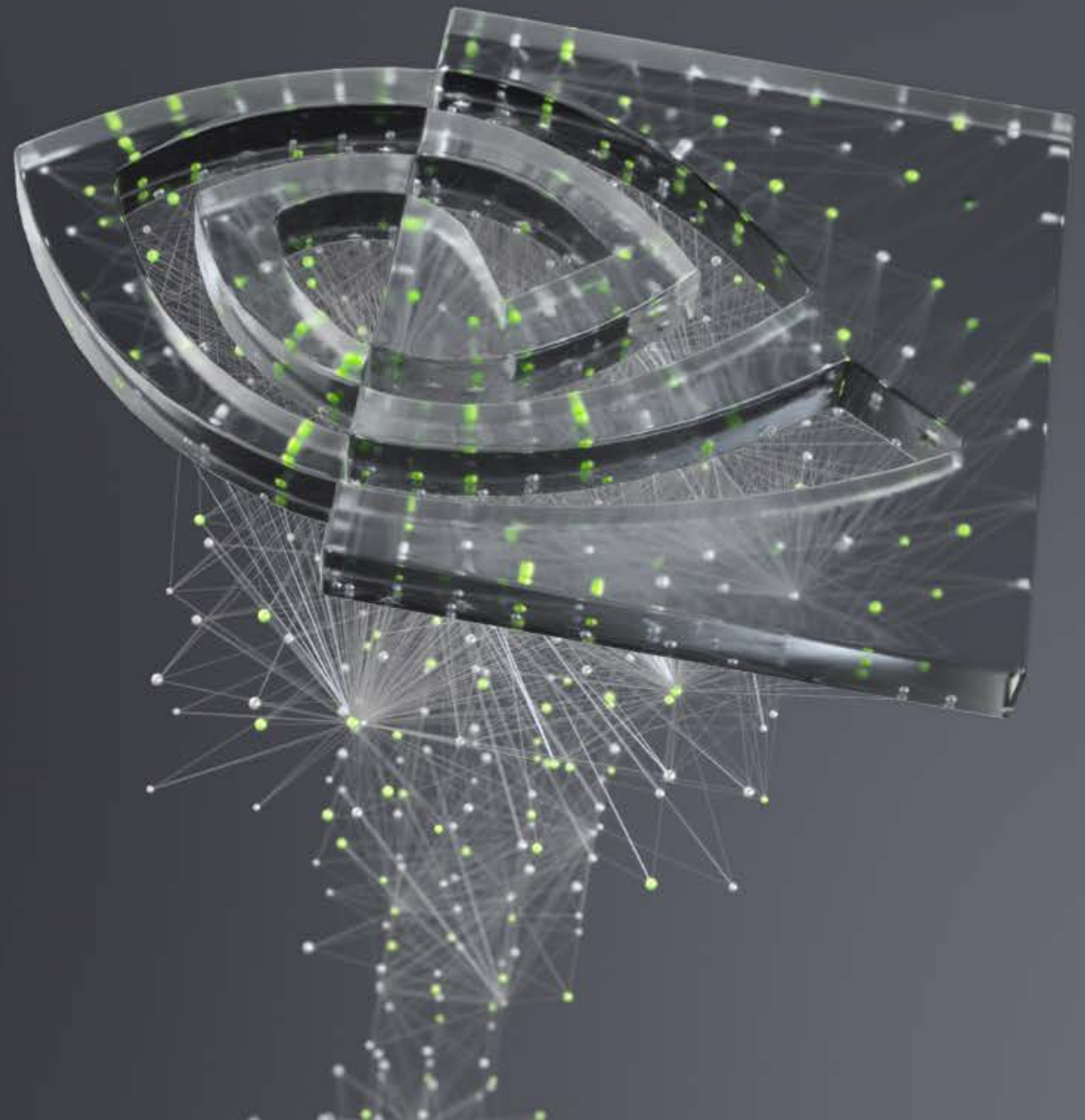
Table of Contents

https://trent.me/oak-ridge-nvidia-cuda-training-series



CUDA C++ BASICS

NVIDIA Corporation

WHAT IS CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Expose/Enable performance
- CUDA C++
  - Based on industry-standard C++
  - Set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C++
  - Other languages/bindings available: Fortran, Python, Matlab, etc.

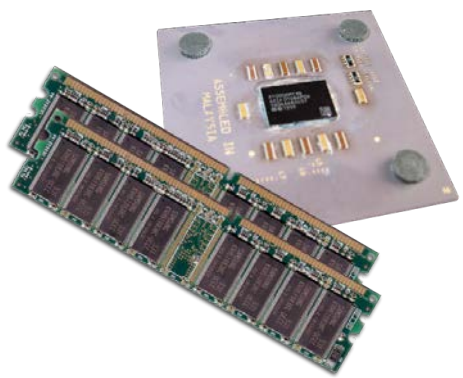
INTRODUCTION TO CUDA C++

- What will you learn in this session?
  - Start with vector addition
  - Write and launch CUDA C++ kernels
  - Manage GPU memory
  - (Manage communication and synchronization)-> next session

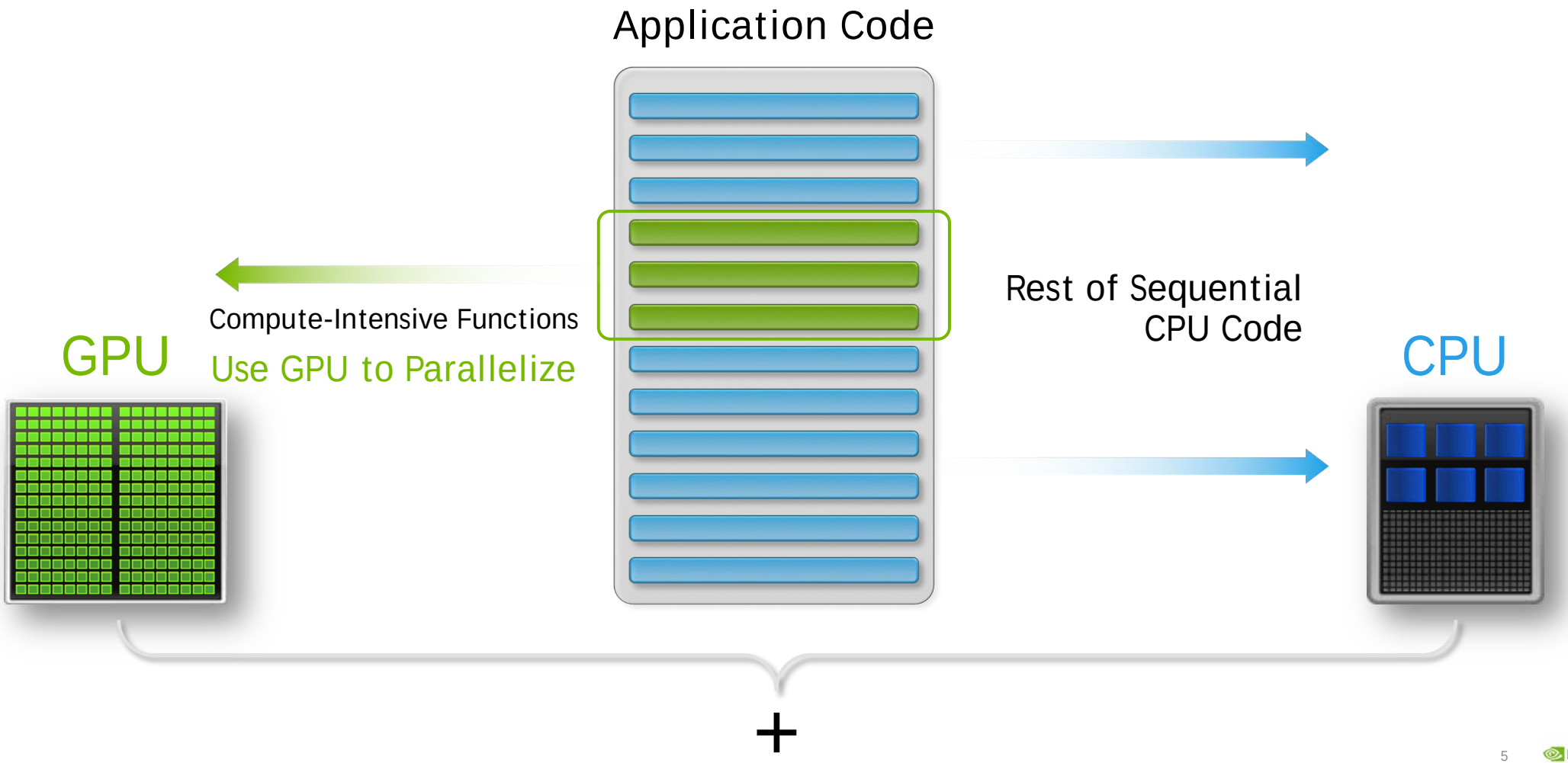
- (Some knowledge of C or C++ programming is assumed.)

HETEROGENEOUS COMPUTING

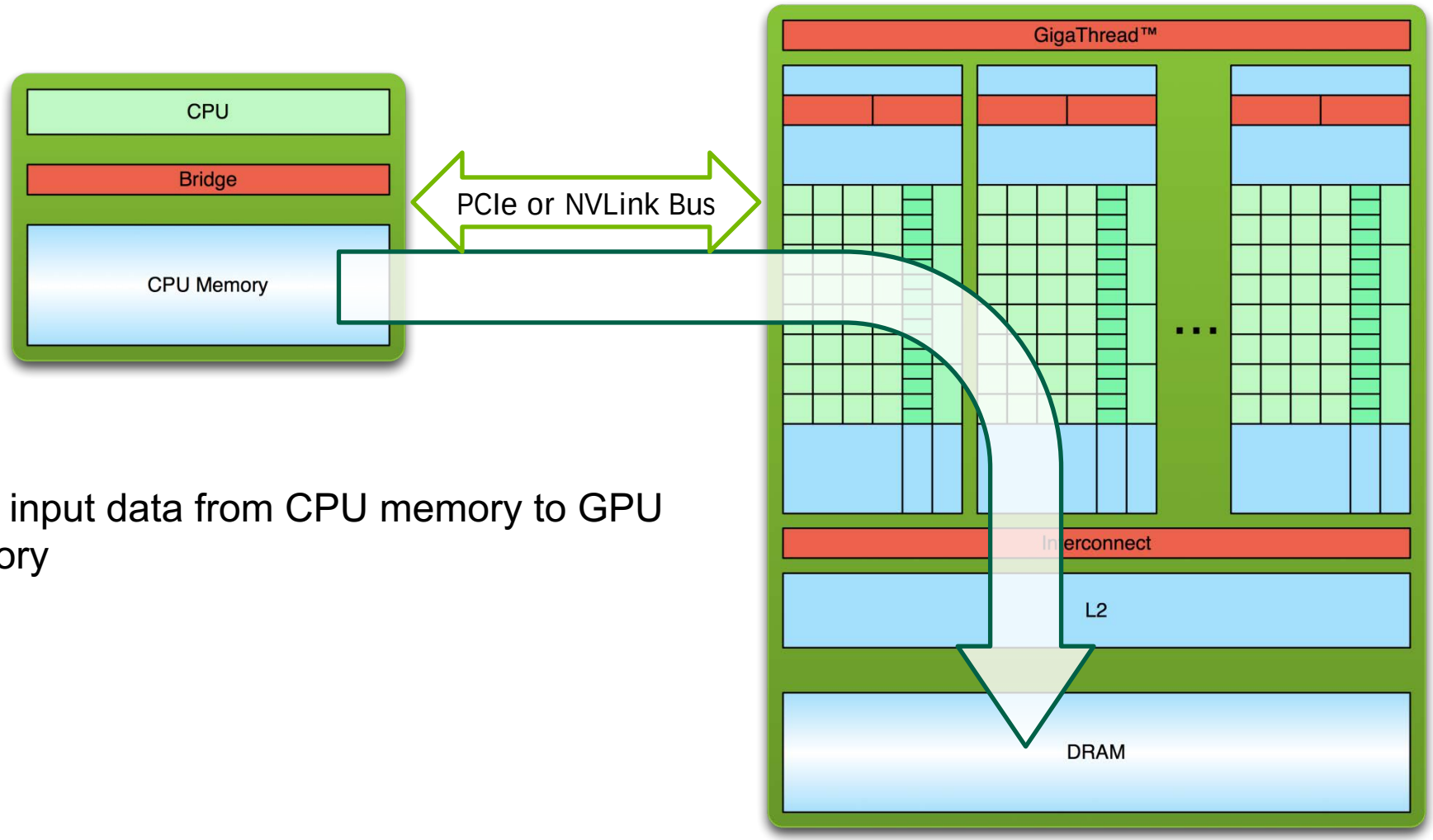
- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)



PORTING TO CUDA

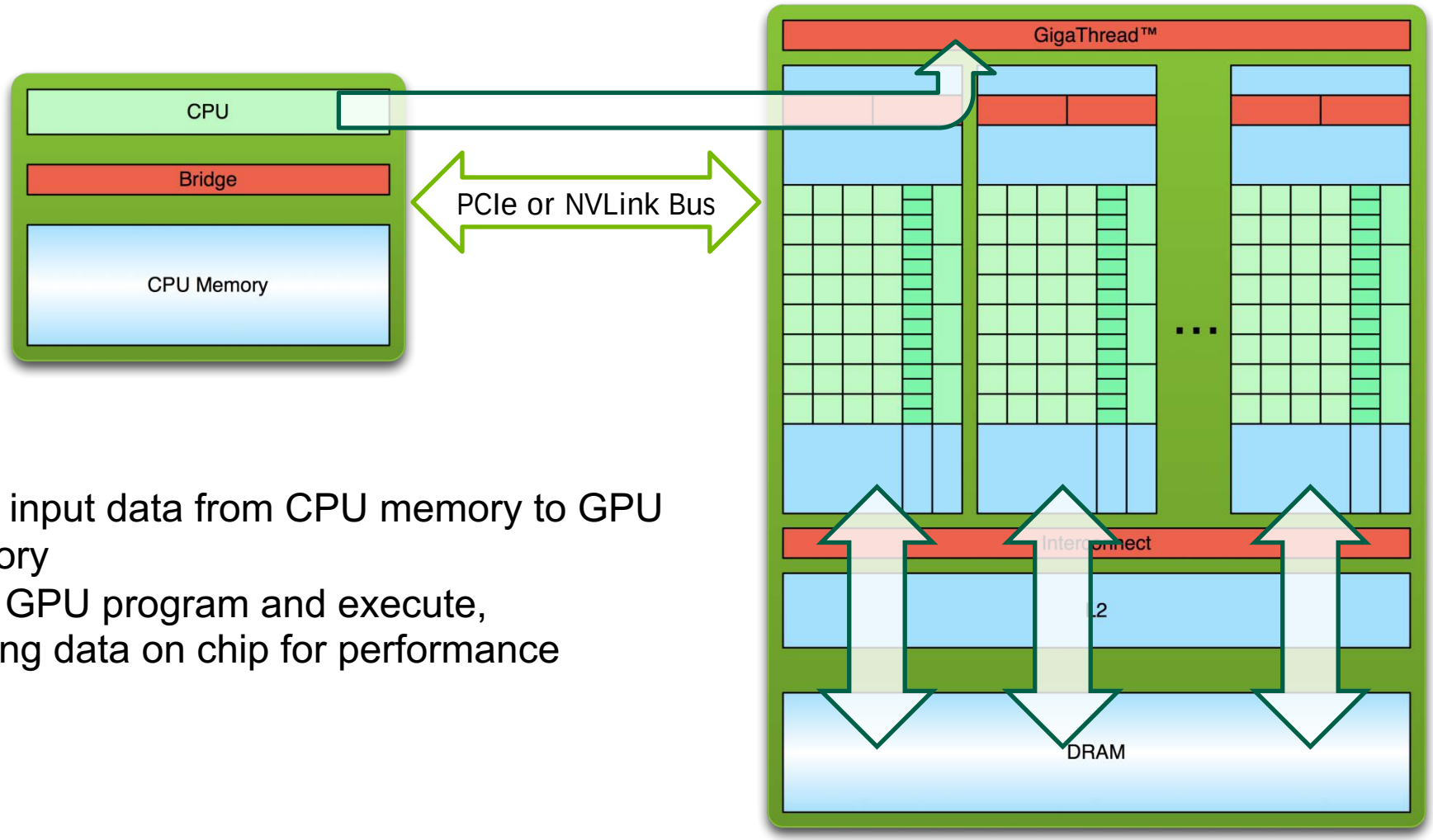


SIMPLE PROCESSING FLOW



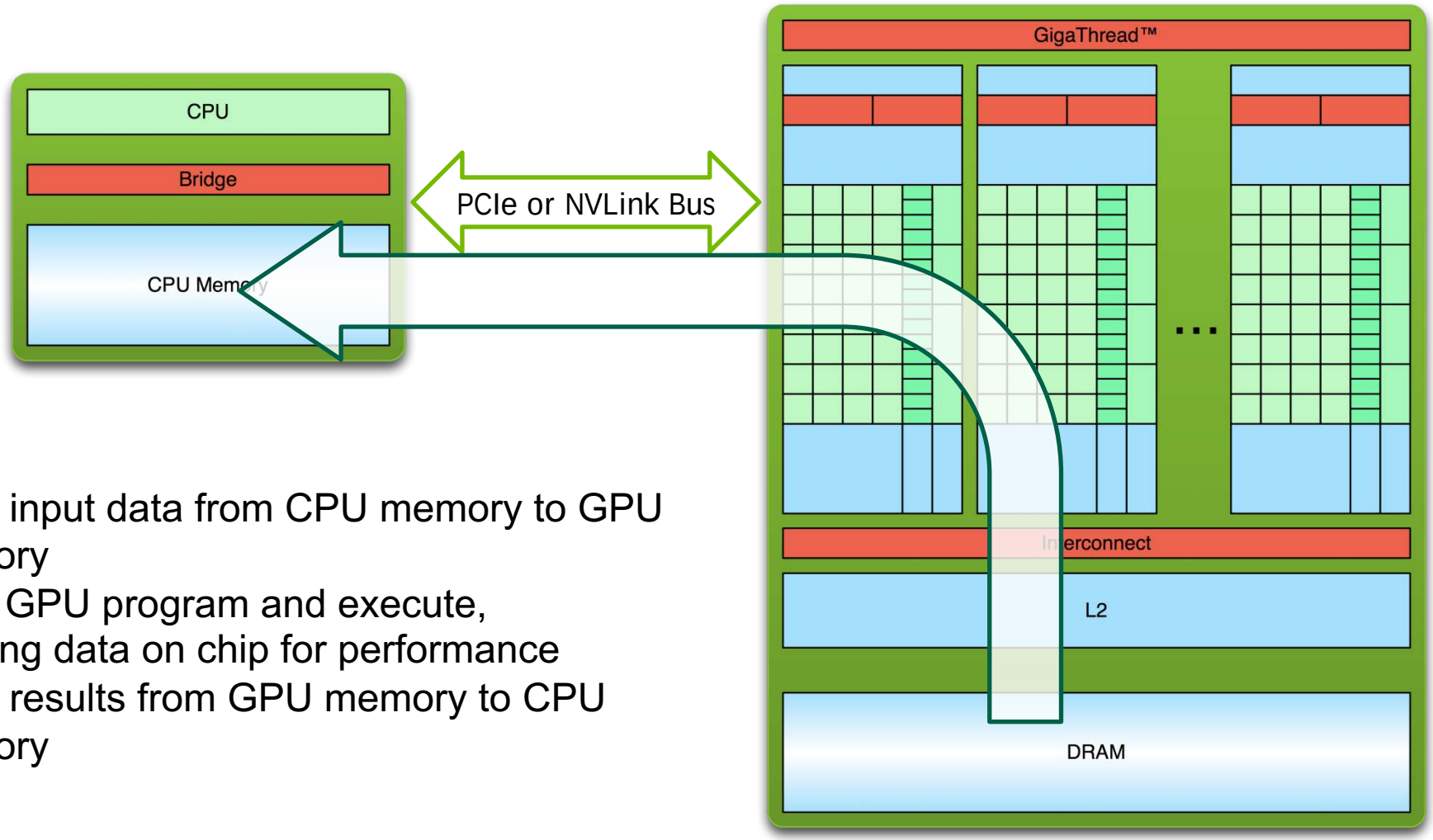
1. Copy input data from CPU memory to GPU memory

SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

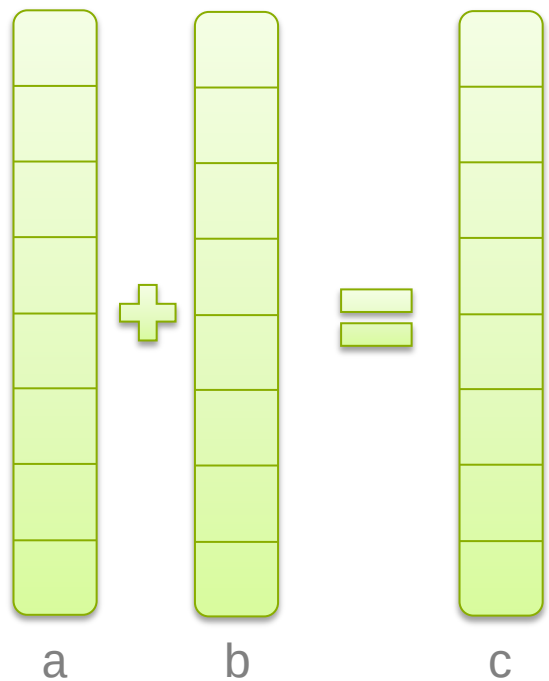
SIMPLE PROCESSING FLOW



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

PARALLEL PROGRAMMING IN CUDA C++

- GPU computing is about massive parallelism!
- We need an interesting example…
- We'll start with vector addition



GPU KERNELS: DEVICE CODE

```
__global__ void mykernel(void) {
}
```

- CUDA C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code (can also be called from other device code)
- **nvcc** separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler:
    - `gcc`, `cl.exe`

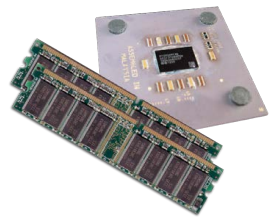
GPU KERNELS: DEVICE CODE

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
  - The parameters inside the triple angle brackets are the CUDA kernel execution configuration
- That’s all that is required to execute a function on the GPU!

MEMORY MANAGEMENT

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - Typically passed to device code
    - Typically *not* dereferenced in host code
  - *Host* pointers point to CPU memory
    - Typically not passed to device code
    - Typically *not* dereferenced in device code
  - (Special cases: Pinned pointers, ATS, managed memory)
- Simple CUDA API for handling device memory
  - **`cudaMalloc()`**, **`cudaFree()`**, **`cudaMemcpy()`**
  - Similar to the C equivalents **`malloc()`**, **`free()`**, **`memcpy()`**



RUNNING CODE IN PARALLEL

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

- Instead of executing **`add()`** once, execute N times in parallel

VECTOR ADDITION ON THE DEVICE

- With **add()** running in parallel we can do vector addition
- Terminology: each parallel invocation of **add()** is referred to as a block
  - The set of all blocks is referred to as a grid
  - Each invocation can refer to its block index using **blockIdx.x**

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using **blockIdx.x** to index into the array, each block handles a different index
- Built-in variables like **blockIdx.x** are zero-indexed (C/C++ style), 0..**N**-1, where **N** is from the kernel execution configuration indicated at the kernel launch

VECTOR ADDITION ON THE DEVICE

```
#define N 512
int main(void) {
   int *a, *b, *c;             // host copies of a, b, c
   int *d_a, *d_b, *d_c;       // device copies of a, b, c
   int size = N * sizeof(int);
   // Alloc space for device copies of a, b, c
   cudaMalloc((void **)&d_a, size);
   cudaMalloc((void **)&d_b, size);
   cudaMalloc((void **)&d_c, size);
   // Alloc space for host copies of a, b, c and setup input values
   a = (int *)malloc(size); random_ints(a, N);
   b = (int *)malloc(size); random_ints(b, N);
   c = (int *)malloc(size);
```

VECTOR ADDITION ON THE DEVICE

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

REVIEW (1 OF 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host (or possibly from other device code)
- Passing parameters from host code to a device function

REVIEW (2 OF 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

CUDA THREADS

- Terminology: a block can be split into parallel threads
- Let's change **`add()`** to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use **`threadIdx.x`** instead of **`blockIdx.x`**
- Need to make one change in **`main()`**:

```
add<<< 1, N >>>();
```

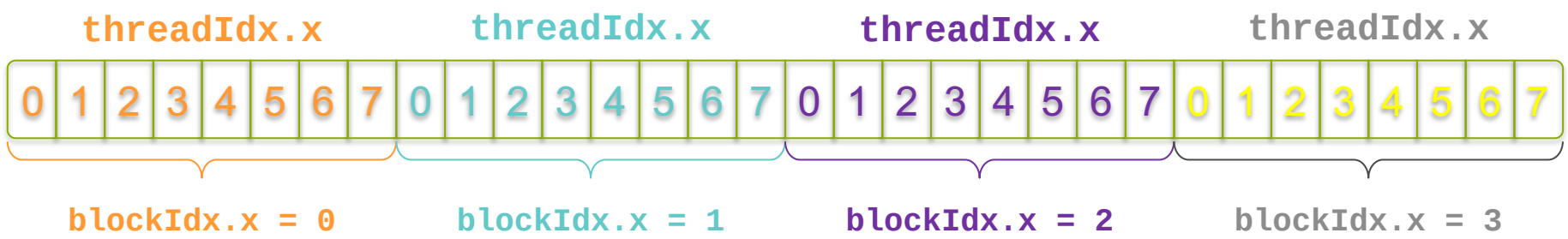
COMBINING BLOCKS *AND* THREADS

- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads

- Let’s adapt vector addition to use both *blocks* and *threads*
- Why? We’ll come to that…
- First let’s discuss data indexing…

INDEXING ARRAYS WITH BLOCKS AND THREADS

- No longer as simple as using **`blockIdx.x`** and **`threadIdx.x`**
  - Consider indexing an array with one element per thread (8 threads/block):

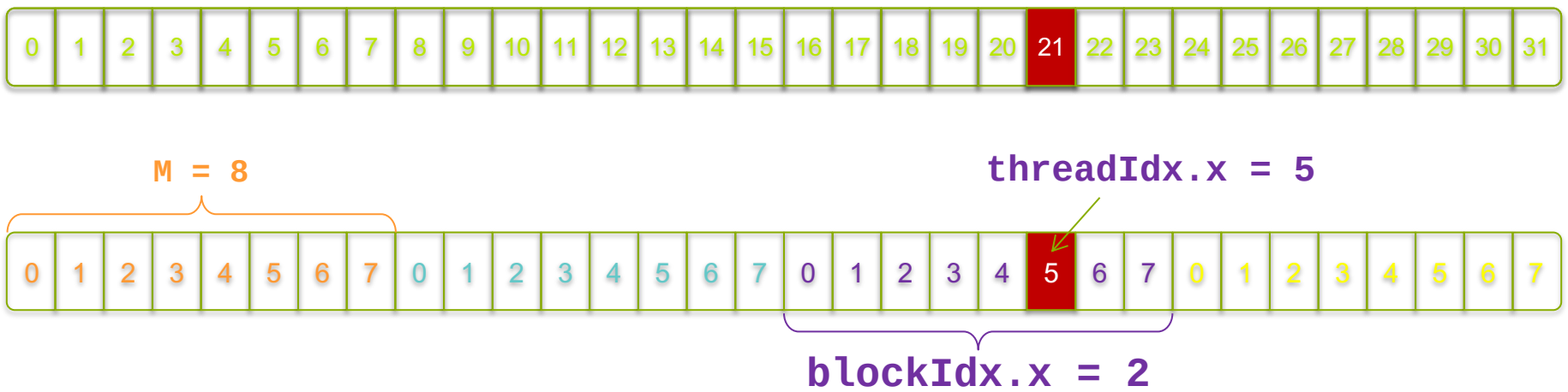


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

INDEXING ARRAYS: EXAMPLE

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +     2      * 8;
          = 21;
```

VECTOR ADDITION WITH BLOCKS AND THREADS

- Use the built-in variable **`blockDim.x`** for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

- Combined version of **`add()`** to use parallel threads *and* parallel blocks:

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in **`main()`**?

ADDITION WITH BLOCKS AND THREADS

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);
    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

ADDITION WITH BLOCKS AND THREADS

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

HANDLING ARBITRARY VECTOR SIZES

- Typical problems are not friendly multiples of **`blockDim.x`**
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

WHY BOTHER WITH THREADS?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example… (next session)

REVIEW

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N/M,M>>>(…);`
  - Use **`blockIdx.x`** to access block index
  - Use **`threadIdx.x`** to access thread index within block

- Assign elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

FUTURE SESSIONS

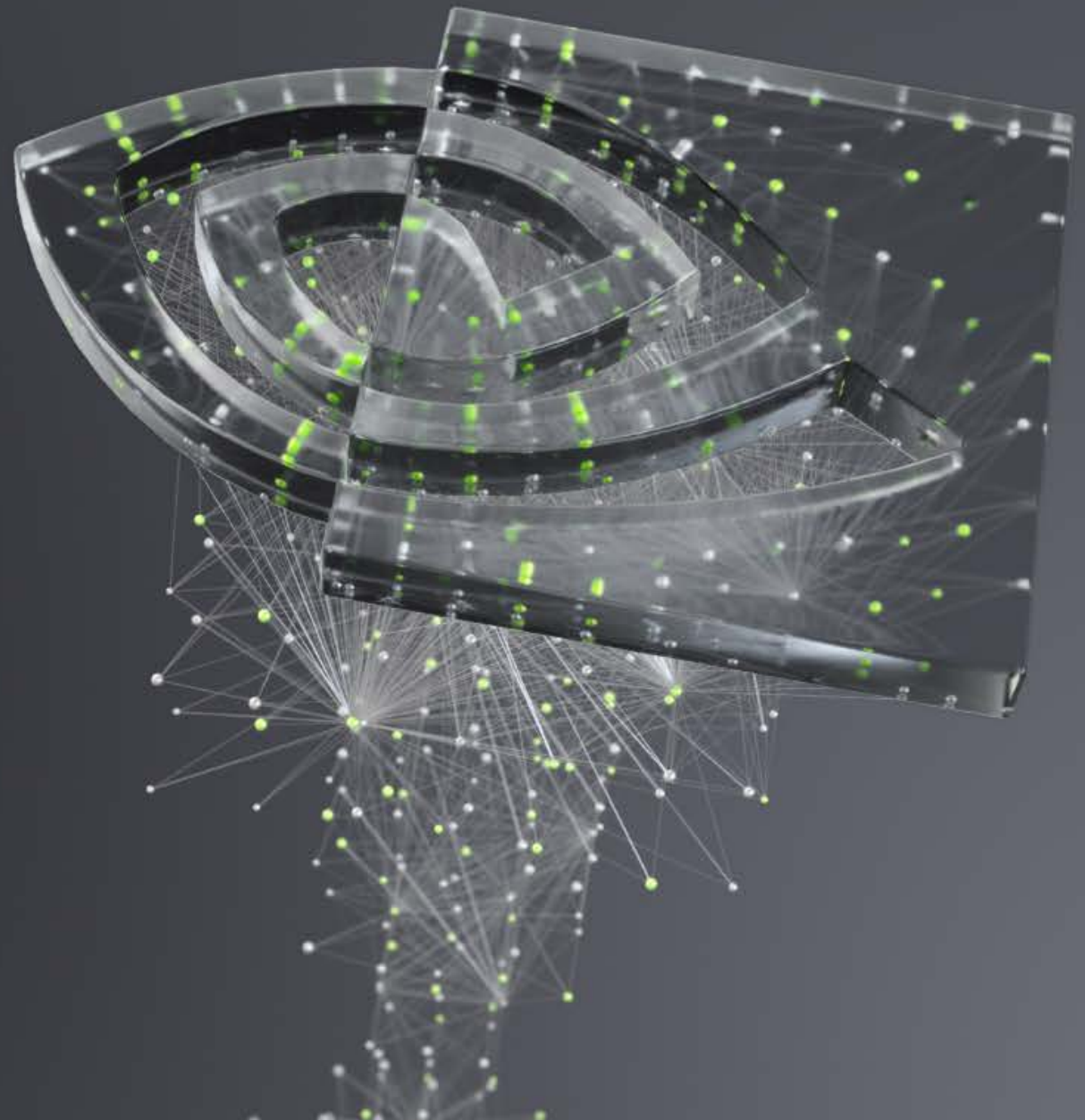
- CUDA Shared Memory
- CUDA GPU architecture and basic optimizations
- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

- An introduction to CUDA:
  - https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/
- Another introduction to CUDA:
  - https://devblogs.nvidia.com/even-easier-introduction-cuda/
- CUDA Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA Documentation:
  - https://docs.nvidia.com/cuda/index.html
  - https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (runtime API)

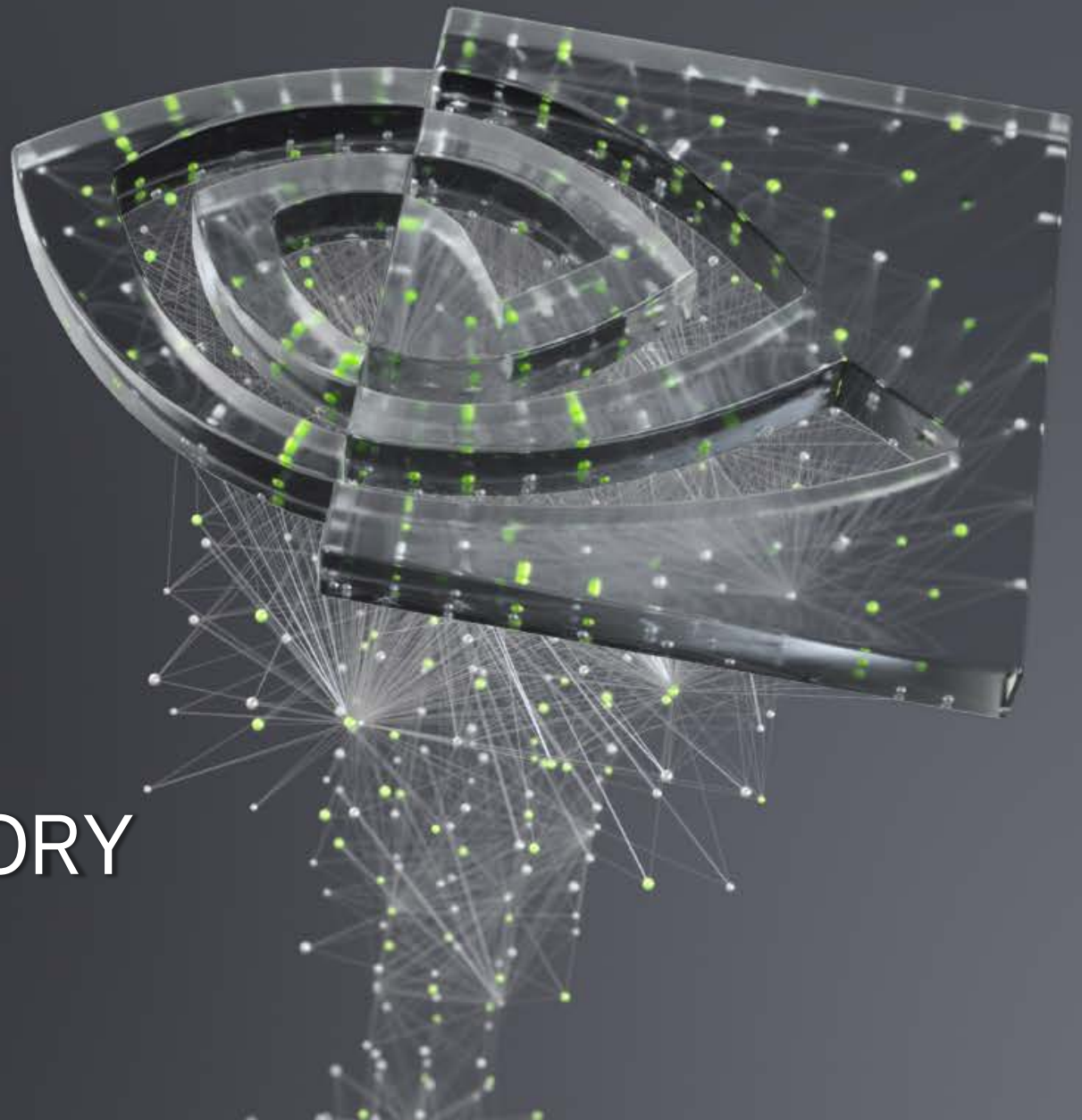
HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw1/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



NVIDIA

QUESTIONS?



CUDA SHARED MEMORY

NVIDIA Corporation

REVIEW (1 OF 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host (or possibly from other device code)
- Passing parameters from host code to a device function

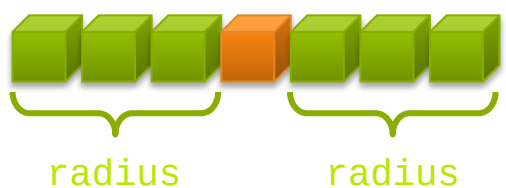
REVIEW (2 OF 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

1D STENCIL

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

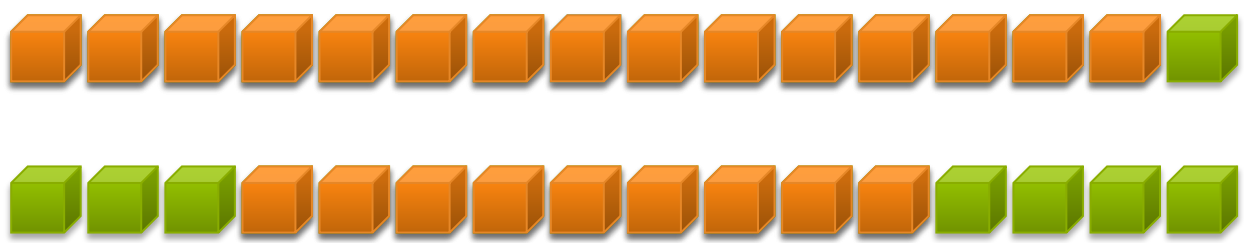
- If radius is 3, then each output element is the sum of 7 input elements:



IMPLEMENTING WITHIN A BLOCK

- Each thread processes one output element
  - `blockDim.x` elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times

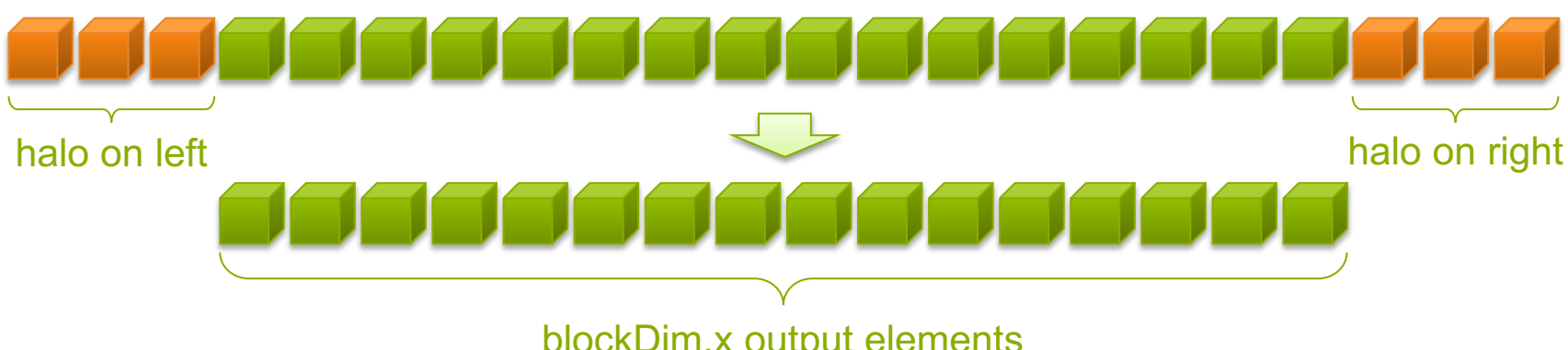


SHARING DATA BETWEEN THREADS

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

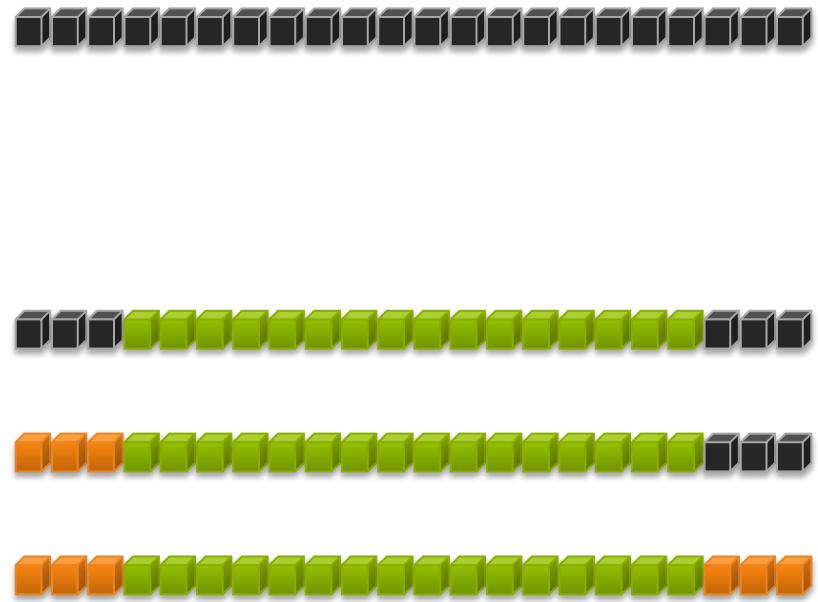
IMPLEMENTING WITH SHARED MEMORY

- Cache data in shared memory
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory
- Each block needs a halo of `radius` elements at each boundary



STENCIL KERNEL

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



STENCIL KERNEL

```
  // Apply the stencil
 int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```

DATA RACE!

- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched

```
temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex – RADIUS] = in[gindex – RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    int result = 0;
    result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

__SYNCTHREADS()

- `void __syncthreads();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

STENCIL KERNEL

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex – RADIUS] = in[gindex – RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
```

STENCIL KERNEL

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

REVIEW

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

LOOKING FORWARD

Cooperative Groups: a flexible model for synchronization and communication within groups of threads.

At a glance

Scalable Cooperation among groups of threads

Flexible parallel decompositions

Composition across software boundaries

Deploy Everywhere

Benefits all applications

Examples include:
- Persistent RNNs
- Physics
- Search Algorithms
- Sorting

FOR EXAMPLE: THREAD BLOCK

Implicit group of all the threads in the launched thread block

Implements the same interface as thread_group:

```
void sync();                // Synchronize the threads in the group
unsigned size();            // Total number of threads in the group
unsigned thread_rank();     // Rank of the calling thread within [0, size)
bool is_valid();            // Whether the group violated any API constraints
```

And additional thread_block specific functions:

```
dim3 group_index();         // 3-dimensional block index within the grid
dim3 thread_index();        // 3-dimensional thread index within the block
```

NARROWING THE SHARED MEMORY GAP

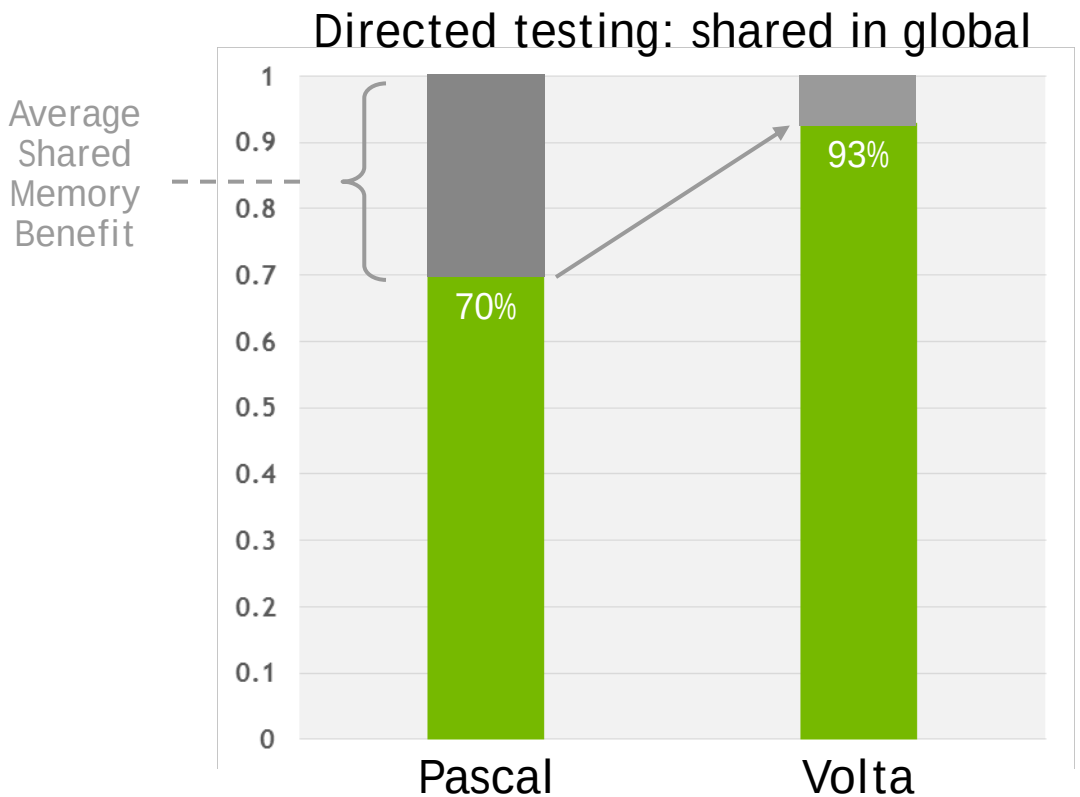
with the GV100 L1 cache

Cache: vs shared

- Easier to use
- 90%+ as good

Shared: vs cache

- Faster atomics
- More banks
- More predictable



FUTURE SESSIONS

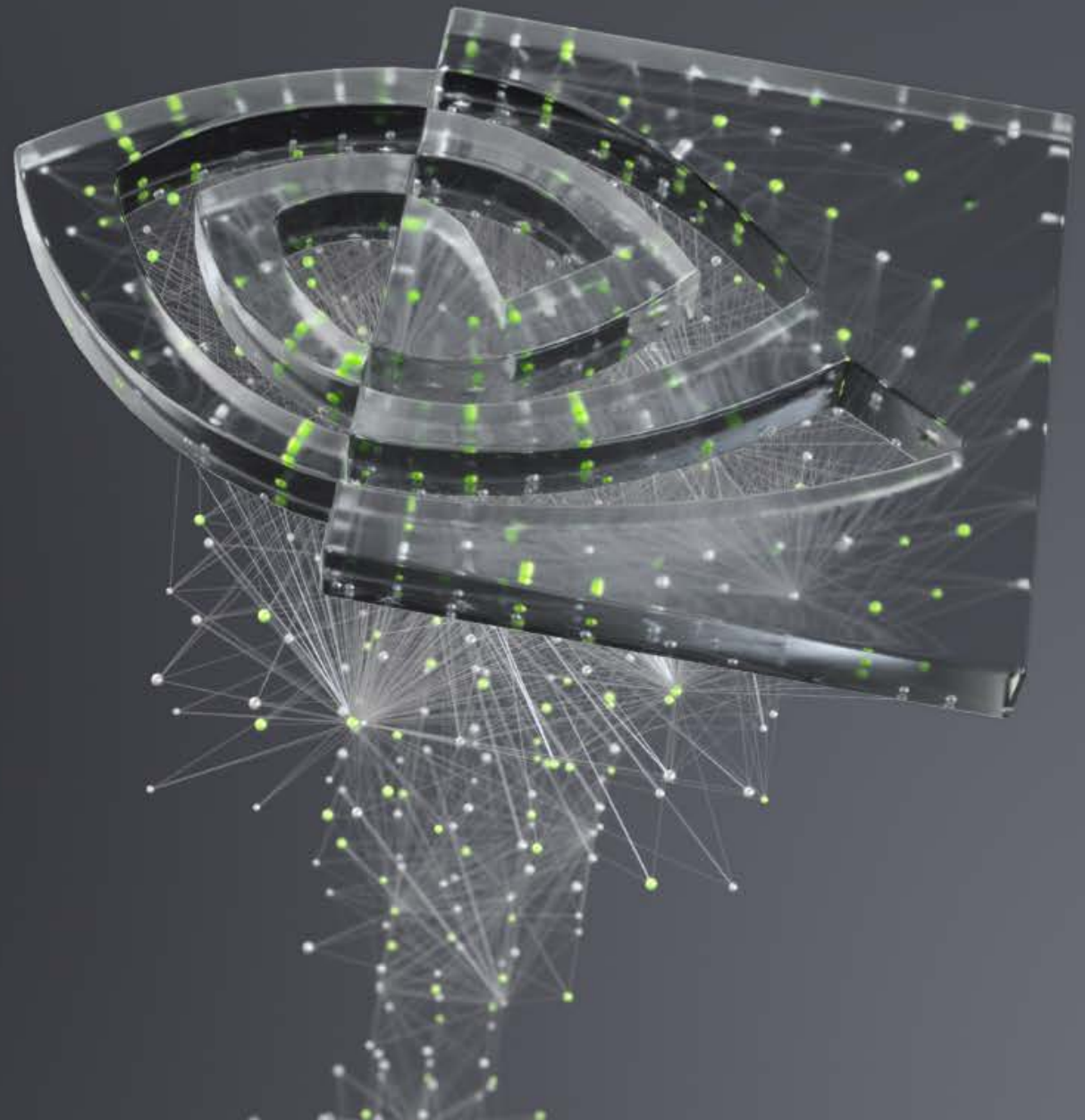
- CUDA GPU architecture and basic optimizations
- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

- Shared memory:
  - https://devblogs.nvidia.com/using-shared-memory-cuda-cc/
- CUDA Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory
- CUDA Documentation:
  - https://docs.nvidia.com/cuda/index.html
  - https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (runtime API)

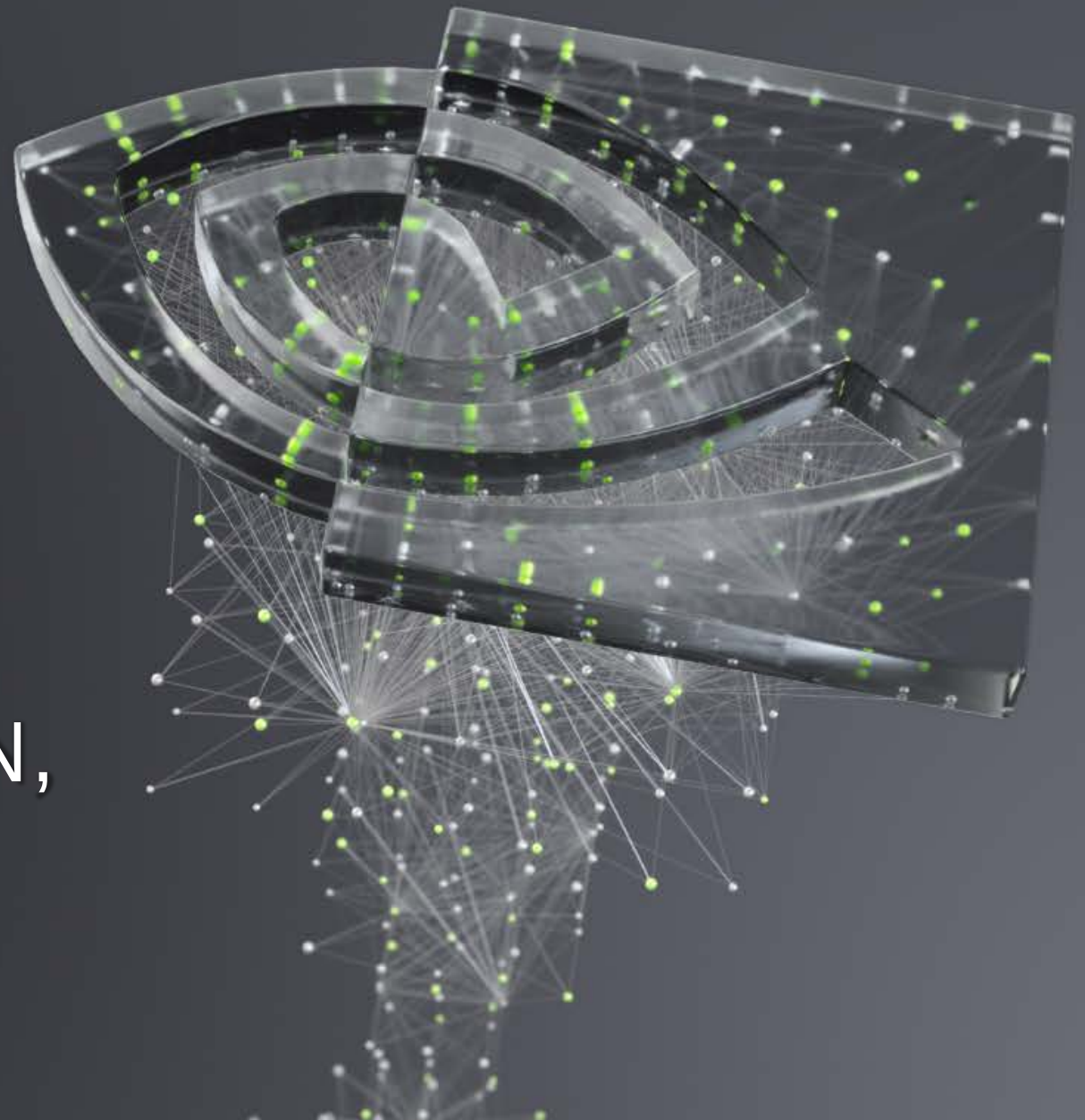
HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw2/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



NVIDIA

QUESTIONS?



CUDA OPTIMIZATION, PART 1

NVIDIA Corporation

OUTLINE

- Architecture:

  Kepler/Maxwell/Pascal/Volta

- Kernel optimizations
  - Launch configuration (use lots of threads)
- Part 2 (next session):
  - Global memory throughput (use memory efficiently)
  - Shared memory access

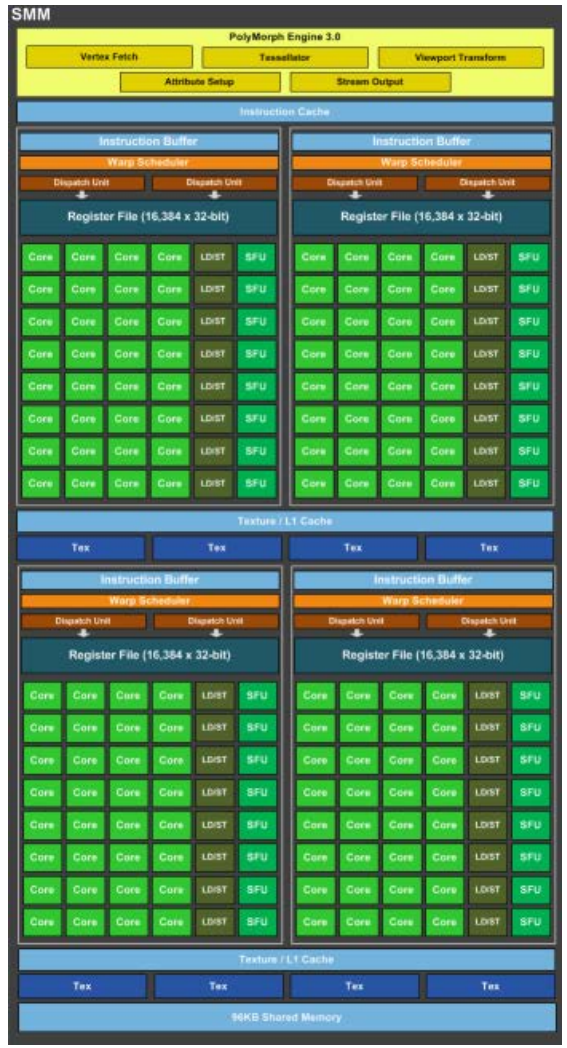
Most concepts in this presentation apply to *any* language or API on NVIDIA GPUs

KEPLER CC 3.5 SM (GK110)

- "SMX" (enhanced SM)
- 192 SP units ("cores")
- 64 DP units
- LD/ST units, 64K registers
- 4 warp schedulers
- Each warp scheduler is dual-issue capable
- K20: 13 SMX's, 5GB
- K20X: 14 SMX's, 6GB
- K40: 15 SMX's, 12GB

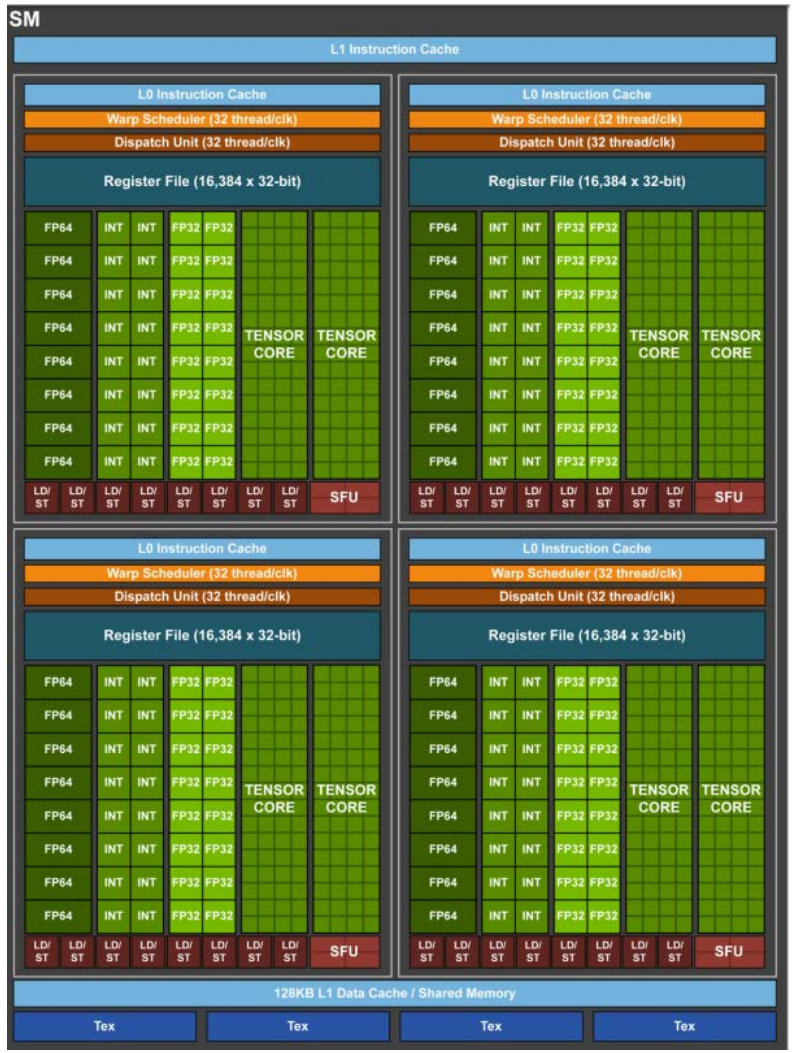
MAXWELL/PASCAL CC5.2, CC6.1 SM

- "SMM" (enhanced SM)
- 128 SP units ("cores")
- 4 DP units
- LD/ST units
- cc 6.1: INT8
- 4 warp schedulers
- Each warp scheduler is dual-issue capable
- M40: 24 SMM's, 12/24GB
- P40: 30 SM's, 24GB
- P4: 20 SM's, 8GB



PASCAL/VOLTA CC6.0/7.0

- 64 SP units (“cores”)
- 32 DP units
- LD/ST units
- FP16 @ 2x SP rate
- cc7.0: TensorCore
- P100/V100 2/4 warp schedulers
- Volta adds separate int32 units
- P100: 56 SM’s, 16GB
- V100: 80 SM’s, 16/32GB



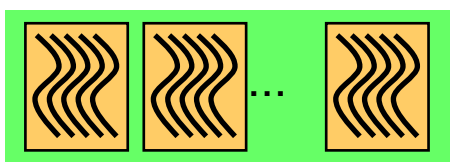
NVIDIA

EXECUTION MODEL

Software

Thread

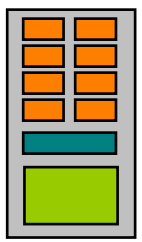
Thread
Block



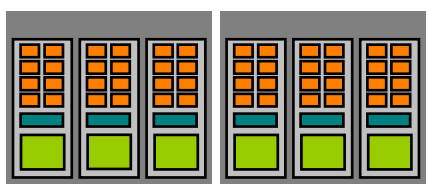
Grid

Hardware

Scalar
Processor



Multiprocessor



Device

Threads are executed by scalar processors

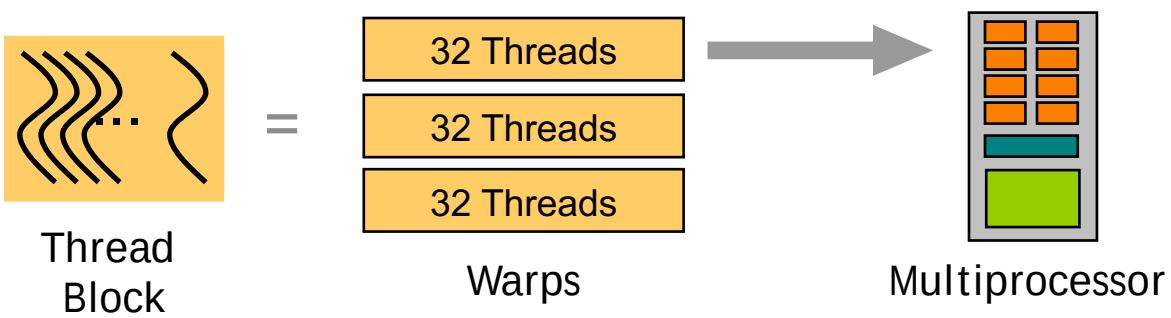
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

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

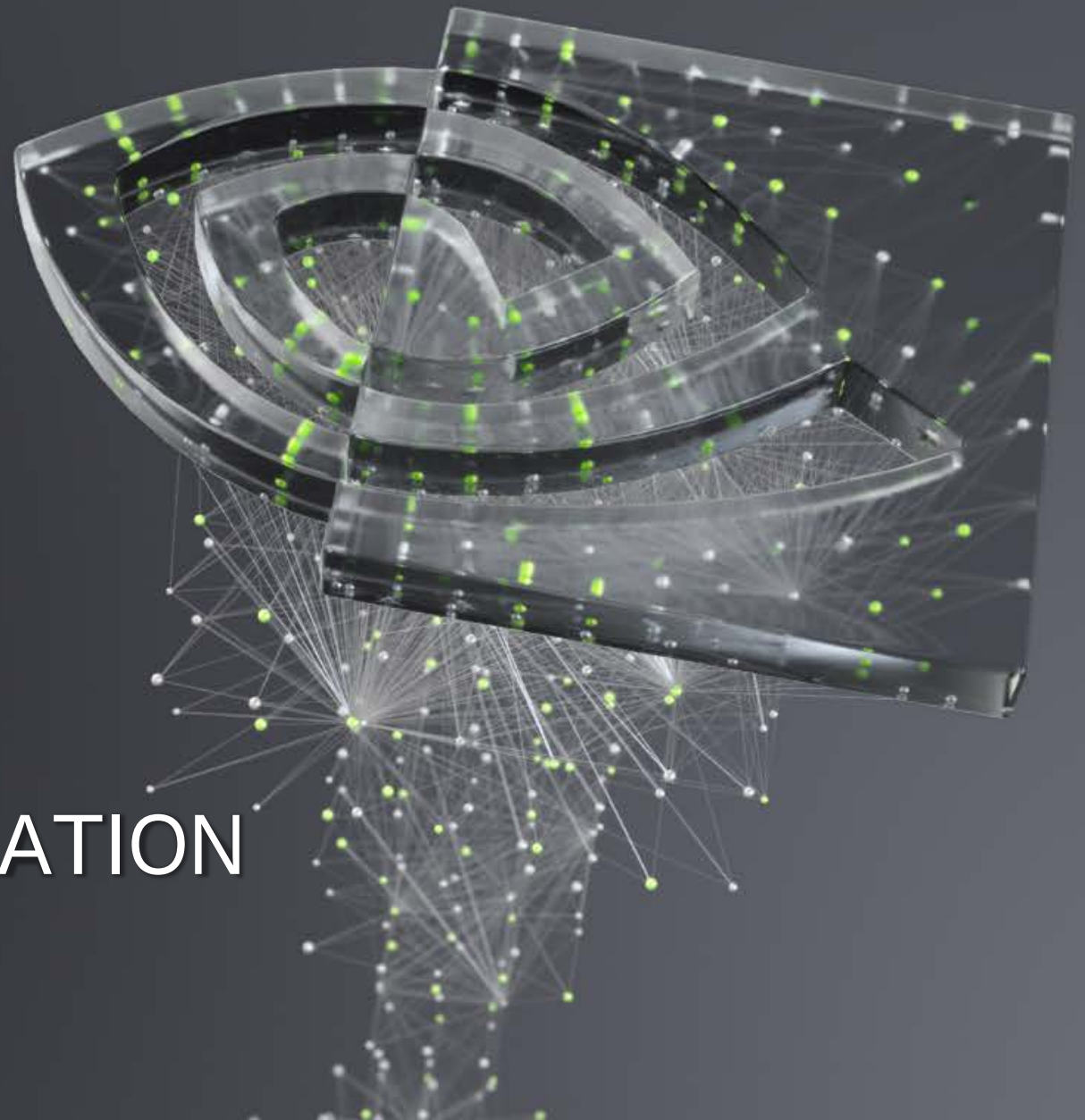
A kernel is launched as a grid of thread blocks

WARPS



A thread block consists of 32-thread warps

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



LAUNCH CONFIGURATION

LAUNCH CONFIGURATION

- Key to understanding:
  - Instructions are issued in order
  - A thread stalls when one of the operands isn't ready:
    - Memory read by itself doesn't stall execution
  - Latency is hidden by switching threads
    - GMEM latency: >100 cycles (varies by architecture/design)
    - Arithmetic latency: <100 cycles (varies by architecture/design)
- How many threads/threadblocks to launch?
- Conclusion:
  - Need enough threads to hide latency

GPU LATENCY HIDING

- In CUDA C source code:
- int idx = threadIdx.x+blockDim.x*blockIdx.x;
- c[idx] = a[idx] * b[idx];

- In machine code:
- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

clock cycles:

C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16 C17 C18 …

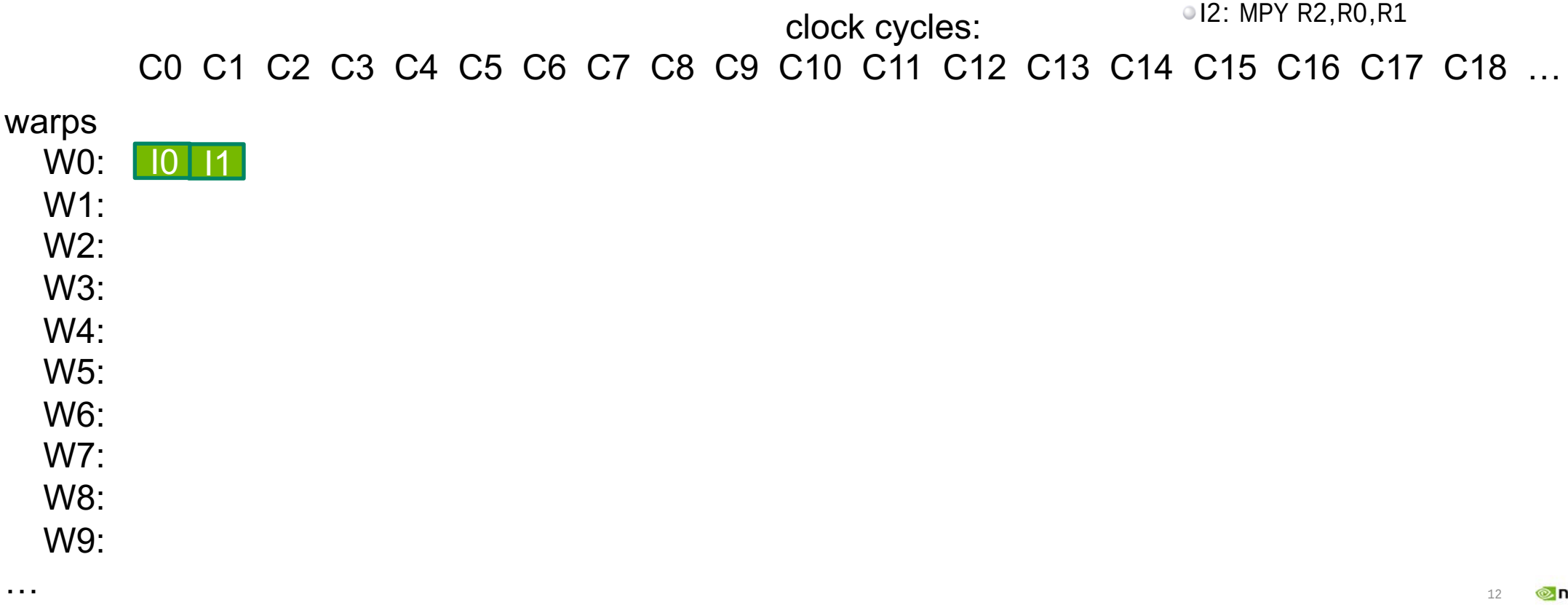
warps

W0:
W1:
W2:
W3:
W4:
W5:
W6:
W7:
W8:
W9:

…

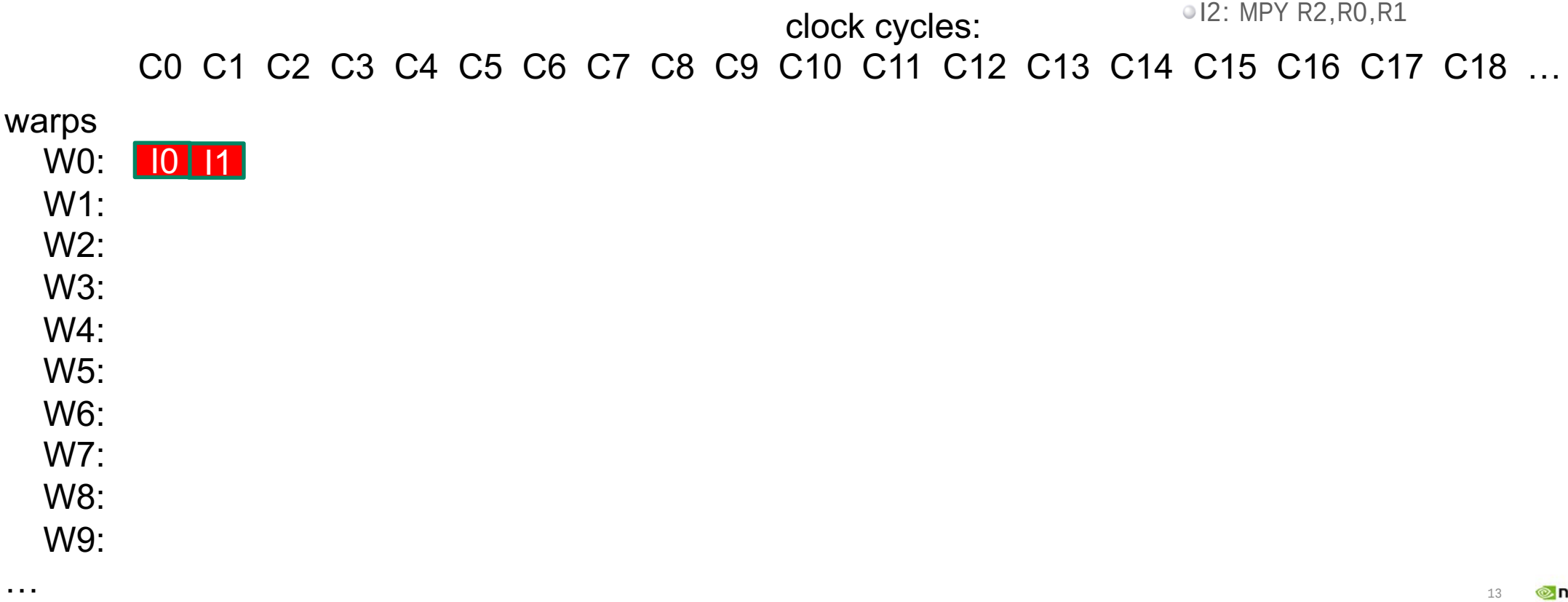
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



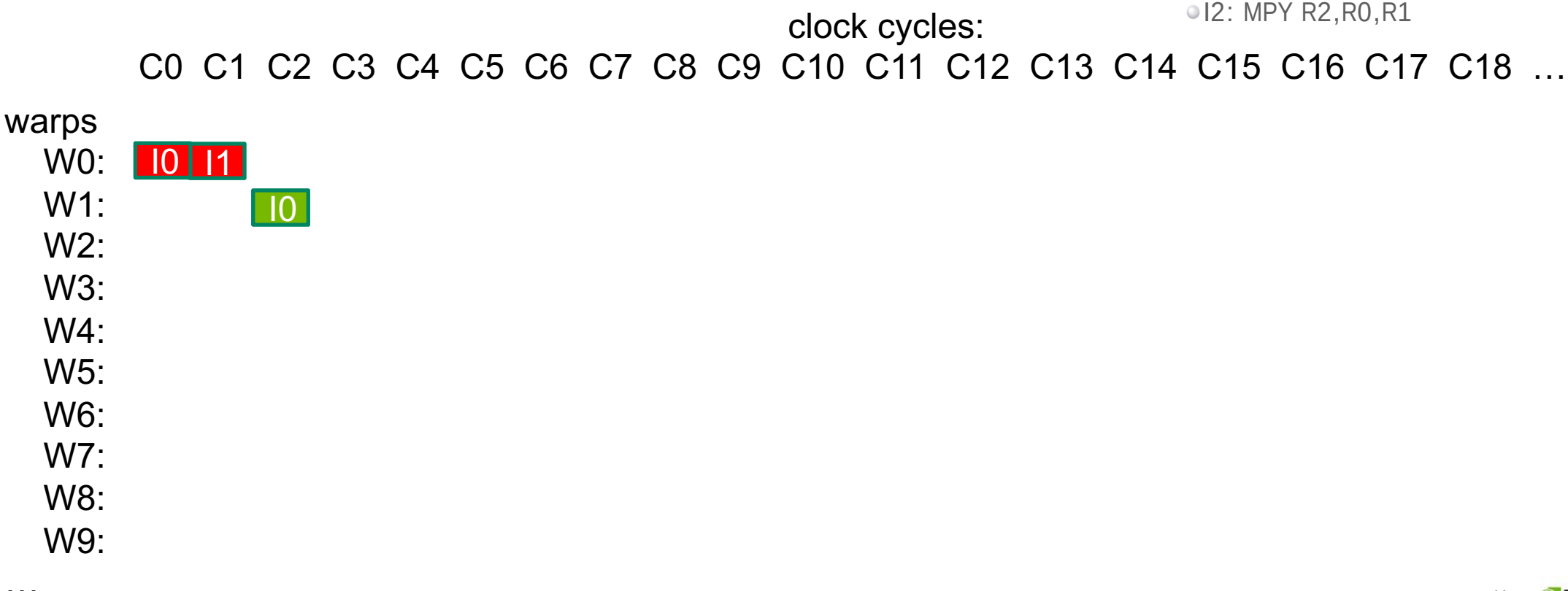
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



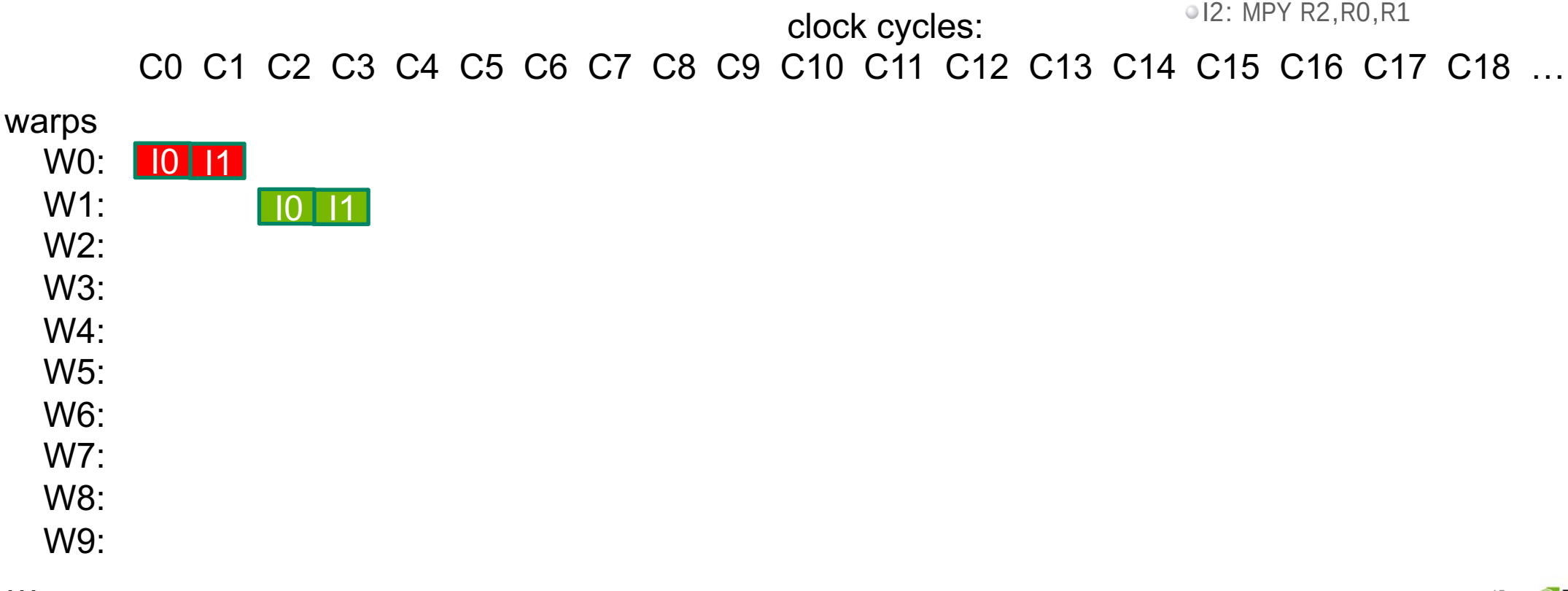
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



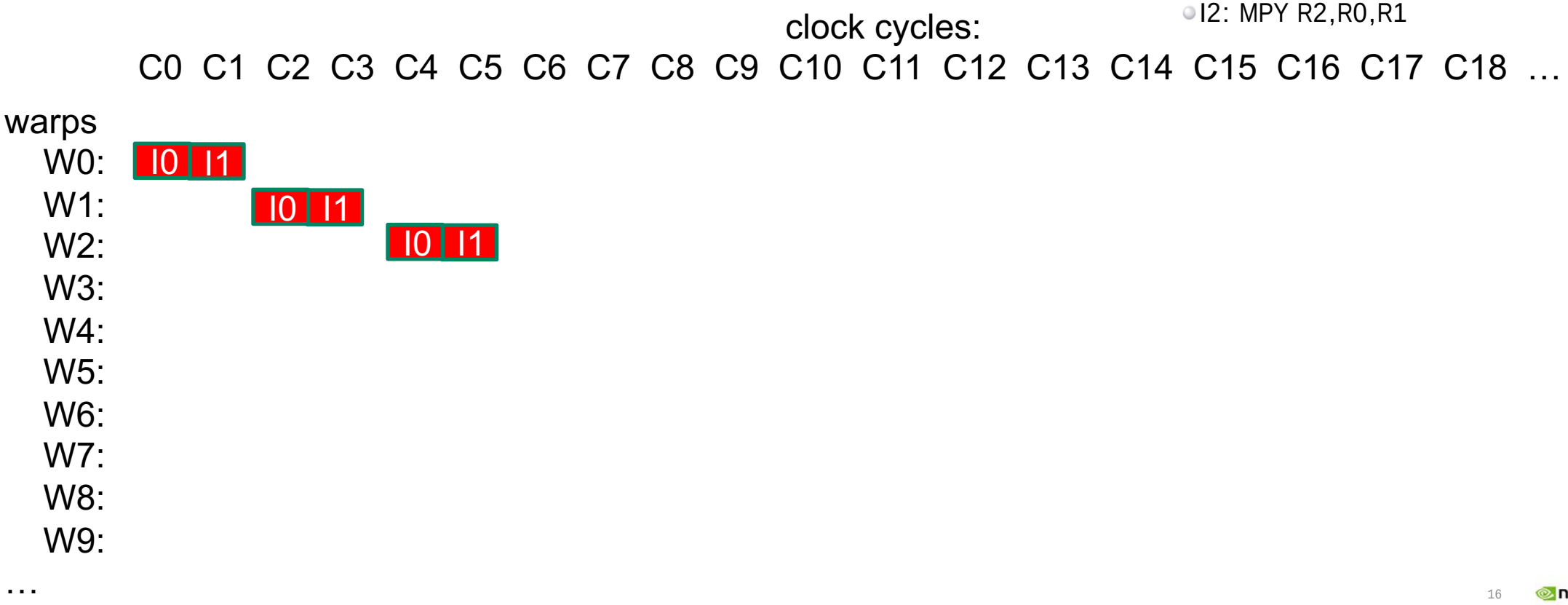
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



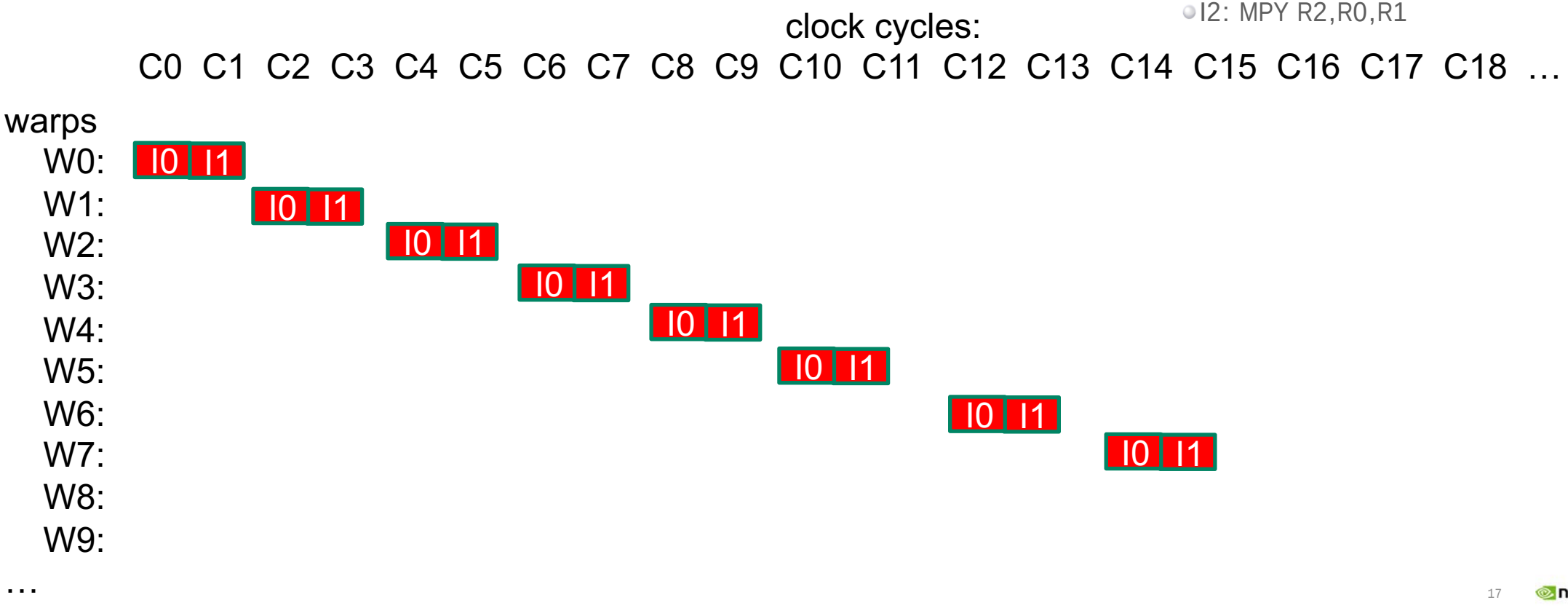
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



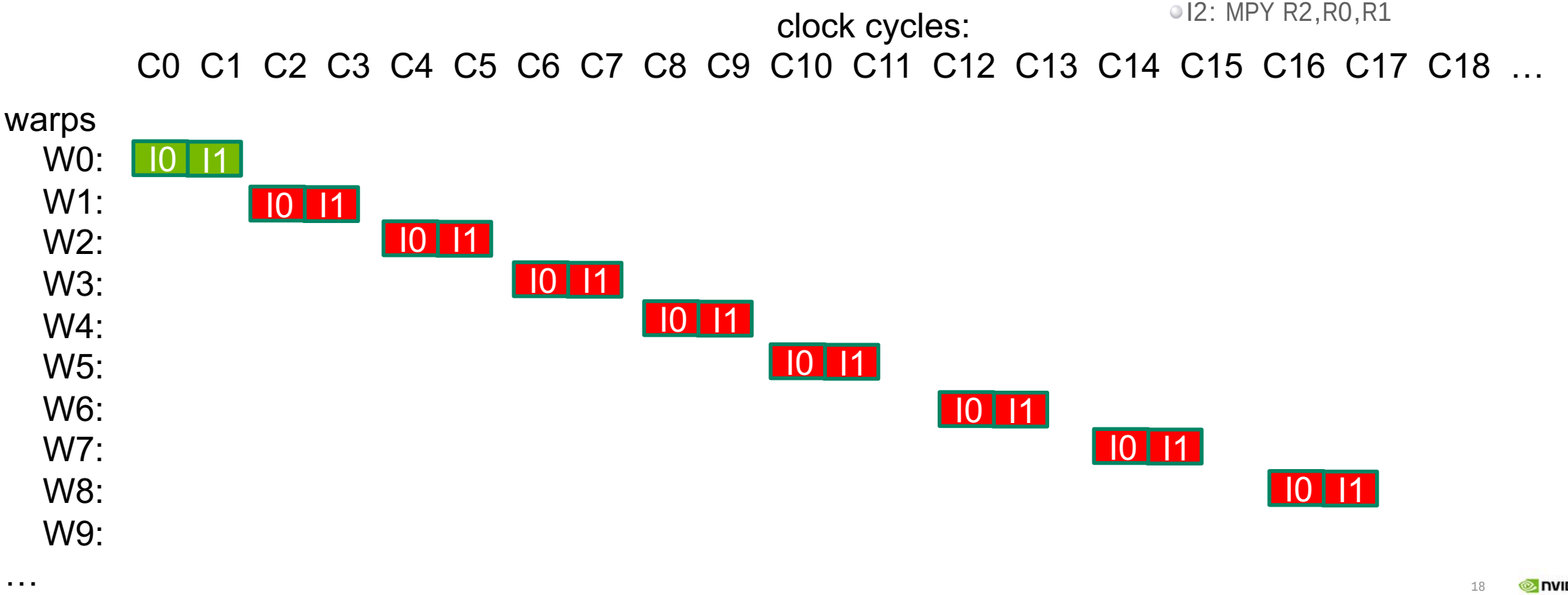
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



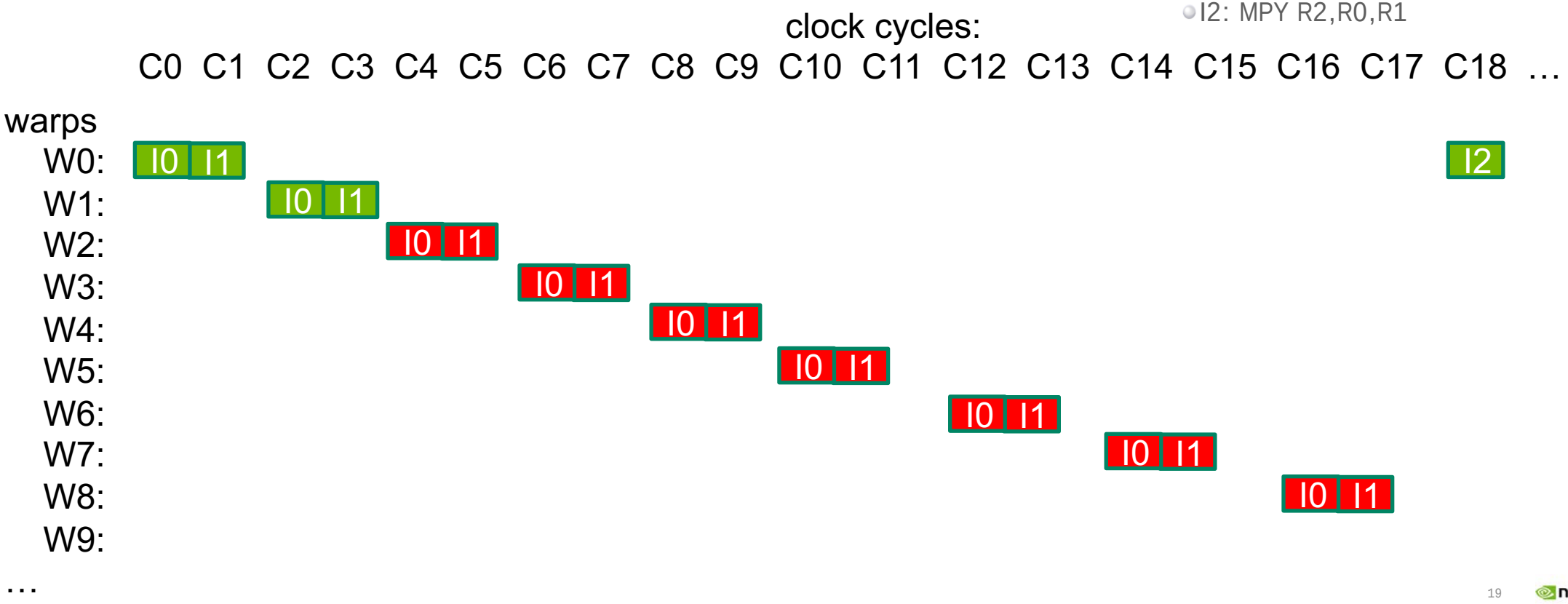
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



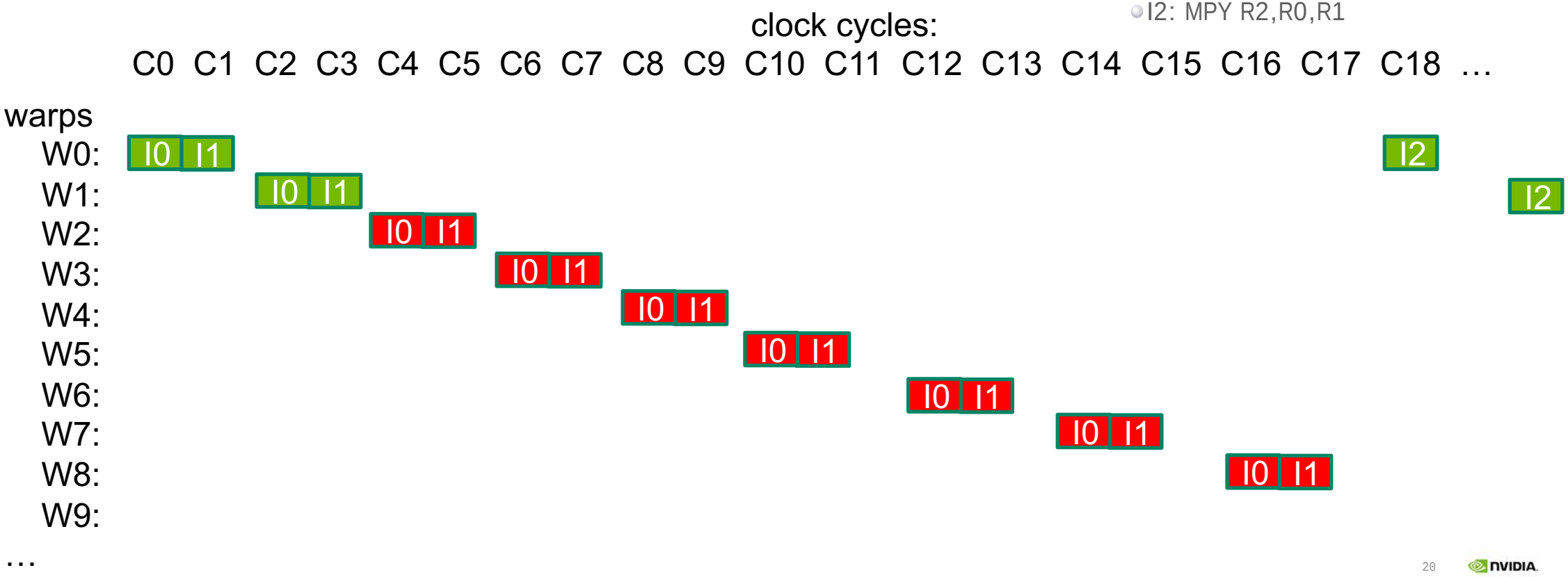
GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



GPU LATENCY HIDING – INSIDE THE SM

- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



LAUNCH CONFIGURATION

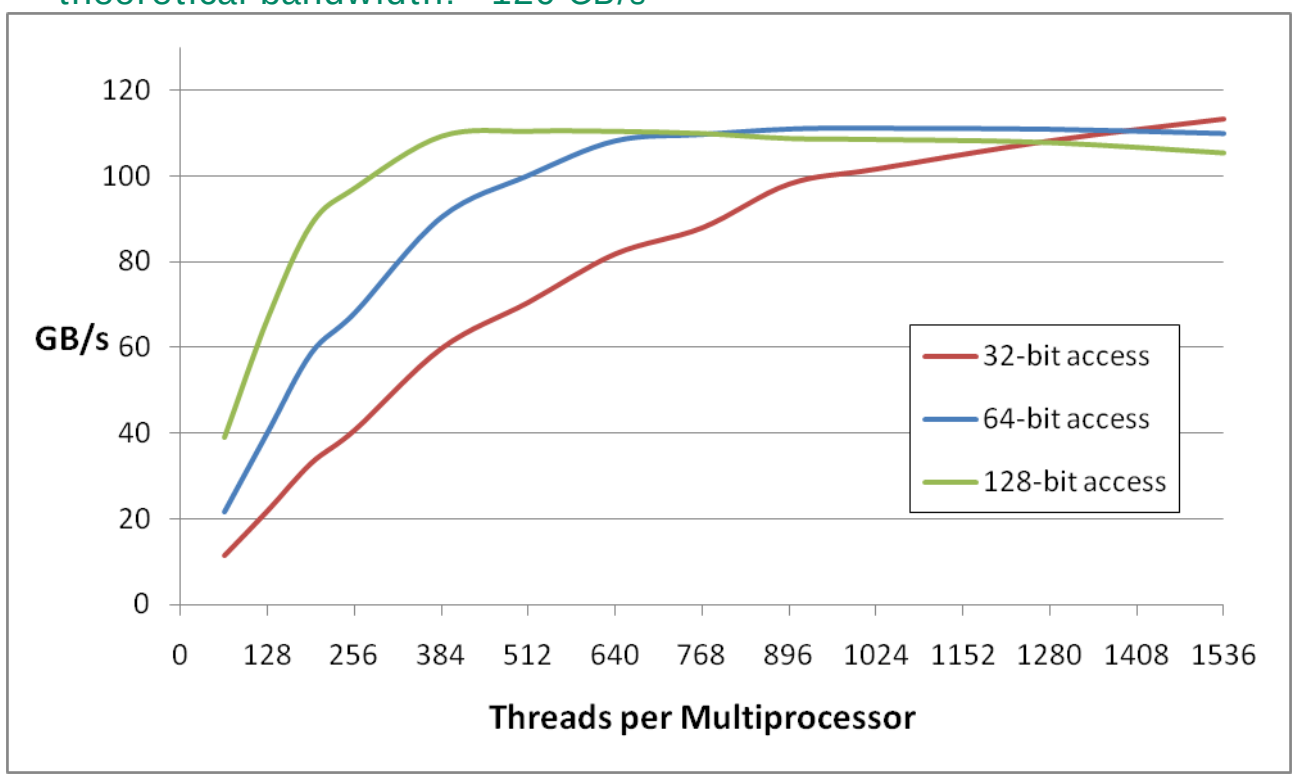
- Hiding arithmetic latency:
  - Need ~10's warps (~320 threads) per SM
  - Or, latency can also be hidden with independent instructions from the same warp
    - ->if instructions never depends on the output of preceding instruction, then only 5 warps are needed, etc.
- Maximizing global memory throughput:
  - Depends on the access pattern, and word size
  - Need enough memory transactions in flight to saturate the bus
    - Independent loads and stores from the same thread
    - Loads and stores from different threads
    - Larger word sizes can also help (float2 is twice the transactions of float, for example)

MAXIMIZING MEMORY THROUGHPUT

Increment of an array of 64M elements

Two accesses per thread (load then store) - dependent, so really 1 access per thread at a time

theoretical bandwidth: ~120 GB/s



Several independent smaller accesses have the same effect as one larger one.

For example:

Four 32-bit ~= one 128-bit

LAUNCH CONFIGURATION: SUMMARY

- Need enough total threads to keep GPU busy
  - Typically, you'd like 512+ threads per SM (aim for 2048 - maximum "occupancy")
    - More if processing one fp32 element per thread
  - Of course, exceptions exist
- Threadblock configuration
  - Threads per block should be a multiple of warp size (32)
  - SM can concurrently execute at least 16 thread blocks (Maxwell/Pascal/Volta: 32)
    - Really small thread blocks prevent achieving good occupancy
    - Really large thread blocks are less flexible
    - Could generally use 128-256 threads/block, but use whatever is best for the application

ASIDE: WHAT IS OCCUPANCY?

- A measure of the actual thread load in an SM, vs. peak theoretical/peak achievable
- CUDA includes an occupancy calculator spreadsheet
- Achievable occupancy is affected by limiters to occupancy
- Primary limiters:
  - Registers per thread (can be reported by the profiler, or can get at compile time)
  - Threads per threadblock
  - Shared memory usage

SUMMARY

- GPU is a massively thread-parallel, latency hiding machine
- Kernel Launch Configuration:
  - Launch enough threads per SM to hide latency
  - Launch enough threadblocks to load the GPU
- Use analysis/profiling when optimizing:
  - “Analysis-driven Optimization” (future session)
  - -> Nsight Compute can show you information about whether you’ve saturated the compute subsystem or the memory subsystem.

FUTURE SESSIONS

- Fundamental Optimization, Part 2
- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

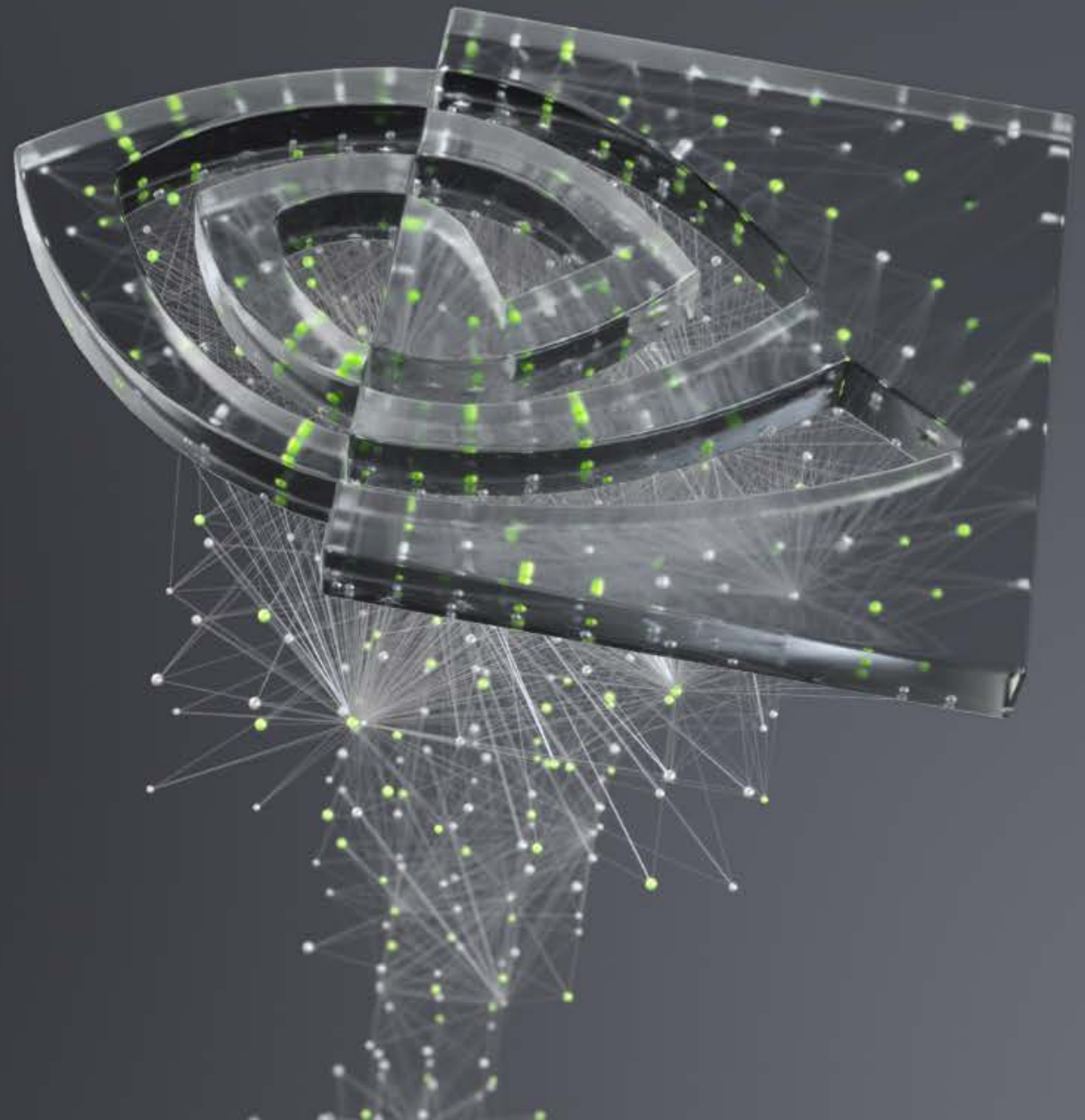
FURTHER STUDY

- Optimization in-depth:
  - http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf
- Analysis-Driven Optimization:
  - http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf
- CUDA Best Practices Guide:
  - https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
- CUDA Tuning Guides:
  - https://docs.nvidia.com/cuda/index.html#programming-guides

  (Kepler/Maxwell/Pascal/Volta)

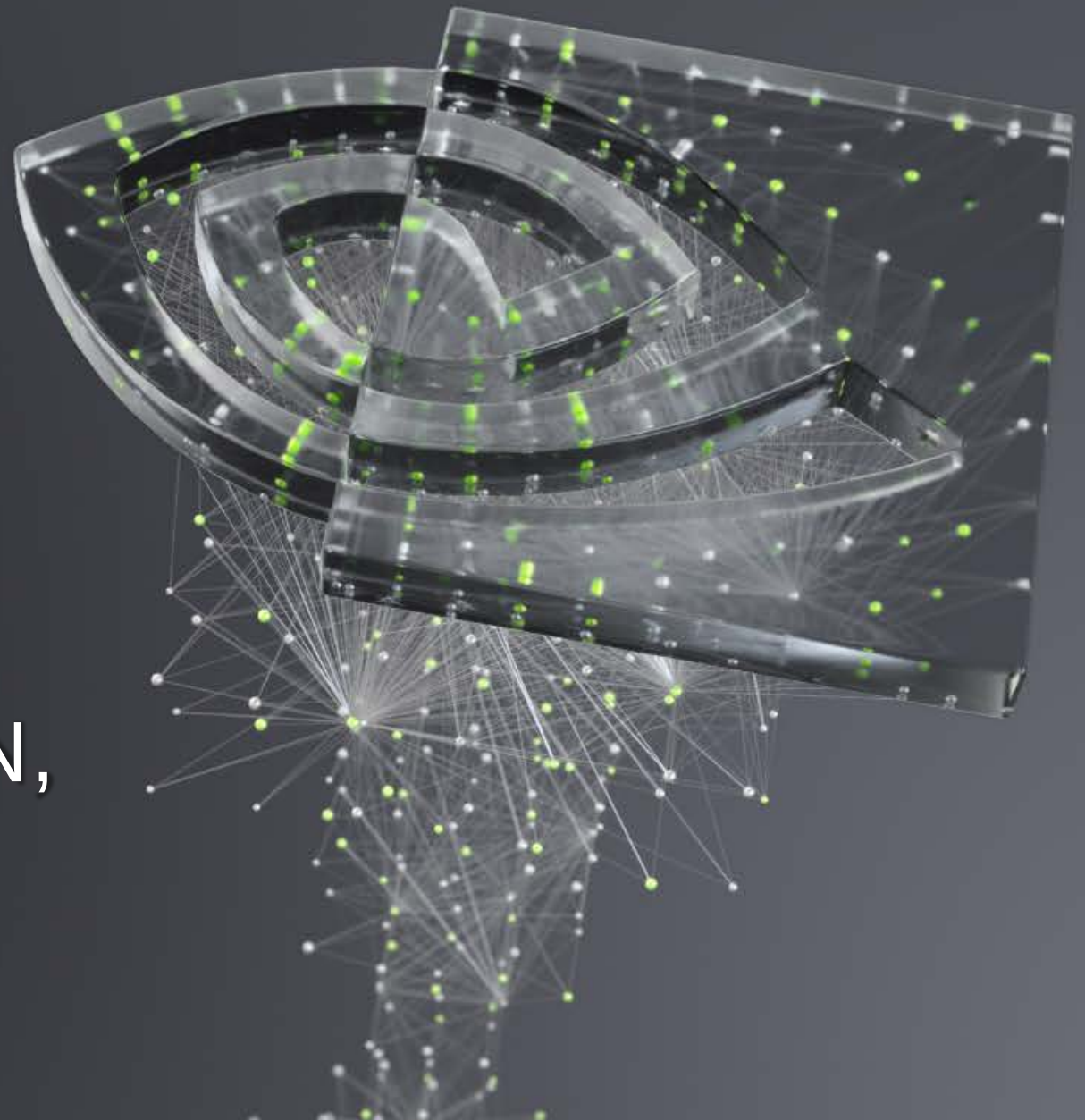
HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw3/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



NVIDIA

QUESTIONS?



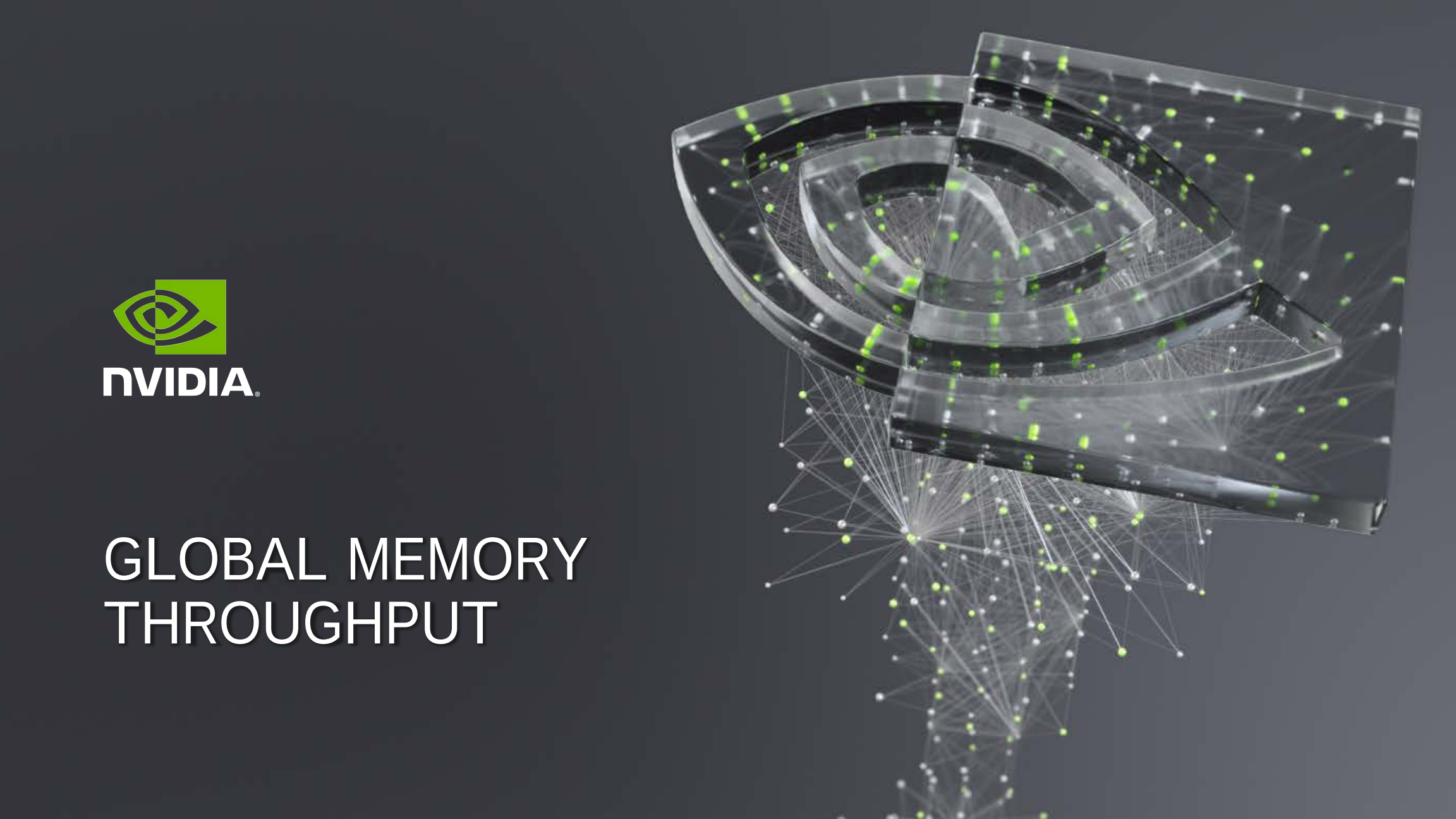
CUDA OPTIMIZATION, PART 2

NVIDIA Corporation

OUTLINE

- Architecture:
  - Kepler/Maxwell/Pascal/Volta
- Kernel optimizations
  - Launch configuration
- Part 2 (this session):
  - Global memory throughput
  - Shared memory access

Most concepts in this presentation apply to *any* language or API on NVIDIA GPUs



GLOBAL MEMORY THROUGHPUT

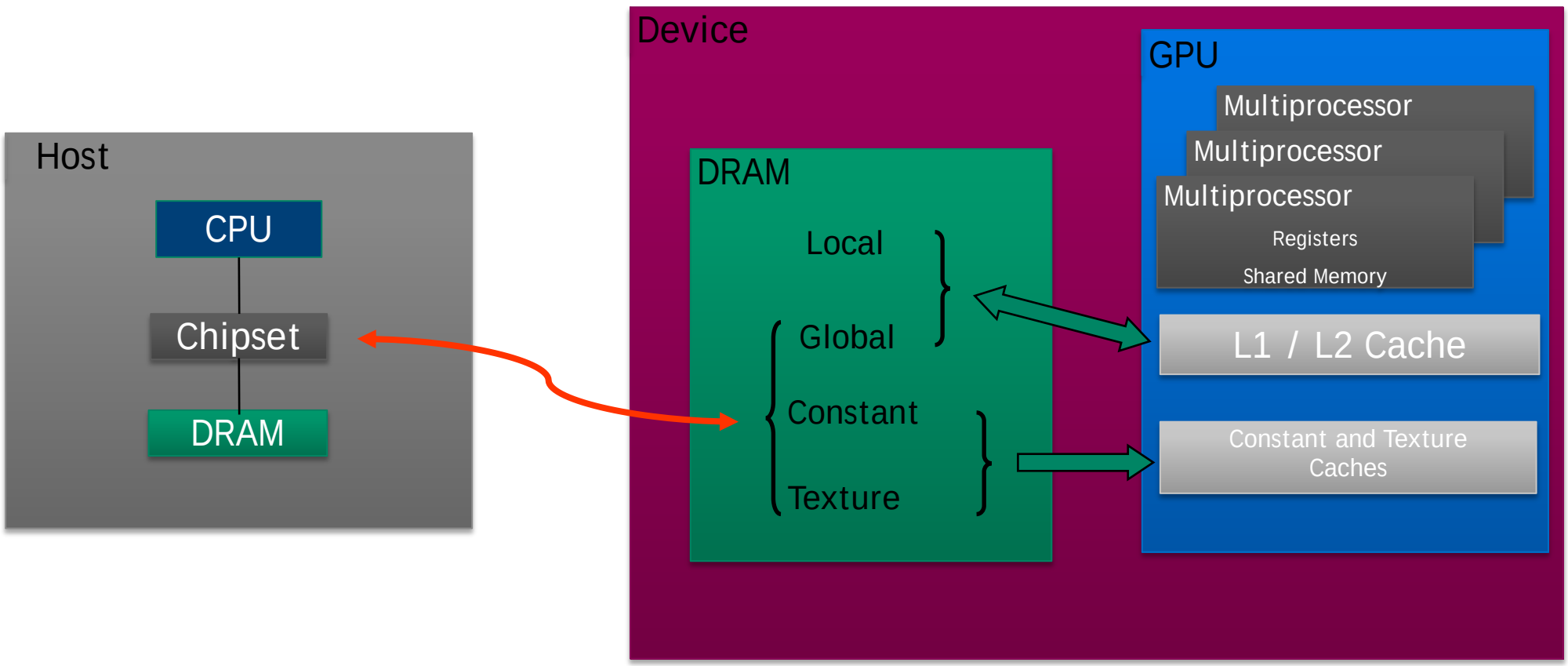
MEMORY HIERARCHY REVIEW

- Local storage
  - Each thread has own local storage
  - Typically registers (managed by the compiler)
- Shared memory / L1
  - Program configurable: typically up to 48KB shared (or 64KB, or 96KB…)
  - Shared memory is accessible by threads in the same threadblock
  - Very low latency
  - Very high throughput: >1 TB/s aggregate

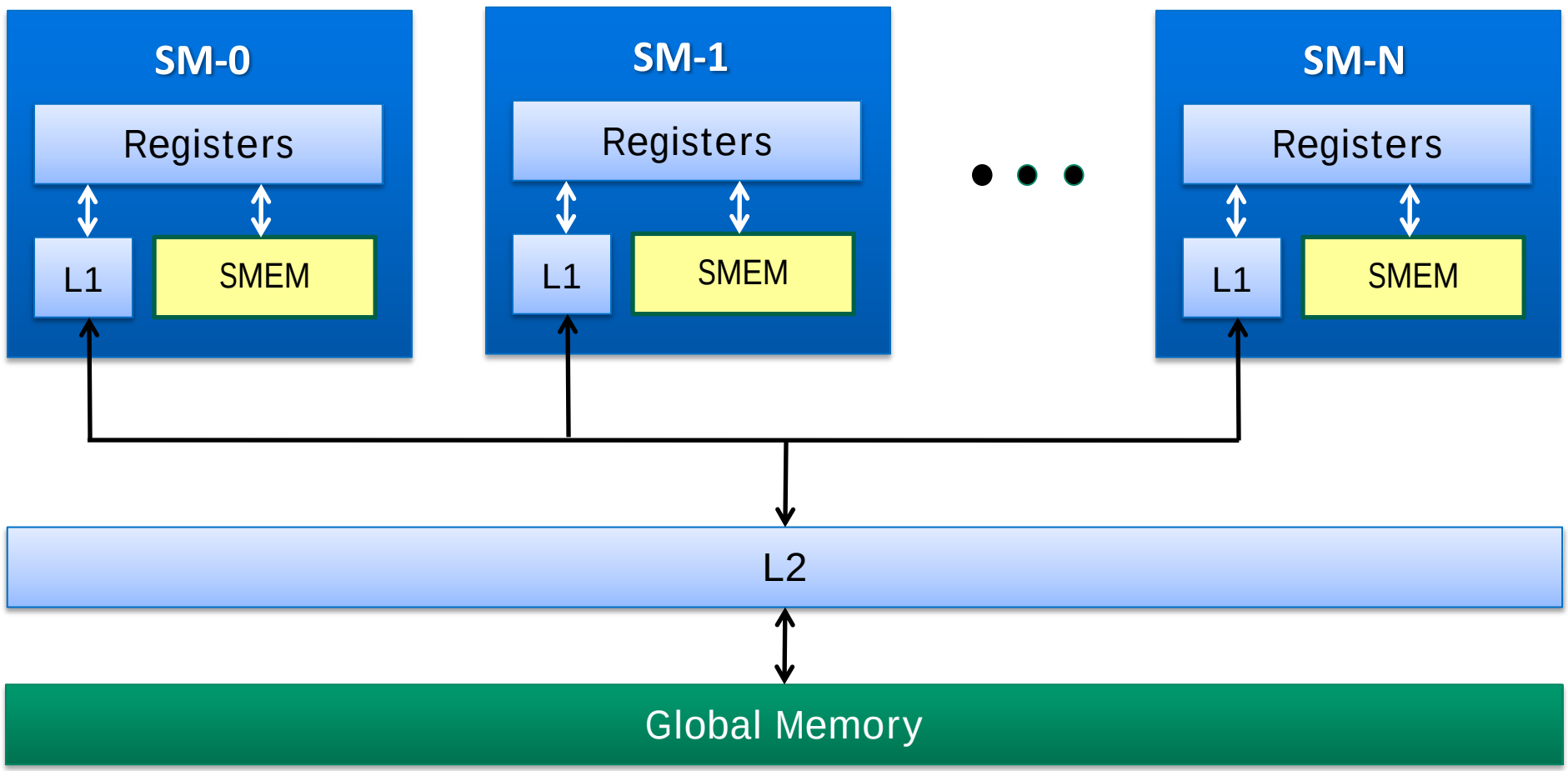
MEMORY HIERARCHY REVIEW

- L2
  - All accesses to global memory go through L2, including copies to/from CPU host
- Global memory
  - Accessible by all threads as well as host (CPU)
  - High latency (hundreds of cycles)
  - Throughput: up to ~900 GB/s (Volta V100)

MEMORY ARCHITECTURE



MEMORY HIERARCHY REVIEW



GMEM OPERATIONS

- Loads:
  - Caching
    - Default mode
    - Attempts to hit in L1, then L2, then GMEM
    - Load granularity is 128-byte line
- Stores:
  - Invalidate L1, write-back for L2

GMEM OPERATIONS

- Loads:
  - Non-caching
    - Compile with –Xptxas –dlcm=cg option to nvcc
    - Attempts to hit in L2, then GMEM

      Do not hit in L1, invalidate the line if it's in L1 already
    - Load granularity is 32-bytes

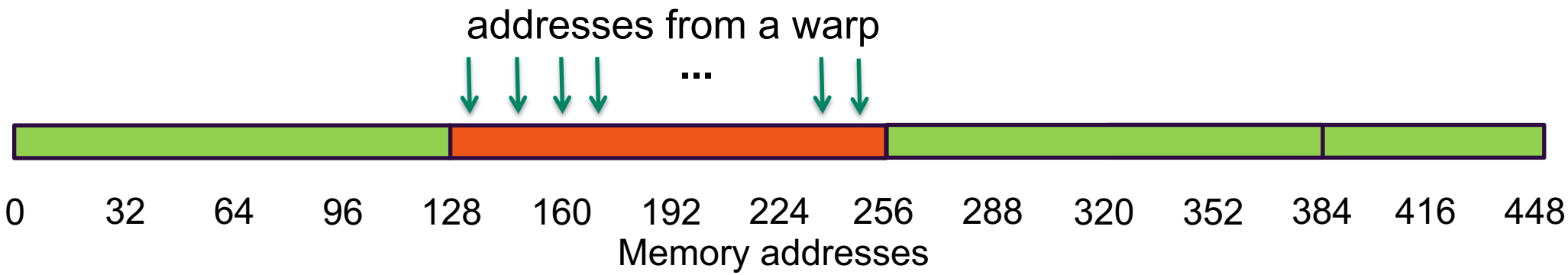
We won't spend much time with non-caching loads in this training session

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

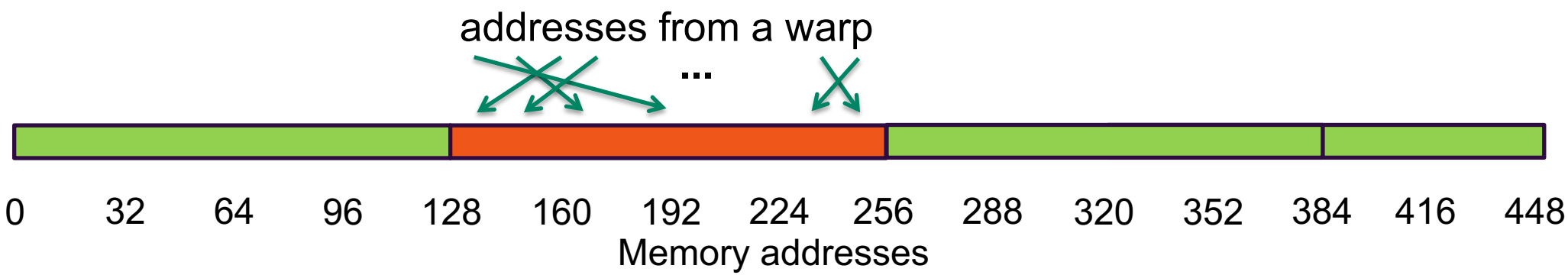
CACHING LOAD

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - int c = a[idx];



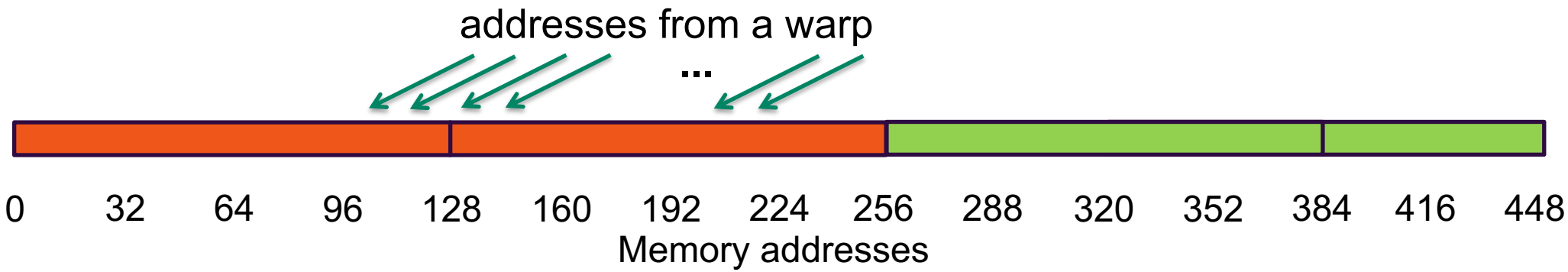
CACHING LOAD

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - int c = a[rand()%warpSize];



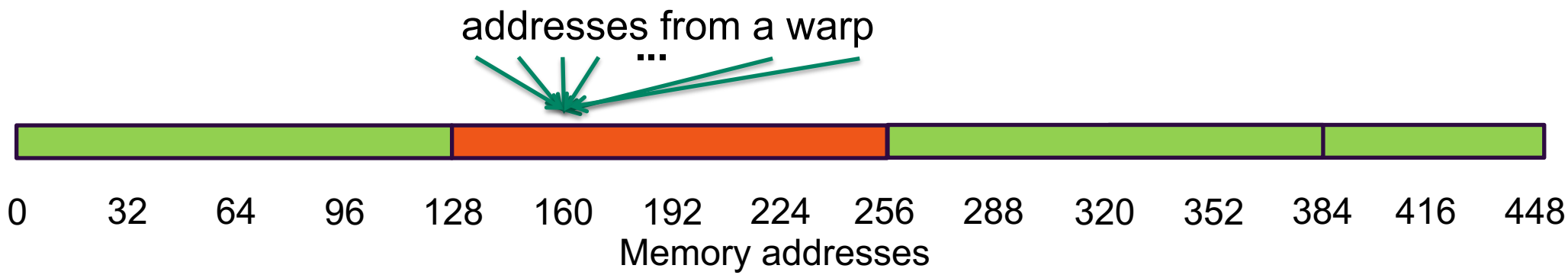
CACHING LOAD

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%
  - int c = a[idx-2];



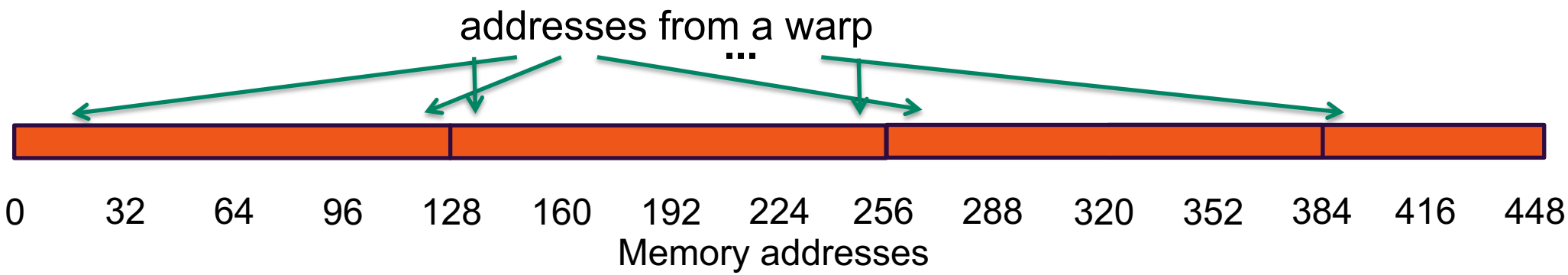
CACHING LOAD

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%
  - int c = a[40];



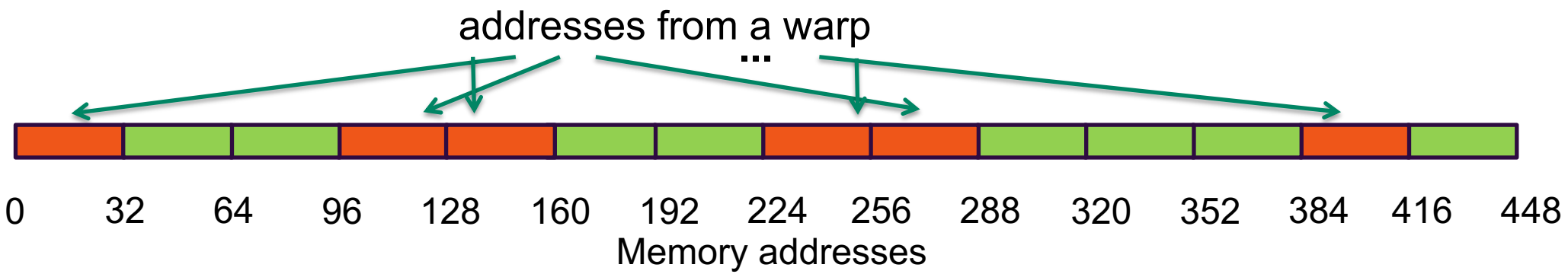
CACHING LOAD

- Warp requests 32 scattered 4-byte words
- Addresses fall within N cache-lines
  - Warp needs 128 bytes
  - N*128 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*128) (3.125% worst case N=32)
  - int c = a[rand()];



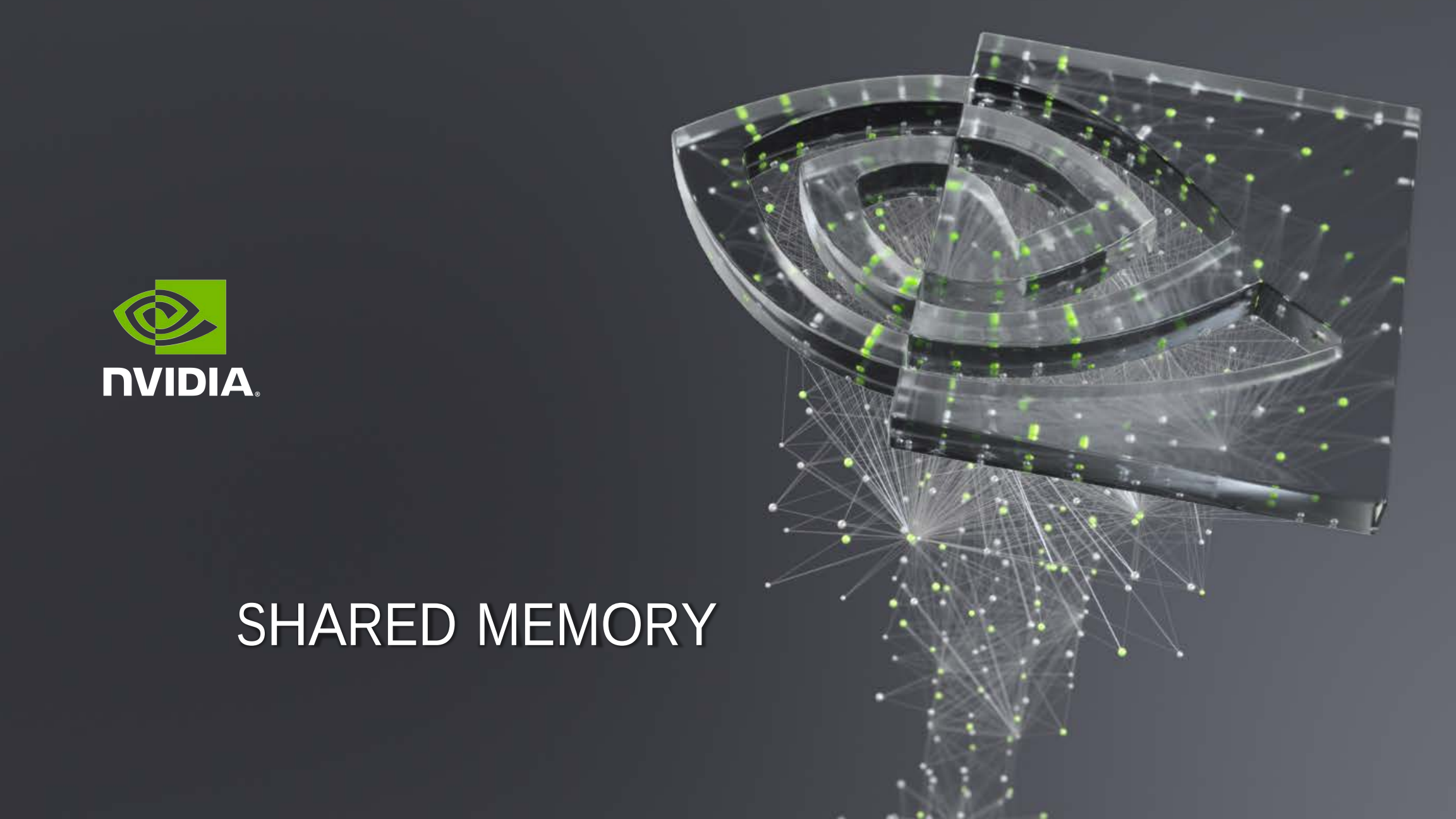
NON-CACHING LOAD

- Warp requests 32 scattered 4-byte words
- Addresses fall within N segments
  - Warp needs 128 bytes
  - N*32 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*32) (12.5% worst case N = 32)
  - int c = a[rand()]; –Xptxas –dlcm=cg



GMEM OPTIMIZATION GUIDELINES

- Strive for perfect coalescing
  - (Align starting address - may require padding)
  - A warp should access within a contiguous region
- Have enough concurrent accesses to saturate the bus
  - Process several elements per thread
    - Multiple loads get pipelined
    - Indexing calculations can often be reused
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)
- Use all the caches!



SHARED MEMORY

SHARED MEMORY

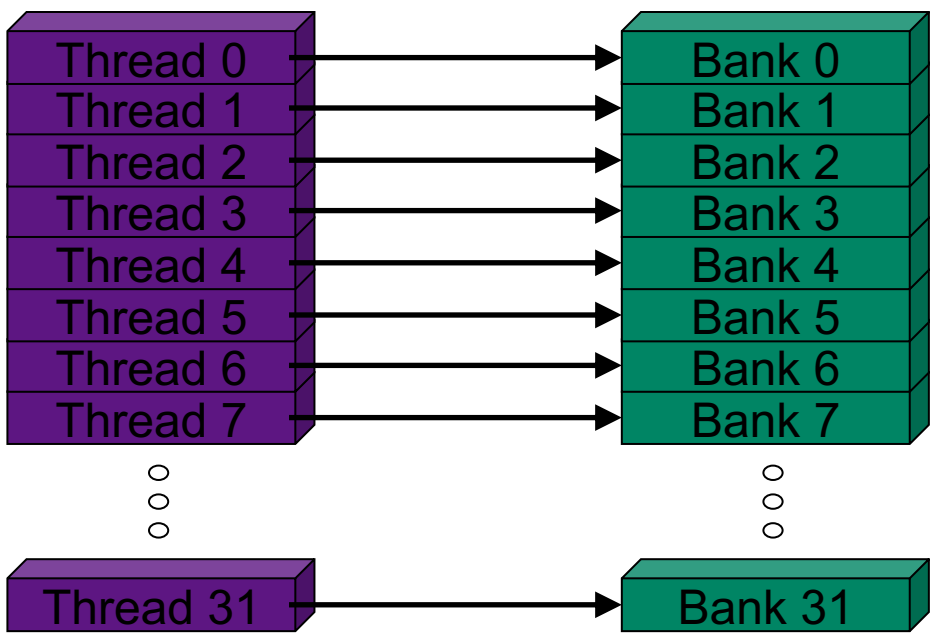
- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Organization:
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks

SHARED MEMORY

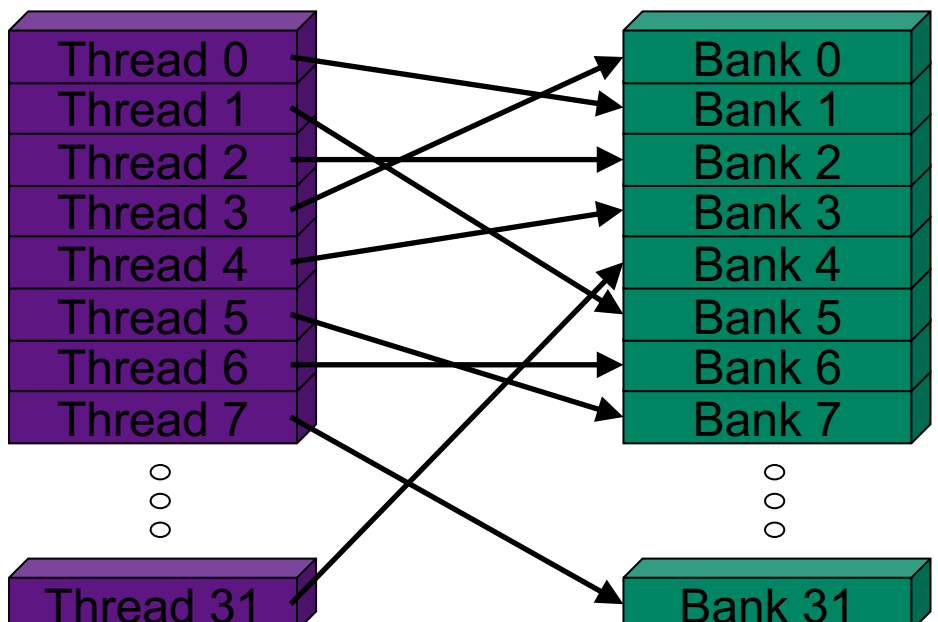
- Performance:
  - Typically: 4 bytes per bank per 1 or 2 clocks per multiprocessor
  - shared accesses are issued per 32 threads (warp)
  - serialization: if *N* threads of 32 access different 4-byte words in the same bank, *N* accesses are executed serially
  - multicast: *N* threads access the same word in one fetch
    - Could be different bytes within the same word

BANK ADDRESSING EXAMPLES

No Bank Conflicts

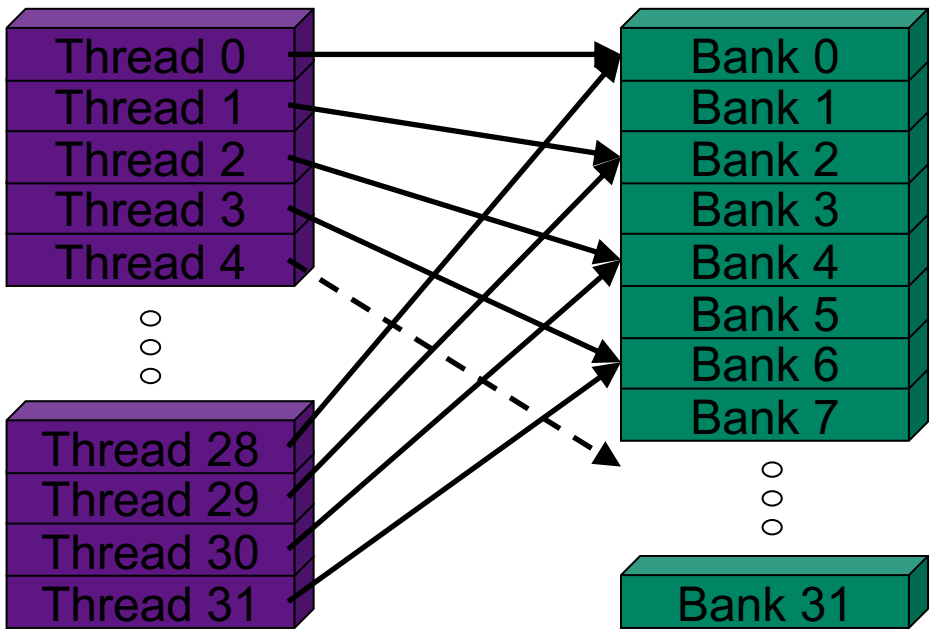


No Bank Conflicts

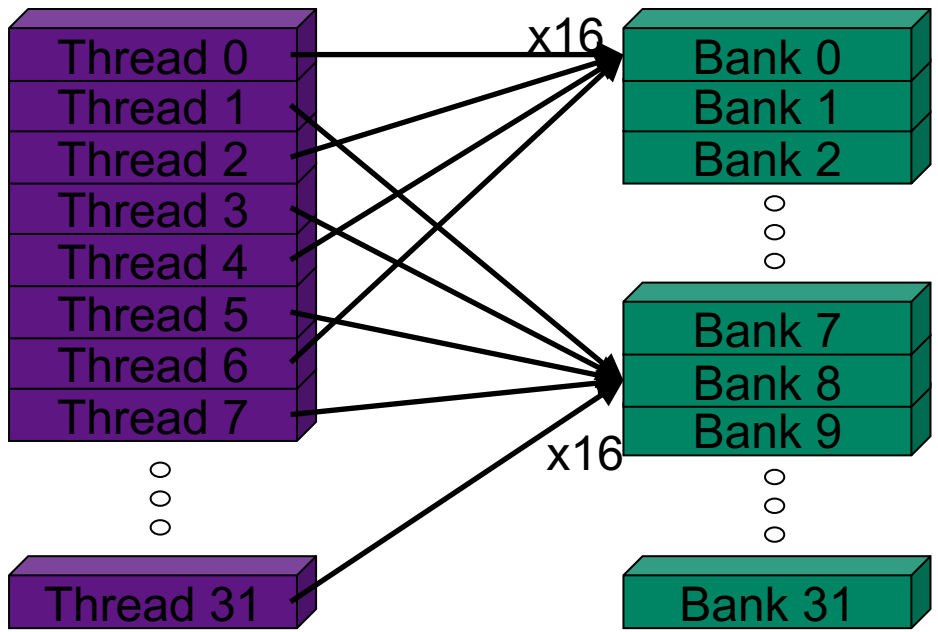


BANK ADDRESSING EXAMPLES

2-way Bank Conflicts

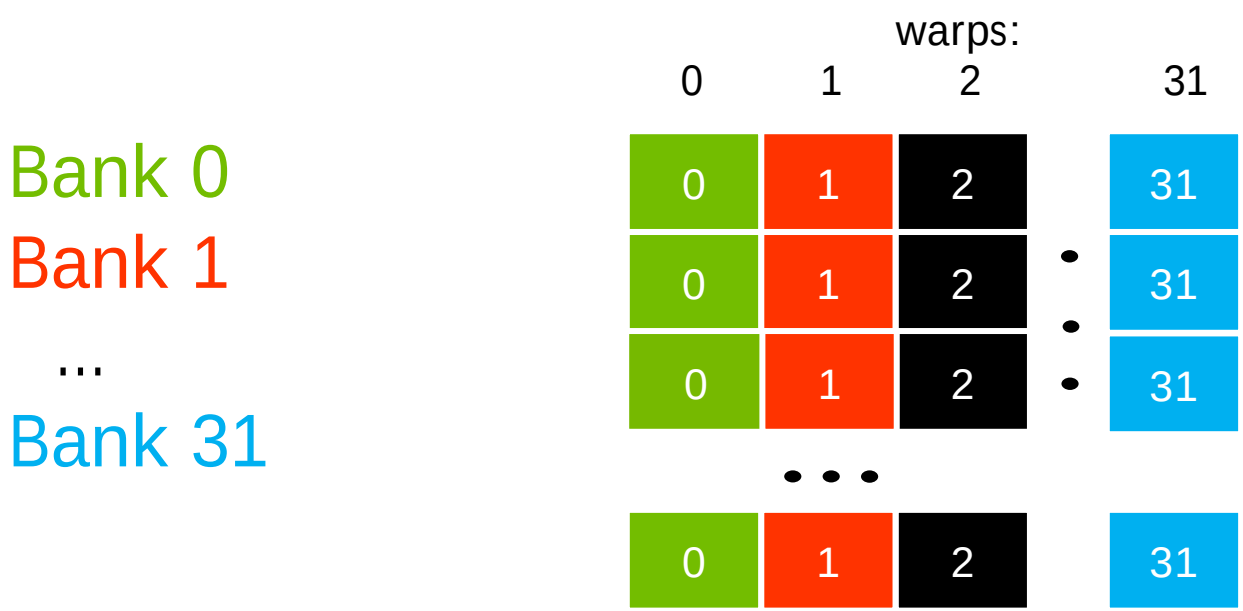


16-way Bank Conflicts



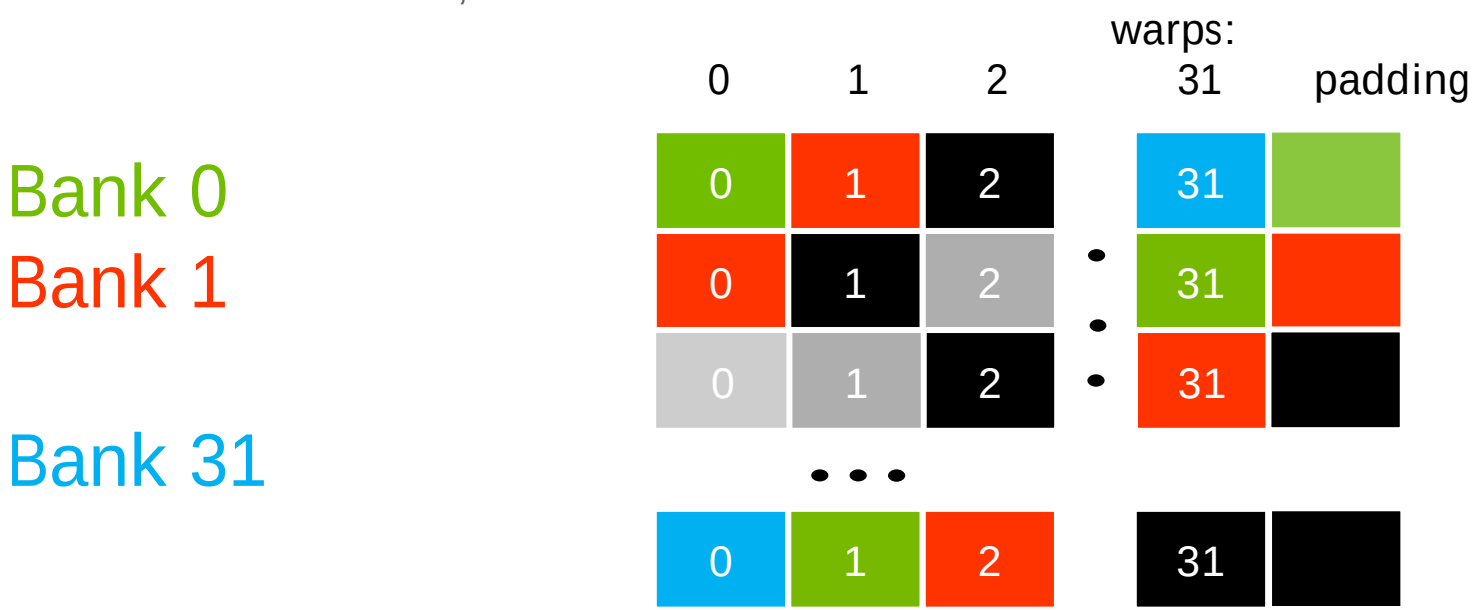
SHARED MEMORY: AVOIDING BANK CONFLICTS

- 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



SUMMARY

- Kernel Launch Configuration:
  - Launch enough threads per SM to hide latency
  - Launch enough threadblocks to load the GPU
- Global memory:
  - Maximize throughput (GPU has lots of bandwidth, use it effectively)
- Use shared memory when applicable (over 1 TB/s bandwidth)
- Use analysis/profiling when optimizing:
  - "Analysis-driven Optimization" (future session)

FUTURE SESSIONS

- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

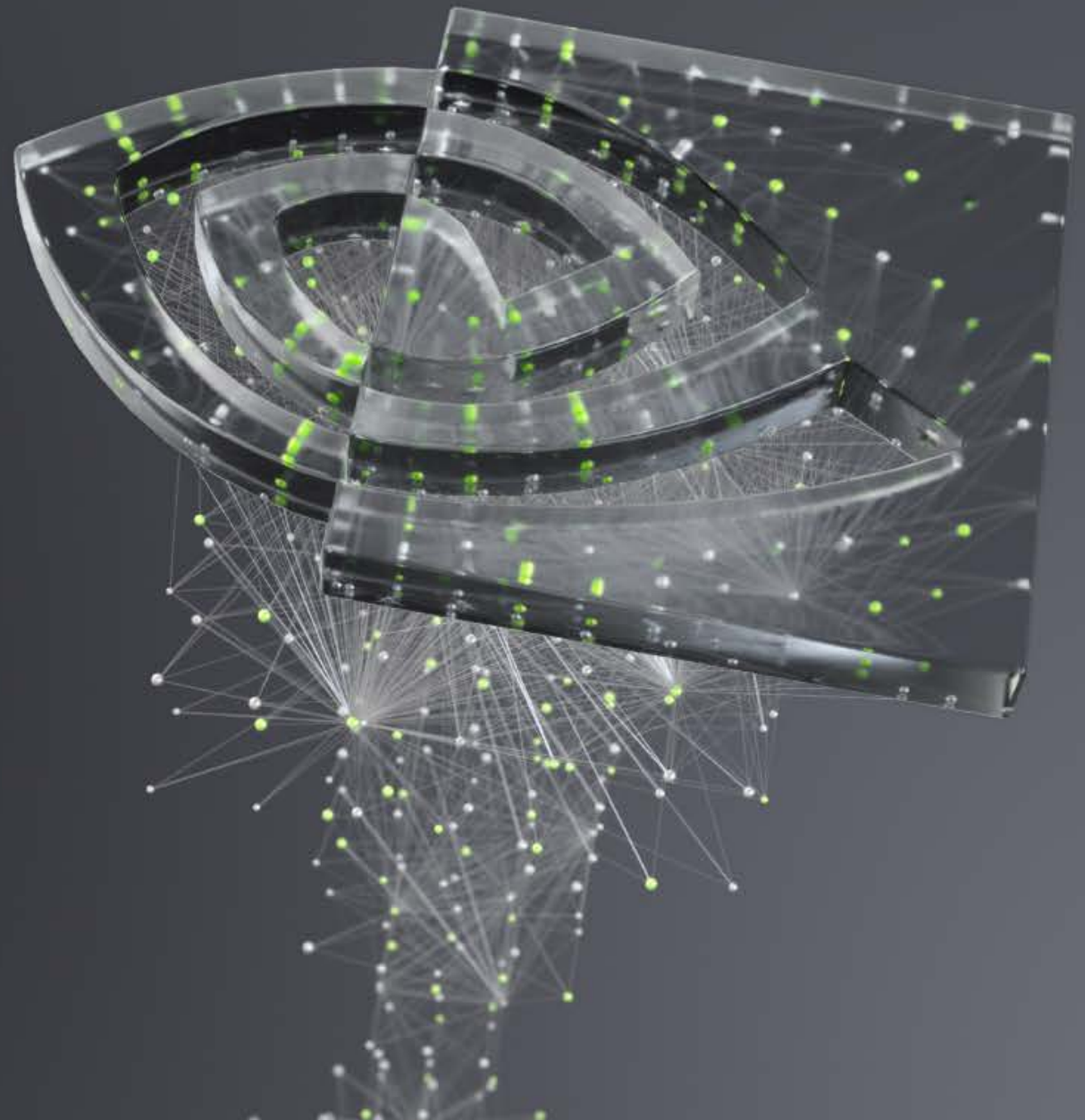
FURTHER STUDY

- Optimization in-depth:
  - http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf
- Analysis-Driven Optimization:
  - http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf
- CUDA Best Practices Guide:
  - https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
- CUDA Tuning Guides:
  - https://docs.nvidia.com/cuda/index.html#programming-guides

  (Kepler/Maxwell/Pascal/Volta)

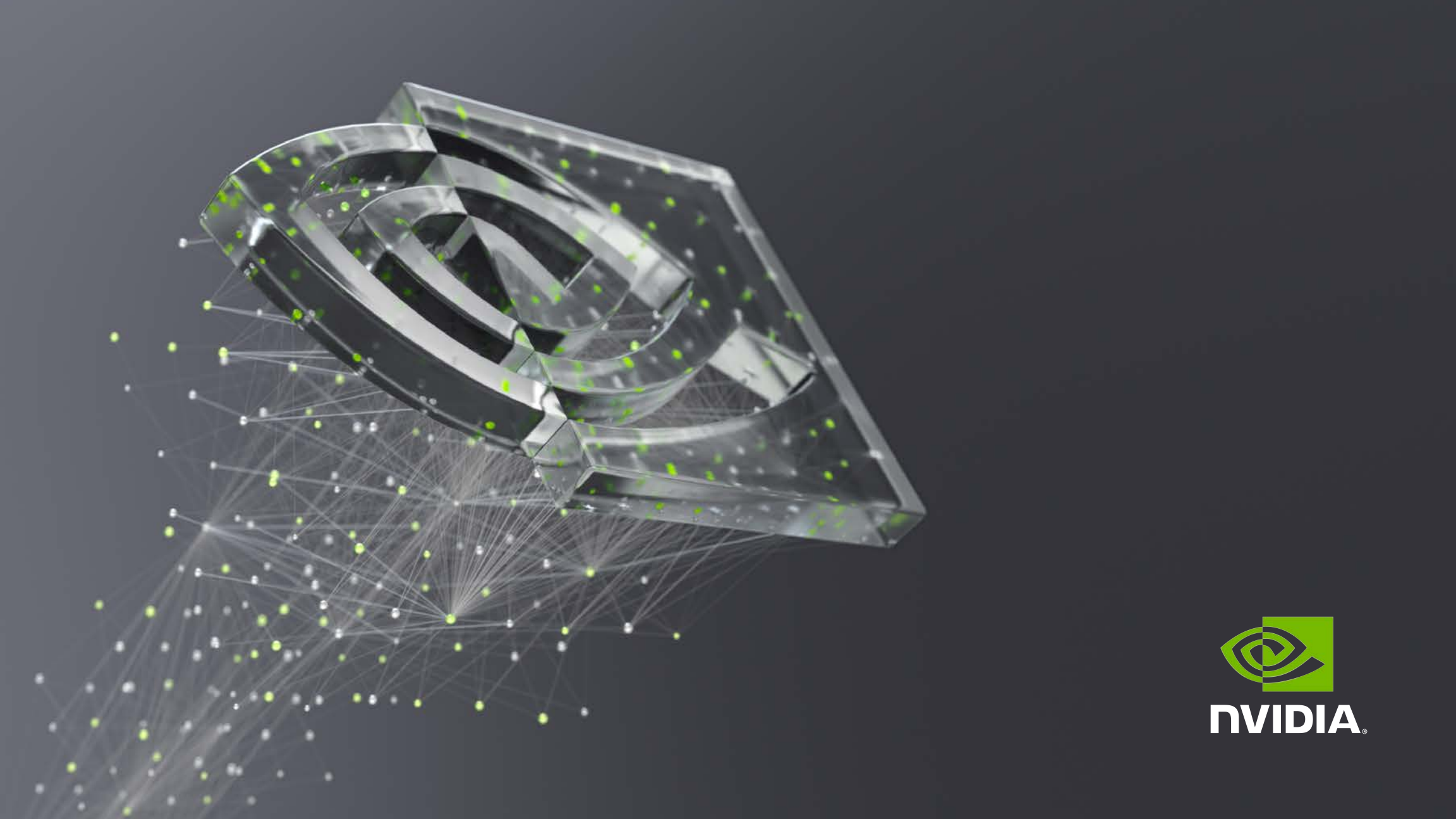
HOMEWORK

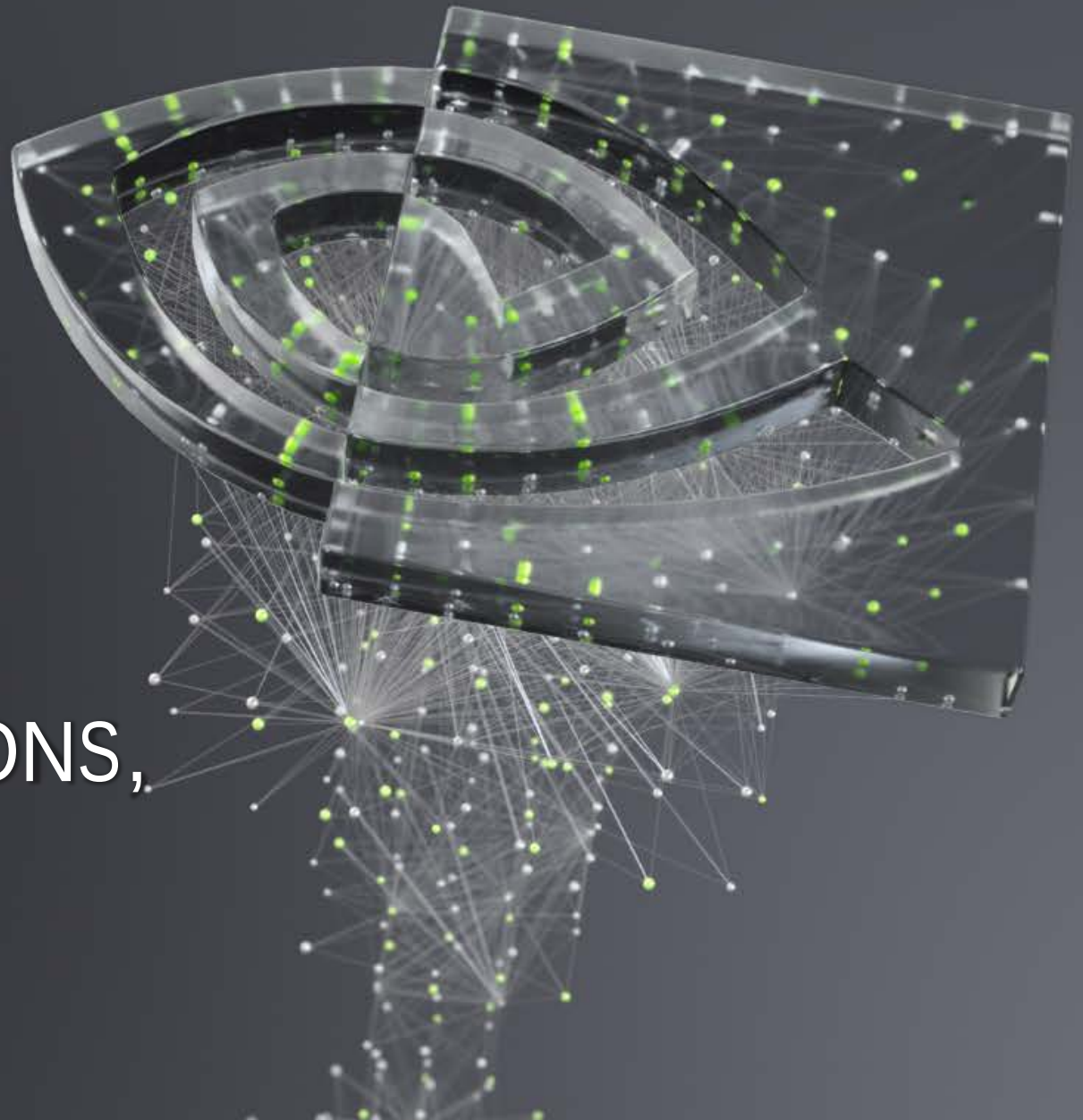
- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw4/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



NVIDIA

QUESTIONS?


NVIDIA



ATOMICS, REDUCTIONS, WARP SHUFFLE

Bob Crovella, 5/13/2020

AGENDA

- Transformations vs. Reductions, Thread Strategy
- Atomics, Atomic Reductions
- Atomic Tips and Tricks
- Classical Parallel Reduction
- Parallel Reduction + Atomics
- Warp Shuffle, Reduction with Warp Shuffle
- Other Warp Shuffle Uses
- Further Study
- Homework


ATOMICS

MOTIVATING EXAMPLE

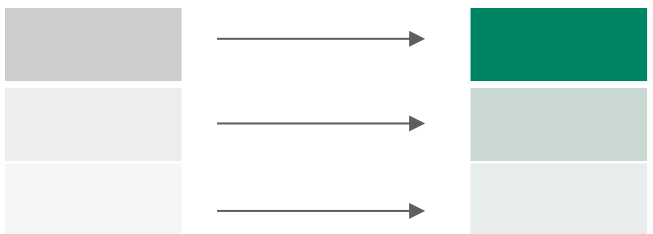
Sum - reduction

```
const int size = 100000;
float a[size] = {…};
float sum = 0;
for (int i = 0; i < size; i++) sum += a[i];
```

-> sum variable contains the sum of all the elements of array a

TRANSFORMATION VS. REDUCTION

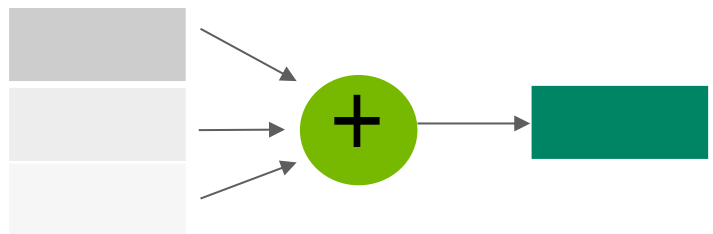
May guide the thread strategy: what will each thread do?



Transformation:

e.g. c[i] = a[i] + 10;

Thread strategy: one thread per output point



Reduction:

e.g. *c = Σ a[i]

Thread strategy: ??

REDUCTION: NAÏVE THREAD STRATEGY

One thread per input point

*c += a[i];

(Doesn't work.) Actual code the GPU executes:

LD R2, a[i] (thread independent)

LD R1, c (READ)

ADD R3, R1, R2 (MODIFY)

ST c, R3 (WRITE)

But every thread is trying to do this, potentially at the same time

The CUDA programming model does not enforce any order of thread execution

ATOMICS TO THE RESCUE

indivisible READ-MODIFY-WRITE

atomicAdd(&c, a[i]); https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

LD R2, a[i] (thread independent)

LD R1, c	(READ)
ADD R3, R1, R2	(MODIFY)
ST R3, c	(WRITE)

→ Becomes one indivisible operation/instruction:

RED.E.ADD.F32.FTZ.RN [c], R2;

Facilitated by special hardware in the L2 cache

May have performance implications

OTHER ATOMICS

- atomicMax/Min – choose the max (or min)
- atomicAdd/Sub – add to (or subtract from)
- atomicInc/Dec – increment (or decrement) and account for rollover/underflow
- atomicExch/CAS – swap values, or conditionally swap values
- atomicAnd/Or/Xor – bitwise ops
- atomics have different datatypes they can work on (e.g. int, unsigned, float, etc.)
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

ATOMIC TIPS AND TRICKS

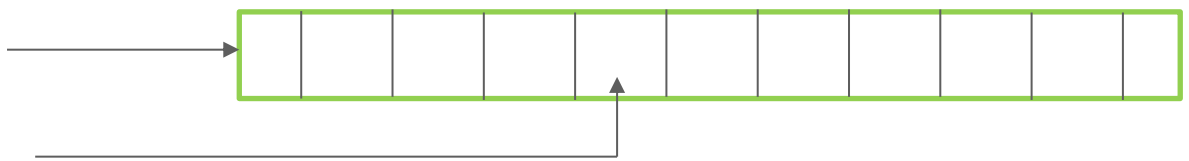
Determine my place in an order

- Could be used to determine next work item, queue slot, etc.
- int my_position = atomicAdd(order, 1);
- Most atomics return a value that is the “old” value that was in the location receiving the atomic update.

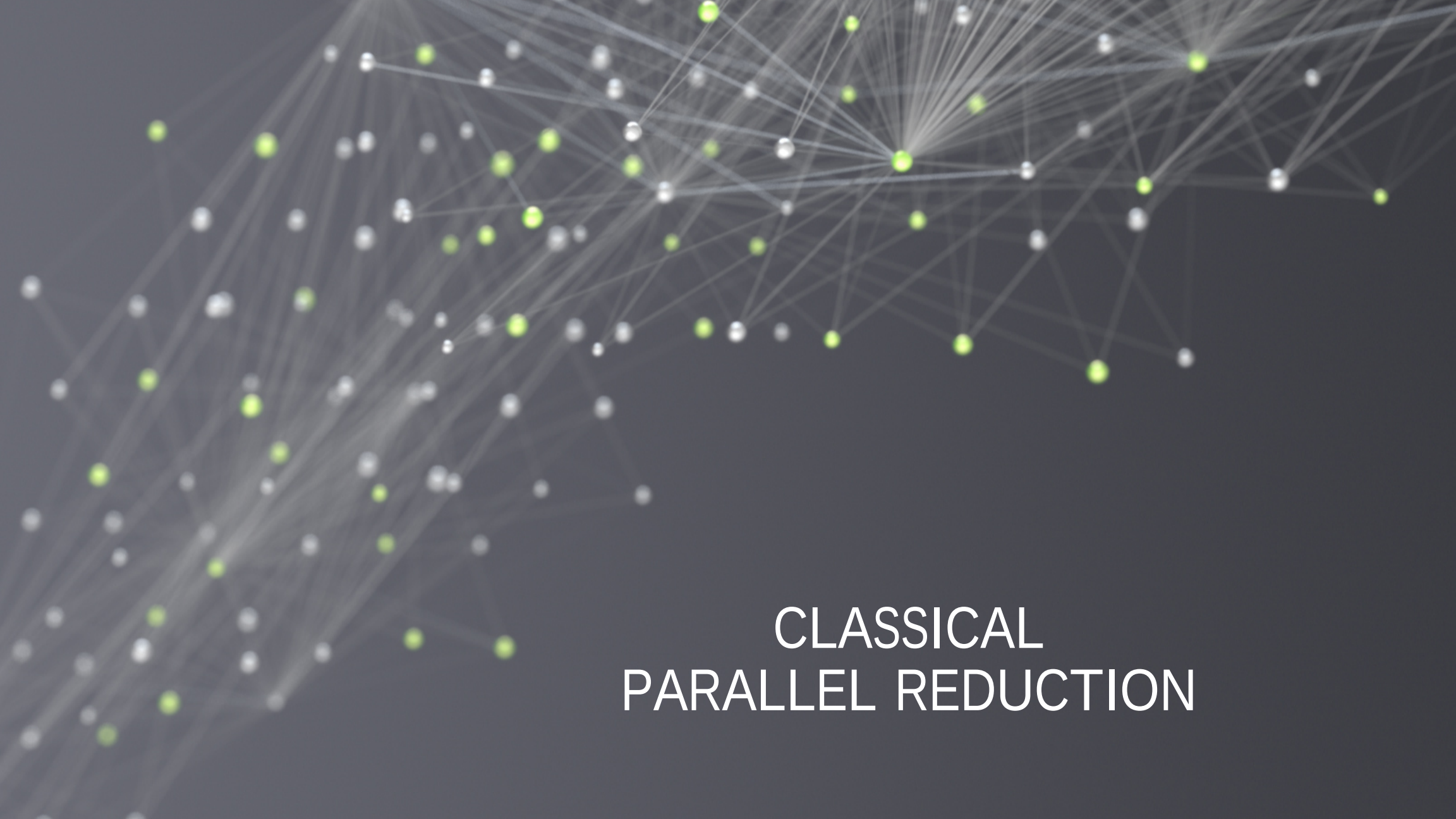
ATOMIC TIPS AND TRICKS

Reserve space in a buffer

- Each thread in my kernel may produce a variable amount of data. How to collect all of this in one buffer, in parallel?
- buffer_ptr:
- buffer_idx



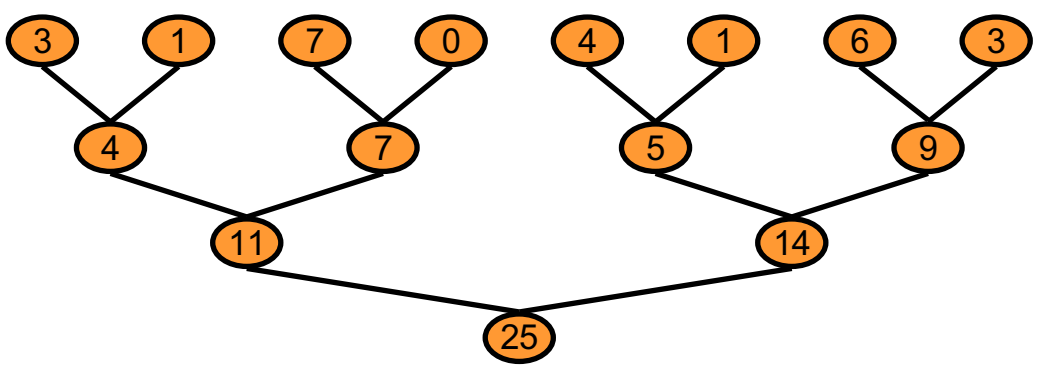
- int my_dsize = var;
- float local_buffer[my_dsize] = {…};
- int my_offset = atomicAdd(buffer_idx, my_dsize);
- // buffer_ptr+my_offset now points to the first reserved location, of length my_dsize
- memcpy(buffer_ptr+my_offset, local_buffer, my_dsize*sizeof(float));


CLASSICAL
PARALLEL REDUCTION

THE CLASSICAL PARALLEL REDUCTION

Atomics don't run at full memory bandwidth…

- We would like a reduction method that is not limited by atomic throughput
- We would like to effectively use all threads, as much as possible
- Parallel reduction is a common and important data parallel primitive
- Naïve implementations will often run into bottlenecks
- Basic methodology is a tree-based approach:

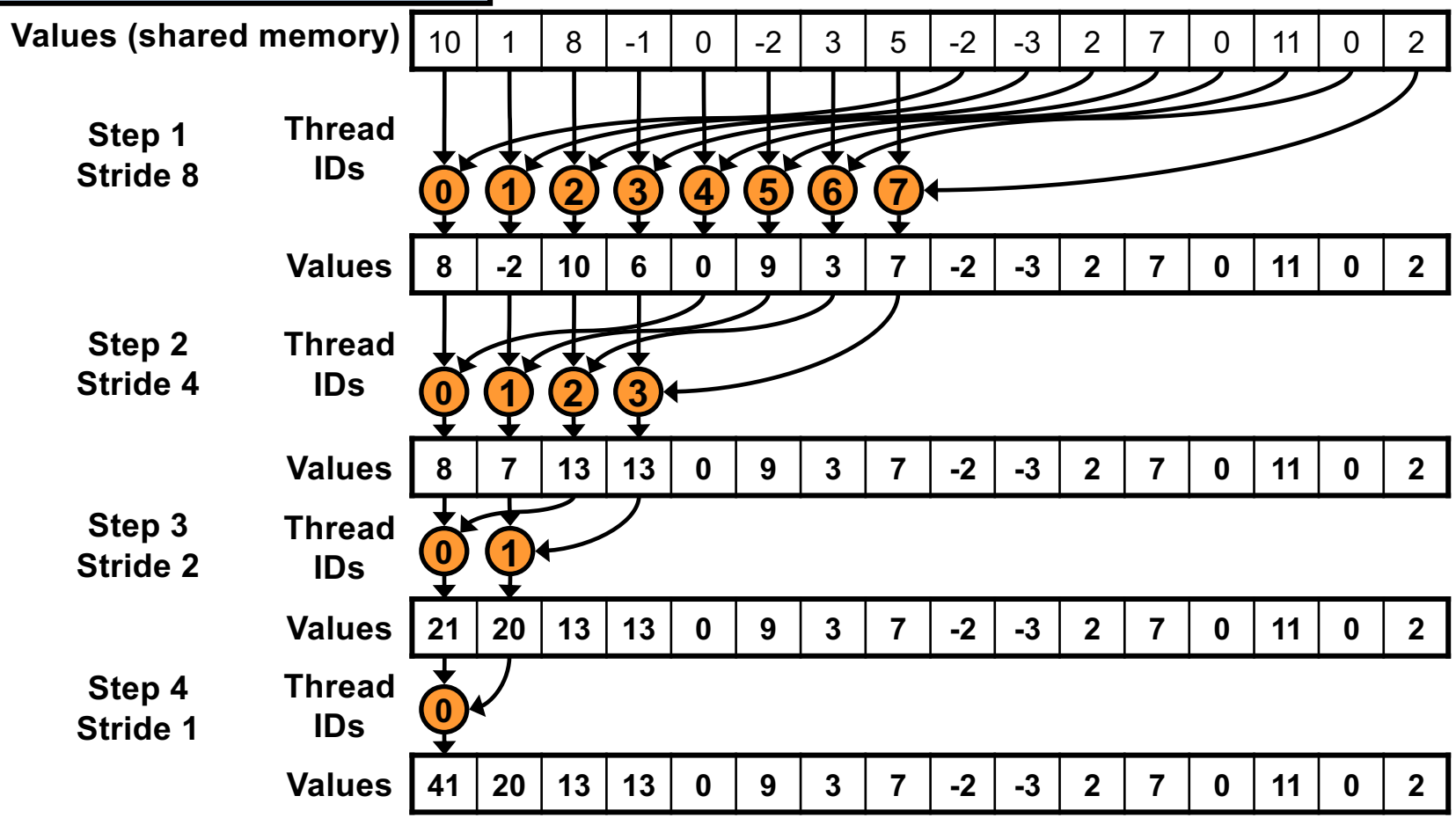


PROBLEM: GLOBAL SYNCHRONIZATION

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- One possible solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has low SW overhead (but not zero)
- Other possible solutions:
  - Use atomics at the end of threadblock-level reduction
  - Use a threadblock-draining approach (see threadFenceReduction sample code)
  - Use cooperative groups – cooperative kernel launch

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
   if (tid < s) {
      sdata[tid] += sdata[tid + s]; }
   __syncthreads();  // outside the if-statement
   }
```

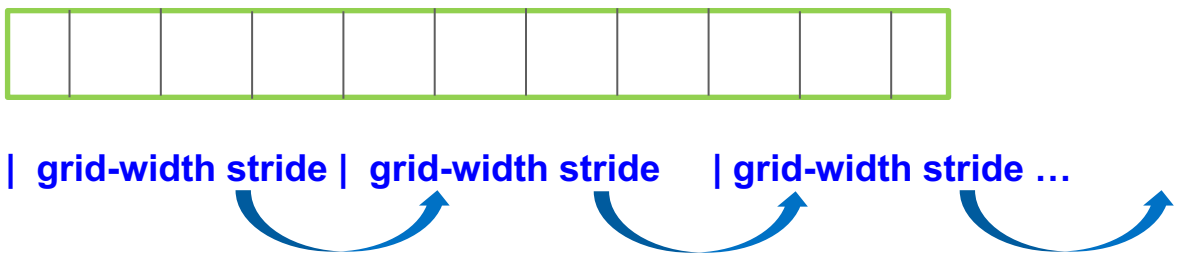
SEQUENTIAL ADDRESSING



Sequential addressing is bank-conflict free

DETOUR: GRID-STRIDE LOOPS

- We'd like to be able to design kernels that load and operate on arbitrary data sizes efficiently
- Want to maintain coalesced loads/stores, efficient use of shared memory
- Can also be used for ninja-level tuning – choose number of blocks sized to the GPU
- gdata[0..N-1]:




```
int idx = threadIdx.x+blockDim.x*blockIdx.x;
while (idx < N) {
  sdata[tid] += gdata[idx];
  idx += gridDim.x*blockDim.x;  // grid width
  }
```

PUTTING IT ALL TOGETHER

```
__global__ void reduce(float *gdata, float *out){
   __shared__ float sdata[BLOCK_SIZE];
   int tid = threadIdx.x;
   sdata[tid] = 0.0f;
   size_t idx = threadIdx.x+blockDim.x*blockIdx.x;

   while (idx < N) {  // grid stride loop to load data
      sdata[tid] += gdata[idx];
      idx += gridDim.x*blockDim.x;
      }

   for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
      __syncthreads();
      if (tid < s)  // parallel sweep reduction
         sdata[tid] += sdata[tid + s];
      }
   if (tid == 0) out[blockIdx.x] = sdata[0];
}
```

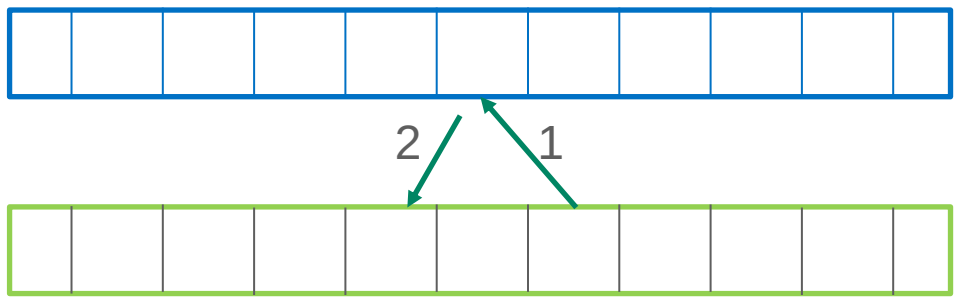
GETTING RID OF THE 2ND KERNEL CALL

```
__global__ void reduce_a(float *gdata, float *out){
   __shared__ float sdata[BLOCK_SIZE];
   int tid = threadIdx.x;
   sdata[tid] = 0.0f;
   size_t idx = threadIdx.x+blockDim.x*blockIdx.x;

   while (idx < N) {  // grid stride loop to load data
      sdata[tid] += gdata[idx];
      idx += gridDim.x*blockDim.x;
      }

   for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
      __syncthreads();
      if (tid < s)  // parallel sweep reduction
         sdata[tid] += sdata[tid + s];
      }
   if (tid == 0) atomicAdd(out, sdata[0]);
}
```

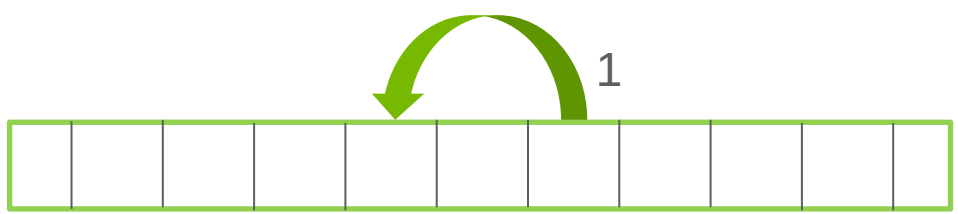

WARP SHUFFLE

INTER-THREAD COMMUNICATION: SO FAR

- Using shared memory:
- Threads:

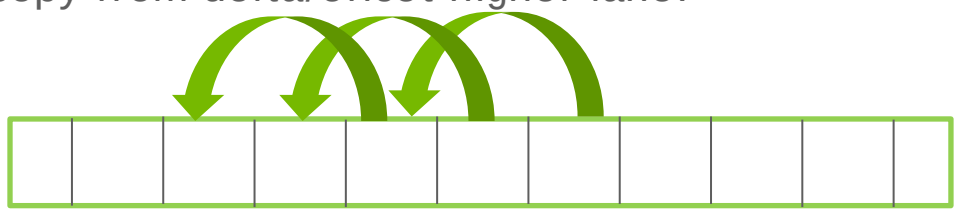


- Wouldn't this be convenient:
- Threads:



INTRODUCING WARP SHUFFLE

- Allows for intra-warp communication
- Various supported movement patterns:
  - __shfl_sync(): copy from lane ID (arbitrary pattern)
  - __shfl_xor_sync(): copy from calculated lane ID (calculated pattern)
  - __shfl_up_sync(): copy from delta/offset lower lane
  - __shfl_down_sync(): copy from delta/offset higher lane:



- Both source and destination threads in the warp must “participate”
- Sync “mask” used to identify and reconverge needed threads

WARP SHUFFLE REDUCTION

```
__global__ void reduce_ws(float *gdata, float *out){
  __shared__ float sdata[32];
  int tid = threadIdx.x;
  int idx = threadIdx.x+blockDim.x*blockIdx.x;
  float val = 0.0f;
  unsigned mask = 0xFFFFFFFFU;
  int lane = threadIdx.x % warpSize;
  int warpID = threadIdx.x / warpSize;
  while (idx < N) {  // grid stride loop to load
    val += gdata[idx];
    idx += gridDim.x*blockDim.x;
    }

// 1st warp-shuffle reduction
  for (int offset = warpSize/2; offset > 0; offset >>= 1)
    val += __shfl_down_sync(mask, val, offset);
  if (lane == 0) sdata[warpID] = val;
 __syncthreads(); // put warp results in shared mem

// hereafter, just warp 0
  if (warpID == 0){
// reload val from shared mem if warp existed
    val = (tid < blockDim.x/warpSize)?sdata[lane]:0;

// final warp-shuffle reduction
    for (int offset = warpSize/2; offset > 0; offset >>= 1)
      val += __shfl_down_sync(mask, val, offset);

    if  (tid == 0) atomicAdd(out, val);
  }
}
```

WARP SHUFFLE BENEFITS

- Reduce or eliminate shared memory usage
- Single instruction vs. 2 or more instructions
- Reduce level of explicit synchronization

WARP SHUFFLE TIPS AND TRICKS

What else can we do with it?

- Broadcast a value to all threads in the warp in a single instruction
- Perform a warp-level prefix sum
- Atomic aggregation

FUTURE SESSIONS

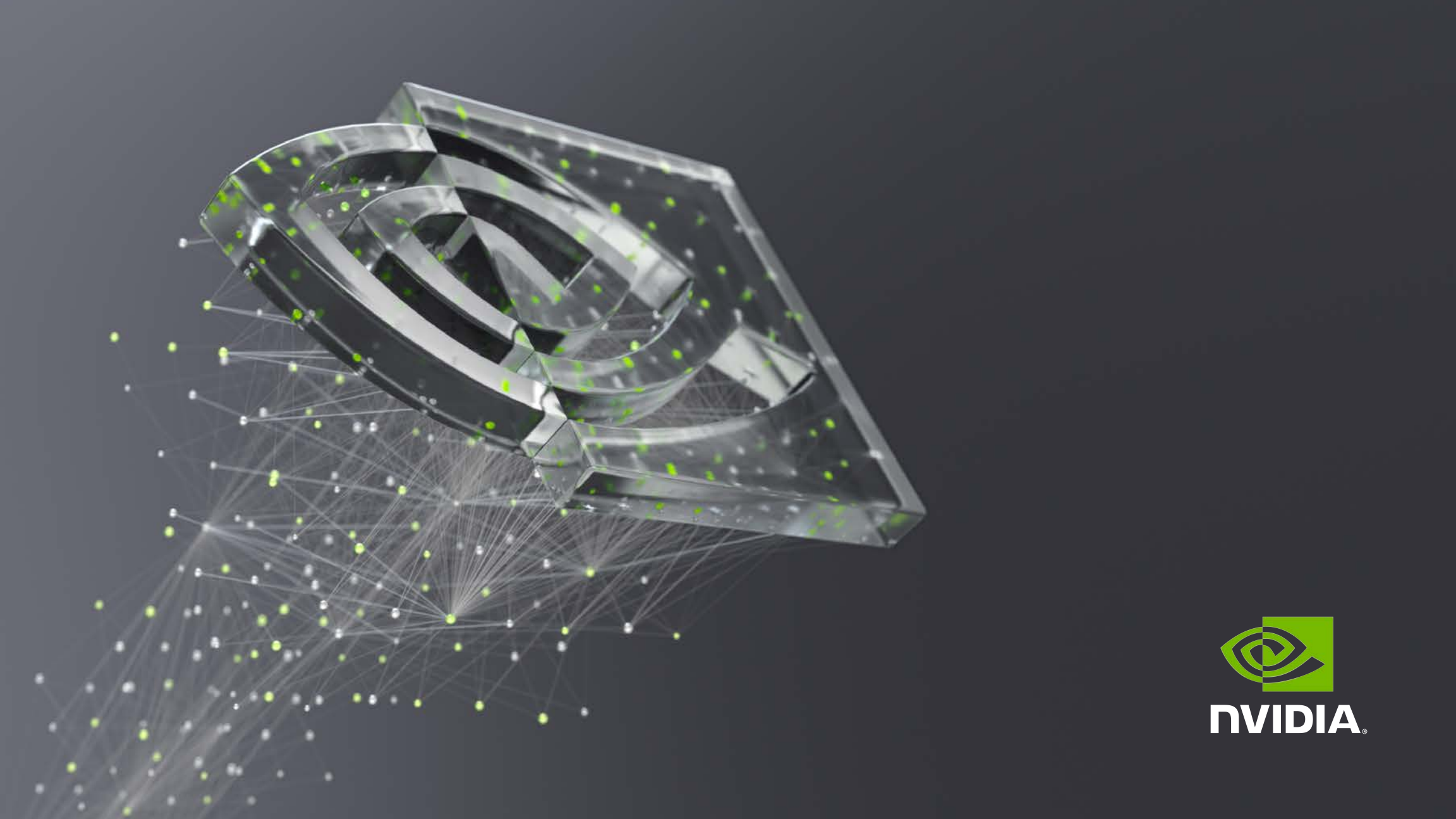
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

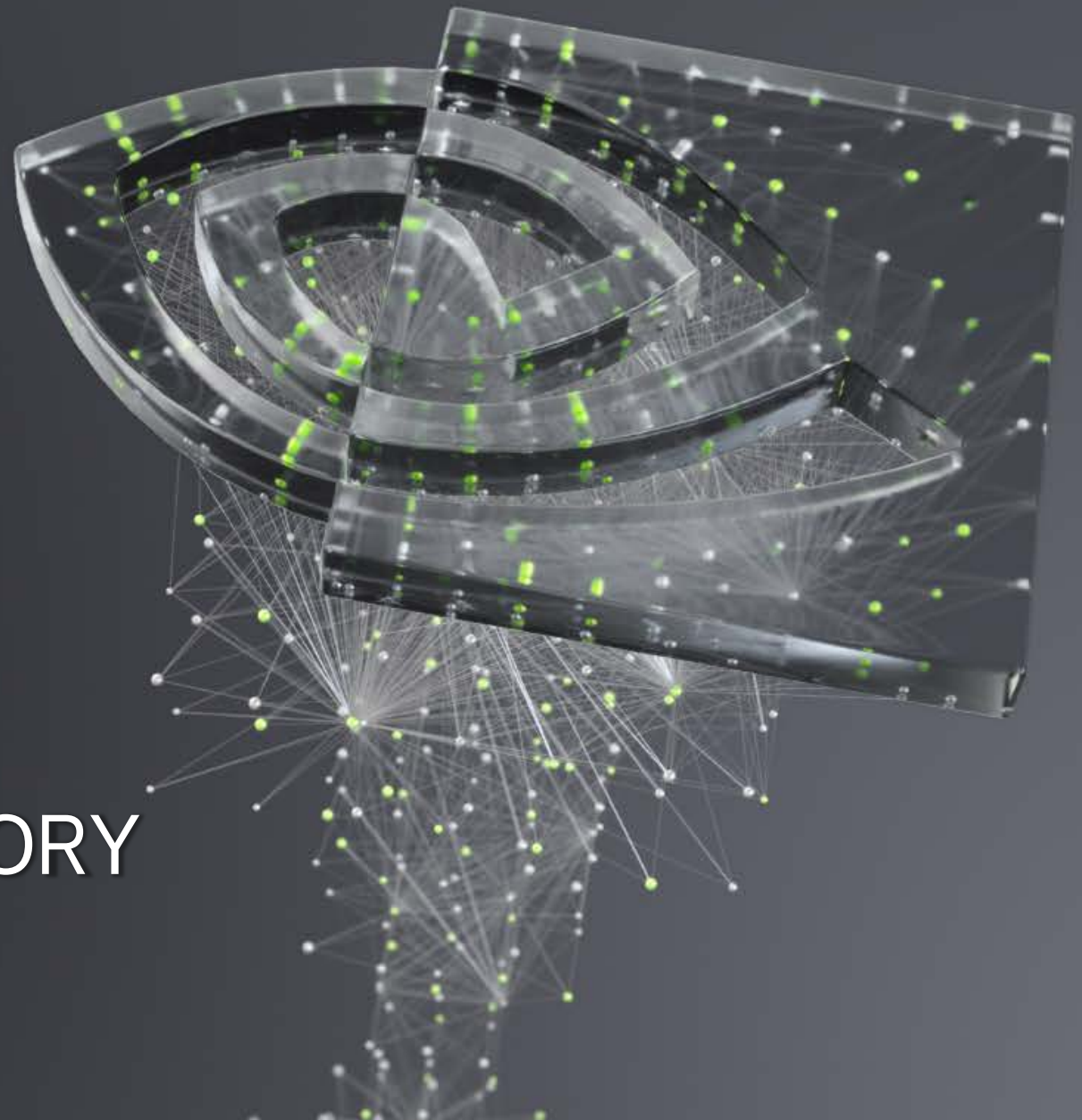
FURTHER STUDY

- Parallel reduction:
  - https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
- Warp-shuffle and reduction:
  - https://devblogs.nvidia.com/faster-parallel-reductions-kepler/
- CUDA Cooperative Groups:
  - https://devblogs.nvidia.com/cooperative-groups/
- Grid-stride loops:
  - https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/
- Floating point:
  - https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf
- CUDA Sample Codes:
  - Reduction, threadFenceReduction, reductionMultiBlockCG

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw5/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming


NVIDIA



CUDA UNIFIED MEMORY

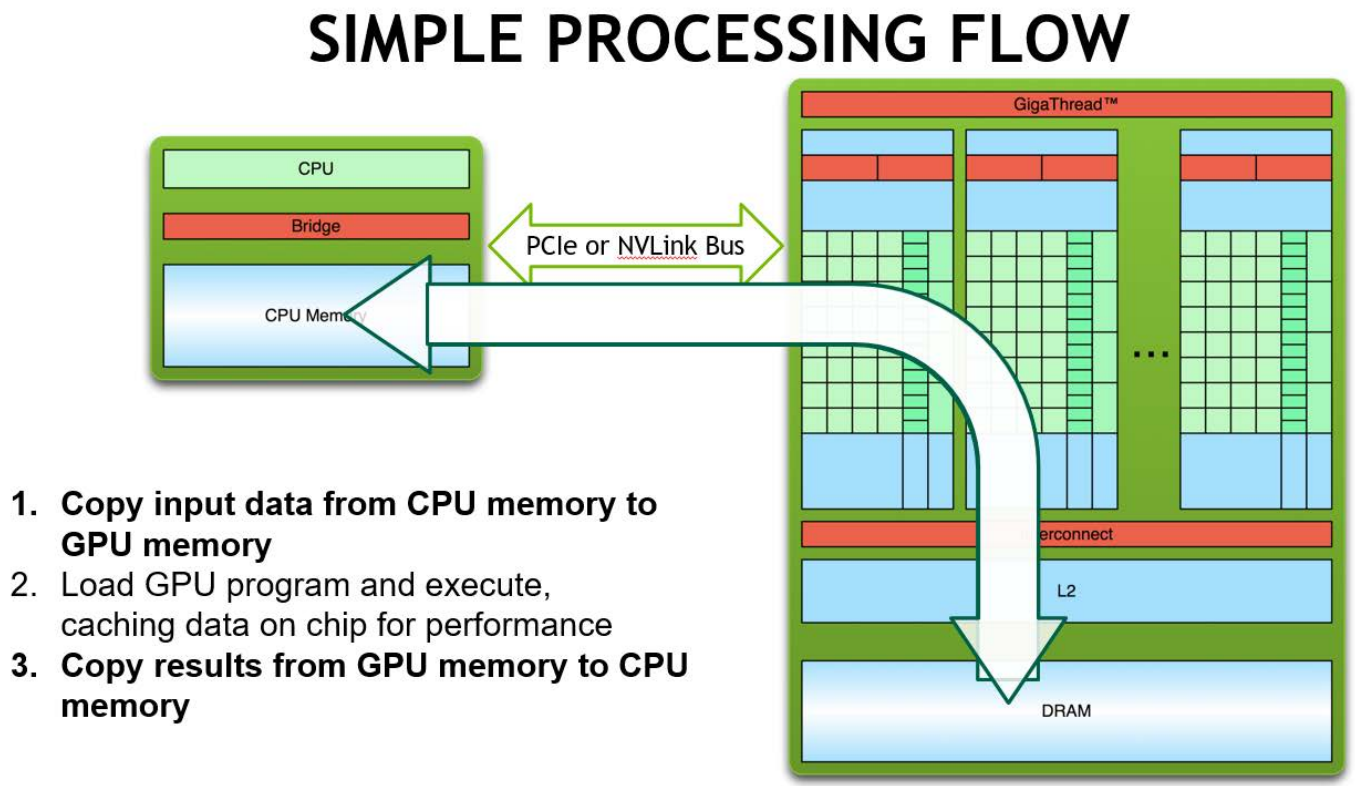
Bob Crovella, 6/18/2020

AGENDA

- Managed Memory - basic idea, objectives, benefits
- Demand-Paging, Oversubscription, Concurrency, Atomics
- Use Cases: Deep Copies, Linked Lists, C++ Objects, Graph Traversal
- Performance: Prefetching, Hints
- Multi-GPU Considerations
- Further Study
- Homework

THE CUDA 3-STEP PROCESSING SEQUENCE

Recall from Module 1…

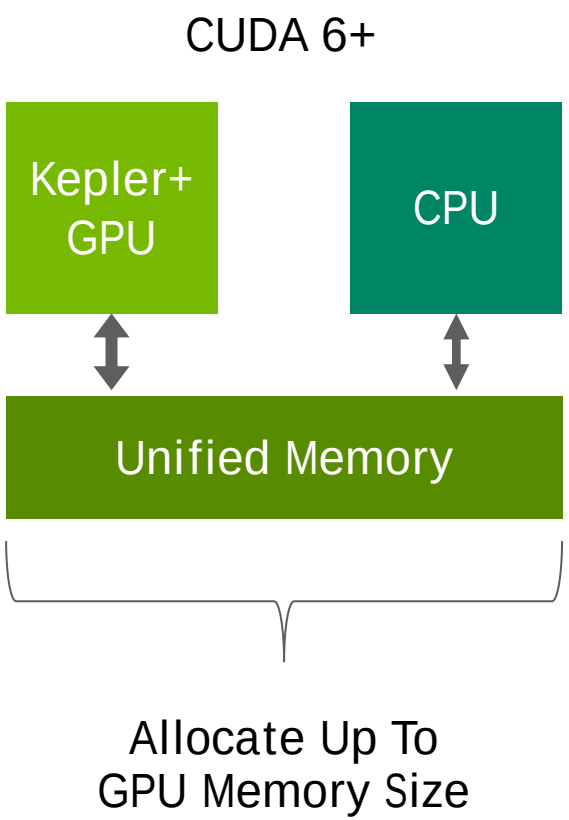


->Wouldn't it be nice if we didn't have to do (i.e. write the code for) steps 1 and 3?

INTRODUCING UNIFIED MEMORY WITH DEMAND PAGING

UNIFIED MEMORY

Reduce Developer Effort



Simpler Programming & Memory Model

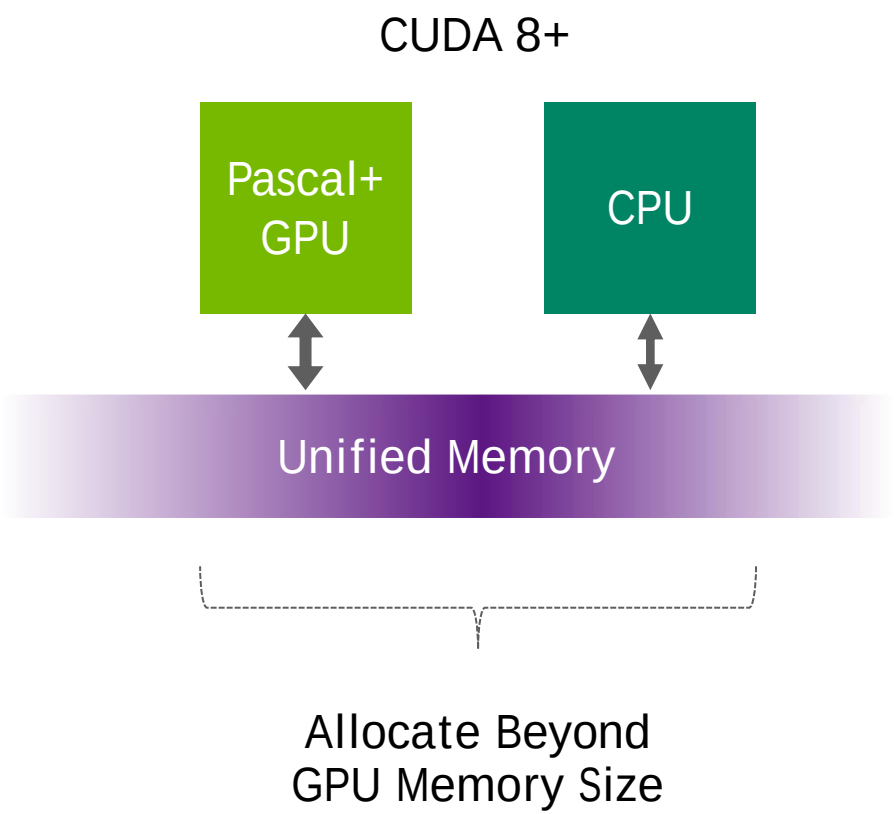
- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit* copy
- Simplifies code porting

Maintain Performance through Data Locality

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

CUDA 8+: UNIFIED MEMORY

Demand Paging For Pascal and Beyond



Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Simpler Data Accesss	CPU/GPU Data coherence Unified memory atomic operations
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API

SIMPLIFIED MEMORY MANAGEMENT CODE

CPU Code

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

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

Ordinary CUDA Code

```
void sortfile(FILE *fp, int N) {
  char *data, *d_data;
  data = (char *)malloc(N);
  cudaMalloc(&d_data, N);
  fread(data, 1, N, fp);
  cudaMemcpy(d_data, data, N, …); // 1
  qsort<<<...>>>(data,N,1,compare); // 2
  cudaMemcpy(data, d_data, N, …); // 3

  use_data(data);
  cudaFree(d_data);
  free(data);
}
```

SIMPLIFIED MEMORY MANAGEMENT CODE

CPU Code

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

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA Code with Unified Memory

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

  fread(data, 1, N, fp);

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

  use_data(data);

  cudaFree(data);
}
```

UNIFIED MEMORY EXAMPLE

With On-Demand Paging

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}


void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

HOW UNIFIED MEMORY WORKS ON PASCAL+

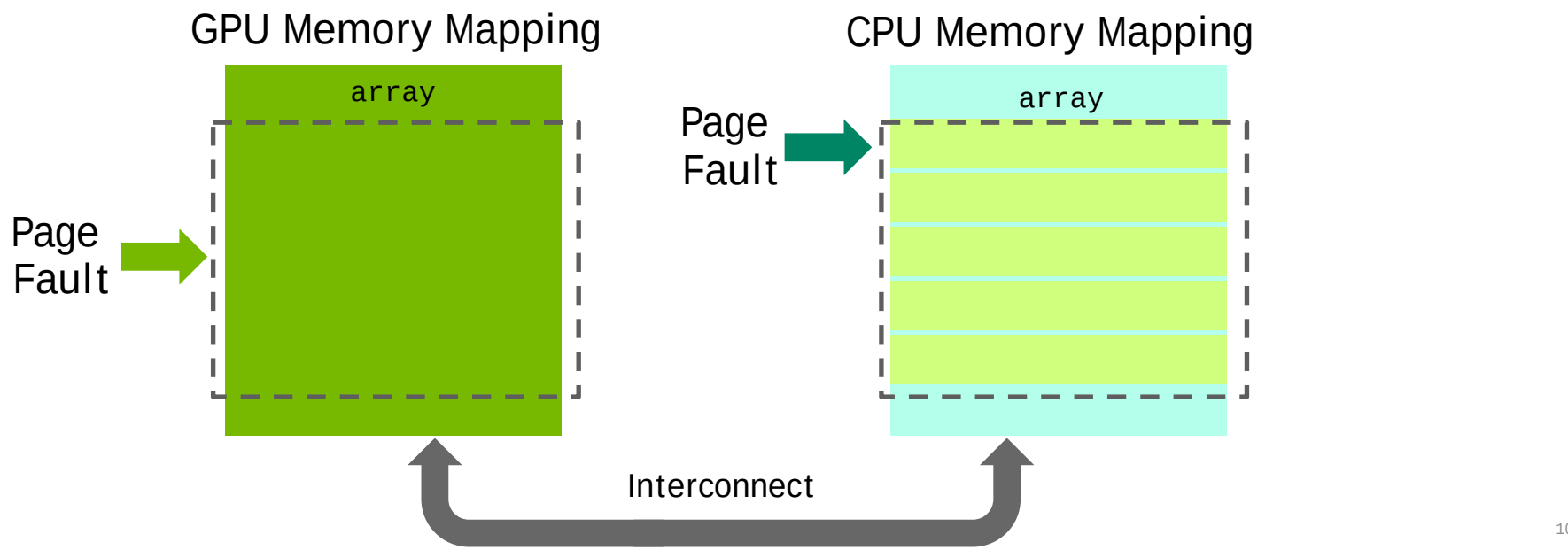
Servicing CPU *and* GPU Page Faults

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



ASIDE: PRE-PASCAL UM REGIME

Summary

- In effect if your device is prior to Pascal (Jetson is a special case)
- In effect if you are on windows OS (CUDA 9.x +).
- Managed data is moved en-masse at point of kernel launch (even data that your kernel may not appear to explicitly touch)
- After a kernel launch, cudaDeviceSynchronize() triggers the runtime to make data available to CPU code again
- No concurrent access, no on-demand migration to GPU, no oversubscription
- Just use cudaMallocManaged() where you would use malloc(), or new
- Use cudaFree() instead of free(), or delete

UNIFIED MEMORY ON PASCAL+

GPU Memory Oversubscription

```
void foo() {
  // Assume GPU has 16 GB memory
  // Allocate 64 GB
  char *data;
  // be careful with size type:
  size_t size = 64ULL*1024*1024*1024;
  cudaMallocManaged(&data, size);
}
```

64 GB allocation

Pascal supports allocations where only a subset of pages reside on GPU. Pages can be migrated to the GPU on demand.

Fails on Kepler/Maxwell

UNIFIED MEMORY ON PASCAL+

Concurrent CPU/GPU Access to Managed Memory

```
__global__ void mykernel(char *data) {
  data[1] = ‘g’;
}

void foo() {
  char *data;
  cudaMallocManaged(&data, 2);

  mykernel<<<...>>>(data);
  // no synchronize here
  data[0] = ‘c’;

  cudaFree(data);
}
```

OK on Pascal+: just a page fault

Concurrent CPU access to ‘data’ on previous GPUs caused a fatal segmentation fault

Note that there may still be ordering issues or data visibility issues; UM concurrency does not provide any ordering or visibility guarantees, but see system-wide atomics

UNIFIED MEMORY ON PASCAL+

System-Wide Atomics

```
__global__ void mykernel(int *addr) {
  atomicAdd_system(addr, 10);
}

void foo() {
  int *addr;
  cudaMallocManaged(addr, 4);
  *addr = 0;

  mykernel<<<...>>>(addr);
  // cpu atomic:
  __sync_fetch_and_add(addr, 10);
}
```

Pascal enables system-wide atomics

- Direct support of atomics over NVLink
- Software-assisted over PCIe

System-wide atomics not available on Kepler / Maxwell

USE CASE: DEEP COPY

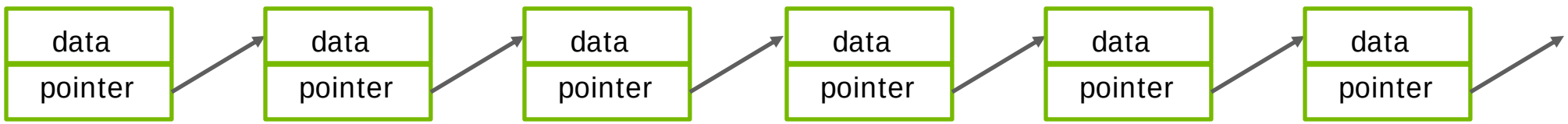
```
struct dataElem {
  int key;
  int len;
  char *name;
}
```

```
char buffer[len];
```

- Both entities (object and buffer) need to be transferred to the device
- Pointer in object needs to be “fixed” to point to new address on device for device copy of buffer

```
void launch(dataElem *elem, int N) { // an array of dataElem
  dataElem *d_elem;
  // Allocate storage for array of struct and copy array to device
  cudaMalloc(&d_elem, N*sizeof(dataElem));
  cudaMemcpy(d_elem, elem, N*sizeof(dataElem), cudaMemcpyHostToDevice);
  for (int i = 0; i < N; i++){ // allocate/fixup each buffer separately
    char *d_name;
    cudaMalloc(&d_name, elem[i].len);
    cudaMemcpy(d_name, elem[i].name, elem[i].len, cudaMemcpyHostToDevice);
    cudaMemcpy(&(d_elem[i].name), &d_name, sizeof(char *), cudaMemcpyHostToDevice);}
  // Finally we can launch our kernel
  Kernel<<< ... >>>(d_elem);}
```

USE CASE: LINKED LIST



- Similar to deep copy case
- Complex to code up the copy operation
- Unified Memory makes it trivial

USE CASE: C++ OBJECTS

Overloading new and delete

Overload new and delete in base class

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

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

inherit to build string class

```
// Deriving allows pass-by-reference
class umString : public Managed {
  int length;
  char *data;

public:
  // UM copy constructor allows
pass-by-value
  umString (const umString &s) {
    length = s.length;
    cudaMallocManaged(&data,
length);
    memcpy(data, s.data, length);
  }
};
```

USE CASE: C++ OBJECTS

Overloading new and delete

Inherit to build my class; embedded string

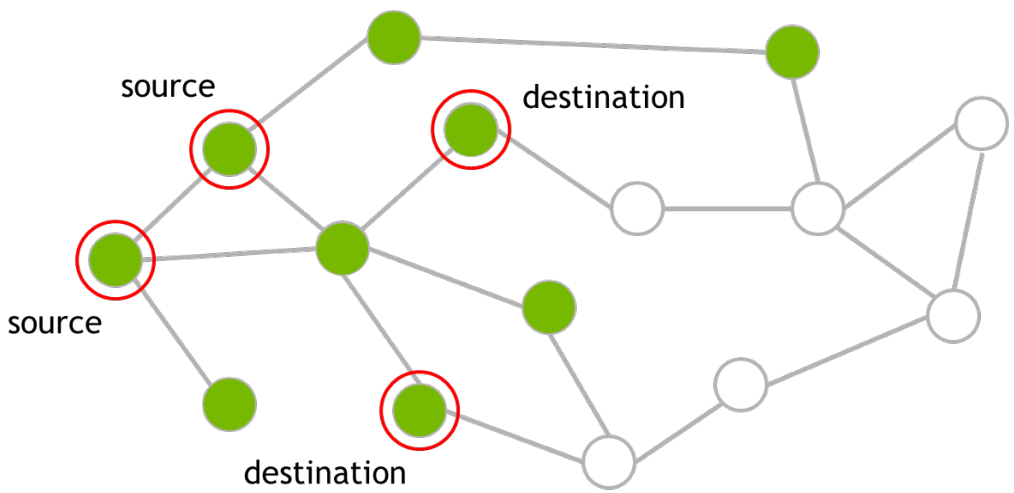
```
// Note “managed” here also

class dataElem : public Managed {
public:
  int key;
  umString name;
};
```

Profit!

```
dataElem *data = new dataElem[N];
…
// C++ now handles our deep copies
Kernel<<< ... >>>(data);}
```

USE CASE: ON-DEMAND PAGING

Graph Algorithms



PERFORMANCE TUNING ON PASCAL+

Demand Paging Impact

This kernel call runs much slower than the Pre-pascal UM 6 case, or the non-UM case.

Each page fault triggers service overhead.

Relying on page faults to move large amounts of data, page-by-page, with overhead on each page, is inefficient.

For bulk movement, a single “memcpy-like” operation is much more efficient

```
__global__ void kernel(float *data){
  int idx = …;
  data[idx] = val;}
…
int n = 256*256;
float *data;
cudaMallocManaged(&data, n*sizeof(float);
Kernel<<<256,256>>>(data);
```

PERFORMANCE TUNING ON PASCAL+

Prefetching

Explicit prefetching:

```
cudaMemPrefetchAsync(ptr, length, destDevice, stream)
```

UM alternative to cudaMemcpy(Async)

Can target any GPU and also the CPU

“Restores” performance

```
__global__ void kernel(float *data){
  int idx = …;
  data[idx] = val;}
…
int n = 256*256;
int ds = n*sizeof(float);
float *data;
cudaMallocManaged(&data, ds);
cudaMemPrefetchAsync(data, ds, 0);
Kernel<<<256,256>>>(data);
cudaMemPrefetchAsync(data, ds,
cudaCpuDeviceId); // copy back to host
```

PERFORMANCE TUNING ON PASCAL+

Explicit Memory Hints

Advise runtime on expected memory access behaviors with:

```
cudaMemAdvise(ptr, count, hint, device);
```

Hints:

- `cudaMemAdviseSetReadMostly`: Specify read duplication
- `cudaMemAdviseSetPreferredLocation`: suggest best location
- `cudaMemAdviseSetAccessedBy`: suggest mapping

Hints don't trigger data movement by themselves

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetReadMostly`

Data will usually be read-only

UM system will make a “local” copy of the data for each processor that touches it

If a processor writes to it, this invalidates all copies except the one written.

Device argument is ignored

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetPreferredLocation`

Suggests which processor is the best location for data

Does not automatically cause migration

Data will be migrated to the preferred processor on-demand (or if prefetched)

If possible, data (P2P) mappings will be provided when other processors touch it

If mapping is not possible, data is migrated

Volta+ adds *access counters* to help GPU make good decisions for you

PERFORMANCE TUNING ON PASCAL+

Hints: `cudaMemAdviseSetAccessedBy`

Does not cause movement or affect location of data

Indicated processor receives a (P2P) mapping to the data

If the data is migrated, mapping is updated

Objective: provide access without incurring page faults

PERFORMANCE

Final Words

- UM is first and foremost about ease of programming and programmer productivity
- UM is not primarily a technique to make well-written CUDA codes run faster
- UM cannot do better than expertly written manual data movement, in most cases
- It can be harder to achieve expected concurrency behavior with UM.
- Misuse of UM can slow a code down dramatically
- There are scenarios where UM may enable a design pattern (e.g. graph traversal).
- Oversubscription does not easily/magically give you GPU-type performance on arbitrary datasets/algorithms
- For codes that tend to use many different libraries, each of which makes some demand on GPU memory with no regard for what other libraries are doing, UM can sometimes be a primary way to tackle this challenge (via use of oversubscription), rather than an entire rewrite of the codebase

MULTI-GPU

- Pre-Pascal Regime:
  - Allocations occur on currently selected device (just like cudaMalloc)
  - All other devices in P2P clique will receive peer mappings
  - Non-P2P: managed allocations happen in zero-copy (host) memory (performance implications!)
- Pascal+ Demand-Paging:
  - Visible to any processor on demand, with or without P2P capability/clique
  - Use prefetching/hints to guide system behavior

POWER9 NOTES

UM vs. ATS vs. HMM

- CPU/GPU dynamic memory allocations (malloc, new) can be coherently accessed among all processors
  - In particular, CPU malloc/new allocations can be read inside of a kernel instead of needing pinned memory
  - Virtual to physical address translations occur in hardware (CPU and GPU MMUs can talk to each other)
- Data is *not* migrated on-demand, but can be manually migrated with cudaMemPrefetchAsync (at lower performance than with UM)
- NVIDIA and others are working on making this functionality available more broadly through the software implementation HMM in the Linux kernel

FUTURE SESSIONS

- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

UM basics:

https://devblogs.nvidia.com/unified-memory-cuda-beginners/

https://devblogs.nvidia.com/unified-memory-in-cuda-6/

optimization:

https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

UM architecture:

http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

Programming Guide:

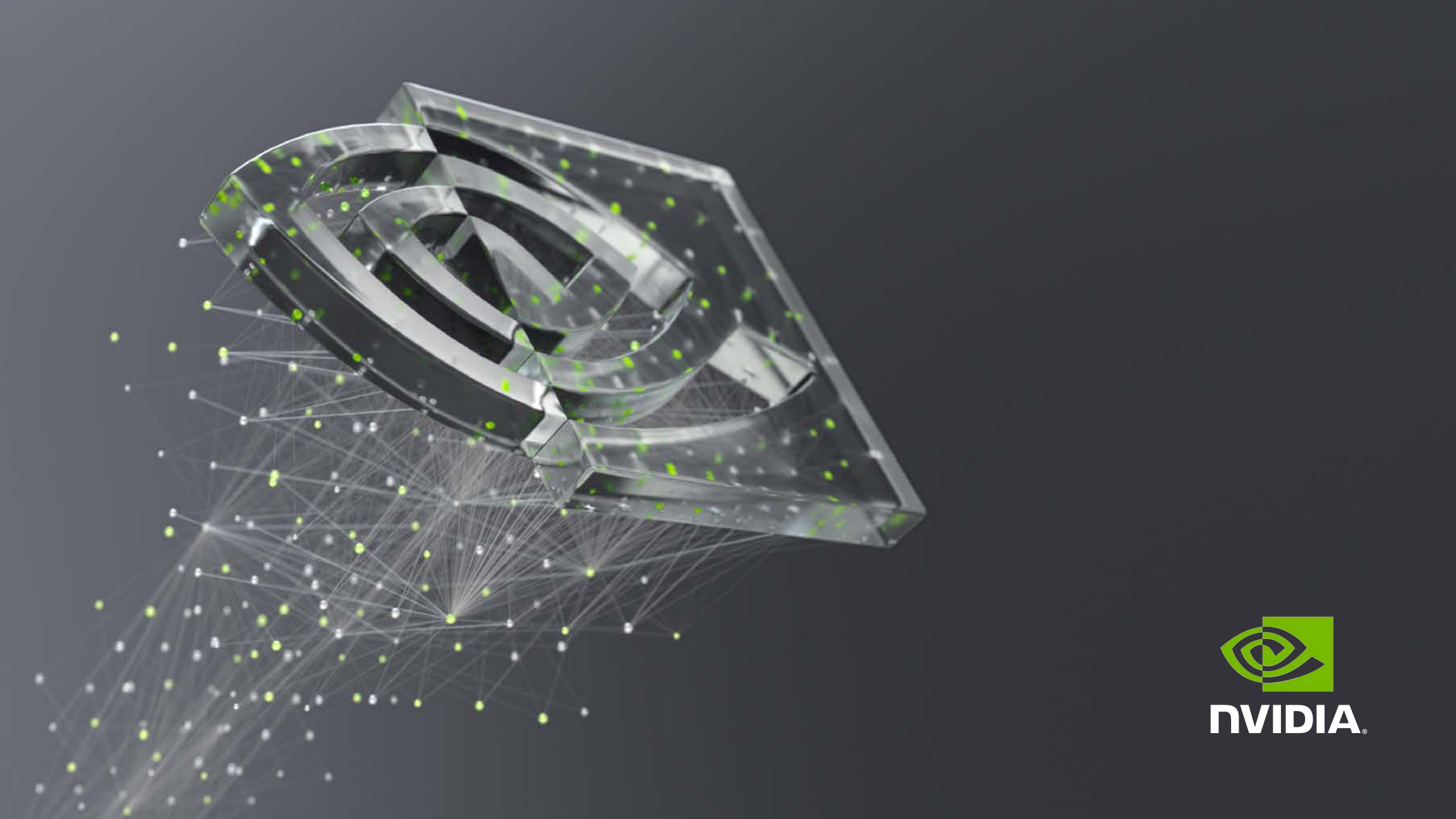
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

CUDA Sample Code: conjugateGradientUM

DLI: Introduction to Accelerated Computing with CUDA C++ (3 labs)

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw6/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming


NVIDIA


BACKUP

HOW UNIFIED MEMORY WORKS IN CUDA 6

En-masse Movement of Data to GPU

GPU Code

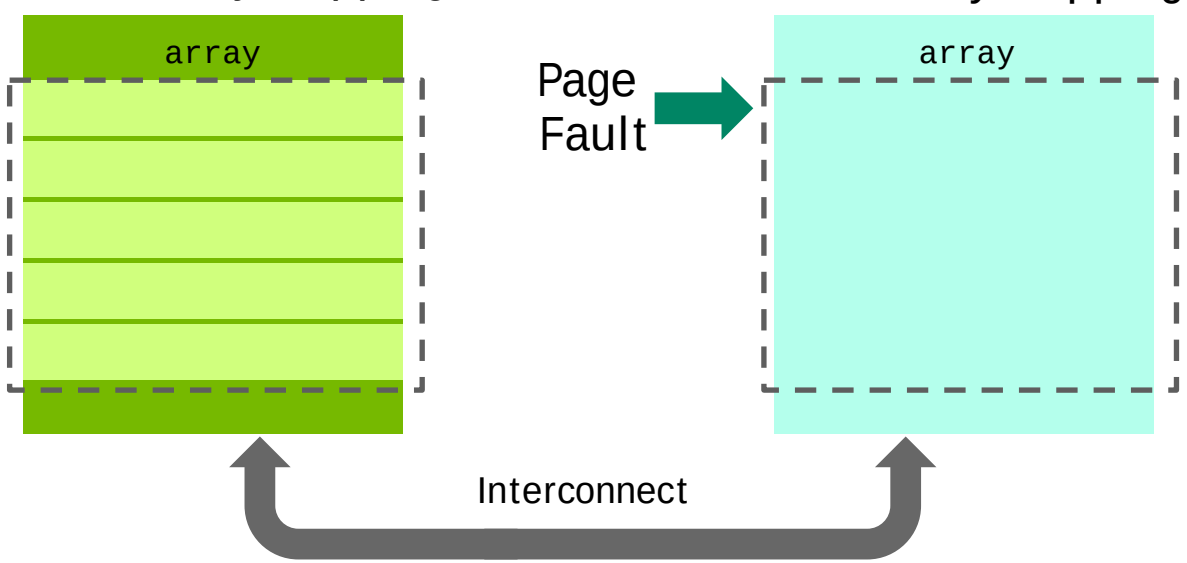
```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```

GPU Memory Mapping

CPU Memory Mapping



CUDA CONCURRENCY

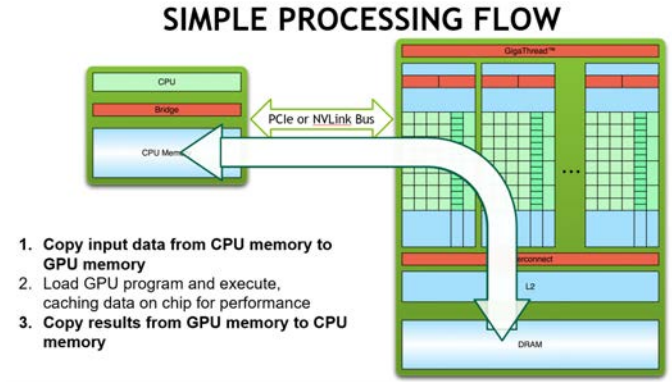
Bob Crovella, 7/21/2020

AGENDA

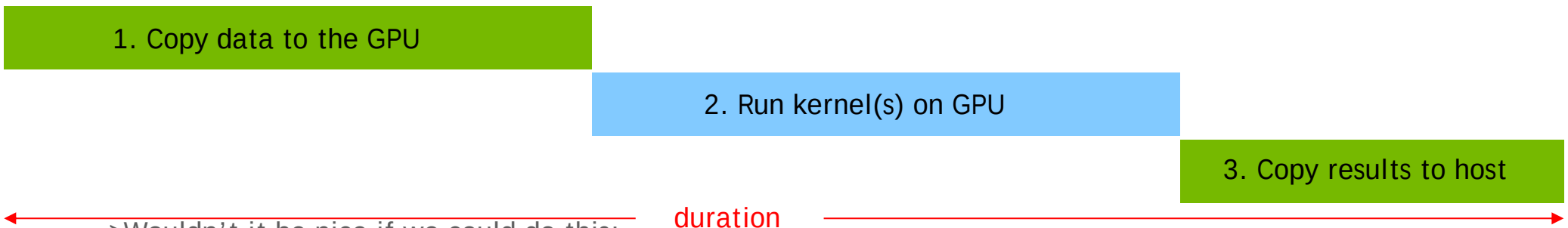
- Concurrency - Motivation
- Pinned Memory
- CUDA Streams
- Overlap of Copy and Compute
- Use Case: Vector Math/Video Processing Pipeline
- Additional Stream Considerations
- Copy-Compute Overlap with Managed Memory
- Multi-GPU Concurrency
- Other Concurrency Scenarios: Kernel Concurrency, Host/Device Concurrency
- Further Study
- Homework

MOTIVATION

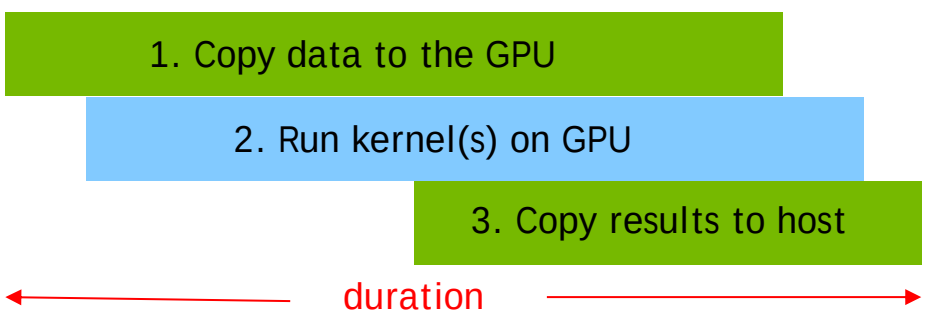
Recall 3 steps from session 1:



Naïve implementation leads to a processing flow like this:



->Wouldn't it be nice if we could do this:




PINNED MEMORY

PINNED (NON-PAGEABLE) MEMORY

- Pinned memory enables:
  - faster Host<->Device copies
  - memcopies asynchronous with CPU
  - memcopies asynchronous with GPU
- Usage
  - `cudaHostAlloc` / `cudaFreeHost`
    - instead of malloc / free or new / delete
  - `cudaHostRegister` / `cudaHostUnregister`
    - pin regular memory (e.g. allocated with malloc) after allocation
- Implication:
  - pinned memory is essentially removed from host virtual (pageable) memory


CUDA STREAMS

STREAMS AND ASYNC API OVERVIEW

- Default API:
  - Kernel launches are asynchronous with CPU
  - cudaMemcpy (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver (legacy default stream)
- Streams and async functions provide:
  - cudaMemcpyAsync (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
  - Concurrent copies in both directions (D2H, H2D) possible on most GPUs
- Stream = sequence of operations that execute in issue-order on GPU
  - Operations from different streams may be interleaved
  - A kernel and memcopy from different streams can be overlapped

STREAM SEMANTICS

1. Two operations issued into the same stream will *execute in issue-order*. Operation B issued after Operation A will not begin to execute until Operation A has completed.
2. Two operations issued into separate streams have *no ordering prescribed by CUDA*. Operation A issued into stream 1 may execute before, during, or after Operation B issued into stream 2.

- Operation: Usually, cudaMemcpyAsync or a kernel call. More generally, most CUDA API calls that take a stream parameter, as well as stream callbacks.

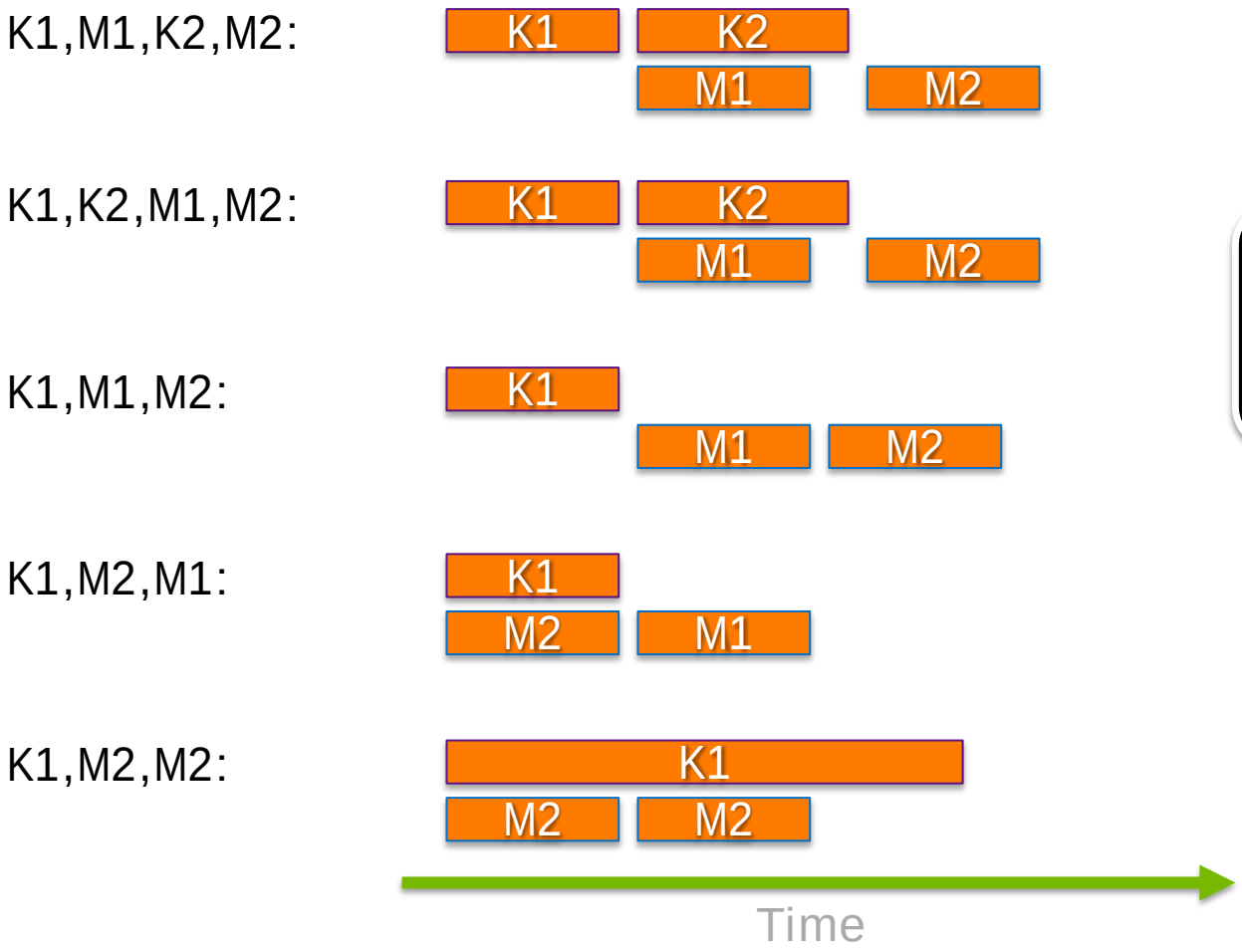
STREAM CREATION AND COPY/COMPUTE OVERLAP

- 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>>>(…);

cudaStreamQuery(stream1);          // test if stream is idle
cudaStreamSynchronize(stream2);    // force CPU thread to wait
cudaStreamDestroy(stream2);
```

potentially overlapped

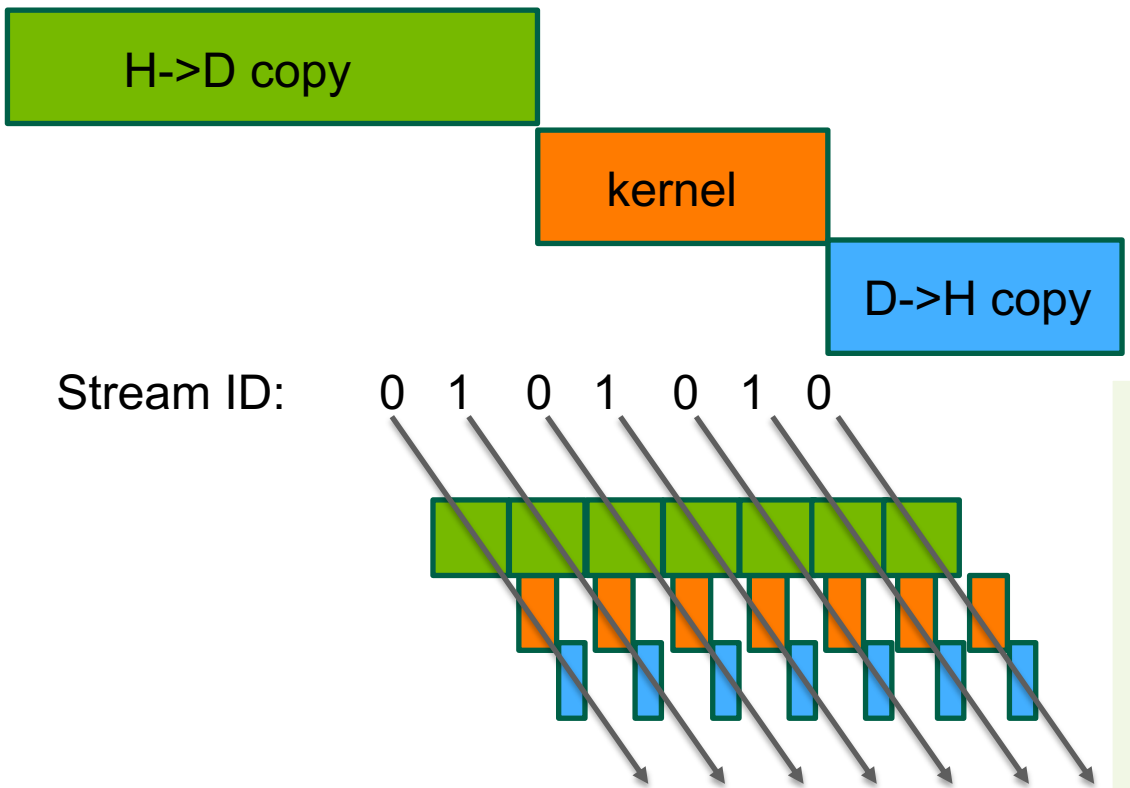
STREAM EXAMPLES



K: Kernel
M: Memcopy
Integer: Stream ID

EXAMPLE STREAM BEHAVIOR FOR VECTOR MATH

(assumes algorithm decomposability)



Similar: video processing pipeline

non-streamed

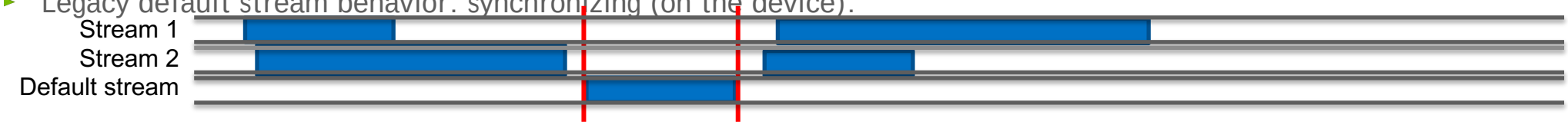
```
cudaMemcpy(d_x, h_x, size_x,
cudaMemcpyHostToDevice);
Kernel<<<b, t>>>(d_x, d_y, N);
cudaMemcpy(h_y, d_y, size_y,
cudaMemcpyDeviceToHost);
```

streamed

```
for (int i = 0, i<c; i++){
  size_t offx = (size_x/c)*i;
  size_t offy = (size_y/c)*i;
  cudaMemcpyAsync(d_x+offx, h_x+offx,
size_x/c, cudaMemcpyHostToDevice,
stream[i%ns]);
  Kernel<<<b/c, t, 0,
stream[i%ns]>>>(d_x+offx, d_y+offy,
N/c);
  cudaMemcpyAsync(h_y+offy, d_y+offy,
size_y/c, cudaMemcpyDeviceToHost,
stream[i%ns]);}
```

DEFAULT STREAM

- Kernels or cudaMemcpy… that do not specify stream (or use 0 for stream) are using the default stream
- Legacy default stream behavior: synchronizing (on the device):



  - All device activity issued prior to the item in the default stream must complete before default stream item begins
  - All device activity issued after the item in the default stream will wait for the default stream item to finish
  - All host threads share the same default stream for legacy behavior
  - Consider avoiding use of default stream during complex concurrency scenarios
- Behavior can be modified to convert it to an “ordinary” stream
  - nvcc --default-stream per-thread …
  - Each host thread will get its own “ordinary” default stream

CUDALAUNCHHOSTFUNC() (STREAM “CALLBACKS”)

- Allows definition of a host-code function that will be issued into a CUDA stream
- Follows stream semantics: function will not be called until stream execution reaches that point
- Uses a thread spawned by the GPU driver to perform the work
- Has limitations: do not use any CUDA runtime API calls (or kernel launches) in the function
- Useful for deferring CPU work until GPU results are ready
- cudaLaunchHostFunc() replaces legacy cudaStreamAddCallback()

COPY-COMPUTE OVERLAP WITH MANAGED MEMORY

In particular, with demand-paging

- Follow same pattern, except use cudaMemPrefetchAsync() instead of cudaMemcpyAsync()
- Stream semantics will guarantee that any needed migrations are performed in proper order
- However, cudaMemPrefetchAsync() has more work to do than cudaMemcpyAsync() (updating of page tables in CPU and GPU)
- This means the call can take substantially more time to return than an "ordinary" async call – can introduce unexpected gaps in timeline
- Behavior varies for "busy" streams vs. idle streams. Counterintuitively, "busy" streams may result in better throughput
- Read about it:
  - https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

ASIDE: CUDAEVENT

- cudaEvent is an entity that can be placed as a “marker” in a stream
- A cudaEvent is said to be “recorded” when it is issued
- A cudaEvent is said to be “completed” when stream execution reaches the point where it was recorded
- Most common use: timing

```
cudaEvent_t start, stop;        // cudaEvent has its own type
cudaEventCreate(&start);        // cudaEvent must be created
cudaEventCreate(&stop);         // before use
cudaEventRecord(start);         // “recorded” (issued) into default stream
Kernel<<<b, t>>>(…);            // could be any set of CUDA device activity
cudaEventRecord(stop);
cudaEventSynchronize(stop);     // wait for stream execution to reach “stop” event
cudaEventElapsedTime(&float_var, start, stop);   // measure Kernel duration
```

- Also useful for arranging complex concurrency scenarios
- Event-based timing may give unexpected results for host activity or complex concurrency scenarios


MULTI-GPU

MULTI-GPU – DEVICE MANAGEMENT

- Not a replacement for OpenMP, MPI, etc.
- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)

cudaSetDevice(int device)

cudaGetDevice(int *device)

cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple host threads can share a device

- A single host thread can manage multiple devices

`cudaSetDevice(i)` to select current device

`cudaMemcpyPeerAsync(…)` for peer-to-peer copies

MULTI-GPU – STREAMS

- Streams (and cudaEvent) have implicit/automatic *device association*
- Each device also has its own unique default stream
- Kernel launches will fail if issued into a stream not associated with current device
- cudaStreamWaitEvent() can synchronize streams belonging to separate devices, cudaEventQuery() can test if an event is “complete”
- Simple device concurrency:

```
cudaSetDevice(0);
cudaStreamCreate(&stream0);         //associated with device 0
cudaSetDevice(1);
cudaStreamCreate(&stream1);         //associated with device 1
Kernel<<<b, t, 0, stream1>>>(…);    // these kernels have the possibility
cudaSetDevice(0);
Kernel<<<b, t, 0, stream0>>>(…);    // to execute concurrently
```

MULTI-GPU – DEVICE-TO-DEVICE DATA COPYING

- If system topology supports it, data can be copied directly from one device to another over a fabric (PCIE, or NVLink)
- Device must first be explicitly placed into a peer relationship (“clique”)
- Must enable “peering” for both directions of transfer (if needed)
- Thereafter, memory copies between those two devices will not “stage” through a system memory buffer (GPUDirect P2P transfer)

```
cudaSetDevice(0);
cudaDeviceCanAccessPeer(&canPeer, 0, 1); // test for 0, 1 peerable
cudaDeviceEnablePeerAccess(1, 0);        // device 0 sees device 1 as a “peer”
cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0, 0);        // device 1 sees device 0 as a “peer”
cudaMemcpyPeerAsync(dst_ptr, 0, src_ptr, 1, size, stream0); //dev 1 to dev 0 copy
cudaDeviceDisablePeerAccess(0);          // dev 0 is no longer a peer of dev 1
```

- Limit to the number of peers in your “clique”

OTHER CONCURRENCY SCENARIOS

- Host/Device execution concurrency:

```
Kernel<<<b, t>>>(…);   // this kernel execution can overlap with
cpuFunction(…);        // this host code
```

- Concurrent kernels:

```
Kernel<<<b, t, 0, streamA>>>(…);   // these kernels have the possibility
Kernel<<<b, t, 0, streamB>>>(…);   // to execute concurrently
```

  - In practice, concurrent kernel execution on the same device is hard to witness
  - Requires kernels with relatively low resource utilization and relatively long execution time
  - There are hardware limits to the number of concurrent kernels per device
  - Less efficient than saturating the device with a single kernel

STREAM PRIORITY

- CUDA streams allow an optional definition of a *priority*
- This affects execution of concurrent kernels (only).
- The GPU block scheduler will attempt to schedule blocks from high priority (stream) kernels before blocks from low priority (stream) kernels
- Current implementation only has 2 priorities
- Current implementation does not cause preemption of blocks

```
// get the range of stream priorities for this device
int priority_high, priority_low;
cudaDeviceGetStreamPriorityRange(&priority_low, &priority_high);
// create streams with highest and lowest available priorities
cudaStream_t st_high, st_low;
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking, priority_high);
cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking, priority_low);
```

CUDA GRAPHS (OVERVIEW)

- New feature in CUDA 10
- Allows for the definition of a sequence of stream(s) work (kernels, memory copy operations, callbacks, host functions, graphs)
- Each work item is a *node* in the graph
- Allows for the definition of *dependencies* (e.g. these 3 nodes must finish before this one can begin)
- Dependencies are effectively graph edges
- Once defined, a graph may be executed by launching it into a stream
- Once defined, a graph may be re-used
- Has both a manual definition method and a “capture” method

FUTURE SESSIONS

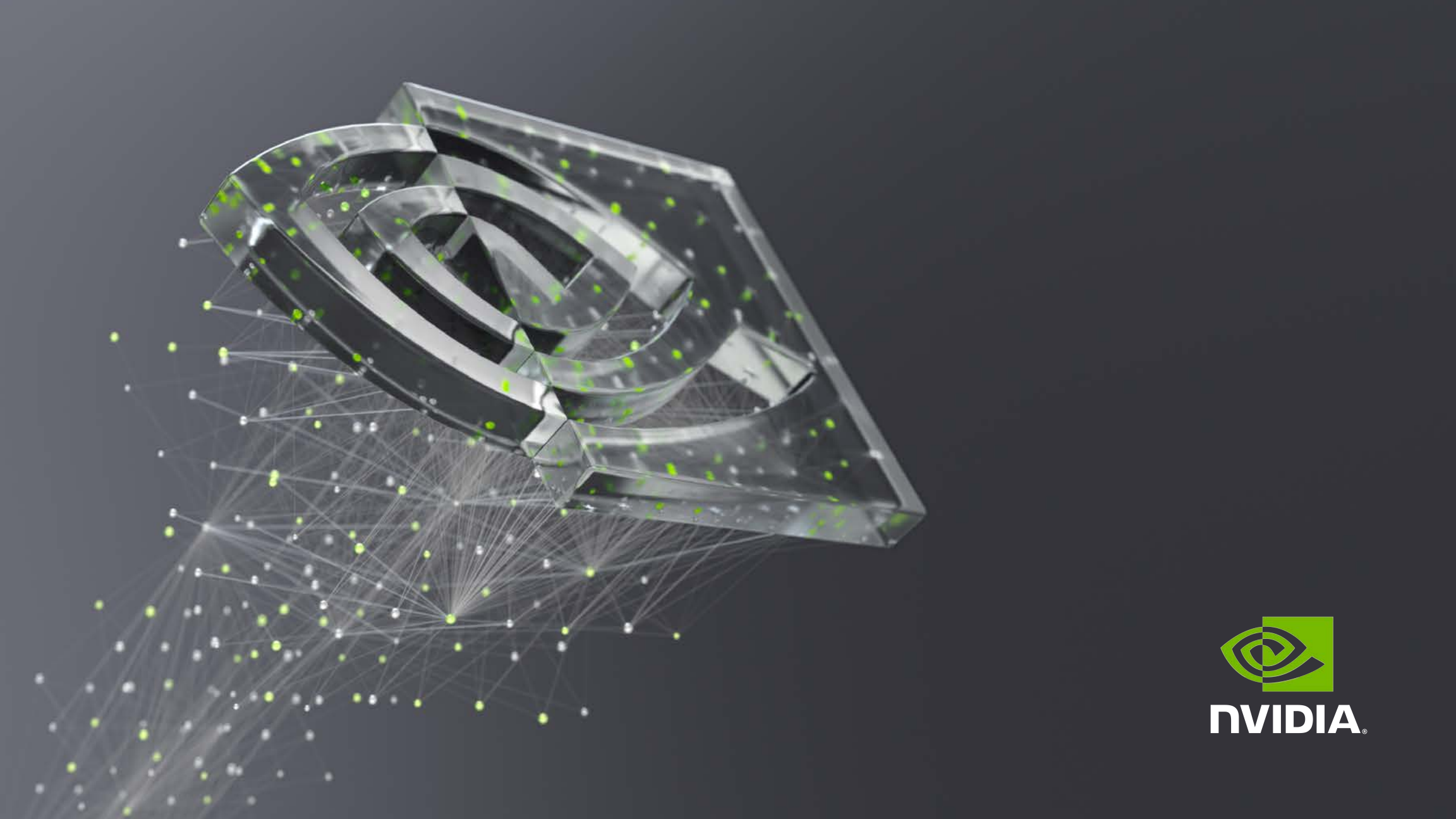
- Analysis Driven Optimization
- Cooperative Groups

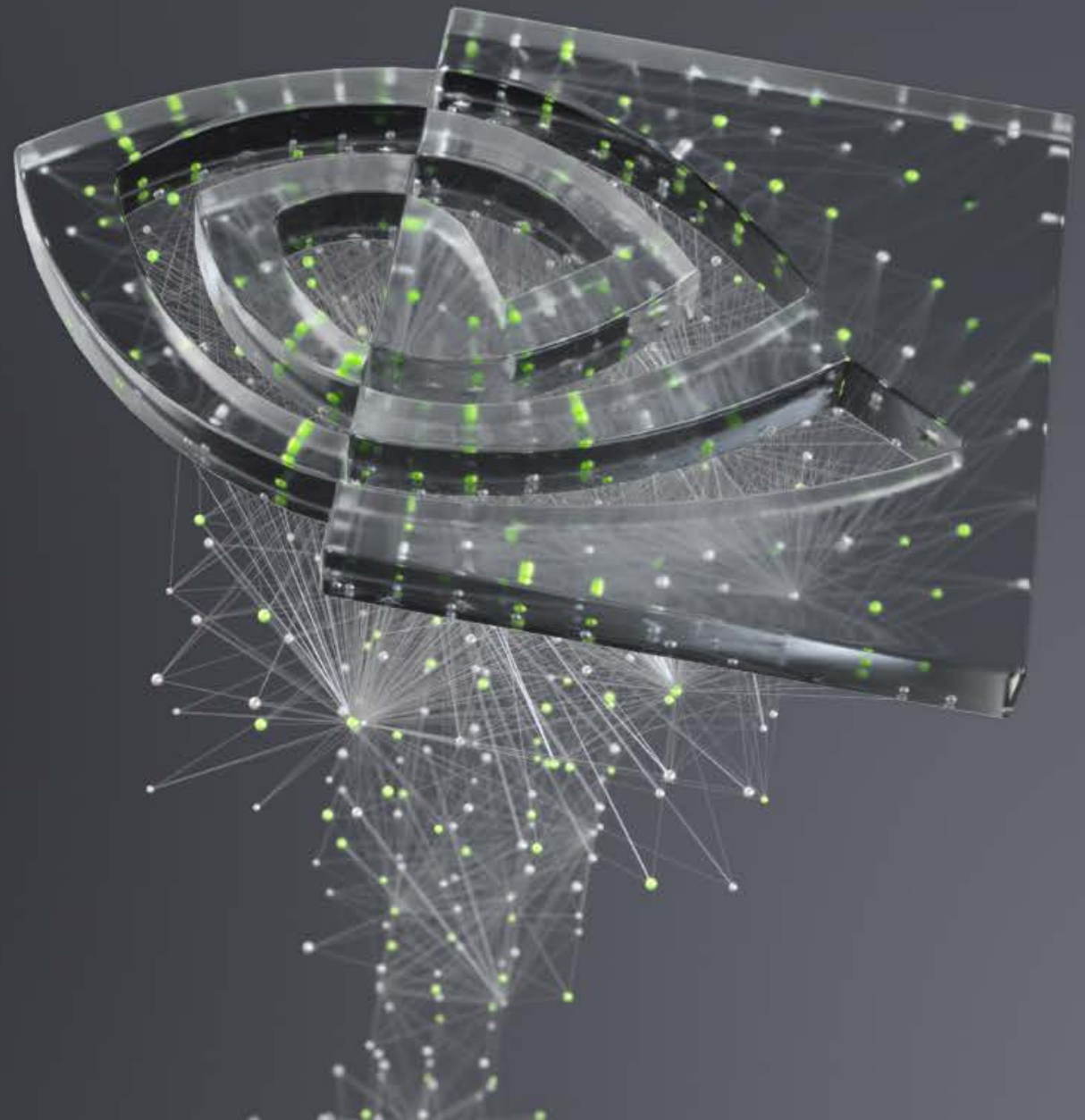
FURTHER STUDY

- Concurrency with Unified Memory:
  - https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/
- Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-concurrent-execution
- CUDA Sample Codes: concurrentKernels, simpleStreams, asyncAPI, simpleCallbacks, simpleP2P
- Video processing pipeline with callbacks:
  - https://stackoverflow.com/questions/31186926/multithreading-for-image-processing-at-gpu-using-cuda/31188999#31188999

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw7/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming


NVIDIA



GPU PERFORMANCE ANALYSIS

Bob Crovella, 8/18/2020

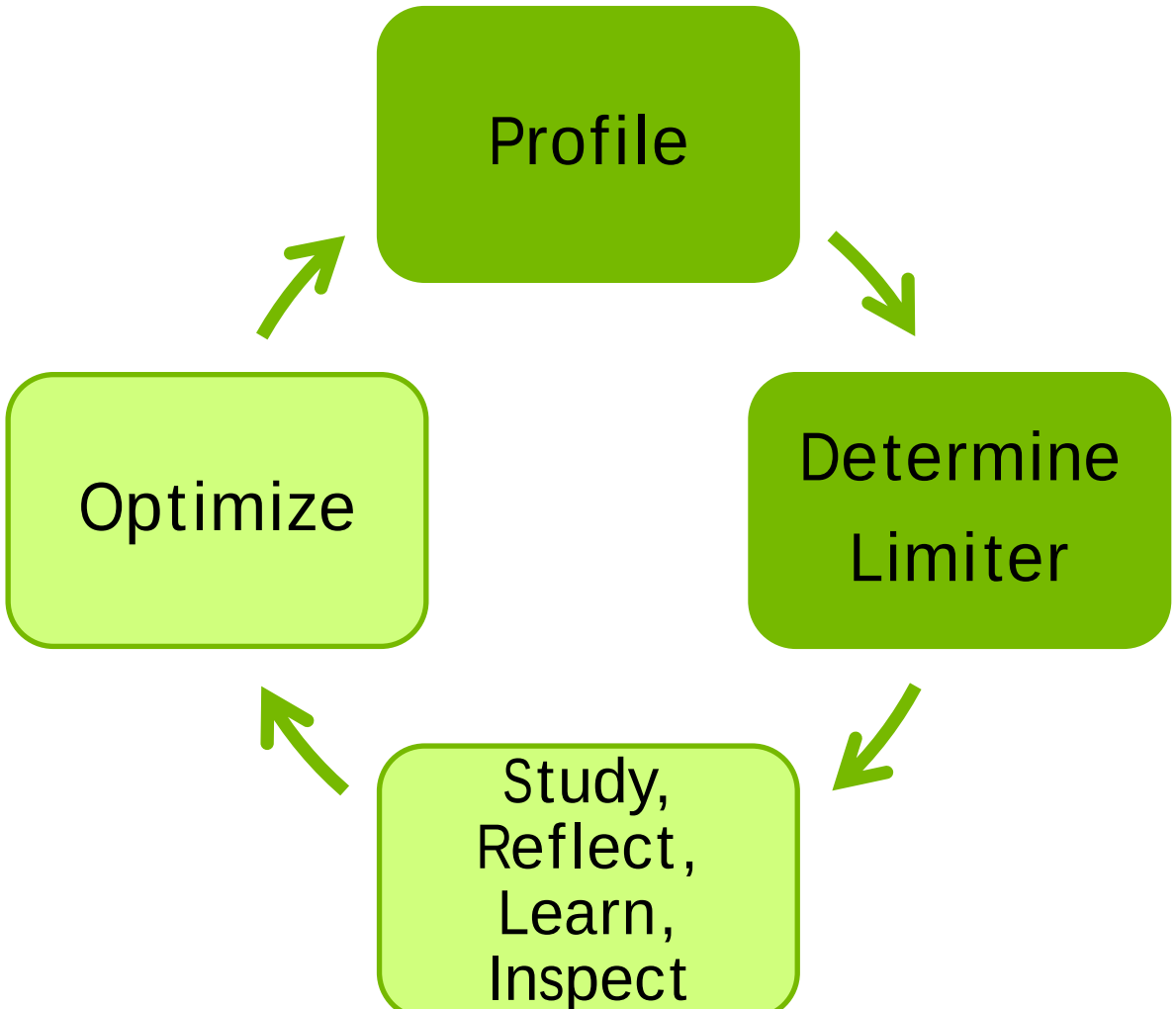
AGENDA

- Analysis Driven Optimization
- Understanding Performance Limiters
- Metrics Review
- Memory Bound Analysis
- Compute Bound Analysis
- Future Sessions
- Further Study
- Homework

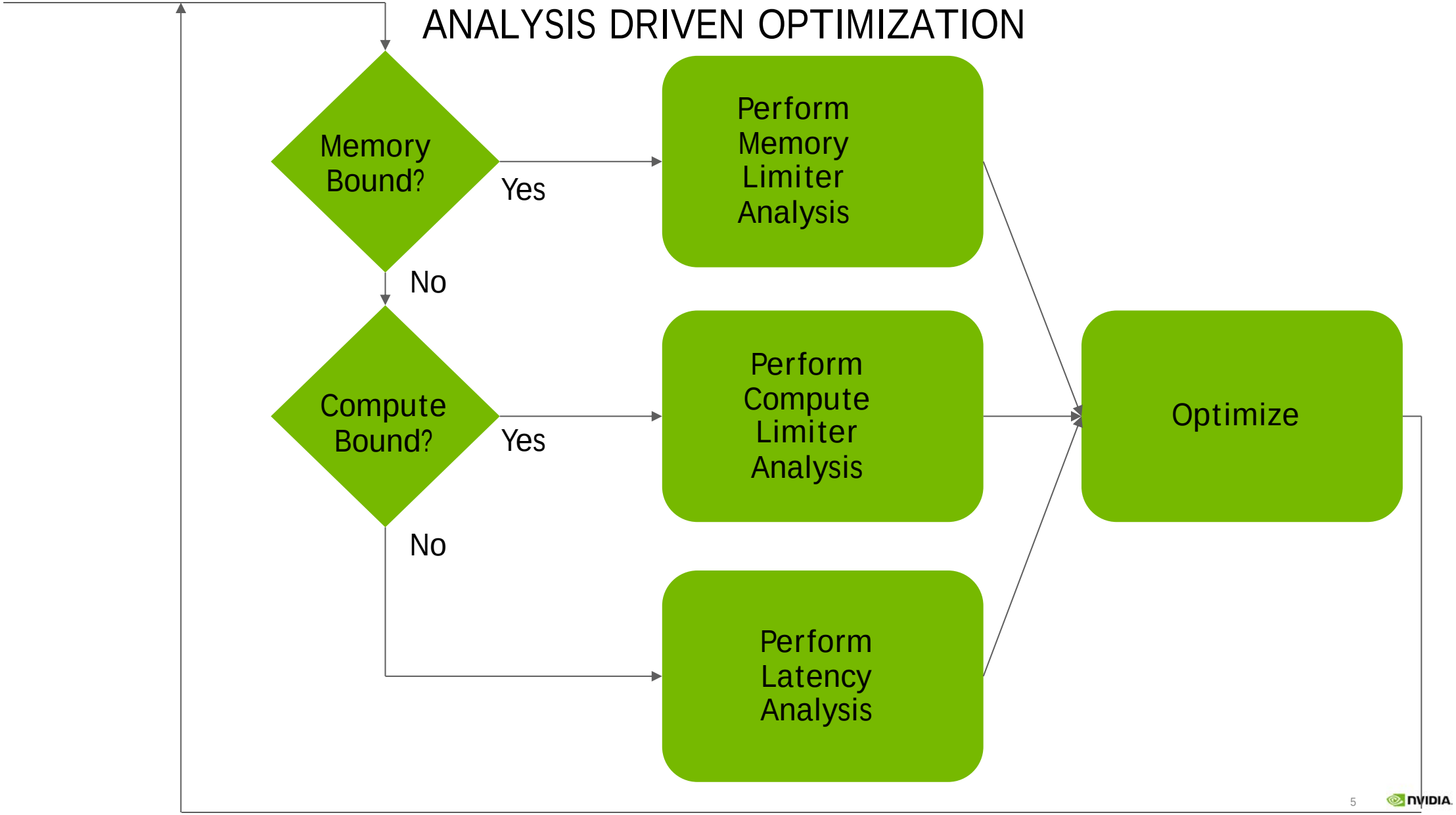
REVIEW: TOP-LEVEL PERFORMANCE CODING OBJECTIVES

- Make efficient use of the memory subsystem
  - Efficient use of global memory (coalesced access)
  - Intelligent use of the memory hierarchy
    - shared, constant, texture, caches, etc.
- Expose enough parallelism (work) to saturate the machine and hide latency
  - Threads/blocks
  - Occupancy
  - Work per thread
  - Execution efficiency

ANALYSIS DRIVEN OPTIMIZATION



ANALYSIS DRIVEN OPTIMIZATION



TOP-LEVEL PERFORMANCE BEHAVIOR - LIMITERS

- Memory Bound – A code is memory bound, when the measured memory system performance is at or close to the expected maximum. (saturate memory bus)
- Compute Bound – A code is compute bound when the compute instruction throughput is at or close to the expected maximum.
- Latency Bound – One of the indicators for a latency bound code is when neither of the above are true.
- (Analysis-driven) Optimization uses the above determination to direct code refactoring efforts in the first stage.
- Limiting behavior of a code may change over the duration of its execution cycle.
- It’s desirable to analyze small sections of code e.g. one kernel at a time

METRICS FOR DETERMINING COMPUTE VS. MEMORY BOUND

https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#nvprof-metric-comparison

Latency metrics:
“sm efficiency”: smsp__cycles_active.avg.pct_of_peak_sustained_elapsed

Memory metrics:
“dram utilization”: dram__throughput.avg.pct_of_peak_sustained_elapsed
“L2 utilization”: lts__t_sectors.avg.pct_of_peak_sustained_elapsed
“shared utilization”:
l1tex__data_pipe_lsu_wavefronts_mem_shared.avg.pct_of_peak_sustained_elapsed

Compute metrics:
“DP Utilization”: smsp__inst_executed_pipe_fp64.avg.pct_of_peak_sustained_active
“SP Utilization”: smsp__pipe_fma_cycles_active.avg.pct_of_peak_sustained_active
“HP Utilization”: smsp__inst_executed_pipe_fp16.avg.pct_of_peak_sustained_active
“TC Utilization”: sm__pipe_tensor_op_hmma_cycles_active.avg.pct_of_peak_sustained_active
“Integer Utilization”:
smsp__sass_thread_inst_executed_op_integer_pred_on.avg.pct_of_peak_sustained_active

MEMORY BOUND

- A code can be memory bound when either it is limited by memory bandwidth or latency. We will lump memory latency bound codes in with the general latency case.
- For a memory bandwidth bound code, we will seek to optimize usage of the various memory subsystems, taking advantage of the memory hierarchy where possible.
  - Optimize use of global memory
  - Under data reuse scenarios, make (efficient) use of higher levels of the memory hierarchy, and optimize these usages (L2 cache, shared memory).
  - Take advantage of cache "diversification" using special GPU caches – constant cache, read-only cache, texture cache/memory, surface memory.
- For a code that is memory bandwidth bound, we can compute the actual throughput vs. peak theoretical

COMPUTE BOUND

- A code is compute bound when the performance of a particular type of compute instruction/operation is at or near the limit of the functional unit servicing that type
- Optimization strategy involves optimizing the use of that functional unit type, as well as (possibly) seeking to shift the compute load to other types
- For a code that is dominated by a particular type (e.g. single precision floating point multiply/add) we can compare the actual throughput vs. peak theoretical.

LATENCY BOUND

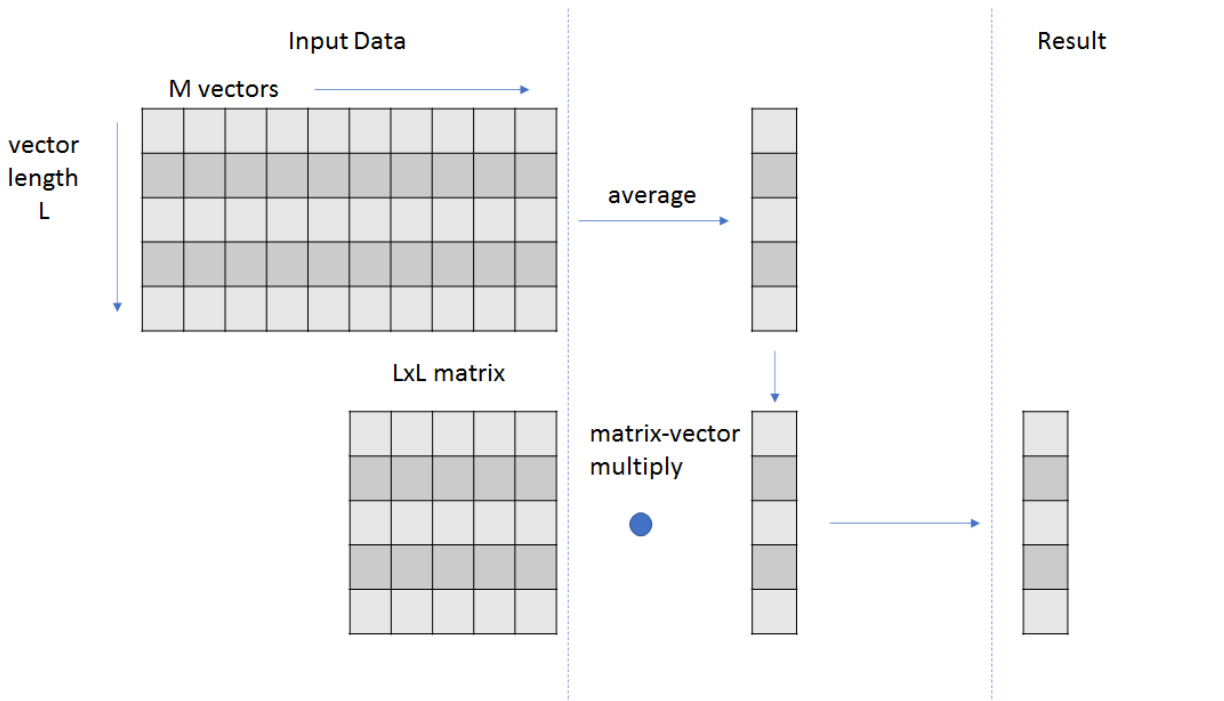
- A code is latency bound when the GPU cannot keep busy with the available/exposed parallel work.
- The general strategy for a latency bound code will be to expose more parallel work
  - Make sure that you are launching a large number of threads
  - Increase the work per thread (e.g. via a loop over input elements)
  - Use “vector load” to allow a single thread to process multiple input elements
  - Strive for maximum occupancy

WHAT IS OCCUPANCY?

- A measure of the actual thread load in an SM, vs. peak theoretical/peak achievable
- CUDA includes an occupancy calculator spreadsheet
- Higher occupancy is sometimes a path to higher performance
- Achievable occupancy is affected by limiters to occupancy
- Primary limiters:
  - Registers per thread (can be reported by the profiler, or can get at compile time)
  - Threads per threadblock
  - Shared memory usage

WALK-THRU

- What the code does:



This process is repeated N times, using N sets of input vectors, reusing the matrix, producing N result vectors.

FUTURE SESSIONS

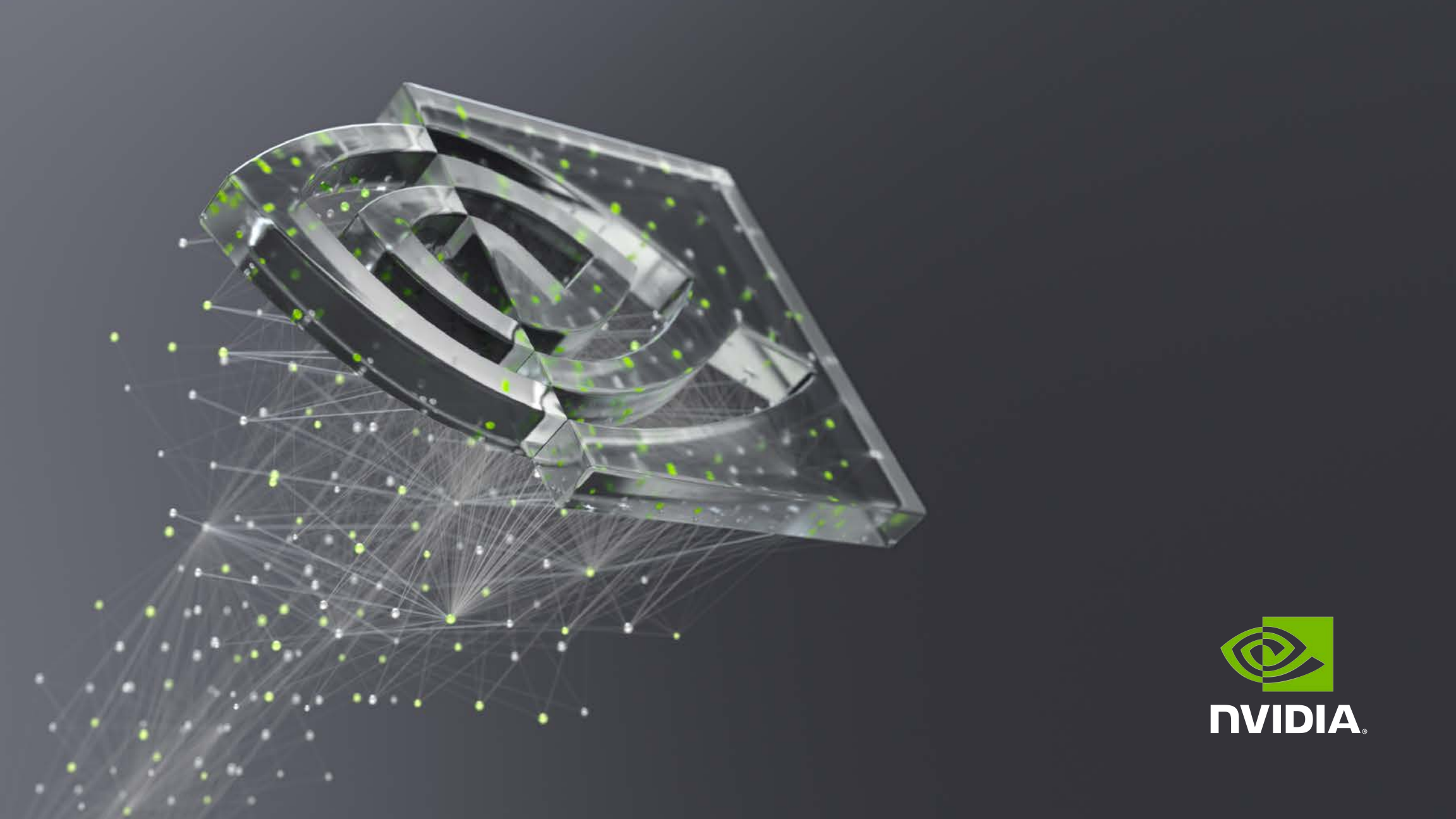
- Cooperative Groups

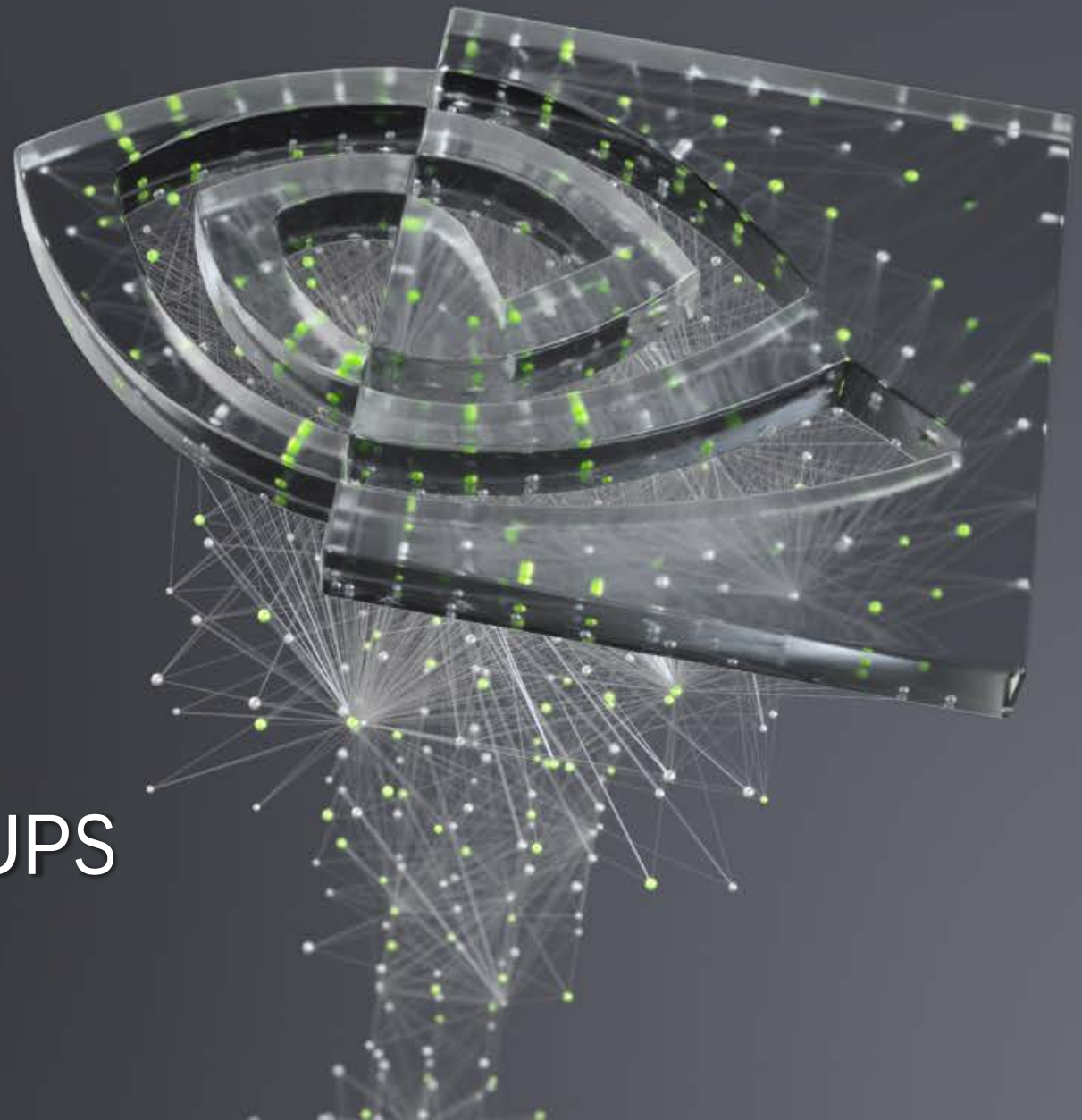
FURTHER STUDY

- Analysis Driven optimization:
  - http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf
  - http://www.nvidia.com/content/GTC-2010/pdfs/2012_GTC2010.pdf
  - Google “gtc cuda optimization”
- New tools blogs:
  - https://developer.nvidia.com/blog/migrating-nvidia-nsight-tools-nvvp-nvprof/
  - https://developer.nvidia.com/blog/transitioning-nsight-systems-nvidia-visual-profiler-nvprof/
  - https://developer.nvidia.com/blog/using-nsight-compute-to-inspect-your-kernels/

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw7/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming


NVIDIA



COOPERATIVE GROUPS

Bob Crovella, 3/28/2019

AGENDA

Cooperative Groups

- Threadblock Level
- Grid Level
- Multi-Device
- Coalesced Group

Further Study

Homework


COOPERATIVE GROUPS

Cooperative Groups: a flexible model for synchronization and communication within groups of threads.

At a glance

Scalable Cooperation among groups of threads

Flexible parallel decompositions

Composition across software boundaries

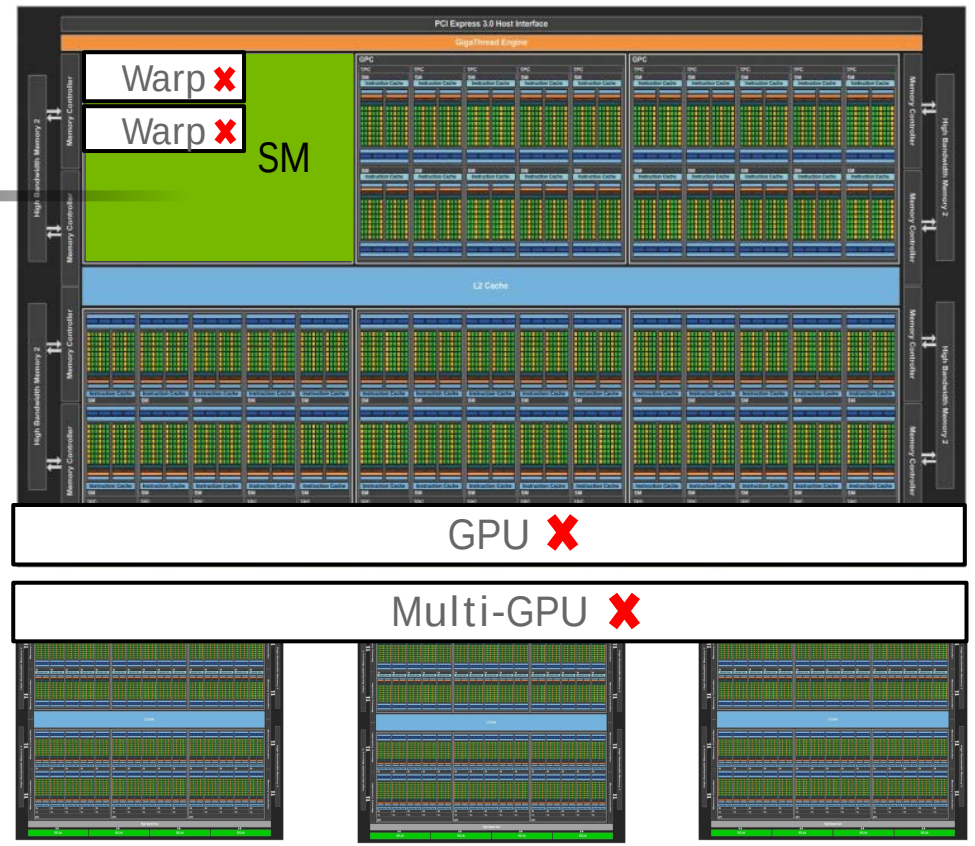
Obvious benefit: grid-wide sync

Examples include:

- Persistent RNNs
- Reductions
- Search Algorithms
- Sorting

LEVELS OF COOPERATION: PRE CUDA 9.0

`__syncthreads()`: block level synchronization barrier in CUDA



LEVELS OF COOPERATION: CUDA 9.0

For current coalesced set of threads:

```
auto g = coalesced_threads();
```

For warp-sized group of threads:

```
auto block = this_thread_block();
auto g = tiled_partition<32>(block)
```

For CUDA thread blocks:

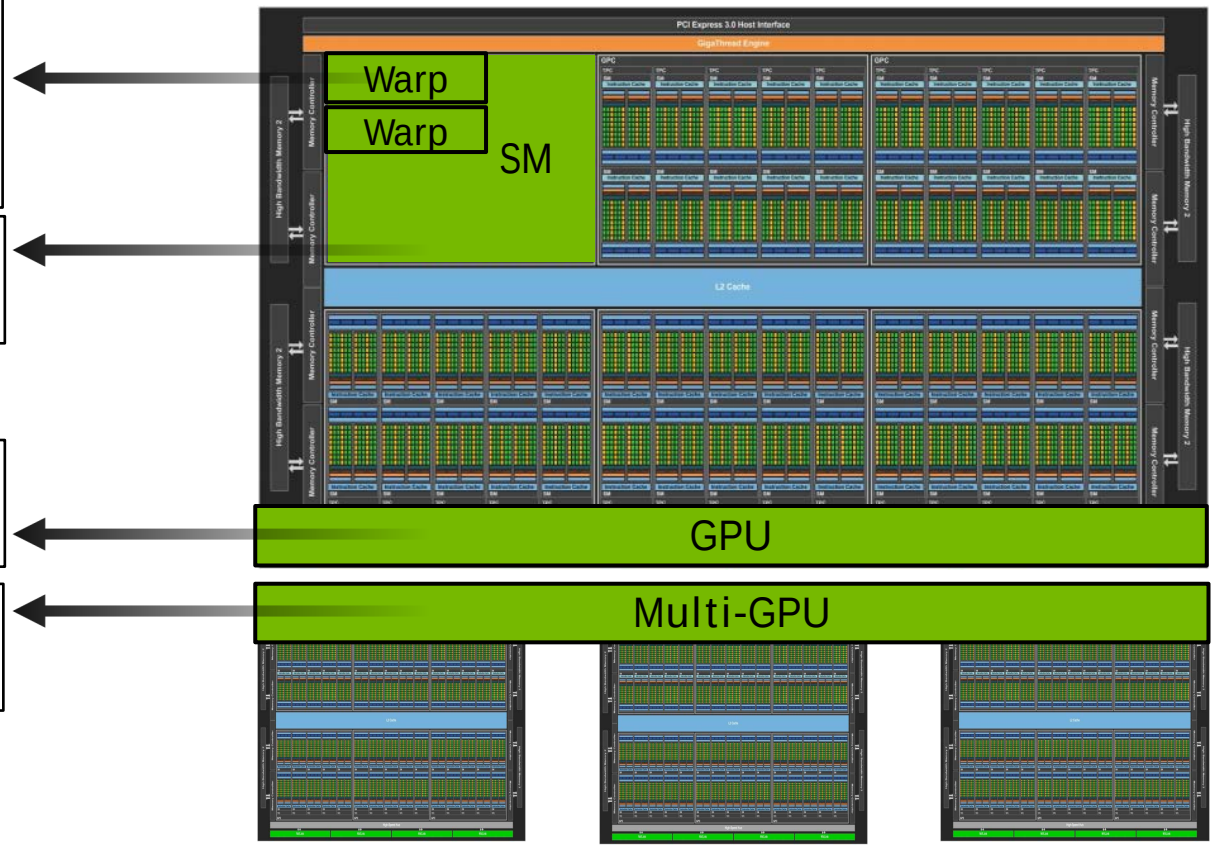
```
auto g = this_thread_block();
```

For device-spanning grid:

```
auto g = this_grid();
```

For multiple grids spanning GPUs:

```
auto g = this_multi_grid();
```



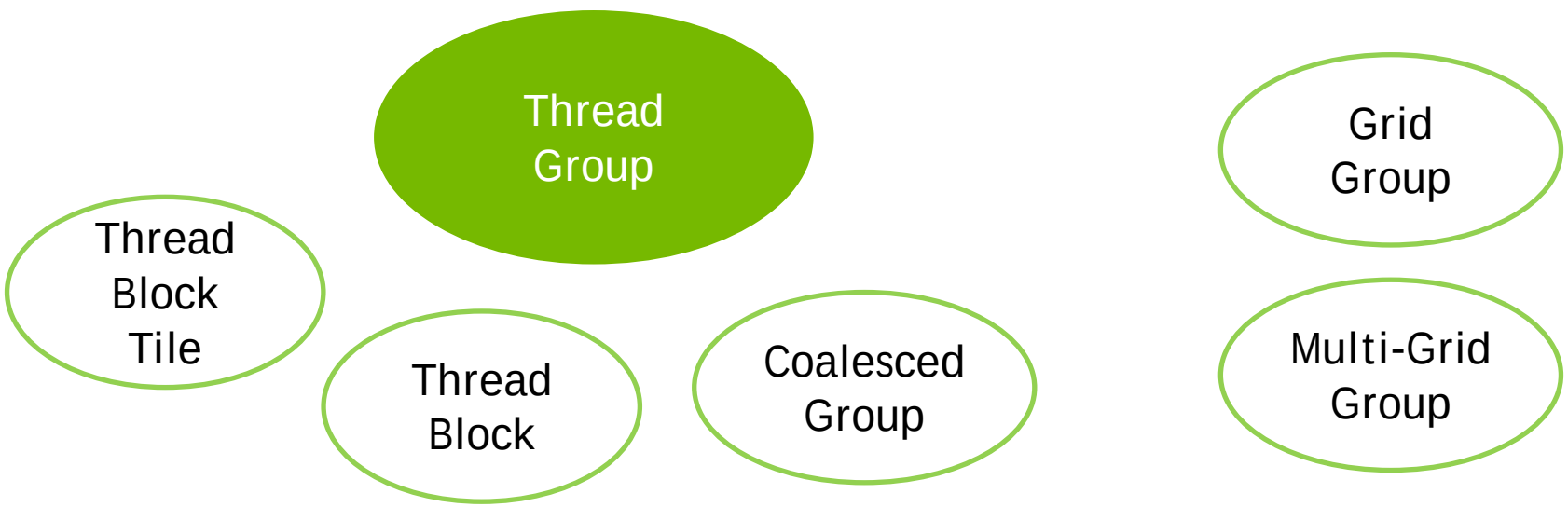
All Cooperative Groups functionality is within a cooperative_groups:: namespace

THREAD GROUP

Base type, the implementation depends on its construction.

Unifies the various group types into one general, collective, thread group.

We need to extend the CUDA programming model with handles that can represent the groups of threads that can communicate/synchronize



THREAD BLOCK

Implicit group of all the threads in the launched thread block

Implements the same interface as thread_group:

```
void sync();                // Synchronize the threads in the group
unsigned size();            // Total number of threads in the group
unsigned thread_rank();     // Rank of the calling thread within [0, size)
bool is_valid();            // Whether the group violated any API constraints
```

And additional thread_block specific functions:

```
dim3 group_index();         // 3-dimensional block index within the grid
dim3 thread_index();        // 3-dimensional thread index within the block
```

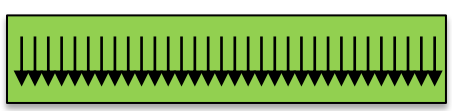
PROGRAM DEFINED DECOMPOSITION

All threads launched

```
thread_block g = this_thread_block();
```

All threads in thread block

```
thread_group tile32 = tiled_partition(g, 32);
```



```
thread_group tile4 = tiled_partition(tile32, 4);
```

Restricted to powers of two, and <= 32 in initial release

GENERIC PARALLEL ALGORITHMS

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition(this_thread_block(), 32);
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;        g.sync();
    if (lane < i) val += x[lane + i];  g.sync();
  }
  return val;
}
```

THREAD BLOCK TILE

A subset of threads of a thread block, divided into tiles in row-major order

```
thread_block_tile<32> tile32 = tiled_partition<32>(this_thread_block());
```

```
thread_block_tile<4> tile4 = tiled_partition<4>(this_thread_block());
```

Exposes additional functionality:

```
.shfl()
.shfl_down()
.shfl_up()
.shfl_xor()
```

```
.any()
.all()
.ballot()
.match_any()
.match_all()
```

Size known at compile time = fast!

STATIC TILE REDUCE

Per-Tile of 16 threads

```
g = tiled_partition<16>(this_thread_block());
tile_reduce(g, myVal);
```

```
template <unsigned size>
__device__ int tile_reduce(thread_block_tile<size> g, int val) {
  for (int i = g.size()/2; i > 0; i /= 2) {
    val += g.shfl_down(val, i);
  }
  return val;
}
```

GRID GROUP

A set of threads within the same grid, guaranteed to be resident on the device

New CUDA Launch API to opt-in:
`cudaLaunchCooperativeKernel(…)`

```
__global__ kernel() {
    grid_group grid = this_grid();
    // load data
    // loop - compute, share data
        grid.sync();
        // device wide execution barrier
}
```

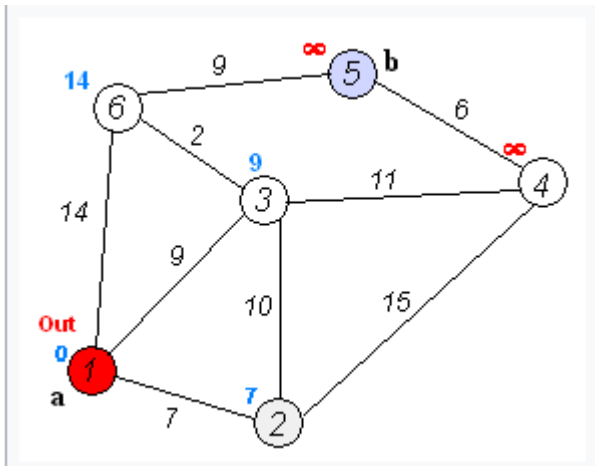
Device needs to support the `cooperativeLaunch` property.

```
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, kernel, numThreads, 0);
```

GRID GROUP

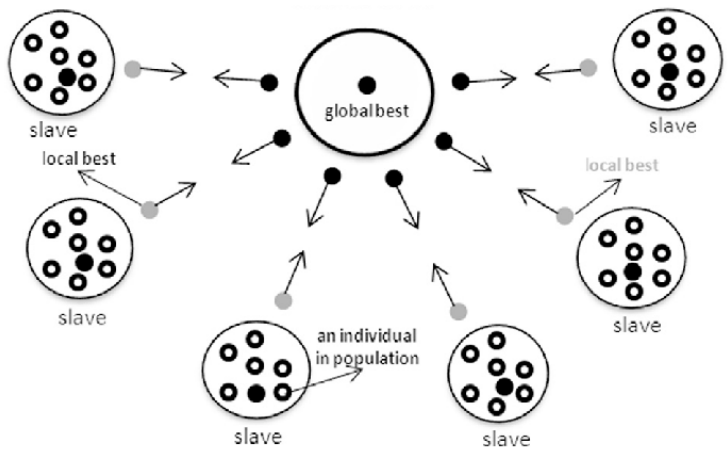
The goal: keep as much state as possible resident

Shortest Path / Search



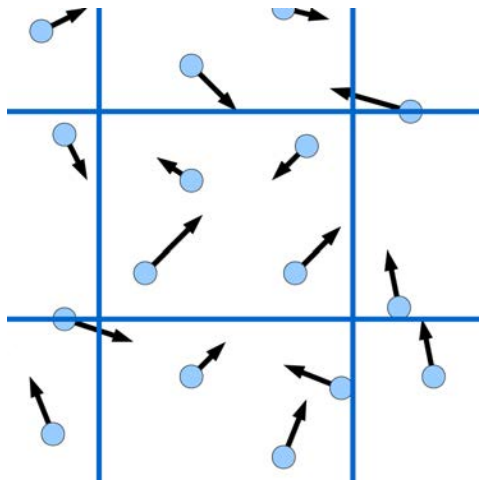
Weight array perfect for persistence
Iteration over vertices?
Fuse!

Genetic Algorithms / Master driven algorithms



Synchronization between a master block and slaves

Particle Simulations

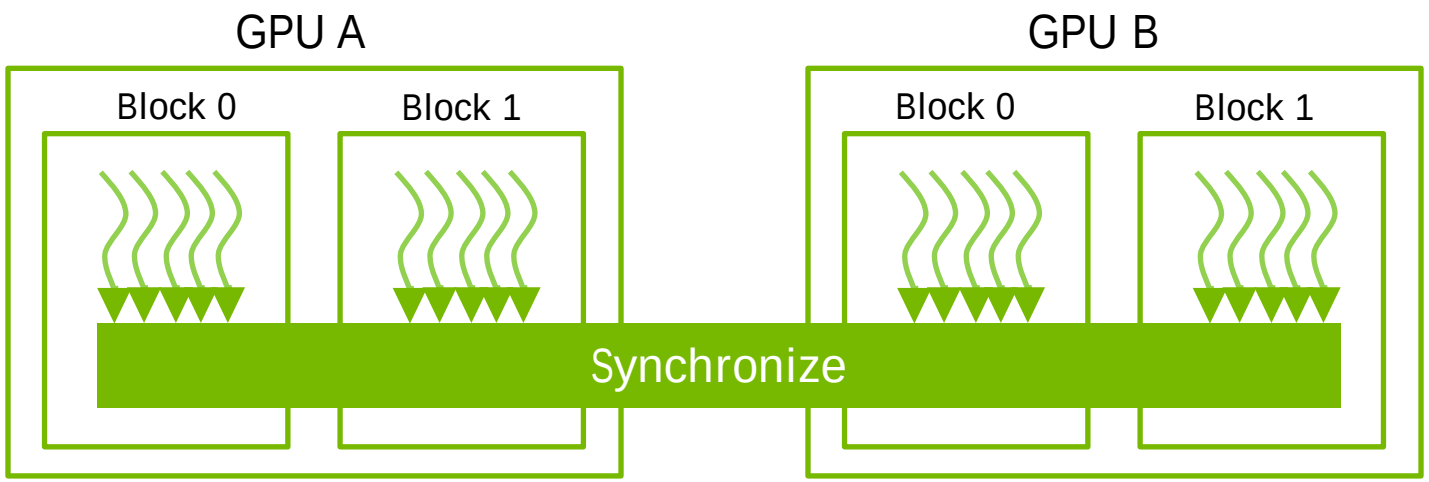


Synchronization between update and collision simulation

MULTI GRID GROUP

A set of threads guaranteed to be resident on the same system, on multiple devices

```
__global__ void kernel() {
    multi_grid_group multi_grid = this_multi_grid();
    // load data
    // loop - compute, share data
        multi_grid.sync();
        // devices are now synced, keep on computing
}
```



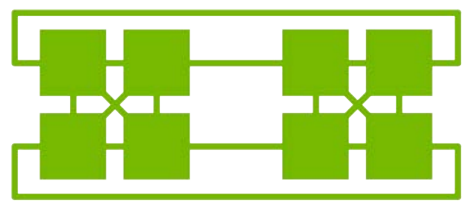
MULTI GRID GROUP

Launch on multiple devices at once

New CUDA Launch API to opt-in:
`cudaLaunchCooperativeKernelMultiDevice(…)`

Devices need to support the `cooperativeMultiDeviceLaunch` property.

```
struct cudaLaunchParams params[numDevices];
for (int i = 0; i < numDevices; i++) {
    params[i].func = (void *)kernel;
    params[i].gridDim = dim3(…); // Use occupancy calculator
    params[i].blockDim = dim3(…);
    params[i].sharedMem = …;
    params[i].stream = …; // Cannot use the NULL stream
    params[i].args = …;
}
cudaLaunchCooperativeKernelMultiDevice(params, numDevices);
```

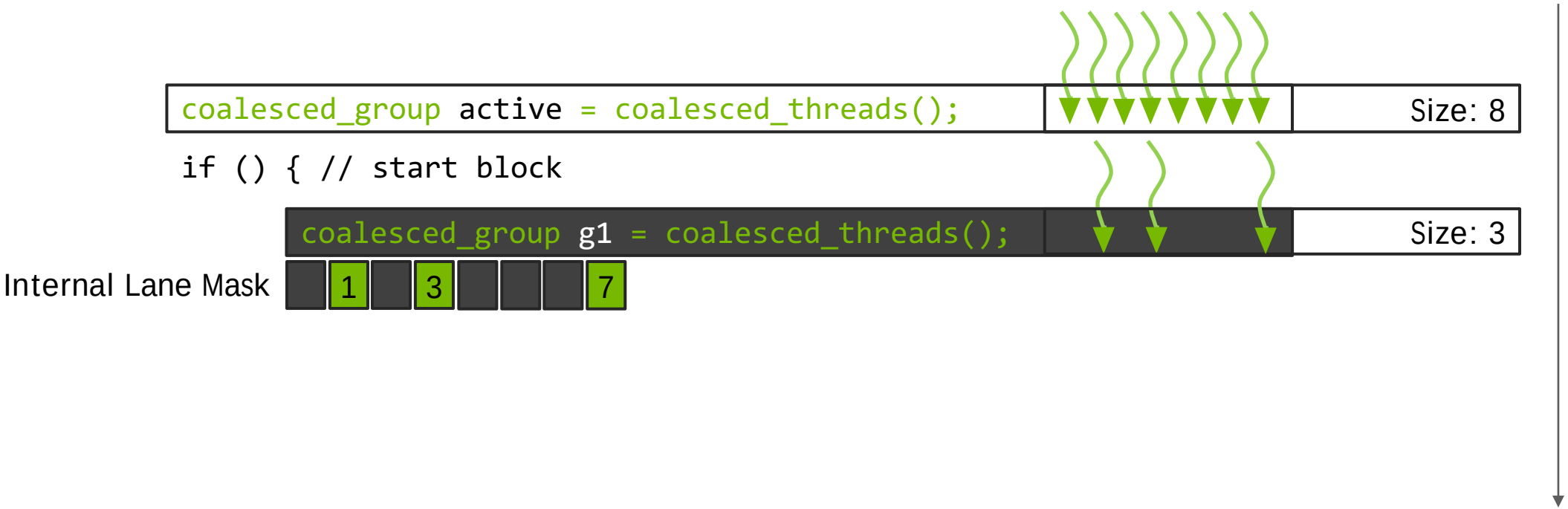


COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

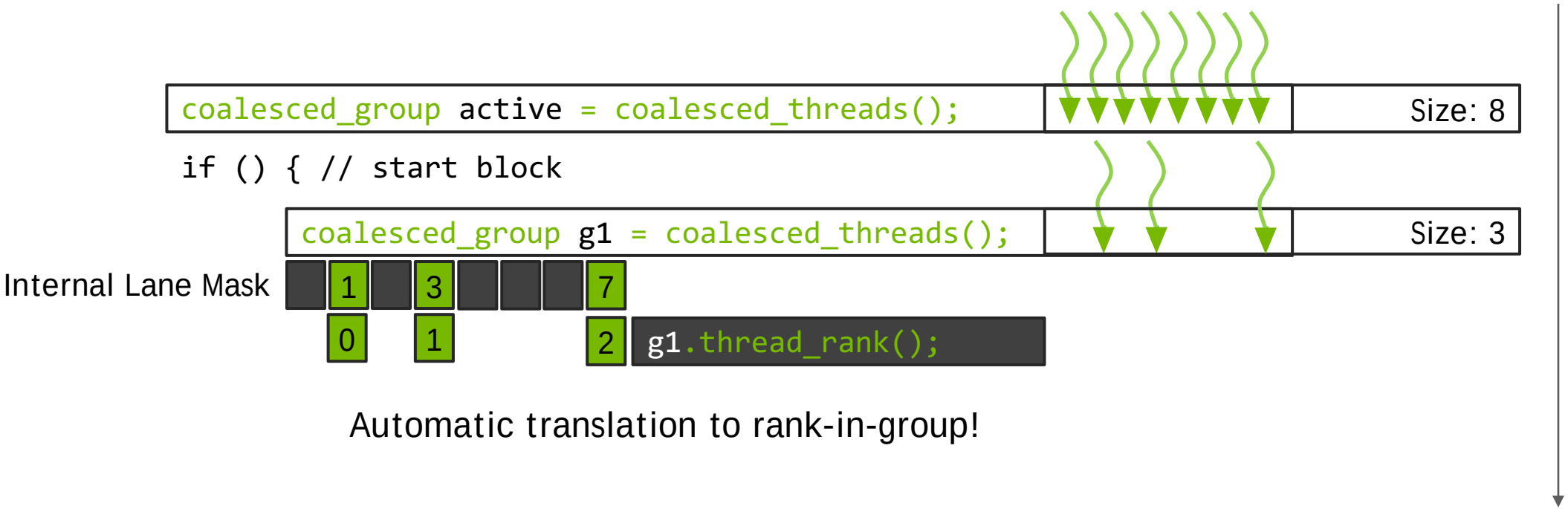
COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

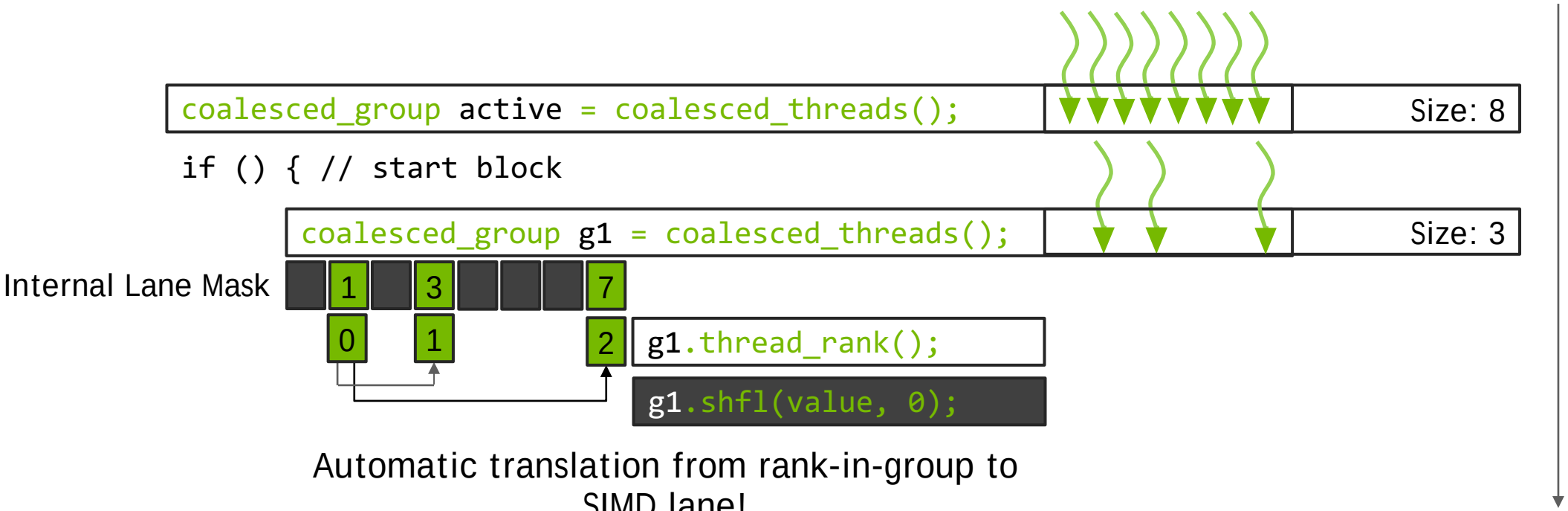
Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



Automatic translation to rank-in-group!

COALESCED GROUP

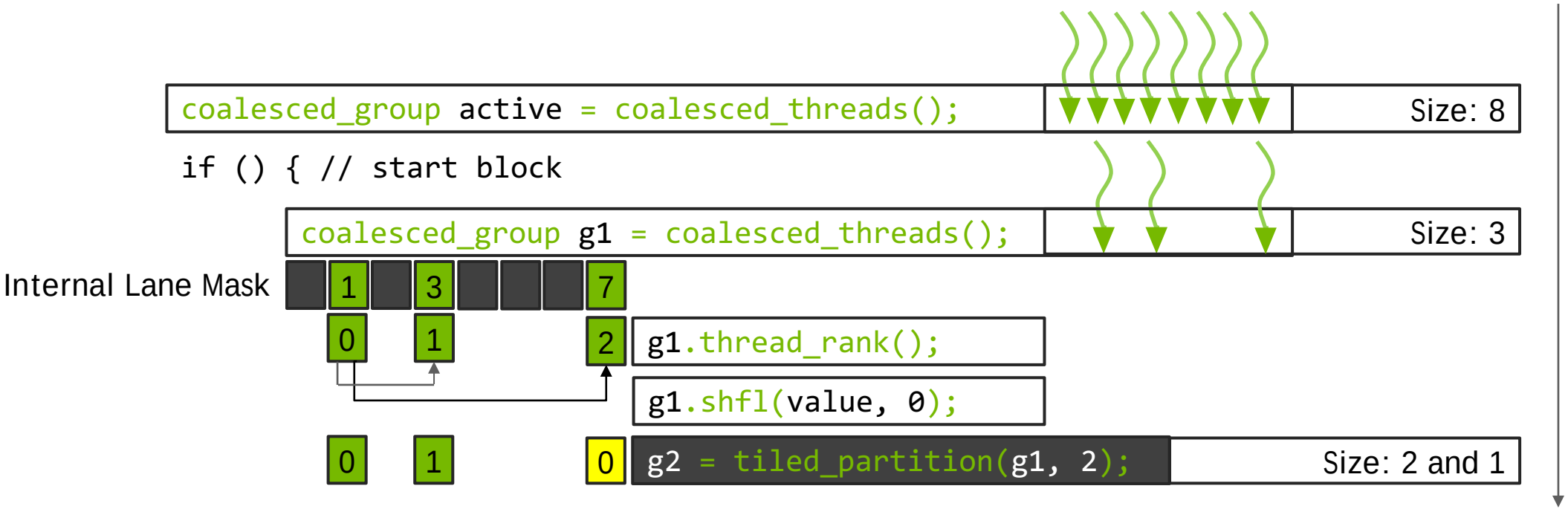
Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



Automatic translation from rank-in-group to SIMD lane!

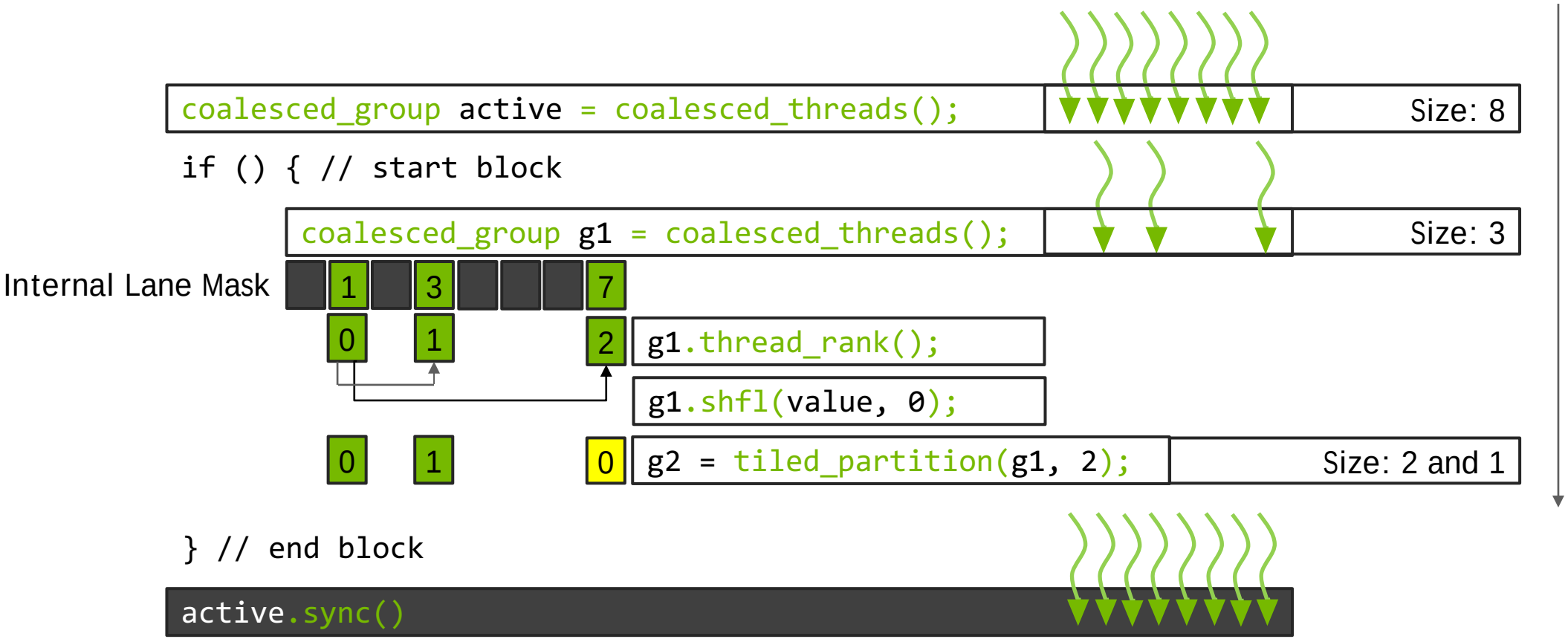
COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



ATOMIC AGGREGATION

Opportunistic Cooperation Within a Warp

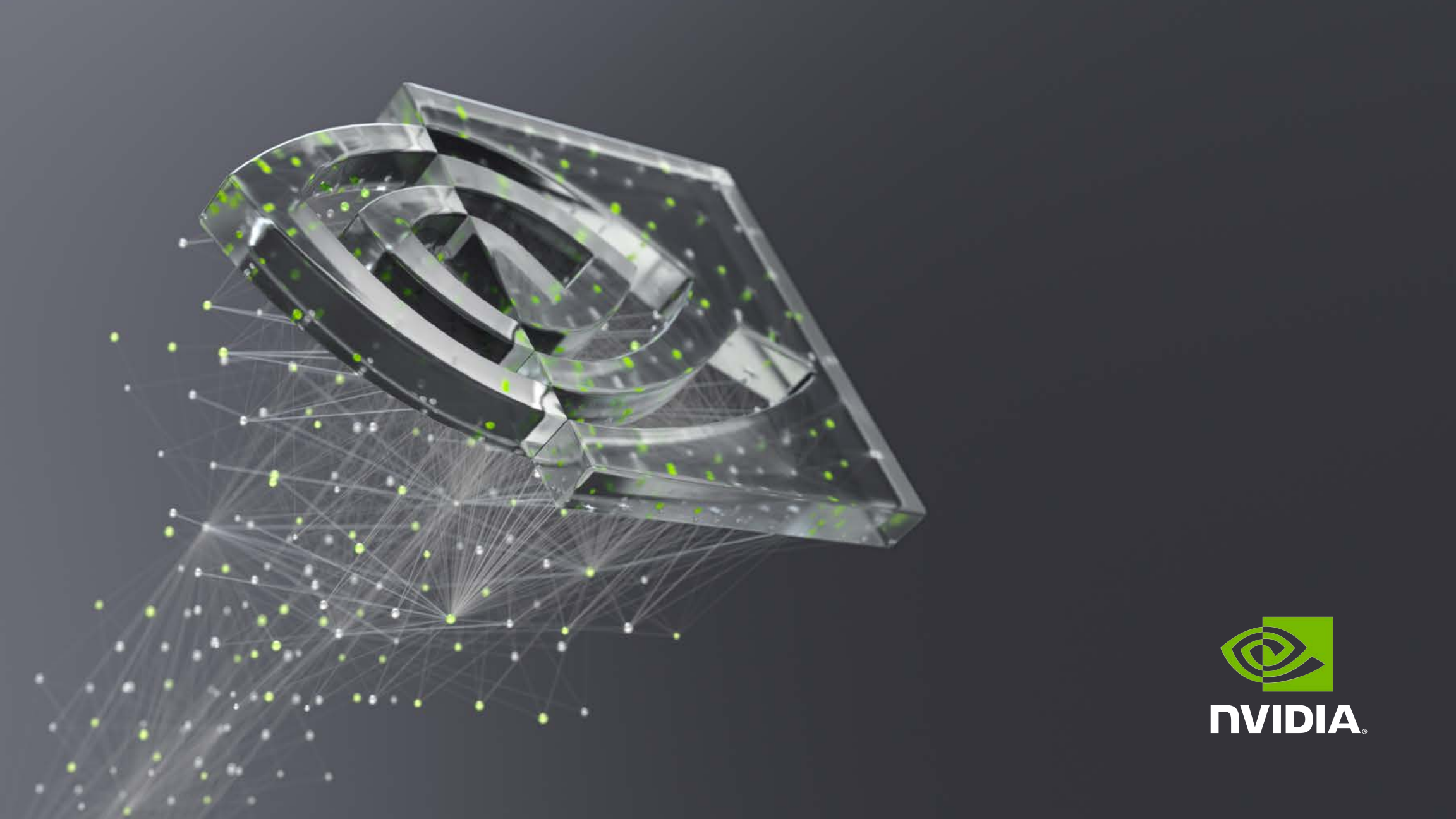
```
inline __device__ int atomicAggInc(int *p)
{
    coalesced_group g = coalesced_threads();
    int prev;
    if (g.thread_rank() == 0) {
        prev = atomicAdd(p, g.size());
    }
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

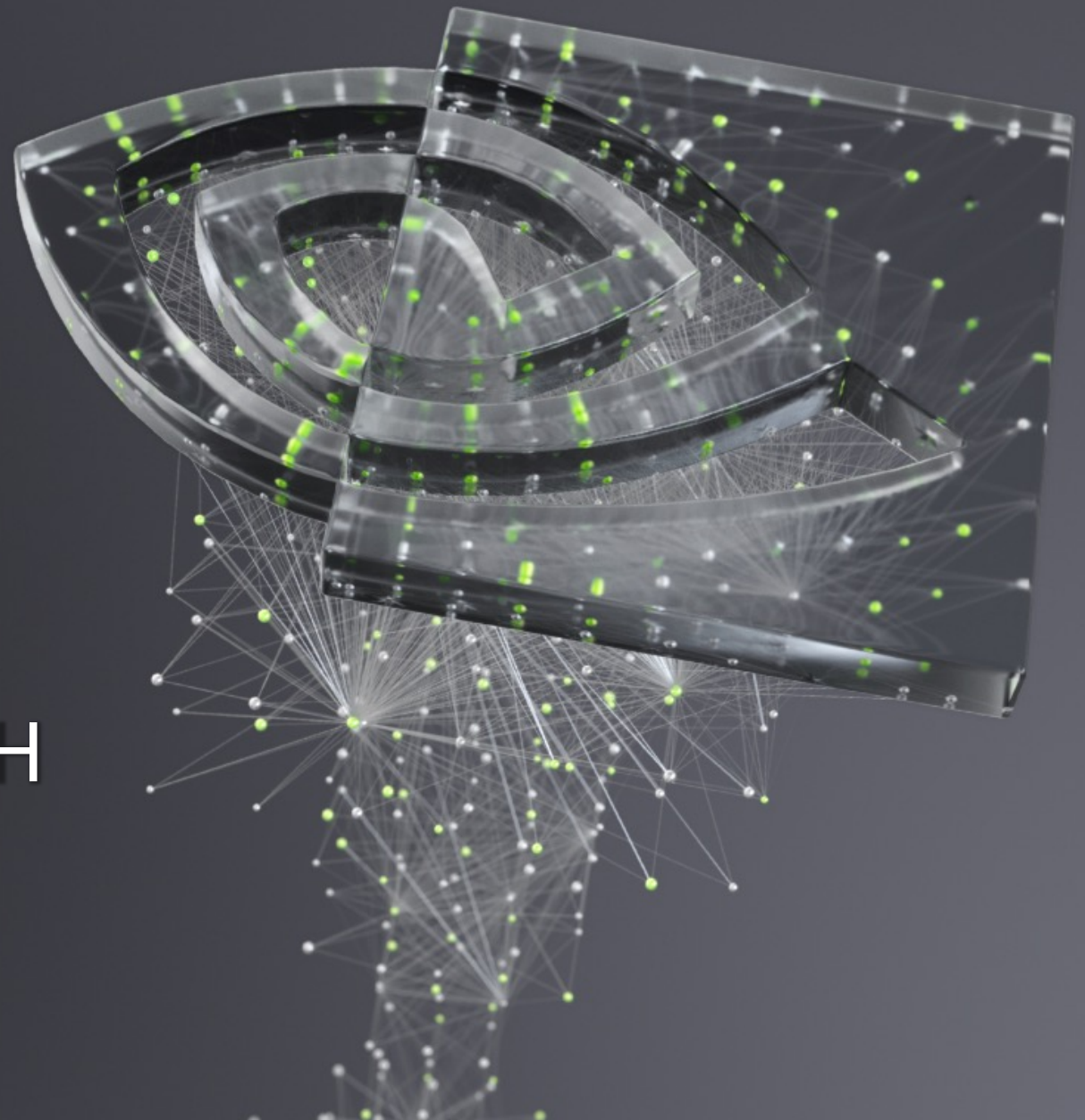
FURTHER STUDY

- GTC 2017 On-Demand Recording:
  - http://on-demand.gputechconf.com/gtc/2017/presentation/s7622-Kyrylo-perelygin-robust-and-scalable-cuda.pdf (slides)
  - http://on-demand.gputechconf.com/gtc/2017/video/s7622-perelygin-robust-scalable-cuda-parallel-programming-model.mp4 (recording)
- Sample Codes:
  - conjugateGradientMultiBlockCG, conjugateGradientMultiDeviceCG, reductionMultiBlockCG, warpAggregatedAtomicsCG
- Blog:
  - https://devblogs.nvidia.com/cooperative-groups/
- Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups
- Persistent kernels, grid sync, RNN state:
  - https://svail.github.io/persistent_rnns/

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw9/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming


NVIDIA



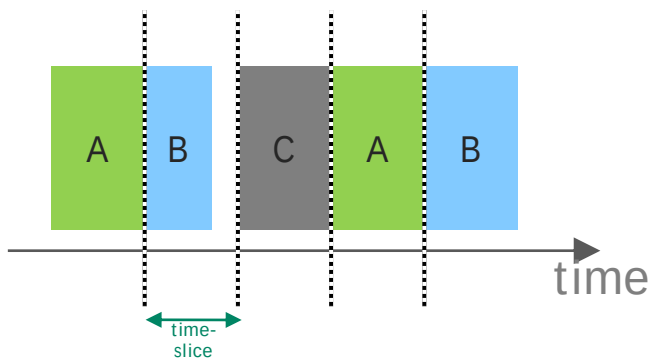
CONCURRENCY WITH MULTITHREADING

Robert Searles, 7/16/2021

EXECUTION SCHEDULING & MANAGEMENT

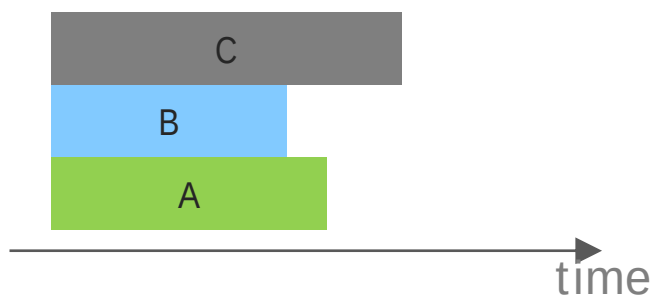
Pre-emptive scheduling

Processes share GPU through time-slicing
Scheduling managed by system



Concurrent scheduling

Processes run on GPU simultaneously
User creates & manages scheduling streams




CUDA STREAMS

STREAM SEMANTICS

1. Two operations issued into the same stream will *execute in issue-order*. Operation B issued after Operation A will not begin to execute until Operation A has completed.
2. Two operations issued into separate streams have *no ordering prescribed by CUDA*. Operation A issued into stream 1 may execute before, during, or after Operation B issued into stream 2.

- Operation: Usually, cudaMemcpyAsync or a kernel call. More generally, most CUDA API calls that take a stream parameter, as well as stream callbacks.

STREAM CREATION AND COPY/COMPUTE OVERLAP

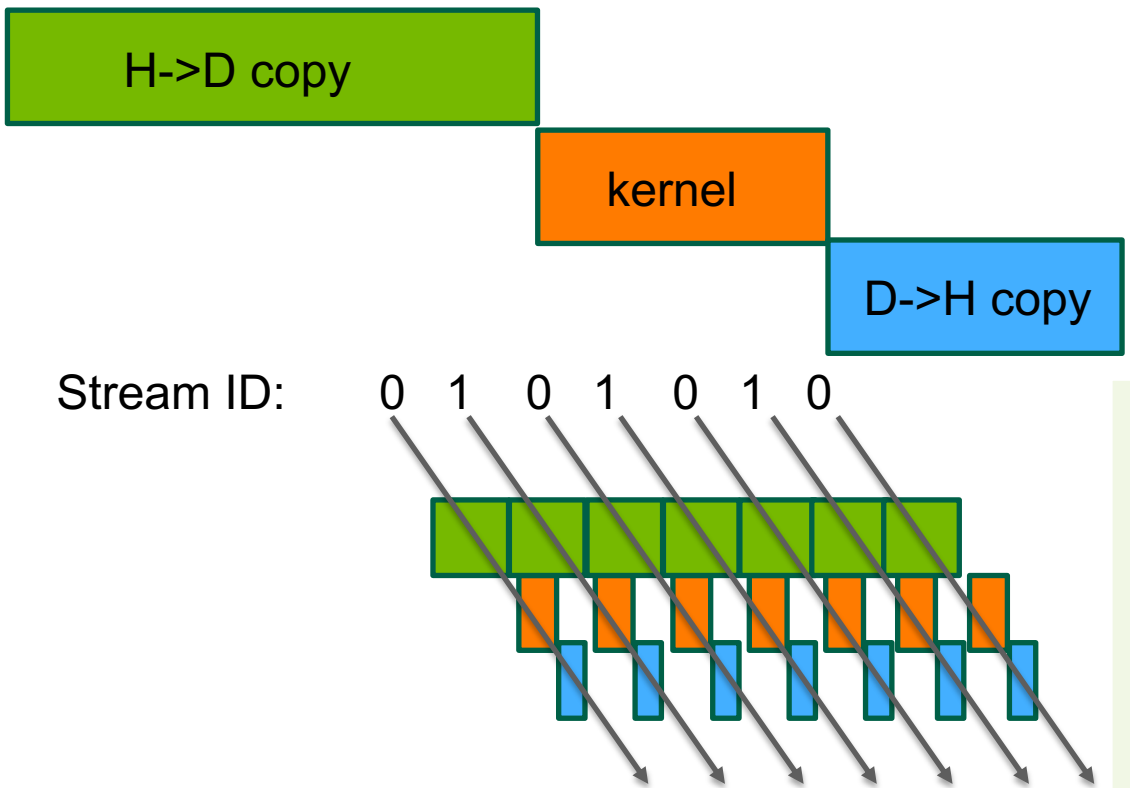
- Requirements:
  - D2H or H2D memcopy from pinned memory
  - Kernel and memcopy in different, non-0 streams
- Code:

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

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

cudaStreamQuery(stream1);          // test if stream is idle
cudaStreamSynchronize(stream2);    // force CPU thread to wait
cudaStreamDestroy(stream2);
```

potentially overlapped

EXAMPLE STREAM BEHAVIOR FOR VECTOR MATH

(assumes algorithm decomposability)



Similar: video processing pipeline

non-streamed

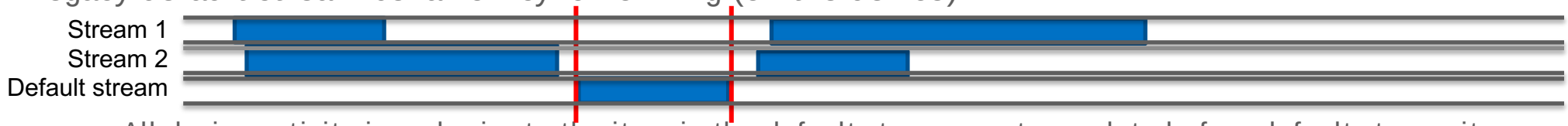
```
cudaMemcpy(d_x, h_x, size_x,
cudaMemcpyHostToDevice);
Kernel<<<b, t>>>(d_x, d_y, N);
cudaMemcpy(h_y, d_y, size_y,
cudaMemcpyDeviceToHost);
```

streamed

```
for (int i = 0, i<c; i++){
  size_t offx = (size_x/c)*i;
  size_t offy = (size_y/c)*i;
  cudaMemcpyAsync(d_x+offx, h_x+offx,
size_x/c, cudaMemcpyHostToDevice,
stream[i%ns]);
  Kernel<<<b/c, t, 0,
stream[i%ns]>>>(d_x+offx, d_y+offy,
N/c);
  cudaMemcpyAsync(h_y+offy, d_y+offy,
size_y/c, cudaMemcpyDeviceToHost,
stream[i%ns]);}
```

DEFAULT STREAM

- Kernels or cudaMemcpy… that do not specify stream (or use 0 for stream) are using the default stream
- Legacy default stream behavior: synchronizing (on the device):



  - All device activity issued prior to the item in the default stream must complete before default stream item begins
  - All device activity issued after the item in the default stream will wait for the default stream item to finish
  - All host threads share the same default stream for legacy behavior
  - Consider avoiding use of default stream during complex concurrency scenarios
- Behavior can be modified to convert it to an “ordinary” stream
  - nvcc --default-stream per-thread …
  - Each host thread will get its own “ordinary” default stream

OTHER CONCURRENCY SCENARIOS

- Host/Device execution concurrency:

```
Kernel<<<b, t>>>(…);   // this kernel execution can overlap with
cpuFunction(…);        // this host code
```

- Concurrent kernels:

```
Kernel<<<b, t, 0, streamA>>>(…);   // these kernels have the possibility
Kernel<<<b, t, 0, streamB>>>(…);   // to execute concurrently
```

  - In practice, concurrent kernel execution on the same device is hard to witness
  - Requires kernels with relatively low resource utilization and relatively long execution time
  - There are hardware limits to the number of concurrent kernels per device
  - Less efficient than saturating the device with a single kernel

MPI DECOMPOSITION

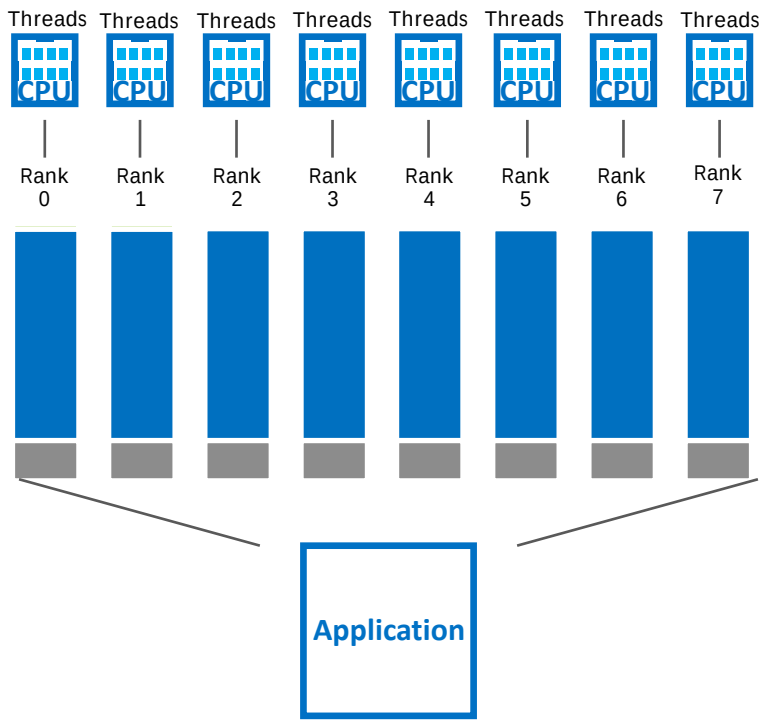
Very common in HPC

Many legacy codes use MPI + OpenMP

MPI handles inter-node communcation

OpenMP provides better shared memory multithreading within each node

How can we add GPUs into the mix?



MULTITHREADING + CUDA STREAMS

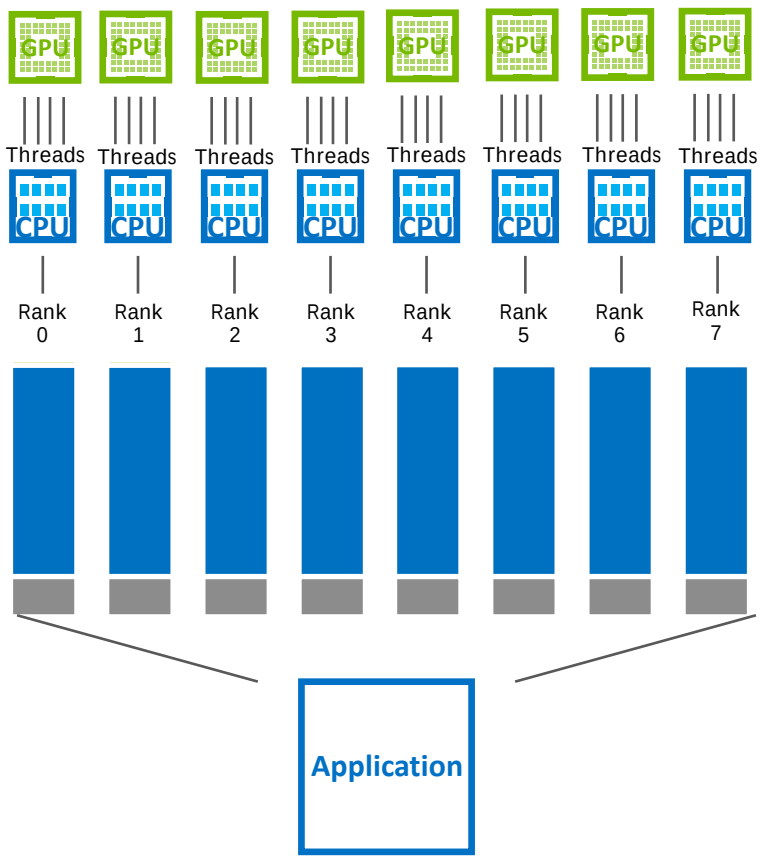
Easier than rewriting entire legacy code

Individual OpenMP threads may still have a significant amount of work

Streams allow multiple threads to submit kernels for concurrent execution on a single GPU

Not possible pre-R465

Supported starting with CUDA 11.4/R470



SINGLE GPU EXAMPLE

- Multithreading + Concurrent kernels:

```
cudaStream_t streams[num_streams];
for (int j=0; j<num_streams; j++)
    cudaStreamCreate(&streams[j]);

#pragma omp parallel for
for (int i=0; i<N; i++)                                  // execute concurrently across
    Kernel<<<b/N, t, 0, streams[i % num_streams]>>>(…);  // threads + streams
```

  - Worth it if each thread has enough work to offset kernel launch overhead
  - Requires less programmer overhead than rewriting entire codebase to submit single, large kernels to each GPU (remove OpenMP and replace with CUDA)
  - Less efficient than saturating the device with streams from a single thread
  - Less efficient than saturating the device with a single kernel

MULTI-GPU – STREAMS

- Streams (and cudaEvent) have implicit/automatic *device association*
- Each device also has its own unique default stream
- Kernel launches will fail if issued into a stream not associated with current device
- cudaStreamWaitEvent() can synchronize streams belonging to separate devices, cudaEventQuery() can test if an event is “complete”
- Simple device concurrency:

```
cudaSetDevice(0);
cudaStreamCreate(&stream0);         //associated with device 0
cudaSetDevice(1);
cudaStreamCreate(&stream1);         //associated with device 1
Kernel<<<b, t, 0, stream1>>>(…);    // these kernels have the possibility
cudaSetDevice(0);
Kernel<<<b, t, 0, stream0>>>(…);    // to execute concurrently
```

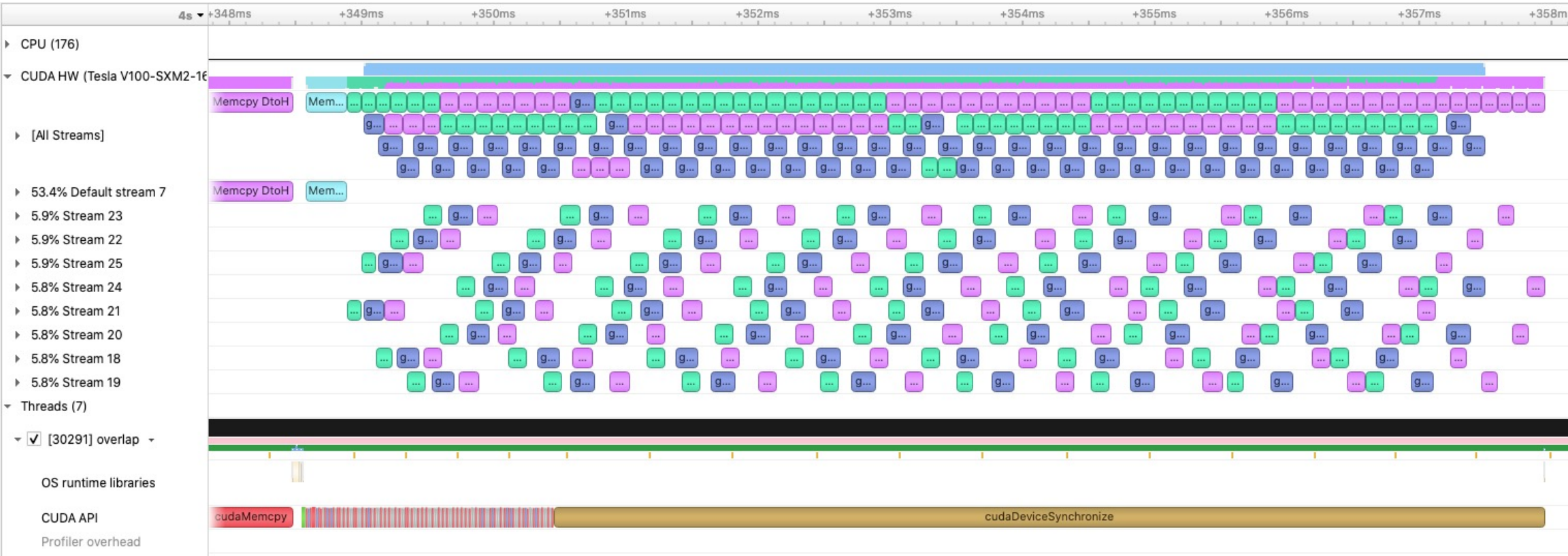
MULTI-GPU EXAMPLE

- Multithreading + Concurrent kernels:

```
cudaStream_t streams[num_streams];
#pragma omp parallel for
for (int i=0; i<N; i++){
    int j = i % num_streams;            // Stream number
    cudaSetDevice(j % num_gpus);        // Round-robin across on-node GPUs
    cudaStreamCreate(&streams[j]);     // Associated with device j % num_gpus
    Kernel<<<b/N, t, 0, streams[j]>>>(…);} // execute across threads/streams/GPUs
```

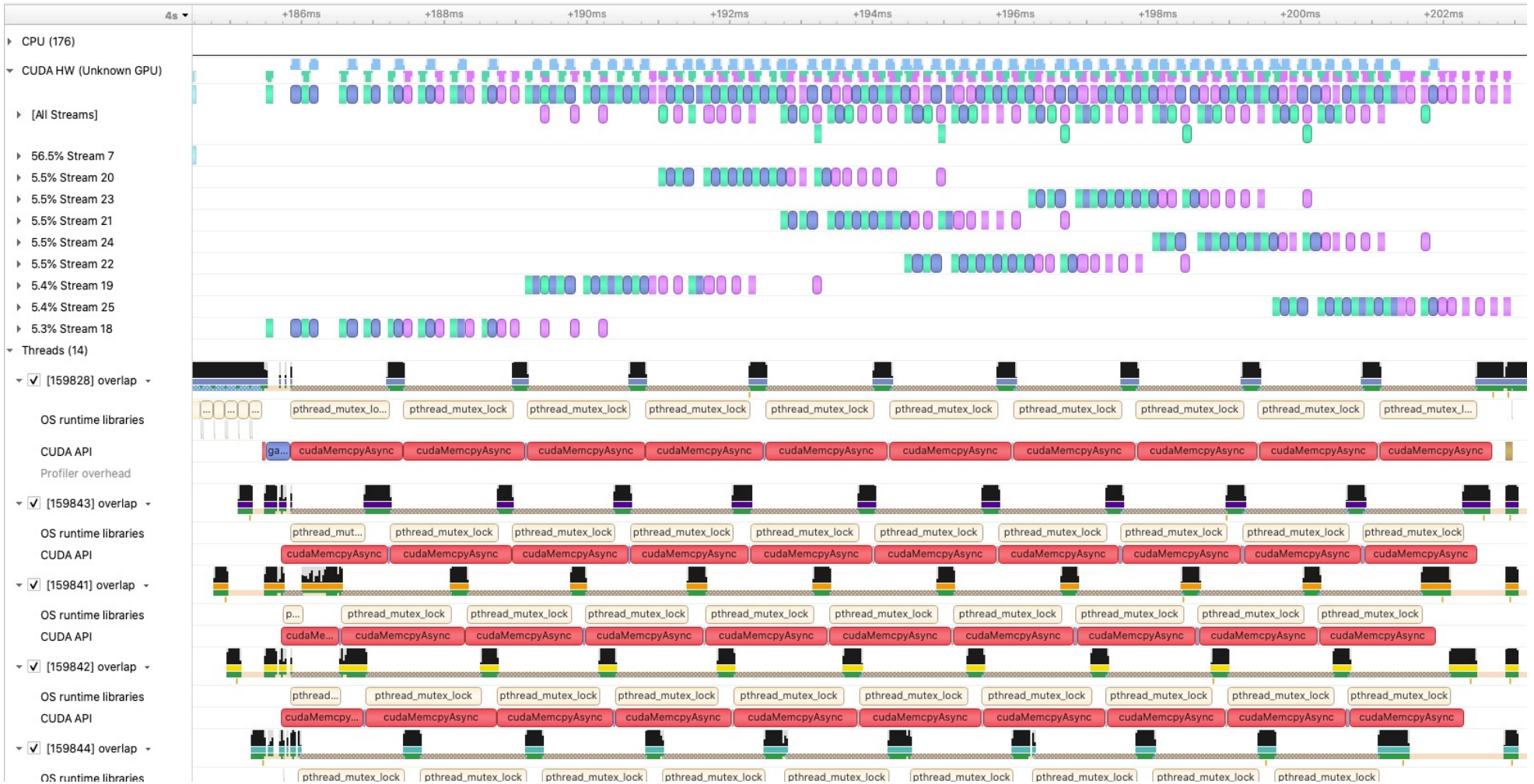
  - Multiple threads submitting kernels across a number of streams distributed across available GPUs
  - Example: 16 threads, 64 streams, 8 GPUs and N=1024
    - 8 streams per GPU, 16 kernels per stream
  - Should have at least 1 stream per GPU
    - More will be optimal; Need as many streams on a GPU as it takes concurrent kernels to saturate that GPU

SINGLE THREAD + CUDA STREAMS



NVIDIA

MULTITHREADING + CUDA STREAMS



NVIDIA

MULTITHREADING + CUDA STREAMS

- Runtimes
  - Single Thread + Default Stream = 0.01879s
  - Single Thread + 8 CUDA Streams = 0.00781s
  - 8 OpenMP Threads + 8 CUDA Streams (without profiling) = 0.00835s
  - 8 OpenMP Threads + 8 CUDA Streams (with profiling) = 0.01798s
- Issue with serialization when using the profiler
  - We're working on that

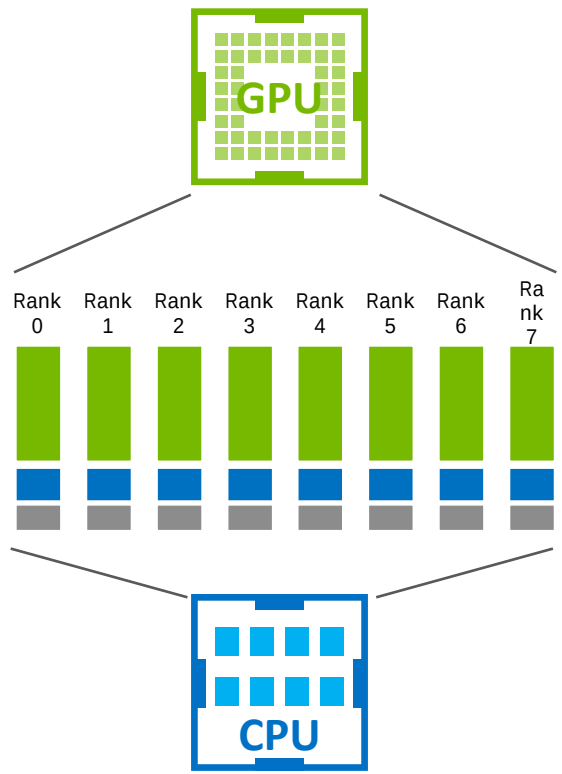
MULTI-PROCESS SERVICE (MPS) OVERVIEW

Better solution in terms of performance

Designed to concurrently map multiple MPI ranks onto a single GPU

Used when each rank is too small to fill the GPU on its own

On Summit, use *–alloc_flags=gpumps* when submitting a job with *bsub*

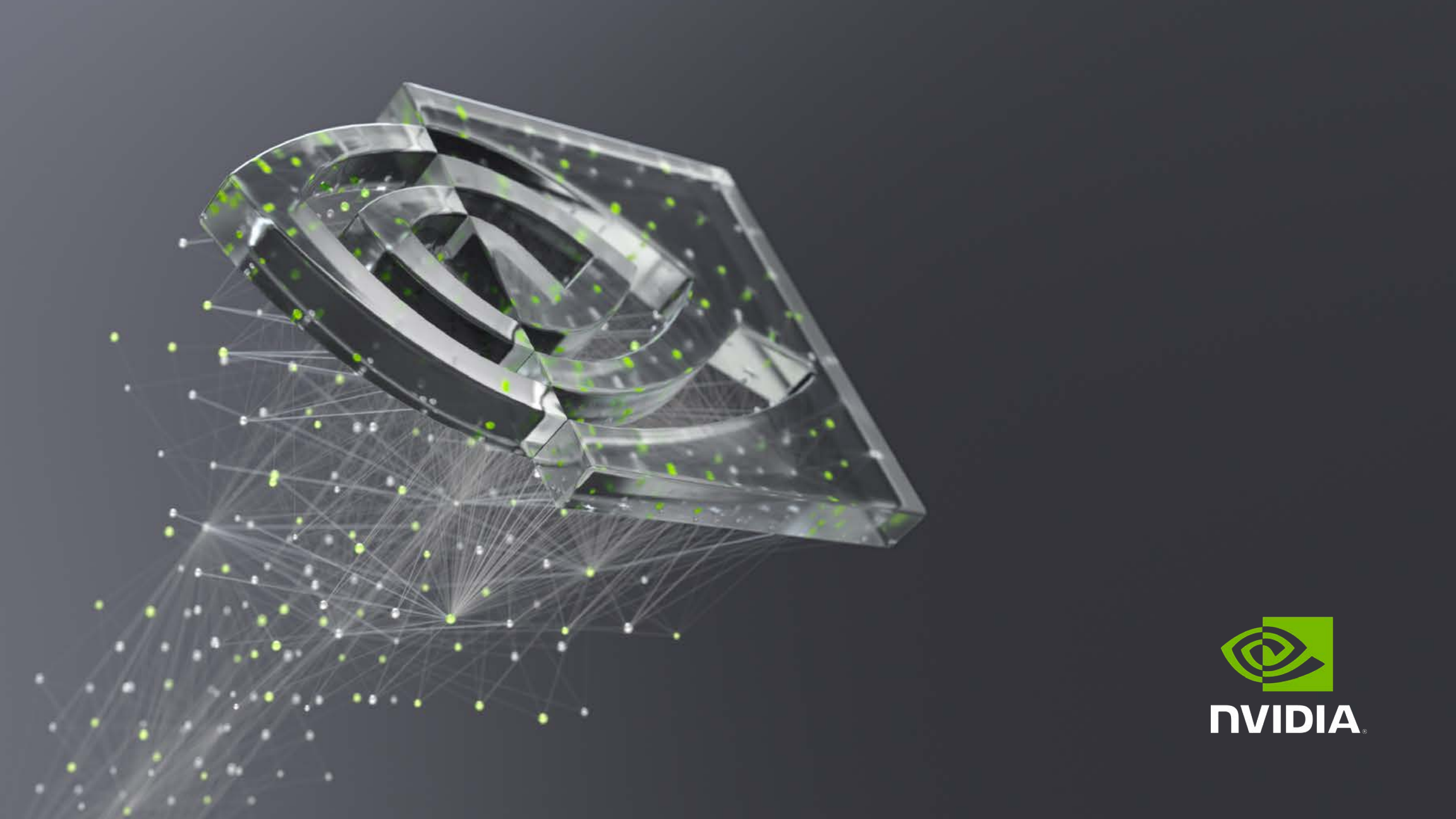


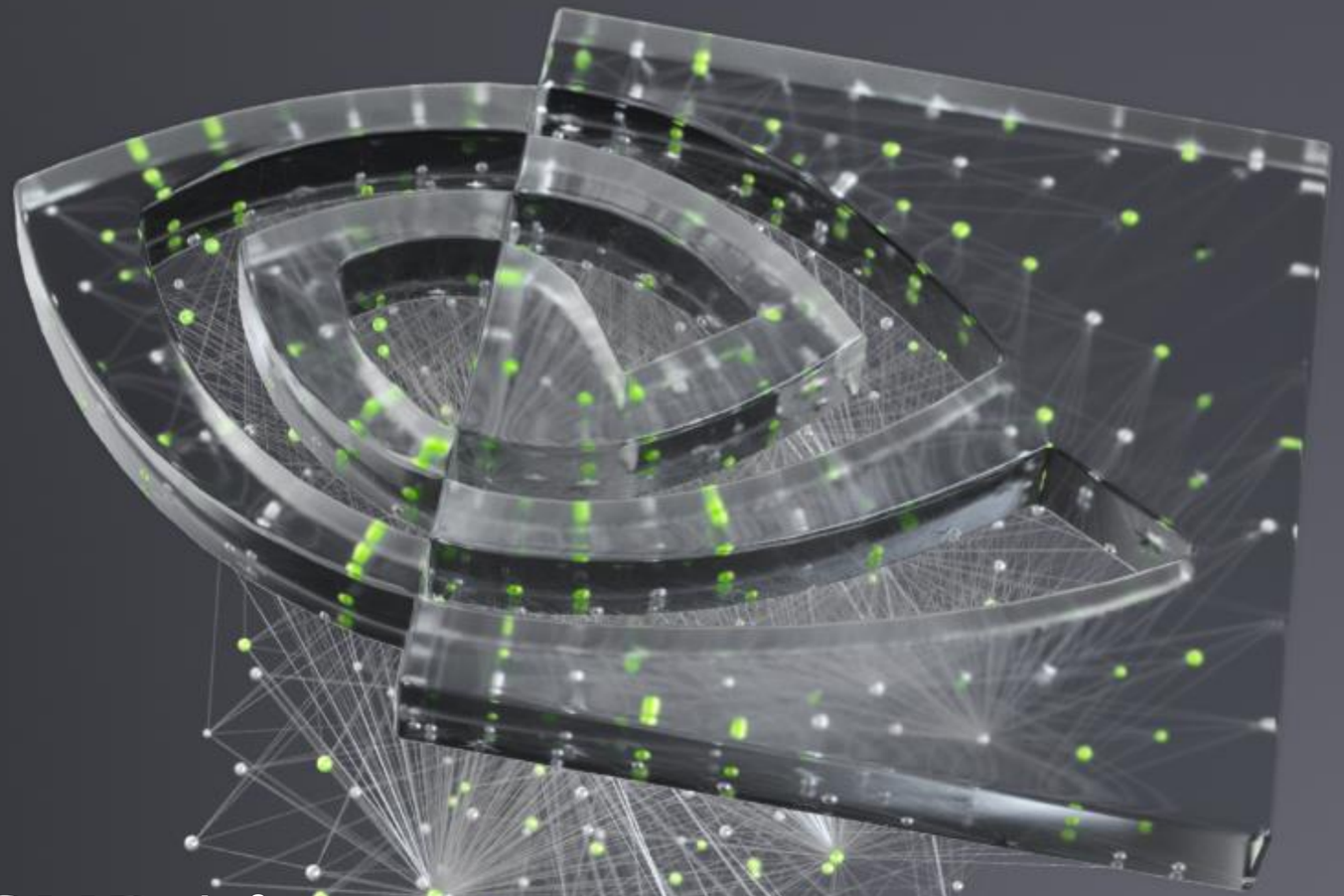
FUTURE SESSIONS

- MPI/MPS
- CUDA Debugging

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw10/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming


NVIDIA



INTRODUCTION TO CUDA's MULTI-PROCESS SERVICE (MPS)

MOTIVATING USE CASE

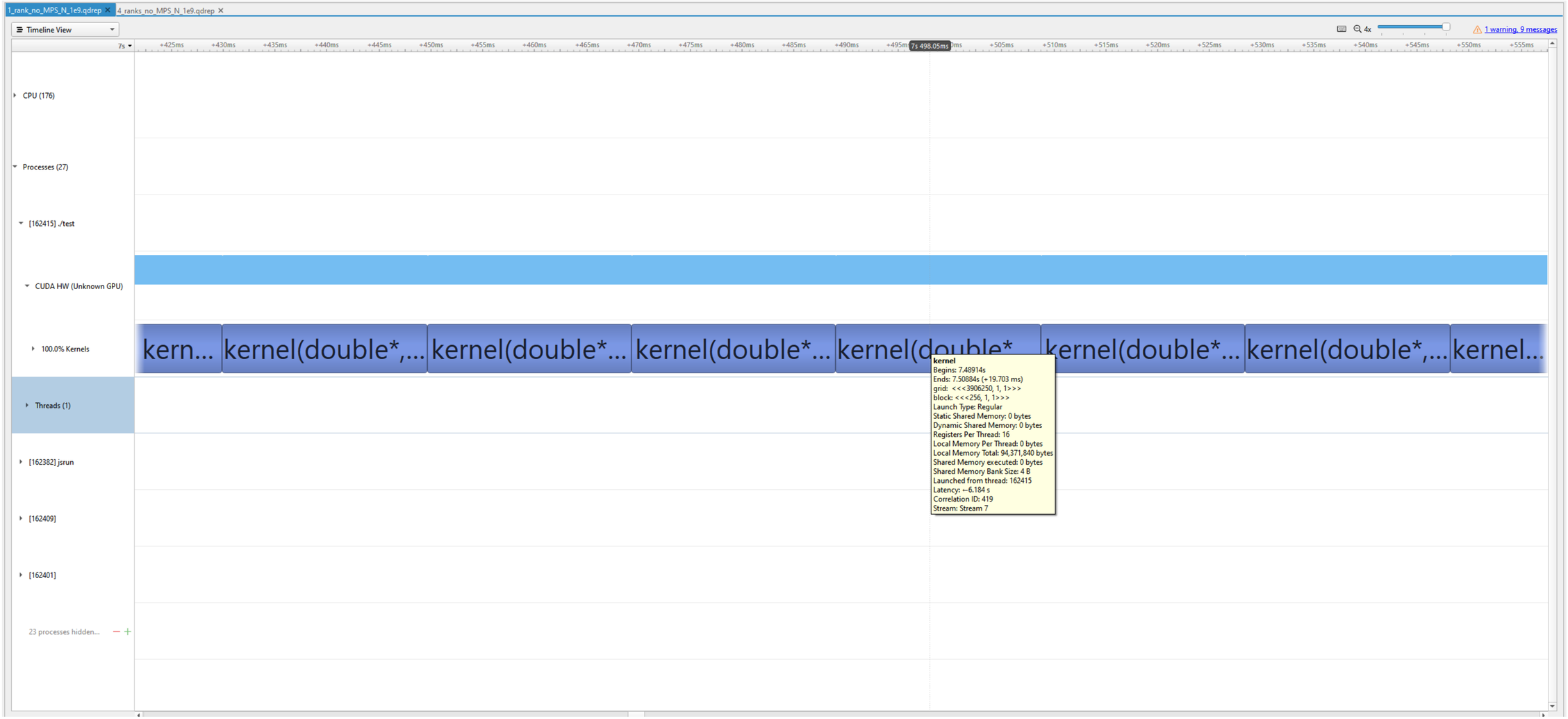
Given a fixed amount of work to do, divided evenly among N MPI ranks:

- What is the optimal value of N?
- How many GPUs should we distribute these N ranks across?

```
__global__ void kernel (double* x, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N) {
        x[i] = 2 * x[i];
    }
}
```

BASE CASE: 1 RANK

Run with N = 1024^3



GPU COMPUTE MODES

NVIDIA GPUs have several compute modes

Default: multiple processes can run at one time

Exclusive Process: only one process can run at one time

Prohibited: no processes can run

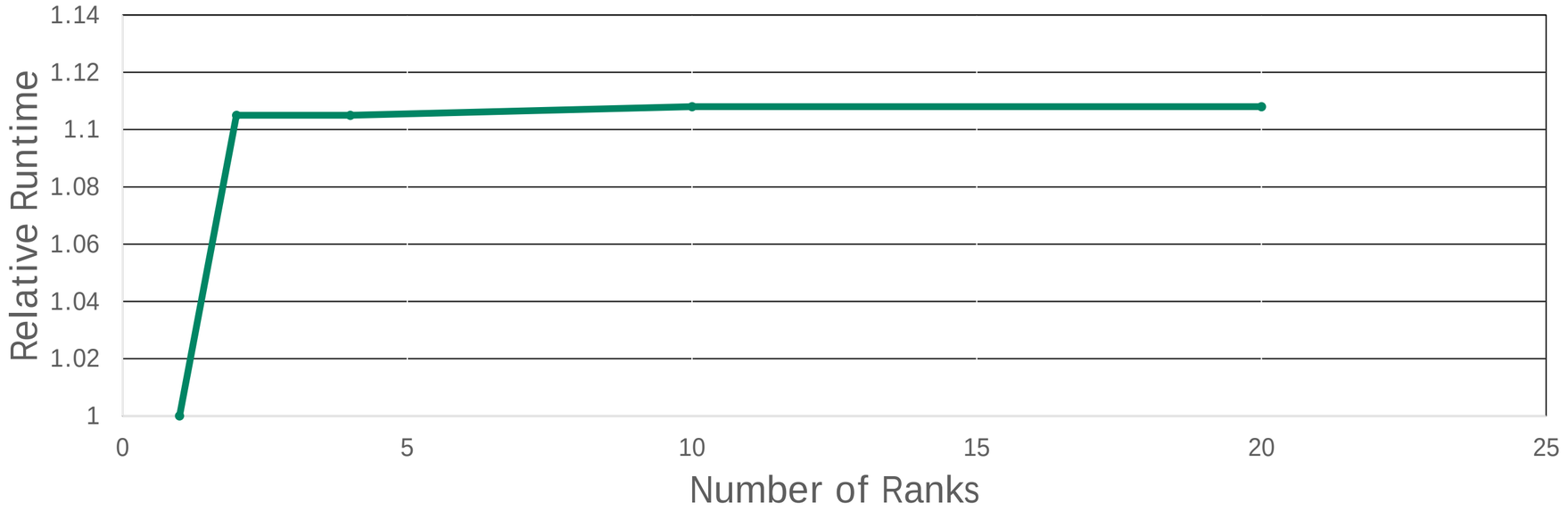
Controllable with `nvidia-smi --compute-mode`; generally needs elevated privileges (so e.g. `bsub -alloc_flags gpudefault` on Summit)

SIMPLE OVERSUBSCRIPTION

The most common oversubscription case uses default mode

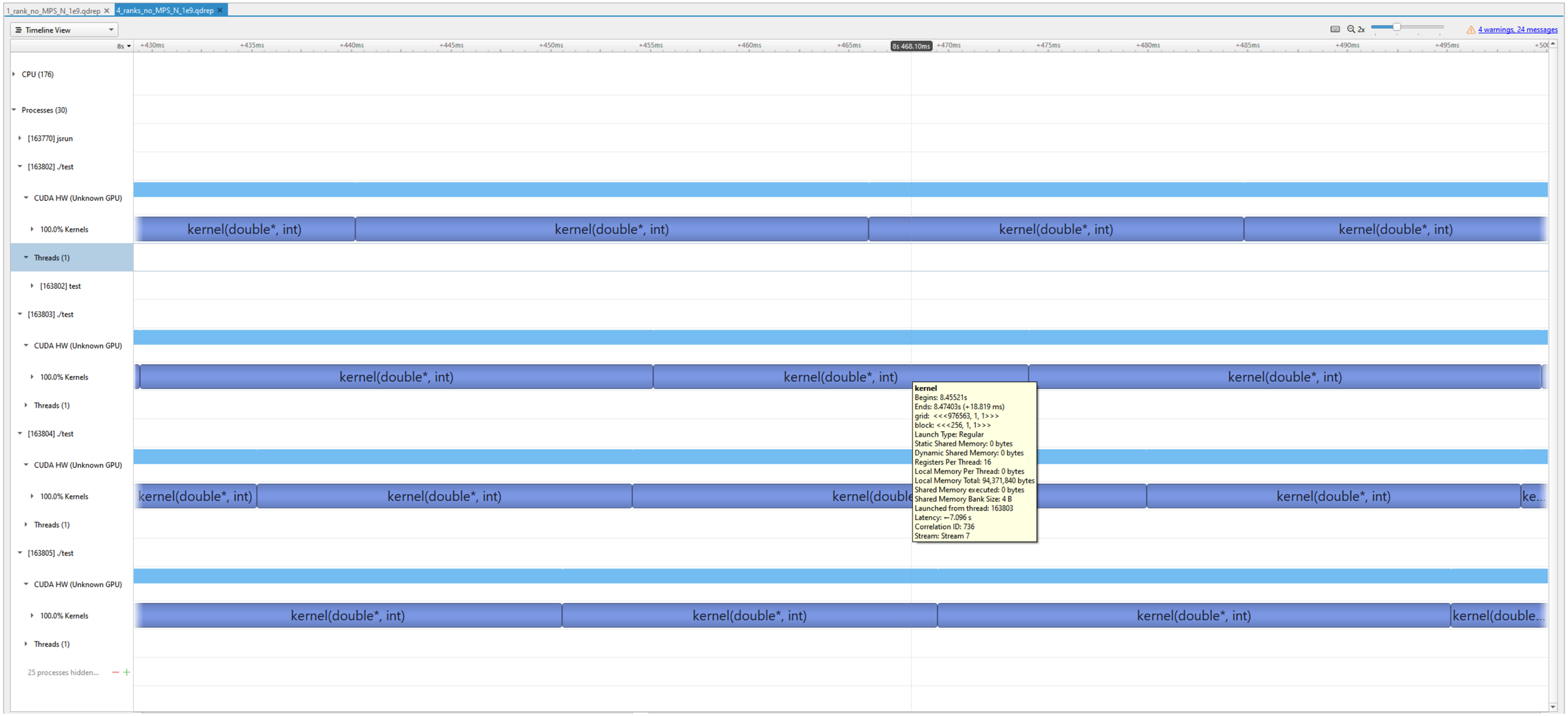
We simply target the same GPU with N ranks

```
$ jsrun -n 1 -a <NUM_RANKS> -g 1 –c <NUM_RANKS> ./test 1073741824
```



OVERSUBSCRIPTION: 4 RANKS

Run with N = 1024^3

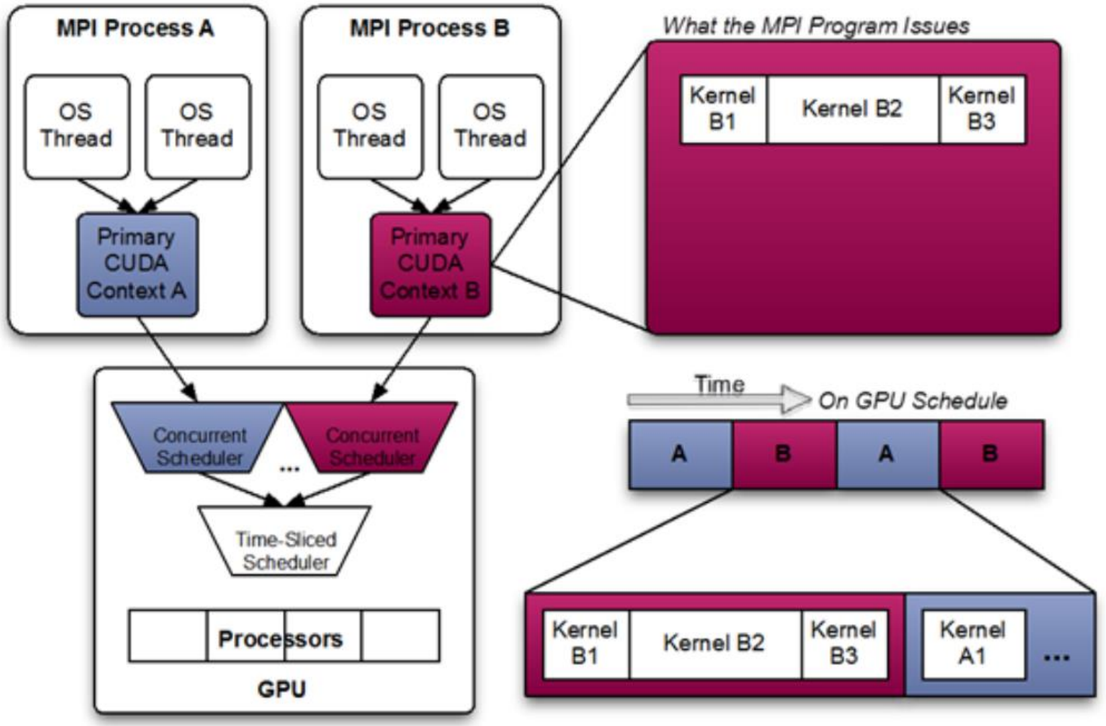


SIMPLE OVERSUBSCRIPTION

Each rank operates fully independently of all other ranks

Individual processes operate in time slices

- A performance penalty is paid for switching between time slices

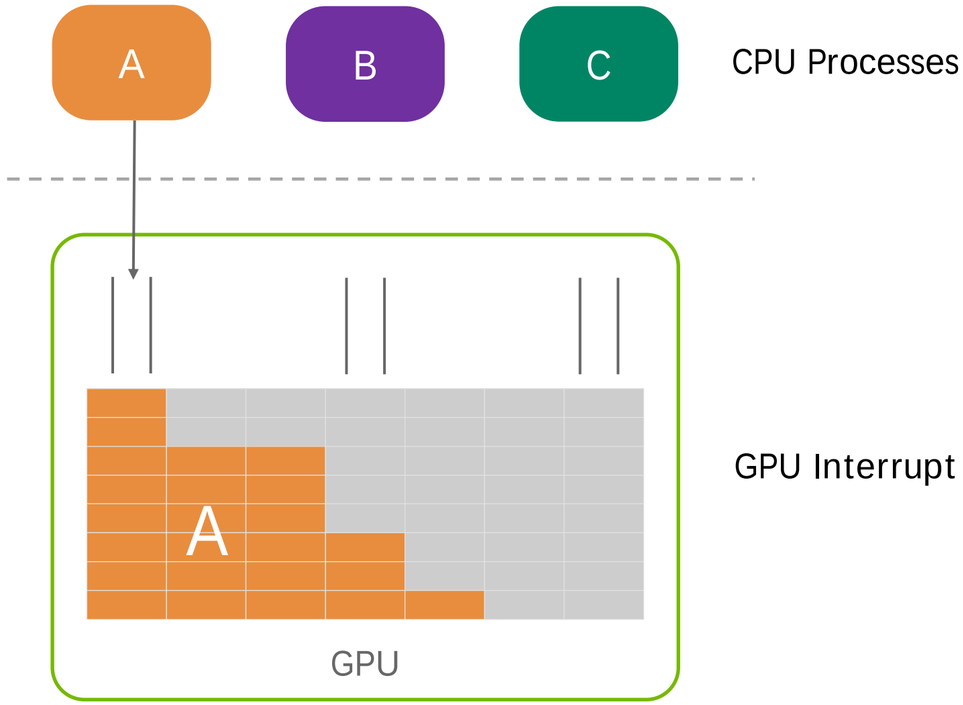


ASIDE: CUDA CONTEXTS

Every process creates its own *CUDA context*

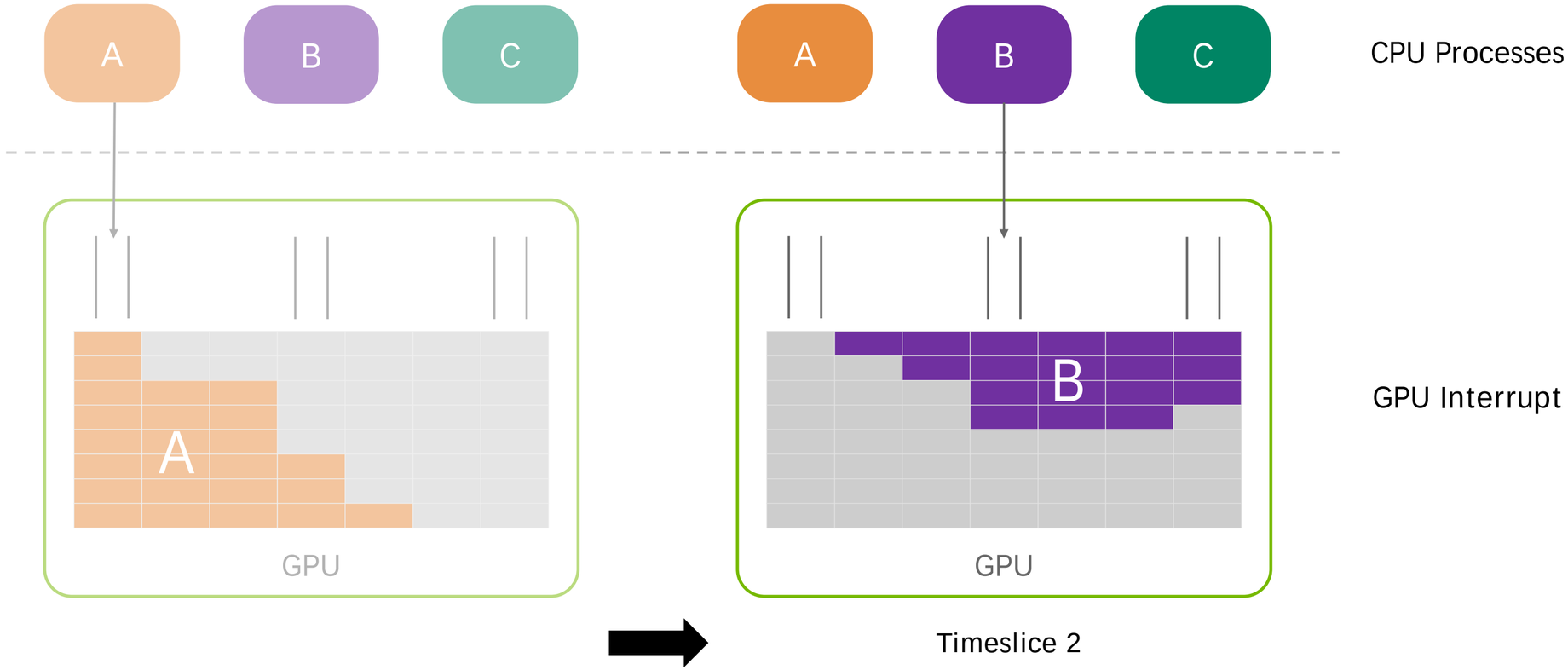
- The context is a stateful object required to run CUDA
- Automatically created for you when using the CUDA runtime API
- On V100, the size is ~300 MB + your GPU code size
- This limits the number of ranks we can fit on the GPU regardless of application data
- Context size is partially controlled by `cudaLimitStackSize` (more on that later)

MULTI-PROCESS TIMESLICING

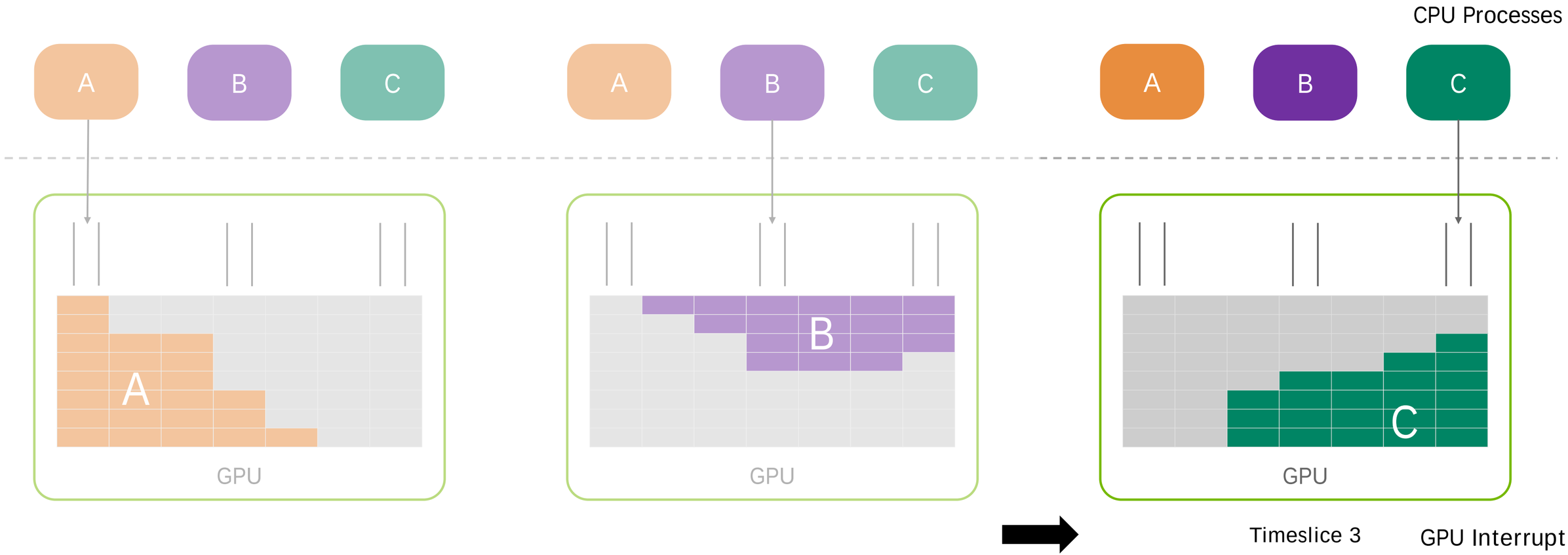


Timeslice 1

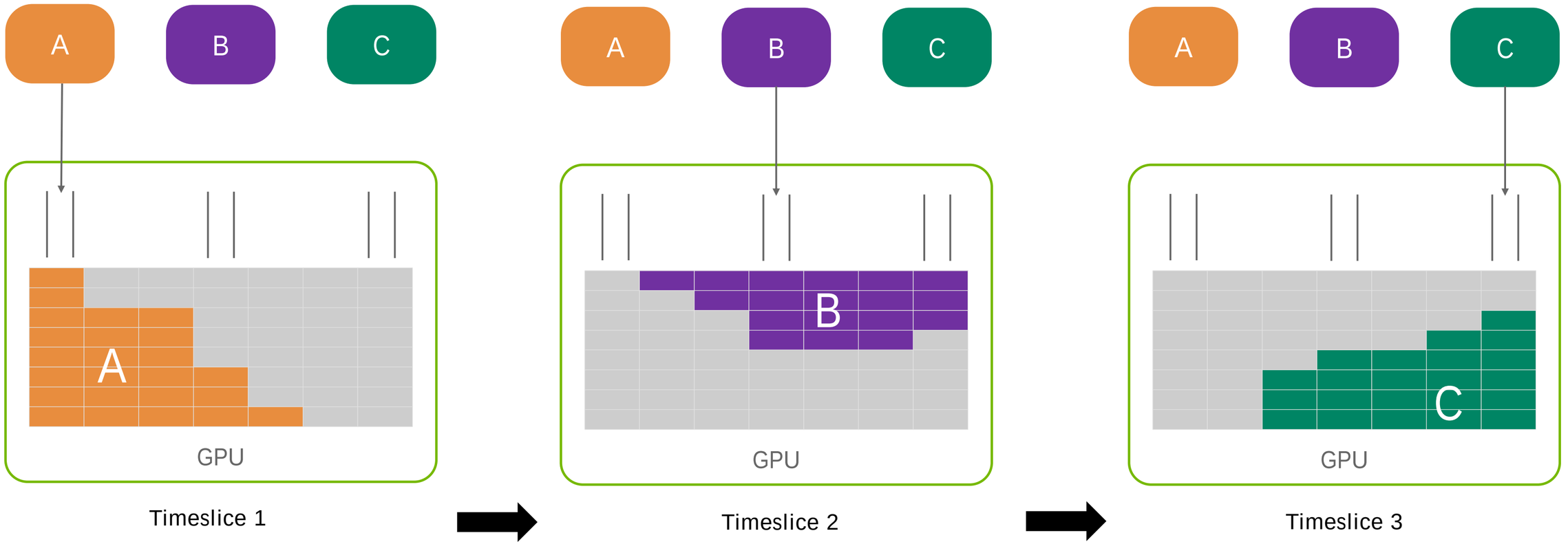
MULTI-PROCESS TIMESLICING



MULTI-PROCESS TIMESLICING



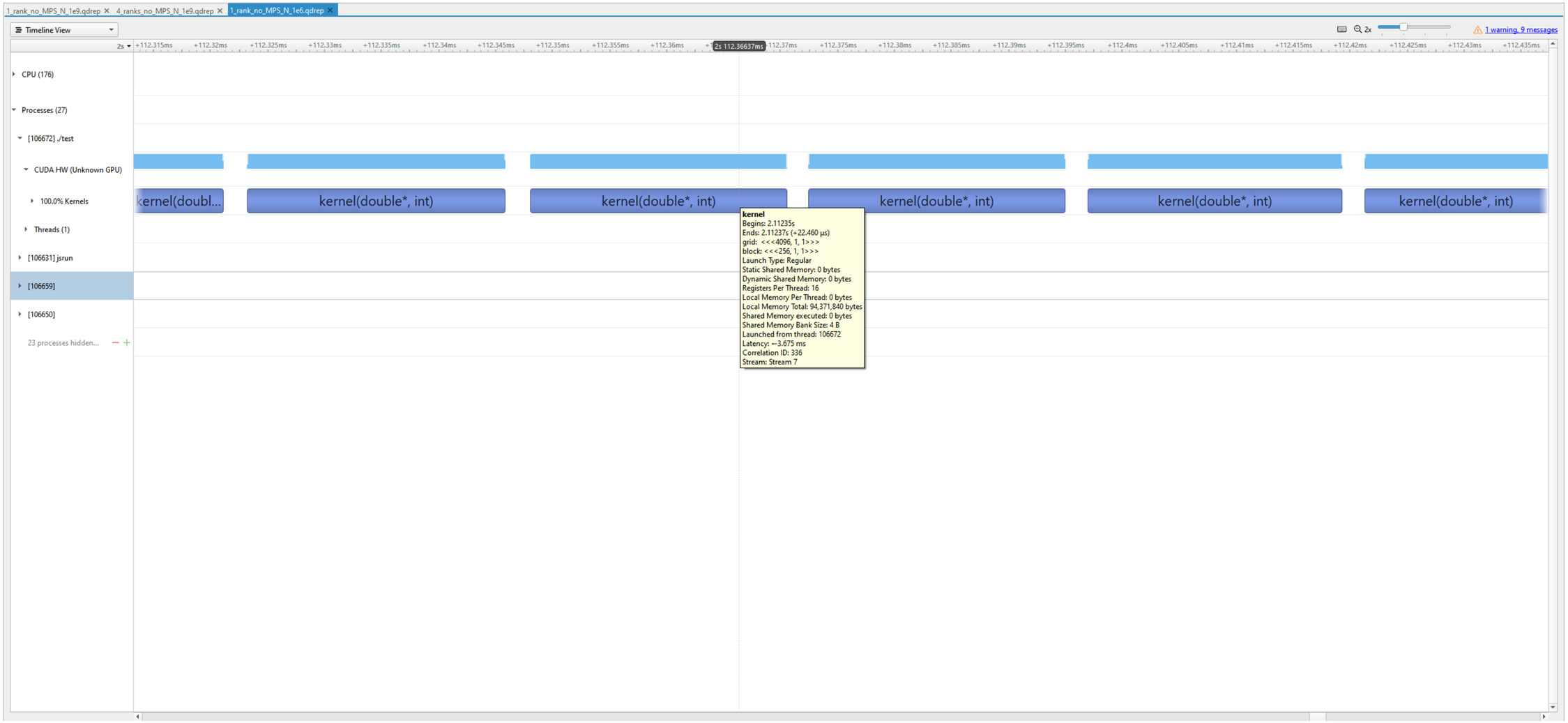
MULTI-PROCESS TIMESLICING



Full process isolation, peak throughput optimized for each process

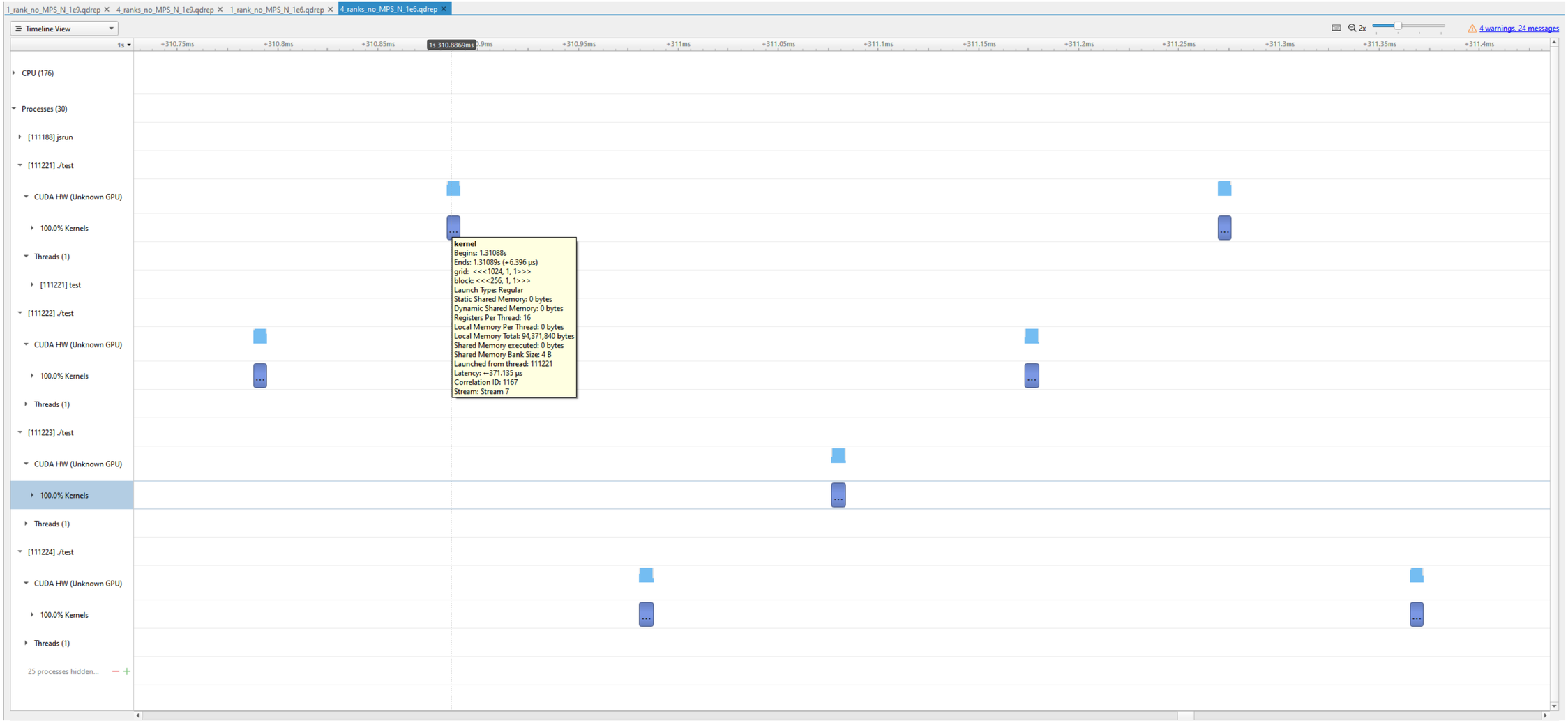
WHEN DOES OVERSUBSCRIPTION HELP?

Perhaps a smaller case where launch latency is relevant? (N = 10^6)



WHEN DOES OVERSUBSCRIPTION HELP?

Unfortunately, this isn't better.



OVERSUBSCRIPTION CONCLUSIONS

(when running with the default compute mode)

No free lunch theorem applies: if GPU is fully utilized, cannot get faster answers

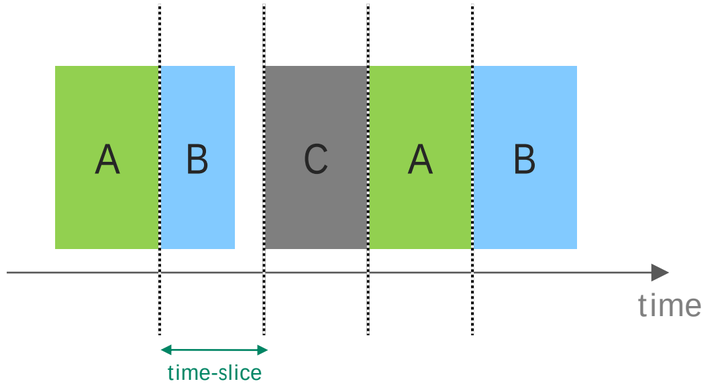
For cases that don't fully utilize the GPU, we'd like to fill in gaps in the timeline

- But with GPU-only workloads, this rarely works out just right to be beneficial
- Typically performs better when there is CPU-only work to interleave

SCHEDULING: HOW COULD WE DO BETTER?

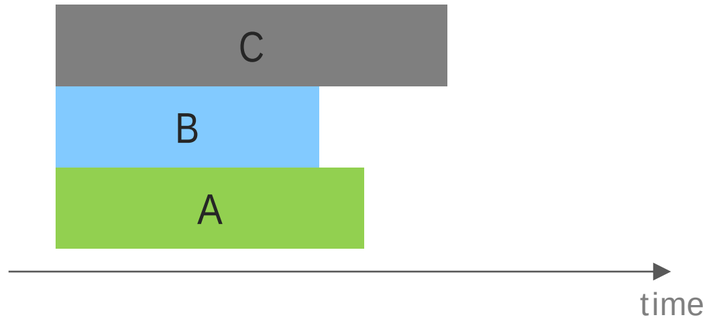
Pre-emptive scheduling

Processes share GPU through time-slicing
Scheduling managed by system



Concurrent scheduling

Processes run on GPU simultaneously
User creates & manages scheduling streams

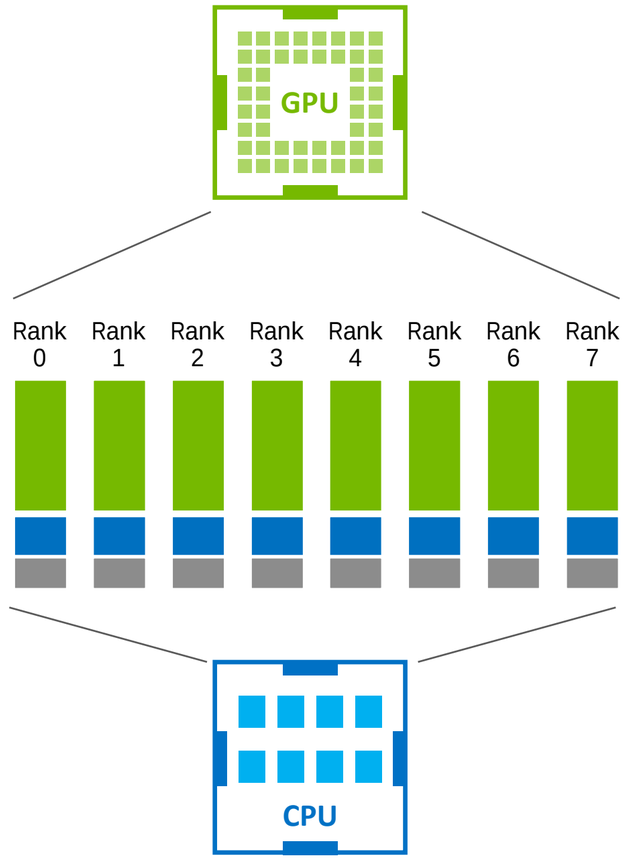


MULTI-PROCESS SERVICE

NVIDIA MPS (Multi-Process Service) improves the situation by allowing multiple process to (instantaneously) share GPU compute resources (SMs)

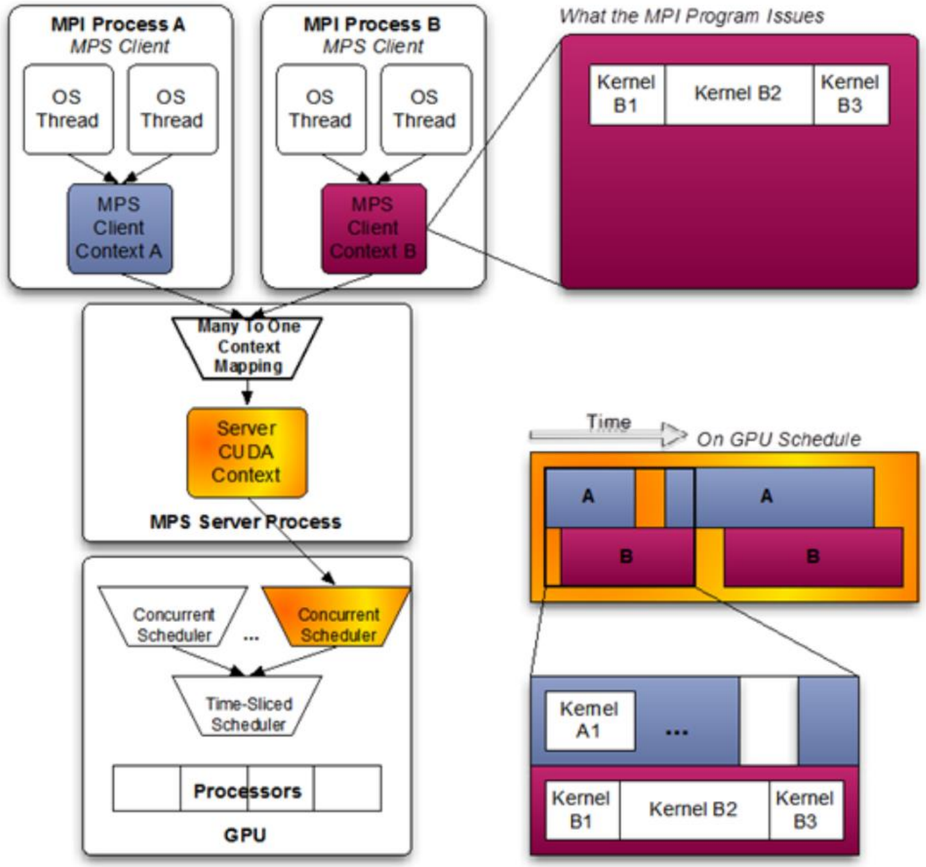
Designed to concurrently map multiple MPI ranks onto a single GPU

Used when each rank is too small to fill the GPU on its own



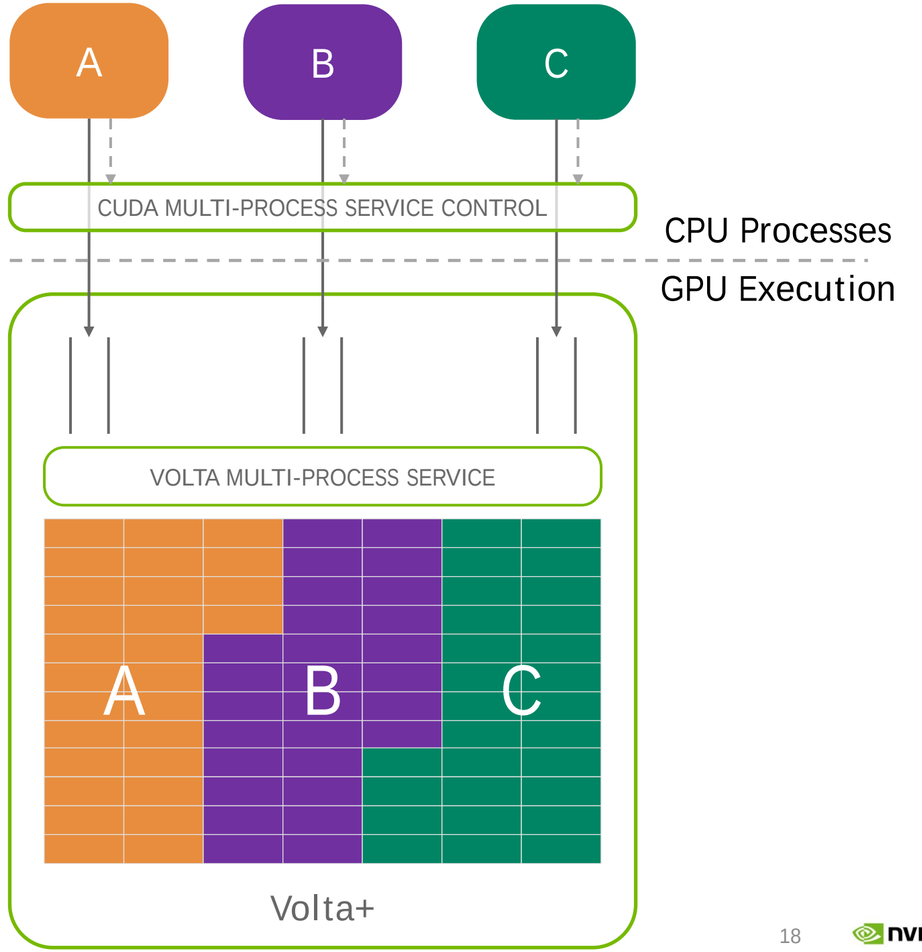
MULTI-PROCESS SERVICE

Improving on what we had before!



Hardware Accelerated Work Submission

Hardware Isolation

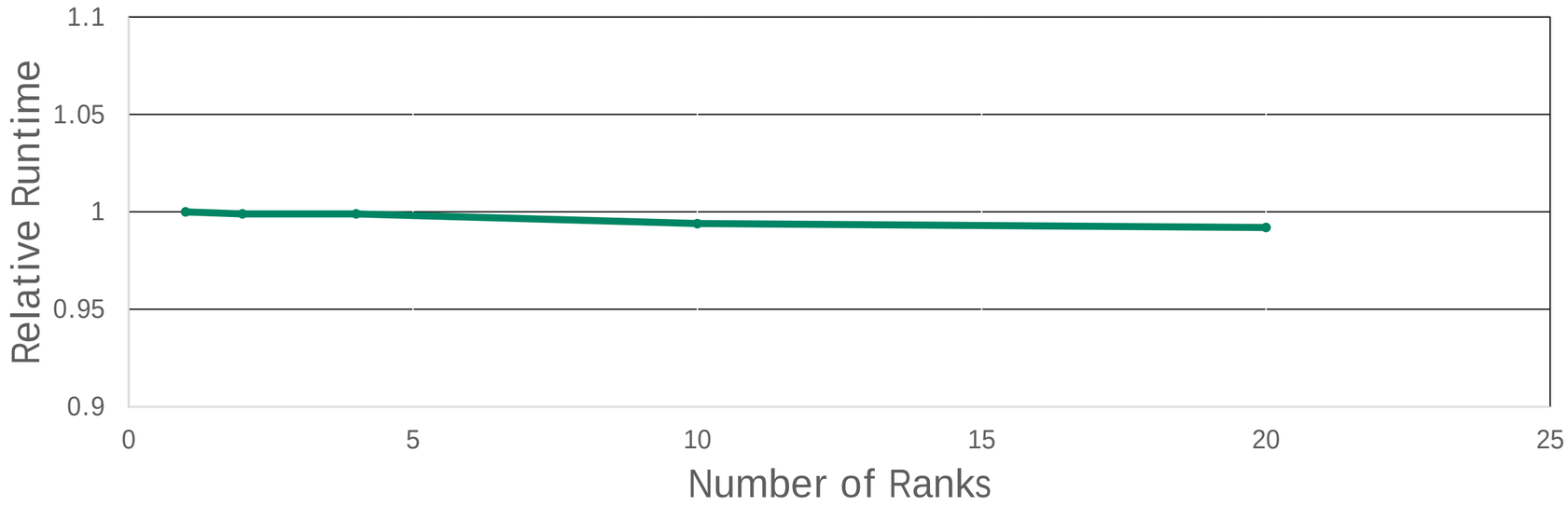


OVERSUBSCRIPTION WITH MPS

Same case as earlier with N = 10^9

MPS mostly recovers performance losses due to context switching

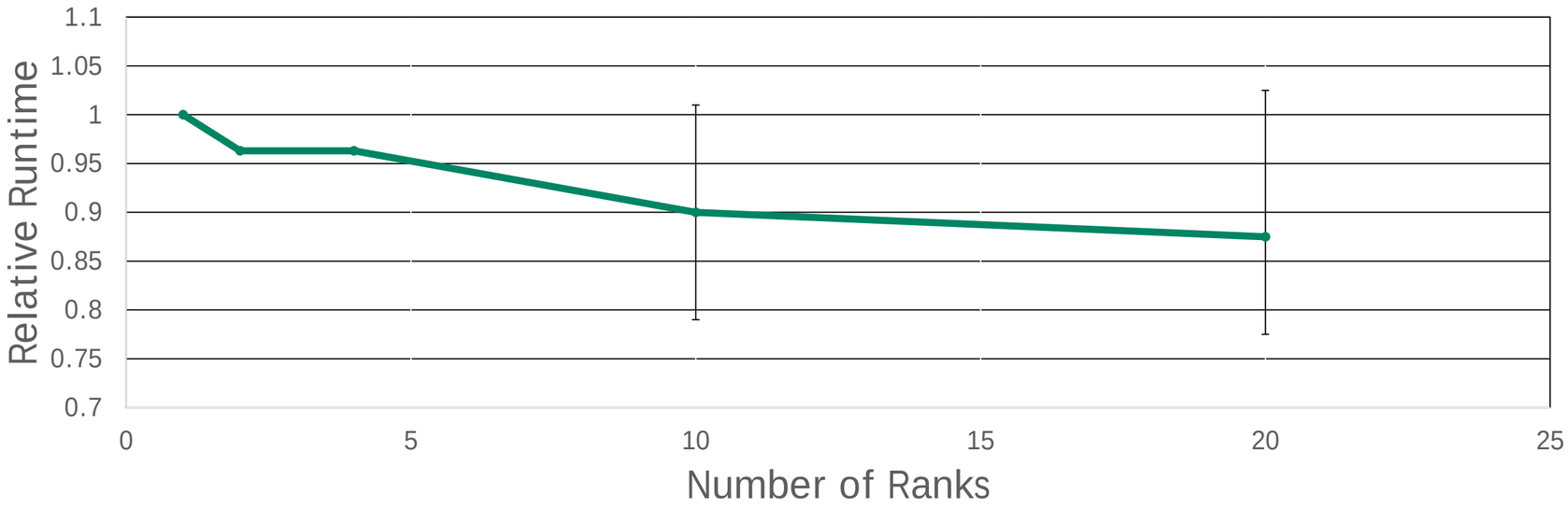
But again, no free lunch theorem applies (no significant speedup either)



OVERSUBSCRIPTION WITH MPS

A smaller case: N = 2 * 10^7

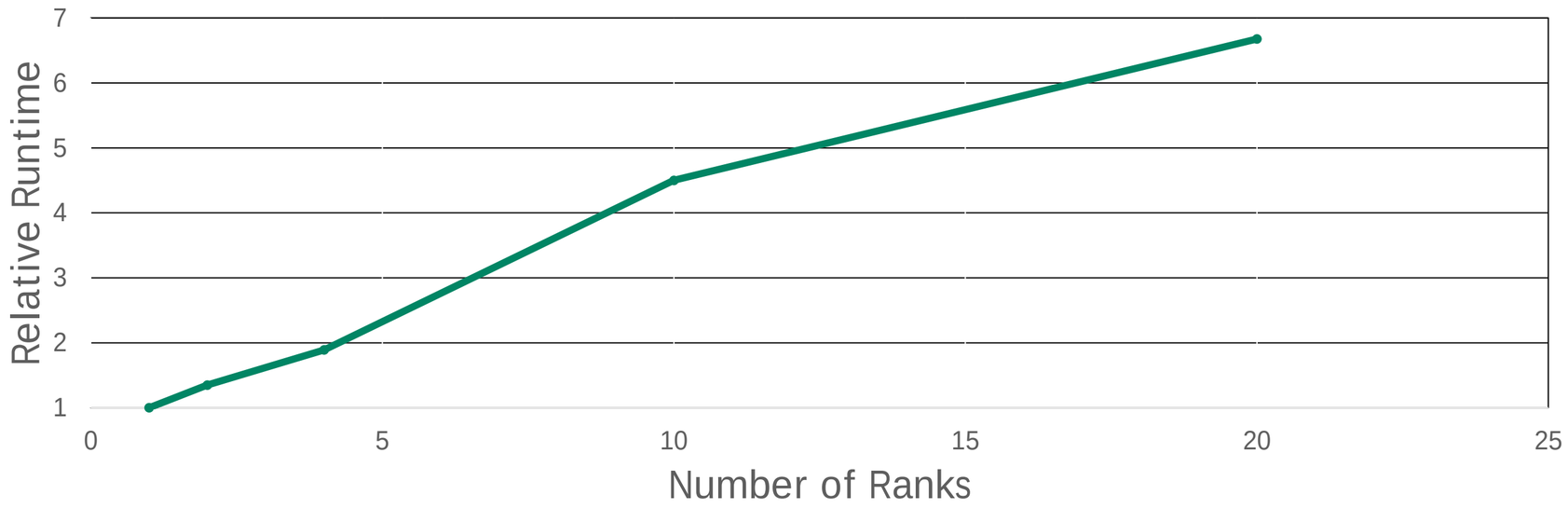
Whether or not there's a speedup depends substantially on precise timing



OVERSUBSCRIPTION WITH MPS

A much smaller case: N = 10^5

Splitting up work is a clear loser here (quickly get hit by launch latency)



OVERSUBSCRIPTION CONCLUSIONS REDUX

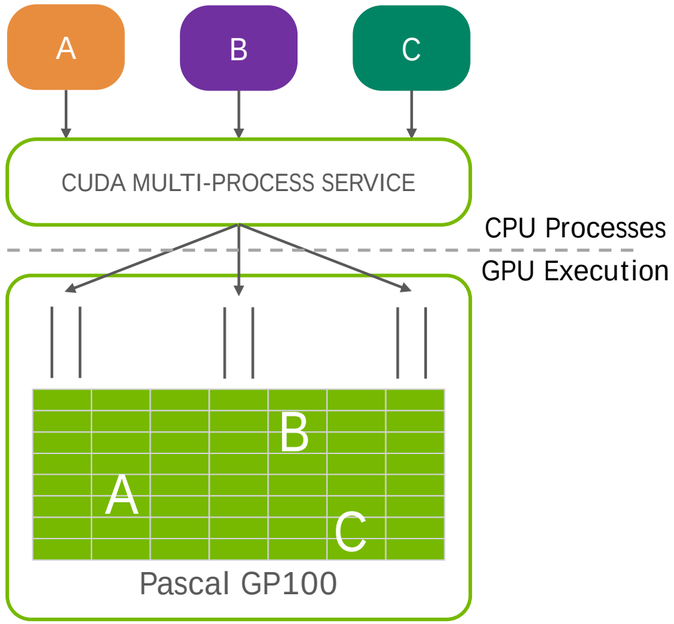
No free lunch theorem still applies: if GPU is fully utilized, cannot get faster answers

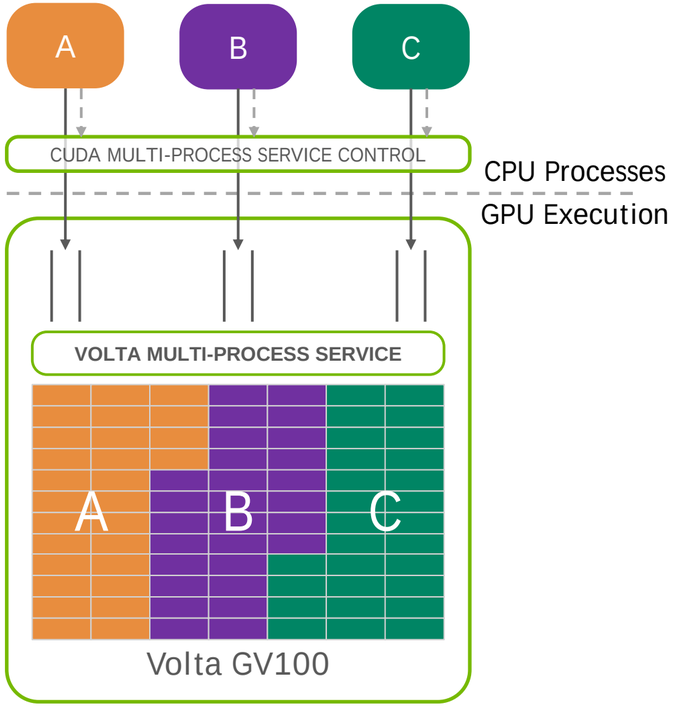
Strive to write your application *so that you don't need MPS*

If you are unable to write kernels that fully saturate the GPU, then consider oversubscription, and MPS is usually always worth turning on for that case

Profile your code to understand why MPS did or did not help

COMPARISON OF PRE- AND POST-VOLTA MPS





Software work submission

Limited isolation

16 clients per GPU

No provisioning

Faster, hardware-accelerated work submission

Hardware memory isolation

48 clients per GPU

Execution resource provisioning

KEY DIFFERENCES BETWEEN PRE- AND POST-VOLTA MPS

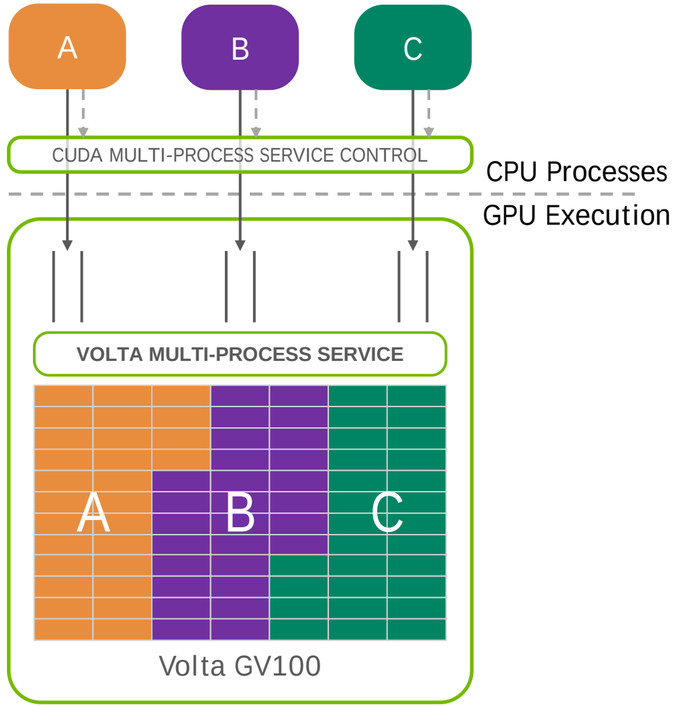
More MPS clients per GPU: 48 instead of 16

Less overhead: Volta MPS clients submit work directly to the GPU without passing through the MPS server.

More security: Each Volta MPS client owns its own GPU address space instead of sharing GPU address space with all other MPS clients.

More control: Volta MPS supports limited execution resource provisioning for Quality of Service (QoS). -> `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`

Independent work submission: Each process has private work queues, allowing concurrent submission without contending over locks.



USING MPS

No application modifications necessary

Not limited to MPI applications

MPS control daemon spawns MPS server upon CUDA application startup

Profiling tools are MPS-aware; cuda-gdb doesn't support attaching but you can dump core files

```
# Manually

nvidia-smi -c EXCLUSIVE_PROCESS

nvidia-cuda-mps-control –d

# On Summit

bsub –alloc_flags gpumps
```

Compute modes

- PROHIBITED (cannot set device)
- EXCLUSIVE_PROCESS (single shared device)
- DEFAULT (per-process device)

On shared systems, recommended to use EXCLUSIVE_PROCESS mode to ensure that only a single MPS server is using the GPU

MPS CONTROL: ENVIRONMENT VARIABLES

These are set per-process; can also manage MPS system-wide via control daemon

CUDA_VISIBLE_DEVICES

Sets devices which an application can see. When set on MPS daemon, limits visible GPUs for all clients.

CUDA_MPS_PIPE_DIRECTORY

Directory where MPS control daemon pipes are created. Clients & daemon must set to same value. Default is `/var/log/nvidia-mps`.

CUDA_MPS_LOG_DIRECTORY

Directory where MPS control daemon log is created. Default is `/tmp/nvidia-mps`.

CUDA_DEVICE_MAX_CONNECTIONS

Sets number of hardware work queues that CUDA streams map to. MPS clients all share the same pool, so if set in an MPS-attached process sets this it may limit the max number of MPS processes.

CUDA_MPS_ACTIVE_THREAD_PERCENTAGE

Controls what fraction of GPU may be used by a process - see next slides.

EXECUTION RESOURCE PROVISIONING WITH MPS

Using MPS, applications can assign fractions of a GPU to each process

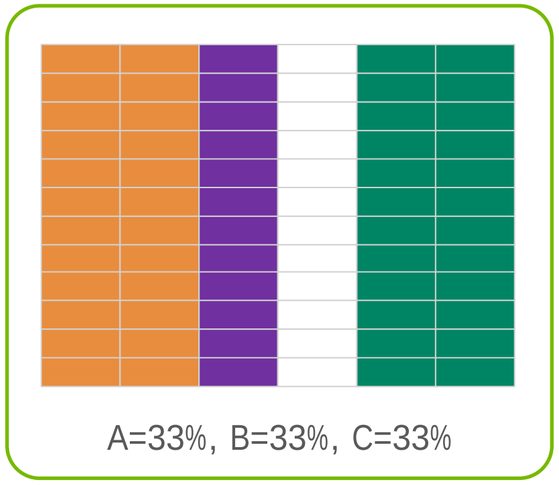
$ export CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=*percentage*

- Environment variable: configures maximum fraction of a GPU available to an MPS-attached process
- Guarantees a process will use at most *percentage* execution resources (SMs)
- Over-provisioning is permitted: sum across all MPS processes may exceed 100%
- Provisions only execution resources (SMs) - does not provision memory bandwidth or capacity
- Before CUDA 11.2, all processes be set to the same percentage
- Since CUDA 11.2, percentage may be different for each process

Full details at: https://docs.nvidia.com/deploy/mps/index.html#topic_5_2_5

GPU PROVISIONING WITH MPS

Using MPS, applications can assign fractions of a GPU to each process

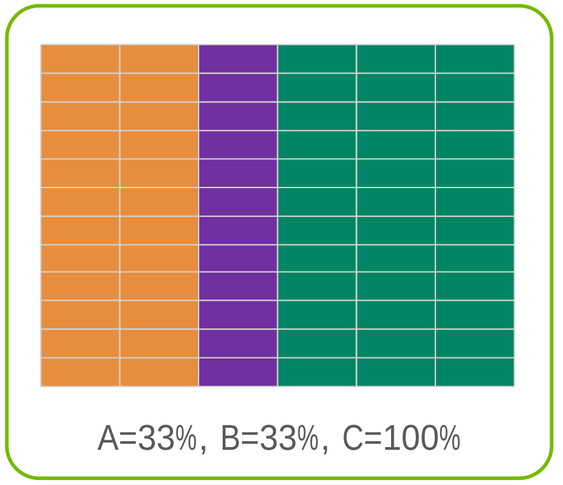


A=33%, B=33%, C=33%

Fractional Provisioning

Process C could use more, but is limited to just 33% of execution resources

Process B is guaranteed space if needed



A=33%, B=33%, C=100%

Using Oversubscription

Process B is not using all of its allocation

Process C may grow to fill available space

Additional B work may have to wait for resources

A B C ← 3 concurrent MPS processes

THINGS TO WATCH OUT FOR

See https://docs.nvidia.com/deploy/mps/index.html for more details

Memory Footprint

To provide a per-thread stack, CUDA reserves 1kB of GPU memory per thread

This is (2048 threads per SM x 1kB per thread) = 2 MB per SM used, or 164 MB per client for V100 (221 MB for A100)

CUDA_MPS_ACTIVE_THREAD_PERCENTAGE reduces max SM usage, and so reduces memory footprint

Each MPS process also uploads a new copy of the executable code, which adds to the memory footprint

Work Queue Sharing

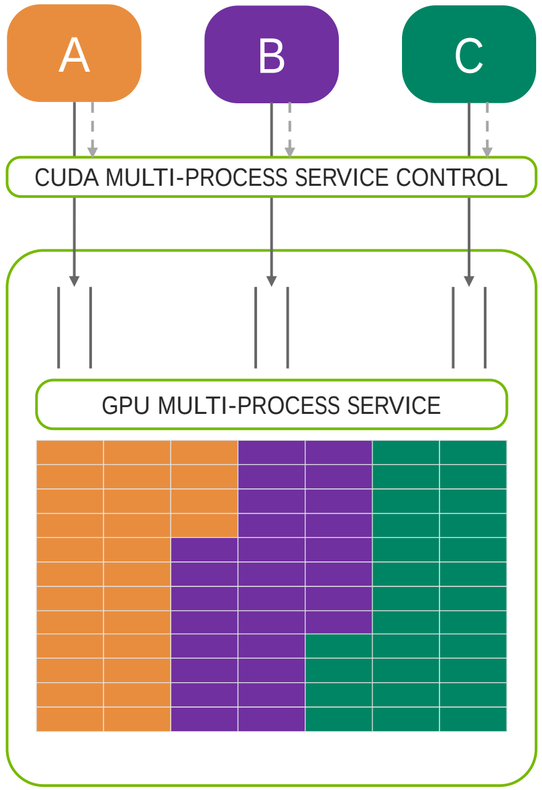
CUDA maps streams onto CUDA_DEVICE_MAX_CONNECTIONS hardware work queues

Queues are normally per-process, but MPS allows 96 hardware queues to be shared among up to 48 clients

MPS automatically reduces connections-per-client unless environment variable is set

If CUDA_DEVICE_MAX_CONNECTIONS is set (e.g. to enable more concurrency within a process), this can reduce the maximum number of concurrent clients

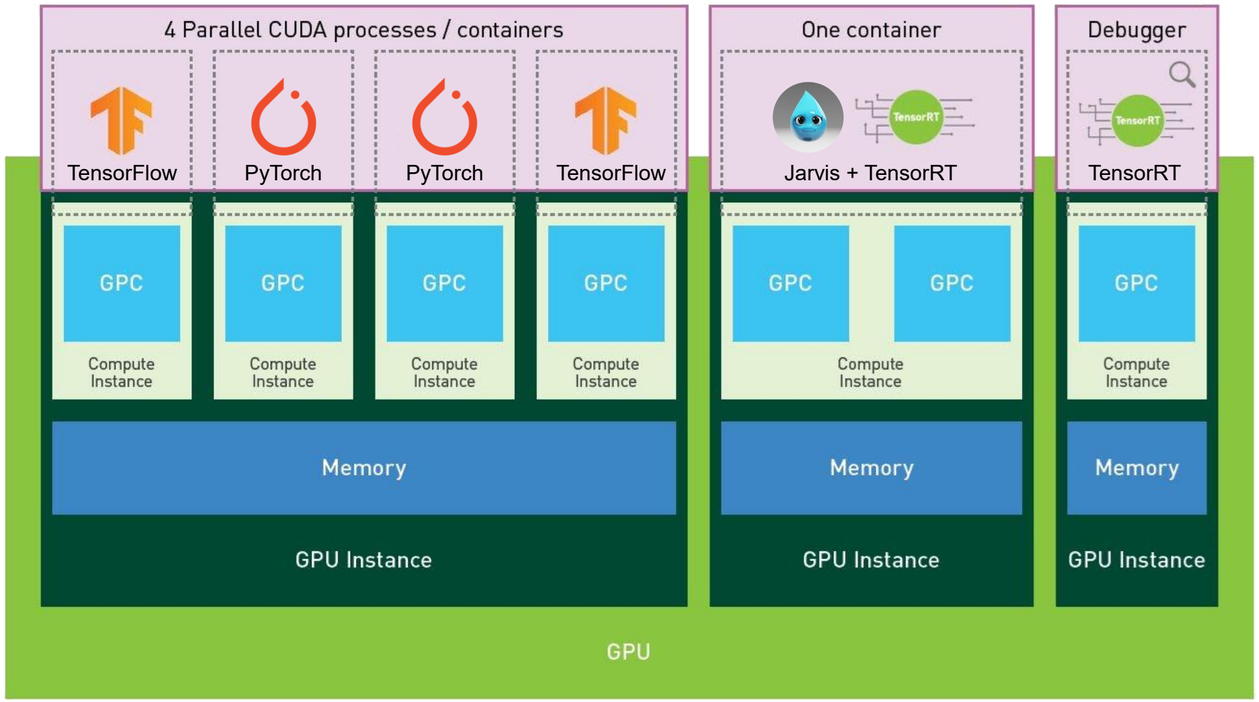
MPS LOGICAL VS. MIG PHYSICAL PARTITIONING



Multi-Process Service

Dynamic contention for GPU resources

Single tenant



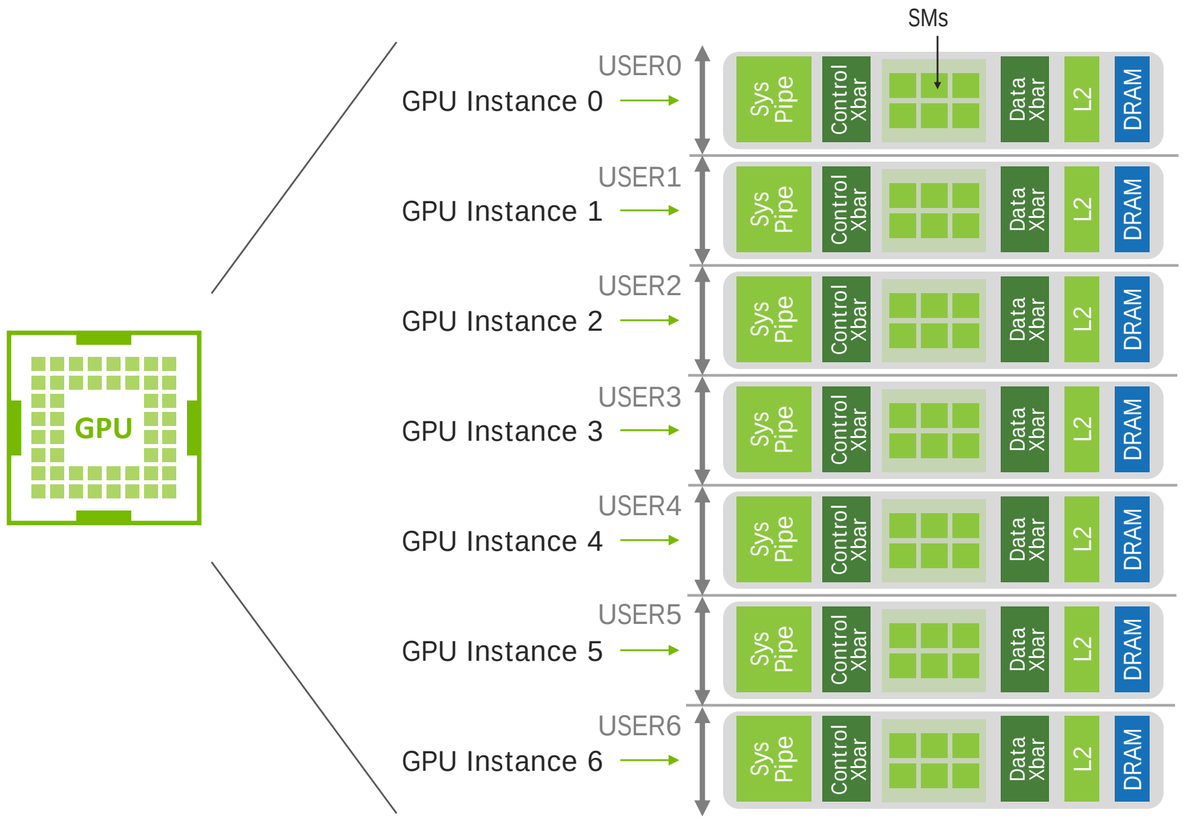
Multi-Instance GPU

Hierarchy of instances with guaranteed resource allocation

Multiple tenants

MULTI-INSTANCE GPU (MIG)

Divide a Single A100 GPU Into Multiple *Instances*, Each With Isolated Paths Through the Entire Memory System



Up To 7 GPU Instances In a Single A100

Full software stack enabled on each instance, with dedicated SM, memory, L2 cache & bandwidth

Simultaneous Workload Execution With Guaranteed Quality Of Service

All MIG instances run in parallel with predictable throughput & latency, fault & error isolation

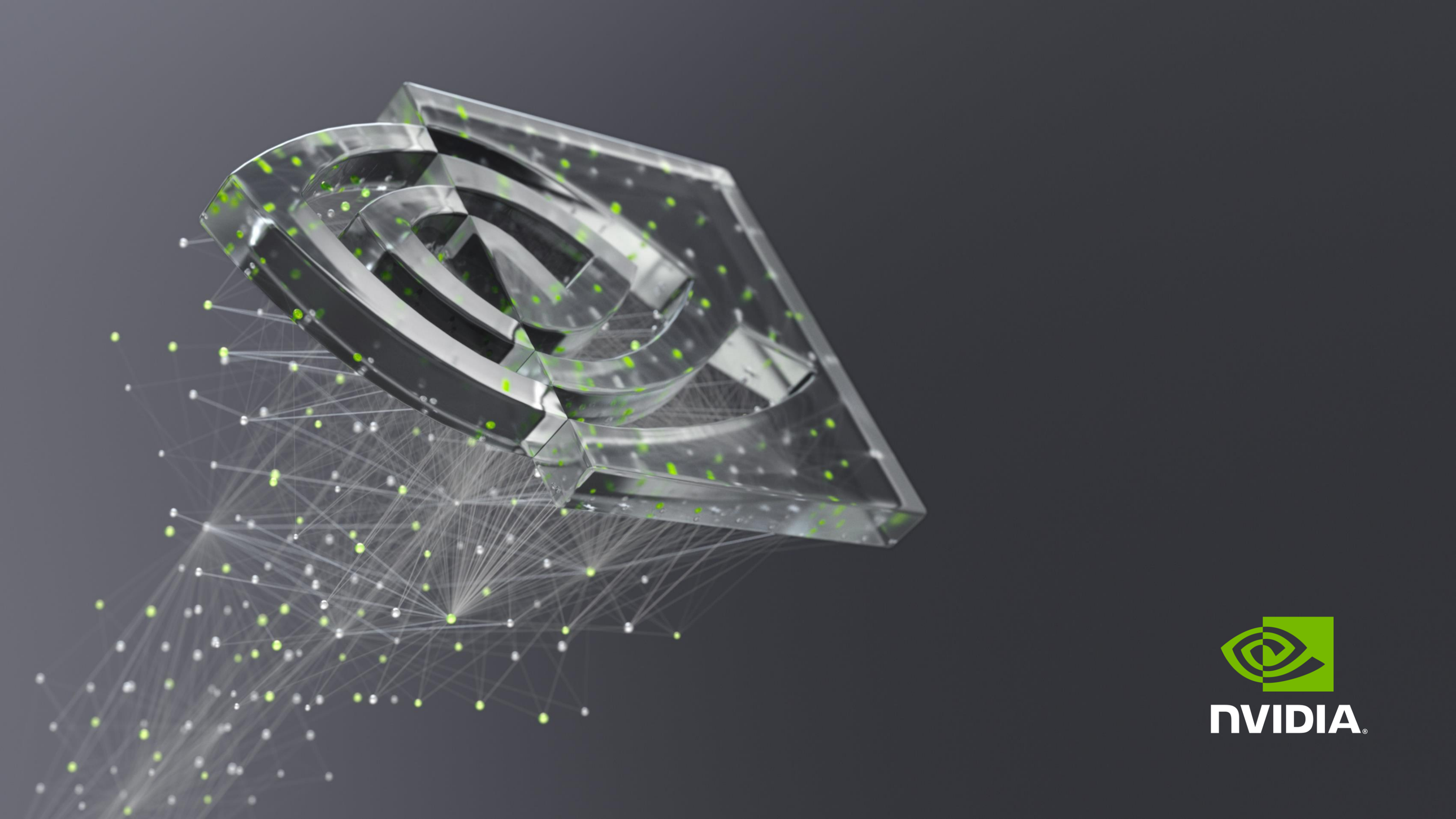
Diverse Deployment Environments

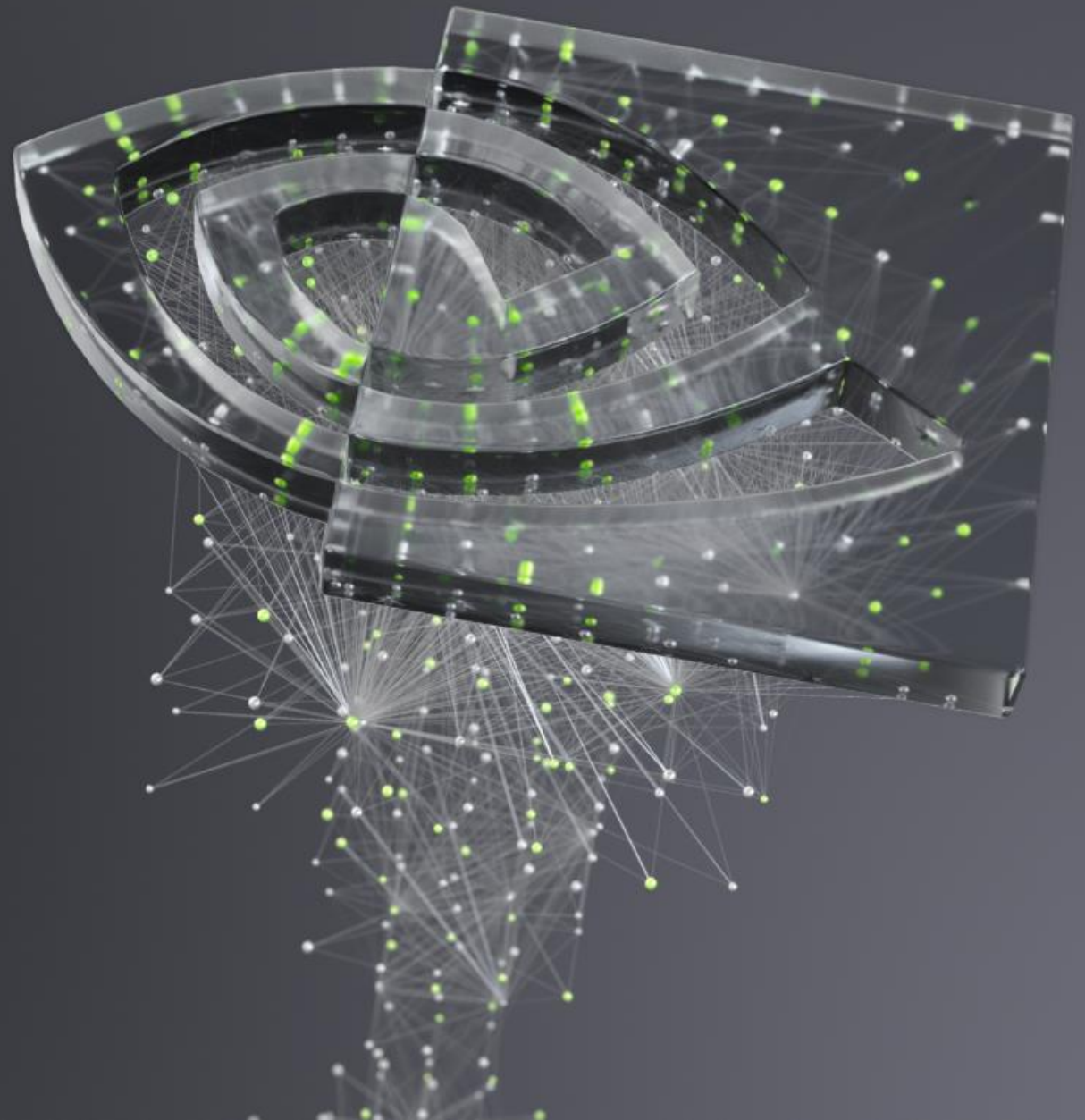
Supported with Bare metal, Docker, Kubernetes Pod, Virtualized Environments

CUDA CONCURRENCY MECHANISMS

	Streams	MPS	MIG
Partition Type	Single process	Logical	Physical
Max Partitions	Unlimited	48	7
Performance Isolation	No	By percentage	Yes
Memory Protection	No	Yes	Yes
Memory Bandwidth QoS	No	No	Yes
Error Isolation	No	No	Yes
Cross-Partition Interop	Always	IPC	Limited IPC
Reconfigure	Dynamic	Process launch	When idle

MPS: Multi-Process Service
MIG: Multi-Instance GPU


NVIDIA



CUDA DEBUGGING

Bob Crovella, 9/14/2021

AGENDA

CUDA Error Management

compute-sanitizer

cuda-gdb

Further Study

Homework

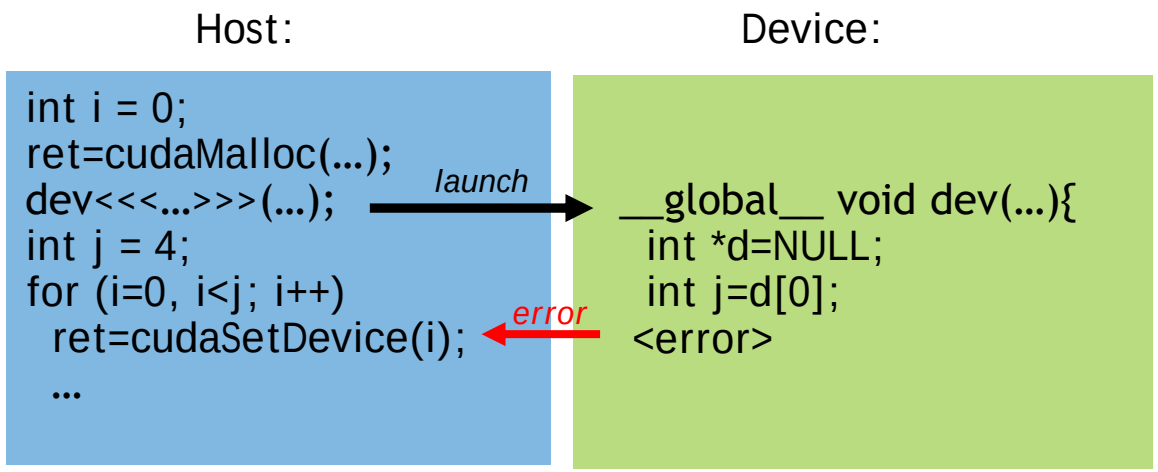
ERROR MANAGEMENT

BASIC CUDA ERROR CHECKING

- All CUDA runtime API calls return an error code.
  - CUDA runtime API: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html
  - Example: cudaError_t cudaSetDevice (int device)
  - cudaError_t is an enum type, with all possible error codes, examples:
    - cudaSuccess (no error)
    - cudaErrorMemoryAllocation (out of memory error)
- cudaGetErrorString(cudaError_t err) converts an error code to human-readable string
- Best practice is to always check these codes and handle appropriately. Just do it!
- The usual kernel launch syntax (kernel_name**<<<…>>>(…)**) is not a CUDA runtime API call and does not return an error code per-se

ASYNCHRONOUS ERRORS

- CUDA kernel launches are *asynchronous*
  - The kernel may not begin executing right away
  - The host thread that launches the kernel continues, without waiting for the kernel to complete
- It is possible for a CUDA error to be detected during kernel execution
- That error will be signalled at the *next* CUDA runtime API call, *after* the error is detected



KERNEL ERROR CHECKING

- CUDA kernel launches can produce two types of errors:
  - Synchronous: detectable right at launch
  - Asynchronous: occurs during device code execution
- Detect Synchronous errors right away with cudaGetLastError() or cudaPeekAtLastError()
- Asynchronous error checking involves tradeoffs
  - Can force immediate checking with a synchronizing call like cudaDeviceSynchronize() but this breaks asynchrony/concurrency structure
  - Optionally use a debug macro
  - Optionally set CUDA_LAUNCH_BLOCKING environment variable to 1

Kernel error checking example:

```
dev<<<...>>>(...);
ret = cudaGetLastError();
if (debug) ret = cudaDeviceSynchronize();
```

STICKY VS. NON-STICKY ERRORS

- A non-sticky error is recoverable
  - Example: ret = cudaMalloc(100000000000000000000000000000); (out of memory error)
  - Such errors do not “corrupt the CUDA context”
  - Subsequent CUDA runtime API calls behave normally
- A sticky error is not recoverable
  - A sticky error is usually (only) resulting from a kernel code execution error
  - Examples: kernel time-out, illegal instruction, misaligned address, invalid address
  - CUDA runtime API is no longer usable in that process
  - All subsequent CUDA runtime API calls will return the same error
  - Only “recovery” process is to terminate the owning host process (i.e. end the application).
  - A multi-process application can be designed to allow recovery: https://stackoverflow.com/questions/56329377

EXAMPLES

- shared_mem_size=32768;
- k<<<1024, 1024, shared_mem_size*sizeof(double), stream>>>(…);
- cudaGetLastError() gets the last error *and clears it if it is not sticky*
- cudaPeekAtLastError() gets last error but does not clear it
- cudaMemcpy(dptr, hptr, size, cudaMemcpyDeviceToHost);
- ret = cudaMemcpy(dptr2, hptr2, size2, cudaMemcpyHostToDevice);

EXAMPLES

- Macro example - macro instead of function

```
#include <stdio.h>

#define cudaCheckErrors(msg) \
    do { \
        cudaError_t __err = cudaGetLastError(); \
        if (__err != cudaSuccess) { \
            fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n", \
                msg, cudaGetErrorString(__err), \
                __FILE__, __LINE__); \
            fprintf(stderr, "*** FAILED - ABORTING\n"); \
            exit(1); \
        } \
    } while (0)
```


COMPUTE-SANITIZER TOOL

COMPUTE-SANITIZER

- A functional correctness checking tool, installed with CUDA toolkit
- Provides “automatic” runtime API error checking – even if your code doesn’t handle errors
- Can work with various language bindings: CUDA Fortran, CUDA C++, CUDA Python, etc.
- Sub-tools:
  - memcheck (default): detects illegal code activity: illegal instructions, illegal memory access, misaligned access, etc.
  - racecheck: detects shared memory race conditions/hazards: RAW, WAW, WAR
  - initcheck: detects accesses to global memory which has not been initialized
  - synccheck: detects illegal use of synchronization primitives (e.g. __syncthreads())
- Many command line options to modify behavior:
  - https://docs.nvidia.com/cuda/sanitizer-docs/ComputeSanitizer/index.html#command-line-options

MEMCHECK SUB-TOOL

- The “default” tool – its recommended to run this tool first, before using other tools
- Basic usage: compute-sanitizer ./my_executable
- Kernel execution errors:
  - Invalid/out-of-bounds memory access
  - Invalid PC/Invalid instruction
  - Misaligned address for data load/store
- Provides error localization when your code is compiled with -lineinfo
  - This is useful for other tools also, e.g. source-level work in the profilers (nsight compute)
- Has a performance impact on speed of kernel execution
- Can also do leak checking for device-side memory allocation/free
- Error checking is “tighter” than ordinary runtime error checking

MEMCHECK EXAMPLE

Out-of-bounds detection

```
$ cat t1866.cu
__global__ void k(char *d){
  d[43] = 0;
}
int main(){
  char *d;
  cudaMalloc(&d, 42);
  k<<<1,1>>>(d);
  cudaDeviceSynchronize();
}
$ nvcc -o t1866 t1866.cu -lineinfo
$ ./t1866
$
```

```
$ compute-sanitizer ./t1866
========= COMPUTE-SANITIZER
========= Invalid __global__ write of size 1 bytes
=========     at 0x40 in
/home/user2/misc/t1866.cu:2:k(char*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7fe035a0002b is out of bounds
=========     Saved host backtrace …
=========     Host Frame:cuLaunchKernel
[0x7fe0685de728]
…
=========     Host Frame: [0x4034b1]
=========                 in /home/user2/misc/./t1866
=========
========= Program hit unspecified launch failure
(error 719) on CUDA API call to cudaDeviceSynchronize.
…
========= ERROR SUMMARY: 2 errors
```

RACECHECK SUB-TOOL

- CUDA specifies no order of execution among threads
- Shared memory is commonly used for inter-thread communication
- In this scenario, ordering of reads and writes often matters for correctness
- Basic usage: compute-sanitizer --tool racecheck ./my_executable
- Finds shared memory (only) race conditions:
  - WAW – two writes to the same location that don't have intervening synchronization
  - RAW – a write, followed by a read to a particular location, without intervening synchronization
  - WAR – a read, followed by a write, without intervening synchronization
- Detailed reporting is available:
  - https://docs.nvidia.com/cuda/sanitizer-docs/ComputeSanitizer/index.html#racecheck-report-modes

RACECHECK EXAMPLE

RAW hazard

```
$ cat t1866.cu
const int bs = 256;
__global__ void reverse(char *d){
  __shared__ char s[bs];
  s[threadIdx.x] = d[threadIdx.x];
  d[threadIdx.x] = s[bs-threadIdx.x-1];
}
int main(){
  char *d;
  cudaMalloc(&d, bs);
  reverse<<<1,bs>>>(d);
  cudaDeviceSynchronize();
}
$ nvcc -o t1866 t1866.cu -lineinfo
$ compute-sanitizer ./t1866
========= COMPUTE-SANITIZER
========= ERROR SUMMARY: 0 errors
$
```

```
$ compute-sanitizer --tool racecheck ./t1866
========= COMPUTE-SANITIZER
========= ERROR: Race reported between Write
access at 0x70 in
/home/user2/misc/t1866.cu:4:reverse(char*)
=========     and Read access at 0x80 in
/home/user2/misc/t1866.cu:5:reverse(char*) [256
hazards]
=========
========= RACECHECK SUMMARY: 1 hazard displayed (1
error, 0 warnings)
$
```

INITCHECK SUB-TOOL

Detects use of uninitialized device global memory

```
$ cat t1866.cu
const int bs = 1;
__global__ void k(char *in, char *out){
  out[threadIdx.x] = in[threadIdx.x];
}
int main(){
  char *d1, *d2;
  cudaMalloc(&d1, bs);
  cudaMalloc(&d2, bs);
  k<<<1,bs>>>(d1, d2);
  cudaDeviceSynchronize();
}
$ nvcc -o t1866 t1866.cu -lineinfo
$ compute-sanitizer ./t1866
========= COMPUTE-SANITIZER
========= ERROR SUMMARY: 0 errors
$
```

```
$ compute-sanitizer --tool initcheck ./t1866
========= COMPUTE-SANITIZER
========= Uninitialized __global__ memory read of
size 1 bytes
=========     at 0x50 in
/home/user2/misc/t1866.cu:3:k(char*,char*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7fc543a00000
=========     Saved host backtrace up to driver
entry point at kernel launch time
=========     Host Frame:cuLaunchKernel
[0x7fc57546a728]
=========                in /lib64/libcuda.so.1
…
========= ERROR SUMMARY: 1 error
$
```

SYNCCHECK SUB-TOOL

- Applies to usage of __syncthreads(), __syncwarp(), and CG equivalents (e.g. this_group.sync())
- Typical usage is for detection of illegal use of synchronization, where not all necessary threads can reach the sync point:
  - Threadblock level
  - Warp level
- In addition, the __syncwarp() intrinsic can take a mask parameter, which specifies expected threads
  - Detects invalid usage of the mask
- Basic usage: compute-sanitizer --tool synccheck ./my_executable
- Applicability is limited on cc 7.0 and beyond due to volta execution model relaxed requirements
- Example:
  - https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html#synccheck-demo-illegal-syncwarp

DEBUGGING WITH CUDA-GDB

CUDA-GDB

- Based on widely-used gdb debugging tool (part of gnu toolchain). (This is not a tutorial on gdb)
- “command-line” debugger, allows for typical operations like:
  - setting breakpoints (e.g. b)
  - single-stepping (e.g. s)
  - inspection of data (e.g. p)
  - And others
- cuda-gdb uses the same command syntax where possible, and provides certain command extensions
- Generally, you want to build a debug code to use with the debugger
- The focus here will be on debugging device code. Assumption is you already know how to debug host code
- Supports debug of both CUDA C++ and CUDA Fortran applications

BUILDING DEBUG CODE

- Fundamentally, the compile command line for nvcc should include:
  - -g – standard gnu switch for building a debug (host) code
  - -G – builds debug device code
- This makes the necessary symbol information available to the debugger so that you can do “source-level” debugging.
- The –G switch has a substantial impact on device code generation. Use it for debug purposes only.
  - **Don’t do performance analysis on device code built with the -**G switch
  - The –G switch will often make your code run slower
  - In rare cases, the –G switch may change the behaviour of your code
- Make sure your code is compiled for the correct target: e.g. -arch=sm_70

ADDITIONAL PREP SUGGESTIONS

- If possible, make sure your code completes the various sanitizer tool tests
- If possible, make sure your host code is “sane” e.g. does not seg fault
- If possible, make sure your kernels are actually being launched, e.g:
  - nsys profile --stats=true ./my_executable (and check e.g. “CUDA Kernel Statistics”

CUDA SPECIFIC COMMANDS

- set cuda … <used to set general options and advanced settings>
  - launch_blocking (on/off) <make launches pause the host thread>
  - break_on_launch (option) <break on every new kernel launch>
- info cuda … <get general information on system configuration>
  - devices, sms, warps, lanes, kernels, blocks, threads, …
- cuda … <used to inspect or set current focus>
  - (cuda-gdb) cuda device sm warp lane block thread <display current focus coordinates>
  - block (0,0,0), thread (0,0,0), device 0, sm 0, warp 0, lane 0
  - (cuda-gdb) cuda thread (15) <change coordinate(s)>

DEMO

ADDITIONAL NOTES, TIPS, TRICKS

- synccheck tool may have limited usefulness due to Volta execution model – relaxed sync requirements
- CUDA Fortran debugging “print” commands not working correctly – expected to be fixed in a future tool chain
- Cannot inspect device memory (e.g. with “print”) unless stopped at a breakpoint in device code
- compute-sanitizer host backtrace will be improved in the future
- How to “look up” an error code (e.g. 719), two ways:
  - Search in …/cuda/include/driver_types.h
  - Docs: runtime API section 6.36, Data types

FURTHER STUDY

- CUDA error checking:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#error-checking
  - https://stackoverflow.com/questions/14038589/what-is-the-canonical-way-to-check-for-errors-using-the-cuda-runtime-api
  - CUDA context: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#context
- compute-sanitizer:
  - https://docs.nvidia.com/cuda/sanitizer-docs/ComputeSanitizer/index.html
- cuda-gdb:
  - https://docs.nvidia.com/cuda/cuda-gdb/index.html
- Simple gdb tutorial:
  - https://www.cs.cmu.edu/~gilpin/tutorial/

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw12/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming

BACKUP: BASIC GDB SYNTAX

BASIC GDB

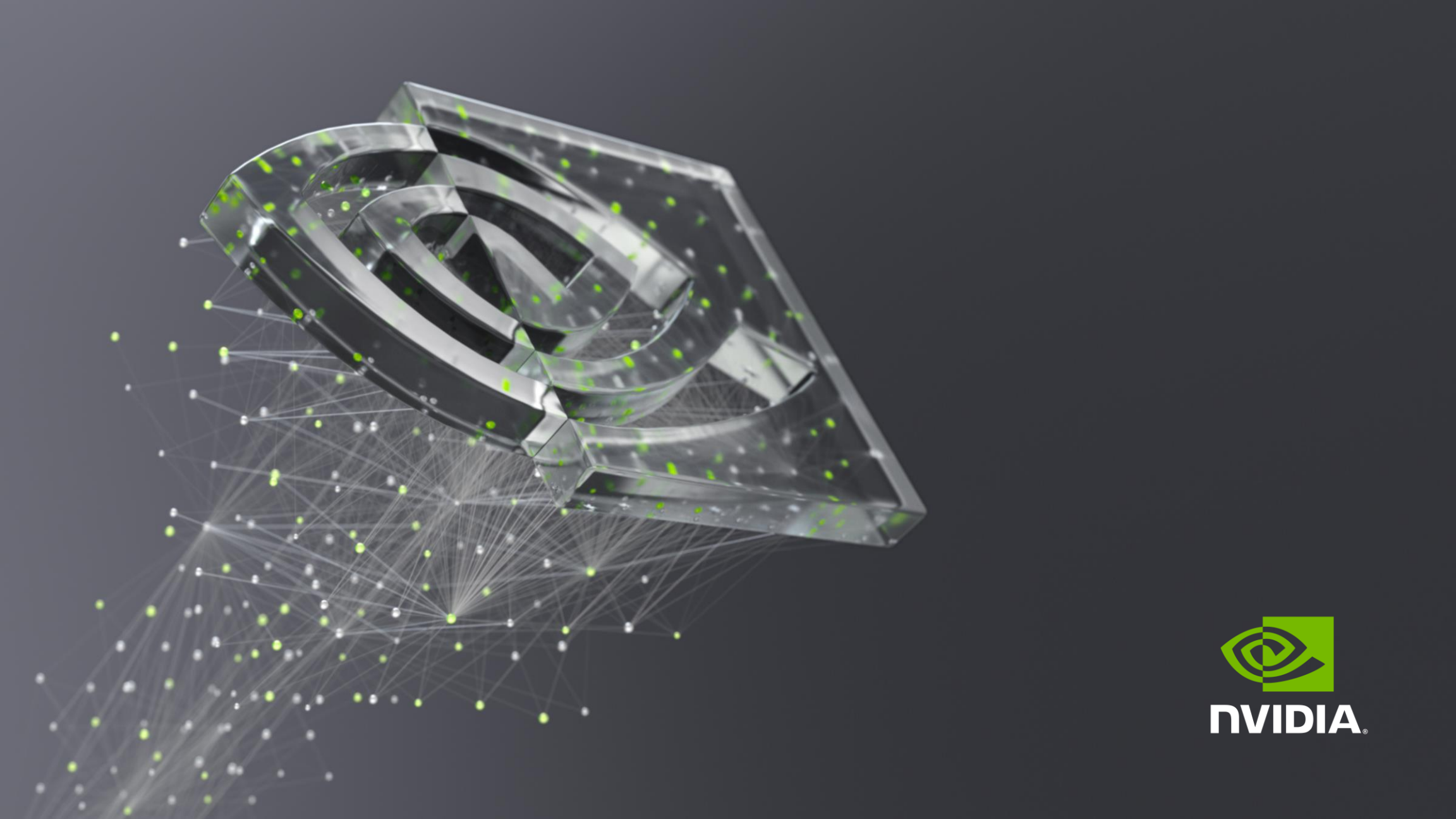
Getting started, setting a breakpoint, running, single-step, continuing

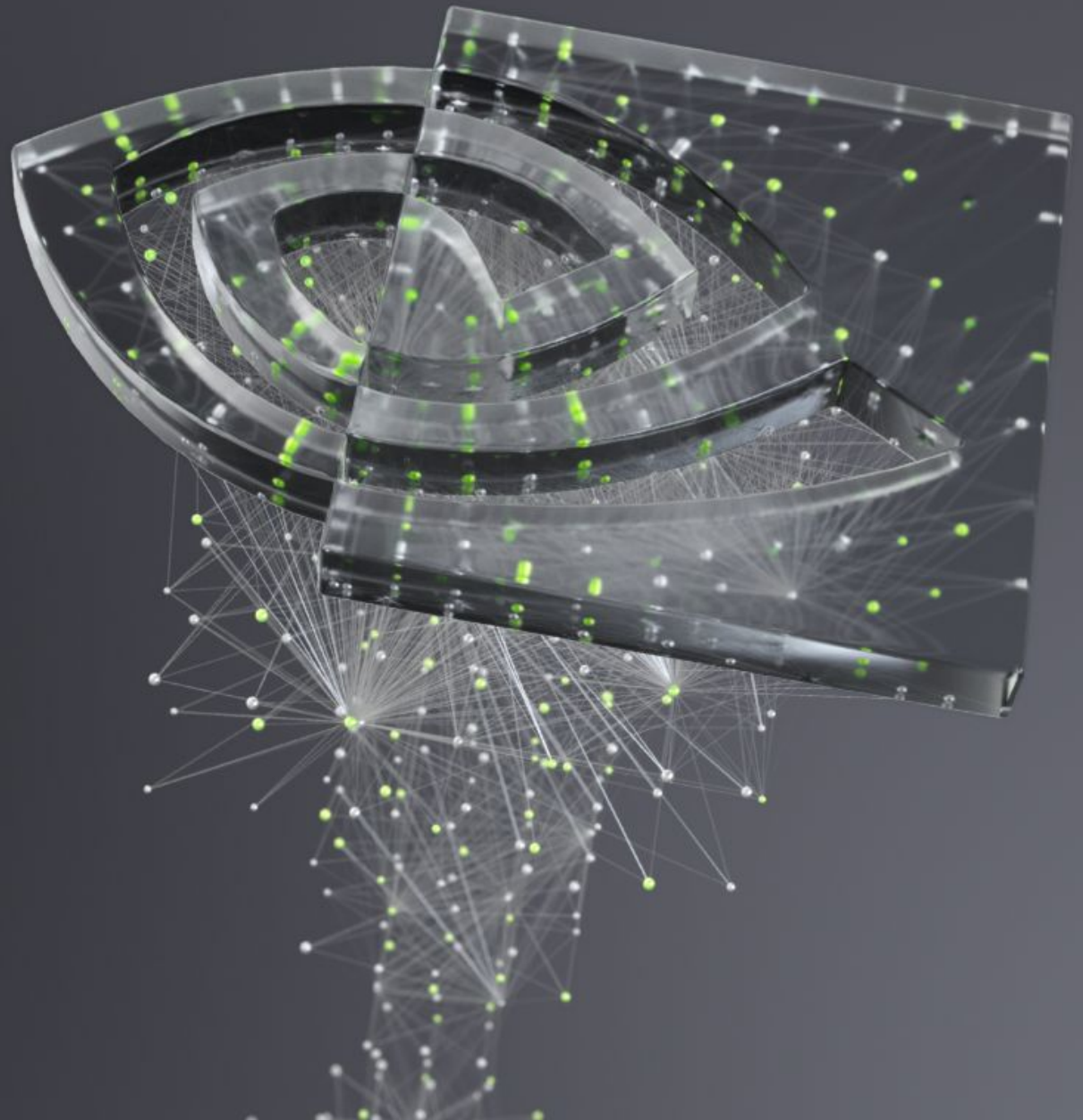
- Compile your code with -g (host debug) and -G (device debug)
- gdb ./my_executable
- Set a breakpoint: b command
  - if only one file: (gdb) b <line_number>
  - If multiple source files: (gdb) b <file_name:line_number>
- Run-from-start: r command
- Single step: s command ("step into")
- Step next: n command ("step over")
- Continue : c command

BASIC GDB

Inspecting data, clearing breakpoints, conditional breakpoints

- Print data: p command
  - symbolically: p s[0]
  - multiple values: p s[0]@8
- Removing breakpoints:
  - clear <file-name:line-number> (removes breakpoint based on location)
  - delete <breakpoint-number> (removes breakpoint based on id)
- Conditional breakpoints:
  - Set a breakpoint first
  - condition <breakpoint-id> <Boolean-test>
  - condition 1 i<32

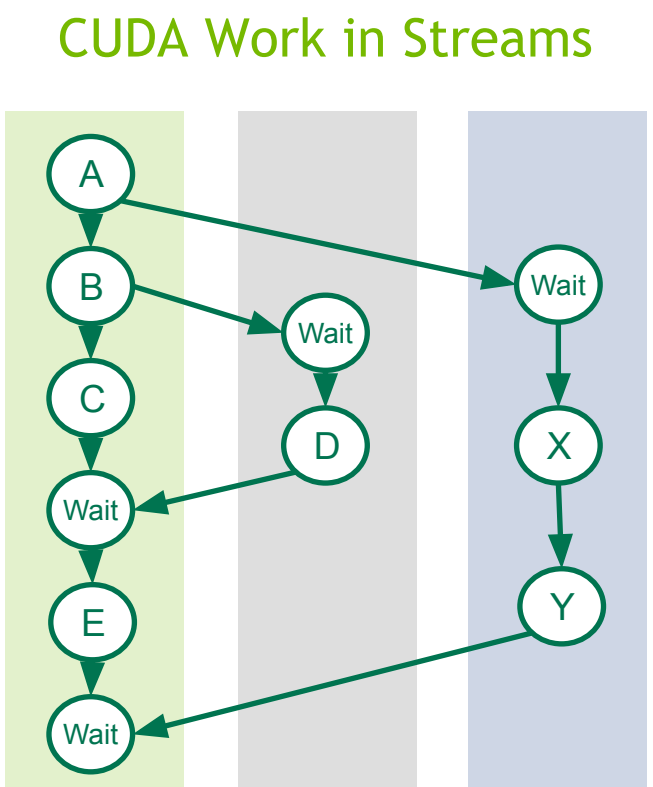

NVIDIA



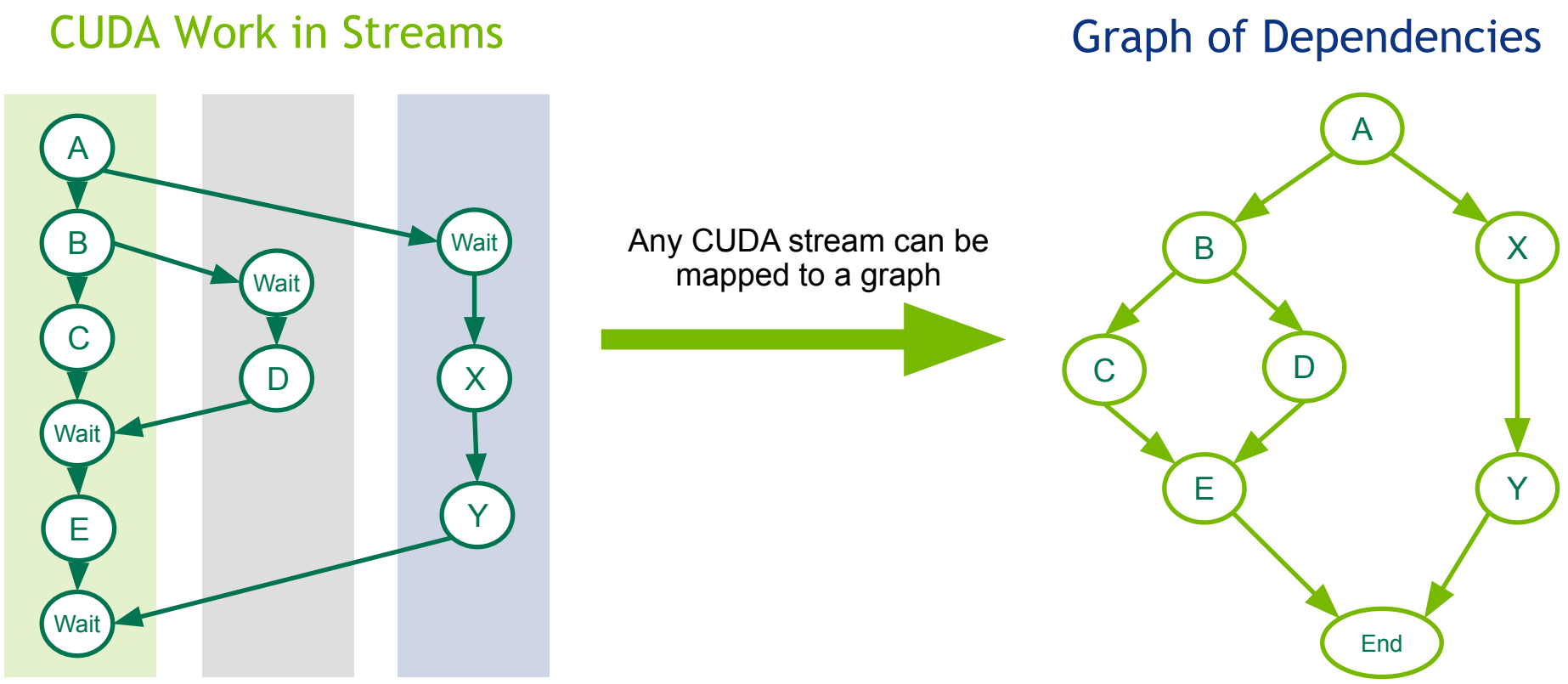
CUDA Graphs

NVIDIA Corporation

ALL CUDA WORK FORMS A GRAPH



ALL CUDA WORK FORMS A GRAPH



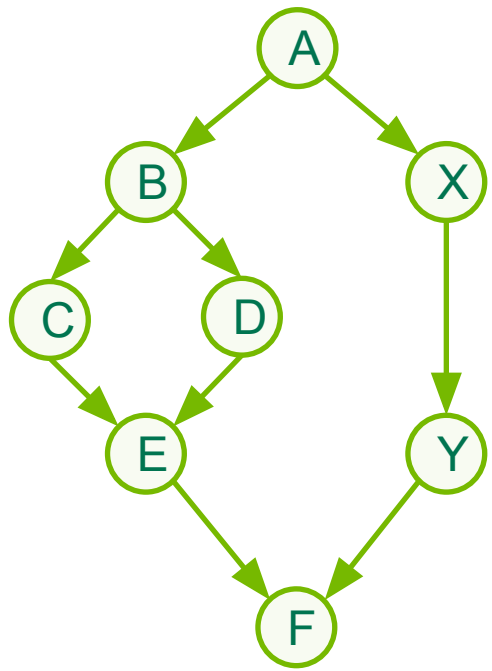
DEFINITION OF A CUDA GRAPH

A graph node is any **asynchronous** CUDA operation

Sequence of operations, connected by dependencies.

Operations are one of:

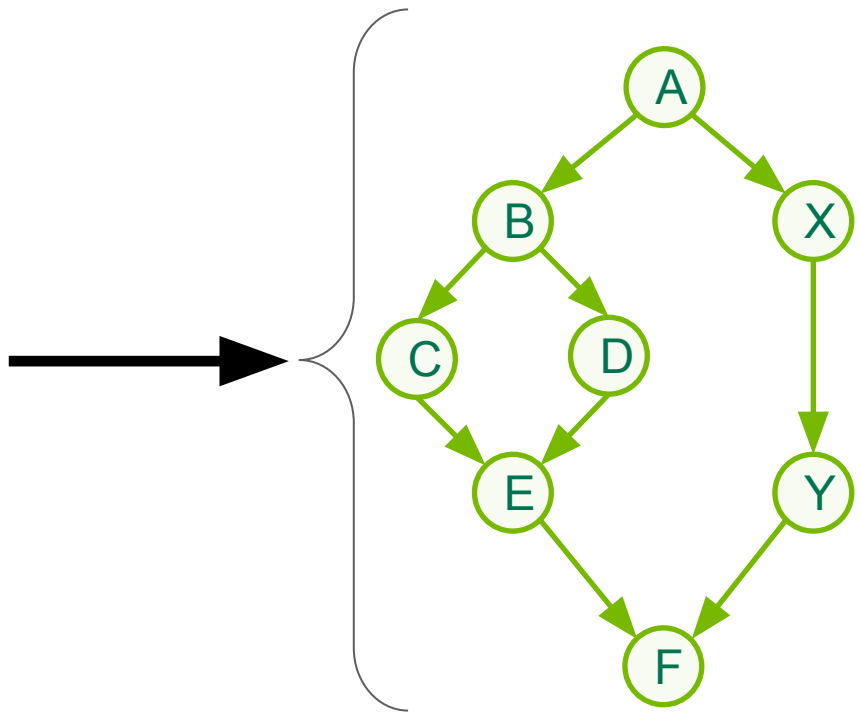
Kernel Launch	CUDA kernel running on GPU
CPU Function Call	Callback function on CPU
Memcopy/Memset	GPU data management
Memory Alloc/Free	Inline memory allocation
Sub-Graph	Graphs are hierarchical



NEW EXECUTION MECHANISM

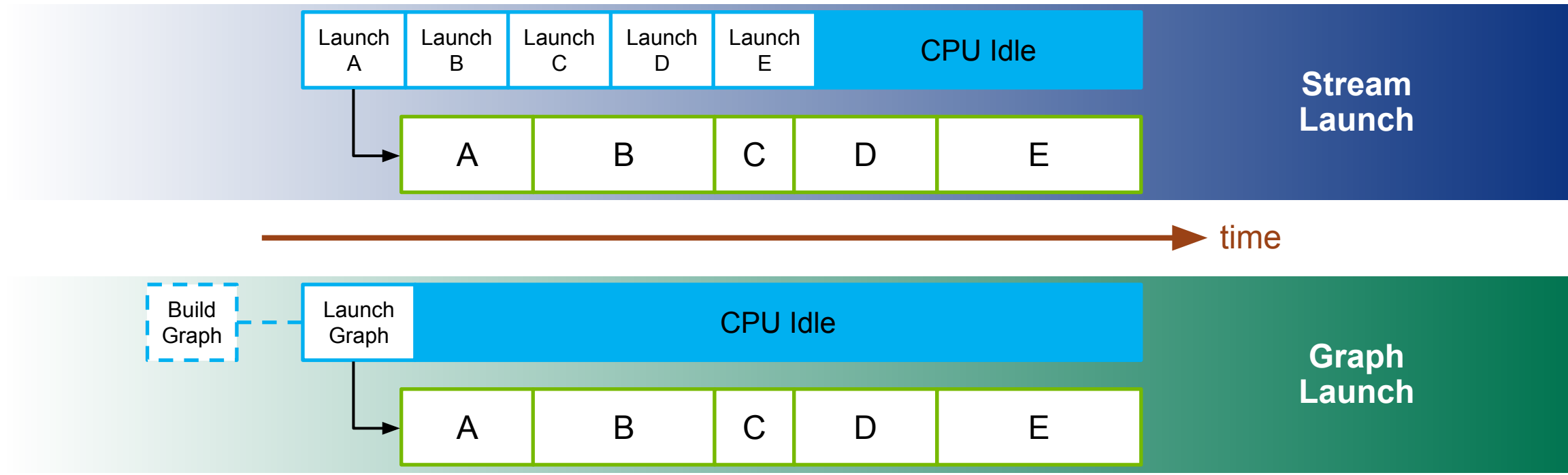
Graphs Can Be Generated Once Then Launched Repeatedly

```
for(int i=0; i<1000; i++) {
    launch_graph( G );
}
```



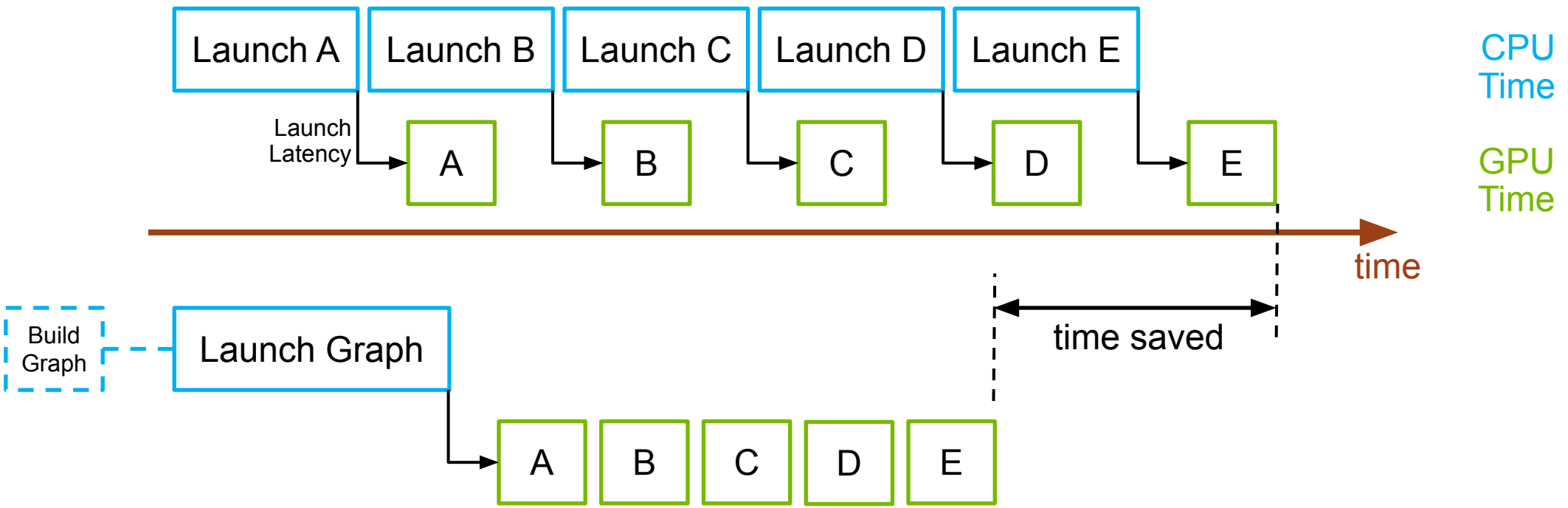
FREE UP CPU RESOURCES

Release CPU Time For Lower Power, or Running Other Work



LAUNCH OVERHEAD REDUCTION

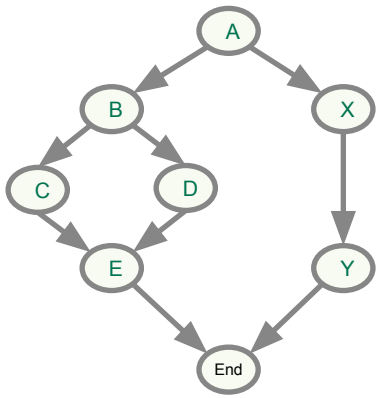
Graph launch submits all work at once, reducing CPU cost



When kernel runtime is short, execution time is dominated by CPU launch cost

THREE-STAGE EXECUTION MODEL

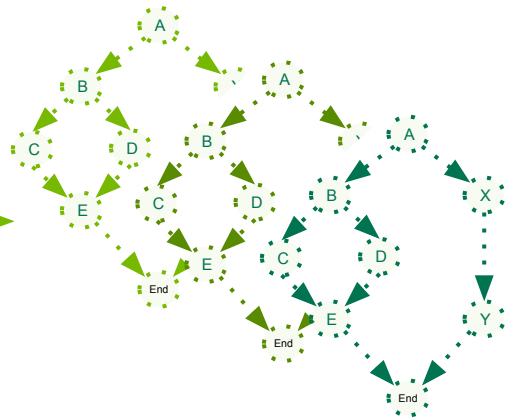
Define



Single Graph “Template”

Created in host code,
or loaded from disk,
or built up from libraries

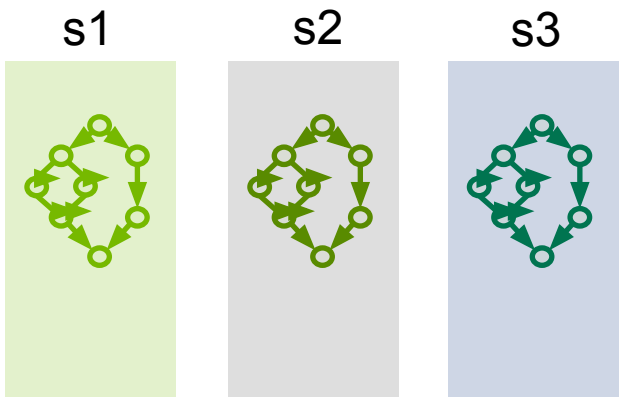
Instantiate



Multiple “Executable Graphs”

Snapshot of template
Sets up & initializes GPU
execution structures
(create once, run many times)

Execute

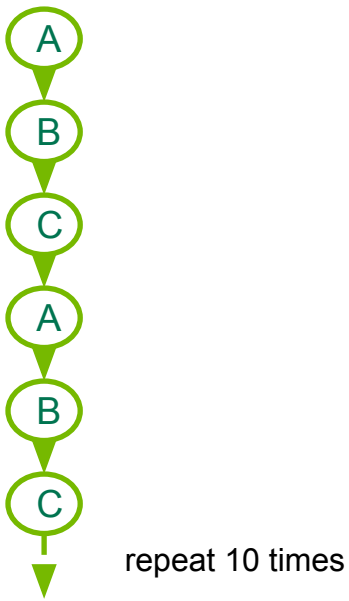


Executable Graphs
Running in CUDA Streams

Concurrency in graph
is not limited by stream
(see later)

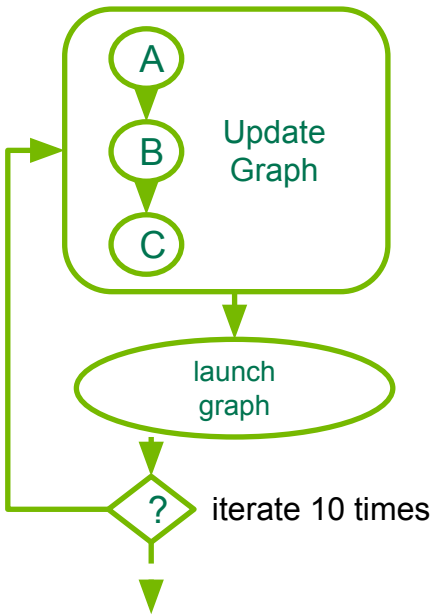
MODIFYING GRAPHS IN-PLACE

Stream Launch



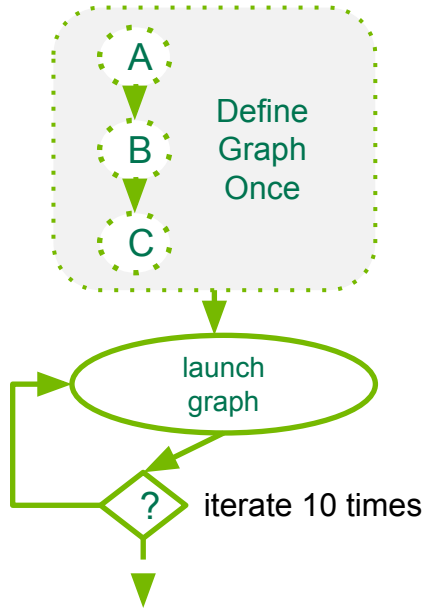
Parameters: **may** change
Topology: **may** change

Graph Update



Parameters: **may** change
Topology: **may not** change

Graph Re-Launch



Parameters: **may not** change
Topology: **may not** change

PROGRAMMING MODEL

ASYNCHRONOUS OPERATIONS ONLY

Typically Shows Up During Stream Capture

Stream Capture

- Very convenient way of creating a graph from existing library calls (see later slide)
- Records operations without actually launching a kernel
- Library must call an API to tell if kernels are being captured instead of launched

Problem if library calls *cudaStreamSynchronize()* or any other synchronous operation.

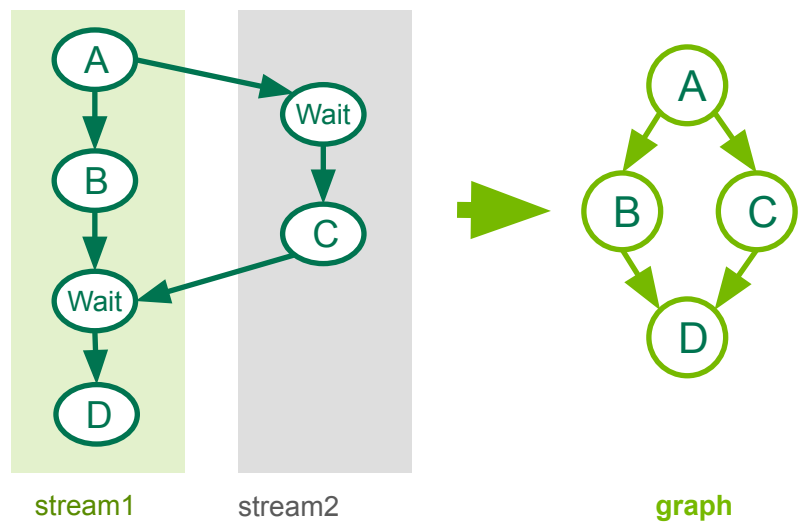
Capture is not launching anything so synchronize cannot wait for anything.

Capture operation fails.

CAPTURE CUDA STREAM WORK INTO A GRAPH

Construct a graph from normal CUDA stream syntax

```
// Start by initiating stream capture
cudaStreamBeginCapture(&stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

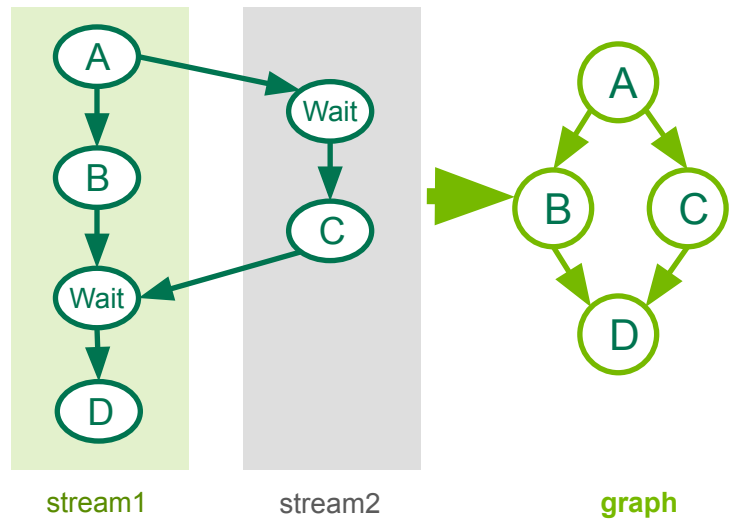


CAPTURE CUDA STREAM WORK INTO A GRAPH

Construct a graph from normal CUDA stream syntax

```
// Start by initiating stream capture
cudaStreamBeginCapture(&stream1);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
     cudaStreamWaitEvent(str
           eam2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
     cudaStreamWaitEvent(str
           eam1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

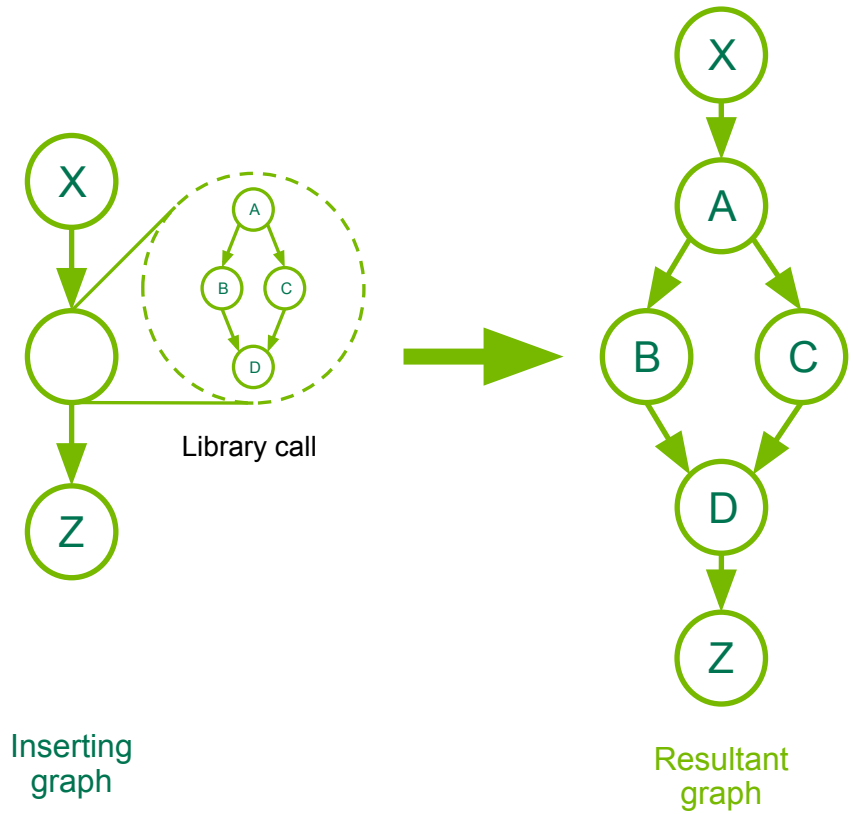
Capture follows inter-stream dependencies to create forks & joins



CAPTURE EXTERNAL WORK

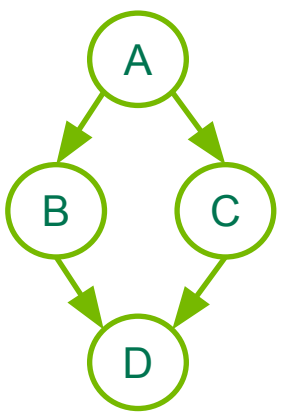
Stream Capture

```
// Start by initiating stream capture
cudaStreamBeginCapture(&stream);

// Captures my kernel launches and inside library calls
X<<< ..., stream >>>();
libraryCall(stream);        // Launches A, B, C, D
Z<<< ..., stream >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream, &graph);
```

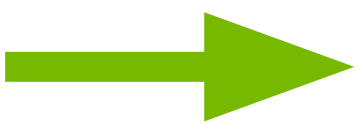


CREATE GRAPHS DIRECTLY

Map Graph-Based Workflows Directly Into CUDA



Graph from framework



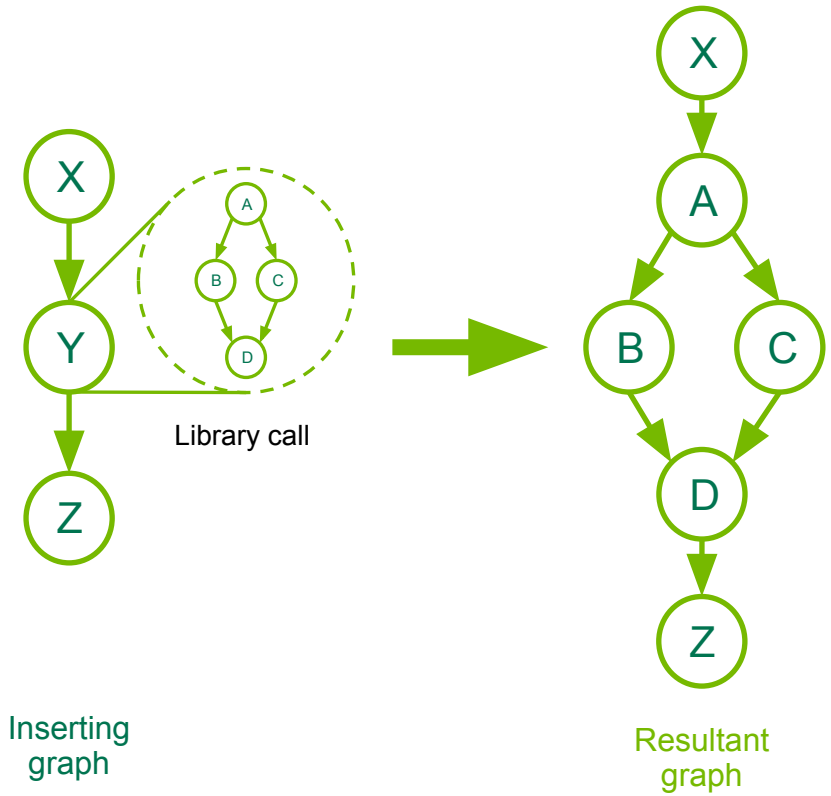
```
// Define graph of work + dependencies
cudaGraphCreate(&graph);

cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times
for(int i=0; i<100; i++)
      cudaGraphLaunch(instance, stream);
```

(Full list of API calls in the CUDA Docs)

COMBINING GRAPH & STREAM WORK

Capturing Streams Into An Existing Graph

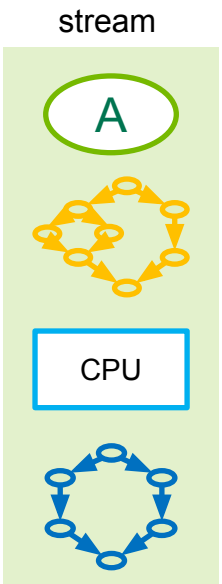
```
// Create root node of graph via explicit API
cudaGraphAddNode(main_graph, X, {}, ...);

// Capture the library call into a subgraph
cudaStreamBeginCapture(&stream);
libraryCall(stream);          // Launches A, B, C, D
cudaStreamEndCapture(stream, &library_graph);

// Insert the subgraph into main_graph as node "Y"
cudaGraphAddChildGraphNode(Y, main_graph, { X } ... libraryGrpah);

// Continue building main graph via explicit API
cudaGraphAddNode(main_graph, Z, { Y }, ...);
```



GRAPH EXECUTION SEMANTICS

Order Graph Work With Other Non-Graph CUDA Work

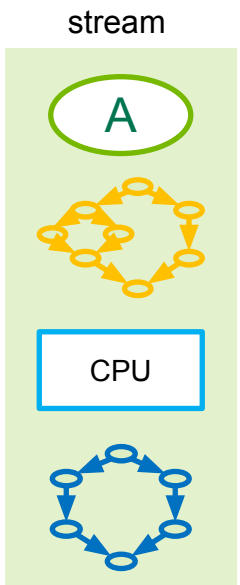
```
launchWork(cudaGraphExec_t i1, cudaGraphExec_t i2,
           CPU_Func cpu, cudaStream_t stream) {

    A <<< 256, 256, 0, stream >>>();        // Kernel launch
    cudaGraphLaunch(i1, stream);            // Graph launch
    cudaStreamAddCallback(stream, cpu);     // CPU callback
    cudaGraphLaunch(i2, stream);            // Graph launch

    cudaStreamSynchronize(stream);
}
```



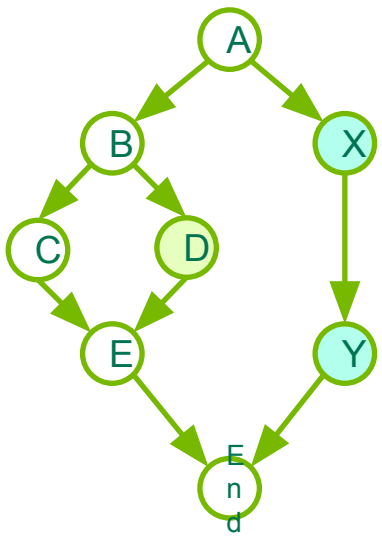
If you can put it in a CUDA stream, you can run it together with a graph

GRAPHS IGNORE STREAM SERIALIZATION RULES

Launch Stream Is Used Only For Ordering With Other Work



Branches in graph still execute concurrently even though graph is launched into a stream

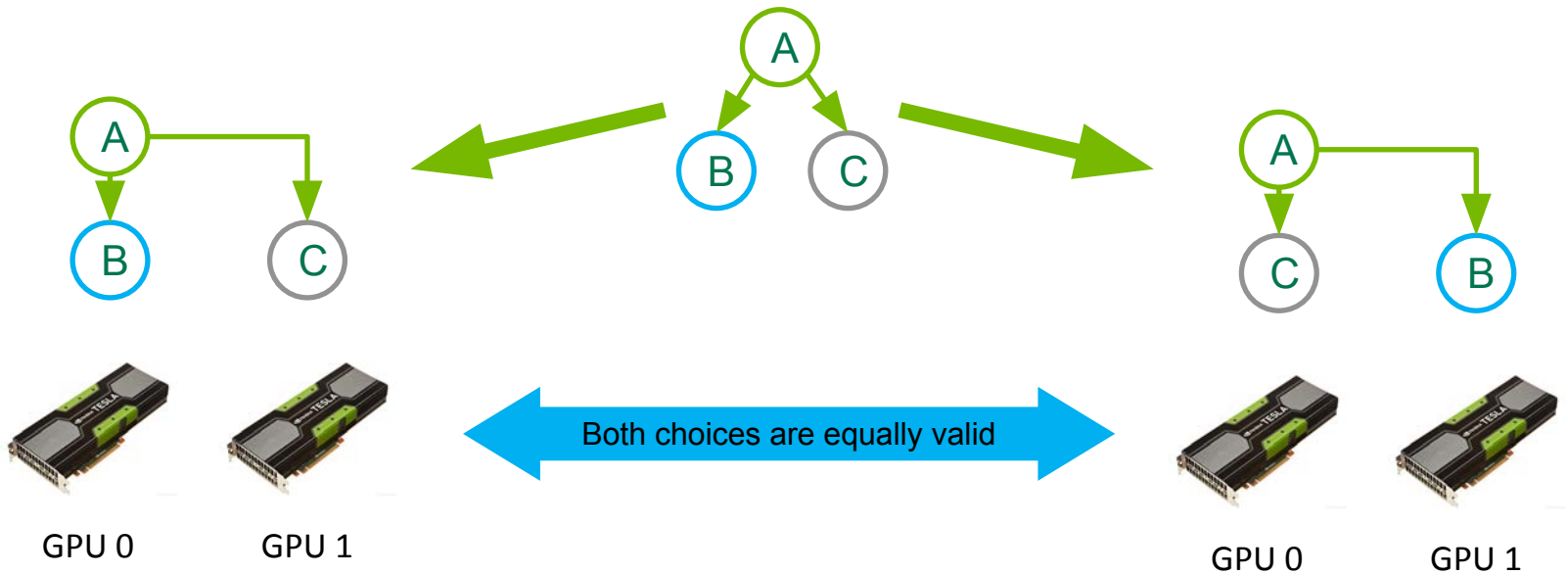


WHAT CAN YOU NOT DO WITH IT?

NO AUTOMATIC PLACEMENT

User Must Define Execution Location For Each Node

If fork in graph can run on 2 GPUs,
how do we pick what runs where?



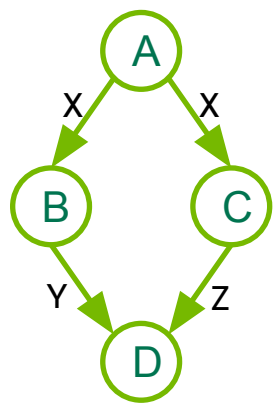
Best choice may depend on data locality – unknown at execution layer

EXECUTION DEPENDENCIES

CUDA Dependencies are Execution Dependencies

Data dependency graph definition

Task	Inputs	Outputs
A	none	X
B	X	Y
C	X	Z
D	Y, Z	none



Execution dependency graph definition

```
cudaGraphAddNode(graph, A, {}, ...);
cudaGraphAddNode(graph, B, { A }, ...);
cudaGraphAddNode(graph, C, { A }, ...);
cudaGraphAddNode(graph, D, { B, C }, ...);
```

All data dependencies can trivially be mapped to execution dependencies, but
Not all execution dependencies can be mapped to data dependencies

WHAT CAN YOU DO WITH IT?

RAPID RE-ISSUE OF WORK

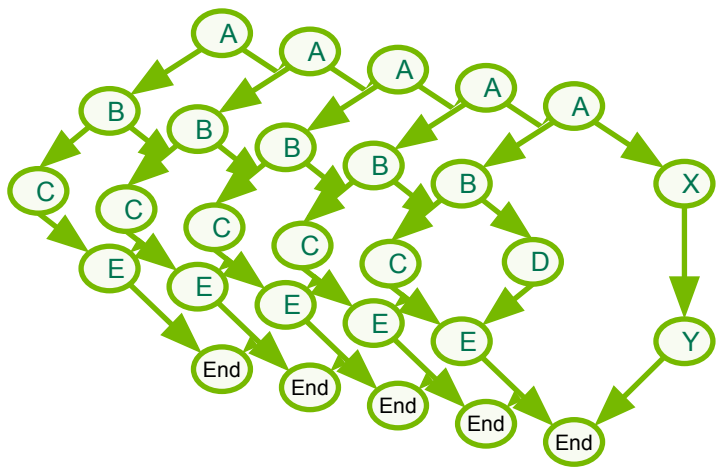
Graphs Can Be Generated Once And Executed Repeatedly

Cost of graph instantiation

≈

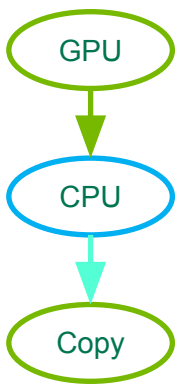
Cost of normal launch

```
for(int i=0; i<5; i++) {
    launch_graph( G );
}
```



HETEROGENEOUS NODE TYPES

Graph Nodes Include GPU Work, CPU Work and Data Movement



Heterogeneous Execution

Data management **may** be optimized transparently

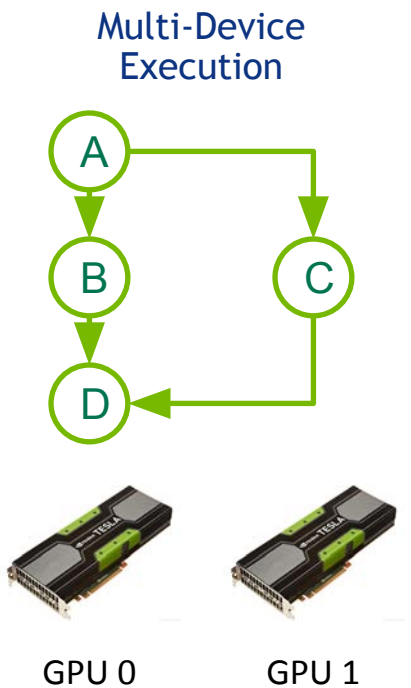
- Prefetching
- Read duplication
- Subdivision to finer granularity

Optimize for **bandwidth** and **latency** of memory access

Optimize for bandwidth of **interconnect** (PCI, QPI, NVLink)

CROSS-DEVICE DEPENDENCIES

CUDA Can Sync Multiple GPUs



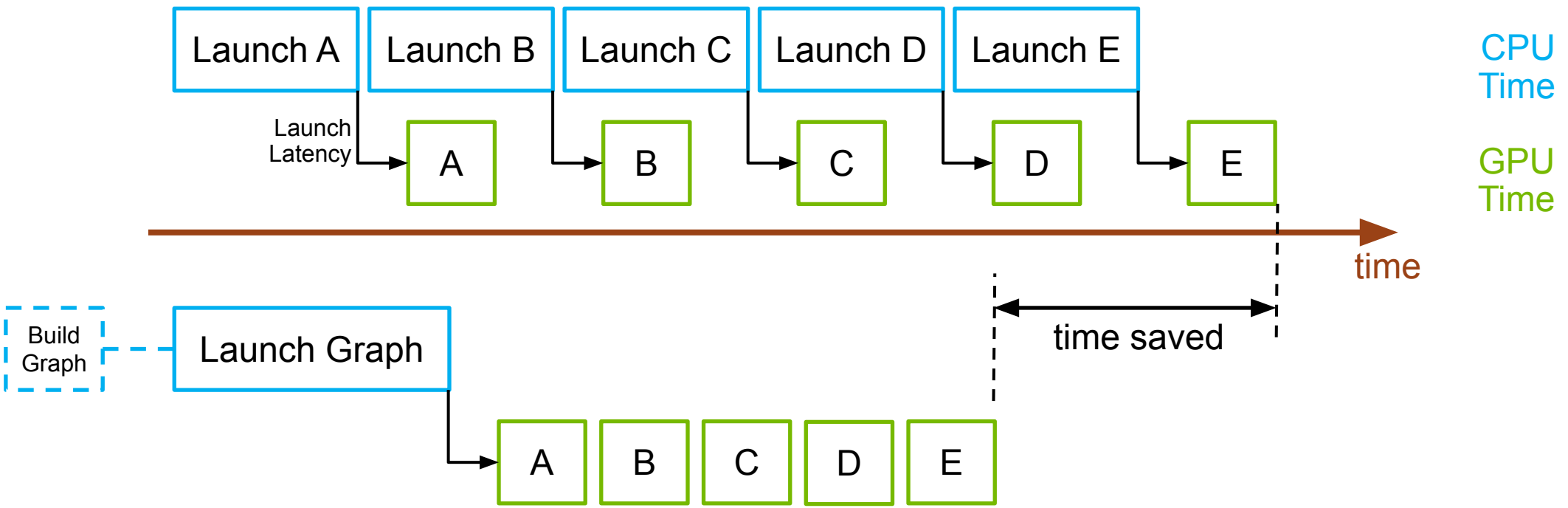
CUDA is closest to the O/S and the hardware

- Can optimize **multi-device** dependencies
- Can optimize **heterogeneous** dependencies
- Especially if executing Graphs

EXECUTION DETAILS

LAUNCH OVERHEAD REDUCTION

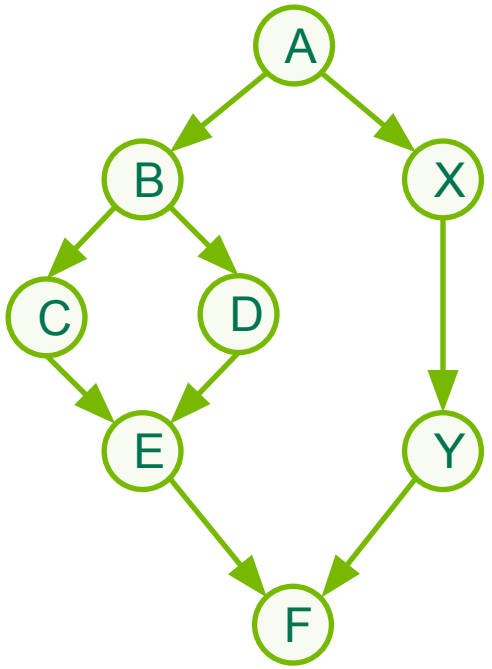
Reducing System Overheads Around Short-Running Kernels



When kernel runtime is short, execution time is dominated by CPU launch cost

TAKEAWAYS

- Cuda Graphs
  - Efficient way to express dependency
- Performance Optimization
  - Launch latency



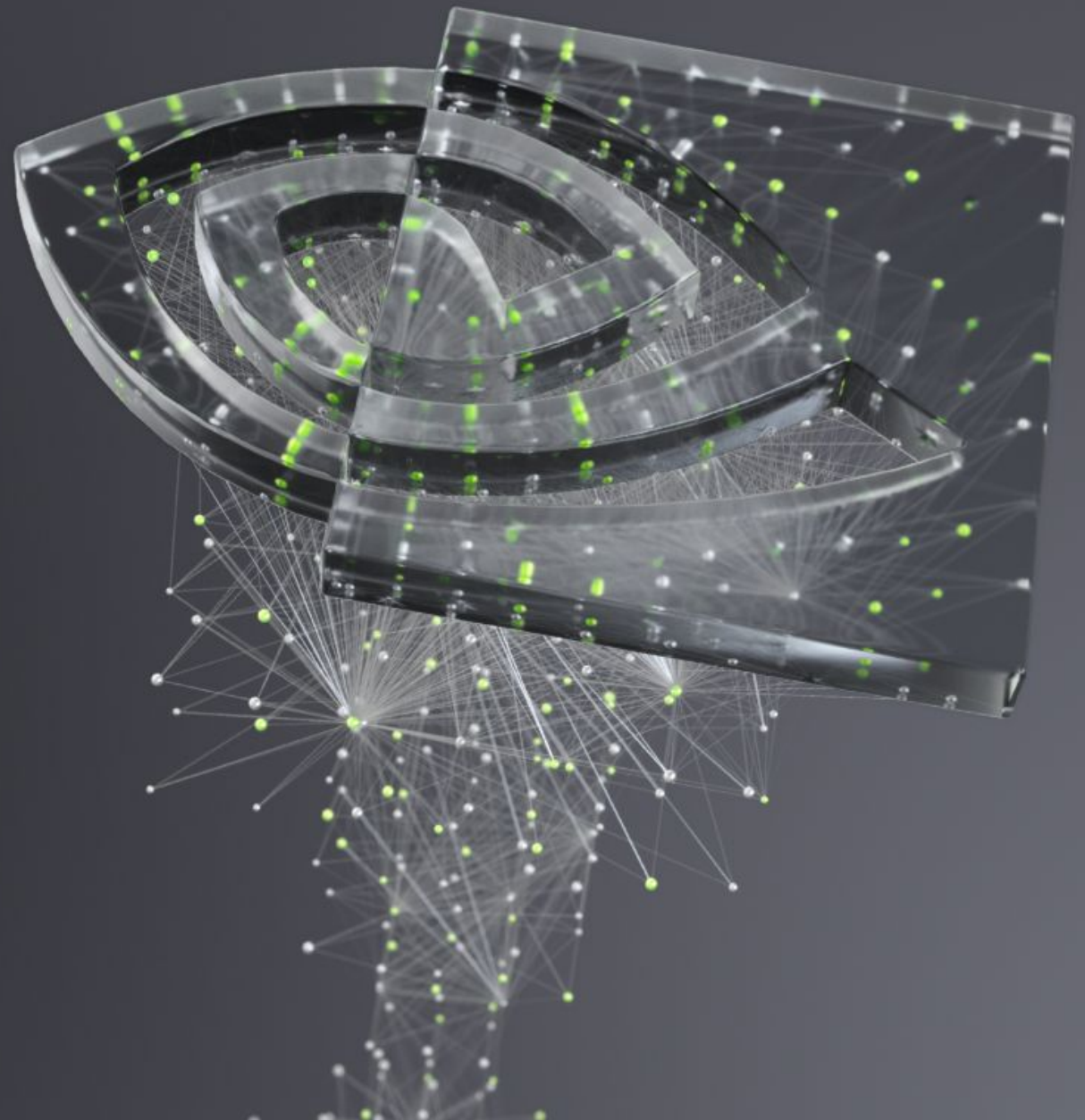
FURTHER STUDY

- Effortless CUDA Graphs GTC Spring 2021 talk
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32082/
- Cuda Memory Nodes
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#graph-memory-nodes
- Cuda Graphs API Documentation:
  - https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

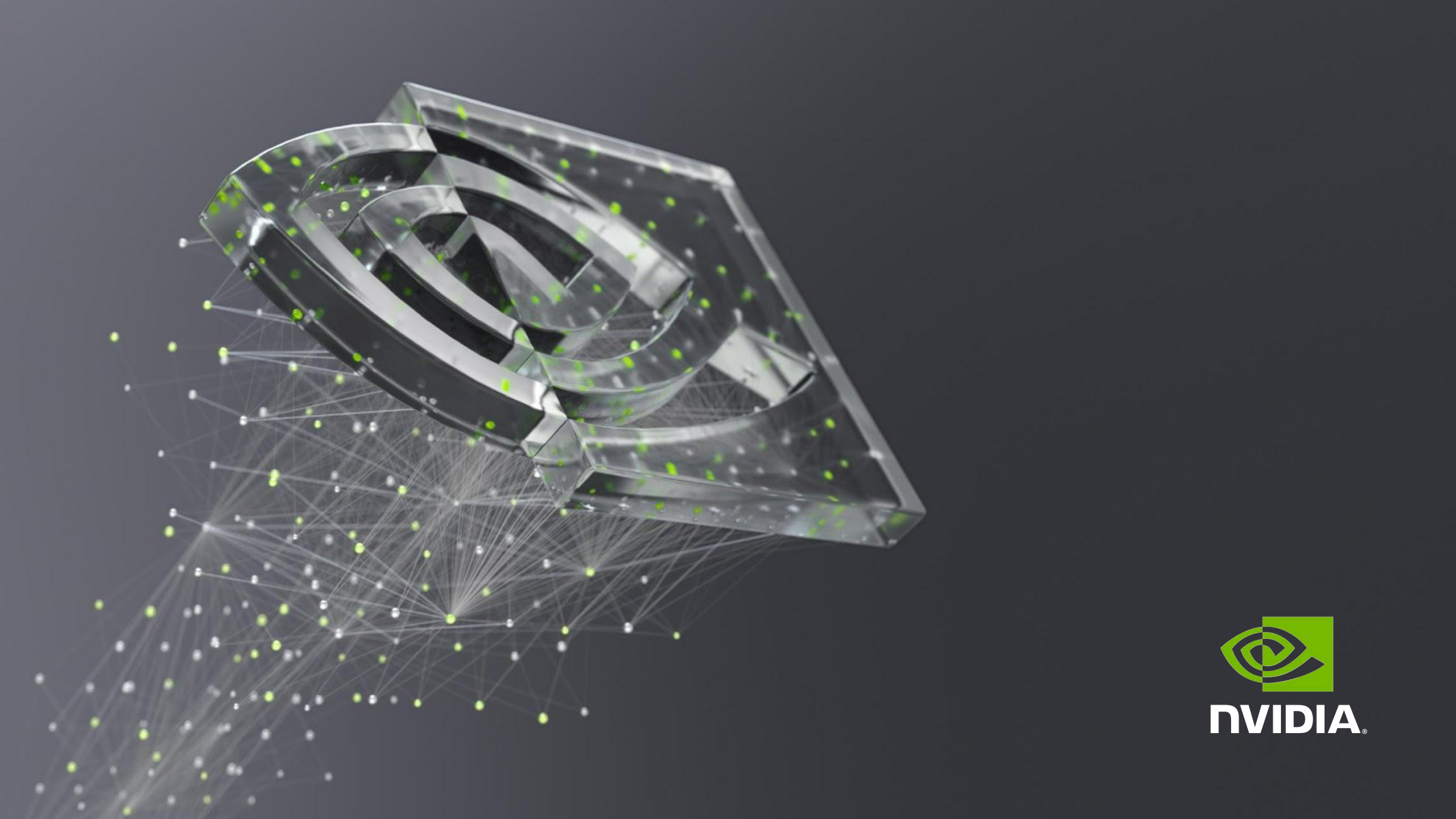
HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw13/README.md

- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming



QUESTIONS?


NVIDIA