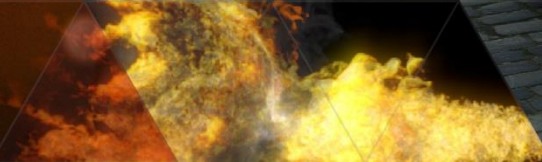
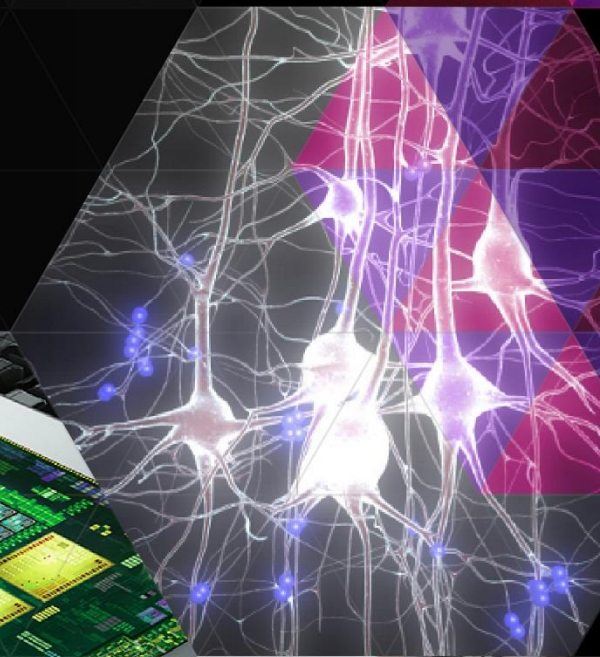
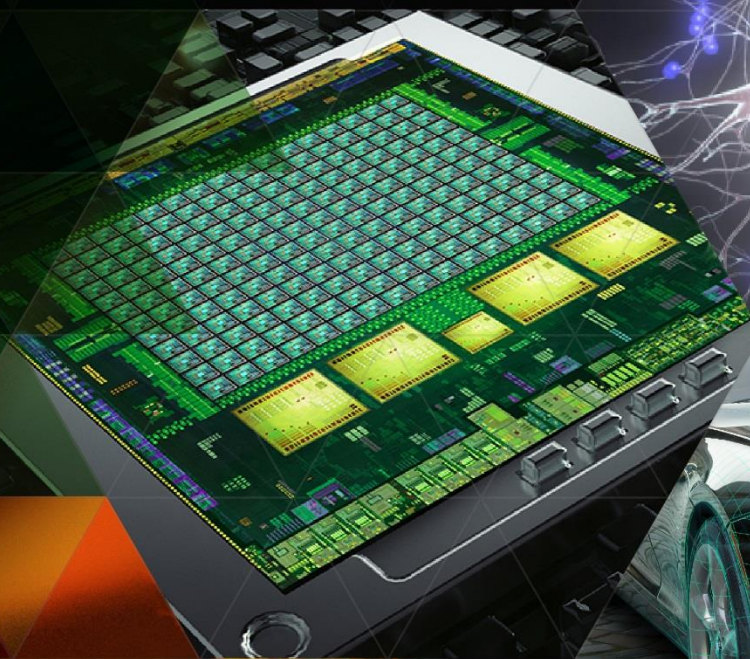




# ADVANCED CUDA

5/11/2015

Peter Messmer - [pmessmer@nvidia.com](mailto:pmessmer@nvidia.com)

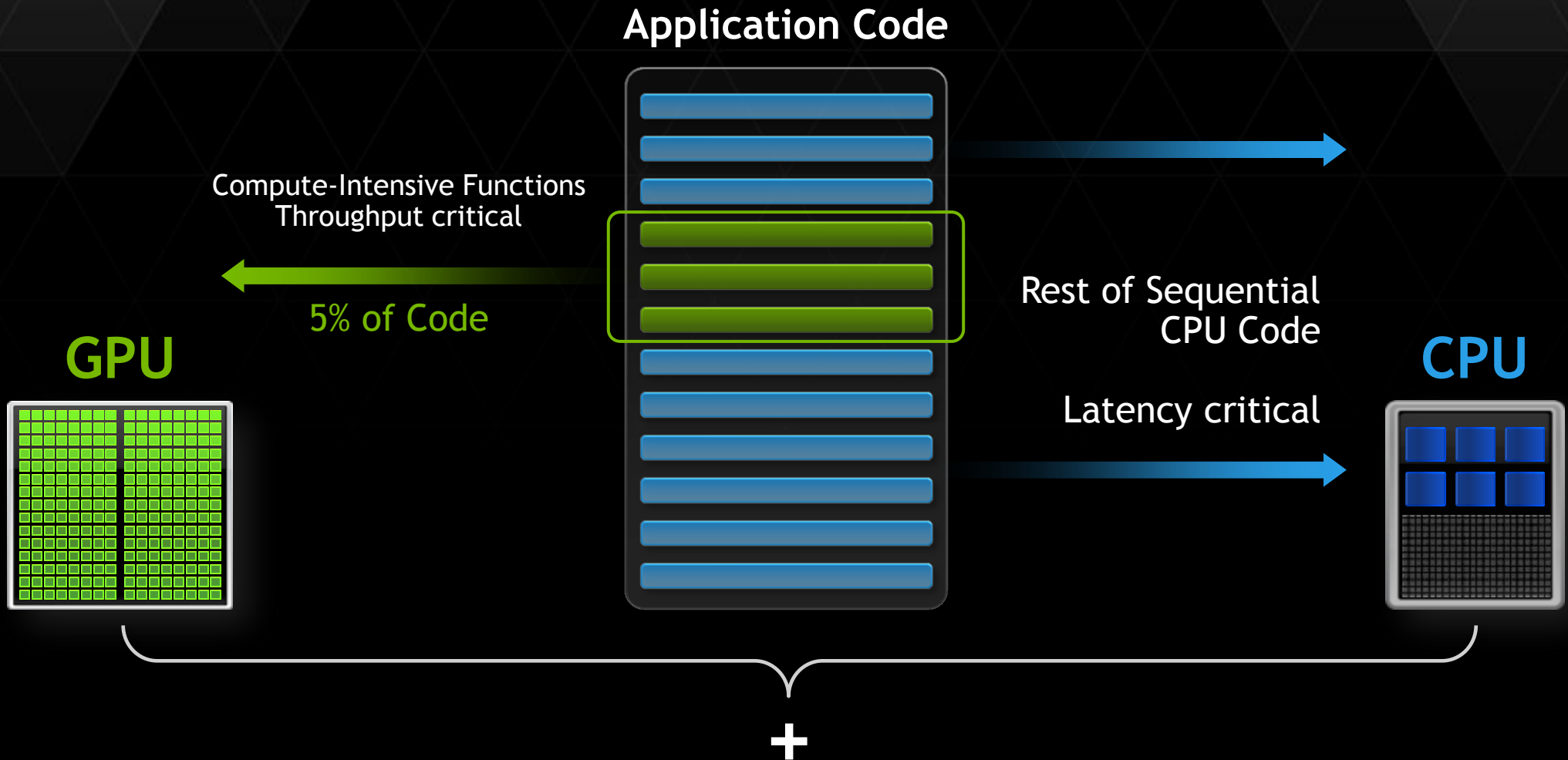




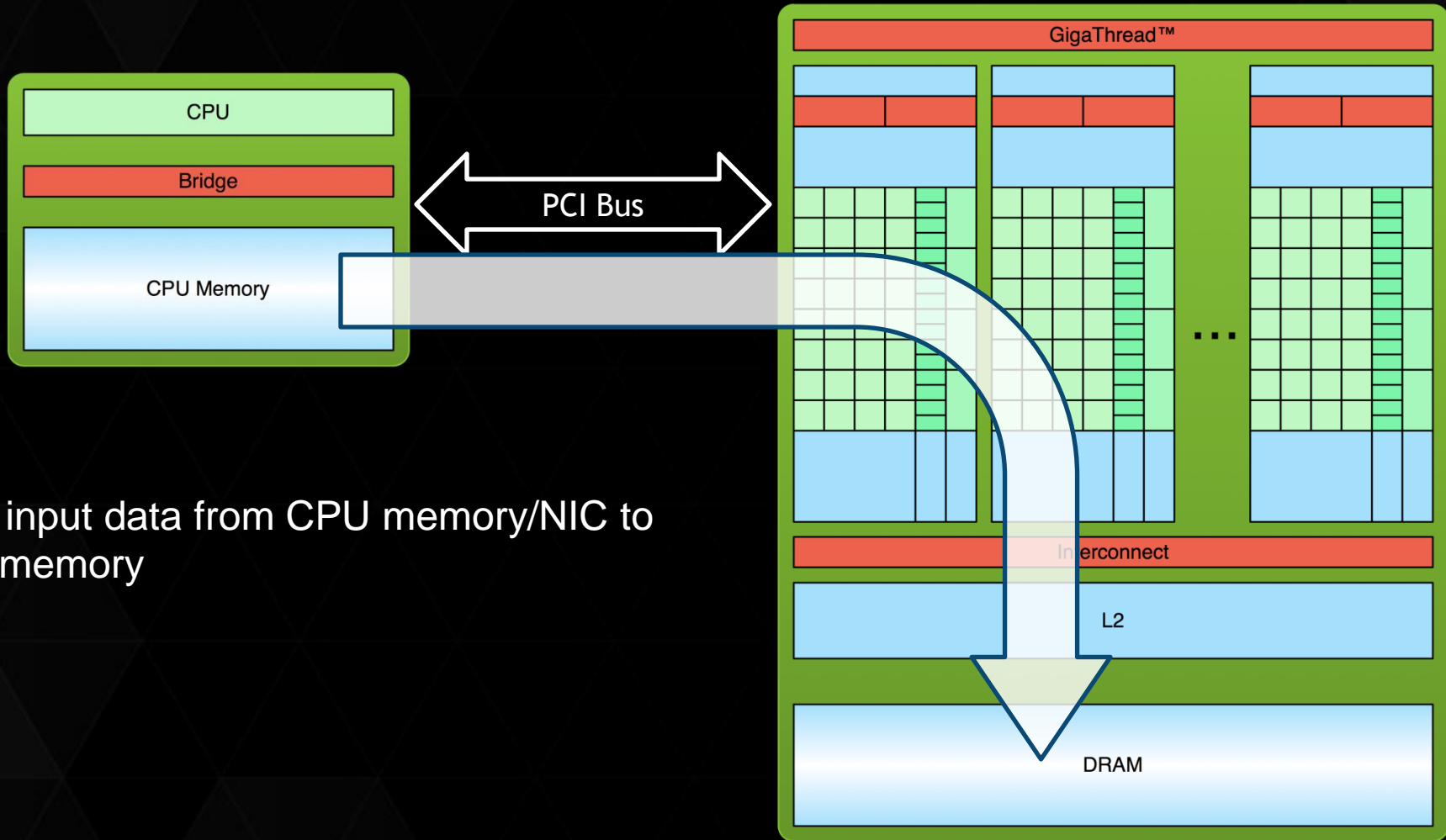
The background features a green grid pattern that is distorted by black wavy shapes, creating a sense of depth and movement. The grid lines are thin and closely spaced, while the black shapes are thick and fluid, resembling liquid or smoke. The overall effect is a dynamic and modern aesthetic.

*What has been covered so far?*

# HOW GPU ACCELERATION WORKS

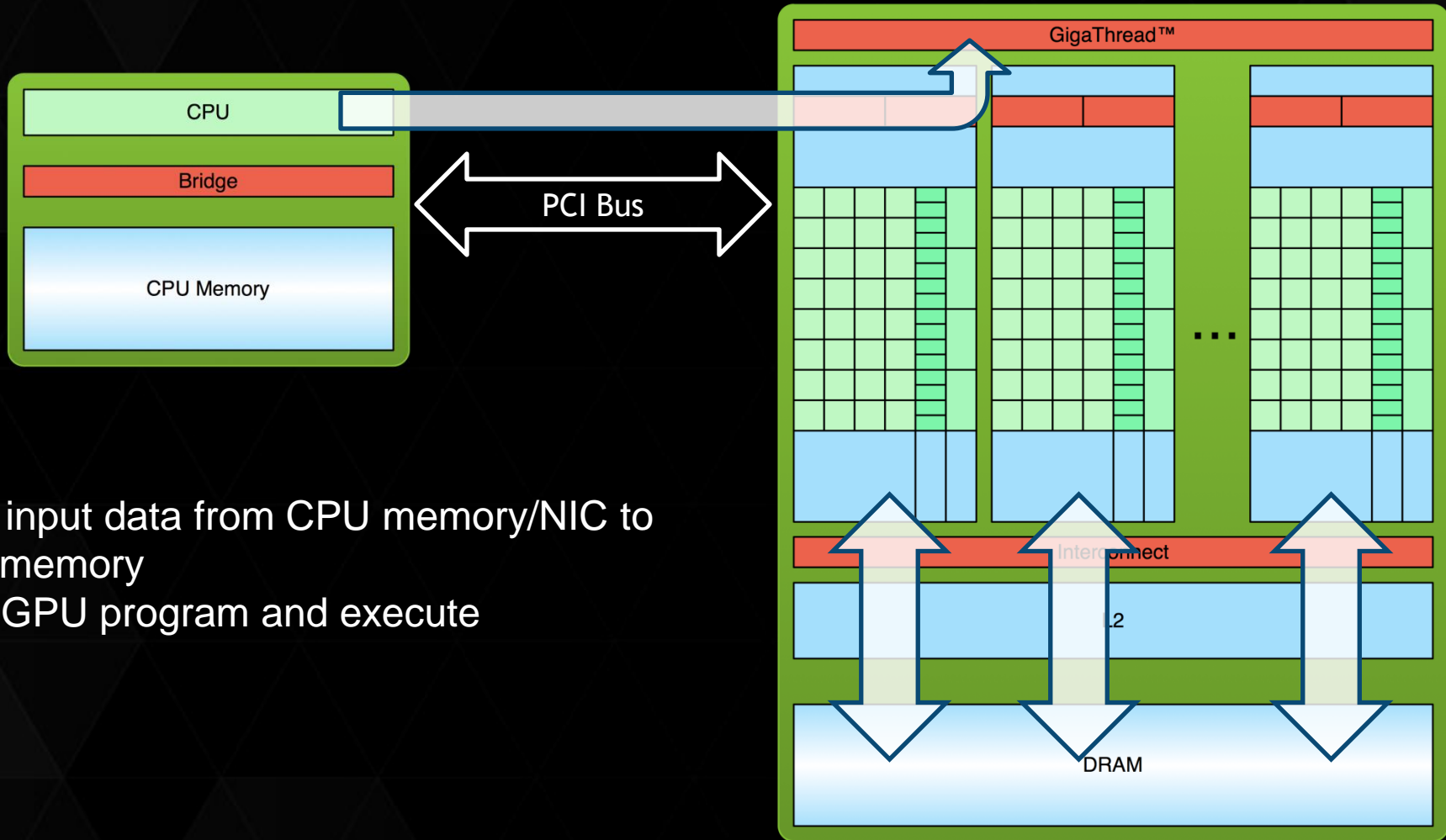


# SIMPLE PROCESSING FLOW



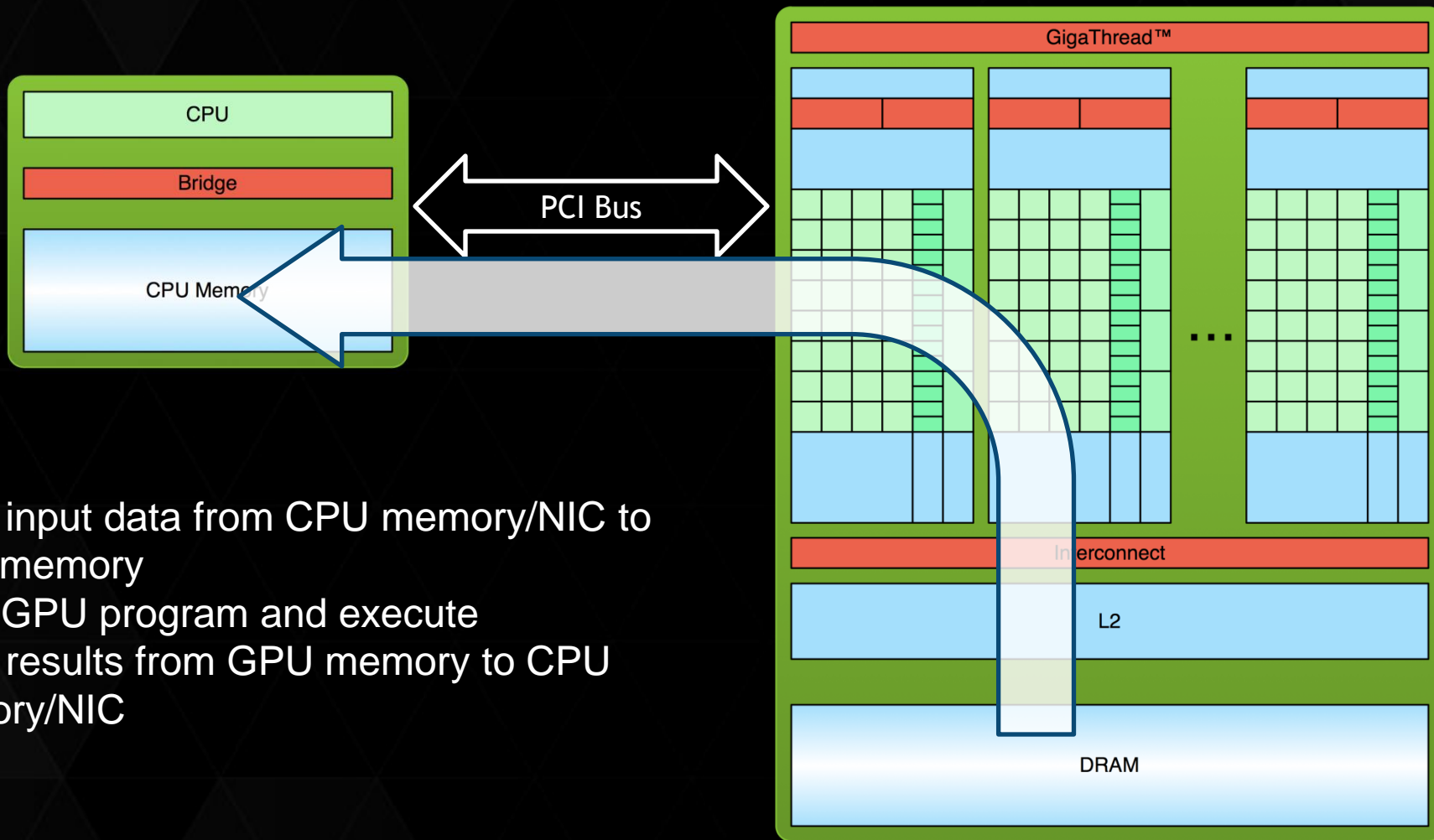
1. Copy input data from CPU memory/NIC to GPU memory

# SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute

# SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC

# 3 WAYS TO PROGRAM GPUS

Applications

Libraries

“Drop-in”  
Acceleration

OpenACC  
Directives

Easily Accelerate  
Applications

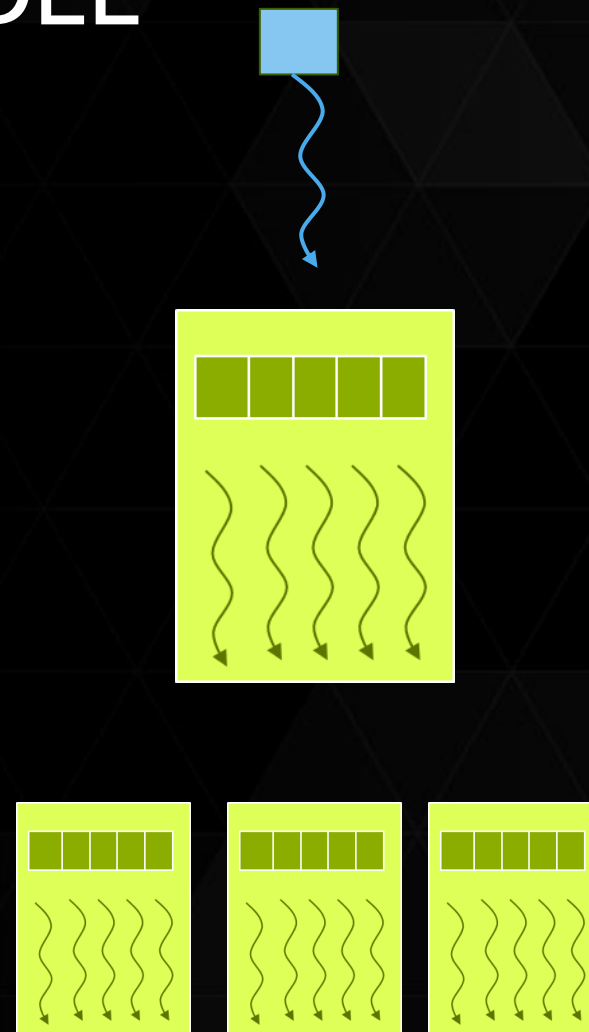
Programming  
Languages

Maximum  
Flexibility



# CUDA EXECUTION MODEL

- ▶ **Thread**: Sequential execution unit
  - ▶ Threads execute in parallel
- ▶ **Thread Block**: a group of threads
  - ▶ Executes on a single Streaming Multiprocessor (SM)
  - ▶ Threads within a block can cooperate
    - ▶ Light-weight synchronization
    - ▶ Data exchange
- ▶ **Grid**: a collection of thread blocks
  - ▶ Thread blocks of a grid execute across multiple SMs
  - ▶ Thread blocks do not synchronize with each other
  - ▶ Communication between blocks is expensive



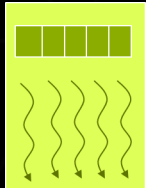


# EXECUTION MODEL

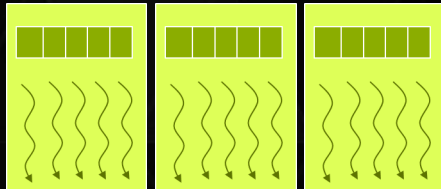
## Software



Thread



Thread Block

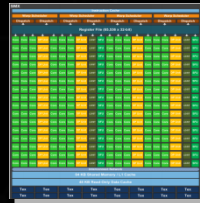


Grid

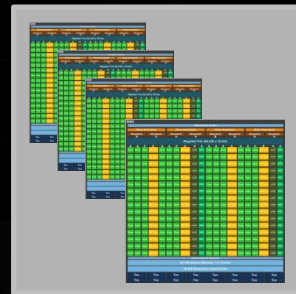
## Hardware



CUDA  
Core



Multiprocessor



Device

Threads are executed by scalar CUDA Cores

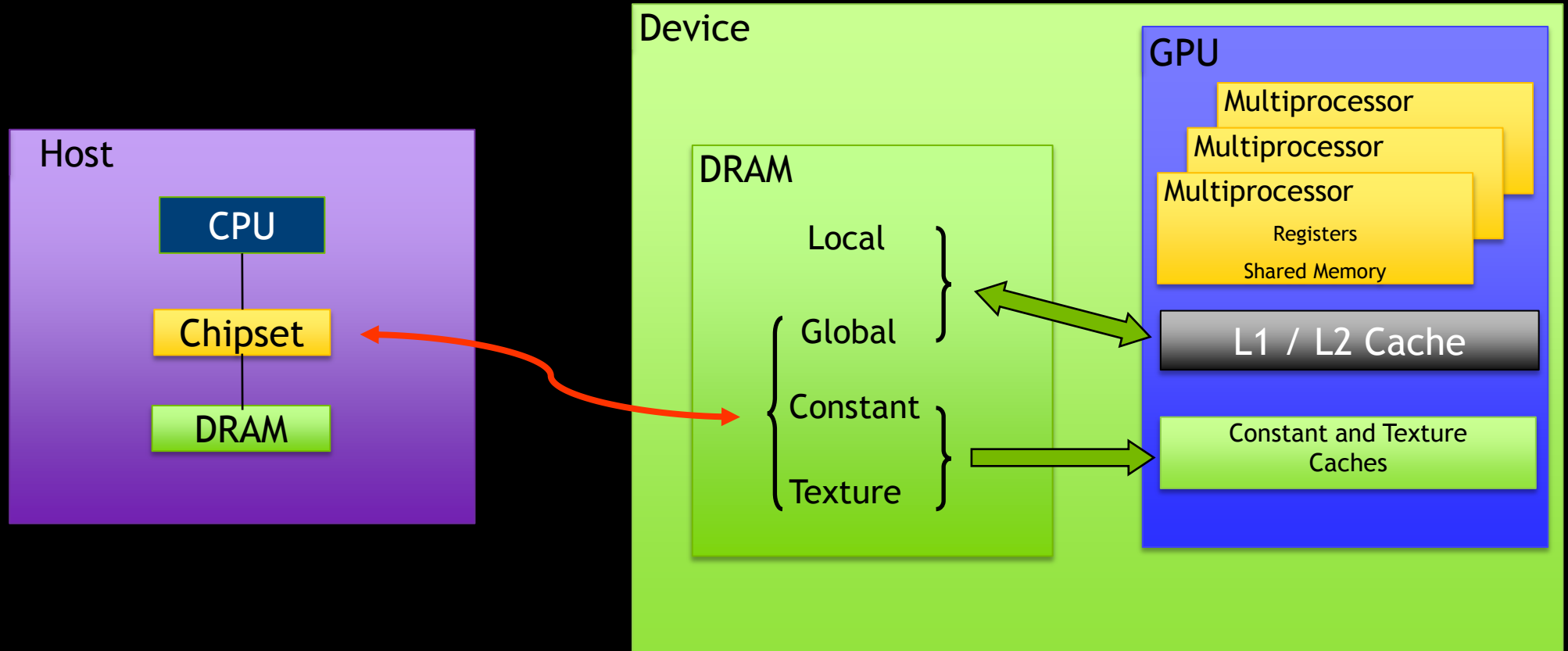
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

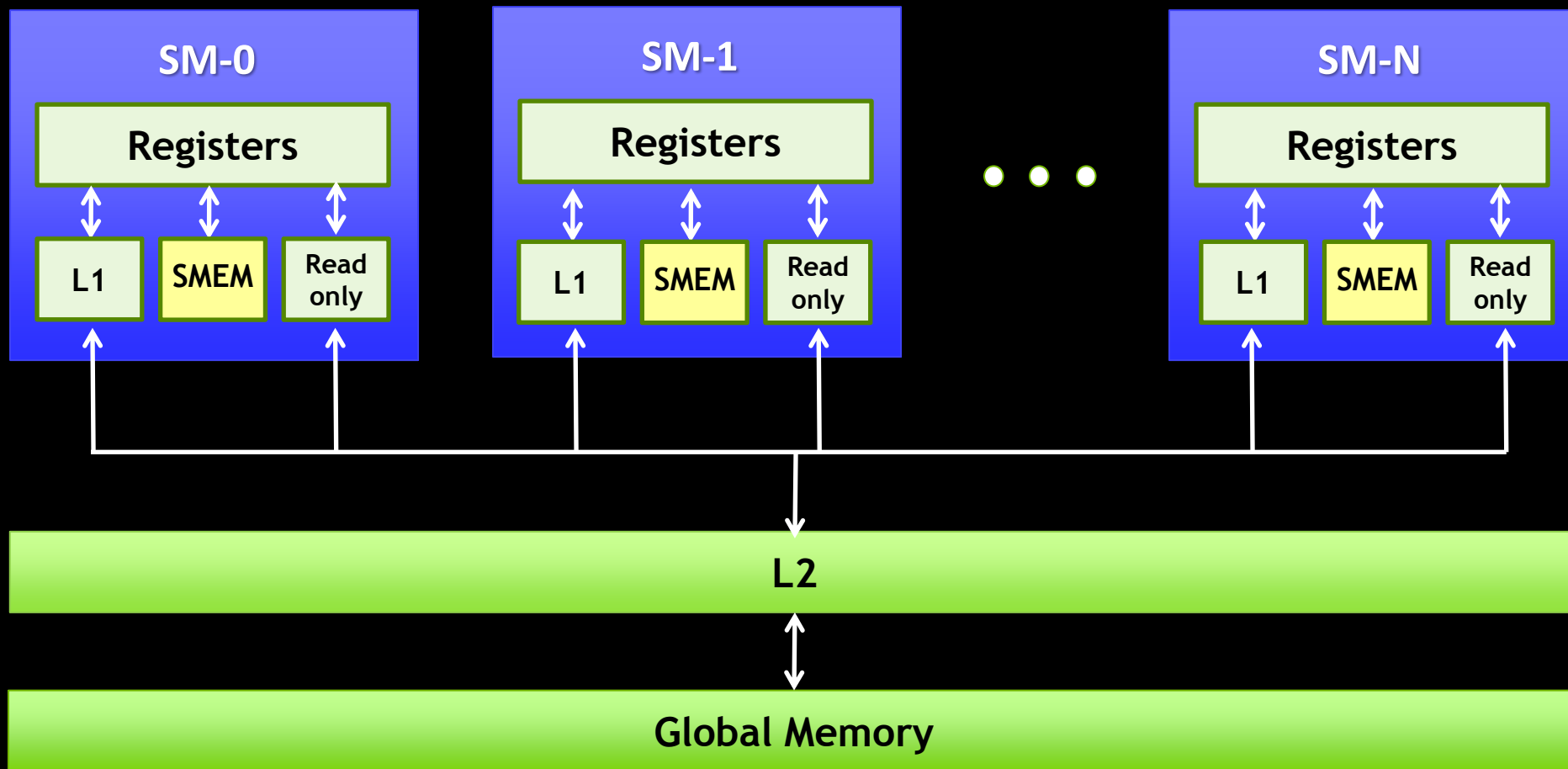
Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

# CUDA Memory Architecture



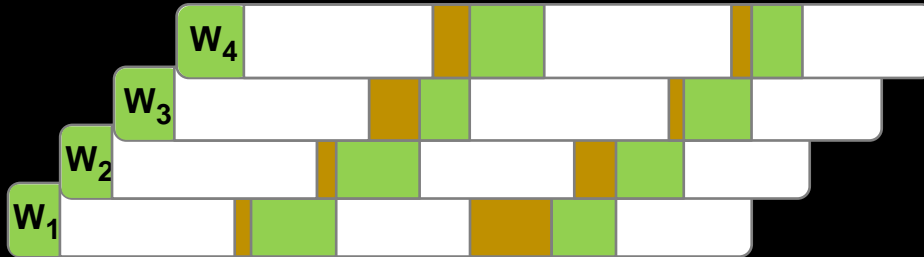
# Kepler Memory Hierarchy



# Low Latency or High Throughput?

- CPU architecture must **minimize latency** within each thread
- GPU architecture **hides latency** with computation from other (warps of) threads

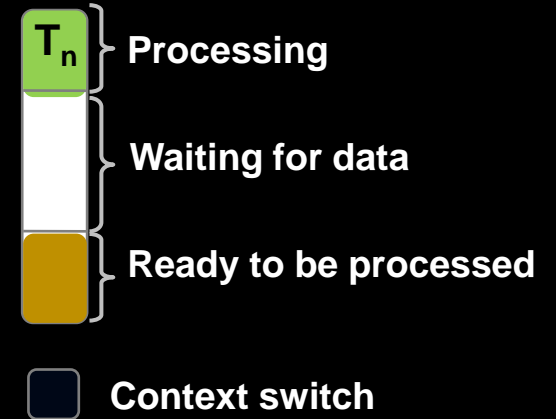
GPU Streaming Multiprocessor – High-throughput Processor



CPU core – Low-latency Processor



Computation Thread/Warp



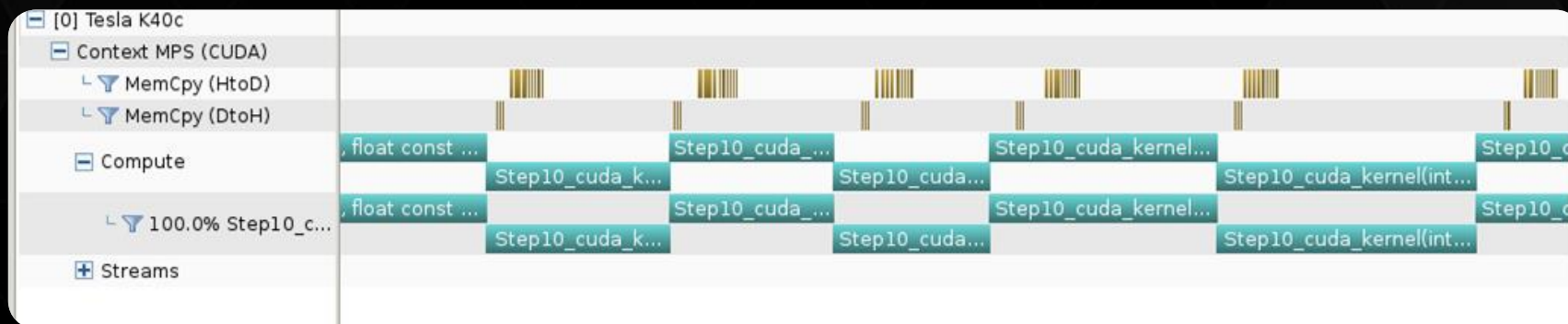


The background features a green grid pattern that is distorted by black wavy shapes, creating a sense of depth and movement.

# *General Optimizations*

# NVVP: NVIDIA'S VISUAL PROFILER

## Timeline



## Guided System

### 1. CUDA Application Analysis

### 2. Performance-Critical Kernels

### 3. Compute, Bandwidth, 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 bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10\_cuda\_kernel" is most likely limited by compute.

#### Perform Compute Analysis

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

#### Perform Latency Analysis

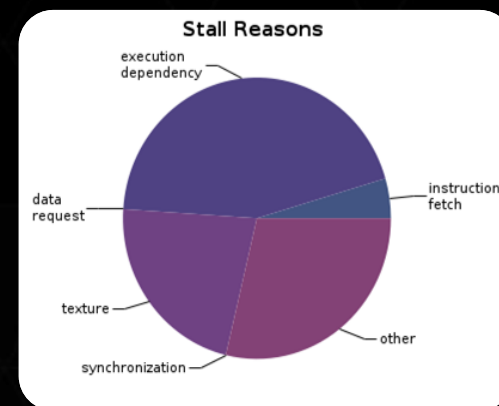
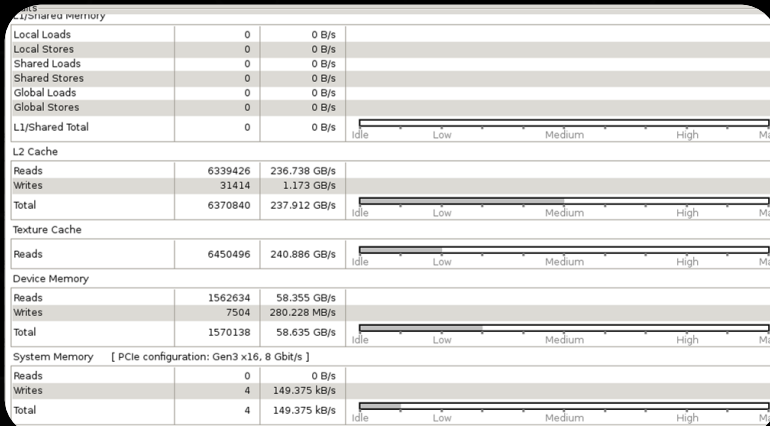
#### Perform Memory Bandwidth Analysis

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

#### Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

## Analysis



# WHICH KERNEL SHOULD WE OPTIMIZE?



# BEFORE OPTIMIZING YOUR KERNELS

- ▶ Always use NVVP to determine if the kernel is the limiter
- ▶ Remember Amdahl:  $S = \frac{1}{(1-P) + \frac{P}{N}} \approx \frac{1}{(1-P)}$
- ▶ Kernels may not always be the limiter





# OPTIMIZE LOCALITY AND CONCURRENCY

- ▶ Manage locality: Move data where it is used
  - ▶ Primary focus of OpenACC
  - ▶ Simplified by unified memory available since CUDA 6
- ▶ Keep both CPU and GPU busy
- ▶ Asynchronous transfers: No need to stall compute for transfer

The background features a green grid pattern that is warped and curved, creating a sense of depth and movement. Overlaid on this grid are several large, solid black, wavy shapes that resemble liquid droplets or flowing forms. The overall aesthetic is modern and technical.

# *Kernel Optimizations*

# KERNEL LAUNCH CONFIGURATION

- A kernel is a function that runs on the GPU
- A kernel is launched as a **grid** of **blocks** of **threads**
- Launch configuration is the number of blocks and number of threads per block, expressed in CUDA with the <<< >>> notation:

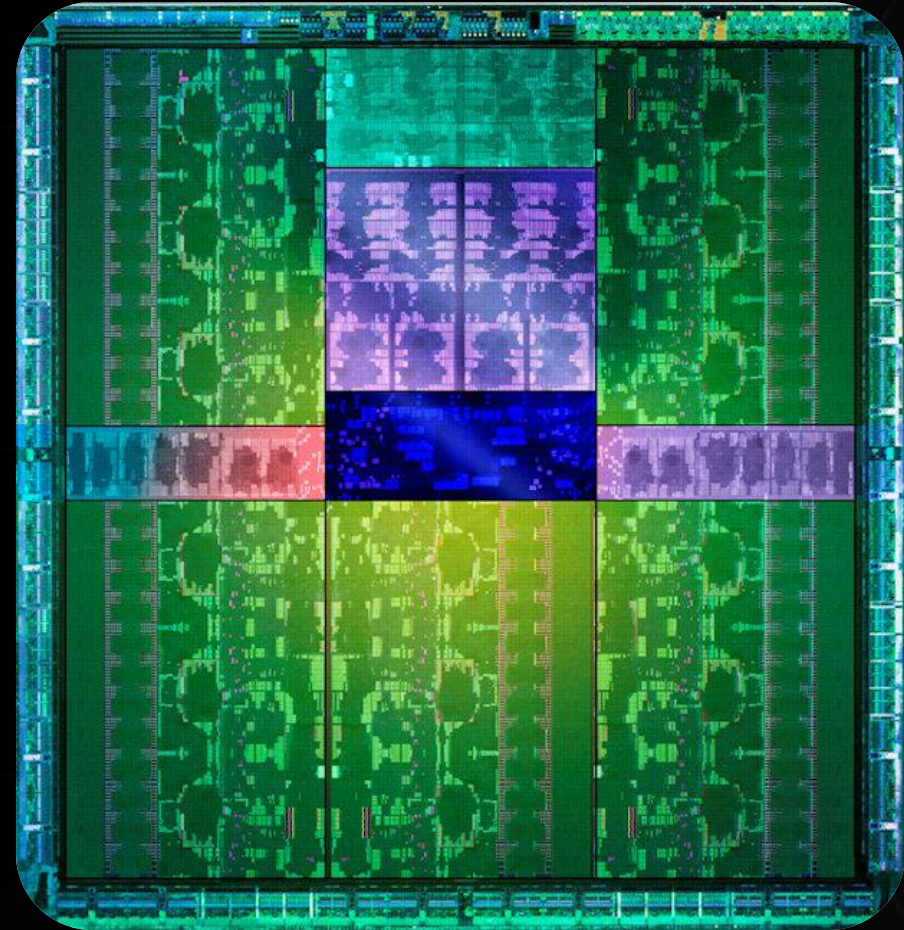
```
mykernel<<<blocks_per_grid, threads_per_block>>> (...);
```

- What values should we pick for these?
  - Need enough total threads to process entire input
  - Need enough threads to keep the GPU busy
  - Selection of block size is an optimization step involving **warp occupancy**



# HIGH-LEVEL VIEW OF GPU ARCHITECTURE

- Several Streaming Multiprocessors
  - E.g., Kepler GK110 has up to 15 SMs
- L2 Cache shared among SMs
- Multiple channels to DRAM



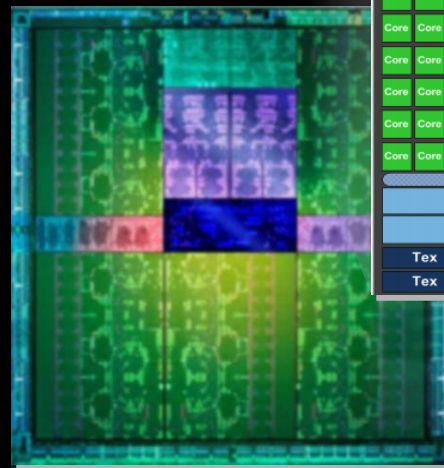
**Kepler GK110**



# KEPLER STREAMING MULTIPROCESSOR (SMX)

## Per SMX:

- 192 SP CUDA Cores
- 64 DP CUDA Cores
- 4 warp schedulers
  - Up to 2048 concurrent threads
  - One or two instructions issued per scheduler per clock from a single warp
- Register file (256KB)
- Shared memory (48KB)



# LAUNCH CONFIGURATION: GENERAL GUIDELINES

How many blocks should we use?

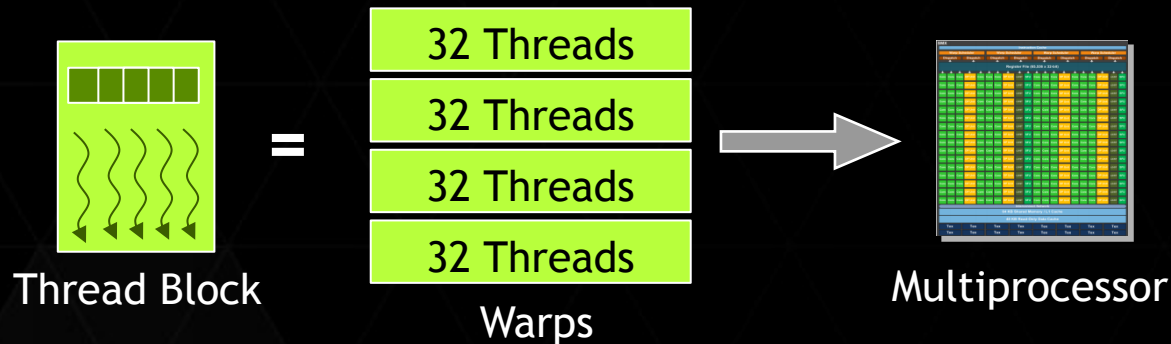
- 1,000 or more thread blocks is best
  - Rule of thumb: enough blocks to fill the GPU at least 10s of times over
  - Makes your code ready for several generations of future GPUs

# LAUNCH CONFIGURATION: GENERAL GUIDELINES

How many threads per block should we choose?

- The really short answer: 128, 256, or 512 are often good choices
- The slightly longer answer:
  - Pick a size that suits the problem well
  - Multiples of 32 threads are best
  - Pick a number of threads per block (and a number of blocks) that is sufficient to keep the SM busy

# WARPS



A thread block consists of *warps of 32 threads*

A warp is executed physically in parallel on some multiprocessor.



Threads of a warp issue instructions in lock-step (as with SIMD)



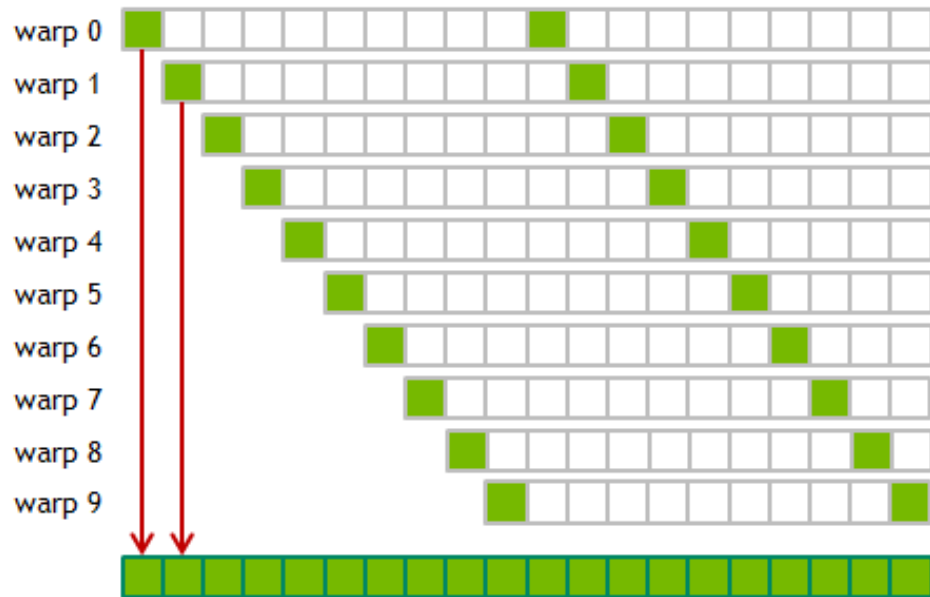
# CONCURRENCY OFFERED BY A K20X (GK110)

Number of SMX	:	14 (15 on K40, K80)
Number of warps per SM	:	64
Number of threads/warp	:	32
Warps per device	:	$14 * 64 = 896$
Active threads per device	:	$14 * 64 * 32 = 28'672$

# LATENCY HIDING

-  The warp issues
-  The warp waits (latency)

## Fully covered latency



## Exposed latency



# Occupancy

- Need enough concurrent warps per SM to **hide latencies**:
  - Instruction latencies
  - Memory access latencies
- Hardware resources determine number of warps that fit per SM

$$\text{Occupancy} = N_{\text{actual}} / N_{\text{max}}$$

Start	588.755 ms
End	588.808 ms
Duration	53.344 $\mu$ s
Grid Size	[ 64,64,1 ]
Block Size	[ 16,8,1 ]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	17.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	17.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	91.3%
Theoretical	100%
Throughput	100%
Accessed	21.3%
Occupancy	
Throughput	0%

# Occupancy Limiters

- Full occupancy: Maximum choice for scheduler

- Hardware limits

- Registers per thread
- Shared memory per thread block
- Threads per thread block
- Thread blocks per SMX

## Kepler SM resources:

- 64K 32-bit registers
- Up to 48 KB of shared memory
- Up to 2048 concurrent threads
- Up to 16 concurrent thread blocks

- Optimal choice: balance resource consumption and concurrency

# Occupancy and Performance

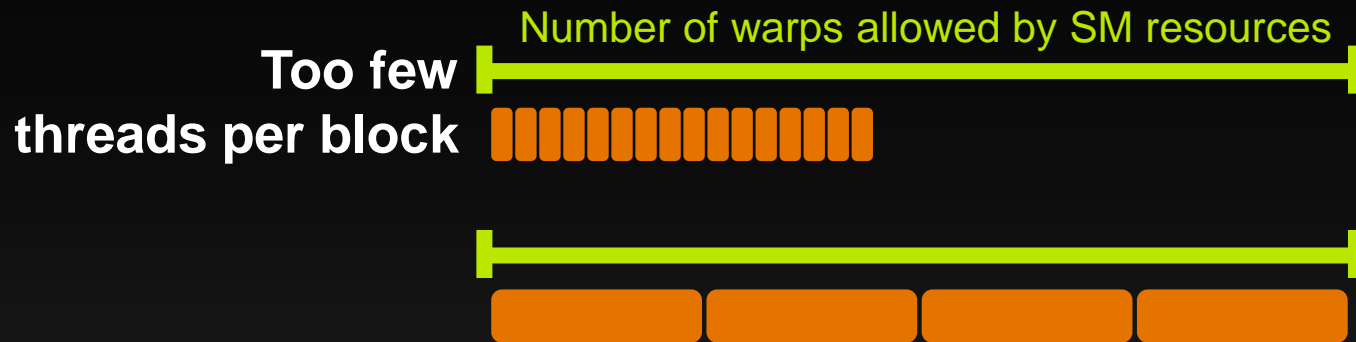
- Note that 100% occupancy isn't needed to reach maximum performance
  - Once the “needed” occupancy (enough warps to switch among to cover latencies) is reached, further increases won't improve performance
- Level of occupancy needed depends on the code
  - More independent work per thread -> less occupancy is needed
  - Memory-bound codes tend to need more occupancy
    - Higher latency than for arithmetic, need more work to hide it

# Thread Block Size and Occupancy

- Thread block size is a multiple of warp size (32)
  - Even if you request fewer threads, hardware rounds up
- Thread blocks can be too small
  - Kepler SM can run up to 16 thread blocks concurrently
  - SM can reach the block count limit before reaching good occupancy
    - E.g.: 1-warp blocks = 16 warps/SM on Kepler (25% occ - probably not enough)
- Thread blocks can be too big
  - Enough SM resources for more threads, but not enough for a whole block
  - A thread block isn't started until resources are available for all of its threads



# Thread Block Sizing



## SM resources:

- Registers
- Shared memory



# CUDA Occupancy Calculator

- Analyze effect of resource consumption on occupancy

## CUDA GPU Occupancy Calculator

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

1.) Select Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:			(Help)
Threads Per Block		256	
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:			(Help)
Active Threads per Multiprocessor		2048	
Active Warps per Multiprocessor		64	
Active Thread Blocks per Multiprocessor		8	
Occupancy of each Multiprocessor		100%	

Physical Limits for GPU Compute Capability: 3.5	
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	128	16
Shared Memory (Bytes)	4096	49152	12

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor			Blocks/SM * Warps/Block = Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	16		
Limited by Shared Memory per Multiprocessor	12		

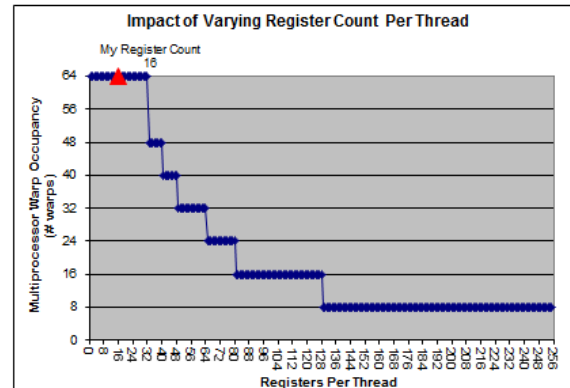
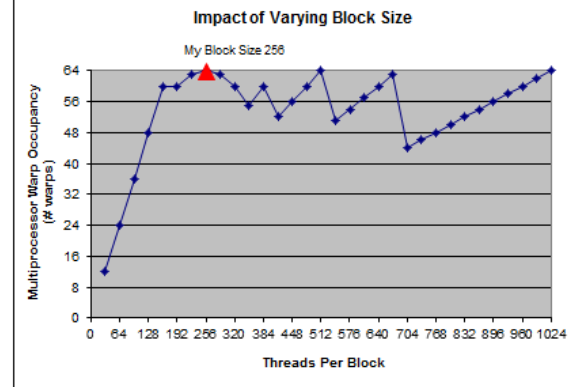
Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64  
Occupancy = 64 / 64 = 100%

[Click Here for detailed instructions on how to use this occupancy calculator.](#)

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



The background features a green grid pattern that is distorted by a wavy, undulating effect, creating a sense of depth and movement. Overlaid on this grid are several large, solid black, wavy shapes that resemble stylized hills or clouds, adding a layer of complexity to the design.

*Thank You!*