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) : Brandon Gautama

Student 2 (print name) : Fan Jin

Git Repo link: https://github.com/cse260-wi20/pa1-pa1-bgautama-f1jin

Name of AMI instance (name your instance with your team name as in "pa1-b5chin-tsundara"): pa1-bgautama-f1jin

Checkpoint Submission (above)

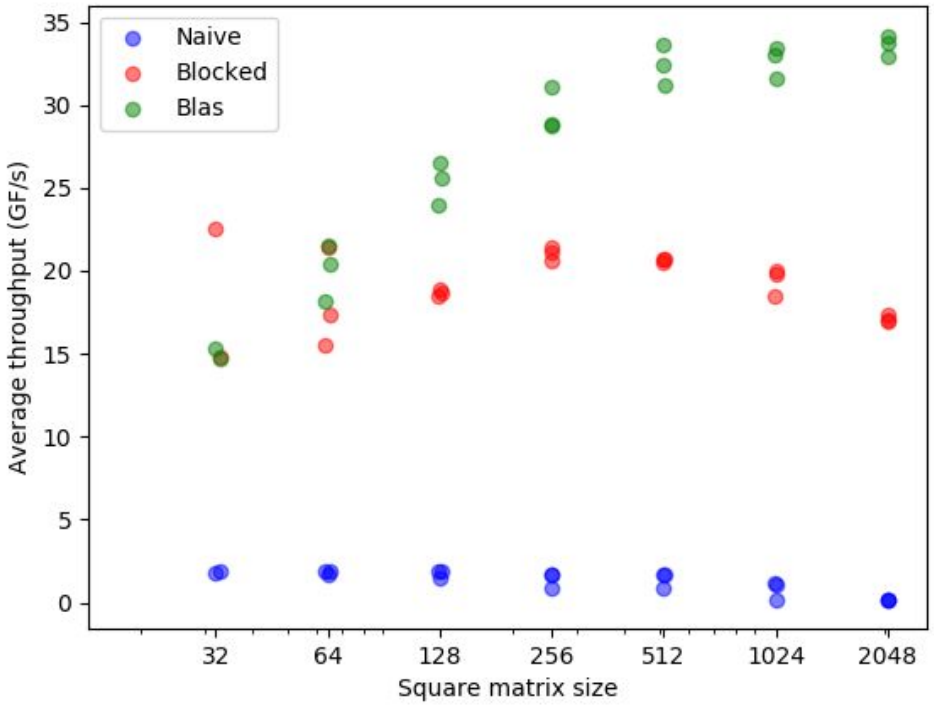
The Analysis for the report should address the following questions based on the optimization you do in the program. You are free to add extra insights regarding the assignment.

Introduction

In this project, we try to optimize matrix multiply routine on one core CPU t2.micro instance by exploiting cache memory hierarchy/layout and SIMD vectorization using AVX2. There are 16 AVX2 registers and they are each 256 bits wide. Since each element of the matrix is a double (64 bits), we can fit 4 doubles in one vector register. benchmark.c is the main driver program that will drive the routine and it will call our optimized code dgemm-blocked.c to do the matrix multiplication. We are also given benchmark-blas and benchmark-naive for us to compare our performance against measured in Gflops/sec.

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

Matrix Size (N)	Naive Code (Gflops/sec)	Optimized Code (Gflops/sec)
124	0.559	20.9
128	0.515	18.3
512	0.309	21.1
513	0.365	20.8
768	0.31	15.5
769	0.249	21.1

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.

We find that performance at matrix sizes that are multiples of 128 is lower (does not scale like other nearby matrix sizes). For example, at sizes 128, 768, the performance dipped compared to other nearby matrix sizes. This is because when we access the block of the next row in the same column, there is cache aliasing (when block size is multiples of 128). It happens when the next block is assigned to the same set in the cache and there are limited number of ways the cache can store. So these conflict misses affects the performance significantly.

However, this is not the case for size 512 (also a multiple of 128). Previously, this performance dip is seen in all matrix sizes that are multiples of 128. However, after we optimized our code by exploiting memory layout, this dip becomes less significant (still worse compared to nearby matrix sizes) and at some sizes such as 512, the performance did not dip at all.

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

N	Peak GF (with padding)	Peak GF (w/o padding)
32	**21.7**	22.2
64	**20.8**	20.7
128	**18.3**	12.8
256	**20.5**	14.5
511	**20.9**	13.8
512	**21.1**	14.3
513	**20.8**	15.2
1023	**19.6**	14.1
1024	**19.6**	14.4
1025	**20.3**	14
2047	**17.7**	13.1
2048	**17.7**	13.2

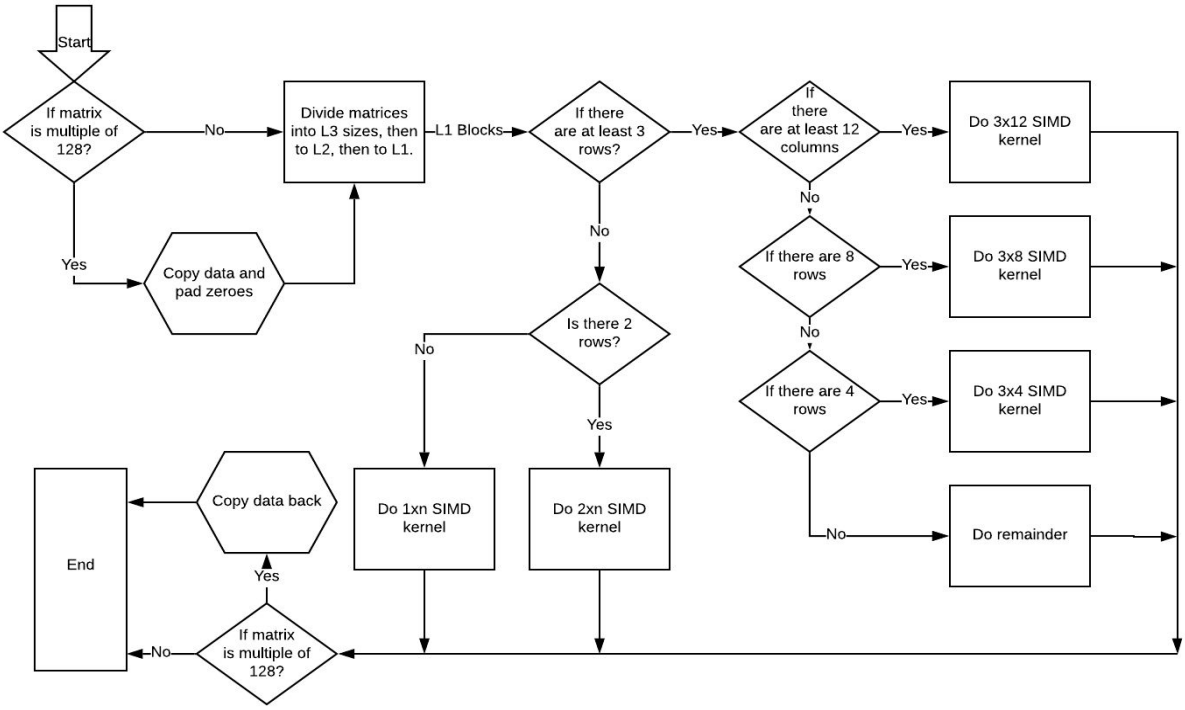
Note: We include 2 columns for performance with and without padding to show the effects of padding. But our final optimized code is the one with padding in the middle column.

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.

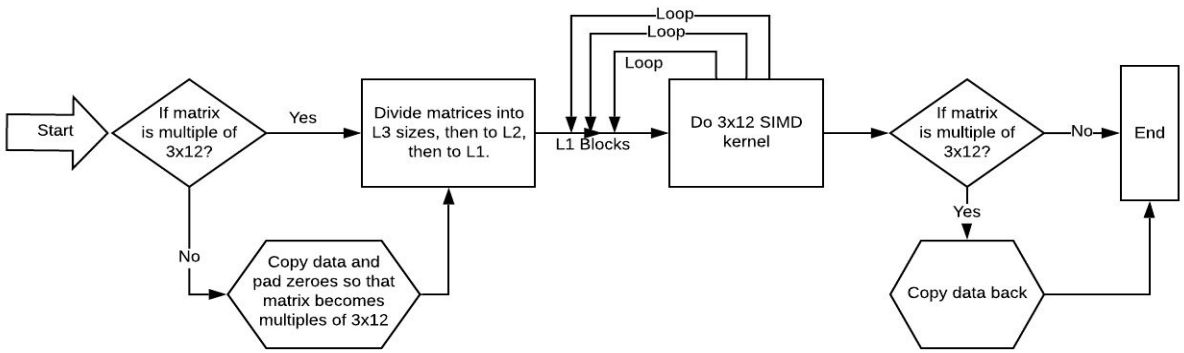
The following chart shows the general approach before we remove the branch switch statements as our last optimization. We think it is useful to include it here because this is the general approach of our optimized code.



As shown in the flow chart above, we first determine if the matrix size is a multiple of 128 and pad it with zeroes if it is. This is done to reduce cache aliasing. Then we do multilevel cache blocking with tuned sizes from autotuning. In the innermost kernel, we use SIMD of different sizes according to the matrix left in the L1 cache. We first try if the leftover size is greater than 3x12, 3x8, 3x4, 2x12, 2x8, etc. In this way, we can make use of SIMD as much as possible.

As will be described below in the optimization steps, we then removed the branch switch statements which improves and stabilizes our code performance. In other words, we now pad it with zeroes so that it becomes multiples of 3x12 and we do not need to

check which size is left in the L1 cache. The following chart shows the program flow of our final optimized code:



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.

The following are the steps we took in optimizing the performance (naive code is 2 GFlops/sec):

1. Multilevel Cache Blocking (2.2 GFlops/sec): There is a speedup of 1.1x over the naive code. The reason that there is not much speedup here is because we are still naively computing the innermost kernel.

2. Matrix Stride (Copy the L1 memory into contiguous layout to prevent cache aliasing, 1.8 GFlops/sec): The performance worsen slightly because we were not reusing the malloc'ed memory and there is a memory copy overhead.

3. SIMD 1x4 kernel (5.2 GFlops/sec): The performance increases by almost 2.5x compared to just multilevel cache blocking. The reason is because AVX can operate on 4 doubles in one instruction. At the same time, we are using register tiling. Combining SIMD, register tiling and the effects of multilevel cache blocking, we were able to gain a significant performance.

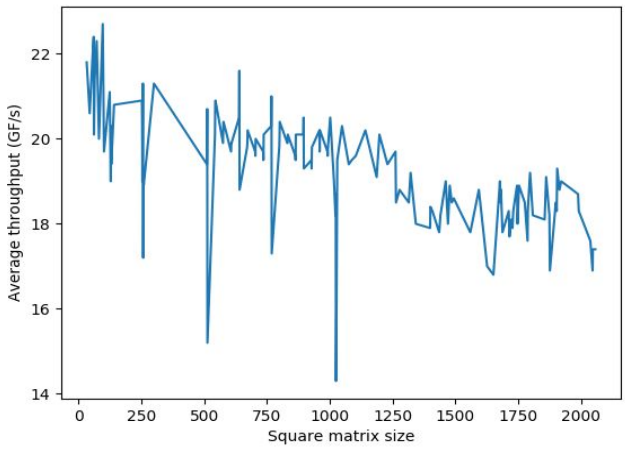
4. SIMD Loop Unroll (Row): We tried these kernel sizes: 2x4 (8 Gf/sec), 3x4 (10Gf/sec), 4x4 (12 Gf/sec), 5x4 (14 Gf/sec), 6x4 (15Gf/sec), 7x4 (16 Gf/sec), 8x4 (15 Gf/sec). As you can see, while we were unrolling the rows, the performance kept on increasing until 7x4 kernel and decreases at 8x4 kernel. The increase is because as we unroll the loop, we increase the performance with less jump/branch instructions and there can be more chances for out of order execution. The 16 registers for AVX can be maximally utilized.

The decrease for 8x4 kernel is because we are using more than the available 16 registers.

5. SIMD Loop Unroll (Column): So we tried nx8 sizes and nx12 sizes. Initially when we increase n, the performance increases. We still observe the above decrease in performance as we increase n to use more than 16 registers. The most optimal performance we reach is 3x12 kernel giving 17 GFlops/sec. The slight increase in performance from using more columns (nx4 to nx12) is because we are accessing more elements per row since the matrix is stored in row major order.

6. Pad Zeroes and Memory Alignment: The above performances are the average values for most matrix sizes. However, the performance dipped significantly on sizes that are multiples of 128 (powers of 2) such as 512, 768, 832, 1024. The reason is because since the matrix size is of multiples of 128, there is cache aliasing when we access the next block of matrix B which is one row below. To solve this, we try to pad zeroes on those matrix sizes. This slightly increases the performance on those specific matrix sizes but it worsen the performance on other matrix sizes. The reason is because there is overhead for copying memory. So we decided to use padding only on those specific sizes. Memory alignment did not help performance because the way we organize the block sizes into the different cache levels has taken alignment into consideration.

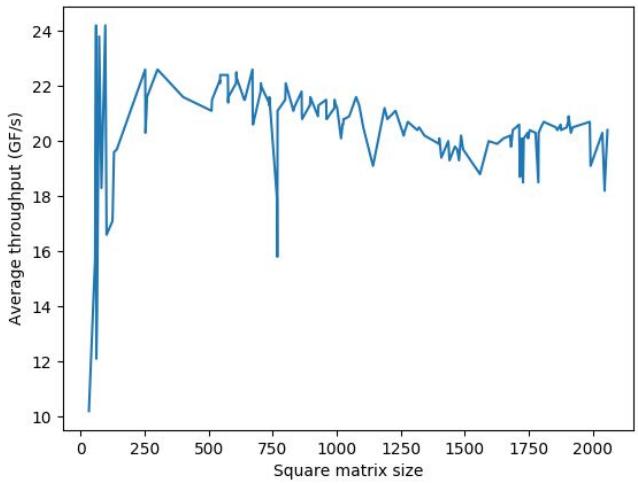
7. Nonsquare sizes and autotune (20 GFlops/sec): Then we tried to use non square block sizes for the three cache levels for maximal use of cache line size. The performance increased from 18 GFlops/sec to 20 GFlops/sec because by using nonsquare block sizes, we were able to ensure that the blocksize is a close multiple of the simd kernel, which we cant ensure of if we are using square blocks. The following graph is the performance we obtained:



As you can see, there are significant performance dips at sizes that are multiples of 128. We will address this issue below.

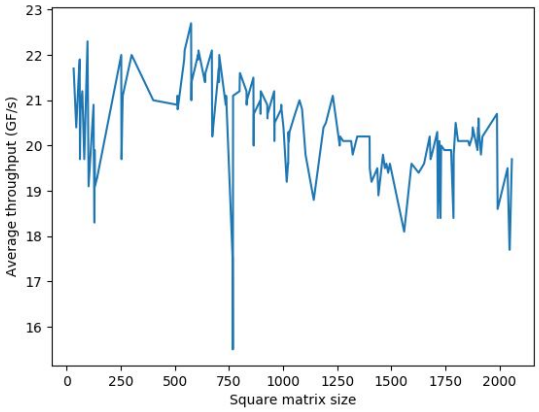
8. Copying layout to contiguous memory to avoid cache aliasing: In this case, we try to reuse malloc'ed memory instead of creating new on each iteration. However, the performance did not increase because there is too much copying overhead involved.

9. Branch Removal (22 GFlops/sec): After all the previous iterations, we still have performance dips on matrix sizes that are multiples of 128, which we suspected to be due to cache aliasing. The way we wrote the innermost kernel before is that we will try the largest SIMD kernel (3x12, 3x8, 3x4, 3x8, 2x8) and go smaller and smaller. Also, if there is less than 4 columns left, we do matrix multiply naively. The way we did this is by using switch statement. In this last optimization, we try to remove this switch statement and to do this, we have to make sure that the matrix size is a multiple of innermost kernel 3x12. For matrix sizes that are not multiples of 3x12, we pad them with zeroes. With this optimization, the performance increase to 22 GFlops/sec and it is very stable across all the values. The reason may be because of less overhead by removing jump/branch statements and we do not have to handle the small cases (less than 4 columns) by doing naive computation. In this case, preparing the matrix with memory copying overhead is not significant compared to the performance gained. The following shows the performance obtained:



As you can see, there is no more performance dips and it is very stable across all values, except small matrix sizes (we address this below).

10. Special case for smaller matrices: The above achieved stable values and have good performance on matrix sizes greater than 130. This is because for smaller matrix sizes, the overhead to copy decreases the performance. So we handle smaller matrices as a special case by keeping the branch statements. And if the size is greater than 130, then we use the last branch optimization above. This allows us to achieve high performance for all matrix sizes. The right figure shows the performance we obtained on our final code.



Q3c. supporting data - e.g. analysis of cache behavior, 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.

(1) Cache Aliasing: We attribute the performance drop at certain sizes to cache aliasing, based on two facts. First, the cache sets are indexed by 2, which means two adjacent rows in a matrix could alias in the cache if the size (lda) is a power of 2. Second, we used valgrind tool to test L1 data cache miss rate. When the size is a multiple of power of two, the cache miss rate increases significantly compared to nearby sizes.

D1 cache miss rate (valgrind tool): valgrind --tool=cachegrind ./benchmark-blocked

Size 2048 = 10%

Size 2038 = 3.3%

Size 2058 = 3.4%

Size 2149 = 2.5%

(2) Blocksize Tuning and Cache Capacity: We determined our block size from the parametric search and it confirmed that the optimal block size should be close to but not bigger than the cache size (obtained from lscpu). It is interesting that the optimal sizes only use about 80% of cache capacity, because we reserve the remaining 20% capacity for potential cache aliasing entries.

Note: we do not include graph for autotuning as we are using non square block sizes and each cache level will have 3 dimensions. That gives a total of 9 dimensions which is hard to visualize.

We are calculating matrix C = A x B, with sizes as follows: [M x N] = [M x K] x [K x N]

Cache	Capacity (Bytes)	M Dimension	N Dimension	K Dimension	Total Used (Bytes)
L1	32K	30	36	32	25,536
L2	256K	60	108	96	180,864
L3	30720K	720	1296	1152	26,044,416

(3) Maximize utilization of AVX registers: Another reason why 3x12 works best is because it utilizes 15 registers whereas other sizes are using less than that (given there are 16 available registers). In this way, we reduce jump/branch instructions that take more cycle and we also made more out of order executions possible, therefore reducing the number of idle instructions.

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

1. Transpose matrix B in the order of AVX loads (zigzag). Only copy matrix B. No need to copy A or C, reduces overhead.

2. Use branch prediction profiler to verify if misprediction is an overhead.

3. Study the cache more carefully and profile the cache as we try to optimize to make full use of the cache.

(Optional) Any additional insight or optimizations that you tried implementing and how it affected the performance.

We find that our performance exceeds that of BLAS for matrix size N < 50. We also have similar performance to BLAS for matrix size 50 < N < 200. It is only for greater matrix sizes that BLAS have a significantly better performance.

Q4 References

- Jim Demmel On "Designing fast linear algebra kernels in the presence of memory hierarchies" Jim Demmel's on-line reader on BLAS kernels. Look at Algorithm 3: Square blocked matrix multiply. https://people.eecs.berkeley.edu/~demmel/cs267/lecture02.html
- Jim Demmel Lecture slides https://people.eecs.berkeley.edu/~demmel/cs267_Spr12/Lectures/lecture02_memhier_jwd2012.ppt
- https://software.intel.com/sites/landingpage/IntrinsicsGuide/