

An abstract 3D graphic featuring several curved, metallic-looking bands that overlap and curve around a central point, creating a sense of depth and motion. The bands have a brushed metal texture and are set against a dark, textured background.

GPU Performance Analysis and Optimisation

Thomas Bradley, NVIDIA Corporation



Outline



- **What limits performance?**
- **Analysing performance: GPU profiling**
- **Exposing sufficient parallelism**
- **Optimising for Kepler**

Additional Resources



- **More information on topics we cannot cover today**
- **Kepler architecture:**
 - GTC on Demand: Session S0642 – Inside Kepler
 - Kepler whitepapers: <http://www.nvidia.com/kepler>
- **More details:**
 - GTC on Demand
 - S0514 – GPU Performance Analysis and Optimization
 - S0419 – Optimizing Application Performance with CUDA ProfilingTools
 - S0420 – Nsight IDE for Linux and Mac
 - CUPTI documentation (describes all the profiler counters)
 - Included in every CUDA toolkit (`/cuda/extras/cupti/doc/Cupti_Users_Guide.pdf`)
- **GPU computing webinars in general:**
 - <http://developer.nvidia.com/gpu-computing-webinars>
 - In particular: register spilling

Example Workflow – Getting Started on GPU



- **Overall goal is application performance, combination of factors**
 - inter-node/inter-process communication
 - CPU-GPU communication
 - CPU/GPU performance
- **Start by analysing realistic data at realistic scale**
 - Various tools such as Vampir, TAU, Scalasca help identify hotspots
- **Extract data for development**
 - Select timesteps for shorter runtime
 - Select subdomain for smaller scale

Example Workflow – Optimising

- **Communication (node/process)**
- **Computation**
- **In this talk we're focussing on the GPU**
 - CPU, DMA, GPU overlap
 - Kernel optimisation

PERFORMANCE LIMITERS

What Limits Communication with the GPU?



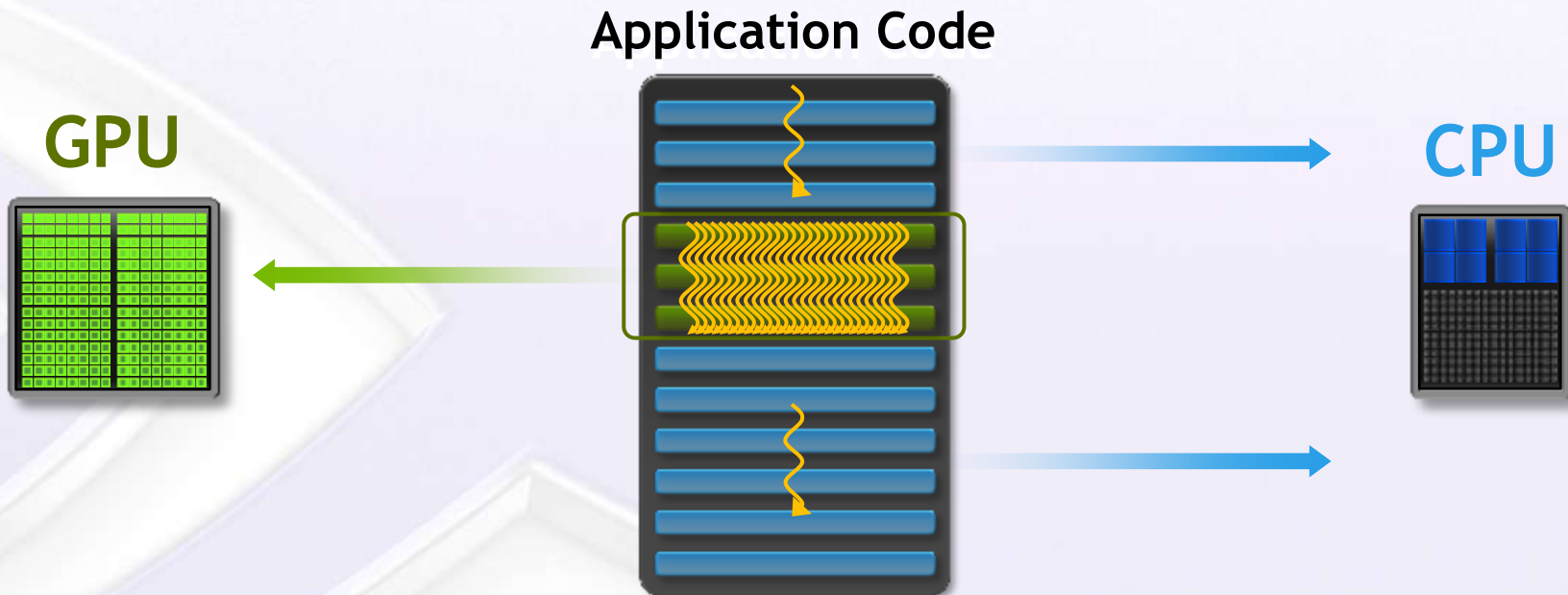
- **PCIe bus connects GPU to CPU/network**
 - Gen 2 (Fermi): 8 GB/s in each direction
 - Gen 3 (Kepler): 16 GB/s in each direction
- **Tesla GPUs have dual DMA engines**
 - Two memcpys (in different streams, different directions)
 - Overlap with kernel and CPU execution
- **GPUDirect RDMA**
 - e.g. MPI directly from GPU memory

What Limits Kernel Performance?

- **Memory bandwidth**
 - Low number of operations per byte
- **Instruction bandwidth**
 - High number of operations per byte
- **Latency**
 - Unable to fill memory or arithmetic pipelines
- **How do we determine what is limiting a given kernel?**
 - Profiling tools (e.g. nsight, nvvp, nvprof)

PROFILING

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

NVIDIA Profilers

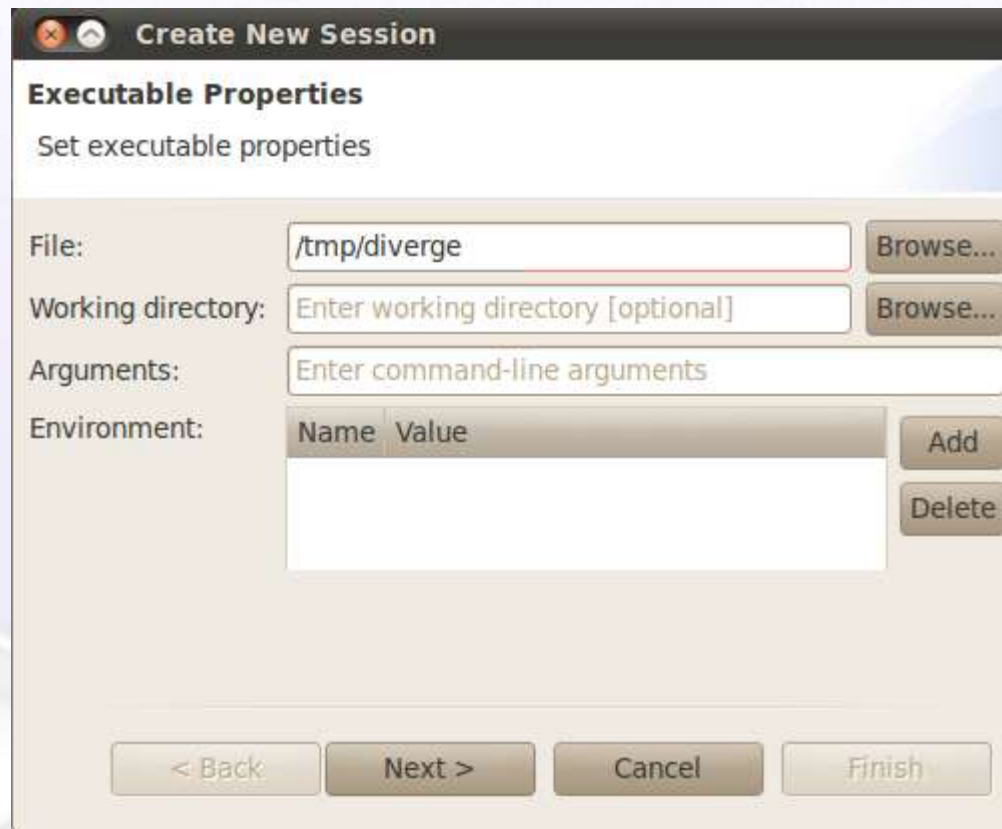
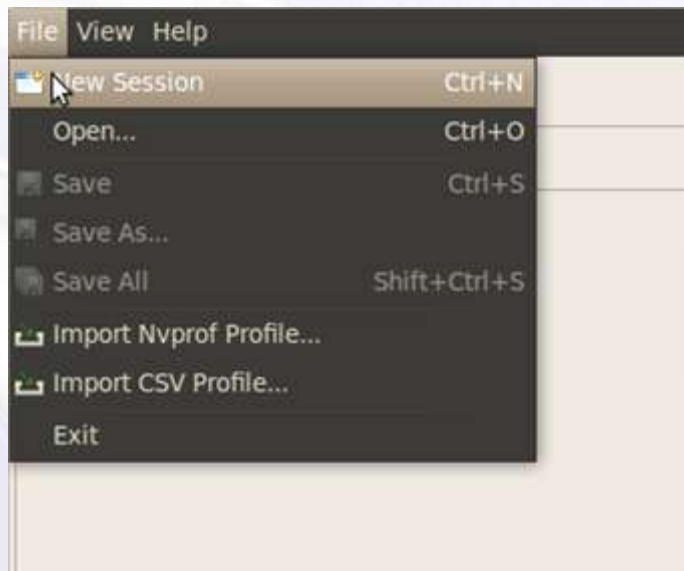


- **NVIDIA® Visual Profiler**
 - Standalone (nvvp)
 - Integrated into NVIDIA® Nsight™ Eclipse Edition (nsight)
- **NVIDIA® Nsight™ Visual Studio Edition**
- **nvprof**
 - Command-line
- **Driver-based profiler still available**
 - Command-line, controlled by environment variables

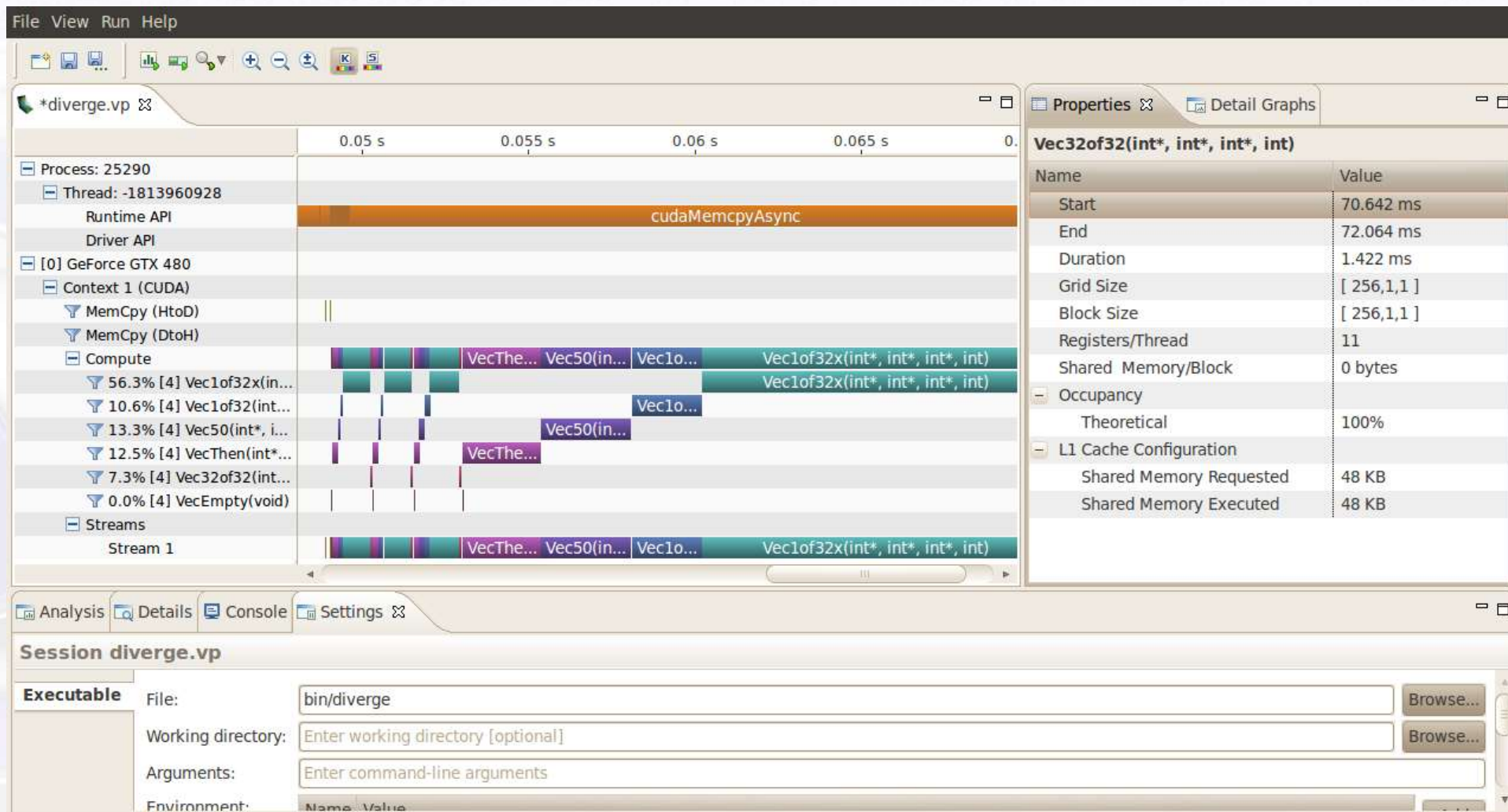


NVIDIA VISUAL PROFILER

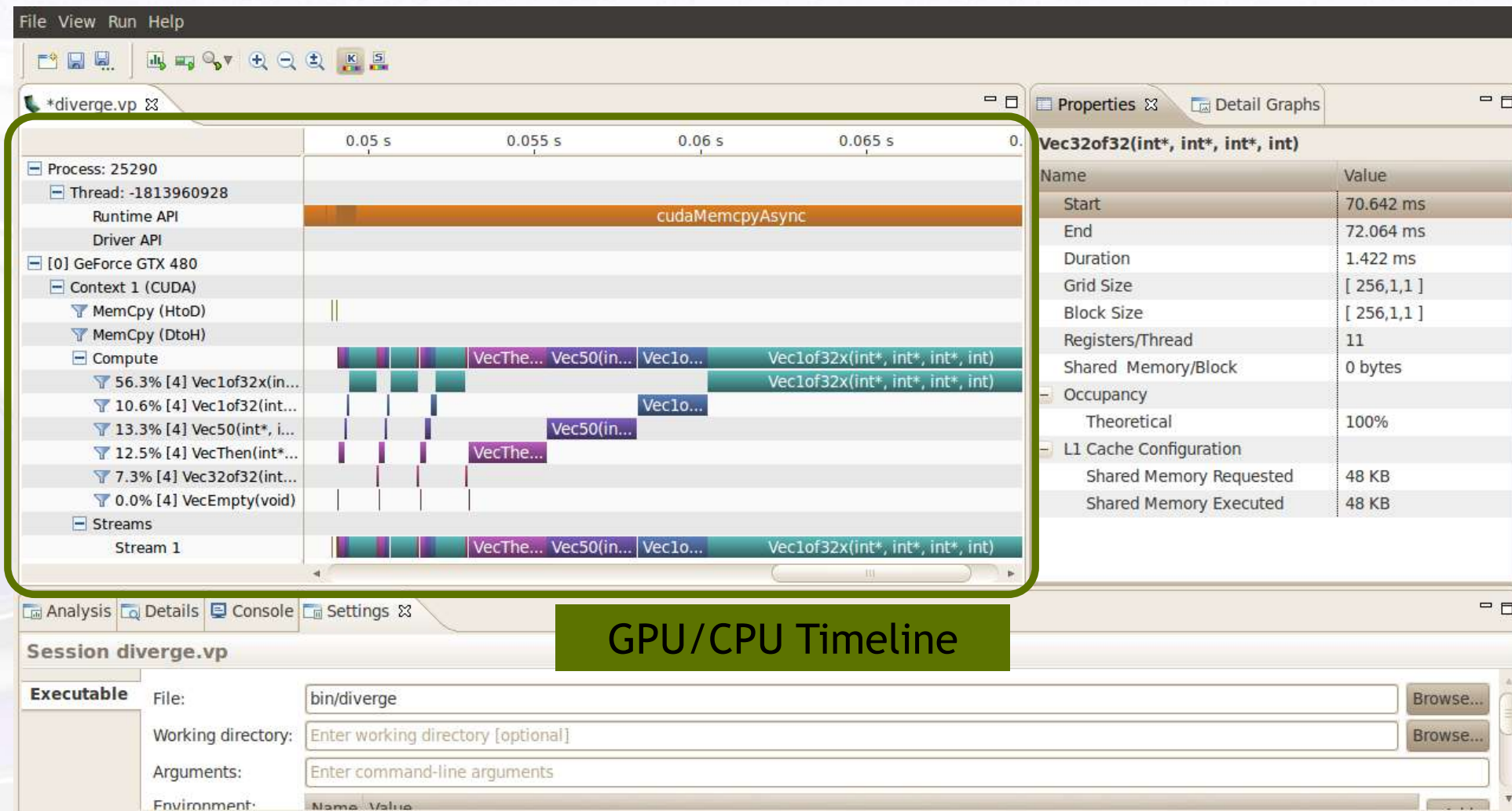
Profiling Session



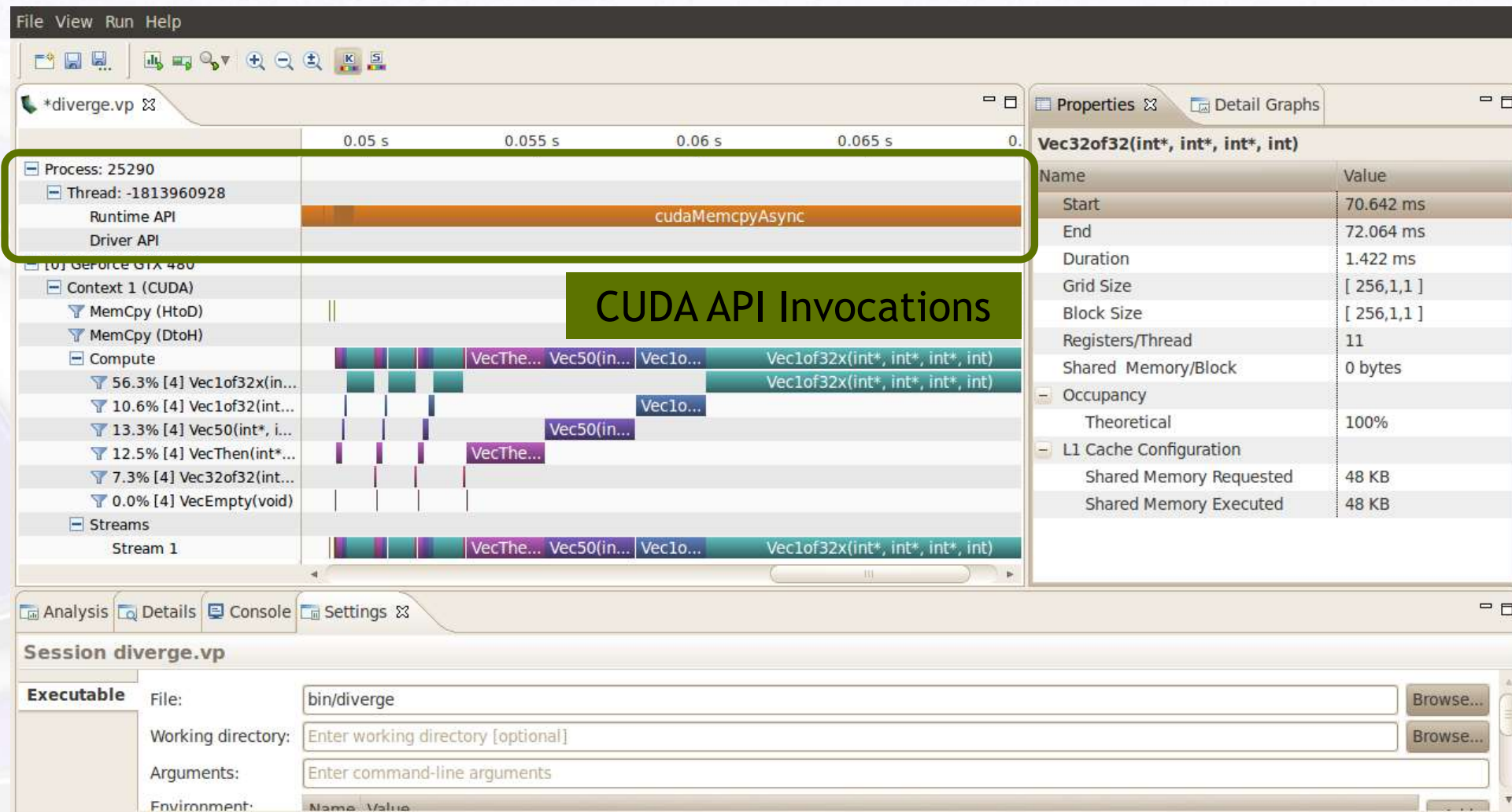
NVIDIA Visual Profiler



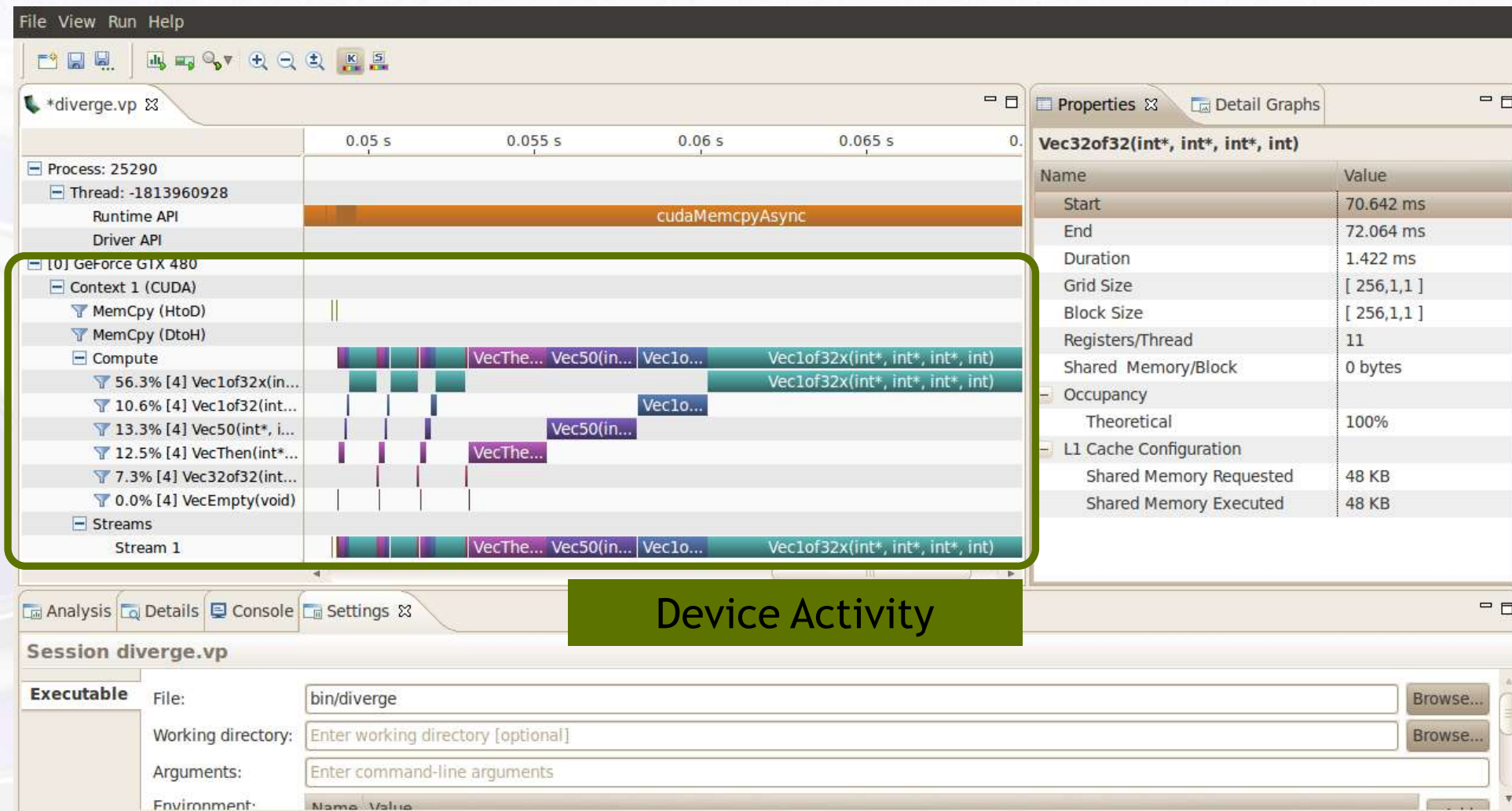
Timeline



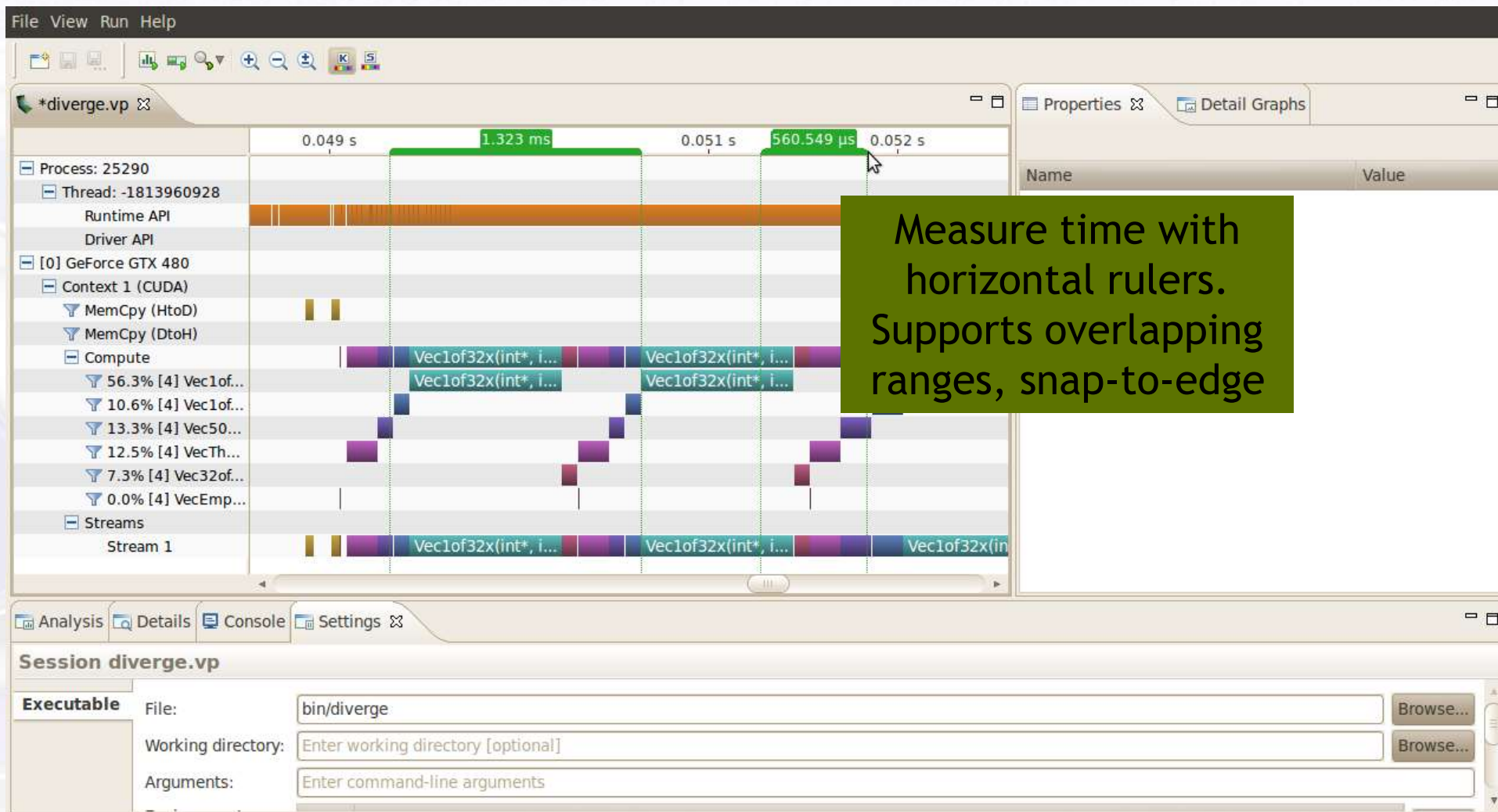
CPU Timeline



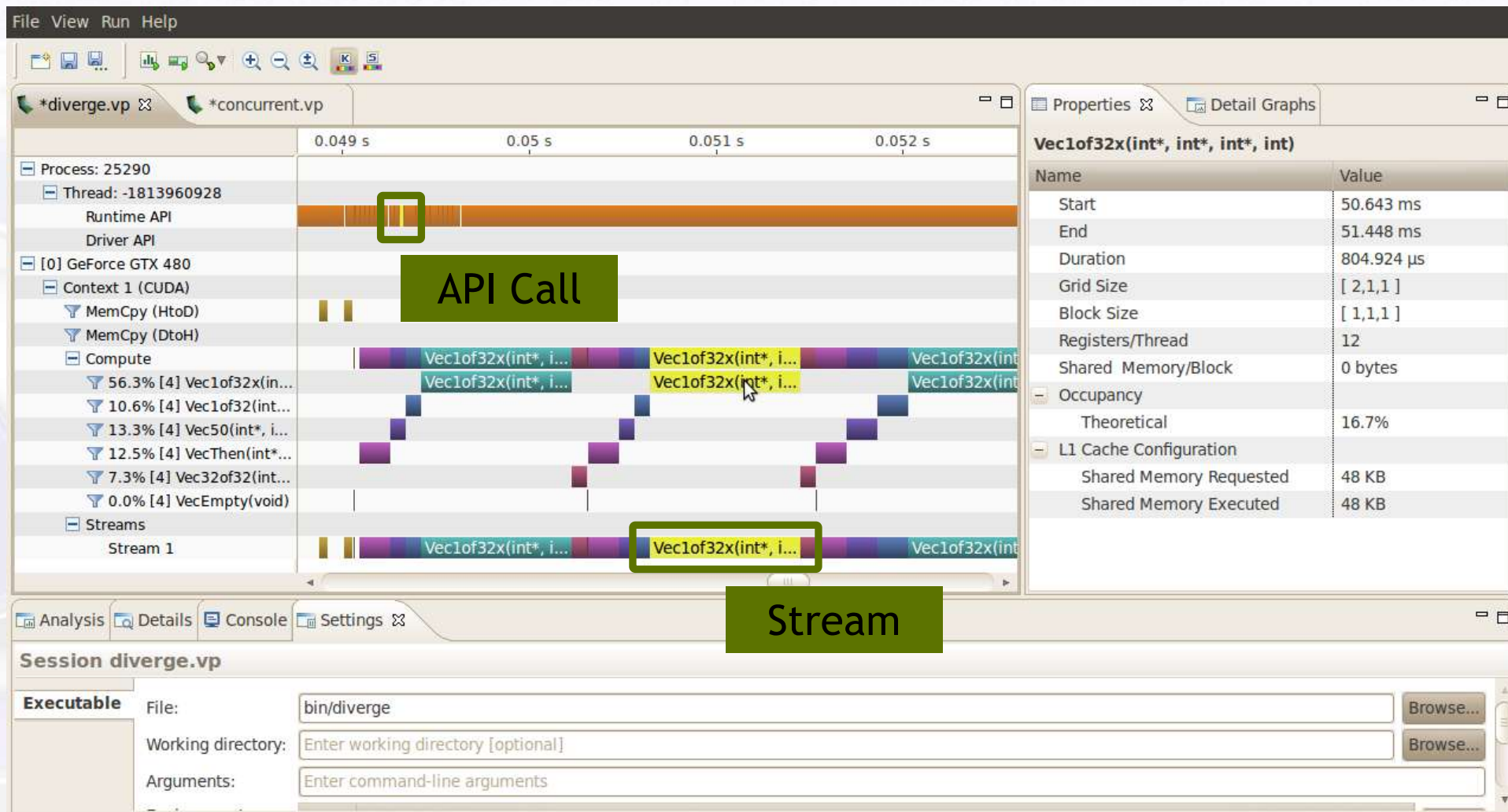
GPU Timeline



Measuring Time



Correlating CPU and GPU Activity



Properties - Kernel



The screenshot displays the NVIDIA Visual Profiler interface. The main window shows a timeline of execution for a process named '25290'. The timeline is divided into sections for 'Runtime API', 'Driver API', and 'Compute'. The 'Compute' section is expanded, showing a list of kernels and their execution times. The kernel 'Vec1of32x(int*, int*, int*)' is highlighted in yellow. A tooltip window is open, displaying the properties for this kernel.

Vec1of32x(int*, int*, int*)

Name	Value
Start	50.643 ms
End	51.448 ms
Duration	804.924 μ 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

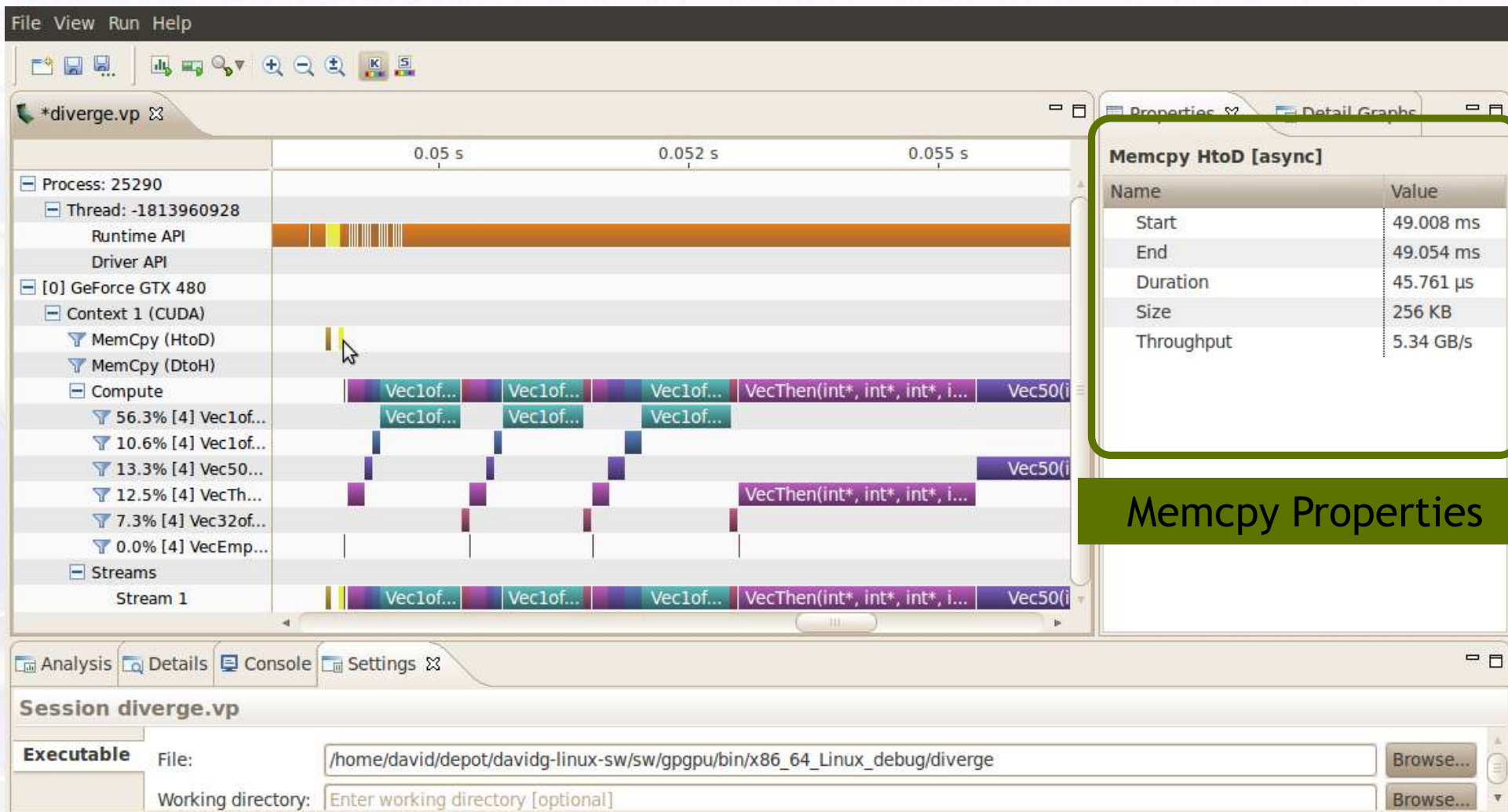
Executable

File: bin/diverge Browse...

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

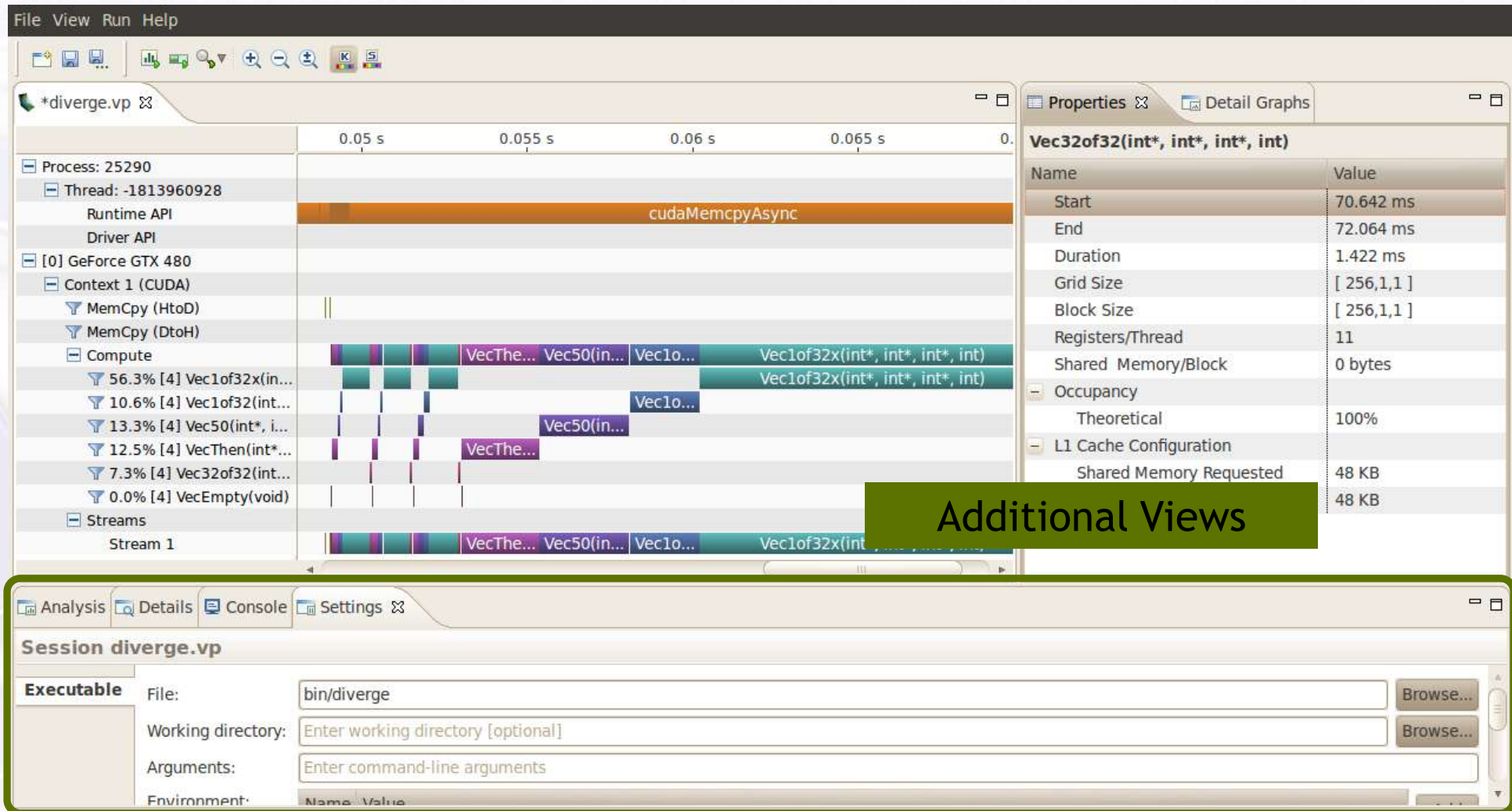
Arguments: Enter command-line arguments

Properties - Malloc

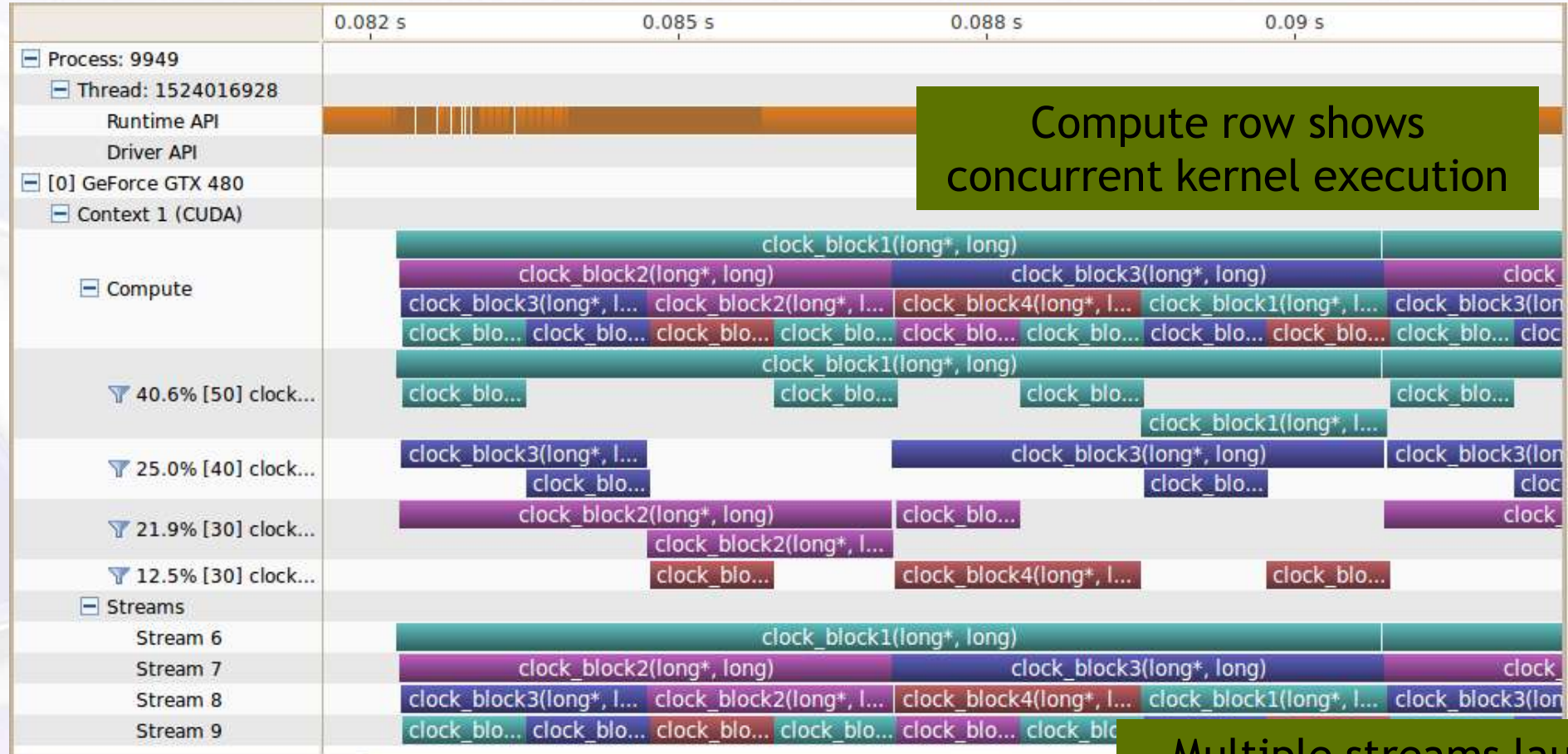


Memcpy Properties

Analysis, Details, etc.



Concurrent Kernels



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 optimisation opportunities**

Annotations: NVIDIA Tools Extension

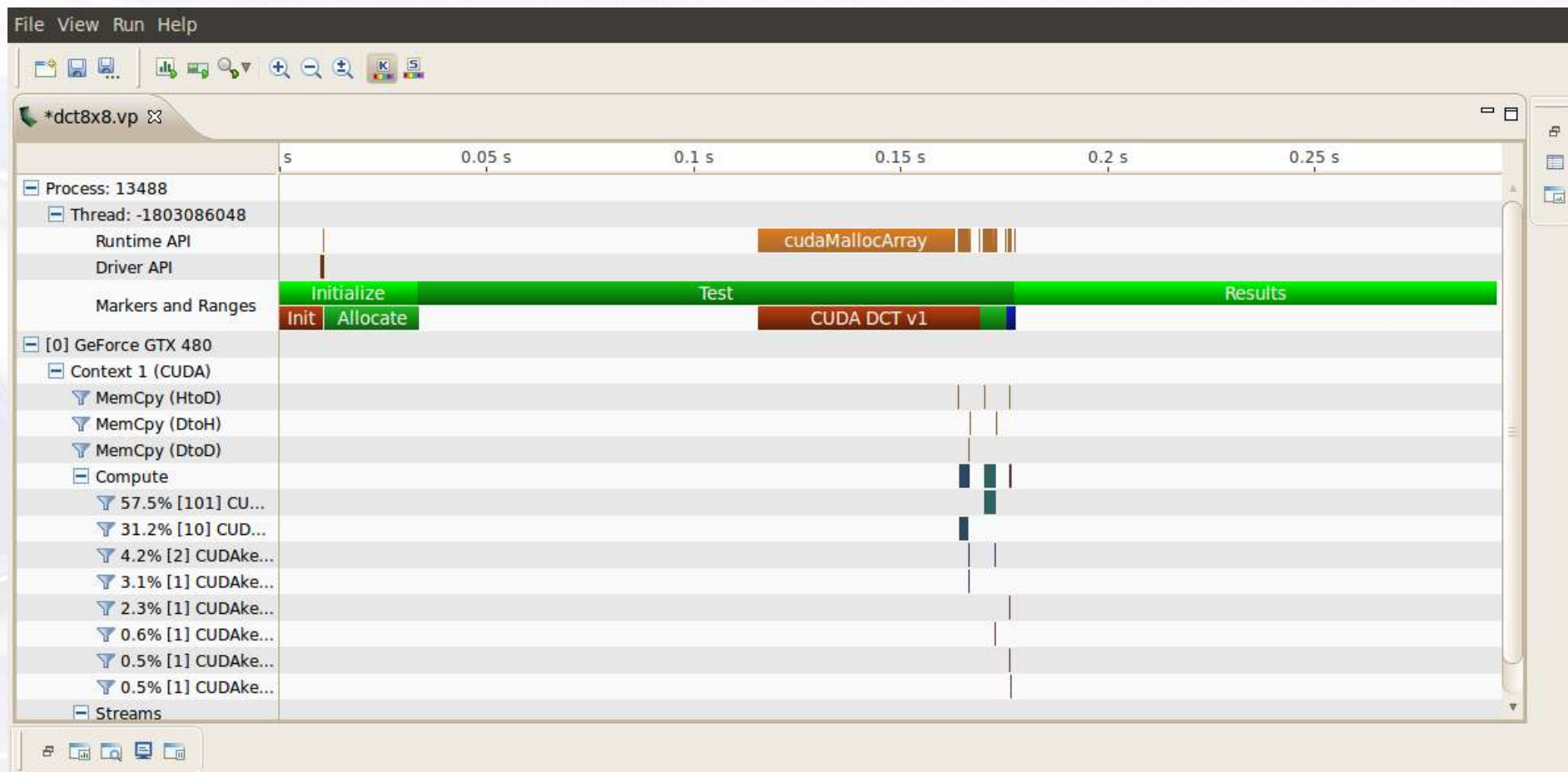
- **Developer API for CPU code**
- **Installed with CUDA Toolkit (`libnvToolsExt.so`)**
- **Naming**
 - Host OS threads: `nvtxNameOsThread()`
 - CUDA device, context, stream: `nvtxNameCudaStream()`
- **Time Ranges and Markers**
 - Range: `nvtxRangeStart()`, `nvtxRangeEnd()`
 - Instantaneous marker: `nvtxMark()`

Example: Time Ranges

- Testing algorithm in testbench
- Use time ranges API to mark initialisation, 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();  
}
```

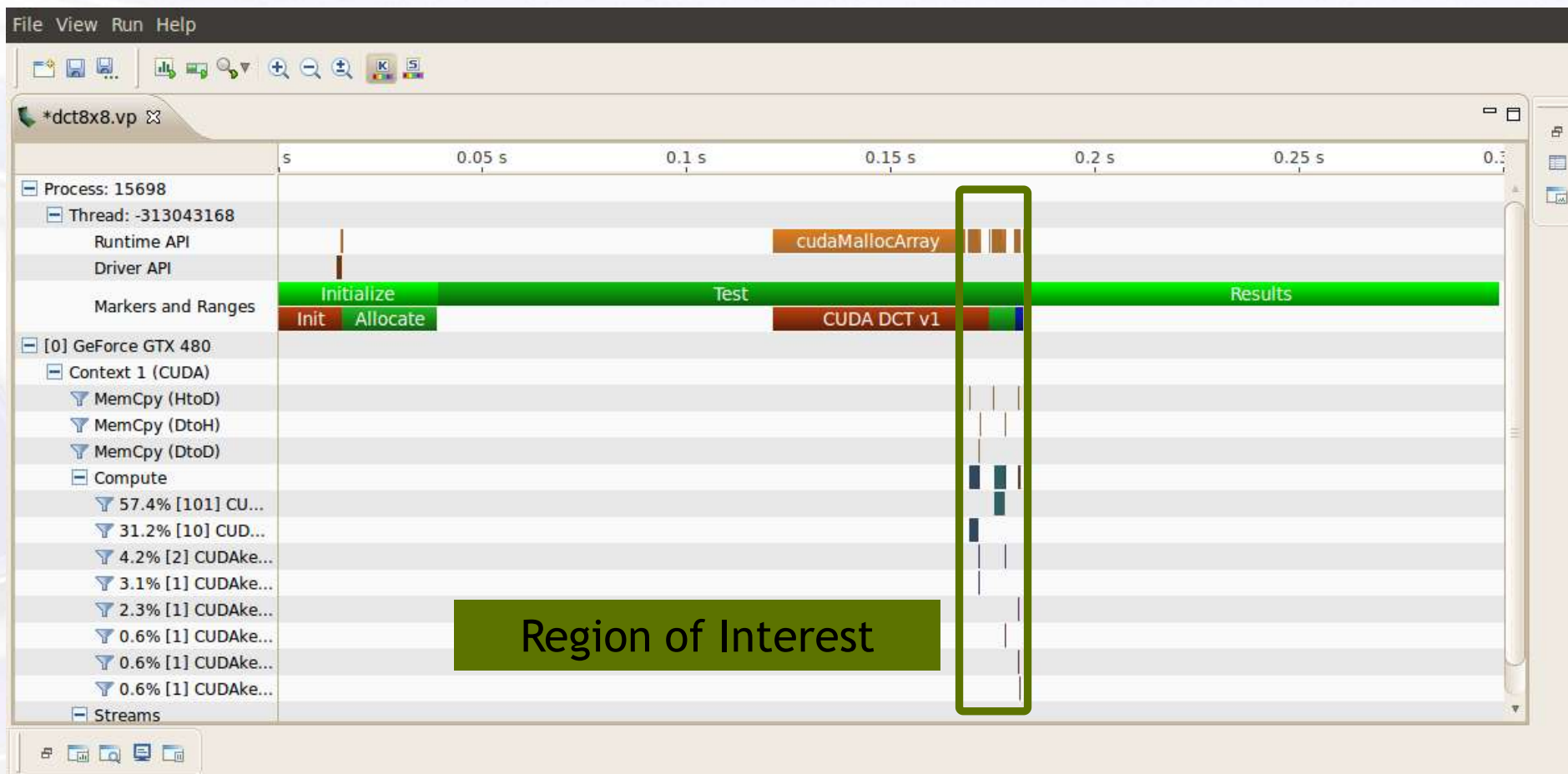

Enable Region Of Interest



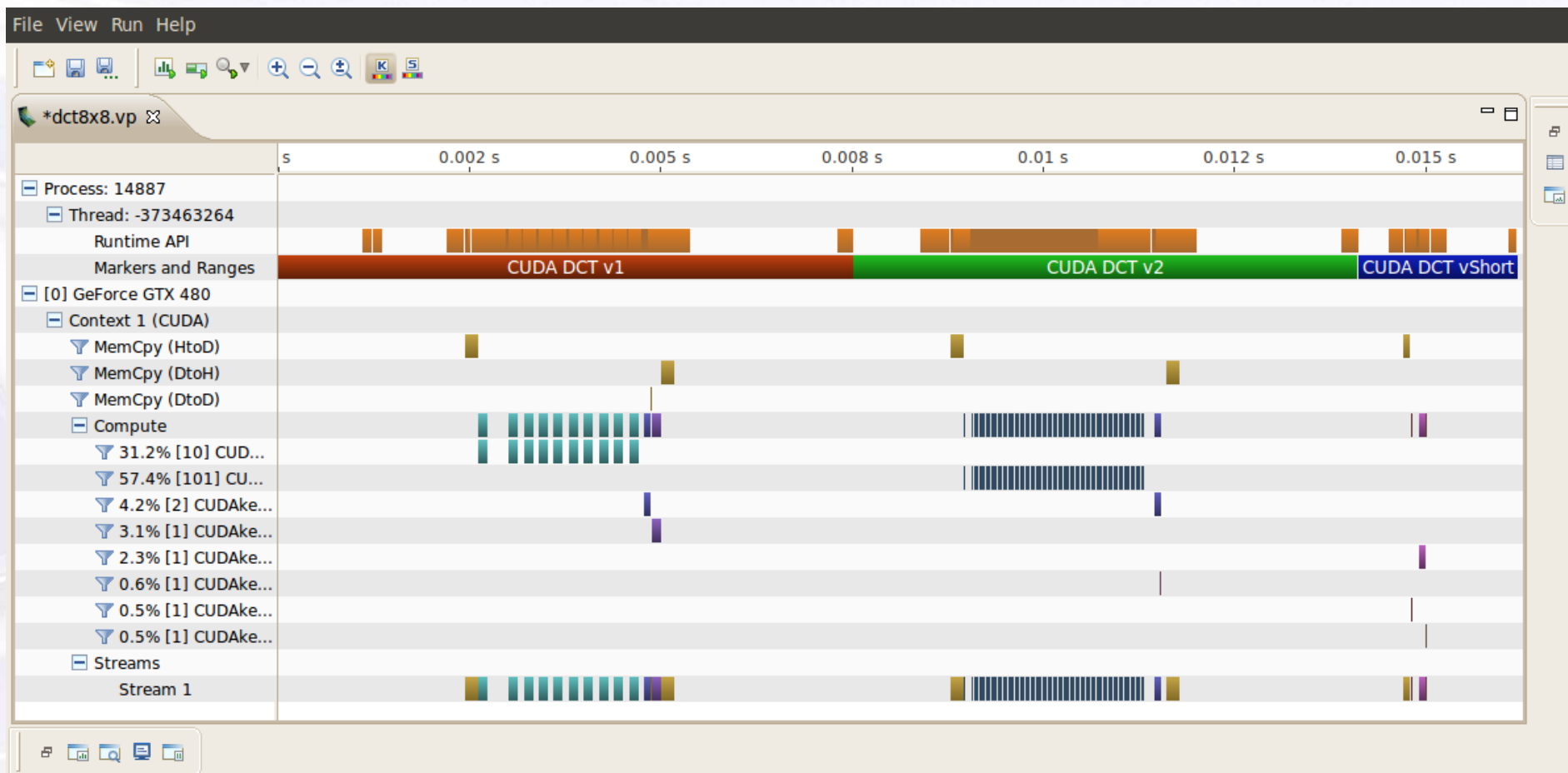
- Insert `cudaProfilerStart()` / `cudaProfilerStop()`
- Disable profiling at start of application

The screenshot shows the 'Settings' tab of the NVIDIA Visual Profiler. The session name is 'dct8x8.vp'. Under the 'Executable' section, the 'File' field contains 'bin/dct8x8' with a 'Browse...' button. The 'Working directory' field contains 'Enter working directory [optional]' with a 'Browse...' button. The 'Arguments' field contains '-noprompt'. The 'Environment' section has a table with columns 'Name' and 'Value', and buttons 'Add' and 'Delete'. The 'Execution timeout' field contains 'Enter maximum execution timeout in seconds [optional]' with a 'seconds' label. At the bottom, there are two checkboxes: 'Start execution with profiling enabled' (which is unchecked and highlighted with a green box) and 'Enable concurrent kernel profiling' (which is checked).

Example: Without cudaProfilerStart/Stop



Example: With cudaProfilerStart/Stop



Analysis



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

Visual Inspection

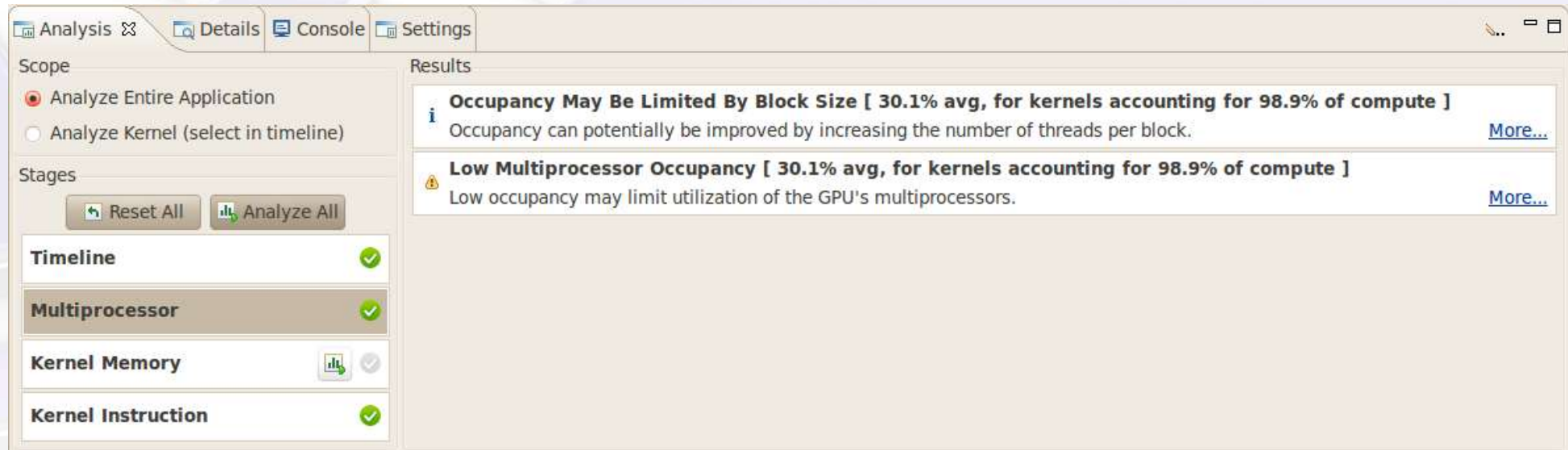
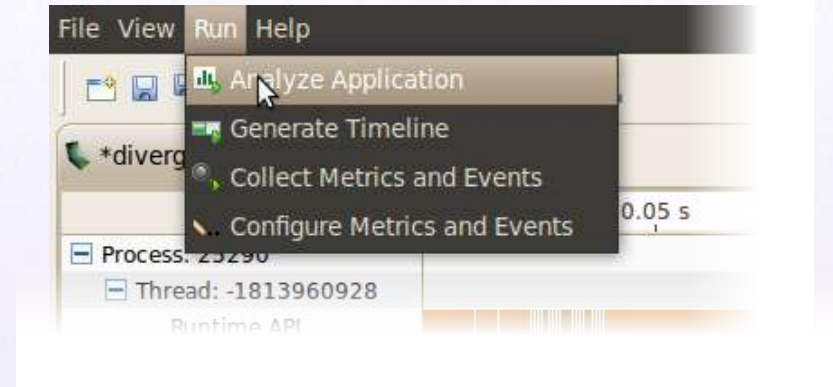


- **Understand CPU/GPU interactions**
 - Use nvToolsExt to mark time ranges on CPU
 - Is application taking advantage of both CPU and GPU?
 - Is CPU waiting on GPU? Is GPU waiting on CPU?
- **Look for potential concurrency opportunities**
 - Overlap memcpy and kernel
 - Concurrent kernels
- **Automated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

Automated Analysis - Application



- Analyse entire application
 - Timeline
 - Hardware performance counters



Analysis Documentation



Low Memcpy Throughput [997.19 MB/s avg, for memcpyys 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 (selected), Asynchronous Transfers, Zero Copy, Device Memory Spaces, 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 includes 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.' It also explains that pinned memory is allocated using `cudaMallocHost()` or `cudaHostAlloc()` and that the `bandwidthTest.cu` program in the CUDA SDK demonstrates its use. A warning states that pinned memory should not be overused as it is a scarce resource. The parent topic is 'Data Transfer Between Host and Device'. The footer includes the copyright notice 'Copyright © 2011 NVIDIA Corporation | www.nvidia.com' and the NVIDIA logo.

Search: Go [Scope: All topics](#)

Content: Visual Profiler Optimization

- Preface
- Parallel Computing with CUDA
- Performance Metrics
- Memory Optimizations
 - Data Transfer Between Host and Device
 - Pinned Memory**
 - Asynchronous Transfers
 - Zero Copy
 - Device Memory Spaces
 - Allocation
- Execution Configuration Options
- Instruction Optimizations
- Control Flow
- Recommendations and Best Practices
- NVCC Compiler Switches

Pinned Memory

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.

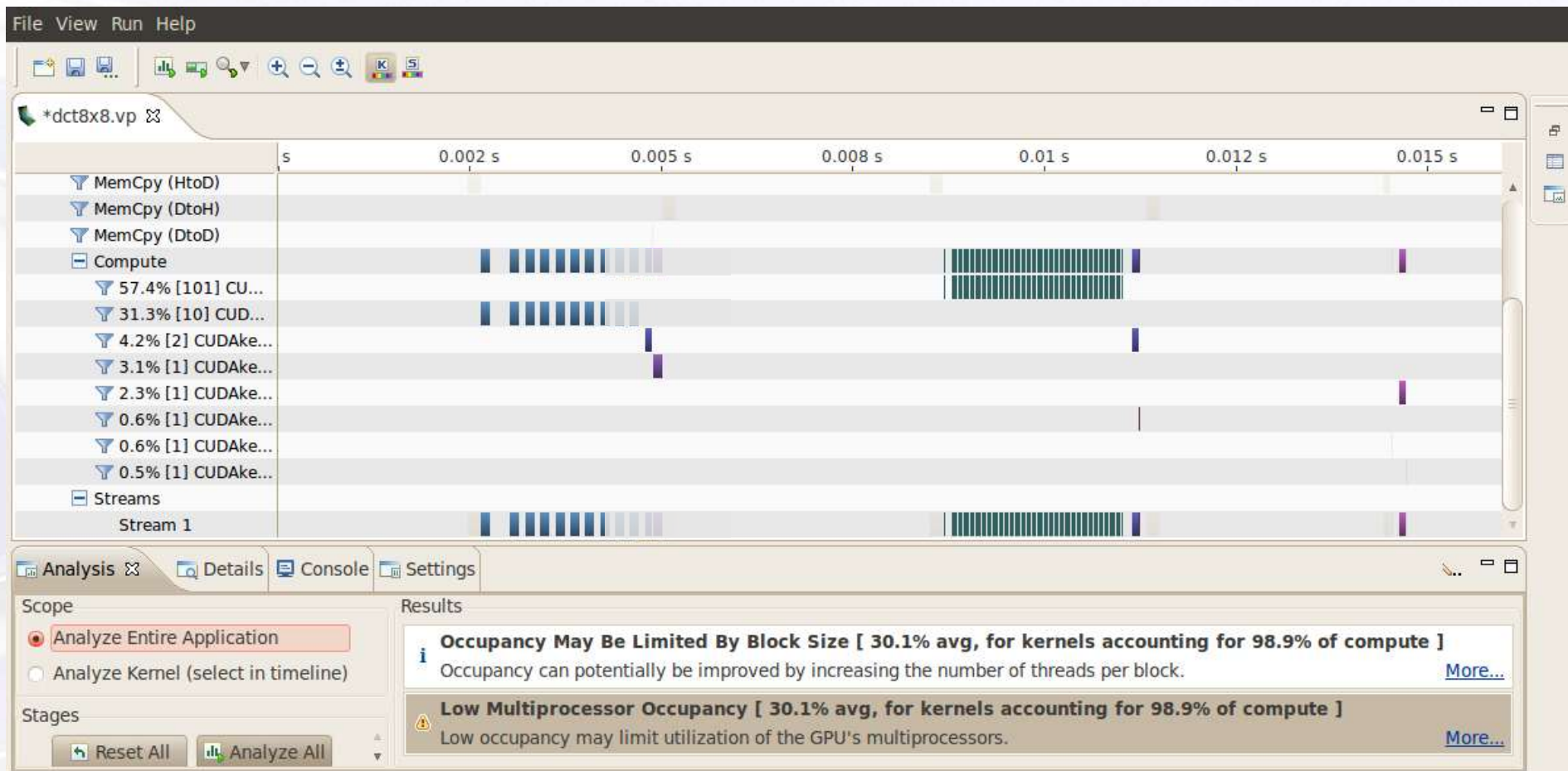
Pinned memory is allocated using the `cudaMallocHost()` or `cudaHostAlloc()` functions in the Runtime API. The `bandwidthTest.cu` program in the CUDA SDK shows how to use these functions as well as how to measure memory transfer performance.

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

Results Correlated With Timeline



Analysis Properties

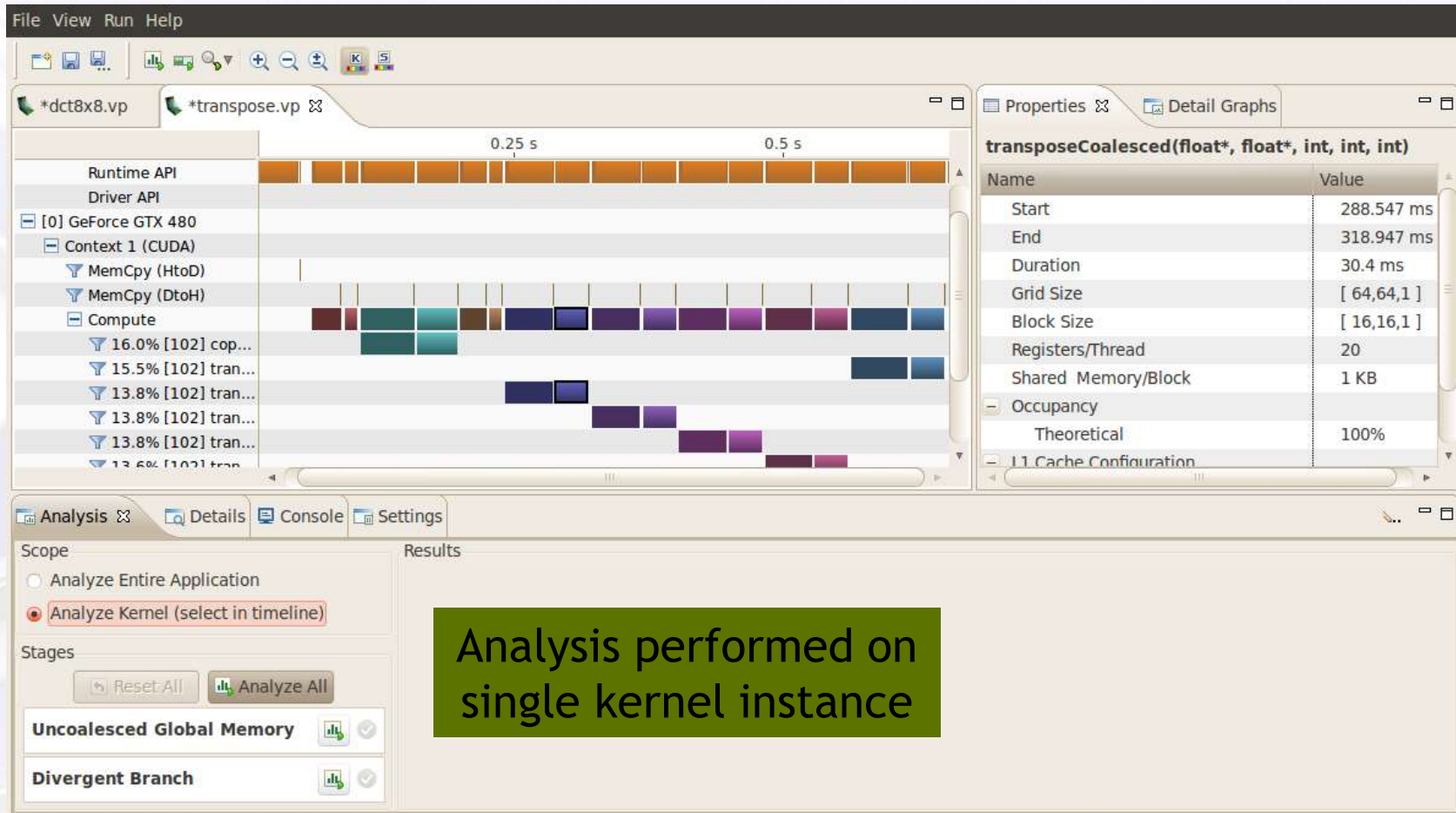


- **Highlight a kernel or memcopy in timeline**
 - **Properties shows analysis results for that specific kernel / memcopy**
 - **Optimisation opportunities are flagged**

The screenshot shows the 'Properties' window in NVIDIA Nsight Visual Studio Edition. The title bar indicates the selected item is 'CUDAkernel2DCT(float*, float*, int)'. The window is divided into two tabs: 'Properties' (active) and 'Detail Graphs'. The 'Properties' tab displays a table of analysis results for the selected kernel. The table has two columns: 'Name' and 'Value'. The data is organized into expandable sections: 'Memory' (showing Global Load and Store Efficiency at 100%), 'Instruction' (showing Branch Divergence Overhead at 0%), 'Occupancy' (showing Achieved at 29.4% with a warning icon, Theoretical at 33.3%, and Limiter as Block Size), and 'L1 Cache Configuration' (showing Shared Memory Requested and Executed at 48 KB).

Name	Value
Duration	21.117 μ s
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 displays the NVIDIA Visual Profiler interface. The left sidebar contains the 'Analysis' tab, 'Scope' (with 'Analyze Kernel' selected), 'Stages' (with 'Uncoalesced Global Memory' and 'Divergent Branch' checked), and a 'Results' pane. The 'Results' pane shows a warning icon and a message: '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.' Below this is a table 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. The top section shows the source code for a CUDA kernel named `transpose.cu`. The code is as follows:

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

for (int r=0; r < nreps; r++) {
    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];
    }
}
```

The right-hand pane shows the properties for the `transposeCoalesced(float*, float*, int, int, int)` kernel. The table below summarizes the data shown in this pane:

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%
1.1 Cache Configuration	

The bottom section of the interface shows the 'Analysis' tab with 'Scope' set to 'Analyze Kernel (select in timeline)'. Under 'Stages', 'Uncoalesced Global Memory' is selected and marked with a green checkmark. The 'Results' pane displays a warning about uncoalesced global memory accesses:

Uncoalesced Global Memory Accesses

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

Select from the table below to see the source code which generates the inefficient global loads and stores.

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 NVIDIA Nsight Visual Studio Edition interface. The left sidebar contains the 'Analysis' tab, which is active. Under 'Scope', the 'Analyze Kernel (select in timeline)' option is selected. Under 'Stages', the 'Uncoalesced Global Memory' stage is selected, and the 'Divergent Branch' stage is also selected, indicated by a green checkmark. The main 'Results' pane shows a warning icon and the title 'Divergent Branches'. The text states: 'Branches have high level of divergence, leading to significant instruction issue overhead. Select from the table below to see the source code which generates the divergent branches.' A 'More...' link is available. 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 with the following components:

- File View Help** menu bar.
- Tab Bar** showing three files: `*dct8x8.vp`, `*transpose.vp`, and `dct8x8_kernel_short.cu`.
- Source Code Editor** displaying the following C++ 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** on the right, titled **CUDAKernelShortDCT(short*, int)**, showing a table of performance metrics:

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

- Analysis Details Console Settings** at the bottom left:

- Scope:** ☐ Analyze Entire Application, ☒ Analyze Kernel (select in timeline)
- Stages:** ☒ Uncoalesced Global Memory, ☒ Divergent Branch
- Buttons: **Reset All**, **Analyze All**

- Results Panel** on the bottom right:

- Divergent Branches** section with a warning icon and text: "Branches have high level of divergence, leading to significant instruction issue overhead. Select from the table below to see the source code which generates the divergent branches." (Link: [More...](#))
- Table of divergent branches:

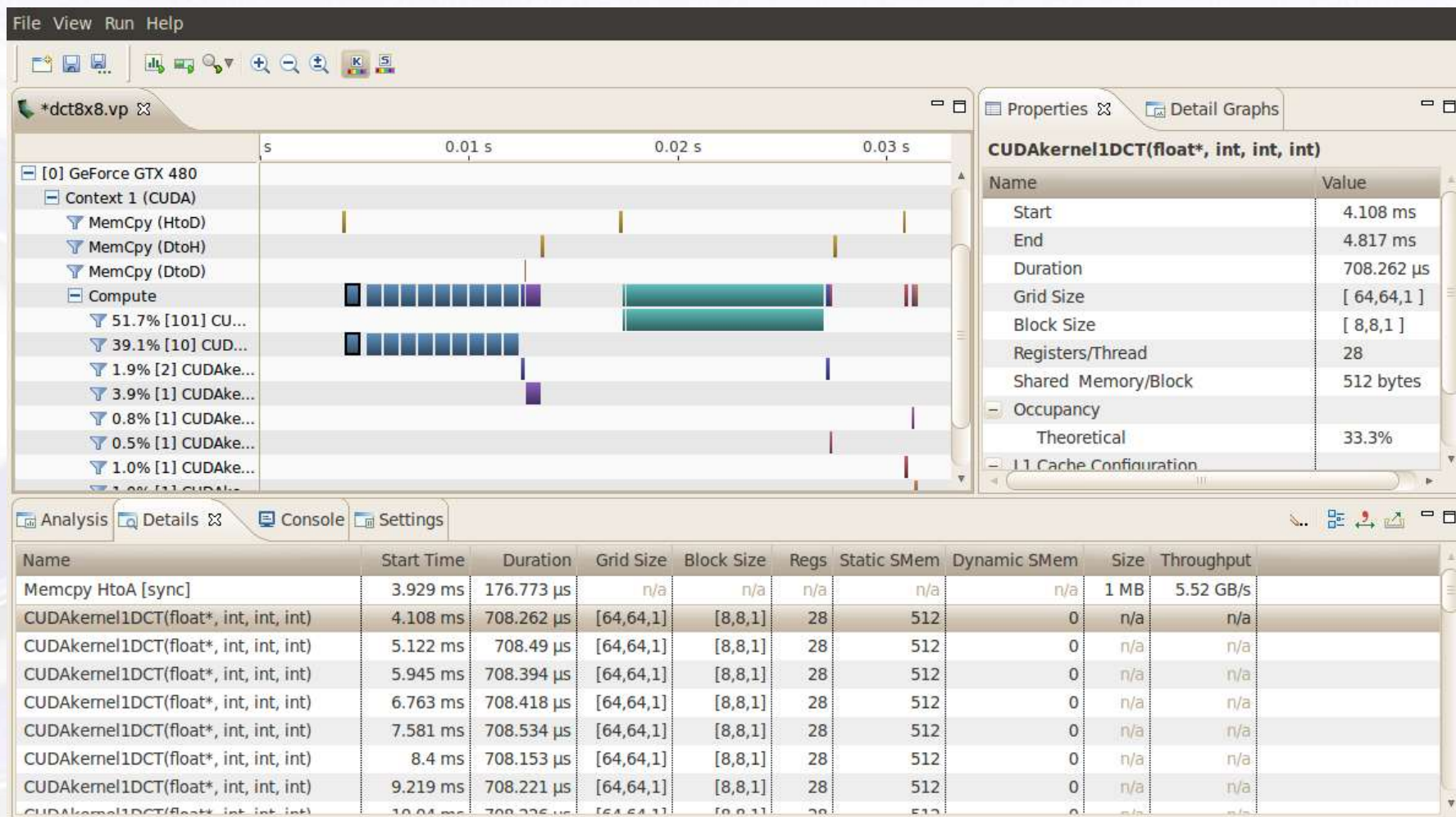
Location	Description
File: dct8x8_kernel_short	
Line: 451	Divergence = 100.0% [1024 divergent executions out of 1024 total executions]
Line: 464	Divergence = 100.0% [1024 divergent executions out of 1024 total executions]

Enabling Source Correlation

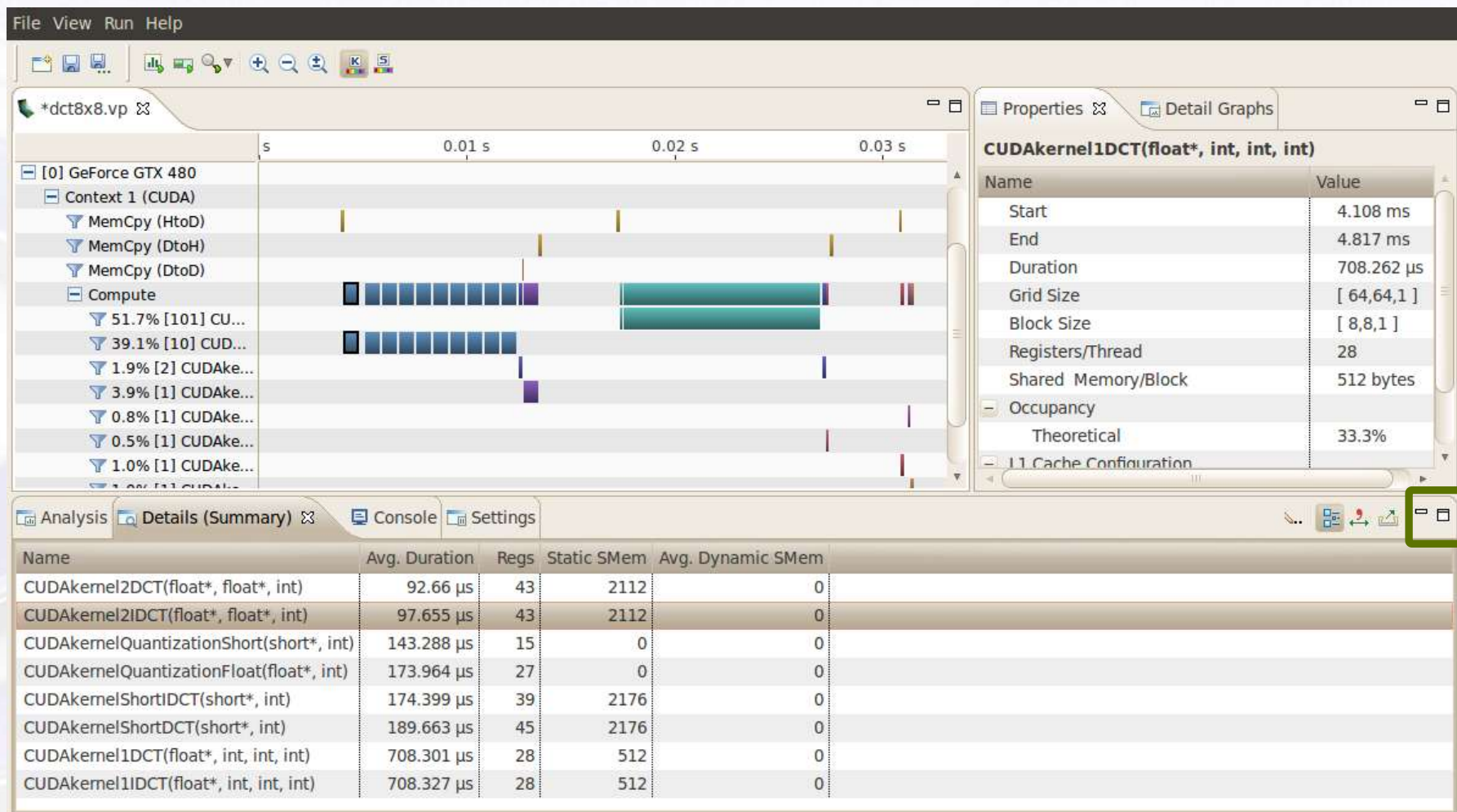


- **Source correlation requires that source/line information be embedded in executable**
 - Available in debug executables: `nvcc -G`
 - New flag for optimised executables: `nvcc -lineinfo`

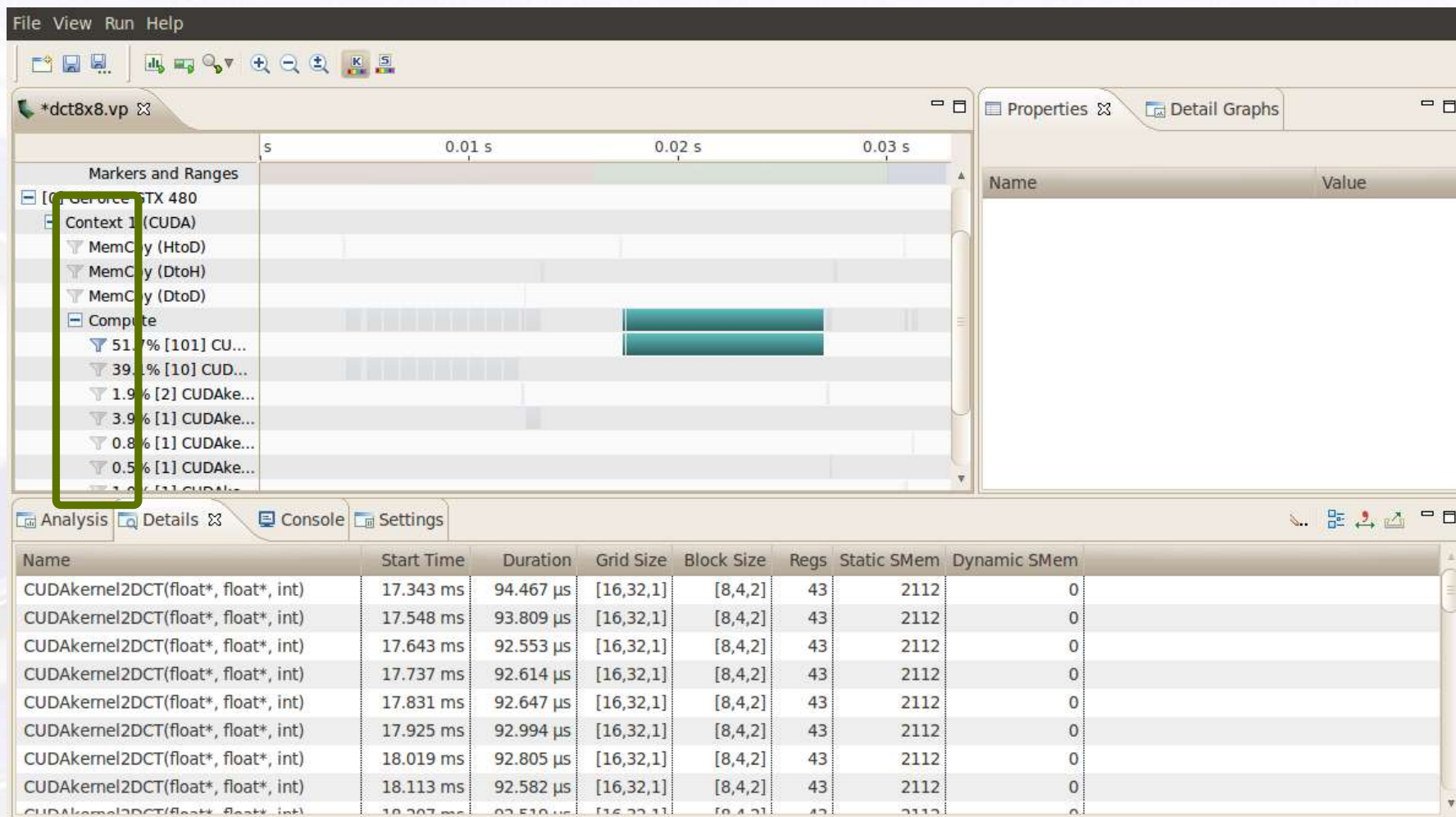
Detailed Profile Data



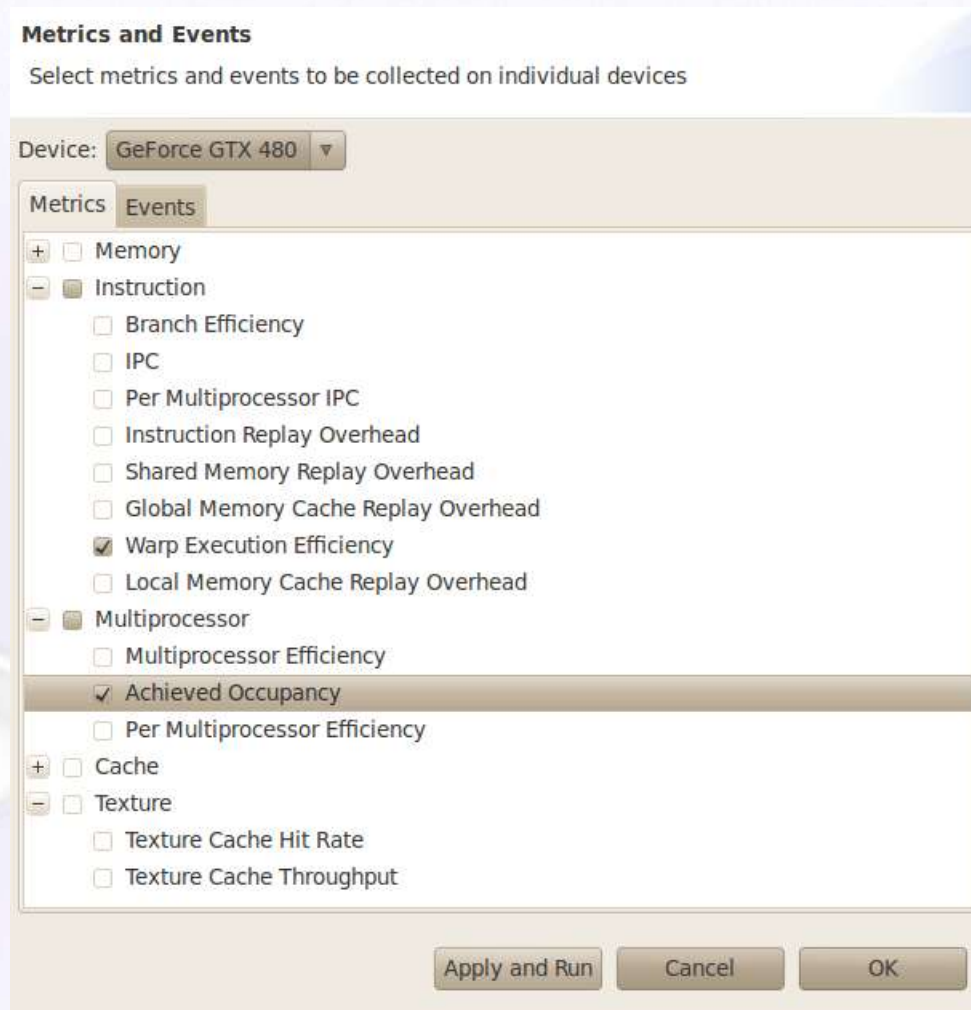
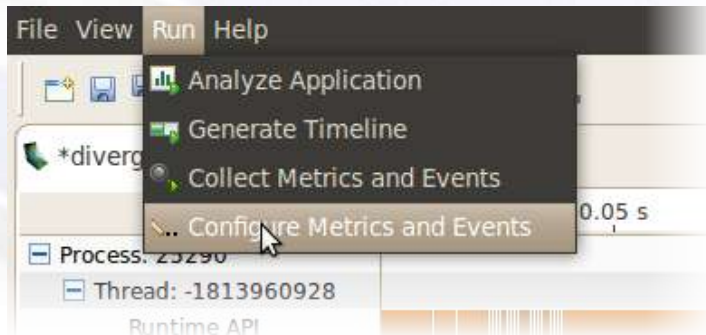
Detailed Summary Profile Data



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 μ 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 μ s	100%	0.328	[64,64,1]	[8,8,1]	28	512	0	
CUDAkernel1DCT(float*, int, int, int)	5.122 ms	708.49 μ s	100%	0.328	[64,64,1]	[8,8,1]	28	512	0	
CUDAkernel1DCT(float*, int, int, int)	5.945 ms	708.394 μ s	100%	0.327	[64,64,1]	[8,8,1]	28	512	0	
CUDAkernel1DCT(float*, int, int, int)	6.763 ms	708.418 μ s	100%	0.328	[64,64,1]	[8,8,1]	28	512	0	
CUDAkernel1DCT(float*, int, int, int)	7.581 ms	708.534 μ s	100%	0.327	[64,64,1]	[8,8,1]	28	512	0	
CUDAkernel1DCT(float*, int, int, int)	8.4 ms	708.153 μ s	100%	0.327	[64,64,1]	[8,8,1]	28	512	0	
CUDAkernel1DCT(float*, int, int, int)	9.219 ms	708.221 μ 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 μ s	43	2112	0	
CUDAkernel2IDCT(float*, float*, int)	100%	0.302	97.655 μ s	43	2112	0	
CUDAkernelQuantizationShort(short*, int)	67.5%	0.317	143.288 μ s	15	0	0	
CUDAkernelQuantizationFloat(float*, int)	98.7%	0.318	173.964 μ s	27	0	0	
CUDAkernelShort1DCT(short*, int)	74.7%	0.468	174.399 μ s	39	2176	0	
CUDAkernelShortDCT(short*, int)	75%	0.376	189.663 μ s	45	2176	0	
CUDAkernel1DCT(float*, int, int, int)	100%	0.328	708.301 μ s	28	512	0	
CUDAkernel1IDCT(float*, int, int, int)	100%	0.328	708.327 μ 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
 - Visualise 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	CUDAKernel12DCT(float*, float*, int)
37.47	7.08ms	10	708.31us	707.99us	708.50us	CUDAKernel11DCT(float*,int, int,int)
3.75	708.42us	1	708.42us	708.42us	708.42us	CUDAKernel11IDCT(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	CUDAKernel12IDCT(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 dct8x8
```

=====
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: CUDAKernel1IDCT(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 Export/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
```

nvprof – MPI



- **Each rank must output to separate file**
- **Launch nvprof wrapper with mpirun**
 - **Set output file name based on rank**
 - **Limit which ranks are profiled**
 - **Example script in nvvp help for OpenMPI and MVAPICH2**
 - **Remember to disable profiling at start if using `cudaProfilerStart()/cudaProfilerStop()`**

EXPOSING SUFFICIENT PARALLELISM

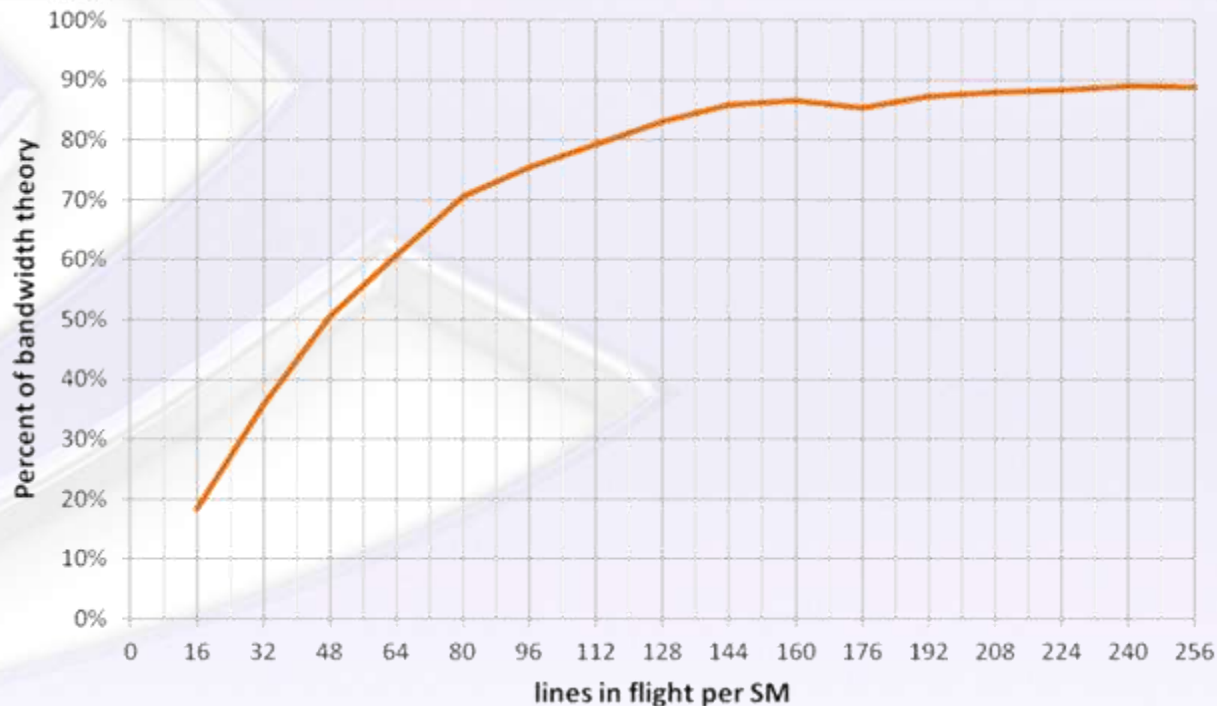
Kepler: Level of Parallelism Needed

- **To saturate instruction bandwidth:**
 - Fp32 math: **~1.7K** independent instructions per SM
 - Lower for other, lower-throughput instructions
 - Keep in mind that Kepler SM can track up to 2048 threads
- **To saturate memory bandwidth:**
 - **100+** independent lines per SM

Memory Parallelism



- Achieved Kepler memory throughput
 - As a function of the number of independent requests per SM
 - Request: 128-byte line



Exposing Sufficient Parallelism



- **What hardware ultimately needs:**
 - **Arithmetic pipes:** Sufficient number of independent instructions (accommodate multi-issue and latency hiding)
 - **Memory system:** Sufficient requests in flight to saturate bandwidth (Little's Law)
- **Two ways to increase parallelism**
 - **More independent work within a thread (warp)**
 - ILP for math, independent accesses for memory
 - **More concurrent threads (warps)**

Occupancy



- **Occupancy: number of concurrent threads per SM**
 - Expressed as either:
 - the number of threads (or warps)
 - percentage of maximum threads
- **Determined by several factors**
 - (refer to Occupancy Calculator, CUDA Programming Guide for full details)
 - **Registers per thread**
 - SM registers are partitioned among the threads
 - **Shared memory per threadblock**
 - SM shared memory is partitioned among the blocks
 - **Threads per threadblock**
 - Threads are allocated at threadblock granularity

Kepler SM resources

- 64K 32-bit registers
- Up to 48 KB of shared memory
- Up to 2048 concurrent threads
- Up to 16 concurrent threadblocks

Occupancy and Performance



- **Note that 100% occupancy isn't needed to reach maximum performance**
 - Sufficient occupancy to hide latency, higher occupancy will not improve performance
- **“Sufficient” occupancy depends on the code**
 - More independent work per thread → less occupancy is needed
 - Memory-bound codes tend to need higher occupancy
 - Higher latency (than for arithmetic) needs more work

Exposing Parallelism: Grid Configuration

- **Grid: arrangement of threads into threadblocks**
- **Two goals:**
 - Expose enough parallelism to an SM
 - Balance work across the SMs
- **Several things to consider when launching kernels:**
 - Number of threads per threadblock
 - Number of threadblocks
 - Amount of work per threadblock

Threadblock Size and Occupancy

- **Threadblock size is a multiple of warp size (32)**
 - Even if you request fewer threads, HW rounds up
- **Threadblocks can be too small**
 - Kepler SM can run up to 16 threadblocks concurrently
 - SM may reach the block limit before reaching good occupancy
 - Example: 1-warp blocks -> 16 warps per Kepler SM (frequently not enough)
- **Threadblocks can be too big**
 - Quantization effect:
 - Enough SM resources for more threads, not enough for another large block
 - A threadblock isn't started until resources are available for all of its threads

Threadblock Sizing



**Too few
threads per
block**



- **SM resources:**
 - Registers
 - Shared memory

**Too many
threads per
block**



Case Study: Threadblock Sizing



- **Non-hydrostatic Icosahedral Model (NIM)**
 - Global weather simulation code, NOAA
 - vdmintv kernel:
 - 63 registers per thread, 3840 bytes of SMEM per warp
 - At most **12** warps per Fermi SM (limited by SMEM)
- **Initial grid: 32 threads per block, 10,424 blocks**
 - Blocks are too small:
 - **8** warps per SM, limited by number of blocks (Fermi's limit was 8)
 - Code achieves a small percentage (~30%) of both math and memory bandwidth
 - Time: 6.89 ms

Case Study: Threadblock Sizing

- **Optimized config: 64 threads per block, 5,212 blocks**
 - Occupancy: **12** warps per SM, limited by SMEM
 - Time: 5.68 ms (1.21x speedup)
- **Further optimization:**
 - Reduce SMEM consumption by moving variables to registers
 - 63 registers per thread, 1536 bytes of SMEM per warp
 - Occupancy: **16** warps per SM, limited by registers
 - Time: 3.23 ms (2.13x speedup over original)

General Guidelines



- **Threadblock size choice:**
 - **Start with 128-256 threads per block**
 - Adjust up/down by what best matches your function
 - Example: stencil codes prefer larger blocks to minimize halos
 - **Multiple of warp size (32 threads)**
 - **If occupancy is critical to performance:**
 - Check that block size isn't precluding occupancy allowed by register and SMEM resources
- **Grid size:**
 - **1,000 or more threadblocks**
 - 10s of waves of threadblocks: no need to think about tail effect
 - Makes your code ready for several generations of future GPUs

OPTIMISING FOR KEPLER

Kepler Architecture Family



- **Two architectures in the family:**
 - **GK104 (Tesla K10, GeForce: GTX690, GTX680, GTX670, ...)**
 - Note that K10 is 2 GK104 chips on a single board
 - **GK110 (Tesla K20, ...)**
- **GK110 has a number of features not in GK104:**
 - **Dynamic parallelism, HyperQ**
 - **More registers per thread, more fp64 throughput**
 - **For full details refer to:**
 - **Kepler Whitepaper (<http://www.nvidia.com/kepler>)**
 - **GTC12 Session 0642: “Inside Kepler”**

Good News About Kepler Optimisation



- **The same optimisation fundamentals that applied to Fermi, apply to Kepler**
 - There are no new fundamentals
- **Main optimization considerations:**
 - **Expose sufficient parallelism**
 - SM is more powerful, so will need more work
 - **Coalesce memory access**
 - Exactly the same as on Fermi
 - **Have coherent control flow within warps**
 - Exactly the same as on Fermi

Level of Parallelism



- **Parallelism for memory is most important**
 - **Most codes don't achieve peak fp throughput because:**
 - Stalls waiting on memory (latency not completely hidden)
 - Execution of non-fp instructions (indexing, control-flow, etc.)
 - NOT because of lack of independent fp math
- **GK104:**
 - **Compared to Fermi, needs ~2x concurrent accesses per SM to saturate memory bandwidth**
 - Memory bandwidth comparable to Fermi
 - 8 SMs while Fermi had 16 SMs
 - **Doesn't necessarily need twice the occupancy of your Fermi code**
 - If Fermi code exposed more than sufficient parallelism, increase is less than 2x

Kepler SM Improvements for Occupancy



- **2x registers**
 - Both GK104 and GK110
 - **64K** registers (Fermi had 32K)
 - Code where occupancy is limited by registers will readily achieve higher occupancy (run more concurrent warps)
- **2x threadblocks**
 - Both GK104 and GK110
 - Up to **16** threadblocks (Fermi had 8)
- **1.33x more threads**
 - Both GK104 and GK110
 - Up to **2048** threads (Fermi had 1536)

Increased Shared Memory Bandwidth

- Both GK104 and GK110
- To benefit, code must access 8-byte words
 - No changes for double-precision codes
 - Single-precision or integer codes should group accesses into **float2**, **int2** structures to get the benefit
- Refer to Case Study 6 for a usecase sample

SM Improvements Specific to GK110



- **More registers per thread**
 - A thread can use up to **255** registers (Fermi had **63**)
 - Improves performance for some codes that spilled a lot of registers on Fermi (or GK104)
 - Note that more registers per thread still has to be weighed against lower occupancy
- **Ability to use read-only cache for accessing global memory**
 - Improves performance for some codes with scattered access patterns, lowers the overhead due to replays
- **Warp-shuffle instruction (tool for ninjas)**
 - Enables threads in the same warp to exchange values without going through shared memory

Considerations for Dynamic Parallelism



- **GPU threads are able to launch work for GPU**
 - GK110-specific feature
- **Same considerations as for launches from CPU**
 - Same exact considerations for exposing sufficient parallelism as for “traditional” launches (CPU launches work for GPU)
 - A single launch doesn’t have to saturate the GPU:
 - GPU can execute up to 32 different kernel launches concurrently

Conclusion



- **When programming and optimising think about:**
 - Exposing sufficient parallelism
 - Coalescing memory accesses
 - Having coherent control flow within warps
- **Use profiling tools when analyzing performance**
 - Determine performance limiters first
 - Diagnose memory access patterns