GPU Performance Analysis and Optimisation

Thomas Bradley, NVIDIA Corporation

NVIDIA

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

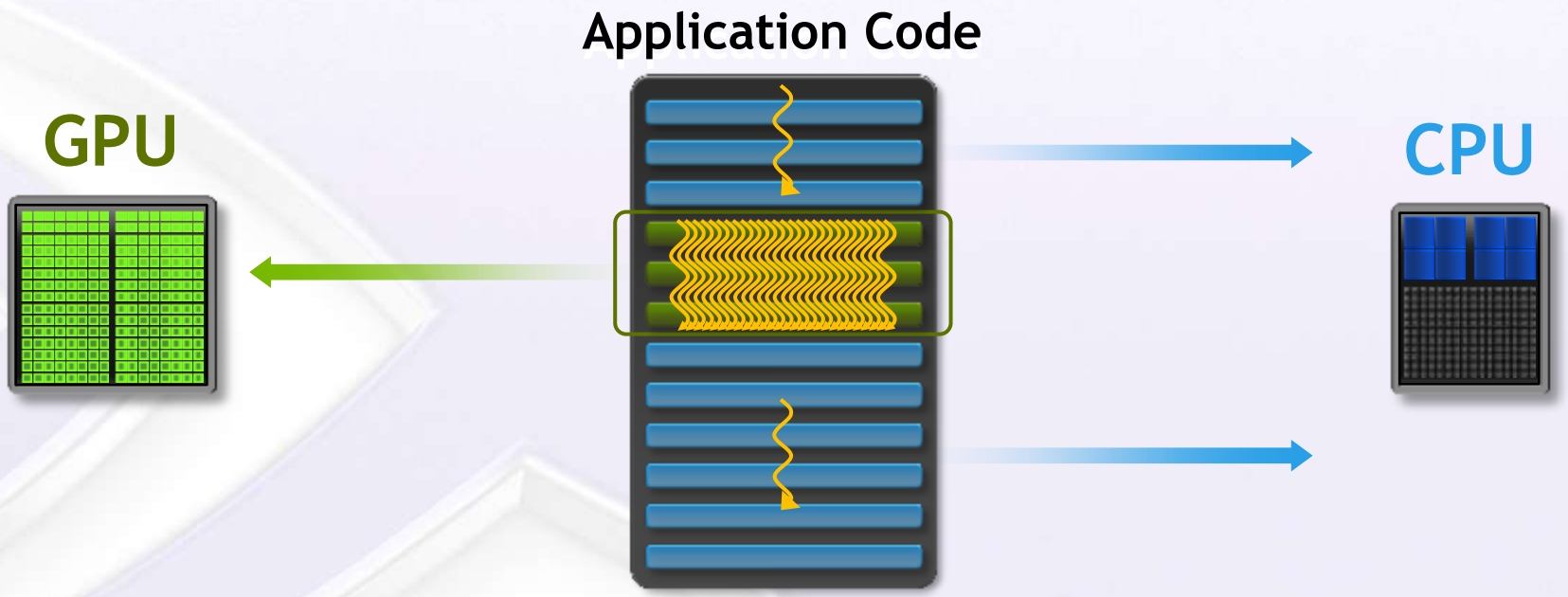
What Limits Kernel Performance?

- 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

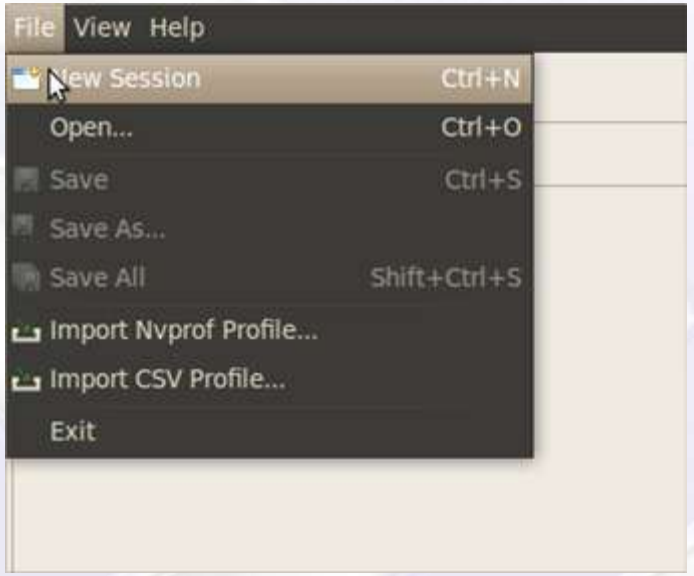
- A few cores
- 10's of threads
- Good memory bandwidth
- Best at serial execution

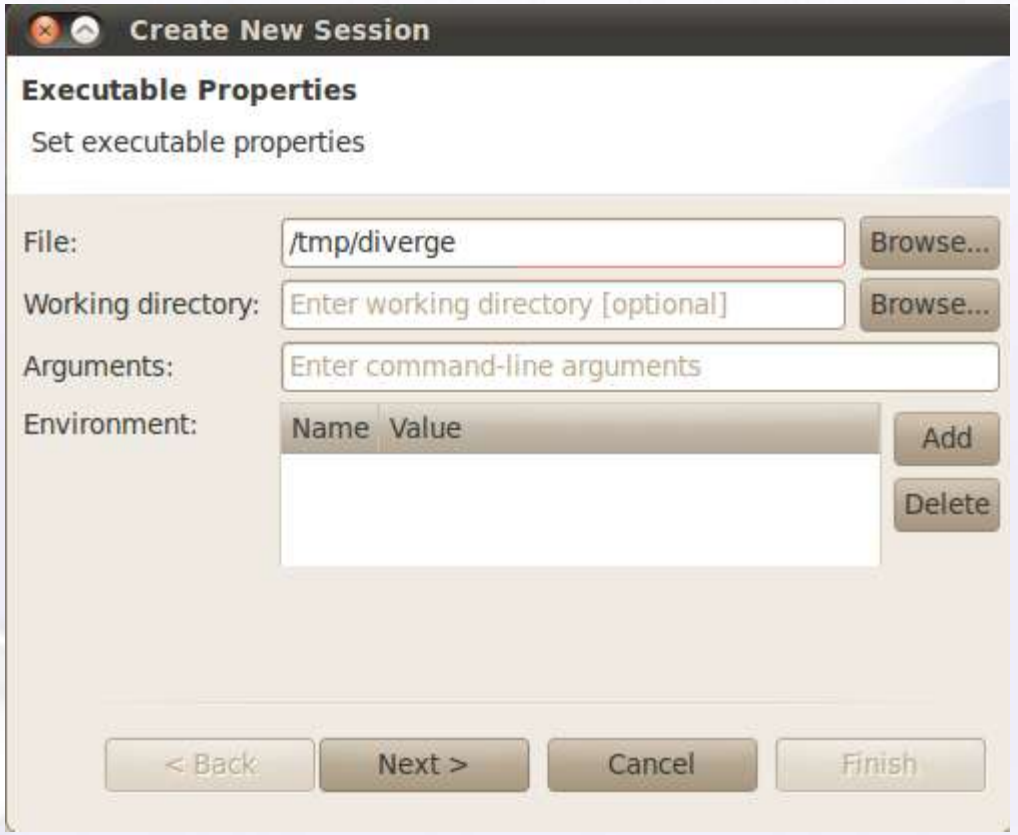
NVIDIA Profilers

- 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

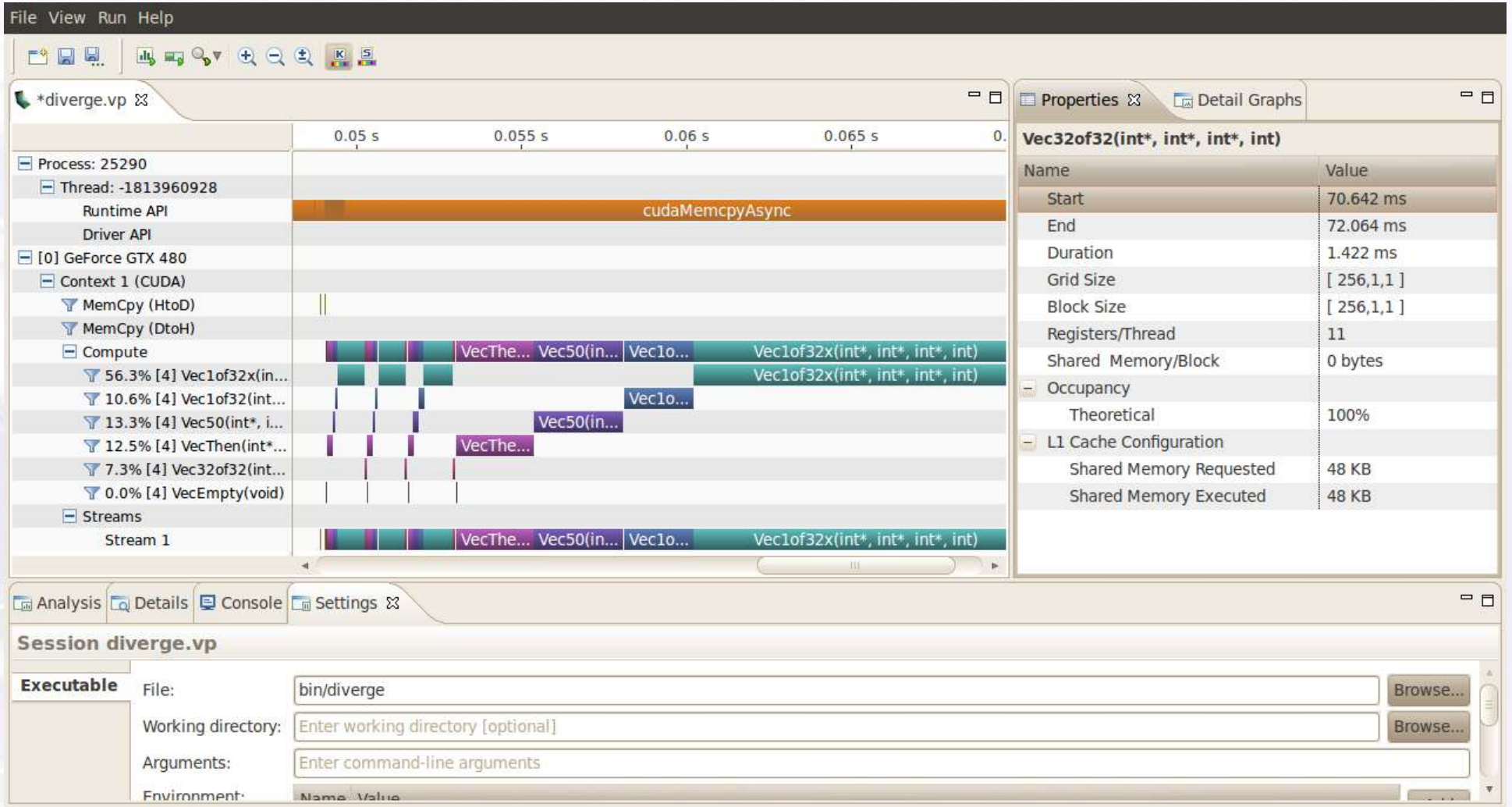
NVIDIA VISUAL PROFILER

Profiling Session

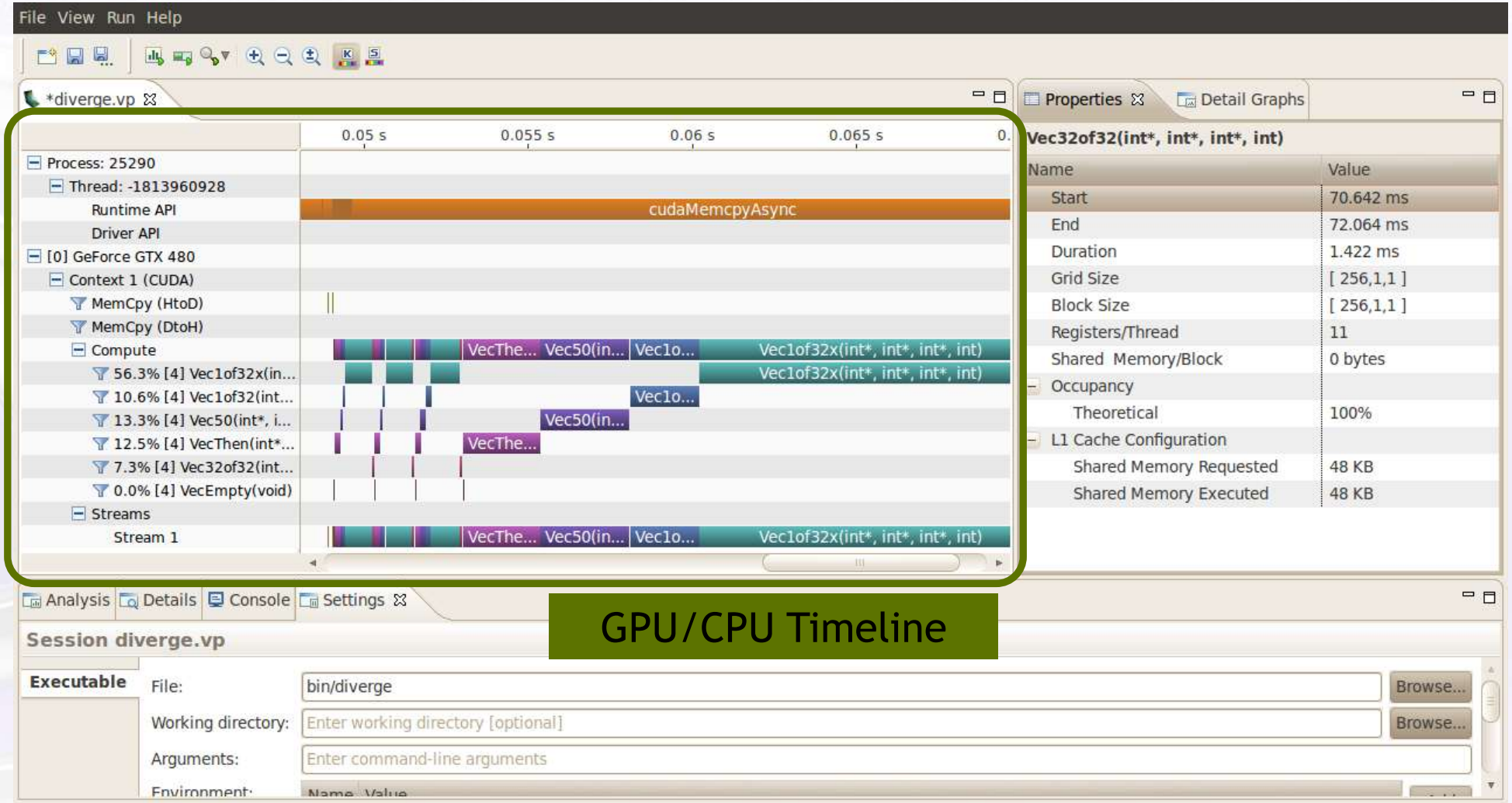




NVIDIA Visual Profiler

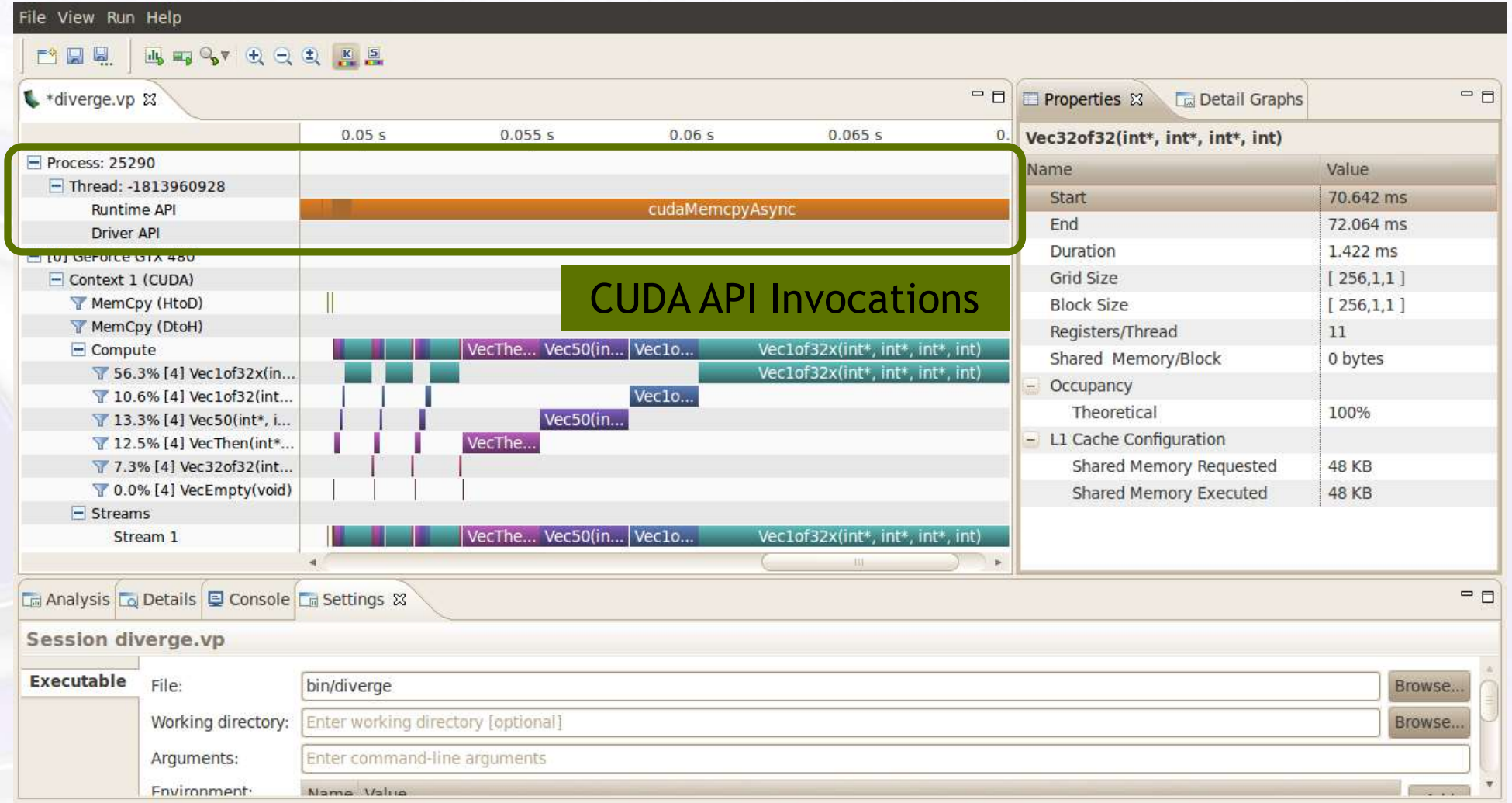


Timeline

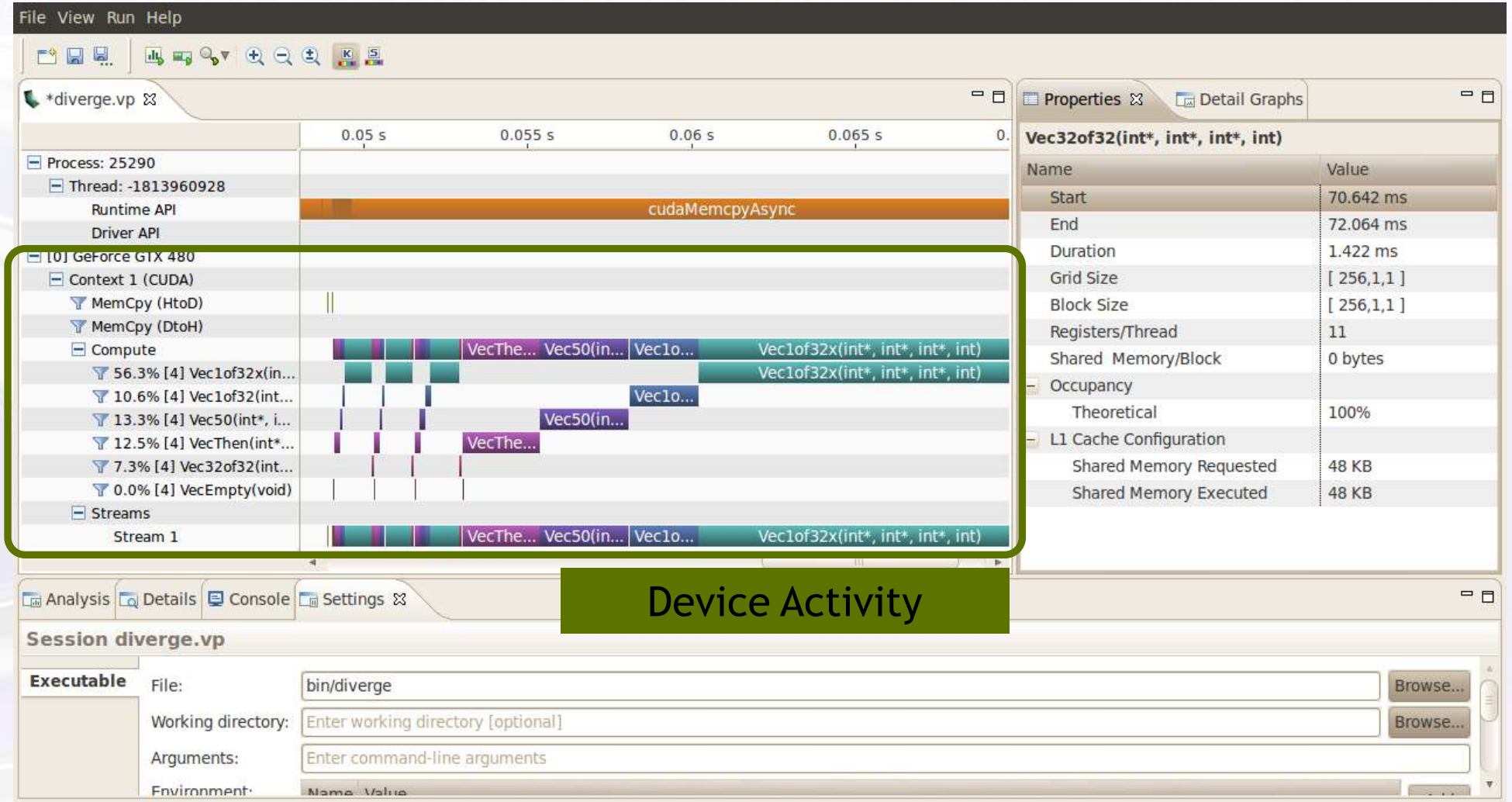


GPU/CPU Timeline

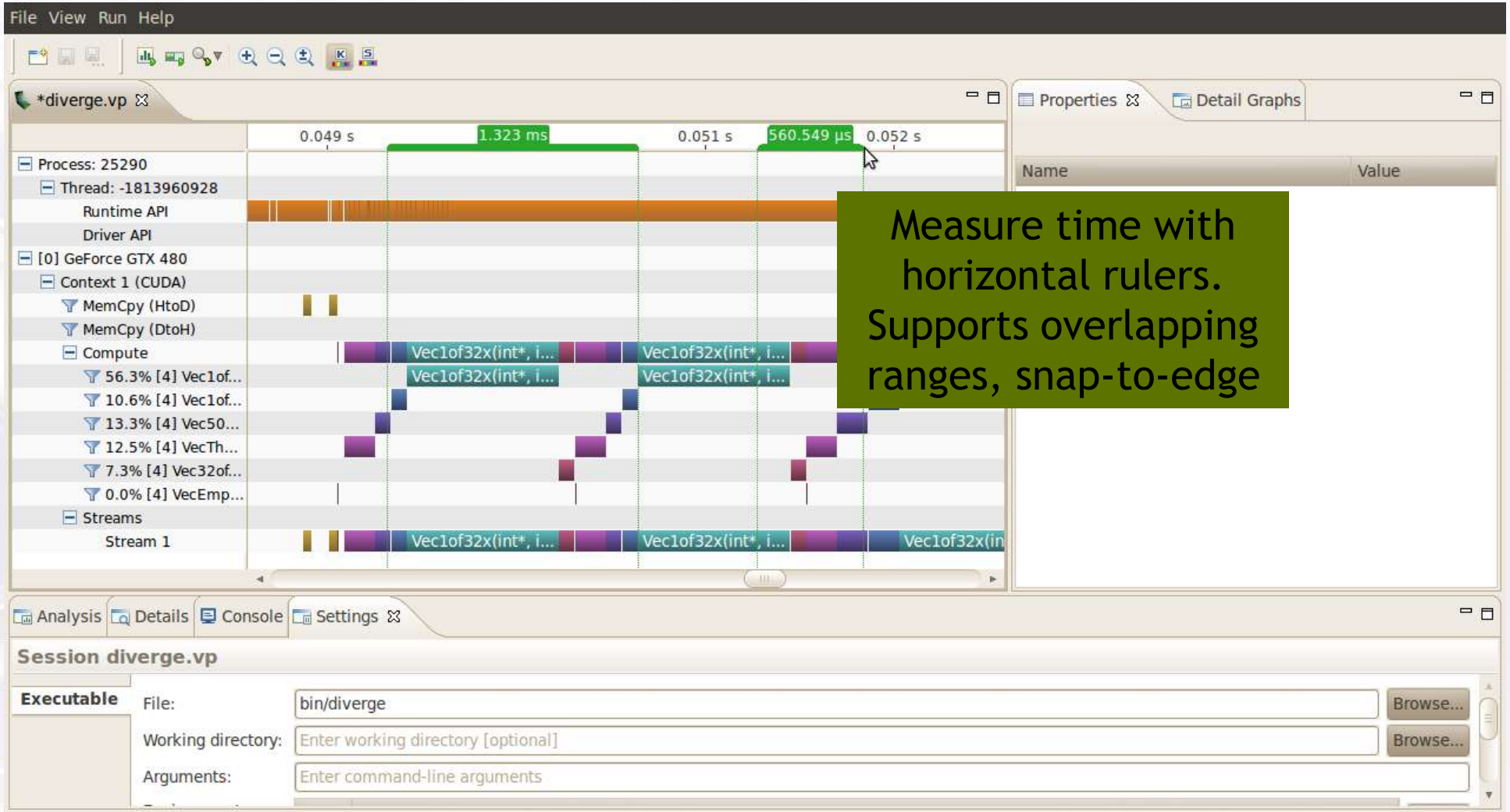
CPU Timeline



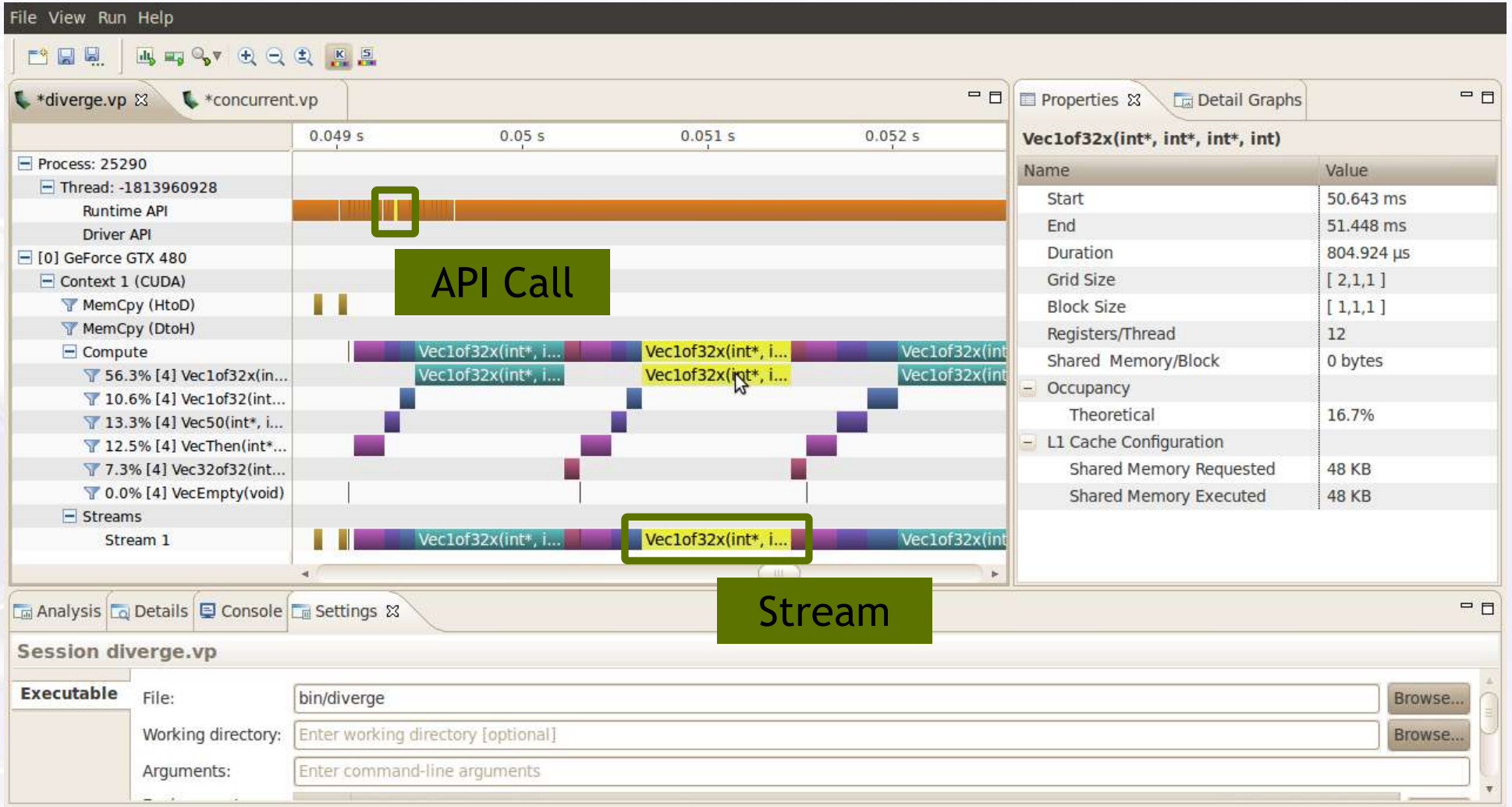
GPU Timeline



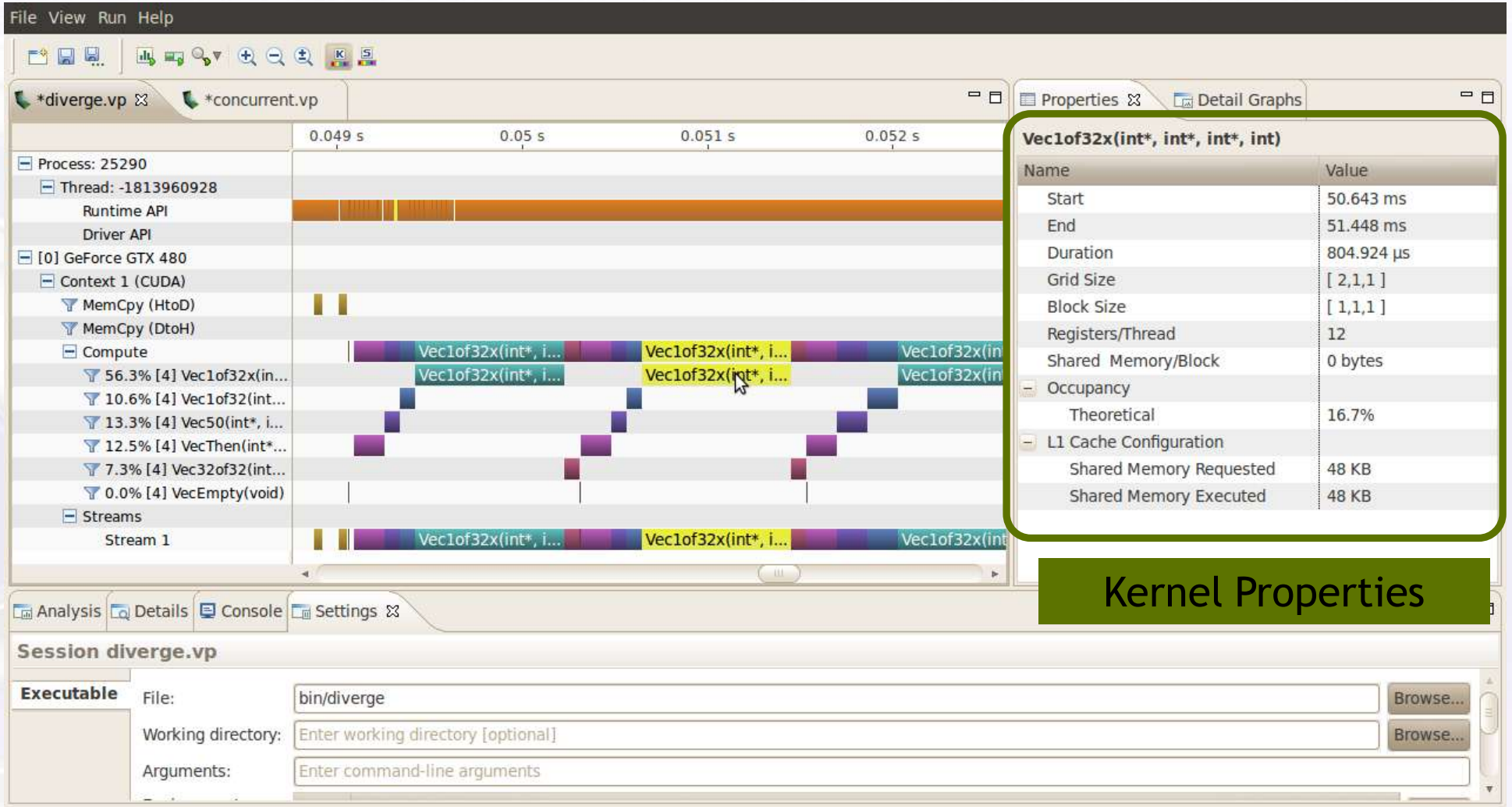
Measuring Time



Correlating CPU and GPU Activity

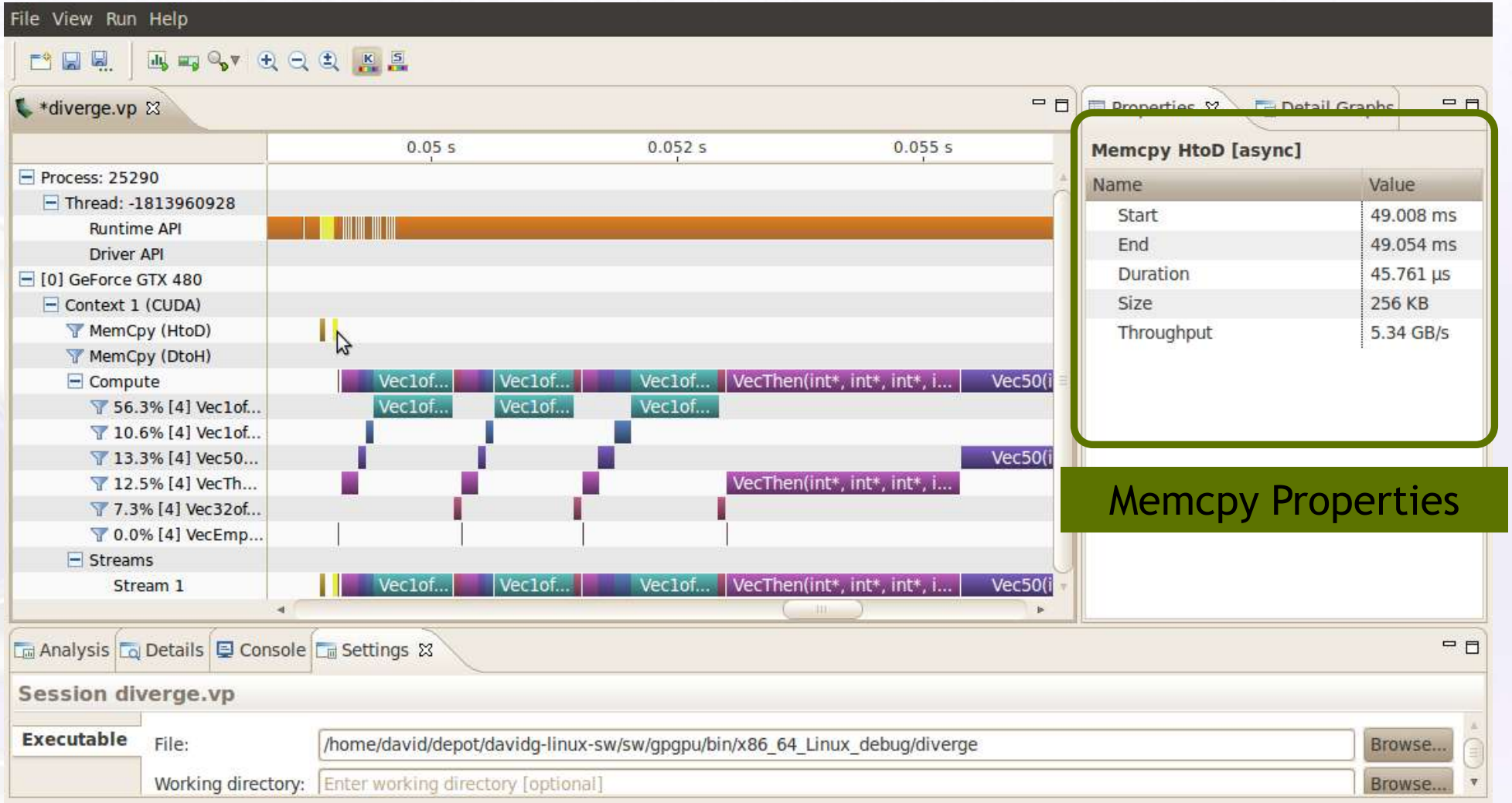


Properties - Kernel

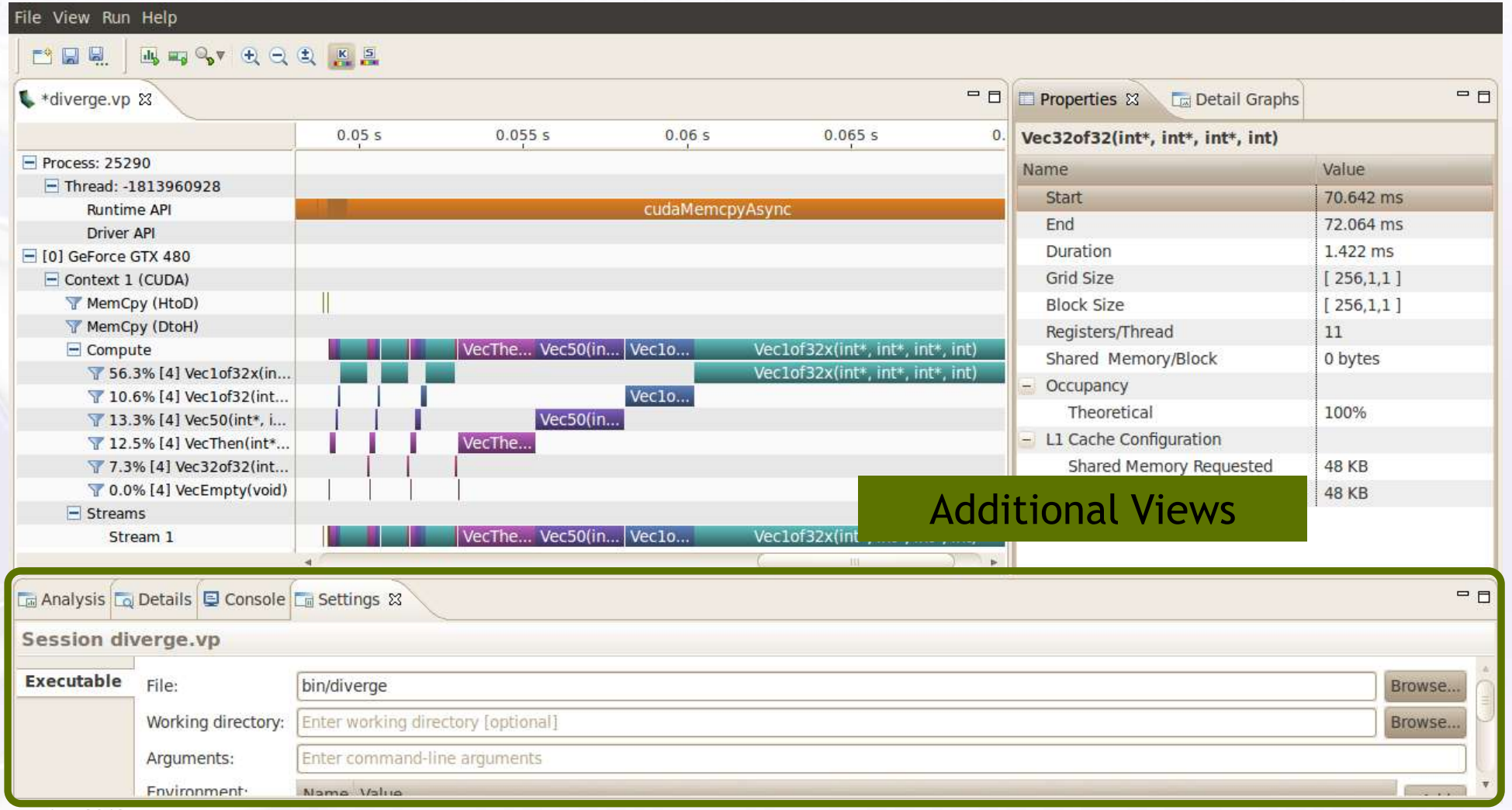


Kernel Properties

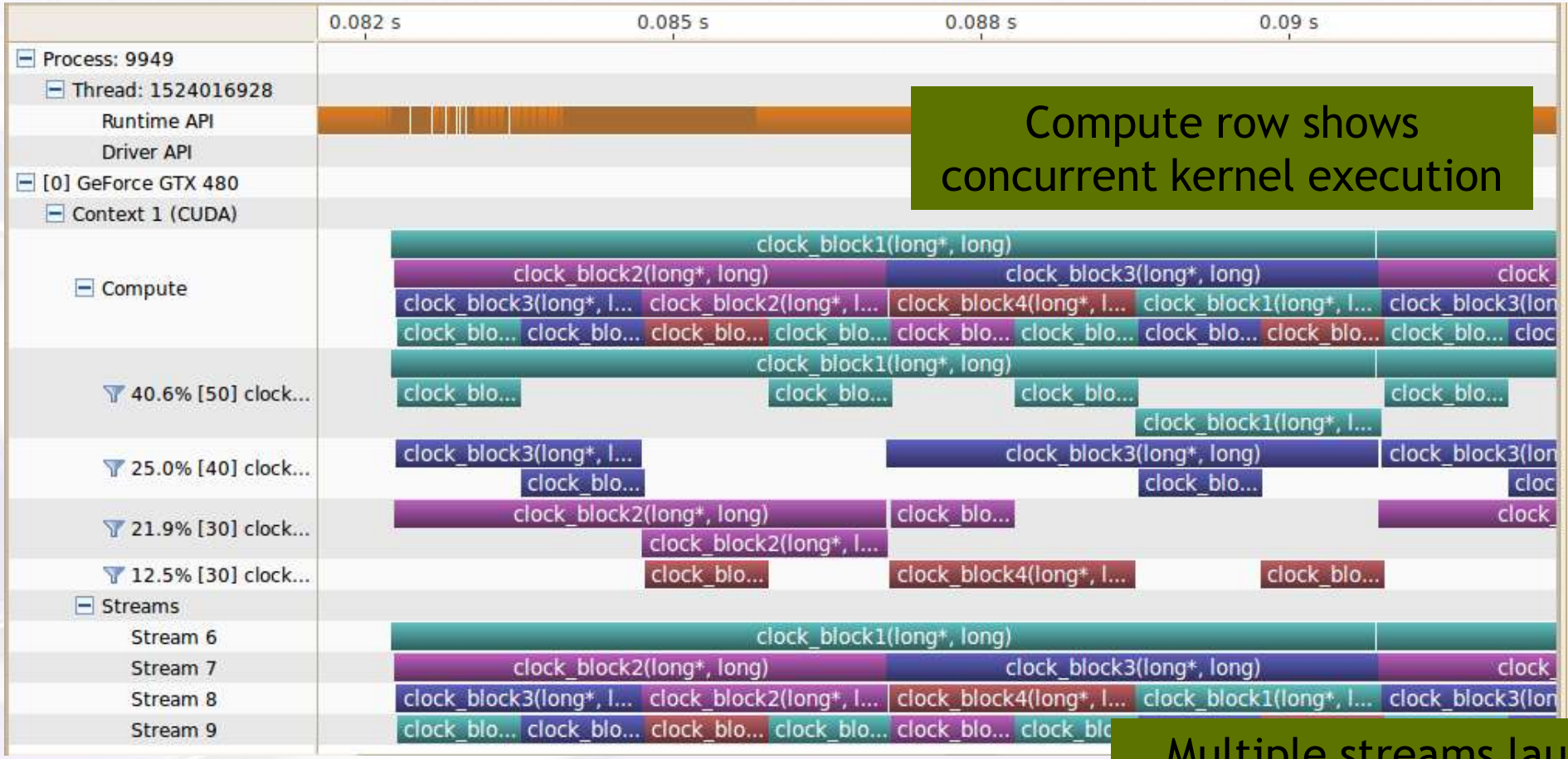
Properties - Memcpy



Analysis, Details, etc.



Concurrent 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

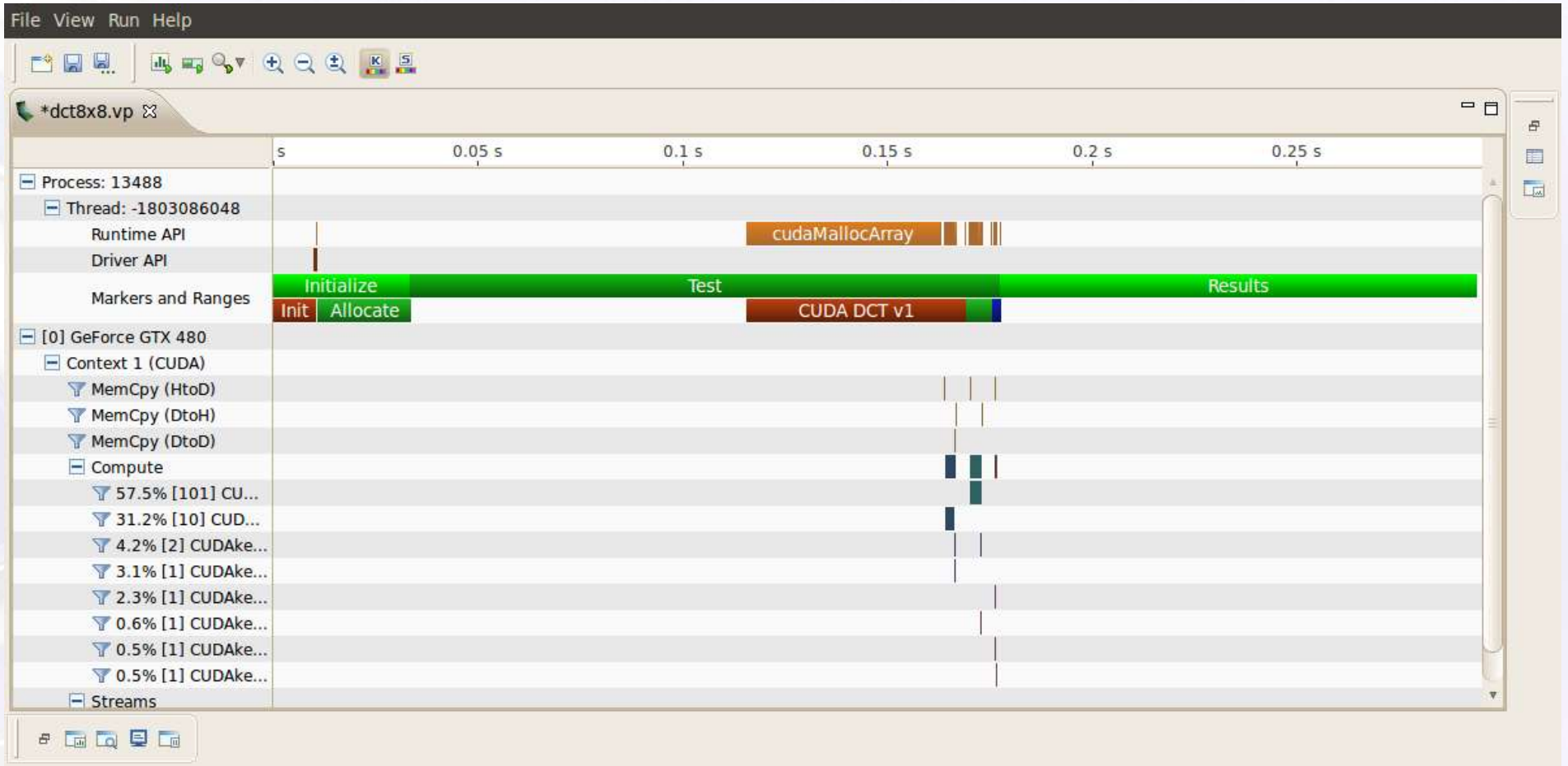
- Developer API for CPU code
- Installed with CUDA Toolkit (`libnvToolsExt.so`)
- Naming
  - Host OS threads: nvtxNameOsThread()
  - CUDA device, context, stream: nvtxNameCudaStream()
- Time Ranges and Markers
  - Range: nvtxRangeStart(), nvtxRangeEnd()
  - Instantaneous marker: nvtxMark()

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



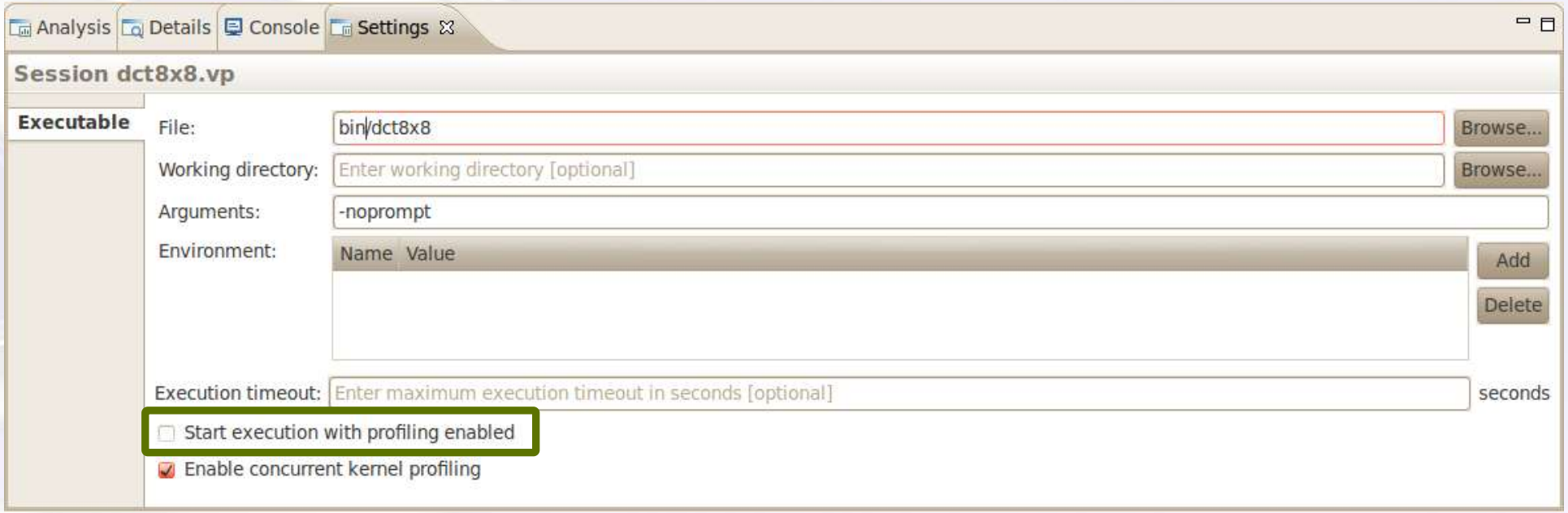
Profile Region Of Interest

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

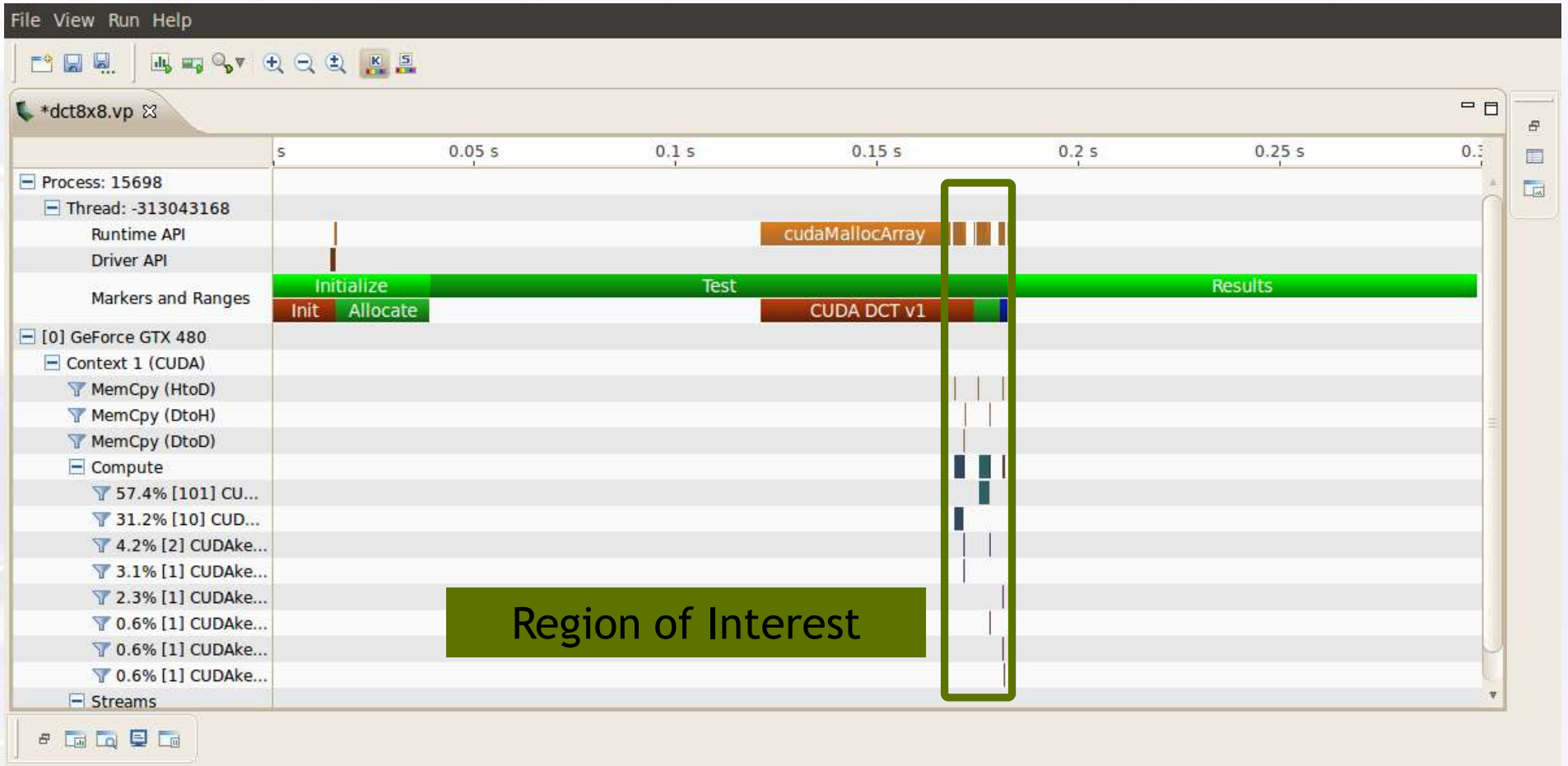
```
for (i = 0; i < N; i++) {
    if (i == 12) cudaProfilerStart();
    <loop body>
    if (i == 15) cudaProfilerStop();
}
```

Enable Region Of Interest

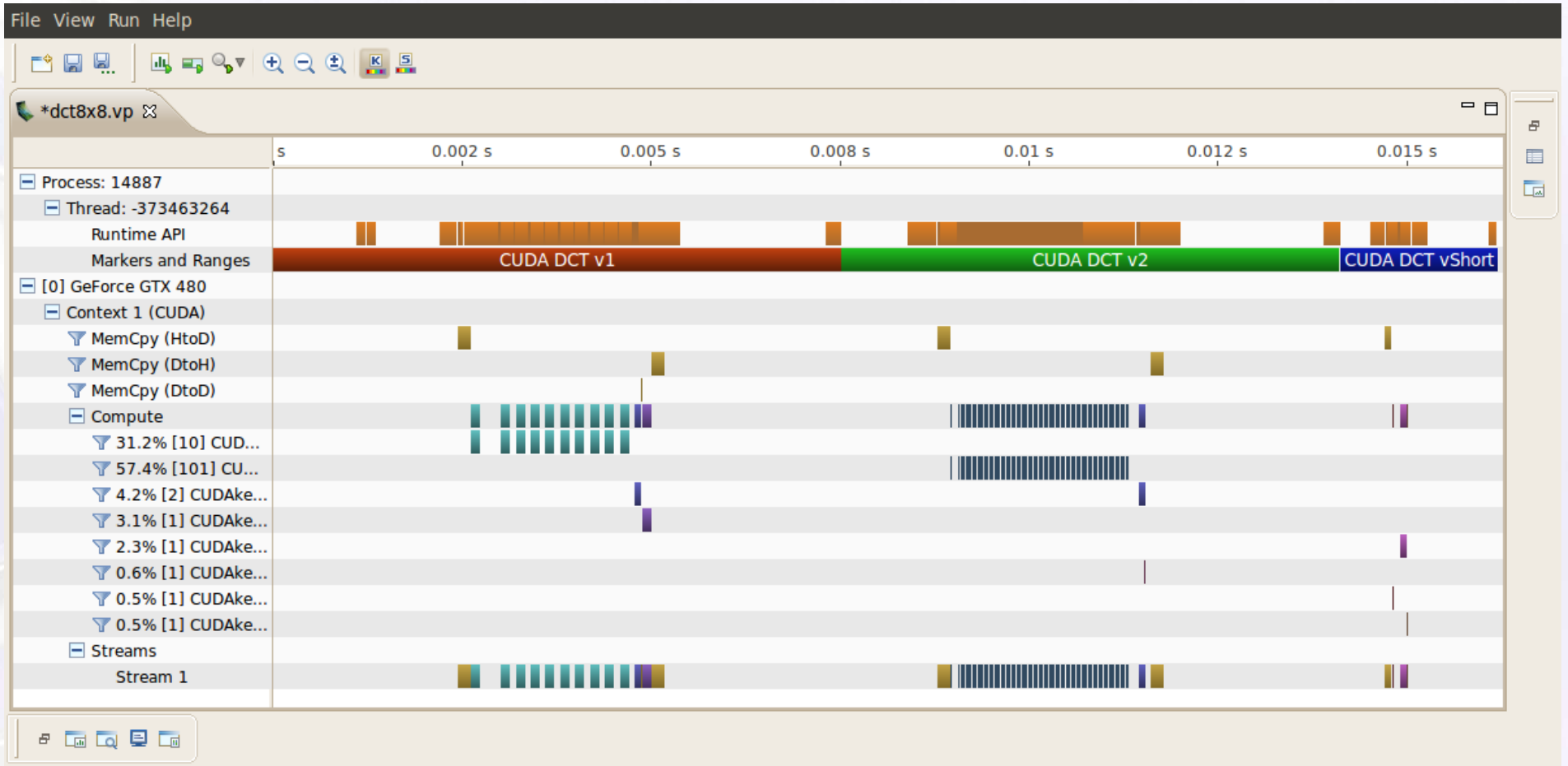
- Insert cudaProfilerStart() / cudaProfilerStop()
- Disable profiling at start of application



Example: Without cudaProfilerStart/Stop



Example: With cudaProfilerStart/Stop



Analysis

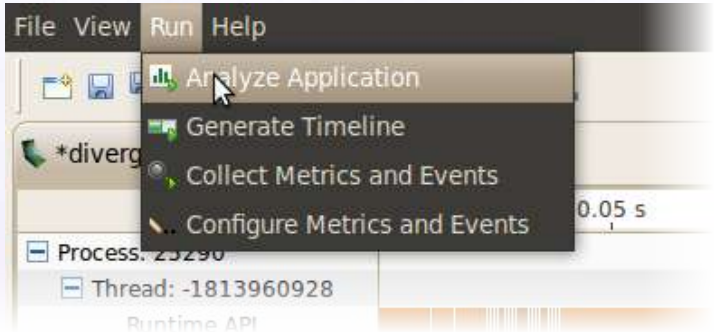
- Visual inspection of timeline
- Automated Analysis
- Metrics and Events

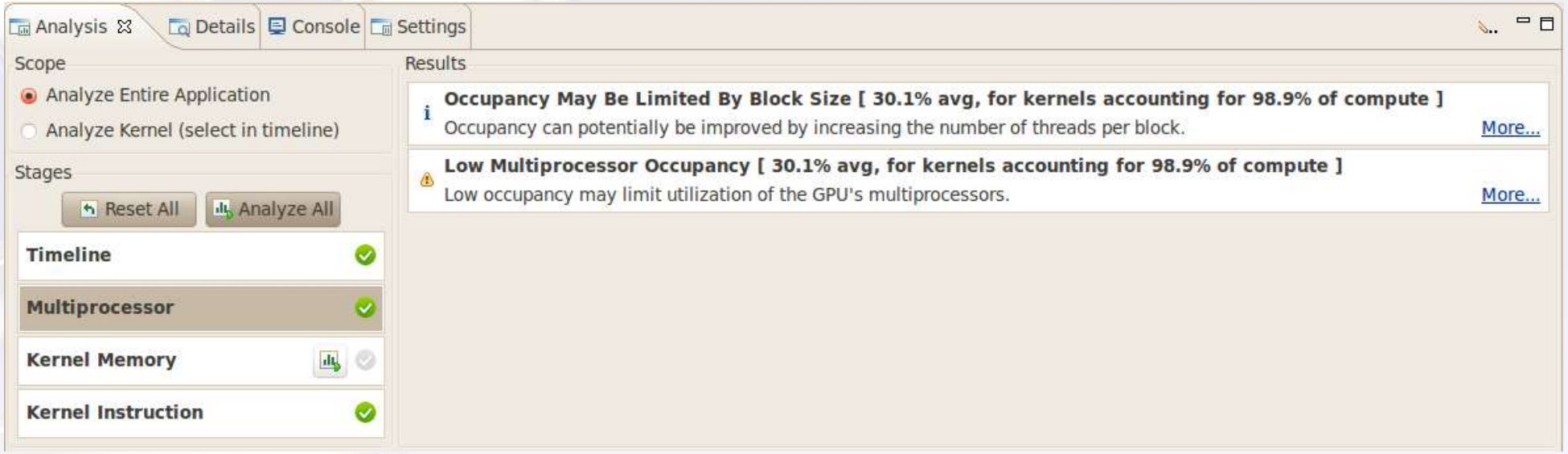
Visual Inspection

- 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

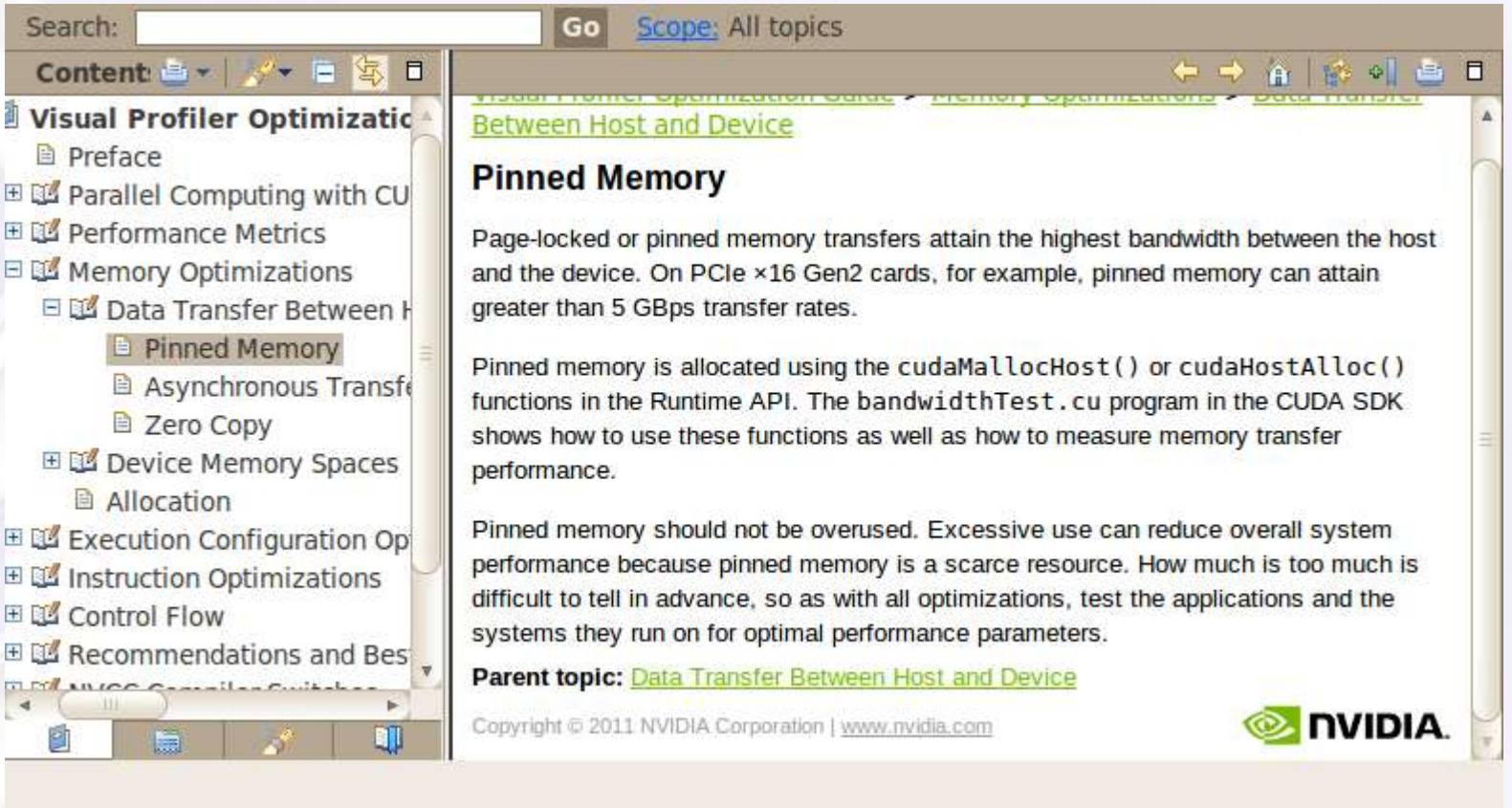
- **Analyse entire application**
  - **Timeline**
  - **Hardware performance counters**



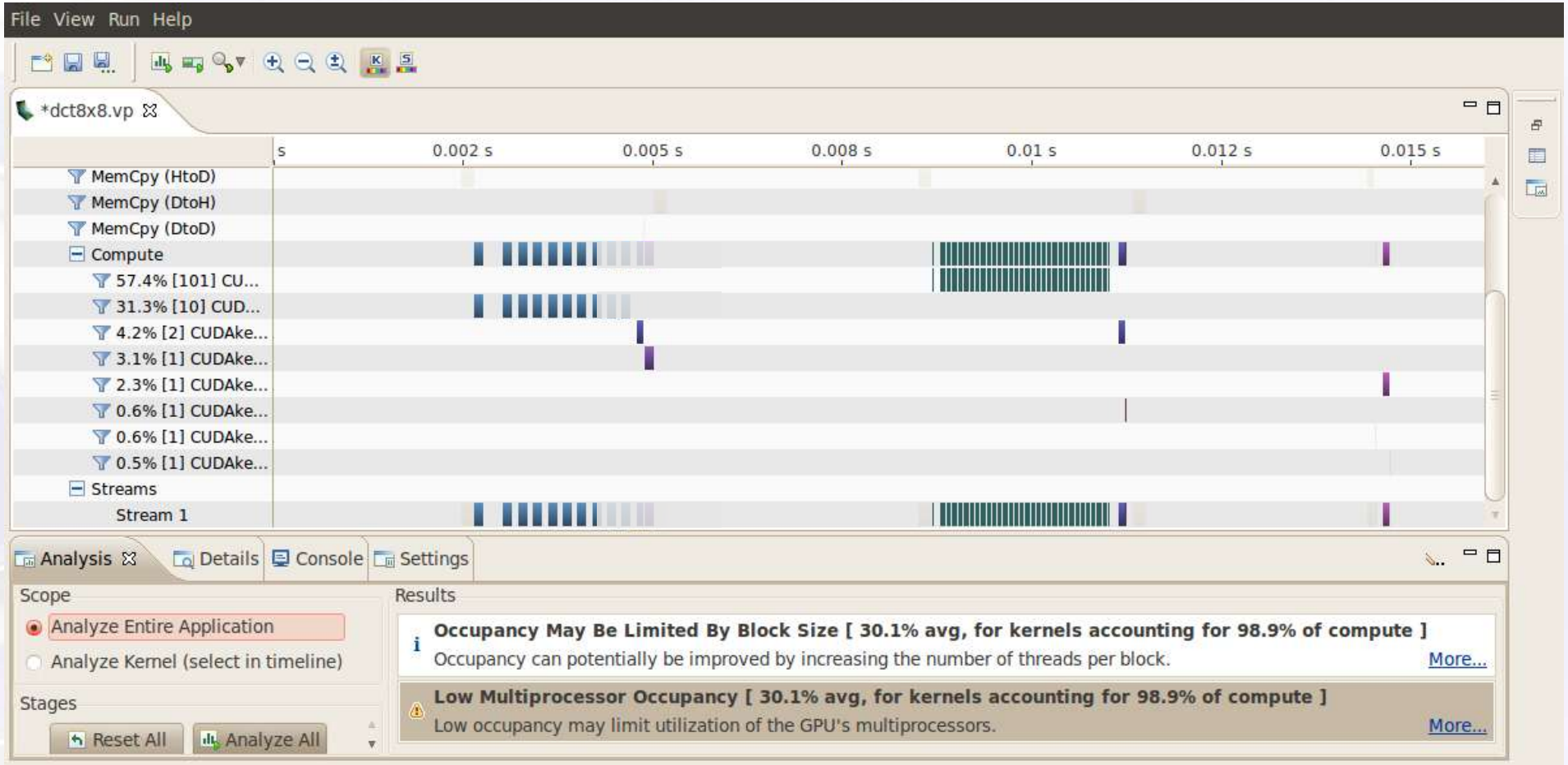


Analysis Documentation

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

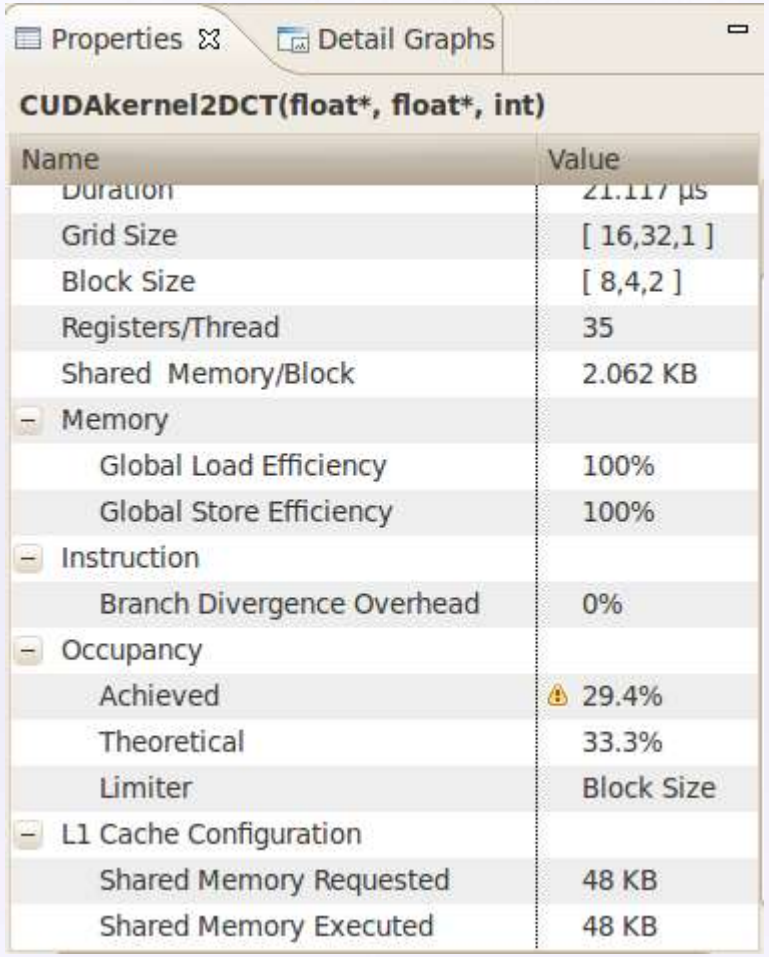


Results Correlated With Timeline

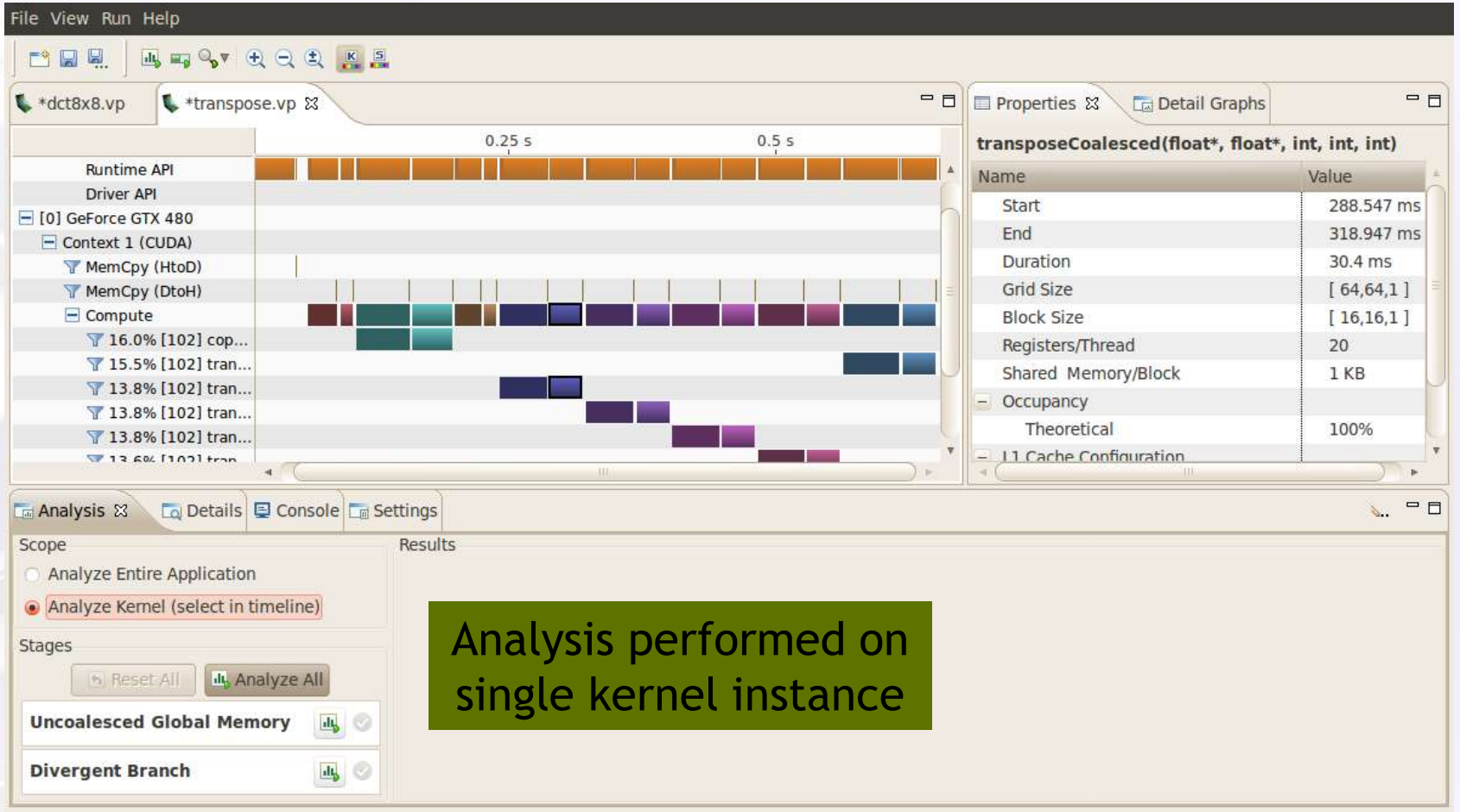


Analysis Properties

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



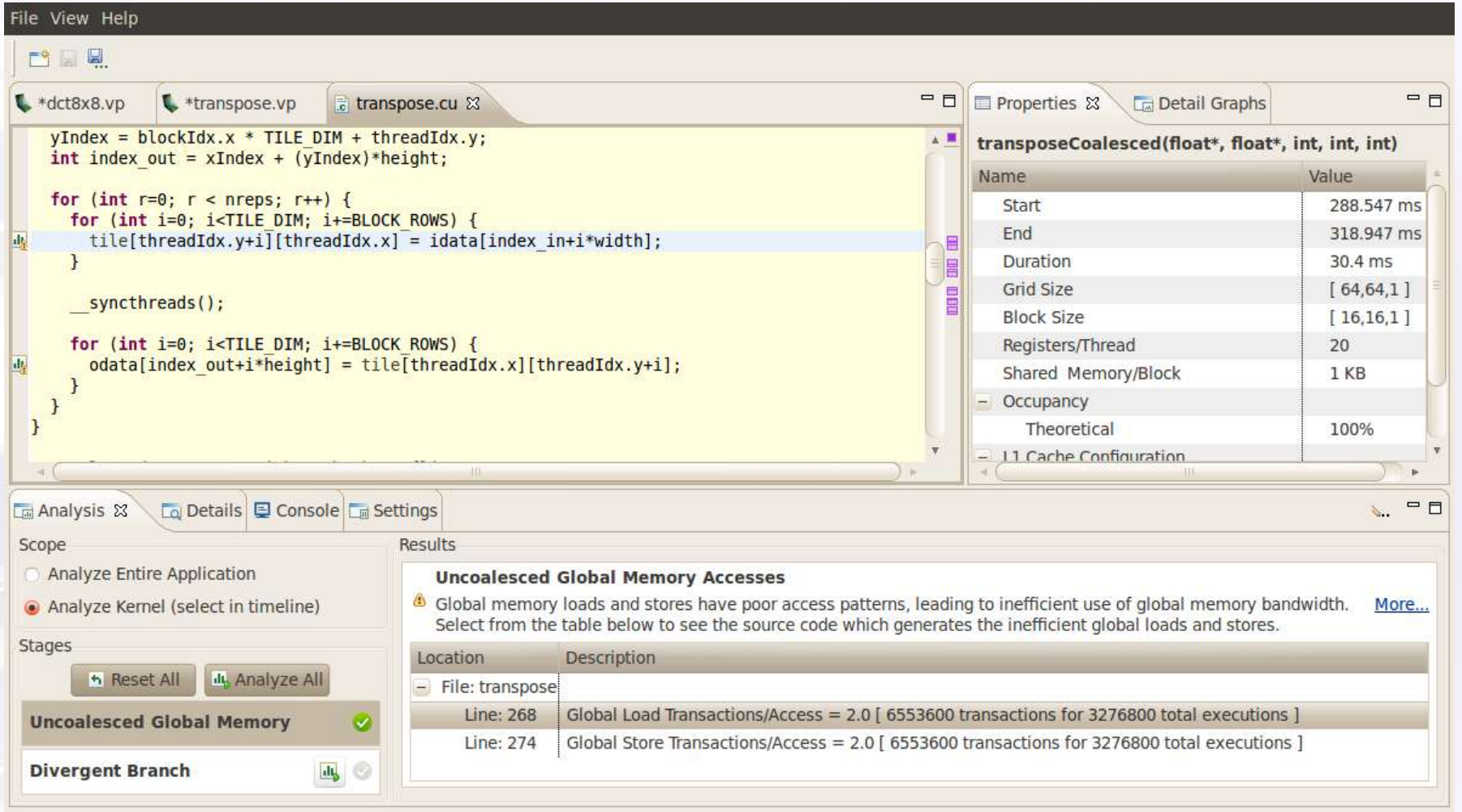
Automated Analysis – Single Kernel



Uncoalesced Global Memory Accesses

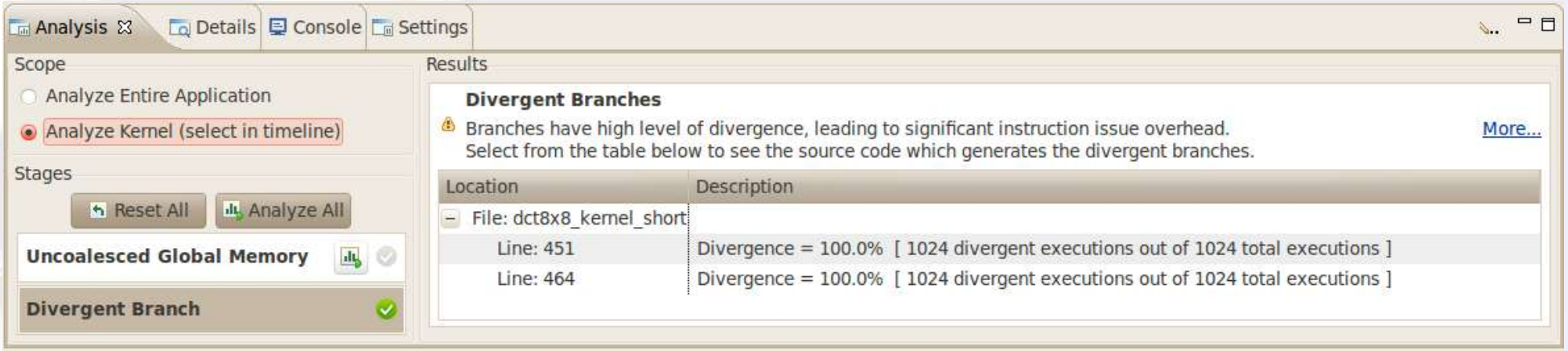
- **Access pattern determines number of memory transactions**
  - **Report loads/stores where access pattern if inefficient**

Source Correlation

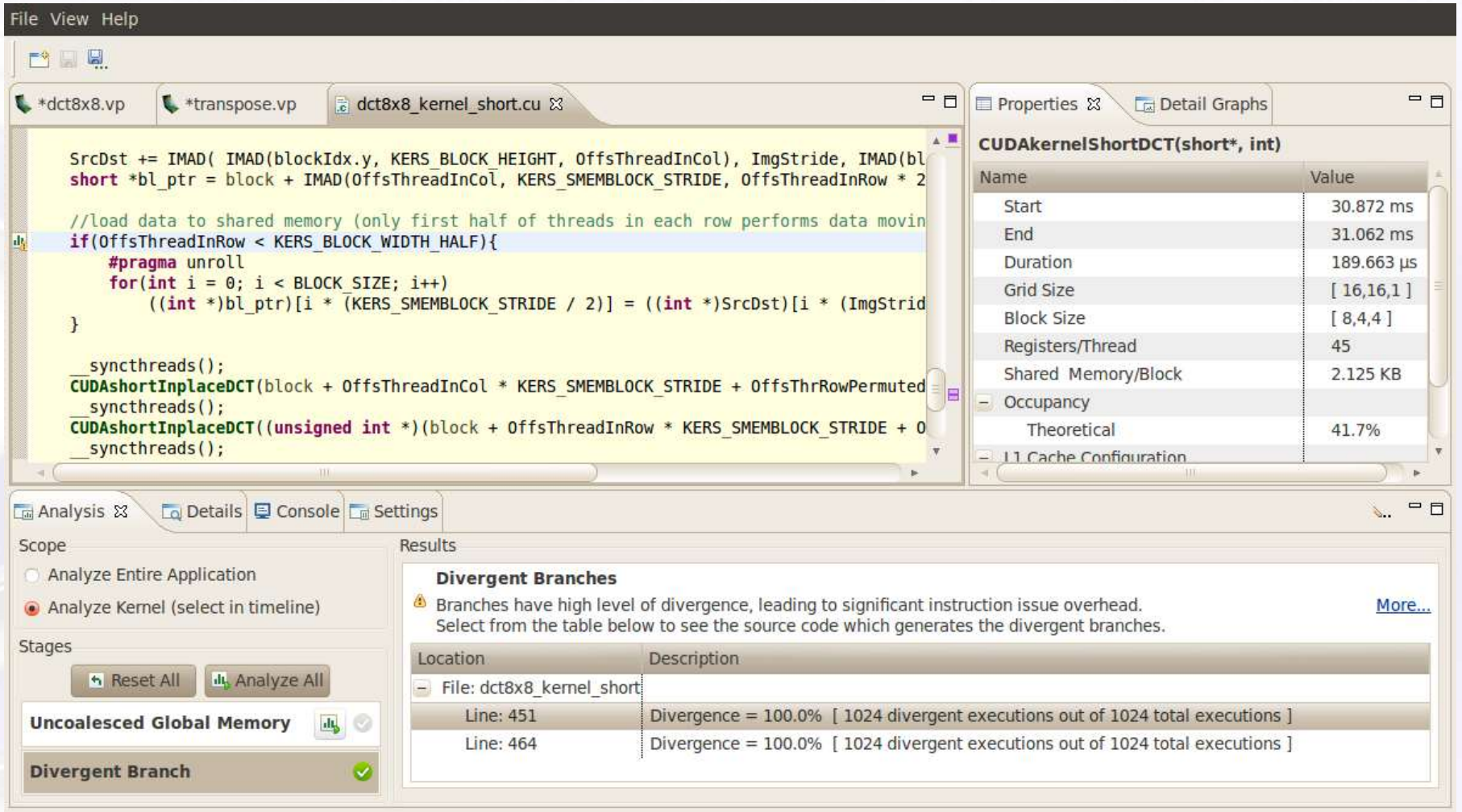


Divergent Branches

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



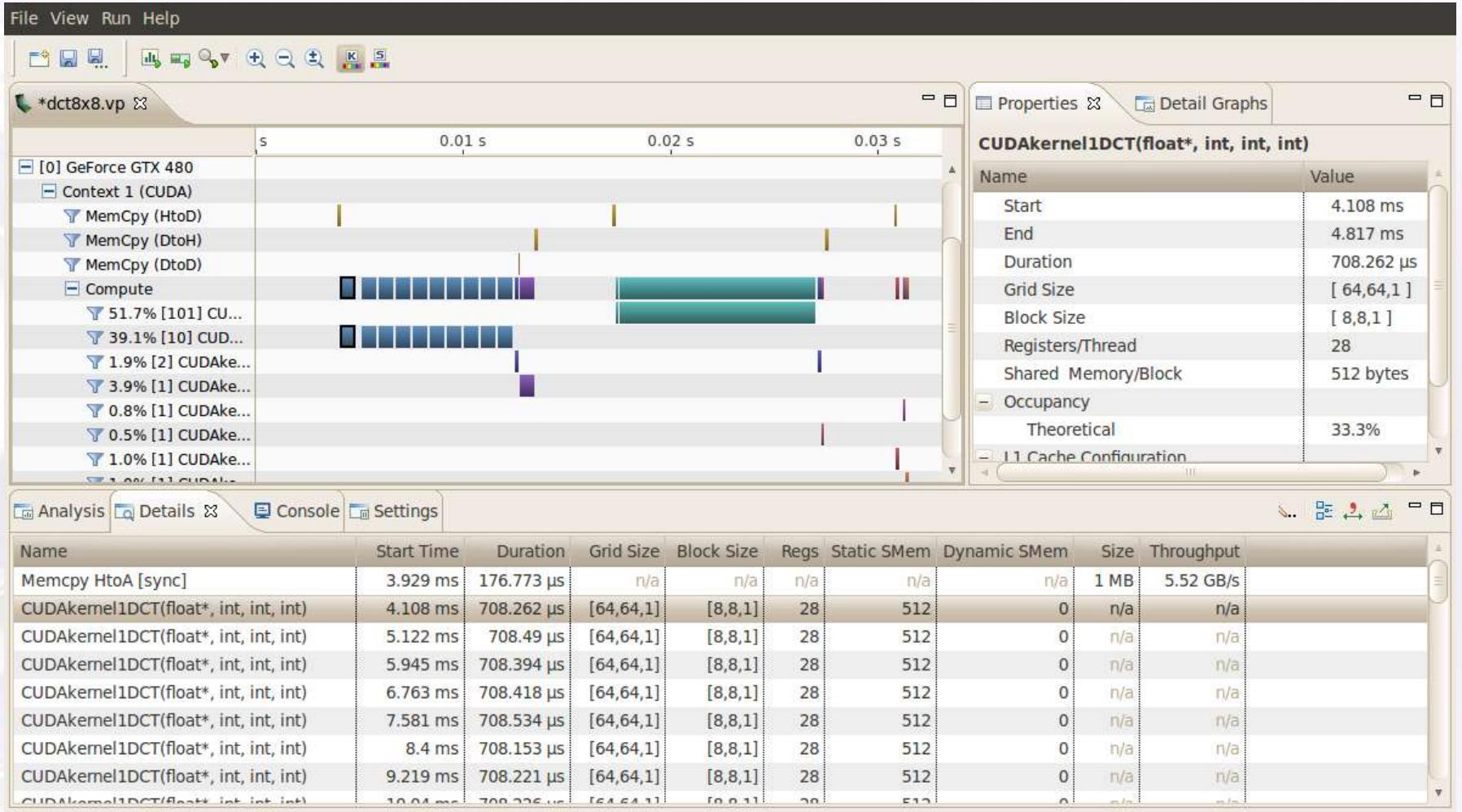
Source Correlation



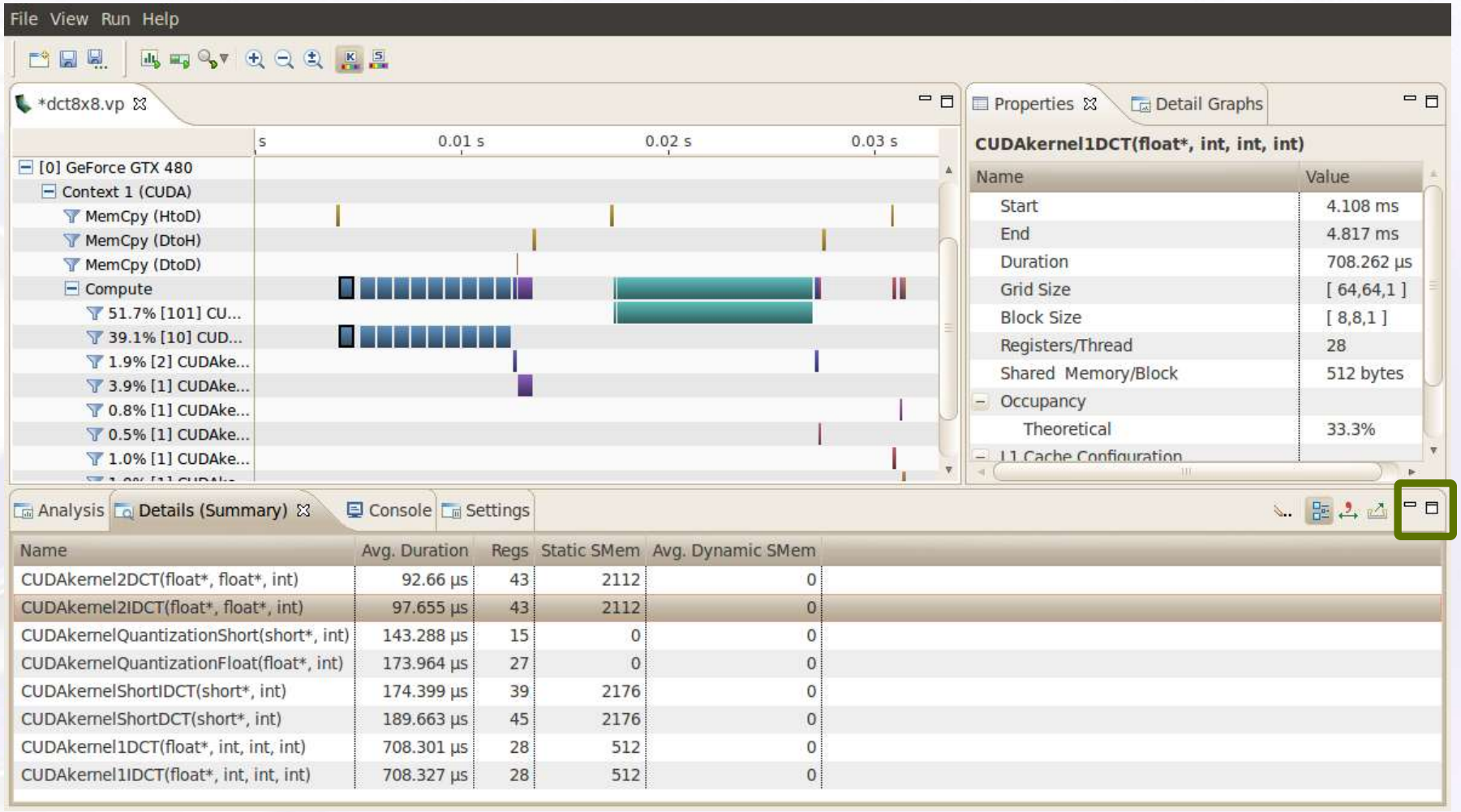
Enabling Source Correlation

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

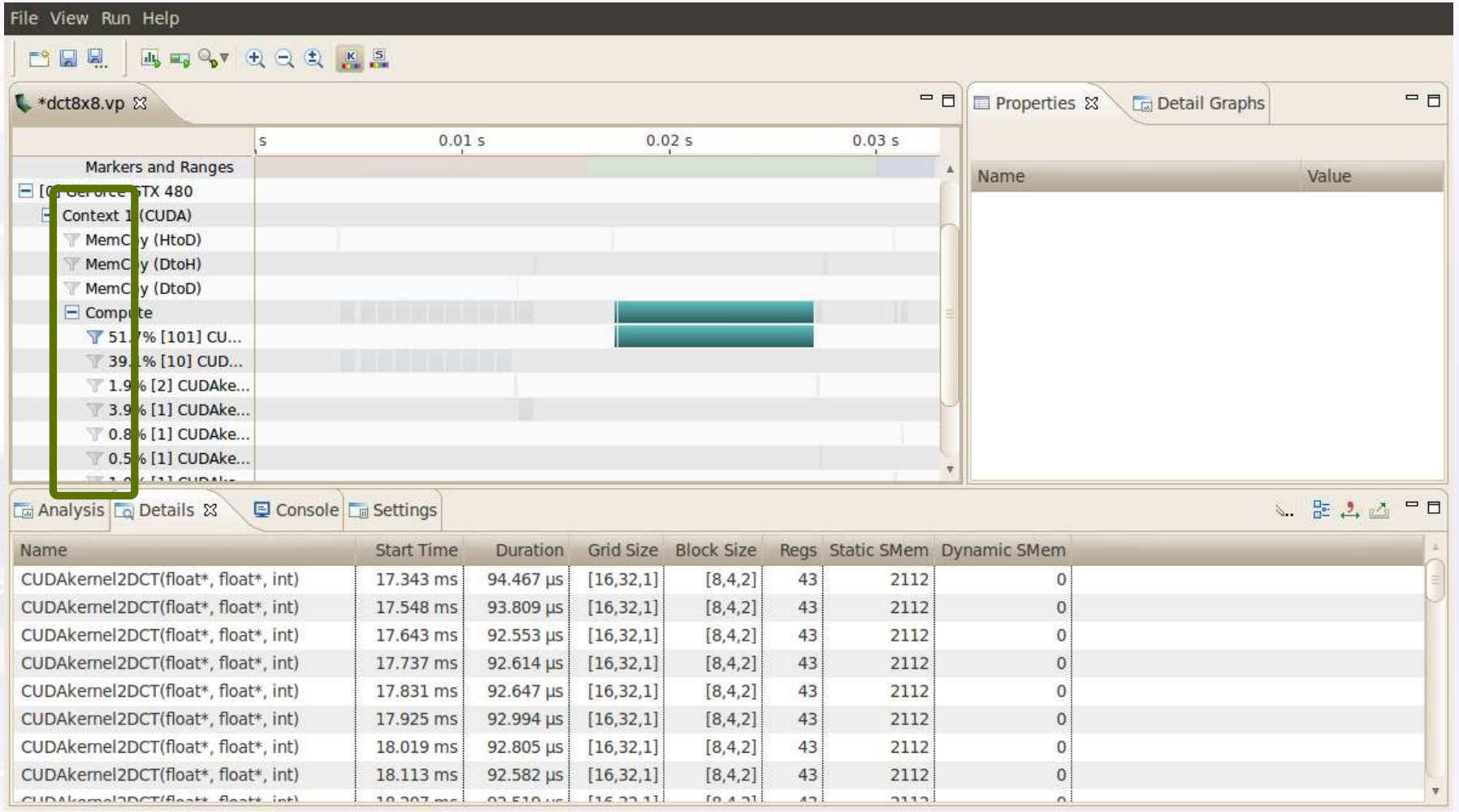
Detailed Profile Data



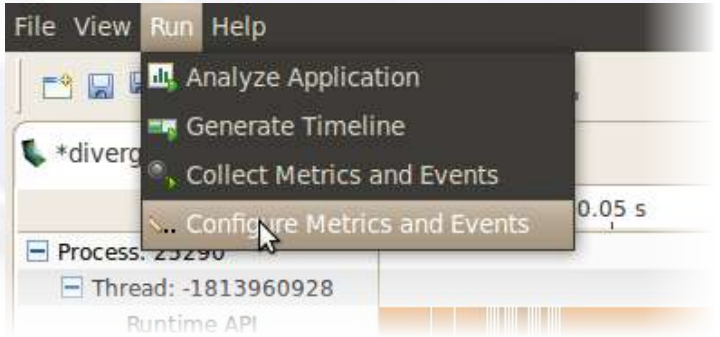
Detailed Summary Profile Data

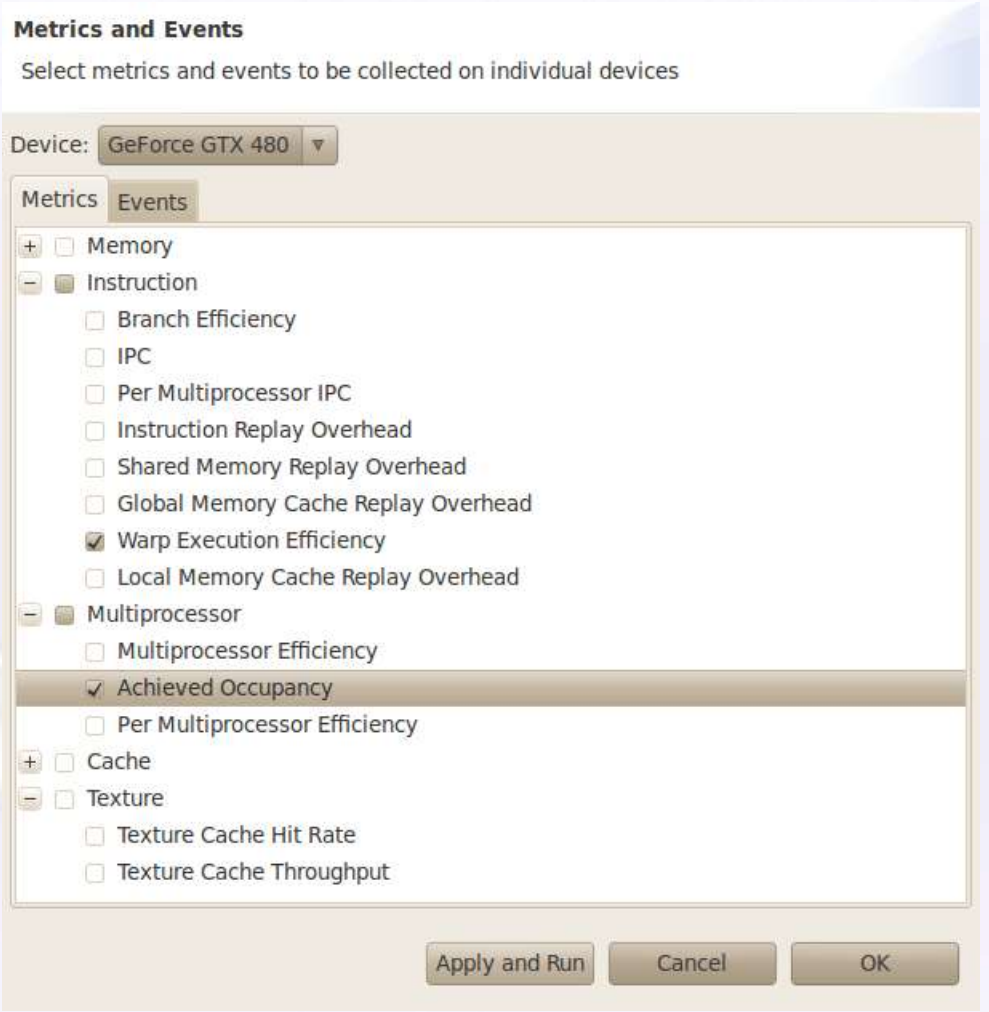


Filtering



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 S
Memcpy HtoA [sync]	3.929 ms	176.773 µs	n/a	n/a	n/a	n/a	n/a	n/a	n/a
CUDAkernel1DCT(float*, int, int, int)	4.108 ms	708.262 µs	100%	0.328	[64,64,1]	[8,8,1]	28	512	0
CUDAkernel1DCT(float*, int, int, int)	5.122 ms	708.49 µs	100%	0.328	[64,64,1]	[8,8,1]	28	512	0
CUDAkernel1DCT(float*, int, int, int)	5.945 ms	708.394 µs	100%	0.327	[64,64,1]	[8,8,1]	28	512	0
CUDAkernel1DCT(float*, int, int, int)	6.763 ms	708.418 µs	100%	0.328	[64,64,1]	[8,8,1]	28	512	0
CUDAkernel1DCT(float*, int, int, int)	7.581 ms	708.534 µs	100%	0.327	[64,64,1]	[8,8,1]	28	512	0
CUDAkernel1DCT(float*, int, int, int)	8.4 ms	708.153 µs	100%	0.327	[64,64,1]	[8,8,1]	28	512	0
CUDAkernel1DCT(float*, int, int, int)	9.219 ms	708.221 µs	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 µs	43	2112	0
CUDAkernel2IDCT(float*, float*, int)	100%	0.302	97.655 µs	43	2112	0
CUDAkernelQuantizationShort(short*, int)	67.5%	0.317	143.288 µs	15	0	0
CUDAkernelQuantizationFloat(float*, int)	98.7%	0.318	173.964 µs	27	0	0
CUDAkernelShortIDCT(short*, int)	74.7%	0.468	174.399 µs	39	2176	0
CUDAkernelShortDCT(short*, int)	75%	0.376	189.663 µs	45	2176	0
CUDAkernel1DCT(float*, int, int, int)	100%	0.328	708.301 µs	28	512	0
CUDAkernel1IDCT(float*, int, int, int)	100%	0.328	708.327 µs	28	512	0

NVPROF

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  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 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: 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 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

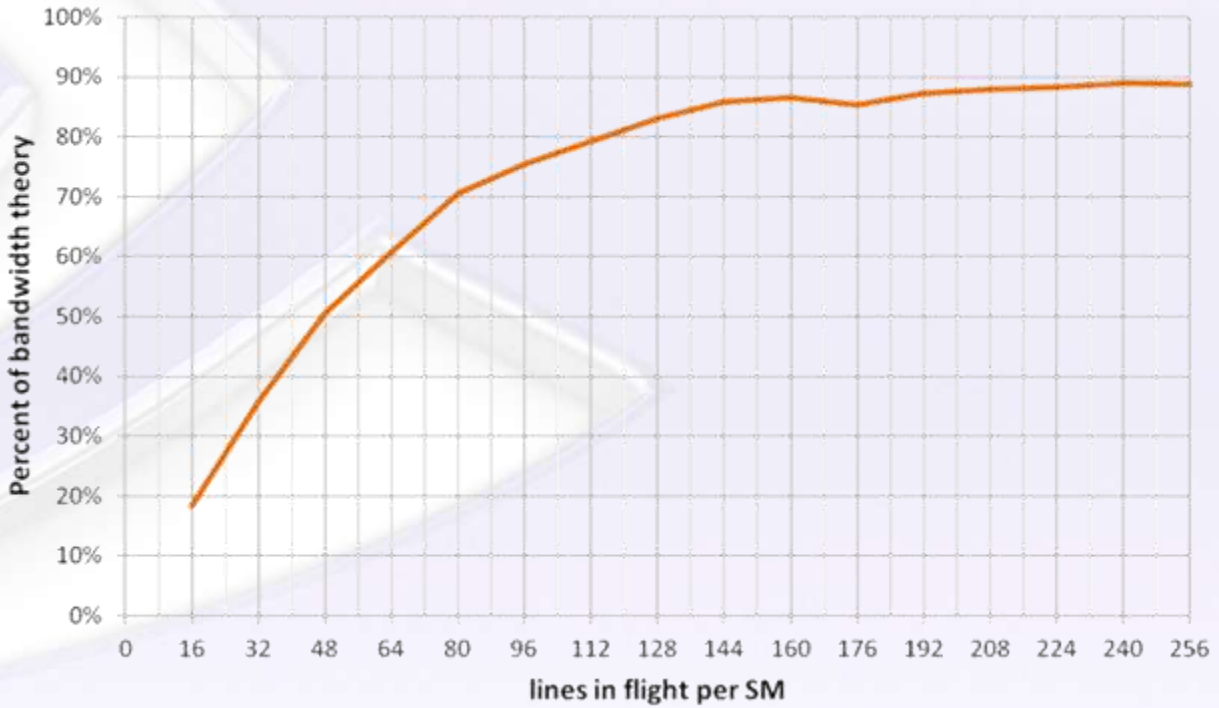
Kepler: Level of Parallelism Needed

- 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

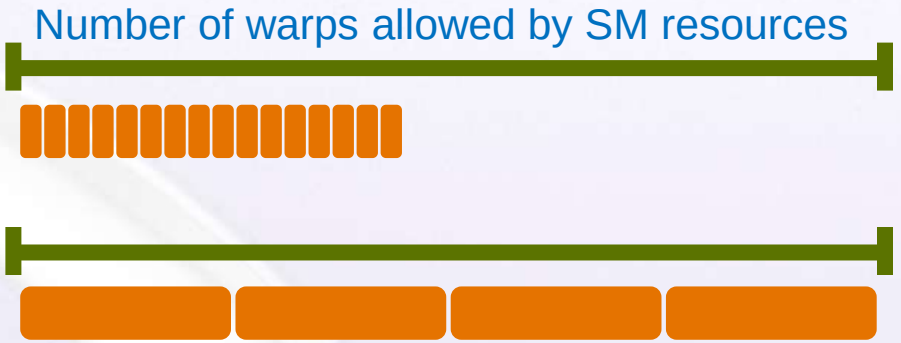
- Grid: arrangement of threads into threadblocks
- Two goals:
  - Expose enough parallelism to an SM
  - Balance work across the SMs
- Several things to consider when launching kernels:
  - Number of threads per threadblock
  - Number of threadblocks
  - Amount of work per threadblock

Threadblock Size and Occupancy

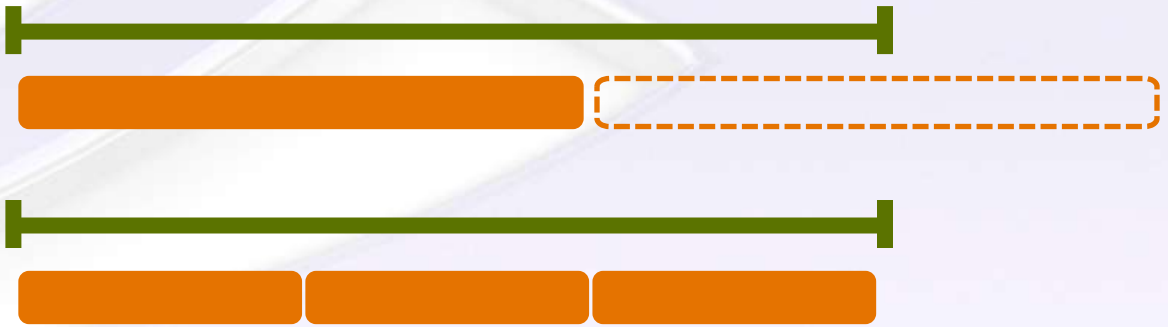
- Threadblock size is a multiple of warp size (32)
  - Even if you request fewer threads, HW rounds up
- Threadblocks can be too small
  - Kepler SM can run up to 16 threadblocks concurrently
  - SM may reach the block limit before reaching good occupancy
    - Example: 1-warp blocks -> 16 warps per 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



Too many threads per block



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

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 strutures 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