Asynchronous Streaming, and Visual Profiling with CUDA C/C++

CUDA

The CUDA toolkit ships with the **Nsight Systems**, a powerful GUI application to support the development of accelerated CUDA applications. Nsight Systems 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**.

In this lab, you will be using the Nsight Systems 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 particle 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 **Nsight Systems** to visually profile the timeline of GPU-accelerated CUDA applications.
- Use Nsight Systems 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 device memory allocation, including allocating pinned memory, in order to asynchronously transfer data in concurrent CUDA streams.

Running Nsight Systems

For this interactive lab environment, we have set up a remote desktop you can access from your browser, where you will be able to launch and use Nsight Systems.

You will begin by creating a report file for an already-existing vector addition program, after which you will be walked through a series of steps to open this report file in Nsight Systems, and to make the visual experience nice.

Generate Report File

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 [5]:
```
!nvcc -o vector-add-no-prefetch 01-vector-add/01-vector-add.cu -run
```

```
Success! All values calculated correctly.
```

Next, use `nsys profile --stats=true` to create a report file that you will be able to open in the Nsight Systems visual profiler. Here we use the `-o` flag to give the report file a memorable name:

In [6]:
```
!nsys profile --stats=true -o vector-add-no-prefetch-report ./vector-add-no-prefetch
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-
tree.
Collecting data...
Success! All values calculated correctly.
Processing events...
Saving temporary "/tmp/nsys-report-f265-10a1-ab68-76c3.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-f265-10a1-ab68-76c3.qdrep"
Exporting 10077 events: [=================================================100%]

Exported successfully to
/tmp/nsys-report-f265-10a1-ab68-76c3.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum    Maximum           Name
 -------  ---------------  ---------  ----------  --------  ---------  ---------------------
    60.7        148298522          3  49432840.7     13951  148254781  cudaMallocManaged
    31.0         75803281          1  75803281.0  75803281   75803281  cudaDeviceSynchronize
     8.3         20250826          3   6750275.3   5993438    7840882  cudaFree
     0.0            51681          1     51681.0     51681      51681  cudaLaunchKernel



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average    Minimum   Maximum                     Name
 -------  ---------------  ---------  ----------  --------  --------  ------------------------------------------
   100.0         75794123          1  75794123.0  75794123  75794123  addVectorsInto(float*, float*, float*, int)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum              Operation
 -------  ---------------  ----------  -------  -------  -------  ---------------------------------
    80.0         43237264        8203   5270.9     1822    79360  [CUDA Unified Memory memcpy HtoD]
    20.0         10837032         768  14110.7     1375    80416  [CUDA Unified Memory memcpy DtoH]
```

```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average  Minimum  Maximum              Operation
 ----------  ----------  -------  -------  --------  ---------------------------------
 393216.000        8203   47.936    4.000  1012.000  [CUDA Unified Memory memcpy HtoD]
 131072.000         768  170.667    4.000  1020.000  [CUDA Unified Memory memcpy DtoH]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    86.4       1777406599         98  18136802.0    24720  100140281  poll
     9.2        189426307         86   2202631.5    14860   20414149  sem_timedwait
     2.3         46375028        621     74678.0     1030   12386334  ioctl
     1.0         20165135         23    876745.0     1130    7806771  mmap
     1.0         19868203         23    863834.9     1140   19780418  fopen
     0.1          1935496         64     30242.1     2480     567880  mmap64
     0.0           811893         76     10682.8     3761      45091  open64
     0.0           140994         11     12817.6     8310      25521  write
     0.0           139843          4     34960.8    27721      45440  pthread_create
     0.0            48211          1     48211.0    48211      48211  fgets
     0.0            36701         12      3058.4     1380       5160  munmap
     0.0            26631          5      5326.2     2470       8320  open
     0.0            15090          3      5030.0     1170       9880  fread
     0.0            13640          3      4546.7     1390       7610  fgetc
     0.0            11620          6      1936.7     1070       3040  read
     0.0            11090          6      1848.3     1040       3570  fclose
     0.0            11080          2      5540.0     3060       8020  socket
     0.0             7820          1      7820.0     7820       7820  connect
     0.0             7290          3      2430.0     1060       5080  fcntl
     0.0             7190          1      7190.0     7190       7190  pipe2
     0.0             3700          1      3700.0     3700       3700  sem_wait
     0.0             1770          1      1770.0     1770       1770  bind

Report file moved to "/dli/task/vector-add-no-prefetch-report.qdrep"
Report file moved to "/dli/task/vector-add-no-prefetch-report.sqlite"
```

Open the Remote Desktop

Run the next cell to generate a link to the remote desktop. Then, read the instructions that follow in the notebook.

In [7]:
```
%%js
var port = ((window.location.port == 80) ? "" : (":"+window.location.port));
var url = 'http://' + window.location.hostname + port + '/nsight/vnc.html?resize=scale';
let a = document.createElement('a');
a.setAttribute('href', url)
a.setAttribute('target', '_blank')
a.innerText = 'Click to open remote desktop'
element.append(a);
```

After clicking the *Connect* button you will be asked for a password, which is `nvidia`.

Open Nsight Systems

To open Nsight Systems, double-click the "NVIDIA Nsight Systems" icon on the remote desktop.

open nsight

Enable Usage Reporting

When prompted, click "Yes" to enable usage reporting:

enable usage

Select GPU Rows on Top

When prompted, select *GPU Rows on Top* and then click *Okay*.

gpu)_rows_on_top

Open the Report File

Open this report file by visiting *File* -> *Open* from the Nsight Systems menu and select `vector-add-no-prefetch-report.qdrep`:

open-report

Ignore Warnings/Errors

You can close and ignore any warnings or errors you see, which are just a result of our particular remote desktop environment:

ignore errors

Make More Room for the Timelines

To make your experience nicer, full-screen the profiler, close the *Project Explorer* and hide the *Events View*:

make nice

Your screen should now look like this:

now nice

Expand the CUDA Unified Memory Timelines

Next, expand the *CUDA* -> *Unified memory* and *Context* timelines, and close the *Threads* timelines:

open memory

Observe Many Memory Transfers

From a glance you can see that your application is taking about 1 second to run, and that also, during the time when the `addVectorsInto` kernel is running, that there is a lot of UM memory activity:

memory and kernel

Zoom into the memory timelines to see more clearly all the small memory transfers being caused by the on-demand memory page faults. A couple tips:

1. You can zoom in and out at any point of the timeline by holding `CTRL` while scrolling your mouse/trackpad
2. You can zoom into any section by click + dragging a rectangle around it, and then selecting *Zoom in*

Here's an example of zooming in to see the many small memory transfers:

many transfers

Comparing Code Refactors Iteratively with Nsight Systems

Now that you have Nsight Systems up and running and are comfortable moving around the timelines, you will be profiling a series of programs that were iteratively improved using techniques already familiar to you. Each time you profile, information in the timeline will give information supporting how you should next modify your code. Doing this will further increase your understanding of how various CUDA programming techniques affect application performance.

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

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`). 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 [8]:
```
!nvcc -o vector-add-prefetch 01-vector-add/solutions/01-vector-add-prefetch-solution.cu -run
```

```
Success! All values calculated correctly.
```

Now create a report file for this version of the application:

In [9]:
```
!nsys profile --stats=true -o vector-add-prefetch-report ./vector-add-prefetch
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-
tree.
Collecting data...
Success! All values calculated correctly.
Processing events...
Saving temporary "/tmp/nsys-report-a3af-bbd9-98f1-faa4.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-a3af-bbd9-98f1-faa4.qdrep"
Exporting 2082 events: [===================================================100%]

Exported successfully to
/tmp/nsys-report-a3af-bbd9-98f1-faa4.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum    Maximum           Name
 -------  ---------------  ---------  ----------  --------  ---------  ---------------------
    74.0        152144398          3  50714799.3     14610  152099457  cudaMallocManaged
     9.6         19673682          3   6557894.0    136683   10809113  cudaMemPrefetchAsync
     9.6         19646641          3   6548880.3   5849574    7438613  cudaFree
     6.8         14056831          1  14056831.0  14056831   14056831  cudaDeviceSynchronize
     0.0            32100          1     32100.0     32100      32100  cudaLaunchKernel



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum                    Name
 -------  ---------------  ---------  --------  -------  -------  ------------------------------------------
   100.0           849855          1  849855.0   849855   849855  addVectorsInto(float*, float*, float*, int)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average   Minimum  Maximum              Operation
 -------  ---------------  ----------  --------  -------  -------  ---------------------------------
    73.9         30536557         192  159044.6   158720   159487  [CUDA Unified Memory memcpy HtoD]
    26.1         10807623         768   14072.4     1374    78559  [CUDA Unified Memory memcpy DtoH]
```

```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average   Minimum   Maximum              Operation
 ----------  ----------  --------  --------  --------  --------------------------------
 393216.000         192  2048.000  2048.000  2048.000  [CUDA Unified Memory memcpy HtoD]
 131072.000         768   170.667     4.000  1020.000  [CUDA Unified Memory memcpy DtoH]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    85.1       1727067489         95  18179657.8    24950  100133139  poll
     9.0        182675555         81   2255253.8    14750   20415335  sem_timedwait
     3.8         77151711        630    122463.0     1040   10719812  ioctl
     1.0         19525661         23    848941.8     1120    7389342  mmap
     0.9         18299666         21    871412.7     1300   18182735  fopen
     0.1          2045267          3    681755.7    14580    1983666  sem_wait
     0.1          1959180         64     30612.2     2680     574220  mmap64
     0.0           803016         76     10566.0     5130      45471  open64
     0.0           202644          5     40528.8    30231      60261  pthread_create
     0.0           164580         12     13715.0     8590      28150  write
     0.0            57101          1     57101.0    57101      57101  fgets
     0.0            50992         13      3922.5     1380      15530  munmap
     0.0            32410          5      6482.0     3570      11580  open
     0.0            24581         12      2048.4     1060       3390  fclose
     0.0            16531          4      4132.8     1020       9540  fgetc
     0.0            16380          8      2047.5     1060       3180  read
     0.0            12740          2      6370.0     3810       8930  socket
     0.0             9100          3      3033.3     1120       5010  fread
     0.0             9090          1      9090.0     9090       9090  connect
     0.0             7480          1      7480.0     7480       7480  pipe2
     0.0             5440          2      2720.0     1080       4360  fcntl
     0.0             1960          1      1960.0     1960       1960  bind
     0.0             1030          1      1030.0     1030       1030  listen

Report file moved to "/dli/task/vector-add-prefetch-report.qdrep"
Report file moved to "/dli/task/vector-add-prefetch-report.sqlite"
```

Open the report in Nsight Systems, leaving the previous report open for comparison.

- How does the execution time compare to that of the `addVectorsInto` kernel prior to adding asynchronous prefetching?

- Locate `cudaMemPrefetchAsync` in the *CUDA API* section of the timeline.
- How have the memory transfers changed?

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, 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 [10]:
```
!nvcc -o init-kernel 02-init-kernel/solutions/01-init-kernel-solution.cu -run
```

```
Success! All values calculated correctly.
```

Now create a report file for this version of the application:

In [11]:
```
!nsys profile --stats=true -o init-kernel-report ./init-kernel
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-
tree.
Collecting data...
Success! All values calculated correctly.
Processing events...
Saving temporary "/tmp/nsys-report-d8ba-a084-fb9b-235f.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-d8ba-a084-fb9b-235f.qdrep"
Exporting 1769 events: [==================================================100%]

Exported successfully to
/tmp/nsys-report-d8ba-a084-fb9b-235f.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  ---------------------
    88.6        162006472          3  54002157.3    20910  161905280  cudaMallocManaged
     9.2         16911395          3   5637131.7  4178913    8445537  cudaFree
     1.2          2166428          3    722142.7   696513     762653  cudaMemPrefetchAsync
     0.9          1684859          1   1684859.0  1684859    1684859  cudaDeviceSynchronize
     0.0            59541          4     14885.3     5900      37241  cudaLaunchKernel



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum                    Name
 -------  ---------------  ---------  --------  -------  -------  -------------------------------------------
    50.2           856575          1  856575.0   856575   856575  addVectorsInto(float*, float*, float*, int)
    49.8           848478          3  282826.0   280128   284447  initWith(float, float*, int)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum              Operation
 -------  ---------------  ----------  -------  -------  -------  ---------------------------------
   100.0         10865936         768  14148.4     1439    80704  [CUDA Unified Memory memcpy DtoH]
```

```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average  Minimum  Maximum              Operation
 ----------  ----------  -------  -------  --------  ---------------------------------
 131072.000         768  170.667    4.000  1020.000  [CUDA Unified Memory memcpy DtoH]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    75.2        481447946         32  15045248.3    30470  100139254  poll
    11.1         70866827         27   2624697.3    15871   20551968  sem_timedwait
     7.5         48115636        628     76617.3     1040    9147949  ioctl
     3.0         19433847         22    883356.7     1000   19324526  fopen
     2.6         16822413         24    700933.9     1010    8384286  mmap
     0.3          2009097         64     31392.1     2500     564999  mmap64
     0.1           807761         76     10628.4     3390      29821  open64
     0.0           163121          4     40780.3    30210      61641  pthread_create
     0.0           157142         11     14285.6    11040      19211  write
     0.0            48481          1     48481.0    48481      48481  fgets
     0.0            42830         12      3569.2     1790       7870  munmap
     0.0            28952          5      5790.4     2420       8501  open
     0.0            24250          4      6062.5     1820      15040  fread
     0.0            17170          7      2452.9     1020       5690  fclose
     0.0            15240          4      3810.0     1170       7550  fgetc
     0.0            13541          2      6770.5     5140       8401  socket
     0.0            13211          8      1651.4     1060       3670  read
     0.0             8860          1      8860.0     8860       8860  connect
     0.0             7480          1      7480.0     7480       7480  pipe2
     0.0             7040          2      3520.0     1040       6000  fcntl
     0.0             2290          1      2290.0     2290       2290  bind
     0.0             1190          1      1190.0     1190       1190  listen

Report file moved to "/dli/task/init-kernel-report.qdrep"
Report file moved to "/dli/task/init-kernel-report.sqlite"
```

Open the new report file in Nsight Systems and do the following:

- Compare the application and `addVectorsInto` run times to the previous version of the application, how did they change?

- Look at the *Kernels* 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?
    - 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, 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 [12]:
```
!nvcc -o prefetch-to-host 04-prefetch-check/solutions/01-prefetch-check-solution.cu -run
```

```
Success! All values calculated correctly.
```

Now create a report file for this version of the application:

In [13]:
```
!nsys profile --stats=true -o prefetch-to-host-report ./prefetch-to-host
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-
tree.
Collecting data...
Success! All values calculated correctly.
Processing events...
Saving temporary "/tmp/nsys-report-7cad-1f62-2474-6f32.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-7cad-1f62-2474-6f32.qdrep"
Exporting 1057 events: [===================================================100%]

Exported successfully to
/tmp/nsys-report-7cad-1f62-2474-6f32.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  ---------------------
    78.2        152915703          3  50971901.0    14301  152841041  cudaMallocManaged
    13.0         25351408          4   6337852.0   513809   23729960  cudaMemPrefetchAsync
     8.0         15610730          3   5203576.7  4143812    7150374  cudaFree
     0.9          1688789          1   1688789.0  1688789    1688789  cudaDeviceSynchronize
     0.0            48841          4     12210.3     4651      31710  cudaLaunchKernel



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum                     Name
 -------  ---------------  ---------  --------  -------  -------  -------------------------------------------
    50.4           858654          1  858654.0   858654   858654  addVectorsInto(float*, float*, float*, int)
    49.6           844350          3  281450.0   279135   282751  initWith(float, float*, int)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average   Minimum  Maximum              Operation
 -------  ---------------  ----------  --------  -------  -------  ---------------------------------
   100.0          9991305          64  156114.1   155935   156799  [CUDA Unified Memory memcpy DtoH]
```

```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average   Minimum   Maximum              Operation
 ----------  ----------  --------  --------  --------  --------------------------------
 131072.000          64  2048.000  2048.000  2048.000  [CUDA Unified Memory memcpy DtoH]




Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    68.2        341994799         27  12666474.0    20310  100129950  poll
    14.9         74909308        629    119092.7     1000   23685749  ioctl
    13.1         65860455         23   2863498.0    14431   20454453  sem_timedwait
     3.1         15520750         22    705488.6     1100    7120193  mmap
     0.4          1986120         64     31033.1     2940     583811  mmap64
     0.2           789856         76     10392.8     3220      38071  open64
     0.0           152551          4     38137.8    29050      51911  pthread_create
     0.0           142602         11     12963.8     8440      21060  write
     0.0           102110         22      4641.4     1130      19620  fopen
     0.0            48640          1     48640.0    48640      48640  fgets
     0.0            48130         11      4375.5     1300      16510  munmap
     0.0            27590          5      5518.0     2460       8480  open
     0.0            17930          4      4482.5     1390       8000  fgetc
     0.0            12850          6      2141.7     1130       3780  read
     0.0            12680          1     12680.0    12680      12680  sem_wait
     0.0            10990          4      2747.5     1700       3760  fread
     0.0            10880          7      1554.3     1030       2920  fclose
     0.0            10680          2      5340.0     3530       7150  socket
     0.0             9590          5      1918.0     1080       4520  fcntl
     0.0             6910          1      6910.0     6910       6910  pipe2
     0.0             6570          1      6570.0     6570       6570  connect
     0.0             1660          1      1660.0     1660       1660  bind
     0.0             1000          1      1000.0     1000       1000  listen

Report file moved to "/dli/task/prefetch-to-host-report.qdrep"
Report file moved to "/dli/task/prefetch-to-host-report.sqlite"
```

Open this report file in Nsight Systems, 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.

Concurrent CUDA Streams

You are now going to learn about a new concept, **CUDA Streams**. After an introduction to them, you will return to using Nsight Systems to better evaluate their impact on your application's performance.

The following video presents upcoming material visually, at a high level. Click watch it before moving on to more detailed coverage of their topics in following sections.

In [14]:

```
from IPython.display import HTML

video_url = "https://d36m44n9vdbmda.cloudfront.net/assets/s-ac-04-v1/task3/NVVP-Streams-1.mp4"

video_html = f"""
<video controls width="640" height="360">
    <source src="{video_url}" type="video/mp4">
    Your browser does not support the video tag.
</video>
"""

display(HTML(video_html))
```

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 in fact, 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 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 using the code execution block below. You should see the numbers `0` through `4` printed.

In [15]:
```
!nvcc -o print-numbers 05-stream-intro/01-print-numbers.cu -run
```

```
0
1
2
3
4
```

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. Create a report file in the cell below and open it in Nsight Systems to confirm your answer.

In [16]:

```
!nsys profile --stats=true -o print-numbers-report ./print-numbers
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-
tree.
Collecting data...
0
1
2
3
4
Processing events...
Saving temporary "/tmp/nsys-report-36c2-2d80-4fe1-80d9.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-36c2-2d80-4fe1-80d9.qdrep"
Exporting 941 events: [===================================================100%]

Exported successfully to
/tmp/nsys-report-36c2-2d80-4fe1-80d9.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  ---------------------
    99.9        136181700          5  27236340.0     4520  136158989  cudaLaunchKernel
     0.1           156602          1    156602.0   156602     156602  cudaDeviceSynchronize



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum        Name
 -------  ---------------  ---------  -------  -------  -------  ----------------
   100.0           148128          5  29625.6    28128    33728  printNumber(int)



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum  Maximum        Name
 -------  ---------------  ---------  ----------  -------  --------  --------------
    63.5        134499923         13  10346147.9    18841  45072014  poll
    25.3         53446117        625     85513.8     1000   9676686  ioctl
     9.5         20042816         21    954419.8     1240  19960393  fopen
```

```
     0.9         1914382          64    29912.2     2330    573389  mmap64
     0.4          775851          76    10208.6     3240     34130  open64
     0.1          282513          10    28251.3    14710     61231  sem_timedwait
     0.1          156783          12    13065.3     8460     23501  write
     0.1          148132           4    37033.0    33180     46450  pthread_create
     0.1          141422          17     8318.9     1410     33000  mmap
     0.0           72211           8     9026.4     1850     50501  munmap
     0.0           48681           1    48681.0    48681     48681  fgets
     0.0           36991           5     7398.2     2400     21511  open
     0.0           16500           3     5500.0     1180     10770  fread
     0.0           13150           3     4383.3     1310      7110  fgetc
     0.0           12640           1    12640.0    12640     12640  sem_wait
     0.0           12251           2     6125.5     3800      8451  socket
     0.0           11601           7     1657.3     1081      2330  read
     0.0           11500           6     1916.7     1040      3150  fclose
     0.0            8000           1     8000.0     8000      8000  connect
     0.0            7280           3     2426.7     1010      4990  fcntl
     0.0            7160           1     7160.0     7160      7160  pipe2
     0.0            1620           1     1620.0     1620      1620  bind

Report file moved to "/dli/task/print-numbers-report.qdrep"
Report file moved to "/dli/task/print-numbers-report.sqlite"
```

Exercise: Implement Concurrent CUDA Streams

Both because all 5 kernel launches occurred 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 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 if you get stuck.

In [17]:
```
!nvcc -o print-numbers-in-streams 05-stream-intro/01-print-numbers.cu -run
```

```
0
1
2
3
4
```

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 report file in Nsight Systems to view its actual behavior. You should notice that now, there are additional rows in the *CUDA* section for each of the non-default streams you created:

In [18]:
```
!nsys profile --stats=true -o print-numbers-in-streams-report print-numbers-in-streams
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-
tree.
Collecting data...
0
1
2
3
4
Processing events...
Saving temporary "/tmp/nsys-report-977f-b500-1785-02d4.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-977f-b500-1785-02d4.qdrep"
Exporting 965 events: [====================================================100%]

Exported successfully to
/tmp/nsys-report-977f-b500-1785-02d4.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  ---------------------
    99.1        147694977          5  29538995.4     2010  147683527  cudaStreamCreate
     0.9          1273422          5    254684.4     5670    1245131  cudaLaunchKernel
     0.0            69801          1     69801.0    69801      69801  cudaDeviceSynchronize
     0.0            23101          5      4620.2     2350       7371  cudaStreamDestroy



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum        Name
 -------  ---------------  ---------  -------  -------  -------  ----------------
   100.0           225854          5  45170.8    34048    58592  printNumber(int)



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum  Maximum        Name
 -------  ---------------  ---------  ----------  -------  --------  --------------
    63.3        147143478         13  11318729.1    32710  46507120  poll
```

```
    27.3         63486696        628    101093.5      1040    9829697  ioctl
     7.7         17820573         23    774807.5      1430   17712751  fopen
     0.9          1992916         64     31139.3      2570     571410  mmap64
     0.3           801351         76     10544.1      4591      33061  open64
     0.2           574259         10     57425.9     23170     231264  sem_timedwait
     0.1           170884         12     14240.3     11430      18320  write
     0.1           147832          4     36958.0     30540      48441  pthread_create
     0.1           124582         17      7328.4      1150      34131  mmap
     0.0            58041          1     58041.0     58041      58041  fgets
     0.0            39951          5      7990.2      1290      21131  fgetc
     0.0            31560          5      6312.0      3570       9440  open
     0.0            23822          7      3403.1      2260       4070  munmap
     0.0            19060         11      1732.7      1030       2920  fclose
     0.0            18960         13      1458.5      1010       3270  read
     0.0            17150          2      8575.0      6150      11000  socket
     0.0            14551          1     14551.0     14551      14551  pipe2
     0.0            11830          1     11830.0     11830      11830  connect
     0.0             5240          2      2620.0      2320       2920  fread
     0.0             4290          1      4290.0      4290       4290  fcntl
     0.0             3060          1      3060.0      3060       3060  bind
     0.0             1240          1      1240.0      1240       1240  listen

Report file moved to "/dli/task/print-numbers-in-streams-report.qdrep"
Report file moved to "/dli/task/print-numbers-in-streams-report.sqlite"
```

streams print

Exercise: Use Streams for Concurrent Data Initialization Kernels

The vector addition application you have been working with, 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 see the success message print when compiling and running with the code execution cell below. Refer to the solution if you get stuck.

In [19]:
```
!nvcc -o init-in-streams 04-prefetch-check/solutions/01-prefetch-check-solution.cu -run
```

```
Success! All values calculated correctly.
```

Open a report in Nsight Systems to confirm that your 3 initialization kernel launches are running in their own non-default streams, with some degree of concurrent overlap.

In [20]:

```
!nsys profile --stats=true -o init-in-streams-report ./init-in-streams
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-
tree.
Collecting data...
Success! All values calculated correctly.
Processing events...
Saving temporary "/tmp/nsys-report-0157-ab65-f1a5-23d6.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-0157-ab65-f1a5-23d6.qdrep"
Exporting 1069 events: [==================================================100%]

Exported successfully to
/tmp/nsys-report-0157-ab65-f1a5-23d6.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  ---------------------
    76.1        151833493          3  50611164.3    22320  151722872  cudaMallocManaged
    14.9         29726116          4   7431529.0   631191   27680462  cudaMemPrefetchAsync
     8.0         16011217          3   5337072.3  4127099    7631087  cudaFree
     0.8          1674147          1   1674147.0  1674147    1674147  cudaDeviceSynchronize
     0.0            65251          4     16312.8     7650      38371  cudaLaunchKernel
     0.0            60241          3     20080.3     4110      51001  cudaStreamDestroy
     0.0            33860          3     11286.7     2590      28290  cudaStreamCreate



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum                    Name
 -------  ---------------  ---------  --------  -------  -------  -------------------------------------------
    53.3           979517          3  326505.7   298911   351487  initWith(float, float*, int)
    46.7           857182          1  857182.0   857182   857182  addVectorsInto(float*, float*, float*, int)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average   Minimum  Maximum              Operation
 -------  ---------------  ----------  --------  -------  -------  ---------------------------------
   100.0         10023362          64  156615.0   155871   160319  [CUDA Unified Memory memcpy DtoH]
```

```
CUDA Memory Operation Statistics (by size in KiB):

   Total      Operations  Average   Minimum   Maximum               Operation
 ----------  ----------  --------  --------  --------  ---------------------------------
 131072.000          64  2048.000  2048.000  2048.000  [CUDA Unified Memory memcpy DtoH]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    66.1        341446469         27  12646165.5    29550  100142691  poll
    14.4         74604005        632    118044.3     1020   27613580  ioctl
    12.0         61756137         23   2685049.4    18960   20632574  sem_timedwait
     3.8         19690907         22    895041.2     1170   19586287  fopen
     3.1         15982909         22    726495.9     1440    7592067  mmap
     0.4          1935598         64     30243.7     2370     551979  mmap64
     0.1           765501         76     10072.4     3620      32750  open64
     0.0           149473          4     37368.3    30250      49251  pthread_create
     0.0           135632         11     12330.2     8690      16781  write
     0.0            50231          5     10046.2     2790      31080  open
     0.0            48770         13      3751.5     1370       7920  munmap
     0.0            48730          1     48730.0    48730      48730  fgets
     0.0            14230          8      1778.8     1280       3590  read
     0.0            14061          8      1757.6     1020       4500  fclose
     0.0            13900          4      3475.0     1190       6980  fgetc
     0.0            10732          2      5366.0     3061       7671  socket
     0.0            10290          4      2572.5     1770       3870  fread
     0.0             8930          4      2232.5     1000       5650  fcntl
     0.0             7660          1      7660.0     7660       7660  connect
     0.0             6500          1      6500.0     6500       6500  pipe2
     0.0             1570          1      1570.0     1570       1570  bind
     0.0             1210          1      1210.0     1210       1210  listen

Report file moved to "/dli/task/init-in-streams-report.qdrep"
Report file moved to "/dli/task/init-in-streams-report.sqlite"
```

Summary

At this point in the lab you are able to:

- Use the **Nsight Systems** to visually profile the timeline of GPU-accelerated CUDA applications.
- Use Nsight Systems 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 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 simulator predicts the individual motions of a group of objects interacting with each other gravitationally. 01-nbody.cu contains a simple, though working, n-body simulator for bodies moving through 3 dimensional space.

In its current CPU-only form, this application takes about 5 seconds to run on 4096 particles, and **20 minutes** to run on 65536 particles. Your task is to GPU accelerate the program, retaining the correctness of the simulation.

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.
- The code base 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!

Use this cell to compile the nbody simulator. Although it is initially a CPU-only application, is does accurately simulate the positions of the particles.

In [1]:
```
!nvcc -std=c++11 -o nbody 09-nbody/01-nbody.cu
```

It is highly recommended you use the profiler to assist your work. Execute the following cell to generate a report file:

In [2]:
```
!nsys profile --stats=true --force-overwrite=true -o nbody-report ./nbody
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-tree.
Collecting data...
4096 Bodies: average 35.810 Billion Interactions / second
Processing events...
Saving temporary "/tmp/nsys-report-5c1f-899e-ce56-15dc.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-5c1f-899e-ce56-15dc.qdrep"
Exporting 1057 events: [===================================================100%]

Exported successfully to
/tmp/nsys-report-5c1f-899e-ce56-15dc.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average      Minimum     Maximum            Name
 -------  ---------------  ---------  ------------  ----------  ----------  ---------------------
    99.9       3356582440          1  3356582440.0  3356582440  3356582440  cudaMallocManaged
     0.1          3829524         21      182358.3        1290     1024610  cudaDeviceSynchronize
     0.0           837505         20       41875.3        4330      734974  cudaLaunchKernel
     0.0            65711          1       65711.0       65711       65711  cudaFree



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum              Name
 -------  ---------------  ---------  --------  -------  -------  ---------------------------
    98.9          3708445         10  370844.5   297952  1021599  bodyForce(Body*, float, int)
     1.1            40157         10    4015.7     3936     4416  add(Body*, float, int)



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    96.4       3345904422         48  69706342.1    23020  100155388  poll
     1.8         62741156        620    101195.4     1030   13306282  ioctl
     1.2         41811401         13   3216261.6    15130   20459258  sem_timedwait
     0.5         16898101         24    704087.5     1731   16734037  fopen
```

```
     0.1         1975477          64     30866.8      2940       545740  mmap64
     0.0          796303          76     10477.7      2810        29331  open64
     0.0          184172          17     10833.6      1090        67881  mmap
     0.0          144743           4     36185.8     31370        44891  pthread_create
     0.0          127503          11     11591.2      8440        18301  write
     0.0           48431           1     48431.0     48431        48431  fgets
     0.0           29630           5      5926.0      2380         8000  open
     0.0           28341           9      3149.0      1470         4260  munmap
     0.0           21710           3      7236.7      2010        12940  fwrite
     0.0           21430           9      2381.1      1080         4650  fclose
     0.0           15040           4      3760.0      1250         7010  fgetc
     0.0           14810           2      7405.0      4400        10410  fopen64
     0.0           14111           2      7055.5      5220         8891  fread
     0.0           13160           2      6580.0      3820         9340  socket
     0.0           12050           7      1721.4      1100         3260  read
     0.0           10580           3      3526.7      1450         4630  fcntl
     0.0            7360           1      7360.0      7360         7360  connect
     0.0            5331           1      5331.0      5331         5331  pipe2
     0.0            4390           1      4390.0      4390         4390  fflush
     0.0            1980           1      1980.0      1980         1980  bind

Report file moved to "/dli/task/nbody-report.qdrep"
Report file moved to "/dli/task/nbody-report.sqlite"
```

Here we import a function that will run your `nbody` simulator against a various number of particles, checking for performance and accuracy.

In [3]:
```python
from assessment import run_assessment
```

Execute the following cell to run and assess `nbody`:

In [4]:
```python
run_assessment()
```

```
Running nbody simulator with 4096 bodies
----------------------------------------

Application should run faster than 0.9s
Your application ran in: 0.1157s
Your application reports  4096 Bodies: average 44.232 Billion Interactions / second

Your results are correct

Running nbody simulator with 65536 bodies
----------------------------------------

Application should run faster than 1.3s
Your application ran in: 0.2103s
Your application reports  65536 Bodies: average 430.530 Billion Interactions / second

Your results are correct

Congratulations! You passed the assessment!
See instructions below to generate a certificate, and see if you can accelerate the simulator even more!
```

Generate a Certificate

If you passed the assessment, please return to the course page (shown below) and click the "ASSESS TASK" button, which will generate your certificate for the course.

run_assessment

Advanced Content

The following sections, for those of you with time and interest, introduce more intermediate techniques involving some manual device 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 Device 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 device 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 device memory allocation and copy techniques, before extending these techniques to overlap data copies with computational work.

Here are some CUDA commands for manual device 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 Device 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.
```

Exercise: Manually Allocate Host and Device Memory

The most recent iteration of the vector addition application, 01-stream-init-solution, 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 transferred, for verification. This is a sensible approach, but it is worth experimenting with some manual device memory allocation and copying to observe its impact on the application's performance.

Refactor the 01-stream-init-solution 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 if you get stuck.

In [21]:
```
!nvcc -o vector-add-manual-alloc 06-stream-init/solutions/01-stream-init-solution.cu -run
```

```
Success! All values calculated correctly.
```

After completing the refactor, open a report in Nsight Systems, 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, compare the run times 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 video presents upcoming material visually, at a high level. Click watch it before moving on to more detailed coverage of their topics in following sections.

In [22]:
```python
from IPython.display import HTML

video_url = "https://d36m44n9vdbmda.cloudfront.net/assets/s-ac-04-v1/task3/NVVP-Streams-3.mp4"

video_html = f"""
<video controls width="640" height="360">
    <source src="{video_url}" type="video/mp4">
    Your browser does not support the video tag.
</video>
"""

display(HTML(video_html))
```

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 occurring 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 occurring 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 stream
until
  // 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, 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 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 if you get stuck.

In [23]:
```
!nvcc -o vector-add-manual-alloc 07-manual-malloc/solutions/01-manual-malloc-solution.cu -run
```

```
Success! All values calculated correctly.
```

After completing the refactor, open a report in Nsight Systems, 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.