PA4: Multiple GPU

Hannah Munoz

2/22/18

Introduction

In this project, we compare speedups from sequential algorithms to multi gpu algorithms using CUTThreads and the compare the results. The speedup, throughput, and timing of the parallel implementation is calculated.

Method

Both algorithms start by filling the array with random numbers. Then, the timer is started. The overarching algorithm of both algorithms is

$$resultMat = (Mat_1 + Mat_2) + (Mat_3 \; x \; Mat_4) + (Mat_5 + Mat_6)...$$

Sequential Algorithm

When a matrix needs to be added, each cell in the matrix is iterated through and added to the corresponding matrix cell of the result matrix. When multiplication is needed, each row, cell, and column of the array is iterated through as into the resulting matrix as seen below.

$$MatC\,[i * size + j] = \sum_{0}^{size} (MatA[i * size + k] * MatB[k * size + j])$$

Once all the matrices have been calculated, the timer is stopped.

Parallel Algorithm

Each matrix is allocated in unified memory. A dataset, DataStruct of information about the runtime, including blocks, threads, matrix size, and matrices themselves are constructed. Each DataStruct is given a GPU to run on. A set of CUTThreads the size of the amount of matrices are constructed and started. Some threads go to the add routine while the other go to the multiple routine. The main program waits for all threads to finish and the timer stops.

Results and Conclusion

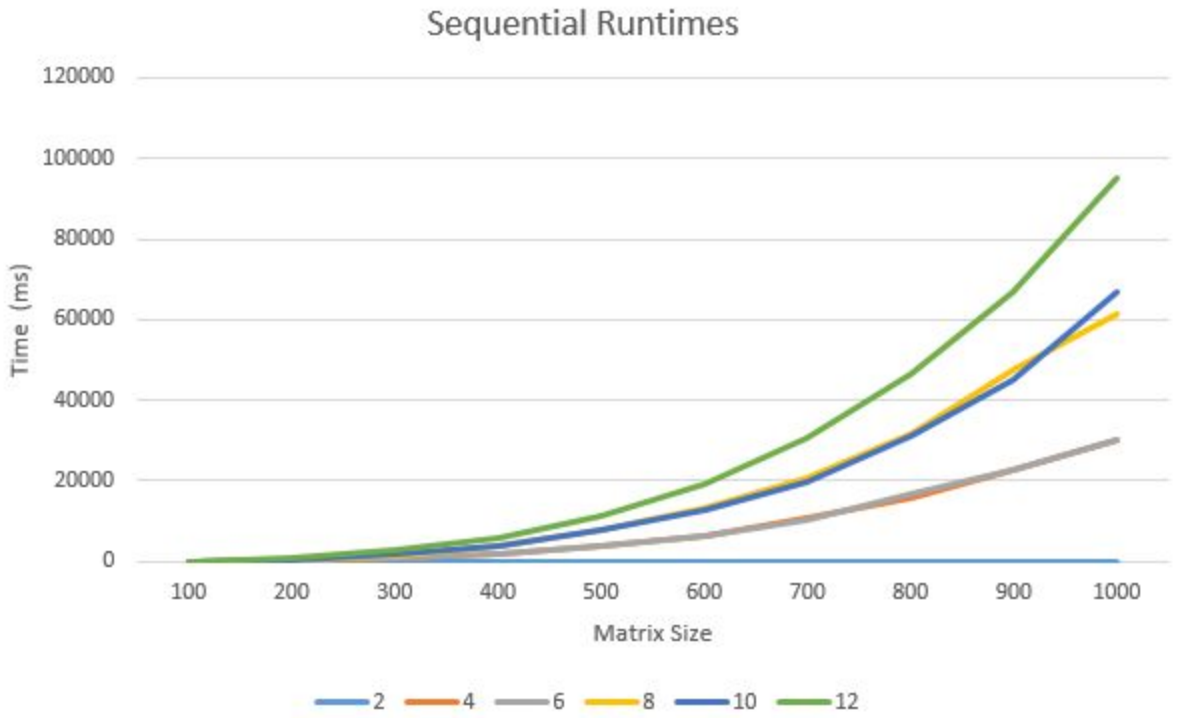
Sequential

The sequential algorithm's execution times in Table 1 and Fig. 1.

Table 1: The sequential execution times in ms

# of Matrices/Matrix Size	2	4	6	8	10	12
100	0.537792	28.0054	28.6308	68.2393	69.9427	84.369
200	1.94502	278.059	284.502	578.46	556.224	850.728
300	4.38019	934.737	903.655	1565.49	1898.83	2652.24
400	7.44669	1958.8	2010.78	3854.03	3718.02	5977.63
500	11.3952	3650.71	3749.14	7852.76	7647.9	1115
600	16.7005	6433.4	6340.8	13084.7	12656.5	19304.2
700	19.0808	10740.7	10509.5	20839.2	19932	30710.1
800	29.2385	15736.4	16688.1	31857.8	31050	46460.8
900	40.8808	22570.3	22700	47755.5	44857.1	66704.2
1000	35.9852	30190.5	30210.2	61510.2	66827	95349.5

Figure 1: A graph of the sequential execution times found in Table 1.



Parallel

The parallel algorithms execution times for 2 GPU are in Table 2 and Fig. 2. 10 threads were used in each and the appropriate amount of blocks was determined from the matrix size.

Table 2: The parallel execution times for 2 GPU

# of Matrices/Matrix Size	**2**	**4**	**6**	**8**	**10**	**12**
1000	4.58944	366.947	385.036	390.535	322.745	335.307
2000	11.795	371.419	374.192	366.682	371.001	401.777
3000	26.4886	348.914	371.268	351.521	370.153	366.825
4000	42.163	339.122	346.693	368.322	365.126	365.554
5000	57.8348	315.248	353.38	393.303	363.504	351.955
6000	101.387	394.911	346.769	360.061	352.569	368.123
7000	147.928	381.976	379.297	364.997	327.647	337.554
8000	182.46	350.628	366.133	374.584	363.995	370.268
9000	193.762	352.203	366.2	395.266	387.728	387.234
10000	266.59	372.516	381.387	366.586	373.448	393.795

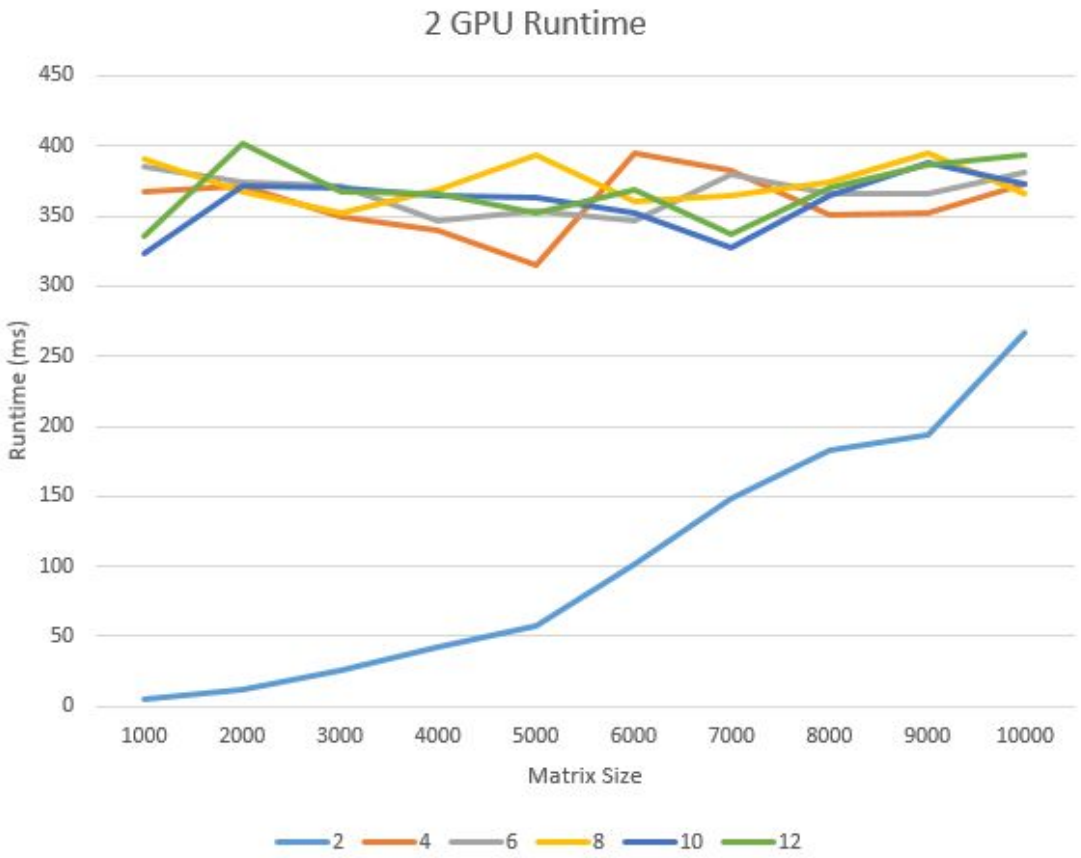


Figure 2: The execution times for 2 GPU over varying matrix sizes.

The parallel algorithms execution times for 3 GPU are in Table 3 and Fig. 3.

Table 3: The parallel execution times for 3 GPU

# of Matrices/Matrix Size	2	4	6	8	10	12
1000	3.47037	373.89	636.969	704.855	647.463	737.367
2000	17.3529	312.876	627.279	691.942	663.059	687.891
3000	24.6417	383.511	635.507	676.941	721.489	706.104
4000	42.8763	341.49	639.152	707.231	708.573	710.502
5000	57.9292	368.454	658.316	648.996	588.886	689.913
6000	82.996	357.503	648.33	666.118	667.18	676.718
7000	116.179	330.228	625.007	632.139	711.198	620.948
8000	151.901	375.65	683.279	667.309	613.991	741.671
9000	222.353	340.111	622.945	694.882	695.572	654.673
10000	241.266	371.838	718.319	668.916	698.268	755.595

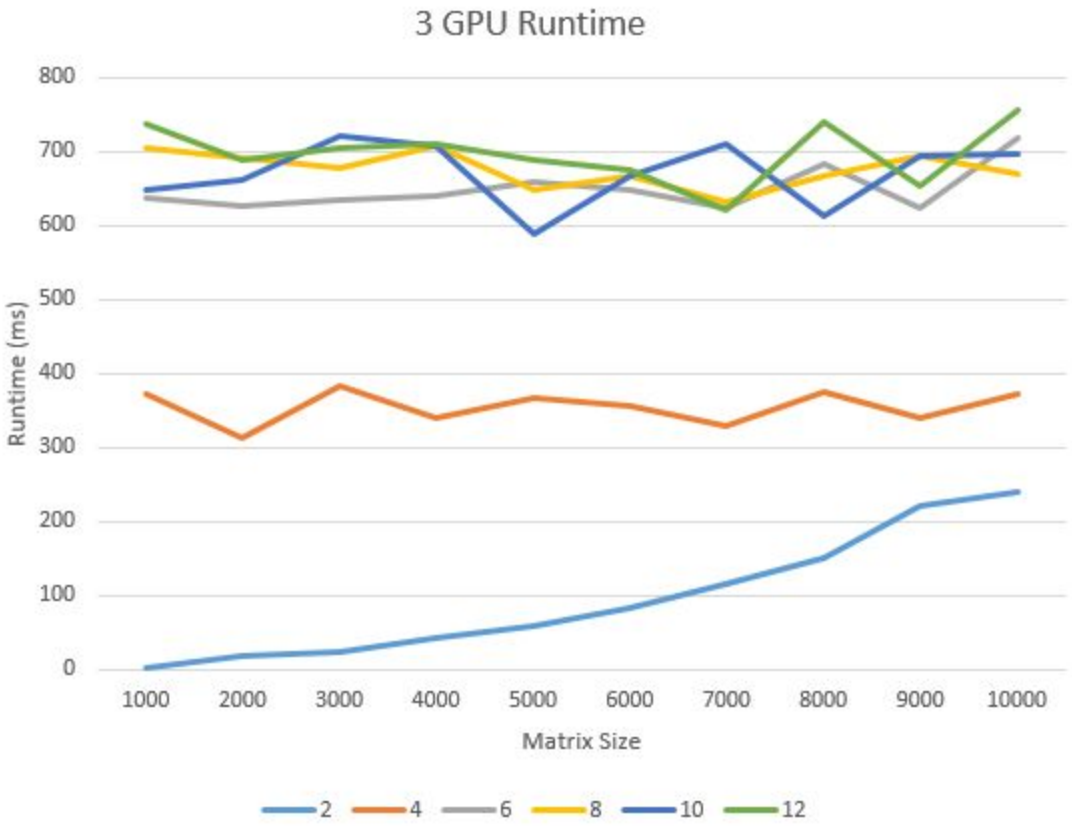


Figure 3: The execution times for 3 GPU over varying matrix sizes

The parallel algorithms execution times for 4 GPU are in Table 4 and Fig. 4.

Table 4: The parallel execution times for 4 GPU

# of Matrices/Matrix Size	2	4	6	8	10	12
1000	5.02176	384.6	653.713	922.105	1039.44	1043.87
2000	16.9959	329.43	671.343	1037.98	975.478	949.649
3000	24.2787	387.765	610.451	1023.31	950.086	977.161
4000	36.9939	352.962	655.246	979.124	1105.64	1017.95
5000	57.3397	370.104	651.137	973.464	994.956	956.601
6000	112.312	354.422	684.338	1058.31	930.787	1091.42
7000	113.902	357.952	645.848	998.509	1064.13	998.883
8000	155.571	378.825	618.236	1062.33	936.157	1001.65
9000	220.88	347.849	727.271	946.851	1043.36	1011.57
10000	235.262	364.533	653.075	1013.26	1060.38	1079.08

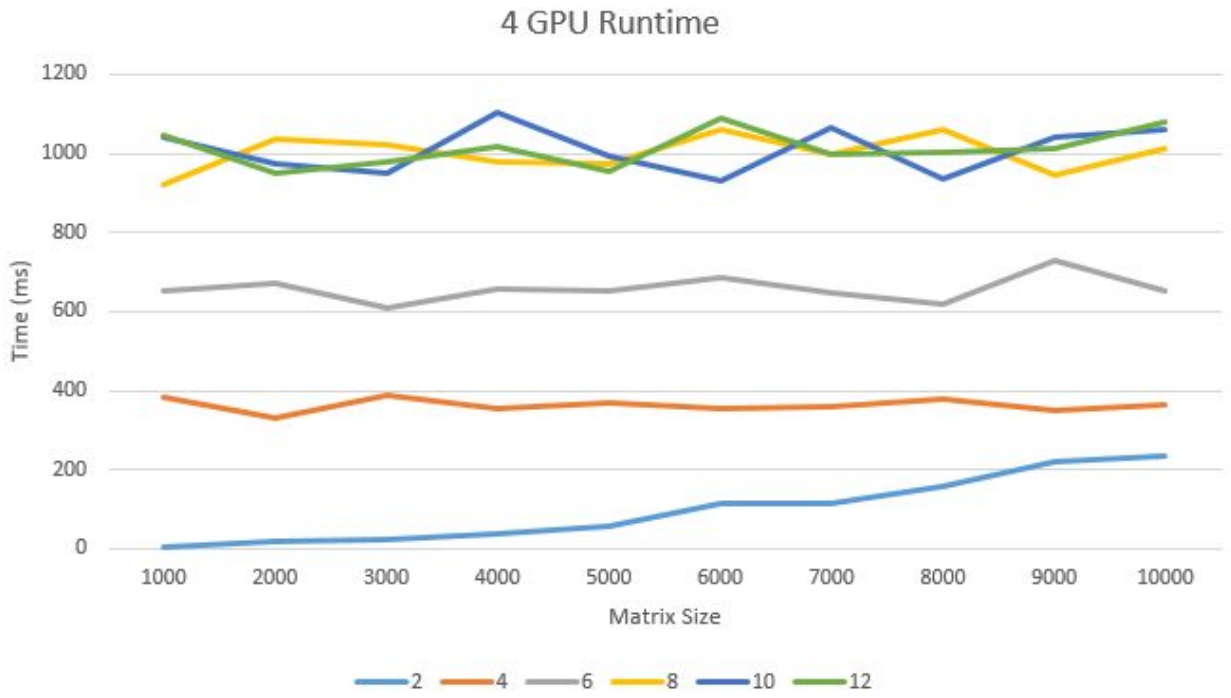
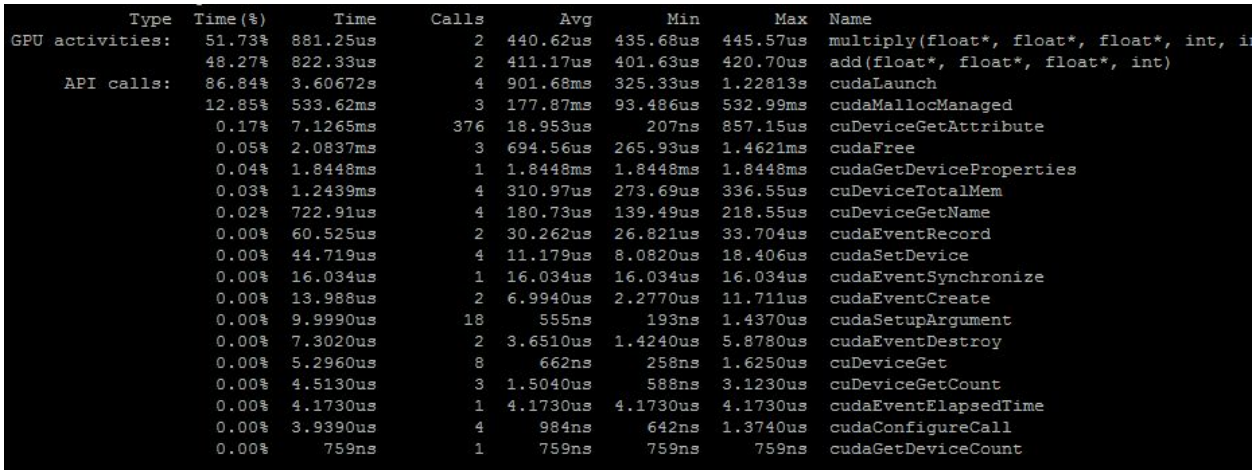


Figure 4: The execution times for 4 GPU over varying matrix sizes

The runtimes for the GPU increase as the amount of GPUs increases. I believe this is because of the overhead of unified memory across multiple GPUs. I could not find whether unified memory was optimized across multiple GPU. Multiple devices accessing the PCIe could cause an increase in runtime. Figures 5 and 6 show the runtime profilers of 2 GPU and 4 GPU side by side.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	50.72%	398.78us	2	199.39us	1.5680us	397.22us	add(float*, float*, float*, int)
	49.28%	387.40us	2	193.70us	3.0720us	384.32us	multiply(float*, float*, float*, int, i
API calls:	72.43%	1.27549s	4	318.87ms	395.24us	425.10ms	cudaLaunch
	27.20%	478.97ms	3	159.66ms	40.816us	478.67ms	cudaMallocManaged
	0.17%	3.0648ms	188	16.302us	206ns	719.29us	cuDeviceGetAttribute
	0.09%	1.5495ms	1	1.5495ms	1.5495ms	1.5495ms	cudaGetDeviceProperties
	0.05%	824.68us	3	274.89us	109.25us	573.45us	cudaFree
	0.03%	584.29us	2	292.14us	271.88us	312.41us	cuDeviceTotalMem
	0.02%	312.02us	2	156.01us	139.34us	172.68us	cuDeviceGetName
	0.00%	39.641us	2	19.820us	12.554us	27.087us	cudaEventRecord
	0.00%	31.289us	4	7.8220us	5.5830us	9.3310us	cudaSetDevice
	0.00%	17.561us	1	17.561us	17.561us	17.561us	cudaEventSynchronize
	0.00%	9.6260us	2	4.8130us	1.1350us	8.4910us	cudaEventCreate
	0.00%	8.1340us	18	451ns	204ns	917ns	cudaSetupArgument
	0.00%	5.7250us	2	2.8620us	1.1210us	4.6040us	cudaEventDestroy
	0.00%	3.9740us	1	3.9740us	3.9740us	3.9740us	cudaEventElapsedTime
	0.00%	3.2670us	3	1.0890us	322ns	2.3300us	cuDeviceGetCount
	0.00%	3.1890us	4	797ns	267ns	1.6560us	cuDeviceGet
	0.00%	3.0040us	4	751ns	519ns	970ns	cudaConfigureCall
	0.00%	388ns	1	388ns	388ns	388ns	cudaGetDeviceCount

Figure 5: The profile of 2 GPU on eight 10x10 matrices.



Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	51.73%	881.25us	2	440.62us	435.68us	445.57us	multiply(float*, float*, float*, int, i
	48.27%	822.33us	2	411.17us	401.63us	420.70us	add(float*, float*, float*, int)
API calls:	86.84%	3.60672s	4	901.68ms	325.33us	1.22813s	cudaLaunch
	12.85%	533.62ms	3	177.87ms	93.486us	532.99ms	cudaMallocManaged
	0.17%	7.1265ms	376	18.953us	207ns	857.15us	cuDeviceGetAttribute
	0.05%	2.0837ms	3	694.56us	265.93us	1.4621ms	cudaFree
	0.04%	1.8448ms	1	1.8448ms	1.8448ms	1.8448ms	cudaGetDeviceProperties
	0.03%	1.2439ms	4	310.97us	273.69us	336.55us	cuDeviceTotalMem
	0.02%	722.91us	4	180.73us	139.49us	218.55us	cuDeviceGetName
	0.00%	60.525us	2	30.262us	26.821us	33.704us	cudaEventRecord
	0.00%	44.719us	4	11.179us	8.0820us	18.406us	cudaSetDevice
	0.00%	16.034us	1	16.034us	16.034us	16.034us	cudaEventSynchronize
	0.00%	13.988us	2	6.9940us	2.2770us	11.711us	cudaEventCreate
	0.00%	9.9990us	18	555ns	193ns	1.4370us	cudaSetupArgument
	0.00%	7.3020us	2	3.6510us	1.4240us	5.8780us	cudaEventDestroy
	0.00%	5.2960us	8	662ns	258ns	1.6250us	cuDeviceGet
	0.00%	4.5130us	3	1.5040us	588ns	3.1230us	cuDeviceGetCount
	0.00%	4.1730us	1	4.1730us	4.1730us	4.1730us	cudaEventElapsedTime
	0.00%	3.9390us	4	984ns	642ns	1.3740us	cudaConfigureCall
	0.00%	759ns	1	759ns	759ns	759ns	cudaGetDeviceCount

Figure 6: The profile of 4 GPU on eight 10x10 matrices.

Comparing Figures 5 and 6 we see there's a dramatic increase in runtime only in the GPU activities. However, because both processes should be doing the exact same amount of work within the addition and multiplication routines, I believe that the increased runtime is due to unified memory overhead.

To calculate speed up, the sequential timings were calculated using the data found in Table 1. The results can be seen in Table 5 and Figure 7.

Table 5: The parallel execution times for 4 GPU

# of Matrices/Matrix Size	**2**	**4**	**6**	**8**	**10**	**12**
1000	35.9852	30190.5	30210.2	61510.2	66827	95349.5
2000	162.586	31212.7	210082.49	453669.16	548920.2	613505
3000	410.953	288310	661872.89	1482237.36	2096034	2251089
4000	720.605	265517.34	1505063.3	3445605.56	5290748	5575273
5000	1155.31	872090.14	2859653.7	6643773.76	10733062	11186057
6000	1495.22	2034662.9	4845644.1	11376741.96	19022976	19683441
7000	1995.95	3933235.7	7583034.5	17944510.16	30760490	31667425
8000	2500.47	6747808.5	11191825	26647078.36	46545604	47738009
9000	3220.84	10658381	15792015	37784446.56	66978318	68495193
10000	4226.22	15844954	21503606	51656614.76	92658632	94538977

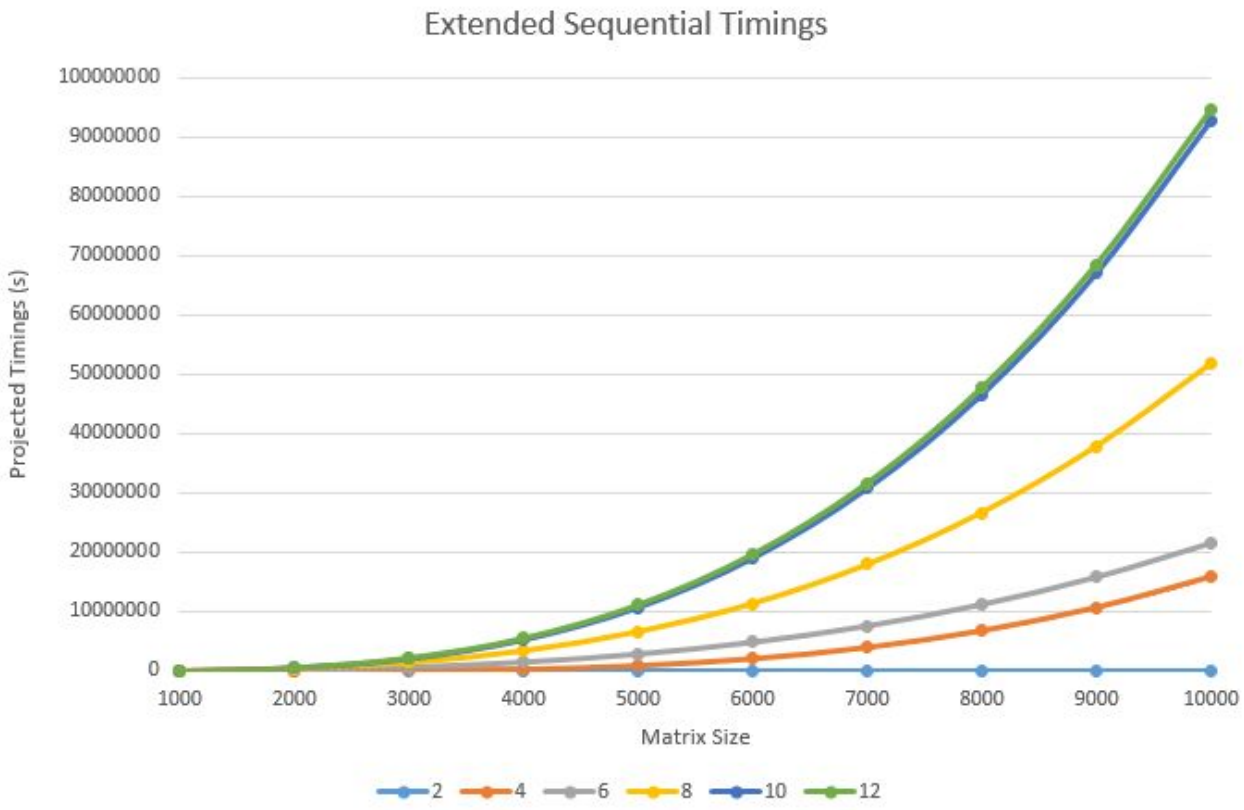
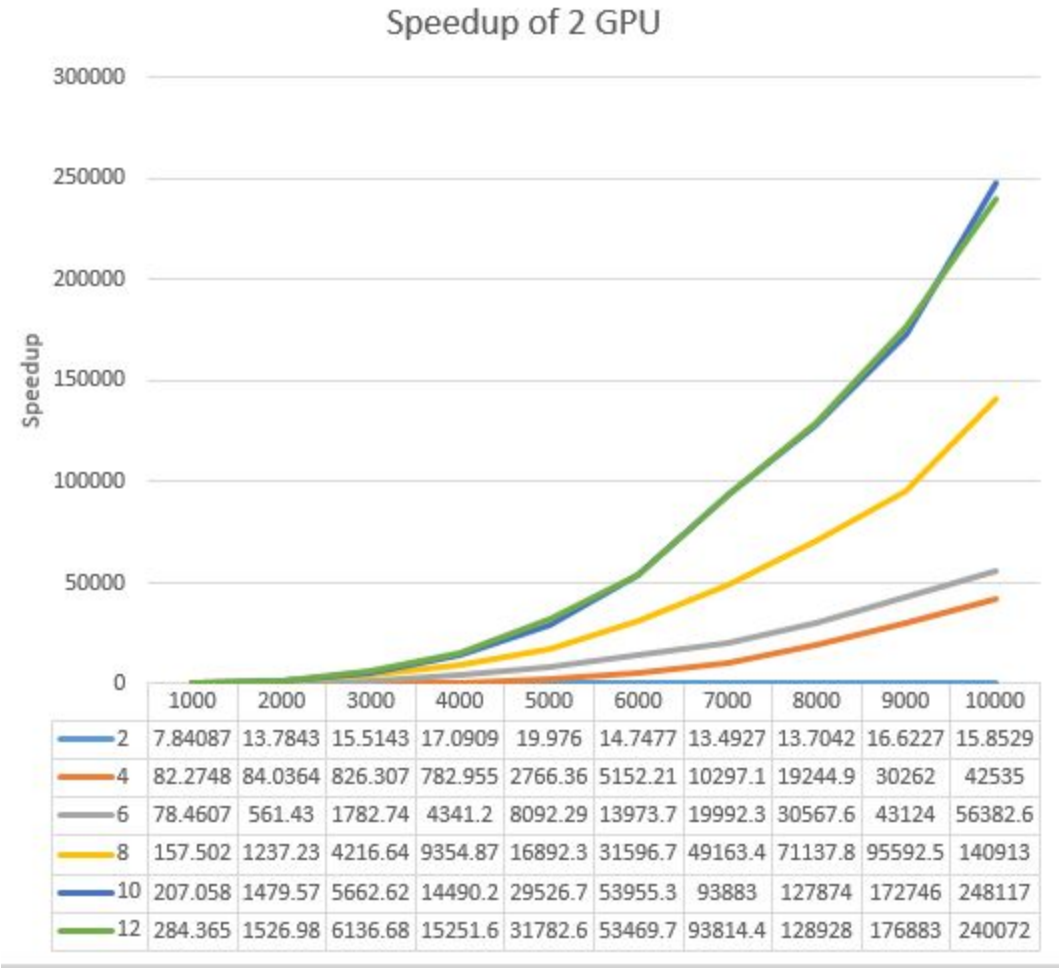


Figure 7: The projected sequential timings used to calculate speedup.

The equation used to calculate speedup in Figures 7,8 and 9 is.

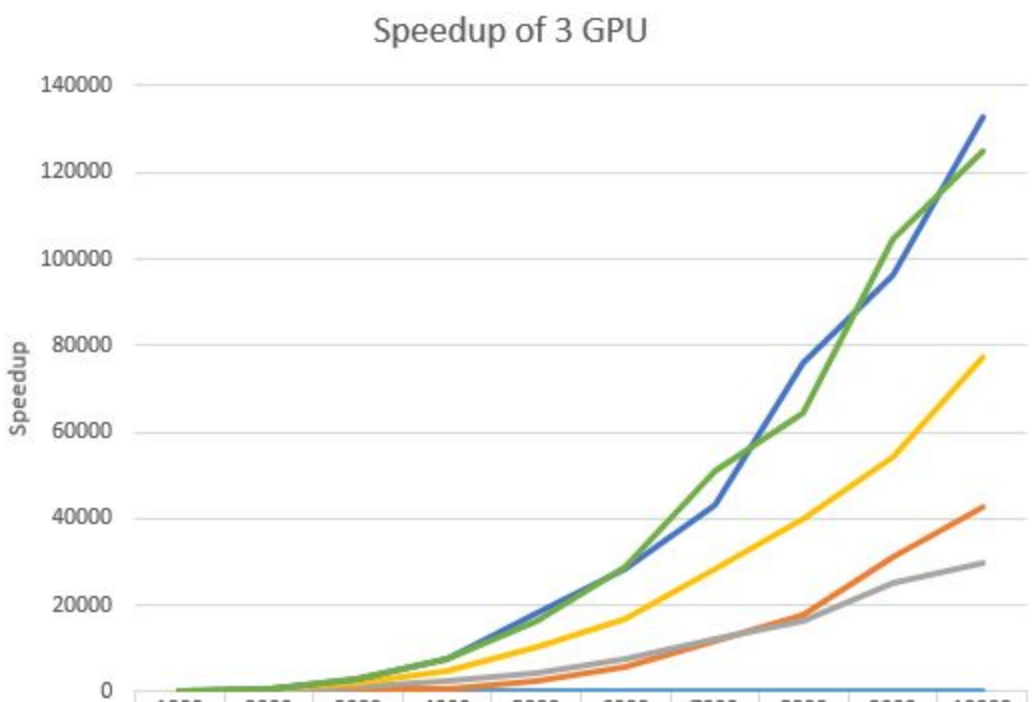
$$speedup = \frac{sequential\ time}{GPU\ time}$$

Figure 7: The speedup factor when using 2 GPU.



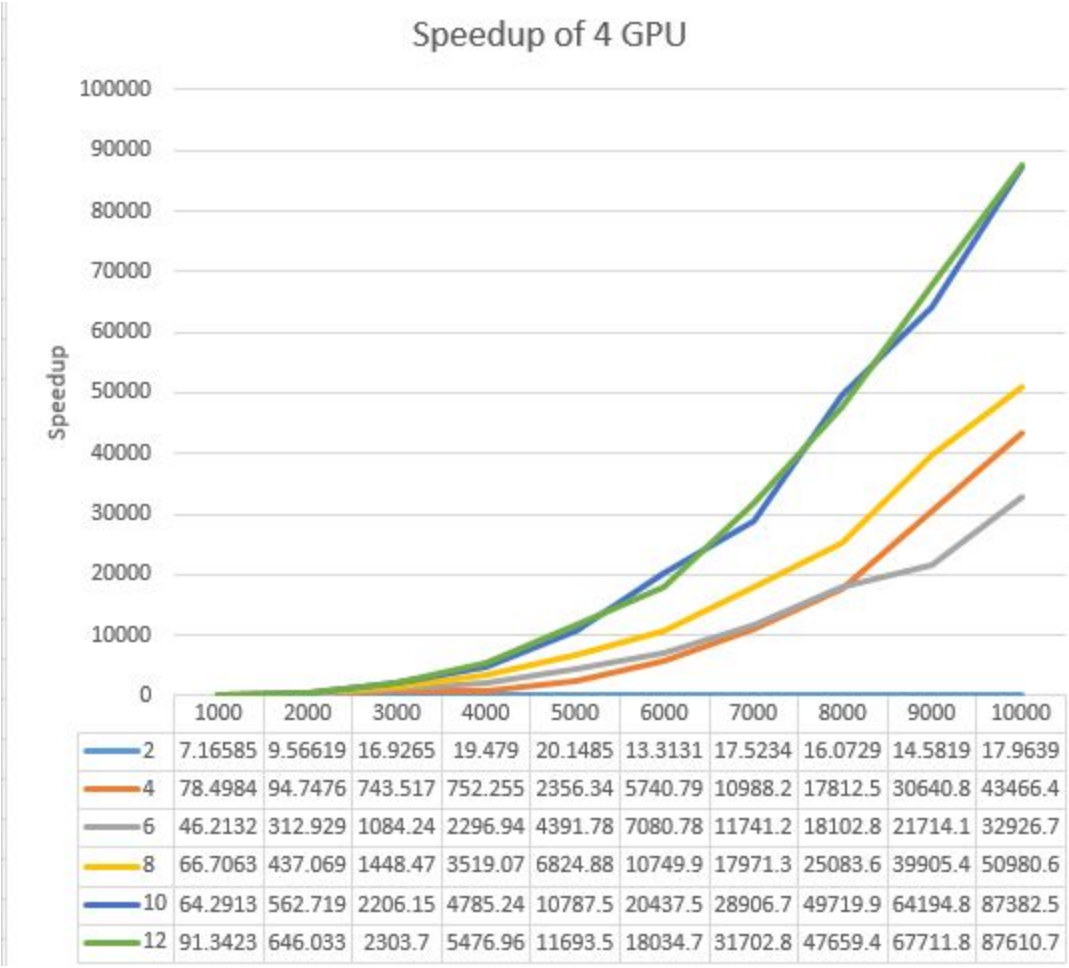
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
2	7.84087	13.7843	15.5143	17.0909	19.976	14.7477	13.4927	13.7042	16.6227	15.8529
4	82.2748	84.0364	826.307	782.955	2766.36	5152.21	10297.1	19244.9	30262	42535
6	78.4607	561.43	1782.74	4341.2	8092.29	13973.7	19992.3	30567.6	43124	56382.6
8	157.502	1237.23	4216.64	9354.87	16892.3	31596.7	49163.4	71137.8	95592.5	140913
10	207.058	1479.57	5662.62	14490.2	29526.7	53955.3	93883	127874	172746	248117
12	284.365	1526.98	6136.68	15251.6	31782.6	53469.7	93814.4	128928	176883	240072

Figure 8: The speedup factor when using 3 GPU.



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
2	10.3693	9.36938	16.6771	16.8066	19.9435	18.0156	17.18	16.4612	14.4853	17.5168
4	80.747	99.7606	751.765	777.526	2366.89	5691.32	11910.7	17963	31337.9	42612.5
6	47.4281	334.911	1041.49	2354.78	4343.89	7474.04	12132.7	16379.6	25350.6	29936
8	87.2665	655.646	2189.61	4871.97	10237	17079.2	28387	39932.1	54375.3	77224.4
10	103.214	827.86	2905.15	7466.77	18226	28512.5	43251.7	75808.3	96292.4	132698
12	129.311	891.864	3188.04	7846.95	16213.7	29086.6	50998.5	64365.5	104625	125119

Figure 9: The speedup factor when using 4 GPU.

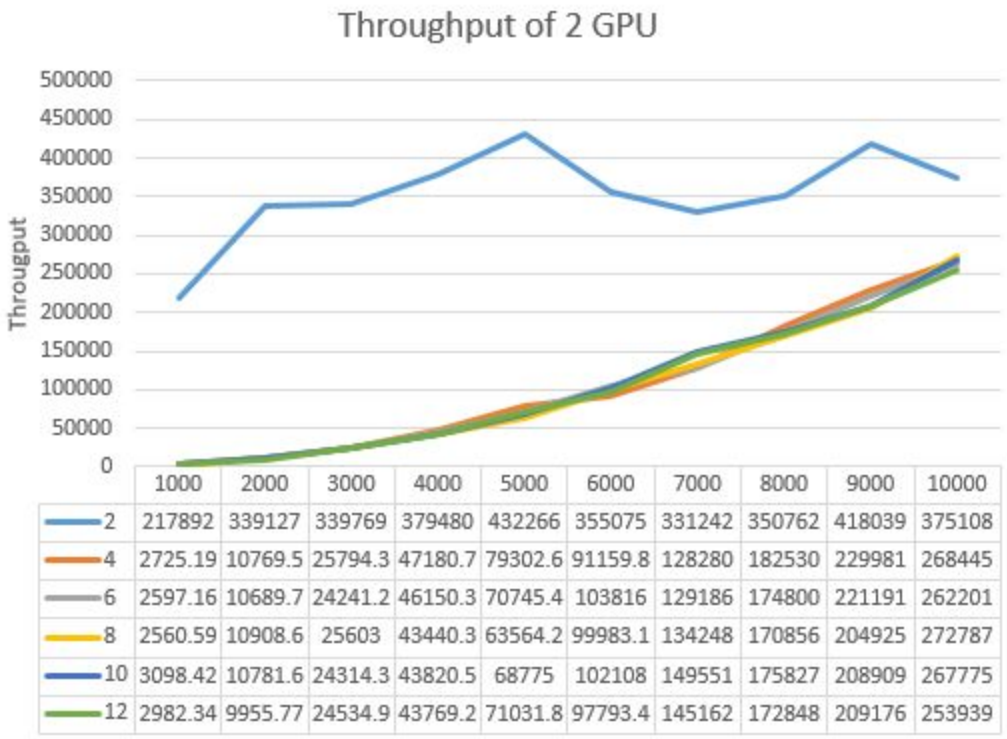


	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
2	7.16585	9.56619	16.9265	19.479	20.1485	13.3131	17.5234	16.0729	14.5819	17.9639
4	78.4984	94.7476	743.517	752.255	2356.34	5740.79	10988.2	17812.5	30640.8	43466.4
6	46.2132	312.929	1084.24	2296.94	4391.78	7080.78	11741.2	18102.8	21714.1	32926.7
8	66.7063	437.069	1448.47	3519.07	6824.88	10749.9	17971.3	25083.6	39905.4	50980.6
10	64.2913	562.719	2206.15	4785.24	10787.5	20437.5	28906.7	49719.9	64194.8	87382.5
12	91.3423	646.033	2303.7	5476.96	11693.5	18034.7	31702.8	47659.4	67711.8	87610.7

In Figures 7 and 9 we see exactly what is expected, as the matrices get larger, speedup becomes more larger. It's better to do large calculations on the GPU. Figure 8 has 10 and 12 matrices intertwining, however this could be due to noise in the original measurements, as they should have been taking multiple times and averaged.

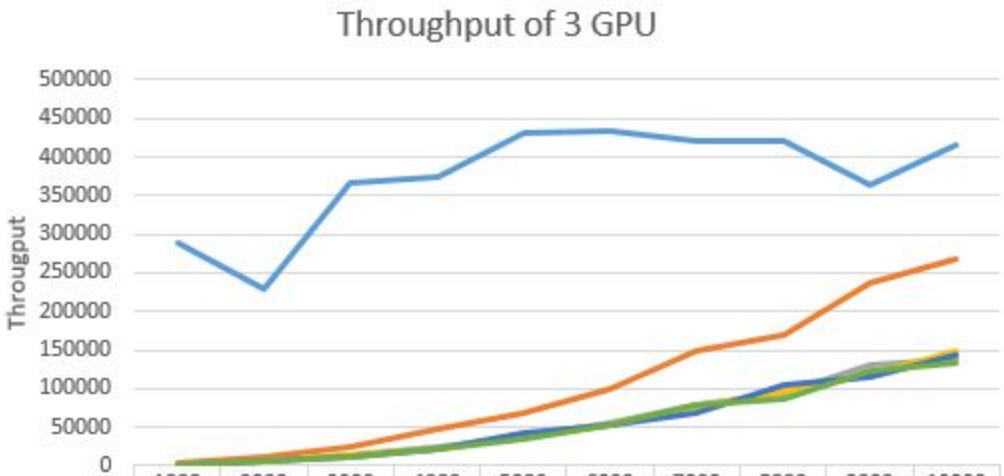
The equation used to calculate throughput in Figure 10, 11, and 12 is

$$throughput = \frac{Matrix\ Size\ (N\ x\ N)}{GPU\ execution\ time\ (s)}$$



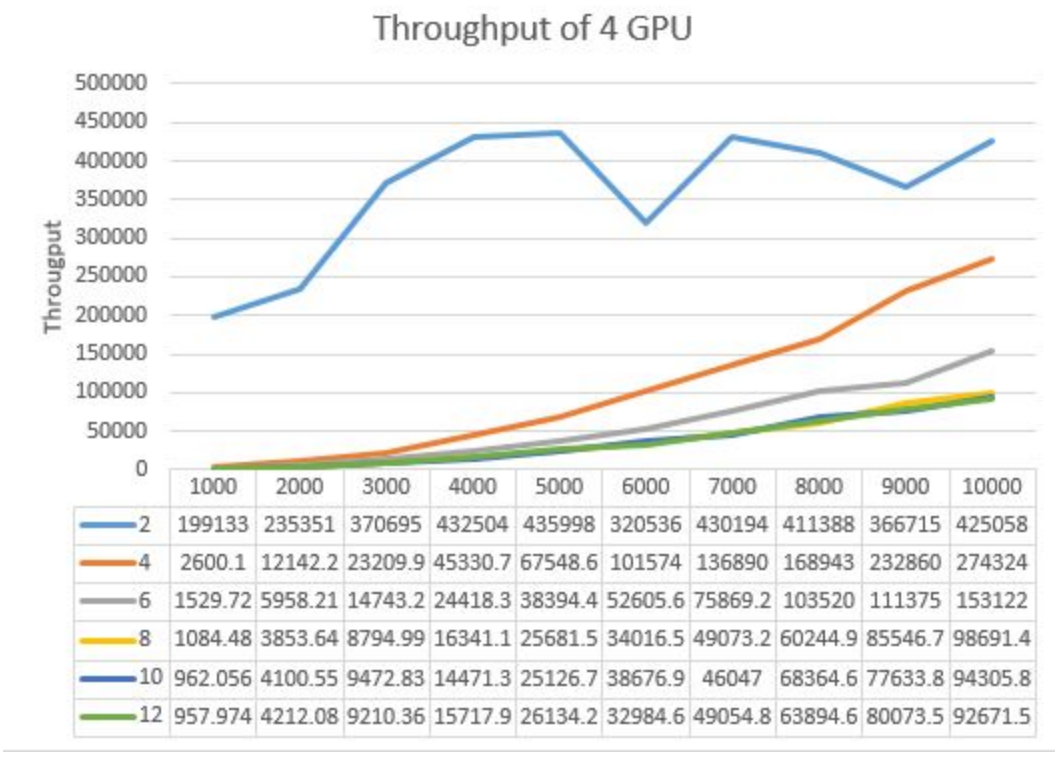
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
2	217892	339127	339769	379480	432266	355075	331242	350762	418039	375108
4	2725.19	10769.5	25794.3	47180.7	79302.6	91159.8	128280	182530	229981	268445
6	2597.16	10689.7	24241.2	46150.3	70745.4	103816	129186	174800	221191	262201
8	2560.59	10908.6	25603	43440.3	63564.2	99983.1	134248	170856	204925	272787
10	3098.42	10781.6	24314.3	43820.5	68775	102108	149551	175827	208909	267775
12	2982.34	9955.77	24534.9	43769.2	71031.8	97793.4	145162	172848	209176	253939

Figure 10: The throughput of 2 GPU.



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
2	288154	230509	365235	373167	431561	433756	421763	421327	364286	414480
4	2674.58	12784.6	23467.4	46853.5	67851.1	100698	148382	170371	238158	268934
6	1569.94	6376.75	14161.9	25033.2	37975.7	55527.3	78399.1	93666	130028	139214
8	1418.73	5780.83	13295.1	22623.4	38521	54044.5	77514.6	95907.6	116567	149496
10	1544.49	6032.65	12474.2	22580.6	42453	53958.5	68897.8	104236	116451	143211
12	1356.18	5814.87	12746	22519.3	36236.5	53197.9	78911.6	86291.6	123726	132346

Figure 11: The throughput of 3 GPU.



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
2	199133	235351	370695	432504	435998	320536	430194	411388	366715	425058
4	2600.1	12142.2	23209.9	45330.7	67548.6	101574	136890	168943	232860	274324
6	1529.72	5958.21	14743.2	24418.3	38394.4	52605.6	75869.2	103520	111375	153122
8	1084.48	3853.64	8794.99	16341.1	25681.5	34016.5	49073.2	60244.9	85546.7	98691.4
10	962.056	4100.55	9472.83	14471.3	25126.7	38676.9	46047	68364.6	77633.8	94305.8
12	957.974	4212.08	9210.36	15717.9	26134.2	32984.6	49054.8	63894.6	80073.5	92671.5

Figure 12: The throughput of 4 GPU.

The less matrices that need processing, the higher the throughput is. This makes sense, as more matrices will have their threads wait for their assigned GPU to open, essentially making them semi-sequential. The throughput begins to decrease as the number of GPUs increase, once again because of the overhead caused by unified memory. The 2 matrices have so much more throughput than the other matrix lines because there is no multiplication being done on 2 matrices. Multiplication takes far more time than addition does.