Lab 6 - CUDA debugging, profiling and optimisation

Files to be used in this lab:
- transpose.cu, run_batch.sh

Lab materials are also available on colab.

This week we'll focus on debugging, profiling and optimisation. You may often find yourselves in a situation where you either need to find a bug in your own code or debug someone else's code. These tools will come handy in both cases.

1) Debugging

The main debugging tool for CUDA is CUDA-GDB. It's built on top of the open source GNU debugger (gdb), so if you're familiar with that then you should be able to get the hang of CUDA-GDB fairly quickly. The main highlights of that tool are the ability to set breakpoints in kernel code, single-step a warp of threads, inspect GPU memory (global, shared) and inspect block/thread allocation. It also allows connecting to running applications, even if they're deadlocked.

CUDA-GDB is available on:

- students in `/tmp/nvidia/cuda-gdb`;
- entropy at `/usr/local/cuda/bin/cuda-gdb`. Remember it's visible from computation node only.

As files inside `tmp` on students may disappear, please copy it to your home dir.

To use CUDA-GDB, you need to compile your code with debugging support enabled. You can do it like so: `nvcc -g -G source.cu -o my-app`. Then you need to run the following: `cuda-gdb ./my-app`. This begins the debugging session - your terminal prompt should change to `(cuda-gdb)`. To execute the program, use the `run` command. To set a breakpoint at a kernel named ker, type `break ker`. To execute your code step by step, you can use the `next` command. In the following examples we'll debug the single.cu file from lab5.

```
(cuda-gdb) break kernel
Breakpoint 1 at 0x40344c: file single.cu, line 6.
(cuda-gdb) run
Starting program: /home/lr370887/lab5/streams/./single
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7fffefa74700 (LWP 31821)]
[New Thread 0x7fffef273700 (LWP 31822)]
[New Thread 0x7fffee9f1700 (LWP 31823)]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0, sm 0, warp 0, lane 0]

Breakpoint 1, kernel<<<(4096,1,1),(256,1,1)>>> (a=0x1050da00000, b=0x1050de00000, c=0x1050e200000) at single.cu
7           int tid = threadIdx.x + blockIdx.x * blockDim.x;
(cuda-gdb) next
8           if (tid < N) {
```

When you're within a kernel, you can use several commands to enforce more fine-grained control. To see which thread within a block is active, type `cuda thread`. You can change focus between threads within a block by typing `cuda thread tX,tY,tZ`, where `tX,tY,tZ` are thread coordinates. If you want to change the block, you need to use `cuda block bX,bY,bZ thread tX,tY,tZ`, where `bX,bY,bZ` are block coordinates and `tX,tY,tZ` are thread coordinates.

The `info cuda devices` command shows all GPUs in your system, with * denoting the one in focus.

?

```
(cuda-gdb) info cuda devices
Dev PCI Bus/Dev ID               Name Description SM Type SMs Warps/SM Lanes/Warp Max Regs/Lane Active SMs Ma
* 0        02:00.0 GeForce GTX 1080 Ti     GP102-A   sm_61  28       64         32           256 0xffffffff
  1        03:00.0 GeForce GTX 1080 Ti     GP102-A   sm_61  28       64         32           256 0x00000000
  2        83:00.0 GeForce GTX 1080 Ti     GP102-A   sm_61  28       64         32           256 0x00000000
  3        84:00.0 GeForce GTX 1080 Ti     GP102-A   sm_61  28       64         32           256 0x00000000
```

Similarly, `info cuda kernels` shows all active kernels:

```
(cuda-gdb) info cuda kernels
Kernel Parent Dev Grid Status   SMs Mask    GridDim  BlockDim Invocation
* 0      -   0    1 Active 0x0fffffff (4096,1,1) (256,1,1) kernel(a=0x1050da00000, b=0x1050de00000, c=0x1050e2
```

To see threads active within a given kernel, type `info cuda threads kernel N`, where `N` is the kernel id:

```
(cuda-gdb) info cuda threads kernel 0
 BlockIdx ThreadIdx To BlockIdx ThreadIdx Count         Virtual PC  Filename  Line
Kernel 0
* (0,0,0)   (0,0,0)     (0,0,0)  (31,0,0)    32 0x0000000000b44ac8 single.cu     8
  (0,0,0)  (32,0,0)     (0,0,0) (127,0,0)    96 0x0000000000b44a70 single.cu     7
  (0,0,0) (128,0,0)     (0,0,0) (255,0,0)   128 0x0000000000b44910 single.cu     6
  (1,0,0)   (0,0,0)     (1,0,0) (127,0,0)   128 0x0000000000b44a70 single.cu     7
  (1,0,0) (128,0,0)     (1,0,0) (159,0,0)    32 0x0000000000b44910 single.cu     6
  (1,0,0) (160,0,0)     (1,0,0) (191,0,0)    32 0x0000000000b44908 single.cu     6
  (1,0,0) (192,0,0)     (1,0,0) (223,0,0)    32 0x0000000000b44910 single.cu     6
  (1,0,0) (224,0,0)     (1,0,0) (255,0,0)    32 0x0000000000b44908 single.cu     6
  (2,0,0)   (0,0,0)     (2,0,0) (127,0,0)   128 0x0000000000b44a70 single.cu     7
  ...
```

During step-by-step execution, you can print the value of a variable by typing `print var`, where `var` is a variable name. You can also read values from device memory.

```
(cuda-gdb) print tid
$1 = 0
(cuda-gdb) print a[0]
$2 = 1804289383
(cuda-gdb) print a[1]
$3 = 1681692777
```

Along with CUDA-GDB, NVIDIA also provides the CUDA Memory Checker. It can be used either from within CUDA-GDB or as a standalone utility - `cuda-memcheck`. It greatly enhances reporting of invalid (global) memory accesses. Normally, such behaviour results in applications just stopping, without any useful messages. The `cuda-memcheck` tool actively looks for invalid accesses and thus provides useful error reporting. To use it, run the following command: `cuda-memcheck ./my-app`.

On Windows, you can use another debugging tool - NVIDIA Parallel Nsight. It's a plugin for Visual Studio - it integrates with that IDE quite well. It provides very similar functionality to that of CUDA-GDB, so we won't describe it in detail.

2) Profiling

After you find and fix all the bugs with a debugging tool, the next step in optimising your CUDA applications should be performance profiling. It's possible to write perfectly functioning code which just doesn't perform as quickly as you may have initially thought. Fortunately NVIDIA provides tools that can help with identifying any bottlenecks in your applications and suggest possible solutions.

We have already used one of the profiling tools - NVIDIA Nsight Systems. However, it provides only high level information about interaction between GPU and the host. Now we want to dig into the GPU and find out if we use hardware in optimal way.

For this purpose we introduce the next tool, `NVIDIA Nsight Compute`. Nsight Compute is available on entropy cluster the same way as nvcc. It may also be downloaded from https://developer.nvidia.com/gameworksdownload. It may require registering to the nvidia site. Files are also available on students at `/tmp/nvidia/`. The tool should be used via `ncu` and `ncu-ui` binaries.

Important note!
There are some limitations to the external resources we use. You have to remember that:

- On computers in the lab:

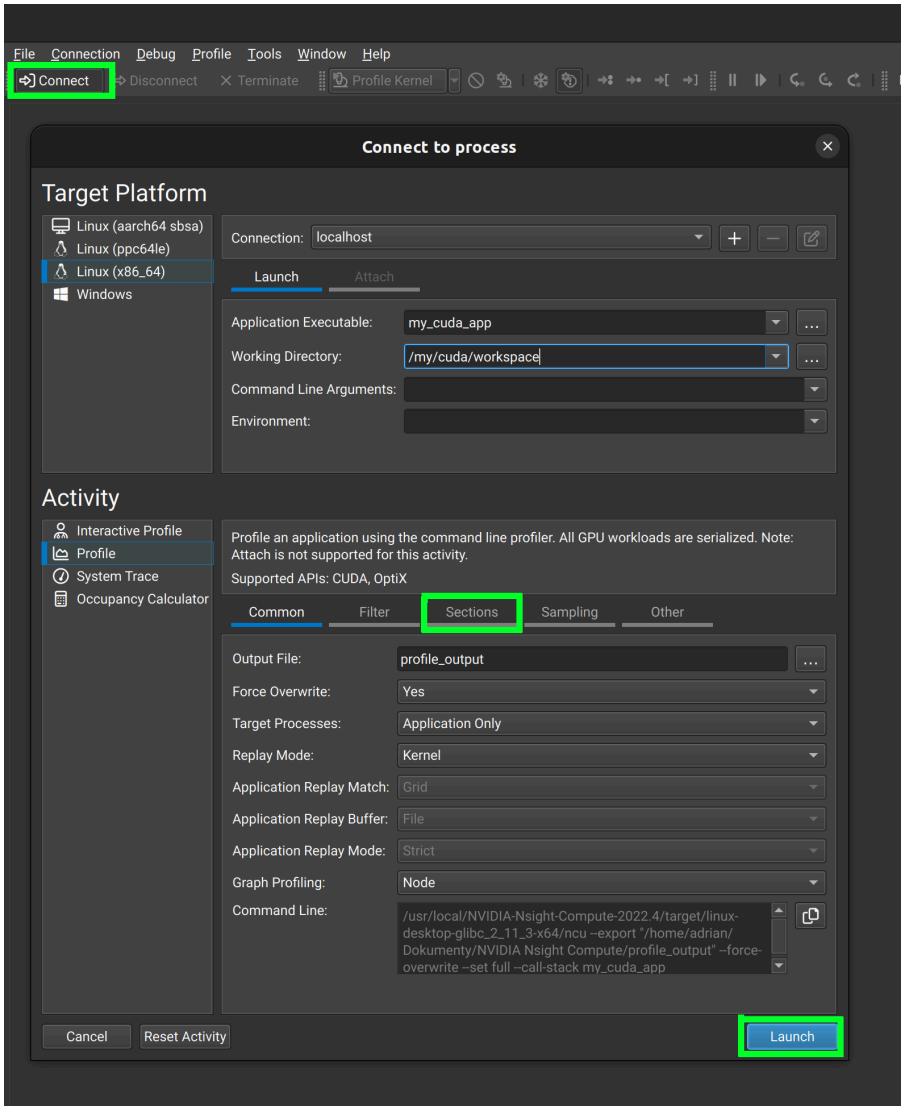
1. You cannot use `ncu` to profile GPU applications, because non-root users don't have access to GPU profiling.
2. `NVIDIA Nsight Compute UI (ncu-ui)` can be used for reading the profiler output.

- On the entropy cluster:

1. You can use `NVIDIA Nsight Compute CLI (ncu)` to profile an application and generate report file.
   It's available only on computation nodes at `/usr/local/cuda/bin/ncu`.
2. `NVIDIA Nsight Compute UI (ncu-ui)` does not work, so you cannot read profiler output there.

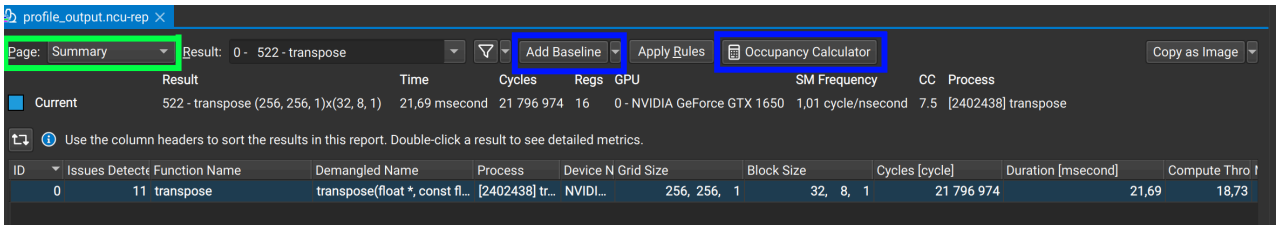
In order to profile an application, we suggest the following:

1. On the entropy cluster, use the provided "`run_batch.sh <DIR> <BIN> ncu`" script or the following command:
   `srun --partition=hpc --gres=gpu:1 bash -c "TMPDIR=/home/<LOGIN> /usr/local/cuda/bin/ncu --export profiler_output --force-overwrite --set full <BIN>"`
   or the provided "`run_batch.sh <DIR> <BIN> ncu`" script.
   Setting the TMPDIR variable is required when mutliple users use the same machine.
2. Report will be generated and stored in the `profiler_output.ncu-rep` file.
3. As `ncu-ui` does not work on entropy, you need to:
   - copy generated report to a computer with `ncu-ui` installed (in the lab, for example)
   - Open `ncu-ui` and choose File -> Open File... -> Select the report file. You will now see the Summary page.
4. If you are running on a computer with GPU profiling enabled and X available, you may run profiling directly from the `ncu-ui`:
   - Click on the `Connect` button. It should open the following dialog. In the highlighted `Sections` tab you may specify how many informations to collect. As we are running relatively small kernels, start with `full` specification. When everything is done, let's start by clicking `Launch`.

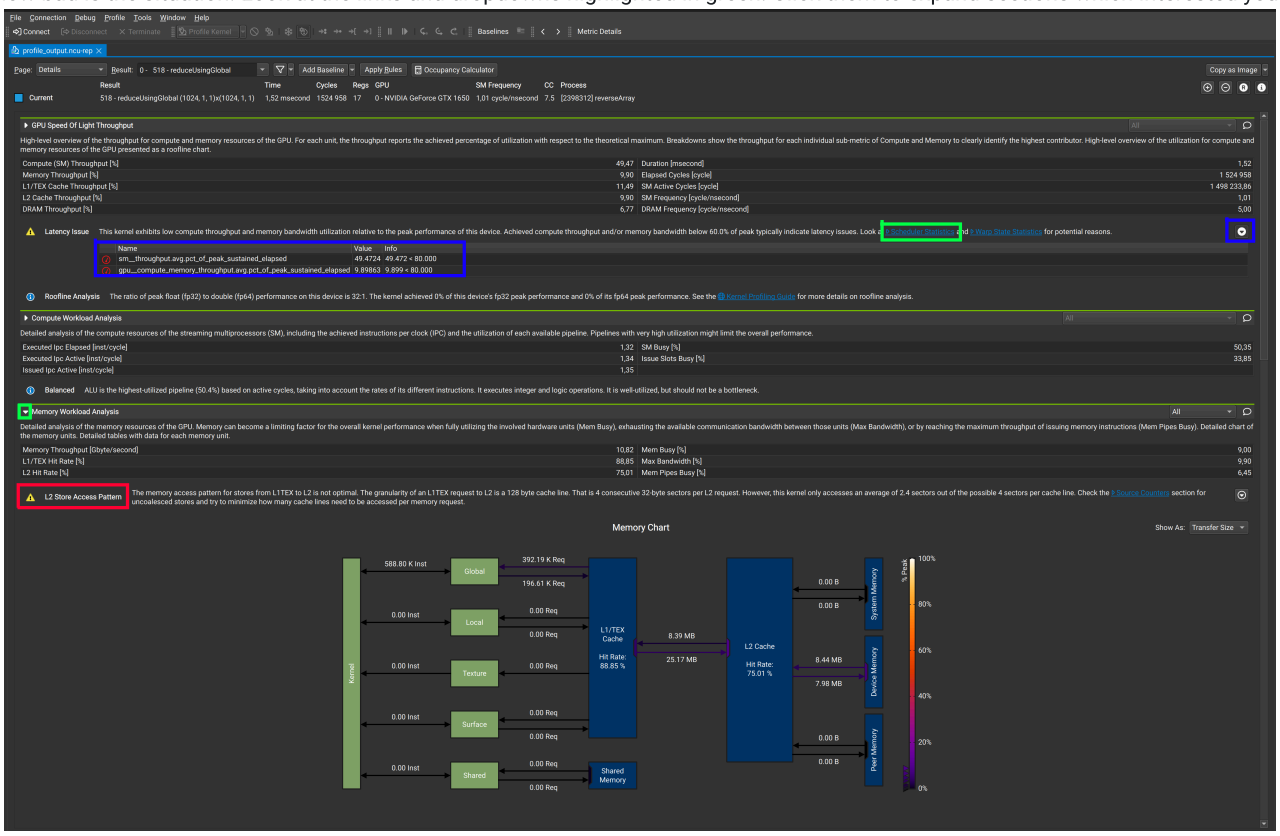


Regardless of how you open the report, you should now see the following:

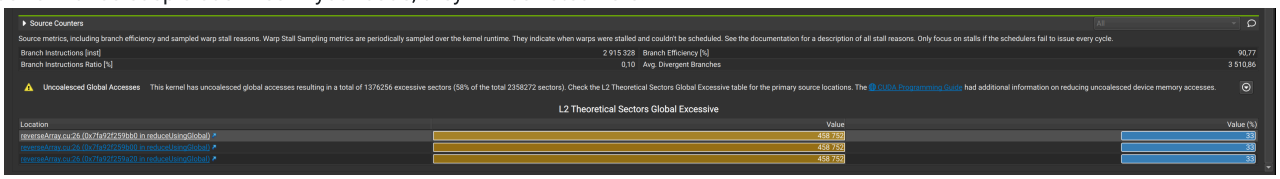
On the first page you see a brief summary of the kernel run. Note that when running multiple kernels, each of them will appear on the list. There are some features marked with blue boxes, you may find those helpful. For now, let's click on the `Page` dropdown, marked with green. Two most useful pages are `Details` and `Source` (make sure to compile with debug options to see the code).



Let's look at the `Details` page. Each section provides some useful information. The most important parts are marked with warning signs (red box). Next to the warning, on the right side (blue box) there is dropdown. Click it to expand additional warning info (another blue box). It shows why specific warning is shown using inequality with threshold values. Thanks to this you may estimate how bad is the situation. Look at the links and dropdowns highlighted in green. Click them to expand sections which interested you.

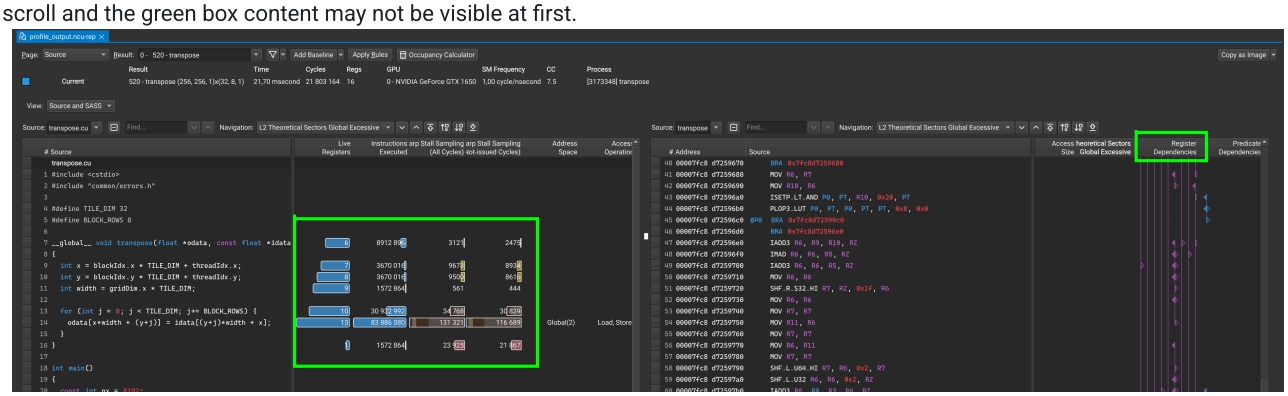


At the bottom of the `Details` page there is a `Source counters` section. If kernel was compiled with debug symbols there are performance-suspicious lines in your code, they will be listed here.



The `Source` page is also very informative. We won't go into much details here, just keep in mind that some windows have horizontal

scroll and the green box content may not be visible at first.



As you can see, NVIDIA Nsight Compute can be really helpful in identifying various performance problems in CUDA applications. It's a very powerful tool, so use it whenever possible.

3) Optimisation

As a result of profiling, you may need to optimise your code. Guided analysis in NVIDIA Nsight Compute will help you in most cases, but there may be issues that you need to inspect further.

One such problem could be that of inefficient occupancy. Occupancy is defined as the ratio of active warps on a streaming multiprocessor (SM) to the maximum number of active warps supported by the SM. Occupancy varies over time as warps begin and end, and can be different for each SM.

Low occupancy results in poor instruction issue efficiency, because there are not enough eligible warps to hide latency between dependent instructions. When occupancy is at a sufficient level to hide latency, increasing it further may degrade performance due to the reduction in resources per thread.

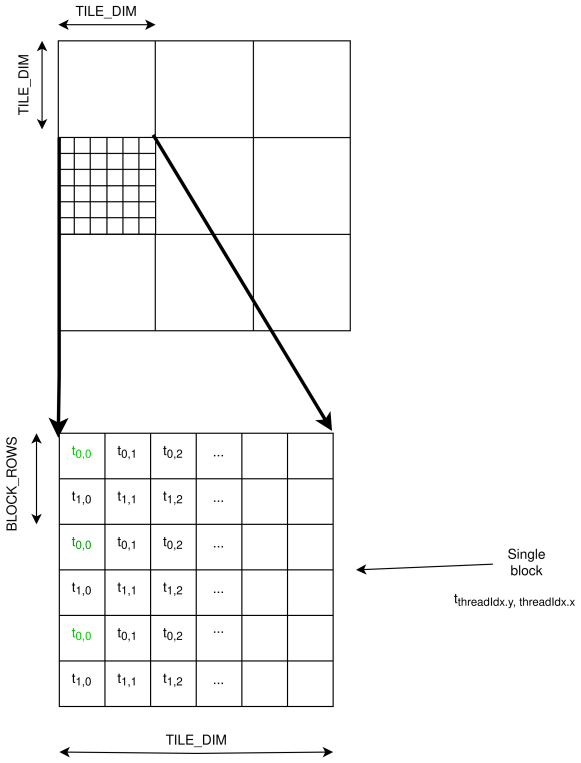
Nsight Compute provides a tool that can be used to calculate an optimal number of threads per block, shared memory size per block and registers per thread, depending on the compute capability (i.e. your GPU generation) and configured shared memory size. You can use it to maximise occupancy. You may access it in two ways:

- Click `Connect` from the top menu. Then from the `Activity` menu select `Occupancy Calculator`. Press `Launch`. This way is useful if you are prototyping the algorithm. If you don't know the `Compute Capability` you may run `nvidia-smi --query-gpu=compute_cap --format=csv` command on the target device.
- During a profiling session select `Occupancy Calculator` from the report view. Note that it will set all the parameters to the values used by the GPU and the selected kernel.

4) Exercise (1 point)

Download the transpose.cu file. It contains an incorrect and suboptimal implementation of matrix transpose on the GPU. Your task is to use CUDA-GDB to localise any bugs and fix them. Hint: look for bugs in the kernel code. Then, you have to profile the application and find performance bottlenecks. Hint - shared memory may be used as a buffer to better organize global memory accesses. Optimise the code and profile it again.

Image below shows how threads are assigned to the matrix elements:



5) Sources

- CUDA Programming Guide
- *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Sanders J., Kandrot E.
- *Programming Massively Parallel Processors: A Hands-on Approach (Third Edition)*, Kirk B., Hwu W.

Ostatnia modyfikacja: czwartek, 3 kwietnia 2025, 18:59