TDT4200: Parallel Computing

Parallel Computing - Assignment 4

October 26, 2018

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

- **Delivery deadline: November 3rd, 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++.
- Use only functions present up to and including C++11.
- 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 - Crash Course CUDA

We have previously looked at how to fully utilise the compute power of the CPU by using threads, and how to execute a program on a potentially large number of machines simultaneously using MPI.

However, we have not yet touched upon the device which is responsible for lots of pretty pictures in games, used extensively in data centres, and which is single handedly responsible for allowing neural networks to be simulated and trained within a reasonable amount of time.

Working with a graphics card can be quite different compared to writing code for the CPU. This assignment therefore aims to be an introduction to get you started with writing code for the GPU.

Although all operating systems can be used for solving this assignment [1], Linux is highly recommended and best supported. The main reason for this is that on Windows, whenever a display is connected to a graphics card, the card is required to use the so-called "Windows Display Driver Model" variant of the graphics driver. This driver will cut off any GPU program executing more than a few seconds. So if your program crashes consistently after a certain amount of time, it may be the driver doing that.

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.

[1] The previous ones have used a linux-specific method for parsing command line options. I've now migrated that to a library that's cross-platform :) It should therefore now be possible to use Windows too.

Lab machines

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.

There are, however, some caveats.

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 weekly help session just like TDT4200. Please do not use the lab machines during these periods. I'm not sure whether it's the case right now, but there should be a schedule on the door of the Tulipan lab showing when the room is reserved exclusively for a particular course.

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. 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. There is more than enough time and machines for everyone to do their assignments. [2].

Best of luck!

[2] If I count 12 hours of lab use per day, 13 days until the deadline, and 300 students by combining the number of people taking TDT4195 and TDT4200 (not excluding people not using the lab and people taking both courses), I get about 12.5 hours per student.

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++. [3]
- A CUDA installation [4]

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

c) Clone the assignment repository using the command:

```
git clone https://github.com/bartvbl/TDT4200-Assignment-4.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:

```
cpurender/cpurender
```

[3] Note that on Linux, CUDA does not support the latest version of g++. On ubuntu, you can install the latest version of g++ 6 using "sudo apt install g++-6". The project's included CMakelists makes sure g++ 6.x is used.

[4] 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" instead.

Overview

In this assignment, you will witness the ~~fire~~power of a fully armed and operational ~~battle station~~ GPU.

For this assignment, we'll be returning to the software rasteriser we've used in assignment 2. The objective is to port it so that it runs efficiently on the GPU.

Task 1: More OpenMP [1.5 points]

We'll be comparing the runtime of the GPU implementation with the one on the CPU. However, the baseline implementation from assignment 2 is only single threaded. Measuring the algorithm's speedup on the GPU compared to the CPU is thus not representative of what the CPU is capable of. Therefore, before we start porting the rasterisation algorithm over to the GPU, we'll make the CPU version run faster by ensuring it is using all available cores on the CPU.

In order to get most out of the CPU, you usually get most of the way there by spreading your program between all cores, and using SIMD instructions (SSE and AVX) as much as possible. However, SSE takes quite a bit of time to implement here, so we'll only be parallelising the main rendering loop. This also yields us the greatest gains.

If you look through the project, you may notice things have changed a bit too. First off, I have created a separate section for the CPU and GPU parts of the program. This allows us to start with a "fresh copy" of the source code once we move over the the GPU. If you're wondering why a number of functions have been inlined, it was a quick and dirty way of making sure C++ could live with multiple copies of the same function in different parts of the program.

Second, CUDA defines its own float2, float3, and float4 types. The CUDA variants unfortunately do not define as many operators as the ones we have used before from *floats.hpp*. The starting project for the GPU has therefore become a bit more verbose, although it is still functionally equivalent to the original. I have also restructured the GPU starting version so that it would be much easier for you to port it to the GPU.

Third, because it significantly simplifies the implementation on the CPU and GPU alike, I've changed the recursive rendering function such that it only creates a list of work to be done. Once this list of work items is constructed, the program loops over each item to render it.

a) **[0.2 points]** [**report**] Choose some settings (image width, image height, and recursion depth) that take a significant amount of time to run on your machine (in the order of 10-30 seconds). When modifying the image resolution, I recommend scaling each side uniformly as the camera "lens" is configured for the aspect ratio of 1920x1080 (1.7778).

Measure the execution time for the loop which iterates over workQueue items in rasteriseCPU() to render them. Show the runtimes of this loop for a few runs of the program in your report, as well as the settings you chose..

b) **[0.2 points]** Use OpenMP to evaluate the loop responsible for rendering the workQueue in rasteriseCPU() in parallel.

c) **[0.3 points]** [**report**] Run the program a few times and look at the images it produces. Are they correct? Are they always the same? Fix any race conditions that were caused by the loop parallelisation in the previous subtask.

Document the cause of any race conditions you found in your report. Also include one of the incorrect images.

If you parallelised the loop as I did, you should get exactly one race condition here.

d) **[0.3 points]** [**report**] Try using the three different OpenMP scheduling strategies (static, dynamic, and guided) on the parallelised for loop. Use the same program settings as the ones you chose in the first subtask.

e) **[0.3 points]** [**report**] What are the speedups of the different scheduling strategies relative to the single threaded version?

Document the runtime of each scheduling strategy and speedups relative to the single threaded variant in your report.

Make sure to only measure the execution time of the loop you parallelised itself.

f) **[0.2 points]** [**report**] Are the runtimes you measured for the different scheduling strategies what you would expect them to be? Can you expect one specific scheduling strategy to be faster than the others?

Support your answer.

Task 2: Getting Started [0.4 points]

Time to move over to the GPU! You'll need to work in the source files within the GPU folder for this task.

a) **[0.1 point]** We'll start with some easy steps. We'll enumerate the available compute devices, and create a context on one of our choice. Use the following function near the start of rasteriseGPU():

```
cudaError_t cudaGetDeviceCount(int* count);
```

Note that you need to allocate an integer on the stack to hold the device count, and pass a reference to it into the getDeviceCount() function. Also note that the function returns a special struct that indicates whether an error occurred. Most CUDA functions have this behaviour. You can wrap the function call in a checkCudaErrors() macro, which will check the contents of the struct for whether any errors occurred, and print them out if they did. I've taken the liberty to include it into the gpurasteriser.cu file by default :)

You should wrap all cuda..() function calls for this assignment in such a checkCudaErrors() macro, wherever applicable.

Make sure at least 1 GPU is detected. If an error occurs, or no devices have been detected, your driver may not be installed correctly.

b) **[0.2 points]** **[report]** GPU devices available on your computer are given IDs starting at 0. We can request their properties using the following function:

```
cudaError_t cudaGetDeviceProperties (cudaDeviceProp* prop, int device);
```

cudaDeviceProp is a struct which contains a slew of information about your graphics card. Allocate one on the stack, and pass a reference into the function. The device parameter is the device ID. If you have a single GPU this ID is always 0.

Print the contents of the "name" field to the command line, and show a screenshot of it in your report.

You can find the complete list of fields in the struct here:

`https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html`

c) **[0.1 points]** Now that we know the GPU is detected and available, let's create a CUDA context on it. All subsequent CUDA function calls that have some kind of interaction with the GPU will be directed towards that GPU.

Use the following function to create a CUDA context on your GPU of choice:

```
cudaError_t cudaSetDevice(int deviceID);
```

Task 3: Some Planning [0.5 points]

Before we move on, we should think a bit about how we're going to divide the work between threads.

In the subsequent tasks, we'll be parallelising the "renderMeshes()" function.

a) **[0.3 points]** **[report]** The best means by which we can find code we can run in parallel is to look for repeating sections of identical code: loops.

Starting from the renderMeshes() function, draw a tree structure in your report (as in: something like a folder tree in a file manager), showing all loops encountered by the program. The loops should be shown in the order the program encounters them, as well as which loops are nested inside which other loops.

Mark each loop with roughly how expensive it is (for example: short, medium, or long). If you're not entirely sure about the execution time, use std::chrono to measure it, just like we did in assignment 1. Just make sure you measure the time of a *single* iteration of the loop, not the execution time of the loop overall.

b) **[0.2 points]** **[report]** Develop a strategy for how you will divide the program between threads. Which loops will you "break apart" in order to divide them between more threads, and which will you leave intact?

Briefly discuss your strategy, and support why you chose it.

Task 4: Setting things up [1.4 points]

We need to do a few more things before we can move the algorithm itself over to the GPU, primarily copy our input data over to the GPU's memory banks.

a) **[0.1 points]** Use cudaMalloc() to allocate a frame buffer and depth buffer on the GPU. Here's its function signature:

```
cudaError_t cudaMalloc(void** devicePointer, size_t sizeInBytes);
```

Note that the devicePointer variable should be a reference to a pointer variable you allocate on the stack. SizeInBytes is the number of bytes you would like to allocate. You can use C++'s sizeof() operator to determine the number of bytes a specfic data type occupies in memory.

b) **[0.5 points]** Since cudaMalloc() only returns a pointer to an allocated area of memory, we'll either need to initialise it with some kind of default value, or copy data into it from main memory (RAM) before we can use it. If you look inside the rasteriseGPU() function, you can see what the initial values of the depthbuffer and framebuffer should be.

Write two small kernels and properly initialise the frame and depth buffers, just as is done in the rasteriseGPU() function.

Do not use a for loop inside the kernel.

Launch the kernel with a block size that is greater than 1x1x1 threads.

Make sure to call cudaDeviceSynchronize() to make sure your kernel finishes execution:

```
cudaError_t cudaDeviceSynchronize();
```

This will cause your program to wait for the GPU to finish its tasks before continuing.

c) **[0.2 points]** Use cudaMalloc to allocate the workQueue in GPU memory. Transfer the contents of the buffer using cudaMemCpy():

```
cudaError_t cudaMemcpy(void* destination, void* source,
                       size_t count, cudaMemcpyKind kind);
```

Notice that destination comes before source. For some reason. Count is the number of *bytes* to transfer.

cudaMemcpyKind is essentially which direction you want to transfer your memory to. Its value is usually either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost.

d) **[0.6 points]** Of the variables that need to be allocated and copied to the GPU, the array of meshes is the most complicated, but at this point that should not be all that difficult to do. If you look at the definition of GPUMesh, you can see I simplified its definition a lot. So hopefully that saves you a fair amount of work :)

First, allocate an array of meshes *both* in CPU *and* GPU memory of the same size as the one already in CPU memory.

Next, for each mesh in the mesh array, do the following:

- Allocate a vertex and normal array on the GPU (of the correct size). Note there are just as many normals as there are vertices, and the GPUMesh structure has a vertexCount field.
- Use cudaMemCpy() to copy the contents of those buffers over to the GPU.
- Store the device pointers in the allocated array on the CPU side which you allocated previously.
- Also copy over the Mesh's fields into the one in the CPU array. No need for using cudaMalloc and cudaMemcpy here, just simple assignments.

Finally, use cudaMemcpy() to copy over the CPU allocated array to the GPU. The structures in this array now contain pointers to data in GPU memory, so the GPU is able to process it.

Task 5: You may fire when ready! [0.6 points]

a) **[0.5 points]** Convert the renderMeshes() function into a kernel using the strategy you came up with previously.

Again, make sure to call cudaDeviceSynchronize() after launching the kernel. Also remember to wrap it in a checkCudaErrors() macro to make sure your program reports any errors that occurred while executing the kernel.

b) **[0.1 points]** Copy the completed framebuffer back to main memory using the cudaMemcpy() function. If you copy it into the frameBuffer variable on the CPU, the final few lines of rasteriseGPU() should take care of handing it off to the image writing code properly.

Task 6: Solving Issues and Evaluating the Result [0.6 points]

You should now be able to produce images rendered on the GPU.

Now we just need to make sure the output produced by your program is equivalent to that of the CPU version.

a) **[0.4 points]** Closely observe a generated output image. You may see that it doesn't quite match the output of the CPU version of the program. This is unfortunately caused by multiple race conditions.

Solving all of these can be quite complicated, so to for this task you only need to resolve *one* of them. There will be a few pixels that have weird colours in the image, but it should mostly be fine.

Remember that race conditions only occur in places where memory is both read and written to by different threads at different times. Looking for places where this occurs should point you directly at the problem. Also, since the CPU and GPU variants are essentially the same, looking at the race condition you found while working with OpenMP can help here.

Hint: you may want to look at the available atomic functions in CUDA: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomic-functions`

b) **[0.2 points]** **[report]** Measure the runtime of your kernel. When rendering an image using the same settings as the ones you used for the OpenMP measurements, what is the speedup you achieved over the single-threaded CPU and multi-threaded CPU variants?

What is the achieved speedup when you also include the time spent allocating and copying buffers to and from the GPU?

OPTIONAL Task: Cleaning up [0 points]

a) **[0.0 points]** It's good practice to free any heap-allocated resources once we're done. While the operating system usually manages to free all resources, having code that does not produce memory leaks makes can save you hunting for memory leaks down the line.

You can use the cudaFree() function to delete any memory allocated using cudaMalloc():

```
cudaError_t cudaFree(void* devicePointer);
```