parallel computing Ilker Canpolat & Manuel Göster 13/09/2018

Assignment 01: Profiling and Optimization

Task1: Internal and external profiling

a)
Chosen Profiling technique: callgrind. Implemented by: valgrind tool callgrind + kcachegrind
Why I have chosen this technique: With this tools it is possible to view the execution times in a graphical view. Thus, it is possible to compare runtimes quickly in order to find bottlenecks.
Using this, the functions that take most of the cpu time can be determined. These are the bottlenecks where optimization should start. In this case, most of the time is spent for rasterise() and inside this function, most of the time is used for getTriangleBarycentricWeights() and rasteriseTriangles().

valgrind --tool=callgrind ./cpurender/cpurender -i ./input/sphere.obj -o ./output/sphere.png -w 370 -h 190
loading mesh object: 0.02%
rasterisation: 99.75%
writing png image: 0.2%

valgrind --tool=callgrind ./cpurender/cpurender -i ./input/sphere.obj -o ./output/sphere.png -w 50 -h 50
loading mesh object: 0.02%
rasterisation: 99.35%
writing png image: 0.1%

Loading the mesh object does not depend on the width and heigth parameter at all. So its execution time stays constant when changing -w and -h. Writing the png image scales linear to the -w and -h parameters, as the image size gets bigger when more pixels are used. Rasterisation scales polynomial or exponential to -w and -h.

Changing height and width to lower resolution is faster but the height scales better

b)
foreach pixel, most of the time is spent for
- runFragmentShader
- interpolateNormals
- getTriangleBarycentricWeights

This was measured by running each code piece 100000 times in a loop.

measurement 1 took 7 ms --> getTriangleBarycentricWeights
measurement 2 took 0 ms --> getTrianglePixelDepth
measurement 3 took 16 ms --> interpolateNormals
measurement 4 took 2 ms --> normalise length
measurement 5 took 34 ms --> runFragmentShader
measurement 6 took 0 ms --> Z-clipping

c)
ran on an pool pc
without compiler optimization :

real 2m30.967s
user 2m30.933s
sys 0m0.008s

with compiler optimization (3):
real 0m29.888s
user 0m29.878s
sys 0m0.004s

Assuming constant -w and -h, the runtime mainly depends on how many triangles a mesh obj consists of
--> `for(unsigned int triangleIndex = 0; triangleIndex < triangleCount; triangleIndex++) { ... }`

triangle count:
- sphere.obj: 264
- prop.obj : 7176
- head.obj : 69128
- scene.obj : 379363

my idea: run the program for the first 100 triangles and estimate the total time based on the time it took to compute the first 100 triangles.

times for the first 100 triangles (1920x1080, without compiler optimization):
- sphere: 57362ms
- prop : 57276ms
- head : stopped executing this because it seems like the runtime only depends on the amount of triangles! --> estimation for prop and scene based on first 100 triangles of sphere
- scene :

estimated times of other mesh objects (1920x1080, without compiler optimization):
- sphere: 114724ms --> 1.91min
- head : 4066596ms --> 67.77min
- prop : 39653203ms --> 660.88min
- scene : 217510204ms --> 3625.17min

internal and external profiling overhead:
used sphere.obj with -w 480 -h 270 without compiler optimization

without profiling : 0m9.366s
with internal profiling : 0m0.057s (this is faster because I measure the runtime of one triangle only --> estimated for all triangles: 15.048s
with external profiling : 7m33.080s

Did my internal profiling technique add any overhead? YES

Task 2: Optimization

a) Cache

```
for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++) {
                array[j][i] = 5;
        }
}
```

This code example illustrates cache misses in which spartial locality is violated. So the code jumps around in memory and fetching neighboring elements is not possible to use because they are not in this cache line. The program does not use the neighboring values that are in the cache, so another request is made to the ram memory due to the swapping of columns and rows.

b) Loops

initial times: -i input/sphere.obj -o output/sphere.png -w 800 -h 600

real 0m36.560s
user 0m36.254s
sys 0m0.012s

unrolled loop at line 312, replaced

```
        for (unsigned int i = 0; i < pixelColour.size(); i++) {
                frameBuffer.at(pixelBaseCoordinate + i) = pixelColour.at(i);
    }
```

with

```
        // ---- LOOP UNROLLING ----
        frameBuffer.at(pixelBaseCoordinate)   = pixelColour.at(0);
        frameBuffer.at(pixelBaseCoordinate+1) = pixelColour.at(1);
        frameBuffer.at(pixelBaseCoordinate+2) = pixelColour.at(2);
        frameBuffer.at(pixelBaseCoordinate+3) = pixelColour.at(3);
```

measurement after adding this optimization:

real 0m35.456s
user 0m35.443s
sys 0m0.008s

Explanation: By using loop unrolling the execution time is reduced because the loop counter has to be loaded less often or not at all into a CPU register. This also improves the register distribution. Furthermore, it is not necessary evaluate the loop condition and no counter has to be incremented. This technique can be used anywhere where k iterations can be computed independently of each other.

c) Memory

c1) Move vertex0,1,2 normal0,1,2 and interpolatedNormal to the top of the function in order to allocate memory on the heap only once at the beginning. Free them once at the end. Explanation: This is faster because memory is allocated once and not at each iteration.

measurement after adding this optimization:

real 0m34.792s
user 0m34.783s

sys 0m0.004s

c2) Move them from heap to stack. allocate memory for vertex0,1,2 normal0,1,2 and interpolatedNormal just once. Explanation: In general, stack allocation is faster than heap allocation.
measurement after adding this optimization:
real 0m33.586s
user 0m33.567s
sys 0m0.008s

d) Inlining
inline the function getTrianglePixelDepth. Explanation: The code of the called function gets copied into the calling functions → less function calls → faster, because no overhead for the function calls (build stack, set registers...)

measurement after adding this optimization:
real 0m33.839s
user 0m33.827s
sys 0m0.008s

→ the inline optimization speedup is to small to measure it every run…

e)
e1) call getTriangleBarycentricWeights & interpolateNormals just once. Explanation: Less computation work. Values that are used several times, are computed only once.

measurement after adding this optimization:
real 0m26.764s
user 0m26.750s
sys 0m0.008s

e2) Do not copy float3 and float4 structs when calling functions and compute some values just once in getTriangleBarycentricWeights. Explanation: No unnecessary copying of values.

measurement after adding this optimization:
real 0m25.497s
user 0m25.489s
sys 0m0.004s

f)

How long does each object take to render?
measured with options -w 50 -h 50

sphere.png:
real 0m0.211s
user 0m0.204s
sys 0m0.000s

prop.png:

real 0m5.240s
user 0m5.228s
sys 0m0.001s

head.png:
real 0m38.062s
user 0m37.986s
sys 0m0.064s

scene.png:
real 3m29.927s
user 3m29.561s
sys 0m0.364s

What can you see on the images rendered from the prop and head mesh file?
Prop → a chair
Head → an animal/pokemon? on a chair

How big is the total speedup of your optimized rasterisation algorithm?
initial times: -i input/sphere.obj -o output/sphere.png -w 800 -h 600
real 0m36.560s
user 0m36.254s
sys 0m0.012s

optimized:
real 0m25.497s
user 0m25.489s
sys 0m0.004s

→ 1.43x faster

Task 3: SSE

a)
initial times:
- sphere:

real 0m0.070s
user 0m0.065s
sys 0m0.004s

- prop:

real 0m1.621s
user 0m1.618s
sys 0m0.000s

- head:

real 0m15.409s
user 0m15.392s
sys 0m0.016s

- scene:

real 1m24.469s
user 1m24.413s
sys 0m0.056s

b) See source code directory **task3**
c)
Give a short explanation in your own words, what SSE instructions do and why they are faster.
SSE is the short form of Streaming SIMD Extensions whereas SIMD means Single Instruction Multiple Data. A single instruction of this extension can perform the same calculation on four values (32bit) in parallel.

How long does the benchmark take on the sphere, prop, head and scene mesh objects?
Sse times:
- sphere:

time cpurender/cpurender --sse -i input/sphere.obj
real 0m0.042s
user 0m0.030s
sys 0m0.004s

- prop:

real 0m0.669s
user 0m0.658s
sys 0m0.008s

- head:

real 0m6.207s
user 0m6.198s

sys 0m0.008s

- scene:

real 0m34.346s
user 0m34.263s
sys 0m0.060s
Achieved speedup (scene): 84.469s / 34.346s = 2.459x faster

Is there an architectural dependency or can it always be used?
There is an architectural dependency: it requires aligned memory.