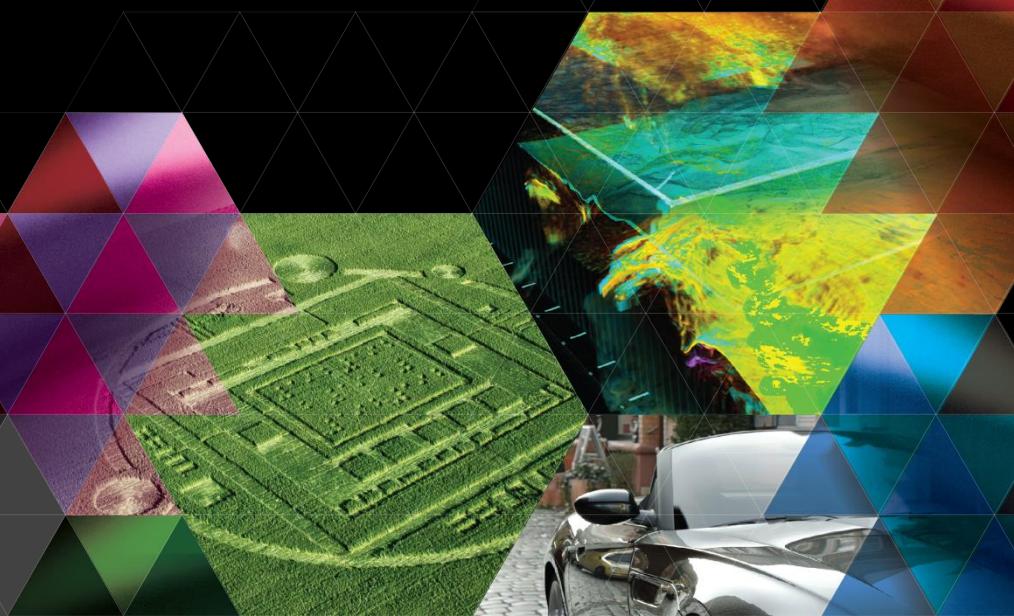


CUDA PROFILING TOOLS



GRAPHICAL AND COMMAND-LINE PROFILING TOOLS

- NVIDIA® Visual Profiler

- Standalone (`nvvp`)   

- Integrated into NVIDIA® Nsight™ Eclipse Edition (`nsight`)  

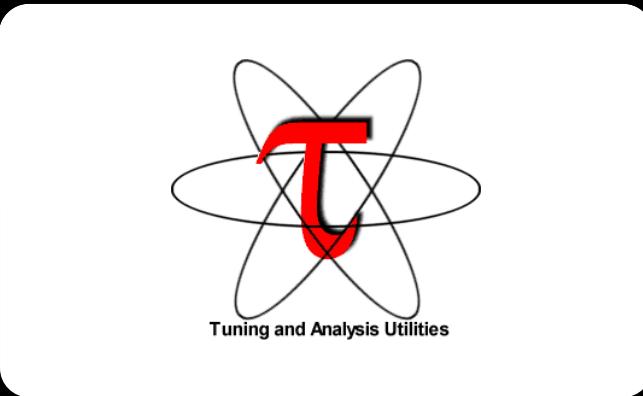
- `nvprof`     *

- NVIDIA® Nsight™ Visual Studio Edition 

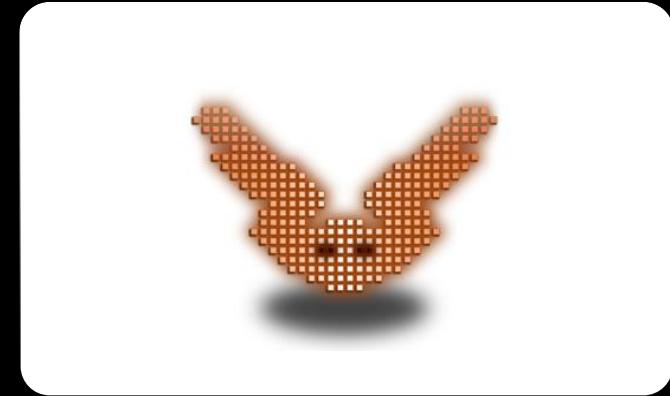
- Old environment variable based command-line profiler still available     *

* Android CUDA APK profiling not supported (yet)

3RD PARTY PROFILING TOOLS



TAU Performance System ®



VampirTrace



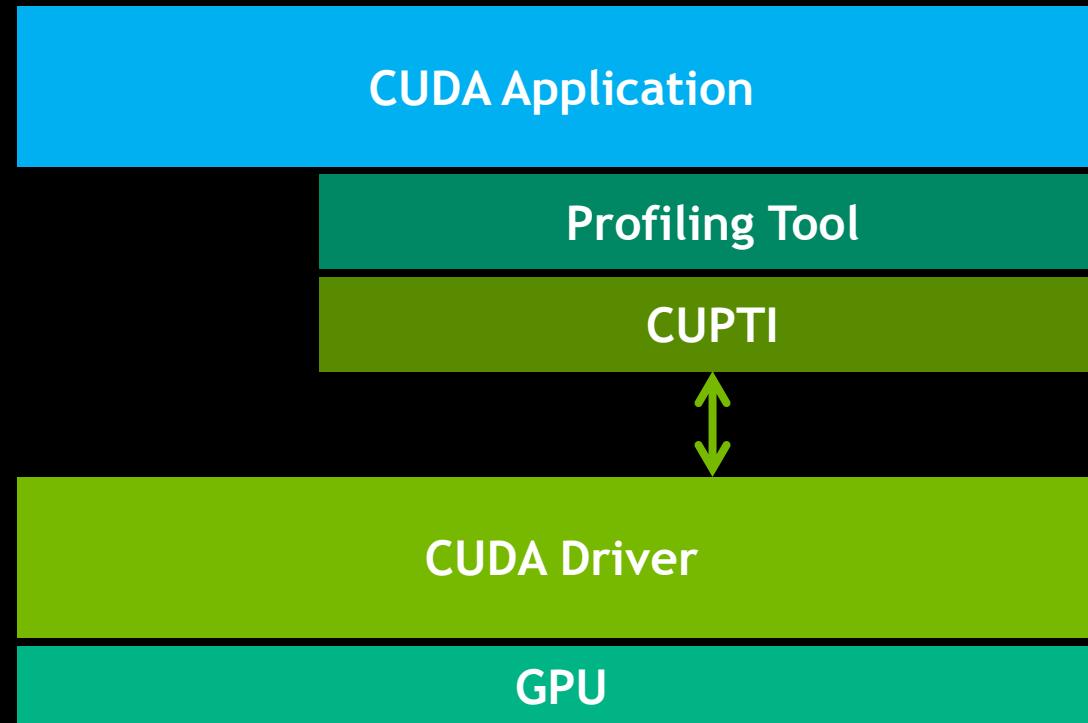
PAPI CUDA Component



HPC Toolkit

CUDA PROFILING TOOLS INTERFACE(CUPTI)

- Enables creation of profiling and tracing tools that target CUDA applications
- Designed for tools vendors, not CUDA developers

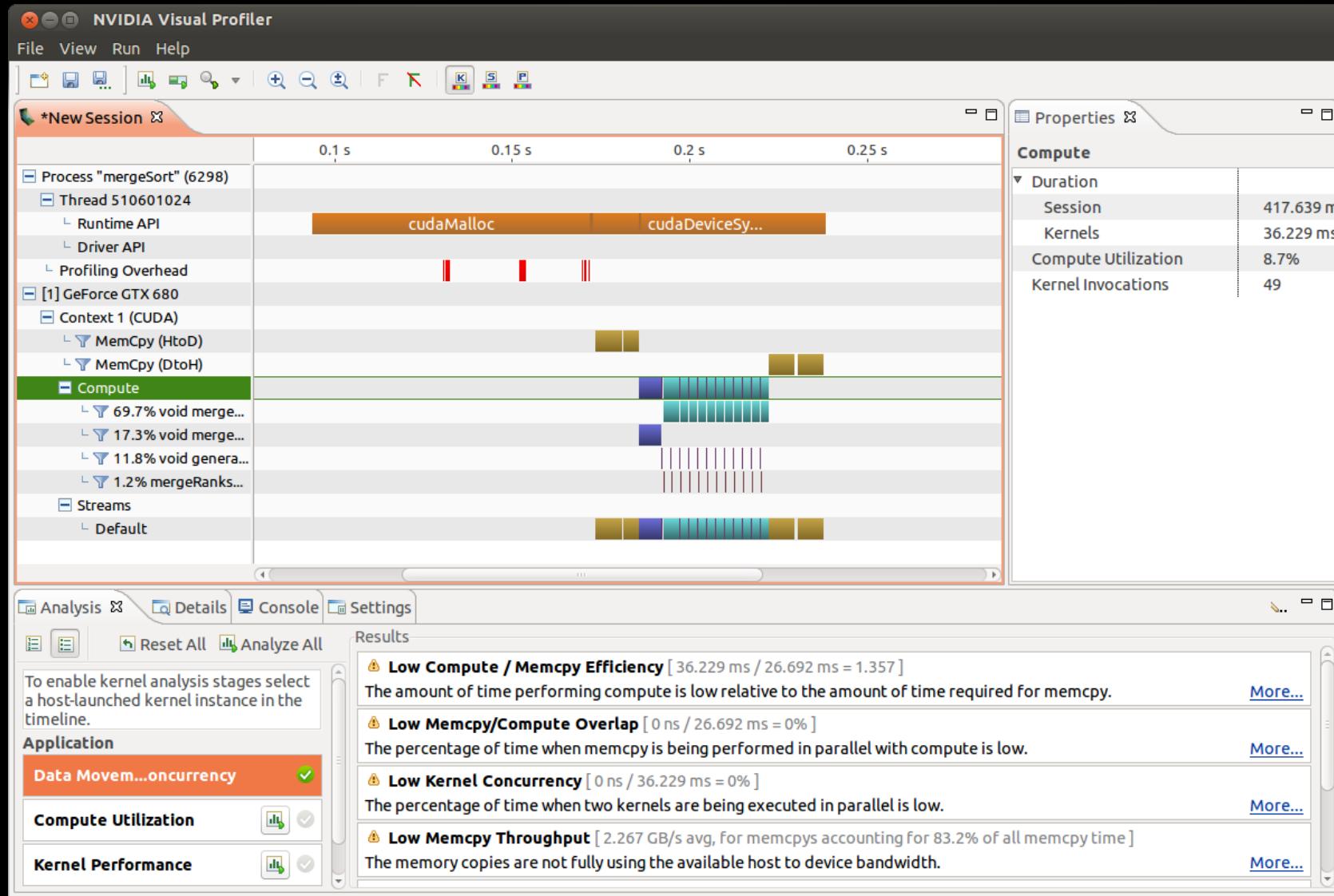


PERFORMANCE OPPORTUNITIES

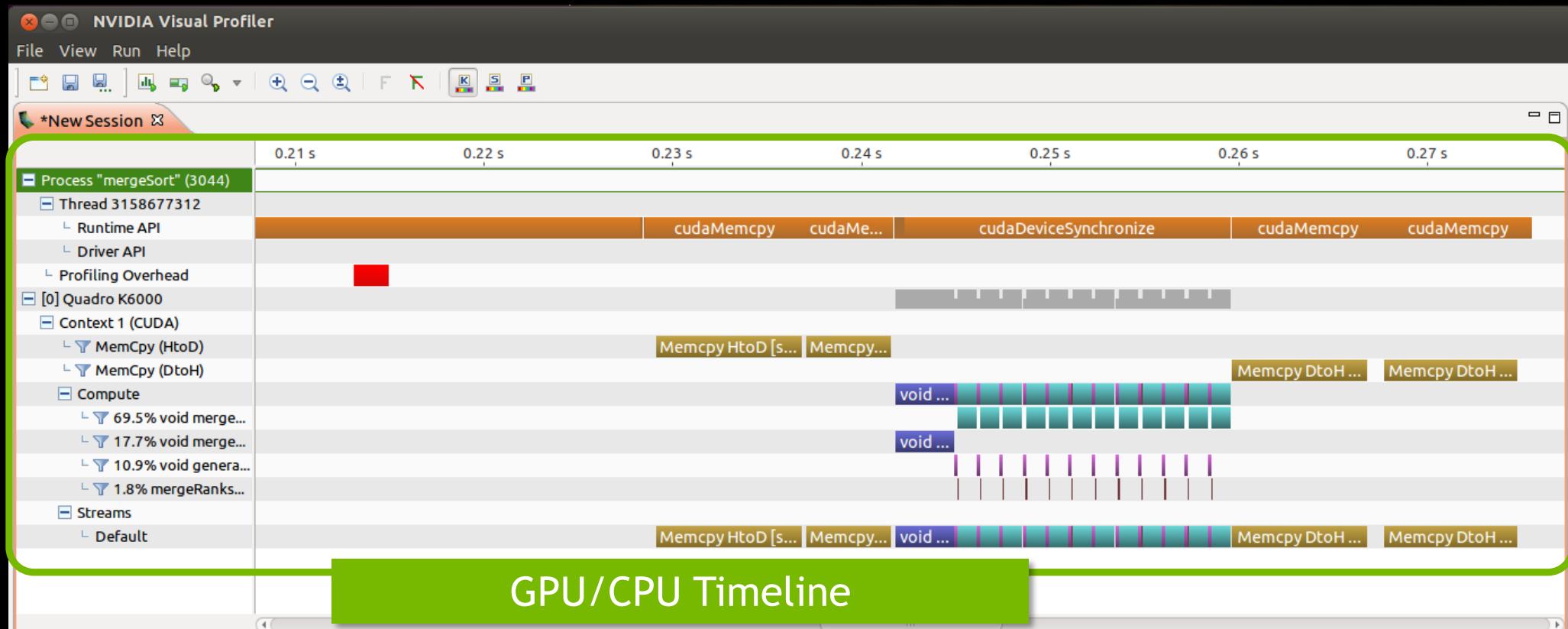
- Application level opportunities
 - Overall GPU utilization and efficiency
 - Memory copy efficiency
- Kernel level opportunities
 - Instruction and memory latency hiding/reduction
 - Efficient use of memory bandwidth
 - Efficient use of compute resources



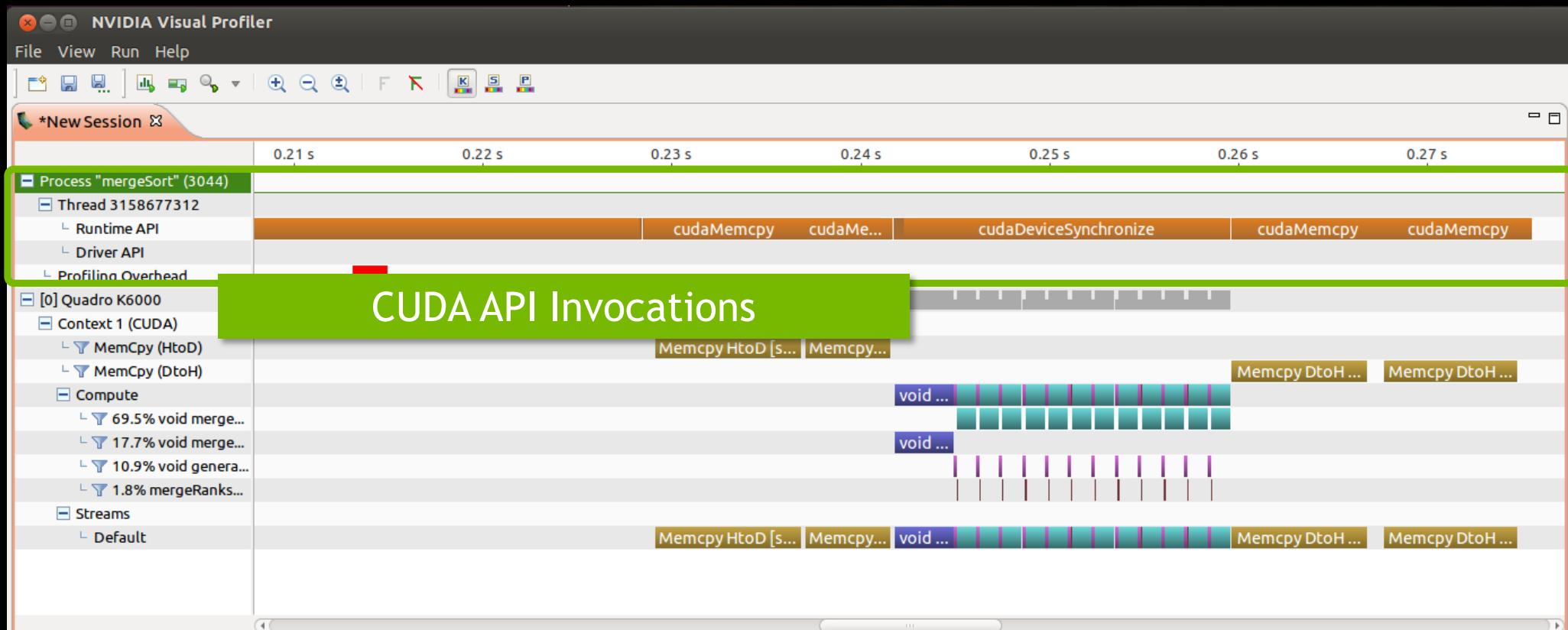
VISUAL PROFILER



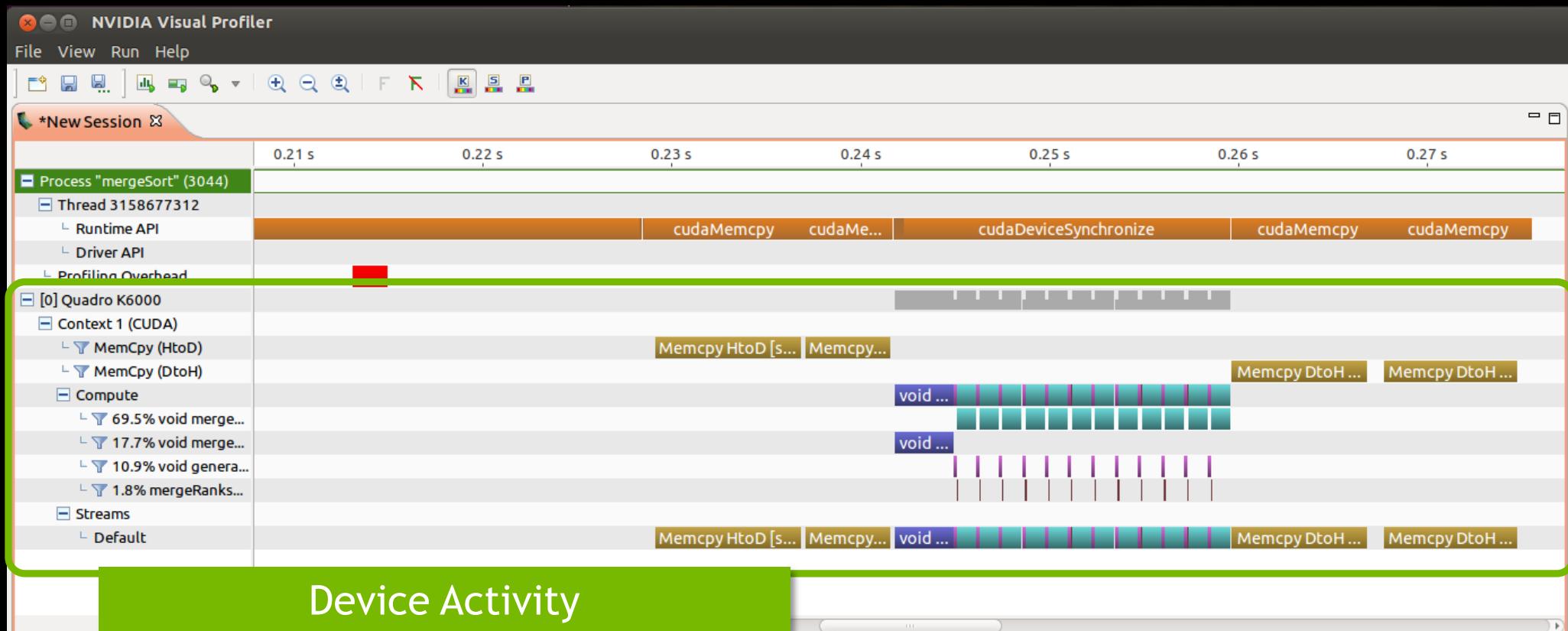
VISUAL PROFILER



VISUAL PROFILER



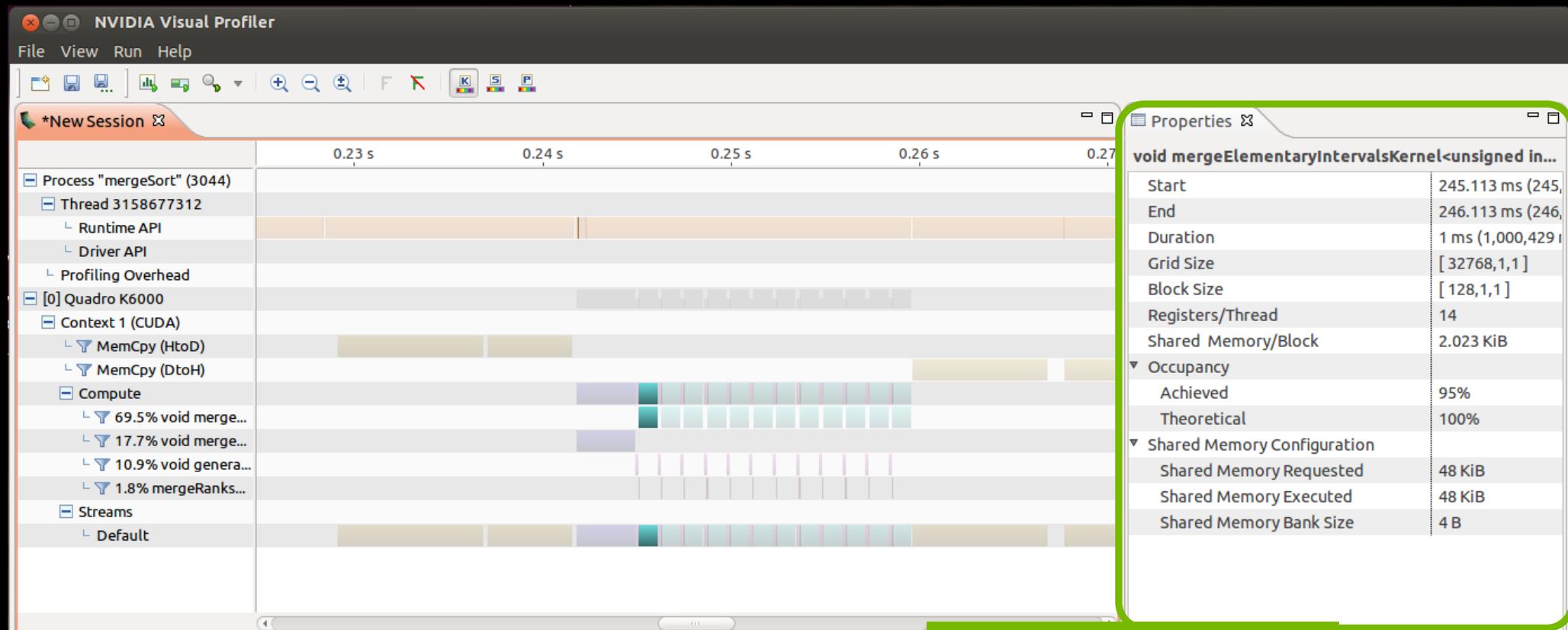
VISUAL PROFILER



GPU

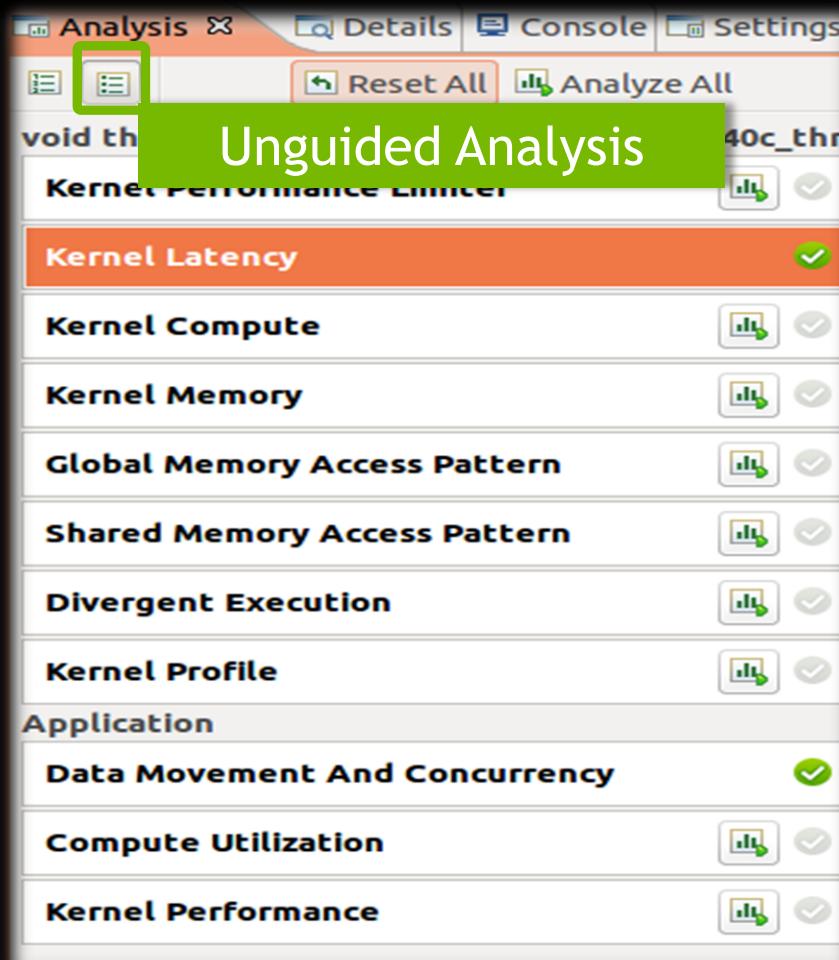
TECHNOLOGY
CONFERENCE

VISUAL PROFILER



Kernel Properties

VISUAL PROFILER



VISUAL PROFILER

The screenshot shows the Visual Profiler application window. At the top, there is a menu bar with tabs: Analysis (selected), Details, Console, and Settings. Below the menu is a toolbar with icons for Analysis, Details, Console, and Settings, followed by an 'Export PDF Report' button. The main content area is titled 'Guided Analysis' and contains four numbered steps: 1. Compute, 2. Performance-Critical Kernels, 3. Compute, Bandwidth, or Latency Bound, and 4. Compute Resources. Step 1 is highlighted with a green background. Step 4 has a tooltip explaining GPU compute resources limit performance when insufficient or poorly utilized. Below step 4 is a large orange button labeled 'Show Kernel Profile'. A detailed description follows, explaining that the kernel profile shows execution count, inactive threads, and predicated threads for each source and assembly line of the kernel, helping to pinpoint inefficient resource use due to divergence and predication. At the bottom is a 'Rerun Analysis' button with a tooltip stating that modification of the kernel requires rerunning the application to update the analysis.

Analysis X Details Console Settings

Export PDF Report

1. C Guided Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

4. Compute Resources

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized.

Show Kernel Profile

The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Rerun Analysis

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

VISUAL PROFILER: APPLICATION LEVEL OPPORTUNITIES

The screenshot shows the NVIDIA Visual Profiler interface. The top navigation bar includes tabs for Analysis, Details, Console, and Settings. Below the navigation bar, there are two main sections: "1. CUDA Application Analysis" and "2. Check Overall GPU Usage". The "1. CUDA Application Analysis" section contains a summary message about potential performance issues related to GPU utilization and provides a link to "Examine Individual Kernels". The "2. Check Overall GPU Usage" section contains a summary message about overall GPU usage. The right side of the interface displays a list of performance results under the heading "Results". These results include:

- Low Compute / Memcpy Efficiency** [17.639 ms / 26.429 ms = 0.667] More...
- Low Memcpy/Compute Overlap** [0 ns / 17.639 ms = 0%] More...
- Low Kernel Concurrency** [0 ns / 17.639 ms = 0%] More...
- Low Memcpy Throughput** [2.296 GB/s avg, for memcpys accounting for 82.9% of all memcpy time] More...
- Low Memcpy Overlap** [0 ns / 12.211 ms = 0%] More...
- Low Compute Utilization** [17.639 ms / 484.212 ms = 3.6%] More...

Below the results, there is a section titled "Compute Utilization" with a note about the device timeline.

Profiler Suggestions

VISUAL PROFILER: KERNEL LEVEL OPPORTUNITIES

The screenshot shows the NVIDIA Visual Profiler interface. At the top, there are tabs for Analysis, Details, Console, and Settings. Below the tabs, there are two main sections: "1. CUDA Application Analysis" and "2. Performance-Critical Kernels". A large green callout box highlights the "Kernel Optimization Priorities" section under "2. Performance-Critical Kernels". This section contains a table titled "Results" with the following data:

Rank	Description
100	[12 kernel instances] void mergeElementaryIntervalsKernel<unsigned int=1>(unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int*)
25	[1 kernel instances] void mergeSortSharedKernel<unsigned int=1>(unsigned int*, unsigned int*, unsigned int*, unsigned int*, unsigned int)
15	[12 kernel instances] void generateSampleRanksKernel<unsigned int=1>(unsigned int*, unsigned int*, unsigned int*, unsigned int, unsigned int, unsigned int, unsigned int)
1	[10 kernel instances] mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int, unsigned int, unsigned int, unsigned int)
1	[2 kernel instances] mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int, unsigned int, unsigned int)
1	[6 kernel instances] mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int, unsigned int, unsigned int, unsigned int)
1	[6 kernel instances] mergeRanksAndIndicesKernel(unsigned int*, unsigned int*, unsigned int, unsigned int, unsigned int, unsigned int)

A green button labeled "Kernel Priorities" is located at the bottom right of the highlighted area.

WHAT'S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling

WHAT'S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling

UNIFIED MEMORY

CUDA 6 Code with Unified Memory

```
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);  
}
```

- ← Allocate Unified Memory
- ← Page faults, copy buffer (DtoH)
- ← Copy buffer (HtoD)
- ← Page faults, copy buffer (DtoH)

NVPROF: UNIFIED MEMORY

- Use **--unified-memory-profiling** to collect Unified Memory profiling data.

```
$ nvprof --unified-memory-profiling per-process-device UnifiedMemoryStreams
...
==25394== Profiling application: UnifiedMemoryStreams
==25394== Profiling result:
==25394== Unified Memory profiling result:
Device "Quadro K6000 (0)"
```

	Count	Avg	Min	Max
Host To Device (bytes)	13	5.2297e+07	0	1.10e+08
Device To Host (bytes)	140	2.5555e+07	0	1.10e+08
CPU Page faults	140	13728.75	0	26925

- Use **--print-gpu-trace** for detailed information

NVPROF: UNIFIED MEMORY

```
$ nvprof --unified-memory-profiling per-process-device --print-gpu-trace UnifiedMemoryStreams
...
==25394== Profiling application: UnifiedMemoryStreams
==25394== Profiling result:
      Start ...      Unified Memory  Name
348.87ms ...           0 B [Unified Memory Memcpy DtoH]
348.87ms ...           0 [Unified Memory CPU page faults]
350.94ms ...       610304 B [Unified Memory Memcpy DtoH]
350.94ms ...        149 [Unified Memory CPU page faults]
...
608.23ms ...           0 B [Unified Memory Memcpy HtoD]
610.30ms ...       6320128 B [Unified Memory Memcpy HtoD]
...
632.71ms ...           - void gemv2N_kernel_val<double, int=128, int=8, int=4, int=4> ...
633.00ms ...           24876 [Unified Memory CPU page faults]
633.00ms ...       101892096 B [Unified Memory Memcpy DtoH]
```

Page faults, copy buffer
(DtoH)

NVPROF: UNIFIED MEMORY

```
$ nvprof --unified-memory-profiling per-process-device --print-gpu-trace ./UnifiedMemoryStreams
...
==25394== Profiling application: UnifiedMemoryStreams
==25394== Profiling result:
      Start ...      Unified Memory  Name
348.87ms ...           0 B [Unified Memory Memcpy DtoH]
348.87ms ...           0 [Unified Memory CPU page faults]
350.94ms ...     610304 B [Unified Memory Memcpy DtoH]
350.94ms ...        149 [Unified Memory CPU page faults]
...
608.23ms ...           0 B [Unified Memory Memcpy HtoD]
610.30ms ...    6320128 B [Unified Memory Memcpy HtoD]
...
632.71ms ... - void gemv2N_kernel_val<double, int=128, int=8, int=4, int=4> ...
633.00ms ...       24876 [Unified Memory CPU page faults]
633.00ms ... 101892096 B [Unified Memory Memcpy DtoH]
```

Copy buffer (HtoD)

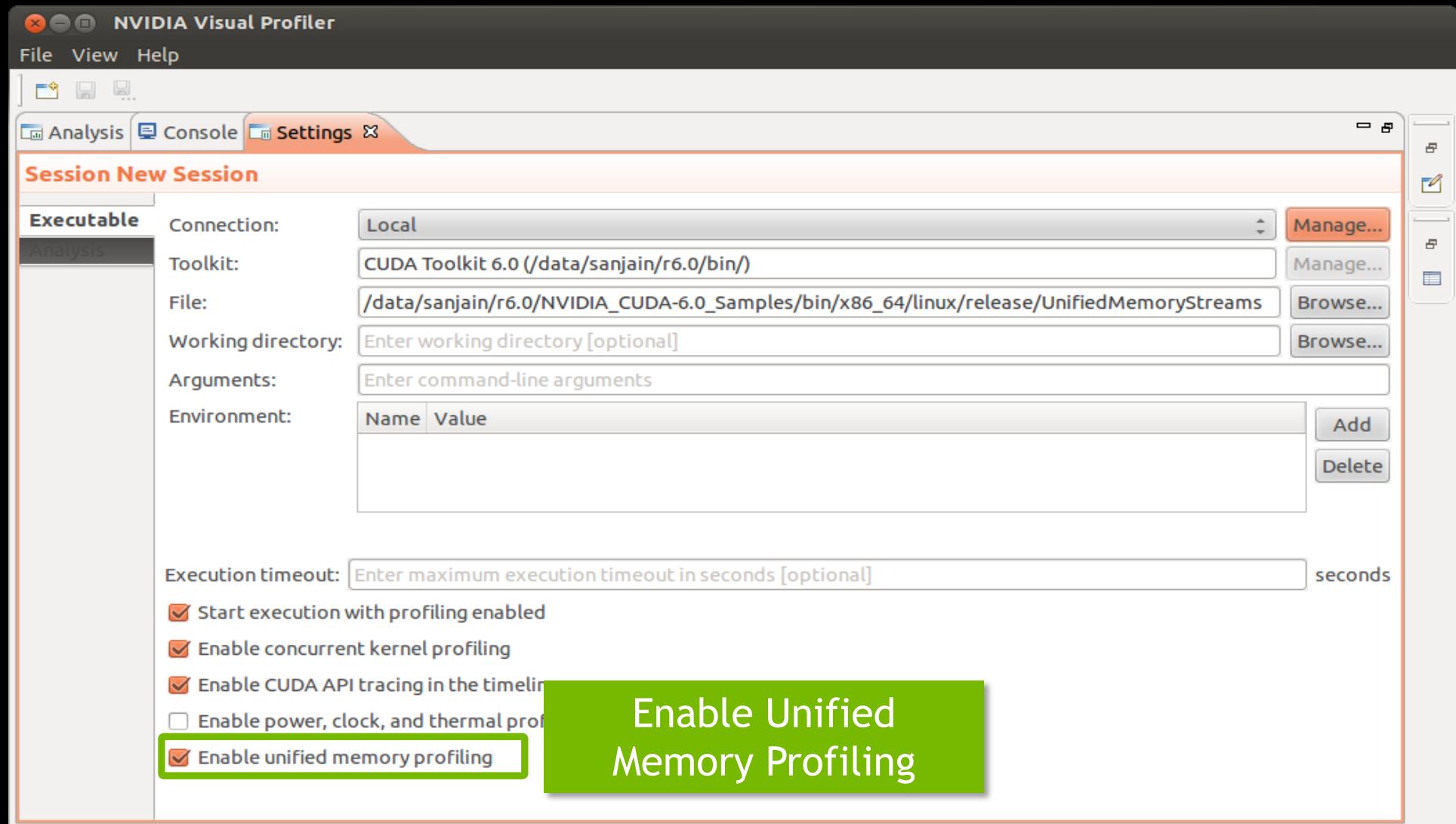
Kernel Launch

NVPROF: UNIFIED MEMORY

```
$ nvprof --unified-memory-profiling per-process-device --print-gpu-trace ./UnifiedMemoryStreams
...
==25394== Profiling application: UnifiedMemoryStreams
==25394== Profiling result:
      Start ...      Unified Memory  Name
348.87ms ...           0 B [Unified Memory Memcpy DtoH]
348.87ms ...           0 [Unified Memory CPU page faults]
350.94ms ...       610304 B [Unified Memory Memcpy DtoH]
350.94ms ...        149 [Unified Memory CPU page faults]
...
608.23ms ...           0 B [Unified Memory Memcpy HtoD]
610.30ms ...     6320128 B [Unified Memory Memcpy HtoD]
...
632.71ms ...           - void gemv2N_kernel_val<double, int=128, int=8, int=4, int=4> ...
633.00ms ...        24876 [Unified Memory CPU page faults]
633.00ms ...    101892096 B [Unified Memory Memcpy DtoH]
```

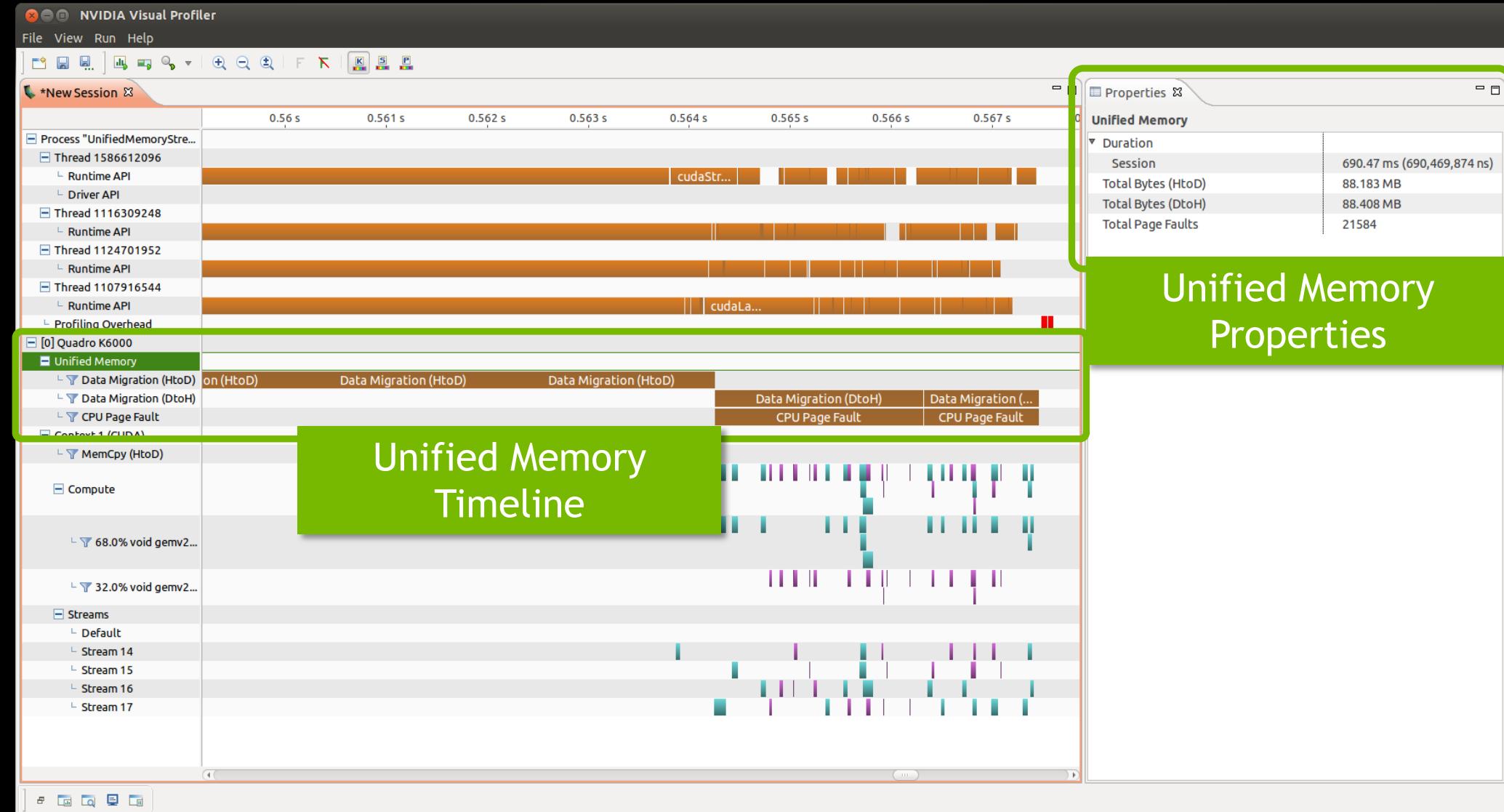
Page faults, copy buffer
(DtoH)

NVVP: UNIFIED MEMORY



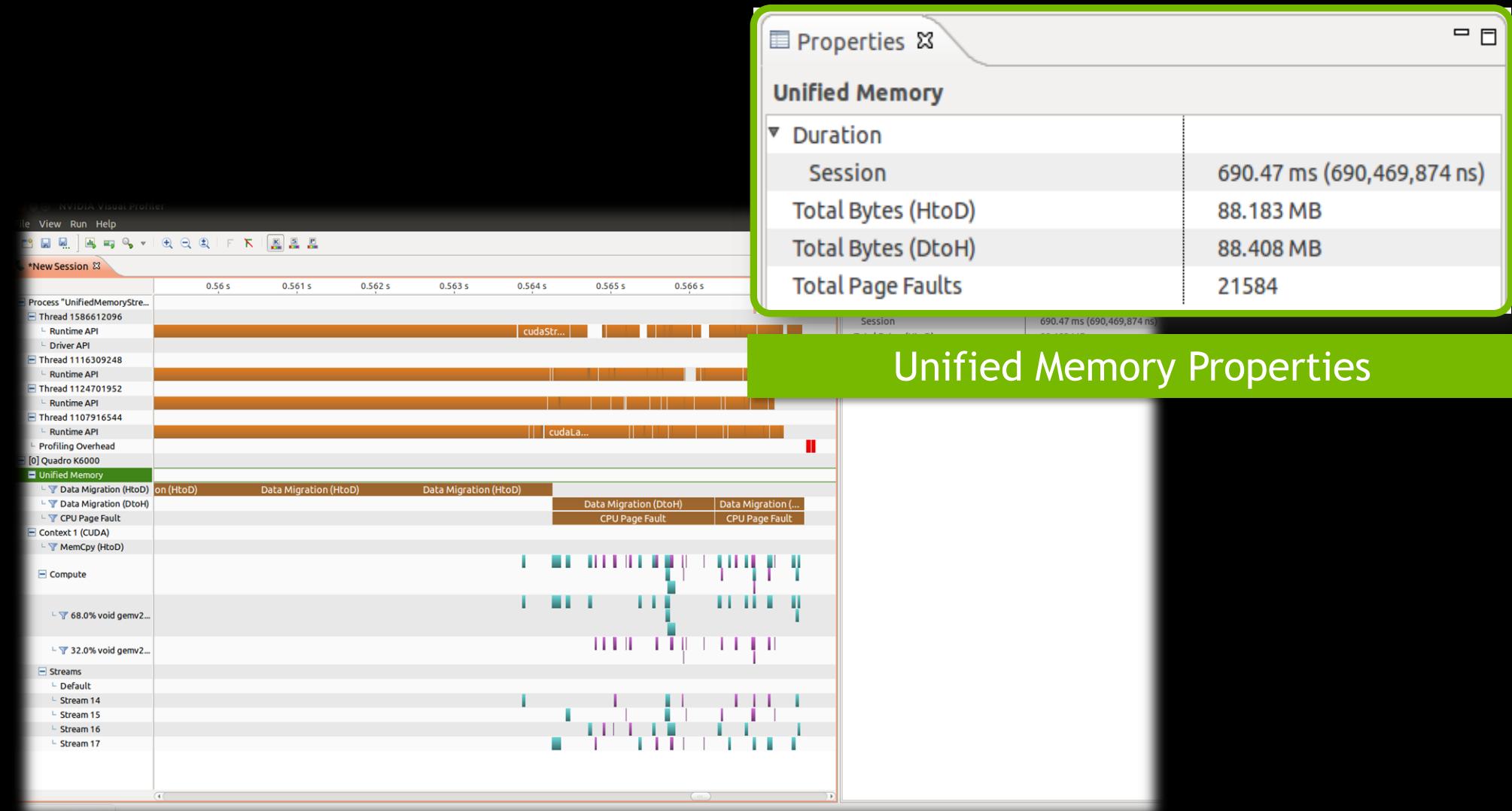
Enable Unified
Memory Profiling

NVVP: UNIFIED MEMORY

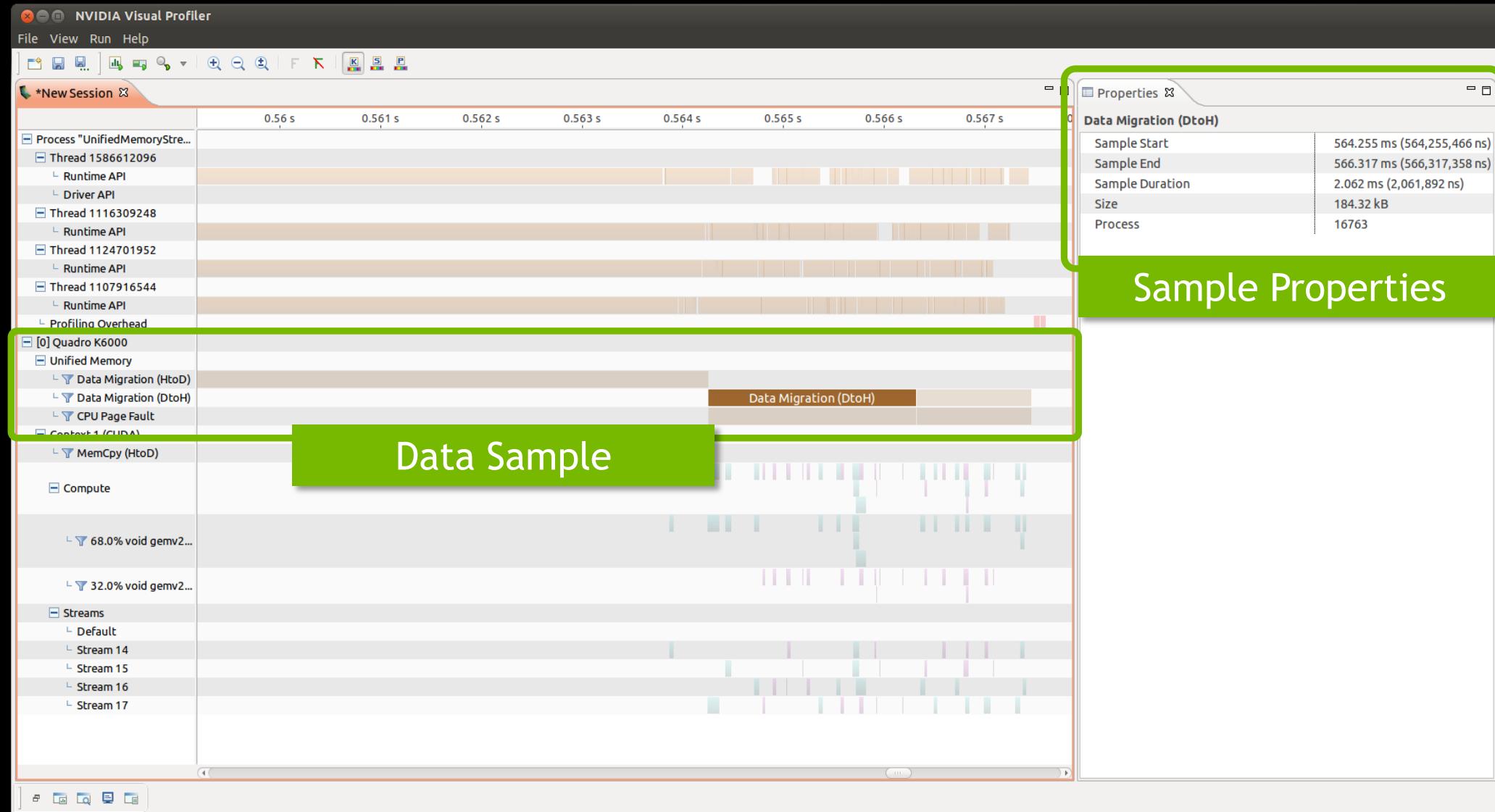


Unified Memory Properties

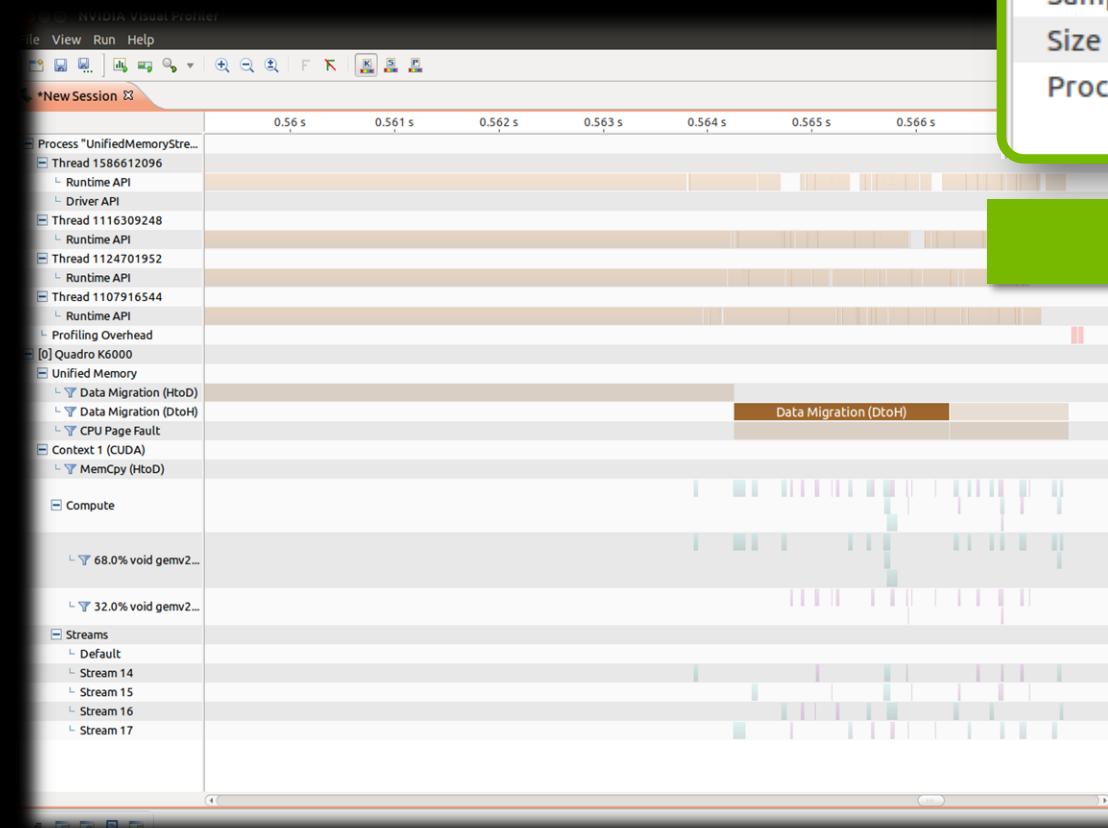
NVVP: UNIFIED MEMORY



NVVP: UNIFIED MEMORY



NVVP: UNIFIED MEMORY



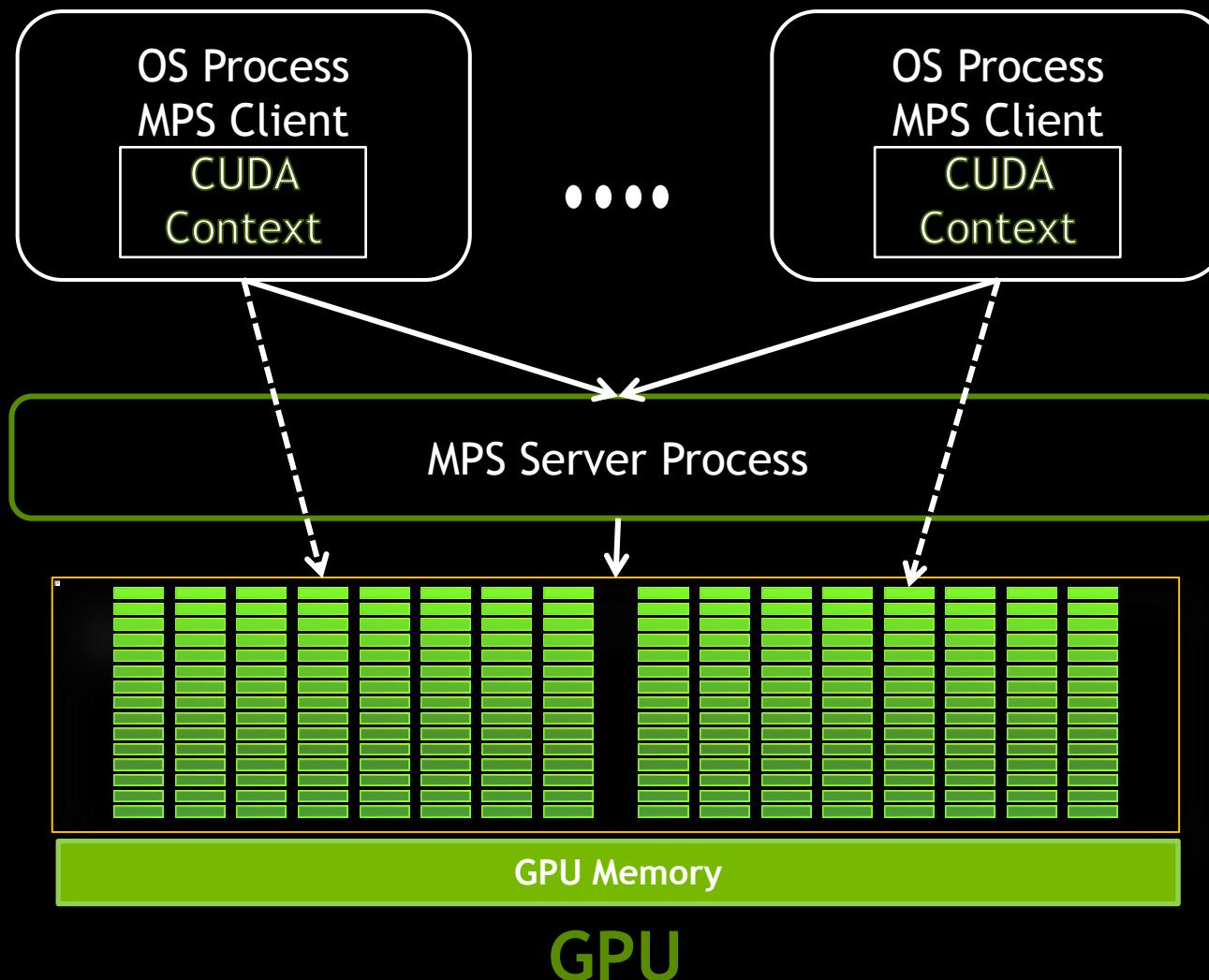
Data Migration (DtoH)	
Sample Start	564.255 ms (564,255,466 ns)
Sample End	566.317 ms (566,317,358 ns)
Sample Duration	2.062 ms (2,061,892 ns)
Size	184.32 kB
Process	16763

Sample Properties

WHAT'S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling

MULTI-PROCESS SERVICE (MPS)



MPS PROFILING WITH NVPROF

Step 1: Launch MPS daemon

```
$ nvidia-cuda-mps-control -d
```

Step 2: Run nvprof with --profile-all-processes

```
$ nvprof --profile-all-processes -o maxConcurrency_%p
===== Profiling all processes launched by user "sanjain"
===== Type "Ctrl-c" to exit
```

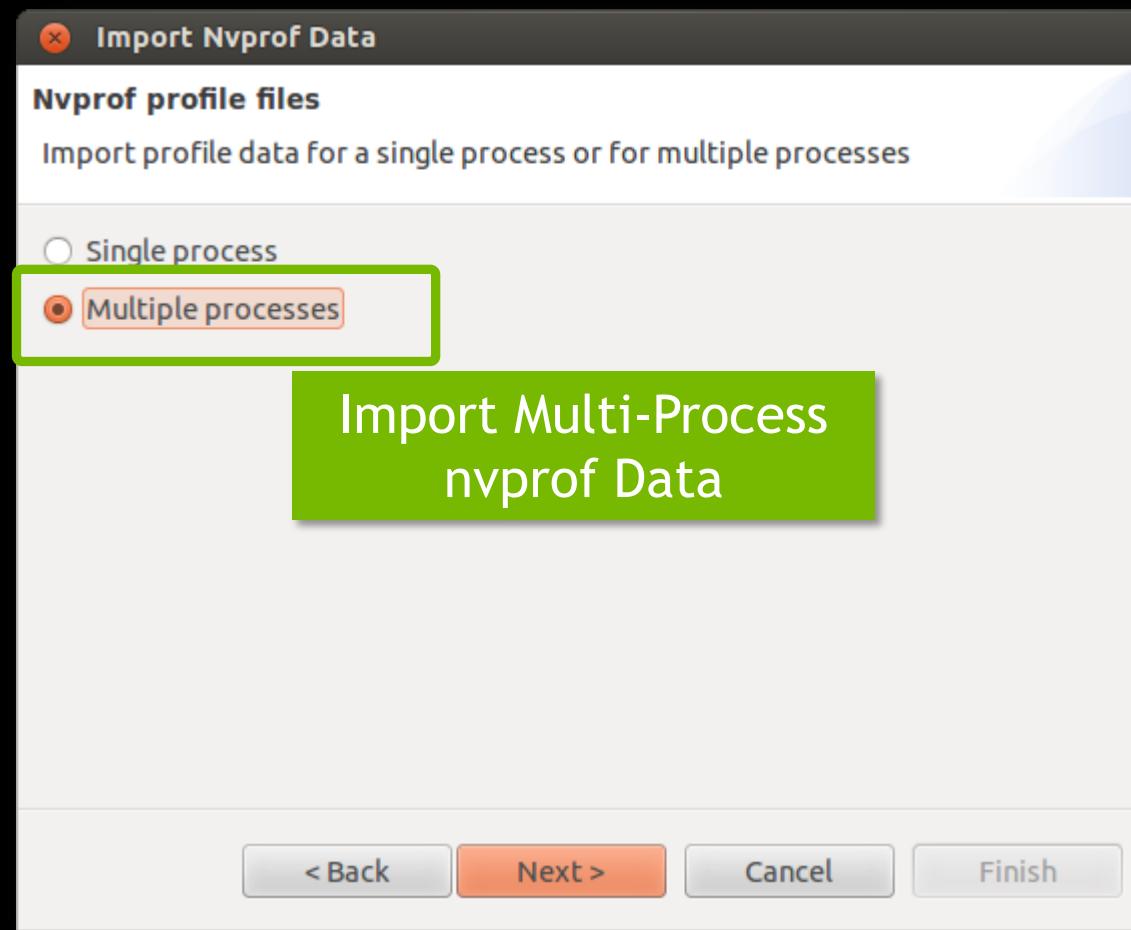
Step 3: Run application in different terminal normally

```
$ maxConcurrency
```

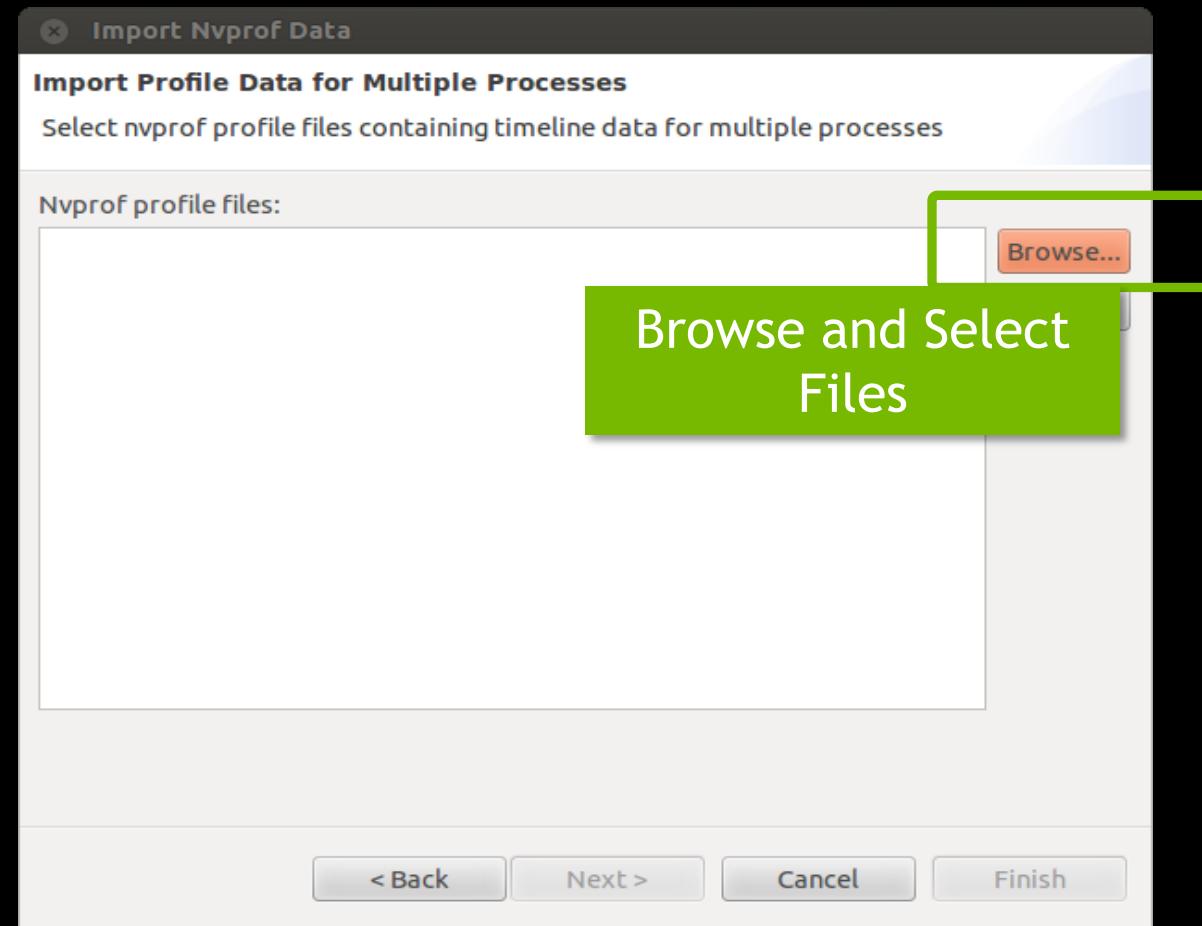
Step 4: Exit nvprof by typing Ctrl+C

```
==5844== NVPROF is profiling process 5844, command: maxConcurrency
==5840== NVPROF is profiling process 5840, command: maxConcurrency
...
==5844== Generated result file: /data/sanjain/r6.0/maxConcurrency_5844
==5840== Generated result file: /data/sanjain/r6.0/maxConcurrency_5840
^C===== Exiting nvprof
```

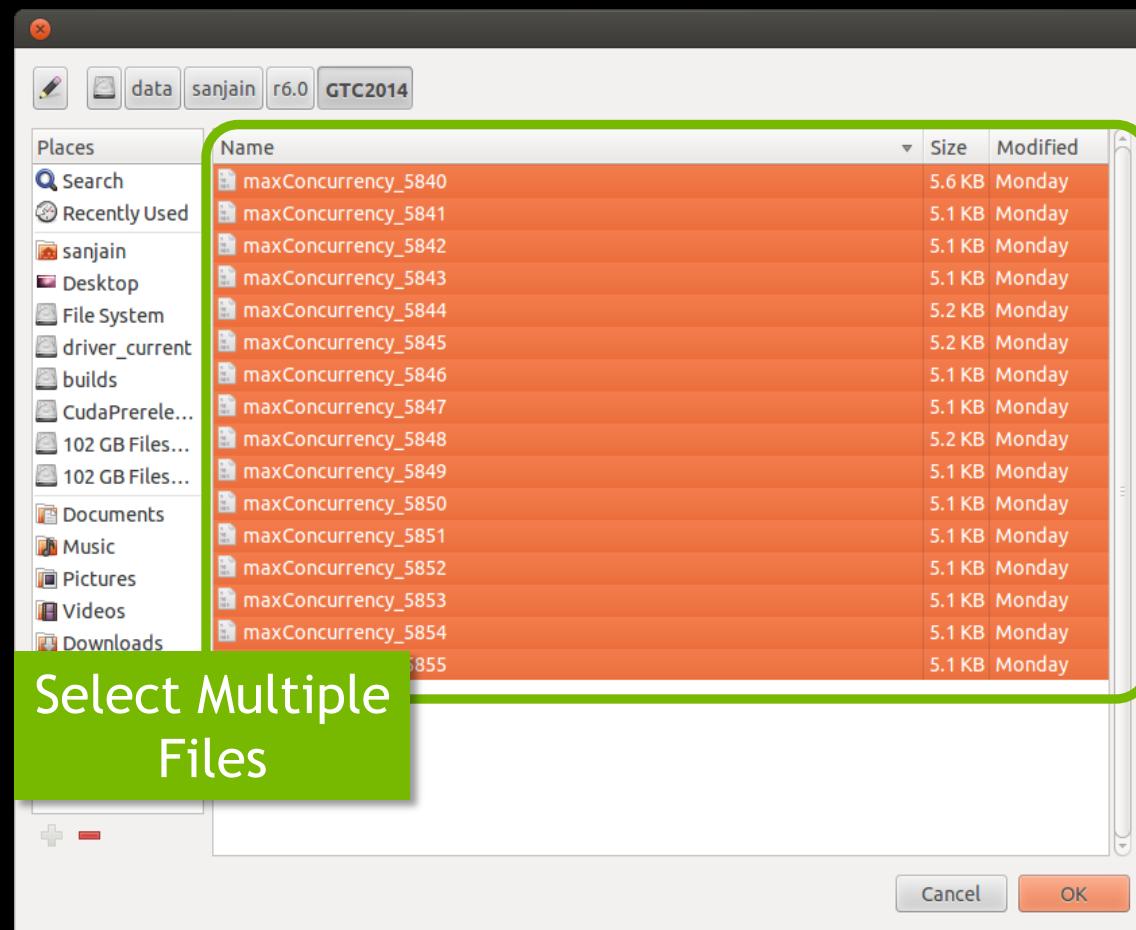
VIEWING MPS TIMELINE IN VISUAL PROFILER



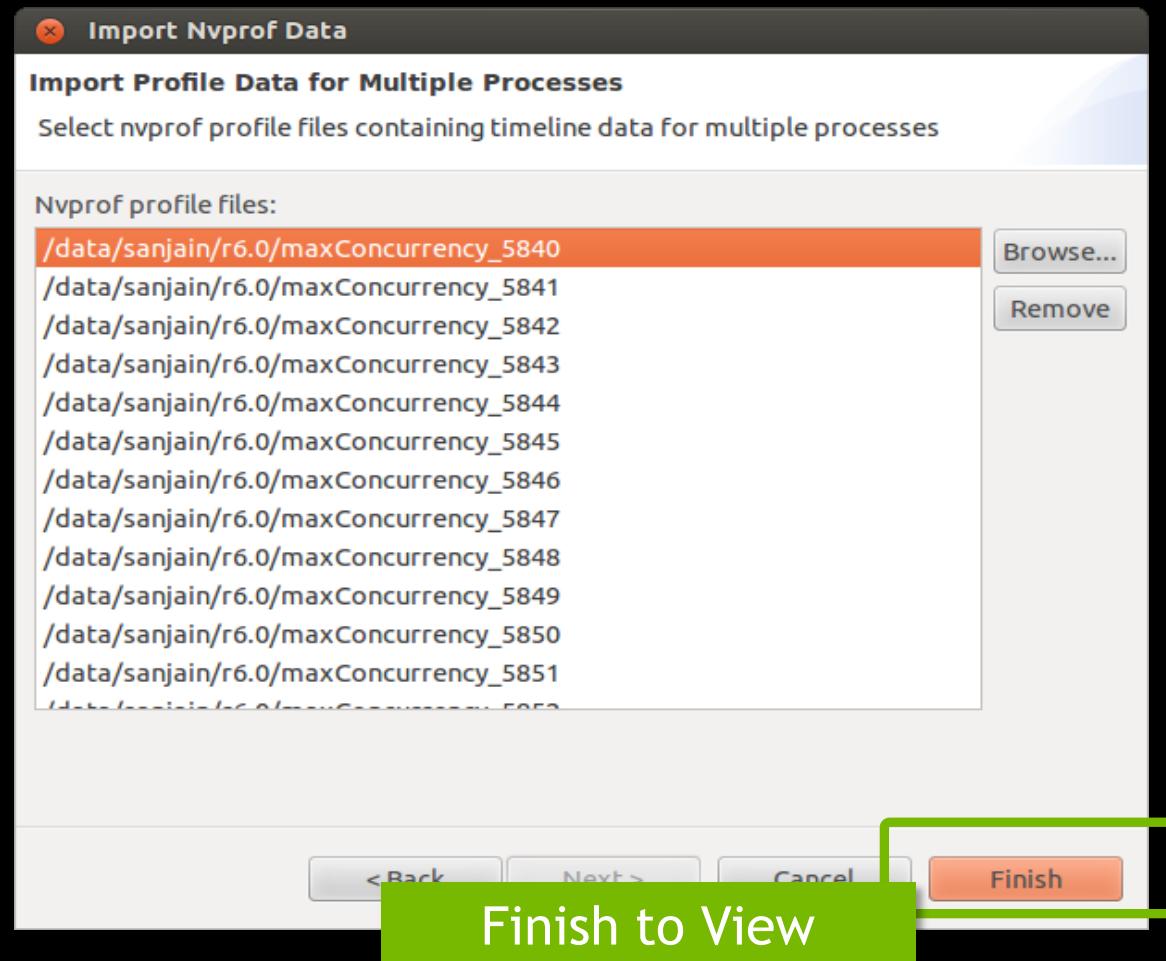
VIEWING MPS TIMELINE IN VISUAL PROFILER



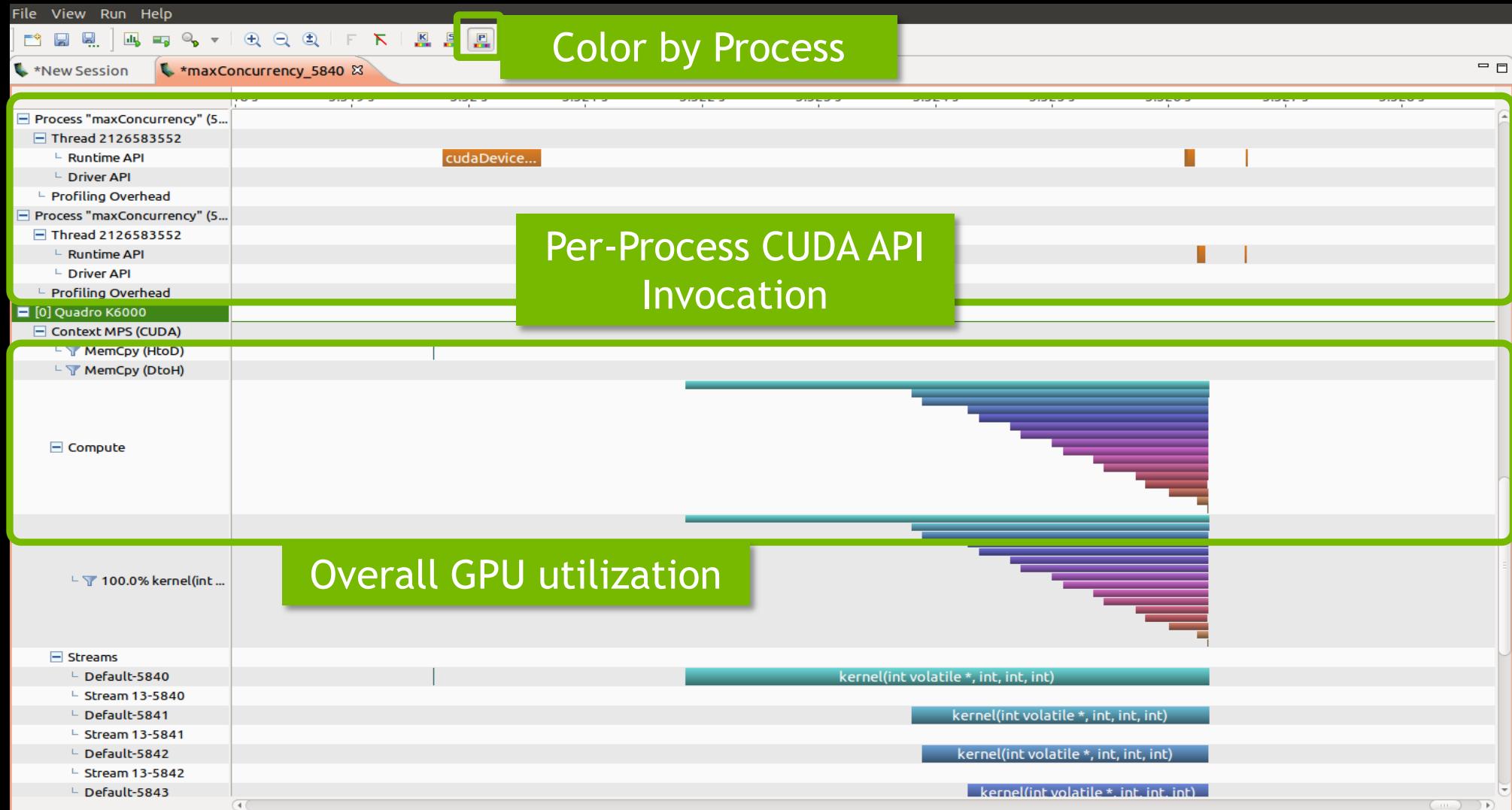
VIEWING MPS TIMELINE IN VISUAL PROFILER



VIEWING MPS TIMELINE IN VISUAL PROFILER



VIEWING MPS TIMELINE IN VISUAL PROFILER

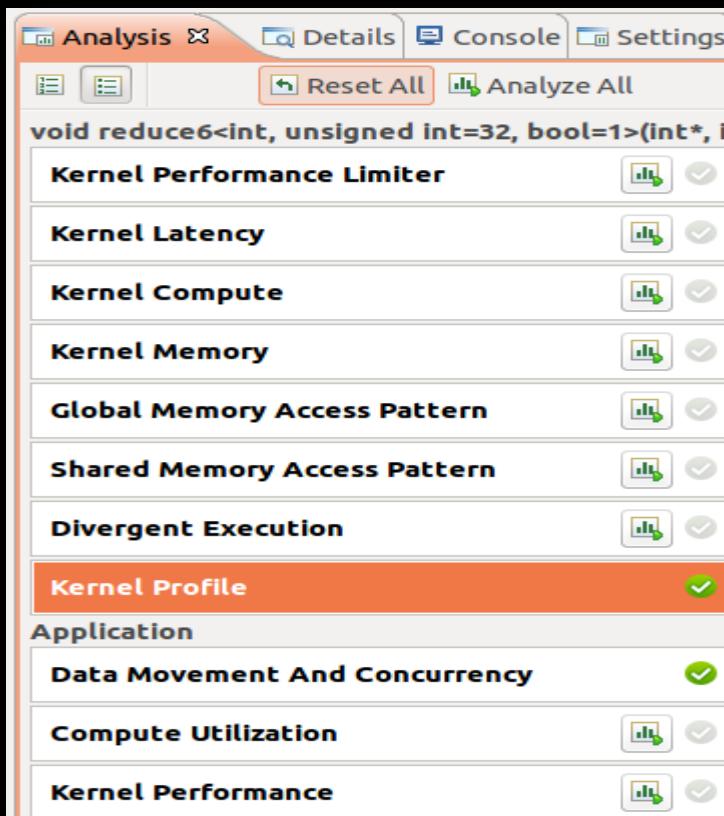


WHAT'S NEW IN 6.0?

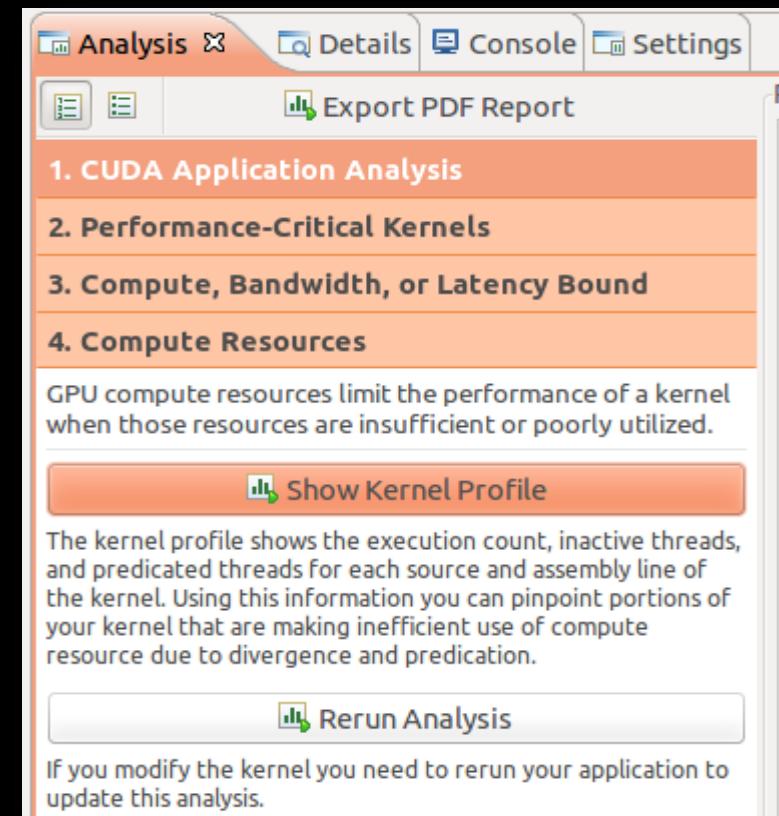
- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Shared Memory Source Level Analysis
- Mix of Instructions for a Kernel
- Inefficient SM Utilization Detection
- Remote Profiling

WHERE CAN I FIND THE NEW OPTION?

Unguided Analysis

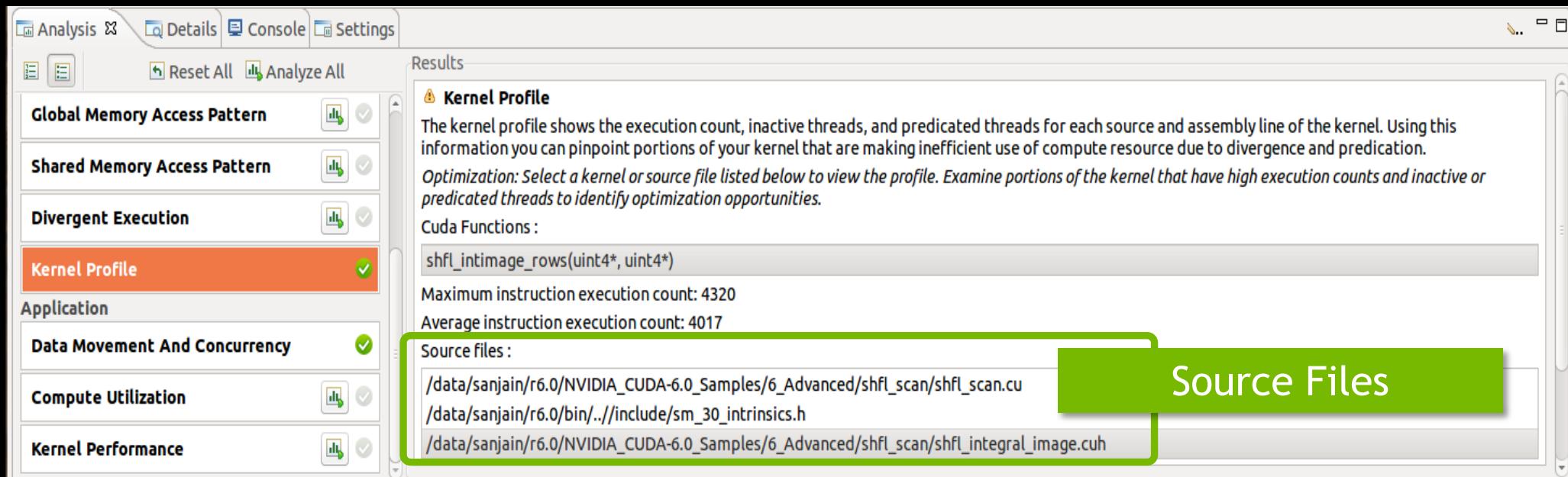


Guided Analysis



DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE



TIP: Don't forget to compile your application with `-lineinfo`

DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

The screenshot shows the 'Kernel Profile' section of the Nsight interface. The left sidebar lists analysis categories: Global Memory Access Pattern, Shared Memory Access Pattern, Divergent Execution, Kernel Profile (selected), Application, Data Movement And Concurrency, Compute Utilization, and Kernel Performance. The main pane displays the results for the selected 'Kernel Profile'. It includes a warning icon and the heading 'Kernel Profile'. A descriptive text states: 'The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.' An optimization tip follows: 'Optimization: Select a kernel or source file listed below to view the profile. Examine portions of the kernel that have high execution counts and inactive or predicated threads to identify optimization opportunities.' Below this, a 'Cuda Functions:' list shows 'shfl_intimage_rows(uint4*, uint4*)'. A green callout box highlights the 'Instruction Execution Count' for this function, with the text: 'Maximum instruction execution count: 4320' and 'Average instruction execution count: 4017'. The 'Source files:' list at the bottom includes '/data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shfl_scan.cu', '/data/sanjain/r6.0/bin/..//include/sm_30_intrinsics.h', and '/data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shfl_integral_image.cuh'.

Kernel Profile

The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Optimization: Select a kernel or source file listed below to view the profile. Examine portions of the kernel that have high execution counts and inactive or predicated threads to identify optimization opportunities.

Cuda Functions :

shfl_intimage_rows(uint4*, uint4*)

Maximum instruction execution count: 4320

Average instruction execution count: 4017

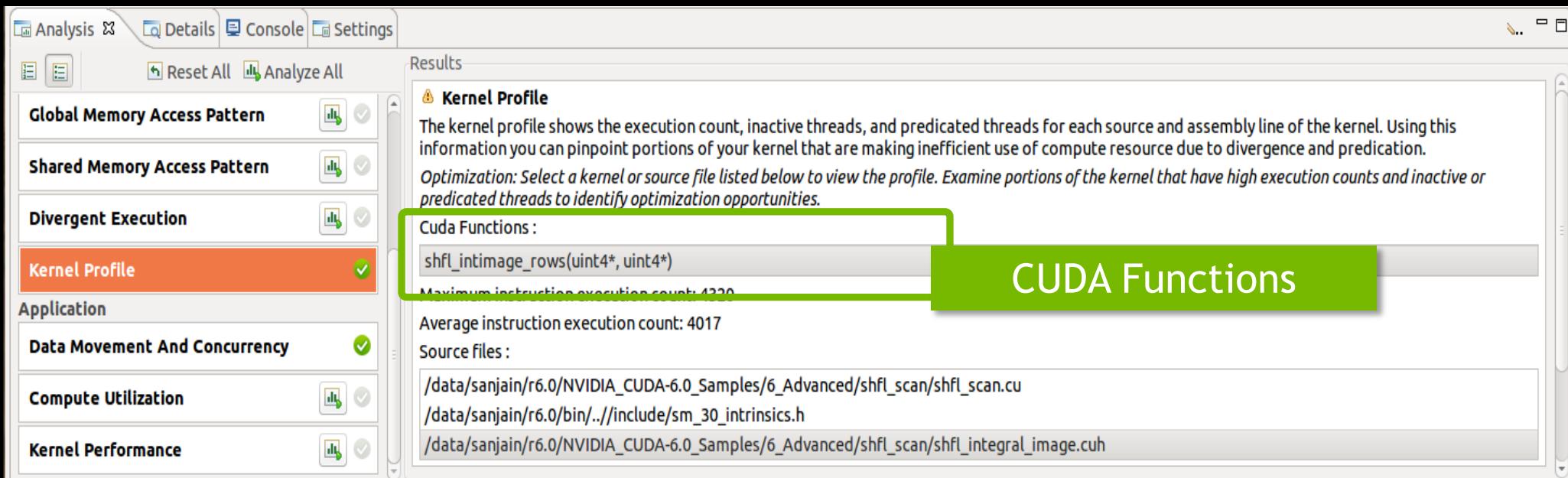
Source files :

/data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shfl_scan.cu
/data/sanjain/r6.0/bin/..//include/sm_30_intrinsics.h
/data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shfl_integral_image.cuh

Instruction Execution Count

DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE



DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

Buttons to Set the View

Line	Exec Count	File - /data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shf	Exec Count	Disassembly
104		// last thread in the warp holding sum of the warp	4320	BAR.SYNC 0x0;
105		// places that in shared		ISETP.NE.AND P5, PT, R12, RZ, PT;
106	43200	if(threadIdx.x % warpSize == warpSize-1)		SSY `(.L_8);
107		{		@P5 NOP.S;
108	8640	sums[warp_id] = result[15];		SHF.L.W R17, RZ, 0x2, R11;
109		}		MOV32I R20, 0x20;
110				LDS R18, [R17];
111	4320	__syncthreads();		SHFL.UP PT, R19, R18, 0x1, 0x0;
112				SEL R19, RZ, R19, P0;
113	8640	if(warp_id == 0)		IADD R18, R19, R18;
114		{		SHFL.UP PT, R19, R18, 0x2, 0x0;
115	2160	int warp_sum = sums[lane_id];		ISETP.LT.U32.AND P0, PT, R11, 0x20, PT;
116		#pragma unroll		SEL R19, RZ, R19, P1;
117				IADD R18, R19, R18;
118		for (int i=1; i<=32; i*=2)		SHFL.UP PT, R19, R18, 0x4, 0x0;
119		{		SEL R19, RZ, R19, P2;
120		int n=__shfl_up(warp_sum, i, 32);		IADD R18, R19, R18;
121				SHFL.UP PT, R19, R18, 0x8, 0x0;
122	14040	if(lane_id >= i) warp_sum += n;		SEL R19, RZ, R19, P3;
123		}		IADD R18, R19, R18;
124				SHFL.UP PT, R19, R18, 0x10, 0x0;
125	1080	sums[lane_id] = warp_sum;		SEL R19, RZ, R19, P4;
126		}		IADD R18, R19, R18;
127				SHFL.UP PT, R19, R18, R20, 0x0;
128	4320	__syncthreads();		SEL R11, RZ, R19, P0;
129				IADD R11, R11, R18;
130		int blockSum = 0;		STS.S [R17], R11;
131				.L_8:
132				BAR.SYNC 0x0;
133		// fold in unused warp		ISETP.LT.AND P0, PT, R12, 0x1, PT;
134	8640	if(warp_id >0)		@P0 BRA.U `(.L_9);
135		{		@!P0 LDS R11, [R16+-0x4];
136	3240	blockSum = sums[warp_id-1];		@!P0 IADD R26, R26, R11;
137		#pragma unroll		...
138				

DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

The screenshot shows a GPU debugger interface with two main panes. The left pane displays the CUDA source code for the kernel `shfl_intimage_rows`. The right pane shows the corresponding SASS assembly code. A green callout box labeled "CUDA Source" points to the source code, and another green callout box labeled "Corresponding SASS" points to the assembly code.

CUDA Source:

```
Line      Exec Count File - /data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shf
104          // last thread in the warp holding sum of the warp
105          // places that in shared
106          if (threadIdx.x % warpSize == warpSize-1)
107          {
108              sums[warp_id] = result[15];
109          }
110
111          __syncthreads();
112
113          if (warp_id == 0)
114          {
115              int warp_sum = sums[lane_id];
116
117              #pragma unroll
118
119                  for (int i=1; i<=32; i*=2)
120                  {
121                      int n = shfl_up(warp_sum, i, 32);
122
123                      if (lane_id >= i) warp_sum += n;
124
125                  sums[lane_id] = warp_sum;
126
127          __syncthreads();
128
129
130          int blockSum = 0;
131
132          // fold in unused warp
133          if (warp_id >0)
134          {
135              blockSum = sums[warp_id-1];
136
137          #pragma unroll
138      }
```

Corresponding SASS:

```
Line      Exec Count Disassembly
104          BAR.SYNC 0x0;
105          ISETP.NE.AND P5, PT, R12, RZ, PT;
106          SSY `(.L_8);
107          @P5 NOP.S;
108          SHF.L.W R17, RZ, 0x2, R11;
109          MOV32I R20, 0x20;
110          LDS R18, [R17];
111
112          SHFL.UP PT, R19, R18, 0x2, 0x0;
113          IADD R18, R19, R18;
114          SHFL.LT.U32.AND P0, PT, R11, 0x20, PT;
115          SEL R19, RZ, R19, P1;
116          IADD R18, R19, R18;
117          SHFL.UP PT, R19, R18, 0x4, 0x0;
118          SEL R19, RZ, R19, P2;
119          IADD R18, R19, R18;
120          SHFL.UP PT, R19, R18, 0x8, 0x0;
121          SEL R19, RZ, R19, P3;
122          IADD R18, R19, R18;
123          SHFL.UP PT, R19, R18, 0x10, 0x0;
124          SEL R19, RZ, R19, P4;
125          IADD R18, R19, R18;
126          SHFL.UP PT, R19, R18, R20, 0x0;
127          SEL R11, RZ, R19, P0;
128          IADD R11, R11, R18;
129
130          STS.S [R17]
131
132          .L_8:
133          BAR.SYNC 0x0
134          ISETP.LT.AND PT, PT, R12, RZ, PT;
135          @P0 BRA.U `(.L_9);
136          @!P0 LDS R11, [R16+0x4];
137          @!P0 IADD R26, R26, R11;
138          @!P0 IADD R26, R26, R11;
```

DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

The screenshot shows the NVIDIA Nsight Compute Profiler interface. The left panel displays the source code of a CUDA kernel named `shfl_intimage_rows`. The right panel shows the corresponding assembly code and execution counts. Two specific sections of the assembly code are highlighted with orange boxes and labeled "Execution Count".

Source Code (Left Panel):

```
Line   Exec Count
104   / last thread in the warp holding sum of the warp
105   / places that in shared
106  43200 if(threadIdx.x % warpSize == warpSize-1)
107   {
108     8640 sums[warp_id] = result[15];
109   }
110
111  4320 __syncthreads();
112
113  8640 if(warp_id == 0)
114   {
115     2160 int warp_sum = sums[lane_id];
116     #pragma unroll
117
118     for(int i=1; i<=32; i*=2)
119     {
120       int n = __shfl_up(warp_sum, i, 32);
121
122     14040 if(lane_id >= i) warp_sum += n;
123   }
124
125   1080 sums[lane_id] = warp_sum;
126
127   4320 __syncthreads();
128
129   int blockSum = 0;
130
131   8640
132   {
133     3240 blockSum = sums[warp_id-1];
134     #pragma unroll
135
136   }
```

Assembly and Execution Counts (Right Panel):

A large green box highlights the following assembly code:

```
BAR.SYNC 0x0;
ISETP.NE.AND P5, PT, R12, RZ, PT;
SSY `(.L_8);
@P5 NOP.S;
SHF.L.W R17, RZ, 0x2, R11;
MOV32I R20, 0x20;
LDS R18, [R17];
SHFL.UP PT, R19, R18, 0x1, 0x0;
SEL R19, RZ, R19, P0;
IADD R18, R19, R18;
SHFL.UP PT, R19, R18, 0x2, 0x0;
ISETP.LT.U32.AND P0, PT, R11, 0x20, PT;
SEL R19, RZ, R19, P1;
IADD R18, R19, R18;
SHFL.UP PT, R19, R18, 0x4, 0x0;
SEL R19, RZ, R19, P2;
IADD R18, R19, R18;
SHFL.UP PT, R19, R18, 0x8, 0x0;
SEL R19, RZ, R19, P3;
IADD R18, R19, R18;
SHFL.UP PT, R19, R18, 0x10, 0x0;
SEL R19, RZ, R19, P4;
IADD R18, R19, R18;
SHFL.UP PT, R19, R18, R20, 0x0;
SEL R11, RZ, R19, P0;
IADD R11, R11, R18;
STS.S [R17], R11;
```

A smaller green box highlights the following assembly code:

```
L_8:
BAR.SYNC 0x0;
ISETP.LT.AND P0, PT, R12, 0x1, PT;
@P0 BRA.U `(.L_9);
@!P0 LDS R11, [R16+-0x4];
@!P0 IADD R26, R26, R11;
BRA .L_10 R26, R26, R11;
```

A green box also highlights the word "Execution Count" in the source code area.

DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

File View Help

*New Session shfl_intimage_rows

Line	Exec Count	File	Disassembly
104		/data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shf	BAR.SYNC 0x0;
105			ISETP.NE.AND P5, PT, R12, RZ, PT;
106	43200	// last thread in the warp holding sum of the warp	SSY `(.L_8);
107		// places that in shared	@P5 NOP.S;
108	8640	if(threadIdx.x % warpSize == warpSize-1)	SHF.L.W R17, RZ, 0x2, R11;
109		{	MOV32I R20, 0x20;
110		sums[warp_id] = result[15];	LDS R18, [R17];
111	4320	}	SHFL.UP PT, R19, R18, 0x1, 0x0;
112		__syncthreads();	SEL R19, RZ, R19, P0;
113	8640	if (warp_id == 0)	IADD R18, R19, R18;
114		{	SHFL.UP PT, R19, R18, 0x2, 0x0;
115	2160	int warp_sum = sums[lane_id];	ISETP.LT.U32.AND P0, PT, R11, 0x20, PT;
116		#pragma unroll	SEL R19, RZ, R19, P1;
117			IADD R18, R19, R18;
118		for (int i=1; i<=32; i*=2)	SHFL.UP PT, R19, R18, 0x4, 0x0;
119		{	SEL R19, RZ, R19, P2;
120		int n = __shfl_up(warp_sum, i, 32);	IADD R18, R19, R18;
121		if (lane_id >= i) warp_sum += n;	SHFL.UP PT, R19, R18, 0x8, 0x0;
122	14040	}	SEL R19, RZ, R19, P3;
123		sums[lane_id] = warp_sum;	IADD R18, R19, R18;
124		}	SHFL.UP PT, R19, R18, 0x10, 0x0;
125	1080	__syncthreads();	SEL R19, RZ, R19, P4;
126			IADD R18, R19, R18;
127			SHFL.UP PT, R19, R18, R20, 0x0;
128	4320	int blockSum = 0;	SEL R11, RZ, R19, P0;
129			IADD R11, R11, R18;
130		// fold in unused warp	STS.S [R17], R11;
131		if (warp_id >0)	.L_8:
132		{	BAR.SYNC 0x0;
133	8640	blockSum = sums[warp_id-1];	ISETP.LT.AND P0, PT, R12, 0x1, PT;
134		#pragma unroll	@P0 BRA.U `(.L_9);
135			@!P0 LDS R11, [R16+-0x4];
136	3240		@!P0 IADD R26, R26, R11;
137			...
138			...

Hot-Spots

DETAILED KERNEL PROFILE

VIEW SOURCE AND ASSEMBLY SIDE-BY-SIDE

File View Help

*New Session shfl_intimage_rows

Line	Exec Count	File - /data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/shfl_scan/shf	Exec Count	Disassembly
104		// last thread in the warp holding sum of the warp	4320	BAR SYNC 0x0;
105		// places that in shared	4320	ISETP.NE.AND P5, PT, R12, RZ, PT;
106	43200	if(threadIdx.x % warpSize == warpSize-1)	4320	SSY `(.L_8);
107		{	4320	@P5 NOP.S;
108	8640	sums[warp_id] = result[15];	1080	SHF.L.W R17, RZ, 0x2, R11;
109		}	1080	MOV32I R20, 0x20;
110			1080	LDS R18, [R17];
111	4320	__syncthreads();	1080	SHFL.UP PT, R19, R18, 0x1, 0x0;
112			1080	SEL R19, RZ, R19, P0;
113	8640	if(warp_id == 0)	1080	IADD R18, R19, R18;
114		{	1080	SHFL.UP PT, R19, R18, 0x2, 0x0;
115	2160	int warp_sum = sums[lane_id];	1080	ISETP.LT.U32.AND P0, PT, R11, 0x20, PT;
116		#pragma unroll	1080	SEL R19, RZ, R19, P1;
117			1080	IADD R18, R19, R18;
118		for (int i=1; i<=32; i*=2)	1080	SHFL.UP PT, R19, R18, 0x4, 0x0;
119		{	1080	SEL R19, RZ, R19, P2;
120		int n = __shfl_up(warp_sum, i, 32);	1080	IADD R18, R19, R18;
121			1080	SHFL.UP PT, R19, R18, 0x8, 0x0;
122	14040	if(lane_id >= i) warp_sum += n;	1080	SEL R19, RZ, R19, P3;
123		}	1080	IADD R18, R19, R18;
124			1080	SHFL.UP PT, R19, R18, 0x10, 0x0;
125	1080	sums[lane_id] = warp_sum;	1080	SEL R19, RZ, R19, P4;
126		}	1080	IADD R18, R19, R18;
127			1080	SHFL.UP PT, R19, R18, R20, 0x0;
128	4320	__syncthreads();	1080	SEL R11, RZ, R19, P0;
129			1080	IADD R11, R11, R18;
130		int blockSum = 0;	1080	STS.S [R17], R11;
131			4320	.L_8:
132			4320	BAR SYNC 0x0;
133		// fold in unused warp	4320	ISETP.LT.AND P0, PT, R12, 0x1, PT;
134	8640	if(warp_id >0)	4320	@P0 BRA.U `(.L_9);
135		{		
136	3240	blockSum = sum;		
137		#pragma unroll		
138				

Inefficient Execution

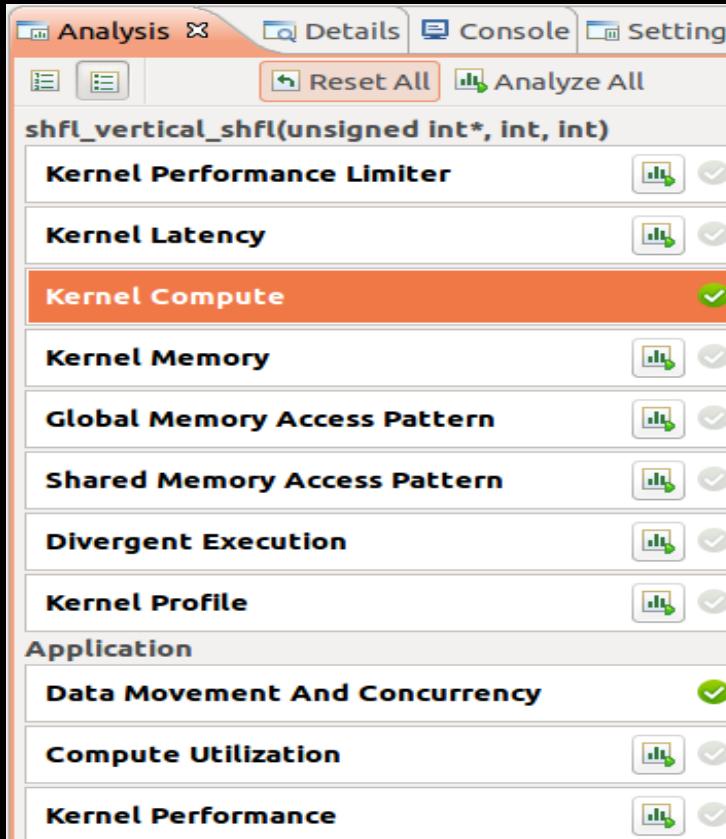
Inactive threads (red) = 6% [8640 inactive threads out of 138240 total threads]
 Predicated off threads (blue) = 68% [95040 predicated off threads out of 138240 total threads]

WHAT'S NEW IN 6.0?

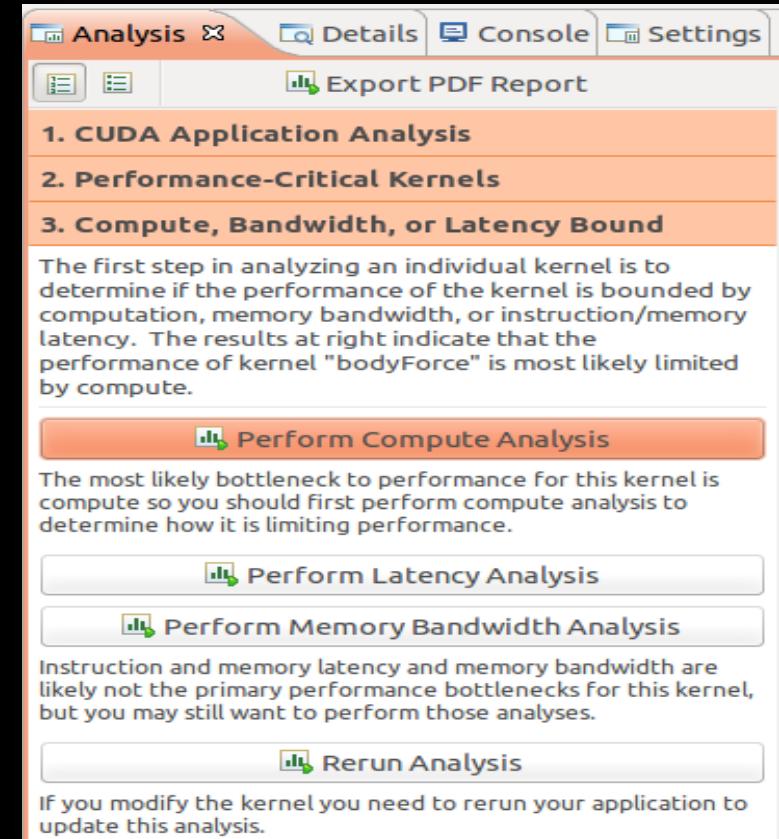
- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling

WHERE CAN I FIND THE NEW OPTION?

Unguided Analysis



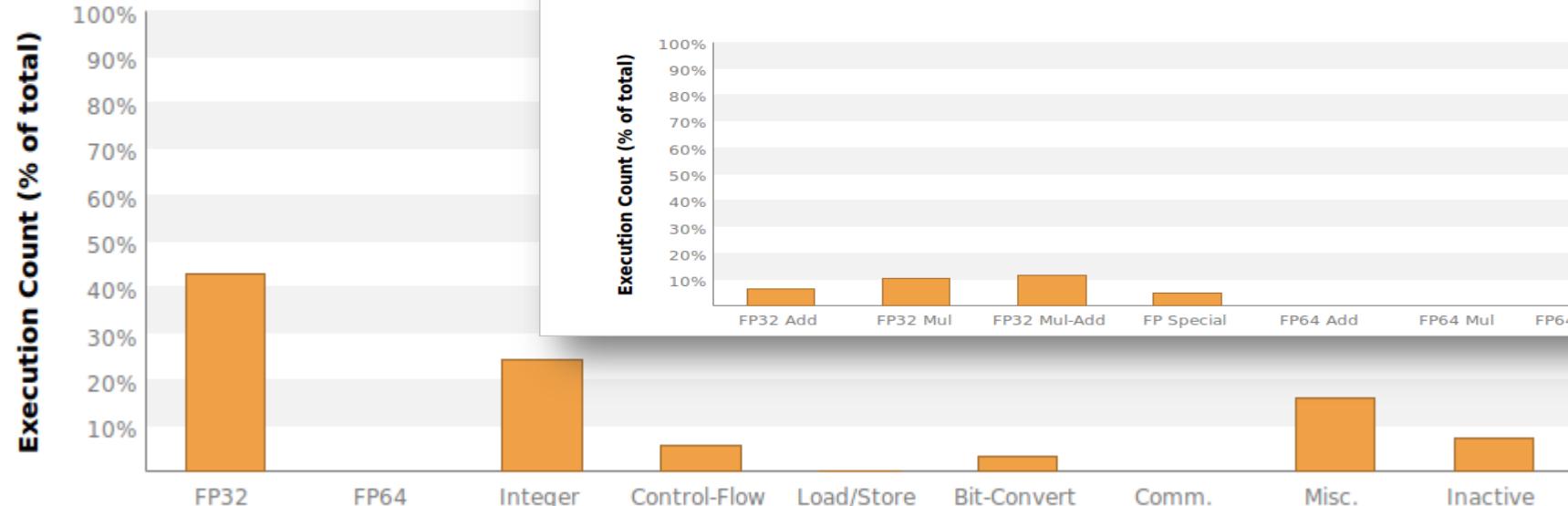
Guided Analysis



MIX OF INSTRUCTIONS FOR A KERNEL

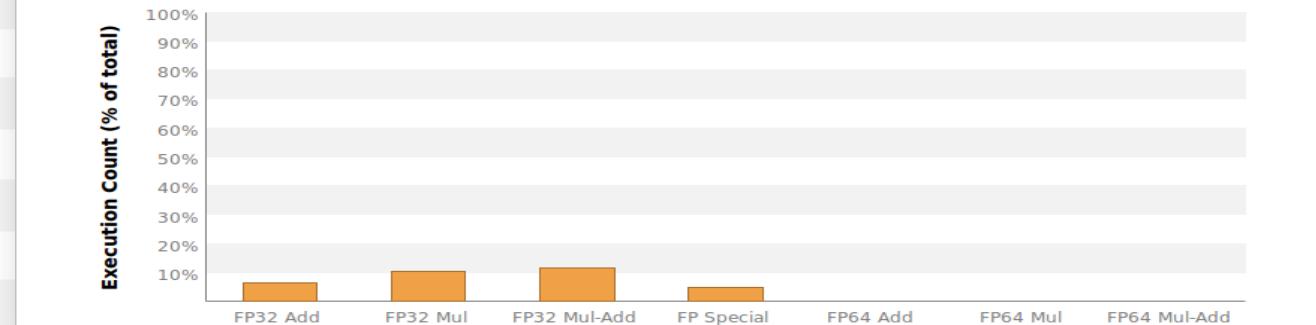
i Instruction Execution Counts

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.



i Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.

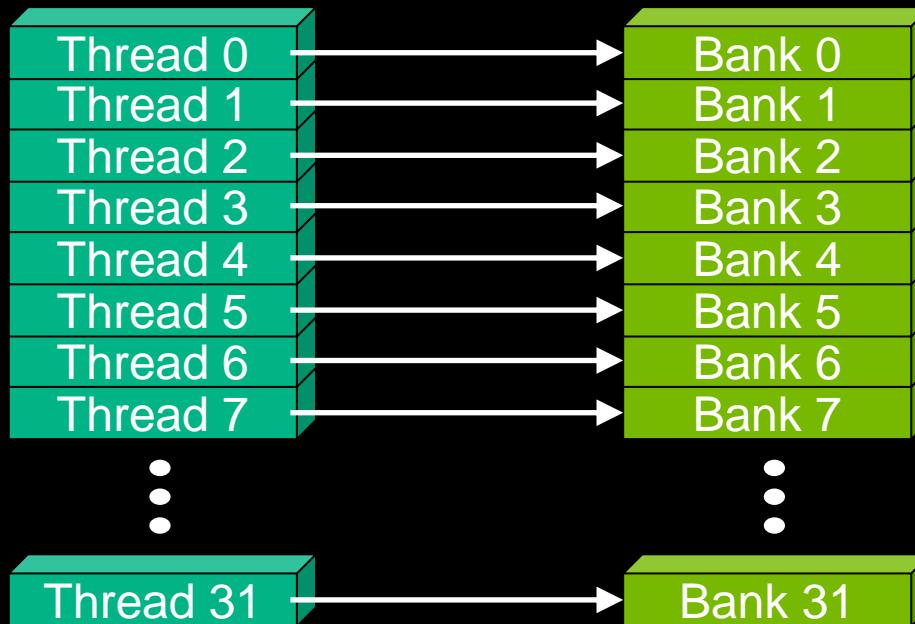


WHAT'S NEW IN 6.0?

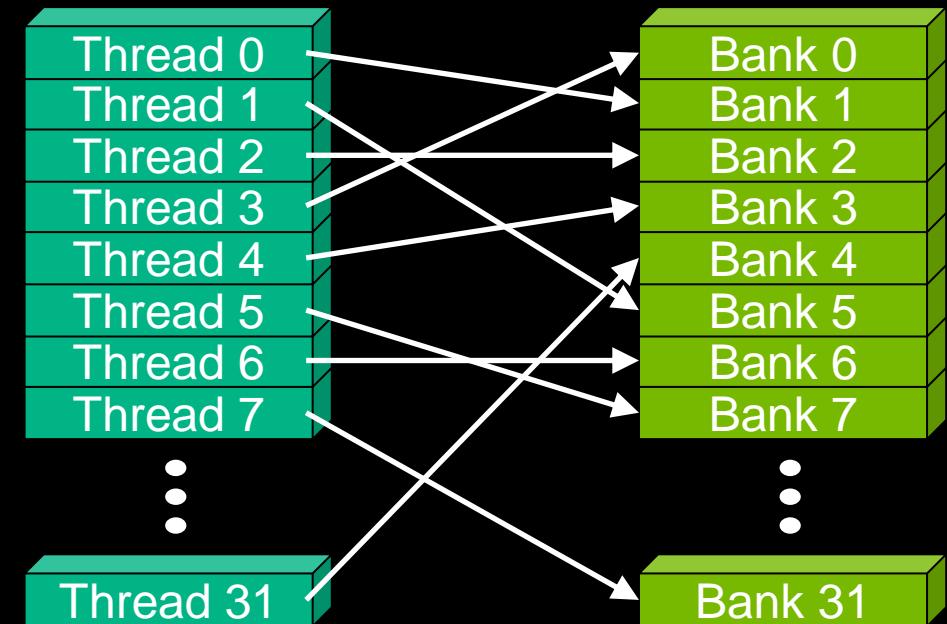
- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling

SHARED MEMORY BANK CONFLICT

- No Bank Conflicts

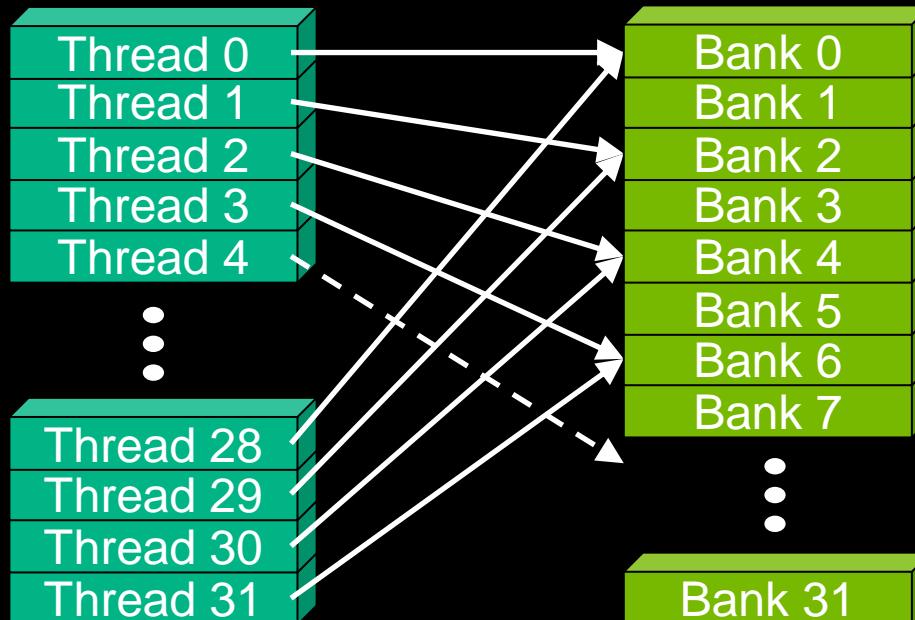


- No Bank Conflicts

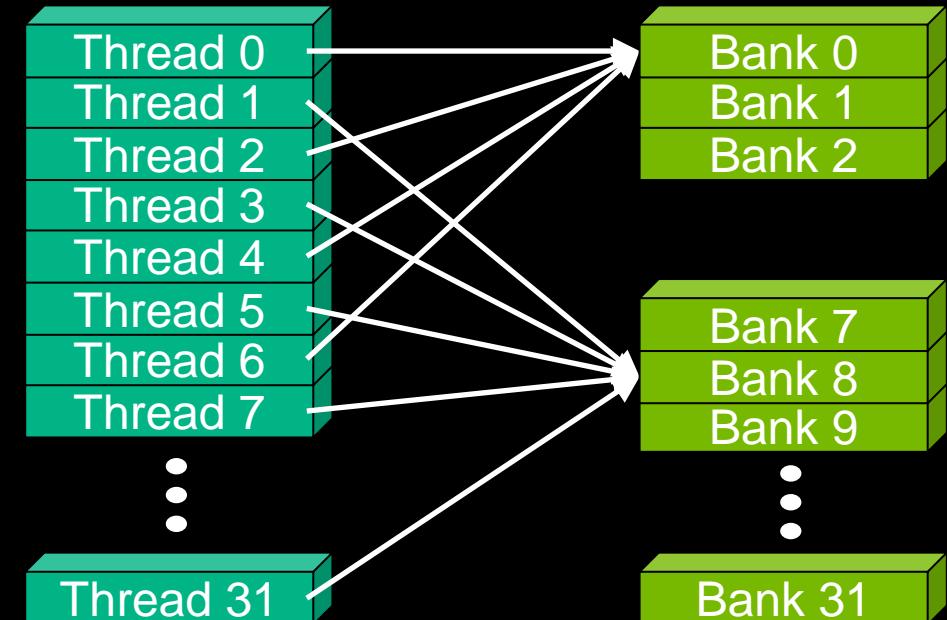


SHARED MEMORY BANK CONFLICT

- 2-way Bank Conflicts

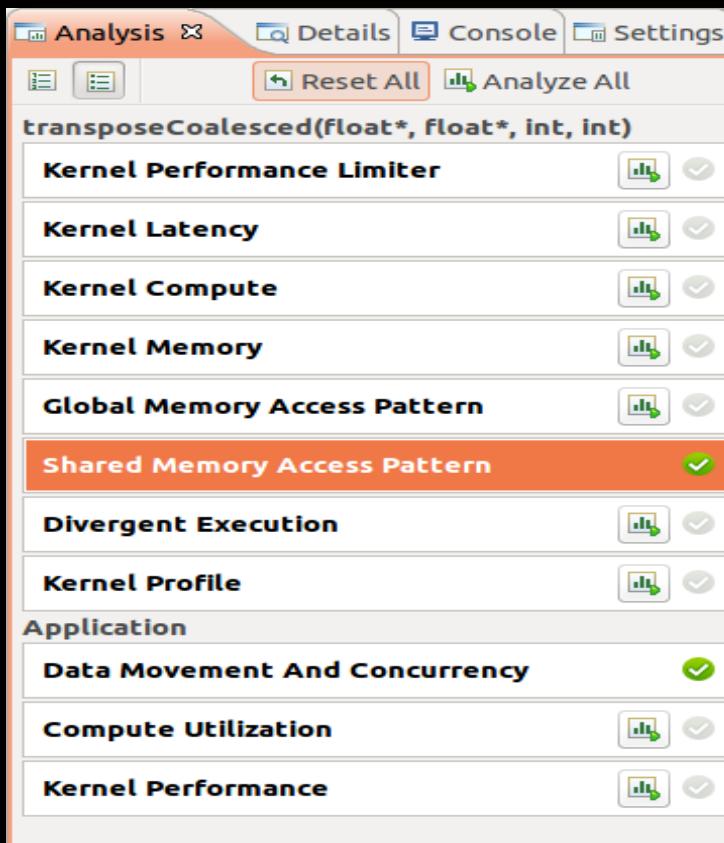


- 8-way Bank Conflicts

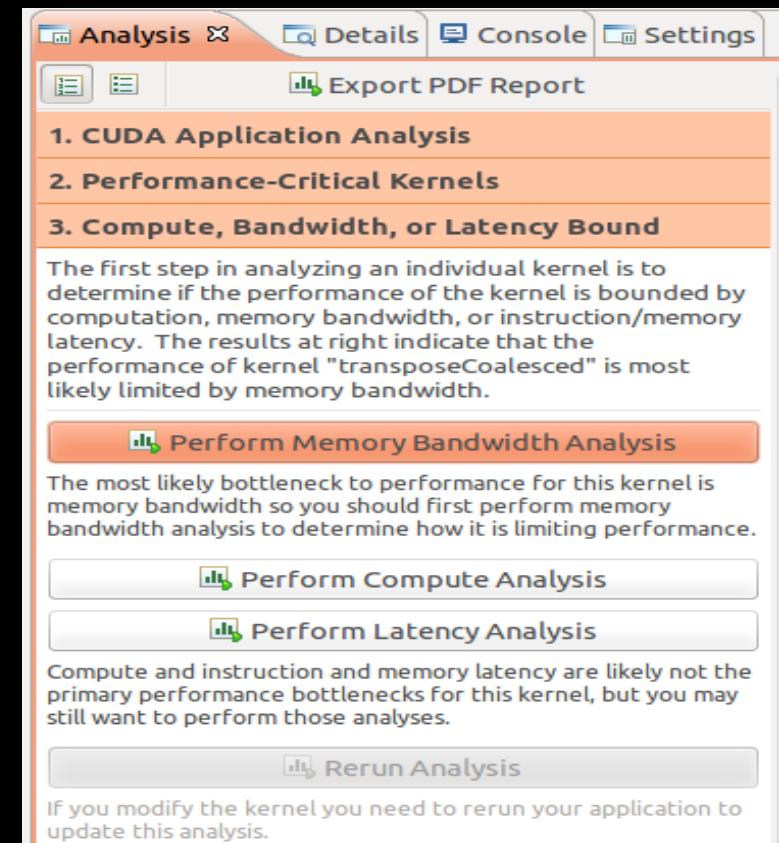


WHERE CAN I FIND THE NEW OPTION?

Unguided Analysis



Guided Analysis



SHARED MEMORY SOURCE LEVEL ANALYSIS

Results

⚠ Shared Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

[More...](#)

Line / File

transpose.cu - /data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/6_Advanced/transpose

146

Shared Load Transactions/Access = 4, Ideal Transactions/Access = 1 [131072 transactions for 32768 total executions]

Bank conflict

Source Line with
Inefficient Access

```
_global_ void transposeCoalesced(float *odata, float *idata, int width, int height)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
    }
}
```

WHAT'S NEW IN 6.0?

- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling

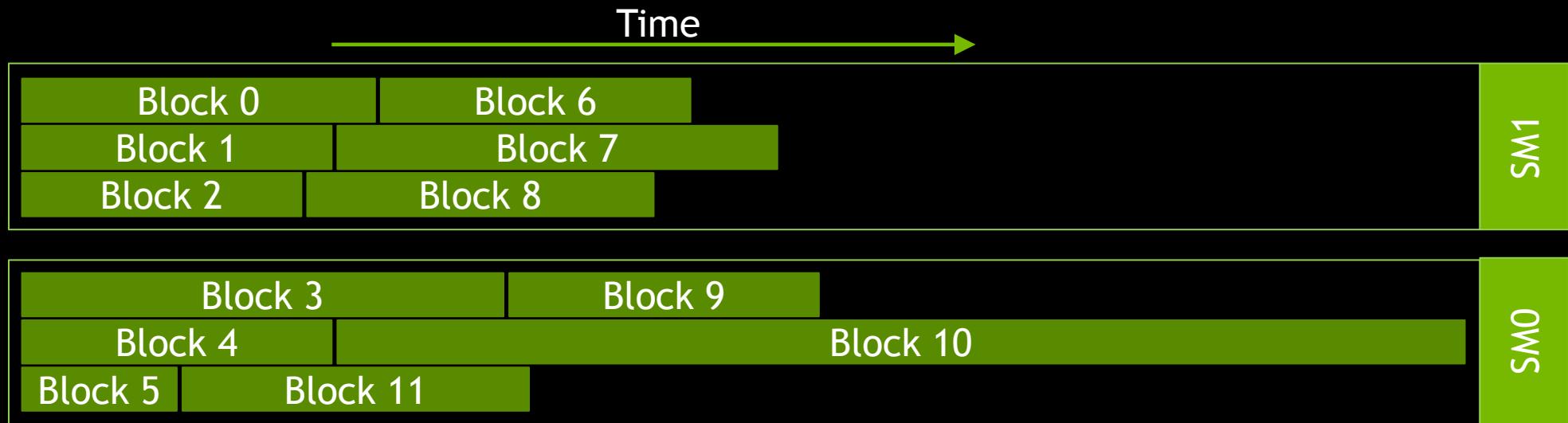
INEFFICIENT SM UTILIZATION

- In theory... kernel allows enough warps on each SM



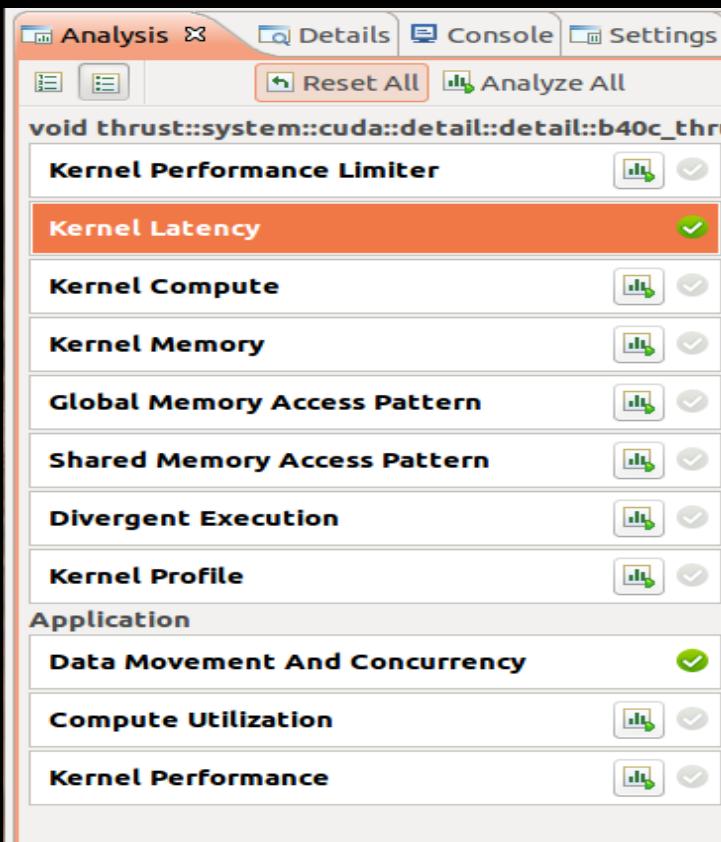
INEFFICIENT SM UTILIZATION

- In theory... kernel launches sufficient warps on each SM
- but why is achieved occupancy for kernel low?
- Likely cause is that all SMs do not remain equally busy over duration of kernel execution

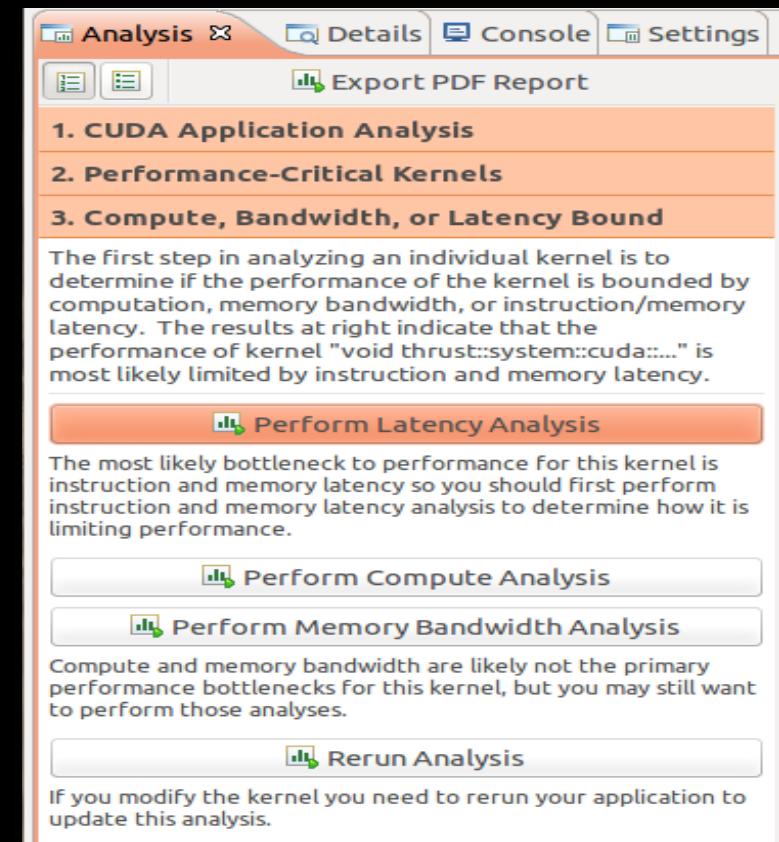


WHERE CAN I FIND THE NEW OPTION?

Unguided Analysis



Guided Analysis



PER-SM ACTIVITY

Results

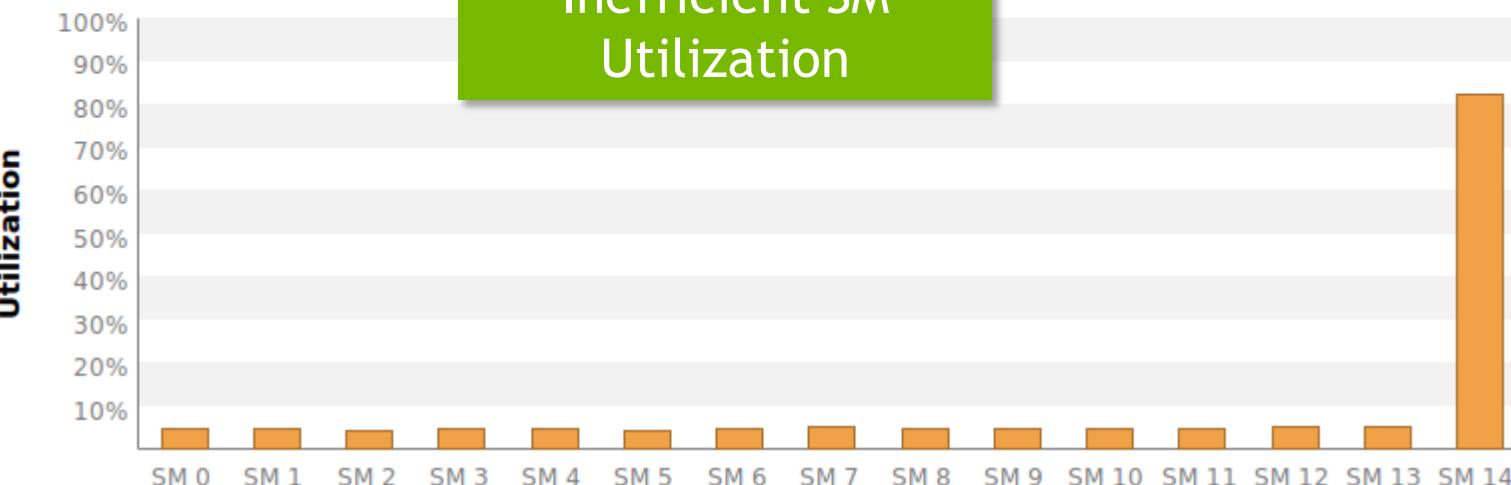
⚠ Achieved Occupancy Is Low

Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy. The kernel's achieved occupancy of 14.1% is significantly lower than its theoretical occupancy of 56.2%. Most likely this indicates that there is an imbalance in how the kernel's blocks are executing on the SMs so that all SMs are not equally busy over the entire execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.

Optimization: Make sure that all blocks are doing roughly the same amount of work. It may also help to increase the number of blocks executed by the kernel.

[More...](#)

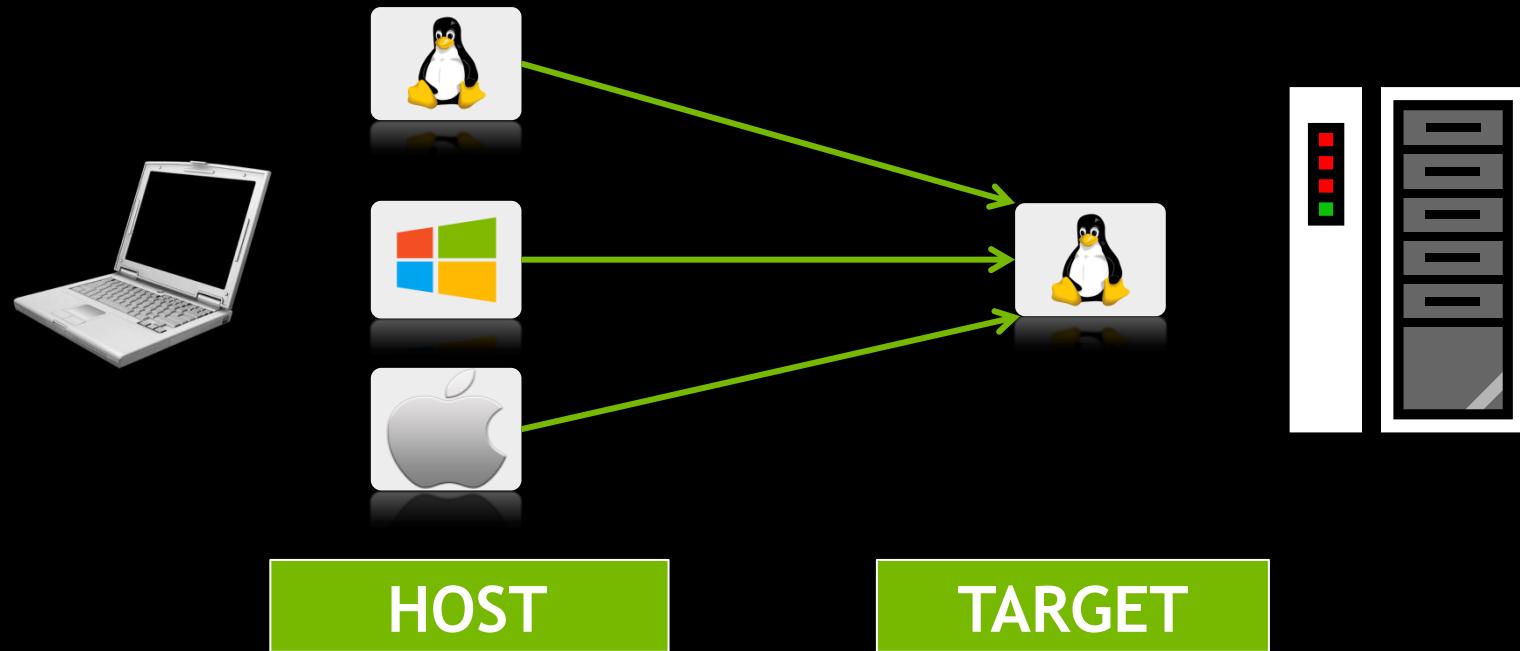
Inefficient SM
Utilization



WHAT'S NEW IN 6.0?

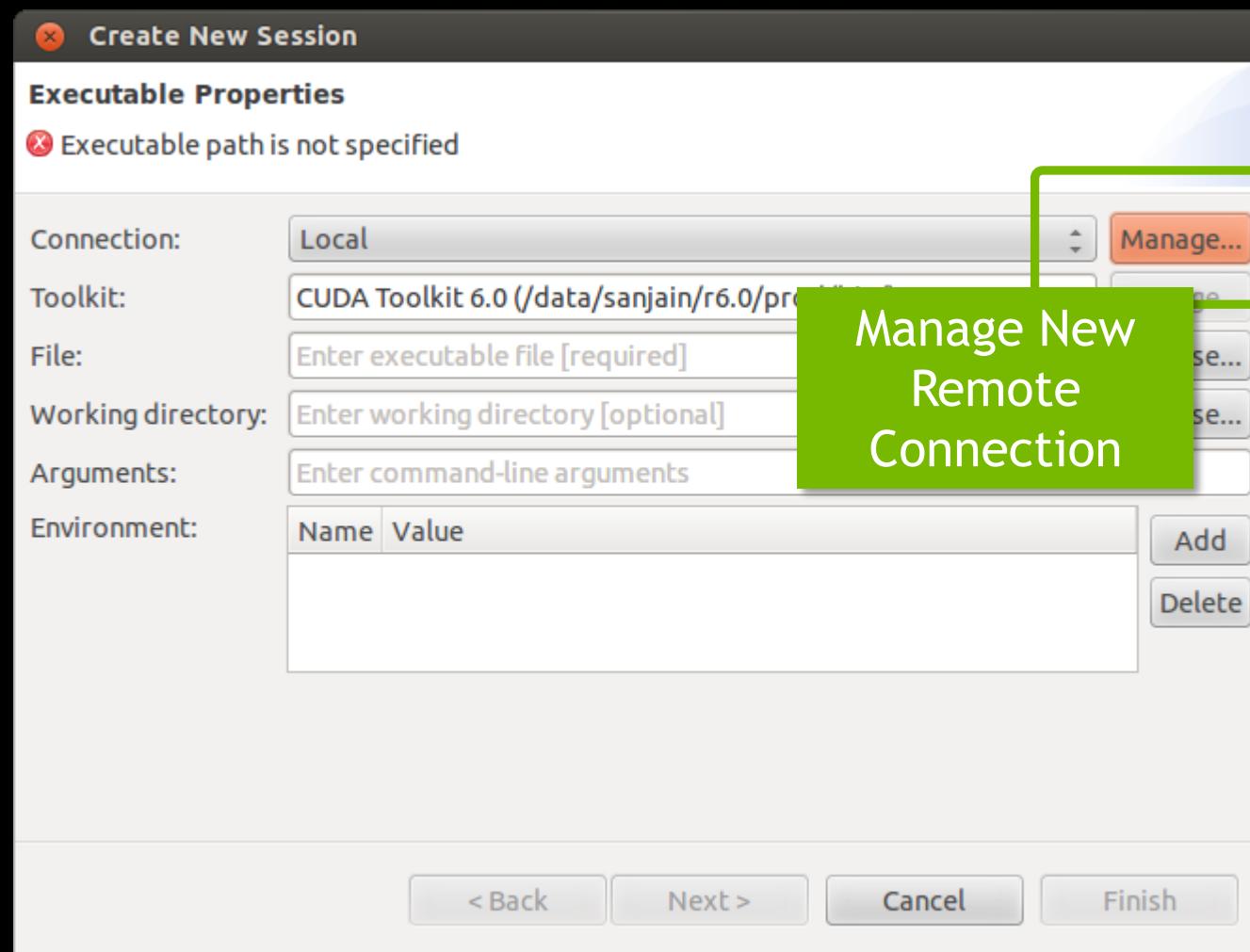
- Unified Memory Profiling
- Multi-Process Service (MPS) Profiling
- Detailed Kernel Profile
- Mix of Instructions for a Kernel
- Shared Memory Source Level Analysis
- Inefficient SM Utilization Detection
- Remote Profiling

REMOTE PROFILING SCENARIOS

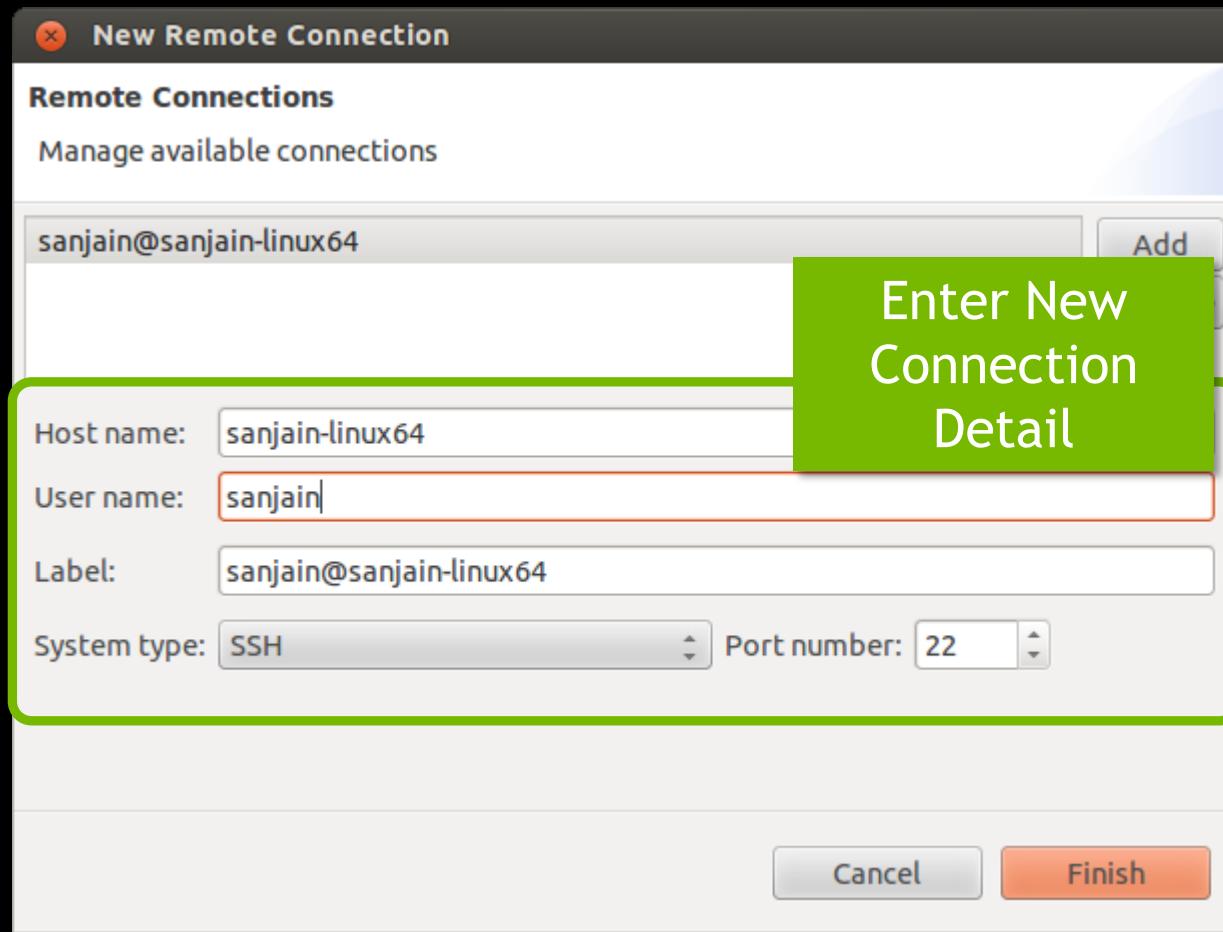


- Host- All supported x86 distributions
- Target- Can be any supported x86/ARM Linux
- Not necessary for **Host System** to have an NVIDIA GPU

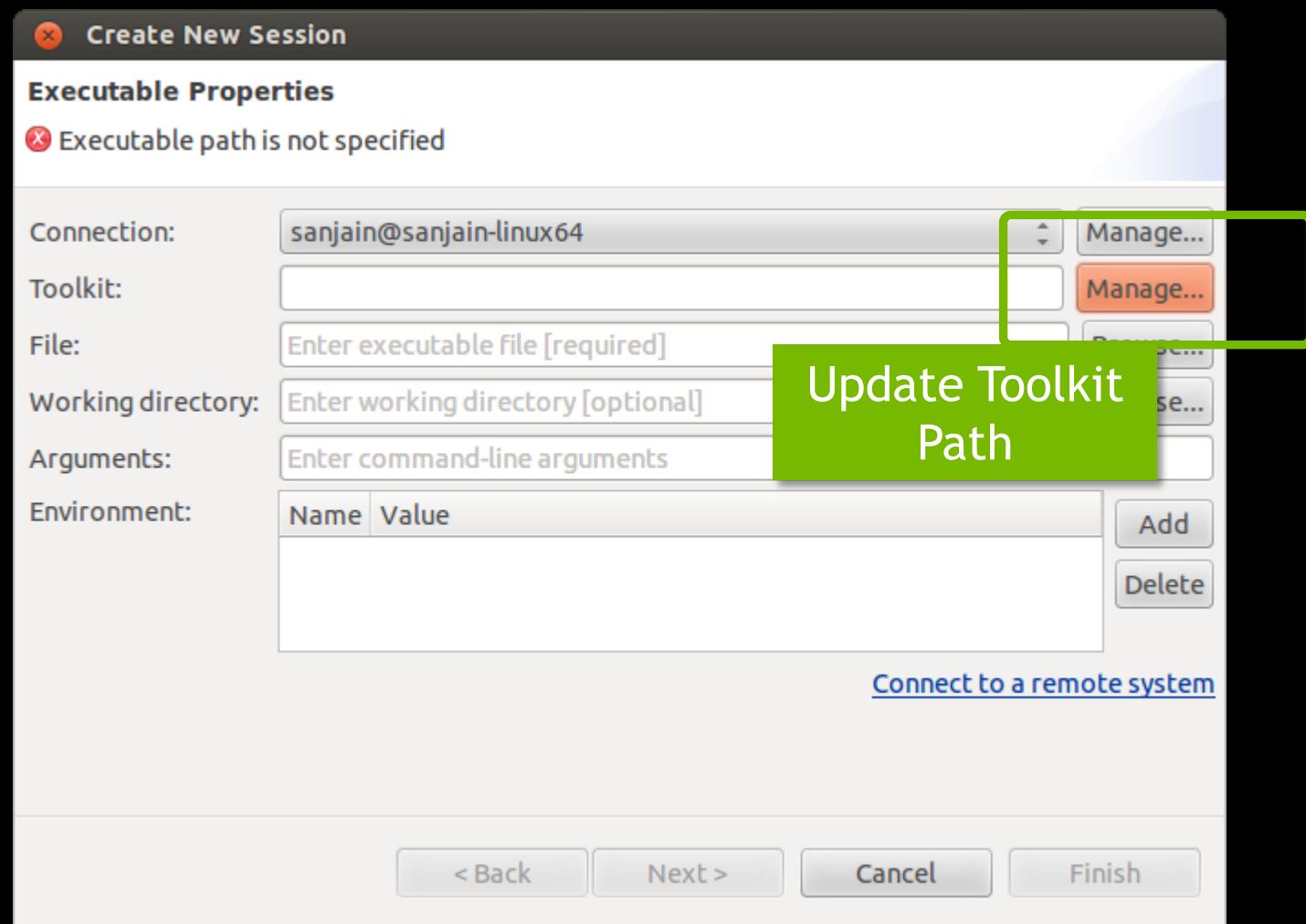
NVVP REMOTE PROFILING SETUP



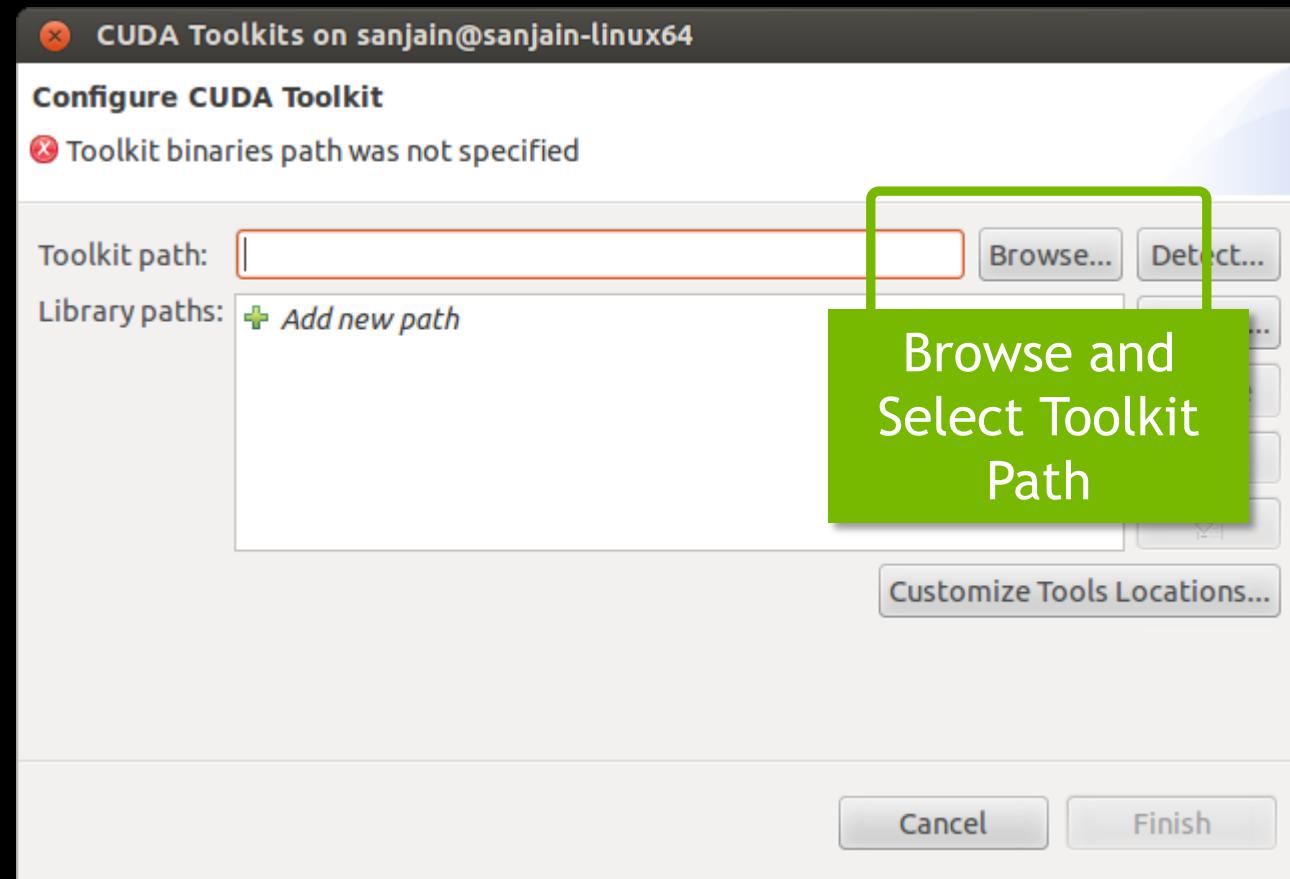
NVVP REMOTE PROFILING SETUP



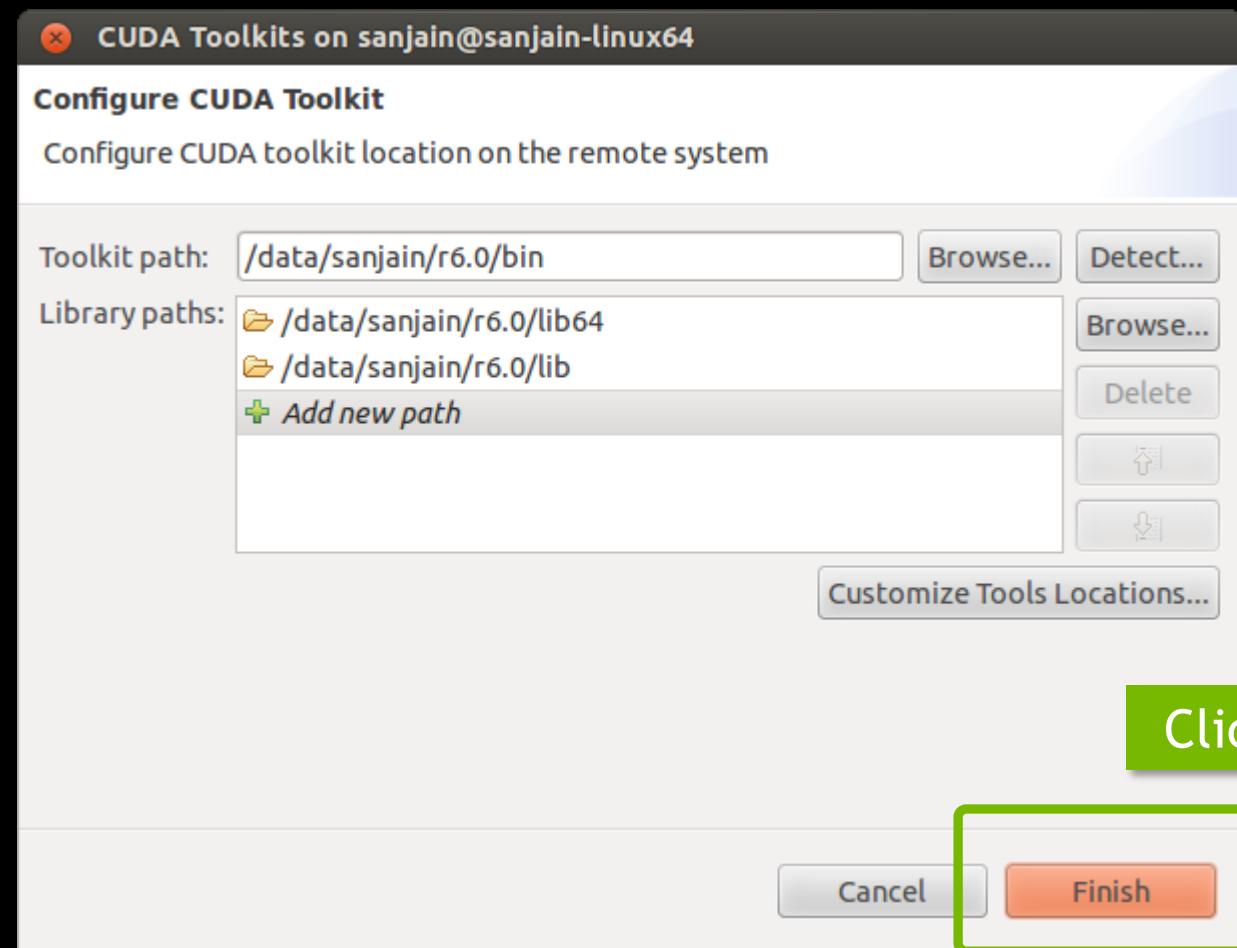
NVVP REMOTE PROFILING SETUP



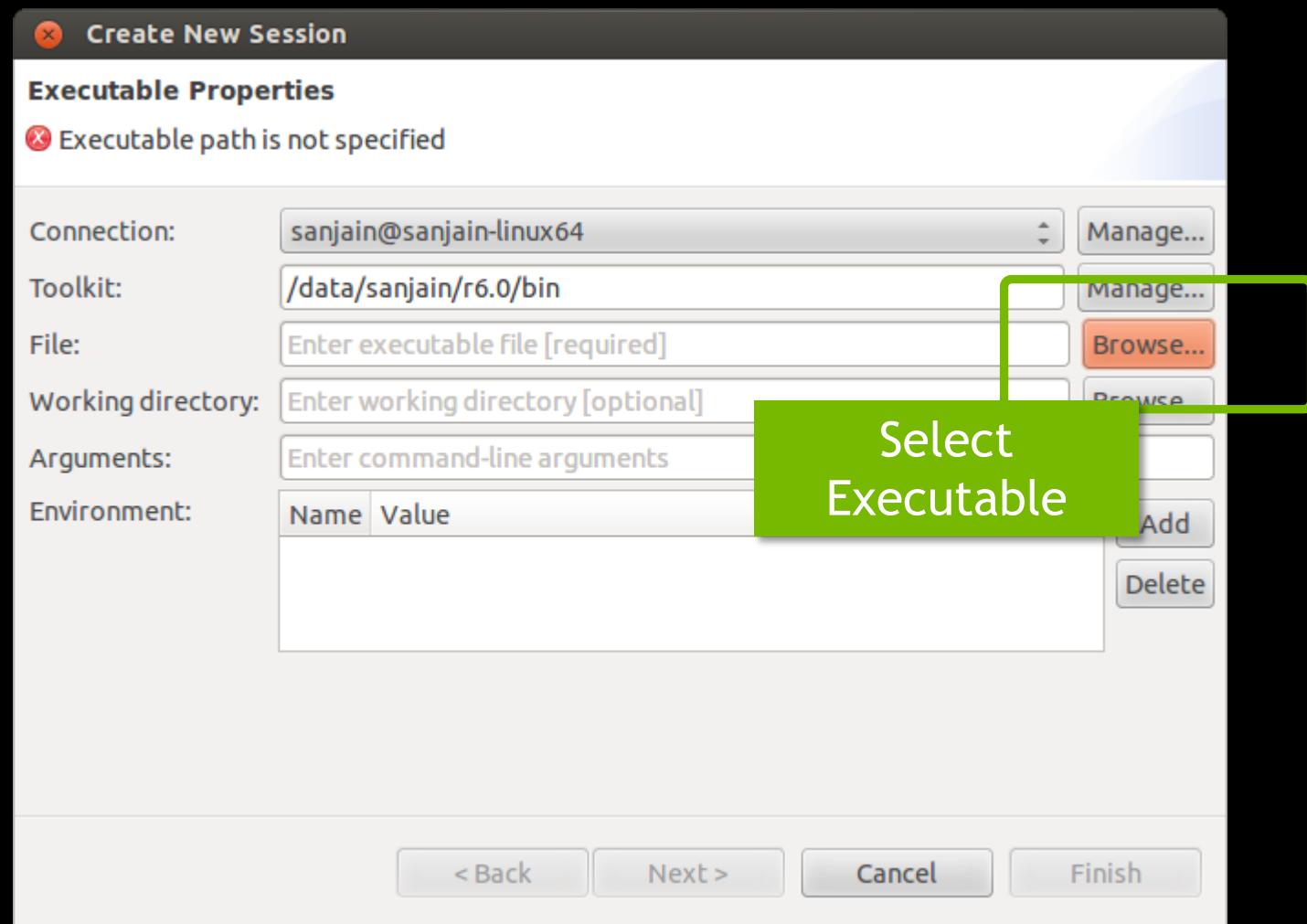
NVVP REMOTE PROFILING SETUP



NVVP REMOTE PROFILING SETUP



NVVP REMOTE PROFILING SETUP



NVVP REMOTE PROFILING SETUP

Create New Session

Executable Properties

Set executable properties

Connection: sanjain@sanjain-linux64

Toolkit: /data/sanjain/r6.0/bin

File: /data/sanjain/r6.0/NVIDIA_CUDA-6.0_Samples/bin/x86_64/linux

Working directory: Enter working directory [optional]

Arguments: Enter command-line arguments

Environment:

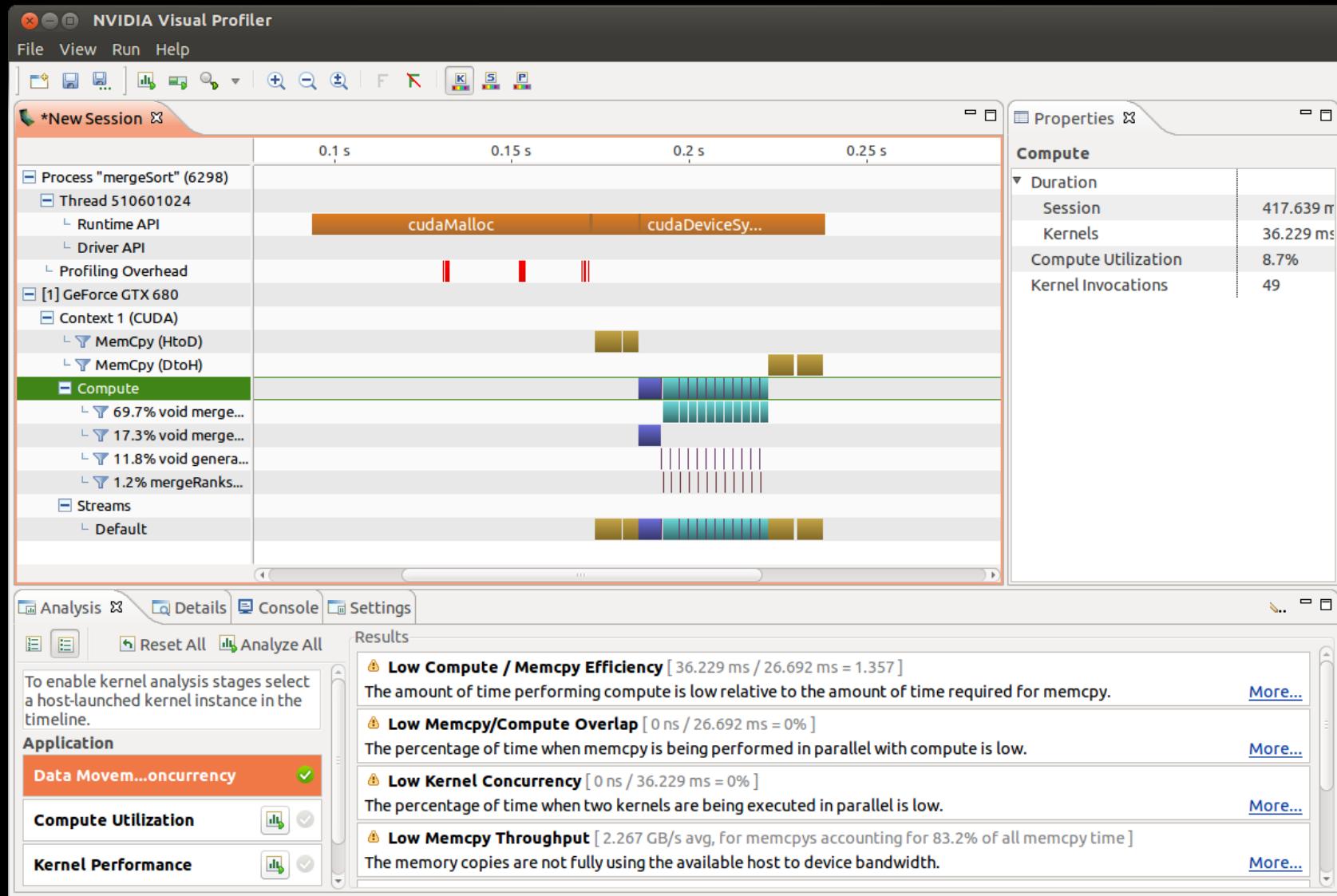
Name	Value

[Connect to a remote system](#)

< Back

Finish and Continue

NVVP REMOTE PROFILING SETUP



NEXT STEPS

- Download free CUDA Toolkit: www.nvidia.com/getcuda
 - Join the community: developer.nvidia.com/join
 - Post at Developer Zone Forums: devtalk.nvidia.com
 - Visit Hangout Pods and Developer Demo Stations
-
- S4794 - Optimizing CUDA Application Performance with NVIDIA® Visual Profiler
 - Wednesday, 03/26, 17:00
 - S4591 - CUDA Application Development Life Cycle Using NVIDIA® Nsight™, Eclipse Edition
 - Thursday, 03/27, 15:00