CUDA 2D Matrix Addition
Benjamin A. Slack
CS5260
Assignment #4
12.02.2017

Description:

In this experiment, I implemented a CUDA routine for adding arbitrary M x N matrices. I implemented a basic matrix struct type, containing dimensions and an array for contents. I then devised procedures for allocating this struct on the host machine and the CUDA device, as well as copying them back and forth. Kernel functions initialized the arrays on the CUDA device and performed the additions. The results are then retrieved by the host. The algorithm I developed maintains the two dimensional indexing of matrix elements as much as possible, mapping block and thread id's to the m and n coordinates of the matrices. I drove this program via BASH script and the TORQUE queue, setting command line parameters for the program that set problem size, grid and thread per block specifications. Each configuration ran five times to supply data for averaging. I used an average of the median and first and third quartile to lessen the effect of outliers. I tested with the following configurations:

N x N matrices, with n = [512,1024, 2048, 4096, 8192]

D x D grids, with d = [64, 128, 256, 512, 1024]

Threads per block = [1, 4, 16, 32, 64]

All tests ran on the THOR cluster, using the Tesla M2090 node. Memory limitations didn't allow for higher N values consistently. I left a re-write to allow for segmentation of the matrices into submatrices (and thereby higher values on N) for a later experiment.

Results:

Note, all timing results show the average cpu timing in multiple rows. This is because these values tie directly to my spreadsheet which generates the charts.

512 x 512 Matrix: Time (seconds) vs Grid Size (DxD) x Number of Threads

512	64	128	256	512	1024	cpu
1	0.002624	0.002700	0.002970	0.003770	0.007020	0.007816
4	0.001461	0.001525	0.001719	0.002530	0.005412	0.007816
16	0.001139	0.001185	0.001414	0.002123	0.004820	0.007816

32	0.001076	0.001139	0.001354	0.002078	0.004766	0.007816
64	0.001041	0.001113	0.001293	0.001988	0.004610	0.007816

1024 x 1024 Matrix: Time (seconds) vs Grid Size (DxD) x Number of Threads

1024	64	128	256	512	1024	cpu
1	0.009065	0.009213	0.009573	0.010637	0.013873	0.033059
4	0.004522	0.004616	0.004879	0.005694	0.008946	0.033059
16	0.003349	0.003208	0.003609	0.004423	0.007321	0.033059
32	0.003065	0.003143	0.003387	0.004210	0.007082	0.033059
64	0.002961	0.003013	0.003239	0.003986	0.006745	0.033059

2048 x 2048 Matrix: Time (seconds) vs Grid Size (DxD) x Number of Threads

2048	64	128	256	512	1024	cpu
1	0.033356	0.033813	0.034438	0.035754	0.040159	0.431084
4	0.015578	0.015557	0.015758	0.017186	0.020073	0.431084
16	0.010901	0.010764	0.011220	0.011960	0.015238	0.431084
32	0.009854	0.009724	0.010177	0.011167	0.014227	0.431084
64	0.009424	0.009502	0.009721	0.010587	0.013163	0.431084

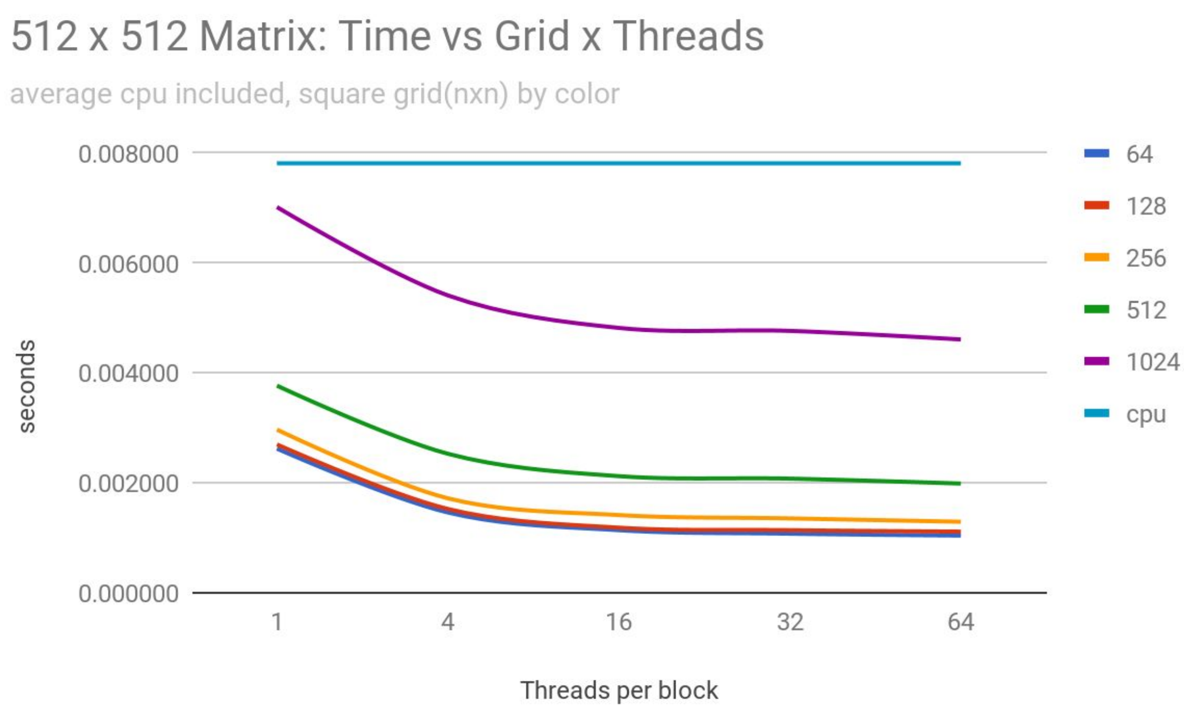
4096 x 4096 Matrix: Time (seconds) vs Grid Size (DxD) x Number of Threads

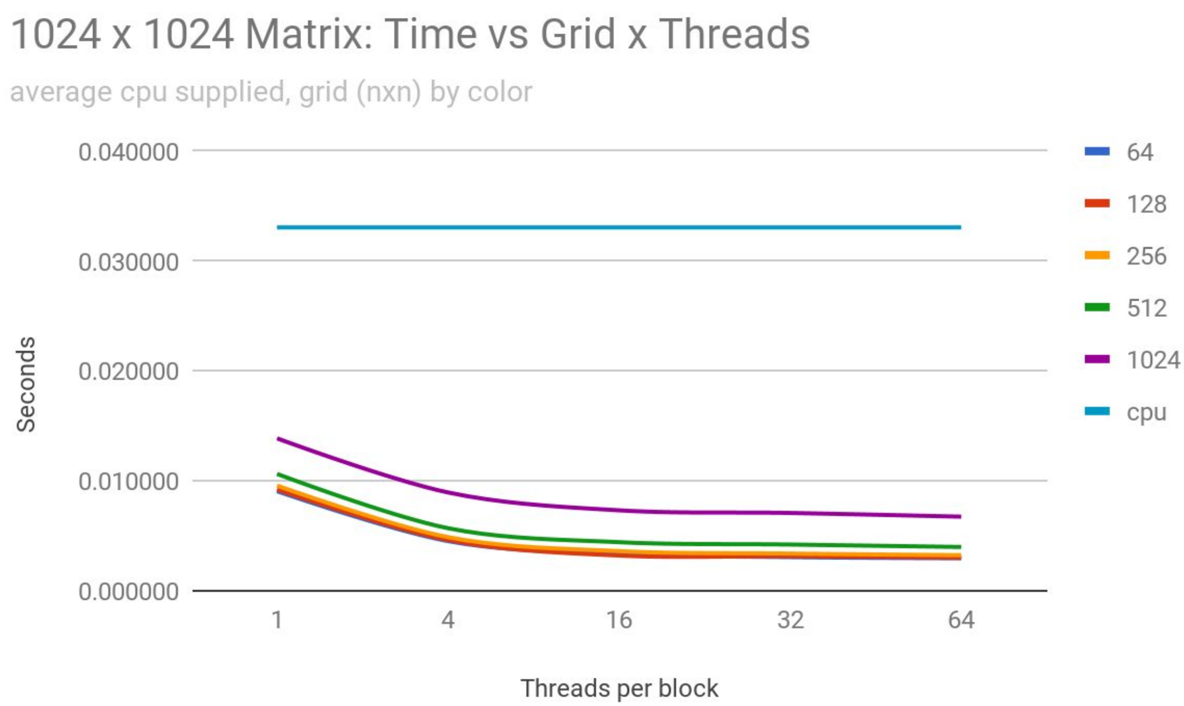
4096	64	128	256	512	1024	cpu
1	0.131139	0.131243	0.134809	0.134932	0.142017	1.863788
4	0.059215	0.059558	0.059880	0.061486	0.065525	1.863788
16	0.040552	0.040661	0.041023	0.041976	0.044949	1.863788
32	0.036487	0.036595	0.036849	0.038535	0.041850	1.863788
64	0.034818	0.035095	0.036025	0.036018	0.039465	1.863788

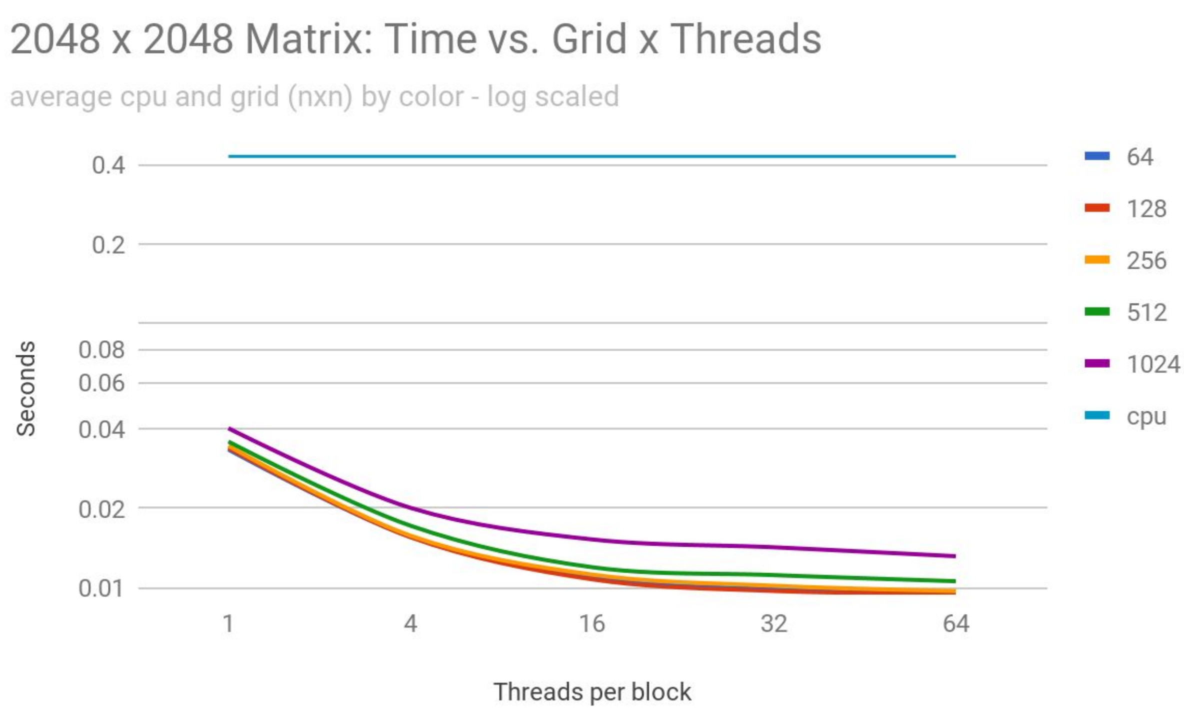
8192 x 8192 Matrix: Time (seconds) vs Grid Size (DxD) x Number of Threads

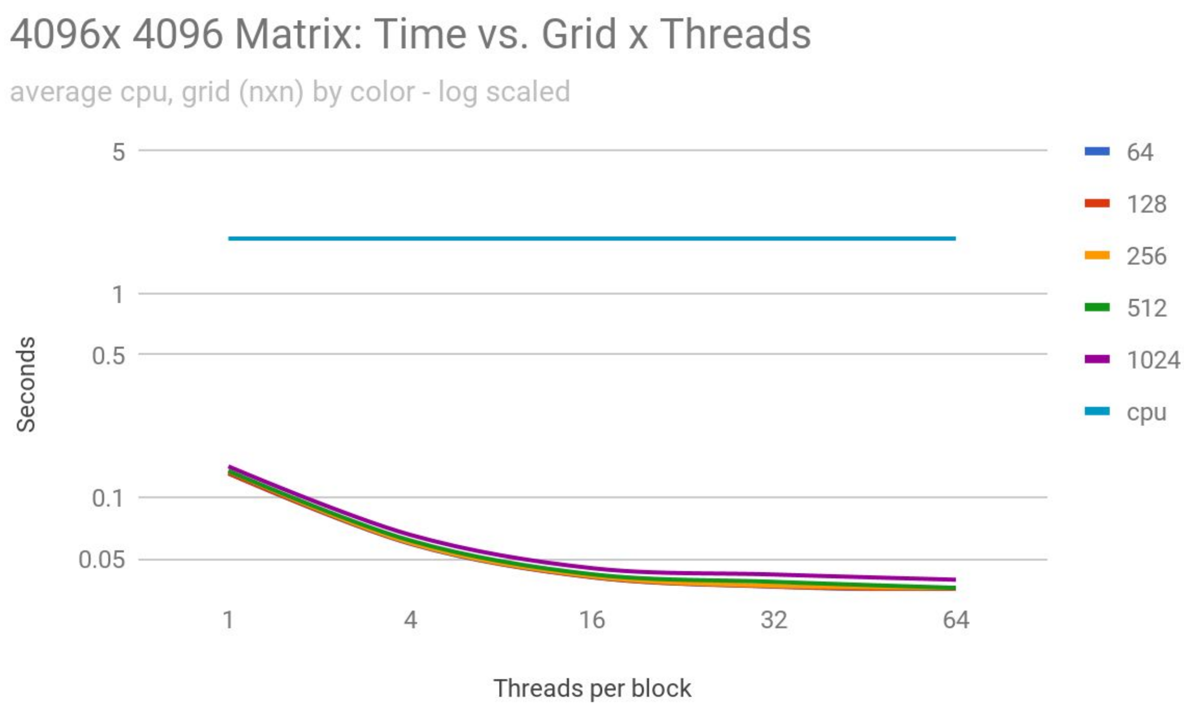
8192	64	128	256	512	1024	cpu
1	0.520756	0.530036	0.524187	0.526051	0.545551	7.596068
4	0.233351	0.233688	0.234732	0.237564	0.246609	7.596068
16	0.162900	0.159950	0.160413	0.161487	0.165071	7.596068

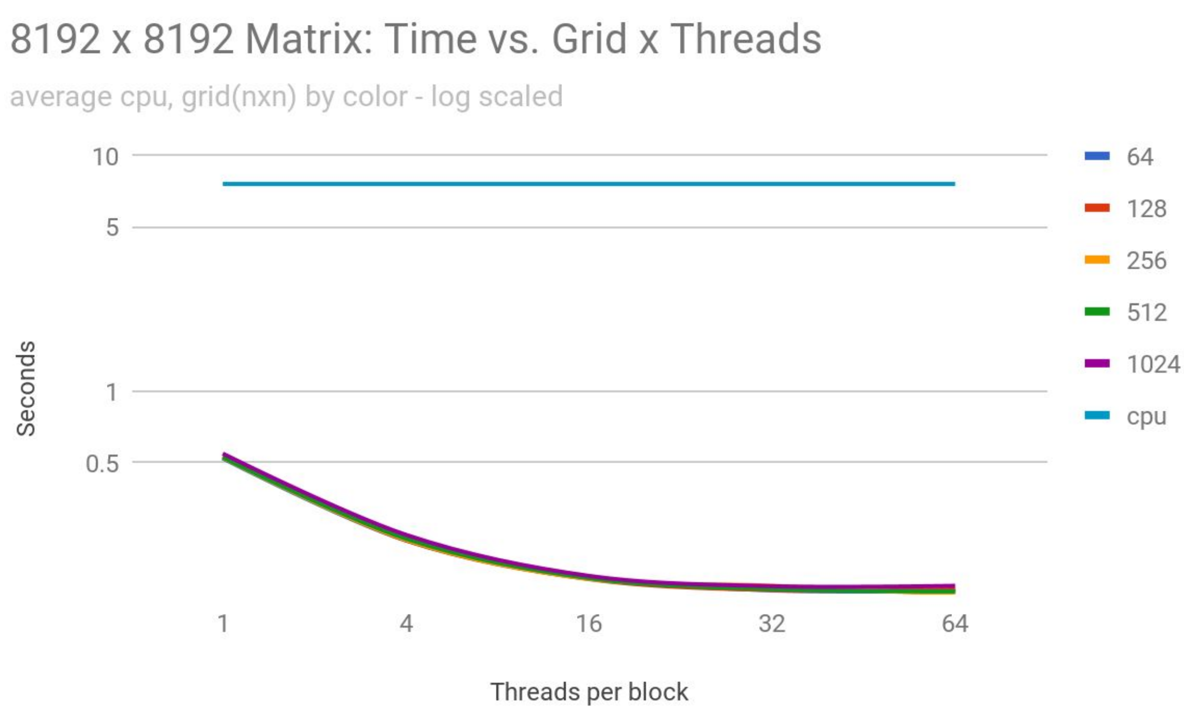
32	0.143327	0.143716	0.149889	0.145071	0.149283	7.596068
64	0.142609	0.144708	0.137527	0.141820	0.149818	7.596068




1024 x 1024 Matrix: Time vs Grid x Threads
average cpu supplied, grid (nxn) by color
0.040000
0.030000
0.020000
0.010000
0.000000
Seconds
1
4
16
32
64
Threads per block
64
128
256
512
1024
cpu


2048 x 2048 Matrix: Time vs. Grid x Threads
average cpu and grid (nxn) by color - log scaled
0.4
0.2
0.08
0.06
0.04
0.02
0.01
Seconds
1
4
16
32
64
Threads per block
64
128
256
512
1024
cpu





512 x 512 Matrix: Speedup vs Grid Size (DxD) x Number of Threads

512	64	128	256	512	1024

1	2.973827976	2.889616002	2.691211135	2.065169334	1.11097393
4	5.313412409	5.10861014	4.547886778	3.089998682	1.44250077
16	6.870392042	6.575365579	5.521800613	3.681846734	1.620081599
32	7.266501394	6.858104154	5.783111768	3.767367239	1.642212742
64	7.483034571	7.023952096	6.03120165	3.932092555	1.695611308

1024 x 1024 Matrix: Speedup vs Grid Size (DxD) x Number of Threads

1024	64	128	256	512	1024
1	3.752454495	3.611006585	3.495299443	3.116448873	2.399072539
4	7.220567637	7.200909944	6.683495013	5.697792869	3.687022616
16	9.762990245	10.18341474	9.354424534	7.784368405	4.609479579
32	10.84250598	10.30943796	9.899724464	8.053290047	4.816192045
64	10.79365079	10.51078659	10.15877753	8.154290015	4.725093892

2048 x 2048 Matrix: Speedup vs Grid Size (DxD) x Number of Threads

2048	64	128	256	512	1024
1	12.87738215	12.87295938	12.47464042	12.02501329	10.70960191
4	27.83831554	27.51230957	27.16744934	25.08443082	21.45333776
16	39.96388821	39.82092094	38.4241659	35.96811505	28.17331292
32	43.42587018	45.24208975	42.18747544	38.5576085	29.96874561
64	46.13277685	46.22231188	44.25175736	40.62222922	32.50771101

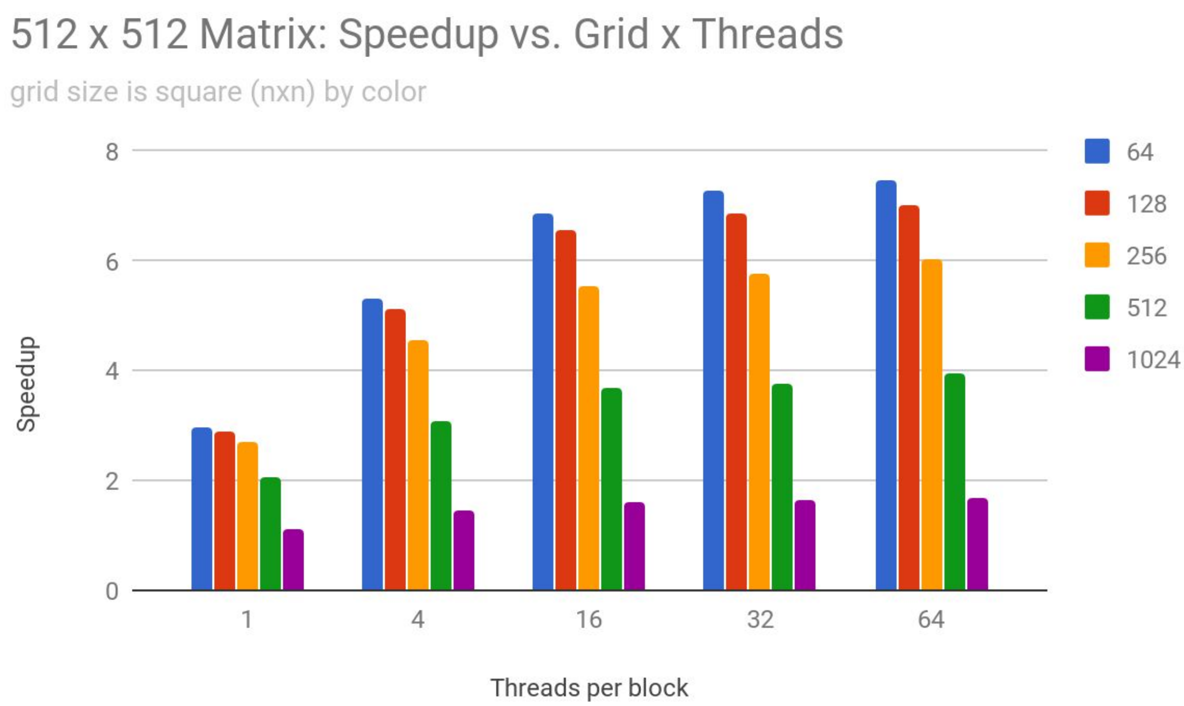
4096 x 4096 Matrix: Speedup vs Grid Size (DxD) x Number of Threads

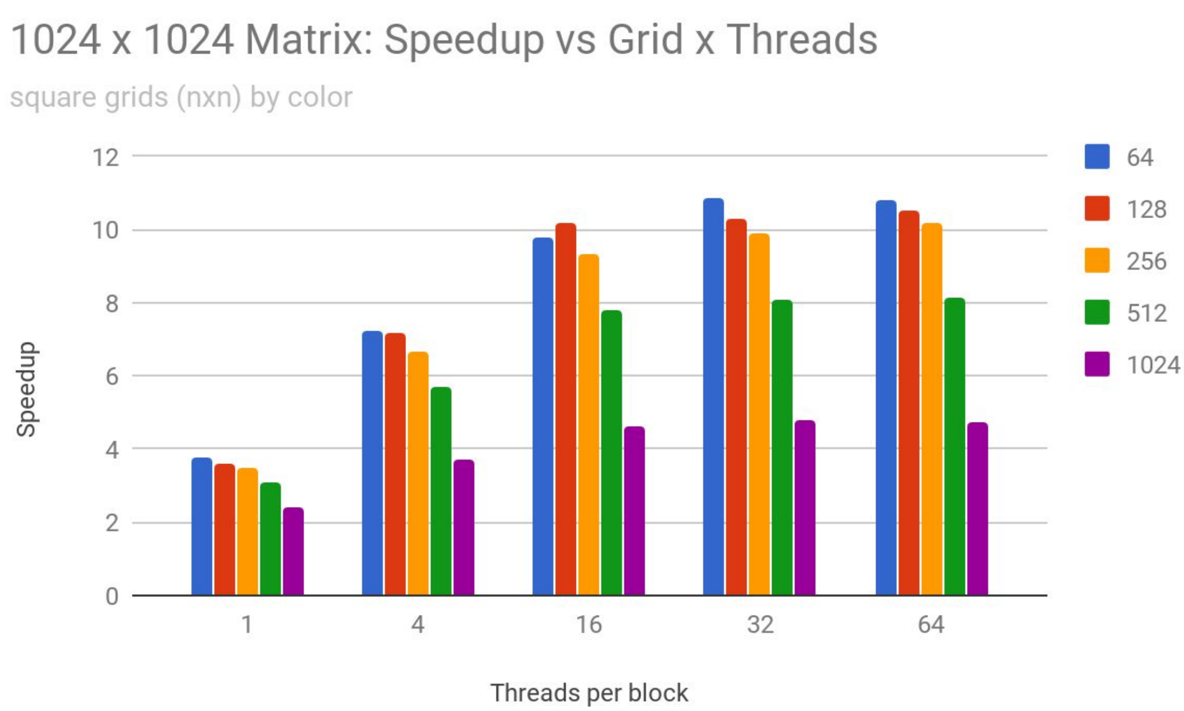
4096	64	128	256	512	1024
1	14.16674419	14.32691096	14.04776882	13.77290784	13.10097383
4	31.36648372	31.17356653	31.04717212	30.23918724	28.36422045
16	45.86626225	45.69775049	45.36552666	44.26649567	41.34392798
32	50.95300652	50.73613641	50.45274451	49.11925193	44.46136997
64	53.35488009	53.05467014	52.5342358	51.57504049	47.10302713

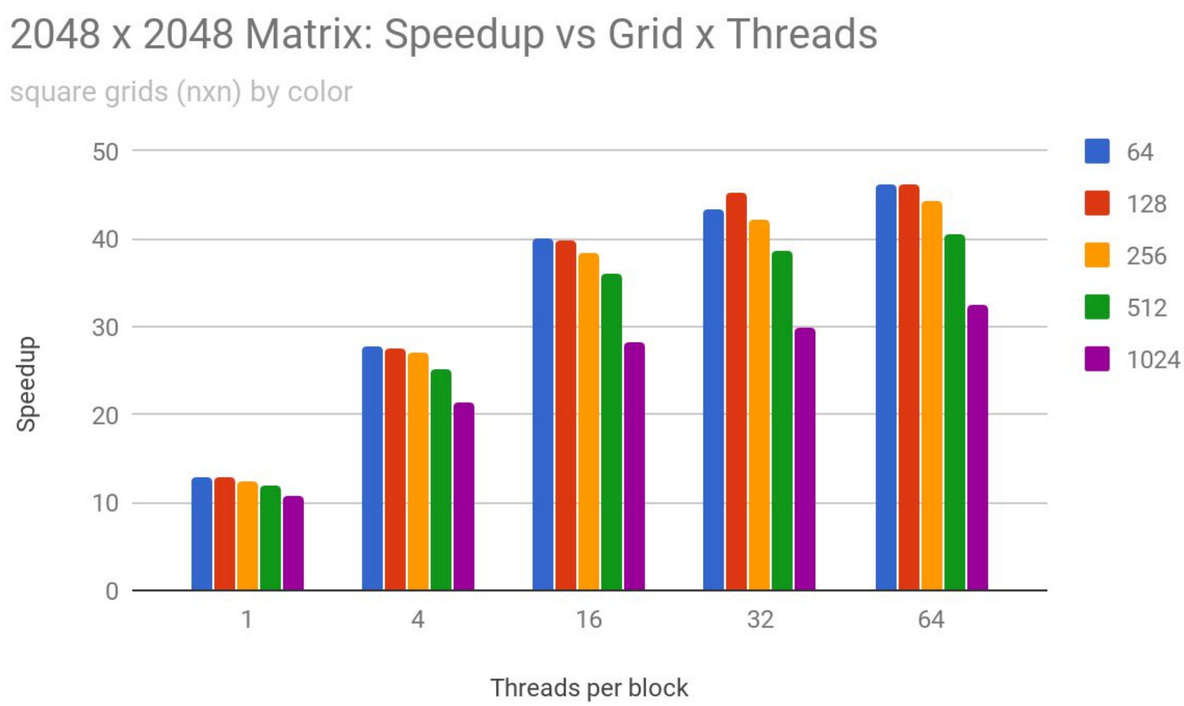
8192 x 8192 Matrix: Speedup vs Grid Size (DxD) x Number of Threads

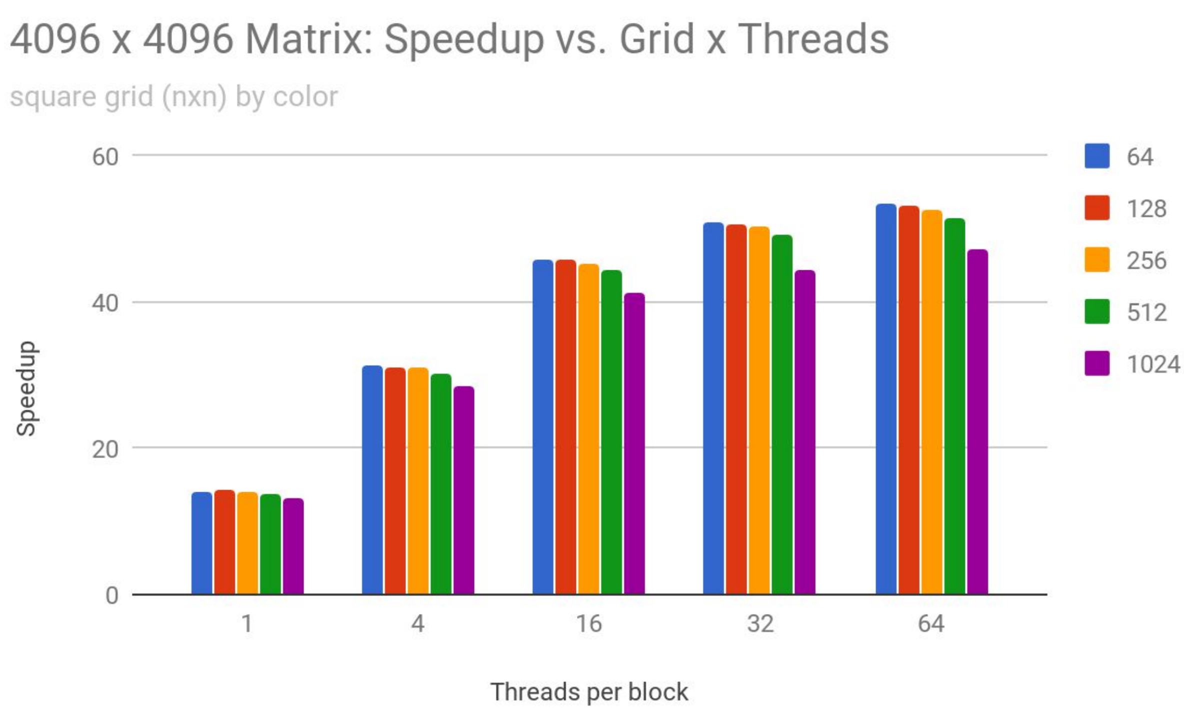
8192	64	128	256	512	1024
1	14.31875881	14.91563152	14.39992191	14.17817589	14.03497318
4	32.33210438	31.91260867	31.99958676	31.38097049	31.11253577
16	46.62131651	46.62837631	46.51027036	48.92873111	45.17890318
32	52.01042374	52.49693609	51.90266597	51.91396626	49.95199934

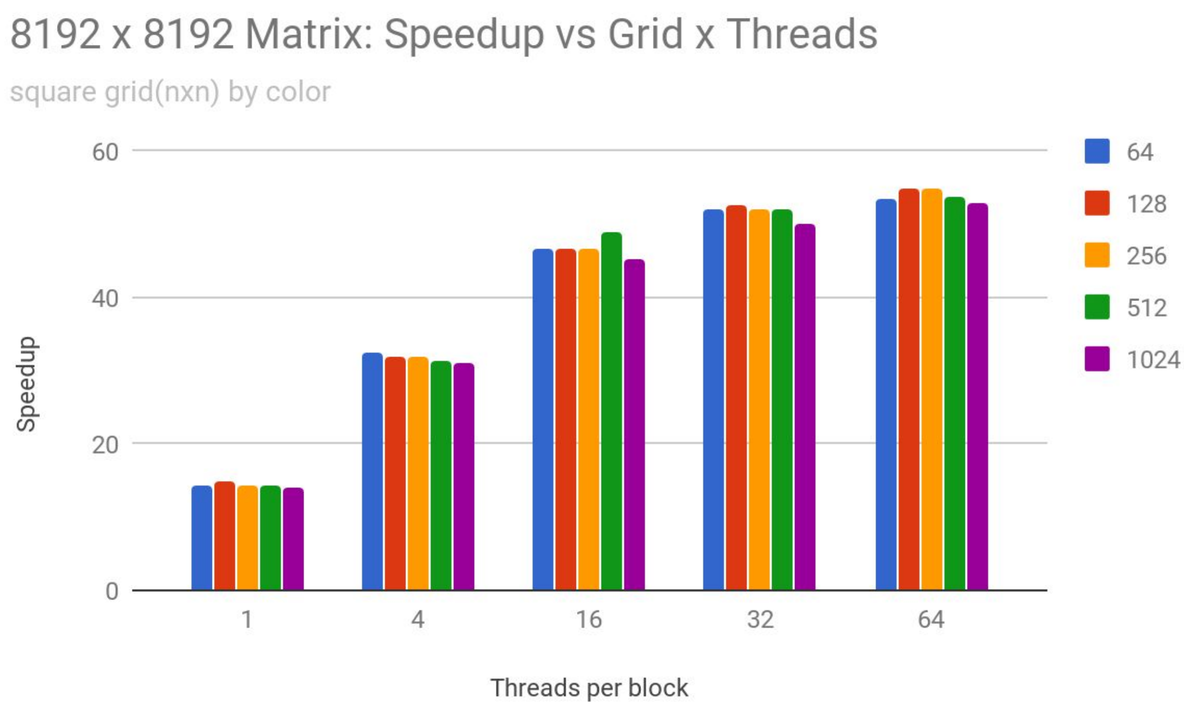
64	53.42120296	54.75493638	54.84490561	53.60690547	52.78215394






2048 x 2048 Matrix: Speedup vs Grid x Threads
square grids (nxn) by color
Speedup
Threads per block
0
10
20
30
40
50
1
4
16
32
64
64
128
256
512
1024


4096 x 4096 Matrix: Speedup vs. Grid x Threads
square grid (nxn) by color
Speedup
Threads per block
0
20
40
60
1
4
16
32
64
64
128
256
512
1024



Discussion:

The most striking aspect of the results so far have been the incongruities in thread distribution. I deliberately chose a few configurations that would “line up” the actual thread counts. However, blocks with single threads (and larger number of blocks) consistently under performed against numerically identical configurations with less blocks and more threads per block.

I did a bit of research online and stumbled on the concept of GPU warp value. According to my sources[1][2], the warp number refers to the number of bundled threads controlled by a “hardware scheduling unit.” Threads in a warp fire in a SIMD fashion, with all threads executing the same instruction. When threads diverge, either by being declared in blocks of non-multiple warp size, or by conditional execution, those threads get idled. By declaring thread values in multiples of this value, you get a more favorable mapping of tasks to threads.

To this end, I actually changed my code defaults to 32 threads per block. I had allow for X, Y thread dimension values (originally I’d set up blocks to be uniformly square values). The data (and graphs) seem to bare out the fact that matching threads per block to multiples of warp value increases uniformity of performance and utilization.

My initial testing strategy also had an impact on my data set. As I mentioned previously, I’d originally not included the warp value (32 thread) configuration as an option, instead opting for squares. As I was testing this new configuration, I also put some pauses to pad between testing calls, in the hopes of allowing for higher N in input. I still ran into the hardware memory limitations, but I did note that my returns had much less noise in the timing values. This made me suspect that the Torque system might have been running multiple jobs on the same node, causing them to interfere. Additional testing showed this to be the case. I changed my testing shell scripts to put 30 second gaps between tests. This eliminated the noise caused by overloading the GPU with requests.

Knowing now about the nature of the warp value, in general my results match expectations. Smaller data sets suffer from higher block and thread configurations that largely go unused and or interfere with execution of the kernel. As the data set steps up in size, the sweet spot of block size seems to move in tandem. However, high grid dimensions seem to underperform low consistently, regardless of thread per block settings.

Conclusion:

This simple experiment points out the need for tuning when it comes to implementation of GPU based solutions. The grid dimensions interact with the block thread sizes in ways that are not intuitive. Working from a block size based on the warp values seems the correct place to start, but experimentation would be required to dial in the appropriate numbers for maximum performance. Also, the size of the dataset being processed of course impacts those numbers as well.

References:

[1]Daniel Moth. “Warp or wavefront of GPU threads.” *Parallel Programming in Native Code*, blogs.msdn.microsoft.com/nativeconcurrency/2012/03/26/warp-or-wavefront-of-gpu-threads/.

[2]Stack Overflow "Why Bother to Know about CUDA Warps?." Gpu - Why Bother to Know about CUDA Warps? - Stack Overflow. N.p., n.d. Web. 2 Dec. 2017. <https://stackoverflow.com/questions/11816786/why-bother-to-know-about-cuda-warps>.