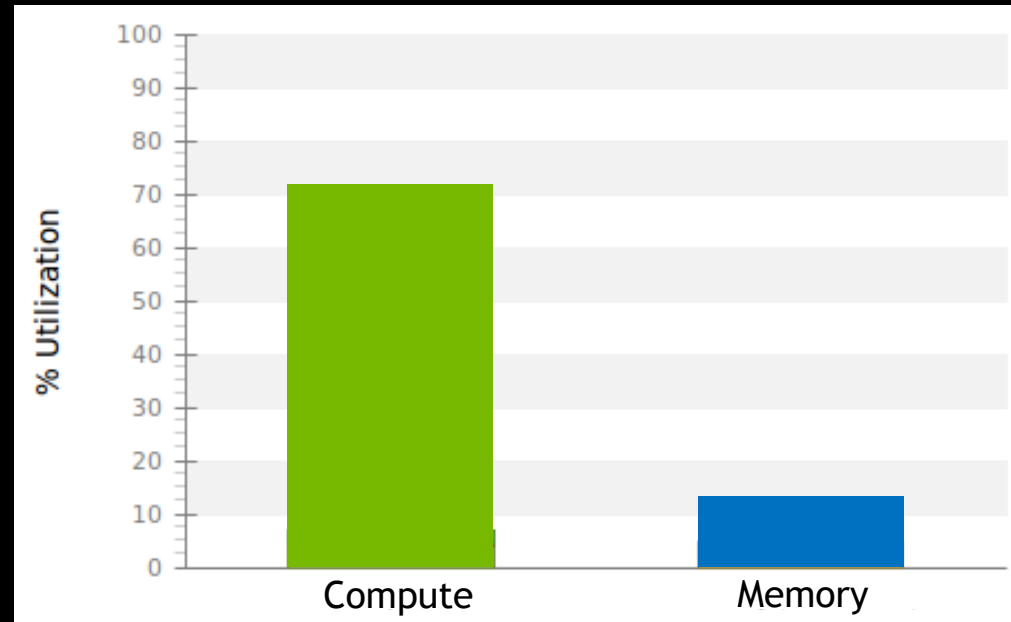
- Memory high, compute low → memory bandwidth limited



# Calculating Performance Limiter

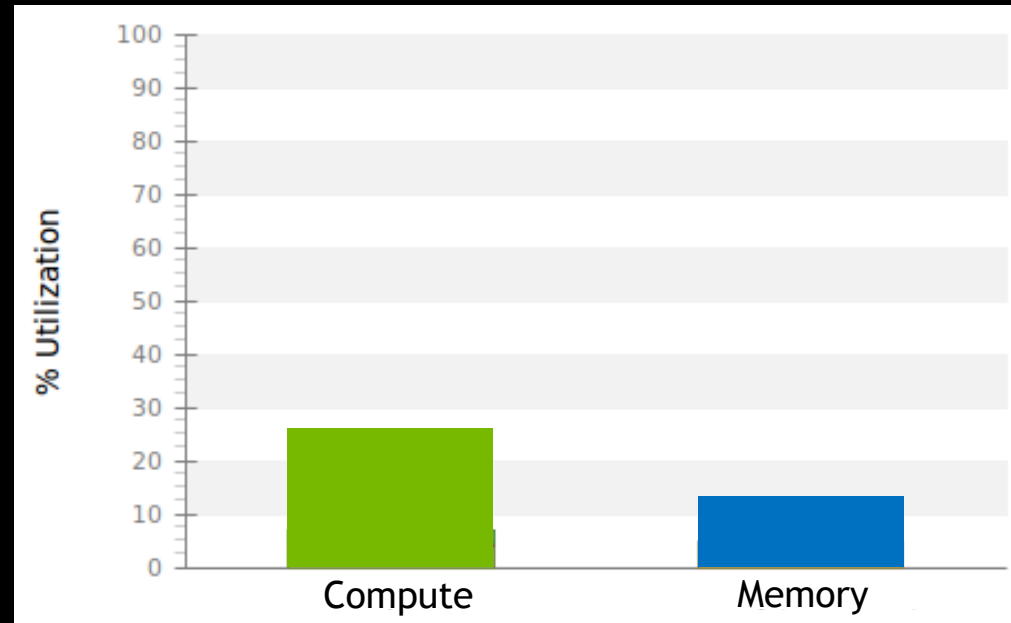
- Utilizations

- Compute high, memory low → compute resource limited



# Calculating Performance Limiter

- Utilizations
  - Both low → latency limited



# Demo - Performance Limiter




# Limiters-Specific Optimization Analysis


- Memory Bandwidth Limited
- Compute Limited
- Latency Limited

## 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 Limited

- Global Memory Load/Store
  - Access pattern inefficiencies
  - Misaligned
- Memory Bandwidth Limits
- Local Memory Overhead
  - Register spilling
  - Local stack variables

# Compute Resource Limited

- Divergent Branches
- Low Warp Execution Efficiency
  - Due to divergent branches
- Over-subscribed function units
  - Load/Store
  - Arithmetic
  - Control-Flow
  - Texture

# Latency Limited

- Low Theoretical Occupancy
  - Block Size
  - Registers
  - Shared Memory
  
- Low Achieved Occupancy
- Too Few Blocks
- Instruction Stalls

# Demo - Limiter Details

# Summary

- Visual Profiler Guided Analysis gives step-by-step optimization advice
- Kernel Analysis strategy based on identifying primary limiter
  - Memory bandwidth
  - Compute resources
  - Instruction and memory latency
- Visual result and integrated documentation

# Upcoming GTC Express Webinars

**September 17** - ArrayFire: A Productive GPU Software Library for Defense and Intelligence Applications

**September 19** - Learn How to Debug OpenGL 4.2 with NVIDIA® Nsight™ Visual Studio Edition 3.1

**September 24** - Pythonic Parallel Patterns for the GPU with NumbaPro

**September 25** - An Introduction to GPU Programming

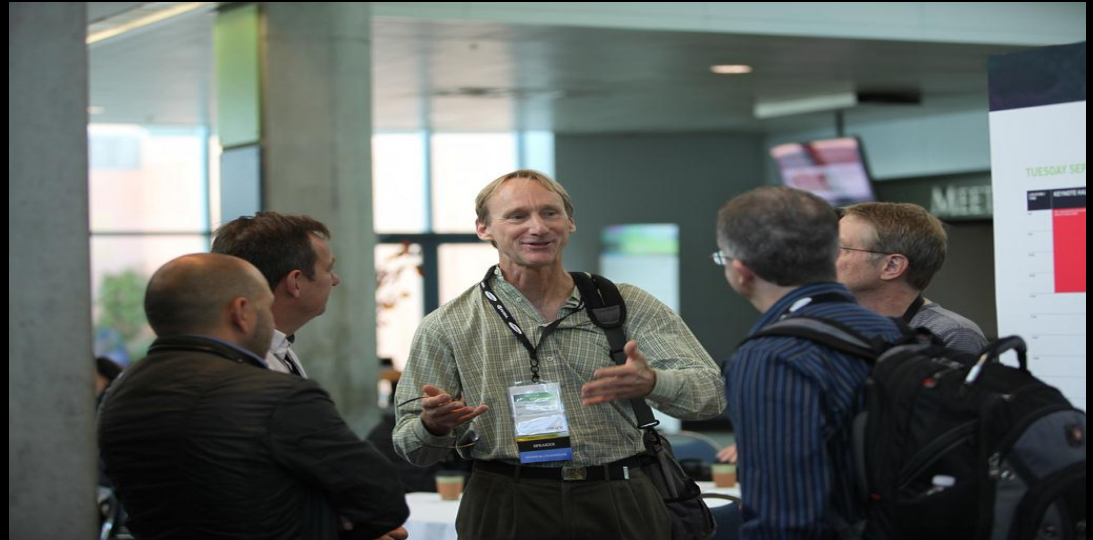
**September 26** - Learn How to Profile OpenGL 4.2 with NVIDIA® Nsight™ Visual Studio Edition 3.1

**Register at [www.gputechconf.com/gtcexpress](http://www.gputechconf.com/gtcexpress)**

# GTC 2014 Call for Submissions

Looking for submissions in the fields of

- Science and research
- Professional graphics
- Mobile computing
- Automotive applications
- Game development
- Cloud computing



**Submit by September 27 at [www.gputechconf.com](http://www.gputechconf.com)**



**Extra**

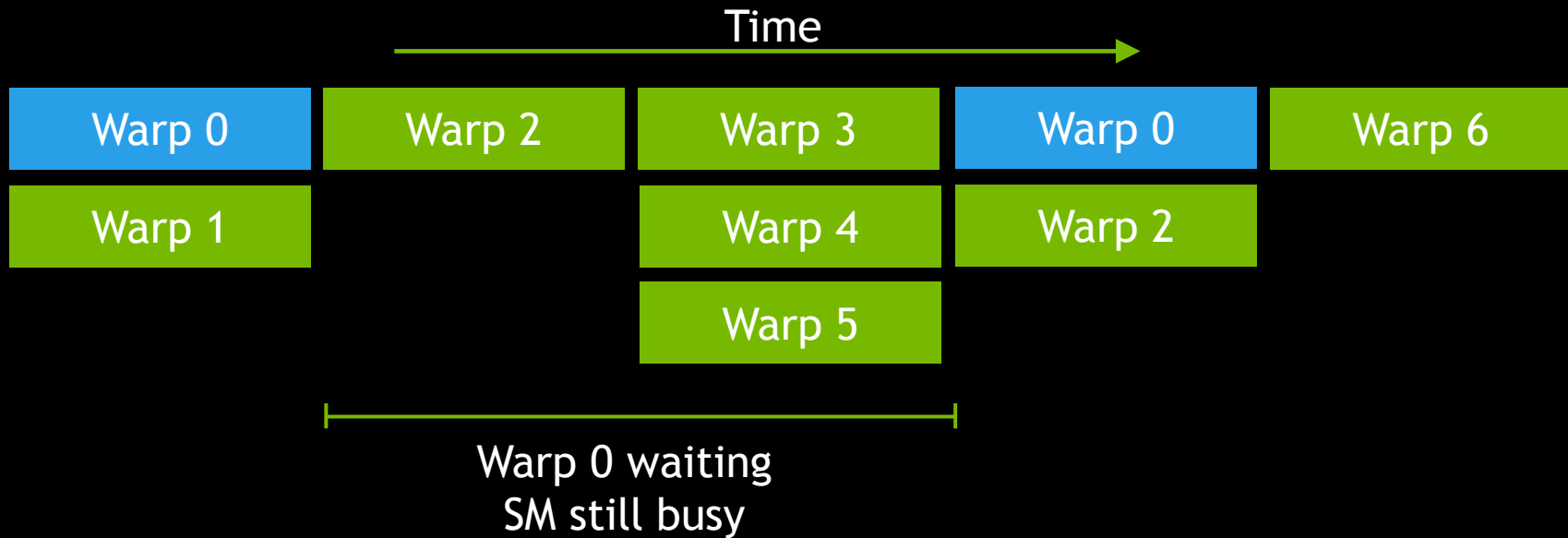
# Understanding Latency

- 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

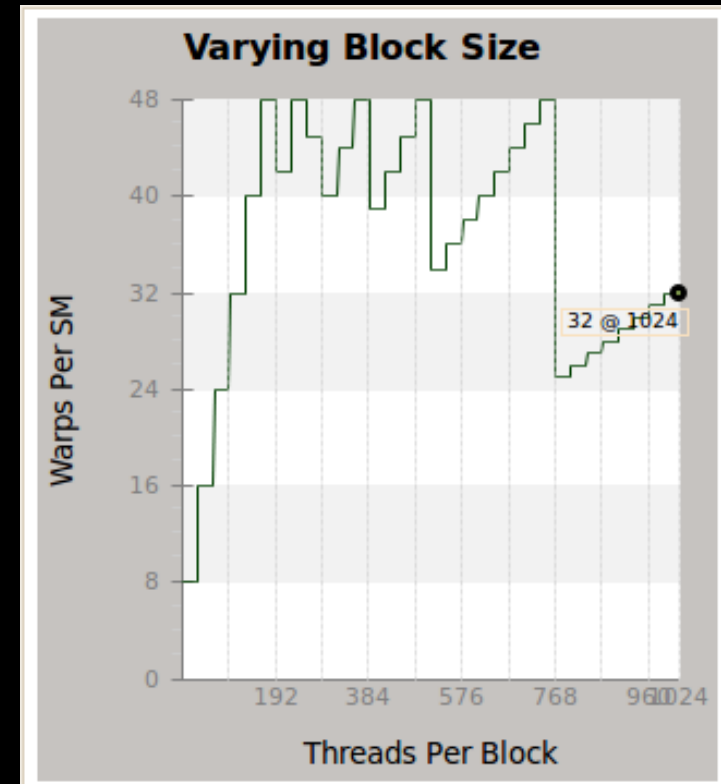
# Occupancy

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



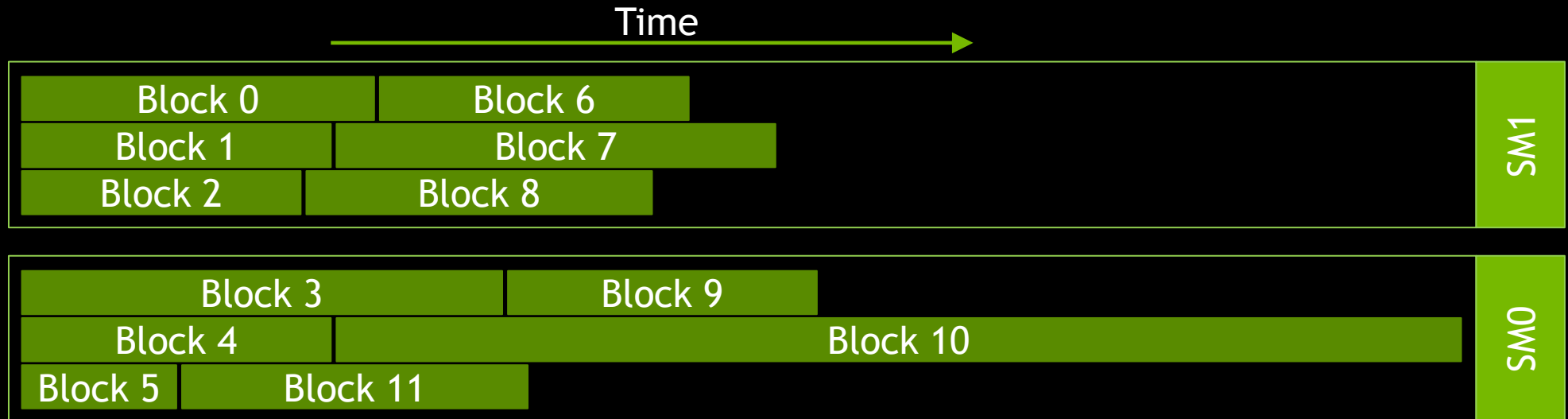
# Low Achieved Occupancy

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



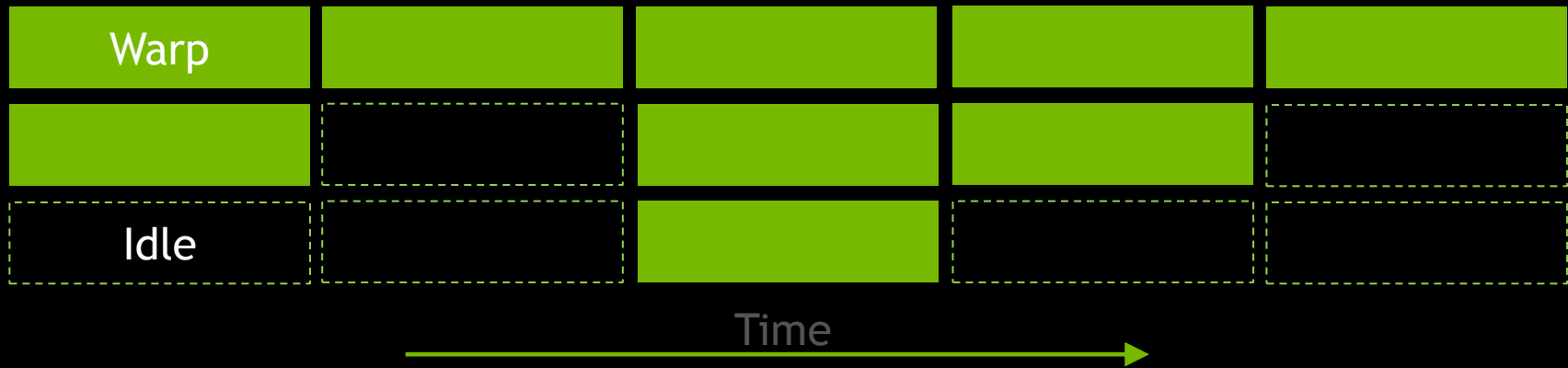
# Low Achieved Occupancy

- 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



# Lots Of Warps, Not Enough Can Execute

- SM can do many things at once...



- No “ready” warp...

