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.

Prerequisites

To get the most out of this lab you should already be able to:

- Write, compile, and run C/C++ programs that both call CPU functions and launch GPU kernels.
- Control parallel thread hierarchy using execution configuration.
- Refactor serial loops to execute their iterations in parallel on a GPU.
- Allocate and free CUDA Unified Memory.
- Understand the behavior of Unified Memory with regard to page faulting and data migrations.
- Use asynchronous memory prefetching to reduce page faults and data migrations.

Objectives

By the time you complete this lab you will be able to:

- Use the **NVIDIA Visual Profiler** (**nvvp**) to visually profile the timeline of GPU-accelerated CUDA applications.
- Use nvvp to identify, and exploit, optimization opportunities in GPU-accelerated CUDA applications.
- Utilize CUDA streams for concurrent kernel execution in accelerated applications.
- (**Optional Advanced Content**) Use manual memory allocation, including allocating pinned memory, in order to asynchronously transfer data in concurrent CUDA streams.

Setting Up the NVIDIA Visual Profiler

Click on this link to nvvp (/novnc) to open an nvvp session in another tab. Use the password `cuda` and connect, and you will have access to nvvp. In the next section you will start using it to profile CUDA code.

NOTE: Users with Windows based touch screen laptops might experience some issues using nvvp. If this is the case, you can resolve the issue by using the [Firefox web browser].

If you are asked to use a workspace, simply accept the selected default workspace. nvvp will open automatically shortly thereafter.

If at anytime in the lab your connection to nvvp times out, you should be able to simply click the connect button to reconnect.

Comparing Code Refactors Iteratively with nvvp

The following series of exercises will serve to familiarize you in interacting with the nvvp timeline. You will be profiling a series of programs that were iteratively improved using techniques that are already familiar to you. Each time you profile, information in the timeline will give information supporting the next iteration. This will further increase your understanding of how various CUDA programming techniques affect application performance.

After completing this series of exercises, you will utilize the nvvp timeline to support your learning new CUDA programming techniques that involve the use of concurrent CUDA streams, and using non-managed CUDA memory allocation and copy techniques.

Exercise: Examine Timeline of Compiled CUDA Code

`01-vector-add.cu` (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/01-vector-add/01-vector-add.cu) (<-------- click on these links to source files to edit them in the browser) contains a working, accelerated, vector addition application. Use the code execution cell directly below (you can execute it, and any of the code execution cells in this lab by `CTRL` + clicking it) to compile and run it. You should see a message printed that indicates it was successful.

In [ ]:
```
!nvcc -arch sm_70 -o vector-add-no-prefetch 01-vector-add/01-vector-add.cu -run
```

Once you have successfully compiled the application, open the compiled executable with nvvp (/novnc).

In order to find the executable to open in nvvp, you will first need to click the “Browse…” button for the “File” form option, and then, on the left hand side of the menu that opens, the “File System” button. At that point, the files are at the following path: `/dli/tasks/task1/task/03_AC_STREAMS_NVVP/` . Fortunately, once you have located the executable at this path, further attempts to open nvvp sessions will already be in the relevant directory.

- Create a timeline ruler that will show the execution time of the `addVectorsInto` kernel.
- Identify the locations in the application timeline when there are CPU page faults. Identify where in the application source code (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/01-vector-add/01-vector-add.cu) these CPU page faults are being caused.
- Locate the *Data Migration (DtoH)* (Device to Host) events in the timeline. These are occuring around the same time as the CPU page faults after the kernel executes. Why do they occur here, but not during the CPU page faults prior to the kernel's execution?

- What is the relationship on the timeline between GPU page faults, HtoD data migration events, and the execution of the `addVectorsInto` kernel? Can you articulate, referring to the source code, why these are occuring in the way that they are?

Exercise: Add Asynchronous Prefetching

`01-vector-add-prefetch-solution.cu` (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/01-vector-add/solutions/01-vector-add-prefetch-solution.cu) refactors the vector addition application from above so that the 3 vectors needed by its `addVectorsInto` kernel are asynchronously prefetched to the active GPU device prior to launching the kernel (using `cudaMemPrefetchAsync` (http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge8dc9199943d421bc8bc7f473df12e42)). Open the source code and identify where in the application these changes were made.

After reviewing the changes, compile and run the refactored application using the code execution cell directly below. You should see its success message printed.

Exercise: Compare the Timelines of Prefetching vs. Non-Prefetching

`01-vector-add-prefetch-solution.cu` (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/01-vector-add/solutions/01-vector-add-prefetch-solution.cu) refactors the vector addition application from above so that the 3 vectors needed by its `addVectorsInto` kernel are asynchronously prefetched to the active GPU device prior to launching the kernel (using `cudaMemPrefetchAsync` (http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge8dc9199943d421bc8bc7f473df12e42)). Open the source code and identify where in the application these changes were made.

After reviewing the changes, compile and run the refactored application using the code execution cell directly below. You should see its success message printed.

```
In [ ]: !nvcc -arch sm_70 -o vector-add-prefetch 01-vector-add/solutions/01-vector-add-prefetch-solution.cu -run
```

Open the compiled executable with nvvp (/novnc), leaving the previous session, with the vector add application prior to implementing prefetching, open. Maximize the timeline window, and do the following:

- Create a timeline ruler that will show the execution time of the `addVectorsInto` kernel. How does the execution time compare to that of the `addVectorsInto` kernel prior to adding asynchronous prefetching?
- Locate `cudaMemPrefetchAsync` in the *Runtime API* section of the timeline.
- Referring to the timeline for the application before the refactor, you can see in the *Unified Memory* section of the timeline, that there were several groupings of *GPU page faults* during the kernel execution, as unified memory became in demand. Now that you have implemented prefetching, do these page faults exist?
- Even though the GPU page faults have gone away, the data still needed to be migrated from the host to the device. Use the *Data Migration (HtoD)* section of the timeline to compare these migrations between the two applications. Compare how many of them there were, the amount of time they took to execute, and when they occured relative to the `addVectorsInto` kernel execution.
- Look at the overall runtimes of the 2 applications, how do they compare?

Exercise: Profile Refactor with Launch Init in Kernel

In the previous iteration of the vector addition application, the vector data is being initialized on the CPU, and therefore needs to be migrated to the GPU before the `addVectorsInto` kernel can operate on it.

The next iteration of the application, 01-init-kernel-solution.cu (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/02-init-kernel/solutions/01-init-kernel-solution.cu), the application has been refactored to initialize the data in parallel on the GPU.

Since the initialization now takes place on the GPU, prefetching has been done prior to initialization, rather than prior to the vector addition work. Review the source code to identify where these changes have been made.

After reviewing the changes, compile and run the refactored application using the code execution cell directly below. You should see its success message printed.

In [ ]:
```
!nvcc -arch=sm_70 -o init-kernel 02-init-kernel/solutions/01-init-kernel-solution.cu -run
```

Open the compiled executable in another session in nvvp and do the following:

- Create timeline rulers to measure the overall runtime of the application, the runtime of the `addVectorsInto` kernel, and also the runtime of the initialization kernel. Compare the application and `addVectorsInto` runtimes to the previous version of the application, how did they change?
- Look at the *Compute* section of the timeline. Which of the two kernels (`addVectorsInto` and the initialization kernel) is taking up the majority of the time on the GPU?

- Which of the following does your application contain?
    - CPU Page Faults
    - GPU Page Faults
    - Data Migration (HtoD)
    - Data Migration (DtoH)

Exercise: Profile Refactor with Asynchronous Prefetch Back to the Host

Currently, the vector addition application verifies the work of the vector addition kernel on the host. The next refactor of the application, 01-prefetch-check-solution.cu (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/04-prefetch-check/solutions/01-prefetch-check-solution.cu), asynchronously prefetches the data back to the host for verification.

After reviewing the changes, compile and run the refactored application using the code execution cell directly below. You should see its success message printed.

```
In [ ]: !nvcc -arch=sm_70 -o prefetch-to-host 04-prefetch-check/solutions/01-prefetch-check-solution.cu -run
```

Open this newly-compiled executable in nvvp, maximize the timeline, and do the following:

- Use the *Unified Memory* section of the timeline to compare and contrast the *Data Migration (DtoH)* events before and after adding prefetching back to the CPU.
    - How do the number of CPU Page Faults compare?
    - How does the amount of time DtoH migrations are taking overall compare?
- As a segue into the next section, look at the *Streams* section of the timeline. Notice that all kernel execution is occuring in the *Default* stream, and that execution of the kernels in the default stream happens serially. Streams will be covered beginning in the next section.

Concurrent CUDA Streams

The following slides present upcoming material visually, at a high level. Click through the slides before moving on to more detailed coverage of their topics in following sections.

In [1]:
```
%%HTML

<div align="center"><iframe src="https://docs.google.com/presentation/d/e/2PACX-1vRVgzpDzp5fWAu-Zpuyr09rmIqE4FTFESja
```

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:

Exercise: Predict Default Stream Behavior

The 01-print-numbers (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/05-stream-intro/01-print-numbers.cu) application has a very simple `printNumber` kernel which accepts an integer and prints it. The kernel is only being executed with a single thread inside a single block, however, it is being executed 5 times, using a for-loop, and passing each launch the number of the for-loop's iteration.

Compile and run 01-print-numbers (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/05-stream-intro/01-print-numbers.cu) using the code execution block below. You should see the numbers `0` through `4` printed.

In [ ]:
```
!nvcc -arch=sm_70 -o print-numbers 05-stream-intro/01-print-numbers.cu -run
```

Knowing that by default kernels are executed in the default stream, would you expect that the 5 launches of the `print-numbers` program executed serially, or in parallel? You should be able to mention two features of the default stream to support your answer. Open the executable in a new sesssion in nvvp, maximize the timeline, and zoom in on the kernel launches to confirm your answer.

Exercise: Implement Concurrent CUDA Streams

Both because all 5 kernel launches occured in the same stream, you should not be surprised to have seen that the 5 kernels executed serially. Additionally you could make the case that because the default stream is blocking, each launch of the kernel would wait to complete before the next launch, and this is also true.

Refactor 01-print-numbers (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/05-stream-intro/01-print-numbers.cu) so that each kernel launch occurs in its own non-default stream. Be sure to destroy the streams you create after they are no longer needed. Compile and run the refactored code with the code execution cell directly below. You should still see the numbers `0` through `4` printed, though not necessarily in ascending order. Refer to the solution (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/05-stream-intro/solutions/01-print-numbers-solution.cu) if you get stuck.

In [ ]:
```
!nvcc -arch=sm_70 -o print-numbers-in-streams 05-stream-intro/01-print-numbers.cu -run
```

Now that you are using 5 different non-default streams for each of the 5 kernel launches, do you expect that they will run serially or in parallel? In addition to what you now know about streams, take into account how trivial the `printNumber` kernel is, meaning, even if you predict parallel runs, will the speed at which one kernel will complete allow for complete overlap?

After hypothesizing, open a new nvvp session with the executable of your refactored program, maximize the timeline and zoom into kernel execution to view its actual behavior.

Exercise: Use Streams for Concurrent Data Initialization Kernels

The vector addition application you have been working with, 01-prefetch-check-solution.cu (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/04-prefetch-check/solutions/01-prefetch-check-solution.cu), currently launches an initialization kernel 3 times - once each for each of the 3 vectors needing initialization for the `vectorAdd` kernel. Refactor it to launch each of the 3 initialization kernel launches in their own non-default stream. You should still be see the success message print when compiling and running with the code execution cell below. Refer to the solution (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/06-stream-init/solutions/01-stream-init-solution.cu) if you get stuck.

In [ ]:
```
!nvcc -arch=sm_70 -o init-in-streams 04-prefetch-check/solutions/01-prefetch-check-solution.cu -run
```

Open the compiled binary in nvvp, maximize the timeline and confirm that your 3 initialization kernel launches are running in their own non-default streams, with some degree of concurrent overlap.

Summary

At this point in the lab you are able to:

- Use the **NVIDIA Visual Profiler** (**`nvvp`**) to visually profile the timeline of GPU-accelerated CUDA applications.
- Use nvvp to identify, and exploit, optimization opportunities in GPU-accelerated CUDA applications.
- Utilize CUDA streams for concurrent kernel execution in accelerated applications.

At this point in time you have a wealth of fundamental tools and techniques for accelerating CPU-only applications, and for then optimizing those accelerated applications. In the final exercise, you will have a chance to apply everything that you've learned to accelerate an n-body (https://en.wikipedia.org/wiki/N-body_problem) simulator, which predicts the individual motions of a group of objects interacting with each other gravitationally.

Final Exercise: Accelerate and Optimize an N-Body Simulator

An n-body (https://en.wikipedia.org/wiki/N-body_problem) simulator predicts the individual motions of a group of objects interacting with each other gravitationally. 01-nbody.cu (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/09-nbody/01-nbody.cu) contains a simple, though working, n-body simulator for bodies moving through 3 dimensional space. The application can be passed a command line argument to affect how many bodies are in the system.

In its current CPU-only form, working on 4096 bodies, this application is able to calculate about 30 million interactions between bodies in the system per second. Your task is to:

- GPU accelerate the program, retaining the correctness of the simulation
- Work iteratively to optimize the simulator so that it calculates over 30 billion interactions per second while working on 4096 bodies `(2<<11)`
- Work iteratively to optimize the simulator so that it calculates over 325 billion interactions per second while working on ~65,000 bodies `(2<<15)`

After completing this, go back in the browser page you used to open this notebook and click the Assess button. If you have retained the accuracy of the application and accelerated it to the specifications above, you will receive a certification for your competency in the *Fundamentals of Accelerating Applications with CUDA C/C++*.

Considerations to Guide Your Work

Here are some things to consider before beginning your work:

- Especially for your first refactors, the logic of the application, the `bodyForce` function in particular, can and should remain largely unchanged: focus on accelerating it as easily as possible.
- You will not be able to accelerate the `randomizeBodies` function since it is using the `rand` function, which is not available on GPU devices. `randomizeBodies` a host function. Do not touch it at all.
- The codebase contains a for-loop inside `main` for integrating the interbody forces calculated by `bodyForce` into the positions of the bodies in the system. This integration both needs to occur after `bodyForce` runs, and, needs to complete before the next call to `bodyForce`. Keep this in mind when choosing how and where to parallelize.
- Use a profile driven and iterative approach.
- You are not required to add error handling to your code, but you might find it helpful, as you are responsible for your code working correctly.

Have Fun!

In [ ]:
```
!nvcc -arch=sm_70 -o nbody 09-nbody/01-nbody.cu
```

In [ ]:
```
!./nbody 11 # This argument is passed as `N` in the formula `2<<N`, to determine the number of bodies in the system
```

In [ ]:
```
!nvprof ./nbody
```

Advanced Content

The following sections, for those of you with time and interest, introduce more intermediate techniques involving some manual memory management, and using non-default streams to overlap kernel execution and memory copies.

After learning about each of the techniques below, try to further optimize your nbody simulation using these techniques.

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 c
opy.
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.
```

Exercise: Manually Allocate Host and Device Memory

The most recent iteration of the vector addition application, 01-stream-init-solution (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/06-stream-init/solutions/01-stream-init-solution.cu), is using `cudaMallocManaged` to allocate managed memory first used on the device by the initialization kernels, then on the device by the vector add kernel, and then by the host, where the memory is automatically transfered, for verification. This is a sensible approach, but it is worth experimenting with some manual memory allocation and copying to observe its impact on the application's performance.

Refactor the 01-stream-init-solution (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/06-stream-init/solutions/01-stream-init-solution.cu) application to **not** use `cudaMallocManaged`. In order to do this you will need to do the following:

- Replace calls to `cudaMallocManaged` with `cudaMalloc`.

- Create an additional vector that will be used for verification on the host. This is required since the memory allocated with `cudaMalloc` is not available to the host. Allocate this host vector with `cudaMallocHost` .
- After the `addVectorsInto` kernel completes, use `cudaMemcpy` to copy the vector with the addition results, into the host vector you created with `cudaMallocHost` .
- Use `cudaFreeHost` to free the memory allocated with `cudaMallocHost` .

Refer to the solution (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/07-manual-malloc/solutions/01-manual-malloc-solution.cu) if you get stuck.

In [ ]:
```
!nvcc -arch=sm_70 -o vector-add-manual-alloc 06-stream-init/solutions/01-stream-init-solution.cu -run
```

After completing the refactor, open the executable in a new nvvp session, and use the timeline to do the following:

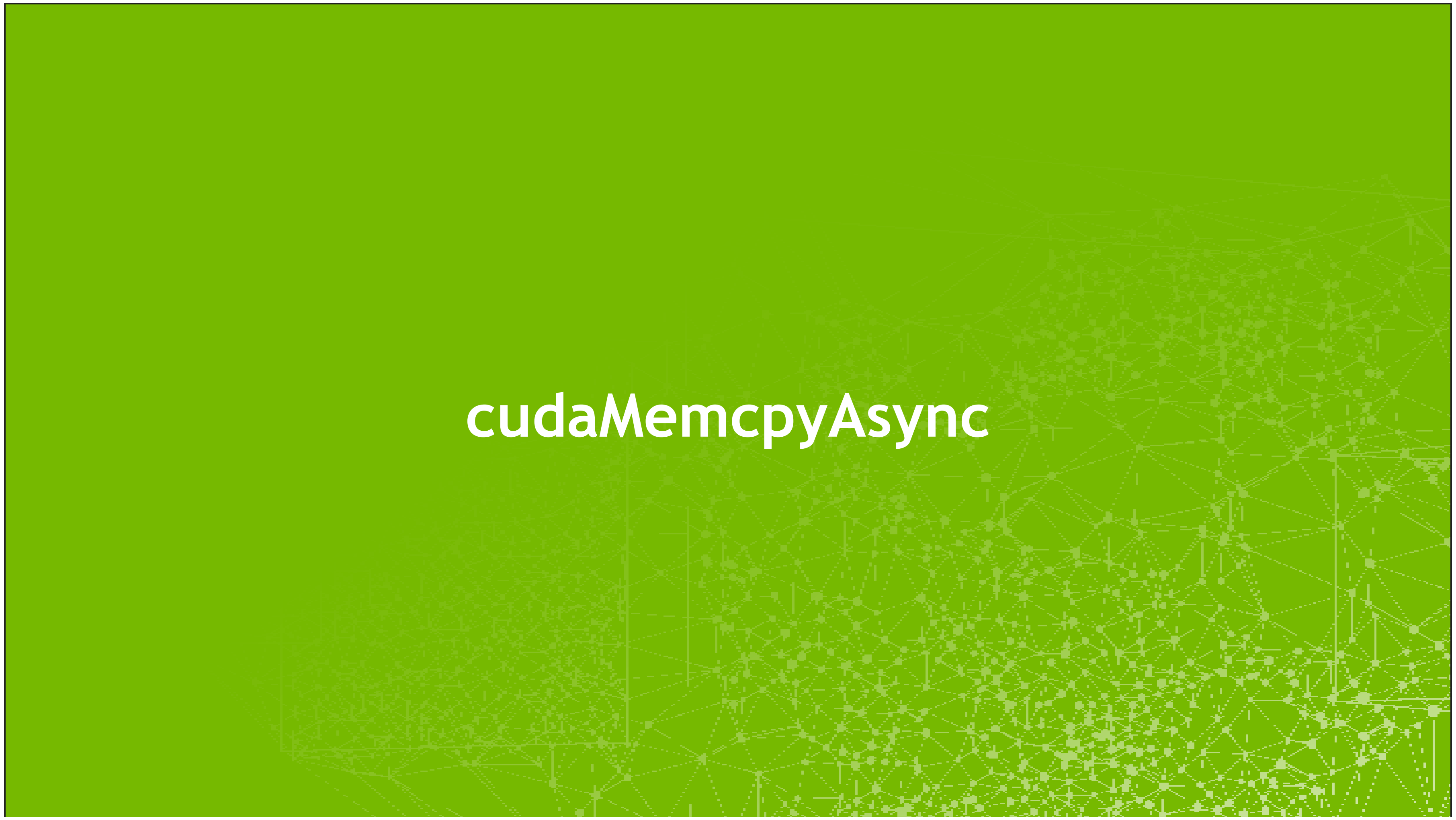
- Notice that there is no longer a *Unified Memory* section of the timeline.
- Comparing this timeline to that of the previous refactor, use timeline rulers to compare the runtimes of `cudaMalloc` in the current application vs. `cudaMallocManaged` in the previous.
- Notice how in the current application, work on the initialization kernels does not start until a later time than it did in the previous iteration. Examination of the timeline will show the difference is the time taken by `cudaMallocHost` . This clearly points out the difference between memory transfers, and memory copies. When copying memory, as you are doing presently, the data will exist in 2 different places in the system. In the current case, the allocation of the 4th host-only vector incurs a small cost in performance, compared to only allocating 3 vectors in the previous iteration.

Using Streams to Overlap Data Transfers and Code Execution

The following slides present upcoming material visually, at a high level. Click through the slides before moving on to more detailed coverage of their topics in following sections.

In [6]:
```
%%HTML

<div align="center"><iframe src="https://docs.google.com/presentation/d/e/2PACX-1vQdHDR62S4hhvq02CZreC_Hvb9y89_IRIKt
```



cudaMemcpyAsync

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 segmen
ts.
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 stream unt
il
  // all stream operations are complete.
  cudaStreamDestroy(stream);
}
```

Exercise: Overlap Kernel Execution and Memory Copy Back to Host

The most recent iteration of the vector addition application, 01-manual-malloc-solution.cu (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/07-manual-malloc/solutions/01-manual-malloc-solution.cu), is currently performing all of its vector addition work on the GPU before copying the memory back to the host for verification.

Refactor 01-manual-malloc-solution.cu (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/07-manual-malloc/solutions/01-manual-malloc-solution.cu) to perform the vector addition in 4 segments, in non-default streams, so that asynchronous memory copies can begin before waiting for all vector addition work to complete. Refer to the solution (../../../../../edit/tasks/task1/task/03_AC_STREAMS_NVVP/08-overlap-xfer/solutions/01-overlap-xfer-solution.cu) if you get stuck.

In [ ]:
```
!nvcc -arch=sm_70 -o vector-add-manual-alloc 07-manual-malloc/solutions/01-manual-malloc-solution.cu -run
```

After completing the refactor, open the executable in a new nvvp session, and use the timeline to do the following:

- Note when the device to host memory transfers begin, is it before or after all kernel work has completed?
- Notice that the 4 memory copy segments themselves do not overlap. Even in separate non-default streams, only one memory transfer in a given direction (DtoH here) at a time can occur simultaneously. The performance gains here are in the ability to start the transfers earlier than otherwise, and it is not hard to imagine in an application where a less trivial amount of work was being done compared to a simple addition operation, that the memory copies would not only start earlier, but also overlap with kernel execution.