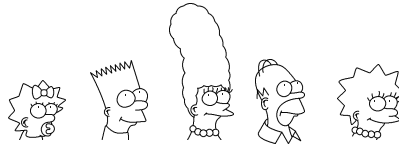


046278
Accelerators and Accelerated Systems
Assignment 2

Avraham Ayaso - 305036352, Yonatan Nakonechny - 200752061

June 5, 2020



1 CUDA Streams

1.2 Run the program in streams mode with load=0 and report the throughput in the report. We'll refer to the throughput you get here as maxLoad.

```
u_305036352@gpu-03:~/hw/hw2$ ./ex2 streams 0
Number of devices: 1

=== Randomizing images ===
total time 65.869049 [msec]

=== CPU ===
total time 47.394265 [msec]

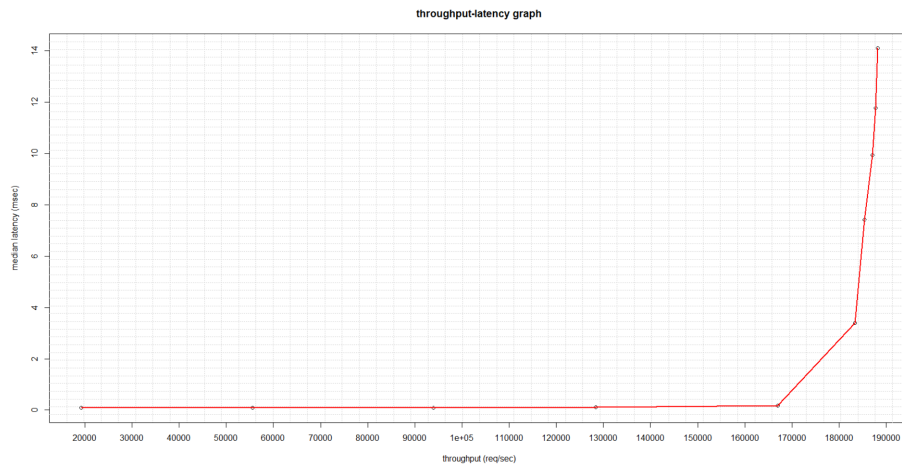
=== Client-Server ===
mode = streams
load = 0.0 (req/sec)
distance from baseline 0 (should be zero)
throughput = 186490.5 (req/sec)
latency [msec]:
      avg      min      median 99th perc.      max
      0.0160    0.0121    0.0159    0.0179    0.0535
```

$$\text{maxLoad} = 186490.5 \left[\frac{\text{req}}{\text{sec}} \right]$$

1.3 Vary the load from load=maxLoad/10 to load=maxLoad*2, in 10 equal steps. In each run write down the load, latency and throughput in a table in the report.

load (req/sec)	median latency (msec)	throughput (req/sec)
18649.05	0.0819	19272.8
55947.15	0.0820	55580.7
93245.25	0.0865	93921.6
130543.35	0.1031	128357.0
167841.45	0.1794	166959.2
205139.55	3.3862	183348.8
242437.65	7.4288	185343.4
279735.75	9.9496	187086.8
317033.85	11.7804	187728.8
372981	14.1029	188194.3

1.4 From the samples you collected, draw a throughput-latency graph: X-axis is the throughput, and Y-axis is the median latency. Make sure to annotate the axes with clear names, units and values. Use linear scale for the X-axis. Make sure that the sample points are marked in the graph. Add the graph to the report and explain it (what can we learn from it?)



Conclusions from the graph:

- The throughput **increases** as we increase the load **until** we reach a peak around maxLoad. Any increase in the load beyond maxLoad won't affect the throughput.
 - As we found out when we ran the program without limits, maxLoad is the maximum throughput possible, so it makes sense that we can utilize the resources until we reach that max point.
- The latency increase rate **increases** as we increase the load.
 - This also makes sense because as we get close to maxLoad we reach a point in which the input req/sec rate is getting bigger than the output req/sec, therefore we are having more jobs that cannot be processed and have to wait for an available stream.

2 Producer Consumer Queues

2.1 “In your code compute how many threadblocks can concurrently run in the GPU... Explain how you compute this number in the report.”

In order to determine the amount of threadblocks that can run concurrently on the GPU we will look at the three following limits on the number of concurrent threadblocks on a specific SM, take the minimum and multiply by the total amount of SMs.

$$L_1 = \frac{TotalSharedMemPerSM}{UsedSharedMemPerBlock}$$

$$L_2 = \frac{TotalRegsPerSM}{UsedThreadsPerBlock \cdot RegistersPerThread}$$

$$L_3 = \frac{MaxThreadsPerSM}{UsedThreadsPerBlock}$$

$$N_{ThreadBlocks} = N_{SM} \cdot \min(L_1, L_2, L_3)$$

2.4.1. Run the program in queue mode with `#threads = 1024` and `load = 0` and report the throughput in the report. We’ll refer to the throughput you get here as `maxLoad`.

$$maxLoad_{1024} = 1184046.8 \left[\frac{req}{sec} \right]$$

2.4.2. Vary the load from `load = maxLoad / 10` to `load = maxLoad * 2`, in ~10 equal steps. In each run write down the load, median latency and throughput in a table in the report..

Load	Throughput	Median Latency
0	1184046.8	1.8446
118404	117278.1	0.0237
368370	367194.7	0.0252
618335	618235.4	0.0270
868300	864545.3	0.0296
1118266	1114736.2	0.0373
1368231	1180128.0	0.5480
1618197	1170631.5	1.2046
1868162	1194757.3	1.4524
2118128	1147405.2	1.8971
2368093	1151790.6	2.2022

2.5.1-2.5.2 Repeat 2.4.1-2.4.2 with #threads=512.

$$maxLoad_{512} = 1202951.1 \left[\frac{req}{sec} \right]$$

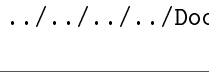
Load	Throughput	Median Latency
0	1202951.1	1.3091
120295	119536.7	0.0399
374251	372034.4	0.0435
628207	620782.6	0.0503
882164	877988.3	0.0597
1136120	1131326.2	0.0771
1390076	1212255.4	0.4203
1644033	1210566.0	0.9972
1897989	1211251.6	1.3985
2151945	1203784.7	1.7417
2405902	1206065.2	1.9867

2.6.1-2.6.2 Repeat 2.4.1-2.4.2 with #threads=256.

$$maxLoad_{256} = 1283840.6 \left[\frac{req}{sec} \right]$$

Load	Throughput	Median Latency
0	1283840.6	0.1820
128384	125937.6	0.0775
399417	393653	0.0889
670450	658882	0.0992
941483	888723.7	0.3726
1212516	1102721.2	0.4063
1483549	1273908.7	0.5419
1754582	1276879.1	1.0528
2025615	1278732.8	1.4969
2296648	1276408.1	1.7430
2567681	1273181.5	2.0147

2.7 From the samples you collected, create a latency-throughput graph which compares all 4 previous experiments (streams, queues with `#threads=1024`, queues with `#threads=512`, and queues with `#threads=256`). Make sure to add a legend to the graph.



../../../../Documents/GitHub/acc_hw2/pasted3.png

2.8 In the report, explain what we can learn from the differences between the throughput- latency graphs with different `#threads`.

- The GPU-CPU shared memory is a much **quicker** method than copying memory as we did in the stream server.
- The amount of concurrent threadblocks is the important factor in terms of performance.
 - The **decrease** in the **number of threads per block** allows us to use **more concurrent threadblocks**, which results in an **increase in the throughput**.
- Again, as we saw in the streams method, the **increase** in load causes an **increase** in throughput only **untill** we reach the `maxLoad`.

2.9. A wise man suggested to move the CPU-to-GPU queue to the memory of the GPU (assume it is still accessible by CPU), and to keep the GPU-to-CPU queue in the CPU memory. He claimed it might result in better performance. Explain why. Think in the terms: PCIe reads and writes, posted and non-posted transactions.

Because `write()` transactions are posted (doesn't wait for completion), and `read()` transactions are non-posted, we would get better performance if we could use `write()` transactions between the CPU-GPU instead of `read()` transactions. Therefore, because the CPU-to-GPU queue is on the memory of the GPU, the CPU uses `write()` in order to push new requests to the GPU inside the CPU-to-GPU queue and the GPU just reads from his own memory. And the same goes for the GPU-to-CPU queue, the GPU uses `write()` in order to push finished jobs to the CPU inside the GPU-to-CPU queue and the CPU just reads from his own memory.

2.10. In order to place the CPU-to-GPU queue in the GPU memory, we will need to make it accessible by CPU. Explain roughly what should be done for this to happen (In terms of PCIe MMIO).

The CPU needs to map the addresses of the CPU-to-GPU queues (which are in the GPU memory) to his virtual memory address space (MMIO). In order for that to be possible, the GPU needs to implement the interface between the PCIe and itself (via the GPU's driver), and expose his memory for write transactions.