NVIDIA PROFILING TOOLS

Jeff Larkin, August 08, 2019; Some slides courtesy Tom Papatheodore (ORNL)

NVIDIA PROFILING ON SUMMIT

NVPROF

Command-line Data Gathering

Simple, high-level text output

Gather hardware metrics

Export data to other tools

VISUAL PROFILER

Graphical display of nvprof data

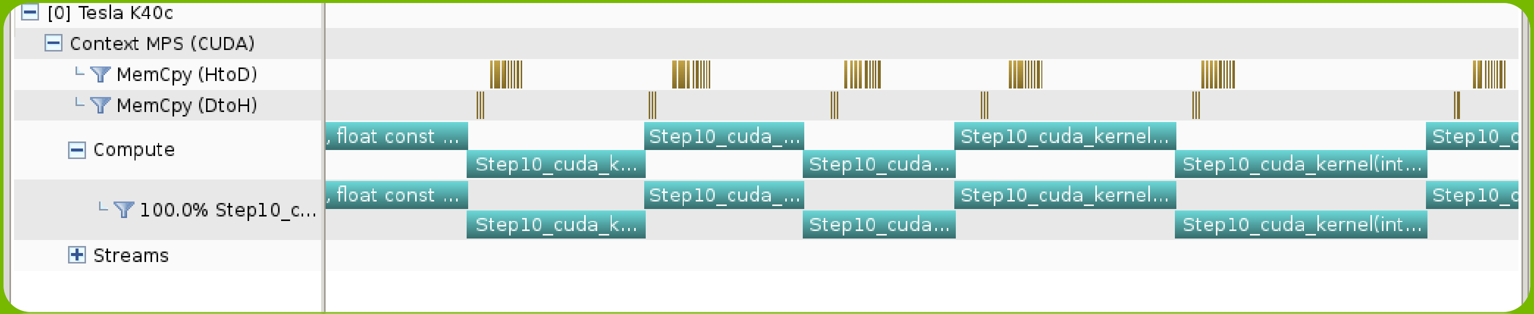
“Big picture” analysis

Very good visualization of data movement and kernel interactions

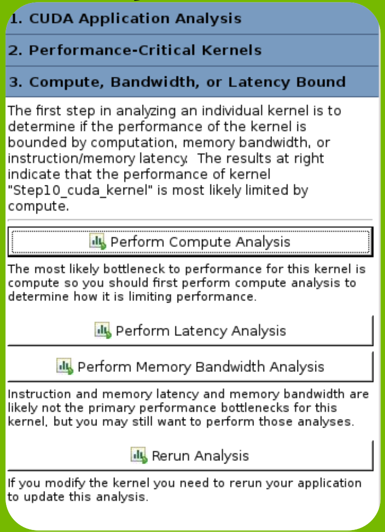
Best run locally from your machine

NVIDIA'S VISUAL PROFILER (NVVP)

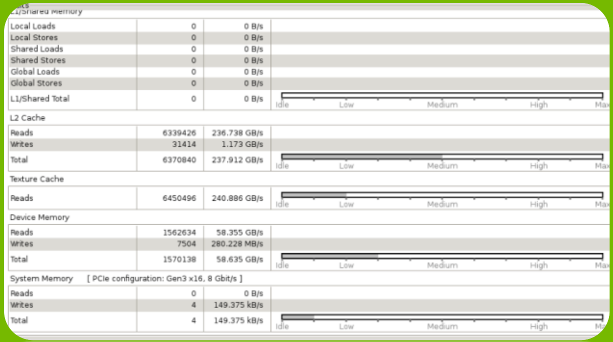
Timeline

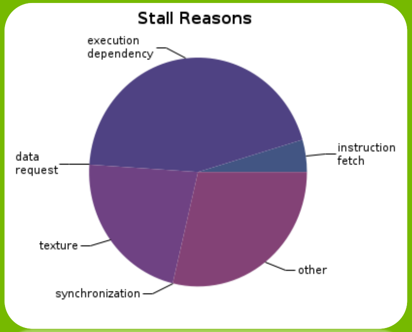


Guided System



Analysis





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

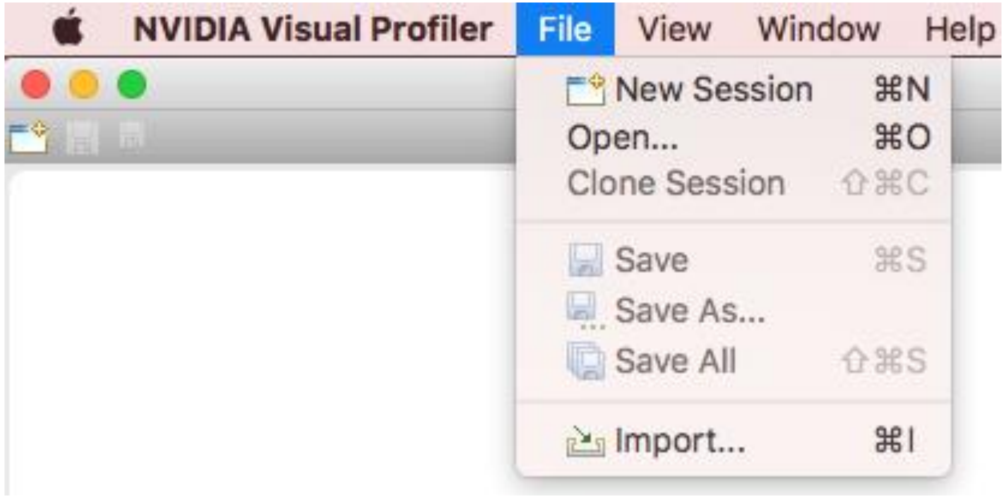
Now, transfer the .nvvp file from Ascent to your local machine to view in NVIDIA Visual Profiler.

From your local system:

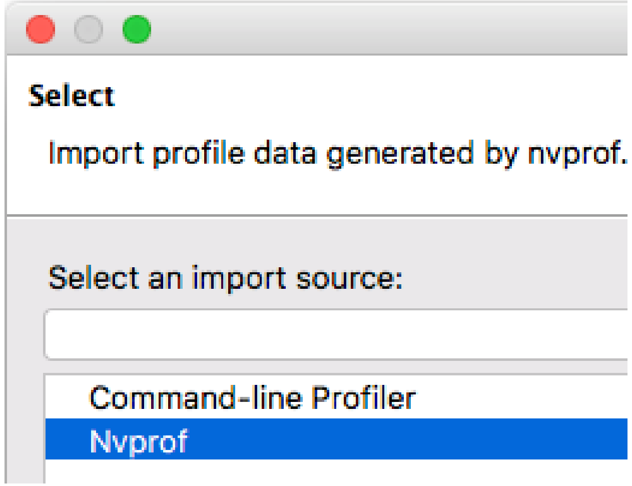
```
$ scp USERNAME@login1.ascent.ccs.ornl.gov:/path/to/file/remote /path/to/desired/location/local
```

VECTOR ADDITION EXAMPLE - VISUAL PROFILER

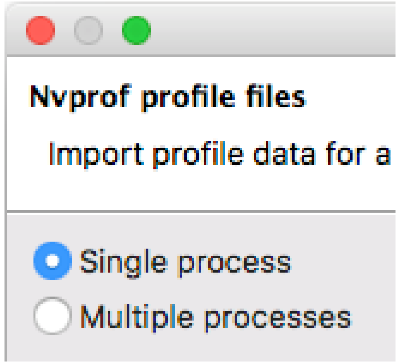
1 File->Import



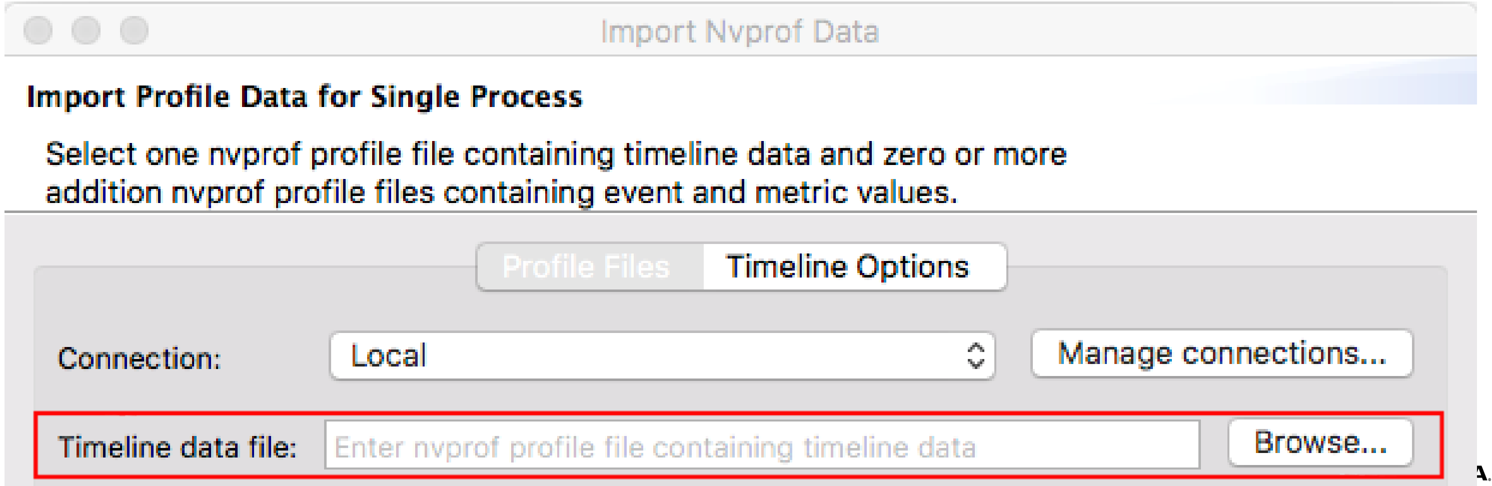
2 Select "Nvprof" then "Next >"



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"

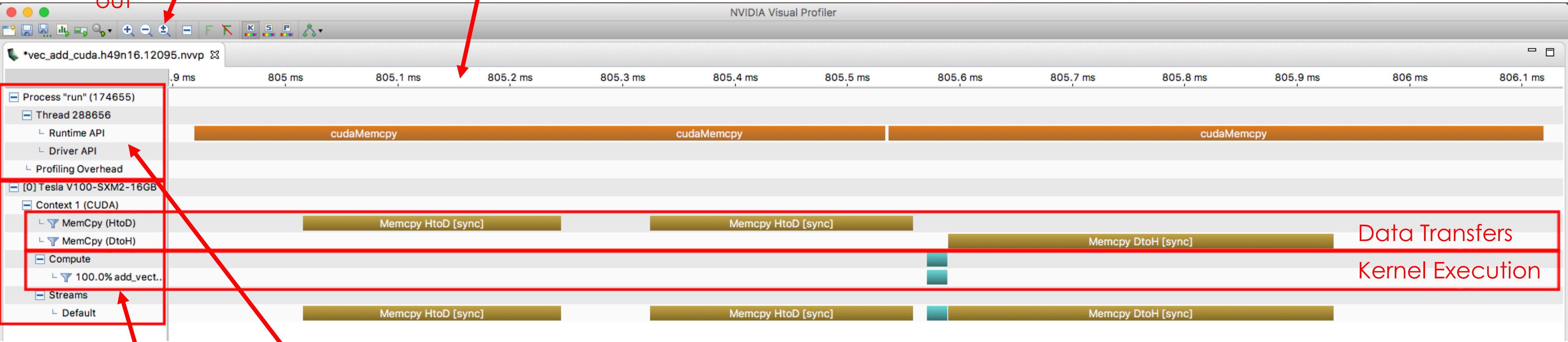


VECTOR ADDITION EXAMPLE - VISUAL PROFILER

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

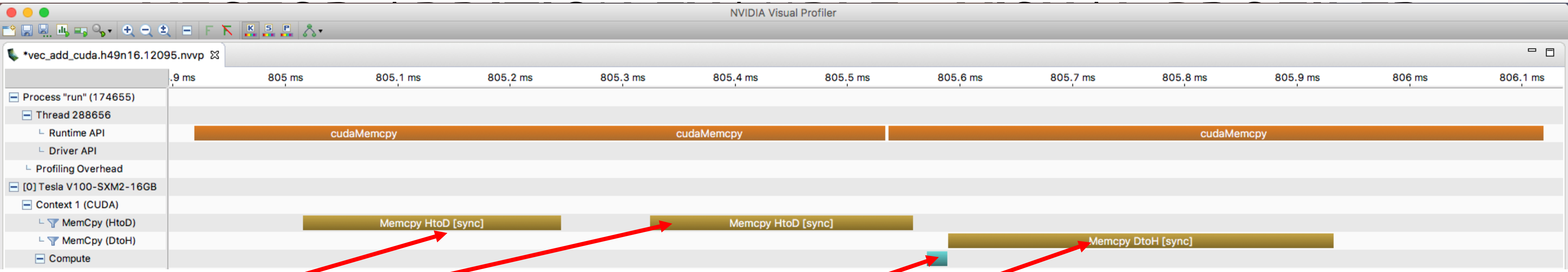


Data Transfers

Kernel Execution

CUDA API activity from CPU process

GPU activity

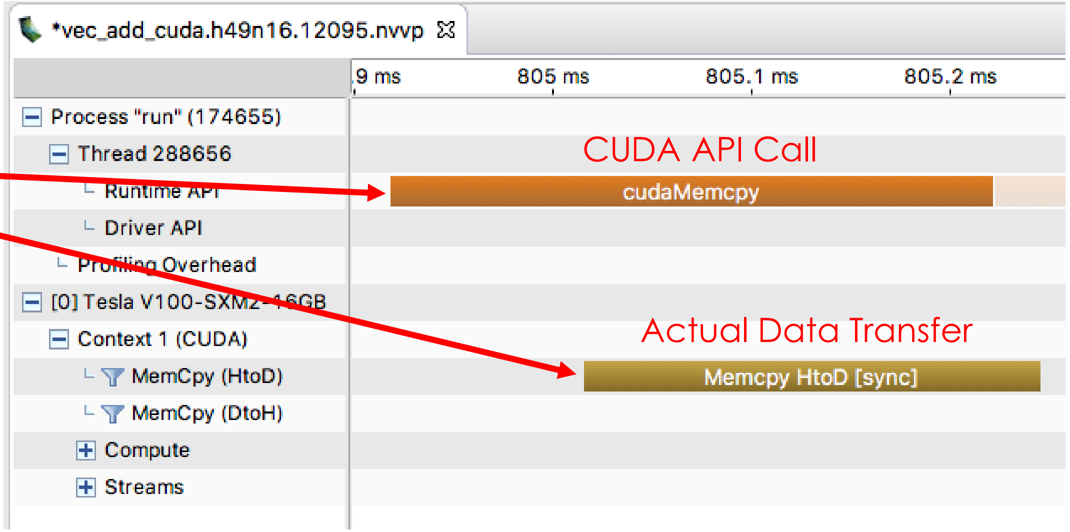



```
// Copy data from host arrays A and B to device arrays d_A and d_B
cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);

// 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);
```

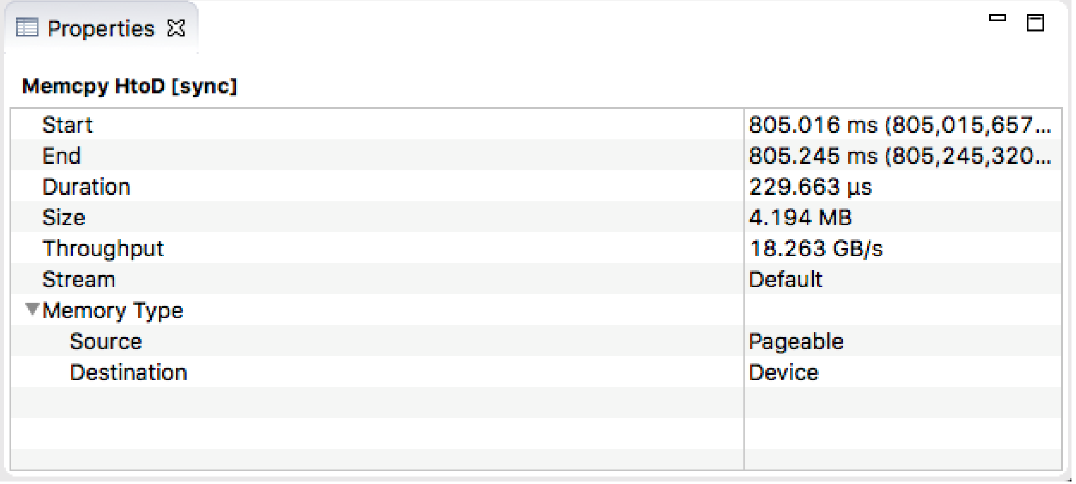
Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	56.25%	463.36us	2	231.68us	229.66us	233.70us	[CUDA memcpy HtoD]
	41.59%	342.56us	1	342.56us	342.56us	342.56us	[CUDA memcpy DtoH]
	2.16%	17.824us	1	17.824us	17.824us	17.824us	add_vectors(int*, int*, int*)

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);
```

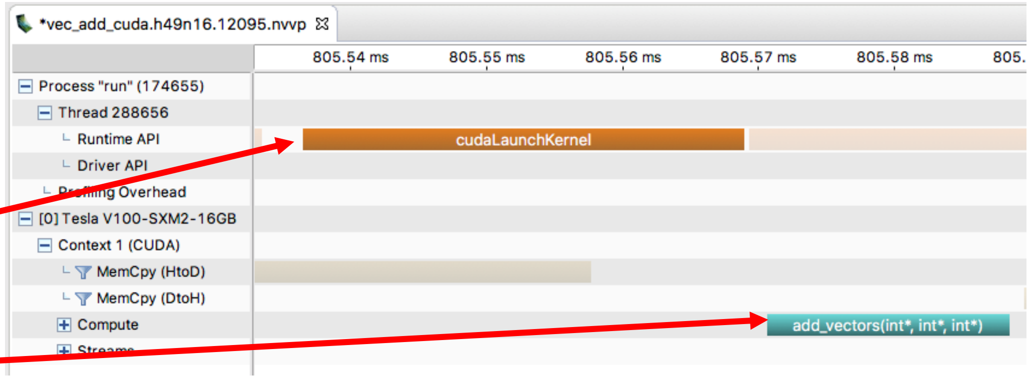


Details about the data transfer



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

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)

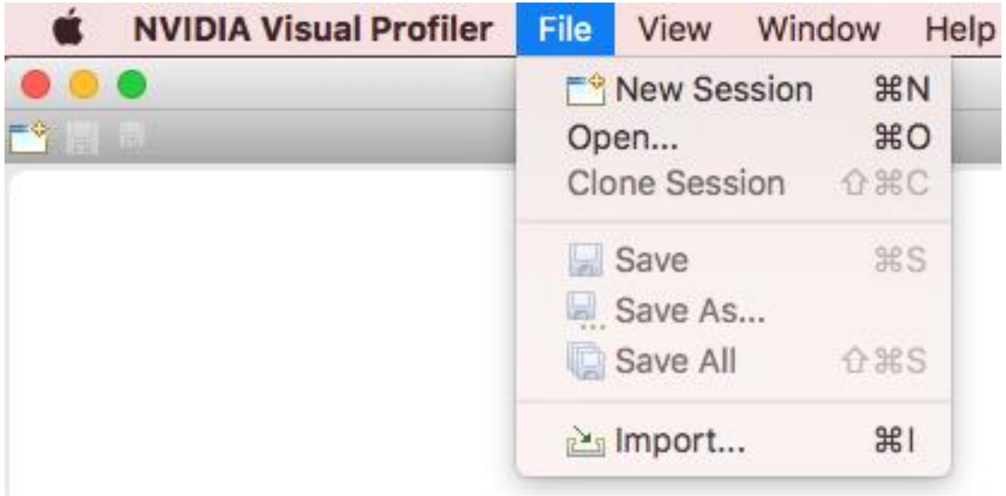
From submit.lsf

```
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
```

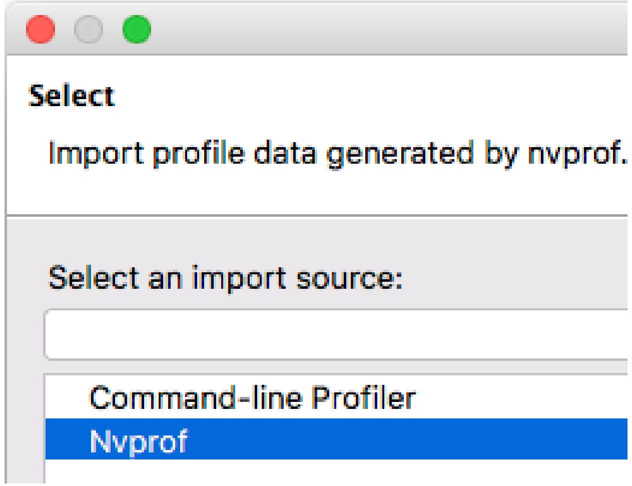
```
$ cat mat_mul.12233
...

==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

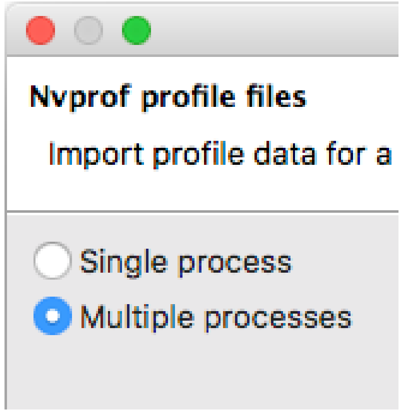
1 File->Import



2 Select “Nvprof” then “Next >”



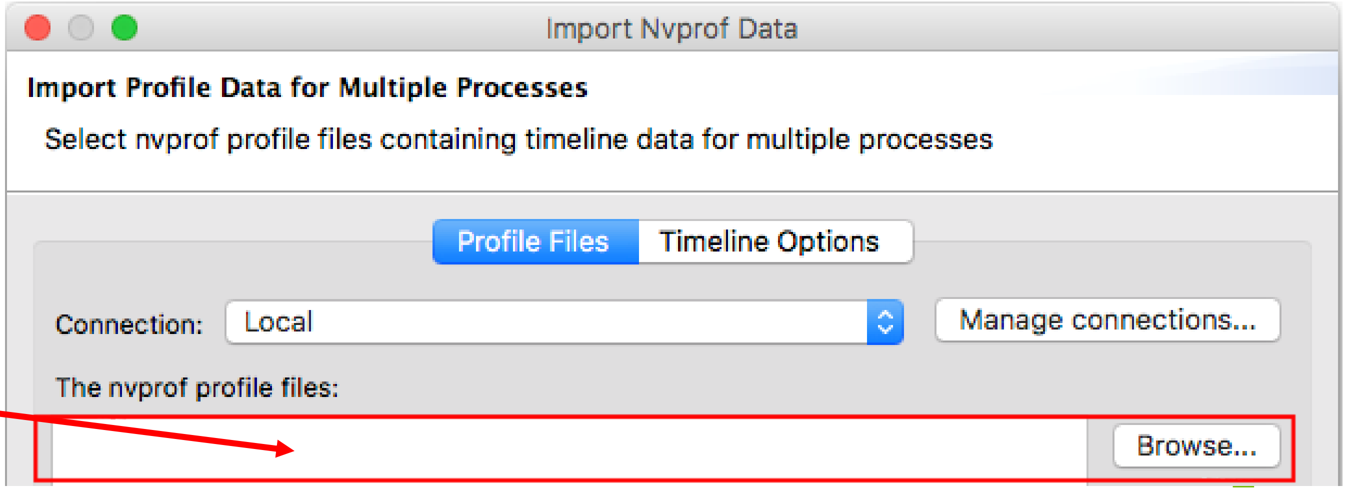
3 Select “Multiple Process” then “Next >”



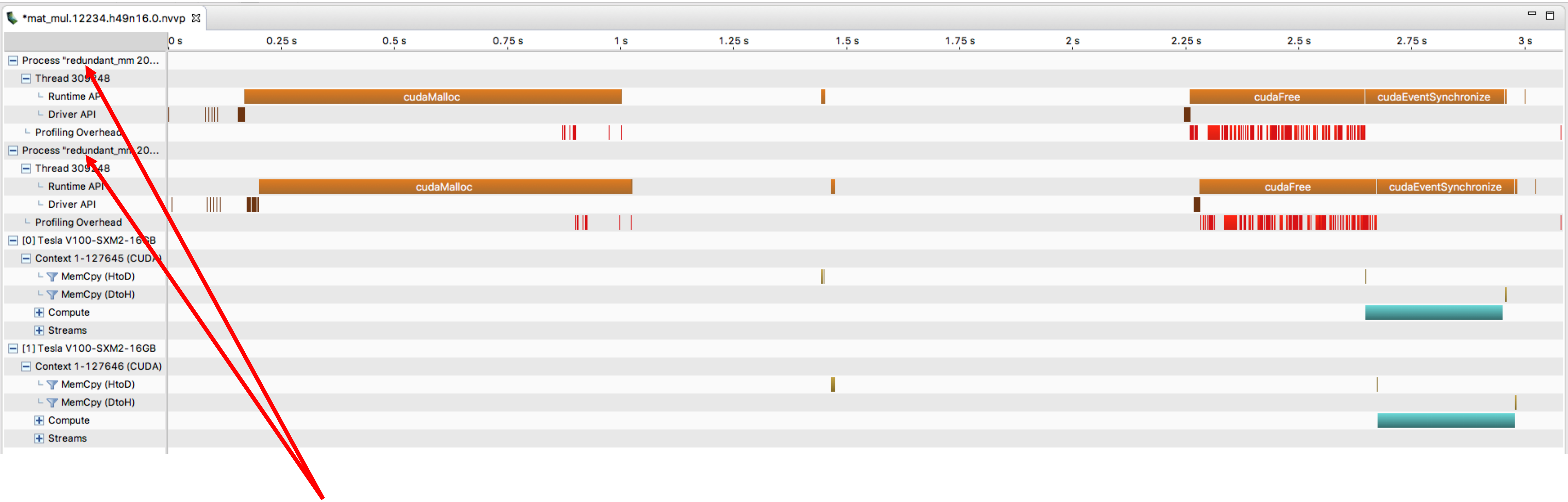
4

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



2 MPI Processes

MULTIPLE MPI RANKS

Run the code

```
$ bsub submit_named.lsf
```

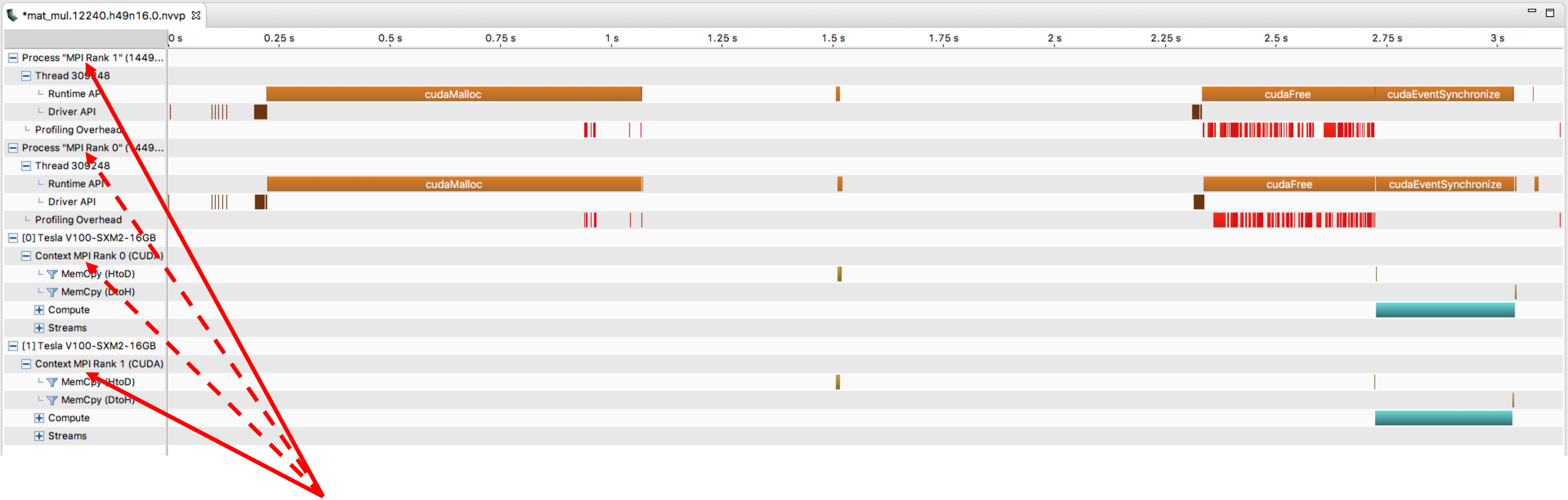
From submit.lsf

```
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`

NVTX ANNOTATIONS

The NVIDIA Tools Extensions (NVTX) allow you to annotate the profile:

```
#include <nvToolsExt.h> // Link with -lnvToolsExt

nvtxRangePushA("timestep");

timestep();

nvtxRangePop();
```

See https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvtx for more features, including V3 usage.

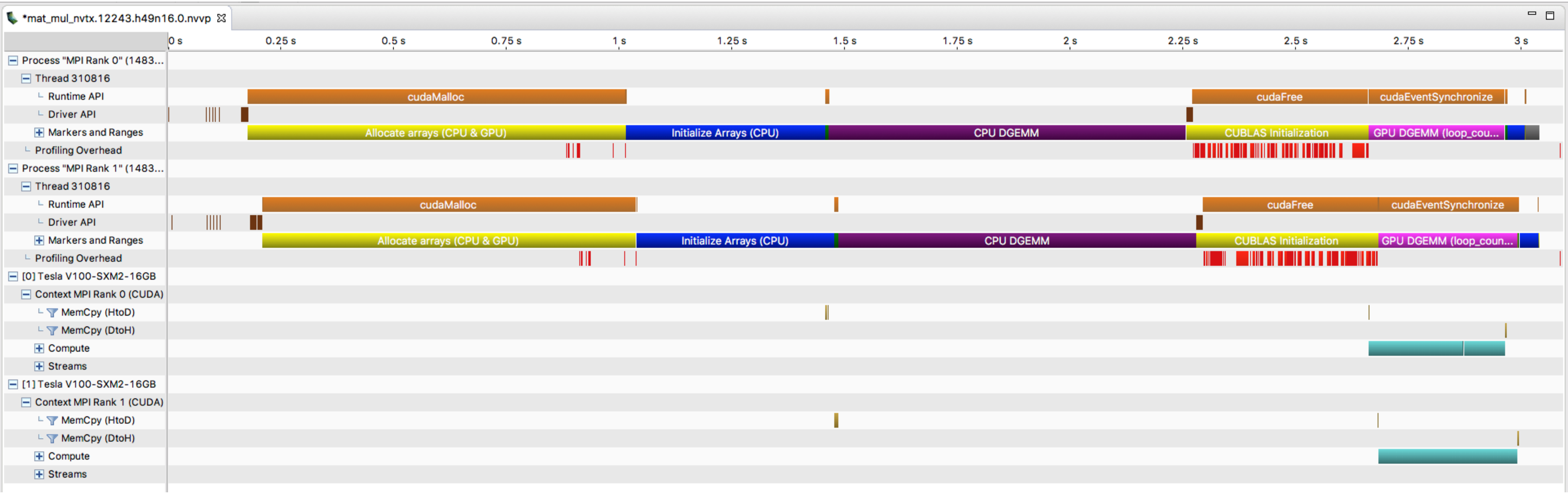
REDUNDANT MATMUL – VISUAL PROFILER + NVTX

```
#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`

REDUNDANT MATMUL – 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`

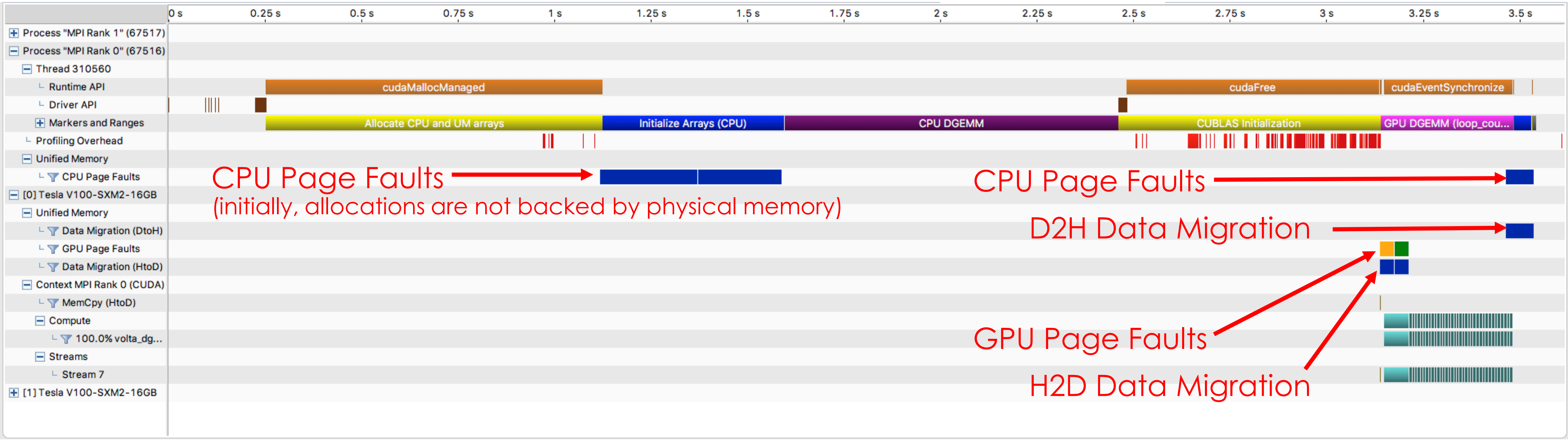
REDUNDANT MATRIX MULTIPLY - VISUAL PROFILER + UM + NVTX

```
/* -------------------------------------------------------------
    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;
```

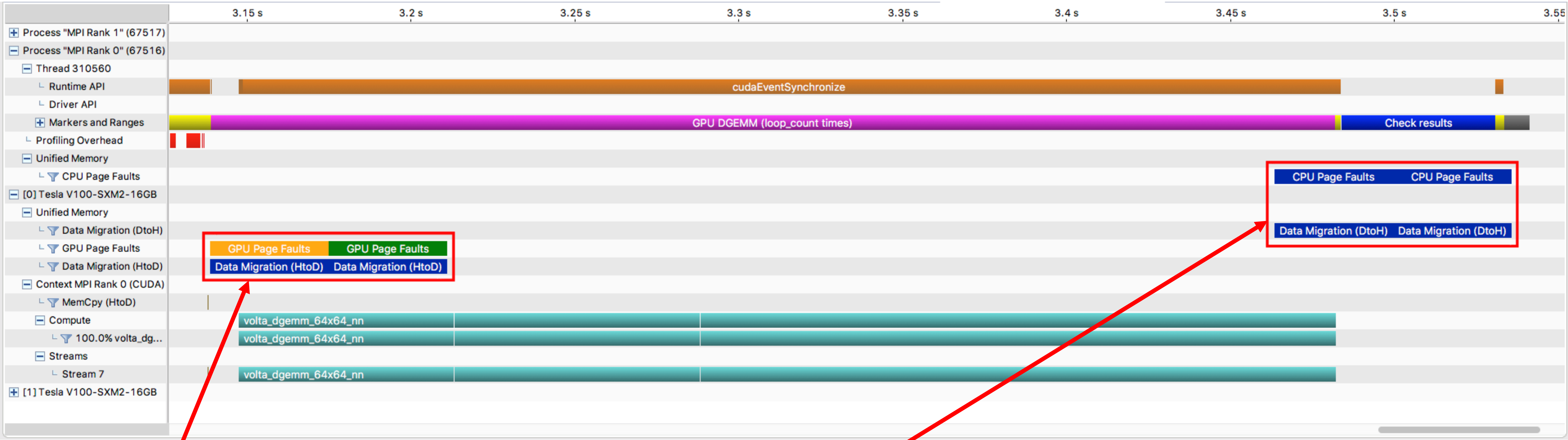
```
/* -------------------------------------------------------------------
    Transfer data from CPU to GPU
--------------------------------------------------------------------- */

// 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

REDUNDANT MATRIX MULTIPLY - VISUAL PROFILER



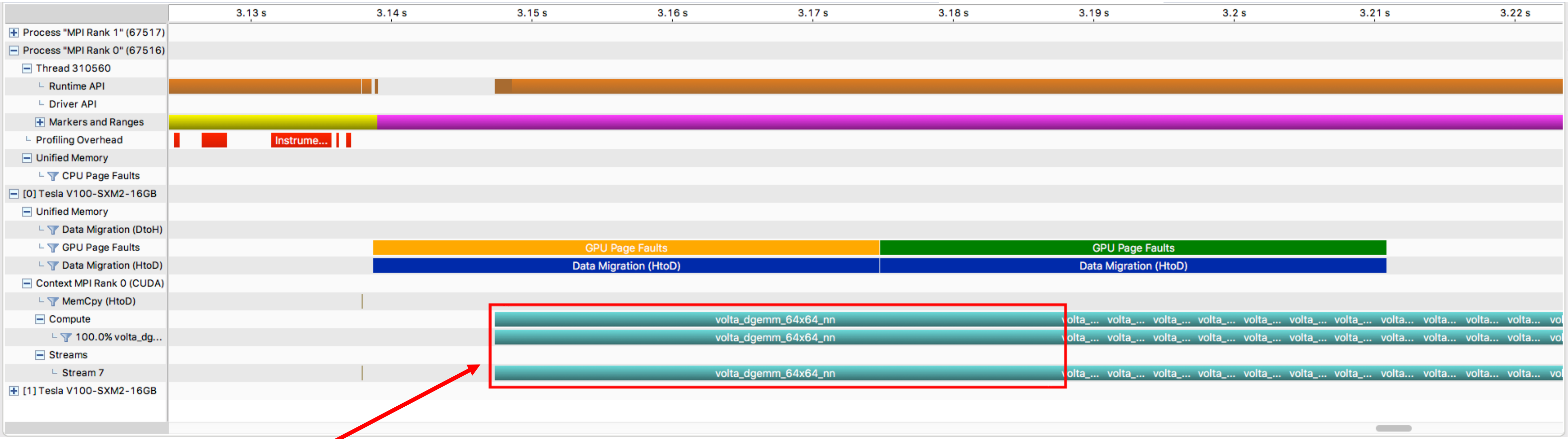
REDUNDANT MATRIX MULTIPLY – VISUAL PROFILER



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

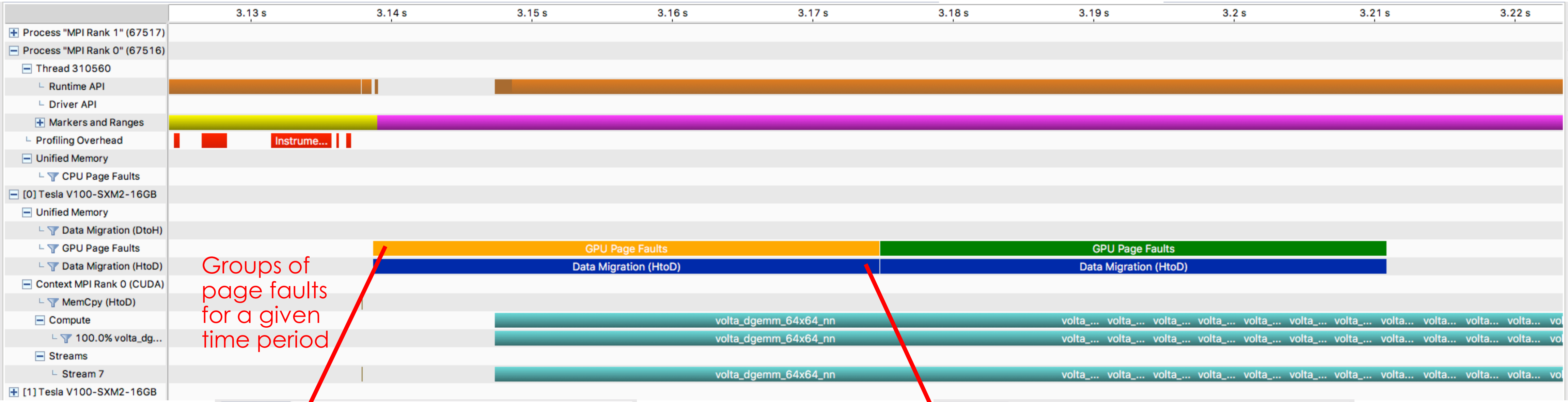
When data is needed on CPU (to compare CPU/GPU results), CPU page faults trigger data migration from GPU to CPU.

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 MATMUL - VISUAL PROFILER



Groups of page faults for a given time period

Properties

GPU Page Fault groups

The segment mode is used for this timeline. In this mode the timeline is split into equal width segments and only aggregated data values for each time segment are shown.

Start	3.139 s (3,138,706,395 ns)
End	3.175 s (3,174,783,480 ns)
Duration	36.077 ms (36,077,085 ns)
Virtual Address Range	0x200080000000 - 0x20...
GPU Page Faults	14022
GPU Page Fault groups	105
Duration of GPU page faults	26.821 ms
Process	67516

The time taken to resolve GPU page faults within the segment

- 0 - 10 % [0 - 3.608 ms]
- 10 - 20 % [3.608 ms - 7.215 ms]
- 20 - 30 % [7.215 ms - 10.823 ms]
- 30 - 40 % [10.823 ms - 14.431 ms]
- 40 - 50 % [14.431 ms - 18.039 ms]
- 50 - 60 % [18.039 ms - 21.646 ms]
- 60 - 70 % [21.646 ms - 25.254 ms]
- 70 - 80 % [25.254 ms - 28.862 ms]
- 80 - 90 % [28.862 ms - 32.469 ms]
- 90 - 100 % [> 32.469 ms]

Properties

Data Migration (HtoD)

The segment mode is used for this timeline. In this mode the timeline is split into equal width segments and only aggregated data values for each time segment are shown.

Start	3.139 s (3,138,706,395 ns)
End	3.175 s (3,174,783,480 ns)
Duration	36.077 ms (36,077,085 ns)
Size	65.012 MB
Throughput	
Min	18.124 GB/s
Max	41.98 GB/s
Virtual Address Range	0x200080000000 - 0x20...
Duration of HtoD data migrations	2.305 ms
Process	67516

The time taken for data migrations from host to device within the segment

- 0 - 10 % [0 - 3.608 ms]
- 10 - 20 % [3.608 ms - 7.215 ms]
- 20 - 30 % [7.215 ms - 10.823 ms]
- 30 - 40 % [10.823 ms - 14.431 ms]
- 40 - 50 % [14.431 ms - 18.039 ms]
- 50 - 60 % [18.039 ms - 21.646 ms]
- 60 - 70 % [21.646 ms - 25.254 ms]
- 70 - 80 % [25.254 ms - 28.862 ms]
- 80 - 90 % [28.862 ms - 32.469 ms]
- 90 - 100 % [> 32.469 ms]

NVIDIA.

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.

SELECTIVE PROFILING

When the profiler API still isn’t enough, selectively profile kernels, particularly with performance counters.

```
$ nvprof --kernels :::1 --analysis-metrics …
```

context:stream:kernel:invocation

Record metrics for only the first invocation of each kernel.

PROFILER API

Real applications frequently produce too much data to manage.

Profiling can be programmatically toggled:

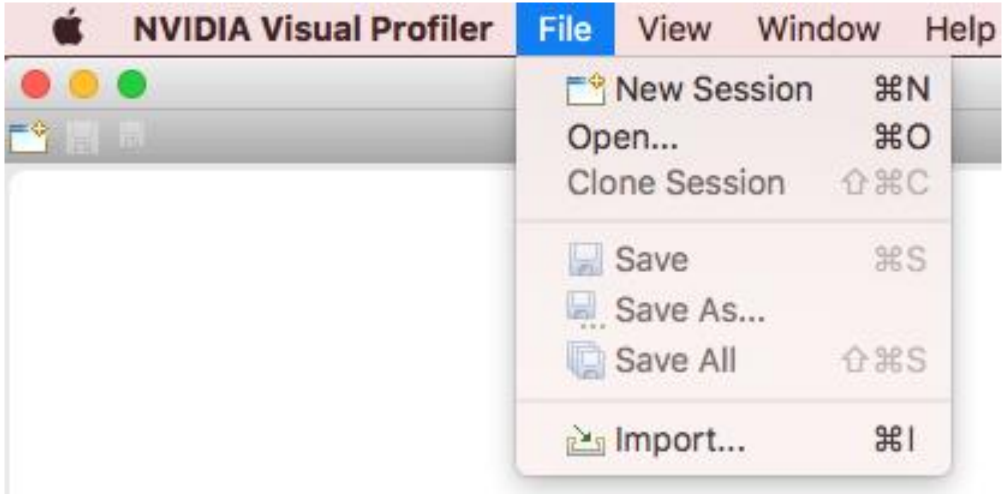
```
#include <cuda_profiler_api.h>

cudaProfilerStart();

…

cudaProfilerStop();
```

This can be paired with nvprof:

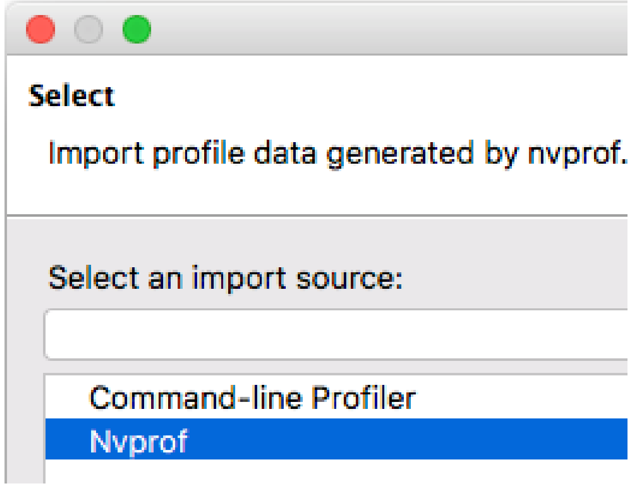
$ nvprof --profile-from-start off …

KERNEL DETAILS - IMPORT INTO VISUAL PROFILER

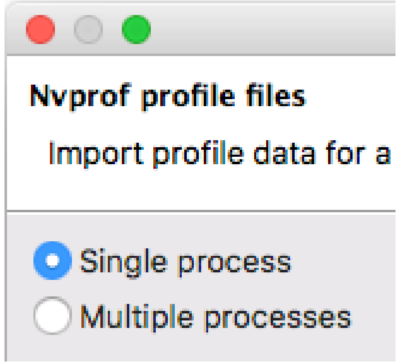
1 File->Import



2 Select "Nvprof" then "Next >"

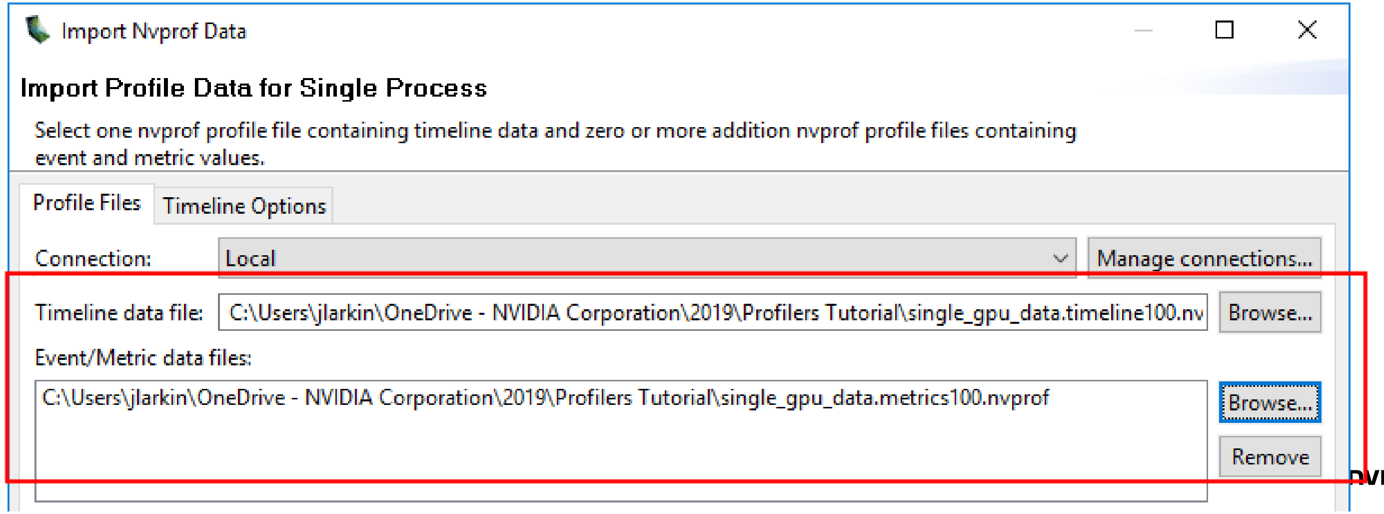


3 Select "Single Process" then "Next >"

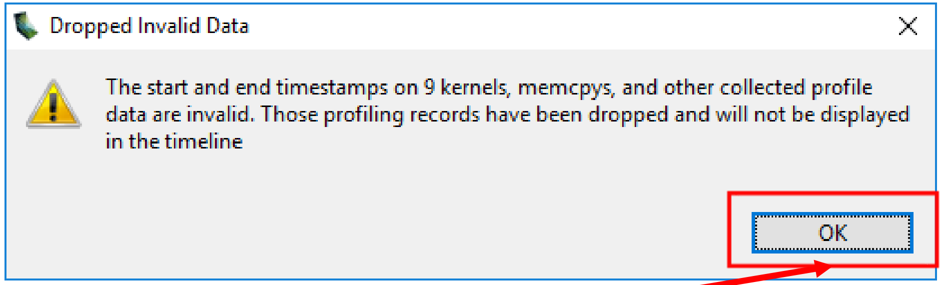


4

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"



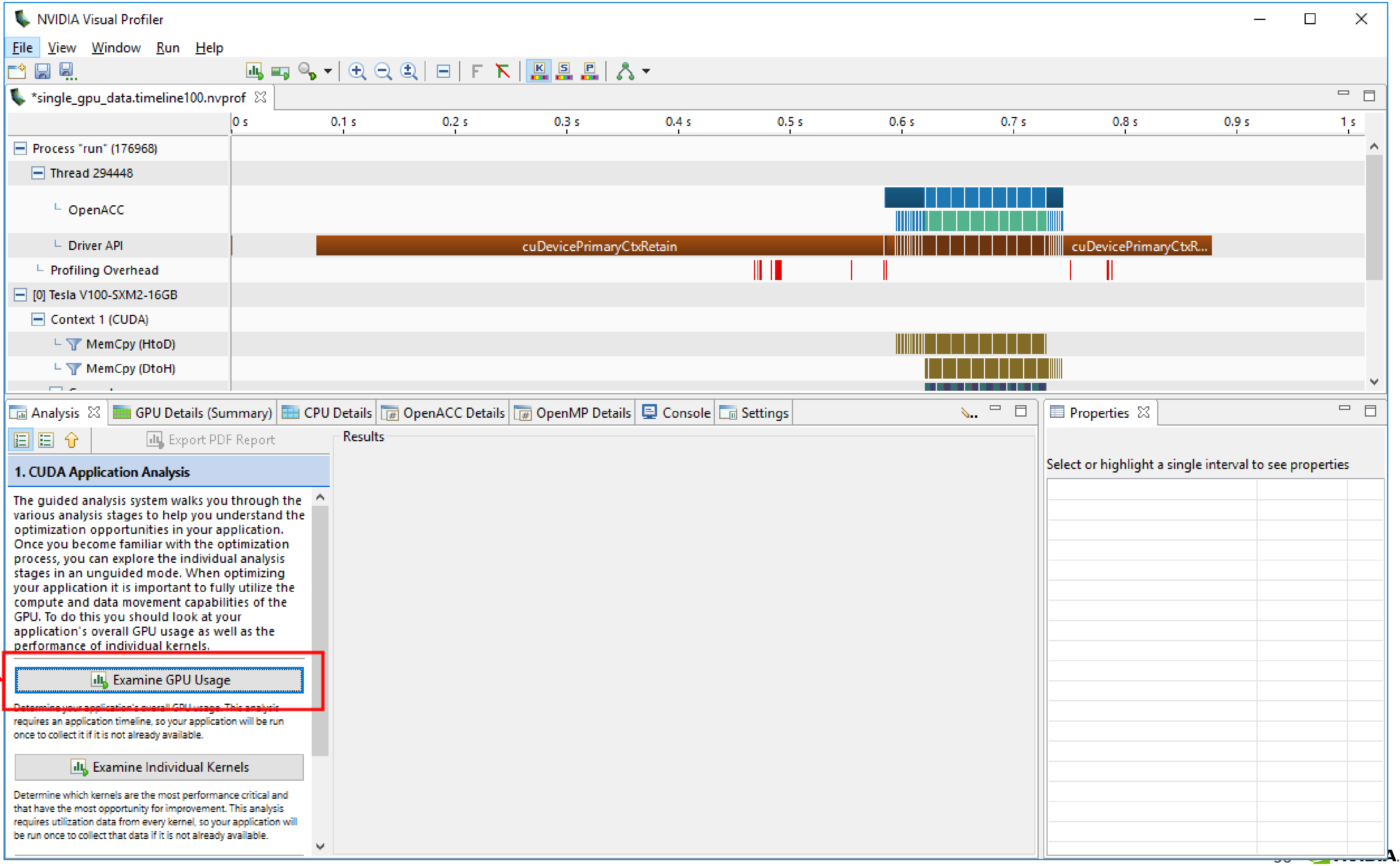
VISUAL PROFILER IMPORT – COMMON WARNING



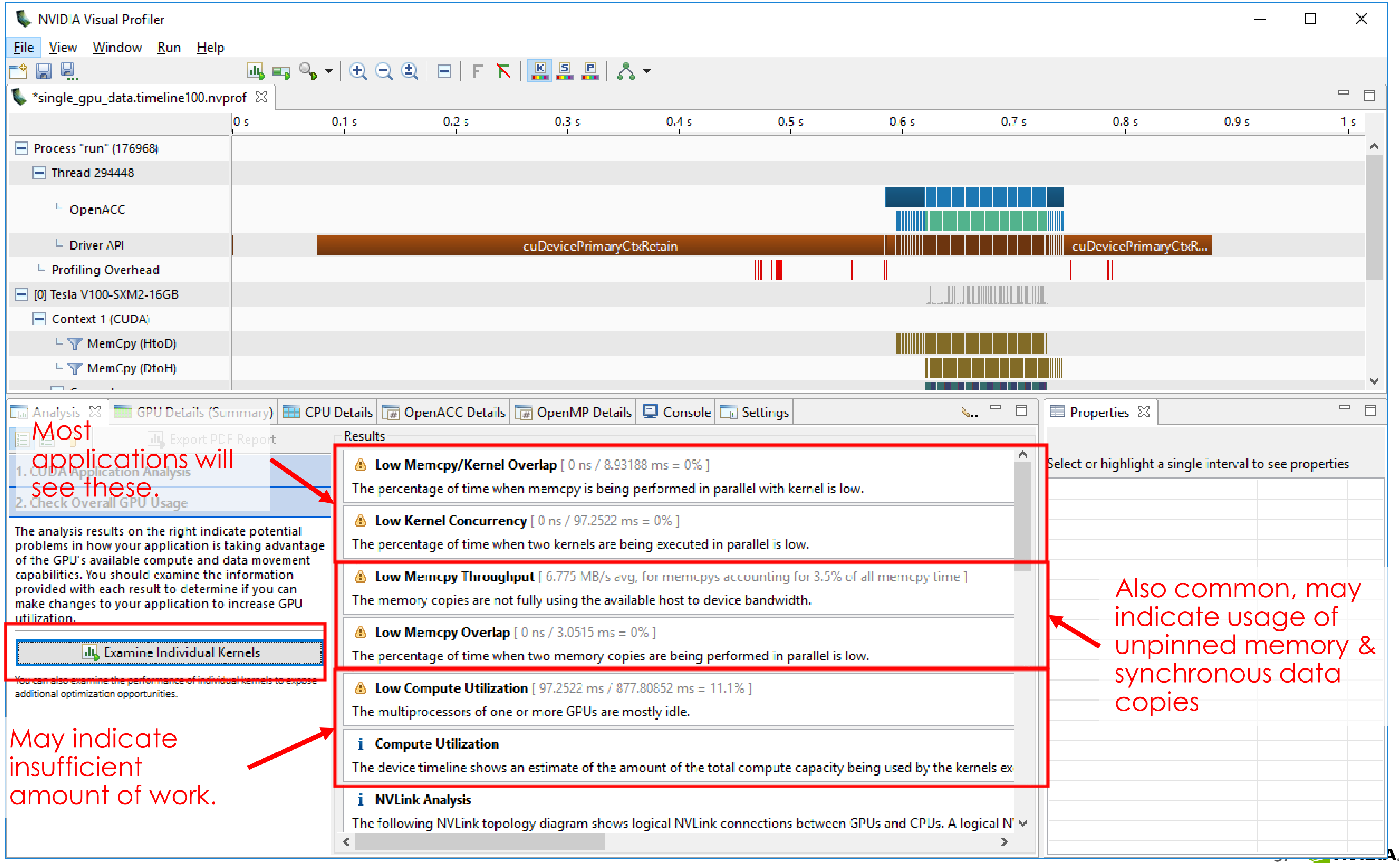
This warning is very common when importing both timelines and metrics, particularly on very short runs. It can be safely ignored.

VISUAL PROFILER – GUIDED ANALYSIS

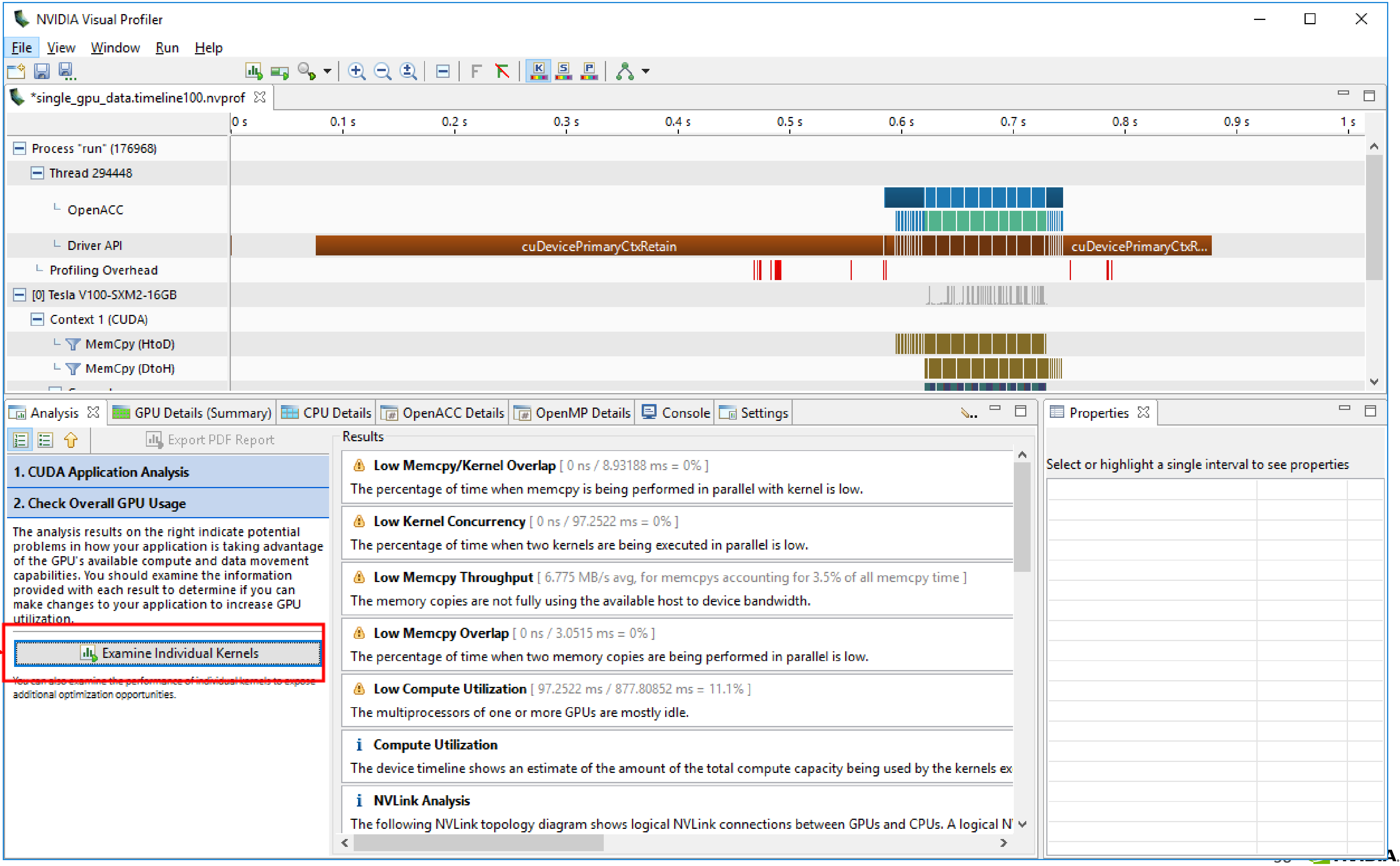
Start with a high-level overview of application performance.



VISUAL PROFILER - GUIDED ANALYSIS

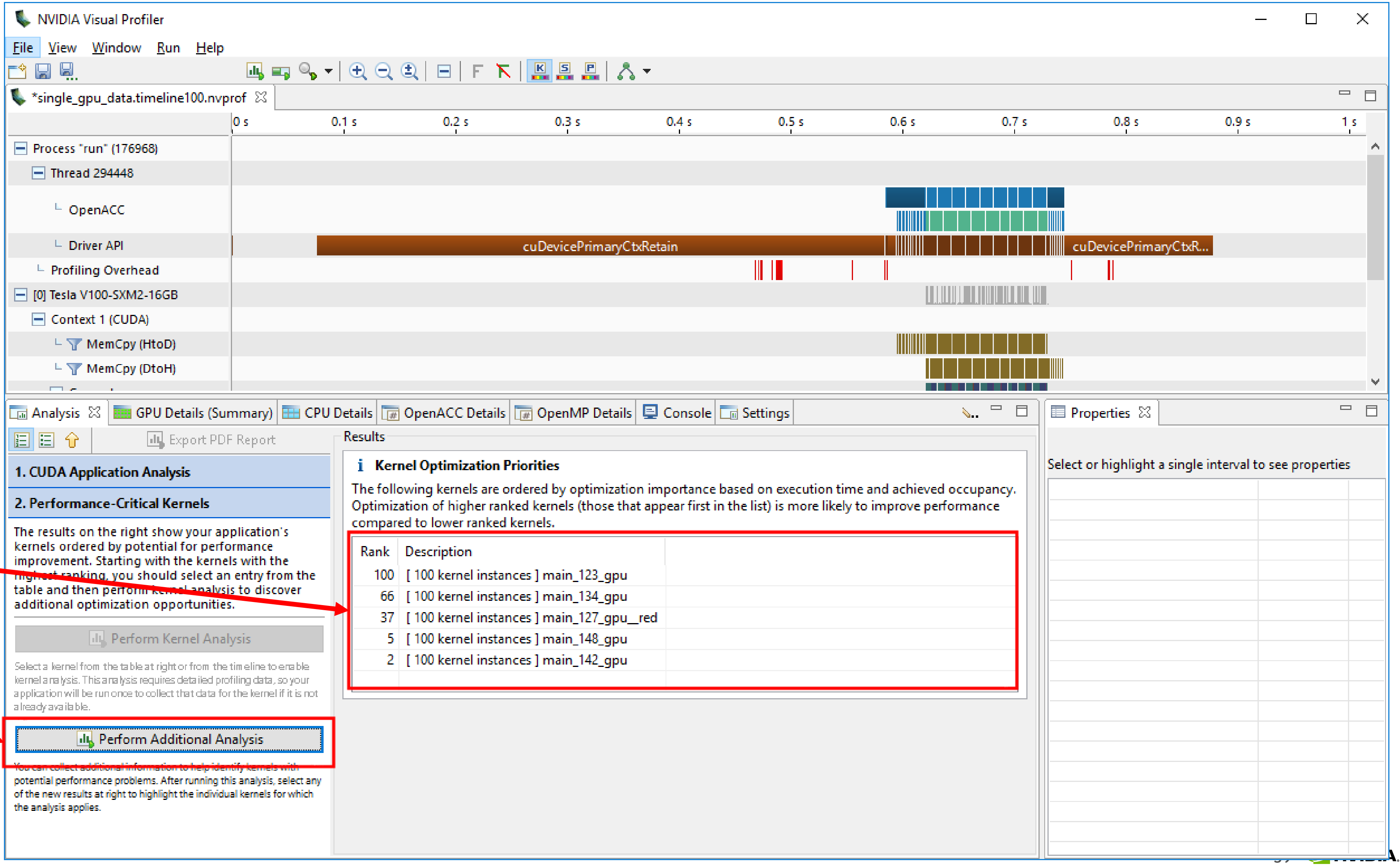


VISUAL PROFILER - GUIDED ANALYSIS



Next zoom in on individual kernel optimizations.

VISUAL PROFILER - GUIDED ANALYSIS



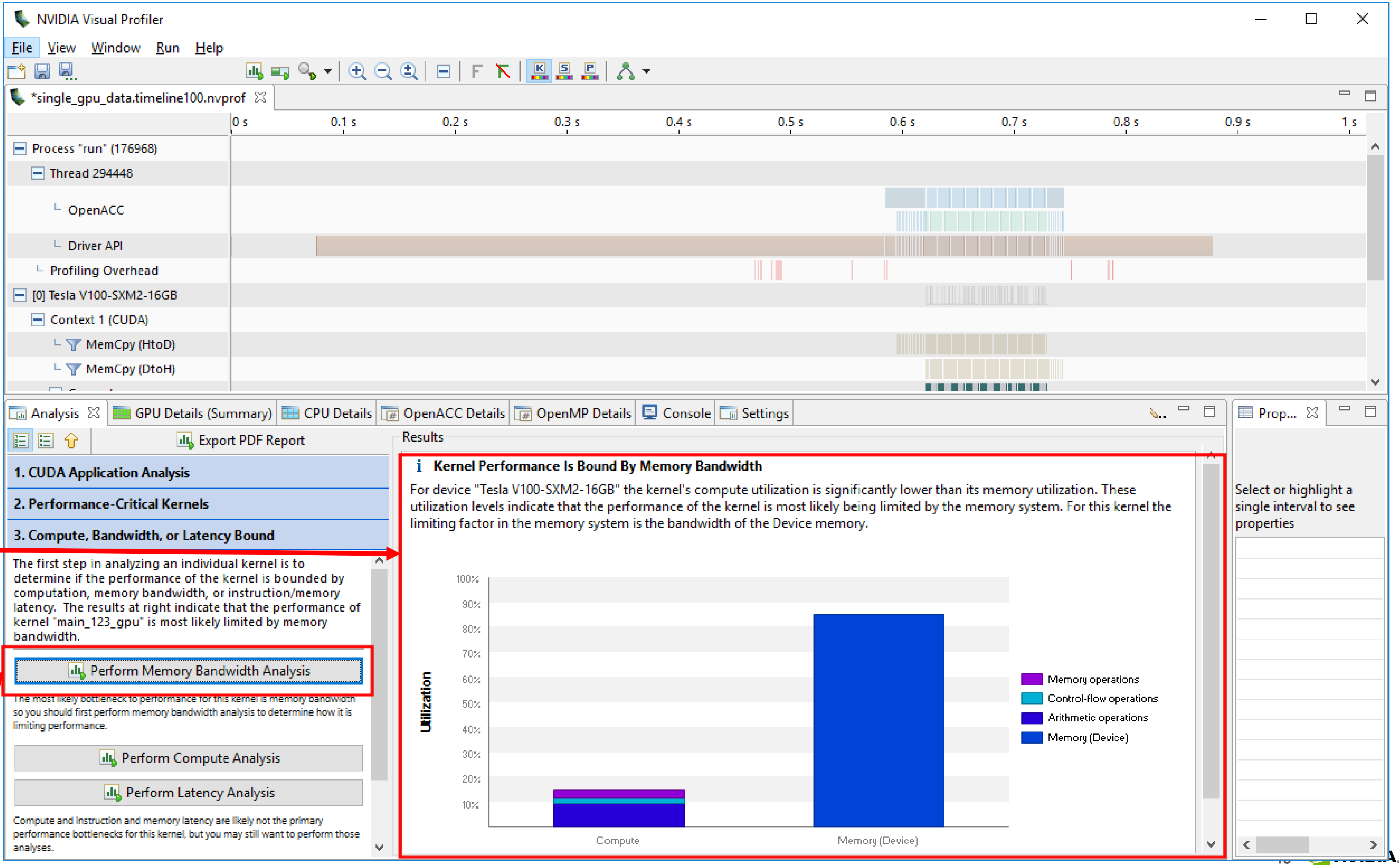
This table ranks the kernels by bang for buck, click the top one.

Click here to deep dive on the selected kernel.

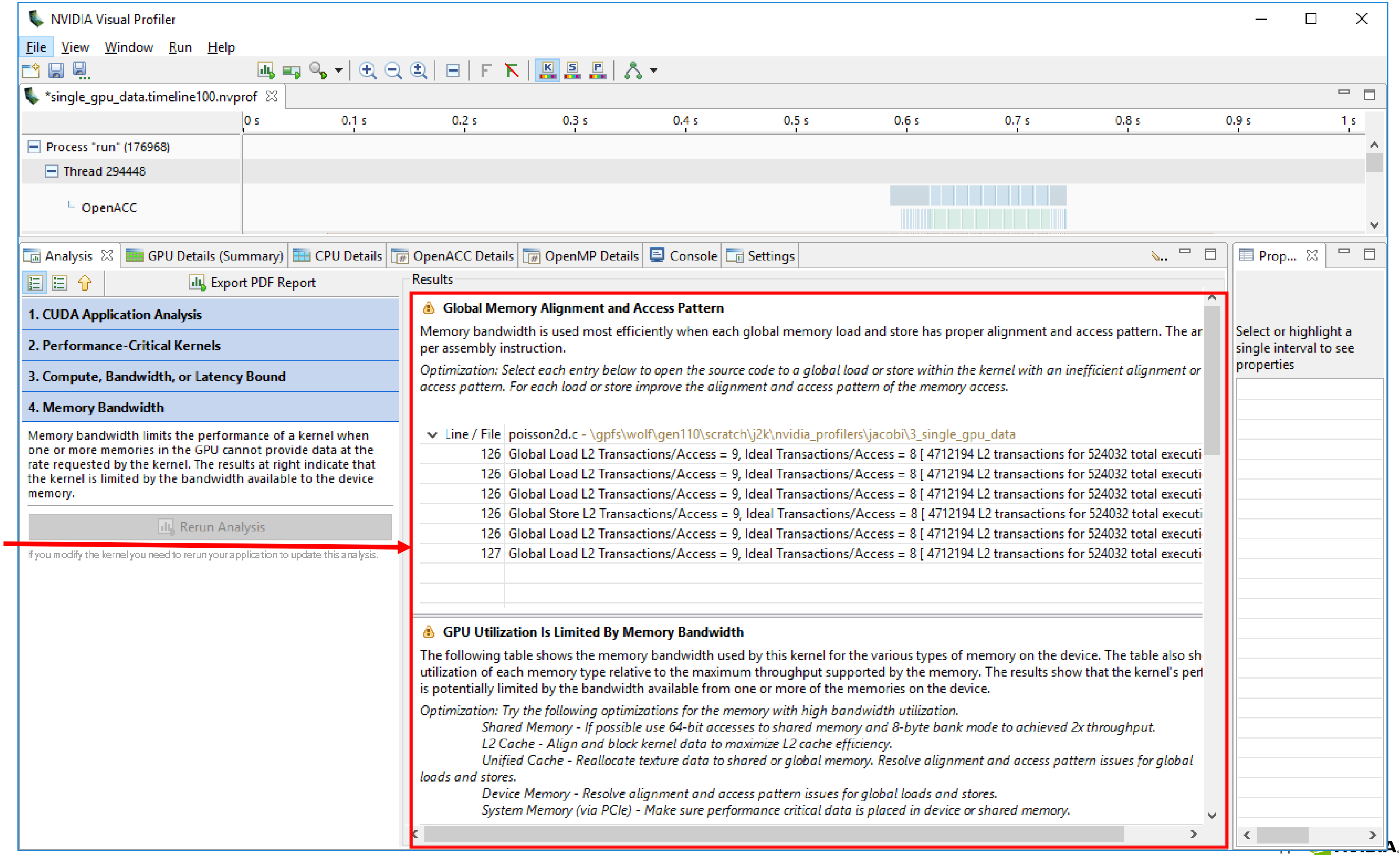
NVVP - GUIDED ANALYSIS - BANDWIDTH BOUND

This box will estimate the performance limiter of your kernel

Click here to dive deeper on that performance limiter



NVVP- GUIDED ANALYSIS - BANDWIDTH BOUND

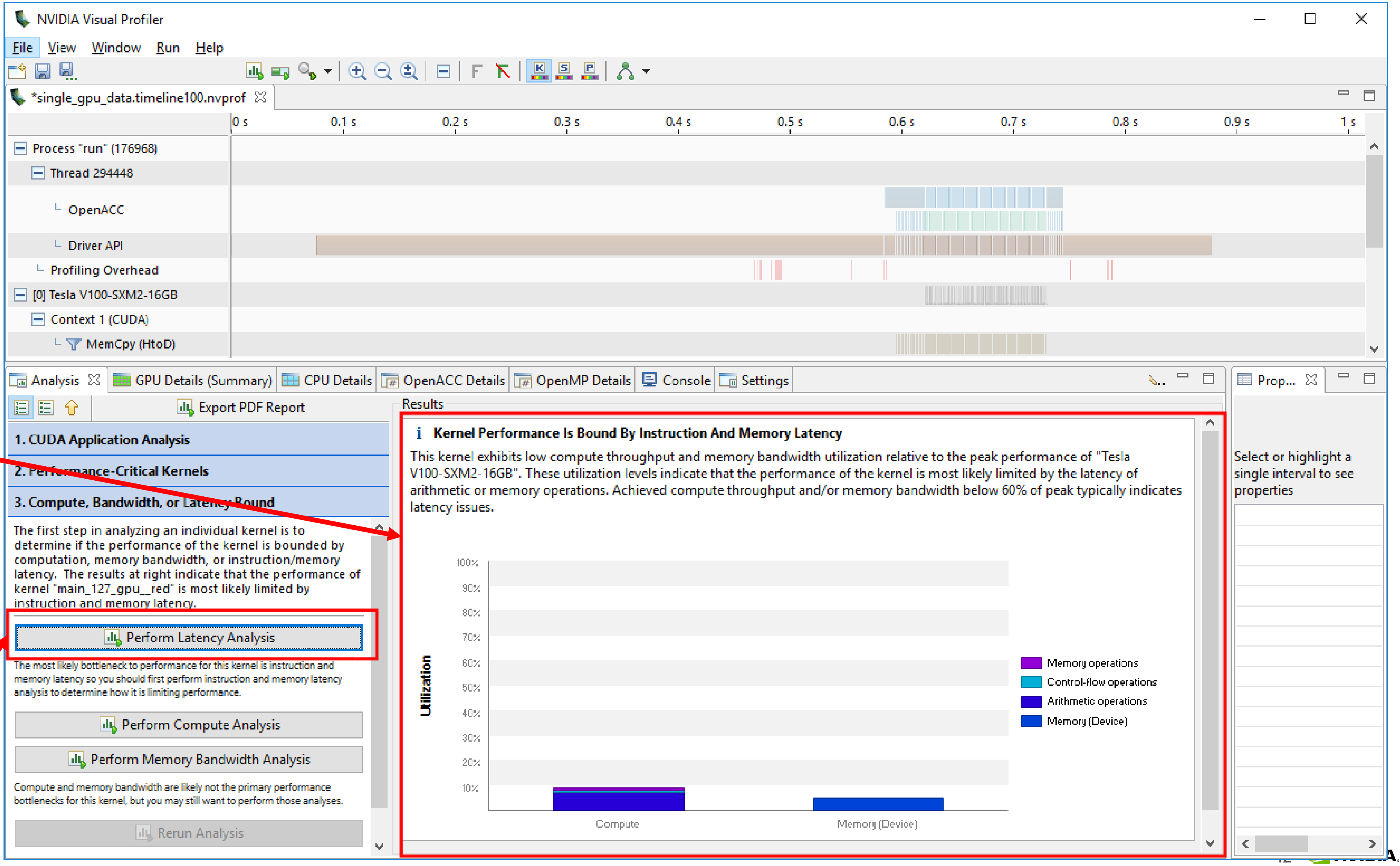


This is the final set of suggestions for this kernel.

NVVP - GUIDED ANALYSIS - LATENCY BOUND

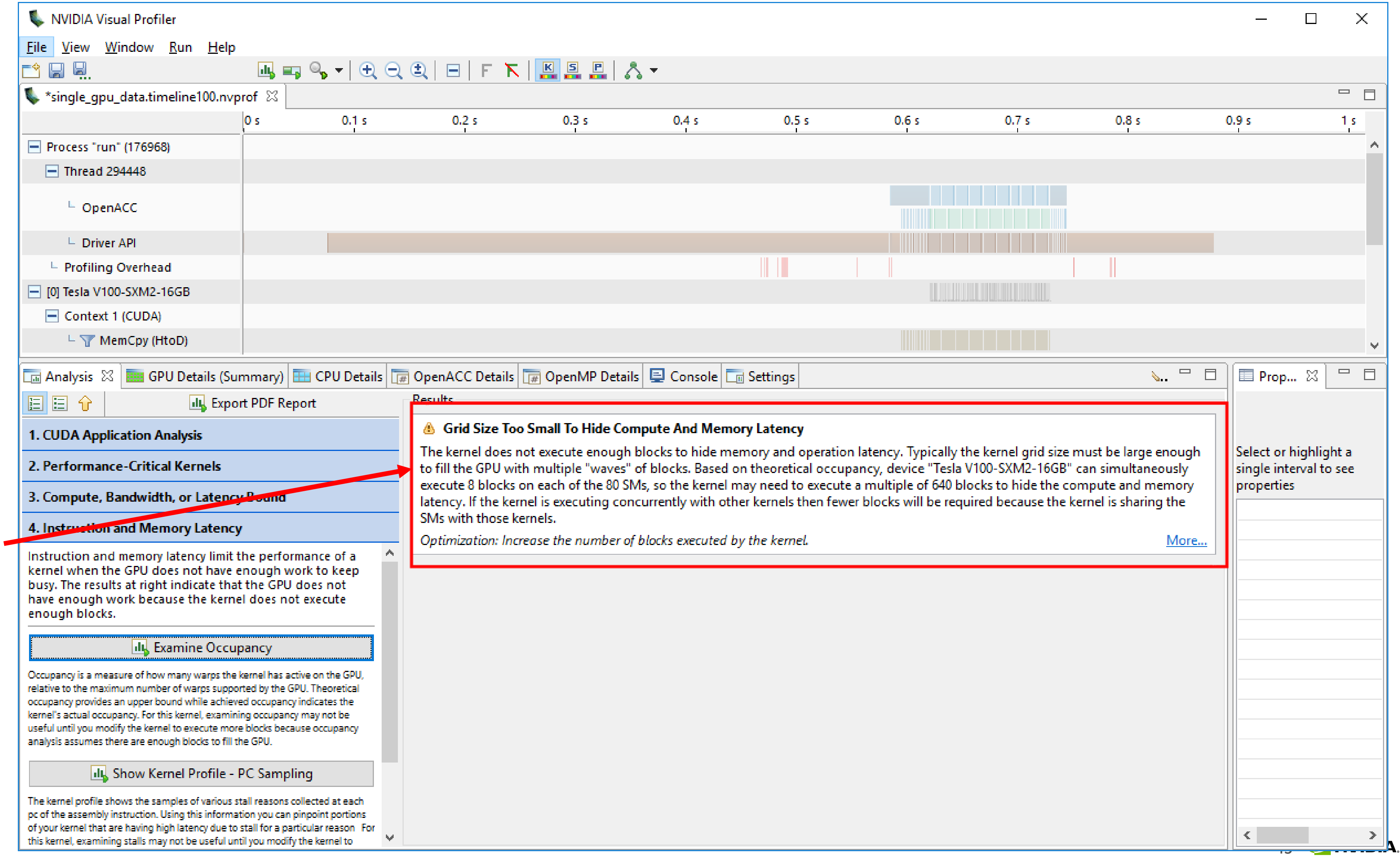
Low Compute & Memory utilization points to being latency bound.

Now a latency analysis is suggested



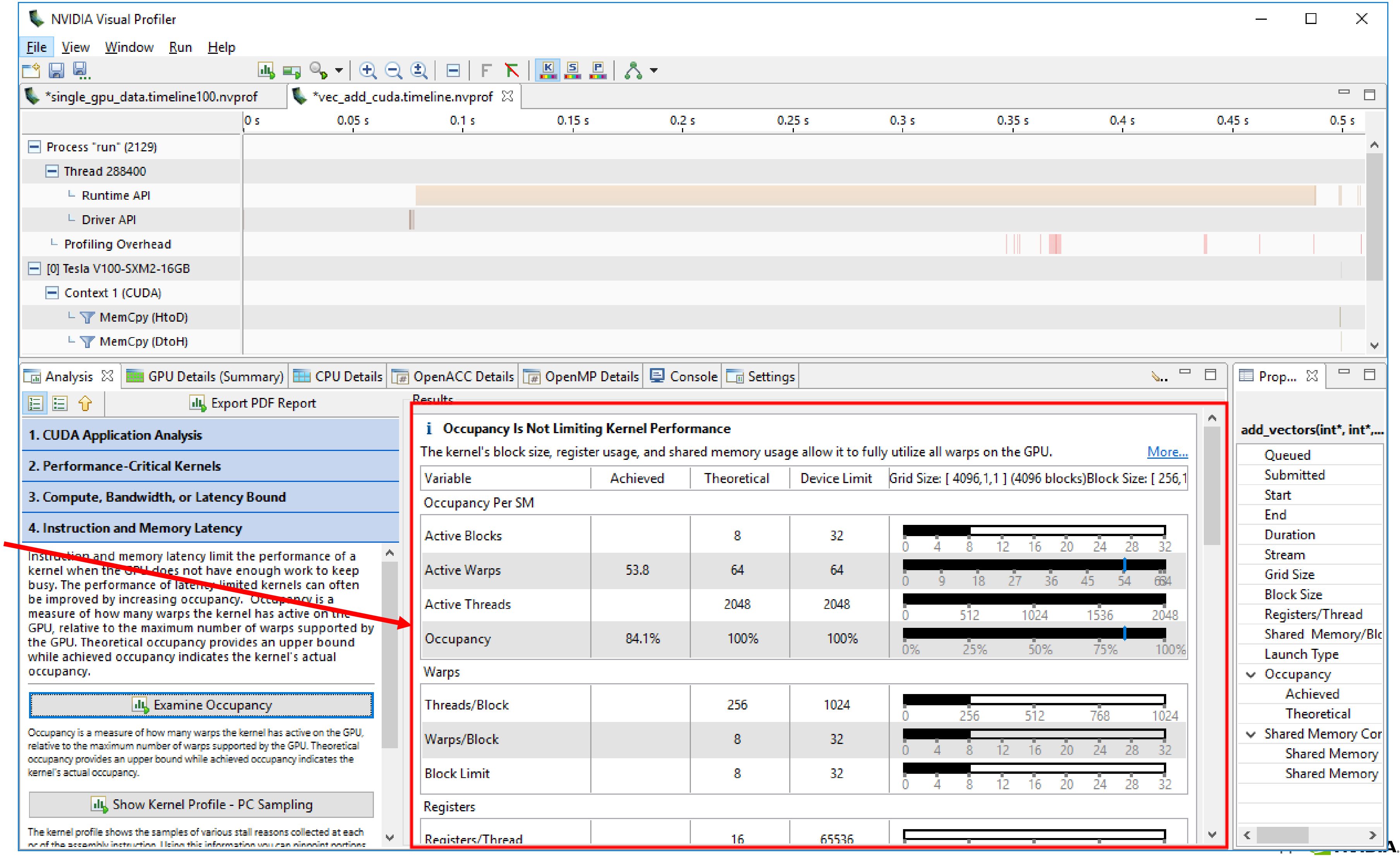
NVVP- GUIDED ANALYSIS - LATENCY BOUND

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



NVVP - GUIDED ANALYSIS - LATENCY BOUND

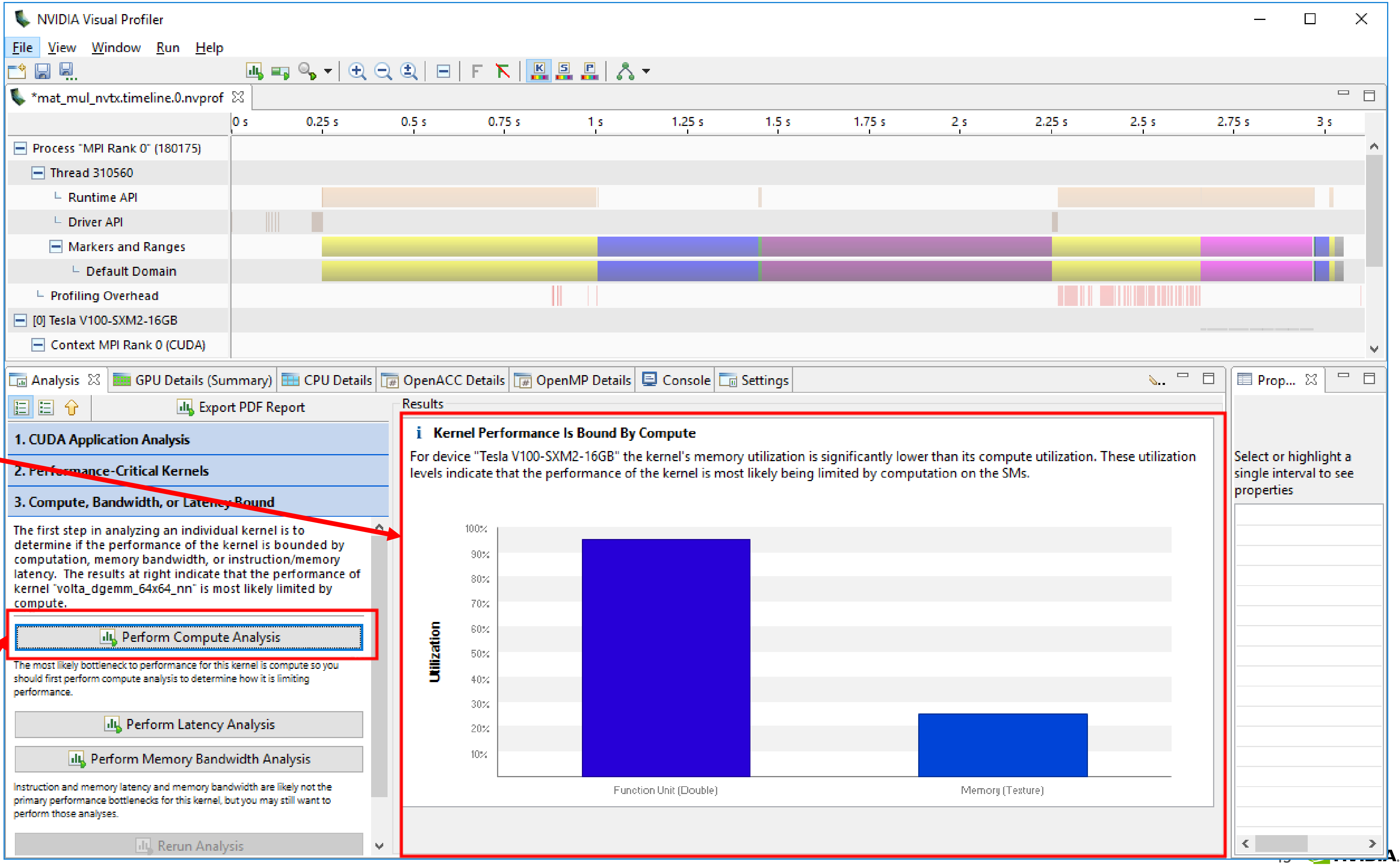
In other cases an occupancy analysis may be performed.



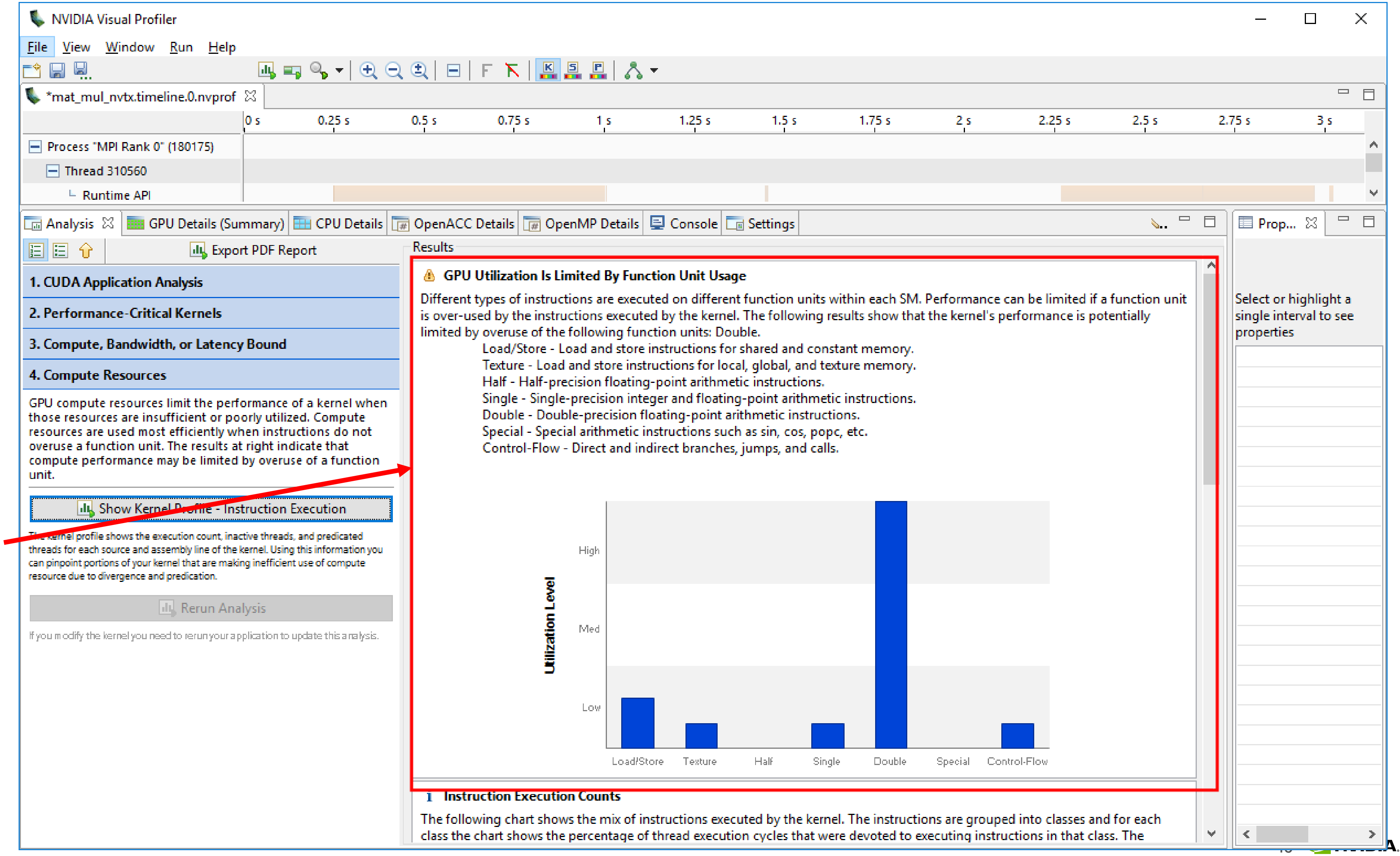
NVVP- GUIDED ANALYSIS – COMPUTE BOUND

High compute utilization indicates the kernel is compute bound.

Now a compute analysis is suggested



NVVP- 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.

CPU SAMPLING

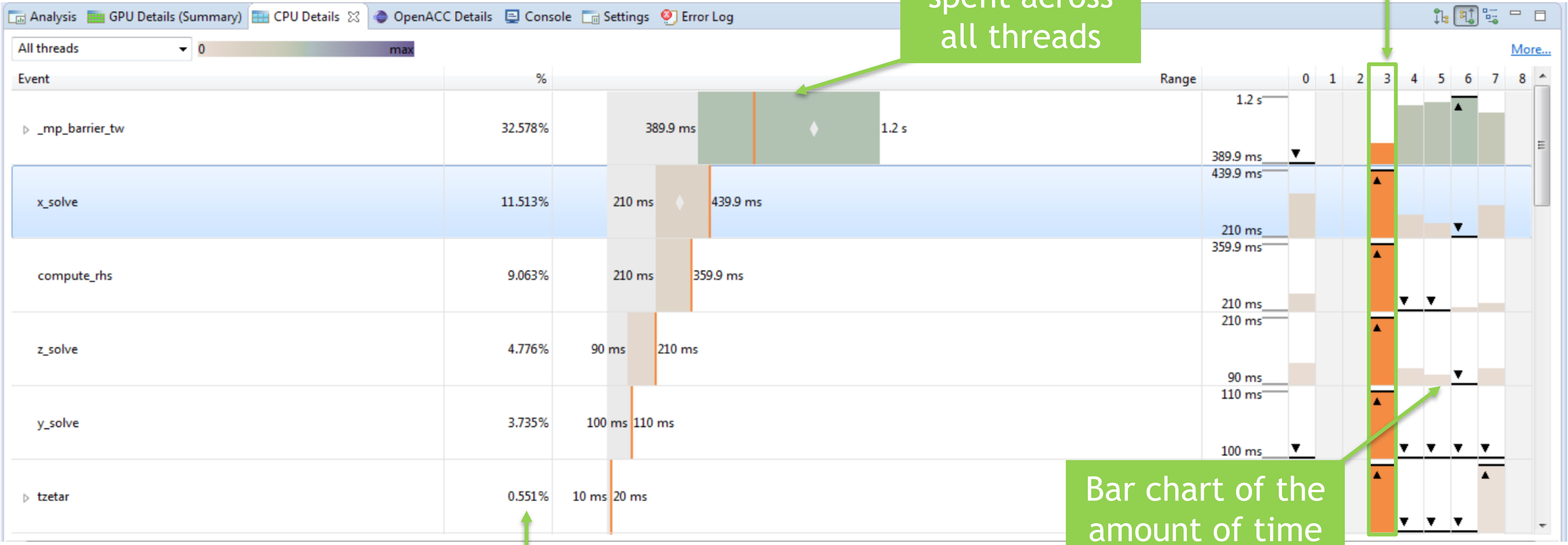
- CPU profile is gathered by periodically sampling the state of each thread in the running application.
- The CPU details view summarizes the samples collected into a call-tree, listing the number of samples (or amount of time) that was recorded in each function.

VISUAL PROFILER

CPU Sampling

Selected thread is highlighted in Orange

Range of time spent across all threads



Bar chart of the amount of time spent by thread

Percentage of time spent collectively by all threads

PC SAMPLING

PC sampling feature is available for device with CC >= 5.2

Provides CPU PC sampling parity + additional information for warp states/stalls reasons for GPU kernels

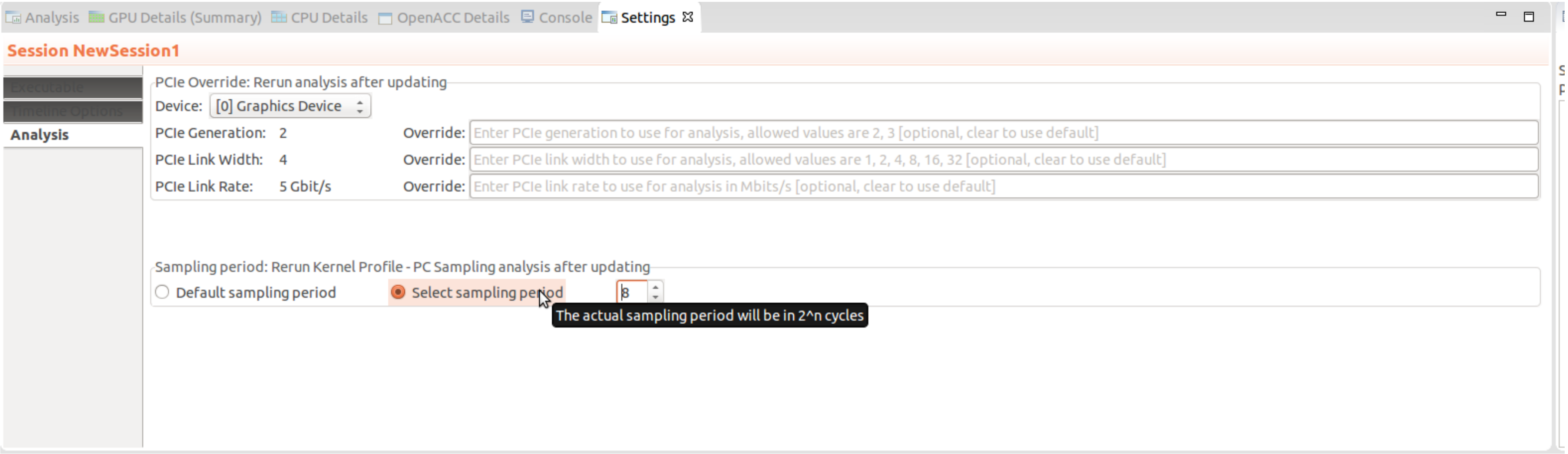
Effective in optimizing large kernels, pinpoints performance bottlenecks at specific lines in source code or assembly instructions

Samples warp states periodically in round robin order over all active warps

No overheads in kernel runtime, CPU overheads to parse the records

VISUAL PROFILER - PC SAMPLING

Option to select sampling period

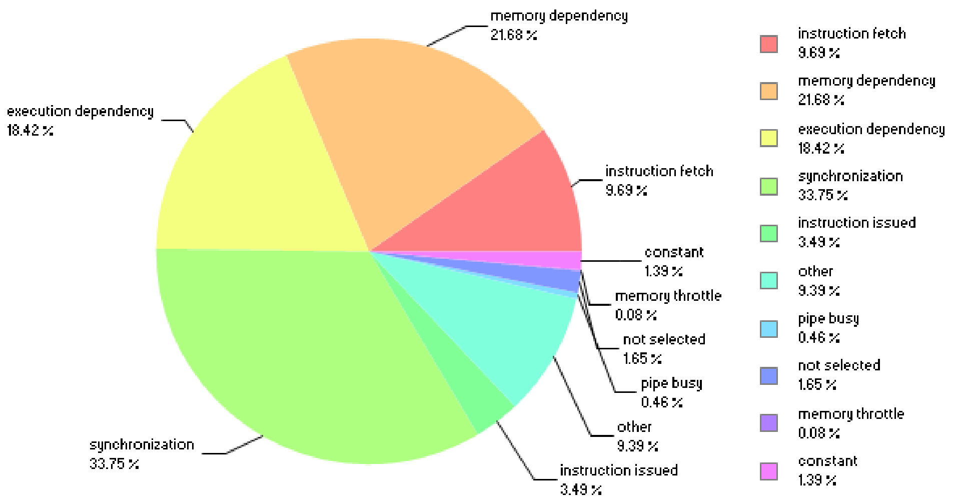


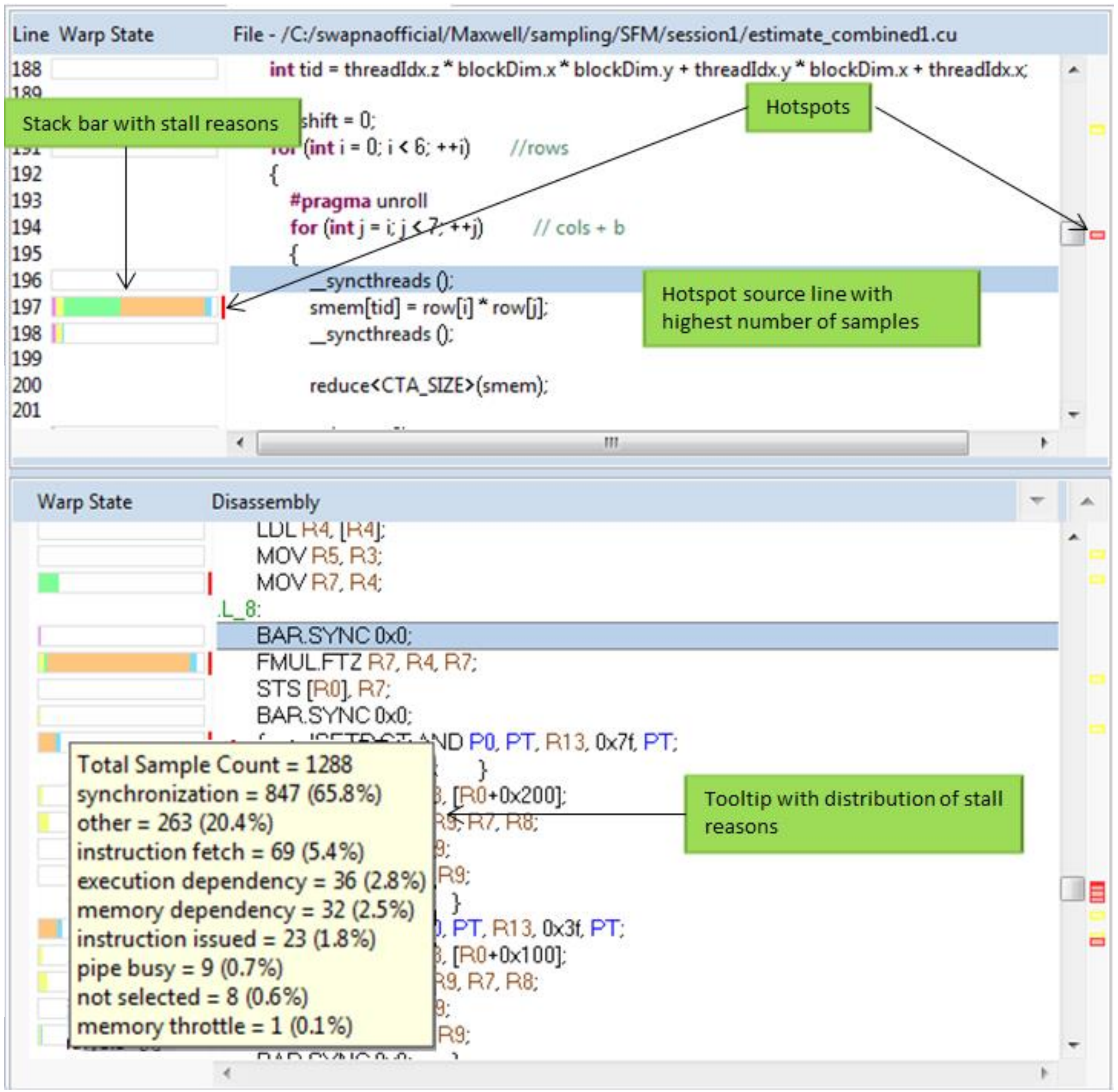
VISUAL PROFILER

PC SAMPLING UI

Pie chart for sample distribution for a CUDA function

Sample distribution





Source-Assembly view

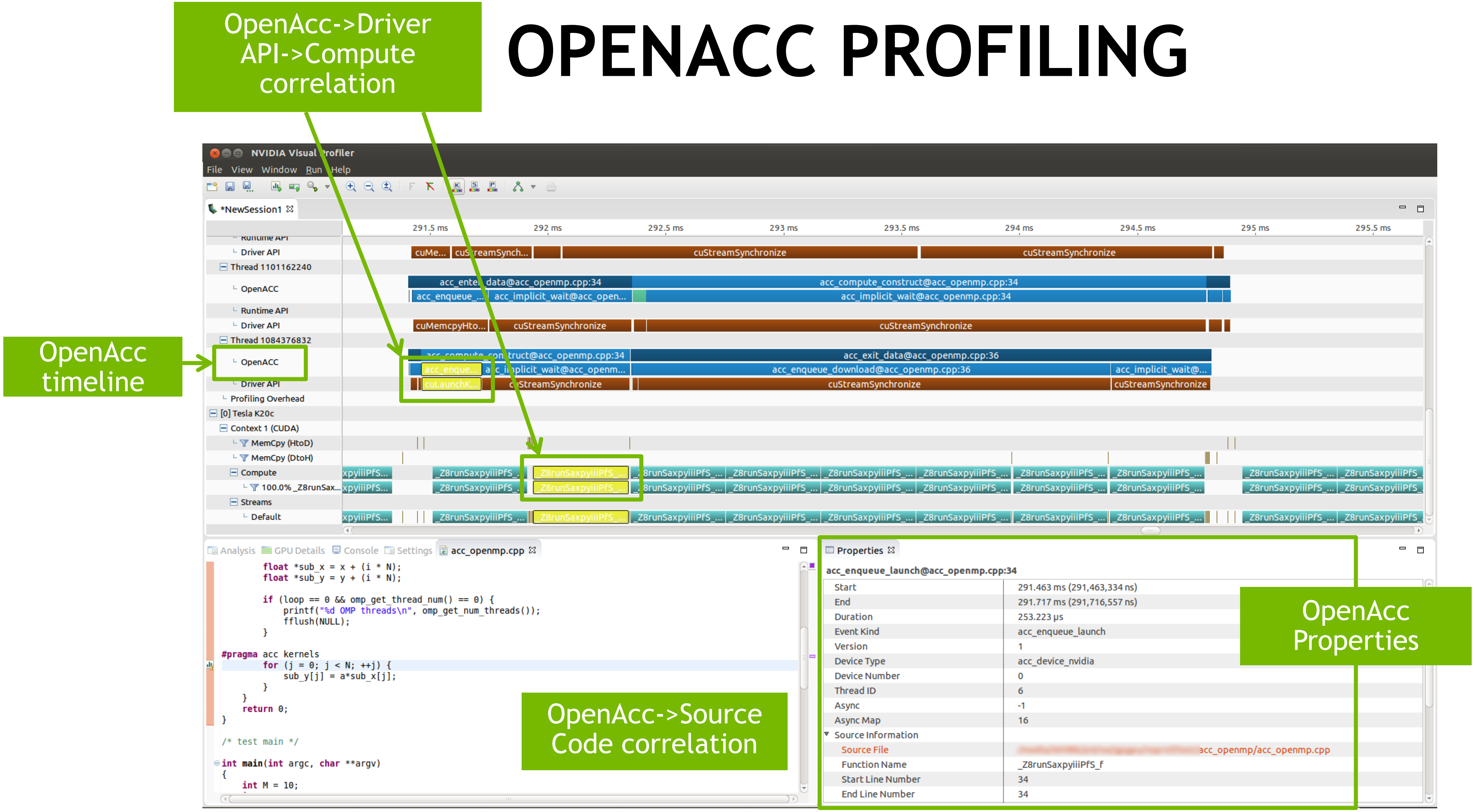
EXPORTING DATA

It’s often useful to post-process nvprof data using your favorite tool (Python, Excel, …):

```
$ nvprof --csv --log-file output.csv \
         -i profile.nvprof
```

It’s often necessary to massage this file before loading into your favorite tool.

OPENACC PROFILING



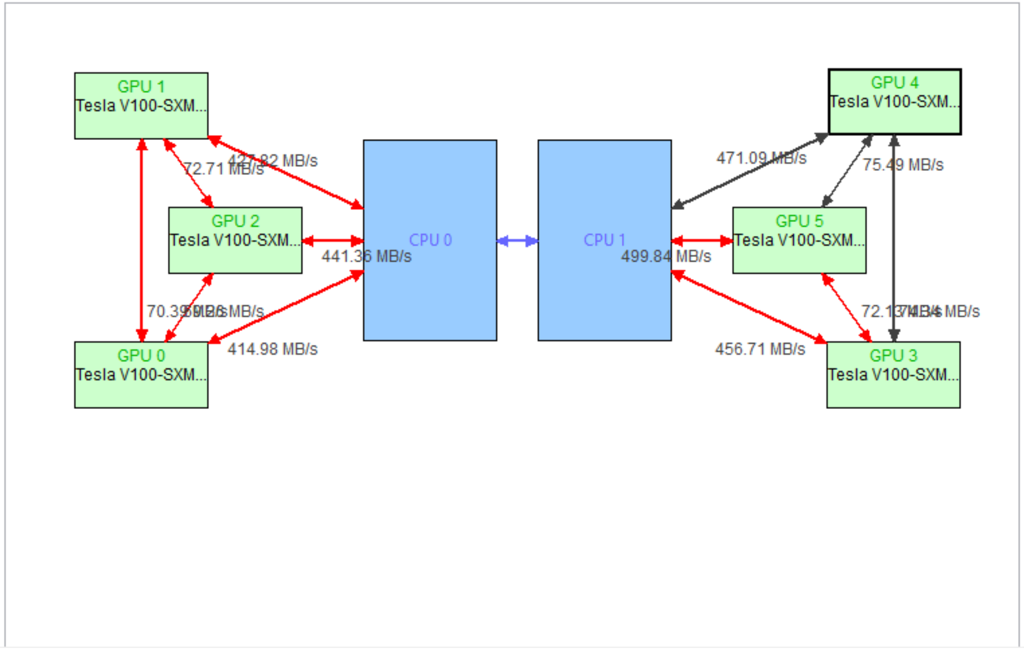
SUMMIT NVLINK TOPOLOGY

Results

NVLink Analysis

The following NVLink topology diagram shows logical NVLink connections between GPUs and CPUs. A logical NVLink can contain one or more physical links. When two devices A and B are connected by an NVLink, the receive throughput of device A is same as the transmit throughput of device B. The tables on right hand side show the properties for each logical NVLink.

* NVLink utilization may vary in accuracy, because any activity within the sampling period is treated as active, even though most of that period could be idle.



Logical NVLink Properties

Logical NVLink	PeakBandwidth	PhysicalNVLinks	PeerAccess	SystemAccess	PeerAtomic	SystemAtomic	Utilization %	Idl
GPU0<-->CP...	100 GB/s	2	No	Yes	No	Yes	0	
GPU0<-->GP...	100 GB/s	2	Yes	No	Yes	No	0	
GPU0<-->GP...	100 GB/s	2	Yes	No	Yes	No	0	
GPU0<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU0<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU0<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU1<-->CP...	100 GB/s	2	No	Yes	No	Yes	0	
GPU1<-->GP...	100 GB/s	2	Yes	No	Yes	No	0	
GPU1<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU1<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU1<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU2<-->CP...	100 GB/s	2	No	Yes	No	Yes	0	
GPU2<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU2<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU2<-->GP...	84 GB/s	2	Yes	No	Yes	No	0	
GPU3<-->CP...	100 GB/s	2	No	Yes	No	Yes	0	
GPU3<-->GP...	100 GB/s	2	Yes	No	Yes	No	0	
GPU3<-->GP...	100 GB/s	2	Yes	No	Yes	No	0	
GPU4<-->CP...	100 GB/s	2	No	Yes	No	Yes	0	
GPU4<-->GP...	100 GB/s	2	Yes	No	Yes	No	0	
GPU5<-->CP...	100 GB/s	2	No	Yes	No	Yes	0	

Logical NVLink Throughput

NSIGHT PRODUCT FAMILY (COMING SOON)

Standalone Performance Tools

Nsight Systems - System-wide application algorithm tuning

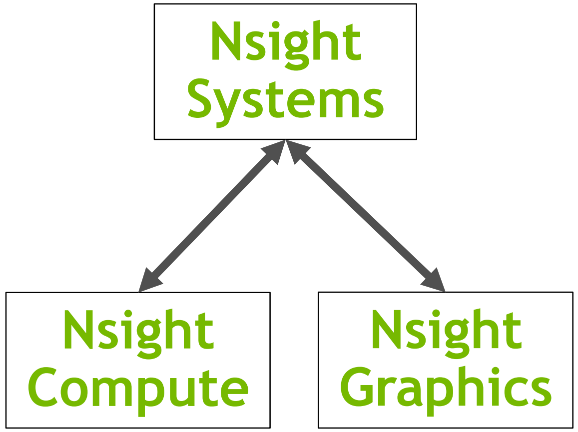
Nsight Compute - Debug/optimize specific CUDA kernel

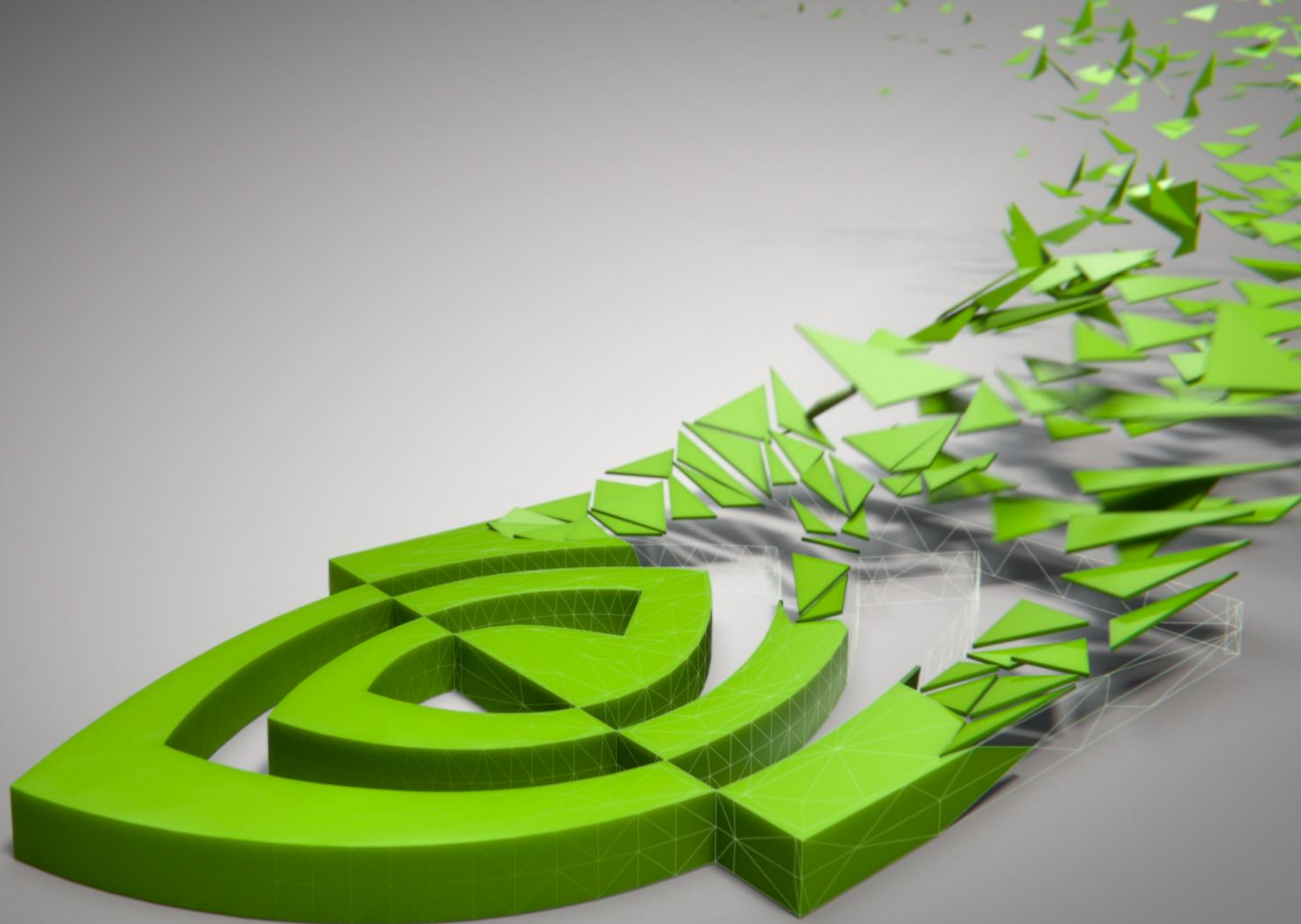
Nsight Graphics - Debug/optimize specific graphics shader

IDE Plugins

Nsight Eclipse Edition/Visual Studio – editor, debugger, some perf analysis

Workflow

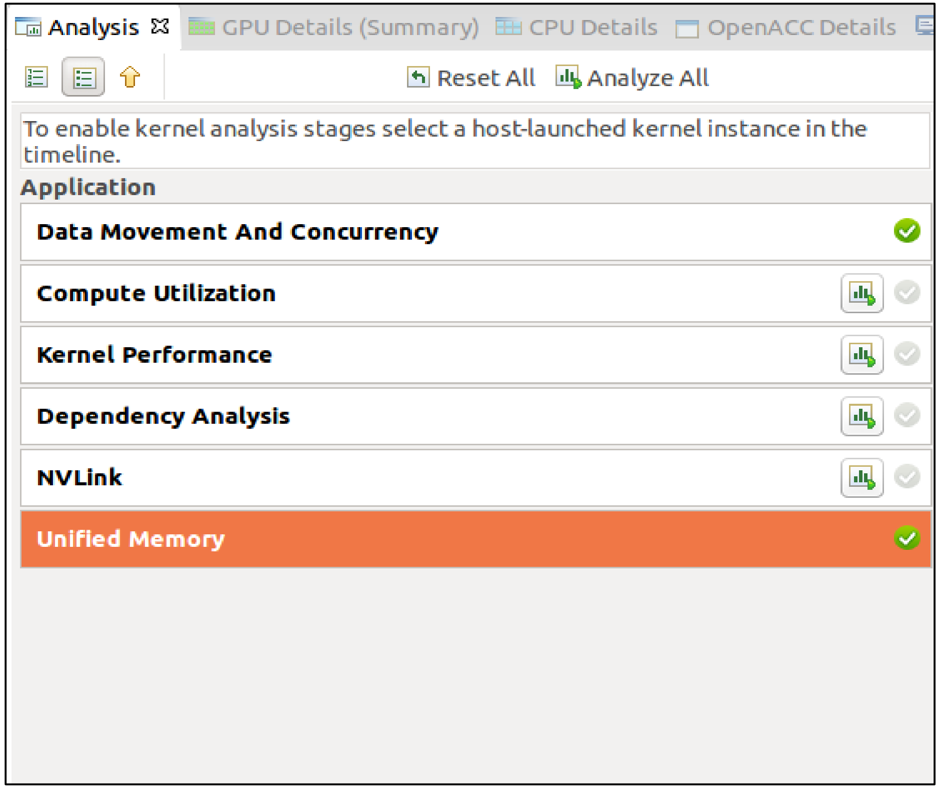




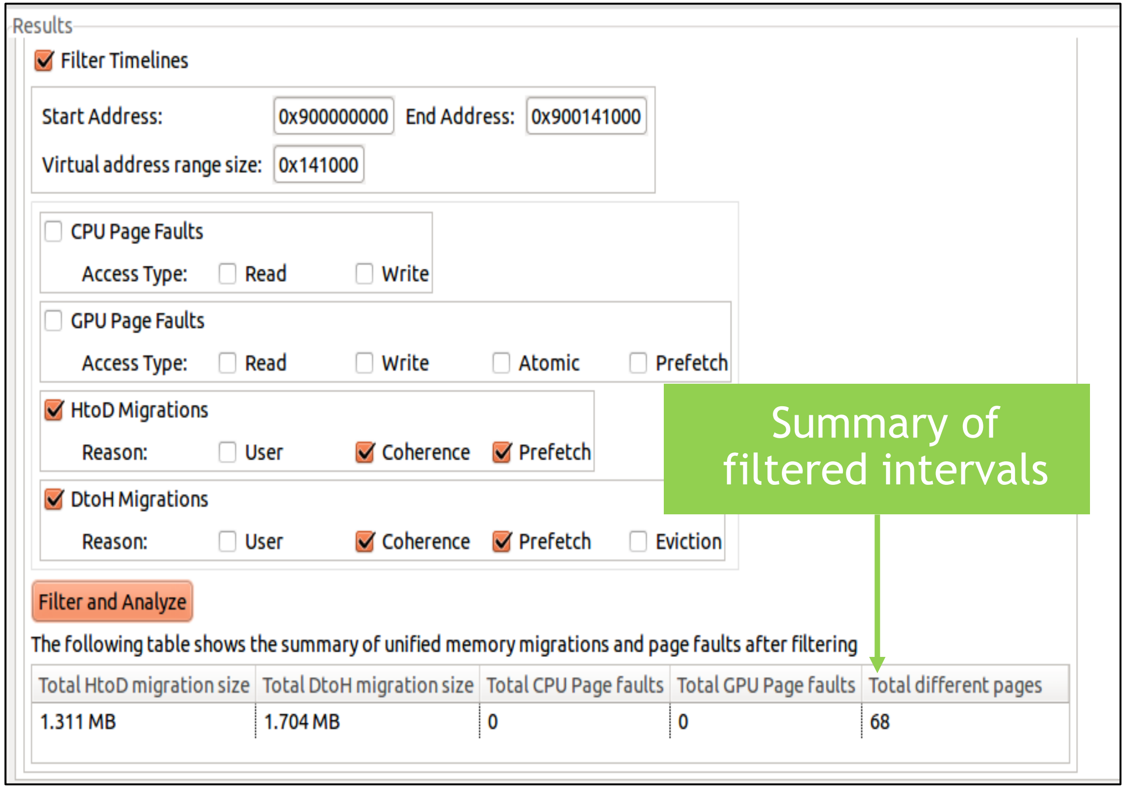
BACKUP SLIDES

FILTER AND ANALYZE

1 Select unified memory in the unguided analysis section

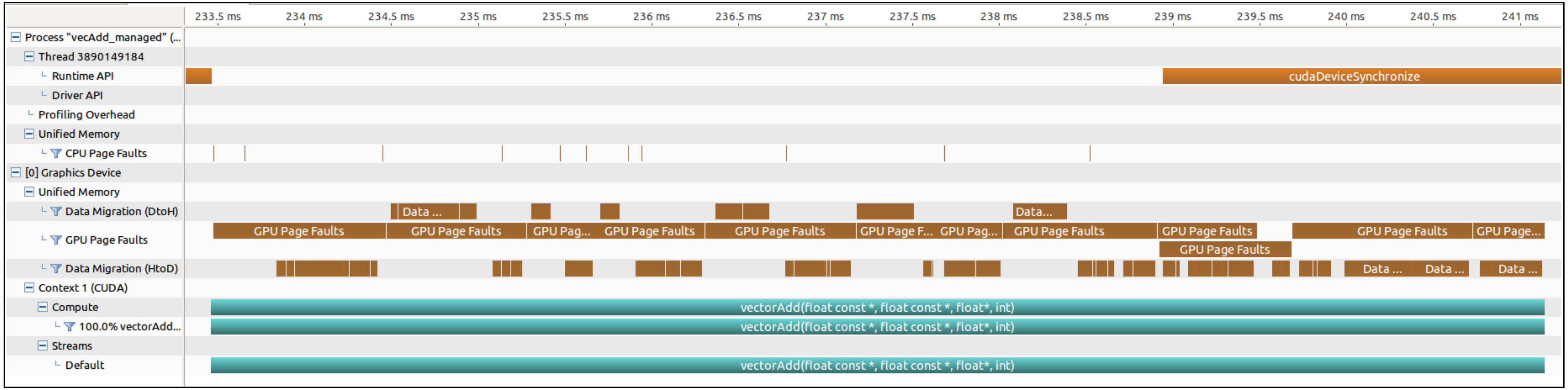


2 Select required events and click on 'Filter and Analyze'



FILTER AND ANALYZE

Unfiltered



EXPANDING THE NSIGHT FAMILY

NSIGHT SYSTEMS

Overview

System-wide application algorithm tuning
- Multi-process tree support

Locate optimization opportunities
- Visualize millions of events on a very fast GUI timeline
- Or gaps of unused CPU and GPU time

Balance your workload across multiple CPUs and GPUs
- CPU algorithms, utilization, and thread state
- GPU streams, kernels, memory transfers, etc

OS: Linux x86_64, Windows, MacOSX (host only)
- No plans for Linux Power

Docs/product: https://developer.nvidia.com/nsight-systems

NSIGHT SYSTEMS

Features

Multicore CPU and multi-GPU system activity trace

CPU utilization, thread state and core/thread migration

OS library trace with blocking call backtraces

pthread, semaphore, file I/O, network I/O, poll/select, sleep/yield, ioctl, syscall, fork

CUDA, OpenACC, OpenGL API and GPU Workload trace
- Includes UVM events
- cuDNN and cuBLAS

NVidia Tools eXtension (NVTX) user annotations

Command line

NSIGHT COMPUTE

NVIDIA NSIGHT COMPUTE

Next-Gen Kernel Profiling Tool

NEW

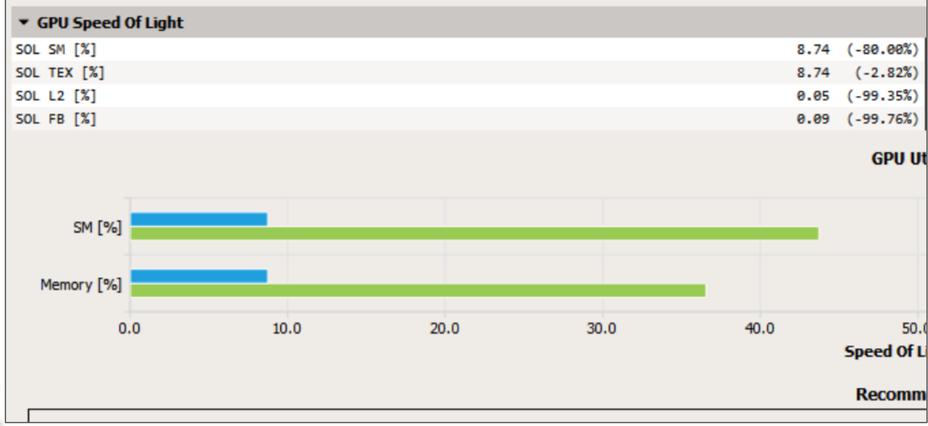
Key Features:

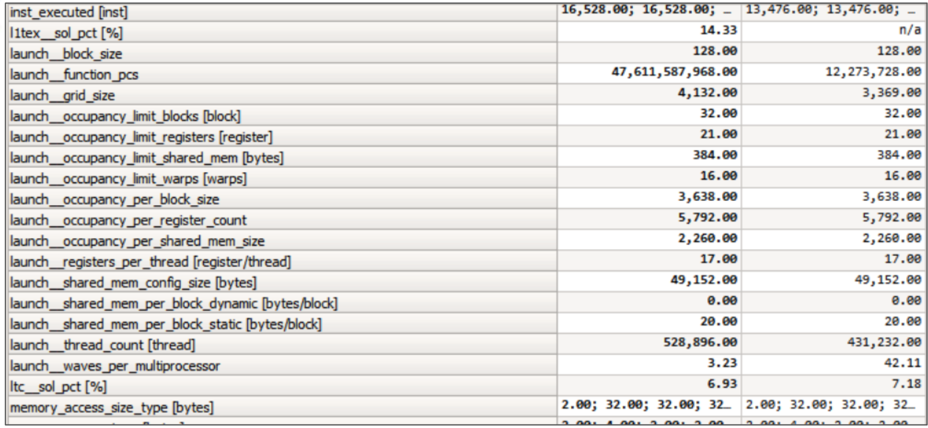
- Interactive CUDA API debugging and kernel profiling
- Fast Data Collection
- Improved Workflow (Diff'ing Results)
- Fully Customizable (Programmable UI/Rules)
- Command Line, Standalone, IDE Integration

OS: Linux x86_64, Windows, MacOSX (host only)
Linux Power planned for Q2 2019

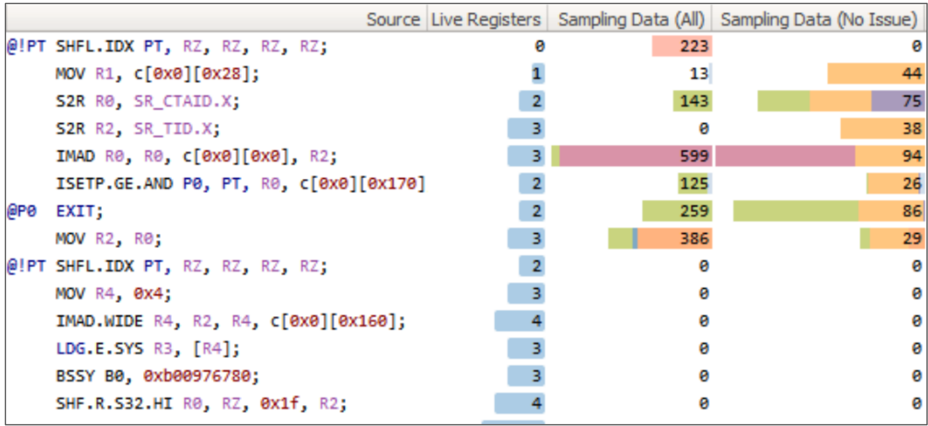
GPUs: Pascal, Volta, Turing

Docs/product: https://developer.nvidia.com/nsight-compute



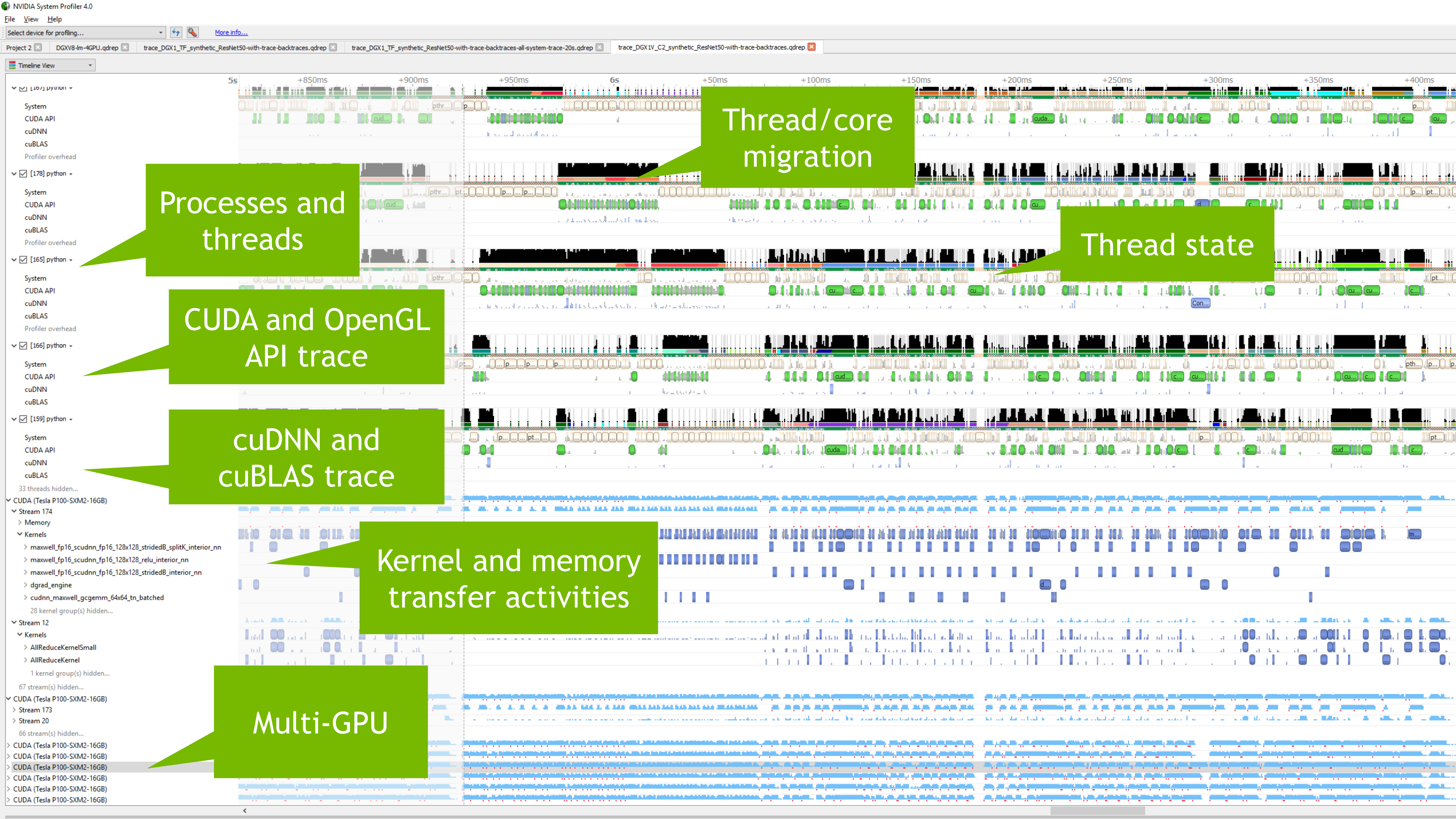


inst_executed [inst]	16,528.00; 16,528.00; …	13,476.00; 13,476.00; …
l1tex__sol_pct [%]	14.33	n/a
launch__block_size	128.00	128.00
launch__function_pcs	47,611,587,968.00	12,273,728.00
launch__grid_size	4,132.00	3,369.00
launch__occupancy_limit_blocks [block]	32.00	32.00
launch__occupancy_limit_registers [register]	21.00	21.00
launch__occupancy_limit_shared_mem [bytes]	384.00	384.00
launch__occupancy_limit_warps [warps]	16.00	16.00
launch__occupancy_per_block_size	3,638.00	3,638.00
launch__occupancy_per_register_count	5,792.00	5,792.00
launch__occupancy_per_shared_mem_size	2,260.00	2,260.00
launch__registers_per_thread [register/thread]	17.00	17.00
launch__shared_mem_config_size [bytes]	49,152.00	49,152.00
launch__shared_mem_per_block_dynamic [bytes/block]	0.00	0.00
launch__shared_mem_per_block_static [bytes/block]	20.00	20.00
launch__thread_count [thread]	528,896.00	431,232.00
launch__waves_per_multiprocessor	3.23	42.11
ltc__sol_pct [%]	6.93	7.18
memory_access_size_type [bytes]	2.00; 32.00; 32.00; 32…	2.00; 32.00; 32.00; 32…



Source	Live Registers	Sampling Data (All)	Sampling Data (No Issue)
@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;	0	223	0
MOV R1, c[0x0][0x28];	1	13	44
S2R R0, SR_CTAID.X;	2	143	75
S2R R2, SR_TID.X;	3	0	38
IMAD R0, R0, c[0x0][0x0], R2;	3	599	94
ISETP.GE.AND P0, PT, R0, c[0x0][0x170]	2	125	26
@P0 EXIT;	2	259	86
MOV R2, R0;	3	386	29
@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;	2	0	0
MOV R4, 0x4;	3	0	0
IMAD.WIDE R4, R2, R4, c[0x0][0x160];	4	0	0
LDG.E.SYS R3, [R4];	3	0	0
BSSY B0, 0xb00976780;	3	0	0
SHF.R.S32.HI R0, RZ, 0x1f, R2;	4	0	0

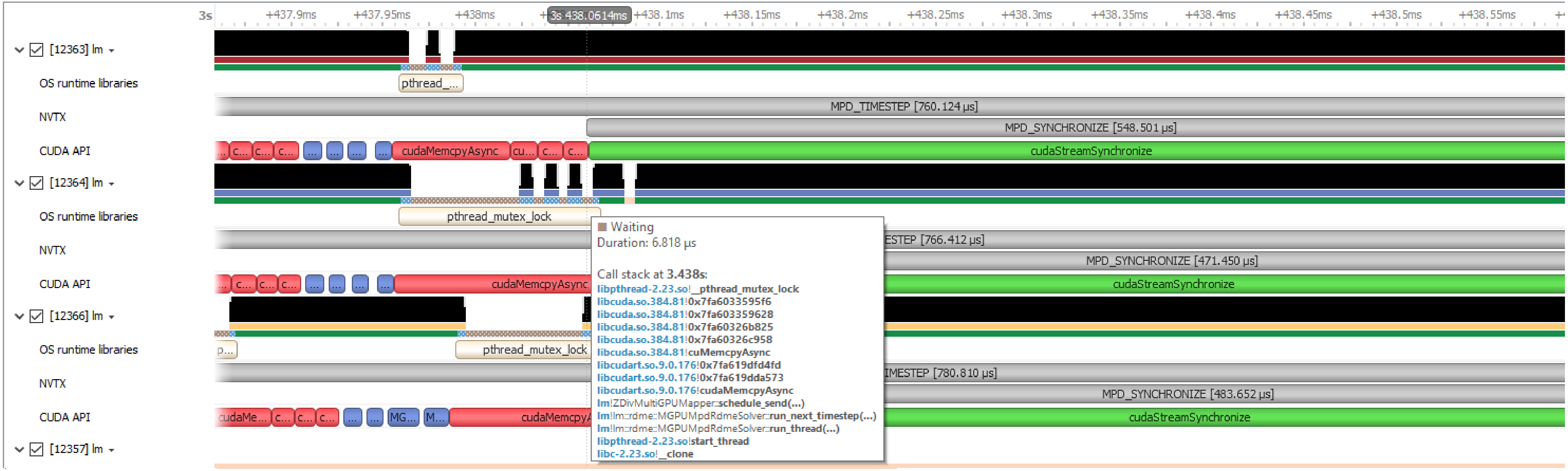
NSIGHT SYSTEMS



NVIDIA System Profiler 4.0
File
View
Help
Select device for profiling...
More info...
Project 2
DGXV8-lm-4GPU.qdrep
trace_DGX1_TF_synthetic_ResNet50-with-trace-backtraces.qdrep
trace_DGX1_TF_synthetic_ResNet50-with-trace-backtraces-all-system-trace-20s.qdrep
trace_DGX1V_C2_synthetic_ResNet50-with-trace-backtraces.qdrep
Timeline View
5s
+850ms
+900ms
+950ms
6s
+50ms
+100ms
+150ms
+200ms
+250ms
+300ms
+350ms
+400ms
[178] python
[165] python
[166] python
[159] python
System
CUDA API
cuDNN
cuBLAS
Profiler overhead
33 threads hidden...
CUDA (Tesla P100-SXM2-16GB)
Stream 174
Memory
Kernels
maxwell_fp16_scudnn_fp16_128x128_stridedB_splitK_interior_nn
maxwell_fp16_scudnn_fp16_128x128_relu_interior_nn
maxwell_fp16_scudnn_fp16_128x128_stridedB_interior_nn
dgrad_engine
cudnn_maxwell_gcgemm_64x64_tn_batched
28 kernel group(s) hidden...
Stream 12
AllReduceKernelSmall
AllReduceKernel
1 kernel group(s) hidden...
67 stream(s) hidden...
Stream 173
Stream 20
66 stream(s) hidden...
Thread/core migration
Processes and threads
Thread state
CUDA and OpenGL API trace
cuDNN and cuBLAS trace
Kernel and memory transfer activities
Multi-GPU

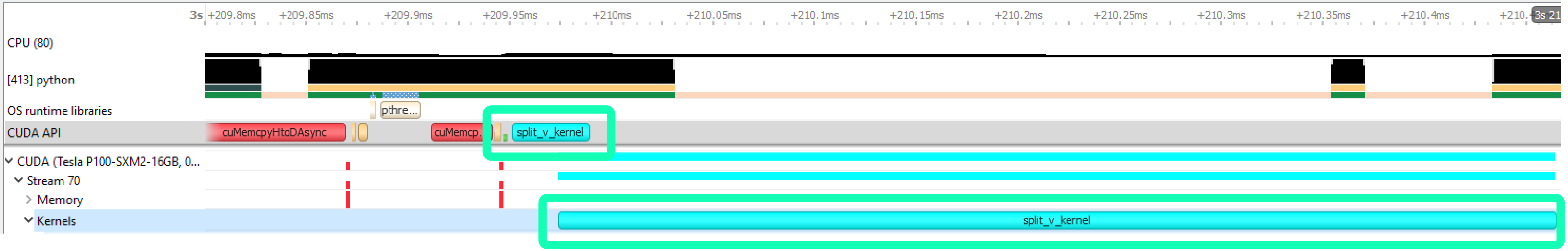
NSIGHT SYSTEMS

OS runtime trace



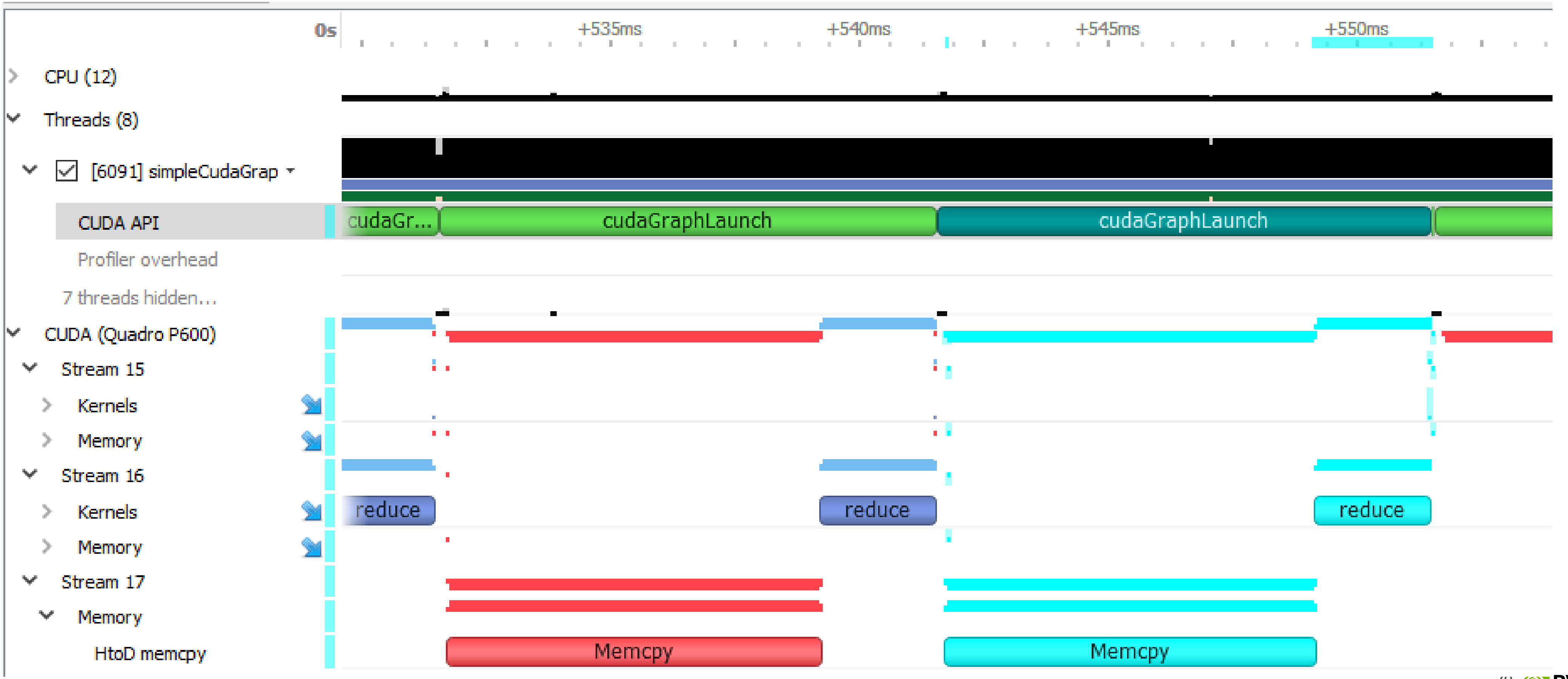
NSIGHT SYSTEMS

GPU API launch to HW workload correlation



NSIGHT SYSTEMS

CUDA 10 Graphs – all related GPU ranges



TRANSITIONING TO PROFILE A KERNEL

Dive into kernel analysis

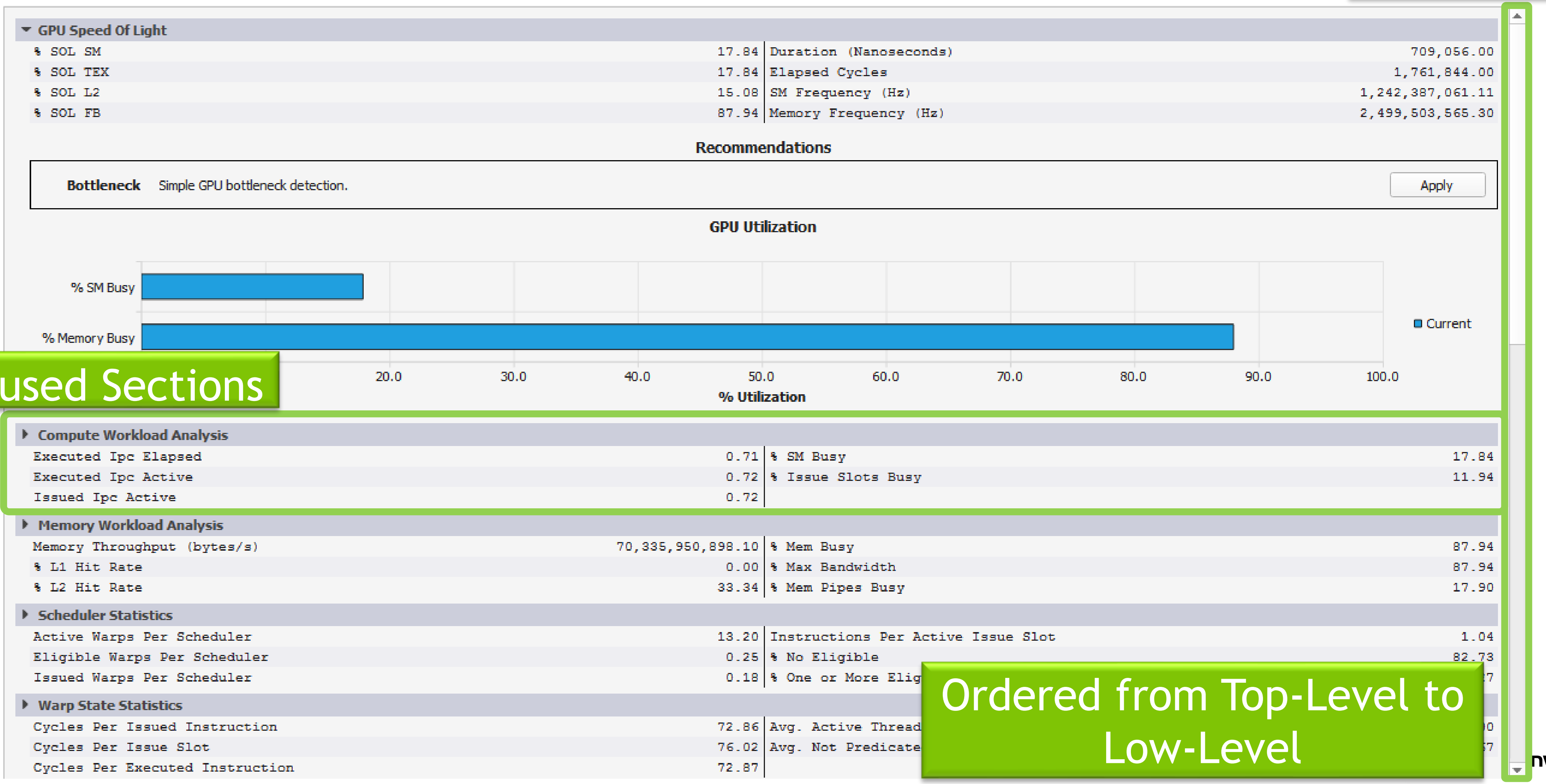
NVIDIA

NSIGHT COMPUTE

NSIGHT COMPUTE

Profile Report – Details Page

All Data on Single Page



Focused Sections

Ordered from Top-Level to Low-Level

NVIDIA.

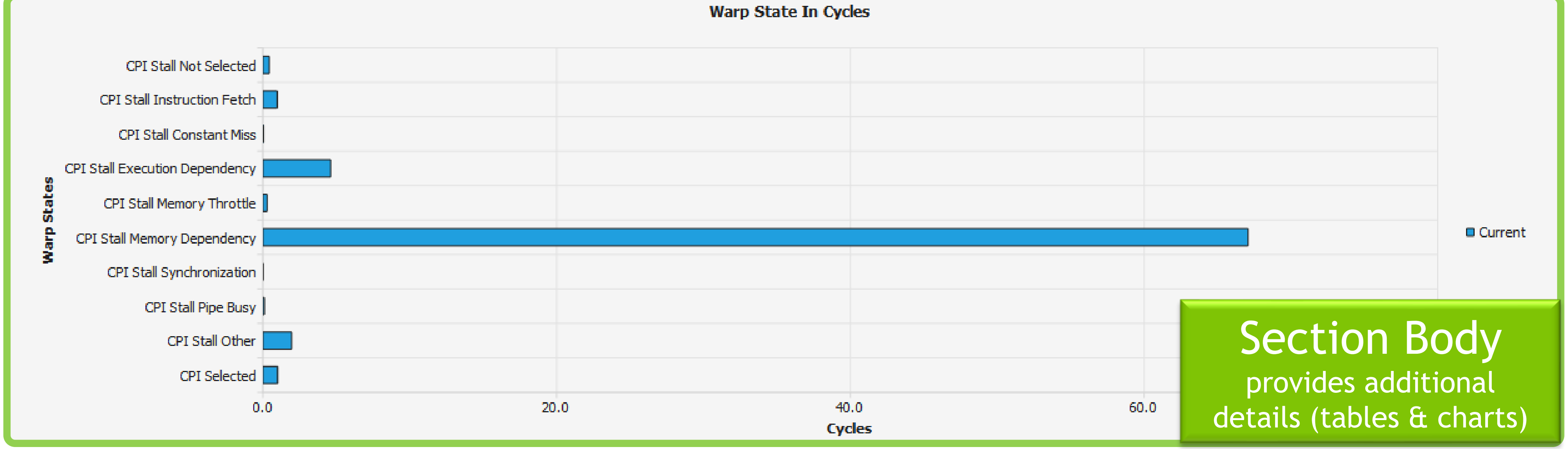
NSIGHT COMPUTE

Section Example

Section Header
provides overview & context for other sections

Warp State Statistics

Cycles Per Issued Instruction	72.86	Avg. Active Threads Per Warp	32.00
Cycles Per Issue Slot	76.02	Avg. Not Predicated Off Threads Per Warp	30.67
Cycles Per Executed Instruction	72.87		



Section Body
provides additional details (tables & charts)

Section Config
completely data driven
add/modify/change sections

NSIGHT COMPUTE

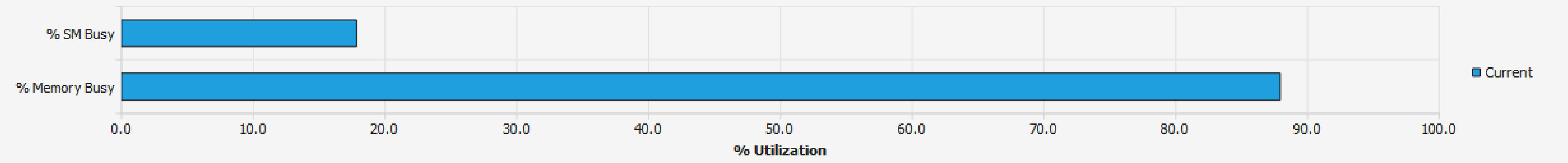
Unguided Analysis / Rules System

Analysis Rules
recommendations from nvvp and more

17.84	Duration (Nanoseconds)	709,056.00
17.84	Elapsed Cycles	1,761,844.00
15.08	SM Frequency (Hz)	1,242,387,061.11
87.94	Memory Frequency (Hz)	2,499,503,565.30

Recommendations

Bottleneck [Warning] Memory is more heavily utilized than Compute: Look at `Memory Workload Analysis` report section to see where the memory system bottleneck is. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can recompute.



Rules Config
completely data driven
add/modify/change rules

NSIGHT COMPUTE

Diffing kernel runs

Baseline
from any previous profile report (different kernel, gpu, …)

Metric delta
current values and changes from baseline

▾ GPU Speed Of

% SOL SM	18.23 (+318.69%)	Duration (Nanoseconds)	190,208.00 (-76.03%)
% SOL TEX	18.23 (+39.56%)	Elapsed Cycles	470,600.00 (-76.12%)
% SOL L2	15.42 (+4.81%)	SM Frequency (Hz)	1,237,066,790.04 (-0.37%)
% SOL FB	86.97 (+1.36%)	Memory Frequency (Hz)	2,499,327,052.49 (-0.02%)

Recommendations

Bottleneck Simple GPU bottleneck detection. Apply

GPU Utilization

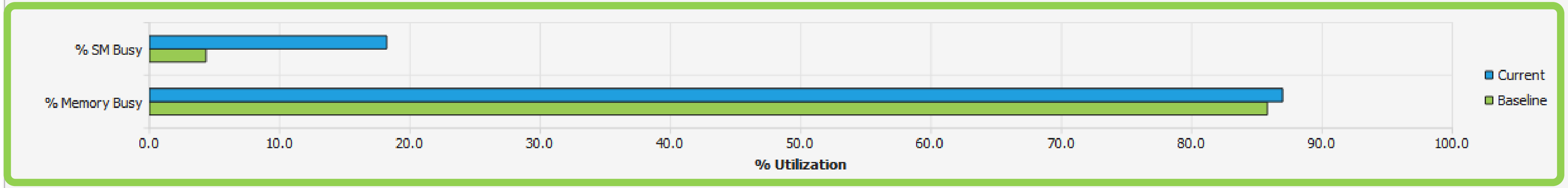


Chart difference
current values and baseline values

	0.61 (+318.69%)	% SM Busy	18.23 (+318.69%)
	0.61 (+321.96%)	% Issue Slots Busy	10.25 (+321.96%)
	0.61 (+321.96%)		

	69,557,705,248.99 (+1.33%)	% Mem Busy	86.97 (+1.36%)
% L1 Hit Rate	100.00 (+0.00%)	% Max Bandwidth	86.97 (+1.36%)
% L2 Hit Rate	33.34 (+0.02%)	% Mem Pipes Busy	18.43 (+321.96%)

▸ Scheduler Statistics

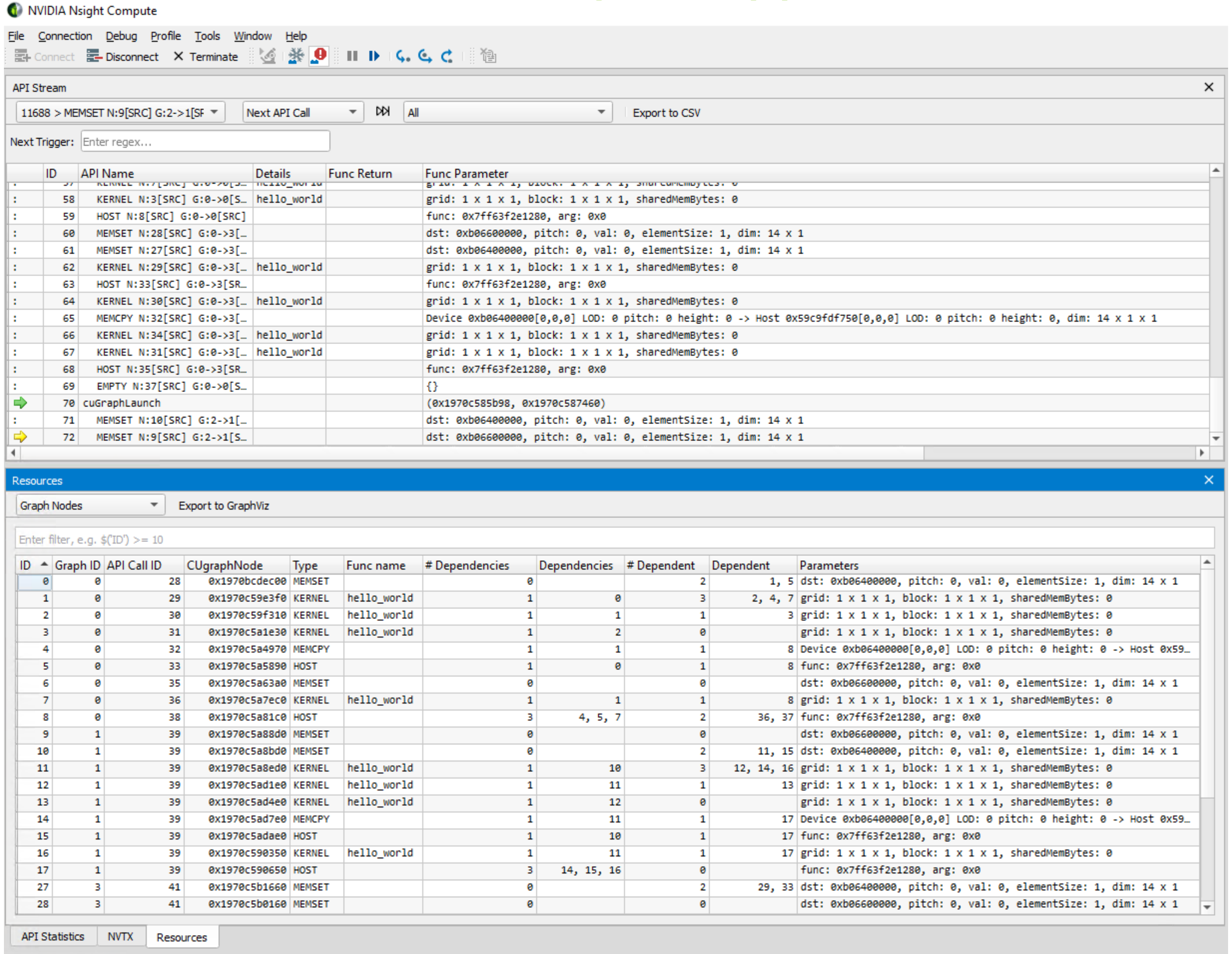
Active Warps Per Scheduler	13.83 (+1.73%)	Instructions Per Active Issue Slot	1.11 (+0.00%)
Eligible Warps Per Scheduler	0.17 (+383.47%)	% No Eligible	86.06 (-11.00%)
Issued Warps Per Scheduler	0.15 (+322.90%)	% One or More Eligible	13.92 (+322.90%)

▸ Warp State Statistics

Cycles Per Issued Instruction	89.24 (-75.99%)	Avg. Active Threads Per Warp	32.00 (+0.00%)
Cycles Per Issue Slot	99.14 (-75.99%)	Avg. Not Predicated Off Threads Per Warp	30.40 (+0.00%)

NSIGHT COMPUTE

CUDA 10 Graphs support



ID	API Name	Details	Func Return	Func Parameter
58	KERNEL N:3[SRC] G:0->0[S…	hello_world		grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
59	HOST N:8[SRC] G:0->0[SRC]			func: 0x7ff63f2e1280, arg: 0x0
60	MEMSET N:28[SRC] G:0->3[…			dst: 0xb06600000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
61	MEMSET N:27[SRC] G:0->3[…			dst: 0xb06400000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
62	KERNEL N:29[SRC] G:0->3[…	hello_world		grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
63	HOST N:33[SRC] G:0->3[SR…			func: 0x7ff63f2e1280, arg: 0x0
64	KERNEL N:30[SRC] G:0->3[…	hello_world		grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
65	MEMCPY N:32[SRC] G:0->3[…			Device 0xb06400000[0,0,0] LOD: 0 pitch: 0 height: 0 -> Host 0x59c9fdf750[0,0,0] LOD: 0 pitch: 0 height: 0, dim: 14 x 1 x 1
66	KERNEL N:34[SRC] G:0->3[…	hello_world		grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
67	KERNEL N:31[SRC] G:0->3[…	hello_world		grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
68	HOST N:35[SRC] G:0->3[SR…			func: 0x7ff63f2e1280, arg: 0x0
69	EMPTY N:37[SRC] G:0->0[S…			{}
70	cuGraphLaunch			(0x1970c585b98, 0x1970c587460)
71	MEMSET N:10[SRC] G:2->1[…			dst: 0xb06400000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
72	MEMSET N:9[SRC] G:2->1[S…			dst: 0xb06600000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1

ID	Graph ID	API Call ID	CUgraphNode	Type	Func name	# Dependencies	Dependencies	# Dependent	Dependent	Parameters
0	0	28	0x1970bcdec00	MEMSET		0		2	1, 5	dst: 0xb06400000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
1	0	29	0x1970c59e3f0	KERNEL	hello_world	1	0	3	2, 4, 7	grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
2	0	30	0x1970c59f310	KERNEL	hello_world	1	1	1	3	grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
3	0	31	0x1970c5a1e30	KERNEL	hello_world	1	2	0		grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
4	0	32	0x1970c5a4970	MEMCPY		1	1	1	8	Device 0xb06400000[0,0,0] LOD: 0 pitch: 0 height: 0 -> Host 0x59…
5	0	33	0x1970c5a5890	HOST		1	0	1	8	func: 0x7ff63f2e1280, arg: 0x0
6	0	35	0x1970c5a63a0	MEMSET		0		0		dst: 0xb06600000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
7	0	36	0x1970c5a7ec0	KERNEL	hello_world	1	1	1	8	grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
8	0	38	0x1970c5a81c0	HOST		3	4, 5, 7	2	36, 37	func: 0x7ff63f2e1280, arg: 0x0
9	1	39	0x1970c5a88d0	MEMSET		0		0		dst: 0xb06600000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
10	1	39	0x1970c5a8bd0	MEMSET		0		2	11, 15	dst: 0xb06400000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
11	1	39	0x1970c5a8ed0	KERNEL	hello_world	1	10	3	12, 14, 16	grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
12	1	39	0x1970c5ad1e0	KERNEL	hello_world	1	11	1	13	grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
13	1	39	0x1970c5ad4e0	KERNEL	hello_world	1	12	0		grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
14	1	39	0x1970c5ad7e0	MEMCPY		1	11	1	17	Device 0xb06400000[0,0,0] LOD: 0 pitch: 0 height: 0 -> Host 0x59…
15	1	39	0x1970c5adae0	HOST		1	10	1	17	func: 0x7ff63f2e1280, arg: 0x0
16	1	39	0x1970c590350	KERNEL	hello_world	1	11	1	17	grid: 1 x 1 x 1, block: 1 x 1 x 1, sharedMemBytes: 0
17	1	39	0x1970c590650	HOST		3	14, 15, 16	0		func: 0x7ff63f2e1280, arg: 0x0
27	3	41	0x1970c5b1660	MEMSET		0		2	29, 33	dst: 0xb06400000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1
28	3	41	0x1970c5b0160	MEMSET		0		0		dst: 0xb06600000, pitch: 0, val: 0, elementSize: 1, dim: 14 x 1

NSIGHT COMPUTE

CUDA 10 Graphs export