Accelerators

Profiling, debugging and optimization

EURO

Leon Bogdanović

University of Ljubljana, FME, LECAD lab

GPU optimization

Optimization can be done on:

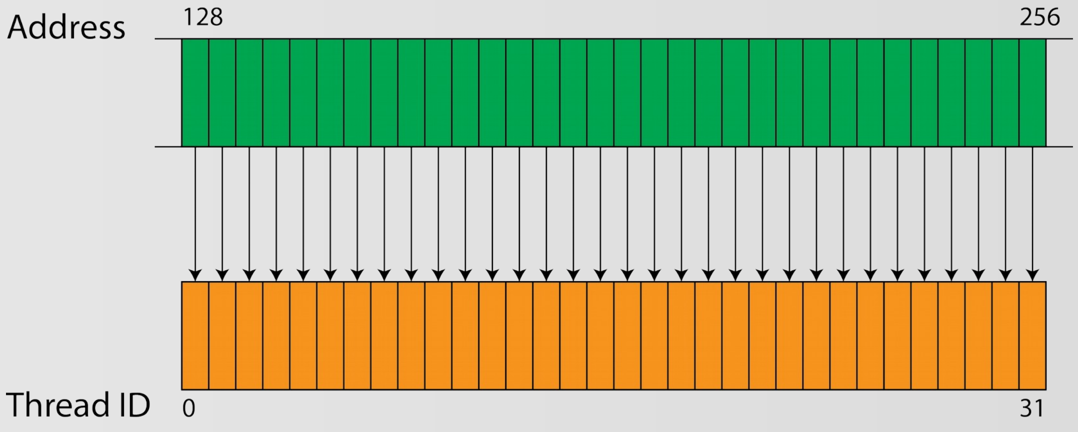
- **algorithms**
- **memory access** and **usage**
- **execution** configuration
- **instruction** performance

Algorithm optimization

Algorithms should be designed for:

- maximizing **independent parallelism**
- maximizing **arithmetic intensity** (computation/bandwidth)
- **minimizing data memory transfer** to/from host
- **minimizing memory caching**

Memory optimization



coalescing (source: cvw.cac.cornell.edu)

Memory access:

- ideally **coalesced** (combining multiple memory accesses into a single transaction)
- **avoid high-degree bank conflicts** in shared memory

Memory usage:

- **shared memory** (faster than global memory, threads can cooperate, for avoiding non-coalesced access)
- **effective bandwidth** of memory transfer

Execution optimization

- **multiprocessor occupancy:**
  - hardware must be kept busy
  - 100% occupancy: maximum number of warps of threads that can run concurrently
  - limited by resource usage (shared memory, registers)
- **blocks per multiprocessor** ratio:
  - all multiprocessors should have at least one block to execute
  - to keep multiprocessors busy multiple blocks should be executed on them
- **latency hiding:**
  - at least 192 threads (6 warps of 32 threads) per multiprocessor should be executed
  - limited by number of registers per kernel and amount of shared memory
- **threads per block**
  - should be a multiple of warp size (32 threads)
  - 64 threads per block (minimum), better choice: 192 or 256 threads per block

Instruction optimization

- **instruction throughput** dependent on:
  - nominal instruction throughput
  - memory latency and bandwidth
- **maximizing** usage of **high-bandwidth memory** by:
  - maximizing use of shared memory
  - minimizing accesses to global memory
  - maximizing coalescing of global memory accesses
- **overlapping memory accesses** with **computation:**
  - high ratio of computational operations to memory transactions
  - concurrency of many threads

Tools for performance evaluation

Tools capable of profiling and/or tracing CUDA and OpenCL codes:

- CUDA:
  - **nvprof**: command line profiling tool from CUDA toolkit
  - **nvvp**: visual profiler (GUI) tool from CUDA toolkit
  - nvprof and nvvp **deprecated** in CUDA 11: replaced by **Nsight Tools**
  - **TAU** (Tuning and Analysis Utilities): open source tool for profiling and tracing
  - other tools: **vampir**, **SCALASCA** (for large scale applications)...
- OpenCL:
  - on **NVIDIA cards**: OpenCL profiling **not supported** since CUDA 8
  - on **AMD cards**: OpenCL profiling with **Radeon GPU profiler**
  - **TAU** (Tuning and Analysis Utilities): open source tool for profiling and tracing
  - other tools: **vampir**, Intel **VTune Amplifier**...

Tools on HPCFS for GPU profiling/tracing

On HPCFS available:

- **nvprof, nvvp, TAU** for CUDA
- **TAU** for OpenCL
- **clone the repository from bitbucket** to your `viz.hpc.fs.uni-lj.si` account:

```
$ git clone https://bitbucket.org/lecad-peg/eurocc-accelerators.git
$ cd eurocc-accelerators/ex-3_riemann
```

- start the **environment with profiling tools** by following these steps on your `viz.hpc.fs.uni-lj.si` account:

```
$ module purge
$ module load tau/2.29.1-CUDA
$ module load jre
```

interactive session: **not recommended**

```
$ env --unset=LD_PRELOAD TMOUT=600 srun --time=1:0:0 --partition=gpu --x11 --pty bash -i
$ env --unset=LD_PRELOAD srun --partition=gpu nvprof ./riemann_cuda_double
```

running jobs: **recommended**

CUDA profiling with nvprof and nvvp

- **executables to profile**: riemann_cuda_double, riemann_cuda_double_reduce
- **compilation** is done with:

```
$ nvcc -o riemann_cuda_double riemann_cuda_double.cu
$ nvcc -o riemann_cuda_double_reduce riemann_cuda_double_reduce.cu
```

- **profiling** is done with:

```
$ env --unset=LD_PRELOAD srun --partition=gpu nvprof ./riemann_cuda_double
$ env --unset=LD_PRELOAD srun --partition=gpu --x11 nvvp ./riemann_cuda_double
```

(not recommended)

- `nvprof` **available options and query metrics**:

```
$ env --unset=LD_PRELOAD srun --partition=gpu nvprof -h
$ env --unset=LD_PRELOAD srun --partition=gpu nvprof --query-metrics
```

CUDA profiling on login node with nvvp

- CUDA profiles visualization with nvvp is recommended on the login node
- **create profile** with nvprof:

```
$ env --unset=LD_PRELOAD srun --partition=gpu nvprof -s -o \
riemann_cuda_double.nvprof ./riemann_cuda_double
```

- **start** nvvp (on the login node):

```
$ nvvp
```

- **select the created profile** (`riemann_cuda_double.nvprof`) and visualize it with:

```
File -> Import -> Nvprof (Select an import source) -> Multiple processes ->
Browse... -> select "riemann_cuda_double.nvprof" -> OK -> Finish
```

Example 1: CUDA profiling - riemann_cuda_double

outputs of **nvprof**:

```
$ env --unset=LD_PRELOAD srun --partition=gpu nvprof ./riemann_cuda_double
```

```
==40078== NVPROF is profiling process 40078, command: ./riemann_cuda_double
Found GPU 'Tesla K80' with 11.173 GB of global memory, max 1024 threads per block, and 13 multiprocessors
CUDA kernel 'medianTrapezoid' launch with 976563 blocks of 1024 threads
Riemann sum CUDA (double precision) for N = 1000000000    : 0.34134474606857 29
Total time (measured by CPU)                              : 6.320000 s
==40078== Profiling application: ./riemann_cuda_double
==40078== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.43%  2.93369s         1  2.93369s  2.93369s  2.93369s  [CUDA memcpy DtoH]
                    5.57%  173.20ms         1  173.20ms  173.20ms  173.20ms  medianTrapezoid(double*, int)
      API calls:   93.70%  3.10713s         1  3.10713s  3.10713s  3.10713s  cudaMemcpy
                    5.64%  187.00ms         1  187.00ms  187.00ms  187.00ms  cudaMalloc
                    0.22%  7.2943ms         1  7.2943ms  7.2943ms  7.2943ms  cudaFree
                    0.20%  6.5305ms       582  11.220us     308ns  425.41us  cuDeviceGetAttribute
                    0.18%  6.0319ms         6  1.0053ms  1.0032ms  1.0135ms  cuDeviceTotalMem
                    0.03%  1.0710ms         1  1.0710ms  1.0710ms  1.0710ms  cudaGetDeviceProperties
                    0.02%  526.71us         6  87.784us  85.138us  98.957us  cuDeviceGetName
                    0.01%  352.23us         1  352.23us  352.23us  352.23us  cudaLaunchKernel
                    0.00%  28.732us         6  4.7880us  2.7430us  11.986us  cuDeviceGetPCIBusId
                    0.00%  10.917us         1  10.917us  10.917us  10.917us  cudaSetDevice
                    0.00%  8.4950us        12     707ns     372ns  1.3430us  cuDeviceGet
                    0.00%  3.0550us         3  1.0180us     323ns  1.7320us  cuDeviceGetCount
                    0.00%  2.5450us         6     424ns     354ns     563ns  cuDeviceGetUuid
```

Example 1: CUDA profiling – riemann_cuda_double (cont.)

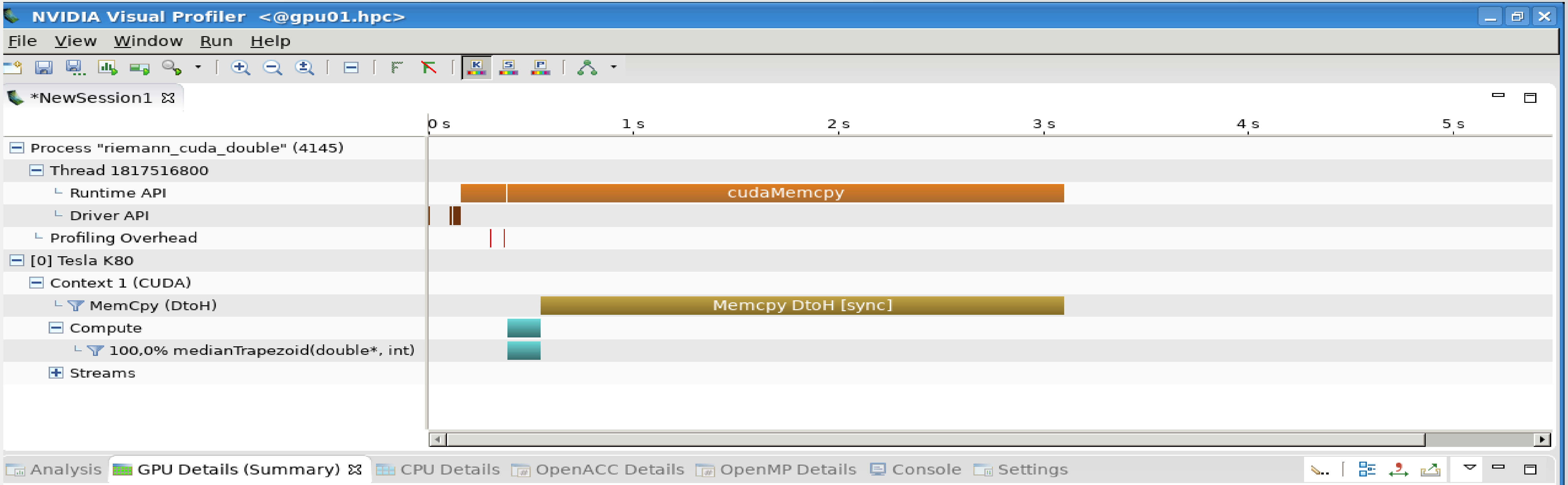
outputs of **nvprof**:

```
$ env --unset=LD_PRELOAD srun --partition=gpu nvprof --metrics flop_count_dp \
--metrics dram_read_throughput --metrics dram_write_throughput –metrics \
achieved_occupancy ./riemann_cuda_double
```

```
==40178== NVPROF is profiling process 40178, command: ./riemann_cuda_double
Found GPU 'Tesla K80' with 11.173 GB of global memory, max 1024 threads per block, and 13 multiprocessors
CUDA kernel 'medianTrapezoid' launch with 976563 blocks of 1024 threads
==40178== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "medianTrapezoid(double*, int)" (done)
Riemann sum CUDA (double precision) for N = 1000000000    : 0.34134474606857 29
Total time (measured by CPU)                              : 31.340000 s
==40178== Profiling application: ./riemann_cuda_double
==40178== Profiling result:
==40178== Metric result:
Invocations                          Metric Name                            Metric Description         Min         Max         Avg
Device "Tesla K80 (0)"
    Kernel: medianTrapezoid(double*, int)
          1                        flop_count_dp   Floating Point Operations(Double Precision)  8.6000e+10  8.6000e+10  8.6000e+10
          1                 dram_read_throughput               Device Memory Read Throughput  5.7400MB/s  5.7400MB/s  5.7400MB/s
          1                dram_write_throughput              Device Memory Write Throughput  56.288GB/s  56.288GB/s  56.288GB/s
```

Example 1: CUDA profiling – riemann_cuda_double (cont.)

profiles/traces with **nvvp**:



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	Floating Point Operations(Double Precision)	Achieved Occupancy
medianTrapezoid(double*, int)	1	162,37872 ms	26	0	0	86000005646	0,911

Example 1: CUDA profiling – riemann_cuda_double (cont.)

Analysis of profiles:

- **bottleneck**: memory **transfer from device to host** (Memcpy DtoH)
- **multiprocessor occupancy**: 91.1% (`medianTrapezoid`)
- **device memory read** throughput: 5.8172 MB/s (`medianTrapezoid`)
- **device memory write** throughput: 56.239 GB/s (`medianTrapezoid`)
- **FLOPS** for `medianTrapezoid`: 86000005646/162.37872*1000/10^9 = 529.63 GFLOPS

Possible optimizations:

- **reducing** memory **transfer from device to host**
- **increasing** device **memory throughput**: 240.6 GB/s (theoretical memory bandwidth of Tesla K80)
- **Increasing** kernel **throughput**: 1371 GFLOPS (theoretical FP64 (double) performance of Tesla K80)

Exercise 1: CUDA profiling – riemann_cuda_double_reduce

Analyze the `riemann_cuda_double_reduce` executable with `nvprof` and `nvvp`:

- **determine** execution times, FLOPS, multiprocessor occupancy and read/write memory throughputs **for both kernels** (from nvprof output)
- **determine memory transfers** times to/from device (from nvprof output) and identify possible bottlenecks
- **visualize the traces** with nvvp: how are kernels deployed in the default stream?
- on the basis of the analysis **suggest any possible optimizations**

Solution (Exer. 1):
CUDA profiling – riemann_cuda_double_reduce

outputs of **nvprof**:

```
==40407== NVPROF is profiling process 40407, command: ./riemann_cuda_double_reduce
Found GPU 'Tesla K80' with 11.173 GB of global memory, max 1024 threads per block, and 13 multiprocessors
CUDA kernel 'medianTrapezoid' launch with 976563 blocks of 1024 threads
CUDA kernel 'reducerSum' launch with 1 blocks of 1024 threads

Riemann sum CUDA (double precision) for N = 1000000000    : 0.34134474606854243
Total time (measured by CPU)                              : 2.040000 s
==40407== Profiling application: ./riemann_cuda_double_reduce
==40407== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.53%  672.68ms         1  672.68ms  672.68ms  672.68ms  reducerSum(double*, double*, int, int)
                   20.47%  173.17ms         1  173.17ms  173.17ms  173.17ms  medianTrapezoid(double*, int)
                    0.00%  6.2400us         1  6.2400us  6.2400us  6.2400us  [CUDA memcpy DtoH]
      API calls:   49.24%  1.01211s         3  337.37ms  7.1700us  1.00526s  cudaFree
                   41.16%  845.92ms         1  845.92ms  845.92ms  845.92ms  cudaMemcpy
                    8.92%  183.34ms         2  91.670ms  491.18us  182.85ms  cudaMalloc
                    0.31%  6.2819ms       582  10.793us     287ns  412.87us  cuDeviceGetAttribute
                    0.28%  5.6530ms         6  942.16us  935.95us  950.47us  cuDeviceTotalMem
                    0.05%  1.0165ms         1  1.0165ms  1.0165ms  1.0165ms  cudaGetDeviceProperties
                    0.03%  709.07us         6  118.18us  81.701us  276.88us  cuDeviceGetName
                    0.01%  224.59us         2  112.29us  17.374us  207.22us  cudaLaunchKernel
                    0.00%  28.150us         6  4.6910us  2.6080us  12.404us  cuDeviceGetPCIBusId
                    0.00%  11.132us         1  11.132us  11.132us  11.132us  cudaSetDevice
                    0.00%  7.7420us        12     645ns     322ns  1.3770us  cuDeviceGet
                    0.00%  3.1880us         3  1.0620us     337ns  1.9930us  cuDeviceGetCount
                    0.00%  2.3140us         6     385ns     330ns     515ns  cuDeviceGetUuid
```

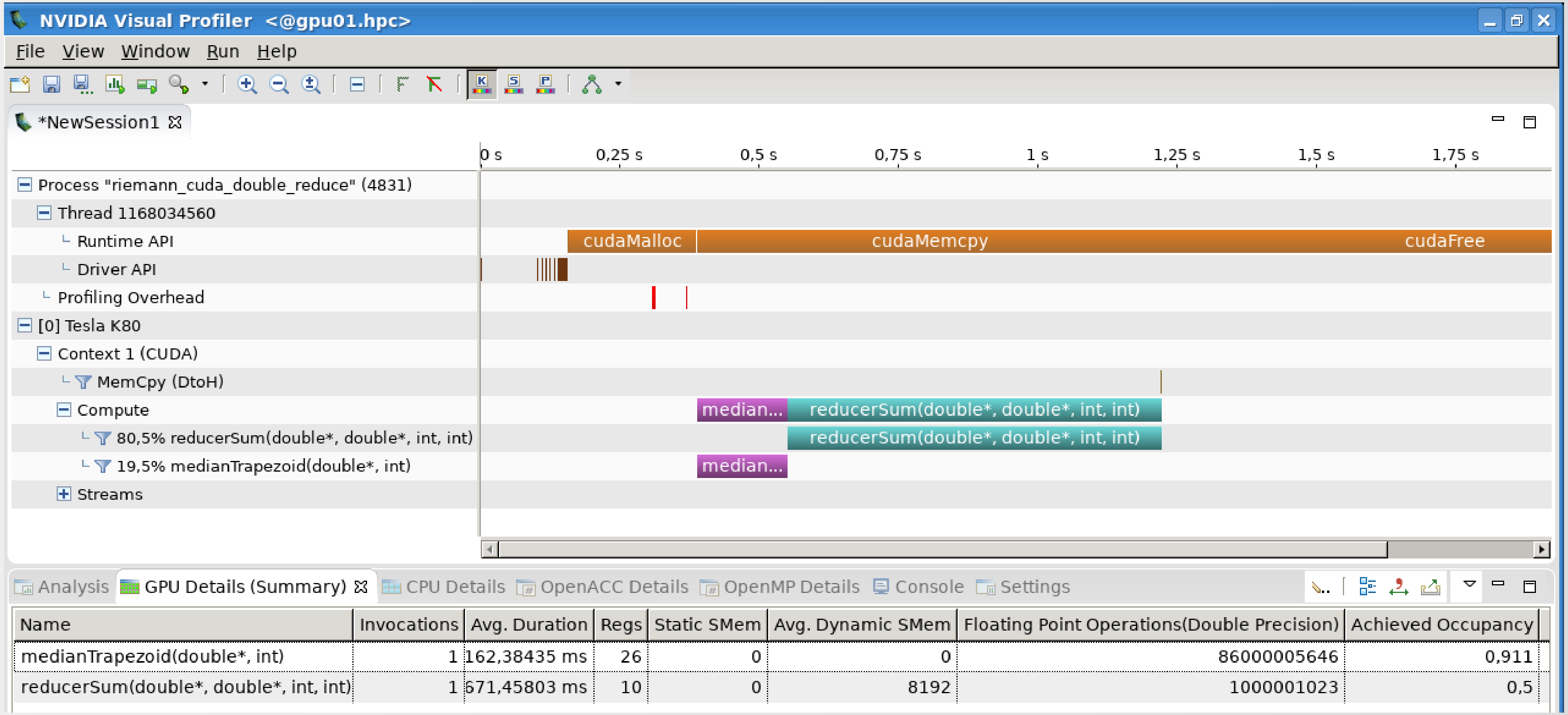
Solution (Exer. 1):
CUDA profiling – riemann_cuda_double_reduce (cont.)

outputs of **nvprof**:

```
==40131== NVPROF is profiling process 40131, command: ./riemann_cuda_double_reduce
Found GPU 'Tesla K80' with 11.173 GB of global memory, max 1024 threads per block, and 13 multiprocessors
CUDA kernel 'medianTrapezoid' launch with 976563 blocks of 1024 threads
==40131== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "medianTrapezoid(double*, int)" (3 of 3)...
        fb_subp1_read_sectors
Replaying kernel "medianTrapezoid(double*, int)" (done)
Replaying kernel "reducerSum(double*, double*, int, int)" (done)
Riemann sum CUDA (double precision) for N = 1000000000    : 0.34134474606854243
Total time (measured by CPU)                              : 35.070000 s
==40131== Profiling application: ./riemann_cuda_double_reduce
==40131== Profiling result:
==40131== Metric result:
Invocations                               Metric Name                            Metric Description         Min         Max         Avg
Device "Tesla K80 (0)"
    Kernel: medianTrapezoid(double*, int)
          1                             flop_count_dp   Floating Point Operations(Double Precision)  8.6000e+10  8.6000e+10  8.6000e+10
          1                      dram_read_throughput                 Device Memory Read Throughput  4.6853MB/s  4.6853MB/s  4.6853MB/s
          1                     dram_write_throughput                Device Memory Write Throughput  56.183GB/s  56.183GB/s  56.183GB/s
    Kernel: reducerSum(double*, double*, int, int)
          1                             flop_count_dp   Floating Point Operations(Double Precision)  1000001023  1000001023  1000001023
          1                      dram_read_throughput                 Device Memory Read Throughput  13.820GB/s  13.820GB/s  13.820GB/s
          1                     dram_write_throughput                Device Memory Write Throughput  807.000B/s  807.000B/s  806.000B/s
```

Solution (Exer. 1):
CUDA profiling – riemann_cuda_double_reduce (cont.)

profiles/traces with **nvvp**:



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	Floating Point Operations(Double Precision)	Achieved Occupancy
medianTrapezoid(double*, int)	1	162,38435 ms	26	0	0	86000005646	0,911
reducerSum(double*, double*, int, int)	1	671,45803 ms	10	0	8192	1000001023	0,5

Solution (Exer. 1):

CUDA profiling – riemann_cuda_double_reduce (cont.)

Analysis of profiles:

- **minimal** (just one double float) memory **transfer from device to host** (Memcpy DtoH)
- **multiprocessor occupancy**: 91.1% (`medianTrapezoid`), 50% (`reducerSum`)
- **device memory read** throughput: 5.766 MB/s (`medianTrapezoid`), 13.866 GB/s (`reducerSum`)
- **device memory write** throughput: 56.231 GB/s (`medianTrapezoid`), 524.000 B/s (`reducerSum`)
- **FLOPS** for `medianTrapezoid`: 86000005646/162.37872*1000/10^9 = 529.63 GFLOPS
- **FLOPS** for `reducerSum`: 1000001023/672.17*1000/10^9 = 1.488 GFLOPS

Possible further optimizations:

- **sum reduce** kernel with **many blocks of threads** instead of 1 block for achieving better device memory and kernel throughput
- **combining both kernels into one kernel** for achieving **better overall** device memory and kernel **throughput**

Example 2: CUDA profiling with TAU

- **executables to profile**: riemann_cuda_double_reduce
- **profiling** is done with:

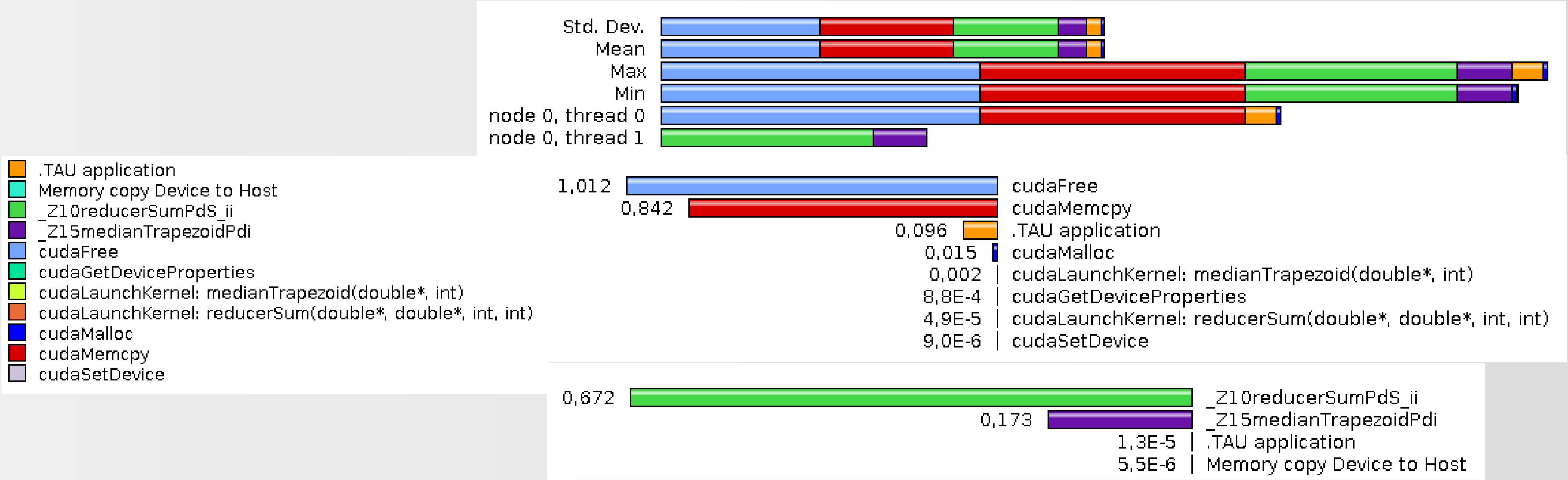
```
$ env --unset=LD_PRELOAD srun --partition=gpu tau_exec -T serial -cupti \
./riemann_cuda_double_reduce
$ pprof
$ paraprof
```

- **tracing** is done with:

```
$ env --unset=LD_PRELOAD srun --partition=gpu env TAU_TRACE=1 tau_exec -T \
serial -cupti ./riemann_cuda_double_reduce
$ tau_treemerge.pl
$ tau2slog2 tau.trc tau.edf -o tau.slog2
$ jumpshot tau.slog2
```

Example 2: CUDA profiling with TAU (cont.)

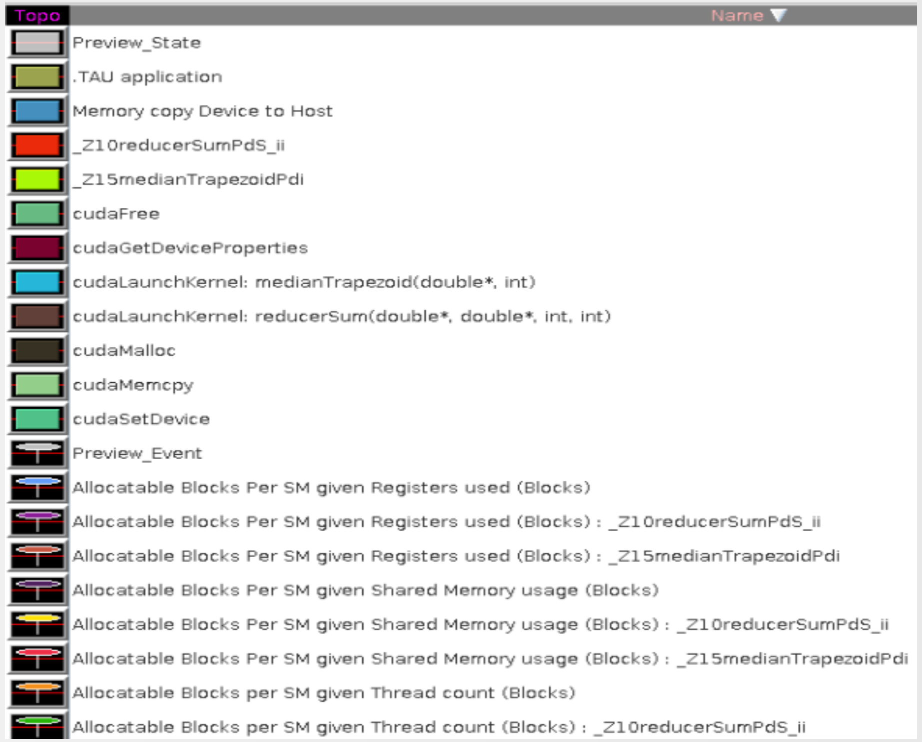
profiles with **paraprof**:

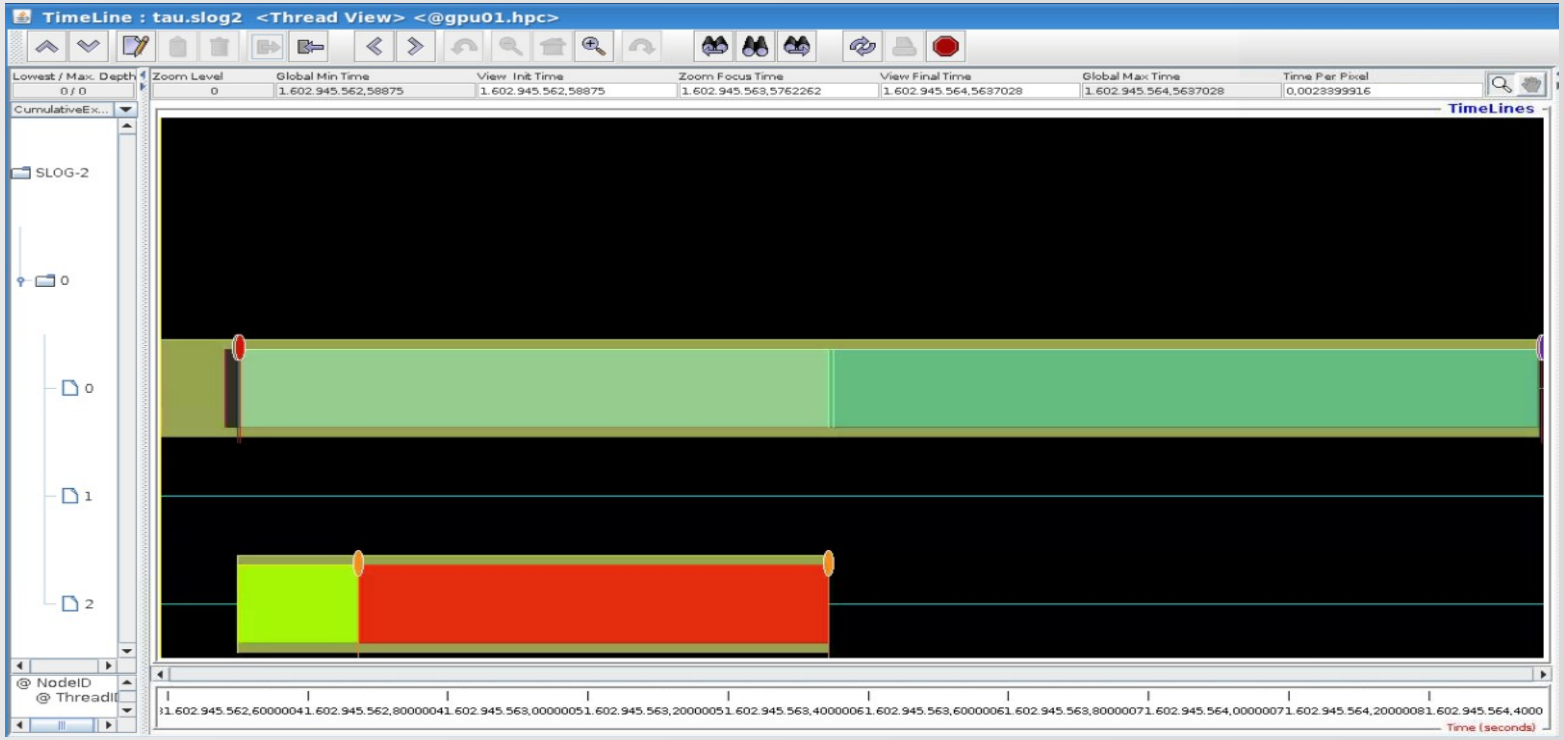


Example 2: CUDA profiling with TAU (cont.)

SLING EURO

traces with jumpshot:





Exercise 2: OpenCL profiling with TAU

- **analyze** the `riemann_opencl_double_reduce` executable with TAU
- **compilation** is done with:

```
$ gcc -o riemann_opencl_double_reduce riemann_opencl_double_reduce.c -lOpenCL
```

- **generate** profiles with:

```
$ env --unset=LD_PRELOAD srun --partition=gpu tau_exec -T serial -opencl \
./riemann_opencl_double_reduce
```

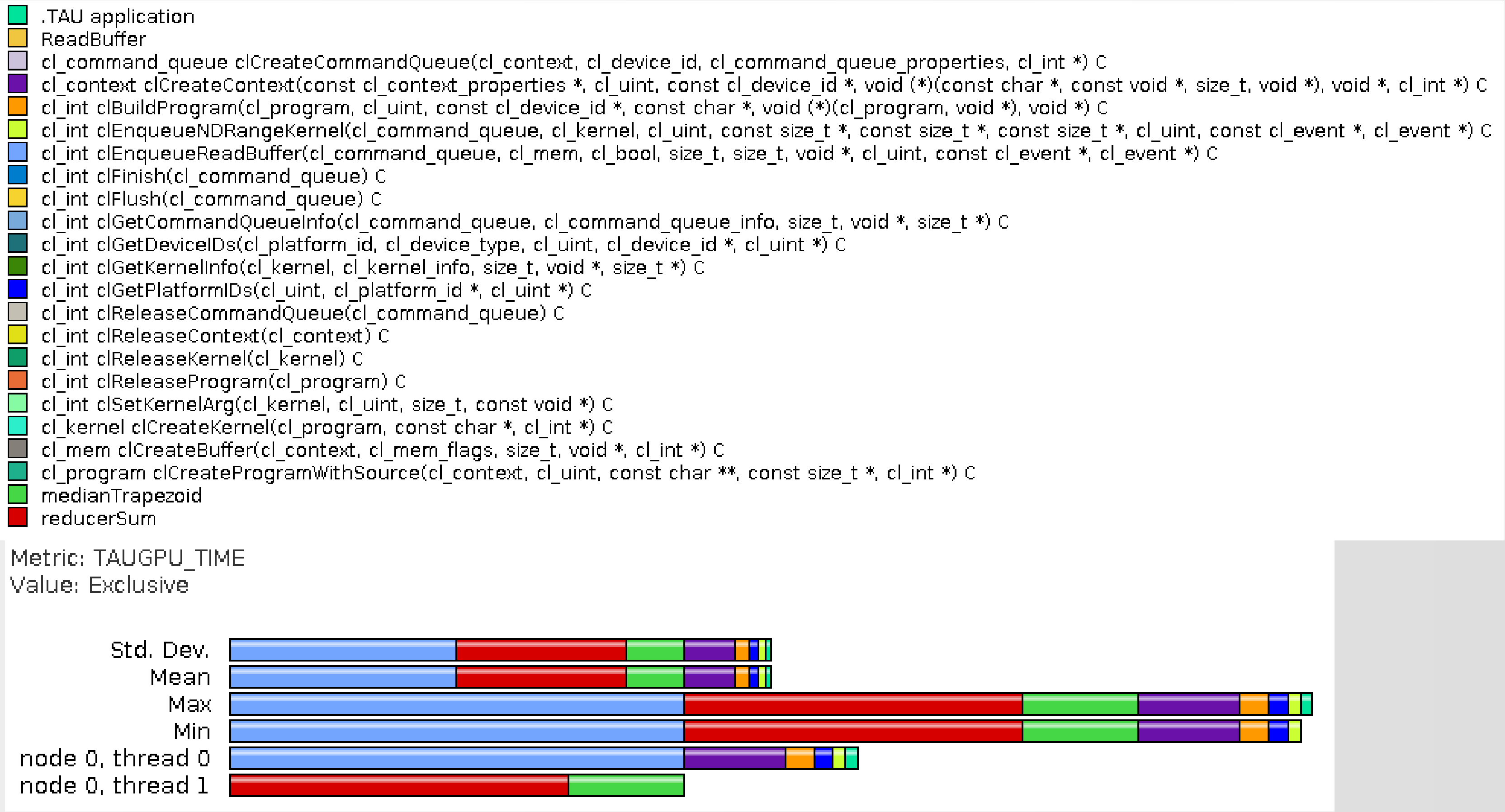
- **use** pprof and paraprof for profiling
- **generate** traces with:

```
$ env --unset=LD_PRELOAD srun --partition=gpu env TAU_TRACE=1 tau_exec -T serial \
-opencl ./riemann_opencl_double_reduce
```

- **use** jumpshot for visualizing traces

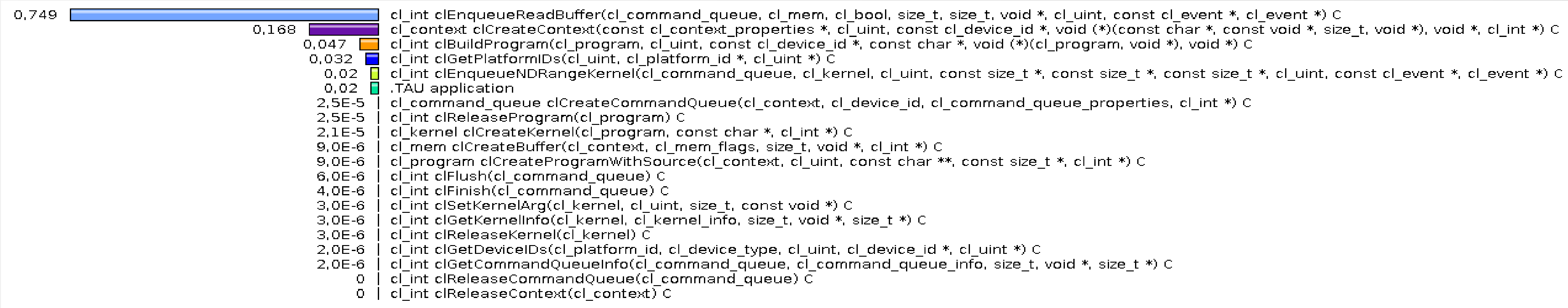
Solution (Exer. 2):
OpenCL profiling – riemann_opencl_double_reduce

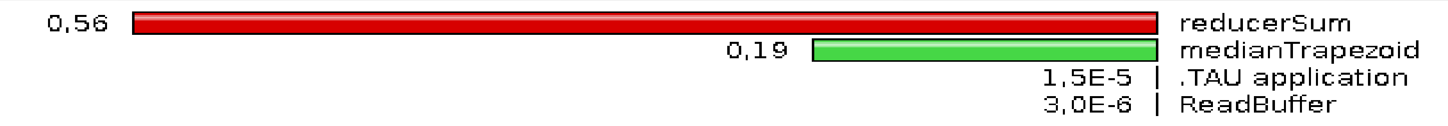
profiles with paraprof:



Solution (Exer. 2):
OpenCL profiling – riemann_opencl_double_reduce (cont.)

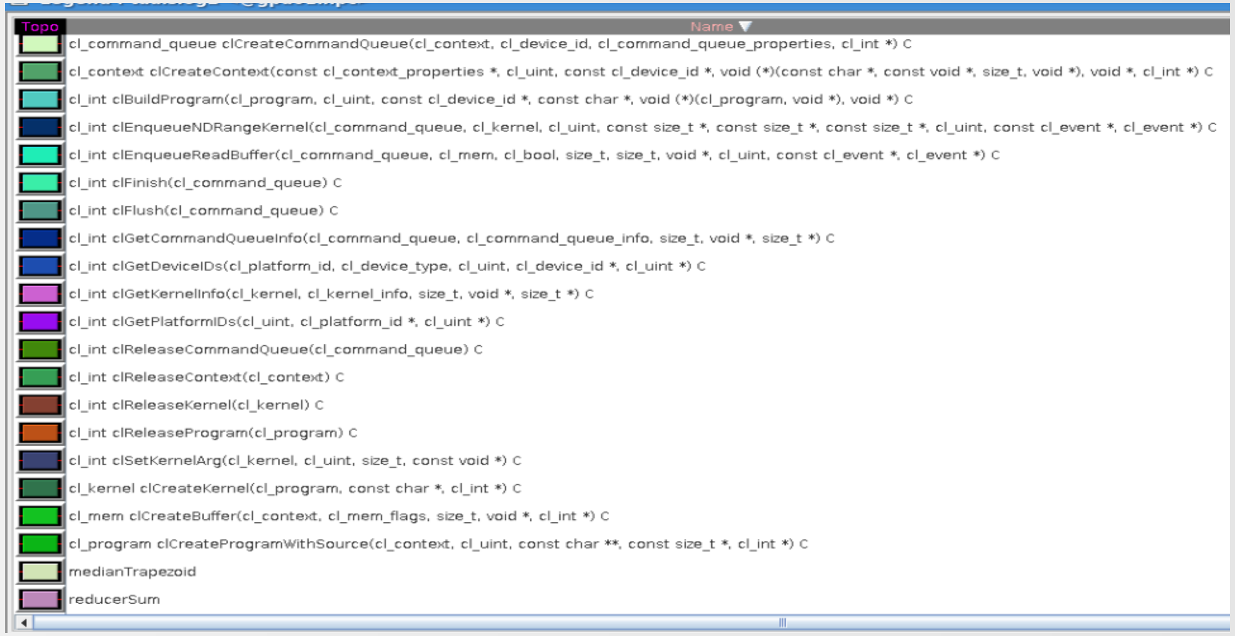
profiles with paraprof:

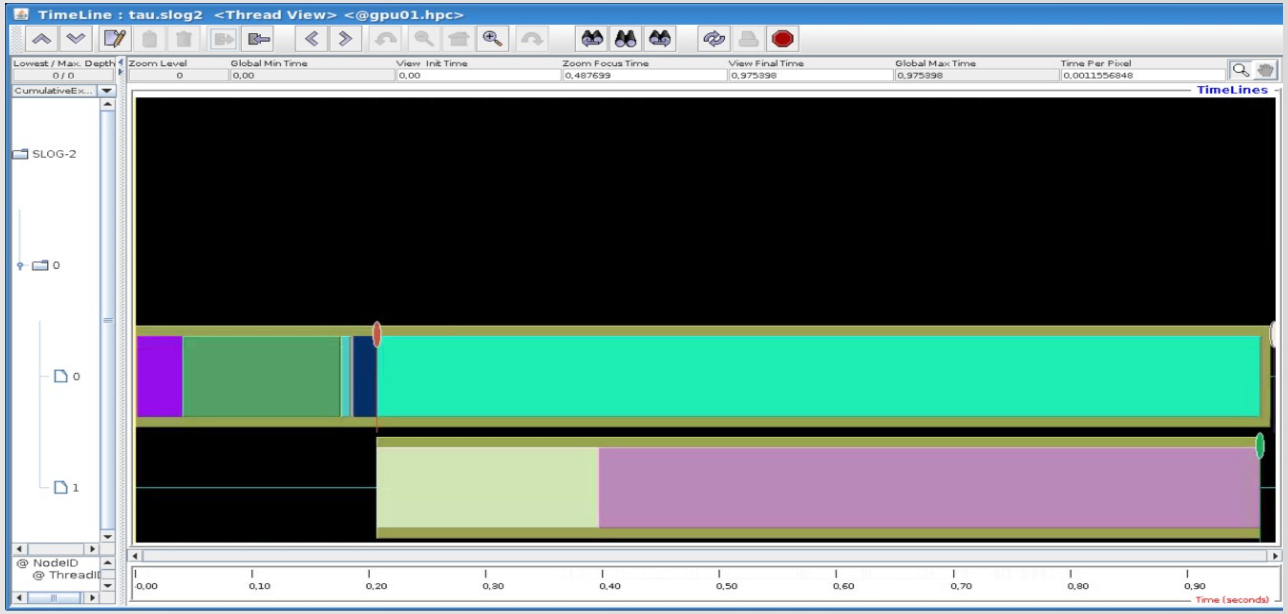




Solution (Exer. 2):
OpenCL profiling – riemann_opencl_double_reduce (cont.)

traces with jumpshot:





Multiple GPUs in CUDA

In a **multiple GPU** set-up:

- all CUDA API calls are issued into a current GPU
- `cudaSetDevice(ID)`: for changing the current GPU to GPU with id ID
- GPU IDs always in range [0, number of GPUs), GPUs count can be obtained with `cudaGetDeviceCount()` or by invoking `nvidia-smi`
- **kernel calls and asynchronous memory copying** functions are **in principle non-blocking** towards CPU thread execution and therefore towards switching GPUs

Example 3: Domain decomposition on multiple GPUs

```
__global__ void medianTrapezoid(double *a, int n, int dev)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  double x = (double)(idx + n * dev) / (double)(2 * n);

  if(idx < n)
    a[idx] = (exp(-x * x / 2.0) +
              exp(-(x + 1 / (double)(2 * n)) *
              (x + 1 / (double)(2 * n)) / 2.0)) / 2.0;
}
```

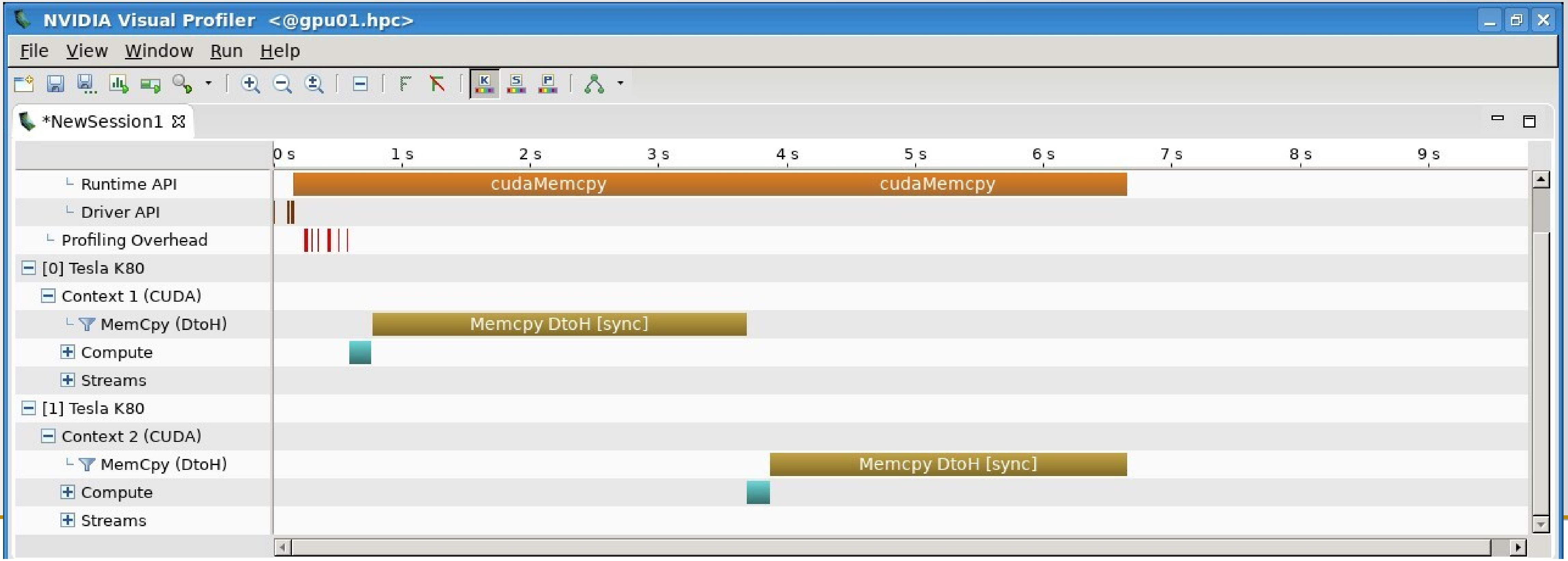
Domain composition for numerical integration in CUDA on **two GPUs**:

- the kernel `medianTrapezoid` is modified for calculations on two sub-domains (dev = 0, i.e. GPU with ID 0 calculates on the first sub-interval of the integrating domain, dev = 1, i.e. GPU with ID 1 calculates on the second sub-interval of the integrating domain)
- both GPUs return the array with trapezoid medians calculated on the specific sub-interval of the integrating domain

Example 3: Domain decomposition on multiple GPUs (cont.)

Code `riemann_cuda_double_multiple` (in `eurocc-accelerators/multiple_gpus`):

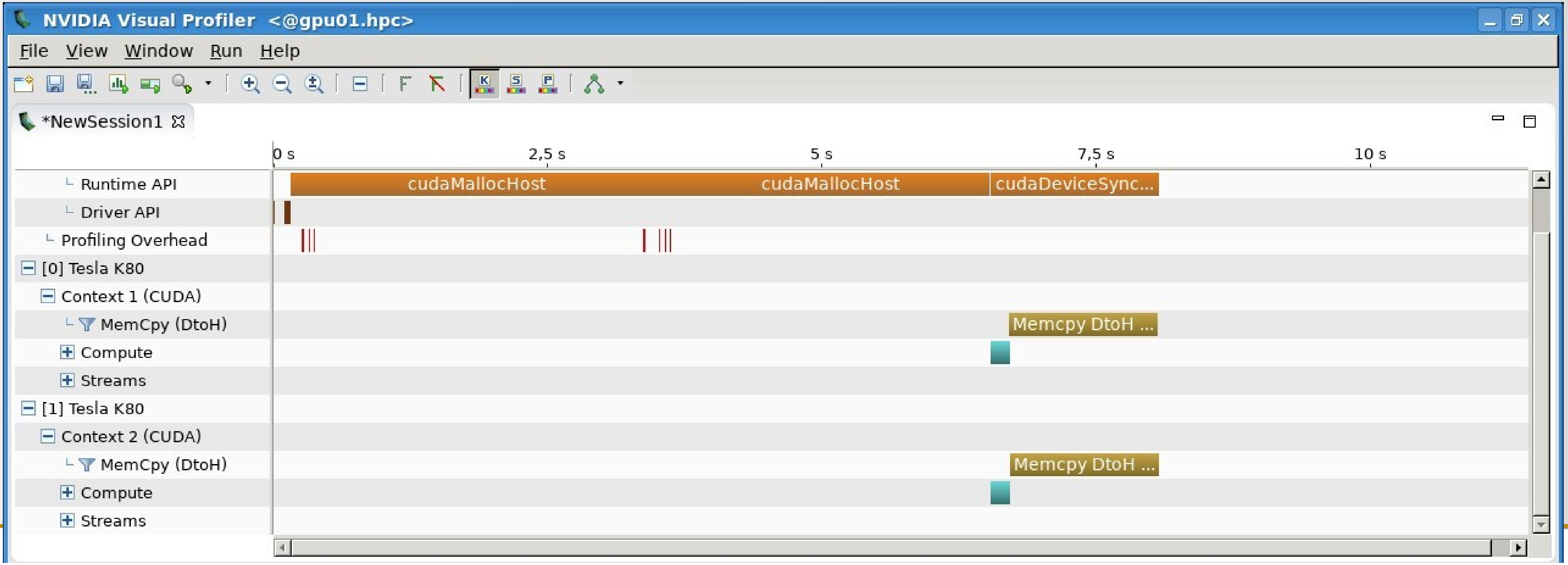
- uses **normal host memory allocation** with `malloc()` and **synchronous memory transfer** from device to host with `cudaMemcpy()`
- **concurrency** of `medianTrapezoid` kernel executions on GPUs is **NOT achieved**



Example 3: Domain decomposition on multiple GPUs (cont.)

Code `riemann_cuda_double_multiple_concurrency`:

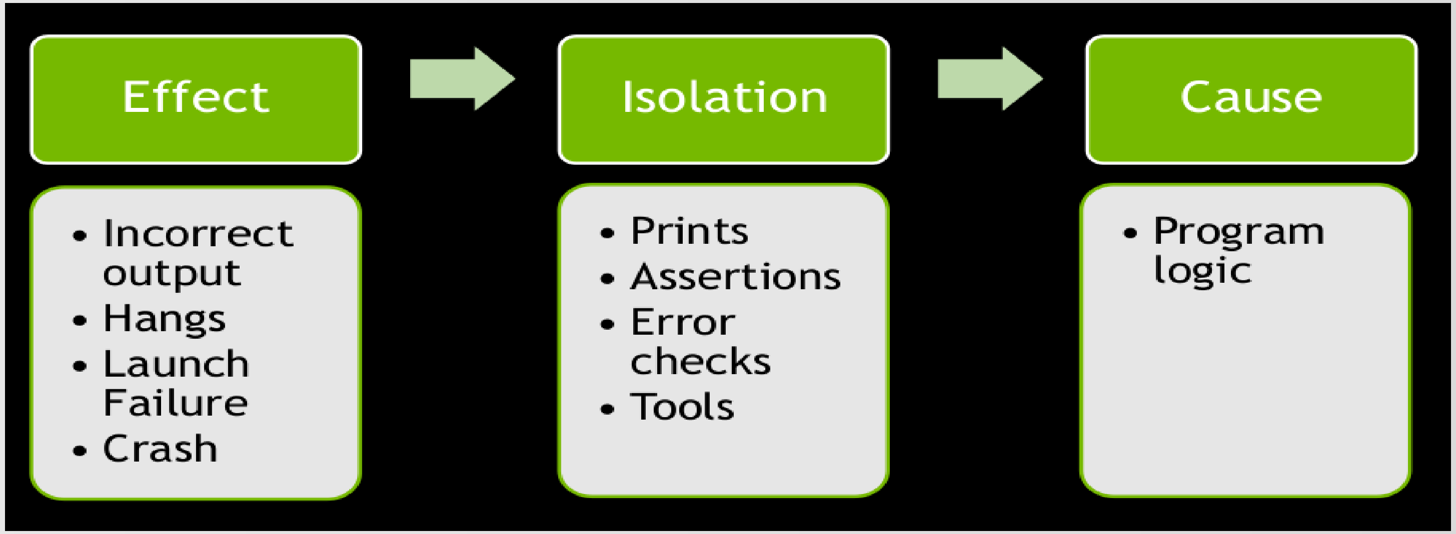
- uses **pinned host memory allocation** with `cudaMallocHost()` and **asynchronous memory transfer** from device to host with `cudaMemcpyAsync()`
- **concurrency** of `medianTrapezoid` kernel executions on GPUs **IS achieved**



CUDA / OpenCL debugging

- **Debugging** possibilities:

- **in-program** checking:
  - use of printf() in device code
  - use of assert() in device code
  - data checks
  - CUDA/OpenCL API call checks



Debugging applications (source: V. Venkataraman)

- **tools for debugging**:
  - for CUDA: CUDA-MEMCHECK, CUDA-GDB (command-line or GUI in NVIDIA Nsight Eclipse Edition)...
  - for OpenCL: Oclgrind, GDB...

CUDA API call checking

- **checking errors** in runtime API code **through** an **assert style handler function** and **wrapper macro**:

```
#define gpuErrchk(ans) {gpuAssert((ans), __FILE__, __LINE__);}
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}
```

- return **status** of the **API call**:

```
gpuErrchk(cudaMalloc(&a_d, size * sizeof(float)));
```

- checking **errors in kernel launch**:

```
kernel<<<1,1>>>(a);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
```

Example 4: CUDA API call checking

- **checking for errors** when executing the compiled `hello_cuda.cu` example
- the code is compiled with kernel launch `<<<16, 0>>>`:

```
#define NUM_BLOCKS 16
#define BLOCK_WIDTH 1
...
hello<<<NUM_BLOCKS, BLOCK_WIDTH-1>>>();
```

- output of executing the program **without** using the **checking errors wrapper macro**:

```
$ ./hello_cuda
That's all!
```

- output of executing the program **with** using the **checking errors wrapper macro**:

```
$ ./hello_cuda
GPUassert: invalid configuration argument hello.cu 27
```

Exercise 3: CUDA-MEMCHECK

- **check** the compiled `hello_cuda.cu` (from Example 3) with cuda-memcheck
- **launch** the utility from command-line with:

```
$ cuda-memcheck ./hello_cuda
```

- **compare** the output with the output from CUDA API call checking

OpenCL API call checking

- **checking errors** in runtime API code **through** a **handler function**:

```
void checkErrors(cl_int status, char *label, int line)
{
  switch (status)
  {
      case CL_SUCCESS:
        return;
      case CL_BUILD_PROGRAM_FAILURE:
        fprintf(stderr, "OpenCL error (at %s, line %d): CL_BUILD_PROGRAM_FAILURE\n", label, line);
        break;
…
case CL_PROFILING_INFO_NOT_AVAILABLE:
        fprintf(stderr, "OpenCL error (at %s, line %d): CL_PROFILING_INFO_NOT_AVAILABLE\n", label, line);
        break;
  }
  exit(status);
}
```

- return **status** of the **API call** example:

```
ret = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &globalItemSize,
                             &localItemSize, 0, NULL, NULL);
checkErrors (ret, "clEnqueueNDRangeKernel", __LINE__);
```

Example 5: OpenCL API call checking

- **checking for errors** when executing the compiled `hello_opencl.c` example
- the code is compiled with kernel launch `(..., 16, 3, ...)`:

```
size_t globalItemSize = 16;
size_t localItemSize = 3;
ret = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &globalItemSize,
                             &localItemSize, 0, NULL, NULL);
```

- output of executing the program **without** using the **checking errors handler function**:

```
$ ./hello_opencl
That's all!
```

- output of executing the program **with** using the **checking errors handler function**:

```
$ ./hello_opencl
-OpenCL error (at clEnqueueNDRangeKernel, line 202): CL_INVALID_WORK_GROUP_SIZE
```

Thanks!

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro