CUDA Profiling and Debugging

30.03.22

Available Tools

- Debugging
  - cuda-gdb
    - Command line tool
  - cuda-memcheck
    - Command line tool
  - Nsight
    - Integrates with Visual Studio and Eclipse
- Profiling
  - nvprof
    - Command line tool
  - nvvp
    - Visual profiler
  - Nsight Compute
    - For the GPU side
  - Nsight Systems
    - For the CPU side
  - CUPTI
    - API for programmatically collecting performance measures

Compute Sanitizer

- A functional correctness checking suite with the following tools
  - Memcheck
  - Racecheck
  - Initcheck
  - Synccheck

CUDA-MEMCHECK

- Does not need any special compilation flags
  - But output sometimes clearer with **`-G`** or **`-lineinfo`**
- Reports
  - Memory access errors
    - Misaligned access & out of bounds
  - Hardware exception
  - Malloc/free errors
  - CUDA API errors
  - Memory leaks
    - Both cudaMalloc & device heap

```
cuda-memcheck [options] app_name [app_options]
```

```
compute-sanitizer --tool memcheck [options] app_name [app_options]
```

CUDA-MEMCHECK

```
__global__ void test_memcheck(const int* in, int* out, int num_elements) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    out[tid] = in[tid];
}

// on host
int* gpu_in = nullptr, *gpu_out = nullptr;
cudaMalloc(&gpu_in, sizeof(int) * num_elements));
cudaMalloc(&gpu_out, sizeof(int) * num_elements));
test_memcheck<<<(num_elements / 256) + 1, 256>>>(gpu_in, gpu_out, num_elements);
```

```
clubber:gpuprog > cuda-memcheck ./test-memcheck
========= CUDA-MEMCHECK
========= Invalid __global__ write of size 4
=========     at 0x000002d0 in memcheck.cu:3:test_memcheck(int const *, int*, int)
=========     by thread (255,0,0) in block (128,0,0)
=========     Address 0x703b003fc is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/lib/x86_64-linux-gnu/libcuda.so.1 [0x253eba]
=========     Host Frame:./test-memcheck [0xc8cb]
=========     Host Frame:./test-memcheck [0x5fff0]
=========     Host Frame:./test-memcheck [0x8280]
```

CUDA Initcheck

- Tests uninitialized device memory accesses
  - Only supports global memory as of now

```
__global__ void test_initcheck(const int* in, int* out, int num_elements) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_elements)
        return;

    if (tid == 0)
        out[tid] = in[tid];
    else
        out[tid] = tid;
}

// on host
int* gpu_in = nullptr, *gpu_out = nullptr ;
cudaMalloc(&gpu_in, sizeof(int) * num_elements));
cudaMalloc(&gpu_out, sizeof(int) * num_elements));
test_initcheck<<<(num_elements / 256) + 1, 256>>>(gpu_in, gpu_out, num_elements);
```

CUDA Initcheck

```
clubber:gpuprog > cuda-memcheck --tool initcheck ./test-initcheck
========= CUDA-MEMCHECK
========= Uninitialized __global__ memory read of size 4
=========     at 0x00000268 in initcheck.cu:7:test_initcheck(int const *, int*, int)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x703ac0000
=========     Saved host backtrace up to driver entry point
=========     Host Frame:/lib/x86_64-linux-gnu/libcuda.so.1 [0x253eba]
=========     Host Frame:./test-initcheck [0xc8cb]
=========     Host Frame:./test-initcheck [0x5fff0]
=========     Host Frame:./test-initcheck [0x8280]
=========     Host Frame:./test-initcheck [0x8108]
=========     Host Frame:./test-initcheck [0x814f]
=========     Host Frame:./test-initcheck [0x7f45]
=========     Host Frame:./test-initcheck [0x7d4d]
=========     Host Frame:/lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xf2) [0x28cb2]
=========     Host Frame:./test-initcheck [0x7b4e]
=========
========= ERROR SUMMARY: 1 error
```

CUDA Racecheck

- Tests shared memory data access hazards
  - Helps identify memory access race conditions
    - Write-After-Write (WAW) hazards
      - Two threads write data to same location without sync
    - Write-After-Read (WAR) hazards
      - Two threads access same location, one writes, and one reads
    - Read-After-Write (RAW) hazards
      - Same as WAR, different order

CUDA Racecheck

- Reduction example

```
__global__ void test_racecheck(const int* in, int* out, int num_elements) {
    __shared__ volatile float data[256];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_elements)
        return;

    // load data from global -> reduce values down until only one warp left ...

    if (threadIdx.x < 32) {
        data[threadIdx.x] += data[threadIdx.x + 32];
        data[threadIdx.x] += data[threadIdx.x + 16];
        data[threadIdx.x] += data[threadIdx.x + 8];
        data[threadIdx.x] += data[threadIdx.x + 4];
        data[threadIdx.x] += data[threadIdx.x + 2];
    }

    // thread 0 writes to global
}
```

CUDA Racecheck

```
clubber:gpuprog > cuda-memcheck --tool racecheck ./test-racecheck
========= CUDA-MEMCHECK
========= WARN: Race reported between Write access at 0x00000768 in racecheck.cu:13:test_racecheck(int const *, int*, int)
=========     and Read access at 0x00000818 in racecheck.cu:14:test_racecheck(int const *, int*, int) [112 hazards]
=========
========= WARN: Race reported between Write access at 0x000003f8 in racecheck.cu:11:test_racecheck(int const *, int*, int)
=========     and Read access at 0x000004b0 in racecheck.cu:12:test_racecheck(int const *, int*, int) [64 hazards]
=========
========= WARN: Race reported between Write access at 0x00000918 in racecheck.cu:14:test_racecheck(int const *, int*, int)
=========     and Read access at 0x000009d0 in racecheck.cu:15:test_racecheck(int const *, int*, int) [120 hazards]
=========
========= WARN: Race reported between Write access at 0x000005b0 in racecheck.cu:12:test_racecheck(int const *, int*, int)
=========     and Read access at 0x00000668 in racecheck.cu:13:test_racecheck(int const *, int*, int) [96 hazards]
=========
========= RACECHECK SUMMARY: 4 hazards displayed (0 errors, 4 warnings)
```

CUDA Racecheck

```
__global__ void test_racecheck(const int* in, int* out, int num_elements) {
    __shared__ volatile float data[256];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_elements)
        return;

    // load data from global -> reduce values down until only one warp left ...

    if (threadIdx.x < 32) {
        x += data[threadIdx.x + 32]; __syncwarp(); data[threadIdx.x] = x; __syncwarp();
        x += data[threadIdx.x + 16]; __syncwarp(); data[threadIdx.x] = x; __syncwarp();
        x += data[threadIdx.x + 8];  __syncwarp(); data[threadIdx.x] = x; __syncwarp();
        x += data[threadIdx.x + 4];  __syncwarp(); data[threadIdx.x] = x; __syncwarp();
        x += data[threadIdx.x + 2];  __syncwarp(); data[threadIdx.x] = x; __syncwarp();
    }

    // thread 0 writes to global
}
```

CUDA Synccheck

- Checks correct usage of synchronization primitives
  - Divergent threads in block/warp
  - Invalid arguments to primitives

```
__global__ void test_synccheck(const int* in, int* out, int num_elements) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_elements)
        return;

    if (tid == 0) {
        out[tid] = in[tid];
        __syncwarp(0xFFFFFFFE);
    } else {
        out[tid] = tid;
        __syncwarp(0xFFFFFFFE);
    }
}
```

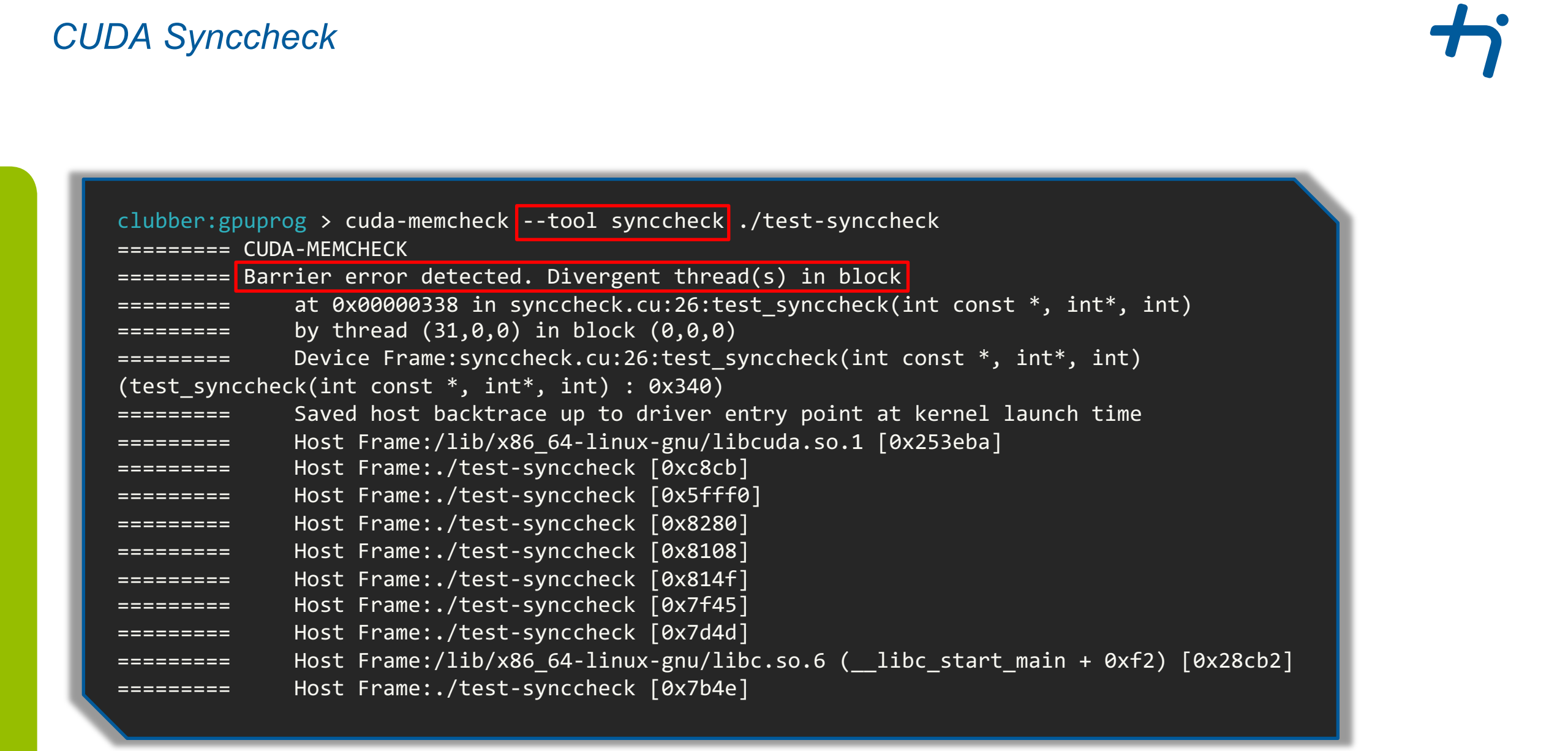
CUDA Synccheck

```
clubber:gpuprog > cuda-memcheck --tool synccheck ./test-synccheck
cuda-memcheck --tool synccheck ./test-synccheck
========= CUDA-MEMCHECK
========= Barrier error detected. Invalid arguments
=========     at 0x00000030 in __cuda_sm70_warpsync
=========     by thread (0,0,0) in block (0,0,0)
=========     Device Frame:__cuda_sm70_warpsync (__cuda_sm70_warpsync : 0x30)
=========     Device Frame:cuda/targets/x86_64-
linux/include/sm_30_intrinsics.hpp:110:_INTERNAL_44_tmpxft_0000dccf_00000000_7_synccheck_cpp1_ii_1521c963::__syncwarp(unsigned int)
(_INTERNAL_44_tmpxft_0000dccf_00000000_7_synccheck_cpp1_ii_1521c963::__syncwarp(unsigned int) : 0x100)
=========     Device Frame:synccheck.cu:10:test_synccheck(int const *, int*, int) (test_synccheck(int const *, int*, int) : 0x4f0)
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/lib/x86_64-linux-gnu/libcuda.so.1 [0x253eba]
=========     Host Frame:./test-synccheck [0xc9cb]
=========     Host Frame:./test-synccheck [0x600f0]
=========     Host Frame:./test-synccheck [0x837a]
=========     Host Frame:./test-synccheck [0x8202]
=========     Host Frame:./test-synccheck [0x8249]
=========     Host Frame:./test-synccheck [0x803f]
=========     Host Frame:./test-synccheck [0x7d67]
=========     Host Frame:/lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xf2) [0x28cb2]
=========     Host Frame:./test-synccheck [0x7b4e]
=========
========= ERROR SUMMARY: 1 error
```

CUDA Synccheck

```
__global__ void test_synccheck(const int* in, int* out, int num_elements) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_elements)
        return;

    if (tid < 32) {
        out[tid] = in[tid];
        __syncthreads();
    } else {
        out[tid] = tid;
    }
    __syncthreads();
}
```

CUDA Synccheck



```
clubber:gpuprog > cuda-memcheck --tool synccheck ./test-synccheck
========= CUDA-MEMCHECK
========= Barrier error detected. Divergent thread(s) in block
=========     at 0x00000338 in synccheck.cu:26:test_synccheck(int const *, int*, int)
=========     by thread (31,0,0) in block (0,0,0)
=========     Device Frame:synccheck.cu:26:test_synccheck(int const *, int*, int)
(test_synccheck(int const *, int*, int) : 0x340)
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/lib/x86_64-linux-gnu/libcuda.so.1 [0x253eba]
=========     Host Frame:./test-synccheck [0xc8cb]
=========     Host Frame:./test-synccheck [0x5fff0]
=========     Host Frame:./test-synccheck [0x8280]
=========     Host Frame:./test-synccheck [0x8108]
=========     Host Frame:./test-synccheck [0x814f]
=========     Host Frame:./test-synccheck [0x7f45]
=========     Host Frame:./test-synccheck [0x7d4d]
=========     Host Frame:/lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main + 0xf2) [0x28cb2]
=========     Host Frame:./test-synccheck [0x7b4e]
```

CUDA-GDB

- Runs kernels in the gdb debugger
- Requires **`-G`** and **`-lineinfo`** for debugging
- Allows
  - Variable inspection
  - Line info
  - Backtrace
  - Stepping
- Works with `cuda-memcheck`

```
cuda-gdb [options] app_name [app_options]
```

CUDA-GDB

- Single-GPU debugging starting with compute capability 6.0
  - For compute capability 5.x, enable software preemption

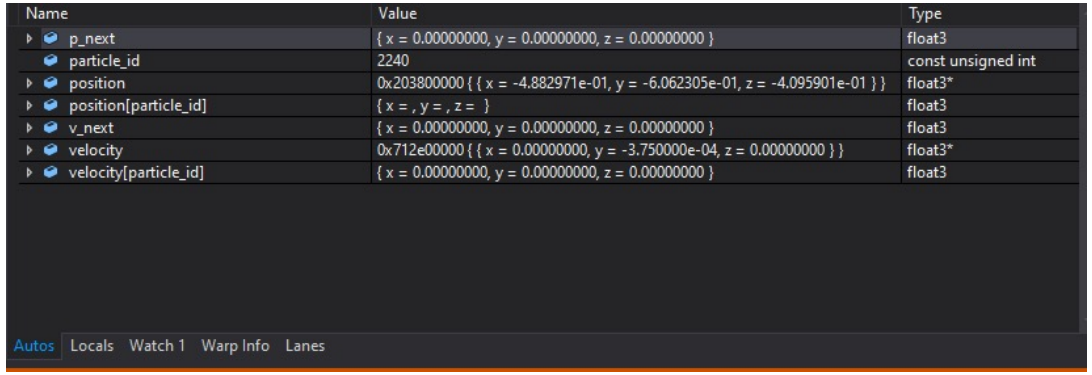
```
clubber:gpuprog > cuda-gdb ./test-memcheck
NVIDIA (R) CUDA Debugger
(cuda-gdb) run
Starting program: test-memcheck
fatal:  All CUDA devices are used for display and cannot be used while debugging.
(error code = CUDBG_ERROR_ALL_DEVICES_WATCHDOGGED(0x18)
(cuda-gdb) set cuda software_preemption on
(cuda-gdb) run[Detaching after fork from child process 11536]
[New Thread 0x7ffff624b000 (LWP 11540)]
[New Thread 0x7ffff5a4a000 (LWP 11541)]
[New Thread 0x7ffff5249000 (LWP 11542)]
[Thread 0x7ffff5249000 (LWP 11542) exited]
[Thread 0x7ffff5a4a000 (LWP 11541) exited]
[Thread 0x7ffff624b000 (LWP 11540) exited]
[Inferior 1 (process 11534) exited normally]
```

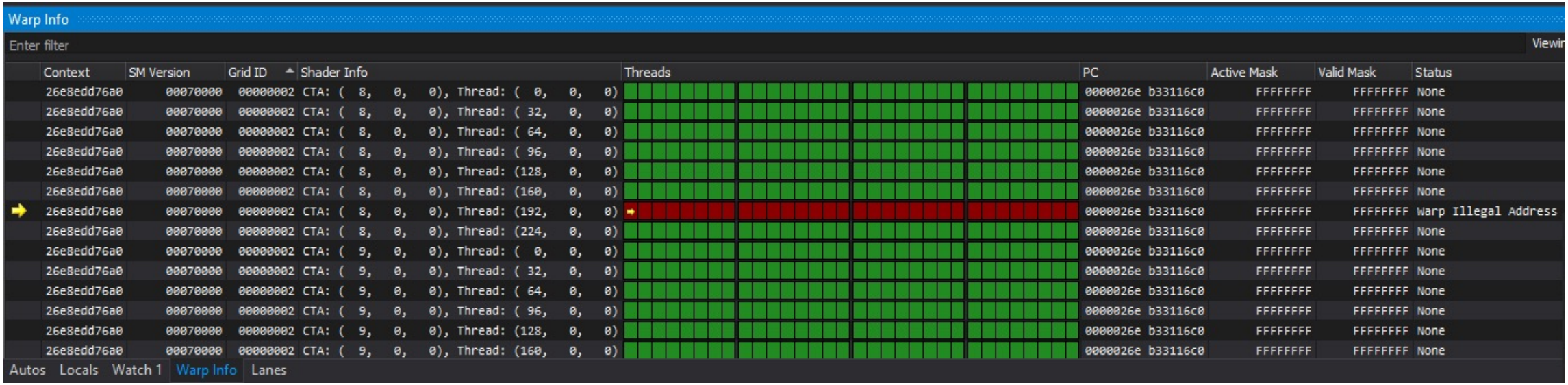
CUDA-GDB

```
clubber:gpuprog > cuda-gdb ./test-memcheck
NVIDIA (R) CUDA Debugger
(cuda-gdb) set cuda memcheck on
(cuda-gdb) run

Illegal access to address (@global)0x707f00000 detected.


Thread 1 "test-memcheck" received signal CUDA_EXCEPTION_1, Lane Illegal Address.
[Switching focus to CUDA kernel 0, grid 1, block (128,0,0), thread (0,0,0), device 0, sm 2, warp 0, lane 0]
0x0000555555a60070 in test_memcheck (in=0x707ec0000, out=0x707ee0000, num_elements=32768) at memcheck.cu:3
3           out[tid] = in[tid];
(cuda-gdb) print tid
$1 = 32768
```

Nsight Visual Studio Debugger

- Set breakpoints in GPU code
- Inspect memory
- View values of locals
- Perform memory checks





Honorable Mention

Our lord and savior `int printf(const char* format, …);`

- Because sometimes the debugger will
  - Not work
  - Make symptoms go away
  - Lie to you
- Use with care
  - Outputs to a FIFO buffer
    - Size can be changed via `cudaDeviceSetLimit()`
  - Displayed only at certain points
    - e.g. `cudaDeviceSynchronize()`
  - Can make symptoms go away

Profiling

Profiling Tools

- Full support on all GPUs up to Volta
  - nvprof: command line profiler
  - nvvp: visual profiler
    - supports profiles from nvprof
- For all newer GPUs, initial support for Volta
  - Nsight Compute
    - ncu: command line profiler
    - ncu-ui: visual profiler
  - Nsight Systems
    - nsys: command line profiler
    - nsys-ui: visual profiler

nvprof

```
clubber:gpuprog > nvprof ./test-memcheck
==5661== NVPROF is profiling process 5661, command: ./test-memcheck
==5661== Profiling application: ./test-memcheck
==5661== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  6.9430us         1  6.9430us  6.9430us  6.9430us  test_memcheck(int const *, int*, int)
      API calls:   99.09%  127.62ms         2  63.812ms  3.8340us  127.62ms  cudaMalloc
                    0.31%  393.17us       101  3.8920us     298ns  232.55us  cuDeviceGetAttribute
                    0.27%  347.10us         1  347.10us  347.10us  347.10us  cudaGetDeviceProperties
                    0.21%  268.78us         1  268.78us  268.78us  268.78us  cuDeviceTotalMem
                    0.06%  77.591us         2  38.795us  5.4790us  72.112us  cudaFree
                    0.03%  38.749us         1  38.749us  38.749us  38.749us  cuDeviceGetName
                    0.02%  25.060us         1  25.060us  25.060us  25.060us  cudaLaunchKernel
                    0.01%  7.7300us         1  7.7300us  7.7300us  7.7300us  cudaDeviceSynchronize
                    0.00%  5.7840us         1  5.7840us  5.7840us  5.7840us  cuDeviceGetPCIBusId
                    0.00%  2.3100us         3     770ns     410ns  1.4680us  cuDeviceGetCount
                    0.00%  1.4160us         2     708ns     340ns  1.0760us  cuDeviceGet
                    0.00%     588ns         1     588ns     588ns     588ns  cuDeviceGetUuid
```

nvprof & nvvp

- Profiling using nvprof
  - Profile metrics, events, etc. , e.g., `nvprof --metrics all ./binary`
- Profiling using nvvp
  - Support for local and remote profiling
  - Support for `nvprof` profile for visualization

```
clubber:gpuprog > nvprof --analysis-metrics --output-profile test-memcheck.nvvp ./test-memcheck
==10227== NVPROF is profiling process 10227, command: ./test-memcheck
==10227== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "test_memcheck(int const *, int*, int)" (done)
==10227== Generated result file: test-memcheck.nvvp

clubber:gpuprog > nvvp test-memcheck.nvvp
```

nsys

System-Wide Performance Analysis

```
clubber:gpuprog > nsys profile --stats=true ./test-memcheck
...
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls  Average (ns)  Minimum (ns)  Maximum (ns)  StdDev (ns)           Name
 -------  ---------------  ---------  ------------  ------------  ------------  ------------  ---------------------
    99,0      101.572.635          2  50.786.317,0         2.609   101.570.026  71.819.009,0  cudaMalloc
     0,0           69.700          2      34.850,0         3.551        66.149      44.263,0  cudaFree
     0,0           14.228          1      14.228,0        14.228        14.228           0,0  cudaLaunchKernel
     0,0            7.951          1       7.951,0         7.951         7.951           0,0  cudaDeviceSynchronize



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average (ns)  Minimum (ns)  Maximum (ns)  StdDev (ns)                  Name
 -------  ---------------  ---------  ------------  ------------  ------------  -----------  ------------------------------------
   100,0            4.672          1       4.672,0         4.672         4.672          0,0  test_memcheck(const int *, int *, int)



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls  Average (ns)  Minimum (ns)  Maximum (ns)  StdDev (ns)        Name
 -------  ---------------  ---------  ------------  ------------  ------------  ------------  --------------
    65,0      100.300.343         13   7.715.411,0         1.346    47.060.062  14.452.623,0  poll
    32,0       50.452.225        642      78.586,0         1.003    16.646.029     711.406,0  ioctl
     0,0        1.336.198         68      19.650,0         1.632       459.311      54.658,0  mmap64
     0,0          487.311         27      18.048,0         1.057       414.655      79.269,0  fopen
     0,0          [illegible]          3.561,0         1.313        12.756       1.324,0  open64
```

nsys & nsys-ui

System-Wide Performance Analysis

- Profiling using nsys
  - Profile application timeline including CPU and GPU
- Profiling using `nsys-ui`
  - Support for local and remote profiling
  - Support for `nsys` profile for visualization

```
clubber:gpuprog > nsys profile -o profile-memcheck ./test-memcheck
Generating '/tmp/nsys-report-34f4.qdstrm'
[1/1] [========================100%] test-memcheck.nsys-rep
Generated:
    profile-memcheck.nsys-rep

clubber:gpuprog > nsys-ui profile-memcheck.nsys-rep
```

ncu

Kernel Profiling

```
clubber:gpuprog > ncu --set full ./test-memcheck
==PROF== Connected to process 80111 (/home/membarth/gpuprog/samples/build/test-memcheck)
going to use NVIDIA GeForce RTX 2080
==PROF== Profiling "test_memcheck" - 1: 0%....50%....100% - 32 passes
==PROF== Disconnected from process 80111
[80111] test-memcheck@127.0.0.1
  test_memcheck(const int *, int *, int), 2022-Apr-13 16:01:14, Context 1, Stream 7
    Section: GPU Speed Of Light Throughput
    ---------------------------------------------------------------------- -------------- ------------------------------
    DRAM Frequency                                                    cycle/nsecond                           2.70
    SM Frequency                                                      cycle/usecond                         606.04
    Elapsed Cycles                                                            cycle                          3,345
    Memory [%]                                                                    %                          13.98
    DRAM Throughput                                                               %                          13.98
    Duration                                                                usecond                           5.50
    L1/TEX Cache Throughput                                                       %                           7.45
    L2 Cache Throughput                                                           %                           9.98
    SM Active Cycles                                                          cycle                       1,797.89
    Compute (SM) [%]                                                              %                           4.01
    ---------------------------------------------------------------------- -------------- ------------------------------
    WRN   This kernel grid is too small to fill the available resources on this device, resulting in only 0.7 full
          waves across all SMs. Look at Launch Statistics for more details.

    INF   The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 0% of
          this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide
          (https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline) for mode details on roofline
          analysis.

    Section: Compute Workload Analysis
```

ncu & ncu-ui

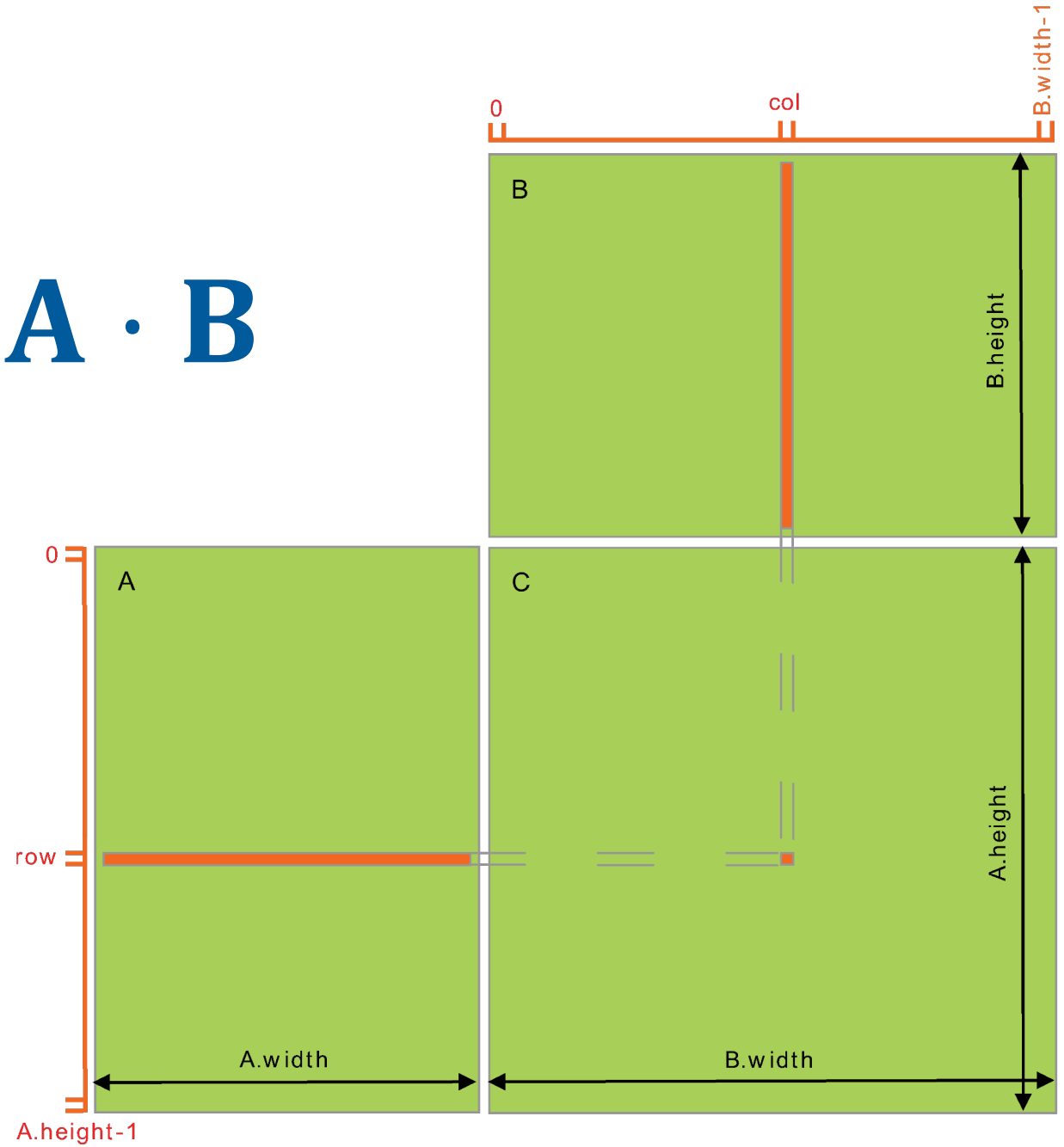
Kernel Profiling

- Profiling using ncu
  - Profile metrics, events, etc. , e.g., `ncu --set full ./binary`
- Profiling using `ncu-ui`
  - Support for local and remote profiling
  - Support for `ncu` profile for visualization

```
clubber:gpuprog ncu -o profile-memcheck --set full ./test-memcheck
==PROF== Connected to process 80144 (/home/membarth/gpuprog/samples/build/test-memcheck)
==PROF== Profiling "test_memcheck" - 1: 0%....50%....100% - 32 passes
==PROF== Disconnected from process 80144
==PROF== Report: profile-memcheck.ncu-rep

clubber:gpuprog > ncu-ui profile-memcheck.ncu-rep
```

Example: Matrix Multiplication

$$C = A \cdot B$$


0
col
B.width-1
B
B.height
0
A
C
row
A.height
A.width
B.width
A.height-1

CPU Implementation

```
std::vector<float> cpu_multiply(const std::vector<float>& A, size_t A_cols, size_t A_rows,
                                const std::vector<float>& B, size_t B_cols, size_t B_rows) {
    size_t C_rows = A_rows;
    size_t C_cols = B_cols;
    std::vector<float> C(C_rows * C_cols);

    size_t elements = A_cols;
    auto c_it = C.begin();

    for (size_t y = 0; y < C_rows; ++y) {
        for (size_t x = 0; x < C_cols; ++x) {
            float c_xy{0};
            for (size_t i = 0; i < elements; ++i)
                c_xy += A[y*A_cols + i] * B[i*B_cols + x];
            *c_it++ = c_xy;
        }
    }

    return C;
}
```

loop-based parallelism

GPU Implementation

```
__global__ void d_multiply(const float* A, int A_cols, int A_rows,
                           const float* B, int B_cols, int B_rows, float* C) {
    int C_rows = A_rows;
    int C_cols = B_cols;

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int elements = A_cols;
    float c_xy{0};

    for (auto i = 0; i < elements; ++i)
        c_xy += A[y*A_cols + i] * B[i*B_cols + x];

    C[y*C_cols + x] = c_xy;
}
```

```
for (size_t y = 0; y < C_rows; ++y) {
    for (size_t x = 0; x < C_cols; ++x) {
```

Kernel Call

```
for (size_t y = 0; y < C_rows; ++y) {
    for (size_t x = 0; x < C_cols; ++x) {
```

```
// the kernel call
dim3 block_size(16, 16, 1);
dim3 grid_size(C_cols / block_size.x + 1, C_rows / block_size.y + 1, 1);
d_multiply<<<grid_size, block_size>>>(d_A, A_cols, A_rows, d_B, B_cols, B_rows, d_C);

// synchronize
cudaError err = cudaDeviceSynchronize();
```

Matrix Multiplication: CPU vs. GPU Execution

```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce RTX 2080
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the CPU done in 1.65184 seconds
computing C on the GPU done in 0.00626 seconds (0.00619s on device)
computing difference
max absolute difference: 0.00007@ -57.35104 -57.35097
avg relative difference: 0.00000
```

```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce GTX 970
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the CPU done in 12.04407 seconds
computing C on the GPU done in  0.02084 seconds (0.02077s on device)
computing difference
max absolute difference: 0.00007@ -57.35104 -57.35097
avg relative difference: 0.00000
```

Matrix Multiplication: Memory

```
// global memory allocation
float* d_A, *d_B, *d_C;
cudaMalloc(&d_A; A_cols*A_rows*sizeof(float));
cudaMalloc(&d_b; B_cols*B_rows*sizeof(float));
cudaMalloc(&d_C; C_cols*C_rows*sizeof(float));

// memory transfer
cudaMemcpy(d_A, &A[0], A_cols*A_rows*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, &B[0], B_cols*B_rows*sizeof(float), cudaMemcpyHostToDevice);

// the kernel call
dim3 block_size(16, 16, 1);
dim3 grid_size(C_cols / block_size.x + 1, C_rows / block_size.y + 1, 1);
d_multiply<<<grid_size, block_size>>>(d_A, A_cols, A_rows, d_B, B_cols, B_rows, d_C);

// synchronize
cudaError err = cudaDeviceSynchronize();

// copy to host
std::vector<float> C(C_rows*C_cols);
cudaMemcpy(&C[0], d_C, C_cols*C_rows*sizeof(float), cudaMemcpyDeviceToHost);

// memory clean up
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Example: Tiled Matrix Multiplication

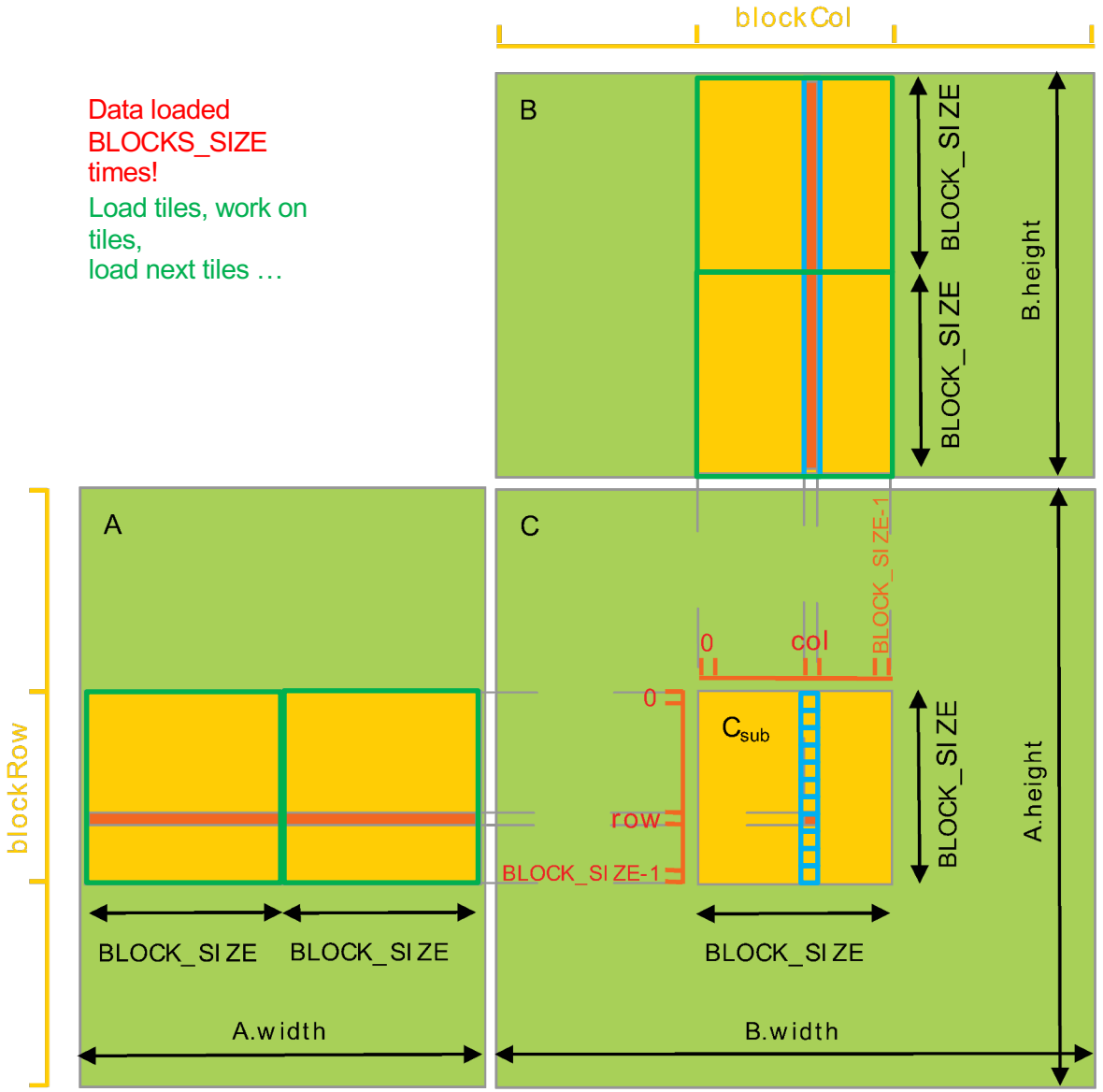
Matrix Multiplication: Problems

- A lot of memory access with little computations only
- Memory access is all going to slow global memory
- In a block the same memory is needed by multiple threads

- Solution
  - Use shared memory to load one tile of data
  - Consume the data together
  - Advance to next block

Data loaded BLOCKS_SIZE times!

Load tiles, work on tiles,
load next tiles …



```
template <int TILE_WIDTH>
__global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
                                 const float* B, int B_cols, int B_rows, float* C) {
    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];

    int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
    int y = blockIdx.y * TILE_WIDTH + threadIdx.y;

    int elements = A_cols;
    float c_xy{0};
    int tiles = elements/TILE_WIDTH;
    for (int tile = 0; tile < tiles; ++i) {
        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];

        for (auto i = 0; i < TILE_WIDTH; ++i)
            c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];

    }
    C[y*B_cols + x] = c_xy;
}
```

Blocksize:
TILE_WIDTH * TILE_WIDTH

```
c_xy += A[y*A_cols + i] * B[i*B_cols + x];
```

Matrix Multiplication: Tiled GPU Execution

```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce RTX 2080
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.00626 seconds (0.00620s on device)
computing C on the GPU in tiled mode done in 0.00364 seconds (0.00357s on device)
computing difference
max absolute difference: 249.19531@ -93.76251 155.43280
avg relative difference: 12.17100
```

```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce GTX 970
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.02081 seconds (0.02074s on device)
computing C on the GPU in tiled mode done in 0.00677 seconds (0.00670s on device)
computing difference
max absolute difference: 249.66107@ -89.97697 159.68411
avg relative difference: 11.61633
```

```
template <int TILE_WIDTH>
__global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
                                 const float* B, int B_cols, int B_rows, float* C) {
    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];

    int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
    int y = blockIdx.y * TILE_WIDTH + threadIdx.y;

    int elements = A_cols;
    float c_xy{0};
    int tiles = elements/TILE_WIDTH;
    for (int tile = 0; tile < tiles; ++i) {
        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
        __syncthreads();
        for (auto i = 0; i < TILE_WIDTH; ++i)
            c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];

    }
    C[y*B_cols + x] = c_xy;
}
```

Matrix Multiplication: Tiled GPU Execution

```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce RTX 2080
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.00625 seconds (0.00619s on device)
computing C on the GPU in tiled mode done in 0.00380 seconds (0.00373s on device)
computing difference
max absolute difference: 12.48769@ 23.67817 36.16587
avg relative difference: 0.00607
```

```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce GTX 970
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.02082 seconds (0.02075s on device)
computing C on the GPU in tiled mode done in 0.00735 seconds (0.00728s on device)
computing difference
max absolute difference: 7.04418@ 71.03987 63.99569
avg relative difference: 0.00008
```

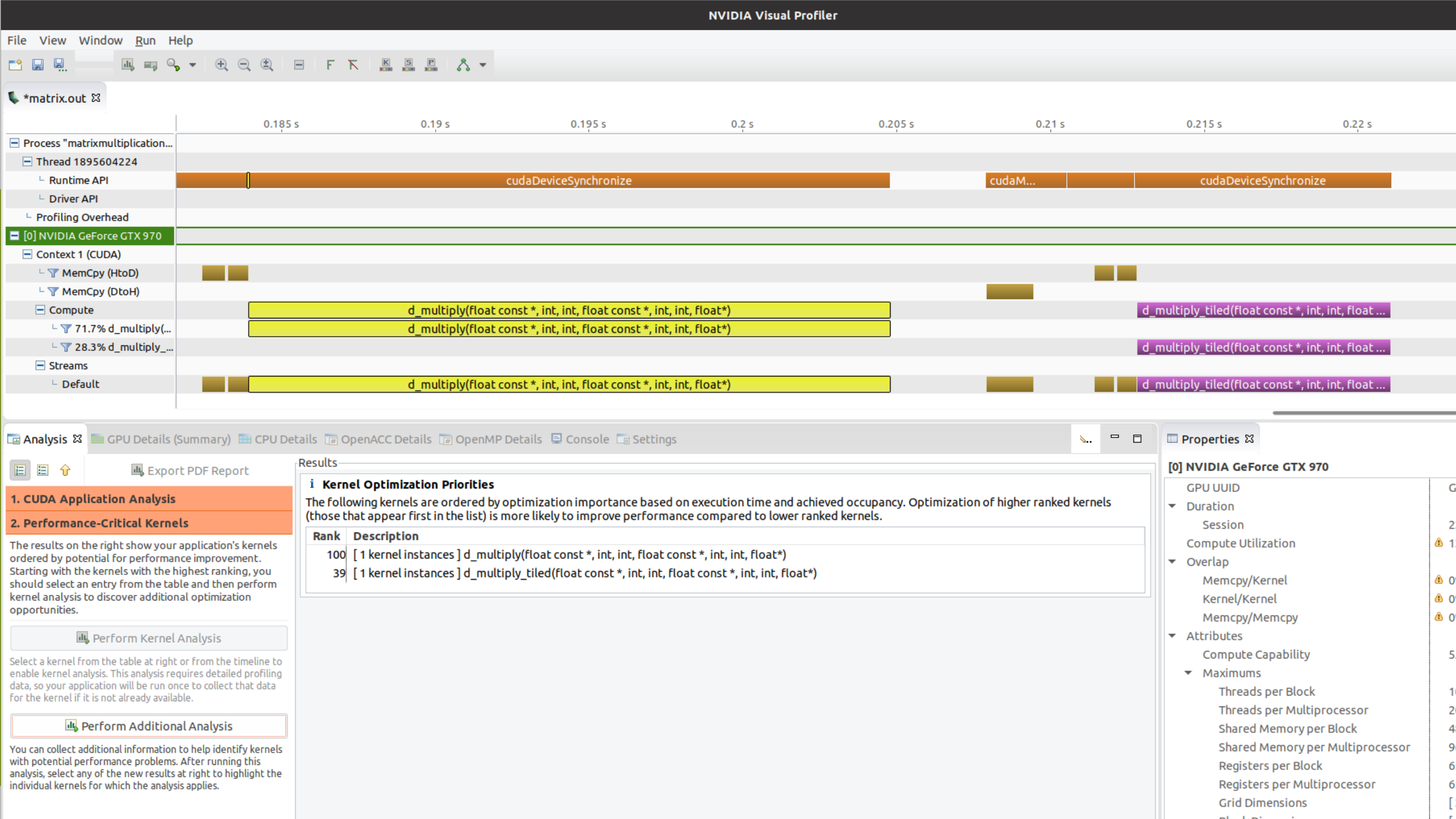
```cpp
template <int TILE_WIDTH>
__global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
                                 const float* B, int B_cols, int B_rows, float* C) {
    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];

    int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
    int y = blockIdx.y * TILE_WIDTH + threadIdx.y;

    int elements = A_cols;
    float c_xy{0};
    int tiles = elements/TILE_WIDTH;
    for (int tile = 0; tile < tiles; ++i) {
        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
        __syncthreads();
        for (auto i = 0; i < TILE_WIDTH; ++i)
            c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];
        __syncthreads();
    }
    C[y*B_cols + x] = c_xy;
}
```

Overwritten before read from other thread!!

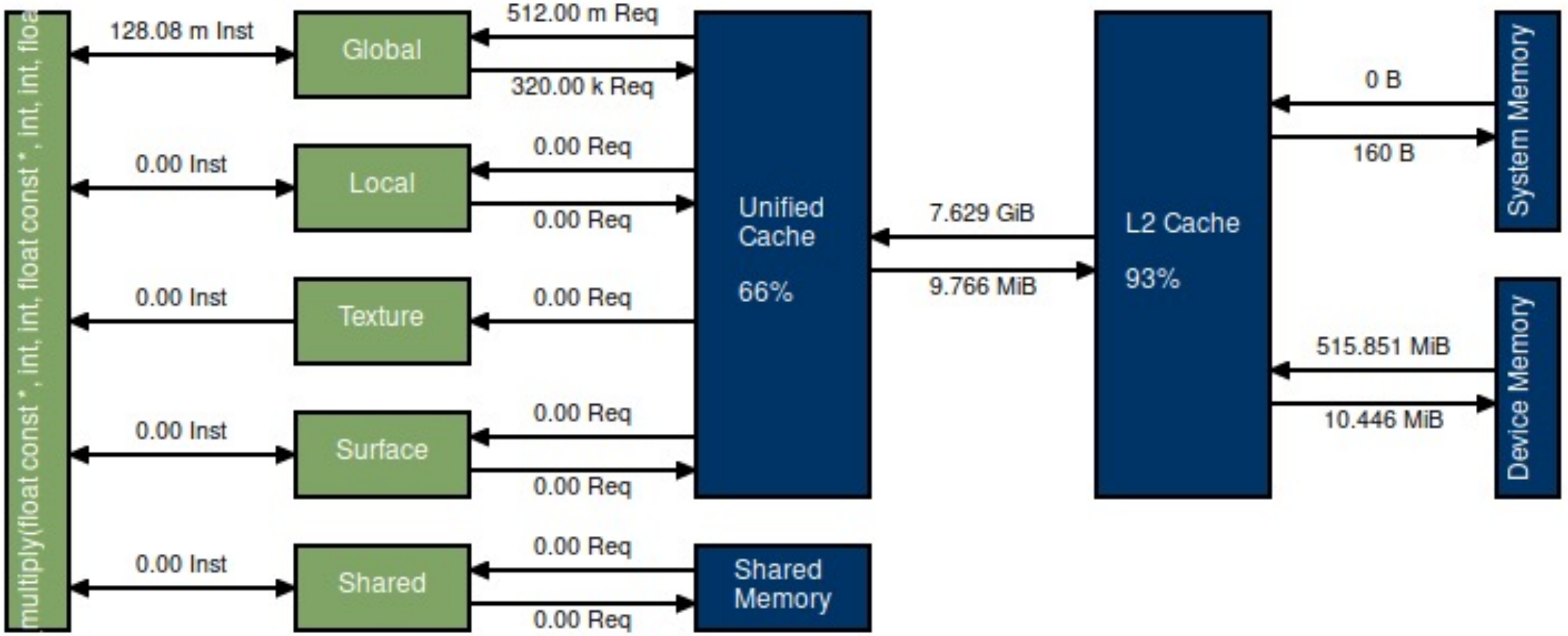
Matrix Multiplication: Tiled GPU Execution

```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce RTX 2080
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.00864 seconds (0.00858s on device)
computing C on the GPU in tiled mode done in 0.00428 seconds (0.00420s on device)
computing difference
max absolute difference: 0.00000@ 0.00000 0.00000
avg relative difference: 0.00000
```

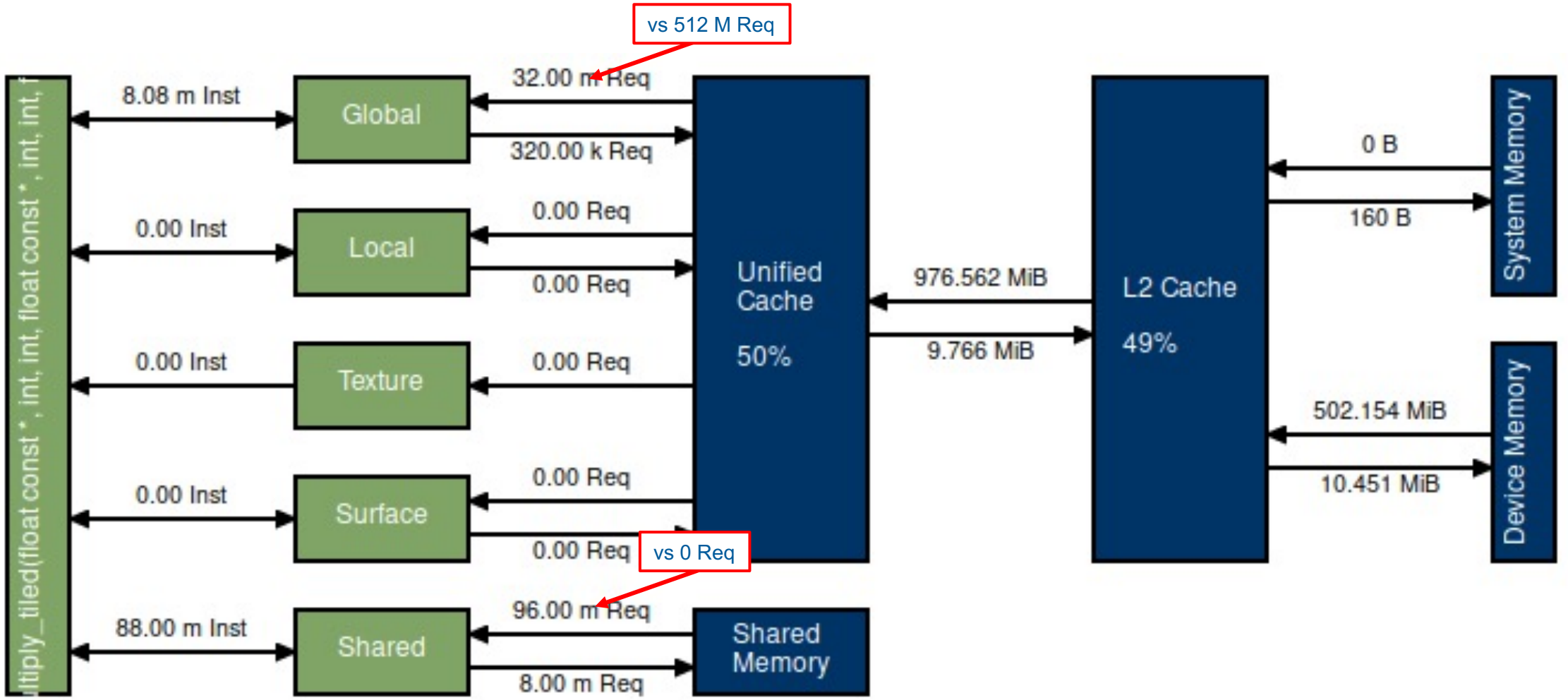
```
clubber:gpuprog > ./matrixmultiplication
going to use NVIDIA GeForce GTX 970
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.02079 seconds (0.02072s on device)
computing C on the GPU in tiled mode done in 0.00834 seconds (0.00826s on device)
computing difference
max absolute difference: 0.00000@ 0.00000 0.00000
avg relative difference: 0.00000
```


NVIDIA Visual Profiler
File View Window Run Help
*matrix.out
0.185 s
0.19 s
0.195 s
0.2 s
0.205 s
0.21 s
0.215 s
0.22 s
Process "matrixmultiplication...
Thread 1895604224
Runtime API
Driver API
Profiling Overhead
[0] NVIDIA GeForce GTX 970
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
71.7% d_multiply(...
28.3% d_multiply_...
Streams
Default
cudaDeviceSynchronize
cudaM...
cudaDeviceSynchronize
d_multiply(float const *, int, int, float const *, int, int, float*)
d_multiply_tiled(float const *, int, int, float ...
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
OpenMP Details
Console
Settings
Export PDF Report
1. CUDA Application Analysis
2. Performance-Critical Kernels
The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the kernels with the highest ranking, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.
Perform Kernel Analysis
Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your application will be run once to collect that data for the kernel if it is not already available.
Perform Additional Analysis
You can collect additional information to help identify kernels with potential performance problems. After running this analysis, select any of the new results at right to highlight the individual kernels for which the analysis applies.
Results
Kernel Optimization Priorities
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.
Rank
Description
100
[1 kernel instances] d_multiply(float const *, int, int, float const *, int, int, float*)
39
[1 kernel instances] d_multiply_tiled(float const *, int, int, float const *, int, int, float*)
Properties
[0] NVIDIA GeForce GTX 970
GPU UUID
Duration
Session
Compute Utilization
Overlap
Memcpy/Kernel
Kernel/Kernel
Memcpy/Memcpy
Attributes
Compute Capability
Maximums
Threads per Block
Threads per Multiprocessor
Shared Memory per Block
Shared Memory per Multiprocessor
Registers per Block
Registers per Multiprocessor
Grid Dimensions

Memory Statistics: Non-Tiled Matrix Multiplication



Memory Statistics: Tiled Matrix Multiplication



Development Systems at THI

- Development systems
  - Intel Xeon E5-1620 CPU
  - NVIDIA GTX 970
  - Hosts
    - ei165a15.rz.fh-ingolstadt.de
    - ei165b15.rz.fh-ingolstadt.de (offline right now)
- Remote access via ssh
  - `ssh -l <username> ei165a15.rz.fh-ingolstadt.de`
  - Requires VPN

Development Systems at THI

- Development systems
  - OS: Ubuntu 18.04 LTS
  - Intel Core i9-10900K CPU
  - NVIDIA Quadro RTX 4000
  - CIP pool G308
- CUDA Toolkit 11.4 (/opt/cuda-11.4 installation)
  - Need to setup environment to find binaries and libraries

```
clubber:gpuprog > tail -n 3 ~/.bashrc
# setup paths for CUDA installastion
PATH=/opt/cuda-11.4/bin:$PATH
LD_LIBRARY_PATH=/opt/cuda-11.4/lib64:$LD_LIBRARY_PATH
```