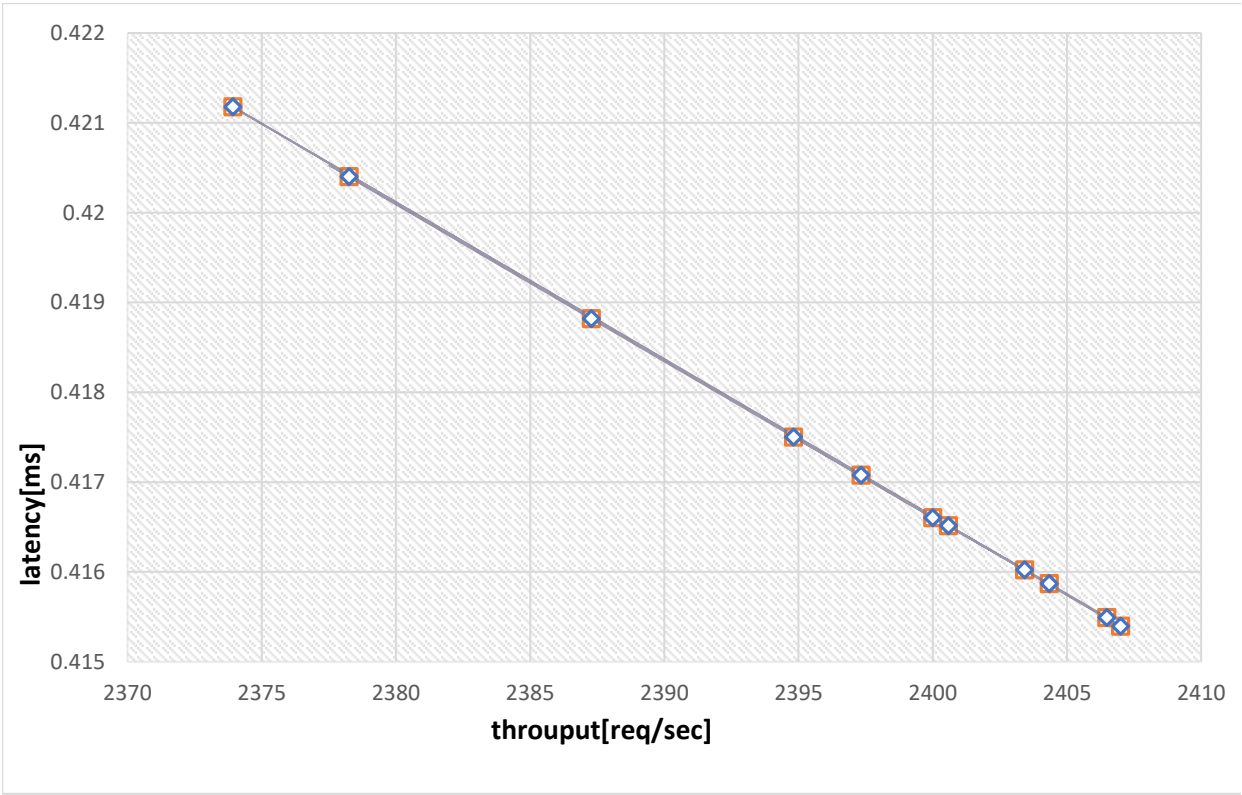
Report HW2

1. CUDA Streams

1.2 throughput = 2412.631436 (req/sec)
1.3

load	throughput[req/sec]	latency[ms]
241.2631436	2406.490172	0.415492
699.6631164	2404.344481	0.415869
1158.063089	2400.581536	0.416514
1616.463062	2373.92619	0.421178
2074.863035	2407.006967	0.415394
2533.263008	2378.252612	0.420403
2991.662981	2387.279924	0.418819
3450.062953	2399.999461	0.416603
3908.462926	2397.327892	0.417078
4366.862899	2403.422492	0.416023
4825.262872	2394.815799	0.417503

1.4 The higher the throughput, the lower the latency.



2. Producer Consumer Queues

2.1

I calculated the minimum number of Blocks Per Multiprocessor . Then multiplying it by number of Multiprocessor;.

we have considered: Threads Per Block, Registers Per Block, Shared Memory Per Block.

every one of these resources affect the number of concurrently scheduled blocks. For example, If you start increasing Shared Memory Per Block, the Max Blocks per Multiprocessor will continue to be your limiting factor until you reach a threshold at which Shared Memory Per Block becomes the limiting factor.

* **Threads Per Block**, limited by Max Blocks per Multiprocessor

num1= Max Blocks per Multiprocessor / used Threads Per Block

* **Registers Per Block** , limited by Max Registers per Multiprocessor

num2= Max Registers per Multiprocessor/ used Registers Per Block

* **shared memory Per Block**, limited by Max shared memory per Multiprocessor

num3= Max shared memory per Multiprocessor/ used shared memory Per Block

number of blocks = min(num1,num2,num3)* (number of Multiprocessors)

2.4.1 throughput = 27118.931345 (req/sec)

2.4.2

load	throughput[req/sec]	latency[ms]
2711.893135	27330.17061	0.107077
7864.490091	26350.81713	0.112275
13017.08705	27837.64185	0.105529
18169.684	26982.09011	0.104506
23322.28096	27369.21445	0.107942
28474.87791	28937.87541	0.104252
33627.47487	27258.50997	0.107274
38780.07182	28715.66792	0.103587
43932.66878	27768.06076	0.105962
49085.26573	27071.07612	0.110915
54237.86269	28297.20075	0.105481

2.5.1 throughput = 29308.076107 (req/sec)

2.5.2

load	throughput[req/sec]	latency[ms]
5460.183055	53334.16668	0.069764
15834.53086	53347.85117	0.070039
26208.87866	52078.43297	0.067868
36583.22647	52966.70443	0.070093
46957.57427	55791.42507	0.066511
57331.92208	53999.18263	0.066238
67706.26988	54479.5316	0.069204
78080.61768	56671.09871	0.063062
88454.96549	53743.82983	0.070536
98829.31329	56700.20321	0.063111
109203.6611	52212.79774	0.067913

2.6.1 throughput = 438031.76586 (req/sec)

2.6.2

load	throughput[req/sec]	latency[ms]
4380.317659	42384.76384	0.070809
47745.46248	41249.86279	0.072417
91110.60731	41042.65992	0.071804
134475.7521	23728.90193	0.124551
177840.897	43784.35351	0.067981
221206.0418	41415.57128	0.070079
264571.1866	43807.01811	0.06795
307936.3314	41701.91884	0.072335
351301.4763	42344.88962	0.06982
394666.6211	42344.88962	0.06982
438031.7659	42043.28734	0.069738

2.7 when threads number is higher, the number of threads per block is lower, which increases the program latency.

2.8 Transfers between the host and device are the slowest link of data movement involved in GPU computing. So if we move the queue form CPU to GPU ,we save reading transfers across the PCI-e bus form CPU to GPU . why it happens (CPU still reads responses!)?
the reason is, that GPU is faster than CPU.
In conclusion, considering the cost of moving data across the PCI-e bus, the new program performance will increase - because of minimizing data transfer.

2.9
MMIO: the memory of GPU is memory mapped I/O in CPU.
when CPU reading from these space addresses of memory ,it will be across the PCI-e bus, until getting the GPU “translated” memory.
the same issue when writing data from CPU to GPU.