Accelerators and accelerated systems 046278
HW 02

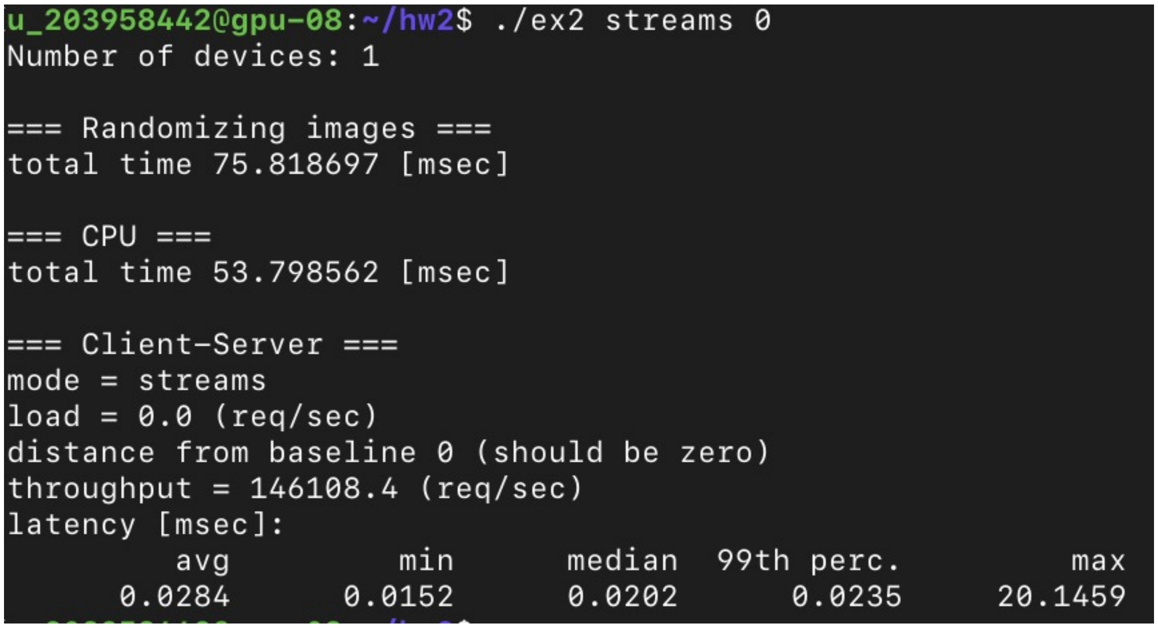
May 30, 2020

Barak Gahtan	203958442
Nadav Gelfer	304849821

Q1

SQ1.2

Running with $load = 0$ we get $maxLoad = 146108$:

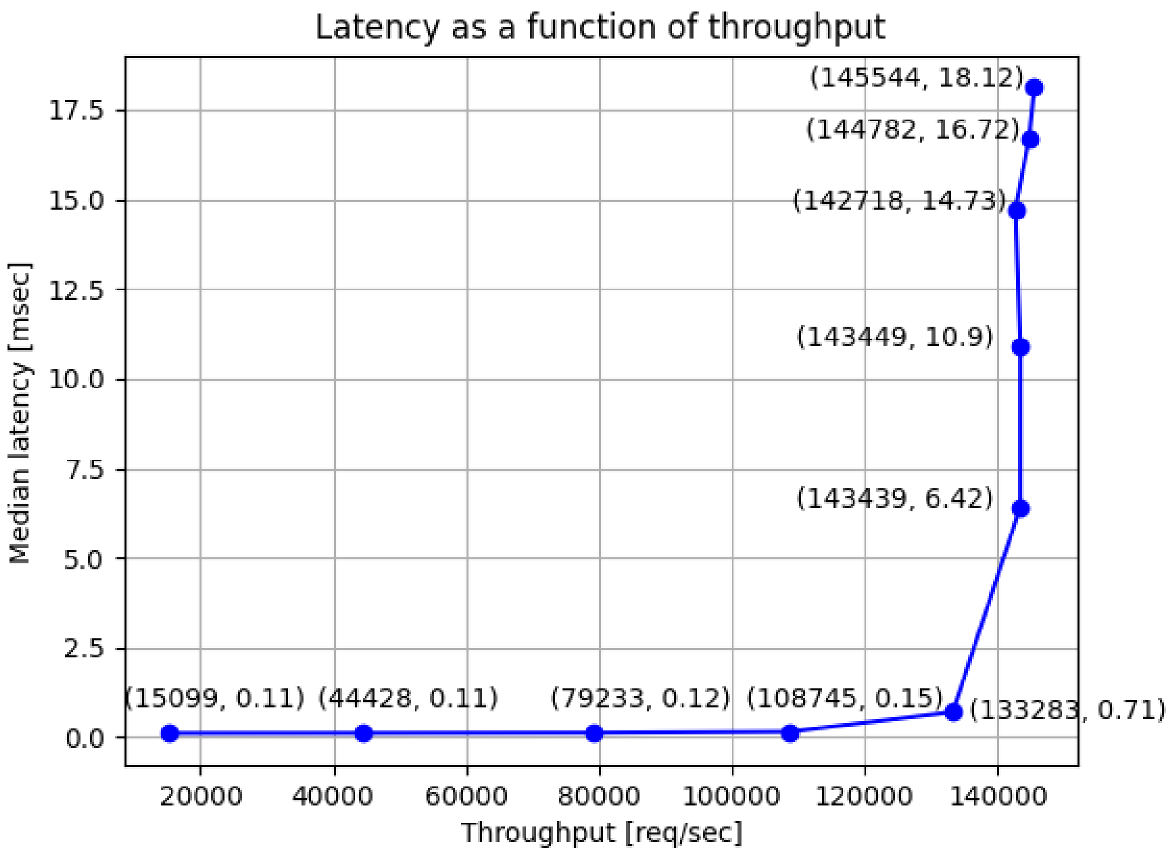


```
u_203958442@gpu-08:~/hw2$ ./ex2 streams 0
Number of devices: 1

=== Randomizing images ===
total time 75.818697 [msec]

=== CPU ===
total time 53.798562 [msec]

=== Client-Server ===
mode = streams
load = 0.0 (req/sec)
distance from baseline 0 (should be zero)
throughput = 146108.4 (req/sec)
latency [msec]:
          avg          min       median   99th perc.          max
       0.0284       0.0152       0.0202       0.0235      20.1459
```

SQ1.3

step	load [req/sec]	tp [req/sec]	latency [msec]
1	14611	15099	0.1102
2	45456	44428	0.1144
3	76301	79233	0.1229
4	107146	108745	0.1517
5	137991	133283	0.7058
6	168836	143439	6.4154
7	199681	143449	10.8993
8	230526	142718	14.7343
9	261371	144782	16.7218
10	292216	145544	18.1211

SQ1.4



We can see that as the throughput rises the median latency rises as well, with a big jump towards our 6th iteration.

When we ran the program with $load = 0$ we got the maximum throughput possible (i.e. ideal conditions). We can then see that as long as we don't "choke" the device with maximal throughput, we get low median latency.

On our 6th iteration we reach our maximum throughput and so every increase in load over the maximal throughput results in diminishing in throughput and huge payoff in median latency.

Graph numbers are rounded for readability.

Q2

SQ2.1

Let us define the following:

$$\begin{aligned} r_1 &= \frac{\text{maximum number of threads per SM}}{\text{required number of threads per TB}} \\ r_2 &= \frac{\text{number of registers per SM}}{\text{number of required registers per TB}} \\ r_3 &= \frac{\text{shared memory per SM}}{\text{required shared memory per TB}} \\ \alpha &= \text{number of SM} \end{aligned}$$

Using these we get the maximum number of threadblocks by:

$$maxNumberOfTB = \min(r_1, r_2, r_3) \cdot \alpha$$

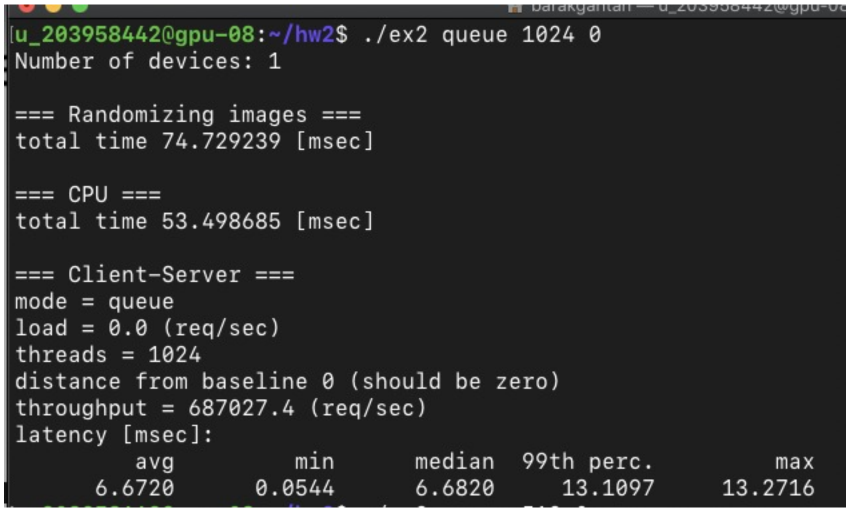
r_1 is dictated by the number of maximum number of threads per SM and it is computed by dividing that number by the number of threads per threadblock.

r_2 is dictated by the number of registers per SM and it is computed by dividing that number by the number of registers per threads (which was set at 32) multiplied by the number of threads per threadblock.

r_3 is dictated by the size of shared memory per SM, and it is computed by dividing that number by the size of required shared memory per TB.

SQ2.4.1

Running with $\#threads = 1024$ and $load = 0$ we get $maxLoad = 687027$:



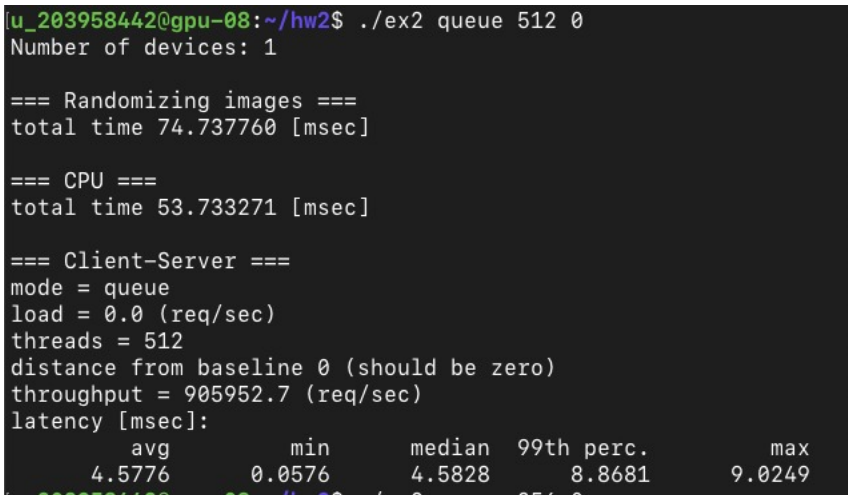
```
u_203958442@gpu-08:~/hw2$ ./ex2 queue 1024 0
Number of devices: 1

=== Randomizing images ===
total time 74.729239 [msec]

=== CPU ===
total time 53.498685 [msec]

=== Client-Server ===
mode = queue
load = 0.0 (req/sec)
threads = 1024
distance from baseline 0 (should be zero)
throughput = 687027.4 (req/sec)
latency [msec]:
          avg         min      median   99th perc.        max
       6.6720      0.0544      6.6820     13.1097     13.2716
```

SQ2.4.2

step	load [req/sec]	tp [req/sec]	latency [msec]
1	68703	68385	0.8843
2	213742	209715	0.9175
3	358781	345194	0.9715
4	503820	478009	0.9837
5	648859	608681	1.0191
6	793898	674852	1.4821
7	938937	689177	2.2606
8	1083977	673262	3.0877
9	1229016	695770	3.2812
10	1374055	693810	3.7468

SQ2.5.1

Running with $\#threads = 512$ and $load = 0$ we get $maxLoad = 905952$:



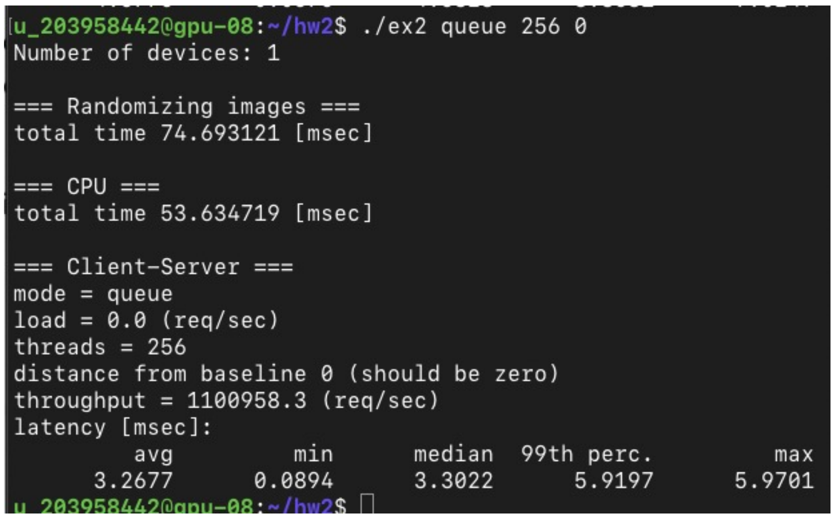
```
u_203958442@gpu-08:~/hw2$ ./ex2 queue 512 0
Number of devices: 1

=== Randomizing images ===
total time 74.737760 [msec]

=== CPU ===
total time 53.733271 [msec]

=== Client-Server ===
mode = queue
load = 0.0 (req/sec)
threads = 512
distance from baseline 0 (should be zero)
throughput = 905952.7 (req/sec)
latency [msec]:
         avg         min      median   99th perc.          max
      4.5776      0.0576      4.5828       8.8681       9.0249
```

SQ2.5.2

step	load [req/sec]	tp [req/sec]	latency [msec]
1	68703	68455	0.9728
2	213742	208004	1.0399
3	358781	343569	1.1062
4	503820	468709	1.3526
5	648859	588647	1.5131
6	793898	714923	1.4060
7	938937	816910	1.4947
8	1083977	844722	1.8299
9	1229016	868100	2.0707
10	1374055	872261	2.4138

SQ2.6.1

Running with $\#threads = 256$ and $load = 0$ we get $maxLoad = 1100958$:

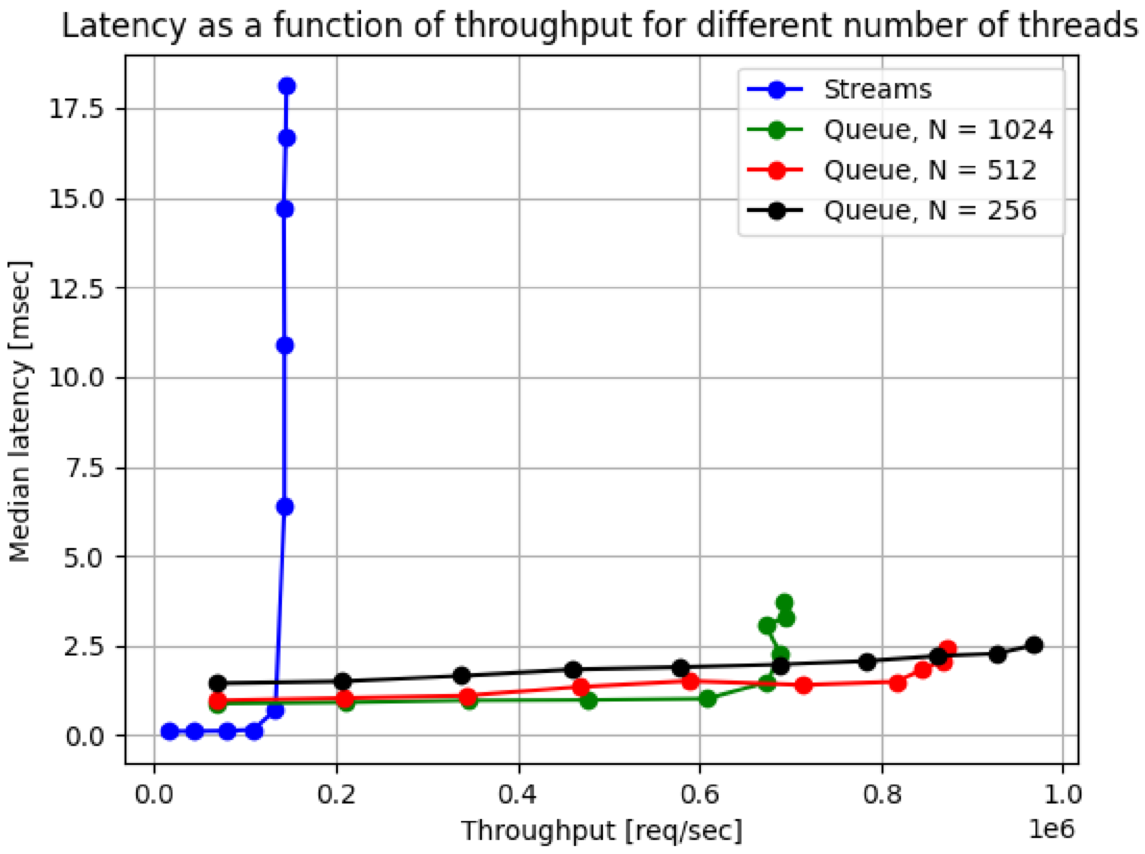


```
u_203958442@gpu-08:~/hw2$ ./ex2 queue 256 0
Number of devices: 1

=== Randomizing images ===
total time 74.693121 [msec]

=== CPU ===
total time 53.634719 [msec]

=== Client-Server ===
mode = queue
load = 0.0 (req/sec)
threads = 256
distance from baseline 0 (should be zero)
throughput = 1100958.3 (req/sec)
latency [msec]:
         avg         min      median  99th perc.         max
      3.2677      0.0894      3.3022      5.9197      5.9701
u_203958442@gpu-08:~/hw2$
```

SQ2.6.2

step	load [req/sec]	tp [req/sec]	latency [msec]
1	68703	68291	1.4510
2	213742	206075	1.5027
3	358781	337997	1.6594
4	503820	459416	1.8344
5	648859	579072	1.8981
6	793898	689058	1.9762
7	938937	783736	2.0737
8	1083977	861633	2.2105
9	1229016	927247	2.2795
10	1374055	968818	2.5200

SQ2.7



SQ2.8

The first thing we can see is that the median latency using queues is smaller by an order of magnitude compared to that of the streams (maximum median latency using streams is almost at 20 compared to the maximum median latency using queues, which is about 4).

Comparing the queues graphs we can see that by increasing the number of threads we get lower throughput. Since more threads per threadblock means every threadblock task will be completed faster (meaning we get lower latency per task). However, increasing the number of threads per threadblock reduced the overall number of threadblocks meaning we have less parallelism. Since we have a queue per threadblock, increasing the number of threads reduces the number of queues, and that is why the throughput goes down as we increase the number of threads.

SQ2.9

We should expect better performance using the proposed queue structure. For the CPU to GPU queue, the CPU performs write operations and the GPU performs read operations.

When the GPU accessed the main memory it is done with a PCIe transaction. The same thing applies for the CPU when writing to the GPU memory. These transactions are costly and are subject to hardware limitations such as latency and bandwidth and will lead with high probability to a computational bottleneck. Hence we should reduce these to a minimum.

In order to reduce PCIe transactions and optimize our queue latency, we want to minimize the non-posted transactions (reads) over PCIe. This can be achieved by moving the CPU to GPU queue closer to the GPU which performs the reads.

SQ2.10

In order to enable the CPU to access the GPU memory, we will use MMIO. Using MMIO will enable the CPU to access the GPU memory the same way it accesses every other memory address.

We will need to map the GPU memory physical addresses to the CPU virtual memory by asking the OS to configure the CPU MMU in such way that will allow the mentioned setup.