

Pascal Deep Dive

Peter Messmer, 2/28/2017



INTRODUCING TESLA P100

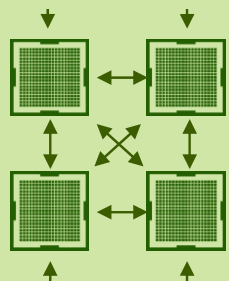
New GPU Architecture to Enable the World's Fastest Compute Node

Pascal Architecture



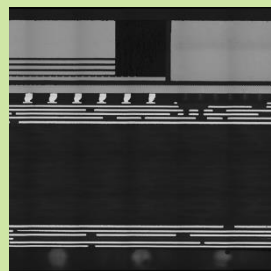
Highest Compute Performance

NVLink



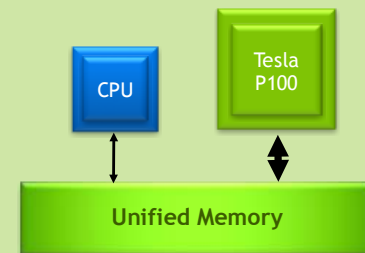
GPU Interconnect for Maximum Scalability

HBM2 Stacked Memory

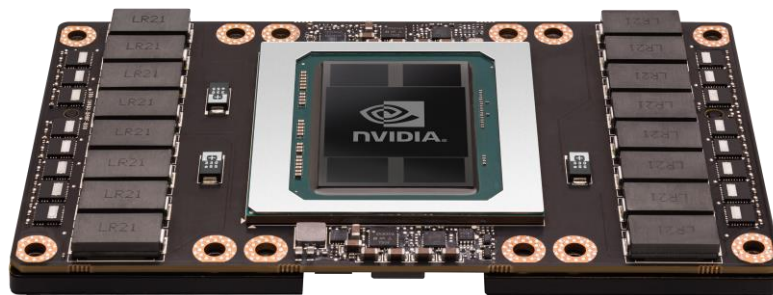


Unifying Compute & Memory in Single Package

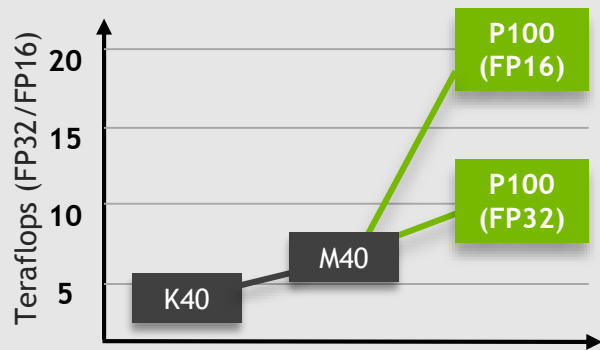
Page Migration Engine



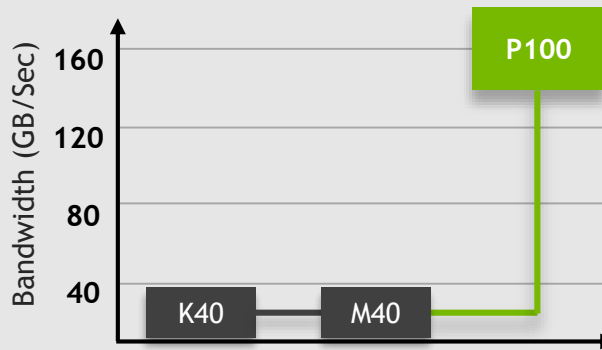
Simple Parallel Programming with 512 TB of Virtual Memory



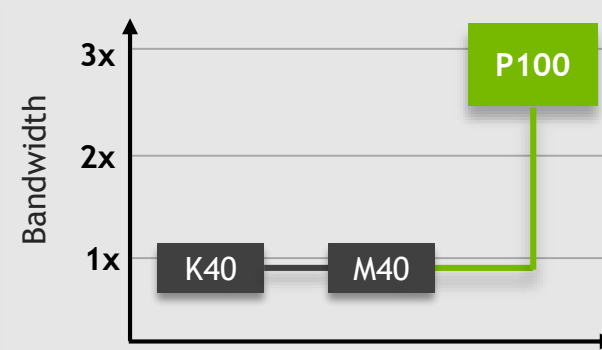
GIANT LEAPS IN EVERYTHING



3x Compute



5x GPU-GPU BW



3x GPU Mem BW

PASCAL ARCHITECTURE

Tesla P100 GPU: GP100

56 SMs

3584 CUDA Cores

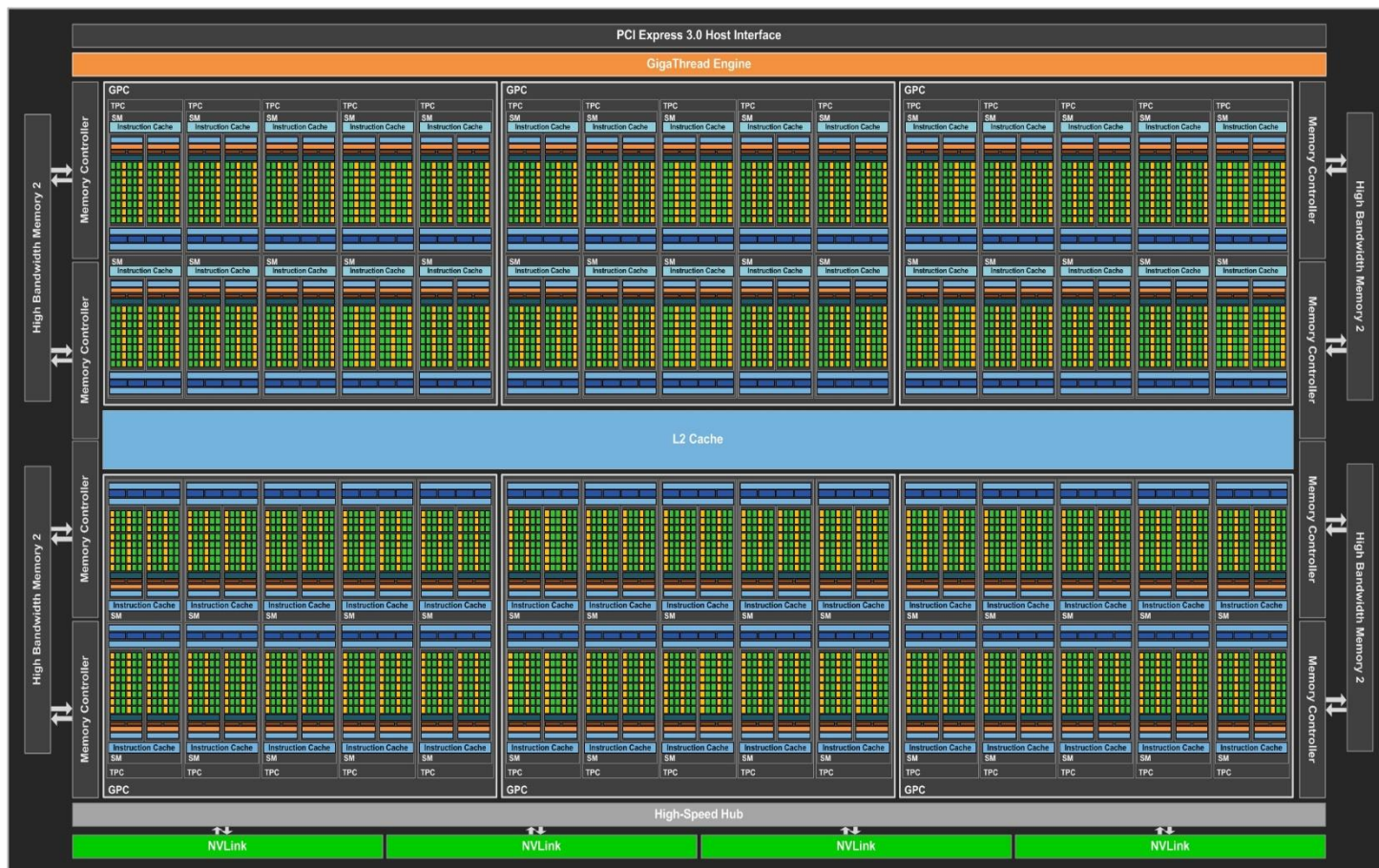
5.3 TF Double Precision

10.6 TF Single Precision

21.2 TF Half Precision

16 GB HBM2

720 GB/s Bandwidth



GPU Performance Comparison

	P100	M40	K40
Double Precision TFlop/s	5.3	0.2	1.4
Single Precision TFlop/s	10.6	7.0	4.3
Half Precision Tflop/s	21.2	NA	NA
Memory Bandwidth (GB/s)	720	288	288
Memory Size	16GB	12GB, 24GB	12GB

GP100 SM

GP100

CUDA Cores 64

Register File 256 KB

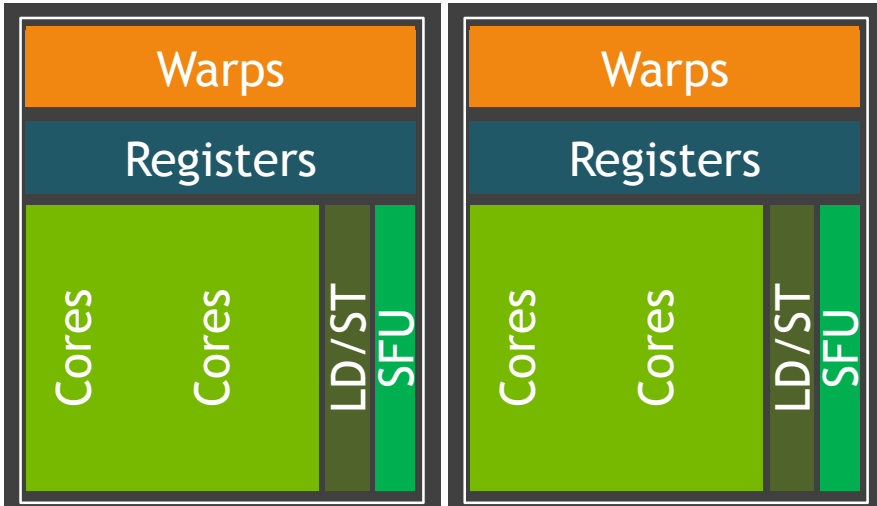
Shared Memory 64 KB

Active Threads 2048

Active Blocks 32



P100 SM



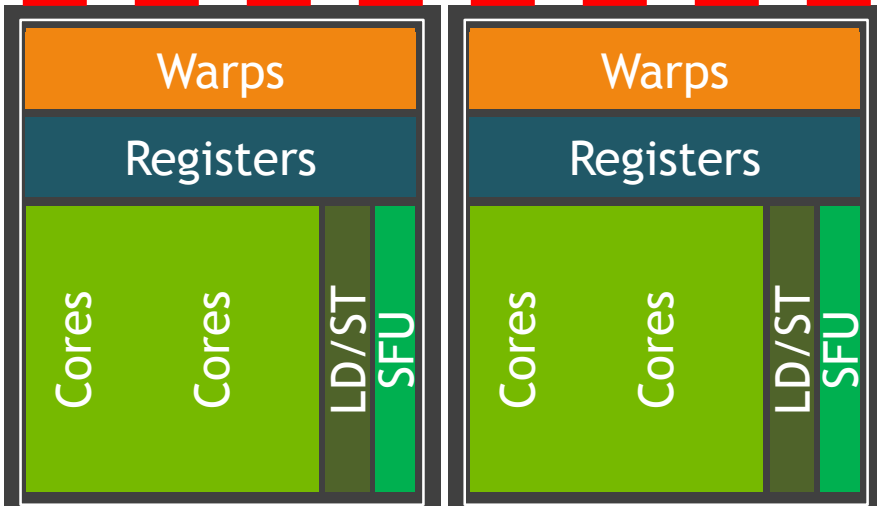
More resources per core

- 2x Registers
- 1.33x Shared Memory Capacity
- 2x Shared Memory Bandwidth
- 2x Warps

Maxwell SM





P100 SM



Higher Instruction Throughput

IEEE 754 Floating Point on GP100

3 sizes, 3 speeds, all fast

Feature	 Half precision	Single precision	Double precision
Layout	s5.10	s8.23	s11.52
Issue rate	pair every clock	1 every clock	1 every 2 clocks
Subnormal support	Yes	Yes	Yes
Atomic Addition	Yes	Yes	 Yes

Half-precision Floating Point (FP16)

16 bits



1 sign bit, 5 exponent bits, 10 fraction bits

2^{40} Dynamic range

Normalized values: 1024 values for each power of 2,
from 2^{-14} to 2^{15}

Subnormals at full speed: 1024 values from 2^{-24} to 2^{-15}

Special values

+/- Infinity, Not-a-number

USE CASES

Deep Learning Training

Radio Astronomy

Sensor Data

Image Processing

NVLink

NVLink

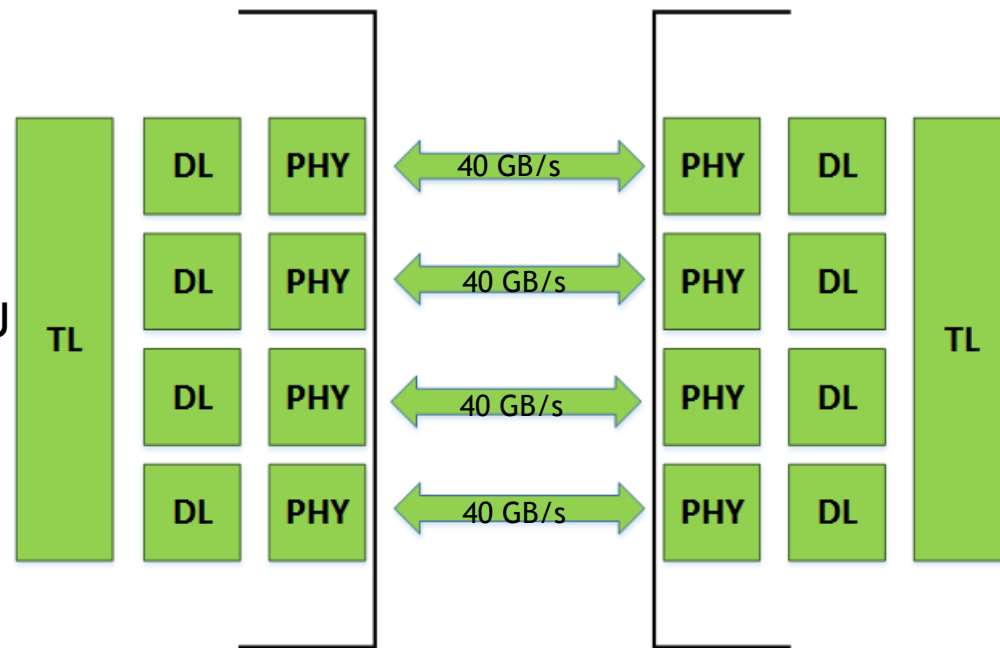
P100 supports 4 NVLinks

Up to 94% bandwidth efficiency

Supports read/writes/atomics to peer GPU

Supports read/write access to NVLink-enabled CPU

Links can be ganged for higher bandwidth



NVLink on Tesla P100

NVLINK - GPU CLUSTER

Two fully connected quads,
connected at corners

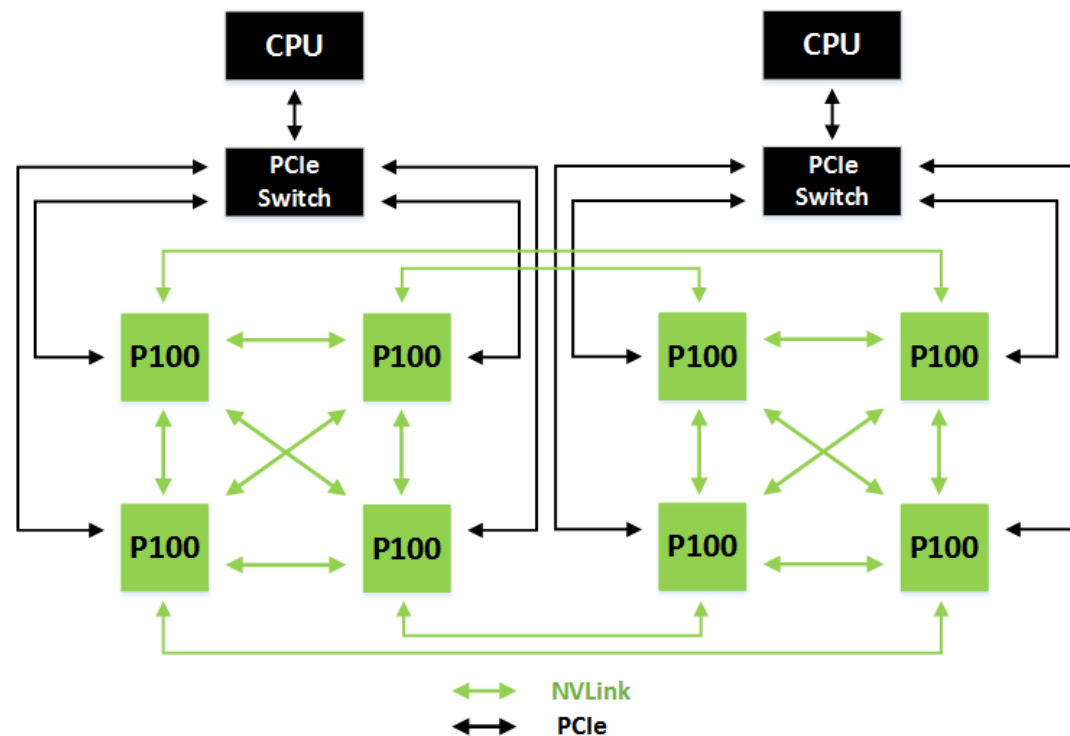
160GB/s per GPU bidirectional to Peers

Load/store access to Peer Memory

Full atomics to Peer GPUs

High speed copy engines for bulk data copy

PCIe to/from CPU



NVLINK to CPU

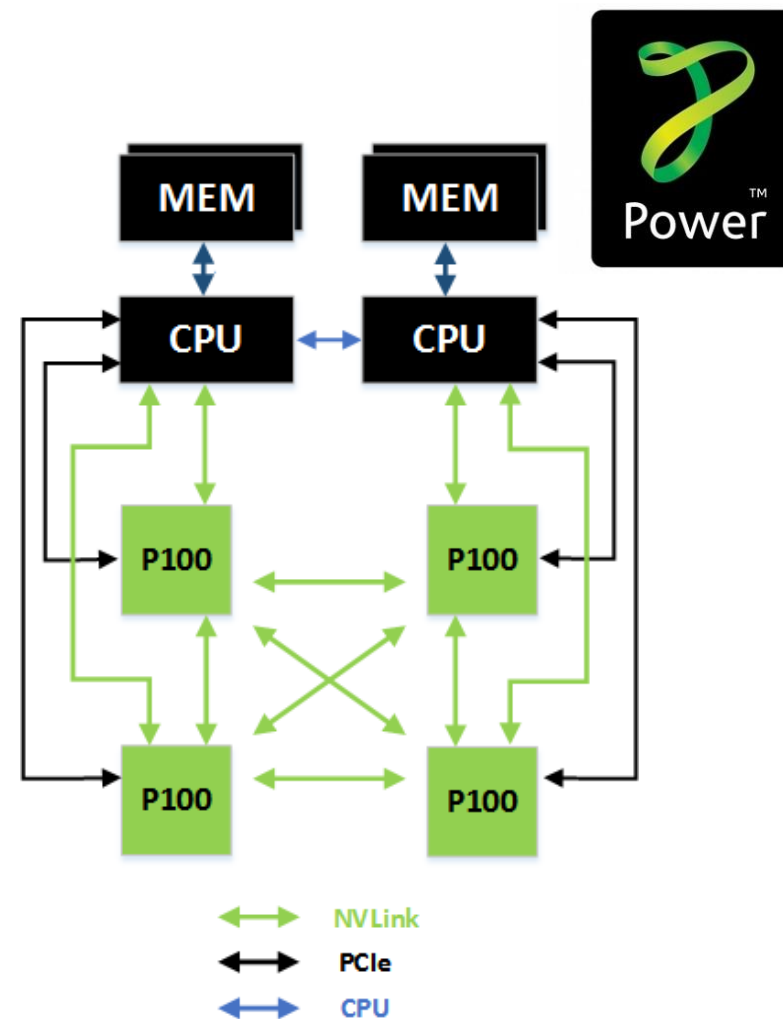
Fully connected quad

120 GB/s per GPU bidirectional for peer traffic

40 GB/s per GPU bidirectional to CPU

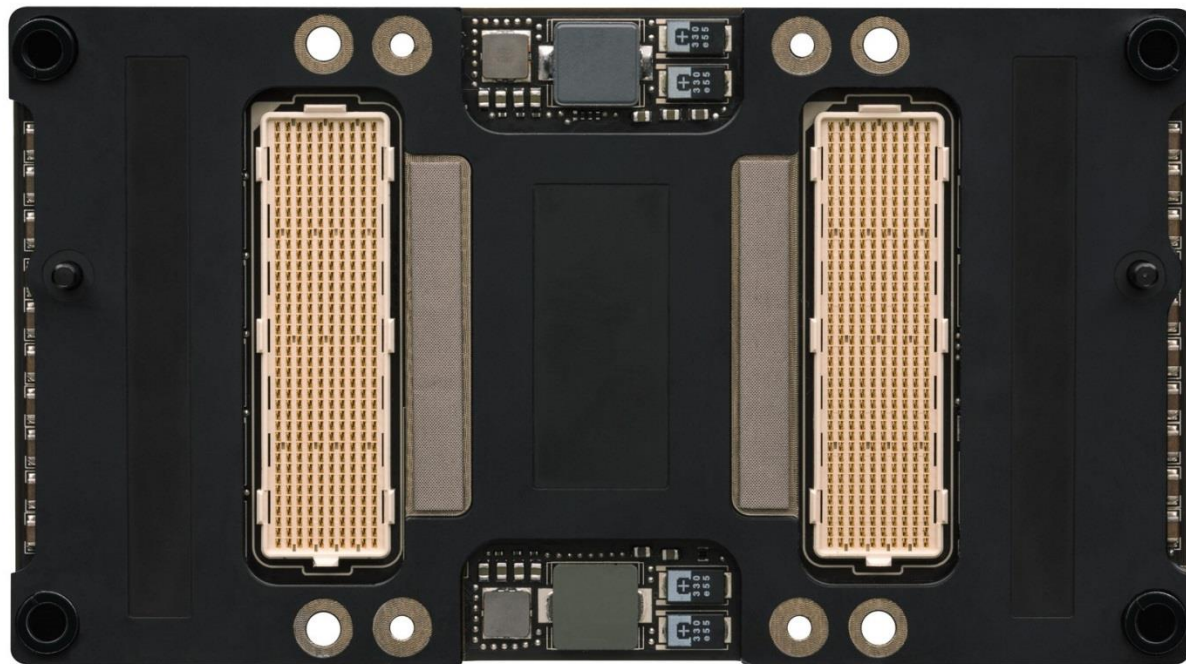
Direct Load/store access to CPU Memory

High Speed Copy Engines for bulk data movement



Tesla P100 Physical Connector

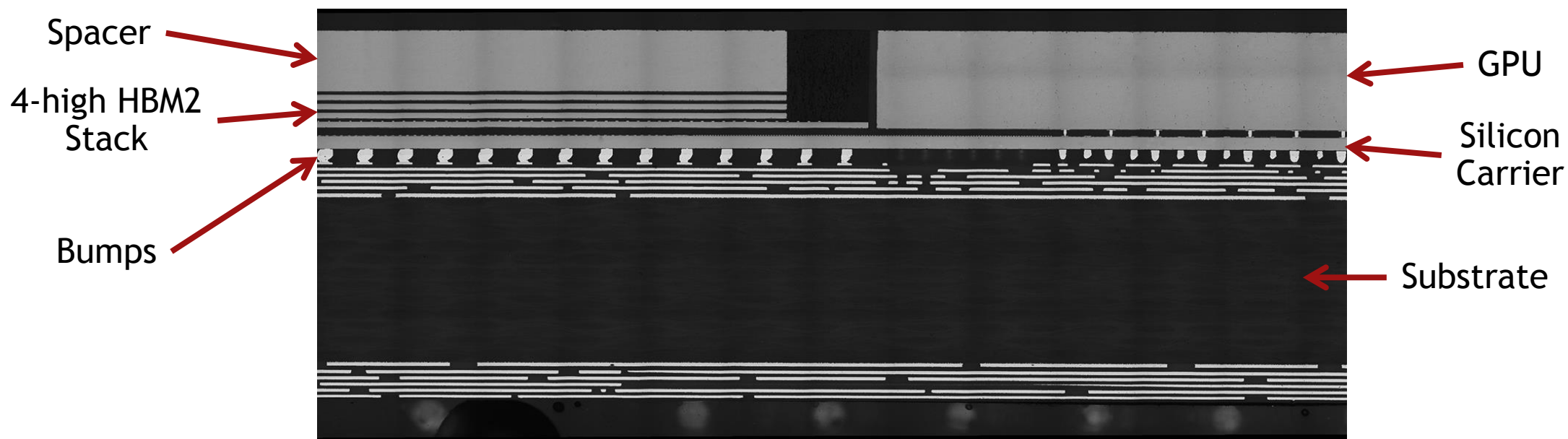
With NVLink



HBM2 STACKED MEMORY

HBM2 : 720GB/sec bandwidth

And ECC is free



UNIFIED MEMORY

Page Migration engine

Support Virtual Memory Demand Paging

49-bit Virtual Addresses

Sufficient to cover 48-bit CPU address + all GPU memory

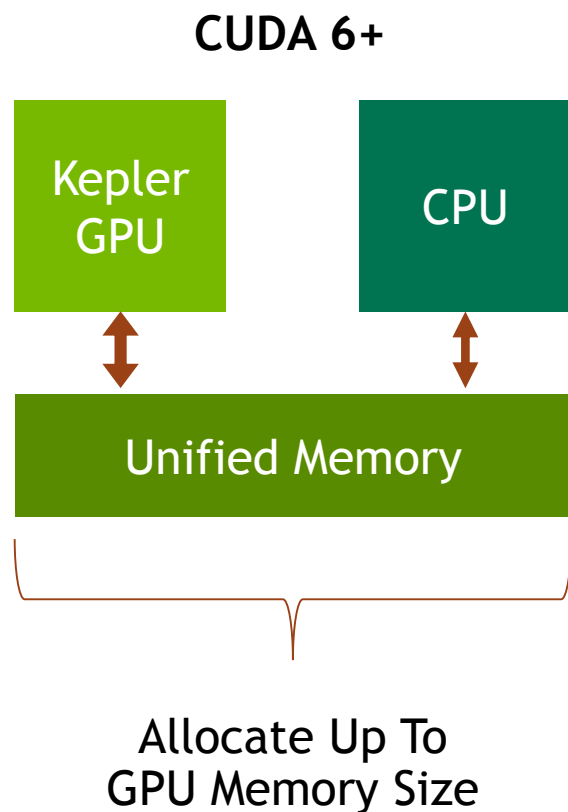
GPU page faulting capability

Can handle thousands of simultaneous page faults

Up to 2 MB page size

Better TLB coverage of GPU memory

Kepler/Maxwell Unified Memory



Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit copy*
- Greatly simplifies code porting

Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherency
- Still allows explicit hand tuning

Pascal Unified Memory

Large datasets, simple programming, High Performance

CUDA 8



Unified Memory

Allocate Beyond
GPU Memory Size

Enable Large
Data Models

Oversubscribe GPU memory
Allocate up to system memory size

Tune
Unified Memory
Performance

Usage hints via `cudaMemAdvise` API
Explicit prefetching API

Simpler
Data Access

CPU/GPU Data coherence
Unified memory atomic operations

Introducing Tesla p100

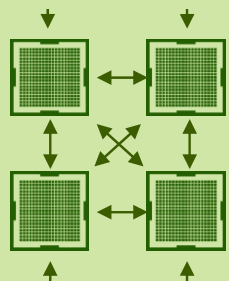
New GPU Architecture to Enable the World's Fastest Compute Node

Pascal Architecture



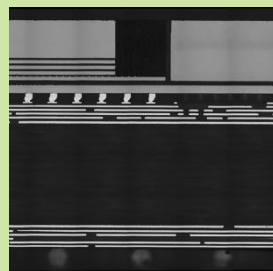
Highest Compute Performance

NVLink



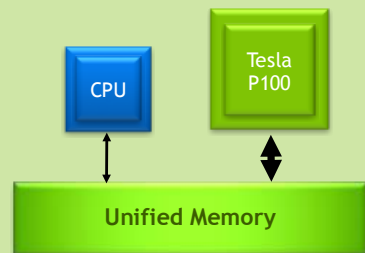
GPU Interconnect for Maximum Scalability

HBM2 Stacked Memory



Unifying Compute & Memory in Single Package

Page Migration Engine



Simple Parallel Programming with 512 TB of Virtual Memory

More P100 Features: compute preemption, new instructions, larger L2 cache, more...

Find out more at <http://devblogs.nvidia.com/parallelforall/inside-pascal>

OPTIMIZATION 1

LAUNCH CONFIGURATION

Optimization priorities

Focusing on the tall poles

- Parallelize sequential code
- Minimize host-device data transfers
- Optimize resource utilization
 - Maximize device utilization
 - Efficient global memory accesses
 - Minimize long sequences of divergent execution within warps

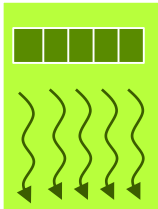
<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>

Execution Model

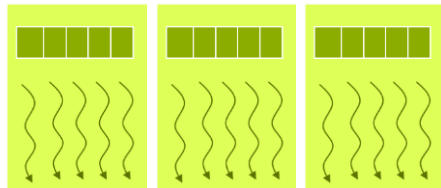
Software



Thread



Thread Block



Grid

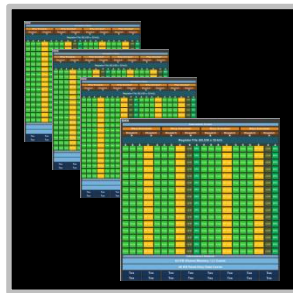
Hardware



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

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

Example P100

56 SM @ 32 thread = 1792 threadblocks to fill the GPU **ONCE**

But effective number of threadblocks per SM depends on the kernel

Launch Configuration: General Guidelines

How many threads per block should we choose?

The really short answer: 128, 256, or 512 are often good choices

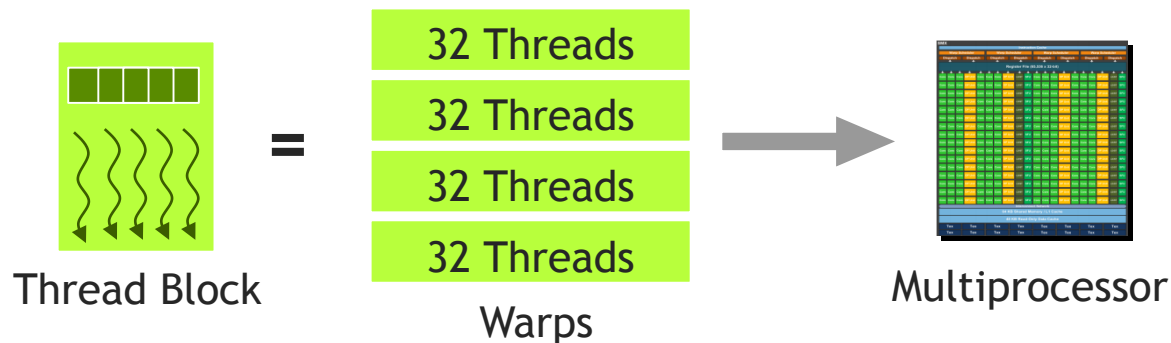
The slightly longer answer:

- 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





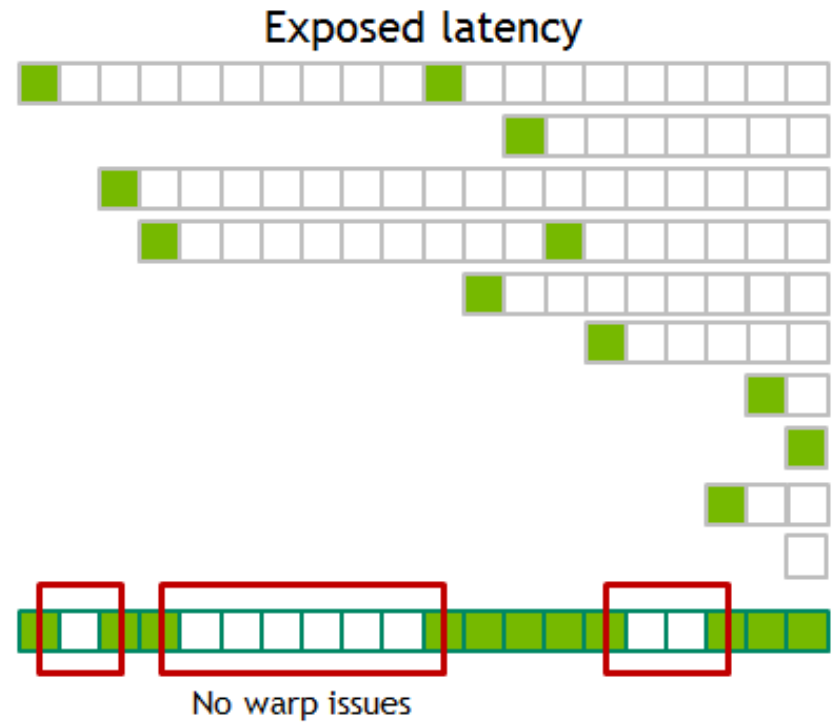
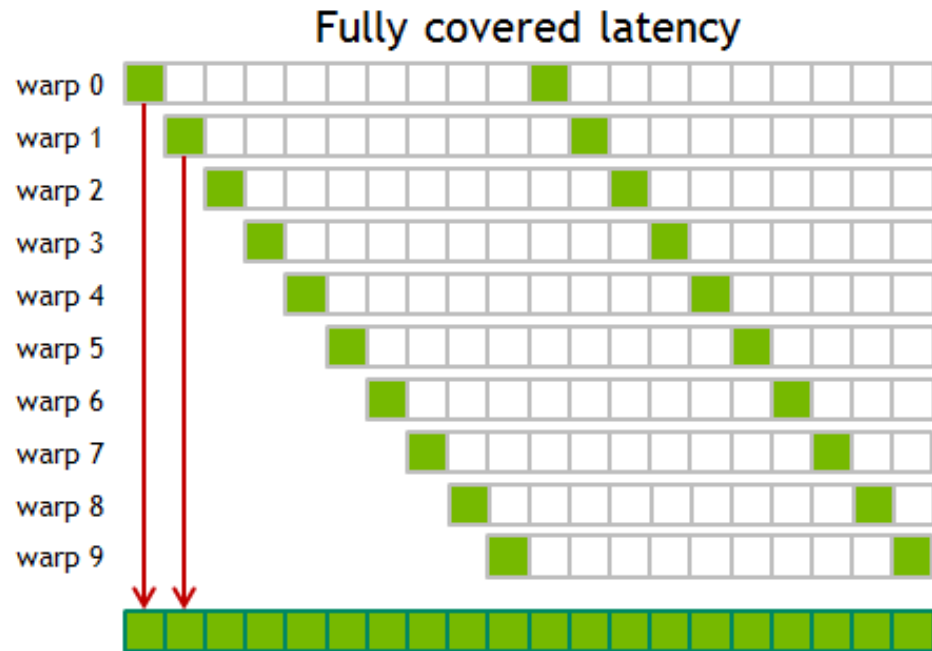
A thread block consists of *warps of* 32 threads

A warp is executed physically in parallel on some multiprocessor.

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

Latency Hiding

-  The warp issues
-  The warp waits (latency)



Latency Hiding

Instruction latencies

Roughly 10-20 cycles for arithmetic operations

Global memory 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%)

Occupancy limiters:

Threads per thread block

Registers per thread

Shared memory per thread block

Pascal P100 SM resources:

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

Occupancy and Performance

Note that 100% occupancy isn't needed to reach maximum performance

- Higher occupancy gives SM scheduler more choice to select next warp

- 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

Occupancy examples

Maximal occupancy, e.g. for BW limited kernel:

64 warps -> maximum choice for scheduler

32 thread blocks -> minimal impact of barriers

=> 2 warps / thread block = 64 threads optimal (Kepler GK110: 128)

Maximum number of registers per thread without limiting occupancy:

$64k \text{ regs} / (64 \text{ warps} * 32 \text{ threads/warp}) = 32 \text{ regs/thread}$

Maximum amount of shared memory per block:

$48KB / 32 \text{ thread blocks} = 1526 \text{ B}$

Pascal P100 SM resources:

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

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

Pascal SM can run up to 32 thread blocks concurrently

SM can reach the block count limit before reaching good occupancy

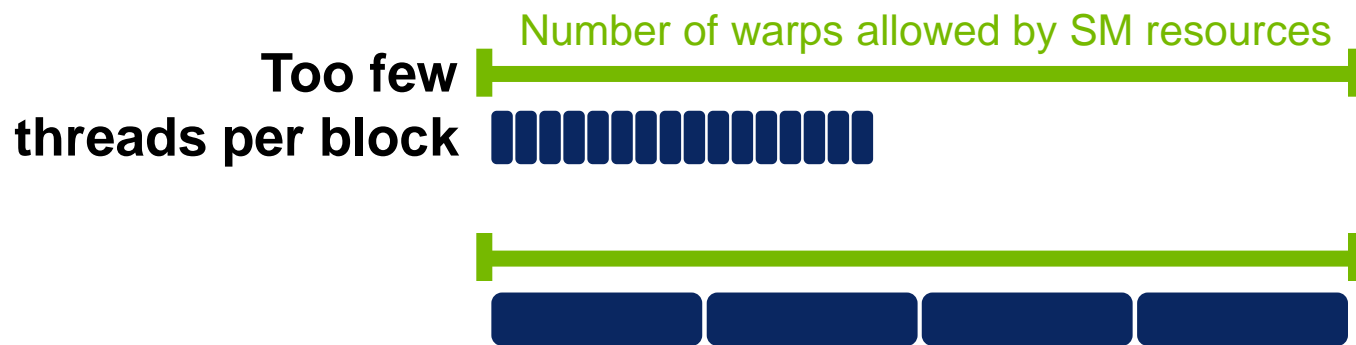
E.g.: 1-warp blocks = 16 warps/SM on P100 (50% occ - lower end)

Thread blocks can be too big

Enough SM resources for more threads, but not enough for a whole block

A thread block isn't started until resources are available for all of its threads

Thread Block Sizing



SM resources:

Registers

Shared memory



CUDA Occupancy Calculator

Analyze effect of resource consumption on occupancy

CUDA GPU Occupancy Calculator

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

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

2.) Enter your resource usage:
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	Blocks Per SM	= Allocatable
Warps (Threads Per Block / Threads Per Warp)	8	64	8	
Registers (Warp limit per SM due to per-warp reg count)	8	128	16	
Shared Memory (Bytes)	4096	49152	12	

Note: SM is an abbreviation for (Streaming) Multiprocessor

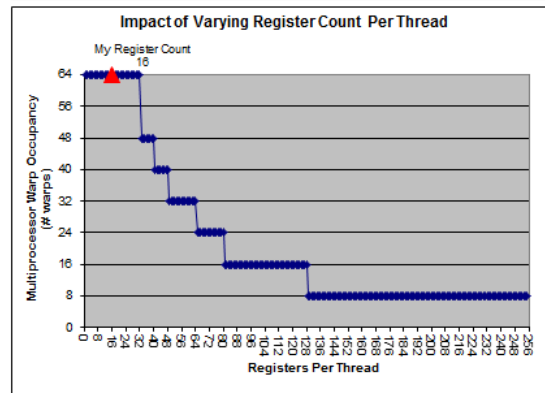
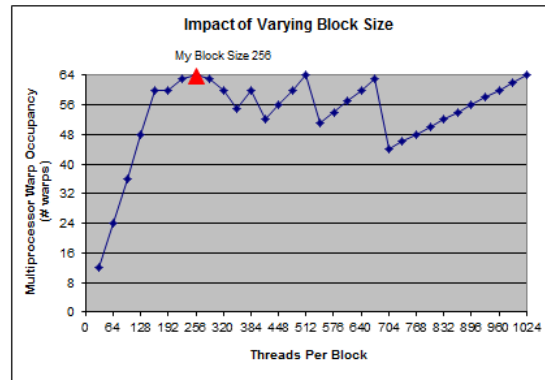
Maximum Thread Blocks Per Multiprocessor	Blocks/SM * Warps/Block = Warps/SM		
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	16		
Limited by Shared Memory per Multiprocessor	12		

Note: Occupancy limited to 100% in orange

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

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit <http://developer.nvidia.com/cuda>

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.



Physical Max Warps/SM	64	Occupancy = 64 / 64 = 100%
Physical Max Blocks/SM	16	
Physical Max Registers/SM	128	
Physical Max Shared Memory/SM (bytes)	49152	
Physical Max Thread Blocks/SM	8	
Physical Max Threads/SM	2048	



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%

Requests per Thread and Performance



Experiment: vary size of accesses by threads of a warp, check performance

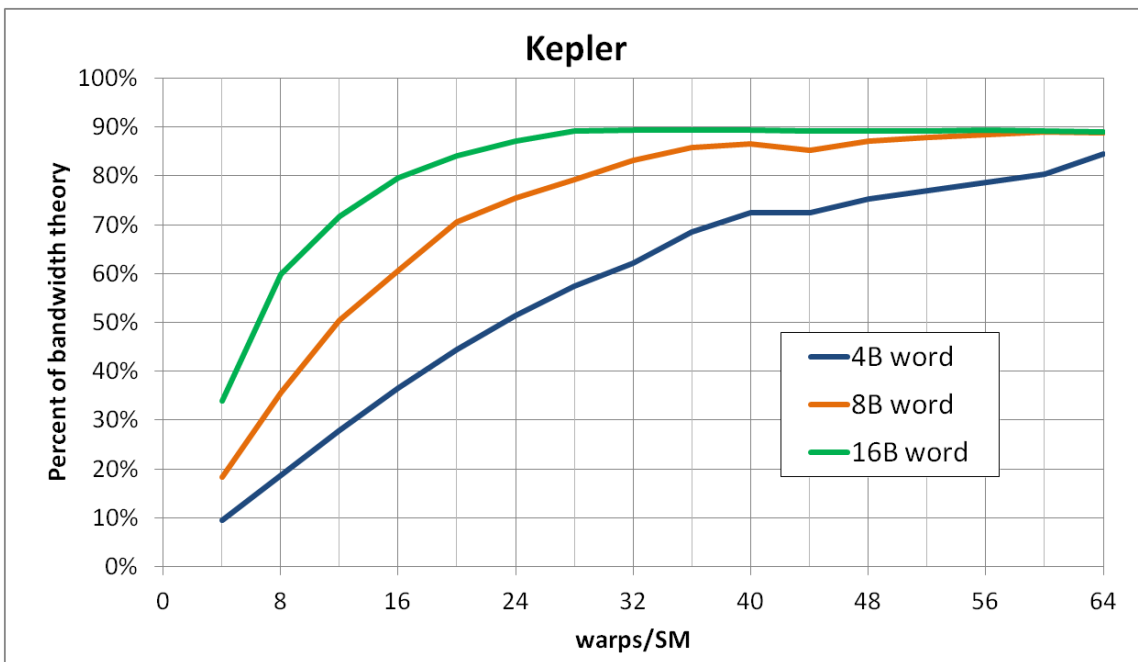
Memcopy kernel: each warp has 2 concurrent requests (one write and the read following it)

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

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

OPTIMIZATION 2

GLOBAL MEMORY ACCESS

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

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

Access Patterns vs. Memory Throughput

Scenario:

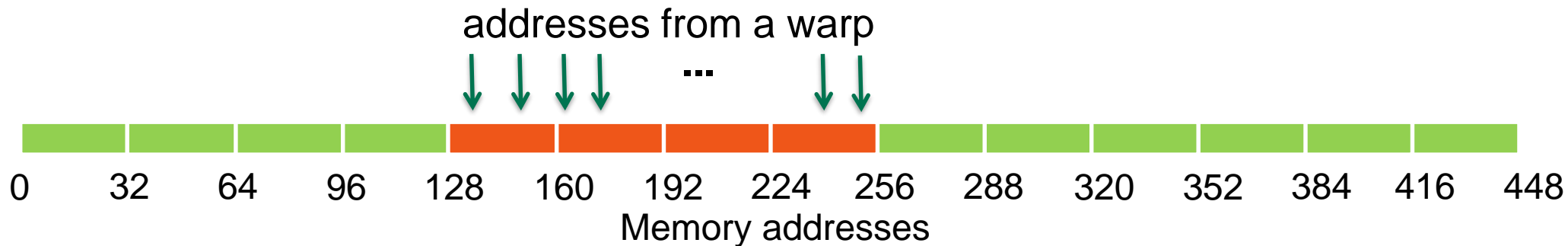
Warp requests 32 aligned, consecutive 4-byte words

Addresses fall within 4 segments

Warp needs 128 bytes

128 bytes move across the bus

Bus utilization: 100%



Access Patterns vs. Memory Throughput

Scenario:

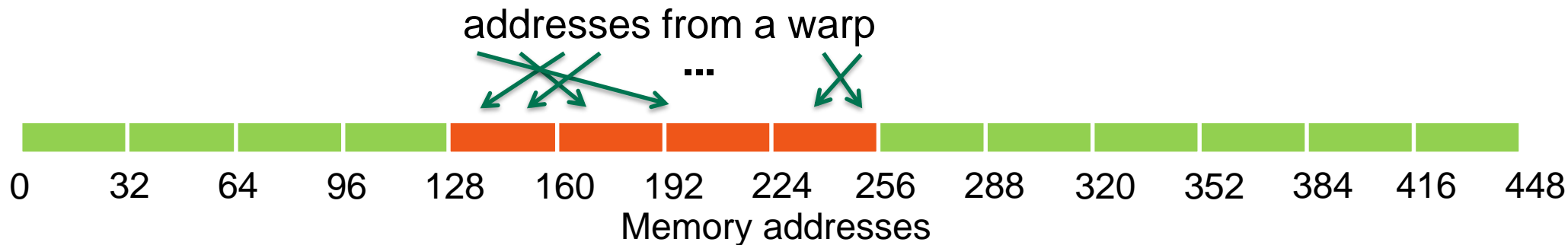
Warp requests 32 aligned, permuted 4-byte words

Addresses fall within 4 segments

Warp needs 128 bytes

128 bytes move across the bus

Bus utilization: 100%



Access Patterns vs. Memory Throughput

Scenario:

Warp requests 32 misaligned, consecutive 4-byte words

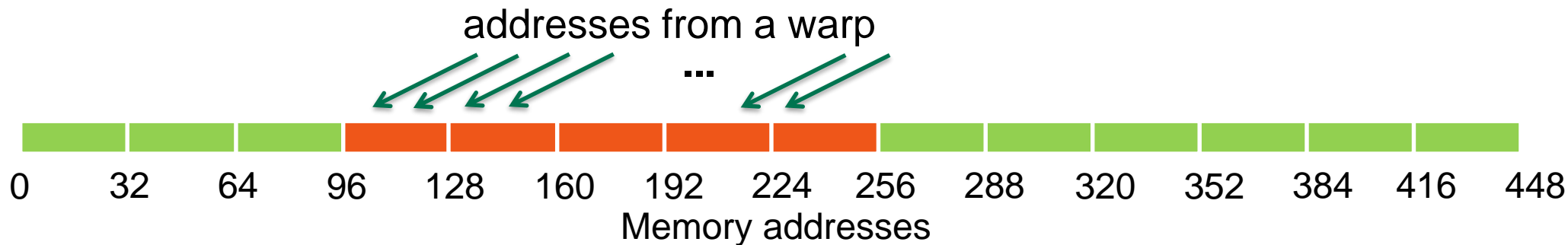
Addresses fall within at most 5 segments

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



Access Patterns vs. Memory Throughput

Scenario:

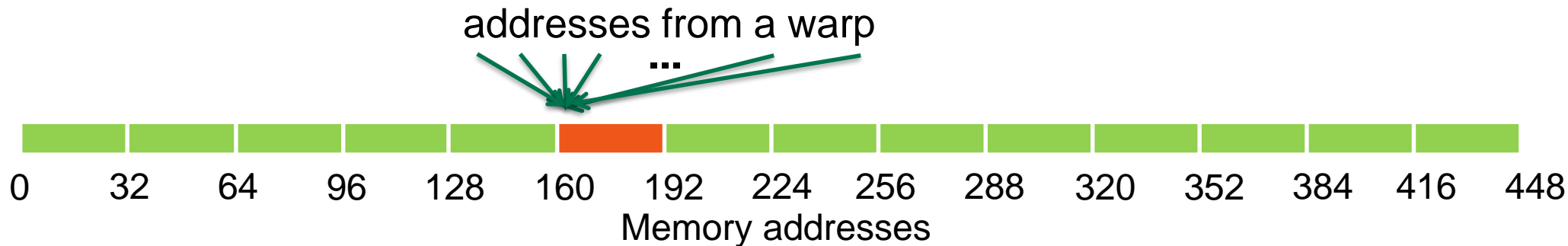
All threads in a warp request the same 4-byte word

Addresses fall within a single segment

Warp needs 4 bytes

32 bytes move across the bus

Bus utilization: 12.5%



Access Patterns vs. Memory Throughput

Scenario:

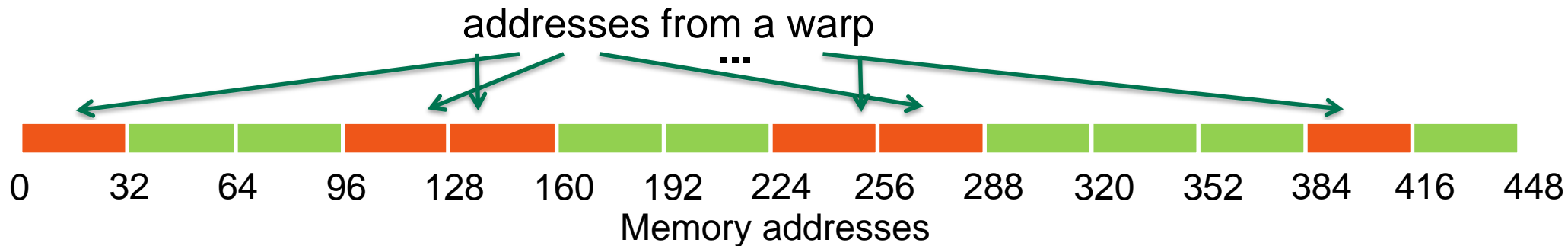
Warp requests 32 scattered 4-byte words

Addresses fall within N segments

Warp needs 128 bytes

$N \times 32$ bytes move across the bus

Bus utilization: $128 / (N \times 32)$



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];
    ...
}
```


Structure of Non-Native Size

Compiler converts `temp = data[idx]` into 3 loads:

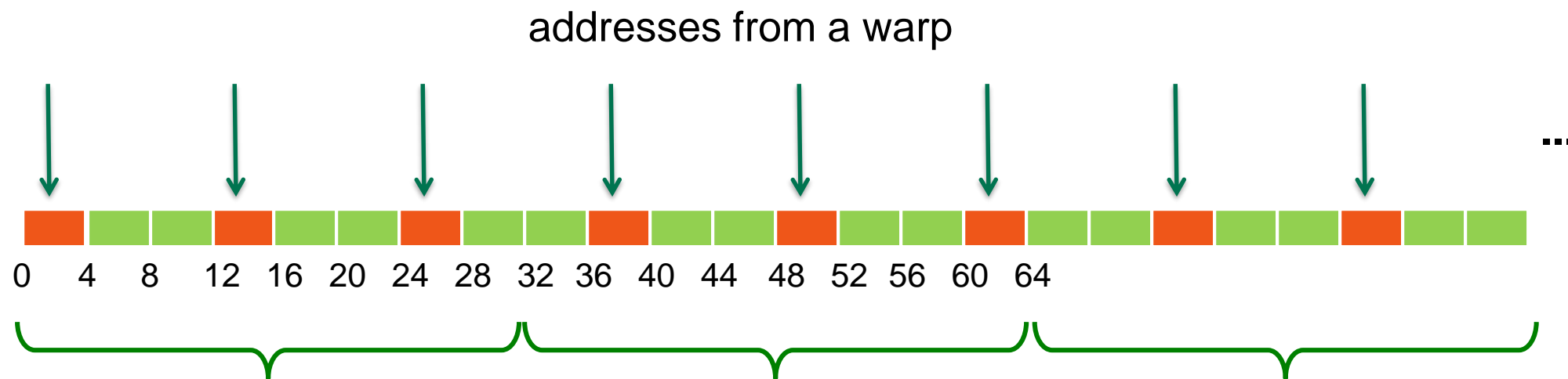
- Each loads 4 bytes

- 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

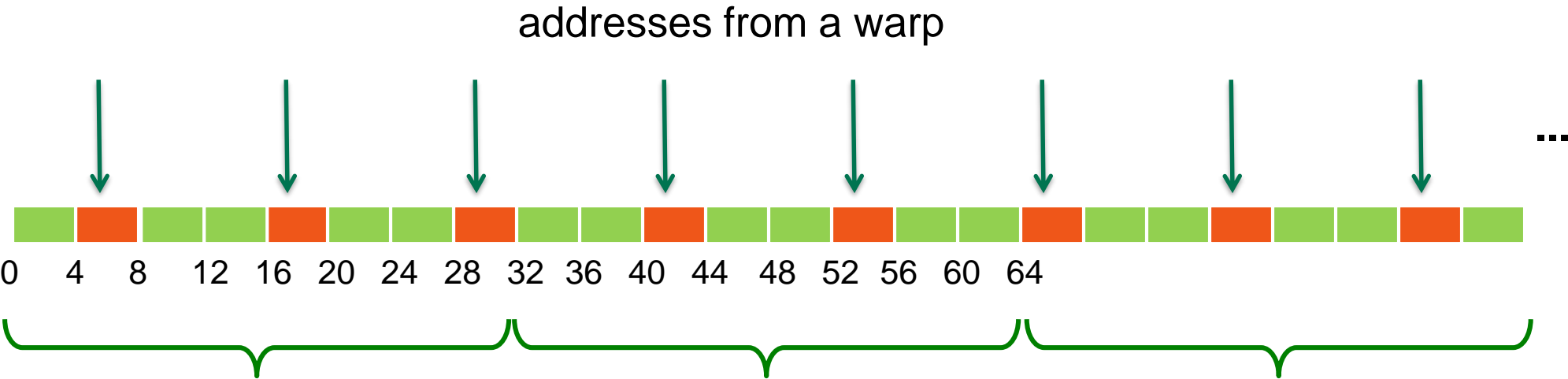
Addresses per warp for each of the loads:

- Successive threads read 4 bytes at 12-byte stride

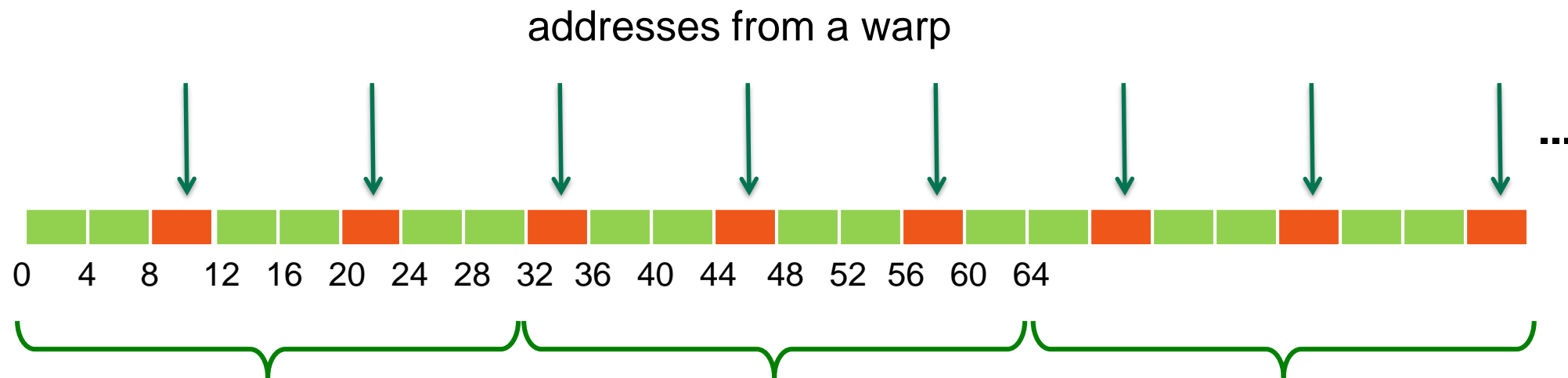
First Load Instruction



Second Load Instruction



Third Load Instruction



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

A note about caches

L1 and L2 caches

Ignore in software design

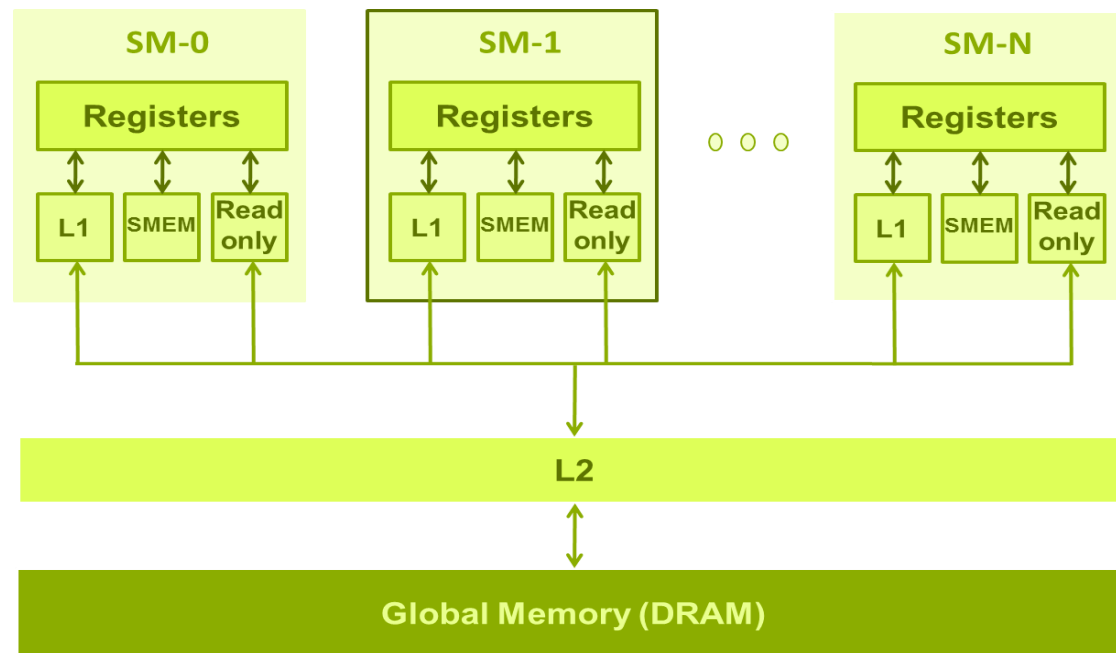
Thousands of concurrent threads -
cache blocking difficult at best

Read-only Data Cache

Shared with texture pipeline

Useful for uncoalesced reads

Handled by compiler when `const __restrict__` is used, or use `__ldg()` primitive



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

OPTIMIZATION 3

INSTRUCTION THROUGHPUT

Exposing Sufficient Parallelism

What SMX ultimately needs:

- Sufficient number of independent instructions

Two ways to increase parallelism:

- More independent instructions (ILP) within a thread (warp)

- More concurrent threads (warps)

Independent Instructions: ILP vs. TLP

SM 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, Pascal SM can leverage that. (E.g., a small loop that is unrolled.)

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

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 {...}
```

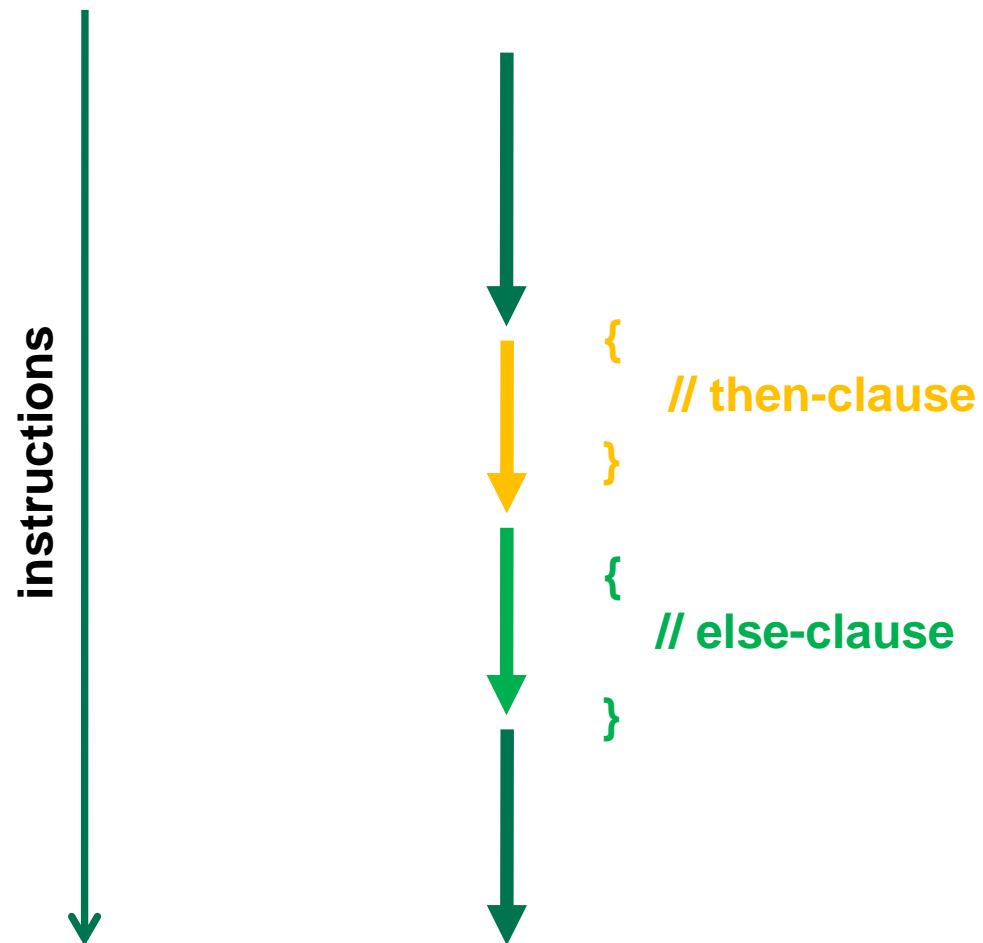
Branch granularity < warp size

Example without divergence:

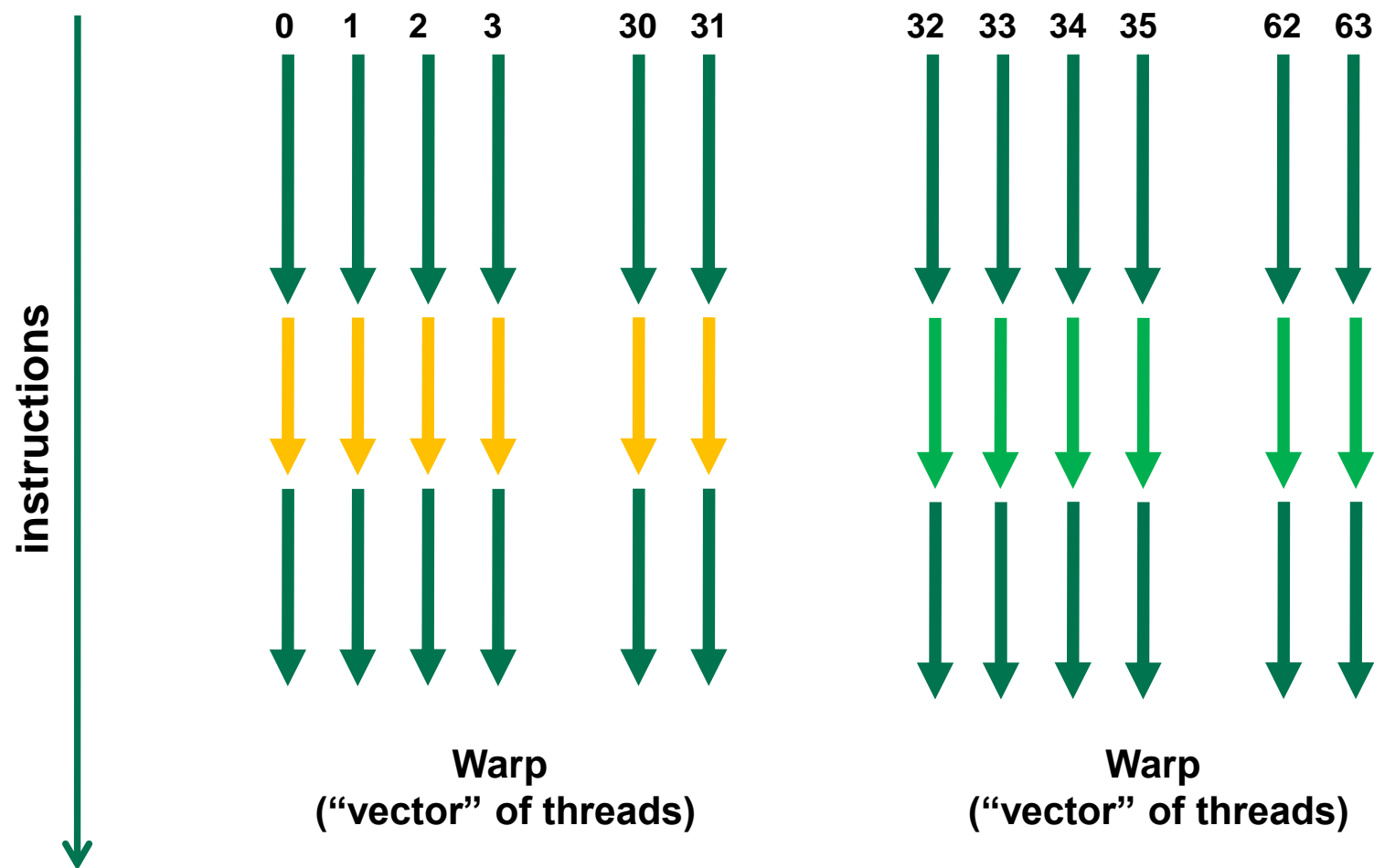
```
if (threadIdx.x / warpSize > 2) {...} else {...}
```

Branch granularity is a whole multiple of warp size

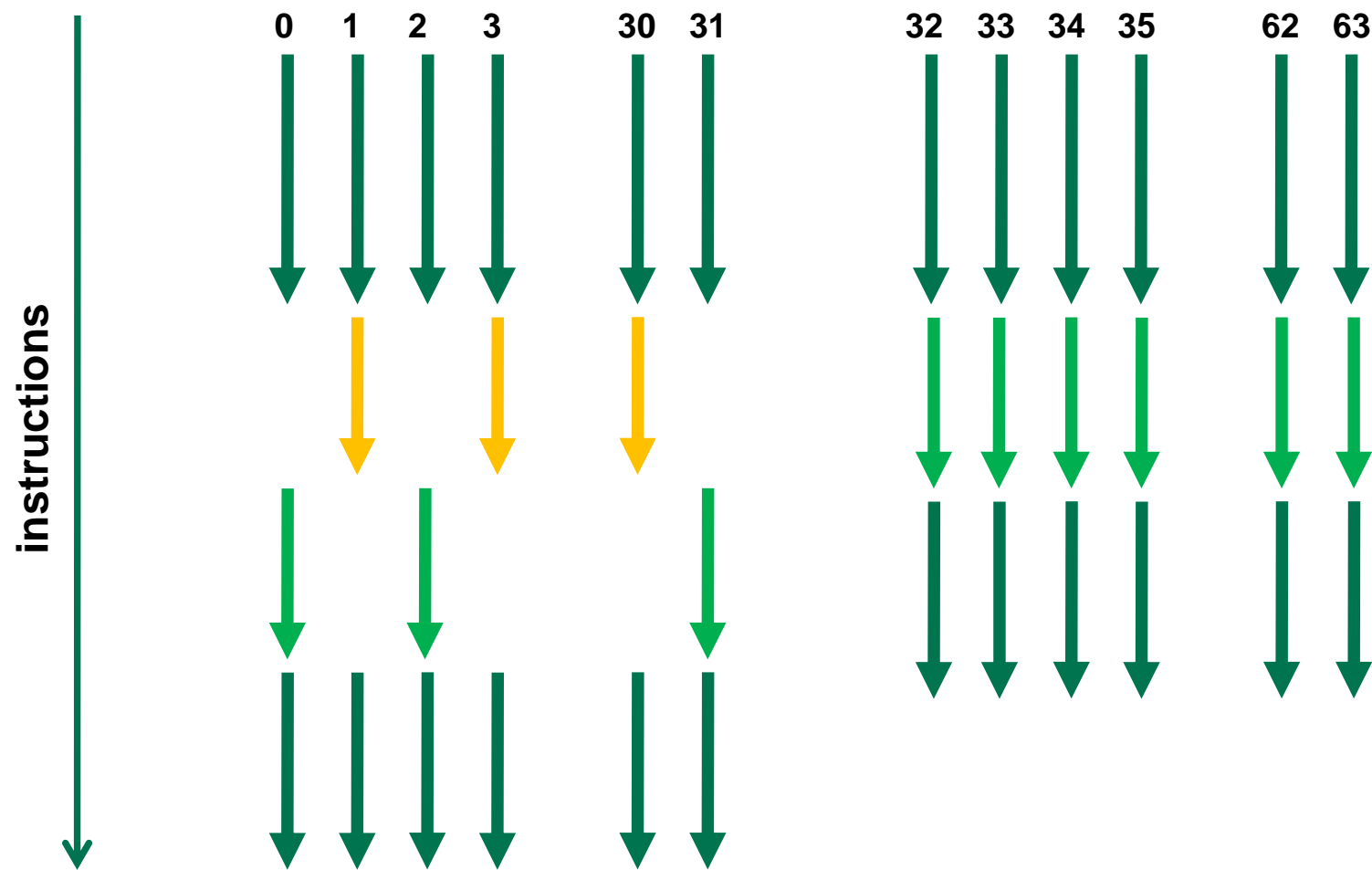
Control Flow



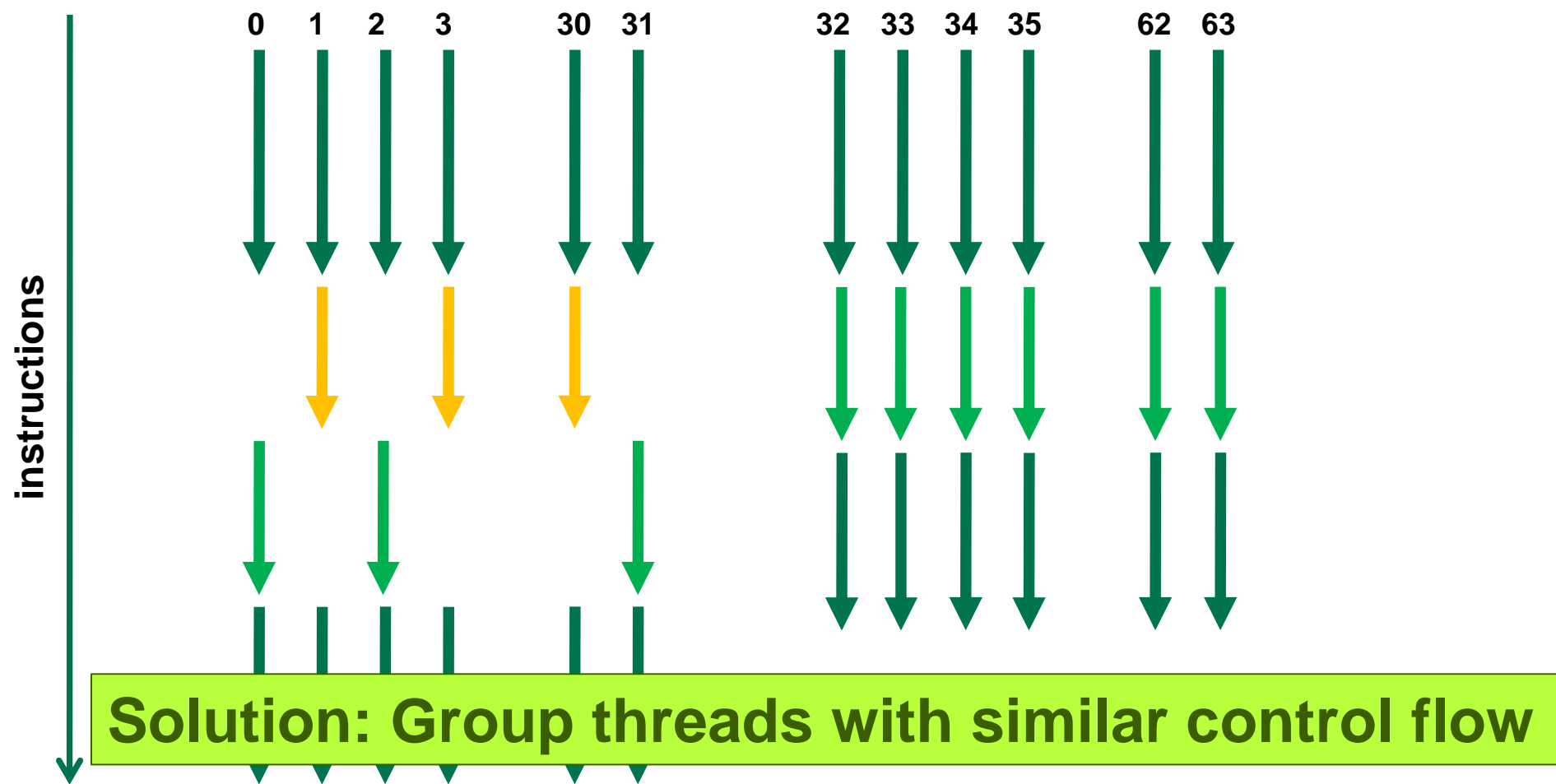
Execution within warps is coherent



Execution diverges within a warp



Execution diverges within a warp



Topics not covered here..

Streams

- Improved device utilization via concurrent kernels

- Asynchronous data transfers

Unified memory

- Minimizing impact of host/device transfers

- Memory hints

- Global atomics

New Instructions

- FP16, INT8 vector instructions

<http://docs.nvidia.com/cuda/pascal-tuning-guide/>

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/>

