



NVIDIA PROFILING TOOLS OVERVIEW

François Courteille, Principal Solution Architect, fcourteille@nvidia.com



AGENDA

Introduction to NVIDIA profilers

Nvprof and the Visual Profiler

Nsight Systems

Profiling from CLI

NVTX (NVIDIA Tools Extension)

Getting Started Resources

NVIDIA PROFILING TOOLS

Nsight Systems



Nsight Compute



Nsight Visual Studio Edition

Nsight Eclipse Edition



NVIDIA Profiler (nvprof)

NVIDIA Visual Profiler (nvvp)



CUPTI (CUDA Profiling Tools Interface)

TOOLS COMPARISON

	NVIDIA© Nsight™ Systems	NVIDIA© Nsight™ Compute	NVIDIA© Visual Profiler	Intel© VTune™ Amplifier	Linux perf OProfile
Target OS	Linux, Windows	Linux, Windows	Linux, Mac, Windows	Linux, Windows	Linux
GPUs	Pascal+	Pascal+	Kepler+	None	None
CPUs	x86_64	x86_64	x86, x86_64, Power	x86, x86_64	x86, x86_64, Power
Trace	NVTX, OS runtime, CUDA, CuDNN, CuBLAS, OpenACC, OpenGL, DX12	NVTX, CUDA	MPI, CUDA, OpenACC, NVTX	MPI, ITT	Kernel
CPU PC Sampling	High Speed	No	Yes	High Speed	High Speed
NVLINK, GPU Power, Thermal	Future		Yes	No	No
Src Code View	No	Yes	Yes	Yes	No
Compare Sessions	No	Yes	No	Yes	No

The background features a complex network of thin, light green lines connecting various nodes. The nodes are represented by small, bright green circles of varying sizes, some of which are slightly blurred, giving a sense of depth and movement. The overall aesthetic is futuristic and technical, typical of a presentation on computer graphics or AI.

NVPROF AND THE VISUAL PROFILER

NVIDIA PROFILER

Some examples of how to use it

- GUI : `nvvp`
- Command line:
`nvprof <application> <application options>`
- Command line with GUI output:
`nvprof -o prof.nvvp <application> <application options>`
- Command line, MPI applications, GUI output:
`mpirun -np 4 nvprof -o prof_%h_%p.nvvp <application> <application options>`
(%h and %p will be replaced by host and pid)
- Command line, capturing all low level metrics for later GUI analysis (slow!)
`nvprof --analysis-metrics -o prof.nvvp <application> <application options>`
- Command line, all low level metrics for later GUI analysis only for the 2nd occurrence of each kernel:
`nvprof --kernels :::2 --analysis-metrics -o prof.nvvp <application> <application options>`
- Command line with GUI output, limiting the profiling to the first 60seconds (60s after GPU context creation)
`nvprof -t 60 -o prof.nvvp <application> <application options>`

See the profiler documentation
for more information

NVPROF - Basic Usage

```
$ nvprof ./cg
==88109== NVPROF is profiling process 88109, command: ./cg
<program output>
==88109== Profiling application: ./cg
==88109== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
 95.62%  4.74211s      101  46.952ms  43.029ms  435.09ms  matvec(matrix const &, vector const &,
vector const &)_12_gpu
  3.28%  162.50ms      302  538.09us  236.55us  31.155ms  waxpby(double, vector const &, double,
vector const &, vector const &)_26_gpu
  0.73%  36.165ms      200  180.83us  138.18us  223.26us  dot(vector const &, vector const
&)_10_gpu
  0.37%  18.310ms      200  91.549us  89.664us  93.441us  dot(vector const &, vector const
&)_10_gpu_red
  0.00%  100.13us      200    500ns    480ns    1.0240us  [CUDA memcpy HtoD]
  0.00%  81.408us      200    407ns    384ns    416ns    [CUDA memcpy DtoH]

==88109== Unified Memory profiling result:
Device "Tesla P100-SXM2-16GB (0)"
  Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
  13783  197.40KB  64.000KB  960.00KB  2.594727GB  99.26990ms  Host To Device
   7681      -      -      -      -      298.1818ms  GPU Page fault groups
Total CPU Page faults: 7973
```

CUDA VISUAL PROFILER

Overview of key features

Kernel profile - memory hierarchy view

Unified Memory

NVLink

PC sampling

OpenACC/OpenMP Profiling

NVTX

NVIDIA'S VISUAL PROFILER (NVVP)

Guided System

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

[Perform Compute Analysis](#)

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

[Perform Latency Analysis](#)

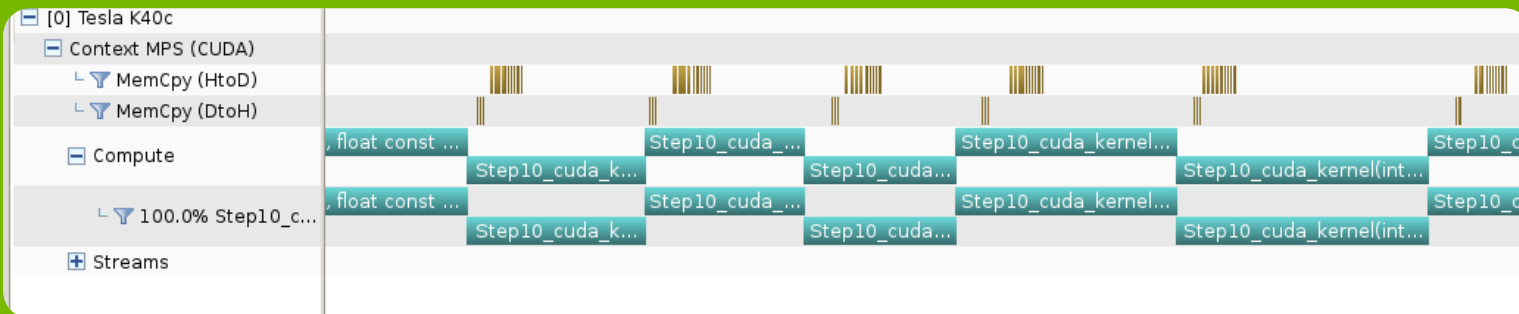
[Perform Memory Bandwidth Analysis](#)

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

[Rerun Analysis](#)

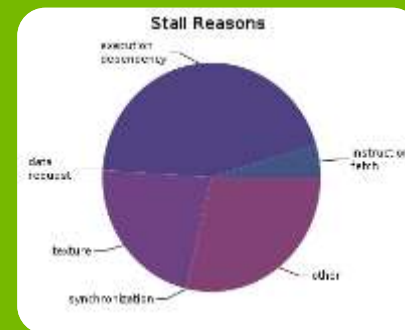
If you modify the kernel you need to rerun your application to update this analysis.

Timeline

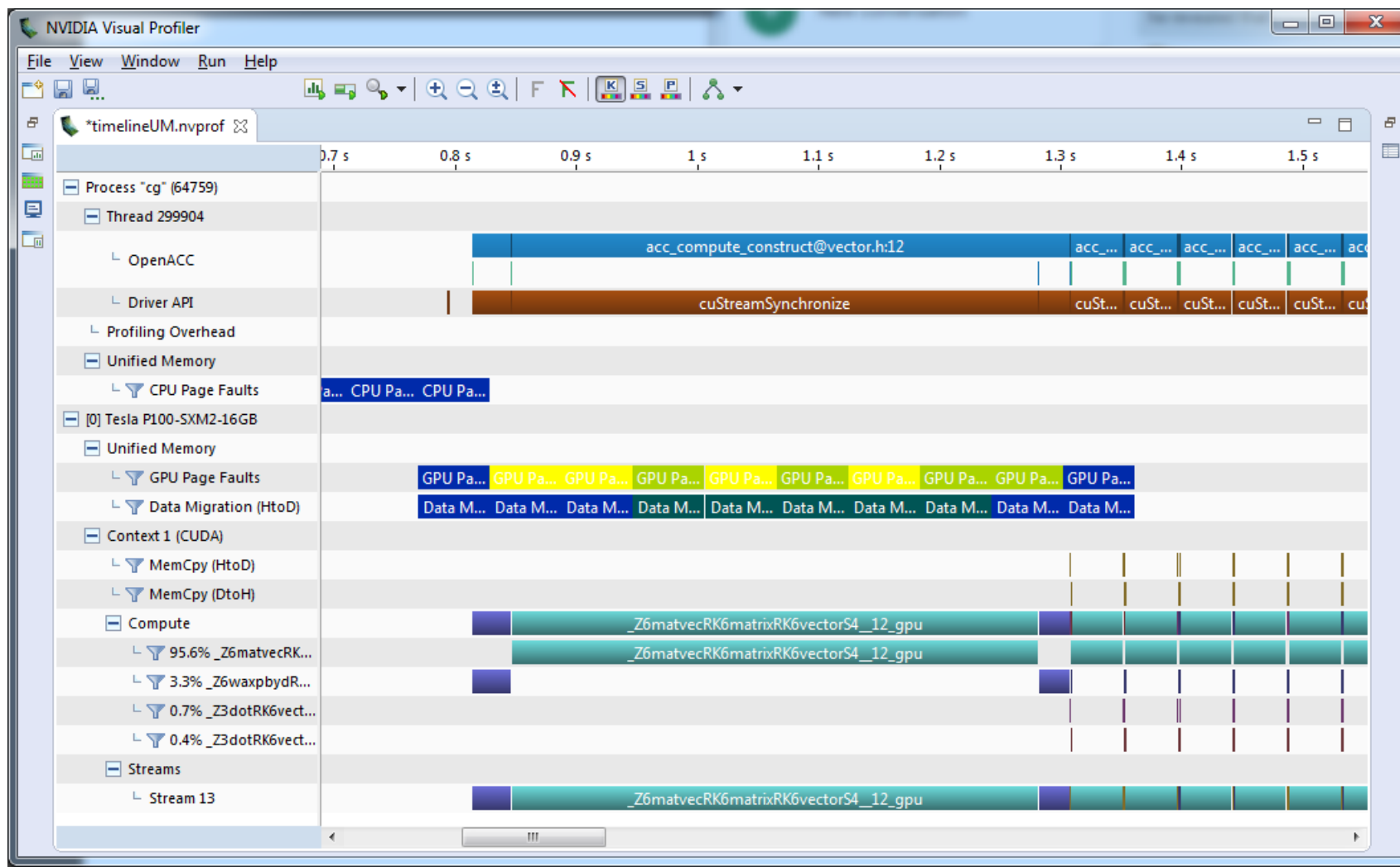


Analysis

Category	Value	Unit
Local Memory	0	0.000
Global Memory	0	0.000
Shared Memory	0	0.000
Global L2 Cache	0	0.000
Global L1 Cache	0	0.000
Global L0 Cache	0	0.000
Global L3 Cache	0	0.000
Global L4 Cache	0	0.000
Global L5 Cache	0	0.000
Global L6 Cache	0	0.000
Global L7 Cache	0	0.000
Global L8 Cache	0	0.000
Global L9 Cache	0	0.000
Global L10 Cache	0	0.000
Global L11 Cache	0	0.000
Global L12 Cache	0	0.000
Global L13 Cache	0	0.000
Global L14 Cache	0	0.000
Global L15 Cache	0	0.000
Global L16 Cache	0	0.000
Global L17 Cache	0	0.000
Global L18 Cache	0	0.000
Global L19 Cache	0	0.000
Global L20 Cache	0	0.000
Global L21 Cache	0	0.000
Global L22 Cache	0	0.000
Global L23 Cache	0	0.000
Global L24 Cache	0	0.000
Global L25 Cache	0	0.000
Global L26 Cache	0	0.000
Global L27 Cache	0	0.000
Global L28 Cache	0	0.000
Global L29 Cache	0	0.000
Global L30 Cache	0	0.000
Global L31 Cache	0	0.000
Global L32 Cache	0	0.000
Global L33 Cache	0	0.000
Global L34 Cache	0	0.000
Global L35 Cache	0	0.000
Global L36 Cache	0	0.000
Global L37 Cache	0	0.000
Global L38 Cache	0	0.000
Global L39 Cache	0	0.000
Global L40 Cache	0	0.000
Global L41 Cache	0	0.000
Global L42 Cache	0	0.000
Global L43 Cache	0	0.000
Global L44 Cache	0	0.000
Global L45 Cache	0	0.000
Global L46 Cache	0	0.000
Global L47 Cache	0	0.000
Global L48 Cache	0	0.000
Global L49 Cache	0	0.000
Global L50 Cache	0	0.000
Global L51 Cache	0	0.000
Global L52 Cache	0	0.000
Global L53 Cache	0	0.000
Global L54 Cache	0	0.000
Global L55 Cache	0	0.000
Global L56 Cache	0	0.000
Global L57 Cache	0	0.000
Global L58 Cache	0	0.000
Global L59 Cache	0	0.000
Global L60 Cache	0	0.000
Global L61 Cache	0	0.000
Global L62 Cache	0	0.000
Global L63 Cache	0	0.000
Global L64 Cache	0	0.000
Global L65 Cache	0	0.000
Global L66 Cache	0	0.000
Global L67 Cache	0	0.000
Global L68 Cache	0	0.000
Global L69 Cache	0	0.000
Global L70 Cache	0	0.000
Global L71 Cache	0	0.000
Global L72 Cache	0	0.000
Global L73 Cache	0	0.000
Global L74 Cache	0	0.000
Global L75 Cache	0	0.000
Global L76 Cache	0	0.000
Global L77 Cache	0	0.000
Global L78 Cache	0	0.000
Global L79 Cache	0	0.000
Global L80 Cache	0	0.000
Global L81 Cache	0	0.000
Global L82 Cache	0	0.000
Global L83 Cache	0	0.000
Global L84 Cache	0	0.000
Global L85 Cache	0	0.000
Global L86 Cache	0	0.000
Global L87 Cache	0	0.000
Global L88 Cache	0	0.000
Global L89 Cache	0	0.000
Global L90 Cache	0	0.000
Global L91 Cache	0	0.000
Global L92 Cache	0	0.000
Global L93 Cache	0	0.000
Global L94 Cache	0	0.000
Global L95 Cache	0	0.000
Global L96 Cache	0	0.000
Global L97 Cache	0	0.000
Global L98 Cache	0	0.000
Global L99 Cache	0	0.000

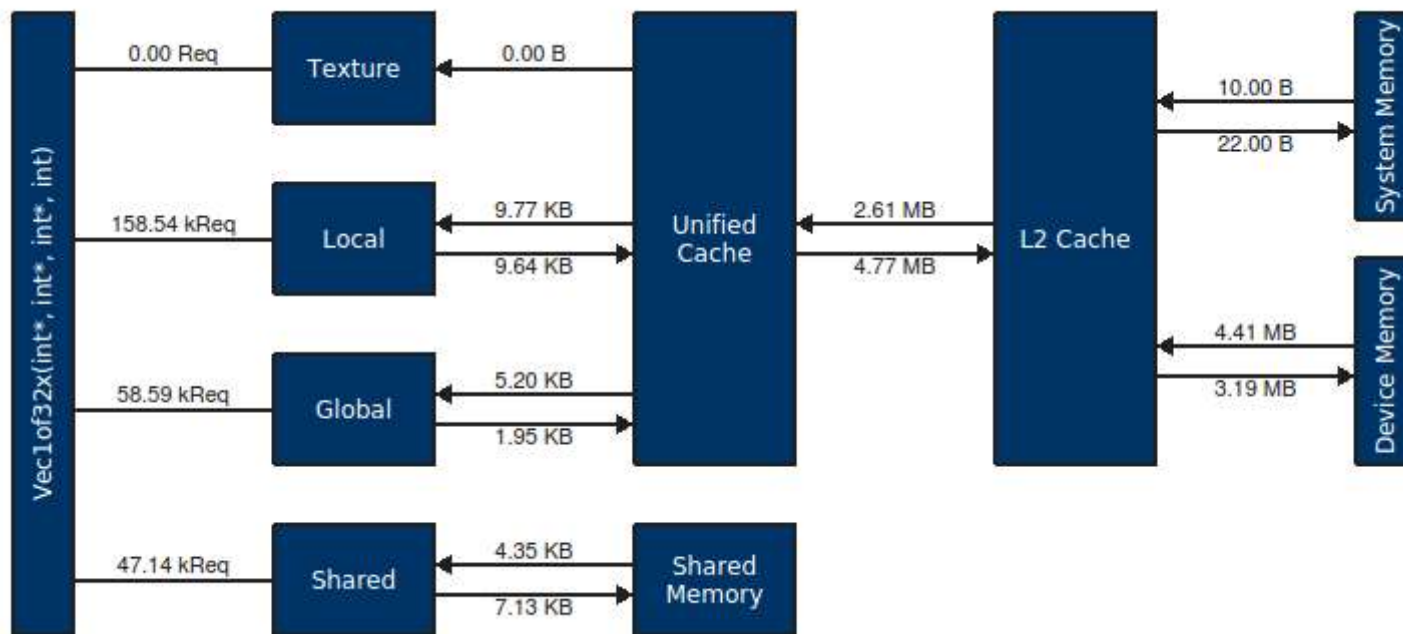


UVM IN VISUAL PROFILER



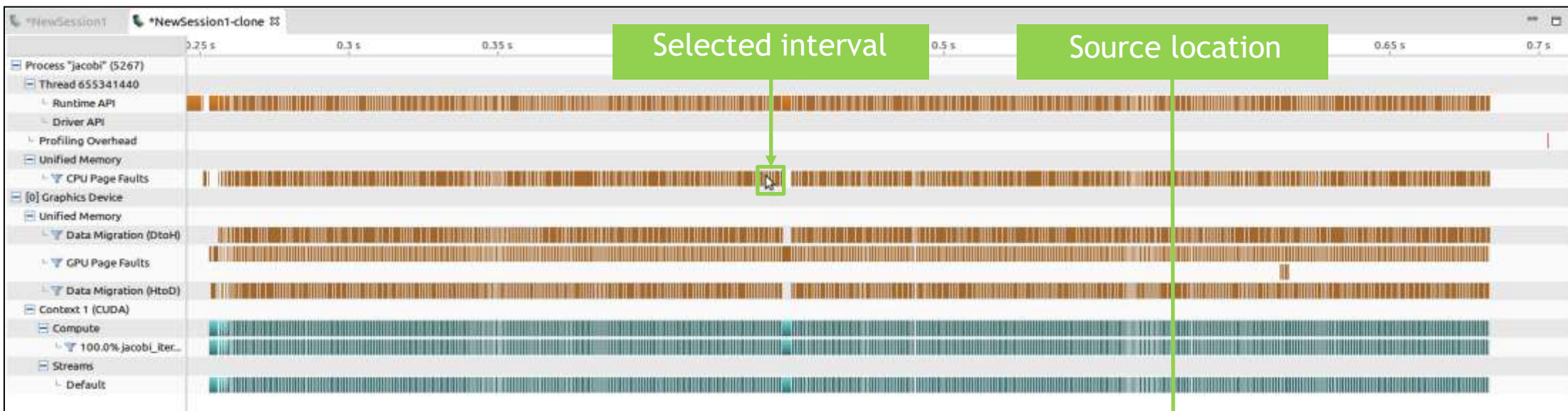
KERNEL PROFILE

Memory hierarchy view



VISUAL PROFILER

CPU Page Fault Source Correlation



Properties	
CPU Page Faults	
Timestamp	440.45958 ms (440,459,581 ns)
Memory Access Type	Write
Virtual Address	0x900100000
Source Location	main@jacobi.cu:130
Process	25684

VISUAL PROFILER

CPU Page Fault Source Correlation

Properties	
CPU Page Faults	
Timestamp	440.45958 ms (440,459,581 ns)
Memory Access Type	Write
Virtual Address	0x900100000
Source Location	main@jacobi.cu:130
Process	25684

Source line causing
CPU page fault

```
*NewSession1  jacobi.cu

float * a;
float * a_new;
float * weights;

CUDA_CALL(cudaMallocManaged(&a, nx*ny*sizeof(float)));
CUDA_CALL(cudaMallocManaged(&a_new, nx*ny*sizeof(float)));
CUDA_CALL(cudaMallocManaged(&weights, n_weights*sizeof(float)));

init(a,a_new,nx,ny,weights,n_weights);

cudaEvent_t start,stop;
CUDA_CALL(cudaEventCreate(&start));
CUDA_CALL(cudaEventCreate(&stop));

CUDA_CALL(cudaDeviceSynchronize());
CUDA_CALL(cudaEventRecord(start));

PUSH_RANGE("while loop",0)
int iter = 0;
while ( iter <= iter_max )
{
    PUSH_RANGE("jacobi step",1)
    jacobi_iteration<<<dim3(nx/32,ny/4),dim3(32,4)>>>(a_new,a,nx,ny,weights[0]);
    CUDA_CALL(cudaGetLastError());
    CUDA_CALL(cudaDeviceSynchronize());
    POP_RANGE

    std::swap(a,a_new);

    PUSH_RANGE("periodic boundary conditions",2)
    //Apply periodic boundary conditions
    for (int ix = 0; ix < nx; ++ix)
    {
        a[ 0*nx+ix]=a[(ny-2)*nx+ix];
        a[(ny-1)*nx+ix]=a[ 1*nx+ix];
    }
    POP_RANGE

    if ( 0 == iter%100 )
        std::cout<<iter<<std::endl;
    iter++;
}

CUDA_CALL(cudaEventRecord(stop));
CUDA_CALL(cudaDeviceSynchronize());
POP_RANGE
```


VISUAL PROFILER

NVLINK visualization

Unguided Analysis

The screenshot shows the 'NVLink Analysis' results in the Visual Profiler. It includes a topology diagram, a legend for bandwidth usage, and two tables: 'Logical NVLink Properties' and 'Logical NVLink Throughput'. Annotations in green boxes point to various features: 'Unguided Analysis' points to the top-left navigation; 'Option to collect NVLink information' points to the 'NVLink' checkbox in the left sidebar; 'Version' points to 'NVLink version 1.0'; 'Color codes for NVLink' points to the bandwidth usage legend; 'Topology' points to the central diagram; 'Selected NVLink' points to the 'GPU0->GPU1' entry in the throughput table; 'Static properties' points to the 'Logical NVLink Properties' table; and 'Runtime values' points to the 'Utilization %' and 'Idle time %' columns in the same table.

Logical NVLink Properties

Logical NVLink	Peak Bandwidth	Physical NVLinks	Peer Access	System Access	Peer Atomic	System Atomic	Utilization %	Idle time %
GPU0->CPU0	80 GB/s	2	No	Yes	No	No	0	10
GPU0-<-CPU0	80 GB/s	2	Yes	No	Yes	No	n/a	100
GPU1-<-GPU0	80 GB/s	2	No	Yes	No	No	0	10
GPU2-<-GPU1	80 GB/s	2	Yes	No	Yes	No	n/a	100
GPU3-<-GPU1	80 GB/s	2	No	Yes	No	No	0	10

Logical NVLink Throughput

Logical NVLink	Avg Throughput	Max Throughput	Min Throughput
GPU0->CPU0	90.917 MB/s	36.085 GB/s	5.691 kB/s
GPU0-<-CPU0	138.065 MB/s	32.203 GB/s	1.897 kB/s
GPU0->GPU1	0 B/s	0 B/s	0 B/s
GPU0-<-GPU1	0 B/s	0 B/s	0 B/s
GPU1->CPU0	90.777 MB/s	36.031 GB/s	5.847 kB/s
GPU1-<-CPU0	138.241 MB/s	33.14 GB/s	1.949 kB/s
GPU2->CPU1	96.692 MB/s	14.798 GB/s	14.791 kB/s
GPU2-<-CPU1	141.799 MB/s	16.495 GB/s	5.764 kB/s
GPU2->GPU3	0 B/s	0 B/s	0 B/s
GPU2-<-GPU3	0 B/s	0 B/s	0 B/s
GPU3->CPU1	96.856 MB/s	14.786 GB/s	13.863 kB/s
GPU3-<-CPU1	142.063 MB/s	16.497 GB/s	1.764 kB/s

Option to collect NVLink information

Version

Color codes for NVLink

Topology

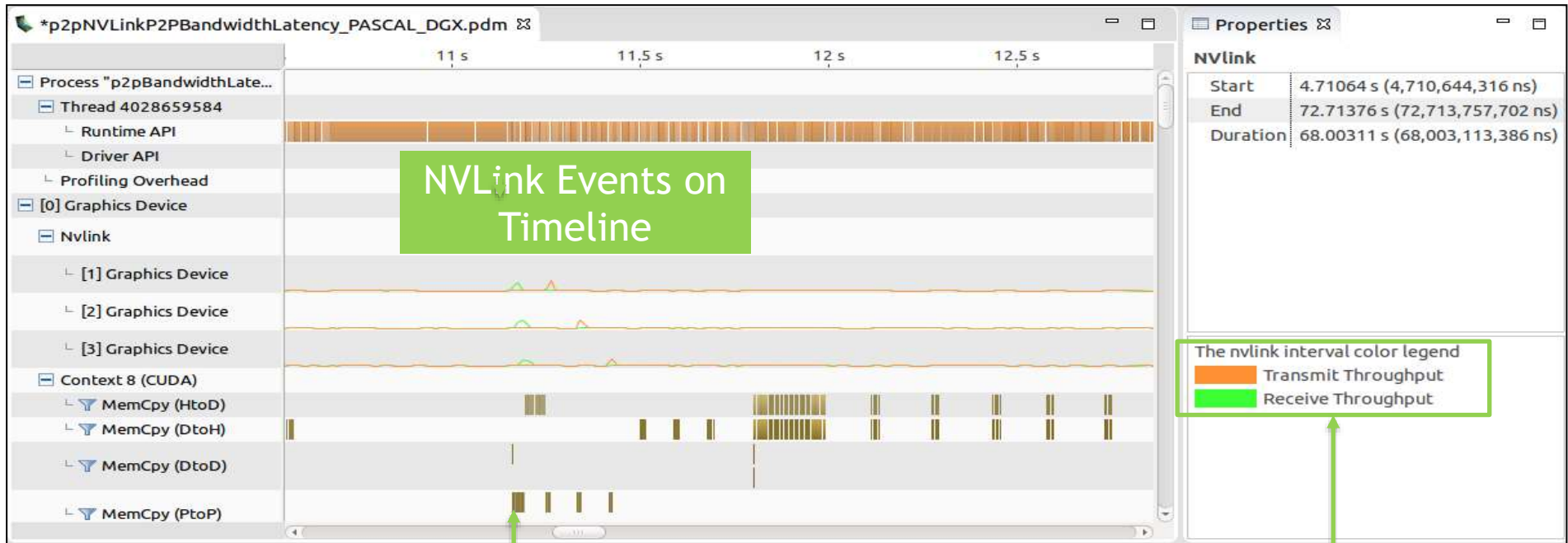
Selected NVLink

Static properties

Runtime values

VISUAL PROFILER

NVLink events on timeline



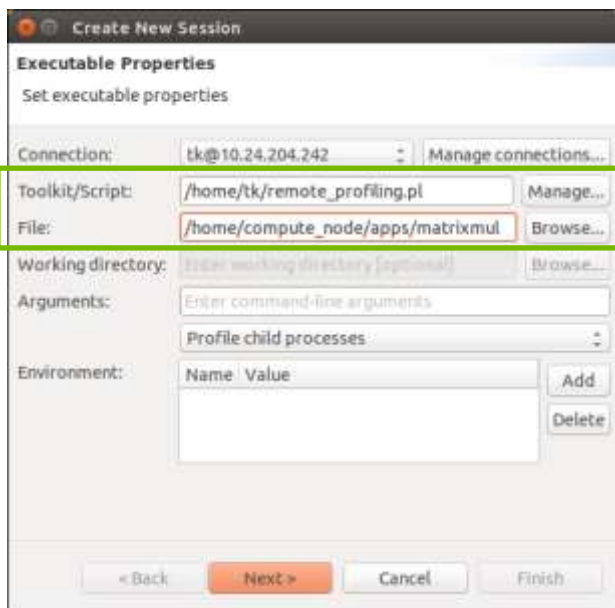
MemCpy API

Color Coding of NVLink Events

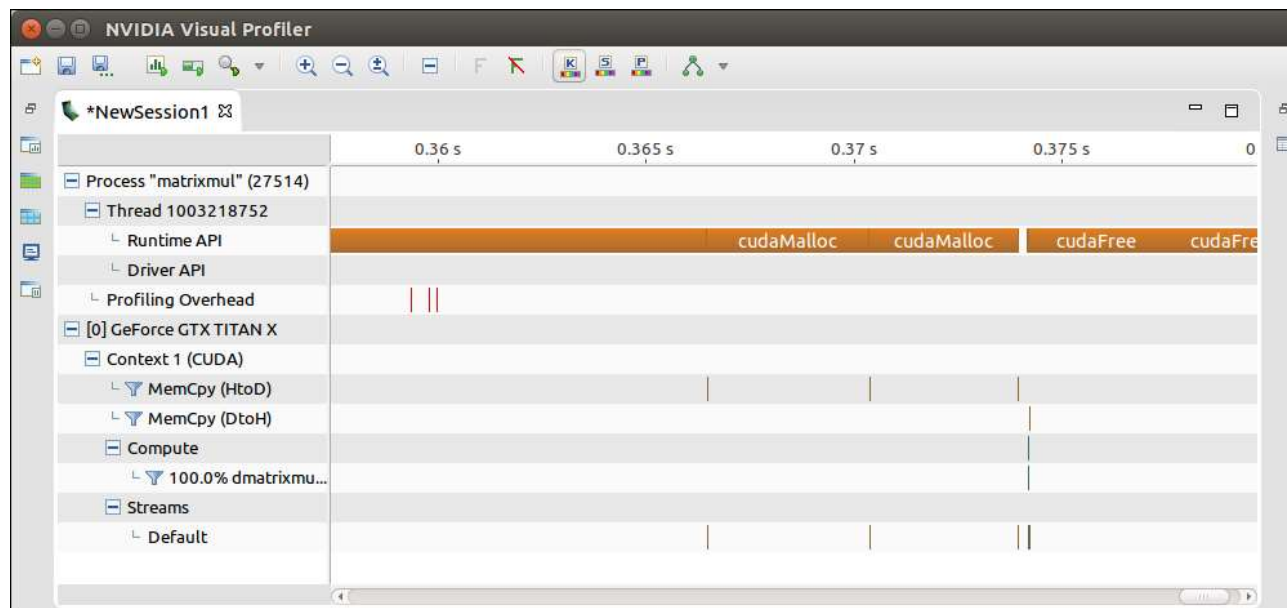
VISUAL PROFILER

Multi-hop remote profiling - Application Profiling

- 1 Select custom script, then create a remote session as usual



- 2 Application transparently runs on compute node and profiling data is displayed in the Visual Profiler

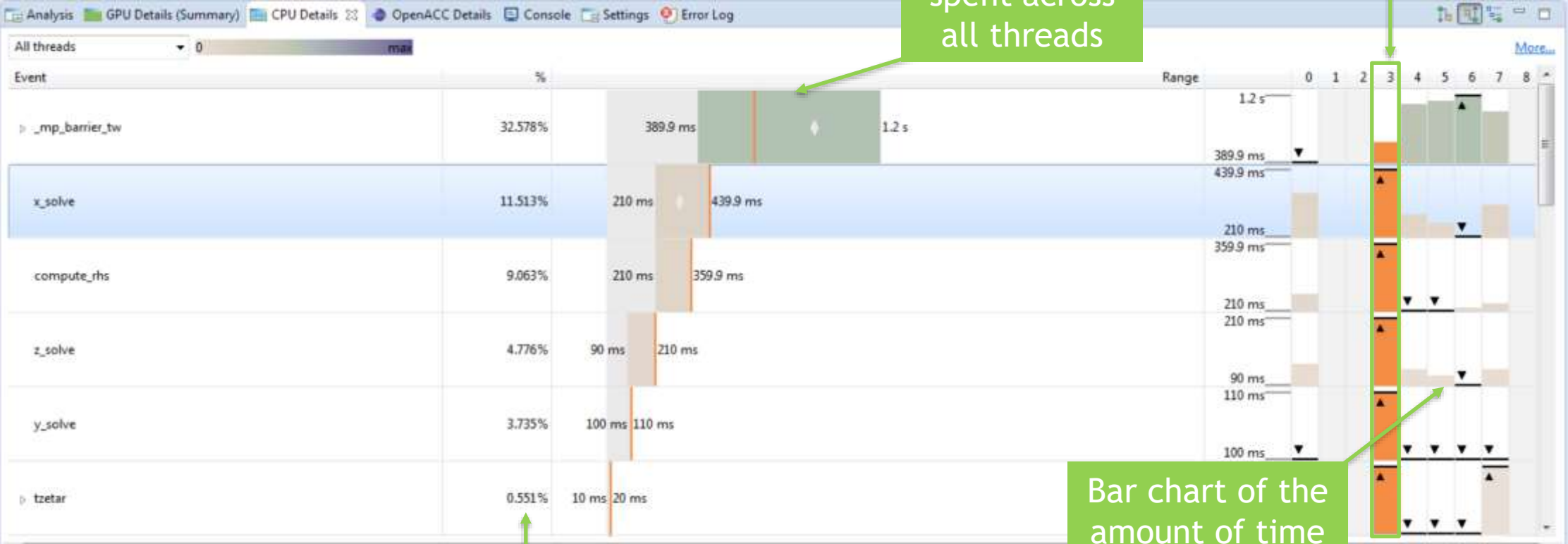


CPU SAMPLING

- 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



Range of time spent across all threads

Selected thread is highlighted in Orange

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 \geq 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

The screenshot shows the 'Settings' window in the Visual Profiler application. The window title is 'Session NewSession1'. The 'Analysis' section is active, showing various configuration options. Under the 'PC Sampling' section, the 'Select sampling period' radio button is selected, and the value '8' is entered in the adjacent input field. A tooltip points to this field with the text 'The actual sampling period will be in 2^n cycles'. Other settings include 'Device: [0] Graphics Device', 'PCIe Generation: 2', 'PCIe Link Width: 4', and 'PCIe Link Rate: 5 Gbit/s', each with an 'Override' field for manual input.

Analysis

PCIe Override: Rerun analysis after updating

Device: [0] Graphics Device

PCIe Generation: 2 Override: Enter PCIe generation to use for analysis, allowed values are 2, 3 [optional, clear to use default]

PCIe Link Width: 4 Override: Enter PCIe link width to use for analysis, allowed values are 1, 2, 4, 8, 16, 32 [optional, clear to use default]

PCIe Link Rate: 5 Gbit/s Override: Enter PCIe link rate to use for analysis in Mbits/s [optional, clear to use default]

Sampling period: Rerun Kernel Profile - PC Sampling analysis after updating

Default sampling period Select sampling period 8

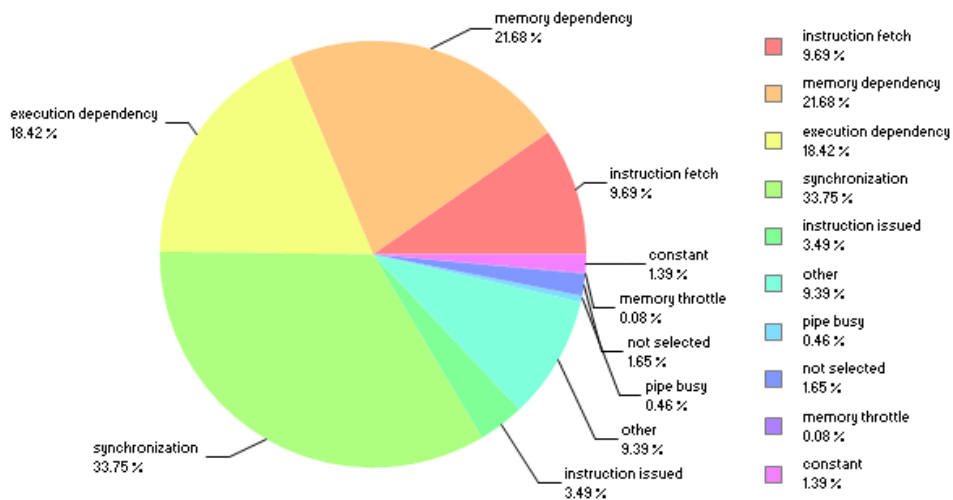
The actual sampling period will be in 2^n cycles

VISUAL PROFILER

PC SAMPLING UI

Pie chart for sample distribution for a CUDA function

Sample distribution



Hotspots

Hotspot source line with highest number of samples

Stack bar with stall reasons

```
Line Warp State File - /C:/swapnaofficial/Maxwell/sampling/SFM/session1/estimate_combined1.cu
188 int tid = threadIdx.z * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;
189
191 shift = 0;
192 for (int i = 0; i < 6; ++i) //rows
193 {
194 #pragma unroll
195 for (int j = i; j < 7; ++j) // cols + b
196 {
197 _syncthreads ();
198 smem[tid] = row[i] * row[j];
199 _syncthreads ();
200
201 reduce<CTA_SIZE>(smem);
```

Warp State Disassembly

```
LLL R4, [R4];
MOV R5, R3;
MOV R7, R4;
L_8:
BAR.SYNC 0x0;
FMUL.FTZ R7, R4, R7;
STS [R0], R7;
BAR.SYNC 0x0;
CONSTANT AND P0, PT, R13, 0x7f, PT;
```

Tooltip with distribution of stall reasons

Total Sample Count = 1288
synchronization = 847 (65.8%)
other = 263 (20.4%)
instruction fetch = 69 (5.4%)
execution dependency = 36 (2.8%)
memory dependency = 32 (2.5%)
instruction issued = 23 (1.8%)
pipe busy = 9 (0.7%)
not selected = 8 (0.6%)
memory throttle = 1 (0.1%)

Source-Assembly view

NVPROF - MPI Profiling

NVPROF & Visual Profiler do not natively understand MPI

It is possible to load data from multiple MPI ranks (same or different GPUS) into Visual Profiler though

Trick: Label your data to know which MPI rank it came from:

- `--process-name "MPI Rank %q{OMPI_COMM_WORLD_RANK}" --context-name "MPI Rank %q{OMPI_COMM_WORLD_RANK}"`

Trick: Give every rank its own filename

- `-o timeline.%q{OMPI_COMM_WORLD_RANK}.nvprof`

See <https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-track-mpi-calls-nvidia-visual-profiler/> for one more trick for showing MPI calls on your timeline.

NVPROF - MPI Profiling

```
$ mpirun -n 4 --gpu nvprof --process-name "MPI Rank %q{OMPI_COMM_WORLD_RANK}"  
--context-name "MPI Rank %q{OMPI_COMM_WORLD_RANK}" -o  
timeline.%q{OMPI_COMM_WORLD_RANK}.nvprof ./laplace2d.solution  
==89073== NVPROF is profiling process 89073, command: ./laplace2d.solution  
==89070== NVPROF is profiling process 89070, command: ./laplace2d.solution  
==89075== NVPROF is profiling process 89075, command: ./laplace2d.solution  
==89066== NVPROF is profiling process 89066, command: ./laplace2d.solution  
<program output>  
==89075== Generated result file: timeline.3.nvprof  
==89073== Generated result file: timeline.2.nvprof  
==89070== Generated result file: timeline.1.nvprof  
==89066== Generated result file: timeline.0.nvprof
```

MULTI-PROCESS PROFILING

When running nvprof with multiple processes, it's useful to label each process:

```
$ nvprof -o timeline_rank%q{OMPI_COMM_WORLD_RANK} \  
    --context-name "MPI Rank %q{OMPI_COMM_WORLD_RANK} \  
    --process-name "MPI Rank %q{OMPI_COMM_WORLD_RANK} \  
    --annotate-mpi openmpi ...
```

PROFILING MPI+CUDA APPLICATIONS

Using `nvprof`+`NVVP`

New since CUDA 9

Embed MPI rank in output filename, process name, and context name (OpenMPI)

```
nvprof --output-profile profile.%q{OMPI_COMM_WORLD_RANK} \
      --process-name "rank %q{OMPI_COMM_WORLD_RANK}" \
      --context-name "rank %q{OMPI_COMM_WORLD_RANK}" \
      --annotate-mpi openmpi
```

Alternatives:

Only save the textual output (`--log-file`)

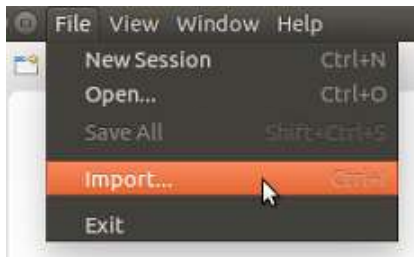
Collect data from all processes that run on a node (`--profile-all-processes`)

```
MVAPICH2: MV2_COMM_WORLD_RANK
      --annotate-mpi mpich
```

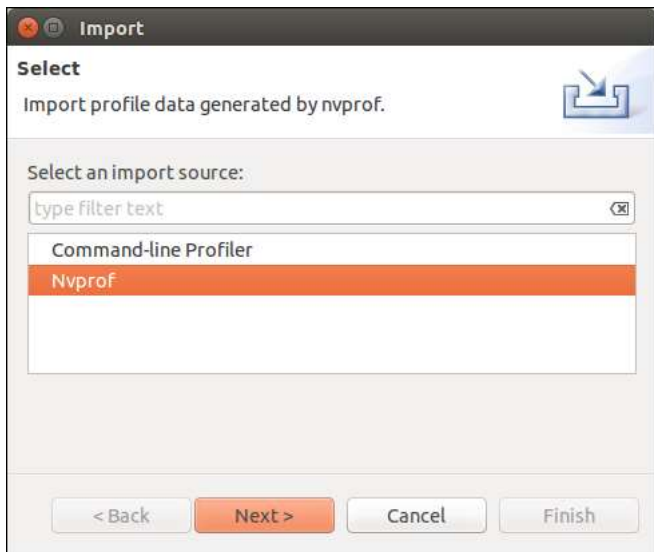
MPI PROFILING

Importing into the Visual Profiler

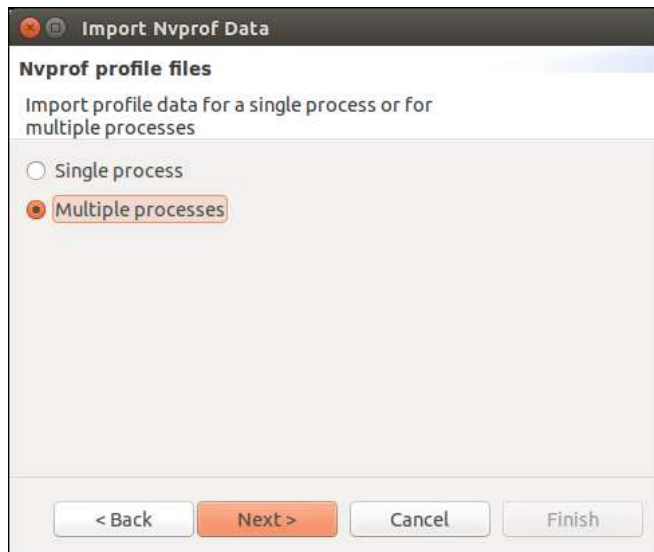
1



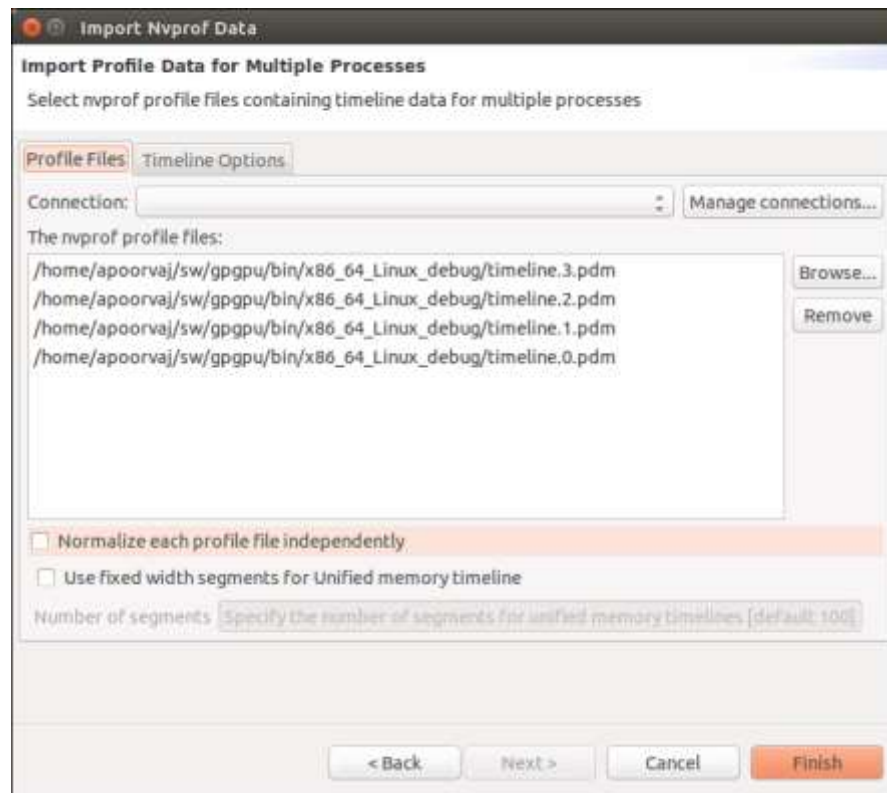
2



3



4



MPI PROFILING

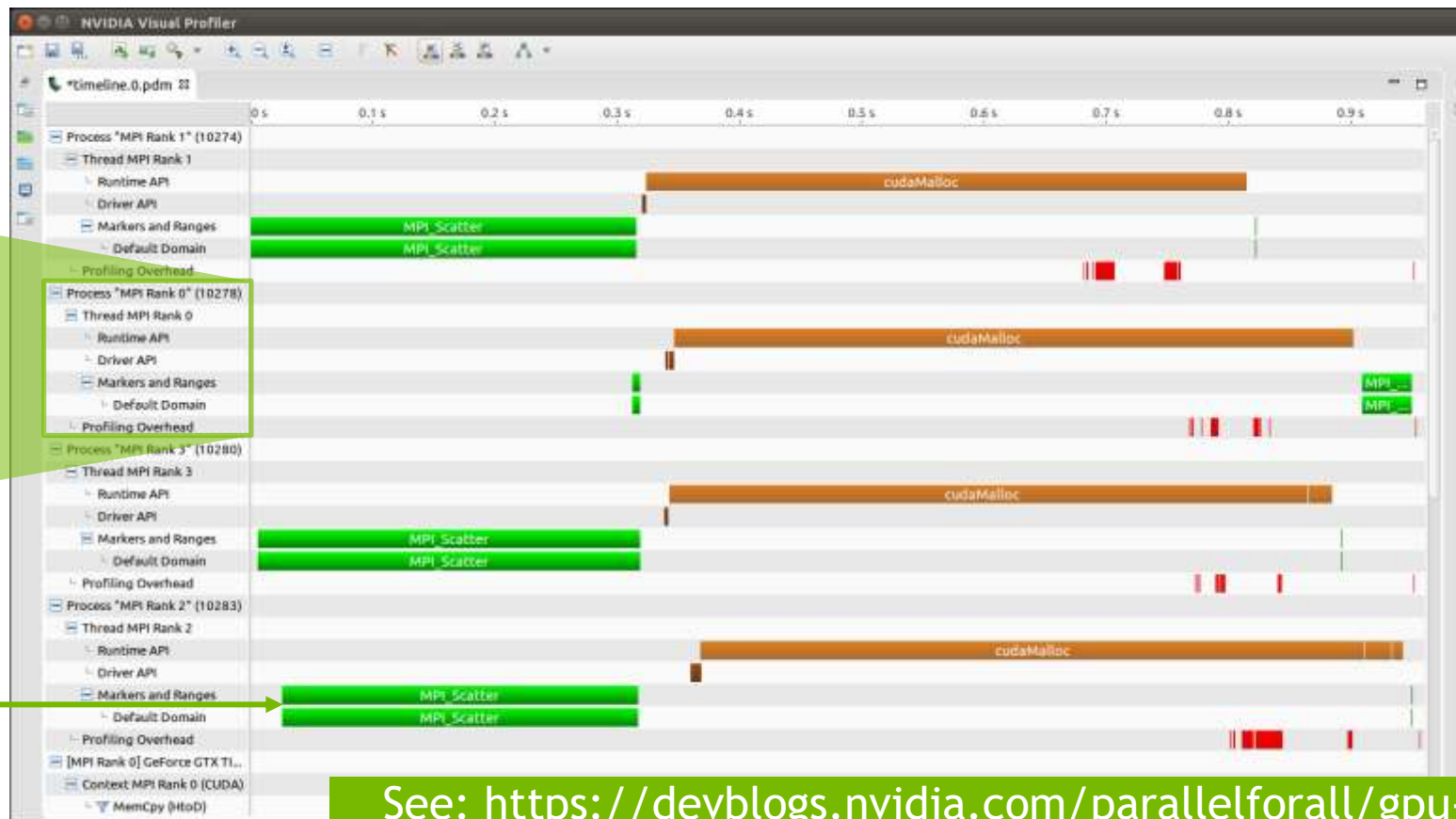
Visual Profiler

MPI Rank-based naming

- Process "MPI Rank 0" (10278)
- Thread MPI Rank 0
 - Runtime API
 - Driver API
 - Markers and Ranges
 - Default Domain
 - Profiling Overhead

- Process "MPI Rank 0" (10278)
- Thread MPI Rank 0
 - Runtime API
 - Driver API
 - Markers and Ranges
 - Default Domain
 - Profiling Overhead

NVTX Markers & Ranges



See: <https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-track-mpi-calls-nvidia-visual-profiler>

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 ...
```

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.

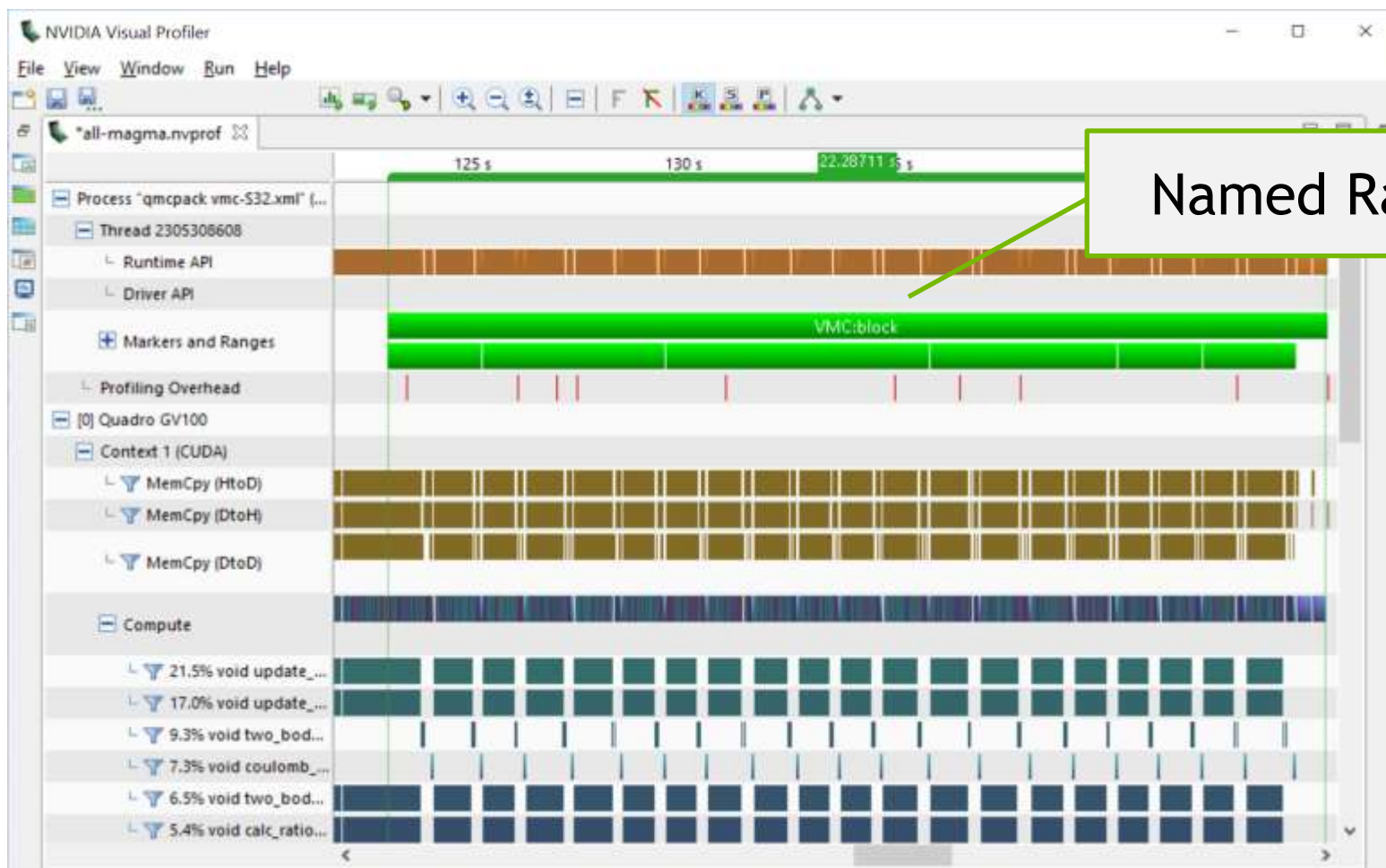
NVTX ANNOTATIONS

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.

NVTX IN VISUAL PROFILER



EXPORTING DATA

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

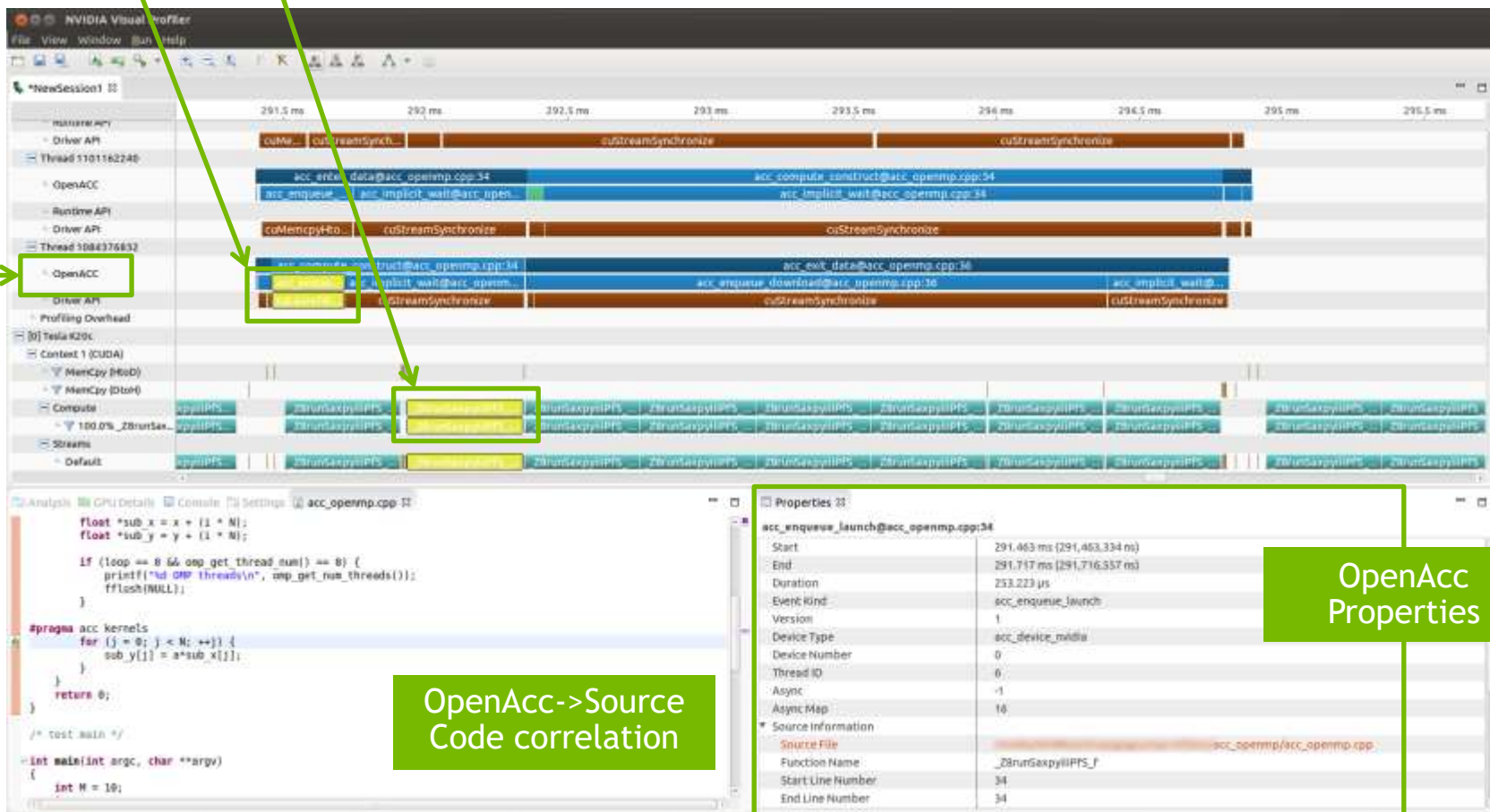
```
$ nvprof --csv --log-file output.csv \  
-i profile.nvprof
```

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

OPENACC PROFILING

OpenAcc->Driver
API->Compute
correlation

OpenAcc
timeline



OpenAcc->Source
Code correlation

OpenAcc
Properties

OPENMP PROFILING

Information about OpenMP regions using the OpenMP tools interface (OMPT) starting CUDA 10.0

Supported on x86_64 and Power Linux with PGI runtime 18.1+

Supported added in the CUPTI, nvprof and Visual Profiler

OPENMP PROFILING IN NVPROF

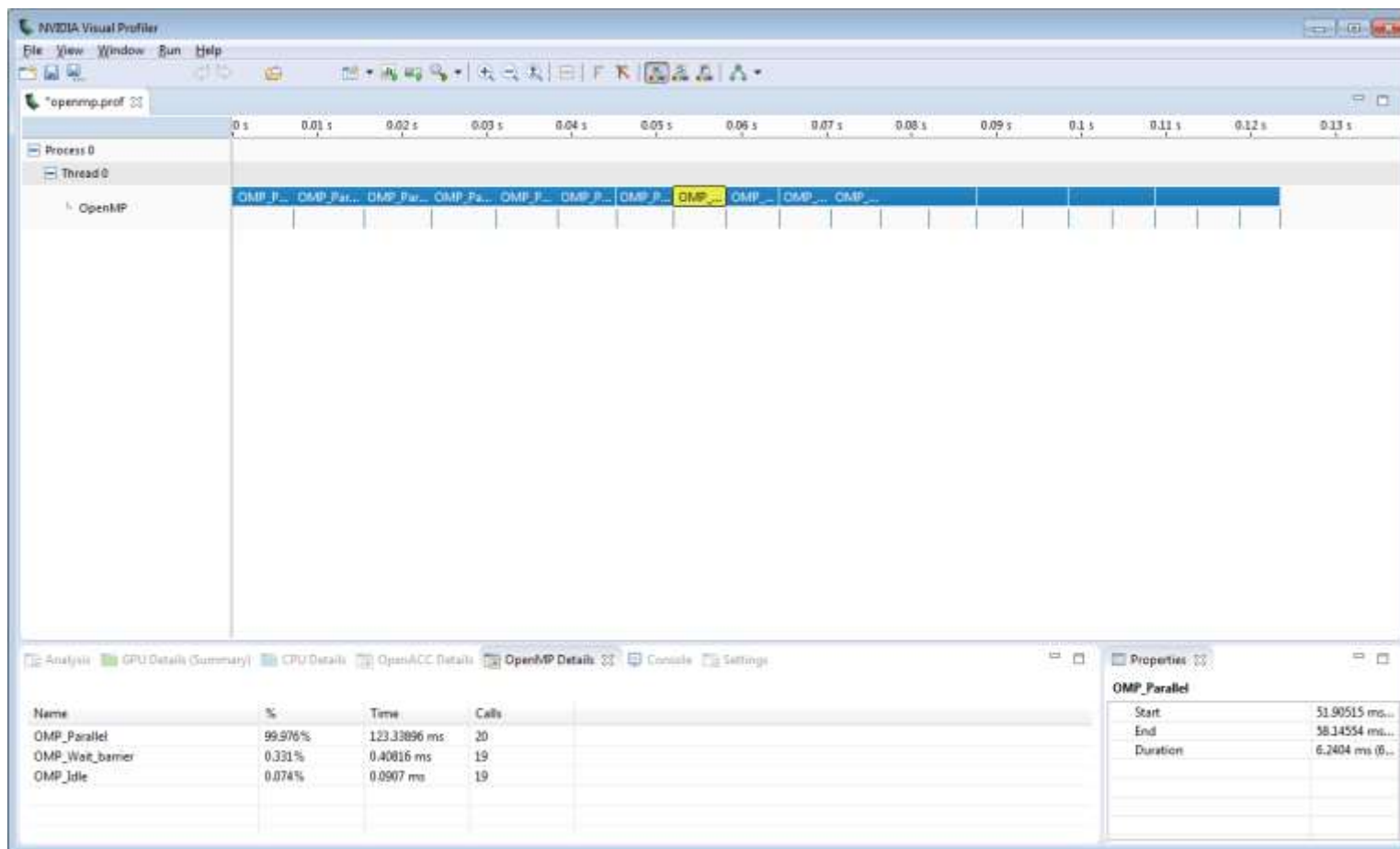
nvprof option openmp-profiling to enable/disable the OpenMP profiling, default on

```
$nvprof openmp-profiling on ./omp-app
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
OpenMP (incl):	99.97%	277.10ms	20	13.855ms	13.131ms	18.151ms	omp_parallel
	0.03%	72.728us	19	3.8270us	2.9840us	9.5610us	omp_idle
	0.00%	7.9170us	7	1.1310us	1.0360us	1.5330us	omp_wait_barrier

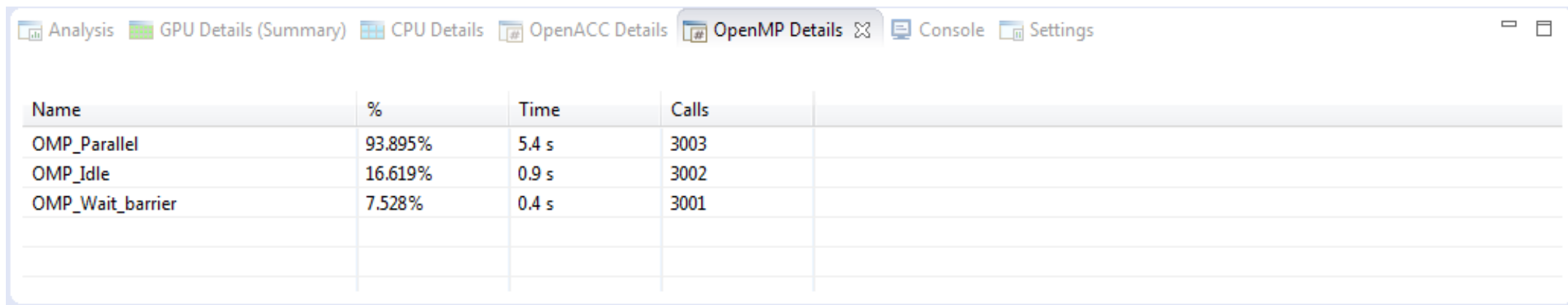
Option --print-openmp-summary to print a summary of all recorded OpenMP activities

OPENMP PROFILING IN VISUAL PROFILER



OPENMP PROFILING IN VISUAL PROFILER

Table View



The screenshot shows the 'OpenMP Details' tab in the Visual Profiler interface. The table displays the following data:

Name	%	Time	Calls
OMP_Parallel	93.895%	5.4 s	3003
OMP_Idle	16.619%	0.9 s	3002
OMP_Wait_barrier	7.528%	0.4 s	3001

PROFILING NVLINK USAGE

Using `nvprof`+`NVVP`

Run `nvprof` multiple times to collect metrics

```
nvprof --output-profile profile.<metric>.%q{OMPI_COMM_WORLD_RANK}\  
--aggregate-mode off --event-collection-mode continuous \  
--metrics <metric> -f
```

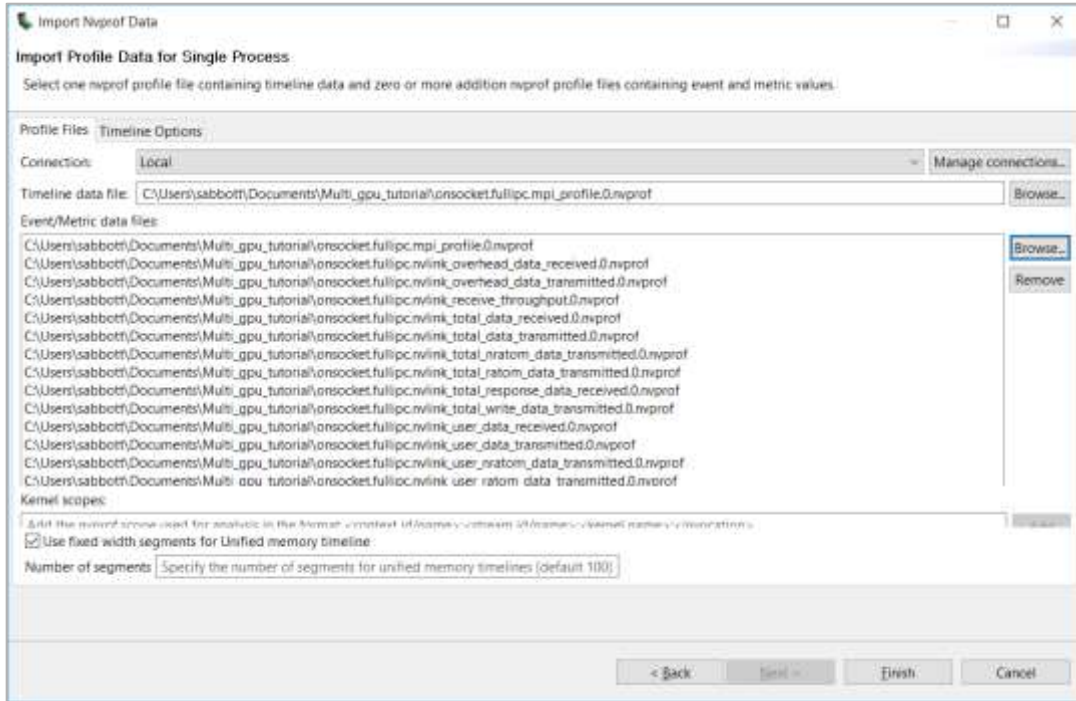
```
nvprof --output-profile profile.<metric>.%q{OMPI_COMM_WORLD_RANK}\  
--aggregate-mode off --event-collection-mode continuous \  
--metrics <metric> -f
```

Use `--query-metrics` and `--query-events` for full list of metrics (-m) or events (-e)

Combine with an MPI annotated timeline file for full picture

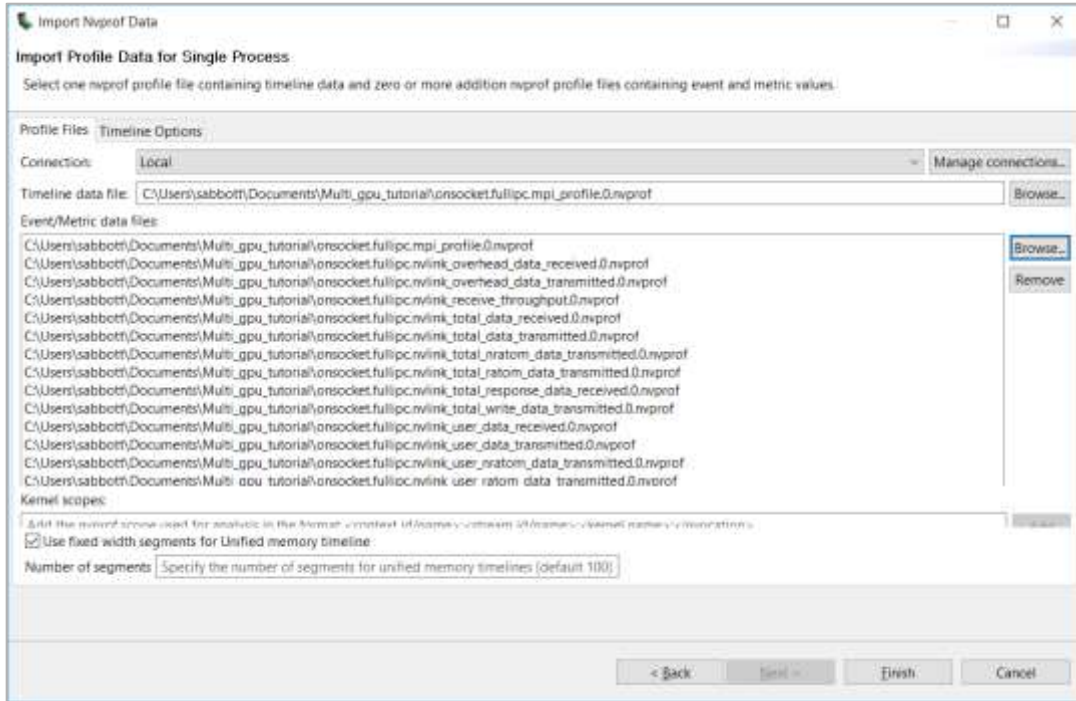
PROFILING NVLINK USAGE

Using `nvprof`+NVVP




PROFILING NVLINK USAGE

Using `nvprof`+NVVP

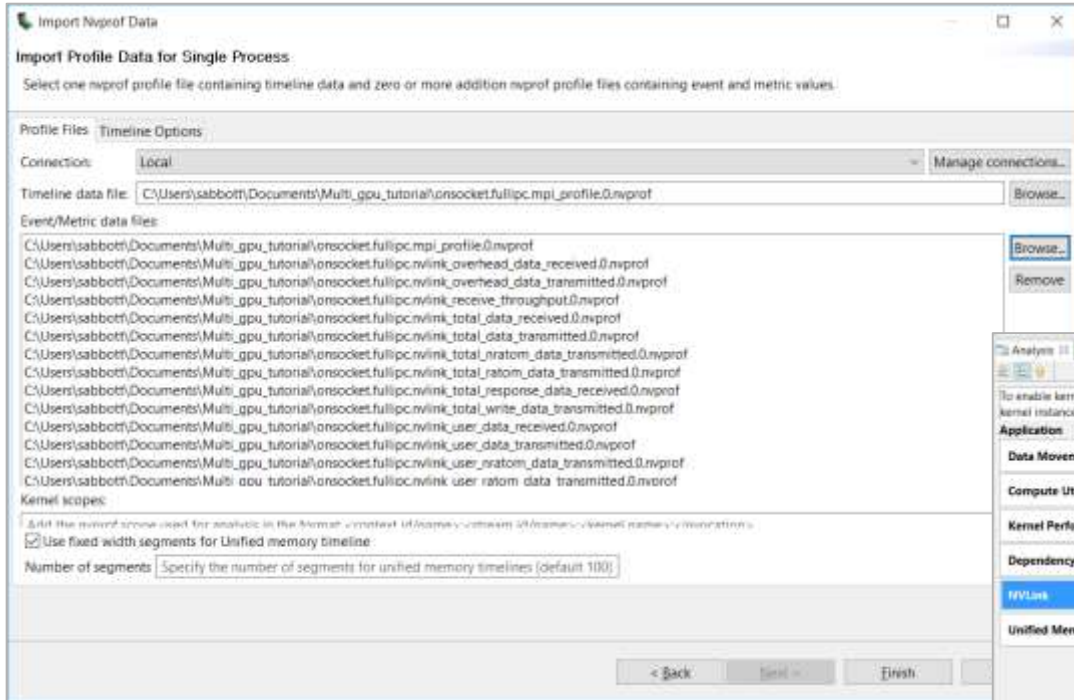


analysis information may be stale and should be deleted before continuing.

 Switch to unguided analysis

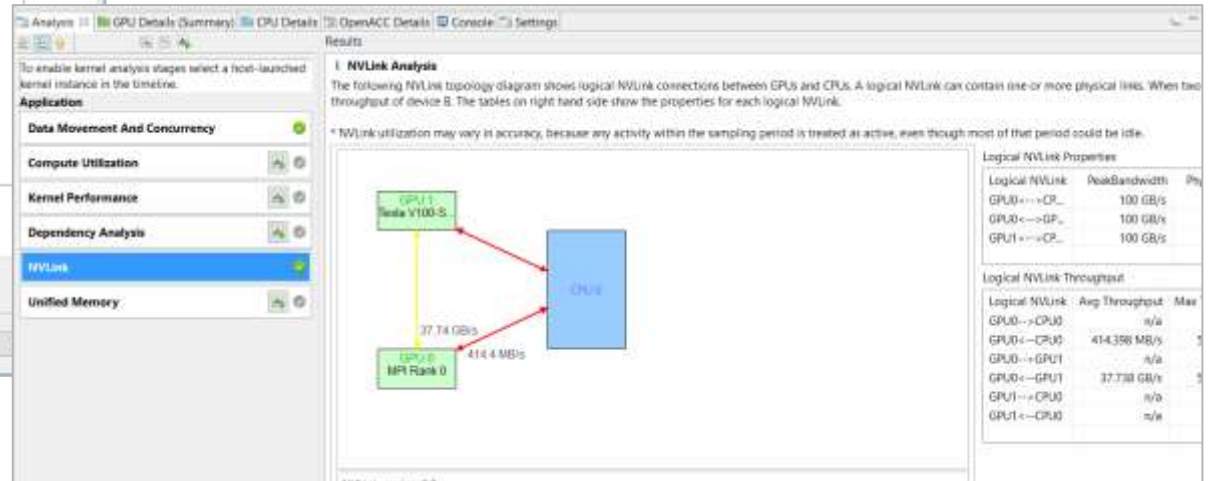
PROFILING NVLINK USAGE

Using `nvprof`+NVVP



analysis information may be stale and should be deleted before continuing.

Switch to unguided analysis



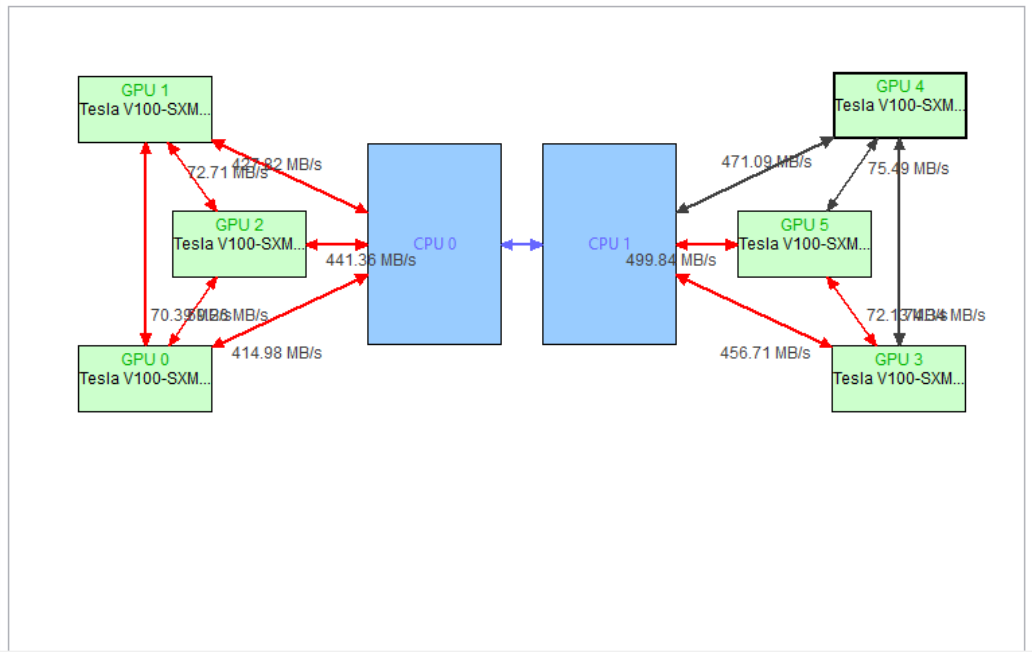
SUMMIT NVLINK TOPOLOGY

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	
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	
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	
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

CPU PAGE FAULT SOURCE CORRELATION

Unguided Analysis

Summary of all CPU page faults

To enable kernel analysis stages select a host-launched kernel instance in the timeline.

Application

- Data Movement And Concurrency ✓
- Compute Utilization ✓
- Kernel Performance ✓
- Dependency Analysis ✓
- NVLink ✓
- Unified Memory** ✓

Results

The following table shows the top locations where CPU page faults occurred (Double-click to open the location in source code)

CPU page faults	Source location
1001	main@jacobi.cu:130
1001	main@jacobi.cu:130
4	Unknown
2	Unknown
1	_Z4initPFS_iis_j@jacobi.cu:85
1	Unknown

The location in source code where the CPU fault occurred

Option to collect Unified Memory information



INTRODUCTION TO NSIGHT SYSTEMS

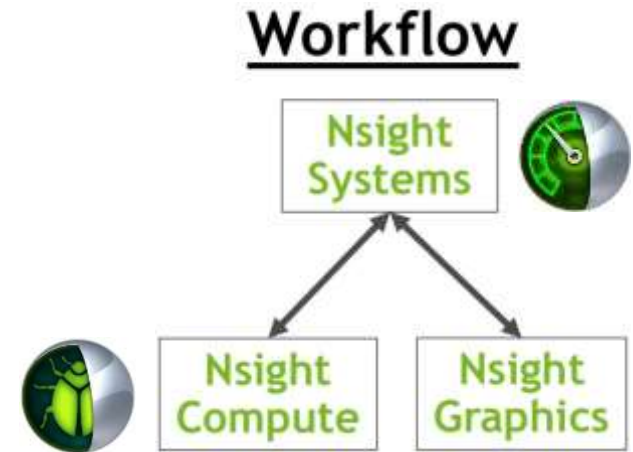
NSIGHT PRODUCT FAMILY

- ▶ Standalone Performance Tools

- ▶ **Nsight Systems** system-wide application algorithm tuning
- ▶ **Nsight Compute** Debug/optimize specific CUDA kernel
- ▶ **Nsight Graphics** Debug/optimize specific graphics

- ▶ IDE plugins

- ▶ **Nsight Eclipse Edicion/Visual Studio** editor, debugger, some perf analysis





NSIGHT SYSTEMS

Overview

- ▶ Profile **System-wide** application
 - ▶ Multi-process tree, GPU workload trace, etc
- ▶ Investigate your workload across multiple CPUs and GPUs
 - ▶ CPU algorithms, utilization, and thread states
 - ▶ GPU streams kernels, memory transfers, etc
 - ▶ NVTX, CUDA & Library API, etc
- ▶ Ready for Big Data
 - ▶ docker, user privilege (linux), cli, etc





Thread/core migration

Processes and threads

Thread state

CUDA and OpenGL API trace

cuDNN and cuBLAS trace

Kernel and memory transfer activities

Multi-GPU

CPU (80)

Threads (78)

✓ [1221583] Caffe2F -

NVTX

CUDA API

NVTX Tracing

✓ [1221585] Caffe2F -

NVTX

CUDA API

✓ [1221589] Caffe2F -

NVTX

CUDA API

✓ [1221587] Caffe2F -

✓ [1221582] Caffe2F -

73 threads hidden...

CUDA (Tesla V100-SXM2)

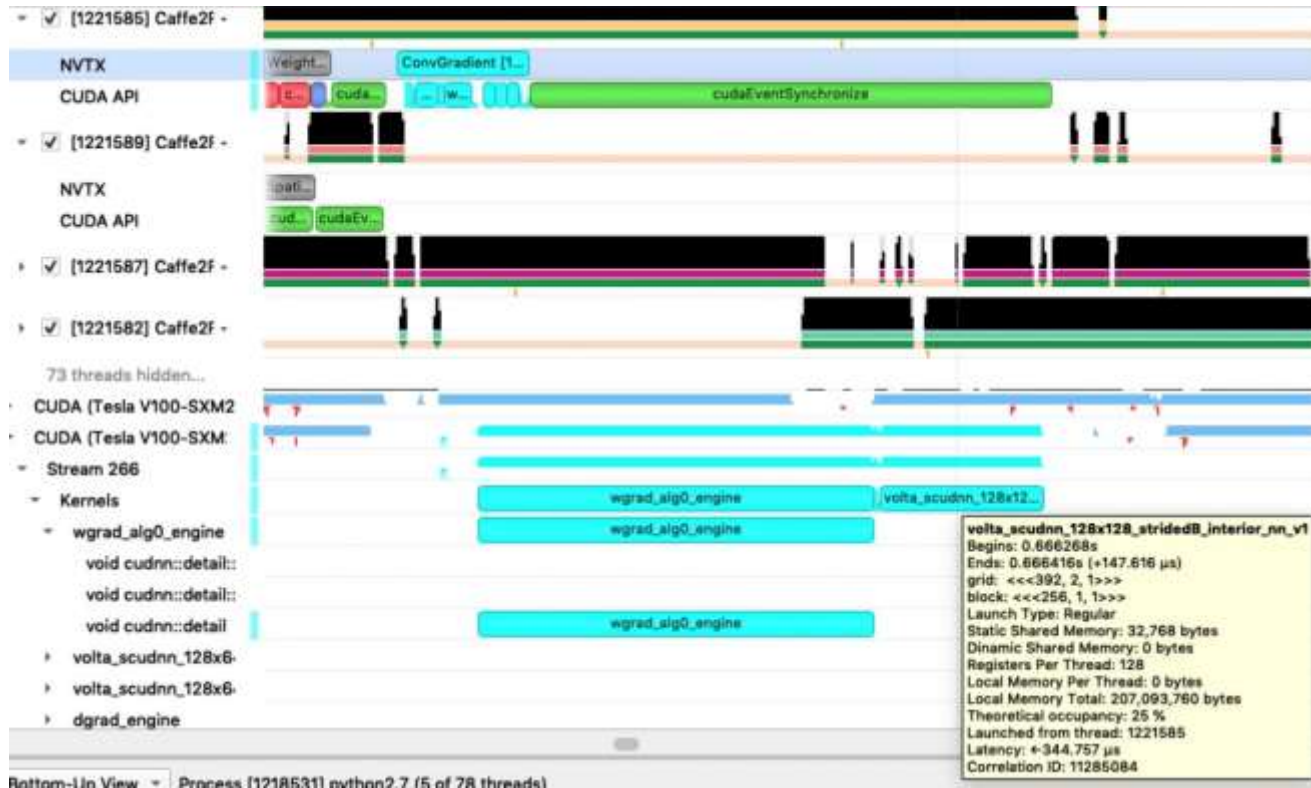
CUDA (Tesla V100-S)



■ CUDA Kernel running
Time: 0.666129s

TRANSITIONING TO PROFILE A KERNEL

Dive into kernel analysis



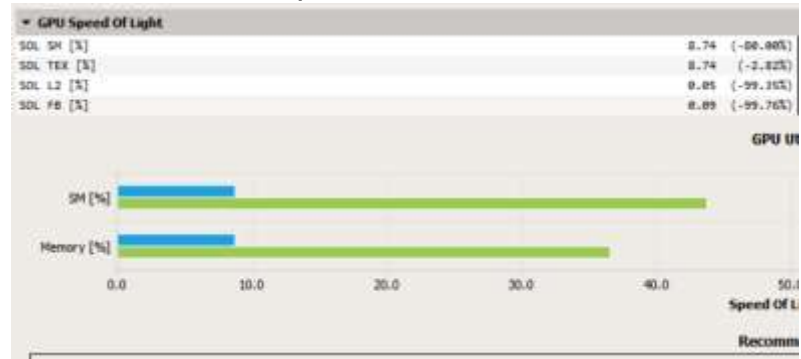


NVIDIA NSIGHT COMPUTE

Next Generation Kernel Profiler

- ▶ Interactive **CUDA API debugging** and **kernel profiling**
- ▶ **Fast** Data Collection
- ▶ Graphical and multiple kernel comparison reports
- ▶ Improved Workflow and Fully Customizable (Baselining, Programmable UI/Rules)
- ▶ Command Line, Standalone, IDE Integration
- ▶ Platform Support
 - ▶ OS: Linux(x86,ARM), Windows, OSX (host only)
 - ▶ GPUs: Pascal, Volta, Turing

Kernel Profile Comparisons with Baseline



Metric Data

Metric	Value 1	Value 2	Value 3	Value 4
inst_executed [inst]	16,528,000	16,528,000	17,476,000	11,476,000
l1tex_acc_pct [%]	14.33	n/a	n/a	n/a
launch_block_size	128.00	128.00	128.00	128.00
launch_function_pcs	47,411,687,948.00	12,273,708.00		
launch_grid_size	4,132.00	2,389.00		
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	2,360.00	2,360.00		
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]	528,496.00	411,232.00		
launch_warps_per_multiprocessor	5.23	42.11		
l1c_acc_pct [%]	6.93	7.18		
memory_access_size_type [bytes]	1.00; 32.00; 32.00; 32.00	2.00; 32.00; 32.00; 32.00		

Source Correlation

Source	Live Registers	Sampling Data (All)	Sampling Data (No Issue)
@!PT SHFL.IDX PT, R2, R2, R2, R2;	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
@!PE EXIT;	2	255	86
MOV R2, R0;	3	386	29
@!PT SHFL.IDX PT, R2, R2, R2, R2;	2	0	0
MOV R-, 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, 0x000767B0;	1	0	0
SHF.R.S32.HI R0, R2, 0x1f, R2;	4	0	0

PROFILING GPU APPLICATION

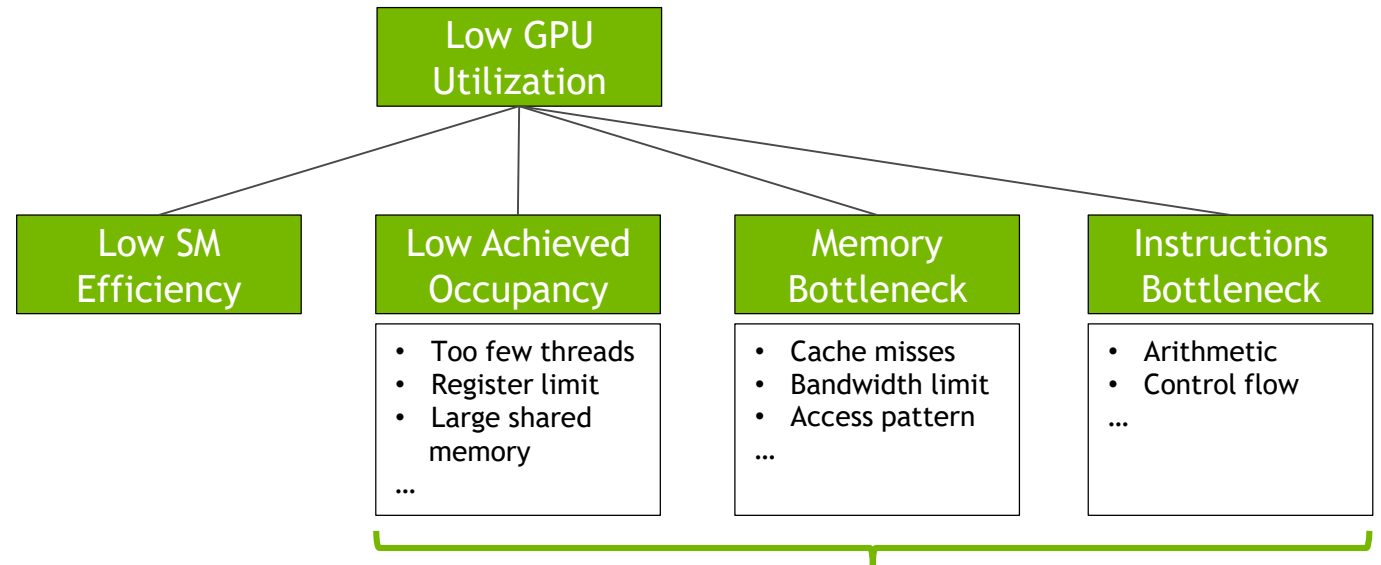
Focusing GPU Computing

How to measure

GPU Profiling

CPU/GPU
Tracing

Application
Tracing



NVIDIA (Visual) Profiler / Nsight Compute

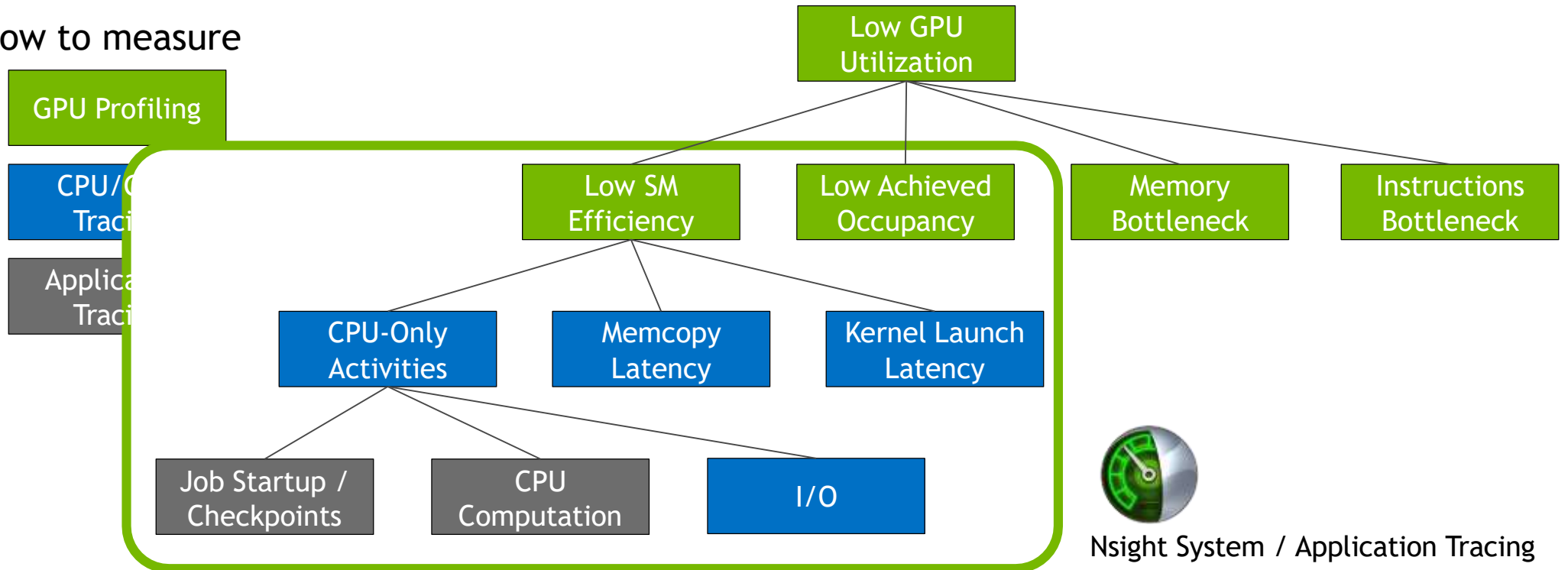


NVIDIA Supports them with cuDNN, cuBLAS, and so on

PROFILING GPU APPLICATION

Focusing System Operation

How to measure



An abstract network diagram with a dark blue background. It features numerous bright green nodes of varying sizes, connected by thin, light green lines. The nodes are scattered across the frame, with a higher density on the left side. Some nodes are larger and more prominent, while others are smaller and less distinct. The overall effect is that of a complex, interconnected system or network.

HOW TO USE

NSIGHT SYSTEMS PROFILE

Profile with CLI

APIs to be traced

```
$ nsys profile -t cuda,osrt,nvtx,cudnn,cublas \  
-o baseline.qdstrm -w true python main.py
```

Name of
output file

Show output
on console

Application
command

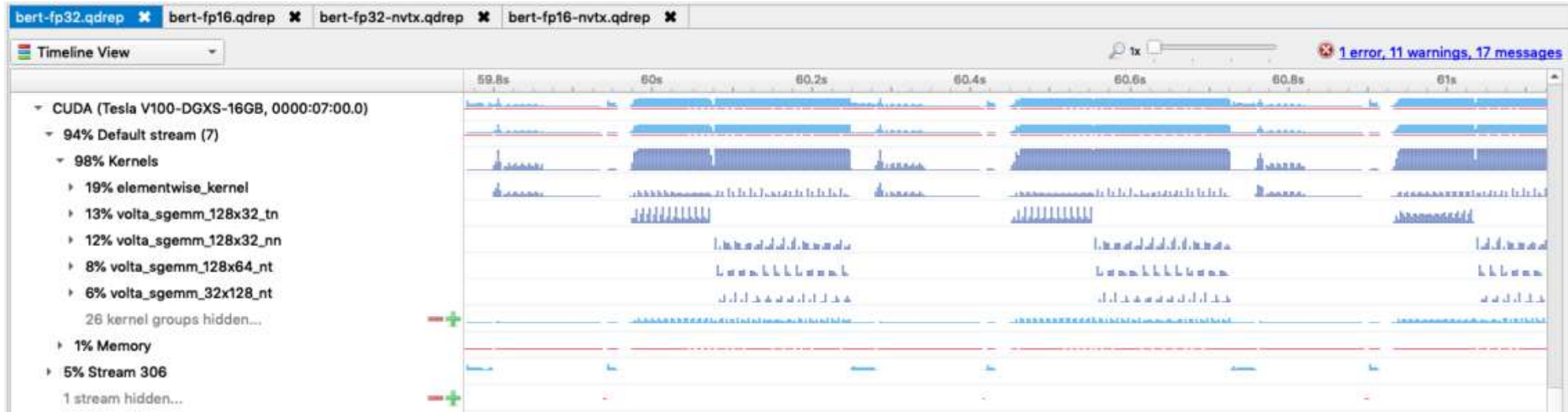
Automatic conversion of .qdstrm temp results file to .qdrep format if converter utility is available.

cuda - GPU kernel
osrt - OS runtime
nvtx - NVIDIA Tools Extension
cudnn - CUDA Deep NN library
cublas - CUDA BLAS library

https://docs.nvidia.com/nsight-systems/#nsight_systems/2019.3.6-x86/06-cli-profiling.htm

NSIGHT SYSTEMS PROFILE

No NVTX



- ▶ Difficult to understand → no useful

An abstract visualization of a network or data flow. It features a dark blue background with numerous thin, light green lines connecting various points. These points are represented by small, bright green circular nodes of varying sizes. The lines and nodes are scattered across the frame, creating a complex, interconnected web. The overall effect is that of a digital or neural network structure.

NVTX (NVIDIA TOOLS EXTENSION)

NVTX ANNOTATIONS

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):

        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
```

NVTX ANNOTATIONS

NVTX in PyTorch

```
import torch.cuda.nvtx as nvtx
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        nvtx.range_push("Batch " + str(batch_idx))
        nvtx.range_push("Copy to device")
        data, target = data.to(device), target.to(device)
        nvtx.range_pop()
        nvtx.range_push("Forward pass")
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        nvtx.range_pop()
        nvtx.range_pop()
```

Batch %d



copy to device



Forward pass

NVTX ANNOTATIONS

NVTX using cupy package

```
from cupy.cuda import nvtx
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        nvtx.RangePush("Batch " + str(batch_idx))
        nvtx.RangePush("Copy to device")
        data, target = data.to(device), target.to(device)
        nvtx.RangePop()
        nvtx.RangePush("Forward pass")
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        nvtx.RangePop()
        nvtx.RangePop()
```

Batch %d



copy to device



Forward pass

NSIGHT SYSTEM PROFILE

NVTX range marker tip

- ▶ NVTX for data loading (data augmentation)

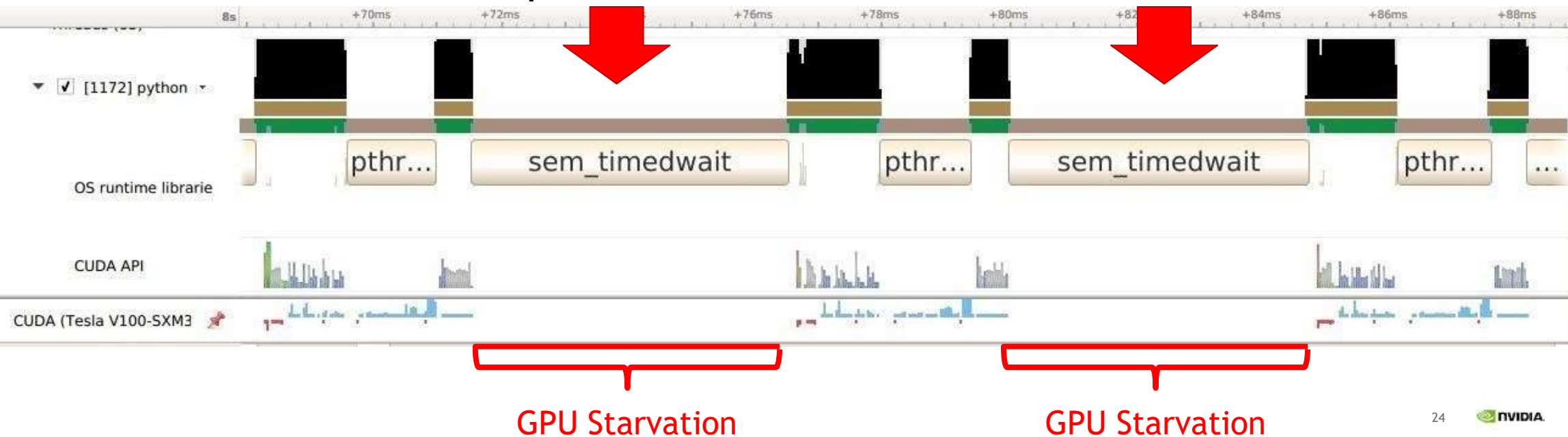
```
for batch_idx, (data,target) in enumerate(train_loader):
```



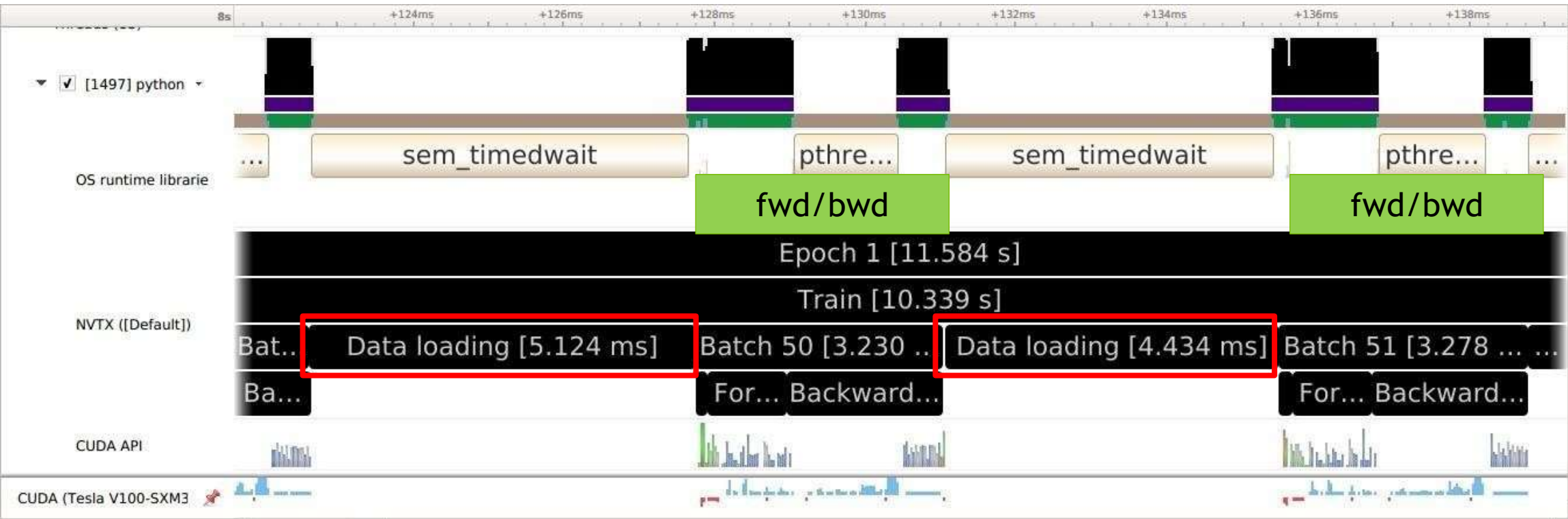
```
nvtx.range_push('Data loading')  
(data,target) = train_loader.next()  
nvtx.range_pop()
```

BASELINE PROFILE

- ▶ MNIST Training: 89 sec, <5% utilization
- ▶ CPU waits on a semaphore and starves the GPU!



BASELINE PROFILE (WITH NVTX)



- ▶ GPU is idle during **data loading**
- ▶ Data is loaded using a single thread. This starves the GPU!

5.1ms

OPTIMIZE SOURCE CODE

- ▶ Data loader was configured to use 1 worker thread

```
kwargs = {'num_workers': 1, 'pin_memory': True if use_cuda else {}}
```

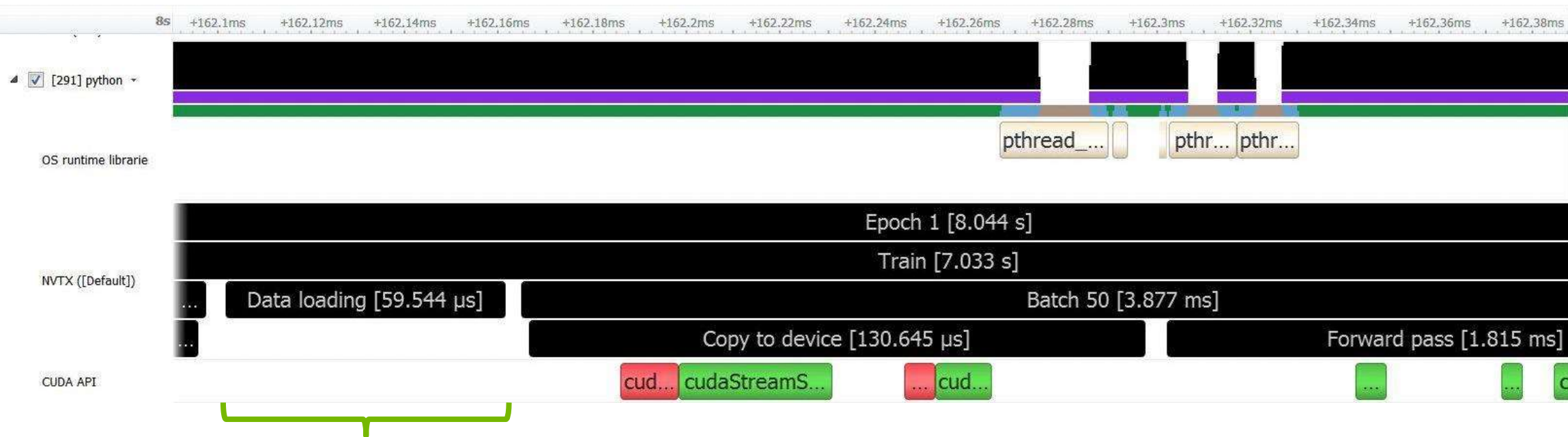


- ▶ Let's switch to using 8 worker threads:

```
kwargs = {'num_workers': 8, 'pin_memory': True if use_cuda else {}}
```

AFTER OPTIMIZATION

- ▶ Time for data loading reduced for each bath



60us

Reduced from 5.1ms to 60us for batch 50





CASE STUDY: OPENACC SAMPLE


OPENACC SAMPLE

- [Sample from https://devblogs.nvidia.com/getting-started-openacc](https://devblogs.nvidia.com/getting-started-openacc)
- Solves 2-D Laplace equation with iterative Jacobi solver
- Each iteration
 1. A stencil calculation
 2. Update the matrix
 3. Check if error tolerance is met. If not, go to step 1.


SAMPLE (CPU VERSION)


```
while ( error > tol && iter < iter_max ) {  Convergence loop
    error = 0.0;
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```


 Stencil calculation

 Update matrix

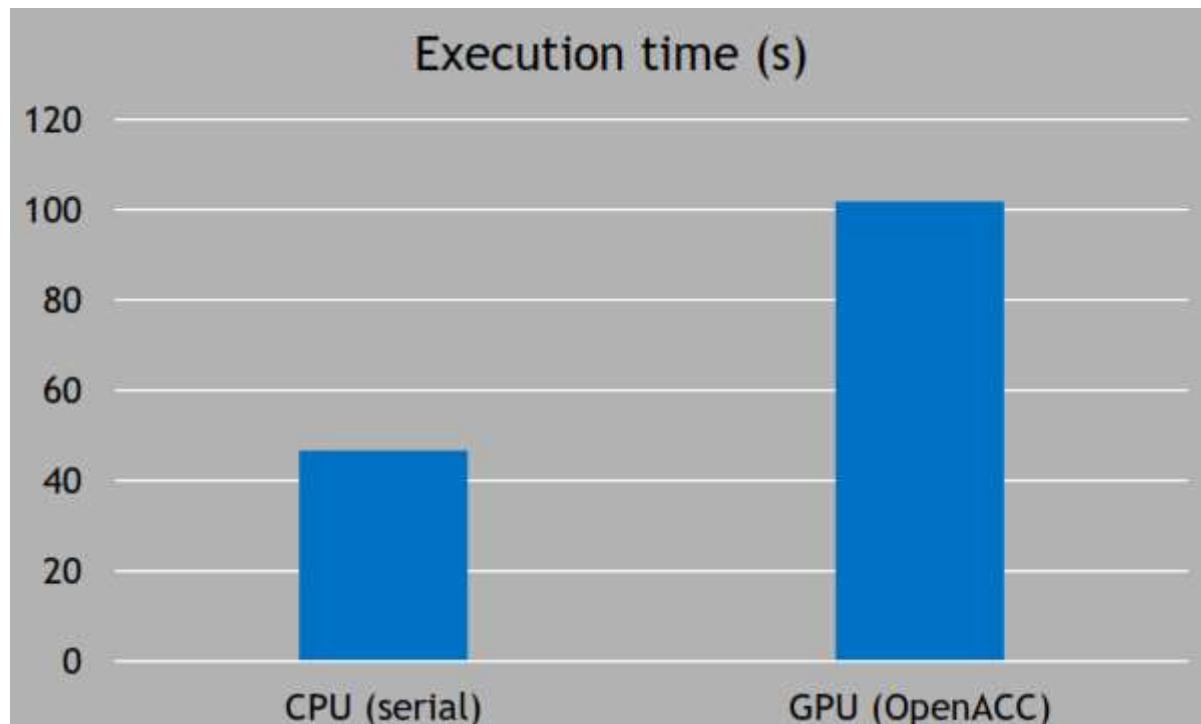
OPENACC SAMPLE

```
while ( error > tol && iter < iter_max ) {  Convergence loop
    error = 0.0;
    #pragma acc kernels
    {
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                Anew[j][i] = ...
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }
    iter++;
}
```

 Stencil calculation

 Update matrix

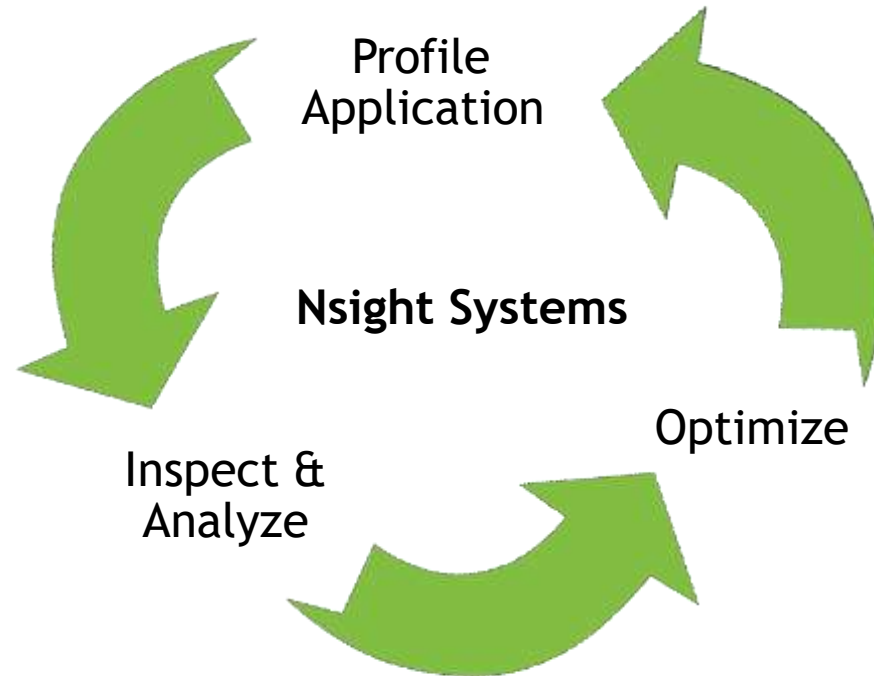
PERFORMANCE



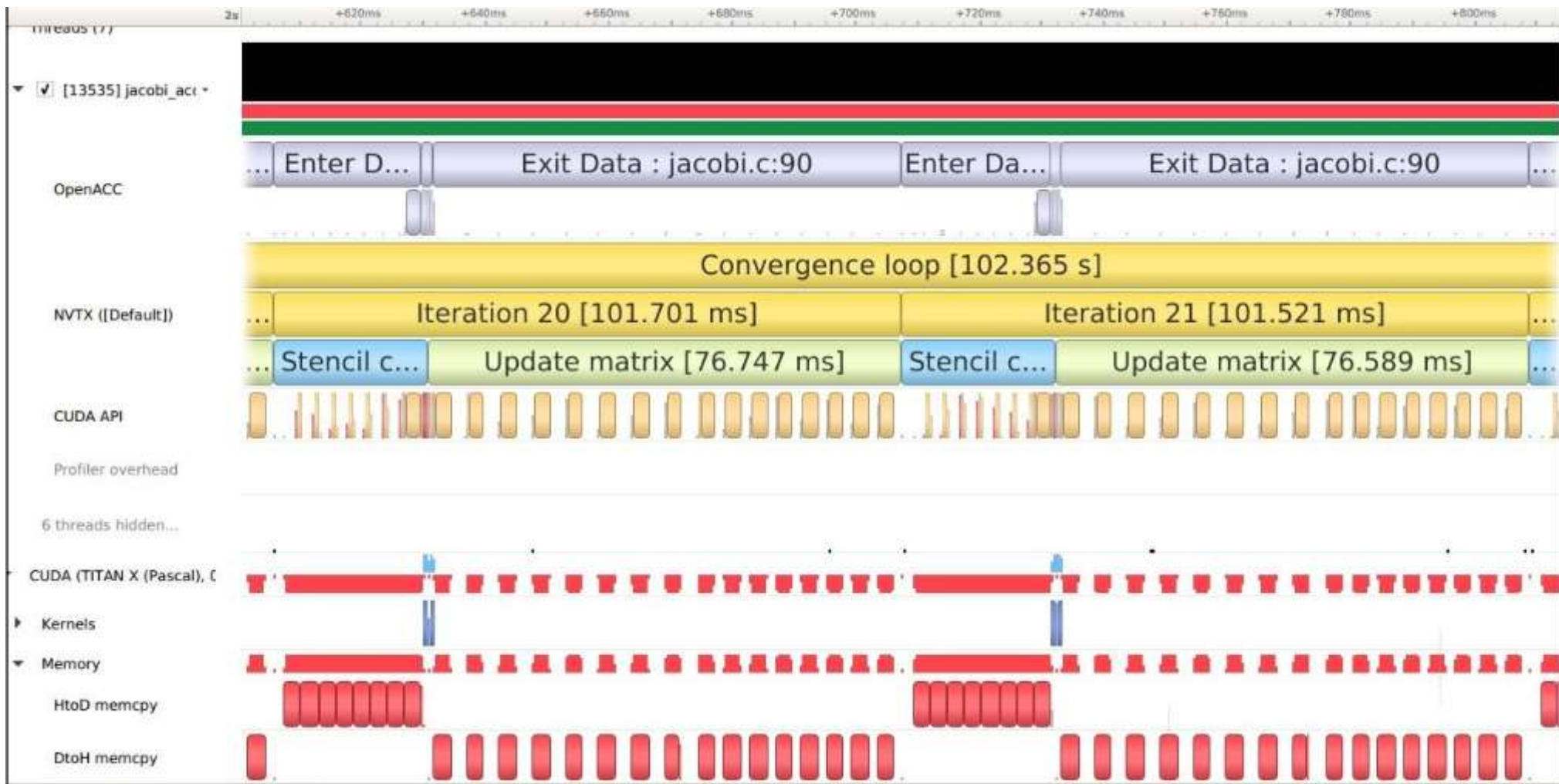
Execution time for 1000 iterations on a system with:
Intel® Core™ i7-6850K CPU
NVIDIA TITAN X (Pascal) GPU

That is unexpected!

OPTIMIZATION WORKFLOW





BASELINE PROFILE




Excessive data copies slowing down GPU


OPENACC SAMPLE


```
while ( error > tol && iter < iter_max ) {  Convergence loop
    error = 0.0;
    #pragma acc kernels
    {
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                Anew[j][i] = ...
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }
    iter++;
}
```


 Stencil calculation

 Update matrix

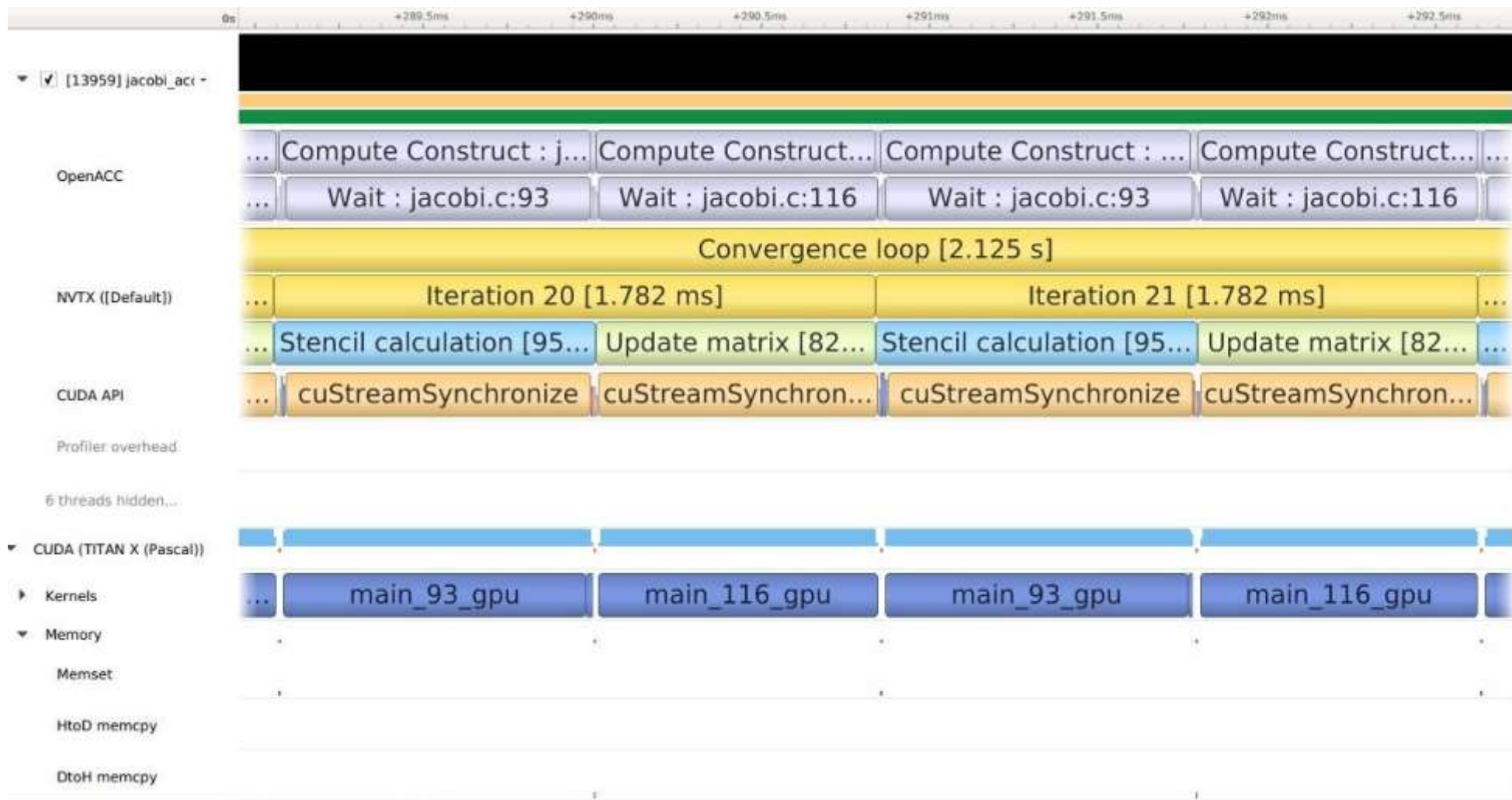
OPENACC SAMPLE

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {  Convergence loop
    error = 0.0;
    #pragma acc kernels
    {
        for( int j = 1; j < n-1; j++ ) {
            for( int i = 1; i < m-1; i++ ) {
                Anew[j][i] = ...
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
        for( int j = 1; j < n-1; j++ ) {
            for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }
    iter++;
}
```

 Stencil calculation

 Update matrix

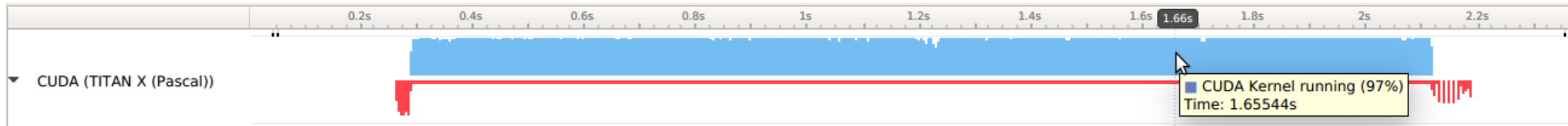
AFTER OPTIMIZATION



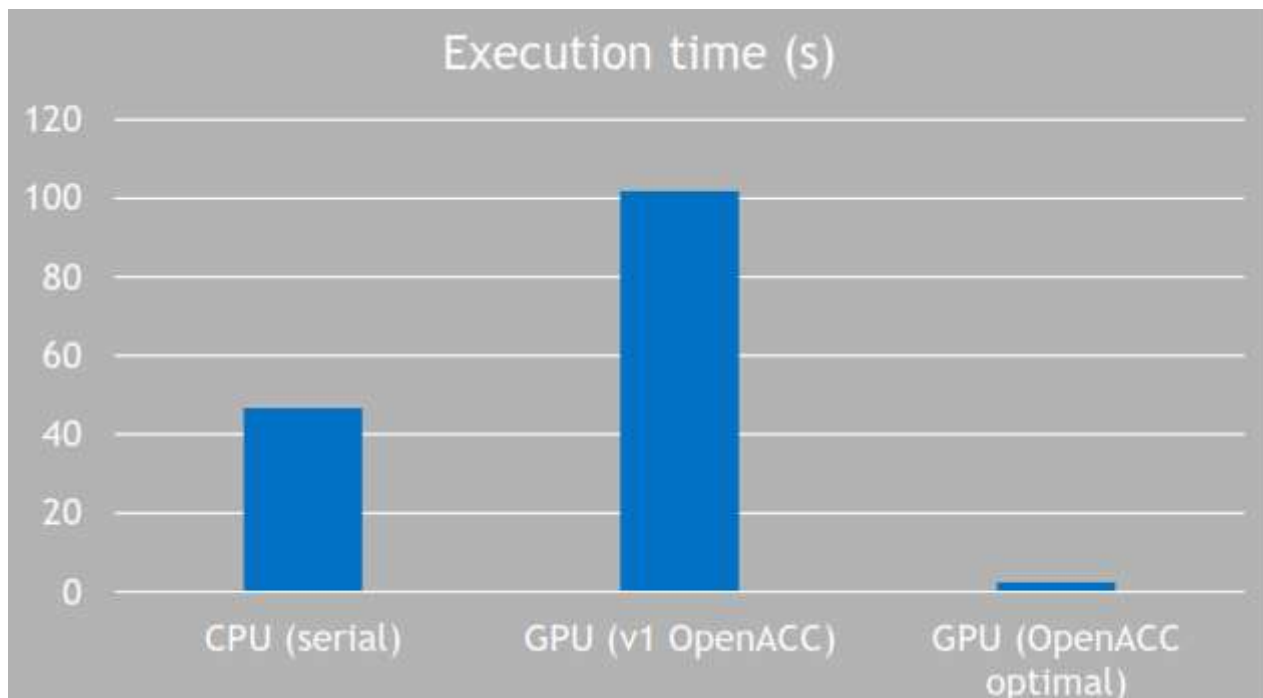
Excessive data copies eliminated

AFTER OPTIMIZATION

CUDA kernel coverage on GPU is ~97%



AFTER OPTIMIZATION



Execution time for 1000 iterations on a system with:
Intel® Core™ i7-6850K CPU
NVIDIA TITAN X (Pascal) GPU

44x speedup!

<https://www.openacc.org/resources> for more best practices

COMMON OPTIMIZATION OPPORTUNITIES

▶ CPU

- Thread synchronization
- Algorithm bottlenecks starve the GPUs (case study 1)

▶ Multi GPU

- Communication between GPUs
Lack of Stream Overlap in memory management, kernel execution

▶ Single GPU

- Memory operations - blocking, serial, unnecessary (case study 2)
- Too much synchronization - device, context, stream, default stream, implicit
- CPU GPU Overlap - avoid excessive communication