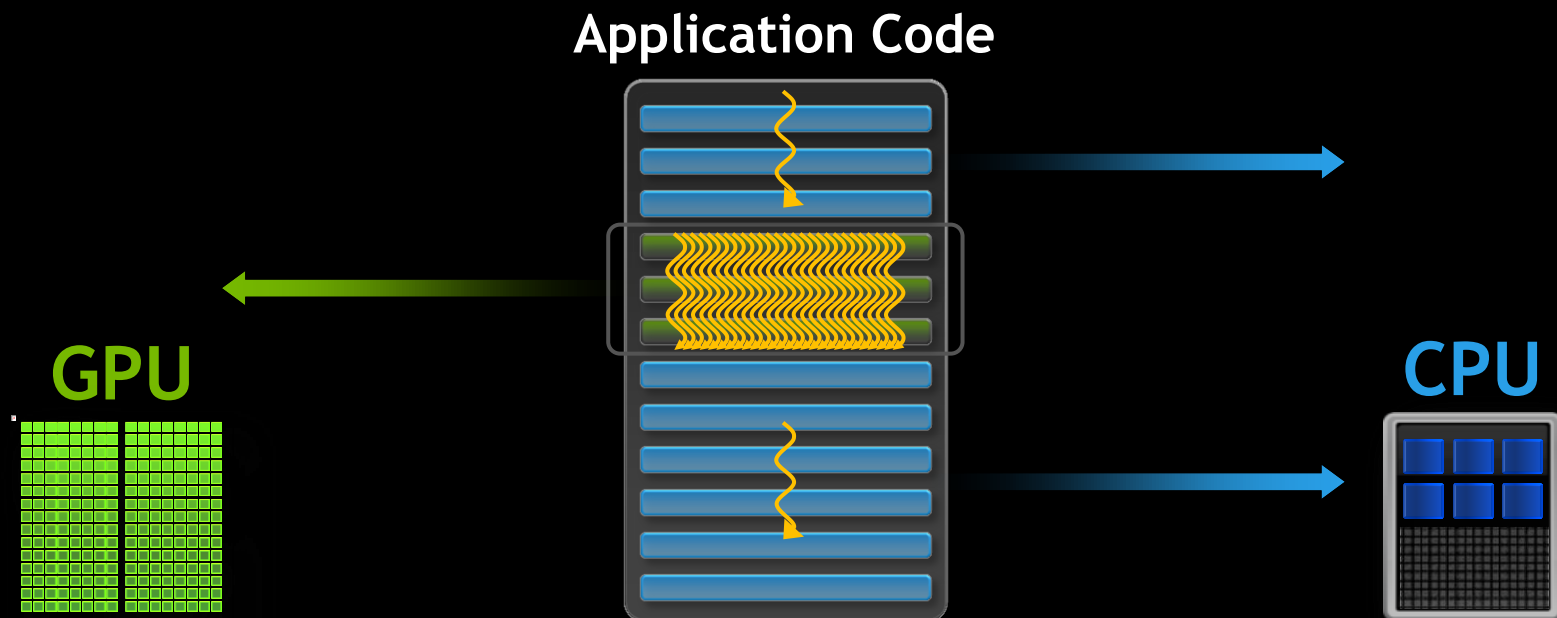


# Optimizing Application Performance with CUDA Profiling Tools



# Why Profile?



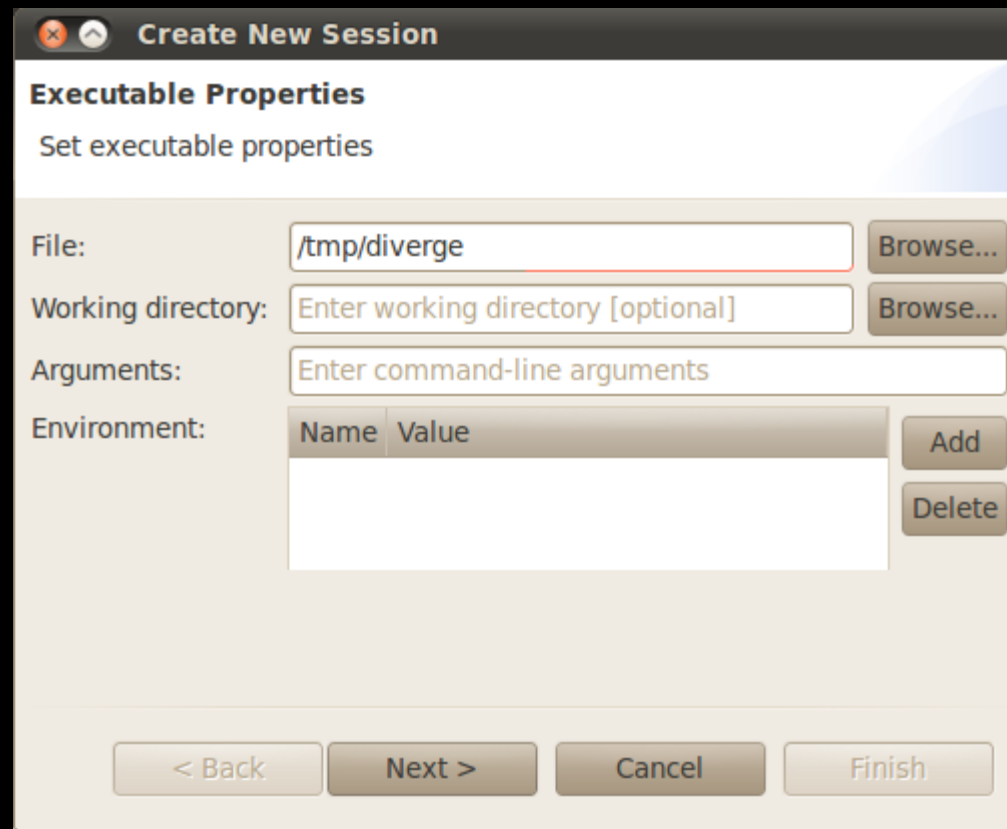
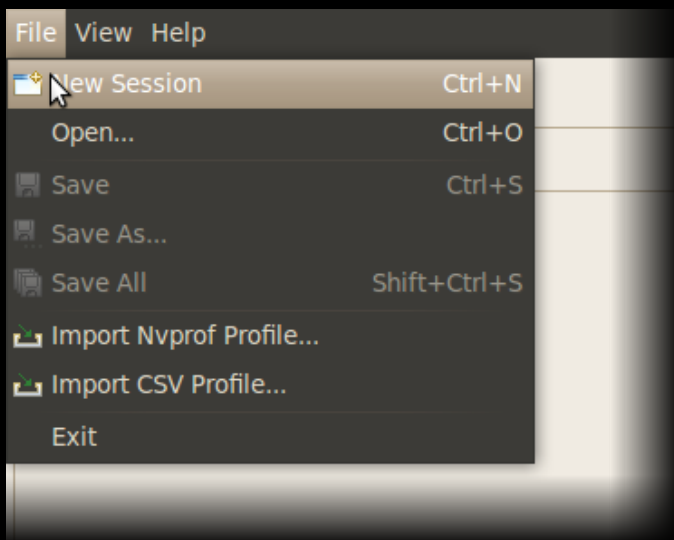
- 100's of cores
- 10,000's of threads
- Great memory bandwidth
- Best at parallel execution

- A few cores
- 10's of threads
- Good memory bandwidth
- Best at serial execution

# Graphical and Command-Line

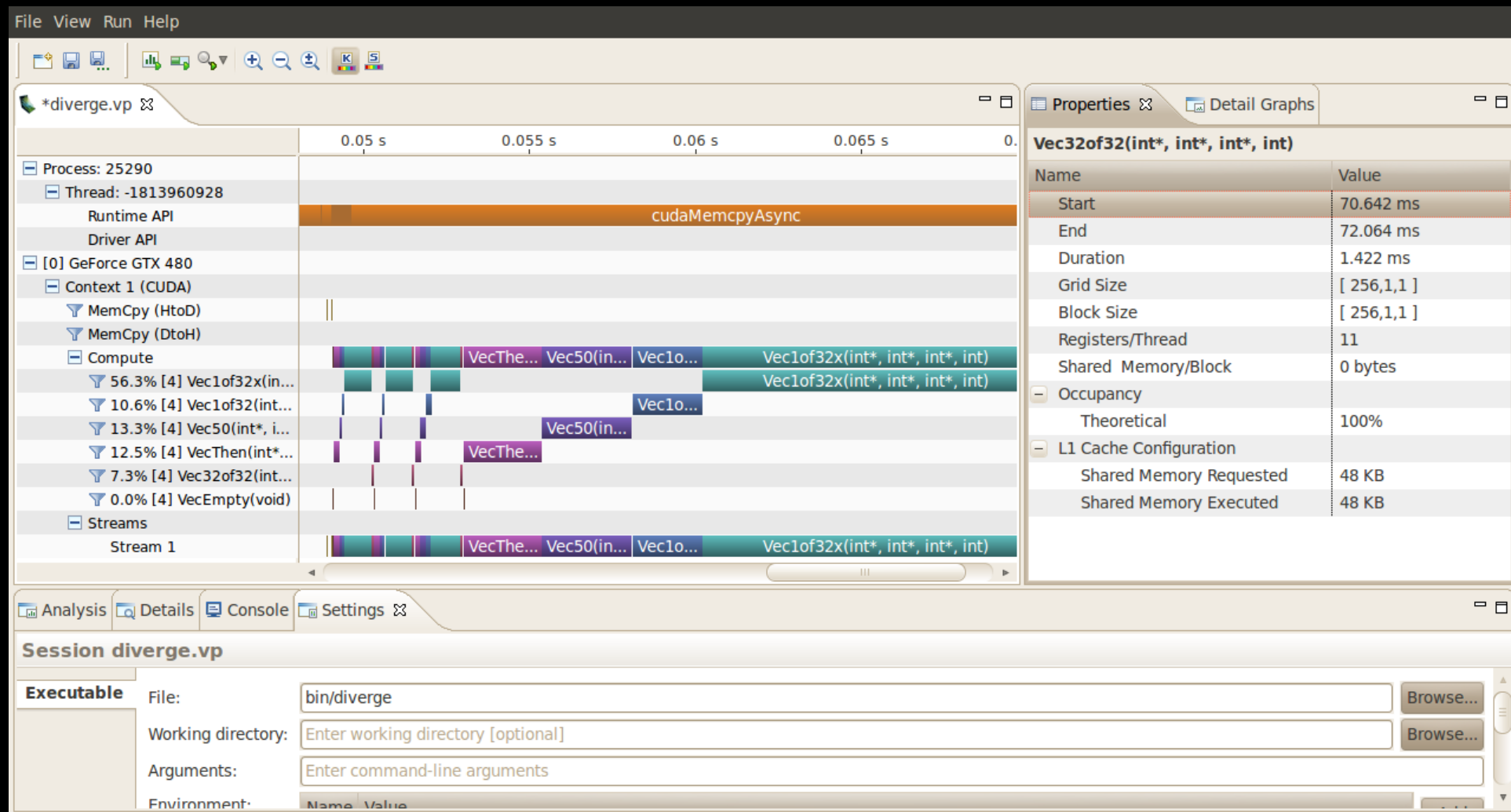
- NVIDIA® Visual Profiler
  - Standalone (nvvp)
  - Integrated into NVIDIA® Nsight™ Eclipse Edition (nsight)
- nvprof
  - Command-line profiler
- Current command-line profiler still available

# Profiling Session

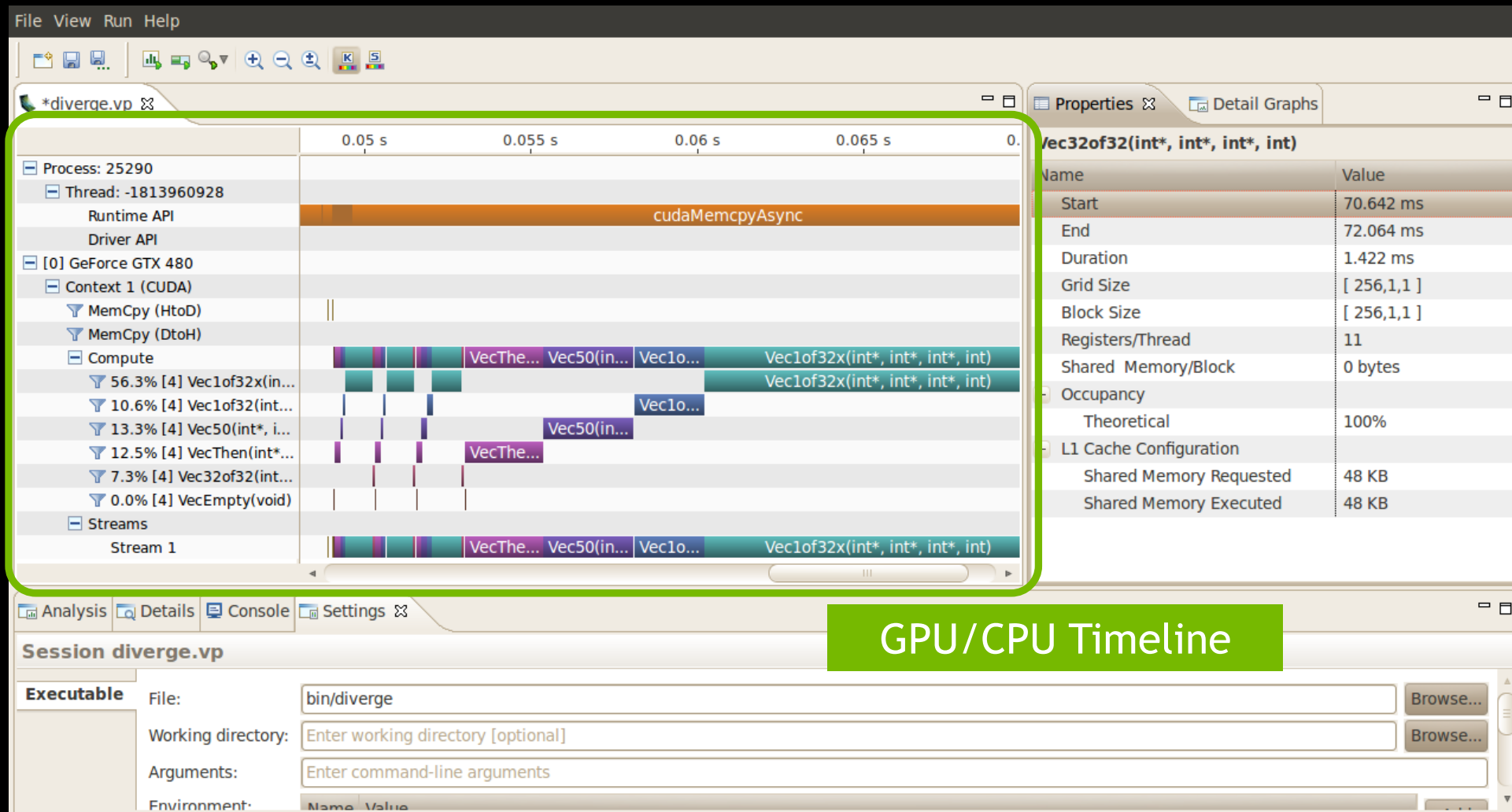




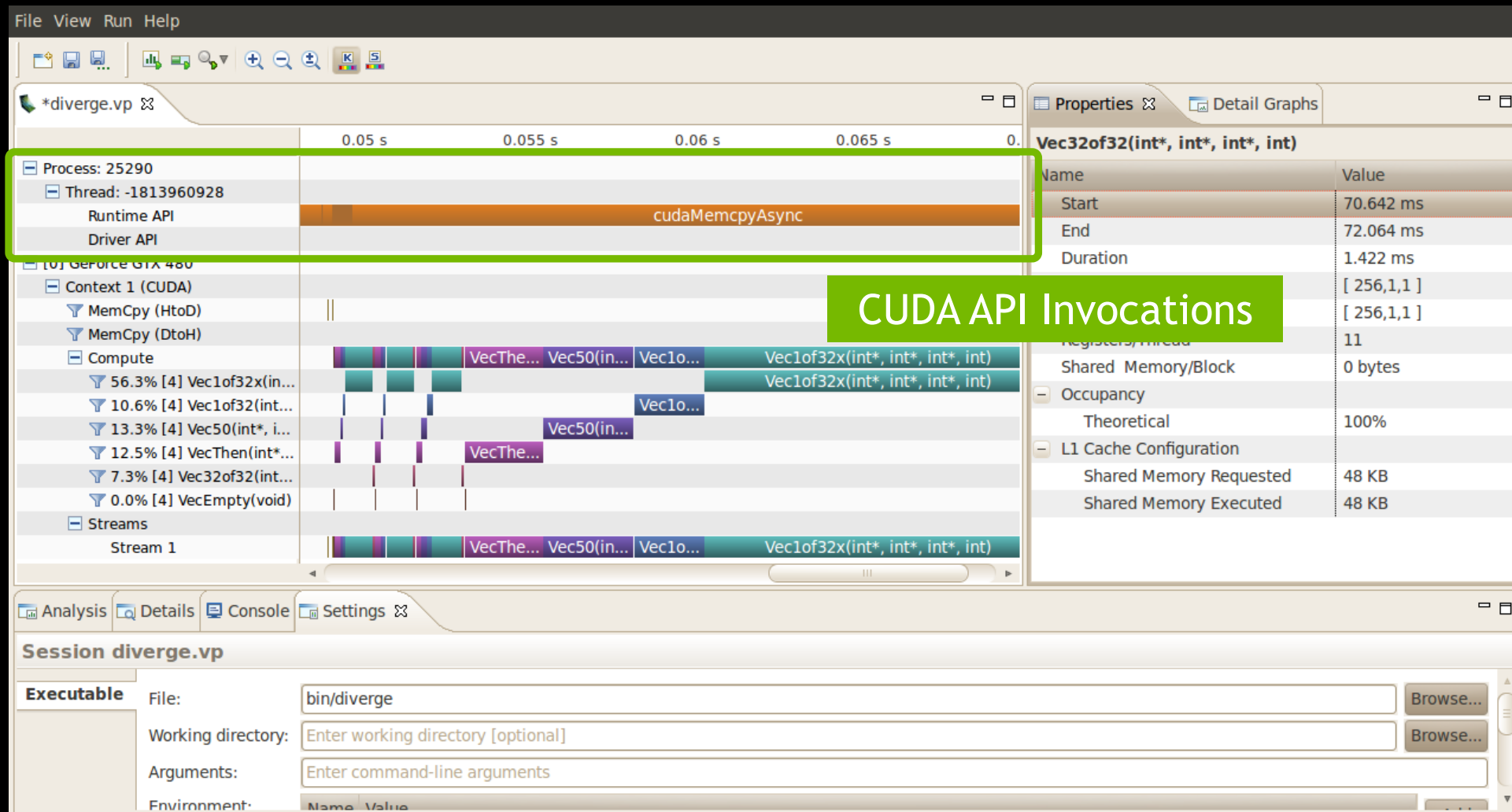
# NVIDIA Visual Profiler



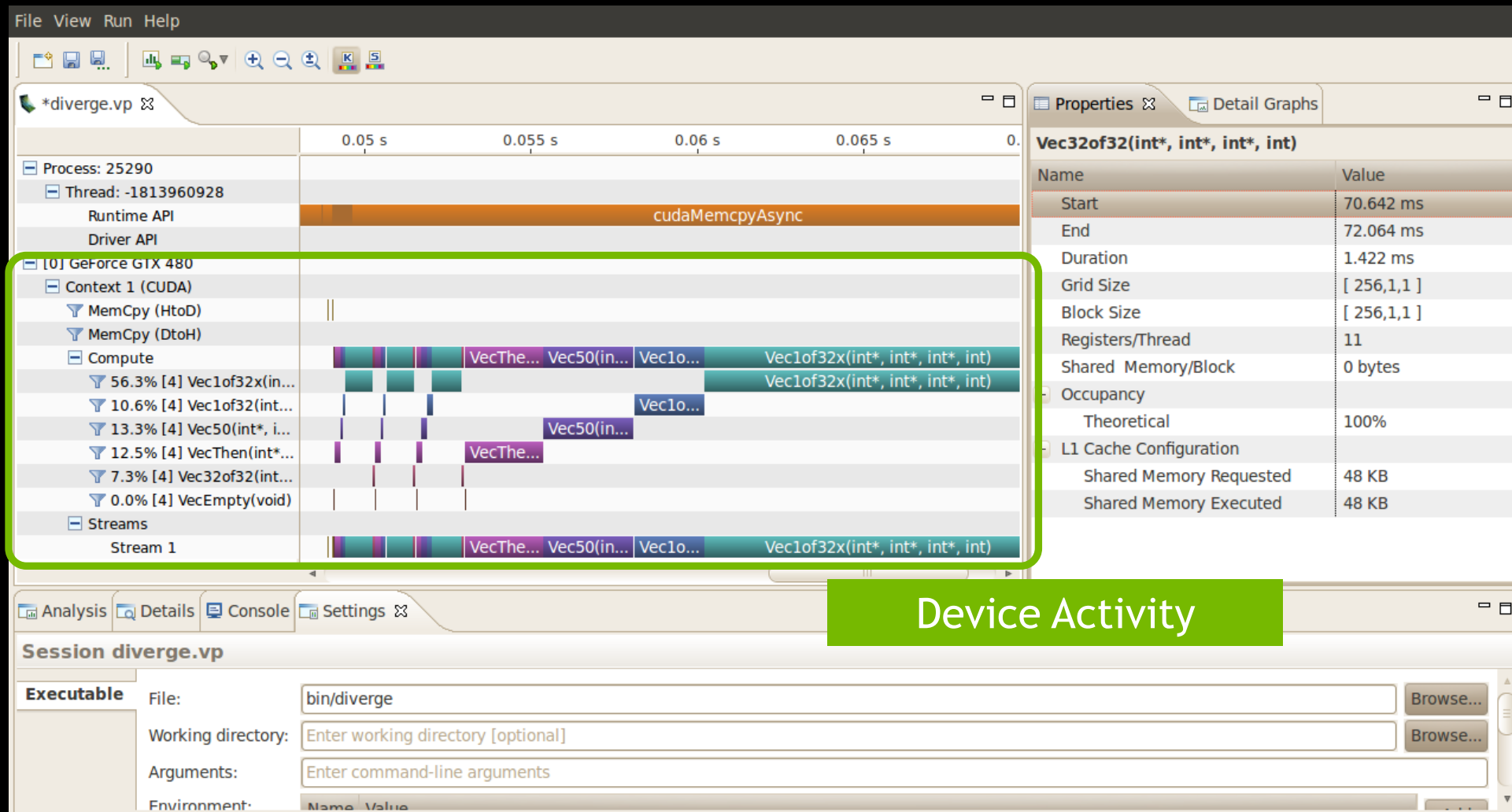
# Timeline



# CPU Timeline

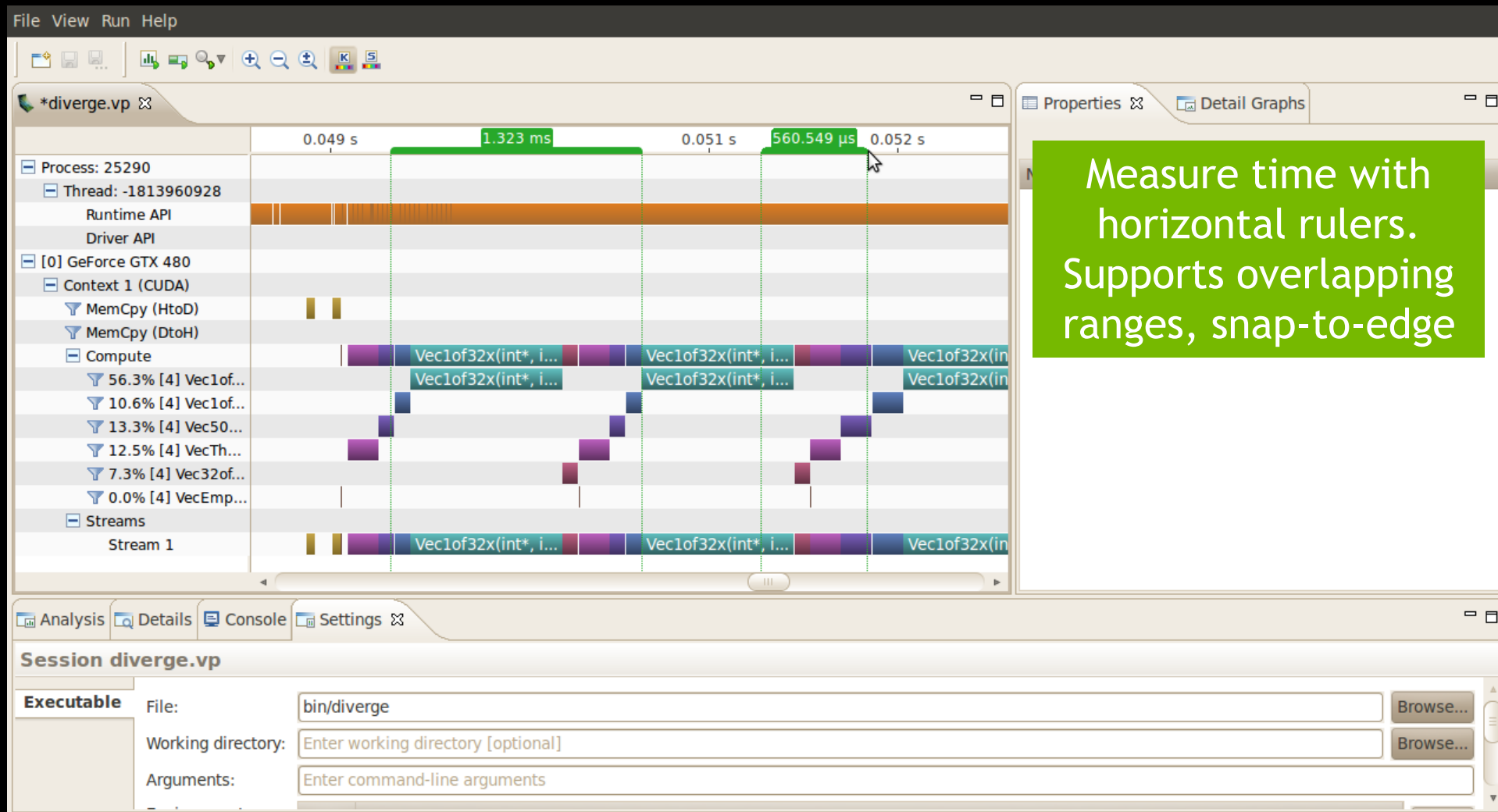


# GPU Timeline

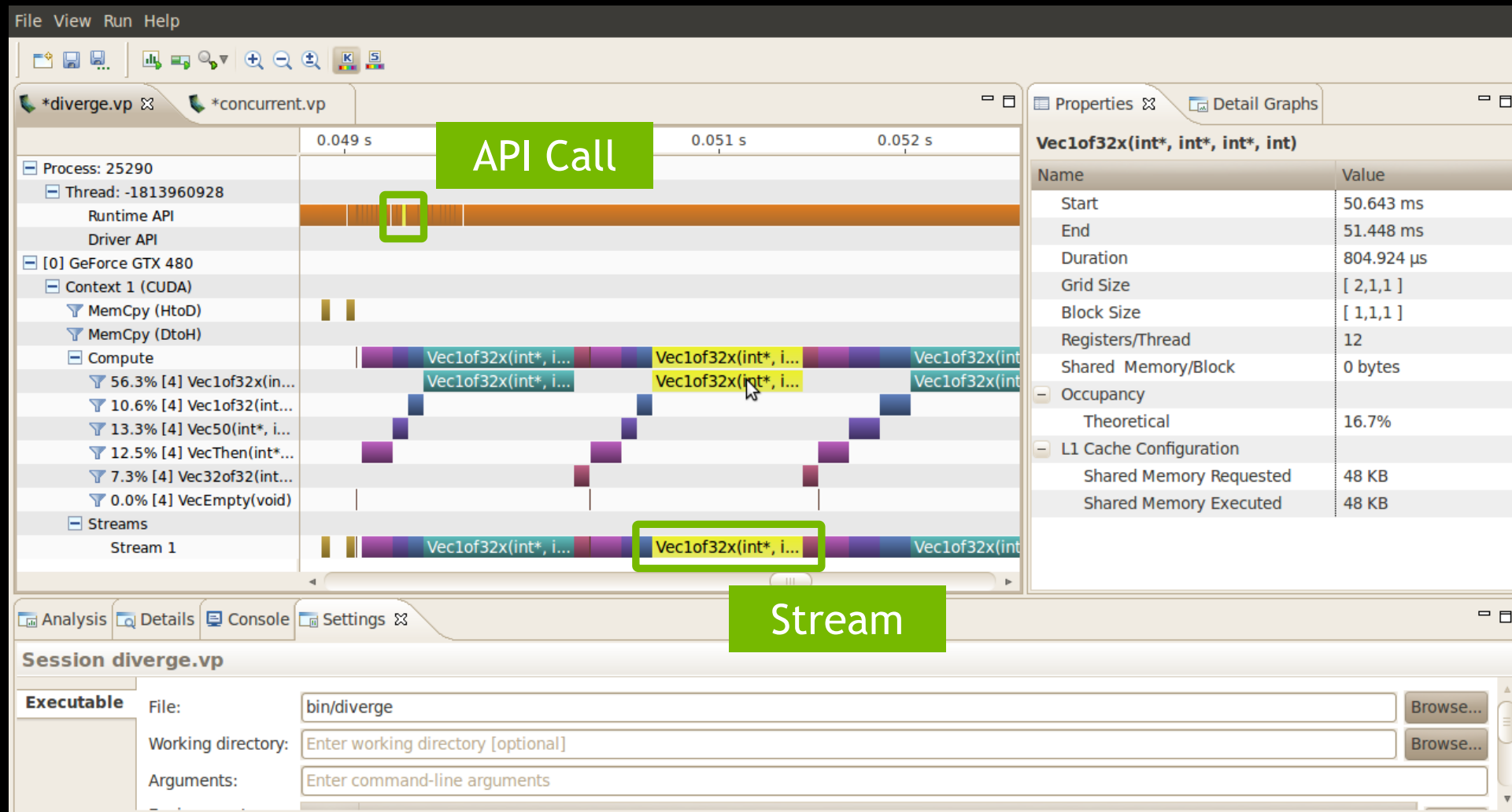




# Measuring Time



# Correlating CPU and GPU Activity



# Properties - Kernel

The screenshot displays the NVIDIA Visual Profiler interface. The left sidebar shows the execution hierarchy: Process (25290) -> Thread (-1813960928) -> Runtime API -> Driver API -> [0] GeForce GTX 480 -> Context 1 (CUDA) -> Compute. The main window shows a timeline of execution from 0.049 s to 0.052 s. A green box highlights the 'Properties' tab for the kernel 'Vec1of32x(int\*, int\*, int\*, int)'. Below the timeline, the 'Session diverge.vp' section shows the executable path 'bin/diverge' and fields for working directory and arguments.

Name	Value
Start	50.643 ms
End	51.448 ms
Duration	804.924 $\mu$ s
Grid Size	[ 2,1,1 ]
Block Size	[ 1,1,1 ]
Registers/Thread	12
Shared Memory/Block	0 bytes
Occupancy	
Theoretical	16.7%
L1 Cache Configuration	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

**Kernel Properties**

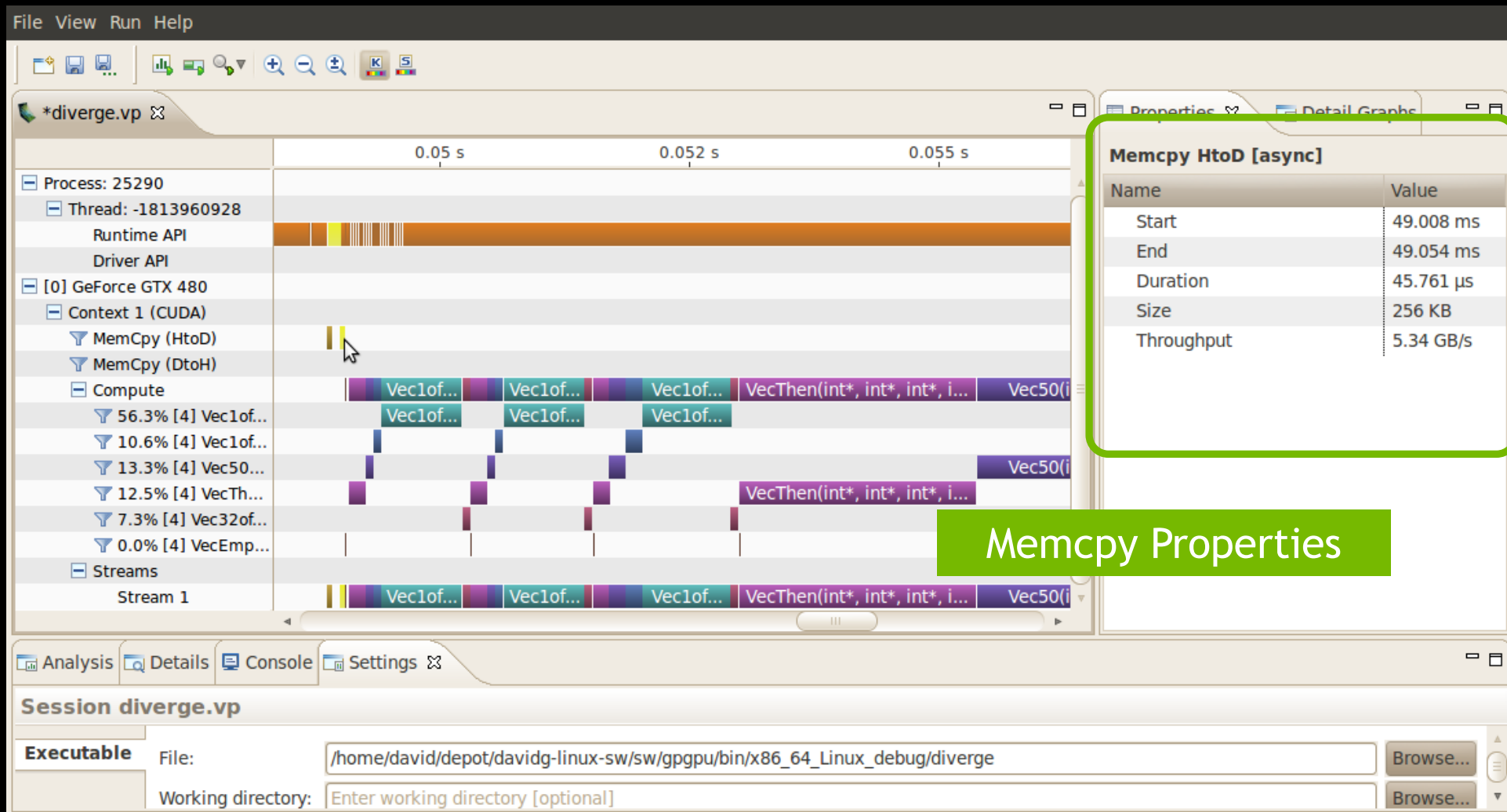
Session diverge.vp

Executable File: bin/diverge [Browse...]

Working directory: [Enter working directory [optional]] [Browse...]

Arguments: [Enter command-line arguments]

# Properties - Malloc





# Analysis, Details, etc.

The screenshot displays the NVIDIA Visual Profiler interface. The main window shows a timeline of GPU activity for a process named 'diverge.vp'. The timeline includes various stages such as 'Runtime API', 'Driver API', 'Context 1 (CUDA)', 'MemCpy (HtoD)', 'MemCpy (DtoH)', 'Compute', and 'Streams'. The 'Compute' section is expanded, showing a detailed view of the execution of the 'Vec32of32(int\*, int\*, int\*, int)' kernel. The timeline shows the kernel's execution duration and the time spent on memory copies and other operations.

On the right side, the 'Properties' panel is visible, showing details for the selected kernel. The properties include:

Name	Value
Start	70.642 ms
End	72.064 ms
Duration	1.422 ms
Grid Size	[ 256,1,1 ]
Block Size	[ 256,1,1 ]
Registers/Thread	11
Shared Memory/Block	0 bytes
Occupancy	
Theoretical	100%
L1 Cache Configuration	
Shared Memory Requested	48 KB

A green box labeled 'Additional Views' is overlaid on the bottom right of the timeline.

At the bottom, the 'Session diverge.vp' panel is visible, showing configuration options for the analysis session:

- Executable**
  - File: bin/diverge
  - Working directory: Enter working directory [optional]
  - Arguments: Enter command-line arguments
  - Environment: Name Value

Compute row shows concurrent kernel execution

Multiple streams launch

## Multiple streams launch independent kernels

# Profiling Flow

- Understand CPU behavior on timeline
  - Add profiling “annotations” to application
  - NVIDIA Tools Extension
    - Custom markers and time ranges
    - Custom naming
- Focus profiling on region of interest
  - Reduce volume of profile data
  - Improve usability of Visual Profiler
  - Improve accuracy of analysis
- Analyze for optimization opportunities

# Annotations: NVIDIA Tools Extension

- Developer API for CPU code
- Installed with CUDA Toolkit (`libnvToolsExt.so`)
- Naming
  - Host OS threads: `nvtxNameOsThread()`
  - CUDA device, context, stream: `nvtxNameCudaStream()`
- Time Ranges and Markers
  - Range: `nvtxRangeStart()`, `nvtxRangeEnd()`
  - Instantaneous marker: `nvtxMark()`

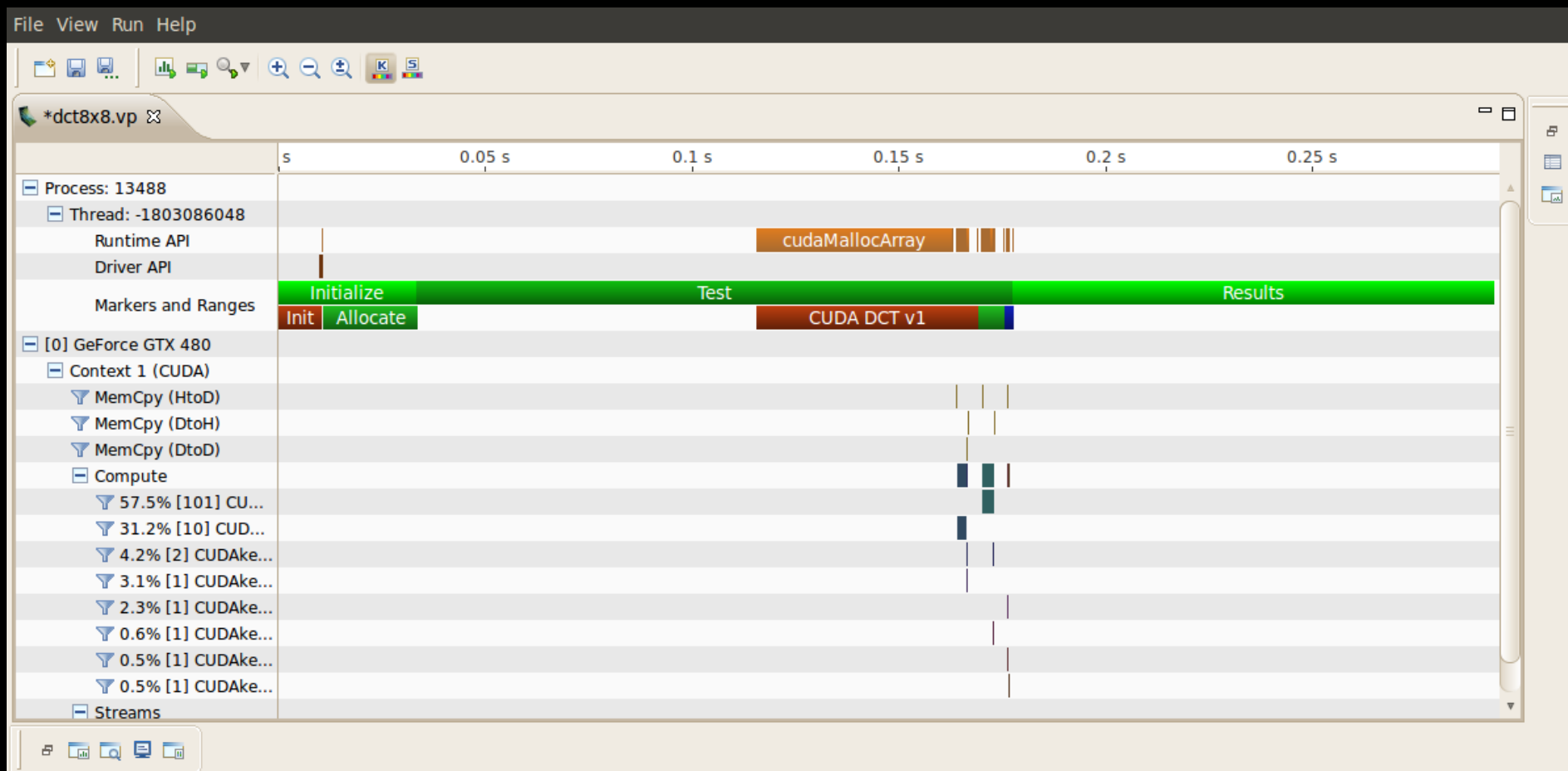


# Example: Time Ranges

- Testing algorithm in testbench
- Use time ranges API to mark initialization, test, and results

```
...  
nvtxRangeId_t id0 = nvtxRangeStart("Initialize");  
< init code >  
nvtxRangeEnd(id0);  
nvtxRangeId_t id1 = nvtxRangeStart("Test");  
< compute code >  
nvtxRangeEnd(id1);  
...
```

# Example: Time Ranges



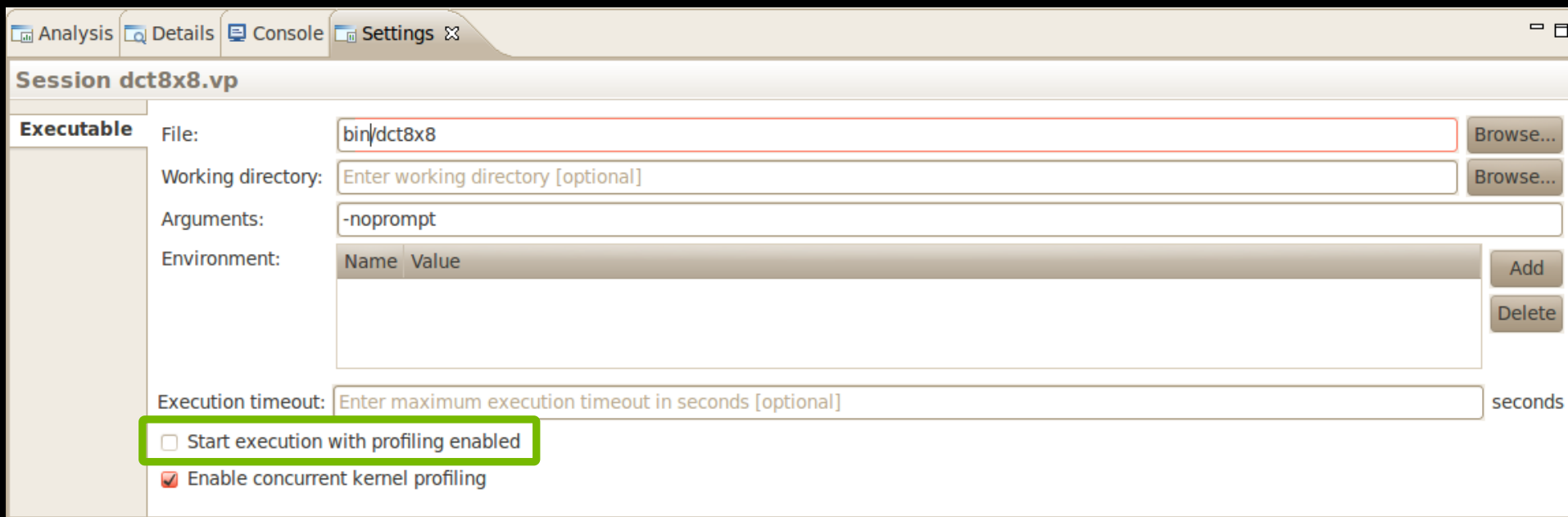
# Profile Region Of Interest

- `cudaProfilerStart()` / `cudaProfilerStop()` in CPU code
- Specify representative subset of app execution
  - Manual exploration and analysis simplified
  - Automated analysis focused on performance critical code

```
for (i = 0; i < N; i++) {  
    if (i == 12) cudaProfilerStart();  
    <loop body>  
    if (i == 15) cudaProfilerStop();  
}
```

# Enable Region Of Interest

- Insert `cudaProfilerStart()` / `cudaProfilerStop()`
- Disable profiling at start of application



Analysis Details Console Settings

Session **dct8x8.vp**

**Executable**

File:  Browse...

Working directory:  Browse...

Arguments:

Environment:

Name	Value
------	-------

Add Delete

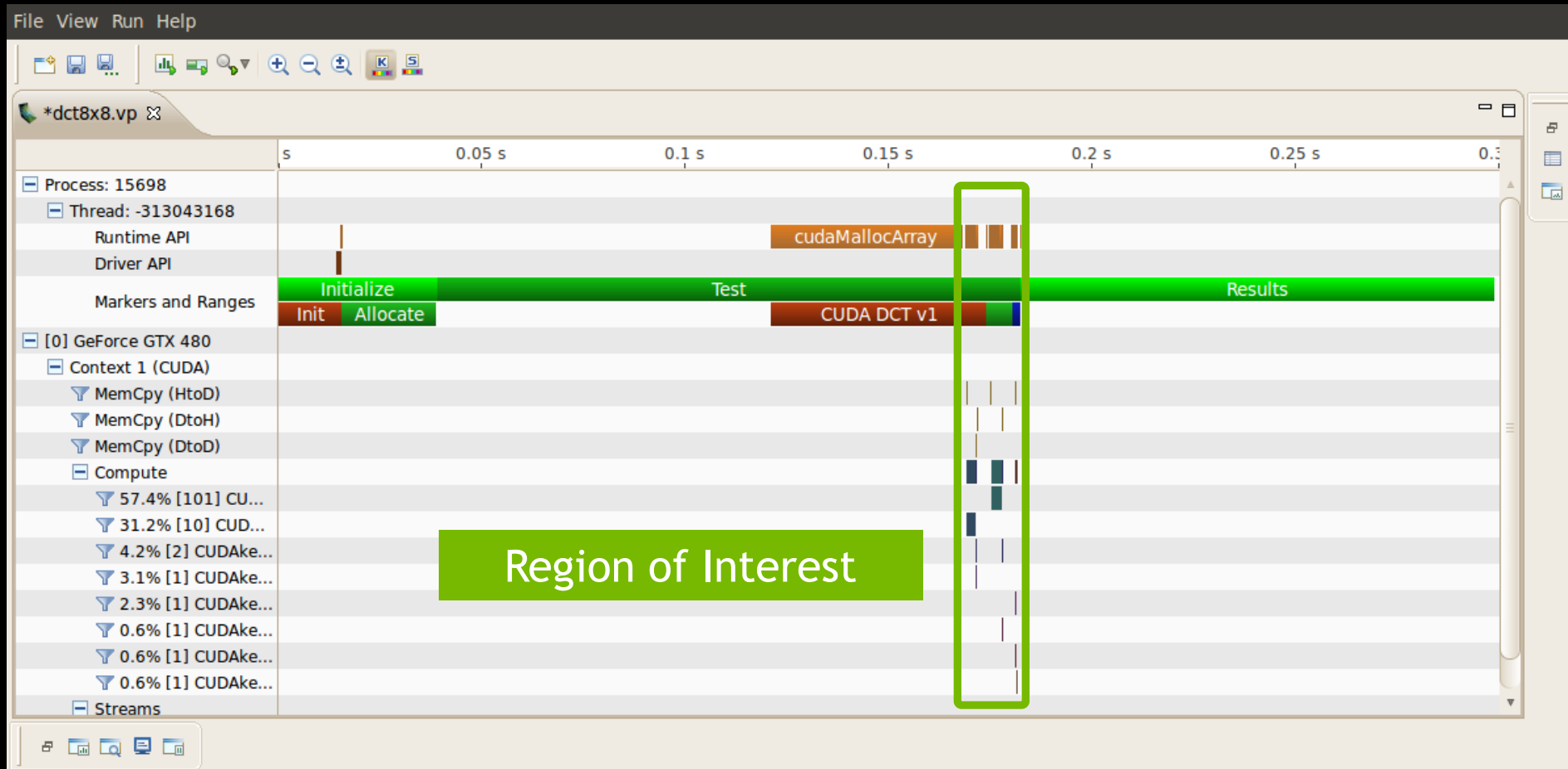
Execution timeout:  seconds

☐ Start execution with profiling enabled

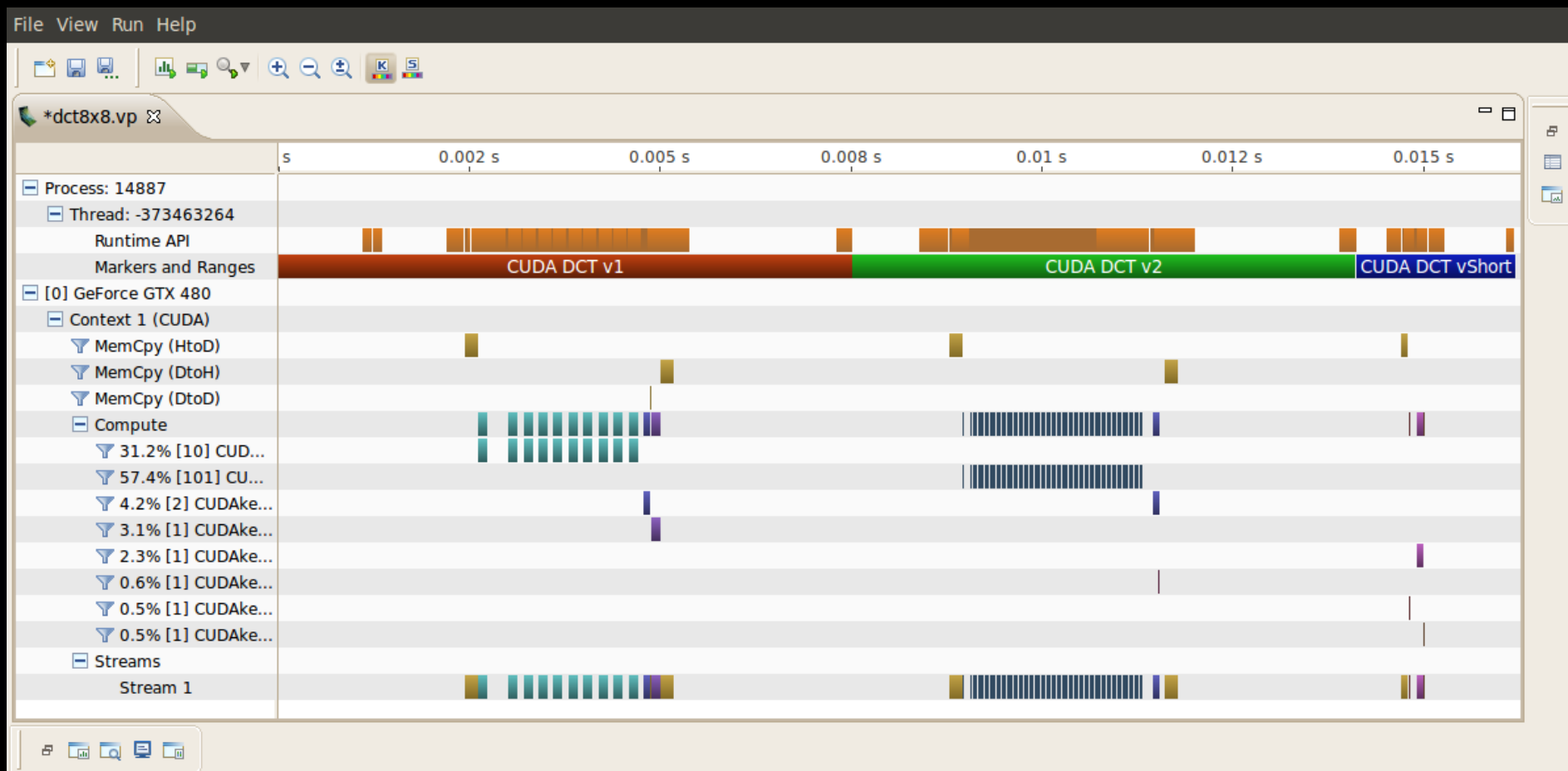
☒ Enable concurrent kernel profiling



# Example: Without cudaProfilerStart/Stop

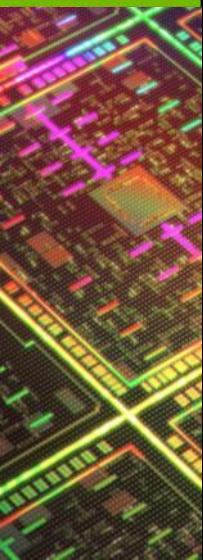


# Example: With cudaProfilerStart/Stop



# Analysis

- Visual inspection of timeline
- Automated Analysis
- Metrics and Events



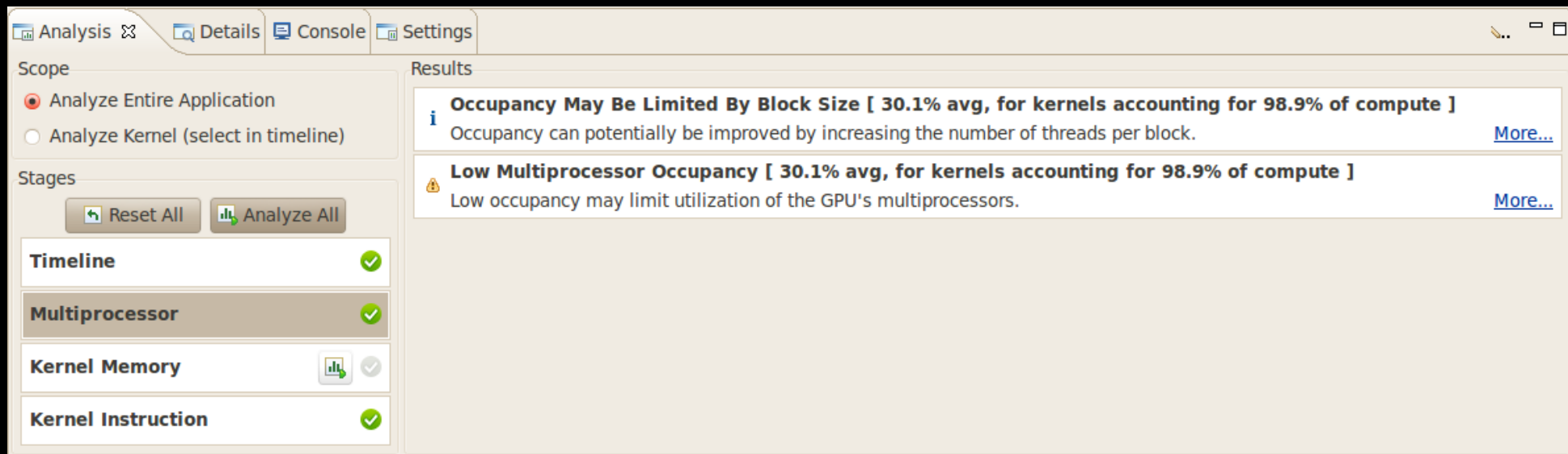
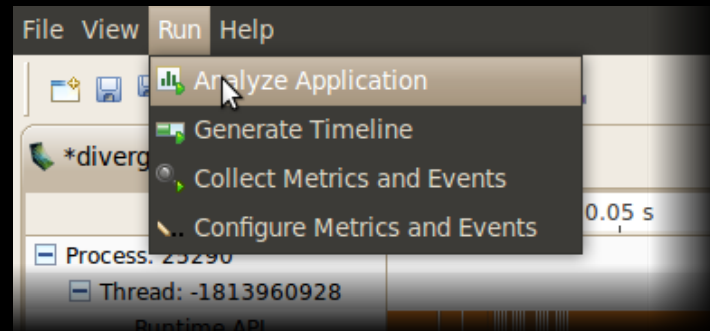
# Visual Inspection

- Understand CPU/GPU interactions
  - Use nvToolsExt to mark time ranges on CPU
  - Is application taking advantage of both CPU and GPU?
  - Is CPU waiting on GPU? Is GPU waiting on CPU?
- Look for potential concurrency opportunities
  - Overlap memcpy and kernel
  - Concurrent kernels
- Automated analysis does some of this




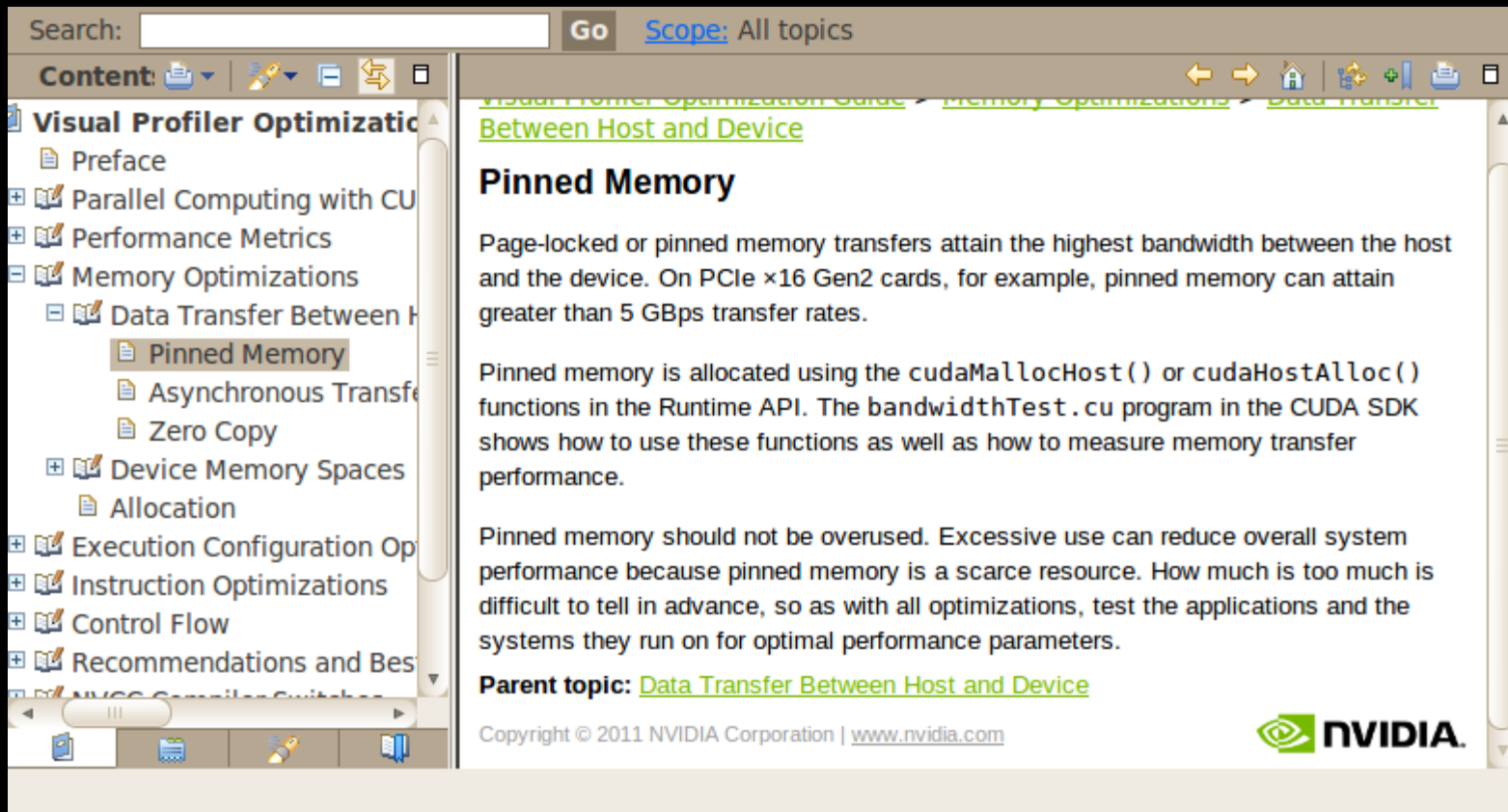
# Automated Analysis - Application

- Analyze entire application
  - Timeline
  - Hardware performance counters



# Analysis Documentation

 **Low Memcpy Throughput [ 997.19 MB/s avg, for memcpy accounting for 68.1% of all memcpy time ]**  
The memory copies are not fully using the available host to device bandwidth. [More..](#)



The screenshot shows a web browser displaying the NVIDIA Visual Profiler documentation. The left sidebar contains a table of contents with the following items: Visual Profiler Optimization, Preface, Parallel Computing with CUDA, Performance Metrics, Memory Optimizations (expanded), Data Transfer Between Host and Device (expanded), Pinned Memory (highlighted), Asynchronous Transfers, Zero Copy, Device Memory Spaces (expanded), Allocation, Execution Configuration Options, Instruction Optimizations, Control Flow, Recommendations and Best Practices, and NVCC Compiler Switches. The main content area is titled "Pinned Memory" and contains the following text:


Page-locked or pinned memory transfers attain the highest bandwidth between the host and the device. On PCIe x16 Gen2 cards, for example, pinned memory can attain greater than 5 GBps transfer rates.

Pinned memory is allocated using the `cudaMallocHost()` or `cudaHostAlloc()` functions in the Runtime API. The `bandwidthTest.cu` program in the CUDA SDK shows how to use these functions as well as how to measure memory transfer performance.

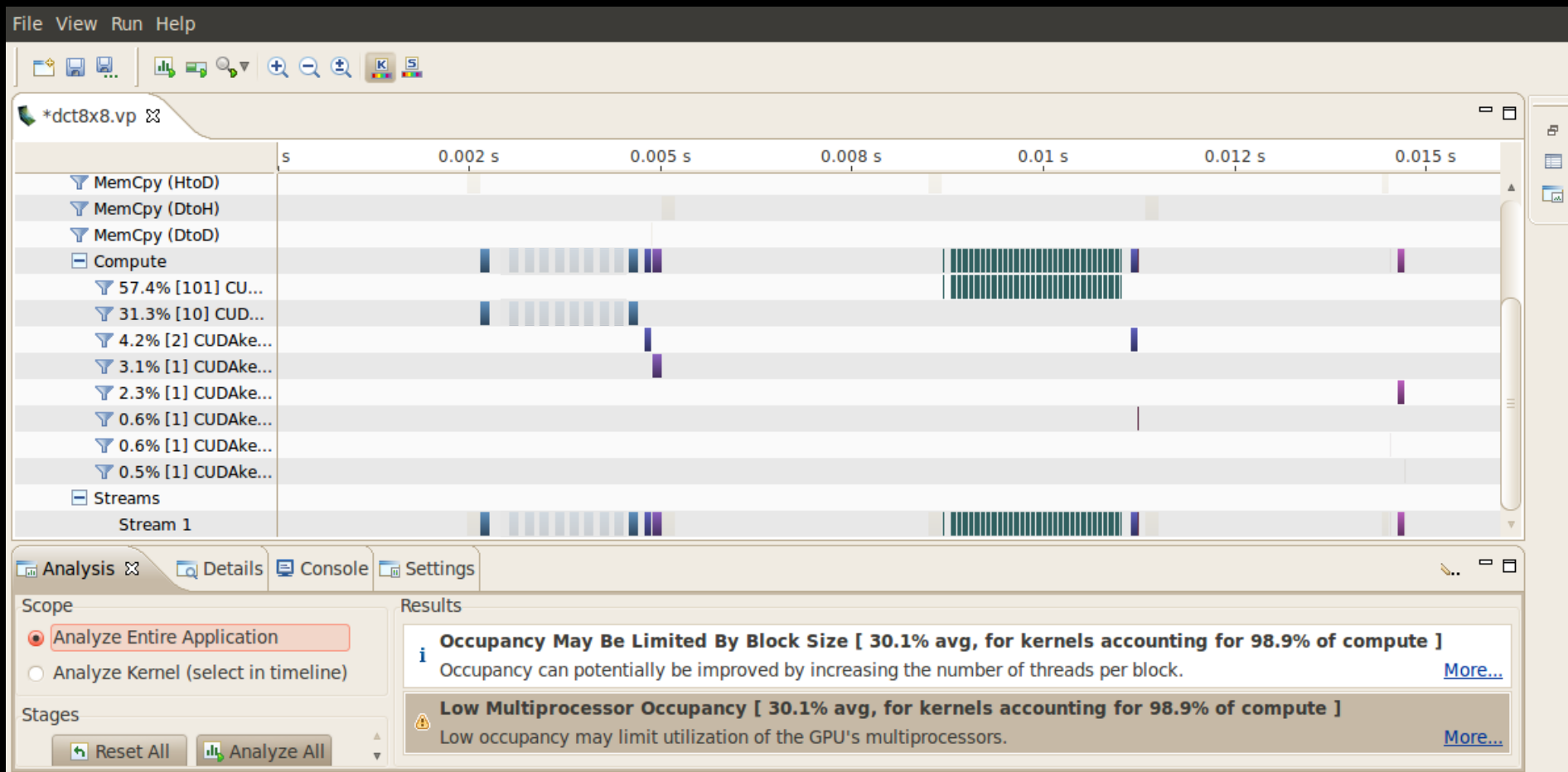
Pinned memory should not be overused. Excessive use can reduce overall system performance because pinned memory is a scarce resource. How much is too much is difficult to tell in advance, so as with all optimizations, test the applications and the systems they run on for optimal performance parameters.

**Parent topic:** [Data Transfer Between Host and Device](#)

Copyright © 2011 NVIDIA Corporation | [www.nvidia.com](http://www.nvidia.com)



# Results Correlated With Timeline

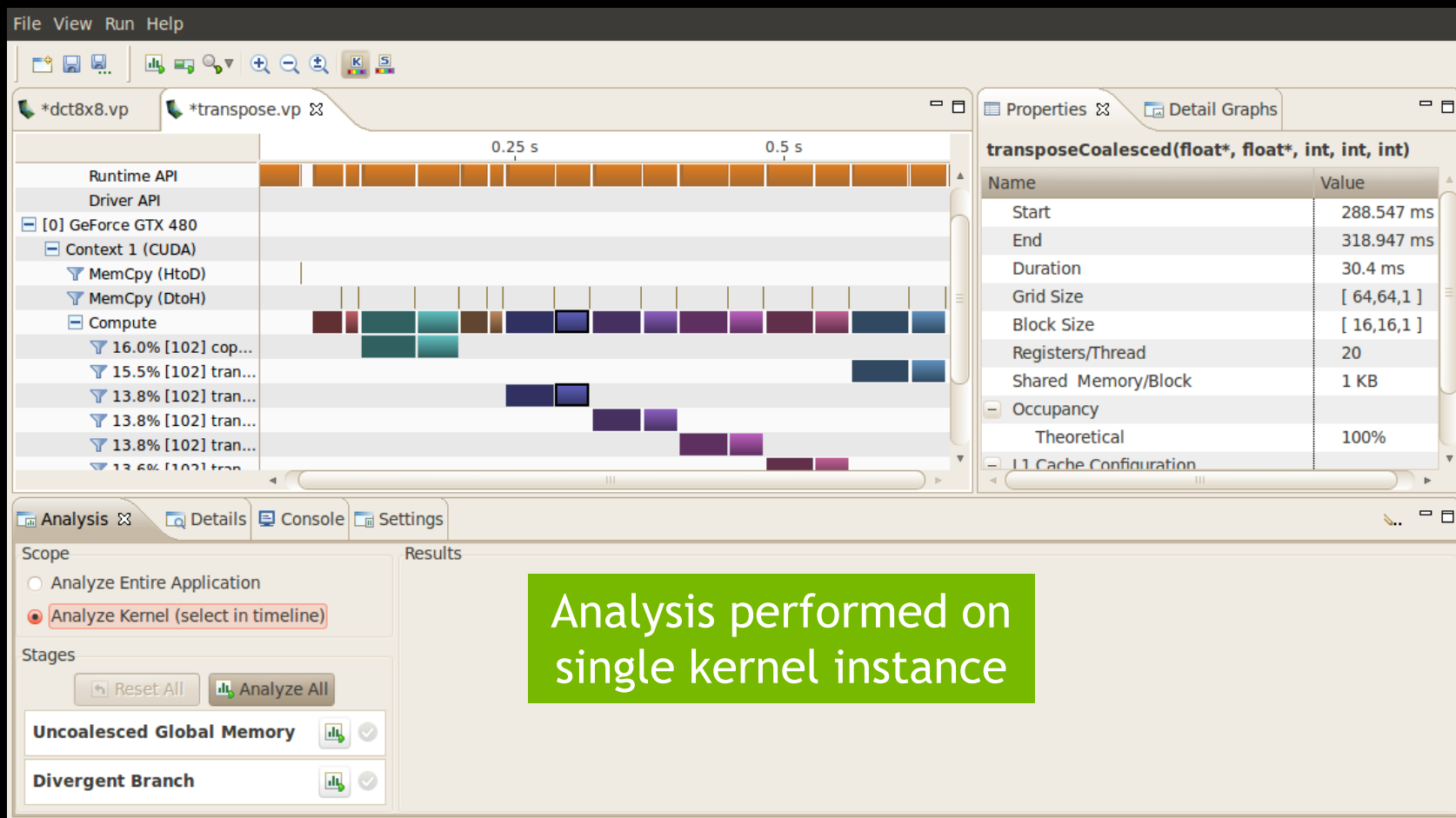


# Analysis Properties

- Highlight a kernel or memcpy in timeline
  - Properties shows analysis results for that specific kernel / memcpy
  - Optimization opportunities are flagged

Properties		Detail Graphs
CUDatakernel2DCT(float*, float*, int)		
Name	Value	
Duration	21.117 μs	
Grid Size	[ 16,32,1 ]	
Block Size	[ 8,4,2 ]	
Registers/Thread	35	
Shared Memory/Block	2.062 KB	
Memory		
Global Load Efficiency	100%	
Global Store Efficiency	100%	
Instruction		
Branch Divergence Overhead	0%	
Occupancy		
Achieved	⚠ 29.4%	
Theoretical	33.3%	
Limiter	Block Size	
L1 Cache Configuration		
Shared Memory Requested	48 KB	
Shared Memory Executed	48 KB	

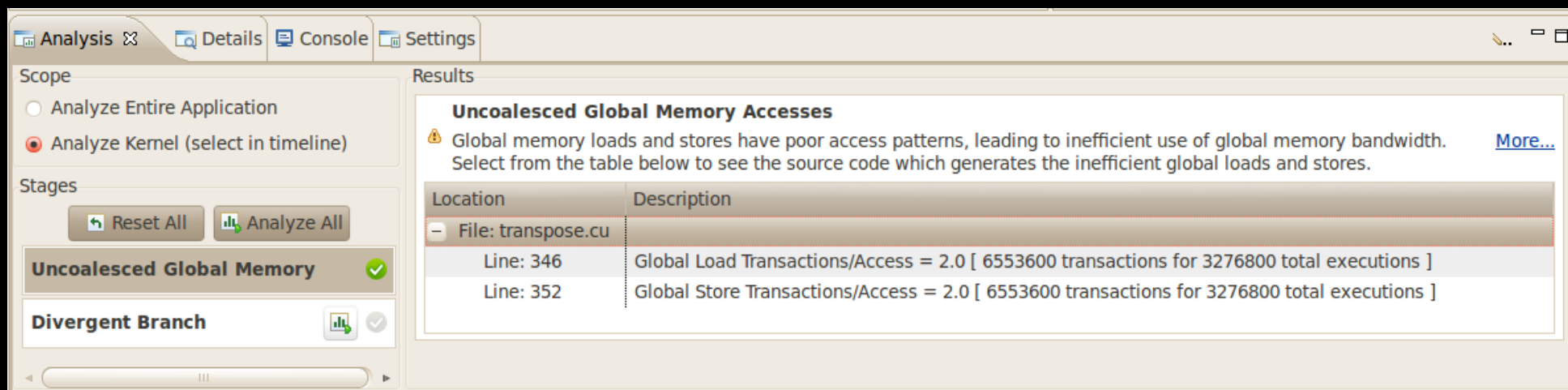
# Automated Analysis - Single Kernel





# Uncoalesced Global Memory Accesses

- Access pattern determines number of memory transactions
  - Report loads/stores where access pattern is inefficient



The screenshot shows the Analysis window in NVIDIA Nsight Visual Studio Edition. The left sidebar has tabs for Analysis, Details, Console, and Settings. Under the Analysis tab, the Scope is set to 'Analyze Kernel (select in timeline)'. The Stages list includes 'Uncoalesced Global Memory' (checked) and 'Divergent Branch'. The Results pane displays the title 'Uncoalesced Global Memory Accesses' with a warning icon. Below the title, a message states: 'Global memory loads and stores have poor access patterns, leading to inefficient use of global memory bandwidth. Select from the table below to see the source code which generates the inefficient global loads and stores.' A 'More...' link is present. A table follows with two columns: 'Location' and 'Description'.

Location	Description
File: transpose.cu	
Line: 346	Global Load Transactions/Access = 2.0 [ 6553600 transactions for 3276800 total executions ]
Line: 352	Global Store Transactions/Access = 2.0 [ 6553600 transactions for 3276800 total executions ]

# Source Correlation

The screenshot displays the NVIDIA Visual Profiler interface with the following components:

- Code Editor:** Shows the source code for `transpose.cu`. The line `tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];` is highlighted, corresponding to the selected analysis results.
- Properties Panel:** Displays performance metrics for the `transposeCoalesced(float*, float*, int, int, int)` kernel.
 

Name	Value
Start	288.547 ms
End	318.947 ms
Duration	30.4 ms
Grid Size	[ 64,64,1 ]
Block Size	[ 16,16,1 ]
Registers/Thread	20
Shared Memory/Block	1 KB
Occupancy	
Theoretical	100%
L1 Cache Configuration	
- Analysis Panel:** Shows the selected analysis type as **Uncoalesced Global Memory** with a green checkmark.
- Results Panel:** Displays the **Uncoalesced Global Memory Accesses** results.
 

Global memory loads and stores have poor access patterns, leading to inefficient use of global memory bandwidth. [More...](#)

Location	Description
File: transpose	
Line: 268	Global Load Transactions/Access = 2.0 [ 6553600 transactions for 3276800 total executions ]
Line: 274	Global Store Transactions/Access = 2.0 [ 6553600 transactions for 3276800 total executions ]

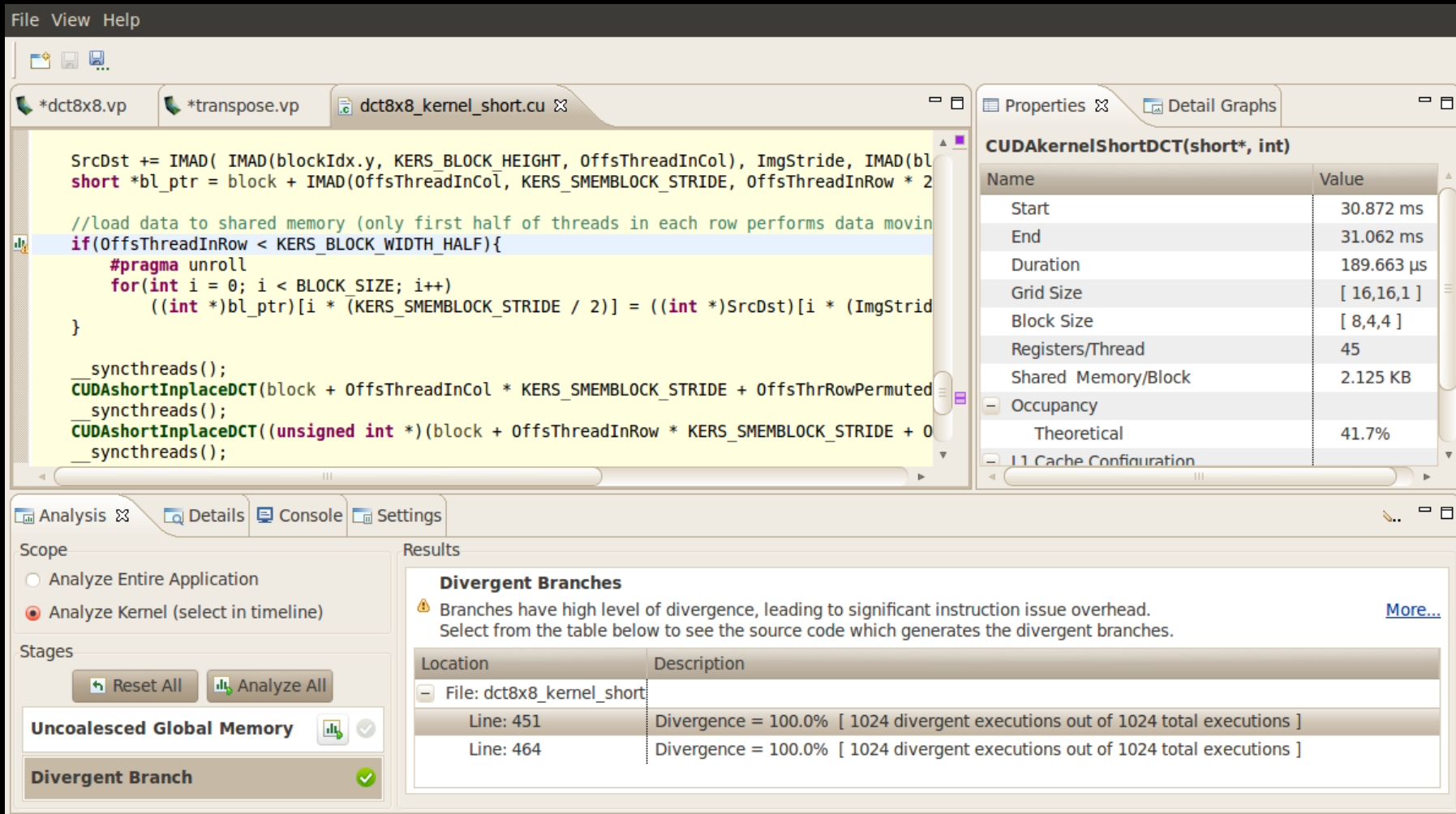
# Divergent Branches

- Divergent control-flow for threads within a warp
  - Report branches that have high average divergence

The screenshot displays the GPU Analysis tool interface. The left sidebar contains the 'Scope' section with 'Analyze Kernel (select in timeline)' selected, and the 'Stages' section with 'Uncoalesced Global Memory' and 'Divergent Branch' (marked with a green checkmark). The main 'Results' pane shows a warning icon and the title 'Divergent Branches'. It contains a message about high divergence leading to instruction issue overhead, with a 'More...' link. Below this is a table with two columns: 'Location' and 'Description'.

Location	Description
File: dct8x8_kernel_short	
Line: 451	Divergence = 100.0% [ 1024 divergent executions out of 1024 total executions ]
Line: 464	Divergence = 100.0% [ 1024 divergent executions out of 1024 total executions ]

# Source Correlation



The screenshot displays the NVIDIA Visual Profiler interface, showing the source code of a CUDA kernel and its performance metrics.

**Source Code:**

```
SrcDst += IMAD( IMAD(blockIdx.y, KERS_BLOCK_HEIGHT, OffsThreadInCol), ImgStride, IMAD(bl
short *bl_ptr = block + IMAD(OffsThreadInCol, KERS_SMEMBLOCK_STRIDE, OffsThreadInRow * 2

//load data to shared memory (only first half of threads in each row performs data movin
if(OffsThreadInRow < KERS_BLOCK_WIDTH_HALF){
    #pragma unroll
    for(int i = 0; i < BLOCK_SIZE; i++)
        ((int *)bl_ptr)[i * (KERS_SMEMBLOCK_STRIDE / 2)] = ((int *)SrcDst)[i * (ImgStrid
}

__syncthreads();
CUDAShortInplaceDCT(block + OffsThreadInCol * KERS_SMEMBLOCK_STRIDE + OffsThrRowPermuted
__syncthreads();
CUDAShortInplaceDCT((unsigned int *) (block + OffsThreadInRow * KERS_SMEMBLOCK_STRIDE + 0
__syncthreads();
```

**Properties Panel:**

Name	Value
Start	30.872 ms
End	31.062 ms
Duration	189.663 $\mu$ s
Grid Size	[ 16,16,1 ]
Block Size	[ 8,4,4 ]
Registers/Thread	45
Shared Memory/Block	2.125 KB
Occupancy	
Theoretical	41.7%
L1 Cache Configuration	

**Analysis Panel:**

Scope:

- ☐ Analyze Entire Application
- ☒ Analyze Kernel (select in timeline)

Stages:

**Uncoalesced Global Memory** ☒

**Divergent Branch** ☒

**Results Panel:**

**Divergent Branches**

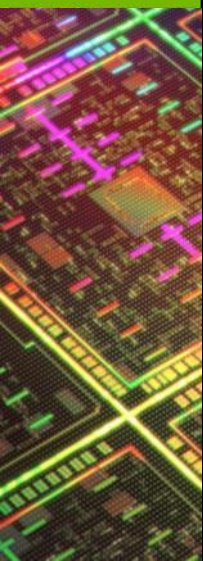
Branches have high level of divergence, leading to significant instruction issue overhead. [More...](#)

Select from the table below to see the source code which generates the divergent branches.

Location	Description
File: dct8x8_kernel_short	
Line: 451	Divergence = 100.0% [ 1024 divergent executions out of 1024 total executions ]
Line: 464	Divergence = 100.0% [ 1024 divergent executions out of 1024 total executions ]

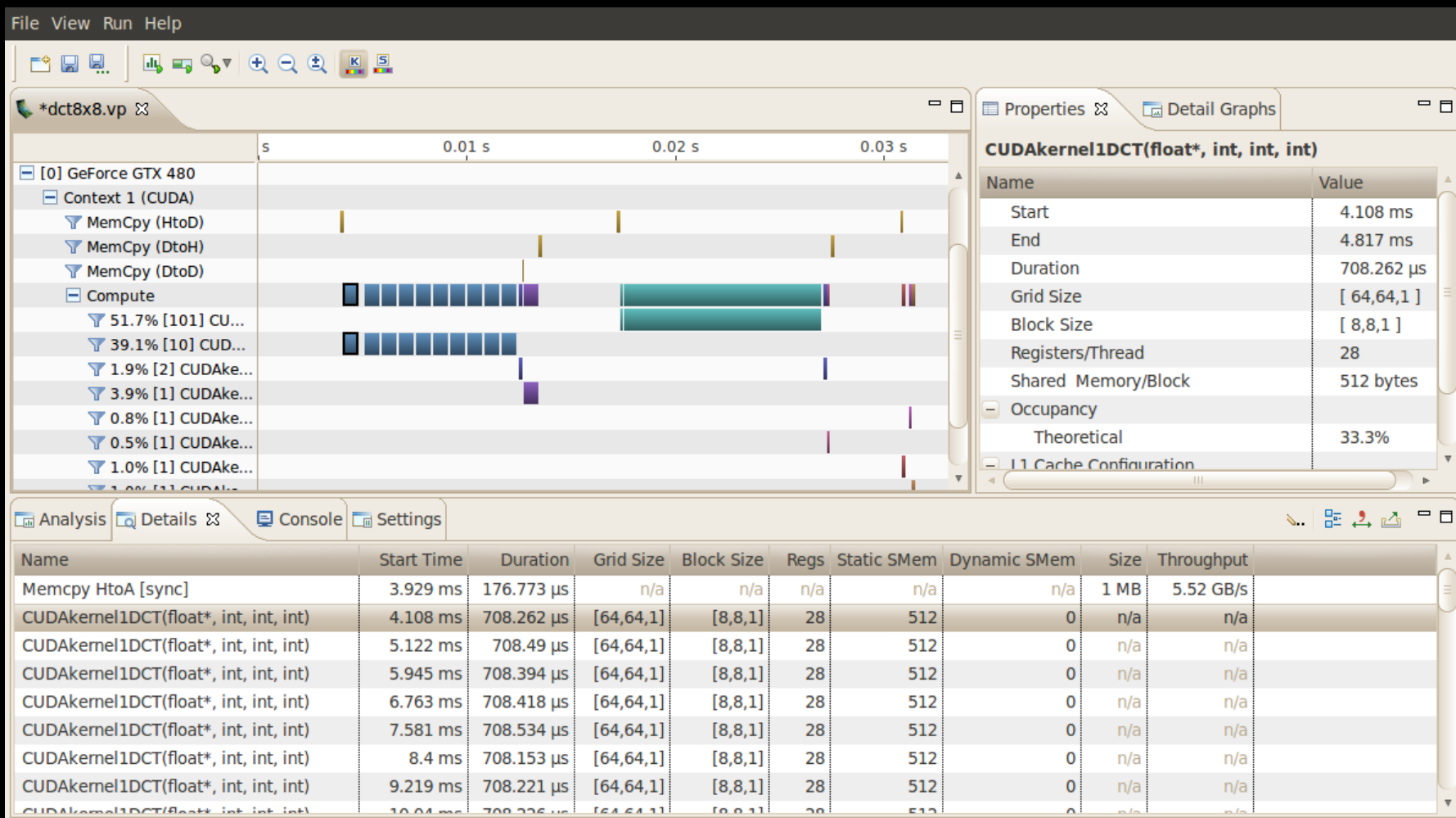
# Enabling Source Correlation

- Source correlation requires that source/line information be embedded in executable
  - Available in debug executables: `nvcc -G`
  - New flag for optimized executables: `nvcc -lineinfo`

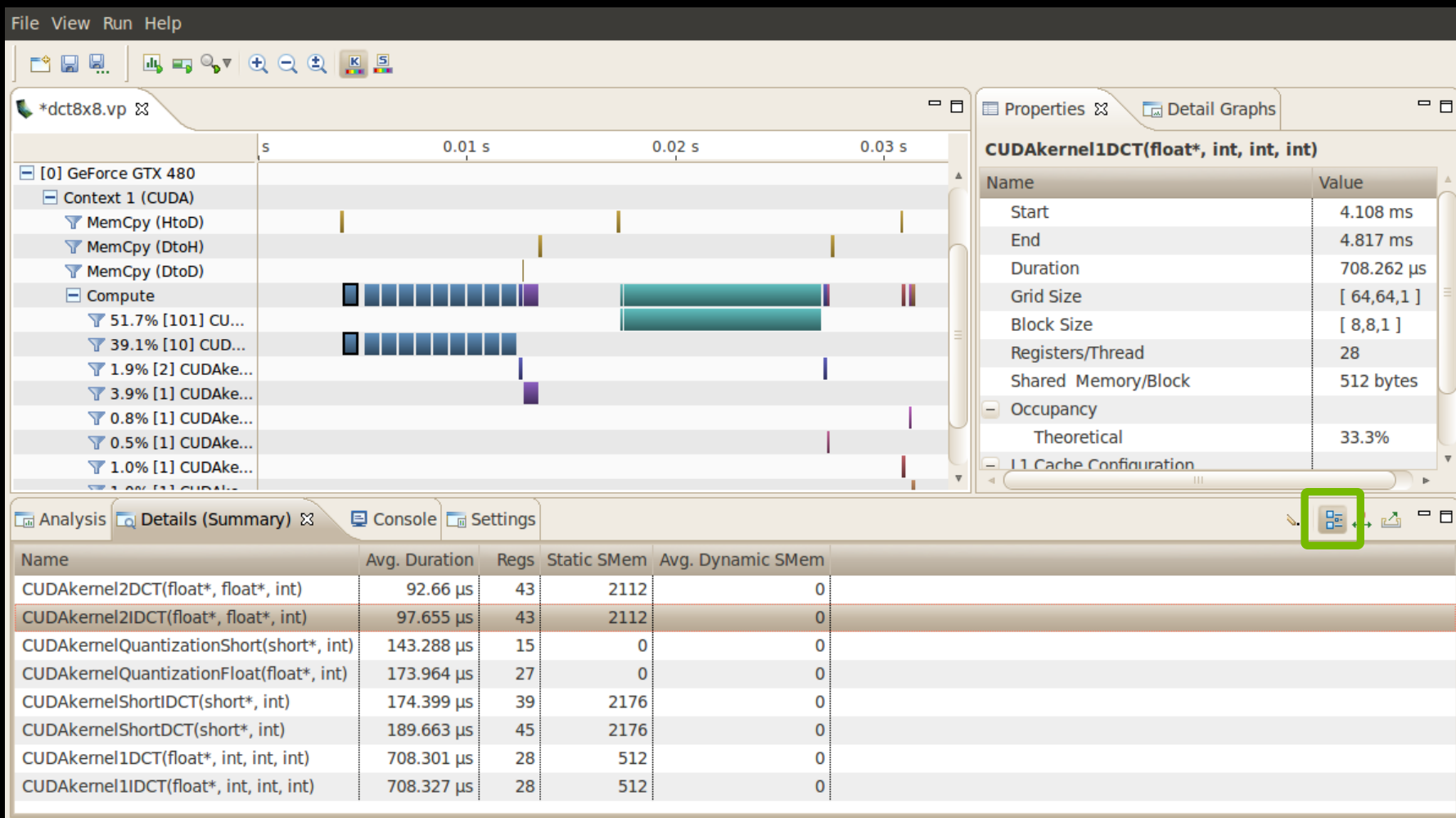




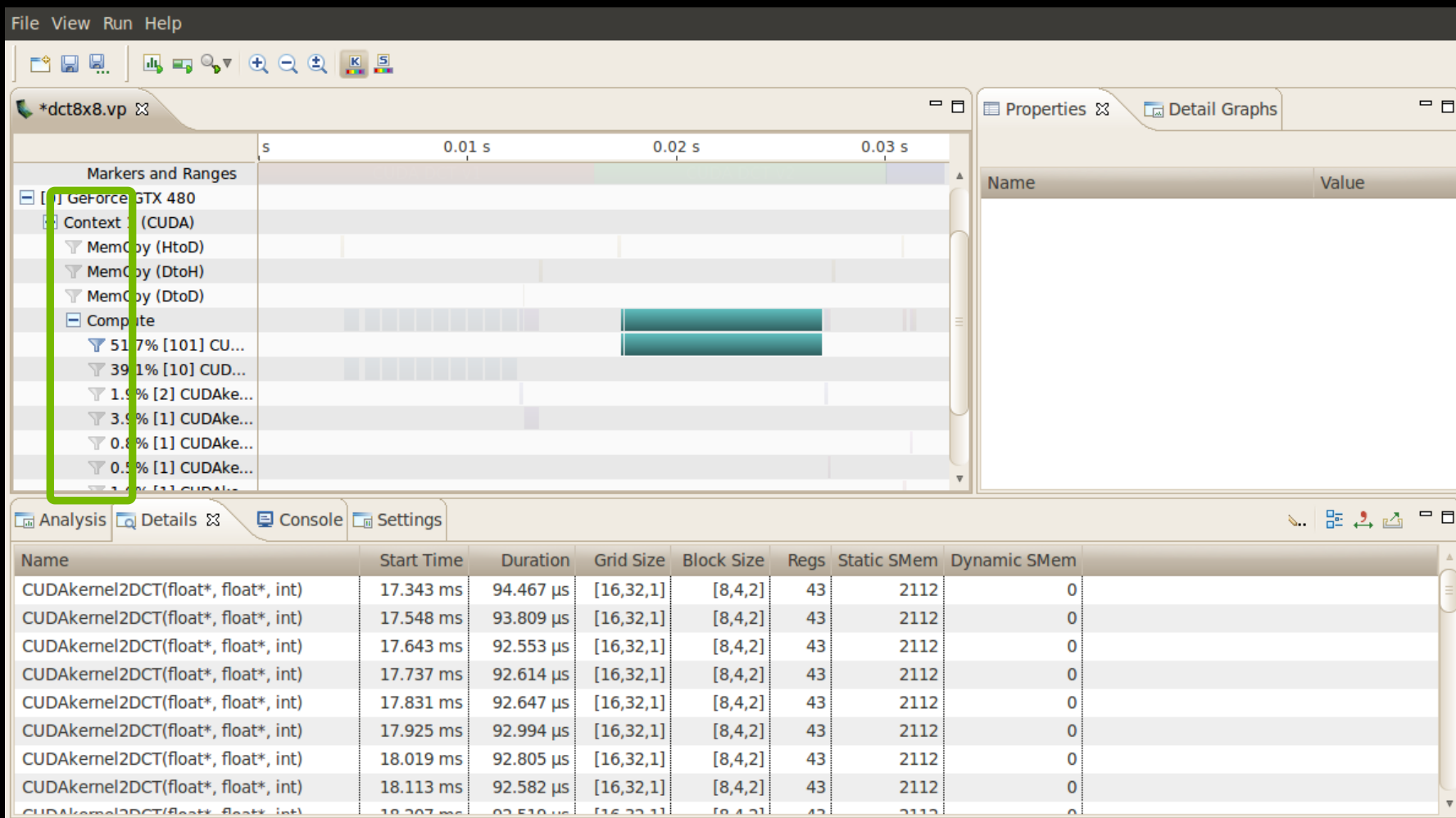
# Detailed Profile Data



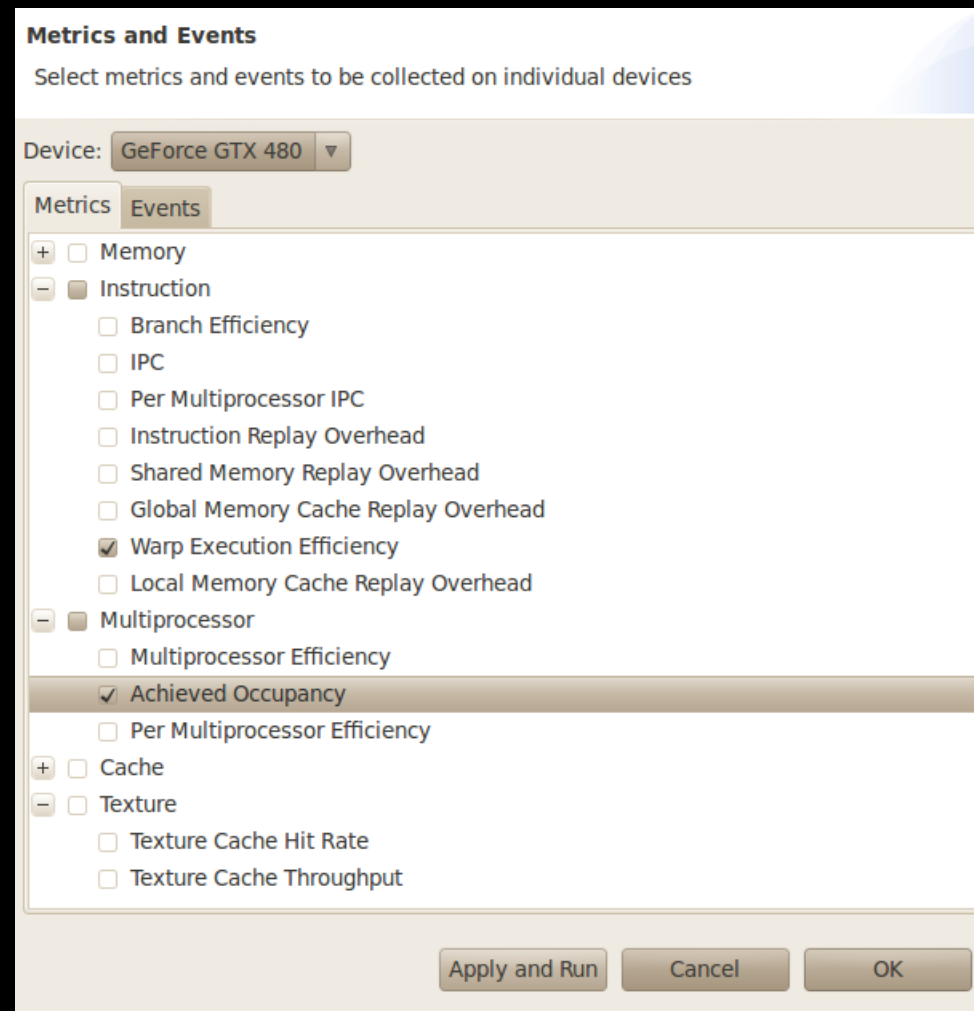
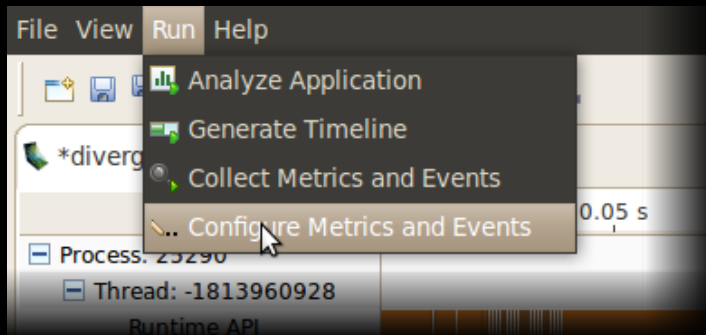
# Detailed Summary Profile Data



# Filtering



# Metrics and Events



# Metrics and Events

Analysis Details Console Settings										
Name	Start Time	Duration	Warp Execution Efficiency	Achieved Occupancy	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	
Memcpy HtoA [sync]	3.929 ms	176.773 $\mu$ s	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
CUDAKernel1DCT(float*, int, int, int)	4.108 ms	708.262 $\mu$ s	100%	0.328	[64,64,1]	[8,8,1]	28	512	0	
CUDAKernel1DCT(float*, int, int, int)	5.122 ms	708.49 $\mu$ s	100%	0.328	[64,64,1]	[8,8,1]	28	512	0	
CUDAKernel1DCT(float*, int, int, int)	5.945 ms	708.394 $\mu$ s	100%	0.327	[64,64,1]	[8,8,1]	28	512	0	
CUDAKernel1DCT(float*, int, int, int)	6.763 ms	708.418 $\mu$ s	100%	0.328	[64,64,1]	[8,8,1]	28	512	0	
CUDAKernel1DCT(float*, int, int, int)	7.581 ms	708.534 $\mu$ s	100%	0.327	[64,64,1]	[8,8,1]	28	512	0	
CUDAKernel1DCT(float*, int, int, int)	8.4 ms	708.153 $\mu$ s	100%	0.327	[64,64,1]	[8,8,1]	28	512	0	
CUDAKernel1DCT(float*, int, int, int)	9.219 ms	708.221 $\mu$ s	100%	0.327	[64,64,1]	[8,8,1]	28	512	0	

Analysis Details (Summary) Console Settings							
Name	Warp Execution Efficiency	Achieved Occupancy	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	
CUDAKernel2DCT(float*, float*, int)	100%	0.3	92.66 $\mu$ s	43	2112	0	
CUDAKernel2IDCT(float*, float*, int)	100%	0.302	97.655 $\mu$ s	43	2112	0	
CUDAKernelQuantizationShort(short*, int)	67.5%	0.317	143.288 $\mu$ s	15	0	0	
CUDAKernelQuantizationFloat(float*, int)	98.7%	0.318	173.964 $\mu$ s	27	0	0	
CUDAKernelShortIDCT(short*, int)	74.7%	0.468	174.399 $\mu$ s	39	2176	0	
CUDAKernelShortDCT(short*, int)	75%	0.376	189.663 $\mu$ s	45	2176	0	
CUDAKernel1DCT(float*, int, int, int)	100%	0.328	708.301 $\mu$ s	28	512	0	
CUDAKernel1IDCT(float*, int, int, int)	100%	0.328	708.327 $\mu$ s	28	512	0	



# nvprof

- Textual reports
  - Summary of GPU and CPU activity
  - Trace of GPU and CPU activity
  - Event collection
- Headless profile collection
  - Use nvprof on headless node to collect data
  - Visualize timeline with Visual Profiler

# nvprof Usage

```
$ nvprof [nvprof_args] <app> [app_args]
```

- Argument help

```
$ nvprof --help
```

# nvprof - GPU Summary

```
$ nvprof dct8x8
```

```
===== Profiling result:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
49.52	9.36ms	101	92.68us	92.31us	94.31us	CUDAKernel2DCT(float*, float*, int)
37.47	7.08ms	10	708.31us	707.99us	708.50us	CUDAKernel1DCT(float*,int, int,int)
3.75	708.42us	1	708.42us	708.42us	708.42us	CUDAKernel1IDCT(float*,int,int,int)
1.84	347.99us	2	173.99us	173.59us	174.40us	CUDAKernelQuantizationFloat()
1.75	331.37us	2	165.69us	165.67us	165.70us	[CUDA memcpy DtoH]
1.41	266.70us	2	133.35us	89.70us	177.00us	[CUDA memcpy HtoD]
1.00	189.64us	1	189.64us	189.64us	189.64us	CUDAKernelShortDCT(short*, int)
0.94	176.87us	1	176.87us	176.87us	176.87us	[CUDA memcpy HtoA]
0.92	174.16us	1	174.16us	174.16us	174.16us	CUDAKernelShortIDCT(short*, int)
0.76	143.31us	1	143.31us	143.31us	143.31us	CUDAKernelQuantizationShort(short*)
0.52	97.75us	1	97.75us	97.75us	97.75us	CUDAKernel2IDCT(float*, float*)
0.12	22.59us	1	22.59us	22.59us	22.59us	[CUDA memcpy DtoA]

# nvprof - GPU Summary (csv)

```
$ nvprof --csv dct8x8
```

```
===== Profiling result:
```

```
Time(%),Time,Calls,Avg,Min,Max,Name
```

```
,ms,,us,us,us,
```

```
49.51,9.35808,101,92.65400,92.38200,94.19000,"CUdAkernel2DCT(float*, float*, int)"
```

```
37.47,7.08288,10,708.2870,707.9360,708.7070,"CUdAkernel1DCT(float*, int, int, int)"
```

```
3.75,0.70847,1,708.4710,708.4710,708.4710,"CUdAkernel1IDCT(float*, int, int, int)"
```

```
1.84,0.34802,2,174.0090,173.8130,174.2060,"CUdAkernelQuantizationFloat(float*, int)"
```

```
1.75,0.33137,2,165.6850,165.6690,165.7020,"[CUdA memcpy DtoH]"
```

```
1.42,0.26759,2,133.7970,89.89100,177.7030,"[CUdA memcpy HtoD]"
```

```
1.00,0.18874,1,188.7360,188.7360,188.7360,"CUdAkernelShortDCT(short*, int)"
```

```
0.94,0.17687,1,176.8690,176.8690,176.8690,"[CUdA memcpy HtoA]"
```

```
0.93,0.17594,1,175.9390,175.9390,175.9390,"CUdAkernelShortIDCT(short*, int)"
```

```
0.76,0.14281,1,142.8130,142.8130,142.8130,"CUdAkernelQuantizationShort(short*, int)"
```

```
0.52,0.09758,1,97.57800,97.57800,97.57800,"CUdAkernel2IDCT(float*, float*, int)"
```

```
0.12,0.02259,1,22.59300,22.59300,22.59300,"[CUdA memcpy DtoA]"
```

# nvprof - GPU Trace

```
$ nvprof --print-gpu-trace dct8x8
```

```
===== Profiling result:
```

Start	Duration	Grid Size	Block Size	Regs	SSMem	DSMem	Size	Throughput	Name
167.82ms	176.84us	-	-	-	-	-	1.05MB	5.93GB/s	[CUDA memcpy HtoA]
168.00ms	708.51us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
168.95ms	708.51us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
169.74ms	708.26us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
170.53ms	707.89us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
171.32ms	708.12us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
172.11ms	708.05us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
172.89ms	708.38us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
173.68ms	708.31us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
174.47ms	708.15us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
175.26ms	707.95us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
176.05ms	173.87us	(64 64 1)	(8 8 1)	27	0B	0B	-	-	CUDAKernelQuantization (...)
176.23ms	22.82us	-	-	-	-	-	1.05MB	45.96GB/s	[CUDA memcpy DtoA]



# nvprof - CPU/GPU Trace

```
$ nvprof --print-gpu-trace --print-api-trace dct8x8
```

```
===== Profiling result:
```

Start	Duration	Grid Size	Block Size	Regs	SSMem	DSMem	Size	Throughput	Name
167.82ms	176.84us	-	-	-	-	-	1.05MB	5.93GB/s	[CUDA memcpy HtoA]
167.81ms	2.00us	-	-	-	-	-	-	-	cudaSetupArgument
167.81ms	38.00us	-	-	-	-	-	-	-	cudaLaunch
167.85ms	1.00ms	-	-	-	-	-	-	-	cudaDeviceSynchronize
168.00ms	708.51us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)
168.86ms	2.00us	-	-	-	-	-	-	-	cudaConfigureCall
168.86ms	1.00us	-	-	-	-	-	-	-	cudaSetupArgument
168.86ms	1.00us	-	-	-	-	-	-	-	cudaSetupArgument
168.86ms	1.00us	-	-	-	-	-	-	-	cudaSetupArgument
168.87ms	0ns	-	-	-	-	-	-	-	cudaSetupArgument
168.87ms	24.00us	-	-	-	-	-	-	-	cudaLaunch
168.89ms	761.00us	-	-	-	-	-	-	-	cudaDeviceSynchronize
168.95ms	708.51us	(64 64 1)	(8 8 1)	28	512B	0B	-	-	CUDAKernel1DCT(float*, ...)

# nvprof - Event Query

```
$ nvprof --devices 0 --query-events
```

```
===== Available Events:
```

	Name	Description
Device 0:		
Domain domain_a:		
	sm_cta_launched:	Number of thread blocks launched on a multiprocessor.
	l1_local_load_hit:	Number of cache lines that hit in L1 cache for local memory load accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
	l1_local_load_miss:	Number of cache lines that miss in L1 cache for local memory load accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
	l1_local_store_hit:	Number of cache lines that hit in L1 cache for local memory store accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.

# nvprof - Event Collection

```
$ nvprof --devices 0 --events branch,divergent_branch
```

```
===== Profiling result:
```

	Invocations	Avg	Min	Max	Event Name
Device 0					
Kernel: CUDAKernel1IDCT(float*, int, int, int)					
	1	475136	475136	475136	branch
	1	0	0	0	divergent_branch
Kernel: CUDAKernelQuantizationFloat(float*, int)					
	2	180809	180440	181178	branch
	2	6065	6024	6106	divergent_branch
Kernel: CUDAKernel1DCT(float*, int, int, int)					
	10	475136	475136	475136	branch
	10	0	0	0	divergent_branch
Kernel: CUDAKernelShortIDCT(short*, int)					
	1	186368	186368	186368	branch
	1	2048	2048	2048	divergent_branch
Kernel: CUDAKernel2IDCT(float*, float*, int)					
	1	61440	61440	61440	branch
	1	0	0	0	divergent_branch

# nvprof - Profile Data Import

- Produce profile into a file using -o

```
$ nvprof -o profile.out <app> <app args>
```

- Import into Visual Profiler

- File menu -> Import nvprof Profile...

- Import into nvprof to generate textual outputs

```
$ nvprof -i profile.out
```

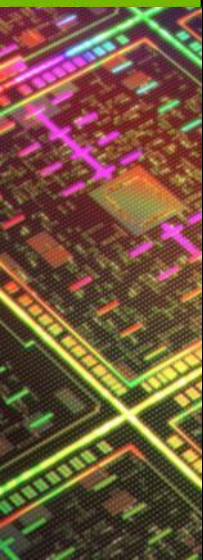
```
$ nvprof -i profile.out --print-gpu-trace
```

```
$ nvprof -i profile.out --print-api-trace
```

# Get Started

- **Download** free CUDA Toolkit: [www.nvidia.com/getcuda](http://www.nvidia.com/getcuda)
  - **Join** the community: [developer.nvidia.com/join](http://developer.nvidia.com/join)
  - **Visit** Experts Table, Developer Demo Stations
  - **Optimize** your application with CUDA Profiling Tools
- 
- S0420 - Nsight Eclipse Edition for Linux and Mac
    - Wed. 5/16, 9am, Room A5
  - S0514 - GPU Performance Analysis and Optimization
    - Wed. 5/16, 3:30pm, Hall 1





# Questions?