Fundamentals of Accelerated Computing with CUDA C/C++ – NVIDIA

# Accelerating Applications with CUDA C/C++

Accelerated computing is replacing CPU-only computing as best practice. The litany of breakthroughs driven by accelerated computing, the ever increasing demand for accelerated applications, programming conventions that ease writing them, and constant improvements in the hardware that supports them, are driving this inevitible transition.

At the center of accelerated computing's success, both in terms of its impressive performance, and its ease of use, is the [CUDA](https://developer.nvidia.com/about-cuda) compute platform. CUDA provides a coding paradigm that extends languages like C, C++, Python, and Fortran, to be capable of running accelerated, massively parallelized code on the world's most performant parallel processors: NVIDIA GPUs. CUDA accelerates applications drastically with little effort, has an ecosystem of highly optimized libraries for [DNN](https://developer.nvidia.com/cudnn), [BLAS](https://developer.nvidia.com/cublas), [graph analytics](https://developer.nvidia.com/nvgraph), [FFT](https://developer.nvidia.com/cufft), and more, and also ships with powerful [command line](http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview) and [visual profilers](http://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual).

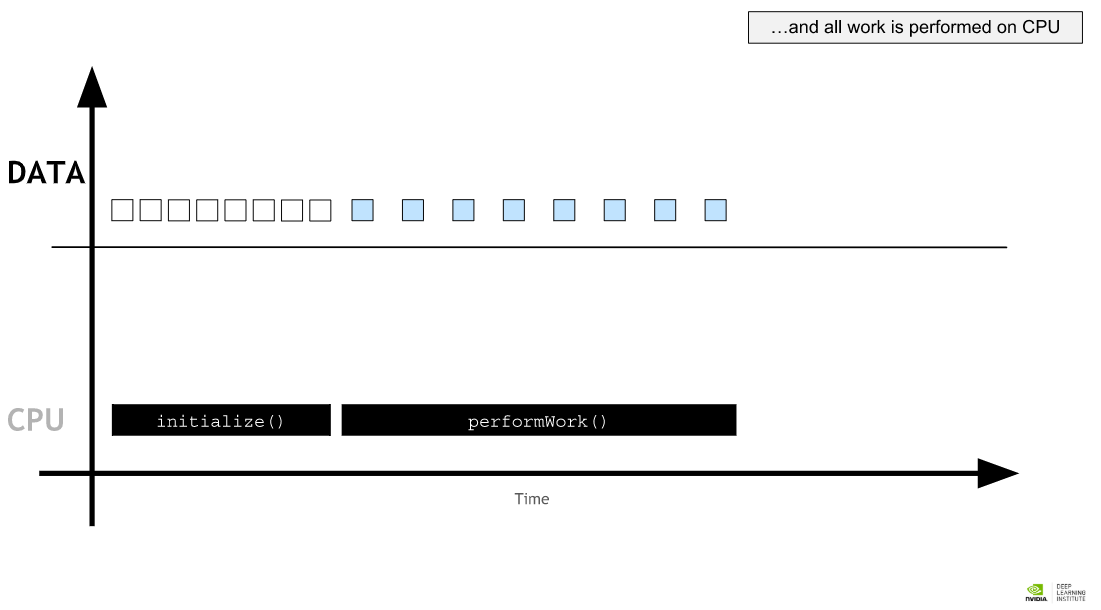
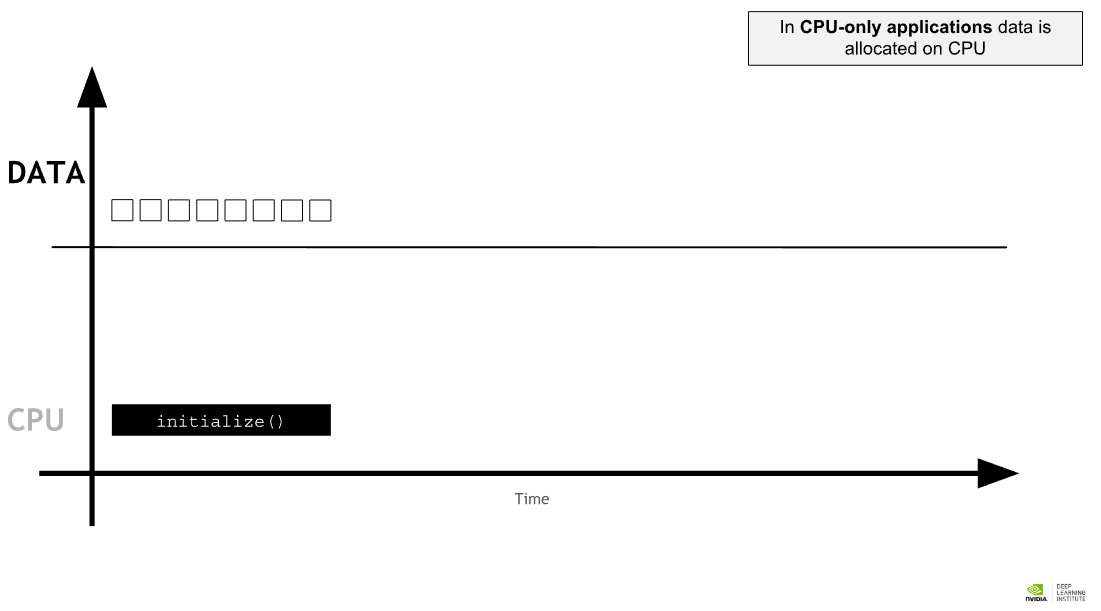
CUDA supports many, if not most, of the [world's most performant applications](https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/?product_category_id=58,59,60,293,98,172,223,227,228,265,487,488,114,389,220,258,461&search=) in, [Computational Fluid Dynamics](https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/?product_category_id=10,12,16,17,19,51,53,71,87,121,124,156,157,195,202,203,204,312,339,340,395,407,448,485,517,528,529,541,245,216,104,462,513,250,492,420,429,490,10,12,16,17,19,51,53,71,87,121,124,156,157,195,202,203,204,312,339,340,395,407,448,485,517,528,529,541,245,216,104,462,513,250,492,420,429,490,10,12,16,17,19,51,53,71,87,121,124,156,157,195,202,203,204,312,339,340,395,407,448,485,517,528,529,541,245,216,104,462,513,250,492,420,429,490&search=), [Molecular Dynamics](https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/?product_category_id=8,57,92,123,211,213,237,272,274,282,283,307,325,337,344,345,351,362,365,380,396,398,400,435,507,508,519,8,57,92,123,211,213,237,272,274,282,283,307,325,337,344,345,351,362,365,380,396,398,400,435,507,508,519,8,57,92,123,211,213,237,272,274,282,283,307,325,337,344,345,351,362,365,380,396,398,400,435,507,508,519,8,57,92,123,211,213,237,272,274,282,283,307,325,337,344,345,351,362,365,380,396,398,400,435,507,508,519&search=), [Quantum Chemistry](https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/?product_category_id=8,57,92,123,211,213,237,272,274,282,283,307,325,337,344,345,351,362,365,380,396,398,400,435,507,508,519,8,57,92,123,211,213,237,272,274,282,283,307,325,337,344,345,351,362,365,380,396,398,400,435,507,508,519&search=), [Physics](https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/?product_category_id=6,24,116,118,119,135,229,231,372,373,392,393,489,493,494,495,496,497,498,67,170,216,281,6,24,116,118,119,135,229,231,372,373,392,393,489,493,494,495,496,497,498,67,170,216,281,6,24,116,118,119,135,229,231,372,373,392,393,489,493,494,495,496,497,498,67,170,216,281,6,24,116,118,119,135,229,231,372,373,392,393,489,493,494,495,496,497,498,67,170,216,281,6,24,116,118,119,135,229,231,372,373,392,393,489,493,494,495,496,497,498,67,170,216,281&search=) and HPC.

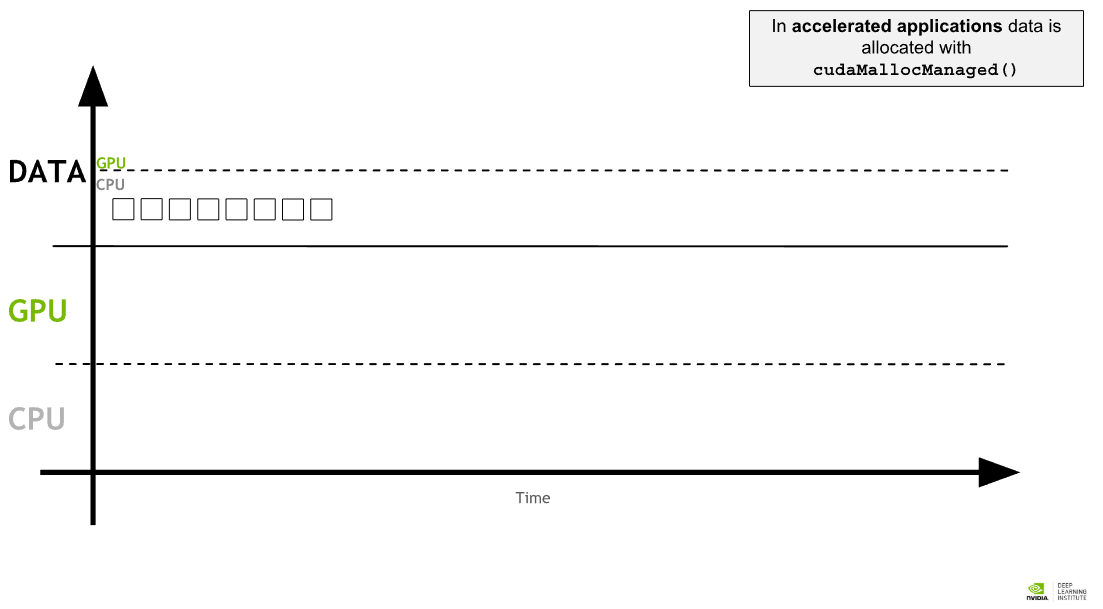
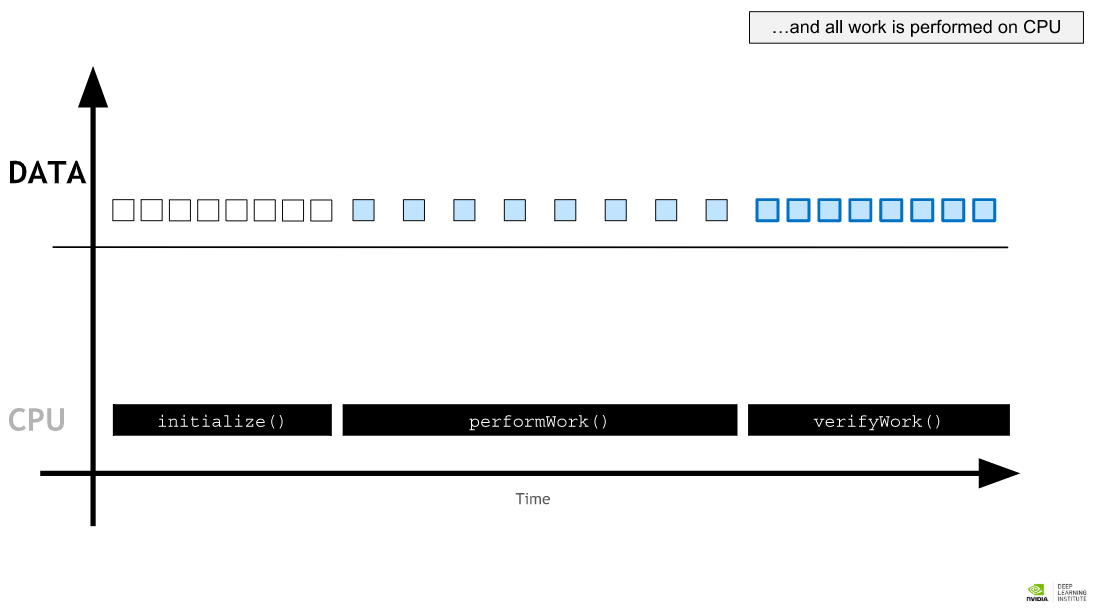
Learning CUDA will enable you to accelerate your own applications. Accelerated applications perform much faster than their CPU-only couterparts, and make possible computations that would be otherwise prohibited given the limited performance of CPU-only applications. In this lab you will receive an introduction to programming accelerated applications with CUDA C/C++, enough to be able to begin work accelerating your own CPU-only applications for performance gains, and for moving into novel computational territory.

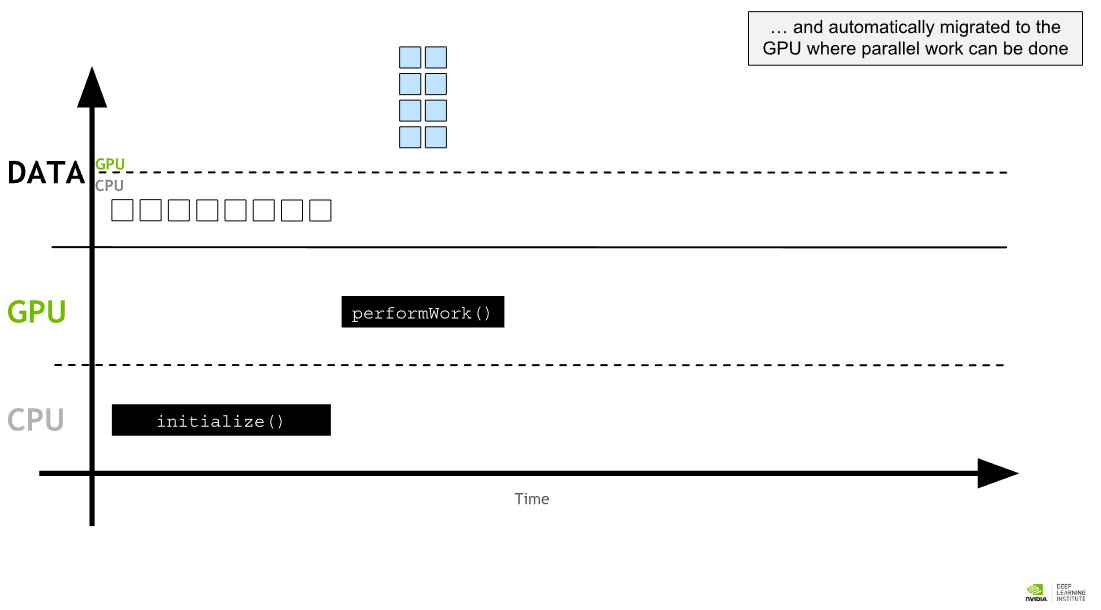
## Accelerated Systems

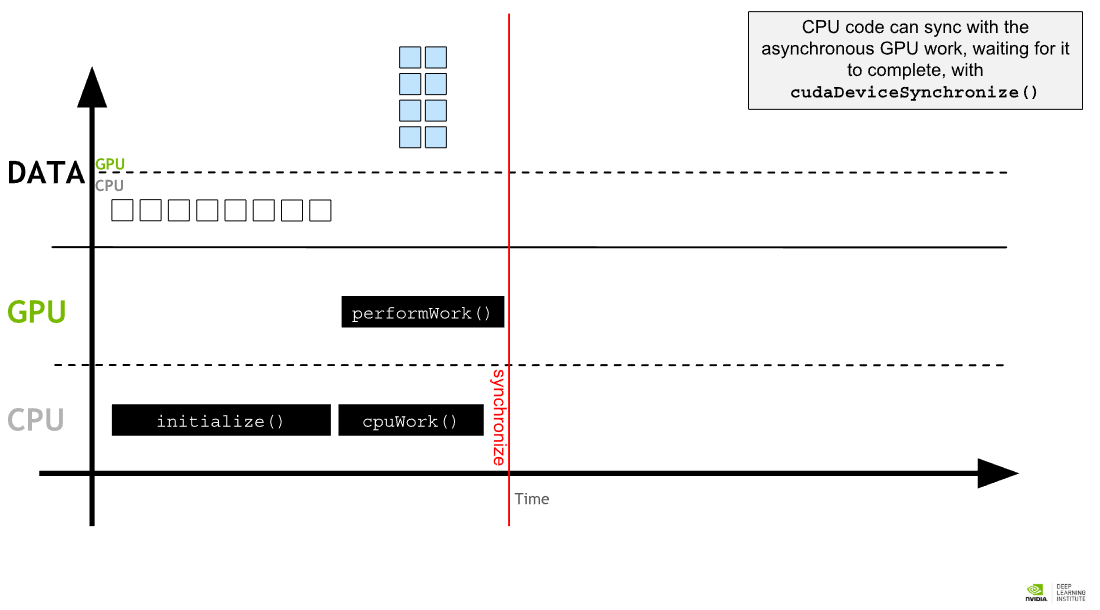
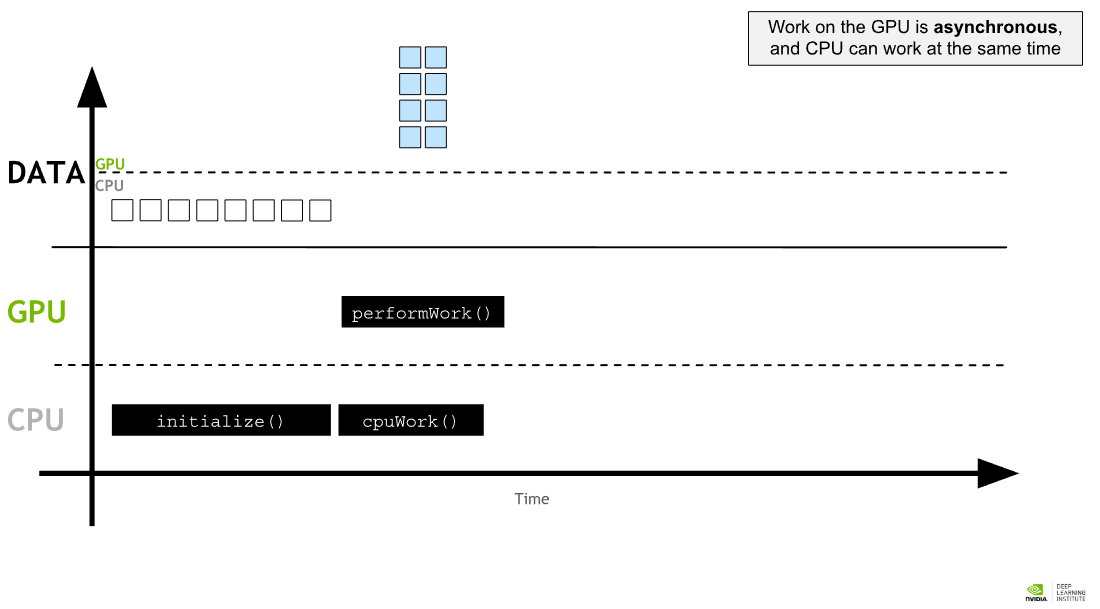
Accelerated systems, also referred to as heterogeneous systems, are those composed of both CPUs and GPUs. Accelerated systems run CPU programs which in turn, launch functions that will benefit from the massive parallelism provided by GPUs.

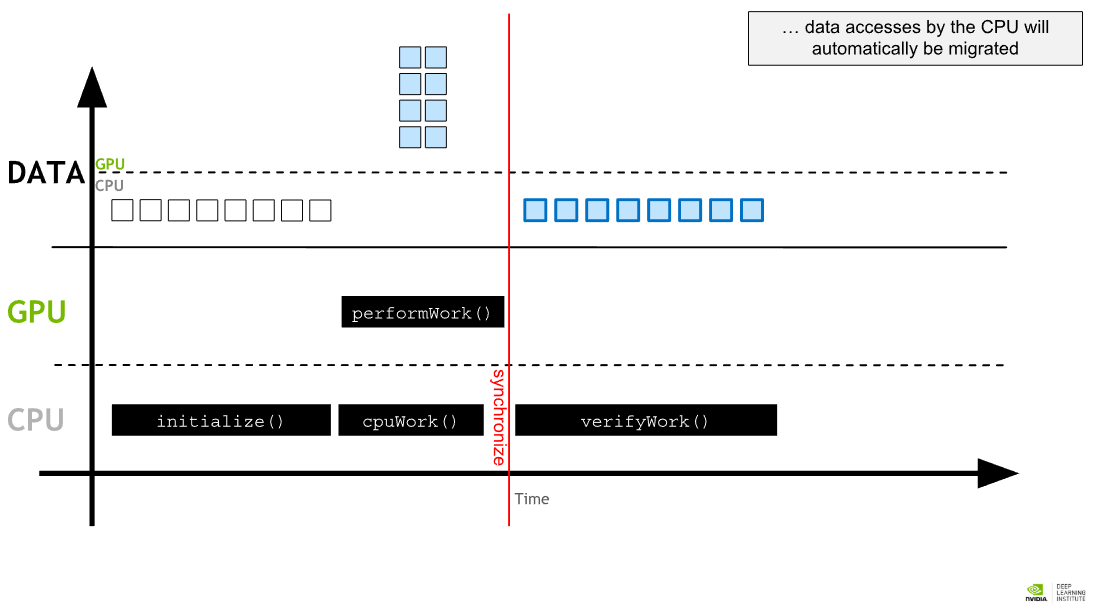
## GPU-accelerated Vs. CPU-only Applications











## Writing Application Code for the GPU

CUDA provides extensions for many common programming languages, in the case of this lab, C/C++. These language extensions easily allow developers to run functions in their source code on a GPU.

Below is a .cu file (.cu is the file extension for CUDA-accelerated programs). It contains two functions, the first which will run on the CPU, the second which will run on the GPU. Spend a little time identifying the differences between the functions, both in terms of how they are defined, and how they are invoked.

void CPUFunction()

{

printf("This function is defined to run on the CPU.\n");

}

\_\_global\_\_ void GPUFunction()

{

printf("This function is defined to run on the GPU.\n");

}

int main()

{

CPUFunction();

GPUFunction**<<<**1, 1**>>>**();

cudaDeviceSynchronize();

}

Here are some important lines of code to highlight, as well as some other common terms used in accelerated computing:

\_\_global\_\_ void GPUFunction()

* The \_\_global\_\_ keyword indicates that the following function will run on the GPU, and can be invoked **globally**, which in this context means either by the CPU, or, by the GPU.
* Often, code executed on the CPU is referred to as **host** code, and code running on the GPU is referred to as **device** code.
* Notice the return type void. It is required that functions defined with the \_\_global\_\_ keyword return type void.

GPUFunction<<<1, 1>>>();

* Typically, when calling a function to run on the GPU, we call this function a **kernel**, which is **launched**.
* When launching a kernel, we must provide an **execution configuration**, which is done by using the <<< ... >>> syntax just prior to passing the kernel any expected arguments.
* At a high level, execution configuration allows programmers to specify the **thread hierarchy** for a kernel launch, which defines the number of thread groupings (called **blocks**), as well as how many **threads** to execute in each block. Execution configuration will be explored at great length later in the lab, but for the time being, notice the kernel is launching with 1 block of threads (the first execution configuration argument) which contains 1 thread (the second configuration argument).

cudaDeviceSynchronize();

* Unlike much C/C++ code, launching kernels is **asynchronous**: the CPU code will continue to execute without waiting for the kernel launch to complete.
* A call to cudaDeviceSynchronize, a function provided by the CUDA runtime, will cause the host (CPU) code to wait until the device (GPU) code completes, and only then resume execution on the CPU.

## Compiling and Running Accelerated CUDA Code

This section contains details about the nvcc command you issued above to compile and run your .cu program.

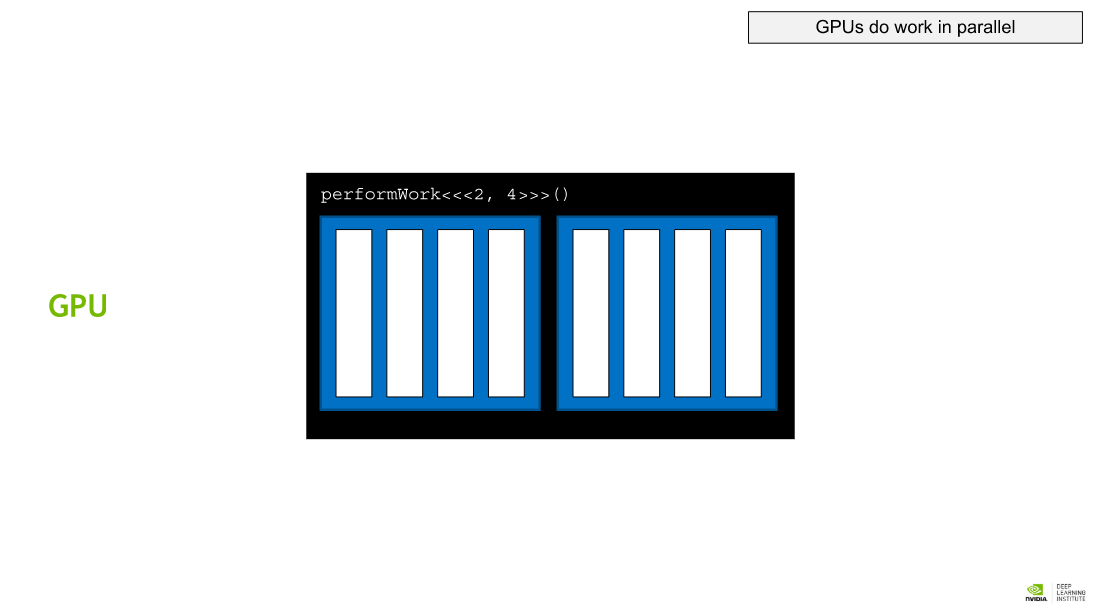
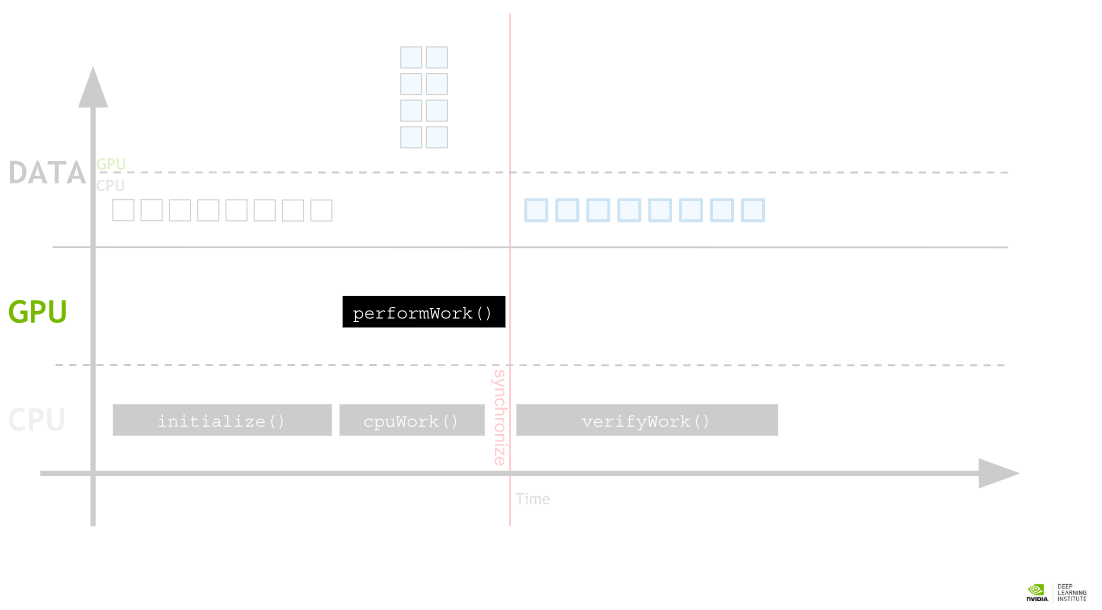
The CUDA platform ships with the [NVIDIA CUDA Compiler](http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html) nvcc, which can compile CUDA accelerated applications, both the host, and the device code they contain. For the purposes of this lab, nvcc discussion will be pragmatically scoped to suit our immediate needs. After completing the lab, For anyone interested in a deeper dive into nvcc, start with the [documentation](http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html).

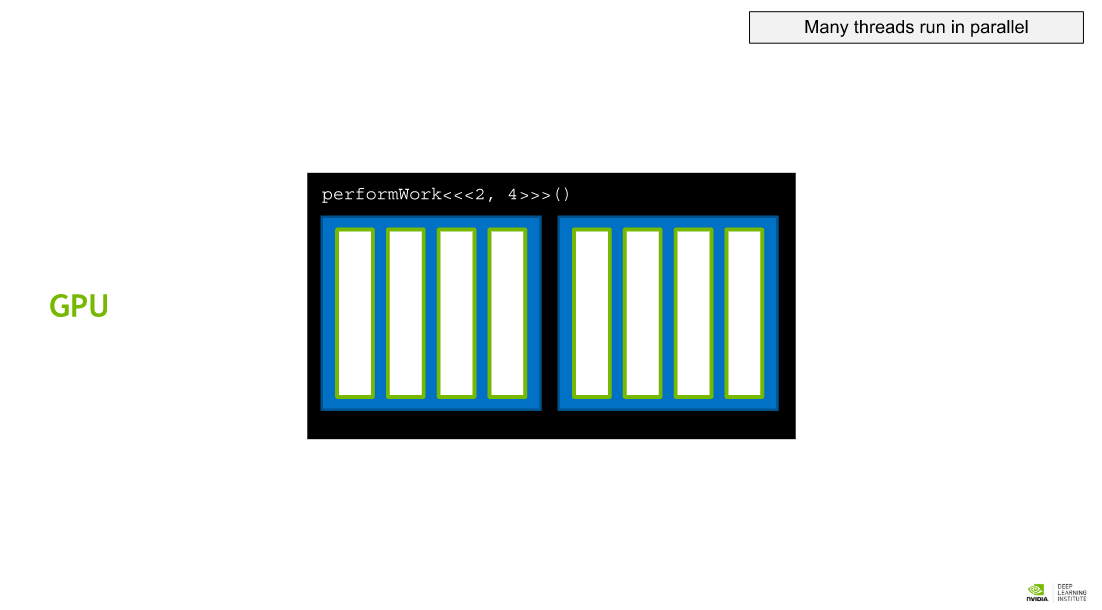
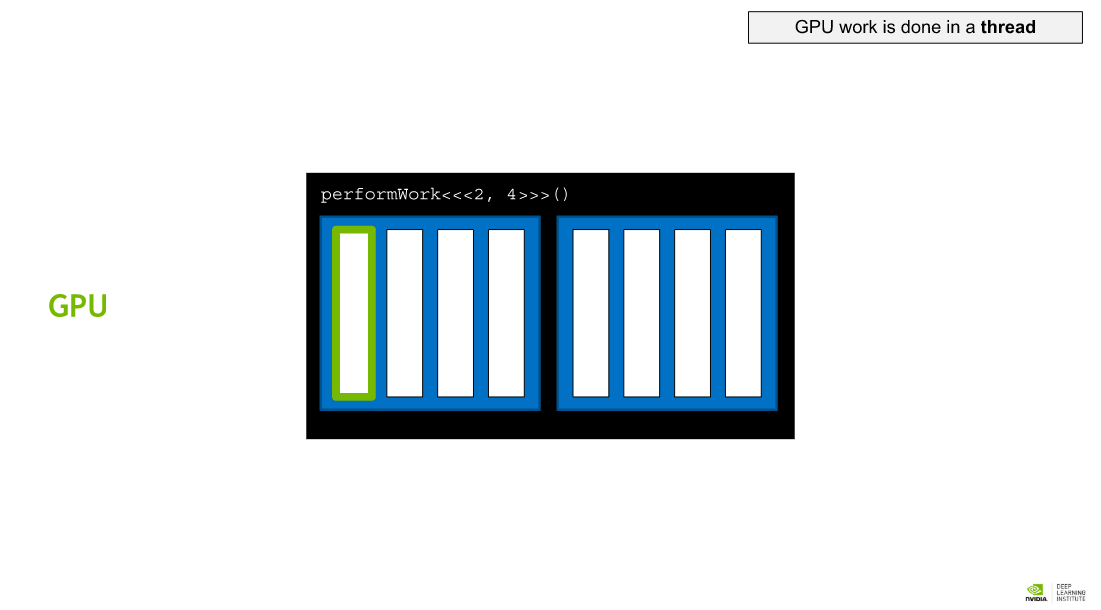
nvcc will be very familiar to experienced gcc users. Compiling, for example, a some-CUDA.cu file, is simply:

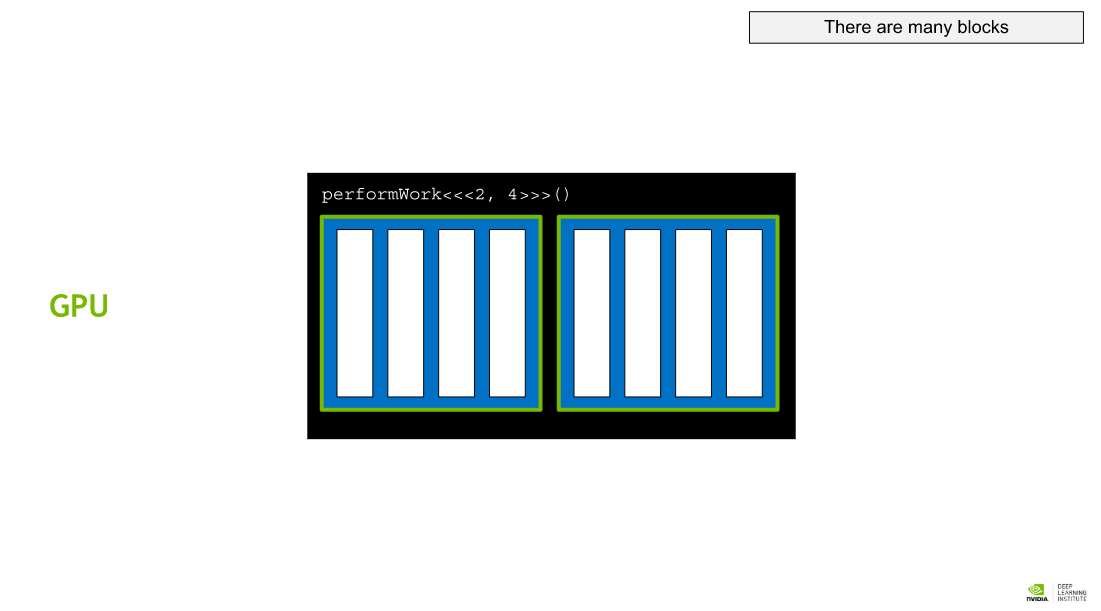
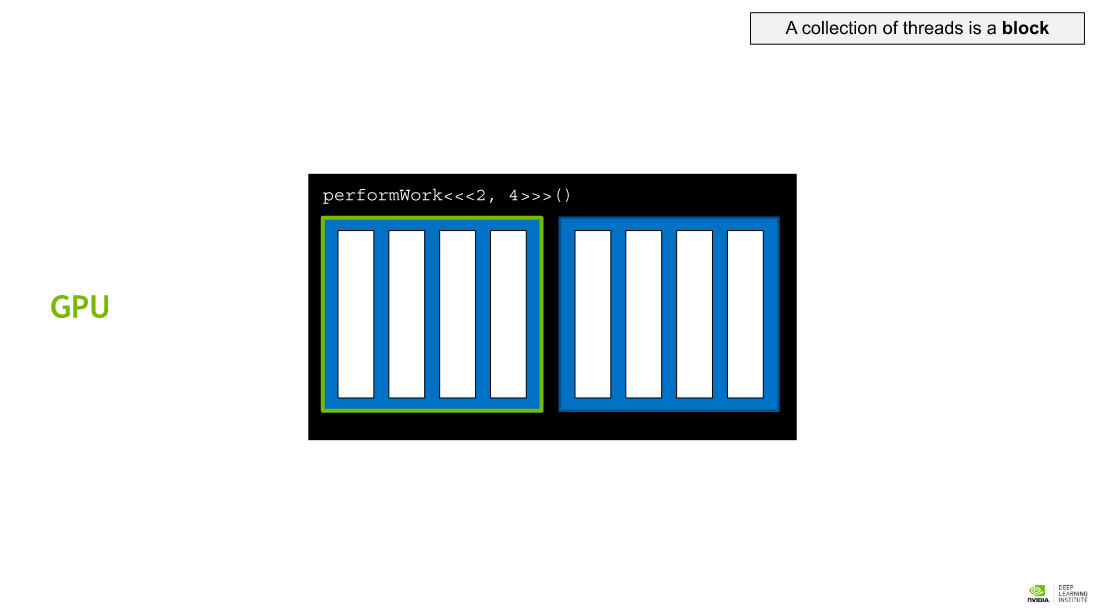
nvcc -arch=sm\_70 -o out some-CUDA.cu -run

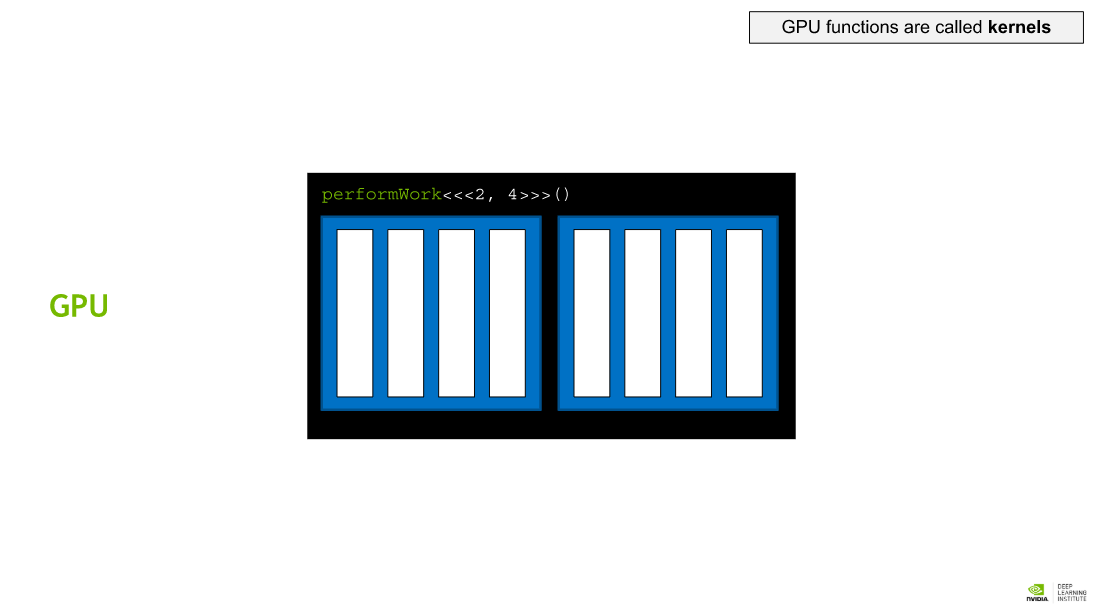
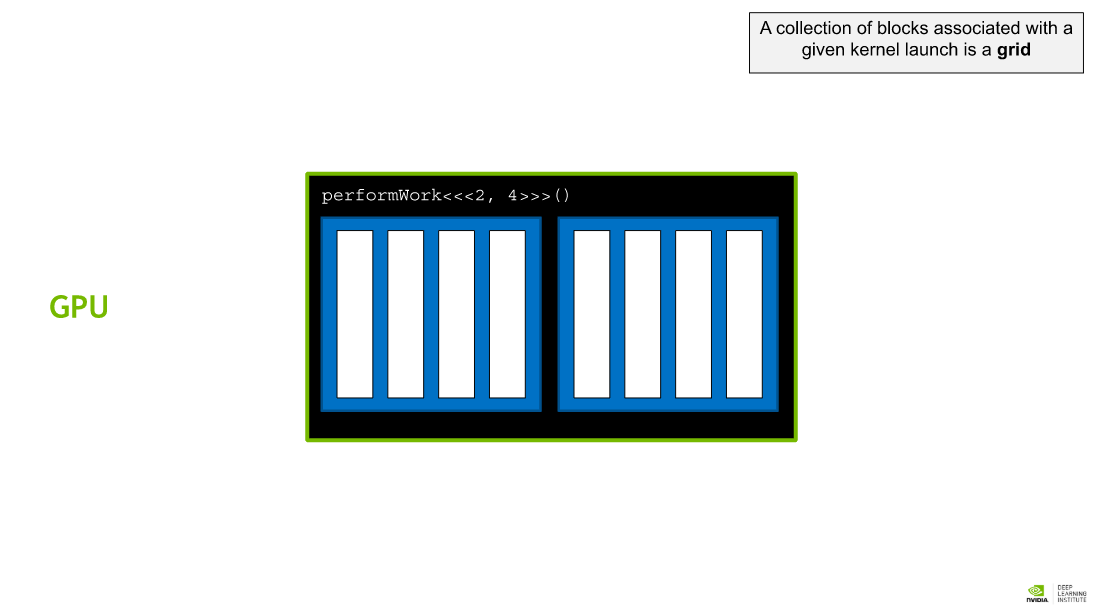
* nvcc is the command line command for using the nvcc compiler.
* some-CUDA.cu is passed as the file to compile.
* The o flag is used to specify the output file for the compiled program.
* The arch flag indicates for which architecture the files must be compiled. For the present case sm\_70 will serve to compile specifically for the Volta GPUs this lab is running on, but for those interested in a deeper dive, please refer to the docs about the [arch flag](http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-gpu-code-generation), [virtual architecture features](http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#gpu-feature-list) and [GPU features](http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#gpu-feature-list).
* As a matter of convenience, providing the run flag will execute the successfully compiled binary.

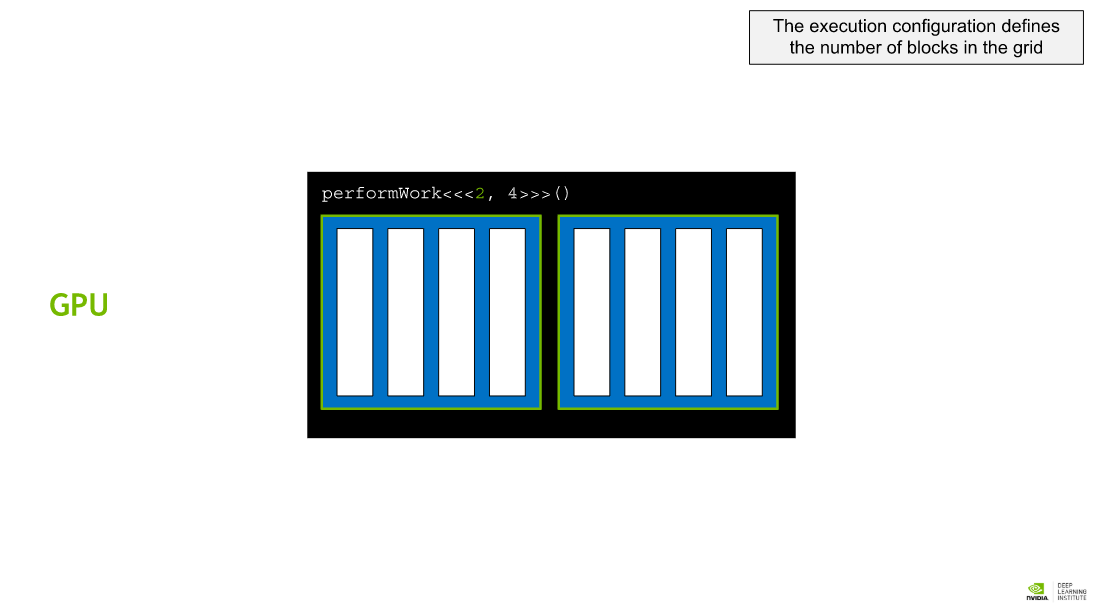
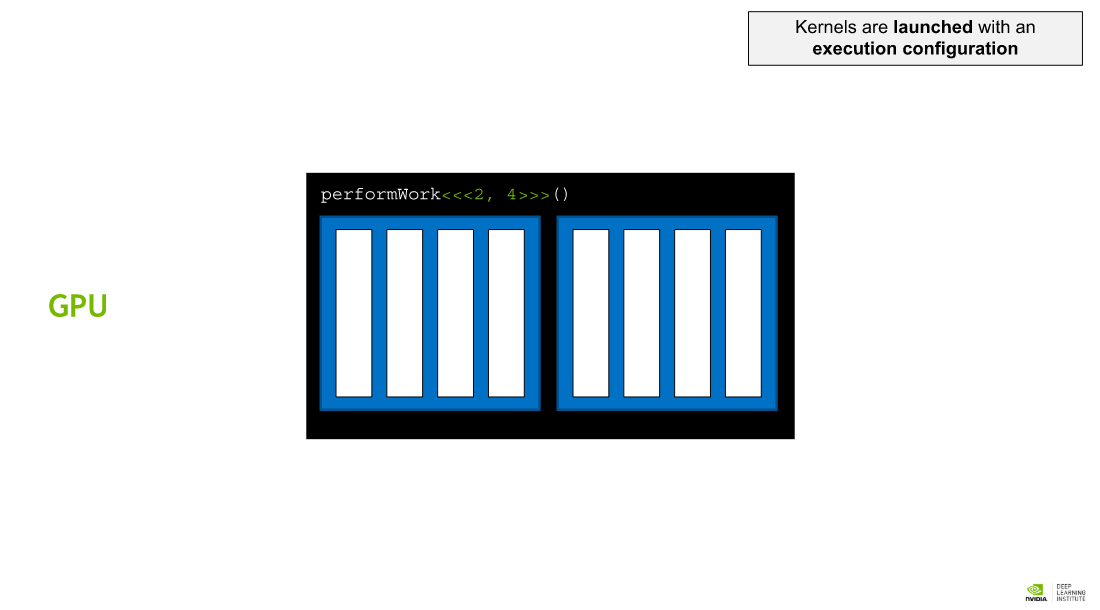
## CUDA Thread Hierarchy

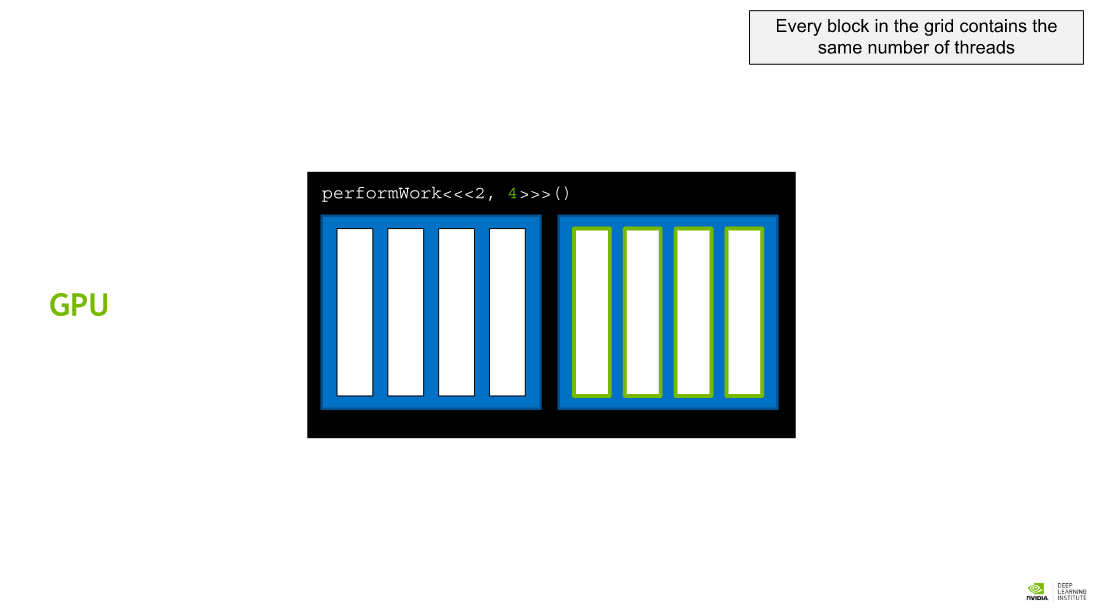
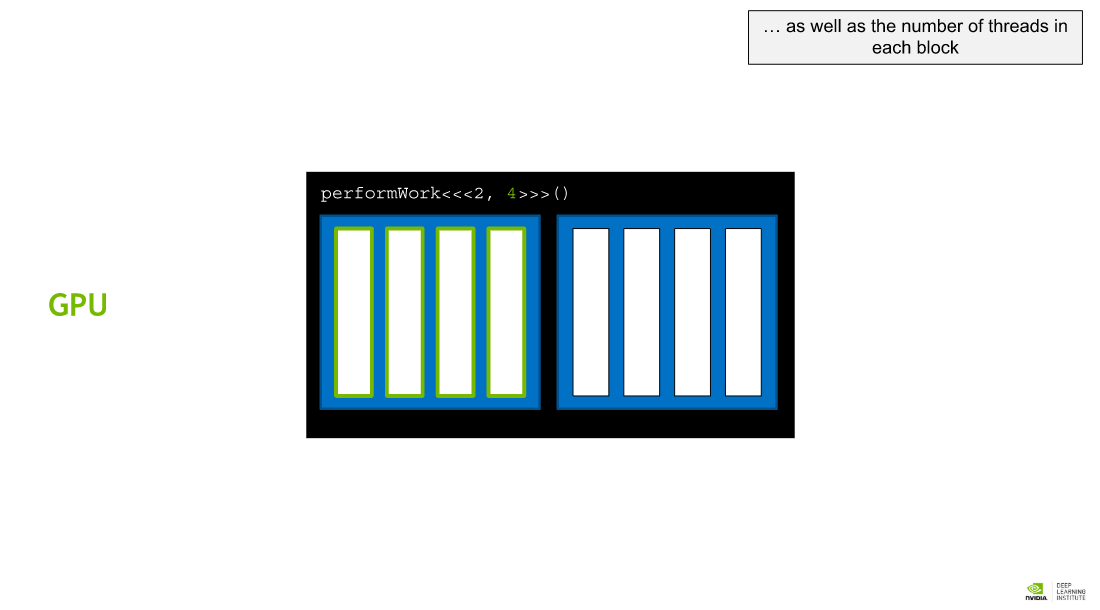












## Launching Parallel Kernels

The execution configuration allows programmers to specify details about launching the kernel to run in parallel on multiple GPU **threads**. More precisely, the execution configuration allows programmers to specifiy how many groups of threads - called **thread blocks**, or just **blocks** - and how many threads they would like each thread block to contain. The syntax for this is:

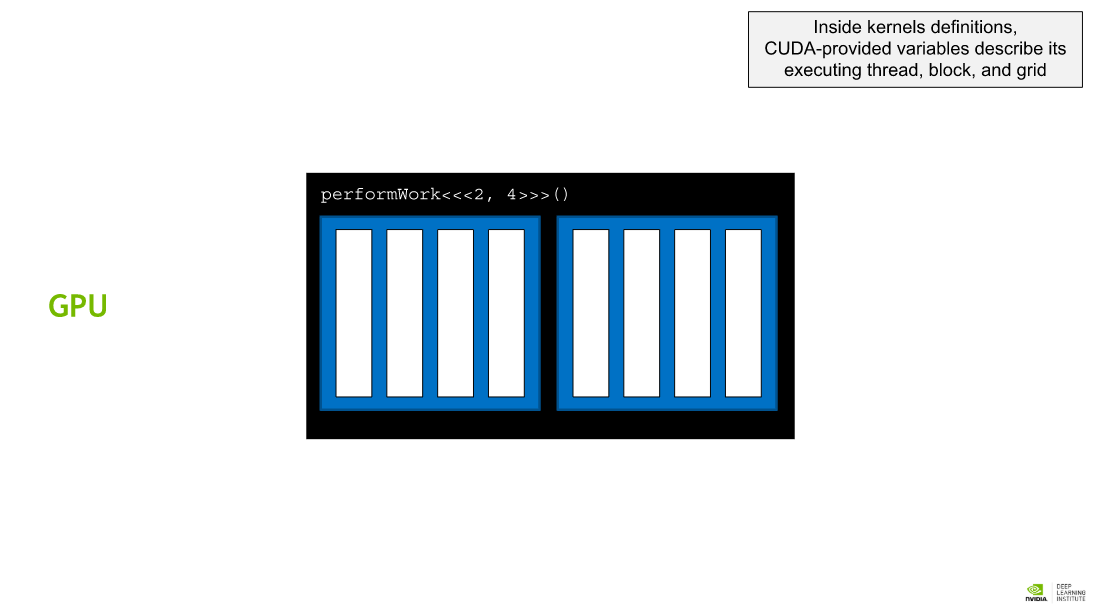
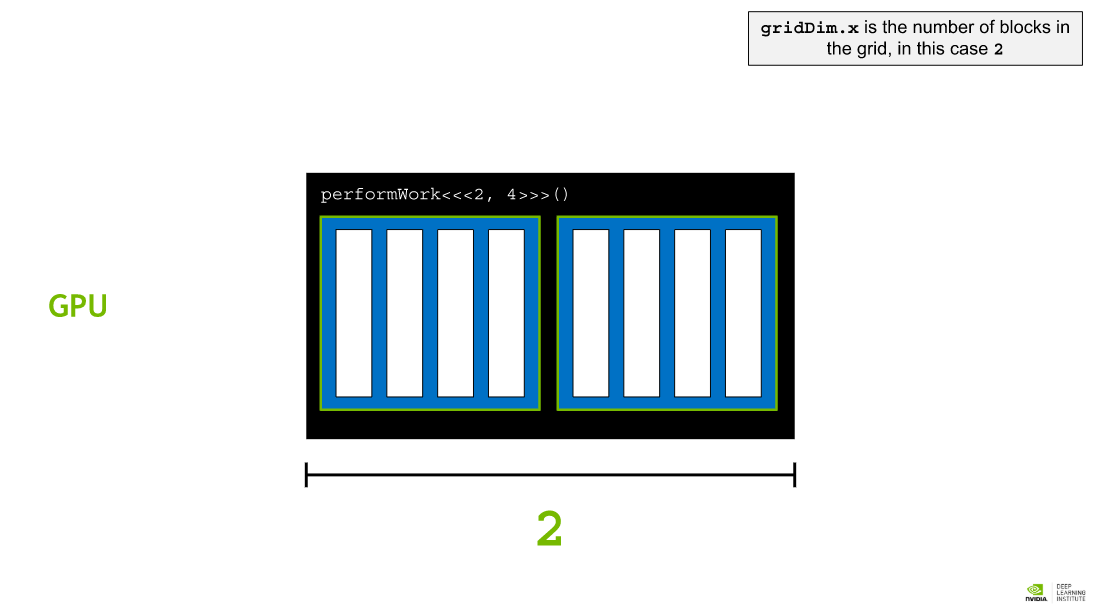
<<< NUMBER\_OF\_BLOCKS, NUMBER\_OF\_THREADS\_PER\_BLOCK>>>

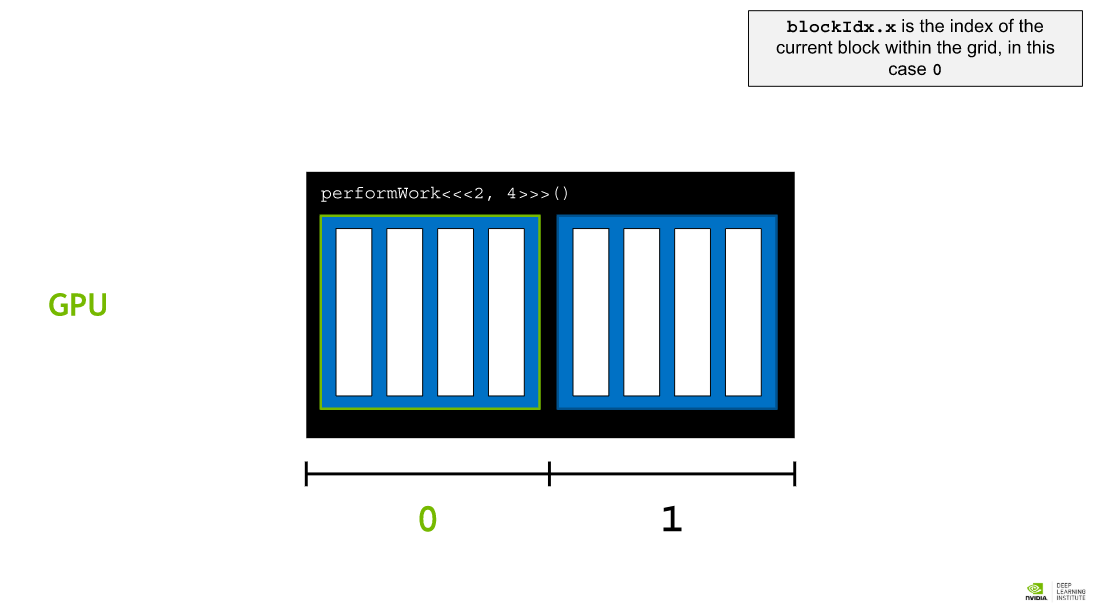
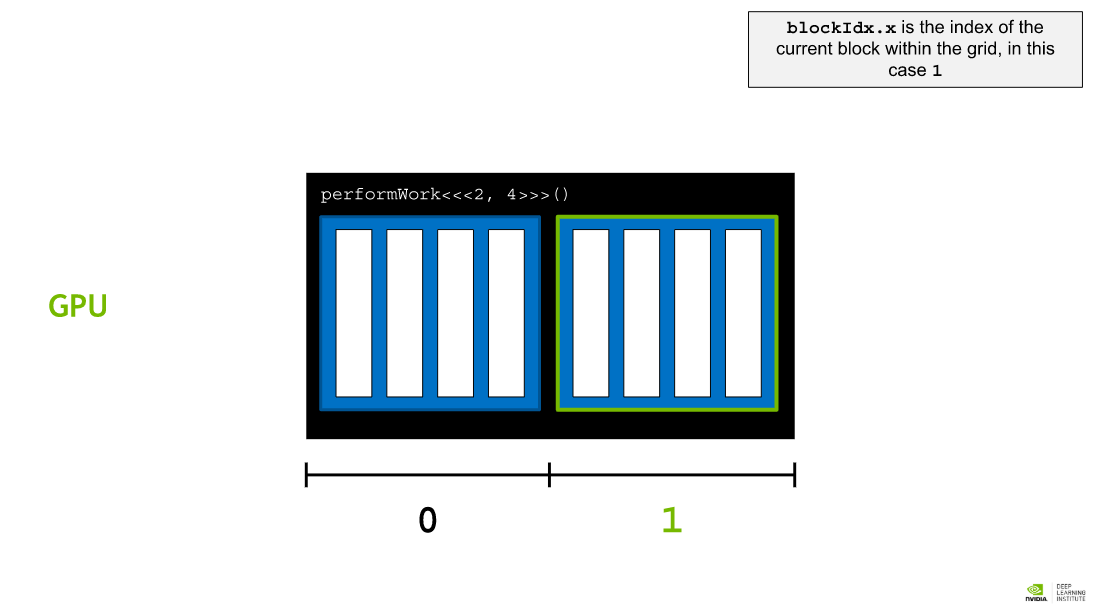
**The kernel code is executed by every thread in every thread block configured when the kernel is launched.**

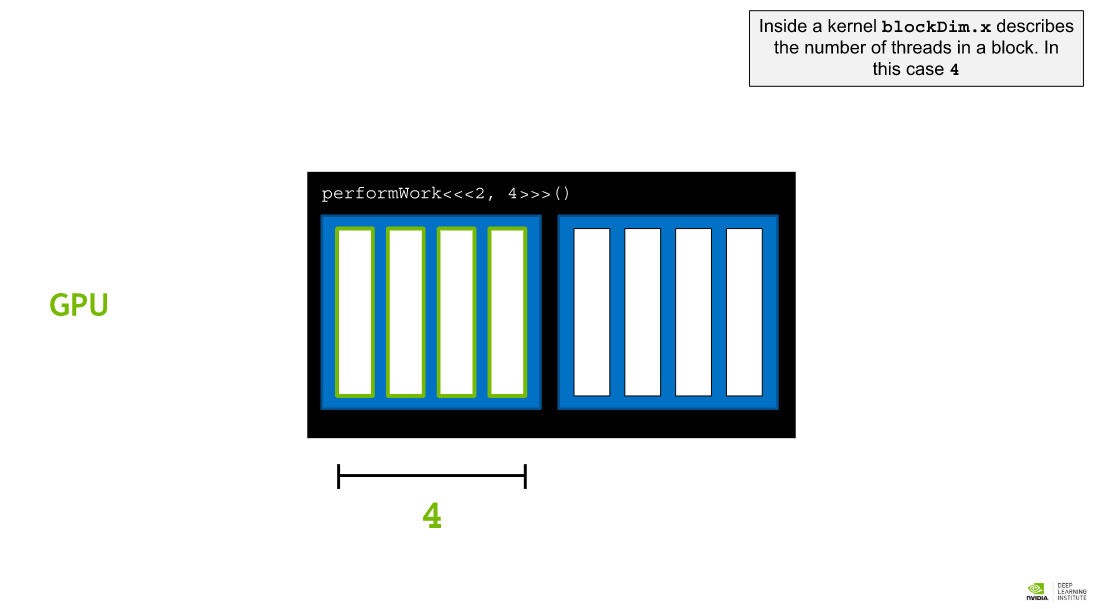
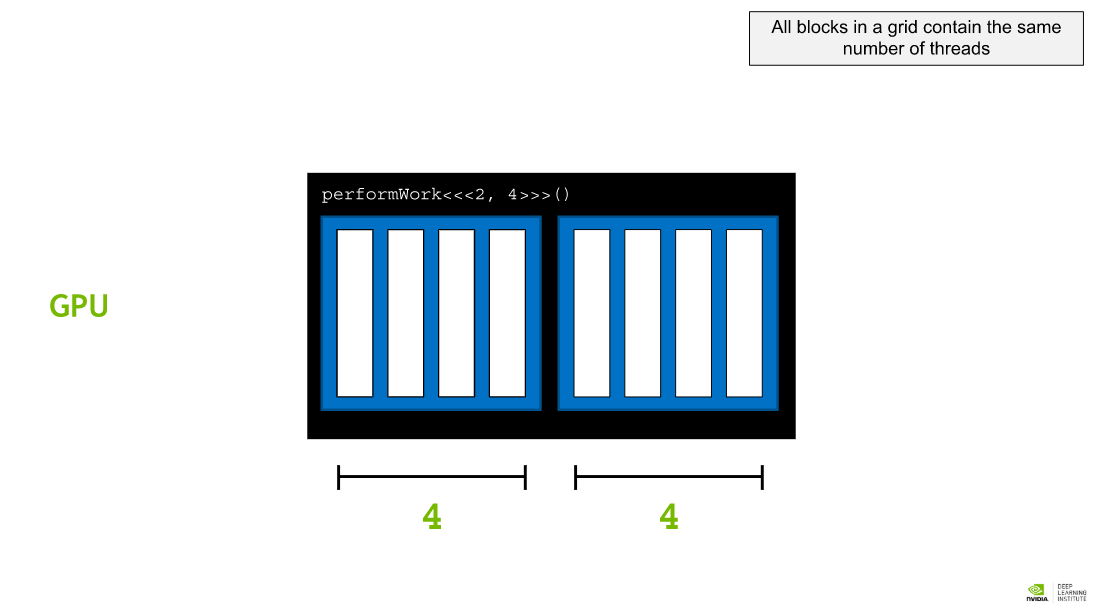
Thus, under the assumption that a kernel called someKernel has been defined, the following are true:

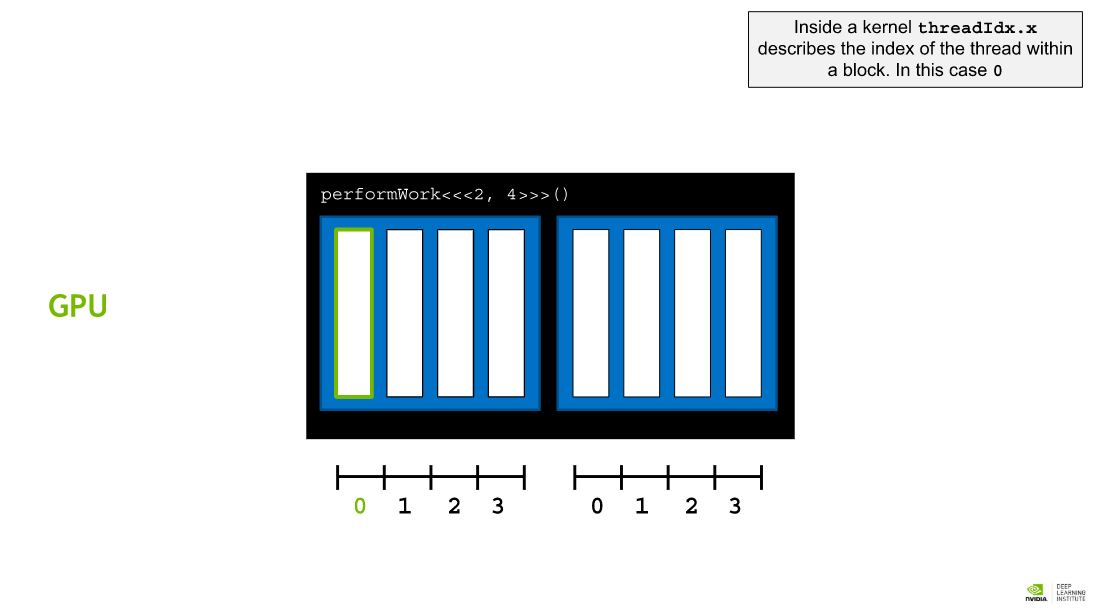
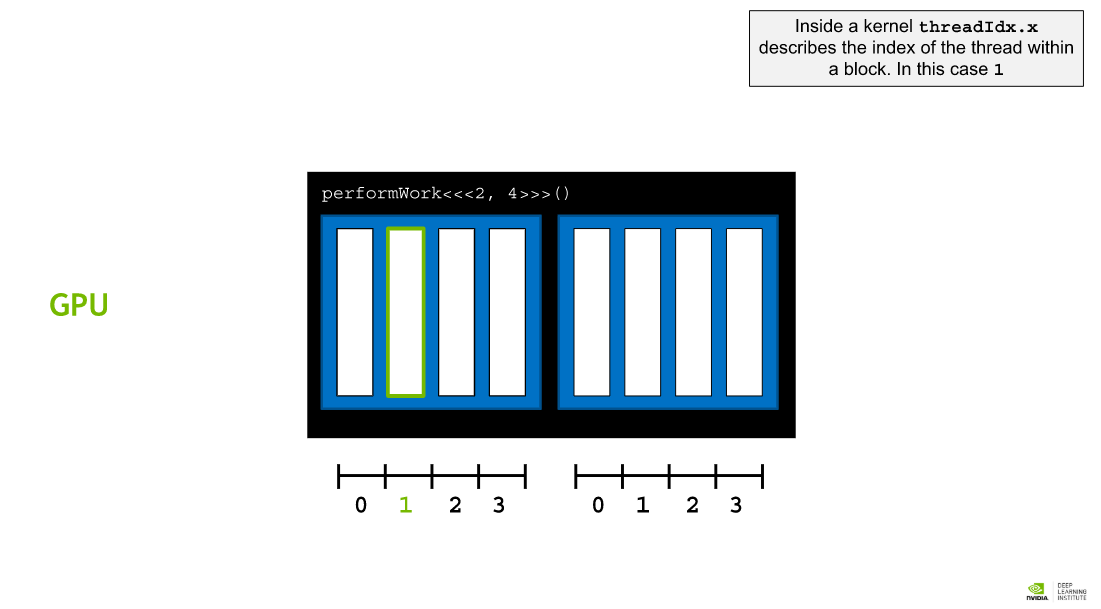
* someKernel<<<1, 1>>() is configured to run in a single thread block which has a single thread and will therefore run only once.
* someKernel<<<1, 10>>() is configured to run in a single thread block which has 10 threads and will therefore run 10 times.
* someKernel<<<10, 1>>() is configured to run in 10 thread blocks which each have a single thread and will therefore run 10 times.
* someKernel<<<10, 10>>() is configured to run in 10 thread blocks which each have 10 threads and will therefore run 100 times.

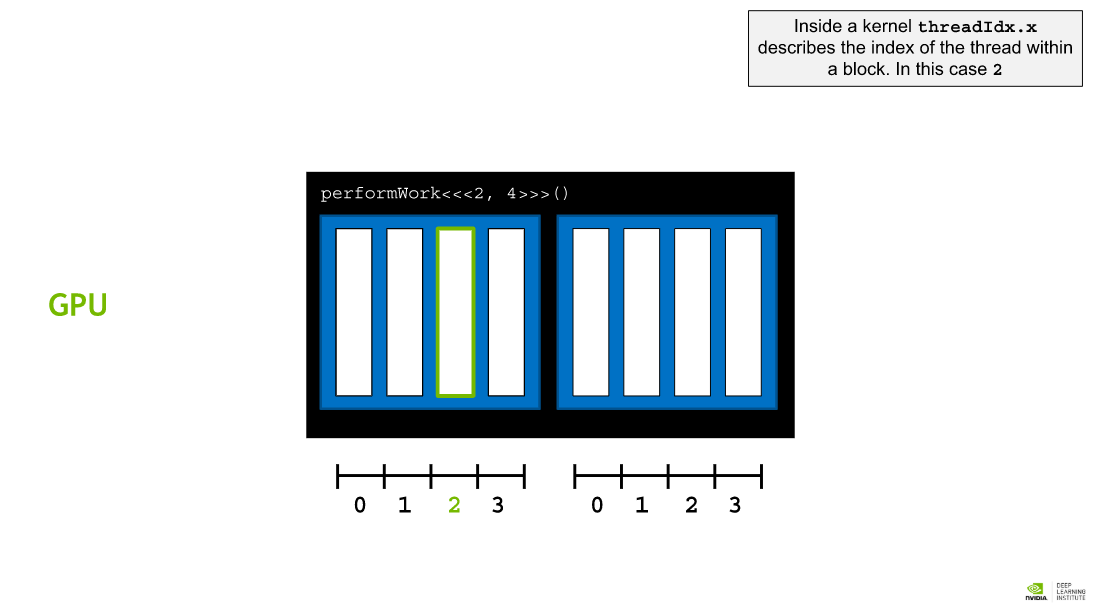
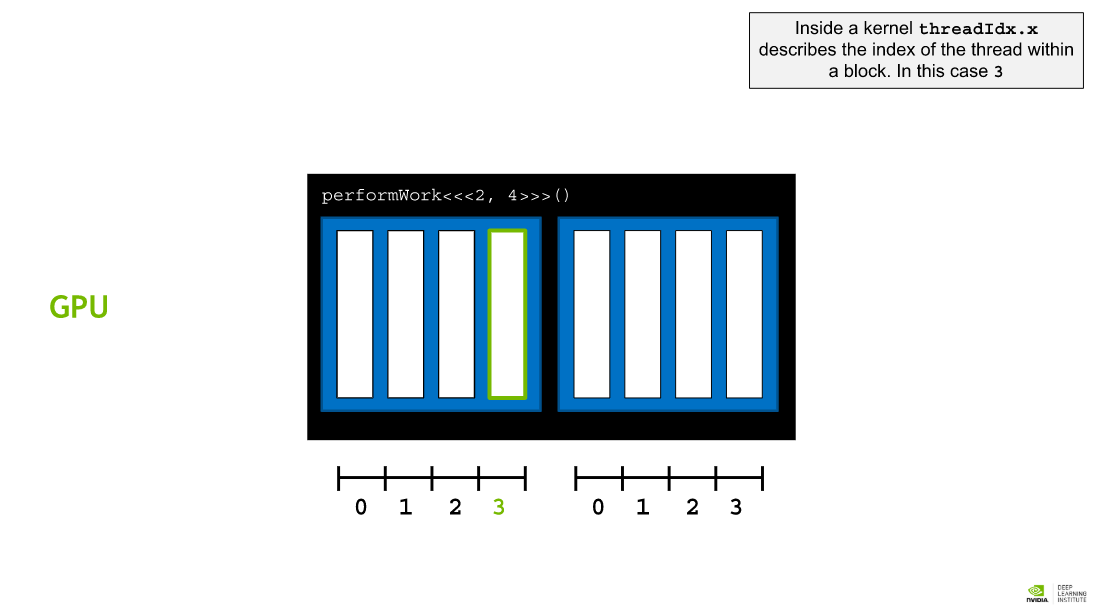
## CUDA-Provided Thread Hierarchy Variables

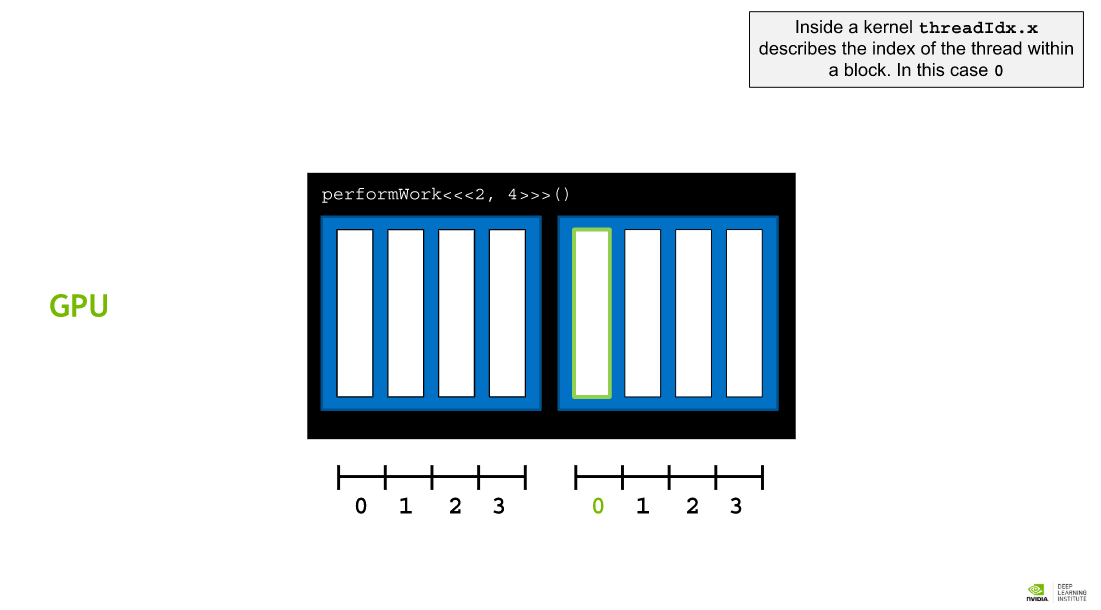
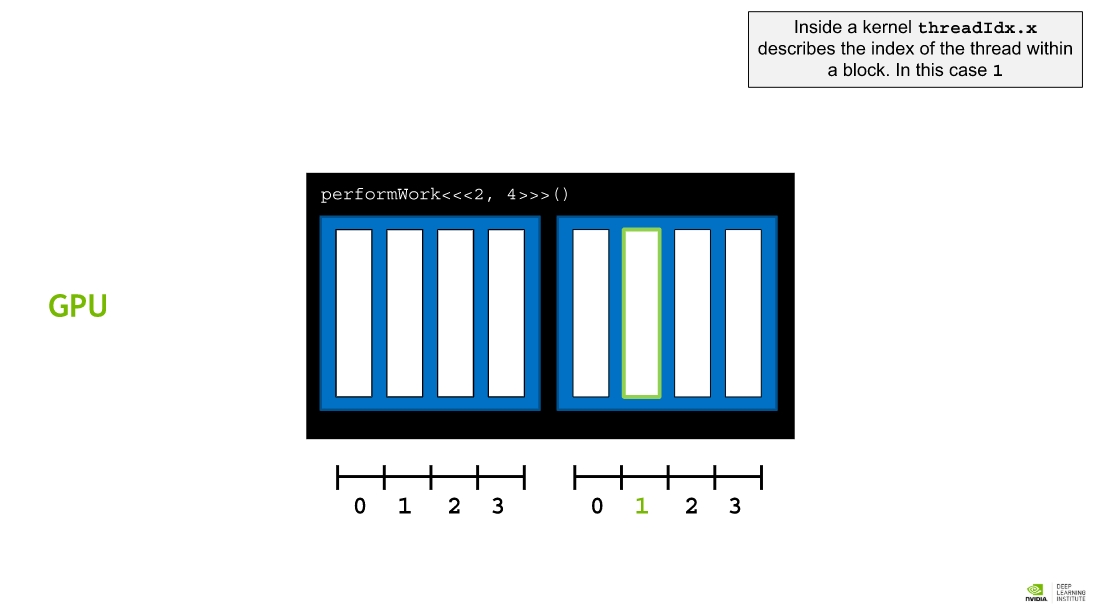
 

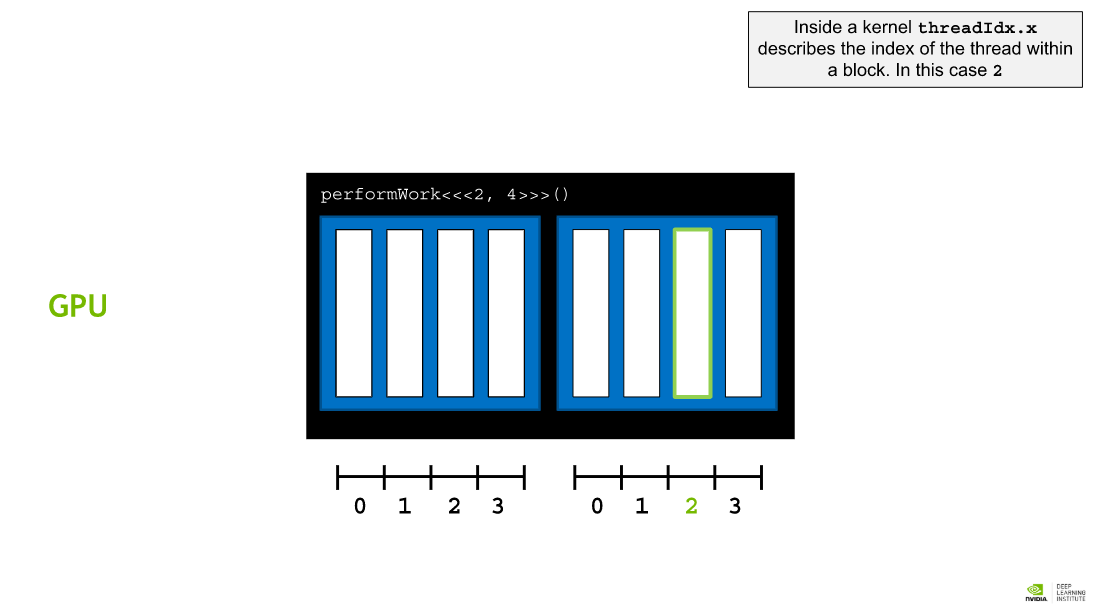
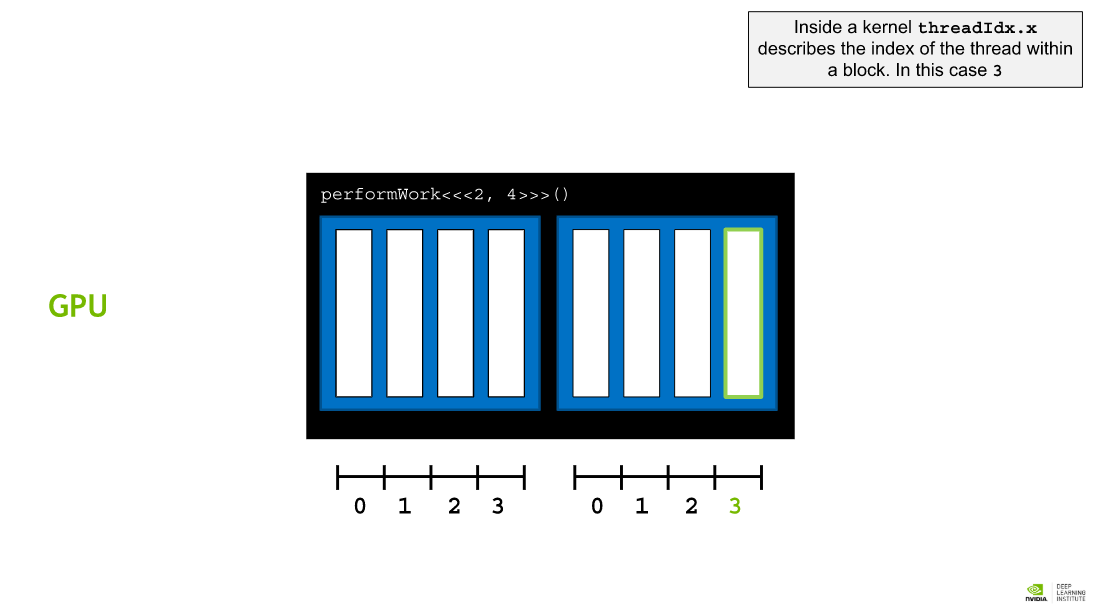
 

## Thread and Block Indices

Each thread is given an index within its thread block, starting at 0. Additionally, each block is given an index, starting at 0. Just as threads are grouped into thread blocks, blocks are grouped into a **grid**, which is the highest entity in the CUDA thread hierarchy. In summary, CUDA kernels are executed in a grid of 1 or more blocks, with each block containing the same number of 1 or more threads.

CUDA kernels have access to special variables identifying both the index of the thread (within the block) that is executing the kernel, and, the index of the block (within the grid) that the thread is within. These variables are threadIdx.x and blockIdx.x respectively.

## Accelerating For Loops

For loops in CPU-only applications are ripe for acceleration: rather than run each iteration of the loop serially, each iteration of the loop can be run in parallel in its own thread. Consider the following for loop, and notice, though it is obvious, that it controls how many times the loop will execute, as well as defining what will happen for each iteration of the loop:

int N = 2<<20;

for (int i = 0; i < N; ++i)

{

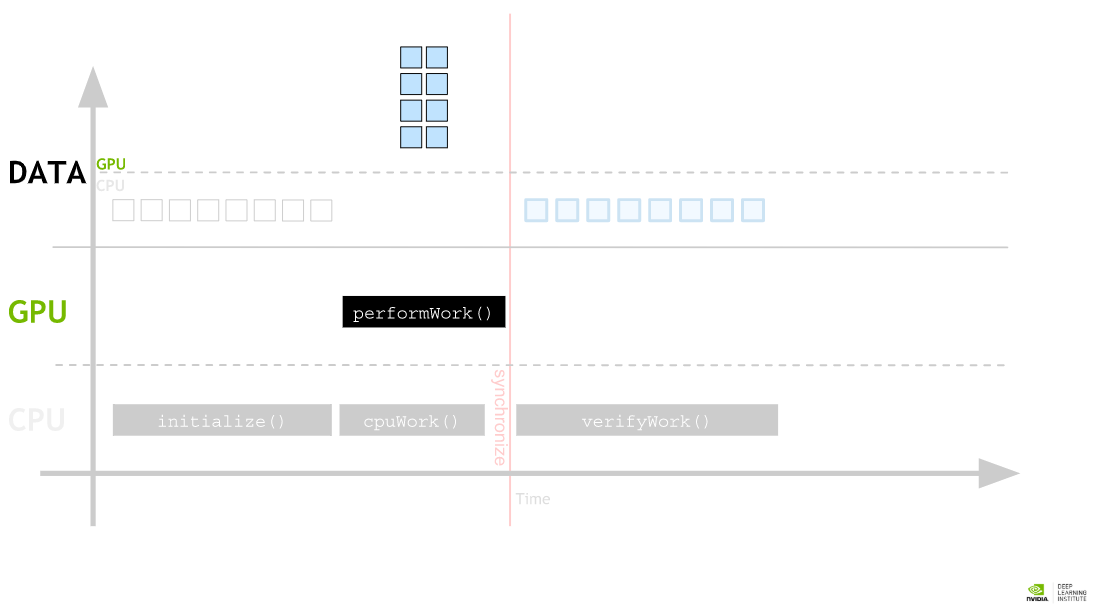
printf("%d\n", i);

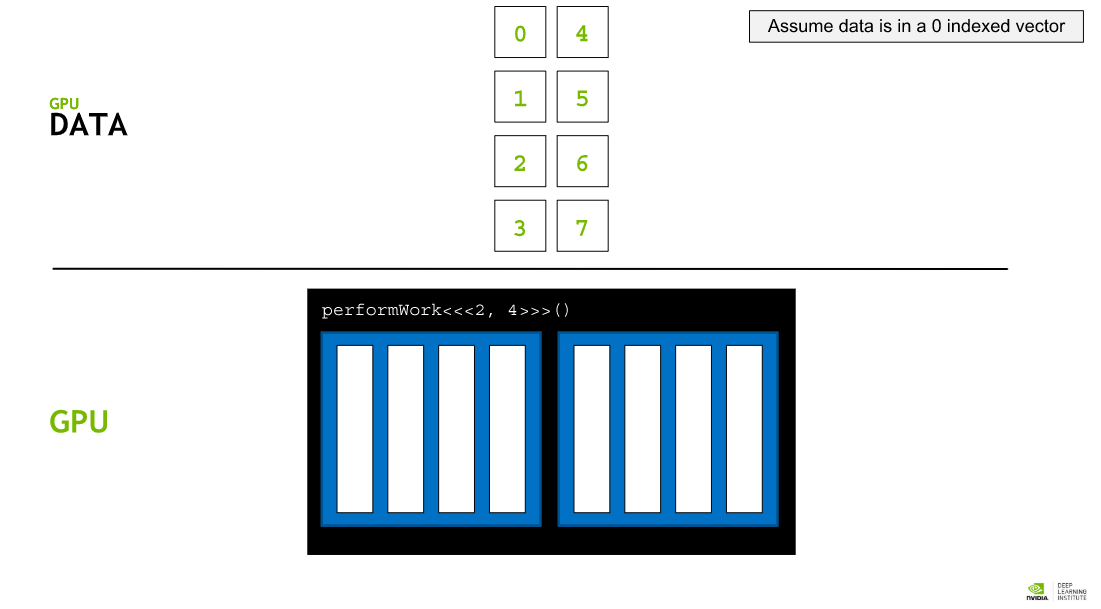
}

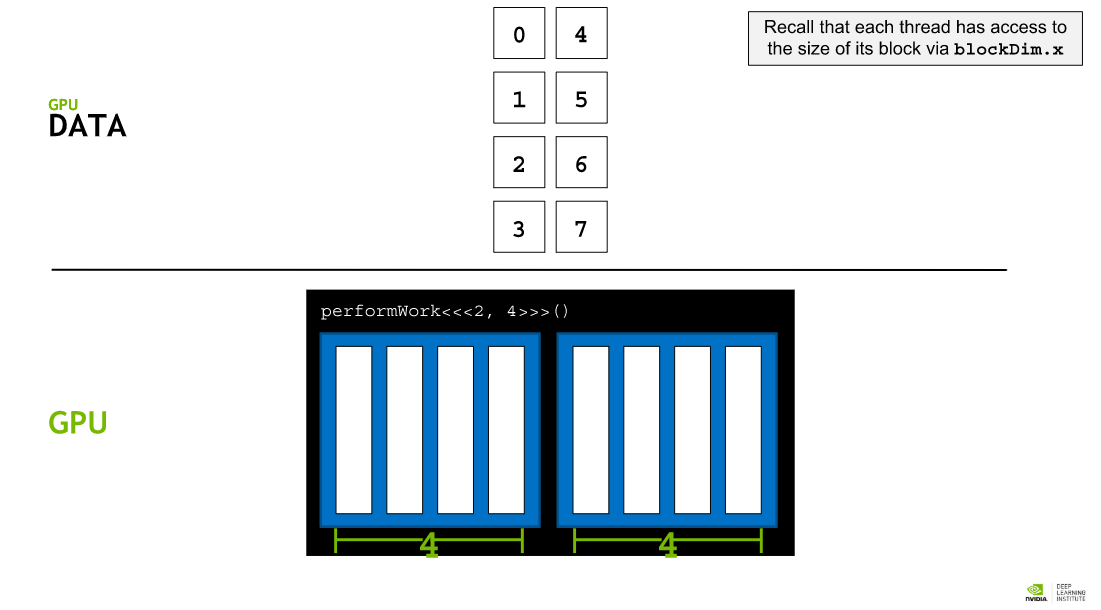
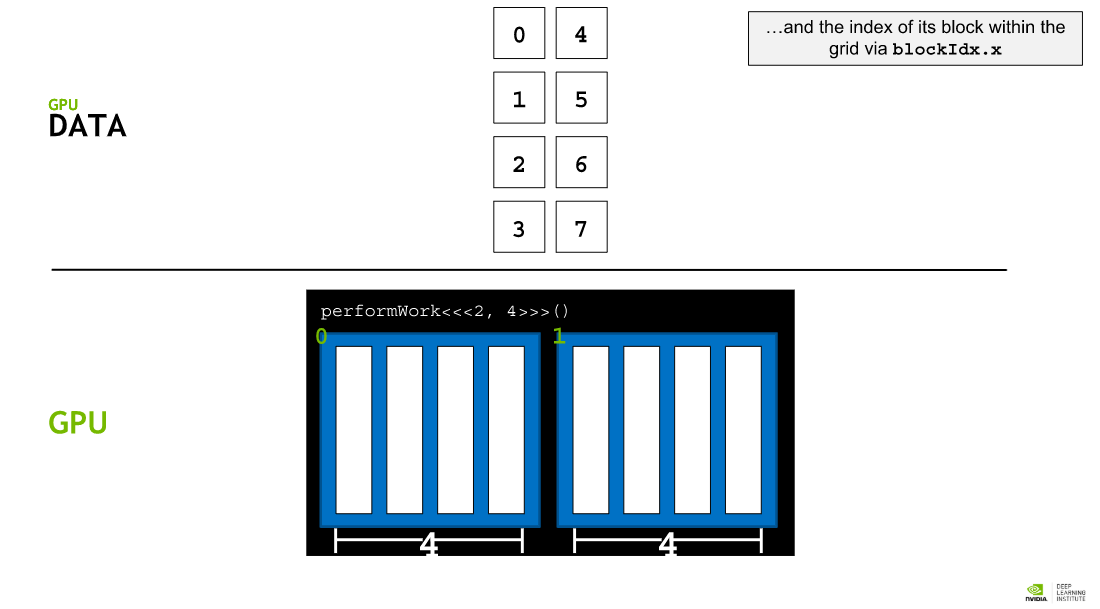
In order to parallelize this loop, 2 steps must be taken:

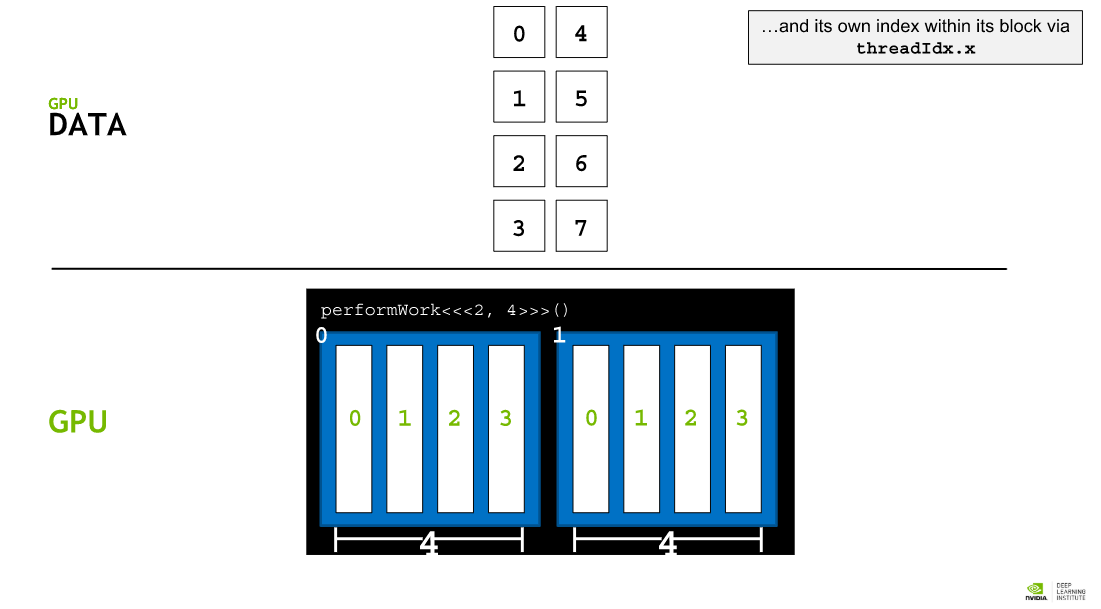
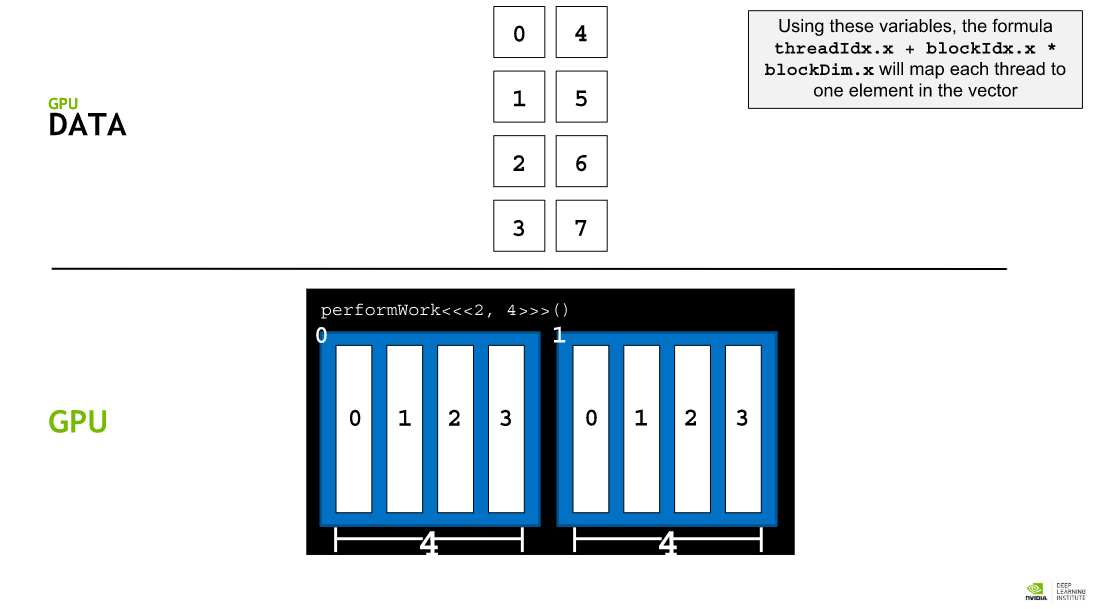
* A kernel must be written to do the work of a **single iteration of the loop**.
* Because the kernel will be agnostic of other running kernels, the execution configuration must be such that the kernel executes the correct number of times, for example, the number of times the loop would have iterated.

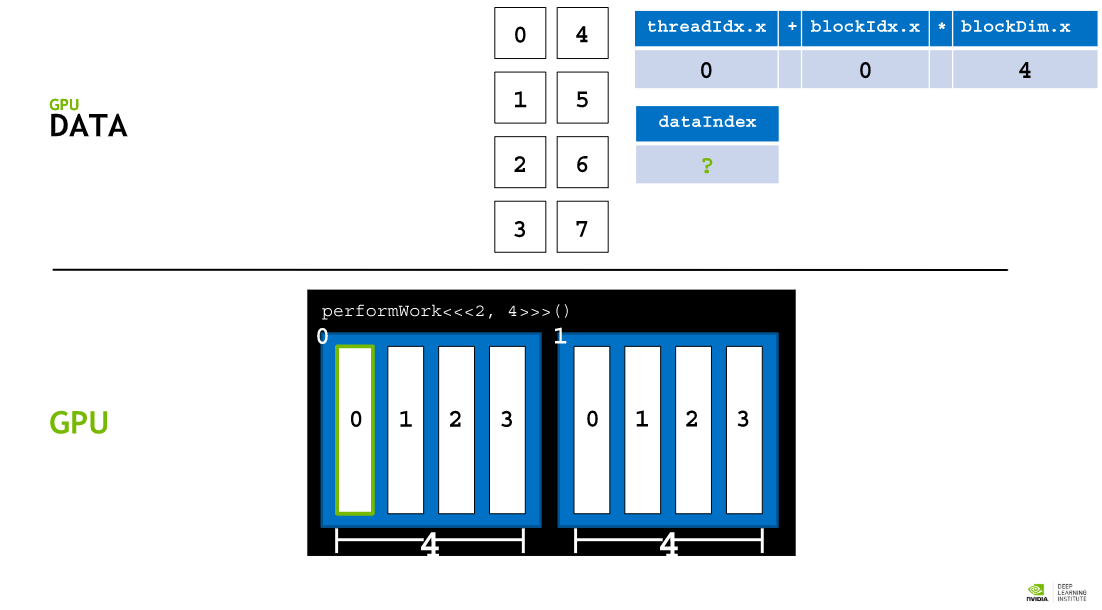
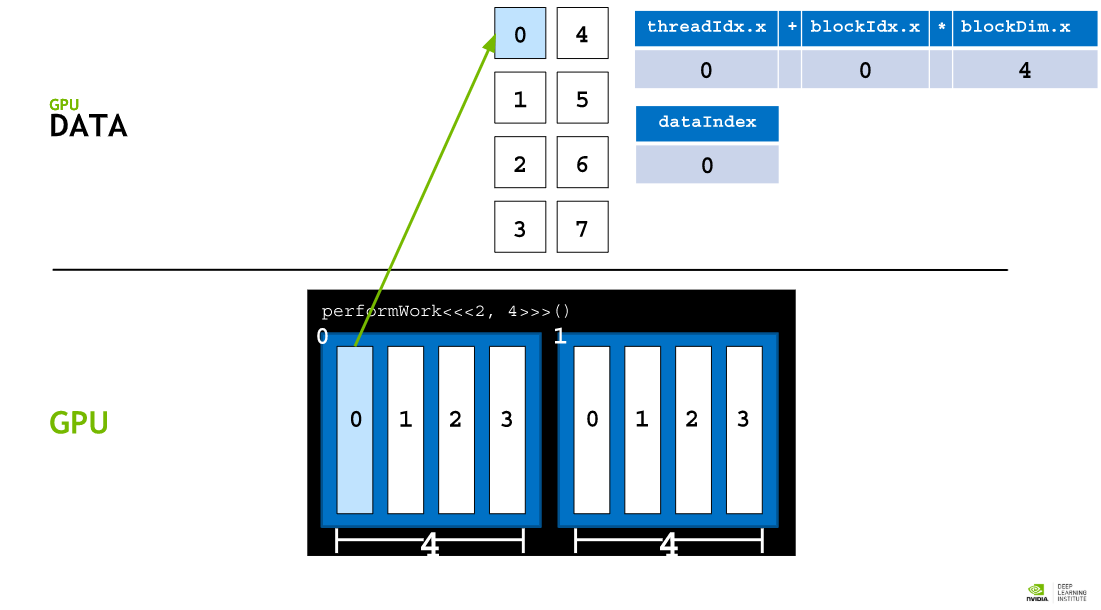
## Coordinating Parallel Threads

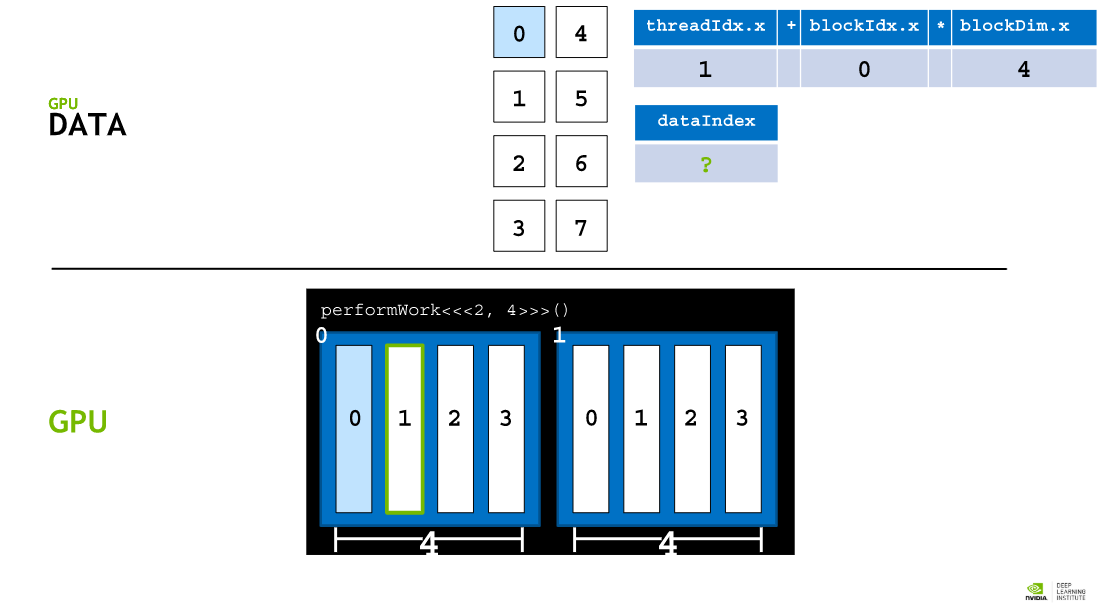
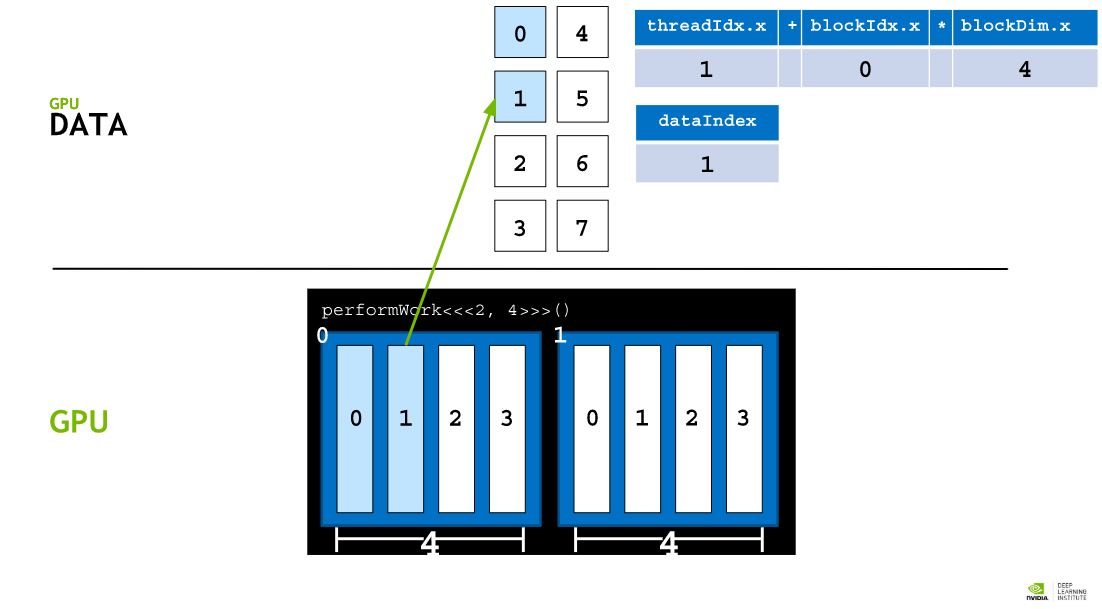
 

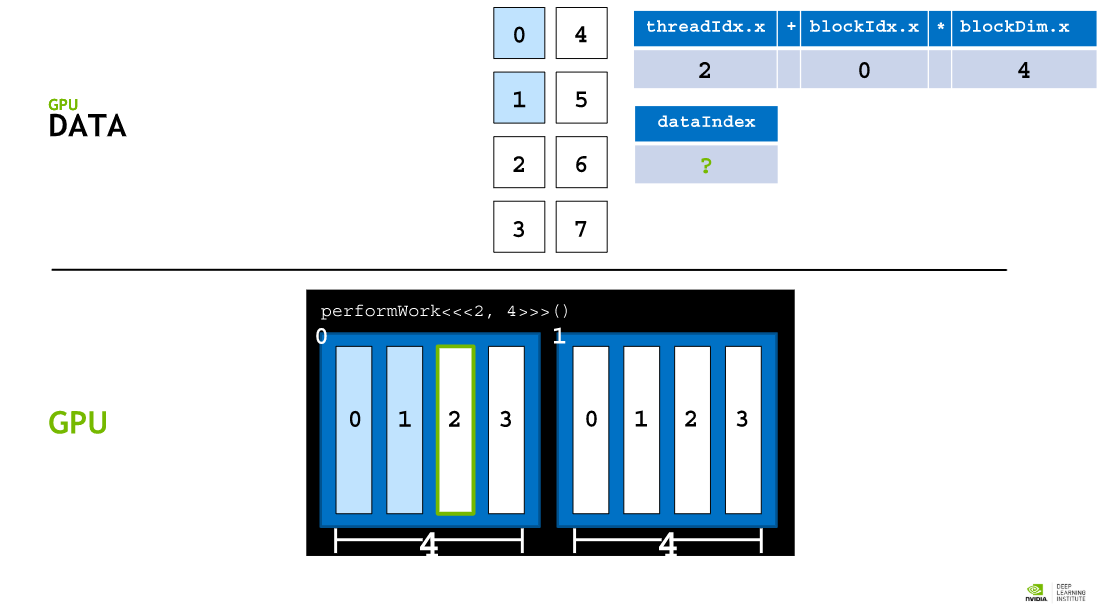
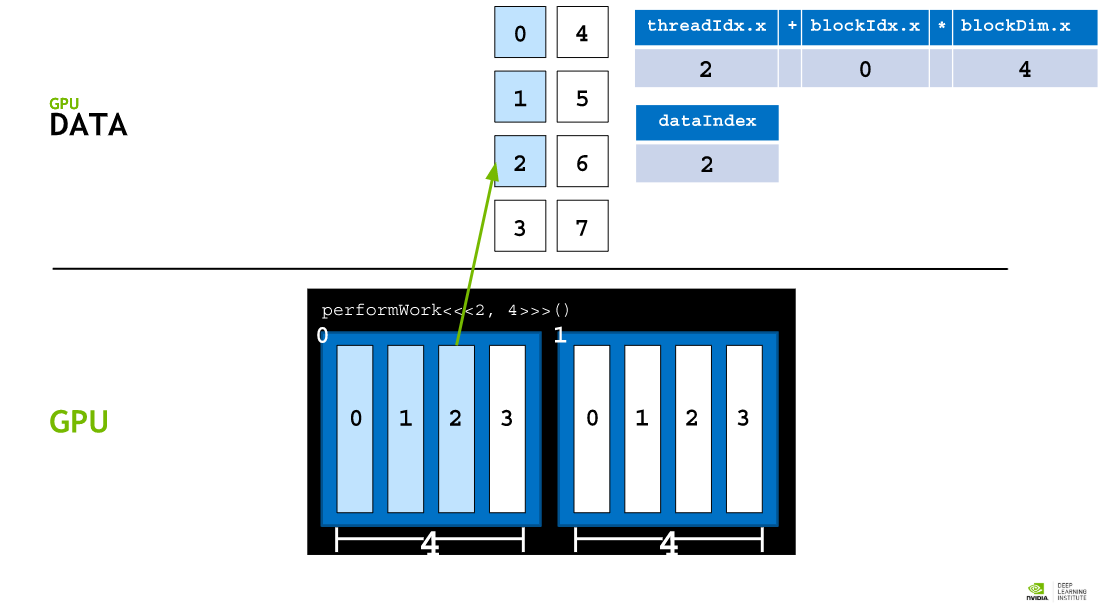
 

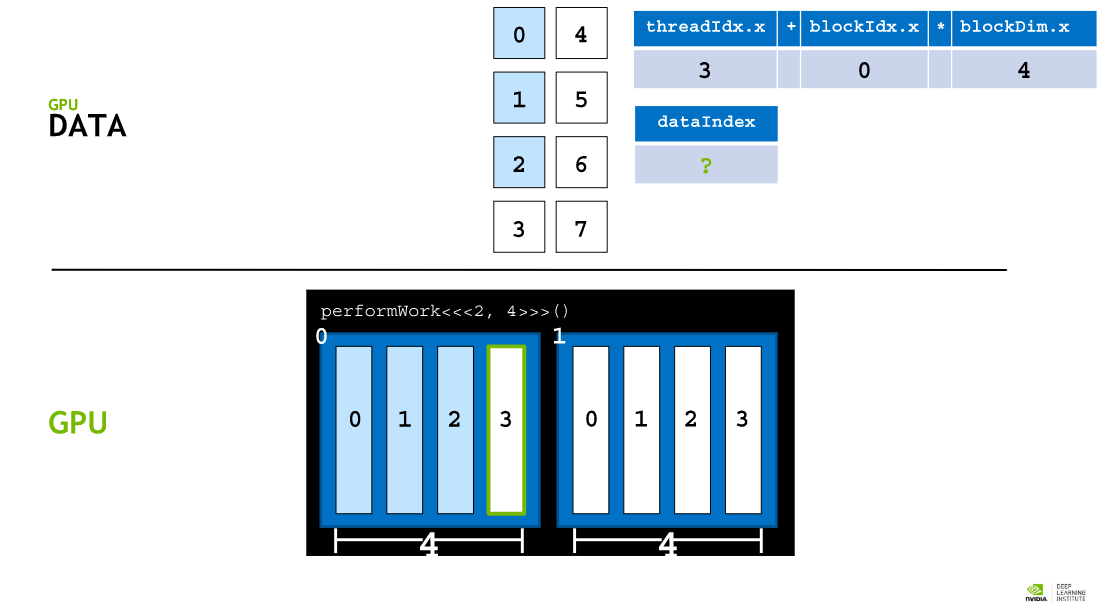
 

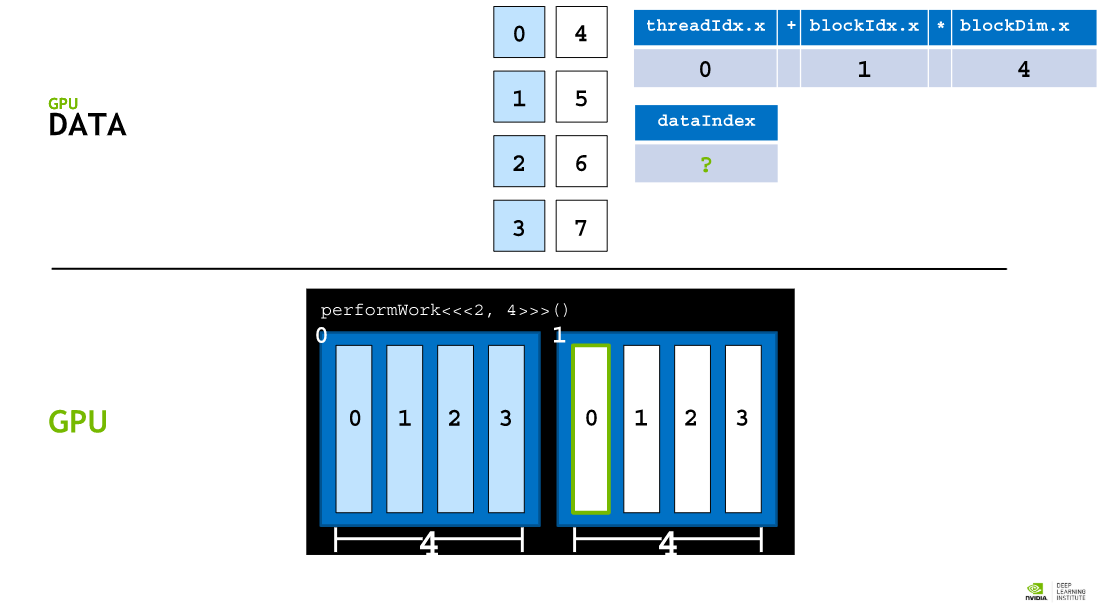
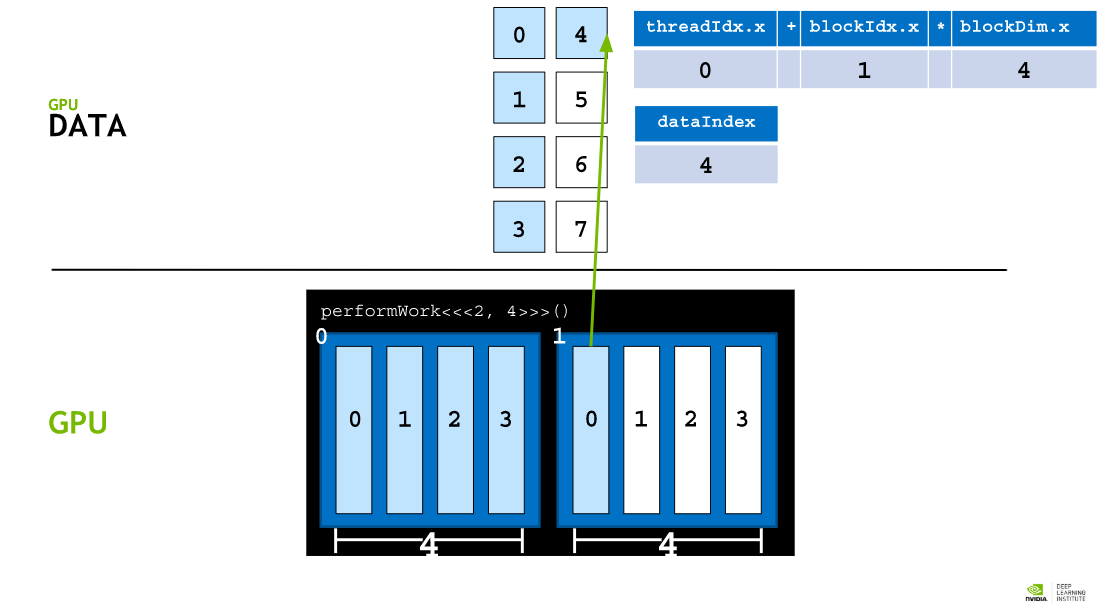
 

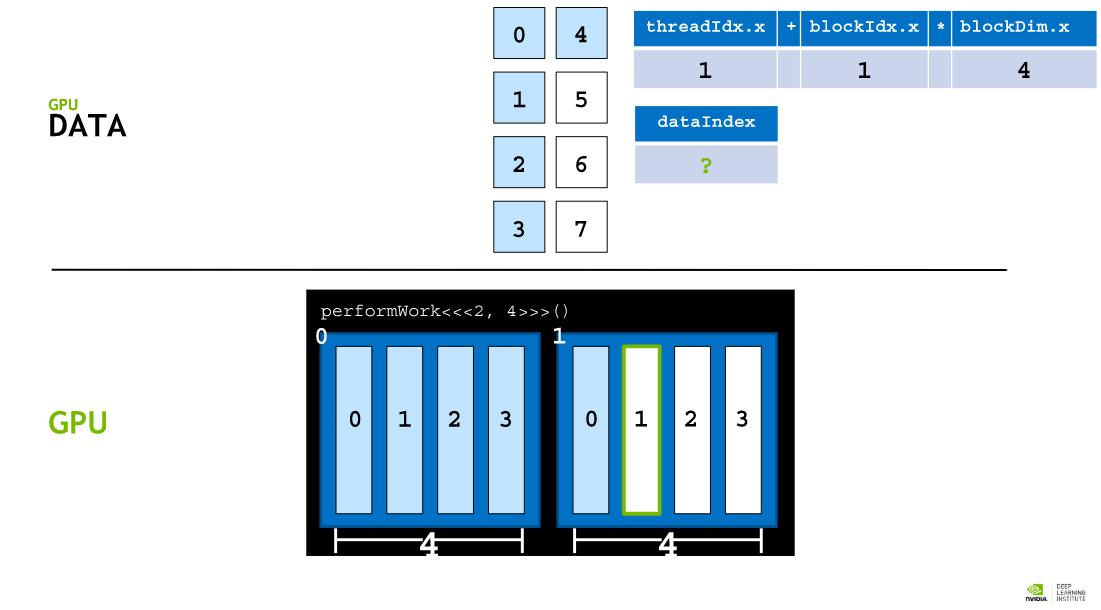
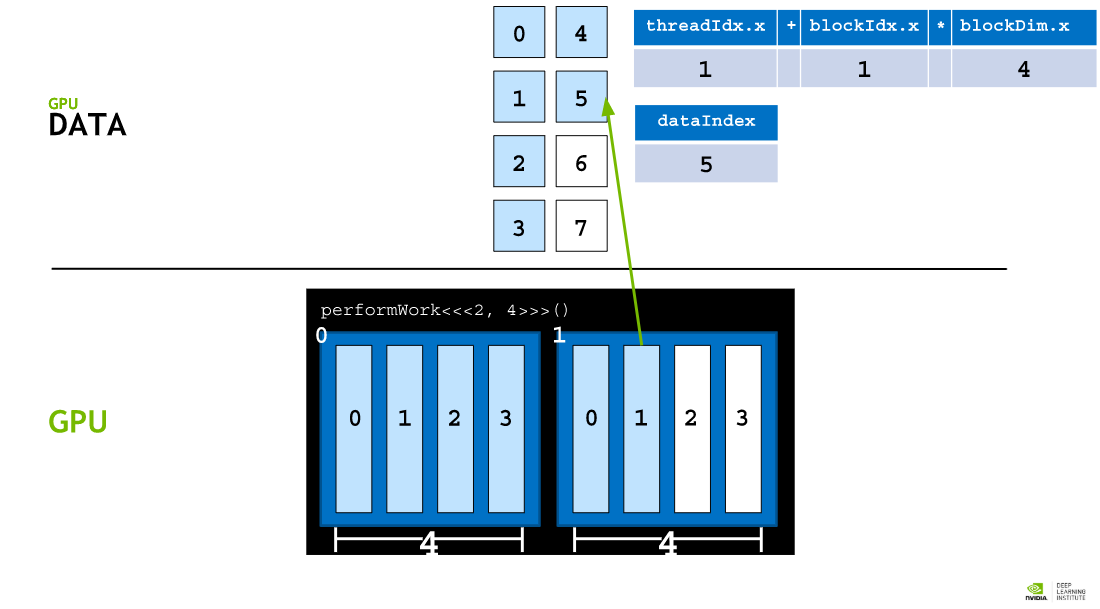
 

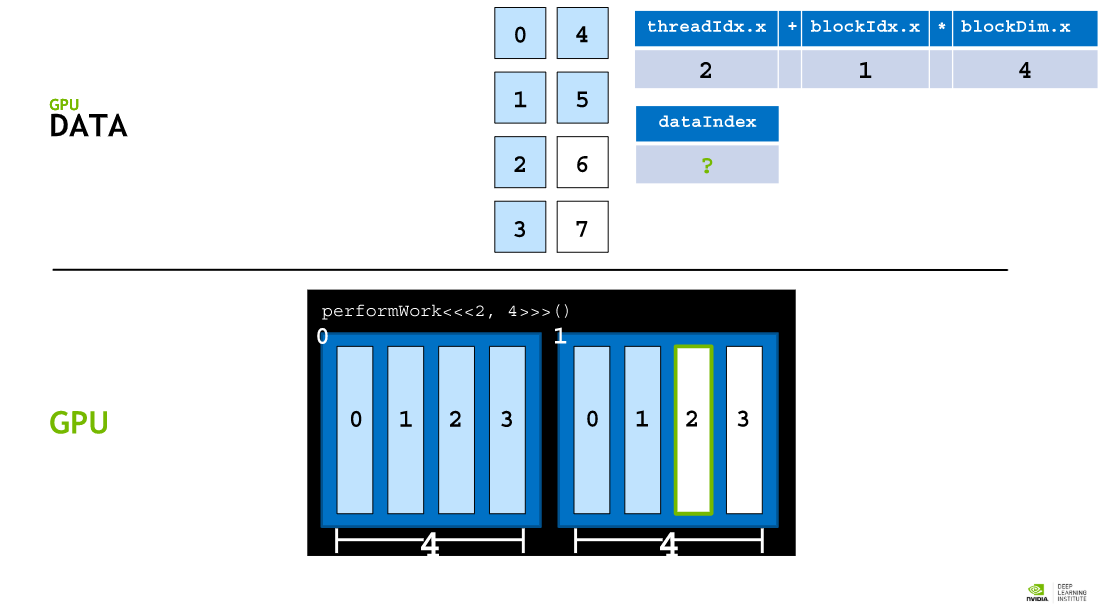
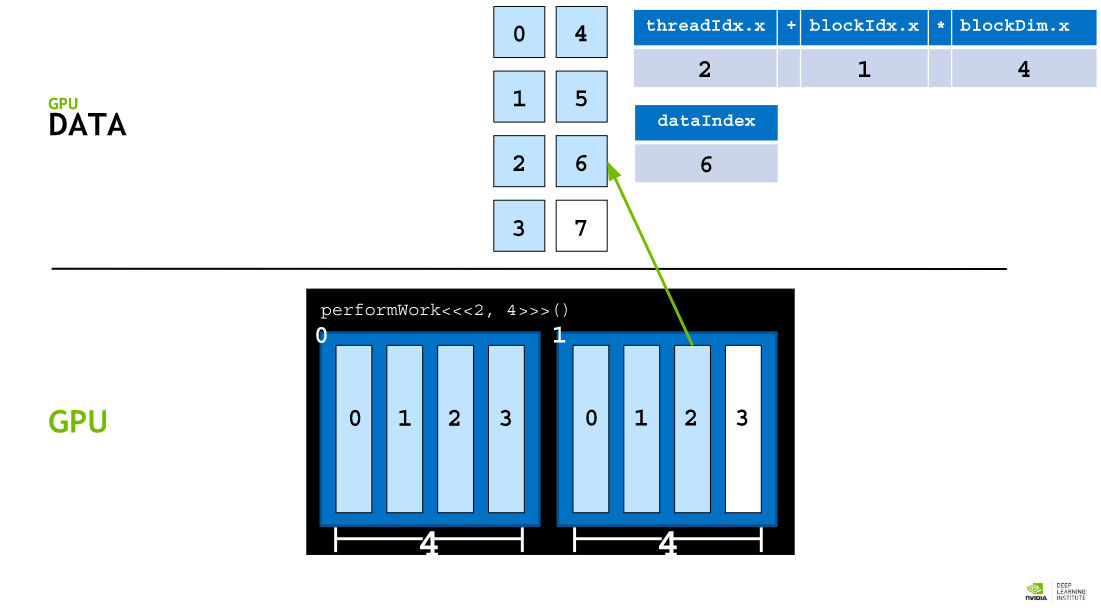
 

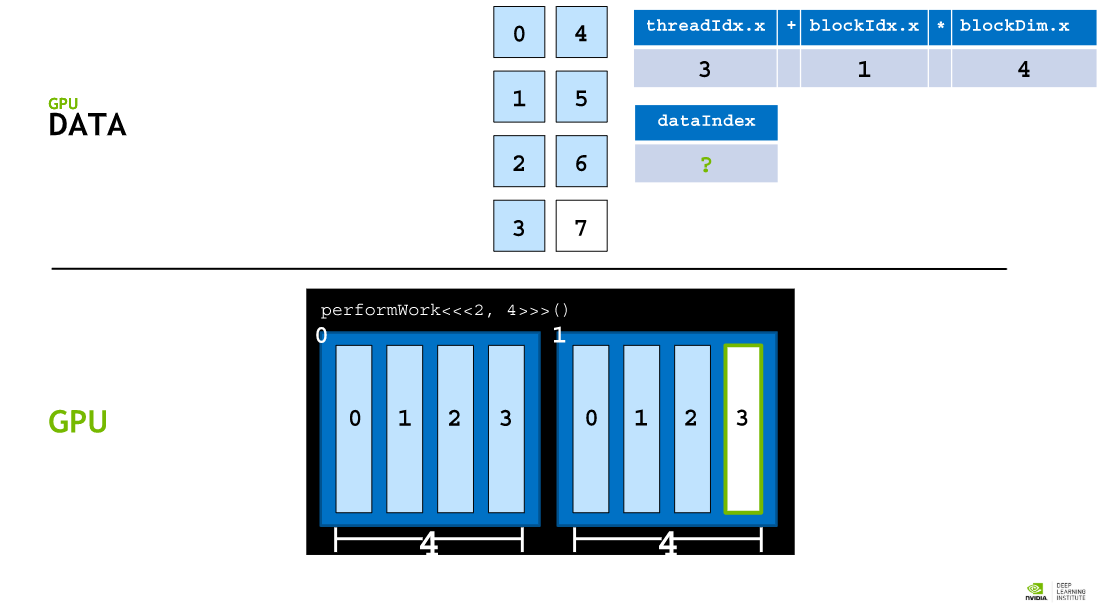
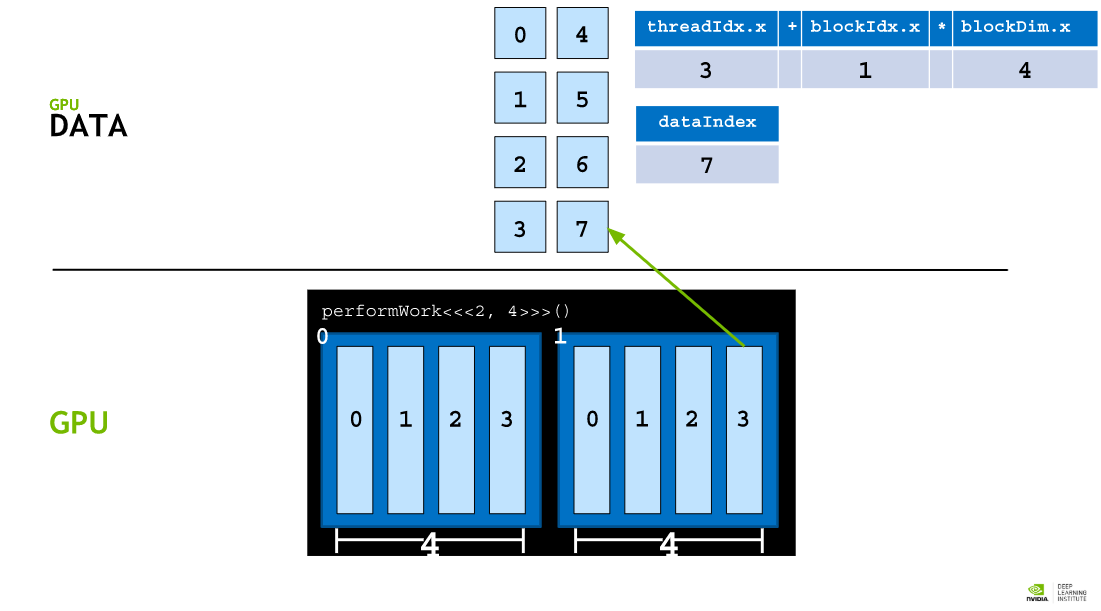
 

## Using Block Dimensions for More Parallelization

There is a limit to the number of threads that can exist in a thread block: 1024 to be precise. In order to increase the amount of parallelism in accelerated applications, we must be able to coordinate among multiple thread blocks.

CUDA Kernels have access to a special variable that gives the number of threads in a block: blockDim.x. Using this variable, in conjunction with blockIdx.x and threadIdx.x, increased parallelization can be accomplished by organizing parallel execution accross multiple blocks of multiple threads with the idiomatic expression threadIdx.x + blockIdx.x \* blockDim.x. Here is a detailed example.

The execution configuration <<<10, 10>>> would launch a grid with a total of 100 threads, contained in 10 blocks of 10 threads. We would therefore hope for each thread to have the ability to calculate some index unique to itself between 0 and 99.

If block blockIdx.x equals 0, then blockIdx.x \* blockDim.x is 0. Adding to 0 the possible threadIdx.x values 0 through 9, then we can generate the indices 0 through 9 within the 100 thread grid.

If block blockIdx.x equals 1, then blockIdx.x \* blockDim.x is 10. Adding to 10 the possible threadIdx.x values 0 through 9, then we can generate the indices 10 through 19 within the 100 thread grid.

If block blockIdx.x equals 5, then blockIdx.x \* blockDim.x is 50. Adding to 50 the possible threadIdx.x values 0 through 9, then we can generate the indices 50 through 59 within the 100 thread grid.

If block blockIdx.x equals 9, then blockIdx.x \* blockDim.x is 90. Adding to 90 the possible threadIdx.x values 0 through 9, then we can generate the indices 90 through 99 within the 100 thread grid.

## Allocating Memory to be accessed on the GPU and the CPU

More recent versions of CUDA (version 6 and later) have made it easy to allocate memory that is available to both the CPU host and any number of GPU devices, and while there are many [intermediate and advanced techniques](http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations) for memory management that will support the most optimal performance in accelerated applications, the most basic CUDA memory management technique we will now cover supports fantastic performance gains over CPU-only applications with almost no developer overhead.

To allocate and free memory, and obtain a pointer that can be referenced in both host and device code, replace calls to malloc and free with cudaMallocManaged and cudaFree as in the following example:

// CPU-only

int N = 2<<20;

size\_t size = N \* sizeof(int);

int \*a;

a = (int \*)malloc(size);

// Use `a` in CPU-only program.

free(a);

// Accelerated

int N = 2<<20;

size\_t size = N \* sizeof(int);

int \*a;

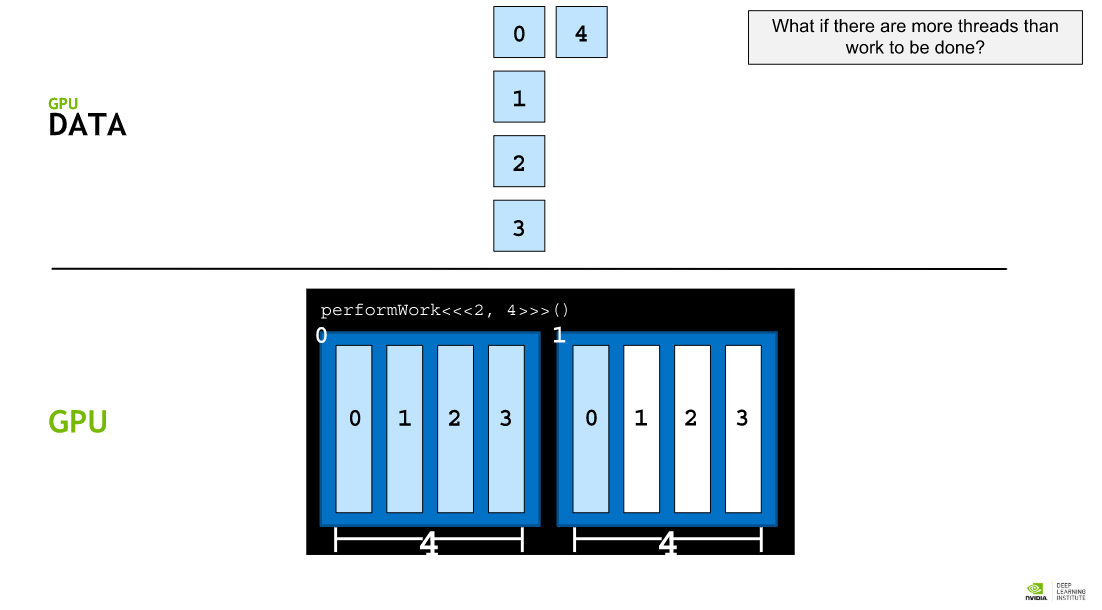
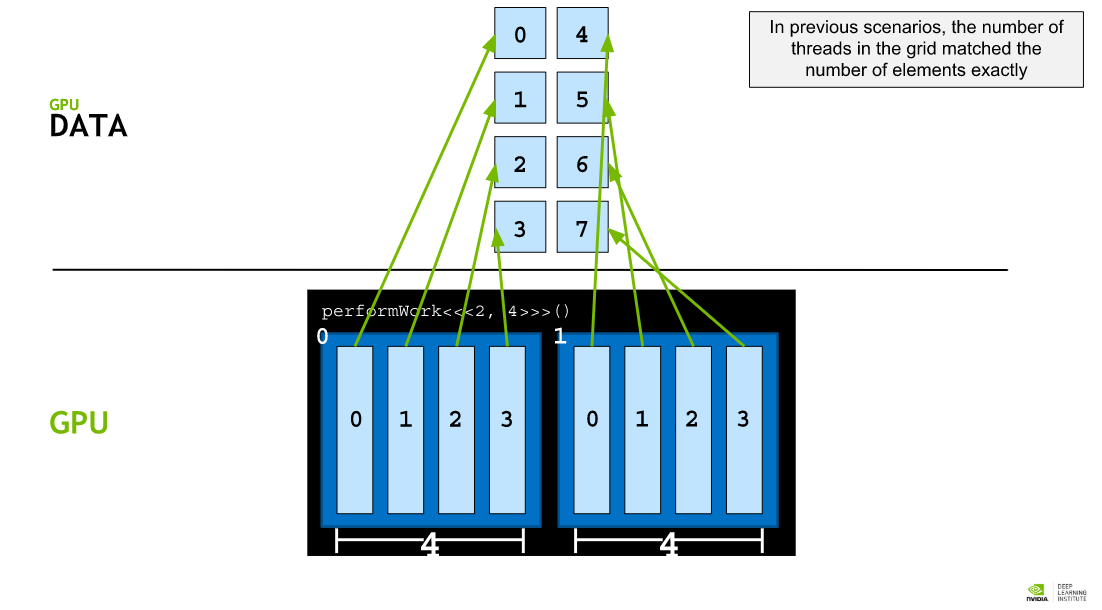
// Note the address of `a` is passed as first argument.

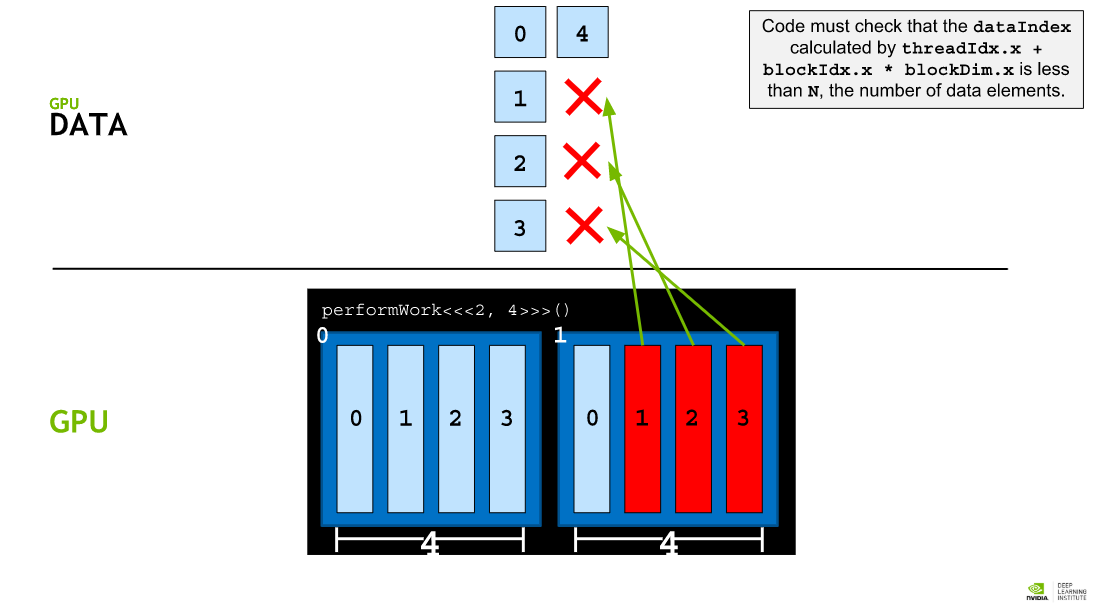
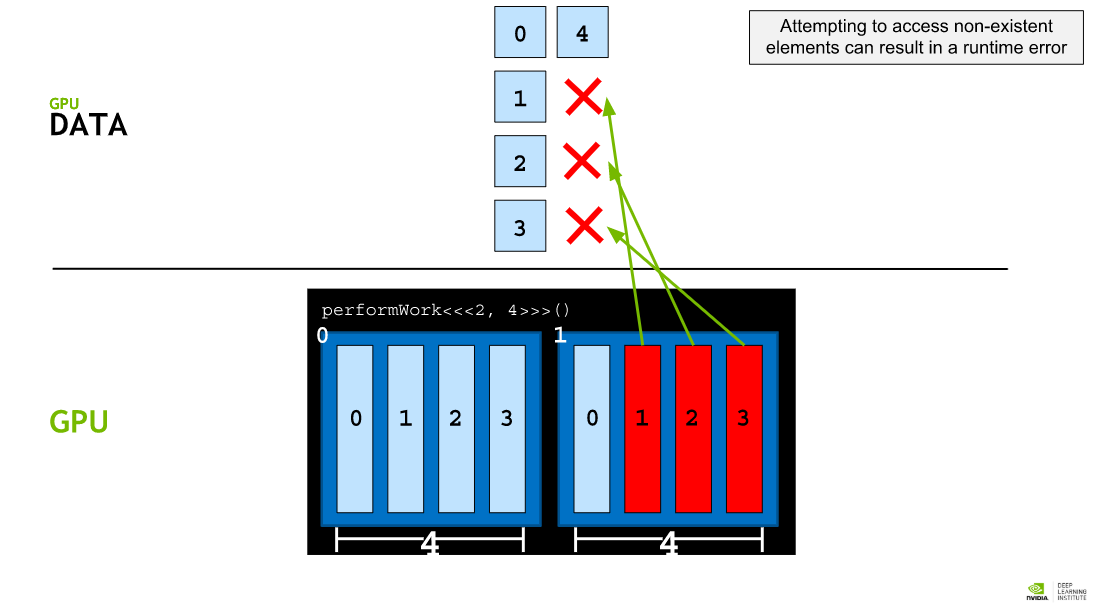
cudaMallocManaged(&a, size);

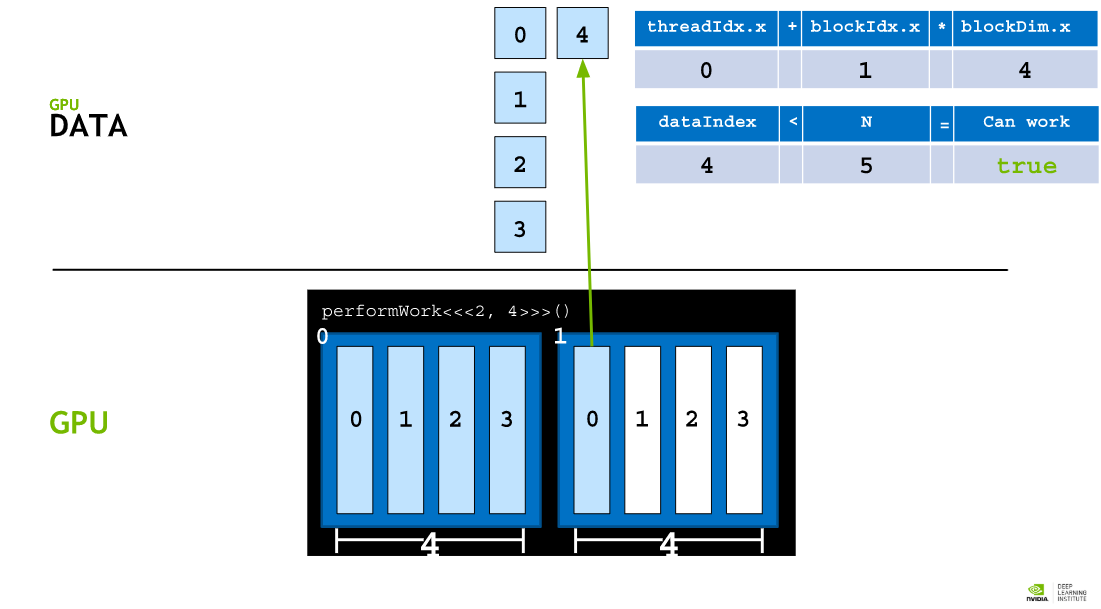
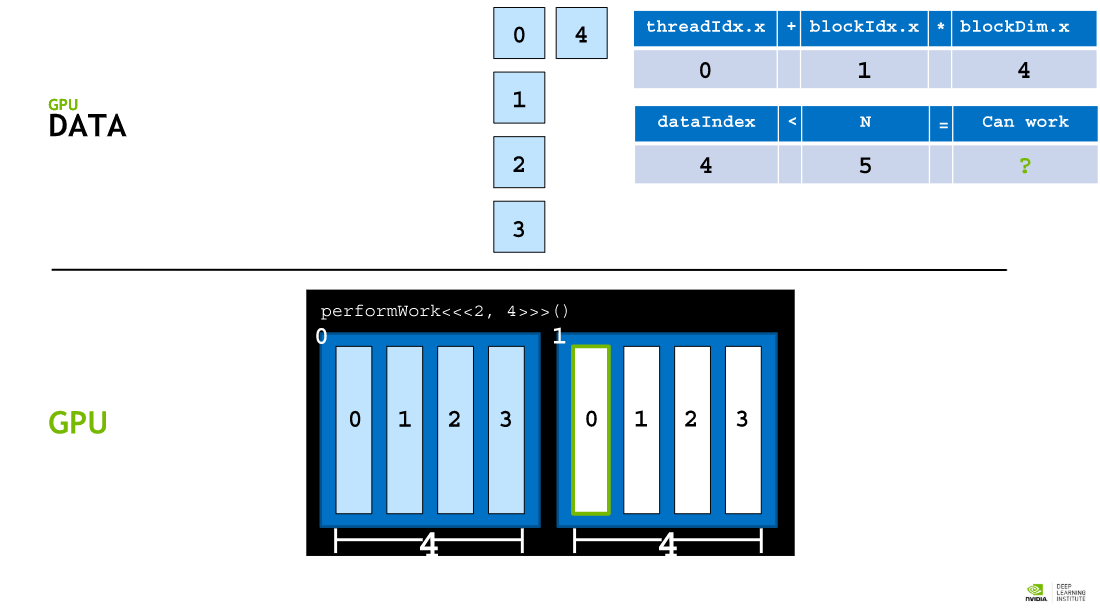
// Use `a` on the CPU and/or on any GPU in the acceleratedsystem.

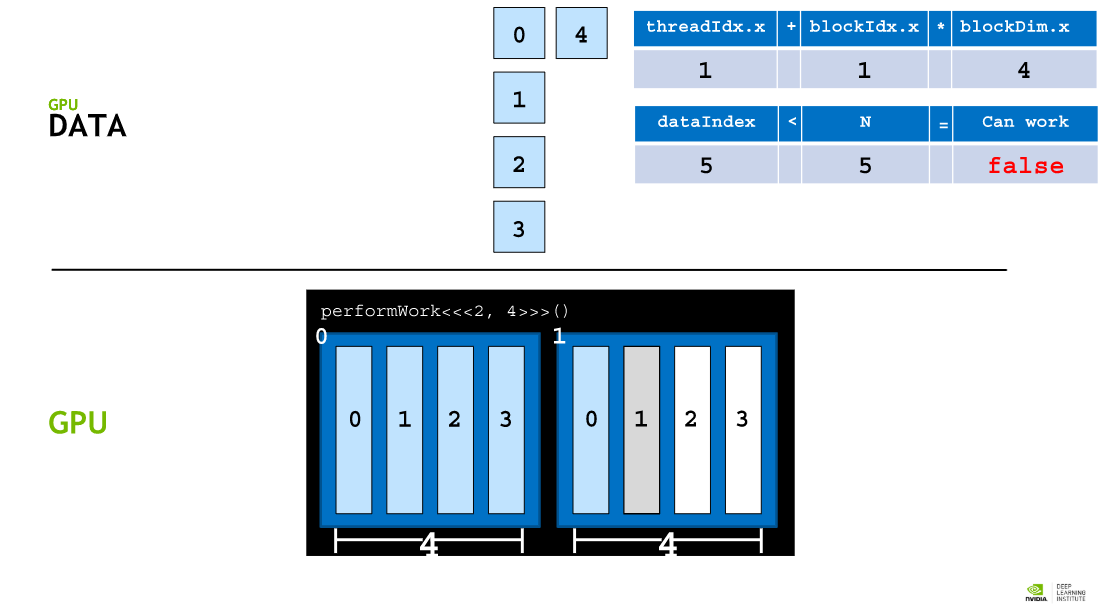
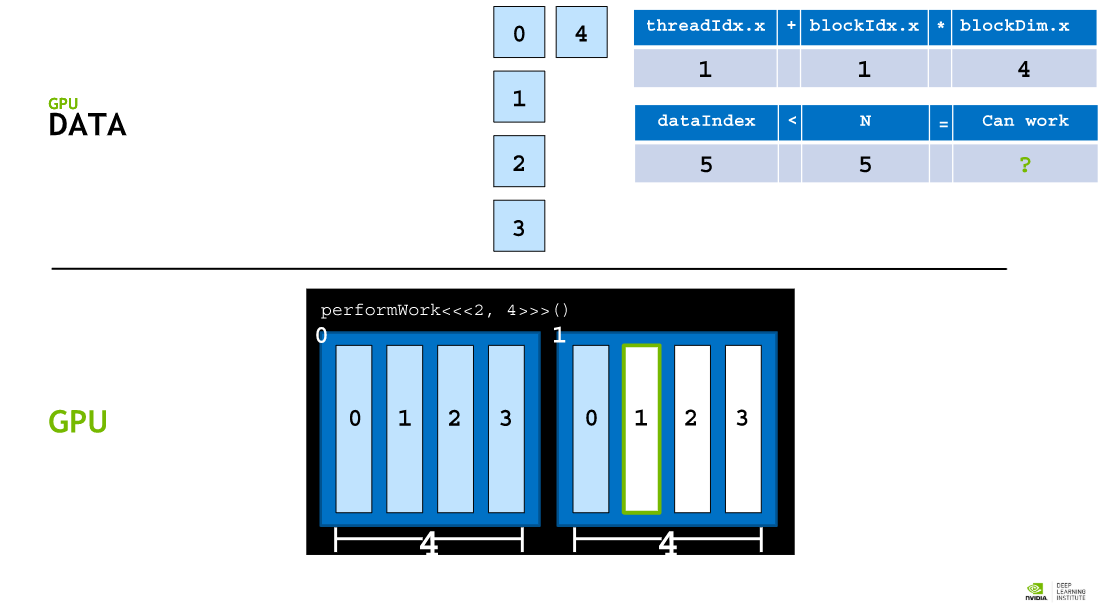
cudaFree(a);

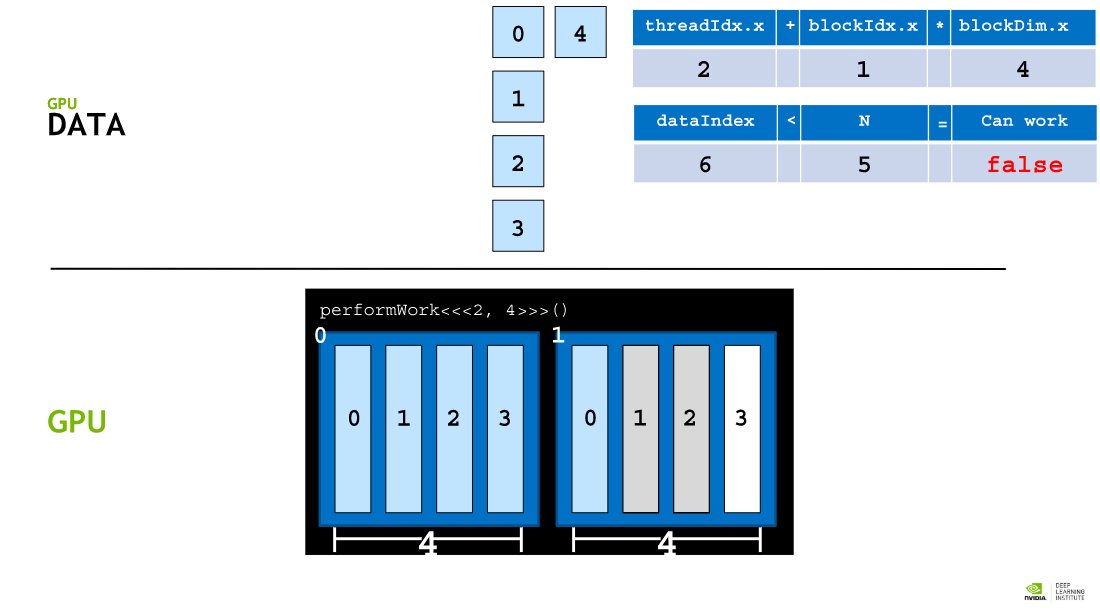
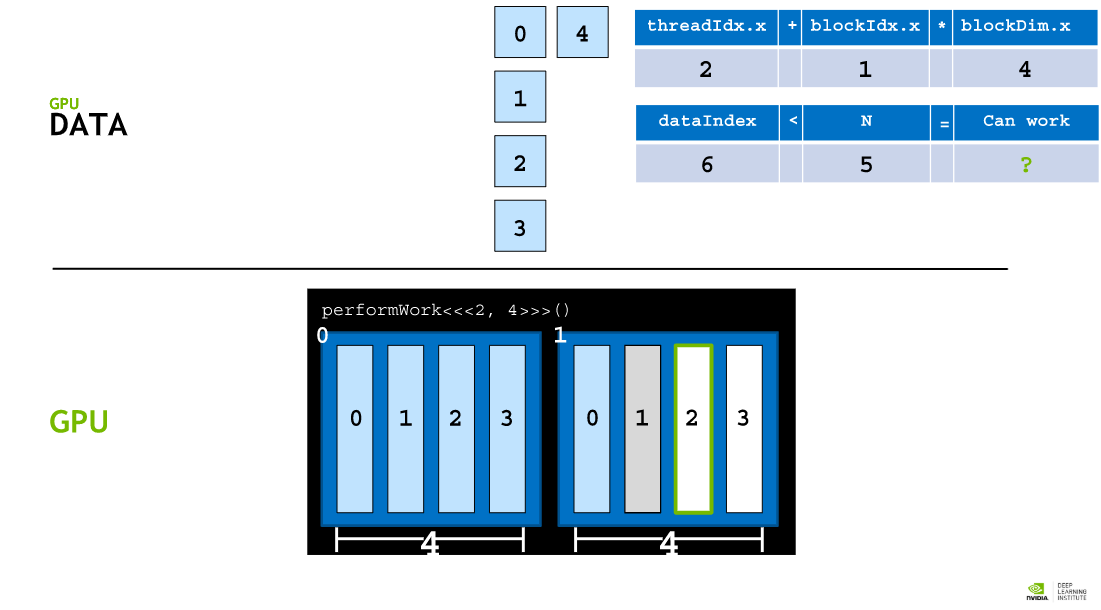
## Grid Size Work Amount Mismatch

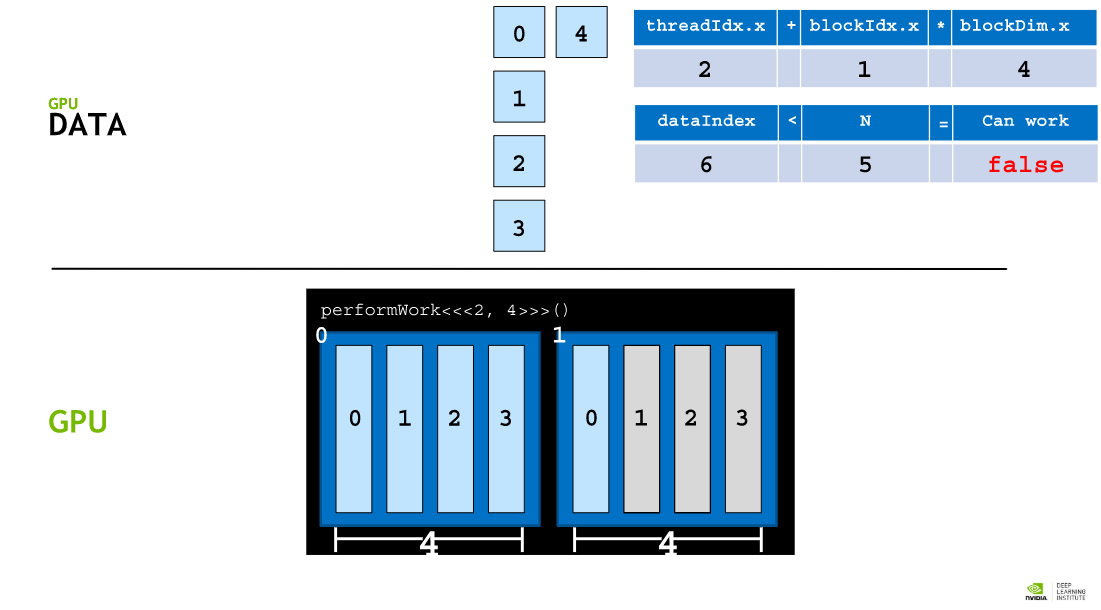
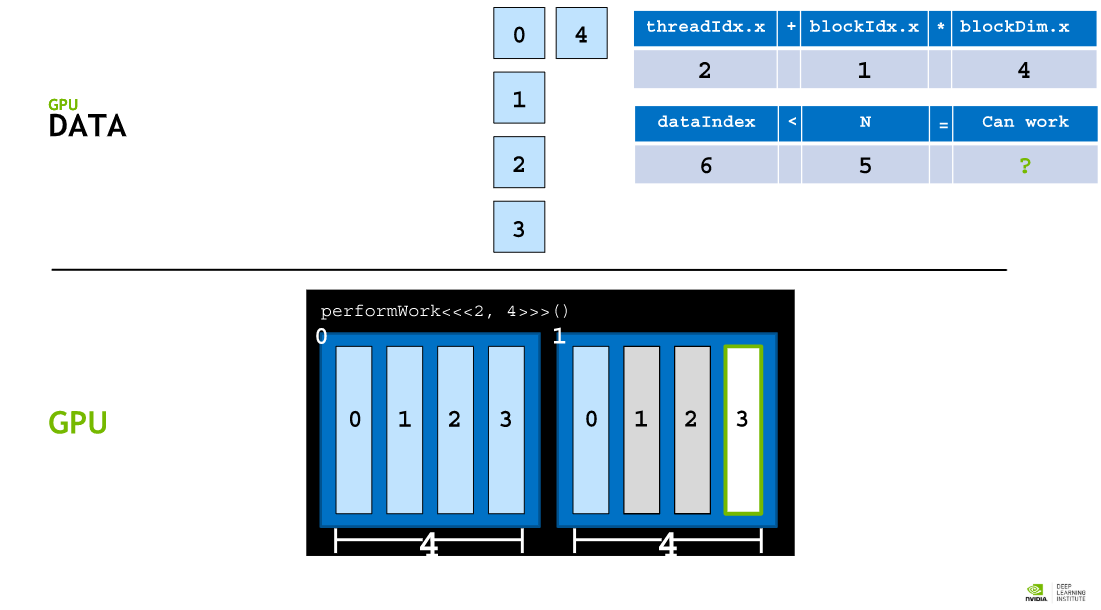












## Handling Block Configuration Mismatches to Number of Needed Threads

It may be the case that an execution configuration cannot be expressed that will create the exact number of threads needed for parallelizing a loop.

A common example has to do with the desire to choose optimal block sizes. For example, due to GPU hardware traits, blocks that contain a number of threads that are a multiple of 32 are often desirable for performance benefits. Assuming that we wanted to launch blocks each containing 256 threads (a multiple of 32), and needed to run 1000 parallel tasks (a trivially small number for ease of explanation), then there is no number of blocks that would produce an exact total of 1000 threads in the grid, since there is no integer value 32 can be multiplied by to equal exactly 1000.

This scenario can be easily addressed in the following way:

* Write an execution configuration that creates more threads than necessary to perform the allotted work.
* Pass a value as an argument into the kernel (N) that represents to the total size of the data set to be processed, or the total threads that are needed to complete the work.
* After calculating the thread's index within the grid (using tid+bid\*bdim), check that this index does not exceed N, and only perform the pertinent work of the kernel if it does not.

Here is an example of an idiomatic way to write an execution configuration when both N and the number of threads in a block are known, and an exact match between the number of threads in the grid and N cannot be guaranteed. It ensures that there are always at least as many threads as needed for N, and only 1 additional block's worth of threads extra, at most:

// Assume `N` is known

int N = 100000;

// Assume we have a desire to set `threads\_per\_block` exactly to `256`

size\_t threads\_per\_block = 256;

// Ensure there are at least `N` threads in the grid, but only 1 block's worth extra

size\_t number\_of\_blocks = (N + threads\_per\_block - 1) / threads\_per\_block;

some\_kernel<<<number\_of\_blocks, threads\_per\_block>>>(N);

Becuase the execution configuration above results in more threads in the grid than N, care will need to be taken inside of the some\_kernel definition so that some\_kernel does not attempt to access out of range data elements, when being executed by one of the "extra" threads:

\_\_global\_\_ some\_kernel(int N)

{

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

if (idx < N) // Check to make sure `idx` maps to some value within `N`

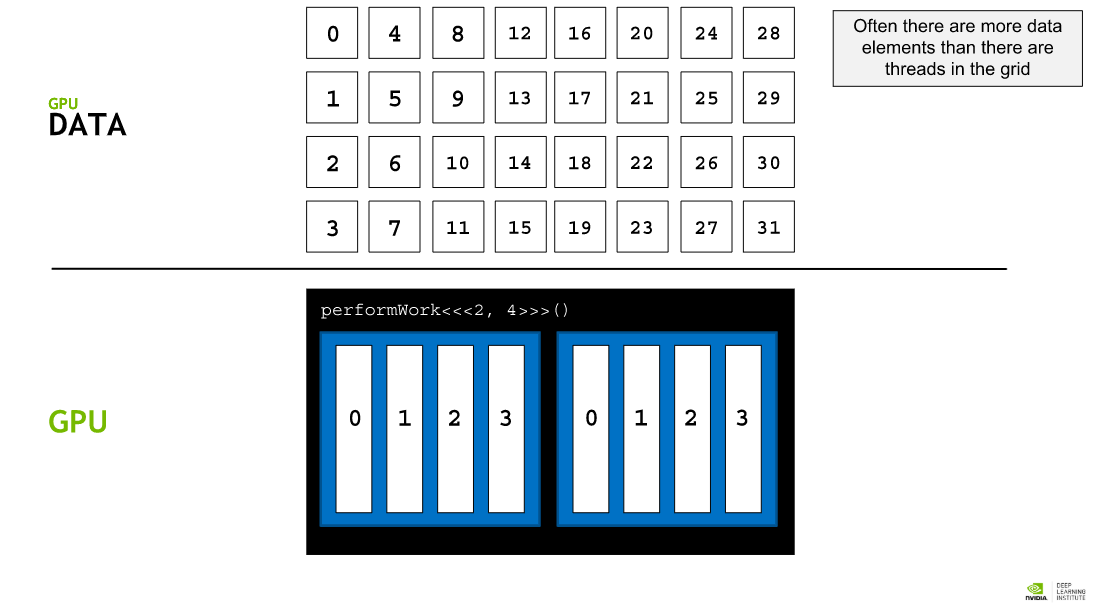
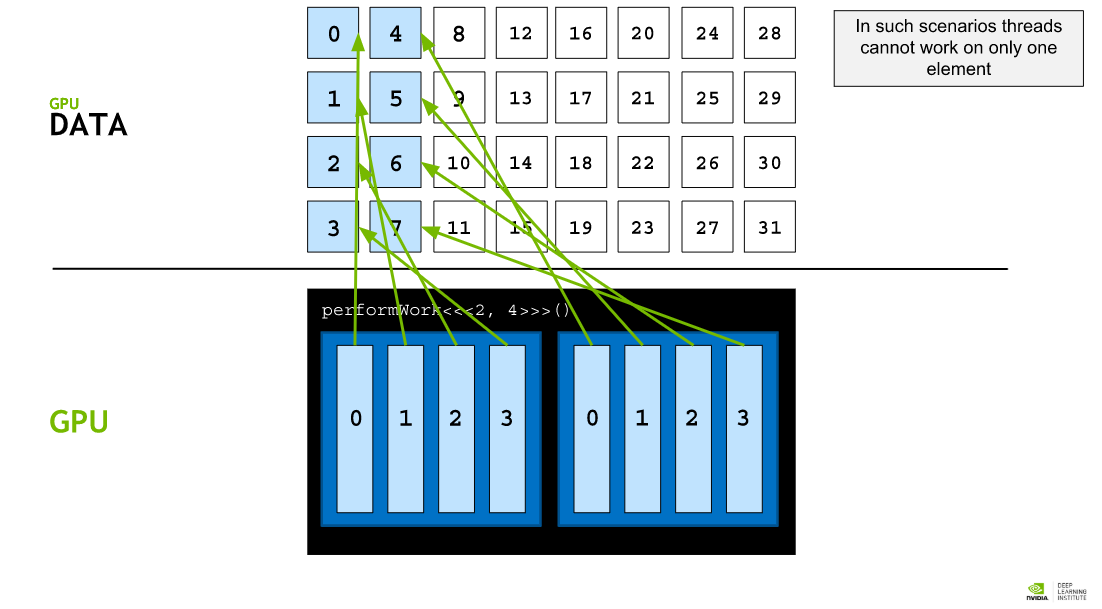
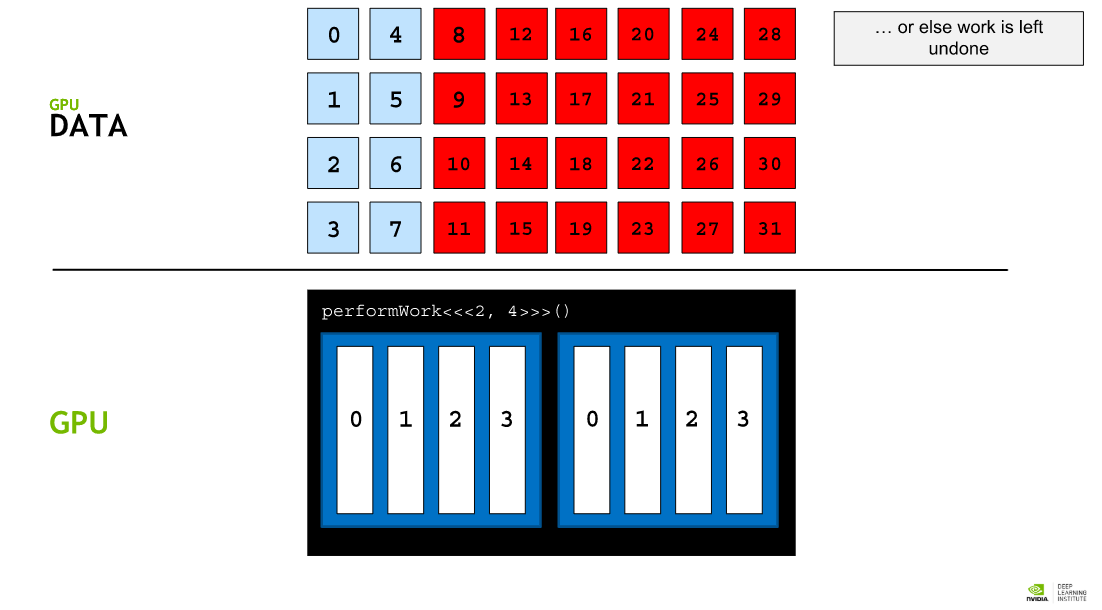
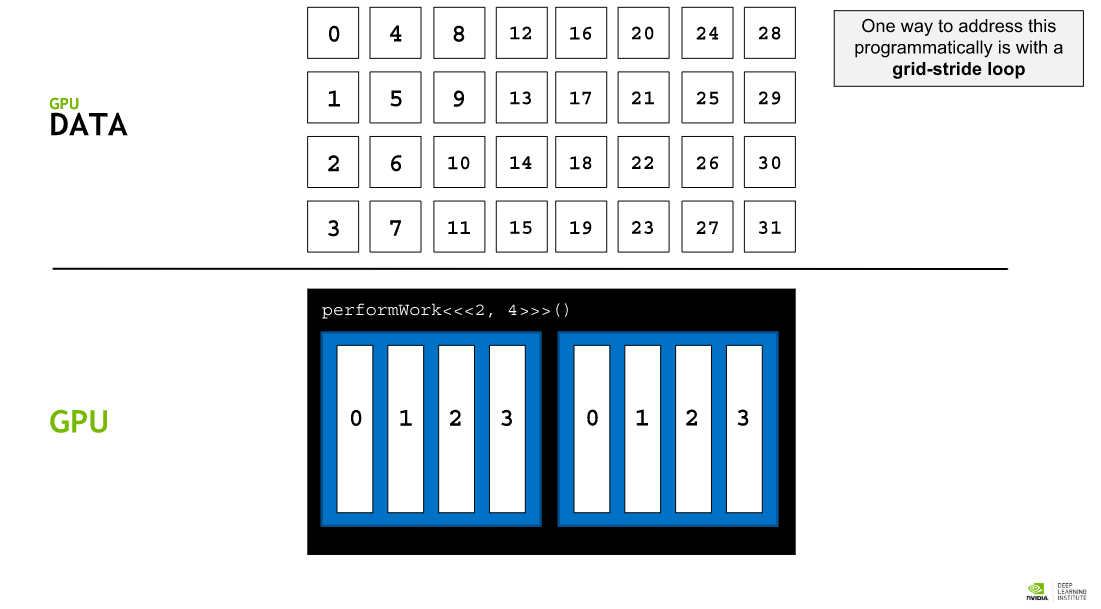
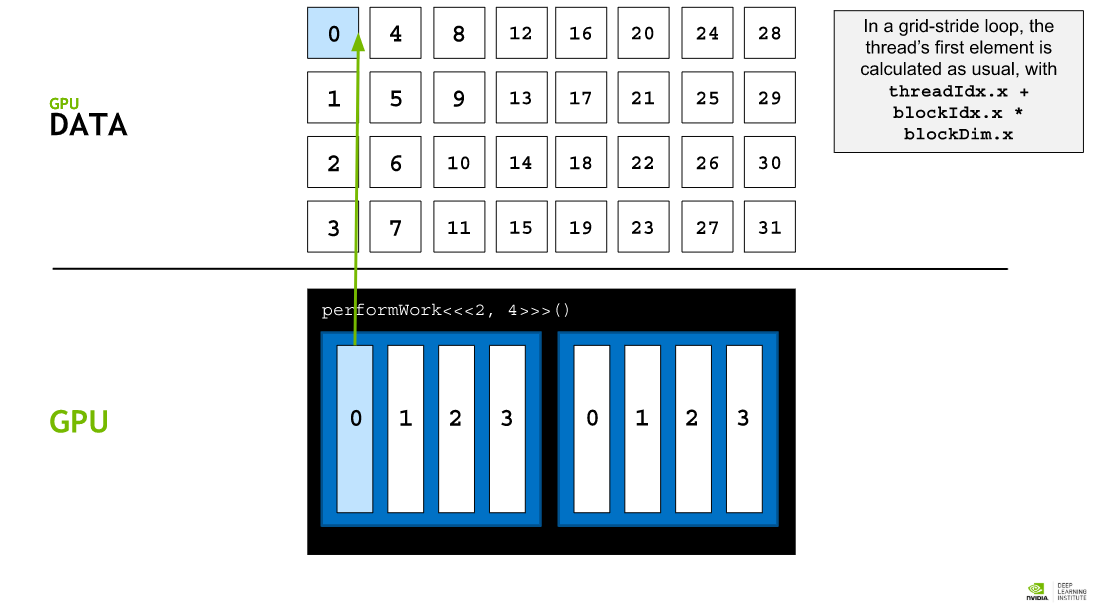
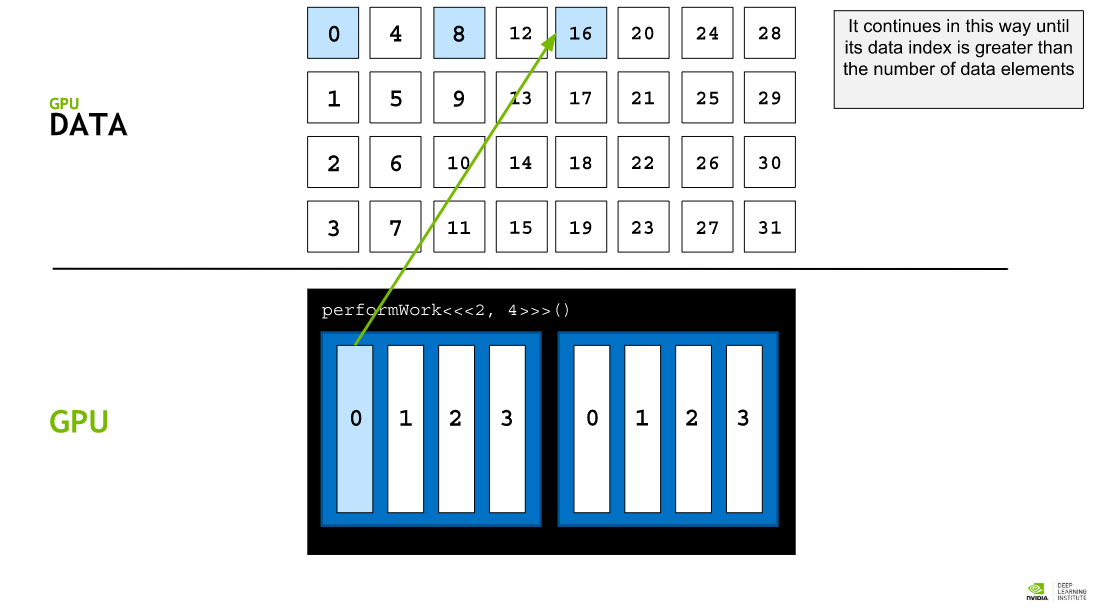
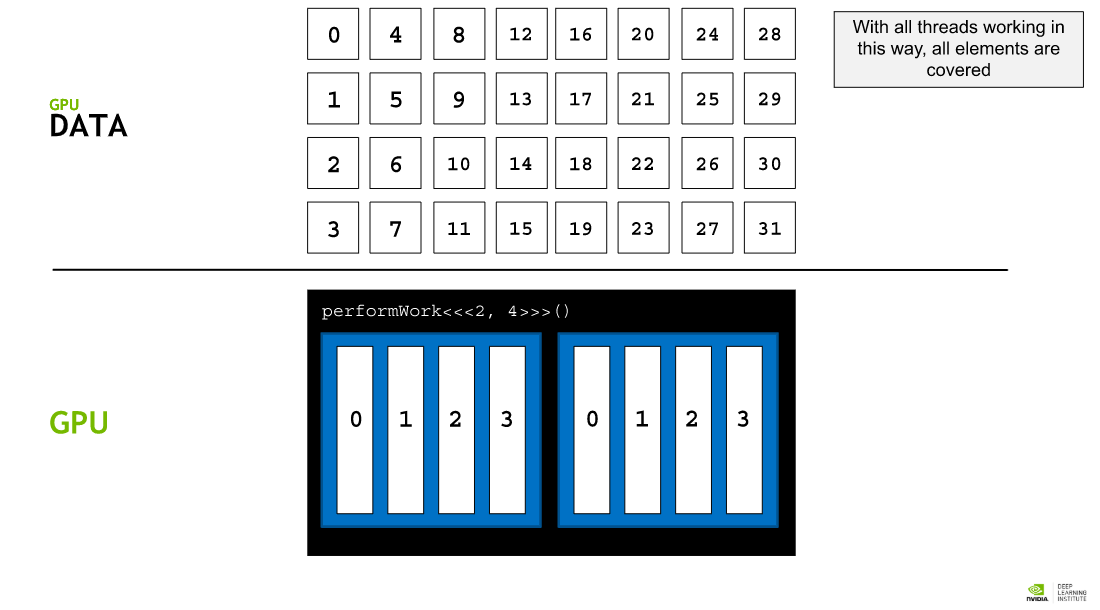
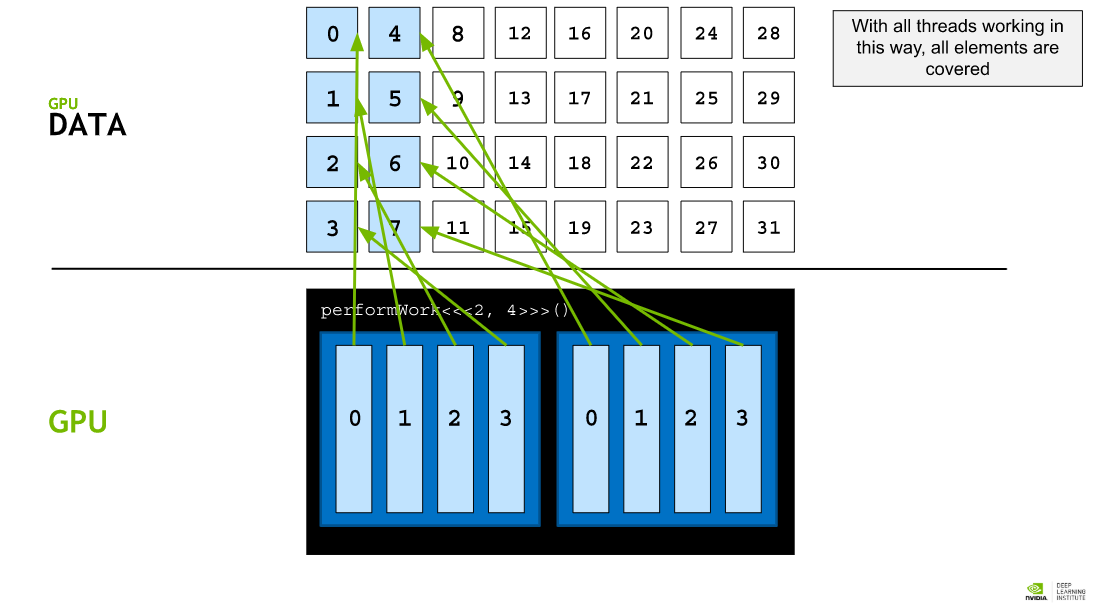
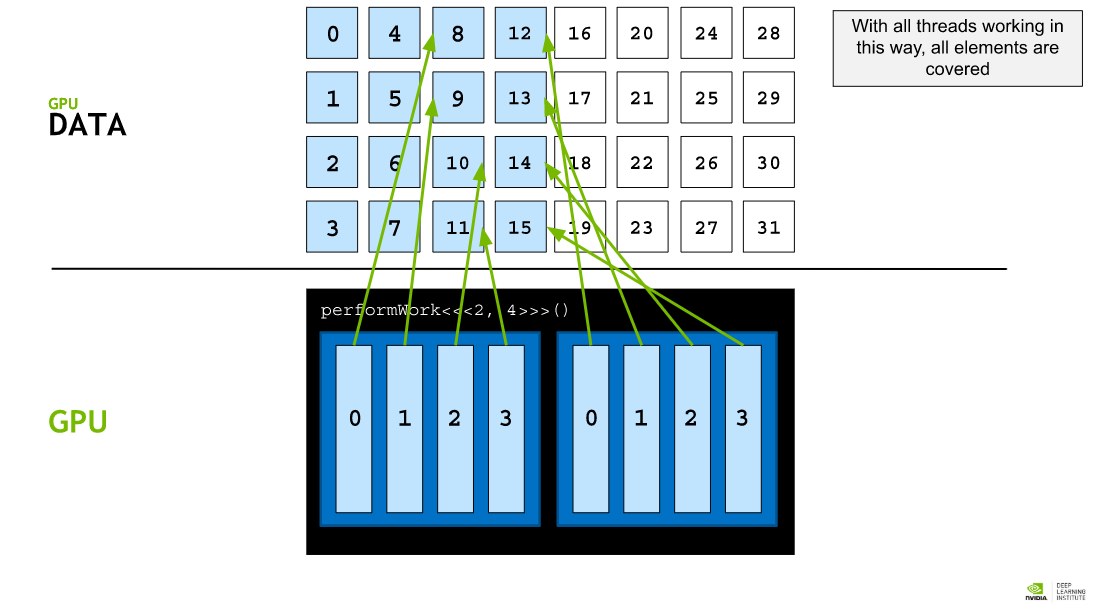
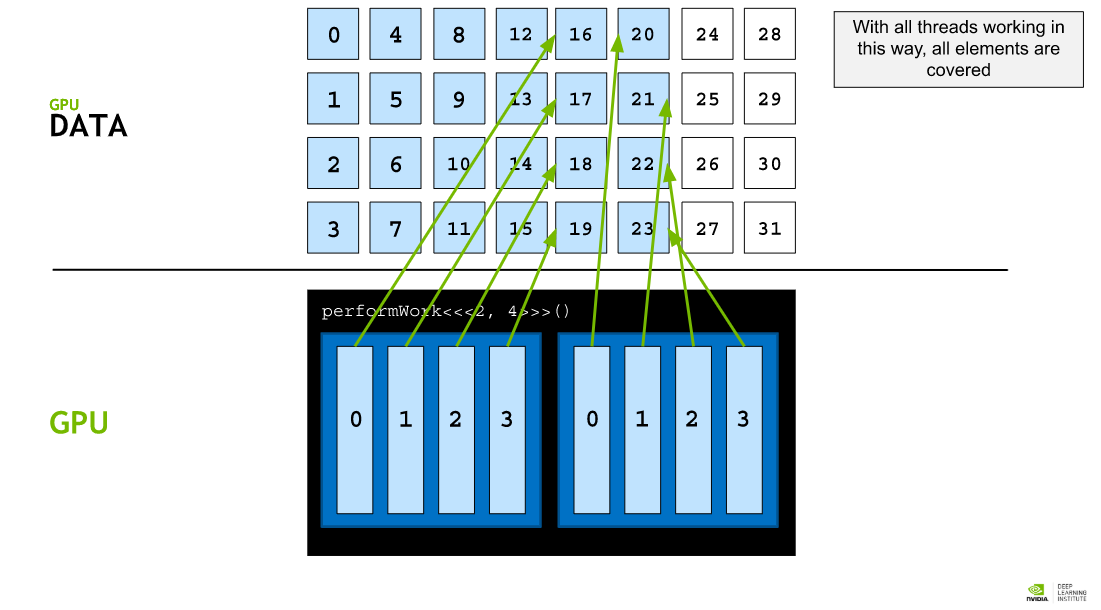
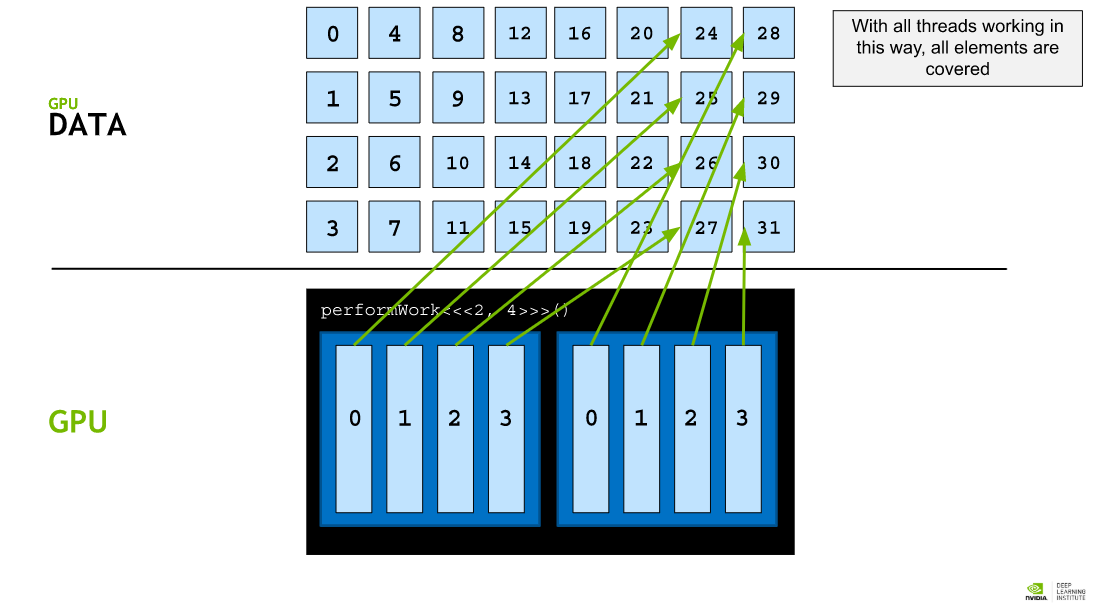
{

// Only do work if it does

}

}

## Grid-Stride Loops

## Data Sets Larger than the Grid

Either by choice, often to create the most performant execution configuration, or out of necessity, the number of threads in a grid may be smaller than the size of a data set. Consider an array with 1000 elements, and a grid with 250 threads (using trivial sizes here for ease of explanation). Here, each thread in the grid will need to be used 4 times. One common method to do this is to use a **grid-stride loop** within the kernel.

In a grid-stride loop, each thread will calculate its unique index within the grid using tid+bid\*bdim, perform its operation on the element at that index within the array, and then, add to its index the number of threads in the grid and repeat, until it is out of range of the array. For example, for a 500 element array and a 250 thread grid, the thread with index 20 in the grid would:

* Perform its operation on element 20 of the 500 element array
* Increment its index by 250, the size of the grid, resulting in 270
* Perform its operation on element 270 of the 500 element array
* Increment its index by 250, the size of the grid, resulting in 520
* Because 520 is now out of range for the array, the thread will stop its work

CUDA provides a special variable giving the number of blocks in a grid, gridDim.x. Calculating the total number of threads in a grid then is simply the number of blocks in a grid multiplied by the number of threads in each block, gridDim.x \* blockDim.x. With this in mind, here is a verbose example of a grid-stride loop within a kernel:

\_\_global void kernel(int \*a, int N)

{

int indexWithinTheGrid = threadIdx.x + blockIdx.x \*blockDim.x;

int gridStride = gridDim.x \* blockDim.x;

for (int i = indexWithinTheGrid; i < N; i += gridStride)

{

// do work on a[i];

}

}

## Error Handling

As in any application, error handling in accelerated CUDA code is essential. Many, if not most CUDA functions (see, for example, the [memory management functions](http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY)) return a value of type cudaError\_t, which can be used to check whether or not an error occured while calling the function. Here is an example where error handling is performed for a call to cudaMallocManaged:

cudaError\_t err;

err = cudaMallocManaged(&a, N) // Assume the existence of `a` and `N`.

if (err != cudaSuccess)//‘cudaSuccess` is provided by CUDA

{

printf("Error: %s\n", cudaGetErrorString(err)); // cudaGetErrorString` is provided by CUDA.

}

Launching kernels, which are defined to return void, do not return a value of type cudaError\_t. To check for errors occuring at the time of a kernel launch, for example if the launch configuration is erroneous, CUDA provides the cudaGetLastError function, which does return a value of type cudaError\_t.

/\*

\* This launch should cause an error, but the kernel itself

\* cannot return it.

\*/

someKernel<<<1, -1>>>(); // -1 is not a valid number of threads.

cudaError\_t err;

err = cudaGetLastError(); // `cudaGetLastError` will return the error from above.

if (err != cudaSuccess)

{

printf("Error: %s\n", cudaGetErrorString(err));

}

Finally, in order to catch errors that occur asynchronously, for example during the execution of an asynchronous kernel, it is essential to check the status returned by a subsequent synchronizing cuda runtime API call, such as cudaDeviceSynchronize, which will return an error if one of the kernels launched previously should fail.

## CUDA Error Handling Function

It can be helpful to create a macro that wraps CUDA function calls for checking errors. Here is an example, feel free to use it in the remaining exercises:

#include <stdio.h>

#include <assert.h>

inline cudaError\_t checkCuda(cudaError\_t result)

{

if (result != cudaSuccess) {

fprintf(stderr, "CUDA Runtime Error: %s\n",cudaGetErrorString(result));

assert(result == cudaSuccess);

}

return result;

}

int main()

{

/\*

\* The macro can be wrapped around any function

\* returning a value of type `cudaError\_t`.

\*/

checkCuda( cudaDeviceSynchronize() )

}

## Grids and Blocks of 2 and 3 Dimensions

Grids and blocks can be defined to have up to 3 dimensions. Defining them with multiple dimensions does not impact their performance in any way, but can be very helpful when dealing with data that has multiple dimensions, for example, 2d matrices. To define either grids or blocks with two or 3 dimensions, use CUDA's dim3 type as such:

dim3 threads\_per\_block(16, 16, 1);

dim3 number\_of\_blocks(16, 16, 1);

someKernel<<<number\_of\_blocks, threads\_per\_block>>>();

Given the example just above, the variables gridDim.x, gridDim.y, blockDim.x, and blockDim.y inside of someKernel, would all be equal to 16.

# Managing Accelerated Application Memory with CUDA Unified Memory and nvprof

The [CUDA Best Practices Guide](http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations), a highly recommended follow-up to this and other CUDA fundamentals labs, recommends a design cycle called **APOD**: **A**ssess, **P**arallelize, **O**ptimize, **D**eploy. In short, APOD prescribes an iterative design process, where developers can apply incremental improvements to their accelerated application's performance, and ship their code. As developers become more competent CUDA programmers, more advanced optimization techniques can be applied to their accelerated codebases.

This lab will support such a style of iterative development. You will be using the **NVIDIA Command Line Profiler** to qualitatively measure your application's performance, and to identify opportunities for optimization, after which you will apply incremental improvements before learning new techniques and repeating the cycle. As a point of focus, many of the techniques you will be learning and applying in this lab will deal with the specifics of how CUDA's **Unified Memory** works. Understanding Unified Memory behavior is a fundamental skill for CUDA developers, and serves as a prerequisite to many more advanced memory management techniques.

## Iterative Optimizations with the NVIDIA Command Line Profiler

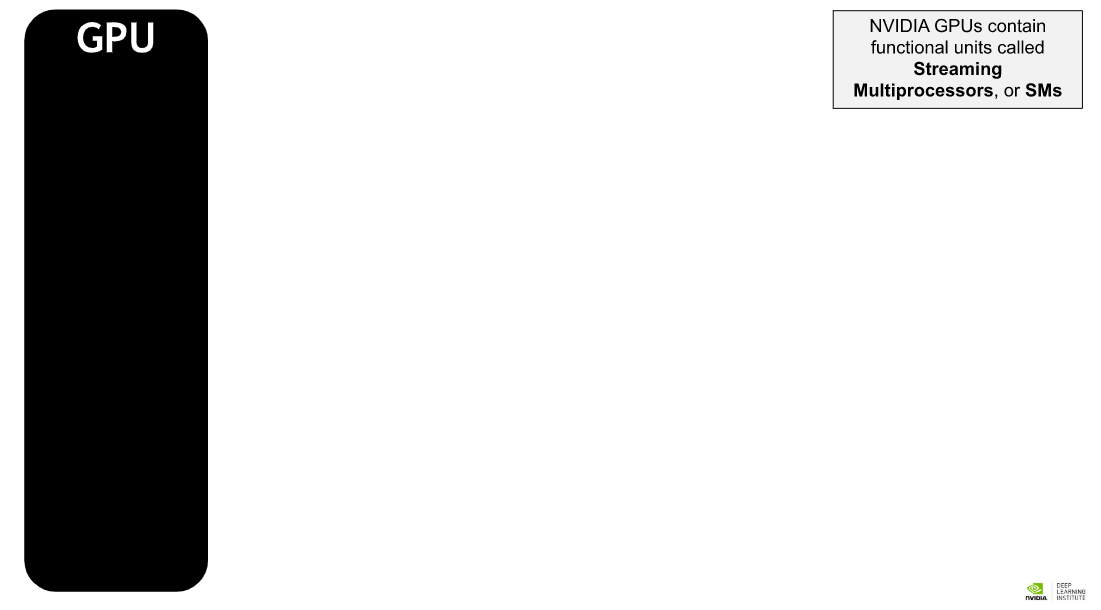
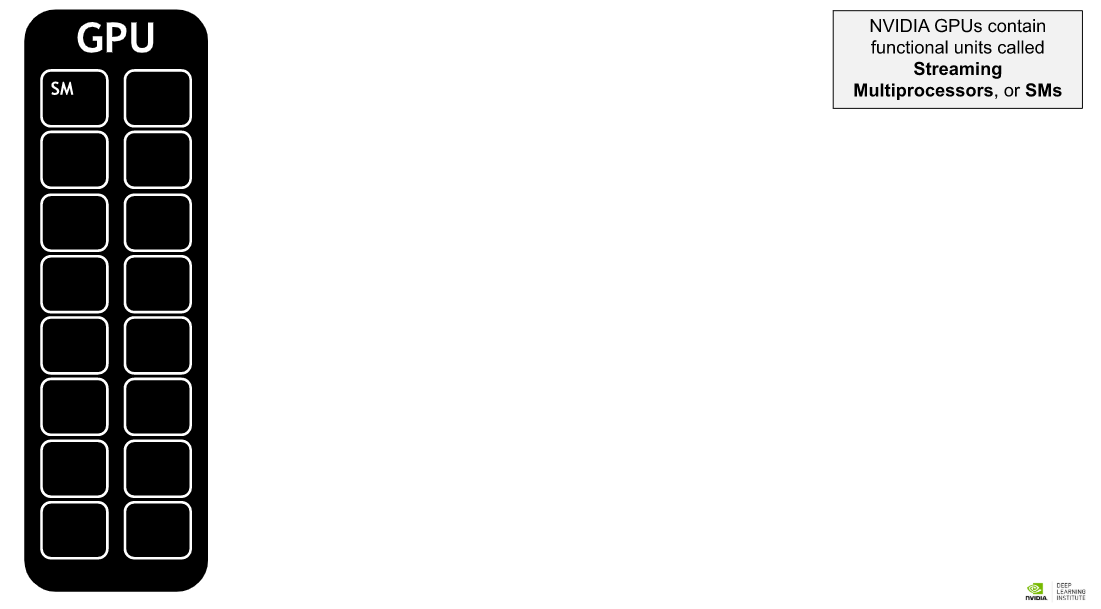
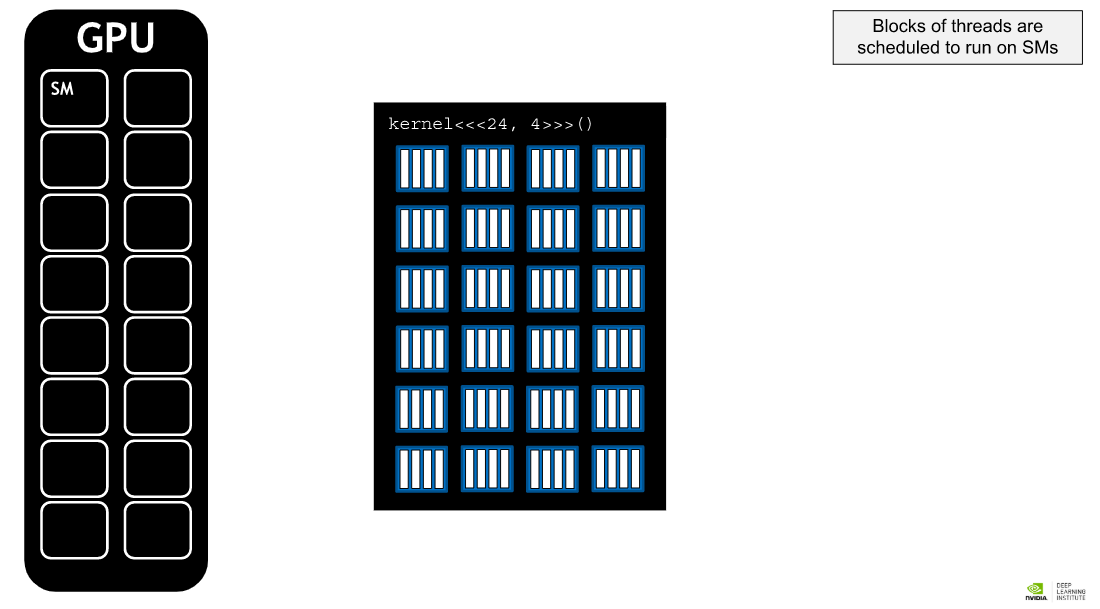
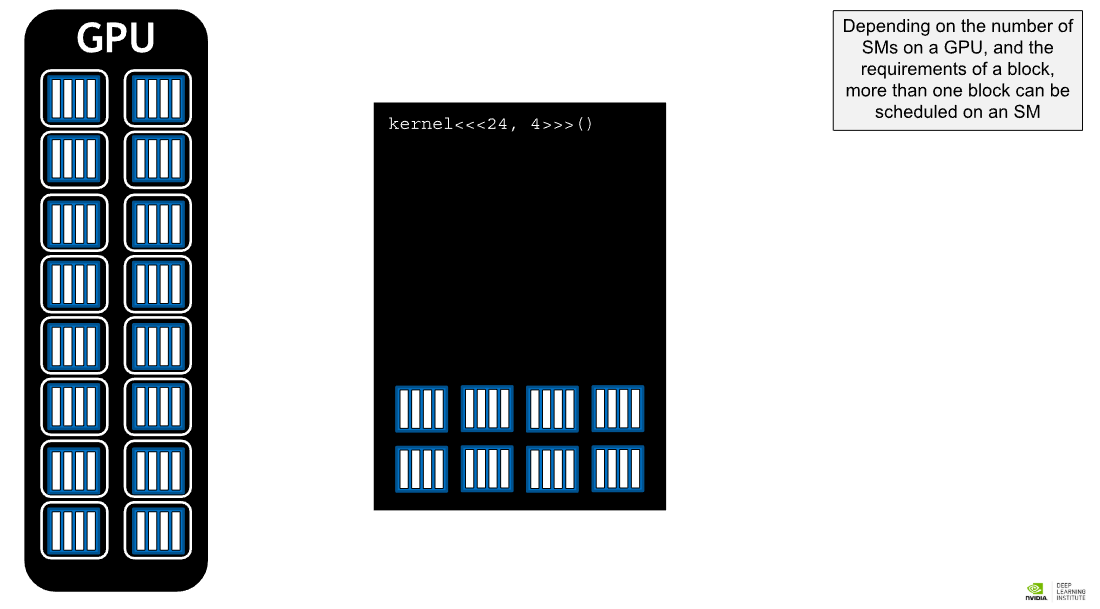
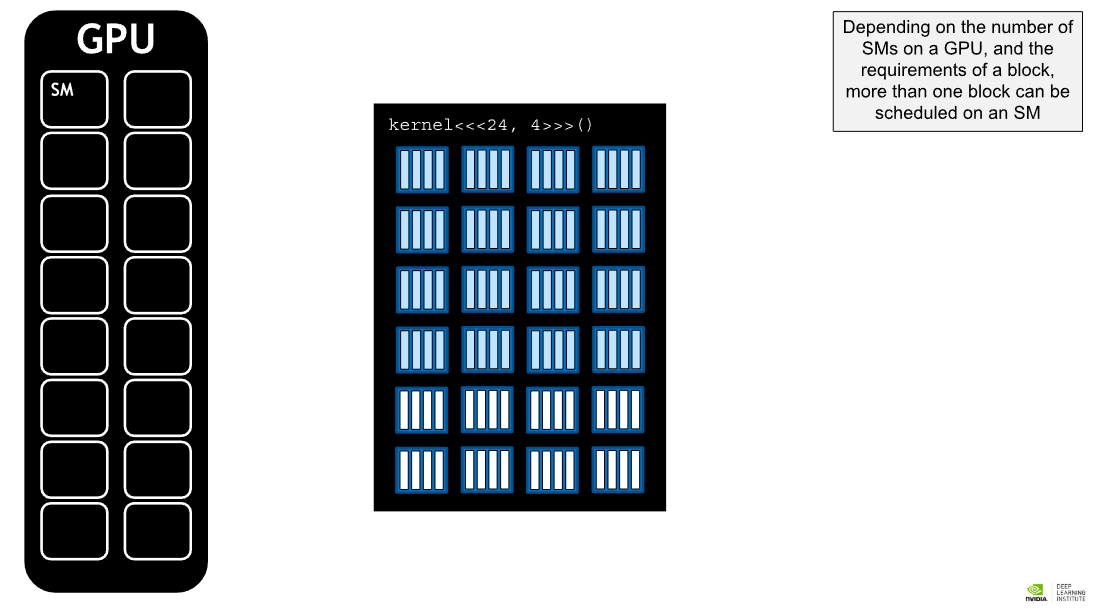
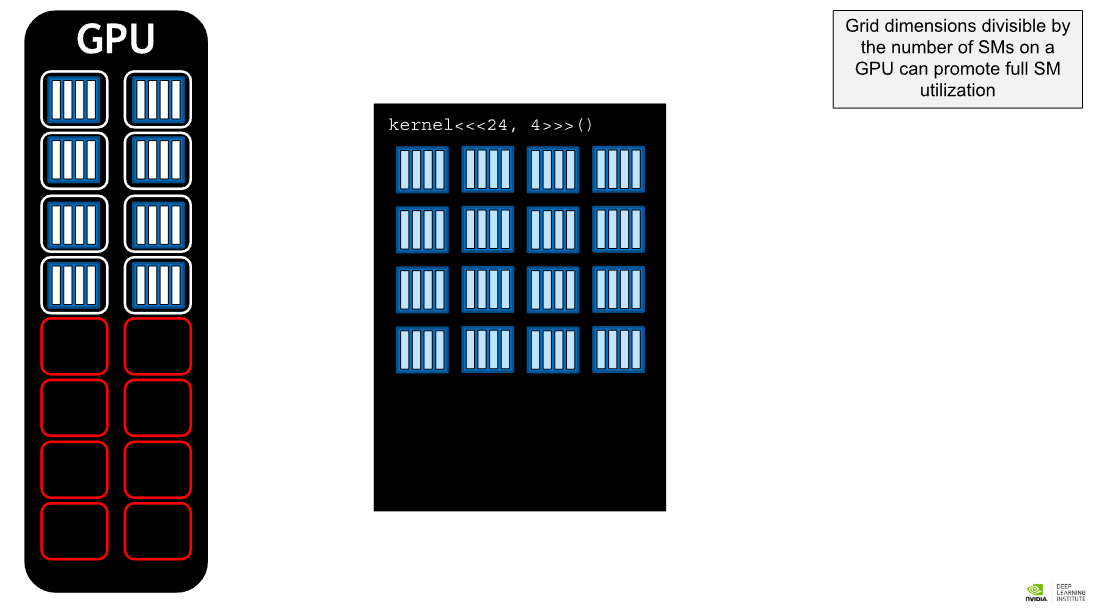
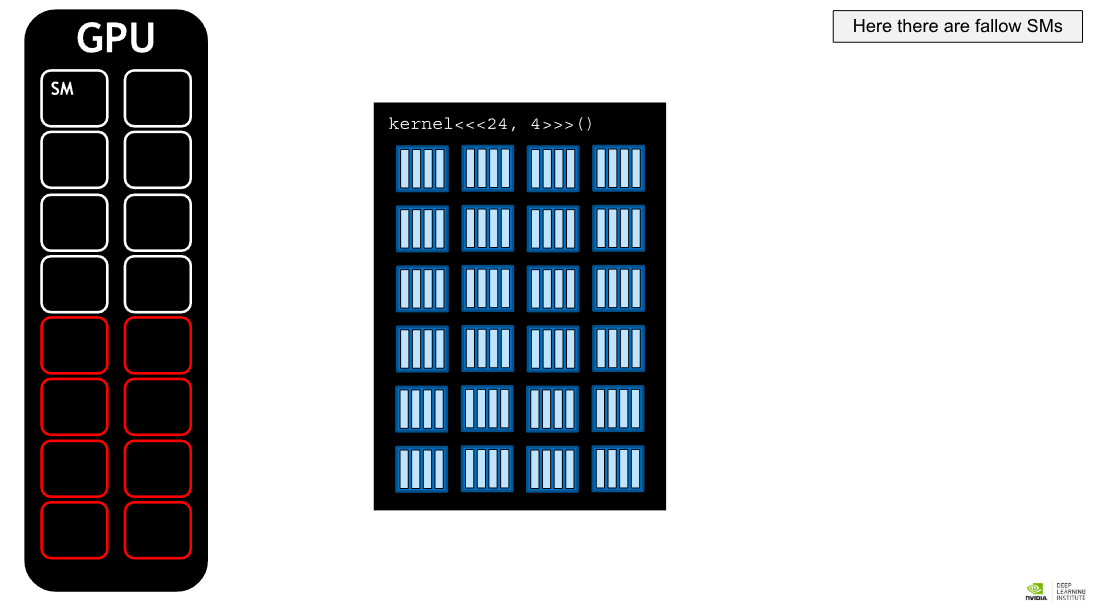
The only way to be assured that attempts at optimizing accelerated code bases are actually successful is to profile the application for quantitative information about the application's performance. nvprof is the NVIDIA command line profiler. It ships with the CUDA toolkit, and is a powerful tool for profiling accelerated applications.

nvprof is easy to use. Its most basic usage is to simply pass it the path to an executable compiled with nvcc. nvprof will proceed to execute the application, after which it will print a summary output of the application's GPU activities, CUDA API calls, as well as information about **Unified Memory** activity, a topic which will be covered extensively later in this lab.

When accelerating applications, or optimizing already-accelerated applications, take a scientific and iterative approach. Profile your application after making changes, take note, and record the implications of any refactoring on performance. Make these observations early and often: frequently, enough performance boost can be gained with little effort such that you can ship your accelerated application. Additionally, frequent profiling will teach you how specific changes to your CUDA codebases impact its actual performance: knowledge that is hard to acquire when only profiling after many kinds of changes in your codebase.

## Streaming Multiprocessors and Querying the Device

This section explores how understanding a specific feature of the GPU hardware can promote optimization. After introducing **Streaming Multiprocessors**, you will attempt to further optimize the accelerated vector addition program you have been working on.

### Streaming Multiprocessors and Warps

The GPUs that CUDA applications run on have processing units called **streaming multiprocessors**, or **SMs**. During kernel execution, blocks of threads are given to SMs to execute. In order to support the GPU's ability to perform as many parallel operations as possible, performance gains can often be had by choosing a grid size that has a number of blocks that is a multiple of the number of SMs on a given GPU.

Additionally, SMs create, manage, schedule, and execute groupings of 32 threads from within a block called **warps**. A more [in depth coverage of SMs and warps](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation) is beyond the scope of this course, however, it is important to know that performance gains can also be had by choosing a block size that has a number of threads that is a multiple of 32.

### Programmatically Querying GPU Device Properties

In order to support portability, since the number of SMs on a GPU can differ depending on the specific GPU being used, the number of SMs should not be hard-coded into a codebase. Rather, this information should be acquired programmatically.

The following shows how, in CUDA C/C++, to obtain a C struct which contains many properties about the currently active GPU device, including its number of SMs:

int deviceId;

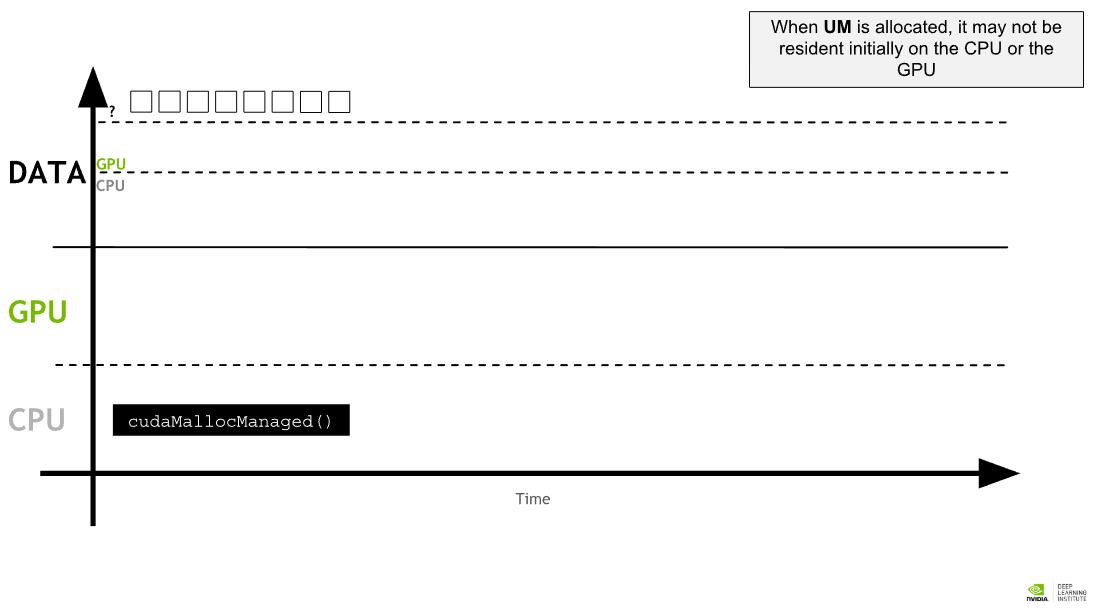
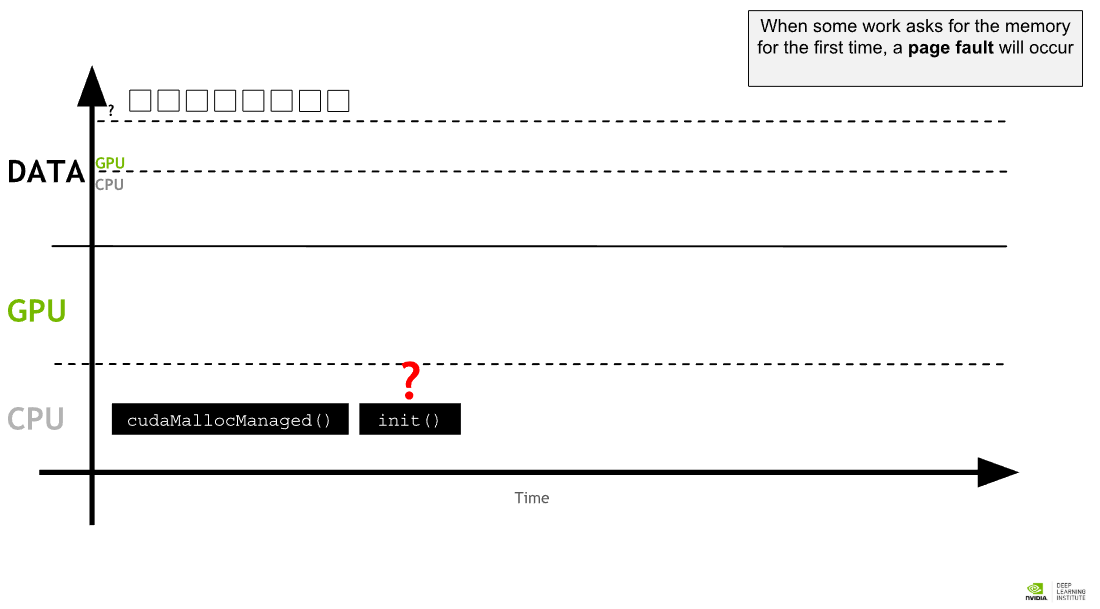
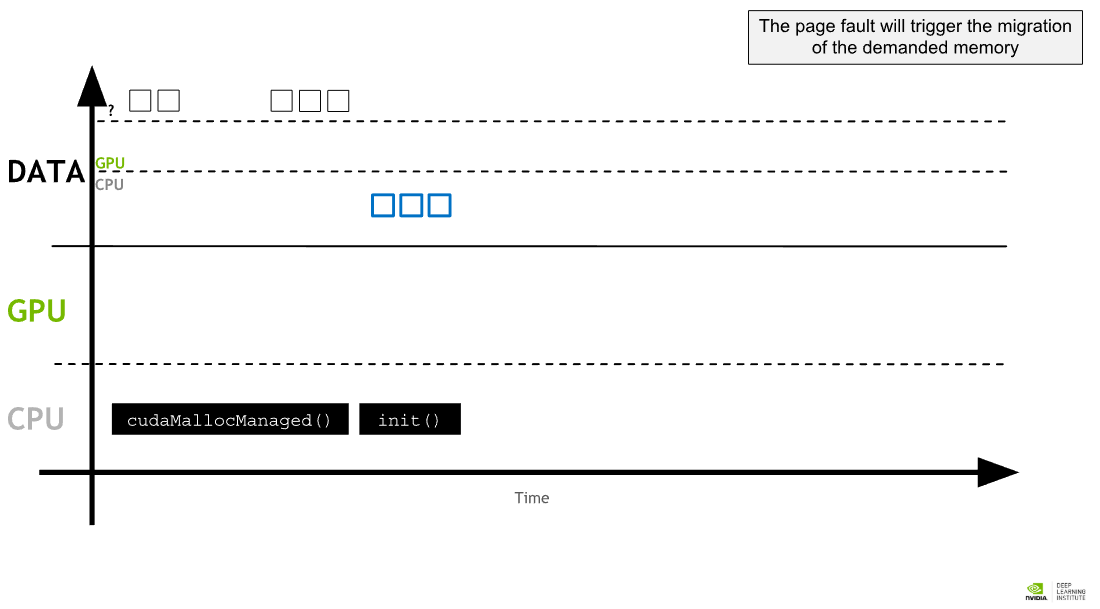
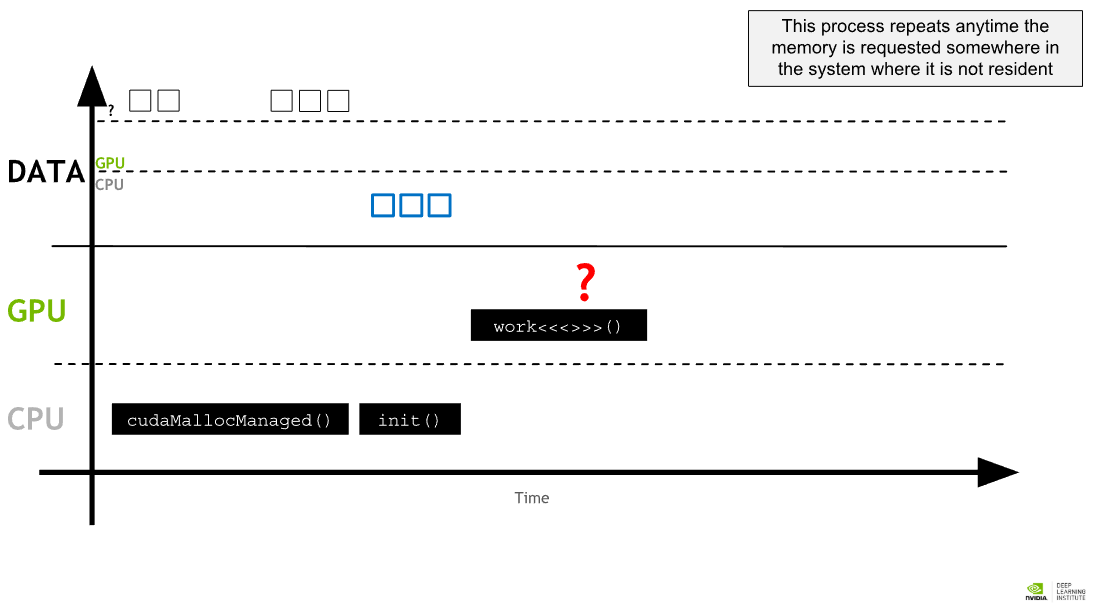
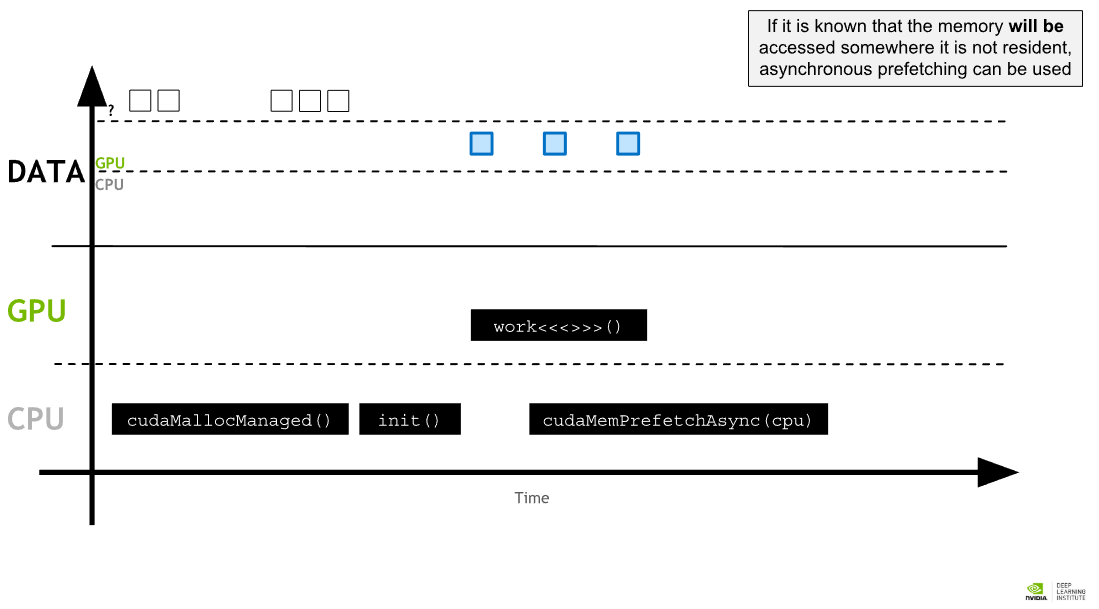
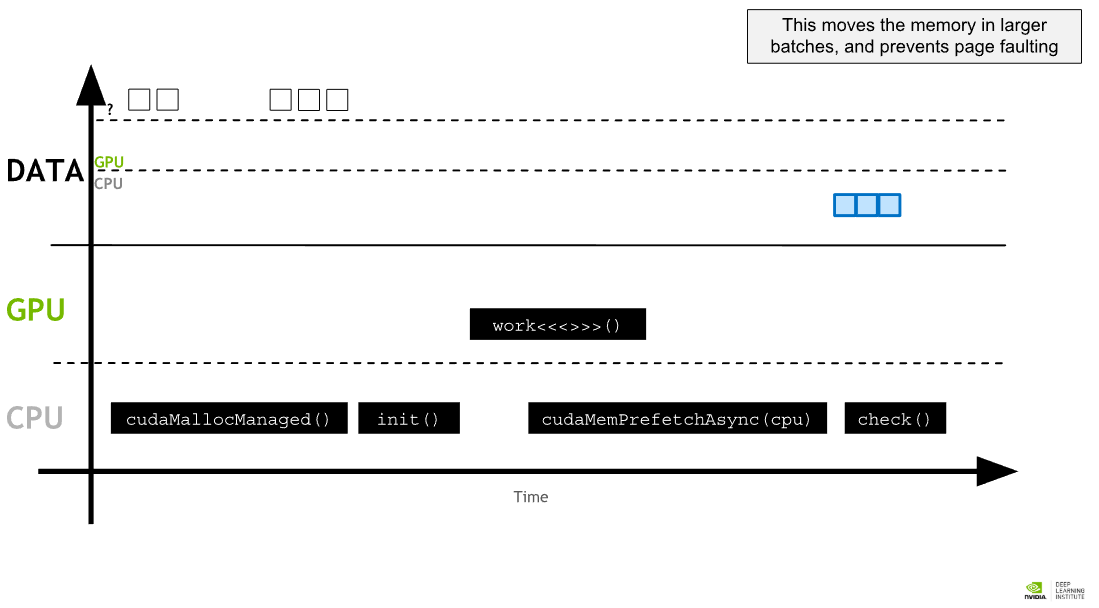
cudaGetDevice(&deviceId); // `deviceId` now points to the id of the currently active GPU.

cudaDeviceProp props;

cudaGetDeviceProperties(&props, deviceId); // `props` now has many useful properties about active GPU device*.*

## Unified Memory Details

You have been allocting memory intended for use either by host or device code with cudaMallocManaged and up until now have enjoyed the benefits of this method - automatic memory migration, ease of programming - without diving into the details of how the **Unified Memory (UM)** allocated by cudaMallocManaged actual works. nvprof provides details about UM management in accelerated applications, and using this information, in conjunction with a more-detailed understanding of how UM works, provides additional opportunities to optimize accelerated applications.

### Unified Memory Migration

When UM is allocated, the memory is not resident yet on either the host or the device. When either the host or device attempts to access the memory, a [page fault](https://en.wikipedia.org/wiki/Page_fault) will occur, at which point the host or device will migrate the needed data in batches. Similarly, at any point when the CPU, or any GPU in the accelerated system, attempts to access memory not yet resident on it, page faults will occur and trigger its migration.

The ability to page fault and migrate memory on demand is tremendously helpful for ease of development in your accelerated applications. Additionally, when working with data that exhibits sparse access patterns, for example when it is impossible to know which data will be required to be worked on until the application actually runs, and for scenarios when data might be accessed by multiple GPU devices in an accelerated system with multiple GPUs, on-demand memory migration is remarkably beneficial.

There are times - for example when data needs are known prior to runtime, and large contiguous blocks of memory are required - when the overhead of page faulting and migrating data on demand incurs an overhead cost that would be better avoided.

Much of the remainder of this lab will be dedicated to understanding on-demand migration, and how to identify it in the profiler's output. With this knowledge you will be able to reduce the overhead of it in scenarios when it would be beneficial.

## Asynchronous Memory Prefetching

A powerful technique to reduce the overhead of page faulting and on-demand memory migrations, both in host-to-device and device-to-host memory transfers, is called **asynchronous memory prefetching**. Using this technique allows programmers to asynchronously migrate unified memory (UM) to any CPU or GPU device in the system, in the background, prior to its use by application code. By doing this, GPU kernels and CPU function performance can be increased on account of reduced page fault and on-demand data migration overhead.

Prefetching also tends to migrate data in larger chunks, and therefore fewer trips, than on-demand migration. This makes it an excellent fit when data access needs are known before runtime, and when data access patterns are not sparse.

CUDA Makes asynchronously prefetching managed memory to either a GPU device or the CPU easy with its cudaMemPrefetchAsync function. Here is an example of using it to both prefetch data to the currently active GPU device, and then, to the CPU:

int deviceId;

cudaGetDevice(&deviceId); // The ID of the currently active GPU device.

cudaMemPrefetchAsync(pointerToSomeUMData, size, deviceId); // Prefetch to GPU device.

cudaMemPrefetchAsync(pointerToSomeUMData, size, cudaCpuDeviceId); // Prefetch to host. `cudaCpuDeviceId` is a built-in CUDA variable.

# Asynchronous Streaming, and Visual Profiling for Accelerated Applications with CUDA C/C++

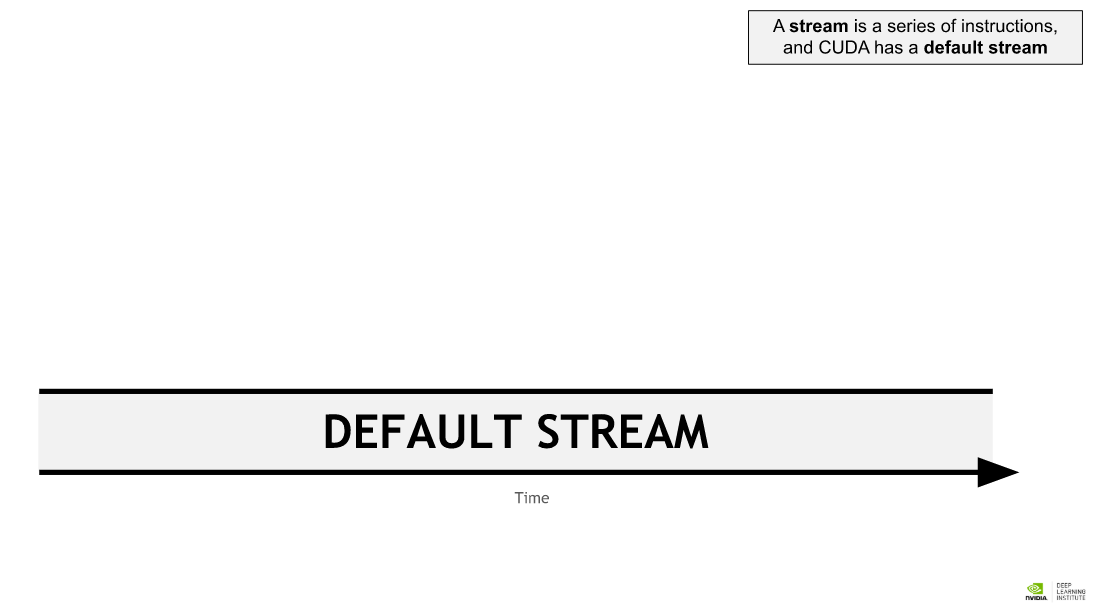
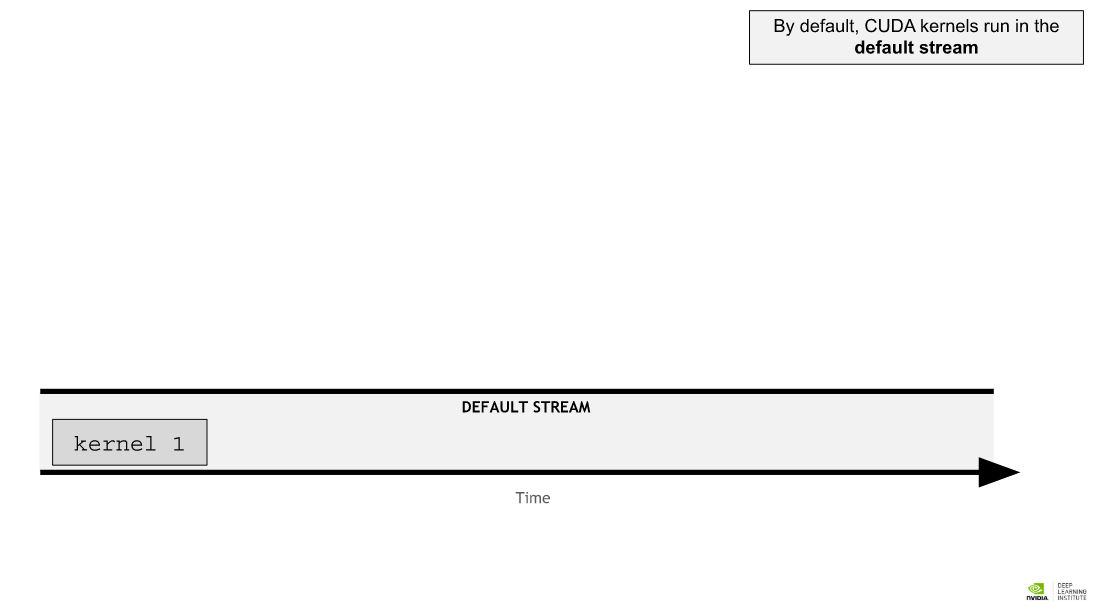
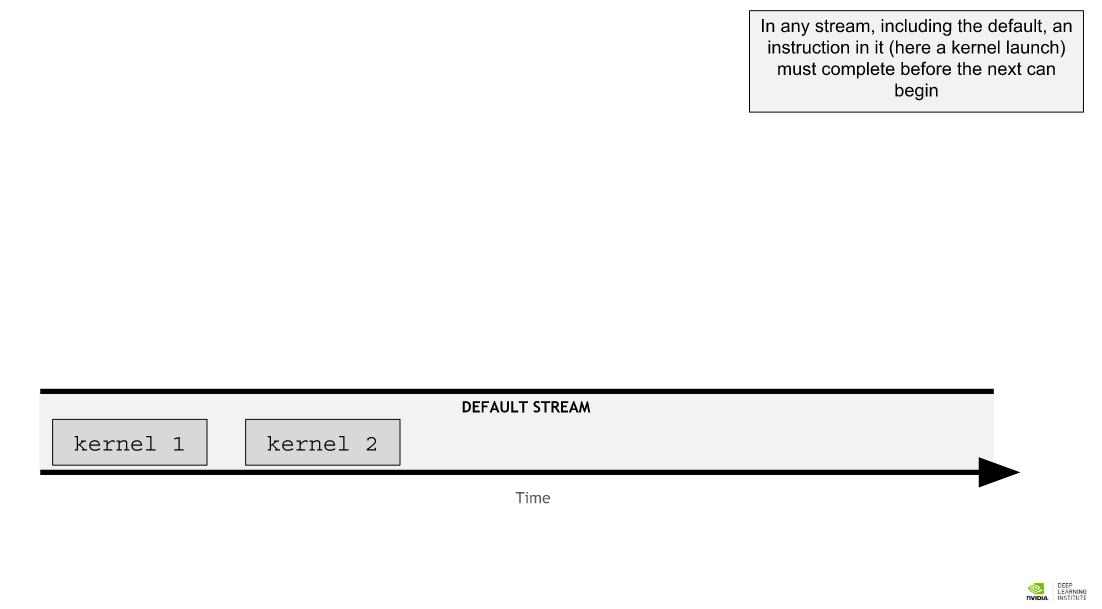
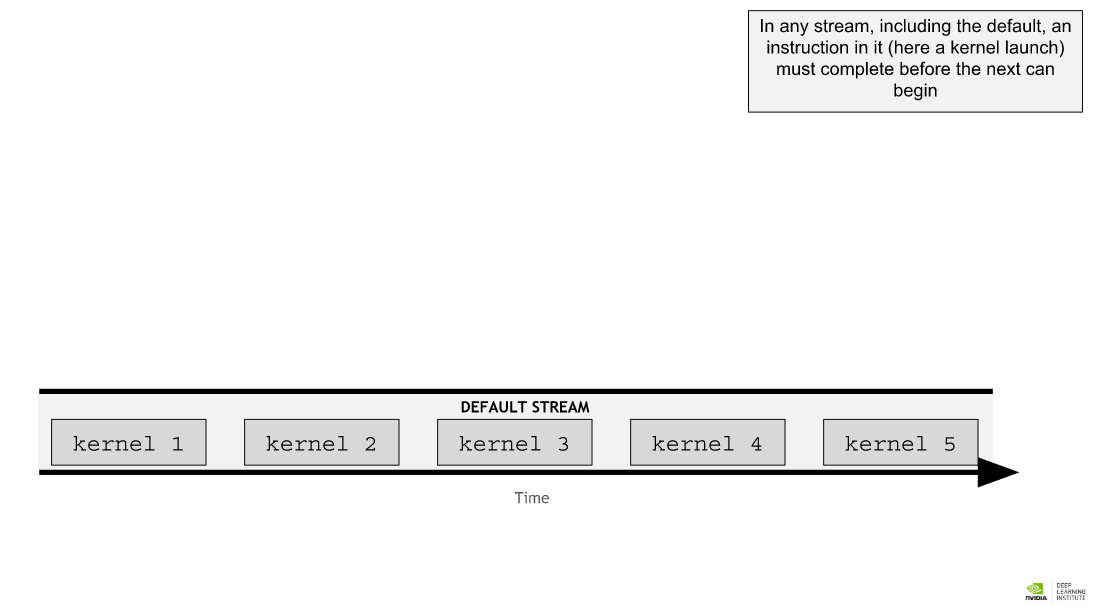
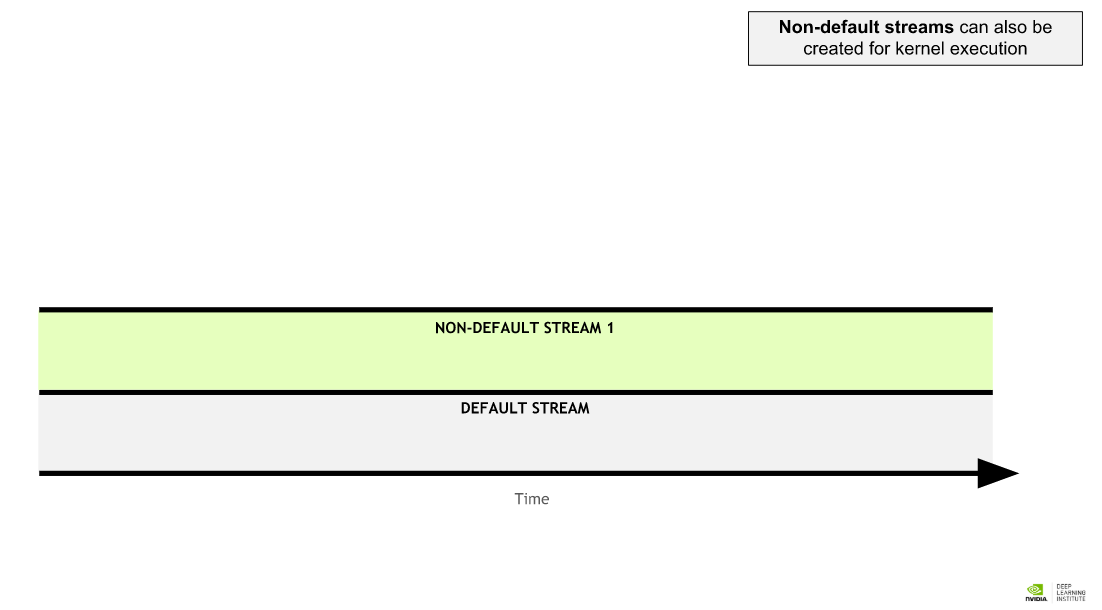
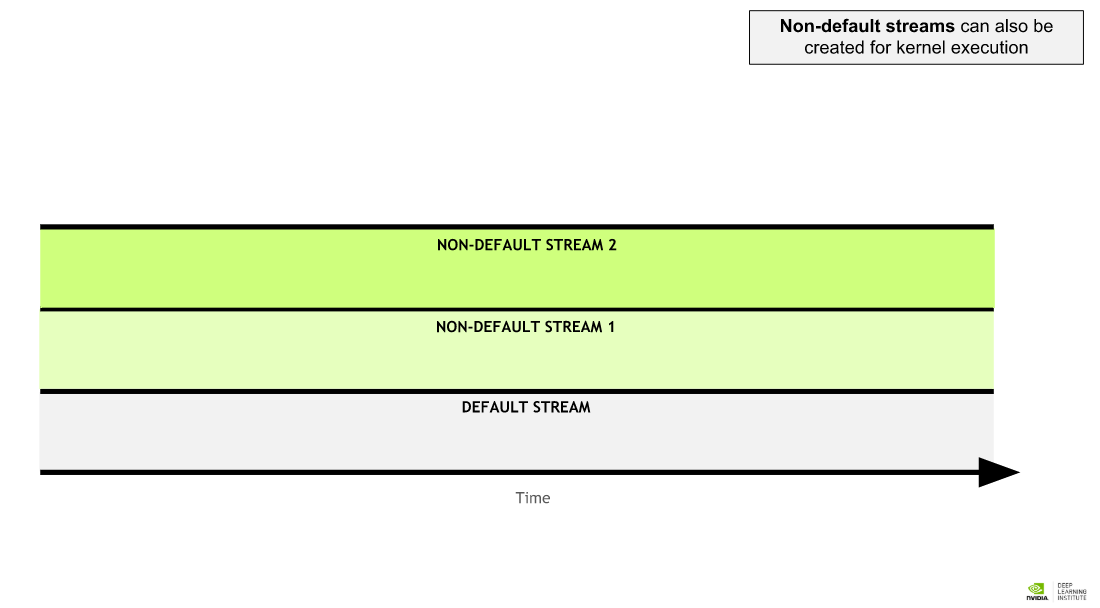
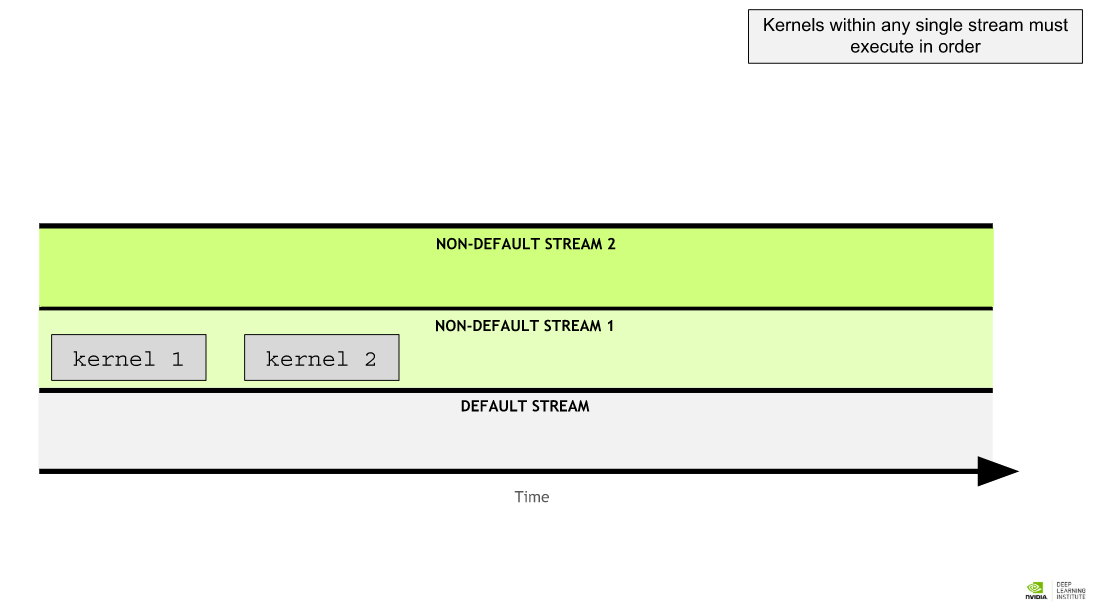
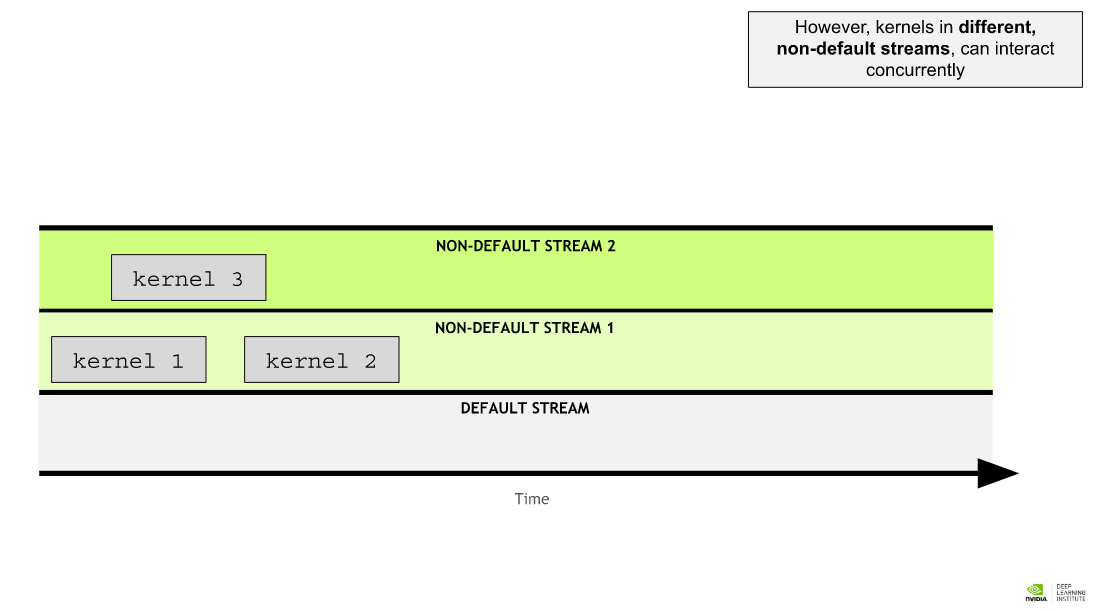
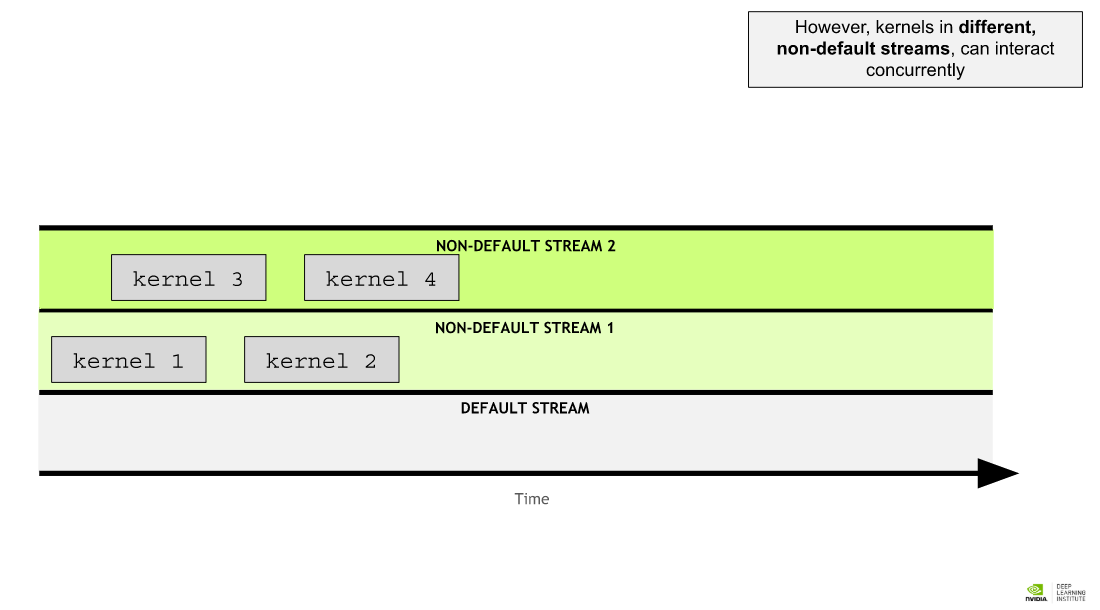
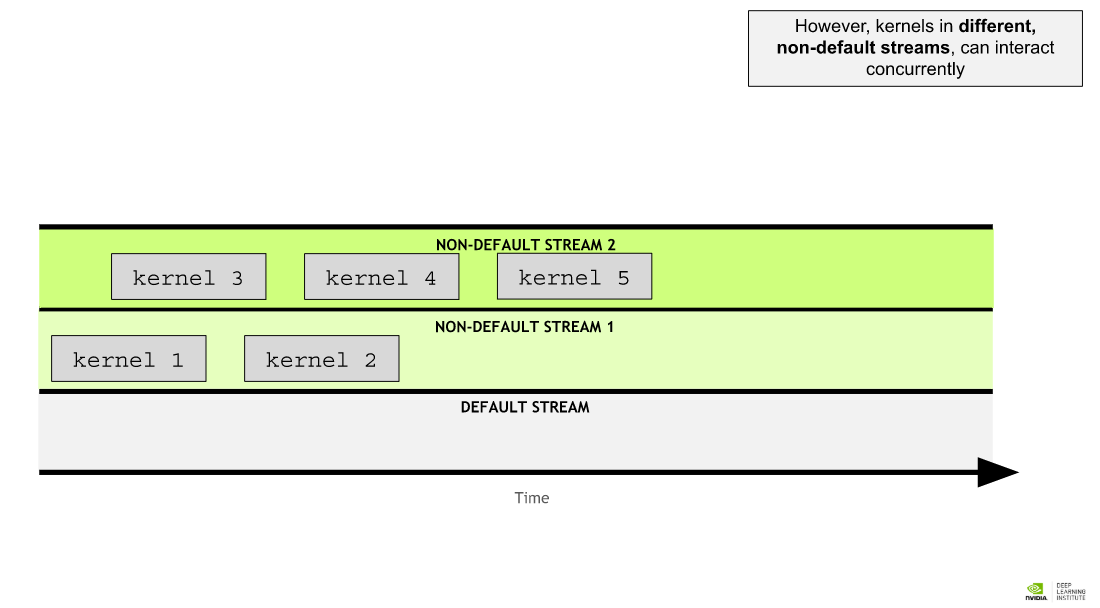
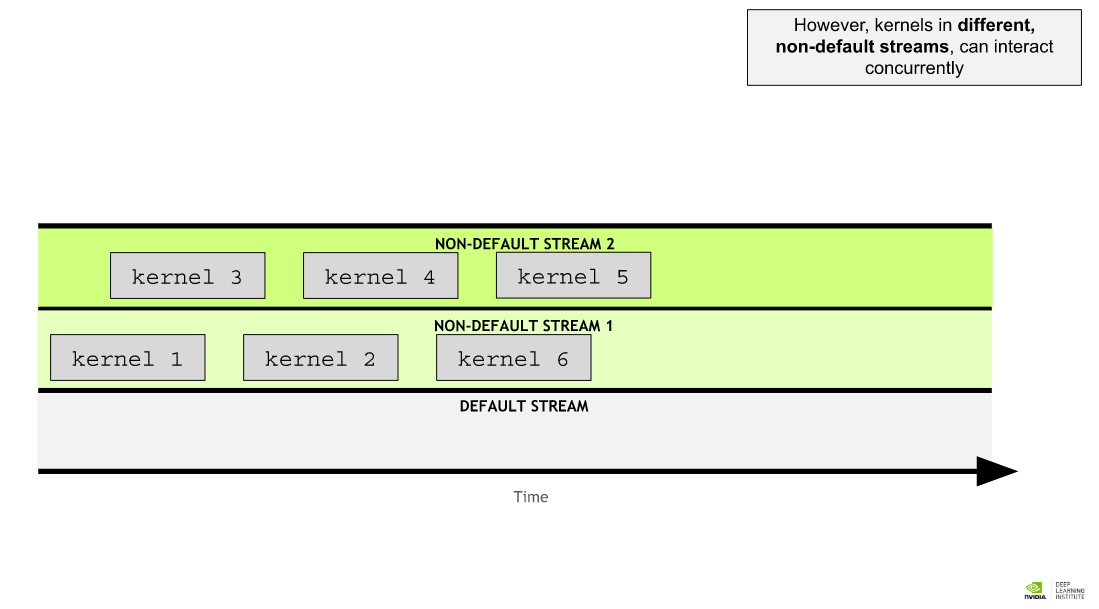
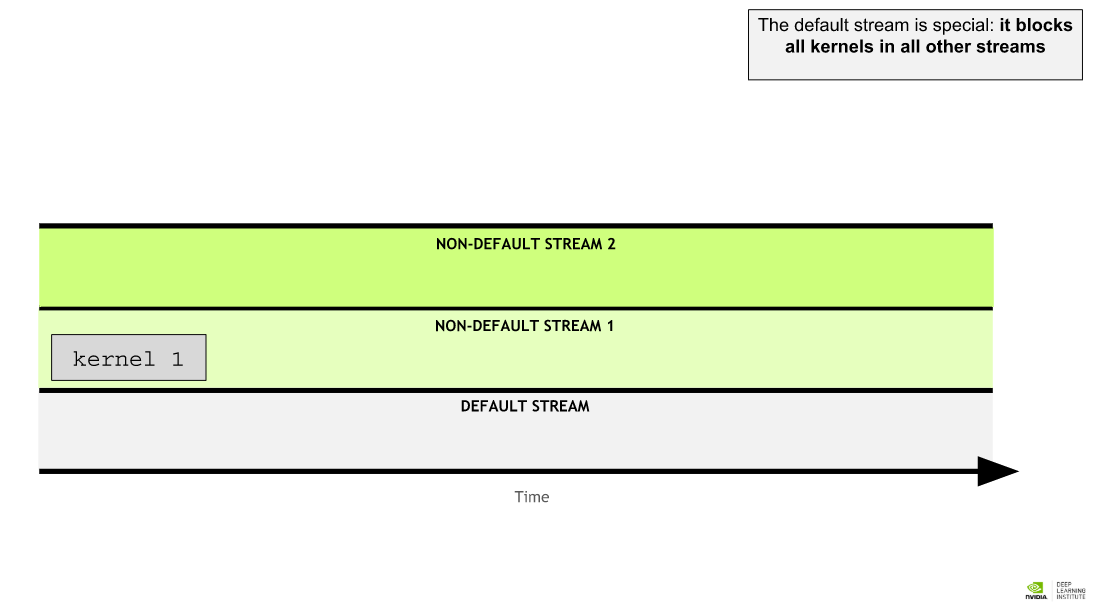
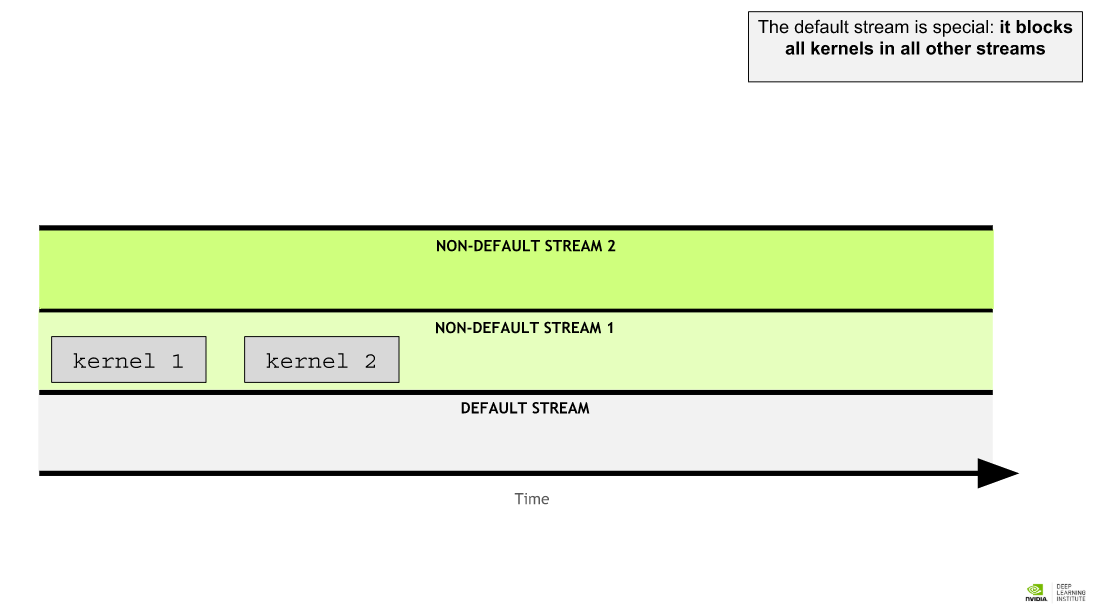
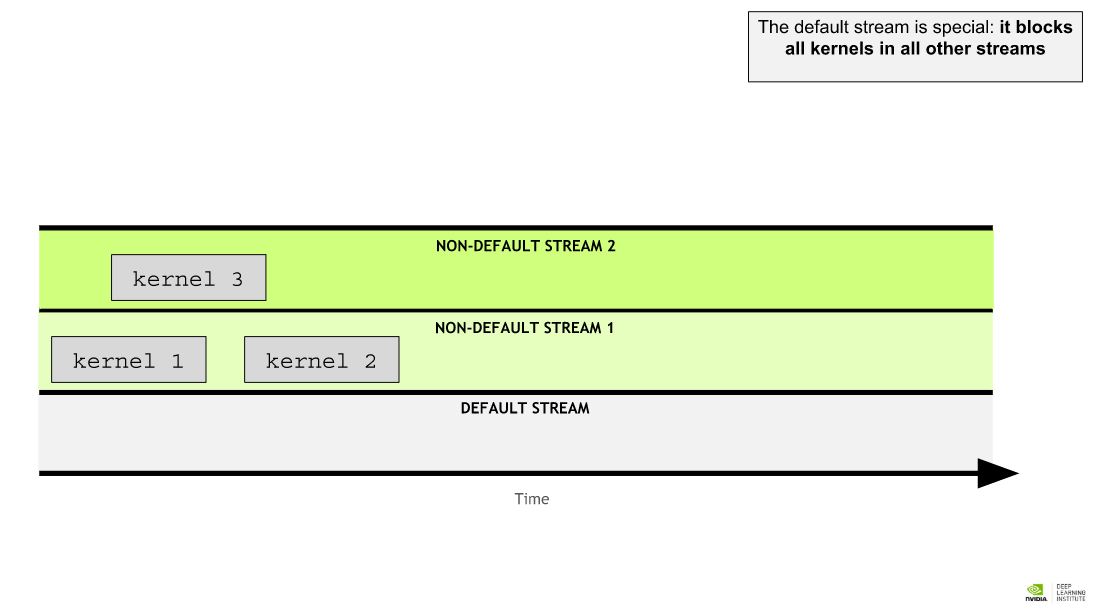
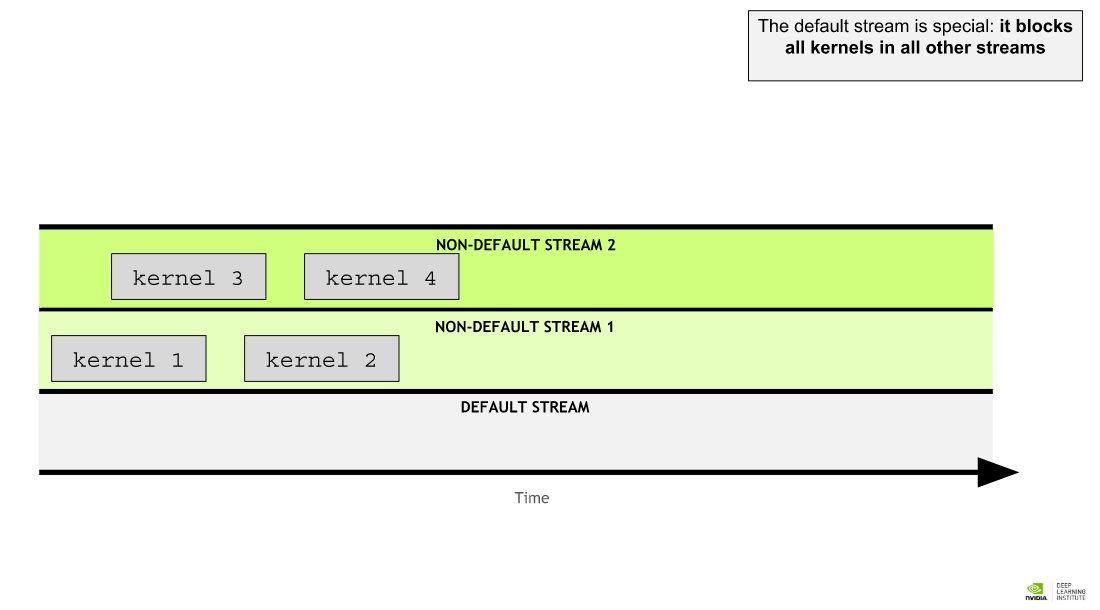
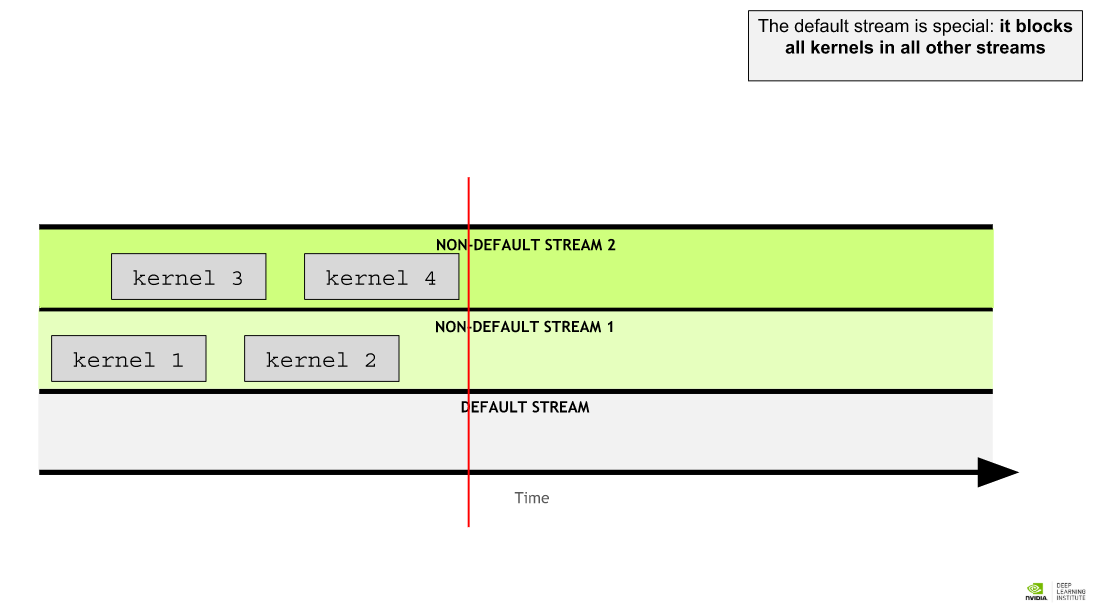
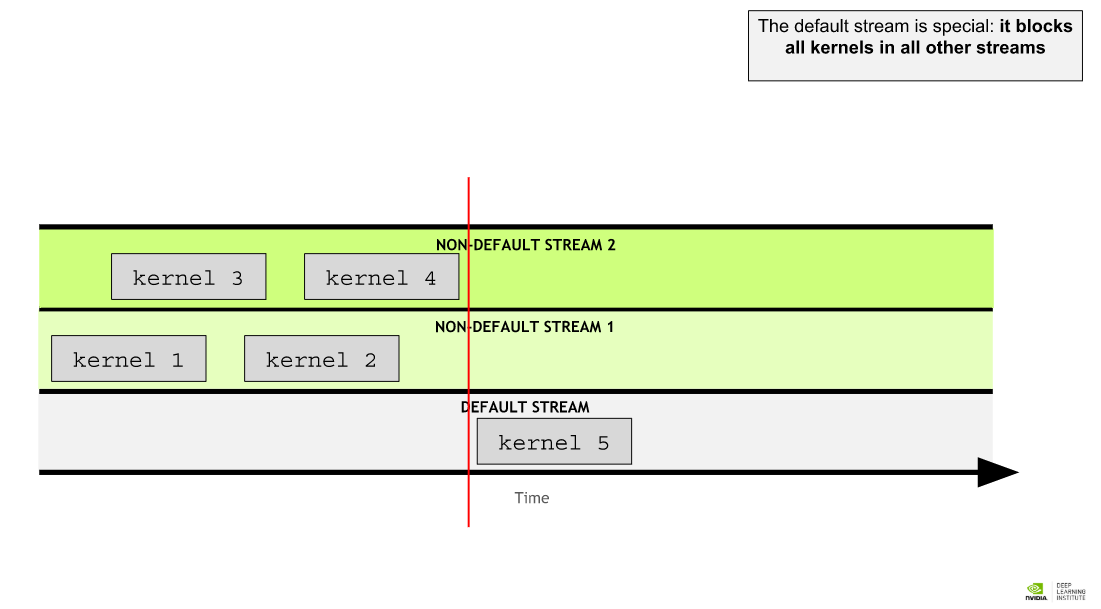
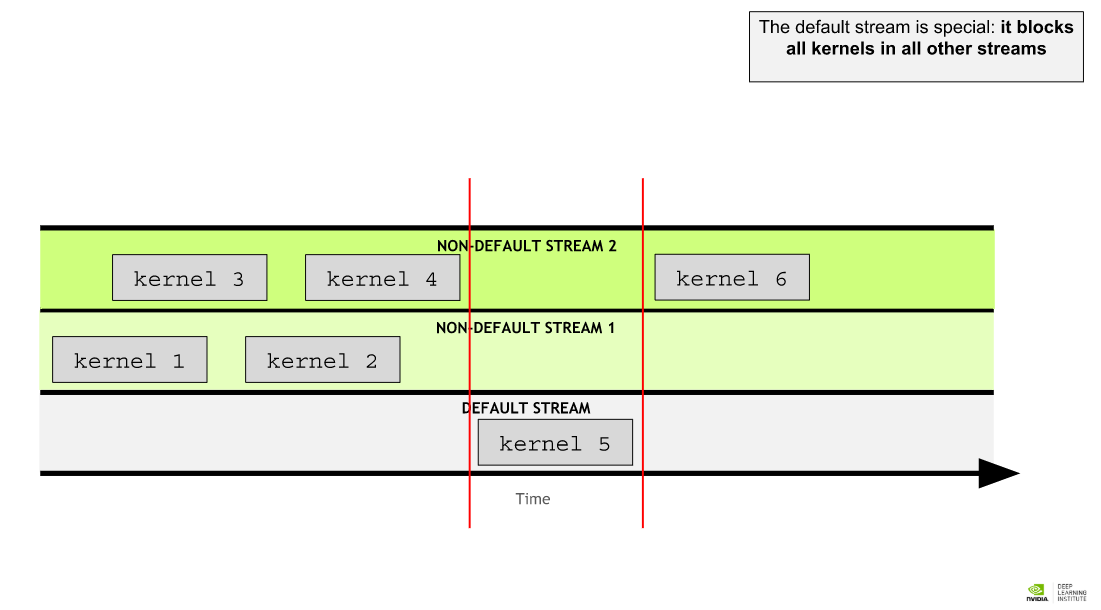
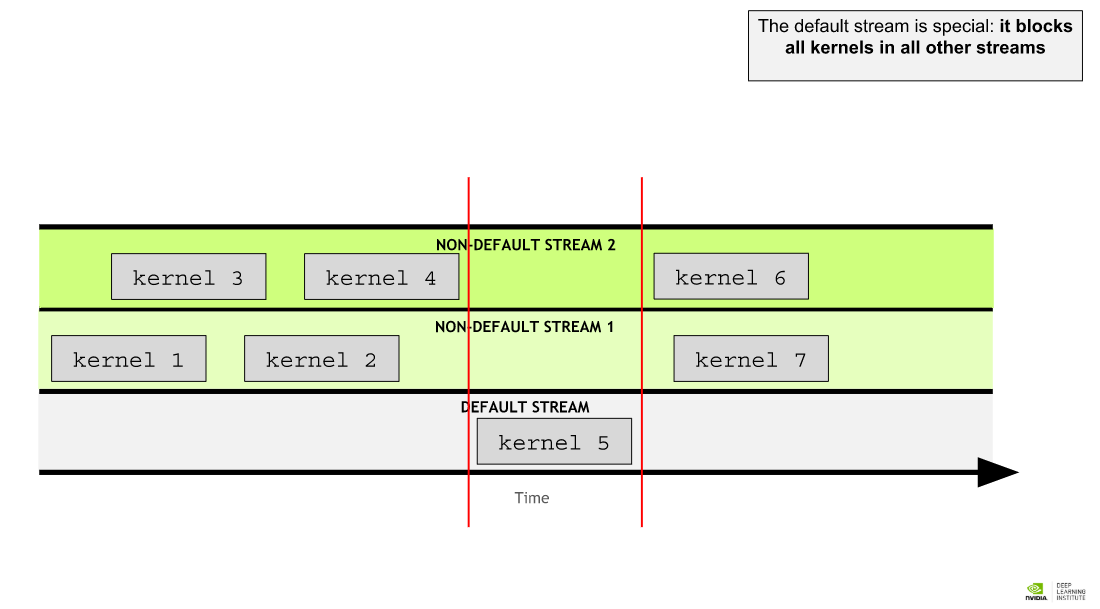
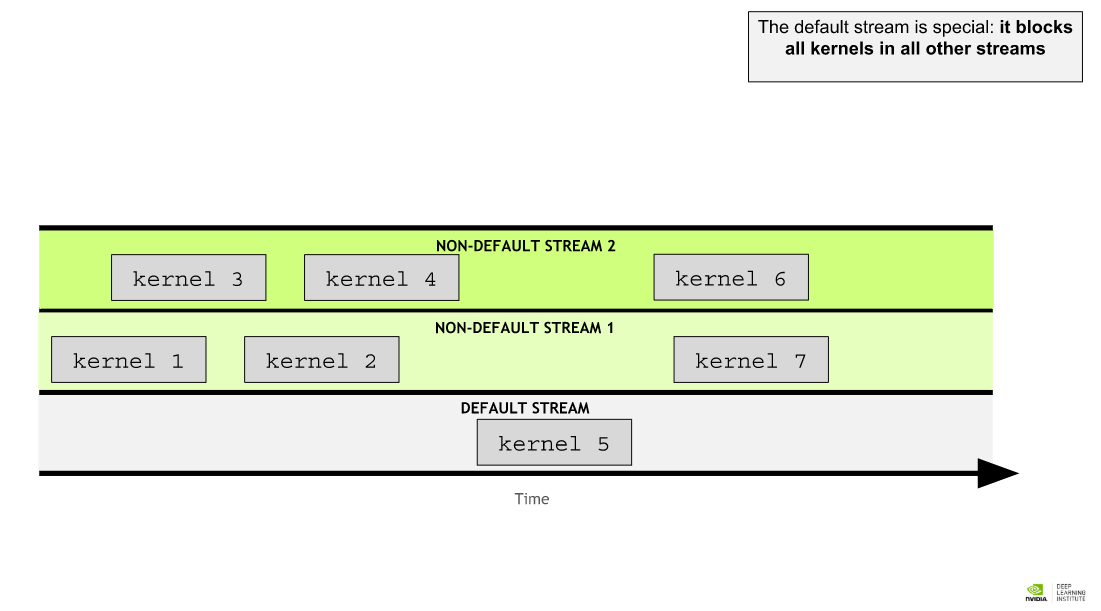
The CUDA tookit ships with the **NVIDIA Visual Profiler**, or **nvvp**, a powerful GUI application to support the development of accelerated CUDA applications. nvvp generates a graphical timeline of an accelerated application, with detailed information about CUDA API calls, kernel execution, memory activity, and the use of **CUDA streams**.

Additionally, nvvp provides a suite of analysis tools that developers can run to receive intelligent recommendations about how to best optimize their accelerated applications. Learning nvvp well is a must for CUDA developers.

In this lab, you will be using the nvvp timeline to guide you in optimizing accelerated applications. Additionally, you will learn some intermediate CUDA programming techniques to support your work: **unmanaged memory allocation and migration**; **pinning**, or **page-locking host memory**; and **non-default concurrent CUDA streams**.

At the end of this lab, you will be presented with an assessment, to accelerate and optimize a simple n-body simulator, which will allow you to demonstrate the skills you have developed during this course. Those of you who are able to accelerate the simulator while maintaining its correctness, will be granted a certification as proof of your competency.

## Concurrent CUDA Streams

In CUDA programming, a **stream** is a series of commands that execute in order. In CUDA applications, kernel execution, as well as some memory transfers, occur within CUDA streams. Up until this point in time, you have not been interacting explicitly with CUDA streams, but as you saw in the nvvp timeline in the last exercise, your CUDA code has been executing its kernels inside of a stream called the *default stream*.

CUDA programmers can create and utilize non-default CUDA streams in addition to the default stream, and in doing so, perform multiple operations, such as executing multiple kernels, concurrently, in different streams. Using multiple streams can add an additional layer of parallelization to your accelerated applications, and offers many more opportunities for application optimization.

### Rules Governing the Behavior of CUDA Streams

There are a few rules, concerning the behavior of CUDA streams, that should be learned in order to utilize them effectively:

* Operations within a given stream occur in order.
* Operations in different non-default streams are not guaranteed to operate in any specific order relative to each other.
* The default stream is blocking and will both wait for all other streams to complete before running, and, will block other streams from running until it completes.

### Creating, Utilizing, and Destroying Non-Default CUDA Streams

The following code snippet demonstrates how to create, utilize, and destroy a non-default CUDA stream. You will note, that to launch a CUDA kernel in a non-default CUDA stream, the stream must be passed as the optional 4th argument of the execution configuration. Up until now you have only utilized the first 2 arguments of the execution configuration:

cudaStream\_t stream; // CUDA streams are of type `cudaStream\_t`.

cudaStreamCreate(&stream); // Note that a pointer must be passed to `cudaCreateStream`.

someKernel<<<number\_of\_blocks, threads\_per\_block, 0, stream>>>(); // `stream` is passed as 4th EC argument.

cudaStreamDestroy(stream); // Note that a value, not a pointer, is passed to `cudaDestroyStream`.

Outside the scope of this lab, but worth mentioning, is the optional 3rd argument of the execution configuration. This argument allows programmers to supply the number of bytes in shared memory (an advanced topic that will not be covered presently) to be dynamically allocated per block for this kernel launch. The default number of bytes allocated to shared memory per block is 0, and for the remainder of the lab, you will be passing 0 as this value, in order to expose the 4th argument, which is of immediate interest.

## Manual Memory Allocation and Copying

While cudaMallocManaged and cudaMemPrefetchAsync are performant, and greatly simplify memory migration, sometimes it can be worth it to use more manual methods for memory allocation. This is particularly true when it is known that data will only be accessed on the device or host, and the cost of migrating data can be reclaimed in exchange for the fact that no automatic on-demand migration is needed.

Additionally, using manual memory management can allow for the use of non-default streams for overlapping data transfers with computational work. In this section you will learn some basic manual memory allocation and copy techniques, before extending these techniques to overlap data copies with computational work.

Here are some CUDA commands for manual memory management:

* cudaMalloc will allocate memory directly to the active GPU. This prevents all GPU page faults. In exchange, the pointer it returns is not available for access by host code.
* cudaMallocHost will allocate memory directly to the CPU. It also "pins" the memory, or page locks it, which will allow for asynchronous copying of the memory to and from a GPU. Too much pinned memory can interfere with CPU performance, so use it only with intention. Pinned memory should be freed with cudaFreeHost.
* cudaMemcpy can copy (not transfer) memory, either from host to device or from device to host.

### Manual Memory Management Example

Here is a snippet of code that demonstrates the use of the above CUDA API calls.

int \*host\_a, \*device\_a; // Define host-specific and device-specific arrays.

cudaMalloc(&device\_a, size); // `device\_a` is immediately available on the GPU.

cudaMallocHost(&host\_a, size); // `host\_a` is immediately available on CPU, and is page-locked, or pinned.

initializeOnHost(host\_a, N); // No CPU page faulting since memory is already allocated on the host.

// `cudaMemcpy` takes the destination, source, size, and a CUDA-provided variable for the direction of the copy.

cudaMemcpy(device\_a, host\_a, size, cudaMemcpyHostToDevice);

kernel<<<blocks, threads, 0, someStream>>>(device\_a, N);

// `cudaMemcpy` can also copy data from device to host.

cudaMemcpy(host\_a, device\_a, size, cudaMemcpyDeviceToHost);

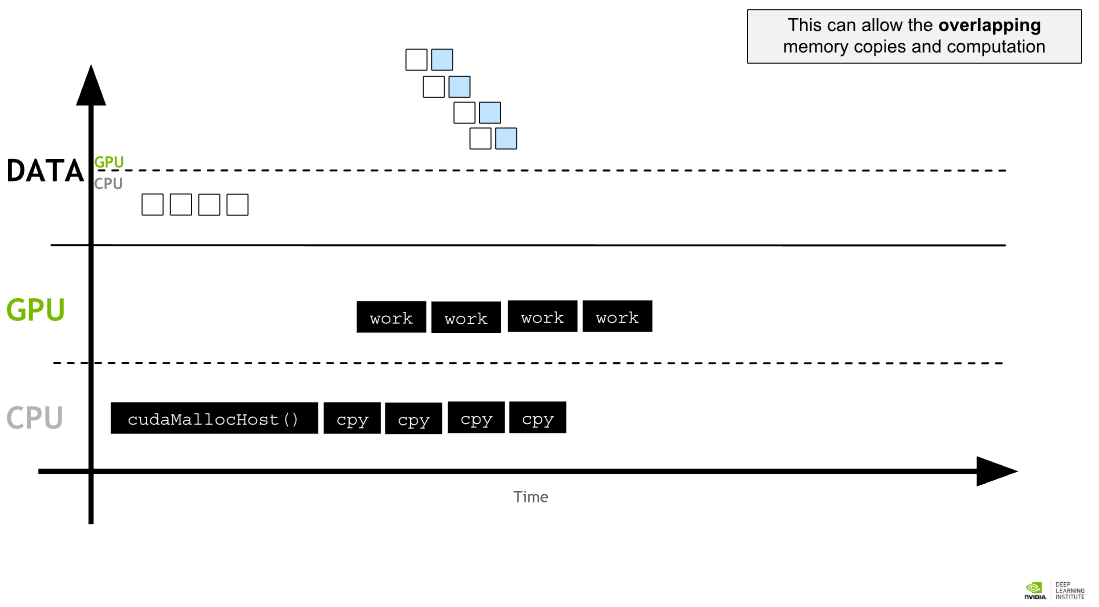
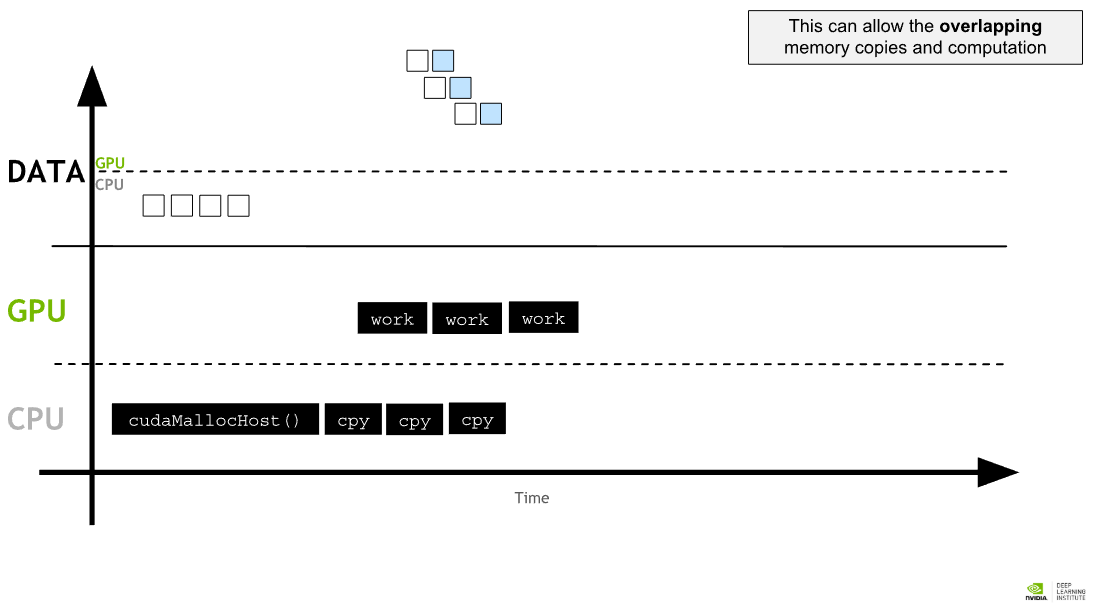
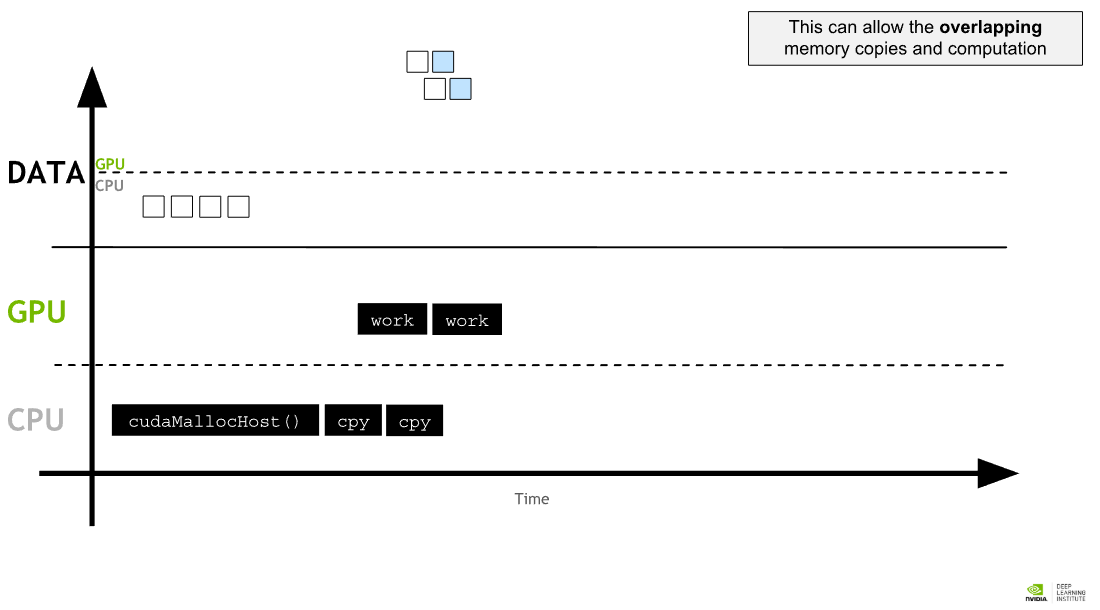
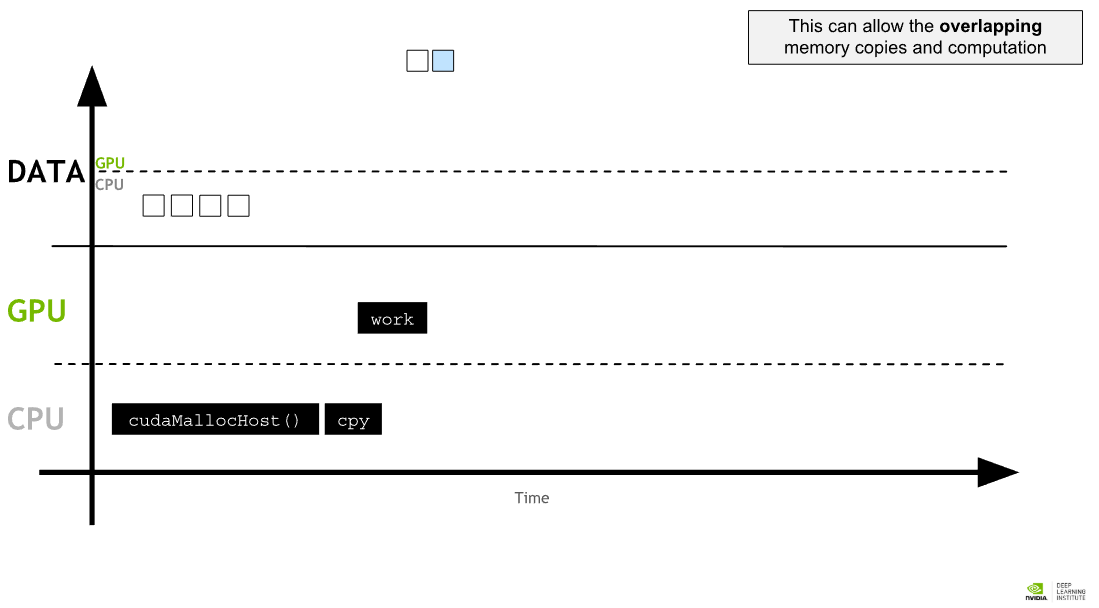
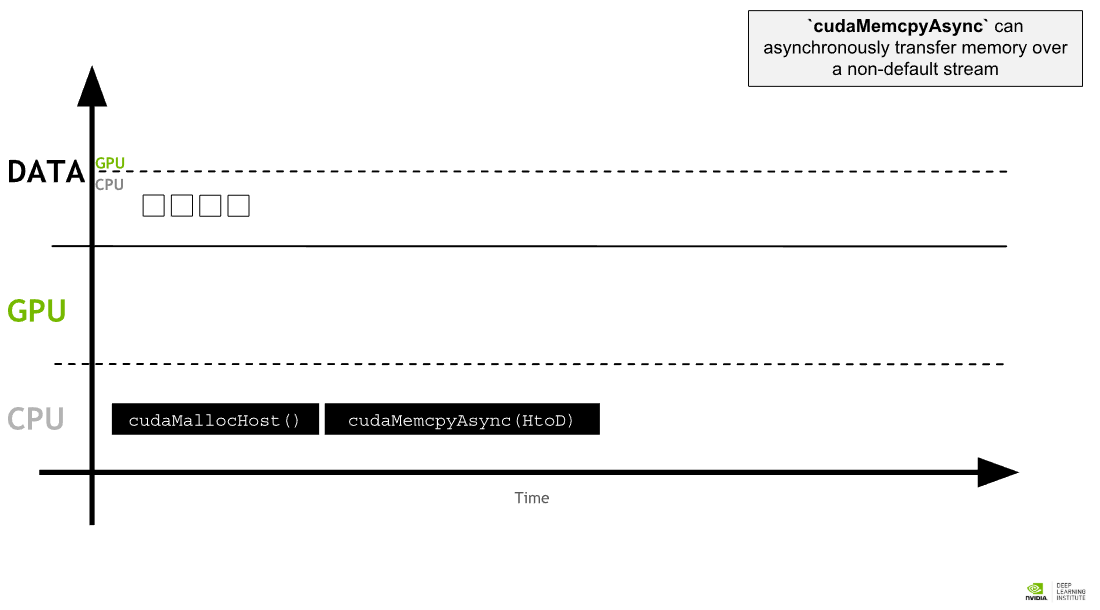
verifyOnHost(host\_a, N);

cudaFree(device\_a);

cudaFreeHost(host\_a); // Free pinned memory like this.

### ff

## Using Streams to Overlap Data Transfers and Code Execution



In addition to cudaMemcpy is cudaMemcpyAsync which can asynchronously copy memory either from host to device or from device to host as long as the host memory is pinned, which can be done by allocating it with cudaMallocHost.

Similar to kernel execution, cudaMemcpyAsync is only asynchronous by default with respect to the host. It executes, by default, in the default stream and therefore is a blocking operation with regard to other CUDA operations occuring on the GPU. The cudaMemcpyAsync function, however, takes as an optional 5th argument, a non-default stream. By passing it a non-default stream, the memory transfer can be concurrent to other CUDA operations occuring in other non-default streams.

A common and useful pattern is to use a combination of pinned host memory, asynchronous memory copies in non-default streams, and kernel executions in non-default streams, to overlap memory transfers with kernel execution.

In the following example, rather than wait for the entire memory copy to complete before beginning work on the kernel, segments of the required data are copied and worked on, with each copy/work segment running in its own non-default stream. Using this technique, work on parts of the data can begin while memory transfers for later segments occur concurrently. Extra care must be taken when using this technique to calculate segment-specific values for the number of operations, and the offset location inside arrays, as shown here:

int N = 2<<24;

int size = N \* sizeof(int);

int \*host\_array;

int \*device\_array;

cudaMallocHost(&host\_array, size); // Pinned host memory allocation.

cudaMalloc(&device\_array, size); // Allocation directly on the active GPU device.

initializeData(host\_array, N); // Assume this application needs to initialize on the host.

const int numberOfSegments = 4; // This example demonstrates slicing the work into 4 segments.

int segmentN = N / numberOfSegments; // A value for a segment's worth of `N` is needed.

size\_t segmentSize = size / numberOfSegments; // A value for a segment's worth of `size` is needed.

// For each of the 4 segments...

for (int i = 0; i < numberOfSegments; ++i)

{

// Calculate the index where this particular segment should operate within the larger arrays.

segmentOffset = i \* segmentN;

// Create a stream for this segment's worth of copy and work.

cudaStream\_t stream;

cudaStreamCreate(&stream);

// Asynchronously copy segment's worth of pinned host memory to device over non-default stream.

cudaMemcpyAsync(&device\_array[segmentOffset], // Take care to access correct location in array.

&host\_array[segmentOffset], // Take care to access correct location in array.

segmentSize, // Only copy a segment's worth of memory.

cudaMemcpyHostToDevice,

stream); // Provide optional argument for non-default stream.

// Execute segment's worth of work over same non-default stream as memory copy.

kernel<<<number\_of\_blocks, threads\_per\_block, 0, stream>>>(&device\_array[segmentOffset], segmentN);

// `cudaStreamDestroy` will return immediately (is non-blocking), but will not actually destroy streamuntil all stream operations are complete.

cudaStreamDestroy(stream);

}