CUDA
Overview, New Features and Optimization

Peter Messmer

GPU Programming Workshop

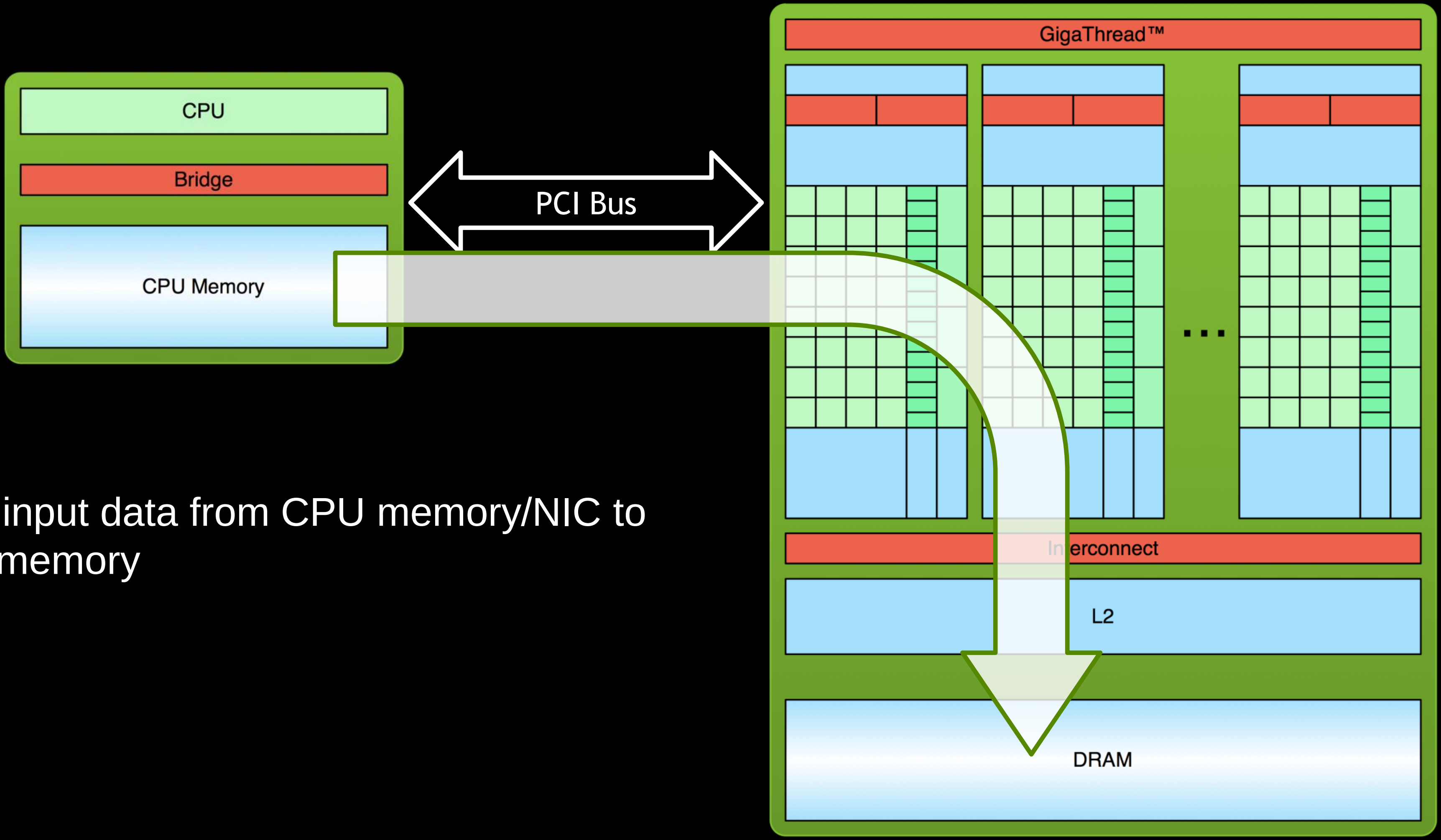
RZG, Garching, 14-15 May, 2014

Agenda: Wednesday, May 14, 2014

13:30-13:45	CUDA Refresher
13:45-14:00	Dynamic Parallelism
14:00-14:30	Unified Memory
14:30-15:00	Break
15:00-16:30	GPU Optimization

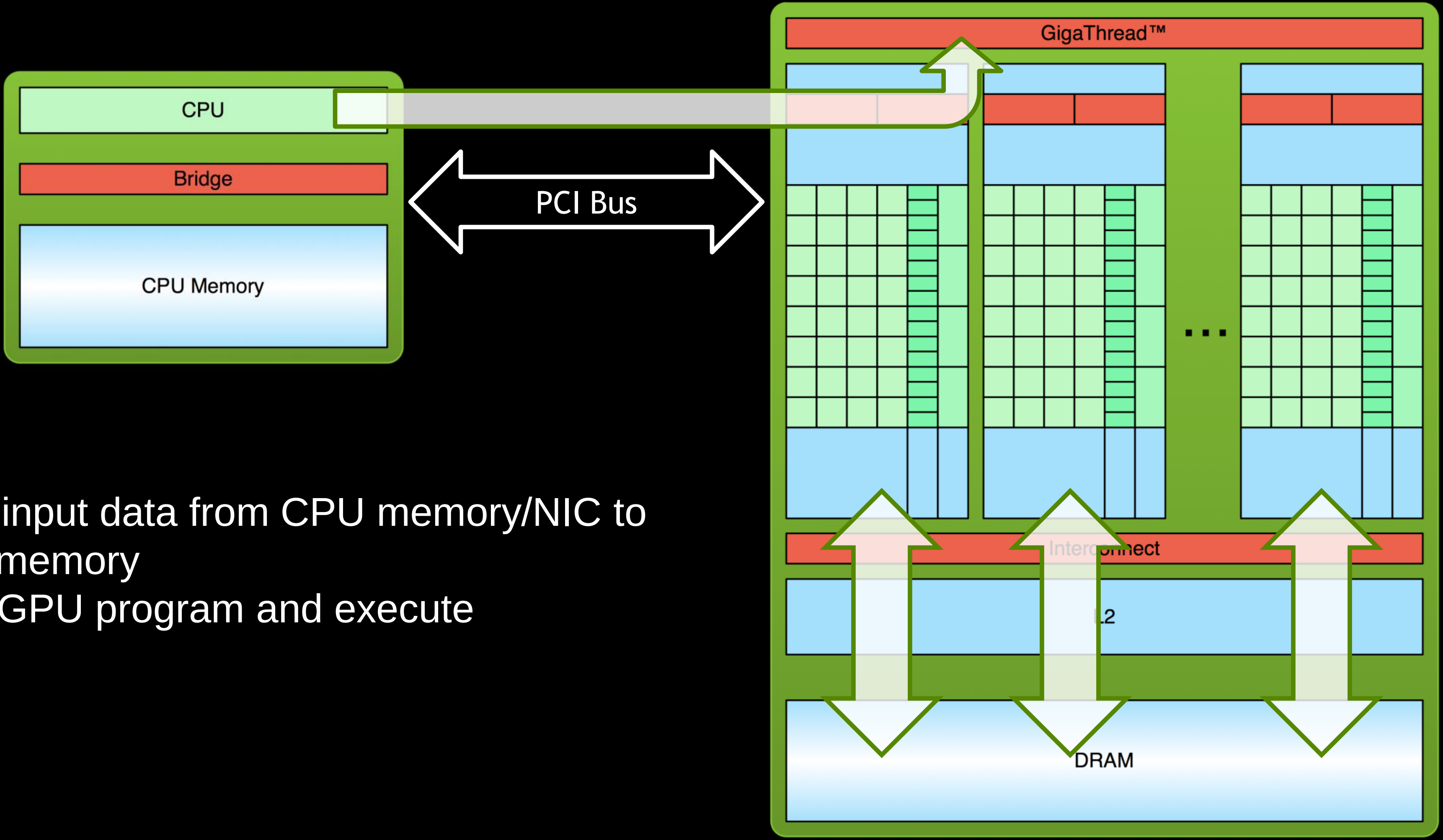
CUDA in 5 Slides

Simple Processing Flow



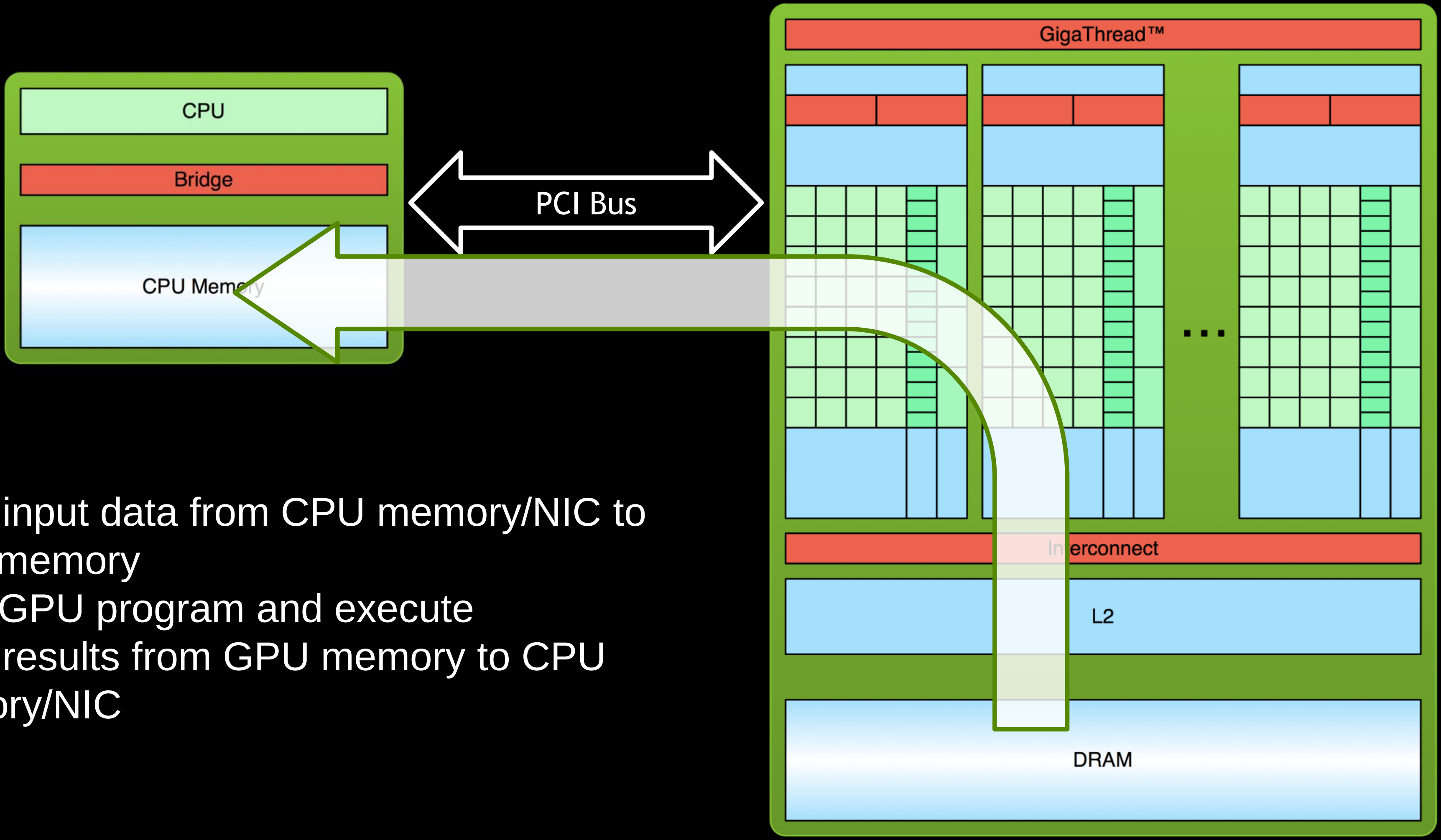
1. Copy input data from CPU memory/NIC to GPU memory

Simple Processing Flow



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

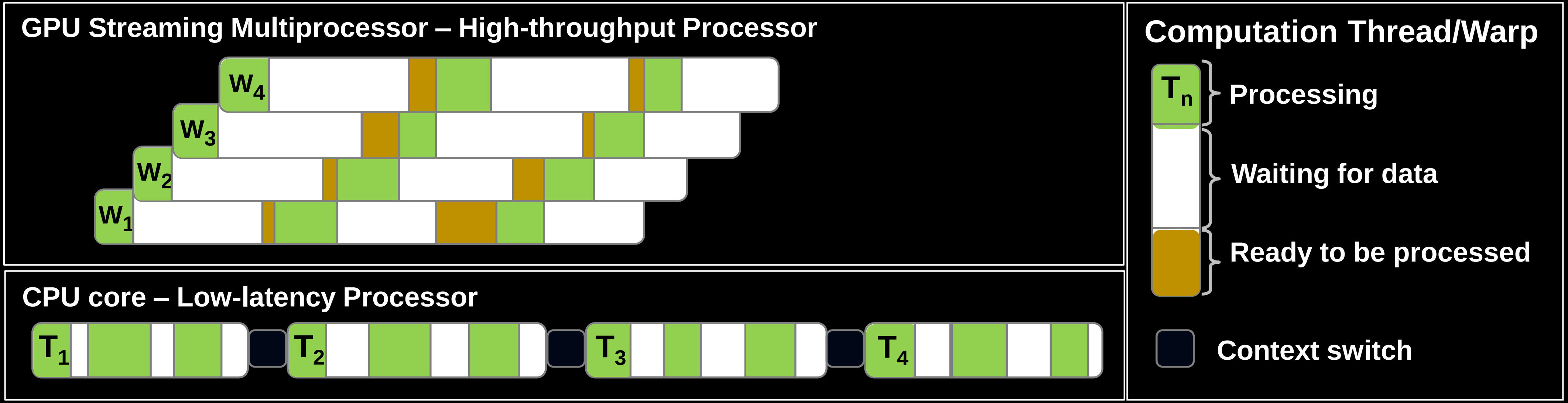
Simple Processing Flow



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

Low Latency or High Throughput?

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



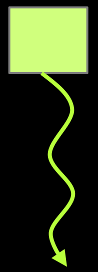
```
#include <cuda.h>

__global__ void kernel(float* x, int n){
     int tid = threadIdx.x;
     if(threadIdx.x < n) x[tid] = x[tid] + (float) tid ;
}

int main(int argc, char** argv){
     const int n = 100;
     float *d_x; float *h_x = (float*) malloc(n * sizeof(float));
…
     cudaMemcpy(d_x, h_x, n*sizeof(float), cudaMemcpyHostToDevice);
     kernel<<<1, n>>>(d_x, n);
     cudaMemcpy(h_x, d_x, n*sizeof(float), cudaMemcpyDeviceToHost);
..
}
```

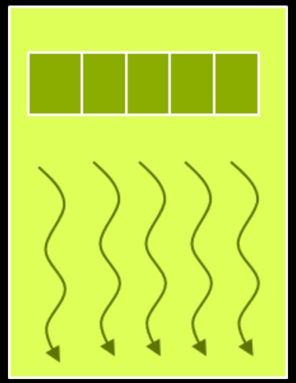
CUDA Execution Model

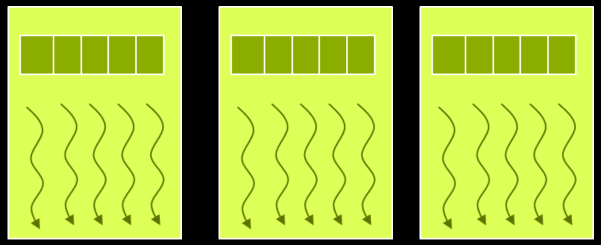
- Thread: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel

- Threads Block: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange

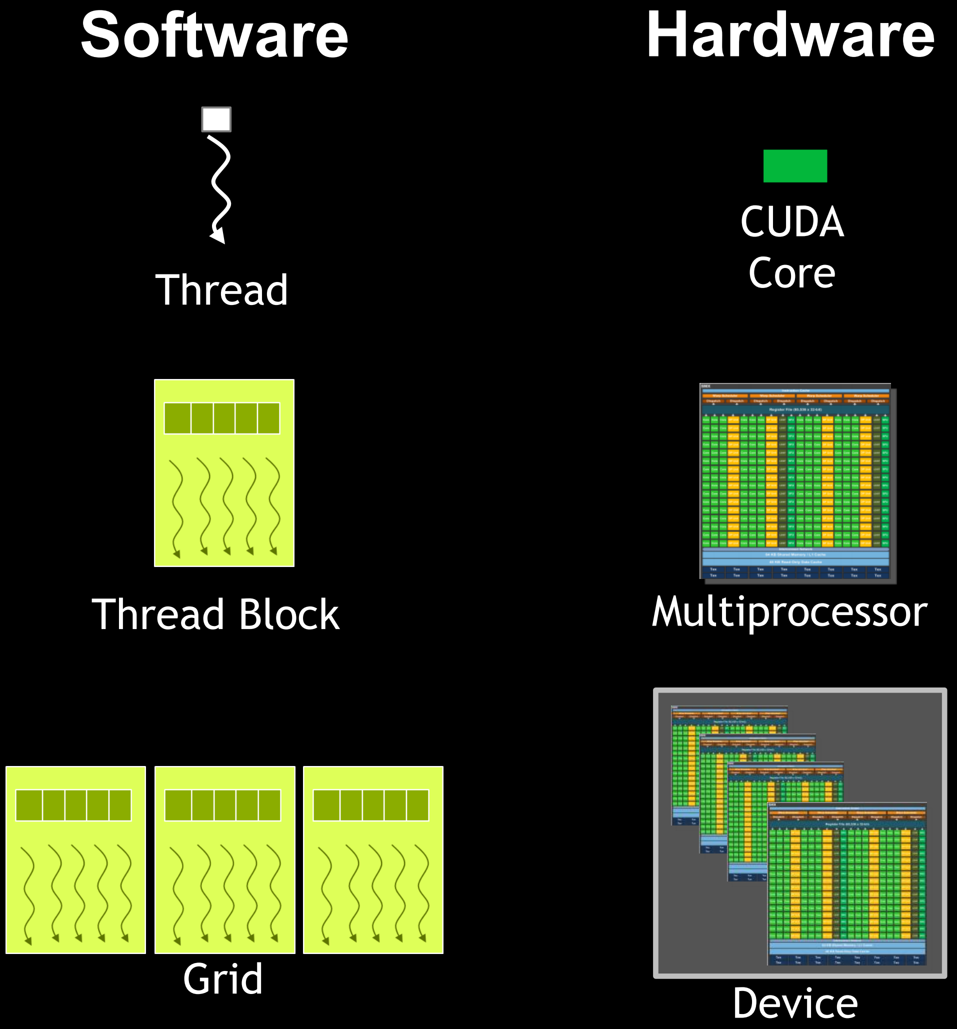
- Grid: a collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive







Execution Model



Threads are executed by scalar CUDA Cores

Thread blocks are executed on multiprocessors

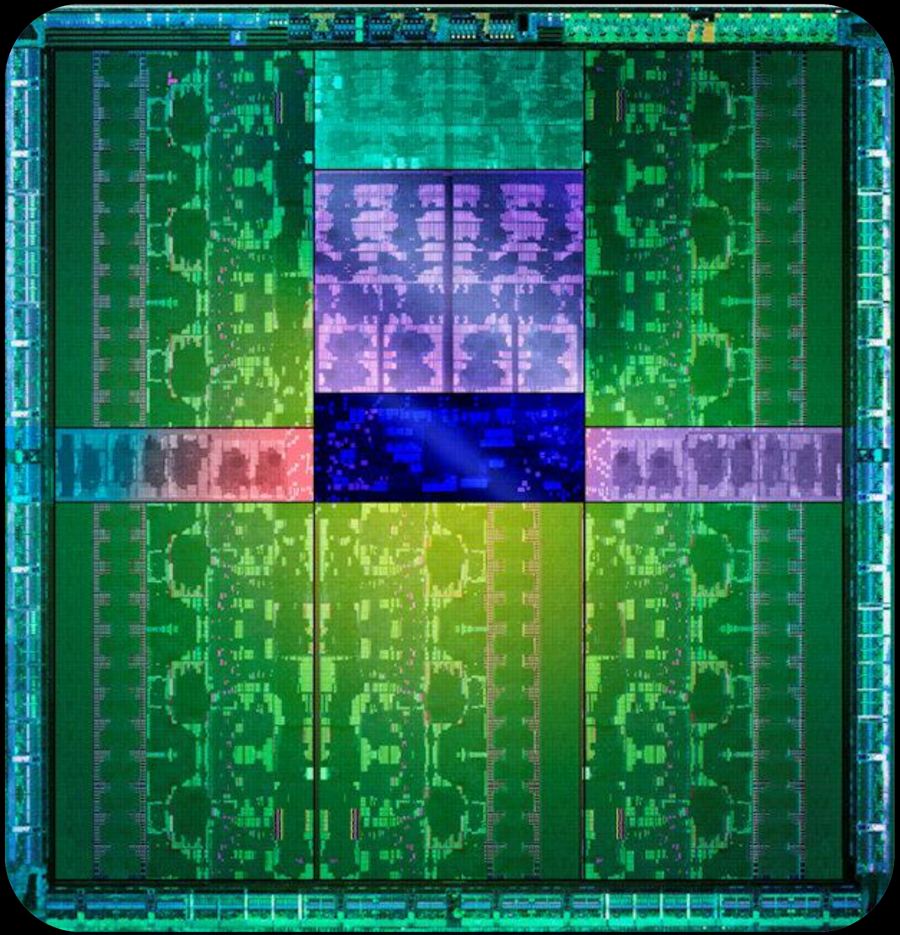
Thread blocks do not migrate

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

High-level view of GPU Architecture

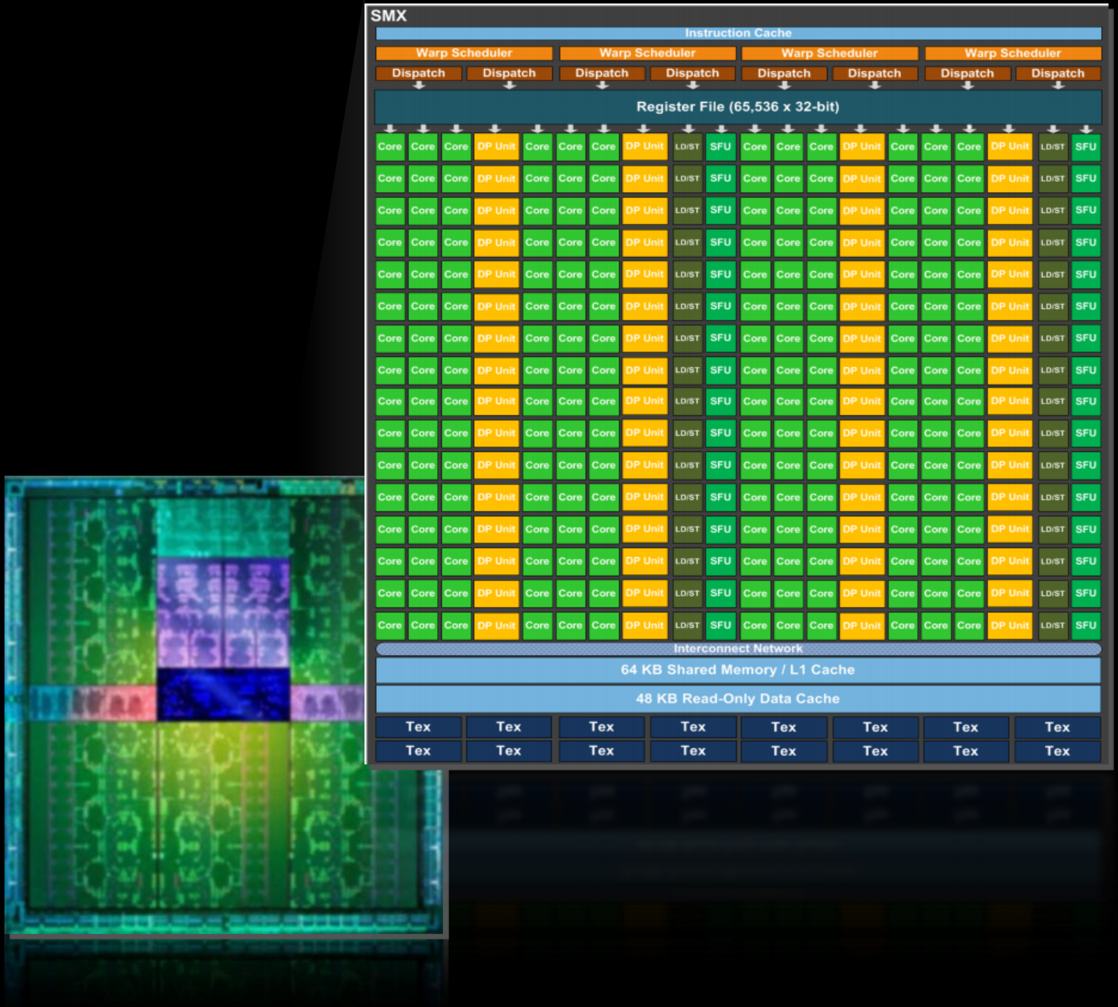
- Several Streaming Multiprocessors
  - E.g., Kepler GK110 has up to 15 SMs
- L2 Cache shared among SMs
- Multiple channels to DRAM


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)



Dynamic Parallelism

CUDA Dynamic Parallelism

Kernel launches grids

Identical syntax as host

CUDA runtime function in
`cudadevrt` library

Enabled via nvcc flag

`-rdc=true`

```
__global__ void childKernel()
{
 printf("Hello %d", threadIdx.x);
}
```

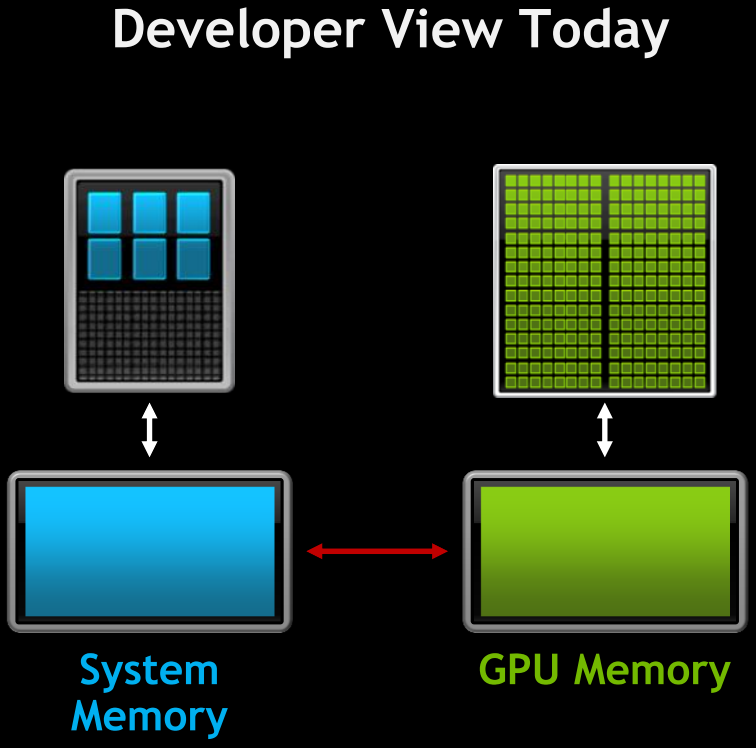
```
__global__ void parentKernel()
{
    childKernel<<<1,10>>>();
    cudaDeviceSynchronize();
    printf("World!\n");
}
```

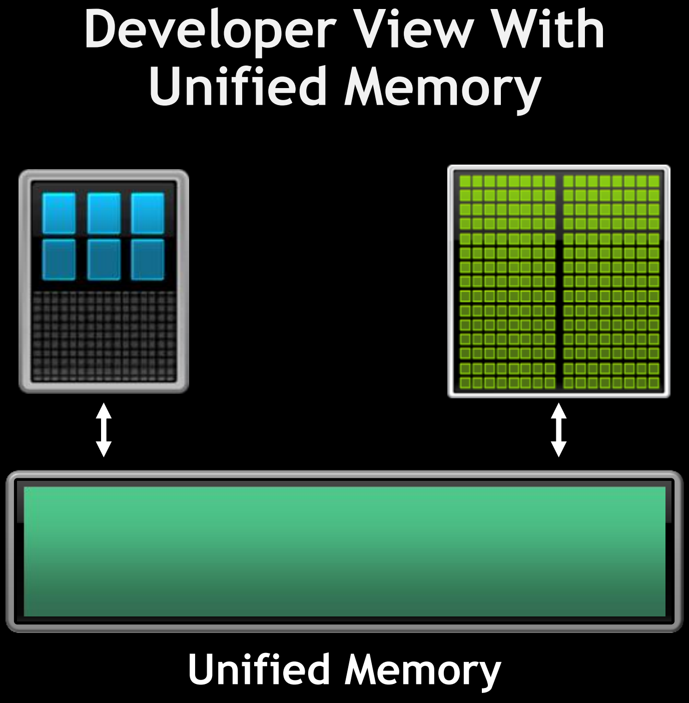
```
int main(int argc, char *argv[])
{
  parentKernel<<<1,1>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Unified Memory in Depth

Peter Messmer

Unified Memory
Dramatically Lower Developer Effort





Super Simplified Memory Management Code

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

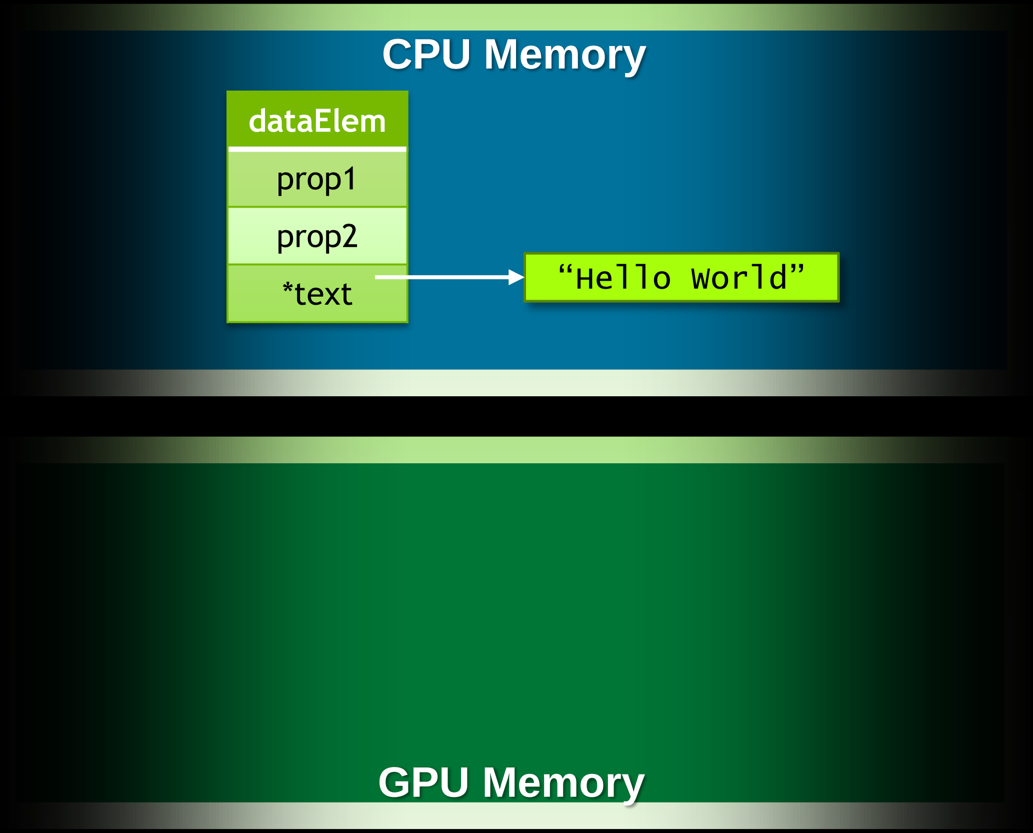
Unified Memory Delivers

1. Simpler Programming & Memory Model
   - Single pointer to data, accessible anywhere
   - Tight language integration
   - Greatly simplifies code porting

2. Performance Through Data Locality
   - Migrate data to accessing processor
   - Guarantee global coherency
   - Still allows *cudaMemcpyAsync()* hand tuning

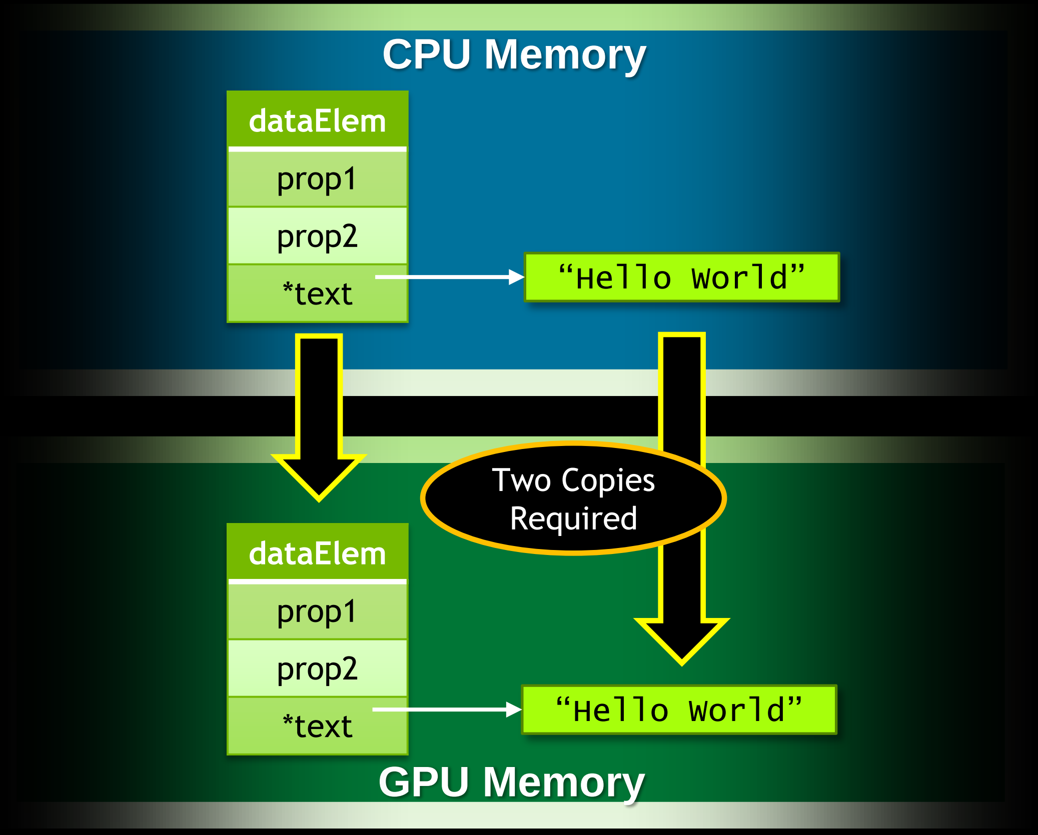
Simpler Memory Model:
Eliminate Deep Copies

```
struct dataElem
{
    int prop1;
    int prop2;
    char *text;
};
```



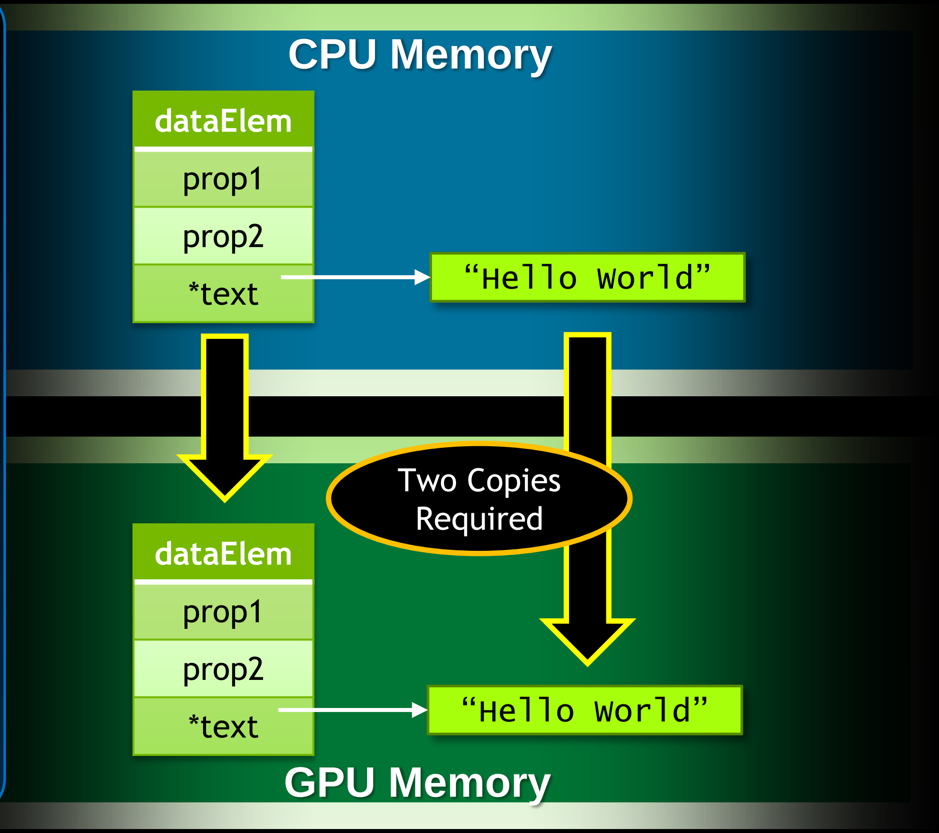
Simpler Memory Model:
Eliminate Deep Copies

```
struct dataElem
{
    int prop1;
    int prop2;
    char *text;
};
```



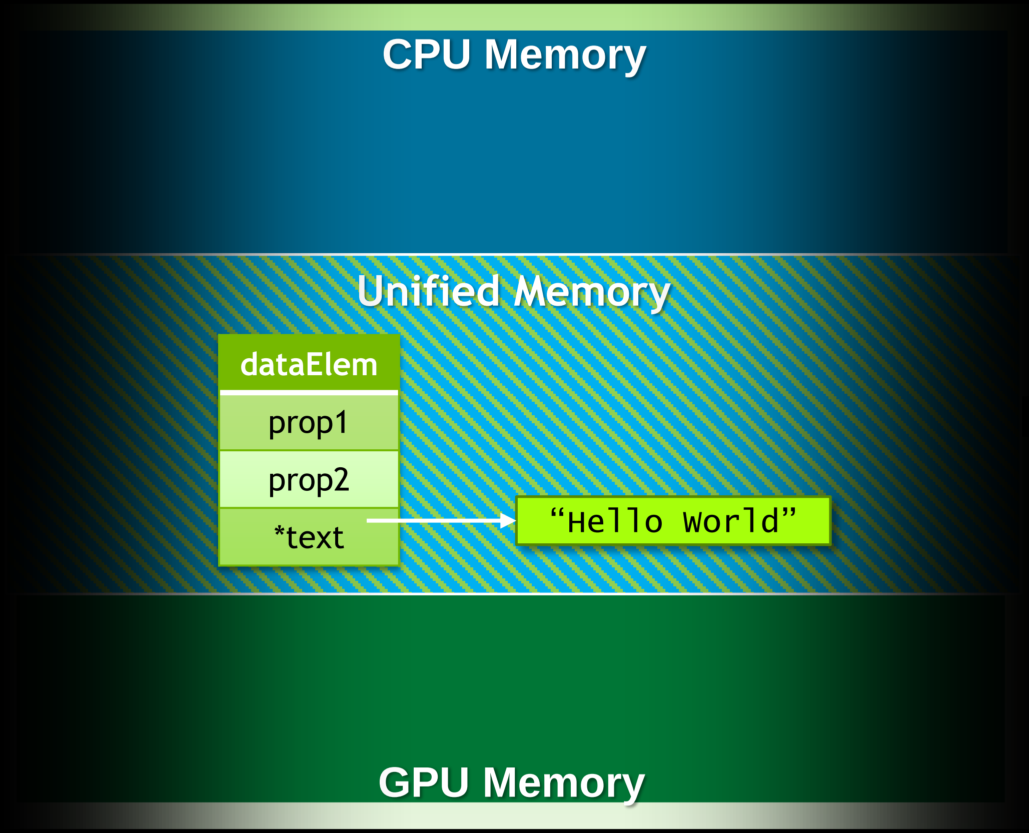
Simpler Memory Model:
Eliminate Deep Copies

```
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int textlen = strlen(elem->text) + 1;

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, textlen);

    // Copy up each piece separately, including
    // new “text” pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, textlen);
    cudaMemcpy(&(g_elem->text), &g_text,
                                 sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU & GPU use different copies of “elem”
    kernel<<< ... >>>(g_elem);
}
```



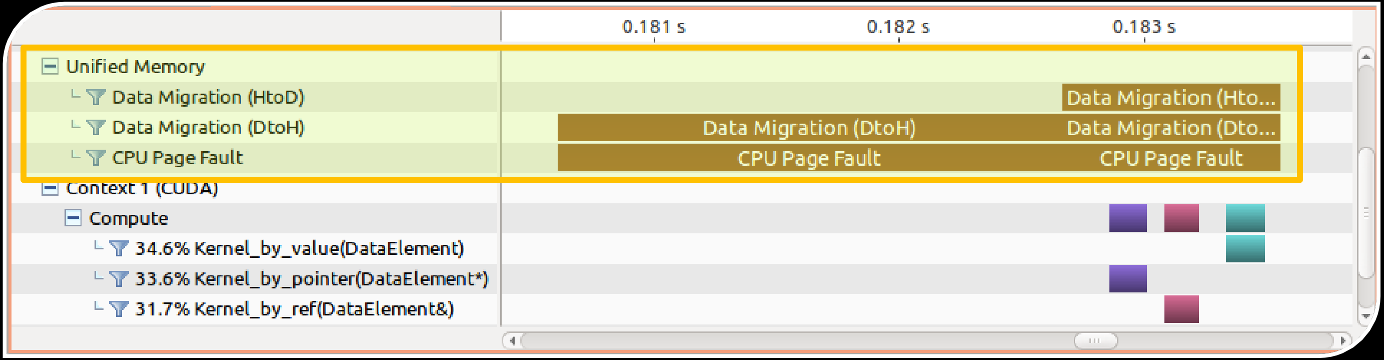
Simpler Memory Model:
Eliminate Deep Copies

```
void launch(dataElem *elem) {
    kernel<<< ... >>>(elem);
}
```

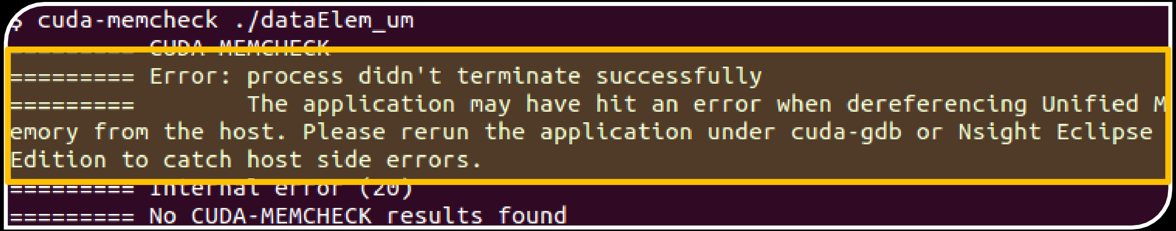


Developer Tools Support Unified Memory

- Visual Profiler, nvprof:

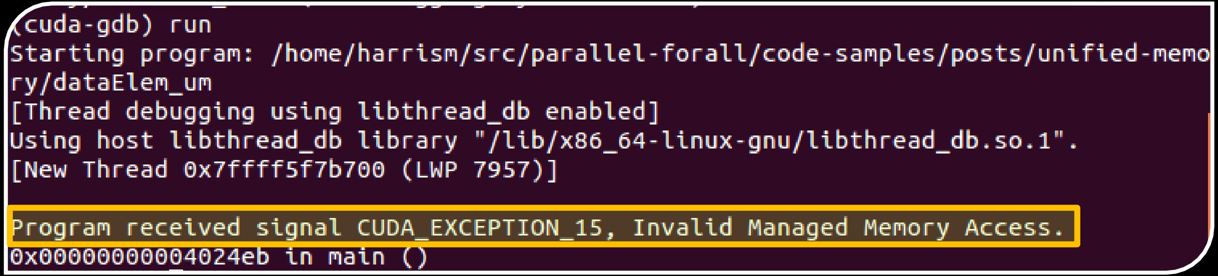


- cuda-memcheck:



```
$ cuda-memcheck ./dataElem_um
========= CUDA-MEMCHECK
========= Error: process didn't terminate successfully
=========        The application may have hit an error when dereferencing Unified M
emory from the host. Please rerun the application under cuda-gdb or Nsight Eclipse
Edition to catch host side errors.
========= Internal error (20)
========= No CUDA-MEMCHECK results found
```

- cuda-gdb:



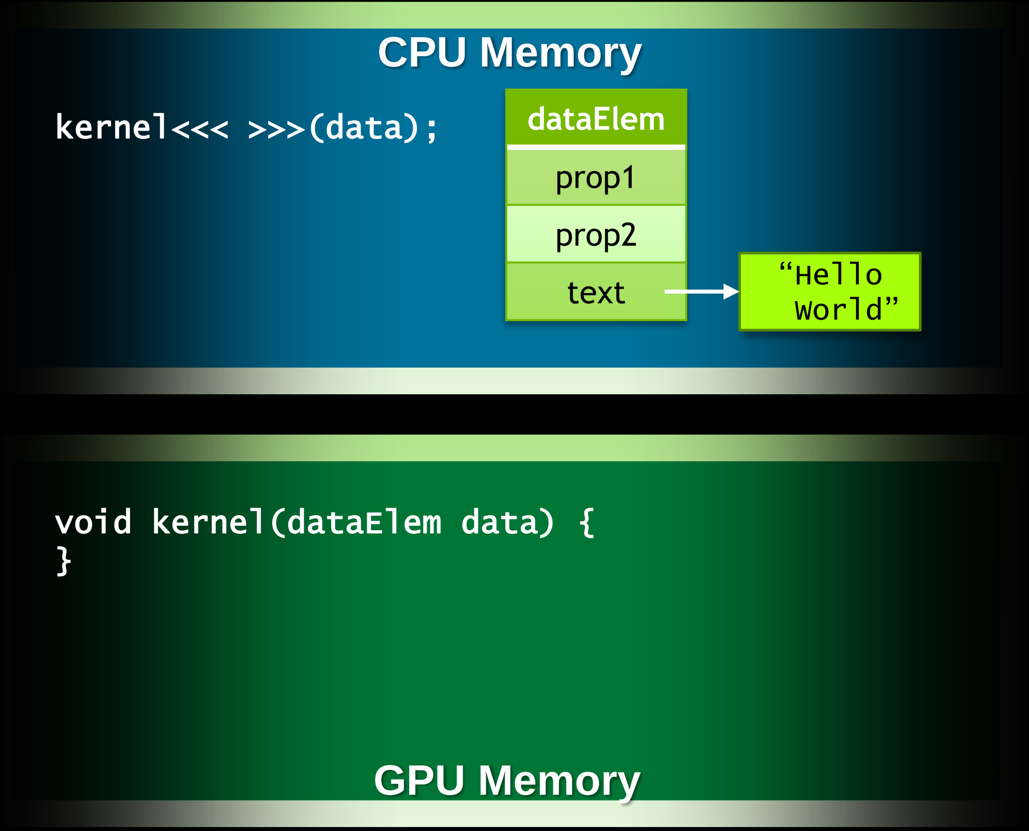
```
(cuda-gdb) run
Starting program: /home/harrism/src/parallel-forall/code-samples/posts/unified-memo
ry/dataElem_um
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff5f7b700 (LWP 7957)]

Program received signal CUDA_EXCEPTION_15, Invalid Managed Memory Access.
0x00000000004024eb in main ()
```

Unified Memory with C++

Host/Device C++ integration has been difficult in CUDA

- Cannot construct GPU class from CPU
- References fail because of no deep copies

```
// Ideal C++ version of class
class dataElem {
    int prop1;
    int prop2;
    String text;
};
```



```
kernel<<< >>>(data);
```

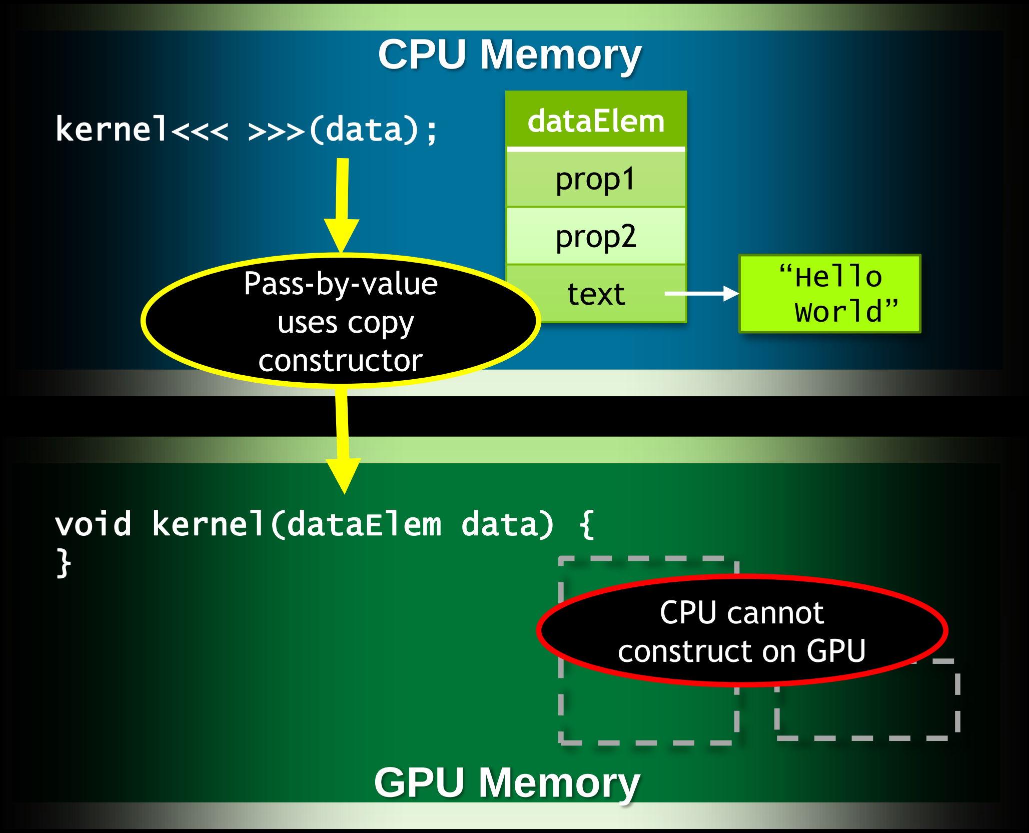
```
void kernel(dataElem data) {
}
```

Unified Memory with C++

Host/Device C++ integration has been difficult in CUDA

- Cannot construct GPU class from CPU
- References fail because of no deep copies

```
// Ideal C++ version of class
class dataElem {
    int prop1;
    int prop2;
    String text;
};
```


CPU Memory
kernel<<< >>>(data);
dataElem
prop1
prop2
text
"Hello World"
Pass-by-value uses copy constructor
void kernel(dataElem data) {
}
CPU cannot construct on GPU
GPU Memory

Unified Memory with C++

C++ objects migrate easily when allocated on managed heap

- Overload *new* operator* to use C++ in unified memory region

```
class Managed {
    void *operator new(size_t len) {
        void *ptr;
        cudaMallocManaged(&ptr, len);
        return ptr;
    }

    void operator delete(void *ptr) {
        cudaFree(ptr);
    }
};
```

* (or use placement-new)

Unified Memory with C++

Pass-by-reference enabled with *new* overload

```
// Deriving from “Managed” allows pass-by-reference
class String : public Managed {
    int length;
    char *data;

    // Copy constructor using new allocates CPU-only data
    String (const String &s) {
        length = s.length;
        data = new char[length+1];
        strcpy(data, s.data);
    }
};
```

NOTE: CPU/GPU class sharing is restricted to POD-classes only (i.e. no virtual functions)

Unified Memory with C++

Pass-by-value enabled by managed memory copy constructors

```
// Deriving from “Managed” allows pass-by-reference
class String : public Managed {
    int length;
    char *data;

    // Unified memory copy constructor allows pass-by-value
    String (const String &s) {
        length = s.length;
        cudaMallocManaged(&data, length+1);
        strcpy(data, s.data);
    }
};
```

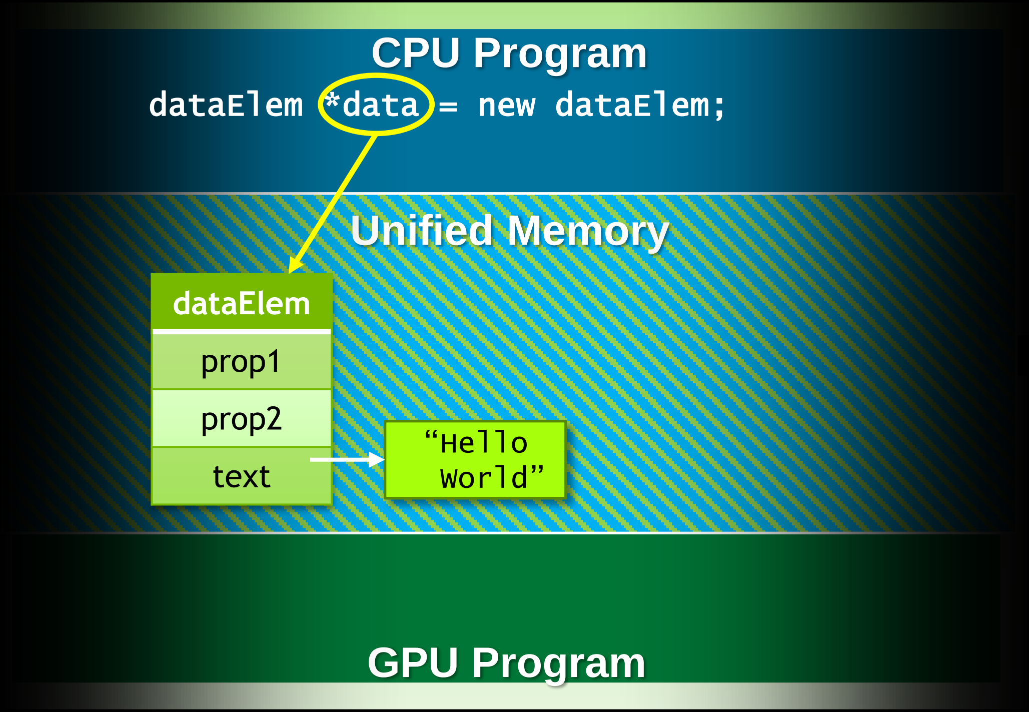
NOTE: CPU/GPU class sharing is restricted to POD-classes only (i.e. no virtual functions)

Unified Memory with C++

Combination of C++ and Unified Memory is very powerful

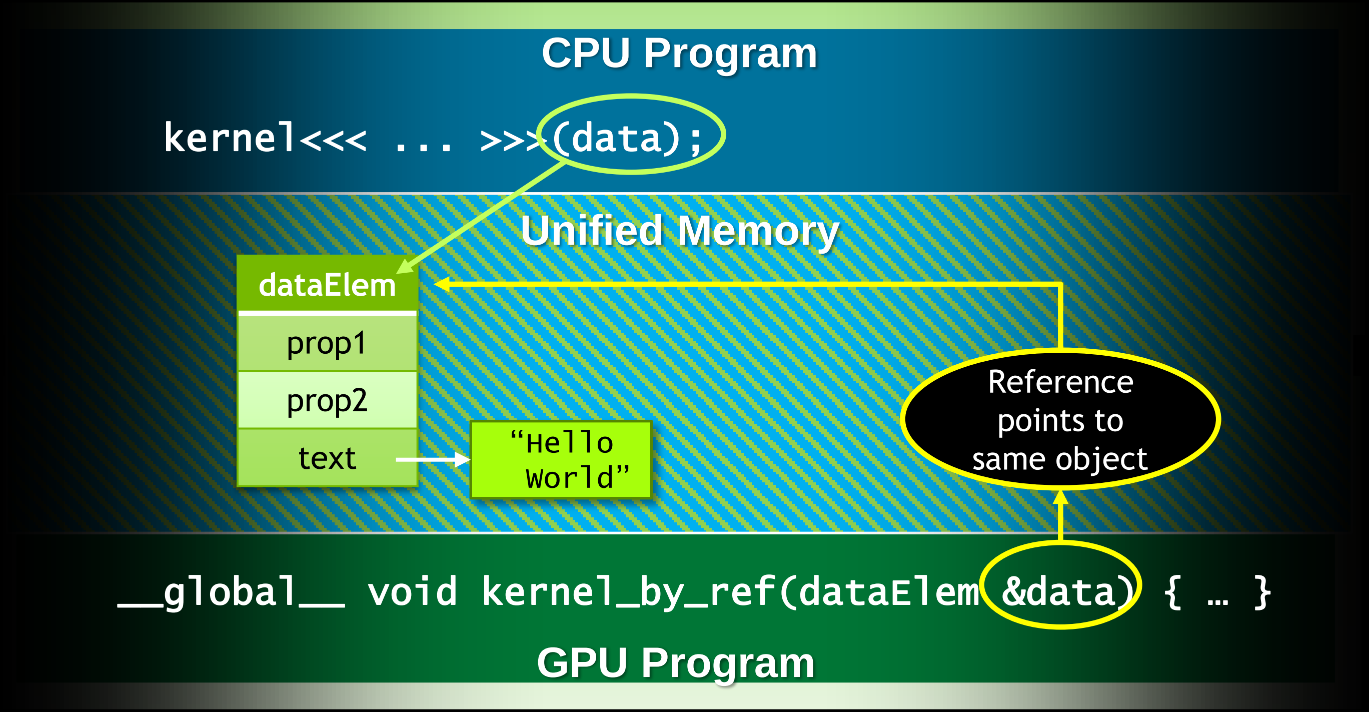
- Concise and explicit: let C++ handle deep copies
- Pass by-value or by-reference without memcpy shenanigans

```
// Note "Managed" on this class, too.
// C++ now handles our deep copies
class dataElem : public Managed {
    int prop1;
    int prop2;
    String text;
};
```



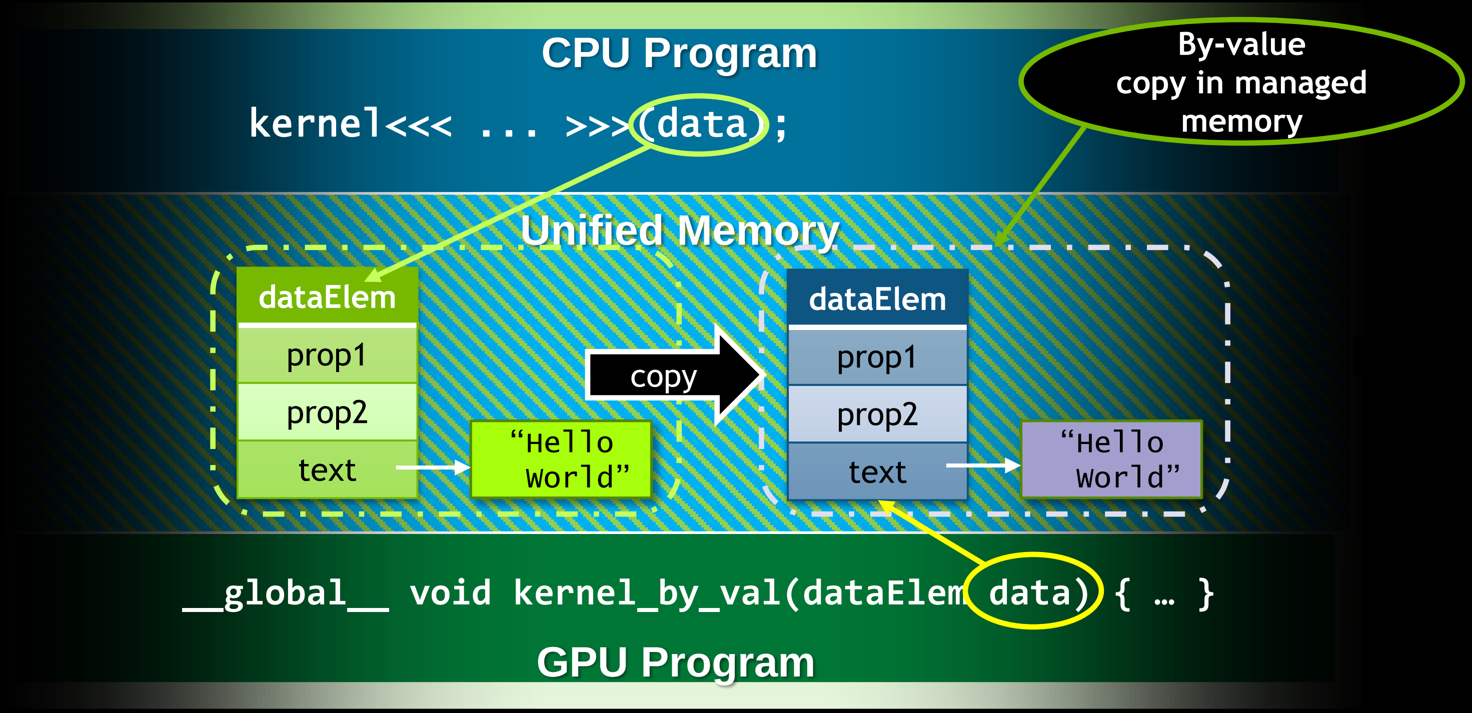
C++ Pass By Reference

Single pointer to data makes object references just work



C++ Pass By Value

Copy constructors from CPU create GPU-usable objects



Unified Memory Roadmap

CUDA 6: Ease of Use

Single Pointer to Data

No Memcopy Required

Coherence @ launch & sync

Shared C/C++ Data Structures

Next: Optimizations

Prefetching

Migration Hints

Additional OS Support

Future GPUs

Finer Grain Migration

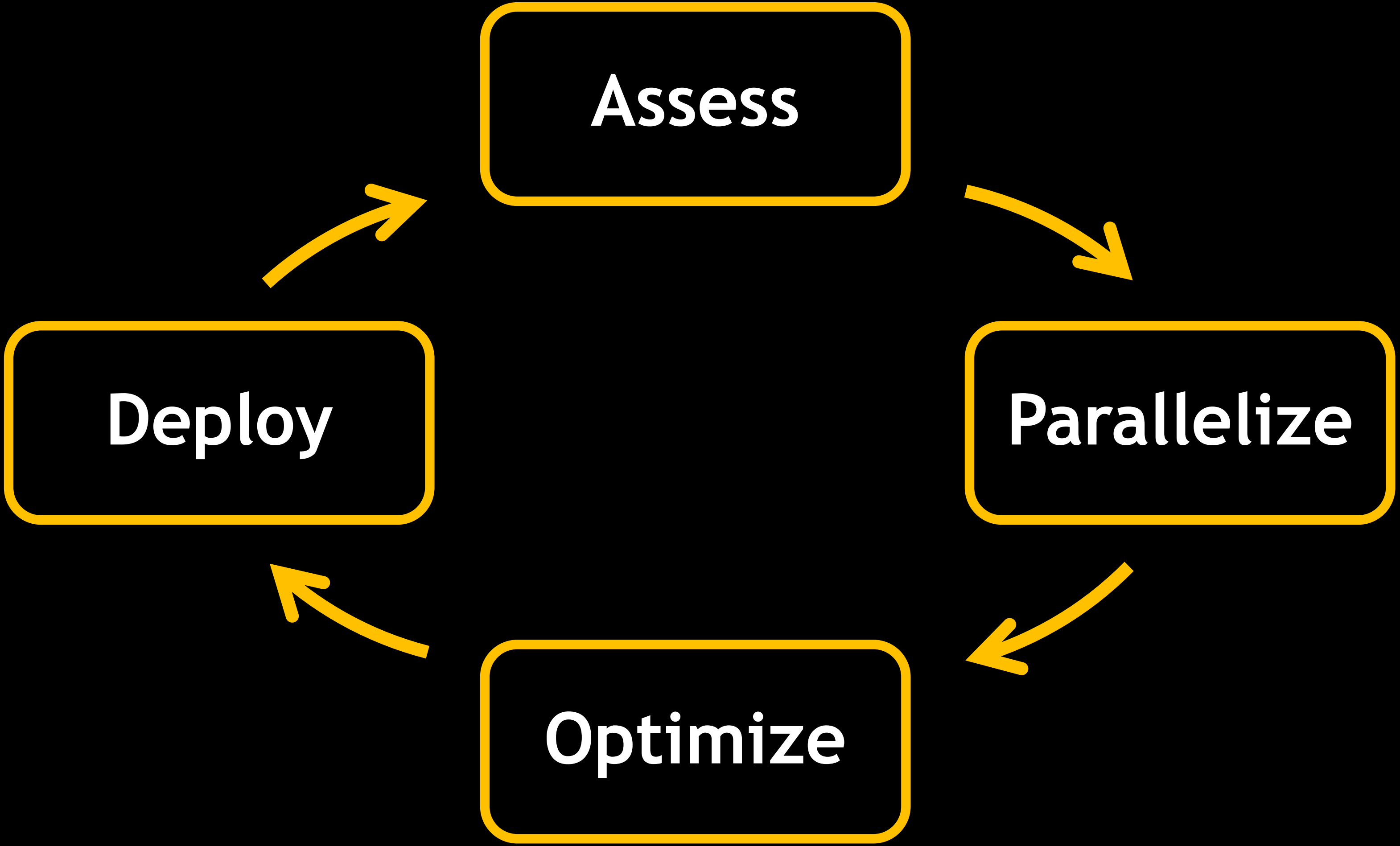
Not Limited to GPU Memory Size

PARALLEL FORALL Learn More: http://bit.ly/um-p4a

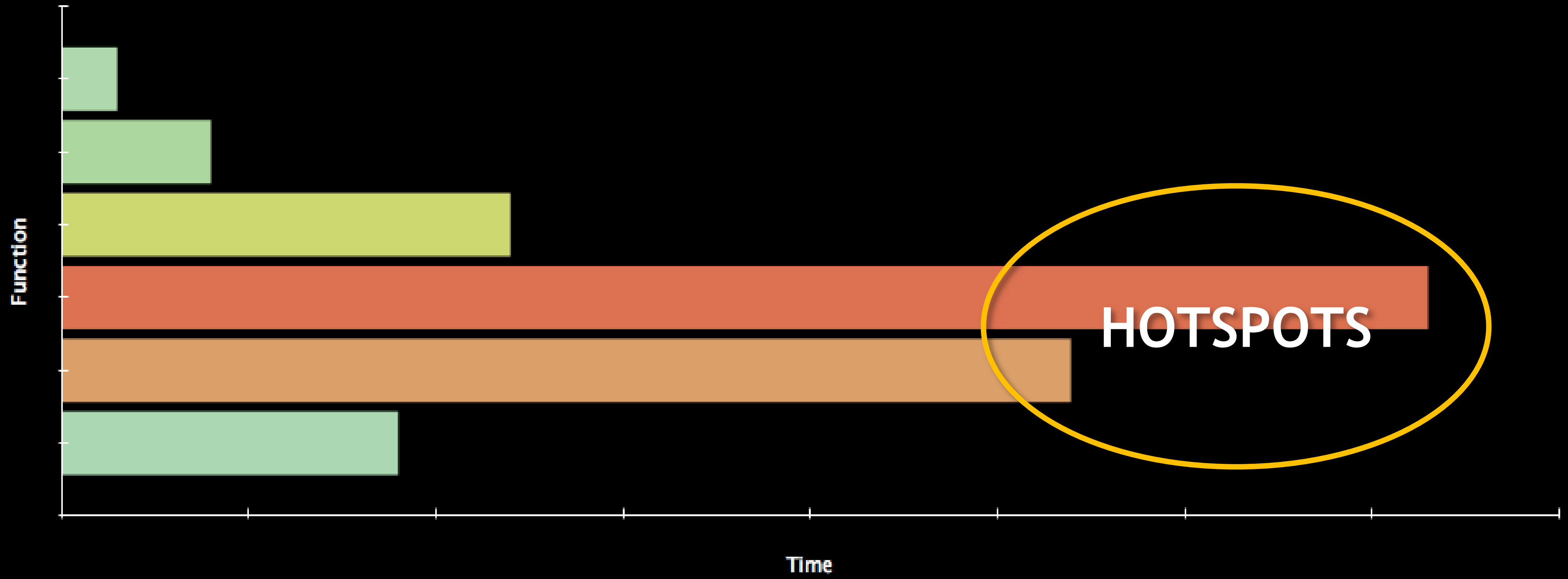
GPU Optimization

GPU OPTIMIZATION FUNDAMENTALS

APOD: A Systematic Path to Performance



Assess



- Identify hotspots (total time, number of calls)
- Understand scaling (strong and weak)

Assess: Understanding Scaling

Strong Scaling

- A measure of how, for fixed overall problem size, the time to solution decreases as more processors are added to a system
- Linear strong scaling: speedup achieved is equal to number of processors used

- Amdahl's Law:

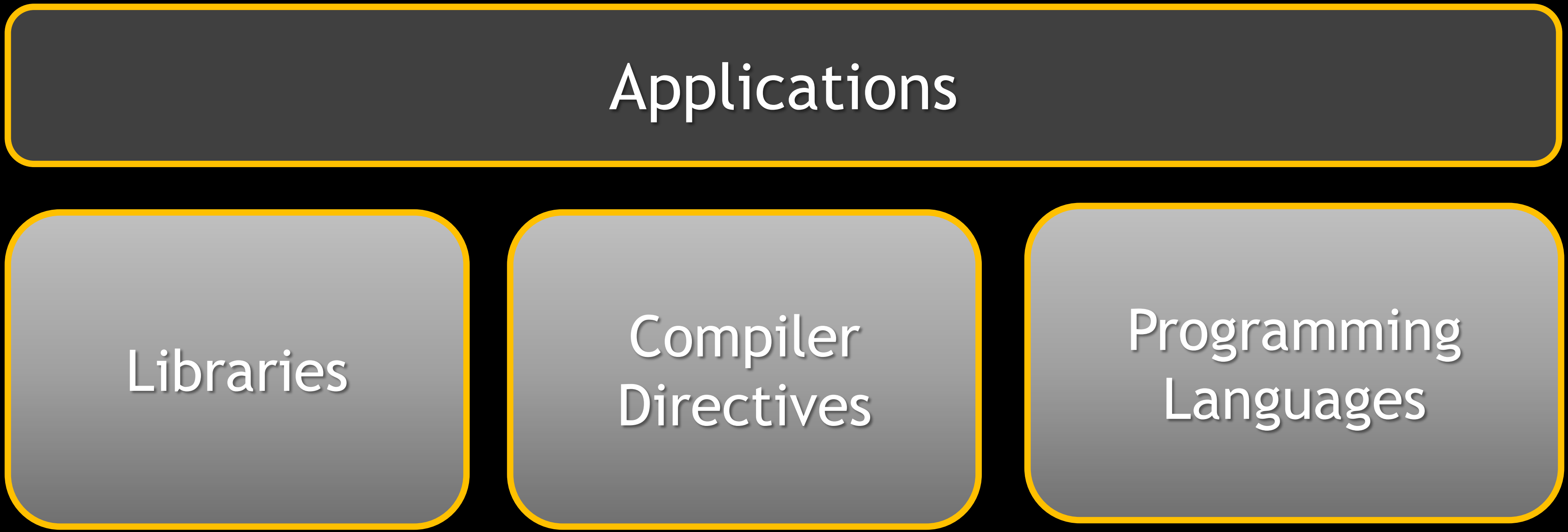
$$S = \frac{1}{(1-P) + \frac{P}{N}} \approx \frac{1}{(1-P)}$$

Assess: Speed of Light

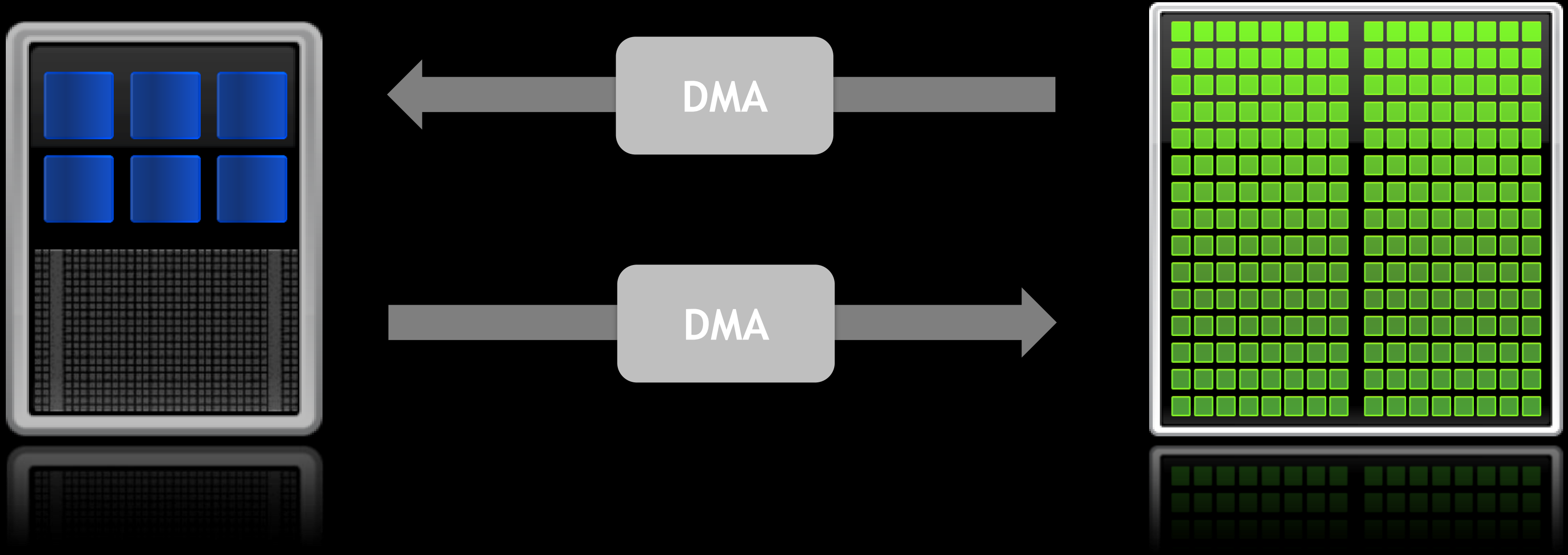
- What’s the limiting factor?
  - Memory bandwidth?
  - Compute throughput?
  - Latency?

  - Not sure?
    - Get a rough estimate by counting bytes per instruction, compare it to “balanced” peak ratio $\frac{GBytes/sec}{Ginsns/sec}$
    - Profiler will help you determine this

Parallelize

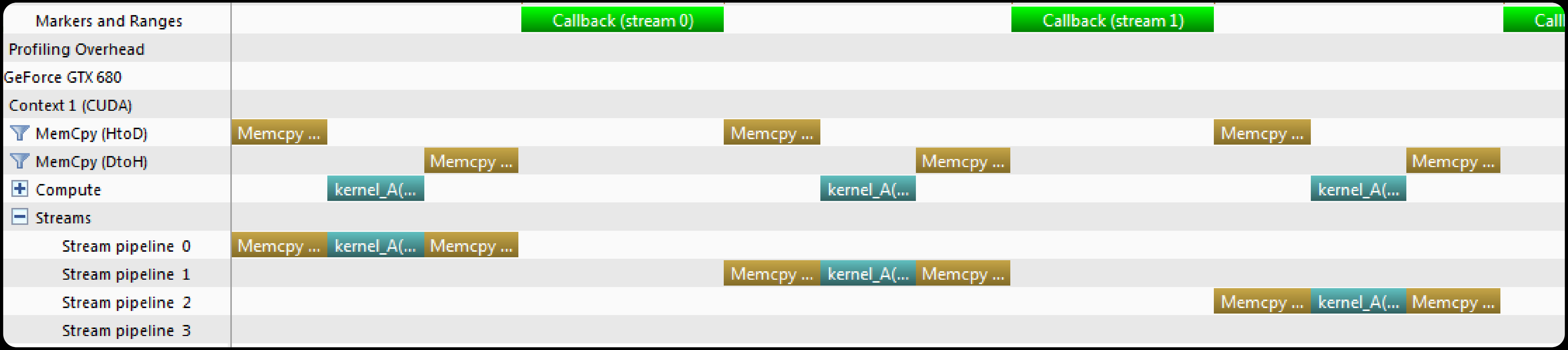


Asynchronicity = Overlap = Parallelism



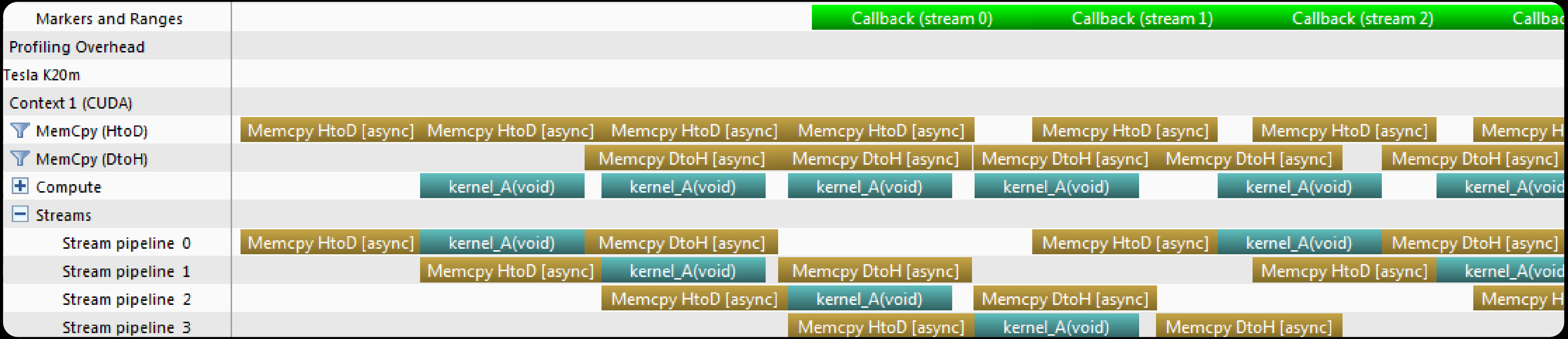
Heterogeneous system: overlap work and data movement

Asynchronicity



- This is the kind of case we would be concerned about
  - Found the top kernel, but the GPU is mostly idle – *that* is our bottleneck
  - Need to overlap CPU/GPU computation and PCIe transfers

Parallelize: Achieve Asynchronicity



What we *want* to see is maximum overlap of all engines

Optimize

- Profile-driven optimization

- Tools:
    - nsight Visual Studio Edition or Eclipse Edition
    - nvvp NVIDIA Visual Profiler
    - nvprof Command-line profiling

Deploy

Productize

- Check API return values
- Run cuda-memcheck tools

- Library distribution
- Cluster management

Early gains
Subsequent changes are evolutionary

OPTIMIZE

Main Requirements for GPU Performance

- Expose sufficient parallelism
- Utilize parallel execution resources efficiently
  - Use memory system efficiently
    - Coalesce global memory accesses
    - Use shared memory where possible
  - Have coherent execution within *warps* of threads

GPU Optimization Fundamentals

- Find ways to parallelize sequential code
- Adjust kernel launch configuration to maximize device utilization
- Ensure global memory accesses are coalesced
- Minimize redundant accesses to global memory
- Avoid different execution paths within the same warp
- Minimize data transfers between the host and the device

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

GPU Optimization Fundamentals

- Find ways to parallelize sequential code
- Kernel optimizations
  - Launch configuration
  - Global memory throughput
  - Shared memory access
  - Instruction throughput / control flow
- Optimization of CPU-GPU interaction
  - Maximizing PCIe throughput
  - Overlapping kernel execution with memory copies

OPTIMIZE

Kernel Optimizations: *Kernel Launch Configuration*

Kernel Launch Configuration

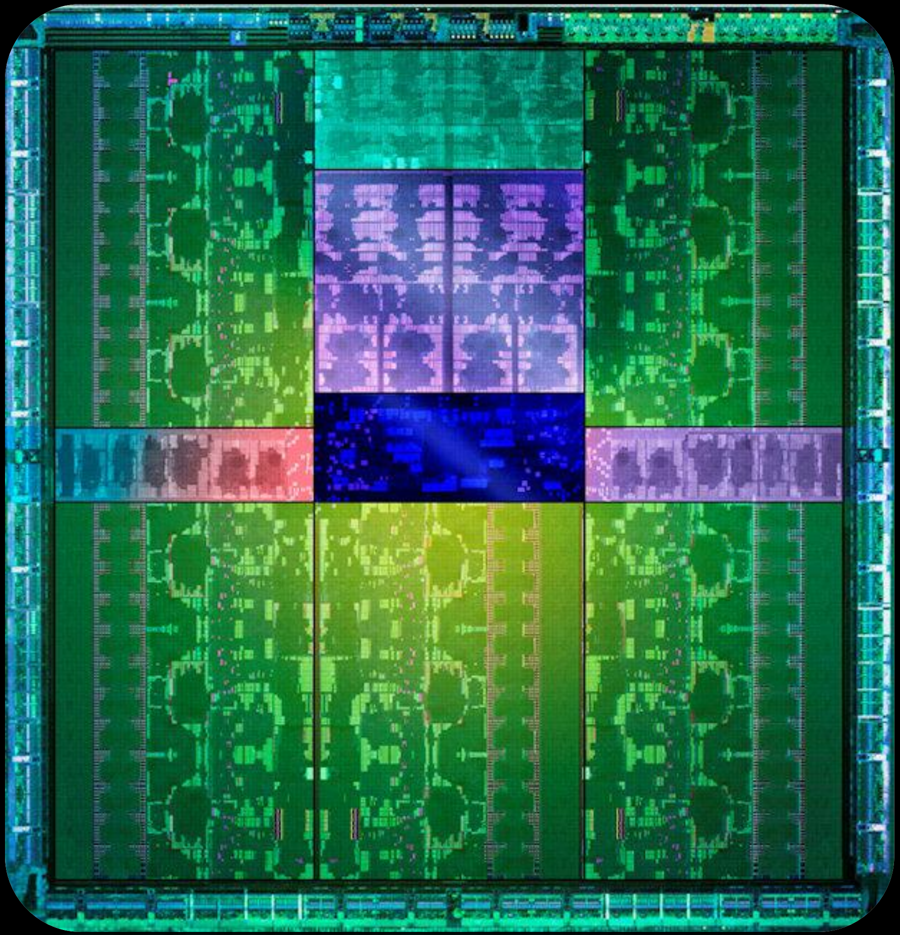
- A kernel is a function that runs on the GPU
- A kernel is launched as a grid of blocks of threads
- Launch configuration is the number of blocks and number of threads per block, expressed in CUDA with the <<< >>> notation:

```
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

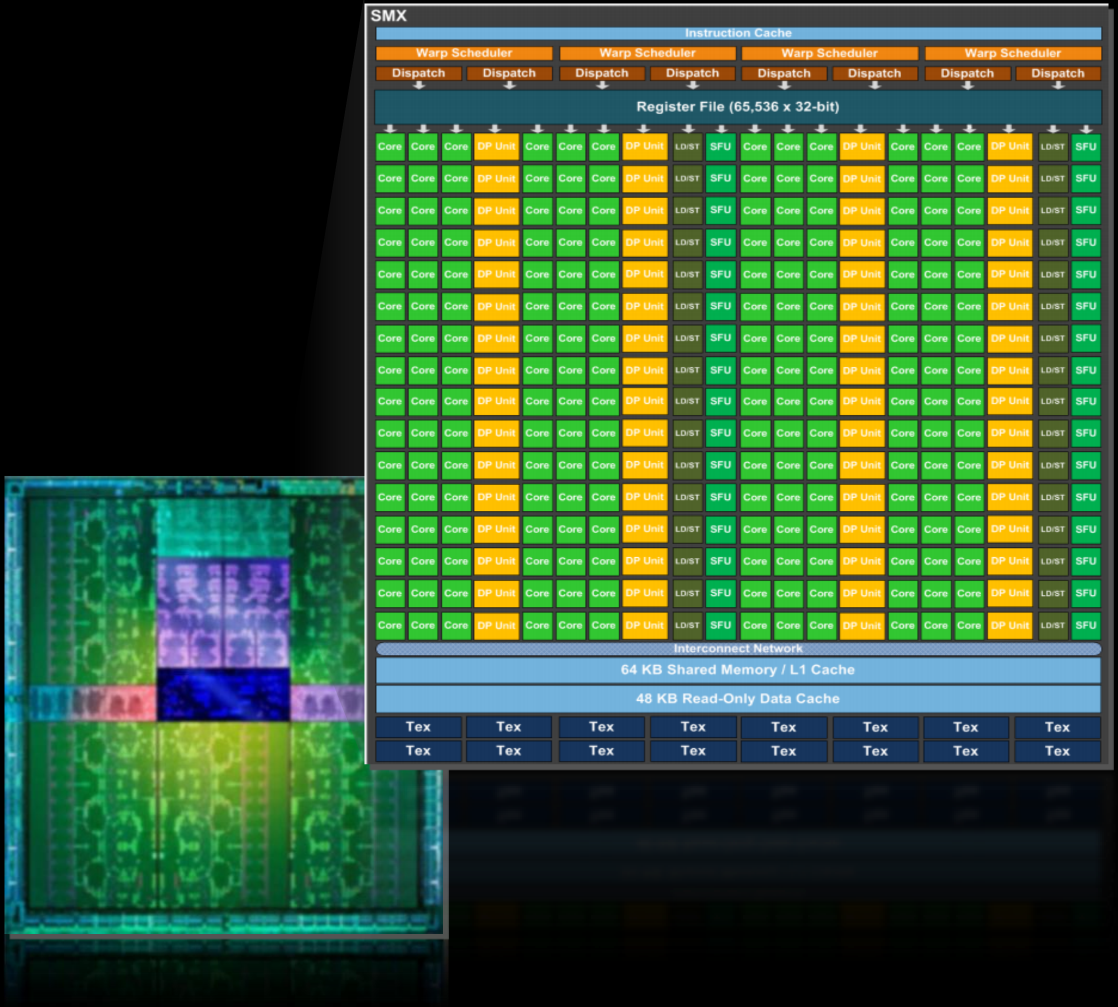
- Several Streaming Multiprocessors
  - E.g., Kepler GK110 has up to 15 SMs
- L2 Cache shared among SMs
- Multiple channels to DRAM


Kepler GK110

Kepler Streaming Multiprocessor (SMX)

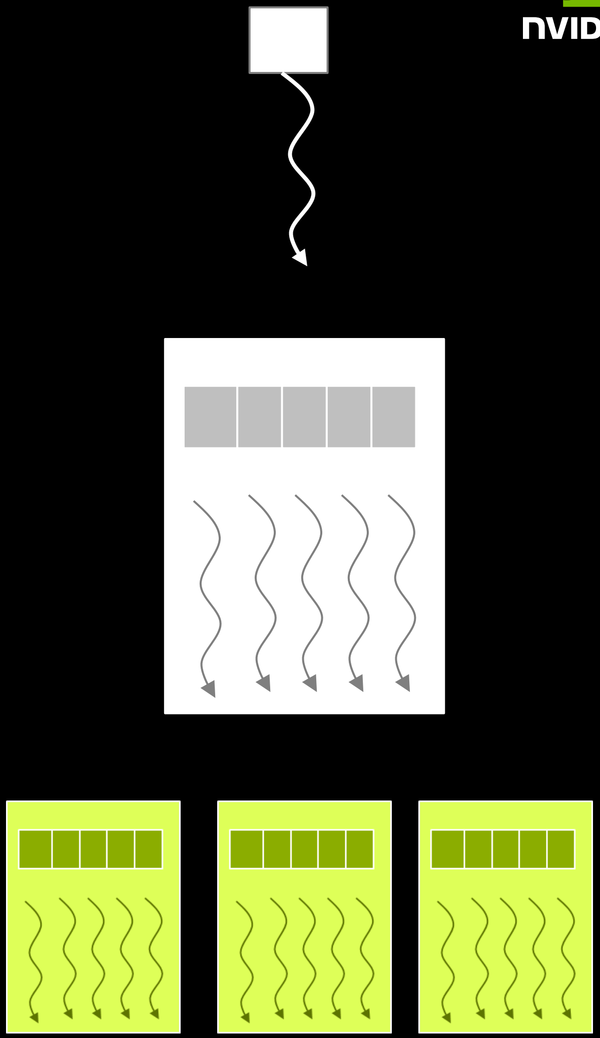
Per SMX:

- 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)

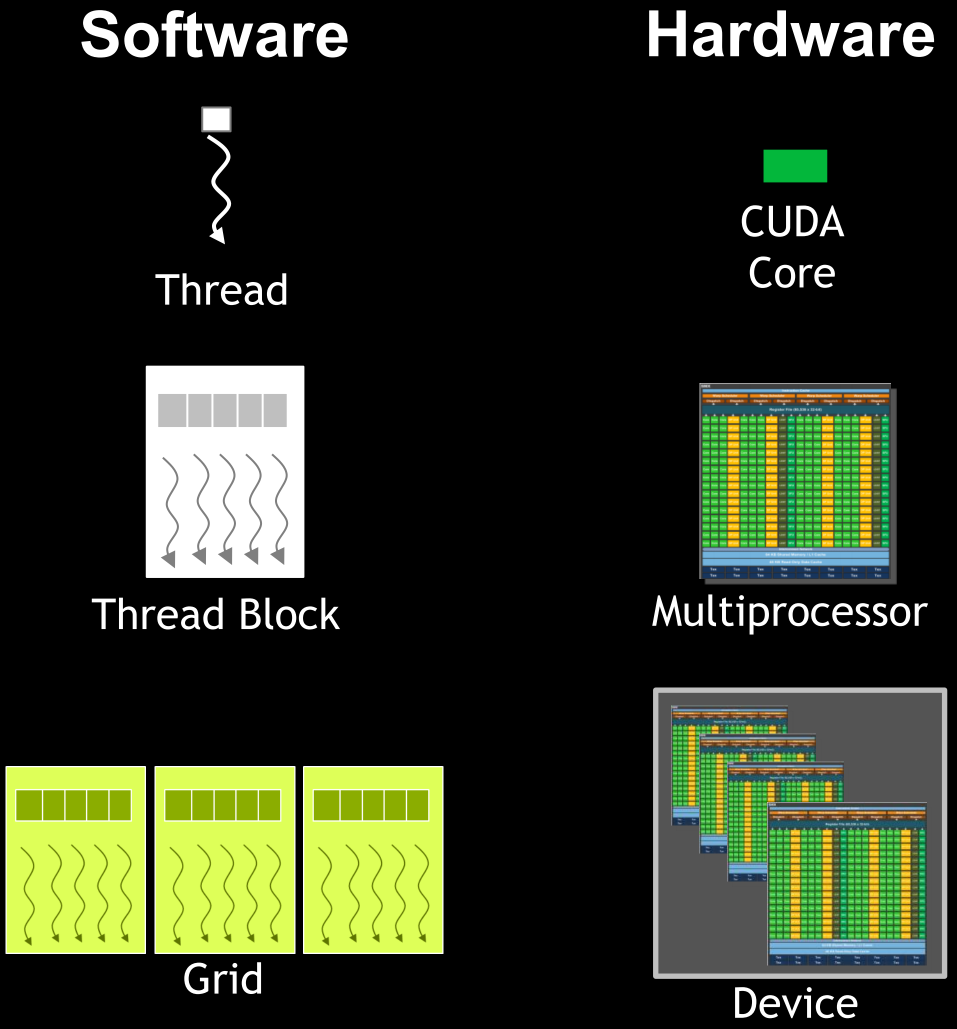

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

CUDA Execution Model

- Thread: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel
- Threads 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



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

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

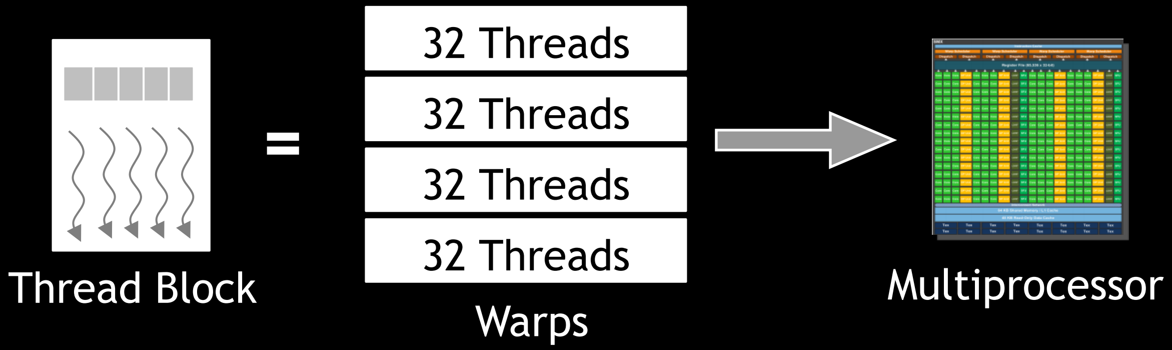
Launch Configuration: General Guidelines

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

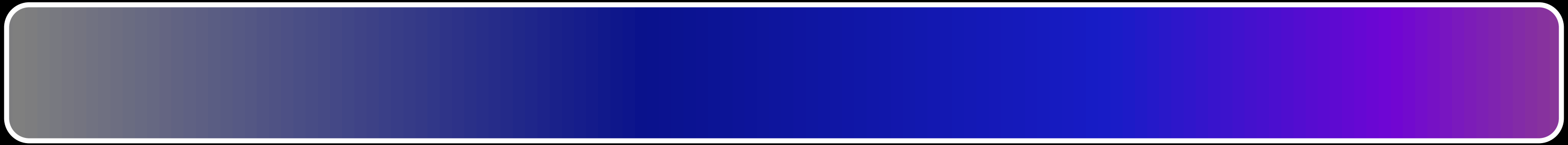


A thread block consists of *warps of* 32 threads

A warp is executed physically in parallel on some multiprocessor.

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

Hardware Levels of Parallelism



Single Instruction, Multiple Data
In-core parallelism

SIMD

Simultaneous Multithreading
Cross-core, Cross-socket
Single Computer
OpenMP, pthreads

SMT

Multiple “computers”
Tightly-coupled
Supercomputing apps

MPI

SIMT

Single Instruction, Multiple Threads
In-processor parallelism
Many threads on many cores

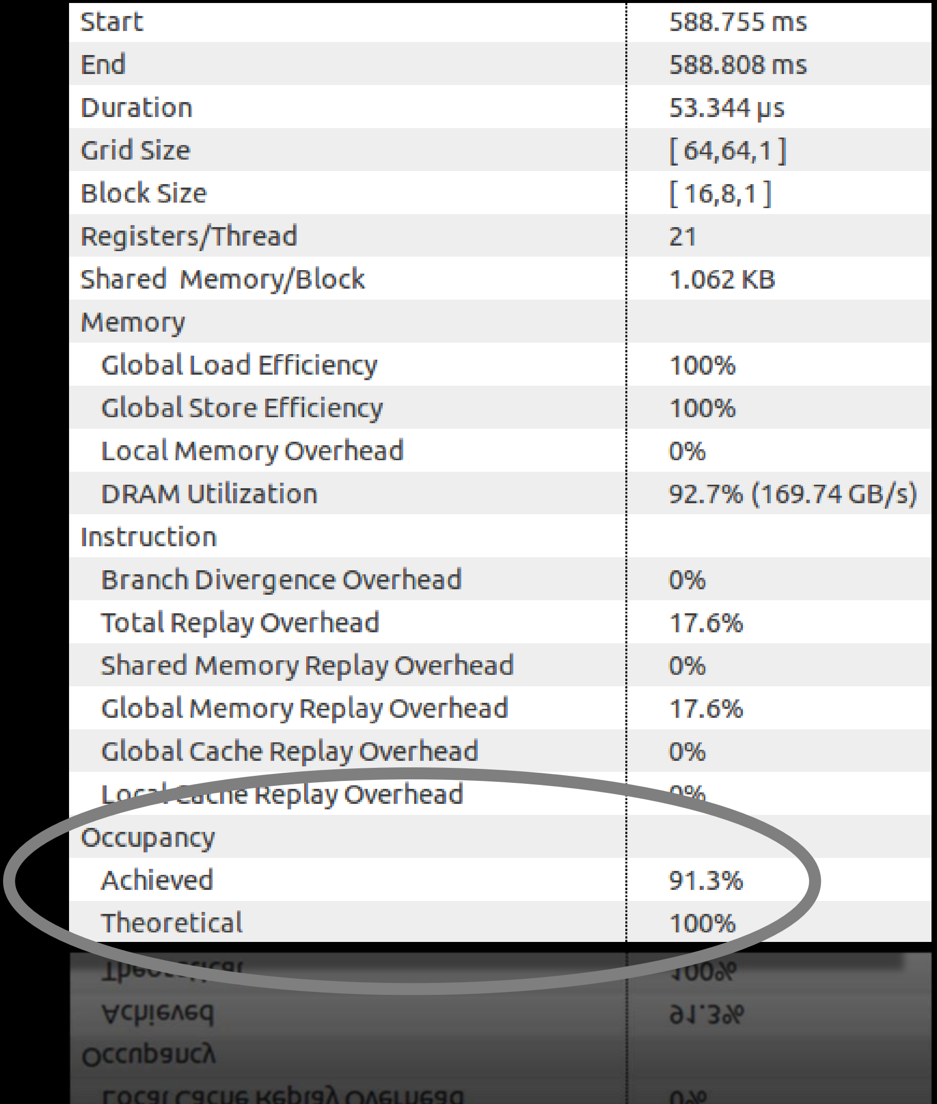
These form a continuum. Best performance is achieved with a mix.

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

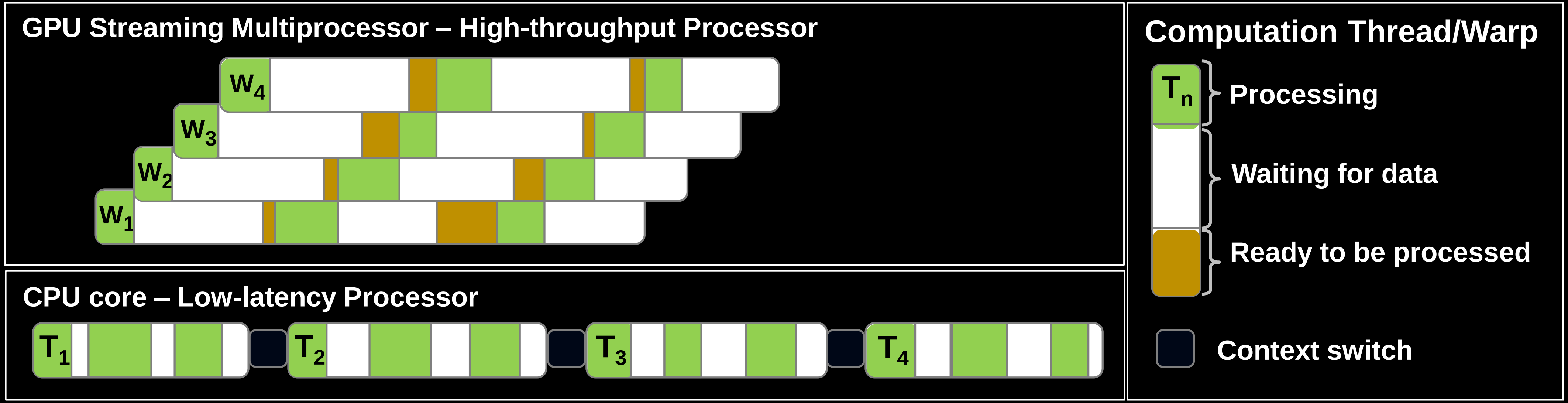
$$\text{Occupancy} = N_{\text{actual}} \ / \ N_{\text{max}}$$



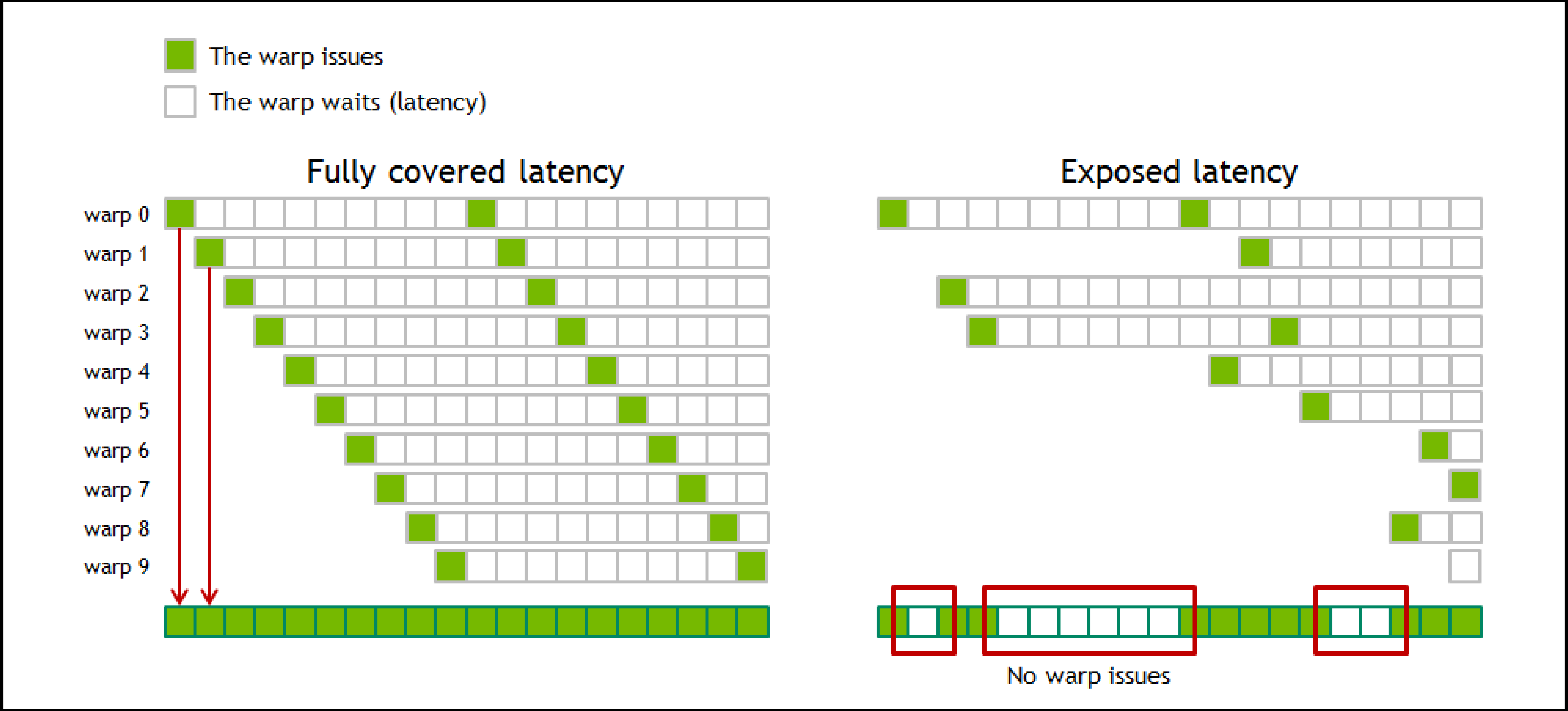
Start	588.755 ms
End	588.808 ms
Duration	53.344 μs
Grid Size	[64,64,1]
Block Size	[16,8,1]
Registers/Thread	21
Shared Memory/Block	1.062 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	92.7% (169.74 GB/s)
Instruction	
Branch Divergence Overhead	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%

Low Latency or High Throughput?

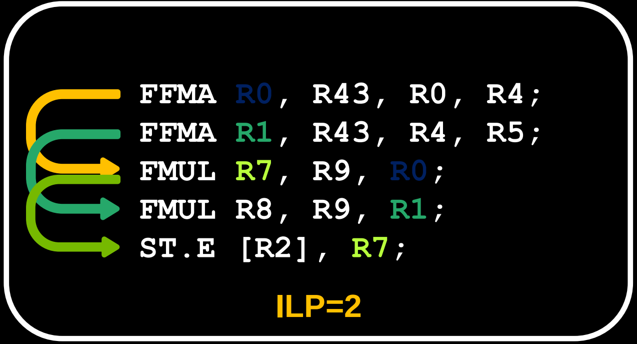
- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other (warps of) threads



Latency hiding



Latency Hiding



```
FFMA R0, R43, R0, R4;
FFMA R1, R43, R4, R5;
FMUL R7, R9, R0;
FMUL R8, R9, R1;
ST.E [R2], R7;
```

ILP=2

- Instruction latencies:
  - Roughly 10-20 cycles for arithmetic operations
  - DRAM accesses have higher latencies (400-800 cycles)
- Instruction Level Parallelism (ILP)
  - Independent instructions between two dependent ones
  - ILP depends on the code, done by the compiler
- Switching to a different warp
  - If a warp must stall for *N* cycles due to dependencies, having *N* other warps with eligible instructions keeps the SM going
  - Switching among concurrently resident warps has no overhead
    - State (registers, shared memory) is partitioned, not stored/restored

Occupancy

- Occupancy: number of concurrent warps per SM, expressed as:
  - Absolute number of warps of threads that fit concurrently (e.g., 1..64), or
  - Ratio of warps that fit concurrently to architectural maximum (0..100%)

- Number of warps that fit determined by resource availability:
  - Threads per thread block
  - Registers per thread
  - Shared memory per thread block

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

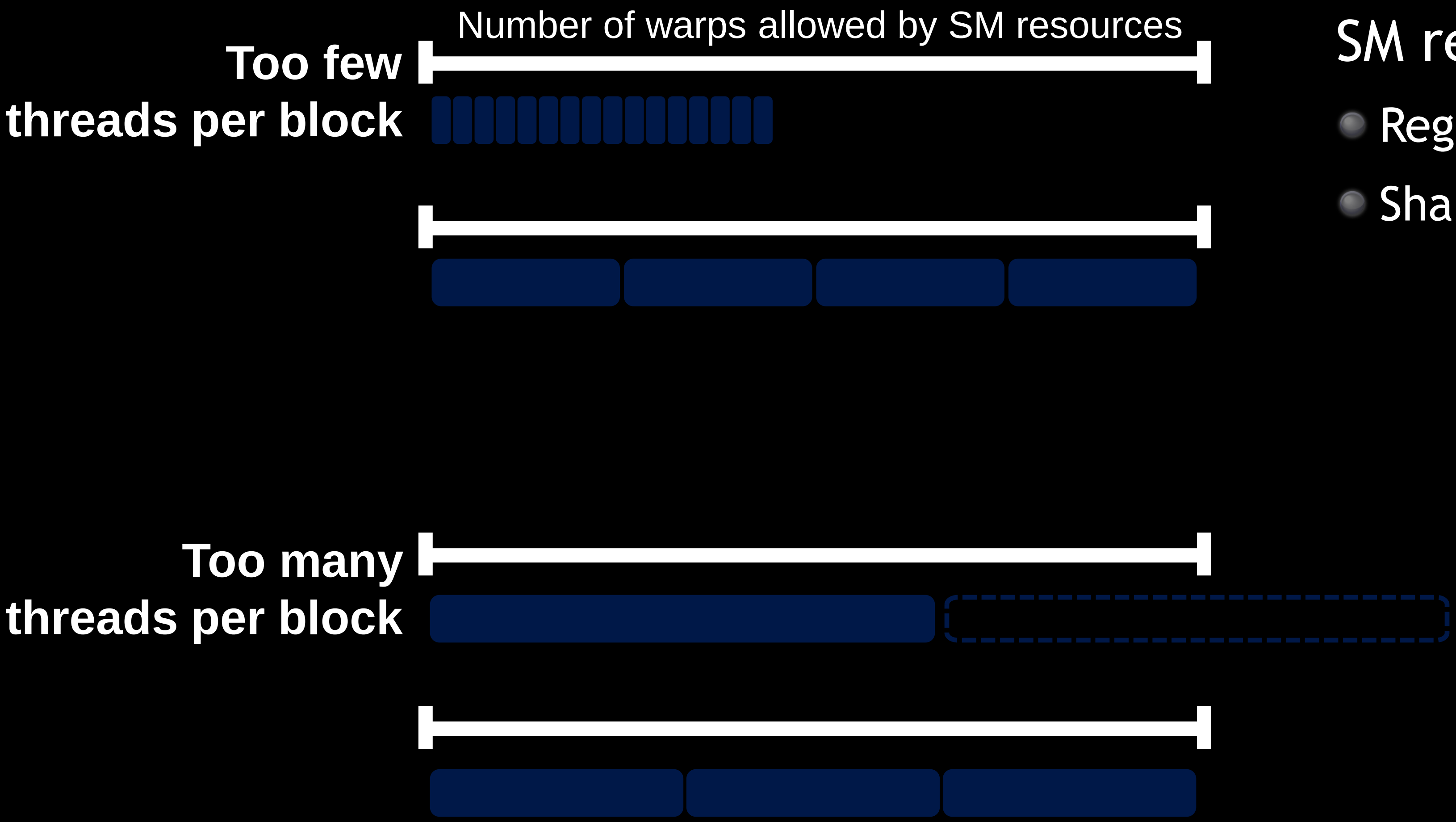
Occupancy and Performance

- 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

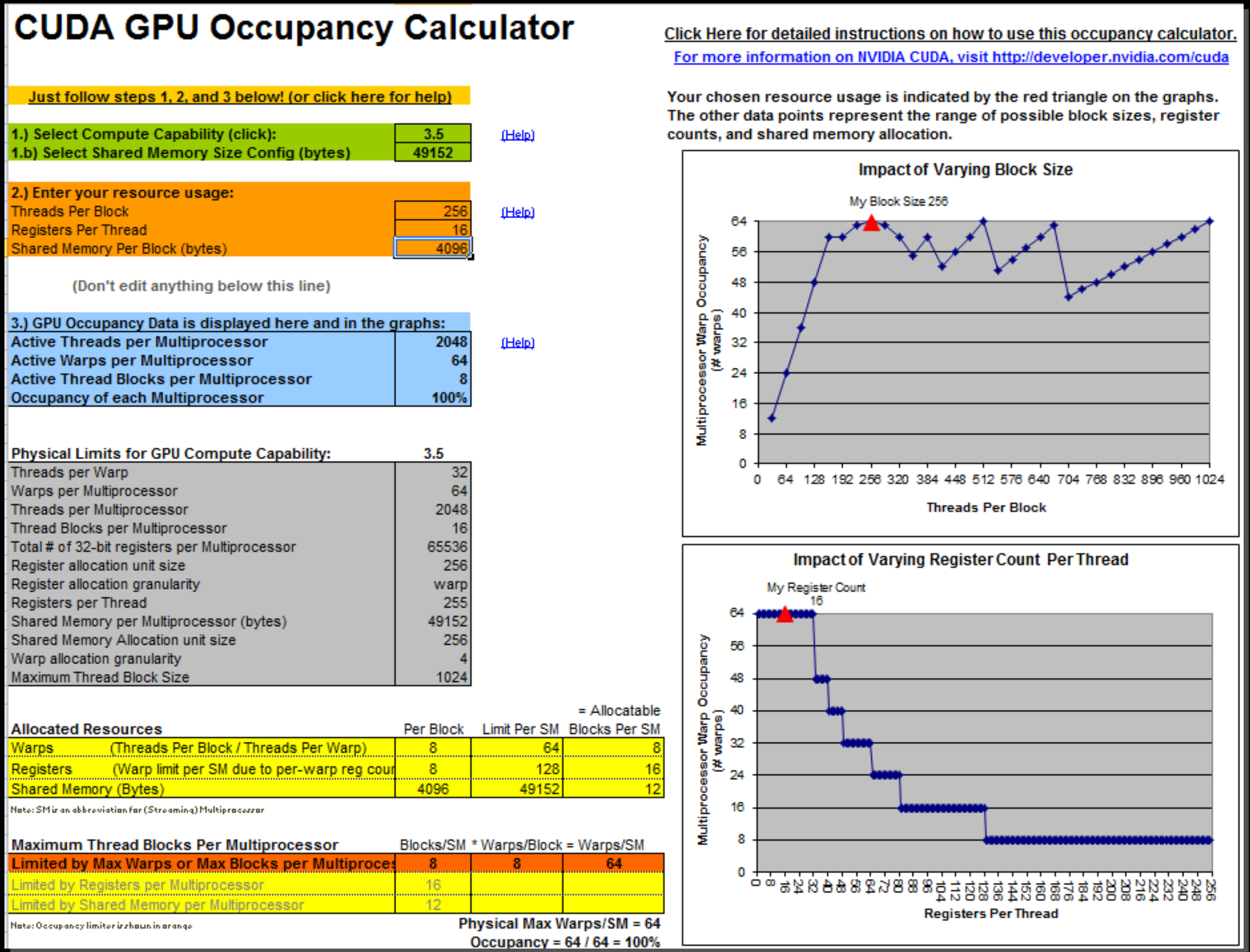

Number of warps allowed by SM resources
Too few threads per block
Too many threads per block

SM resources:

- Registers
- Shared memory

CUDA Occupancy Calculator

- Analyze effect of resource consumption on occupancy



CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
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 Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

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	2048	(Help)
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 cour	8	128	16
Shared Memory (Bytes)		4096	49152	12

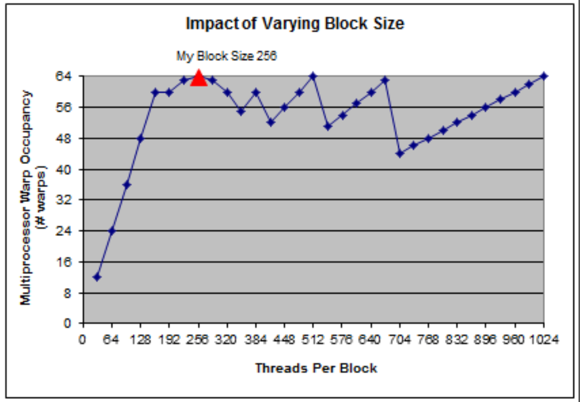
Note: SM is an abbreviation for (Streaming) Multiprocessor

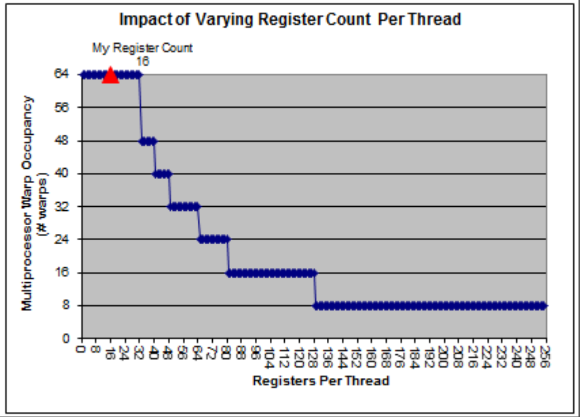
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiproces	8	8	64
Limited by Registers per Multiprocessor	16		
Limited by Shared Memory per Multiprocessor	12		

Note: Occupancy limiter is shown in orange

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

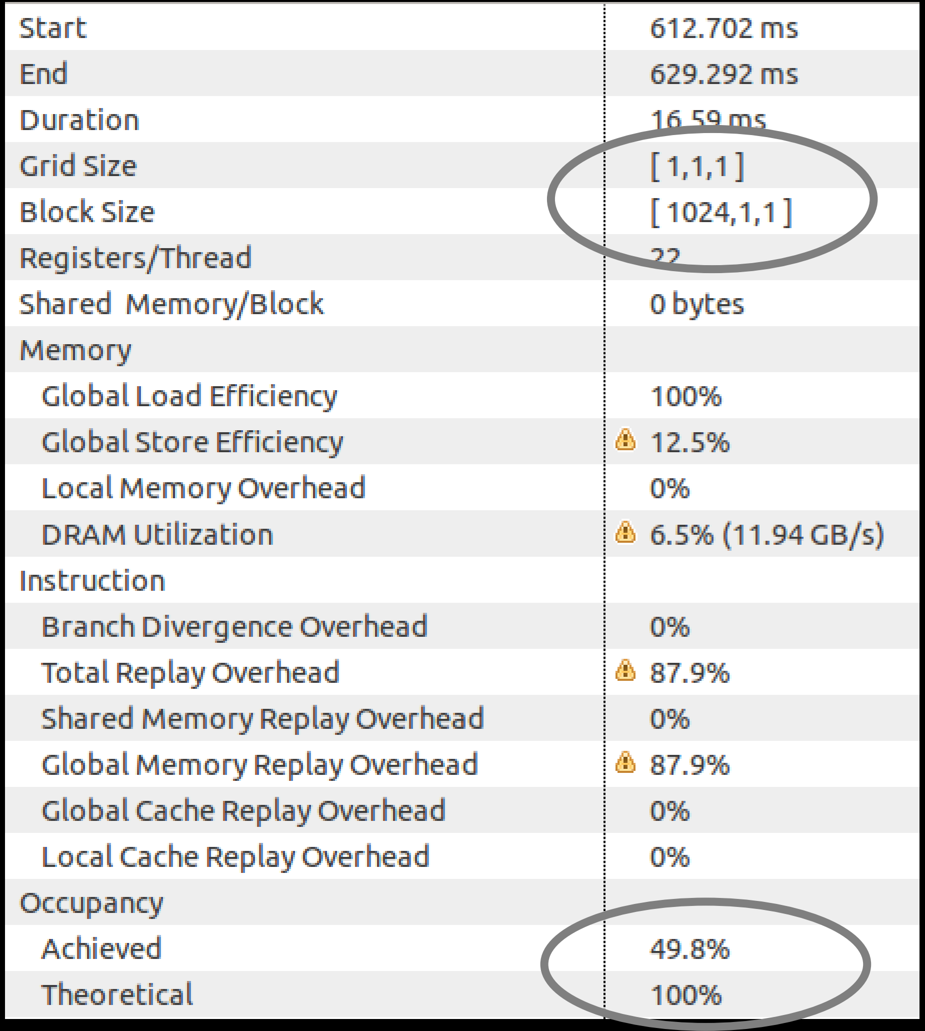
Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





Occupancy Analysis in NVIDIA Visual Profiler

- Occupancy here is limited by grid size and number of threads per block



Start	612.702 ms
End	629.292 ms
Duration	16.59 ms
Grid Size	[1,1,1]
Block Size	[1024,1,1]
Registers/Thread	22
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	100%
Global Store Efficiency	⚠ 12.5%
Local Memory Overhead	0%
DRAM Utilization	⚠ 6.5% (11.94 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 87.9%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	⚠ 87.9%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	49.8%
Theoretical	100%

Kepler: Level of Parallelism Needed

- To saturate instruction bandwidth:
  - Fp32 math: ~1.7K independent instructions per SM
  - Fewer for lower-throughput instructions
  - Keep in mind that Kepler can track up to 2048 threads per SM
- To saturate memory bandwidth:
  - 100+ concurrent independent 128-byte lines per SM