מאיצים חישוביים ומערכות מואצות
046853

אביב תשע"ט
תרגיל בית 2

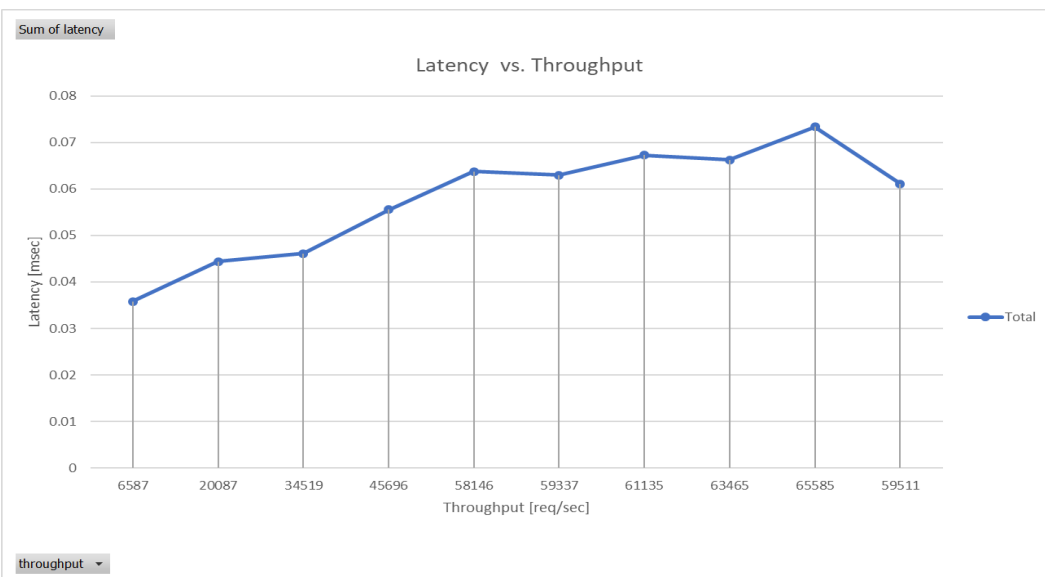
שם: יחזקאל עידו **תעודת זהות:** 204397368

שם: זוהר אוהד **תעודת זהות:** 305285694

1. CUDA Streams:

Average max load is: 66,000 [request / sec].

Iteration #	Load [request/sec]	Throughput [request/sec]	Latency [mSec]
1	6,600	6587.57	0.035831
2	20,533	20087.81	0.044421
3	34,666	34519.61	0.046088
4	48,400	45696.43	0.055533
5	62,333	59511.10	0.061123
6	76,266	58146.55	0.063793
7	90,200	63465.94	0.066275
8	104,133	61135.12	0.067265
9	118,066	59337.43	0.062947
10	132,000	65585.75	0.073375



We can learn from the graph 3 main conclusion:

- As the throughput is getting higher also the latency is getting higher. We assume this happen because the GPU must handle more request in parallel.
- The throughput is little below the load we set because of some management overheads.
- For some problem (in our case the image Histogram equalization) there is a maximum throughput the GPU can handle using streams, as one can see although the load is higher than 66,000 [request/sec] the throughput never exceed that bound.

2. Producer Consumer Queues:

2.1. The way we calculate the maximum number of thread blocks we can invoke is using the method we learned in class:

Getting thread block properties:

- Shared memory limit: each of thread blocks is using two arrays of 256 integers, 1 array of 256 bytes and 1-Byte (for request validator) total of: 2305 Bytes.
- Registers limit: using 32 registers.
- Threads per block: depending on user input.

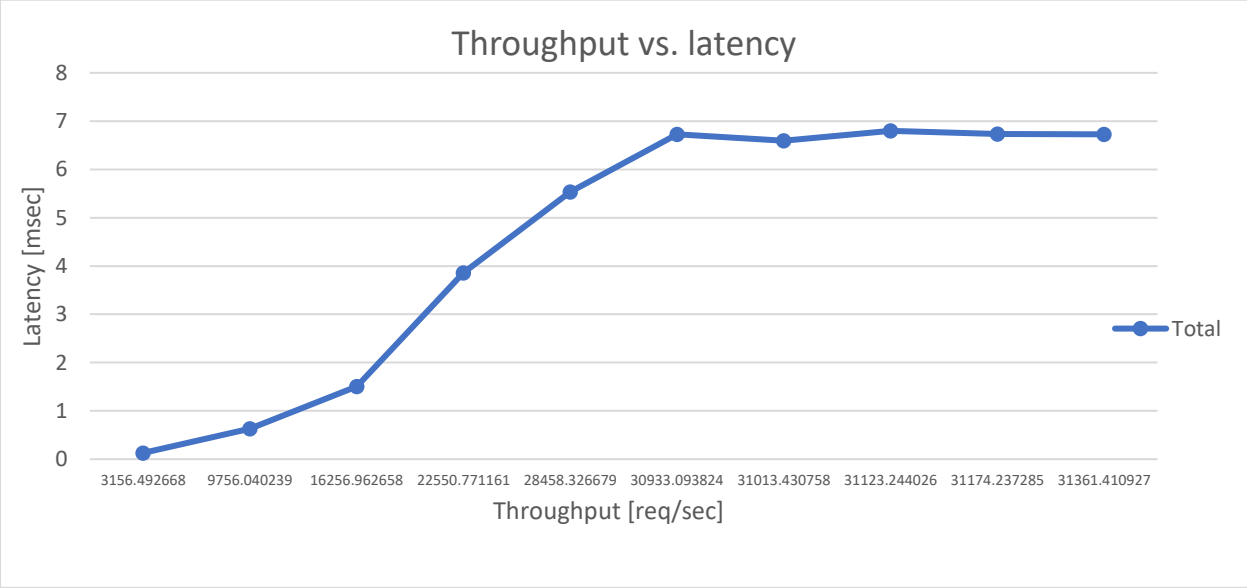
Getting Device properties:

- Shared memory limit: getting shared memory per SM.
- Registers limit: using getting registers per SM.
- Threads per block: max number of threads per SM.

After getting all the properties we calculate the number of thread blocks per SM by each limit (Hardware limitation / Block requirements)and the minimum was chosen.

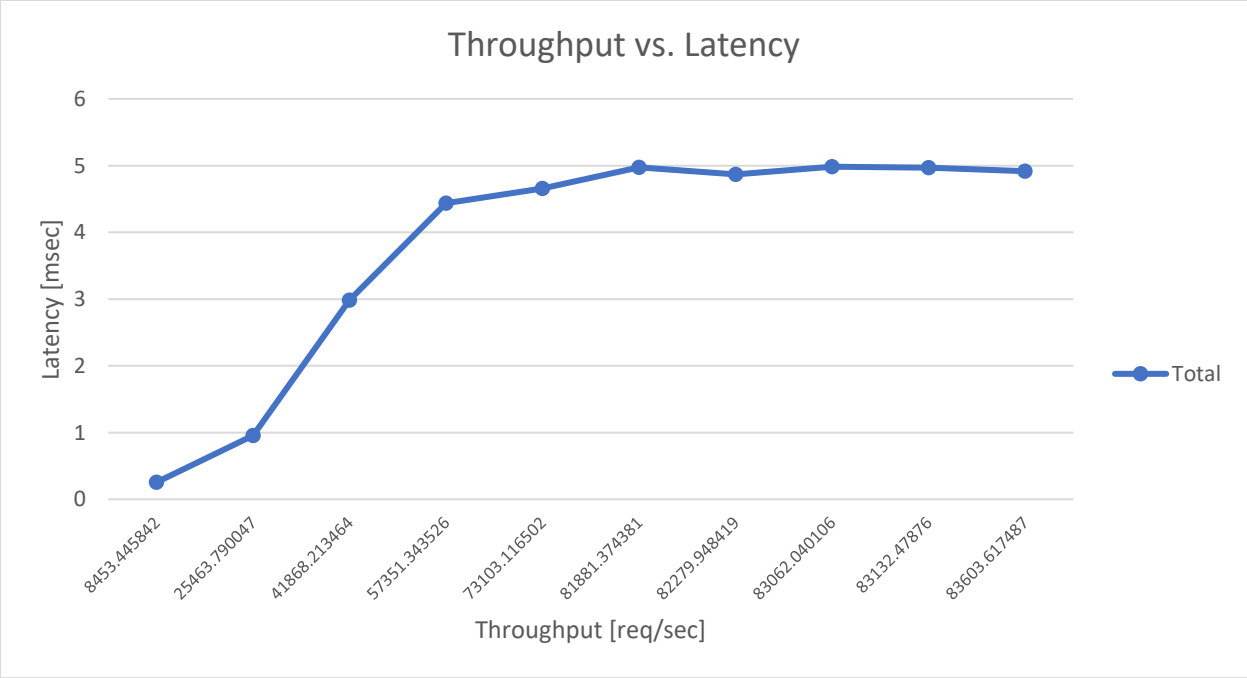
2.4. Average max load with 1024 thread per block is: 31,000 [request / sec].

Iteration #	Load [request/sec]	Throughput [request/sec]	Latency [mSec]
1	3100	3156.492668	0.129453
2	9644	9756.040239	0.628256
3	16188	16256.96266	1.507485
4	22733	22550.77116	3.858226
5	29277	28458.32668	5.532938
6	35822	31013.43076	6.596588
7	42366	30933.09382	6.725752
8	48911	31174.23729	6.734633
9	55455	31361.41093	6.730209
10	62000	31123.24403	6.799045



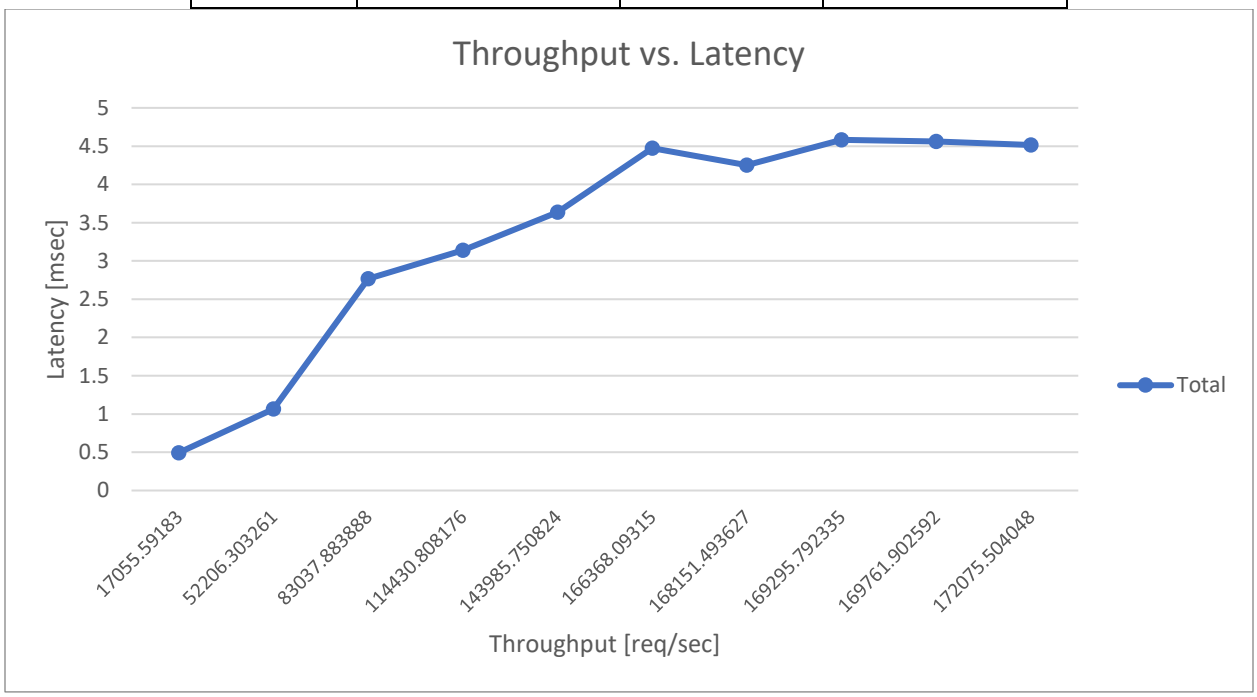
2.5. Average max load with 512 thread per block is: 83,000 [request / sec].

Iteration #	Load [request/sec]	Throughput [request/sec]	Latency [mSec]
1	8300	8453.445842	0.255284
2	25822	25463.79005	0.954448
3	43344	41868.21346	2.983908
4	60866	57351.34353	4.438241
5	78388	73103.1165	4.657392
6	95911	82279.94842	4.868808
7	113433	81881.37438	4.974646
8	130955	83603.61749	4.914363
9	148477	83132.47876	4.971361
10	166000	83062.04011	4.983748



2.6. Average max load with 512 thread per block is: 172,000 [request / sec].

Iteration #	Load [request/sec]	Throughput [request/sec]	Latency [mSec]
1	17200	17055.59183	0.494182
2	53511	52206.30326	1.064131
3	89822	83037.88389	2.768765
4	126133	114430.8082	3.137782
5	162444	143985.7508	3.637051
6	198755	168151.4936	4.250774
7	235066	166368.0932	4.474595
8	271377	169761.9026	4.560923
9	307688	172075.504	4.516301
10	344000	169295.7923	4.582194



2.7. In order to get better analysis of the results, we need to describe our request allocation policy:
For each request, we iterated through the threadblocks' queues in a id-based ascending order, looking for the first non-full queue, and allocating the request to this queue when found. This policy is rather naive, and in retrospective not optimal: for a low load rate, it leads to unbalanced allocation of requests between TBs, and thus higher-than-optimal latency.
With this analysis of our request allocation policy, we can explain some interesting phenomenon's in the graphs:

1. For higher load rates in low number of threads per block, the requests are being distributed more evenly between more thread block's queues, resulting in lower latency compering higher load rates in high number of threads per block.
2. The latency is getting higher when the throughput is rising, regardless number of threads per block because more parallel memory access.
3. As one can see the latency is much higher than streams implantation because of the penalty for CPU memory is more expensive.

2.8. It is better to locate the CPU-GPU queue in the GPU memory because we can reduce PCIe transaction in this implantation. The GPU is reading from this queue and for each read generates 2 PCIe transaction because read is non posted
On the other hand the CPU is writing to this queue without PCIe transaction because it locates in its local memory. When transfer the queue from CPU memory to GPU memory the CPU will write to it using PCIe transaction when each write generates only 1 transaction and the GPU will read from it without PCIe transaction because it is located in its local memory, therefore we can save half of the PCIe transactions. For the GPU-CPU queue is the opposite case.

2.9. We need to map the CPU-GPU queue which is now part of the GPU memory as a part of the CPU process' virtual memory. This can be done with MMIO, which will enable the process to write and/or read from this memory as a normal memory, delegating to the OS the responsibility of translating these memory operations to PCIe transactions.