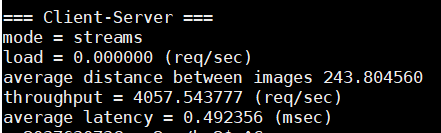
**Accelerators and Accelerated Systems**

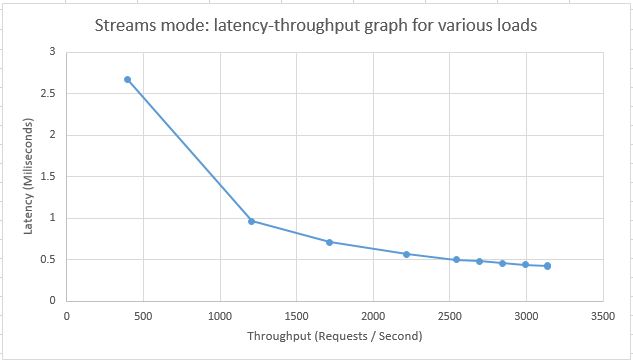
**HW 2 – report**

1. **Streams Implementation**
   1. Throughput for **load = 0**: *4057.543777* req/sec



* 1. Number of run with various loads:

|  |  |  |
| --- | --- | --- |
| **throughput (reqs/sec)** | **latency (msec)** | **load** |
| 398.660753 | 2.673019 | 405.7544 |
| 1206.531737 | 0.964623 | 1262.347 |
| 1715.66365 | 0.707637 | 2118.94 |
| 2218.175088 | 0.563871 | 2975.532 |
| 2540.675385 | 0.502209 | 3832.125 |
| 2693.659502 | 0.478339 | 4688.717 |
| 2842.119098 | 0.457451 | 5545.31 |
| 2991.635421 | 0.440879 | 6401.902 |
| 3138.569202 | 0.422929 | 7258.495 |
| 3135.346558 | 0.426348 | 8115.088 |



Observations:

1. as we increase the load we get better latency and throughput.
2. For lower loads, increase in load by improves latency exponentially. As we reach higher loads we can see that the latency improves linearly.
3. In every increase in load, throughput improves by smaller amounts (exponentially).

The results we are getting make sense. If we set the load to a low value, the gpu does not receive tasks fast enough to be efficient, i.e. it sometimes stays idle waiting for tasks to be sent to it or for finished calculation results to be collected from it. As we increase the load, tasks are sent to it fast which then improves the GPU's utilization, resulting in increase in throughput and decrease in latency. As we are sending more and more tasks in smaller intervals we are beginning to reach the GPU maximum compute capability, causing the performance to improve by smaller and smaller amounts.

1. **Producer-Consumer Implementation**

2.1. In order to calculate the maximum number of threadblocks to run, we consider these three resources:

- Shared memory: Shared memory is used to calculate the histograms and distances. We obviously do not want to exceed the maximum possible shared memory size.

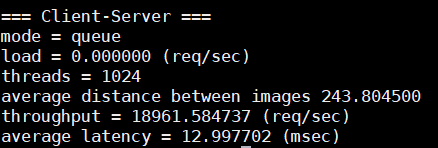
Therefore: We divide the SM shared memory by the shared memory we use in each threadblock, and multiply the result by the number of SMs.

- SM boundary: There is a limitation for the number of threads which can run on a single SM. By dividing that number by the number of threads in each thread block, we can get the number of threadblocks which can run concurrently run on a SM. Afterwards we should multiply the given result by the number of SMs in the system in order to get the upper bound for the total number of threadblocks.

- # of registers: The compilation command assigns 32 registers for each thread. Since each SM has a fixed number of registers, we will need to divide that number 32, which will provide us with the threads per SM limit. Dividing that result with the number of threads per threadblock will result with the threadblock boundary per SM. We then multiply the result with the number of SMs of course.

The maximum number of threadblocks is the minimum of the above limitations (strictest limitation).

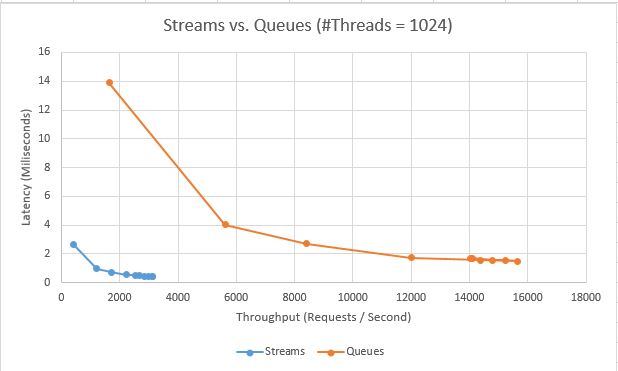
2.4.1. Throughput for **load = 0, #threads = 1024**: *18961.584737* reqs/sec.



2.4.2.

|  |  |  |
| --- | --- | --- |
| **throughput (reqs/sec)** | **latency (msec)** | **load** |
| 1646.689302 | 13.890456 | 1896.158 |
| 5620.734362 | 4.031477 | 5899.16 |
| 8415.277509 | 2.692663 | 9902.161 |
| 12007.39557 | 1.727553 | 13905.16 |
| 15644.21032 | 1.494067 | 17908.16 |
| 15237.97619 | 1.548804 | 21911.16 |
| 14374.48697 | 1.552689 | 25914.17 |
| 14796.64012 | 1.570418 | 29917.17 |
| 14093.87498 | 1.682353 | 33920.17 |
| 14027.64226 | 1.705405 | 37923.17 |

2.4.3.

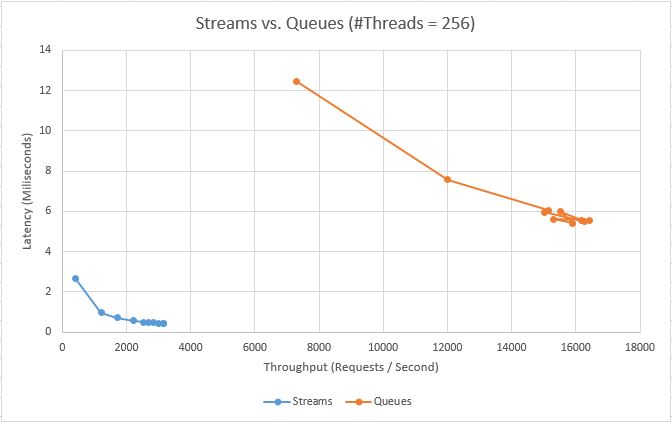


2.5.1. Throughput for **load = 0, #threads = 256**: *77630.1575* reqs/sec.

2.5.2.

|  |  |  |
| --- | --- | --- |
| **throughput (reqs/sec)** | **latency (msec)** | **load** |
| 7278.434227 | 12.464941 | 7763.01575 |
| 11999.54416 | 7.579962 | 24151.60455 |
| 15140.80654 | 6.049534 | 40540.19336 |
| 15011.61421 | 5.960775 | 56928.78216 |
| 16181.7284 | 5.529621 | 73317.37097 |
| 16260.08514 | 5.510266 | 89705.95977 |
| 15530.59976 | 5.993923 | 106094.5486 |
| 15906.19076 | 5.40157 | 122483.1374 |
| 15302.66474 | 5.614358 | 138871.7262 |
| 16429.39771 | 5.564964 | 155260.315 |

2.5.3.

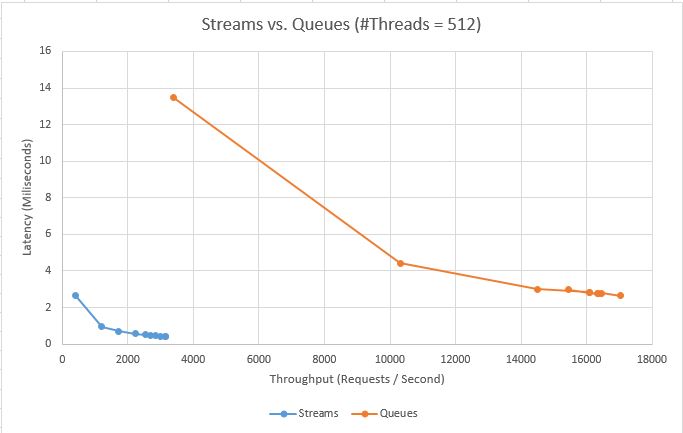


2.6.1. Throughput for **load = 0, #threads = 512**: *36106.30203* reqs/sec.

2.6.2.

|  |  |  |
| --- | --- | --- |
| **throughput (reqs/sec)** | **latency (msec)** | **load** |
| 3397.67853 | 13.4826 | 3610.630203 |
| 10308.50258 | 4.408692 | 11233.07174 |
| 14497.71403 | 3.013062 | 18855.51328 |
| 16076.24284 | 2.828425 | 26477.95482 |
| 16084.56334 | 2.81903 | 34100.39636 |
| 16374.32605 | 2.745726 | 41722.8379 |
| 15456.81375 | 2.97238 | 49345.27944 |
| 16327.76066 | 2.74874 | 56967.72098 |
| 16469.22495 | 2.770487 | 64590.16252 |
| 17040.65127 | 2.638487 | 72212.60406 |

2.6.3.



2.7. First, we can denote that the highest maxLoad was received in the case where #threads is 256. When comparing the graphs while considering the different loads, we see that running with 256 threads gives the best throughputs for lower loads. This is due to having the highest maxLoad which results in a higher maxLoad/10. But, as we increase the load, the advantage of running with 256 shrinks and we begin to receive similar throughputs among the different workgroup structures.

Moreover, we can denote that the higher the threads amount, the lower the latency – especially in large scale throughputs.

2.8. Such implementation would most likely result in better performance, here's why:

Suppose the CPU-to-GPU queue is in the host (i.e. CPU) memory.

Every action that involves access to the other party's memory is very expensive, time-wise.

Since in our example, the CPU-to-GPU queue is in the CPU memory, the CPU writes to its local memory, while the GPU reads from the remote party's memory.

One can observe that for the write action, we can send data, and, when asynchronous, keep working. However, for the read action, we have to wait for the other party to receive our request, and for its response to return, in order to resume working.

Since the read action is much more expensive, we would want the reads to be local and the writes to be remote. That is why we would want the CPU-to-GPU queue to be in the GPU, and the GPU-to-CPU queue to be in the CPU.

2.9. In order for the GPU memory to be accessible by the CPU, the GPU global memory need to be exposed and mapped onto the CPU address space, i.e. the OS needs to map the GPU's memory onto the CPU's address space using PCIe BAR. That way, whenever the CPU wants to produce, i.e. send data, it can access the GPU's part of the address space.