CSE 260: Project 1
Winter 2018

Kunal Kashilkar, Debosmit Majumder

Contents

1 Introduction

In recent times, matrix multiplication has seen various useful applications in Machine Learning and Artificial Intelligence, including but not limited to data mining and analytics. In a world where huge amounts of data are being generated that need to be processed, matrix multiplication deals with ever-increasing computational complexity. Thus, there is a lot of literature that talks about various useful optimizations applied to this fundamental operation[2, 5]

The project involves improving the matrix multiplication performance in a single threaded program. In the report we first present the results of various optimizations that we have applied. The next section analyzes the speed-up obtained due to each optimization and also talks about techniques that did not lead to significant improvements.

The program has been written in C and the test environment provided to us is the CSME program's Bang Cluster. This cluster consists of 35 Dell PowerEdge 1950 servers, which are interconnected with an infiniband switch. Each compute server has two quad-core Intel Xeon E5345 ("Clovertown") 2.33GHz CPUs, for a total of 8 cores per compute node, and 16GB of RAM.

We use optimization techniques such as two-level cache blocking, register tiling, aligning the matrices, etc.

2 Implementation and Analysis

2.1 Compiler Options

The Bang Cluster uses GCC version 4.8.4 (GCC). When compiling the code we can use various compiler flags to enable optimizations. These flags optimize the code at the expense of increasing compilation time and the size of the compiled code.[6][3] We use the following compiler options -

Listing 1: Compiler Options

```
gcc -O4 -mfpmath=sse -mno-align-double -march=core2
```

The following figure compares matrix multiplication performance with and without the compiler options.

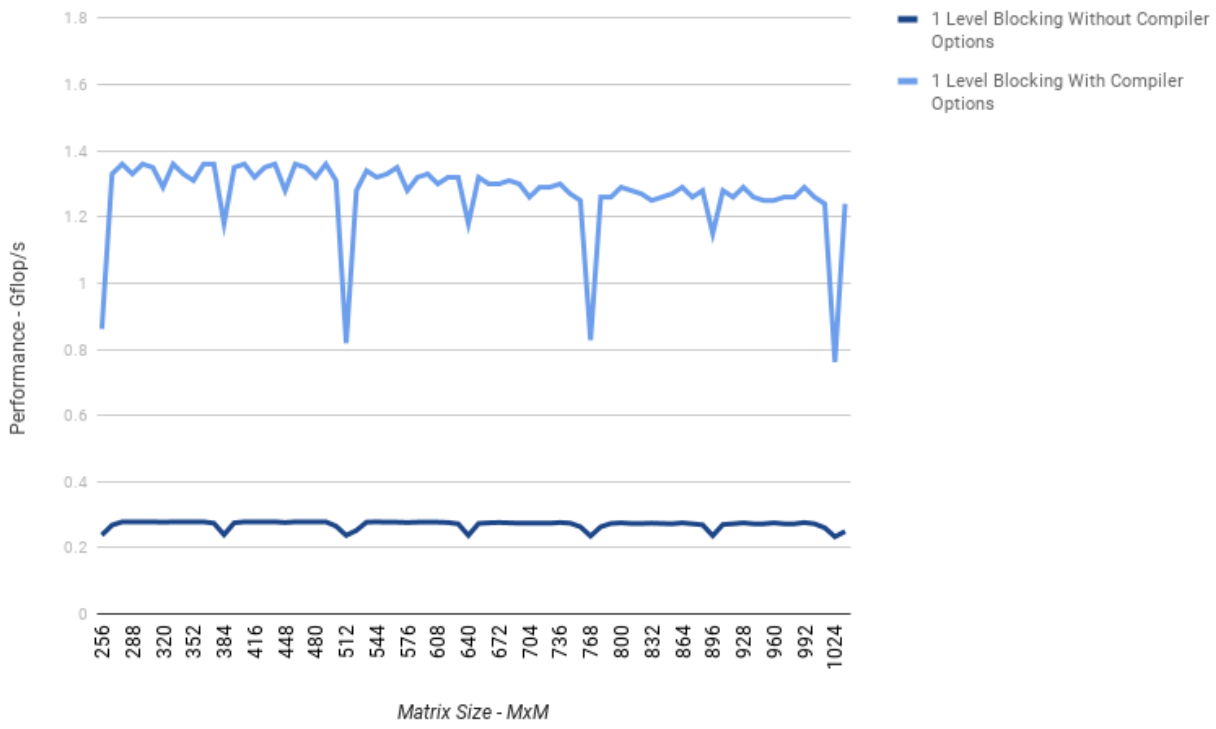

Figure 1: Compiler Options Performance

All the following sections contain measurements using these compiler options.

2.2 Cache Blocking

The memory in modern systems is arranged in a hierarchical manner to offset speed vs. cost trade-off. Thus these systems have several levels of memory. Each server in the Bang Cluster has two levels of cache. The L1 Cache is 32KB in size and is 8-way set associative and two shared L2 caches that are 4MB in size each and 16-way set associative.[4] Accessing data in the cache takes 2-15 cycles whereas accessing the data in memory can take around 200 cycles. Therefore modern processors employ fetching techniques that improve access to data that has temporal or spatial locality.

Matrix multiplication is an operation that can benefit from the spatial locality of data. Therefore, performing arithmetic operations on the data that has already been fetched will lead to better overall performance. But, given the small size of L1 and L2 Cache, we cannot fit the entire matrix in the

cache. A matrix containing double precision elements will be (M*N*8) bytes in size, where M is the number of rows of the matrix and N is the number of columns.
Assuming that the three matrices involved in matrix multiplication are square, the maximum size of each matrix that can fit in L1 cache is around 37x37, while the L2 cache can store three matrices of dimension 416x416. Thus, for matrices that are larger in size, blocking is employed to improve matrix multiplication performance and make the operation independent of the matrix size.

Blocking is a technique that divides the matrices into smaller sized blocks and matrix multiplication is performed on these blocks[5]

We first implemented blocking that divided the matrices into smaller sized square matrices that could fit in L2 Cache. These smaller blocks were further divided into square blocks that could fit in L1 Cache.

The Block sizes that we used were 416x416 for the first level of blocking and 32x32 for the second level of blocking. The following figure compares the performance of blocked matrix multiplication with the naive implementation.

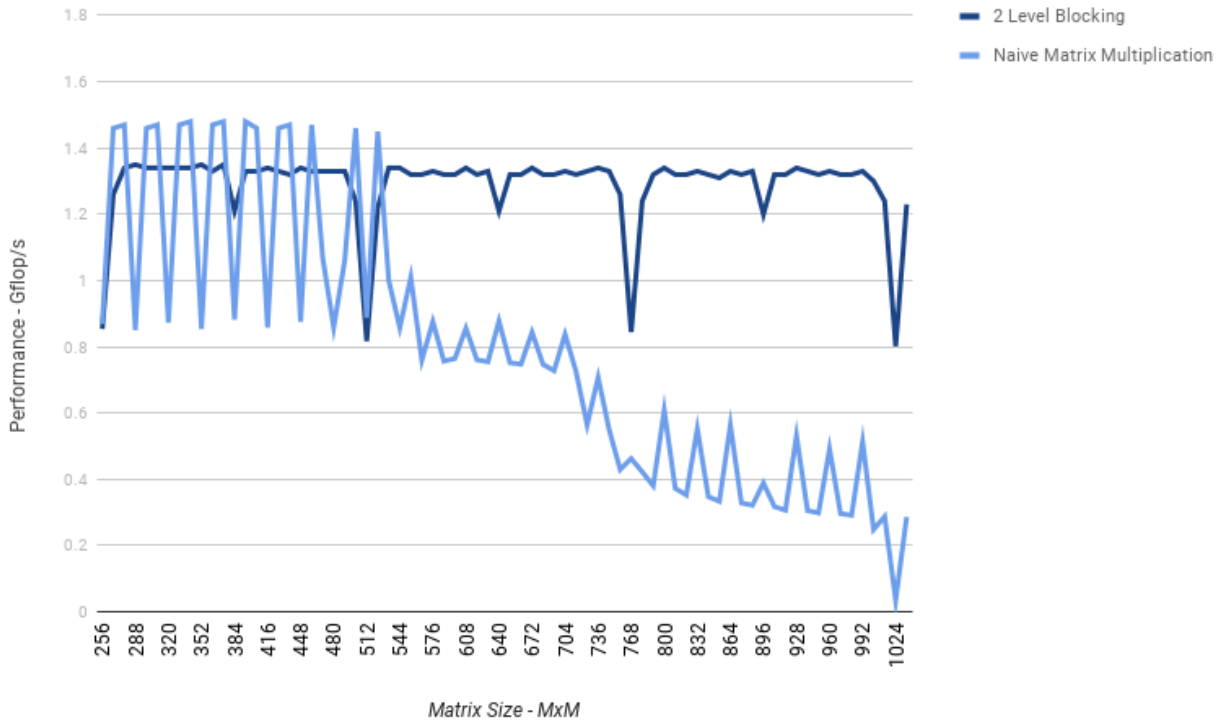

Figure 2: Matrix Multiplication Performance with 2 Levels of Cache Blocking

As seen in the figure, blocking makes the performance independent of the matrix size and provides significant improvement for large matrix sizes.

2.3 Vectorization

Flynn's Taxonomy classifies a set of instructions called SIMD (Single-Instruction-Multiple-Data). These instructions involves performing the same operations on multiple data points simultaneously. We use Intel's SSE/SSE2 and AVX registers and instructions corresponding to them[7]. Here, we discuss about SSE register implementation on Bang and Section 4 talks about AVX implementation on Amazon AWS.

Bang provides sixteen 128 bit SSE2 registers where each register can be used to store 2 double-precision numbers. This method is called Register Tiling. Since matrix multiplication involves the

same values being added to a large number of data points, this method tries to make maximal use of values present in registers. SSE instructions operate on the loaded data points in a single operation.

2.3.1 Register Tiling(2x2)

In this case, we load 4 data points and performing addition and multiplication operations on them at the same time. Instead of calculating the matrix product per element, we performed 2x2 register tiling. The expression in the matrix to be multiplied looks like the one given below

$$\begin{aligned} C_{00} += A_{00} \times B_{00} \quad + \quad A_{01} \times B_{10} \\ C_{10} += A_{10} \times B_{00} \quad + \quad A_{11} \times B_{10} \\ C_{01} += A_{00} \times B_{01} \quad + \quad A_{01} \times B_{11} \\ C_{11} += A_{10} \times B_{01} \quad + \quad A_{11} \times B_{11} \end{aligned} \tag{1}$$

C_{00} and C_{01} are loaded into one 128-bit register. B_{00} and B_{01} are loaded into another register. While, A_{00} is duplicated into one register. Appropriate SSE instructions like _mm_mul_pd _mm_add_pd are used to perform addition and multiplication operations.
The improvement was lesser than expected, which could be due to the unaligned load and store operations that had to be performed to populate the SSE registers with the required values. Also, with 2x2 register tiling we do not fully utilize the 16 SSE register available on the Bang Cluster.

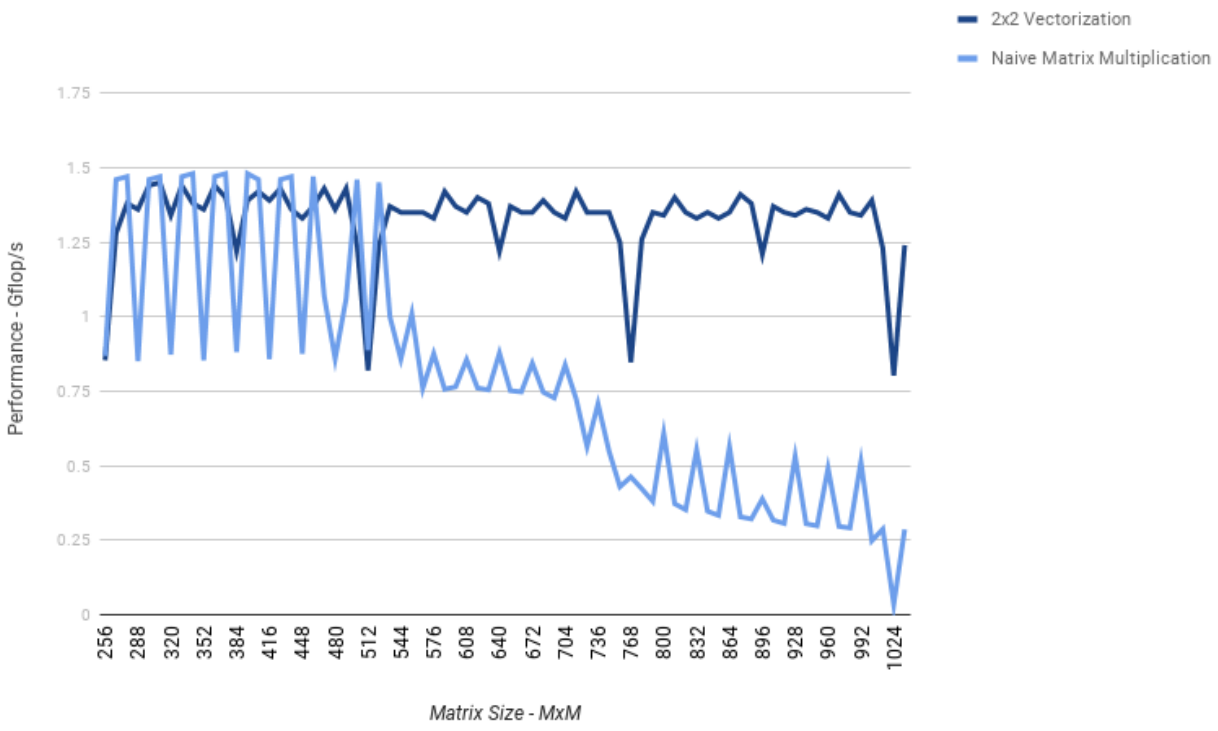

Figure 3: Matrix Multiplication Performance with 2x2 Vectorization

We expected that using a bigger tile size will show an improvement in performance.

2.3.2 Register Tiling(4x4)

We increased the tile size to 4x4 and observed an improved performance over a smaller kernel size. This was because now we can unroll the loops by incrementing each row and column by 4. Thus, we are performing matrix multiplication for 16 elements in each iteration using SSE registers. Increasing the tile size decreases the number of iterations in the 1st level block. Since, we are loading more values

of A and B into the registers, we are exploring spatial locality to a greater extent in a particular cache line.

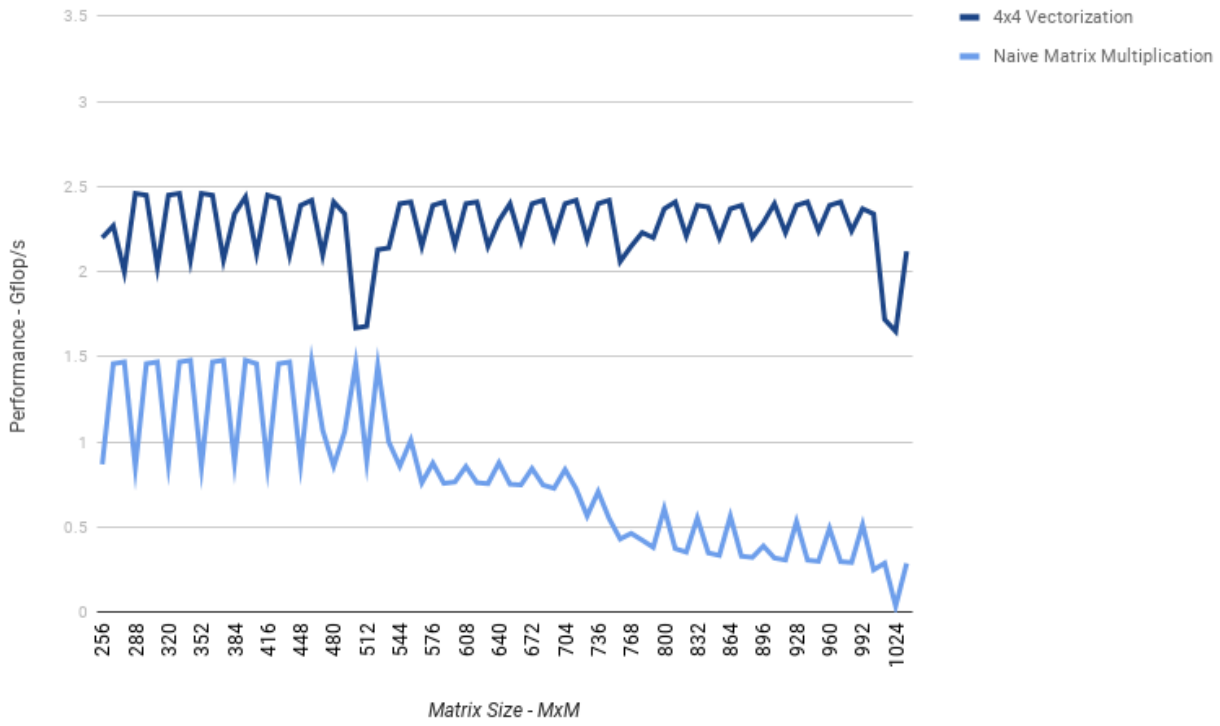

Figure 4: Matrix Multiplication Performance with 4x4 Vectorization

Important Note: Initially, we were doing unaligned loads while using SSE instructions. This was because the addresses were not aligned to 64 byte cache line boundary. Thus, multiple cache lines were accessed to access one contiguous block of 64 byte data. This resulted in lower utilization of SSE registers since we were using _mm_loadu_pd and _mm_storeu_pd. Thus, we used padding of matrices with 0s to align the addresses to the cache line and we could use _mm_load_pd and _mm_store_pd. Padding of matrices and its analysis are discussed in Section 3.4.

2.4 Matrix Alignment

As discussed in the previous section, aligned loads and stores are expected to improve the performance significantly. Our initial attempt used dynamically allocated buffers that were aligned on 64 byte boundaries. We used the following method to allocate aligned buffers.

Listing 2: posix_memalign

```
posix_memalign((void **)&A_padded, 64, size*size*sizeof(double));
```

For matrices that were even sized, we allocated three aligned buffers of the same size and copied the values to these buffers.
For odd sized matrices, we allocated buffers of size = (matrix_size + 1)x(matrix_size + 1) and zero padded the last rows and columns.
We then modified vectorization to use _mm_load_pd and _mm_store_pd.

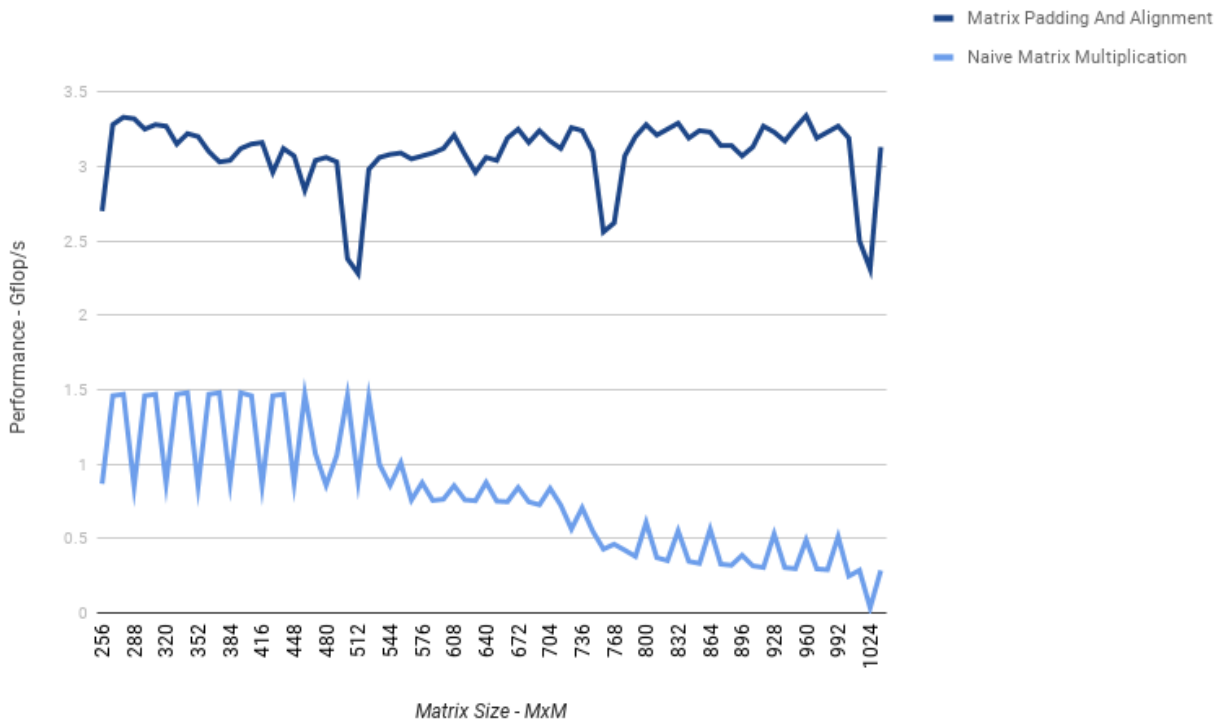

Figure 5: Matrix Multiplication Performance with Matrix Alignment and Padding

Dynamic memory allocation is expected to add an overhead as the processor will have to first find contiguous blocks of the required size and then allocate those. Freeing the allocated blocks will also lead to an overhead. Therefore we used static allocation of matrices and the *memcpy* function for copying data between the input matrices and the padded matrices. The improvements due to this are discussed in the next section.

2.5 Memory Allocation

We optimized the allocation of padded matrices by using static allocation.We also optimized the copying between these matrices by using

Listing 3: posix_memalign

```
memcpy(A_padded, A, M*M*sizeof(double));
```

where M is the size of the block of matrix A currently being padded.

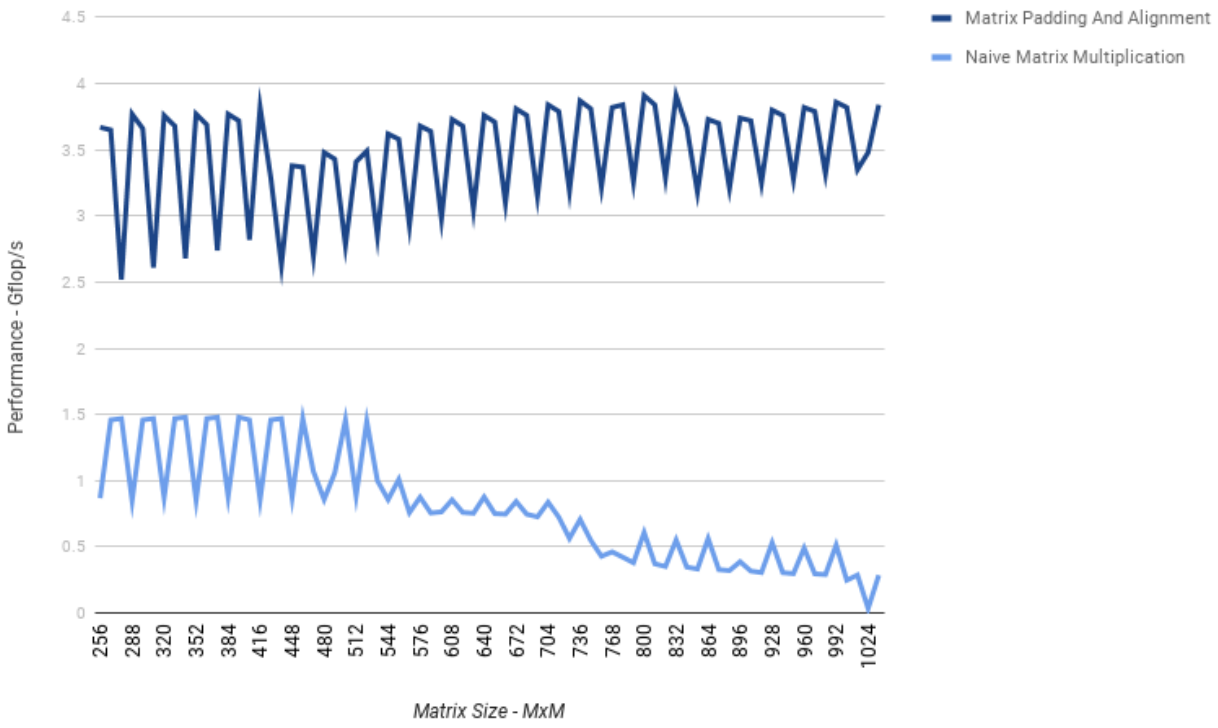

Figure 6: Matrix Multiplication Performance with Static Memory Allocation and Memcpy

The values for which we observed lower performance were odd-sized matrices and some even-sized matrices. This could be due to the naive matrix multiplication required for some rows/columns of the matrices.

2.6 Restrict Pointers and Prefetching

We used restrict pointers throughout our implementations for optimization. This means only the pointer itself or the value directly derived from it will be used to access the object to which it points to. Thus it limits the effect of pointer aliasing.

We are also using $\_\_builtin\_prefetch$ options to pre-fetch those cache lines which will be used in the next iteration. This ensures that the values that are used in the next iteration are already fetched with the next cache line. This led to slight improvement in the performance.

2.7 Rectangular Block Sizes

While implementing 2-level blocking sizes was easier, it was observed that changing the 2nd-level (L2) cache block into a rectangular dimension achieved higher performance than using bigger square block tiles. This is due to lesser number of cache evictions due to aliasing in the cache. For odd sizes of the matrix, we convert odd values of blocks to even sized blocks so that vectorization is performed for all the corner cases.

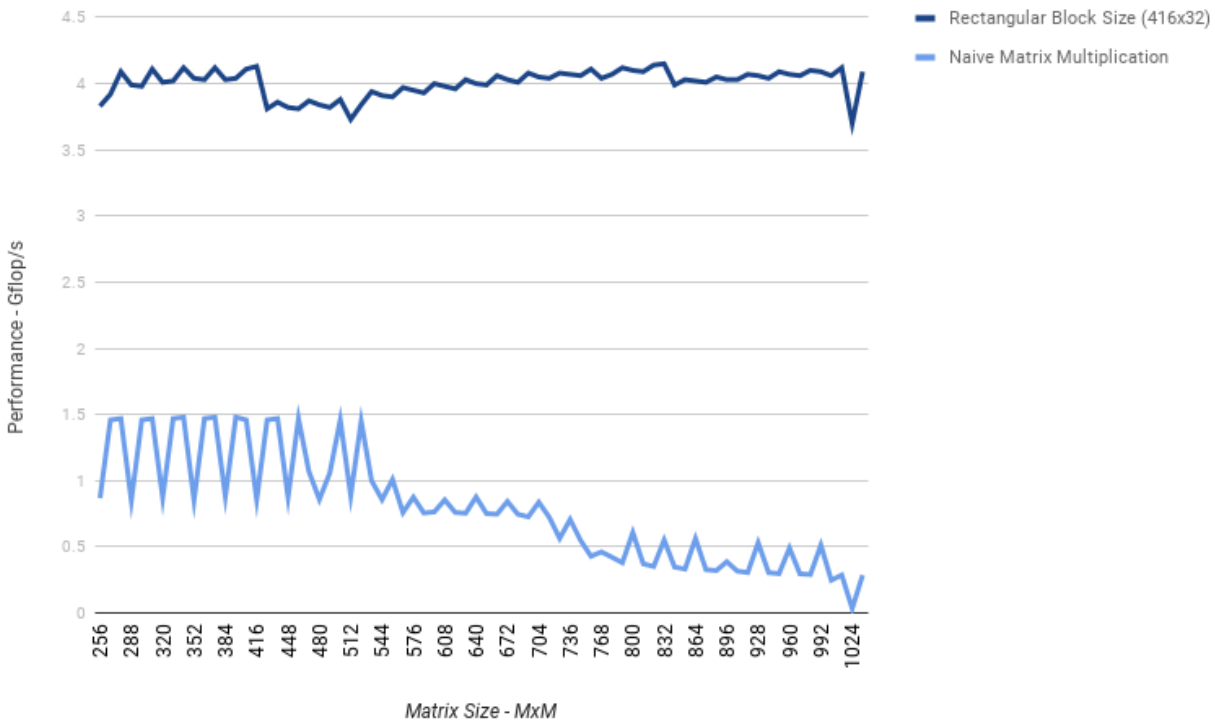

Figure 7: Matrix Multiplication Performance with Rectangular Block sizes

3 Results

In this section we present the results of our optimizations. The pseudo code provides a high level overview of the optimizations that we implemented.

Listing 4: Psuedo Code

```
//Implementation of L2 blocking
void square_dgemm(lda, A, B, C){
        transpose(A)

    // i goes from 0 to lda, incrementing by BLOCK_SIZE_2
    for i in (0, lda, BLOCK_SIZE_2){
        for j in (0, lda, BLOCK_SIZE_2){
            //copy contents of C in an aligned matrix C_padded
                for k in (0, lda, BLOCK_SIZE_2_K){
                // copy A, B into padded matrices

                /* call do_block_1 which corresponds to first level
                 * of cache blocking */
            }
            // copy C back from C_padded matrix
        }
    }
    transpose(A)
}
Implementation of L1 blocking
void do_block_1(lda, M, N, K, A, B, C) {
```

```
    for i in (0, M, BLOCK_SIZE){
        for j in (0, N, BLOCK_SIZE) {
                for k in (0, K, BLOCK_SIZE) {
                /* depending on whether A and B can be
                * multiplied using vectorization,
                * call the vectorized_multipy method
                * or call the naive matrix multiplication method
                */
            }
        }
    }
}
```

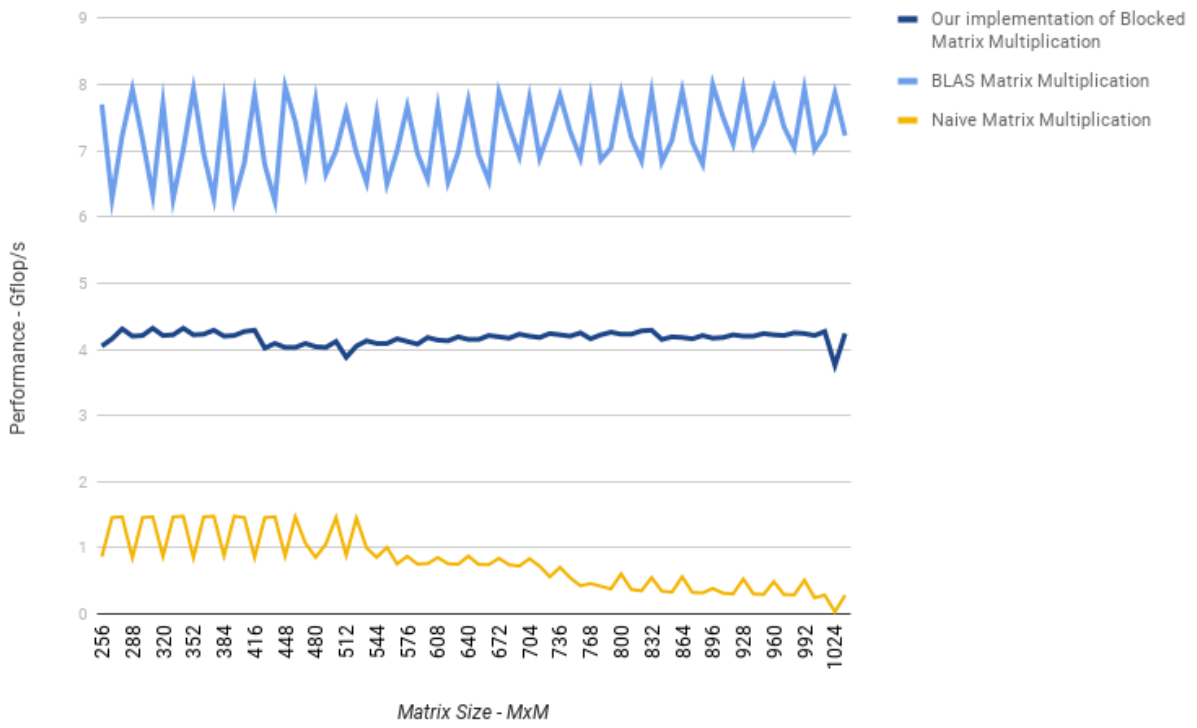

Figure 8: Performance for Naive implementation, our implementation and BLAS implementation

4 AVX Implementation

Advanced Vector Extensions is an extension to SSE and expands most instruction commands to 256-bits[1]. This enables us to load 4 values at the same time and perform operations on 8 data points simultaneously. We are using the instructions _mm256_set1_pd to load the value at A[0] and duplicating/broadcasting it 4 times to store it into one 256-bit register. We are loading 4 double-precision values of B simultaneously thus exploring the spatial locality in the particular cache line. Thus, 4 parallel multiplication and addition operations lead to significant improvement in performance. 4x4 kernel size is used to perform the matrix operations in this case.

Bang offers 16 256-bit AVX registers. It was observed that using AVX registers improves the performance over using SSE instructions. The compiler option that was used was "-mavx".

It was observed that AVX implementation gave approximately double the performance over SSE vectorization.

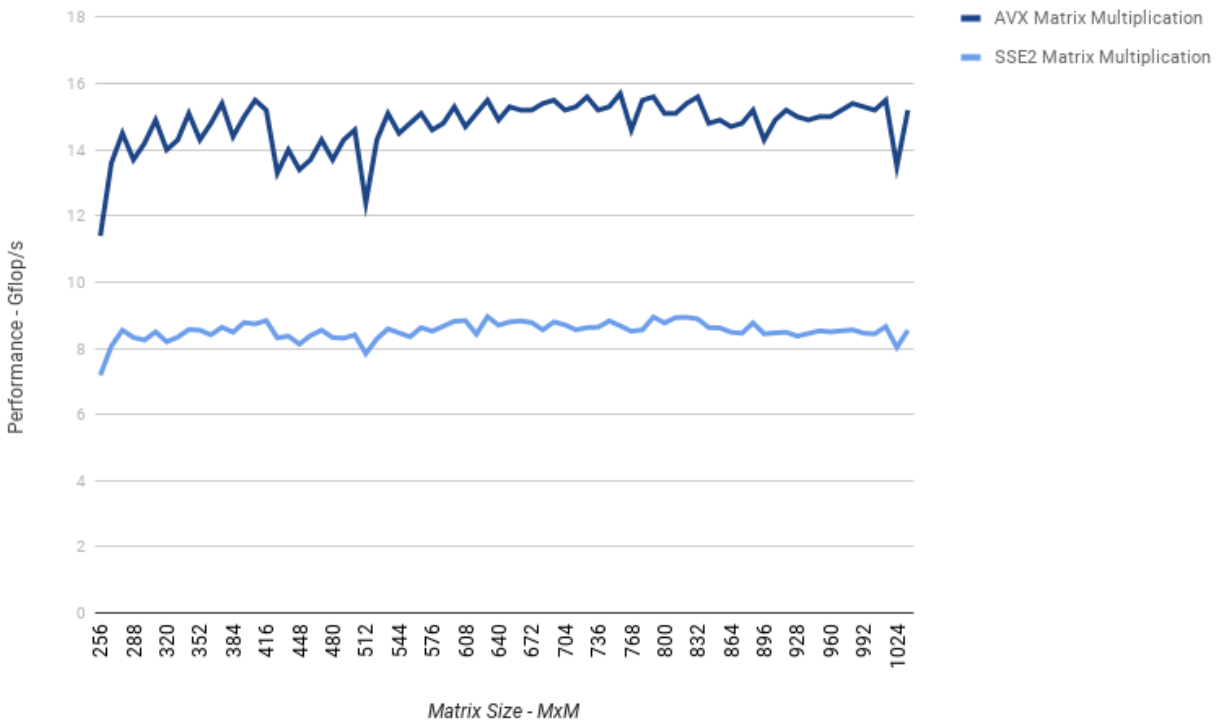

Figure 9: Matrix Multiplication Performance with AVX Registers

5 Conclusion

It was observed that optimizations such as cache blocking, SSE vectorization, data alignment, static memory allocation, etc. gave us performance improvements by 4 times over the naive matrix multiplication. Moreover, after using AVX registers, there were further improvements in performance over SSE vectorization. This goes to show that SIMD instructions with wider registers can explore more parallelism in the code.

6 Future Work

It was observed that for some matrix sizes, the code was not giving consistent performance. For such matrices dynamic block sizes could have led to improved performance.

Checking whether vectorization is using all the 14 registers (that we used) by disassembling the code. This would have allowed for better utilization of the registers and improved SIMD arithmetic performance.

Since, Amazon AWS machine had 3-levels of cache, we could have used 3-level cache-blocking along with our AVX implementation.This would have led to more consistent throughput performance with matrix sizes.

References

[1] *Advacned Vector Extensions*. Tech. rep. San Francisco. URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.

[2] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-performance Matrix Multiplication". In: *ACM Trans. Math. Softw.* 34.3 (May 2008), 12:1–12:25. ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. URL: http://doi.acm.org/10.1145/1356052.1356053.

[3] *Intel 386 and AMD x86-64 Options*. Tech. rep. Santa Clara, CA. URL: https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/i386-and-x86_002d64-Options.html.

[4] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Tech. rep. Santa Clara, CA. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[5] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms". In: *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1991, pp. 63–74.

[6] *Options That Control Optimization*. Tech. rep. Santa Clara, CA. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[7] *SSE Registers and Instructions*. Tech. rep. Santa Clara, CA. URL: https://software.intel.com/sites/default/files/m/a/9/b/7/b/1000-SSE.pdf.