CSE 260 PA #1

Here is a general rubric. You may get credit for additional things not on the rubric (e.g. extra graphs, introduction, other stuff we haven't explicitly thought of so think of this as a general guide. Depth of analysis and clarity are key objectives.

***** Your report must follow the format given here so we can easily find the different sections. Reports that require extensive time for us to read will be penalized. (up to -10 pts if report is not organized well).**

Checkpoint Submission (below)

Partner Waiver (fill out only if you have permission to do the assignment alone):

I have talked to _________(Name of the TA/Professor) on ________ (date) regarding the assignment to be done individually.

Reason:

Q1. Please fill out the following information (3 pt).

Student 1 (print name) : Mridul Kavidayal

Student 2 (print name) : Shivam Lakhotia

Git Repo link: https://github.com/cse260-wi20/pa1-pa1-mkaviday-slakhoti

Name of AMI instance (name your instance with your team name as in "pa1-b5chin-tsundara"):

Checkpoint Submission (above)

The Analysis for the report should address the following questions based on the optimization you do in the program. You are free to add extra insights regarding the assignment.

Q2. RESULTS - 15 pts

Give a performance study for a few values(about 20 different values) of N from 32 to 2048 on your optimized code - data should be in the file data.txt (see "what to submit->data file"   for specific format. You will lose points if you do not follow this format.)

Q2a Show speedup over naive code for 6 interesting points you select

Size	**GFLops naive**	**GFLops optimized**
513	1.68	5.96
641	1.55	8.55
669	1.55	7.1
896	.885	15.1
640	.875	14.9
1024	.145	11.2

Q2b Point out and high level explain irregularities in the data (Places where performance scales in a non-linear way). Detailed analysis should be in the next section.

For sizes which have a higher chunk not fitting in exactly to the block sizes, we see a lower Gflop. Also these have high cache misses as is shown using valgrind in later section.

Q2c Show performance for the following numbers: (fill out the table).

N	Peak GF
32	0.784
64	3.16
128	13.3
256	13.3
511	8.87
512	11.3
513	5.96
1023	9.24
1024	11.2
1025	7.18
2047	9.3
2048	11

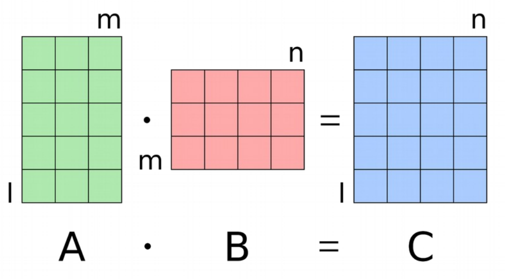
Q3. ANALYSIS 35 pts

Clearly Describe:

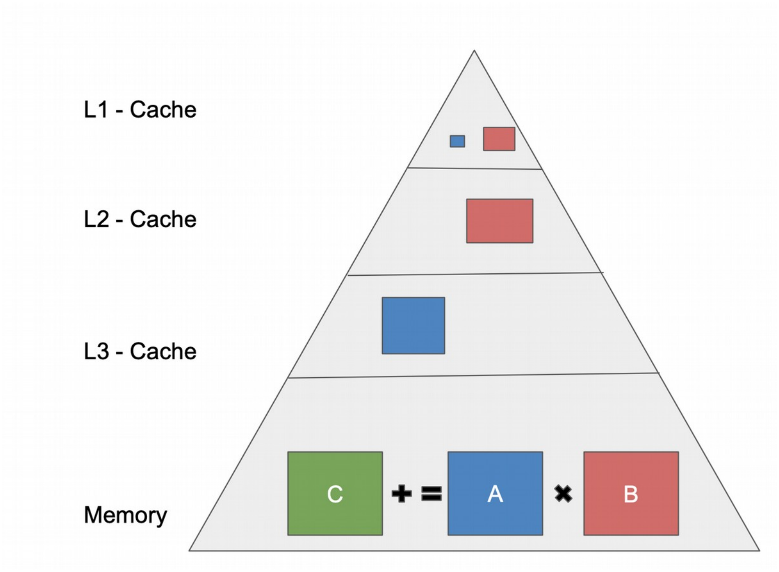
Q3a How does the program work - don't include the source code, instead describe it in prose, flow chart, p-code, etc.

Basics:

Matrix Multiplication:
The task was to improve the speed of matrix multiplication. In other words, given three NxN matrices, A, B, and C, we were supposed to improve the speed of the following operation: C+=A.B. Matrix multiplication is an O(N3) operation as for each element of C we have to perform N operations and since there are N2 elements in C, the order is N3 The following diagram explains how matrix multiplication works.



Now, the flow of our method is as follows:
We first divide the matrix into blocks of different sizes. The figure below shows how we divide the matrix and store which portion in which cache.



Memory:

The order in which we process these blocks is **i->j->k.** This order of i, j and k experimentally gave us the best results.

L3 Cache:

At this level, we also cache the A block i.e we copy the A block in a separate array so that it fits in the L3 cache and is stored in continuous memory. At L3 cache the flow is **i->k->j.** This is done to obtain the entire row of B.

L2 Cache:

We copy the block of B in a separate memory at this level in a continuous memory location as we will be accessing B many times. At L2 cache the flow is **j->k->i.** This is because now we want to use the entire B block for this cached A block.

L1 Cache:

We copy the sub-block of B in yet another continuous location. Now, we call the block SIMD to process these blocks in size of 8x4 using AVX intrinsics. At L1 cache the flow is **i->k->j.**
This is done to process the entire cached-block in L2.

SIMD:

Finally, an 8x4 block of A is multiplied with a 4x4 block of B to generate a 4x4 block of C.

Q3b. Development process: What did you try, what worked, what didn't work, theories on why. Negative results are sometimes as illuminating as positive results, so try to explain as best as you can.

Include the necessary graphs for the optimizations implemented.

Dividing the matrices in blocks: This is the first step we perform. We divide All the 3 matrices into blocks and apply matrix multiplication on these blocks. In the diagram above you can assume that each box is a block and multiplication os blocks is done in the same way, two matrices are multiplied.

This was done because for large Matrices (2048x2048), they don't fit in the caches. Of dividing by dividing them into blocks we can perform the matrix multiplication more efficiently. The problem with this approach is that consider matrix B. To get the first block of B, we will need to do multiple Cache loads because the Matrix is stored in Row-Major order in C, so the second row of B will start only after the entire first row. Now, when we access the first element of B, a block of B will be loaded into the cache. It is highly likely that this is partial first row. To access the second row, we will have to load this again as these elements won't be their in the cache.

Transposing the B Matrix:

If B were stored as its transpose, then this problem wouldn't have occurred. Because for each row of A, you multiply it with the same row in B. So this was the first approach we tried and it did give us some better results as there were more cache hits. Apart from increasing performance, we also observed that speed remains consistent, meaning previously we noticed that matrix multiplication of some matrix was fast for some it was slow. There was a lot of variance because of cache-misses.

SIMD:

Next, we were taught SIMD instruction. That is Single-Instruction-Multiple-Data. This was done using AVX intrinsics. In the beginning, we implemented a 2x4 block multiply using AVX intrinsics. This gave us a boost to 2 GFlops. But the problem was this couldn't be used in combination with transpose. After writing AVX code for multiplying the transposed matrix, we didn't see any improvement. The reason was that for each multiplication we had to write it back to C in memory, this I/O resulted in a lot of latency and we didn't get any good results with transposed Matrix.

SIMD - Enhanced:

After obtaining really good results with SIMD, we experimented with various block sizes.

Block Size	GFLops
2x4	~4.7
4x4	~7.4
5x4	~8.5
8x4 then 2x2 on remaining	~10.5

Finally, we decided to use 8x4 because it gave the maximum speed up.

Loop Unrolling:

Next, we implemented loop unrolling. This gave us some marginal improvements. The reason this was done to reduce the jump statements in the I-Cache.

Caching:

Till this point, we were not storing the blocks. This resulted in a lot of cache-misses as for the second line of each block we had to read from the memory. To fix this we copied the block of B in a new memory location. This ensured that we won't get cache-misses now when accessing the rows of B. This was also done for A but not for C.

Level based caching:

After this, we implemented separate caching for L2 cache and L1 cache. As L1 cache is much smaller than L2, now we faced the issue of L1 cache miss instead of L2 which we previously fixed.

<u>Rectangular block caching:</u>

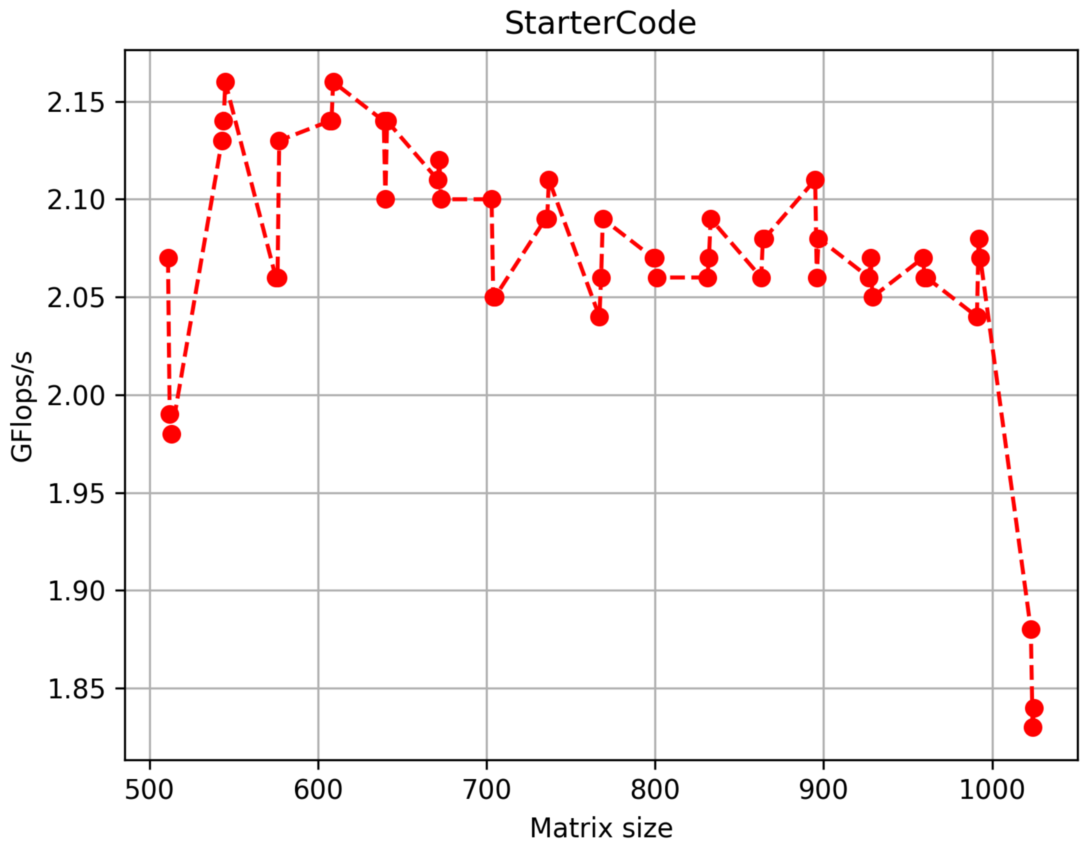
Next, we implemented rectangular block caching as compared to square blocks. The motivation behind this was to reduce the cache-misses while caching the block. By caching a rectangular block, keeping the (# of cols) > (# of rows), ensures that we don't need to read more lines for the same amount of data.

<u>Padding:</u>

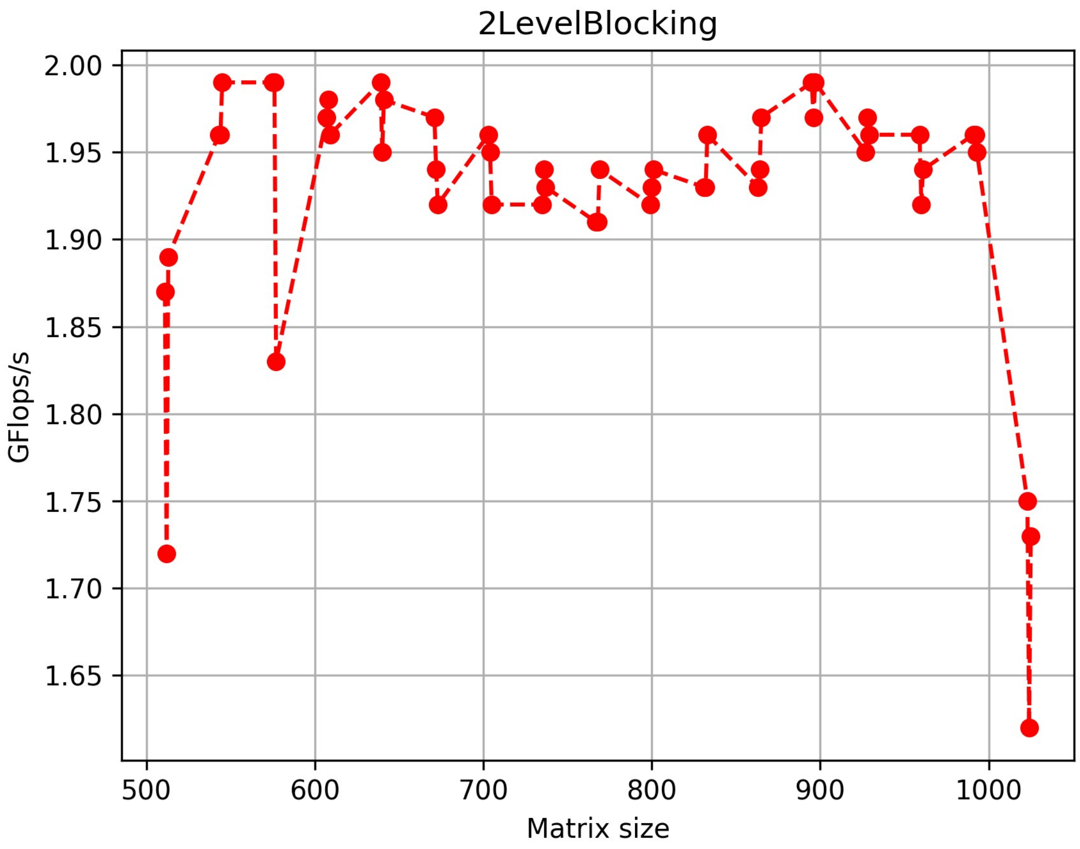
For all the above code, we had to handle the corner cases separately when the rows and cols of the Matrix were not multiples of out blocks. We speculated that this was causing a lot of jumps in the code. To ameliorate this, we padded our original matrix so that it was now a multiple of block size.

<u>Block sizes:</u>

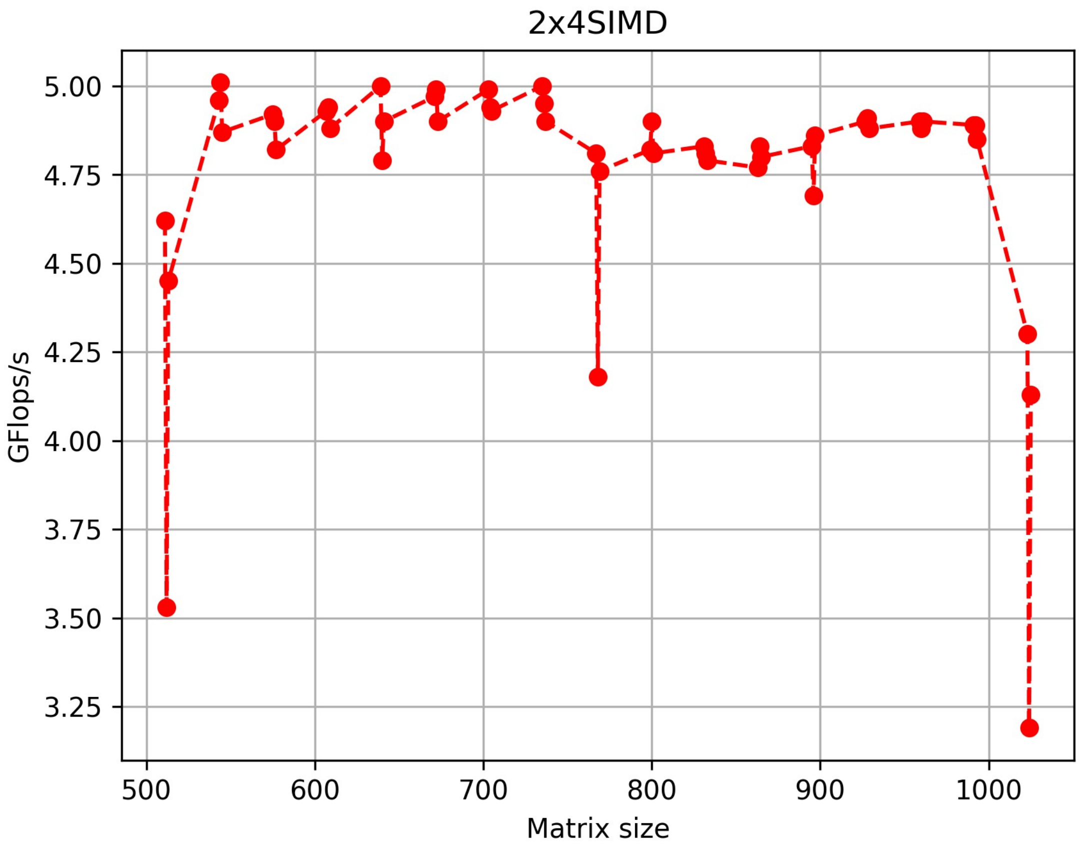
Further, we decided to keep the block sizes at all levels multiple of 2. This ensured that after padding, we were able to experiment with multiple sizes of L2 and L1 cache blocks.



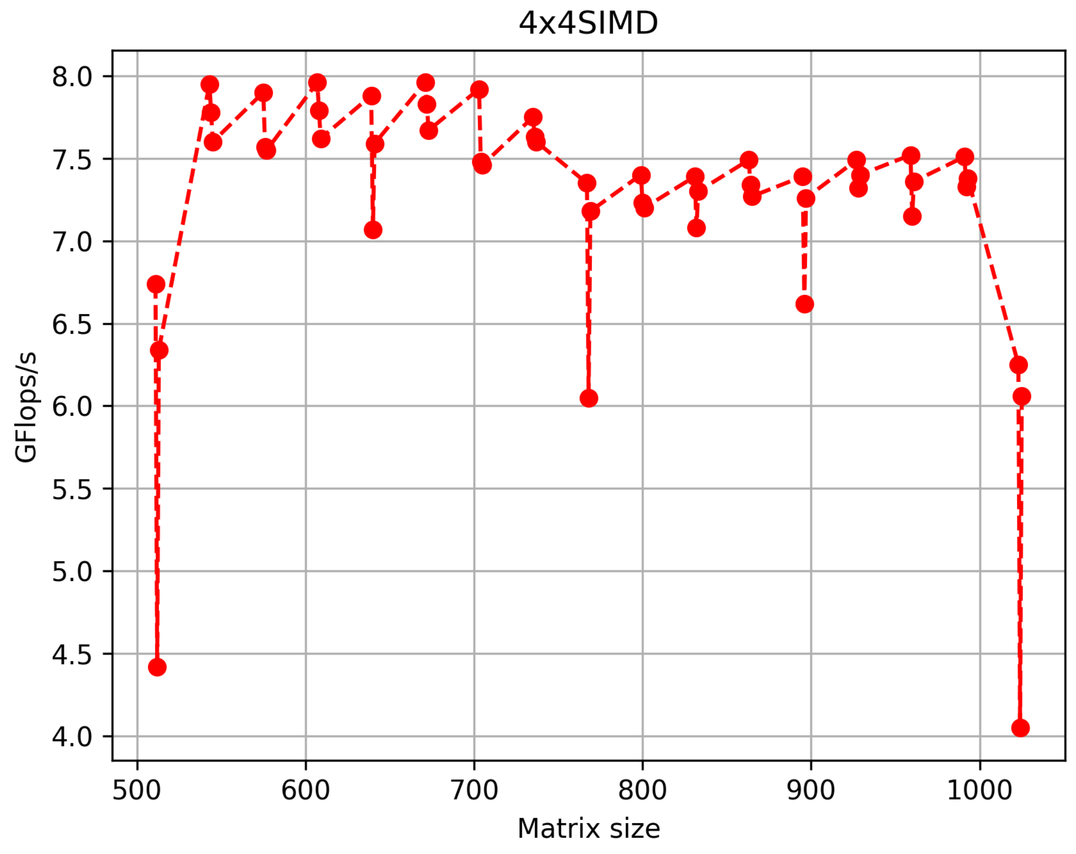
Starter Code. Compiled with optimizations and Transpose flag enabled.



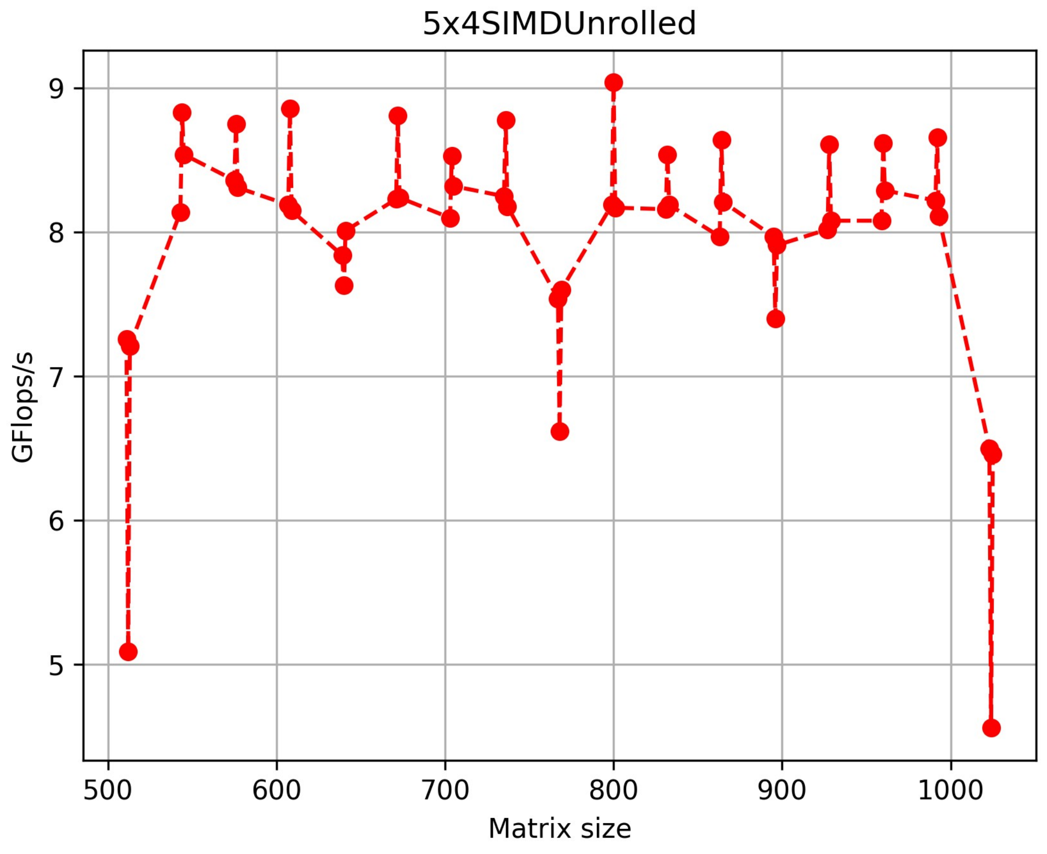
2 Level Blocking. Code compiled with optimizations and Transpose flag enabled.



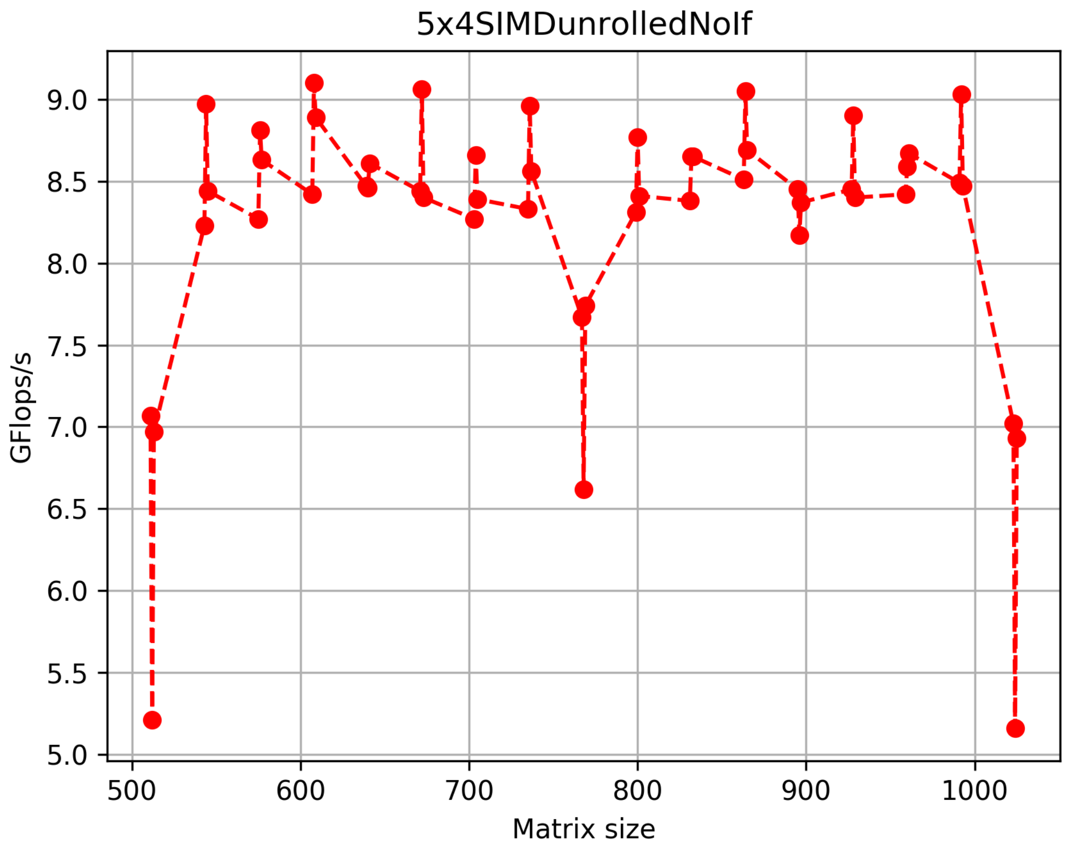
2x4 SIMD multiplication. Remaining portion handled with naive multiplication.



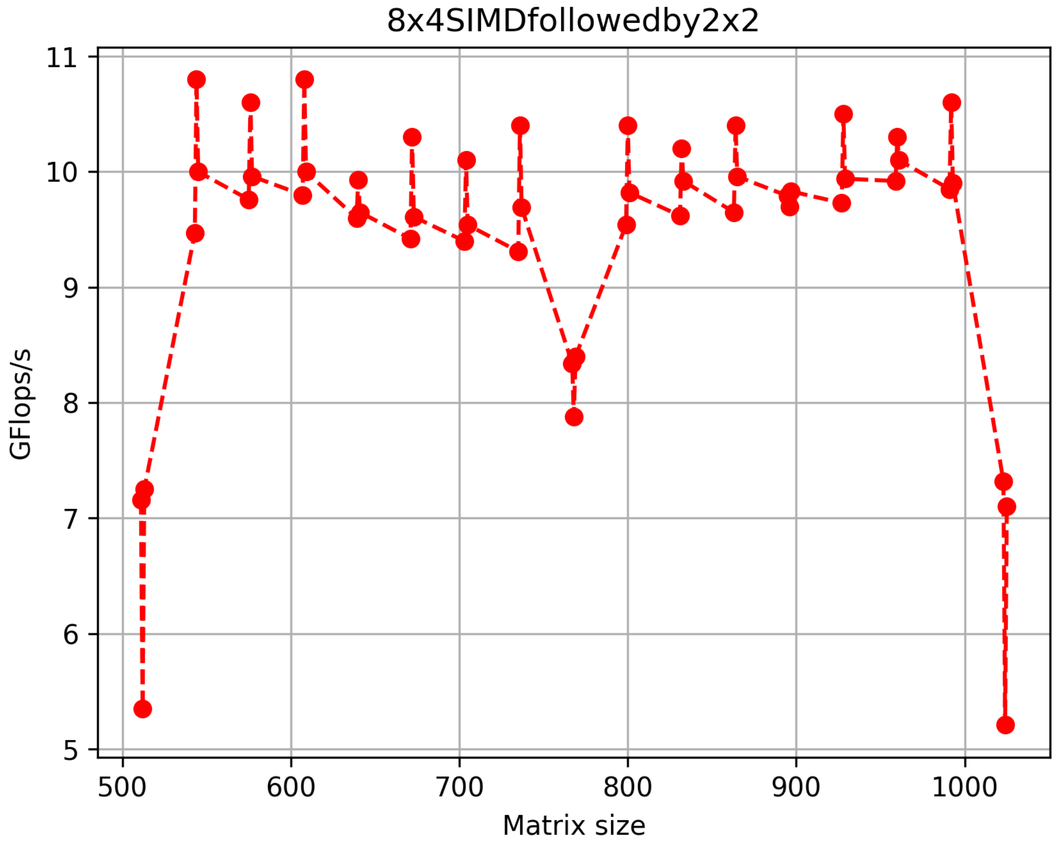
4x4 SIMD multiplication. Remaining portion handled with naive multiplication.



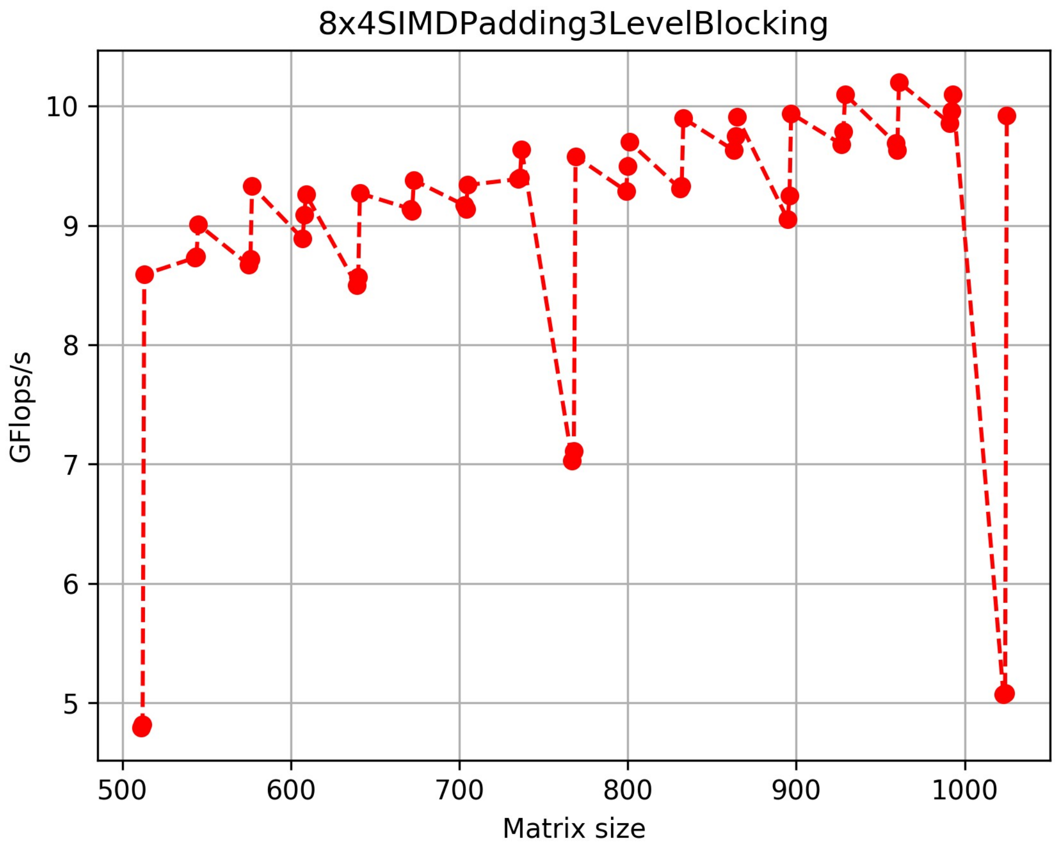
5x4 SIMD multiplications and innermost loop unrolled. Remaining portion handled with naive multiplication.



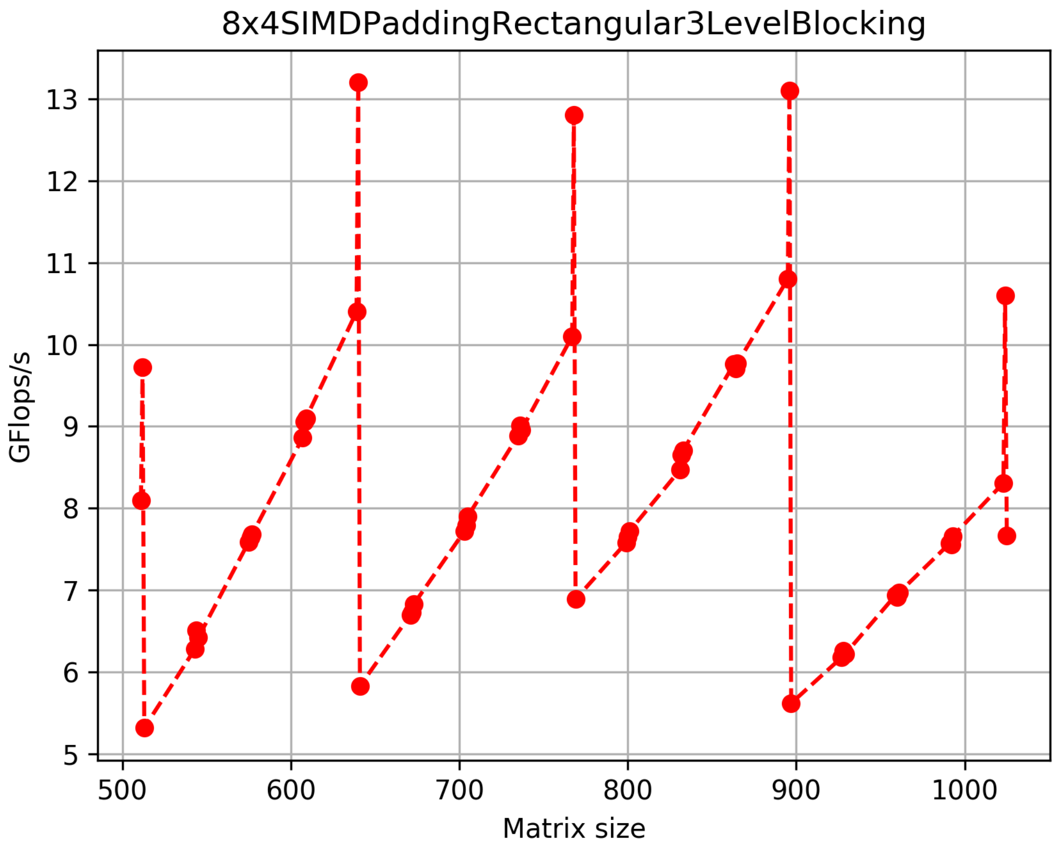
5x4 SIMD multiplications and innermost loop unrolled. If check on size removed. Remaining portion handled with naive multiplication.



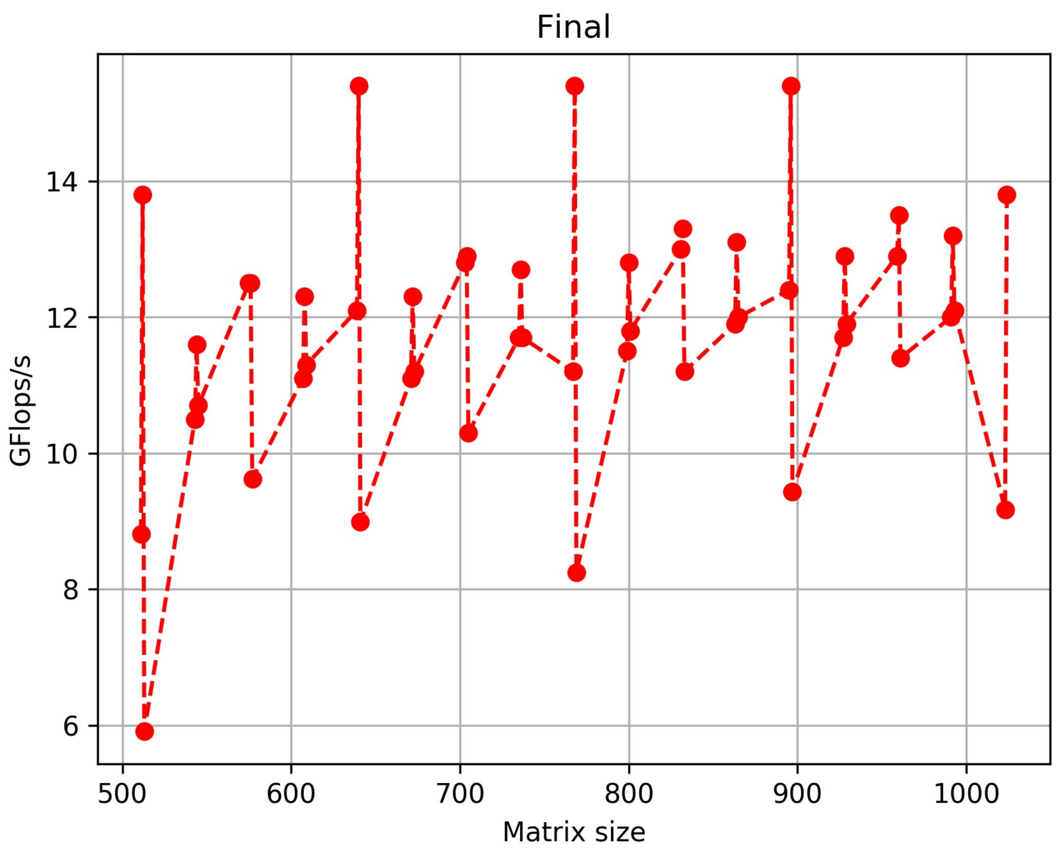
8x4 SIMD multiplications and innermost loop unrolled. Remaining portion handled with 2x2 SIMD. Finally naive.



8x4 SIMD multiplications and innermost loop unrolled. Remaining portion handled with naive. Copying while padding is creating overhead!

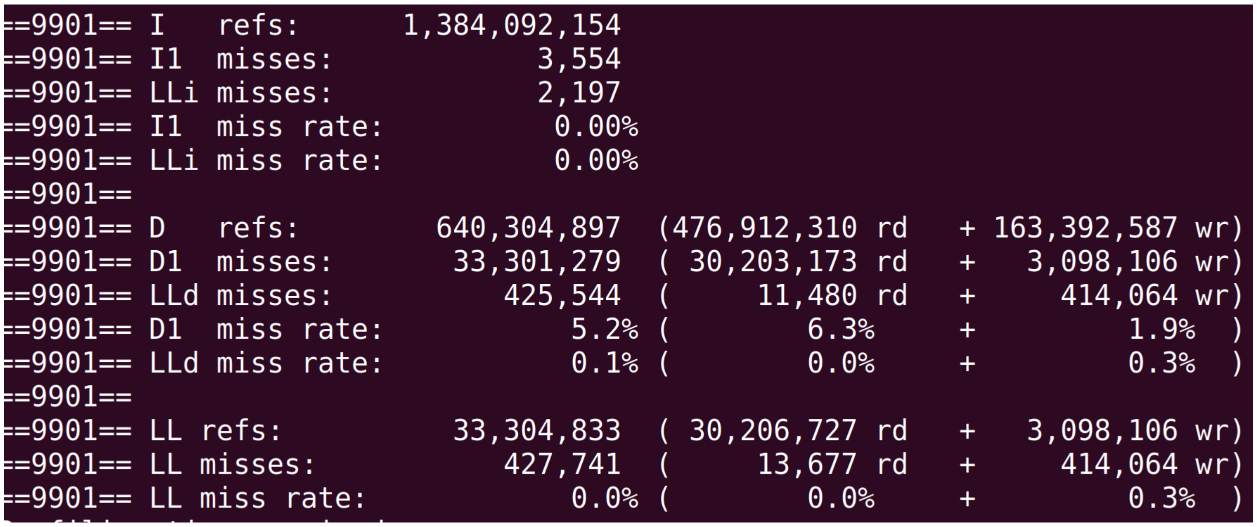


8x4 SIMD multiplications and innermost loop unrolled. Remaining portion handled with naive. Not able to find the right block sizes..


Final
GFlops/s
Matrix size
6
8
10
12
14
500
600
700
800
900
1000

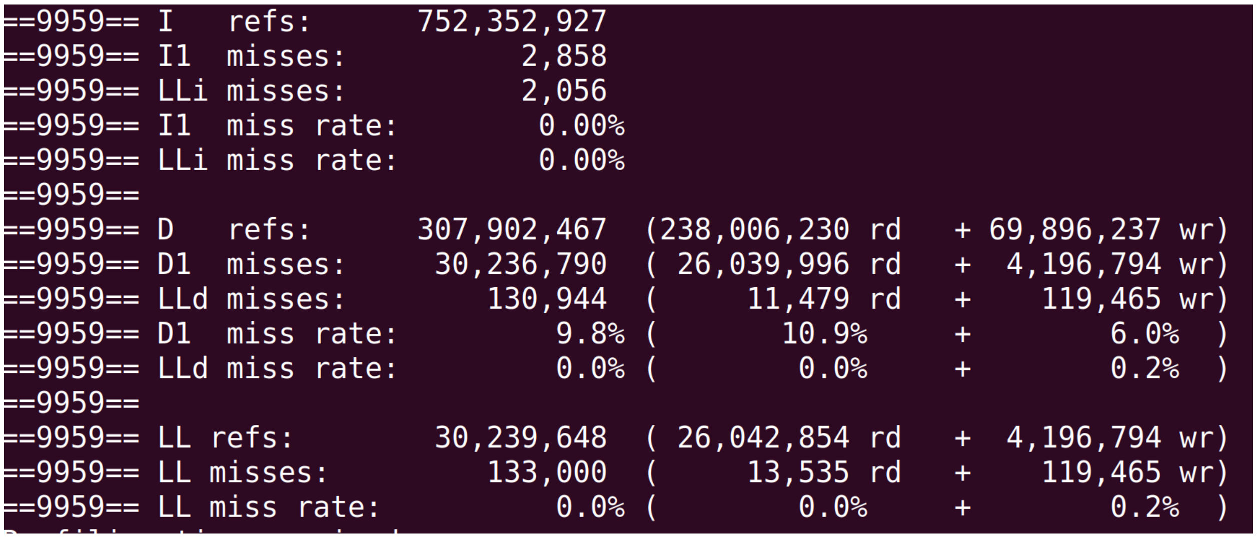
Q3c. supporting data - e.g. analysis of <u>cache behavior</u>, parametric searches,   or whatever will support your conclusions. Feel free to use tools such as cachegrind and knowledge of the machine's micro-architecture to support your theory.

For sizes 256 and 512, the following is the output of valgrind. L1 miss rate is around 5.2% while L3 miss rate is around 0%



```
==9901== I   refs:      1,384,092,154
==9901== I1  misses:            3,554
==9901== LLi misses:            2,197
==9901== I1  miss rate:          0.00%
==9901== LLi miss rate:          0.00%
==9901==
==9901== D   refs:        640,304,897  (476,912,310 rd   + 163,392,587 wr)
==9901== D1  misses:       33,301,279  ( 30,203,173 rd   +   3,098,106 wr)
==9901== LLd misses:          425,544  (     11,480 rd   +     414,064 wr)
==9901== D1  miss rate:           5.2% (        6.3%     +         1.9%  )
==9901== LLd miss rate:           0.1% (        0.0%     +         0.3%  )
==9901==
==9901== LL refs:          33,304,833  ( 30,206,727 rd   +   3,098,106 wr)
==9901== LL misses:           427,741  (     13,677 rd   +     414,064 wr)
==9901== LL miss rate:            0.0% (        0.0%     +         0.3%  )
```

For sizes 257 and 513, the following is the output of valgrind. L1 miss rate is around 9.8% while L3 miss rate is around 0%.



```
==9959== I   refs:      752,352,927
==9959== I1  misses:          2,858
==9959== LLi misses:          2,056
==9959== I1  miss rate:        0.00%
==9959== LLi miss rate:        0.00%
==9959==
==9959== D   refs:      307,902,467  (238,006,230 rd   + 69,896,237 wr)
==9959== D1  misses:     30,236,790  ( 26,039,996 rd   +  4,196,794 wr)
==9959== LLd misses:        130,944  (     11,479 rd   +    119,465 wr)
==9959== D1  miss rate:         9.8% (       10.9%     +        6.0%  )
==9959== LLd miss rate:         0.0% (        0.0%     +        0.2%  )
==9959==
==9959== LL refs:        30,239,648  ( 26,042,854 rd   +  4,196,794 wr)
==9959== LL misses:         133,000  (     13,535 rd   +    119,465 wr)
==9959== LL miss rate:          0.0% (        0.0%     +        0.2%  )
```

As we had seen in the table, 513 had lower Gflops compared to 512 and 257 had lower Gflops compared to 256. One reason could be significant chache misses which becomes apparent here.

Q3d. Future work - what could you do if you had more time

We would want to copying matrices A and B at higher levels so that they make use of cache effectively. Thoroughly understanding “Anatomy of High Performance Matrix Multiplication” and implementing it.

TEAMEVAL :

CSE 260, Parallel Computation

(Winter 2020)
Department of Computer Science and Engineering
University of California, San Diego

Team Self Evaluation Form, Assignment #1

Each team must submit one copy of this self evaluation form.
The members of each team should discuss how they
worked together and what to write for the evaluation.
If you worked alone leave column B blank

(1) List the names of your team members:

A: Mridul Kavidayal

B: Shivam Lakhotia

(2) Estimate how much time each team member devoted to this project, in HOURS.

	A	B
meetings	10	10
coding	40	40
writeup	8	8
planning (alone)	2	2
total (including meetings)	60	60

```
(3) Discuss the following.
(a) If you worked in a team, what were the major responsibilities
    of each team member?
    Both of us planned the work required to do and then divided it
accordingly.
    Mridul did the SIMD and Blocking. Shivam did the 3-Level caching,
transpose and goto-paper impl.
(b) Did you complete the assignment?
    Yes.
(c) What were your major strengths and weaknesses in doing thisx
    assignment (individually or in a team)?
    Strength was trying out a lot of possibilities. We experimented a lot.
Only a few experiments did give us results but we
    did learn a lot.
    Weakness was not able to figure out how to combine appropriately various
method. A lot methods were giving us improvements but
    we couldn't combnine all and get the best improvement.
    If in a team, also discuss how your team worked together.
(d) What  lessons did you learn from these events.
    It is important incremently keep taking one branch forward in parallel to
experimenting.
(e) If in a team, whether and how you plan to change the way
    your work together .
    No. I think worked pretty in sync and did the tasks as expected.
(f) Anything else that comes to mind.
    This assignment taught us a lot but loosing masks hurt!
(g) What suggestions do you have to improve this assignment?
    Give all posibilities/ways to imporve to the students and let students try
out all of them
    instead of them finding what are ways to improve because sometimes we get
stuck with one method.
    And that takes up a lot of time.
```

REFERENCES

Anatomy of High-Performance

MatrixMultiplication