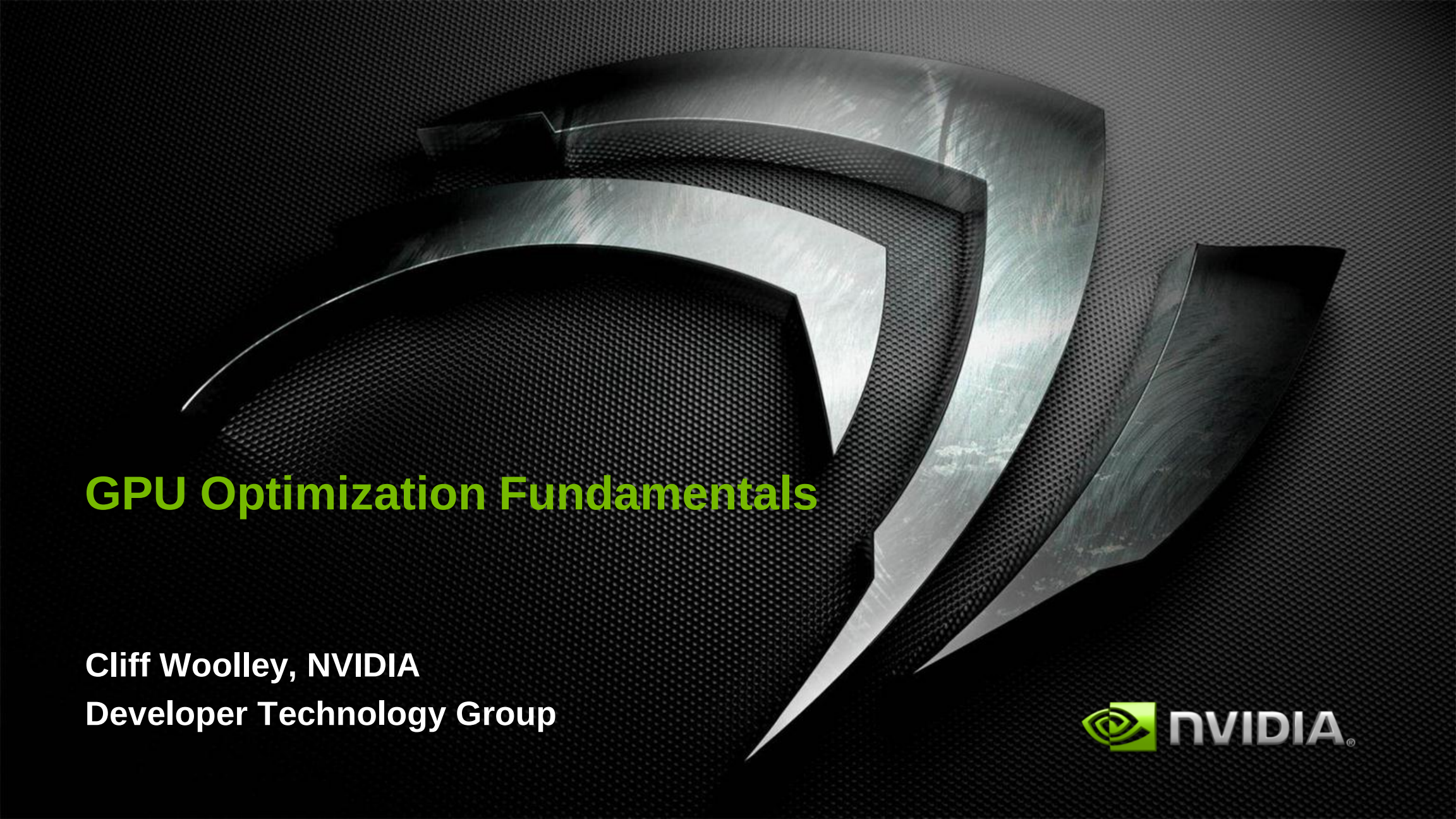


GPU Optimization Fundamentals

Cliff Woolley, NVIDIA
Developer Technology Group

NVIDIA

Main Requirements for GPU Performance

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

GPU Optimization Fundamentals

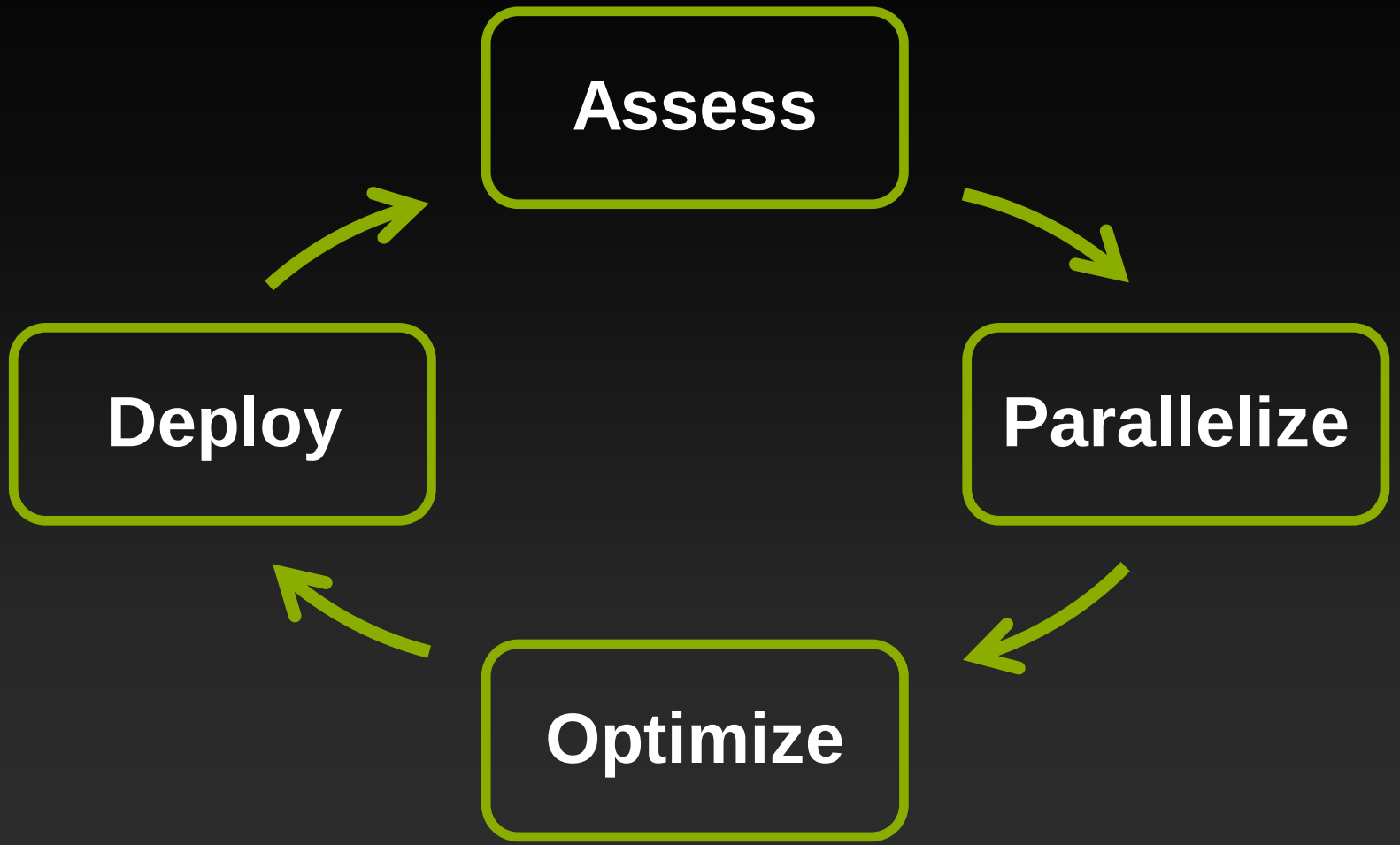
- Find ways to parallelize sequential code
- Adjust kernel launch configuration to maximize device utilization
- Ensure global memory accesses are coalesced
- Minimize redundant accesses to global memory
- Avoid different execution paths within the same warp
- Minimize data transfers between the host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

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

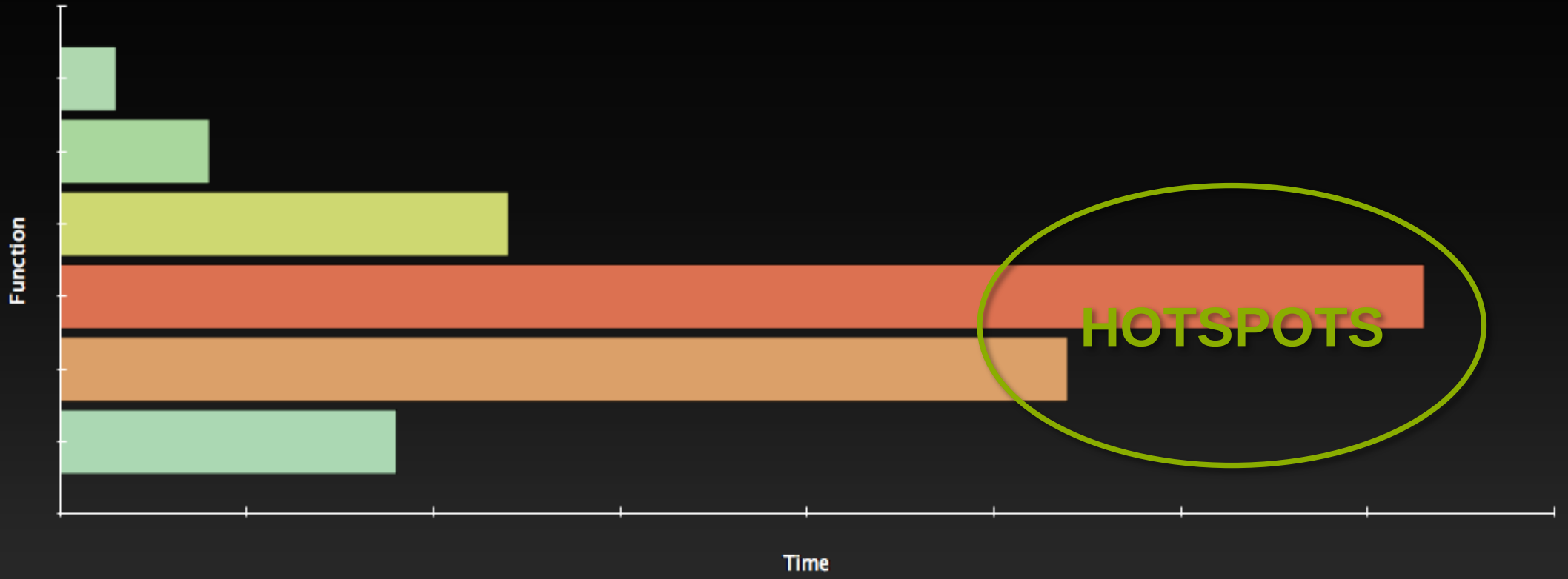
GPU Optimization Fundamentals

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

APOD: A Systematic Path to Performance

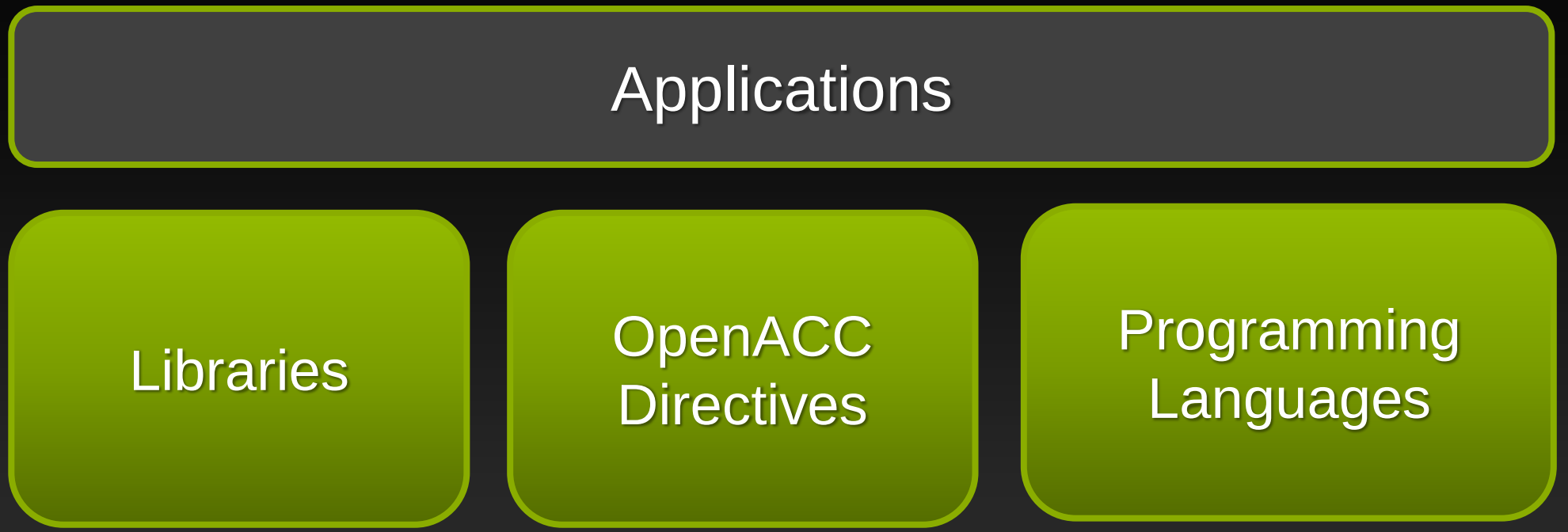


Assess



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

Parallelize



Optimize

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

Deploy

- Check API return values
- Run cuda-memcheck tools
- Library distribution
- Cluster management

Early gains
Subsequent changes are evolutionary

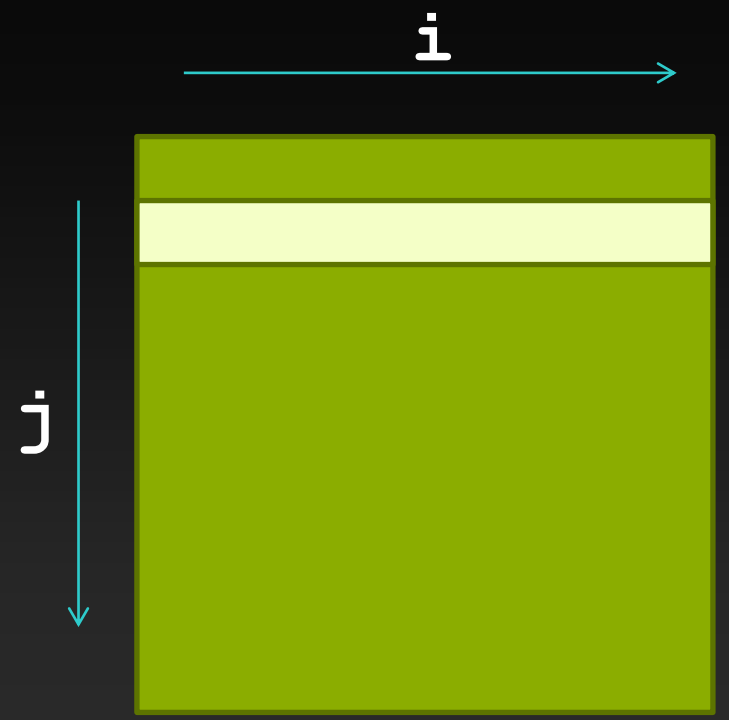
PARALLELIZE

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

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];
}
```



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);
```

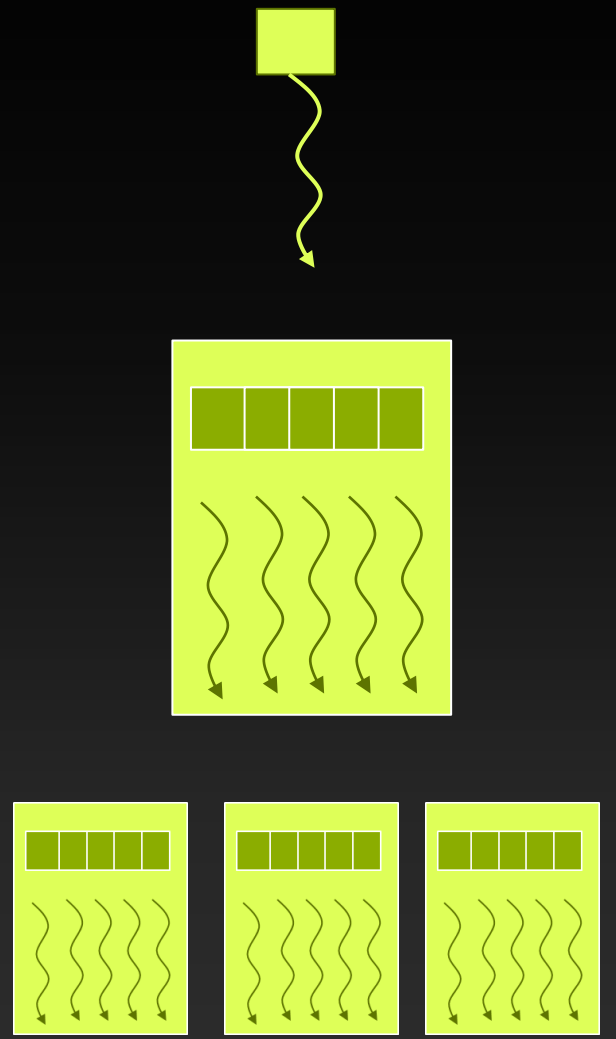
+ Quickly implemented

- Performance weak

⇒Need to expose parallelism!

CUDA Execution Model

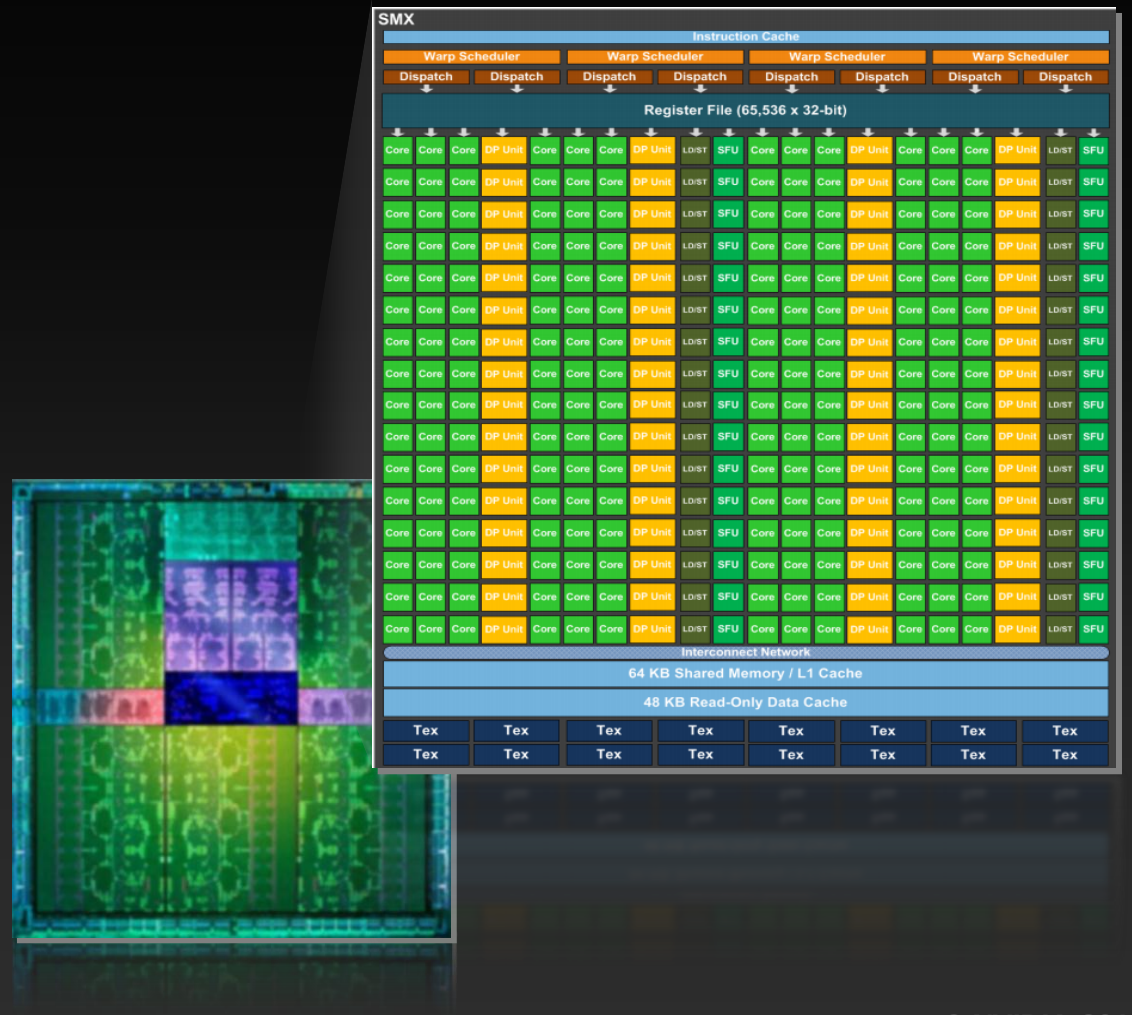
- Thread: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel
- Threads Block: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange
- 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



Kepler Streaming Multiprocessor (SMX)

Per SMX:

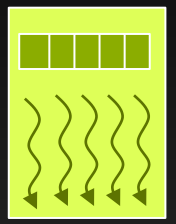
- 192 SP CUDA Cores
- 64 DP CUDA Cores
- 4 warp schedulers
  - Up to 2048 concurrent threads
  - One or two instructions issued per scheduler per clock from a single warp
- Register file (256KB)
- Shared memory (48KB)



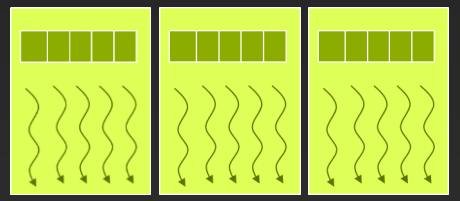
Execution Model

Software

Thread



Thread Block

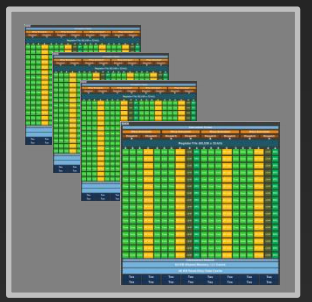


Grid

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

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

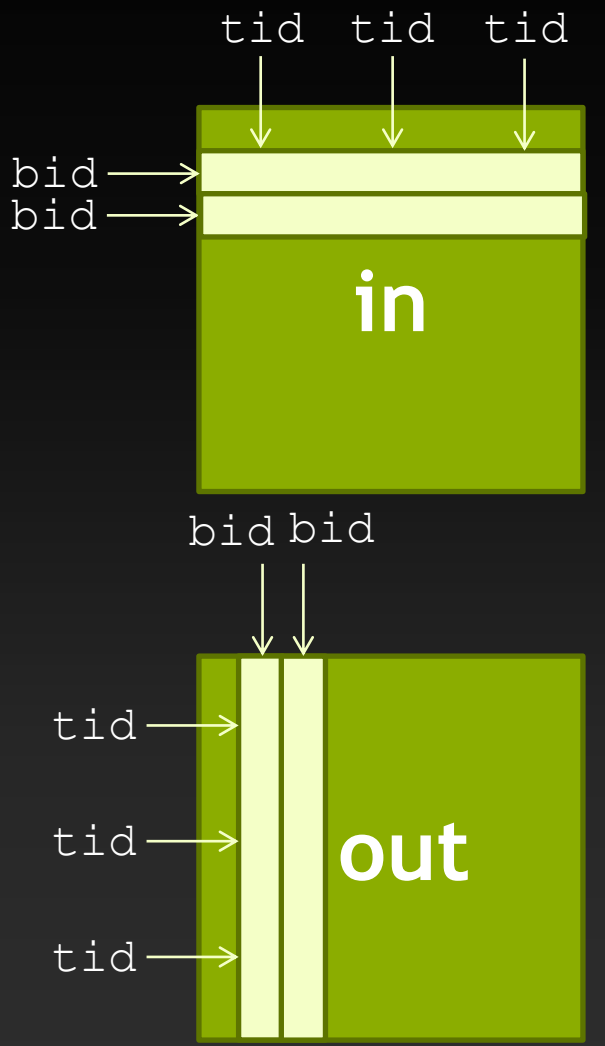
- Performance weak

⇒Need to expose parallelism!

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);
```



OPTIMIZE

OPTIMIZE

Kernel Optimizations: *Launch Configuration*

Launch Configuration

- Launch configuration is the number of blocks and number of threads per block, expressed in CUDA with the <<< >>> notation:

```
mykernel<<<num_blocks,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*

Warps

A thread block consists of 32-thread *warps*

A warp is executed physically in parallel (SIMD) on a multiprocessor

Occupancy

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

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

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%

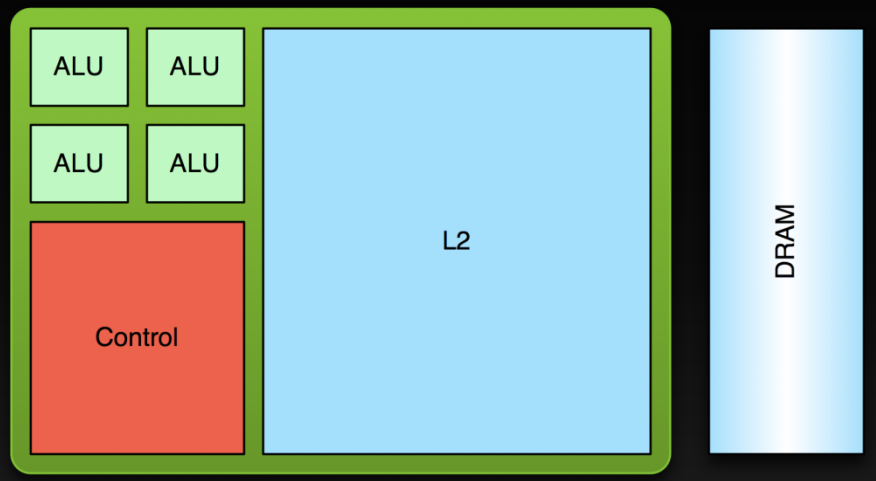
Occupancy

- Occupancy: number of concurrent threads per SM, expressed as:
  - Number of concurrent warps of threads, or
  - Percentage of maximum concurrent threads
- Determined by several factors :
  - Registers per thread
    - SM registers are partitioned among the threads
  - Shared memory per thread block
    - SM shared memory is partitioned among the blocks
  - Threads per thread block
    - Threads are allocated at thread block granularity

Kepler SM resources:

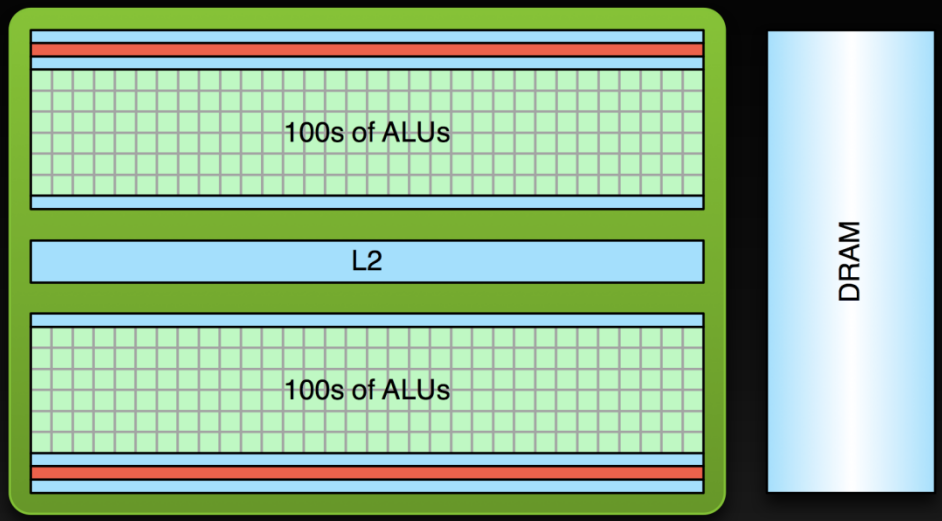
- 64K 32-bit registers
- Up to 48 KB of shared memory
- Up to 2048 concurrent threads
- Up to 16 concurrent thread blocks

Low Latency or High Throughput?



CPU

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

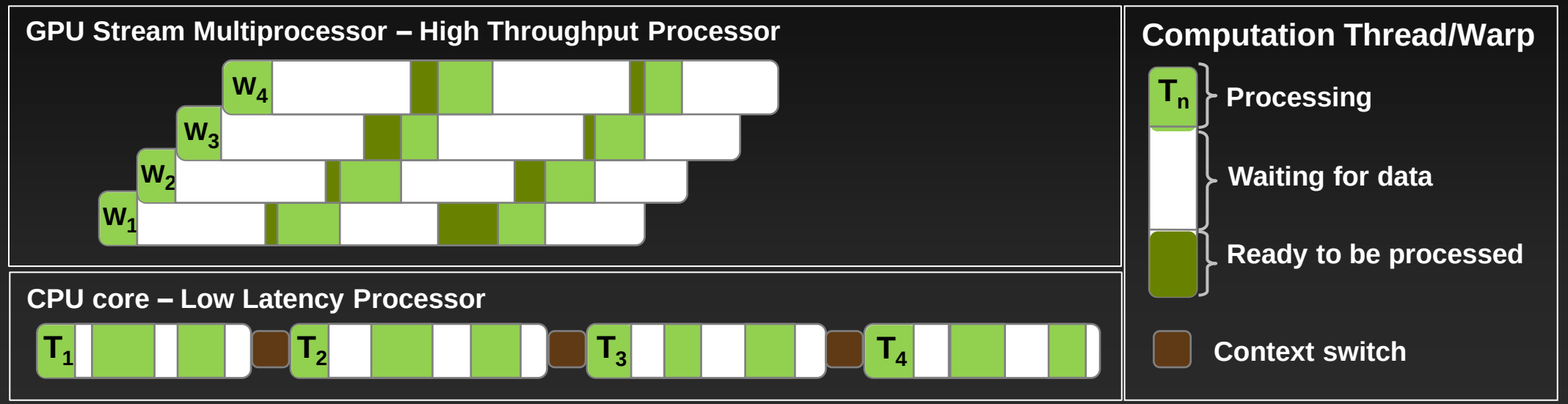


GPU

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

Low Latency or High Throughput?

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



Occupancy and Performance

- Note that 100% occupancy isn’t needed to reach maximum performance
  - Once the “needed” occupancy is reached, further increases won’t improve performance
- Needed occupancy 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

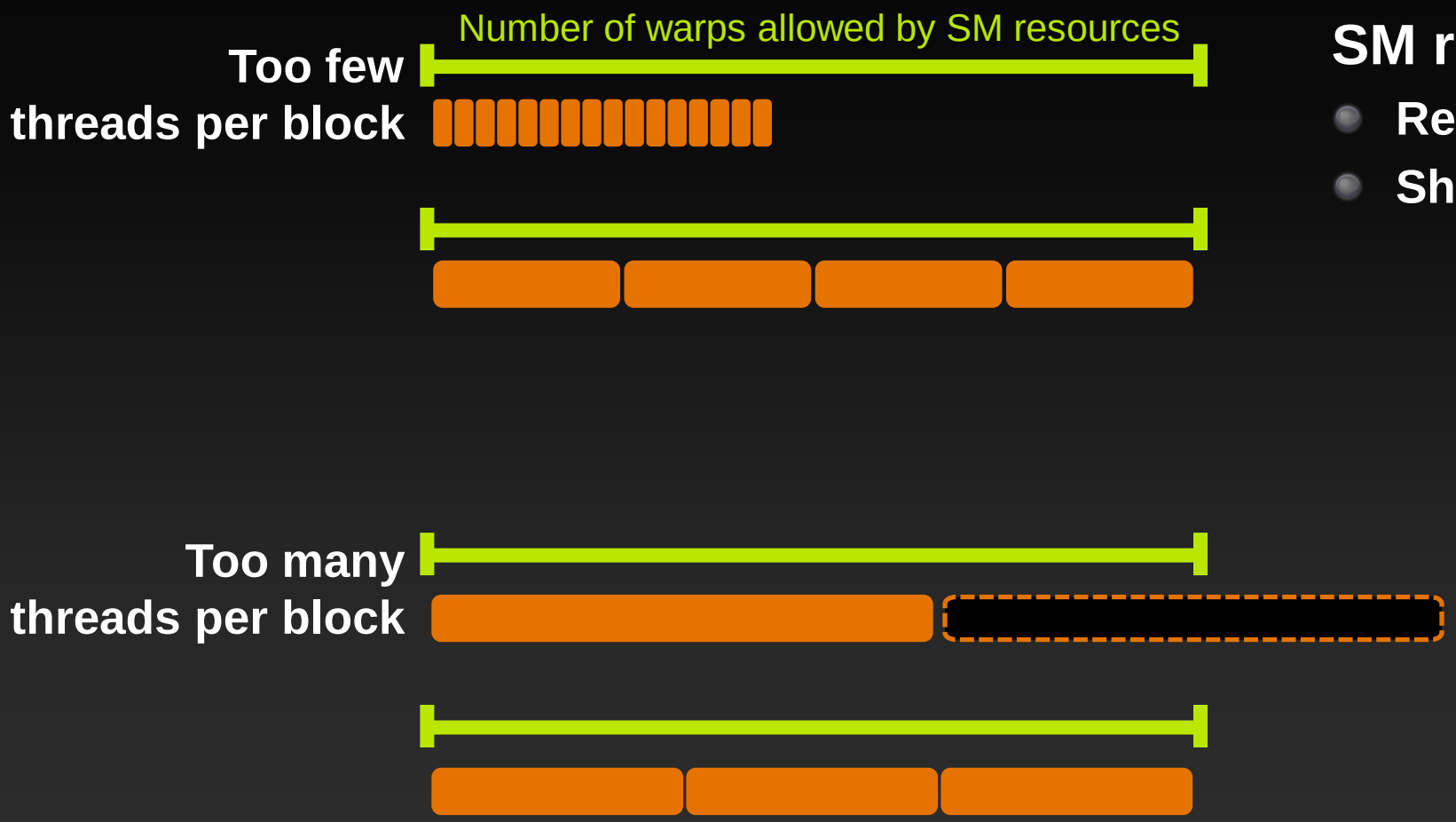
- Limiting resources:
  - Number of threads
  - Number of registers per thread
  - Number of blocks
  - Amount of shared memory per block
- Don’t need for 100% occupancy for maximum performance

Start	588.755 ms
End	588.808 ms
Duration	53.344 us
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%

Thread Block Size and Occupancy

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

Thread Block Sizing



SM resources:

- Registers
- Shared memory

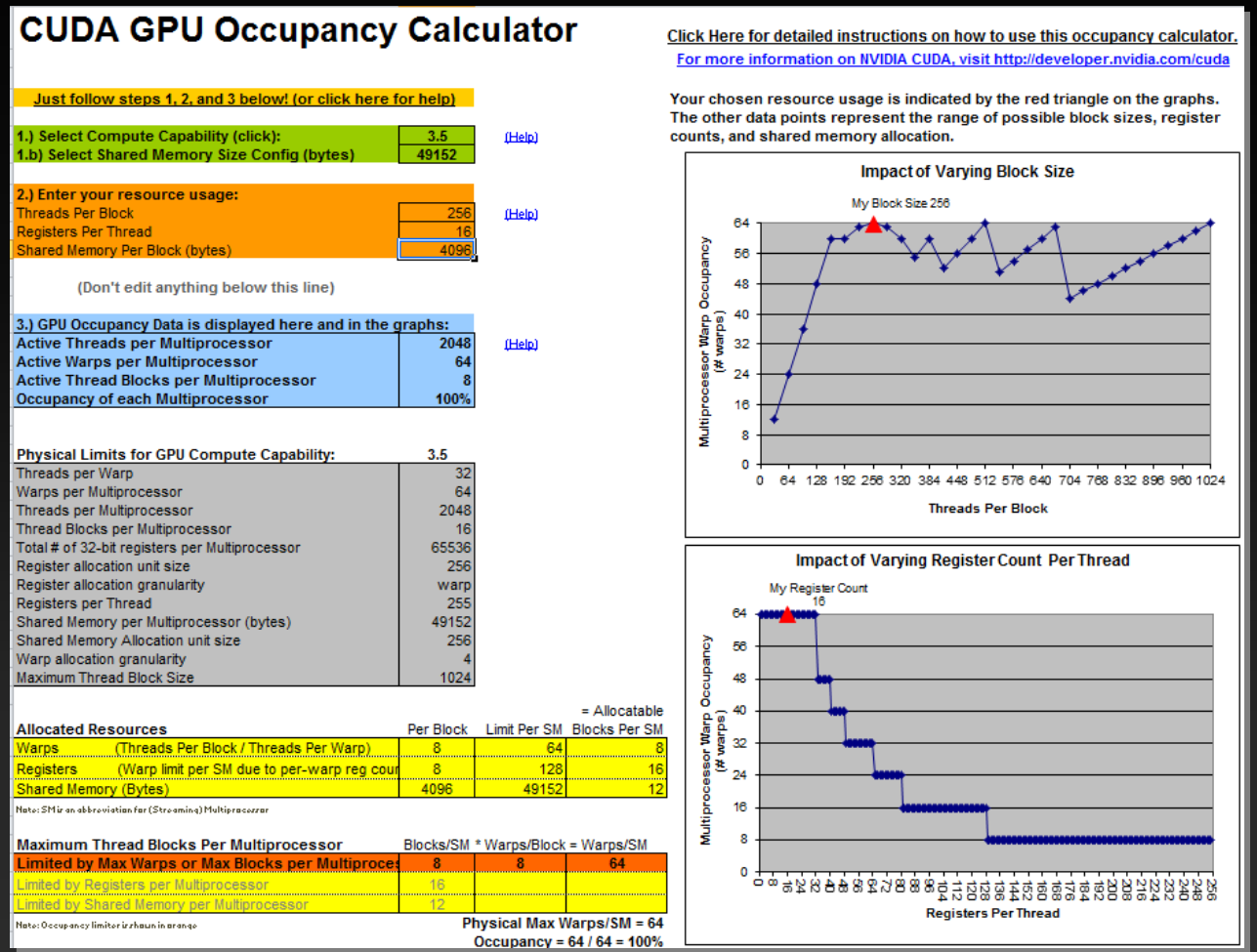
Occupancy Example

- 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%

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%	

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.

Impact of Varying Block Size

Impact of Varying Register Count Per Thread

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 cou	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%

General Guidelines

- Thread block size choice:
  - Start with 128-256 threads per block
    - Adjust up/down by what best matches your function
    - Example: stencil codes prefer larger blocks to minimize halos
  - Multiple of warp size (32 threads)
  - If occupancy is critical to performance:
    - Check that block size isn't precluding occupancy allowed by register and shared memory resources
- Grid size:
  - 1,000 or more thread blocks
    - 10s of "waves" of thread blocks: no need to think about tail effect
    - Makes your code ready for several generations of future GPUs

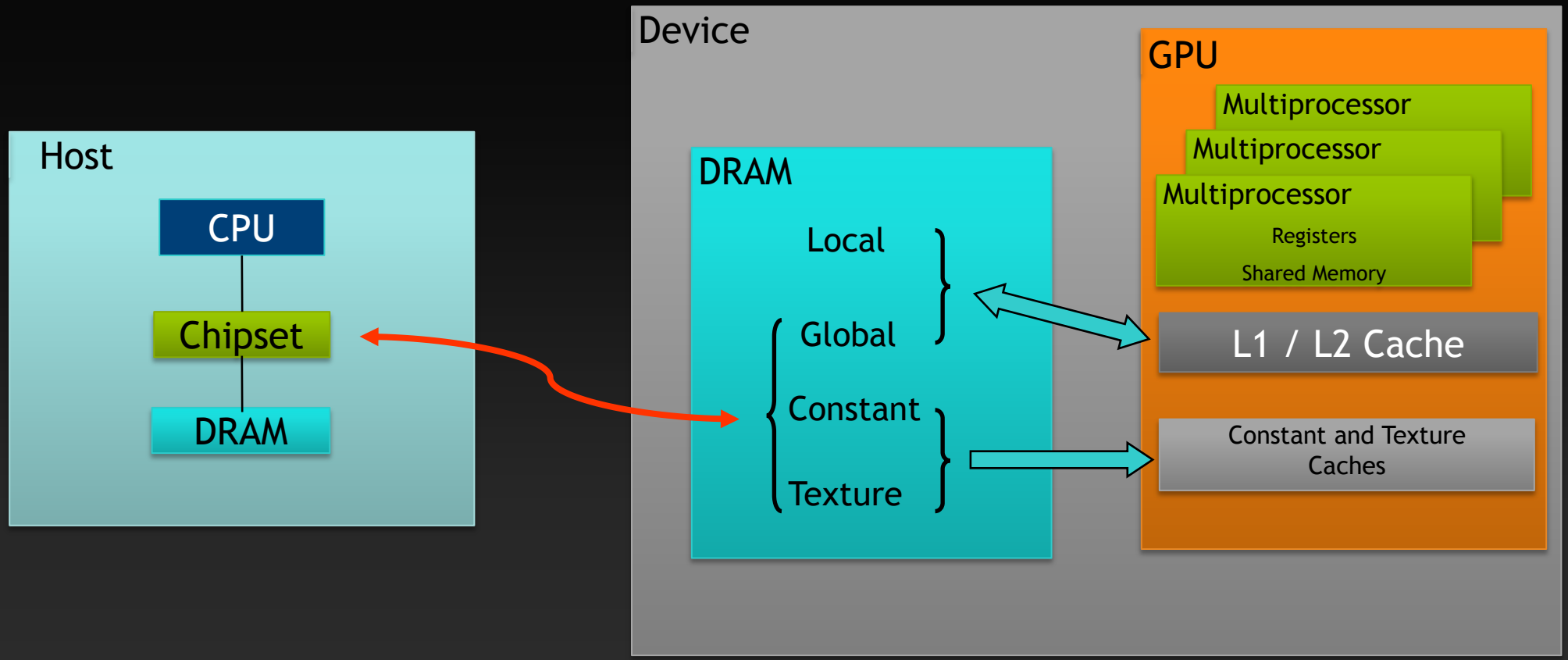
Kepler: Level of Parallelism Needed

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

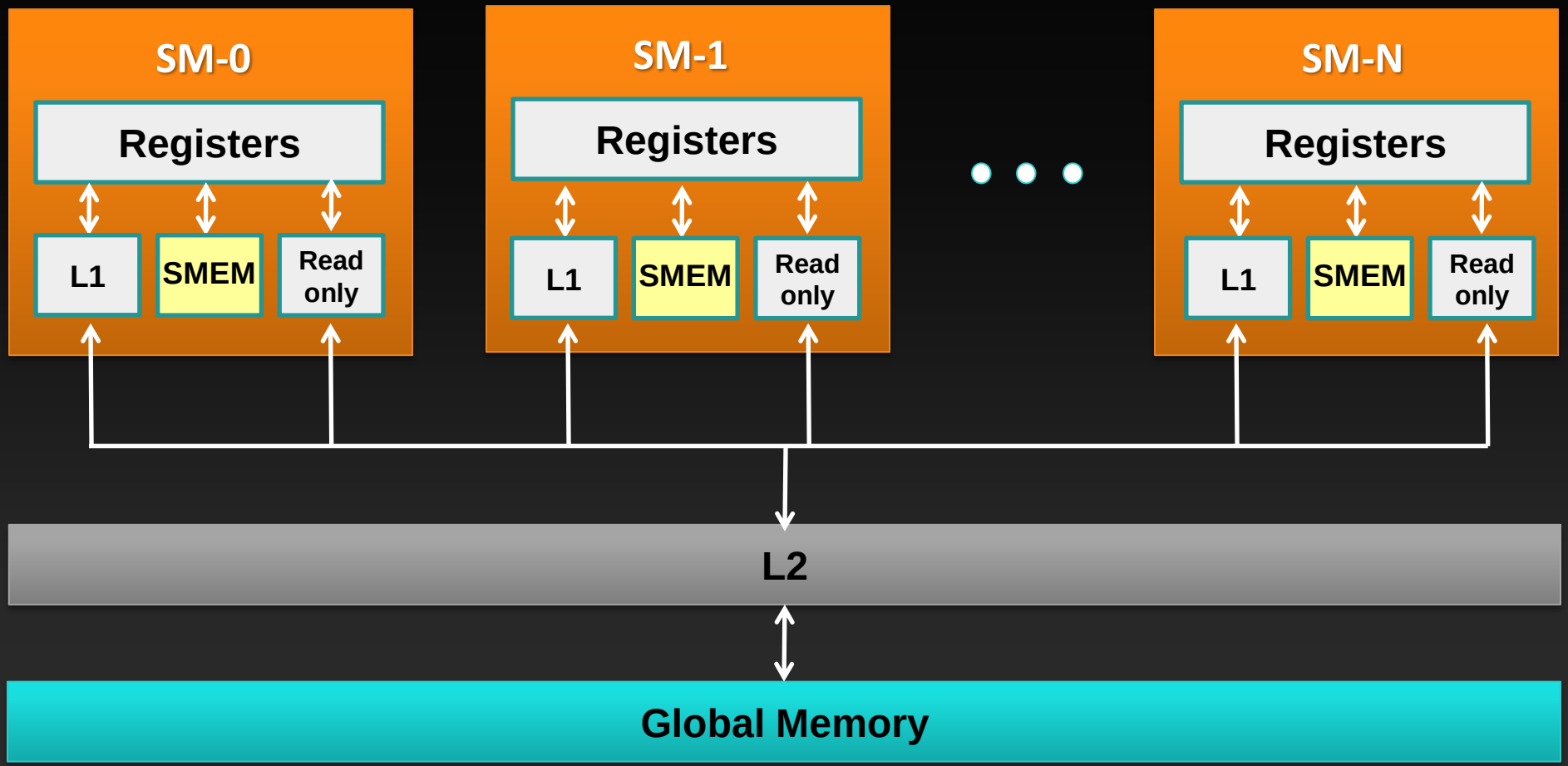
OPTIMIZE

Kernel Optimizations: *Global Memory Throughput*

CUDA Memory Architecture



Kepler Memory Hierarchy



Kepler Memory Hierarchy

- Registers
  - Storage local to each threads
  - Compiler-managed
- Shared memory / L1 cache
  - 64 KB, program-configurable into shared:L1
  - Program-managed
  - Accessible by all threads in the same thread block
  - Low latency, high bandwidth: ~2.5 TB/s
- Read-only cache
  - Up to 48 KB per Kepler SM
  - Hardware-managed (also used by texture units)
  - Used for read-only GMEM accesses (not coherent with writes)

Kepler Memory Hierarchy

- L2
  - 1.5 MB
  - Hardware-managed: all accesses to global memory go through L2, including CPU and peer GPU
- Global memory
  - 6 GB, accessible by all threads, host (CPU), other GPUs in the same system
  - Higher latency (400-800 cycles)
  - 250 GB/s

Load Operation

- Memory operations are issued per warp (32 threads)
  - Just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

Memory Throughput Analysis

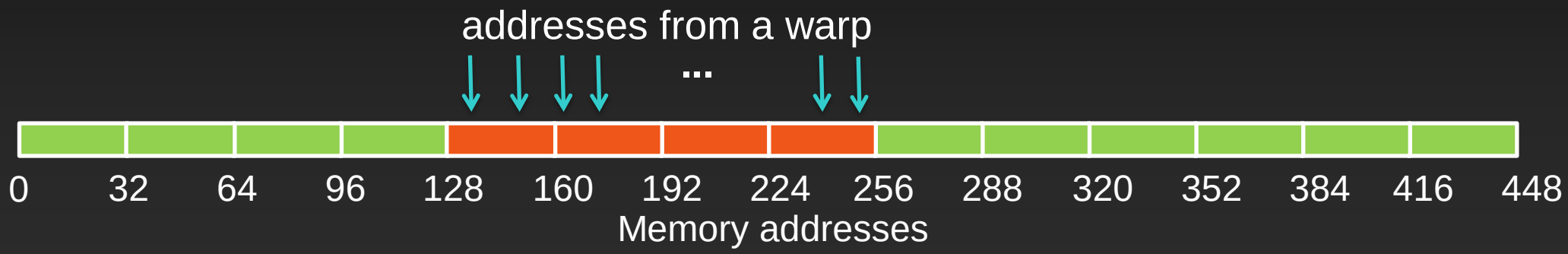
- Two perspectives on the throughput:
  - Application's point of view:
    - count only bytes requested by application
  - HW point of view:
    - count all bytes moved by hardware
- The two views can be different:
  - Memory is accessed at 32 byte granularity
    - Scattered/offset pattern: application doesn't use all the hw transaction bytes
  - Broadcast: the same small transaction serves many threads in a warp
- Two aspects to inspect for performance impact:
  - Address pattern
  - Number of concurrent accesses in flight

Global Memory Operation

- Memory operations are executed per warp
  - 32 threads in a warp provide memory addresses
  - Hardware determines into which lines those addresses fall
    - Memory transaction granularity is 32 bytes
    - There are benefits to a warp accessing a contiguous aligned region of 128 or 256 bytes
- Access word size
  - Natively supported sizes (per thread): 1, 2, 4, 8, 16 bytes
    - Assumes that each thread's address is aligned on the word size boundary
  - If you are accessing a data type that's of non-native size, compiler will generate several load or store instructions with native sizes

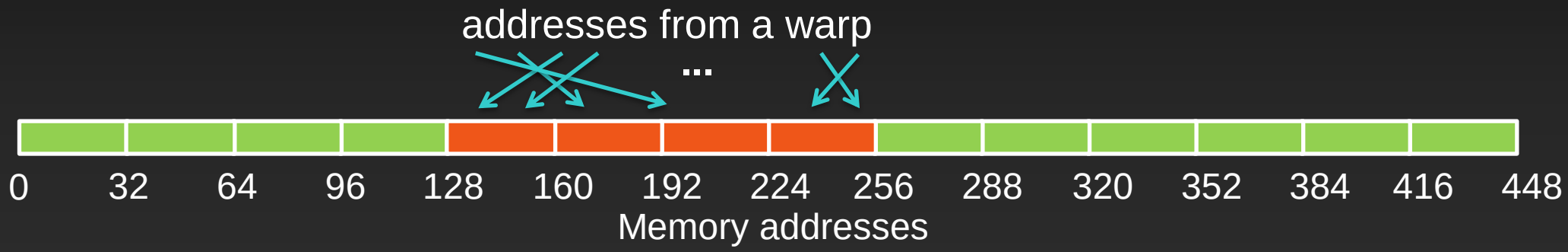
Access Patterns vs. Memory Throughput

- **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%**



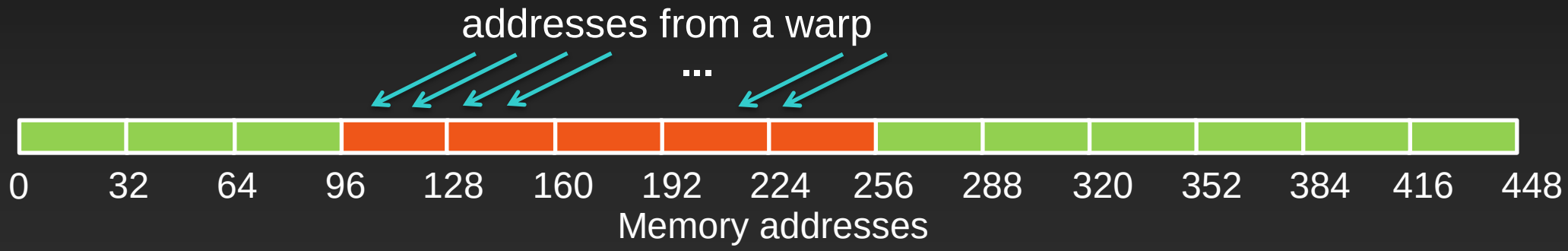
Access Patterns vs. Memory Throughput

- **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%**



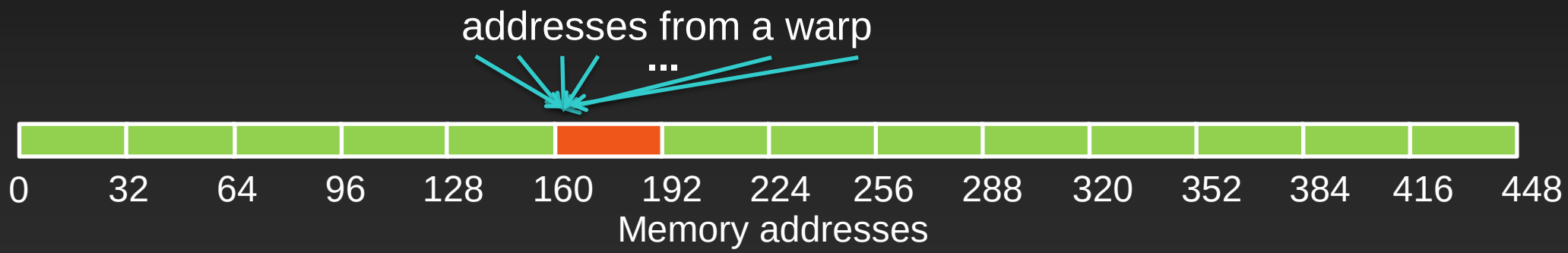
Access Patterns vs. Memory Throughput

- 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



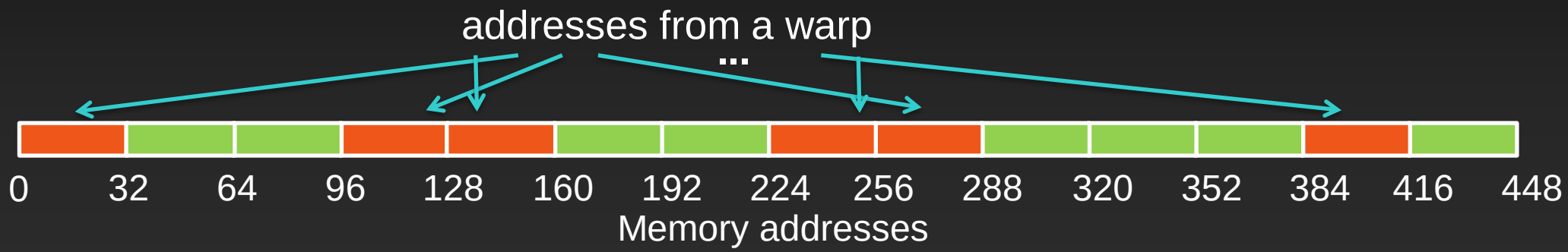
Access Patterns vs. Memory Throughput

- **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**32 bytes move across the bus**
  - **Bus utilization: 128 / (*N**32)**



Structures of Non-Native Size

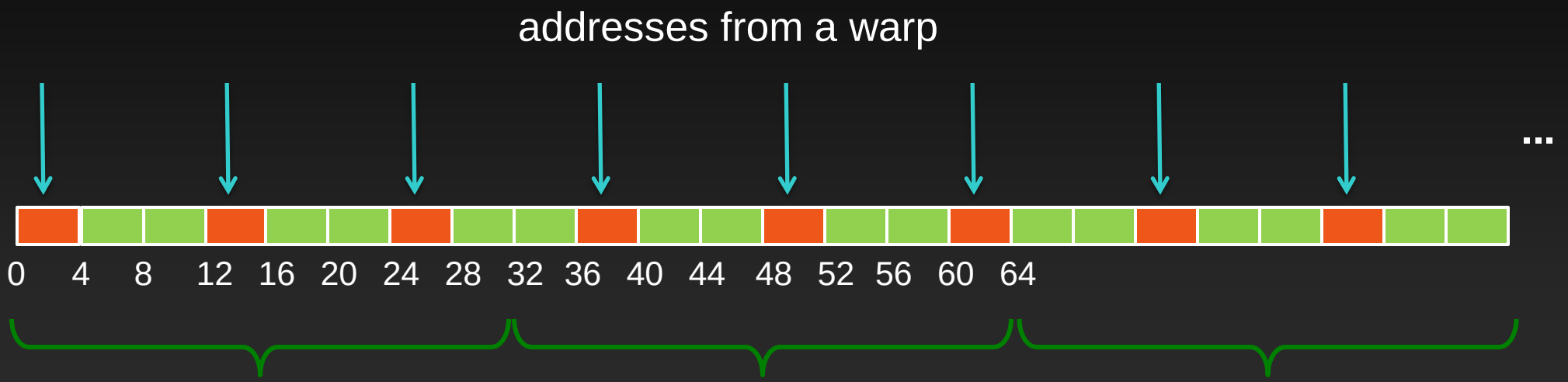
- Say we are reading a 12-byte structure per thread

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

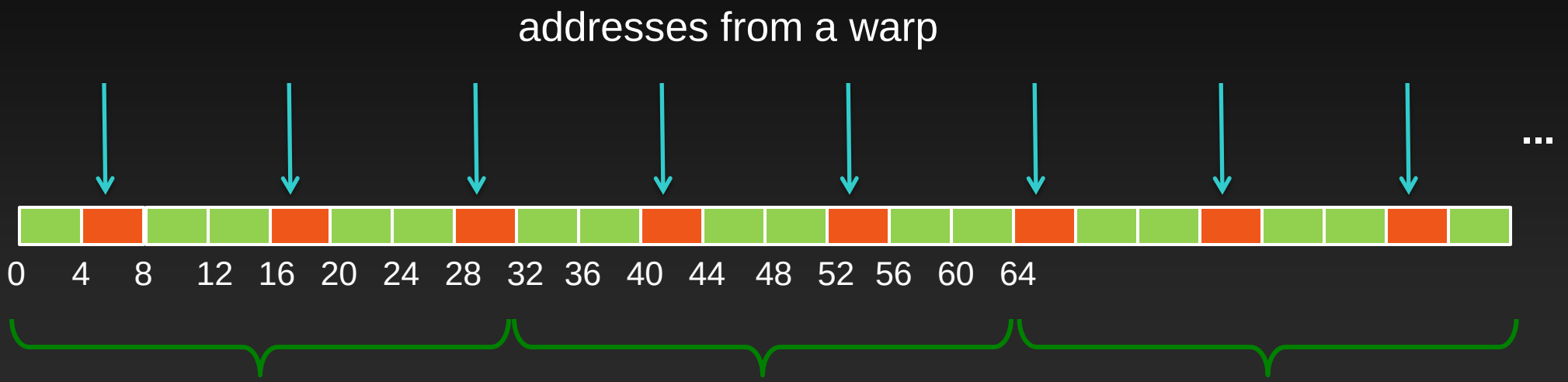
Structure of Non-Native Size

- 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

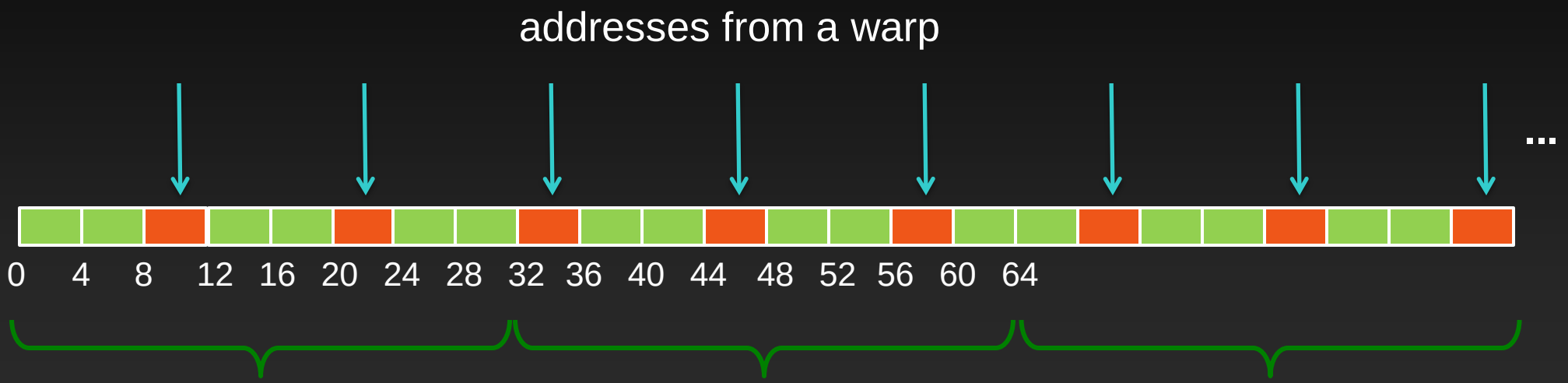
First Load Instruction



Second Load Instruction



Third Load Instruction

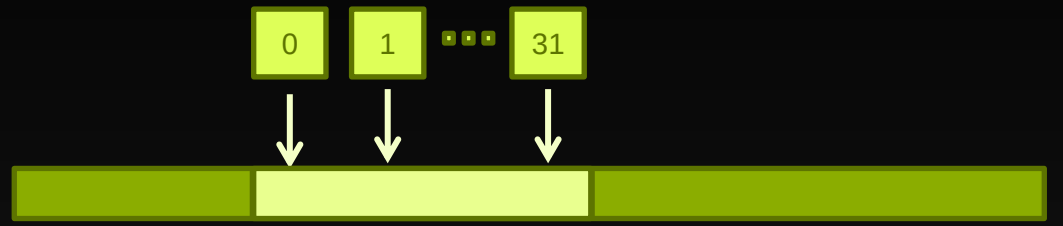


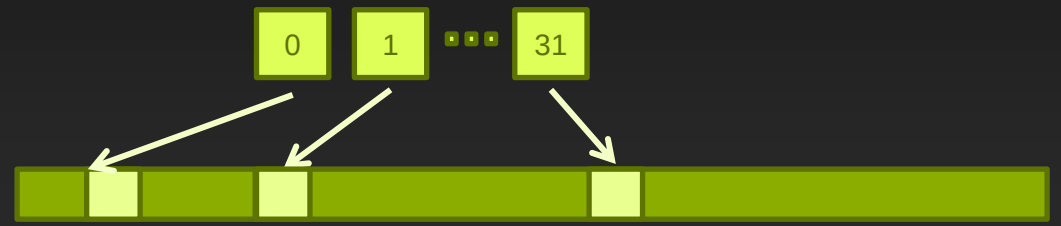
Performance and Solutions

- 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

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)]





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

Optimizing Access Concurrency

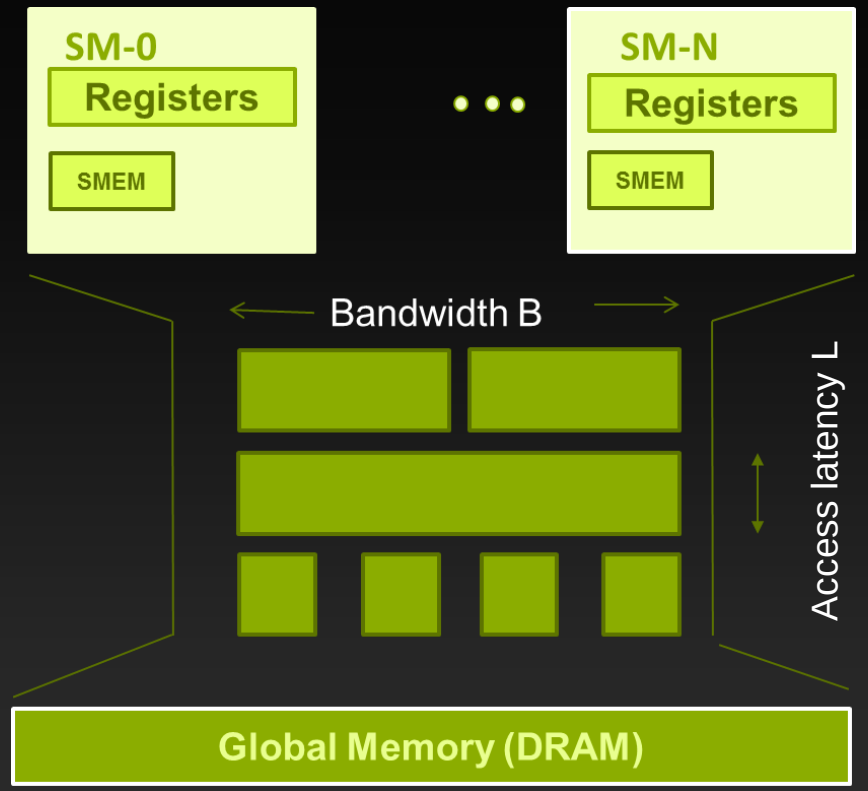
- Goal: utilize all available memory bandwidth
- Little's Law:

  # bytes delivered = latency * bandwidth

⇒ Increase parallelism (bytes delivered)

(or)

⇒ Reduce latency (time between requests)

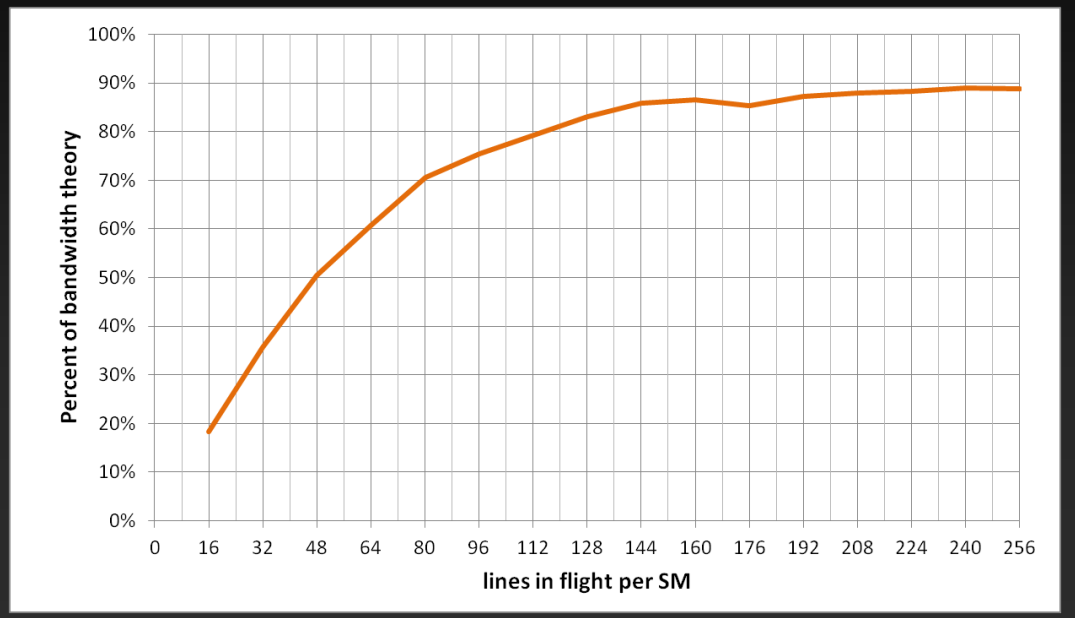


Exposing Sufficient Parallelism

- What memory system hardware ultimately needs:
  - Sufficient requests in flight to saturate bandwidth
- Two ways to increase parallelism:
  - More independent accesses within a thread (warp)
  - More concurrent threads (warps)

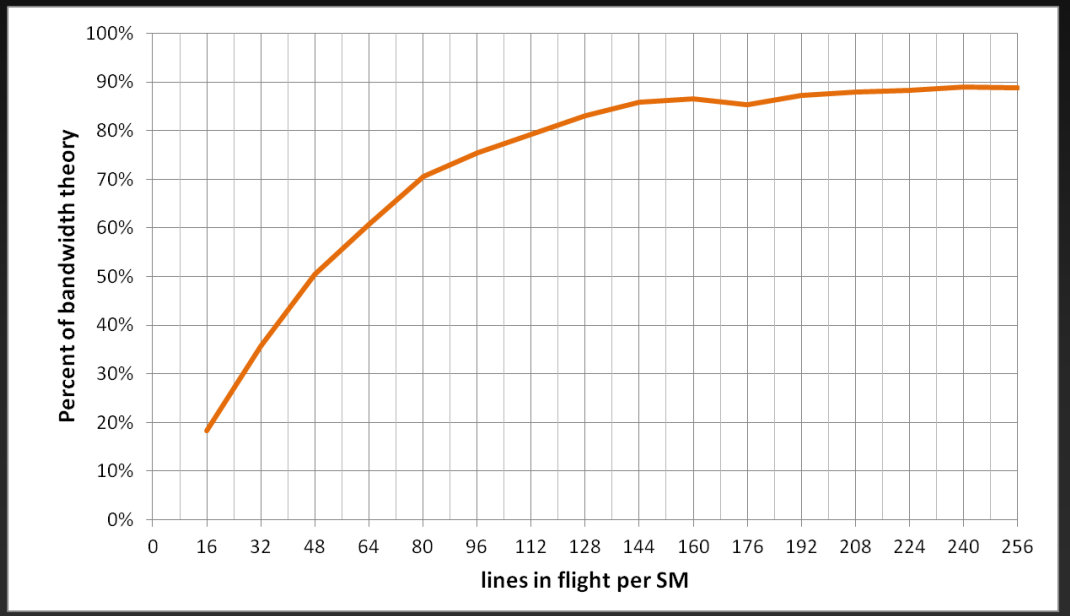
Memory-Level Parallelism = Bandwidth

- In order to saturate memory bandwidth, SM must issue enough independent memory requests concurrently



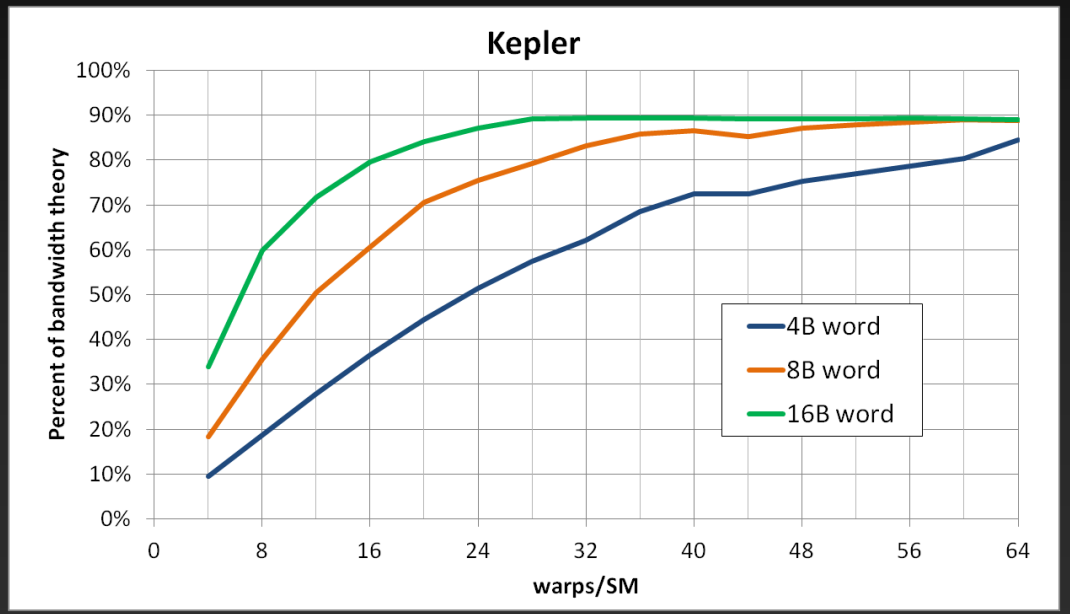
Memory-Level Parallelism: Requests in flight

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



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

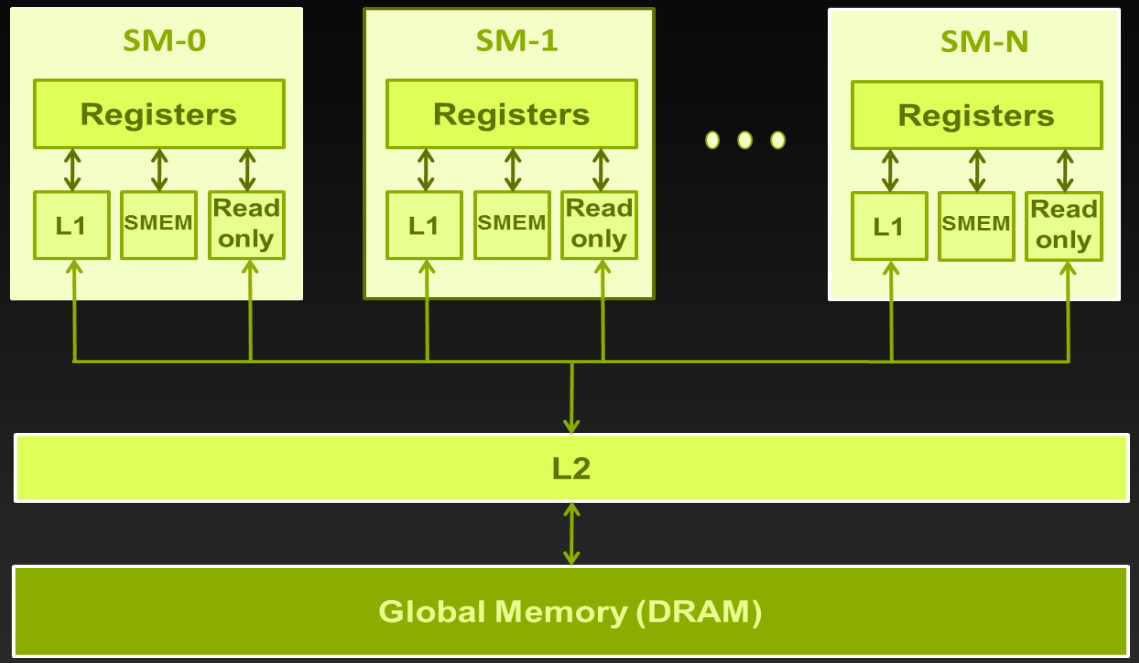
To achieve same throughput at lower occupancy or with smaller words, need more independent requests per warp

Optimizing Access Concurrency

- Have enough concurrent accesses to saturate the bus
  - Little's law: need (mem_latency)x(bandwidth) bytes
- Ways to increase concurrent accesses:
  - Increase occupancy (run more warps concurrently)
    - Adjust thread block dimensions
      - To maximize occupancy at given register and smem requirements
    - If occupancy is limited by registers per thread:
      - 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

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



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

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];
}
```

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] );
}
```

Blocking for L1, Read-only, L2 Caches

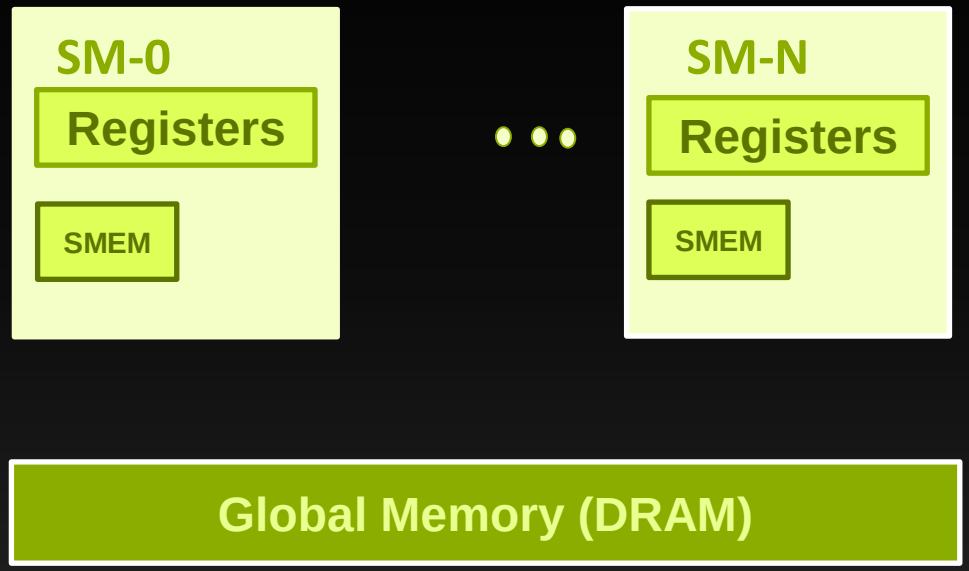
- Short answer: DON'T
- GPU caches are not intended for the same use as CPU caches
  - Smaller size (especially per thread), so not aimed at temporal reuse
  - Intended to smooth out some access patterns, help with spilled registers, etc.
- Usually not worth trying to cache-block like you would on CPU
  - 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

OPTIMIZE

Kernel Optimizations: *Shared Memory Accesses*

Shared Memory

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion



Shared Memory/L1 Sizing

- Shared memory and L1 use the same 64KB
  - 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

Shared Memory

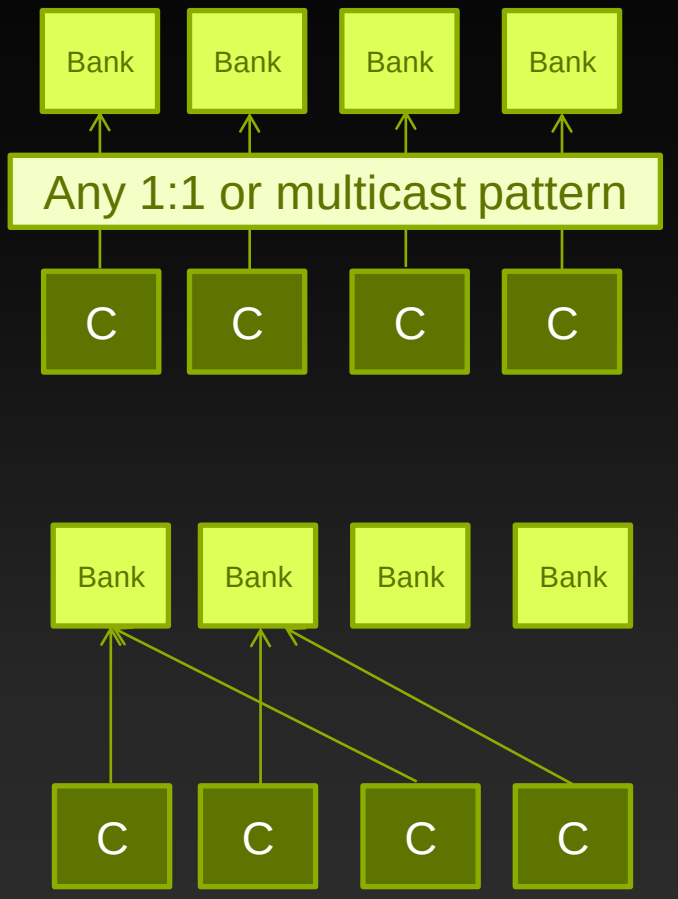
- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Organization:
  - 32 banks, 4-byte (or 8-byte) banks
  - Successive words accessed through different banks

Shared Memory

- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Performance:
  - smem accesses are issued per warp
  - Throughput is 4 (or 8) bytes per bank per clock per multiprocessor
  - serialization: if *N* threads of 32 access different words in the same bank, *N* accesses are executed serially
  - multicast: *N* threads access the same word in one fetch
    - Could be different bytes within the same word

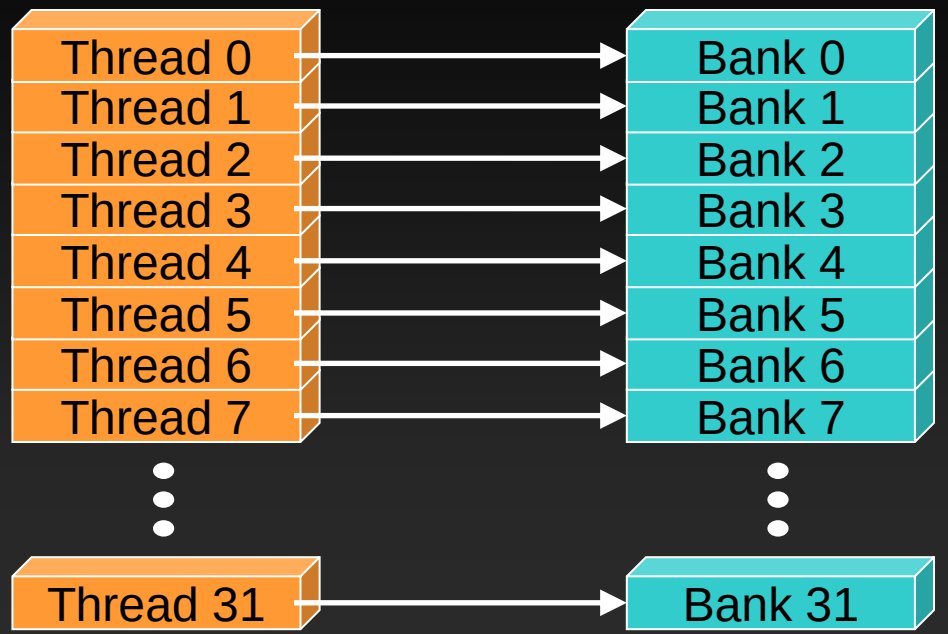
Shared Memory Organization

- 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

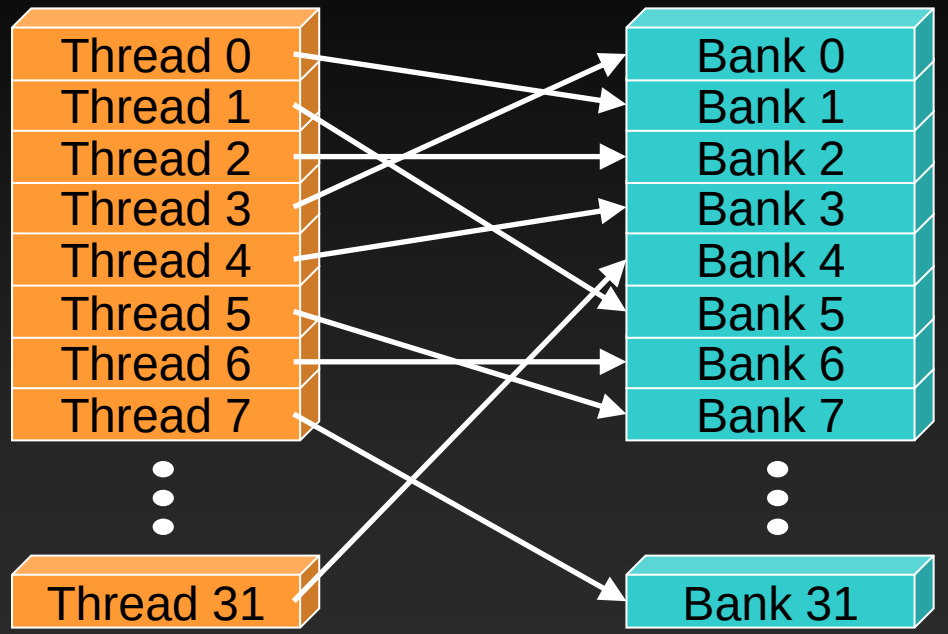


Bank Addressing Examples

- No Bank Conflicts

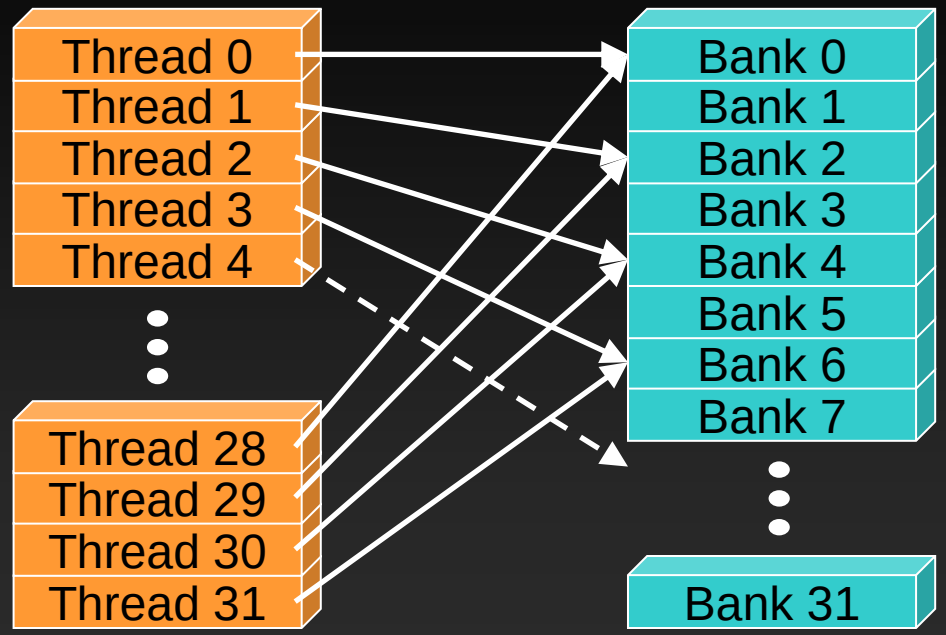


- No Bank Conflicts

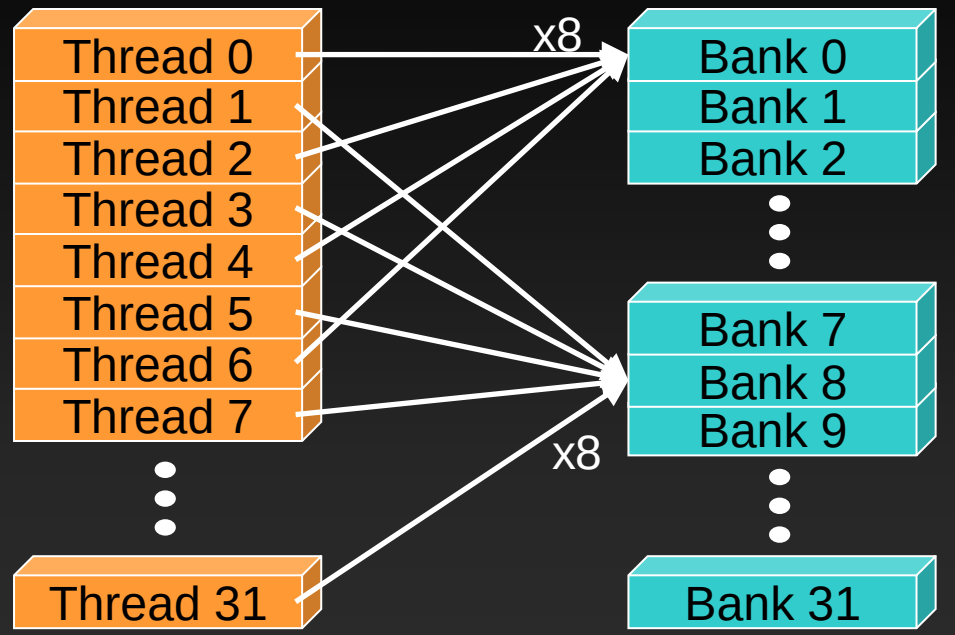


Bank Addressing Examples

- 2-way Bank Conflicts



- 8-way Bank Conflicts

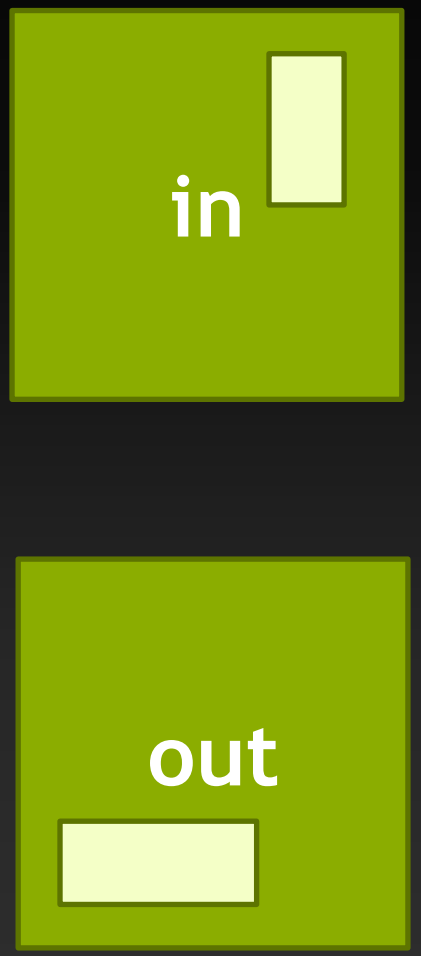


Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)

⇒ Process matrix tile, not single row/column, per block

⇒ Transpose matrix tile within block



Case Study: Matrix Transpose

- Coalesced read
- Scattered write (stride N)

- Transpose matrix tile within block

⇒ **Need threads in a block to cooperate: use shared memory**

Transpose with coalesced read/write

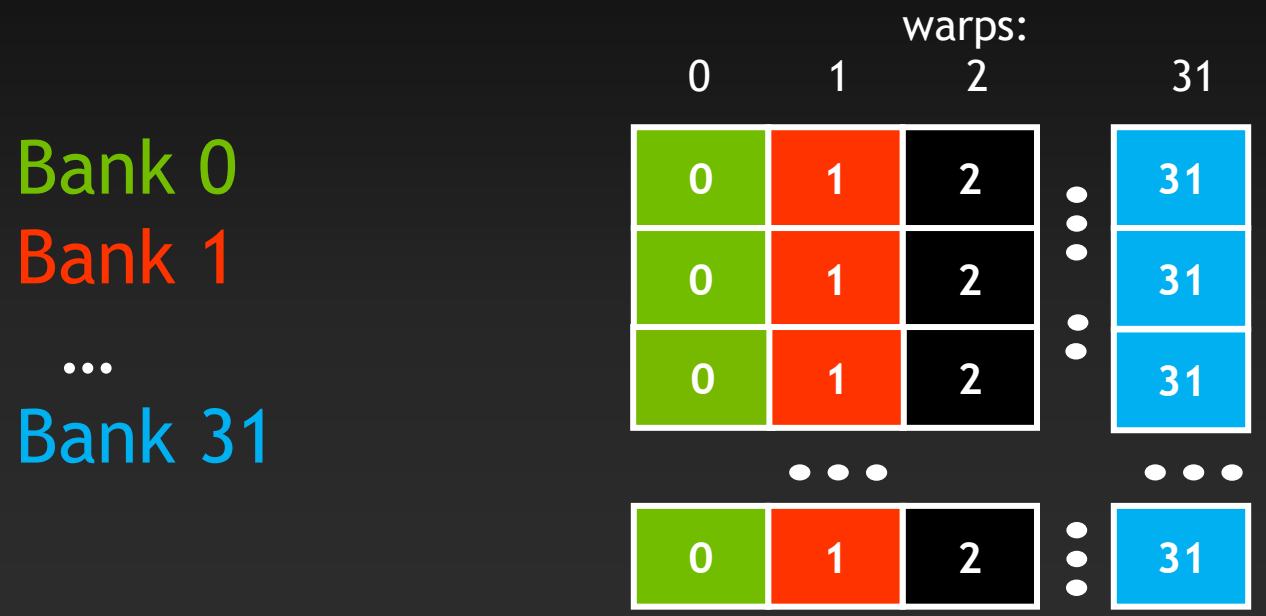
```
__global__ transpose(float in[], float out[])
{

  __shared__ float tile[TILE][TILE];

  int glob_in = xIndex + (yIndex)*N;
  int glob_out = xIndex + (yIndex)*N;

  tile[threadIdx.y][threadIdx.x] = in[glob_in];

  __syncthreads();

  out[glob_out] = tile[threadIdx.x][threadIdx.y];

}
```

Fixed GMEM coalescing, but introduced SMEM bank conflicts

```
transpose<<<grid, threads>>>(in, out);
```

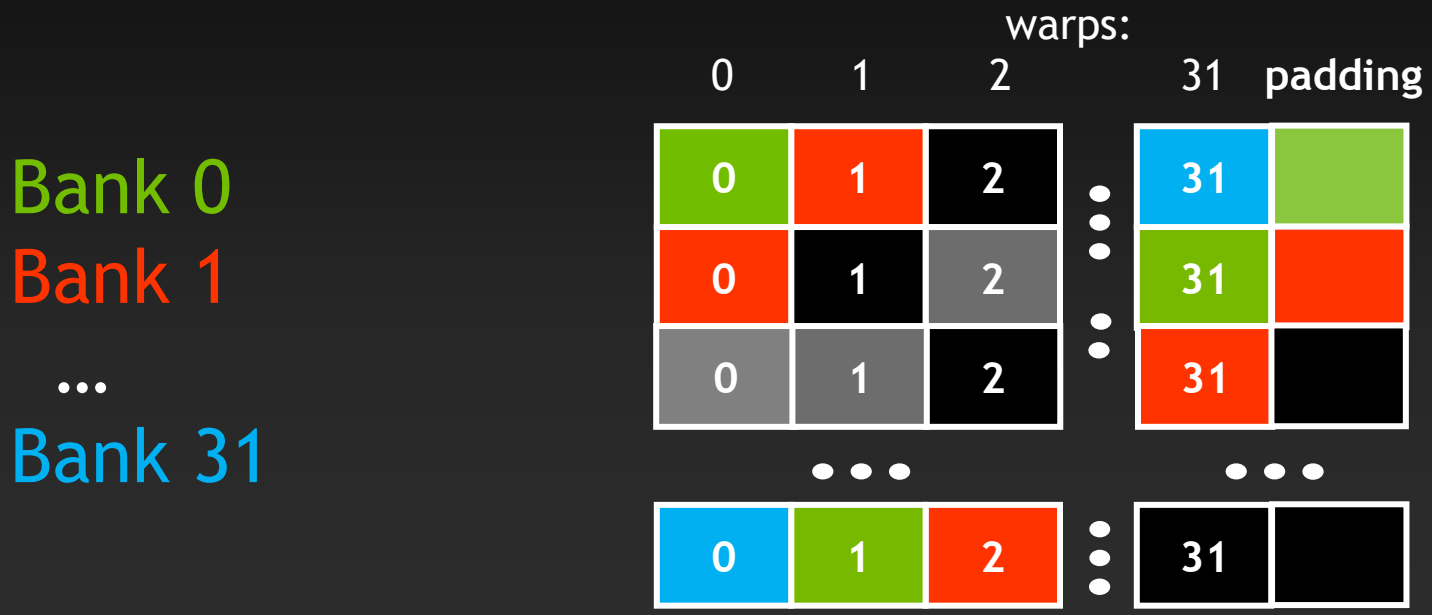
Shared Memory: Avoiding Bank Conflicts

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



Shared Memory: Avoiding Bank Conflicts

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



OPTIMIZE

Kernel Optimizations: *Instruction Throughput / Control Flow*

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)

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)

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)

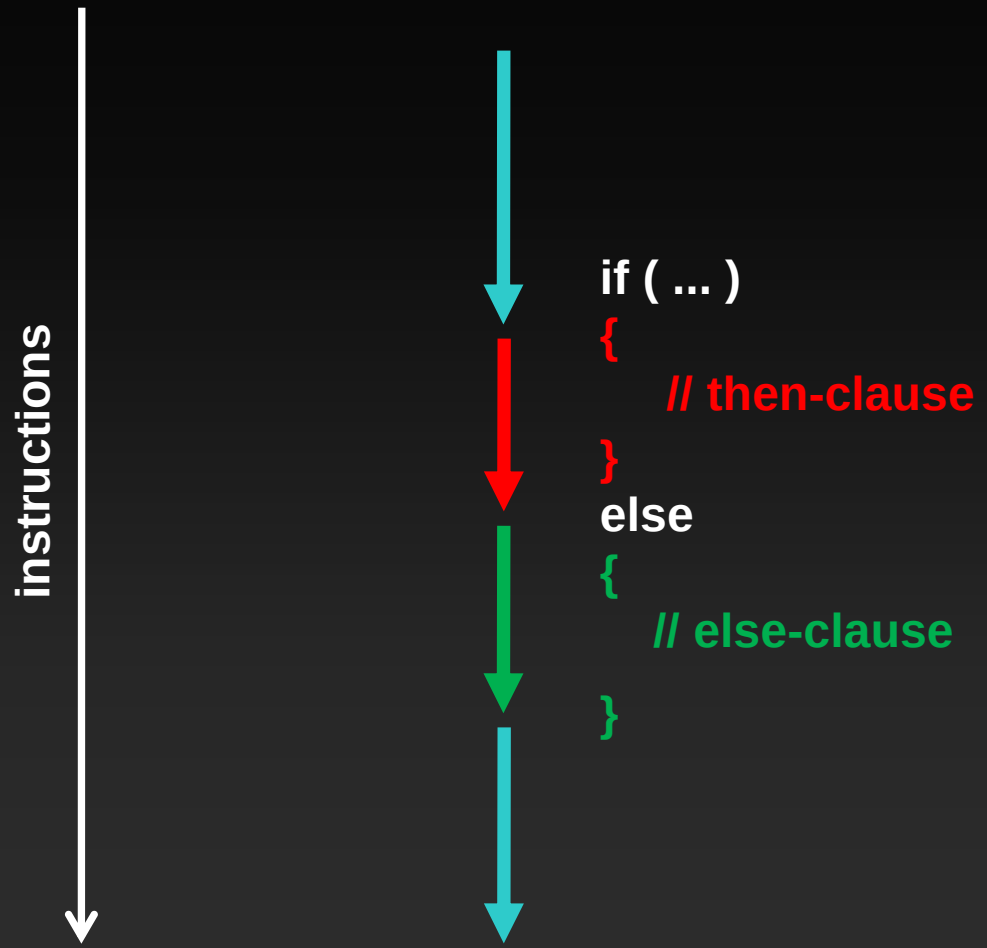
Control Flow

- 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

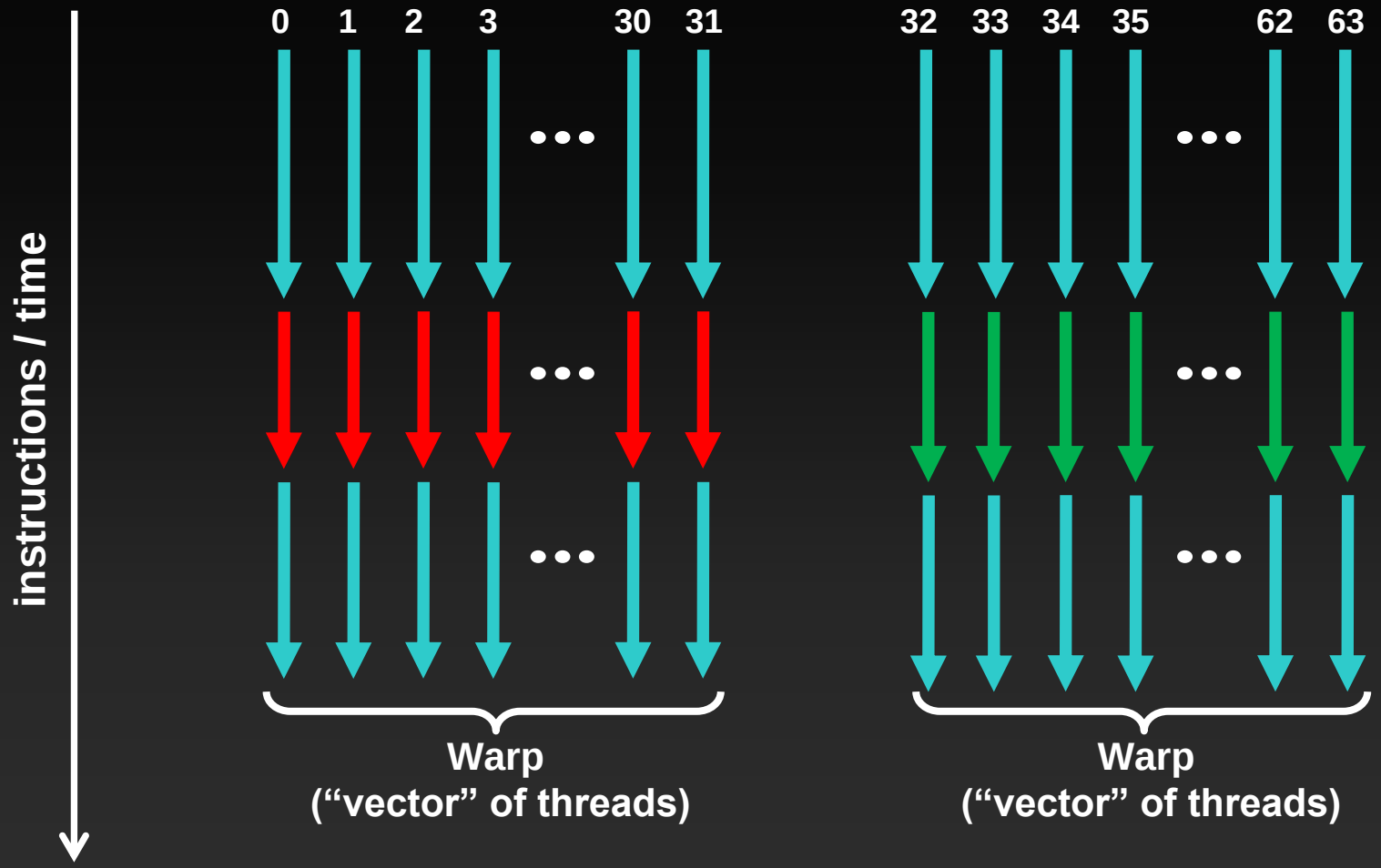
Control Flow

- Avoid diverging within a warp
- 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

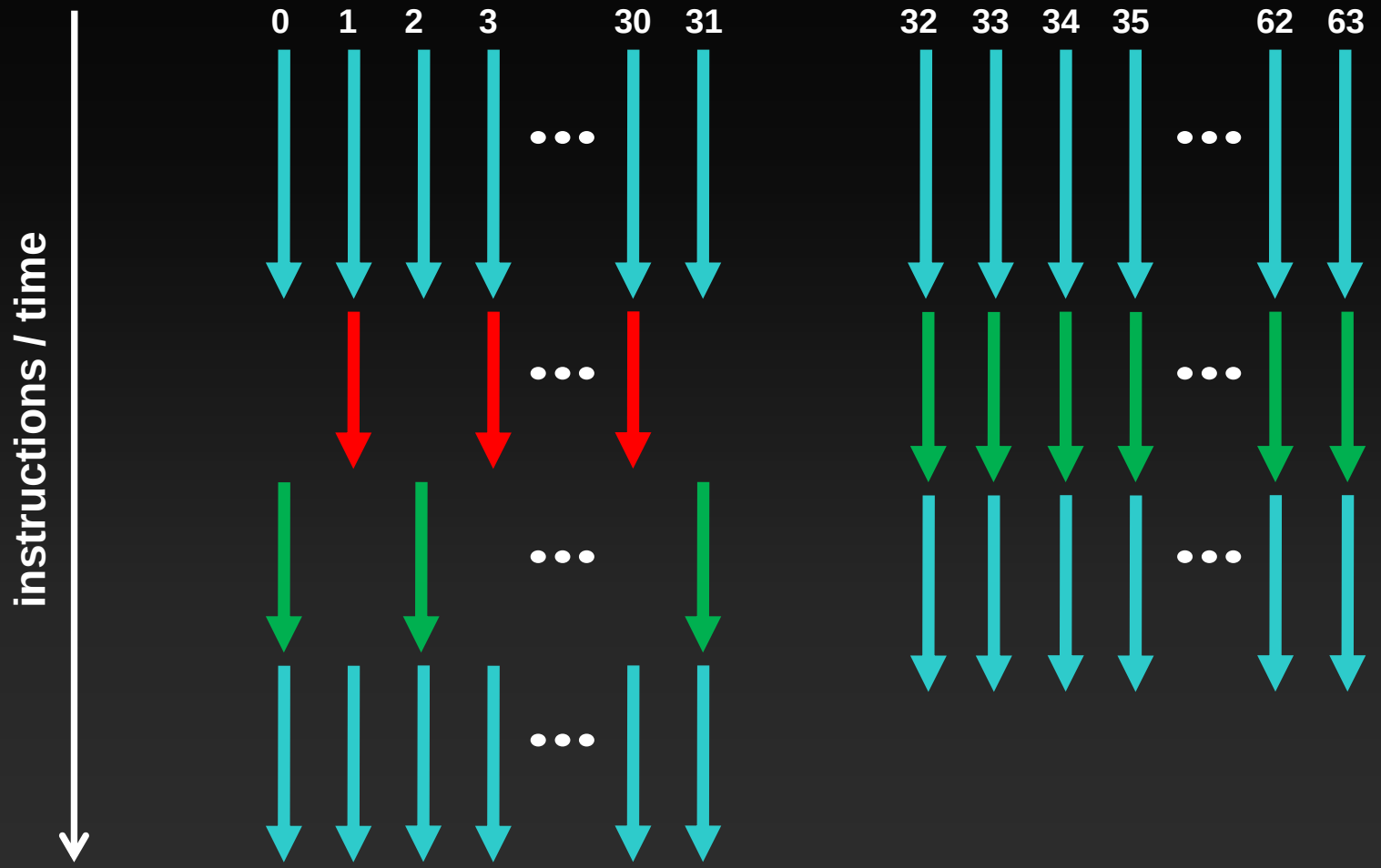
Control Flow



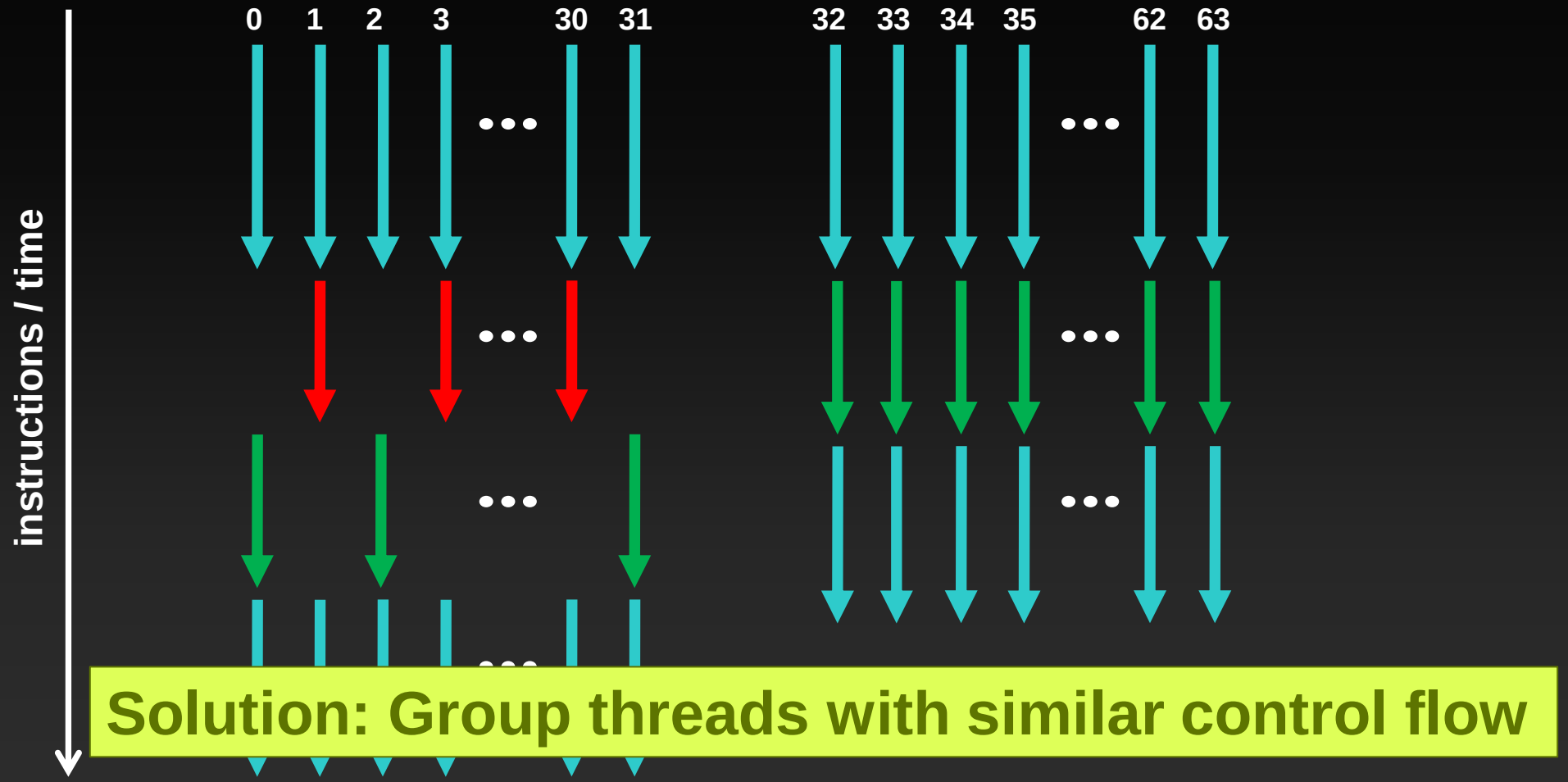
Execution within warps is coherent



Execution diverges within a warp



Execution diverges within a warp



Solution: Group threads with similar control flow

OPTIMIZE

Optimizing CPU-GPU Interaction: *Maximizing PCIe Throughput*

Maximizing PCIe Throughput

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

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

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

OPTIMIZE

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

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

Call Sequencing for Optimal Overlap

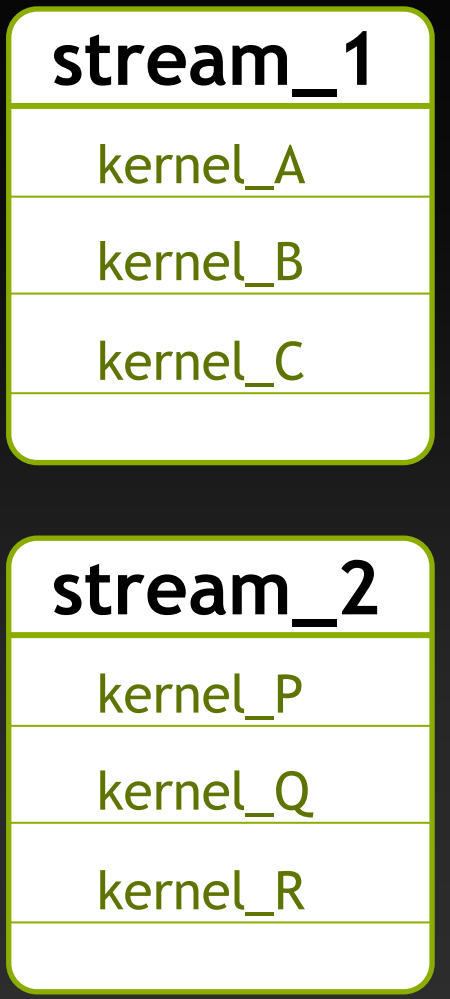
- CUDA calls are dispatched in the sequence they were issued
- Kepler can concurrently execute:
  - Up to 32 kernels
  - Up to 2 memcopies, as long as they are in different directions (D2H, H2D)
- A call is dispatched if both are true:
  - Resources are available
  - Preceding calls in the same stream 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

Hyper-Q Enables Efficient Scheduling

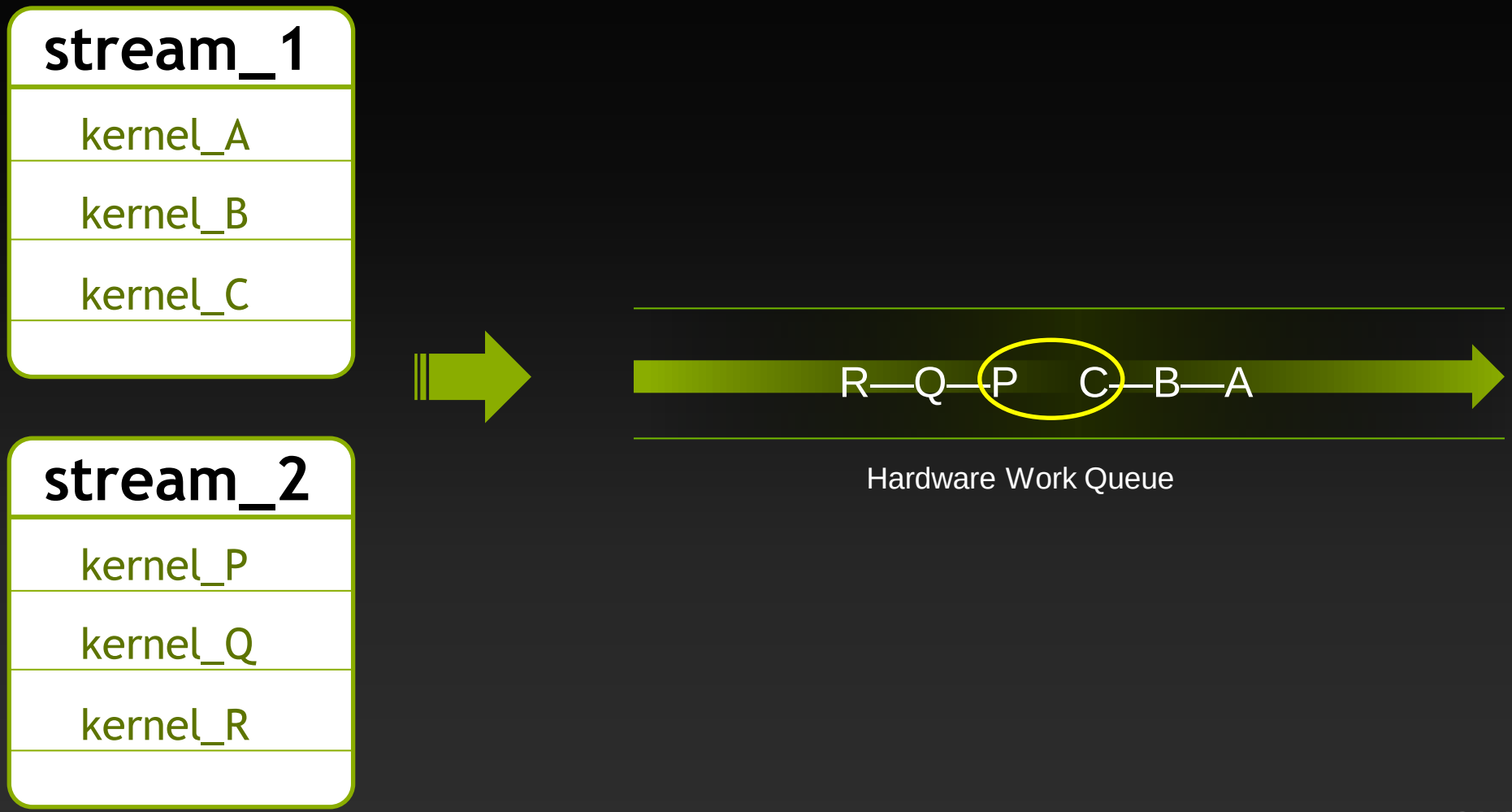
- Grid Management Unit selects most appropriate task from up to 32 hardware queues (CUDA streams)
- Improves scheduling of concurrently executed grids
- Particularly interesting for MPI applications when combined with CUDA Proxy (though not limited to MPI applications)

Stream Dependencies Example

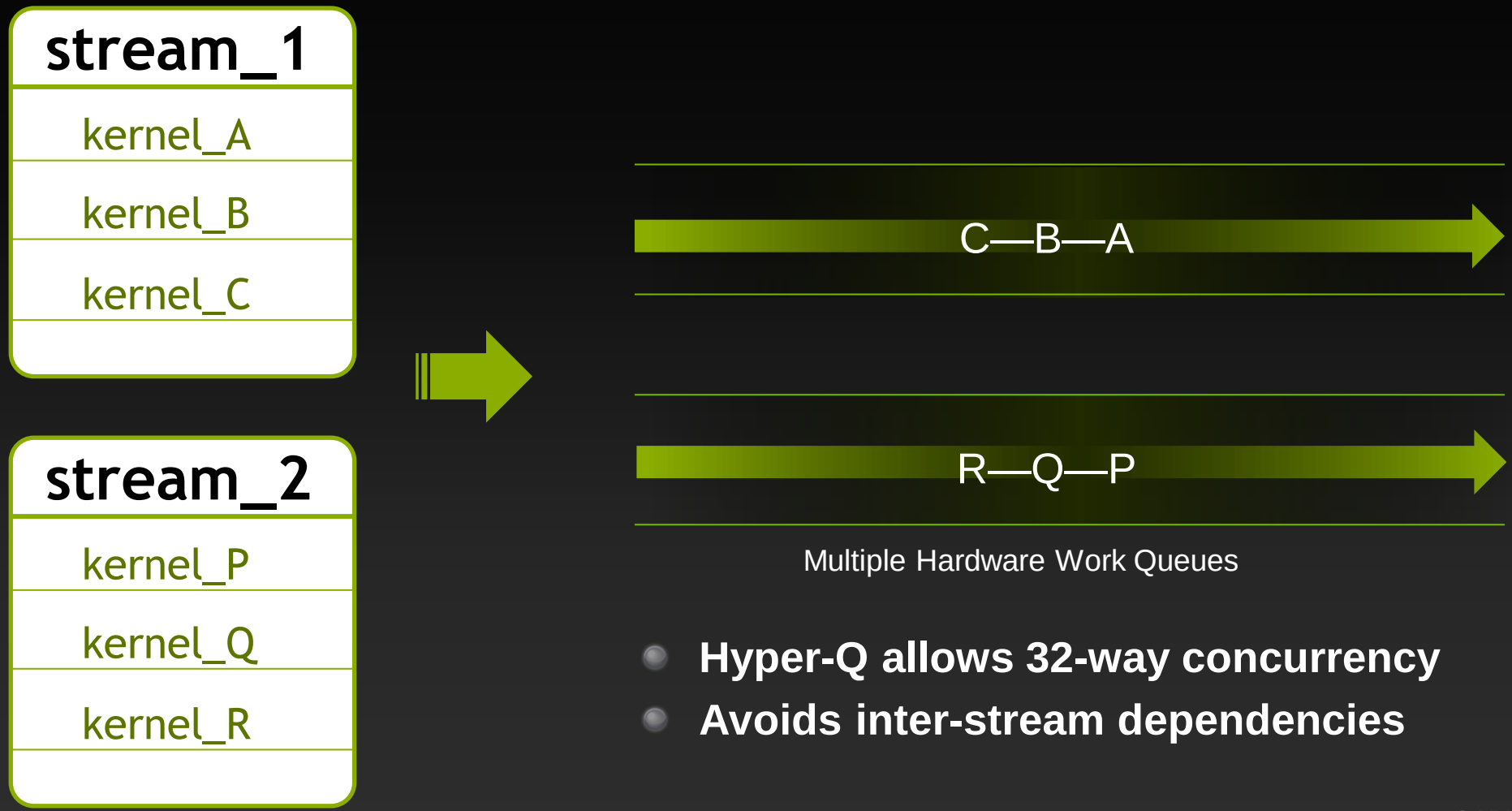
```
void foo(void)
{
    kernel_A<<<g,b,s, stream_1>>>();
    kernel_B<<<g,b,s, stream_1>>>();
    kernel_C<<<g,b,s, stream_1>>>();
}

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



Stream Dependencies without Hyper-Q

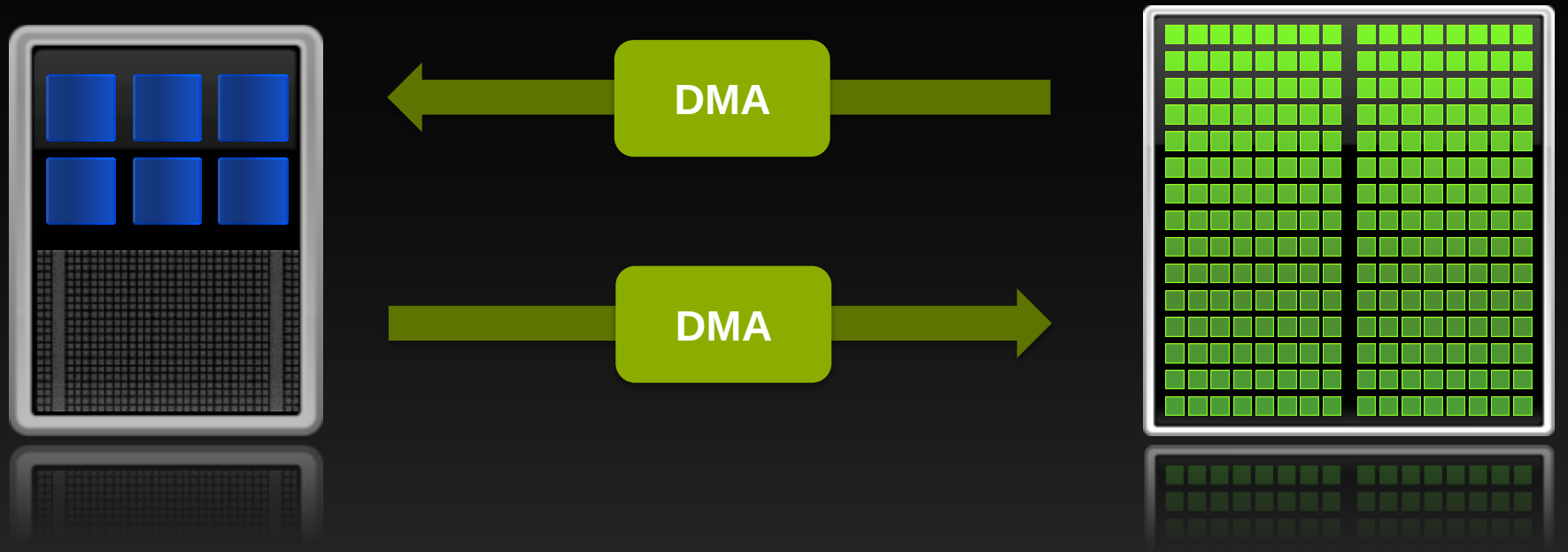


Stream Dependencies with Hyper-Q



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

Hyper-Q Example: Building a Pipeline

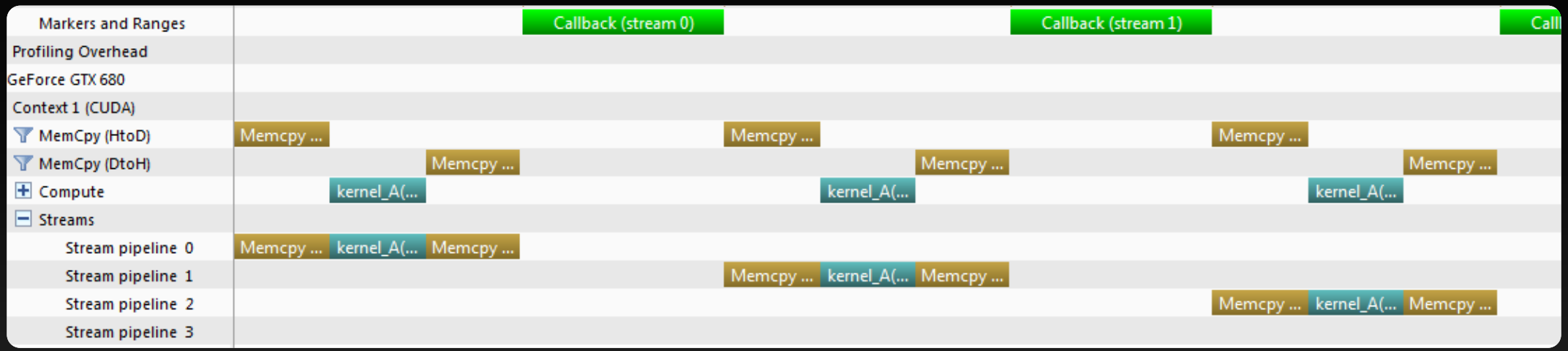


- Heterogeneous system: overlap work and data movement
- Kepler + CUDA 5: Hyper-Q and CPU Callbacks

Pipeline Code

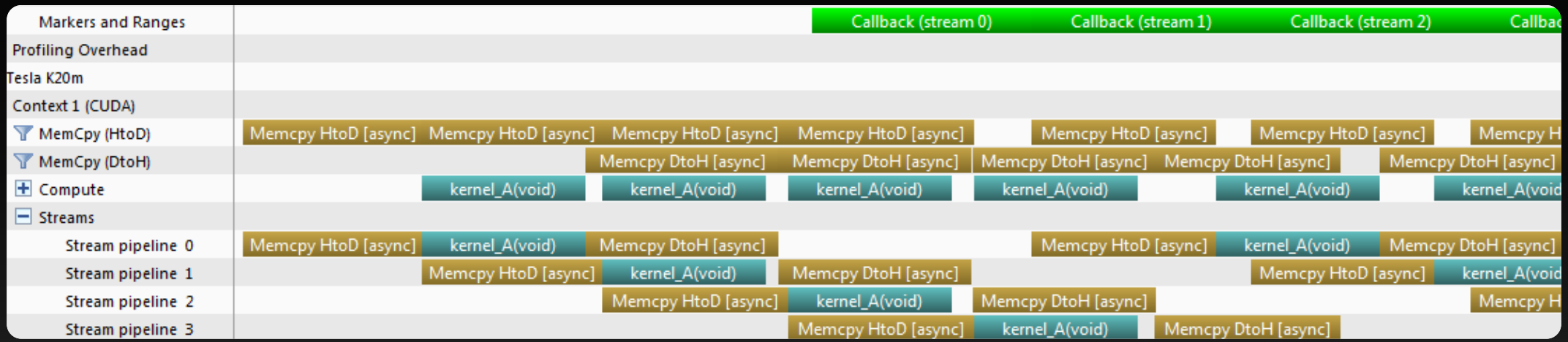
```
for (unsigned int i = 0 ; i < nIterations ; ++i)
{
    // Copy data from host to device
    cudaMemcpyAsync(d_data, h_data, cpybytes, cudaMemcpyHostToDevice,
                    *r_streams.active());

    // 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();
}
```

Pipeline Without Hyper-Q



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

Pipeline With Hyper-Q



- Full overlap of all engines
- Simple to program

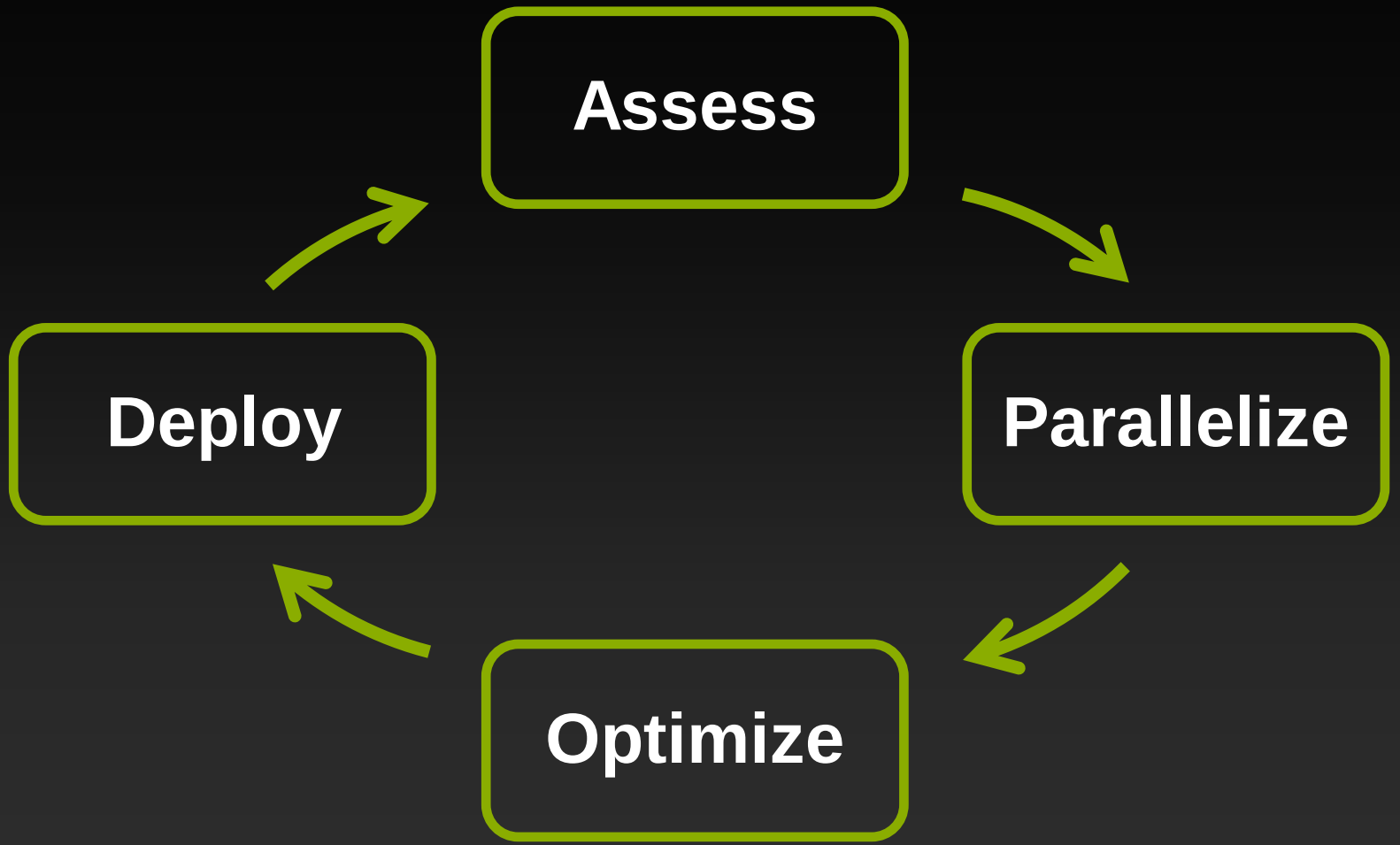
Hyper-Q also enables CUDA Proxy

- No application modifications necessary
  - Start proxy daemon by setting `CRAY_CUDA_PROXY=1` in batch script
  - 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*

But Hyper-Q != CUDA Proxy

- One process: No proxy 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 Proxy
  - Leverages task-level parallelism across processes (e.g., MPI ranks)
  - MPI is not required for proxy – it's just the common case for HPC

APOD: A Systematic Path to Performance



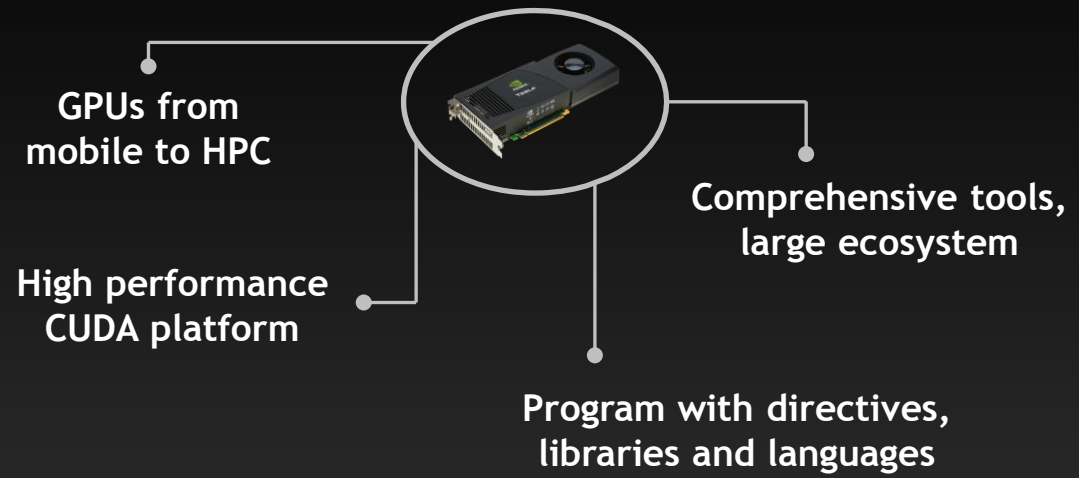
Additional Information

nvidia.com/cuda

nvidia.com/kepler

docs.nvidia.com/cuda

gputechconf.com



Additional Information: GTC

- Kepler architecture:
  - GTC 2012 S0642: Inside Kepler
- Assessing performance limiters:
  - GTC 2012 S0514: GPU Performance Analysis and Optimization
- Profiling tools:
  - GTC 2012 S0419: CUDA Performance Tools
  - GTC 2012 S0420: Nsight IDE for Linux and Mac
- GPU computing webinars:
  - developer.nvidia.com/gpu-computing-webinars

http://www.gputechconf.com/gtcnew/on-demand-gtc.php