CSE 260 Project 1 Report

Shaowei Liu, Shuhao Chang
Department of Computer Science and Engineering
University of California, San Diego
shl044@ucsd.edu, s2chang@ucsd.edu

2020.1.30

1 Information

Student 1: Shuhao Chang A53306667

Student 2: Shaowei Liu A53301774

Git Repo link: https://github.com/cse260-wi20/pa1-pa1-s2chang-shl044

Name of AMI instance: cse260-pa1-s2chang-shl044

2 Results

A performance study for a few values (about 20 different values) of N from 32 to 2048 on our optimized code - data is in the file data.txt.

2.1 Speedup over naive code for interesting points

We compare our optimized code against naive version over matrix size 32, 64, 128, 256, 512, 1024, 2048. As can be seen in Table 1, our optimized code could achieve 6.3, 9.5, 7.9, 19.1, 21.0, 130.7 and 129.3 times faster than naive one respectively. Since our optimized code can better leverage the cache hierarchy and register for computation, thus achieves higher GFlops with the increase of matrix size, while the performance of naive version drops significantly.

2.2 Irregularities in data

Normally, the matrix multiplication performance increases with the matrix size. However, different implementations will have irregular effects with some particular matrix sizes. In this section, we mainly focus on the performance of our best implementation. As shown in Figure 1, the Gflops of size 63, 64, 65 are much higher than 31, 32, 33 and 127, 128, 129. This is related to the setting of L1, L2, L3 block sizes. In this implementation, the L1 block size is 36×36, L2 block size is 108×108. For matrix sizes of 31, 32 and 33, they have to be padded to 36×36 for later computation. However, since the matrix is too small, such padding, buffering operations consumes more proportional of time than larger matrices, which leads to a bad performance. For matrix sizes of 127, 128 and 129, they cannot fit into L1 cache and part of the matrix is stored in L2 cache, which increases the data access time. Therefore, matrices of 63, 64 and 65 performs better than the others.

However, for some values in between there would either be sudden dip or spike in the plot. You could include those points in the report and explain it behaves like that. It's up to your discretion to choose the points.

Matrix Sizes	Naive	Optimized
32	1.74	10.9
64	1.65	15.6
128	1.68	13.3
256	0.87	16.6
512	0.87	18.3
1024	0.15	19.6
2048	0.14	18.1

Table 1: Speed up over naive code for insteresting points

2.3 Show performance for the following numbers

The performance can be found in Table 2.

Matrix Sizes	Gflops
32	10.9
64	15.6
128	13.3
256	16.6
511	18.3
512	18.3
513	18.0
1023	20.0
1024	19.6
1025	19.4
2047	18.3
2048	18.1

Table 2: Gflops measurement with different matrix sizes

3 Analysis

3.1 How does the program work

We first introduce the runtime environment and basic principles in our matrix multiplication problem. Then we introduce the principle of each module used in our program. More detailed discussions can be seen in Section 3.2.

3.1.1 Runtime Environment

- CPU Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
- Memory 32 KB L1 cache, 256 KB L2 cache, 30720 KB L3 cache, 16 256 bits register
- OS Ubuntu 16.04.2 LTS, GCC 5.4.0

3.1.2 Multiplication Algorithm

We take the inner product of matrices as our basic multiply algorithm. Given two matrix A ($m \times k$) and B ($k \times n$), the multiplication result is matrix C ($m \times n$), which can be obtained as follows in equation 1.

$$C(i,j) = \sum_{t=i}^{k} A(i,t) \cdot B(t,j) \tag{1}$$

The runtime complexity for matrix multiplication is $O(n^3)$. The detailed algorithm for square matrix multiplication can be seen in Alg 1.

Algorithm 1 Matrix Multiplication Algorithm

```
for i ∈ [1, n] do
    for j ∈ [1, n] do
        for k ∈ [1, n] do
            C(i, j) += A(i, k) · B(k, j);
        end for
    end for
end for
```

3.1.3 Blocked Matrix Multiplication

In order to use faster memory and increase the cache locality, we can divide matrix A, B, C into $N \times N$ subblocks (block size $b = n/N$) and let all three blocks fit into the cache. Then we perform the matrix multiplications on subblocks. The detailed algorithm can be seen in Alg 2.

Algorithm 2 Blocked Matrix Multiplication Algorithm

```
Divide A, B, C into N × N sub blocks (block size b);
for i ∈ [1, N] do
    for j ∈ [1, N] do
        for k ∈ [1, N] do
            C(i, j) += A(i, k) · B(k, j) (b × b matrix multiply);
        end for
    end for
end for
```

3.1.4 Cache Hierarchy

As there are 3 level cache block on the experiment machine, we can further utilize cache hierarchy to accelerate computation. We first design cache blocking in the L3 cache, which is the closest cache to the main memory. Then we use similar strategies to conduct cache blocking for L2 cache and L1 cache. At each level, we must make sure that all three blocks of A, B and C must fit into the cache. Thus, assuming the block size is b, the cache size is M KB, each matrix element is 8 byte double precision, then the block size b must satisfy the following equation 2. More discussion about cache strategy can be found in paper [1].

$$b \leq \sqrt{\frac{M \times 1024}{3 \times 8}} \tag{2}$$

In order to make loading and writing data faster, we can further open buffers in the memory to store each sub blocks and also align the data to remove cache line crossing. In this way, if we want to load 128-bit into cache at each time, then we can get those aligned data in one access.

3.1.5 Vectorization

We can further optimize our codes with SIMD (Single instruction multiple data). There are totally sixteen 256bits TMM data registers in the AVX2 architecture. Thus it can operate 4 doubles in a single instruction each time, which may bring possible 4x speed up. When we do the block matrix multiplication, we can unroll the for loop and load, compute and store in parallel. We use 3×12 AVX kernel size as an example to show how to parallelize our codes. The 3×12 AVX kernel vectorization are shown in Alg 3. Every load instruction can load 4 double type matrix elements, inside the algorithm, we use broadcast and fmadd AVX

function to do parallel computation. The detailed introduction of each function can be seen at https://software.intel.com/sites/landingpage/IntrinsicsGuide/ and paper [2].

Algorithm 3 Vectorized Block Matrix Multiplication Algorithm

for $i = 1, i \leq b, i+ = 3$ **do**
  for $j = 1, j \leq b, j+ = 12$ **do**
    Load $c00\_c01\_c02\_c03 = \&C[i, j]$;
    Load $c04\_c05\_c06\_c07 = \&C[i, j] + 4$;
    Load $c08\_c09\_c00\_c01 = \&C[i, j] + 8$;
    Load $c10\_c11\_c12\_c13 = \&C[i + 1, j]$;
    Load $c14\_c15\_c16\_c17 = \&C[i + 1, j] + 4$;
    Load $c19\_c19\_c10\_c11 = \&C[i + 1, j] + 8$;
    Load $c20\_c21\_c22\_c23 = \&C[i + 2, j]$;
    Load $c24\_c25\_c26\_c27 = \&C[i + 2, j] + 4$;
    Load $c28\_c29\_c20\_c21 = \&C[i + 2, j] + 8$;
    for $k = 1, k \leq b, k+ = 1$ **do**
      Load $a0x = broadcast(\&A[i, k])$;
      Load $a1x = broadcast(\&A[i + 1, k])$;
      Load $a2x = broadcast(\&A[i + 2, k])$;
      Load $b0123 = \&B[k, j]$;
      Load $b4567 = \&B[k, j] + 4$;
      Load $8901 = \&B[k, j] + 8$;
      $c00\_c01\_c02\_c03+ = fmadd(a0x * b0123)$;
      $c10\_c11\_c12\_c13+ = fmadd(a1x * b0123)$;
      $c20\_c21\_c22\_c23+ = fmadd(a2x * b0123)$;
      $c04\_c05\_c06\_c07+ = fmadd(a0x * b4567)$;
      $c14\_c15\_c16\_c17+ = fmadd(a1x * b4567)$;
      $c24\_c25\_c26\_c27+ = fmadd(a2x * b4567)$;
      $c08\_c09\_c00\_c01+ = fmadd(a0x * b8901)$;
      $c18\_c19\_c10\_c11+ = fmadd(a1x * b8901)$;
      $c28\_c29\_c20\_c21+ = fmadd(a2x * b8901)$;
    end for
    Store $c00\_c01\_c02\_c03 > \&C[i, j]$;
    Store $c04\_c05\_c06\_c07 > \&C[i, j] + 4$;
    Store $c08\_c09\_c00\_c01 > \&C[i, j] + 8$;
    Store $c10\_c11\_c12\_c13 > \&C[i + 1, j]$;
    Store $c14\_c15\_c16\_c17 > \&C[i + 1, j] + 4$;
    Store $c19\_c19\_c10\_c11 > \&C[i + 1, j] + 8$;
    Store $c20\_c21\_c22\_c23 > \&C[i + 2, j]$;
    Store $c24\_c25\_c26\_c27 > \&C[i + 2, j] + 4$;
    Store $c28\_c29\_c20\_c21 > \&C[i + 2, j] + 8$;
  end for
end for

3.2 Development process

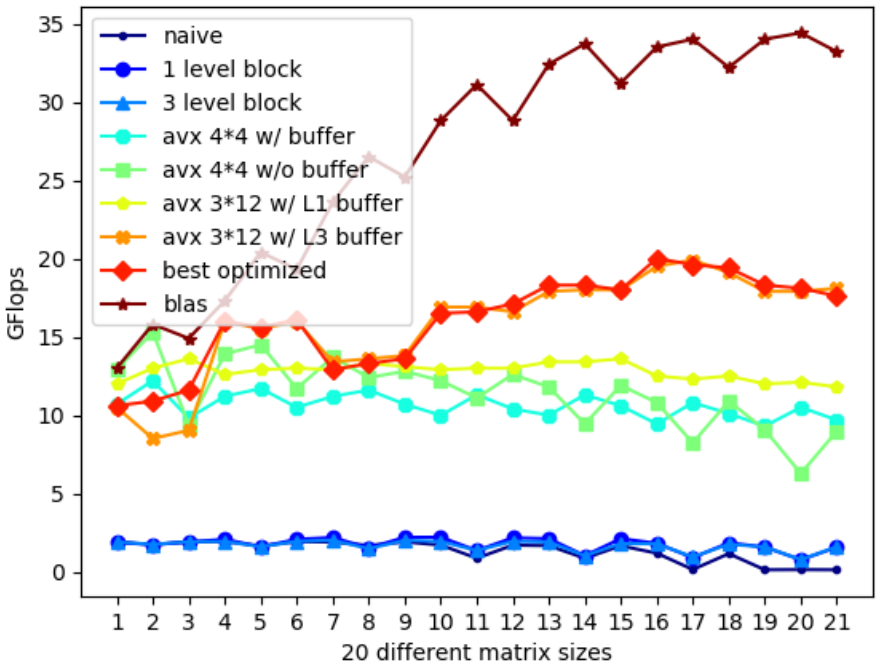

Figure 1: Incremental optimizations of matrix multiplication

We design multiple strategies to accelerate computation and optimize our codes step by step. We incrementally introduce 8 implementations as shown in Figure 1. The x axis is 20 different matrix size as listed in file data.txt.

Baseline 1,2,3 Our first baseline is the naive implementation as illustrated in equation 1. The second one is by using 1 level cache block with block matrix multiplication. The third one is to utilize 3 level blocking. As can be seen from the result, by leveraging cache hierarchy, we can slightly improve the performance against the naive version, especially when the matrix size is larger than 512.

Baseline 4,5 Next, we vectorize our codes by using AVX instruction set and tiling for registers. For baseline 4, we design a 4*4 matmul with AVX2 intrinsics. Inside the block, we use fused-multiply-add function to combine multiplication and addition in a singe instruction. To achieve the goal of utilizing 4 values at a time, we further allocate a buffer initialized with 0 for corner cases in inner-most loop. Then we find out that by explicitly dealing the fringe cases independently and padding with 0 in for loops, we can further improve the speed. This would be the baseline 5 of avx 4*4 matmul without using buffer.

Baseline 6,7 To better utilize the 16 registers in matrix multiplication, we further do some calculations of the AVX kernel size, the detail discussion can be found in Section 3.2.1 and Section 3.2.4. Finally we decide to use 3*12 as the AVX kernel size. Since our configuration is non-square, we have to bring back buffers for submatrix padding. We design two baseline models, one is to allocate a buffer at L1 cache, the other is to allocate the buffer at L3 cache. Allocating buffer at L1 cache can reduce the buffer size, but would also increase the memory copy cost. After experimenting

with both cases, we decide to allocate the buffer at L3 level. This two options would baseline 6 and 7.

Best optimized Finally, we add more optimization tricks like using restrict pointer to remove false dependencies, using prefetch instructions to load data into cache more efficiently and also tuning the cache size parameters. The final result is the implementation 8 marked as "best optimized".

3.2.1 Non-square Configurations

Since loading data from registers and taking operations is much faster than doing so with data in memory, it is efficient to make use of as many registers as possible when doing multiplication. There are totally sixteen 256bits TMM data registers in the AVX2 architecture. Therefore, we compute the theoretical best solution for register usage with different non-square configurations. This is related to how we design the matrix dimensions when doing the basic multiplication.

We assume loop unrolling in not used in the basic matrix multiplication. The dimensions of matrix A, B, and C are $M \times K$, $K \times N$, $M \times N$ as shown in Figure 2 below. We take the inner product to implement the matrix multiplication.

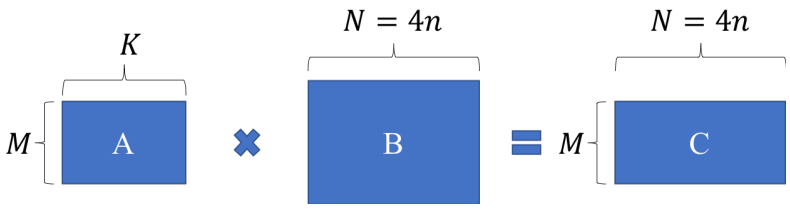

Figure 2: Basic matrix inner product dimensions

Since we are going to use the load instruction of AVX2, we first pad the matrix width to be the multiple of 4, which corresponds to $N = 4n$. To do the basic matrix inner product, we first load elements of C into registers. This step costs $M \times n$ registers. In the inner loop of K, we load one element of A per row and broadcast it as 256bits, which costs M registers. Besides, each time in the loop of K, we load one row of B, which costs n registers. Therefore, the total number of registers we are going to use is $M \times n + M + n$. Next, we solve the inequation for register number and try to find integer solutions.

$$f(M, n) = Mn + M + n \tag{3}$$

$$\arg\min_{M,n} f(M, n), \quad f(M, n) \leq 16 \tag{4}$$

Through brute force method, we find the three optimal solution for this optimization problem as shown in Table 3. According to the results, we can at most make full use of 15 registers with a 1×28, 3×12 or 7×4 basic matrix size. Another consideration is that we expect to do as many multiply-add operations as possible in one loop of K, which means the least loop times. In each loop, the number of multiply-add operations is $M \times n$. Therefore, we finally choose $M = 3, n = 3, N = 12$ as the matrix dimensions of a basic inner product.

M	n	$N = 4n$	$f(M, n)$
1	7	28	15
3	3	12	15
7	1	4	15

Table 3: Optimal integer solutions for the inequation of M and n

3.2.2 Matrix Padding

The AVX2 instructions require that we have to load or store 4 values each time. If the input matrix size cannot be exact divided by 4, we have to use a branch instruction to judge and detect the edge and adopt correlated solutions. In order to optimize this step, an efficient way is to pad the matrix with zeros so as to be exactly divided by 4. Doing so will not have impact on the accuracy of results. Here, one problem is where to do the padding operation. If we do padding at the beginning immediately after input, the advantage is we only allocate a large part of memory and store values in it only once and following blocked matrix multiplications all benefit from this. If we do padding at the lowest level (L1 block), the advantage is that we can obtain better spatial correlation when traverse from one row to the next row. In other words, we may find the next value from the same cache instead of a long distance away.

However, through practice we find that to pad the matrix at the beginning may performs better. Here are two reasons. First, we save the memory setting time by storing the whole matrix into a larger buffer. The lowest level multiplication do not need to set the memory as zeros each time it is traversed. Second, padding the matrix at the lowest level still loses spatial correlation when fetching data from the original matrix to the buffer, which do not bring time efficiency.

3.2.3 Restrict Pointer

Restrict pointer is a new feature of the new C language standard. It is used to announce that only one pointer in the program is used to point to the data. By doing so, the compiler will not generate other additional pointers and might improve the access speed of data. We also try to add this indicator to our program. However, this does not have obvious benefits on the performance. Since the major time consumption of matrix multiplication is not data access through pointers, we believe that restrict pointers do not help in this case.

3.2.4 Loop Unrolling

When traversing cases with a loop function, there is a branch instruction to be executed each time. It is known that executing branch instruction consumes more time than normal computation instructions. Therefore, it may save time to reduce the branch instructions by loop unrolling. The most efficient loop unrolling is done in the L1 block since it affects more traversing cases during the iterative call by L2 and L3 blocks. Since the L1 block computation has used almost all the AVX2 registers, we may consider change the non-square configurations to make use of all 16 registers with loop unrolling. Here, we further explore the situation of Section 3.2.1 by adding the consideration of loop unrolling.

Now we are going to unroll the loop of K by replacing the incremental step from 1 to p, which means we do p times of the same computation in one loop. Therefore, there are $(M + n) \times p$ registers used in each loop. The total registers we use becomes:

$$f(M, n, p) = Mn + (M + n) \times p = Mn + Mp + np \tag{5}$$

Therefore, the optimization problem becomes to find the integer solution for:

$$\arg\min_{M,n,p} f(M, n, p), \quad f(M, n, p) \leq 16 \tag{6}$$

Through brute force method, we find the three optimal solution for this new optimization problem as shown in Table **??**. According to the results, we can at most make full use of 16 registers with a 2×8 matrix size with 3 times loop unrolling, 2×12 matrix size with 2 times loop unrolling or 3×8 matrix size with 2 times loop unrolling. Now, the number of operations in each loop becomes $M \times n \times p$, which is the same among these three cases.

It seems that we not only make use of 16 registers, but also reduce branch instructions with loop unrolling. However, the practical experiments shows that this setting performs worse than the

M	n	p	$N = 4n$	$f(M, n, p)$
2	2	3	8	16
2	3	2	12	16
3	2	2	8	16

Table 4: Optimal integer solutions for the inequation of M, n and p

original 3×12 basic matrix size. Our explanation for this is that actual multiply-add instruction number in each loop decreases. Although there are 12 instructions in a loop now, it is the result of 3 times loop unrolling. The actual multiply-add operations in a loop is 4 which is far less than 9. In other words, by forcing the program to use 16 registers we sacrifices the size of matrix we are able to compute each time. Therefore, this is not a good implementation with loop unrolling.

3.3 Supporting Data

3.3.1 Cache Behaviour

We use **cachegrind** to detect the cache behaviour of different implementations. We choose the matrix size as 512 in the following experiments. Table 5 shows the result. According the L1 data cache missing rate results, we can conclude several interesting points. First, implementations that do not use AVX2 vectorizations all have high cache missing rates. This is reasonable since we even do not carefully make use of cache. Second, storing data into buffer does reduce the cache missing rate but may or may not improve the Gflops. Third, add buffer into L1 block instead of L3 block further reduce missing rate but may or may not improve the Gflops. Finally, we see that our best optimized implementation still has a relatively high cache missing rate. This enlighten us try to further reduce the cache missing rate and improve the performance in the future.

Implementation	**Read Miss Rate**	**Write Miss Rate**
naive	48.2%	9.3%
1 level block	47.9%	51.1%
3 level block	48.8%	54.9%
avx 4*4 w/ buf	3.5%	0.3%
avx 4*4 w/o buf	18.5%	20.7%
avx 3*12 w/ L1 buf	6.7%	8.6%
avx 3*12 w/ L3 buf	18.5%	4.1%
best optimized	18.0%	4.7%
blas	12.6%	4.4%

Table 5: L1 data cache miss rate with different implementations

3.3.2 Block Size Tuning

According to the L1, L2, L3 cache sizes, we are able to calculate the sizes of three levels of blocks, which are 36, 104, 1144 respectively. However, these numbers cannot fit our basic matrix sizes perfectly. According to previous analysis and experiments, we finally choose the width of a basic matrix as 12. Therefore, the L1, L2 and L3 block sizes should also be set as the multiple of 12. Doing so also saves the time of the edge condition judgement. We manually tried several combinations of multiple of 12 and finally set the three levels of block sizes as 36, 108 and 1140. These numbers gives the current best performance. When the L1 block size is smaller than 36, one block cannot full fill the L1 cache and therefore causes a waste of cache space. When the L1 block size is larger than that, the missing rate of data cache increases and Gflops decrease.

3.4 Future Work

According to the theoretical analysis and practical experiments, we summarize the performance of each optimization method and conclude the future directions for further improvement. First, the usage of registers greatly improve the computation speed. However, AVX2 only has 16 256bits registers which limits the performance. We plan to adopt AVX-512 instruction set which allows 32 ZMM 512bits registers in the future. Second, similar to Section 3.2.1, we are going to explore the theoretical optimal solutions for non-square configurations with more registers able to use. Finally, we may further explore the method to replace the use of buffer to pad the matrix and reduce the cache missing rate. By normally computing the intact part of matrix and specially consider the edge multiplication may prevent storing data into buffers, which is believed to further improve the performance.

References

[1] P. Gepner, V. Gamayunov, D. L. Fraser, E. Houdard, L. Sauge, D. Declat, and M. Dubois. Evaluation of dgemm implementation on intel xeon phi coprocessor. *JCP*, 9(7):1566–1571, 2014.

[2] H. Jeong, S. Kim, W. Lee, and S.-H. Myung. Performance of sse and avx instruction sets. *arXiv preprint arXiv:1211.0820*, 2012.