STANFORD
UNIVERSITY

CME213/ME339 Nvidia Visual Profiler

Jonathan Wong

Nvidia Tools

STANFORD UNIVERSITY

- nvcc – compiler
  - Generates object files, executable
  - Displays register and shared memory usage
- cuda-gdb – CUDA debugger
  - Used to debug code
    - Compile with "–g –G"
- cuda-memcheck – CUDA version of memcheck
  - Checks for obvious memory leaks
- nvvp – Nvidia Visual Profiler (formerly cudaprof)
  - Used for measuring and optimizing code

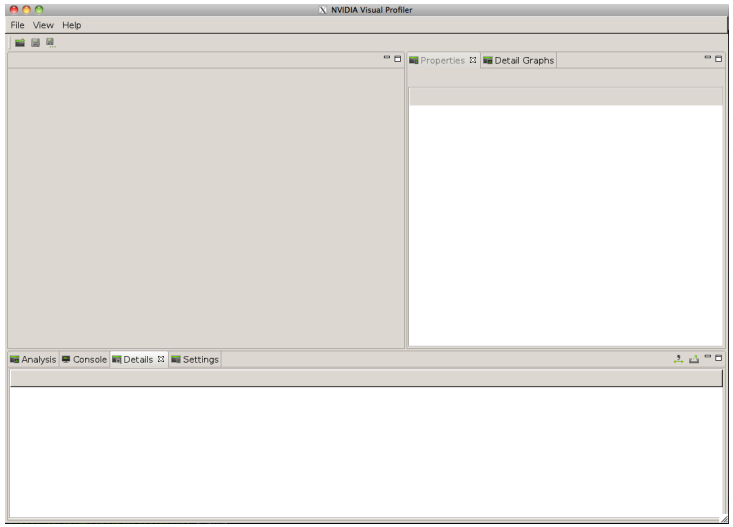
Profiling

STANFORD UNIVERSITY

- Wikipedia
  - “In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the usage of memory, the usage of particular instructions, or frequency and duration of function calls. The most common use of profiling information is to aid program optimization.
  - Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler).”

Nvidia Visual Profiler

- Visual CUDA executable profiler
  - Comes with CUDA 4.0+
  - Displays several interesting metrics
    - Global Memory
    - Shared Memory
    - Branching
    - # instructions issued
    - # registers used
    - Grid, block sizes
    - Execution time
  - PATH to the visual profiler:
    - /usr/local/cuda/bin/nvvp



Profiler Requirements

STANFORD UNIVERSITY

- What requirements are necessary for profiling CUDA kernels?
  - Don't have to do anything!

- Considerations when profiling:
  - Debug options "-g" or "-G" will affect profile metrics!
  - Optimization options "-OX", where X = [0:3], will also be different
  - Times are averaged over runs

Example

- Hypothetical problem description:
  - Compute the following:

$$y = \sqrt{f(x)} + (x \bmod 7)$$

where, $f(\circ) = \begin{cases} \cos(\circ) & \text{if } x \text{ is odd;} \\ \sin(\circ) & \text{if } x \text{ is even.} \end{cases}$

Kernel0

```
1.  template <uint BLOCK_SIZE>
2.  __global__ void kernel0(uint size, double *input, double *output) {
3.
4.    __shared__ double dontask[BLOCK_SIZE];
5.
6.    uint tid = threadIdx.x;
7.    uint ndx = tid + blockDim.x * blockIdx.x;
8.
9.    dontask[tid] = 5*tid + 2;
10.
11.   if (ndx < size) {
12.     if (ndx % 2) {
13.       output[ndx] = sqrt(cos(input[ndx])+1) + input[ndx] / 7;
14.     } else {
15.       output[ndx] = sqrt(sin(input[ndx])+1) + input[ndx] / 7;
16.     }
17.   }
18.
19. }
```

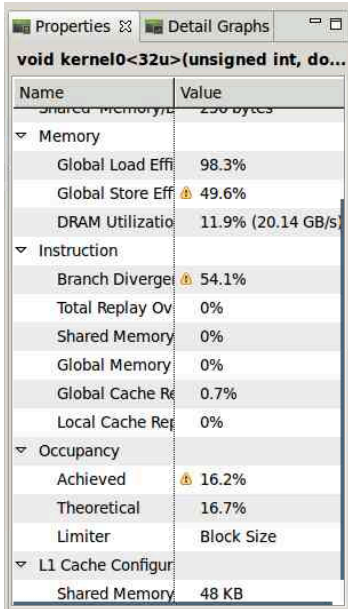
Demonstration Time

- Load CUDA-compiled binary in nvvp
  - Profile and Optimize!

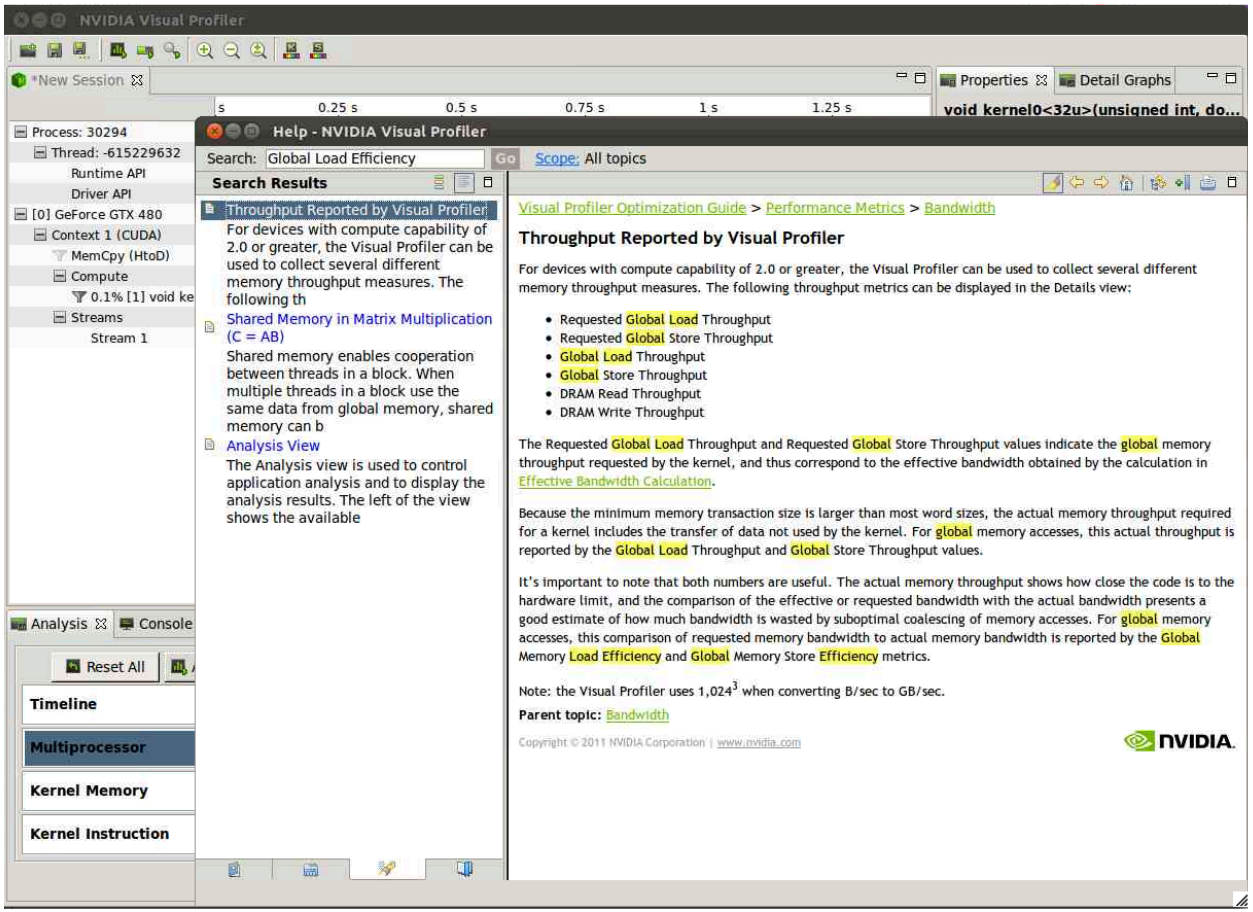
Kernel0

- Details:
  - blocksize = 32
  - Global Store Efficiency : 49.6%
  - Branch Divergence : 54.1 %
  - Occupancy: 16.2 %

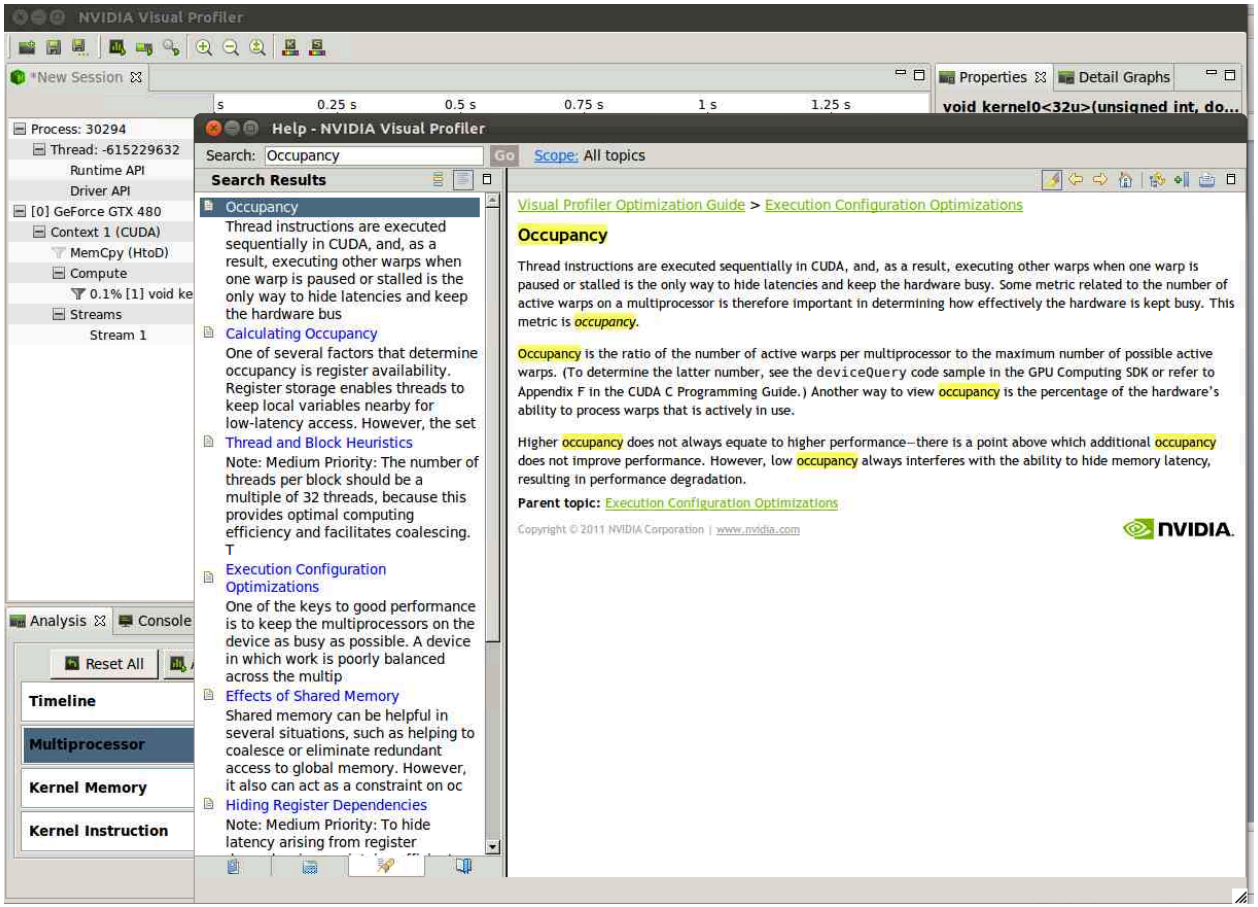
- Let's look at what these terms mean
  - Nvidia Visual Profiler - Help



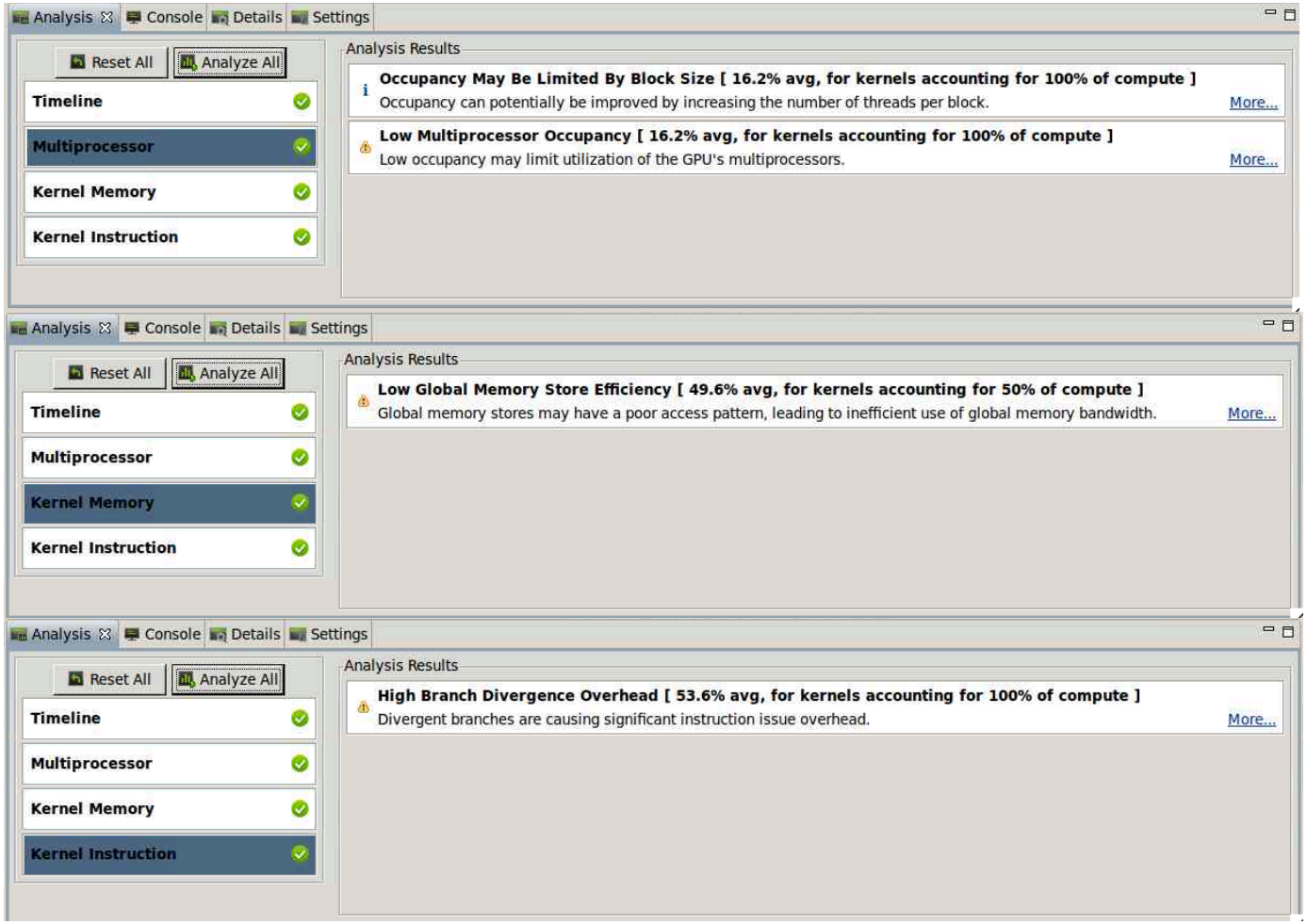
Global Load/Store Efficiency



Occupancy


NVIDIA Visual Profiler
*New Session
Properties
Detail Graphs
0.25 s
0.5 s
0.75 s
1 s
1.25 s
void kernel0<32u>(unsigned int, do...
Process: 30294
Thread: -615229632
Runtime API
Driver API
[0] GeForce GTX 480
Context 1 (CUDA)
MemCpy (HtoD)
Compute
0.1% [1] void ke
Streams
Stream 1
Help - NVIDIA Visual Profiler
Search: Occupancy
Go
Scope: All topics
Search Results
Occupancy
Thread instructions are executed sequentially in CUDA, and, as a result, executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware bus
Calculating Occupancy
One of several factors that determine occupancy is register availability. Register storage enables threads to keep local variables nearby for low-latency access. However, the set
Thread and Block Heuristics
Note: Medium Priority: The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing. T
Execution Configuration Optimizations
One of the keys to good performance is to keep the multiprocessors on the device as busy as possible. A device in which work is poorly balanced across the multip
Effects of Shared Memory
Shared memory can be helpful in several situations, such as helping to coalesce or eliminate redundant access to global memory. However, it also can act as a constraint on oc
Hiding Register Dependencies
Note: Medium Priority: To hide latency arising from register
Visual Profiler Optimization Guide > Execution Configuration Optimizations
Occupancy
Thread instructions are executed sequentially in CUDA, and, as a result, executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware busy. Some metric related to the number of active warps on a multiprocessor is therefore important in determining how effectively the hardware is kept busy. This metric is occupancy.
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. (To determine the latter number, see the deviceQuery code sample in the GPU Computing SDK or refer to Appendix F in the CUDA C Programming Guide.) Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use.
Higher occupancy does not always equate to higher performance--there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.
Parent topic: Execution Configuration Optimizations
Copyright © 2011 NVIDIA Corporation | www.nvidia.com
NVIDIA
Analysis
Console
Reset All
Timeline
Multiprocessor
Kernel Memory
Kernel Instruction

Analysis information

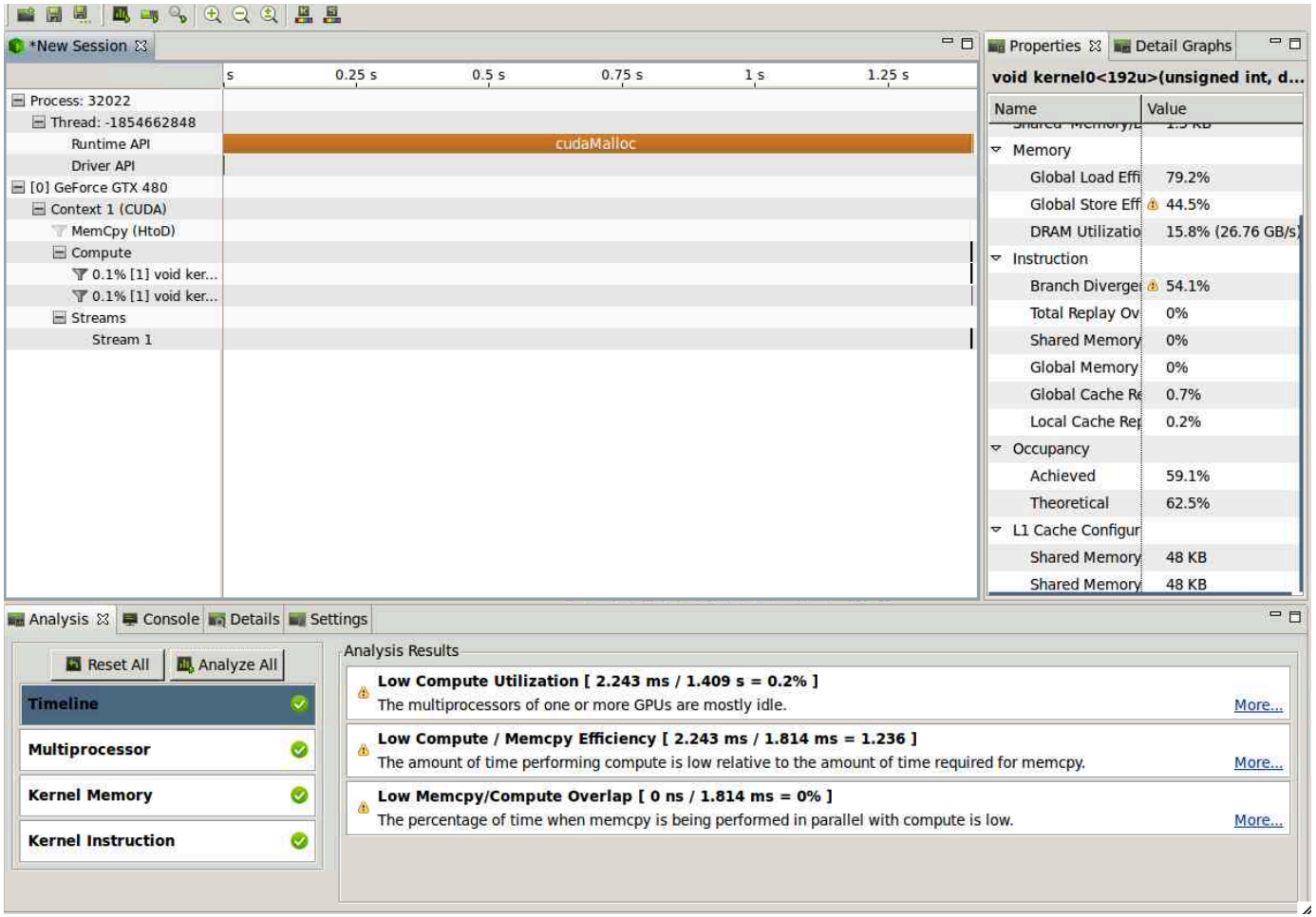


Kernel0

- Details:
  - blocksize = 32
  - Global Store Efficiency : 49.6%
  - Branch Divergence : 54.1 %
  - Occupancy: 16.2 %

- 1. Let's just increase the block size to help occupancy

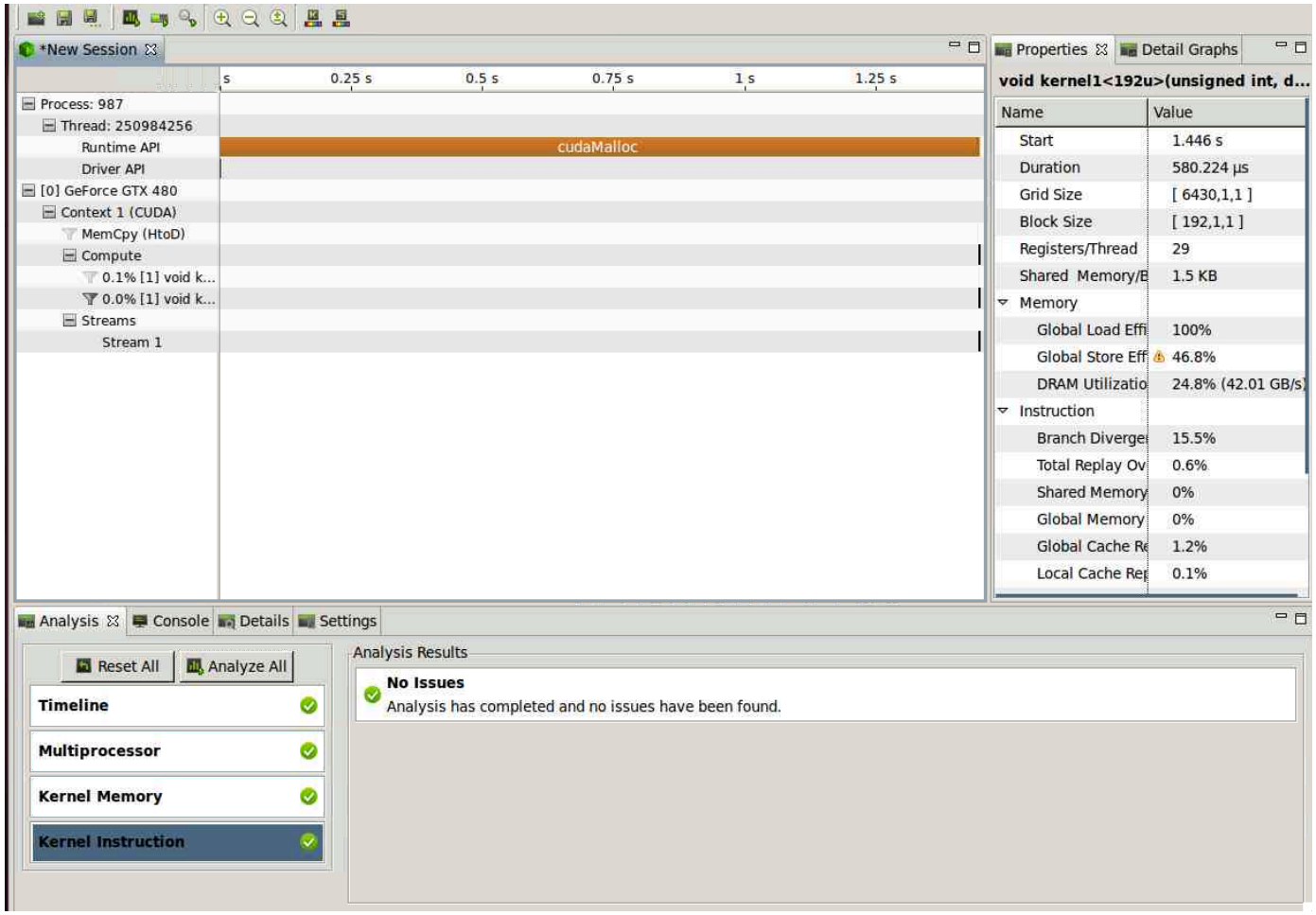
Kernel 0 - Profile



Kernel1

```
1.  template <uint BLOCK_SIZE>
2.  __global__ void kernel1(uint size, double *input, double *output) {
3.
4.    __shared__ double dontask[BLOCK_SIZE];
5.
6.    uint tid = threadIdx.x;
7.    uint blockstart = blockDim.x * blockIdx.x;
8.    uint ndx = tid + blockstart;
9.
10.   dontask[tid] = 5*tid + 2;
11.
12.   if (ndx < size) {
13.
14.     if (tid < BLOCK_SIZE/2) {
15.       ndx = 2*tid + blockstart;
16.       output[ndx] = sqrt(sin(input[ndx])+1) + input[ndx] / 7;
17.     } else {
18.       ndx = 2*(tid - BLOCK_SIZE/2)+1 + blockstart;
19.       output[ndx] = sqrt(cos(input[ndx])+1) + input[ndx] / 7;
20.     }
21.   }
22.
23. }
```

Kernel 1 - Profile



Improvements?

- Get rid of shared memory
- Larger block size to help with occupancy?

Kernel 2

```
1.  template <uint BLOCK_SIZE>
2.  __global__ void kernel2(uint size, double *input, double *output) {
3.
4.    uint tid = threadIdx.x;
5.    uint blockstart = blockDim.x * blockIdx.x;
6.    uint ndx = tid + blockstart;
7.
8.    if (ndx < size) {
9.
10.     if (tid < BLOCK_SIZE/2) {
11.       ndx = 2*tid + blockstart;
12.       output[ndx] = sqrt(sin(input[ndx])+1) + input[ndx] / 7;
13.     } else {
14.       ndx = 2*(tid - BLOCK_SIZE/2)+1 + blockstart;
15.       output[ndx] = sqrt(cos(input[ndx])+1) + input[ndx] / 7;
16.     }
17.   }
18.
19. }
```

Kernel 3

```
1.    template <uint BLOCK_SIZE>
2.    __global__ void kernel3(uint size, double *input, double *output) {
3.
4.      uint tid = threadIdx.x;
5.      uint blockstart = blockDim.x * blockIdx.x;
6.      uint ndx = tid + blockstart;
7.
8.      __shared__ double shared[BLOCK_SIZE];
9.
10.     if (ndx < size) {
11.       uint sid;
12.       ndx = tid + blockstart;
13.       shared[tid] = input[ndx]; // coalesced
14.
15.       __syncthreads();
16.
17.       if (tid < BLOCK_SIZE/2) {
18.         sid = 2*tid;
19.         shared[sid] = sqrt(sin(shared[sid])+1) + shared[sid] / 7;
20.       } else {
21.         sid = 2*tid - BLOCK_SIZE+1;
22.         shared[sid] = sqrt(cos(shared[sid])+1) + shared[sid] / 7;
23.       }
24.
25.       __syncthreads();
26.       output[blockstart+tid] = shared[tid];
27.     }
28.
29.   }
```

Debug flags

- What happens when debug flags are used?

Debug Flags

```
user@hostname:~/Desktop/cudaprofiler$ ./profileme
Let's look at some unoptimized kernels!
Launching Kernel kernel0
Kernel:kernel0:time is 0.00115008s
Launching Kernel kernel1
Kernel:kernel1:time is 0.0005864s
Launching Kernel kernel2
Kernel:kernel2:time is 0.000573696s
Launching Kernel kernel3
Kernel:kernel3:time is 0.000593696s
Launching Kernel kernel3
Kernel:kernel3:time is 0.000573408s
Launching Kernel kernel3
Kernel:kernel3:time is 0.000572384s
```

```
user@hostname:~/Desktop/cudaprofiler$ ./profileme
Let's look at some unoptimized kernels!
Launching Kernel kernel0
Kernel:kernel0:time is 0.00524582s
Launching Kernel kernel1
Kernel:kernel1:time is 0.00267126s
Launching Kernel kernel2
Kernel:kernel2:time is 0.00259254s
Launching Kernel kernel3
Kernel:kernel3:time is 0.00258394s
Launching Kernel kernel3
Kernel:kernel3:time is 0.0025992s
Launching Kernel kernel3
Kernel:kernel3:time is 0.00265152s
```

- Be careful when profiling against compile-optimized libraries!