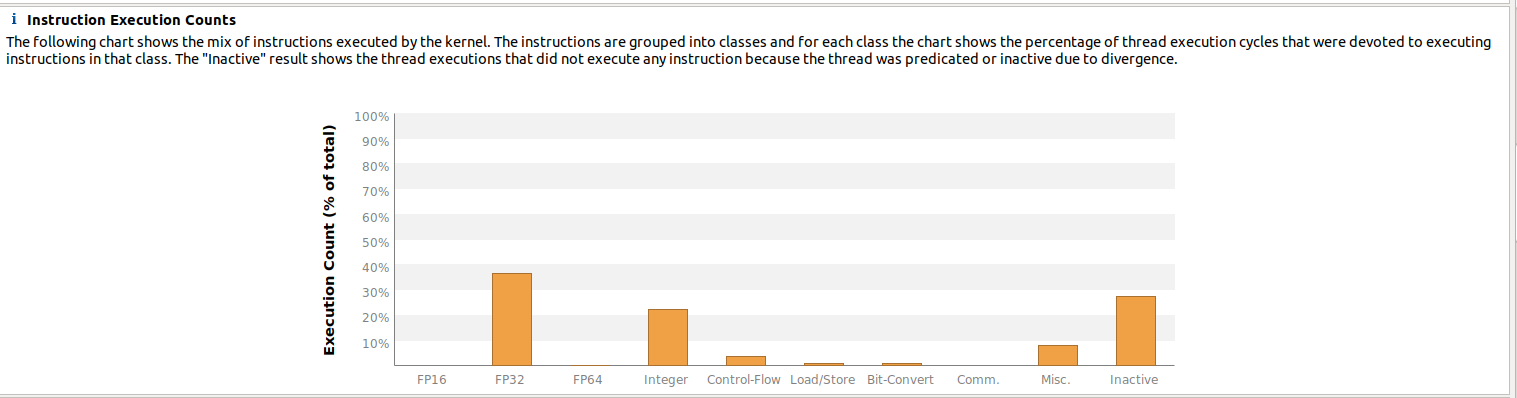
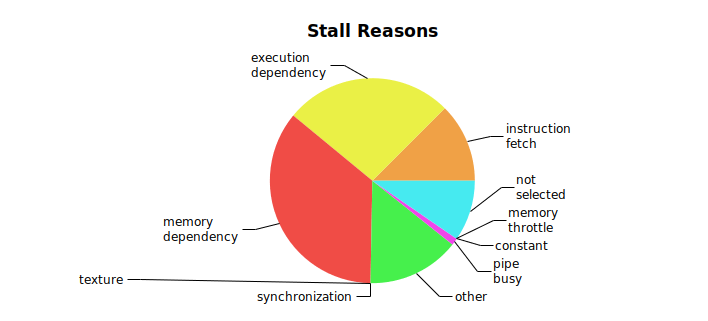
a)

Percentage of time spent

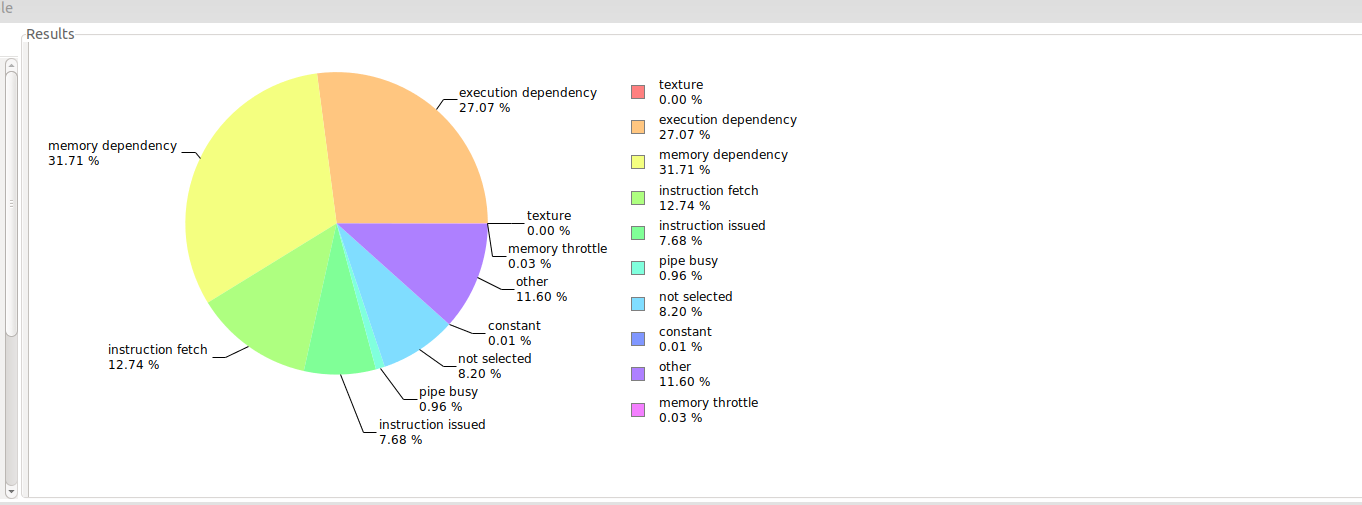


b)

From “kernel latency”



c)

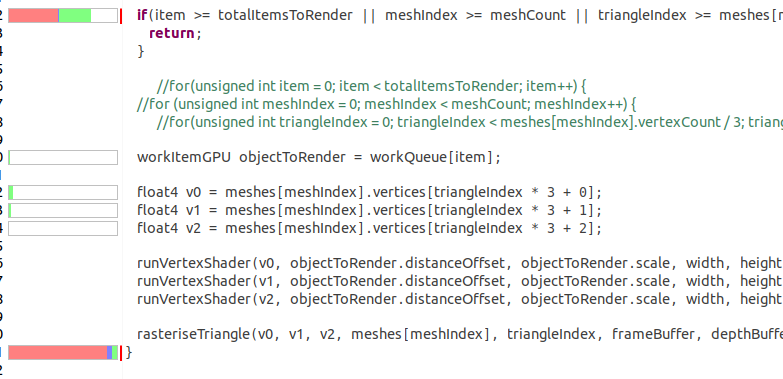
Which might indicate that rougly a third of the time when a warp is unable to execute on a given cycle it is due to memory dependency. Memory dependency is described as when a load/store is unable to be executed, because the resourc is unavailable or is fully utilized.

In this case, we may guess that it is due to the memory buy being busy ( A memory request is being processed)

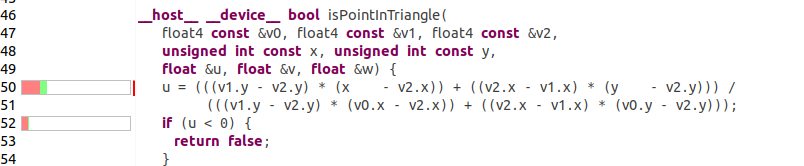
c)

From “Kernel Profile – Instruction execution”

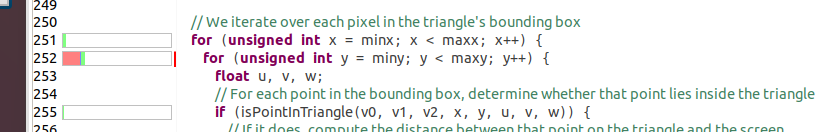
The section of cod that is deffinitely executed by the least ammount of threads in a warp is the else- clause when checking if a thread has a pircr of work to render.

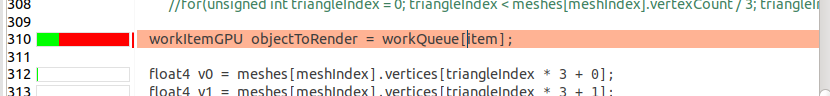
Aditionally, almost half of the threads do not ecvaluate the entire is-statement (Most likely due to short-circuting). But these are not very long sections of code, and we are unable to do all that much about it ( The else has to be threre, and the if is not made any faster by removing the short-circuit,

Additionally, checking if a point is within a triangle is also a bit problematic:



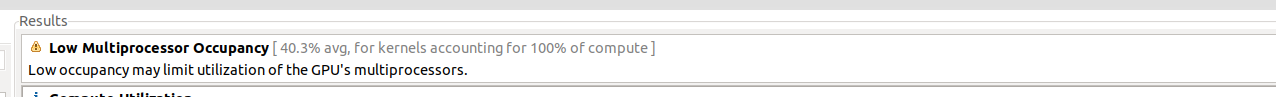
Aditionally, itterating over all the pixels in a triangle can also lead to some problems.



d) Found in “Global memory access pattern”

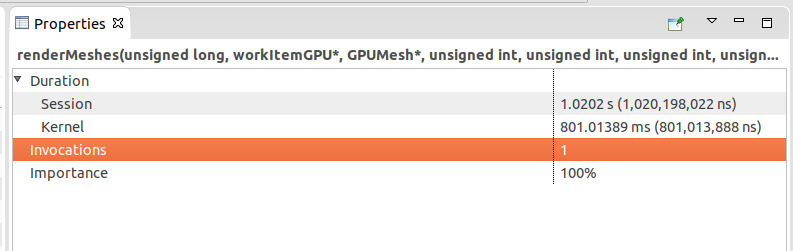
The line with definitely the worst memory-access pattern is getting the items from the work-queue on the GPU.

f) Found in “Compute utilization”



The total achieved occupancy on the kernel is roughly equal to 40%. Since renderMeshes takes up nearly all the GPU-time in the program, the result is that this is also rougly the expected occupancy of the kernel we want to optimize

g )



Because the renderMeshes-kernel only is called from the CPU once, the average execution time is the same as the total exacution-time. As a result. The average execution-time is 801 ms.

2)

a)

Occupancy is given by the total number of warps executing, divided by the maximum possible number of warps that could be executing.

Since (we assume that) most of the time the GPU spends running is spent on computing a task. An increase in occupancy will also mean an increase in the amount of time the GPU spends computing a task, during the same duration of real-time.

b)

The main way the number if warps that can activate simoultaniously on an SM (Streaming Multiprocessor) can be limited by the ammount of memory available on it. If each warp uses a larger fraction of memory than the fraction of available warps, the memory-usage will be the limiting factor on the number runable warps. This is by far the most common limiting factor.

There are also some additional, and far less important potential causes for the limit of runable warps on a GPU:

There is a device limit on the nubmber of warps that can execute on a GPU ( limited number of warps)

Whole blocks are assigned to the SM. This can mean that if a block has a number of warps that do not fit well with the capacaty of the SM, there will be some wasted resources.

c) The term “Latency hiding” refers to scheduling another thread while one thread waits for a memory operation from main memory. The same can be done for a warp in the SM. Latency hiding makes it possible to reduce the perceived delay when accessing memory when only regarding throughput. As a result latency hiding can greatly increase the throughput in some cases.

d)

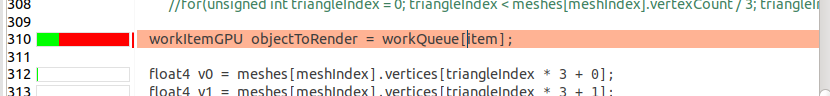
The taks was not siper-clear on what exactly we were supposed to do in order to improve the memory access-pattern, so we will make an assupmtion:

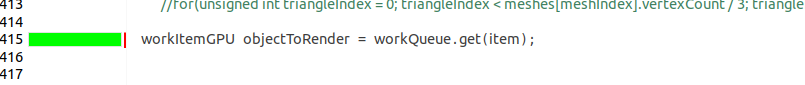
At the moment, only one processing-element accesses a single work-item, so we can not reduce the number of accesses to memory.

Instead, we will try to reduce the number of cache-misses on the work-queue by lumping the variables one warp will acces at once closer together. ( We make a tuple of lists instead of a list of tuples, but we encapsulate if within an object to keep things form getting too messy)

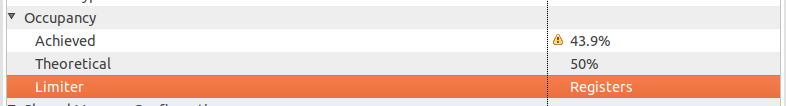
(I thought I had done something wrong, when the rendered spheres were less sparse, compared to earlier exercises, but I ran it against a clean copy and got the same result.)

When comparing the number of addresses per cache-line that were accessed, but not used when reading from the common work-queue. The difference went rougly from a quarter of the data being useful, to almost all of it being used.



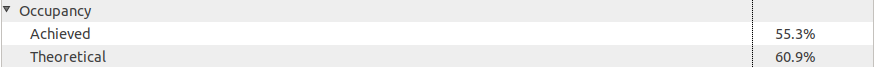


When comparing the occupancy, it went from 43.9% to 43.9% …...

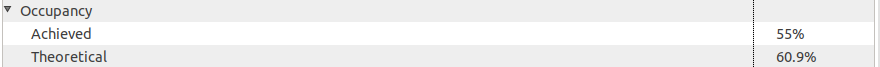


e)

1) (4, 24)



2) (6,16)



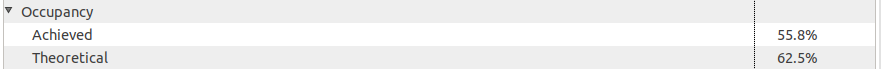
3) (8,8)



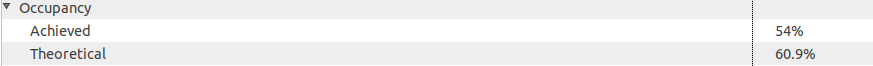
4) (8,12)



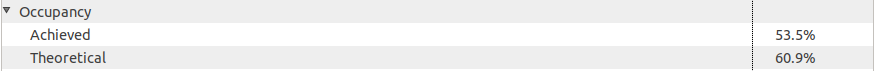
5) (8,16)



6) (12,8)



7) (16x6)



8) (16,8)



Testing a number of different block dimensions, block dimensions of 8,16,1 gave the best occupancy. That is threadsPerWorkQueueBlock = 8, threadsPerVertexBlock = 16.

f) With the previous improvements, the execution of the renderMeshes kernel is betteen 470 and 530ms, compared to between 760 and 820ms in the beginning. A nice speedup of 1.58.

Task 3

a) Because threads are scheduled as warps, and when one thread is running longer than the others in the same warp, potentially a lot of cores will be idle.

b)

--depth=5

--height=1080

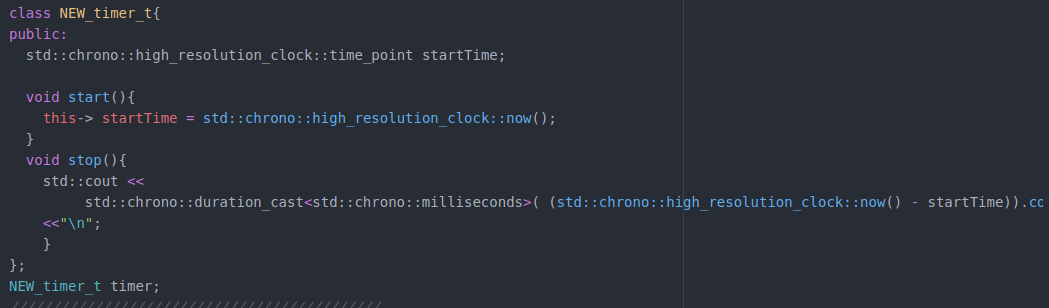
--width=1920



The ocupancy improved rather significantly, as it became 45.3%

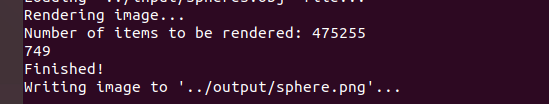
c)

We compare the running time by using the std::chrono library

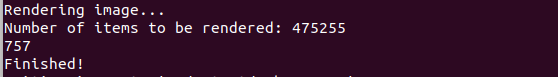


We put start before calling renderMeshes<<<>>>(), and stop after cudaDeciceSyncronise(): The reuslt is:

New:



Old:



The run-time for the kernel went from 0.757 seconds to 0.749 seconds. Thevery next iteraion of running the refference- program, we got 0.739. Thus, we may conclude that there was hardly any gin form trying to share the load.