DevTools GTC 2020 Lab Content

Requirements

- Ubuntu 18.04 host (other OS variants may provide different results)
- Eclipse 4.9 (2018-09) for C++ developers
- default-jre apt package
- python apt package
- CUDA 11.0 host (content compatible with CUDA 10.2)
- Turing GPU (tested on RTX 2080Ti)
  - If the device has multiple GPUs, ensure that compute (CUDA) and display (OpenGL) are on the same device, i.e. set CUDA_VISIBLE_DEVICES to the one with the display. Otherwise, CUDA plus OpenGL will result in high memory mapping overhead.
- libglew-dev apt package
- freeglut3-dev package

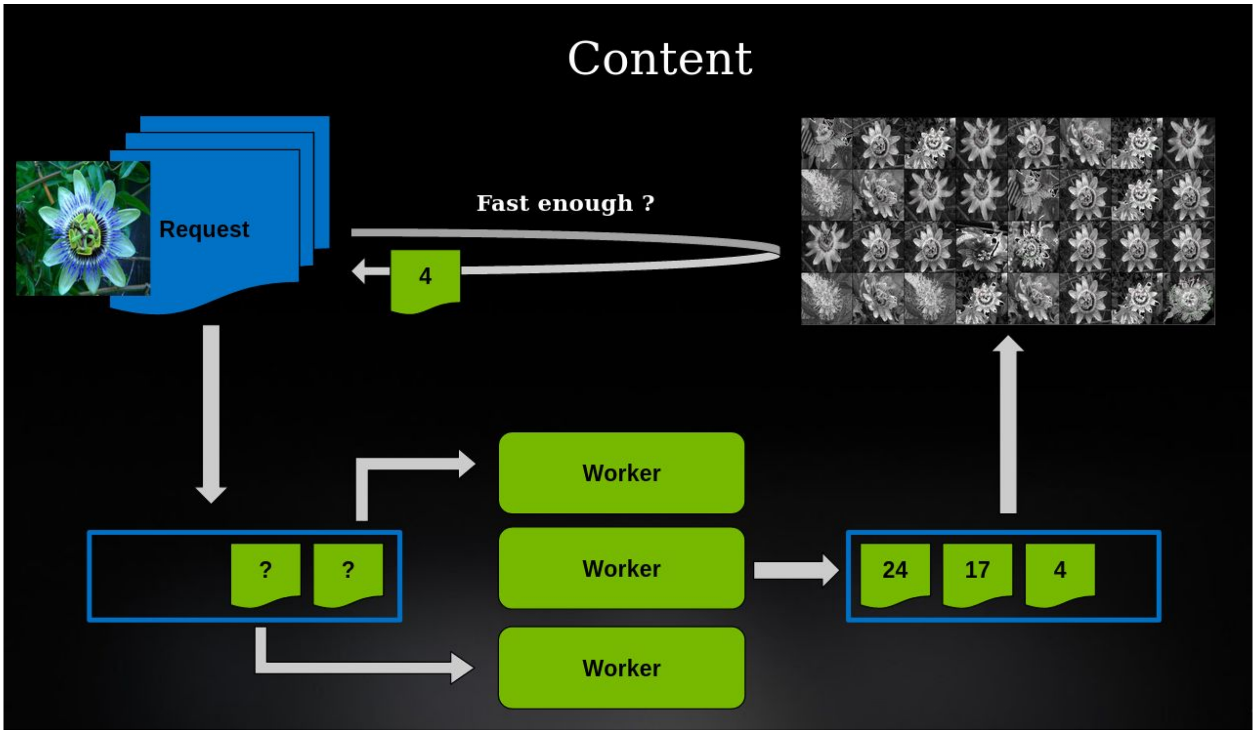
Story

The exercise represents a cloud-based image classification application. It contains a database (DB) of flower images. Users can submit their own flower pictures and the app classifies them against the DB and sends the result back to the user. The application allows users to tweak several factors including the threshold, number of source threads generating requests, and the number of worker threads processing these requests.

For simplicity, the DB contains a fixed number of flower images, and users only submit pictures of one of those flowers. For further simplification, the user images are identical to the DB versions, except for added white noise to simulate some random errors when taking the snapshot.

The exercise is to first fix several bugs in the application using correctness analysis tools to make the classification work properly. Then, users must optimize the application (framework and kernels) so that requests are served within some defined time threshold (e.g. 100ms). It is structured into steps that build on each other, but it is possible to start at any chosen step, as each one will contain all previous exercise content (e.g. all bugs are fixed in the first performance analysis exercise).

This exercise uses NVIDIA® Nsight™ Eclipse Edition but all steps can be completed when compiling and running directly from the command line. These steps are marked as “CLI Alternative” below.



Content

Request

Fast enough ?

4

?

?

Worker

Worker

Worker

24

17

4

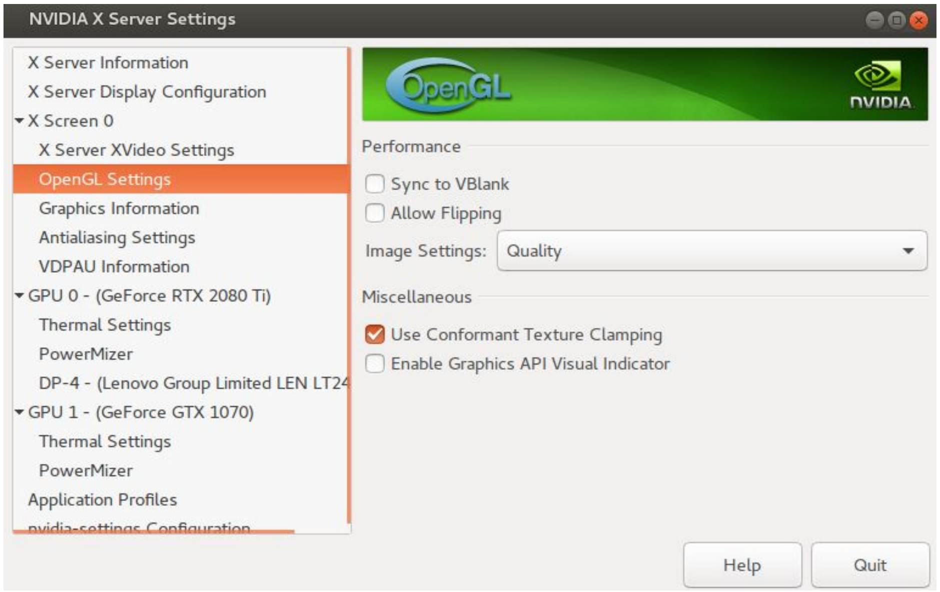
Recordings

- Step 0: Nsight Eclipse Edition https://developer.nvidia.com/gtc/2020/video/t21395-1
- Step 1: cuda-gdb https://developer.nvidia.com/gtc/2020/video/t21395-2
- Step 3-4: Nsight Systems https://developer.nvidia.com/gtc/2020/video/t21395-3
- Step 5-7: Nsight Compute https://developer.nvidia.com/gtc/2020/video/t21395-4

Setup

Host Setup

1. CUDA toolkit installation (done via laptop imaging)
    a. Desktop
        i. CUDA 10.2 with driver
        ii. Disable VSync and Flipping in Nvidia OpenGL settings for best perf



Performance

□ Sync to VBlank

□ Allow Flipping

Image Settings: Quality

Miscellaneous

☑ Use Conformant Texture Clamping

□ Enable Graphics API Visual Indicator

2. Extract/git clone <insert tarfile name> lab files.
3. Install required packages

```
$ sudo apt-get install default-jre python libglew-dev freeglut3-dev
```

4. Install (download and unpack) Eclipse 4.9, e.g. from https://download.eclipse.org/technology/epp/downloads/release/2018-09/R/eclipse-cpp-2018-09-linux-gtk-x86_64.tar.gz
5. Install the Nsight Eclipse Edition eclipse plugin
    a. Desktop: by running <cuda install dir>/bin/nsight_ee_plugins_manage.sh install <eclipse-path>
6. Generate the Nsight Eclipse Edition projects and launch configurations
    a. Create clean directory /home/nvidia/cuda-workspace
    b. We use the generate_nsightee_projects.py script

```
generate_nsightee_projects.py --src <code location> --dst
/home/nvidia/cuda-workspace --steps 7
```

    c. This will create all the project steps, a launch configuration for each step, as well as copy the image sources to the workspace directory

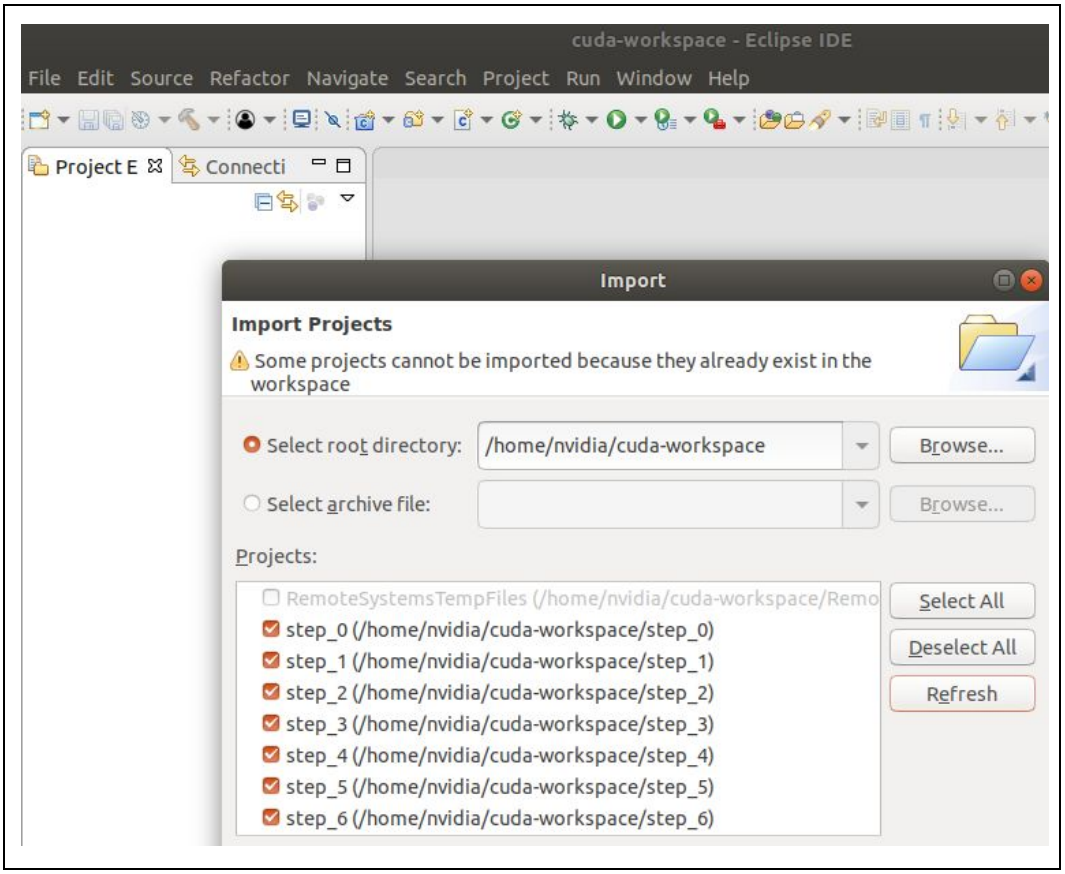
```
$ ls -l ~/cuda-workspace/
drwxrwxr-x 2 nvidia nvidia 4096 Jan 10 16:24 img
-rw-rw-r-- 1 nvidia nvidia 1003 Jan 16 17:55 launch_step_0.launch
-rw-rw-r-- 1 nvidia nvidia 1003 Jan 16 17:55 launch_step_1.launch

-rw-rw-r-- 1 nvidia nvidia 1003 Jan 16 17:55 launch_step_3.launch
-rw-rw-r-- 1 nvidia nvidia 1003 Jan 16 17:55 launch_step_4.launch
-rw-rw-r-- 1 nvidia nvidia 1003 Jan 16 17:55 launch_step_5.launch
-rw-rw-r-- 1 nvidia nvidia 1003 Jan 16 17:55 launch_step_6.launch
-rw-rw-r-- 1 nvidia nvidia 1003 Jan 16 17:55 launch_step_7.launch
drwxrwxr-x 2 nvidia nvidia 4096 Jan 16 18:15 step_0
drwxrwxr-x 2 nvidia nvidia 4096 Jan 16 18:17 step_1

drwxrwxr-x 2 nvidia nvidia 4096 Jan 16 18:25 step_3
drwxrwxr-x 2 nvidia nvidia 4096 Jan 16 17:55 step_4
drwxrwxr-x 2 nvidia nvidia 4096 Jan 16 17:55 step_5
drwxrwxr-x 2 nvidia nvidia 4096 Jan 16 17:55 step_6
drwxrwxr-x 2 nvidia nvidia 4096 Jan 16 17:55 step_7
```

Step 0: Nsight Eclipse Edition (plugin)

Step 0 Video: https://developer.nvidia.com/gtc/2020/video/t21395-1

Building the application using Nsight Eclipse Edition (plugin).

1. Open Eclipse, decide on Nsight Eclipse Edition plugin data collection preference
2. If asked, select ~/cuda-workspace as default workspace
3. Import projects from /home/nvidia/cuda-workspace
    a. Project Explorer > Right Click > Import > General > Existing Projects into Workspace > Next
    b. Specify /home/nvidia/cuda-workspace and select Refresh
    c. Keep all projects enabled, note that step 2 will be missing
    d. Finish


cuda-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Project E
Connecti
Import
Browse...
Browse...
Select All
Deselect All
Refresh

Import Projects

⚠ Some projects cannot be imported because they already exist in the workspace

● Select root directory: /home/nvidia/cuda-workspace

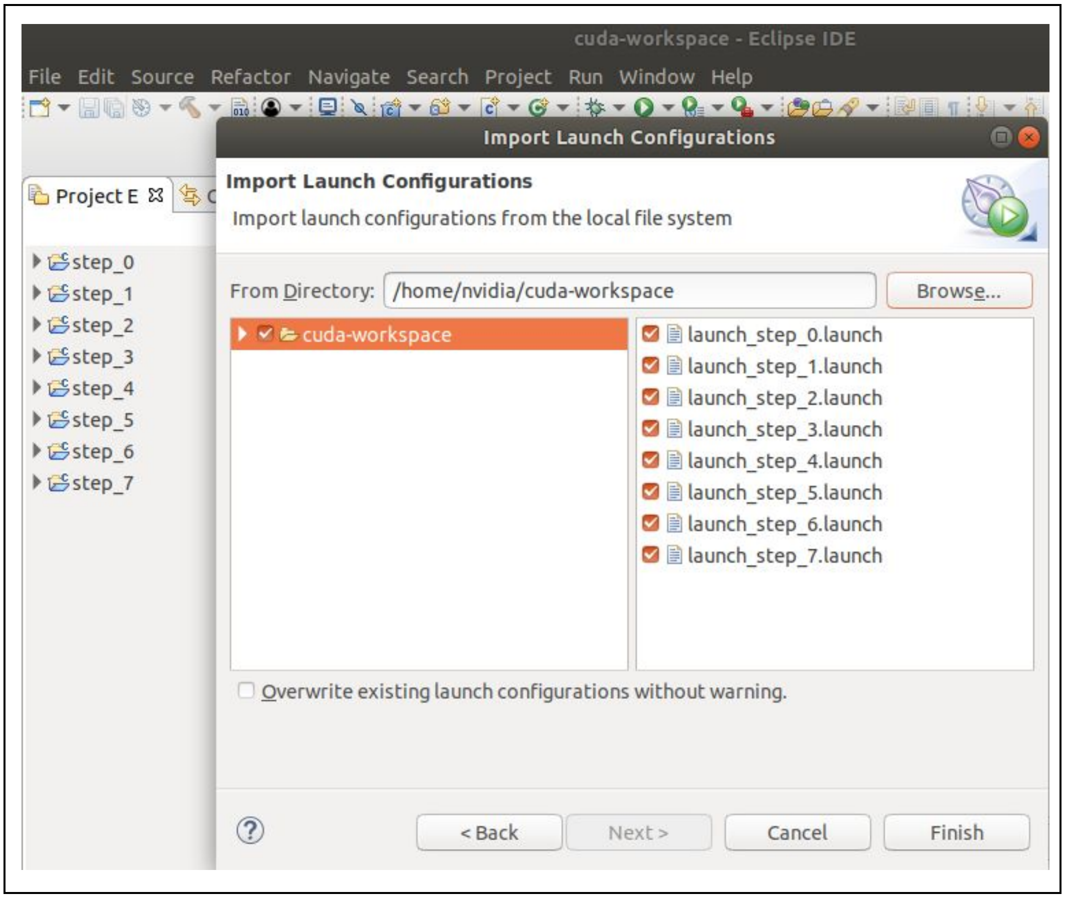
○ Select archive file:

Projects:

- □ RemoteSystemsTempFiles (/home/nvidia/cuda-workspace/Remo
- ☑ step_0 (/home/nvidia/cuda-workspace/step_0)
- ☑ step_1 (/home/nvidia/cuda-workspace/step_1)
- ☑ step_2 (/home/nvidia/cuda-workspace/step_2)
- ☑ step_3 (/home/nvidia/cuda-workspace/step_3)
- ☑ step_4 (/home/nvidia/cuda-workspace/step_4)
- ☑ step_5 (/home/nvidia/cuda-workspace/step_5)
- ☑ step_6 (/home/nvidia/cuda-workspace/step_6)

e.

4. Import launch configs from /home/nvidia/cuda-workspace
    a. Project Explorer > Right Click > Import > Run/Debug> Launch Configurations > Next
    b. Browse /home/nvidia/cuda-workspace and select checkbox
    c. Keep all launch configs enabled
    d. Finish



Import Launch Configurations

Import launch configurations from the local file system

From Directory: /home/nvidia/cuda-workspace

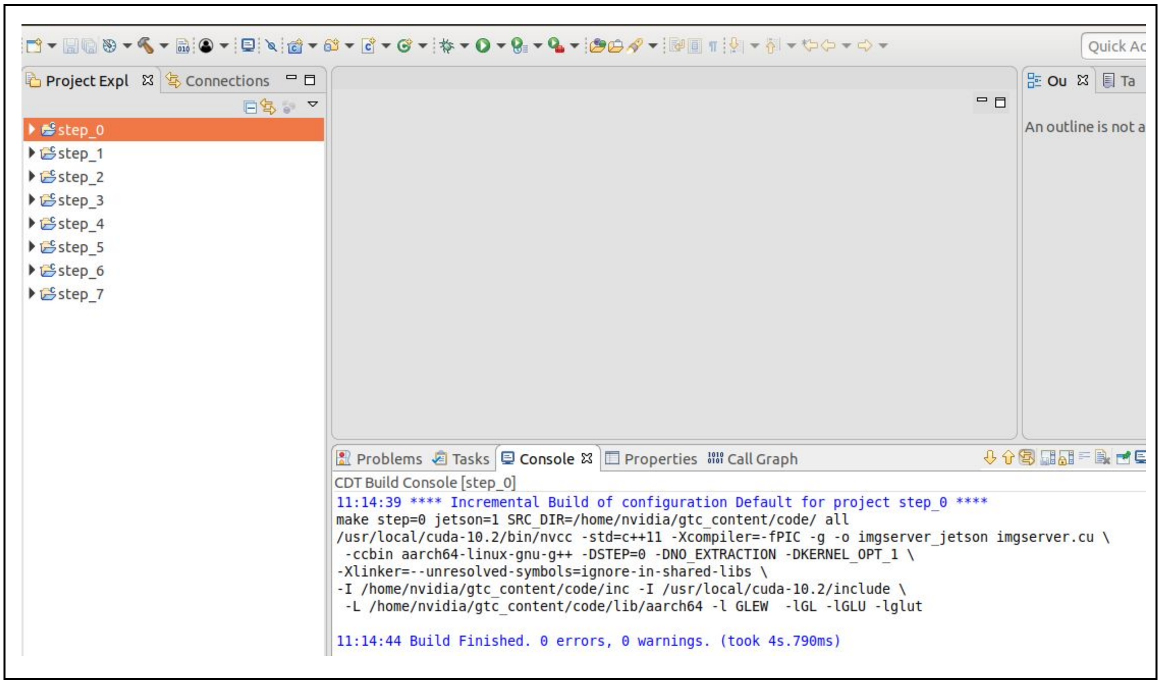
☑ cuda-workspace

☑ launch_step_0.launch
☑ launch_step_1.launch
☑ launch_step_2.launch
☑ launch_step_3.launch
☑ launch_step_4.launch
☑ launch_step_5.launch
☑ launch_step_6.launch
☑ launch_step_7.launch

□ Overwrite existing launch configurations without warning.

cuda-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Project E
step_0
step_1
step_2
step_3
step_4
step_5
step_6
step_7
Browse...
< Back
Next >
Cancel
Finish

5. Select step_0 and build (hammer icon)

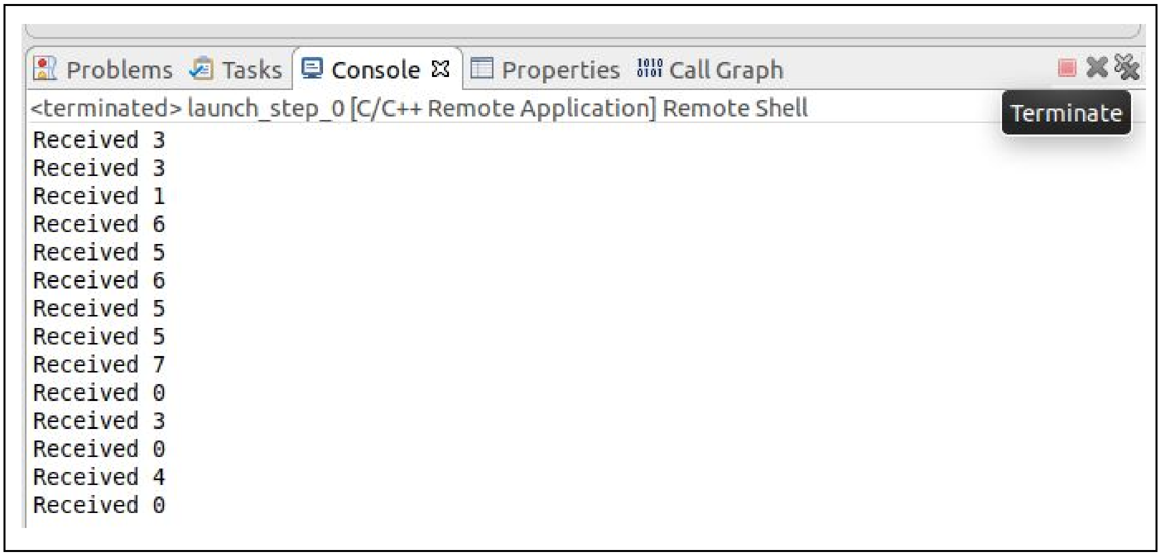


CDT Build Console [step_0]

```
11:14:39 **** Incremental Build of configuration Default for project step_0 ****
make step=0 jetson=1 SRC_DIR=/home/nvidia/gtc_content/code/ all
/usr/local/cuda-10.2/bin/nvcc -std=c++11 -Xcompiler=-fPIC -g -o imgserver_jetson imgserver.cu \
 -ccbin aarch64-linux-gnu-g++ -DSTEP=0 -DNO_EXTRACTION -DKERNEL_OPT_1 \
-Xlinker=--unresolved-symbols=ignore-in-shared-libs \
-I /home/nvidia/gtc_content/code/inc -I /usr/local/cuda-10.2/include \
 -L /home/nvidia/gtc_content/code/lib/aarch64 -l GLEW  -lGL -lGLU -lglut

11:14:44 Build Finished. 0 errors, 0 warnings. (took 4s.790ms)
```

6. Launch using the green Run button

7. Terminate using the red Terminate button once you see Received N messages



```
<terminated> launch_step_0 [C/C++ Remote Application] Remote Shell
Received 3
Received 3
Received 1
Received 6
Received 5
Received 6
Received 5
Received 5
Received 7
Received 0
Received 3
Received 0
Received 4
Received 0
```

CLI Alternative:

1. Build the application with*

```
$ make clean
$ make step=0
```

2. Run the app to see the anticipated app behavior (without any actual classification happening, all requests are simply rendered directly).

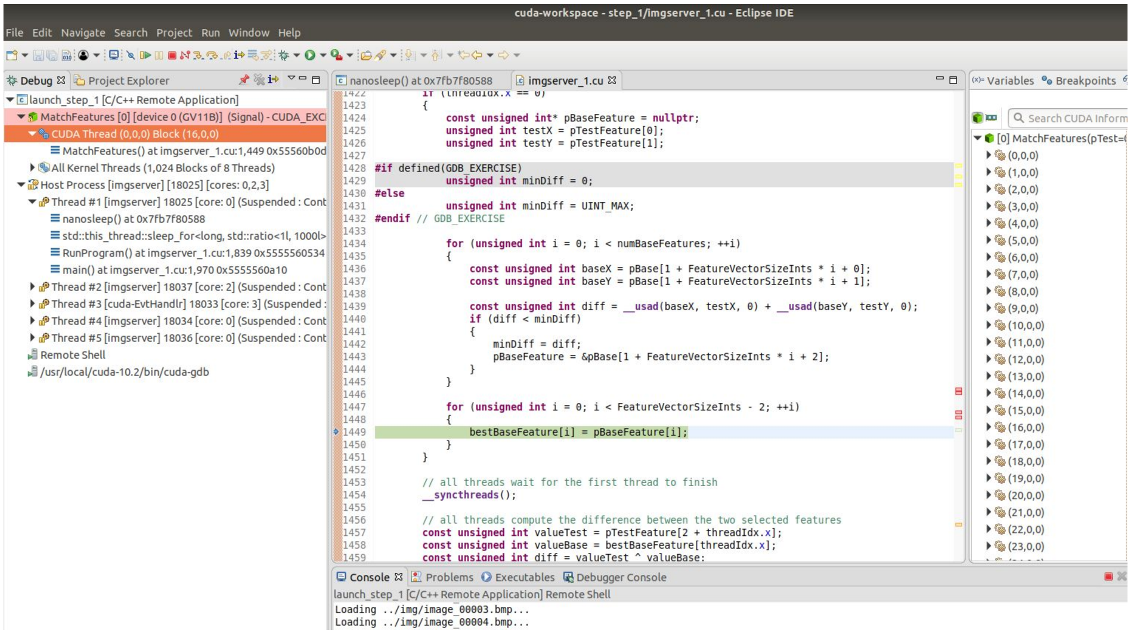
```
$ ./imgserver 50 100
```

Step 1: cuda-gdb

Step 1 Video: https://developer.nvidia.com/gtc/2020/video/t21395-2

Debugging a kernel crash using cuda-gdb.

1. Build the step_1 Nsight Eclipse Edition project using the hammer icon
2. Launch the project using the Run button (the connection had already been setup in step_0)
3. Watch the application crash and terminate in the Console window
4. Launch the project under cuda-gdb using the Debug button.
5. Wait until the application breaks in the kernel under the debugger. Then follow the same code inspection steps as in the CLI variant below.


cuda-workspace - step_1/imgserver_1.cu - Eclipse IDE
File Edit Navigate Search Project Run Window Help
Debug
Project Explorer
nanosleep() at 0x7fb7f80588
imgserver_1.cu
Variables
Breakpoints
Search CUDA Inform
Console
Problems
Executables
Debugger Console

```
launch_step_1 [C/C++ Remote Application]
  MatchFeatures [0] [device 0 (GV11B)]  (Signal) - CUDA_EXC
    CUDA Thread (0,0,0) Block (16,0,0)
      MatchFeatures() at imgserver_1.cu:1,449 0x55560b0d
    All Kernel Threads (1,024 Blocks of 8 Threads)
  Host Process [imgserver] [18025] [cores: 0,2,3]
    Thread #1 [imgserver] 18025 [core: 0] (Suspended : Cont
      nanosleep() at 0x7fb7f80588
      std::this_thread::sleep_for<long, std::ratio<1l, 1000l>
      RunProgram() at imgserver_1.cu:1,839 0x5555560534
      main() at imgserver_1.cu:1,970 0x5555560a10
    Thread #2 [imgserver] 18037 [core: 2] (Suspended : Cont
    Thread #3 [cuda-EvtHandlr] 18033 [core: 3] (Suspended :
    Thread #4 [imgserver] 18034 [core: 0] (Suspended : Cont
    Thread #5 [imgserver] 18036 [core: 0] (Suspended : Cont
  Remote Shell
  /usr/local/cuda-10.2/bin/cuda-gdb
```

```
        if (threadIdx.x == 0)
        {
            const unsigned int* pBaseFeature = nullptr;
            unsigned int testX = pTestFeature[0];
            unsigned int testY = pTestFeature[1];

#if defined(GDB_EXERCISE)
            unsigned int minDiff = 0;
#else
            unsigned int minDiff = UINT_MAX;
#endif // GDB_EXERCISE

            for (unsigned int i = 0; i < numBaseFeatures; ++i)
            {
                const unsigned int baseX = pBase[1 + FeatureVectorSizeInts * i + 0];
                const unsigned int baseY = pBase[1 + FeatureVectorSizeInts * i + 1];

                const unsigned int diff = __usad(baseX, testX, 0) + __usad(baseY, testY, 0);
                if (diff < minDiff)
                {
                    minDiff = diff;
                    pBaseFeature = &pBase[1 + FeatureVectorSizeInts * i + 2];
                }
            }

            for (unsigned int i = 0; i < FeatureVectorSizeInts - 2; ++i)
            {
                bestBaseFeature[i] = pBaseFeature[i];
            }
        }

        // all threads wait for the first thread to finish
        __syncthreads();

        // all threads compute the difference between the two selected features
        const unsigned int valueTest = pTestFeature[2 + threadIdx.x];
        const unsigned int valueBase = bestBaseFeature[threadIdx.x];
        const unsigned int diff = valueTest ^ valueBase;
```

```
[0] MatchFeatures(pTest=(
  (0,0,0)
  (1,0,0)
  (2,0,0)
  (3,0,0)
  (4,0,0)
  (5,0,0)
  (6,0,0)
  (7,0,0)
  (8,0,0)
  (9,0,0)
  (10,0,0)
  (11,0,0)
  (12,0,0)
  (13,0,0)
  (14,0,0)
  (15,0,0)
  (16,0,0)
  (17,0,0)
  (18,0,0)
  (19,0,0)
  (20,0,0)
  (21,0,0)
  (22,0,0)
  (23,0,0)
```

```
launch_step_1 [C/C++ Remote Application] Remote Shell
Loading ../img/image_00003.bmp...
Loading ../img/image_00004.bmp...
```

6. Fix the initialization, recompile and Run again to see that the application now doesn't crash anymore.

CLI Alternative:

1. Build the application on the CLI with

```
$ make clean
$ make step=1
```

2. Run the application and observe kernel crash, caught by checkCudaErrors() macro, leading to application termination

```
$ ./imgserver 50 100
Exercise step 1
Loading ../FlowersDataset/bmp/image_00001.bmp...
Loading ../FlowersDataset/bmp/image_00002.bmp...
Loading ../FlowersDataset/bmp/image_00003.bmp...
Loading ../FlowersDataset/bmp/image_00004.bmp...
Loading ../FlowersDataset/bmp/image_00005.bmp...
Loading ../FlowersDataset/bmp/image_00006.bmp...
Loading ../FlowersDataset/bmp/image_00007.bmp...
Loading ../FlowersDataset/bmp/image_00008.bmp...
Loading ../FlowersDataset/bmp/image_00009.bmp...
Loading ../FlowersDataset/bmp/image_00010.bmp...
Loading smiley.bmp...
checkCudaErrors() API error = 0077 (an illegal memory access was
encountered) from file <imgserver.cu>, line 1518.
freeglut  ERROR:  Internal <Window Enumeration> function called without
first calling 'glutInit'.
checkCudaErrors() API error = 0029 (driver shutting down) from file
```

```
<imgserver.cu>, line 560.
```

3. Run the application under cuda-gdb until it breaks at the device-side kernel exception. For the CLI, see below. For Nsight Eclipse Edition, select Debug as Remote Application.

```
$ /usr/local/cuda-10.2/bin/cuda-gdb ./imgserver 50 100
NVIDIA (R) CUDA Debugger
10.2 release
...
Reading symbols from ./imgserver...done.
(cuda-gdb) r
Starting program:
/media/WORK/p4/sw/devtools/Presentations/2020/GTC/labs/code/imgserver
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Exercise step 1
Loading ../FlowersDataset/bmp/image_00001.bmp...
...
Loading smiley.bmp...
[New Thread 0x7fffe7f0e700 (LWP 18699)]
...

CUDA Exception: Warp Illegal Address
The exception was triggered at PC 0x13fa1e0 (imgserver.cu:1411)

Thread 7 "imgserver" received signal CUDA_EXCEPTION_14, Warp Illegal Address.
[Switching focus to CUDA kernel 0, grid 12, block (141,0,0), thread (0,0,0),
device 0, sm 0, warp 1, lane 0]
0x00000000013fa1f0 in MatchFeatures<<<(1024,1,1),(8,1,1)>>>
(pTest=0x7fffbfe65400, pBase=0x7fffbfe00000, pResult=0x7fffbfe6f600) at
imgserver.cu:1411
1411                             bestBaseFeature[i] = pBaseFeature[i];
(cuda-gdb)
```

4. Inspect the pBaseFeature variable, detect that it’s set to 0x0. Inspect the code to see that it is set only under the (diff < minDiff) condition. Inspect minDiff to see that it is set (and initialized to) 0, so the condition will never evaluate to true. Inspect the initialization point to see that it must be fixed to minDiff = UINT_MAX.

```
(cuda-gdb) p pBaseFeature
$1 = (const unsigned int * @generic) 0x0
(cuda-gdb) p minDiff
$2 = 0
...
#if defined(GDB_EXERCISE)
        unsigned int minDiff = 0;
#else
        unsigned int minDiff = UINT_MAX;
#endif
        for (unsigned int i = 0; i < numBaseFeatures; ++i)
        {
            const unsigned int baseX = pBase[1 + FeatureVectorSizeInts
* i + 0];
            const unsigned int baseY = pBase[1 + FeatureVectorSizeInts
* i + 1];

            const unsigned int diff = __usad(baseX, testX, 0) +
__usad(baseY, testY, 0);
            if (diff < minDiff)
            {
                  minDiff = diff;
                  pBaseFeature = &pBase[1 + FeatureVectorSizeInts * i +
2];
            }
        }

        for (unsigned int i = 0; i < FeatureVectorSizeInts - 2; ++i)
        {
            bestBaseFeature[i] = pBaseFeature[i];
        }
```

5. Fix the initialization, recompile using step=1 and run again to see that the application doesn’t crash anymore.

Step 2: Compute Sanitizer*

* Compute Sanitizer is not available in CUDA 10.2. Skip this step if using CUDA 10.2.

Debugging unexpected application behavior using sanitizer’s memcheck and initcheck tools.

1. Build the application with step=2 (or the respective Nsight Eclipse Edition project), this equals the fixed version of the debugger step.

```
$ make clean
$ make step=2
```

2. Run the application and note that almost all classification steps will fail.

```
$ ./imgserver
Exercise step 2
...
Loading ../FlowersDataset/bmp/image_00010.bmp...
Loading smiley.bmp...
Received 9, Classified as 9 - MATCH
Received 3, Classified as 7 - FAIL
Received 8, Classified as 1 - FAIL
Received 5, Classified as 3 - FAIL
Received 4, Classified as 3 - FAIL
Received 1, Classified as 9 - FAIL
Received 9, Classified as 7 - FAIL
Received 1, Classified as 9 - FAIL
Received 0, Classified as 1 - FAIL
Received 0, Classified as 3 - FAIL
Received 7, Classified as 7 - MATCH
Received 1, Classified as 6 - FAIL
Received 3, Classified as 0 - FAIL
Received 6, Classified as 1 - FAIL
Received 0, Classified as 2 - FAIL
Received 7, Classified as 3 - FAIL
...
```

3. Run the application again under compute-sanitizer, terminate the app after a few classification steps were reported. Note that sanitizer reports no errors.

```
$ compute-sanitizer ./imgserver
========= COMPUTE-SANITIZER
Exercise step 2
Loading ../FlowersDataset/bmp/image_00001.bmp...
...
Loading smiley.bmp...
Received 9, Classified as 8 - FAIL
Received 3, Classified as 8 - FAIL
Received 8, Classified as 1 - FAIL
Received 5, Classified as 4 - FAIL
Received 4, Classified as 5 - FAIL
Received 1, Classified as 4 - FAIL
Received 9, Classified as 1 - FAIL
<manually terminate application>
========= ERROR SUMMARY: 0 errors
```

4. Note that Compute Sanitizer has four sub-tools (memcheck (default), initcheck, syncheck and racecheck). Once memcheck doesn’t report any errors, we can use the other three tools. We don’t know which one to start with, but for the sake of time, we can start with initcheck, which will show the prepared error.
5. Run the application again using Compute Ssanitizer’s initcheck tool until errors are reported, then terminate the app (e.g. by pressing Ctrl-C) and inspect the errors.

```
$ compute-sanitizer --tool initcheck ./imgserver
========= Uninitialized __global__ memory read of size 4 bytes
=========     at 0x140 in
/media/WORK/p4/sw/devtools/Presentations/2020/GTC/labs/code/imgserver.cu:1388:
MatchFeatures(unsigned int const *,unsigned int const *,unsigned int*)
=========     by thread (0,0,0) in block (367,0,0)
=========     Address 0x7f4a17a68d5c
=========     Saved host backtrace up to driver entry point at kernel launch
time
=========     Host Frame:cuLaunchKernel [0x7f4a4713b546]
=========                 in /usr/lib/x86_64-linux-gnu/libcuda.so
=========     Host Frame: [0x443312]
=========                 in
/media/WORK/p4/sw/devtools/Presentations/2020/GTC/labs/code/./imgserver
=========     Host Frame: [0x443507]
=========                 in
/media/WORK/p4/sw/devtools/Presentations/2020/GTC/labs/code/./imgserver
...
=========     Host Frame: [0x7f4a4a313c80]
=========                 in /usr/lib/x86_64-linux-gnu/libstdc++.so.6
=========     Host Frame: [0x7f4a4a7e86ba]
=========                 in /lib/x86_64-linux-gnu/libpthread.so.0
=========     Host Frame:clone [0x7f4a49d8241d]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= Internal Sanitizer Warning: Error buffer overflow has been detected.
Some records have been dropped
=========
Received 9, Classified as 9 - MATCH
========= ERROR SUMMARY: 2570 errors
```

6. Inspect the referenced code location (exact line may change from what is listed in this document). Apparently the feature we read from is not properly initialized. From the comment, we see which feature entry is selected by each block.

```
        // from the vector, select the feature assigned to this block
        //            vvvvv
        // [<num>|..0..|..1..|..2..|..3..|..numTestFeatures-1..]
        //  <--->                  <--->
        //  sizeof(unsigned int)   sizeof(int) * FeatureVectorSizeInts
        const unsigned int* pTestFeature = &pTest[1 + FeatureVectorSizeInts *
blockIdx.x];
        __shared__ unsigned int bestBaseFeature[FeatureVectorSizeInts - 2];

        // the first thread in the block searches the best-matching feature in
the base features vector
        if (threadIdx.x == 0)
        {
                const unsigned int* pBaseFeature = nullptr;
                unsigned int testX = pTestFeature[0];
```

7. We need to inspect the location where the features are written, which is in the ExtractFeatures kernel. In the code and comment, we see that the written entry is supposed to be selected based on the fvi variable (feature vector index), but we failed to multiply the entry size with this index, so we always write to only the second feature, for all blocks.

```
        // atomically select the next feature vector index (fvi)
        const unsigned int fvi = atomicAdd(&pFeatureVectors[0], 1);
        if (fvi >= MaxFeatureVectors)
        {
                // no more features available, mark this in the output image
                pOut[outIdx] = make_uchar4(0, 255, 0, 255);
        }
        else
        {
                // compute the feature vector for this keypoint

                // we want to use the same pseudo-random number sequence for
each keypoint,
                // so we initialize the cuRand state every time with the same
seeds
                curandState_t randState;
                curand_init(0, 0, 0, &randState);

                // keypoints are randomly selected in a 65x65 search are
                constexpr int patchRadius = 32;

                // from the vector, select the feature assigned to this block
                //           fvi
                //           vvvvv
                // [<num>|..0..|..1..|..2..|..3..|..numTestFeatures-1..]
                //  <--->                  <--->
                //  sizeof(unsigned int)   sizeof(int) * FeatureVectorSizeInts
#if defined(GDB_EXERCISE) || defined(SANITIZER_EXERCISE)
                unsigned int* pFeature = &pFeatureVectors[1 +
FeatureVectorSizeInts];
#else
             unsigned int* pFeature = &pFeatureVectors[1 +
FeatureVectorSizeInts * fvi];
#endif
```

8. After fixing the bug and re-compiling using step=2, we see that initcheck reports no errors anymore and classification is working as expected.

```
compute-sanitizer --tool initcheck ./imgserver
========= COMPUTE-SANITIZER
Exercise step 3
Loading ../FlowersDataset/bmp/image_00001.bmp...
...
Loading smiley.bmp...
Received 9, Classified as 9 - MATCH
Received 3, Classified as 3 - MATCH
Received 8, Classified as 8 - MATCH
Received 1, Classified as 1 - MATCH
Received 5, Classified as 5 - MATCH
Received 1, Classified as 1 - MATCH
Received 9, Classified as 9 - MATCH
Received 4, Classified as 4 - MATCH
========= ERROR SUMMARY: 0 errors
```

9. We can now run the app without Compute Sanitizer to see the same working behavior. However, it's still too slow, so many user requests are over the time threshold, resulting in sad smileys rather than flower images. This leads us to the performance analysis part of the training.

Steps 3-5: Nsight Systems

Step 3-4 Video: https://developer.nvidia.com/gtc/2020/video/t21395-3

Step 3: Analyze original application behavior

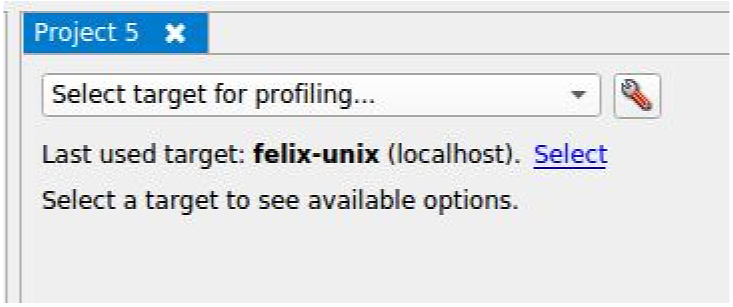
1. Build the first application step without functional bugs

```
$ make clean
$ make step=3
```

2. Set the correct kernel perf event paranoid flag, to collect as much CPU data as possible

```
$ sudo sh -c 'echo 1 > /proc/sys/kernel/perf_event_paranoid'
```

3. Start Nsight Systems and open a new project. Connect to your localhost.



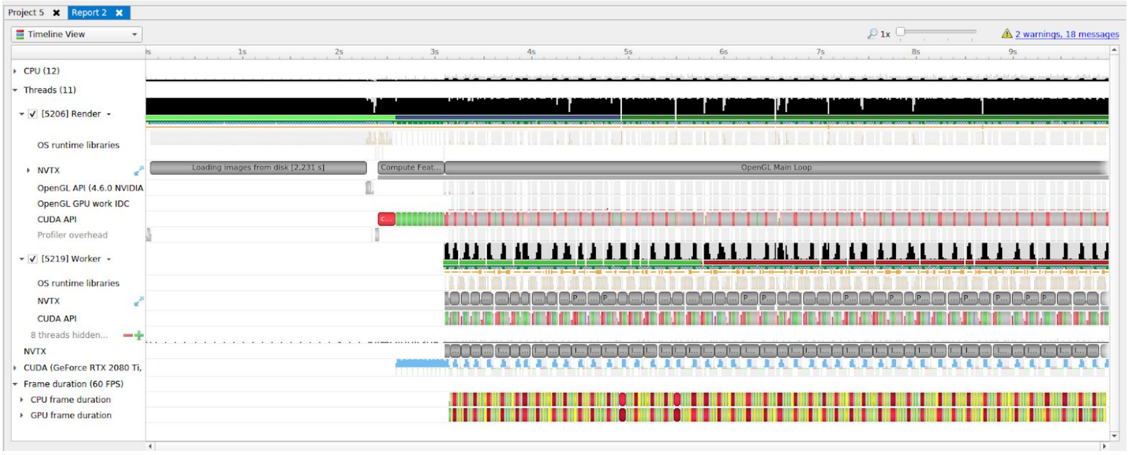
Project 5 ✖

Select target for profiling... ▾

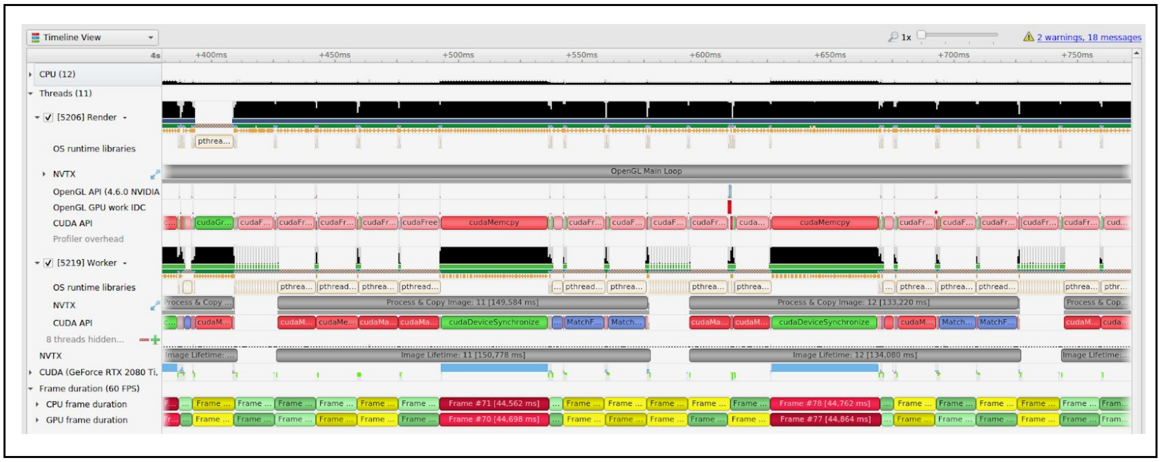
Last used target: **felix-unix** (localhost). Select

Select a target to see available options.

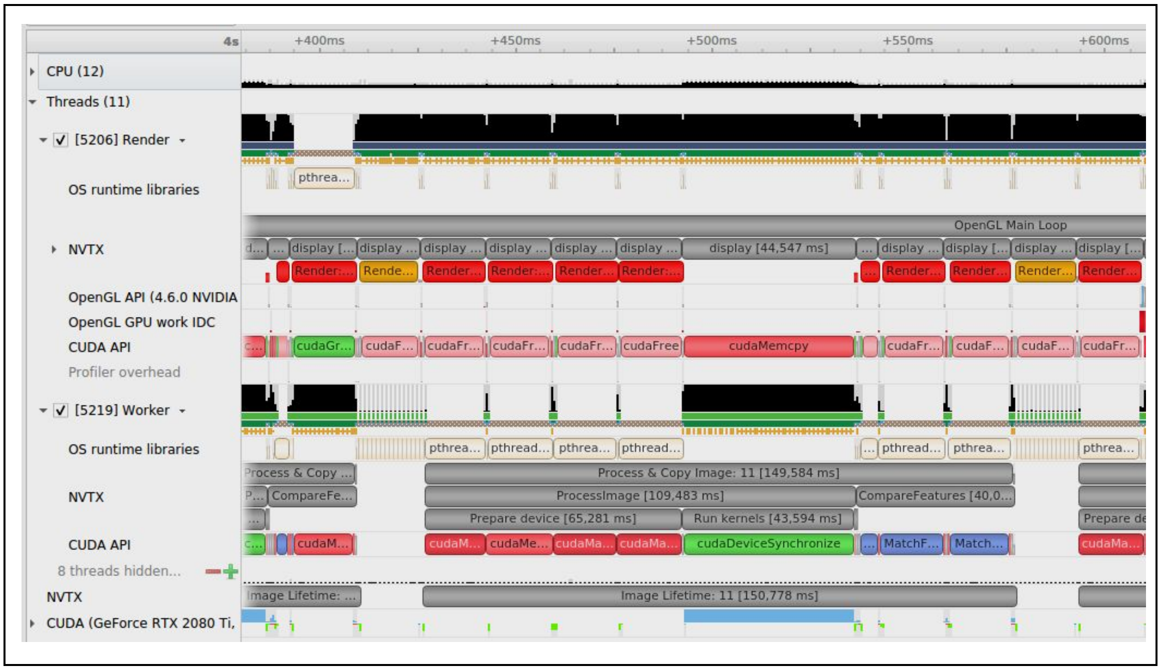
4. Once connected, fill in the project details.
    a. Enable Collect Call stacks of executing thread, CUDA, OpenGL, NVTX and OS lib trace enabled,
    b. Fill in target application and working dir
    App: /home/nvidia/cuda-workspace/step_3/imgserver 70 100
    WD: /home/nvidia/cuda-workspace/step_3
5. Launch data collection with Start and inspect the resulting trace. Once the application ends, stop data collection.
During data collection, most requests should exceed the threshold, resulting in sad smiley renderings. The trace will show 2-3 threads (Worker and Render) in three phases: Loading images from disk, Compute Features DB and the actual request processing. We are interested in optimizing the last phase (OpenGL Main Loop).



6. Zoom into the last phase until you see the Process & Copy image patterns. We see kernel execution (light blue) on the GPU, but we also see that the Render thread spends a considerable amount of time in cudaFree and cudaMemcpy API calls.



7. Zooming in further into the Render thread and expanding the NVTX rows provides more details: While the next request is processed on the Worker thread, the Render thread continuously calls display() which tries to Render to the output grid. During each Render call, we seem to free (and allocate, when zooming in further) device memory.



8. It would be smarter to allocate this buffer only once, and free it once all requests are done, since all request images are of the same size in our application. This is done for this buffer (and several others) in the next step.

Step 4: Buffer reuse to minimize cudaMalloc/cudaFree

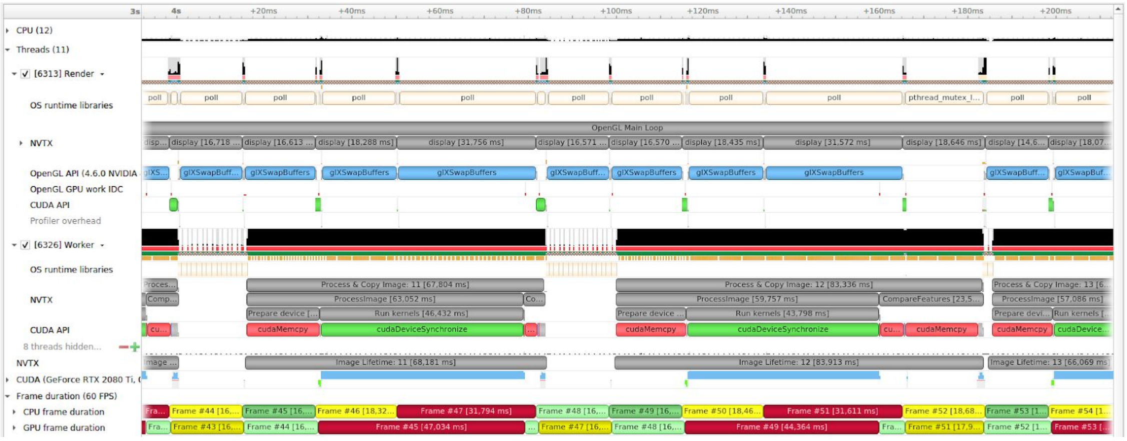
1. Build the next step, which enables device buffer reuse

```
$ make clean
$ make step=4
```

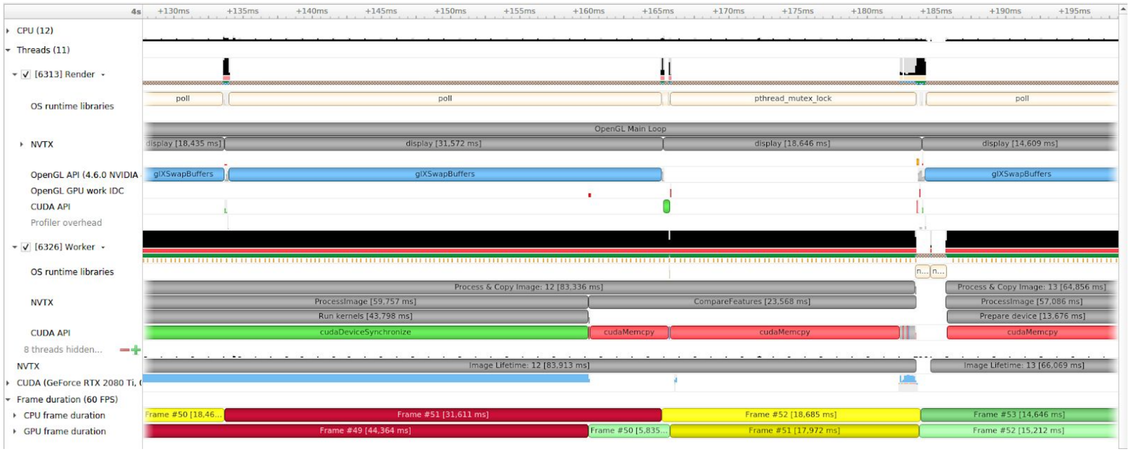
2. Repeat the steps from the previous step 3 to collect a new trace in Nsight Systems. If Nsight Systems is still open, simply select the same Project and click Start. Then inspect

the collected trace. We see that the cudaFree calls are gone and so are several cudaMalloc calls (e.g. during the “Prepare device” NVTX range).

Desktop:



3. Even though the Image Lifetime NVTX ranges became faster, we still see the large gaps between the processing phases, and we still see the Render thread calling display() continuously. There are various CUDA and OpenGL API calls happening, many of which can interfere with our Worker thread. For example, in the screenshot we see that several cudaMemcpy calls are extremely long even though they transfer the same amount of data as very small ones in that phase.



4. The solution to this is simple: since our application is a very strict pipeline (Source request, Worker processing, Render output), there is no need for the Render thread to become active until the Worker thread is done processing the request and adding it to the transfer queue.

Step 5: Rendering on Demand

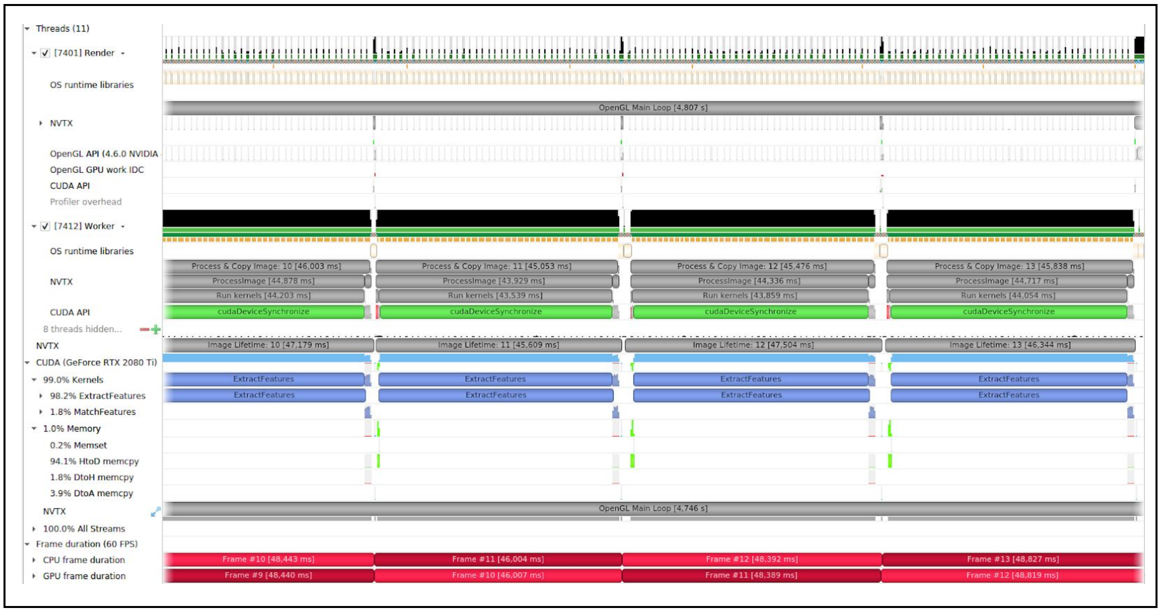
Step 5-7 Video: https://developer.nvidia.com/gtc/2020/video/t21395-4

1. Build the next step, which updates the Render thread to become active only when a new request was processed

```
$ make clean
$ make step=5
```

5. Repeat the steps from the previous step 4 to collect a new trace in Nsight Systems. If Nsight Systems is still open, simply select the same Project and click Start. During the capture, we should now see some of the requests being served in time, which has them render the proper image to the output window. Afterwards, inspect the collected trace.

6. We see that each iteration is now executing much more regularly. Also, the Render thread only becomes active between two Process & Copy ranges. The average frame duration reported by Nsight Systems (which is somewhat equal to our Image Lifetime NVTX range) is ~50ms. The ExtractFeatures kernel is now the clear performance bottleneck. In the next exercise, we will use Nsight Compute to optimize this kernel.



Steps 6-7: Nsight Compute

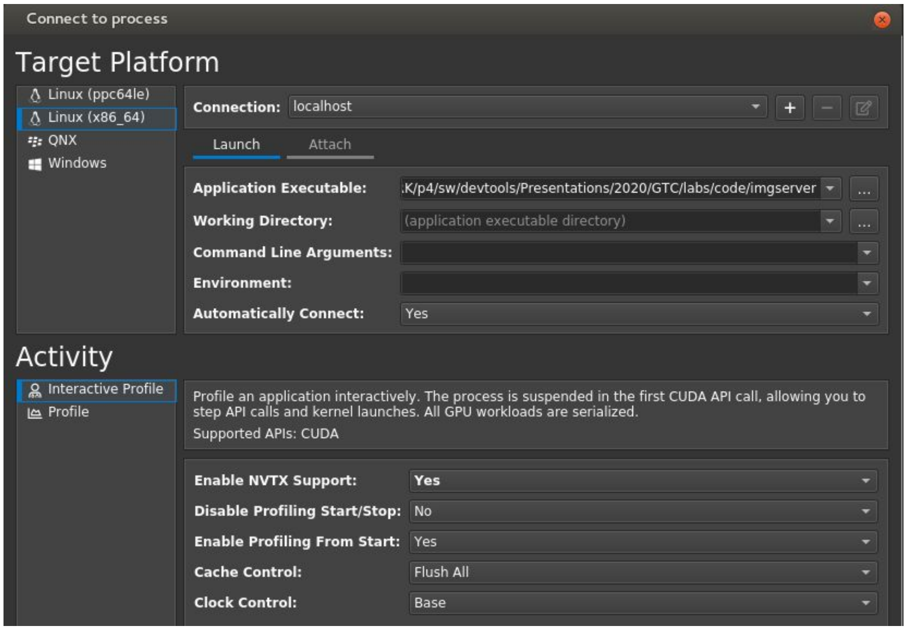
Analyzing and optimizing kernel performance using Nsight Compute interactive profiling.

Step 6.1: Analyze ExtractFeatures kernel

1. Re-use the application built as part of the last Nsight Systems exercise, or rebuild it using:

```
$ make clean
$ make step=5
```

2. Nsight Systems has shown that the ExtractFeatures kernel is the primary bottleneck, so we start by analyzing that kernel. Start the Nsight Compute UI. In CUDA 10.2, the tool is called `nv-nsight-cu`. In CUDA 11 it is `ncu-ui`. Open a new project file and fill in the Application Executable to point to step_5/imgserver. Set Command Line Arguments to “50 100”, then Launch the application.



Connection: localhost

Application Executable: .K/p4/sw/devtools/Presentations/2020/GTC/labs/code/imgserver

Working Directory: (application executable directory)

Command Line Arguments:

Environment:

Automatically Connect: Yes

Profile an application interactively. The process is suspended in the first CUDA API call, allowing you to step API calls and kernel launches. All GPU workloads are serialized.

Supported APIs: CUDA

Enable NVTX Support: Yes

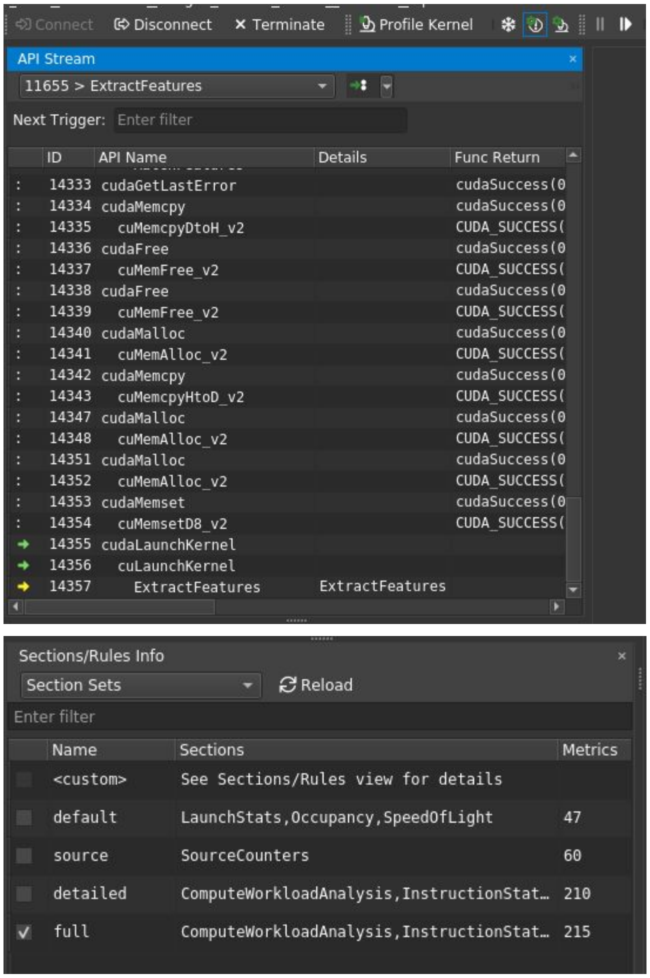
Disable Profiling Start/Stop: No

Enable Profiling From Start: Yes

Cache Control: Flush All

Clock Control: Base

3. Once connected, the app is suspended in the first CUDA API call. Select “Run to Next Kernel” until you are at an ExtractFeatures kernel (it should be the very first one).Ensure that the “full” section set is enabled before selecting “Profile Kernel”.



11655 > ExtractFeatures

Next Trigger: Enter filter

	ID	API Name	Details	Func Return
:	14333	cudaGetLastError		cudaSuccess(0
:	14334	cudaMemcpy		cudaSuccess(0
:	14335	cuMemcpyDtoH_v2		CUDA_SUCCESS(
:	14336	cudaFree		cudaSuccess(0
:	14337	cuMemFree_v2		CUDA_SUCCESS(
:	14338	cudaFree		cudaSuccess(0
:	14339	cuMemFree_v2		CUDA_SUCCESS(
:	14340	cudaMalloc		cudaSuccess(0
:	14341	cuMemAlloc_v2		CUDA_SUCCESS(
:	14342	cudaMemcpy		cudaSuccess(0
:	14343	cuMemcpyHtoD_v2		CUDA_SUCCESS(
:	14347	cudaMalloc		cudaSuccess(0
:	14348	cuMemAlloc_v2		CUDA_SUCCESS(
:	14351	cudaMalloc		cudaSuccess(0
:	14352	cuMemAlloc_v2		CUDA_SUCCESS(
:	14353	cudaMemset		cudaSuccess(0
:	14354	cuMemsetD8_v2		CUDA_SUCCESS(
➜	14355	cudaLaunchKernel		
➜	14356	cuLaunchKernel		
➜	14357	ExtractFeatures	ExtractFeatures	

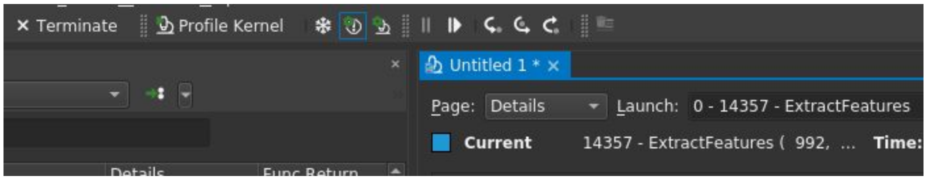
Sections/Rules Info

Section Sets   Reload

Enter filter

	Name	Sections	Metrics
□	<custom>	See Sections/Rules view for details	
□	default	LaunchStats,Occupancy,SpeedOfLight	47
□	source	SourceCounters	60
□	detailed	ComputeWorkloadAnalysis,InstructionStat…	210
☑	full	ComputeWorkloadAnalysis,InstructionStat…	215

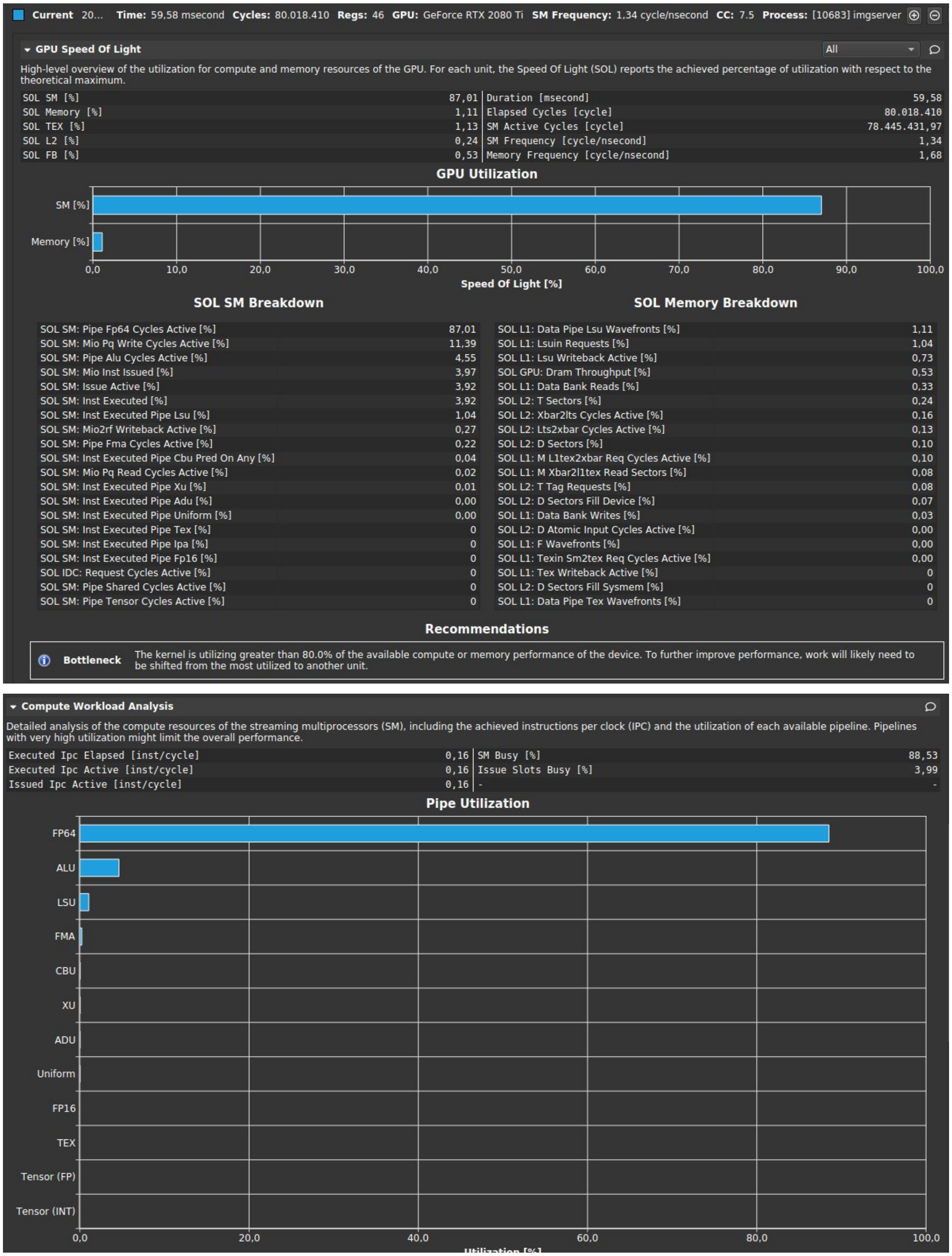
4. After profiling is done and the report was created, we can Terminate the target application and start analyzing the collected report.



Page: Details   Launch: 0 - 14357 - ExtractFeatures

Current   14357 - ExtractFeatures (992, ...   Time:

5. We can see that the kernel is heavily compute bound, utilizing the SM units > 80%. We can see from the SOL SM Breakdown that the Fp64 (64bit/double floating point math) pipeline is by far the biggest contributor. Similar data can be found in the Compute Workload Analysis section



Current 20... **Time:** 59,58 msecond **Cycles:** 80.018.410 **Regs:** 46 **GPU:** GeForce RTX 2080 Ti **SM Frequency:** 1,34 cycle/nsecond **CC:** 7.5 **Process:** [10683] imgserver

GPU Speed Of Light

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.

SOL SM [%]	87,01	Duration [msecond]	59,58
SOL Memory [%]	1,11	Elapsed Cycles [cycle]	80.018.410
SOL TEX [%]	1,13	SM Active Cycles [cycle]	78.445.431,97
SOL L2 [%]	0,24	SM Frequency [cycle/nsecond]	1,34
SOL FB [%]	0,53	Memory Frequency [cycle/nsecond]	1,68

SOL SM Breakdown

SOL SM: Pipe Fp64 Cycles Active [%]	87,01
SOL SM: Mio Pq Write Cycles Active [%]	11,39
SOL SM: Pipe Alu Cycles Active [%]	4,55
SOL SM: Mio Inst Issued [%]	3,97
SOL SM: Issue Active [%]	3,92
SOL SM: Inst Executed [%]	3,92
SOL SM: Inst Executed Pipe Lsu [%]	1,04
SOL SM: Mio2rf Writeback Active [%]	0,27
SOL SM: Pipe Fma Cycles Active [%]	0,22
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	0,04
SOL SM: Mio Pq Read Cycles Active [%]	0,02
SOL SM: Inst Executed Pipe Xu [%]	0,01
SOL SM: Inst Executed Pipe Adu [%]	0,00
SOL SM: Inst Executed Pipe Uniform [%]	0,00
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL IDC: Request Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	1,11
SOL L1: Lsuin Requests [%]	1,04
SOL L1: Lsu Writeback Active [%]	0,73
SOL GPU: Dram Throughput [%]	0,53
SOL L1: Data Bank Reads [%]	0,33
SOL L2: T Sectors [%]	0,24
SOL L2: Xbar2lts Cycles Active [%]	0,16
SOL L2: Lts2xbar Cycles Active [%]	0,13
SOL L2: D Sectors [%]	0,10
SOL L1: M L1tex2xbar Req Cycles Active [%]	0,10
SOL L1: M Xbar2l1tex Read Sectors [%]	0,08
SOL L2: T Tag Requests [%]	0,08
SOL L2: D Sectors Fill Device [%]	0,07
SOL L1: Data Bank Writes [%]	0,03
SOL L2: D Atomic Input Cycles Active [%]	0,00
SOL L1: F Wavefronts [%]	0,00
SOL L1: Texin Sm2tex Req Cycles Active [%]	0,00
SOL L1: Tex Writeback Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0

Recommendations

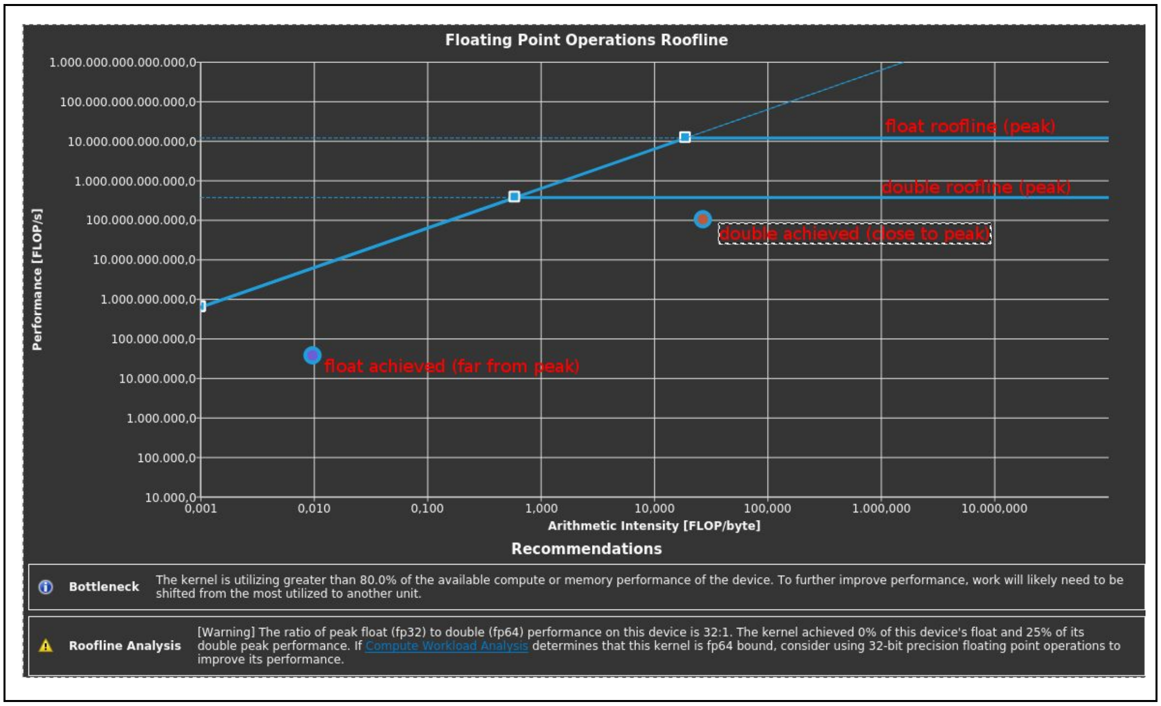
Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	0,16	SM Busy [%]	88,53
Executed Ipc Active [inst/cycle]	0,16	Issue Slots Busy [%]	3,99
Issued Ipc Active [inst/cycle]	0,16	-	-

6. If using Nsight Compute from CUDA 11.0, the roofline chart guides the user to use float instead of double due to the much 32:1 peak perf ratio between 32 and 64bit units. Red labels added in screenshot for explanatory purposes.

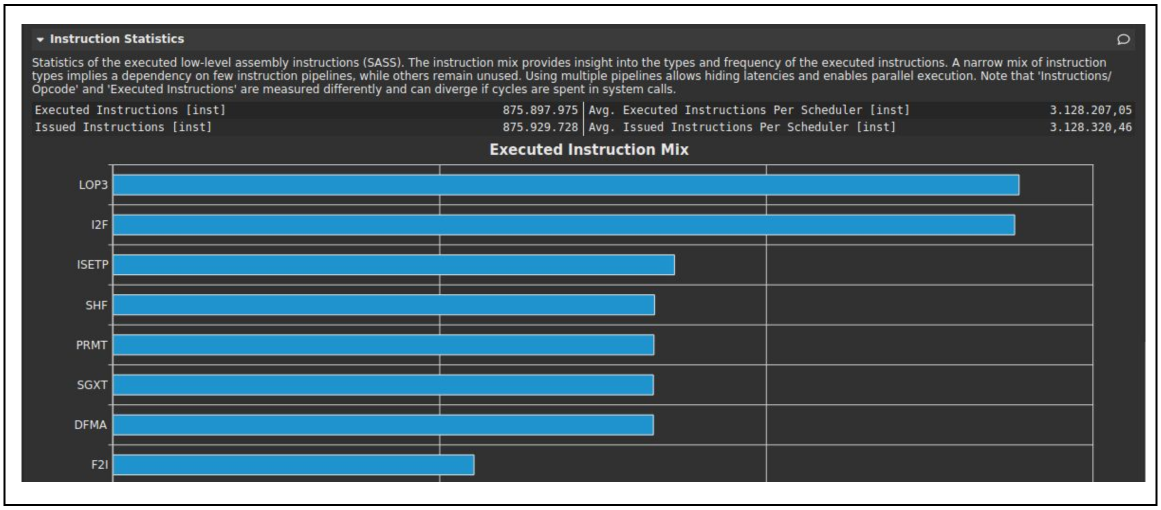


Recommendations

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit.

Roofline Analysis [Warning] The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 0% of this device's float and 25% of its double peak performance. If Compute Workload Analysis determines that this kernel is fp64 bound, consider using 32-bit precision floating point operations to improve its performance.

7. We can use the Instruction Statistics section to learn more about the types and counts of instructions executed, which show a large number of floating point related instructions for this kernel

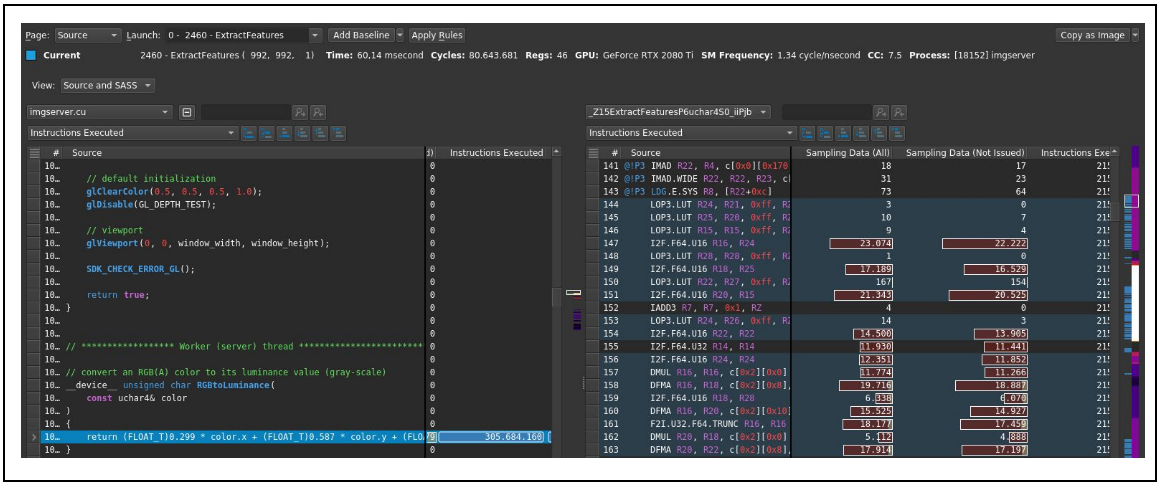


Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

```
Executed Instructions [inst]                 875.897.975 | Avg. Executed Instructions Per Scheduler [inst]     3.128.207,05
Issued Instructions [inst]                   875.929.728 | Avg. Issued Instructions Per Scheduler [inst]       3.128.320,46
```

8. Switching to the Source page using the Page dropdown on the top of the report, we can see where in the code those instructions are executed. For that, make sure that “Instructions Executed” is selected in the metrics drop down and then use the “Move to the row with the highest value” button from the navigation buttons. We see that most instructions are executed as part of the RGB to luminance/grayscale conversion



Page: Source Launch: 0 - 2460 - ExtractFeatures

Current 2460 - ExtractFeatures (992, 992, 1) **Time:** 60,14 msecond **Cycles:** 80.643.681 **Regs:** 46 **GPU:** GeForce RTX 2080 Ti **SM Frequency:** 1,34 cycle/nsecond **CC:** 7.5 **Process:** [18152] imgserver

View: Source and SASS

imgserver.cu — Instructions Executed

```
Source                                                                          Instructions Executed
                                                                                0
    // default initialization                                                   0
    glClearColor(0.5, 0.5, 0.5, 1.0);                                           0
    glDisable(GL_DEPTH_TEST);                                                   0
                                                                                0
    // viewport                                                                 0
    glViewport(0, 0, window_width, window_height);                              0
                                                                                0
    SDK_CHECK_ERROR_GL();                                                       0
                                                                                0
    return true;                                                                0
}                                                                               0
                                                                                0
                                                                                0
// ***************** Worker (server) thread ************************           0
                                                                                0
// convert an RGB(A) color to its luminance value (gray-scale)                  0
__device__ unsigned char RGBtoLuminance(                                        0
    const uchar4& color                                                         0
)                                                                               0
{                                                                               0
    return (FLOAT_T)0.299 * color.x + (FLOAT_T)0.587 * color.y + (FLO           305.684.160
}                                                                               0
```

_Z15ExtractFeaturesP6uchar4S0_iiPjb — Instructions Executed

#	Source	Sampling Data (All)	Sampling Data (Not Issued)	Instructions Exe
141	@!P3 IMAD R22, R4, c[0x0][0x170	18	17	215
142	@!P3 IMAD.WIDE R22, R22, R23, c	31	23	215
143	@!P3 LDG.E.SYS R8, [R22+0xc]	73	64	215
144	LOP3.LUT R24, R21, 0xff, RZ	3	0	215
145	LOP3.LUT R25, R20, 0xff, RZ	10	7	215
146	LOP3.LUT R15, R15, 0xff, RZ	9	4	215
147	I2F.F64.U16 R16, R24	23.074	22.222	215
148	LOP3.LUT R28, R28, 0xff, RZ	1	0	215
149	I2F.F64.U16 R18, R25	17.189	16.529	215
150	LOP3.LUT R22, R27, 0xff, RZ	167	154	215
151	I2F.F64.U16 R20, R15	21.343	20.525	215
152	IADD3 R7, R7, 0x1, RZ	4	0	215
153	LOP3.LUT R24, R26, 0xff, RZ	14	3	215
154	I2F.F64.U16 R22, R22	14.500	13.905	215
155	I2F.F64.U32 R14, R14	11.930	11.441	215
156	I2F.F64.U16 R24, R24	12.351	11.852	215
157	DMUL R16, R16, c[0x2][0x0]	11.774	11.266	215
158	DFMA R16, R18, c[0x2][0x8],	19.716	18.887	215
159	I2F.F64.U16 R18, R28	6.338	6.070	215
160	DFMA R16, R20, c[0x2][0x10]	15.525	14.927	215
161	F2I.U32.F64.TRUNC R16, R16	18.177	17.459	215
162	DMUL R20, R18, c[0x2][0x0]	5.112	4.888	215
163	DFMA R20, R22, c[0x2][0x8],	17.914	17.197	215

9. On the left side, we see the CUDA C code, with the correlated SASS highlighted on the right side. Checking the definition of FLOAT_T in our code, we see that it is currently defined to double, which is why we do 64bit/double math right now.

```
#if defined(KERNEL_OPT_1)
       #define FLOAT_T double
#else
       #define FLOAT_T float
#endif
```

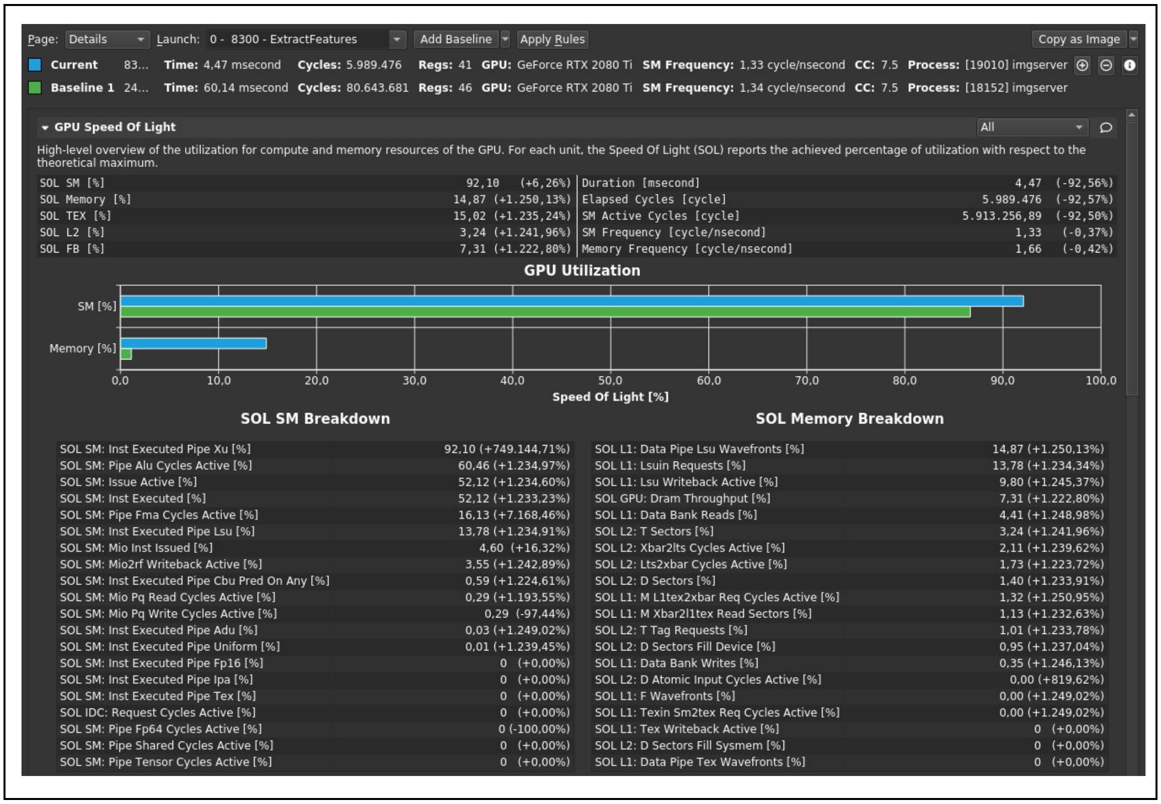
10. Since we don't really need double-precision, we can use 32bit/float math instead, given that we have a 32:1 ratio for float to double units on Turing RTX GPUs (for desktop, the user is notified of this by the roofline analysis).

Step 6.2: Analyze the 32bit float math ExtractFeatures kernel

1. Build the optimized kernel using 32bit instead of 64bit math using:

```
$ make clean
$ make step=6
```

2. Then, repeat the steps from the previous exercise 6.1 to collect the data for the optimized ExtractFeatures kernel in a second report.
3. We can briefly go back to the first report, select "Add Baseline", and then switch to the current once again. We now see all data from the first, unoptimized kernel compared against the Current (blue) kernel. Most notably, it is 92% faster (Duration), and doesn't use 64bit (Fp64) math anymore. Since we are blocked less by the fp64 pipeline, we now have a 12x better memory units utilization.



Current 83... **Time:** 4,47 msecond **Cycles:** 5.989.476 **Regs:** 41 **GPU:** GeForce RTX 2080 Ti **SM Frequency:** 1,33 cycle/nsecond **CC:** 7.5 **Process:** [19010] imgserver

Baseline 1 24... **Time:** 60,14 msecond **Cycles:** 80.643.681 **Regs:** 46 **GPU:** GeForce RTX 2080 Ti **SM Frequency:** 1,34 cycle/nsecond **CC:** 7.5 **Process:** [18152] imgserver

GPU Speed Of Light

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.

SOL SM [%]	92,10 (+6,26%)	Duration [msecond]	4,47 (-92,56%)
SOL Memory [%]	14,87 (+1.250,13%)	Elapsed Cycles [cycle]	5.989.476 (-92,57%)
SOL TEX [%]	15,02 (+1.235,24%)	SM Active Cycles [cycle]	5.913.256,89 (-92,50%)
SOL L2 [%]	3,24 (+1.241,96%)	SM Frequency [cycle/nsecond]	1,33 (-0,37%)
SOL FB [%]	7,31 (+1.222,80%)	Memory Frequency [cycle/nsecond]	1,66 (-0,42%)

GPU Utilization

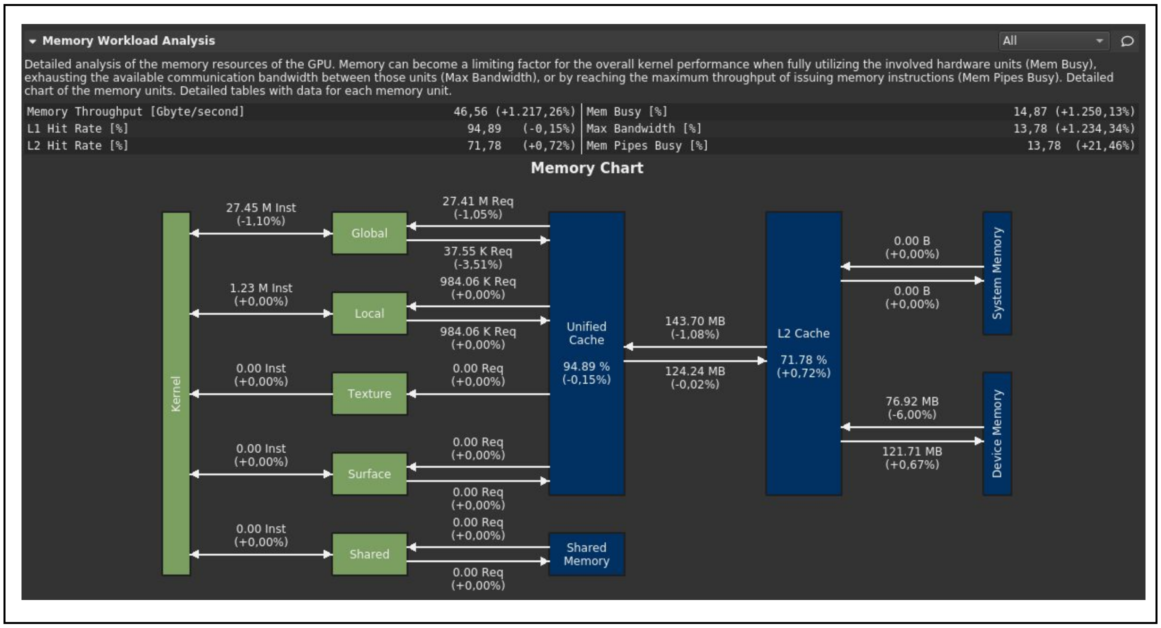
SOL SM Breakdown

SOL SM: Inst Executed Pipe Xu [%]	92,10 (+749.144,71%)
SOL SM: Pipe Alu Cycles Active [%]	60,46 (+1.234,97%)
SOL SM: Issue Active [%]	52,12 (+1.234,60%)
SOL SM: Inst Executed [%]	52,12 (+1.233,23%)
SOL SM: Pipe Fma Cycles Active [%]	16,13 (+7.168,46%)
SOL SM: Inst Executed Pipe Lsu [%]	13,78 (+1.234,91%)
SOL SM: Mio Inst Issued [%]	4,60 (+16,32%)
SOL SM: Mio2rf Writeback Active [%]	3,55 (+1.242,89%)
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	0,59 (+1.224,61%)
SOL SM: Mio Pq Read Cycles Active [%]	0,29 (+1.193,55%)
SOL SM: Mio Pq Write Cycles Active [%]	0,29 (-97,44%)
SOL SM: Inst Executed Pipe Adu [%]	0,03 (+1.249,02%)
SOL SM: Inst Executed Pipe Uniform [%]	0,01 (+1.239,45%)
SOL SM: Inst Executed Pipe Fp16 [%]	0 (+0,00%)
SOL SM: Inst Executed Pipe Ipa [%]	0 (+0,00%)
SOL SM: Inst Executed Pipe Tex [%]	0 (+0,00%)
SOL IDC: Request Cycles Active [%]	0 (+0,00%)
SOL SM: Pipe Fp64 Cycles Active [%]	0 (-100,00%)
SOL SM: Pipe Shared Cycles Active [%]	0 (+0,00%)
SOL SM: Pipe Tensor Cycles Active [%]	0 (+0,00%)

SOL Memory Breakdown

SOL L1: Data Pipe Lsu Wavefronts [%]	14,87 (+1.250,13%)
SOL L1: Lsuin Requests [%]	13,78 (+1.234,34%)
SOL L1: Lsu Writeback Active [%]	9,80 (+1.245,37%)
SOL GPU: Dram Throughput [%]	7,31 (+1.222,80%)
SOL L1: Data Bank Reads [%]	4,41 (+1.248,98%)
SOL L2: T Sectors [%]	3,24 (+1.241,96%)
SOL L2: Xbar2lts Cycles Active [%]	2,11 (+1.239,62%)
SOL L2: Lts2xbar Cycles Active [%]	1,73 (+1.223,72%)
SOL L2: D Sectors [%]	1,40 (+1.233,91%)
SOL L1: M L1tex2xbar Req Cycles Active [%]	1,32 (+1.250,95%)
SOL L1: M Xbar2l1tex Read Sectors [%]	1,13 (+1.232,63%)
SOL L2: T Tag Requests [%]	1,01 (+1.233,78%)
SOL L2: D Sectors Fill Device [%]	0,95 (+1.237,04%)
SOL L1: Data Bank Writes [%]	0,35 (+1.246,13%)
SOL L2: D Atomic Input Cycles Active [%]	0,00 (+819,62%)
SOL L1: F Wavefronts [%]	0,00 (+1.249,02%)
SOL L1: Texin Sm2tex Req Cycles Active [%]	0,00 (+1.249,02%)
SOL L1: Tex Writeback Active [%]	0 (+0,00%)
SOL L2: D Sectors Fill Sysmem [%]	0 (+0,00%)
SOL L1: Data Pipe Tex Wavefronts [%]	0 (+0,00%)

4. Inspecting the Memory Workload Analysis section, we see that we have a high cache hit rate in both L1 and L2, indicating that we re-read the same global memory multiple times.



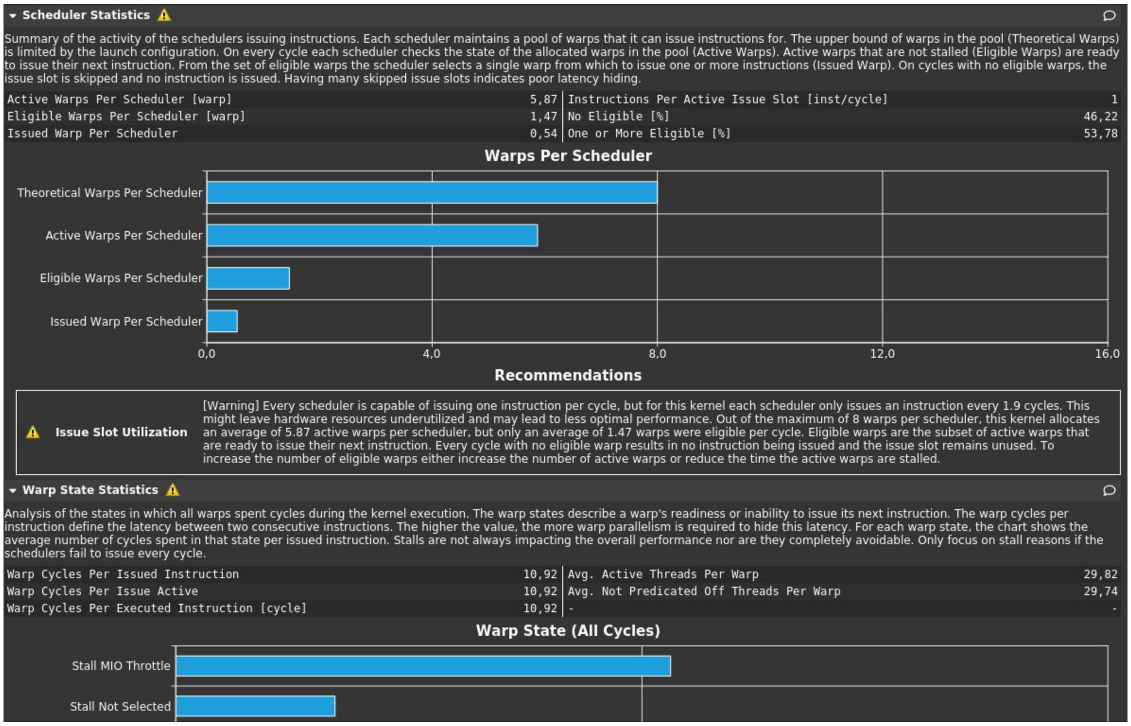
Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	46,56 (+1.217,26%)	Mem Busy [%]	14,87 (+1.250,13%)
L1 Hit Rate [%]	94,89 (-0,15%)	Max Bandwidth [%]	13,78 (+1.234,34%)
L2 Hit Rate [%]	71,78 (+0,72%)	Mem Pipes Busy [%]	13,78 (+21,46%)

Memory Chart

5. Analyzing the two sections with analysis recommendations, Scheduler and Warp State Statistics, we see that we issue only every second cycle, instead of every cycle, and that we are primarily stalled by MIO Throttle, which can also hint towards math instructions (see rule message).



Scheduler Statistics ⚠

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	5,87	Instructions Per Active Issue Slot [inst/cycle]	1
Eligible Warps Per Scheduler [warp]	1,47	No Eligible [%]	46,22
Issued Warp Per Scheduler	0,54	One or More Eligible [%]	53,78

Warps Per Scheduler

Theoretical Warps Per Scheduler
Active Warps Per Scheduler
Eligible Warps Per Scheduler
Issued Warp Per Scheduler
0,0
4,0
8,0
12,0
16,0

Recommendations

⚠ **Issue Slot Utilization**

[Warning] Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 1.9 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 8 warps per scheduler, this kernel allocates an average of 5.87 active warps per scheduler, but only an average of 1.47 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

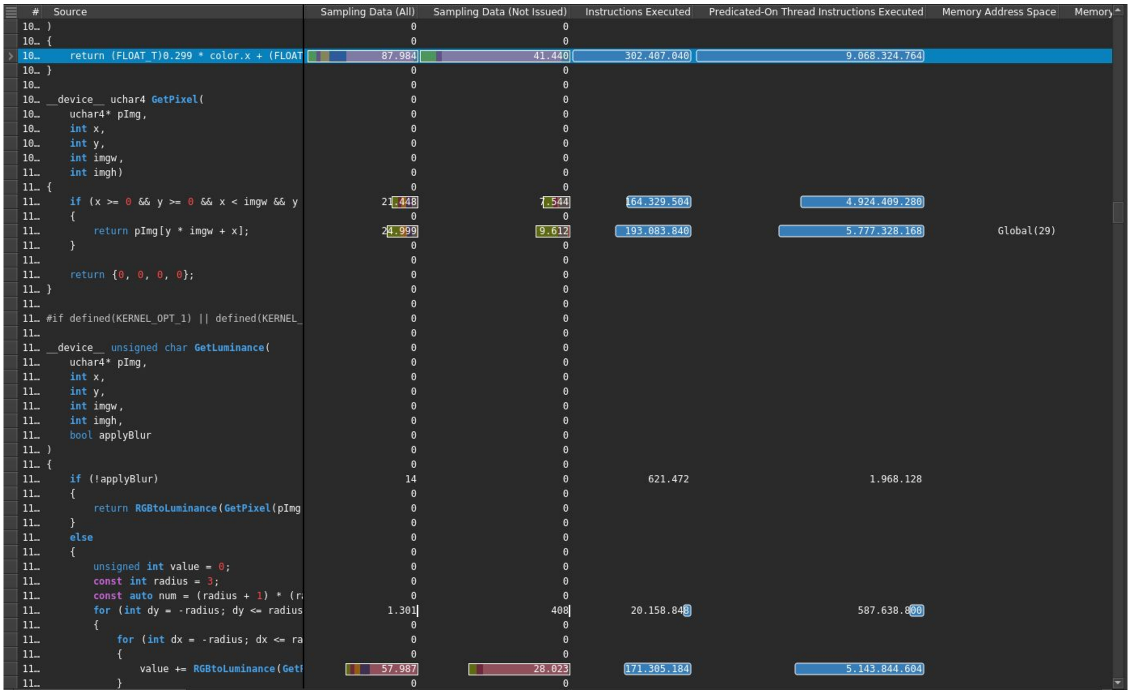
Warp State Statistics ⚠

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle.

Warp Cycles Per Issued Instruction	10,92	Avg. Active Threads Per Warp	29,82
Warp Cycles Per Issue Active	10,92	Avg. Not Predicated Off Threads Per Warp	29,74
Warp Cycles Per Executed Instruction [cycle]	10,92	-	-

Warp State (All Cycles)

6. Moving to the Source page and selecting "Sampling Data (All)" from the metrics drop down, we can see when navigating to the top N stall locations that most of them are within the functions reading pixels, converting them to luminance and applying gaussian blur.



#	Source	Sampling Data (All)	Sampling Data (Not Issued)	Instructions Executed	Predicated-On Thread Instructions Executed	Memory Address Space	Memory
10…	)	0	0				
10…	{	0	0				
10…	return (FLOAT_T)0.299 * color.x + (FLOAT	87.984	41.440	302.407.040	9.068.324.764		
10…	}	0	0				
10…		0	0				
10…	__device__ uchar4 GetPixel(	0	0				
10…	uchar4* pImg,	0	0				
10…	int x,	0	0				
10…	int y,	0	0				
10…	int imgw,	0	0				
11…	int imgh)	0	0				
11…	{	0	0				
11…	if (x >= 0 && y >= 0 && x < imgw && y	21.448	7.544	164.329.504	4.924.409.280		
11…	{	0	0				
11…	return pImg[y * imgw + x];	24.999	9.612	193.083.840	5.777.328.168	Global(29)	
11…	}	0	0				
11…		0	0				
11…	return {0, 0, 0, 0};	0	0				
11…	}	0	0				
11…		0	0				
11…	#if defined(KERNEL_OPT_1) || defined(KERNEL_	0	0				
11…		0	0				
11…	__device__ unsigned char GetLuminance(	0	0				
11…	uchar4* pImg,	0	0				
11…	int x,	0	0				
11…	int y,	0	0				
11…	int imgw,	0	0				
11…	int imgh,	0	0				
11…	bool applyBlur	0	0				
11…	)	0	0				
11…	{	0	0				
11…	if (!applyBlur)	14	0	621.472	1.968.128		
11…	{	0	0				
11…	return RGBtoLuminance(GetPixel(pImg	0	0				
11…	}	0	0				
11…	else	0	0				
11…	{	0	0				
11…	unsigned int value = 0;	0	0				
11…	const int radius = 3;	0	0				
11…	const auto num = (radius + 1) * (r	0	0				
11…	for (int dy = -radius; dy <= radius	1.301	408	20.158.848	587.638.800		
11…	{	0	0				
11…	for (int dx = -radius; dx <= ra	0	0				
11…	{	0	0				
11…	value += RGBtoLuminance(GetP	57.987	28.023	171.305.184	5.143.844.604		
11…	}	0	0				

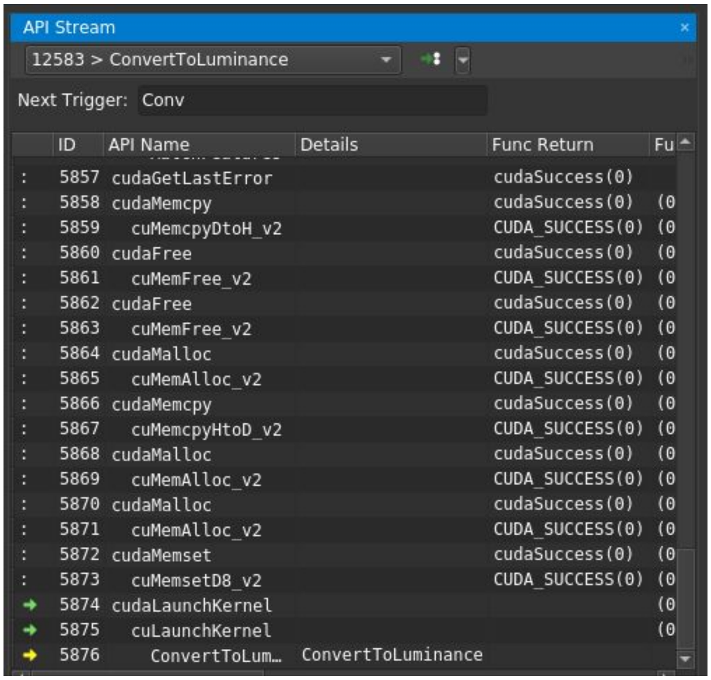
7. The current implementation of the kernel uses an RGB input image and does all luminance and blur computations “live”, i.e. whenever a pixel is read. If pixels are read multiple times, because they are part of multiple keypoint feature extractions for example, all computations will be done multiple times. This can be made more efficient by extracting the grayscale conversion and gaussian blur steps into separate kernels, run prior to the actual feature extraction kernel. Then, both grayscale conversion and blur computation are done exactly once per pixel.

Step 7: Analyze the three optimized kernels

1. Build the optimized application using three consecutive kernels per image with

```
$ make clean
$ make step=7
```

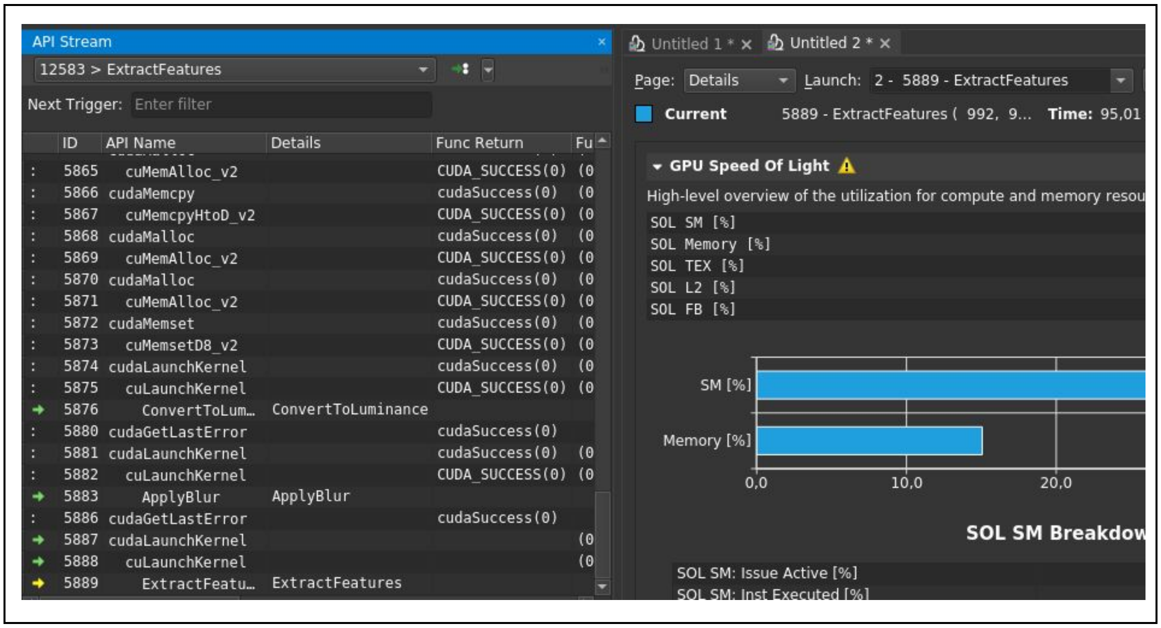
2. Connect to the application again using the Interactive Profile activity. Once more, Resume the application to a steady state, and Pause.
3. We want to analyze three matching ConvertToLuminance, ApplyBlur and ExtractFeatures kernels. If not already suspended at a ConvertToLuminance kernel, insert “Conv” into the Next Trigger edit and select “Run to Next Kernel”. Then remove “Conv” from the Next Trigger edit.



Next Trigger: Conv

	ID	API Name	Details	Func Return	Fu
:	5857	cudaGetLastError		cudaSuccess(0)	
:	5858	cudaMemcpy		cudaSuccess(0)	(0
:	5859	cuMemcpyDtoH_v2		CUDA_SUCCESS(0)	(0
:	5860	cudaFree		cudaSuccess(0)	(0
:	5861	cuMemFree_v2		CUDA_SUCCESS(0)	(0
:	5862	cudaFree		cudaSuccess(0)	(0
:	5863	cuMemFree_v2		CUDA_SUCCESS(0)	(0
:	5864	cudaMalloc		cudaSuccess(0)	(0
:	5865	cuMemAlloc_v2		CUDA_SUCCESS(0)	(0
:	5866	cudaMemcpy		cudaSuccess(0)	(0
:	5867	cuMemcpyHtoD_v2		CUDA_SUCCESS(0)	(0
:	5868	cudaMalloc		cudaSuccess(0)	(0
:	5869	cuMemAlloc_v2		CUDA_SUCCESS(0)	(0
:	5870	cudaMalloc		cudaSuccess(0)	(0
:	5871	cuMemAlloc_v2		CUDA_SUCCESS(0)	(0
:	5872	cudaMemset		cudaSuccess(0)	(0
:	5873	cuMemsetD8_v2		CUDA_SUCCESS(0)	(0
→	5874	cudaLaunchKernel			(0
→	5875	cuLaunchKernel			(0
→	5876	ConvertToLum…	ConvertToLuminance		

4. Use the “full” section set to profile this kernel, then two times advance to the next kernel using “Run to Next Kernel” and profile both the ApplyBlur and the ExtractFeatures kernels using the same set.



API Stream

12583 > ExtractFeatures

Next Trigger: Enter filter

	ID	API Name	Details	Func Return	Fu
:	5865	cuMemAlloc_v2		CUDA_SUCCESS(0)	(0
:	5866	cudaMemcpy		cudaSuccess(0)	(0
:	5867	cuMemcpyHtoD_v2		CUDA_SUCCESS(0)	(0
:	5868	cudaMalloc		cudaSuccess(0)	(0
:	5869	cuMemAlloc_v2		CUDA_SUCCESS(0)	(0
:	5870	cudaMalloc		cudaSuccess(0)	(0
:	5871	cuMemAlloc_v2		CUDA_SUCCESS(0)	(0
:	5872	cudaMemset		cudaSuccess(0)	(0
:	5873	cuMemsetD8_v2		CUDA_SUCCESS(0)	(0
:	5874	cudaLaunchKernel		cudaSuccess(0)	(0
:	5875	cuLaunchKernel		CUDA_SUCCESS(0)	(0
➜	5876	ConvertToLum…	ConvertToLuminance		
:	5880	cudaGetLastError		cudaSuccess(0)	
:	5881	cudaLaunchKernel		cudaSuccess(0)	(0
:	5882	cuLaunchKernel		CUDA_SUCCESS(0)	(0
➜	5883	ApplyBlur	ApplyBlur		
:	5886	cudaGetLastError		cudaSuccess(0)	
➜	5887	cudaLaunchKernel			(0
➜	5888	cuLaunchKernel			(0
➜	5889	ExtractFeatu…	ExtractFeatures		

Untitled 1 * × Untitled 2 * ×

Page: Details Launch: 2 - 5889 - ExtractFeatures

Current 5889 - ExtractFeatures (992, 9… Time: 95,01

GPU Speed Of Light ⚠

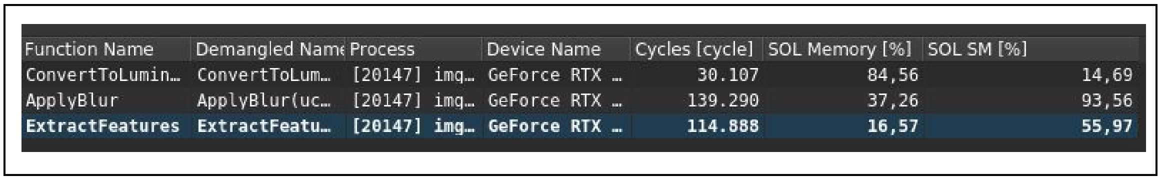
High-level overview of the utilization for compute and memory resou

SOL SM [%]
SOL Memory [%]
SOL TEX [%]
SOL L2 [%]
SOL FB [%]

SOL SM Breakdow

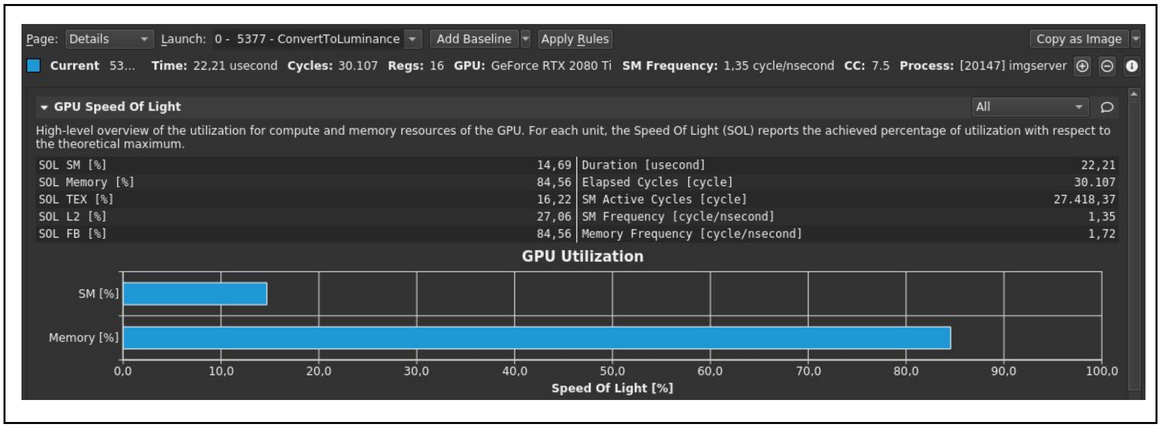
SOL SM: Issue Active [%]
SOL SM: Inst Executed [%]

5. When done, Terminate the target application and inspect the collected report.
6. Switching to the Summary page, we can quickly see that the runtime (in cycles) of the three kernels combined is much smaller than the previous runtime of ExtractFeatures (~200k cycles vs ~6000k cycles). We cannot do a useful direct kernel-to-kernel comparison anymore, since the workload from ExtractFeatures got split up across the three new kernels.



Function Name	Demangled Name	Process	Device Name	Cycles [cycle]	SOL Memory [%]	SOL SM [%]
ConvertToLumin…	ConvertToLum…	[20147] img…	GeForce RTX …	30.107	84,56	14,69
ApplyBlur	ApplyBlur(uc…	[20147] img…	GeForce RTX …	139.290	37,26	93,56
ExtractFeatures	**ExtractFeatu…**	**[20147] img…**	**GeForce RTX …**	**114.888**	**16,57**	**55,97**

7. Back on the Details page, we can use the Launch drop down to switch to the first profiled kernel, ConvertToLuminance. We see that it is memory-bound, utilizing the memory units to ~85%. The same conclusion can be drawn from the Compute Workload Analysis section, which shows that LSU (Load Store Unit) is the most-utilized pipeline, and in the Warp State Statistics, which is clearly dominated by Long Scoreboard Stalls, indicating warps waiting on global memory.



Page: Details Launch: 0 - 5377 - ConvertToLuminance

Current 53… **Time:** 22,21 usecond **Cycles:** 30.107 **Regs:** 16 **GPU:** GeForce RTX 2080 Ti **SM Frequency:** 1,35 cycle/nsecond **CC:** 7.5 **Process:** [20147] imgserver

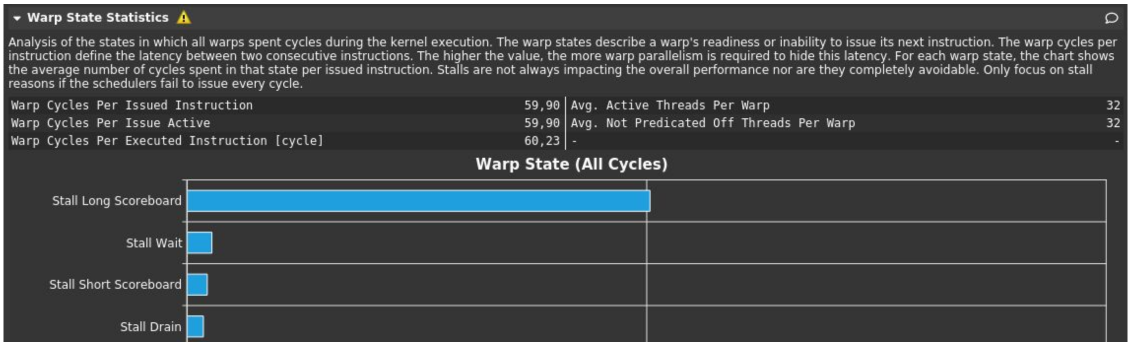
GPU Speed Of Light

All

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.

SOL SM [%]	14,69	Duration [usecond]	22,21
SOL Memory [%]	84,56	Elapsed Cycles [cycle]	30.107
SOL TEX [%]	16,22	SM Active Cycles [cycle]	27.418,37
SOL L2 [%]	27,06	SM Frequency [cycle/nsecond]	1,35
SOL FB [%]	84,56	Memory Frequency [cycle/nsecond]	1,72

GPU Utilization

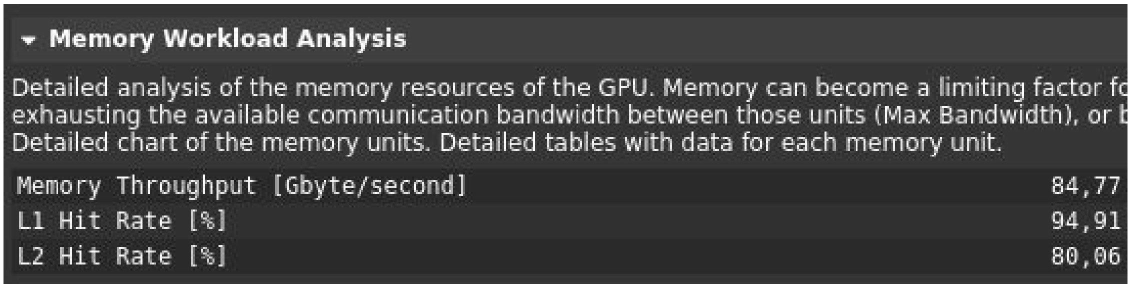


Warp State Statistics ⚠

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle.

Warp Cycles Per Issued Instruction	59,90	Avg. Active Threads Per Warp	32
Warp Cycles Per Issue Active	59,90	Avg. Not Predicated Off Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	60,23	-	-

8. The second kernel, ApplyBlur, which reads memory but also does comparatively many math operations, is again compute bound. Since the same pixels are read from global memory by multiple threads in a block, we see high cache hit rates for both L1 and L2. Warp stalls are dominated by MIO Throttle (math) and Short Scoreboard (Shared Memory/L1 reads).

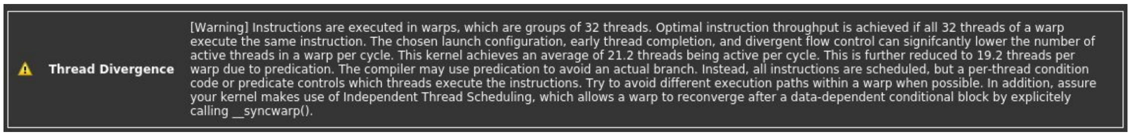


Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor f
exhausting the available communication bandwidth between those units (Max Bandwidth), or b
Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	84,77
L1 Hit Rate [%]	94,91
L2 Hit Rate [%]	80,06

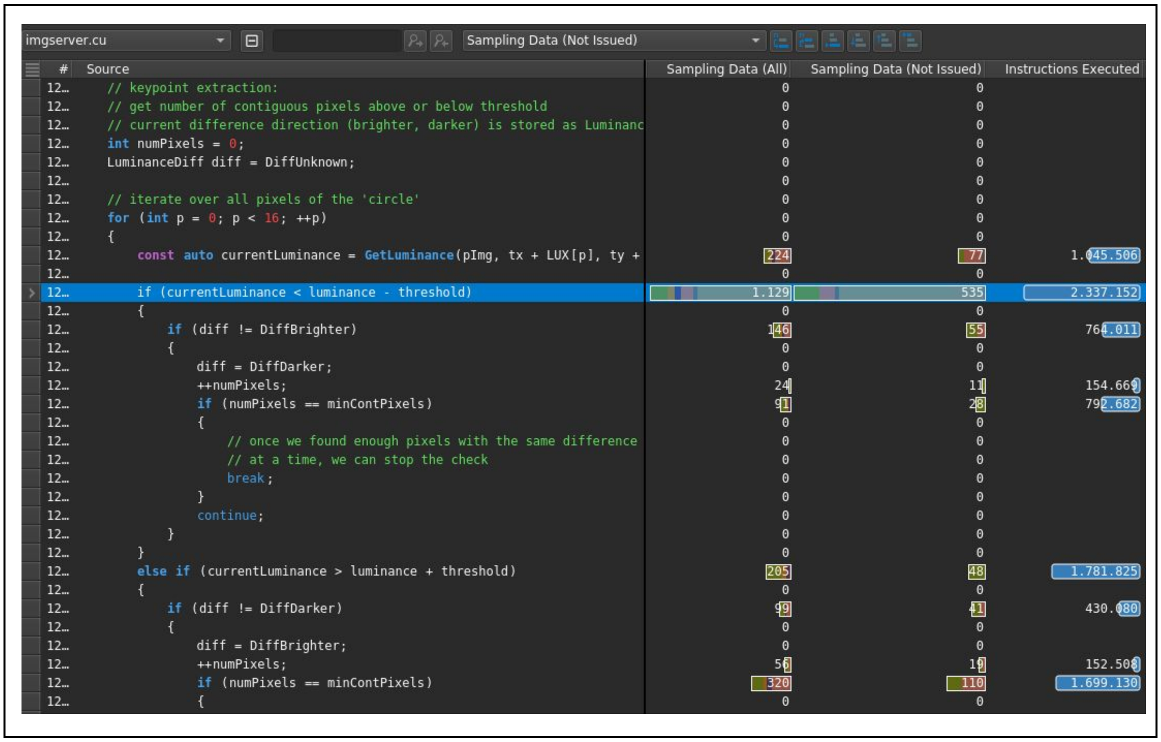
9. The third kernel, ExtractFeatures, is latency bound as indicated by the GPU Speed Of Light section warning. Inspecting the Scheduler and Warp State Statistics, we see that we issue around every 1.7 cycles (instead of every cycle). The metrics and recommendation in the Warp State Statistics section highlight the high code divergence within this kernel, which is a result of the applied algorithm.



⚠ **Thread Divergence**

[Warning] Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can signifcantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 21.2 threads being active per cycle. This is further reduced to 19.2 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, assure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitely calling __syncwarp().

10. Inspecting the kernel's primary Sampling Data (Not Issued) on the Source page, we see that warps are mostly stalled at control flow conditions during the keypoint extraction phase (in which case control flow depends on the local pixel luminance values).



imgserver.cu

Sampling Data (Not Issued)

```
Source                                                                    Sampling Data (All)  Sampling Data (Not Issued)  Instructions Executed
    // keypoint extraction:                                                                 0                           0
    // get number of contiguous pixels above or below threshold                             0                           0
    // current difference direction (brighter, darker) is stored as Luminanc                0                           0
    int numPixels = 0;                                                                      0                           0
    LuminanceDiff diff = DiffUnknown;                                                       0                           0
                                                                                            0                           0
    // iterate over all pixels of the 'circle'                                              0                           0
    for (int p = 0; p < 16; ++p)                                                            0                           0
    {                                                                                       0                           0
        const auto currentLuminance = GetLuminance(pImg, tx + LUX[p], ty +                224                          77              1.045.506
                                                                                            0                           0
        if (currentLuminance < luminance - threshold)                                   1.129                         535              2.337.152
        {                                                                                   0                           0
            if (diff != DiffBrighter)                                                     146                          55                764.011
            {                                                                               0                           0
                diff = DiffDarker;                                                          0                           0
                ++numPixels;                                                               24                          11                154.669
                if (numPixels == minContPixels)                                            91                          28                792.682
                {                                                                           0                           0
                    // once we found enough pixels with the same difference                 0                           0
                    // at a time, we can stop the check                                     0                           0
                    break;                                                                  0                           0
                }                                                                           0                           0
                continue;                                                                   0                           0
            }                                                                               0                           0
        }                                                                                   0                           0
        else if (currentLuminance > luminance + threshold)                                205                          48              1.781.825
        {                                                                                   0                           0
            if (diff != DiffDarker)                                                        99                          41                430.080
            {                                                                               0                           0
                diff = DiffBrighter;                                                        0                           0
                ++numPixels;                                                               56                          19                152.508
                if (numPixels == minContPixels)                                           320                         110              1.699.130
                {                                                                           0                           0
```

See if you can find any ways to improve the performance of these kernels. Performance analysis and optimization is an iterative process that only ends when you determine the work required is not worth the return on investment. This exercise introduced several tools to help you with that process as well tools to help debug CUDA code.