MP1

by Abhishek Johri

FEBRUARY 22, 2017

CS 484 PARALLEL PROGRAMMING
University of Illinois at Urbana Champaign

Part A

Part 1:

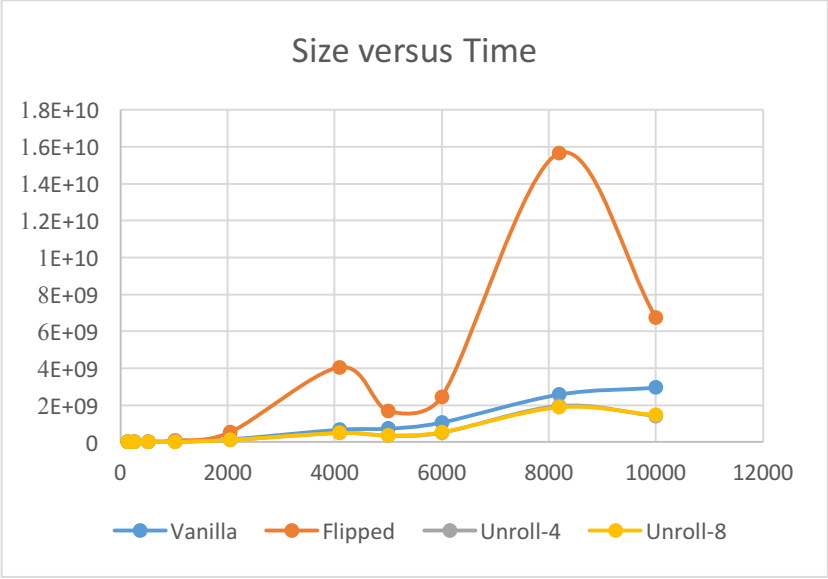
The following table is the data that I collected by running the following bash commands

./basic_perf.exe 128 128 32 32
./basic_perf.exe 256 256 32 32
./basic_perf.exe 512 512 32 32
./basic_perf.exe 1024 1024 32 32
./basic_perf.exe 2048 2048 32 32
./basic_perf.exe 4096 4096 32 32
./basic_perf.exe 5000 5000 32 32
./basic_perf.exe 6000 6000 32 32
./basic_perf.exe 8192 8192 32 32
./basic_perf.exe 10000 10000 32 32

Table:

Size	Vanilla	Flipped	Unroll-4	Unroll-8
128	246248	326721	198310	204565
256	1948228	4397125	956315	1052237
512	7848766	18042924	4302470	4365532
1024	29907185	71471315	17107408	15393311
2048	152598258	521175636	115509850	116417792
4096	674744449	4044595264	497935343	488876274
5000	731384655	1677261508	347155133	365839351
6000	1066980769	2450581355	511367341	533807923
8192	2583739079	15651455066	1953372307	1888490992
10000	2954561669	6725570670	1411969686	1470220268

Here is the graph of the following data:



Analysis:
So the first thing to notice in this experiment is that we kept the block size constant but change the size of the array thus increasing the number of elements we have in total. So we have 4 different parts ways of executing the same code to see which way give us the best cache performance. We have a Vanilla structure which is a simple brute force method, a Flipped structure which is the same as the Vanilla's brute force structure just that the inner and out loops are switched. The final two structures are similar since they are both unrolling structures but with a different unrolling amount. From the graph we can see that based off the Vanilla structure (being our baseline due to basic brute force structure) that our Flipped structure is not very cache efficient due to the increase in runtime. This increase in runtime is due from the fact that the program needs to access the actual memory more often than what is stored in the cache. After this we can see that the two programs that are more cache efficient than the Vanilla structure are the Unrolling structures. This is due to the fact that both of these method load a certain amount of elements at once in each iteration thus loading it into the cache right away. This helps because then a certain number of elements, that are repeatedly needed, are on the cache for quick access to. The only time that this wouldn't work would be if the fixed number of elements you choose to go through in one iteration is greater than the cache line.

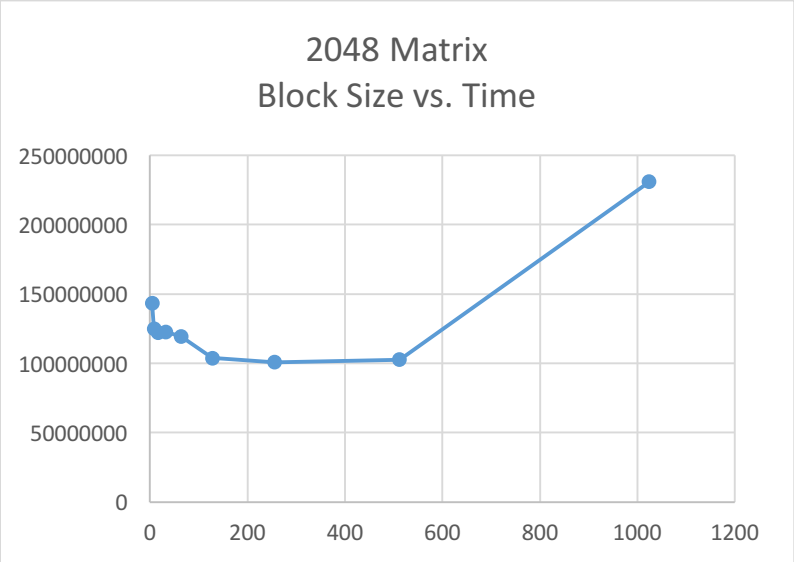
Part 2:
The following table is the data that I collected by running the following bash commands:

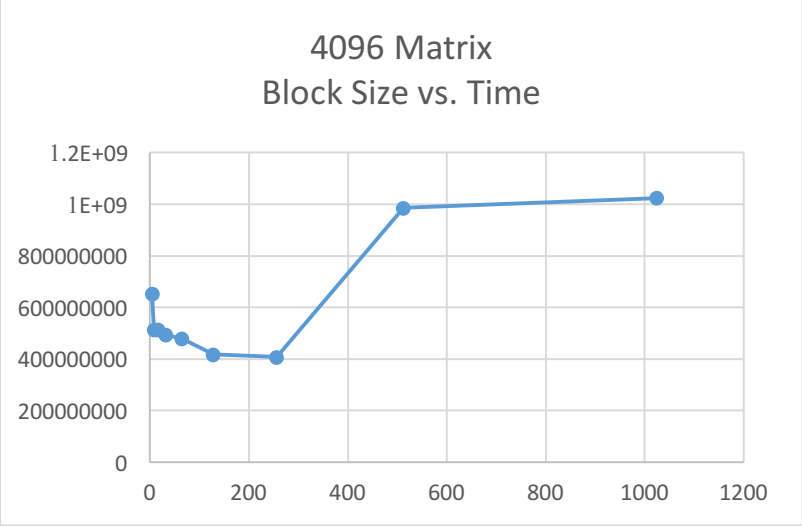
```
./basic_perf.exe 2048 2048 4 4
./basic_perf.exe 2048 2048 8 8
./basic_perf.exe 2048 2048 16 16
./basic_perf.exe 2048 2048 32 32
./basic_perf.exe 2048 2048 64 64
./basic_perf.exe 2048 2048 128 128
./basic_perf.exe 2048 2048 256 256
./basic_perf.exe 2048 2048 512 512
./basic_perf.exe 2048 2048 1024 1024

./basic_perf.exe 4096 4096 4 4
./basic_perf.exe 4096 4096 8 8
./basic_perf.exe 4096 4096 16 16
./basic_perf.exe 4096 4096 32 32
./basic_perf.exe 4096 4096 64 64
./basic_perf.exe 4096 4096 128 128
./basic_perf.exe 4096 4096 256 256
./basic_perf.exe 4096 4096 512 512
./basic_perf.exe 4096 4096 1024 1024
```

Table:

2048 Matrix		4096 Matrix	
Block Size	Time	Block Size	Time
4	143529251	4	651767983
8	124894741	8	513867767
16	122279548	16	514259963
32	122835633	32	493769709
64	119327383	64	480038509
128	103872830	128	417607486
256	100811823	256	407882871
512	102547141	512	985631486
1024	230939097	1024	1024154124

Here are the graphs for the following data:





Analysis:
So the first thing to note is that we are keeping the size of our matrix constant throughout all of the experiments for this part. Different from the first part our program now has a blocking structure which in some ways is similar to that of the unrolling structure. The main thing that we are changing here is the block size for our program to see around where the program is most cache efficient. So what we expect is that the graph will dip down till it hits a global minimum. This minimum indicates the point at which the program is most cache efficient. This is because since more as-needed data is on the cache the program will run faster giving a lower time. This is shown and supported in the graph.

Part B

Data:

32x32 Matrix	Threads	Tile Size	Time
Sequential			0.007508
Pipeline	1	2	0.016802
	1	4	0.008716
	1	8	0.008604
	1	16	0.009263
	1	32	0.009673
Diagonal	1		0.032572
Pipeline	8	2	0.032572
	8	4	0.025689
	8	8	0.020615
	8	16	0.016623
	8	32	0.016287
Diagonal	8		0.270051
Pipeline	16	2	0.046505
	16	4	0.040551
	16	8	0.03358
	16	16	0.034711
	16	32	0.021051
Diagonal	16		0.359218

128x128 Matrix	Threads	Tile Size	Time
Sequential			0.108634
Pipeline	1	2	0.142069
	1	4	0.094838
	1	8	0.093328
	1	16	0.099562
	1	32	0.103528

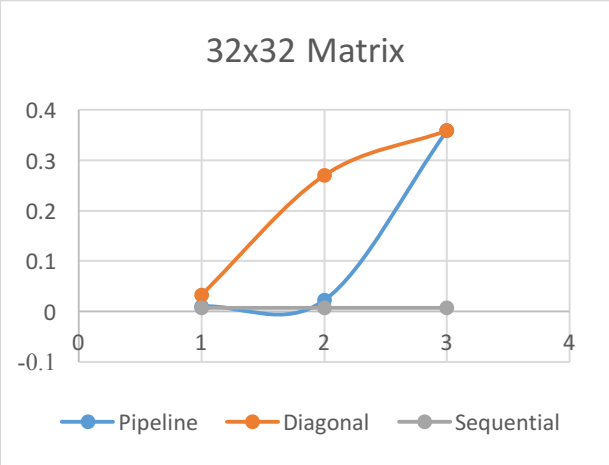
Diagonal	1		0.13691
Pipeline	8	2	0.139784
	8	4	0.094034
	8	8	0.061982
	8	16	0.048418
	8	32	0.054444
Diagonal	8		1.138972
Pipeline	16	2	0.194434
	16	4	0.133815
	16	8	0.094425
	16	16	0.073615
	16	32	0.074475
Diagonal	16		1.619333

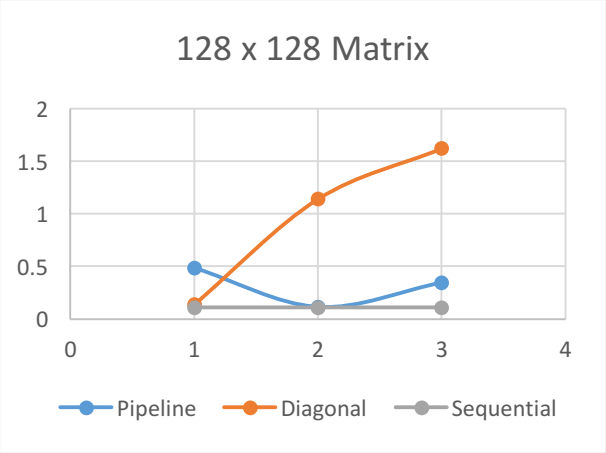
512x512

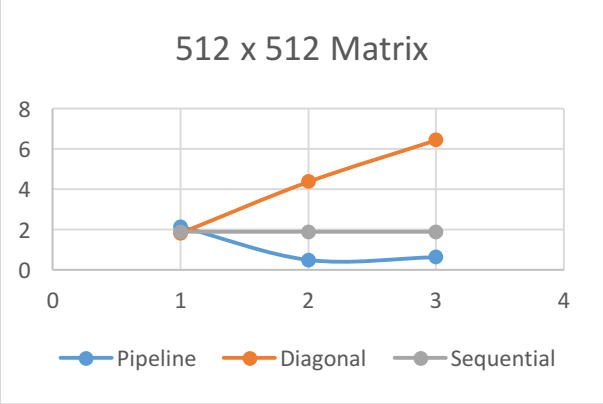
Matrix	Threads	Tile Size	Time
Sequential			1.874097
Pipeline	1	2	2.599762
	1	4	2.193649
	1	8	1.997742
	1	16	1.928024
	1	32	1.892272
Diagonal	1		1.807138
Pipeline	8	2	0.766532
	8	4	0.507296
	8	8	0.389101
	8	16	0.35484
	8	32	0.374735
Diagonal	8		4.356323
Pipeline	16	2	1.073982
	16	4	0.689934
	16	8	0.492032
	16	16	0.405074
	16	32	0.403495
Diagonal	16		6.430475

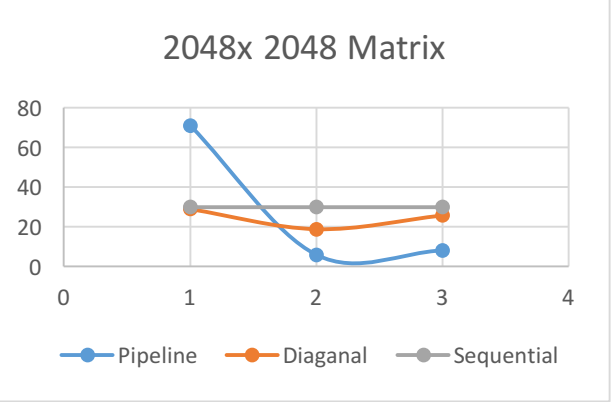
2048x2048 Matrix	Threads	Tile Size	Time
Sequential			29.980702
Pipeline	1	2	109.425753
	1	4	97.853393
	1	8	62.029272
	1	16	45.708248
	1	32	40.08166
Diagonal	1		29.00359
Pipeline	8	2	8.560414
	8	4	6.113461
	8	8	4.856881
	8	16	4.613234
	8	32	4.558289
Diagonal	8		18.861675
Pipeline	16	2	9.359526
	16	4	5.936645
	16	8	4.578692
	16	16	9.230311
	16	32	10.693376
Diagonal	16		25.821092

Graphs:









a) Only in the last case when the matrix size was 2048 x 2048 was when the diagonal method had a better running time as the number of threads had increased. Overall though the pipeline method had the best results when the size was 128 or greater. As the number of threads had increased the running time would improve.
b) The best chunk size was of size 32. This is because the chunk sizes were able to fit in the LC1 cache and thus larger chunks would have better runtimes. This is due to the fact that the larger values were loaded into the cache and so there was no need to reload them.
c) The diagonal method only improved when the size of the matrix was 2048 x 2048 as the number of threads increased. For the pipelined method the execution time improved as the number of threads increased. For the gradient of the curve it levels off as one would expect it to. This happened because of other factors that limited it's functionality. These factors are things like cache size or the number of chunks that could be calculated at any given moment.