Cliff Woolley
Developer Technology Engineer

Note: Fundamentals will apply broadly

- Example performance numbers are presented for Tesla K20X, which is based on the Kepler GK110 GPU
- Same general optimization concepts apply to other GPUs, though some parameters may be different, e.g.:
  - Number of SMs per GPU
  - Number of functional units per SM
  - Maximum number of concurrent warps per SM
  - Shared memory size per SM
  - Register file size per SM
- Developer tools from NVIDIA help you analyze the concepts without having to memorize parameters of each architecture

© NVIDIA 2013

GPU OPTIMIZATION FUNDAMENTALS

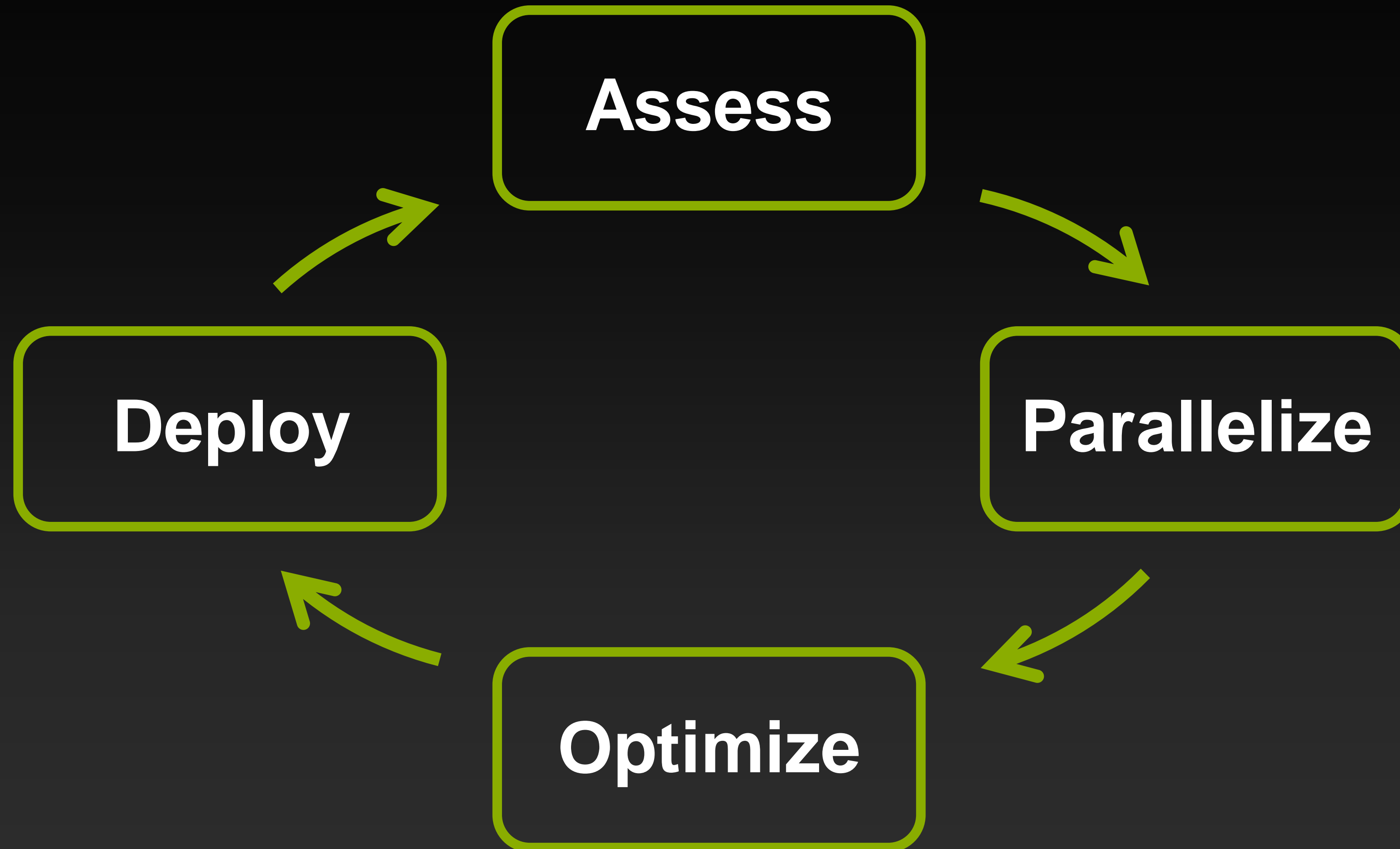
© NVIDIA 2013

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

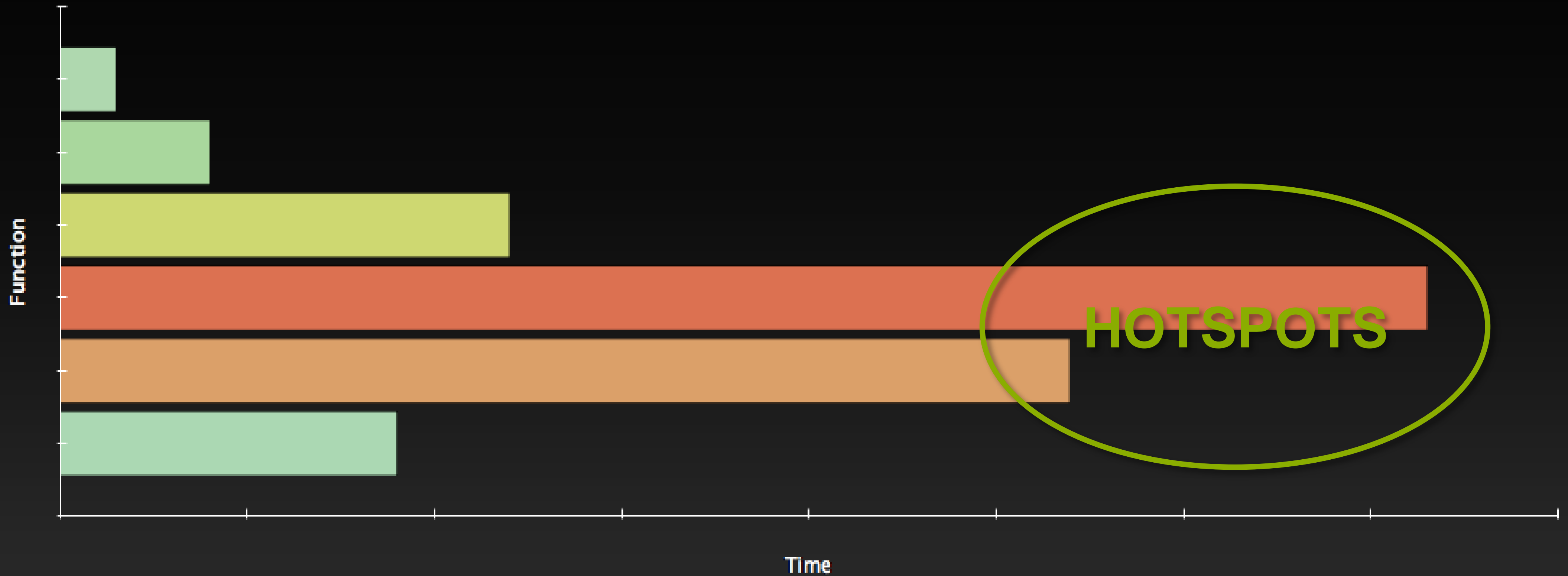
© NVIDIA 2013

APOD: A Systematic Path to Performance



© NVIDIA 2013

Assess



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

© NVIDIA 2013

Parallelize

Applications

Libraries

Compiler Directives

Programming Languages

© NVIDIA 2013

Optimize

- Profile-driven optimization

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

© NVIDIA 2013

Deploy

Productize

- Check API return values
- Run cuda-memcheck tools

- Library distribution
- Cluster management

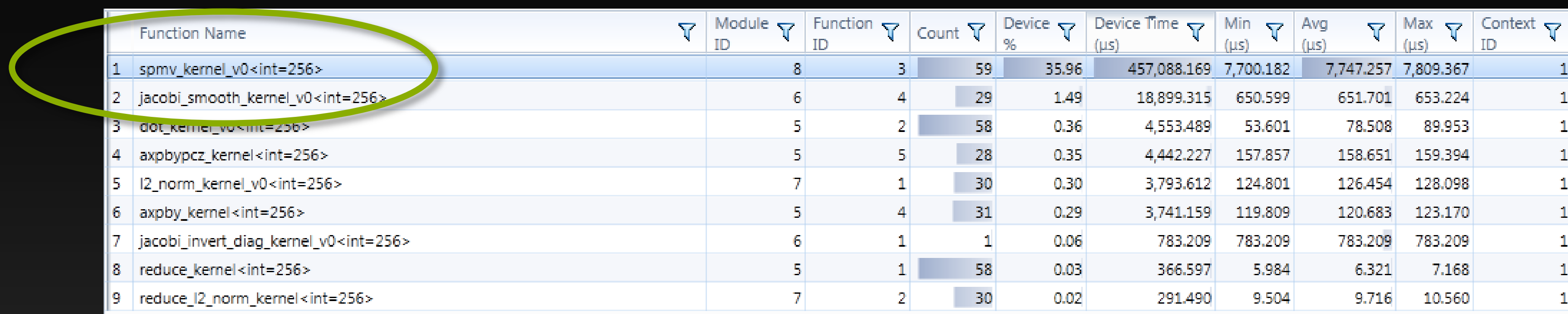
Early gains
Subsequent changes are evolutionary

© NVIDIA 2013

ASSESS

© NVIDIA 2013

Assess



	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID
1	spmv_kernel_v0<int=256>	8	3	59	35.96	457,088.169	7,700.182	7,747.257	7,809.367	1
2	jacobi_smooth_kernel_v0<int=256>	6	4	29	1.49	18,899.315	650.599	651.701	653.224	1
3	dot_kernel_v0<int=256>	5	2	58	0.36	4,553.489	53.601	78.508	89.953	1
4	axpbypcz_kernel<int=256>	5	5	28	0.35	4,442.227	157.857	158.651	159.394	1
5	l2_norm_kernel_v0<int=256>	7	1	30	0.30	3,793.612	124.801	126.454	128.098	1
6	axpby_kernel<int=256>	5	4	31	0.29	3,741.159	119.809	120.683	123.170	1
7	jacobi_invert_diag_kernel_v0<int=256>	6	1	1	0.06	783.209	783.209	783.209	783.209	1
8	reduce_kernel<int=256>	5	1	58	0.03	366.597	5.984	6.321	7.168	1
9	reduce_l2_norm_kernel<int=256>	7	2	30	0.02	291.490	9.504	9.716	10.560	1

- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit

© NVIDIA 2013

Assess

- We've found a hotspot to work on!
  - What percent of our total time does this represent?
  - How much can we improve it? What is the "speed of light"?
  - How much will this improve our overall performance?

© NVIDIA 2013

Assess

- Let’s investigate…
  - Strong scaling and Amdahl’s Law
  - Weak scaling and Gustafson’s Law
  - Expected perf limiters: Bandwidth? Computation? Latency?

© NVIDIA 2013

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:

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

© NVIDIA 2013

Assess: Understanding Scaling

Weak Scaling

- A measure of how time to solution changes as more processors are added with fixed problem size *per processor*
- Linear weak scaling: overall problem size increases as num. of processors increases, but execution time remains constant

- Gustafson's Law:

$$S = N + (1 - P)(1 - N)$$

© NVIDIA 2013

Assess: Applying Strong and Weak Scaling

- Understanding which type of scaling is most applicable is an important part of estimating speedup:
  - Sometimes problem size will remain constant
  - Other times problem size will grow to fill the available processors
- Apply either Amdahl's or Gustafson's Law to determine an upper bound for the speedup

© NVIDIA 2013

Assess: Applying Strong Scaling

- **Recall that in this case we are wanting to optimize an existing kernel with a pre-determined workload**
- **That's strong scaling, so Amdahl's Law will determine the maximum speedup**

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID
1	spmv_kernel_v0<int=256>	8	3	59	35.96	457,088.169	7,700.182	7,747.257	7,809.367	1
2	jacobi_smooth_kernel_v0<int=256>	6	4	29	1.49	18,899.315	[illegible]	[illegible]	653.224	1
3	[illegible]	5	2	58	0.36	[illegible]	[illegible]	[illegible]	[illegible]	1
4	axpbypcz_kernel<int=256>	5	5	28	0.35	[illegible]	[illegible]	[illegible]	159.394	1
5	l2_norm_kernel_v0<int=256>	7	1	30	0.30	[illegible]	[illegible]	[illegible]	128.098	1
6	axpby_kernel<int=256>	5	4	31	0.29	[illegible]	[illegible]	[illegible]	[illegible]	1
7	jacobi_invert_diag_kernel_v0<int=256>	6	1	1	0.06	783.209	[illegible]	[illegible]	783.209	1
8	reduce_kernel<int=256>	5	1	58	0.03	366.597	[illegible]	[illegible]	7.168	1
9	reduce_l2_norm_kernel<int=256>	7	2	30	0.02	291.490	[illegible]	[illegible]	10.560	1

© NVIDIA 2013

Assess: Applying Strong Scaling

Say, for example, our kernel is ~93% of total time:

- Speedup $S = \frac{1}{(1-P)+\frac{P}{S_P}}$ (S_P = speedup in parallel part)
- In the limit when S_P is huge, S will approach $\frac{1}{1-0.93} \approx 14.3\times$
- In practice, it will be less than that depending on the S_P achieved
- Getting S_P to be high is the goal of optimizing, of course

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID
1	spmv_kernel_v0<int=256>	8	3	59	35.96	457,088.169	7,700.182	7,747.257	7,809.367	1
2	jacobi_smooth_kernel_v0<int=256>	6	4	29	1.49	18,899.315	[illegible]	[illegible]	653.224	1
3	[illegible]	5	2	58	0.36	[illegible]	[illegible]	[illegible]	[illegible]	1
4	axpbypcz_kernel<int=256>	5	5	28	0.35	[illegible]	[illegible]	[illegible]	159.394	1
5	l2_norm_kernel_v0<int=256>	7	1	30	0.30	[illegible]	[illegible]	[illegible]	128.098	1
6	axpby_kernel<int=256>	5	4	31	0.29	[illegible]	[illegible]	[illegible]	[illegible]	1
7	jacobi_invert_diag_kernel_v0<int=256>	6	1	1	0.06	783.209	[illegible]	[illegible]	783.209	1
8	reduce_kernel<int=256>	5	1	58	0.03	366.597	[illegible]	[illegible]	7.168	1
9	reduce_l2_norm_kernel<int=256>	7	2	30	0.02	291.490	[illegible]	[illegible]	10.560	1

~93%

Device Time (µs)
[illegible]
18,899.315
4,553.489
4,442.227
3,793.612
3,741.159
783.209
366.597
291.490

© NVIDIA 2013

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

© NVIDIA 2013

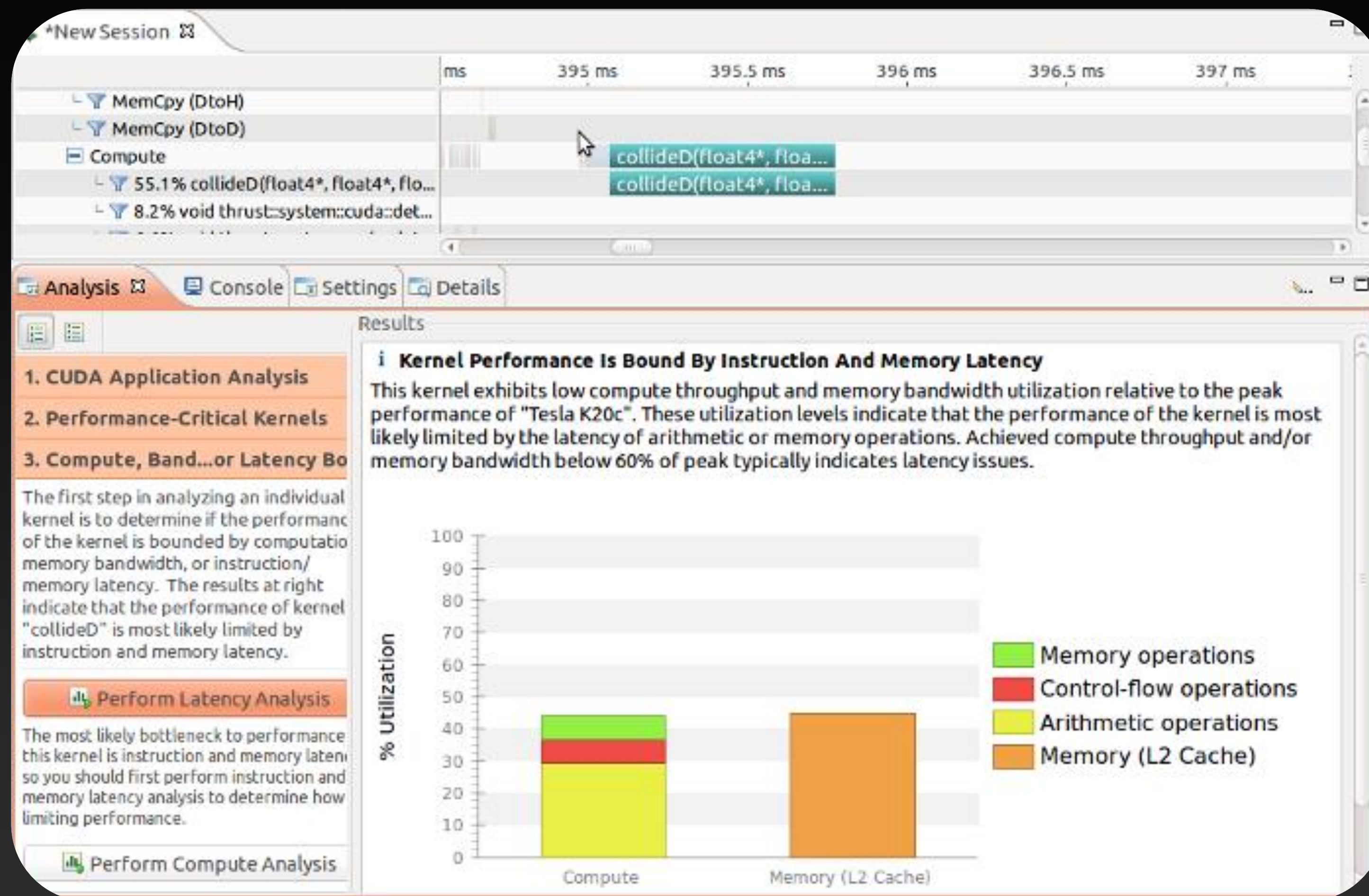
Assess: Limiting Factor

- Comparing bytes per instr. will give you a guess as to whether you’re likely to be bandwidth-bound or instruction-bound

- Comparing actual achieved GB/s vs. theory and achieved Ginstr/s vs. theory will give you an idea of how well you’re doing
  - If both are low, then you’re probably latency-bound and need to expose more (concurrent) parallelism

© NVIDIA 2013

Assess: Limiting Factor



© NVIDIA 2013

Assess: Speed of Light

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

- Consider SpMV: intuitively expect it to be bandwidth-limited
  - Say we discover we're getting only ~38% of peak bandwidth
  - If we aim to get this up to ~65% of peak, that's 1.7× for this kernel
  - 1.7× for this kernel translates into 1.6× overall due to Amdahl:

$$S = \frac{1}{(1-0.93)+\frac{0.93}{1.7}} \approx 1.6\times$$

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID
1	spmv_kernel_v0<int=256>	8	3	59	35.96	457,088.169	7,700.182	7,747.257	7,809.367	1
2	jacobi_smooth_kernel_v0<int=256>	6	4	29	1.49	18,899.315	[illegible]	[illegible]	653.224	1
3	dot_kernel_v0<int=256>	5	2	58	0.36	[illegible]	[illegible]	[illegible]	[illegible]	1
4	axpbypcz_kernel<int=256>	5	5	28	0.35	[illegible]	[illegible]	[illegible]	159.394	1
5	l2_norm_kernel_v0<int=256>	7	1	30	0.30	793.612	[illegible]	[illegible]	128.098	1
6	axpby_kernel<int=256>	5	4	31	0.29	[illegible]	[illegible]	[illegible]	[illegible]	1
7	jacobi_invert_diag_kernel_v0<int=256>	6	1	1	0.06	783.209	[illegible]	[illegible]	783.209	1
8	reduce_kernel<int=256>	5	1	58	0.03	366.597	[illegible]	[illegible]	7.168	1
9	reduce_l2_norm_kernel<int=256>	7	2	30	0.02	291.490	[illegible]	[illegible]	10.560	1

Device Time (µs)
18,899.315
4,553.489
4,442.227
3,793.612
3,741.159
783.209
366.597
291.490

~93%

© NVIDIA 2013

Assess: Limiting Factor

- For our example SpMV kernel, our first discovery was that we're latency-limited, not bandwidth, since utilization was so low
- This tells us our first "optimization" step actually needs to be related how we expose (memory-level) parallelism

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID
1	spmv_kernel_v0<int=256>	8	3	59	35.96	457,088.169	7,700.182	7,747.257	7,809.367	1
2	jacobi_smooth_kernel_v0<int=256>	6	4	29	1.49	18,899.315	[illegible]	[illegible]	653.224	1
3	[illegible]	5	2	58	0.36	[illegible]	[illegible]	[illegible]	[illegible]	1
4	axpbypcz_kernel<int=256>	5	5	28	0.35	[illegible]	[illegible]	[illegible]	159.394	1
5	l2_norm_kernel_v0<int=256>	7	1	30	0.30	793.612	[illegible]	[illegible]	128.098	1
6	axpby_kernel<int=256>	5	4	31	0.29	[illegible]	[illegible]	[illegible]	[illegible]	1
7	jacobi_invert_diag_kernel_v0<int=256>	6	1	1	0.06	783.209	[illegible]	[illegible]	783.209	1
8	reduce_kernel<int=256>	5	1	58	0.03	366.597	[illegible]	[illegible]	7.168	1
9	reduce_l2_norm_kernel<int=256>	7	2	30	0.02	291.490	[illegible]	[illegible]	10.560	1

~93%

Device Time (µs)
18,899.315
4,553.489
4,442.227
3,793.612
3,741.159
783.209
366.597
291.490

© NVIDIA 2013

PARALLELIZE

© NVIDIA 2013

PARALLELIZE

Computation

© NVIDIA 2013

Parallelize

Applications

Libraries

Compiler Directives

Programming Languages

Pick the best tool for the job

© NVIDIA 2013

Parallelize: e.g., with GPU Accelerated Libraries

NVIDIA cuBLAS

NVIDIA cuSPARSE

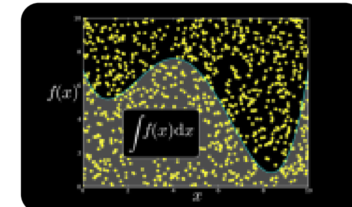
NVIDIA NPP

NVIDIA cuFFT

Matrix Algebra on GPU and Multicore

GPU Accelerated Linear Algebra

Vector Signal Image Processing


NVIDIA cuRAND

IMSL Library

CenterSpace NMath

ArrayFire

Building-block Algorithms

C++ Templated Parallel Algorithms

© NVIDIA 2013

Parallelize: e.g., with Thrust

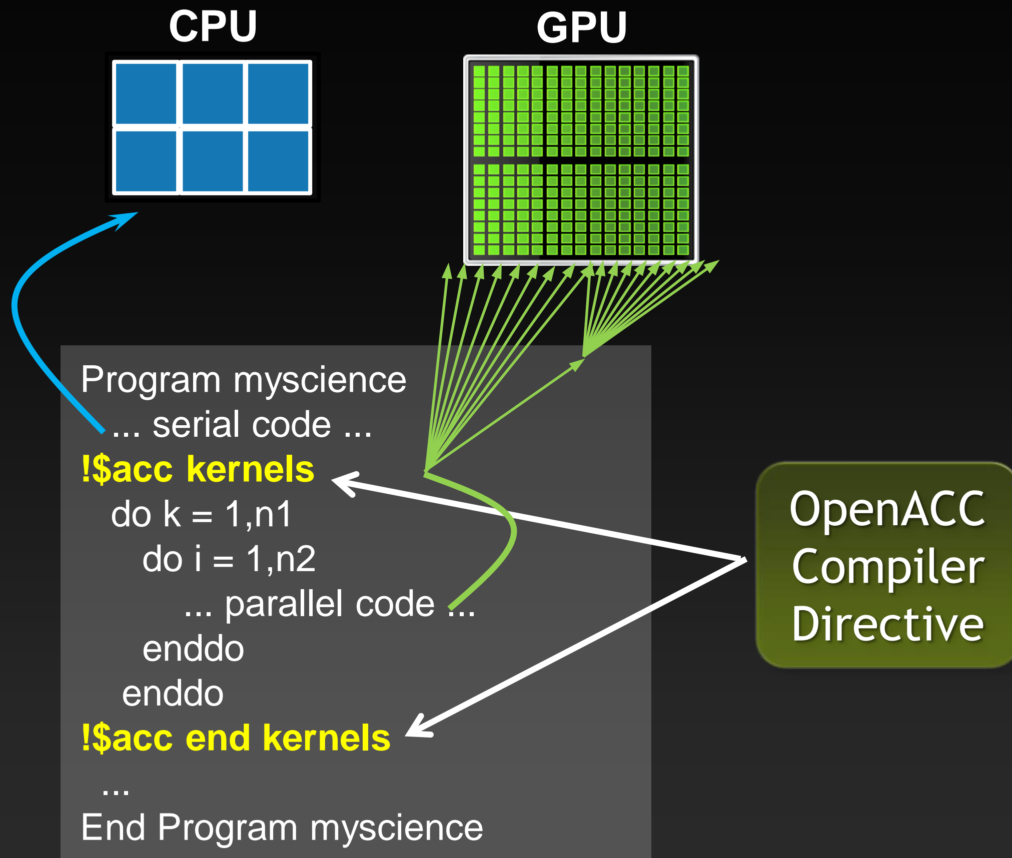
- Similar to C++ STL
- High-level interface
  - Enhances developer productivity
  - Enables performance portability between GPUs and multicore CPUs
- Flexible
  - Backends for CUDA, OpenMP, TBB
  - Extensible and customizable
  - Integrates with existing software
- Open source

```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

thrust.github.com or developer.nvidia.com/thrust

© NVIDIA 2013

Parallelize: e.g., with OpenACC



Directives-based approach

Compiler parallelizes code

Works on many-core GPUs & multicore CPUs

www.nvidia.com/gpudirectives

© NVIDIA 2013

Parallelize: e.g., with CUDA C

Standard C Code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

CUDA C Code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x * blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

developer.nvidia.com/cuda-toolkit

© NVIDIA 2013

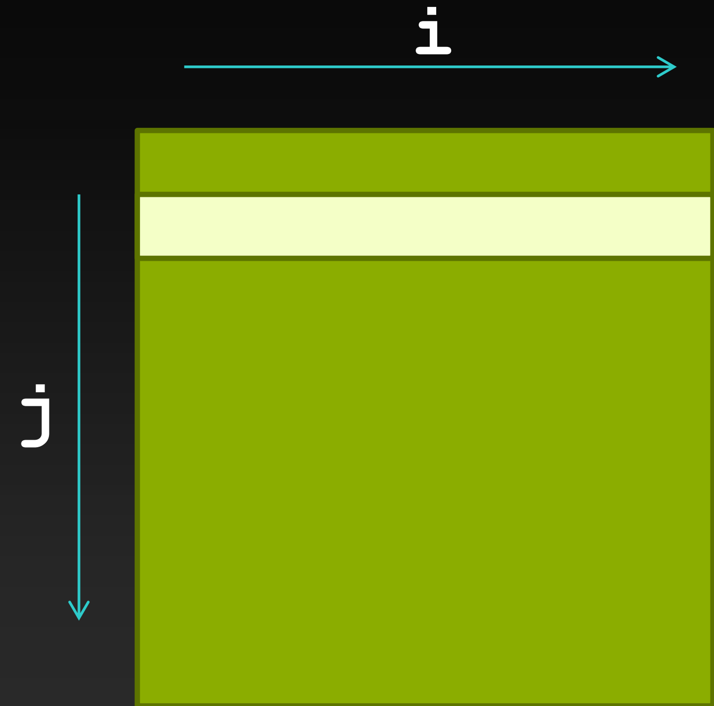
Parallelism Needed

- GPU is a parallel machine
  - Lots of arithmetic pipelines
  - Multiple memory banks
- To get good performance, your code must expose sufficient parallelism for 2 reasons:
  - To actually give work to all the pipelines
  - To hide latency of the pipelines
- Rough rule of thumb for Tesla K20X:
  - You want to have 14K or more threads running concurrently

© NVIDIA 2013

Case Study: Matrix Transpose

```
void transpose(float in[][], float out[][], int N)
{
  for(int j=0; j < N; j++)
    for(int i=0; i < N; i++)
      out[j][i] = in[i][j];
}
```



© NVIDIA 2013

An Initial CUDA Version

```
__global__ void transpose(float in[], float out[], int N)
{
  for(int j=0; j < N; j++)
     for(int i=0; i < N; i++)
       out[i*N+j] = in[j*N+i];

}


float in[N*N], out[N*N];
…
transpose<<<1,1>>>(in, out, N);
```

+ Quickly implemented

- Performance weak

⇒Need to expose parallelism!

© NVIDIA 2013

An Initial CUDA Version

```
__global__ void transpose(float in[], float out[], int N)
{
  for(int j=0; j < N; j++)
     for(int i=0; i < N; i++)
       out[i*N+j] = in[j*N+i];

}


float in[N*N], out[N*N];
…
transpose<<<1,1>>>(in, out, N);
```

+ Quickly implemented

- Performance weak

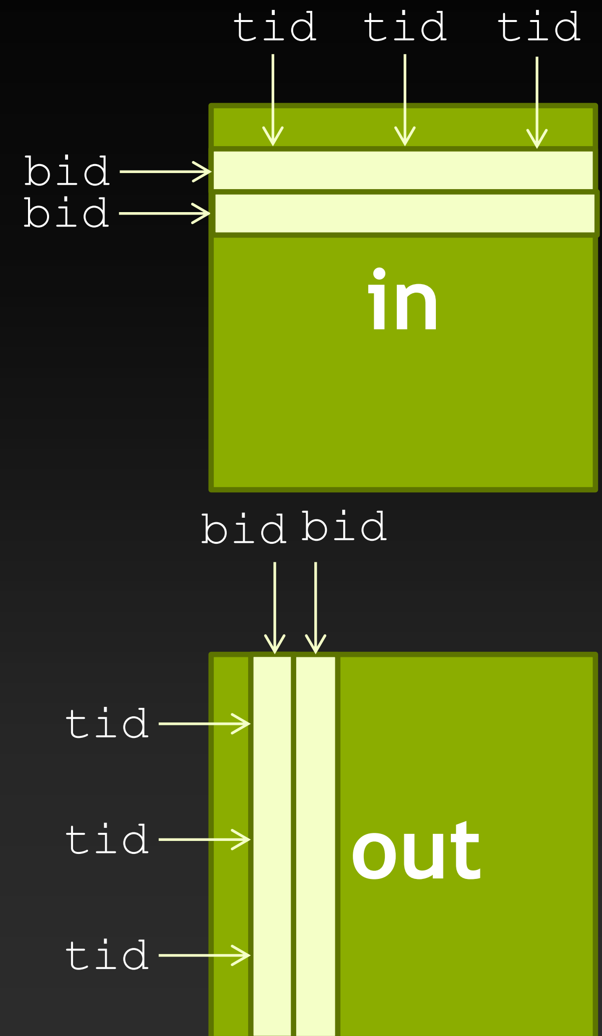
⇒Need to expose parallelism!

© NVIDIA 2013

Parallelize across matrix elements

Process elements independently

```
__global__ transpose(float in[], float out[])
{
  int tid = threadIdx.x;
  int bid = blockIdx.x;

  out[tid*N+bid] = in[bid*N+tid];
}


float in[], out[];
…
transpose<<<N,N>>>(in, out);
```



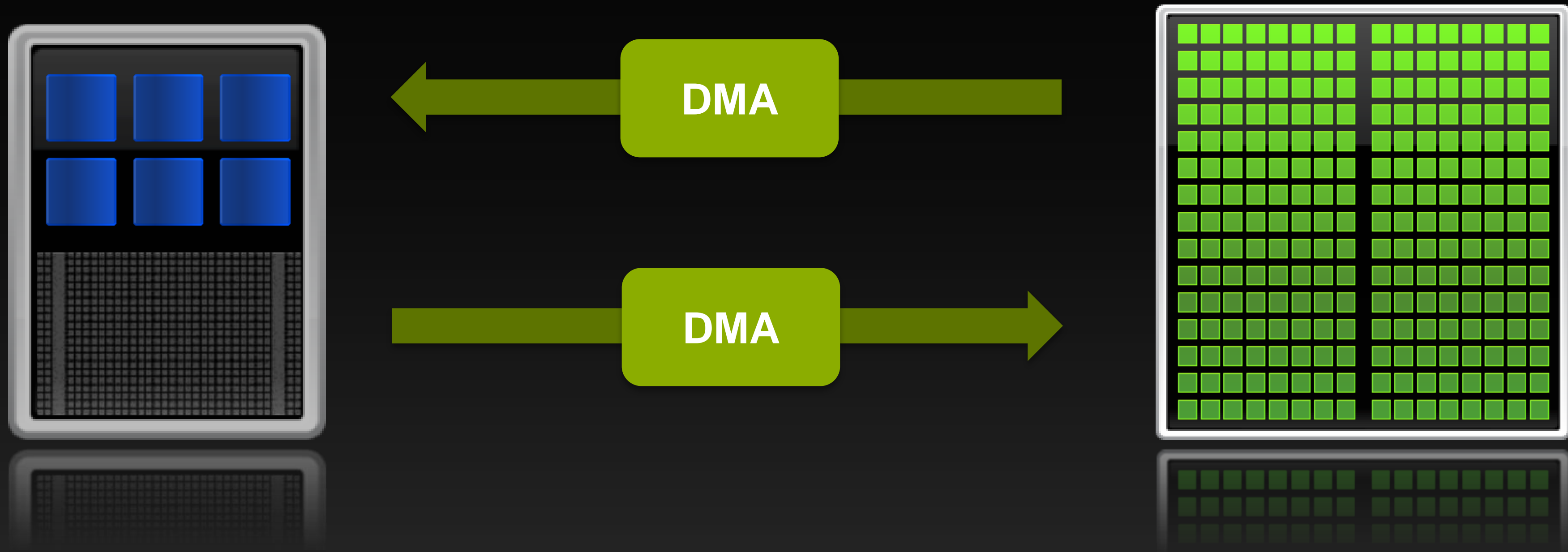
© NVIDIA 2013

PARALLELIZE

Data Transfer

© NVIDIA 2013

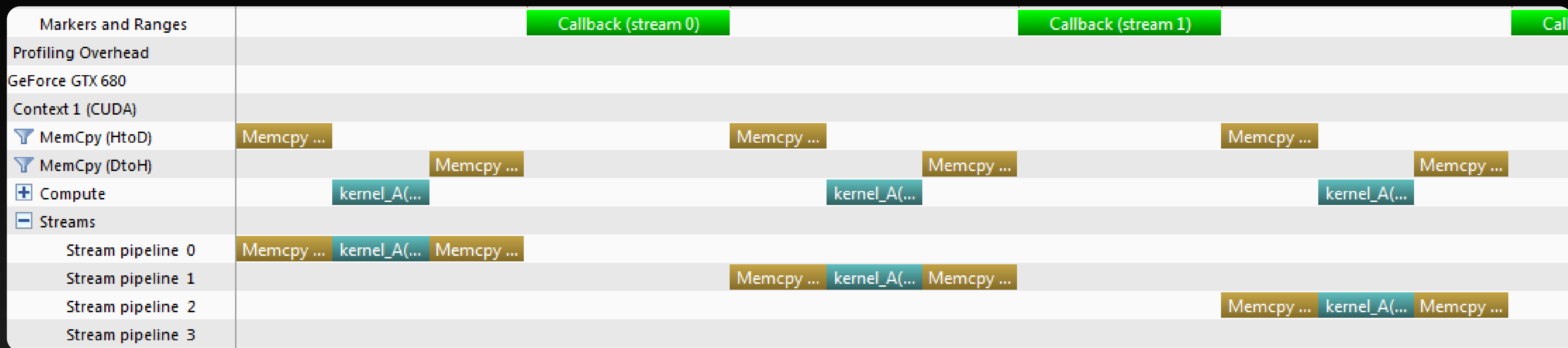
Asynchronicity = Overlap = Parallelism



Heterogeneous system: overlap work and data movement

© NVIDIA 2013

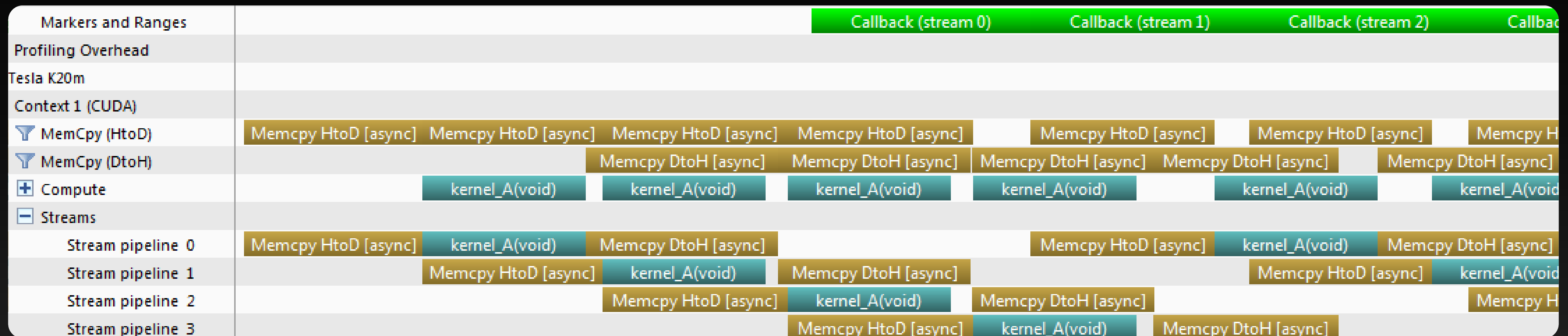
Asynchronicity



- 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

© NVIDIA 2013

Parallelize: Achieve Asynchronicity



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

© NVIDIA 2013

OPTIMIZE

© NVIDIA 2013

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

© NVIDIA 2013

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/

© NVIDIA 2013

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

© NVIDIA 2013

OPTIMIZE

Kernel Optimizations: ***Kernel Launch Configuration***

© NVIDIA 2013

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:

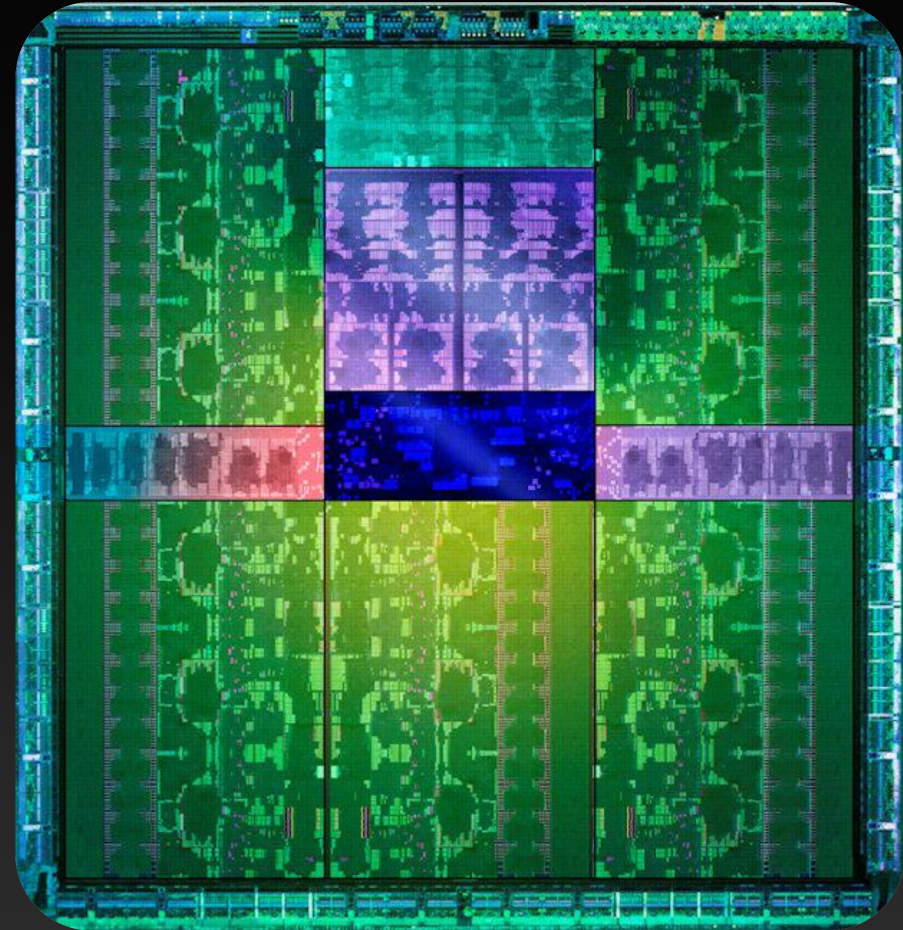
```
mykernel<<<blocks_per_grid,threads_per_block>>>(…);
```

- 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*

© NVIDIA 2013

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

© NVIDIA 2013

Kepler Streaming Multiprocessor (SMX)

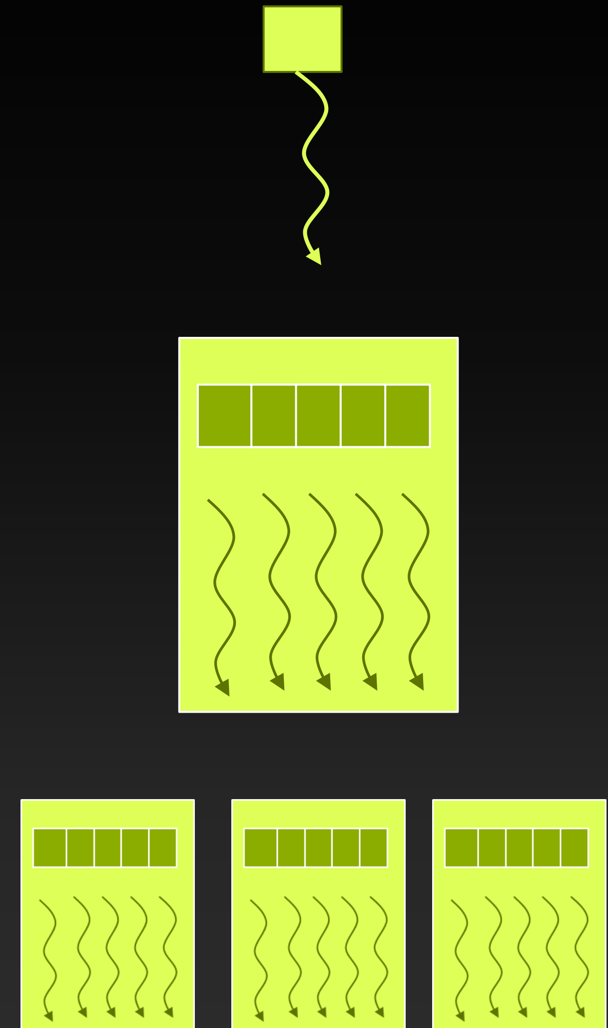
Per SMX:

- 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)

© NVIDIA 2013

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

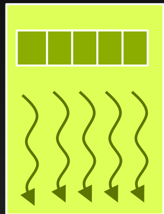


© NVIDIA 2013

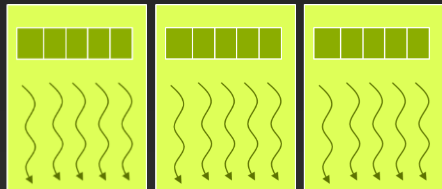
Execution Model

Software

Thread



Thread Block

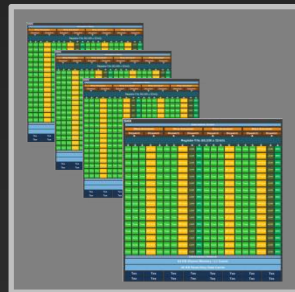


Grid

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

© NVIDIA 2013

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

© NVIDIA 2013

Launch Configuration: General Guidelines

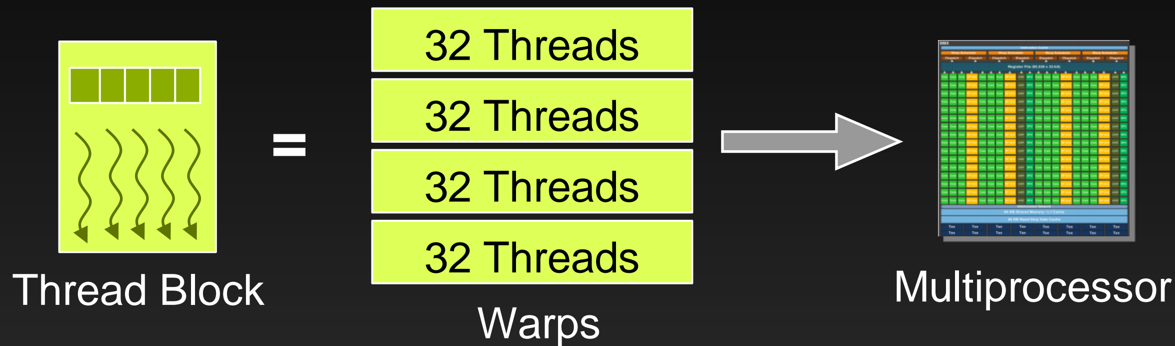
How many threads per block should we choose?

- 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

© NVIDIA 2013

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)

© NVIDIA 2013

Hardware Levels of Parallelism

Single Instruction, Multiple Data
In-core parallelism

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

Multiple “computers”
Tightly-coupled
Supercomputing apps

SIMD **SMT** **MPI**

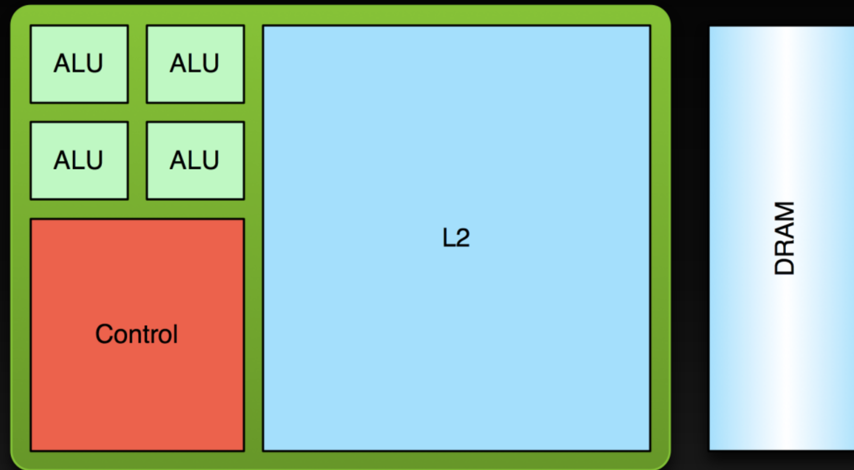
SIMT

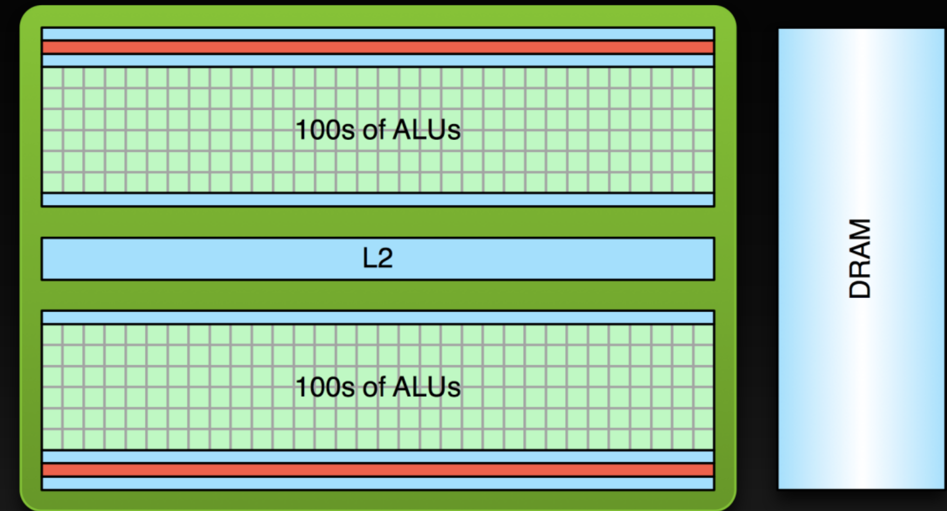
Single Instruction, Multiple Threads
In-processor parallelism
Many threads on many cores

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

© NVIDIA 2013

Low Latency or High Throughput?





CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

© NVIDIA 2013

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

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

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%

© NVIDIA 2013

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

W_4

W_3

W_2

W_1

CPU core – Low-latency Processor

T_1 T_2 T_3 T_4

Computation Thread/Warp

T_n Processing

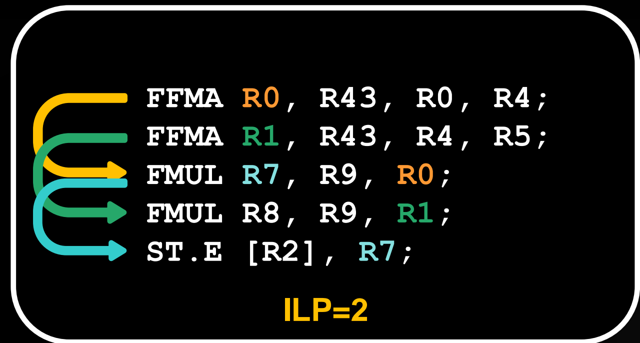
Waiting for data

Ready to be processed

Context switch

© NVIDIA 2013

Latency Hiding



- 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

© NVIDIA 2013

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

© NVIDIA 2013

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

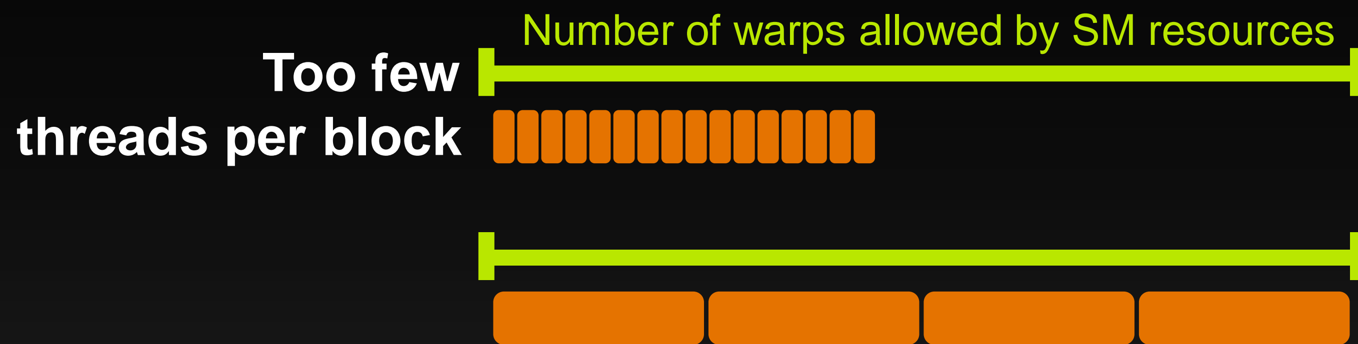
© NVIDIA 2013

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

© NVIDIA 2013

Thread Block Sizing



SM resources:

- Registers
- Shared memory

© NVIDIA 2013

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

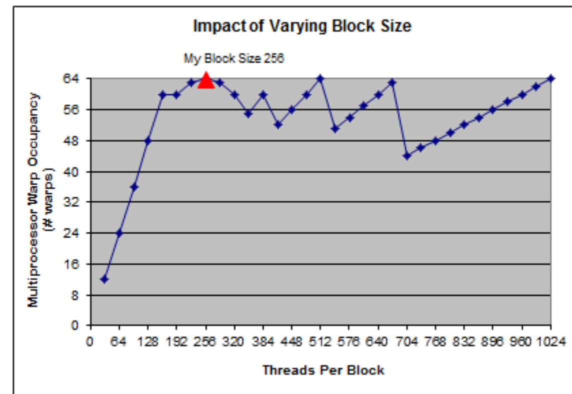
Note: SM is an abbreviation for (Streaming) Multiprocessor

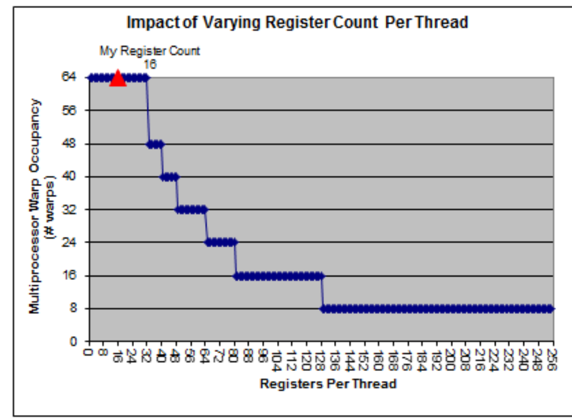
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiproces	8	8	64
Limited by Registers per Multiprocessor	16		
Limited by Shared Memory per Multiprocessor	12		

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.





© NVIDIA 2013

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%

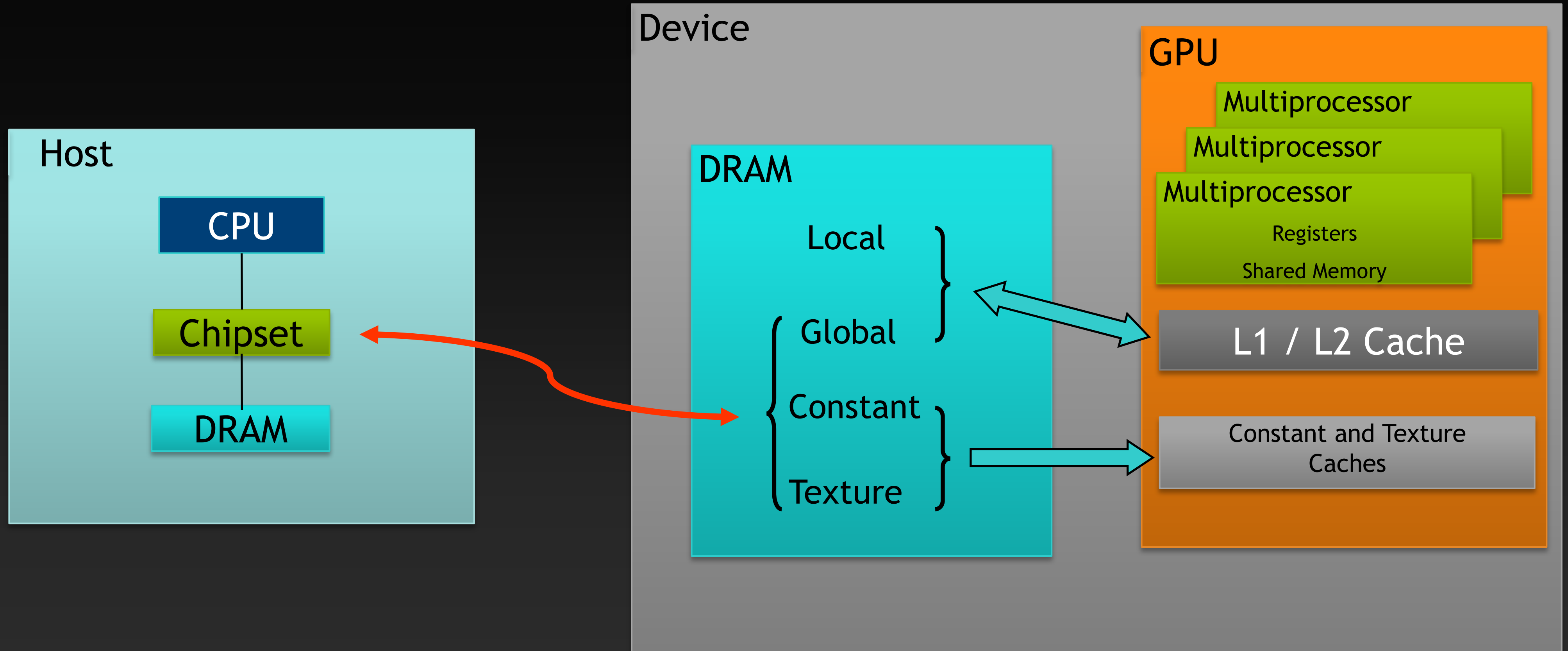
© NVIDIA 2013

OPTIMIZE

Kernel Optimizations: *Global Memory Throughput*

© NVIDIA 2013

CUDA Memory Architecture



© NVIDIA 2013

Optimizing Memory Throughput

- **Goal: utilize all available memory bandwidth**

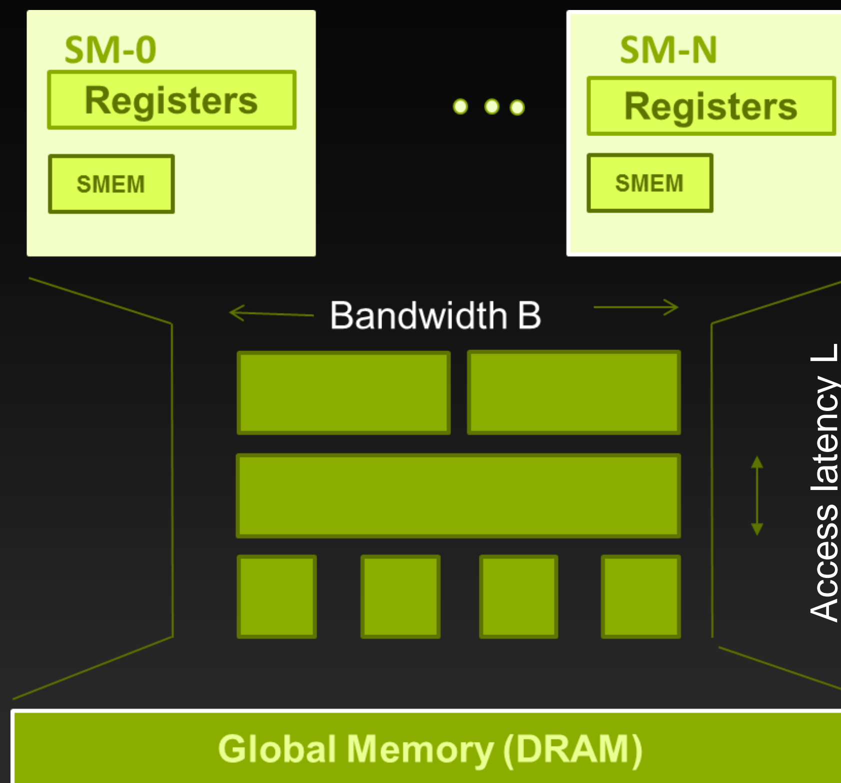
- **Little's Law:**

  # bytes in flight = latency * bandwidth

⇒ Increase parallelism (bytes in flight)

(or)

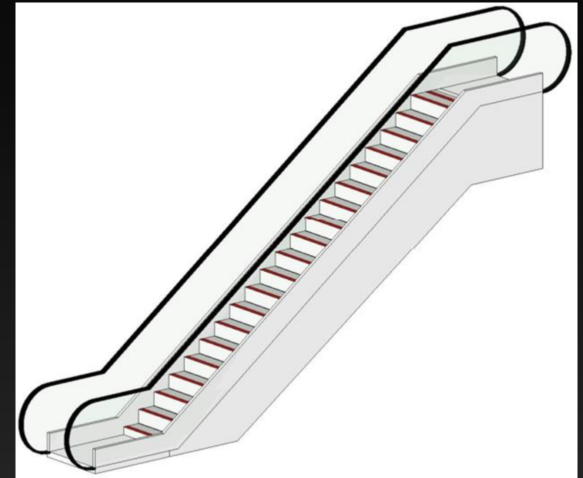
⇒ Reduce latency (time between requests)



© NVIDIA 2013

Illustration: Little's Law for Escalators

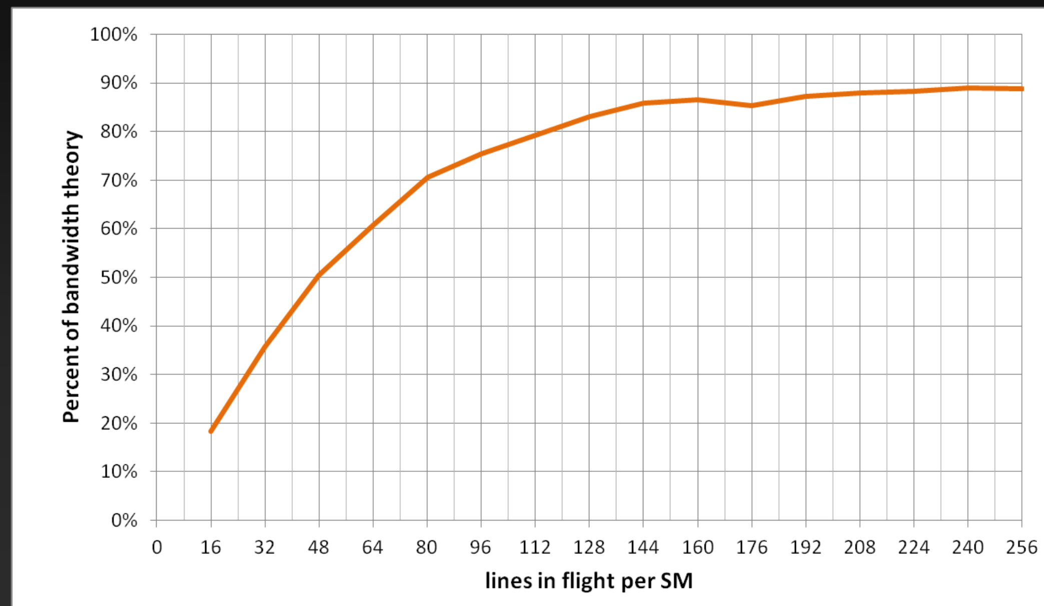
- Say the parameters of our escalator are:
  - 1 person fits on each step
  - Step arrives every 2 secs (*bandwidth*=0.5 persons/s)
  - 20 steps tall (*latency*=40 seconds)
- 1 person in flight: 0.025 persons/s achieved
- To saturate bandwidth:
  - Need 1 person arriving every 2 s
  - Means we'll need 20 persons in flight
- The idea: Bandwidth × Latency
  - It takes *latency* time units for the first person to arrive
  - We need *bandwidth* persons to get on the escalator every time unit



© NVIDIA 2013

Memory-Level Parallelism = Bandwidth

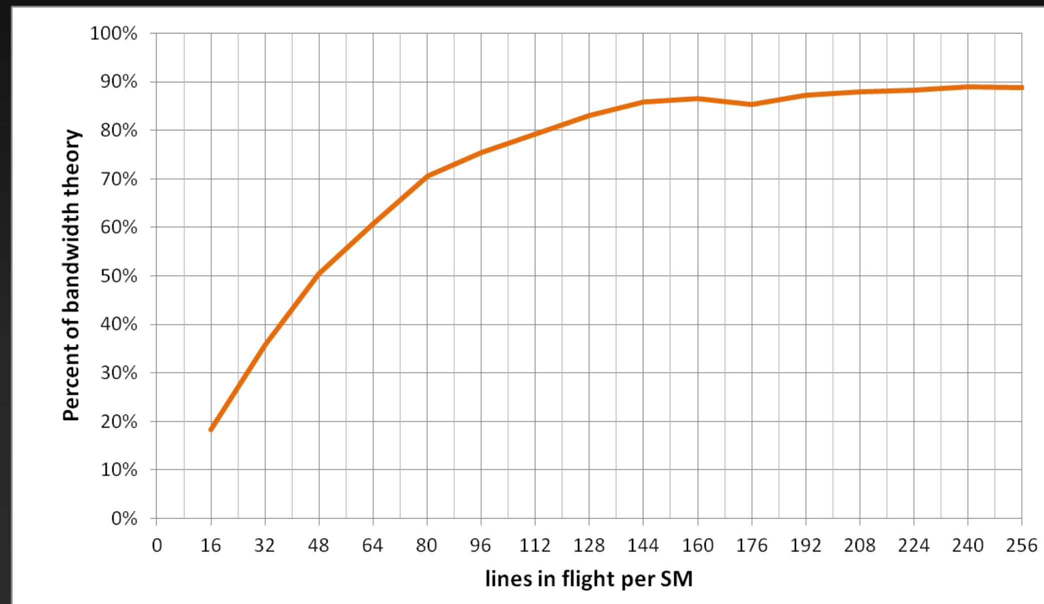
- **In order to saturate memory bandwidth, SM must have enough independent memory requests in flight concurrently**



© NVIDIA 2013

Memory-Level Parallelism: Requests in flight

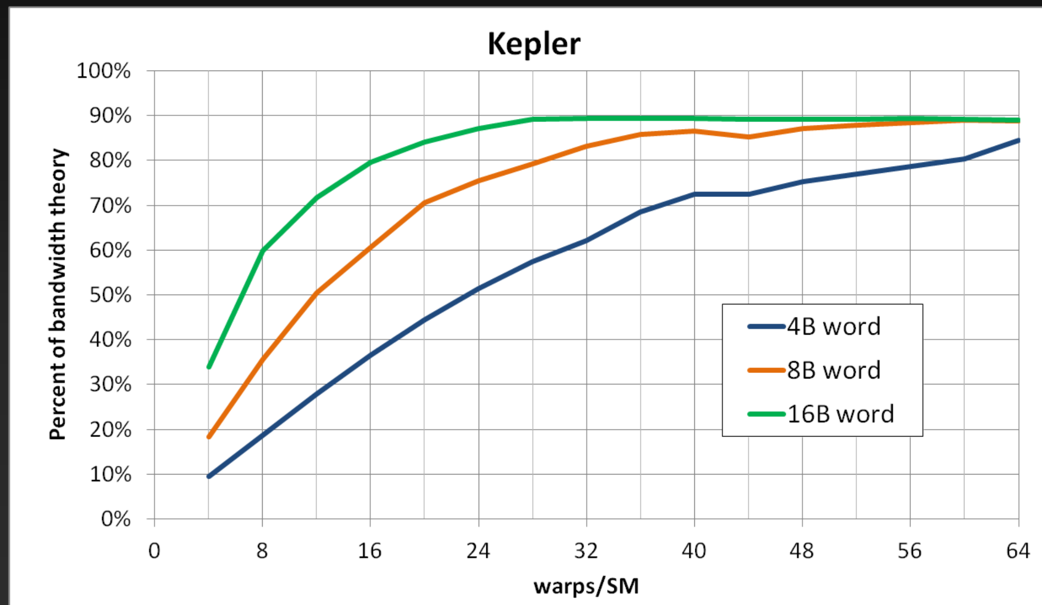
- Achieved Kepler memory throughput
  - Shown as a function of number of concurrent requests per SM with 128-byte lines



© NVIDIA 2013

Requests per Thread and Performance

- **Experiment: vary size of accesses by threads of a warp, check performance**
  - Memcopy kernel: each warp has 2 concurrent requests (one write and the read following it)



Accesses by a warp:

4B words: 1 line

8B words: 2 lines

16B words: 4 lines

To achieve same throughput at lower occupancy or with smaller words, need more independent requests per warp

© NVIDIA 2013

Optimizing Access Concurrency

- Ways to increase concurrent accesses:
  - Increase occupancy (run more warps concurrently)
    - Adjust block dimensions to maximize occupancy
    - If occupancy is limited by registers per thread, try to reduce register count (-maxrregcount option or __launch_bounds__)
  - Modify code to process several elements per thread
    - Doubling elements per thread doubles independent accesses per thread

© NVIDIA 2013

OPTIMIZE

Kernel Optimizations: *Global Memory Access Coalescing*

© NVIDIA 2013

Mechanics of a Memory Access

- Memory operations are issued *per warp*
  - Just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
  - Hardware determines which lines/segments are needed, fetches them

© NVIDIA 2013

Memory Access Efficiency Analysis

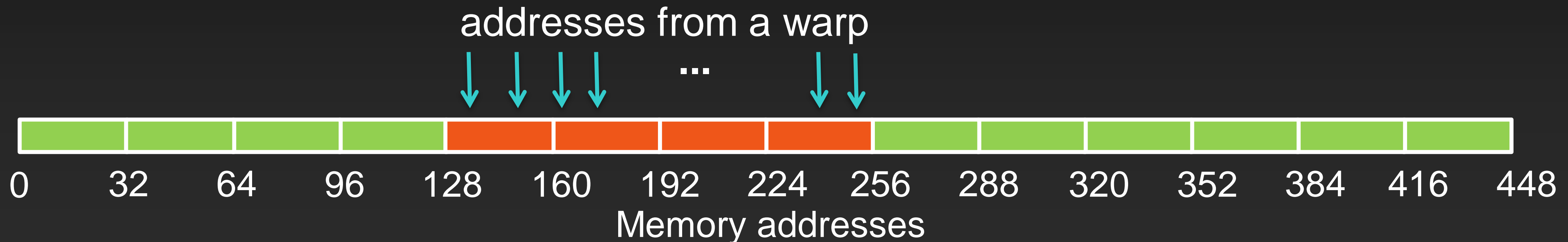
- Two perspectives on the throughput:
  - Application's point of view: count only bytes requested by application
  - HW point of view: count all bytes moved by hardware

- The two views can be different:
  - Memory is accessed at 32 byte granularity
    - With a scattered or offset pattern, the application doesn't use all the bytes the hardware actually transferred
  - Broadcast: the same small transaction serves many threads in a warp

© NVIDIA 2013

Access Patterns vs. Memory Throughput

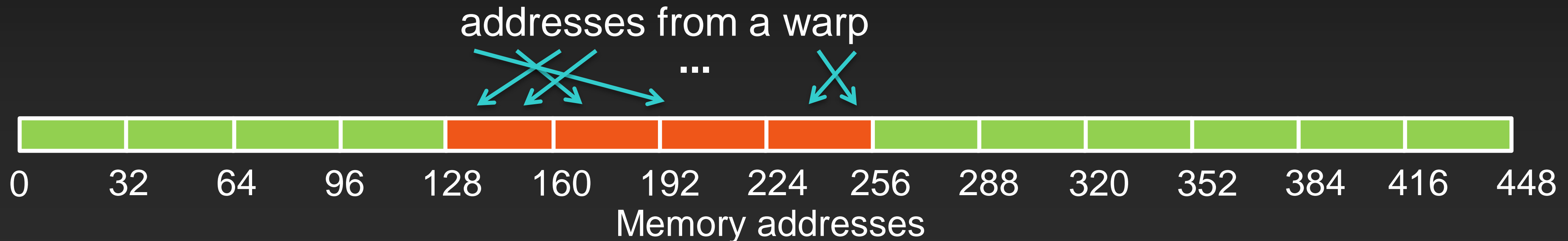
- **Scenario:**
  - **Warp requests 32 aligned, consecutive 4-byte words**
- **Addresses fall within 4 segments**
  - **Warp needs 128 bytes**
  - **128 bytes move across the bus**
  - **Bus utilization: 100%**



© NVIDIA 2013

Access Patterns vs. Memory Throughput

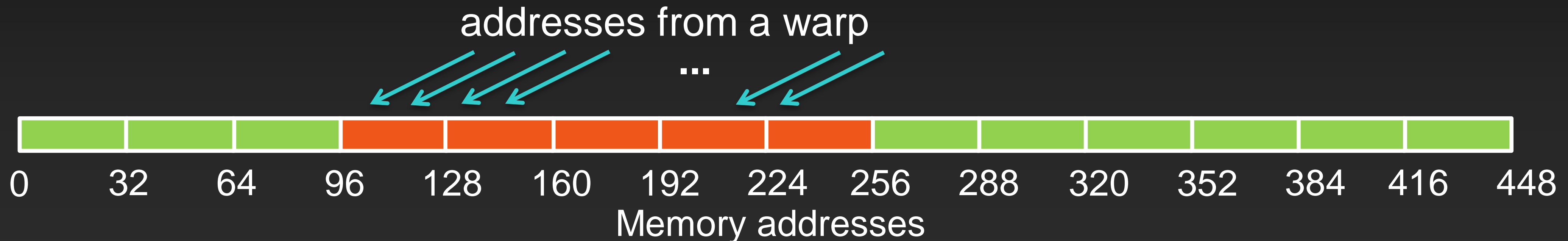
- **Scenario:**
  - **Warp requests 32 aligned, permuted 4-byte words**
- **Addresses fall within 4 segments**
  - **Warp needs 128 bytes**
  - **128 bytes move across the bus**
  - **Bus utilization: 100%**



© NVIDIA 2013

Access Patterns vs. Memory Throughput

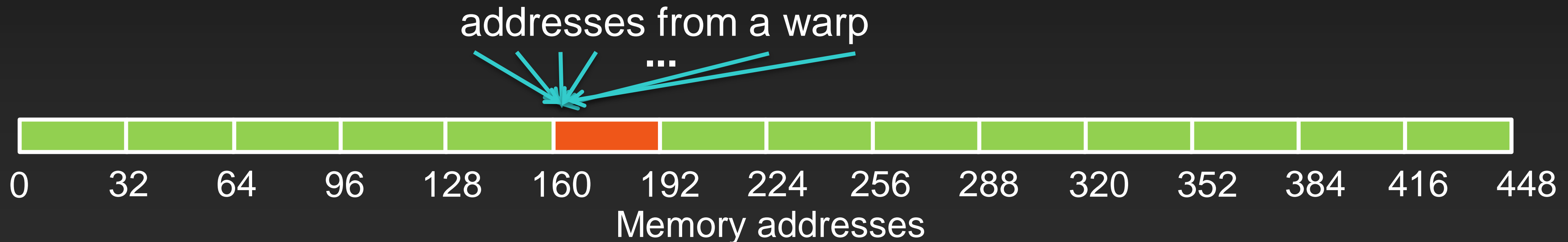
- Scenario:
  - Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization



© NVIDIA 2013

Access Patterns vs. Memory Throughput

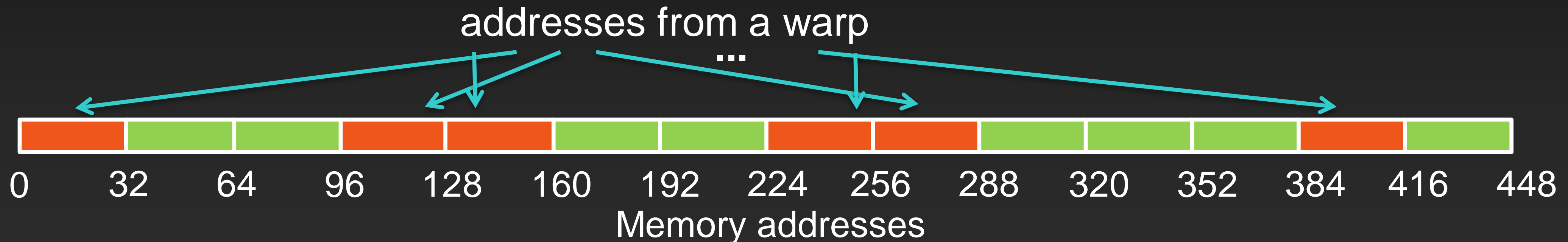
- **Scenario:**
  - **All threads in a warp request the same 4-byte word**
- **Addresses fall within a single segment**
  - **Warp needs 4 bytes**
  - **32 bytes move across the bus**
  - **Bus utilization: 12.5%**



© NVIDIA 2013

Access Patterns vs. Memory Throughput

- **Scenario:**
  - **Warp requests 32 scattered 4-byte words**
- **Addresses fall within *N* segments**
  - **Warp needs 128 bytes**
  - ***N**32 bytes move across the bus**
  - **Bus utilization: 128 / (*N**32)**



© NVIDIA 2013

Parallelizing SAXPY

```
void saxpy(int n, float a, float * x, float
    * y)
{
  for(int i=0; i<n; i++)
  {
   y[base +i] += a * x[base+ i];
  }
}
```

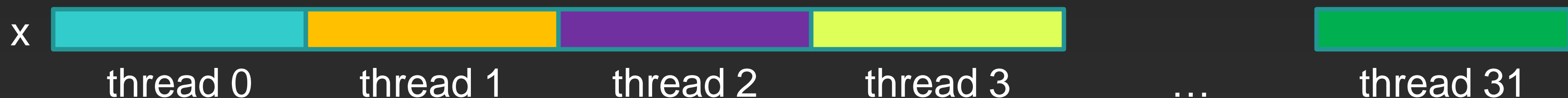
- Divide the work equally among T threads
- Each thread is responsible for computing one contiguous 'region' of the arrays
- This is good for pthreads

© NVIDIA 2013

Parallelizing SAXPY

```
__global__ void saxpy1(int n, float a, float
    * x, float * y)
{
  int workPerThread = 1 + n/blockDim.x;
  int base = threadIdx.x * workPerThread;

  for(int i=0; i<workPerThread; i++)
  {
    if(base + i < n)
    {
      y[base +i] += a * x[base+ i];
    }
  }
}
```

- Divide the work equally among T threads
- Each thread is responsible for computing one contiguous 'region' of the arrays
- This is good for pthreads

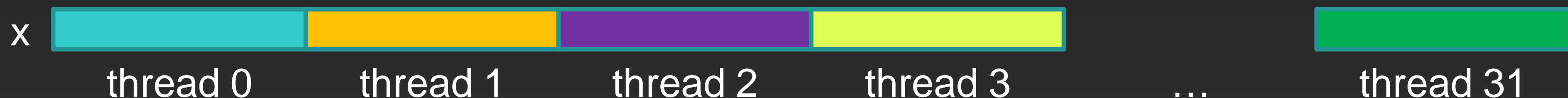


© NVIDIA 2013

Parallelizing SAXPY

```
__global__ void saxpy1(int n, float a, float
   * x, float * y)
{
  int workPerThread = 1 + n/blockDim.x;
  int base = threadIdx.x * workPerThread;

  for(int i=0; i<workPerThread; i++)
  {
    if(base + i < n)
    {
      y[base +i] += a * x[base+i];
    }
  }
}
```

- **In SIMT, 32 threads of a warp issue the x[base+i] instruction simultaneously.**
  - **Each thread has different value of base**
- **if workPerThread > 1, this becomes a strided load**

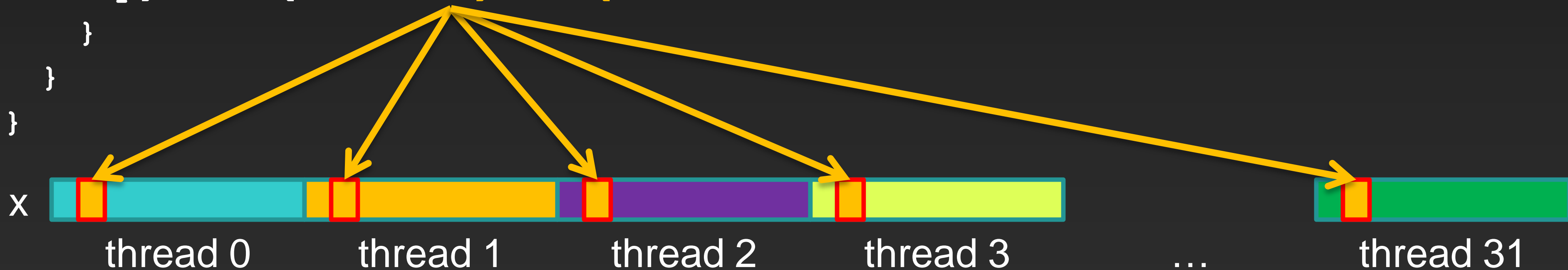


© NVIDIA 2013

Parallelizing SAXPY

```
__global__ void saxpy1(int n, float a, float
    * x, float * y)
{
  int workPerThread = 1 + n/blockDim.x;
  int base = threadIdx.x * workPerThread;

  for(int i=0; i<workPerThread; i++)
  {
    if(base + i < n)
    {
      y[base +i] += a * x[base+i];
    }
  }
}
```

- In SIMT, 32 threads of a warp issue the x[base+i] instruction simultaneously.
  - Each thread has different value of base
- if workPerThread > 1, this becomes a strided load

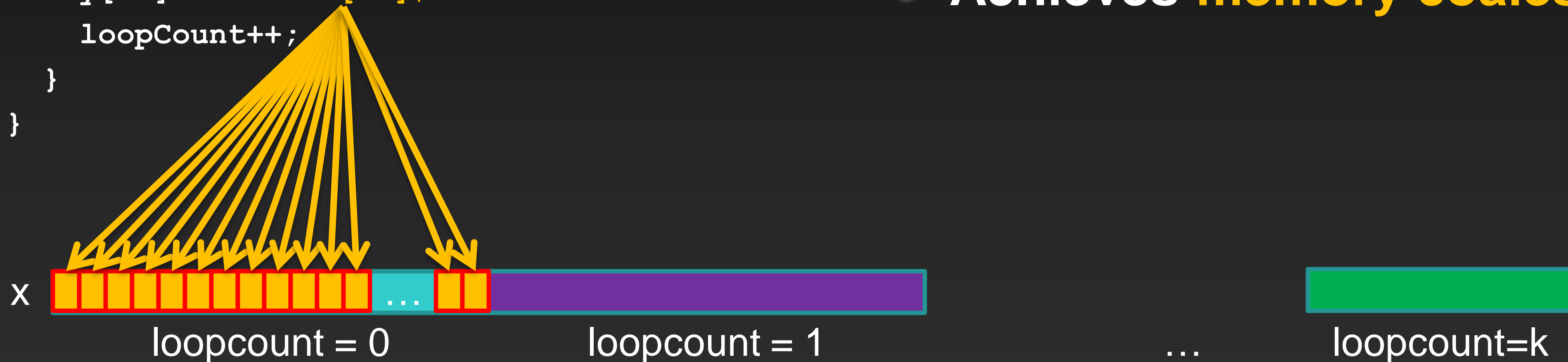


© NVIDIA 2013

A Better Way to Parallelize SAXPY

```
__global__ void saxpy2(int n, float a, float
    * x, float * y)
{
  int id;
  int loopCount = 0;
  while(id < n)
  {
    id = loopCount*blockDim.x + threadIdx.x;
    y[id] += a * x[id];
    loopCount++;
  }
}
```

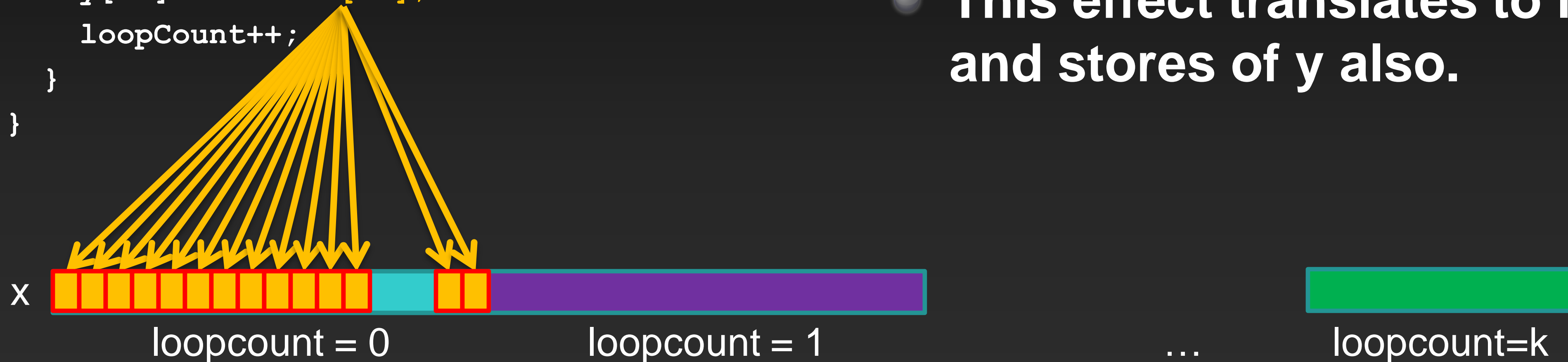
- Divide work up so that each pass through the loop, the thread block computes one ‘contiguous region’ of the array.
- Achieves memory coalescing



© NVIDIA 2013

A Better Way to Parallelize SAXPY

```
__global__ void saxpy2(int n, float a, float
    * x, float * y)
{
  int id;
  int loopCount = 0;
  while(id < n)
  {
    id = loopCount*blockDim.x + threadIdx.x;
    y[id] += a * x[id];
    loopCount++;
  }
}
```



- The area of X addressed by each **warp** is contiguous in global memory.
- The number of global memory transactions is minimized.
- This effect translates to loads and stores of y also.

© NVIDIA 2013

Structures of Non-Native Size

- Say we are reading a 12-byte structure per thread

```
struct Position
{
    float x, y, z;
};
...
__global__ void kernel( Position *data, ... )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    Position temp = data[idx];
    ...
}
```

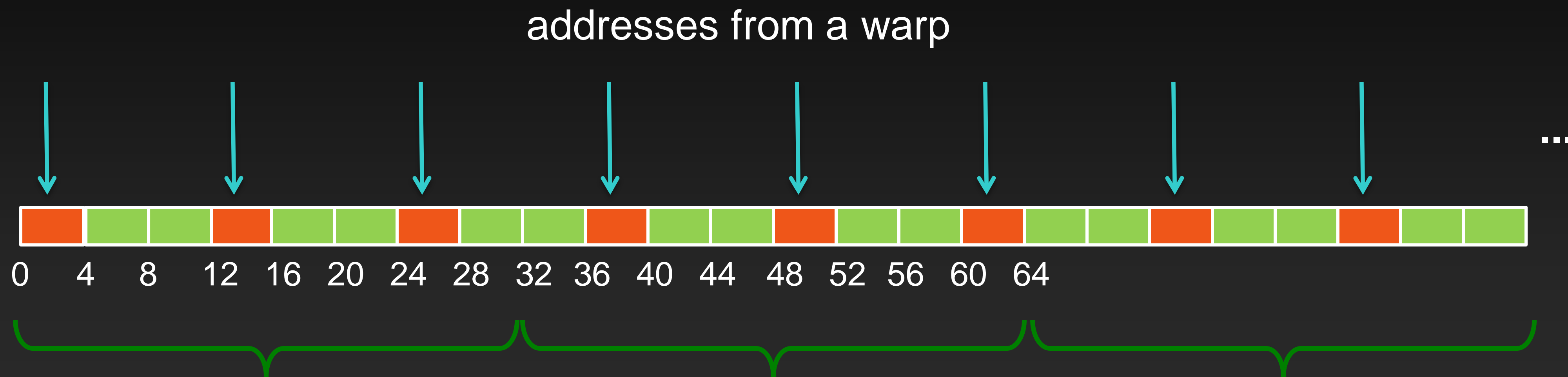
© NVIDIA 2013

Structure of Non-Native Size

- Compiler converts `temp = data[idx]` into 3 loads:
  - Each loads 4 bytes
  - Can't do an 8 and a 4 byte load: 12 bytes per element means that every other element wouldn't align the 8-byte load on 8-byte boundary
- Addresses per warp for each of the loads:
  - Successive threads read 4 bytes at 12-byte stride

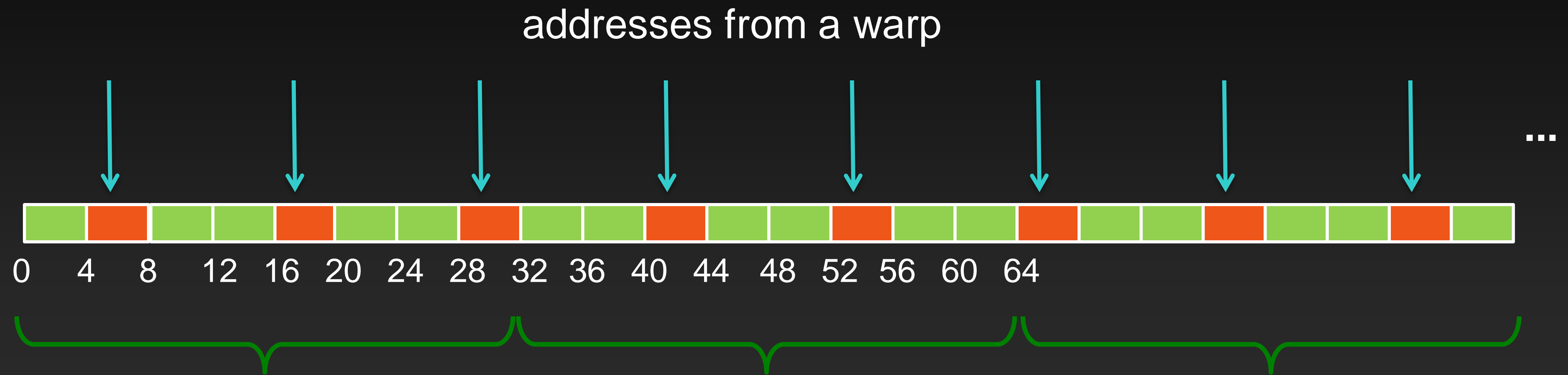
© NVIDIA 2013

First Load Instruction



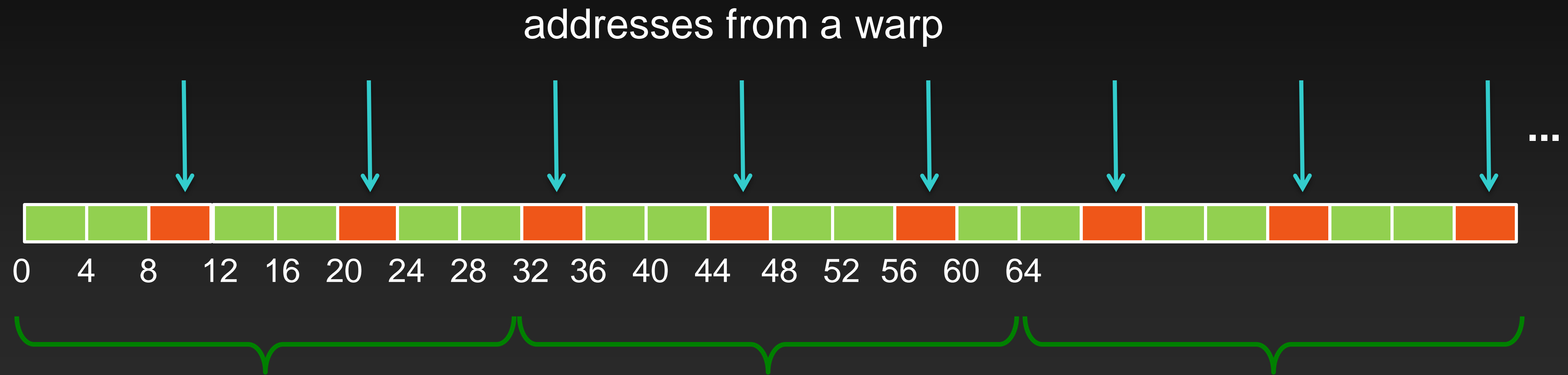
© NVIDIA 2013

Second Load Instruction



© NVIDIA 2013

Third Load Instruction



© NVIDIA 2013

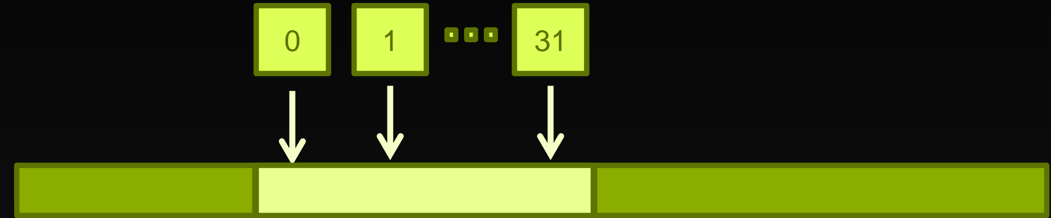
Performance and Solutions

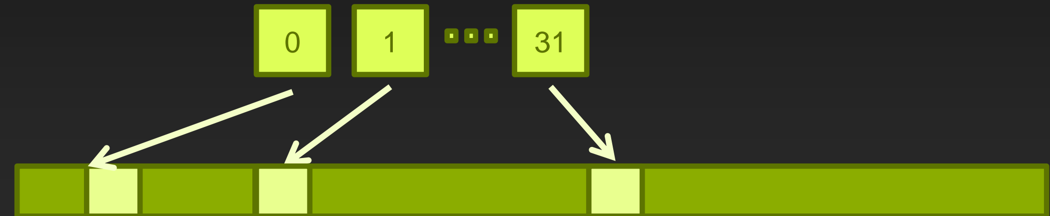
- Because of the address pattern, we end up moving 3x more bytes than application requests
  - We waste a lot of bandwidth, leaving performance on the table
- Potential solutions:
  - Change data layout from array of structures to structure of arrays
    - In this case: 3 separate arrays of floats
    - The most reliable approach (also ideal for both CPUs and GPUs)
  - Use loads via read-only cache
    - As long as lines survive in the cache, performance will be nearly optimal
  - Stage loads via shared memory

© NVIDIA 2013

Global Memory Access Patterns

- **SoA vs AoS:**
  - Good: point.x[i]
  - Not so good: point[i].x
- **Strided array access:**
  - ~OK: x[i] = a[i+1] – a[i]
  - Slower: x[i] = a[64*i] – a[i]
- **Random array access:**
  - Slower: a[rand(i)]





© NVIDIA 2013

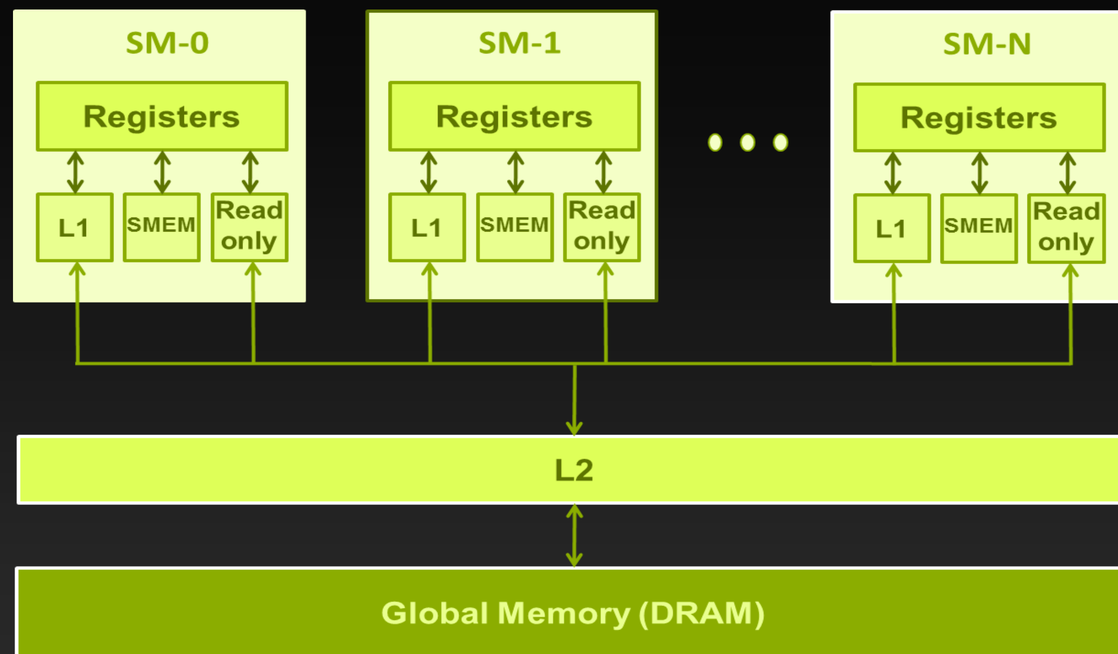
Summary: GMEM Optimization

- Strive for perfect address coalescing per warp
  - Align starting address (may require padding)
  - A warp will ideally access within a contiguous region
  - Avoid scattered address patterns or patterns with large strides between threads
- Analyze and optimize address patterns:
  - Use profiling tools (included with CUDA toolkit download)
  - Compare the transactions per request to the ideal ratio
  - Choose appropriate data layout (prefer SoA)
  - If needed, try read-only loads, staging accesses via SMEM

© NVIDIA 2013

A note about caches

- L1 and L2 caches
  - Ignore in software design
  - Thousands of concurrent threads – cache blocking difficult at best
- Read-only Data Cache
  - Shared with texture pipeline
  - Useful for uncoalesced reads
  - Handled by compiler when `const __restrict__` is used, or use `__ldg()` primitive



© NVIDIA 2013

Blocking for GPU Memory Caches

- Short answer: DON'T
- GPU caches are not intended for the same use as CPU caches
  - Smaller size (especially per thread), so not aimed at temporal reuse
  - Intended to smooth out some access patterns, help with spilled registers, etc.
- Usually not worth trying to cache-block like you would on CPU
  - 100s to 1,000s of run-time scheduled threads competing for the cache
  - If it is possible to block for L1 then it's possible block for SMEM
    - Same size
    - Same or higher bandwidth
    - Guaranteed locality: hw will not evict behind your back

© NVIDIA 2013

Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: const __restrict__
    - All other pointer arguments: __restrict__
      – Conveys to compiler that no aliasing will occur
  - Using __ldg() intrinsic
    - Requires no pointer decoration

© NVIDIA 2013

Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer argumen
    - Pointer of interest: const _
    - All other pointer argument
      – Conveys to compiler th
  - Using __ldg() intrinsic
    - Requires no pointer decor

```
__global__ void kernel(
                  int* __restrict__ output,
            const int* __restrict__ input )
{
    ...
    output[idx] = input[idx];
}
```

© NVIDIA 2013

Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer argumen
    - Pointer of interest: const _
    - All other pointer argument
      - Conveys to compiler th
  - Using __ldg() intrinsic
    - Requires no pointer decor

```
__global__ void kernel( int *output,
                        int *input )
{
    ...
    output[idx] = __ldg( &input[idx] );
}
```

© NVIDIA 2013

Texture and Constant Memory

- Read-only
- Data resides in global memory
- Read via special-purpose caches

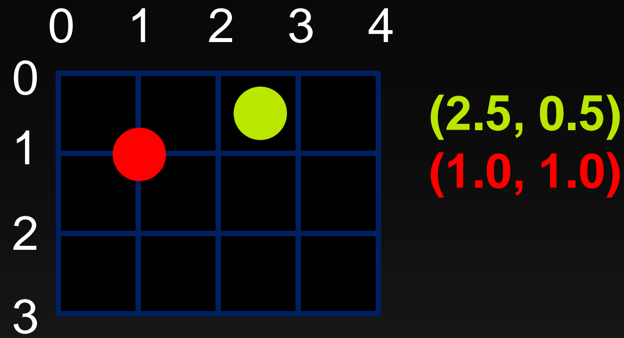
© NVIDIA 2013

Texture

- Separate cache
- Dedicated texture cache hardware provides:
  - Out-of-bounds index handling
    - clamp or wrap-around
  - Optional interpolation
    - Think: using fp indices for arrays
    - Linear, bilinear, trilinear
      - Interpolation weights are 9-bit
  - Optional format conversion
    - {char, short, int} -> float
  - All of these are “free”

© NVIDIA 2013

Examples of Texture Object Indexing

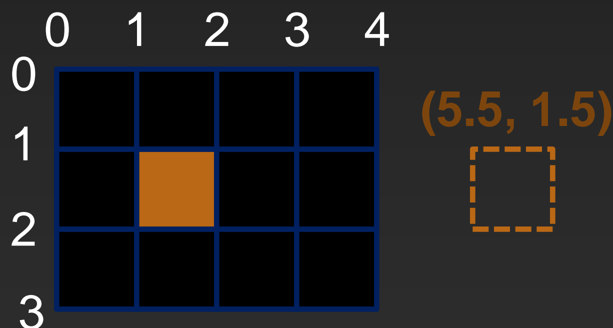


Integer indices fall between elements

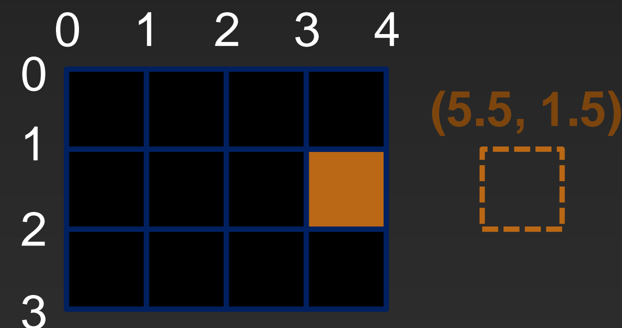
Optional interpolation:

Weights are determined by coordinate distance

Index Wrap:



Index Clamp:



© NVIDIA 2013

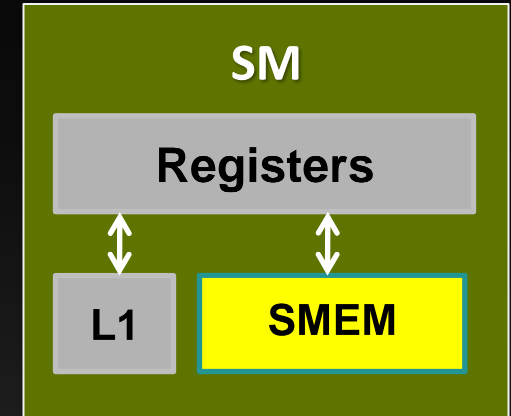
OPTIMIZE

Kernel Optimizations: *Shared Memory Accesses*

© NVIDIA 2013

Shared Memory

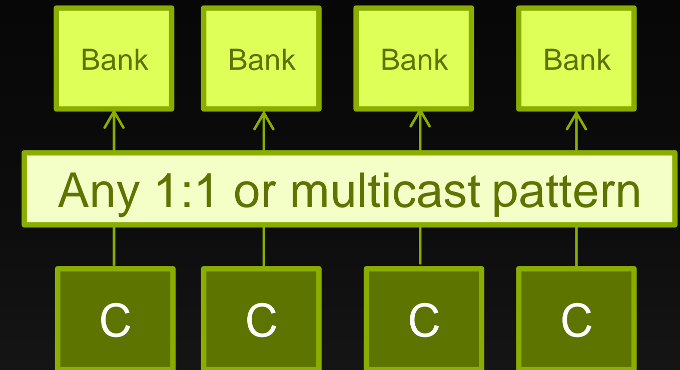
- Fast, on-chip memory
- Accessible by all threads within a thread block
  - Common allocation for entire thread block
- Variety of uses:
  - Software managed cache (e.g., tiled DGEMM)
  - Global memory coalescing (e.g., transpose)
  - Communication within a thread block (e.g., FFT, reductions)
- Limited Resource
  - Use of shared memory affects occupancy

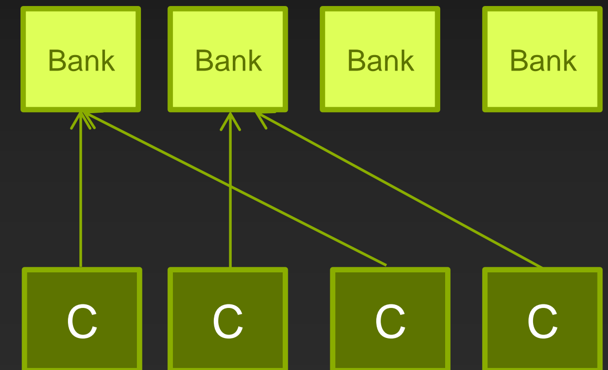


© NVIDIA 2013

Shared Memory Organization

- Organized in 32 independent banks
- Optimal access: no two words from same bank
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize

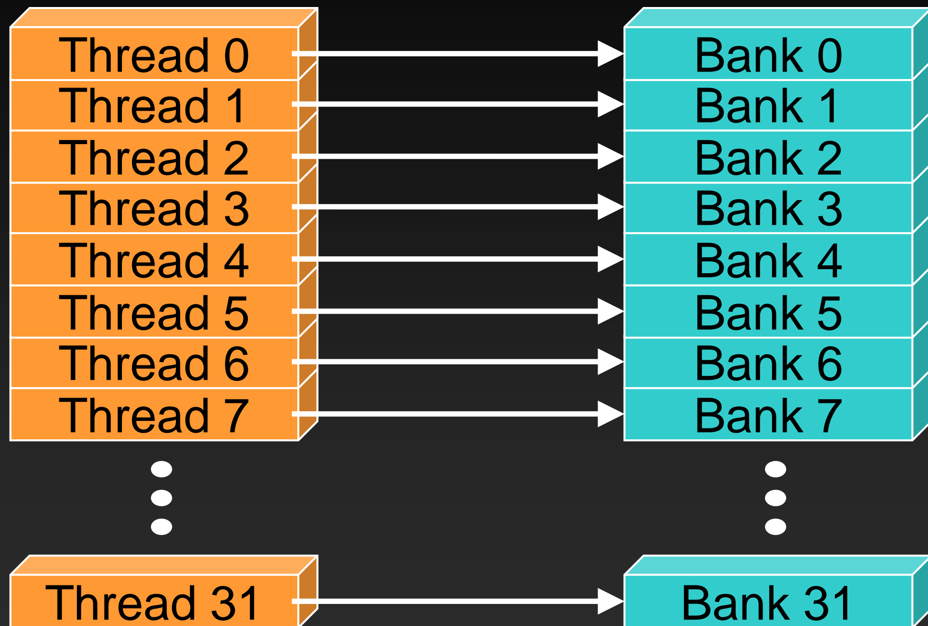




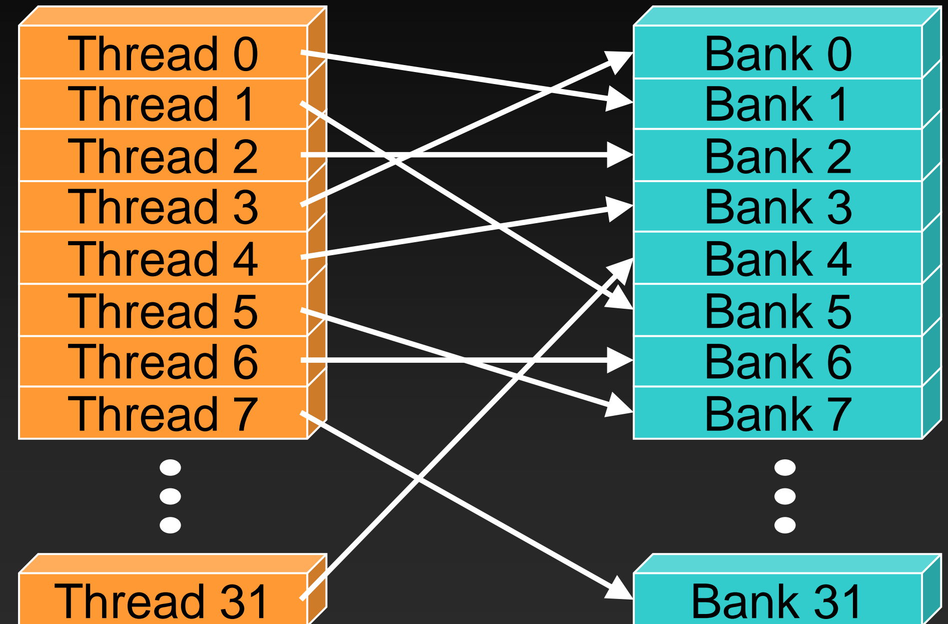
© NVIDIA 2013

Bank Addressing Examples

- No Bank Conflicts



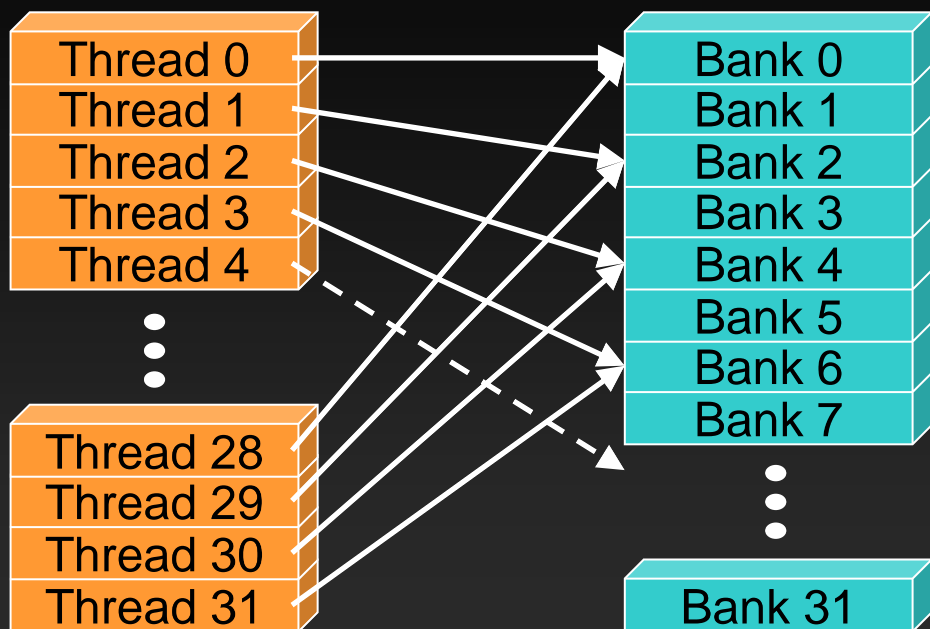
- No Bank Conflicts



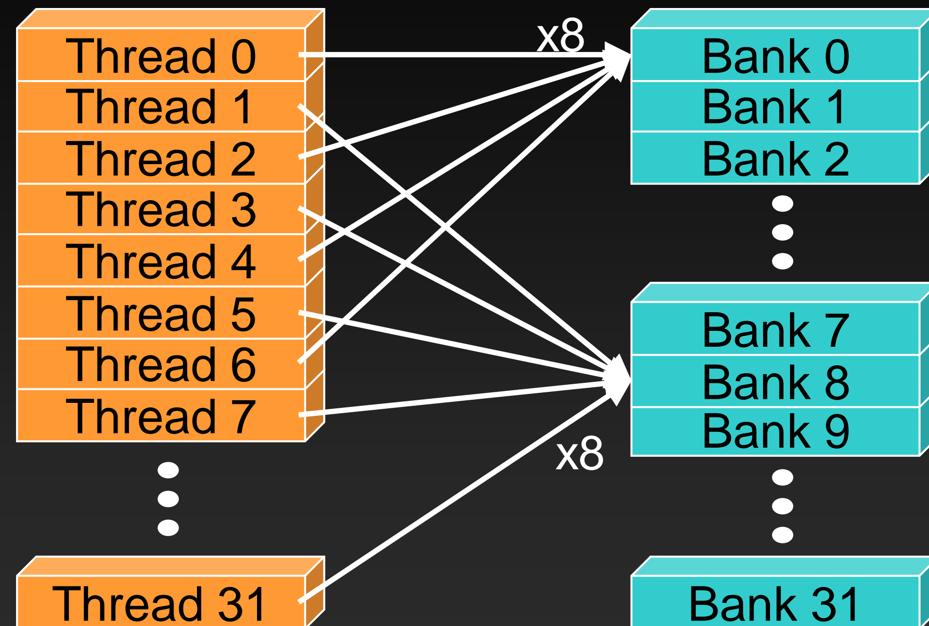
© NVIDIA 2013

Bank Addressing Examples

- 2-way Bank Conflicts



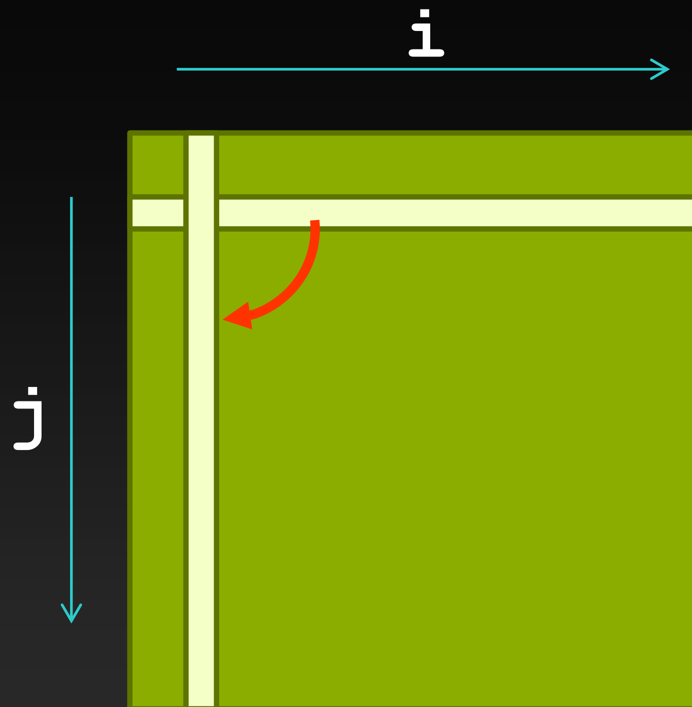
- 8-way Bank Conflicts



© NVIDIA 2013

Motivating Example: Matrix Transpose

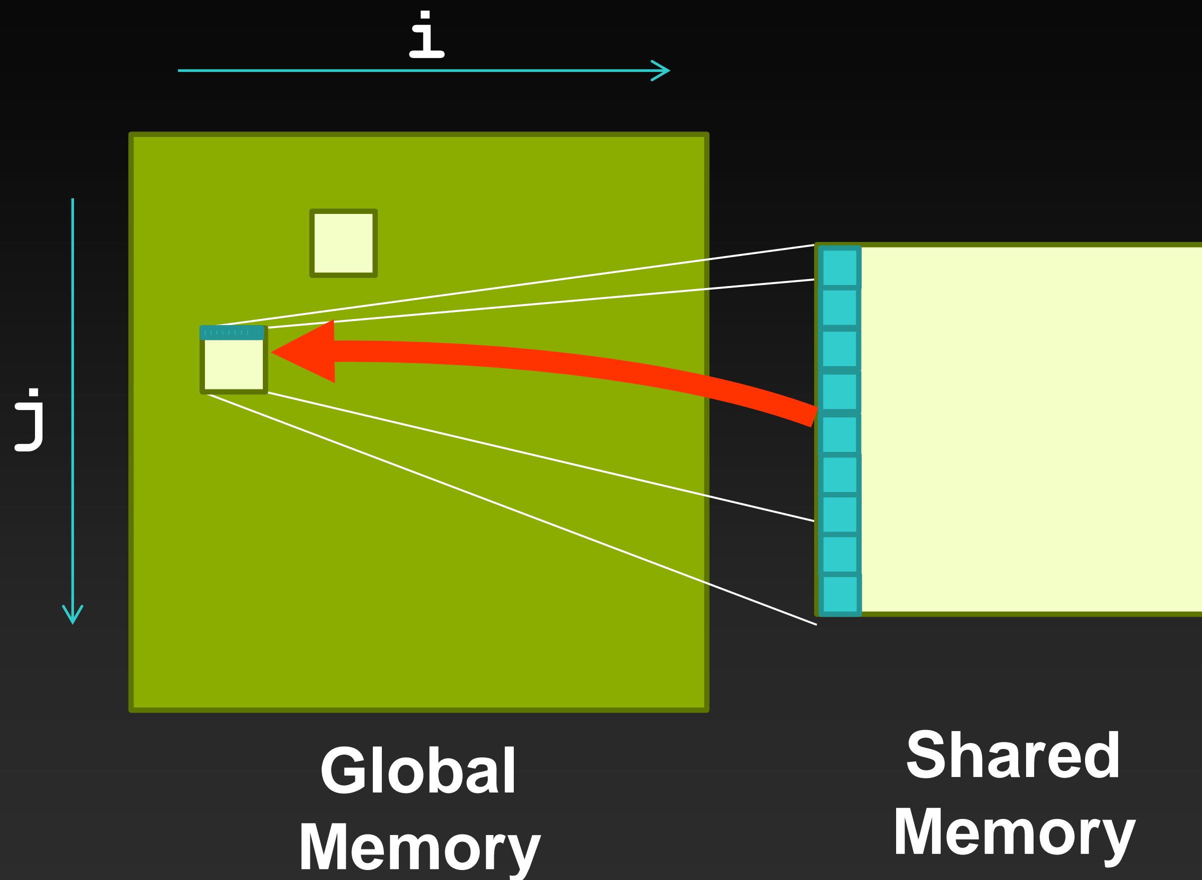
```
_global__ void gpuTranspose_kernel(int rows,
int cols, float *in, float *out)
{
  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  out[i * rows + j] = in[j * cols + i];
}
```



- Either write or read is strided in gmem and uncoalesced
- Solution: tile in shared memory

© NVIDIA 2013

Transposing with Shared Memory

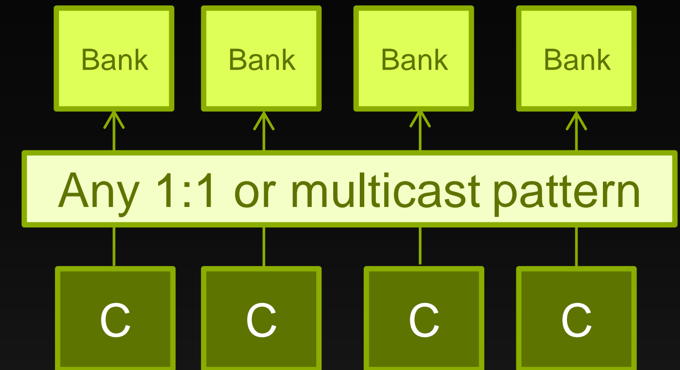


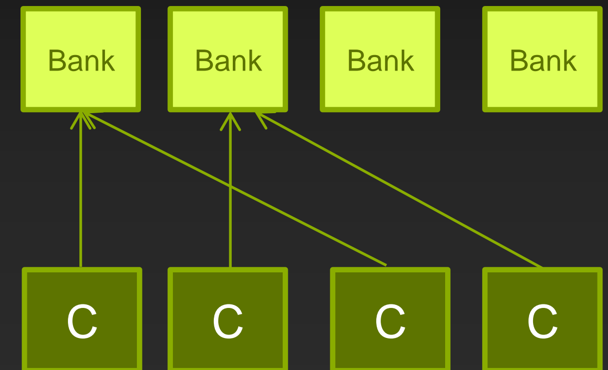
1. Read block_ij into shared memory
   - Reads are coalesced
2. Transpose shared memory indices
3. Write transposed block to global memory
   - Writes are coalesced

© NVIDIA 2013

Shared Memory Organization

- Organized in 32 independent banks
  - Note: same as warp size. Not a coincidence.
- Every 32byte word is in the next bank, modulo 32.
- Optimal access: no two words from same bank
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize
  - Called bank conflict, causes instruction replay



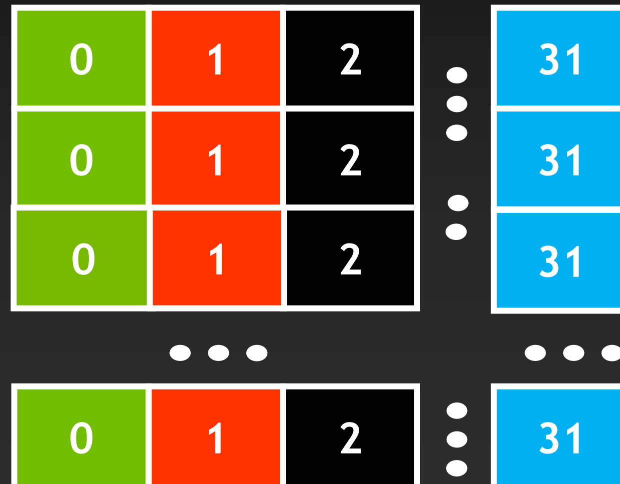


© NVIDIA 2013

Shared Memory: Avoiding Bank Conflicts

- Example: 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)

Bank 0
Bank 1
…
Bank 31

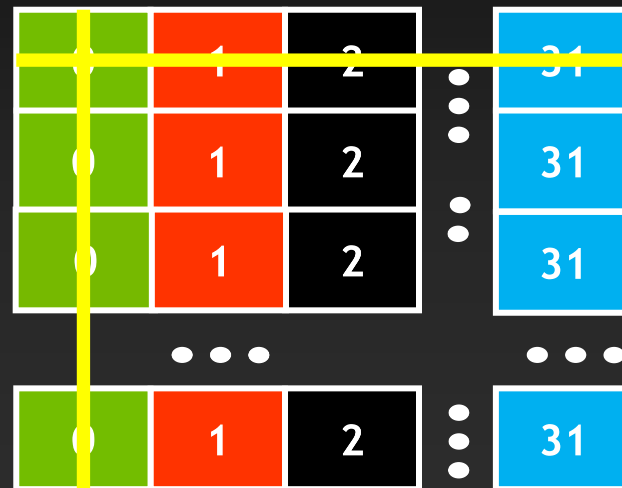


© NVIDIA 2013

Shared Memory: Avoiding Bank Conflicts

- Example: 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)

Bank 0
Bank 1
…
Bank 31



Accesses along row produces 0 bank conflicts

Accesses along column produces 32 bank conflicts (replays)

© NVIDIA 2013

Shared Memory: Avoiding Bank Conflicts

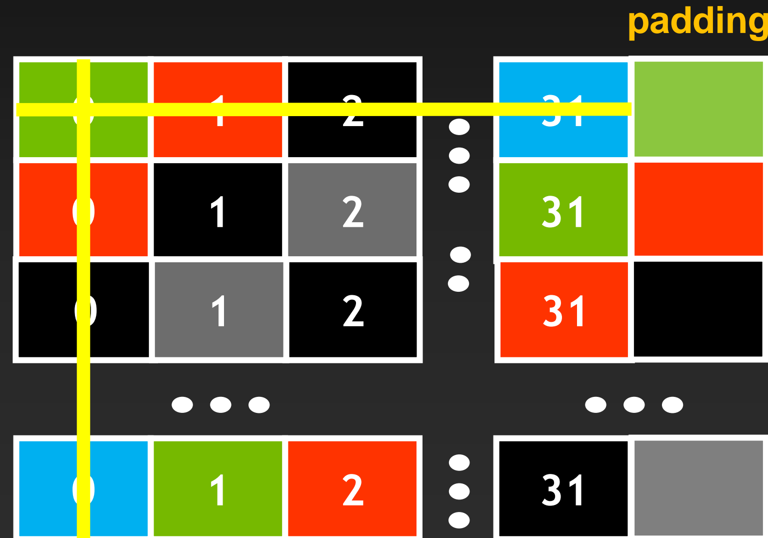
- **Add a column for padding:**
  - **32x33 SMEM array**
- **Warp accesses a column:**
  - **32 different banks, no bank conflicts**

Bank 0

Bank 1

…

Bank 31



Accesses along row produces no bank conflicts

Accesses along column produces no bank conflicts

© NVIDIA 2013

Shared Memory/L1 Sizing

- Shared memory and L1 use the same 64KB physical memory
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)
- Large SMEM can improve performance when:
  - Occupancy is limited by SMEM

© NVIDIA 2013

Final Notes on Shared Memory

- Fast: high bandwidth, low latency
- Useful as user managed cache for coalescing, caching, and communication within a thread block
- Shared memory size / L1 cache size is API-configurable
  - 16k L1 / 48k Shared (default on both Fermi and Kepler)
  - 48k L1 / 16k Shared
  - 32k L1 / 32k Shared (Kepler only).
- Be careful of:
  - Overuse: Excessive allocation can hurt occupancy
  - Access pattern: Lots of bank conflicts can hurt performance

© NVIDIA 2013

OPTIMIZE

Kernel Optimizations: ***Instruction Throughput / Control Flow***

© NVIDIA 2013

Exposing Sufficient Parallelism

- What SMX ultimately needs:
  - Sufficient number of independent instructions
  - Kepler GK110 is “wider” than Fermi or GK104; needs more parallelism
- Two ways to increase parallelism:
  - More independent instructions (ILP) within a thread (warp)
  - More concurrent threads (warps)

© NVIDIA 2013

Independent Instructions: ILP vs. TLP

- SMX can leverage available Instruction-Level Parallelism more or less interchangeably with Thread-Level Parallelism

- Sometimes easier to increase ILP than to increase TLP
  - E.g., # of threads may be limited by algorithm or by HW resource limits
  - But if each thread has some degree of independent operations to do, Kepler SMX can leverage that. (E.g., a small loop that is unrolled.)

- In fact, some degree of ILP is actually *required* to approach theoretical max Instructions Per Clock (IPC)

© NVIDIA 2013

Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a *single warp* take different paths
    - `if-else, ...`
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance

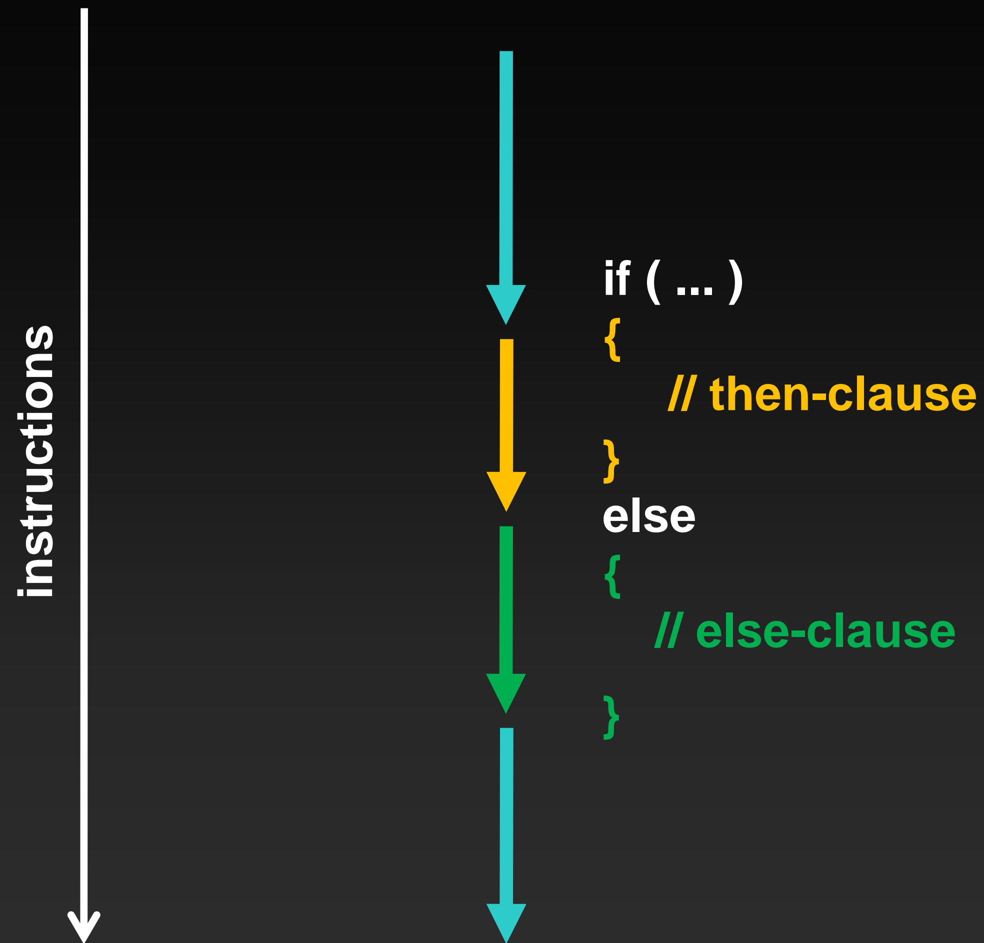
© NVIDIA 2013

Control Flow

- Avoid diverging within a warp
  - Note: *some* divergence is not necessarily a problem, but large amounts impacts execution efficiency
- Example with divergence:
  - `if (threadIdx.x > 2) {...} else {...}`
  - Branch granularity < warp size
- Example without divergence:
  - `if (threadIdx.x / warpSize > 2) {...} else {...}`
  - Branch granularity is a whole multiple of warp size

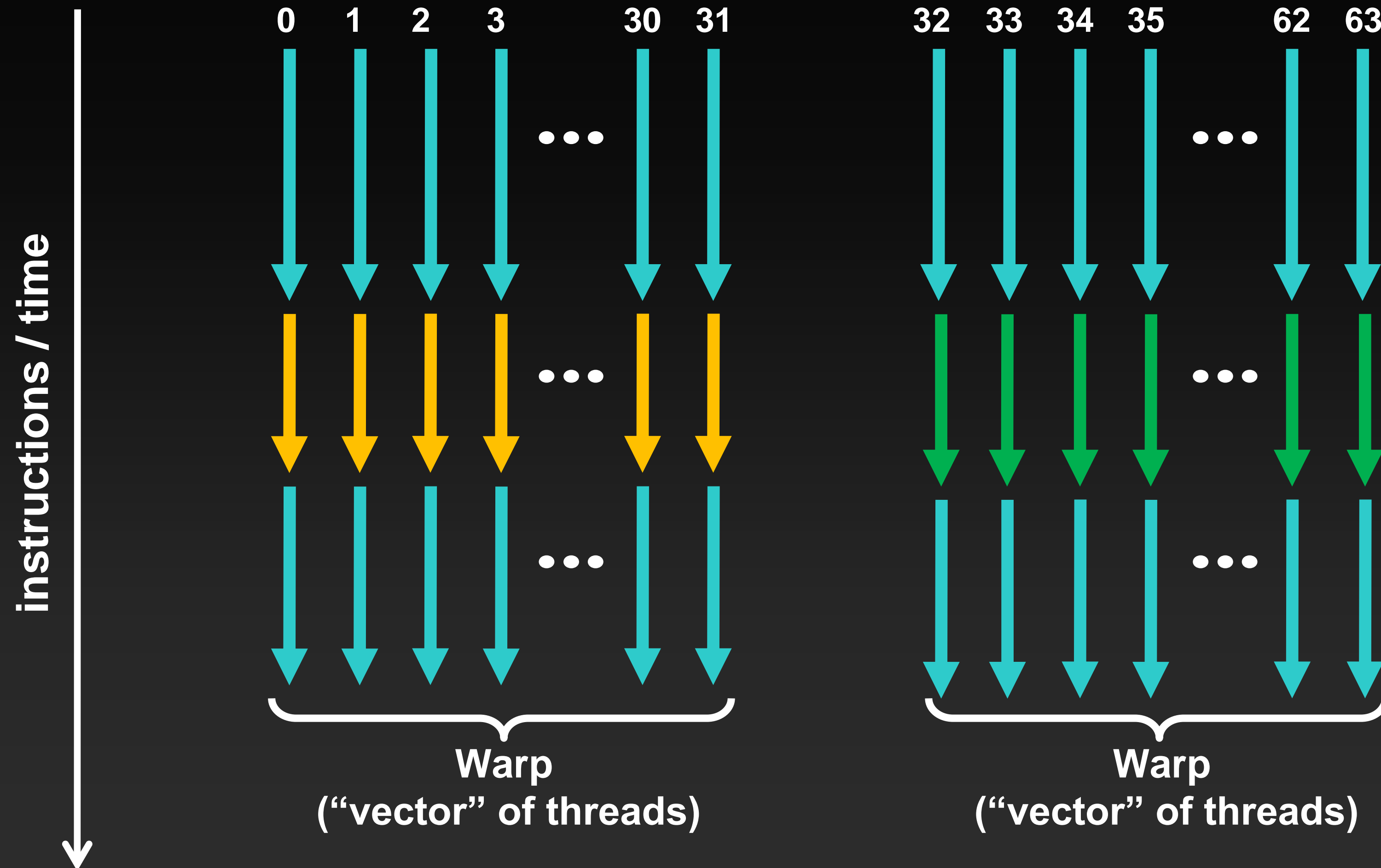
© NVIDIA 2013

Control Flow



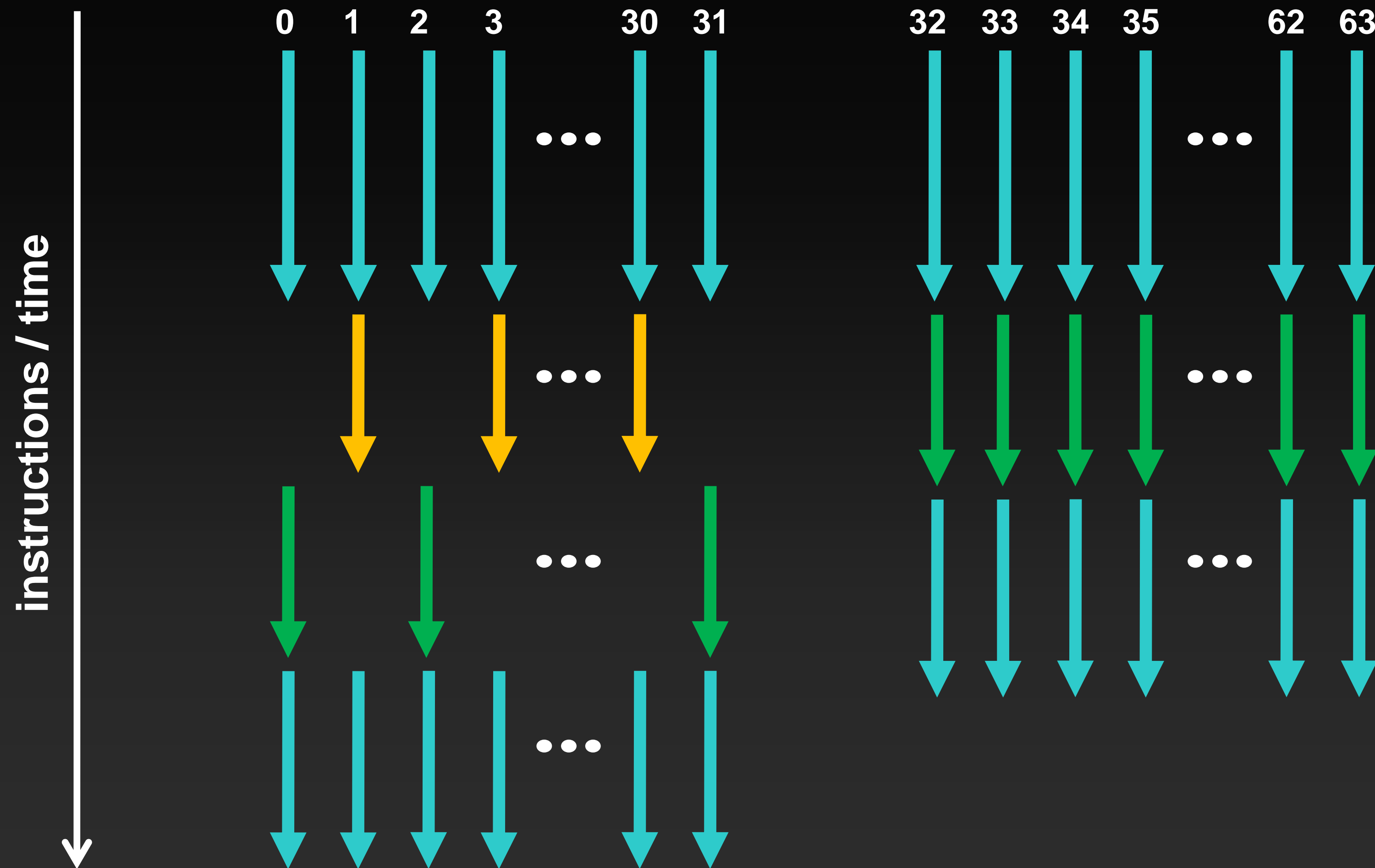
© NVIDIA 2013

Execution within warps is coherent



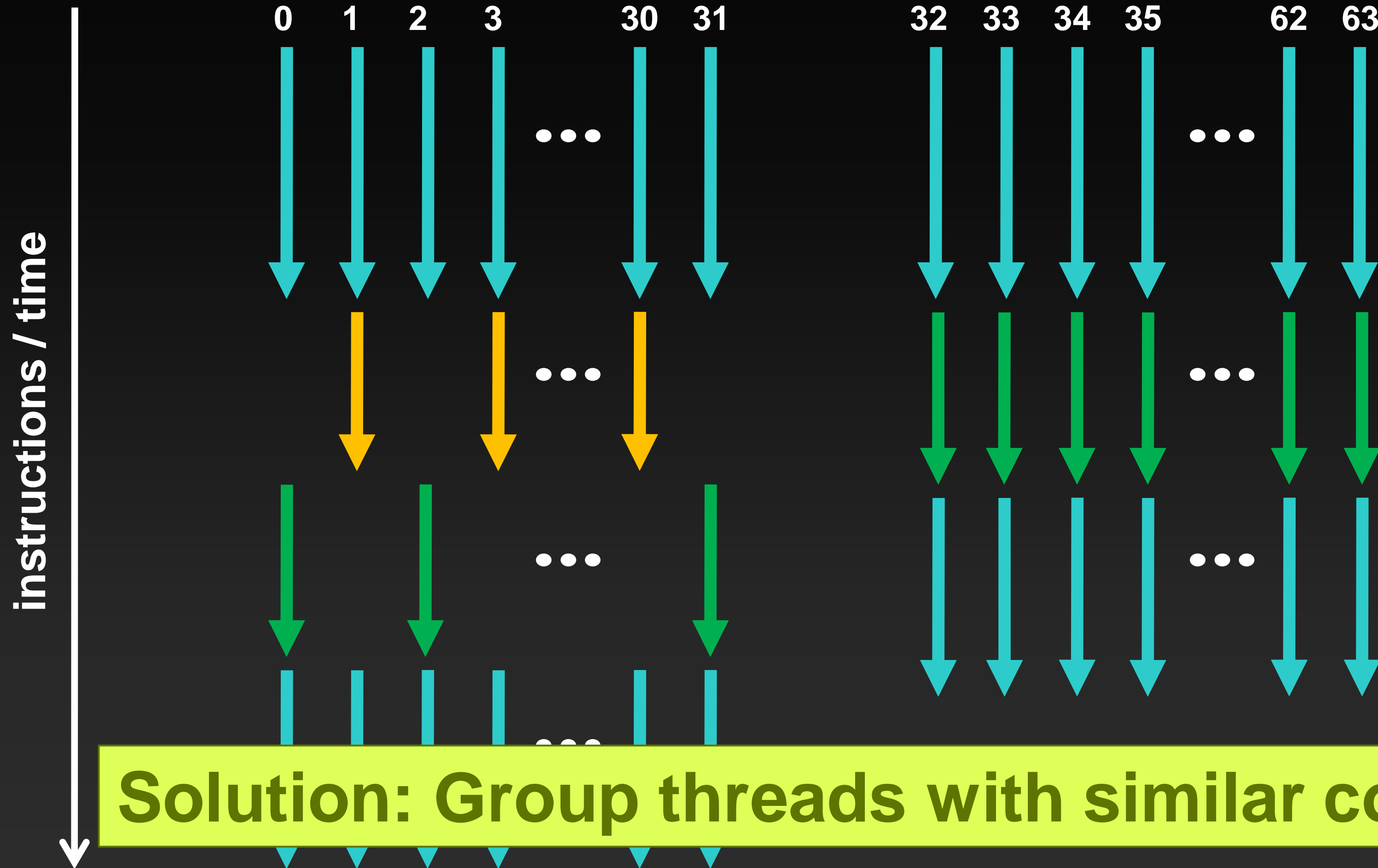
© NVIDIA 2013

Execution diverges within a warp



© NVIDIA 2013

Execution diverges within a warp



Solution: Group threads with similar control flow

© NVIDIA 2013

Runtime Math Library and Intrinsics

- Two types of runtime math library functions
  - __func(): many map directly to hardware ISA
    - Fast but lower accuracy (see CUDA Programming Guide for full details)
    - Examples: __sinf(x), __expf(x), __powf(x, y)
  - func(): compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x, y)
- A number of additional intrinsics:
  - __sincosf(), __frcp_rz(), ...
  - Explicit IEEE rounding modes (rz,rn,ru,rd)

© NVIDIA 2013

OPTIMIZE

Optimizing CPU-GPU Interaction: *Maximizing PCIe Throughput*

© NVIDIA 2013

Maximizing PCIe Throughput

- Use transfers that are of reasonable size (a few MB, at least)
- Use pinned system memory
- Overlap memcopies with useful computation

© NVIDIA 2013

Pinned (non-pageable) memory

- Pinned memory enables:
  - faster PCIe copies
  - memcopies asynchronous with CPU
  - memcopies asynchronous with GPU
- Usage
  - `cudaHostAlloc` / `cudaFreeHost`
    - instead of malloc / free
  - `cudaHostRegister` / `cudaHostUnregister`
    - pin regular memory after allocation
- Implication:
  - pinned memory is essentially removed from host virtual memory

© NVIDIA 2013

Asynchronicity in CUDA

- Default:
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- Streams and async functions provide additional asynchronicity:
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute kernels and memcopies
- *Stream*: sequence of ops that execute in issue-order on GPU
  - Operations from different streams may be interleaved
  - Kernels and memcopies from different streams can be overlapped

© NVIDIA 2013

OPTIMIZE

Optimizing CPU-GPU Interaction: *Overlapping Kernel Execution with Memory Copies*

© NVIDIA 2013

Overlap kernel and memory copy

- Requirements:
  - D2H or H2D memcopy from pinned memory
  - Kernel and memcopy in different, non-0 streams
- Code:

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

} potentially overlapped

© NVIDIA 2013

Call Sequencing for Optimal Overlap

- CUDA calls are dispatched in the sequence they were issued
- Kepler can concurrently execute:
  - Up to 32 kernels
  - Up to 2 memcopies, as long as they are in different directions (D2H, H2D)
- A call is dispatched if both are true:
  - Resources are available
  - Preceding calls in the same stream have completed
- Scheduling:
  - Kernels are executed in the order in which they were issued
  - Thread blocks for a given kernel are scheduled if all thread blocks for preceding kernels have been scheduled and SM resources still available

© NVIDIA 2013

Hyper-Q Enables Efficient Scheduling

- Grid Management Unit selects most appropriate task from up to 32 hardware queues (CUDA streams)
- Improves scheduling of concurrently executed grids
- Particularly interesting for MPI applications when combined with CUDA MPS (though not limited to MPI applications)

© NVIDIA 2013

Stream Examples without Hyper-Q

K1,M1,K2,M2:

K1,K2,M1,M2:

K1,M1,M2:

K1,M2,M1:

K1,M2,M2:

K: Kernel
M: Memcopy
Integer: Stream ID

© NVIDIA 2013

Stream Examples with Hyper-Q

K1,M1,K2,M2:

K1,K2,M1,M2:

K1,M1,M2:

K1,M2,M1:

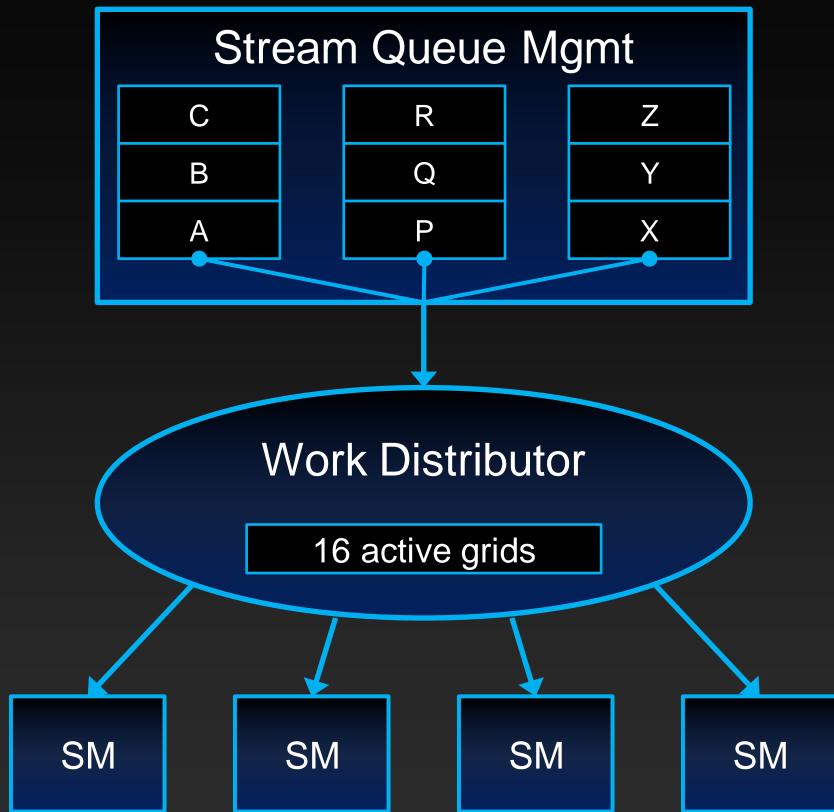
K1,M2,M2:

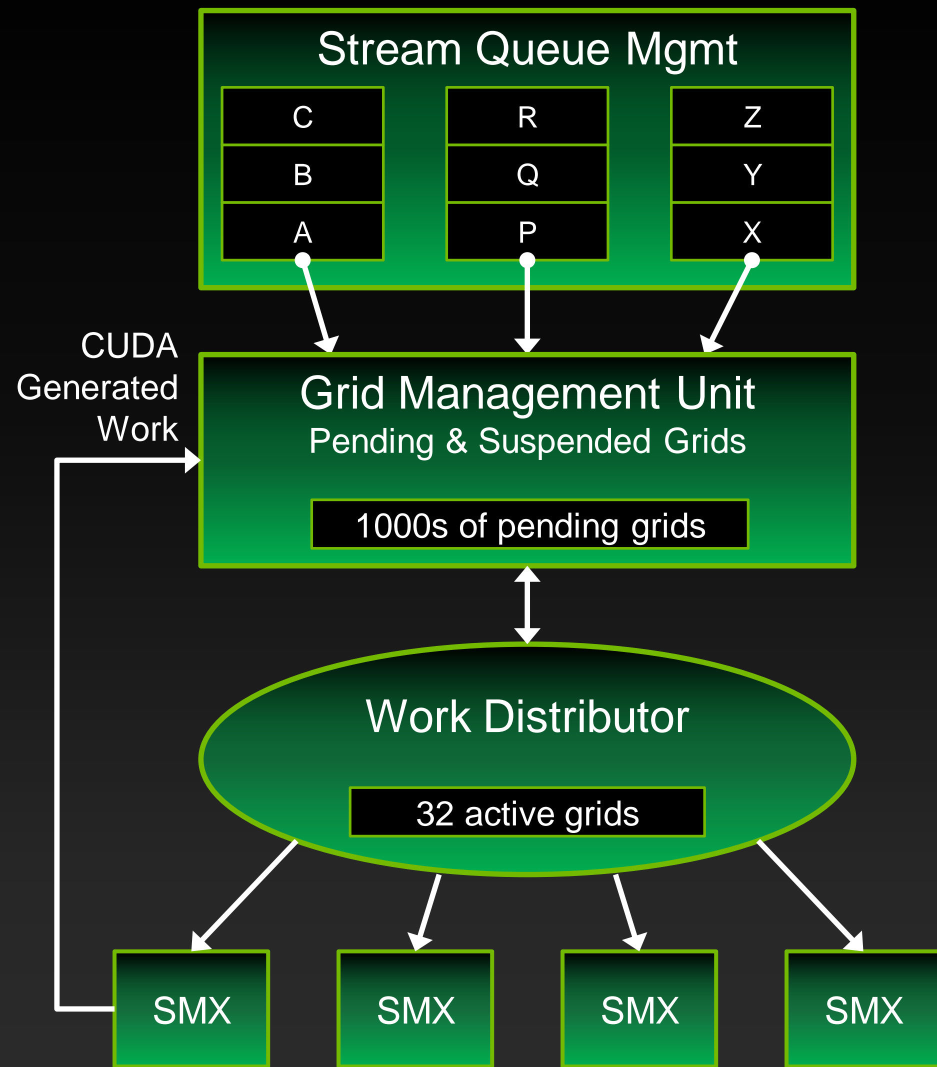
Time

K:	Kernel
M:	Memcopy
Integer:	Stream ID

© NVIDIA 2013

Grid Management

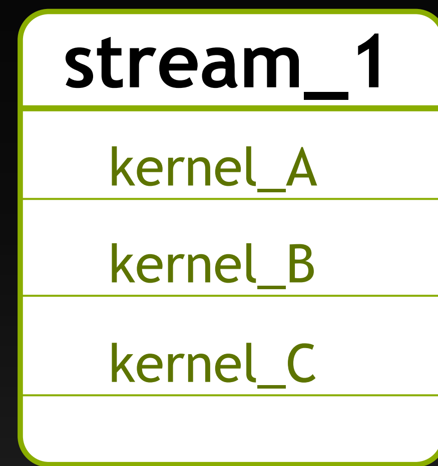


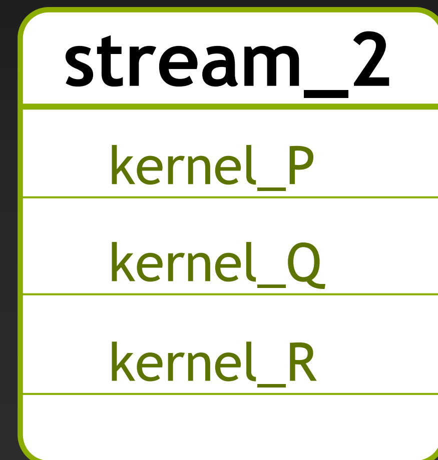


© NVIDIA 2013

Stream Dependencies Example

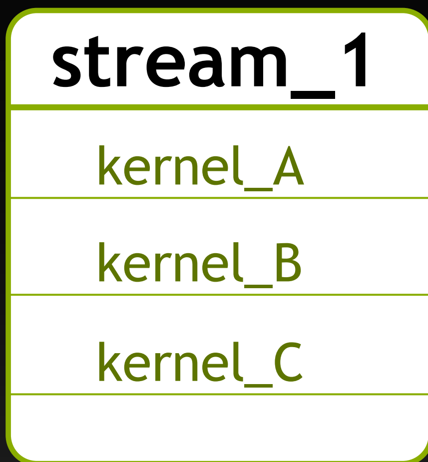
```
void foo(void)
{
    kernel_A<<<g,b,s, stream_1>>>();
    kernel_B<<<g,b,s, stream_1>>>();
    kernel_C<<<g,b,s, stream_1>>>();
}

void bar(void)
{
    kernel_P<<<g,b,s, stream_2>>>();
    kernel_Q<<<g,b,s, stream_2>>>();
    kernel_R<<<g,b,s, stream_2>>>();
}
```

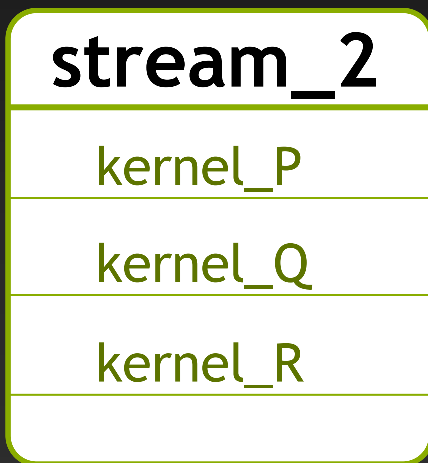


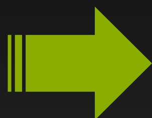


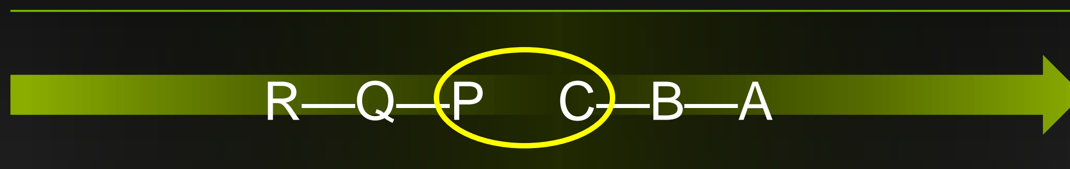
© NVIDIA 2013

Stream Dependencies without Hyper-Q





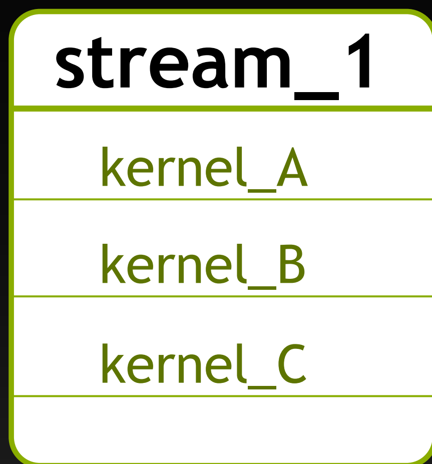


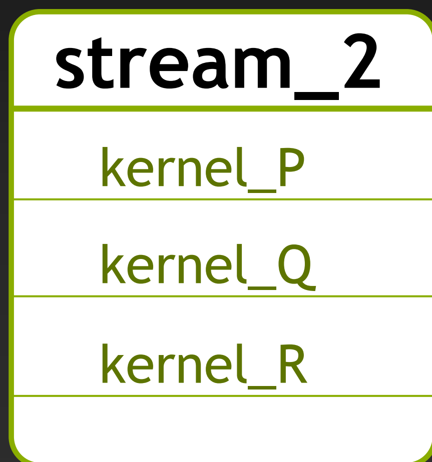


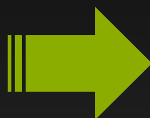
Hardware Work Queue

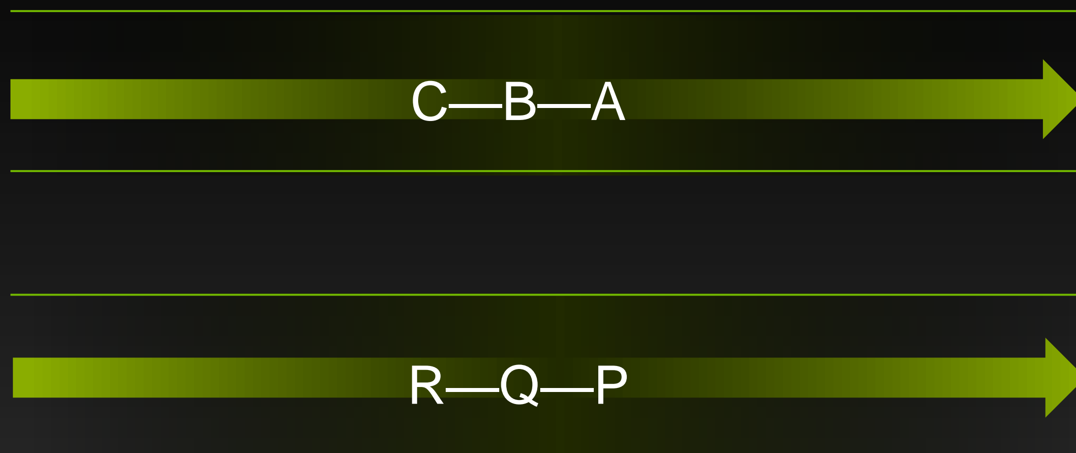
© NVIDIA 2013

Stream Dependencies with Hyper-Q







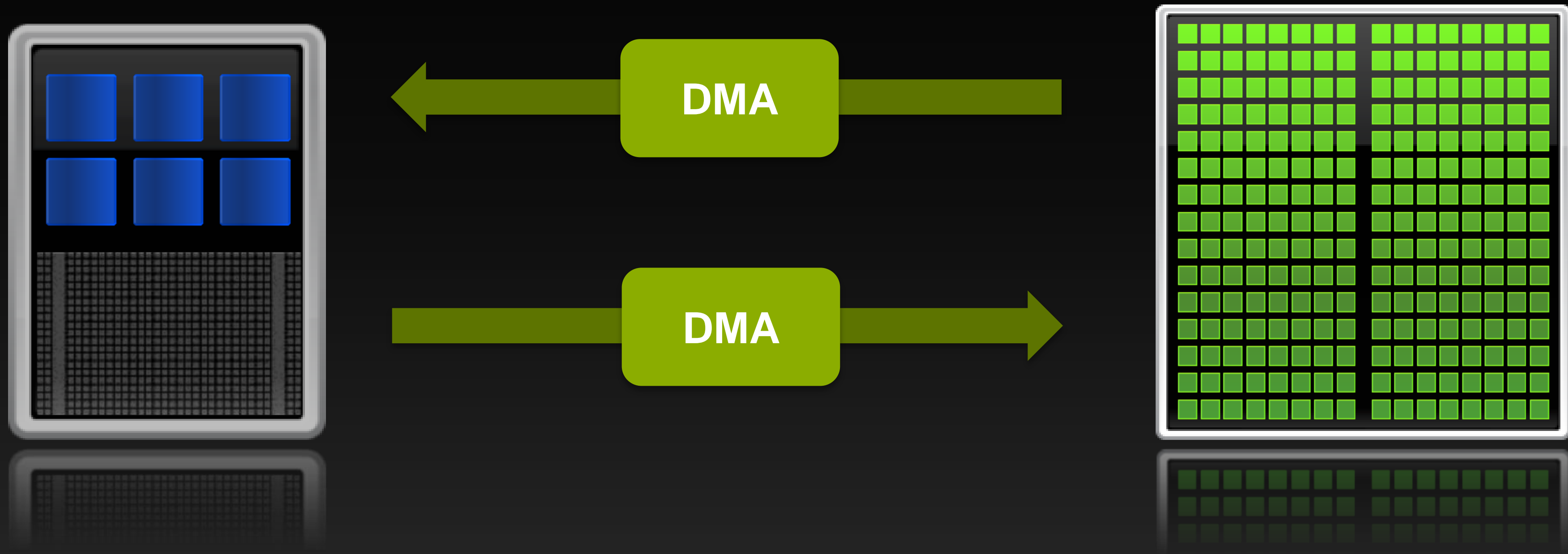


Multiple Hardware Work Queues

- **Hyper-Q allows 32-way concurrency**
- **Avoids inter-stream dependencies**

© NVIDIA 2013

Hyper-Q Example: Building a Pipeline



- Heterogeneous system: overlap work and data movement
- Kepler + CUDA 5: Hyper-Q and CPU Callbacks

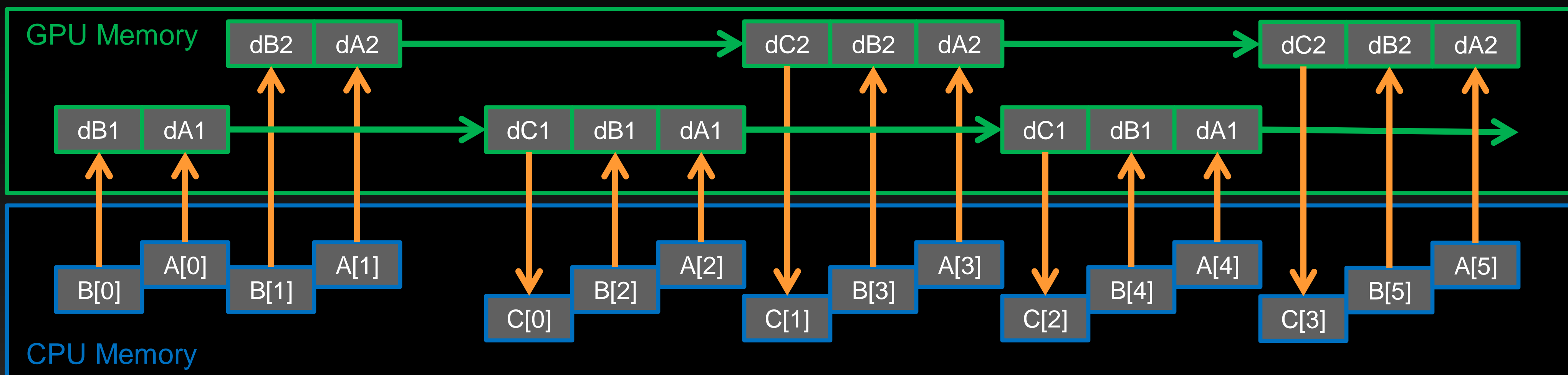
© NVIDIA 2013

Tick-Tock Matrix Multiply

```
cudaMemcpyAsync(devA1, A[tile0], N, stream1);
cudaMemcpyAsync(devB1, B[tile0], N, stream1);
DGEMM<<<g,b,s, stream1>>>(devA1, devB1, devC1);

cudaMemcpyAsync(devA2, A[tile1], N, stream2);
cudaMemcpyAsync(devB2, B[tile1], N, stream2);
DGEMM<<<g,b,s, stream2>>>(devTileA, devTileB, devC1);

cudaMemcpyAsync(C[tile0], devC,  N, D2H, stream1);
cudaMemcpyAsync(devA1, A[tile2], N, H2D, stream1)
cudaMemcpyAsync(devB1, B[tile2], N, D2H, stream1)
DGEMM<<<g,b,s, stream1>>>(devA1, devB1, devC1);

cudaMemcpyAsync(C[tile1], devC,  N, D2H, stream1);
cudaMemcpyAsync(devA1, A[tile4], N, H2D, stream1);
cudaMemcpyAsync(devB1, B[tile4], N, D2H, stream1);
DGEMM<<<g,b,s, stream1>>>(devA1, devB1, devC1);
```

© NVIDIA 2013

Tick-Tock Matrix Multiply



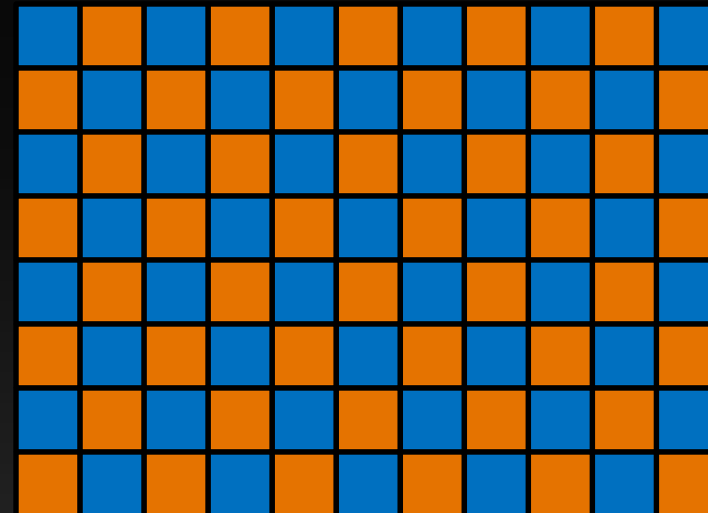
stream 1	memcpy	DGEMM dC_1 =dA_1 x dB_1	memcpy	DGEMM C_1 =A_1 x B_1	memcpy	DGEMM C_1 =A_1 x B_1
stream 2		memcpy	DGEMM dC_2 =dA_2 x dB_2	memcpy	DGEMM C_2 =A_2 x B_2	memcpy

© NVIDIA 2013

Just a Higher Level of Parallelism

- Problem is decomposed into parallel "workers".
- At any given time
  - 1 worker is using compute resources
  - 1 worker is using copy transfers
- Importantly:
  - The PCI-E link is kept saturated with useful work.
  - For DGEMM, compute is also saturated.
- Arch specific balancing
  - Depends on CPU and GPU characteristics.

Result Matrix:



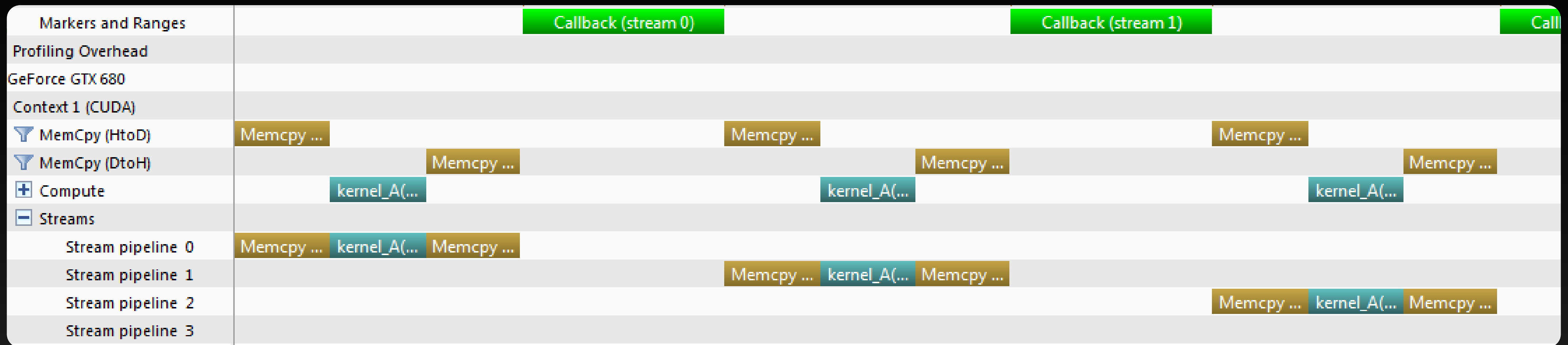
© NVIDIA 2013

Pipeline Code

```
for (unsigned int i = 0 ; i < nIterations ; ++i)
{
    // Copy data from host to device
    cudaMemcpyAsync(d_data, h_data, cpybytes, cudaMemcpyHostToDevice,
                    *r_streams.active());

    // Launch device kernel A
    kernel_A<<<gdim, bdim, 0, *r_streams.active()>>>();

    // Copy data from device to host
    cudaMemcpyAsync(h_data, d_data, cpybytes, cudaMemcpyDeviceToHost,
                    *r_streams.active());

    // Launch host post-process
    cudaStreamAddCallback(*r_streams.active(), cpu_callback,
                          r_streamids.active(), 0);

    // Rotate streams
    r_streams.rotate(); r_streamids.rotate();
}
```

© NVIDIA 2013

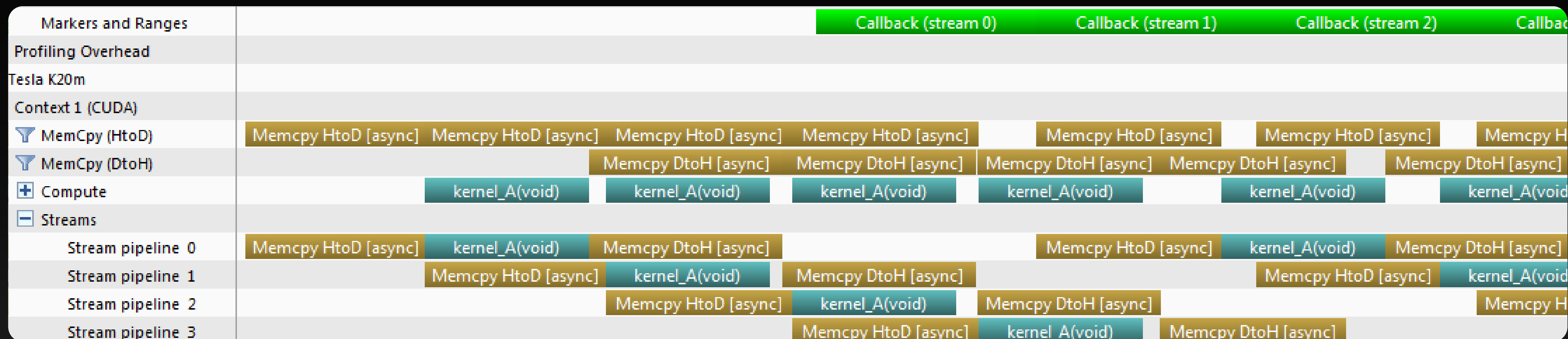
Pipeline Without Hyper-Q



- False dependencies prevent overlap
- Breadth-first launch gives overlap, requires more complex code

© NVIDIA 2013

Pipeline With Hyper-Q



- Full overlap of all engines
- Simple to program

© NVIDIA 2013

Hyper-Q also enables CUDA MPS

- No application modifications necessary
  - Start MPS daemon using `nvidia_cuda_mps_control -d`
  - CUDA driver detects daemon and routes GPU accesses through it
- Combines requests from several processes into one GPU context (shared virtual memory space, concurrent kernels possible, etc.)
- Allows for overlap of kernels with memcopies *without explicit use of streams*

© NVIDIA 2013

But Hyper-Q != CUDA MPS

- One process: No MPS required!
  - Automatically utilized
  - One or many host threads no problem
  - Just need multiple CUDA streams
  - Removes false dependencies among CUDA streams that reduce effective concurrency on earlier GPUs
- Multi-process: Use CUDA MPS
  - Leverages task-level parallelism across processes (e.g., MPI ranks)
  - MPI is not required for MPS – it's just the common case for HPC

© NVIDIA 2013

Deploy

- We've removed (or reduced) some bottleneck
- Our app is now faster while remaining fully functional*
- Let's take advantage of that!

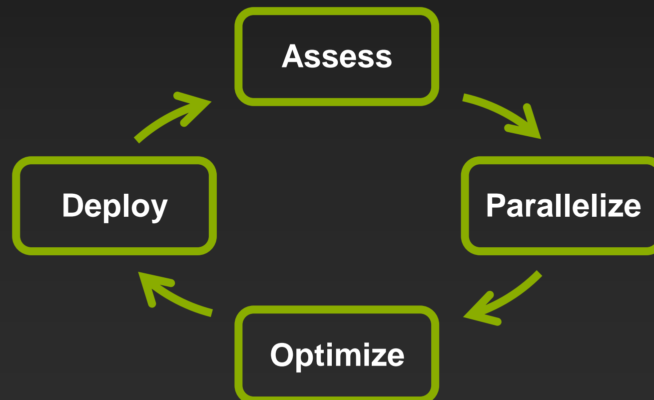
- *Don't forget to check correctness at every step

© NVIDIA 2013

GPU Optimization Fundamentals

Recap:

- Develop systematically with APOD
- Expose sufficient parallelism
- Utilize parallel processing resources efficiently



© NVIDIA 2013

Online Resources

www.udacity.com

devtalk.nvidia.com

developer.nvidia.com

docs.nvidia.com

www.stackoverflow.com

© NVIDIA 2013