GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

OPTIMIZING APPLICATION PERFORMANCE WITH CUDA® PROFILING TOOLS

Swapna Matwankar, April 7, 2016

PRESENTED BY

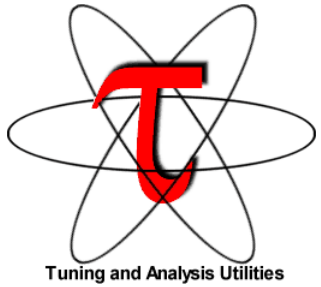
NVIDIA.

CUDA PROFILING TOOLS

- NVIDIA® Visual Profiler
  - Standalone (nvvp)
  - Integrated into NVIDIA® Nsight™ Eclipse Edition (nsight)
- nvprof *
- NVIDIA® Nsight™ Visual Studio Edition
- Old environment variable based command-line profiler is discontinued from 8.0.

* Android CUDA APK profiling not supported (yet)

3RD PARTY PROFILING TOOLS



TAU Performance System ®

VampirTrace

PAPI CUDA Component

HPC Toolkit

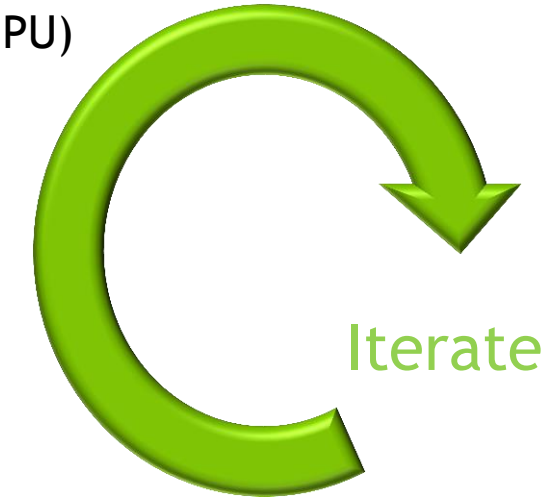
PERFORMANCE OPPORTUNITIES

Application level opportunities

- Overall application performance
  - Overlap CPU and GPU work, identify the bottlenecks (CPU or GPU)
- Overall GPU utilization and efficiency
  - Overlap compute and memory copies
  - Utilize compute and copy engines effectively

Kernel level opportunities

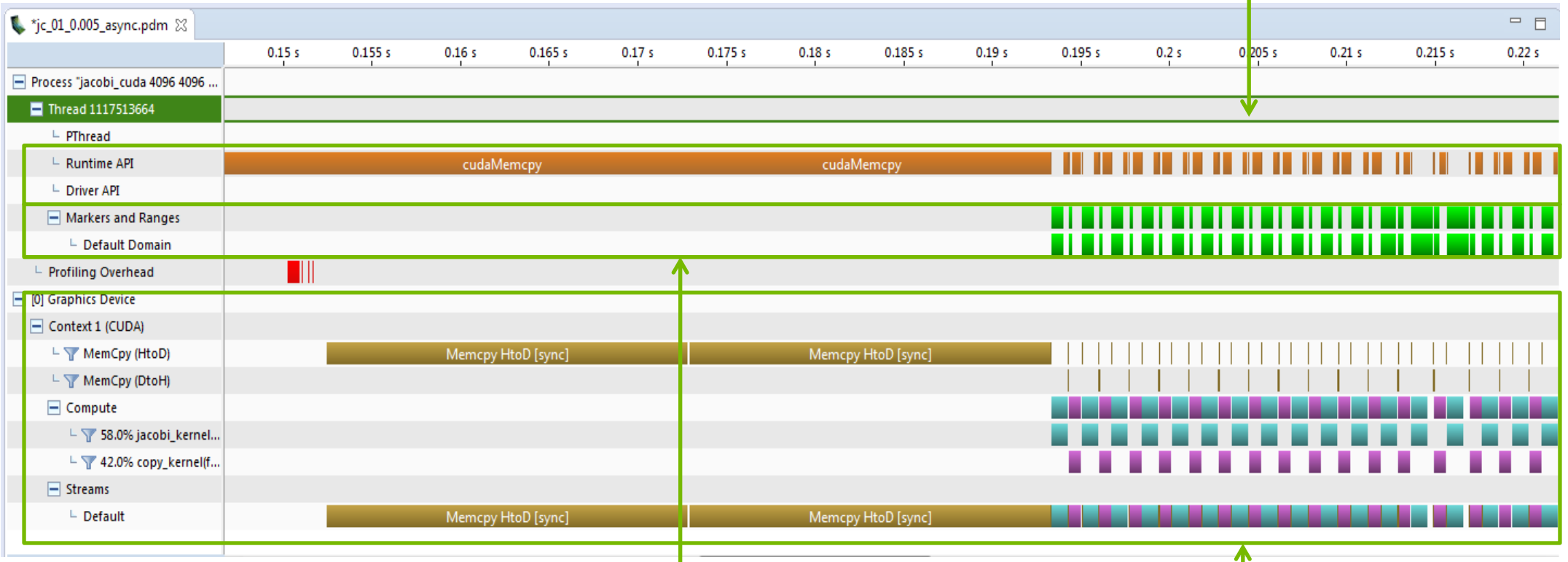
- Use memory bandwidth efficiently
- Use compute resources efficiently
- Hide instruction and memory latency



PERFORMANCE OPPORTUNITIES

Application level

API invocation



NVTX markers and ranges

GPU activities

PERFORMANCE OPPORTUNITIES

Kernel level

Guided Analysis

Analysis | GPU Details | CPU Details | Console | Settings

Export PDF Report

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 "ComputeBoundFp32" 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.

Analysis | GPU Details | CPU Details | Console | Settings

Reset All | Analyze All

ComputeBoundFp32(int*, int*, int*, int)

- Kernel Performance Limiter
- Kernel Latency
- Kernel Compute
- Kernel Memory
- Global Memory Access Pattern
- Shared Memory Access Pattern
- Divergent Execution
- Kernel Profile - Instruction Execution
- Kernel Profile - PC Sampling

Application

- Data Movement And Concurrency
- Compute Utilization
- Kernel Performance
- Dependency Analysis
- NVLink

Unguided Analysis

What's new in 8.0?

- Dependency Analysis
- NVLink Analysis
- Unified memory profiling
- Instruction Level Profiling (PC sampling)
- Combined source-assembly view
- FP16 Analysis
- OpenAcc on Timeline
- CPU profiling
- Nvidia Tools Extension V2

Features listed in green are Pascal specific features

DEPENDENCY ANALYSIS

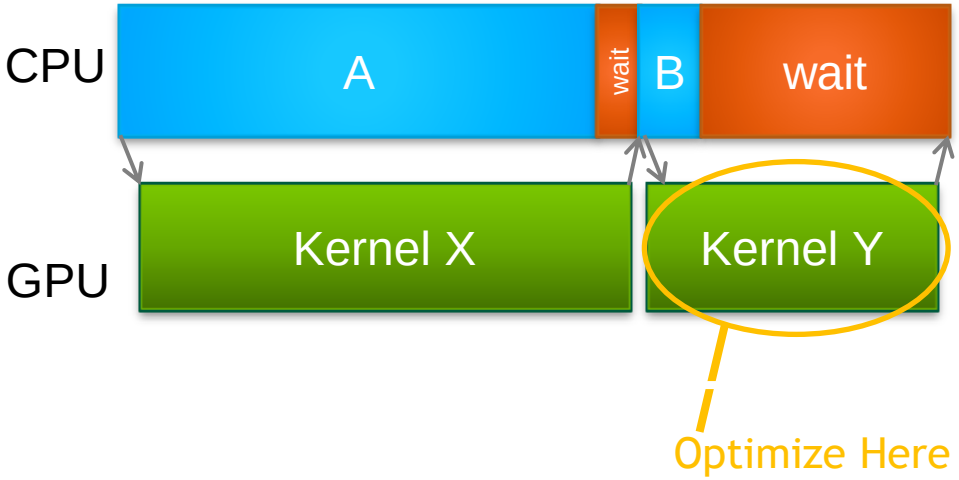
DEPENDENCY ANALYSIS

Motivation

Not always

- **GPU kernels are bottleneck in application**
- **Optimizing kernel taking highest time will give more performance benefits**

It is important to identify right bottlenecks in the application to get good ROI



DEPENDENCY ANALYSIS

How is it done?

In 8.0, profiling tools supports identifying critical path in the application

- Analyzes CPU threads (POSIX) and GPU activities
- Graph is generated by post-processing execution traces of application (negligible execution overhead)
- Dependencies are defined by CUDA API contract

DEPENDENCY ANALYSIS

Results

- Critical path that includes CUDA APIs, GPU activities, thread activities
- For all CUDA APIs, GPU activities and thread activities
  - Time on Critical Path – Optimizing this will improve overall execution time
  - Waiting time – Reducing waiting time will improve load imbalance
  - Inbound/outbound dependencies – To traverse the issues in both directions

DEPENDENCY ANALYSIS

nvprof

Command: `./nvprof --dependency-analysis --cpu-thread-tracing on ./jacobi_cuda 4096 4096 0.005`

Output:

```
==13269== Dependency Analysis:
==13269== Analysis progress: 100%
Critical path(%)  Critical path  Waiting time  Name
          70.99%   994.926674ms           0ns  <Other>
          13.95%   195.570089ms           0ns  cudaMalloc
           5.69%    79.785085ms           0ns  jacobi_kernel(float const *, float*, int, int, float*)
           4.71%    66.064614ms           0ns  copy_kernel(float*, float const *, int, int)
           2.87%    40.197672ms   19.969822ms  cudaMemcpy
           1.42%    19.969822ms           0ns  [CUDA memcpy DtoH]
```

Note: `--cpu-thread-tracing on` option is required only for multithreaded applications

DEPENDENCY ANALYSIS

nvprof

Use `--print-dependency-analysis-trace` argument along with `--dependency-analysis` to get the time on critical path and waiting time of each instance of a function

Command: `./nvprof` **`--print-dependency-analysis-trace --dependency-analysis --cpu-thread-tracing on`** `./jacobi_cuda 4096 4096 0.005`

```
==6012== Dependency Analysis:
==6012== Analysis progress: 100%
   Start  Duration  Critical path  Waiting time  Name
118.84ms  1.6540us     1.654000us           0ns  cuDeviceGetCount
118.85ms     350ns          350ns           0ns  cuDeviceGetCount
118.85ms     265ns          265ns           0ns  cuDeviceGet
118.85ms     506ns          506ns           0ns  cuDeviceGetAttribute
118.86ms     487ns          487ns           0ns  cuDeviceGet
118.87ms     383ns          383ns           0ns  cuDeviceGetAttribute
```

```
430.86ms  9.2270us            0ns           0ns  cudaLaunch
430.86ms  2.3360us     2.336000us           0ns  [CUDA memcpy HtoD]
430.87ms  819.60us            0ns    1.696000us  cudaMemcpy
430.88ms  796.50us   796.500000us           0ns  jacobi_kernel(float const *, float*, int, int, float*)
431.68ms  1.6960us     1.696000us           0ns  [CUDA memcpy DtoH]
438.51ms  10.439us            0ns           0ns  cudaMemcpy
438.52ms  43.980us            0ns    1.696000us  cudaMemcpy
438.53ms  11.775us    11.775000us           0ns  [CUDA memcpy HtoD]
```

DEPENDENCY ANALYSIS

Visual Profiler: Critical path

Unguided Analysis

Analysis | GPU Details | CPU Details | Console | Settings

Reset All | Analyze All

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

Dependency Analysis

Results

Dependency Analysis

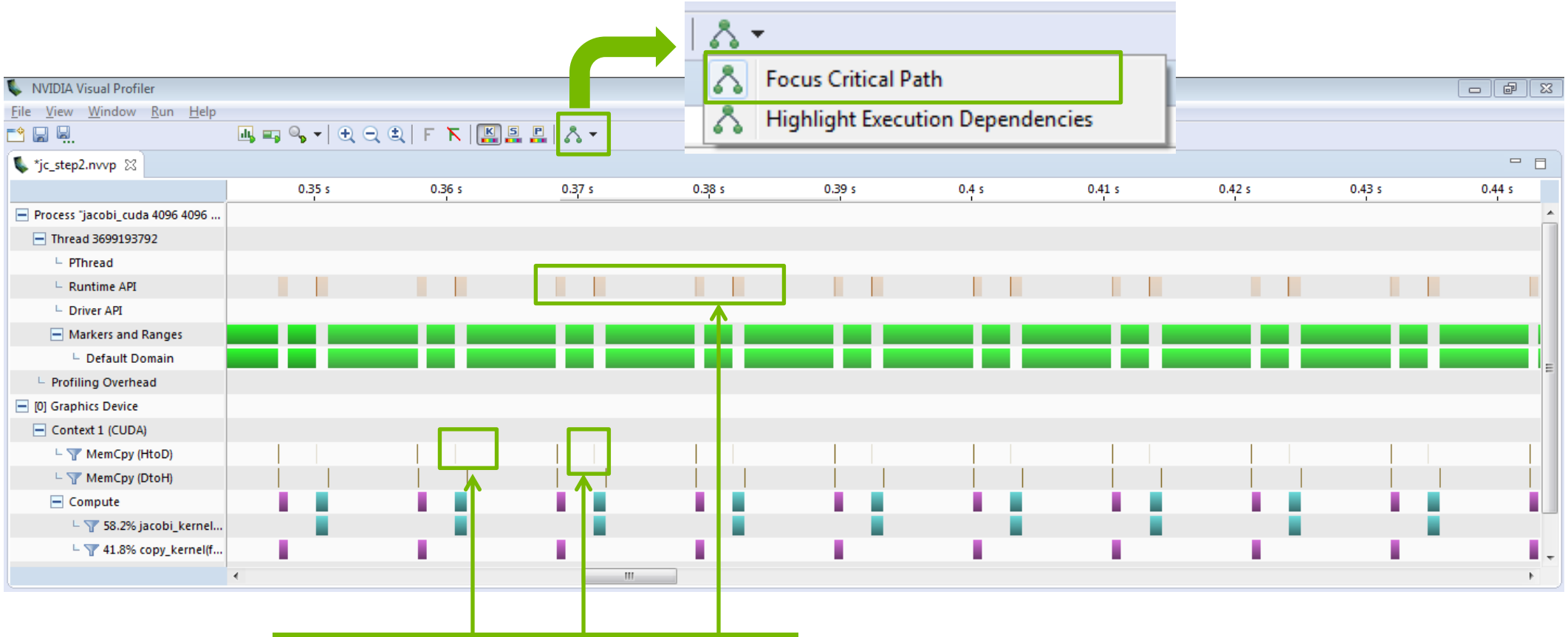
The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline. More...

Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
<Other>	73.07 %	931.895 ms	0 ns
cudaMalloc	9.95 %	126.876 ms	0 ns
jacobi_kernel(float const *, float*, int, int, float*)	7.06 %	90.008 ms	0 ns
copy_kernel(float*, float const *, int, int)	5.07 %	64.657 ms	0 ns
cudaMemcpy	3.15 %	40.154 ms	19.273 ms
[CUDA memcpy DtoH]	1.51 %	19.273 ms	0 ns
cudaSetupArgument	0.04 %	487.264 µs	0 ns
cudaFree	0.04 %	460.216 µs	0 ns
[CUDA memcpy HtoD]	0.03 %	402.204 µs	0 ns
cuDeviceGetAttribute	0.03 %	334.525 µs	0 ns
cudaGetDeviceProperties	0.03 %	322.983 µs	0 ns
cudaConfigureCall	0.01 %	187.042 µs	0 ns
cuDeviceTotalMem_v2	0.01 %	182.71 µs	0 ns
cuDeviceGetName	0.00 %	17.664 µs	0 ns
cudaSetDevice	0.00 %	12.913 µs	0 ns
cudaDeviceSynchronize	0.00 %	5.952 µs	64.656 ms
cuDeviceGetCount	0.00 %	3.346 µs	0 ns
cuDeviceGet	0.00 %	926 ns	0 ns
pthread_enter	0.00 %	0 ns	0 ns
pthread_exit	0.00 %	0 ns	0 ns
cudaLaunch	0.00 %	0 ns	0 ns

Functions on critical path

DEPENDENCY ANALYSIS

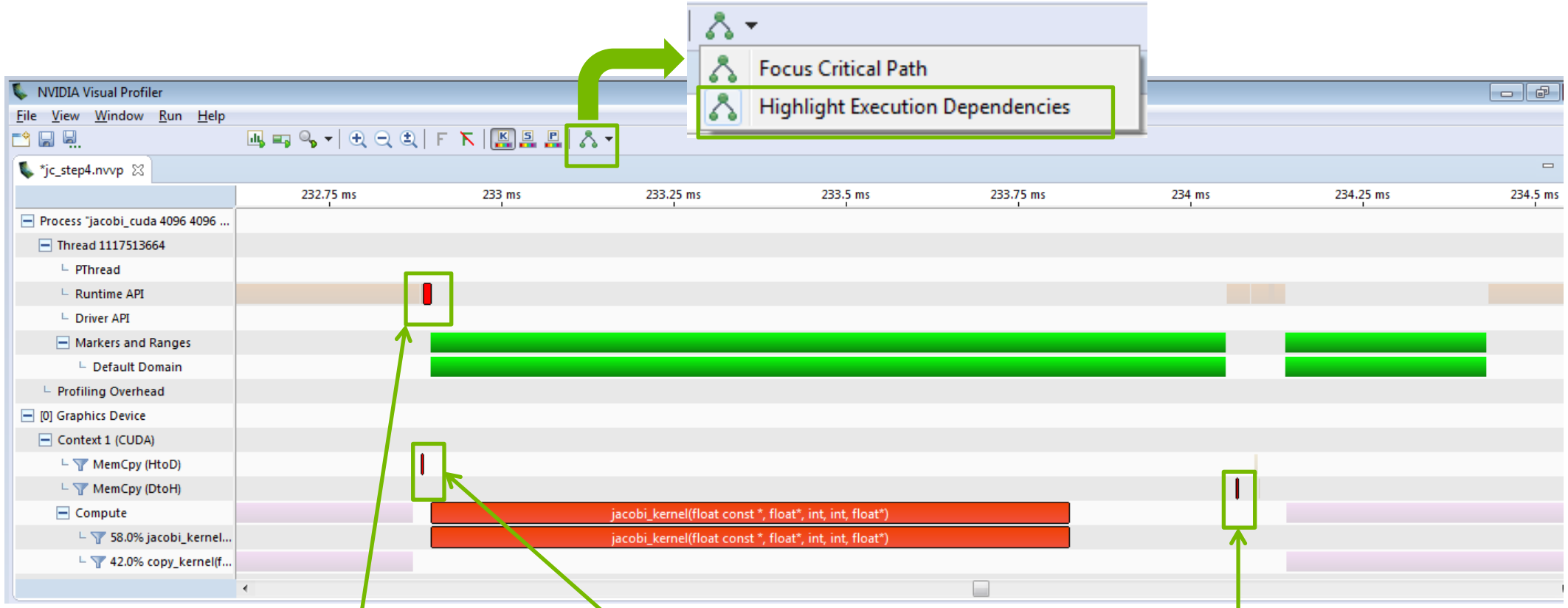
Visual Profiler



APIs, GPU activities not in critical path are greyed out

DEPENDENCY ANALYSIS

Visual Profiler



Launch jacobi_kernel

MemCpy HtoD [sync]

Inbound dependencies

MemCpy DtoH [sync]

Outbound dependencies

DEPENDENCY ANALYSIS

Visual Profiler

GPU kernel properties

Property view

Properties

jacobi_kernel(float const *, float*, int, int, float*)

Start	210.914 ms (210,913,664 ns)
End	211.838 ms (211,837,811 ns)
Duration	924.147 µs
Stream	Default
Grid Size	[257,255,1]
Block Size	[16,16,1]
Registers/Thread	12
Shared Memory/Block	0 B
Occupancy	
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	64 KiB
Shared Memory Executed	64 KiB
Shared Memory Bank Size	4 B
Dependency Analysis	
Time on Critical Path	924.147 µs
Waiting Time	0 ns

API properties

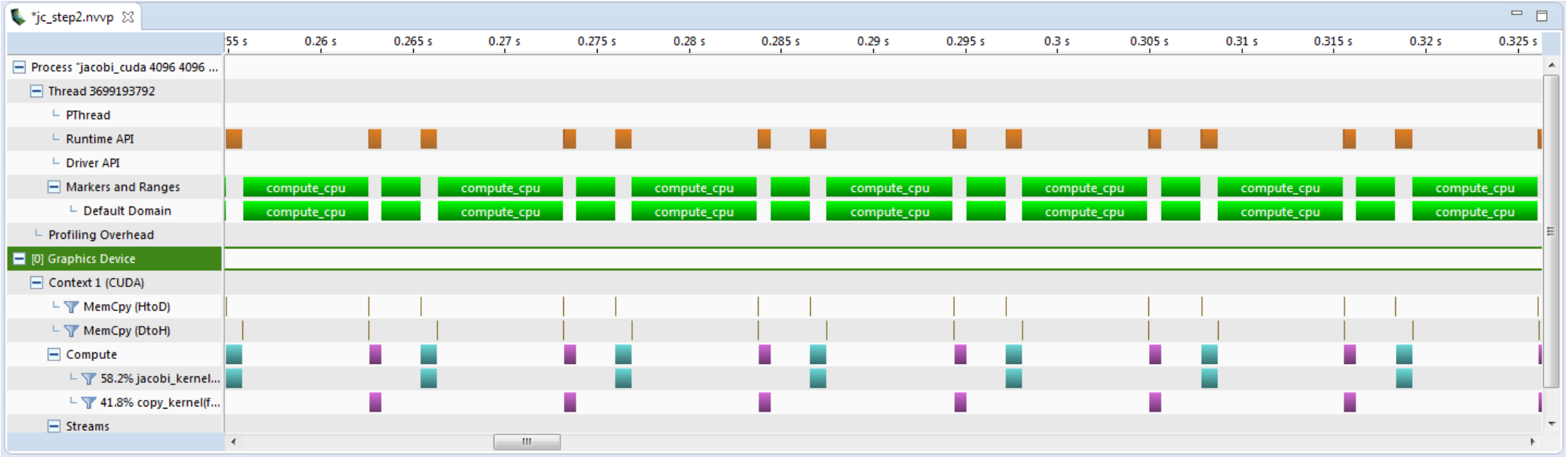
Properties

cudaDeviceSynchronize

Start	212.108 ms (212,107,870 ns)
End	212.582 ms (212,581,575 ns)
Duration	473.705 µs
Dependency Analysis	
Time on Critical Path	0 ns
Waiting Time	470.571 µs

DEPENDENCY ANALYSIS

Example: Step 1



Iterative execution pattern: 1. compute GPU+CPU 2. copy GPU+CPU

DEPENDENCY ANALYSIS

Example: Step 1

Results

Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[100 kernel instances] jacobi_kernel(float const *, float*, int, int, float*)
71	[100 kernel instances] copy_kernel(float*, float const *, int, int)

Guided analysis: Optimize jacobi_kernel

Properties

jacobi_kernel(float const *, float*, int, int, float*)

Duration	
Session	1.308 s (1,307,83...
Kernel	90.008 ms (90,00...
Invocations	100
Importance	58.2%

Kernel duration 6% of total session duration, kernel optimization may not impact application performance

DEPENDENCY ANALYSIS

Example: Step 1

Analysis | GPU Details | CPU Details | Console | Settings

Reset All | Analyze All

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

Results

Dependency Analysis

The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline. More...

Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
<Other>	73.07 %	931.895 ms	0 ns
cudaMalloc	9.95 %	126.876 ms	0 ns
jacobi_kernel(float const *, float*, int, int, float*)	7.06 %	90.008 ms	0 ns
copy_kernel(float*, float const *, int, int)	5.07 %	64.657 ms	0 ns
cudaMemcpy	3.15 %	40.154 ms	19.273 ms
[CUDA memcpy DtoH]	1.51 %	19.273 ms	0 ns
cudaSetupArgument	0.04 %	487.264 µs	0 ns
cudaFree	0.04 %	460.216 µs	0 ns
[CUDA memcpy HtoD]	0.03 %	402.204 µs	0 ns
cuDeviceGetAttribute	0.03 %	334.525 µs	0 ns
cudaGetDeviceProperties	0.03 %	322.983 µs	0 ns
cudaConfigureCall	0.01 %	187.042 µs	0 ns

Dependency analysis feature points that ‘Other’ CPU accounts for 73% of critical path

DEPENDENCY ANALYSIS

Example: Step 1

Results

Dependency Analysis

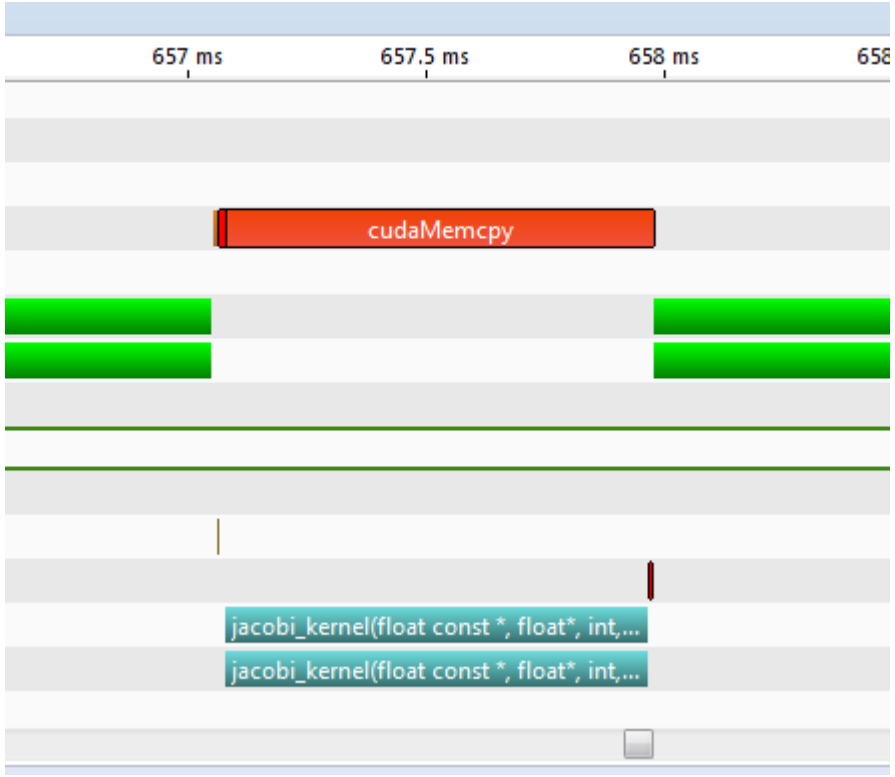
The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline. More...

Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
cudaDeviceSynchronize	0.00 %	5.952 µs	64.656 ms
cudaMemcpy	3.15 %	40.154 ms	19.273 ms
pthread_enter	0.00 %	0 ns	0 ns
pthread_exit	0.00 %	0 ns	0 ns
cuDeviceGetCount	0.00 %	3.346 µs	0 ns
cuDeviceGet	0.00 %	926 ns	0 ns
cuDeviceGetAttribute	0.03 %	334.525 µs	0 ns
cuDeviceGetName	0.00 %	17.664 µs	0 ns
cuDeviceTotalMem_v2	0.01 %	182.71 µs	0 ns
cudaGetDeviceProperties	0.03 %	322.983 µs	0 ns

Critical path sorted by waiting time

DEPENDENCY ANALYSIS

Example: Step 1



Properties

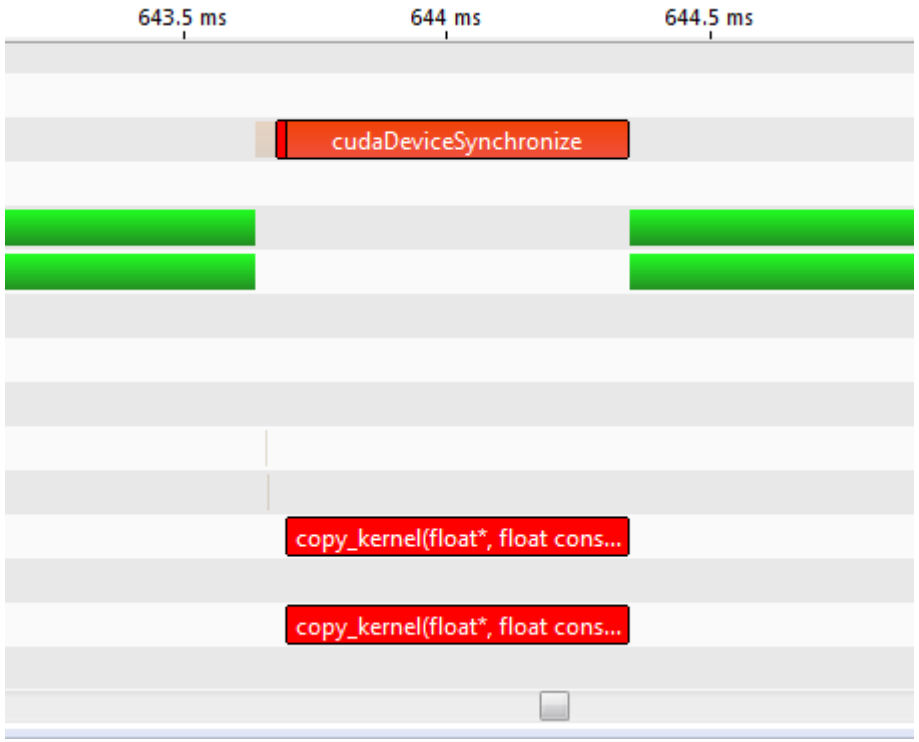
cudaMemcpy

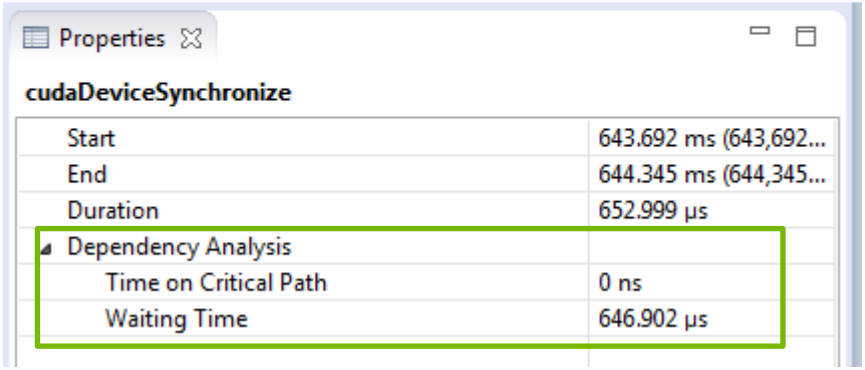
Start	657.08 ms (657,079,...
End	657.977 ms (657,977...
Duration	897.862 µs
Memory Copy	
Description	Memcpy DtoH [sync]
Start	657.971 ms (657,971...
End	657.972 ms (657,972...
Duration	640 ns
Size	4 B
Throughput	6.25 MB/s
Stream	Default
Memory Type	
Source	Device
Destination	Pageable
Dependency Analysis	
Time on Critical Path	0 ns
Waiting Time	640 ns

cudaMemcpy waiting for jacobi_kernel to finish

DEPENDENCY ANALYSIS

Example: Step 1





cudaDeviceSynchronize waiting for copy_kernel to finish

DEPENDENCY ANALYSIS

Sample code

Step 1 code

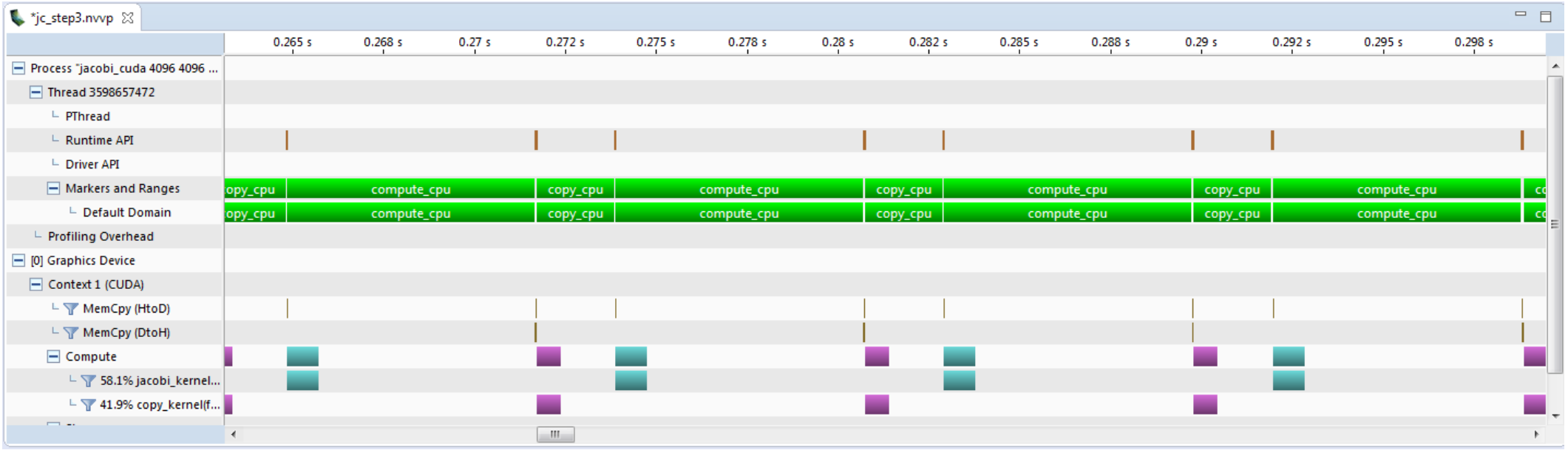
```
jacobi_kernel<<< … >>> (…);
cudaMemcpy(…);

compute_cpu

copy_kernel<<< … >>> (…);
cudaDeviceSynchronize(…);

copy_cpu
```

Step 2 code

```
jacobi_kernel<<< … >>> (…);

compute_cpu

cudaMemcpy(…);
copy_kernel<<< … >>> (…);

copy_cpu
cudaDeviceSynchronize(…);
```

DEPENDENCY ANALYSIS

Example: Step 2



CPU and GPU activities are overlapped

DEPENDENCY ANALYSIS

Results

Dependency Analysis

The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline. More...

Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
<Other>	82.89 %	930.587 ms	0 ns
cudaMalloc	11.39 %	127.891 ms	0 ns
cudaMemcpy	3.56 %	39.997 ms	18.94 ms
[CUDA memcpy DtoH]	1.69 %	18.94 ms	0 ns
cudaLaunch	0.20 %	2.196 ms	0 ns
cudaDeviceSynchronize	0.05 %	599.217 µs	0 ns
cudaFree	0.04 %	503.849 µs	0 ns
cudaSetupArgument	0.04 %	498.877 µs	0 ns
[CUDA memcpy HtoD]	0.04 %	402.07 µs	0 ns
cuDeviceGetAttribute	0.03 %	333.537 µs	0 ns
cudaGetDeviceProperties	0.03 %	323.848 µs	0 ns
cudaConfigureCall	0.02 %	195.514 µs	0 ns

Example: Step 2

GPU kernels are no more in critical path

Properties

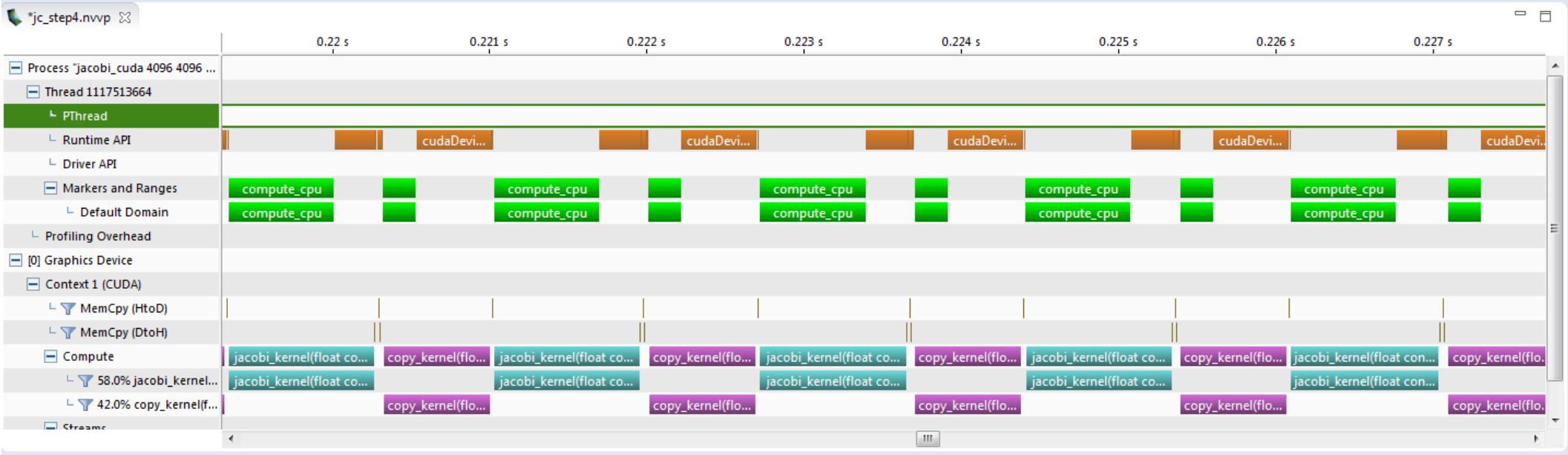
jacobi_kernel(float const *, float*, int, int, float*)

Duration	
Session	1.154 s (1,153,89...
Kernel	89.661 ms (89,66...
Invocations	100
Importance	58.1%

Session time is reduced from 1.3s to 1.15s due to overlap but kernel time is still very less compared to session time

DEPENDENCY ANALYSIS

Example: Step 3



Offload more work on GPU activity by changing CPU compute ratio from 5% to 0.5%

DEPENDENCY ANALYSIS

Example: Step 3

Results

Dependency Analysis

The following table shows metrics collected from a dependency analysis of the program execution. The data is summarized per function type. Use the "Dependency Analysis" menu on the main toolbar to visualize analysis results on the timeline. More...

Function Name	Time on Critical Path (%)	Time on Critical Path	Waiting time
cudaMalloc	32.74 %	127.053 ms	0 ns
jacobi_kernel(float const *, float*, int, int, float*)	20.74 %	80.505 ms	0 ns
copy_kernel(float*, float const *, int, int)	17.43 %	67.658 ms	0 ns
<Other>	12.51 %	48.529 ms	0 ns
cudaMemcpy	10.77 %	41.807 ms	19.946 ms
[CUDA memcpy DtoH]	5.14 %	19.946 ms	0 ns
cudaSetupArgument	0.13 %	519.834 µs	0 ns
cudaFree	0.11 %	411.38 µs	0 ns
[CUDA memcpy HtoD]	0.10 %	400.315 µs	0 ns
cuDeviceGetAttribute	0.09 %	333.638 µs	0 ns
cudaGetDeviceProperties	0.08 %	318.798 µs	0 ns
cudaLaunch	0.05 %	188.456 µs	0 ns

GPU kernels are on critical path. Time to optimize GPU kernels!

Properties

jacobi_kernel(float const *, float*, int, int, float*)

Duration	
Session	417.917 ms (417,...
Kernel	93.584 ms (93,58...
Invocations	100
Importance	58%

Session time is reduced significantly. 2.7X performance improvement without changing kernel

DEPENDENCY ANALYSIS

Limitations

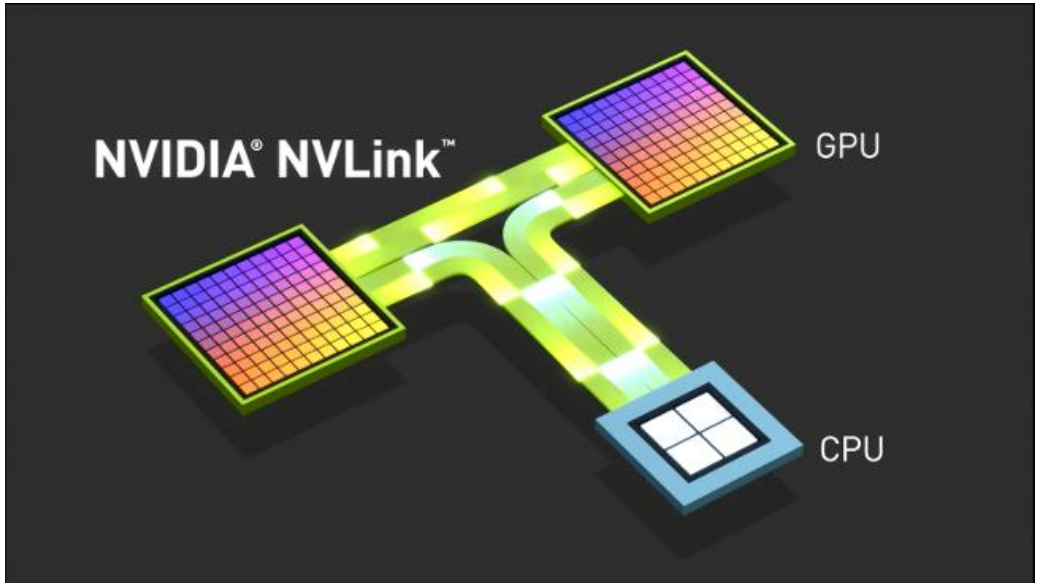
- Doesn't take into account wait states caused by CPU synchronization methods
- Doesn't account for synchronization done by polling memory location that will be updated by GPU activity
- Doesn't include synchronization caused by resource contention
- Limited support for dynamic parallelism – No dependency tracking for device launched kernels

NVLINK ANALYSIS

NVLINK ANALYSIS

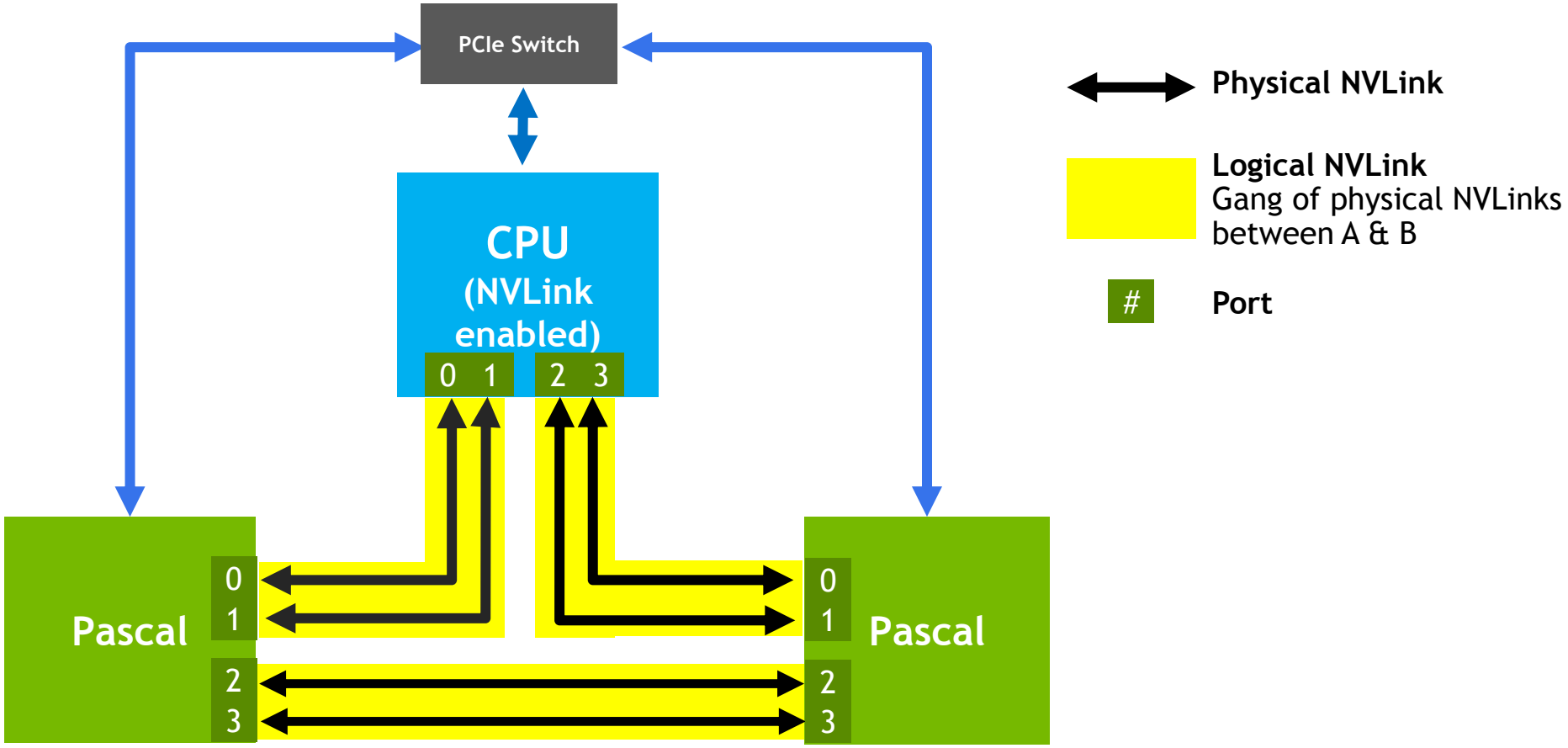
NVIDIA NVLINK HIGH-SPEED INTERCONNECT

- High-bandwidth, energy-efficient interconnect
- Enables ultra-fast communication between the CPU and GPU, and between GPUs
- Allows data sharing at rates 5 to 12 times faster than the traditional PCIe Gen3 interconnect



NVLINK ANALYSIS

Topology



NVLINK ANALYSIS

nvprof

- nvprof supports a new event collection mode “continuous”
- Supported only on Tesla GPUs
- Collects event samples every 2ms (fixed period for now)
- Metrics are collected at device level
- *Example: ./nvprof --aggregate-mode off --event-collection-mode continuous -metrics nvlink_total_data_transmitted,nvlink_total_data_received,nvlink_transmit_throughput,nvlink_receive_throughput -f -o memcpy.out ./memcpy*

nvprof new argument for sampling events

- To get detailed output i.e metric value along with timestamp for each sample use *--print-gpu-trace*

NVLINK ANALYSIS

nvprof

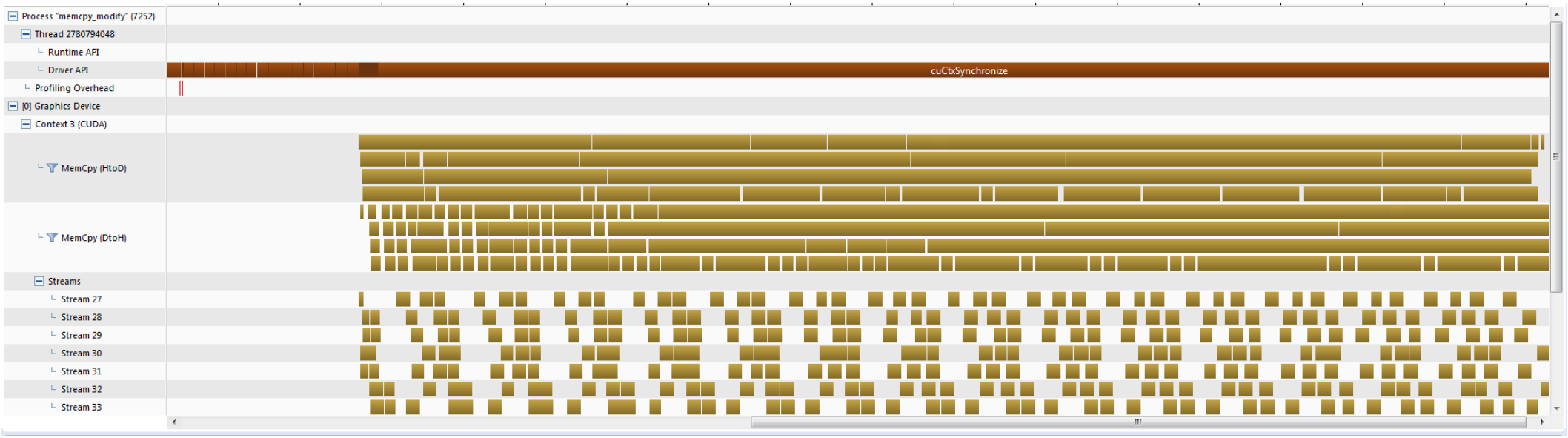
- Nvprof also gives topology information
- Example: nvprof --print-nvlink-topology ./app_name
- Output :

```
Graphics Device 1 port 0, 1, CPU, Nvlink Bandwidth 40.00GB/s, Physical Links 2, Sysmem Access True,
Sysmem Atomic Access False, Peer Access False, Peer Atomic Access False
Graphics Device 0 port 2, 3, CPU, Nvlink Bandwidth 40.00GB/s, Physical Links 2, Sysmem Access True,
Sysmem Atomic Access False, Peer Access False, Peer Atomic Access False
Graphics Device 0 port 0, 1, Graphics Device 1 port 3, 2, Nvlink Bandwidth 40.00GB/s, Physical Links 2,
Sysmem Access False, Sysmem Atomic Access False, Peer Access True, Peer Atomic Access True
```

- NVLink metrics have to be correlated by matching port number in topology record with instance number of metric

NVLINK ANALYSIS

Visual Profiler



Bidirectional memory transfers between CPU and GPU0

NVLINK ANALYSIS

Visual Profiler

Unguided Analysis

Static properties

Runtime values

Analysis | GPU Details | CPU Details | Console | Settings

Reset All | Analyze All

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

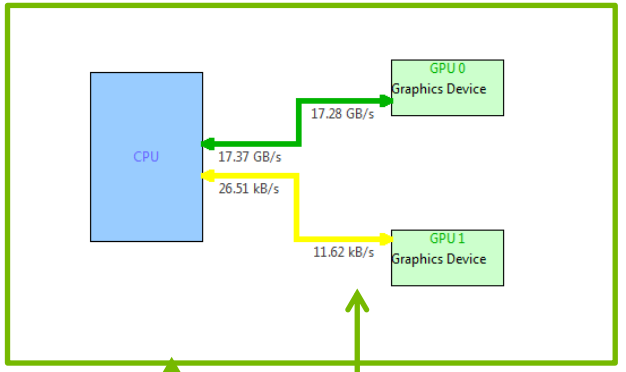
Application

- Data Movement And Concurrency
- Compute Utilization
- Kernel Performance
- Dependency Analysis
- NVLink

Results

NVLink Analysis

The following NVLink topology diagram shows logical NVLink connections between GPUs and CPU. The logical NVLink can contain one or more physical links. For a logical NVLink between two devices A and B, the average receive throughput for device A is shown close to d average receive throughput for device B is shown close to device B. Receive throughput of device A is same as transmit throughput of device B.
The tables on right hand side show the properties, bandwidth utilization and throughput of each logical NVLink.



Logical NVLink Properties

Logical NVLink	PeakBandwidth	PhysicalNVLinks	PeerAccess	SystemAccess	PeerAtomic	SystemAtomic	Utilization %	Idle time %
GPU1<-->CPU	40 GB/s	2	No	Yes	No	No	0	83
GPU0<-->CPU	40 GB/s	2	No	Yes	No	No	86	19

Logical NVLink Throughput

Logical NVLink	Avg Throughput	Max Throughput	Min Throughput
GPU1-->CPU	26.512 kB/s	50.382 kB/s	13.599 kB/s
GPU1<--CPU	11.623 kB/s	21.691 kB/s	5.779 kB/s
GPU0-->CPU	17.372 GB/s	20.042 GB/s	3.774 kB/s
GPU0<--CPU	17.28 GB/s	19.757 GB/s	1.604 kB/s

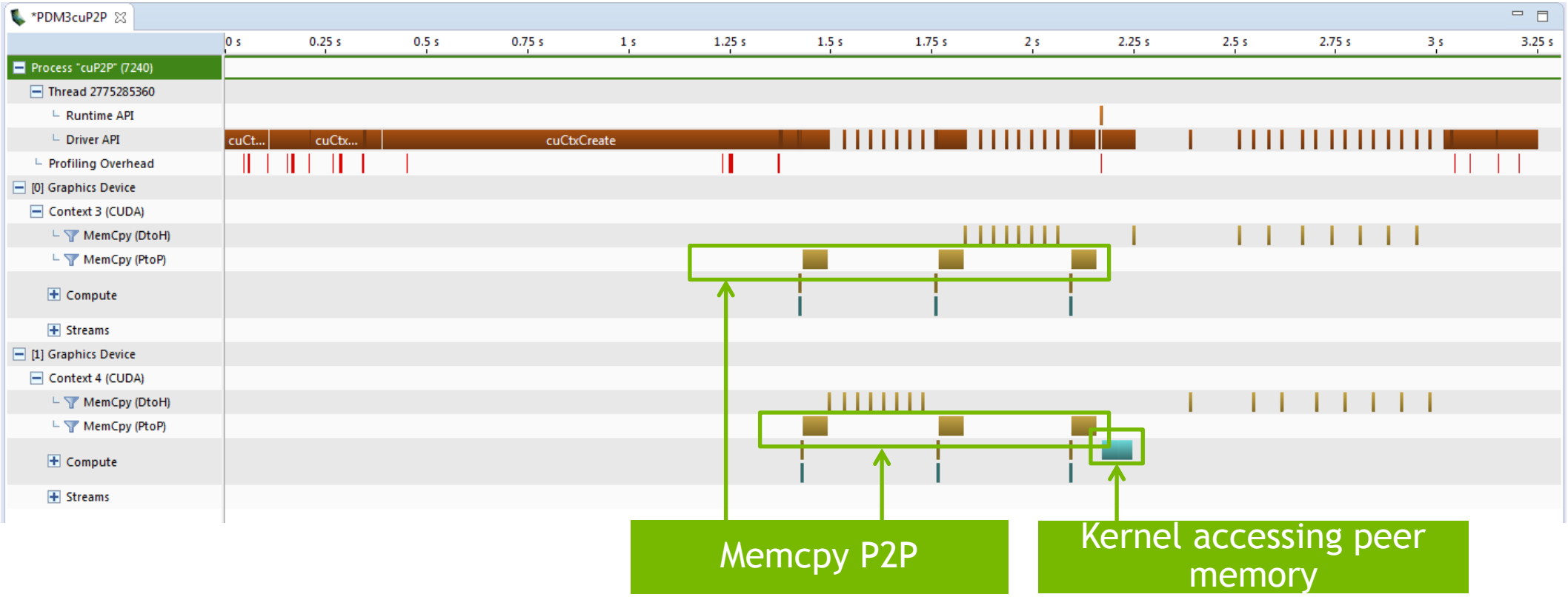
Option to collect NVLink information

Topology

Achieved throughput

NVLINK ANALYSIS

Visual Profiler



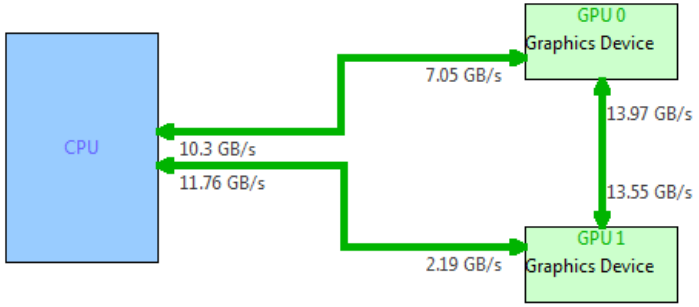
NVLINK ANALYSIS

Visual Profiler

Results

NVLink Analysis

The following NVLink topology diagram shows logical NVLink connections between GPUs and CPU. The logical NVLink can contain one or more physical links. For a logical NVLink between two devices A and B, the average receive throughput for device A
average receive throughput for device B is shown close to device B. Receive throughput of device A is same as transmit throughput of device B.
The tables on right hand side show the properties, bandwidth utilization and throughput of each logical NVLink.



Logical NVLink Properties

Logical NVLink	PeakBandwidth	PhysicalNVLin...	PeerAccess	SystemAccess	PeerAtomic	SystemAtomic	Utilization %	Idle time %
GPU0<-->CPU	40 GB/s	2	No	Yes	No	No	43	64
GPU0<-->GPU1	40 GB/s	2	Yes	No	Yes	No	68	68
GPU1<-->CPU	40 GB/s	2	No	Yes	No	No	34	74

Logical NVLink Throughput

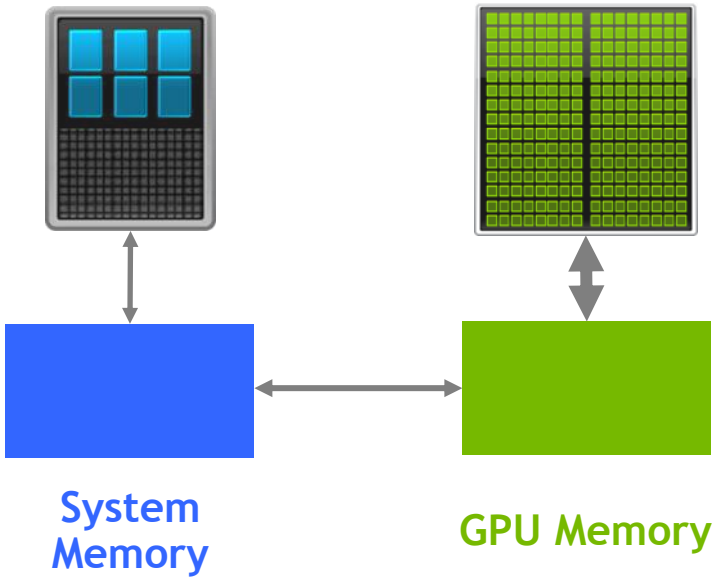
Logical NVLink	Avg Throughput	Max Throughput	Min Throughput
GPU0-->CPU	10.3 GB/s	37.881 GB/s	7.808 kB/s
GPU0<--CPU	7.053 GB/s	38.001 GB/s	11.712 kB/s
GPU0-->GPU1	13.555 GB/s	75.76 GB/s	15.212 kB/s
GPU0<--GPU1	13.968 GB/s	76.002 GB/s	10.14 kB/s
GPU1-->CPU	11.759 GB/s	66.441 GB/s	23.04 kB/s
GPU1<--CPU	2.19 GB/s	11.72 GB/s	7.68 kB/s

UNIFIED MEMORY

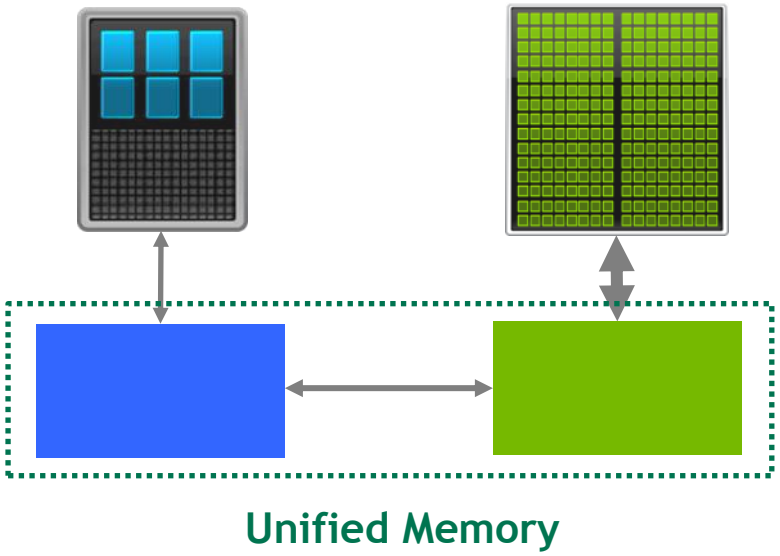
UNIFIED MEMORY

Starting with Kepler and CUDA 6

Custom Data Management



Developer View With Unified Memory



UNIFIED MEMORY

- Single allocation, single pointer accessible everywhere
- Pascal GPUs support demand paging
  - Pages populated and data migrated on first touch, overhead of transferring entire allocation is eliminated
  - Concurrent access to memory from CPU and GPU
  - Enables applications with large data models by allowing to oversubscribe GPU memory by spilling over to CPU memory
  - Can access OS controlled memory on supporting system

UNIFIED MEMORY

CUDA 6.0+ code

```
void sortfile(FILE *fp, int N) {
char *data;
cudaMallocManaged(&data, N);

fread(data, 1, N, fp);

qsort<<<...>>>(data,N,1,compare);
cudaDeviceSynchronize();

use_data(data);

cudaFree(data);
}
```

Pages allocated in GPU memory

CPU page fault, data migrates to CPU

Kernel launch, data migrates to GPU

CUDA 8.0 Code *

```
void sortfile(FILE *fp, int N) {
char *data;
cudaMallocManaged(&data, N);

fread(data, 1, N, fp);

qsort<<<...>>>(data,N,1,compare);
cudaDeviceSynchronize();

use_data(data);

cudaFree(data);
}
```

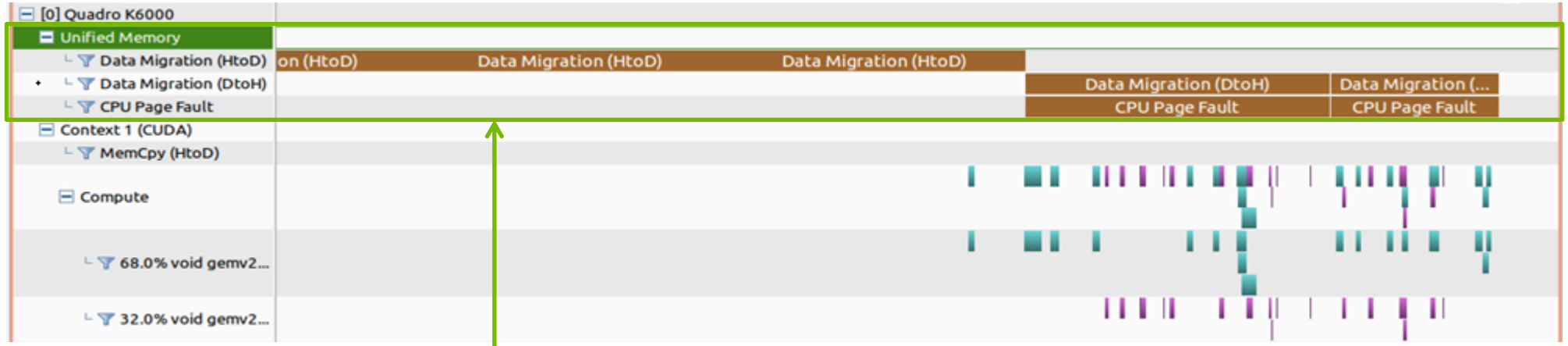
Empty, no pages anywhere

CPU page fault, data allocates on CPU

GPU page fault, data migrates to GPU

UNIFIED MEMORY

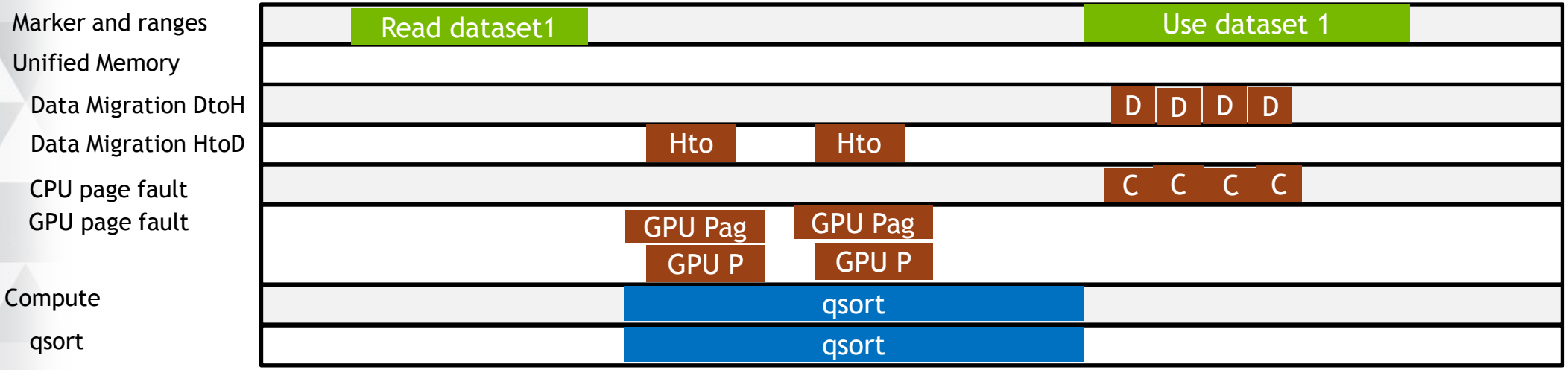
Visual profiler - 6.0+ unified memory



Unified memory timeline

UNIFIED MEMORY

Visual profiler - 8.0 unified memory timeline



Work in progress mockup slides

UNIFIED MEMORY

Visual profiler – Properties of faults and migrations

CPU Fault	
Fault generated	19.250ms
Fault resolved	19.251ms
Page address	0x23456000
Page size	4KB
Access type	Read

GPU Fault	
Fault generated	19.236ms
Fault resolved	19.240ms
Page address	0x23456000
Page size	4KB
Access type	RW

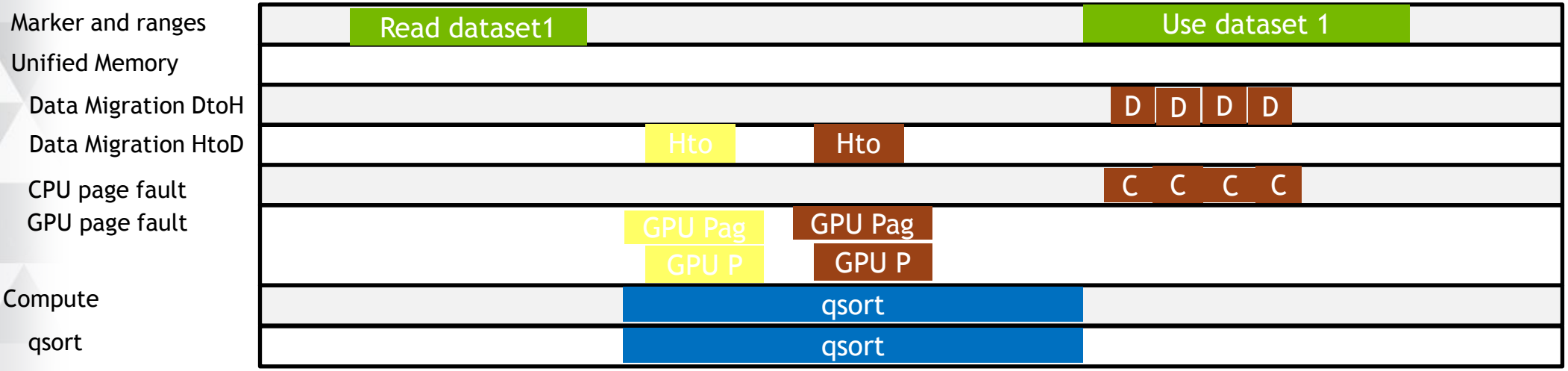
Data Migration (DtoH)	
Start	19.250 ms
End	19.251ms
Duration	1us
Start address	0x23456000
Size	4KB
Process	16763

Data Migration (HtoD)	
Start	19.237 ms
End	19.239ms
Duration	2us
Start address	0x23456000
Size	8KB
Process	16763

Work in progress mockup slides

UNIFIED MEMORY

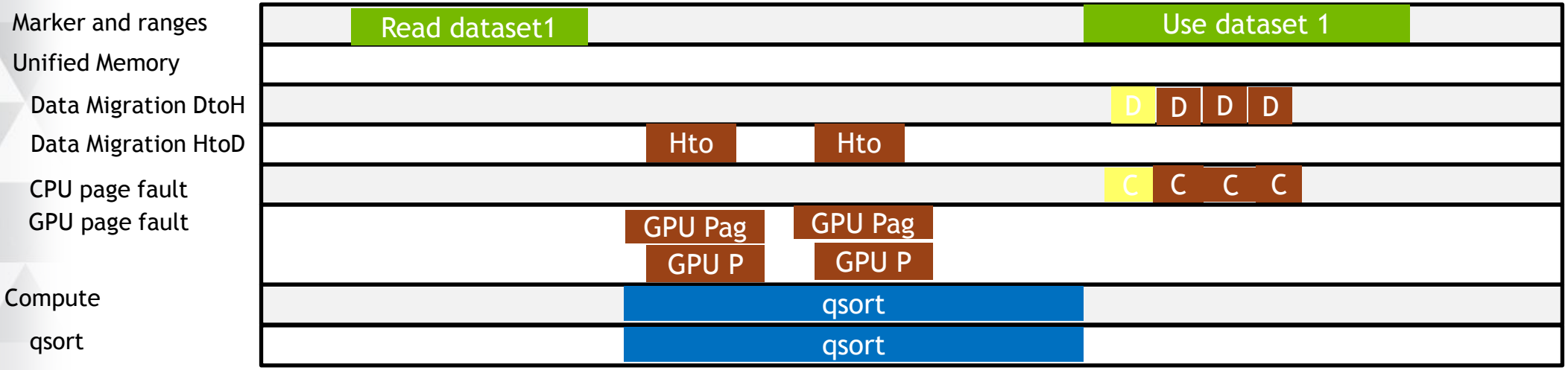
Visual profiler - Fault-migration correlation



Work in progress mockup slides

UNIFIED MEMORY

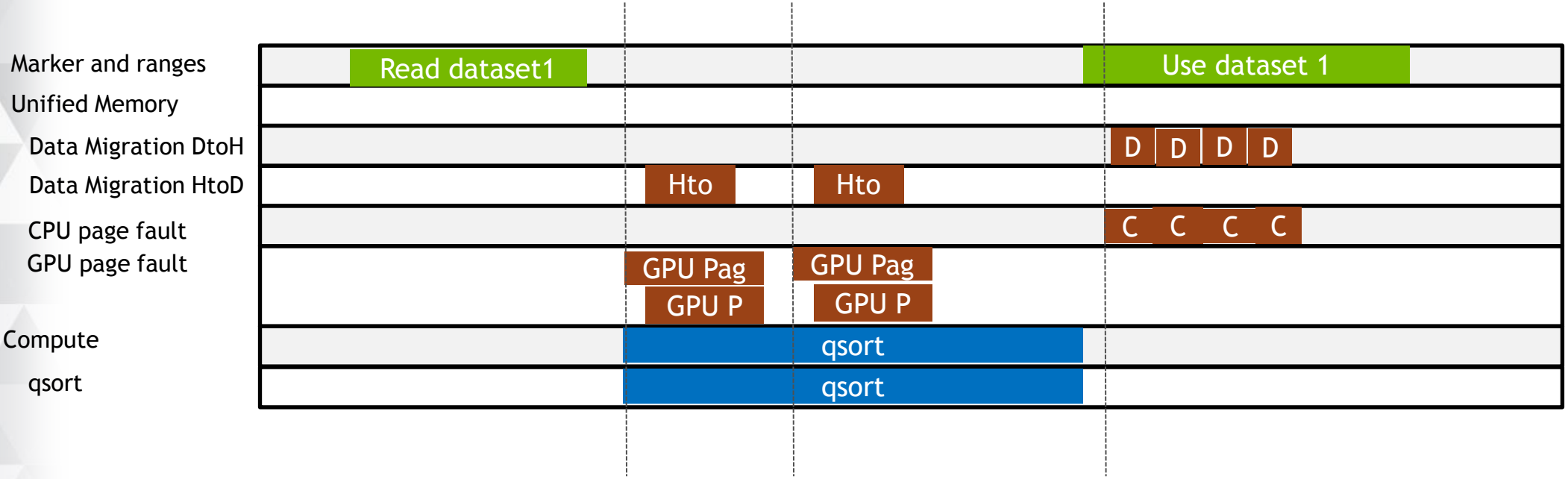
Visual profiler - Fault-migration correlation



Work in progress mockup slides

UNIFIED MEMORY

Visual profiler - Correlating fault to source

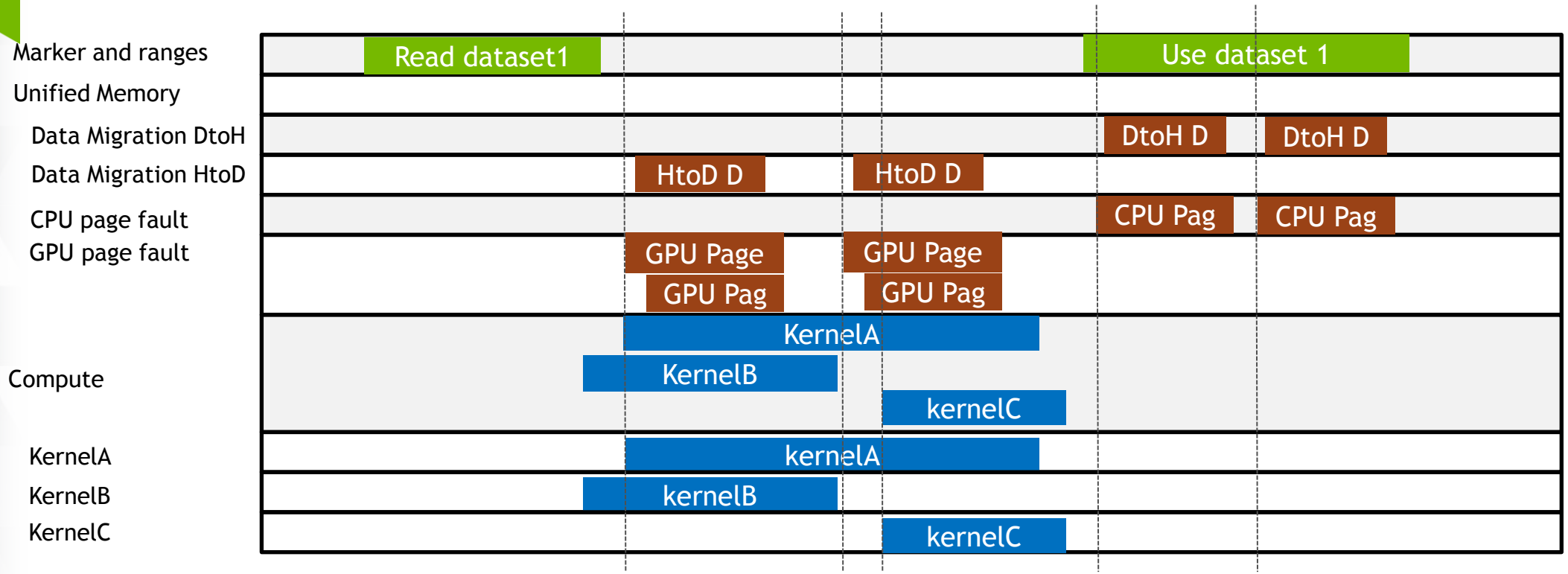


Manually map the GPU page faults to kernels and CPU page faults to NVTX annotated regions on timeline

Work in progress mockup slides

UNIFIED MEMORY

Visual profiler - Correlating fault to source



Use VA range of allocations used in kernels to correlate with page address from corresponding page fault

Work in progress mockup slides

INSTRUCTION LEVEL PROFILING (PC SAMPLING)

PC SAMPLING

- PC sampling feature is introduced in 7.5, available for 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
- Maxwell architecture gives overall view of scheduling in GPU
  - Samples warp states periodically in round robin order over all active warps
  - Sampling rate is fixed in visual profiler for a GPU
  - No overheads in kernel runtime, CPU overheads to parse the records

PC SAMPLING ALGORITHM

	Warp scheduler 0	Warp scheduler 1	Warp scheduler 2	Warp scheduler 3
Max warps	16	16	16	16
Active warps	8 (w0, w4, w8, …, w28)	8 (w1, w5, w9, …, w29)	8 (w2, w6, w10, …, w30)	8 (w3, w7, w11, …, w31)

Time in cycles	Warp scheduler 0	Warp scheduler 1	Warp scheduler 2	Warp scheduler 3
0	w0			
256		w1		
512			w2	
768				w3
1024	w4			
1280		w5		
1536			w6	
1792				w7

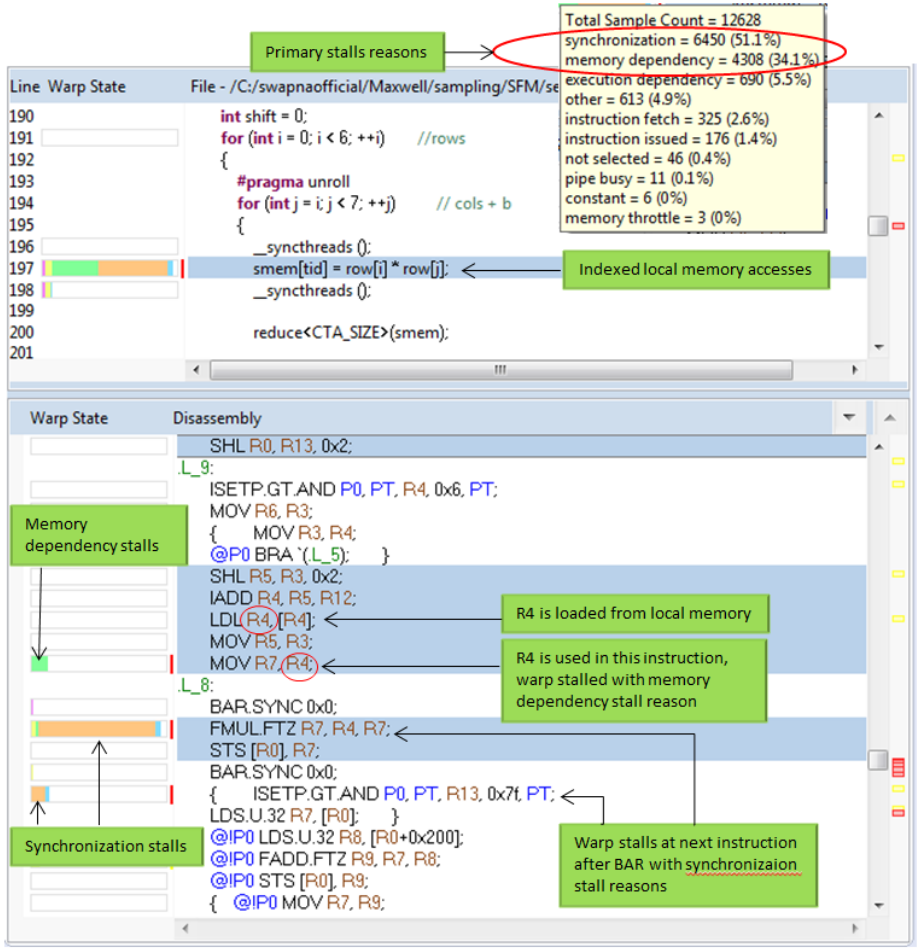
PC SAMPLING

Example

Iterative Closest Point algorithm

Primary stall reasons:

- Memory dependency:
  - LDL ("load local") instructions.
  - Not because of register spilling
  - Local memory is used for local variables with indexed access
- Synchronization stalls
  - BAR.SYNC barrier instruction i.e. __syncthreads()



PC SAMPLING EXAMPLE

Original Code

```
float row[7]
//Initialize array row
int shift = 0;
__shared__ float smem[CTA_SIZE];
for (int i = 0; i < 6; ++i)        // rows
{
    #pragma unroll
    for (int j = i; j < 7; ++j)    // cols + b
    {
        __syncthreads ();                  // sync
        smem[tid] = row[i] * row[j];      // local load
        __syncthreads ();

        reduce(smem);

        if (tid == 0)
            gbuf.ptr (shift++)[blockIdx.x + gridDim.x * blockIdx.y]
                = smem[0];
    }
}
```

New Code (LDL removed)

```
float row0, row1, row2, row3, row4, row5, row6;
//Initialize all elements
#define UNROLL_REDUCE(val, buf)                                    \
    do {                                                           \
        smem[tid] = val;                                           \
            __syncthreads();                                       \
            reduce(smem);                                          \
            if (tid == 0)                                          \
                buf.ptr (shift++)[blockIdx.x + gridDim.x * blockIdx.y]  \
                    = smem[0];                                     \
    } while(0)

UNROLL_REDUCE(row0*row0, gbuf);
UNROLL_REDUCE(row0*row1, gbuf);
UNROLL_REDUCE(row0*row2, gbuf);
UNROLL_REDUCE(row0*row3, gbuf);
UNROLL_REDUCE(row0*row4, gbuf);
```

Perf: 1.6x (2.3ms vs 3.9ms)

COMBINED SOURCE LEVEL ANALYSIS

COMBINED SOURCE LEVEL ANALYSIS

Visual profiler

All the source level analysis are combined in the same view

- Global access
- Shared access
- Divergent branch
- Instruction level execution
- PC sampling
- Register pressure

Easy analysis, can pinpoint issues for stalls in some cases

COMBINED SOURCE LEVEL ANALYSIS

Add/hide source level analysis

Selects hotspot when multiple analyses are enabled

NVIDIA Visual Profiler

File View Window Help

View | Hot Spot - Shared Access

*vpe.pdm | MemoryInefficientShared

Line | Shared Access | Latency Reasons | File - /C:/swapnaofficial/GTC16/sampling/viper_expertsystem.cu

```
261
262  __global__ void
263  MemoryInefficientShared(int *A, int *B, int *C, int N)
264  {
265   const int SZ = (12 * 1024) / sizeof(int);
266   __shared__ int sh[SZ];
267   int tid = threadIdx.x;
268   int index = tid * 8;
269
270   if (tid < SZ) {
271    sh[tid] = 0;
272   }
273
274   if (index < SZ) {
275    int l_data_32 = sh[index];
276    sh[SZ - index - 1] = l_data_32;
277   }
278  }
279
280  // This kernel creates balanced memory and compute utilization. The
281  // goal here is to get an equal compute and memory utilization with
```

Shared Access | Latency Reasons | Disassembly

```
_Z23MemoryInefficientSharedPiS_S_i:
    MOV R1, c[0x0][0x20];
    S2R R0, SR_TID.X;
    SHL R2, R0.reuse, 0x3;
    ISETP.GT.AND P0, PT, R0.reuse, 0xbff, PT;
    ISETP.GT.AND P1, PT, R2, 0xbff, PT;
    @!P0 SHL R0, R0, 0x2;
    @!P0 STS [R0], RZ;
    @P1 EXIT;
    SHL R0, R2.reuse, 0x2;
    LDS.U.32 R0, [R0];
    IADD32I R2, -R2, 0xbff;
    SHL R2, R2, 0x2;
    STS [R2], R0;
    EXIT;
.L_1:
    BRA `(.L_1);
    NOP;
    NOP;
    NOP;
.L_540:
```

Shared memory load/store bank conflicts cause execution dependency and memory throttle stalls

COMBINED SOURCE LEVEL ANALYSIS

Register pressure

NVIDIA Visual Profiler

File View Window Help

View | Hot Spot - Registers

*vpe.pdm | LatencyOccupancyRegister

Line | Latency Reasons | File - /C:/swapnaofficial/GTC16/sampling/viper_expertsystem

```
142
143  #pragma unroll
144    for (j = 0; j < LEN; j++) {
145      Ia[j] = ((i+j) < N) ? C[i] : C[N-1];
146    }
147    /* Some memory accesses to prevent being compute bound
148    A[i] += Ia[j];
149
150  #pragma unroll
151    for (j = 0; j < (LEN - 8); j++) {
152  #pragma unroll
153      for (k = 0; k < 8; k++) {
154        Ib[j] = Ia[j] + Ia[k];
155      }
156      C[i] += Ib[j];
157    }
158  }
159  }
160
```

Latency Reasons | Registers | Disassembly

```
ISETP.LT.AND P4, PT, R9, c[0x0][0x158], PT;
ISETP.LT.AND P5, PT, R11, c[0x0][0x158], PT;
ISETP.LT.AND P6, PT, R13, c[0x0][0x158], PT;
SHR R9, R3, 0x1e;
IADD.X R3, R7, c[0x0][0x154];
{      IADD R6.CC, R8, c[0x0][0x150];
LDG.E R13, [R2];      }
SHL R11, R0, 0x2;
SEL R64, R0.reuse, R5.reuse, P0;
SEL R63, R0.reuse, R5.reuse, P1;
SEL R19, R0.reuse, R5.reuse, P2;
SEL R68, R0.reuse, R5.reuse, P3;
IADD.X R7, R9, c[0x0][0x154];
SEL R67, R0, R5, P4;
SEL R8, R0.reuse, R5.reuse, P5;
SEL R9, R0.reuse, R5, P6;
Registers/Thread = 123 [Maximum Registers/Thread = 125]
LDG.E R11, [R6];      }
SHL R126, R4.reuse, 0x2;
```

Hotspot only for assembly

Registers

Registers/Thread		128	65536	0 8192 16384 24576 32768 40960 49152 57344 65536
Registers/Block		28672	65536	0 16k 32k 48k 64k
Block Limit		2	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32

Register is the limiting factor for occupancy

COMPUTE PREEMPTION

COMPUTE PREEMPTION

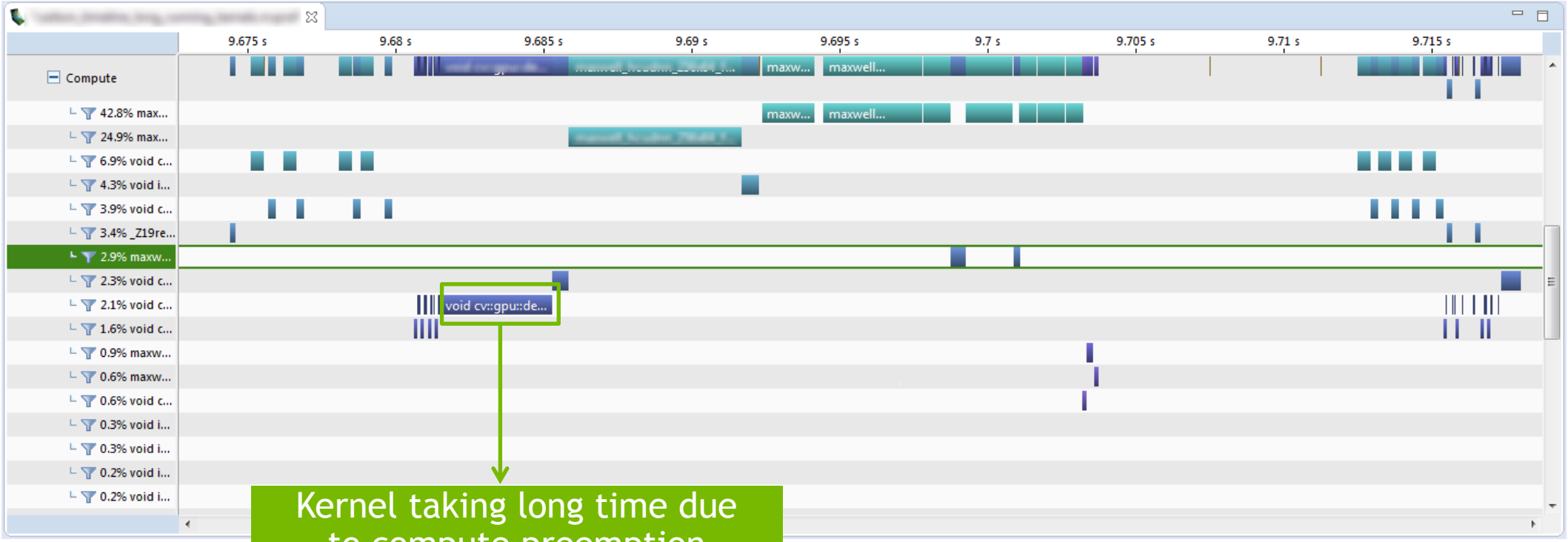
Pascal architecture introduces a new feature compute to give fair chance for all compute contexts while running long tasks.

How it affects profiling results?

- If multiple contexts are running in parallel it is possible that long kernels will get preempted.
- Some kernels may get preempted occasionally due to timeslice expiry for the context
- In CUDA 8.0, if kernel has been preempted mid execution, the time the kernel spends preempted is still counted towards kernel duration
- This can affect the kernel optimization priorities given by visual profiler as there is randomness introduced due to preemption

COMPUTE PREEMPTION

Visual profiler



Kernel taking long time due to compute preemption

COMPUTE PREEMPTION

How to get accurate results?

- Run only one context at a time
  - use as secondary GPU
  - unload display driver in linux
  - run only one process (that uses GPU) at one time

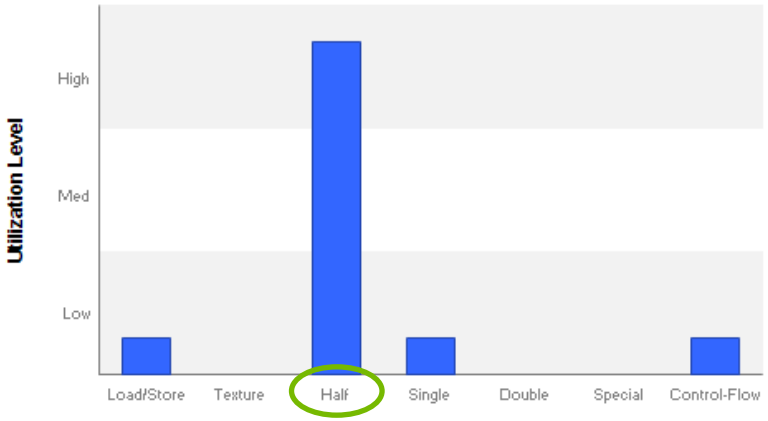
FP16 ANALYSIS

FP16 ANALYSIS

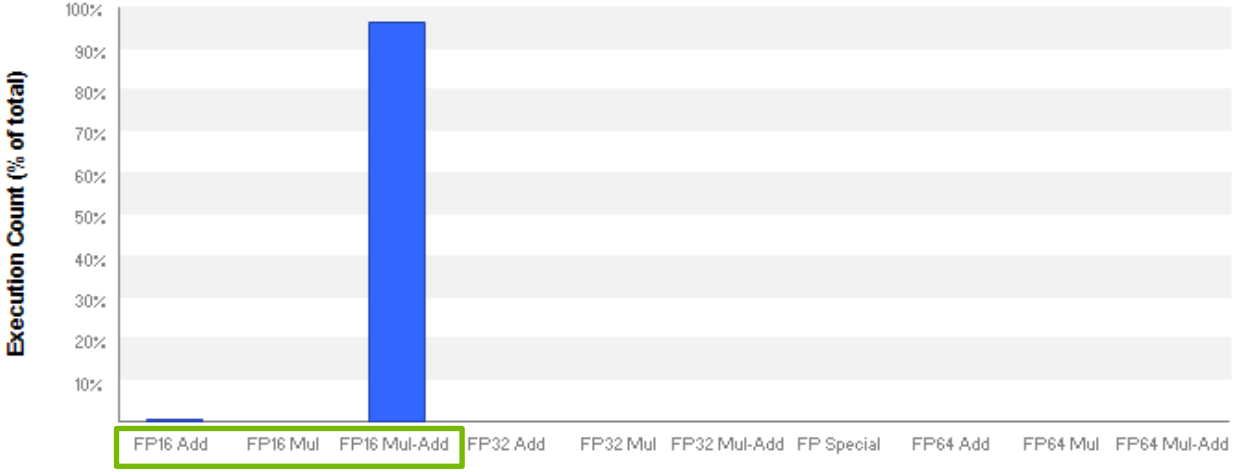
- FP16 (half precision) support added in CC 5.3 and 6.0 (Pascal architecture)
- Stores up to 2x larger models in GPU memory.
- Reduce memory bandwidth requirements by up to 2x.
- Profiler gives the instruction counts, half precision function unit utilization and floating point operations count to analyze performance of fp16

FP16 ANALYSIS

Function unit utilization



Instruction Mix



FLOP count

NVIDIA TOOLS EXTENSION (NVTX) V2

NVTX V2

- NVTX is used for annotating events, code ranges, resources
- Multiple middleware annotating using same strings cause collision
- NVTX V2 introduces domain concept, each middleware can use its own domain
  - Now middleware and your application don’t need to collide
  - Visual profiler shows markers/ranges of each domain on separate timeline
- Synchronization primitives can also be named
  - Tools can track and present why you are blocked with a custom message

NVTX V2

Sample code

```
Module A

 eventAttrib.message.ascii = “Range1";
 nvtxRangeId_t idex0 =
nvtxRangeStartEx(&eventAttrib);
 //CPU code
 nvtxRangeEnd(idex0);
```

```
Module B

eventAttrib.message.ascii = “Range1";
 nvtxRangeId_t idex1 =
nvtxRangeStartEx(&eventAttrib);
 //CPU code
 nvtxRangeEnd(idex1);
```

```
Module A
 nvtxDomainHandle_t domain_a =
nvtxDomainCreateA(“ModuleA");

 eventAttrib.message.ascii = “Range1";
 nvtxRangeId_t idex0 =
nvtxDomainRangeStartEx(domain_a, &eventAttrib);
 //CPU code
 nvtxDomainRangeEnd(domain_a, idex0);
```

Domain A

```
Module B
 nvtxDomainHandle_t domain_b =
nvtxDomainCreateA(“ModuleB");

 eventAttrib.message.ascii = “Range1";
 nvtxRangeId_t idex1 =
nvtxDomainRangeStartEx(domain_b, &eventAttrib);
 //CPU code
 nvtxDomainRangeEnd(domain_b, idex1);
```

Domain B

NVTX V2

nvprof

```
==23855== NVTX result:
==23855==   Thread <unnamed> (id = 3129952128)
==23855==     Domain "<unnamed>"
==23855==       Range "Range1" (4 times, total time: 1.7567ms)
Time(%)      Time     Calls       Avg       Min       Max  Name
 60.12%  392.61us         8  49.076us  32.799us  64.833us  [CUDA memcpy HtoD]
 28.84%  188.32us         4  47.080us  32.159us  62.113us  [CUDA memcpy DtoH]
  5.64%  36.863us         4  9.2150us  4.3200us  14.143us  VecAdd(int const *, int const *, int*, int)
  5.40%  35.295us         4  8.8230us  3.7440us  13.792us  VecSub(int const *, int const *, int*, int)
==23855==     API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 89.32%  830.67us        12  69.222us  9.9990us  207.59us  cudaMemcpyAsync
 10.68%  99.355us         8  12.419us  6.6150us  26.464us  cudaLaunch
```

Range information is grouped based on range name

```
==23845== NVTX result:
==23845==   Thread <unnamed> (id = 3199170432)
==23845==     Domain "Module A"
==23845==       Range "Range1" (2 times, total time: 962.08us)
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.04%  195.78us         4  48.944us  33.279us  64.065us  [CUDA memcpy HtoD]
 29.78%  98.752us         2  49.376us  36.895us  61.857us  [CUDA memcpy DtoH]
  5.67%  18.815us         2  9.4070us  4.7680us  14.047us  VecAdd(int const *, int const *, int*, int)
  5.50%  18.240us         2  9.1200us  4.0960us  14.144us  VecSub(int const *, int const *, int*, int)
==23845==     API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 87.91%  441.43us         6  73.572us  21.199us  168.55us  cudaMemcpyAsync
 12.09%  60.696us         4  15.174us  7.1780us  25.665us  cudaLaunch
==23845==     Domain "Module B"
==23845==       Range "Range1" (2 times, total time: 890.51us)
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.80%  201.73us         4  50.432us  33.183us  70.274us  [CUDA memcpy HtoD]
 29.35%  99.008us         2  49.504us  36.895us  62.113us  [CUDA memcpy DtoH]
  5.55%  18.720us         2  9.3600us  4.2560us  14.464us  VecAdd(int const *, int const *, int*, int)
  5.29%  17.855us         2  8.9270us  3.6800us  14.175us  VecSub(int const *, int const *, int*, int)
==23845==     API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 91.21%  436.15us         6  72.691us  19.169us  204.58us  cudaMemcpyAsync
  8.79%  42.031us         4  10.507us  7.0370us  14.902us  cudaLaunch
```

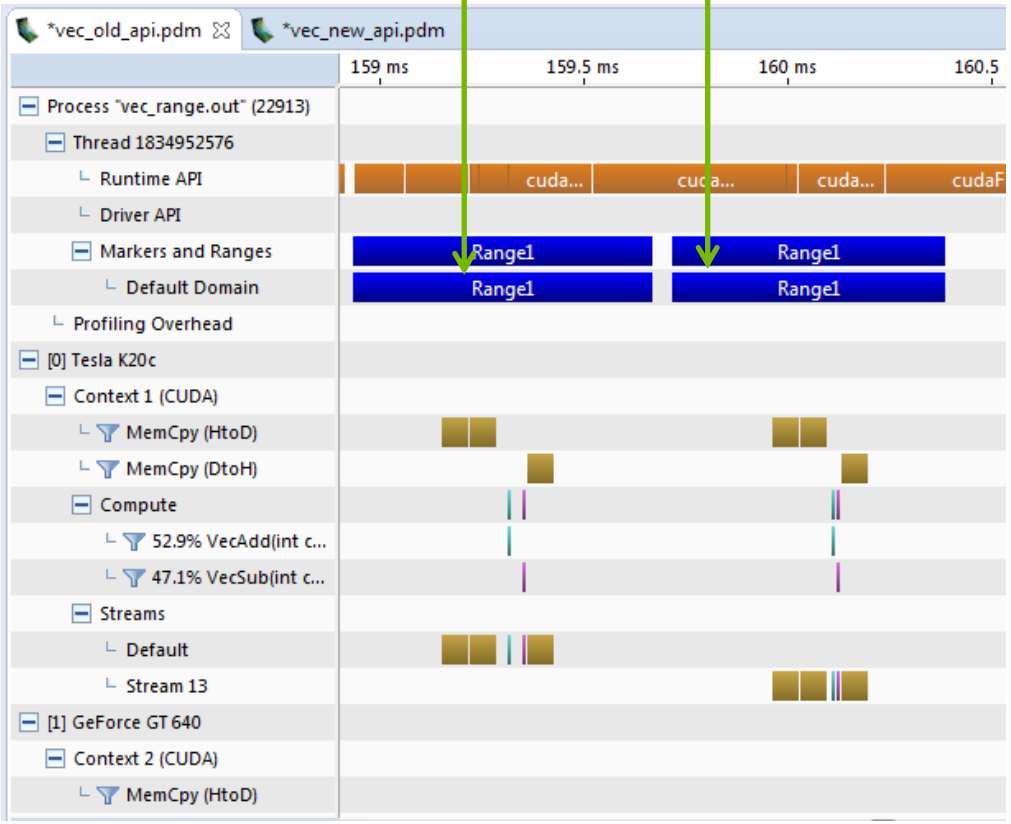
Range information is grouped based on domain first and then range name

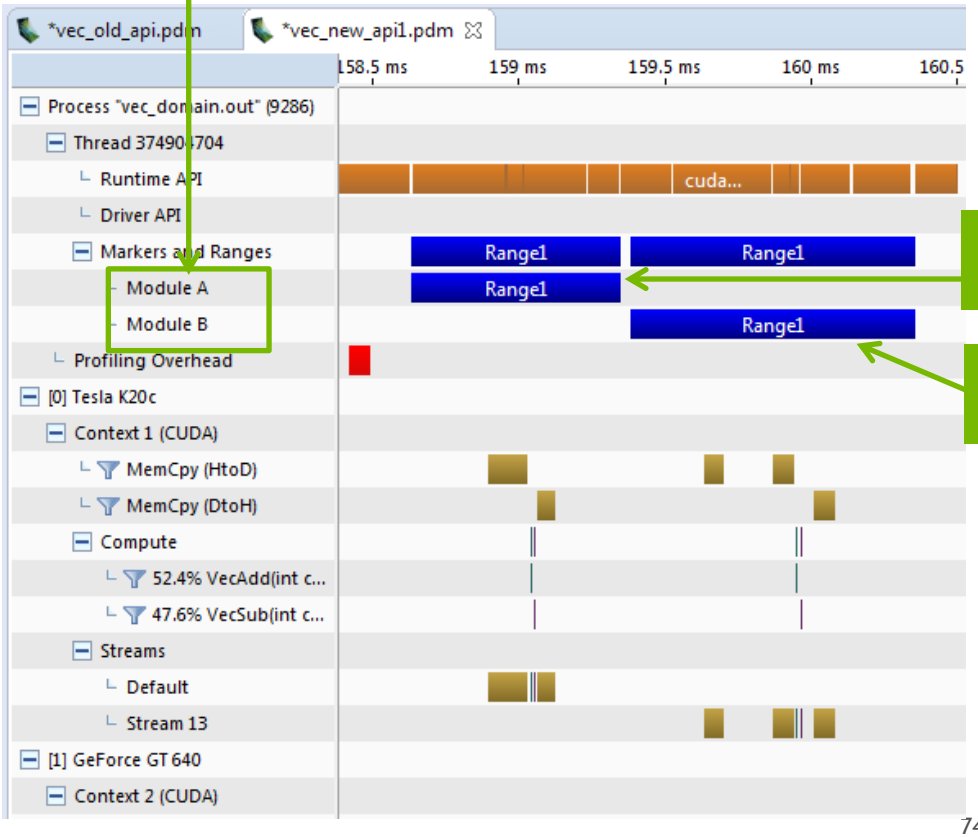
NVTX V2

Visual Profiler

Same range names from different modules

Domain names





From domain 1

From domain 2

CPU PROFILING

CPU PROFILING

Analysis | GPU Details | CPU Details | Console | Settings

TOTAL

Event	%	Time
bench_staggeredleapfrog2_	95.833%	689.695 ms
CCTKi_BindingsFortranWrapperBenchADM	95.833%	689.695 ms
CCTK_CallFunction	95.833%	689.695 ms
__open_nocancel	1.389%	9.996 ms
InitialFlat	1.389%	9.996 ms
__c_mcopy8	1.389%	9.996 ms

```
      FSDY = 4.D0/DY
!$OMP PARALLEL DO
      DO 100 J=1,N
```

Multiple markers at this line
- Generated 6 prefetch instructions for the loop
- Generated vector sse code for the loop
- Generated 5 alternate versions of the loop
- 2 loop-carried redundant expressions removed with 2 operations and 4 arrays
- Intensity = 1.93

```
  100 CONTINUE


C
C     PERIODIC CONTINUATION
C

      DO 110 J=1,N
```

OPENACC PROFILING

OPENACC PROFILING

OpenAcc->Driver API->Compute correlation

OpenAcc timeline

OpenAcc->Source Code correlation

OpenAcc Properties

NVIDIA Visual Profiler

File View Window Run Help

*NewSession1

291.5 ms 292 ms 292.5 ms 293 ms 293.5 ms 294 ms 294.5 ms 295 ms 295.5 ms

Driver API
cuMe... cuStreamSynch... cuStreamSynchronize cuStreamSynchronize

Thread 1101162240
OpenACC
acc_enter_data@acc_openmp.cpp:34 acc_compute_construct@acc_openmp.cpp:34
acc_enqueue_... acc_implicit_wait@acc_open... acc_implicit_wait@acc_openmp.cpp:34

Runtime API
Driver API
cuMemcpyHto... cuStreamSynchronize cuStreamSynchronize

Thread 1084376832
OpenACC
acc_compute_construct@acc_openmp.cpp:34 acc_exit_data@acc_openmp.cpp:36
acc_enque... acc_implicit_wait@acc_openm... acc_enqueue_download@acc_openmp.cpp:36 acc_implicit_wait@...
Driver API
cuLaunchK... cuStreamSynchronize cuStreamSynchronize cuStreamSynchronize

Profiling Overhead
[0] Tesla K20c
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% _Z8runSax...
Streams
Default

Analysis GPU Details Console Settings acc_openmp.cpp

```
        float *sub_x = x + (i * N);
        float *sub_y = y + (i * N);

        if (loop == 0 && omp_get_thread_num() == 0) {
            printf("%d OMP threads\n", omp_get_num_threads());
            fflush(NULL);
        }

#pragma acc kernels
        for (j = 0; j < N; ++j) {
            sub_y[j] = a*sub_x[j];
        }
    }
    return 0;
}

/* test main */

int main(int argc, char **argv)
{
    int M = 10;
```

Properties

acc_enqueue_launch@acc_openmp.cpp:34

Start	291.463 ms (291,463,334 ns)
End	291.717 ms (291,716,557 ns)
Duration	253.223 µs
Event Kind	acc_enqueue_launch
Version	1
Device Type	acc_device_nvidia
Device Number	0
Thread ID	6
Async	-1
Async Map	16
Source Information	
Source File	[redacted]acc_openmp/acc_openmp.cpp
Function Name	_Z8runSaxpyiiiPfS_f
Start Line Number	34
End Line Number	34

OTHER PRESENTATIONS

CUDA 8.0 features:

- S6224 - Featured Presentation: CUDA 8 and Beyond

Unified memory:

- S6216 - The Future of Unified Memory
- S6134 - High Performance and Productivity with Unified Memory and OpenACC: A LBM Case Study

Tools presentations:

- S6615 - Developer Tools Arsenal for Tegra Platforms
- S6784 - Maximize OpenACC Performance with the PGPROF Profiler
- S6531 - CUDA® Debugging Tools in CUDA 8
- S6111 - NVIDIA CUDA® Optimization with NVIDIA Nsight™ Eclipse Edition: A Case Study
- S6112 - NVIDIA CUDA® Optimization with NVIDIA Nsight™ Visual Studio Edition: A Case Study

REFERENCES

NVIDIA toolkit documentation:

- http://docs.nvidia.com/

Pascal architecture:

- https://devblogs.nvidia.com/parallelforall/inside-pascal/

PC sampling blog:

- https://devblogs.nvidia.com/parallelforall/cuda-7-5-pinpoint-performance-problems-instruction-level-profiling/

GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

THANK YOU

JOIN THE CONVERSATION

#GTC16

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join

PRESENTED BY

NVIDIA.