Matrix Multiplication using GotoBLAS

1 Results

1.1 Performance

Square Matrix Size	BLAS (GFlops)	BlisLab (GFlops)	Naive (GFlops)
32	17.655	19.57	2.454
64	23.91	22.75	2.0775
128	28.41	26.38	1.717
256	32.59	26.875	0.8178
511	32.605	21.225	1.6625
512	33.905	26.445	0.857
513	32.13	23.1	1.6745
1023	34.065	23.185	1.124
1024	34.59	26.27	0.1396
1025	33.555	23.97	0.9476
2047	35.025	22.53	NA
2048	34.93	23.555	NA

1.2 Plot

A plot showing the comparison of performance for naive matrix multiplication, BLAS library and our implementation is shown in Figure 1.

2 Analysis

2.1 How does the program work

The program uses the GotoBLAS algorithm [2, 3] to optimize matrix multiplication. Naive matrix multiplication algorithm has a very low FLOP/Byte ratio that limits its maximum attainable performance measured as shown in Figure 2. This is due to the fact that most of its time is spent in retrieving the data from the DRAM rather than performing the actual computations. In simpler terms, there is not enough re-use of the data that is already present in the processor's memory hierarchy. To amortize the cost of accessing the memory, our approach employs a few optimizations that improve the data-reuse in the cache hierarchy, hence increasing the attainable performance. These are discussed below.

First, modern processors employ multiple levels of caches along with DRAM to form a memory hierarchy. To best make use of high-speed caches, it is imperative that the working set of our applications reside in caches and also that cache hit rates are high. Consequently, we split the task of matrix multiplication into multiple small chunks or "macro-kernels", with each chunk further divided into smaller blocks or "micro-kernels" as shown in Figure 3. This is done by keeping in mind the data reuse pattern in performing the matrix multiplication, with the most frequently accessed data kept in the smallest and fastest cache, the next frequently accessed data in the next level cache, and so on. The size of blocks also referred to as "micro-kernels" in the code, are decided to ensure that they fit completely in the CPU registers and the L1

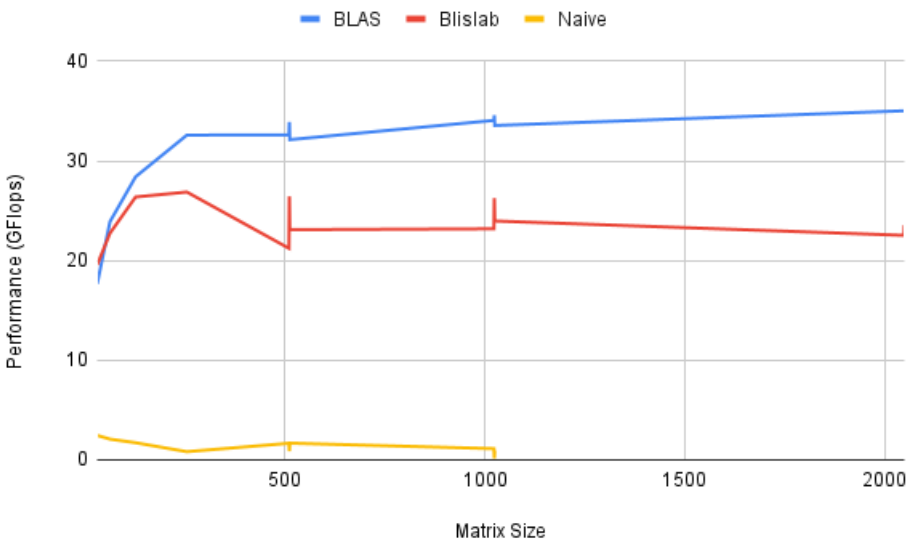

Figure 1: **Performance Comparison**

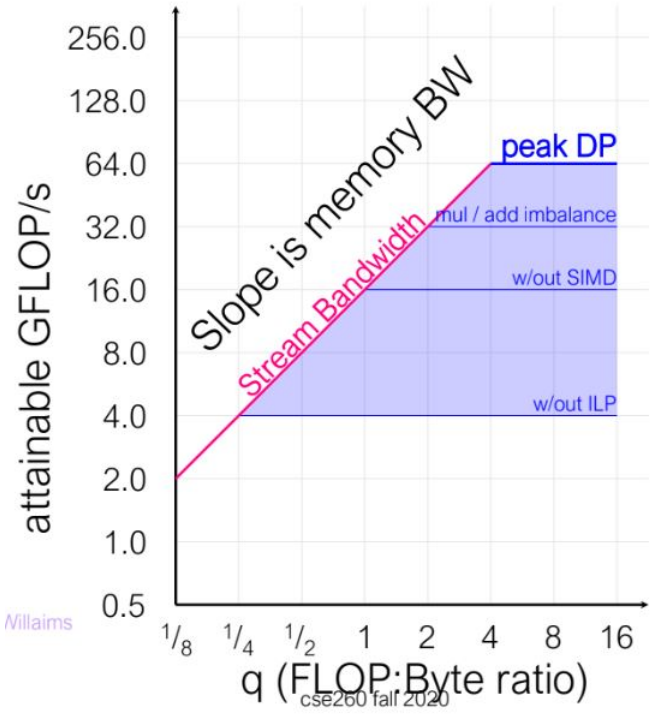

Figure 2: **FLOP Byte Ratio [1]**

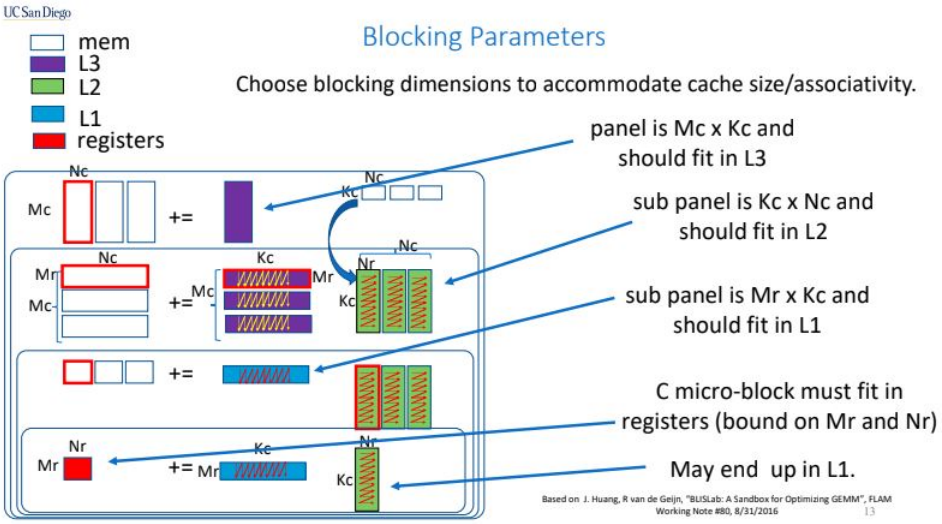

Figure 3: **GotoBlas Technique [1]**

cache, while the dimension of the "macro-kernels" are determined such that they can be completely stored in L2 and Last Level Caches. If properly done this technique enormously improves the FLOP/Byte ratio that has a direct impact on maximum attainable performance. Thus, the final matrix multiplication output is obtained by combining the results of these smaller matrix multiplication operations.

Our second observation is that using the "Vector Outer Product" as opposed to the "Inner Product" allows better data reuse opportunity of the data already present in the cache hierarchy before fetching the next data. However, the outer product traverses elements of our matrices in an order that doesn't effectively utilize the properties of the Cache. Caches exploit the locality of the data in space and time by fetching an entire cache line from the main memory when an address within the cache line is accessed. It does that because there is a high probability of the same data being accessed again (temporal locality) and the data that is near to the one that is currently accessed (spatial locality). However, the "Outer Product" traverses the matrix in strides that are larger than cache line size, thereby not utilizing the benefit provided by the caches. To mitigate this problem, we pack all the data needed for the "Outer Product" being calculated by the micro-kernel in a single contiguous memory array so that all the subsequent accesses to the matrix elements can fit within a single cache line. This way, all the accesses by the micro-kernel can be serviced by the cache-hierarchy improving the FLOP/Byte ratio.

Finally, we make use of SIMD (Single Instruction Multiple Data) vector instructions [4] that allow us to operate on multiple elements at once using the processor's vector processing hardware. SIMD instructions provide the benefit of storing most of the required data in the processor's internal registers, thereby providing very low access time loading and storing these elements. This in turn increases the amount of computation performed per access of the main memory (FLOP/Byte ratio). However, in most of the cases when the input matrix size is not a multiple of the SIMD's matrix processing engine designed, there is not significant performance improvement. One way to tackle these cases is to implement multiple SIMD matrix processing engines of different dimensions based on the available SIMD registers, and send the input matrix for multiplication to the largest processing engine configuration for maximum performance. For the cases where no suitable matrix processing engine's size could be found, we multiplied the matrices by the "Scalar Outer Product". Figure 4 shows how this was performed in our code. The pseudocode is available in Algorithm 1 - the logic for packing matrix B is not shown to conserve space.

2.2 Development process

As mentioned in the PA1 Instructions, we started the project by implementing the packing function. That was the first step towards implementing a high-performance library for matrix multiplication. The packing function was modified to align the data as accessed by the "Outer Product" algorithm for matrix multiplica-

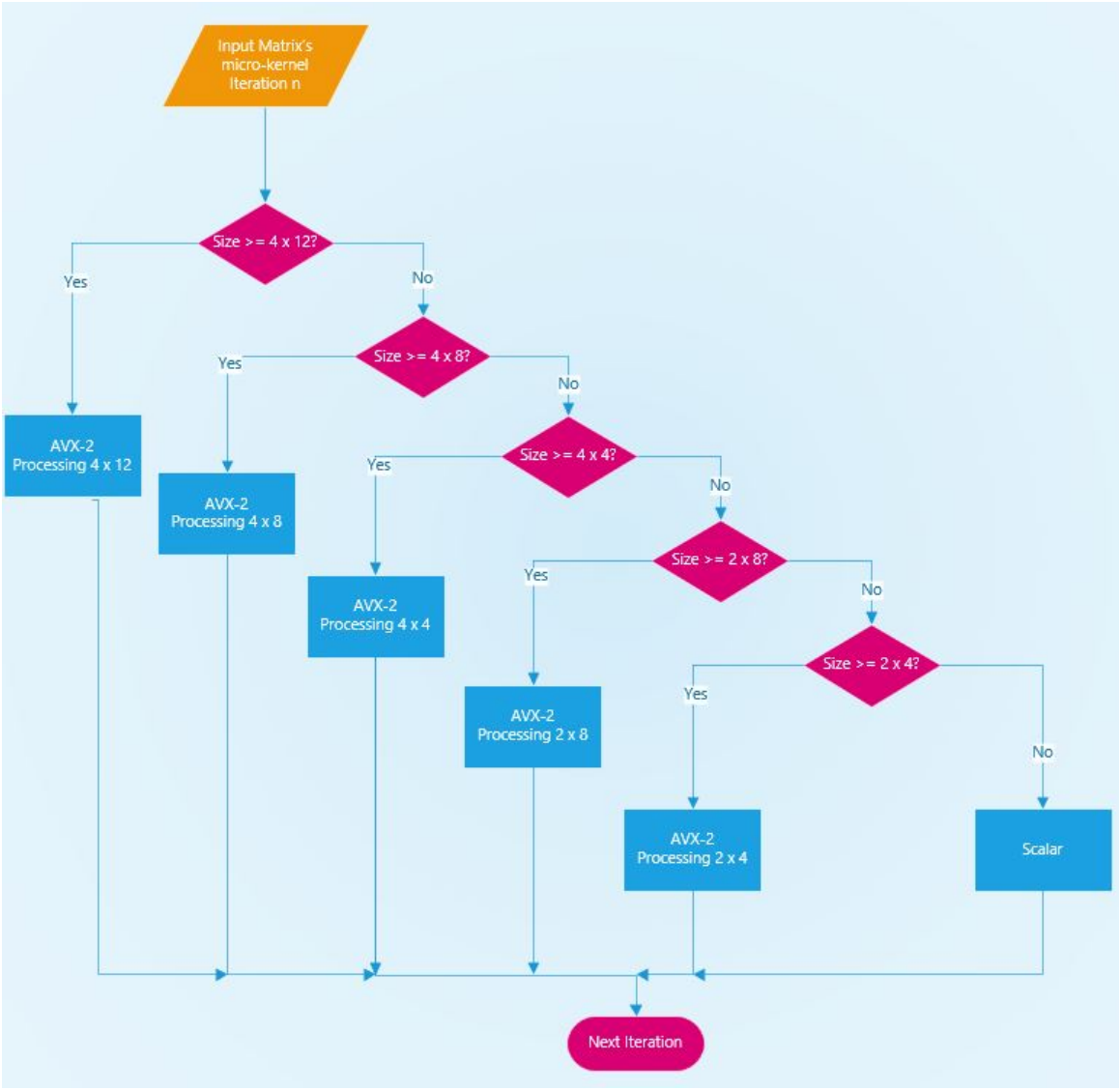

Figure 4: **SIMD flowchart**

Algorithm 1 Matrix Multiplication Pseudocode

```
function PACKA(m, k, A, lda)
    packedA ← ALLOCATE_MEMORY()
    for j = 0 to k do
        addr_first_col ← A + j
        for i = 0 to m do
            addr_req_elem ← addr_first_col + (lda * i)
            APPEND(packedA, A[addr_req_elem])
            i ← i + 1
        end for
        j ← j + 1
    end for
    return packedA
end function
function MICROKERNEL(k, m, n, A, B, c, ldc)
    for l = 0 to k do
        for i = 0 to m do
            for j = 0 to n do
                SIMD_KERNEL()
                j ← j + 12
            end for
            i ← i + 4
        end for
        l ← l + 64
    end for
end function
function MACROKERNEL(m, n, k, packedA, packedB, C, ldc)
    for i = 0 to m do
        for j = 0 to n do
            MICROKERNEL(Mr, Nr, &packedA[i * k], &packedB[j * k], &C[i * ldc + j], ldc)
            j ← j + Nr
        end for
        i ← i + Mr
    end for
end function
function DGEMM(m, n, k, A, lda, B, ldb, C, ldc)
    for i = 0 to m do
        for p = 0 to k do
            packedA ← PACKA(A)
            for j = 0 to Nc do
                packedB ← PACKB(B)
                MACROKERNEL(Mc, Nc, Kc, packedA, packedB, &C[i * ldc + j], ldc)
            end for
            p ← i + Kc
        end for
        i ← i + Mc
    end for
end function
```

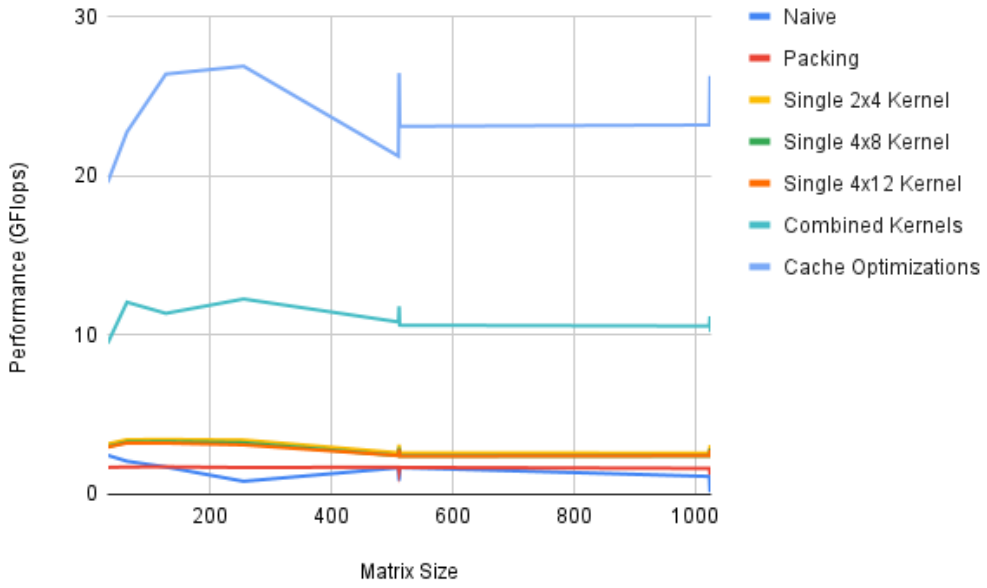

Figure 5: **Performance from Various Optimizations**

tion. It packs data into buffers such that cache hits are maximized to improve the performance as mentioned in Question 2(a). Doing this required modifications to the micro-kernel logic in "bl_dgemm_ukr.c" as we had to reorder our loops and the indices, and completing "packA_mcxkc_d" and "packB_kcxnc_d" functions in the "my_dgemm.c".

On not getting any significant performance improvement, the next thing we implemented was the AVX-2 SIMD micro-kernel for size 2x4 and using the normal scalar multiplications for all sub-blocks that did not fit in 2x4. After getting almost non-existent performance improvement, we implemented the AVX micro-kernels with varying sizes, like 2x8, 4x4, 4x8, and 4x12. When even that did not help us reach the target of 20 GFLOP/sec, we tried combining all these AVX-2 kernel processing sizes together, where the micro-kernel starts testing the sub-block size of the input with the available kernel sizes starting from largest to smallest before sending it to the scalar unit, as shown in Figure 4.

In this step, the main challenge was to implementing 4x12 kernel with only 16 AVX registers. It was done by reusing the available registers to store A. The improved performance showed that the cost of multiple loading in the AVX was amortized by the huge compute benefit that was received. Apart from using the kernels mentioned above, we even tried a few other kernel shapes like, 7x4, 2x16, and 2x24. These shapes did not give the best performance benefit when used in conjunction with other smaller shaped kernels, due to different shapes with respect to the already present. Eg- When 7x4 is too big a kernel, the rest of the small kernel shapes present cannot be used for the remaining dimensions. This causes a lot more use of the scalar multiplications. That is why, in the final code, we did not use these kernel shapes with the rest. Using them does not cause any performance degradation as these shapes are never even used if they don't have to highest priority. But on giving them the highest priority, we see a performance degradation due to the reason given above.

Even this did not improve the performance enough to reach the target of 20 GFLOP/sec. Next, we tried improving the micro-kernel sizes with Mr, Nr, and Kc so that most (all) of the data of the micro-kernel could be accessed from the L1 cache or the CPU registers. The method of determining the final sizes of the micro-kernel is given in the answer of Question 2(d). The performance achieved by each of these optimization is given in Figure 5.

2.3 High-level irregularities in data

1. We observe sharp performance dips for matrix sizes that are not powers of two
2. The ATLAS library scales slightly better than our algorithm for large matrix sizes due to the fact that we have only used multiple AVX kernel sizes, but didn't use padding and masking. These optimizations could scale better than our method. Both perform almost the same at smaller sizes as there won't be much difference among various implementations that could become visible at low matrix sizes.
3. As the graph in Figure 1 was generated using the genDATA script that does not run many matrix sizes, and jump from 1025 to 2048. This cause a perceptible increase or decrease in the graphs even though it is only a single point of evaluation. That could be due to varying loads of the server. Thus, the change from 1k to 2k can be conveniently ignored for all practical purposes due to lack of more data. Instead, it should only be analyzed till 1k to understand the functioning of different implementations.

2.4 Supporting Data

The biggest gain in our performance came by using the proper values for Mr, Nr, and Kc. This was enough to boost our performance by around 10 GFLOP/sec as can be seen in Figure 5. We got the appropriate numbers by hand-based calculations, parametric sweep, and some manual tuning [5].

We started the parametric sweep by getting the L1, L2, and L3 size of t2.micro on the aws. The size of L1, L2, and L3 were 32KB, 256KB, and 30MB respectively. After getting these numbers, it was trivial to know the size of Nc and Mc once we have the value of Kc. SInce we want Nc x Kc to completely fit in the L2 cache, so we put its mathematical expression in the script. Similarly, we wanted Mc x Kc to completely occupy the L3 cache, hence its expression was also computed. The main challenge was to get the optimal values of Mr, Nr, and Kc from the size of L1 cache as we wanted the whole micro-kernel data to completely reside there. For that we naively by incremented the values of Mr and Nr from 2 to 128 and Kc from 32 to 1024, in the increments of the powers of 2. From the results, we found that most of the combinations were useless, but there was a range of values where the performance was significantly better.

Analyzing that parametric data, applying some intuition, and performing some manual tuning, we decided the micro-kernel values of Mr, Nr, and Kc should be a such that the total space occupied by Mr x Kc, Kc x Nr, and Mr x Nr, should be a little larger than the size of L1 cache for optimal performance. It can be explained by understanding the whole outer product micro-kernel operation. In the Outer-Product of AxB, the each value of A is taken and multiplied by the complete row of B. Thus, even if the size of micro-kernel is larger than the L1 cache size, the overhead of fetching a little extra data can be overlapped by the computation.

After getting an acceptable value of all the parameters, we tried to get an optimum value of these parameters with another round of parametric sweep for fine tuning. But unfortunately, the results were not as expected because the performance of all the runs was almost the same around 15 GFLOP/sec, as shown in Figure 6. Apart from that, we also ran "Cachegrind" on our values to calculate the number of hits and misses, which is shown in Figure 7. Since it takes a lot of time, so a proper search of optimal parameters using it could not be done.

2.5 Future Work

There are a couple of things that we could not try given the limited time we had for this assignment. We believe that implementing these ideas could have improved the performance further.

1. Using Padding for the cases where the matrix dimension was such that none of the AVX's matrix processing unit used by us could be used. Right now, most of these cases go to scalar matrix multiplication. Ideally we would have wanted to compare the performance with padding to our present design, so as to come up with an optimal strategy of padding and using multiple dimension processing units. This could have taken us near the BLAS performance.
2. As mentioned in 2 (d), the cache size analysis was not successful. If we had more time, we could have tried the design space exploration again with our script, or with the "cachegrind". This would have improved our performance by some extent.

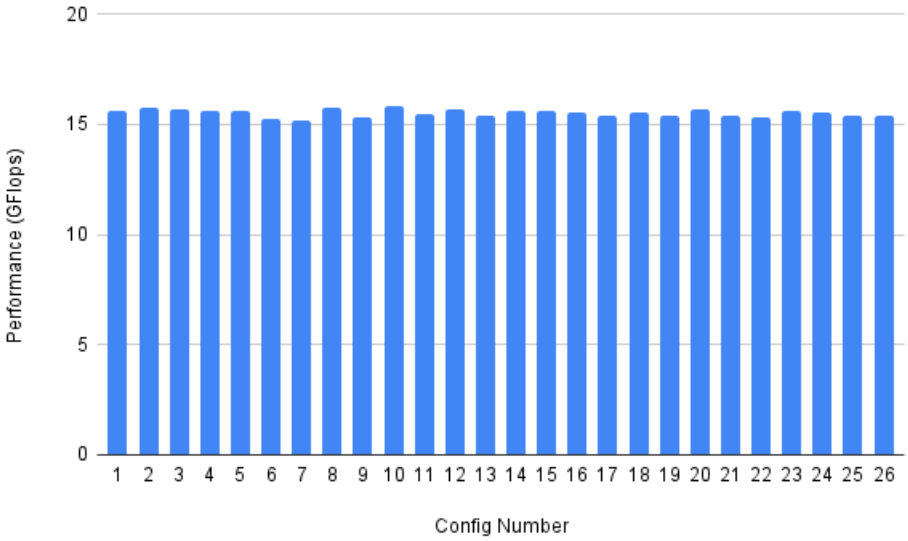

Figure 6: **Fine Tuning Runs**

```
==895== I refs: 76,266,851,516
==895== I1 misses: 35,501
==895== LLi misses: 2,252
==895== I1 miss rate: 0.00%
==895== LLi miss rate: 0.00%
==895==
==895== D refs: 24,992,494,872 (23,026,343,655 rd + 1,966,151,217 wr)
==895== D1 misses: 2,406,782,853 ( 2,339,540,343 rd + 67,242,510 wr)
==895== LLd misses: 501,881 ( 15,574 rd + 486,307 wr)
==895== D1 miss rate: 9.6% ( 10.2% + 3.4% )
==895== LLd miss rate: 0.0% ( 0.0% + 0.0% )
==895==
==895== LL refs: 2,406,818,354 ( 2,339,575,844 rd + 67,242,510 wr)
==895== LL misses: 504,133 ( 17,826 rd + 486,307 wr)
==895== LL miss rate: 0.0% ( 0.0% + 0.0% )
```

Figure 7: **Cachegrind Results**

3. We also could have implemented the "Butterfly" style of data processing in SIMD, instead of "Broadcast". This could have shed some light as to which of them performs better in which scenarios and a trade-off could have been studied.

3 References

References

[1] B. Chin, "Lecture notes in parallel computation - cse 260," April 2022.

[2] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, may 2008.

[3] J. Huang and R. A. van de Geijn, "BLISlab: A sandbox for optimizing GEMM," FLAME Working Note #80, TR-16-13, The University of Texas at Austin, Department of Computer Science, 2016.

[4] "Intel intrinsics guide." https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[5] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Trans. Math. Softw.*, vol. 43, aug 2016.