Introduction to NVIDIA Profilers on Summit

Tom Papatheodore
Oak Ridge Leadership Computing Facility (OLCF)

Jeff Larkin
NVIDIA

Oak Ridge National Laboratory - April 11, 2019

Outline

- System Access & Local NVIDIA Toolkit Install
- Cloning Repository & Setting Up Environment
- A Simple Example: Vector Addition
- Jacobi Iteration
  - Serial
  - Single GPU
  - Single GPU (explicit data movement)
  - Multiple GPU (OpenMP + OpenACC)
- Redundant Matrix Multiply
  - Dealing with multiple MPI ranks
  - Basic annotation of CPU/GPU activities with NVTX
  - Unified Memory
- Remote kernel Analysis

System Access & Local NVIDIA Toolkit Install

If you want to follow along with the hands-on portions of this tutorial, you will need

- to have access to a Summit-like system
- to have a local install of NVIDIA Toolkit (v10+)

Access to the Ascent Training System

If you do not already have access to Summit, you can use the Ascent training system for this tutorial. Please visit the following url for instructions on how to do so:

https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/#obtaining-access-to-ascent

For the Project ID field, please use GEN121

In-Person Attendees Only!

Once you have access, you can login as follows:

`$ ssh USERNAME@login1.ascent.olcf.ornl.gov` (This will drop you into `/ccsopen/home/USERNAME`)

Local Installation of NVIDIA Toolkit (version 10+)

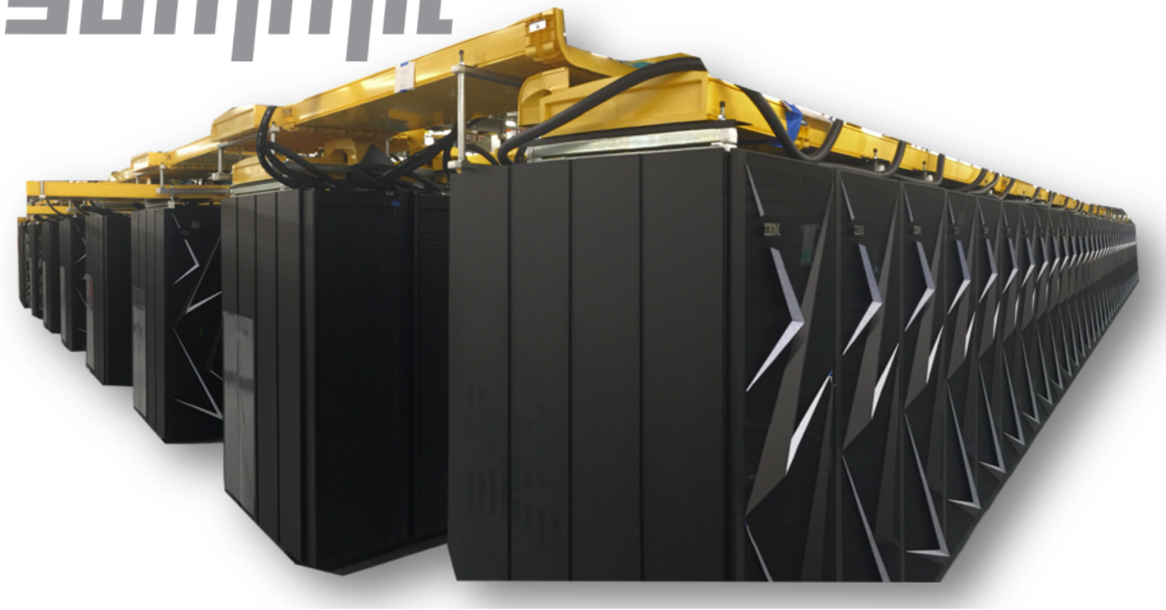
To ensure compatibility, please install NVIDIA Toolkit version 10+. Please visit the following url to download the toolkit:

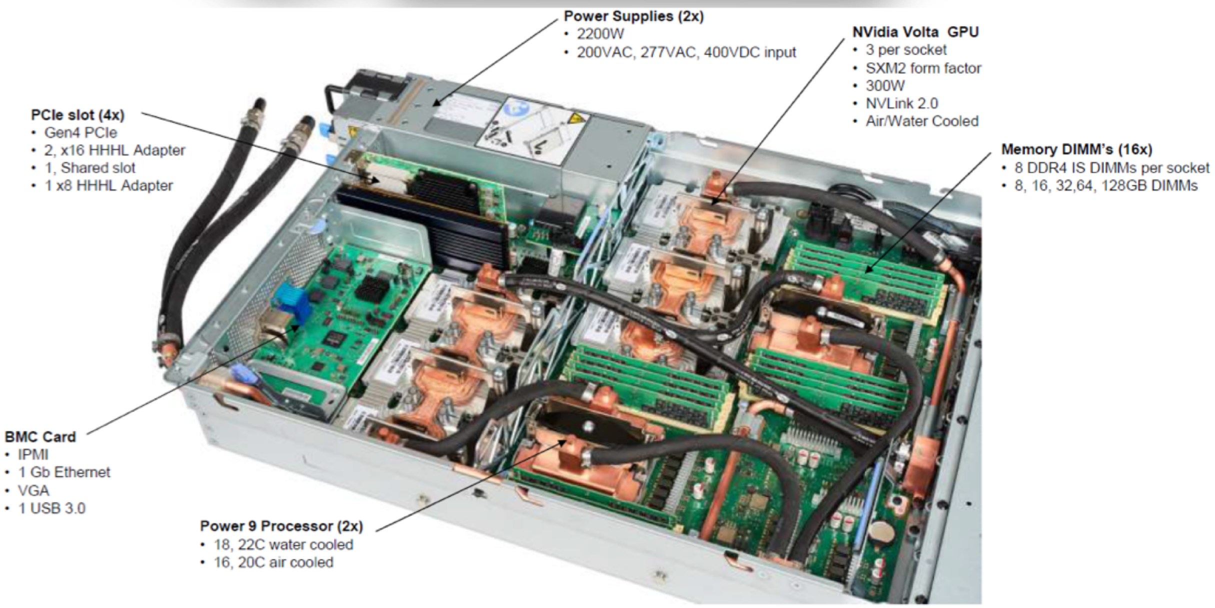
https://developer.nvidia.com/cuda-downloads

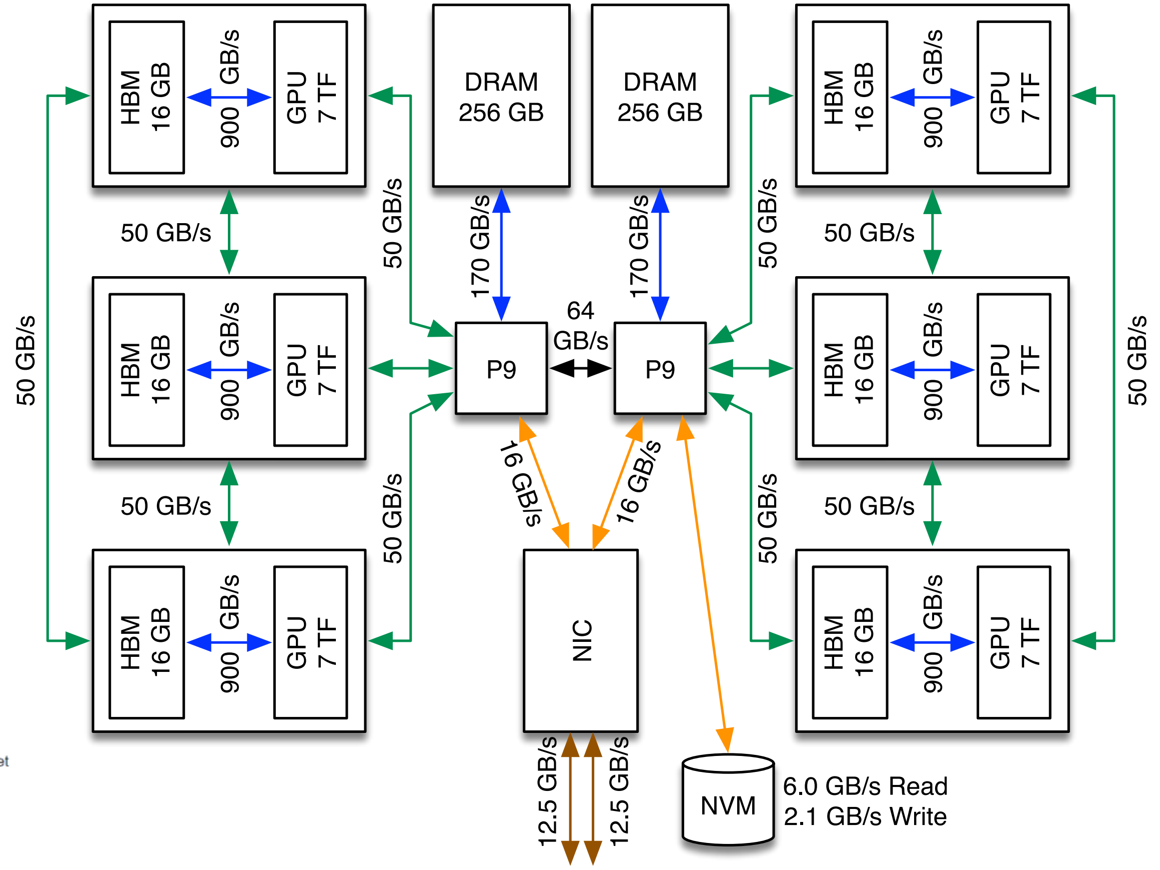
Make sure to download the appropriate version for your local operating system.

NOTE: You do not need an NVIDIA GPU on your local machine to install the toolkit and use the profiler.

summit





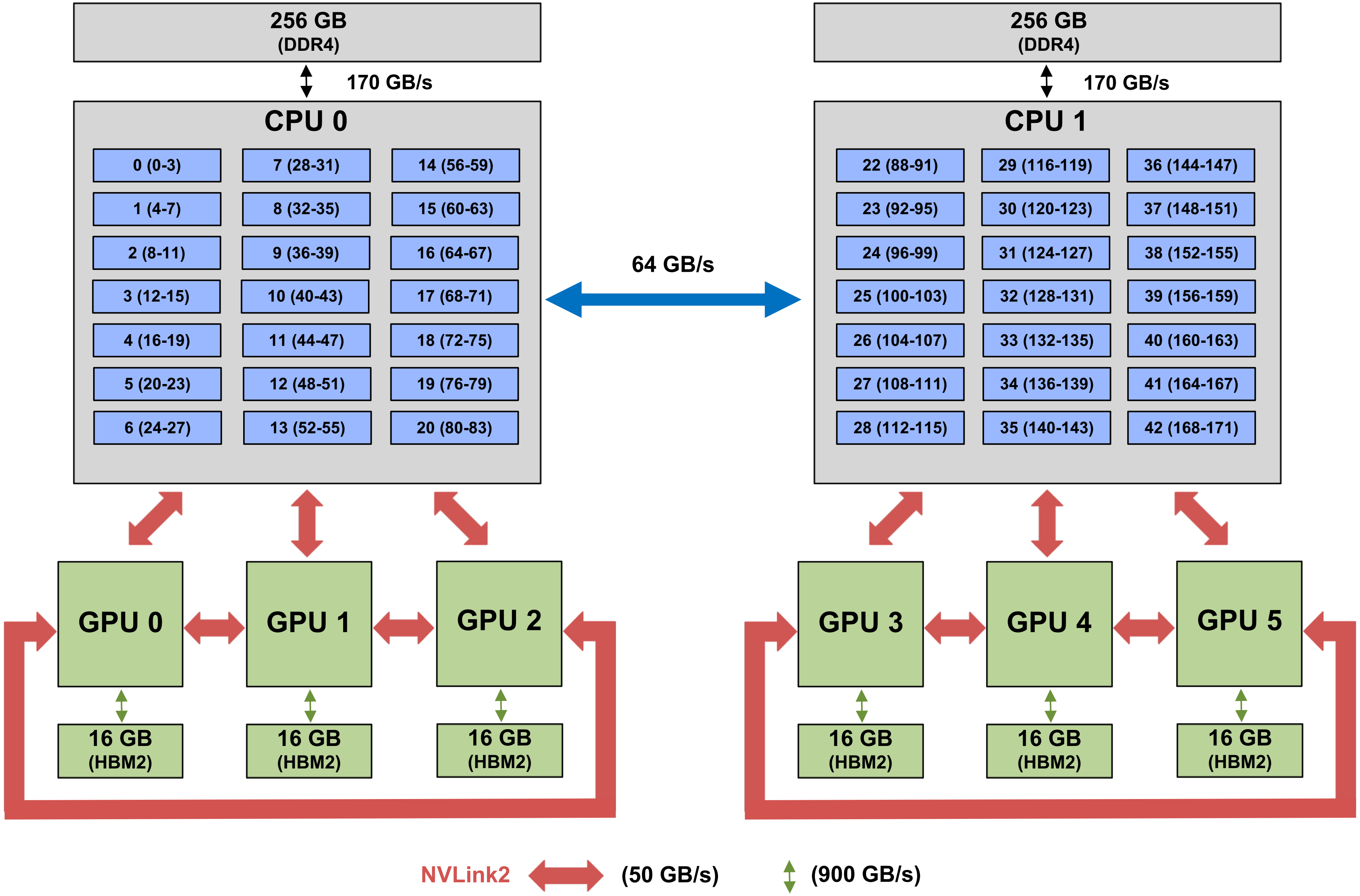


HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) are bi-directional.

Node Overview

Summit Node

(2) IBM Power9 + (6) NVIDIA Volta V100



Cloning Repository & Setting Up Environment

Log Into Ascent and Change Directory

1 From the command line:

```
$ ssh USERNAME@login1.ascent.olcf.ornl.gov
```

(This will drop you into the directory `/ccsopen/home/USERNAME`)

2 Change to the following directory:

```
$ cd /gpfs/wolf/gen121/scratch/USERNAME
```

On Summit, you should navigate to the corresponding Alpine/GPFS directory for your project (since you need read/write access from the compute nodes).

E.g. **`/gpfs/alpine/PROJID/scratch/USERNAME`**

Clone Repository and Set up Programming Environment

3

Once in the appropriate directory from step 2, clone the git repository:

```
$ git clone https://github.com/olcf/nvidia_profilers.git
```

4

cd into directory:

```
$ cd nvidia_profilers
```

5

Run script to set up environment for the tutorial:

```
$ source environment_ascent.sh
```

On Summit, source the environment_summit.sh file instead.

At this point, you prompt should look like this:

```
[USERNAME@login1: /gpfs/wolf/gen121/scratch/USERNAME/nvidia_profilers]$
```

A Simple Example: Vector Addition

CUDA Vector Addition

```
#include <stdio.h>
#define N 1048576

__global__ void add_vectors(int *a, int *b, int *c){
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id < N) c[id] = a[id] + b[id];
}

int main(){
    size_t bytes = N*sizeof(int);

    int *A = (int*)malloc(bytes);
    int *B = (int*)malloc(bytes);
    int *C = (int*)malloc(bytes);

    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    for(int i=0; i<N; i++){
        A[i] = 1;
        B[i] = 2;
    }

    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

    int thr_per_blk = 256;
    int blk_in_grid = ceil( float(N) / thr_per_blk );
    add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);

    free(A);
    free(B);
    free(C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

CUDA Vector Addition

```
#include <stdio.h>
#define N 1048576

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

Vector addition kernel (GPU)

```
int main(){
    size_t bytes = N*sizeof(int);

    int *A = (int*)malloc(bytes);
    int *B = (int*)malloc(bytes);
    int *C = (int*)malloc(bytes);
```

Allocate memory on CPU

```
    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);
```

Allocate memory on GPU

```
    for(int i=0; i<N; i++){
        A[i] = 1;
        B[i] = 2;
    }
```

Initialize arrays on CPU

```
    cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
```

Copy data from CPU to GPU

```
    int thr_per_blk = 256;
    int blk_in_grid = ceil( float(N) / thr_per_blk );
    add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);
```

Set configuration parameters and launch kernel

```
    cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

Copy data from GPU to CPU

```
    free(A);
    free(B);
    free(C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
```

Free memory on CPU and GPU

```
    return 0;
}
```

Vector Addition Example

```
$ cd vector_addition/cuda

$ make

$ bsub submit.lsf
```

Invoke the command line profiler

`-s`: Print summary of profiling results
(default unless -o is used)

`-o`: Export timeline file
(to be opened later in NVIDIA Visual Profiler)

%h: replace with hostname

${LSB_JOBID} holds the job ID assigned by LSF
(NOT specific to NVIDIA profilers)

From **`submit.lsf`**:

```
jsrun -n1 -c1 -g1 -a1 nvprof -s -o vec_add_cuda.${LSB_JOBID}.%h.nvvp ./run
```

Vector Addition Example (nvprof results – text only)

From **`vec_add_cuda.JOBID`**:

```
==174655== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.25%  463.36us         2  231.68us  229.66us  233.70us  [CUDA memcpy HtoD]
                   41.59%  342.56us         1  342.56us  342.56us  342.56us  [CUDA memcpy DtoH]
                    2.16%  17.824us         1  17.824us  17.824us  17.824us  add_vectors(int*, int*, int*)
      API calls:   99.35%  719.78ms         3  239.93ms  1.1351ms  717.50ms  cudaMalloc
                    0.23%  1.6399ms        96  17.082us     224ns  670.19us  cuDeviceGetAttribute
                    0.17%  1.2559ms         3  418.64us  399.77us  454.40us  cudaFree
                    0.16%  1.1646ms         3  388.18us  303.13us  550.07us  cudaMemcpy
                    0.06%  412.85us         1  412.85us  412.85us  412.85us  cuDeviceTotalMem
                    0.03%  182.11us         1  182.11us  182.11us  182.11us  cuDeviceGetName
                    0.00%  32.391us         1  32.391us  32.391us  32.391us  cudaLaunchKernel
                    0.00%  3.8960us         1  3.8960us  3.8960us  3.8960us  cuDeviceGetPCIBusId
                    0.00%  2.2920us         3     764ns     492ns  1.1040us  cuDeviceGetCount
                    0.00%  1.4090us         2     704ns     423ns     986ns  cuDeviceGet
```

Vector Addition Example – Visual Profiler

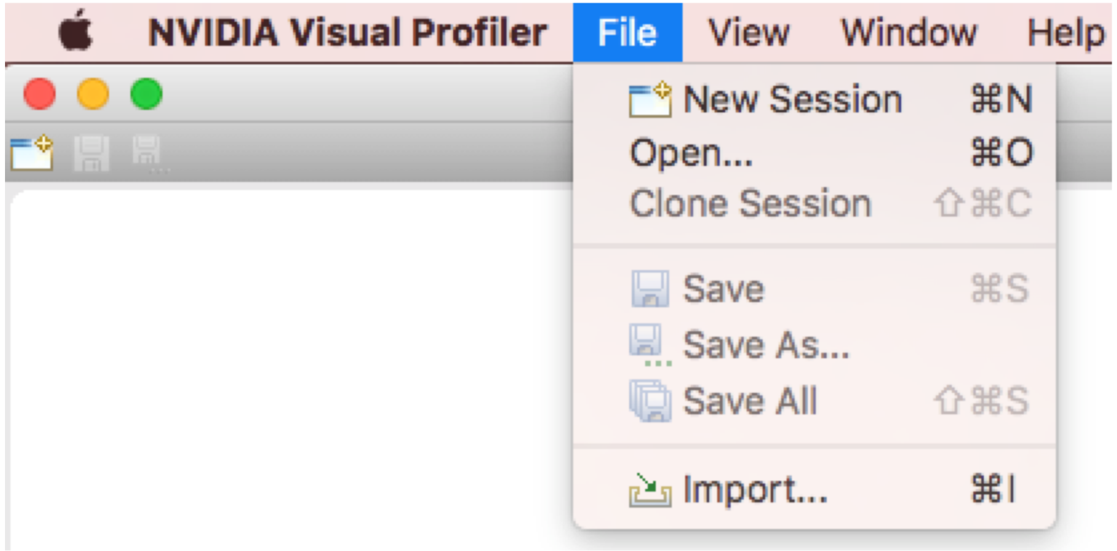
Now, transfer the .nvvp file from Ascent to your local machine to view in NVIDIA Visual Profiler.

From your local system:

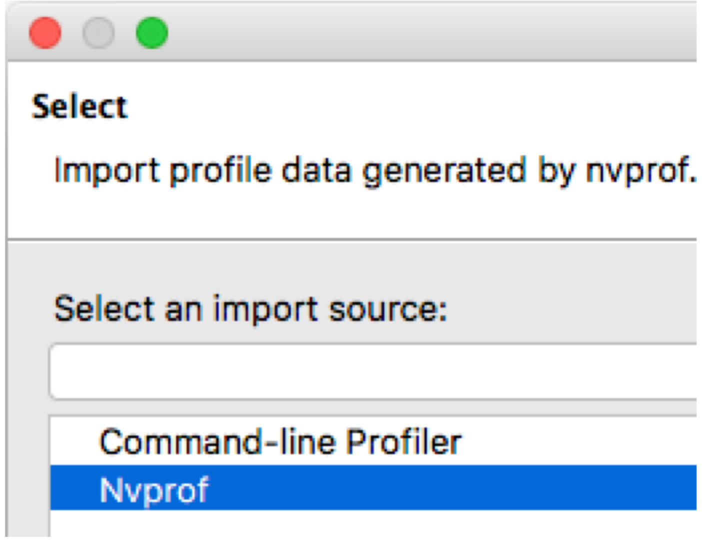
```
$ scp USERNAME@login1.ascent.ccs.ornl.gov:/path/to/file/remote /path/to/desired/location/local
```

Vector Addition Example – Visual Profiler

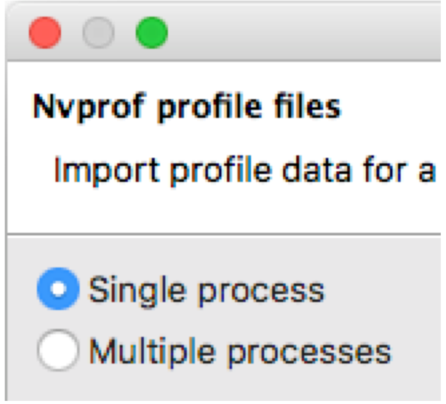
1 File->Import



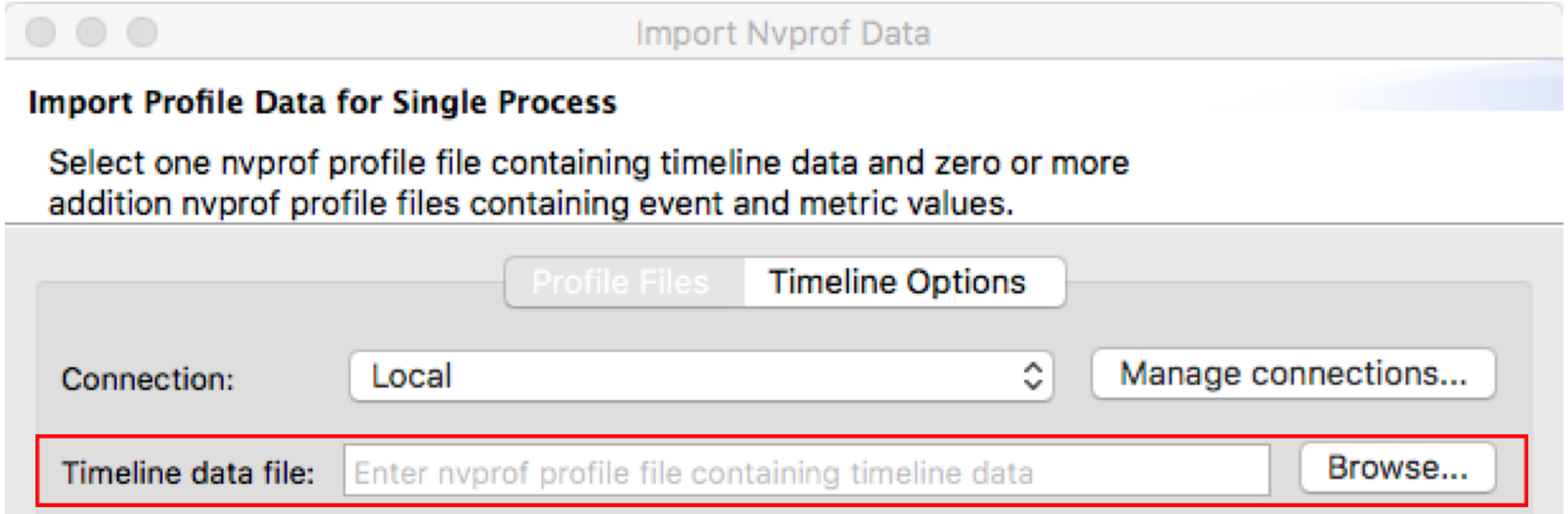
2 Select "Nvprof" then "Next >"



3 Select "Single Process" then "Next >"



4 Click "Browse" next to "Timeline data file" to locate the .nvvp file on your local system, then click "Finish"



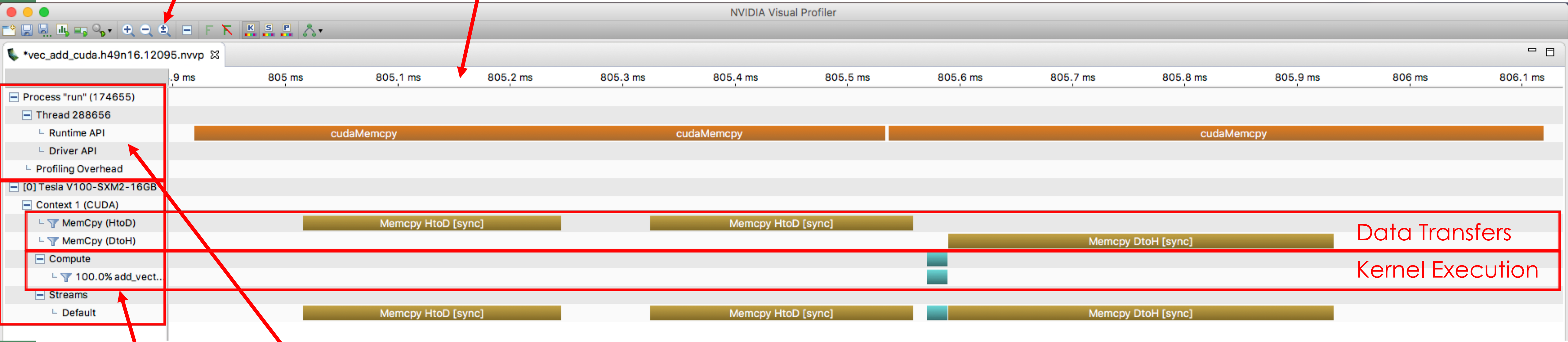
Oak Ridge National Laboratory

Vector Addition Example – Visual Profiler

To zoom in on a specific region, hold Ctrl + left-click and drag mouse (Cmd for Mac)

Zoom all the way out

Left-click the timeline and drag mouse to measure specific activities



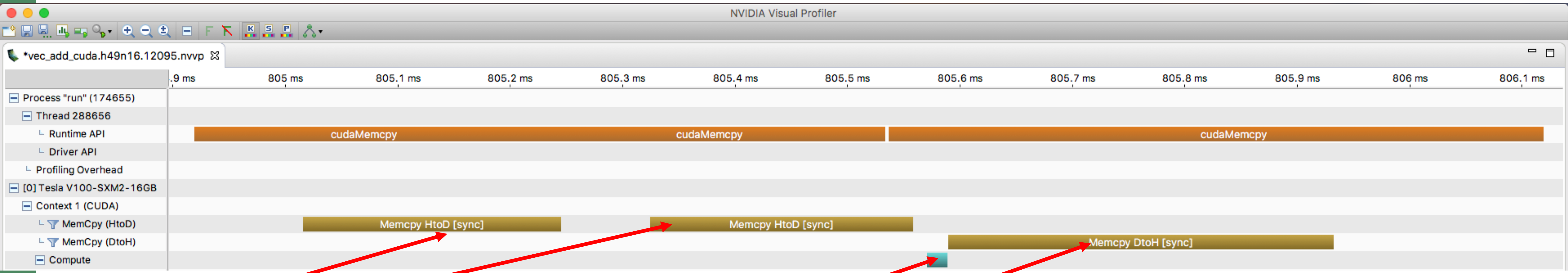
Data Transfers

Kernel Execution

CUDA API activity from CPU process

GPU activity

Vector Addition Example – Visual Profiler



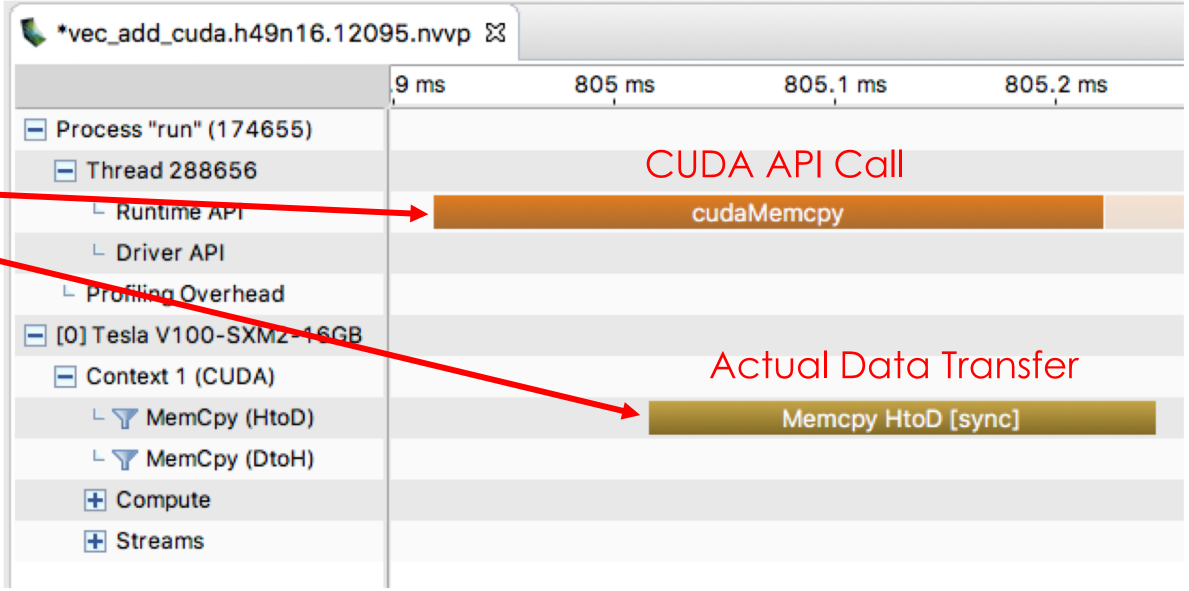

```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//      thr_per_blk: number of CUDA threads per grid block
//      blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	56.25%	463.36us	2	231.68us	229.66us	233.70us	[CUDA memcpy HtoD]
	41.59%	342.56us	1	342.56us	342.56us	342.56us	[CUDA memcpy DtoH]
	2.16%	17.824us	1	17.824us	17.824us	17.824us	add_vectors(int*, int*, int*)

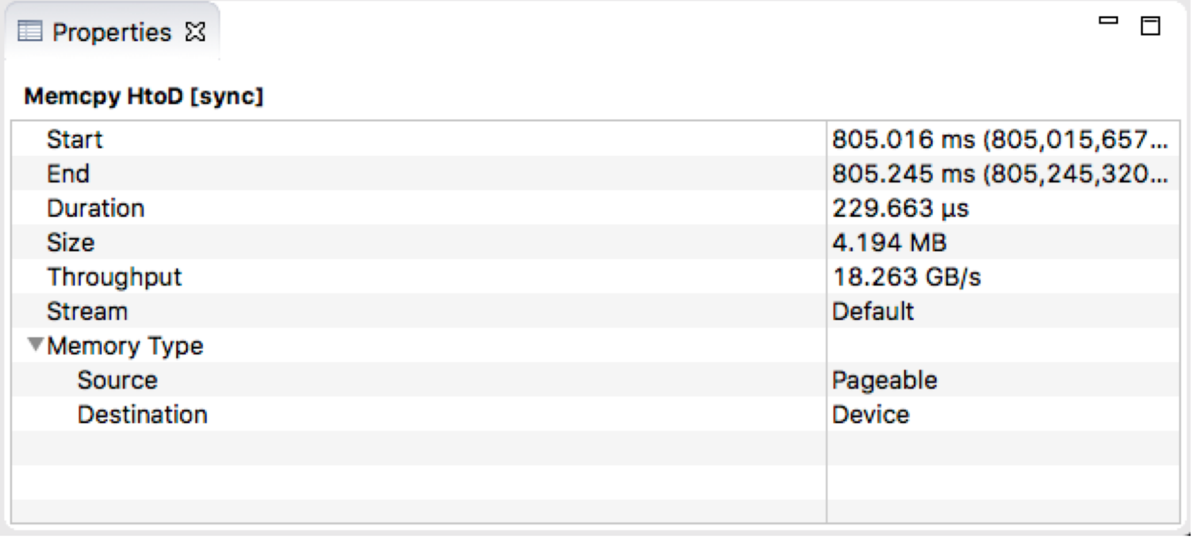
Oak Ridge National Laboratory

Vector Addition Example – Visual Profiler

```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//      thr_per_blk: number of CUDA threads per grid block
//      blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```

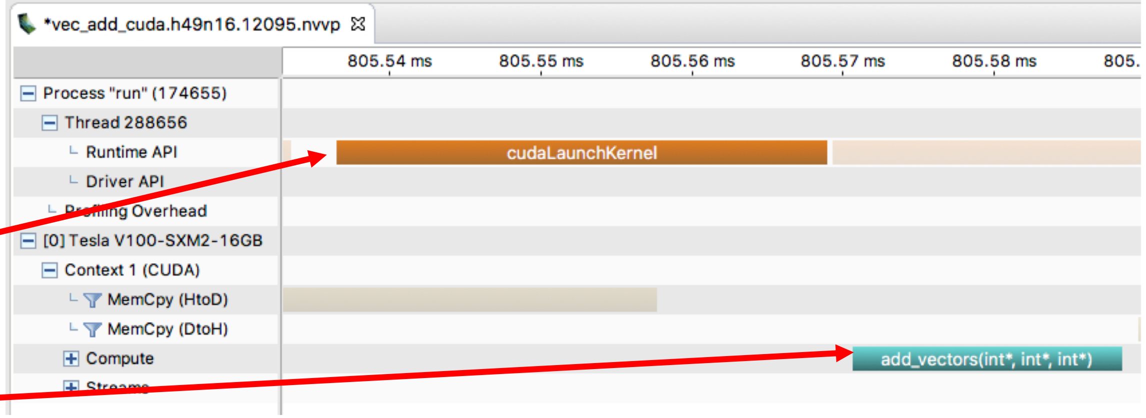


Details about the data transfer

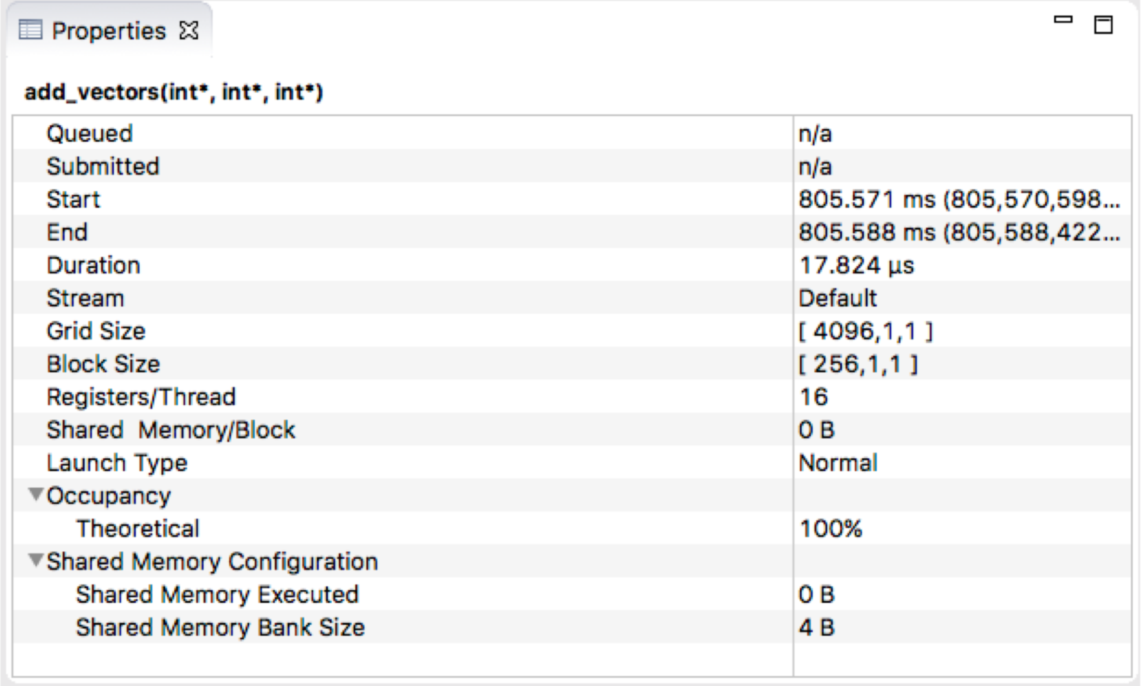


Vector Addition Example – Visual Profiler

```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// Set execution configuration parameters
//      thr_per_blk: number of CUDA threads per grid block
//      blk_in_grid: number of blocks in grid
int thr_per_blk = 256;
int blk_in_grid = ceil( float(N) / thr_per_blk );

// Launch kernel
add_vectors<<< blk_in_grid, thr_per_blk >>>(d_A, d_B, d_C);

// Copy data from device array d_C to host array C
cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
```



Details about the kernel execution


Properties

add_vectors(int*, int*, int*)

Queued	n/a
Submitted	n/a
Start	805.571 ms (805,570,598...
End	805.588 ms (805,588,422...
Duration	17.824 μs
Stream	Default
Grid Size	[4096,1,1]
Block Size	[256,1,1]
Registers/Thread	16
Shared Memory/Block	0 B
Launch Type	Normal
▼Occupancy	
Theoretical	100%
▼Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Oak Ridge National Laboratory

Jacobi Iteration

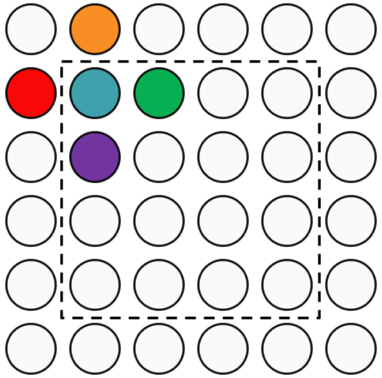
Jacobi Iteration – Problem Description

Use Jacobi Iteration to solve 2D Poisson equation with periodic boundary conditions:

$$\Delta A(y,x) = e^{-10(x*x + y*y)}$$

Execute a Jacobi Step on the Inner Points

$$A_{k+1}(iy,ix) = -0.25 * (rhs(iy,ix) - (A_k(iy,ix-1) + A_k(iy,ix+i) + A_k(iy-1,ix) + A_k(iy+1,ix)))$$



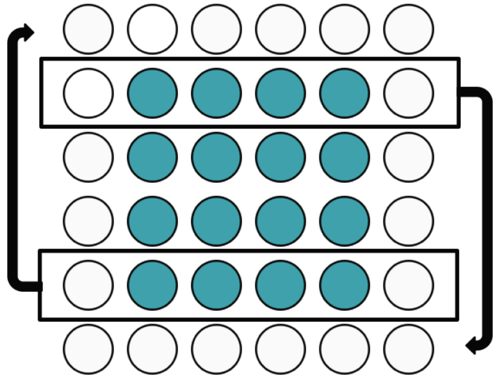
```
for (int iy = 1; iy < NY-1; iy++)
{
    for( int ix = 1; ix < NX-1; ix++ )
    {
        Anew[iy][ix] = -0.25 * (rhs[iy][ix] - ( A[iy][ix+1] + A[iy][ix-1]
                                             + A[iy-1][ix] + A[iy+1][ix] ));
        error = fmax( error, fabs(Anew[iy][ix]-A[iy][ix]));
    }
}
```

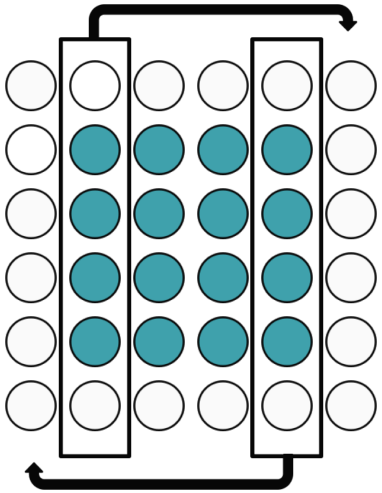
Copy Values of Anew to A

```
for (int iy = 1; iy < NY-1; iy++)
{
    for( int ix = 1; ix < NX-1; ix++ )
    {
        A[iy][ix] = Anew[iy][ix];
    }
}
```

Apply Periodic Boundary Conditions

```
//Periodic boundary conditions
for( int ix = 1; ix < NX-1; ix++ )
{
        A[0][ix]       = A[(NY-2)][ix];
        A[(NY-1)][ix] = A[1][ix];
}
for (int iy = 1; iy < NY-1; iy++)
{
        A[iy][0]       = A[iy][(NX-2)];
        A[iy][(NX-1)] = A[iy][1];
}
```





Serial Version

`jacobi/1_serial`

Serial Runtime

Compile the code

```
$ make
pgcc -Minfo -fast -c poisson2d.c
main:
     54, Generated vector simd code for the loop
         FMA (fused multiply-add) instruction(s) generated
     65, Memory zero idiom, loop replaced by call to __c_mzero8
     84, FMA (fused multiply-add) instruction(s) generated
     90, Generated vector simd code for the loop containing reductions
    100, Memory copy idiom, loop replaced by call to __c_mcopy8
    107, Loop not fused: dependence chain to sibling loop
         Generated vector simd code for the loop
         Residual loop unrolled 2 times (completely unrolled)
    112, Loop not fused: function call before adjacent loop
         Loop unrolled 8 times
pgcc -Minfo -fast poisson2d.o -o run
```

Run the code (on single CPU core)

```
$ bsub submit.lsf
Job <11536> is submitted to default queue <batch>.


$ jobstat
--------------- Running Jobs: 1 (1 of 16 nodes, 6.25%) ---------------
JobId  Username  Project  Nodes  Remain  StartTime       JobName
11536  t4p       GEN117   1      8:48    02/24 10:03:14  serial
--------------- Eligible Jobs: 0 -------------------------------------
--------------- Blocked Jobs: 0 --------------------------------------


$ less serial.11536
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s):  94.9856

...
...
```

(Enter q to quit/exit less)

jacobi/1_serial

Single GPU Version

`jacobi/2_single_gpu`

Difference From Serial Version

- Added OpenACC pragmas to inform compiler where to offload work to GPU

```
#pragma acc kernels
```

- Added (optional) serial version to compare with timing and results of GPU version

```
// Set to 1 to run serial test, otherwise 0
int serial_test = 0;
```


jacobi/2_single_gpu

Runtime of Single GPU Version

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast -c poisson2d.c
main:
    117, Generating implicit copyin(A[:][:],rhs[1:4094][1:4094])
         Generating implicit copyout(Anew[1:4094][1:4094])
    118, Loop is parallelizable
    120, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        118, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        120, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
        124, Generating implicit reduction(max:error)
    128, Generating implicit copyin(Anew[1:4094][1:4094])
         Generating implicit copyout(A[1:4094][1:4094])
    129, Loop is parallelizable
    131, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        129, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        131, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    138, Generating implicit copy(A[:][1:4094])
    139, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        139, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    144, Generating implicit copy(A[1:4094][:])
    145, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        145, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast poisson2d.o -o run
```

Run the code (on single GPU)

```
$ bsub submit.lsf

$ less single_gpu.JOBID
Jacobi relaxation Calculation: 4096 x 4096 mesh
Parallel Execution...
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s) - Parallel:
```


jacobi/2_single_gpu

Runtime of Single GPU Version

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast -c poisson2d.c
main:
    117, Generating implicit copyin(A[:][:],rhs[1:4094][1:4094])
         Generating implicit copyout(Anew[1:4094][1:4094])
    118, Loop is parallelizable
    120, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        118, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        120, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
        124, Generating implicit reduction(max:error)
    128, Generating implicit copyin(Anew[1:4094][1:4094])
         Generating implicit copyout(A[1:4094][1:4094])
    129, Loop is parallelizable
    131, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        129, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        131, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    138, Generating implicit copy(A[:][1:4094])
    139, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        139, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    144, Generating implicit copy(A[1:4094][:])
    145, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        145, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast poisson2d.o -o run
```

Run the code (on single GPU)

```
$ bsub submit.lsf

$ less single_gpu.JOBID
Jacobi relaxation Calculation: 4096 x 4096 mesh
Parallel Execution...
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s) - Parallel: 127.2326
```

Why are we slower than serial version??

How can we answer such questions?

jacobi/2_single_gpu

Using NVIDIA's NVProf Profiler, we see…

```
$ bsub submit.lsf   (jsrun --smpiargs="none" -n1 -c1 -g1 -a1 nvprof -s -o single_gpu.%h.${LSB_JOBID}.nvvp ./run)

$ less single_gpu.JOBID
==56446== NVPROF is profiling process 56446, command: ./run
==56446== Profiling application: ./run

Jacobi relaxation Calculation: 4096 x 4096 mesh
Parallel Execution...
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s) - Parallel: 130.9012

==56446== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   53.55%  14.4180s     41000  351.66us  1.3110us  382.72us  [CUDA memcpy HtoD]
                   42.84%  11.5335s     33000  349.50us  1.7590us  362.53us  [CUDA memcpy DtoH]
                    2.01%  541.55ms      1000  541.55us  539.61us  546.01us  main_120_gpu
                    1.38%  372.18ms      1000  372.18us  369.47us  376.64us  main_131_gpu
                    0.19%  49.816ms      1000  49.815us  48.448us  51.231us  main_124_gpu__red
                    0.02%  6.1174ms      1000  6.1170us  5.7270us  6.9760us  main_145_gpu
                    0.01%  2.1649ms      1000  2.1640us  1.8880us  2.8480us  main_139_gpu
```

Do we really need all these data transfers?

Let's look at visual output (and compiler output) to see what's going on…

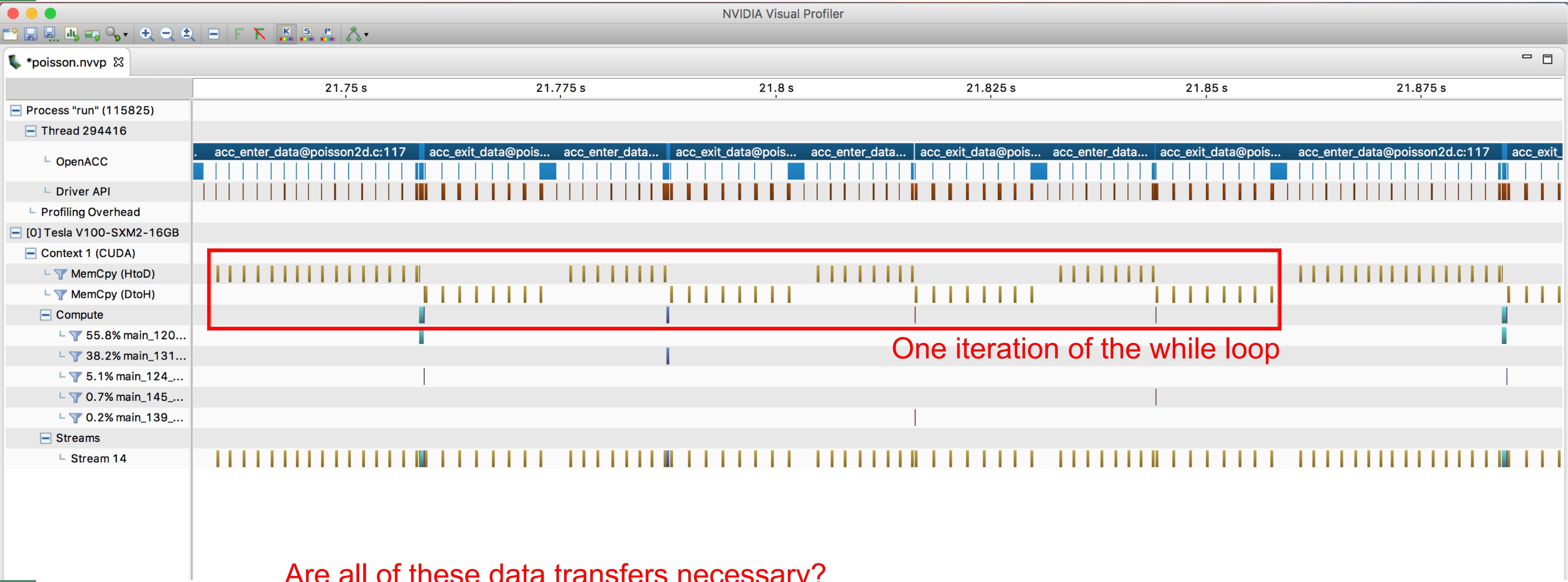
Transfer .nvvp file from Ascent/Summit to local system

From your local system:

```
$ scp USERNAME@login1.ascent.ccs.ornl.gov:/path/to/file/remote /path/to/desired/location/local
```


jacobi/2_single_gpu

Using NVIDIA's Visual Profiler, we see…



Are all of these data transfers necessary?

Let's look back at the code to see how data should be transferred…

Oak Ridge National Laboratory

jacobi/2_single_gpu

Where are arrays actually needed?

```
// Main iteration loop
while ( error > tol && iter < iter_max )
{
    error = 0.0;

    #pragma acc kernels
    for (int iy = 1; iy < NY-1; iy++)
    {
        for( int ix = 1; ix < NX-1; ix++ )
        {
            Anew[iy][ix] = -0.25 * (rhs[iy][ix] - ( A[iy][ix+1] + A[iy][ix-1]
                                                   + A[iy-1][ix] + A[iy+1][ix] ));
            error = fmax( error, fabs(Anew[iy][ix]-A[iy][ix]));
        }
    }

    #pragma acc kernels
    for (int iy = 1; iy < NY-1; iy++)
    {
        for( int ix = 1; ix < NX-1; ix++ )
        {
            A[iy][ix] = Anew[iy][ix];
        }
    }

    //Periodic boundary conditions
    #pragma acc kernels
    for( int ix = 1; ix < NX-1; ix++ )
    {
            A[0][ix]      = A[(NY-2)][ix];
            A[(NY-1)][ix] = A[1][ix];
    }
    #pragma acc kernels
    for (int iy = 1; iy < NY-1; iy++)
    {
            A[iy][0]      = A[iy][(NX-2)];
            A[iy][(NX-1)] = A[iy][1];
    }

    if((iter % 100) == 0) printf("%5d, %0.6f\n", iter, error);

    iter++;
}
```

- `Anew` is updated
- `A` and `rhs` are not updated
- Reduction performed on `error`

- `A` is updated
- `Anew` is not updated

But nowhere in this while loop are `Anew`, `A`, or `rhs` needed on the CPU!

- `A` is updated

- `A` is updated

jacobi/2_single_gpu

Single GPU Version with Data Regions

`jacobi/3_single_gpu_data`

Difference From Initial GPU Version

- Added a data region around while loop

```
#pragma acc data ...
{
  while loop
}
```

- Still have (optional) serial version to compare with timing and results of GPU version

```
// Set to 1 to run serial test, otherwise 0
int serial_test = 0;
```


jacobi/3_single_gpu_data

Runtime of Single GPU Version with Data Directives

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast -c poisson2d.c
main:
    112, Generating copyin(rhs[:][:])
         Generating create(Anew[:][:])
         Generating copy(A[:][:])
    121, Loop is parallelizable
    123, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        121, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        123, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
        127, Generating implicit reduction(max:error)
    132, Loop is parallelizable
    134, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        132, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        134, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    142, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        142, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    148, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        148, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -fast poisson2d.o -o run
```

Run the code (on single GPU)

```
$ bsub submit.lsf

$ less single_gpu_data.JOBID
Jacobi relaxation Calculation: 4096 x 4096 mesh
Parallel Execution…
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s) - Parallel:
```

jacobi/3_single_gpu_data

Using NVIDIA's NVProf Profiler, we see…

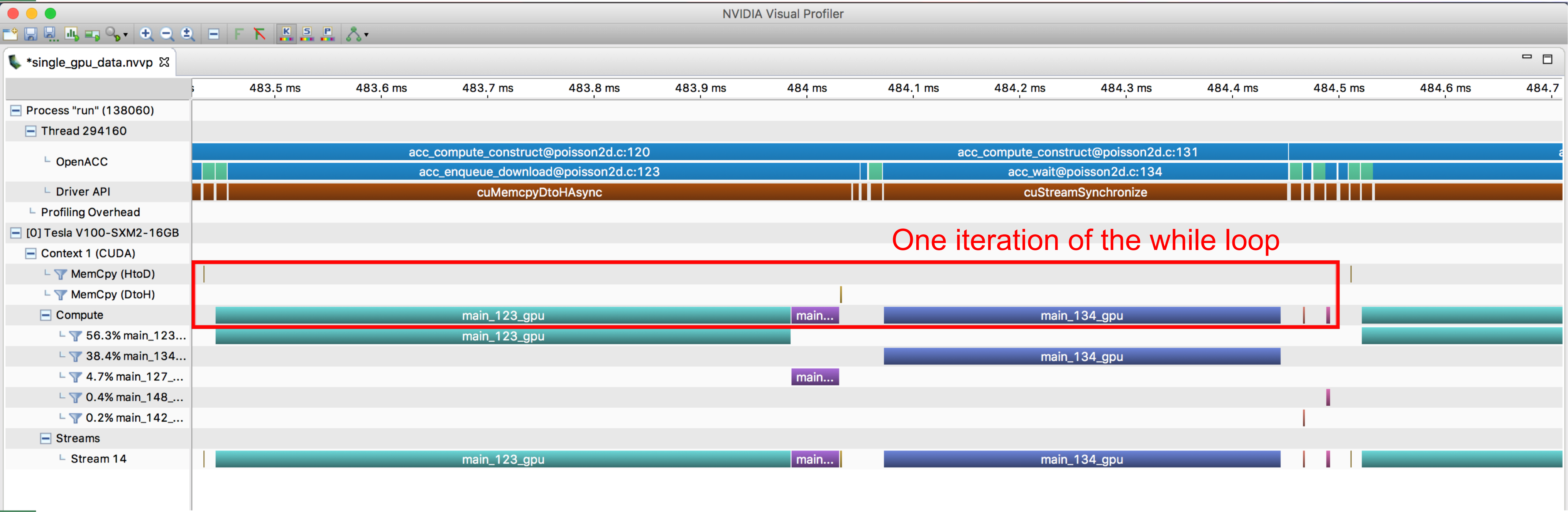
```
$ bsub submit.lsf   (jsrun --smpiargs="none" -n1 -c1 -g1 -a1 nvprof -s -o single_gpu_data.%h.${LSB_JOBID}.nvvp ./run)

$ less single_gpu_data.JOBID
==139388== NVPROF is profiling process 139388, command: ./run
==139388== Profiling application: ./run

Jacobi relaxation Calculation: 4096 x 4096 mesh
Parallel Execution...
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s) - Parallel:    1.9883

==139388== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   55.51%  539.46ms      1000  539.46us  537.53us  542.75us  main_123_gpu
                   38.02%  369.46ms      1000  369.46us  366.65us  373.57us  main_134_gpu
                    4.76%  46.220ms      1000  46.219us  43.935us  51.040us  main_127_gpu__red
                    0.72%  7.0198ms      1016  6.9090us  1.2160us  360.06us  [CUDA memcpy HtoD]
                    0.47%  4.5286ms      1009  4.4880us  1.5990us  359.04us  [CUDA memcpy DtoH]
                    0.35%  3.4053ms      1000  3.4050us  3.0400us  4.4480us  main_148_gpu
                    0.18%  1.7651ms      1000  1.7650us  1.6320us  2.2080us  main_142_gpu
```

Much faster with explicit data management!

Data transfers are no longer dominating run time.

jacobi/3_single_gpu_data

Using NVIDIA's Visual Profiler, we see…



We have eliminated the unnecessary data transfers.

OAK RIDGE National Laboratory

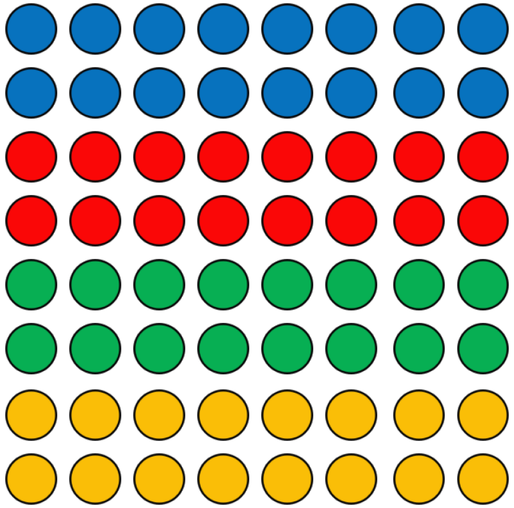
jacobi/3_single_gpu_data

Multiple GPU Version (OpenMP + OpenACC)

`jacobi/4_multiple_gpu_openmp`

Differences from Single GPU Version

- Each OpenMP thread calculates its own loop bounds for its portion of the domain and uses its own GPU.



OpenMP Thread 0 ⇒ GPU 0

OpenMP Thread 1 ⇒ GPU 1

OpenMP Thread 2 ⇒ GPU 2

OpenMP Thread 3 ⇒ GPU 3

`jacobi/4_multiple_gpu_openmp`

Differences from Single GPU Version

```
#pragma omp parallel default(shared) firstprivate(num_threads, thread_num){}
```

```
#ifdef _OPENMP
    num_threads = omp_get_num_threads();
    thread_num  = omp_get_thread_num();
#endif /* _OPENMP */


#ifdef _OPENACC
    int num_devices = acc_get_num_devices(acc_device_nvidia);
    int device_num  = thread_num % num_devices;
    acc_set_device_num(device_num, acc_device_nvidia);
#endif /* _OPENACC */
```

Map OpenMP threads to available GPUs

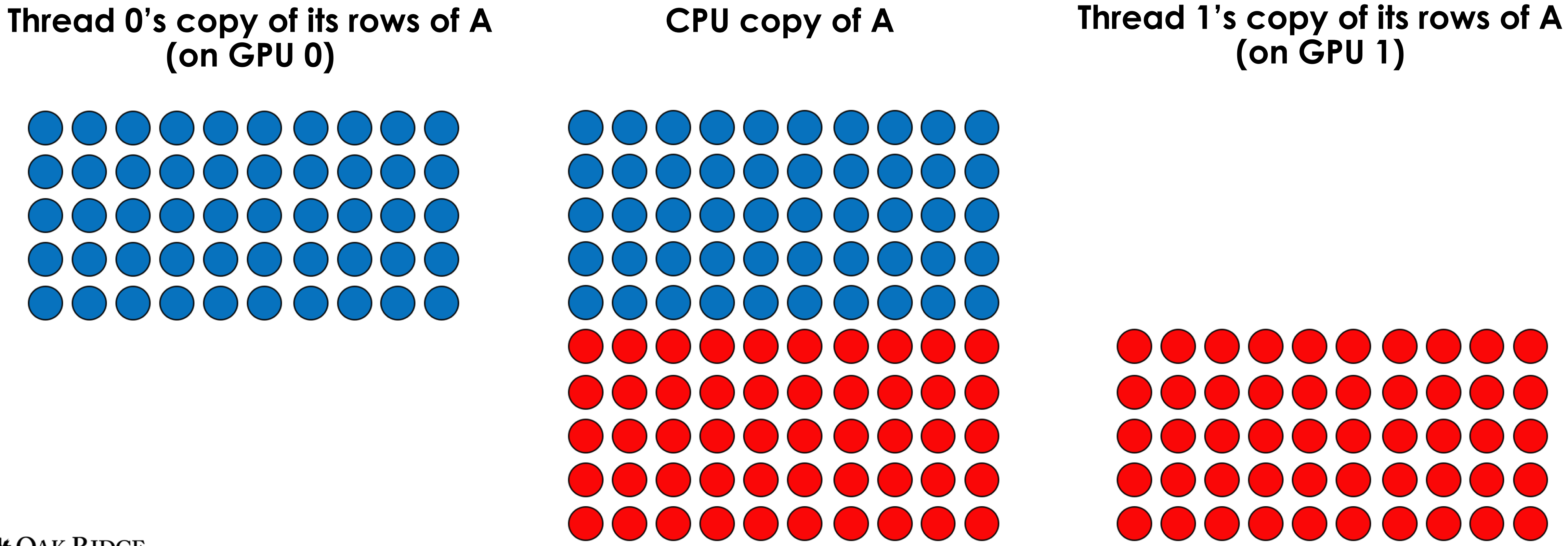
```
#pragma omp master
{
// Set rhs
for (int iy = 1; iy < NY-1; iy++)
{
    for(int ix = 1; ix < NX-1; ix++ )
    {
        const double x = -1.0 + (2.0*ix/(NX-1));
        const double y = -1.0 + (2.0*iy/(NY-1));
        rhs[iy][ix] = exp(-10.0*(x*x + y*y));
    }
}
} /* pragma omp master */
```

Only the master thread needs to set value of rhs

jacobi/4_multiple_gpu_openmp

Differences from Single GPU Version

```
#pragma acc data copy(A[(iy_start-1):(iy_end-iy_start)+2][0:NX]) \
                 copyin(rhs[iy_start:(iy_end-iy_start)][0:NX]) \
                 create(Anew[iy_start:(iy_end-iy_start)][0:NX])
```



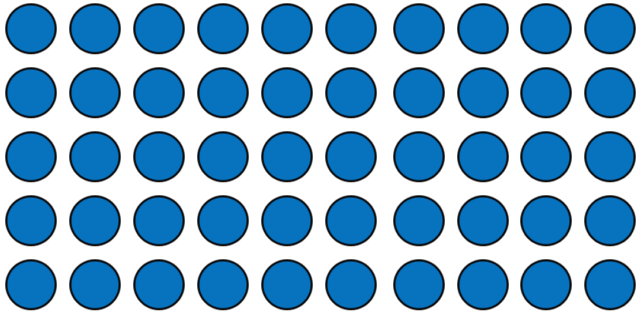
`jacobi/4_multiple_gpu_openmp`

Differences from Single GPU Version

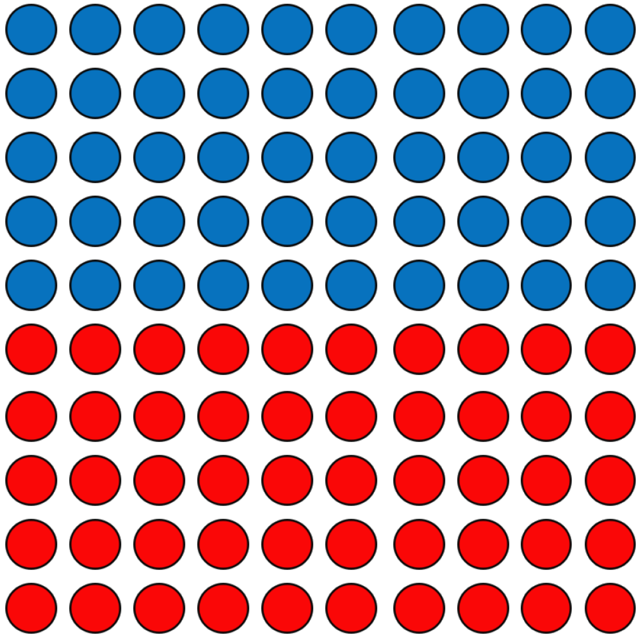
```
#pragma acc kernels
for (int iy = iy_start; iy < iy_end; iy++)
{
    for( int ix = ix_start; ix < ix_end; ix++ )
    {
        Anew[iy][ix] = -0.25 * (rhs[iy][ix] - ( A[iy][ix+1] + A[iy][ix-1]
                                               + A[iy-1][ix] + A[iy+1][ix] ));
        error = fmax( error, fabs(Anew[iy][ix]-A[iy][ix]));
    }
}
```

After GPUs update their values of A, the CPU copy is no longer correct

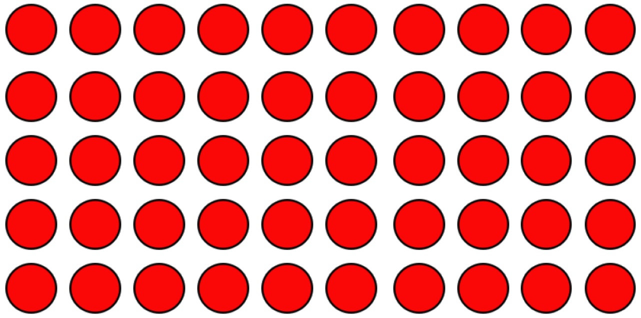
Thread 0's copy of its rows of A (on GPU 0)



CPU copy of A



Thread 1's copy of its rows of A (on GPU 1)



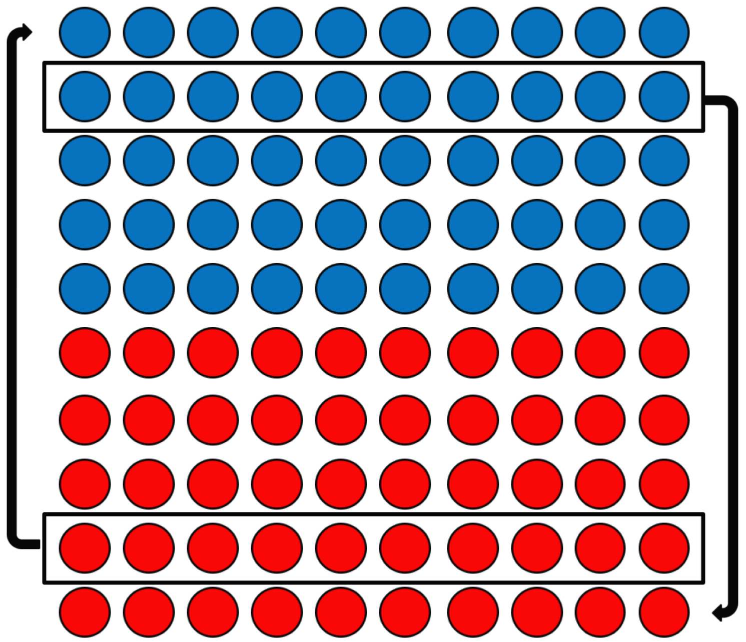
jacobi/4_multiple_gpu_openmp

Differences from Single GPU Version

Recall that boundary conditions must be updated for A matrix as a whole

- But each GPU only has its rows of A
- So some data must be passed back to CPU

CPU copy of A

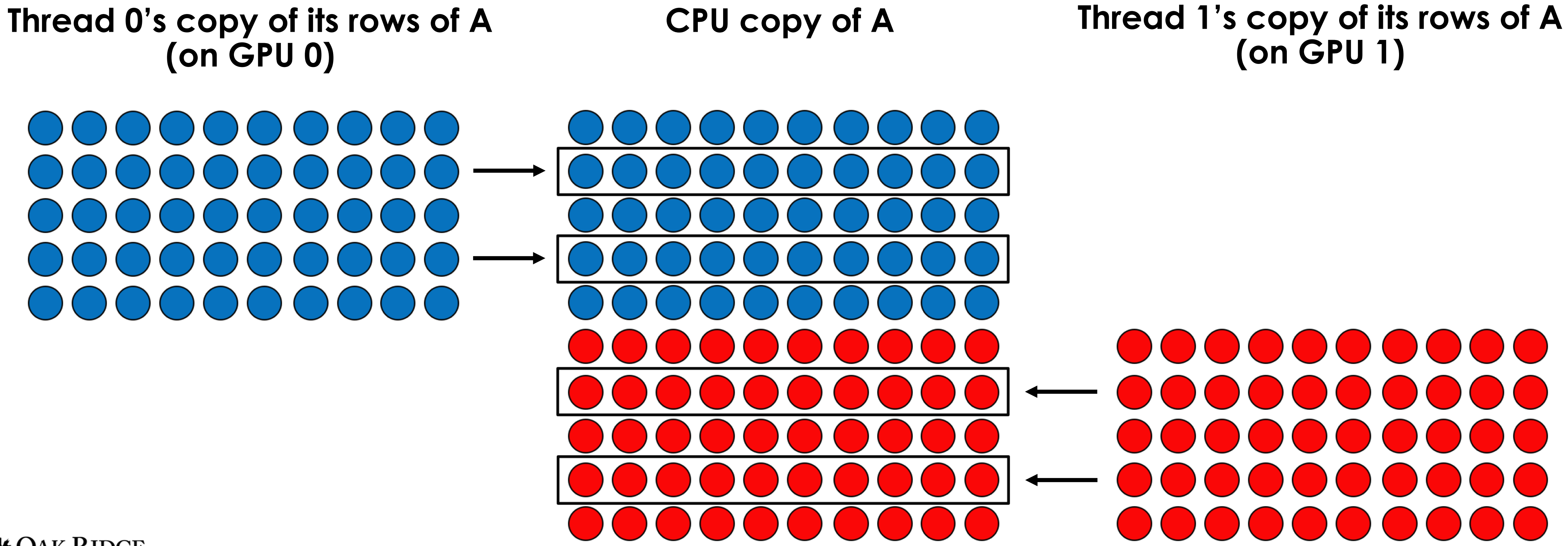


jacobi/4_multiple_gpu_openmp

Differences from Single GPU Version

```
#pragma acc update self(A[iy_start:1][0:NX], A[(iy_end-1):1][0:NX])
```

Each thread updates the "shared" CPU copy of A with its "2nd-to-top" row and "2nd-to-bottom" row



jacobi/4_multiple_gpu_openmp

Differences from Single GPU Version

Top/Bottom Boundaries

```
if(0 == (iy_start-1))
{
    for( int ix = 1; ix < NX-1; ix++ )
    {
        A[0][ix]      = A[(NY-2)][ix];
    }
}
```

```
if((NY-1) == (iy_end))
{
    for( int ix = 1; ix < NX-1; ix++ )
    {
        A[(NY-1)][ix] = A[1][ix];
    }
}
```

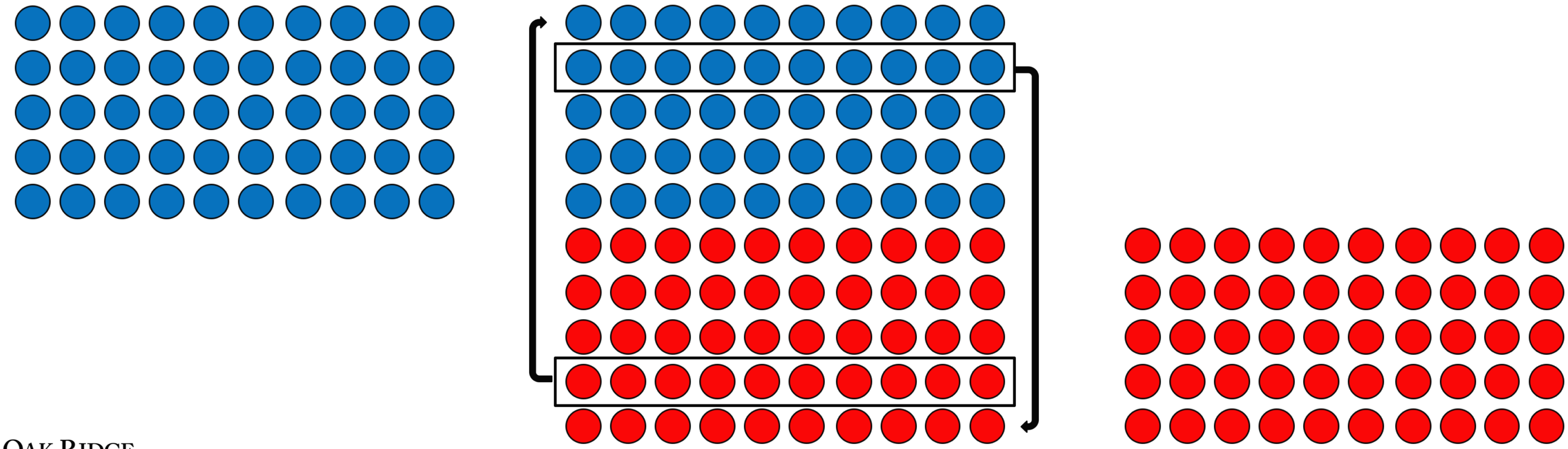
Side Boundaries

Only the threads with (0 == (iy_start-1)) and ((NY-1) == (iy_end)) perform the boundary updates

Thread 0's copy of its rows of A (on GPU 0)

CPU copy of A

Thread 1's copy of its rows of A (on GPU 1)



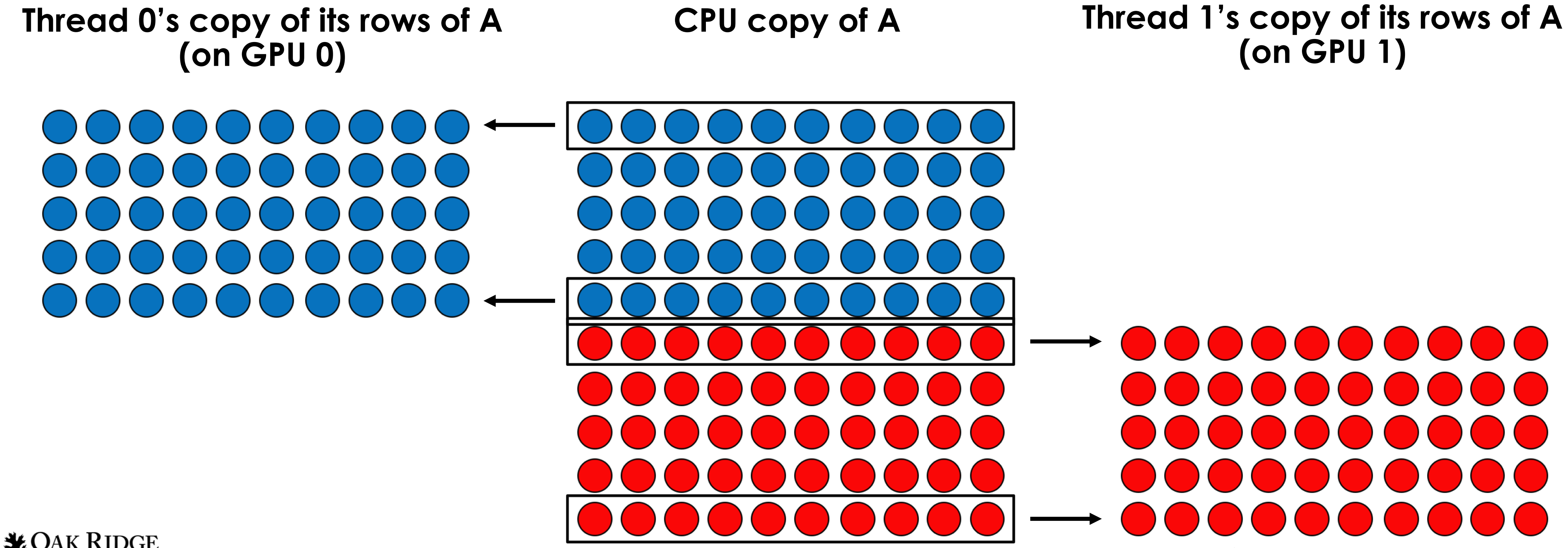
OAK RIDGE National Laboratory

jacobi/4_multiple_gpu_openmp

Differences from Single GPU Version

```
#pragma acc update device(A[(iy_start-1):1][0:NX], A[iy_end:1][0:NX])
```

Each thread updates its "top" row and "bottom" row from the new values of the CPU copy of A



jacobi/4_multiple_gpu_openmp

Runtime of Multi-GPU Version (with Data Directives)

Compile the code

```
$ make
pgcc -acc -Minfo=acc -ta=tesla:cc70 -mp -fast -c poisson2d.c
poisson2d_serial: ...
main:
    103, Generating implicit copyout(A[:][:],A_ref[:][:])
    104, Loop is parallelizable
    106, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        104, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        106, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    167, Generating copyin(rhs[iy_start:iy_end-iy_start][:])
         Generating create(Anew[iy_start:iy_end-iy_start][:])
         Generating copy(A[iy_start-1:iy_end-iy_start+2][:])
    181, Loop is parallelizable
    183, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        181, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        183, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
        187, Generating implicit reduction(max:error)
    200, Loop is parallelizable
    202, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        200, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        202, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    211, Generating update self(A[iy_start][:],A[iy_end-1][:])
    230, Generating update device(A[iy_start-1][:],A[iy_end][:])
    231, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        231, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -acc -Minfo=acc -ta=tesla:cc70 -mp -fast poisson2d.o -o run
```

Run the code (on 2 GPUs)

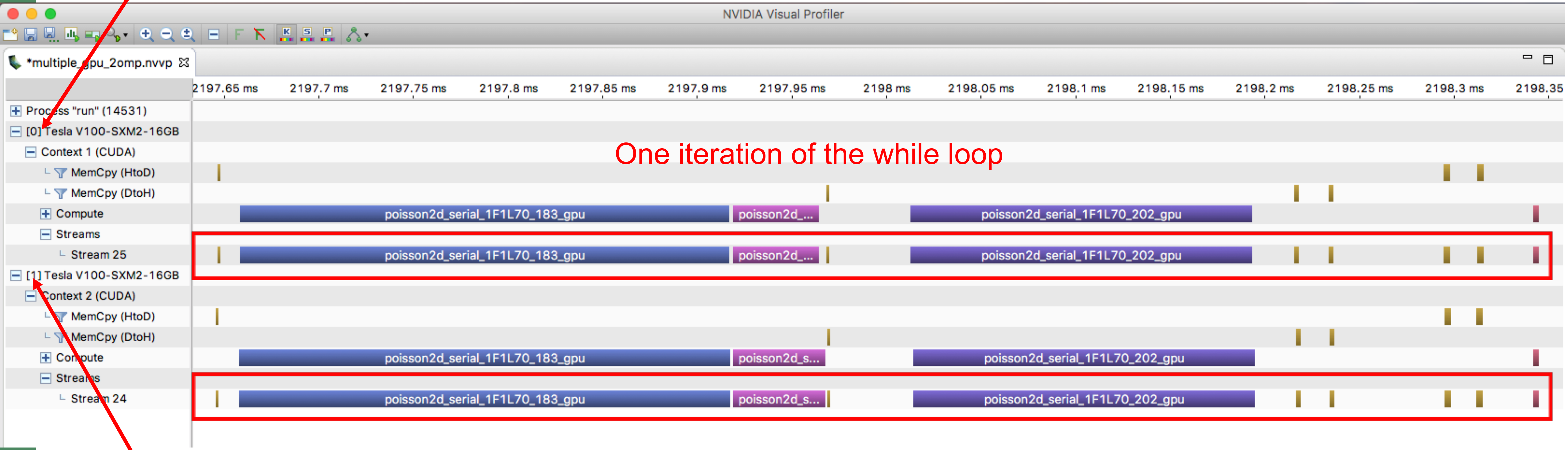
```
$ bsub submit2.lsf

$ less multi_gpu_2omp
Single-GPU Execution...
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Parallel Execution...
    0, 0.250000
  100, 0.249940
  200, 0.249880
  300, 0.249821
  400, 0.249761
  500, 0.249702
  600, 0.249642
  700, 0.249583
  800, 0.249524
  900, 0.249464
Elapsed Time (s) - Serial:     1.0990,
                   Parallel:   0.6692,
                   Speedup:    1.6424
```


OAK RIDGE National Laboratory

jacobi/4_multiple_gpu_openmp

Using NVIDIA's Visual Profiler, we see…

OpenMP Thread 0 (GPU 0)



OpenMP Thread 1 (GPU 1)

`jacobi/4_multiple_gpu_openmp`

Multiple MPI Ranks

Redundant Matrix Multiply

Each MPI rank is mapped to a GPU and performs the same steps (hence, redundant):

- Fill 2 NxN matrices with random numbers
- Perform a matrix multiply on CPU
- Perform a matrix multiply on GPU (loop_count times)
- Check for consistency between CPU and GPU results

Each MPI rank prints

- Its rank ID
- The hardware thread, GPU, and compute node it ran on
- Its total runtime and time spent computing on GPU

Multiple MPI Ranks

`redundant_MM`

Multiple MPI Ranks

Compile the code

```
$ make
```

Run the code

```
$ bsub submit.lsf
```

%q{OMPI_COMM_WORLD_RANK} (Replace with MPI Rank)

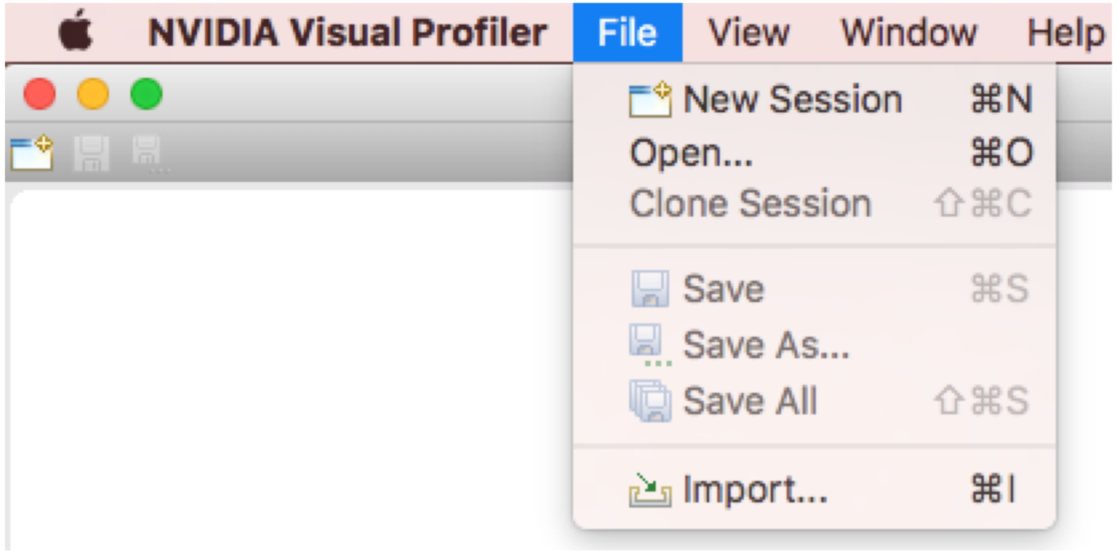
From submit.lsf

```
jsrun -n1 -c42 -g6 -a2 -bpacked:7 nvprof -o mat_mul.${LSB_JOBID}.%h.%q{OMPI_COMM_WORLD_RANK}.nvvp ./redundant_mm 2048 100 | sort
```

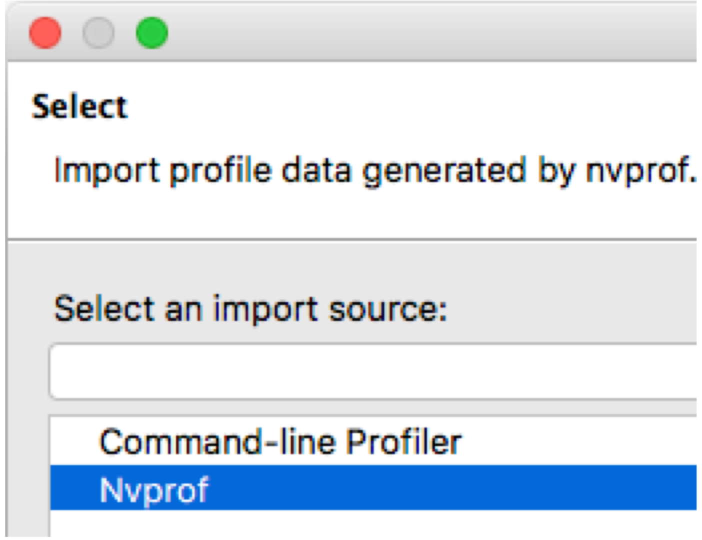
```
$ cat mat_mul.12233
...

==127243== Generated result file: /gpfs/wolf/stf007/scratch/t4p/nvidia_profilers/redundant_MM/mat_mul.12233.h49n16.1.nvvp
==127242== Generated result file: /gpfs/wolf/stf007/scratch/t4p/nvidia_profilers/redundant_MM/mat_mul.12233.h49n16.0.nvvp

(N = 2048) Max Total Time: 3.524076 Max GPU Time: 0.308476
Rank 000, HWThread 008, GPU 0, Node h49n16 - Total Time: 3.520249 GPU Time: 0.308134
Rank 001, HWThread 054, GPU 1, Node h49n16 - Total Time: 3.524076 GPU Time: 0.308476

...
```

Redundant Matrix Multiply – Visual Profiler

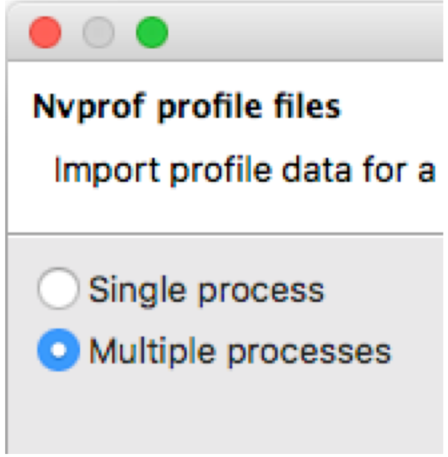
1 File->Import



2 Select "Nvprof" then "Next >"



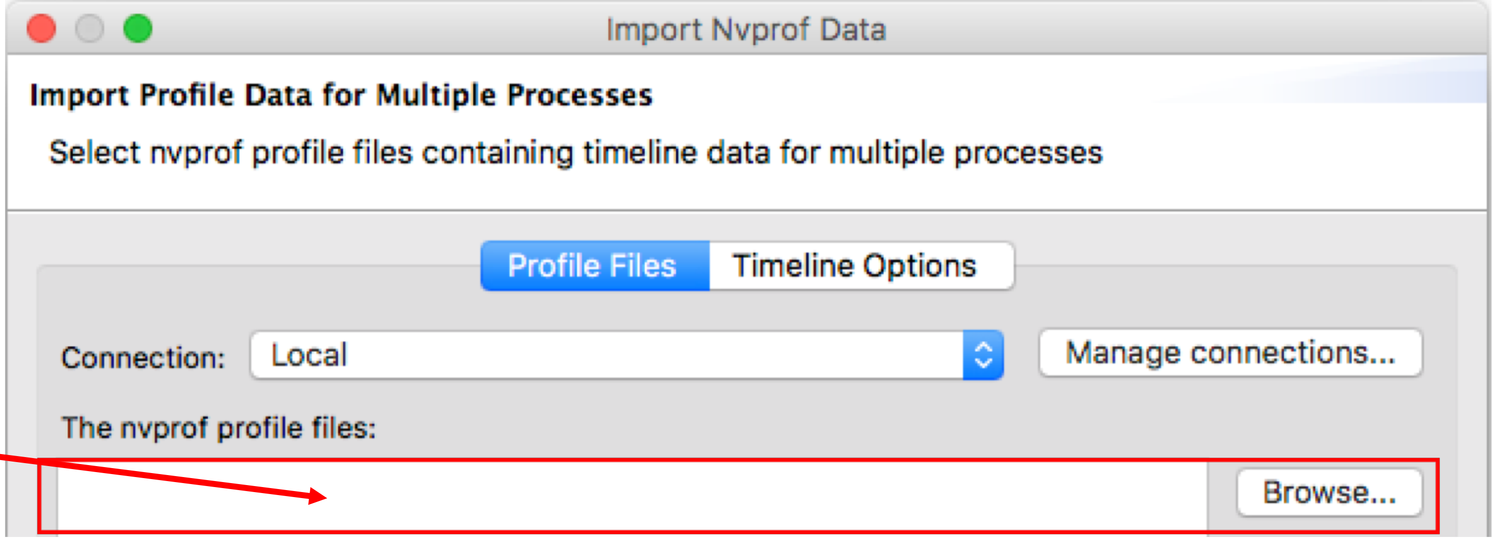
3 Select "Multiple Process" then "Next >"



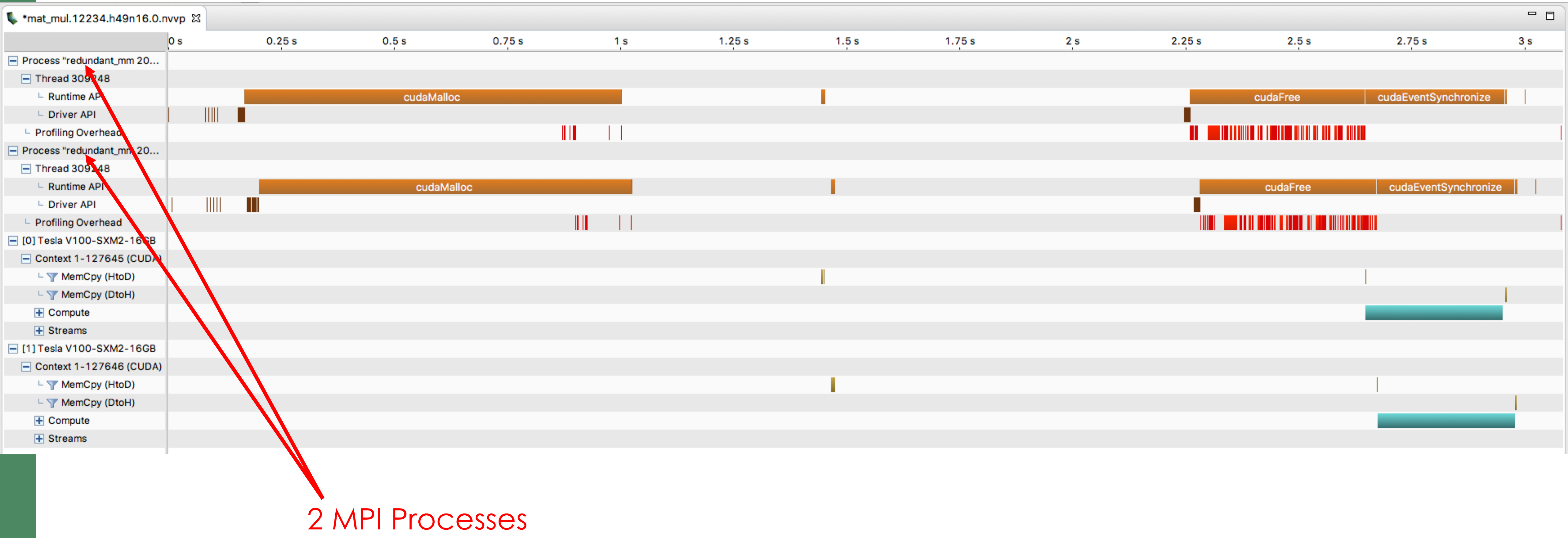
4

Click "Browse" next to "Timeline data file" to locate the .nvvp files on your local system, then click "Finish"

NOTE: Here you select multiple files



Redundant Matrix Multiply – Visual Profiler



Multiple MPI Ranks

Run the code

```
$ bsub submit_named.lsf
```

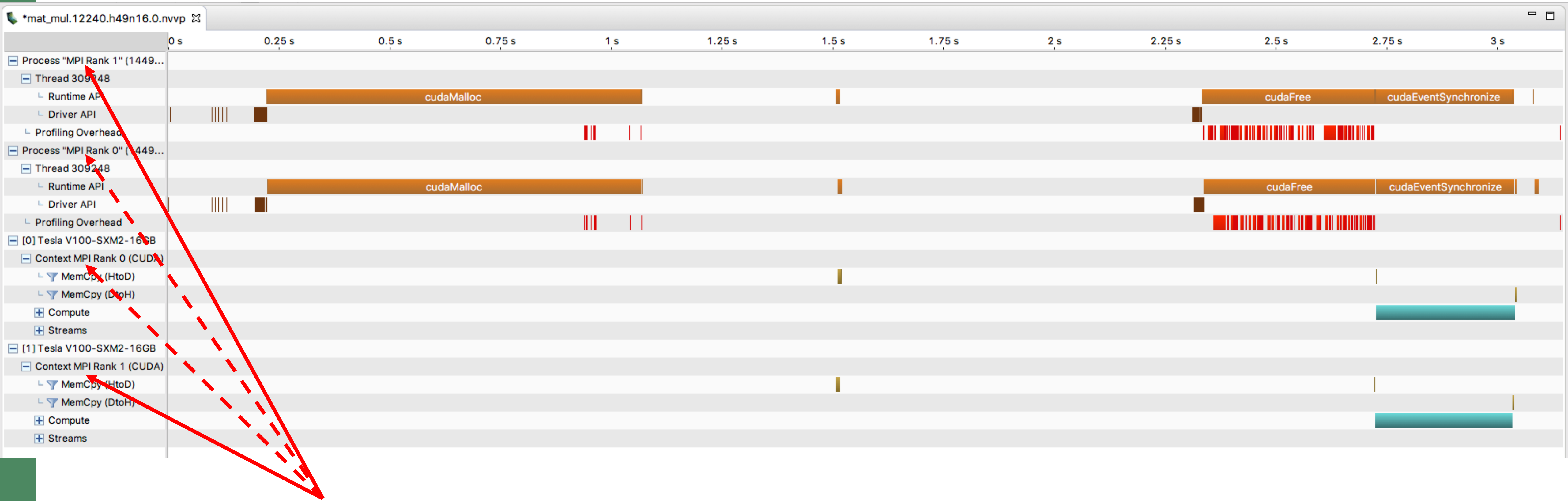
From submit.lsf

```
jsrun -n1 -c42 -g6 -a2 -bpacked:7 \
nvprof -s -o mat_mul.${LSB_JOBID}.%h.%q{OMPI_COMM_WORLD_RANK}.nvvp \
--context-name "MPI Rank %q{OMPI_COMM_WORLD_RANK}" \
--process-name "MPI Rank %q{OMPI_COMM_WORLD_RANK}" ./redundant_mm 2048 100 | sort
```

Name the Process and CUDA Context

```
$ cat mat_mul.12240
...

==144939== Generated result file: /gpfs/wolf/stf007/scratch/t4p/nvidia_profilers/redundant_MM/mat_mul.12240.h49n16.0.nvvp
==144938== Generated result file: /gpfs/wolf/stf007/scratch/t4p/nvidia_profilers/redundant_MM/mat_mul.12240.h49n16.1.nvvp

(N = 2048) Max Total Time: 3.634345 Max GPU Time: 0.311632

Rank 000, HWThread 024, GPU 0, Node h49n16 - Total Time: 3.634345 GPU Time: 0.311632
Rank 001, HWThread 053, GPU 1, Node h49n16 - Total Time: 3.622655 GPU Time: 0.310216

...
```

Redundant Matrix Multiply – Visual Profiler



2 MPI Processes, but now we can tell which is associated with visual profiler sections

Multiple MPI Ranks (annotating with NVTX)

`redundant_MM_nvtx`

Redundant Matrix Multiply – Visual Profiler + NVTX

```
#include <nvToolsExt.h>

// Color definitions for nvtx calls
#define CLR_RED     0xFFFF0000
#define CLR_BLUE    0xFF0000FF
#define CLR_GREEN   0xFF008000
#define CLR_YELLOW  0xFFFFFF00
#define CLR_CYAN    0xFF00FFFF
#define CLR_MAGENTA 0xFFFF00FF
#define CLR_GRAY    0xFF808080
#define CLR_PURPLE  0xFF800080


// Macro for calling nvtxRangePushEx
#define RANGE_PUSH(range_title,range_color) {         \
    nvtxEventAttributes_t eventAttrib = {0};          \
    eventAttrib.version = NVTX_VERSION;               \
    eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE; \
    eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;\
    eventAttrib.colorType = NVTX_COLOR_ARGB;          \
    eventAttrib.color = range_color;                  \
    eventAttrib.message.ascii = range_title;          \
    nvtxRangePushEx(&eventAttrib);                    \
}


// Macro for calling nvtxRangePop
#define RANGE_POP {\
    nvtxRangePop();\
}
```

```
/* ------------------------------------------------------------------
    Fill arrays on CPU
--------------------------------------------------------------------*/


RANGE_PUSH("Initialize Arrays (CPU)", CLR_BLUE);


// Max size of random double
double max_value = 10.0;


// Set A, B, and C
for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
        A[i*N + j] = (double)rand()/(double)(RAND_MAX/max_value);
        B[i*N + j] = (double)rand()/(double)(RAND_MAX/max_value);
        C[i*N + j] = 0.0;
    }
}


RANGE_POP;
```

And added the following NVIDIA Tools Extension library to the Makefile: `-lnvToolsExt`

Multiple MPI Ranks

Compile the code

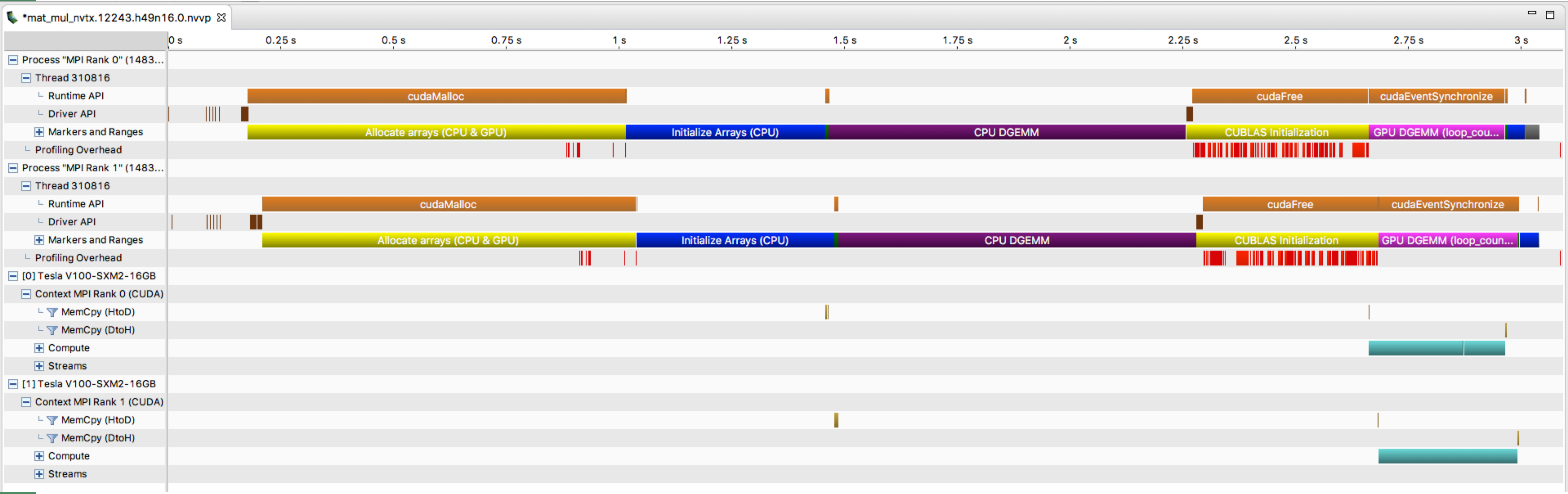
```
$ make
```

Run the code

```
$ bsub submit.lsf
```

Same process as previous version of the code

Redundant Matrix Multiply – Visual Profiler



Now we have a better (and fuller) mapping to what is happening in our code.

Multiple MPI Ranks (Unified Memory)

`redundant_MM_UM`

Redundant Matrix Multiply – Visual Profiler + UM + NVTX

```
/* ---------------------------------------------------------------
    Allocate memory for arrays on CPU and GPU
--------------------------------------------------------------- */

RANGE_PUSH("Allocate CPU and UM arrays", CLR_YELLOW);

// Allocate memory for C_cpu on CPU
double *C_cpu = (double*)malloc(N*N*sizeof(double));

// Allocate memory for A, B, C for use on both CPU and GPU
double *A, *B, *C;
cudaErrorCheck( cudaMallocManaged(&A, N*N*sizeof(double)) );
cudaErrorCheck( cudaMallocManaged(&B, N*N*sizeof(double)) );
cudaErrorCheck( cudaMallocManaged(&C, N*N*sizeof(double)) );

RANGE_POP;
```

```
/* ---------------------------------------------------------------
    Transfer data from CPU to GPU
--------------------------------------------------------------- */

// No explictit data transfer required for arrays allocated with cudaMallocManaged

/* ---------------------------------------------------------------
    Transfer data from GPU to CPU
--------------------------------------------------------------- */

// No explictit data transfer required for arrays allocated with cudaMallocManaged
```

Then use the common pointers on both CPU and GPU

Multiple MPI Ranks

<u>Compile the code</u>

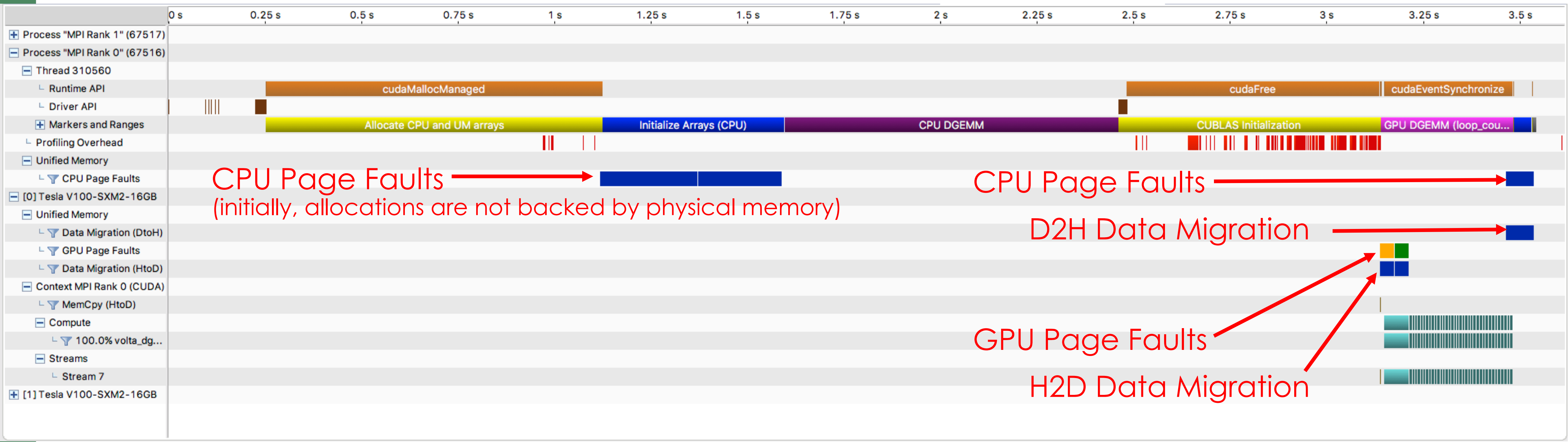
```
$ make
```

<u>Run the code</u>

```
$ bsub submit.lsf
```

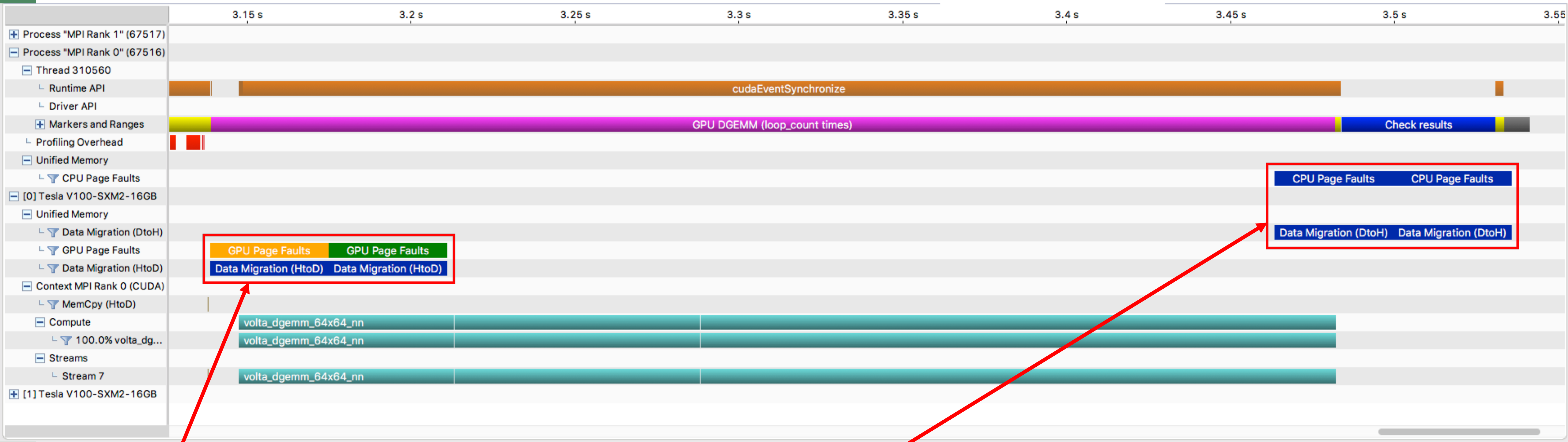
Same process as previous version of the code

Redundant Matrix Multiply – Visual Profiler



Oak Ridge National Laboratory

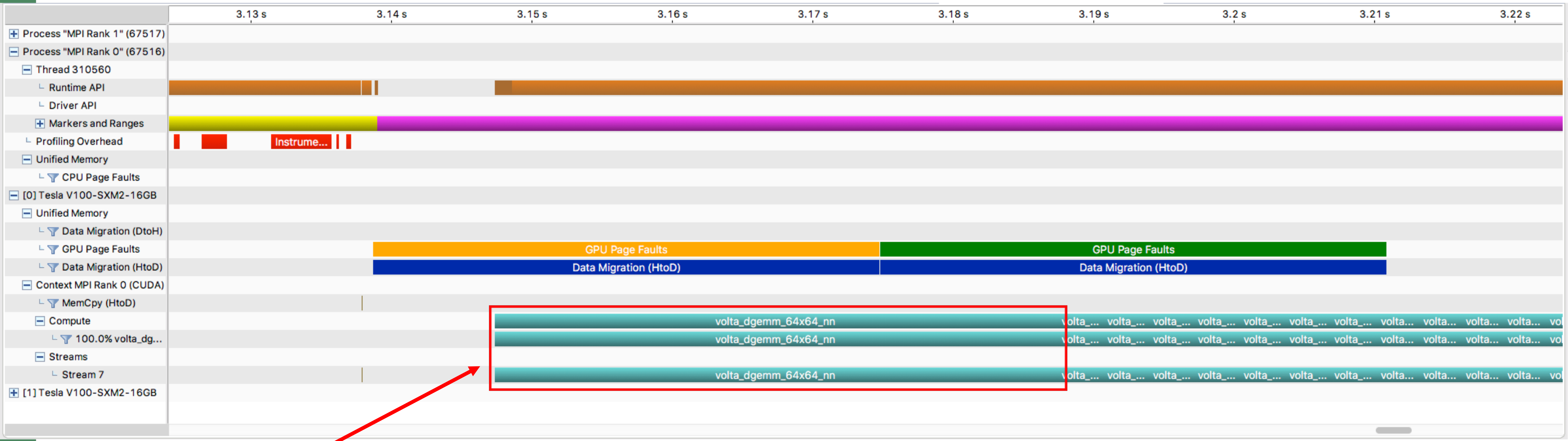
Redundant Matrix Multiply – Visual Profiler



When data is needed on GPU (for the first GPU DGEMM), GPU page faults trigger data migration from CPU to GPU.

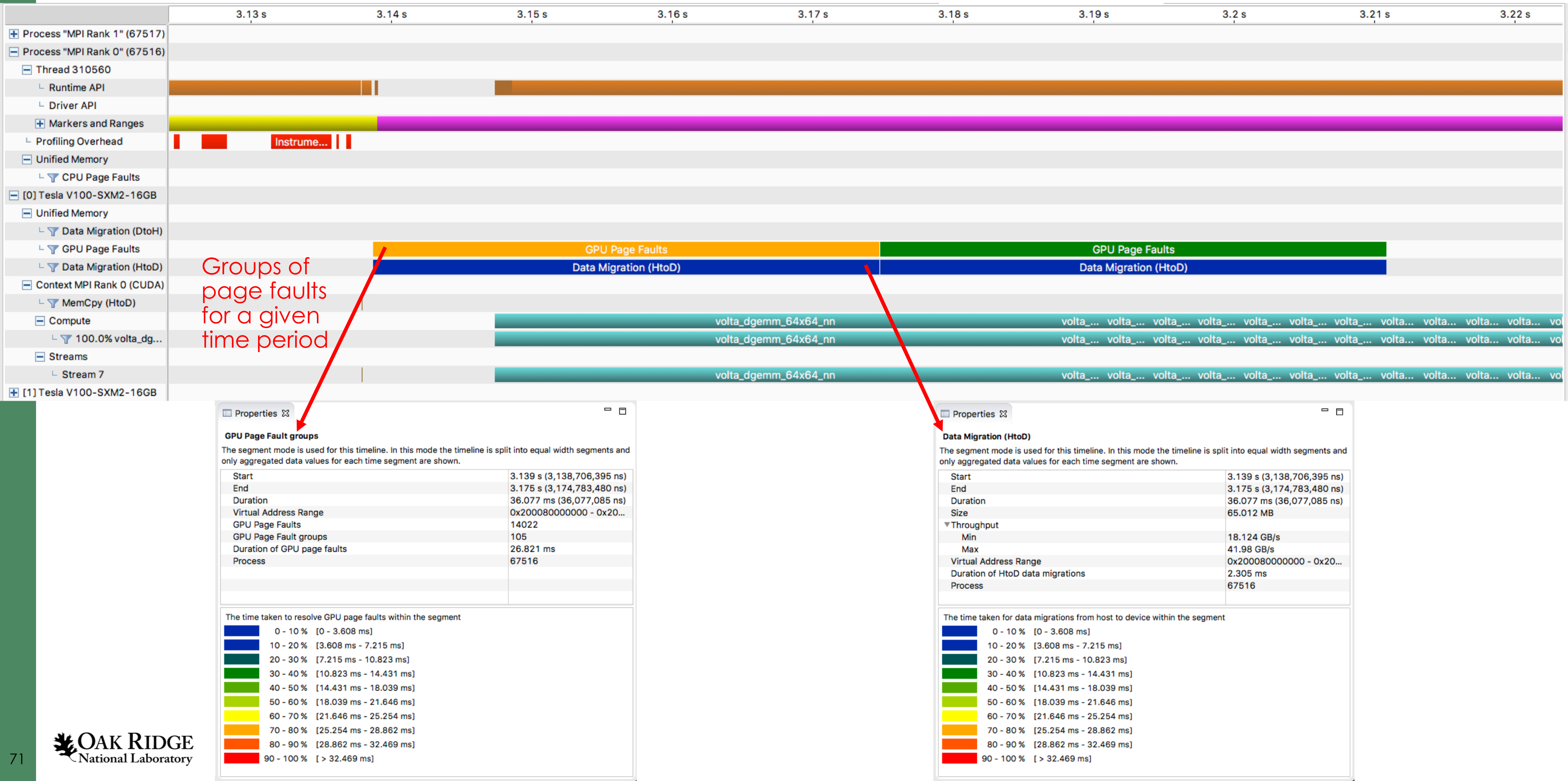
When data is needed on CPU (to compare CPU/GPU results), CPU page faults trigger data migration from GPU to CPU.

Redundant Matrix Multiply – Visual Profiler



The time for the 1st GPU DGEMM is increased due to page faults and data migration, while subsequent calls are not since data is already on the GPU

Redundant Matrix Multiply – Visual Profiler



Kernel Analysis

Kernel Analysis – Gathering Details Remotely

1. Gather a timeline for a ***short*** run.

```
$ jsrun --smpiargs="none" -n1 -c1 -g1 -a1 nvprof -fo
single_gpu_data.timeline100.nvprof ./run
```

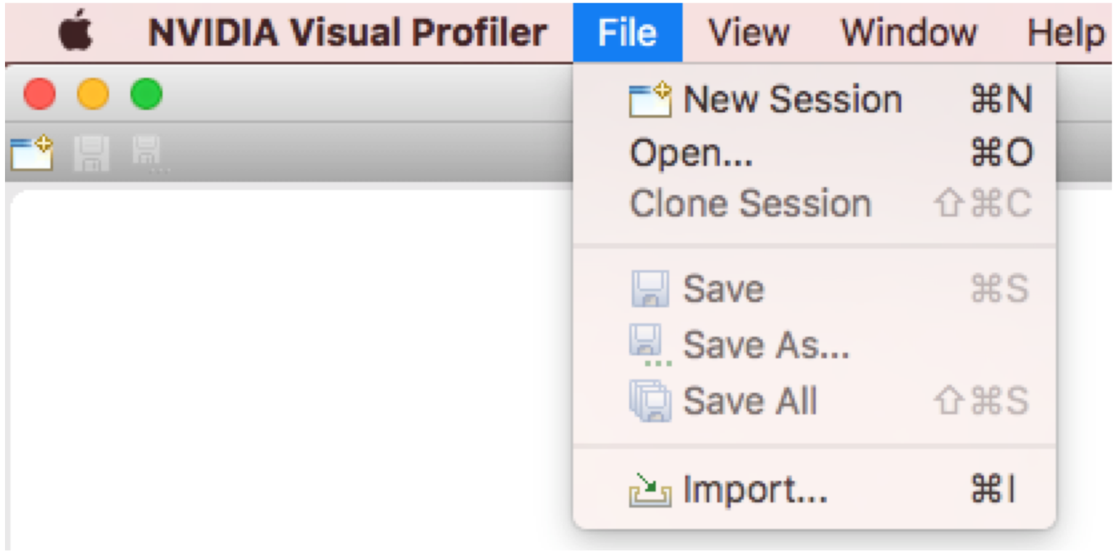
2. Gather matching "analysis metrics" (Runtime will explode due to each kernel being replayed multiple times.

```
$ jsrun --smpiargs="none" -n1 -c1 -g1 -a1 nvprof --analysis-metrics -fo
single_gpu_data.metrics100.nvprof ./run
```

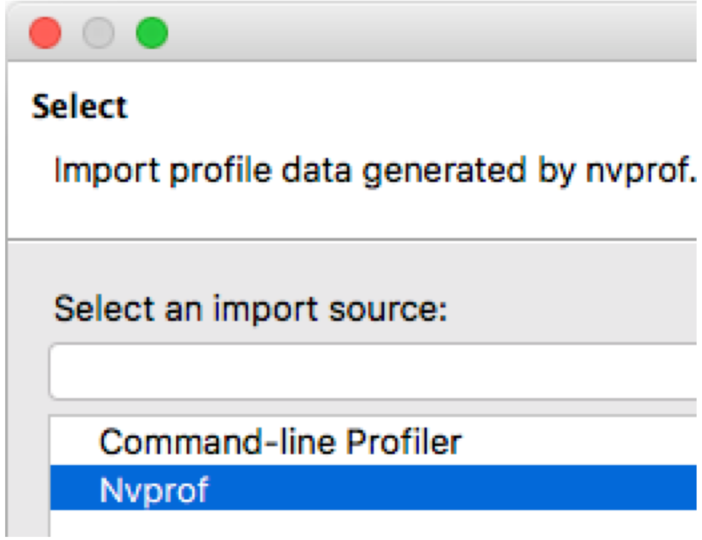
If you cannot shorten your run any longer, it's possible to use the **--kernels** option to only replay some kernels, but guided analysis may not work as well.

Kernel Details – Import into Visual Profiler

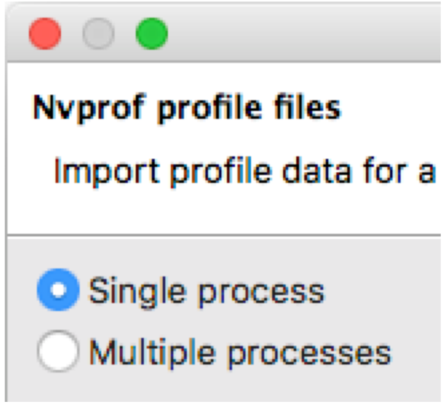
1 File->Import



2 Select “Nvprof” then “Next >”

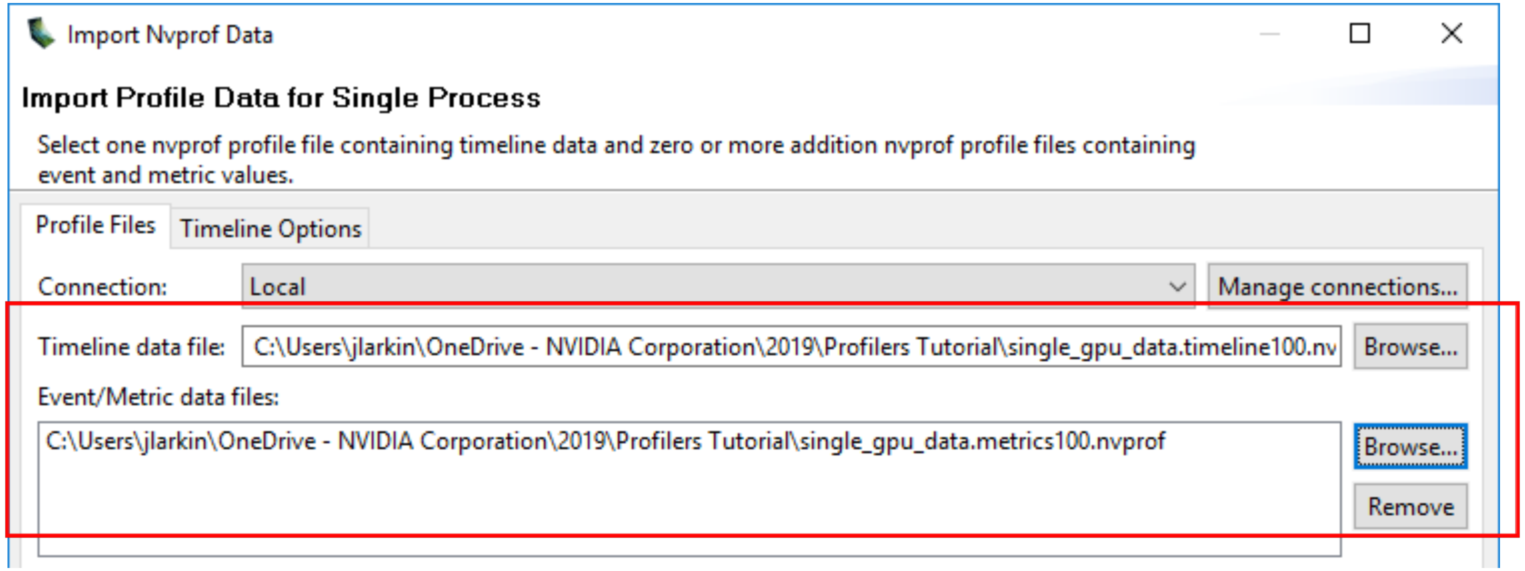


3 Select “Single Process” then “Next >”

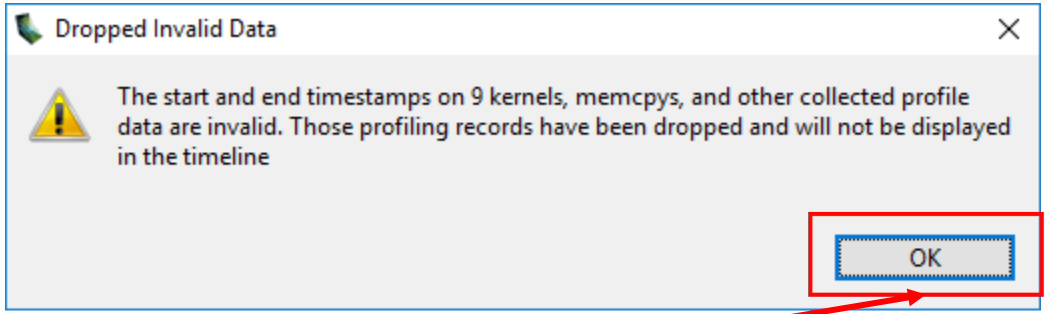


4

Click “Browse” next to “Timeline data file” to locate the .nvprof file on your local system, then do the same for “Event/Metric data files,” then click “Finish”

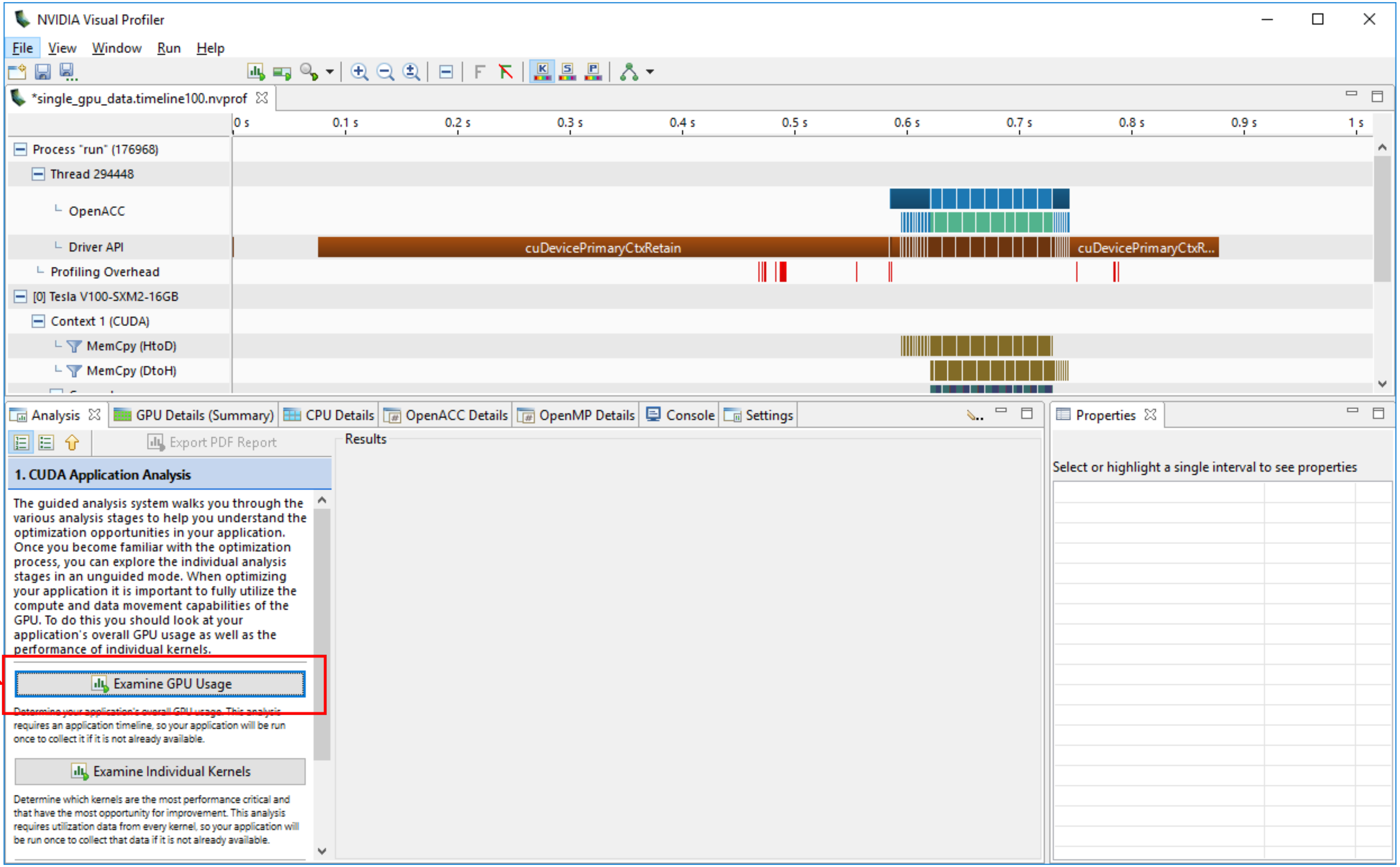


Visual Profiler Import – Common Warning



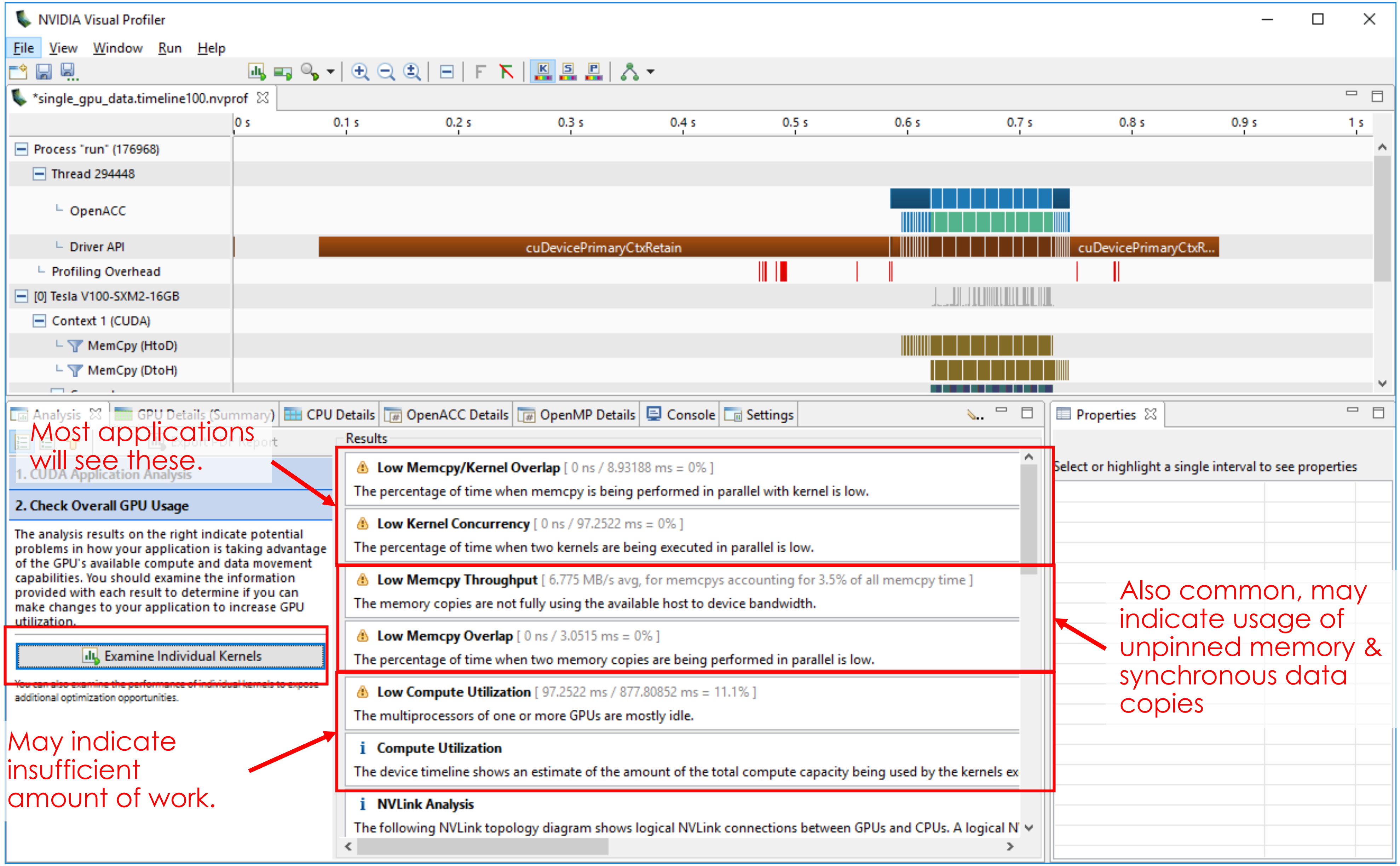
This warning is very common when importing both timelines and metrics, particularly on very short runs. It can be safely ignored.

Visual Profiler – Guided Analysis



Start with a high-level overview of application performance.

Visual Profiler – Guided Analysis

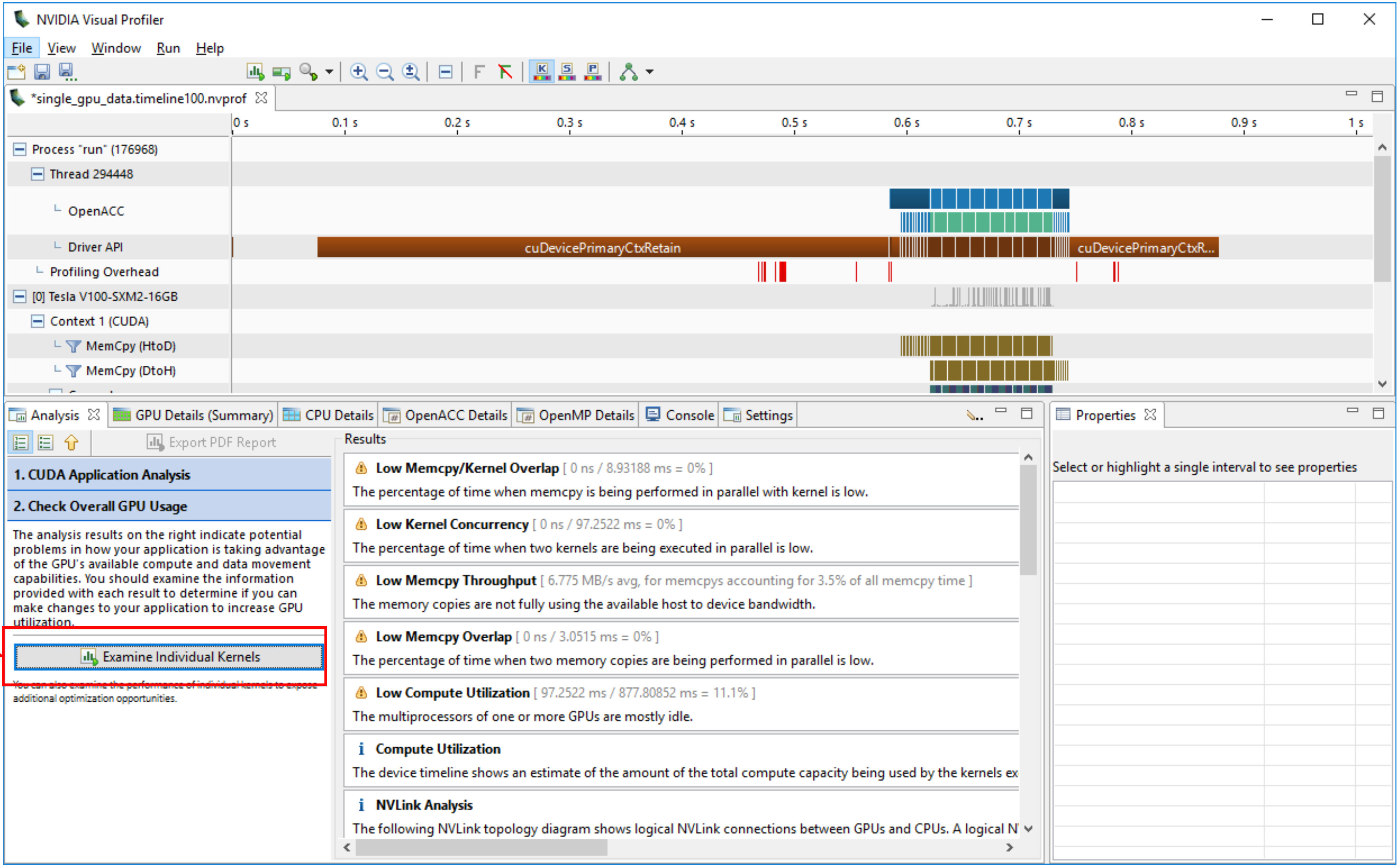


Most applications will see these.

Also common, may indicate usage of unpinned memory & synchronous data copies

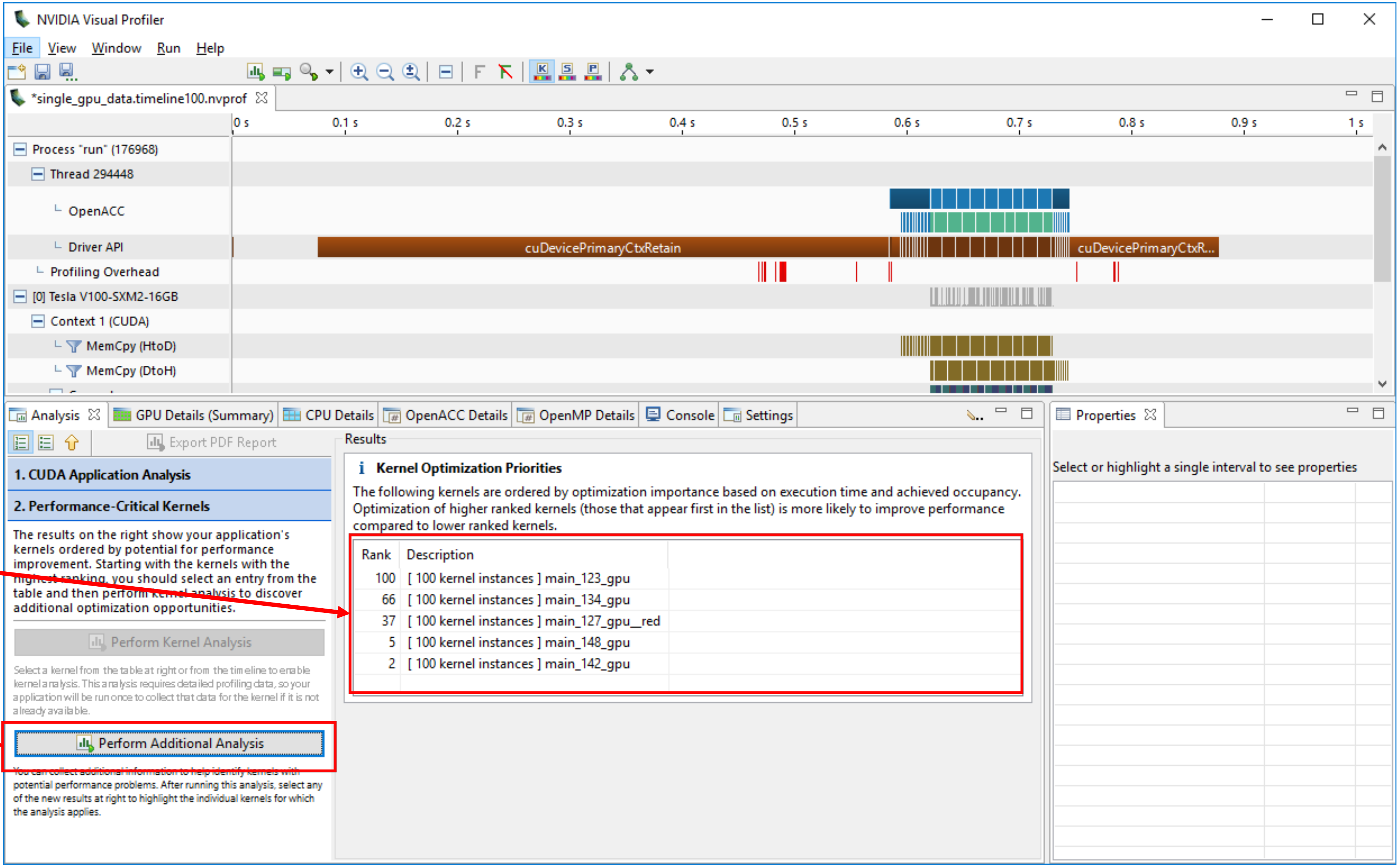
May indicate insufficient amount of work.

Visual Profiler – Guided Analysis



Next zoom in on individual kernel optimizations.

Visual Profiler – Guided Analysis



This table ranks the kernels by bang for buck, click the top one.

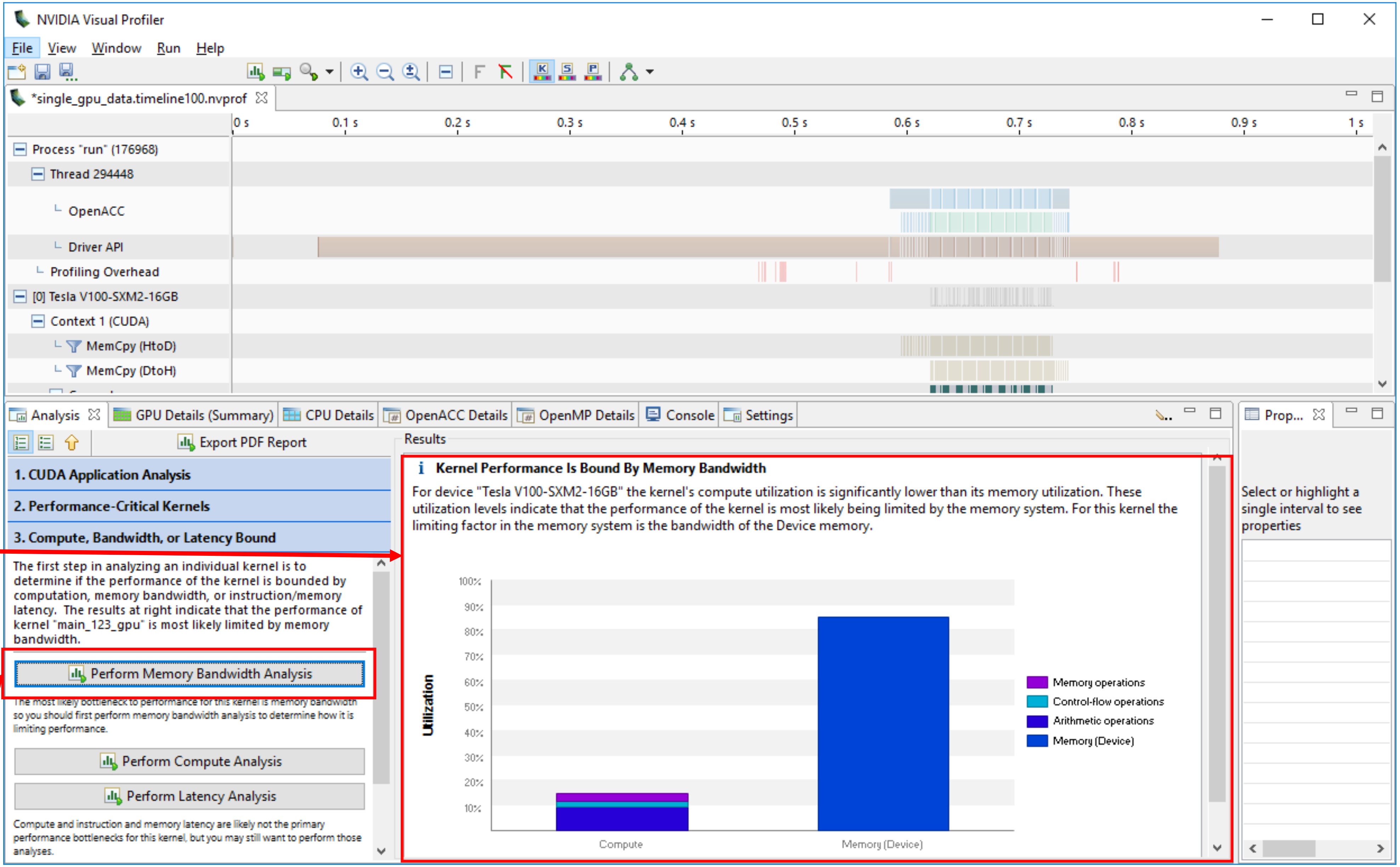
Click here to deep dive on the selected kernel.

OAK RIDGE National Laboratory

Visual Profiler – Guided Analysis – Bandwidth Bound

This box will estimate the performance limiter of your kernel

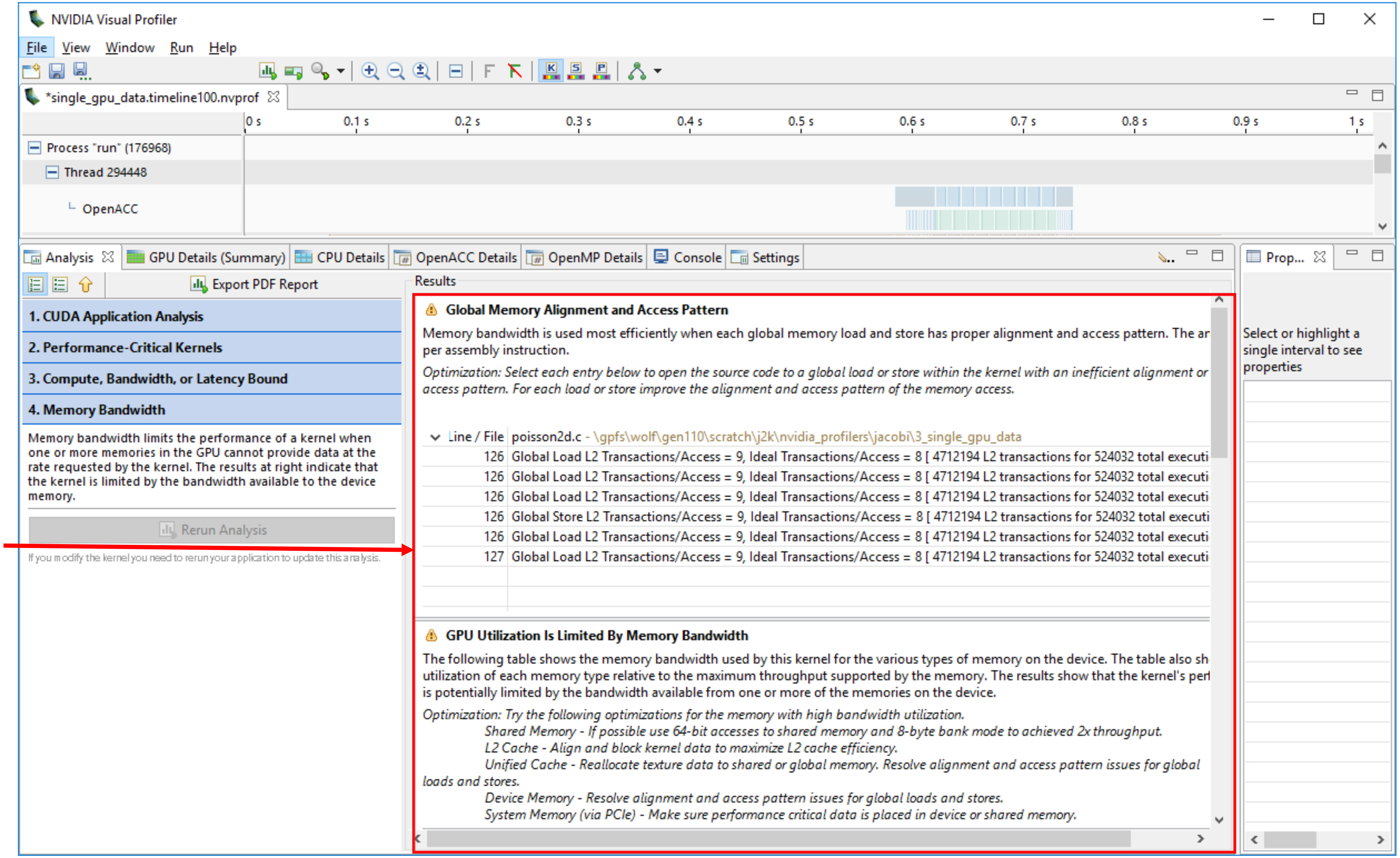
Click here to dive deeper on that performance limiter



OAK RIDGE National Laboratory

Visual Profiler – Guided Analysis – Bandwidth Bound

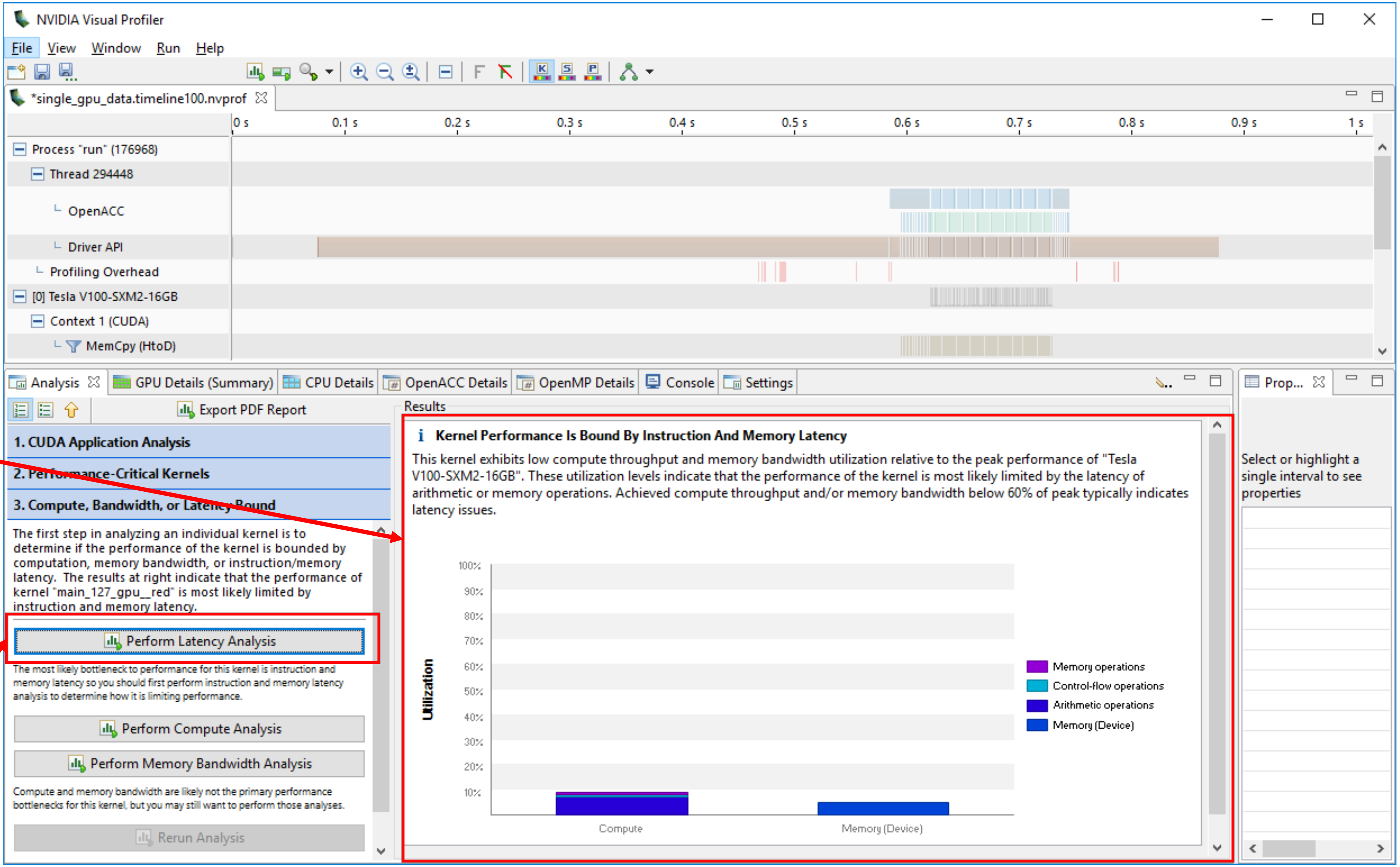
This is the final set of suggestions for this kernel.



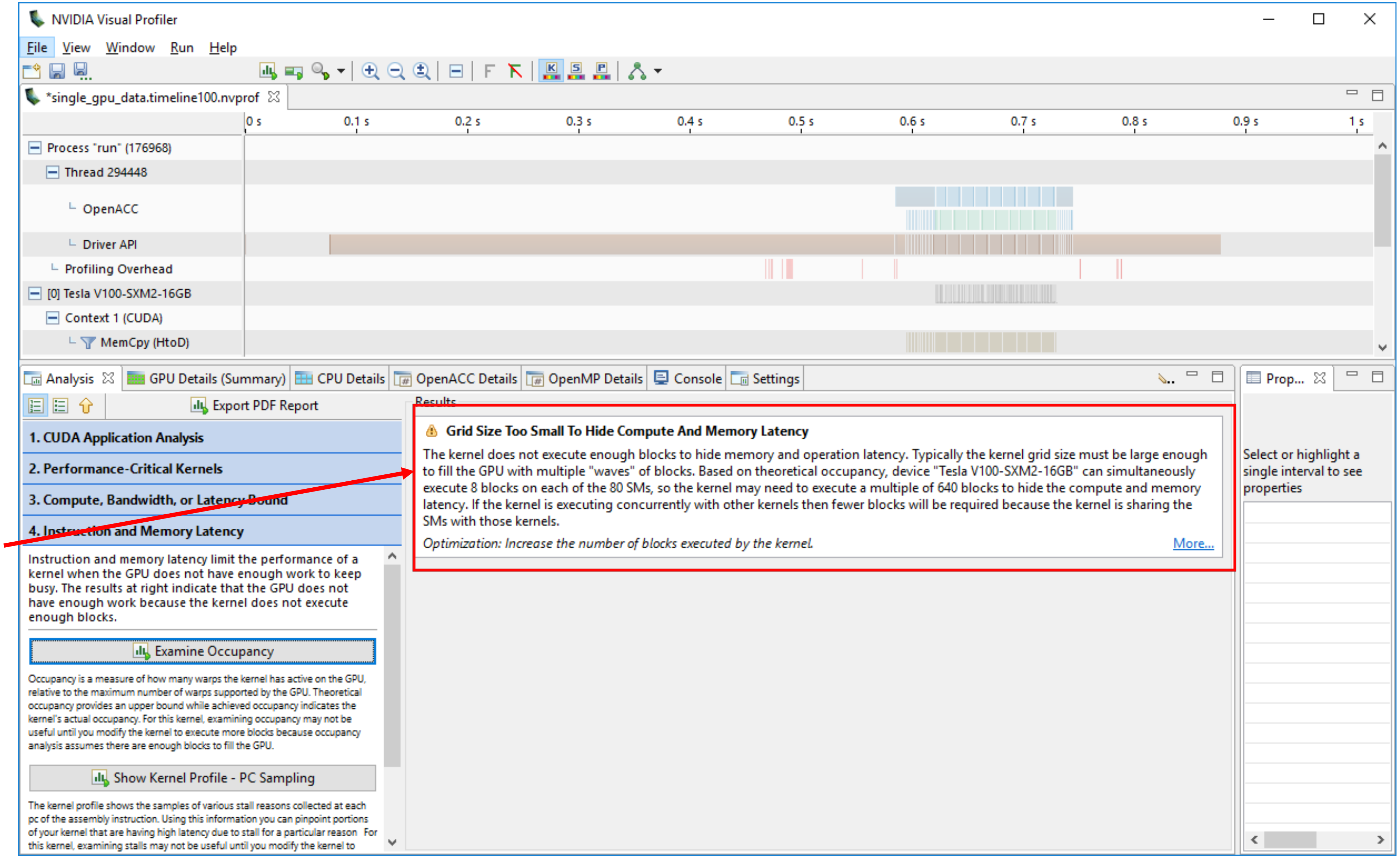
Visual Profiler – Guided Analysis – Latency Bound

Low Compute & Memory utilization points to being latency bound.

Now a latency analysis is suggested

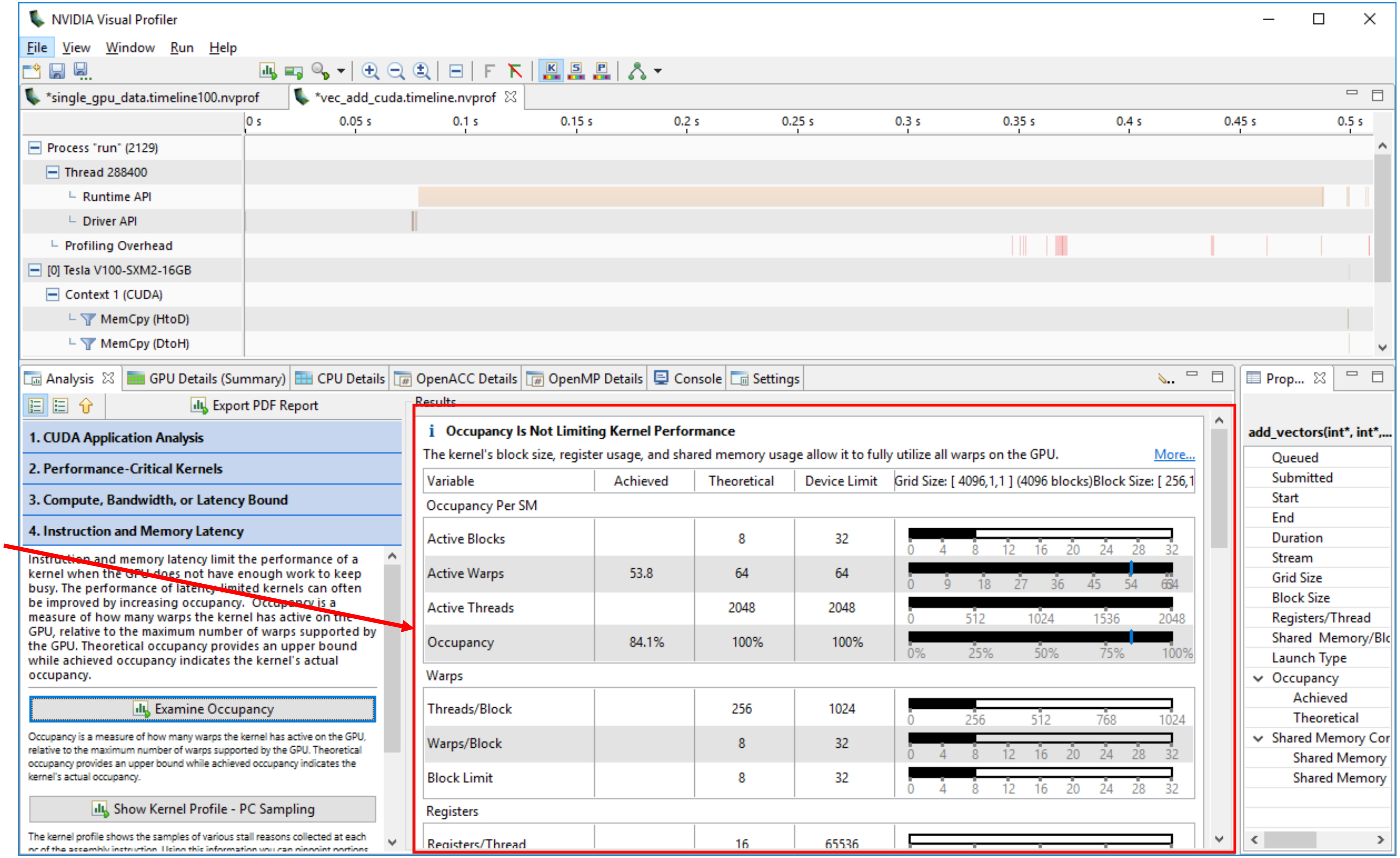


Visual Profiler – Guided Analysis – Latency Bound



The kernel doesn't do enough work for the GPU.

Visual Profiler – Guided Analysis – Latency Bound



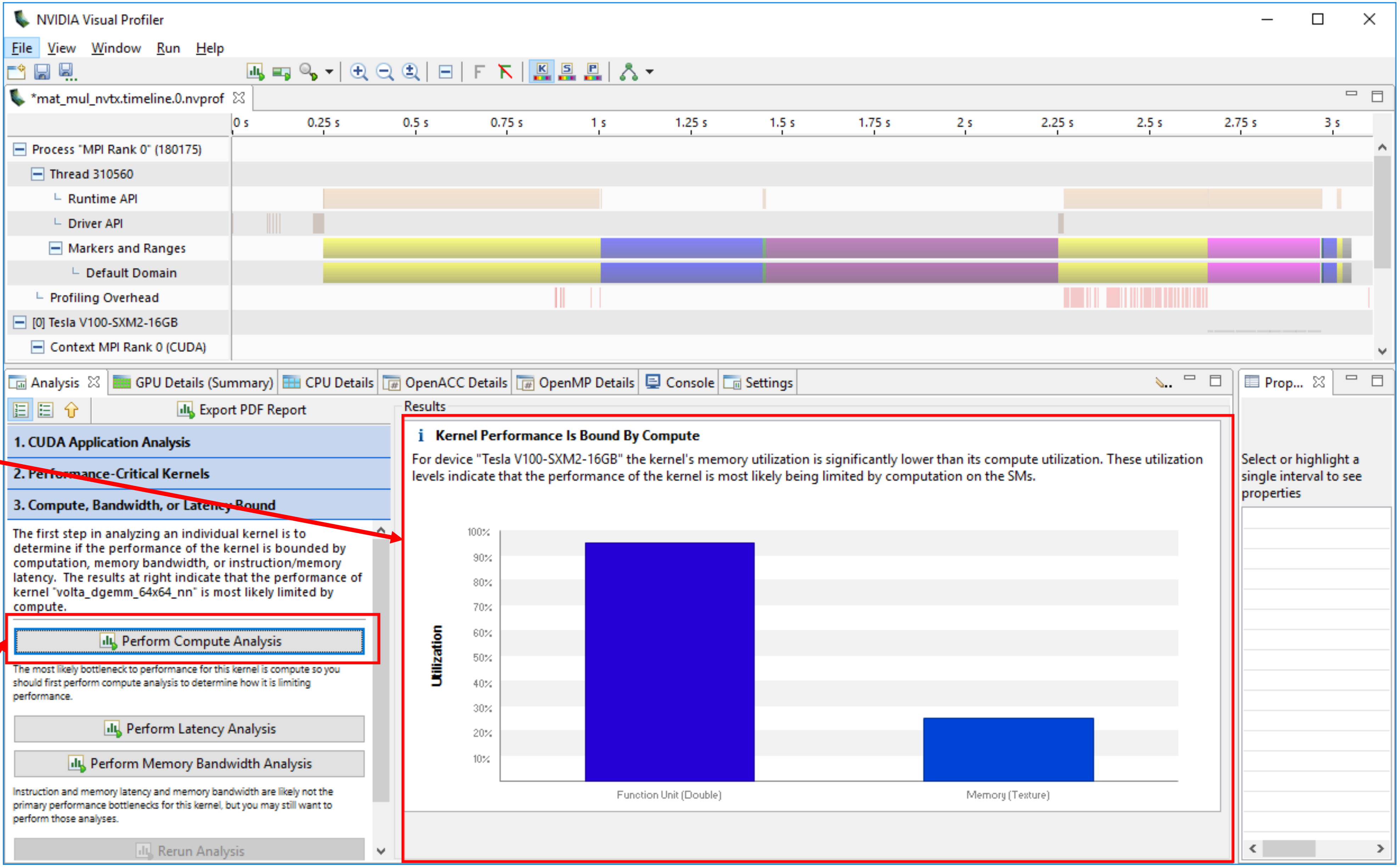
In other cases an occupancy analysis may be performed.

OAK RIDGE National Laboratory

Visual Profiler – Guided Analysis – Compute Bound

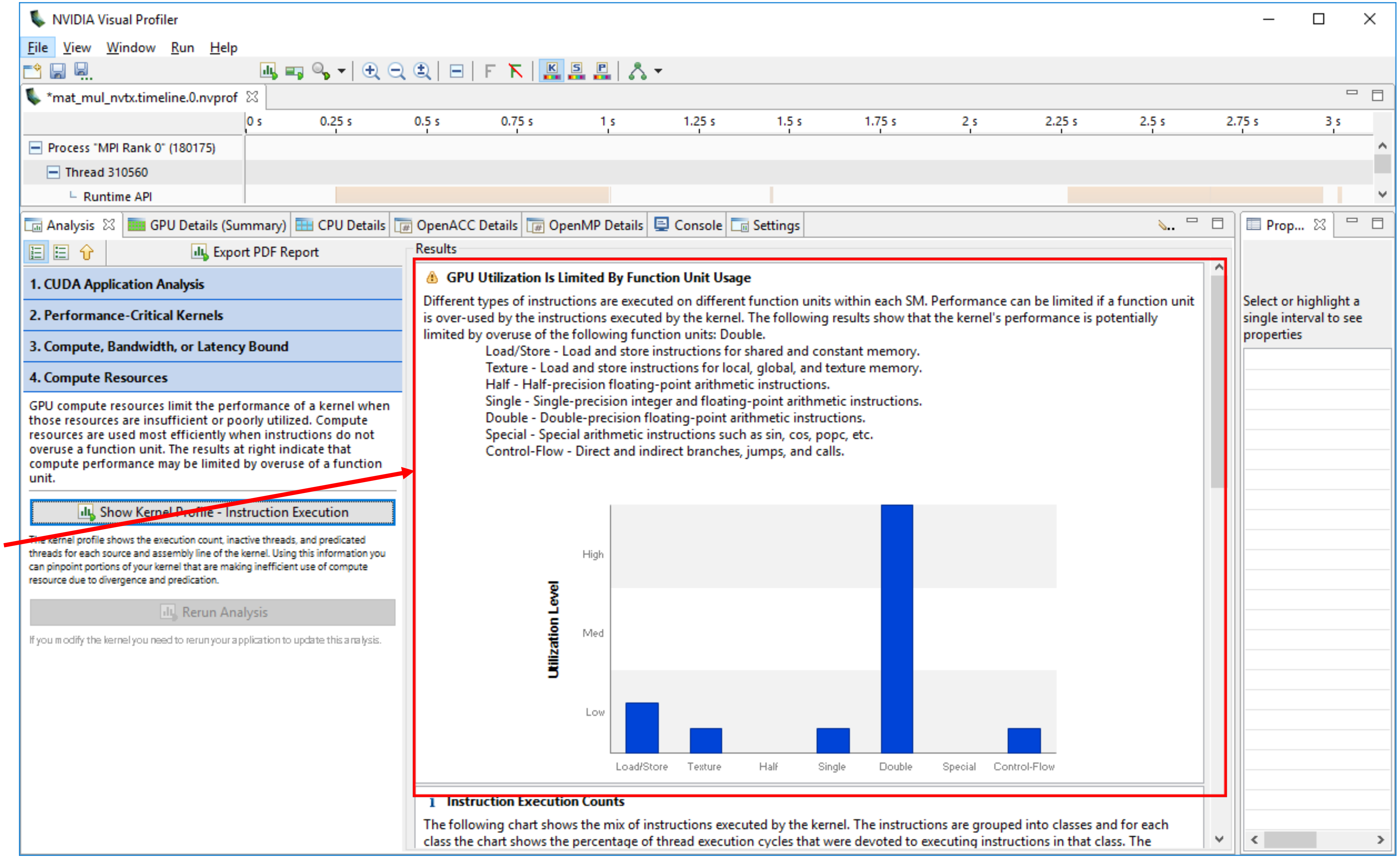
High compute utilization indicates the kernel is compute bound.

Now a compute analysis is suggested



OAK RIDGE National Laboratory

Visual Profiler – Guided Analysis – Compute Bound



This kernel performs a lot of double precision math.

"Poor Man's" Guided Analysis

- Sometimes you can get enough information from a simple nvprof run to get you started.
- Utilization will be shown as a scale from 1 (Low) to 10 (Max)

```
$ jsrun -n1 -c1 -g1 -a1 nvprof -m dram_utilization,l2_utilization,double_precision_fu_utilization,achieved_occupancy ./redundant_mm
2048 100
==13250== NVPROF is profiling process 13250, command: ./redundant_mm 2048 100
==13250== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==13250== Profiling application: ./redundant_mm 2048 100
(N = 2048) Max Total Time: 10.532436 Max GPU Time: 8.349185
Rank 000, HWThread 002, GPU 0, Node h49n16 - Total Time: 10.532436 GPU Time: 8.349185
==13250== Profiling result:
==13250== Metric result:
Invocations                          Metric Name                          Metric Description         Min         Max         Avg
Device "Tesla V100-SXM2-16GB (0)"
    Kernel: volta_dgemm_64x64_nn
        100                     dram_utilization                   Device Memory Utilization     Low (1)     Low (2)     Low (1)
        100                       l2_utilization                        L2 Cache Utilization     Low (2)     Low (2)     Low (2)
        100      double_precision_fu_utilization  Double-Precision Function Unit Utilization    Max (10)    Max (10)    Max (10)
        100                   achieved_occupancy                          Achieved Occupancy    0.114002    0.120720    0.118229
```

Ideally, something will be "High" or "Max". If everything is "Low", check you have enough work and check occupancy.

Summary

- How to generate text and visual output using the NVIDIA profilers

- The workflow for using the NVIDIA profilers on Summit
  - Generate visual output remotely
  - Scp visual output to local machine
  - Explore using NVIDIA Visual Profiler on local machine

- A simple example of how the text+visual profiles might be used when porting an application to run on GPUs

- How to profile multiple MPI ranks (when not too many!)

- How to insert simple annotations into visual profiles using NVTX

- How to interpret Unified Memory results in the visual profiler

- How to perform remote kernel analysis

Thank You.