TDT4200: Parallel Computing

Parallel Computing - Assignment 5

November 5, 2018

Bart van Blokland
Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

- **Delivery deadline: November 16th, 2018 by 23:00.**
- **This assignment counts towards 5% of your final grade.**
- You can work on your own or in groups of two people.
- Deliver your solution on *Blackboard* before the deadline.
- Upload your report as a single PDF file.
- Upload your code and results as a single ZIP file solely containing the source files (src directory). Do not include binaries or additional resources provided with this assignment. Not following this format may result in a score deduction.
- All tasks must be completed using C++.
- Do not include any additional libraries apart from standard libraries or those provided.
- The delivered code is taken into account with the evaluation. Ensure your code is documented and as readable as possible.

Questions which should be answered in the report have been marked with a [**report**] tag.

Objective: Becoming familiar with writing code for the GPU using CUDA, and trying out GPU profiling tools.

Parallel Computing - The power of the GPU

In the previous assignment, we've looked at how to run code on the GPU. In this final assignment, we're going to continue where we left off, and (hopefully) improve its performance.

The first step in this is figuring out exactly what parts of our program we can improve. We'll therefore start by measuring the performance of the program we wrote last time using NVidia's profiling tools.

We afterwards have the option of tapping into two main hardware features which can help us squeeze faster execution times out of the graphics card.

First, shared memory provides a kind of "scratchpad", in which threads can cooperatively work on a small chunk of data. If your algorithm requires frequent reads and write to and from memory, doing parts of it in shared memory instead can yield a significant speedup.

For instance, you can copy a part of an array into a buffer in shared memory, perform computations on it with a number of threads, and write it back when done. As memory shared memory transactions are much faster, this can yield significant speedups.

Second, we can abuse warps to assist each other when the need arises. Special warp-level cooperation instructions allow fast and synchronisation-free communication between threads. Since these instructions are running inside the core, they are extremely fast.

In this assignment, we will only look at the latter of these two.

Keep in mind that you are not allowed to use any additional libraries apart from the C++ standard library or those provided. You are allowed to create additional source files, but your main focus should be working with the existing ones. Please leave the original command line parameters intact.

This assignment will contribute with 5% to your final grade.

Remember that at the very basic level, we're looking for whether you've understood the concepts and methods this assignment touches upon. Make sure you show this when answering a question.

If you have any further questions please ask them first on the blackboard discussion board, as it is likely others have them too. However, make sure not to post large quantities of code there. You can send them by mail to Björn and/or Bart if necessary.

Lab machines

Just like last assignment, if you do not have access to an Nvidia GPU yourself, we have a number of lab machines available to you. There are two ways of using them. First, you can access them remotely through SSH (clab01.idi.ntnu.no to clab26.idi.ntnu.no. Log in through login.stud.ntnu.no first to reach them). Alternatively, you can of course access the lab any time by using your access card to get in.

The caveats are the same as last time.

First, we ask you not to access the lab machines remotely during the daytime (approximately 8:00 to 18:00).

Anyone who shows up physically to the lab has priority to use the machines, and is allowed to forcefully log out people who are logged in remotely. This counts for any time (day or night alike, but I sincerely hope you don't end up needing the latter).

Second, please note that TDT4195 is also making use of the lab, and has a weekly help session on Thursdays from 12 to 14. Please do not use the lab in this time period.

Finally, lab machines are first come first serve. Typically people start working on the assignments quite close to the deadline, which means there are likely going to be shortages of machines towards the end. Just as last time, I can only give you a single advice here:

Start Early.

We do will not accept late submissions as a result of not having access to a GPU.

Best of luck!

Task 0: Preparation [0 points]

a) *Optional*: You will *NOT* be able to complete this assignment using a virtual machine. If you were using one, you will need to migrate your build environment to a native operating system (either install one, or use the one already on your machine). This may require you to install the tools listed under the point below.

b) Ensure the following tools are installed:

- CMake and/or make
- Git
- A C++ compiler, such as G++ or MSVC++. [1]
- A CUDA installation [2]

These have already been installed for you on the lab machines.

c) Clone the assignment repository using the following command:

```
git clone https://github.com/bartvbl/TDT4200-Assignment-5.git
```

d) Compile the project. For this assignment you'll need to use Cmake to generate the build files. If some of the paths don't match (even though they should by default), you may need to edit the CMakelists.txt file. I've installed cuda on a fresh ubuntu installation and everything worked fine, so hopefully it is more or less plug and play for all of you too.

- CMake

```
cd build
cmake ..
make
```

e) Give it a test run either through the shipped Makefile or manually:

```
gpurender/gpurender
```

Unlike last assignment, there is no need to use the -g parameter to run the program on the GPU; the CPU part of the project has been removed.

[1] Note that your version of CUDA explicitly needs to support your compiler. Hopefully you got that working during the first assignment.

[2] If you're on Linux, my experience with the .run installation package you download off of Nvidia's site has been universally terrible. For ubuntu, my recommendation is to use "sudo apt install nvidia-cuda-dev nvidia-cuda-toolkit nvidia-nsight" instead.

Task 1: Getting Started [0.8 points]

The first objective is to measure the performance of our existing program. For this purpose, we'll use Nvidia's own performance visualisation tools called "Nsight". On both Windows and Linux, this program should already be on your system if you have installed the CUDA development tools.

One small thing to note here is that there are some slight differences between Linux and Windows. On Linux, Nsight has been developed as an Eclipse plugin. On Windows, it's an extension to Visual Studio.

While they effectively do the same, and have a similar interface, the Linux version does in my experience tend to be easier to use. This is primarily because the Eclipse version puts some effort into explaining you in words what is going on. While Eclipse is a full blown development environment, I recommend using Nsight for profiling your application only.

Getting Started: Nsight Eclipse Edition

When you start the application for the first time, just select some random workspace.

To start profiling a program, we need to create a profile configuration. In the top menu bar, click on Run > Profile Configurations. Click on C/C++ Application and then on the "New" button on the left hand side, and give it a name.

Under the "Main" tab, click the Browse button by the C/C++ Application text input. Navigate to the build folder of the project, and select your executable. Next, click "Disable Auto Build" near the bottom.

Now click on the "Arguments" tab. You can specify any program argument you'd like to pass in here (such as a desired depth and resolution). Under "working directory", untick the "Use default" box, and select the build directory of your project by clicking on "File System".

Finally, click on "Apply" near the bottom of the window to save your configuration.

You can now start a profiling session by starting your configuration from the dropdown list by the "Profile application" button. For me it's right below the "Search" menu item in the top menu bar.

When you start profiling for the first time, Eclipse asks whether you'd like to switch perspective. Click yes.

When you start profiling, a timeline is created automatically. Click on the kernel you'd like

to profile. Under the Analysis tab, click on the "unguided analysis" button (right below the tab title by default). You can inspect your application here using a series of tests.

Basic Profiling

We're now ready to run some analysis on our program, and measure some basic characteristics from where we can improve.

If the kernel does not take long to execute (less than a second) when generating the base timeline, I recommend increasing the depth limit (–depth parameter) of your program by editing the profiling configuration. If your kernel does not have sufficient data to process, it will not saturate the GPU, which can significantly skew your measurements. On the other hand, some analysis take quite a while to complete. So don't choose a depth that's too high.

For each of the tasks below, use the profiler to determine the requested value, and indicate where you found it (either a screenshot or a single sentence). Each of these refers to the main renderMeshes kernel.

a) **[0.1 points]** **[report]** What is on average the percentage of the time in which warps are executing instructions (not being blocked)?

b) **[0.1 points]** **[report]** What percentage of cycles where threads are blocked is due to a memory request being processed?

c) **[0.1 points]** **[report]** Which section of code is generally executed by the fewest number of threads in a warp?

d) **[0.1 points]** **[report]** Which line(s) of code have the worst memory access pattern?

e) **[0.1 points]** **[report]** What is currently the main factor limiting how many warps can be active on each SM simultaneously?

f) **[0.1 points]** **[report]** What is the achieved occupancy of the kernel?

In order to measure the improvement of our code over time, we'll also take a baseline measure for its runtime. Choose some settings here that cause your program to take at least a few seconds to execute. If you work on Windows and are running into issues where the WDDM driver kills your kernel due to a timeout, pick settings which allow your kernel to run for as long as possible.

g) **[0.2 points]** **[report]** What is the average total time needed to execute the renderMeshes kernel, including the time spent copying data to and from the GPU, and

running the buffer initialisation kernels? Also note down which settings you used (depth limit, as well as the width and height of the image).

Task 2: Low-hanging Fruit [1.5 points]

We can start by applying some easy optimisations first. For instance, there's potential left in the handout code to improve the occupancy.

a) **[0.2 points]** **[report]** Define the term "Occupancy". How does better occupancy generally lead to better performance?

b) **[0.2 points]** **[report]** What are the main ways in which the number of warps which can be active simultaneously on an SM can be limited?

c) **[0.2 points]** **[report]** What does "Latency Hiding" mean, and how does it relate to occupancy?

d) **[0.4 points]** **[report]** Improve the occupancy of the handout code by improving the memory access pattern of the line(s) you found in task 1d). What is the occupancy of the kernel now?

e) **[0.4 points]** **[report]** Improve the occupancy of the handout code by modifying the kernel launch parameters. Try at least 5 different configurations, listing the used parameters and achieved occupancy in your report.

f) **[0.1 points]** **[report]** Measure the execution time of the renderMeshes kernel again. What is the speedup of the changes you've made thus far, relative to your measurement in task 1g)?

Task 3: Engaging the warp drive [2.7 points]

If you look inside rasteriseTriangles(), there's a double for loop. These two nested for loops iterate over the pixels which form the bounding box of the triangle to be rendered. For each pixel inside of this rectangle, they basically do the same operation over and over again. This smells like something we could potentially run in parallel instead!

However, in the vast majority of cases, triangles only cover a few pixels only. The overhead involved in allowing the loop to be run in parallel would likely not make a lot of sense there, and probably only slow things down.

On the other hand, there are sometimes triangles which are quite large, and cover hundreds of pixels. This for example happens when an object is very close to the camera. You may

notice looking at the images produced by the handout code that some scaling has been applied by default to make this occur a little more often ;)

Now this is probably not the bottleneck of this program, and there are dozens of research papers out there with better ideas on how to *properly* manually implement a rendering pipeline on the GPU. However, there's a chance this change leads to a tiny speedup, so let's roll with it anyway.

a) **[0.1 points]** **[report]** Why should we care about whether individual threads take significantly longer than average to render a triangle?

The idea here is that for large triangles, rather than the thread rendering single pixels on its own, the work is divided between all threads in the warp. Here each thread will render one pixel at a time. This allows us to compute 32 pixels simultaneously, which should yield a speedup for larger triangles. However, we only go through the trouble of doing so when the triangle is large enough. Otherwise the overhead involved will cause this process not to be worthwhile.

b) **[0.1 points]** Each time a thread is going to draw a triangle, compute the number of pixels that need to be tested in the bounding box.

c) **[0.2 points]** We only want to parallelise drawing of triangles if they are above a certain size. Otherwise the overhead is not worth it. Find a way that does not use global or shared memory to communicate with other threads in the warp whether the number of pixels in the bounding box you computed exceeds a set threshold. You can hardcode said threshold. I recommend choosing a value over 32.

d) **[0.7 points]** Each thread which has indicated it is processing a large triangle should now be rendered separately. Create a loop which iterates over each such thread. You may find the ___ffs() function useful here.

e) **[0.7 points]** For each triangle the aforementioned loop iterates over, we'll divide every pixel in the bounding box between the threads in the warp. Before we can iterate over each pixel in the bounding box in parallel however, we need to exchange a copy of the triangle which is going to be rendered between the threads in the warp.

Use shuffle instructions to broadcast a copy of the thread-dependent parameters of the rasteriseSinglePixel() function to all threads in the warp. Note that this needs to be done for each thread which previously indicated they had a large triangle to render (no need to create an array of parameters though).

f) **[0.3 points]** Each thread in the warp should now be capable of rendering pixels from the large triangle. Use them to cooperatively render all pixels within the bounding box.

g) **[0.3 points]** There are two edge cases we need to solve. First, any threads that were out of range in renderMeshes() exited early, and will not be taking on any jobs in our new parallelised rasterisation section. Resolve this. Note that these "out of bounds" threads only need to execute this particular section of code. Just pass in uninitialised dummy variables into any functions they otherwise would call. Just make sure that memory reads and writes are protected.

h) **[0.1 points]** The second edge case is that when a large triangle is rasterised in a parallelised fashion, the thread to which that triangle originally belonged should not try to render the original triangle again.

i) **[0.1 points]** **[report]** Measure the occupancy of the updated kernel. Has it improved relative to your previous best?

j) **[0.1 points]** **[report]** Measure the execution time of the renderMeshes kernel, the same way you have before. Did making this extensive change yield a speedup?