Please put your name on top
title of the document helps too
You can organize your documents in your google folder, so I know exactly where to look, i.e. all files for Homework 1 put into folder called Homework1, or something along those lines.

(a) (4 pts) Before you write any code, look at the three algorithms above. Based on what you have learned in class, can you make a prediction, which one would be best/worst? For example, in Alg. 1 you can see that C[i][j] does not depend on index k, so it can be reused in the innermost loop. Besides, matrix A is traversed in the row-major order, as the second index changes faster than the first index. However, matrix B is traversed in column major order, so it may not be the most optimal way. Write down your prediction, and explain why you think it is the best algorithm. It is ok to be wrong with this one, as long as you give me your rationale. You can look at Chapter 1.5.2.3. of the Eijkhout book (see the Resources folder in Google Drive) for extra guidance, but please do not copy the discussion that is there. I am interested in your thoughts on this one.

total score: 20 pts FINAL SCORE: 18 pts
I should significantly reduce your score due to late submission, but I appreciate the extensive testing you have done with the loop unrolling and cache blocking. For that reason I only take off 10%. You got really good results, and provided good discussion. Please work on the language here and there, to improve your scientific writing.

Algorithm 1:

```
for (i = 0; i < n; i++){
  for (j = 0; j < n; j++){
    for (k = 0; k < n; k++){
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

Algorithm 2:

```
for (i = 0; i < n; i++){
  for (k = 0; k < n; k++){
    for (j = 0; j < n; j++){
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

Algorithm 3:

```
for (k = 0; k < n; k++){
  for (i = 0; i < n; i++){
    for (j = 0; j < n; j++){
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

Matrices have a significant impact on modeling large, dynamic systems such as weather and climatic conditions and various scientific models. Therefore, implementing good programming strategies to optimize matrix operations at a lower level would increase the performance of many applications. In order to avoid cache miss and optimize multiplication algorithm, the simplest fix to this is to reorder the loops to get better cache hits, as there are 3 loops, there are a total of 6 possibilities.

But in general, there are only 3 distinct arrangements that differ in the access type

cache

Algorithm 1) matrix C is calculated with temporal locality, therefore as the most inner loop (k) changes, the elements of C are kept at cpu catch and this makes it very efficient. Matrix A is used in a row major order as k represents the columns and for C language this is also beneficial as we consider spatial locality. Matrix B on the other hand, is column major order and therefore is not

optimized in any way. In this algorithm matrix C has been calculated in most efficient way compared to the other two.

Algorithm 2) matrix C is calculated with spatial locality, as the most inner loop (j) changes the elements of columns in C and therefor for next values of k there is high probability that catch would not be large enough and data are flushed for the next use. Matrix A is used with temporal locality and there it remains in catch until the inner loop is completed and is more efficient. Matrix B on the other hand, is row major order and therefore is also optimized to some extent. The performance of the algorithm depends of number of columns of C (j) if it is larger than cache then the performance will drop severely.

Algorithm 3) matrix C is calculated with spatial locality, as the most inner loop (j) changes the elements of columns in C however, compared to algorithm 2 it remains less in catch and is less optimized since the second loop changes the rows instead of the first loop. Matrix A has temporal locality but it is column majored and possibly is less optimized than matrix A in algorithm 2 (it hard to compare the two). Matrix B is row major order but is really hard to compare it to algorithm 2. It appears that each row of B will remain longer in catch since the rows are defined in the first loop whereas in A they are in second and therefore it is more optimized in algorithm 3.

power is work/energy over time :)

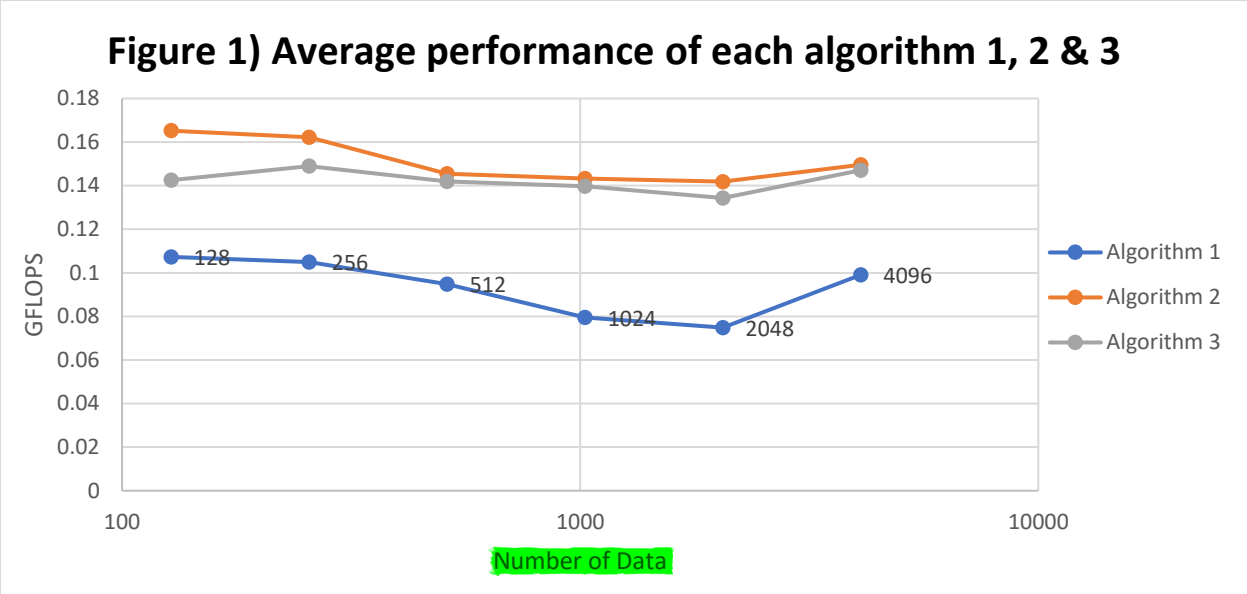
In conclusion, the optimization power of C is like $C_1 > C_2 > C_3$, for A is like: $A_2 > A_1 > A_3$ and for B is like: $B_3 > B_2 > B_1$. Based on my interpretation I would Assume algorithm 1 and maybe 2 would be the most optimized one but its just a hypothesis and it should be tested many times to find the solution. Nonetheless, it strongly depends on the choice of cpu and the size of catch therefore, any of these three might be the most optimized one according to the size of iterations, operations and catch.

picked an algorithm: 1 pts
discussion : 2 pts
merit: 1 pts

##

(b) (8 pts) Implement all three algorithms and time them for a range of n values spanning from 128 to 2048 (remember, we like nice round numbers, so 128, 256, 512, 1024 and 2048 is not a bad choice) and plot the performance of each algorithm (in FLOPS - floating point operations per second) against n for all three algorithms together. If you want, you can further increase the n to see what happens for even larger problems.

Algorithms 1, 2 and 3 in file matrix.c



The results of comparison of performance of each algorithm in terms of GFLOPS indicated that Algorithm 2 has the highest performance in amongst the three as expected from previous discussions in part (a). However, Algorithm 1 unexpectedly had the worst performance and Algorithm 3 performance was nearly identical to Algorithm 2 for larger number of data. The performance of all three algorithms were relatively low and as the number of data increases the performance drops extremely. This is probably due to the triple nested loop that makes the innermost operation the most expensive. Memory access delays are solely responsible and it is found that the latency affects performance of the algorithm. This effect can be mitigated by efficient programming. As result the less cache miss we get the more optimized our algorithm is.

too strong statement, they are close but one is clearly better

"size of the matrix", data is uncountable

As result, Algorithms 2 and 3 are spatially optimized where one achieves temporal locality of cache for A, and spatial optimization for B and C.

implementation: 4 pts
runscript: 1 pts
plots: 2 pts
discussion: 1pts

##

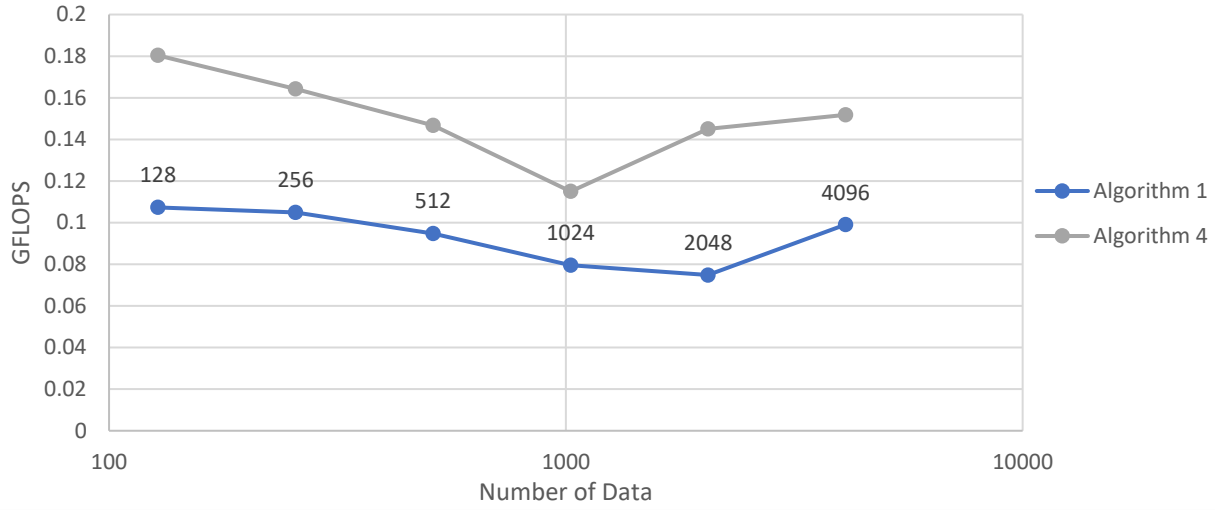
(c) (2 pts) In your chosen algorithm, implement the cache reuse technique (remember Tip 1?). Time your improvement for the range of n values and compare (on a plot) with the unoptimized algorithm. How did the change affect the performance?

Algorithm 4 in file matrix.c

As shown in figure 2. the cache reuse technique greatly enhances algorithm performance especially for fewer number of data. However, as the number of operations increase the boost in performance deteriorates just a bit but there is still a significant difference in terms of boost in performance when cache reuse is implemented. A drop is also observed in middle as we approach 10244 and 2048 but then the performance increased again. This is most likely due to the fact that maybe these size of matrixes A, B and C produces a lot of cache miss and therefore it takes a more times to take data from memory and therefore more time for operation in total and hence the reduction of performance as n gets close to 1000.

implementation: 1 pts
discussion: 1 pts

Figure 2) Performance gained through cache reuse technique



######################

(d) (3 pts) On top of the cache locality change you introduced in point (c), implement the loop unrolling technique (Tip 2) for your selected algorithm. Time your improvement for the range of n values and compare (on a plot) with the unoptimized algorithm and the resiult from point (c). How did the change affect the performance?

Algorithms 5A, 5B, 5C in file matrix.c

Algorithms 5A while using cache locality it implements loop unrolling technique with 16 structures

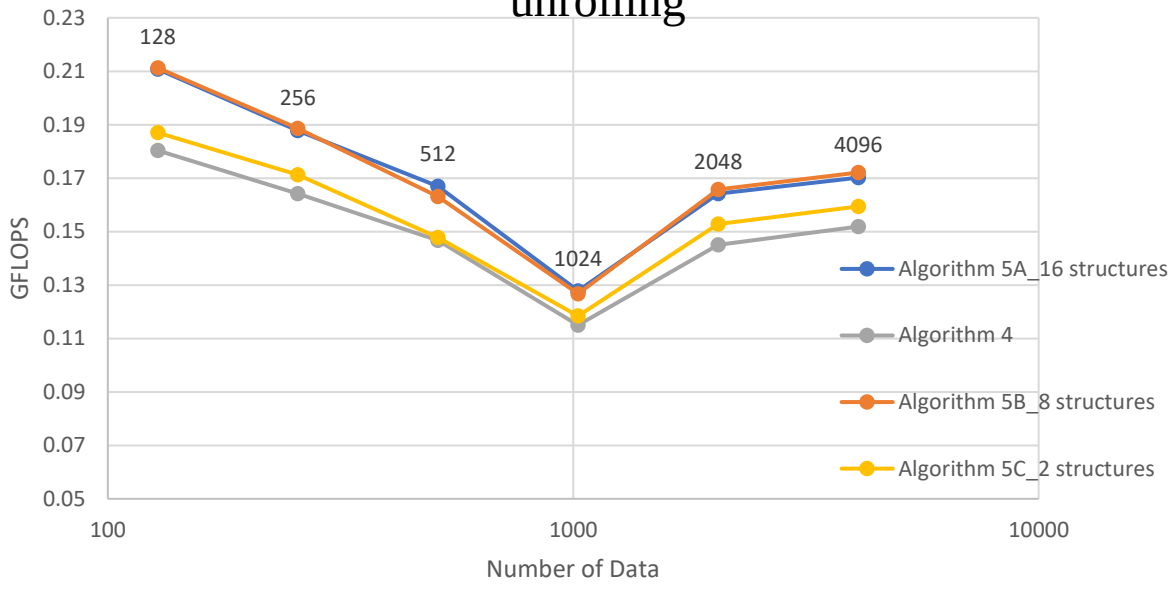
and similarly, 5B with 8 structures

and 5C with 2 structures.

negligible

Implementing loop unrolling technique further helps with performance gain, however this boost is most evident in fewer numbers of data. The difference between using 8 or 16 structures is negatable in terms of gained performance but these two are indeed more effective than 2 structure for this type of problem. Similar problem was observed as we approach number of 1024 we have drop in performance most likely because this number produces more cache misses relative to amount of computations than others.

Figure 3)Performance gained through cache reuse and loop unrolling



implementation+rationale+discussion = 3 pts

######################

(e) (3 pts) On top of the previous changes, implement cache blocking (Tip 3) for the innermost loop. Make sure you choose the block size such that n is always divisible by the block size, and that the block size is divisible by the factor by which you unroll your loops. Again, powers of 2 are a useful guidance here.

Algorithm 6 in file matrix.c

Algorithm 6a this algorithm implements all techniques with cache blocking of block size of 4

Algorithm 6b this algorithm implements all techniques with cache blocking of block size of 16

Algorithm 6c this algorithm implements all techniques with cache blocking of block size of 64

Algorithm 6d this algorithm implements cache reuse and cache blocking of block size of 16

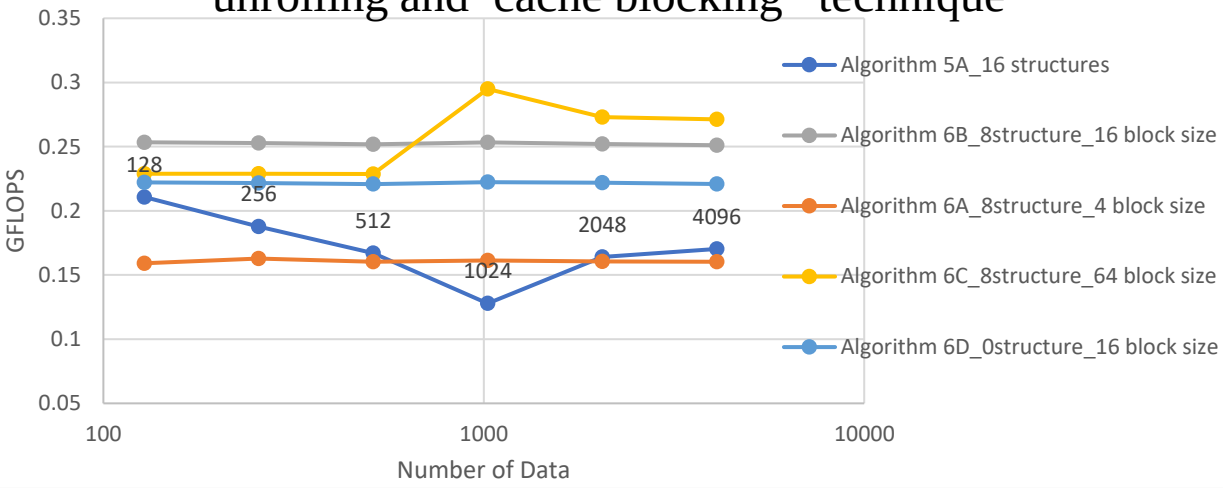
This is the most interesting algorithm since for a block size of 4 (algorithm 6A) the performance reduces compared which means in this case implementing cache blocking with small blocks reduces performance which was expected but interestingly it resolves the cache miss issue at 1024 which means this technique can be used for number of data that do not exactly fit in L cache of CPU to improve the performance.

Increasing the number of blocks to 16 (algorithm 6B) increases the performance and =as it can be seen in figure 4 both 4 and 16 block sizes seem to fit into cache perfectly as the line is straight and very optimum which is ideal.

Further increasing it to 64 (algorithm 6C) at first decreased the performance and this is due smaller size of problem as we get more catch misses with respect to number of data. However, from 1024 and the next points there is an increase in performance compared to 16 blocks, making it more ideal for larger datasets as less cache miss, we have relative to number of operations.

And finally, implementing purely catch blocking (algorithm 6D) with block size of 16 gives less performance when all tips are combined together.

Figure 4) Performance gained through cache reuse, loop unrolling and cache blocking technique



##########

(f) (extra credit) We have assumed so far, that we need to traverse memory by rows or columns, leading to essentially breaking down matrix-matrix multiplication into a series of inner or outer products. What if we divide matrix in blocks, rather than rows and columns. Read the Goto and Geijin (2008) paper referenced below, and propose (and demonstrate) an algorithm which achieves better performance than the best algorithm you have developed in points (a-e).

Algorithm Extra 1 in file matrix.c,

For divide matrix in blocks the following formula has been used:

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{22}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Unfortunately, this method does not give me the same sum of elements of matrixes as the others do, therefore either the proposed formula doesn't work in this case or the algorithm has a bug. Nevertheless, the partitioning of matrix into 4 regions and computing each separately does enhance

the performance but for larger data more blocks are required for higher performance. Turning the matrixes A and B into smaller ones and computing each allows data to stay in CPU Ln caches and therefore computation is done whiteout using the memory at each sub matrix, thus increasing the performance of computation.

##############################

(g) (extra credit) Matrix-matrix product is one of the most important linear algebra operations, widely used throughout scientific computing applications. Naturally, people created very specialized libraries, which can perform this operation in an optimal way. Those libraries are (typically) called BLAS (Basic Linear Algebra Subprograms) and there exists many flavors of it. One BLAS library (GotoBLAS) is references in the paper below, and we have an implementation of OpenBLAS on R1 and R2. If you are curious how does your MXM implementation compare to the state-of-the-art BLAS routines, feel free to time the DGEMM routine and compare results. You can read more about BLAS routines at the links in the references section.

Unfortunately, I was not able to make it work on my code and it was to complex and little time I had for figuring it out. But from literature review I understood that BLAS Implementation is much faster any of the discussed methods. There are numerous articles and different ways of implementing matrix-matrix multiplication and are focused on optimizing the loops and some are available in level 3 functions of BLAS. Besides BLAS and ATLAS there is also BLIS, plus with Intel MKL library and its own way of implementation it does not matter whether you use row or column major storage which is very interesting but requires further reading and coding to get it working.