TDT4200: Parallel Computing

Parallel Computing - Assignment 1

September 7, 2018

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

- **Delivery deadline: September 14th, 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 and, if necessary, profiling instrumentation. Do not include binaries or additional resources provided with this assignment, such as mesh object files. 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.
- You are not allowed to change the calling behaviour of the application.
- 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: Familiarise yourself with C++, internal and external application profiling and basic code and algorithmic optimization.

Parallel Computing - Profiling and Optimization

Software optimization is an essential part of a developer's work to accelerate applications, reduce resource usage or minimize energy consumption. This must not only be taken into account in new developments, but also in existing software to adapt to new requirements. Therefore, profiling is a key technique to retrieve meaningful data from applications to analyze performance, behaviour and unforeseen problems, required by all optimization tasks.

In this assignment you are confronted with a rasterisation application which renders 3D mesh objects into Portable Network Graphic (PNG) images. Unfortunately the rendering is much slower than it can be expected. It turns out that even on a recent x86 processor, nearly a minute is spent on rendering a sphere with very few triangles. There are three more objects provided which are hard or nearly impossible to render within a reasonable amount of time.

Although all operating systems can be used for solving this assignment, Linux is highly recommended and best supported. Keep in mind that all tasks are not about parallel execution and you are not allowed to use any additional libraries apart from standard libraries or those provided. You are allowed of creating additional code files, but your main focus should stay on optimizing the existing ones. Do not change or add parameters, without ensuring that the application stays functional with - and only with - the original parameters (-i,-o,-w,-h, –sse).

This assignment will contribute with 5% to your final grade for TDT4200 Parallel Computing.

Always justify your results! Results without explanations are not counted, while correct reasoning may earn points even without satisfactory outcomes. Please keep it always short and concise. If you have any further questions please ask them first in the blackboard forum, as it is most likely others have them too.

Task 0: Preparation [0 points]

a) Ensure the following tools are installed:

- CMake or make
- Git
- A C++ compiler, such as G++ or MSVC++.

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

On Microsoft Windows, it may be easiest to install Microsoft Visual Studio Express, which can be downloaded for free and comes with the MSVC++ compiler. It is the easiest to work with in combination with CMake. Alternatively, the full version of Microsoft Visual Studio can be downloaded for free through MSDNAA (Gurutjenesten has a page with a link).

b) Clone the assignment repository using the command:

```
git clone https://github.com/bgottschall/TDT4200-Assignment-1.git
```

c) Compile the project. You are provided with three ways for getting started. The included Makefile, which takes care of the compilation, run arguments and verification, CMake for generating automatically a *new* Makefile solely compiling the project or the manual way doing everything on your own. You are free to choose, adapt and extend any of these.

- Make

  ```
  make all
  ```
- CMake

  Choosing this method will override the shipped Makefile. You will not be able to use make features mentioned in the following parts.

  ```
  cmake .
  make
  ```
- Manually

  ```
  g++ -std=c++11 -O0 -c src/utilities/lodepng.cpp
      -o src/utilities/lodepng.o
  g++ -std=c++11 -O0 -c src/utilities/OBJLoader.cpp \
      -o src/utilities/OBJLoader.o
  g++ -std=c++11 -O0 -c src/utilities/geom.cpp -o src/utilities/geom.o
  g++ -std=c++11 -O0 -c src/main.cpp -o src/main.o
  g++ -std=c++11 -O0 -c src/sse_test.cpp -o src/sse_test.o
  g++ -std=c++11 -O0  -c src/rasteriser.cpp -o src/rasteriser.o
  g++ src/utilities/lodepng.o src/utilities/OBJLoader.o \
      src/utilities/geom.o src/main.o src/sse_test.o \
      src/rasteriser.o  -o cpurender/cpurender
  ```

d) Run the project. The provided Makefile should give you a comfortable way to execute the compiled application. However, you should also be able to execute it manually for some tasks or together with external instrumentation.

- Make

  The following lines should give you enough ideas on how to use the provided Makefile. For all available options have a look in the make help page (type in 'make help') or review the Makefile itself.

```
make call
(example) make call OUTPUT=output/prop.png
(example) make call OUTPUT=output/head.png WIDTH=480 HEIGHT=270
(example) make verify OUTPUT=output/sphere.png
(example) make sse INPUT=input/scene.obj
```

- Manually

  The generated binary supports 5 different parameters for defining input, output, width and height in the rendering process. A special parameter is used for the SSE task. The following calls are some examples.

```
cpurender/cpurender -i input/sphere.obj -o output/sphere.png \
    -w 1920 -h 1080
cpurender/cpurender -i input/prop.obj -o output/prop.png \
    -w 480 -h 270
cpurender/cpurender -i input/head.obj --sse
```

e) Familiarize with the this project.

Play around a little bit! Try to render different input meshes and get a feeling for how long it takes. What happens when you change the output size of the rendered picture?

To use compiler optimizations, you have to recompile the whole application. You can choose between 4 optimization levels 0,1,2 and 3, where 0 is no optimization and 3 most optimized. If you choose the CMake approach you have to modify the compiler flags in the CMake definition (CMakeList.txt) at the very beginning:

```
set(CMAKE_CXX_FLAGS "-O3")
```

At the end, rerun the compilation process described in c).

For manual compilation the optimization flag '-O0' seen in c) has to be replaced with a different optimization level. If you are using the Makefile let it do the work for you:

```
make all OPTIMIZATION=3
(or) make call OPTIMIZATION=3
```

Task 1: Internal and external profiling [2 points]

Before we start with optimization tasks, profiling is essential to determine where the focus should lie. Two methods will be used and evaluated: internal and external profiling. It is strongly recommended to solve this task before proceeding.

HINT: As the code is currently very slow, it can be useful to use compiler optimization and the low face-count sphere mesh object to minimize execution time. You are also allowed to reduce the output image resolution.

a) **[0.75 points][report]** Setup at least one external profiling technique. Do not modify the code in any way.

Document which external profiling technique you have chosen and why. Which information can be retrieved that would help you for an optimizing tasks?

This application has three main blocks: loading the mesh object, rasterisation and writing the PNG image. How long does each block take relative to the total execution time?

Adjust the output image resolution using the paramters -w and -h. How does this affect the rasterisation algorithm? Which parts of it scale more or less with the image resolution?

HINT: Compiler optimization can potentially hide a lot of inefficiencies in your application!

b) **[0.75 points]** Implement an internal profiling technique into the rasterisation algorithm. Do not optimize any given code yet!

Identify which parts of the algorithm are interesting to examine, think about a meaningful unit to measure and implement the needed profiling code into the rasterisation algorithm.

HINT: Solving a) first may help you here.

Focus only on the rasterisation algorithm in 'rasteriser.cpp'! Loading the mesh object or writing out the PNG image should not be of any interest.

c) **[0.5 points] [report]** How long does the Full-HD (1920x1080) rendering of the sphere object take with and without compiler optimization?

You are given 3 more unknown mesh objects in the input folder and you have probably tried rendering them with more or less success. Estimate the time needed to render each of those mesh objects in Full-HD resolution. Explain how you found out.

Using internal and external profiling typically adds overhead to your application. Choose one of your external profiling techniques from a), measure how much slower the application is and give a short idea why this is the case. Did your internal profiling technique from b) add any overhead?

Task 2: Optimization [2.25 point]

In this task you will optimize the rasterisation algorithm in 'rasteriser.cpp'. Document each optimization step and accompany it with measurements that demonstrate the speedup. Solving task 1 first will help you a lot here! Optimization steps without justification, why a speedup is seen, are not considered (but please keep it short).

You are given 4 different mesh objects. During your optimization process, you might want to try larger mesh objects or larger image resolutions to stress the rasterisation process even more. However, it's recommended to first start with the sphere mesh object.

NOTE: Even though compiler optimization is valid and should be considered to the overall speed improvement, it is recommended to keep it switched off for the duration of this task, as it can hide a lot!

a) **[0.25 point]** [**report**] Cache

Give an own short code example and explain how the incorrect usage of cache can lead to worse performance.

NOTE: This is a general question and is not related to the given code/application!

b) **[0.25 point]** [**report**] Loops

Find at least one loop that can be optimized using loop unrolling. Why is this technique actually improving the performance and when can it be applied?

c) **[0.5 point]** [**report**] Memory

Find at least two *different* ways to optimize the memory handling in the application and apply it to the entire rasterisation algorithm. Give a short explanation why it is improving the performance.

d) **[0.25 point]** [**report**] Inlining

Find at least one location in the rasterisation algorithm which is improved through inlining and optimize it. Give a short explanation why it's faster.

e) **[0.5 point]** [**report**] Algorithmic

Find at least *two* algorithmic optimizations which are speeding up the rasterisation process without changing the rendered image.

Explain why they can be applied and do not impact the rendered output.

f) **[0.5 point]** [**report**] With all optimizations done, it should be feasible to render the sphere, prop and head mesh objects.

How long does each object take to render? What can you see on the images rendered from the prop and head mesh file? How big is the total speedup of your optimized rasterisation algorithm?

HINT: If your optimized application is still too slow, lower down the output resolution.

Task 3: SSE Optimization [0.75 points]

With this project a synthetic benchmark is provided crunching numbers on the mesh vertices. This benchmark can be executed using the '–sse' parameter together with an input mesh object.

The benchmark can be found in 'sse_test.cpp' together with its header file. There is no need to touch any other files for this task!

HINT: Switch off compiler optimization for this task!

a) **[0 points]** Set up adequate profiling techniques to measure the runtime of this benchmark.

   Measure the runtime of the sphere, prop, head and scene mesh object file and compare your optimization progress to those initial values!

b) **[0.5 points]** Optimize this benchmark using an SSE data type.

   In the accompanied header 'sse_test.hpp' you can find a definition of an SSE enabled float4 data type. Use this data type to optimize the benchmark algorithm. Your are not allowed to change the 'loadFactor' or to remove any computations!

c) **[0.25 points][report]** Give a short explanation in your own words, what SSE instructions do and why they are faster. Is there an architectural dependency or can it always be used?

   How long does the benchmark take on the sphere, prop, head and scene mesh objects? Compare it to the original times and provide the achieved speedup.

Task 4: Optional Bonus Challenges [up to 0.25 bonus points]

This task is optional. These questions are meant as further challenges or to highlight things you may find interesting.

However, if you successfully complete, you will receive a bonus of 0.25 points (although the maximum score of the assignment is still capped at 5%).

[**report**] What if I told you it's possible to optimize the rasterisation algorithm to an incredible low runtime, rendering the scene mesh object in under 5 seconds. Maybe you have already figured out how to do this!

Which optimization step led to the biggest performance improvement? If not already done in task 2, please give a short explanation why.

Proof your optimized rasterisation algorithm and provide the scene mesh object rendered in 8k resolution (7680 x 4320).

```
cpurender/cpurender -i input/scene.obj -o output/scene.png -w 7680 -h 4320
```